SoundByte: An iOS Application to Enhance Music Discovery

Jeff Cohen

March 17, 2016

Abstract

This thesis examines the process from conceiving an idea for a mobile application to building an iOS application that consumers will theoretically use, to App Store inception. The mobile application, SoundByte, is positioned to serve as a solution to optimize music discovery efficiency. The application solves an issue that consumers face on a daily basis: Not having a proficient way to find new music that they enjoy.

The application features a simple interface, and revolves around creating a peer-to-peer social network that relies on users sharing 30-second song clips. The application produces effective deliverables by only using 30-second song clips – a time frame that is long enough to know whether or not the user likes a song. The intent of incorporating a peer-to-peer social network is hinged on the belief that friends know each other's music preferences better than algorithms and websites. Lastly, the application is intended to be easy to use, simple and lightweight which is often overlooked in music discovery applications.

The outline for this thesis is a derivative of the entrepreneurial process from Diana Kander's book, *All in Startup* [6], to fabricate a product. In her book, she breaks down the entrepreneurial process as: coming up with an idea, simultaneously interviewing customers to test your idea, refining your idea, building your idea based off customer responses, and branding your idea. The purpose of her chronological process is to methodically and iteratively create a product that consumers want, without wasting time and money. This thesis delves into and documents the details of my experiences following Kander's first three steps: developing the idea for SoundByte, testing my hypothesis with potential customers, and building SoundByte as an iOS application.

Contents

1 Introduction

The substantial growth in the digital music industry over recent years has been nothing short of explosive. In 2013, various digital channels represented 39% of the global music industry revenue ($6 billion), with numbers growing ever since then [2]. One of the main reasons for this steep growth is the inception of "music streaming". The idea of music streaming was popularized in 2006 by Spotify, and quickly changed the way users consumed music. Instead of the traditional ownership model of music, people can now pay for a subscription from a company like Spotify and can have instant access to their music database with millions of songs. Today, Spotify has 20 million users, which is the most paying subscribers worldwide for a music streaming service [4]. Inspired by Spotify, the lucrative market for music streaming applications and services has captivated numerous entrepreneurs and developers who want to obtain a portion of the industry with their own innovations.

Subscription-based services have drastically changed the way people listen to music, as well as the way people search for and experience music. Music streaming allows users to gain access to massive content libraries and no longer have to pay for music on a song-by-song basis. Therefore, there has been a massive shift in the way companies design their business model. Due to the dominance in the industry by titans like Spotify and Apple Music, companies are less interested in competing in that space and are more interested in creating applications that incorporate the large music libraries that companies like Spotify and Apple Music have. In particular, software companies are putting in extensive work trying to create differentiating factors such as creating superior methods for how to listen to and discover music.

With the evolution of music streaming services, music discovery / recommendation applications like Pandora (Statista.com reports had $920,800,000 in revenue in 2014) [3], have become increasingly popular as users look for more efficient means of sifting through exceptionally large music libraries. In fact, it has become such a paradigm since the inception of music streaming that the term "satisficing" was introduced to represent the concept that over time, users become satisfied with what they are listening to because they do not know how to effectively discover music that they would like better [7]. The fact that this has become such a common scenario in music discovery alludes to the notion that there are opportunities to develop inventive solutions to optimize the current inefficient practices for music discovery.

From a personal standpoint, most of the music discovery applications that I have used, including Spotify, YouTube, SoundCloud, bolster impressive features such as predictive algorithms, or innovative ways to connect users to artists they may like, but there is not a single service that stands out. I find that I either become 'satisficed' with what I am listening to, or I end up wasting significant amounts of time yielding minimal results. Due to these inefficiencies in the music industry as well as my personal frustrations, I have dedicated my thesis to delivering a solution for avid music listeners to use that will help them increase the amount of new music they discover in less time than they would traditionally need.

The purpose of my thesis is two-fold. The first is to report the experiences I have had in applying Diana Kander's startup model[6] to SoundByte, in which she discusses the necessary steps to turn an idea into a business. The second is to produce an iOS mobile application, SoundByte, that utilizes a peer-to-peer social network to help users discover new music. To compliment Diana Kander's model, I will be using exercises from Wendy Kennedy's book, *So What? Who Cares? Why You?*[1] to further develop my ideas for SoundByte.

The thesis is structured as follows. Chapter 2 discusses the initial concept for SoundByte. Chapter 3 focuses on how to properly conduct various types of customer interviews. Chapter 4 provides an analytic report from my customer interviews and turns those responses into feedback that can be used to further SoundByte. Chapter 5 gives a technical discussion of SoundByte's development. Chapter 6 concludes with a discussion for future work in light of SoundByte's prototype launch, and provides a business model canvas based off the feedback SoundByte has received.

Chapter 2 outlines a value proposition for why there is an opportunity to create a mobile application like SoundByte. It refines SoundByte from an abstract concept into a much more concretely devised concept. Essentially, it breaks down the importance of three critical ideas that are crucial to SoundByte. Those three ideas are; creating a peer-to-peer network, creating a more efficient means for music discovery by only allowing users to share segments of a song instead of the whole song, and by creating a lightweight and easy-to-use interface.

Chapter 3 delves into methodologies and purposes for conducting focus group interviews as well as individual interviews. It explains various forms of good and bad practices. It also discusses how to ade-

quately target customer segments to ensure that the right people are being interviewed.

Chapter 4 discusses and analyzes interview results from the first, second, and third rounds of interviews and utilizes the material discussed in Chapter 3 to do so. The first round of interviews is mostly behavioral and pertains to the interviewees' music discovery behaviors, tendencies, and opinions. The second round of interviews involves a prototype of SoundByte being used by a group of interviewees in a closed environment. I allow the interviewees to interact with the application for 5-10 minutes and then ask them a series of questions related to their interaction with it. The third round of interviews consists of me giving interviewees the SoundByte prototype on their phone. They were able to use the application in a live environment where they could interact with other users. After those three days, I spoke with the interviewees about their experiences with the application.

Chapter 5 is a technical discussion of how SoundByte is built as an iOS application. More specifically, it discusses Apple's software development tools like Xcode and Swift, the model-view-controller framework that SoundByte uses for its design pattern, as well as the various APIs SoundByte uses.

Chapter 6 concludes the thesis by discussing future work that can be done to SoundByte. The third round of interviews involves distributing SoundByte in a live environment to a handful of people in which they can use it to interact with one another within their existing social network. Chapter 6 also showcases and discusses SoundByte's business model canvas.

2 Initial Concept

2.1 Introduction

In this section I discuss my proposition for creating SoundByte in light of how it fits in a marketplace. SoundByte's purpose is to serve as a quicker, faster, and more accurate means for users to discover music from their peers. The idea behind SoundByte came out of frustration from repeated unsuccessful attempts at discovering new music that I liked. Personally, I am an avid music listener – I am constantly seeking new music to listen to on a daily basis. I get musical inputs from a variety of different sources including prominent streaming services such as, SoundCloud, Spotify, and YouTube, websites such as, GoodMusicAllDay.com and CamelBakMusic.com, as well as mobile applications such as, Q.us and Mus.x. However, I find that I often waste a significant amount of time sifting through massive amounts of content – some of which I like, but most of which I do not. The problem that I have found with these services is that they are generally geared towards satisfying the majority of a respective user base, and do not focus enough on individual users. I believe that creating my own mobile application would solve this problem by generating a faster deliverable through a strong peer-to-peer social network. The goal of the application is to give users a more efficient music discovery experience.

The reasons listed in Section II outline why I believe that creating an application like SoundByte can effectively serve as an alternative to traditional music discovery practices. The remainder of the chapter is organized as follows. Section 2.2 introduces SoundByte's value proposition and uses exercises from Wendy Kennedy's *So what? Who cares? Why you?* [1] book to help illustrate it. Section 2.3 discusses SoundByte's key attributes in light of its value proposition. Section 2.4 concludes.

2.2 Value Proposition

SoundByte's primary purpose is to serve in the marketplace as a music discovery mobile application that incorporates three cornerstone principles. The first principle, is that people only really need to hear 30 seconds of a song to know whether or not they like it. The second principle is that friends know each other's music tastes better than a computer algorithm. The final principle is that music discovery should

be simple and easy. In my opinion, these apps create more confusion than help, which is why I have never consistently used a mobile application for music discovery. Given these principles, I have determined SoundByte's value proposition to be: "SoundByte provides the simplest and fastest way to discover music from the people that know you best."

2.3 SoundByte's Key Attributes

2.3.1 Time Efficient

One of the core purposes of SoundByte is to help users discover new music in a fraction of the time that they normally spend. One of the ways I think SoundByte can save time came from my personal contention that listening to the majority, or the entirety of a song, is wasteful. Over time, I have realized that I usually can tell whether or not I like a song by listening to a 30-second sample. I want SoundByte to incorporate this belief by prohibiting users from sharing an audio clip that exceeds 30 seconds. The idea behind creating short playback limitations came from my analysis of successful apps such as Snapchat, and in particular, its "My Story" feature. Snapchat's My Story feature allows users to capture and post videos or pictures to their My Story (a queued picture/video slideshow), with the intention of having it seen by their followers - people they have granted viewing rights to. Snapchat enforces a quick transfer of information by requiring pictures and videos to display for between 1-10 seconds. Users can post multiple pictures and videos to their My Story and all of their viewers can see these My Stories as many times as they want for a full 24 hours. Once a video or picture has been posted for 24 hours, it automatically deletes. I incorporated both of those important Snapchat features into SoundByte by confining users to only using 30-second audio clips, and by removing songs from a user's playlists if they are there for 72 hours.

2.3.2 Peer-To-Peer Transactions

Another cornerstone that I consider essential to SoundByte is the belief that peers know each other's music tastes better than the computer algorithms typically employed by existing services. This conjecture is supported by a statistic from a 2012 Nielson study that claims "54% of people are more likely to make a purchase based off a positive recommendation from a friend" [7], while only "25% of people are more

likely to make a purchase based off a music blog/chat rooms" [7]. SoundByte solely consists of to have only peer-to-peer relationships. I firmly believe that my peers have a better understanding of my music preferences, and SoundByte is meant to reflect that. Therefore, SoundByte is designed for users to rely on their connections, and theoretically, this could maximize new music discovery.

2.3.3 Lightweight And Simple Interface

A major flaw I find with many music applications is that they attempt to incorporate numerous services and do not have a singular purpose. I have used mobile applications that intend to be socially oriented through their incorporation of peer-to-peer relationships, but also attempt to serve as both a music streaming and a music discovery service. Consequently, these apps are heavyweight and difficult to navigate. Users looking for a simple way to discover music often dismiss these applications on the basis of difficulty. SoundByte alleviates this pain for consumers by offering a simple interface similar to Tinder's, but is solely focused on music discovery. Tinder is an online dating application, but contrary to traditional services like Match.com, Tinder creates a condensed profile of a user by only using the essential pieces of a user's dating profile such as pictures, hobbies, mutual Facebook friends, and a small bio. A Tinder user's profile is small enough so that you do not have to scroll on your phone to see the entirety of it, making it drastically less information dense than the profiles for other online dating services. Additionally, Tinder incorporates a very simple method for expressing whether or not you are interested in another user – swipe left with your thumb if you are not interested, and swipe right if you are interested. In addition to having a simple, concise, and easy-to-use interface like Tinder, I envision SoundByte to implement their swiping feature for when users like or dislike a song.

2.4 Conclusion

The current solutions for helping users discover music are unproductive and waste a significant amount of time because users are required to sift through numerous songs that they are presented by either inefficient algorithms, or by music blogs that are meant to please the masses. People using these services are provided with a lot of time-consuming suggestions, much of which are not converted into their music li-

braries. Additionally, countless music discovery applications try to incorporate both music streaming and music discovery services, which I contend to be counterintuitive. Music streaming revolves around the idea that a user wants to listen to the entirety of a song. SoundByte on the other hand embodies the the that users only need to listen to a 30-second song sample to know whether or not they like that song.

3 Customer Interview Methodology

3.1 Introduction

This chapter discusses the value in doing customer interviews as well as how to conduct them properly. Customer interviews are an excellent way to find out more about your potential customers as well as get feedback for how to better your product. A common mistake entrepreneurs make is assuming their idea is perfectly viable as is, simply because it solves a problem or presents some sort of gain in their own personal life. Learning about one's potential customers and market is a critically important step in developing a product tailored towards the correct customer segments. Conducting interviews helps an entrepreneur uncover the key ingredients in devising a product that is valued by targeted consumers. On a superficial level, conducting customer interviews may seem like an easy task, but there are countless caveats that can be overlooked. If an interviewer exhibits these bad practices during the interview process, or neglects to do customer interviews at all, they can fall victim to incorrect feedback and information.

The remainder of this chapter is organized as follows. Section 3.2 outlines the various interview types. Section 3.3 and Section 3.4 discusses the common pitfalls of customer interviews as well as best practices and methodologies, respectively, which are influenced by Diana Kander's *All In Startup* [6]. Section 3.5 concludes.

3.2 Interview Styles

There are various styles of interviews that an entrepreneur can conduct to elicit responses from potential customers. The two most popular are individual interviews and focus groups. Individual interviews, also know as personal interviews, involve an interviewer and an interviewee. They revolve around the interviewer asking the interviewee various questions in order to understand that individuals behaviors and attributes. Focus groups are slightly more complex and involve a group of around 5-8 participants, usually from similar backgrounds.

3.2.1 Individual Interviews

There are two types of personal interviews: unstructured informal interviews and structure standardized interviews. Unstructured informal interviews are "normally conducted as a preliminary step in the research process to generate ideas/hypotheses about the subject being investigated" [5]. The purpose of this style of interview is to learn more about how people think and react to certain issues in order to obtain a plethora of feedback. Usually these types of interviews are open-ended and are used for finding customer markets, feedback on products, etc.

Structured standardized interviews are much less open-ended and follow a specific questionnaire to obtain quantitative information. They are used to gather data on a variety of different people and are generally not concerned with open-ended responses. They are usually quantitative because these surveys are primarily used to gain large amounts of data that can be used for a statistical analysis.

3.2.2 Focus Groups

Focus groups are a research tool intended to yield qualitative information about its participants. A moderator directs the group and conversation towards the focus of the researcher. Focus groups are usually comprised of 6-8 participants and the discussions last between 1 and 2 hours. The moderator has a guide list of topics and encourages an informal discussion among the participants in a relaxed environment. The researcher records comments made by the participants and uses that information to help further the researcher's respective product.

3.3 Customer Interview Pitfalls

There are countless errors that could be made by an entrepreneur during the customer interview process. Entrepreneurs commonly overlook one of the most fundamental principles of the customer interview process by creating an environment that causes an interviewee to not give their honest opinion. Entrepreneurs can unknowingly and unintentionally word questions inappropriately that consequently yield dishonest responses. Additionally, entrepreneurs can very easily overlook the fact that interviewees, especially ones that they are friendly with, have intentions of being nice and not hurting their feelings. To avoid hurting

one's feelings, an interviewee will lie and say something to appease the interviewer. Consequently, this leaves the interviewer with fabricated information that can lead them astray in their pursuit of creating an ideal product. Another way an entrepreneur can yield invalid information is by not conducting their interviews in person. If an interviewee feels uncomfortable speaking to an interviewer (presumably someone they do not know very well) over the phone, the interviewer will most likely receive information that is not as thorough or genuine as if it was in-person.

3.3.1 Confirmation Bias

The ways in which an interviewer articulates their questions is an extremely important facet of the interview process especially when the interviewer and interviewee have a pre-existing relationship. In an attempt to find out more information from a customer in regards to their thoughts on an entrepreneur's product, an entrepreneur will often subconsciously ask questions in a leading manner. For instance, asking the question, "Don't you think we need a solution to test water contamination?" is extremely leading because it offers insight as to what the interviewer may think. Leading questions psychologically alter an interviewee's responses because they want to be agreeable with the interviewer. Consequently, an interviewer is left with confirmation bias by an interviewee that may not truly believe their own responses.

