Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

# FEED PLAYER

MUSIC DISCOVERY THROUGH TEXT AND IMAGE ANALYSIS OF FEEDS

**BACHELOR THESIS**

**Internationale Medieninformatik**
**International media and computing**

**Philipp Hofmann**
**Student number: s0522482**

**Hochschule für Technik und Wirtschaft Berlin**
**University of Applied Sciences**

**August 1st 2013**
**Submitted to:**
**Prof. Dr.-Ing. David Strippgen**
**First examiner**
**Prof. Dr. Barbara Kleinen**
**Second examiner**

## Abstract

Reading about music in a preferred media source, finding the music brought to attention and listening to it is a common music discovery pattern. How can it be scaled by automation, and what is the quality of the resulting experience? To answer these questions, state-of-the-art music discovery engines, existing music providers and content aggregators were researched. The building blocks of a novelty concept were established, its qualities reasoned for. As a result, a web application that does RSS feed search, aggregation, parsing for subjects, query for, and display of active music content needed to be implemented. Using an open source web framework, libraries in its eco system, and APIs of services, this was learned and achieved. The targeted automation is possible, demonstrated by proof of concept. The variety the system is able to recommend from depends on the variety within input feeds and their quantity. The subjective quality of the music discovery experience depends on the selected feed content quality. This gives the user control over two key aspects of the music discovery experience: scale and care.

Keywords: web application, music discovery, information retrieval.

## Acknowledgment

I would like to thank Alma Gomera von Hippel-Liedl for her continued intellectual and emotional support, and Jeffrey Paul, Chris Sattinger, Jeff Haynes, Christoph Guttandin, Joel Bernstein and Joshua Moody for important feedback on my work from developer professionals; Miguel Senquiz for encouraging me to take a second look at The Echo Nest API, and Jeff Samuel Stern for his idea to use Google reverse image search in my strategies to detect artist-release information.

# Table of contents

# Table of figures

# Preface

The need for music discovery has reached a new level – just on Spotify, 24 million active listeners can access more than 20 million songs free of charge (Information | Spotify Press 2013). On a mobile device connected to the Internet these all fit into a single pant's pocket. Yet the music discovery problem is far from solved, with each approach showing major deficits that are summarized in this body of work.

At the same time, text and image analysis have made major leaps: The Echo Nest's Application Programming Interface (API) will return artist entities found in arbitrary text (The Echo Nest API 4.2 documentation 2013), and Google Images can provide a highly accurate guess on what a given images represents. (Google Images 2013)

Despite Google shutting down its RSS[1] reader service (Official Blog: A second spring of cleaning 2013) the format still enjoys popularity both among users and publishers. The feed reader Feedly.com alone reached 12 million users in June (Hazard Owen 2013) and claims to continuously update more than 25 million feeds every day. (Building Feedly 2013)

A music discovery engine taking advantage of these facts is proposed in chapter one. Music discovery state-of-the-art is presented in chapter two, chapter three then critically discusses the existing solutions, and chapter four concludes by reasoning for the proposed system, describing what it should be able to do. Options for implementation are discussed in chapter five. Now the high level architecture is outlined in chapter six and the system's implementation is described in seven. After presenting the results in chapter eight, chapter nine summarizes, conclusions are drawn in chapter ten, and chapter eleven gives an outlook on things to come.

The proposed system refines music discovery based on state of the art text and image analysis of articles of critical or editorial review. It is motivated by the author's dissatisfaction with today's music discovery experience. Many discovery engines disregard the rich data of such sources in favor of user activity data altogether (e.g. LastFM[2], Amazon[3], iTunes[4]), those that do take them into account do not give the listener control over the sources of critical review that influence the recommendation (e.g. The Echo Nest). (B. Whitman 2013)

[1] "RSS Rich Site Summary (originally RDF Site Summary, often dubbed Really Simple Syndication) is a family of web feed formats used to publish frequently updated works—such as blog entries, news headlines, audio, and video—in a standardized format." (RSS - Wikipedia, the free encyclopedia 2013)

[2] http://www.last.fm/

[3] htttp://www.amazon.com

[4] http://www.apple.com/itunes/

# 1 Goal

How is it possible to extract music recommendation subjects by parsing sources of critical or editorial review in RSS format? How can the results be turned into a playlist that represents the review's subjects? And finally, given the user can choose the sources, what is the quality of this music discovery experience? The joy of finding music that enriches the experience of life is worth finding out.

The approach is to use a high level scripting language and a popular open source web application framework with a lively eco system of libraries for common programming tasks. That should allow the implementation to focus on the core idea, and create a good starting point should the system be developed further in the open source community.

# 2 State-of-the-art: music discovery

## 2.1 The Echo Nest

Providing a large repository of dynamic music data, The Echo Nest aims to offer an API to music services that help them solve the problems of music recommendation and discovery. Co-founder, and MIT[5] PhD Brian Whitman[6] divides those into solving the problems of artist or song similarity in absence of user activity data, personalized recommendation based on such data, and playlist generation, adding the complexity of order. (B. Whitman 2013) Artist API methods give access to useful data around an artist: related biography, blog, review, and news items, relative rankings of popularity, images, matching songs, and genres. It can also return lists of artists: searches by property are possible, similar artists are listed; it will even return a list of artists found in a given text to parse. (The Echo Nest API 4.2 documentation 2013) The Playlist API provides methods to return static and even dynamic playlists, adjusting to played/skipped songs, rated songs/artists, favourited/banned artists/songs, and thumb up/down steering. Whitman published an article on existing music discovery engines that also offers insight on how his company is doing it just 7 months prior to completion of this thesis. It categorizes existing solutions by the source data they take as input, and evaluates them by the parameters of "scale" and "care" of their music recommendation output (B. Whitman 2013), which will serve as a model for this thesis.

## 2.2 Source of data

Whitman distinguishes four main sources of data music discovery engines work with: acoustic analysis, text analysis, activity data, and critical or editorial review.

### 2.2.1 Acoustic analysis

Whitman says that means of automated acoustic analysis are known to be able to determine tempo, key, and amplitude of songs, and that more sophisticated ones even do signature detection, beat tracking, transcription of dominant melodies, and instrument recognition. Herrada would classify this source as input for a content-based filtering system, using automatic feature extraction, since the key component of this approach is the similarity function among items. (Herrada 2008, 35)

[5] Massachusetts Institute of Technology

[6] http://variogr.am/, http://alumni.media.mit.edu/~bwhitman/

### 2.2.2 Text analysis

Here methods of natural language processing (NLP) are used to learn what people are saying about music on the Internet. At The Echo Nest, information retrieval systems are crawling the web and scan music related pages. The data is filtered to omit spam and non-music related content, and artist entities are extracted from large amounts of text. Also the text around it is parsed. Descriptive terms, and noun phrases are put in relation to artist and song entities, and weighted on occurrence. This data is merged with knowledge base data from community access sites like Wikipedia or Music Brainz, creating cultural vectors that are used to determine relations and similarity between both artists and songs. (B. Whitman 2013) This approach would be characterized as context-based filtering by Herrada. He distinguishes web minig as "analyzing available content on the Web, as well as usage and interaction with the content", while social tagging "mines the information gathered from a community of users that annotate (tag) the items". (Herrada 2008, 38)

### 2.2.3 Activity data

User generated data can take many forms in music discovery systems. Purchase and browsing history, play count, skipped songs, and ratings provide source data. This is widely used as Whitman describes: companies like Amazon, Last FM[7], or iTunes use this data as the primary input to their music discovery systems. (B. Whitman 2013) Herrada labels these systems as "collaborative filtering": they are predicting user preferences based on past user-item relationships of other users. (Herrada 2008, 30f)

### 2.2.4 Critical or editorial review

Data stemming from editorial work, user review or tagging, and user surveys make up the fourth group of source data types for music discovery engines according to Whitman. AllMusicGuide[8], and Pandora[9] are named as examples for companies employing a team for them in-house. AllMusicGuide employs experts labeling music with genres, Pandora employs non-automated musicological analysis, also according to Magno and Sable (Magno und Sable 2008). For Herrada, this source would match multiple categories: content-based filtering based on manual annotation by experts, or tags from a community of users could describe items, and create source data for a similarity algo-

[7] Advertised as a recommendation service for music, Last.fm attempts to suggest music to its users based on their activity data in a radio stream. Their Scrobbler is a tool that once installed, logs and sends music played by its users to Last FM, where it is processed and related to the data other users are providing. (About Last.fm – Last.fm 2013)

[8] http://www.allmusic.com/

[9] http://www.pandora.com/

rithm. Also context based filtering would benefit from this data source as mentioned before. (Herrada 2008, 35,38)

A special type of source are music recommendation authorities, music magazines are an important example[10]. They are mediums that filter content, and provide a source to text analysis by publishing articles of critical or editorial review.

## 2.3 Social music discovery

Whitman classifies friend-to-friend music recommendation enabled by social networks as social music discovery. This is true for users sharing tracks in playlists as on Playground.fm (Playground.fm | VentureBeat 2012), Spotify[11], (Spotify 2013), Whyd, (Whyd 2013), and Minilogs (Minilogs 2013).

## 2.4 Aggregators

### 2.4.1 RSS feed aggregators

RSS aggregators solve part of the problem of reading a lot of different sources of critical or editorial review. By subscribing to feeds of the preferred sources, all their articles can be digested quicker in one view. Usually only headline, summary, and sometimes the text body are encoded in the XML format. A good example is Feedly.com (feedly: your news. delivered. 2013): RSS feeds can be searched for and selected, and placed in folders. Their updated content is then displayed in sequence with the other feeds in that folder. Aggregated music feeds can help discover music.

### 2.4.2 Meta blogs

The HypeMachine (Hype Machine 2013) exemplifies this service: MP3 downloads or streams that blogs offer are aggregated in an instantly playable format on a page, with filtering and sorting options by genre, date of post, popularity and more. Music can be discovered this way. When logged in, artists, blogs, and Twitter[12] friends can be followed to personalize the stream of tracks presented.

[10] Through articles, interviews, reviews, charts, event listings, event pre/reviews and news items, musical products and their protagonists are editorially selected, presented, and ultimately filtered for a certain audience. They can draw from a detailed catalogue, and often times have the market power to demand promotional copies of musical products months before their release date.

[11] Signing up requires a Facebook account and consent to having the company import the users' friends on that platform. Playlists and currently playing song are public to friends by default, which encourages browsing their music selections. Likewise, artists and celebrities have accounts that can be followed. According to TechCrunch (Spotify Launches "Discover" Feature For Its Web App | TechCrunch 2013) they now feature a discovery view which mixes means of music discovery offered by The Echo Nest, and social music discovery cues, as suggested by Whitman (B. Whitman 2013).

[12] https://twitter.com/

# 3 Deficits in existing solutions

## 3.1 Scale & care

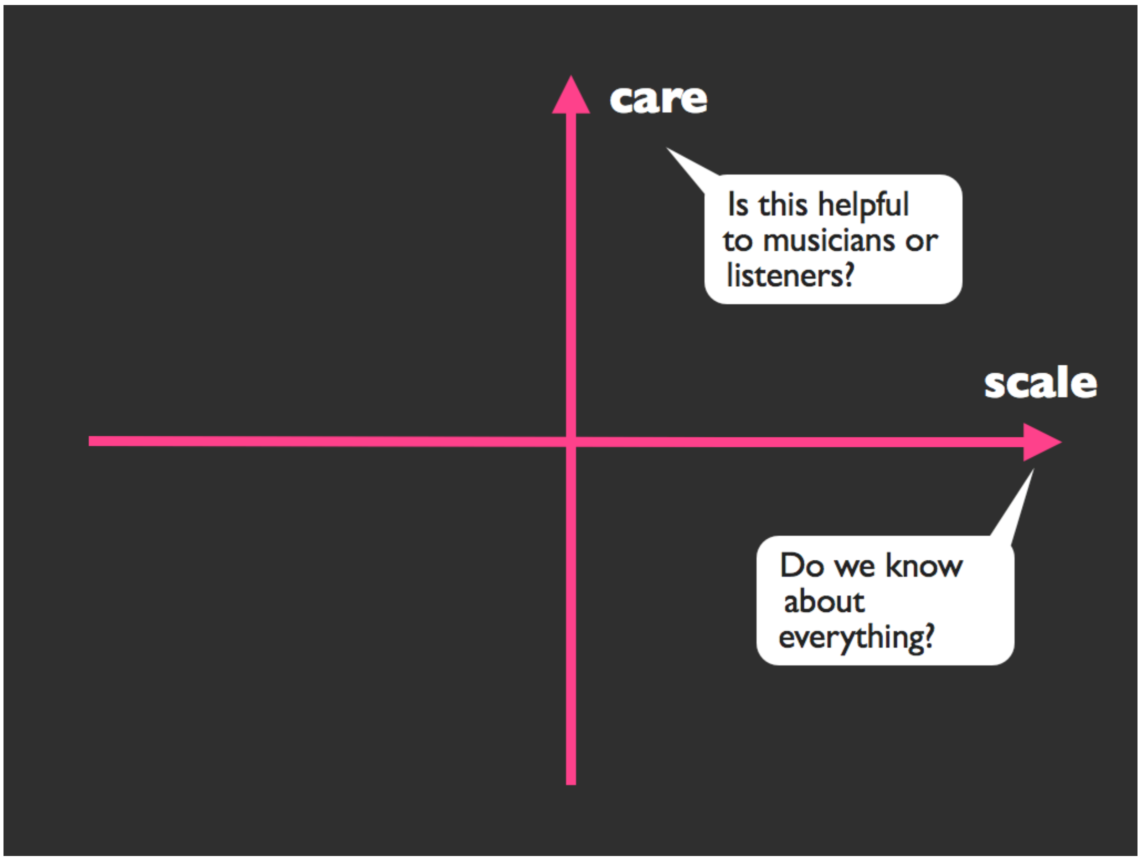

care
Is this helpful to musicians or listeners?
scale
Do we know about everything?

Figure 1: Scale and care evaluation coordinate system (B. Whitman 2013)

In Whitman’s article, “scale” is defined as the amount of music a system is able to recommend. Systems employing critical or editorial review lack “scale” according to him since they are not able to know about all the artists and songs to be. Magno and Sable agree, giving “the relatively slow rate at which new content is added to the Pandora database” as an example, explaining that the system to describe music by musicological means is not automated there. (Magno und Sable 2008, 162)

He argues that the recommendation engines used by LastFM, Amazon or iTunes that employ activity data or collaborative filtering inherently lack “scale”, since they tend to recommend the popular, and fail to be able to recommend music that does not have much activity data, for example for being new. Dahlen agress saying that Last FM “rarely surprises you: It delivers conventional wisdom on hyperdrive, and it always seems to go for the most obvious, common-sense picks.” (Dahlen 2006) Herrada agrees and explains: “popular items of the dataset are similar to (or related with) lots of items.

Thus, it is more probable that the system recommends these popular items”. (Herrada 2008)

“Care” is defined by Whitman as a measure of how helpful the recommendation to musicians and listeners is, how much their ideas of a good music discovery experience is disrupted by algorithmically correct but humanly rejected results. He shows how systems that rely heavily on activity data such as purchase and browse history (Amazon) have very little “care”, while those using critical review and text analysis do much more.

Whitman concludes that acoustic analysis is unable to provide meaningful music recommendation since signal similarity does not translate to perceived cultural music similarity well. (B. Whitman 2013) Herrada supports this view, describing the problem of content-based music recommendation as the problem of the limitation of features that can be extracted. (Herrada 2008, 37f) It is beneficial to “care” for Whitman though, being able to provide for a smoother experience in the sequence of recommended music, for example in playlists. (B. Whitman 2013) Herrada also concedes that automatically extracted descriptors such as harmony and rhythm are useful to compute item similarity. (Herrada 2008, 37f)

Whitman sees social music discovery via social networks as a non-discovery engine for not being automated. Indeed social networks provide amazing music discovery through personal recommendation but require manual parsing and storing of that information. He also argues that they fail at “scale” but are useful for finding music that would otherwise not have been recommended. He finds their value in giving music recommendation authority stating, “People don’t like computers telling them what to do”. (B. Whitman 2013) He advocates their use for mixing with automated approaches, and gives Spotify’s discovery feature[13] as an example. (B. Whitman 2013)

Convenience is needed to make music discovery based on articles of critical or editorial review feasible. Visiting several different websites, scrolling endlessly, and subsequently searching for the recommended content on stream service providers and download stores is not sufficiently convenient. Although RSS aggregators provide a shortcut to articles by multiple music authorities, the recommended content is still at least two clicks away: either the embedded content needs to be accessed by a click on the link to the post’s related URL, or the post’s subject needs to be searched for by hand with a streaming music provider. Meta-blogs are limited to sources that directly publish playable content, through linking to, or embedding it.

[13] According to TechCrunch (Spotify Launches "Discover" Feature For Its Web App | TechCrunch 2013), since they integrated their public APIs, now The Echo Nest also influences Spotify’s discovery view mixing it with social music discovery cues, as suggested by Whitman (B. Whitman 2013)

# 4 Approaching a different solution

Whitman’s argument for the limited “scale” of music discovery systems that rely on editorial review holds true against such a system as proposed. At the same time, multiplying the sources of critical review by simply adding more feeds can augment the “scale” factor. Also, the value of limitation cannot be denied; if the “scale” of possible recommended content is limited by the sources of recommendation that are already trusted, this leads to a higher “care” factor for recommended items. With the proposed system, the popularity bias known from activity data-based music discovery engines would entirely depend on the users’ choice; If only feeds featuring top ten listings are chosen, the bias is maximal, if only obscure blogs on a niche sub genre are selected, popularity bias is not an issue. How helpful the proposed system is to musicians and consumers of music alike also depends entirely on the selection of feeds. The more reliably and accurately the feed represents what the user is interested in musically, the greater the “care” factor.

Making music recommendation as individual as the choice of music recommendation authorities motivates approaching this solution. Reading aggregated RSS feeds, following entities on social networks, and reading meta-blogs does that but lacks the automation to make it convenient. This is where the author hopes to make a difference – creating a system that is able to aggregate musical content, specified by the individual choice of recommendation authority, presented in an instantly playable format - even if no stream or download is provided by that source.

In that, it does not fully fit Whitman’s definition of a music discovery engine; it does not provide information on similarity between artists and songs. Its degree of personalized recommendation is limited to the personal choice of feeds, and its ability to generate playlists is lacking the intentional order such format implies. Instead, it targets the underlying problem those have in common: suggestions on what to listen to next, done as individual in “scale” and “care” as the user desires.

## 4.1 Choice of music providers

The music providers SoundCloud (Soundcloud.com 2013) and YouTube (YouTube 2013) chosen for this proof of concept are described in the following; a summary of discarded options justifies the selection.

