Faculdade de Engenharia da Universidade do Porto

Spotify-ed: Music Recommendation and Discovery in Spotify

José Lage Bateira

Working version

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Fabien Gouyon

Co-Supervisor: Matthew Davies

June 23, 2014

Spotify-ed: Music Recommendation and Discovery in Spotify

José Lage Bateira

Mestrado Integrado em Engenharia Informática e Computação

June 23, 2014

Abstract

Not so long ago, before the Internet boom, listening or discovering new music was a challenge on its own. Now, with a few clicks one can have on their hands such a vast music catalogue that a human mind cannot compute it.

There are dozens of online services that offer exactly that. Some focus on creation/-generation of playlists, others try to expand their music catalogue even further, but others focus more on personalized music recommendation. And these ones present their results to the user with a list or a grid of music artists, for example.

However, lists or grids do not give the user enough information about the relation between the results. One could even say that they are not related to each other, which is not true.

The relations exist and can be represented as a network of interconnected artists in a graph, where a node is a music artist, and each edge between them represents a strong connection. This is the concept that RAMA (Relational Artist MAps), a project developed at INESC Porto, uses.

From a single search, RAMA is able to draw a graph that helps the user to explore new music that might caught his/her interest in a much more natural way.

Nonetheless, when a user wants to listen to an artist's music, Youtube's stream is used. Although one can find a large catalogue of music in Youtube, this service is not Music Oriented and the sound quality is not adequate for a music streaming service.

Youtube's stream needs to be replaced, and Spotify can provide a quality stream and an accurate music catalogue.

But how can RAMA and Spotify be integrated?

This thesis proposes a Spotify App. Will a Spotify user experience a more pleasant and natural way of music discovery from this graphical representation of artist relations within Spotify, than its standard discovery more (with grids)?

That is the main question that this dissertation urges to answer.

Resumo

Bem longe vão os tempos, antes da Internet, em que ouvir e descobrir música nova era um desafio por si só. Agora, com alguns cliques, temos acesso a um catálogo de música tão grande, que o nosso cérebro não consegue processar.

Existem dezenas de serviços online que oferecem isso mesmo. Alguns especializam-se na criação/geração de playlists (que funcionam como rádios), outros em expandir o catálogo de música e outros focam-se mais na sugestão e recomendação de artistas/álbuns/músicas personalizada para os utilizadores. Estes últimos, apresentam as sugestões de conteúdo ao utilizador de uma forma rudimentar como listas ou em grelha.

No entanto, listas ou grelhas não fornecem ao utilizador qualquer tipo de informação adicional sobre a relação entre os artistas nem justificam a sua semelhança. Até fazem parecer que não existe nenhuma relação/ligação entre os artistas recomendados, o que não é verdade.

Essas relações existem e podem ser representadas como uma rede de artistas interligados num grafo, onde cada nó é um artista de música, e cada ligação entre nós representa uma ligação forte de parecença entre os artistas. Este é o conceito que o RAMA (Relational Artist MAps), projeto desenvolvido no INESC Porto, usa.

A partir de uma pesquisa de um artista de música, o RAMA cria e desenha um grafo que ajuda o utilizador a explorar música que lhe possa interessar de uma forma muito mais natural e informativa.

No entanto, quando um utilizador pretende ouvir uma música de um artista, é usado *stream* do Youtube. Apesar de este oferecer um catálogo alargado de música, o mesmo não é indicado para esta funcionalidade pois não fornece uma API nativamente orientada a música, nem a qualidade de som do *stream* é adequada.

A experiência musical do utilizador do RAMA poderá melhorar consideravelmente ao colmatar esta falha. Existe por isso uma necessidade de substituir o Youtube por outro serviço mais orientado a *streaming* de música de qualidade. O Spotify é um deles. Fornece API orientada a música, e o *streaming* é de qualidade adequada para este tipo de funcionalidade.

De que formas é que se pode integrar o RAMA e o Spotify?

A escolha final foi desenvolver uma aplicação (como *plugin*) para o Spotify. Será que um utilizador Spotify ao descobrir música nova de uma forma mais gráfica terá uma experiência de utilizador mais rica e natural do que o modo de descoberta *standard* do Spotify (em grelha)?

Esse é o objetivo primordial desta dissertação: Tentar descobrir se utilizadores Spotify terão uma experiência melhorada ao usar a Aplicação Spotify proposta.

Acknowledgements

I would like to thank my beloved dog. He showed compassion, understanding and above all, unconditional love towards a ranty and tired friend.

José Lage Bateira

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

Not so long ago, before the Internet boom, listening or discovering new music was a challenge on its own. Now, with a few clicks one can have on their hands such a vast music catalogue that a human mind is not able to compute.

There is an uncountable number of music streaming services that offer exactly that[1]. These services are, mostly, web based, although some offer desktop applications. They allow the users to play music, save their collection, create playlists and much more. Most of these services also have social components that allow the users to share what they're listening to with their friends, as well as playlists and much more.

There is always something that makes a music streaming service different from the others. Some services focus on creation/generation of playlists (8tracks[2]), others try to expand their music catalogue even further (Spotify[3], Rdio[4]), while others focus more on personalized music recommendations (Pandora[5]). The latter ones, present their music recommendations to the user with a list or a grid of music artists, for example. However, lists do not provide the user enough information about the relation between the results [1]. One could even say that they are not related to each other, which is not true.

The relations exist and can be represented as a network of interconnected artists in a graph, where a node is a music artist, and each edge between them represents a connection. This is the concept that RAMA (Relational Artist MAps), a project developed at INESC Porto, uses [2] [3] [4] [5].

The concept is not new, and in Chapter 2 some services that use visual tools will be analysed in detail.

[1] Although some of them require the users to subscribe to a monthly fee, for example, in order to fully use the service, or remove the advertisements.

[2] http://8tracks.com

[3] http://spotify.com

[4] rdio.com

[5] pandora.com

1.2 Motivation and Goals

From a single search, RAMA draws a graph that helps the user to explore new music that might caught his/her interest in a much more natural way.

Nonetheless, when a user wants to listen to an artist's music, Youtube's stream is used. Although one can find a large catalogue of music in Youtube, this service is not music oriented and the sound quality is not adequate for a music streaming service.

Youtube's stream needs to be replaced. From the available services that provide a vast music catalogue, Spotify[6] provides a good quality stream and a good developer support for creating Spotify powered Applications.

But how can RAMA and Spotify be integrated?

There are several possibilities that Spotify has made available for developers[7] that can help to improve RAMA's concept. From websites, mobile applications, native applications and even plugins for the Spotify Desktop Client, Spotify's API is very complete.

Given some limitations when using some APIs[8], there are several aspects to take into account when choosing which API to use.

In the end, this dissertation proposes a Spotify Application[9] that works like a plugin to the Spotify's Desktop Client, i.e., it should add something to Spotify's Application. This is a very appealing solution: Spotify Users will have the chance to continue using Spotify as they normally would, but with an extra help to discover new music by using RAMA's application *inside* Spotify. This method works on the assumption that Spotify's music discovery mode can be improved using a visual tool. The reasons for the final decision will be explained with further detail in Chapter 3.

This approach urges to answer the following question: Will a Spotify user experience a more pleasant and natural way of music discovery from this graphical representation of artist relations within Spotify, than its standard discovery mode (with lists)?

To answer that question and to evaluate and validate the final prototype, end-user testing will be done to compare Spotify's user experience with the developed application.

1.3 Methodologies

The proposed work phases contemplate:

State of the Art (Chapter 2)
Initial research on the current state of the art.

Contextualization (Chapter 3)
Detailed analysis of the Spotify environment from the user perspective (applications available, e.g.) and the developer perspective (available APIs, e.g.) in order to determine the feasibility prototype's requirements.

[6] http://spotify.com
[7] https://developer.spotify.com
[8] for example, LibspotifySDK (https://developer.spotify.com/technologies/libspotify/) requires the developer and the user of the application to have a premium account.
[9] https://developer.spotify.com/technologies/apps

Implementation and Validation (Chapter 4)
Definition and implementation of: the prototype's main features/requirements; the development processes and the user validation processes.

Future Work (Chapter 5)
Definition of future work to be done in the prototype (improvements, features, bugs, etc).

1.4 Project Requirements

This application is meant to be an extra mode for discovering new music in Spotify.

This way, a visual representation of an artist network with a graph, similar to RAMA, is proposed.

The application runs inside the Spotify environment (Spotify's Desktop Client) were its main features are: visualization of relations between artists; edition of the visualization parameters (branching and depth); edition of the graph by expanding and deleting the nodes; visualize tags/genres (that describe an artist) in the graph representation.

The main tools used in the development of the application were:

Spotify Desktop Client
The developed application is integrated in the Spotify's desktop client.

Webkit Development Tools - webkit.org
This is the engine used to run Spotify Applications.

Npmjs - npmjs.org
Package manager for development dependencies.

Bower - bower.io
Package manager for runtime dependencies.

Gruntjs - gruntjs.com
Manager for automating tasks. Very usefull for tests, code optimization and other repetitive tasks.

Vis.js - visjs.org
Visualization framework.

1.5 Dissertation Structure

This dissertation contains four additional chapters.

In chapter 2, related works will be presented to evaluate the current state of the art.

In chapter 3, the project's details will be explained, starting with an introduction to the Spotify Apps' development environment and the role of the technologies used during the development of the prototype.

In chapter 4, a more detailed explanation about the developed prototype will be given, as well as some challenges and problems encountered during the development process.

Chapter 5 concludes this report.