Confirmation bias also arises when an interviewer asks hypothetical question that often contain the clause, "would you". Generally, an interviewee will realize a specific response that an interviewer is trying to conjure, and since hypothetical questions have no serious implications, an interviewee will respond accordingly. For instance, asking the question, "would you agree that nobody wants this application because it has zero downloads on the App Store?" inherently puts a belief in a respondent's answer. For an entrepreneur, these types of questions provoke confirmation bias in the form of getting responses that are not genuine.

3.3.2 Not Conducting Interviews In-Person

In today's era, it is far easier, and less time consuming to administer customer interviews over the phone, or by having them respond non-verbally. However, these methods are not effective in evoking

optimal responses. According to Diana Kander, "the majority of an individual's communication is non-verbal" [6]. People provide subconscious cues and tips using body language. Having an interview over the phone or non-verbally eliminates the opportunity to pick up on certain indicators as to which topics should be further probed and explored. Additionally, an interviewee that has not met the interviewer will probably feel less comfortable providing insights and opinions over the phone because there is a lack of trust and personal connection. Conducting interviews in person helps the interviewee and interviewer establish a verbal and visual connection that inherently makes the interviewee more trusting.

3.4 Best Customer Interview Practices

An ideal customer interview is one that elicits the most genuine, thoughtful, and developed responses possible. In order to do this, an interviewer must be able to create an environment that makes an interviewee feel comfortable, in which they feel confident sharing their honest opinions. Creating a comfortable environment is predicated by how well an interviewer can ask open-ended questions with appropriately timed follow-up questions, find the right customer market to interview, and separate the Problem and Solution interviews. Without the aforementioned conditions, an entrepreneur will most likely be led in the wrong direction pursuing dishonest feedback. Therefore, it is vital to create an environment where all feedback is constructive for a respective entrepreneur.

3.4.1 Question Format

An entrepreneur should ask their interviewees open-ended questions in order to facilitate a conversational interview. Asking direct questions, especially yes or no questions, will often create an interview that lacks significant content. On the other hand, open-ended questions garner more useful responses in the form of "the stories [that] subjects are sharing that will convey useful information they might not even understand is important" [6]. For instance, data on their backgrounds, habits, personal beliefs, etc. in the context of an interview question is information that is not typically discovered through the interview process. This type of information can be crucial in learning more about a product's customer segments and can shed light as to why certain markets would find a respective product useful. Therefore, interviewers

should allow interviewees to dominate the conversation in order better understand how a given interviewee operates. Additionally, having an open-ended dialogue will make a customer feel more comfortable, and will be more honest in giving feedback.

A crucial component to having open-ended interview questions is having appropriately timed follow-up questions. If a certain response is not clear, or if an interviewer feels there is a driving force behind a particular response, they should dig deeper. An interviewees "true feelings aren't going to come out through the direct answers to the interviewer's questions, but in the stories the interviewees tell around their answers" [6]. An interviewer should motivate an interviewee to further develop their thoughts in order to gain valuable information and opinions from that user.

3.4.2 Targeting Customer Segments

It is imperative for an entrepreneur to realize that it is extremely rare for a product to be universally valuable. While some people may find it impossible to live without an entrepreneur's product, others may find it utterly useless. Interviewing those that find the product to have no utility is a complete waste of time, as an entrepreneur does not yield any useful information. Instead, an entrepreneur should interview the people that value the product the most. Honing in on these customers and interviewing them extensively will help an entrepreneur formulate a hypothesis about what these customers have in common. From there, an entrepreneur can devise a customer segment – a set of attributes and behaviors to describe that customer base.

3.5 Conclusion

Conducting customer interviews is a crucial component in the product refinement process for an entrepreneur. Customer interviews, whether they are personal or focus group oriented give an entrepreneur key qualitative and quantitative information. Without consistent customer feedback, an entrepreneur cannot verify or reject any parts of their hypothesis. Additionally, customer interviews or focus groups can provide insights that were not originally conceived by the entrepreneur. When conducting customer interviews, it is important to conduct them properly. There are numerous good and bad practices, and the use-

fulness of the information gathered from the customer interview is contingent on how well an entrepreneur avoids bad practices and implements good practices. More specifically, avoiding confirmation bias, and conducting interviews in person are ways ensure that an interviewee's responses are genuine. Additionally, by asking open-ended and encouraging detailed responses as well as making sure the customer being interviewed is in a product's hypothesized customer segment, an entrepreneur can increase their chances of eliciting useful information.

4 SoundByte Meets The Customer

4.1 Introduction

Chapter Four provides an analytical discussion about the results of the first, second, and third rounds of customer interviews. The first series of customer interviews were purely conceptual – the questions pertained strictly to the viability of the value proposition. More specifically, the questions asked interviewees about how they listen to music, discover music, and their other music listening tendencies. The second round of interviews involved vetting a prototype to interviewees in the form of allowing them to use the application on my iPhone 5s in an environment with a seeded database (a database that had data in it from other interviewees) for 5-10 minutes. During this round, SoundByte had all of the functionalities that corresponded to my original value proposition. SoundByte users could add their peers (users who have already created accounts), search for music using Spotify's music streaming service, listen to the 30-second song clips that their peers had added, 'favorite' the songs they liked, and could see those songs in a tab section on the application. The layout of the app was in a tab-bar format, and there were 5 tabs: Timeline, Friend Search, Song Search, Settings, and Favorites. These features as well as the layout are important to SoundByte because they embody my hypothesis that the most efficient and effective way to share music is through using 30-second song clips with peers in a lightweight and easy-to-use application. The third round of interviews was basically the same as the second round, but I installed the application on the interviewees' phones and let them use it for three days in a row.

The purpose of this chapter was to test my hypothesis by showcasing my application to various potential users and conducting in-person interviews using the methodology from Chapter Three. The sole criterion for my selection of interviewees is that they listen to music on a daily basis. Additionally, I opened this opportunity to groups of people that I often discuss music with. The reason behind this is that I tried to pick interviewees that knew one another so that I could have more insights as to how they interact with the people they often share music with.

The remainder of this chapter is organized as follows. Section 4.2 outlines the way I conducted the first and second round of interviews. Section 4.3 discusses the information I received from the first round

of interviews and compares it to my value proposition. Section 4.4 and 4.5 discuss the feedback from the second and third round of interviews, respectively. Section 4.6 concludes.

4.2 Interview Process

All of the interview rounds were conducted by me in a one-on-one manner. The first round of interviews were used to merely identify whether or not my value proposition exhibited opinions that were consistent with other people. The process for conducting this round of interviews was quite simple – I met interviewees at a location of their choice and asked them a series questions and wrote down their answers. The second round of interviews had more of a preparation process and was more interactive. Before I began conducting the interviews, I had all of the interviewees send me 4-5 songs that they liked. I seeded my database with those songs so that when someone came to be interviewed, I could simulate a real environment in which SoundByte would be used in by having authentic data. However, when an interviewee came to be interviewed, I would temporarily disable their account so that they could interact with the application by creating their own accounts and adding their own songs. The third round of interviews was done by simply downloading the prototype on interviewees' phones, and letting them use it in a live environment where they could interact with one another.

4.2.1 First Round Of Interviews

The first round of interviews involved asking a series of twelve questions to volunteers and recording their feedback. The questions that were asked were purely conceptual and attempted to evoke insight into the music behaviors of the interviewees. Listed below are the interview questions that were asked in the order they appear in.

1. Are you an avid music listener?
2. How do you listen to music?
3. How do you discover music?
4. How often do you and your friends share music?

5. What apps do you use to share music with peers? Why?
6. What are the benefits and negatives of those apps?
7. How much time do you spend on new music per week?
8. Do you find your time spent looking for music as well spent?
9. How long does it take when listening to a song to realize that you like or dislike it?
10. What do you think of your peer's music tastes?
11. What percentage of the time do you agree on a song being good?
12. Which one is better for reliability on finding music: music websites and music applications, or friends?

The purpose for ordering these questions as such was simply because they seemed appropriate to ask in that order. The first questions were rather simple, but the following questions helped delve into the answers they provided.

4.3 Second Round Of Interviews

I began each interview by giving the interviewee the opened application on my iPhone 5s (the application was not ready to be used on other phones yet). I did not give them any instructions on how to use the application, but told them to ask call me into the room if they needed assistance or had questions. Upon receiving my phone, the interviewee was prompted by the application to create an account. Once they had successfully created a username and password, they were brought to the Timeline, which is the home screen. This screen is where a given user's peers, or 'friends', are displayed. At this point, I would leave the room and allow them to use the application as they wished. I allowed the interviewees to interact with the application as long as they wanted and told them to call me back into the room when they were finished with their session. Once they called me back in, I would begin a dialogue and document their experience with the application and their overall music habits. Listed below are the interview questions – the order they appear in was generally the order in which they were asked.

1. What did you think of the application?
2. Was there anything confusing about the application?
3. Was it difficult to use?
4. What did you think of the peer-to-peer functionality?
5. What features would you change?
6. What would you add?
7. What would you delete?
8. Does this solve a problem in your life?
9. Would you use it in real life?
10. Can you sum what you think the purpose of this application is?
11. How often do you share your music with others? And how?

The aforementioned questions were asked in that order for a few reasons. Firstly, the earliest questions are meant to convey to the interviewee that the interview is more of a dialogue than an interview. This helps the interviewee relax and inherently creates an environment in which the interviewee feels more comfortable sharing their thoughts and opinions. Secondly, having the more open-ended questions presented first helps to create a flow of ideas that can make the transition to the more specific questions more fluid.

4.4 First Round Interview Results

The first round of interviews involved twelve volunteers (peers and friends of mine), both male and female, aged between 20-23, and who are all Union College students. Of those twelve students, eleven of them said they are avid music listeners. The information I gathered from the interviews provided insightful information in regards to SoundByte's viability as a consumer product. The paragraphs below discuss the findings that arose from the interviews in relation to my originally proposed value proposition discussed in Chapter Two.

4.4.1 Time Efficient

One of my core beliefs when I conceived the idea for SoundByte was that when people listen to a 30-second clip of a song, they know whether or not they like it and do not need to listen to the entirety of that respective song. In essentially every music discovery application, users are presented with full songs, which I found to be inefficient.

The results that I have gathered from the interviews reinforce the notion that users do not need the entirety of a song to know whether or not they like it. Seven of the interviewees said they believe they know whether or not they like a song within the first 30 seconds. Another three said they knew whether or not they liked a song between 30 and 60 seconds. One interviewee said they could tell in 90 seconds, another interviewee said they could only tell after listening to the full song.

In the case of creating a more efficient music discovery experience, the numbers seem to agree that there is value in reducing the length of a song. The fact that seven of the interviewees believe they only need 30 seconds or less to know whether or not they like a song in combination with another three interviewees claiming they can create the same effect using only 60 seconds of a song, shows that creating a more time efficient application has merit.

4.4.2 Peer-To-Peer Transactions

The idea to incorporate peer-to-peer transactions came from the notion that music websites and predictive algorithms are not as effective as peers in terms of delivering enjoyable music to individuals. It is my personal contention that my peers understand my music tastes better, and I can quickly identify individuals that my music tastes align with. Consequently, the chances that I like a song that I receive from that respective individual greatly outweighs the chance that I like a random song given to me by a predictive algorithm or music website.

Based off the results, it appears that the majority of the interviewees concur with my contention – peers are better predictors of song likeness than algorithms and websites. Nine of the respondents said they prefer music inputs from friends rather than from other sources. One respondent that did not agree said, “it depends on the source”, and another disagreeing respondent said they do not really have the same

music taste as their friends. When asked about the percentage of the time that the interviewee and their peers agree on a song likeness, one person said less than 60%, two said between 60-70%, three said between 70-80%, five said between 80-90%, and one said 90+%. All in all, I would say that the majority of people see peer-to-peer music sharing as a more effective means of sharing music than other sources.

4.4.3 Lightweight And Simple Interface

Another big issue I always have with music discovery applications is their inability to deliver a concise and intuitive user experience and user interface. In my opinion, they are difficult to navigate through and create a negative user experience. Additionally, I have found that the majority of music applications blur the lines between serving the purpose as a music discovery application and serving as a music streaming application. Consequently, these applications are very heavyweight and are unable to successfully incorporate both music streaming and music discovery using one easy-to-use interface.

In this round of interviews, I somewhat failed to ask enough questions about the various other music discovery services people use and the difficulty involved in navigating and using those other interfaces. I was hoping that question 5 (asking about the benefits and negatives of the apps people use) would evoke informative responses about user experience and user interface, but it did not do as much as I would have liked. However, three people did say they find SoundCloud's search features difficult to use in terms of finding songs and adding friends. Two others said they find SoundCloud's 'Stream' feature of their mobile application (the automatically generated music feed) not effective in suggesting music. Some other common comments I saw was that sharing music can be difficult if two people do not have accounts with the same services. For instance, interviewees said it was hard to share music with someone on Spotify if they do not have an account.

4.5 Second Round Interview Results

For the second round of interviews, I vetted the SoundByte prototype and interviewed twelve potential customers. The interviewees commented on SoundByte's current state, its features, its overall utility as a product, as well as pieces of advice to help further the product.

The main intention for designing the interview questions in an open-ended manner was to illicit suggestions, criticisms, and various forms of feedback that would not have ordinarily been brought up in an interview that had more quantitative questions. Fortunately, the interviews evoked a plethora of feedback that can be used to further SoundByte's utility and design features. I received qualitative and quantitative feedback on features I should change, features I should add, features I should delete, interviewees' music behaviors, and other information on their experiences with the application. The bullet points below explain the feedback I received and the consequent suggestions I am taking away.

- **Incorporate more music streaming services.** Eleven of the users said that SoundByte should have more music streaming services to search through when adding songs to their playlist. The suggestion to incorporate SoundCloud was mentioned by eight of those eleven users.

- **Peer-to-peer music sharing is efficient.** After using the app, ten (nine of those users being those who responded "definitely" when asked if they prefer their friend's music inputs rather than other music sources) of the users said they found the percentage of songs they discovered that they liked while using the app was higher than inputs from other music sources.

- **Users can tell whether or not they like songs after listening to only a fragment of a given song.** Without being prodded, 9 users said they liked the idea of using clips of a song.

- **The interface is simple to use.** When asked about user interfaces and their user experience, interviewees provided positive remarks. All twelve respondents found the interface easy to navigate through by the end of the session.

- **Make naming conventions and icons easier to understand.** Four of the respondents said that it took them some time to understand what each tab was responsible for. When prodded further, they all seemed to suggest that a lack of instruction and poor icons and naming conventions led to this with one interviewee saying, "it took me a little while to understand what the 'Timeline' view was for. It basically took me like 2 minutes to finally access the music". However, eight of the interviewees said it was easier to understand the navigation process than traditional applications that incorporate music discovery.

- **Add the ability for one to see the songs on their own playlist.** The majority of interviewees said they would like the ability to see the songs they have posted on their own playlist. As the app currently stands, users cannot see which songs are currently on their playlist.

- **Have an easier way to search for friends.** During the interview process, many users added all eleven other people as friends because they could not understand which usernames correlated with which other interviewees. This created some confusion and made it difficult for users to identify whose music they were listening to.

- **Have the ability to access music that is from over 72 hours ago.** During the interview process I told each interviewee that I planned on adding a feature that deletes a song from a user's playlist after 72 hours to ensure that playlists do not become too long and bogged down with old music. However, four interviewees said that there should be an archive like feature where users can access songs that are from more than 72 hours ago.

- **Give the user the ability to choose which 30 seconds of a song they can post.** Some users seemed to think that Spotify's built-in 30-second preview feature did not capture the best 30-seconds of a song. Additionally, two of the three that think 30 seconds is too short of a timeframe said they believe that songs, especially remix based songs, change too much over the course of the song. Additionally, two of the individuals that claim 30 seconds is too short made the comment that Spotify did not pick the best 30-second preview clips for their songs.

