The Report committee for Timothy Andrew Box

Certifies that this is the approved version of the following report:

Swapbeat.com: A Streaming, Adaptive Music Service in the Browser

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor: _____
Christine Julien

_____
Joydeep Ghosh

Swapbeat.com: A Streaming, Adaptive Music Service in the Browser

by

Timothy Andrew Box, B.A.; M.P.Aff.

Report

Presented to the Faculty of the Graduate School

of the University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2015

“Necessity is a ferocious teacher.” - Michel de Montaigne

Acknowledgements

I would like to thank my family – Jim, Nancy and Emily – for their unconditional love and their unwavering support in helping me pursue my dreams. I would also like to thank my advisor, Christine Julien, and my reader, Joydeep Ghosh, for their contributions to this project. Finally, I would like to acknowledge the support of the late Tim Mauldin, a great friend and mentor who passed too soon.

Swapbeat.com: A Streaming, Adaptive Music Service in the Browser

by

Timothy Andrew Box, M.S.E.

The University of Texas at Austin, 2015

SUPERVISOR: Christine Julien

In recent years, ongoing shifts in the economics of the music industry have driven many artists to reevaluate how they intend to make money, with some artists now deciding that they will give their music away for free if it will increase their popularity and drive more fans to buy tickets to see their live shows.

Sites such as SoundCloud.com cater to this dynamic, allowing artists to host their music on the site for unlimited free streaming and download by fans. However, the sheer volume of music being uploaded to SoundCloud makes it impossible for even the most diehard music fans to keep tabs on what has been released and what is becoming popular with other listeners.

This paper therefore discusses the creation of swapbeat.com, a browser-based music streaming service. The site operates by first measuring how frequently new songs are being reviewed across 175+ popular music review sites on the Internet. The most frequently reviewed songs are then added to the swapbeat.com site, where score-based playlists of the songs are regenerated at frequent intervals, based on several explicit and implicit measures of sentiment gathered from users as

they listen to music on the site. Experiences and observations from the first two months of operation for swapbeat.com are then reviewed, and areas for further work are proposed.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Nearly 15 years after the courts declared the online music file-sharing service Napster illegal, the ethos of free (or very cheap), all-you-can-eat digital music that it unleashed continues to shape the music industry and bedevil those that seek to profit from it (Kaplan, 2001).

The most-recent attempts at monetization by the music industry have centered on partnering with digital music streaming services that provide unrestricted access to large catalogues of music for a monthly fee, a rate typically equal to around ten dollars per month. Companies such as Spotify and Pandora were among the first to demonstrate the potential of these streaming subscription services. Then, other major technology players such as Google and Amazon began offering their own branded streaming services as well. In the summer of 2015, one of the final holdouts, Apple, Inc., launched its own "Apple Music" subscription service, potentially cannibalizing its own sales in its iTunes store, in reaction to the growing popularity of these services with users.

Yet one statistic recently put forth in a L.A. Times article shows the continuing enormity of the problem of getting listeners to pay for music in any form: 20 million people used a file-sharing network to illegally download music for free in 2014, while only 7 million paid for a music subscription service (Faughnder, 2015).

These services, the record labels and the artists are therefore left to divide a shrinking pie of revenues from music sales and music subscription services, with

artists in particular often believing that they are unfairly or inadequately compensated for their creative work.

In this environment, an entire generation of music artists now expects and accepts that they will make little or no money on direct record sales. This is particularly true in specific genres of music where the concept of "sampling" audio from another artist's work will often increase the popularity of a song, while also raising the possibility that the artist using the sample may never secure the legal rights to include that track on an album that they can sell commercially. In these genres, albums are therefore often given away for free by artists, in the hopes of building a fan base that will pay to see the artist perform live shows.

One of the main benefactors of this new dynamic has been the music-sharing site "SoundCloud.com". Founded in 2007 by two music artists working in the "dance music" genre where sampling is most prevalent, the site has grown to become the premier location on the Internet for finding music from new and emerging artists. With 175 million unique monthly listeners, SoundCloud.com ranks as the 159th most popular site on the Internet (Alexa Global Site Rankings, 2015). Currently, SoundCloud generates revenue by charging artists a subscription fee to host a page where they can post their music, which can then be streamed and, at the artist's choosing, offered for download by users.

Each day, more than 150,000 new tracks are added to SoundCloud. Assuming an average runtime of approximately three and a half minutes per song, the equivalent of one year's worth of new music is now being uploaded to the site in each 24-hour period. Such a large volume of music is a daunting challenge for even

the most devoted music listener to try to keep track of. Therefore, many music fans now rely on a vast ecosystem of secondary music sites and blogs to help them identify new and emerging artists on SoundCloud.

This report outlines a series of interrelated coding efforts that were undertaken with the sole purpose of discovering the best new music being made available on SoundCloud, and making those discoveries quickly and conveniently available to users for streaming. To do so, the work of this project was divided into two primary efforts. First, a dedicated crawler was created to analyze the popularity of tracks on SoundCloud by evaluating which new tracks were being posted most frequently across music review sites around the Internet. Second, a dedicated site capable of streaming playback of these popular tracks by users was developed and made available to the public at www.swapbeat.com. The actions of users on this site as they listened to music and liked or disliked certain tracks were then recorded and, at frequent intervals, evaluated by automated scripts to periodically regenerate a new set of playlists that ordered tracks based on their popularity over a series of different time intervals.

The chapters that follow provide a detailed discussion of the initial creation of both the crawler and the application. This is then followed by a discussion of the positive and negative observations made from the initial launch of the application and its operation over the course of the first two months of its existence, a period running from early-September to late-October 2015. A series of potential improvements in response to these observations are then discussed, along with a roadmap for the site's further expansion going forward.

Chapter 2: Initial Design

Introduction

In order to detect what is popular across music review sites – and by extension, the Internet at large – this project created a targeted crawler that attempts to determine when the same song is posted across more than one of these sites as a proxy for that song's popularity. The approach behind this targeted webcrawler is discussed below.

Creating the Webcrawler

How the Targeted Crawler Works

Prior to creating the crawler, several months were spent evaluating which music sites on the Internet would be good candidates for inclusion in the crawl. The primary considerations for a site's inclusion in the crawl were:

1.) *Does the site post SoundCloud-specific links?*

Given that the application portion of the site would stream from SoundCloud, it was important that SoundCloud-specific links be available. This precluded a number of newer sites, which tend to post music videos from video-sharing sites like YouTube and VEVO. It also excluded tracks posted in track widgets available from Audiomack (Audiomack, n.d.), an emerging audio-sharing platform and competitor to SoundCloud, which focuses primarily on new or "leaked" hip-hop and rap releases.

2.) *How frequently is the site updated?*

Evidence that the site was posting more than once a week was initially considered an important criterion for inclusion in the crawl. However, once the actual technical solution for the crawl became apparent (particularly in the ability to "bypass" sites that have not been updated) this criterion became less important, and some of these sites were later added back in to the crawl.

3.) *What is the ratio of SoundCloud-linked posts to the overall number of posts?*

Posts are stored in the database for 30-60 days after they are crawled, even if they do not contain SoundCloud links (for reasons discussed later in this report). Therefore, a site that posts SoundCloud links, but in insufficient proportion to their other posts (common on general "entertainment" sites), was generally removed from the sample to reduce the raw number of posts that needed to be stored.

The initial list of sites for the project started at approximately 100, and has grown to over 175 as of this writing. As described below, the approach for locating songs within posts from a site does not rely on English-language processing or semantics. Therefore, numerous non-English speaking sites were able to be included in the crawl.

From SOLR to Default-Enabled RSS feeds in WordPress

When this project was first conceived, considerable time was spent researching how to implement a custom webcrawler using the open source crawling tools available from Apache under the SOLR project (SOLR, n.d.).

However, it quickly became apparent that many of the sites being evaluated for this project have been built using WordPress or similar platforms. By default, WordPress provides a PHP- and MySQL-based content management platform that has now become the de facto standard for the web, with "more than 23.3% of the top 10 million websites using WordPress as of January 2015." (WordPress, n.d.)

One very fortunate side effect of this de facto standardization is that WordPress enables RSS feeds by default when setting up a new site. This means that, for the majority of sites that accept/use the WordPress defaults at creation, an RSS feed will be available. In order to locate this default feed, all one generally needs to do is append "/feed" to the site's existing domain name, or one of several other similar variants.

Utilizing RSS feeds over a more general indexing tool like SOLR was preferable in a targeted crawl, such as the one required for this project, because the WordPress default RSS follows well-formed standards, providing dates of posting and consistent permalinks to the posts. Because so many of the sites were already using WordPress with RSS feeds enabled by default, the decision was made to develop an RSS-dependent crawl.

Therefore, one final question also became key for a site's inclusion in the crawl: Does the site have an identifiable RSS or Atom Feed?

The Simple Pie Library

Once the lists of RSS-specific URLs had been collected, several RSS reader libraries were evaluated for use in parsing the RSS feeds and locating the posts that would need to be crawled.

The Simple Pie library, written in PHP, was chosen for its ease of use and well-documented API (Simple Pie, n.d.). First made available in 2004, the Simple Pie library exposes easy-to-understand methods for handling the various blocks of a standard RSS feed. It also abstracts away the differences between the 8 common RSS formats and 2 common Atom formats to give you a consistent, fault-tolerant mechanism for retrieving RSS feeds. Furthermore, Simple Pie also makes extensive use of caching and hashing to lower its footprint on the sites it targets, and allows users to configure the cache as a directory in the local file system, or as a set of custom tables in a MySQL database with the SQL scripts they provide.

Simple Pie also exposes configurable timeout parameters for the local cache, which allows the user to set an amount of time for which a local copy of the RSS can be considered "fresh" before an update of the actual RSS must be retrieved from the site. Furthermore, Simple Pie will hash the RSS feed itself at each return visit to determine if any new posts have been added since the last visit to that specific RSS's URL.

Extending Simple Pie to Find Music

Once Simple Pie was chosen, it was incorporated into a larger script for identifying and recording SoundCloud links available in each post. The process for identifying SoundCloud links in each of the posts is described in the section below.

What is a SoundCloud Widget?

In order to help artists get discovered, SoundCloud makes available several mechanisms to allow site owners to host and play music on their site. By far, the most popular of these is the "SoundCloud Widget". The code to embed a SoundCloud widget can be easily copied via the "Embed" option available under every song found on SoundCloud. Music site owners therefore copy this code and embed the HTML directly into their reviews when discussing a track on their site. Once posted, the iframe renders from SoundCloud on the third-party site and users are able to listen to the tracks provided. The SoundCloud widget can be customized in several different ways to match the format of the site it is posted on; however, importantly, the SoundCloud widget always contains the unique ID of the track at some position within the iframe.

The Format of the Embeddable SoundCloud Widget Code

The following is an example of the SoundCloud Widget (iframe) code for the song "Love for That" by the artist Mura Masa. As the red bolded text demonstrates, the SoundCloud Widget always includes the track's SoundCloud unique id.

```
<iframe width="100%" height="166" scrolling="no" frameborder="no"
src="https://w.soundcloud.com/player/?url=https%3A//api.soundcloud.com/
tracks/228076072&amp;color=ff5500&amp;auto_play=false&amp;hide_relate
d=false&amp;show_comments=true&amp;show_user=true&amp;show_reposts
=false"></iframe>
```

Figure 1: The iframe for a SoundCloud Widget

An example of a rendered SoundCloud Widget iframe is shown below.

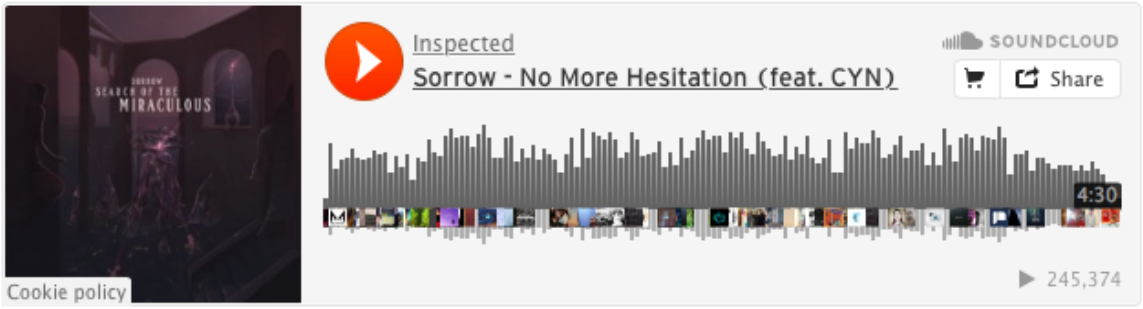

Figure 2: A Rendered SoundCloud Widget

Parsing posts for SoundCloud Widgets and Unique IDs

In initial tests, a regular expression pattern match was used to try to locate the unique ID of the track within the context of the iframe. However, this yielded poor results due to the varying parameters that can be included within the SoundCloud Widget.

Upon further evaluation though, it became apparent that a much simpler string matching approach might be possible. This approach set a series of possible pattern matches that accounted for different possible encoding standards for the widget above. The start position of the pattern match was then captured, the length of the pattern that matched added to it, and a substring taken at the next character for the nine-digit SoundCloud unique ID.

When the first pattern cannot be matched within the post, the script then iterates to the next possible match type and rescans the document. Although this can result in multiple scans of the post body, the linear nature of the string pattern matching using if-then has provided more clarity into how matches were found during testing and this approach continues to be used. Once a post has been parsed, the script then proceeds by examining which of the following scenarios is applicable.

Scenario 1: No Track Found

In the instance where no track is found, the post details are written to the "Posts" table in the database. This is done so that upon a return to the RSS where new posts have been added and the links in the RSS are to be re-crawled, the script can check the database to see if the link is to a "known" post, based on the permalink listed for that post. Once a post is "known", it will not be crawled again for as long as it remains in the database.

Scenario 2: New Track Found

If, during the scanning of a post, a new SoundCloud unique track id is found, a call is then made to the SoundCloud API with the unique id for that track. The SoundCloud API exposes a large quantity of data for each track. The following fields are of particular importance for the discussion of the application and the streaming of sound later in this report:

1.) SoundCloud User ID (for the original user posting the track, typically the artist)
2.) Song Title
3.) Number of Plays on SoundCloud (both on the SoundCloud site and in SoundCloud Widgets)
4.) Date Posted to SoundCloud
5.) URL Link to Album Cover Art in a variety of sizes (conveniently hosted via SoundCloud's Content Delivery Network)

Each of the data points above provide valuable information either for the display of the track in the application or for determining the popularity of the track when examining whether to add it to Swapbeat.

Once the call to SoundCloud returns, the details above are stored in the database in the "SC Details" table. Finally, a record is inserted in a join table between the Post and the SoundCloud details. The use of this join table is particularly important for the following scenario.