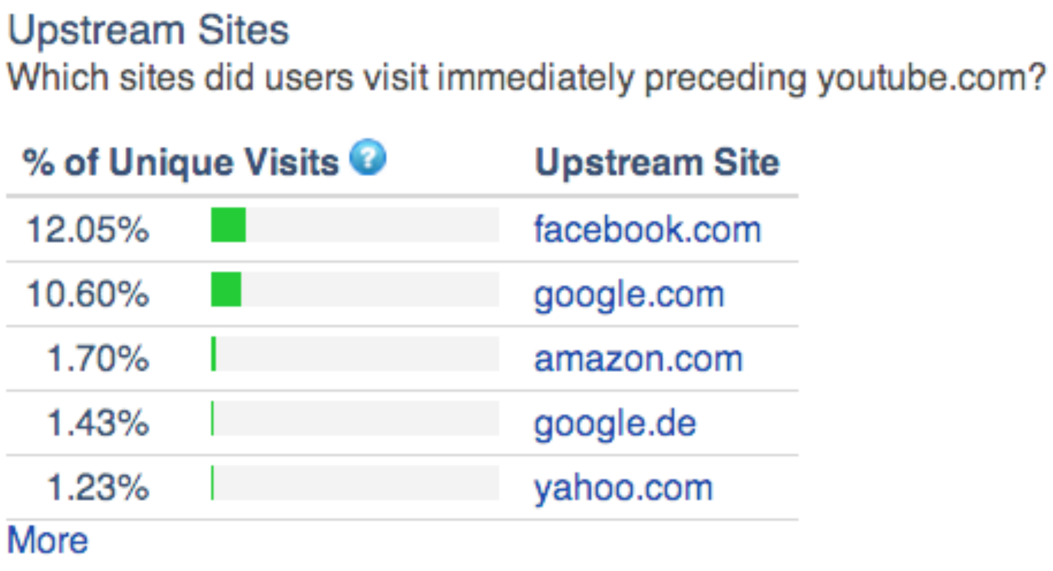
### 4.1.1 SoundCloud

As of December 2012, 180 million users generated and uploaded audio to the social music platform (Techcrunch.com 2012). Since the rights holders themselves provide the music with a full and free license, and decide on its accessibility (SoundCloud - Hear the world's sounds 2013), the company does not have to pay royalties for the music it hosts; Listening to the content made public by its users is free (Soundcloud.com 2013). Early on the platform offered free embedded widgets for the sounds uploaded there, making them a popular standard today: these players are used by artists, music brands, bloggers, users of social networks and music magazines. It has also so far been able to avoid the conflict with intellectual property royalty collection societies like GEMA, arguing that they merely provide a platform for the rights holders to publish and are not charging for the consumption of that content[14]. Its large repository of freely accessible music makes it a good choice for the proposed system. Its popularity with use of embedded content found in articles of critical or editorial review is another reason to choose SoundCloud as the provider of music.

### 4.1.2 YouTube

YouTube makes it possible for users to upload and share video on the Internet. (YouTube 2013) Its significance for music as a provider is huge. A possible clue comes from the referrer statistics:

[14] It can be assumed that this would change once GEMA successfully resolves its ongoing confrontation with Youtube on a similar issue (German court orders Google to police YouTube copyright violations 2012). At the time of writing, music not available on Spotify or Youtube in Germany can very well be found on SoundCloud.


Upstream Sites
Which sites did users visit immediately preceding youtube.com?

| % of Unique Visits | Upstream Site |
|---|---|
| 12.05% | facebook.com |
| 10.60% | google.com |
| 1.70% | amazon.com |
| 1.43% | google.de |
| 1.23% | yahoo.com |

More

Figure 2: YouTube’s number one referrer is facebook.com (Youtube.com Site Info 2013)

This indicates that many users are getting aware of, and interested in content on Youtube while browsing Facebook[15]. Another clue is Nielsen’s recent study asserting that 64% of teens listen to music through YouTube (Music Discovery Still Dominated by Radio, Says Nielsen Music 360 Report | Nielsen 2012). Considering that both Billboard and Nielsen now deem Youtube to be an important input to their chart monitoring, (Billboard.com 2013) and, “nine in 10 of the most popular videos on the service are music related” (IFPI 2013) it can be concluded that YouTube plays a major role as a place where people look for, and listen to music. Its use as hosting place for music, be it with a professionally produced music video as the video component, or just with a still picture of unknown origin is so wide spread that it has become the audio player of convention. For example Discogs.com, a major discography database has adopted it as such; their users select highly accurate representations of the discography entries on the social video service. (Explore Videos on Discogs 2013) YouTube’s ubiquitous use makes it a provider that the proposed system should look for when parsing sources of critical or editorial review for embedded content.

### 4.1.3 Other options

#### 4.1.3.1 Personalized stream radio services

Last FM is a prominent example for this class of services; activity data-based music recommendation in a free, ad supported radio stream. The individual tracks are not ac-

[15] https://www.facebook.com/

cessible or organized in playlists by users, but cued up by the engine. Users tell the system which music they already play, for example via Last FM's "scrobbler". The system selects songs that other users who also like these songs have played. (About Last.fm – Last.fm 2013) Since Last FM does not provide access to full individual tracks at all, it is not interesting as a provider to the proposed system.

#### 4.1.3.2 Streaming music services

Spotify is a prominent representative of this model; With over 20 million songs licensed from rights holders the stream service grants direct access to a large amount of songs and bundles with an ad supported free-, or a monthly fee subscription. Spotify is limited to a narrower licensed catalogue than SoundCloud and features ads on the free version, so it is not chosen for the first iteration of the proposed system.

#### 4.1.3.3 Music download stores

Here music is downloaded for a per-track or bundle price. Their catalogue is more complete then that of most stream services; The royalties paid out to rights holders are substantially higher but so are the stores' prices[16]. Also, this business model has been around for much longer, so licensing is more complete. Download stores offer only low quality preview versions of content for free. On embedded players from SoundCloud, the rights holders often link them, so they are not part of the proof of concept implementation.

## 4.2 Choice of sources of critical or editorial review

In order to reduce the scope of implementation activity, sources of critical or editorial review are limited to those available as RSS feeds. Music magazines and blogs are sufficiently available in that format. Feeds from social networks such as Facebook and Twitter are omitted for this first iteration in order to avoid the complexity their interfaces introduce, as those are not essential to prove the viability of the proposed system.

## 4.3 Targeted users

Primarily those users who already discover music in a similar but not automated pattern will benefit from this system. Their profile includes the awareness of music authorities and an interest in discovery of new music. Readers of print and online magazines and followers of artists and bands on social networks like Twitter and Facebook fall into this

[16] Spotify pays artists only .0.4GBX (pence) per stream according to The Guardian (Musicians' Union demands new pay deal from Spotify 2013), while iTunes only takes a 30% cut of 0.99 USD track sales (What Does an Indie Get Paid? 2013), resulting in per track royalty ~0.60 USD or ~ 40 GBX per track sold, one hundred times as much.

category. Judging from the extensive use of embedded and linked musical content in these publications, being able to instantly consume the discovered music also plays a big role. Those interested in computer aided music discovery might be another group – just trying out a new way to switch on the radio might have made Last FM and many others quite popular. Since the proposed system would produce a different set of recommendations than the recommendation engines massively in use right now do, some users might be interested in trying it out. Together these groups can be characterized as the minority of music consumers who actively seek out and consciously listen to music.

## 4.4 User stories

User Stories as they would build a backlog in the Scrum methodology (Scrum.org | The home of Scrum > Home 2013) are here only used to describe the initial feature set targeted.

As a user:

- I want to specify sources of critical or editorial review so that I can get a playlist representing those sources music recommendations.
- I need a way to find RSS feeds for my preferred sources so I can subscribe to their music recommendations.
- I want to get tracks in my playlist that represent exactly what the author of the feed's post intended, so I can listen to what they recommend.
- I want to view all my music recommendations in one list, no matter which provider they are hosted on so that I can listen to them without switching apps or tabs.

# 5 Available technology and decisions

## 5.1 Desktop or mobile

A layer of complexity on top of the existing application logic and basic interface challenges the implementation of mobile applications. Mobile operating systems like Android[17], iOS[18], or Windows mobile[19] all require knowledge of a custom framework. Application development for mobile devices is further complicated by device limitations and restrictions of the mobile domain. In order to focus on the core functionality necessary for a proof of concept the desktop environment is chosen.

## 5.2 Platform

Three main platforms are to choose from: the native platform, an independent web application, or a web application as part of a third party application ecosystem. A web application seems most suitable due to the optimal choices of frameworks for the problem domain. Connecting to and communicating with resources on the Internet is at the core of the implementation. While a third party eco system application, for example Spotify's[20] would lend itself to the use case of music discovery, the catalogue to select from would be limited. YouTube for example could not be a music provider for this application anymore. An "app" in the context of Facebook, Twitter, or SoundCloud does not represent a platform choice but expresses an identity and gives access to their API. The added complexity of dealing with their APIs is not necessary for the proof of concept, so the independent web application is chosen.

## 5.3 Web framework

When it comes to the framework decision for a web application the power of the used scripting language, the amount of active development on the framework, its eco system, and the amount of applications deployed are important factors. Long-term viability

[17] http://www.android.com/

[18] http://www.apple.com/ios/

[19] http://www.microsoft.com/en-us/download/windowsMobile.aspx?q=windows+mobile

[20] The latest addition to the music discovery portfolio is the advent of Spotify Apps. These are essentially HTML, CSS & JavaScript driven web pages with access to Spotify content and functionality inside the client. Top charts, discovery, concerts & events, games, lyrics, reviews, and social are the main categories of this third party app eco system. Any piece or collection of music an also be shared with friends within Spotify, on social networks like Facebook or Twitter, on a Blog, or via email. (Spotify Apps API - Spotify Developer 2013)

comes to play here as well. Since a considerable learning curve is required for any newly learned language or framework, existing experience is also a valid factor. Problem specific features and libraries help the decision since they can speed up development significantly. Finally, some web frameworks impose licensing limitations that make them hard to use for open source projects. The author chooses Ruby on Rails, as a good fit with the decision factors mentioned, which will be described in the following.

### 5.3.1 Ruby

Ruby is an immensely powerful scripting language[21]. It is very well suited for use in a framework since it is flexible: the user can freely alter all its parts. Essentially a domain specific language can be created. Programming web applications is that specific domain for the Ruby on Rails framework.

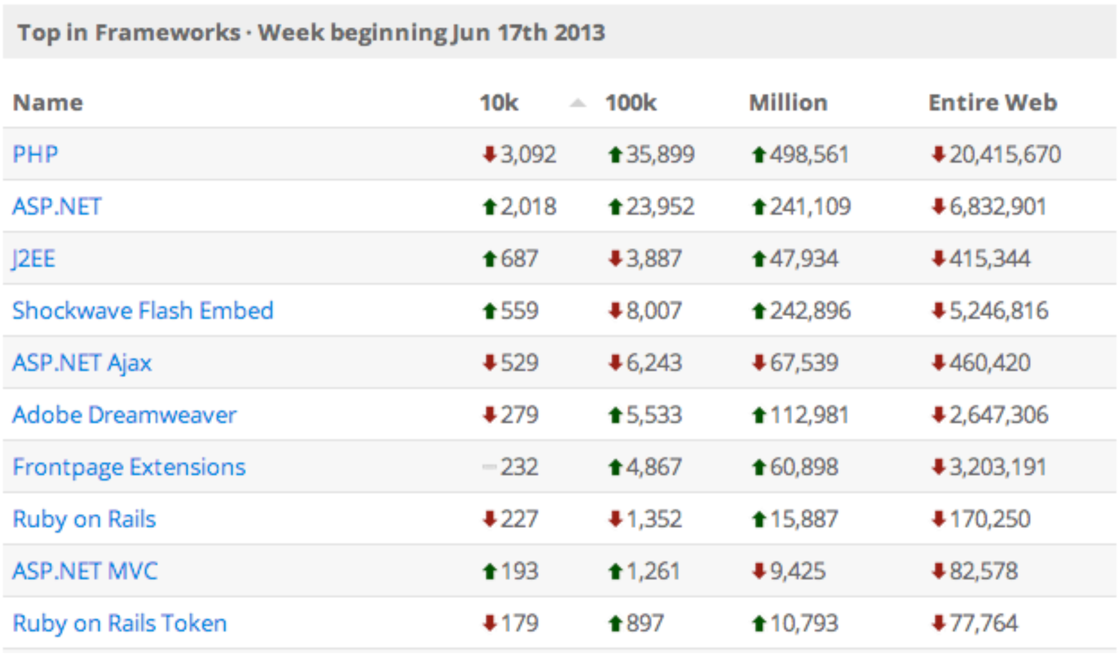
### 5.3.2 Ruby on Rails

Often referred to as "Rails", the open source web application framework is written in Ruby, and champions strong guiding principles: "Convention over configuration" results in strong assumptions on how standard problems in web applications are to be solved, speeding up the development process for those yielding to the conventions. DRY (Don't Repeat Yourself) means the framework is helping the developer to avoid code duplication, as first defined in the book "The Pragmatic Programmer" (Thomas und Hunt 1999). Model-View-Controller (MVC) architecture refers to the decoupling of business logic from the user interface. REST (Representational State Transfer) refers to organizing the application around resources and standard HTTP verbs such as GET and POST. (Getting Started with Rails — Ruby on Rails Guides 2013)

[21] "Its creator, Yukihiro "Matz" Matsumoto, blended parts of his favorite languages (Perl, Smalltalk, Eiffel, Ada, and Lisp) to form a new language that balanced functional programming with imperative programming." (ruby-lang 2013)

### 5.3.2.1 Popularity: Rails in production

Although at the bottom end of popular frameworks in production, Rails can still be called a popular framework.

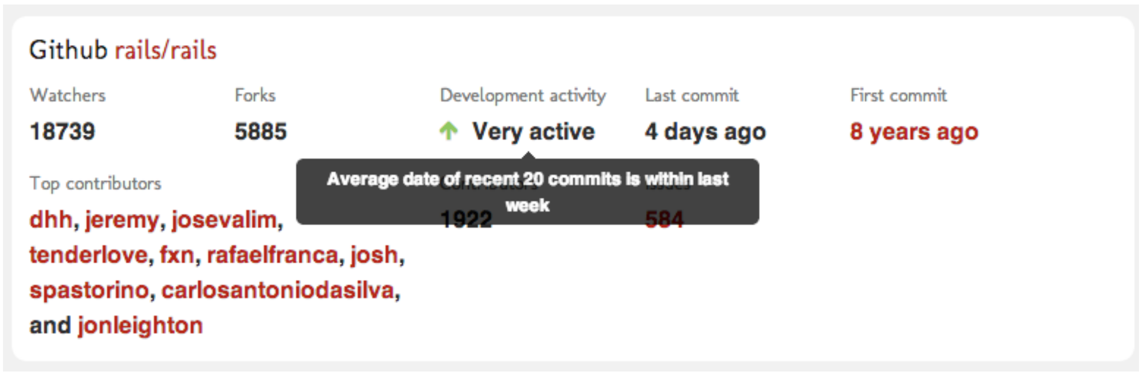

**Top in Frameworks · Week beginning Jun 17th 2013**

| Name | 10k | 100k | Million | Entire Web |
|---|---|---|---|---|
| PHP | ↓3,092 | ↑35,899 | ↑498,561 | ↓20,415,670 |
| ASP.NET | ↑2,018 | ↑23,952 | ↑241,109 | ↓6,832,901 |
| J2EE | ↑687 | ↓3,887 | ↑47,934 | ↓415,344 |
| Shockwave Flash Embed | ↑559 | ↓8,007 | ↑242,896 | ↓5,246,816 |
| ASP.NET Ajax | ↓529 | ↓6,243 | ↓67,539 | ↓460,420 |
| Adobe Dreamweaver | ↓279 | ↑5,533 | ↑112,981 | ↓2,647,306 |
| Frontpage Extensions | –232 | ↑4,867 | ↑60,898 | ↓3,203,191 |
| Ruby on Rails | ↓227 | ↓1,352 | ↑15,887 | ↓170,250 |
| ASP.NET MVC | ↑193 | ↑1,261 | ↓9,425 | ↓82,578 |
| Ruby on Rails Token | ↓179 | ↑897 | ↑10,793 | ↓77,764 |

Figure 3: Top 10k, 100k, million, and entire web websites deploying frameworks in production (Framework technologies Web Usage Statistics 2013)

### 5.3.2.2 Development activity

Rails is under very active development, with an average of at least 20 commits[22] per week. With 1922 contributors, the project can rely on broad community support.


Github rails/rails

| Watchers | Forks | Development activity | Last commit | First commit |
|---|---|---|---|---|
| 18739 | 5885 | ↑ Very active | 4 days ago | 8 years ago |

Top contributors

dhh, jeremy, josevalim, tenderlove, fxn, rafaelfranca, josh, spastorino, carlosantoniodasilva, and jonleighton

Average date of recent 20 commits is within last week

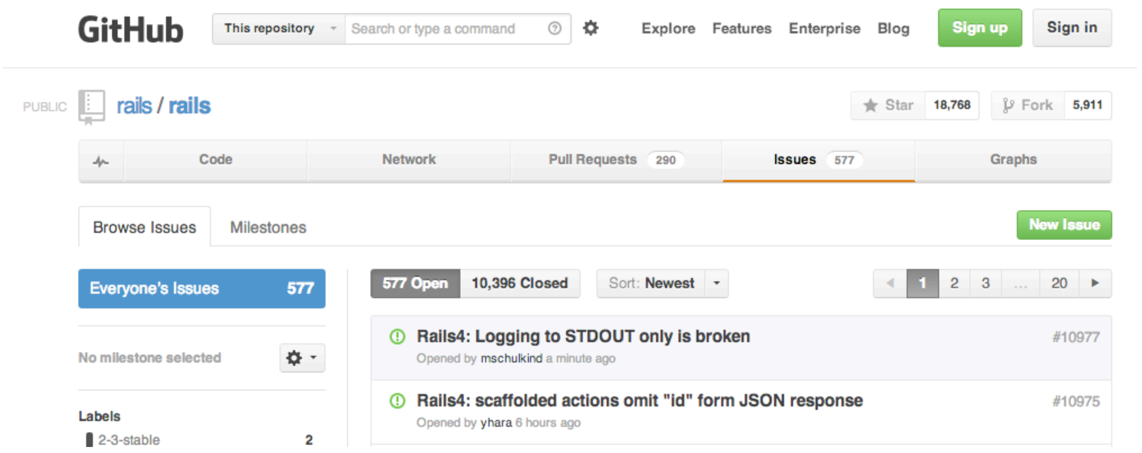
1922

584

Figure 4: Rails statistics on ruby toolbox (The Ruby Toolbox - Ruby on Rails 2013)

[22] Particular version of a program under GIT source control (Git 2013)

Its issue tracker shows a ratio of 577 open to 10396 closed issues, promising community support should a problem arise.



Rails4: Logging to STDOUT only is broken #10977
Opened by mschulkind a minute ago

Rails4: scaffolded actions omit "id" form JSON response #10975
Opened by yhara 6 hours ago

Figure 5: Open to closed issue ratio for rails (Issues · rails/rails 2013)

#### 5.3.2.3 Eco system

At the time of writing, 118,914 questions are tagged with Ruby on Rails on the popular Q&A website for developers Stackoverflow.com, creating an immense knowledge base for the framework. (Feed of questions tagged ruby-on-rails 2013). Domain specific editors and IDEs, performance monitors, hosting, consulting, conferences and workshops around Ruby on Rails complete the picture of a vibrant eco system. (Ruby on Rails: Ecosystem 2013)

#### 5.3.2.4 Libraries – gems

A software package that encapsulates a library or application is called a gem in Ruby. The hoster rubygems.org logged 1,714,851,778 downloads of 58,152 open source gems since July 2009 (rubygems.org 2013), providing a modular toolbox for an incredible amount of standard problems. At ruby-toolbox.com, these gems are ranked by category, and for each gem, resources that help the developer determine if this is the right library for the project are listed and linked. (The Ruby Toolbox - All Categories by name 2013)

#### 5.3.2.5 Use for proposed system

Ruby on Rails provides the basics for the proposed system – a webserver with a RESTful interface, code organization in the Model-View-Controller pattern, and an object oriented abstraction layer to database access, referred to as Object-Relational Mapping (ORM). Its strong ecosystem of open source libraries for common programming tasks makes it a great match for the proposed system, and its popularity and development activity provide long-term viability.