- **Add swiping gestures for when a user wants to exit or favorite a song.** Many of the users were unable to see the exit button (white button at the top-right of the screen) in the playlist view. They said a swipe feature to exit would be much more intuitive and to extend that functionality for when a user wants to favorite a song.

4.6 Third Round Interview Results

The third round of interviews did not provide enough relevant data to make any solid conclusions or inferences on how users interacted with the application in a live setting. Most of the interviewees did not

use the application at all. After speaking with the interviewees, I concluded that this unsuccessful round of interviews was due to a culmination of the following factors: I only gave the application out to ten people, it was the final week of classes (where people often have heavy work loads), and most importantly, the application was not complete enough to be distributed. The application was relatively buggy considering it would crash if users tried to navigate through the application too quickly and it also lacked aesthetic appeal. In today's era, people have a certain expectation when using mobile applications that they will work properly (except for very rare instances) and will provide a quality user interface and user experience. Both SoundByte's user interface and overall performance are two important factors that are not up to par with other applications featured in the App Store.

To some extent, the unsuccessful interview venture helped shed light on where SoundByte needs to be taken in the future if it wants to have potential succeeding in the App Store. First and foremost, SoundByte needs to improve its basic functionality - issues with threading need to be resolved to enhance the application's overall speed and stability. It cannot be crashing on users if they try to navigate too quickly. Secondly and almost as importantly, the user interface and overall user experience needs to be improved. The current layout of the application is essentially just a skeleton that contains all the functionalities of what I proposed in my Initial Concept Chapter. The minimal user interface is perfectly fine to use as a prototype to vet out to potential customers as a means of trying to establish a proof of concept, but needs severe improvements to become a commodity in the App Store.

4.7 Conclusion

In the context of SoundByte, the first round of interviews provides affirmation that my value proposition of creating a mobile application that features peers sharing 30-second song clips with one another in a lightweight environment is useful to various types of music listeners. From there, I began the construction of SoundByte to meet that criterion. After SoundByte had the core features and was functioning properly, I tested a prototype to customers. The second round of interviews was an opportunity to test SoundByte as a physical product to see if it served a purpose to potential customers and whether or not it functioned as a means of fulfilling my value proposition.

Overall, the feedback I received from the second round of interviews affirms the viability of the core foundation of my value proposition and also delivers insight about how to further SoundByte as a mobile application. After spending time using the application in the second round of interviews, ten interviewees said that the percentage of songs they liked from their friends on the app exceeded the percentage of songs they liked when delivered by other music sources. Lastly, eight of the second round interviewees said that they found the application easier to navigate through than traditional music discovery applications. The combination of the aforementioned responses by the interviewees seems to confirm both the viability of SoundByte in a market, as well as the execution of the application in meeting its fundamental principles.

The positive responses from the aforementioned interviews are verification that SoundByte is ready to be further developed. The feedback I received from the second round of interviews in Section 4.4 is primarily what I will use to guide the changes I will make for SoundByte's next iteration. Once I have incorporated all, or the majority of those changes, I will do another round of customer interviews, possibly for a more extended period of time. I will continue to do iterations until I receive feedback suggesting that SoundByte is ready to be exposed to the App Store.

5 SoundByte iOS Development

5.1 Introduction

The notion that there are very few resources that serve as effective means for music discovery is the fundamental principle behind the decision to build SoundByte. In my opinion, music discovery applications and websites do not provide music that I like as often as my friends do, are too complex and bloated, and require a lot of time, but yield minimal results. These ideas were affirmed when I conducted my first round of interviews in which I learned about my peer's music discovery tendencies, behaviors, and opinions (see Chapter 4). The confirmation that other people have the same issues with music discovery drove the desire to create SoundByte – an application for music discovery that relies on peer-to-peer sharing, using only 30-second song clips, while also having a lightweight and easy-to-use interface.

The sections below articulate all of the technical aspects that have gone into creating SoundByte. The remainder of this chapter is organized as follows. Section 5.2 outlines the early design choices I made in regards to making SoundByte an iOS application and developing it using the model-view-controller framework. Section 5.3, 5.4, and 5.5 discuss in detail the overall design and the various factors that contributed to SoundByte's model, view, and controller, respectively. Section 5.6 describes the most recent prototype for SoundByte's. Section 5.7 lists the various issues I had when developing SoundByte. Section 5.8 concludes.

5.2 Early Design Choices

After making the decision to develop SoundByte as an iOS application, I had to learn the entire process for creating iOS applications. Firstly, I needed to choose either Swift or Objective-C as my coding language. I decided to use Swift because it is newer, more dynamic, and more closely resembles the languages I have learned thus far in college (Python and Java) than Objective-C. I also needed to learn how to use Apple's integrated development environment (IDE), Xcode, to write and run the code in.

For a design pattern, I chose to use the model-view-controller (MVC) framework to facilitate the interactions between a user and my application. The reason for choosing to use the MVC framework as my design pattern was because of Xcode's intuitive Cocoa Touch interface that makes connecting actions between the

view and the controller very simple. The three components of a MVC, the Model, the View, and the Controller are all used when processing user interactions. The model contains data objects such as a username and a password. The view is rather self-explanatory – it is what the user sees and interacts with via the iOS touch interface. Lastly, the controller is responsible for updating information in the model when a user does an action in the view and update the view using information from the model.

The purpose of using the MVC framework is to have all three components function together whenever a user interacts with the application. For example, when a person creates an account, they are shown a screen (the view) where they are directed to put in their user credentials in the form of a username and password. After that person inserts their credentials, they click the "Register" button. Once clicked, the "Register" button triggers a function in the controller that runs through a series of validations (i.e. does anyone else have that username, and is the password between 5 and 20 characters) and if all the information is valid, the controller updates the model with that user's information. The model then stores that information accordingly in the database and that user's account is now recognized by SoundByte.

5.3 Model

For SoundByte, the model holds information about data and other attributes that pertain to the objects used within the application. For instance, in Soundbyte, a 'User' object has an objectId, a username, a password, among other pieces of data. In addition to a User object, SoundByte's model contains a 'Follow' object, a 'Like' object, and a 'Playlist' object.

In Soundbyte, a User object gets created and stored in the model when a user signs up an account. Once that user's information is stored in the database, they are assigned an objectId (a sequence of numbers and letters to identify an object), a username, a password, and other information that identifies when each user was created and when their information was last manipulated. Within the application, a user signs in with their username and password credentials, but when the controller queries an object in the code, it is done using that respective object's objectId. For instance, when a user adds a song to their playlist, a playlist object gets created with a pointer (a data attribute that signifies a relationship between two objects) to that respective user's objectId. By including that information, the database now knows which user added that

song.

Besides having different purposes, a Like object and a Playlist object are relatively similar. Both objects have a pointer attribute to a user's objectId to identify which respective object belongs to which respective user. In the database, a Like object is created when a user clicks on the heart icon in the playlist view to acknowledge that they want to add that song to their favorites section. Once instantiated, that respective Like object contains its own objectId, the objectId of the user that liked it, and the Spotify URI (Uniform Resource Indicator) – a distinct number and letter sequence that identifies a song in their database. A Playlist object is identical to a Like object, except the pointer attribute is the objectId of the user that chose to add that respective song from the song search feature.

Lastly, a Follow object is used to create a relationship among users when they add one another using the friend search feature. This object has two objectId pointers, a "fromUser" pointer, and a "toUser" pointer. To explain this better, imagine there are two users: User A, and User B. When User A chooses to add User B using the add friend feature, a Follow Object gets instantiated where User A's objectId gets stored as the "fromUser" pointer, and User B's objectId gets stored as the "toUser" pointer. Through this, the database now acknowledges the relationship between the two users and allows User A to access User B's playlist.

To keep track of the aforementioned objects, SoundByte uses a Backend-as-a-Service provider (BaaS), "Parse". Parse functions as a web-hosted database that operates synchronously with SoundByte whenever an object gets created or updated. For instance, when a user gets created, SoundByte receives the user's data and subsequently provides that information for Parse to store in the database. Parse is also responsible for all queries that get requested by the controller except the ones that pertain to Spotify.

For all music related functions throughout the application, SoundByte relies on Spotify's iOS software development kit (SDK) [8] and its web application program interface (API) [9]. An API is an interface from a service (like Spotify) that allows you to access certain functions and features from that service and an SDK is a collection of APIs and other development tools. SoundByte relies on the model to hold all of the Spotify IDs (an ID specific to each Spotify song) that are used throughout the application. The Spotify IDs are important because they identify a song and all of its pertinent information in Spotify's database. Using this ID, SoundByte is able to extract information about a song from Spotify such as the artist, the song name,

and the song's 30-second preview URL.

5.4 View

A fundamental principle behind SoundByte is having an intuitive and easy-to-use user interface. It is important to keep in mind that people typically use their smart-phones as a computer when they are on the go, and therefore its interface should be simpler than if it were an application for a desktop. In the model-view-controller framework, the view is responsible for presenting the user interface and is also responsible for holding view objects that users can interact with. One of the view's main functions is to communicate with the controller to display changes that occur in the model when a user interacts with the user interface. An example of this interaction is when a user adds a friend from the friend search feature. Once a user adds a friend and goes back to the Timeline view, the view will now display that user's new friend because the controller queried that information from the model.

5.5 Controller

In the MVC framework, the controller is unique in that it communicates directly with both the model and the view to complete an action, whereas the model and view only interact directly with the controller. An example of this interaction is when a user registers a new account. That user first types in their user information (username and password) and after clicking the "Register" button, the information gets passed along from the view to the controller. The controller receives that information, processes it to make sure it is valid (meets all username and password requirements), and then sends that information to the model to create that new user in the database.

Another important function of the controller is to handle the various APIs and SDKs that the application uses. For SoundByte, the controller utilizes Spotify's iOS SDK and their web API for all processes related to searching and listening to music. The reason why SoundByte uses both the iOS SDK and the web API is to maximize Spotify's utilities without having a SoundByte user log in to Spotify. For instance, when a user uses SoundByte's search feature to add a song to their respective playlist, the iOS SDK is used to store that song's Spotify ID sequence to the model. Then, when a different user tries to access that user's playlist,

the controller queries the database and puts the appropriate Spotify IDs through a function that uses the web API to process those Spotify IDs and returns the audio for the 30-second preview URL. This is a major design choice for SoundByte because by not logging into Spotify, users have very limited functionalities. For instance, SoundByte cannot incorporate a feature that allows users to access the entirety of a song. However, giving anyone the ability to use the application even if they do not have a Spotify account in addition to avoiding the annoyance of needing to log in to other applications is an important feature that is for the betterment of SoundByte.

The other SDK that SoundByte uses is Parse.com's iOS SDK, which handles SoundByte's model and database as discussed in Section 5.2. Even though all of Parse's services pertain to the model, all database query methods are done in the controller. The controller manipulates and updates the model using Parse's services. For instance, when a user registers an account, the controller uses Parse's SDK to process the user and then adds them to the database. In essence, Parse is responsible for hosting all backend services such as the database and model, but the controller is what makes query requests to Parse.

5.6 Current Stage of Development

The application begins by directing users to a login screen where they can register or login. After users create a SoundByte account and log in, they are directed to a new view called the 'Timeline' [B.A]. The Timeline gives SoundByte users the ability to share music with one another. Every user has their own Timeline, which contains a list of usernames corresponding to that respective user's 'friends' – people they chose to add in the application using the built-in "Friend Search" feature [B.B]. If a user clicks on one of their friend's names on the Timeline, it plays that respective friend's 'playlist' in a different screen. That different screen shows all of the information and cover art for each song that is currently playing in that playlist [B.F]. A user creates their playlist by using the "Song Search" feature, which allows them to search for music using Spotify's music library and gives them the ability to add Spotify's provided 30-second song previews to their own playlist [B.C]. Each user's Timeline can be accessed by all of their friends. If a user likes a song from a friend's playlist, that song's information (the title and artist), is automatically sent to the user that liked the song in his or her 'Favorites' section [B.E]. The 'Favorites' section contains the artist's

name and song name for all of the songs a user has ever liked. The purpose of this section is to remind a user of the songs they have listened to so that they can download or listen to them using whichever music listening service they choose.

5.7 Technical Challenges

Throughout the development process, I encountered numerous development challenges that affected the final prototype for SoundByte. One of the first challenges I encountered was learning how to incorporate Spotify's services. I began developing SoundByte to only use the iOS SDK, but I was constantly having issues with Spotify's user authentication process - users were having to sign in to their accounts whenever they wanted to search for or listen to music. To combat this issue, I looked into Spotify's web API and realized that I could bypass needing any sort of authentication by using the iOS SDK in combination with the web API. Using the combination, I could get Spotify IDs using functions in the iOS SDK and could get all of that song's information using the web API.

My inability to properly incorporate threading in a lot of places throughout the application causes SoundByte to function irregularly on occasion. In programming, threading is a technology that allows an application to execute multiple code paths concurrently. In SoundByte, there are often multiple threads running at once to query a single playlist and visually display its album artwork, song name, artist name, and album name, and if certain threads finish executing before others, the application crashes. For instance, when a user tries to listen to a friend's playlist, three threads run simultaneously - one queries the database to get the Spotify ID, one takes those Spotify IDs and asynchronously retrieves the song URL and other song information, and the last one updates the UI using the Spotify IDs. If the thread to update the UI is called before the thread to get the Spotify ID finishes its asynchronous query, the application will crash. If I had implemented threading properly, the threads for those queries would execute in proper order and would keep the application from crashing.

5.8 Conclusion

As it currently stands, SoundByte is an iOS application that has all of the functionalities that were initially conceived and then affirmed in the first and second rounds of interviews. It is a prototype, but has the ability to allow users to share music with one another using 30-second song clips using a simple interface. More specifically, it allows users to add one another as friends, search for music using Spotify's music streaming library, add those songs to their playlists for other users to listen to, and has a feature for users to acknowledge that they like a song and saves that song to their Favorites. For the most part, the application works properly, but it is susceptible to crashes if users try to do too many things at once.

To develop SoundByte further to the point where it is ready to be submitted to the App Store, it needs to have a much better UI and needs to function more fluidly without crashing. Currently, the UI is very basic and is not aesthetically pleasing. There is an expectation from consumers for an application to have an attractive visual representation, and SoundByte does not fulfill that expectation. From a functionality standpoint, SoundByte does not work well when it gets overloaded with tasks. If a user switches between multiple screens too quickly, the application will crash. It needs to have better threading and other ways to avoid rare conditions that cause the application to crash.

6 Conclusion

Currently, SoundByte is a functioning iOS application that contains all of the features that were mentioned in the Initial Concept Chapter (and were also confirmed by the first and second round of interviews), and can be used to help users discover music from their peers. The application allows users to add one another, search for songs within Spotify's music library with the ability to add 30-second song clips to their respective playlist, and save songs to their Favorites from other users. Through these features, SoundByte enhances music discovery processes by creating a peer-to-peer network that saves its users time by condensing full songs into 30 second clips. However, before SoundByte is ready to be exposed to the App Store, it needs to be able to handle multiple tasks at once without crashing and also needs a more aesthetically pleasing interface.

The rationale for wanting to create an application that enhanced music discovery practices came from my belief that there are no existing efficient ways to discover music. The current solutions are either based on creating blogs that appeal to a wide group of music listeners (such as GoodMusicAllDay.com) and are inconsistent in delivering music that I enjoy, or are based around predictive algorithms that are not terribly effective either. Additionally, many of the mobile application solutions are too complex to navigate. SoundByte was envisioned to incorporate solutions to all of the aforementioned problems.

After conceiving the initial concept for SoundByte, I developed a value proposition by using exercises from Diana Kander's *So What? Who Cares? Why You?*[6]. To see whether the initial concept and value proposition had viability, I conducted the first of two non-live rounds of interviews - the first being purely behavioral. The behavioral round was conducted via one-on-one interviews with twelve participants and was meant to illicit responses in regards to their music discovery behaviors. From these interviews, I was able to conclude that there were people who had similar contentions to me in regards to music discovery. They believe that they can listen to 30 seconds of a song and know whether or not they like it and believe their friends are a better predictor of music likeness than other music sources.