Scenario 3: Known Track Found

Given that this project seeks to identify popular tracks as they are posted across multiple sites, it is quite common that the result of parsing a post will be the discovery of a SoundCloud unique id that is already known to the database and for which details from SoundCloud have already been retrieved. In this scenario, the script will simply insert a record in the join table matching the new post with the existing SoundCloud details.

An overview of the crawler function is provided in the figure below.

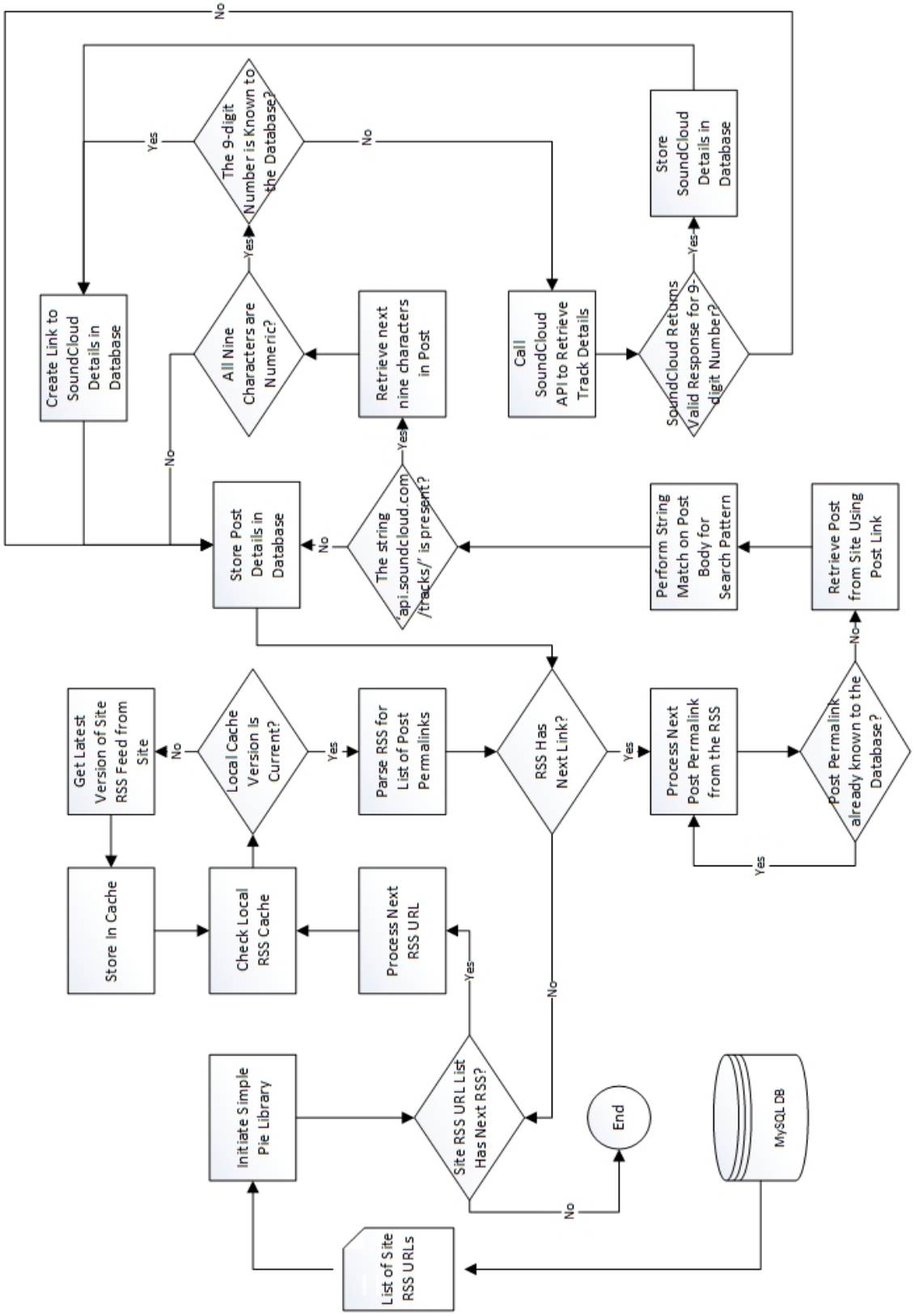

Figure 3: Overview of the Site Crawler Function

Performance and Potential Webcrawler Improvements

During the course of this project, approximately 25,000 posts from 175 sites have been crawled, yielding roughly 4,000 unique SoundCloud track ids. While the CPU time used by the crawler is not of great concern, the self-imposed waits of the crawler when fetching pages from sites means that on particularly high-volume days – typically Tuesdays, when most new music is released in the US – the crawler can take approximately two hours to complete.

Nevertheless, the performance of the webcrawler remains adequate for the volume of posts evaluated daily. However, due to the waits, further improvements to the crawler's performance might be advisable were the number of sites in the crawl to rise. The primary mechanism for performance improvement would be multithreading the crawler itself, although this is not without some potential obstacles, as discussed below.

Migrating the Crawler from Single-threaded to Multi-threaded

PHP has traditionally been a single-threaded language, with only unofficial support for multi-threading having been recently introduced. While these new multi-threading extensions do technically make it possible to introduce a multi-threaded structure for the crawler, the following points would also need to be addressed.

First, the current crawler self-imposes a random wait of between 3-5 seconds between fetches. This is done because site links are gathered one site after the other, so it is assumed that the next fetch is quite frequently from the same site, and that the crawler might be imposing a performance hit on that site were it to fetch one

site's posts/pages too aggressively. In a multi-threaded setup, this concern could be alleviated if all links from all sites were gathered in a first stage, with the links themselves being shuffled before being sent out to the individual threads for fetching in a second stage. In that way, the odds would be sufficiently lowered so that more aggressive fetching with less wait time could be imposed, based on the assumption that no one site is being targeted repeatedly.

However, a further consideration concerns the use of the SoundCloud API key required to fetch the SoundCloud details for a new track. While not official, SoundCloud does watch the usage of API keys for information retrieval, and can revoke privileges for keys calling too aggressively (generally perceived to be more than once per second). Several options, including the queuing/centralization of requests to SoundCloud, could allow for waits to be centrally monitored and controlled when gathering track details from SoundCloud.

Adding Tracks to Be Played

Once the crawl is complete, simple SQL queries embedded in PHP are used to sort the tracks according to how frequently they have been posted, and the PHP file then formulates the iframe for each track's related SoundCloud widget, listing the relevant details and the widgets in order in a single page, as shown below.

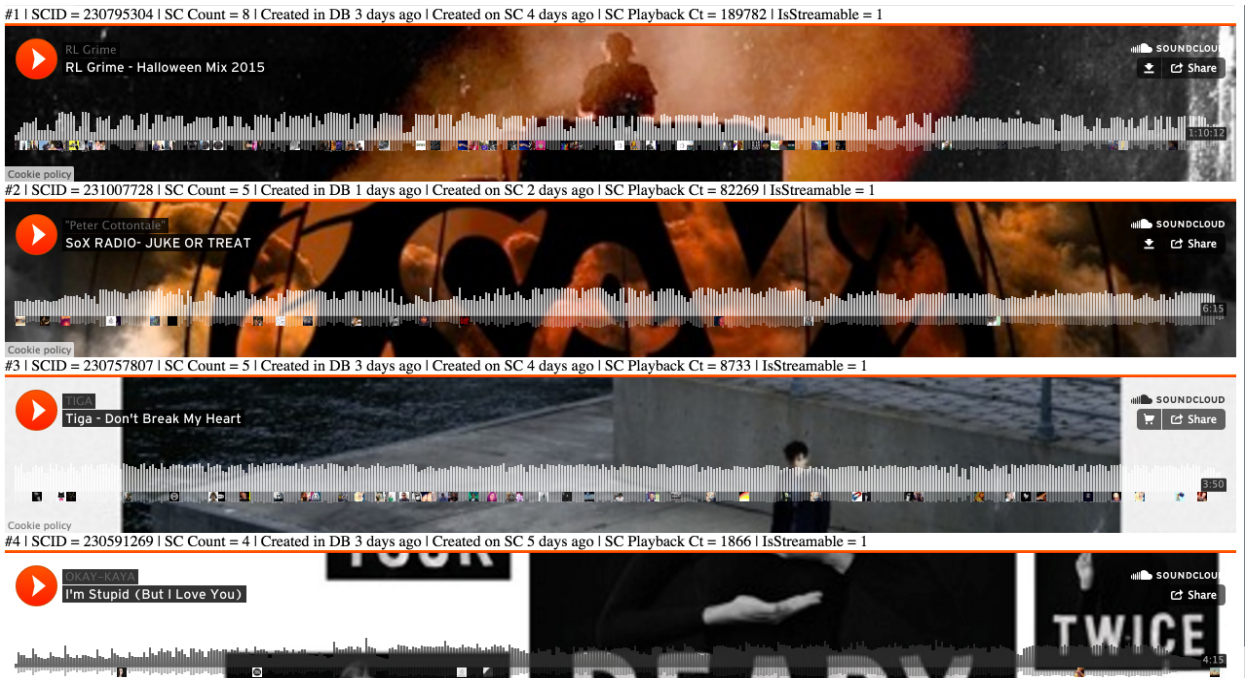

Figure 4: A Screenshot of SoundCloud Widgets Rendered at Crawler Completion

The tracks are then played in order to ensure that the recording quality is high, and that there are no unnecessary audio snippets, such as radio promotions or "dead air", to start the recording that would make it unsuitable for playback as part of a playlist. Some discretion is also exercised in regards to the lyrical content of the songs, as well as the duration.

Finally, some effort is made to determine if the user account that the song has been posted on is the true right's owner for the track. SoundCloud assiduously responds to "takedown requests" from copyright holders and will remove a track when notified that it has been posted illegally. When a track is removed from SoundCloud, any attempt to stream it via the API will result in an error. Swapbeat.com is coded to handle these errors, recognizing the condition and cascading to the next streaming endpoint to play the next song. Furthermore, the site is built to remove the track from all displays, including from users' favorites,

with the simple change of a Boolean value for “isVisible” listed for every track in the database. However, this still results in confusion for the user, so attempts are made to avoid this scenario altogether by validating the ownership of the track at the time of posting.

Creating the Application

Front-end

The Importance of User Experience

Because this project sought to gather information from real anonymous users, it was important to create a front-end application that would perform well, be easily and immediately understandable by a wide-array of users, and entice users to return for subsequent visits.

The following section details the approach taken to create the front-end necessary to provide a high-quality user experience. It starts by discussing several of the technical aspects for creating the single-page application (SPA) for Swapbeat in JavaScript, which was underpinned by client-side storage in the form of an “Application Data Map”, discussed in detail below.

Then follows a discussion of the various libraries needed to support consistent playback of audio across a wide array of potential browser and device types, primarily by using the SoundCloud SDK and the open-source Sound Manager 2 library, along with its “Bar UI” extension for controlling playback and audio.

Finally, the creation of User Accounts, utilizing integration with Facebook's JavaScript SDK for authentication, is discussed, followed by an overview of other key considerations for the layout of the application.

The following diagram shows the placement of many of these components within the three-tier architecture of the application, as well as the application's interaction with the results of the crawler function outlined in the previous section.

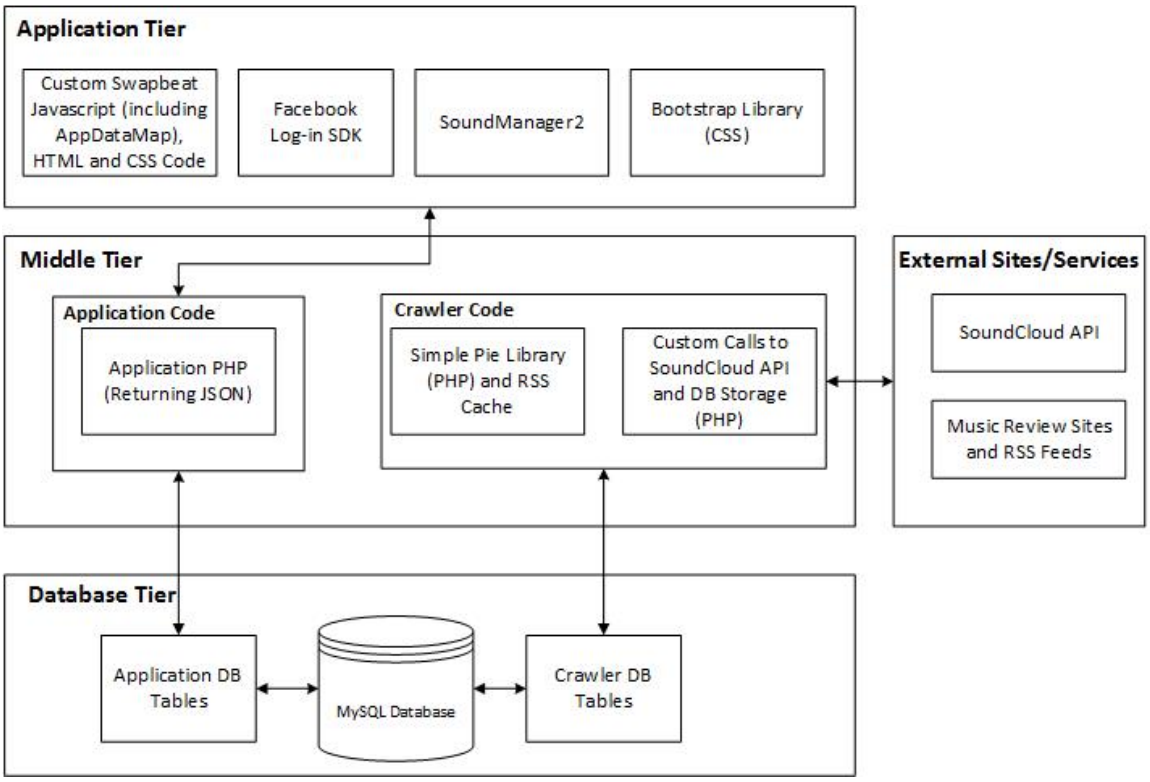

Figure 5: Three-Tiered Architecture Overview

Creating a Single Page Application and the Application Data Map

In recent years, JavaScript has continued to grow in prominence as the primary tool for creating rich user interfaces. Initially employed as an enhancement to otherwise static HTML pages, JavaScript is now frequently used as the primary

mechanism for application organization and display, with HTML frequently being relegated to the role of the "container" for the results of the executed JavaScript.

JavaScript's allure is further enhanced with the continued use of AJAX to asynchronously load data from remote servers, along with newer data delivery mechanisms arriving with HTML5, such as websockets, which can provide persistent, non-blocking communication with the server.