# 6 Design and architecture

## 6.1 Component outline

The system's user navigates to a view with a browser. There, the ability to search for RSS feeds that determine the playlist's content is presented. Once a few feeds are selected and submitted, the view sends feed URLs to the playlist's controller.

Now the feed objects are initialized by the controller by calling the model with these URLs, and persisted in the database through the model's ORM. Then the controller calls feed parsers that retrieve the feed content from webservers, and create post objects which are persisted via their model's ORM. They also call a post parser for each of these posts.

Post parsers are now using a variety of different strategies to extract keywords, and create queries for music provider APIs from the post's content. Service APIs for artist entity extraction, non-RSS feed content from webservers, and related releases listings are needed for this process. If successful, they give the model the attribute values it needs to create tracks and keywords. Keywords are not used in this iteration of the system but might be needed later; tracks are retrieved by the controller and served to the user in a view, where they can be played.

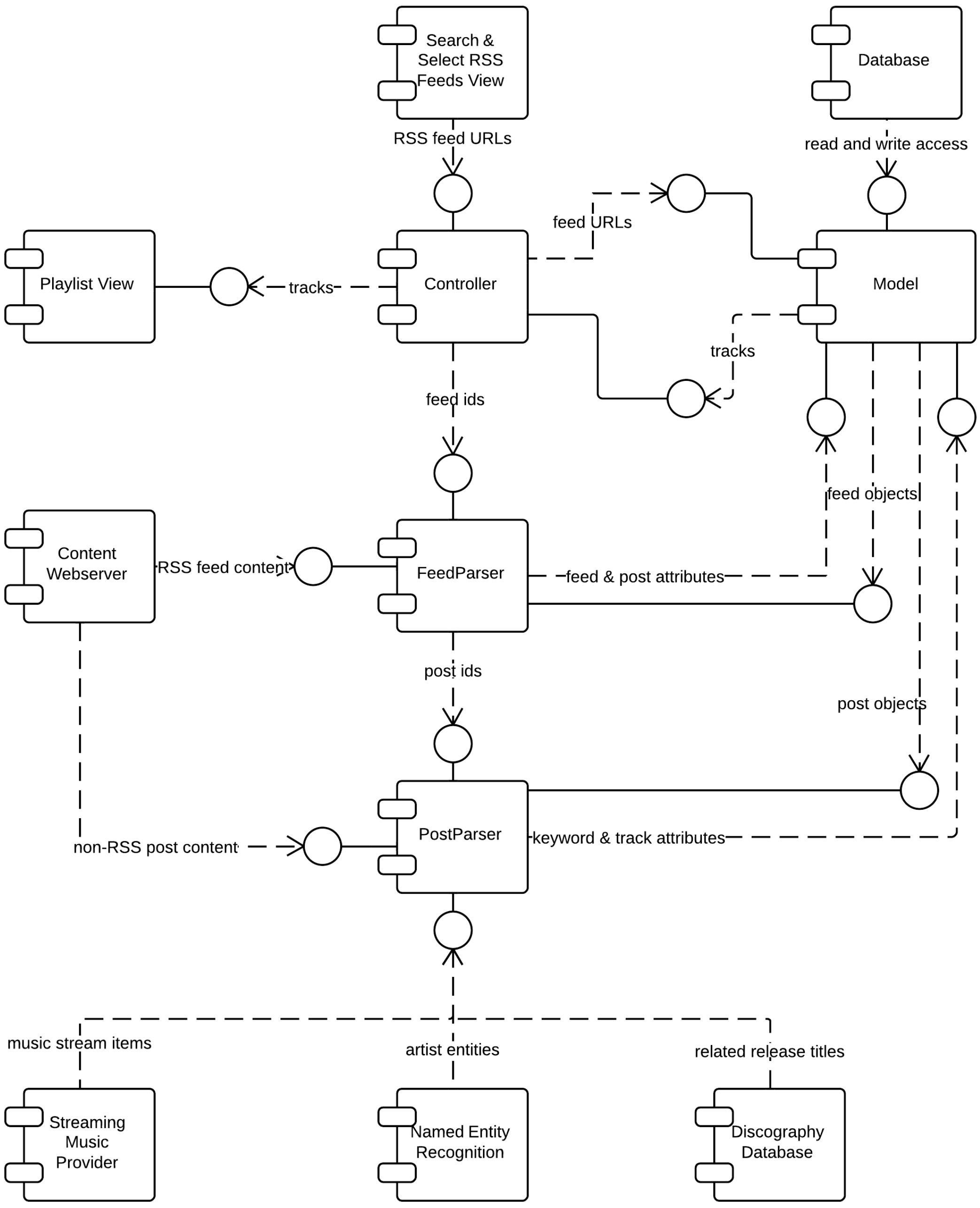


Figure 6: UML component diagram of high-level system architecture

## 6.2 Entity relationships

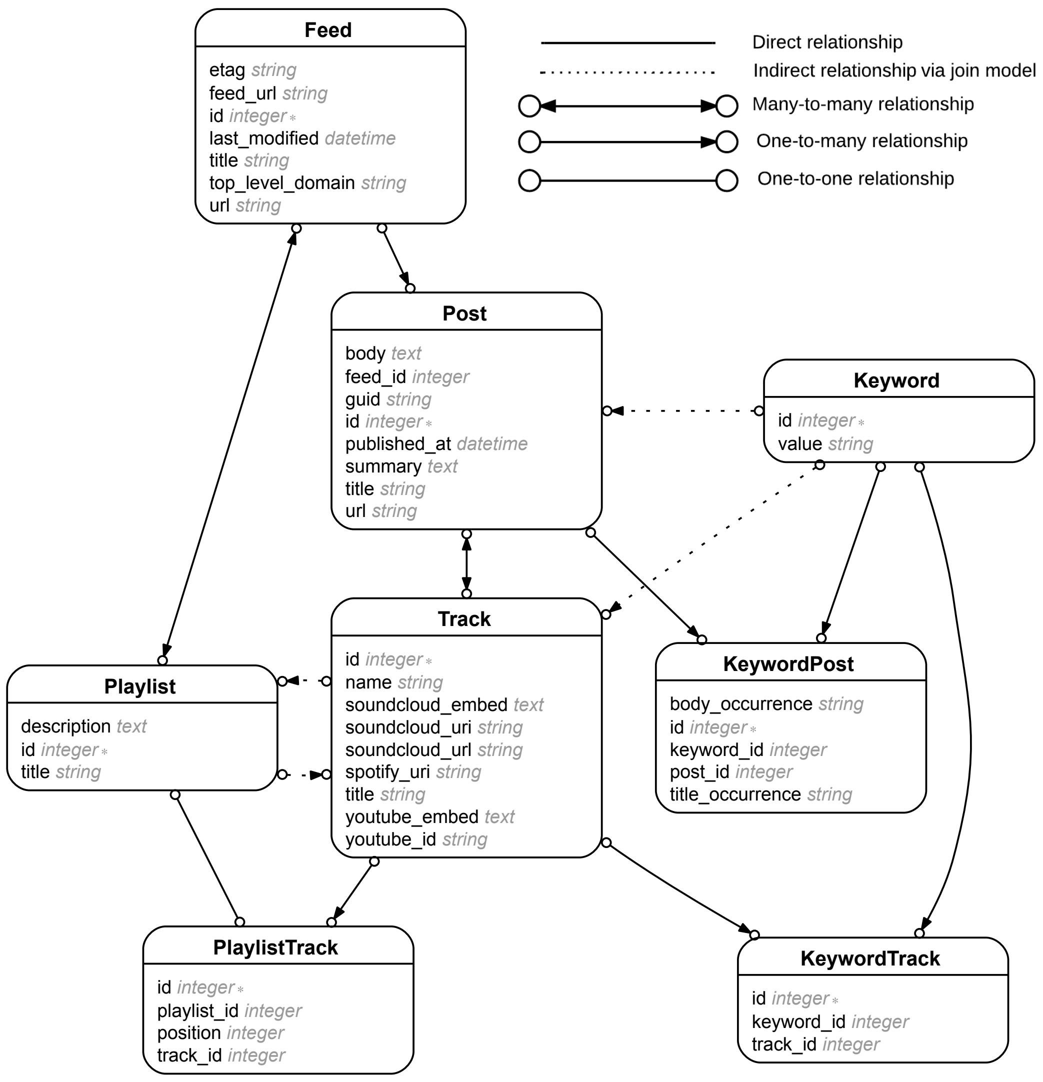


Figure 7: Database model in Bachman notation with attributes, generated with Rails-ERD (Rails ERD – Entity-Relationship Diagrams for Rails 2012)

A Playlist entity models the real world playlist concept, a persistent, ordered collection of tracks. A Feed entity represents an RSS feed. A Playlist can be related to many Feeds, with the inverse also being true. This way, a user can create multiple playlists with shared feed content. Together they form a many-to-many relationship. A Playlist is destroyed by the user. Related Feeds are destroyed via a callback on the lifecycle event unless they are still related to any other Playlists.

A Track models a playable piece of music. It can relate to many Playlists, again a mutu-al relationship. This is a consequence of the design decision to allow a single feed to appear in multiple playlists, but also accounts for the possibility that the same track might appear in different feeds. In order to keep record of the track position in each playlist and speed up access for display, the join model PlaylistTrack holds foreign keys to both Playlist and Track entities, along with the position information. PlaylistTracks are destroyed when their parent Playlists are.

While every Playlist can have only one associated collection of Tracks, and therefore is only associated with one PlaylistTrack, one Track can have many related Playlists, thus can also associate with many PlaylistTracks. The join model is destroyed when its parent Playlist is, also triggering a callback to destroy the related Track unless it is still associated with any Playlists.

A Post models the type of items an RSS feed is comprised of. One Feed can hold many Posts, which they uniquely belong to, forming a one-to-many relationship. When the parent Feed is destroyed, so are all its Posts. A Post's content can semantically reference many Tracks, which could also be referenced in other Posts as well, so a many-to-many relationship is useful.

A Keyword models a text reference to a playable piece of music, or Track. The relationship between Post and Keyword is created indirectly, with the help of the join model KeywordPost. This is chosen to be able to keep track of attributes this relationship might have, for example how many times a Keyword appears in a Post, or in which attribute, for example body or summary. It is possible for a keyword to appear in many posts, and mutually a post can hold many keywords. By associating one Post and one Keyword both with many KeywordPost join model instances, a many-to-many relationship is created. The join model is destroyed when its Keyword or its Post is. Along with it, it triggers a callback to clean up unreferenced Keywords, which destroys its associated Keyword unless it is still associated with another Track or Post.

A keyword that was used to create a successful query for a music provider resulting in a new track forms a useful relationship with that track, since a user might, or might not want to be presented with all tracks associated with that keyword. The association is created through the join model KeywordTrack, so attributes for this relationship could be added later easily. A many-to-many type is chosen, since multiple keywords could fit a given track, and multiple tracks could share certain keywords. KeywordTracks are destroyed with their Keyword or Track. Analogous to KeywordPost's behavior, in this event unreferenced Keywords are destroyed unless they are still associated with any Tracks or Posts.

```
class Track < ActiveRecord::Base

  # declaration of associations and lifecycle callbacks
  has_many :keyword_tracks , :dependent => :destroy
  has_many :keywords, :through => :keyword_tracks
  has_and_belongs_to_many :posts, :after_add => :update_playlists
  has_many :playlist_tracks, :dependent => :destroy
  has_many :playlists, :through => :playlist_tracks

  #save new track to all related playlists, using the post #object
  Rails invokes this callback as a parameter with.
  def update_playlists(post)
    # also the future place to check for user preferences that  #
     would reject the track for the playlist
    post.feed.playlists.each do |playlist|
      PlaylistTrack.create(playlist: playlist, track: self) unless
       playlist.tracks.exists?(self)
    end
  end
```

Figure 8: Track model snippet with declaration of associations and lifecycle callbacks

# 7 Implementation

Following the application's flow of control, notable implementation done is described in this chapter.

## 7.1 Google Feed API & JavaScript HTML form filling

This implementation chapter deals with the user stories:

As a user:

- I want to specify sources of critical or editorial review so that I can get a playlist representing those sources' music recommendations.
- I need a way to find RSS feeds for my preferred sources so I can subscribe to their music recommendations.

Two possible patterns are up for decision: A multi-step wizard with a search form that submits one feed result to the playlist's controller one view at a time, or a client side JavaScript powered view that gathers all desired feeds and submits them to the controller in bulk. As more clicks and more views shown to the user do not make for a better user experience, the client side pattern is chosen.

The view that is presented to the user in order to search for, select, and submit feeds to make up a playlist is called the playlist's "new" view in the following. Here, the user is first asked to enter search terms for those feeds. Once a term is entered, results from the Google Feed API (Google Feed API - Google Developers 2013) are presented. Those can be selected with a button. Entering a new search term clears the results, and makes it possible to restart this process and select as many feeds as desired.

An HTML form on the playlist's "new" view realizes this. It is created with the help of Rails, and JavaScript that handles the feed search, and fills in the form. When the user submits the form, the playlist controller's "create" action receives a parameter hash (here a hash is Ruby's associative array class) parsed from the data in JavaScript Object Notation (JSON) that was submitted via a HTTP POST request, all handled by the framework.

Now the controller creates a new playlist object along with related feed objects; these associated model instances are created parsing the format of the parameter hash received. If a key for attributes of an associated model is detected, and those attributes are whitelisted for mass assignment, and if the format matches the model association multiplicity, Rails knows to create the associated objects, and persist them.

```
def create
  # create playlist with title,
  # and related feeds with feed_urls
  @playlist = Playlist.new(params[:playlist])
```

Figure 9: Statement to create a playlist with associated feeds in a PlaylistsController instance method

```
 --- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
 utf8: ✓
 authenticity_token: MHWjDZgsh0Jeja5gYEOOSuvNTr1FqFdgX29CmnKR1Tg=
 playlist: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
   title: asdg
   feeds_attributes:
!ruby/hash:ActiveSupport::HashWithIndifferentAccess
     '0': !ruby/hash:ActiveSupport::HashWithIndifferentAccess
       feed_url: http://www.localsuicide.com/feed/
     '1': !ruby/hash:ActiveSupport::HashWithIndifferentAccess
       feed_url: http://hainbach.tumblr.com/rss
 commit: Create Playlist
 action: create
 controller: playlists
```

Figure 10: Parameter hash as received by the PlaylistController instance “create” action

The Playlist model instance method that the framework calls to create the related feeds and assign their attributes needs to be overridden in order to add already existing feeds to the playlist and avoid duplication.

```
def feeds_attributes=(hash)
  hash.each do |sequence,feed_values|
    feeds <<  Feed.find_or_create_by_feed_url(
      feed_values[:feed_url])
  end
end
```

Figure 11: Override of Playlist model instance method to create and assign associated model objects

In order to fill the parameter hash with values in this format, the JavaScript has to construct the form as Rails would. The syntax is reverse engineered by first placing a regular form on the view, filling in the field, and then inspecting the HTML element.

```
 <input id="playlist_feeds_attributes_0_feed_url"
name="playlist[feeds_attributes][0][feed_url]" size="30" type="text"
value="http://feeds.urbandictionary.com/UrbanWordOfTheDay">
```

Figure 12: HTML rendered form for associated feeds

Now the JavaScript handling the feed search and form filling can be implemented. Including Google’s (Google Feed API - Google Developers 2013) JavaScript[23] API file in the project provides the global variable “google”. After calling “.load” on it, a callback is triggered providing access to the Google Feed Api. With the help of jQuery (jQuery 2013) the value in the HTML search field element is accessed and used to query the Google Feed Api.

```
//load google feeds api v.1
google.load("feeds", "1", {
  callback: function () {
    var feedSearchValue = '';
    // read from search field on key up event
    $('#feed-search').on('keyup', function () {
      var currentValue = $(this).val();
      if (currentValue !== feedSearchValue) {
        feedSearchValue = $.trim(currentValue);
        if (feedSearchValue !== '') {
```

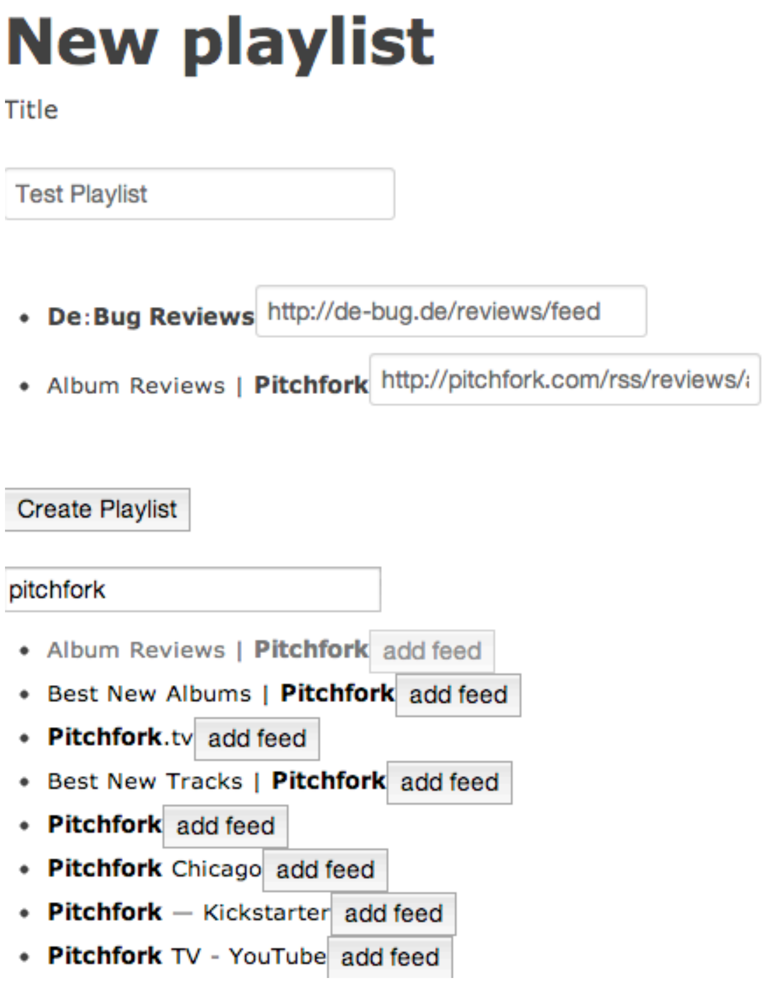
Figure 13: JavaScript snippet to access Google Feed API, and the view’s search form with jQuery

The returned results are appended as elements to the view’s HTML, including a button. This button’s event handler appends the selected feeds’ URLs to the HTML mimicking a filled in form element as expected by Rails.