After receiving affirmation that my value proposition and initial concept had merit, I began working on the design of the application. I designed the application to have a tab bar layout with five tabs that users could navigate through. The first tab (where the user is directed to immediately after signing in) is the

Timeline tab which lists all of the friends of that respective user. The second is a Friend Search tab where users can search for all of their friends. The third is a Song Search tab where users can search for add songs to their playlist using Spotify's music library. The fourth is a Settings tab where users can go to log out, and the fifth is a Favorites tab where users can see all of the songs they have liked. To host the database where all of the information pertaining to SoundByte is stored, I used Parse.com's iOS SDK.

Upon completing the first prototype of the application, I did another round of customer interviews. This round involved vetting a prototype of the application out to a group of twelve interviewees by allowing them to use the application on my iPhone 5s. They were instructed to use the application as they liked for around 5-10 minutes using a database of songs that I collected from the other eleven interviewees. After their 5-10 minutes of interacting with the application, I asked them a series of questions with the intention of creating a dialogue to document their thoughts, suggestions, and opinions about the application.

The second round of interviews was effective in getting feed and opinions on SoundByte, but to gather more information about how it would be used in a live setting, I decided to do a third round of interviews that involved distributing SoundByte in a live environment to friends of mine at Union College. More specifically, I decided to put the application on ten user's phones and allowed them to use the application at their leisure for three full days. This round of interviews did not go as well as planned - the interviewees hardly interacted with the product. This showed me that SoundByte is not yet ready to be deployed to the App Store.

In conclusion, over the course of three terms, I created a mobile application that implemented my personal belief that using a peer-to-peer network in which users share 30-second song clips with an easy-to-use interface is more efficient than current music discovery solutions. I tested this by conducting three rounds of interviews - two of which were successful in proving SoundByte at a conceptual level, while the other proved that it was not ready to be taken into a marketplace. The culmination of interviews has led me to believe that there is a place in the market for SoundByte to exist and thrive, but it needs a lot of further development before it reaches that level.

References

[1] *So What? Who Cares? Why You? The Inventor's Commercialization Toolkit*. Ottawa, ON, Canada, second edition, 2006.

[2] Lighting up new markets. *IFPI Digital Music Report*, 2014.

[3] Streaming infographics. *Statista Infographics*, 2015.

[4] Who's winning the u.s. smartphone market? *Newswire*, November 2015.

[5] I.M. Crawford. *Marketing Research and Information Systems*. Marketing and agribusiness texts. Food and Agriculture Organizations of the United Nations, 1997.

[6] Diana Kander. *All in Startup: Launching a New Idea When Everything Is on the Line*. 2015.

[7] Press Room. Music discovery still dominated by radio. *Nielson Music 360 Report*, 2012.

[8] Spotify. *Spotify iOS SDK Reference*, 2016.

[9] Spotify. *Web API Endpoint Reference*, 2016.

Appendices

A *So What? Who Cares? Why You?* Exercises

Exercises from Chapters 1-5 in Wendy Kennedy's *So What? Who Cares? Why You?* book.

A.A Chapter 1

A.A.1 Exercise 1A: The Napkin Drawing Tool

A.A.2 Exercise 1B: The Idea Scoping Tool

Design	• iOS phone application • Simple interface
Attributes	• Peer-to-peer music sharing • 30-second song samples • Lightweight interface only for discovery and not streaming • No new technologies, but is the only mobile application that incorporates all three
Advantages	• Lightweight and easy to use • 30-second song samples allow users to listen to more songs in a fraction of the time. • The peer-to-peer network allows users to choose who they want to discover music from
Proof Points	• Pre-App Store Prototype completed • Music discovery survey showed that the majority of respondents prefer discovering music from friends, only need 30-seconds to know whether or not they like a song, and have issues with their current means for discovering music
Legal and Regulatory	• Completely my idea

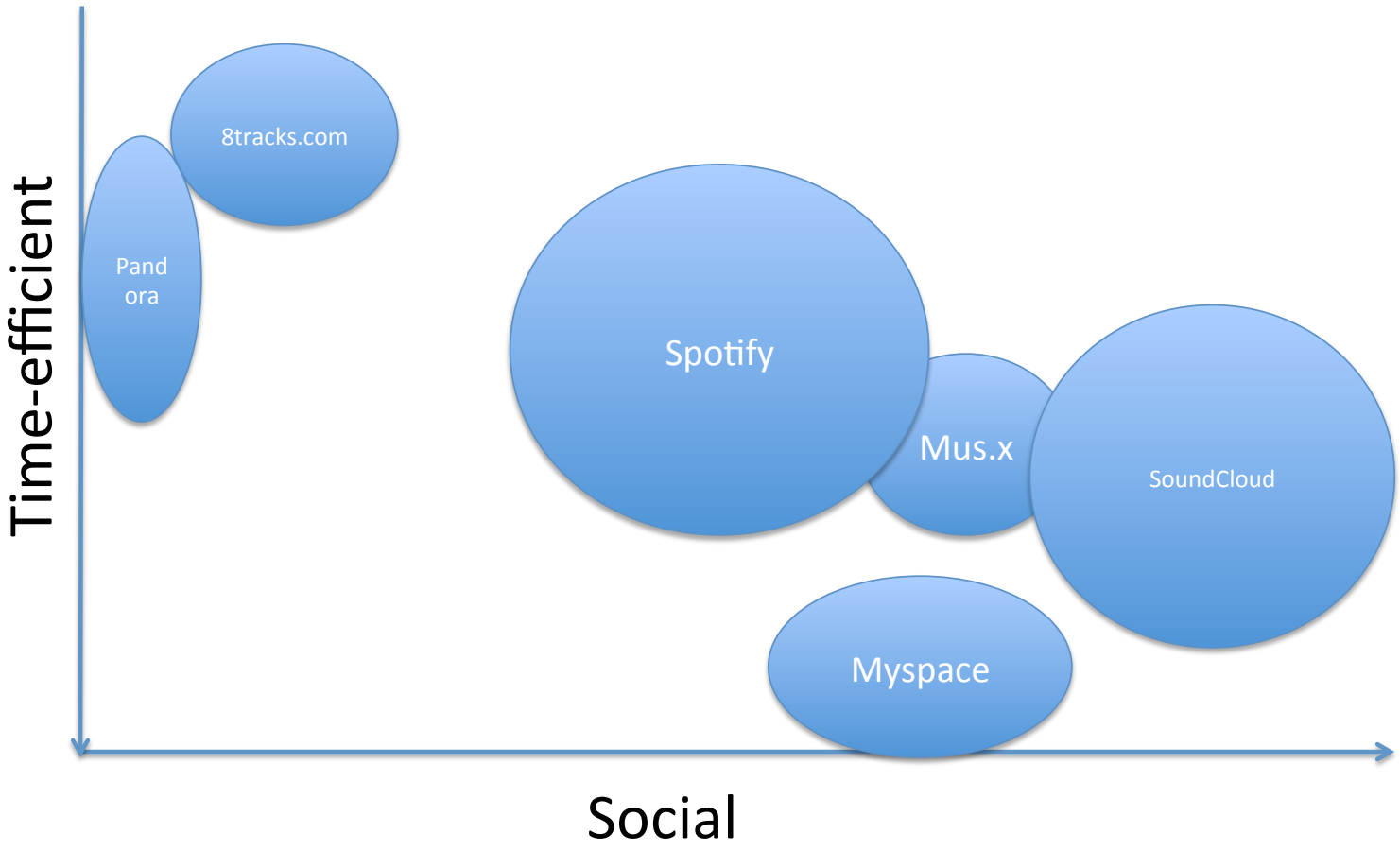
A.B Chapter 2

A.B.1 Exercise 2A: The Commercial Problem Tool

Proof	• No music discovery service is a clear dominant • Customer interviewees said they do not need a full song to know whether or not they like a song • Customers interviewees said they trust their peer's inputs more
Priority	• Big issue for avid music listeners constantly looking for new music • Not as big of an issue for those who do not look for music on a daily or weekly basis
Pain	• Music discovery takes too long • Music discovery services are heavyweight • The algorithms that are currently in use are not accurate
Problem Statement	For avid music listeners, discovering new music is a time-consuming and often unsuccessful endeavor. SoundByte alleviates this pain by providing an environment that delivers new music from ones peers in a timely and efficient manner.

A.C Chapter3

A.C.1 Exercise 3A: The Category Map Tool



A.D Chapter 4

A.D.1 Exercise 4A: The Market Fishbone Tool

Unique Attributes of Your Idea:

- Peer-to-peer
- 30-second song clips
- Lightweight and easy to use

Fin 1: Avid music listeners

Sub-sector: Constantly searching for new music

Sub-sector: Want to show people the music they found

Fin 2: Average music listeners

Sub-sector: Occasionally looking for new music

Sub-sector: Too lazy to use sites

Fin 3: People who have difficulty finding new music

Sub-sector: Do not like current music discovery sources

Sub-sector: Want to save time

Fin 4: People / websites who want to promote their music

Sub-sector: Business opportunity

Sub-sector: Self promotion

A.D.2 Exercise 4B: The Segment Strawman Tool

Name Your Segment: Avid music listeners

Definers: Looking for new music on at least a weekly basis. Rely on friends to get music.

Context: More new music

Descriptors: Want to discover new music, share music with friends, spend less time searching for new music

Compatibility: Saves time, easy to use, social network

A.E Chapter 5

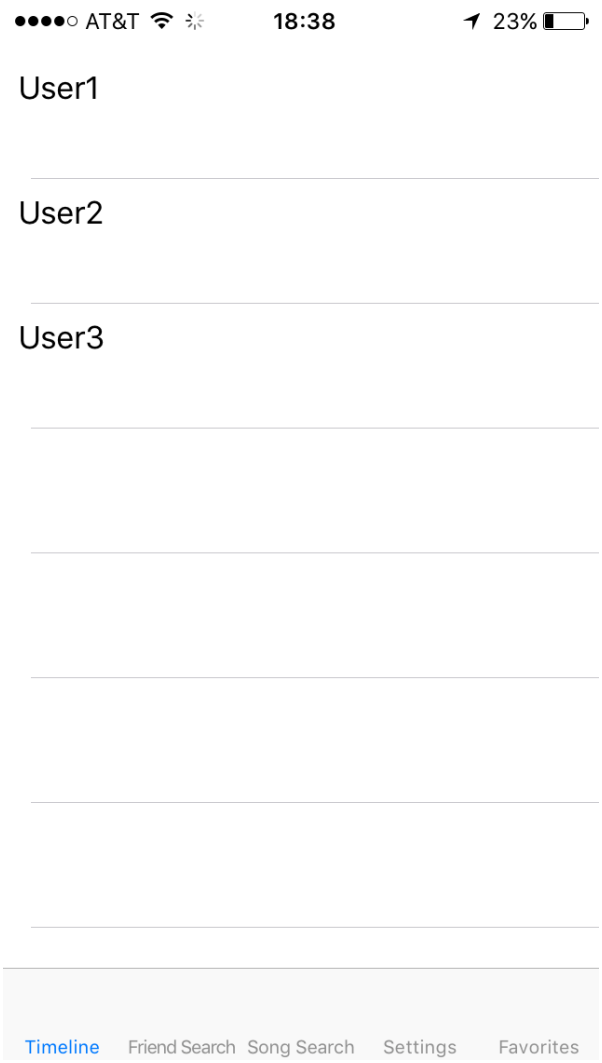
A.E.1 Exercise 5A: The Ecosystem Ladder Tool

	External Specialists	Interviewees	App Store Users
Layers			
Marketing	-User interface -Threading -Speed	-Testimonials	-Reviews
App Store	-Help with review process		-Downloads
User Interface	-Better aesthetically -User experience	-Provide feedback -Give suggestions	
Customer Interviews		-Provide feedback -Viability	
Prototype		-Testing VP out	

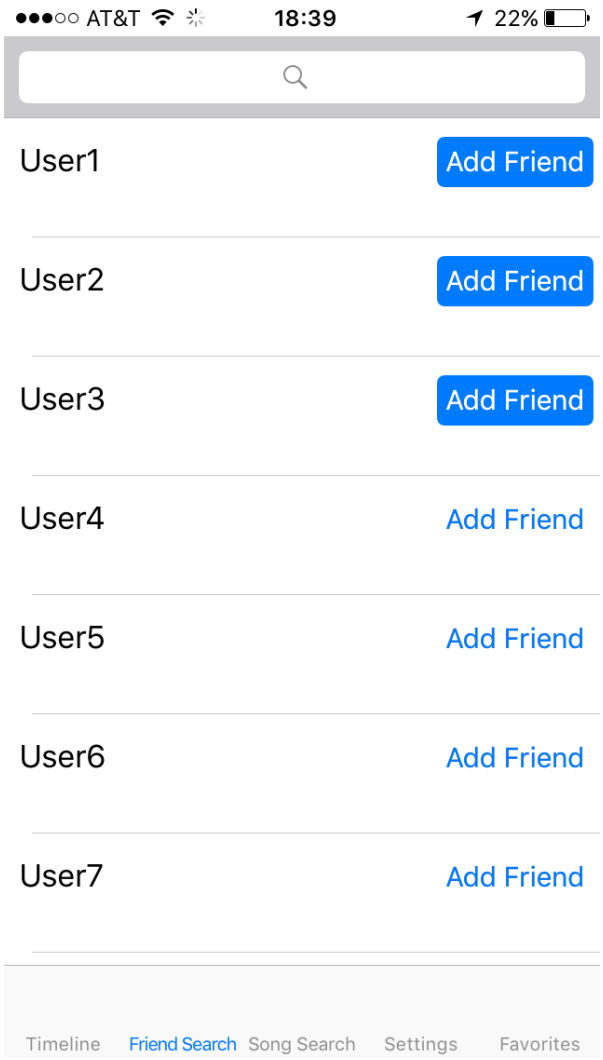
B SoundByte Screenshots

Holds pictures of the various tabs as well as the playlist view.