All of this is intended to provide the user with the quintessential "Web 2.0" user experience – typically defined as navigation within a website that does not require a full reload of a page each time information changes. Instead, Web 2.0 sites focus on only modifying the minimal portion of the page that needs updating, which is typically achieved by asynchronously loading data and updating the related portion of the page upon completion of the request.

In the scenario of a full, "single-page application", this means that only one Document Object Model (DOM) equivalent to the browser window is actually available to the developer, and libraries such as JavaScript and JQuery must be employed to traverse the DOM, locate the elements to be altered, and make the specific updates without a full refresh.

While this approach greatly enhances the end-user experience, it can create new difficulties for the developer. Now, instead of thinking in "web pages", developers must think in terms of DOM elements, and must find mechanisms to keep clear the state of the page and its subsections at varying points in time.

To help address this issue in this project, a single, globally available JavaScript class, referred to as the "Application Data Map" for the application, was

developed. Whenever the application is loaded, the first script to be executed is the "ApplicationDataMap.js" file.

Upon execution of this script, a series of asynchronous calls are made to initialize data for the application, utilizing JavaScript's self-invoking function syntax. When these calls are returned, the results are then stored as variables within the Application Data Map with corresponding getter and setter methods available.

The end result of the Application Data Map script execution is that an "AppDataMap" object is directly appended to the "window" object of the DOM. From this, all calls to retrieve or set data in the AppDataMap can be completed in JavaScript by locating the AppDataMap on the window object using the "window.AppDataMap.[function]" prefix.

Several objects are embedded within the AppDataMap to more clearly delineate functions and data into logical units. These embedded objects are described in the table below:

Embedded Object Name	Functional Description
AppDataMap.appConfig	Launches asynchronous calls to retrieve application configuration information from the Server.
AppDataMap.appDataCache	Client-side storage of tracklist datasets after they have been lazy-loaded from the server. Available methods calculate, set and evaluate timeouts for tracklists, and the cache is checked first before reloading a tracklist from the server.
AppDataMap.spinner	Tracks the status of the "spinner" icon shown when asynchronously loading data from the server.
AppDataMap.soundcloud	Stores the API Key for SoundCloud and tracks other SoundCloud-related details.
AppDataMap.user	Stores logged-in user information for event tracking, along with helper methods for translating Facebook log-in IDs to database user IDs.
AppDataMap.marquee	The marquee object tracks which message to display in the top navbar (yellow highlight). This object also includes "back" and "forward" message stacks enabling the app to retrieve and show the correct marquee message for the application at all times.
AppDataMap.currentTracklistInfo	An object with methods for tracking which song is currently playing.
AppDataMap.chainedPlays	A count of the total number of plays that have occurred without a user interaction. Implemented so that "Are you still listening?" checks can easily be added later on should streaming rate limits from SoundCloud become an issue.
AppDataMap.history	A custom object implemented to track the "pages" a user has visited within the application. Used particularly when a user makes a series of requests for tracks by particular artists and then browses back and forth between them.

Table 1: Objects Embedded in the AppDataMap

Caching for Performance

As mentioned in the previous table, the AppDataMap object contains an embedded object called appDataCache. The appDataCache initializes an empty array defined as tracklistDataSets when it is instantiated. This array is capable of holding multiple tracklist datasets as they are lazy-loaded and retrieved by the user (by clicking the available tabs in the application). Each tracklist dataset contains not only the information for the tracks in the list, but also defines a metadata set for the tracklist that includes variables for “tracklist type” and “timeout”.

Whenever the data for a tracklist is returned from the database, the result set is therefore wrapped in a tracklist dataset object, with a type listed and a timeout defined. This timeout amount is set in the database and retrieved at application launch as part of the calls made by appConfig.

The timeout duration is typically set at 15 minutes, which allows subsequent requests for that tracklist type to be satisfied directly on the client side without making a subsequent request to the database until after the timeout has passed.

At a summary level, this means that once all of the available tracklist types represented by the different tabs on the front-end have been requested and cached, no further loads of data are required until the timeouts are exceeded and it is time to reload the datasets (possibly because new tracks have been added in the 15 minute interval). In fact, the only communications between the client and server that occur are to register “user events”, which are discussed later in this section.

The benefits of this approach are twofold: Users see a near-instantaneous response when switching between already-cached playlists on the front-end, and the application server can support a far higher number of users than a non-cached version.

Special Case: Maintaining the Cache when a User Favorites a Track

While the process of tracking events and, specifically, user favorites is discussed in detail later in this paper, a particularly interesting set of use cases resulted from trying to maximize cache performance while also allowing users to "favorite" tracks from whichever tracklist they were reviewing.

Owing to the design of how tracklists are retrieved from the database, each track on each tracklist received from the database includes a column with a true/false value for "Is Favorite", representing whether the track is a favorite for that particular user.

While the status of a track as either a favorite or not is therefore correct when the tracklist is received from the database, once the tracklist dataset is cached and the user makes changes to their favorites, the client-side version of the tracklist, and specifically the "Is Favorite" value, risks being out of sync with what's been updated in the database.

Therefore, helper functions were added to the appDataCache object specifically to address this problem. In instances where a track is removed from being a favorite (from any tracklist), the removal event and the trackID are passed to a method on the appDataCache. From here, each tracklist dataset in the tracklistDataSets array is inspected individually to see if the track is present within

that dataset. If so, the status of the "Is Favorite" field is modified and the entire dataset is placed back in the cache, ready for the next retrieval.

More interestingly, in the event that a track is being added to the Favorites, in addition to the values previously mentioned, the track data itself is passed to the method. Similar to before, the corresponding values for the "Is Favorite" field are inspected within each of the datasets, except, in this instance, when the method finds that the dataset being inspected is the "My Favorites" dataset, it will also prepend the track data for the new favorite to the head of the tracklist data set, so that the track will appear at the top of the list of "My Favorites" when the user migrates to that tab, all without having to reload any of the tracklists prior to their timeout.

Playing Sound

The single most important function of the Swapbeat application is, of course, playing sound. From the beginning, it was clear that the application would target streaming music from SoundCloud using the streaming abilities SoundCloud makes available to its developer community. The single-most important of these is the "streaming endpoint" defined for each track by sending the trackID and your application's API key to the appropriate endpoint on the SoundCloud API. This returns a URL from which to stream the requested track.

As simple as this sounds, getting this audio to reliably stream across a wide variety of browsers is still a fairly daunting proposition. For starters, different browsers support only certain subsets of audio formats. And while newer browsers may be able to play the audio via new methods defined by the HTML5 standard,

older browsers that do not support HTML5 may still be best served by utilizing a Flash-based player, with the actual player rendered off-screen while audio is emitted.

Moreover, mobile browsers define different behaviors and standards when implementing music players that support tracklists. For instance, Mobile Safari typically requires a user action in its browser window before a sound can play. While this is fine for starting the initial song in a playlist, it will unnecessarily prevent the next track in a playlist from automatically playing if not properly addressed.

Sound Manager 2

Fortunately, an open source project called Sound Manager 2, maintained by Scott Schiller, bridges nearly all of these gaps for reliably playing audio in browsers (Schiller, Sound Manager 2, n.d.). Sound Manager 2 works by defining a "Sound Manager" object on the "window" of the DOM and then operating as a wrapper for controlling sound. The benefit of using Sound Manager 2 is that the complexity of "falling back" between HTML5 and Flash-based audio is completely abstracted away from the developer. At over 6,000 lines of JavaScript, Sound Manager 2 contains powerful code for detecting browser support for playing audio at launch, and accounts for numerous variations and quirks between browsers, such as by seamlessly "chaining" the next play event to the previous play event to address the Mobile Safari issue discussed above.

Developers are therefore able to call simplified methods, such as “play”, “pause” and “stop”, on the Sound Manager 2 object, and can trust that the correct, browser-specific version of those actions will be reliably completed.

In order to fully incorporate the Sound Manager 2 library into this project, only a handful of logging functions were inserted into the base Sound Manager2 code, primarily to allow for event tracking between play events when playing off a tracklist.

Sound Manager 2 Extensions: Bar UI

In addition to the base Sound Manager 2 API, Scott Schiller also makes available several JavaScript-based players for browsers that interact natively with the SoundManager object on the window, referred to as “Bar UI” players (Schiller, Bar UI Player, n.d.). One particular benefit of using these players is the integrated ability to use the “scroll” object to navigate the point in time at which to play the track.

JavaScript supporting the Bar UI defines functions for relating the width of the bar player to the length of the track, calculating the percentages for the placement of the slider on the UI bar in order to calculate the requisite percentage of the track from which to start playback. This “scanning” function is further made possible by the in-built buffering mechanisms of Sound Manager 2, which aggressively buffers the start of the track to begin playback before loading the remainder of the track at a less-aggressive pace.

An example of the Bar UI player in use on the site is shown below.

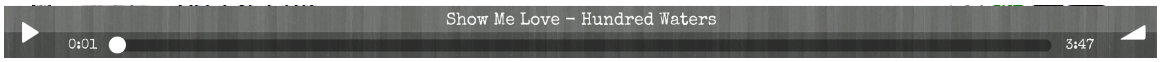

Figure 6: The Bar UI Player

Bringing it all Together

While the Sound Manager 2 API and Bar UI display element greatly simplify the technical complexities of playing sound in the browser, difficulties specific to the particular vision for the customized UI of the playlist for this site nevertheless arose.

Specifically, Sound Manager 2 and Bar UI define a set of HTML elements that are appended to the DOM with CSS classes added to display the player as a bar, which, in the case of this project, was placed in a fixed location at the bottom of the screen. Within this set of HTML, the Bar UI player defines specific HTML elements for appending audio resources to be queued for playing in order, constituting the player's tracklist.

However, because these elements are already styled as part of the Bar UI, and therefore could not also be displayed as part of the scrollable center portion of the application, a second "displayable" version of the tracklist had to be developed. Challenges then arose because both the "visible" and "actual" tracklist data now needed to be kept in sync.

For instance, when a user presses play for a given track, this track's audio is "selected" in the Bar UI player's tracklist. Visually, this must also be tracked in the "visible" playlist in the scrollable portion of the application. When a user navigates to a different playlist while the audio is playing, the visible playlist needs to represent the just-selected playlist info; however, the actual playlist still needs to

continue to play the currently playing song. Add to this the complication of what song should play if the user stays on the just-rendered playlist and the previous song ends. In this scenario, the most common expectation is that the song from the top of the second playlist should begin playing; however, this track data is not present in the "actual" playlist at this time.

Therefore, to get around these complications, a set of helper functions, previously alluded to in the AppDataMap were created. Their primary focus was to document the current tracklist being shown to the user and help compare it to the current audio sources queued in the "actual" tracklist. After some trial and error, it became possible to accurately track and compare the visual playlist against the actual playlist, using a series of JQuery-based checks. The results of these checks are then used to decide whether the existing data in the actual tracklist is still valid after a new tracklist has been loaded, or if the data from the tracklist currently being displayed in the scrollable portion of the app should be queued for play in the actual tracklist.

Rendering the Tracklist

Because the primary focus of the application is to display and play tracklists of songs varying by different time intervals, it became possible to create a single "view" file for rendering the tracklists. The workings of this file are discussed in detail in the following section.

Function of FormatReturnedTracks.js

After researching different aspects of DOM modification in a Single-Page Application, it became apparent that directly manipulating the DOM to remove and append individual tracks sequentially could result in sub-optimal performance.

Instead, the design leveraged the HTML "Document Fragment" approach, which allows a developer to create a document fragment separate from and unattached to the DOM's window object. This fragment can then be manipulated with all the normal DOM operations available in JavaScript and JQuery, and once finalized, can then be inserted into the DOM as a single element update operation.

The process of creating this document fragment, formatting and appending the child elements for each track, and then appending the fragment to the DOM was centralized in a single JavaScript file, called "Format Returned Tracks", which serves as the formatting mechanism for all the tracklists shown on the site. Centralizing these operations had the added benefit of greatly reducing testing efforts and allowed for rapid prototyping of designs during the early phases of this project.

The Format Returned Tracks script defines a for-each loop for every track in a tracklist data set sent to the function. Within this loop, the script creates and appends various document elements, classes, ids and onclick handlers to support the operations defined on tracks in the tracklist once displayed. An example of a formatted tracklist for the "Most Loved – Last 7 Days" playlist is shown below.

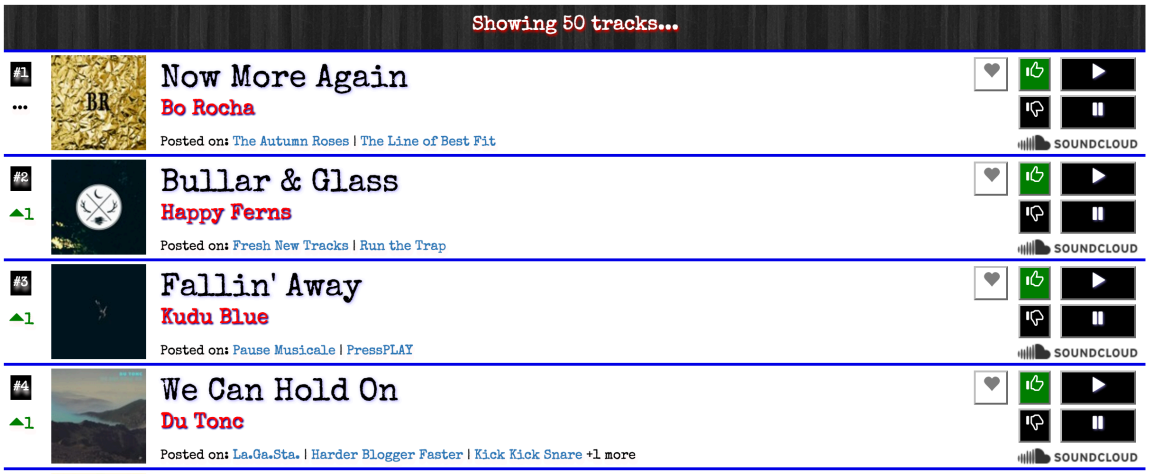

Figure 7: Screenshot of a formatted tracklist

The creation of each child element needed for the display of each individual track is described in the following section.

Track Elements Formatted

Track Position

The track position represents the relative location of the track within its tracklist. The number is created via a counter that is incremented after each track is formatted and added to the tracklist document fragment. The script also assigns a track position class to this element for CSS formatting.

Track Movement Indicator

The Track Movement Indicator is created by evaluating the concatenated string of prior tracklist positions captured in the Tracklist Audit Log discussed later in this document. By evaluating the most recent value from this string against the current tracklist position, the script can determine the absolute value of the movement and the direction. The script then applies Bootstrap glyphicons

representing “up”, “down”, or “no change” indicators, along with classes to define green, red, or black coloring depending on the direction of the movement.