[23] “JavaScript (JS) is an interpreted computer programming language.It was originally implemented as part of web browsers so that client-side scripts could interact with the user, control the browser, communicate asynchronously, and alter the document content that was displayed. More recently, however, it has become common in both game development and the creation of desktop applications.” (JavaScript - Wikipedia, the free encyclopedia 2013)

```
// seach google feeds, encapsulate each request
google.feeds.findFeeds(feedSearchValue,  (function (searchValue) {
  return function findfeedscallback (result) {
    // if the user does not change the request display the results
    if (searchValue === $('#feed-search').val()) {
      if (!result.error) {
        var entry,
          i;
        $('#feed-search-error').empty();
        $('#feed-search-results').empty();
        for (i = 0; i < result.entries.length; i += 1) {
          entry = result.entries[i];
          $('#feed-search-results').append('<li><a href="' + entry.url
            + '">' + entry.title
            + '</a><button name="add-feed">add feed</button></li>');
        }
        // button action to add result to params hash
        $('#feed-search-results button[name="add-feed"]').on(
          'click', function () {
          var index = $('form ul li').length;
          $(this).attr('disabled', 'disabled');
          $('form ul').append('<li>' + $(this).siblings('a').html()
            + '<input id="playlist_feeds_attributes_' + index
            + '_feed_url" name="playlist[feeds_attributes]['
            + index + '][feed_url]" size="30" type="text" value="'
            + $(this).siblings('a').attr('href') + '"></li>');
          $('form button[type="submit"]').show();
        });
```

Figure 14: JavaScript snippet to display feed results for user selection and populate the HTML form


New playlist

Title

Test Playlist

- De:Bug Reviews http://de-bug.de/reviews/feed
- Album Reviews | Pitchfork http://pitchfork.com/rss/reviews/a

Create Playlist

pitchfork

- Album Reviews | Pitchfork add feed
- Best New Albums | Pitchfork add feed
- Pitchfork.tv add feed
- Best New Tracks | Pitchfork add feed
- Pitchfork add feed
- Pitchfork Chicago add feed
- Pitchfork — Kickstarter add feed
- Pitchfork TV - YouTube add feed

Figure 15: The playlist's "new" view showing selected Google Feed API results

## 7.2 RSS feed parsing

The feeds' content now needs to get fetched, its XML structure parsed, and its data stored for access by the system's algorithms. The gem Feedzirra offers the functionality to fetch and parse a feed, returning an object that provides access to all feed attributes. (pauldix/feedzirra · GitHub 2013)

From the PlaylistController's "create" action, a FeedParser gets instantiated, and its parse method is called. By passing the Feedzirra feed object to the Feed model's update method, these attributes are persisted. With a regular expression, the top-level domain is also extracted from the feed's URL for possible display and access by post parsing algorithms.

Now a Post model class method creates instances for each feed entry, populates their attributes, and saves them to the database. It returns an array of only those Post instances that are newly created, comparing the guid property with already existing Post instances; each of those are passed as arguments to PostParser instances.

## 7.3 Post parsing strategies

During implementation, different strategies to extract keywords, construct queries for music providers, filter responses, and create tracks were developed and tested. The underlying goal is to implement this user story:

- As a user, I want to get tracks in my playlist that represent exactly what the author of the feed's post intended, so I can listen to what they recommend.

### 7.3.1 Deserted strategies

#### 7.3.1.1 Parsing with pattern matching

An early idea was to construct regular expressions to match patterns in a title or sentence of a post. Capitalized words following certain characters could indicate a named entity. Though by far not accurate enough, the idea led to finding out about named entity extraction and The Echo Nest.

#### 7.3.1.2 Using press information

The Echo Nest's ability to provide news, reviews, and blog articles for a found artist entity suggested its use to validate extracted entities. This is necessary since there is a large error margin in the returns of The Echo Nest's "extract artist (beta)" API procedure; words in a post's title might not semantically refer to the artist found by the service in that post's context. If an entity was included in a past news, review, or blog post by the outlet publishing the feed, the entity detection could be validated with some probability. Unfortunately this method would not be post accurate and only provide a statistical value. Also the blog, news, and reviews articles returned from The Echo Nest do not have sufficient media coverage: All of the articles on the outlet of the post parsed would need to have been analyzed by The Echo Nest previously.

#### 7.3.1.3 Filter out keywords that do not get repeated in the post body

Validating extracted entities based on their occurrence throughout the post seemed like a good idea initially. That is, until it became apparent that very common words are used as artist names, and sometimes posts would only mention the subject artist only once.

#### 7.3.1.4 Website scraping

Libraries to navigate websites, and extract contents based on CSS or XPath properties exist, and would make it feasible to identify the elements of a website linked by the post that holds artist, or artist-and-release-title information. This approach would be completely specific to a feed though, making it necessary to write a scraper for every feed selected as input. Also any change to the CSS or HTML structure of that website would break the scraper, which led to deserting this strategy.

#### 7.3.1.5 Semantic HTML

Guidelines for semantic markup in HTML and CSS from the W3C exist (RDF Semantics 2013). If those would be implemented by website makers, an algorithm to extract the information offered would be the most accurate, fast, and successful of all. Unfortunately, none of the tested websites did, so that strategy was deserted.

### 7.3.2 Employed strategies

All strategies are described with a flowchart visualizing what is happening and text describing how and why it is. Selected code snippets illustrate the implementation, and a UML sequence diagram shows the interaction of the entities.

#### 7.3.2.1 HTML parsing for embedded content

The first strategy applied is to attempt to extract embedded content from the post, or the website it links to. The most direct way in which an author can express a machine-readable music recommendation is to include a player of a track, or a set of tracks in her post. While text may be ambiguous, this clearly represents what the author recommends to the reader for listening.

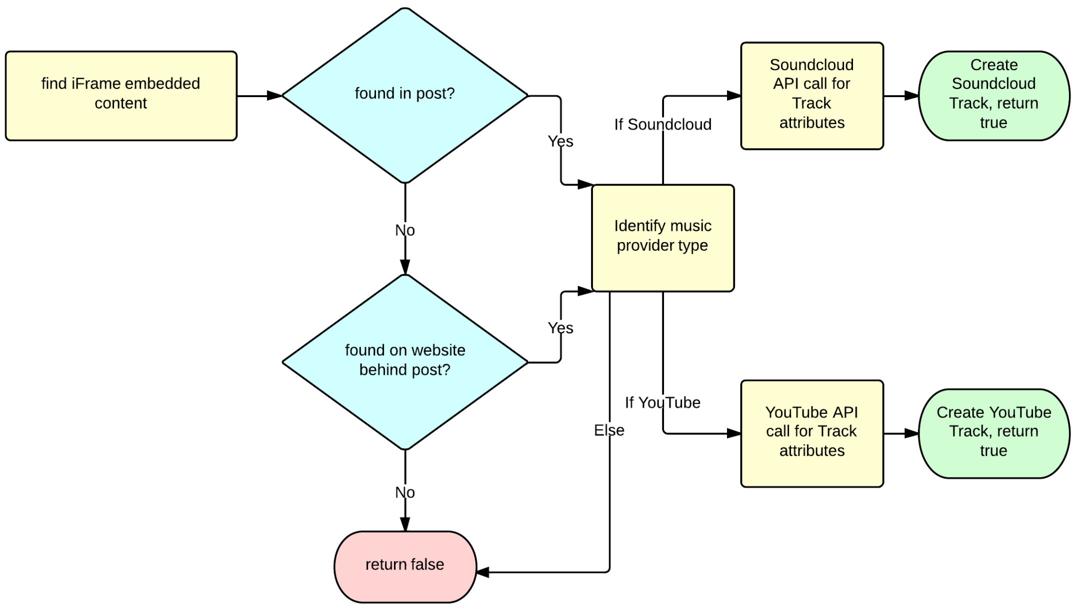

find iFrame embedded content
found in post?
Yes
No
found on website behind post?
Yes
No
return false
Identify music provider type
If Soundcloud
If YouTube
Else
Soundcloud API call for Track attributes
Create Soundcloud Track, return true
YouTube API call for Track attributes
Create YouTube Track, return true

Figure 16: Flowchart for parsing strategy to extract embedded content

#### Find iFrame[24] embedded content

[24] "The <iframe> tag specifies an inline frame.
An inline frame is used to embed another document within the current HTML document." (HTML iframe tag no date)

In a first step, the PostParser attempts to find embedded content in the Post's summary attribute. An object of the class responsible for HTML parsing is instantiated. Its constructor is passed the summary attribute's value containing the HTML, and called to extract player URLs from iFrames found in there. This is done with the help of the Nokogiri gem and its ability to locate and extract properties of HTML tags. "Nokogiri is an HTML, XML, SAX[25], and Reader parser. Among Nokogiri's many features is the ability to search documents via Xpath or CSS3 selectors." (Nokogiri 2013) The source values of iFrame tags found are returned as an array. This process is repeated with the post's body property and the results are appended to the array, unless the body property is nil; the body property is often omitted in feeds. This array is processed by a private helper method of the PostParser instance in order to create and persist the Tracks represented by the embedded content using Rails' ORM.

### Identify music provider type

For this, regular expressions are constructed to match iFrame source values that represent supported player types, again via a private helper method.

```
@re_SoundCloud = /(api\.SoundCloud\.com[^&]*)/
```

Figure 17: Regular expression to match an iFrame source's SoundCloud URI

For the first iteration of the system, the most common formats of SoundCloud, and YouTube iFrame embed codes, in the following called "embeds" are supported. Depending on the detected player type, the attributes necessary to create a Track are extracted from the iFrame's source attribute value.

### SoundCloud API call for track attributes

For a SoundCloud embed, this is the track's URI. The SoundcloudProvider class wraps the access to the SoundCloud API, and provides a method to resolve the URI to a SoundCloud track object, containing attributes needed to create a Track instance.

### YouTube API call for track attributes

Embedded content from YouTube follows a similar pattern: the video's identifier parsed from the iFrame value is used to request a video object from the YouTube API. This identifier is then also passed to the Track model since a YouTube video object does not offer the video identity property that gets saved by the model.

### Create track

[25] "SAX is the Simple API for XML, originally a Java-only API. SAX was the first widely adopted API for XML in Java, and is a "de facto" standard. The current version is SAX 2.0.1, and there are versions for several programming language environments other than Java." (SAX 2004)

This does create Track objects with different attributes set. Should in a future iteration the ability to present all tracks to a user on one provider platform become necessary, fitting converter methods would have to be implemented. For this system iteration however, the most direct expression of the author's intention was valued higher than Track attribute consistency, and in order to play a track only one provider suffices.

If no, or no supported embedded content is found, the next step is to repeat this process, but this time parsing the HTML returned when issuing a get request for the URL associated with the post.

In a future iteration of this system, an algorithm to extract keywords such as artist names from the titles of the created tracks should be implemented, so tracks desired, or not desired by the user can be filtered out. Since this is not necessary for track creation with this strategy, it is left in the backlog for the next iteration.

```
# strategy: first extract from body and summary feed fields. if none
#present head over to the post.url and check there.
def extract_tracks_from_embeds
  player_urls = []
  player_urls =
   HtmlParser.new(@post.summary).extract_player_urls_from_iframes
  player_urls.concat(HtmlParser.new(@post.body).
   extract_player_urls_from_iframes) unless @post.body.empty?
  unless player_urls.empty?
    unless (create_tracks_from_player_urls(player_urls))
      # have to head over to the website to check for embeded content
      # all player_urls were not supported yet
      if (create_tracks_from_embeds_on_website_behind_post)
        return true
      else
        return false
      end
    else
      return true
    end
  else
    # no player_urls where found in neither summary nor body, so head
     #over to the website to check for embeded content
    if (create_tracks_from_embeds_on_website_behind_post)
      return true
    else
      return false
    end
  end
  return false
end
```

Figure 18: PostParser method for "HTML parsing for embedded content" strategy

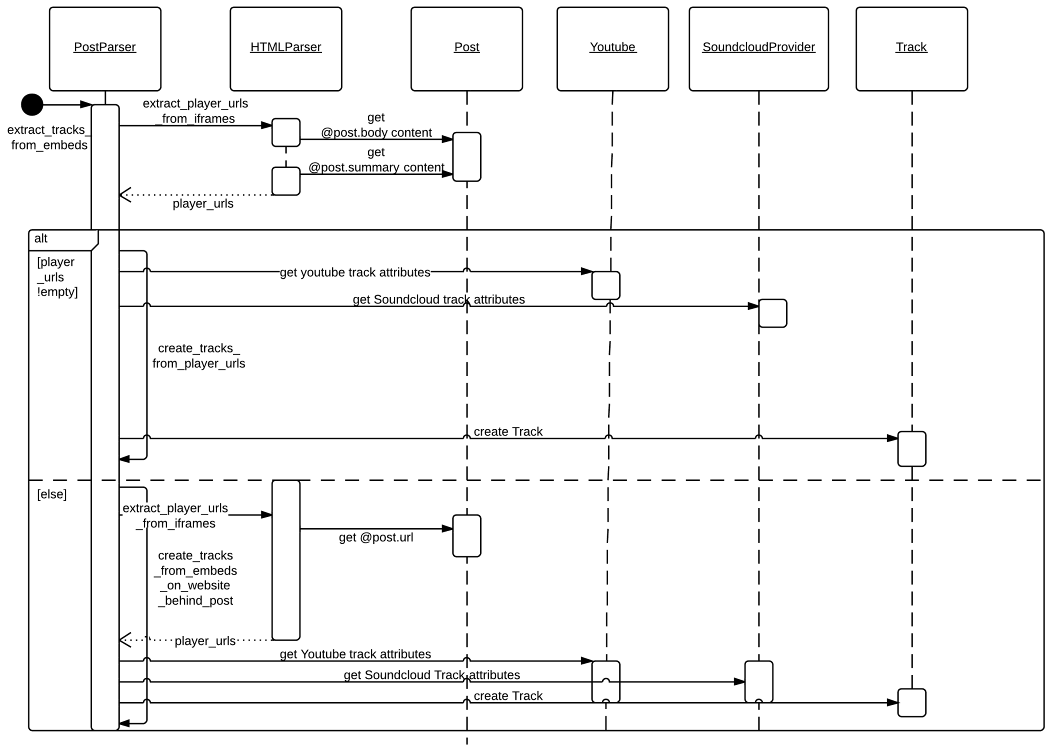
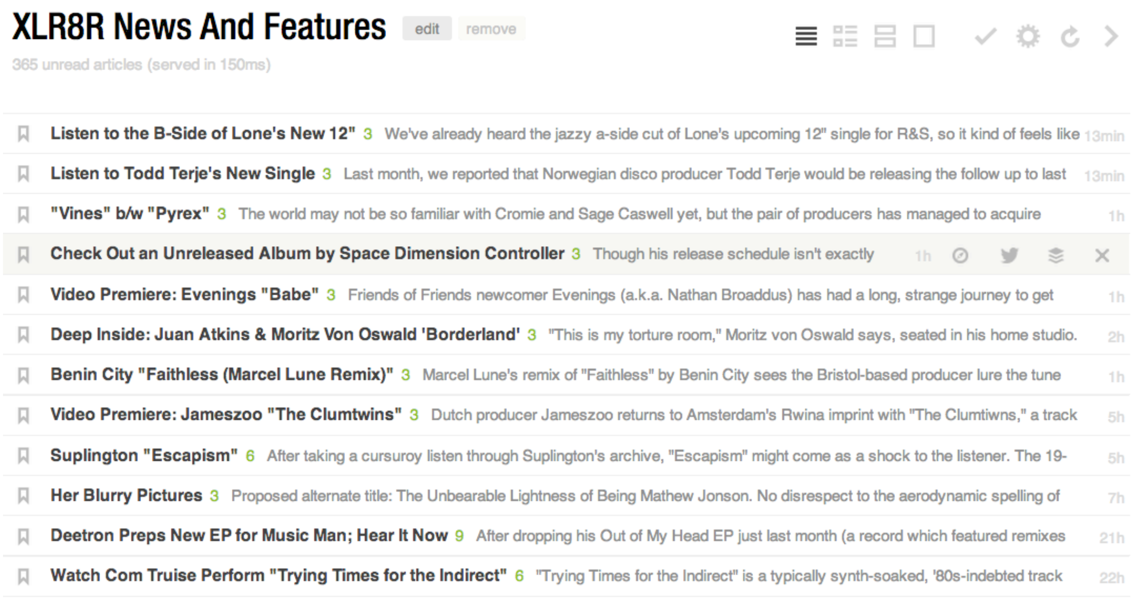

PostParser
HTMLParser
Post
Youtube
SoundcloudProvider
Track
extract_tracks_from_embeds
extract_player_urls from_iframes
get @post.body content
get @post.summary content
player_urls
alt
[player_urls !empty]
get youtube track attributes
get Soundcloud track attributes
create_tracks_from_player_urls
create Track
[else]
extract_player_urls_from_iframes
get @post.url
create_tracks_from_embeds_on_website_behind_post
player_urls
get Youtube track attributes
get Soundcloud Track attributes
create Track

Figure 19: UML Sequence Diagram: HTML parsing for embedded content strategy

#### 7.3.2.2 Google reverse image search

The second strategy applied attempts to identify which track or collection of tracks an author mentions in a post based on the image specified by the open graph tag “og:image” on the website linked by the post. Many music magazines offer structured data through RSS and open graph meta tag properties, making their content usable as input to this parsing strategy. “The Open Graph protocol enables any web page to become a rich object in a social graph. For instance, this is used on Facebook to allow any web page to have the same functionality as any other object on Facebook. […] og:image - An image URL which should represent your object within the graph.” (The Open Graph Protocol 2013)

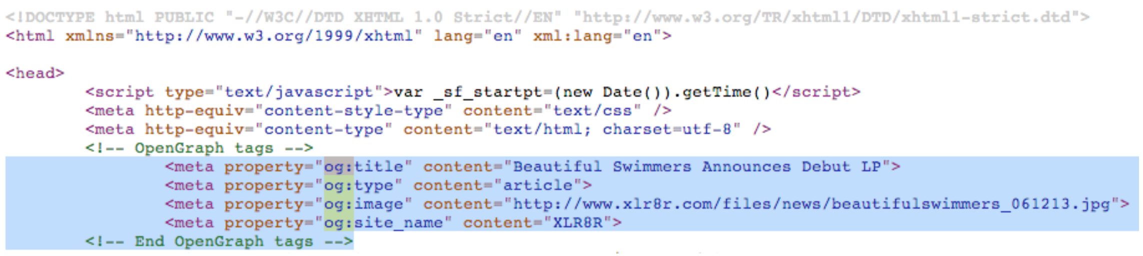


**XLR8R News And Features** edit remove

365 unread articles (served in 150ms)

| | | | |
|---|---|---|---|
| **Listen to the B-Side of Lone's New 12"** | 3 | We've already heard the jazzy a-side cut of Lone's upcoming 12" single for R&S, so it kind of feels like | 13min |
| **Listen to Todd Terje's New Single** | 3 | Last month, we reported that Norwegian disco producer Todd Terje would be releasing the follow up to last | 13min |
| **"Vines" b/w "Pyrex"** | 3 | The world may not be so familiar with Cromie and Sage Caswell yet, but the pair of producers has managed to acquire | 1h |
| **Check Out an Unreleased Album by Space Dimension Controller** | 3 | Though his release schedule isn't exactly | 1h |
| **Video Premiere: Evenings "Babe"** | 3 | Friends of Friends newcomer Evenings (a.k.a. Nathan Broaddus) has had a long, strange journey to get | 1h |
| **Deep Inside: Juan Atkins & Moritz Von Oswald 'Borderland'** | 3 | "This is my torture room," Moritz von Oswald says, seated in his home studio. | 2h |
| **Benin City "Faithless (Marcel Lune Remix)"** | 3 | Marcel Lune's remix of "Faithless" by Benin City sees the Bristol-based producer lure the tune | 1h |
| **Video Premiere: Jameszoo "The Clumtwins"** | 3 | Dutch producer Jameszoo returns to Amsterdam's Rwina imprint with "The Clumtiwns," a track | 5h |
| **Suplington "Escapism"** | 6 | After taking a cursuroy listen through Suplington's archive, "Escapism" might come as a shock to the listener. The 19- | 5h |
| **Her Blurry Pictures** | 3 | Proposed alternate title: The Unbearable Lightness of Being Mathew Jonson. No disrespect to the aerodynamic spelling of | 7h |
| **Deetron Preps New EP for Music Man; Hear It Now** | 9 | After dropping his Out of My Head EP just last month (a record which featured remixes | 21h |
| **Watch Com Truise Perform "Trying Times for the Indirect"** | 6 | "Trying Times for the Indirect" is a typically synth-soaked, '80s-indebted track | 22h |