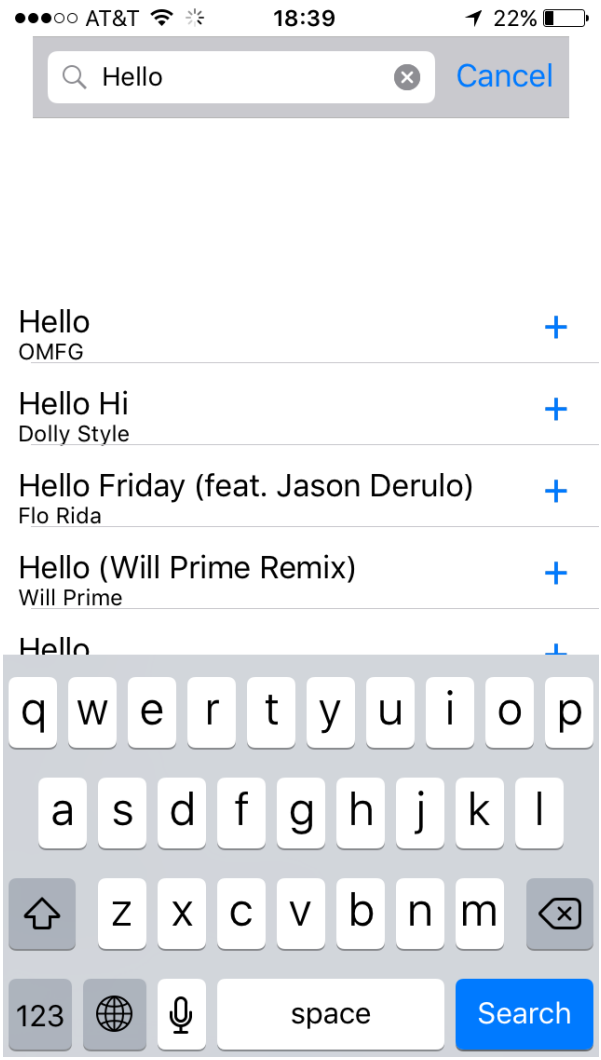
B.A Timeline Tab



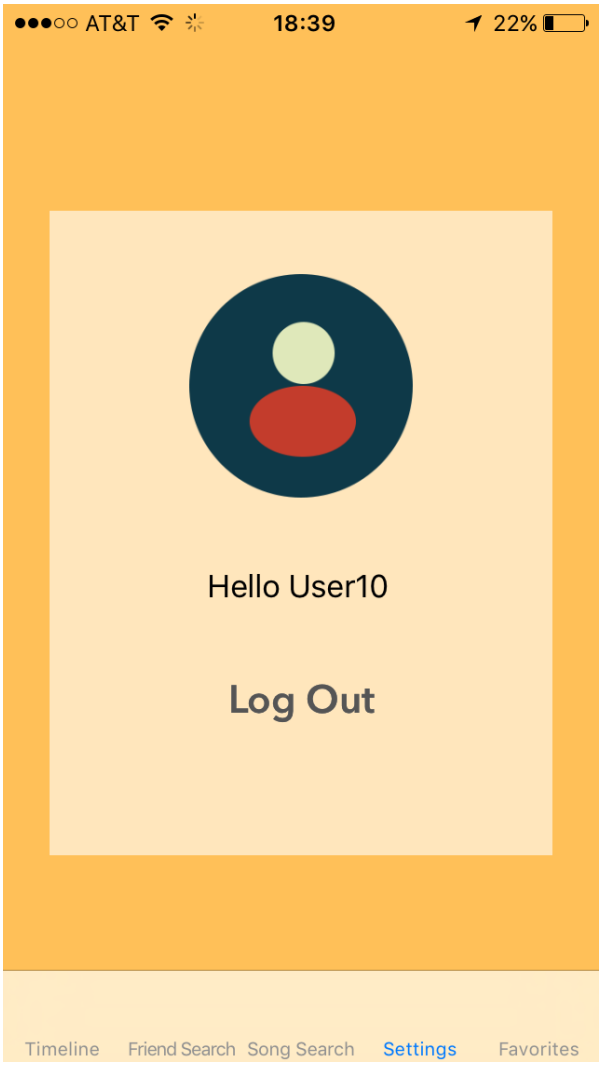
B.B Friend Search Tab



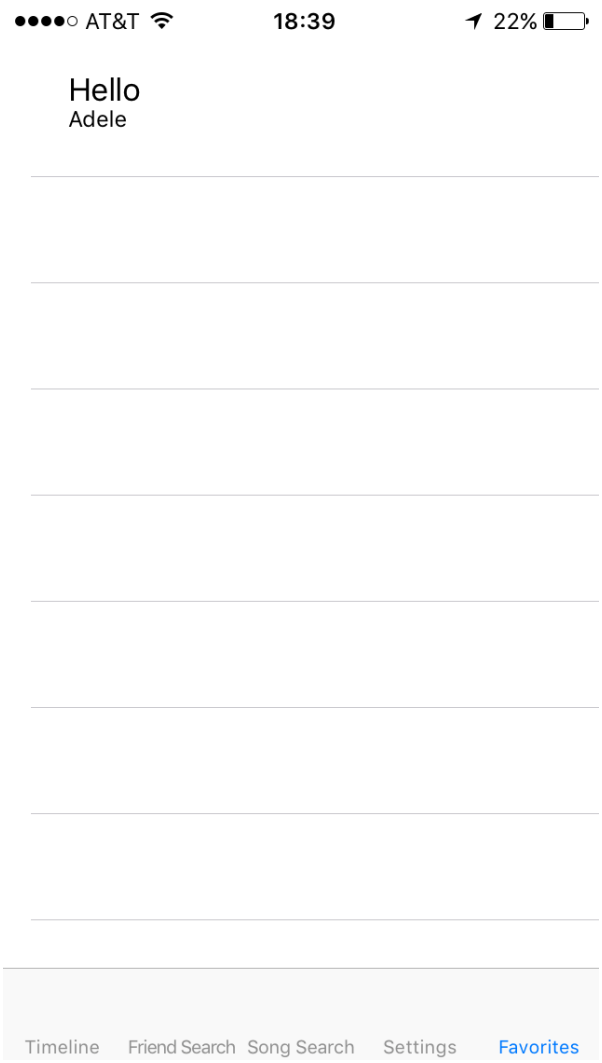
B.C Song Search Tab



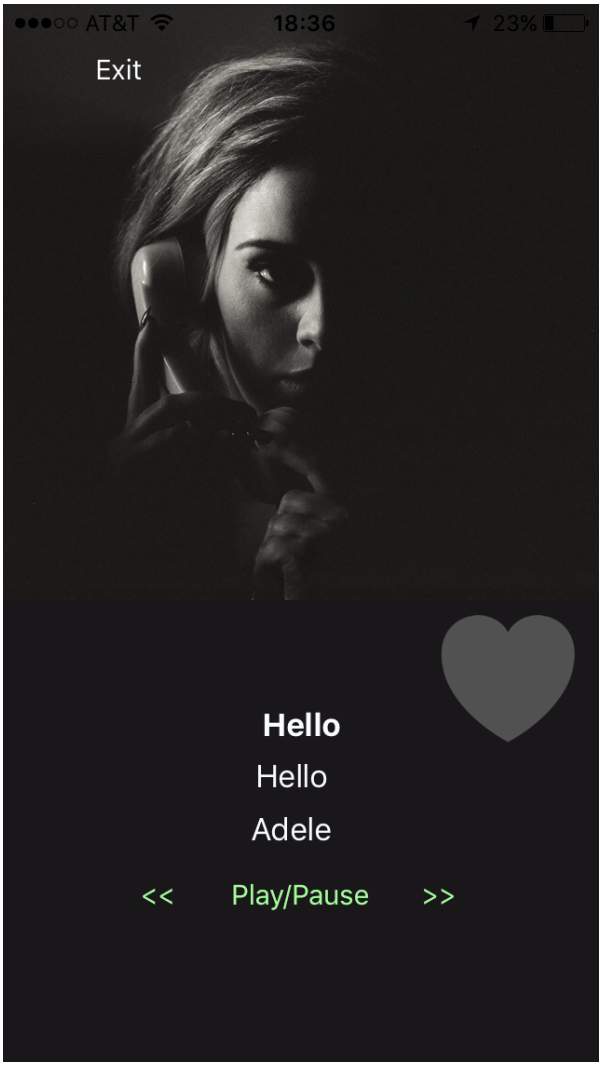
B.D Settings Tab



B.E Favorites Tab



B.F Playlist



```
//
//  AppDelegate.swift
//  SoundByte
//
//  Created by Jeff Cohen on 10/26/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//

import UIKit
import CoreData
import Parse
import Bolts



@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    var auth: SPTAuth = SPTAuth.defaultInstance()
    let kClientID = "cf5b0855e8f440719ad3a1811e704fe3"
    let kCallbackURL = "soundbyte://return-after-login"
    let kTokenSwapURL = ""
    let kTTokenRefreshServiceURL = ""
    var kSessionUserDefaultsKey = "SpotifySession"

    func delay(delay:Double, closure:()->()) {
        dispatch_after(
            dispatch_time(
                DISPATCH_TIME_NOW,
                Int64(delay * Double(NSEC_PER_SEC))
            ),
            dispatch_get_main_queue(), closure)
    }

    func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -
> Bool {

        Parse.enableLocalDatastore()
        // Initialize Parse.

Parse.setApplicationId("WtGRPWzBj8ZHiNsGOTqXWIVE1lPMafB2jTDyhi6H",
            clientKey: "UmJp9oYG8HhqFZk4aXHyD0QJZlmNcPA5AuztdhKb")
        return true
    }

    func application(application: UIApplication, openURL url: NSURL,
sourceApplication: String?, annotation: AnyObject) -> Bool {
```

```
        // Ask SPTAuth if the URL given is a Spotify authentication
callback
        if (SPTAuth.defaultInstance().canHandleURL(url)) {

SPTAuth.defaultInstance().handleAuthCallbackWithTriggeredAuthURL(url,
callback: { (error, session) -> Void in
                if (error != nil) {
                    print("*** Auth error: \(error)")
                    return
                }

                //let nav = self.window?.rootViewController as!
UINavigationController
                //let vc = nav.topViewController as!
SpotifyLoginViewController

            })

            return true
        }

        return false
    }


    func applicationWillResignActive(application: UIApplication) {
        // Sent when the application is about to move from active to
inactive state. This can occur for certain types of temporary
interruptions (such as an incoming phone call or SMS message) or when
the user quits the application and it begins the transition to the
background state.
        // Use this method to pause ongoing tasks, disable timers, and
throttle down OpenGL ES frame rates. Games should use this method to
pause the game.
    }

    func applicationDidEnterBackground(application: UIApplication) {
        // Use this method to release shared resources, save user
data, invalidate timers, and store enough application state
information to restore your application to its current state in case
it is terminated later.
        // If your application supports background execution, this
method is called instead of applicationWillTerminate: when the user
quits.
    }

    func applicationWillEnterForeground(application: UIApplication) {
        // Called as part of the transition from the background to the
inactive state; here you can undo many of the changes made on entering
the background.
```

```
    }

    func applicationDidBecomeActive(application: UIApplication) {
        // Restart any tasks that were paused (or not yet started)
while the application was inactive. If the application was previously
in the background, optionally refresh the user interface.
    }

    func applicationWillTerminate(application: UIApplication) {
        // Called when the application is about to terminate. Save
data if appropriate. See also applicationDidEnterBackground:.
        // Saves changes in the application's managed object context
before the application terminates.
        self.saveContext()
    }

    // MARK: - Core Data stack

    lazy var applicationDocumentsDirectory: NSURL = {
        // The directory the application uses to store the Core Data
store file. This code uses a directory named "SoundByte.SoundByte" in
the application's documents Application Support directory.
        let urls =
NSFileManager.defaultManager().URLsForDirectory(.DocumentDirectory,
inDomains: .UserDomainMask)
        return urls[urls.count-1]
    }()

    lazy var managedObjectModel: NSManagedObjectModel = {
        // The managed object model for the application. This property
is not optional. It is a fatal error for the application not to be
able to find and load its model.
        let modelURL =
NSBundle.mainBundle().URLForResource("SoundByte", withExtension:
"momd")!
        return NSManagedObjectModel(contentsOfURL: modelURL)!
    }()

    lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator?
= {
        // The persistent store coordinator for the application. This
implementation creates and return a coordinator, having added the
store for the application to it. This property is optional since there
are legitimate error conditions that could cause the creation of the
store to fail.
        // Create the coordinator and store
        var coordinator: NSPersistentStoreCoordinator? =
NSPersistentStoreCoordinator(managedObjectModel:
self.managedObjectModel)
        let url =
```

```
self.applicationDocumentsDirectory.URLByAppendingPathComponent("SoundB
yte.sqlite")
        var error: NSError? = nil
        var failureReason = "There was an error creating or loading
the application's saved data."
        do {
            try
coordinator!.addPersistentStoreWithType(NSSQLiteStoreType,
configuration: nil, URL: url, options: nil)
        } catch var error1 as NSError {
            error = error1
            coordinator = nil
            // Report any error we got.
            var dict = [String: AnyObject]()
            dict[NSLocalizedDescriptionKey] = "Failed to initialize
the application's saved data"
            dict[NSLocalizedFailureReasonErrorKey] = failureReason
            dict[NSUnderlyingErrorKey] = error
            error = NSError(domain: "YOUR_ERROR_DOMAIN", code: 9999,
userInfo: dict)
            // Replace this with code to handle the error
appropriately.
            // abort() causes the application to generate a crash log
and terminate. You should not use this function in a shipping
application, although it may be useful during development.
            NSLog("Unresolved error \(error), \(error!.userInfo)")
            abort()
        } catch {
            fatalError()
        }

        return coordinator
    }()

    lazy var managedObjectContext: NSManagedObjectContext? = {
        // Returns the managed object context for the application
(which is already bound to the persistent store coordinator for the
application.) This property is optional since there are legitimate
error conditions that could cause the creation of the context to fail.
        let coordinator = self.persistentStoreCoordinator
        if coordinator == nil {
            return nil
        }
        var managedObjectContext = NSManagedObjectContext()
        managedObjectContext.persistentStoreCoordinator = coordinator
        return managedObjectContext
    }()

    // MARK: - Core Data Saving support
```

```
    func saveContext () {
        if let moc = self.managedObjectContext {
            var error: NSError? = nil
            if moc.hasChanges {
                do {
                    try moc.save()
                } catch let error1 as NSError {
                    error = error1
                    // Replace this implementation with code to handle
the error appropriately.
                    // abort() causes the application to generate a
crash log and terminate. You should not use this function in a
shipping application, although it may be useful during development.
                    NSLog("Unresolved error \(error), \
(error!.userInfo)")
                    abort()
                }
            }
        }
    }

}

//
//  FavoritesTableViewCell.swift
//  SoundByte
//
//  Created by Jeff Cohen on 2/9/16.
//  Copyright (c) 2016 Jeff Cohen. All rights reserved.
//

import UIKit

class FavoritesTableViewCell: UITableViewCell {

    @IBOutlet weak var artistName: UILabel!
    @IBOutlet weak var songName: UILabel!


    override func awakeFromNib() {
        super.awakeFromNib()
        // Initialization code
    }

    override func setSelected(selected: Bool, animated: Bool) {
        super.setSelected(selected, animated: animated)

        // Configure the view for the selected state
    }
```

```
}
//
//  FavoritesViewController.swift
//  SoundByte
//
//  Created by Jeff Cohen on 2/9/16.
//  Copyright (c) 2016 Jeff Cohen. All rights reserved.
//

import UIKit

class FavoritesViewController: UIViewController {
    @IBOutlet weak var tableView: UITableView!
    var songWithInfoDictionary : [String : (String, String)] = [:]
        {
        didSet{
            dispatch_async(dispatch_get_main_queue(), {
                self.tableView.reloadData()
            })
        }
    }
    var songURIArray = [String]()
        //{
//        didSet{
//            tableView.reloadData()
//        }
//    }

    func fetchNewSong(notification: NSNotification){
        let newestSong = notification.userInfo!["newLikedSong"] as!
String
        self.songURIArray.append(newestSong)
        self.fetchNameAndArtist(newestSong)
    }


    func fetchNameAndArtist(uriTrackAsString: String!) -> [String :
(String, String)]{
        let uriTrack = NSURL(string: uriTrackAsString)
        SPTTrack.trackWithURI(uriTrack, session: nil) { (error, track)
-> Void in
            if let track = track as? SPTTrack, artist =
track.artists.first as? SPTPartialArtist{
                self.songWithInfoDictionary.updateValue((track.name,
artist.name), forKey: uriTrackAsString)

            }
        }
```

```
        return self.songWithInfoDictionary
    }

    func getShit(){
        self.songWithInfoDictionary.removeAll()
        let pointer = PFObject(withoutDataWithClassName: "_User",
objectId: PFUser.currentUser()!.objectId!)
        //var query = PFUser.query()
        let likesQuery = PFQuery(className: "Like")
        let finalQuery = likesQuery.whereKey("fromUser", equalTo:
pointer)
        finalQuery.findObjectsInBackgroundWithBlock({
            (results: [PFObject]?, error: NSError?) -> Void in
            if error == nil{
                if let results = results{
                    for result in results{
                        let uriTrackAsString = result["likedSongURI"]
as! String

                        let uriTrack = NSURL(string:
result["likedSongURI"] as! String)
                        SPTTrack.trackWithURI(uriTrack, session: nil)
{ (error, track) -> Void in
                            if let track = track as? SPTTrack, artist
= track.artists.first as? SPTPartialArtist{

self.songWithInfoDictionary.updateValue((track.name, artist.name),
forKey: uriTrackAsString)

self.songURIArray.append(uriTrackAsString)
//
dispatch_async(dispatch_get_main_queue()){
//                                      self.tableView.reloadData()
//                                  }
                            }
                        }
                        //
self.fetchNameAndArtist(result["likedSongURI"] as! String)
                        //self.tableView.reloadData()

                    }
                }
            }
            else{
                return
            }


        })
    }
```

```
    override func viewDidLoad() {
        super.viewDidLoad()
        getShit()
        NSNotificationCenter.defaultCenter().addObserver(self,
selector: "fetchNewSong", name: "likeButtonClicked", object: nil)
    }

    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)


        //fetchLikedSongs()
        //self.tableView.reloadData()
        //NSLog("yo")

        // Do any additional setup after loading the view.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

}

extension FavoritesViewController: UITableViewDataSource{

    func tableView(tableView: UITableView, numberOfRowsInSection
section: Int) -> Int{

        return self.songWithInfoDictionary.count ?? 0
    }

    func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {
        let cell =
tableView.dequeueReusableCellWithIdentifier("favoriteCell") as!
FavoritesTableViewCell
        //NSLog("\(self.songWithInfoDictionary.count)")
        var (songTitle, artistName) =
self.songWithInfoDictionary[self.songURIArray[indexPath.row]]!
        cell.songName.text = songTitle
        cell.artistName.text = artistName
        return cell
    }

}
//
//  FriendPlaylistViewController.swift
```

```
//  SoundByte
//
//  Created by Jeff Cohen on 12/15/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//

import UIKit
import AVKit
import AVFoundation
import Parse
import ConvenienceKit


class FriendPlaylistViewController: UIViewController,
SPTAudioStreamingPlaybackDelegate {
    @IBOutlet weak var likeButton: UIButton!
    var songBeingPlayedURI : String!
    var songDictionary: [NSURL : String] = [:]
    var viaSegue: String!
    var likes = [NSURL]?()
    let kClientID = "cf5b0855e8f440719ad3a1811e704fe3"
    let kCallbackURL = "soundbyte://return-after-login"
    //let kTokenSwapURL = "http://lochttp://localhost:1234/
refreshalhost:1234/swap"
    //let kTokenRefreshURL = ""
    var queuePlayer: AVQueuePlayer!
    var player: SPTAudioStreamingController!
    let spotifyAuthenticator = SPTAuth.defaultInstance()
    var songsArray = [AVPlayerItem]()
    var IDArray = [String]()
    var audioPlayer = AVPlayer()
    // All necessary labels including image views
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var albumLabel: UILabel!
    @IBOutlet weak var shadedCoverView: UIImageView!
    @IBOutlet weak var coverView: UIImageView!
    @IBOutlet weak var artistLabel: UILabel!
    @IBOutlet weak var spinner: UIActivityIndicatorView!

    func wasSongAlreadyLiked(){

dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAU
LT, 0), {() -> Void in
            var pointer = PFObject(withoutDataWithClassName: "_User",
objectId: PFUser.currentUser()!.objectId!)
            // _ = PFUser.query()
            var likesQuery = PFQuery(className: "Like")
            //if (try! likesQuery.findObjects().count > 1){
                var newLikesQuery =
likesQuery.whereKey("likedSongURI", equalTo: self.songBeingPlayedURI)
```

```
                var finalQuery = newLikesQuery.whereKey("fromUser",
equalTo: pointer)

            dispatch_async(dispatch_get_main_queue(), {() -> Void in
                if (try! finalQuery.findObjects().count != 0){
                    self.likeButton.selected = true
                }
                else{
                    self.likeButton.selected = false
                }
            })
            //}
        })
    }

    override func viewDidAppear(animated: Bool) {
        wasSongAlreadyLiked()
    }

    override func viewDidLoad() {
        NSNotificationCenter.defaultCenter().addObserver(self,
selector: "updateUI", name: "sessionUpdated", object: nil)

        super.viewDidLoad()
        self.titleLabel.text = "Nothing Playing"
        self.albumLabel.text = ""
        self.artistLabel.text = ""

        let selectedFriendQuery = PFUser.query()!
        _ = selectedFriendQuery.whereKey("username", equalTo:
viaSegue)
        let selectedFriendName = try!
selectedFriendQuery.getFirstObject()
        let userSelectedFriendName = selectedFriendName.objectId

        let playlistFromFollowedUsers = PFQuery(className: "Playlist")
        let pointer = PFObject(withoutDataWithClassName: "_User",
objectId: userSelectedFriendName)