Artwork Frame

As discussed in the crawler portion of this report, SoundCloud provides an artwork URL for each track added on SoundCloud. SoundCloud conveniently offers this cover artwork in several preformatted sizes, available by appending different size parameters within the URL. These sizes range from 16-pixel by 16-pixel thumbnails to 500-pixel by 500-pixel album cover sizes. Furthermore, this artwork is hosted via SoundCloud’s Content Delivery Network, which allows for rapid loading to the end user. For the current version, the 100-pixel by 100-pixel size was chosen as a good compromise of visibility and speed for the tracklist, although no noticeable decrease in speed was observed when loading the largest format sizes in later testing.

Track Title

The track title is directly taken from the “Title” field in the JSON. However, this portion of the JavaScript also includes a check to see if there is a “remix” artist on the track. If so, the track title portion of the script will pass the remixer name text to the Artist Name script for further formatting and then will add this name to the title.

For example, if the song “Divinity” is remixed by the group “Odesza”, the Track Title will pass “Odesza” to the artist name formatting function, and once returned, the formatted name for Odesza will be placed within the title to make the

text read, “Divinity (Odesza Remix)”, with the “Odesza” portion of the title selectable as further outlined in the Artist Name section below.

Artist Names

Because this project sought to make it simple to see all tracks by a given artist, onclick event handlers are assigned to every artist name displayed on the site, which, when clicked, return the set of all tracks by that artist in the database.

To make this possible, the required fields for the onclick handler “GetArtistTracks” are pre-registered using string concatenation during formatting. To simplify this process, a helper script called “Format Artist Name” was defined. This script serves two primary purposes.

First, when the function is called with an individual name and artist ID, it will format the Artist Name for display and will assign the onclick handler with the artist ID already present in the method signature to retrieve the artist’s tracks.

Its second function, however, is to allow for multiple artist names to be formatted and returned. In particular, the function can accept any number of artists as input and will return a properly separated list with an “and” inserted for two artists or commas and a concluding “and” for more than two artists.

The Format Artist Names function can therefore be called with all the names of the artists, and then can be called again with all the artists that are featured. These two sets of artists can then be concatenated with an appropriate “featuring” inserted between them. Several examples of this formatting are shown below.

Figure 8: Showing a tracklist for an individual artist's tracks

In the example above, the artist "Big Wild" appears as both a primary artist in the third track, and as a remix artist in the first two tracks. The example below shows a more complicated formatting case, where "Major Lazer" and "DJ Snake" are both listed as primary artists, "MO" is listed as the featured artist, and the producer "CRNKN" is shown as the remixer. Each of these artists' names can be clicked to retrieve a listing of all the tracks in the database for which they are listed as a contributor, as was done to capture the "Big Wild" screenshot above.

Figure 9: An example with several selectable artist names

Formatting Site Names

Because the site evaluates which tracks to add based on the postings of numerous music sites, it was important to show the posts themselves along with the track, both to assign appropriate credit to the sites that discovered the track, as well as to lend credibility to the decision to add the track to the site.

Furthermore, because the user interface of the site needed to be kept minimalistic to more closely mirror the display of a standard list of tracks, it was important to be able to show the individual links to the posts without cluttering the interface.

To do so, a helper script was written, called “Format Site Names”. This script takes as input a JSON object consisting of a list of sites, each with a permalink to the post where the track was found on that site.

For an individual site post, the Format Site Names script places the site name in the displayable text of the HTML element and assigns a “href” to the permalink for the post. The links are then pipe-separated and returned by Format Site Names for display.

Due to space constraints, a decision was made to only show a maximum of three site names at a time for each track. In instances where the number of sites is greater than three, the Format Site Names script selects a random value that is between zero and three less than the number of site posts passed in the JSON. The script uses this random number as an index into the JSON, taking a slice of the JSON for the next three site names. These sites are then formatted as described above. In addition, the script will then calculate the additional number of sites, concatenating

this to the end of the list of pipe-separated site names as "+ [Number of sites - 3] more".

An example of the site formatting for the track "Love For That", which was posted on 23 different sites, is shown below. Clicking on the links will take the user to the specific post on the named site where the track was found.

Figure 10: Showing the formatted list of site names where a track was posted

The Favorite Button

As described elsewhere in this report, emphasis was placed on allowing logged-in users the ability to favorite (or unfavorite) a track from any playlist they view. To do this, an "Is Favorite?" Boolean value is passed for every track on every playlist.

When evaluated by the Format Returned Tracks script, the favorites button, represented by a heart icon, is styled to represent whether the song is or is not currently a favorite. An onclick handler is also assigned that will update the favorite status from "favorite" to "not favorite" or vice versa, depending upon the current status when it is clicked.

In addition, to help with the client-side maintenance of the Favorites as described in the Application Data Map portion of this report, the element ID of each individual Favorite button is passed in to its onclick event handler, which allows the

handler to modify the display class for the Favorite button to its new status, either from favorite to not favorite, or from not favorite to favorite, along with updating the button's event handler to correctly handle the next possible favorite/not favorite event.

Like and Dislike Buttons

Simple "Like" and "Dislike" buttons were also added for each track in order to gather user sentiment. Unlike the favorite button, these buttons did not require the user to have an account or to be logged in. When selected, the onclick handler records information about the event and sends an asynchronous call to the database to record it. A temporary message is also posted on the marquee that reads, "Thanks for voting!"

Player Controls

The player controls portion of the Format Returned Tracks script is responsible for creating the "play" and "pause" buttons available for each track. The onclick event handler for the "pause" button assigns a simple pause method that requires minimal logic: any click on any pause button in the tracklist pauses the now playing track.

However, the pause functionality creates several potential scenarios to be considered when the "onclick" for the "play" button is invoked. Specifically, since a "play" button is available for each track, the handler needs to be aware that a track may:

1.) Not be playing,

2.) Be playing and be the track to which this play button is assigned,

3.) Be playing and not be the track to which this play button is assigned,

4.) Be paused and be the track to which this play button is assigned, or

5.) Be paused and not be the track to which this play button is assigned.

Therefore, the "play" click handler needs to be able to evaluate the information above and respond appropriately. In scenarios 1, 3, and 5, the handler should simply play the track immediately as a new track. For scenario 2, the handler should do nothing (and assume that it was a user mistake). For scenario 4, the handler should *resume* the track from the current position – i.e. if the track is paused at 30 seconds, it should resume playing at the 30-second mark. These scenarios are further complicated by the need to keep in sync the Bar UI player displayed at the bottom of the screen.

Therefore, the "play" click handler became a moderately complicated entity, requiring further tracking of the "play" state in the centralized "App Data Map" so that the correct decision on which track to play could be made consistently throughout the application.

Linking to SoundCloud Artist Pages

Finally, the SoundCloud terms of use require that applications that stream music from their service display any of several SoundCloud "marks" provided to developers. Furthermore, these terms also require that applications provide a link to the artist's individual SoundCloud page or to the specific page for the track.

In order to meet these requirements and to conserve space, the decision was made to format the SoundCloud mark as a skin for a button and assign the onclick event of that button to open a link to the artist's page on SoundCloud in a separate window, thereby meeting both requirements in one element.

Event Tracking and Updating the Marquee

Embedded in the event handlers for the "Favorite", "Like/Dislike" and "Play" Buttons are AJAX calls to the Application Server to register the event. These events are then recorded in the database and later used in the evaluations for determining and recreating the tracklists as described in the "Backend" portion of this report.

For events triggered by the user, such as "Favoriting" and "Liking" or "Disliking" a track, a simple "marquee" was created at the top of the page. Whenever one of these events was invoked, the handler would invoke an "Update Marquee" function. This function allows for the caller to also define a timeout. Therefore, when "Favoriting" the example track "No More Hesitation", the click handler can invoke "Update Marquee" with the message "Adding 'No More Hesitation' to Favorites", along with a timeout of 2 seconds. The Update Marquee function will then use JQuery to select the current Marquee text and store it, before modifying the Marquee with the temporary text. Once the timeout expires, Update Marquee will retrieve the previous message and will return it to the display in the Marquee. This allows feedback to be shown to the user without needing to clear pop-up messages or otherwise detracting from the user experience.

User Accounts with Facebook Login

The final key component for the site was the integration of Facebook authentication with the Swapbeat application. Facebook's authorization mechanism continues to rise in popularity due to the pervasive use of Facebook and the fact that Facebook tracking cookies are typically already present on a user's browser if they have logged into Facebook previously. Facebook's authentication mechanism for 3rd party applications primarily consists of allowing registered applications access to the cookie in order to determine who the user is and whether they are currently logged into Facebook.

Registering the Application with Facebook

The Facebook developer's page allows developers to register their application and receive an API key. Furthermore, a developer can register both a site URL and several applications all tied under the same API Key. This allows developers to provide a consistent user experience across multiple platforms and devices all without requiring additional logins for each device if the user is already logged into Facebook or a Facebook cookie is present.

Furthermore, the Facebook developer pages also allow developers to create test instances for their applications, along with sample users for testing.

Initializing the Facebook Authentication SDK

In order to initialize the Facebook Authentication in a JavaScript application, the developer only needs to include the self-executing initialization script in their home page. Developers are required to supply the version of the SDK they wish to

use along with their application's API key in the initial call to load the SDK. The script will then load the Facebook SDK in the DOM, which can be called to check whether the user is "known" and whether they are already logged in.

If so, the Facebook SDK returns that the user is logged in and provides the user's ID for this particular application.[1] An individual application can then typically compare this ID against their list of known IDs to find the user details and proceed with the login process.

Of particular note, the use of Facebook login obviates the developer's need to encrypt, store, manage, and reset users' passwords for their application. In fact, passwords are never seen by the 3rd-party application during the Facebook authentication process, only acknowledgement that the user is a registered user and is currently signed in.

Integrating with the Facebook Log-in Button

Facebook provides two primary mechanisms for completing the login process for both new and existing users that are not currently signed in to Facebook. First, the SDK exposes all the methods required to submit the login request to Facebook, along with providing success callbacks for when the login process completes. This allows for complete customization for the user interface during the login process; however, it requires the developer to observe the login status and provide appropriate visual cues as to whether the user is logged in.

[1] In an attempt to increase security, Facebook now assigns an app-specific user ID for each user for each application, and no longer directly transmits the user's actual Facebook user ID as it previously did. This, however, has exploded the size of ids needed and developers are now advised to store the "Facebook User ID" as an unsigned "BIGINT" value.

The second method involves using the preformed “Facebook login” button. This method provides far less UI customization but is much simpler to implement overall.

The Facebook login button operates by defining an onclick event handler of checkLoginState(). The basic format of this function is shown below:

```
function checkLoginState() {
        FB.getLoginStatus(function(response) {
                statusChangeCallback(response);
        });
}
```

Figure 11: Example code for the checkLoginState() method with callback

Once the call to getLoginStatus() completes, the “statusChangeCallback” is invoked. It is within this callback that developers can include their application-specific login functions, which often involve querying the Facebook Open Graph API for the logged-in user’s details.

To demonstrate this functionality, this project included calls to the Open Graph API to get the user’s name and the user’s profile pic, both available without requesting additional user permissions when they register.

One example of these calls is provided below. Here the Open Graph API is called with the API to get the ‘me’ (current) user’s profile picture and place it in a DIV element on the page as shown below. (Note: the “response.id” is the application-specific user ID for the logged-in user.)

```
FB.api("me", function(response) {
        if(response && !response.error) {
                var picDiv = document.getElementById('userPic');
                        if (picDiv.innerHTML =="") { //prevent loading duplicate images
                        myImage = document.createElement("img")
                        myImage.src =
                "https://graph.facebook.com/"+response.id+"/picture?type=small";
        }
}
```

Figure 12: An example of a call to the Facebook Open Graph API

Middleware

Owing to the heavy reliance on the Single-Page Application for formatting responses into the appropriate pages for viewing, the primary role of the middle layer for this application became the retrieval and return of JSON between the client and the database.

Because of this, only five PHP files, constituting a mere 600 lines of code, were required to support the communication between the client and database. An overview of these six files and their role is given below:

File name	Purpose
GetTracklists.php	Retrieval and return of tracklists from the database based on the time interval selected.
InitAppConfig.php	Invoked by the App Data Map, this retrieves configuration parameters, such as timeout intervals and API keys, from the database at startup.
ManageLogin.php	This is invoked during the log-in process to register both new user log-ins and to update the "last log-in" of existing users.
RegisterEvent.php	Accepts metadata regarding the different tracked user events, such as "like/dislike", "favorite", and "play", and registers these events in the database.
GetTracksByArtistID.php	Invoked when a user requests the application display all the tracks for an artist. Accepts the artist ID as a search parameter.

Table 2: PHP files supporting client communication with the MySQL database

Each of these files use PHP's "MYSQLI" extension to manage communication between PHP and the MySQL database instance described in the following section. The MYSQLI extension primarily implements an interface for creating a database connection, executing a statement, and closing the connection when complete.

In each of these files, once the results array is returned to PHP, the json_encode() function is called on the array and the results are "echoed" back to the client, as shown in the example below.

```
$jsonData = array();
while($array = $result->fetch_array(MYSQLI_ASSOC)) {
        $jsonData[] = $array;
}

echo json_encode($jsonData);
```

Figure 13: An example using the MYSQLi extension in PHP

Backend

For the backend datastore, a MySQL database was created using MySQL Workbench's data modeler. The original schema consisted of 42 tables, 6 views, and 8 stored procedures. Five of these tables specifically relate to the "crawler" function described previously, while the remaining tables support the storage, retrieval, and playback of music in the online application.

Throughout the database, a focus was maintained on using many-to-many join tables, as, for example, artists frequently have many tracks and tracks frequently have many artists.

In addition to standard data modeling, indexing and normalization considerations, several modeling choices specific to the support of an online music application were made for the creation of the site, which are discussed below.

Specific Design Choices for a Music Site

As mentioned previously, an early design choice for Swapbeat included minimizing the performance requirements on the middle layer created in PHP. One mechanism for doing so included the preprocessing and storage of application-ready data directly in the database. This took two primary forms: one, storing the

data from multiple corresponding rows as a single, preformatted JSON string; and, two, preprocessing the results for the different "Most Loved" playlists and storing them in tables for quick retrieval. Each of these enhancements is discussed below.

Storing JSON in the Database

As JSON has continued to rise in prominence for communicating data from servers to clients, new "page-based" data stores such as MongoDB have been developed to directly store JSON and more rapidly serve this information to front-end clients at the expense of relaxing some of the "ACID" properties of more traditional databases.

While this project used a MySQL database and did not implement a newer page-based storage solution, it quickly became apparent that some direct storage of pre-formatted JSON would be useful in the "high-volume" tables returning the different "Most Loved" playlists. This was particularly true of several different data points for which multiple rows relating to a given track might exist in another table. For example, multiple individual artists' names may be present for a track, and multiple posts related to a track may need to be returned for display in the front end. In order to properly format this data in JSON using SQL, extensive use was made of the "concat" and "group_concat" functions available in the MySQL variant of SQL.