Figure 20: RSS feed for the music magazine XLR8R



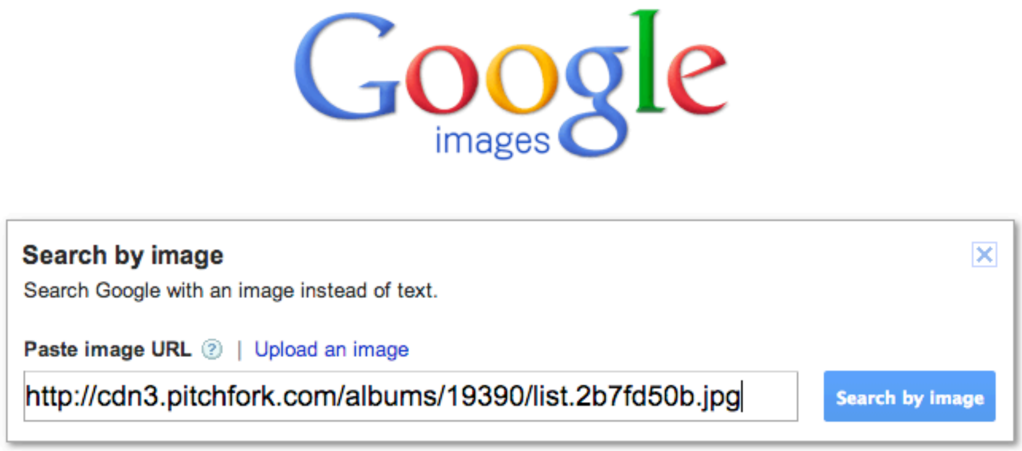
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

<head>
        <script type="text/javascript">var _sf_startpt=(new Date()).getTime()</script>
        <meta http-equiv="content-style-type" content="text/css" />
        <meta http-equiv="content-type" content="text/html; charset=utf-8" />
        <!-- OpenGraph tags -->
                <meta property="og:title" content="Beautiful Swimmers Announces Debut LP">
                <meta property="og:type" content="article">
                <meta property="og:image" content="http://www.xlr8r.com/files/news/beautifulswimmers_061213.jpg">
                <meta property="og:site_name" content="XLR8R">
        <!-- End OpenGraph tags -->
```

Figure 21: Open Graph meta property on music magazine article website source code

Given the presence of the tag, and assuming that the post’s author made a conscious decision for this image to represent that post, the “og:image” tag can express the subject of the post. If the image chosen is an artist picture or cover artwork of a music product such as a CD, LP, or download a music recommendation subject is assumed.

Google reverse image search is a web application feature that takes an image URL as input, and returns information about that image.



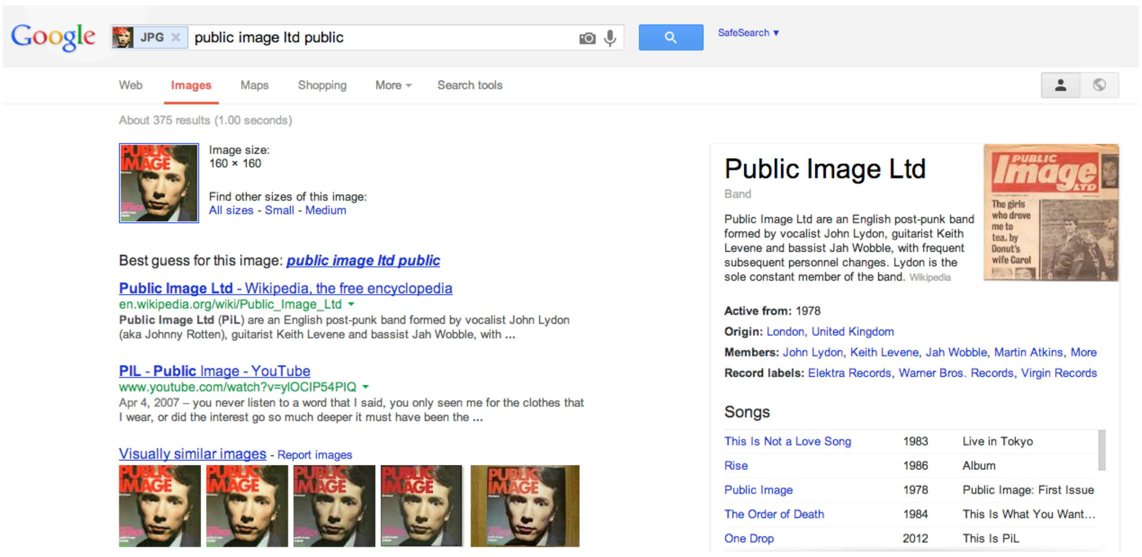
**Search by image**

Search Google with an image instead of text.

**Paste image URL** | Upload an image

http://cdn3.pitchfork.com/albums/19390/list.2b7fd50b.jpg

Figure 22: Google form to submit og:image URL (Google Images 2013)



public image ltd public

About 375 results (1.00 seconds)

Image size:
160 × 160

Find other sizes of this image:
All sizes - Small - Medium

Best guess for this image: ***public image ltd public***

**Public Image Ltd** - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Public_Image_Ltd
**Public Image Ltd** (**PiL**) are an English post-punk band formed by vocalist John Lydon (aka Johnny Rotten), guitarist Keith Levene and bassist Jah Wobble, with ...

**PIL** - **Public** Image - YouTube
www.youtube.com/watch?v=ylOCIP54PIQ
Apr 4, 2007 – you never listen to a word that I said, you only seen me for the clothes that I wear, or did the interest go so much deeper it must have been the ...

Visually similar images - Report images

Public Image Ltd
Band

Public Image Ltd are an English post-punk band formed by vocalist John Lydon, guitarist Keith Levene and bassist Jah Wobble, with frequent subsequent personnel changes. Lydon is the sole constant member of the band. Wikipedia

**Active from:** 1978
**Origin:** London, United Kingdom
**Members:** John Lydon, Keith Levene, Jah Wobble, Martin Atkins, More
**Record labels:** Elektra Records, Warner Bros. Records, Virgin Records

Songs

| | | |
|---|---|---|
| This Is Not a Love Song | 1983 | Live in Tokyo |
| Rise | 1986 | Album |
| Public Image | 1978 | Public Image: First Issue |
| The Order of Death | 1984 | This Is What You Want… |
| One Drop | 2012 | This Is PiL |

Figure 23: Google search results from reverse image search (Google Images 2013)

Google's "Best guess for this image" result turns out to be very useful, describing the image with <artist> <release title> if it holds cover artwork, or just <artist name> if an artist picture is supplied. This is exploited in the implementation described in the following.

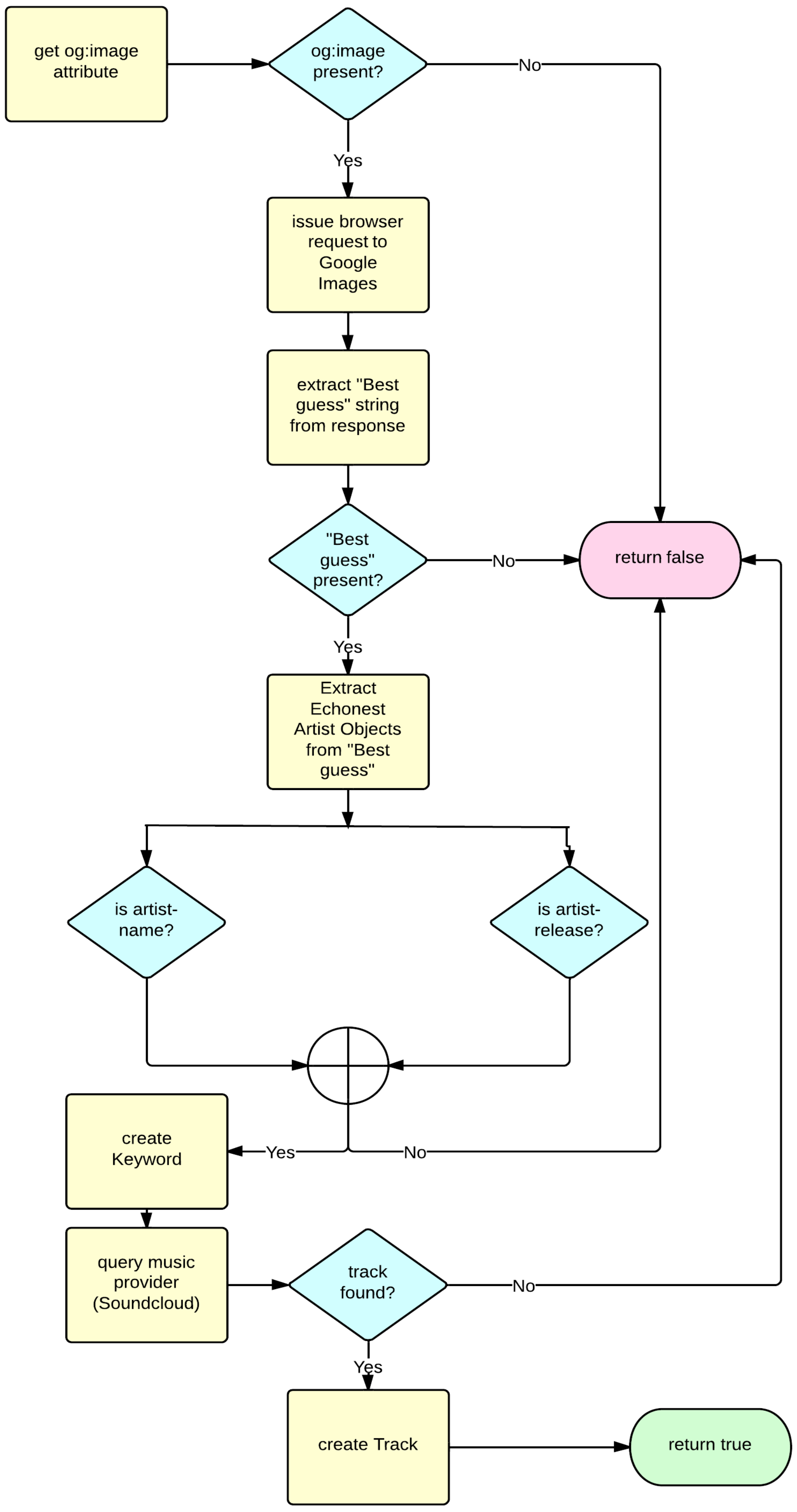


Figure 24: Flowchart for "Google reverse image search" strategy

### Get “og:image” attribute

In order to retrieve the URL in the “og:image” tag on the webpage linked by the post, the HTTParty gem is used. This library provides functionality to make HTTP requests and provides a parsed response. (jnunemaker/httparty - GigHub 2011). In the PostParser instance method “create_tracks_for_coverart” a HtmlParser instance is called to extract the “og:image” property’s value from the HTML response. The HtmlParser instance gets initialized with the HTML document, and there parsed into a node structure with Nokogiri. The PostParser calls its “query_string_for_coverart” method. There the HtmlParsers “get_coverart_url” helper method is called, extracting the URL of the og:image tag’s value.

### “Og:image” present?

Unless the property is not set, the cover art URL is returned to the “query_string_for_coverart_image” method of the HtmlParser, else false is returned, which is handed down to the PostParser’s “create_tracks_for_coverart method”, letting the strategy return its value for unsuccessful execution.

### Issue browser request to Google

Unfortunately, Google does not offer programmatic access to their reverse image search feature, so the system has to make the request by acting as a browser agent, analogous to a blog post found on Skyzerblogger (Schaback 2013). The HTTP GET request is constructed from a base URL that specifies which feature is requested, followed by a parameter specifying the input image URL, followed by the image URL. The header of the request needs to identify the caller as a browser for this to work so the user agent parameter is set accordingly, using HTTParty for convenience.

```
google_reverse_image_search_base =
      "https://www.google.com/searchbyimage?&image_url="
user_agent =
      "Mozilla/5.0 (Windows NT 6.1; WOW64)"
      + "AppleWebKit/537.11 (KHTML, like Gecko)"
      + "Chrome/23.0.1271.97 Safari/537.11"
response =
      HTTParty.get(
            google_reverse_image_search_base
            + coverart_url,
            :headers => {"User-Agent" => user_agent})
```

Figure 25: Request to Google Images acting as browser agent

### Extract “Best guess” string from response

The HTML response is then parsed by Nokogiri, which makes it possible to extract the div[26] element containing the “Best guess” text via XPath query, and in turn find the text value needed with Nokogiri’s ability to access elements in an HTML tag in a node structure.

```
# best guess element present?
if ( best_guess_div  =
 _doc.xpath('//div[contains(text(), "Best guess")]') )
  # return the text value
  return best_guess_div.children.children.to_s
else
   return false # no best guess from google found
end
```

Figure 26: XPath query called on Nokogiri “_doc” to find the div element containing "Best guess" and extract its text value

### “Best guess” string present?

Before the result can be passed on as the “Best guess” string, some validation is needed. Google returns an error page if the same image is searched for twice in a short time or the network is unresponsive. That results in a nil value for the attribute extracted Google might not have a “Best guess” result for the image, in which case the div does not exist. These cases are tested for. If there is no element with the “Best guess” text, the strategy has failed and false is passed down the call chain to the post parser.

### Extract The Echo Nest artist objects from “Best guess” string

If the test passes, the value for “Best guess” is passed back to the PostParser. Here EchonestApi ‘s class method “extract_artist_objects_from_string(query_string)” is called. The Echo Nest’s API functionality “extract artist (beta)” (Artist API Methods - The Echo Nest 4.2 documentation 2013) is called with a GET request through HTTPParty. The functionality identifier, API key and text to parse need to be specified. Since network requests can fail for various reasons, the call is wrapped in a begin-rescue block, which retries the request twice. In case of a network or The Echo Nest related error the HTTP response status code is not 0, so then an exception of the RuntimeError class is raised to enter the retry loop. Unless no artist was found, a Ruby hash containing the artists and their attributes found is returned.

[26] Tag that defines a division or section of a HTML document

```
def self.extract_artist_objects_from_string(query_string)
  #construct the API call.
  base_uri = 'http://developer.echonest.com/api/v4'
  api_type = 'artist'
  api_method = 'extract'
  api_key = 'ZHRJX1PLWUWJZRIL8'
  #grab artist related press
  buckets = '&bucket=news&bucket=blogs&bucket=reviews&bucket=songs'
  # wrap request url in proc for re-use with different queries
  api_request_url =
    Proc.new{|text| base_uri + '/' + api_type + '/'
      + api_method + '?' + 'api_key='+ api_key
      + '&format=json' + "&text=#{CGI.escape(text)}"
      + '&results=10' + buckets}
  retry_count = 0
  # wrap api call related code in proc to retry
  api_call = Proc.new do
    response = HTTParty.get(api_request_url.call(query_string))
    if (response['response']['status']['code'] != 0)
      raise "failed echonest api call with \
       response #{response['response'].to_yaml}"
    end
    artists = response['response']['artists']
    return artists unless artists.nil?
  end
  begin
    api_call.call
  rescue RuntimeError => e
    #retry api request
    retry_count++
    sleep(1)
    retry unless (retry_count > 2)
  end
end
```

Figure 27: Code to extract The Echo Nest artist objects from Google's "Best guess" string

To make it possible to add features that enable users to follow / unfollow a certain artist, or attributes related to that artist such as genre, the system needs to know which artist is found in the “Best guess” text. Also, it is not known if the “og:image” specified for the post parsed actually represents something that can be used to query a provider for music. The conditions that make it usable need to be tested. For this a Named Entity Recognition (NER) system is needed.

“You’d need a huge database of artists (check, we have over 1.6 million), a lot of fast computers (check), and tons of rules learned from our customers over the years about artist resolution— aliases, stopwords, tokenization, merged artists and so on.” (The Echonest Blog 2011)

Jurafsky and Martin describe the basic steps to create a named entity system: A representative document collection similar to the text to parse is compiled, and annotated by hand. Then features of that text are extracted.

| Feature | Explanation |
|---|---|
| Lexical Items | The token to be labled |
| Stemmed lexcial items | Stemmed version of the target token |
| Shape | The ortographic pattern of the target word |
| Character affixes | Character-level affixes of the target and surrounding words |
| Part of speech | Part of speech of the word |
| Syntactic chunk labels | Base-phrase chunk label |
| Gazetteer or name list | Presence of the word in one or more named entity lists |
| Preditive token(s) | Presence of predictive words in surrounding text |
| Bag of words/Bag of N-grams | Words and/or N-grams occuring in the surrounding context |

Figure 28: Features commonly used in training named entity recognition systems (Jurafsky und Martin 2009, 731)

IOB encoding is done - marking words that are inside the target set, outside, and on the boundary. This training data is then used to train classifiers to perform multiway sequence labeling. Maximum Entropy Markov (MEMM), Conditional Random Field (CRF), Support Vector Machine (SVM), and Hidden Markov models are such classifiers. (Jurafsky und Martin 2009)

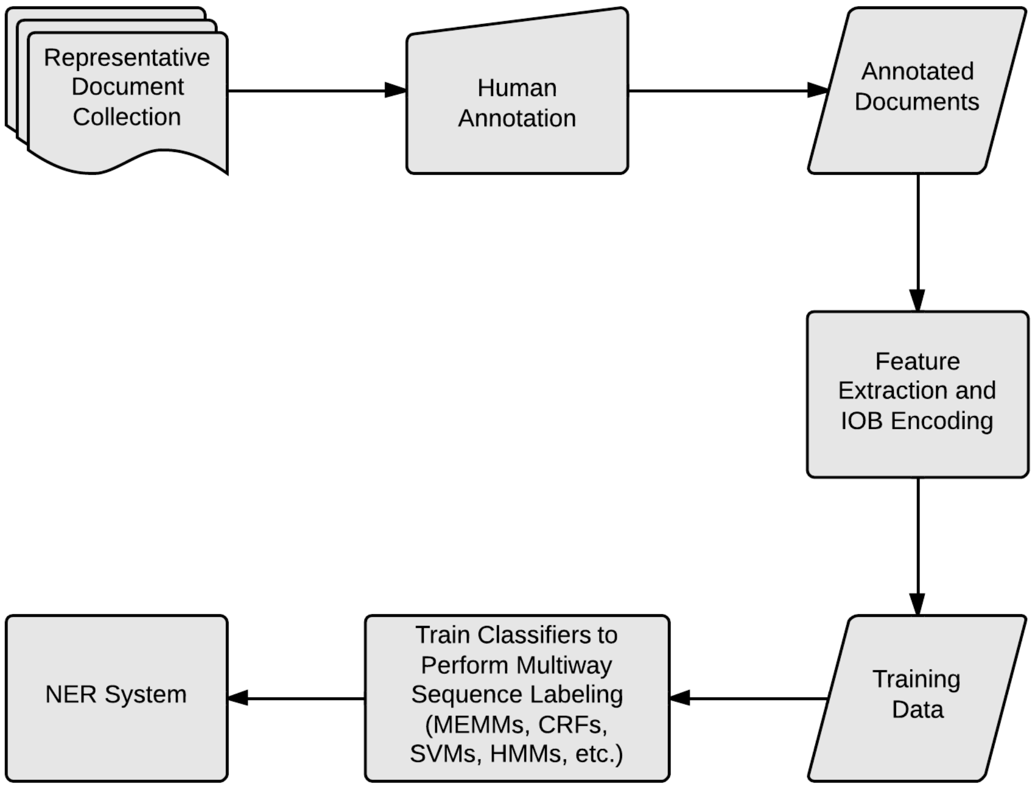

Representative Document Collection
Human Annotation
Annotated Documents
Feature Extraction and IOB Encoding
Training Data
Train Classifiers to Perform Multiway Sequence Labeling (MEMMs, CRFs, SVMs, HMMs, etc.)
NER System

Figure 29: Basic steps in creating a named entity recognition system (Jurafsky und Martin 2009, 733)

Since The Echo Nest provides one via their API, and implementing one is non-trivial, The Echo Nest's ability to extract artist entities from a given text is used so the two possible conditions can be tested for: A single artist or entity name, or a combination of an entity name and a release by that entity.

### Is artist-name, is artist-release

The first condition checks if the "Best guess" text, called "query_string" in the following represents in fact a single artist or band name, and nothing else. This can be the case if the post's "og:image" property linked an artist or band image, a common choice. The class method
"full_artist_name_match?(query_string, echonest_artist_object)" of the EchonestApi class is called from the PostParser. It checks if replacing the part of the "query_string" that matches an artist name returned by The Echo Nest with an empty string does in fact empty the entire "query_string".