Chapter 2

State of The Art

2.1 Introduction

In this chapter, the most relevant web services for this thesis will be analysed.

The proposed methodology focus on how the content is presented and less on what the content is (without discarding its importance). Even so, some projects that focus on the content will be analysed.

The presented projects often use external data bases (like last.fm) to fetch metadata from. This is the preferred way, since those are the most complete sets of information.

2.2 Related and Similar Services

2.2.1 Liveplasma - liveplasma.com

liveplasma.com is a *flash*[1] application that not only it allows to see a graph of music artists, but also of books and movies.

The interaction with the graph is very faulted: no changes to the graph are allowed, and the user can easily make a mistake and perform unwanted actions like redrawing the graph with another artist as the root node.

In 2.1 one can see the search result for "Amália Rodrigues".

On the left side of the application there are some interesting elements: a grid of the artist's albums, a mini-player (stream from Youtube).

In 2.2 the user can have the choice to play tracks *only* from that artist, or play *similar* artists.

2.2.1.1 Pros

This tools, has two interesting aspects to it:

- Links to buy albums of the artist
- Play tracks from similar artists to the search artist.

[1] http://get.adobe.com/flashplayer

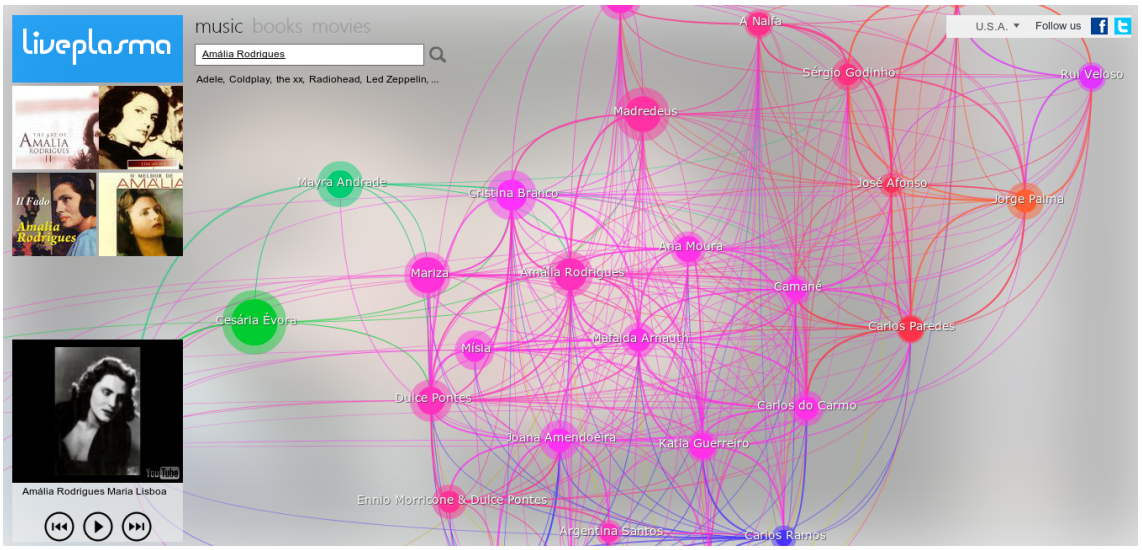


Figure 2.1: liveplasma: search result for "Amália Rodrigues"; upper left corner: artist albums; lower left corner: youtube's *mini-player*

2.2.1.2 Cons

The graph drawn from this simple search, is very cluttered with edges. Two nodes can have several connections between, which seems to overload the graph and making it very confusing.

Different colours are used, but their meaning remains unknown. One can assume that they represent the similarity between artists, but that is just speculation.

It can also be assumed that the size of the nodes (radius value) can be directly proportional to the artist's popularity, but that is, again, just speculation.

One critical detail is that the user cannot visually point out the search node in the graph, given the lack of visual distinction from the other nodes of the graph 2.1.

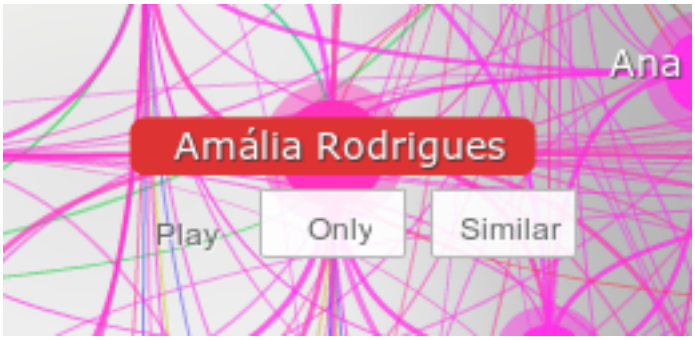


Figure 2.2: liveplasma: interface to start playing tracks. *Similar* button plays tracks from similar artists, whereas, the *only* button only plays tracks from the specified artist.

Figure 2.3: Tuneglue: menu for the node. Appears when the user clicks the node.

2.2.1.3 Summary

In short, liveplasma is not very user friendly. It uses too many colours and edges, which makes the user experience of searching for new music even harder than usual.

2.2.2 Tuneglue - audiomap.tuneglue.net

Tuneglue is another flash application that tries to explore the graphic visualization of network of related artists. Last.fm's metadata API is used to retrieve artist information.

When you start Tuneglue and search for an artist, say "Mariza", the user is presented with a single-node graph. By clicking the node, the user has four options (2.3): expand, releases, lock position and delete.

When you first expand a node, you get the root node with six child nodes 2.4.

So the first feature that brings the user experience to another level (in comparison with liveplasma) is that of graph editing. The user can expand, fix and delete every single node in the graph.

2.2.2.1 Pros

Tuneglue gives control to the user. On one hand, the user is able to craft a graph and tailor it to its needs. The user feels that graph is its own creation.

2.2.2.2 Cons

On the other hand, the user has the responsibility to create the whole graph, which might be too much trouble and deteriorate the user experience.

Again the root node is not highlighted, which might leave the user lost when the graph gets more and more complex.

2.2.2.3 Summary

Tuneglue takes the approach to give the user the power to create what he wants. But with no limit, the user can easily create a very complex graph that deteriorates the user experience.

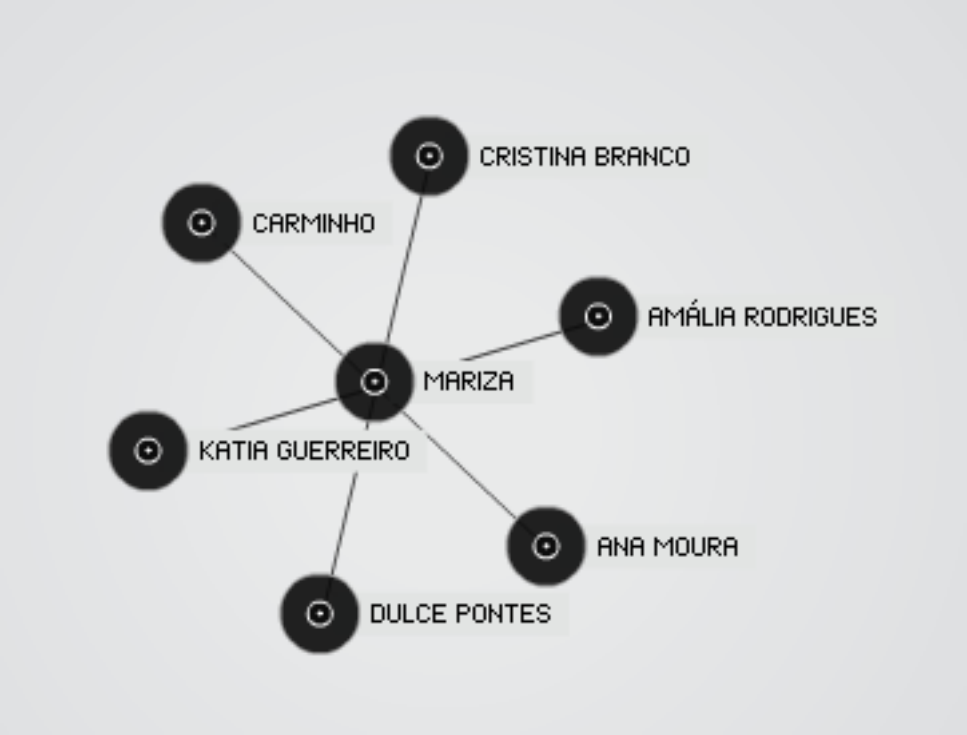


Figure 2.4: Tuneglue: graph after expanding the root node.

2.2.3 MusicRoamer - musicroamer.com

MusicRoamer is yet another flash application. Although it is similar to Tuneglue when it allows the user to expand the graph further and further, it also imposes some limits to the user to avoid getting the graph confusing.

2.2.3.1 Pros

There are three types of search 2.5):

Artist Search
The most used one.

Keyword Search
Search using keywords like genres and tags

Last.fm user search
The search result generates several graphs with the top artists of the user as the root nodes.