For example, the following SQL will look for all the artists associated with a given track, and will "group_concat" (i.e. group concatenate) their Artist ID and their Artist Name, into properly formatted JSON. The concat function will then repeat this for any additional artist names, with properly inserted commas and parentheses

added between each individual group_concat result, maintaining proper JSON formatting.

```
(select trackid, concat(\'[\',group_concat(distinct \'{\"artistID\":\"\', 
at.artistID, \'\",\"artistname\":\"\' , a.artistname, \'\"}\' order by at.artistID 
separator \',\'),\']\') as artists
from artiststracks at, artists a
where a.artistID = at.artistID
and at.isartist = 1
group by trackid) as artistsTemp
```

Figure 14: Example SQL for CONCAT and GROUP_CONCAT

Results from this query can then be selected for and stored in the "artists" column for a track as the following properly formatted JSON (w/ escape slashes for quotes):

```
[{\"artistID\":\"16\",\"artistname\":\"The Chainsmokers\"}]
```

Figure 15: Formatted JSON stored in the database

As mentioned above, if there are multiple artists, such as in the example of two "featured" artists below, these will be added together by the outer concat function to render the following properly formatted JSON, which can then be stored in the "features" column for the track:

```
[{"artistID":"462","artistname":"SiR"},{"artistID":"463","artistname":"Joyce Wrice"}]
```

Figure 16: Formatted JSON for multiple "featured" artists

In addition to formatting artists' names, this approach was also used for site names, where, for several particularly popular tracks, 20 or more site posts related to a single track have been concatenated into a single "Post Info" column for

retrieval. Interestingly, occurrences such as these exceeded MySQL's default setting for the maximum character count allowed when performing a group_concat, requiring this setting to be manually increased in the MySQL instance to 15,000 characters at one point in the project.

Preprocessing the "Most Popular" playlists by Time Period

One of the main areas where performance improvements were sought was in pre-processing the playlist data for the "most popular" playlists for each of the time intervals available to users. Preformatting and storing JSON in the database was particularly important to this approach, as it meant the tracklist could be served to the client immediately, without computationally intensive queries being performed for each new request to the database. The finished results of these queries were therefore processed once per interval and then stored in individual tables representing each playlist. These playlist tables were then dropped and recreated every 12 hours to capture the updated results.

To assist in this process, stored procedures were developed to analyze the available data for each of the time periods required.[2] At runtime, the results of these stored procedures are written to temporary tables, the main tables dropped, and the main tables recreated using "select from" syntax on the temporary tables. Once the main tables have been successfully recreated, the temporary tables are dropped.

Allowing all of the processing to complete before the "main" table is dropped and recreated, when combined with the caching mechanism for each of the playlists

[2] The stored procedure approach was necessary because MySQL does not natively support materialized views.

on the client-side, greatly reduces the chances that a user will request a playlist table that is unavailable in the database in the specific moment when it is being dropped and recreated.

Scoring Tracks for Playlist Inclusion

In addition to the mechanics of dropping and recreating these tables, these stored procedures also contained involved logic for determining how to order the tracks into the playlist for the period.

One specific consideration for creating the updated playlist for each interval was recording different forms of sentiment towards tracks as they existed on the site. The primary mechanisms for registering sentiment for a given track included "plays", "likes" and "dislikes" registered for the track, as well as the number of times the track was "favorited", and the number of times a "buy link", when present, was clicked.[3]

Each of these four event types is registered in the database whenever they occur. Specifically, for each event, the event type, the user ID for the event (if present), the track ID, the tracklist position, the tracklist type and the timestamp for the event are recorded.

Events Weighted by Type

In addition to the raw scoring based on the number of times each event type occurs, two measures were also applied to each event. The first is a simple filter, which applies a fractional proportion to each event type based on its perceived

[3] Buy links were included in the early design of the application, but were removed for the initial go-live phase of the project on the swapbeat.com domain.

relative importance and informational value in comparison to the other event types. During the course of this project, the multipliers applied to each event type were as follows:

Event Type	**Relative Weighting**
"Play" Event Registered	0.1
"Like" or "Dislike" Registered	0.4
"Favorite" Registered	0.6
"Buy Link" Click Registered	0.6

Table 3: Event Types and Relative Weightings

Each of these weightings were set in a parameters table in the DB, allowing them to be read at runtime by the stored procedure, and therefore changed dynamically throughout the course of the project. It is important to note that these relative weightings also incorporate the current use pattern information from the site. Thus, were play events on the site to rise dramatically, these weightings would need to be readjusted to keep the different events in relative balance.

In addition to the simple filtering done above, two additional forms of weighting are applied to each event type to further utilize the information received as a result of each event.

Events Weighted by Tracklist Position

The first of these two weightings applies to the position of the track on the tracklist in which it was presented when the event occurred. A tracklist can consist of up to 50 songs, laid out vertically on the page. Because users are consistently presented with the top portion of the playlists when it is first loaded, and thus shown tracks that are, in theory, already popular for that time period, additional

events registered on these "top" tracks have a lesser informational value than events registered on tracks lower down the playlist.

To capture the information difference between these two types, an inverse decay function calculates an additional score for each event type, based on the tracklist position when the event occurred, so that, for instance, when a track from lower down the playlist is played, liked, etc., its score is "boosted" more than when a track already at the top of a playlist is played, liked, etc. One special case for these weightings is also applied in regard to "dislikes". In this case, when a track at the top of a given tracklist is "disliked", this event is weighted more strongly (a more negative scoring impact) than when a track already positioned lower down the list is disliked. These differences in weightings are applied non-linearly using exponential functions. Generally, the differential applied between "infrequent" events, such as liking a track in position 50 versus liking a track in position 1, were set at 4x, resulting in dynamic track movement without overpowering other contributing factors.

Events Weighted by Time

A similar weighting mechanism is also applied to the timestamp of each event. Here, the theory is that events registered more recently have higher informational value than events registered further in the past. In order to apply this weighting, the events are grouped into the intervals between regeneration of the playlists. A weighting is then applied that is inversely proportional to the number of periods that have occurred since the event was registered. For instance, an event

occurring one period ago is given a higher time-weighted value than an event happening five periods ago.

Calculating the Final Score

The relative importance of these two weighting measures is demonstrated in the graph below. In this graph, the "points" assigned to an event for the weighting type are represented by the y-axis, and both the tracklist position and the number of time intervals that have passed are represented on the x-axis.

Here, the blue line shows that when a track is in first position and "liked", it will receive approximately one weighted point; however, a track in 50th position that is "liked" will receive nearly 4.5 points.

Similarly, an event occurring only one period ago will receive approximately one weighted point for its time component, while an event occurring 50 periods ago will receive approximately .25 of a point for its time component.

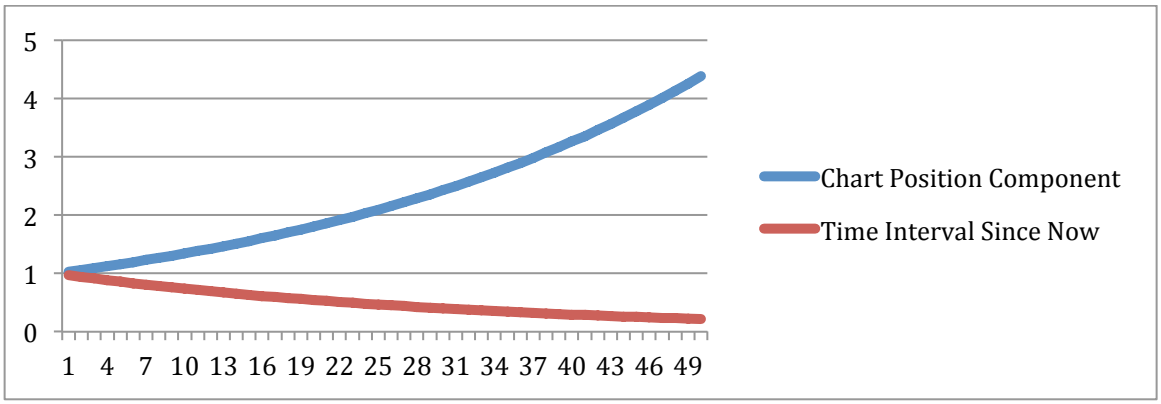

Figure 17: Calculating the Chart Position and Time Component Scores

Once the different weighting components for each event have been calculated, the points available for each track according to each of the weighting

calculations are summarized in a series of temporary tables. The stored procedure then completes several "union" queries to summarize the results from the different temporary tables, and then orders the tracks by their cumulative adjusted scores for inclusion in the playlist.

The Tracklist Audit Log

One final component of the stored procedure for updating and recreating the "Most Loved" playlists is the storage of all tracklists, tracks, and positions in a single database table called the Tracklist Audit Log. This table allows for the detailed analysis of how a track progresses through the application as it moves higher and lower across playlists through time, since the information for each track is stored for every track on every playlist at the generation of each new set of playlists.

In addition to analysis, this table allows for a detailed listing of the last 10 (or more) positions of a given track on a given tracklist type to be concatenated and stored in a single column with each track. Initially, this time-series information for the movement of the track within the playlist was displayed as a graph using Chart JS (Chart JS, n.d.). However, this visualization was removed during the course of the project in favor of a simpler display of the up/down green and red colored arrows displayed alongside the number of positions moved.

Other Database Design Considerations

Returning Tracklists and User Favorites

Another key design decision from the application was that users who have created accounts should be able to favorite any track they see on the site and have it

show in their “My Favorites” tab immediately. Furthermore, whenever a user views a track they have already favorited, they should be able to see that it is a favorite directly in the display for the track wherever it appears.

The mechanism for updating the favorite from the front-end application, including its interaction with the client-side caching mechanism, was discussed earlier in this report. However, when first returning a playlist from the database to the client, it was important to also identify which tracks on a given “popular” playlist have already been favorited by a given user.

To do this, a stored procedure for returning tracklists to the client was implemented. This “Get Tracklist” procedure takes as input the tracklist type to return, along with the User ID. Then, using a case statement to match the tracklist type to return, the procedure determines if any of the tracks returned for the given playlist are user favorites by performing a “left outer join” between the tracklist table and “user favorites” table based on the track ID. When a match is found, a “true” value is registered in the “Is Favorite” column for that track prior to returning the tracklist back to the user.

This allows for the “Heart” icons, representing the user’s favorites to be quickly and correctly displayed at the initial display of each tracklist in the application without having to make a separate round trip to the database to gather the user’s favorites and compare them on the client side.

Retrieving Tracks by Artist ID

One of the key challenges in modeling independent musical data correctly is acknowledging that a single musical artist may be the primary artistic talent, a

featured artists or the remixer for a given track. This is particularly true today of newer artists, who will typically attempt numerous remixes and collaborations in order to get discovered.

One of the early design goals of the Swapbeat front-end application was to have all artists' names be selectable throughout the front-end, so that clicking on an artist's name would retrieve all tracks that that artist has contributed to, regardless of the type of contribution.

Therefore, it was important to capture not only the link between an artist and a track but also the nature of that relationship as well. Thus, in the data model, a join table between artists and tracks was created, called "artists_tracks", which also contained columns to identify "isArtist", "isFeature", and "isRemixer". This structure opens up the possibility that users can therefore click on the artist name of an artist in a track, and from that, a query can be run in the "Artists Tracks" table to locate all tracks where that Artist ID is present (regardless of their role in the track). This list can then be returned to the front-end for all of these tracks.

Infrastructure and Deployment

Because this application sought to replicate a commercial-grade music playing experience for users, sufficient infrastructure was required to ensure smooth, high-availability operation of the site, while also minimizing the complexities of administering servers for high-availability and failover, completing patches, etc.

After careful consideration, the Heroku "platform-as-a-service" (PaaS) was enlisted for serving the application, and an Amazon Webservices MySQL instance

was chosen for the database. A short discussion of the experiences deploying and managing the application with these two services is given in the sections that follow.

Deploying the Application to Heroku

As mentioned previously, Heroku markets itself as a "platform-as-a-service" (PaaS) provider. Heroku's product offering differs from that of a traditional cloud hardware provider due to Heroku's concept of a "dyno". In Heroku's parlance, a dyno provides both the virtual resources of a traditional cloud/virtual container combined with the specific operating environment defined by the application. Because of this standardization to a set number of supported dyno types, Heroku allows applications to be scaled seamlessly across multiple dynos when needed (Heroku, n.d.).

Heroku also provides several different simplified mechanisms for deploying applications. By far, the most popular of these options is the Heroku command line interface (CLI), which integrates with the user's Git repository for their project.

Specifically, the Heroku CLI allows the user to associate their project with a remote Git repository hosted on Heroku. Once this remote repository is created and assigned, the developer can simply navigate to their root directory in the terminal window, complete their Git "add" and "commit" operations for any new or modified files, and then execute the statement "git push heroku master" to have their application code pushed to the remote repository and automatically deployed to the related dynos.

Once complete, the application is then available at a subdomain of "herokuapp.com", with the sub-domain name either randomly assigned or specified

by the user. When using a custom domain name with Heroku, such as the one purchased for this project, www.swapbeat.com, one only needs to create a CNAME reference from the custom domain name to the herokuapp subdomain.

Interestingly, Heroku's platform is itself actually hosted on Amazon Webservices (AWS) in the Northeast region. As discussed in the following section, the MySQL database for this project was deployed directly to an AWS MySQL instance. Because the MySQL instance was also in the Northeast region, the project did not have to pay data transfer fees for communication between the application server and the database that might have been incurred had the application not been hosted on AWS or even hosted in a different region within AWS. And while these costs were not an important consideration during the early stages of this project, were the user base for the application to increase dramatically, the avoidance of these costs could become much more important in the future.

Deploying the DB to AWS

In addition to deployment of the application to Heroku, as mentioned, a MySQL instance was also created within Amazon Webservices (AWS RDS, n.d.). AWS provides numerous compute and memory size options. To balance cost vs. performance considerations, a db.m3.medium AWS RDS MySQL instance was selected for this project. This provides 1 vCPU and approximately 3.75 GB of memory, which has proven more than adequate for the initial stages of this project. Once instantiated, AWS RDS MySQL instances are fully accessible using the standard MySQL connection mechanisms.

Interestingly, Amazon has recently launched their proprietary "Aurora" version of MySQL. Aurora is marketed as a fully compatible version of MySQL with the added improvement that the backup and replication routines have been rewritten and optimized to fully take advantage of AWS's vast storage infrastructure, which allows AWS to promise near-zero latency replication between read-write and read-only copies (AWS Aurora, n.d.). This makes possible many interesting architecture options for developing read and write strategies for MySQL-based applications; however, Amazon currently only offers Aurora-based MySQL instances at resource sizes and price points far above what was required for this project.