```
if ( query_string.downcase.gsub(
echonest_artist_object['name'].downcase, '').length == 0  )
```

The second condition tests if an artist-release combination is present in the "query_string". If the post's "og:image" attribute holds cover artwork of a release, and Google Images identified that release correctly with their "Best guess", then a combination of an artist found by The Echo Nest, and a release by that artist should be present in the "query_string". Discogs.com provides programmatic access to a large database of user generated discography information: a "catalog of more than 3.5 million recordings and 2.5 million artists". (About Discogs 2013) DiscogsAPI is the class wrapping the API access (Discogs API v2.0 Documentation 2013). A DiscogsApi instance is initialized with a Discogs::Wrapper instance to call the API. This class is provided by the gem "discogs-wrapper". (buntine/discogs · GitHub 2013).

The method implementing this test is
"artist_release_combination?(query_string, echonest_artist_object)". It calls upon the instance method "list_titles_by_artist(artist)" where the titles of the main releases by the requested artist are received from Discogs' API. Each of these titles is tested for presence in combination with the artist for those titles in the "query_string" received from Google Images.

```ruby
def artist_release_combination?(query_string, echonest_artist_object)
query_string.downcase!
  list_titles_by_artist(
      echonest_artist_object['name']).each do |title|
    if ( query_string.include?(title[:release_title].downcase) &&
     query_string.include?(echonest_artist_object['name'].downcase) )
      return true
    end
  end
  return false
end
```

Figure 30: DiscogsApi method to validate an artist-release combination

### Create keyword

If any of these validations pass, a Keyword instance for the artist found is created and persisted, associating it with the Post instance it was found in via a KeywordPost join model class method.

### Query music provider (SoundCloud)

At this point, not only one track, but possibly lots of tracks could be created, since provider queries for both tracks by a certain artist, or a release by a certain artist are filtered out. In both cases, it is possible that a provider query will return multiple results. Following the principle of selecting the track closest to the recommendation the author intended, the result that matches the “query_string” best needs to be returned. For the provider SoundCloud, matching every word in the “query_string” with the titles of the resulting SoundCloud tracks and returning the first full match was chosen.

The SoundcloudProvider class method “query(searchTerm)” implements the query for a single track from the streaming music provider SoundCloud.com. A new instance of the SoundCloud class provided by the gem “SoundCloud” (soundcloud | RubyGems.org | your community gem host 2013) is created with a SoundCloud API key, from a previously registered SoundCloud App. On this instance the query is called, specifying the requested types as tracks and sets, the search term, and a filter for only streamable items as parameters. Each result is then looped through to match every word of the original search term; independent of the order they occur in the SoundCloud response object’s title attribute. This again is wrapped in a block to rescue and retry on response errors related to server downtime or network failure.

```
def self.query(searchTerm)
  client = Soundcloud.new(:client_id => SOUNDCLOUD_CLIENT_ID)
  # throws 503 service unavailable, wrap in proc for retry
  retry_count = 0
  api_call = Proc.new do
    #query for playlists and tracks
    playlists = client.get('/playlists',
      :q => searchTerm,
      :filter => 'streamable')
    tracks = client.get('/tracks',
      :q => searchTerm,
      :filter => 'streamable')
    results = [] << playlists << tracks
    # loop through results
    results.each do |result|
      unless (result.empty?)
        #verify and return best result
        result.each do |item|
          (searchTerm.split(' ')).each
          do |search_word|
            unless (
              item.title =~ /#{search_word}/i)
              break
            end
            #found a matching item
            return item
          end
        end
      end
    end
    #didn't find a result
    return false
  end
  # call proc, retry in case of exception
  begin
    api_call.call(searchTerm)
  rescue Soundcloud::ResponseError => e
    retry_count++
    sleep(1)
    retry unless (retry_count > 2)
  end
end
```

Figure 31: SoundcloudProvider's method to request and filter tracks

### Create track

If a SoundCloud track object is returned, the Track class method “create_from_SoundCloud_track” is passed the attribute values necessary to create a new Track record. Here SoundcloudProvider’s method to retrieve an HTML embed code for display is invoked. True is returned to signal the success of the strategy to the PostParser’s caller, false in any other case. The resulting Track object is then associated with the previously created Keyword instance.

```
#stategy - extract coverart from post, reverse google image search
#that, query provider whith google's guess what this image represents.
def create_tracks_for_coverart
  query_string = HtmlParser.new(
    HTTParty.get(@post.url)).query_string_for_coverart_image
  # validate query string to represent either an artist name,
  # artist-release combination. reject if not.
  if (query_string)
    #capitalize so Echonest recognizes artists
    query_string = query_string.split(' ').map(&:capitalize).join(' ')
    EchonestApi.extract_artist_objects_from_string(query_string).
    each do |echonest_artist_object|
      #check if 100% artist name match
      is_artist_name =
       EchonestApi.full_artist_name_match?(
        query_string, echonest_artist_object)
      # check if Artist and release combination
      is_artist_release =
       DiscogsApi.new().
       artist_release_combination?(
        query_string, echonest_artist_object)
        # save found artist keyword, query provider with query string
      if (is_artist_name || is_artist_release)
        # save found artist as keyword
        keyword_post =
         KeywordPost.create_keyword_with_post!(
          echonest_artist_object['name'], @post.id)
        # query provider with validated string
        if (soundcloud_track =
         SoundcloudProvider.query(query_string))
          KeywordTrack.create!(keyword_id: keyword_post.keyword.id,
           track_id: Track.create_from_soundcloud_track(
            soundcloud_track, @post).id )
          return true
        end
      end
    end
    # iterated through all echonest artist objects,
    # didn't create a track, so strategy fail
    return false
  else
    # no query string, so no dice.
    return false
  end
end
```

Figure 32: PostParser method for "Google reverse image search" strategy

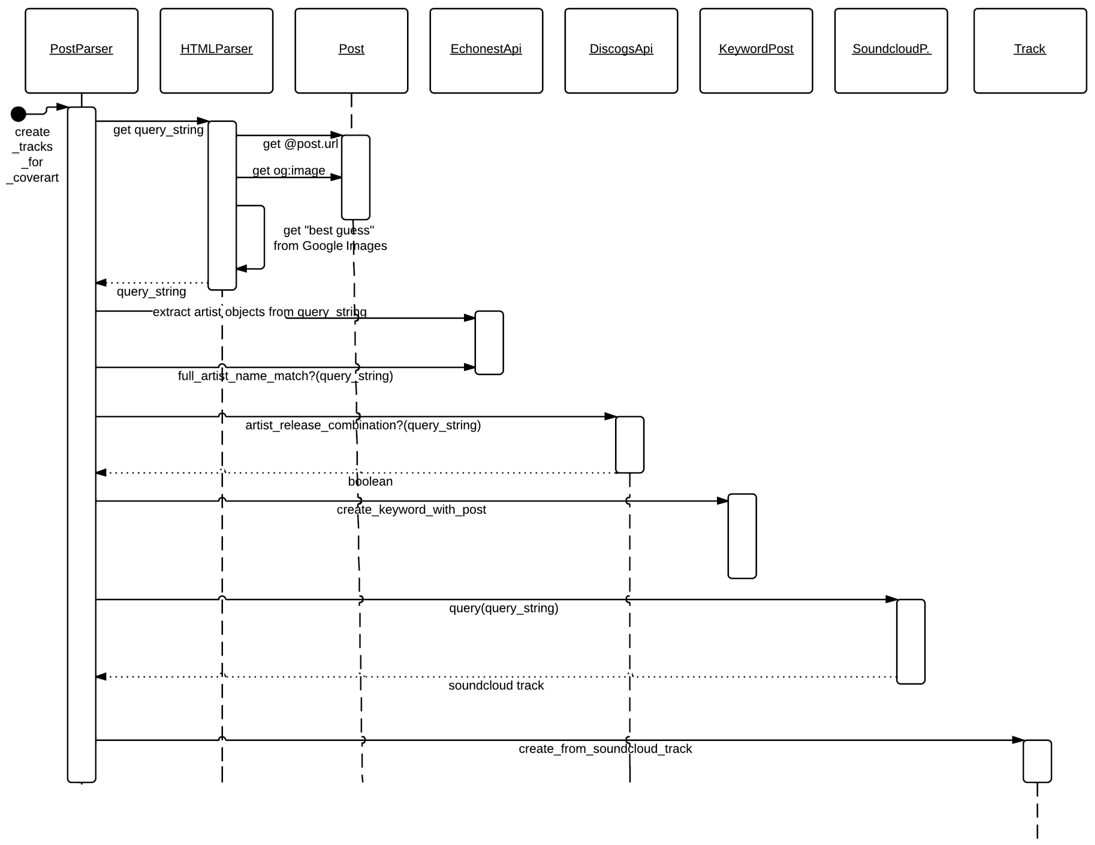

PostParser
HTMLParser
Post
EchonestApi
DiscogsApi
KeywordPost
SoundcloudP.
Track
create _tracks _for _coverart
get query_string
get @post.url
get og:image
get "best guess" from Google Images
query_string
extract artist objects from query_string
full_artist_name_match?(query_string)
artist_release_combination?(query_string)
boolean
create_keyword_with_post
query(query_string)
soundcloud track
create_from_soundcloud_track

Figure 33: UML sequence diagram for Google reverse image search strategy

### 7.3.2.3 Named entity extraction and matching

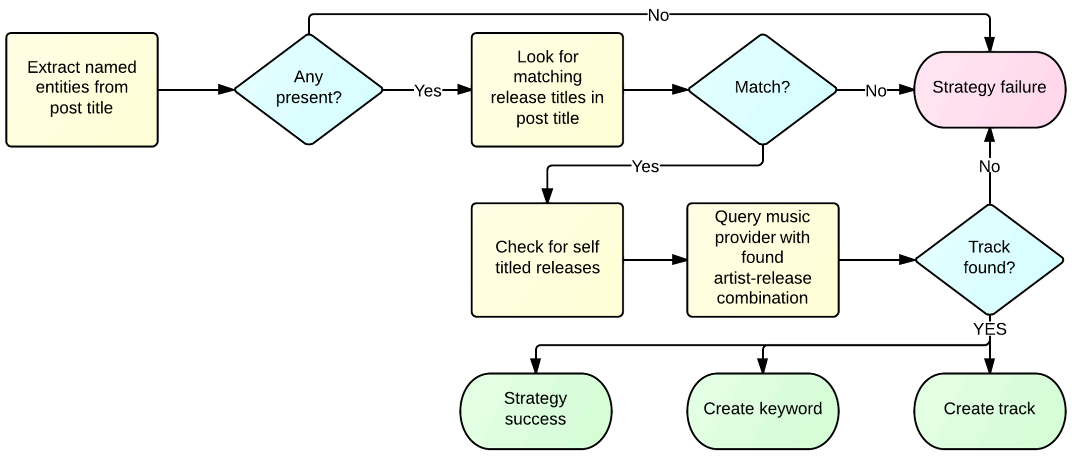

Extract named entities from post title
Any present?
Yes
No
Look for matching release titles in post title
Match?
No
Strategy failure
Yes
Check for self titled releases
Query music provider with found artist-release combination
Track found?
No
YES
Strategy success
Create keyword
Create track

Figure 34: Flow chart for named entity extraction and matching strategy

The third strategy is to extract artist entities from the post's title, and subsequently search for release titles by that entity in the post title. If such a combination is present, the author's recommendation intention is assumed to be that release.

### Extract named entities from post title

To accomplish this goal, the EchonestApi class method "extract_artists_from_titles(post_id)" is called. It works much the same as the method "extract_artist_objects_from_title(post_id)" described before but only returns the artist names found in the post's title as an array. Previous strategy drafts used more than just the name property of The Echo Nest's return, and so might future ones, hence the different signatures.

### Any present?

If the extraction yielded a result, the returned array is not empty and the next process can start. Otherwise, the strategy has failed; without artist entities, a NER system for release titles would be needed, and The Echo Nest does not offer this functionality, nor another API found by the author.

### Look for matching release titles in post title

The Echo Nest's artist extraction service yields many results with most post titles, so the strategy has to determine which ones of those are actually artists semantically mentioned in the post. If artists are validated this way, the release title needed to query a music provider is parsed as a side effect[27].

This is achieved using the DiscogsApi class using the "discogs-api" gem that provides the wrapper class to call API methods on. (buntine/discogs · GitHub 2013) (Discogs API v2.0 Documentation 2013)

With a PostParser helper method, each artist name found by The Echo Nest is validated by requesting a list of main releases from Discogs.com with an API call, using the already described method "list_titles_by_artist(artist_name)", and subsequently checking if any of the returned release titles are included in the post title. If none are present, the strategy has failed.

### Check for self titled releases

If matching release titles are found, the case of self-titled releases needs to be checked. When the matching artist name and release title are identical, that string needs to be present twice in the post title to qualify for a self-titled release. Otherwise, a false positive might occur, validating the artist, and the release on the same substring in the post title.

[27] - killing two birds with one stone.

### Query music provider with found artist-release combination

If this validation passes, a music provider query is constructed with the found artist – title combination, using the previously described “query” class method of Soundcloud-Provider.

Track and Keyword creation are identical to the way the Google reverse image search strategy handles these tasks.

```
def validate_and_create_tracks_semantically
  artist_names = EchonestApi.extract_artists_from_titles(@post.id)
  unless (artist_names.empty?)
    artist_names.each do |artist_name|
      #get a response object from discogs with all main releases for
      #that artist
      d = DiscogsApi.new
      # titles is {:release_id=>"1", :release_title=>"title"}
      titles = d.list_titles_by_artist(artist_name)
      #check that object for post.title release title matches
      titles_found =
      look_for_discogs_artist_titles_in_post_title(titles)
      unless (titles_found.empty?)
        titles_found.each do |title|
          if (artist_name.eql?(title))
            # break if not present in post.title twice
            if ( (@post.title.match(/#{title}/i)).captures.length < 2 )
              next
            end
          end
          #set up keyword for this validated artist
          keyword_post =
           KeywordPost.create_keyword_with_post!(artist_name, @post.id)
          query = artist_name + " " + title
          #found valid artist - release combo, so query provider
          if (track = query_soundcloud_and_create_track(query) )
            KeywordTrack.create!(keyword_id: keyword_post.keyword.id,
              track_id: track.id)
          end
        end
      end
    end
    return false
  else
    return false
  end
end
```

Figure 35: PostParser method for "named entity extraction and matching" strategy

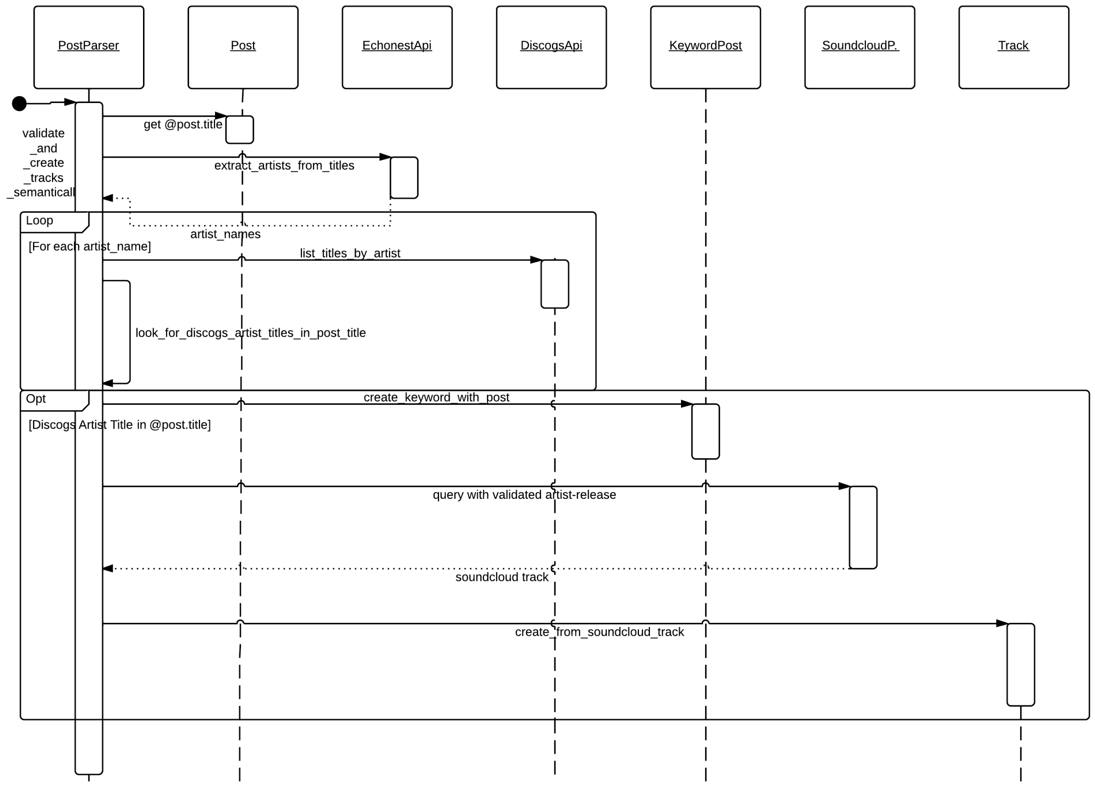

PostParser
Post
EchonestApi
DiscogsApi
KeywordPost
SoundcloudP.
Track
validate_and_create_tracks_semanticall
get @post.title
extract_artists_from_titles
artist_names
Loop
[For each artist_name]
list_titles_by_artist
look_for_discogs_artist_titles_in_post_title
Opt
[Discogs Artist Title in @post.title]
create_keyword_with_post
query with validated artist-release
soundcloud track
create_from_soundcloud_track

Figure 36: UML sequence diagram for "named entity extraction and matching" strategy

### 7.3.3 Order and comparison of post parsing strategies

The return values of the three strategies are chosen so they can be called in this order:

1. HTML parsing for embedded content
2. Google reverse image search
3. Named entity extraction and matching

If a strategy succeeds or fails, it returns true or false making it possible for the caller to finish parsing the post, or move on to the next strategy. The order is chosen to balance their speed, success rate, and accuracy. This is measured through testing as described later.

## 7.4 Playlist display

As shown later, results of post parsing can take longer than a few seconds, so the user interface is presented with a challenge: Once the user submits the desired feeds, immediate playlist display will be expected. The last user story is addressed here:

- I want to view all my music recommendations in one list, no matter which provider they are hosted on so that I can listen to them without switching apps or tabs

### 7.4.1 Multithreading

First, the thread handling the display of the playlist's "show" view that follows the "new" view needs to be relieved from the responsibility to parse the feeds and their posts, so it can serve the next page right away.

Since parsing a post involves multiple calls to webservers and API's, processing multiple posts concurrently could speed it up. All three post-parsing algorithms have a high potential to be processed in parallel since one PostParser instance does not depend on another's data.

To implement this, a library for "background jobs" or asynchronous processing is helpful. The gem 'Sidekiq' (Sidekiq 2013) is chosen, since it uses threads for its workers instead of processes like the gem 'Resque' (resque/resque · GitHub 2013), saving memory. Both are well documented on Ryan Bate's Railscasts (Bates, #271 Resque - RailsCasts 2011), (Bates, #366 Sidekiq - RailsCasts 2012). Sidekiq's retry mechanism is another advantage; if a transient network or API server error is responsible for an uncaught exception, Sidekiq will retry its worker's jobs periodically.

A class that includes Sidekiq's worker module has a perform method which is executed asynchronously. A Redis (Redis 2013) database is needed by Sidekiq internally, and has to be started from the terminal along with the Sidekiq process itself. This is done specifying the maximum number of concurrent threads, and the priority for each queue. The implementation is changed so the playlists controller's "create" method calls FeedWorker's "perform_async" method for each Feed instance. Those in turn call FeedParser instances, who then call the PostWorker's "perform_async" class method for each of their associated Post instances, which call PostParser instance methods. This way, each feed, and each of its posts are parsed concurrently.