Figure 2.5: MusicRoamer: Search options. by artist; by keyword and by Last.fm username

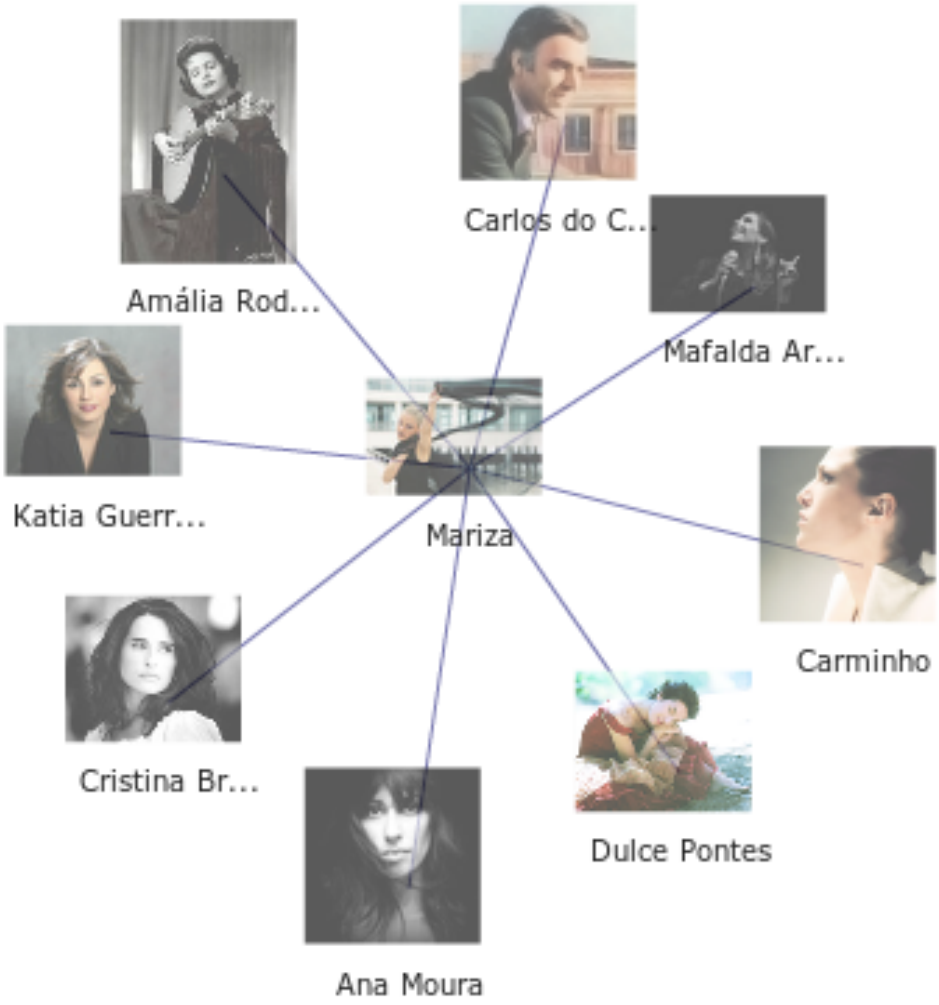


Figure 2.6: MusicRoamer: Visual representation of the artist graph

Independently of the search form used, the result will always be one (or more) graphs where the nodes are music artists.

MusicRoamer is worth mentioning because of the way it shows the graph. In 2.6 one can see the search result for "Mariza".

The images of the music artists are used to represent the nodes. This way, the user has a more friendly mind map of the resulting graph.

There is also some parameters (2.7) that the user can personalize to change the appearance of the graph: zoom; repulsion force between the nodes; size of the artist's images and the number of artist to be used as the branching value.

Figure 2.7: MusicRoamer: Personalizable parameters for the graph

2.2.3.2 Cons

MusicRoamer is a flash application which makes the interface less natural and fluid to a website user.

Another problem occurs when the user starts to expand more and more nodes. The graph starts to get confusing (2.8), the edges are drawn over the images, the artist's names start to get mixed in the images.

2.2.3.3 Summary

Although the MusicRoamer user has a lot freedom when creating the graph, the graph presentation if weak and not very æsthetically pleasant.

2.3 Summary

There is an uncountable number of services to discover new music. The ones presented in the previous examples have a visual representation in graph.

The following services have a some what interesting method to present the users with new music (not necessarily using visual tools):

- liveplasma.com
- audiomapa.tuneglue.net
- musicroamer.com
- discovr.info
- ifyoudig.net
- pitchfork.com
- hypem.com
- awdio.com
- 8tracks.com
- tastekid.com
- songza.com
- thesixtyone.com
- mog.com

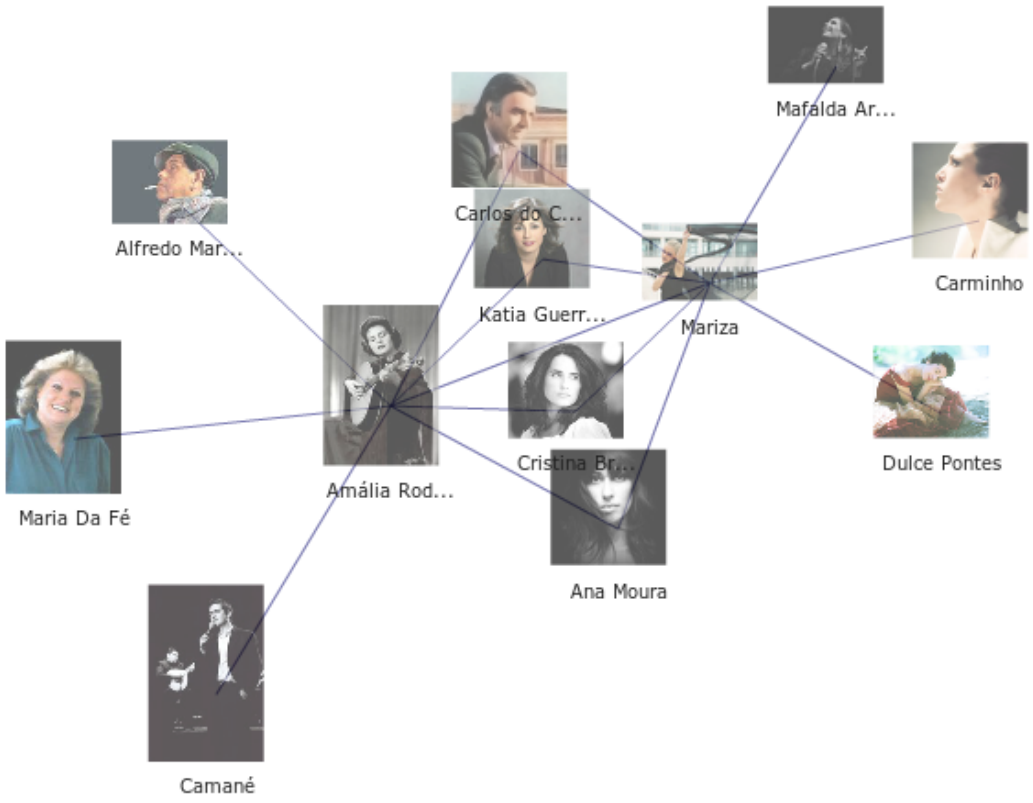


Figure 2.8: MusicRoamer: The graph after expanding one node

- stereogum.com
- gigfi.com
- jango.com
- soundcloud.com
- grooveshark.com
- rdio.com
- pandora.com
- music.google.com

The most important aspect to retain from the previous examples is that the bigger the branching value (ramification) of the graph, the more confusing and cluttered the graph becomes. One could say that the visual tool loses its initial purpose to help the user to discover new music.

A way to avoid that problem would be to force limits in the graph creation process.

Chapter 3

Context

The primal objective of this dissertation, as referred in chapter 1, is to develop one or more software modules that will improve Spotify Users' music discovery and recommendation experience using visual tools to represent the music artists' relations and Spotify's streaming service to provide high quality music stream.

The initial proposal was to develop a module that implements, at least, one of the following features:

1. Integrate Spotify's music stream into RAMA's website
2. Integrate information from the Spotify user into RAMA
3. Improve RAMA's features and design
4. Integrate the RAMA concept into a Spotify Application
5. Integrate RAMA's playlist generation into a Spotify Application
6. Integrate some of the above mentioned modules into a Mobile Application

The first three functionalities (1, 2 and 3) focus on improving RAMA using Spotify's API, i.e. to integrate Spotify into RAMA. Whereas 4 and 5 aim to integrate RAMA's concept into Spotify, through a Spotify Application (it would work as a plugin to Spotify's Desktop Client). The last one (6) would focus on implementing the previous functionalities into an Android, iOS or Windows Phone Application.

This chapter aims to analyse every single drawback of each possibility that affects the choice of which modules do develop, and on which environments it fits better: Spotify Application, Mobile Application, or RAMA improvements.

At first, Spotify's development environment will be introduced 3.1 in order to assess which tools are available for developers. Next, the available tools will be evaluated in order to determine which ones fit the proposed modules to be developed, mostly, through experiments.

By the end of this chapter the modules developed should be clearly stated, as well as which tools were used in the prototype.

The prototype should pursue the objective of contributing to an improved user experience when discovering new music taking advantage of visual tools that implement RAMA's concept.

3.1 Introducing Spotify

Spotify is a Music Streaming Service that allows the user, through an Internet connection, to listen to any track (if available in the user's country) in Spotify's catalogue. The service was launched in 2008 with a native desktop client application.

Now, the service has several types of clients available to the users: desktop client, webplayer and mobile applications.

Desktop Client Desktop version of Spotify, with Windows and Mac versions (and also a Linux preview version).

Webplayer Web version of Spotify. This was released in 2013, and spotify still advises the use of the native applications for a better user experience.

Mobile Applications The mobile applications are available for Android and iOS devices.

3.1.1 Development Tools

Spotify provides a set of tools[1] to develop Third-party Applications (websites, native applications and mobile applications) and Spotify Applications (that run inside Spotify's Desktop Client). There are five tools, each with different purposes.