        playlistFromFollowedUsers.whereKey("user", equalTo: pointer)
        self.queuePlayer = AVQueuePlayer()
           // AVQueuePlayer(items: nil)

        playlistFromFollowedUsers.findObjectsInBackgroundWithBlock({

            (result: [PFObject]?, error: NSError?) -> Void in

            var songIDs = result as [PFObject]!
            //NSLog("\(songIDs.count())")
            if songIDs.count < 1{
                return
```

```
            }
            else{

                for i in 0...songIDs.count-1{
                   _ = songIDs[i].valueForKey("spotifyTrackNumber")
as! String

self.IDArray.append(songIDs[i].valueForKey("spotifyTrackNumber") as!
String)
                    let apiURL = "https://api.spotify.com/v1/tracks/\
(self.IDArray[i])"
                    let url = NSURL(string: apiURL)

                    let urlRequest = NSMutableURLRequest(URL: url!) as
NSMutableURLRequest
                    //let headersAuth = NSString(format: "Bearer %@",
spotifyAuthenticator.session.accessToken)
                    //urlRequest.setValue(headersAuth as? String,
forHTTPHeaderField: "Authorization")

                    let queue = NSOperationQueue()

NSURLConnection.sendAsynchronousRequest(urlRequest, queue: queue,
completionHandler: {(response: NSURLResponse?, recievedData: NSData?,
error: NSError?) -> Void in
                        if error != nil{
                            print(error!.localizedDescription)
                        }
                        else{
                            var err : NSError? = nil
                            let jsonResult : NSDictionary = (try!
NSJSONSerialization.JSONObjectWithData(recievedData!, options:
NSJSONReadingOptions.AllowFragments)) as! NSDictionary
                            if err == nil{
                                let songPreview =
jsonResult.objectForKey("preview_url") as! String
                                let songURI =
jsonResult.objectForKey("uri") as! String
                                var asset: AVURLAsset =
AVURLAsset(URL: (NSURL(string: songPreview))!, options: nil)
                                var playerItem = AVPlayerItem(asset:
asset)


self.queuePlayer.insertItem(playerItem, afterItem:
self.queuePlayer.items().last as? AVPlayerItem! ??
self.queuePlayer.items().first)

self.songDictionary.updateValue(songURI, forKey:
playerItem.valueForKey("URL") as! NSURL)
```

```
                                if (self.queuePlayer.items().count <=
1) {
                                    self.updateUI(NSURL(string:
songURI))
                                    self.songBeingPlayedURI = songURI
                                }
                            }
                            else{
                                print(err?.localizedDescription)
                            }
                        }
                    })


                }
            }
            self.queuePlayer.addObserver(self, forKeyPath:
"currentItem", options: [.New, .Initial], context:
&self.songDictionary)
            self.queuePlayer.play()


        })
    }


    override func observeValueForKeyPath(keyPath: String?, ofObject
object: AnyObject?, change: [String : AnyObject]?, context:
UnsafeMutablePointer<Void>) {
        if keyPath == "currentItem", let player = object as? AVPlayer,
        currentItem = player.currentItem?.asset as? AVURLAsset {
            let newSongURI =
self.songDictionary[currentItem.valueForKey("URL") as! NSURL]
            if newSongURI != nil{
             self.updateUI(NSURL(string: newSongURI!))
             self.songBeingPlayedURI = newSongURI

             wasSongAlreadyLiked()
            }

        }
    }


    func updateUI(uriTrack: NSURL!){
        let auth: SPTAuth = SPTAuth.defaultInstance()
        if uriTrack == nil{
            self.coverView.image = nil
            //self.shadedCoverView.image = nil
            return
```

```
        }
        self.spinner.startAnimating()
        SPTTrack.trackWithURI(uriTrack, session: auth.session)
{ (error, track) -> Void in
            if let track = track as? SPTTrack, artist =
track.artists.first as? SPTPartialArtist{
            self.titleLabel.text = track.name
            self.albumLabel.text = track.album.name
            //var artist = track.artists[0] as! SPTPartialTrack
            self.artistLabel.text = artist.name
            let imageURL = track.album.largestCover.imageURL
            if imageURL == nil{
                NSLog("This album doesnt have any images!",
track.album)
                self.coverView.image = nil
                self.shadedCoverView.image = nil
                return
            }

dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAU
LT, 0),{() -> Void in
                var error: NSError? = nil
                var image: UIImage? = nil
                let imageData = NSData(contentsOfURL: imageURL)
                if imageData != nil{
                    image = UIImage(data: imageData!)
                }
                dispatch_async(dispatch_get_main_queue(), {() -> Void
in
                    self.spinner.stopAnimating()
                    self.coverView.image = image
                    if image == nil{
                        NSLog("Couldnt load cover image ")
                        return
                    }
                    })
                //var blurred: UIImage = self.applyBlurOnImage(image!,
withRadius: 10.0)
//                dispatch_async(dispatch_get_main_queue(), {() ->
Void in
//                    self.shadedCoverView.image = blurred
//                })
            })
        }
        }

    }
```

```
    @IBAction func likeButtonClicked(sender: AnyObject) {
        let pointer = PFObject(withoutDataWithClassName: "_User",
objectId: PFUser.currentUser()!.objectId!)
        //var query = PFUser.query()
        let likesQuery = PFQuery(className: "Like")
        let personQuery = likesQuery.whereKey("fromUser", equalTo:
pointer)
        let newLikesQuery = likesQuery.whereKey("likedSongURI",
equalTo: self.songBeingPlayedURI)
        let finalQuery = newLikesQuery.whereKey("fromUser", equalTo:
pointer)
        if (try! finalQuery.findObjects().count == 0){
            let likeObject = PFObject(className: "Like")
            likeObject.setObject(self.songBeingPlayedURI, forKey:
"likedSongURI")
            likeObject.setObject(PFUser.currentUser()!, forKey:
"fromUser")
            likeObject.saveEventually()
            self.likeButton.selected = true
            var selectedSong = ["newLikedSong" :
self.songBeingPlayedURI]
           NSLog("\(self.songBeingPlayedURI)")

NSNotificationCenter.defaultCenter().postNotificationName("likeButtonC
licked", object: nil, userInfo: selectedSong)
        }
        else{
            //finalQuery.delete(nil)
            finalQuery.findObjectsInBackgroundWithBlock {( results:
[PFObject]?, error: NSError?) -> Void in
                if let results = results as? [PFObject]!{
                    for likes in results{
                        likes.delete(nil) //
deleteInBackgroundWithBlock(nil)
                    }

                }
            }
            self.likeButton.selected = false
        }


    }

    @IBAction func exitButton(sender: AnyObject) {
        self.queuePlayer.pause()
    }

    @IBAction func nextSongButton(sender: AnyObject) {
//        NSLog("\(self.queuePlayer.items().count-1)")
```

```
//        NSLog("\(self.queuePlayer.items().endIndex)")
//        if (self.queuePlayer.items().endIndex == 1){
//            self.queuePlayer.items().first
//            //return
//        }
        self.queuePlayer.advanceToNextItem()
    }

    @IBAction func previousSongButton(sender: AnyObject) {
      //self.queuePlayer.items().
    }


    @IBAction func playAndPauseButton(sender: AnyObject) {
        if self.queuePlayer.rate == 1.0{
            self.queuePlayer.pause()
        }
        else if (self.queuePlayer.rate == 0.0){
            self.queuePlayer.play()
        }
    }


    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}

//
//  FriendSearchTableViewCell.swift
//  SoundByte
//
//  Created by Jeff Cohen on 10/29/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//

import UIKit
import Parse

protocol FriendSearchTableViewCellDelegate: class {
    func cell(cell: FriendSearchTableViewCell, didSelectFollowUser
user: PFUser)
    func cell(cell: FriendSearchTableViewCell, didSelectUnfollowUser
user: PFUser)
}

class FriendSearchTableViewCell: UITableViewCell {

    @IBOutlet weak var usernameLabel: UILabel!
```

```
    @IBOutlet weak var followButton: UIButton!
    weak var delegate: FriendSearchTableViewCellDelegate?

    var user: PFUser? {
        didSet {
            usernameLabel.text = user?.username
        }
    }

    var canFollow: Bool? = true {
        didSet {
            /*
            Change the state of the follow button based on whether or
not
            it is possible to follow a user.
            */
            if let canFollow = canFollow {
                followButton.selected = !canFollow
            }
        }
    }

    @IBAction func followButtonTapped(sender: AnyObject) {
        if let canFollow = canFollow where canFollow == true {
            delegate?.cell(self, didSelectFollowUser: user!)
            self.canFollow = false
        } else {
            delegate?.cell(self, didSelectUnfollowUser: user!)
            self.canFollow = true
        }
    }
}//
//  FriendsSearchViewController.swift
//  SoundByte
//
//  Created by Jeff Cohen on 10/29/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//

import UIKit
import ConvenienceKit
import Parse
import Bond

class FriendsSearchViewController: UIViewController {
    @IBOutlet weak var tableView: UITableView!
    @IBOutlet weak var searchBar: UISearchBar!

    // stores all the users that match the current search query
    var users: [PFUser]?
```

```
    /*
    This is a local cache. It stores all the users this user is
following.
    It is used to update the UI immediately upon user interaction,
instead of waiting
    for a server response.
    */
    var followingUsers: [PFUser]? {
        didSet {
            /**
            the list of following users may be fetched after the
tableView has displayed
            cells. In this case, we reload the data to reflect
"following" status
            */
            tableView.reloadData()
        }
    }

//    func doesUserContain(user: PFUser) -> Bool{
//       // containQuery = ParseHelper.allUsers(updateList)
//        var followedUsers = PFQuery(className: "Follow")
//        var currentUsersFriends = followedUsers.whereKey("fromUser",
equalTo: user) ?? []
////        if currentUsersFriends.countObjects() > 0{
////            return true
////        }
////        else{
////            return false
////        }
//
//    }

    // the current parse query
    var query: PFQuery? {
        didSet {
            // whenever we assign a new query, cancel any previous
requests
            oldValue?.cancel()
        }
    }

    // this view can be in two different states
    enum State {
        case DefaultMode
        case SearchMode
    }

    // whenever the state changes, perform one of the two queries and
```

```
update the list
    var state: State = .DefaultMode {
        didSet {
            switch (state) {
            case .DefaultMode:
                query = ParseHelper.allUsers(updateList)

            case .SearchMode:
                let searchText = searchBar?.text ?? ""
                query = ParseHelper.searchUsers(searchText,
completionBlock:updateList)
            }
        }
    }

    // MARK: Update userlist

    /**
    Is called as the completion block of all queries.
    As soon as a query completes, this method updates the Table View.
    */
    func updateList(results: [PFObject]?, error: NSError?) {
        self.users = results as? [PFUser] ?? []
        self.tableView.reloadData()

    }

    // MARK: View Lifecycle

    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)

        state = .DefaultMode

        // fill the cache of a user's followees
        ParseHelper.getFollowingUsersForUser(PFUser.currentUser()!) {
            (results: [PFObject]?, error: NSError?) -> Void in
            let relations = results as? [PFObject]! ?? []
            // use map to extract the User from a Follow object
            self.followingUsers = relations.map {
                $0.objectForKey(ParseHelper.ParseFollowToUser) as!
PFUser
            }

        }
    }

}

// MARK: TableView Data Source
```

```
extension FriendsSearchViewController: UITableViewDataSource {

    func tableView(tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
        return self.users?.count ?? 0
    }

    func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {
        let cell =
tableView.dequeueReusableCellWithIdentifier("UserCell") as!
FriendSearchTableViewCell

        let user = users![indexPath.row]
        cell.user = user

        if let followingUsers = followingUsers {

            // check if current user is already following displayed
user
            // change button appereance based on result
            cell.canFollow = !followingUsers.contains(user)

           // cell.canFollow = !contains(followingUsers, user)
        }

        cell.delegate = self

        return cell
    }
}

// MARK: Searchbar Delegate

extension FriendsSearchViewController: UISearchBarDelegate {

    func searchBarTextDidBeginEditing(searchBar: UISearchBar) {
        searchBar.setShowsCancelButton(true, animated: true)
        state = .SearchMode
    }

    func searchBarCancelButtonClicked(searchBar: UISearchBar) {
        searchBar.resignFirstResponder()
        searchBar.text = ""
        searchBar.setShowsCancelButton(false, animated: true)
        state = .DefaultMode
    }

    func searchBar(searchBar: UISearchBar, textDidChange searchText:
```

```
String) {
        ParseHelper.searchUsers(searchText,
completionBlock:updateList)
    }

}

// MARK: FriendSearchTableViewCell Delegate

extension FriendsSearchViewController:
FriendSearchTableViewCellDelegate {

    func cell(cell: FriendSearchTableViewCell, didSelectFollowUser
user: PFUser) {

ParseHelper.addFollowRelationshipFromUser(PFUser.currentUser()!,
toUser: user)
        // update local cache
        followingUsers?.append(user)
    }

    func cell(cell: FriendSearchTableViewCell, didSelectUnfollowUser
user: PFUser) {
        if let followers = followingUsers {

ParseHelper.removeFollowRelationshipFromUser(PFUser.currentUser()!,
toUser: user)
            // update local cache
            //followers = followers.filter { $0.username !=
user.username}
            //removeObject(user, fromArray: &followingUsers)
            self.followingUsers = followers
        }
    }

}


//
//  LoginViewController.swift
//  SoundByte
//
//  Created by Jeff Cohen on 10/26/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//

import Foundation
import UIKit
import Parse
import Bolts
```

```
class LoginViewController: UIViewController {

    //text field for username
    @IBOutlet weak var userEmailTextField: UITextField!

    //text field for password
    @IBOutlet weak var userPasswordTextField: UITextField!

    //Segue name
    let loginViewControllerSegue = "LoginSuccessful"

    override func viewDidLoad() {
        if PFUser.currentUser() != nil{

self.performSegueWithIdentifier(self.loginViewControllerSegue, sender:
nil)
        }
        super.viewDidLoad()

        // Do any additional setup after loading the view.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }



    //Action when login button is tapped
    @IBAction func loginButtonTapped(sender: AnyObject) {