```
def parse
  #fetch and parse feed with Feezirra
  feedzirra_feed = Feedzirra::Feed.fetch_and_parse(@feed_url)
  # if this fails feedzirra_feed is the Fixnum error code
  unless feedzirra_feed.is_a?(Fixnum)
    #if !exists, set attributes and save to db
    if @feed.update_from_feed(feedzirra_feed)
      #create posts if !exists, set attributes from RSS feed
      new_feed_entries = Post.update_from_feed(
        feedzirra_feed, @feed.id)
      new_feed_entries.each do |post|
        PostWorker.perform_async(post.id)
      end
    end
  end
end
```

Figure 37: FeedParser method calling PostWorker

```
def perform(post_id)
  # get the embeded content and create tracks
  post_parser = PostParser.new(post_id)
  return if post_parser.extract_tracks_from_embeds
  #else
  return if post_parser.create_tracks_for_coverart
  # else strategy with discogs validated echonest extract artists in
  #post title
  post_parser.validate_and_create_tracks_semantically
end
```

Figure 38: PostWorker method calling PostParser strategies in order

#### 7.4.1.1 PostgrSql instead of Sqlite3

As Sidekiq's workers are accessing the database that Rails is using concurrently, that database engine needs to support concurrent access sufficiently. Rails default Sqlite3 locks the entire table for the time of write access. (File Locking And Concurrency In SQLite Version 3 2013) This is a problem when multiple workers are trying to write to feed, post, track, and keyword tables concurrently. The solution is to use a PostgrSql (PostgreSQL: The world's most advanced open source database 2013) engine instead. It features row locking instead of table locking[28].

#### 7.4.1.2 Fine-tuning the number for concurrent workers for rate-limited API access

Since calls to the Discogs API are effectively rate limited to 60 calls per minute (API / Limit of 1 request per second - Discogs Help Forum 2013), concurrent access needs to

[28] " A row lock is the lowest level of granularity of locking possible in SQL Server. This means one or more specific rows will be locked, and the adjacent rows are still available for locking by concurrent queries.[…] A table lock will lock the complete table." (De Vos 2012)

be limited to this rate. In the development setup, running ten concurrent workers turned out to be a good compromise. Less concurrent workers result in slower processing of posts, more in exceptions thrown by the DiscogsApi instance method accessing the API.

### 7.4.2 Polling

Now the user is presented with a potentially empty playlist view, which only features tracks if a previously parsed feed was selected. New tracks that are being added by parsing the new posts should appear in this playlist. The user would have to reload the page in order to get the new tracks without a mechanism in place that updates the playlist view, which would also interrupt playback. A mechanism for the server to send new tracks to the client as they appear in the database is needed. A web socket (The Web Sockets API 2009) would provide an immediate way to send data to the client. Polling would send the data in intervals. As the use case does not require immediate update, polling is chosen since it saves the system resources to keep a web socket open. A Railscasts on the subject describes the mechanism, and code for this was created analogues to this approach. (Bates, #229 Polling for Changes (revised) - RailsCasts 2013). In regular intervals, an AJAX call to the index action of the TrackController retrieves tracks in the current playlist with a higher "id" value than the ones already featured on the page. Those are added below.

In order to quickly query for Track records that are related to a Playlist, the PlaylistTrack join model needs to be populated with foreign keys upon Track record creation. This is achieved with callbacks within the lifecycle of an entity instance provided by the Rails ORM. As Track objects get created through their association with Post objects, each time a Track object relation is added to a Post object, the :after_add callback is triggered, with that Track object as argument. To implement this callback, the Post object's parent Feed object, and each Playlist object those Feed objects are associated with are iterated, and the argument Track object is associated with the Playlist objects.Should an existing Track be found in a Post, the reverse also needs implementation; As the Post gets associated with the Track, the Track's :after_add callback is triggered, with that Post as argument. Through the Post's parent Feed, its Playlists can again be iterated, and the Track's foreign key added to the PlaylistTrack join tables.

Tracks are displayed for playback on the "show" view simply by rendering their "soundcloud _embed" or "youtube_embed" property values as HTML.

# 8 Results

## 8.1 Test setup

The Program is run with the same feeds for each algorithm. Each post is parsed with the call to the other algorithms commented out, and with the database server and the webserver rebooted. Each post is parsed in its own PostParser concurrently, ten at a time. A log file keeps the time each PostWorker is taking to complete. This determines the speed measure. If a post is parsed to the point where a query string is validated positively this is also logged; A message is written to Rails.logger.debug. That determines the success rate. A ruby script detects the necessary data in the log file and calculates the statistics. Accuracy is determined manually, examining each post title and comparing it to the query result the algorithm validated positively. If the post title and the query result of the algorithm describe the same piece of music, an accurate result is achieved.

## 8.2 Test results

### 8.2.1 Testing each strategy separately

Playlists tested:

a) De:Bug magazine reviews[29], Pitchfork magazine album reviews[30] - feeds with album and single reviews. No embedded content is present on any posts. The og:image meta tag is present on all posts.

b) De:Bug magazine podcast feed[31], Local Suicide blog feed[32] - feeds with mixed music related posts, including embedded content. The og:image meta tag is present on all posts.

Algorithm: HTML parsing for embedded content

Results:

a): 100% fail rate, since no embedded content present on either post summary, body, or the website linked by the posts.

b)

[29] http://de-bug.de/reviews/feed

[30] http://pitchfork.com/rss/reviews/albums/

[31] http://de-bug.de/pod

[32] http://www.localsuicide.com/feed/

- Speed: Average processing time is 0.31 seconds (= 0.01 minutes) per post
- Success Rate: 13 posts of 20 success, success rate is 65.00 %
- Accuracy: 100%

Algorithm: Google reverse image search

Results:

a)

- Speed: average processing time is 7 seconds per post
- Success Rate: 10 of 50 posts success, success rate is 20%
- Accuracy: 8/10 right, 80%

b)

- Speed: average processing time is 10.07 seconds (= 0.17 minutes) per post
- Success Rate: 3 posts of 20 success, success rate is 15 %
- Accuracy: 0/3 right, 0%

Algorithm: Named entity extraction and matching

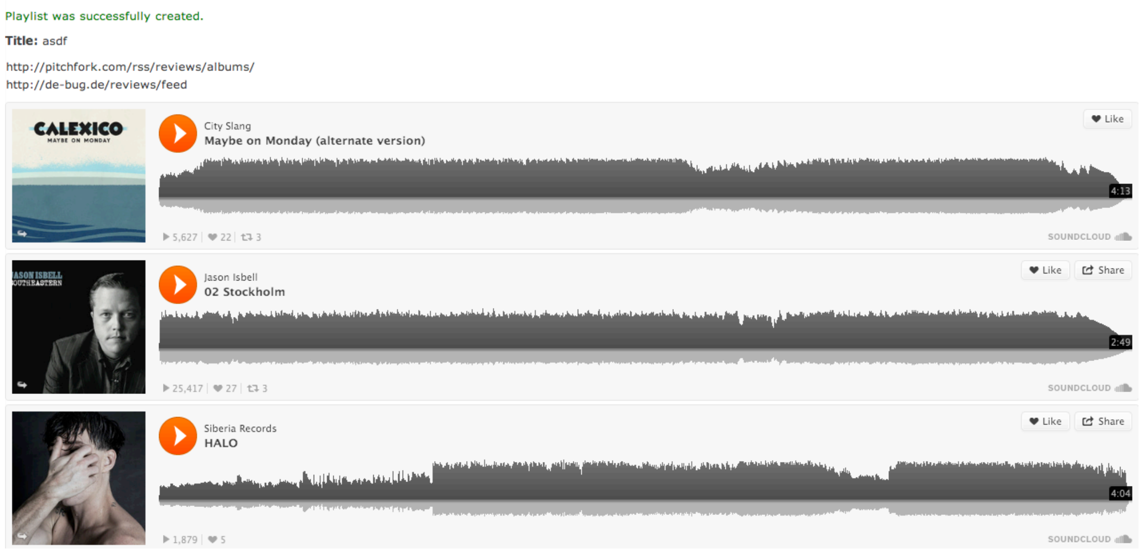
Results:

a)

- Speed: average processing time per post is 36 seconds per post
- Success rate: 24 posts of 50 returned query string for provider, success rate is 48%
- Accuracy: 23/24 right, 96%

b)

- Speed: average processing time is 30.41 seconds (= 0.51 minutes) per post
- Success Rate: 3 posts of 20 success, success rate is 15 %
- Accuracy: 2/3 correct, 66%


Playlist was successfully created.

**Title:** asdf

http://pitchfork.com/rss/reviews/albums/
http://de-bug.de/reviews/feed

Figure 39: Playlist's "show" view with embedded SoundCloud players parsed from RSS feeds from test a)

### 8.2.2 Testing all strategies in order

Playlists tested:

a) Norwegian fish[33], lakes for diving[34] - feeds with content not containing music related posts. The "og:image" tag is present on posts in the Norwegian fish feed.

a) De:Bug magazine reviews[35], Pitchfork magazine album reviews[36] - feeds with album and single reviews. No embedded content is present on any posts. The og:image meta tag is present on all posts.

Algorithms:

1. HTML Parsing For Embedded Content
2. Google Reverse Image Search
3. Named Entity Extraction And Matching

If an algorithm returns true for success, the next post is parsed. If it returns false, the next strategy is called.

Results:

a)

- Speed: average processing time is 11.38 seconds (= 0.19 minutes) per post
- Success Rate: 0 posts of 57 success, success rate is 0 %

[33] http://www.norwegenfisch.de/rss/feed/new

[34] http://www.seen.de/rss/tauchseen.xml

[35] http://de-bug.de/reviews/feed

[36] http://pitchfork.com/rss/reviews/albums/

- Accuracy: 100% - no tracks created

b)

- Speed: average processing time is 39.66 seconds (= 0.66 minutes) per post
- Success Rate: 35 posts of 50 success, success rate is 70 %
- Accuracy: 27/35 correct, 77%

## 8.3 Test weaknesses

The test with all strategies in order was done on a different day than those with each strategy in isolation, so their results are not comparable even if they parse the same feeds. Speed, success rate, and accuracy highly depend on feed content, which is different for every feed, and different among posts in feeds. Total time from submitting the feeds to Sidekiq finishing the posts queue was not logged; In order to really benchmark these algorithms, a representative empirical study with a lot more test data would have to be conducted. However, the results at hand give enough data for interpretation.

## 8.4 Interpretation

Extracting embedded content is relatively fast since it does not require named entity extraction or lengthy loop comparisons. It is also highly accurate, since the exact piece of content found in the post or on the website linked by the post is made into a track. The algorithm is only useful in presence of embedded content.

Constructing a music provider query based on Google's “ Best guess” is moderately successful and accurate, but only in the case of release cover artwork presence. Its validation mechanisms using The Echo Nest and Discogs APIs however are time intensive, and can be improved. Fuzzy string matching of a release title found for an artist in Google's “Best guess” string would improve the success rate, for example a Jaro-Winkler distance algorithm to compute approximate matches. (kiyoka/fuzzy-string-match 2013) Also detecting that Google's guess is a release title by an artist that can be found by The Echo Nest in the post title would improve the success rate and accuracy. The inverse would also; an artist image in a post gets a good guess from Google, The Echo Nest could validate that artist, and a fitting song or release title could be found in the post's title or summary with Discog's data.

Using only the post's title to detect an artist-release combination is the most accurate and successful strategy of the three but also the most time intensive. Its success depends on a matching artist and release title combination being found in the post title string. Since artist and release titles can be as ambiguous as artist: “A” and release title “Why”, this is error prone. Its excessive use of API calls to Discogs slows it down the most, as those take about four seconds (Module: Benchmark (Ruby 2.0) 2013), and are rate limited to 60 calls per minute. (API / Limit of 1 request per second - Discogs Help Forum 2013).

Posts that do not contain critical or editorial review are successfully rejected, and letting all three strategies run in the order described above results in a balancing trade-off between speed, success rate, and accuracy.

# 9 Summary

Music discovery can be evaluated on “scale” and “care” qualities. The main input sources are audio analysis, text analysis, articles of critical or editorial review, and activity data. Image analysis can extend these sources. Text analysis can be applied to sources of critical or editorial review: By parsing articles published by music authorities for embedded content, artist entity extraction with The Echo Nest, and validation with music metadata from Discogs.

In the chapter “Approaching a different solution”, the author agrees with Whitman’s thesis regarding the lack of “scale” of systems employing input data from sources of critical or editorial review but shows a way to improve, extending them with aggregated external sources. It is argued that the remaining limitation in “scale” is a beneficial trade-off to “care”, especially if the goal is to find new music within a niche. It is proposed that desired “scale” and perceived “care” can be controlled by the user to create a flexible music discovery engine.

Technical skills were adopted: fluently using Ruby on Rails’ MVC and ORM patterns, and learning the conventions involved. Ruby’s syntactical and conceptual take on data and control structures was explored as well as constructing and using its regular expressions. Learning how to find the right tools from the rich offerings of open source libraries for Ruby is also a valuable result. The Echo Nest, SoundCloud, Discogs, YouTube and Google Feeds API consumption with JSON and Ruby wrappers, error handling, and web-scraping to make up for a missing Google reverse image search API were learned. Polling for changes, and filling HTML forms with JQuery required learning CoffeeScript and JavaScript syntax. The implementation of the background job pattern with Sidekiq resulted in a deeper understanding of concurrency, and its implications when used with rate-limited APIs and table- or row-locking database engines.

The chapter “Results” shows up to 70% success rate and 77% accuracy level on posts parsed from De:Bug magazine reviews[37], and Pitchfork magazine album reviews[38], making the music discovery experience proposed reality. A user can search for RSS feeds by sources of critical review, select as many as desired, and subsequently listen to music suggested by the chosen sources on one page. Even if those sources did not embedded content for it - and much faster if they did. The automation of music discovery by music authority revives this legacy pattern in a more convenient form with potentially higher “scale”.

[37] http://de-bug.de/reviews/feed

[38] http://pitchfork.com/rss/reviews/albums/

# 10 Conclusion

Essentially the concept and implementation of the prototype has shown that it is possible to extract the subject of articles of music recommendation with text and image analysis in such a way that providers of streaming music can be queried for it successfully. A way to use text and image analysis for music discovery and still give the user control over the sources of recommendation has been implemented.

So how is it possible to extract music recommendation subjects by parsing sources of critical or editorial review in RSS format? This work shows that music recommendation subjects can be obtained from RSS feeds with three different strategies. First, by extracting embedded content from a post, or the website related to a post. Secondly, using Google to analyze a post's related og:image attribute and validating the result with the help of The Echo Nest artist entity recognition, and Discogs for artist-release detection. Thirdly, by applying The Echo Nest artist entity extraction and Discogs validation pattern to a post's title.

How can the results be turned into a playlist that represents the review's subjects? Ultimately, by relating tracks to the posts they are found in. This relates them to a feed, and playlists that feature this feed. Through asynchronous processing of post parsing and polling for new tracks created, the user can view and play results as soon as they are parsed.

And finally, given the user can choose the sources, what is the quality of this music discovery experience? It depends on the feeds chosen as input. If feeds with cover art or artist images as "og:image", artist/ title combinations in the title, or those including embedded content are chosen, this system works best. The higher the variety of music a feed features is, and the more feeds with a high "scale" are chosen, and the more these differ, the higher the "scale" factor for the system. Equally, the closer the feed's author matches the taste of the user, the higher the "care" factor.

# 11 Prospects

Four months time for research, implementation and documentation of results is a tight schedule, and leaves much to be desired.

Providing results with an instantaneous feel should be the goal. The system should also be beneficial to users who are not aware of musical authorities already, and not able to search for feeds that satisfy the tool's content format constraints. For this, feed content needs to get refreshed and parsed asynchronously before a playlist view is loaded. Offering pre-parsed feeds, sorted by genre editorially would get users started quickly. This could also extend the target audience to those who seek to explore a to them previously unknown music genre. Missing feeds could be submitted, ultimately another part of the program could suggest feeds to users based on their activity data, further adding "scale".

Adding more supported types of feeds as input is an important backlog item. Logging in with Facebook, and having the system parse friends, and liked entities' posts could add "scale" to the music discovery experience. Posts hash-tagged #NowPlaying on Twitter are an ideal candidate for parsing, identifying posts with content recommending music to their own discovery service Twitter#Music (Now Playling Twitter Music 2013). Accepting feed sources in JSON format would be necessary[39]. Adding more providers of streaming music would also improve "scale", as the system could return tracks that are not available on SoundCloud. This would also improve the success rate and accuracy of the system.

Currently, productive RSS feed input underlies format constraints, based on assumptions on image content, and the presence of related artist and release information in the title. To remedy this, also the post's summary, body, and linked HTML page should be parsed for artist entities and related songs and releases. Posts with multiple items of recommendation, especially those including top ten listings could result in more successful track return. Further improvement would come from replacing the current exact comparison of releases in the Discogs database with words in the post title. With a fuzzy string matching algorithm, subtle spelling or formatting variations could still be detected.

Adding artist related song detection would also increase the success rate, so would detection of releases by "Various Artists" and artist collaborations not detected by The

[39] Considering that Twitter dropped support for ist RSS feed interface with API v.1.1 (Overview: Version 1.1 of the Twitter API | Twitter Developers 2013), and that Google dropped support for their RSS Reader altogether (Official Blog: A second spring of cleaning 2013), implementing parsing feeds in JSON is indeed a good move to comply with an obvious trend.