3.1.1.1 Spotify Apps

Spotify Applications[2] are a special case in the whole set of tools provided by Spotify. These applications are designed to run *inside* the Desktop Client. Hence, its development is also inside the same environment.

Spotify users can run and install applications from the store called "App Finder". All the applications are free.

In 3.1 on can see the interface of the desktop client. In this case, the discovery mode's interface.

On the left side, in the menu, bellow the "App Finder" item, appears all the applications the user as installed from the store.

In 3.2 the official Last.fm application is opened. Note how the space filled by the applications are always the same.

The Applications' runtime environment is one of a browser-based. More specifically, powered by the Chromium Embedded Framework[3]. This means that the code to develop a Spotify Application follows the same principles as a web application: HTML, CSS and Javascript.

Spotify developed two Frameworks[4] to help developers create these applications: the API 1.x Framework[5] and the Views Framework[6].

The first one provides an interface to use object models, access metadata, control the player, among others. The second offers support for web components like buttons, lists, tabs, among others.

In order to develop the proposed modules 4 and 5, these are the most appropriate tools.

[1] http://developer.spotify.com/technologies
[2] https://developer.spotify.com/technologies/apps
[3] https://code.google.com/p/chromiumembedded
[4] https://developer.spotify.com/technologies/apps/reference
[5] https://developer.spotify.com/docs/apps/api/1.0/
[6] https://developer.spotify.com/docs/apps/views/1.0/

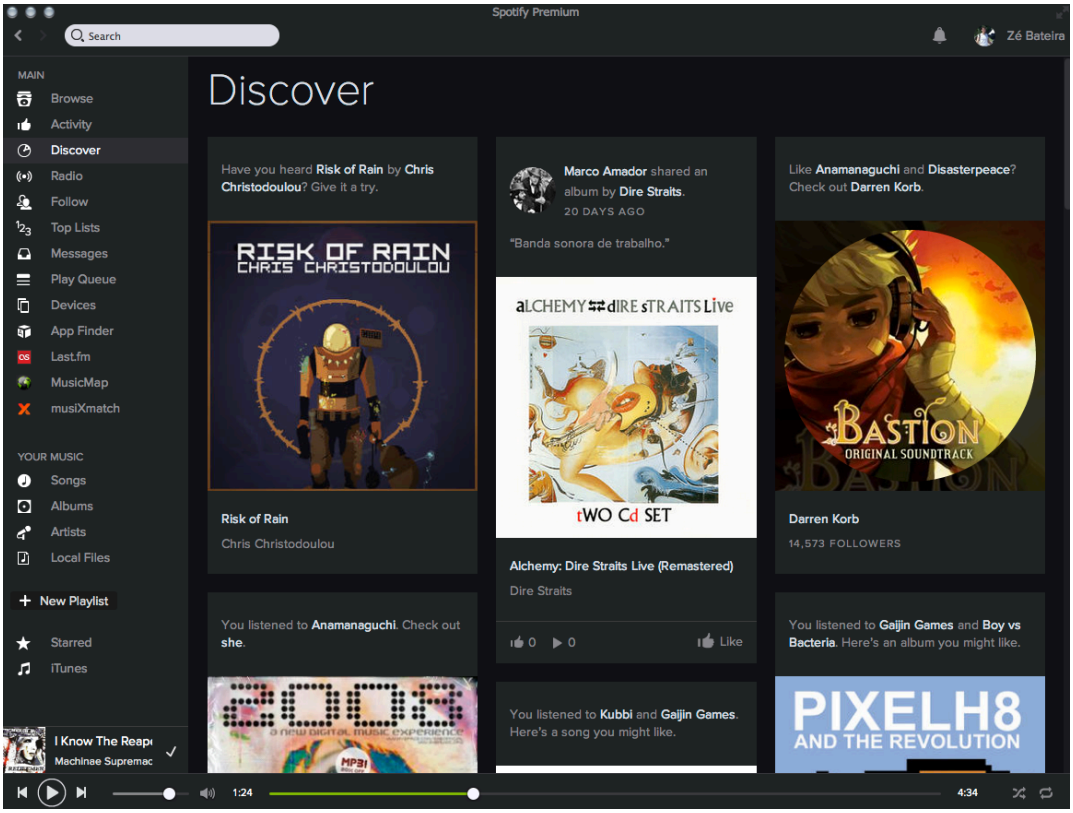


Figure 3.1: Spotify: desktop client's discovery mode interface.

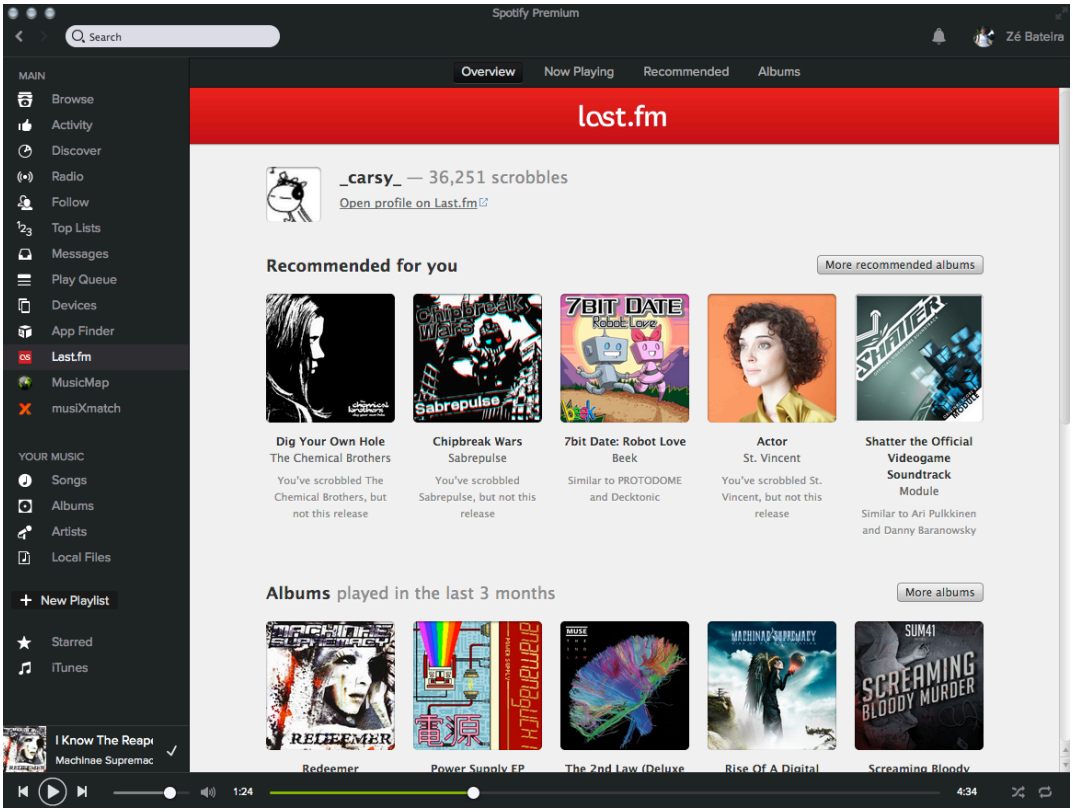


Figure 3.2: Spotify: Last.fm's Spotify Application opened.

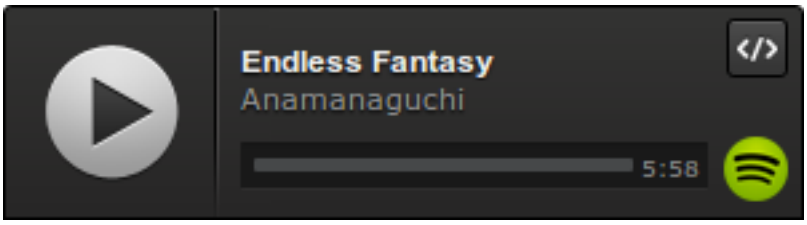


Figure 3.3: Spotify: *Play Button*.

3.1.1.2 Spotify Widgets

Spotify Widgets[7] are small web components that can be embedded in external websites. Spotify provides two components: *Play Button* (3.3) and a *Follow Button* (3.4)

However, there are some limitations. In Spotify, only logged in users can use the service (listen to tracks, etc). This also applies to these widgets - Even if they are in an external application, only Spotify users can interact with them.

This limitation does make sense in the case of the *Follow Button*, but the *Play Button* becomes useless to non-spotify users.

In truth, these widgets are nothing but an hyperlink to a Spotify Client (Web Player or Desktop). With the Play Button, the stream of tracks always played inside Spotify's environment, and not on external applications.

To embed a widget, it is only required to copy-paste Html code into the website, where appropriate:

Listing 3.1: Html code to embed the *Play Button*

```
1 <iframe
2   src="https://embed.spotify.com/?
        uri=spotify:track:1EsdqTsiQPaUJ82iy7KfS1"
3   frameborder="0"
4   width="300"
5   height="380" >
6 </iframe>
```

These widgets are useful to develop the proposed modules 1 and 3.

3.1.1.3 Libspotify SDK

Libspotify SDK[8] is an API that allows for third-party applications to include Spotify's services into them. However, not without some limitations to the users of these applications. The users are limited depending on the type of Spotify Subscription that they have

[7] https://developer.spotify.com/technologies/widgets
[8] https://developer.spotify.com/technologies/libspotify

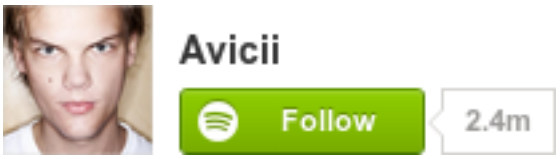


Figure 3.4: Spotify: *Follow Button* Allows the user to follow the music artist.

signed up to.