        var userEmail = userEmailTextField.text
        userEmail = userEmail!.lowercaseString
        let userPassword = userPasswordTextField.text

        PFUser.logInWithUsernameInBackground(userEmail!, password:
userPassword!){
            user, error in
            if user != nil{

self.performSegueWithIdentifier(self.loginViewControllerSegue, sender:
nil)
            }else if let error = error{
                self.showErrorView(error)

            }
        }
    }
```

```
}

//
//  ParseHelper.swift
//  SoundByte
//
//  Created by Jeff Cohen on 11/9/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//

import Foundation
import Parse
// MARK: Following

/**
Fetches all users that the provided user is following.

- parameter user: The user whose followees you want to retrieve
- parameter completionBlock: The completion block that is called when
the query completes
*/

class ParseHelper{

    // Following Relation
    static let ParseFollowClass       = "Follow"
    static let ParseFollowFromUser    = "fromUser"
    static let ParseFollowToUser      = "toUser"
    static let ParseUserUsername      = "username"
    static let ParseSongClass         = "Playlist"
    static let ParseLikeClass         = "Like"
    static let ParseLikeToPost        = "toPost"
    static let ParseLikeFromUser      = "fromUser"

    static func getFollowingUsersForUser(user: PFUser,
completionBlock: PFQueryArrayResultBlock){ //PFArrayResultBlock) {
    let query = PFQuery(className: ParseFollowClass)

    query.whereKey(ParseFollowFromUser, equalTo:user)
    query.findObjectsInBackgroundWithBlock(completionBlock)
}

    static func getFollowingSongsForUser(user: PFUser,
completionBlock: PFQueryArrayResultBlock){
        let query = PFQuery(className: ParseSongClass)
        var pointer = PFObject(withoutDataWithClassName: "_User",
objectId: PFUser.currentUser()!.objectId!)
        query.whereKey("user", equalTo: pointer)
        query.findObjectsInBackgroundWithBlock(completionBlock)
```

```
    }



/**
Establishes a follow relationship between two users.

- parameter user:    The user that is following
- parameter toUser:  The user that is being followed
*/
static func addFollowRelationshipFromUser(user: PFUser, toUser:
PFUser) {
    let followObject = PFObject(className: ParseFollowClass)
    followObject.setObject(user, forKey: ParseFollowFromUser)
    followObject.setObject(toUser, forKey: ParseFollowToUser)

    followObject.saveInBackgroundWithBlock(nil)
}

static func addFollowSongRelationshipToUser(song: AnyObject, user:
PFUser ){
        let followObject = PFObject(className: ParseSongClass)
        followObject.setObject(user, forKey: "user")

        let str = song.uri.description
        let index1 = song.uri.description.startIndex.advancedBy(14)
        let subStr = str.substringFromIndex(index1)
        followObject.setObject(subStr, forKey: "spotifyTrackNumber")
        followObject.saveInBackgroundWithBlock(nil)
}

static func removeFollowSongRelationshipToUser(song: AnyObject, user:
PFUser ){
    var pointer = PFObject(withoutDataWithClassName: "_User",
objectId: PFUser.currentUser()!.objectId!)
    let followObject = PFQuery(className: ParseSongClass)
    let followingUser = followObject.whereKey(user.objectId!, equalTo:
pointer)
    let followingSong = followingUser.whereKey(song as! String,
equalTo: "spotifyTrackNumber")
    followingSong.delete(nil)
    }

/**
Deletes a follow relationship between two users.

- parameter user:    The user that is following
- parameter toUser:  The user that is being followed
*/
static func removeFollowRelationshipFromUser(user: PFUser, toUser:
```

```
PFUser) {
    let query = PFQuery(className: ParseFollowClass)
    query.whereKey(ParseFollowFromUser, equalTo:user)
    query.whereKey(ParseFollowToUser, equalTo: toUser)

    query.findObjectsInBackgroundWithBlock {
        (results: [PFObject]?, error: NSError?) -> Void in

        let results = results as? [PFObject]! ?? []

        for follow in results {
            follow.deleteInBackgroundWithBlock(nil)
        }
    }
}
    static func likePost(user: PFUser) {
        let likeObject = PFObject(className: ParseLikeClass)
        likeObject.setObject(user, forKey: ParseLikeFromUser)
        likeObject.saveInBackgroundWithBlock(nil)
    }

    static func unlikePost(user: PFUser) {
        let query = PFQuery(className: ParseLikeClass)
        query.whereKey(ParseLikeFromUser, equalTo: user)
        //query.whereKey(ParseLikeToPost, equalTo: post)

        query.findObjectsInBackgroundWithBlock { (results:
[PFObject]?, error: NSError?) -> Void in
            if let results = results as? [PFObject]! {
                for likes in results {
                    likes.deleteInBackgroundWithBlock(nil)
                }
            }
        }
    }


// MARK: Users

/**
Fetch all users, except the one that's currently signed in.
Limits the amount of users returned to 20.

- parameter completionBlock: The completion block that is called when
the query completes

- returns: The generated PFQuery
*/
static func allUsers(completionBlock: PFQueryArrayResultBlock) ->
PFQuery {
```

```
    let query = PFUser.query()!
    // exclude the current user
    query.whereKey(ParseHelper.ParseUserUsername,
        notEqualTo: PFUser.currentUser()!.username!)
    query.orderByAscending(ParseHelper.ParseUserUsername)
    query.limit = 20

    query.findObjectsInBackgroundWithBlock(completionBlock)

    return query
}

/**
Fetch users whose usernames match the provided search term.

- parameter searchText: The text that should be used to search for
users
- parameter completionBlock: The completion block that is called when
the query completes

- returns: The generated PFQuery
*/
static func searchUsers(searchText: String, completionBlock:
PFQueryArrayResultBlock)
    -> PFQuery {
        /*
        NOTE: We are using a Regex to allow for a case insensitive
compare of usernames.
        Regex can be slow on large datasets. For large amount of data
it's better to store
        lowercased username in a separate column and perform a regular
string compare.
        */
        let query =
PFUser.query()!.whereKey(ParseHelper.ParseUserUsername,
            matchesRegex: searchText, modifiers: "i")

        query.whereKey(ParseHelper.ParseUserUsername,
            notEqualTo: PFUser.currentUser()!.username!)

        query.orderByAscending(ParseHelper.ParseUserUsername)
        query.limit = 20

        query.findObjectsInBackgroundWithBlock(completionBlock)

        return query
}
}

//
```

```
//  RegisterPageViewController.swift
//  SoundByte
//
//  Created by Jeff Cohen on 10/26/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//

import Foundation
import UIKit

class RegisterPageViewController: UIViewController {

    @IBOutlet weak var userEmailTextField: UITextField!
    @IBOutlet weak var userPasswordTextField: UITextField!
    @IBOutlet weak var repeatPasswordTextField: UITextField!

    let signUpSuccessful = "SignupSuccessful"

    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

//    func validateEmail(candidate: String) -> Bool {
//        let emailRegex = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-
z]{2,6}"
//        return NSPredicate(format: "SELF MATCHES %@",
emailRegex).evaluateWithObject(candidate)
//    }



    @IBAction func registerButton(sender: UIButton) {
        let userEmail = userEmailTextField.text
        let userPassword = userPasswordTextField.text
        let userRepeatPassword = repeatPasswordTextField.text

        //Check for empty fields
        if(userEmail!.isEmpty || userPassword!.isEmpty ||
userRepeatPassword!.isEmpty){
            displayMyAlertMessage("All fields are required")
            return
        }
```

```
        //Validates password length
        if (userPassword!.characters.count > 17 ||
(userPassword!.characters.count<5)){
            displayMyAlertMessage("Password must be between 5 and 12
characters")
            return
        }

        //check if passwords match
        if(userPassword != userRepeatPassword){
            //Display an alert message
            displayMyAlertMessage("Passwords do not match")
            return
        }


        //Store data
        let user = PFUser()
        user.username = userEmailTextField.text
        user.password = userPasswordTextField.text
        user.signUpInBackgroundWithBlock {succeeded,error in
            if succeeded{
                self.performSegueWithIdentifier(self.signUpSuccessful,
sender: nil)
            }
            else if let error = error{
                self.showErrorView(error)
            }

        }

    }
    //Display alert message with confirmation
    func displayMyAlertMessage(userMessage:String){
        let myAlert = UIAlertController(title:"Alert",
message:userMessage, preferredStyle:UIAlertControllerStyle.Alert);

        let okAction = UIAlertAction(title:"OK",
style:UIAlertActionStyle.Default, handler:nil);

        myAlert.addAction(okAction);

        self.presentViewController(myAlert, animated:true,
completion:nil);
    }

}

//
```

```
//  HomeViewController.swift
//  SoundByte
//
//  Created by Jeff Cohen on 10/26/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//


import UIKit
import Parse

class SettingsViewController: UIViewController {

    @IBOutlet weak var userNameLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()
        // Show the current visitor's username
        if let pUserName = PFUser.currentUser()?.username {
            self.userNameLabel.text = "Hello " + pUserName
        }
    }


    @IBAction func logoutButtonTapped(sender: AnyObject) {

    // Send a request to log out a user
    PFUser.logOut()

        dispatch_async(dispatch_get_main_queue(), { () -> Void in
            let viewController:UIViewController = UIStoryboard(name:
"Main", bundle: nil).instantiateViewControllerWithIdentifier("Login")
            self.presentViewController(viewController, animated: true,
completion: nil)
        })

    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }


}

//
//  SongSearchTableViewCell.swift
//  SoundByte
//
```

```
//  Created by Jeff Cohen on 12/11/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//

import UIKit
import Parse

protocol SongSearchTableViewCellDelegate: class {
    func cell(cell: SongSearchTableViewCell, didSelectFollowSong song:
AnyObject?)
    func cell(cell: SongSearchTableViewCell, didSelectUnFollowSong
song: AnyObject?)
}

class SongSearchTableViewCell: UITableViewCell {

    @IBOutlet weak var artistSearchLabel: UILabel!
    @IBOutlet weak var songSearchLabel: UILabel!
    @IBOutlet weak var addSongSearchButton: UIButton!
    weak var delegate: SongSearchTableViewCellDelegate?


    var songURI: AnyObject?

    var canFollow: Bool? = true {
        didSet {
            /*
            Change the state of the follow button based on whether or
not
            it is possible to follow a user.
            */
            if let canFollow = canFollow {
                addSongSearchButton.selected = !canFollow
            }
        }
    }


    @IBAction func songFollowButtonTapped(sender: AnyObject) {
        if let canFollow = canFollow where canFollow == true {
            delegate?.cell(self, didSelectFollowSong: songURI!)
            ParseHelper.addFollowSongRelationshipToUser(songURI!,
user: PFUser.currentUser()!)
            self.canFollow = false
        } else {
            delegate?.cell(self, didSelectUnFollowSong: songURI!)
            ParseHelper.removeFollowSongRelationshipToUser(songURI!,
user: PFUser.currentUser()!)
            self.canFollow = true
        }
```

```
    }
}
//
//  SongSearchViewController.swift
//  SoundByte
//
//  Created by Jeff Cohen on 12/7/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//

import UIKit


//run "ruby spotify_token_swap.rb" to launch server


class SongSearchViewController: UIViewController, SPTAuthViewDelegate,
SPTAudioStreamingPlaybackDelegate {

    let kClientID = "cf5b0855e8f440719ad3a1811e704fe3"
    let kCallbackURL = "soundbyte://return-after-login"
    //let kTokenSwapURL = "http://localhost:1234/swap"
    //let kTokenRefreshURL = "http://localhost:1234/refresh"


    var songsAlreadyLiked: [String]?

    @IBOutlet weak var tableViewSongResults: UITableView!
    @IBOutlet weak var songSearchBar: UISearchBar!
    var player: SPTAudioStreamingController?
    let spotifyAuthenticator = SPTAuth.defaultInstance()
    var spotifyListPage: SPTListPage?

    @IBOutlet weak var spotifyLoginButton: UIButton!

    var followingSongs: [String]?{
        didSet{
            tableViewSongResults.reloadData()
        }
    }

    // the current parse query
    var query: PFQuery? {
        didSet {
            // whenever we assign a new query, cancel any previous
requests
            oldValue?.cancel()
        }
    }
```

```
    // this view can be in two different states
    enum State {
        case DefaultMode
        case SearchMode
    }

    // whenever the state changes, perform one of the two queries and
update the list
    var state: State = .DefaultMode {
        didSet {
            switch (state) {
            case .DefaultMode:
                query = ParseHelper.allUsers(updateList)

            case .SearchMode:
                let searchText = songSearchBar?.text ?? ""
                query = ParseHelper.searchUsers(searchText,
completionBlock:updateList)
            }
        }
    }

    // MARK: Update userlist

    /**
    Is called as the completion block of all queries.
    As soon as a query completes, this method updates the Table View.
    */
    func updateList(results: [PFObject]?, error: NSError?) {
        self.tableViewSongResults.reloadData()

    }



    @IBAction func loginWithSpotify(sender: AnyObject) {
        spotifyAuthenticator.clientID = kClientID
        spotifyAuthenticator.requestedScopes = [SPTAuthStreamingScope]
        spotifyAuthenticator.redirectURL = NSURL(string: kCallbackURL)
        // spotifyAuthenticator.tokenSwapURL = NSURL(string:
kTokenSwapURL)
        //spotifyAuthenticator.tokenRefreshURL = NSURL(string:
kTokenRefreshURL)
        let spotifyAuthenticationViewController =
SPTAuthViewController.authenticationViewController()
        spotifyAuthenticationViewController.delegate = self
        spotifyAuthenticationViewController.modalPresentationStyle =
UIModalPresentationStyle.OverCurrentContext
        spotifyAuthenticationViewController.definesPresentationContext
= true
```

```
        presentViewController(spotifyAuthenticationViewController,
animated: false, completion: nil)
    }

    // SPTAuthViewDelegate protocol methods

    func authenticationViewController(authenticationViewController:
SPTAuthViewController!, didLoginWithSession session: SPTSession!) {
        let auth: SPTAuth = SPTAuth.defaultInstance()
        setupSpotifyPlayer()
        //NSLog("\(auth.session.description)")
        loginWithSpotifySession(auth.session)
    }

    func
authenticationViewControllerDidCancelLogin(authenticationViewControlle
r: SPTAuthViewController!) {
        print("login cancelled")
    }

    func authenticationViewController(authenticationViewController:
SPTAuthViewController!, didFailToLogin error: NSError!) {
        print("login failed")
    }
//
    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)
        state = .DefaultMode
        ParseHelper.getFollowingSongsForUser(PFUser.currentUser()!) {
            (results: [PFObject]?, error: NSError?) -> Void in
            let relations = results as? [PFObject]! ?? []
            // use map to extract the User from a Follow object