Echo Nest. Google's reverse image search often times only returns a release title – those cases could be covered by looking for a matching artist in the post's content. Embedded content has many more sources than SoundCloud and YouTube; adding support for any flash or HTML5 embedded player would increase the success rate further.

An open question is how to properly parse posts that mention an artist but neither a related song nor release. Validation of returns from named entity extraction would need to be done with a probabilistic method, taking the feed's past recommendations, or the users' activity data into account.

A semantic analysis should be the goal; currently only mentions of subjects are detected, independent of a possible contextual evaluation by the author. More research could be done, approaching the problem of post parsing with patterns from the toolkit of natural language processing. Sentiment analysis, word sense disambiguation, and information extraction techniques could help improve the success rate and accuracy. (Jurafsky und Manning 2012).[40]

Another open question is monetization; a fast and accurate iteration of the system might become a commercial API service for anybody in need of turning feeds into playlists, or posts into songs, releases, and artists. Collaborations with stream service playlist aggregators like Minilogs[41] and Whyd[42], and implementing a Spotify app[43] come to mind.

An answer also needs to be found to the question how to limit the amount of songs shown to the user as the amount of feeds subscribed to grows. More feeds equals more "scale", so an activity data-based filter mechanism could help preserve the "care" factor. Users could follow /unfollow detected entities in posts, such as artists, related genres, labels, or the post's author. The keywords necessary for the implementation are already saved and related to posts and tracks by the current system. A thumb up/down voting system could refine and prioritize the remaining results further. Skipped songs, or songs not played to completion would further indicate a priority, or filter decision by the user. This could also be reflected in the track position within a playlist, a currently unused property.

[40] Given that computer systems are apparently able to read Associated Press' hacked Twitter feed with a spoofed message about explosions hitting the White House, and cause a considerable panic on Wall Street allow hope for sematic text analysis technology having advanced enough to solve the problem. Unless they were just doing named entity extraction for „White House" and matched it with a possibly related „explosion" event from a database, in the feed „Associated Press Twitter", as part of the Playlist „potential terrorist attacks", returning either „sell" or „buy" as the song of choice. (AP Twitter hack causes panic on Wall Street and sends Dow plunging | Business | guardian.co.uk 2013)

[41] http://minilogs.com/

[42] http://whyd.com/

[43] https://developer.spotify.com/technologies/apps/

Strategies need to be developed to cover cases where The Echo Nest does not recognize the artist entity subject, or where Discogs has multiple ambiguous records for a single artist name.

Reducing the amount of API calls by sharing data between post parsing strategies, and caching API call responses would speed up the execution time. Some calls could be replaced – Discogs offers a full monthly dump of their database that could be queried locally before resorting to an API call. This would also make it possible to run more concurrent workers.

Faster execution could also be achieved by using a different Ruby implementation than MRI. True parallel execution depends on the Ruby implementation deployed: Ruby's C-based MRI implementation (Download Ruby 2013) features a Global Interpreter Lock (GIL) which makes sure that its C implementation code can only be accessed by one thread at a time. (Storimer 2013) The Java-based JRuby (Home - JRuby.org 2013), or C++ & Ruby based Rubinius (Rubinius : Use Ruby TM 2013) implementations do not have the GIL. According to Rendek, the difference in execution time for the proof of concept would not have been big enough to warrant the possible compatibility issues. (Rendek 2012) Gems used in the project might not be compatible with a different Ruby implementation used. Still, in a production environment switching the Ruby implementation could have optimization potential.

As Whitman points out, an audio analysis system could improve the "care" factor of the discovery experience by giving playlists an order pleasant to the human ear (B. Whitman 2013); A radio-like experience could be realized – in fact a recent Nielsen study suggests that music discovery in form of a radio stream is the preferred pattern. (Music Discovery Still Dominated by Radio, Says Nielsen Music 360 Report | Nielsen 2012)

It will be exciting to see how computers can be programmed to perfectly parse all the sources of recommendation desired and available to translate them into the music that their users then love - next.

# Appendix A: application source code on CD-ROM

# List of abbreviations

| | |
|---|---|
| API | Application programming interface |
| AJAX | Asynchronous JavaScript and XML |
| CD | Compact disc |
| CRF | Conditional random field |
| CSS | Cascading style sheets |
| DRY | Don't repeat yourself |
| GEMA | Gesellschaft für musikalische Aufführungs- und mechanische Vervielfältigungsrechte - german collection society for music related intellectual property royalties |
| GIL | Global interpreter lock |
| HTML | Hypertext markup language |
| HTTP | Hypertext transfer protocol |
| IDE | Integrated development environment |
| JSON | JavaScript object notation |
| LP | Long play record |
| MEMM | Maximum entropy Markov model |
| MIT | Massachusetts institute of technology |
| MP3 | MPEG-2 Audio Layer III |
| MVC | Model view controller |
| NER | Named entity recognition |
| NLP | Natural language processing |
| ORM | Object-relational mapping |
| RDF | Resource description framework |
| REST | Representational state transfer |
| RSS | Rich site summary or really simple syndication |
| SAX | Simple API for XML |
| SVM | Support vector machine |

| | |
|---|---|
| UML | Unified modeling language |
| URI | Uniform resource identifier |
| URL | Uniform resource locator |
| XML | Extensible markup language |
| XPath | XML path language |

# Bibliography

*About Discogs.* 2013. http://www.discogs.com/about (accessed 06 28, 2013).

*About Last.fm – Last.fm.* 2013. http://www.last.fm/about (accessed 07 20, 2013).

*AP Twitter hack causes panic on Wall Street and sends Dow plunging | Business | guardian.co.uk* . 04 23, 2013. http://www.guardian.co.uk/business/2013/apr/23/ap-tweet-hack-wall-street-freefall (accessed 07 18, 2013).

*API / Limit of 1 request per second - Discogs Help Forum.* 02 2013. http://www.discogs.com/help/forums/topic/361485 (accessed 07 20, 2013).

*Artist API Methods - The Echo Nest 4.2 documentation.* 2013. http://developer.echonest.com/docs/v4/artist.html#extract-beta (accessed 06 26, 2013).

Bates, Ryan. *#229 Polling for Changes (revised) - RailsCasts.* 1 14, 2013. http://railscasts.com/episodes/229-polling-for-changes-revised (accessed 7 10, 2013).

—. *#271 Resque - RailsCasts.* 6 20, 2011. http://railscasts.com/episodes/271-resque (accessed 07 10, 2013).

—. *#366 Sidekiq - RailsCasts.* 7 18, 2012. http://railscasts.com/episodes/366-sidekiq (accessed 7 10, 2013).

*Billboard.com.* 02 20, 2013. http://www.billboard.com/articles/news/1549399/hot-100-news-billboard-and-nielsen-add-youtube-video-streaming-to-platforms (accessed 07 31, 2013).

*Building Feedly.* 06 19, 2013. http://blog.feedly.com/2013/06/19/feedly-cloud/ (accessed 07 02, 2013).

*buntine/discogs · GitHub.* 2013. https://github.com/buntine/discogs (accessed 06 26, 2013).

*CoffeeScript.* 06 02, 2013. http://coffeescript.org/ (accessed 07 16, 2013).

Dahlen, Chris. *Articles: Better Than We Know Ourselves | Features | Pitchfork.* 5 22, 2006. http://pitchfork.com/features/articles/6340-better-than-we-know-ourselves/ (accessed 07 26, 2013).

De Vos, Filip. *sql - What are row, page and table locks? And when they are acquired? - Stack Overflow.* 3 20, 2012. http://stackoverflow.com/questions/9784172/what-are-row-page-and-table-locks-and-when-they-are-acquired (accessed 07 29, 2013).

*Discogs API v2.0 Documentation* . 4 8, 2013. http://www.discogs.com/developers/ (accessed 06 26, 2013).

*Download Ruby.* 2013. http://www.ruby-lang.org/en/downloads/ (accessed 07 12, 2013).

*Explore Videos on Discogs.* 06 15, 2013. http://www.discogs.com/explore/videos (accessed 06 15, 2013).

*Feed of questions tagged ruby-on-rails.* 2013. http://stackoverflow.com/questions/tagged/ruby-on-rails (accessed 06 17, 2013).

*feedly: your news. delivered.* 07 16, 2013. cloud.feedly.com/#my (accessed 07 16, 2013).

*File Locking And Concurrency In SQLite Version 3.* 07 23, 2013. http://www.sqlite.org/lockingv3.html (accessed 07 31, 2013).

*Framework technologies Web Usage Statistics.* 06 17, 2013. http://trends.builtwith.com/framework (accessed 06 17, 2013).

*Getting Started with Rails — Ruby on Rails Guides.* 2013. http://guides.rubyonrails.org/getting_started.html (accessed 06 17, 2013).

*Git.* 2013. http://git-scm.com/ (accessed 07 26, 2013).

*Google Feed API - Google Developers.* 2013. https://developers.google.com/feed/ (accessed 07 12, 2013).

*Google Images.* 2013. http://images.google.com/ (accessed 06 24, 2013).

Hazard Owen, Laura. *Feedly hits 12 million users, launches web version and quits relying on Google's backend.* 06 19, 2013. http://paidcontent.org/2013/06/19/feedly-hits-12-million-users-launches-web-version-and-quits-relying-on-googles-backend/ (accessed 07 02, 2013).

Herrada, O`scar Celma. *MUSIC RECOMMENDATION AND DISCOVERY IN THE LONG TAIL* . PhD Thesis, Department of Information and Communication Technologies, Universitat Pompeu Fabra, Barcelona: Universitat Pompeu Fabra, 2008.

*Home - JRuby.org.* 2013. http://jruby.org/ (accessed 07 12, 2013).

*HTML iframe tag.* http://www.w3schools.com/tags/tag_iframe.asp (accessed 07 26, 2013).

*Hype Machine.* 07 31, 2013. http://hypem.com/ (accessed 07 31, 2013).

IFPI. *Digital music report 2013 (pdf).* 02 27, 2013. http://www.ifpi.org/content/library/dmr2013.pdf (accessed 07 31, 2013).

*Information | Spotify Press.* 2013. http://press.spotify.com/us/information/ (accessed 07 04, 2013).

*Issues · rails/rails.* 2013. https://github.com/rails/rails/issues (accessed 06 17, 2013).

*JavaScript - Wikipedia, the free encyclopedia.* 07 25, 2013. https://en.wikipedia.org/wiki/JavaScript (accessed 07 26, 2013).

*jnunemaker/httparty - GigHub.* 2011. https://github.com/jnunemaker/httparty (accessed 06 25, 2013).

*jQuery.* 2013. http://jquery.com/ (accessed 07 30, 2013).

Jurafsky, Dan, and Chris Manning. *1 - 1 - Course Introduction - Stanford NLP - Professor Dan Jurafsky & Chris Manning.* 2012. http://www.youtube.com/watch?v=nfoudtpBV68 (accessed 07 20, 2013).

Jurafsky, Daniel, and James H. Martin. *Speech And Language Processing - An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* 2nd ed. Upper Saddle River, NJ: Pearson Education, 2009.

*kiyoka/fuzzy-string-match.* 03 26, 2013. https://github.com/kiyoka/fuzzy-string-match (accessed 07 30, 2013).

Magno, Terence, and Carl Sable. "A COMPARISON OF SIGNAL-BASED MUSIC RECOMMENDATION TO GENRE LABELS, COLLABORATIVE FILTERING, MUSICOLOGICAL ANALYSIS, HUMAN RECOMMENDATION, AND RANDOM BASELINE." *Proceedings of the 9th International Conference on Music Information Retrieval* . Ismir: International Conference on Music Information Retrieval, 2008. 161ff.

*Minilogs.* 07 31, 2013. http://minilogs.com/ (accessed 07 31, 2013).

*Module: Benchmark (Ruby 2.0)* . 07 20, 2013. http://ruby-doc.org/stdlib-2.0/libdoc/benchmark/rdoc/Benchmark.html (accessed 07 20, 2013).

*Music Discovery Still Dominated by Radio, Says Nielsen Music 360 Report | Nielsen.* 08 14, 2012. http://www.nielsen.com/us/en/press-room/2012/music-discovery-still-dominated-by-radio--says-nielsen-music-360.html (accessed 06 15, 2013).

*Musicians' Union demands new pay deal from Spotify | Technology | The Observer.* 07 20, 2013. http://www.guardian.co.uk/technology/2013/jul/20/spotify-radiohead-musicians-union-rights (accessed 07 20, 2013).

*Nokogiri.* 2013. http://nokogiri.org/ (accessed 06 24, 2013).

*Now Playling Twitter Music.* 04 18, 2013. https://blog.twitter.com/2013/now-playing-twitter-music (accessed 07 18, 2013).

*Official Blog: A second spring of cleaning.* 03 13, 2013. http://googleblog.blogspot.com.au/2013/03/a-second-spring-of-cleaning.html (accessed 07 20, 2013).

*Overview: Version 1.1 of the Twitter API | Twitter Developers.* 05 07, 2013. https://dev.twitter.com/docs/api/1.1/overview (accessed 07 20, 2013).

*pauldix/feedzirra · GitHub.* 2 2013. https://github.com/pauldix/feedzirra (accessed 07 12, 2013).

*Playground.fm | VentureBeat.* 10 11, 2012. http://venturebeat.com/2012/10/11/playground-fm-the-social-pandora-launches-its-free-music-discovery-app/ (accessed 07 02, 2013).

*PostgreSQL: The world's most advanced open source database.* 2013. http://www.postgresql.org/ (accessed 07 10, 2013).

*Rails ERD – Entity-Relationship Diagrams for Rails.* 2012. http://rails-erd.rubyforge.org/ (accessed 07 2, 2013).

*RDF Semantics.* 2013. http://www.w3.org/TR/2004/REC-rdf-mt-20040210/ (accessed 06 28, 2013).

*Redis.* 2013. http://redis.io/ (accessed 07 10, 2013).

Rendek, Josh. *Sidekiq vs Resque, with MRI and JRuby - Josh Rendek.* 11 3, 2012. http://joshrendek.com/2012/11/sidekiq-vs-resque/ (accessed 07 12, 2013).

*resque/resque · GitHub.* 2013. https://github.com/resque/resque (accessed 07 10, 2013).

*RSS - Wikipedia, the free encyclopedia.* 07 26, 2013. http://en.wikipedia.org/wiki/RSS (accessed 07 26, 2013).

*Rubinius : Use Ruby TM.* 2013. http://rubini.us/ (accessed 07 12, 2013).

*Ruby on Rails: Ecosystem.* 2013. http://rubyonrails.org/ecosystem (accessed 06 17, 2013).

*rubygems.org.* 2013. http://rubygems.org/ (accessed 06 17, 2013).

*ruby-lang.* 2013. http://www.ruby-lang.org/en/about/ (accessed 06 17, 2013).

*SAX.* 04 27, 2004. http://www.saxproject.org/ (accessed 07 26, 2013).

Schaback, Artur. *Blog about programming tips, internet marketing, SEO and anything related to internet business: Google Reverse Image Search scraping without API in PHP.* 01 17, 2013. http://skyzerblogger.blogspot.de/2013/01/google-reverse-image-search-scraping.html (accessed 06 26, 2013).

*Scrum.org | The home of Scrum > Home .* 2013. http://www.scrum.org/ (accessed 07 20, 2013).

*Sidekiq.* 2013. http://sidekiq.org/ (accessed 07 10, 2013).

*Simply Measured.* 2013. http://simplymeasured.com/.

*SoundCloud - Hear the world's sounds.* 3 12, 2013. http://soundcloud.com/terms-of-use#license (accessed 06 15, 2013).

*soundcloud | RubyGems.org | your community gem host.* 2013. http://rubygems.org/gems/soundcloud (accessed 06 27, 2013).

Soundcloud. *Docs - API - Using the API.* May 25, 2012. http://developers.soundcloud.com/docs/api/guide#playing (accessed 04 16, 2013).

*Soundcloud.com.* 07 31, 2013. https://soundcloud.com/ (accessed 07 31, 2013).

*Soundcloud.com Site Info.* 2013. http://www.alexa.com/siteinfo/soundcloud.com (accessed 06 17, 2013).

*Spotify.* 2013. https://www.spotify.com/ (accessed 06 15, 2013).

*Spotify Apps API - Spotify Developer.* 2013. https://developer.spotify.com/technologies/apps/ (accessed 07 23, 2013).

*Spotify Launches "Discover" Feature For Its Web App | TechCrunch.* 05 29, 2013. http://techcrunch.com/2013/05/29/spotify-launches-discover-feature-for-its-web-app/ (accessed 06 15, 2013).

*Spotify.com Site Info.* 2013. http://www.alexa.com/siteinfo/spotify.com (accessed 06 15, 2013).

Storimer, Jesse. *Nobody understands the GIL - Part 2: Implementation.* 06 14, 2013. http://www.jstorimer.com/blogs/workingwithcode/8100871-nobody-understands-the-gil-part-2-implementation (accessed 07 12, 2013).

*Techcrunch.com.* 12 04, 2012. http://techcrunch.com/2012/12/04/soundcloud-revamps-site-announces-new-numbers-180m-users-and-counting/ (accessed 07 31, 2013).

*The Asset Pipeline — Ruby on Rails Guides.* http://guides.rubyonrails.org/asset_pipeline.html (accessed 07 23, 2013).

*The Echo Nest API 4.2 documentation.* 2013. http://developer.echonest.com/docs/v4 (accessed 06 25, 2013).

*The Echo Nest Powers Spotify Radio.* 12 16, 2011. http://blog.echonest.com/post/14311681173/spotify-radio-the-echo-nest (accessed 06 15, 2013).

*The Echonest Blog.* 06 15, 2011. http://blog.echonest.com/post/6566331252/artist-entity-extraction-for-all-opening-our-puddle (accessed 06 26, 2013).

*The Open Graph Protocol.* 2013. http://ogp.me/ (accessed 06 24, 2013).

*The Ruby Toolbox - All Categories by name.* 2013. https://www.ruby-toolbox.com/categories/by_name (accessed 06 17, 2013).

*The Ruby Toolbox - Ruby on Rails.* 2013. https://www.ruby-toolbox.com/projects/rails (accessed 06 17, 2013).

*The Web Sockets API.* 12 17, 2009. http://www.w3.org/TR/2009/WD-websockets-20091222/ (accessed 07 31, 2013).

Thomas, Dave, and Andrew Hunt. *The Pragmatic Programmer.* 1st edition. Boston: Addison-Wesley Professional, 1999.

*User Story.* 07 18, 2010. http://c2.com/cgi/wiki?UserStory (accessed 07 20, 2013).

*Websites using Ruby on Rails.* 06 17, 2013. http://trends.builtwith.com/websitelist/Ruby-on-Rails (accessed 06 17, 2013).

*What Does an Indie Get Paid? #1:&nbsp;iTunes - MTT - Music Think Tank.* 02 15, 2013. http://www.musicthinktank.com/blog/what-does-an-indie-get-paid-1-itunes.html (accessed 07 20, 2013).

*What is a gem? Rubygems.org.* 2013. http://guides.rubygems.org/what-is-a-gem/ (accessed 06 24, 2013).

Whitman, Brian. *How music recommendation works -- and doesn't work | Brian Whitman @ variogr.am.* 1 1, 2013. http://notes.variogr.am/post/37675885491/how-music-recommendation-works-and-doesnt-work (accessed 7 1, 2013).

*Whyd.* 07 16, 2013. http://whyd.com/ (accessed 07 16, 2013).

*YouTube.* 2013. http://www.youtube.com/ (accessed 07 30, 2013).

*Youtube.com Site Info.* 06 15, 2013. http://www.alexa.com/siteinfo/youtube.com (accessed 06 15, 2013).

## Eidesstattliche Versicherung

| | | | |
|---|---|---|---|
| Name: | Hofmann | Vorname | Philipp |
| Immatrikulations-nummber | S0522482 | Studiengang: | BA Internationale Medieninformatik |

Ich versichere hiermit, dass ich die vorliegende Bachelorthesis selbstständig und ohne fremde Hilfe angefertigt und keine andere als die angegebene Literatur benutzt habe. Alle von anderen Autoren wörtlich übernommenen Stellen wie auch die sich an die Gedankengänge anderer Autoren eng anlehnenden Ausführungen meiner Arbeit sind besonders gekennzeichnet. Diese Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

| | |
|---|---|
| Ort, Datum | Unterschrift |