There are three different types of subscriptions, but the important part to retain, is the difference between being a Free Subscription Spotify User, and a Paid Subscription Spotify User (premium and unlimited subscriptions). As mentioned before (3.1.1.2), only Spotify users can interact with the widgets. That also applies to third-party applications that are using Libspotify SDK, which allow, for example, the user to login with their Spotify account. But in this case, not only they need to be Spotify users, they also need to have signed up to a paid Spotify subscription. And not only do the users need to pay to use the Spotify-powered application, but the developers as well.

This is a very restrictive environment, although Libspotify SDK comes in many different flavours[9].

This tool would be used to develop modules 1, 2 and 6.

3.1.1.4 Metadata API

The *Metadata API*[10] allows for applications to retrieve information from Spotify's music catalogue: tracks, albums, artists, playlists, and so on.

Requests to the database are done through HTTP and are of two types: *search*[11] e *lookup*[12].

To request detailed information of, e.g., an artist, the URI (used as the unique identifier) of that artist is required. Such ID is of the form:

`spotify:artist:<artist_id>`, where *artist_id* is the unique identifier of the artist.

Example:

`spotify:artist:65nZq8l5VZRG4X445F5kmN`, is the ID for the artist "Mariza".

There's also ID's for albums:

`spotify:album:5d1LpIPmTTrvPltx26TlEU` (album "Fado Tradicional" from "Mariza")

and for tracks:

`spotify:track:2vqYasauhDLVjTt7CGWK6y` (track "Fado Vianinha" of the previous album)

These URI schemes are compliant with Rosetta Stone's ID spaces[13].

First, to get this URI, one needs to search the database.

Search

The base *URL*:

`http://ws.spotify.com/search/1/album`, to search for albums.

For artists, *artist*, for tracks, *track*.

Examples:

[9] https://developer.spotify.com/technologies/libspotify/#libspotify-downloads
[10] https://developer.spotify.com/technologies/web-api
[11] https://developer.spotify.com/technologies/web-api/search
[12] https://developer.spotify.com/technologies/web-api/lookup
[13] http://developer.echonest.com/docs/v4#project-rosetta-stone

http://ws.spotify.com/search/1/album?q=foo
http://ws.spotify.com/search/1/artist.json?q=red+hot

The request response, by default, is formatted in *XML*, although, as the second example demonstrates, *JSON* is also supported.

Given the following query:
http://ws.spotify.com/search/1/artist.json?q=camane

The server responds with:

Listing 3.2: Results ordered by "popularity"

```
{
    "info": {
        "num_results": 2,
        "limit": 100,
        "offset": 0,
        "query": "camane",
        "type": "artist",
        "page": 1
    },
    "artists": [
        {
            "href": "spotify:artist:3MLPFTe4BrpEV2eOVG0gLK",
            "name": "Camane",
            "popularity": "0.27"
        },
        {
            "href": "spotify:artist:5Gwulm1LfURW7dbZD1V3zX",
            "name": "Sergio Godinho/Camane/Carlos Do Carmo",
            "popularity": "0"
        }
    ]
}
```

Lookup

When the URI is known, one can finally lookup detailed information about a database item. With the following *query*:
http://ws.spotify.com/lookup/1/.json?uri=spotify:artist:3MLPFTe4BrpEV2eOVG0gLK

The server responds with:

Listing 3.3: *lookup* of the artist "Camané"

```
{
    "info": {
        "type": "artist"
    },
    "artist": {
        "href": "spotify:artist:3MLPFTe4BrpEV2eOVG0gLK",
        "name": "Camane"
    }
}
```

This API is very useful to all the six proposed modules.

3.1.1.5 iOS SDK (beta)

The iOS SDK supports iOS Application developers. Although still in beta[14], this tool would be used to develop the proposed module 6. Much like the Libspotify SDK, this SDK provides the following APIs:

- User authentication
- Audio playback and stream management
- Metadata (artist, album, track) lookup including artwork
- Playlist management

3.1.2 Experiments

On a first hands-on experience with these tools, a single-page website was developed which allows the users to search and listen to music using Spotify's *Metadata API* and *Widgets*:

http://carsy.github.io/spotify-playground

In 3.5 one can see a search result and the *Widget Play Button* with the selected item.
Both tools turned out to be well documented and easy to use.
Another experiment was made in order to assert the potential of Spotify Applications. There was a need to know if the canvas element was well supported by Spotify's environment, because that is the preferred way to graphically draw a graph.
To test that, a simple application was created with the following code:

Listing 3.4: *iframe* element that allows to embed RAMA's website into the application.

```
1        <iframe src="http://rama.inescporto.pt/app" frameborder="0">
2        </iframe>
```

The final result can be seen in 3.6.
Although the *iframe* and *canvas* elements are supported, there are some that are not. This specific application is not usable since, for example, playing tracks from external sources is not allowed.
Nonetheless, there is a way to test which Html elements are supported, using an internal Spotify application. In 3.7 one can see the 100% supported canvas element.

3.1.3 Conclusion

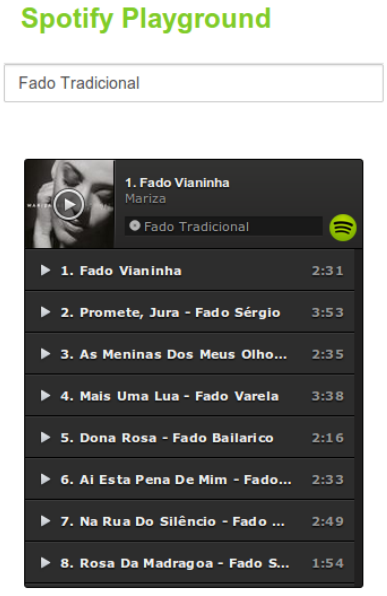
The developed prototype 3.6 revealed to be the most appropriate to well integrate Spotify and RAMA.
Given that, the proposed modules developed are 4 and 5.

3.2 Technologies used

The following technologies were used during the development of the application.

[14] https://developer.spotify.com/technologies/spotify-ios-sdk

(a) Search result for "Mariza"



(b) After selecting the album "Fado Tradicional" the *Play button* displays all of the album's tracks to be played in sequence.

Figure 3.5: Experiment with the *Metadata API* and the *Play Button Widget* (source code: github.com/carsy/spotify-playground)

Figure 3.6: RAMA's website embedded into a Spotify Application.

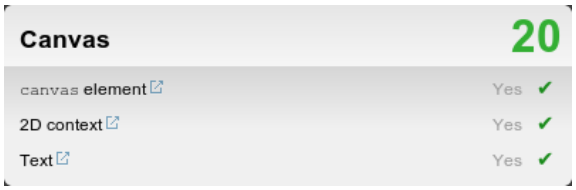


Figure 3.7: Test result for the canvas element.

3.2.1 Spotify Desktop Client

Spotify Applications are developed in its runtime environment - the Spotify Desktop Client.

To open a Spotify Application, locally, one writes the following in the search bar: spotify:app:rama

Where *rama* is the application identifier declared in the *manifest.json* file[15].

Example:

Listing 3.5: manifest.json: *BundleIdentifier* is the application's identifier; *Dependencies* declares the Application's API dependencies.

```
{
  "AppName": {
    "en": "RAMA"
  },
  "BundleIdentifier": "rama",
  "AppDescription": {
    "en": "RAMA: Relational Artist MAps"
  },
  "AcceptedLinkTypes": [
    "playlist"
  ],
  "BundleType": "Application",
  "BundleVersion": "0.2",
  "DefaultTabs": [
    {
      "arguments": "index",
      "title": {
        "en": "Home"
      }
    }
  ],
  "Dependencies": {
    "api": "1.10.2",
    "views": "1.18.1"
  },
  "SupportedDeviceClasses": ["Desktop"],
  "SupportedLanguages": [
    "en"
  ],
  "VendorIdentifier": "pt.inescporto"
}
```

There are useful options for development located in the "Develop" tab (3.8). The "Show Inspector" option opens the *Webkit Development Tools* (3.2.2) window.

[15] file located at the root of the project folder

3.2.2 Webkit Development Tools - webkit.org

The webkit tools provides a bundle of tools for web development.
Being the most important:

Inspector Allows to inspect the resulting Html and CSS and edit the code and see the application automatically reflect those changes (3.9).

Network Shows a timeline list of resources that where loaded from external sources (sometimes local) (3.10).

Profile Allows to identify which parts of the javascript code are being executed frequently, and which ones might be creating a performance issue (3.11).

Audit Helps to understand which CSS rules are not being used (3.12).

Console Javascript interpreter that also works as the log output for the application (3.13).

3.2.3 Gruntjs - gruntjs.com

Gruntjs is a Javascript task runner. It allows to automate most of the repetitive tasks when developing a website. Very useful for testing, compiling and code optimization.

3.2.4 Npmjs - npmjs.org

Package dependency manager for nodejs - Node Packaged Modules. Node packages will be used, since Gruntjs plugins are all nodejs packages (as well as Grunt itself).

A npm configuration file (*package.json*) allows to identify the packages that the application depends upon, as well as its versions.