First Release

The Swapbeat application – available at www.swapbeat.com – was released on August 28th, 2015. Prior to this, several earlier releases were available at specific Heroku subdomains and were shared to get feedback, which was incorporated to the extent possible into the version launched at swapbeat.com.

While the author was able to share the site's development with friends and family, the project also depended in part on usage of the site by anonymous users on the Internet. Therefore, to increase this type of traffic for the site, several efforts were undertaken, as discussed below.

Generating Traffic

Social Media

In order to try to increase site traffic, a Twitter account was created for the site: "@swapbeatmusic". Throughout the course of the two months of the project, frequent tweets were sent to artists' Twitter accounts informing them that their song had been posted. While these tweets often garnered a response from the artist in the form of a "favorite" or a "retweet", these efforts had minimal impact on driving traffic to the site, as Twitter shares analytics for tweets with the author, including the number of clicks on site links embedded in the tweet. Furthermore, all links shared in tweets are minimized using Twitter's proprietary link shortener. Therefore, all link clicks from links shared in Twitter can be seen in Google Analytics as originating from the "t.co" domain. Very few "t.co" domain origination visits were observed in Google Analytics during the first two months.

Interestingly, the only tweet that garnered significant attention was a tweet sent during the Austin City Limits festival of a popular artist named "GRiZ" who was performing. This tweet was "retweeted" by the official Austin City Limits festival twitter account. This resulted in 12,000 views of the tweet in the course of 24 hours and over 14,000 views as of the time of this writing. However, even this tweet only garnered minimal engagement with the actual swapbeat.com site, as shown in the statistics from Twitter below.

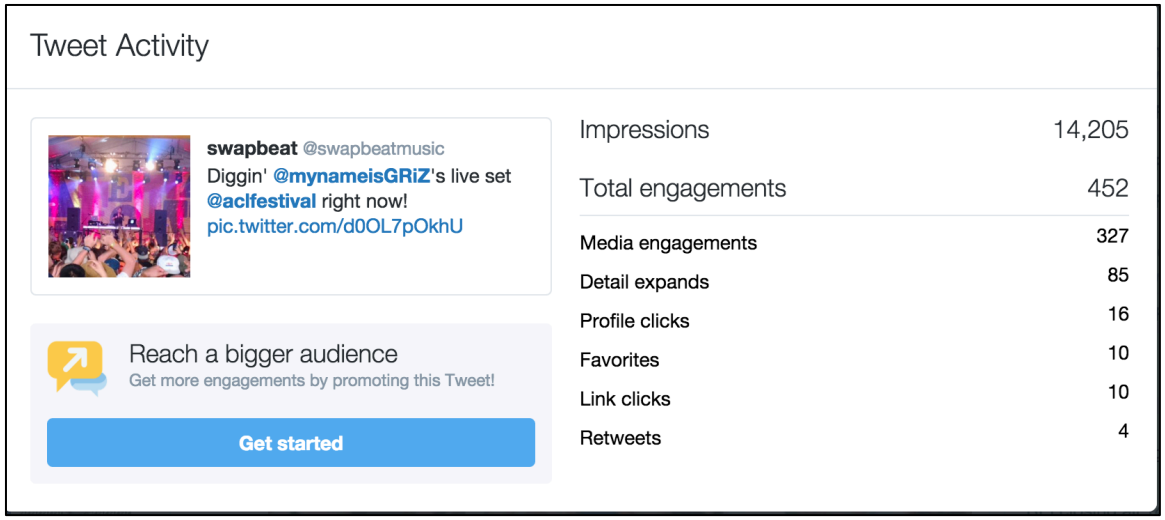

Figure 18: Metrics for a popular tweet sent from the project's Twitter account

Google Adwords

In addition to utilizing social media, a Google Adwords campaign of a few dollars a day was utilized during the course of this project. Keyword selections for "new music", "indie music" along with several unique artist names were used to drive traffic to the site. This keywords campaign was successful in bringing between 5 and 10 additional users to the site each day, with the obvious implication that spending more money would have driven higher number of users, even though the campaign's average "click rate" of 1% was low compared to the common target of approximately 2% (Measuring Traffic To Your Website, n.d.).

Of note, as much as 98% of the traffic sent to the site by the Adwords campaign originated from mobile devices. This is in part due to the keywords selected, particularly of the specific artist names, as well as the increasing migration to mobile for many younger users.

The Adwords campaign was initially targeted to run only for desktop devices. However, on several days, the campaign failed to reach its budgeted maximum, indicating that far fewer searches for the targeted terms were originating from desktop than they were from mobile. Therefore, the campaign was changed to run across all devices in order to maximize exposure for the remainder of the project.

Observations from the First Release

The following sections cover observations gathered during the first two months of operation of the site, along with what improvements would need to be implemented to address several of the problematic observations stemming from the current site's design.

Issues for Search Engine crawls

The primary issues for the first iteration of the site related to the site's performance during web crawls for indexing by popular search engines like Google and Bing. The primary issues are covered in the section below.

301 and 302 Redirects

In an attempt to help improve the visibility of the site and to attach a moniker people would be able to remember, the domain name www.swapbeat.com was purchased for this project. The purchase resulted in a formal transfer from the original owner to the author, at which point the domain name was locked from further transfer for 60 days, per ICAAN's rules.

Unfortunately, the domain registration service used by the original owner to register www.swapbeat.com did not support the more search-friendly domain name

assignment mechanism known as CNAME referencing. Instead, the domain registrar only allowed for 301 and 302 redirects to be used. The unfortunate side effect of this limitation was that search engine crawlers tend to ignore domains that use 301 and 302 redirects. This is reasonable from the perspective of crawlers because they believe that they will have already indexed the site at the redirect elsewhere during the course of their crawl. Furthermore, a 301 would usually (but not always) be used to redirect from a "lower" importance domain to a "higher" importance domain.

This was, however, the opposite situation of what was sought for this project, where the top-level swapbeat.com domain was meant to direct to the application's subdomain on the herokuapp.com domain/platform. Because of this, crawlers appeared to ignore the site for the period under which the site was forced to use a 301 redirect. Once the requisite period passed and a transfer to a registrant allowing CNAME referencing from swapbeat.com was established, this was no longer the case, and the site content was cached by the Google crawler.

High Reliance on Custom JavaScript and AJAX

In addition to the technical issues caused by the forced use of 301 and 302, the site's heavy reliance on custom JavaScript and AJAX also proved to be detrimental to the site's search engine crawl performance.

Specifically, Google's crawler appeared to be unable to recognize/execute the initialization script that loads tracks when a user first logs into the site. This resulted in a Google "cache" view, shown in the screenshot below, after the first round of crawls by Google.

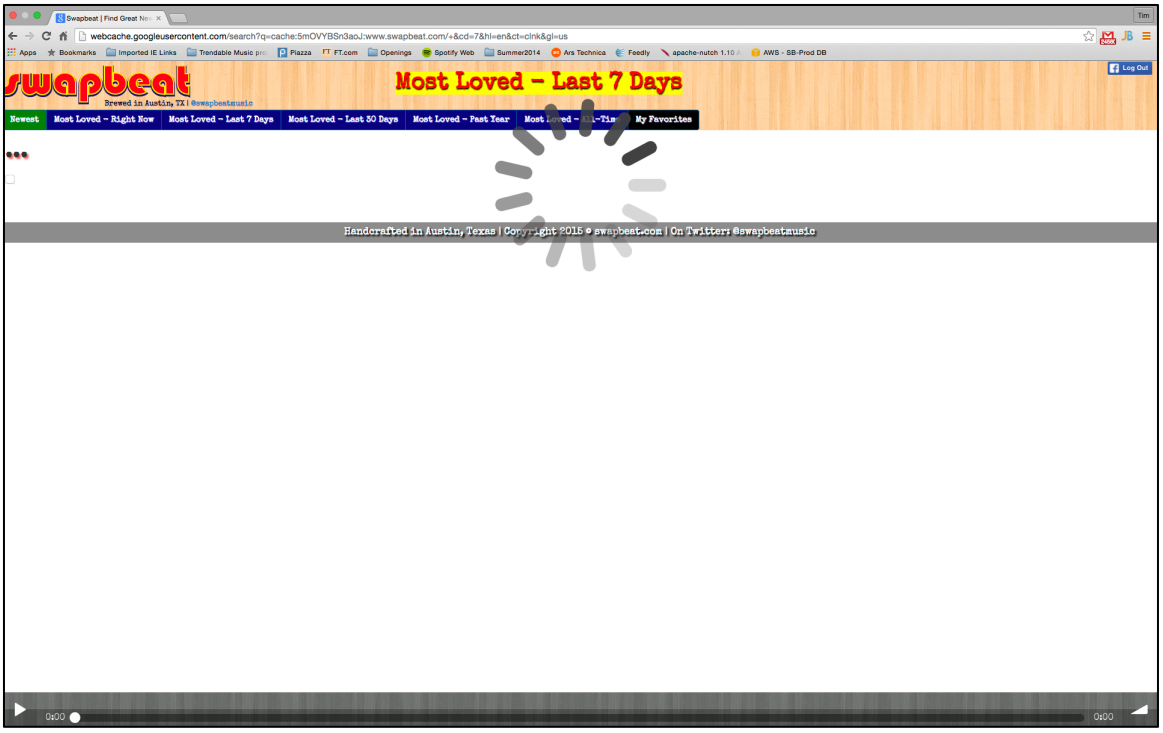

Figure 19: The initial Google cache view for the swapbeat.com site

Not Mobile-Friendly

In April of 2015, Google announced that a significant factor in the weighting of a site's prominence in search results is Google's determination of whether the site is "mobile-friendly" (Makino, Jung, & Phan, n.d.).

Due to the limitations of the crawler to execute the JavaScript for the site during its initial crawls, the swapbeat.com site did not receive a mobile-friendly designation, even though specific mobile-friendly stylesheets, based on Twitter Bootstrap 3, were applied to the site to ensure relative spacing and visibility across devices and screen sizes.

Google is thought to use the "mobile-friendliness" of a site even more prominently when serving searches originating from mobile devices. Based on the results of the Adwords searches in which upwards of 95% of Adword referrals

originated from mobile, it is highly-likely that a significant proportion of organic search traffic is not currently being referred to swapbeat.com at the rate it could be were a “mobile-friendly” designation received.

No Direct and/or Deep Linking Structure

In the first iteration of the application, significant time was expended ensuring a fluid user experience with minimal waits. Moreover, the decision to create a single-page application (SPA) enabled a user to navigate through the site while maintaining a consistent header at the top of the page, and a player element capable of continuing to stream sounds without interruption, even as “pages” of playlists were browsed and navigated between.

However, when creating this user experience and the custom navigation to move between tracklists, the URL structure, and specifically, facilities to allow direct URL navigation to a specific playlist, artist or track were not included. Essentially then, the only URL available for the application was the top-level “swapbeat.com”. While this was tolerable for a proof of concept, it proved highly detrimental in two areas: search engine crawling and social media sharing.

In regards to search engine crawling, the lack of a deep URL structure has resulted in crawlers assuming there is only one page of content, the page that is displayed when the site is first loaded. This means that later on during the project, even when the crawler began executing the JavaScript to cache the first page, it did not execute the content on other tabs within the DOM because it was not aware that they existed, and did not know which JavaScript functions would need to be executed to make additional content appear. This means that only a small portion of

the overall content of the site "exists" in the eyes of the crawlers, and is therefore of limited use in answering user queries.

Secondly, the lack of a deep linking structure to specific artist or track "pages" is detrimental to users' ability to share the content of the site across social media platforms. This is often a key driver for the organic growth of a site such as this one, and its absence is therefore problematic.

High Bounce Rate

One final, important side effect that results from the absence of the linking structure described above is its detrimental effect on an important search engine performance metric, the "Bounce Rate". A site's bounce rate is measured as the percentage of times that a user, when directed to a site, only views the first page before leaving or "bouncing". In the eyes of search engines, this metric is a proxy for the usefulness of a site, as it is assumed that the higher the bounce rate, and thus the fewer additional pages looked at by a user before leaving, the less helpful/informative/satisfactory the site is relative to others.

This metric works adequately when evaluating traditional sites where new pages are requested from the server each time a user wishes to see additional information. However, in a Single-Page Application (SPA), all users are seen as "bouncing", unless additional steps are taken to inform search engines that the site is both an SPA, and that certain navigational elements being requested within the SPA correspond to that of a traditional page request.

Because neither of these elements were present in the initial iteration of this application, the bounce rate for the site remained stubbornly high, regardless of the

amount of time users spent viewing the application, browsing playlists, or listening to songs.

Chapter 3: Further Improvements

Based on the observations from the first iteration of the site discussed in the previous sections, a series of improvements are planned for swapbeat.com over the course of coming months. These improvements are divided into "short-term" and "long-term" improvements and are discussed below.

Short-term Improvements

Two short-term improvements are being targeted for the site: Micro-data tagging and Mobile-only Style Sheets.

Micro-data tagging

Due to the ever-growing number of AJAX and JavaScript-dependent sites, the Schema.org project has been put forward, sponsored by major corporate partners such as Google, Microsoft, Yahoo and Yandex, with the mission to "create, maintain, and promote schemas for structured data on the Internet, on web pages, in email messages, and beyond." (Schema.org, n.d.) In the context of webpages, Schema.org promotes the insertion by developers of specific meta tags into each element of HTML in a page, which in turn help crawlers "digest" the content they are seeing during a crawl.