            self.followingSongs = relations.map {
                $0.valueForKey("spotifyTrackNumber") as! String
            }

        }

    }

    func sessionUpdatedNotification (notification: NSNotification) ->
Void{
        if self.navigationController?.topViewController == self{
            let auth: SPTAuth = SPTAuth.defaultInstance()
            if auth.session.isValid(){

                self.setupSpotifyPlayer()
                self.loginWithSpotifySession(auth.session)

            }
```

```
        }
    }

    var IDArray = [String]()

    override func viewDidLoad() {
        NSNotificationCenter.defaultCenter().addObserver(self,
selector: "sessionUpdatedNotification", name: "sessionUpdated",
object: nil)
        self.spotifyLoginButton.hidden = true
        let followingQuery = PFQuery(className: "Follow")
        followingQuery.whereKey("fromUser",
equalTo:PFUser.currentUser()!)

        let playlistFromFollowedUsers = PFQuery(className: "Playlist")
        playlistFromFollowedUsers.whereKey("user", matchesKey:
"toUser", inQuery: followingQuery)

        playlistFromFollowedUsers.findObjectsInBackgroundWithBlock({

            (result: [PFObject]?, error: NSError?) -> Void in


            var songIDs = result as! [PFObject]!
            if songIDs.count < 1{
                return
            }
            else{
                for i in 0...songIDs.count-1{

self.IDArray.append(songIDs[i].valueForKey("spotifyTrackNumber") as!
String)
                    //self.tableView.reloadData()

                }
            }

        })
    }

    func grabSong(){
        let followingQuery = PFQuery(className: "Follow")
        followingQuery.whereKey("fromUser",
equalTo:PFUser.currentUser()!)

        let playlistFromFollowedUsers = PFQuery(className: "Playlist")
        playlistFromFollowedUsers.whereKey("user", matchesKey:
"toUser", inQuery: followingQuery)

        for i in 0...IDArray.count-1{
```

```
            let SpotifyURI = IDArray[i]
            self.player!.playURIs([NSURL(string: SpotifyURI)!],
withOptions: nil, callback: nil)
        }
    }


    // SPTAudioStreamingPlaybackDelegate protocol methods

    private

    func setupSpotifyPlayer() {
        player = SPTAudioStreamingController(clientId:
spotifyAuthenticator.clientID) // can also use kClientID; they're the
same value
        player!.playbackDelegate = self
        player!.diskCache = SPTDiskCache(capacity: 1024 * 1024 * 64)
    }

    func loginWithSpotifySession(session: SPTSession) {
        if spotifyAuthenticator.session.accessToken != nil{
            self.spotifyLoginButton.hidden = true
        }

        player!.loginWithSession(session, callback: { (error:
NSError!) in
            if error != nil {
                print("Couldn't login with session: \(error)")
                return
            }
            //self.grabSong()

        })
    }

    func useLoggedInPermissions() {

        //let spotifyURI = PFQuery()
        //spotifyURI.whereKey(<#key: String#>, containedIn:
<#[AnyObject]#>)
        //let spotifyURI = PFUser.currentUser().
        //let spotifyURI = "spotify:track:4h0zU3O9R5xzuTmNO7dNDU)"
        //player!.playURIs([NSURL(string: spotifyURI)!], withOptions:
nil, callback: nil)
    }
}

extension SongSearchViewController: UISearchBarDelegate {

    func searchBarTextDidBeginEditing(searchBar: UISearchBar) {
```

```
        searchBar.setShowsCancelButton(true, animated: true)
        state = .SearchMode
    }

    func searchBarCancelButtonClicked(searchBar: UISearchBar) {
        searchBar.resignFirstResponder()
        searchBar.text = ""
        searchBar.setShowsCancelButton(false, animated: true)
        state = .DefaultMode
    }

    func searchBar(searchBar: UISearchBar, textDidChange searchText:
String) {
        SPTSearch.performSearchWithQuery(searchText, queryType:
SPTSearchQueryType.QueryTypeTrack, accessToken: nil, callback:
{( error, result) -> Void in
            if let result = result as? SPTListPage{
                self.spotifyListPage = result

                self.tableViewSongResults.reloadData()
            }
            //  }
            //}
        })

    }

}

extension SongSearchViewController: UITableViewDataSource {

    func tableView(tableView: UITableView, numberOfRowsInSection
section: Int) -> Int{
        if spotifyListPage?.items == nil{
            return 1
        }
        return spotifyListPage!.items.count
    }

    func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {

        let cell =
tableView.dequeueReusableCellWithIdentifier("SongCell") as!
SongSearchTableViewCell


        cell.addSongSearchButton.hidden = true
        if spotifyListPage?.items == nil{
            cell.songSearchLabel!.text = "No Results Found"
```

```
            cell.artistSearchLabel.hidden = true
        }

        else{
            cell.addSongSearchButton.hidden = false
            //var partialTrack =
self.spotifyListPage?.items[indexPath.row].artists?.first.description
            cell.artistSearchLabel!.text =
self.spotifyListPage?.items[indexPath.row].artists?.first!.name
            cell.songSearchLabel!.text =
self.spotifyListPage?.items[indexPath.row].name
            let song = self.spotifyListPage?.items[indexPath.row]
            let URISong = song!.uri.description
            //NSLog("\(song!.uri.description)")
            cell.songURI = song
            if let followingSongs = followingSongs{
                cell.canFollow = !followingSongs.contains(URISong)
            }
        }


        cell.delegate = self

        return cell
    }
}



extension SongSearchViewController: SongSearchTableViewCellDelegate {

    func cell(cell: SongSearchTableViewCell, didSelectFollowSong song:
AnyObject?) {
    }
    func cell(cell: SongSearchTableViewCell, didSelectUnFollowSong
song: AnyObject?) {
    }
}
//
//  SpotifyLoginViewController.swift
//  SoundByte
//
//  Created by Jeff Cohen on 1/12/16.
//  Copyright (c) 2016 Jeff Cohen. All rights reserved.
//

import UIKit

class SpotifyLoginViewController: UIViewController,
SPTAuthViewDelegate, SPTAudioStreamingPlaybackDelegate {
```

```
    let kClientID = "cf5b0855e8f440719ad3a1811e704fe3"
    let kCallbackURL = "soundbyte://return-after-login"
    //let kTokenSwapURL = "http://localhost:1234/swap"
    //let kTokenRefreshURL = "http://localhost:1234/refresh"

    var player: SPTAudioStreamingController?
    let spotifyAuthenticator = SPTAuth.defaultInstance()
    let spotifyLoginViewControllerSegue = "SpotifyLoginSuccessful"

    override func viewDidLoad(){
        super.viewDidLoad()
        NSNotificationCenter.defaultCenter().addObserver(self,
selector: "sessionUpdatedNotification", name:
UIApplicationWillEnterForegroundNotification, object: nil)
//        var auth: SPTAuth = SPTAuth.defaultInstance()
//        if (auth.session.isValid()){
//
self.performSegueWithIdentifier(spotifyLoginViewControllerSegue,
sender: nil)
//        }
    }

    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)
        let auth: SPTAuth = SPTAuth.defaultInstance()
        if (auth.session == nil){
            return
        }

        //check if auth is still valid
        if (auth.session.isValid()){
            NSLog("viewWillAppear shit")

self.performSegueWithIdentifier(spotifyLoginViewControllerSegue,
sender: nil)
        }

        if (auth.hasTokenRefreshService){
            self.renewTokenAndShowPlayer()
            return
        }
    }

    func renewTokenAndShowPlayer(){
        let auth: SPTAuth = SPTAuth.defaultInstance()
        auth.renewSession(auth.session, callback:{(error: NSError!,
session: SPTSession!) -> Void in
            auth.session = session
            if error != nil{
```

```
                NSLog("***Error renewing session: %@", error)
                return
            }
            NSLog("something to do with renewtokenandshow")

self.performSegueWithIdentifier(self.spotifyLoginViewControllerSegue,
sender: nil)
        })
    }

    func sessionUpdatedNotification (notification: NSNotification) ->
Void{

            let auth: SPTAuth = SPTAuth.defaultInstance()
            if auth.session.isValid(){
                NSLog("something to do with sessionupdatedshit")

self.performSegueWithIdentifier(spotifyLoginViewControllerSegue,
sender: nil)


        }
    }



    @IBAction func loginWithSpotify(sender: AnyObject) {
        spotifyAuthenticator.clientID = kClientID
        spotifyAuthenticator.requestedScopes = [SPTAuthStreamingScope]
        spotifyAuthenticator.redirectURL = NSURL(string: kCallbackURL)
        // spotifyAuthenticator.tokenSwapURL = NSURL(string:
kTokenSwapURL)
        //spotifyAuthenticator.tokenRefreshURL = NSURL(string:
kTokenRefreshURL)
        let spotifyAuthenticationViewController =
SPTAuthViewController.authenticationViewController()
        spotifyAuthenticationViewController.delegate = self
        spotifyAuthenticationViewController.modalPresentationStyle =
UIModalPresentationStyle.OverCurrentContext
        spotifyAuthenticationViewController.definesPresentationContext
= true
        presentViewController(spotifyAuthenticationViewController,
animated: false, completion: nil)
    }

    // SPTAuthViewDelegate protocol methods

    func authenticationViewController(authenticationViewController:
SPTAuthViewController!, didLoginWithSession session: SPTSession!) {
        let auth: SPTAuth = SPTAuth.defaultInstance()
```

```
self.performSegueWithIdentifier(spotifyLoginViewControllerSegue,
sender: nil)
        setupSpotifyPlayer()
        loginWithSpotifySession(auth.session)
    }

    func
authenticationViewControllerDidCancelLogin(authenticationViewControlle
r: SPTAuthViewController!) {
        print("login cancelled")
    }

    func authenticationViewController(authenticationViewController:
SPTAuthViewController!, didFailToLogin error: NSError!) {
        print("login failed")
    }

    private

    func setupSpotifyPlayer() {
        player = SPTAudioStreamingController(clientId:
spotifyAuthenticator.clientID) // can also use kClientID; they're the
same value
        player!.playbackDelegate = self
        player!.diskCache = SPTDiskCache(capacity: 1024 * 1024 * 64)
    }


    func loginWithSpotifySession(session: SPTSession) {
        if spotifyAuthenticator.session.accessToken != nil{

self.performSegueWithIdentifier(self.spotifyLoginViewControllerSegue,
sender: nil)
        }
        player!.loginWithSession(session, callback: { (error:
NSError!) in
            if error != nil {
                print("Couldn't login with session: \(error)")
                return
            }


        })
    }
}

//
//  FriendsTableViewController.swift
//  SoundByte
```

```
//
//  Created by Jeff Cohen on 10/29/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//

import UIKit
import Parse

class StartingTabViewController: UITabBarController{

    override func viewWillAppear(animated: Bool) {
        self.tabBarController?.navigationItem.hidesBackButton = true
        super.viewDidLoad()
    }
   }
//
//  TimelineTableViewCell.swift
//  SoundByte
//
//  Created by Jeff Cohen on 12/14/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//

import UIKit

class TimelineTableViewCell: UITableViewCell {


    @IBOutlet weak var usernameLabel: UILabel!
    override func awakeFromNib() {
        super.awakeFromNib()
        // Initialization code
    }

    var passedValue: String!

    override func setSelected(selected: Bool, animated: Bool) {
        super.setSelected(selected, animated: animated)

        // Configure the view for the selected state
    }

}



//
//  TimelineViewController.swift
//  SoundByte
//
```

```
//  Created by Jeff Cohen on 12/1/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//

import AVFoundation
import UIKit
import Parse
import AVKit

public var SelectedSongNumber = Int()
//public var valueToPass: String!

class TimelineViewController: UIViewController{



    var valueToPass: [PFObject]!
    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }
    @IBOutlet weak var tableView: UITableView!
    var nameArray = [String]() {
        didSet{
            tableView.reloadData()
        }
    }

    @IBAction func prepareForUnwind(segue: UIStoryboardSegue) {

    }

    override func canPerformUnwindSegueAction(action: Selector,
fromViewController: UIViewController, withSender sender: AnyObject) ->
Bool {
        if (self.respondsToSelector(action)){
            return true
        }
        return false
    }

    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)
        nameArray.removeAll()
        var usersname = "username"
        let findUserObjectId = PFQuery(className: "Follow")
        findUserObjectId.whereKey("fromUser", equalTo:
PFUser.currentUser()!)

        findUserObjectId.findObjectsInBackgroundWithBlock { (results:
[PFObject]?, error: NSError?) -> Void in
```

```
            if error == nil {
                if let results = results{
                    for result in results {
                        let user : PFUser = result["toUser"] as!
PFUser
                        let queryUsers = PFUser.query()


queryUsers!.getObjectInBackgroundWithId(user.objectId!, block:
{( userGet: PFObject?, error: NSError?) -> Void in
                                if let userGet = userGet{
                                    self.valueToPass?.append(userGet)

self.nameArray.append(userGet.objectForKey("username") as! String)
                                    self.tableView.reloadData()
                                }
                            })
                    }
                }
            } else{
                    print(error)
                    return
            }
        }
    }
}

extension TimelineViewController: UITableViewDataSource {

    override func prepareForSegue(segue: UIStoryboardSegue, sender:
AnyObject?) {
        if (segue.identifier == "friendPlaylist"){
            if let destination = segue.destinationViewController as?
FriendPlaylistViewController{
                let path = tableView.indexPathForSelectedRow!
                //let cell = tableView.cellForRowAtIndexPath(path!)
                destination.viaSegue = self.nameArray[path.row]

            }

        }
    }

        func tableView(tableView: UITableView, numberOfRowsInSection
section: Int) -> Int{
            return self.nameArray.count ?? 0
        }

        func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {
```

```
            let cell =
tableView.dequeueReusableCellWithIdentifier("PostCell") as!
TimelineTableViewCell
            cell.usernameLabel.text = self.nameArray[indexPath.row]
            return cell
        }

        func tableView(tableView: UITableView, didSelectRowAtIndexPath
indexPath: NSIndexPath){
            _ = tableView.indexPathForSelectedRow!
            if let _ = tableView.cellForRowAtIndexPath(indexPath){
                self.performSegueWithIdentifier("friendPlaylist",
sender: self)
            }
            SelectedSongNumber = indexPath.row
        }



    }
//
//  UIViewControllerExtension.swift
//  SoundByte
//
//  Created by Jeff Cohen on 10/27/15.
//  Copyright (c) 2015 Jeff Cohen. All rights reserved.
//

import Foundation
import UIKit

extension UIViewController{

    func showErrorView(error: NSError) {
        if let errorMessage = error.userInfo["error"] as? String {
            let alert = UIAlertController(title: "Error", message:
errorMessage, preferredStyle: UIAlertControllerStyle.Alert)
            alert.addAction(UIAlertAction(title: "Ok", style:
UIAlertActionStyle.Default, handler: nil))
            presentViewController(alert, animated: true, completion:
nil)
        }
    }
}
```