Example:

Listing 3.6: *package.json*: "*" means that npm should always install the latest version of that package.

```
1  {
2    "name": "RAMA",
3    "devDependencies": {
4      "grunt": "~0.4.2",
5      "grunt-contrib-jshint": "*",
6      "grunt-contrib-jasmine": "*",
7      "grunt-contrib-watch": "*"
8    },
9    "version": "0.1.0"
10 }
```

3.2.5 Bower - bower.io

Bower is also a package manager, but oriented for web front-end packages.

3.2.6 vis.js - visjs.org

Javascript framework for visualization. It provides a few visual components, including graphs.

3.3 Summary

The final choice is to develop the Spotify Application.

Although the other proposals were also doable, the possibility to integrate a RAMA-like interface into Spotify's Desktop Client leaves a Spotify User more at ease with the environment.

The prototype should implement the proposed modules 4 and 5.

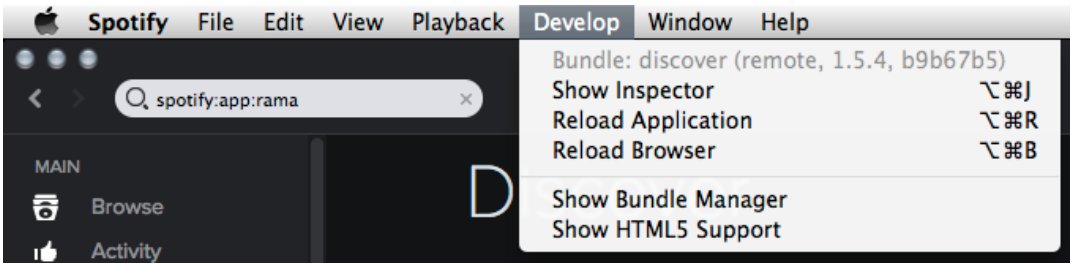


Figure 3.8: *Develop* Tab

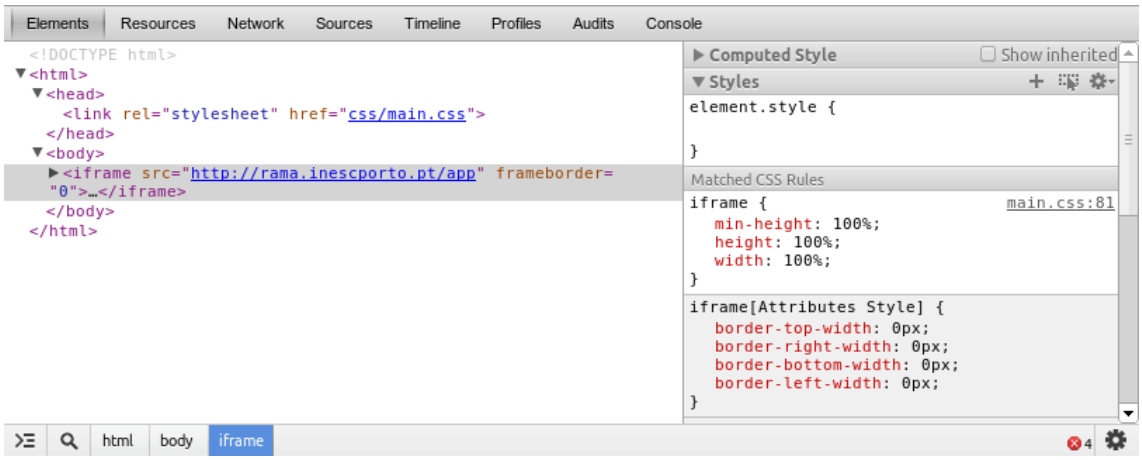


Figure 3.9: Webkit: *Inspector* tab view. Other tools available: *Resources, Network, Sources, Timeline, Profiles, Audits* and *Console*.

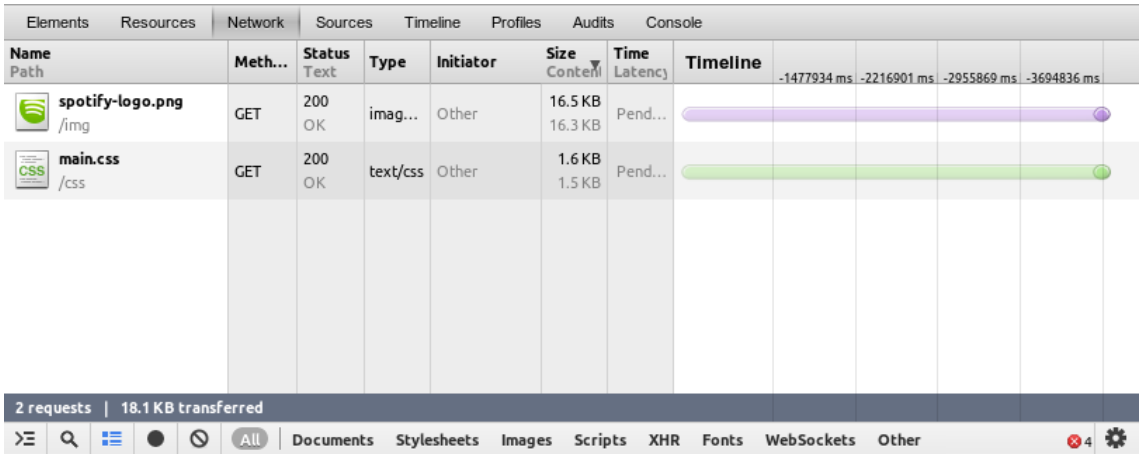


Figure 3.10: Webkit Network

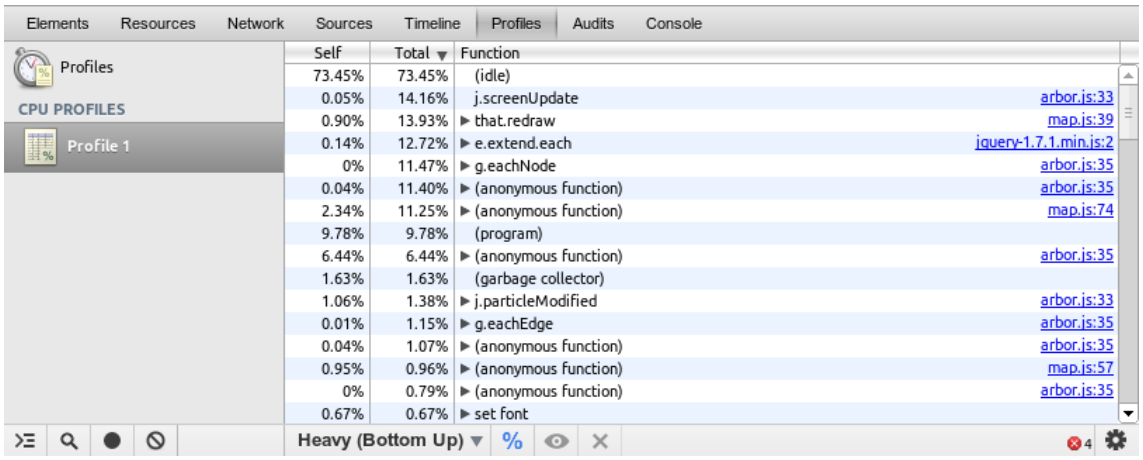


Figure 3.11: Webkit Profile: Canvas render functions are the ones taking up most of the processing cycles. However there is a JQuery function that used 12.75% of processing time, which might indicate a performance issue to be improved.

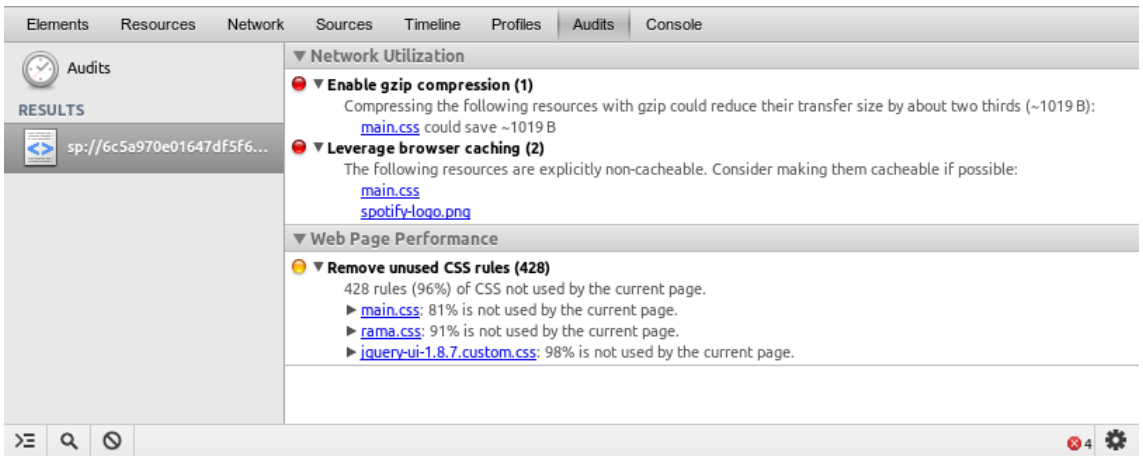


Figure 3.12: Webkit Audit: 96% of the CSS code is not being used indicates a issue to be solved.