Of particular interest for this project was the definition by Schema.org of schemas for music recordings and specifically for individual recordings and playlists. Examples of the additional markup tags defined by Schema.org for a single musical track are highlighted in the overall markup of a track from the site in the code snippet below. These include "name" for the track name and "byArtist" for the

artist's name, along with several more generic Schema.org tags for "url" and "image".

```
<div itemprop="track" itemscope="true" itemtype="http://schema.org/MusicRecording"
id="Playlist Track 14" class="track" data-trackid="339" data-trackpos="14"><div
id="trackMovementIndicator14" class="trackMoveNumNoChange"><span class="glyphicon
glyphicon-option-horizontal"></span></div><button id="faveButton-sm14" class="faveButton-
sm-notfave" onclick="createOrDeleteFavorite(1, 339, 14, 2, &amp;quot;Beyond Our Means&amp;quot;,
&amp;quot;faveButton-sm14&amp;quot;)"><span class="glyphicon glyphicon-
heart"></span></button><div class="sitenames">Posted on: <span class="siteNameText"><a
itemprop="url" href="https://indiemusicfilter.com/listen-beyond-our-means-by-foxtrott"
target="_blank">Indie Music Filter</a></span> | <span class="siteNameText"><a itemprop="url"
href="http://pausemusicale.com/foxtrott-beyond-our-means/" target="_blank">Pause
Musicale</a></span> | <span class="siteNameText"><a itemprop="url"
href="http://feedproxy.google.com/~r/dummymagazine/~3/Y98fjidLAAI/premiere-foxtrott-
beyond-our-means" target="_blank">Dummy Mag</a></span></div><a
class="scLinkButtonMain" itemprop="url"
href="http://soundcloud.com/foxtrottfoxtrott/beyond-our-means" target="_blank"><img
src="image/buttons/sc_sbs_grey_130x20.png" width="130" height="20"></a><span
class="tracknumber">#14</span><span itemprop="name" class="trackname">Beyond Our
Means</span><span class="artistname"><span itemprop="byArtist" class="artistNameText"
onclick="fetchTracksByArtistID(454)">Foxtrott</span></span><button data-
trackidoffset="Track14" class="tracklistPlayButton" onclick="playButtonController(13)"
label="play"><span class="glyphicon glyphicon-play"></span></button><button data-
trackidoffset="Track14" class="tracklistPauseButton" onclick="pauseButtonController()"
label="pause"><span class="glyphicon glyphicon-pause"></span></button><span
class="barUITrackNumRef" data-baruitracknumref="Track14"></span><button
class="posSentButton" id="posSentButton14" onclick="registerEvent(2, 1, 339, 14,
&amp;quot;posSentButton14&amp;quot;)"><span class="glyphicon glyphicon-thumbs-
up"></span></button><button class="negSentButton" id="negSentButton14"
onclick="registerEvent(2, -1, 339, 14, &amp;quot;negSentButton14&amp;quot;)"><span class="glyphicon
glyphicon-thumbs-down"></span></button><div itemprop="image" id="artDivMain14"
class="artDivMain"><img src="https://i1.sndcdn.com/artworks-000133572427-jj2k79-
large.jpg"></div></div>
```

Figure 20: Example track markup with highlighted micro-data tags

Inclusion of these tags has appeared to help the Google crawler better understand the content of the site and has resulted in several keyword searches for popular artists displaying content from swapbeat.com in the cached Google results. Google also provides developers a convenient validation tool for reviewing whether

they have implemented their micro-data tags correctly (Structured Data Testing Tool, n.d.).

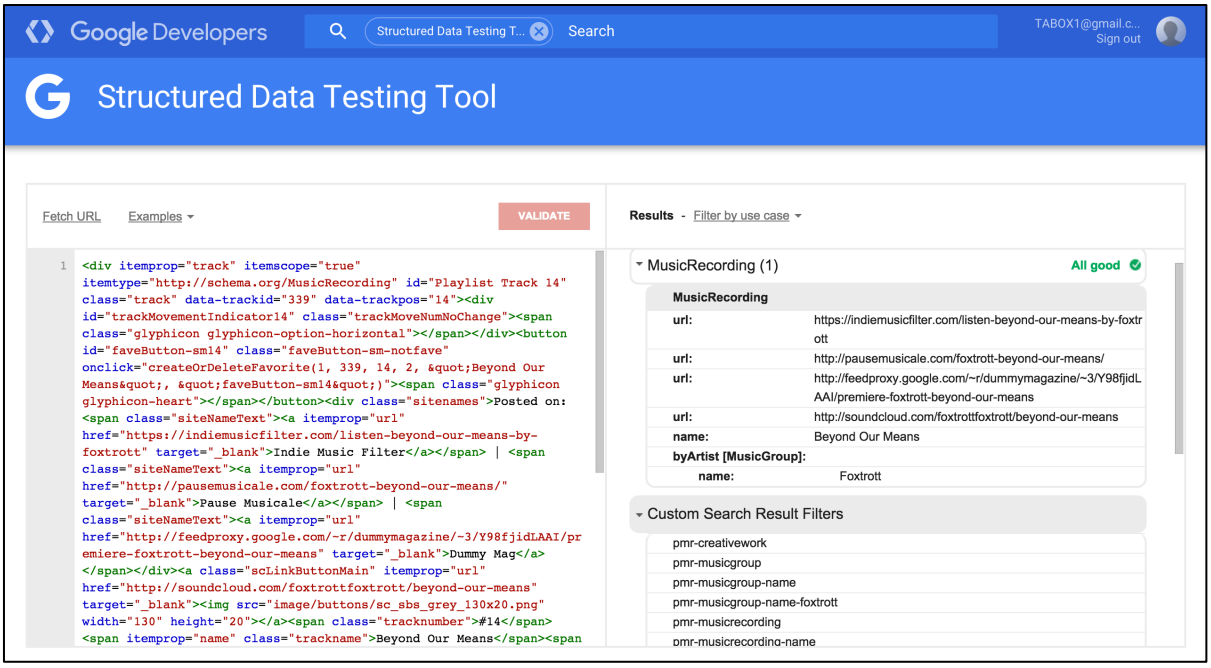

Figure 21: Google's structured data testing tool

In the screenshot above, the HTML from the earlier example has been pasted into the tool on the left-hand side, and Google's view of the data's structure is shown on the right side. In this example, the structured data tags help Google identify that this is a music recording of the song "Beyond Our Means" by the artist "Foxtrott".

Mobile-detection and Mobile-only Style Sheets

Another short-term improvement for the site will be the inclusion of mobile detection scripts that will be run when the site is first launched, redirecting the user to a mobile-focused version of the site driven by style sheets specifically targeted for the smaller dimensions of mobile screens.

In addition to mobile detection and rerouting, more-aggressive use of Bootstrap's "extra small" and "small" grid formats can be employed if the style sheet and class selectors are "mobile only". This enables the possibility to keep the site's business logic untouched while repositioning (and in some instances removing) elements for optimal display on smaller devices. This may allow the site to finally garner a "mobile-friendly" designation, improving its placement in search results temporarily while longer-term fixes are sought.

Longer-term Improvements

While the changes listed above may provide some marginal improvement for the problems of the current site, true corrections for these problems will not be possible until the site is transitioned from its current functional, "proof of concept" format, to a more robust, framework-driven site.

Redesigning the application with Angular JS

In order to make this transition, much of the client-side code will need to be refactored to align with the standard approach of the most-popular framework for developing commercial-grade single-page applications: Angular JS. Some of this work has already been undertaken and is included in the discussion below.

Basics of Angular JS

A Google test engineer named Misko Hevery first released AngularJS in June 2012 (Karpov & Netto, 2015). The goals of the project were primarily to help standardize the approach for building single-page applications (SPAs) in JavaScript,

and to modularize the creation of "pages" within SPAs so that they could more easily be separated from other components and the data mocked for testing.

Client-side MVC: Angular Controllers, Scopes, and Templates

One of the key differentiators in AngularJS's framework was the strict implementation of a Model-View-Controller paradigm within the browser. To do this, AngularJS requires the developer to focus on creating individual "scopes" of data for each JavaScript-based "page" they wish to display.

Each scope is instantiated by means of an assigned controller responsible for retrieving data and storing it on that page's scope. Scoped data can then be accessed and used within HTML "templates" that use Angular JS's directives to embed references to the data to be used directly in the HTML element markup of the element in which it will be displayed.

One example of this can be found in the syntax for the commonly used "ngRepeat" directive, which is used to create repeating lists of HTML elements from controller data. In order to create a list of tracks, for instance, the controller will fetch a JSON-representation of the tracklist and store it on the scope as "tracks". Within the HTML template, once the scope has been assigned, the ngRepeat syntax can be used to create a simple for-loop that easily and succinctly generates the HTML needed to display the list, beginning with the syntax "for track in tracks".

Angular developers can further enhance the functions within the for-loop of ngRepeat by making use of "filters". These filters primarily operate as simple JavaScript functions where specific data elements can be passed to the filter and the result returned while only needing to define a simple tag on the element. The

process of formatting artist names, previously discussed in this report, is a prototypical example of how a filter will be used in Angular JS.

Essentially, these components constitute the basics of the MVC architecture defined by Angular JS, with controllers fetching data and placing the results in the scope, which represents the client-side portion of the "model". The view is then defined and displayed on the basis of the instructions embedded in the HTML templates, directives such as ngRepeat, and further view enhancements made possible by the use of filters.

The Route Provider Module

In addition to the improvements made possible by client-side MVC, AngularJS also provides a sophisticated routing mechanism in the "route provider" module. This module operates by "listening" for specific changes in the "post-hash" portion of the URL. The essential functions of this module can be seen in the source code below.

```
app.config(['$routeProvider',
        function($routeProvider) {
                $routeProvider.
                        when('/Playlists/:type', {
                                templateUrl: 'trackListTemplate.html',
                                controller: 'trackListCtrl'
                        }).
                        when('/Artists/:artistID', {
                                templateUrl: 'trackListTemplate.html',
                                controller: 'artistTrackListCtrl'
                        }).
                        when('/Tracks/:trackID', {
                                templateUrl: 'trackListTemplate.html',
                                controller: 'getTrackCtrl'
                        }).
                        when('/Favorites', {
                                templateUrl: 'trackListTemplate.html',
                                controller: 'favoritesCtrl'
                        }).
                        otherwise({
                                redirectTo: '/Playlists/Newest'
                        });
        }]);
```

Figure 22: An example of the Route Provider syntax in AngularJS

For example, when using the Route Provider module with swapbeat.com, the module would observe a change from the base www.swapbeat.com domain to www.swapbeat.com/#/Playlists/Newest as a call to request the playlist of "Newest" tracks.

This functionality is made possible by defining a series of "when" statements in the route provider, describing which "post-hash" request patterns should be expected. Once a pattern is received and matched, Angular JS will load the controller and template for that "route", which in turn will result in the controller retrieving data and placing it on the scope for that route. In the example above, the "Newest"

designation was passed to the controller as the ":type" via a set of data elements Angular JS provides called "Route Parameters". These parameters can be accessed at the instantiation of the controller to further define what actions should be undertaken based upon which ":type" is present. This has the added effect of greatly increasing the reusability of these routes in instances where the same template and controller may be leveraged with minimal parameterization. Finally, any route not matching a pattern in the list of "when" statements will be caught by the "otherwise" block, which in this example will default the route to "/Playlists/Newest".

Angular JS not only utilizes the structure of the URL to drive the navigation of the site/application, it also natively manages the history of the "post-hash" portion of the site as part of the browser history. Therefore, a user pressing the "back" and "forward" buttons in a browser will be able to move fluidly through the history of the application, just as they would on a non-JavaScript dependent site.

Finally, the Route Provider module also makes possible the full site mapping required by web crawlers. Although a rapidly evolving standard, crawlers will make attempts to crawl the "post-hash" portion of a site if specific "SPA notification" tags for crawlers are added to a site's header. This then allows SPAs using Angular JS the same search engine discovery and optimization benefits enjoyed by standard sites, while also supporting the enhanced user experience made possible by SPAs.

Problems Angular JS can resolve directly

Angular JS provides a clear resolution for many of the problems experienced by the first iteration of the swapbeat.com site. Specifically, as mentioned, Angular JS

squarely addresses the issues related to search engine crawlability for an SPA via the Route Provider module and its handling of the post-hash portion of the URL.

In addition though, the Route Provider module also provides the knock-on effect of creating copy-and-paste ready links that can be shared across social media platforms, which, when entered into a browser, will drive the receiving user to the same exact page display as seen by the original user, something that was not always possible in the first iteration of the site due to a lack of integration with the browser history and the absence of URL-based navigation.

Finally, Angular JS addresses the search engine optimization problems SPAs face in regards to "Bounce Rate", the generally negative measure of single-page views discussed previously that can cause SPAs to perform poorly in search engine results. By integrating URL navigation, Angular JS allows developers to represent user activities as navigated "pages", with each new load of a view equating to a traditional "page load" in a non-SPA application.

Additional Problems Angular JS can help resolve indirectly

Because Angular JS strongly separates the view logic from that of controllers and scopes/models, one additional benefit of Angular JS is the ability it affords to rapidly prototype your view templates and experiment with multiple layouts. Furthermore, media queries in regards to the dimensions of the screen the user is currently accessing the site on can be included at the start of the application. Simple Boolean statements can then be incorporated into the Route Provider module to determine whether a mobile version or desktop version of a template should be used when loading the route.

Alternatively, Angular JS provides custom directives and a series of open source add-ons to standard Angular JS, such as "Mobile Angular UI", which supply custom directives for creating mobile apps. These projects will likely continue to gain support as more sites seek to directly leverage the Angular JS framework for their desktop sites and seek libraries for simplifying how the Angular-based mobile version of their site displays as well.

Google's retraction of the AJAX / JavaScript crawler spec

Finally, some gains in search engine crawl performance by Single-Page Applications may come from changes to the search engines' crawlers themselves. On October 14, 2015, a post on the official Google Webmaster Central Blog was added with the title "Deprecating Our AJAX Crawling Scheme" (Nagayama). Since 2009, Google had "proposed a set of practices that webmasters can follow in order to ensure that their AJAX-based applications are indexed by search engines", known as the "AJAX Crawling Scheme" (AJAX Crawling (Deprecated)). These practices typically required webmasters to detect when crawler user agents were making page requests and rerouting them to "pre-rendered" versions of the pages created as standard HTML. This requirement was so pervasive it resulted in the launch of several start-ups, such as "Prerender IO", that offer services to offload these additional indexing requirements to their servers for a monthly fee (Prerender IO, n.d.).

However, Google now acknowledges in "Deprecating Our AJAX Crawling Scheme" that their crawler is executing sites as a browser would – i.e. as a user interface – as opposed to the more text-based approaches used earlier. Because of

this, Google now states that they will directly index the “post-hash” portion of URLs. In addition to improving Google’s own crawler performance, this shift will likely also help to further drive the adoption of Angular JS (and its supported Route Provider module for managing the “post-hash” portion of URLs) as the premiere framework for developing Single Page Applications going forward.

Chapter 4: Vision and Roadmap

On June 16th, 2015, SoundCloud announced on their developer blog that they would be introducing rate limits for applications that use API keys to access their streaming services (Hudson, n.d.). These limits went into effect on July 1st, 2015, capping the number of streams per 24-hour period to 15,000 per API key. This move came on the heels of several high profile news reports about SoundCloud's financial solvency and its efforts to work with record labels to monetize the music posted on the site (Cookson, 2014), (Geddes, 2015).

To date, SoundCloud continues to make its revenues by charging artists for the ability to create an account and post their music, based on a "free-mium" model, with paid accounts starting at $6 per month for the "pro" level and $15 per month for the "pro unlimited" level (Payments and Billing).

There is a great deal of speculation, however, that SoundCloud would like to also begin charging users for the streaming service. Other monetization efforts are already underway, with SoundCloud recently introducing audio ads that play between songs for users accessing their home site, as is done with traditional terrestrial and digital radio stations.