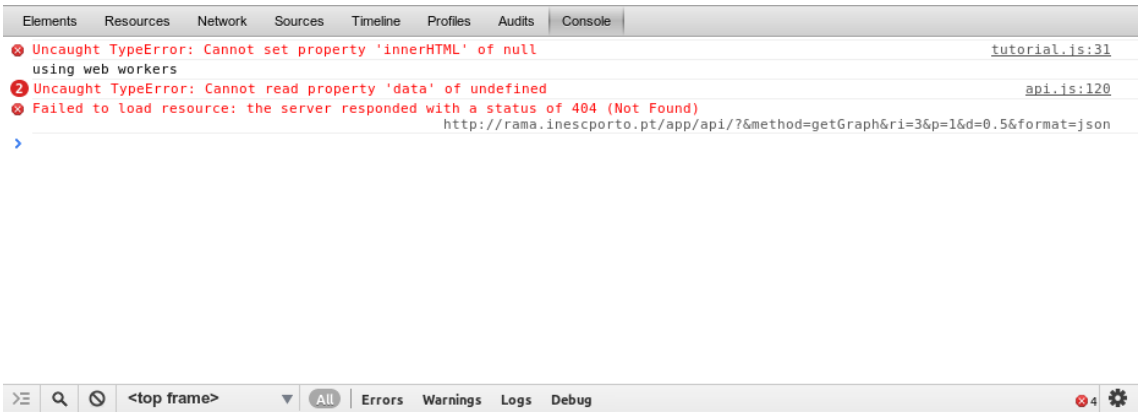


Figure 3.13: Webkit Console: Javascript errors are reported there (and highlighted in red as well).

Chapter 4

Implementation and Validation

In this chapter, further details about the methodologies used in the developed prototype, as well as in the the validation process, will be explored.

The Development Processes section will go into details about the processes used to maintain a sane development environment by taking advantage of several tools that automated most of the common tasks.

By this point, the validation of the prototype will be analysed, by explaining the performed user tests, as well the analysis of the results.

4.1 Prototype

4.1.1 Main Features

The main features of RAMA's Spotify Application are: visualization of a map of a network of connected artists; edition of the visualization parameters; edition of the graph (expand and new map functions); Tags overlay and music artist info.

4.1.1.1 Visulization of the Artists Map

The application automatically draws the map with the current playing artist as the main node, as seen in Figure 4.1.

The graph-like structure is created by recursively fetching a list of related artists from each artist. Once a certain pre-established limit of recursive levels is reached[1], the algorithm stops.

The graph creation algorithm is as follows:

Listing 4.1: Simplified graph creation algorithm in Javascript (duplicate nodes checking is encapsulated in the insertNode function)

```
1 function buildGraph() {
2   // create a node with the root artist and insert it into the graph
3   this.insertNode(this.rootArtist);
4
5   // start constructing the graph recursively
6   this.expandNode(
7     this.depth - 1,
```

[1] depth value of a graph

```
8       this.rootArtist
9     );
10  }
11
12  // Expands the node of the parent artist by this.branching.
13  // It recursively decreases the depth parameter.
14  function expandNode(depth, parentArtist) {
15    var node = this.getNode(parentArtist);
16
17    // after expanding, the node will stop being a leaf
18    node.isLeaf = false;
19
20    // retrieve this.branching number of childs of the parent artist
21    var relatedArtists = parentArtist.getRelatedArtists(this.branching);
22
23    // for each child artist, insert and create a node into the graph
24    // and do the recursive call for the child, but with decreased depth.
25    for (var childArtist in relatedArtists) {
26      this.insertNode(childArtist);
27
28      // note that the stop condition of the recursion is depth <= 0
29      if (depth > 0)
30        expandNode(depth - 1, childArtist);
31    }
32  }
```

This algorithm, albeit simplified, represents the basic flow when constructing a graph, or more specifically, a tree. Since that, in this case of study, the direction of the edges of the graph is not relevant in any way to the artists' map, all of the edges are considered to be undirected.

Assuming that the insertNode() function checks for duplicate nodes, i.e., it only inserts unique nodes into the graph, then the resulting graph is one of a tree, since there are no simple cycles in the graph. An example of this behaviour can be seen in Figure 4.2.

This approach, however, is not showing all of the available information. Given this same example (Figure 4.2), the artist node "Stone Temple Pilots" is a child of the node artist "Faith No More". The algorithm inserted the latter first into the graph. After that, when retrieving the childs of "Jane's Addiction", "Stone Templo Pilots" is contained in that list, and so the insertNode() function discards the node since it is a duplicate. But that means that there is a connection between the both of them that is being discarded.

Graph's Tree Mode

To build a graph with all the connections that exist between all of the artists in the graph, the insertNode() function would need to insert the missing edge into the graph by analysing the current graph state. This method creates a graph by definition, while the previous one created a tree graph. An example of this behaviour can be seen in Figure 4.3.

From now on, if the graph is a tree, it will be said that the algorithm is using the Tree Mode. *So the example 4.2 is a* Tree Mode *example, and the examples 4.3 and 4.4 are not* Tree Mode *examples*[2].

[2] Note that, sometimes, even if the algorithm is not on tree mode, it might generate a tree, simply because there were no missing edges between the nodes to be added to the graph.

Note (4.3) how "Stone Temple Pilots" is now a child of both "Faith No More" and "Jane's Addiction". Also note that the same behaviour is visible with the nodes "Mad Season" and "Screaming Trees".

At this point, the user's perception of the artist "Stone Temple Pilots" is now different from the previous example (Figure 4.2). These added connections are, in a way, clustering together the most connected nodes. In this particular example, one could see an improvement with this approach: it contributes to the user's perception of the artist's network by making sure the user knows that those specific artists are more connected between them, than any others. The fact is that only three extra edges where added, meaning there is not much visual clutter in the visualization.

However, that might not always be the case.

One could argue that the "Mad Season" artist node is already disturbing the visual representation by being drawn over an edge.

In fact, with the exact same values of branching, depth and tree mode off, when creating a graph for the artist "Mariza", the visual clutter is so strong that the graph becomes confusing and not very helpful: Figure 4.4.

The cluster of nodes in the center makes it clear that those specific nodes are really connected between them. But then, the edges, that are forcing those nodes together, are creating a visual clutter that might not be so desirable. Instead of creating a visual map of the artists' network for the user to perceive, explore and change in its own way, the representation is more contained, static and not so visually appealing.

So on one hand, the *clustering* of the nodes seems like an interesting feature. It allows for a much more in-depth user access to the underlining information of the related artists. But on the other hand, the focus of the visualization (to show a map of the artists' network) is shifting to this cluster-like representation.

Both perspectives are advantageous depending on the data they are operating on (as seen in the previous examples). So instead of choosing only one method, both were chosen: by default, the graph creation algorithm is on tree mode, and the user can turn it off from a settings menu. This allows for the user to choose the visual representation method that suits its needs.

Depth and Branching values

The depth and branching values have been mentioned before in this chapter, but not further explained.

The **depth** value of a graph determines how deep the recursive algorithm is. The previously presented example (4.2) has a depth value of 2. The example 4.5 has a depth value of 3. In short, depth is the maximum distance between the root node and any other node in the graph.

The **branching** value of a tree graph determines the maximum number of child nodes a node can have. The example 4.2 has 4 branching.

4.1.1.2 Visualization Parameters

The visualization parameters are the branching and depth values, as well as the option to enable/disable the tree mode.

To allow the user to change these values, a settings menu was added to the application, and can be seen in Figure 4.6. When the user changes the parameters, the graph refreshes

accordingly.

4.1.1.3 Graph Edition

The available features to edit the graph are as follow: expand node, delete node and create a new map. These interactions are available in the Artist Menu (4.1.1.4).

Expand node
This option allows the user to expand a node further (ignoring the graph's branching value). Given the previous example 4.2, if the "Dispatch" node gets expanded, it results in 4.7.

To put it simply, to expand a node, one performs one iteration of the graph creation algorithm.

Delete node
This options allows the user to delete a node from the graph. Useful for when the user wants to construct the graph to its needs.

New map
The user may choose to create a whole new graph from a another node. This way the root node will be the selected node.

4.1.1.4 Artist Info

The user is able to see additional information about artists in the Artist Menu (4.8) such as its popularity value, albums and tags, as well as perform the *expand* and *new map* functions described in 4.1.1.3.

When the user selects a node by clicking on it, the artist menu updates the displayed information.

The *popularity* and the *albums* are metadata information retrieved from Spotify's API framework (3.1.1.1). The tags, however, are not supported by Spotify's API. For that, Echonest's API was used.

Echonest's Terms
Echonest's tags are very similar to the most commonly known *music genres* (like jazz, country, rock), but they also might include more alternative descriptive terms (progressive metal, symphonic, soundtrack).

The connection between the two APIs is possible thanks to the Project Rosetta Stone[3]. For example, to retrieve the terms[4] of an artist from Echonest's metadata API using the Spotify's Artist ID, the following query is used:

`http://developer.echonest.com/api/v4/artist/terms?api_key=API_KEY&format=json&sort=weight&id=spotify-WW:artist:65nZq8l5VZRG4X445F5kmN`

The *id* parameter is similar to the Spotify's URI scheme (3.1.1.4), and this allows for retrieving extra information about the artist that Spotify's API does not provide.

[3] `http://developer.echonest.com/docs/v4#project-rosetta-stone`
[4] Echonest calls it terms. From now on, terms and tags will be used interchangeably

Sort is another interesting parameter. In general, artists have more than 10 or 15 terms. Each term has a value of *frequency* and a value of *weight*, and both are float values that range between zero and one. No official documentation was found to explain what do these values represent, but Paul Lamere[5] explains[6] that:

> *term frequency is directly proportional to how often that term is used to describe that artist*
>
> *term weight is a measure of how important that term is in describing the artist*

So given this, one can conclude that by sorting the terms of an artist by its frequency value, the top terms will be more general (rock, pop, jazz) and not very descriptive or specific of an artist. And by sorting the tags by weight, one will get the most descriptive tags of that specific artist.