In the post discussing the introduction of rate limits, SoundCloud stated, "In the coming months we'll be introducing an application process for developers who'd like additional access." (Hudson, n.d.) Four months later, this application process and the associated terms still have not been outlined publicly. However, it seems likely that SoundCloud will transition to either charging subscription fees to third-

party applications and sites that wish to stream music using the API, or institute a mandatory advertising mechanism for playing advertisements between streams.

The overall trajectory of SoundCloud's streaming API, and, in fact, the financial health of SoundCloud itself, therefore bear heavily on the future growth potential of swapbeat.com as of this writing. Nevertheless, the opportunity exists for swapbeat.com to continue to operate and even grow as a niche site, targeting music lovers who are committed to identifying the best new music within days of its release.

Therefore, work will proceed on redeveloping the Angular JS version of the site, fully supporting mobile display and URL-based navigation within the next 3-6 months. Once the redevelopment is complete, efforts will focus on expanding the site's user base in a controlled manner – one that keeps the daily play count less than 15,000 streams for now – while nevertheless providing a core set of frequent users ready access to the best new music on the Internet.

Appendix: Sites Included in the Crawl for Swapbeat.com

The following is a list of the sites that were seeded to the crawler when searching for new music on the Internet. Without the dedication and passion of these sites' owners and music reviewers, this project would not have been possible.

#	Site Name	Primary Music Review URL
1	Trendable Music	http://trendablemusic.com
2	Dancing Astronaut	www.dancingastronaut.com
3	Lipstick Disco	www.lipstickdisco.co.uk
4	Gotta Dance Dirty	www.gottadancedirty.com
5	Wonky Sensitive	http://wonkysensitive.blogspot.com/
6	EDM Tunes	http://www.edmtunes.com/
7	Spincoaster	www.spincoaster.com
8	Pilerats	www.pilerats.com/music
9	Little Indie Blogs	http://littleindieblogs.blogspot.com
10	Surviving the Golden Age	http://survivingthegoldenage.com/
11	Run the Trap	http://runthetrap.com/
12	Yaqui	http://yaqui.co/
13	Waxhole	http://waxholerecords.com/
14	In the Junk Yard Music	http://inthejunkyardmusic.co.uk/
15	Perfect Midnight World	http://perfectmidnightworld.com/
16	Audio Femme	http://www.audiofemme.com/
17	The Wild Honey Pie	http://www.thewildhoneypie.com
18	Apes on Tape	http://apesontape.com
19	DIY Mag	http://diymag.com
20	Site of Sound	http://www.siteofsound.com
21	The Fader	http://www.thefader.com/music/
22	Pop On And On	http://poponandon.com
23	Keep On Repeat	http://www.keeponrepeat.com
24	Stereogum	http://www.stereogum.com
25	Noisey	http://noisey.vice.com
26	trndmusik	http://trndmusik.de
27	Kick Kick Snare	http://kickkicksnare.com
28	Y Este Finde Que	http://www.yestefindeque.com/
29	Winnie Cooper	http://winniecooper.net
30	City & Sound	http://www.cityandsound.com
31	Mad Decent	http://maddecent.com

32	Silence No Good	http://silencenogood.net/
33	The 405	http://www.thefourohfive.com
34	Ear Milk	http://www.earmilk.com
35	Blushing Panda	http://blushingpanda.org/blog
36	Indie Shuffle	http://indieshuffle.com
37	Harder Blogger Faster	http://www.harderbloggerfaster.com
38	Consequence of Sound	http://consequenceofsound.net
39	Turntable Kitchen	http://www.turntablekitchen.com
40	EDM.com	http://edm.com/
41	Scientists of Sound	http://www.sos-music.co.uk/
42	Disco Belle	http://www.discobelle.net
43	Nialler9	http://nialler9.com/
44	Drowned In Sound	http://drownedinsound.com
45	The Quietus	http://thequietus.com
46	FACT	http://www.factmag.com/
47	Boiler Room	http://boilerroom.tv
48	XLR8R	http://www.xlr8r.com/
49	Okayplayer	http://www.okayplayer.com/
50	Caveman Sound	http://cavemansound.com
51	Phuturelabs	http://www.phuturelabs.com
52	Brooklyn Vegan	http://www.brooklynvegan.com/
53	Dublab	http://dublab.com
54	Neonized	http://neonized.net
55	Hypetrak	http://hypetrak.com
56	The Line of Best Fit	http://www.thelineofbestfit.com
57	Decoder	http://www.secretdecoder.net
58	Spex	http://www.spex.de
59	Rockets Musik	http://www.rocketsmusik.com/
60	Stoney Roads	http://stoneyroads.com
61	Fluid Radio	http://www.fluid-radio.co.uk
62	Inverted Audio	http://inverted-audio.com
63	Tiny Mix Tapes	http://www.tinymixtapes.com/
64	Golden Scissors	http://goldenscissors.info
65	Baltimore Club	http://www.baltimore-club.com/
66	Mixmag	http://www.mixmag.net
67	Pigeons & Planes	http://pigeonsandplanes.com/
68	Oh So Fresh	http://www.ohsofreshmusic.com/
69	Lyfstyl	http://lyfstylmusic.com/
70	Indie Music Filter	http://indiemusicfilter.com
71	Mixtape Riot	http://mixtaperiot.com
72	Buzz Bands LA	http://buzzbands.la

73	Fool's Gold	http://foolsgoldrecs.com
74	Robot Dance Music	http://www.robotdancemusic.com
75	Impose Mag	http://www.imposemagazine.com
76	Igloo Mag	http://igloomag.com
77	Anthem Mag	http://anthemmagazine.com/
78	Ashley Outrageous	http://ashleyoutrageous.com
79	Fresh New Tracks	http://freshnewtracks.com/
80	My Music Dealers	http://musicdealers.in
81	Tonspion	http://www.tonspion.de
82	Chrome Music	http://www.chromemusic.de
83	IHEARTCOMIX	http://iheartcomix.com
84	Music You Want To Listen To	http://musicyouwannalistento.blogspot.com
85	Future Classics	http://futureclassics.ca
86	Exclaim!	http://exclaim.ca/
87	MOARRR	http://moarrr.com/
88	PressPLAY	http://pressplayok.com
89	When the Horn Blows	http://whenthehornblows.blogspot.co.uk
90	Oh My Rock	http://www.ohmyrock.net
91	Vehlinggo	http://vehlinggo.com
92	Pan od Muzyki	http://panodmuzyki.pl
93	Oblivious Pop	http://www.obliviouspop.com
94	Happy	http://hhhhappy.com
95	Idolator	http://www.idolator.com
96	Audio Aquarium	http://www.audio-aquarium.com
97	Free Indie	http://www.freeindie.com
98	Popped Music	http://poppedmusic.co.uk
99	Free Bike Valet	http://freebikevalet.com
100	The Revue	http://therevue.ca
101	Your EDM	http://www.youredm.com
102	Pitchfork	http://pitchfork.com
103	I Love Pie	http://www.ilovepie.co.uk
104	Stereo Fox	http://www.stereofox.com
105	The Autumn Roses	http://theautumnroses.tumblr.com
106	MPMBL	http://mapambulo.blogspot.com
107	Front Stage Music	http://www.frontstagemusic.net
108	The Morning After	https://morningaftershow.wordpress.com
109	Nicorola	http://www.nicorola.de
110	Ovrld	http://ovrld.com
111	Disco Naivete	http://disconaivete.com
112	Spin	http://www.spin.com
113	Drums Eat Everything	http://www.drumseateverything.com

114	Sodwee	http://sodwee.com/blog/
115	Going Solo	http://www.wearegoingsolo.com
116	Hilly Dilly	http://www.hillydilly.com
117	Democrazy	http://www.democrazy.be
118	Dummy Mag	http://www.dummymag.com/
119	Kicking the Habit	http://kickingthehabit.nl
120	Camels and Lions	http://www.camelsandlions.com
121	Poule d'Or	http://www.pouledor.com/
122	Nordic by Nature	http://nordicbynatureberlin.com
123	Hey	http://www.letsgethey.de
124	The Blue Walrus	http://thebluewalrus.com/
125	Sound of Aarhus	http://www.soundofaarhus.com
126	No Fear of Pop	http://nofearofpop.net/
127	Good Because Danish	http://goodbecausedanish.com
128	Don't Watch Me Dancing	http://dontwatchmedancing.com
129	YVYNYL	http://yvynyl.com
130	Abeano	http://www.abeano.com/
131	Breaking More Waves	http://breakingmorewaves.blogspot.com/
132	Cruel Rhythm	http://cruelrhythm.tumblr.com
133	Dots and Dashes	http://dotsanddashes.co.uk
134	Faded Glamour	http://www.fadedglamour.co.uk/
135	The Thin Air	http://thethinair.net
136	The VPME	http://www.thevpme.com/
137	Obscure Sound	http://www.obscuresound.com/
138	Buddyhead	http://www.buddyhead.com/
139	UPROXX Music	http://uproxx.com/music
140	Boi-1da	http://boi-1da.net/
141	Indietronica	http://indietronica.org/
142	The Jack Plug	http://www.thejackplug.com
143	Et Musique Pour Tous	http://www.etmusiquepourtous.com
144	Secret Delivery	http://secretdelivery.net
145	Passion Party	http://passion-prty.de
146	All Tomorrow Music	http://www.alltomorrowmusic.com
147	The Postie	http://thepostie.de
148	Electru	http://www.electru.de
149	HDIYL	http://hdiyl.de
150	Future Dance Music	http://futuredancemusic.com
151	Ja Ja Ja	http://jajajamusic.com
152	Nothing But Hope and Passion	http://nbhap.com
153	Gilles Peterson	http://www.gillespetersonworldwide.com
154	Magnetic Mag	http://www.magneticmag.com

155	High Clouds	http://highclouds.org
156	We Love That	http://welovethat.de
157	Fingers on Blast	http://fingersonblast.com/
158	The Music Ninja	http://www.themusicninja.com
159	Salacious Sound	http://salacioussound.com
160	Some Candy Talkin'	http://www.somecandytalkin.com
161	Complex	http://www.complex.com/
162	La.Ga.Sta.	http://www.lagasta.com/
163	New Dust	http://www.newdust.com
164	Syffal	http://www.syffal.com
165	Hear Ya	http://www.hearya.com/
166	Pop Justice	http://www.popjustice.com
167	Fake Shore Drive	http://www.fakeshoredrive.com/
168	All Hip-Hop	http://allhiphop.com
169	EDM Sauce	http://www.edmsauce.com
170	Indie Music Review	http://www.indiemusicreview.com
171	Jungle Indie	http://jungleindierock.tumblr.com
172	Aquarium Drunkard	http://www.aquariumdrunkard.com
173	La Musique Sismique	http://lamusiquesismique.fr
174	Int'l House of Sound	http://intlhouseofsound.com
175	The Interns	http://theinterns.net
176	Fist in the Air	http://fistintheair.com
177	Hullabaloo Tunes	http://www.hullabalootunes.com
178	Find A Song	http://www.findasongblog.com
179	Music and Other Drugs	http://musicandotherdrugs.com
180	Pause Musicale	http://pausemusicale.com
181	Melbourne Bounce	http://www.melbournebounce.com.au

Bibliography

Audiomack. Retrieved September 15, 2015, from Audiomack: www.audiomack.com

SOLR. Retrieved September 15, 2015, from SOLR: http://lucene.apache.org/solr/

Simple Pie. Retrieved September 19, 2015, from Simple Pie: http://simplepie.org/

Chart JS. Retrieved September 3, 2015, from Chart JS: http://www.chartjs.org/

Heroku. Retrieved September 7, 2015, from Heroku: https://www.heroku.com/home

AWS RDS. Retrieved September 6, 2015, from AWS RDS: https://aws.amazon.com/rds/

AWS Aurora. Retrieved September 8, 2015, from AWS Aurora: https://aws.amazon.com/rds/aurora/

Schema.org. Retrieved September 6, 2015, from Schema.org: www.schema.org

Prerender IO. Retrieved October 27, 2015, from Prerender IO: https://prerender.io/

AJAX Crawling (Deprecated). Retrieved October 28, 2015, from Google Developers: https://developers.google.com/webmasters/ajax-crawling/docs/getting-started?hl=en

Alexa Global Site Rankings. (2015, September). Retrieved September 26, 2015, from www.alexa.com: http://www.alexa.com/topsites/global

Cookson, R. (2014, October 4). *SoundCloud Hits An Impasse With Major Record Labels.* Retrieved September 16, 2015, from FT.com: http://www.ft.com/cms/s/0/e549661c-4ef6-11e4-b205-00144feab7de.html#axzz3qeQOH700

Faughnder, R. (2015, June 28). Music Piracy Is Still Very Much In Play. *L.A. Times*.

Geddes, J. (2015, September 22). *Soundcloud CEO Alexander Ljung Talks About Evolving the Platform, Negotiating with Labels.* Retrieved October 14, 2015, from Tech Times: http://www.techtimes.com/articles/86434/20150922/soundcloud-ceo-alexander-ljung-talks-about-evolving-the-platform-negotiating-with-labels.htm

Hudson, D. *Introducing Rate Limits*. Retrieved September 17, 2015, from Soundcloud Developers: https://developers.soundcloud.com/blog/limits

Kaplan, C. S. (2001, February 23). Legal Expert Sees Napster Competitors Thriving. *New York Times*.

Karpov, V., & Netto, D. (2015). *Professional Angular JS.* New York: Wrox.

Makino, T., Jung, C., & Phan, D. *Finding More Mobile-Friendly Search*. (Google) Retrieved September 9, 2015, from Google Webmaster Central: http://googlewebmastercentral.blogspot.com/2015/02/finding-more-mobile-friendly-search.html

Measuring Traffic To Your Website. Retrieved September 7, 2015, from Google Adwords Help: https://support.google.com/adwords/answer/1722035

Nagayama, K. *Deprecating Our AJAX Crawler Scheme.* Retrieved October 24, 2015, from Google Webmaster Central: http://googlewebmastercentral.blogspot.com/2015/10/deprecating-our-ajax-crawling-scheme.html

Payments and Billing. Retrieved October 18, 2015, from Soundcloud.com: http://help.soundcloud.com/customer/portal/articles/247820-what-s-the-difference-between-each-subscription-level-

Schiller, S. (n.d.). *Bar UI Player*. Retrieved September 4, 2015, from www.schillmaina.com: http://www.schillmania.com/projects/soundmanager2/demo/bar-ui/

Schiller, S. (n.d.). *Sound Manager 2*. Retrieved September 5, 2015, from Schillmania.com: http://www.schillmania.com/projects/soundmanager2/

Structured Data Testing Tool. (Google) Retrieved September 9, 2015, from Google Developers: https://developers.google.com/structured-data/testing-tool/

WordPress. (n.d.). Retrieved September 18, 2015, from Wikipedia: https://en.wikipedia.org/wiki/WordPress