Clearly, in this case, for the Artist Menu the weight parameter is very helpful, and so, the sort parameter used for the query is *weight*. As seen in 4.8, the tags shown are very descriptive of the artist (grunge, 80s, hard rock).

4.1.1.5 Tags Overlay

The tags overlay menu (4.9) is meant to enhance the user's perception on the displayed artists' nodes regarding their tags.

These tags are the same ones used in the Artist Menu (4.8).

The tags are selectable, and so, when clicked, the respective artist nodes that are described by those tags, are highlighted (as seen in Figure 4.9).

The tags shown in this menu, are just a small sample of the artists' tags of the whole graph. For example, for the graph in the Figure 4.10, the total count of unique tags in the whole graph is 93. To select which tags to show, those same 93 tags were sorted by frequency in the graph, and then the top 12 ones are shown.

This way, the tags in the overlay are significant for the graph, by helping the user to visually group together some related artists.

4.1.2 Development Processes

The primal objectives for the RAMA Spotify Application were to implement the basic functionalities that RAMA's website[7] offers.

Incrementally, they were implemented, tested and added to the application. A timeline (chronological order from bottom to top) with the added features (each update release) can be seen in

`https://github.com/carsy/rama-spotify/releases`

Unit tests were added for each feature. The tests were run locally to ensure that nothing broke mean while. Also to ensure that every release had the tests passing, the project was integrated in Travis's Continuous Integration[8] system:

[5] Paul Lamere is Director of Developer Platform at `the.echonest.com` (`http://the.echonest.com/company`). Also blogs here: `http://musicmachinery.com`

[6] thread in the forum for Echonest's API users - `http://developer.echonest.com/forums/thread/353`

[7] `http://rama.inescporto.pt/app`

[8] `http://travis-ci.org`

http://travis-ci.org/carsy/rama-spotify.

This way, after every push of a new release, the project gets build and the unit tests run automatically.

The solution was also submitted to validation several times during its development. After each release, a constant small group of test subjects (3 people) reviewed the state of the application (alfa testers). Their feedback would be carried over the next release: bug fixes, improvements, etc. This was the testing cycle that allowed to validate the implemented features. Given this iterative process of implementation and feedback, the solution would be better prepared for future occasional beta-testing.

4.2 Validation

In order to validate the proposed method, user tests were done with Spotify users and non-Spotify Users.

4.2.1 User Tests

The proposed application is targeted at Spotify Users. However, non-Spotify users also tested RAMA's Spotify Application after a few moments of using Spotify's interface.

During the test, the user was asked to first use Spotify only. The task was to find a couple of new artists that the user liked. This first task should take no more than 7 minutes.

Next, the user was asked to open the RAMA application and do the same.

Given this, the user is forced to compare the two approaches: discovering new music with Spotify only, and discovering new music with RAMA's Spotify Application.

4.2.2 Data Analysis

4.3 Summary

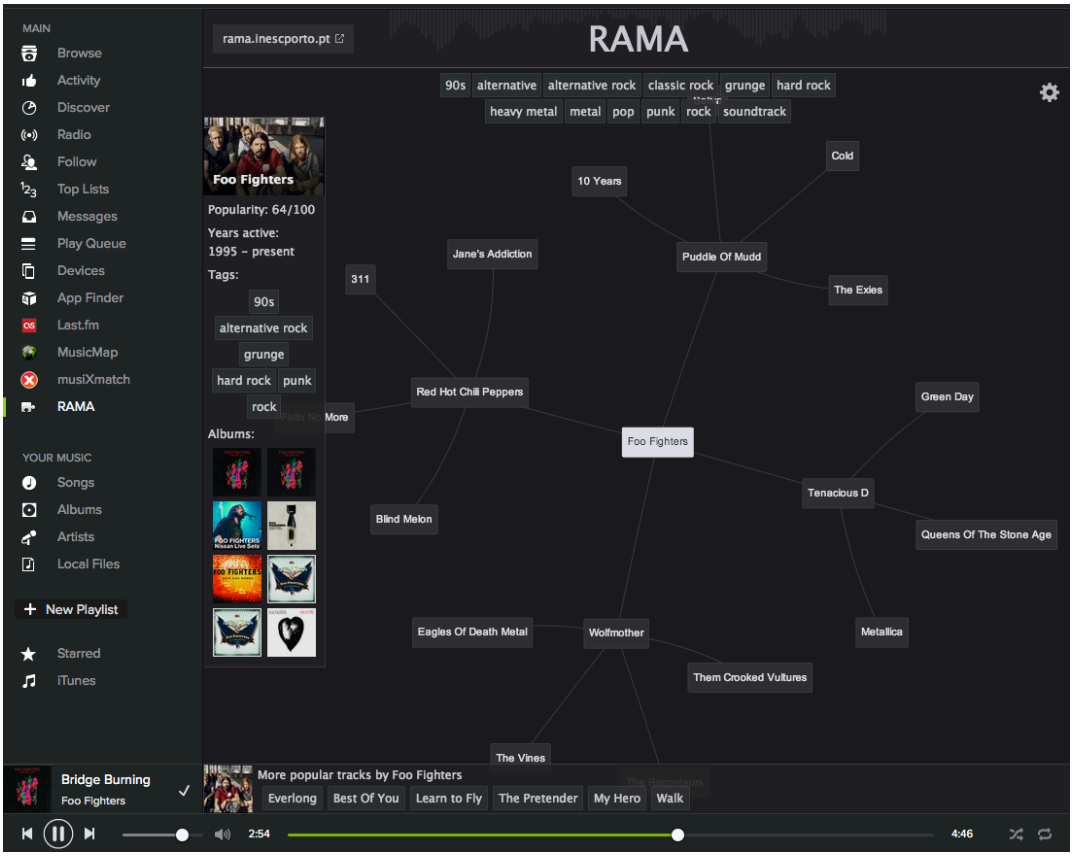


Figure 4.1: The first drawn graph uses the current playing artist (lower left corner) as the root node.

Figure 4.2: Graph created like a tree with "Red Hot Chilli Peppers" as the root node.

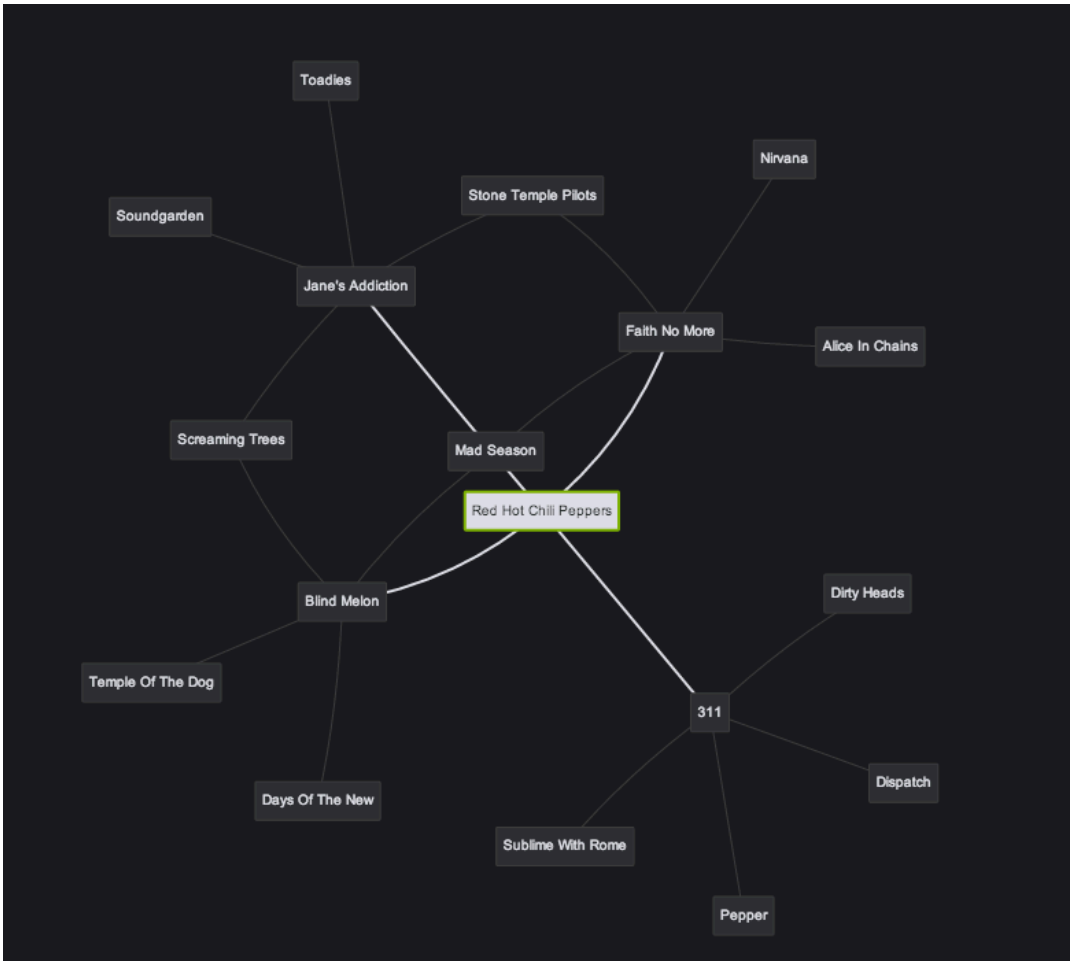


Figure 4.3: Graph created with all the connections with "Red Hot Chilli Peppers" as the root node.

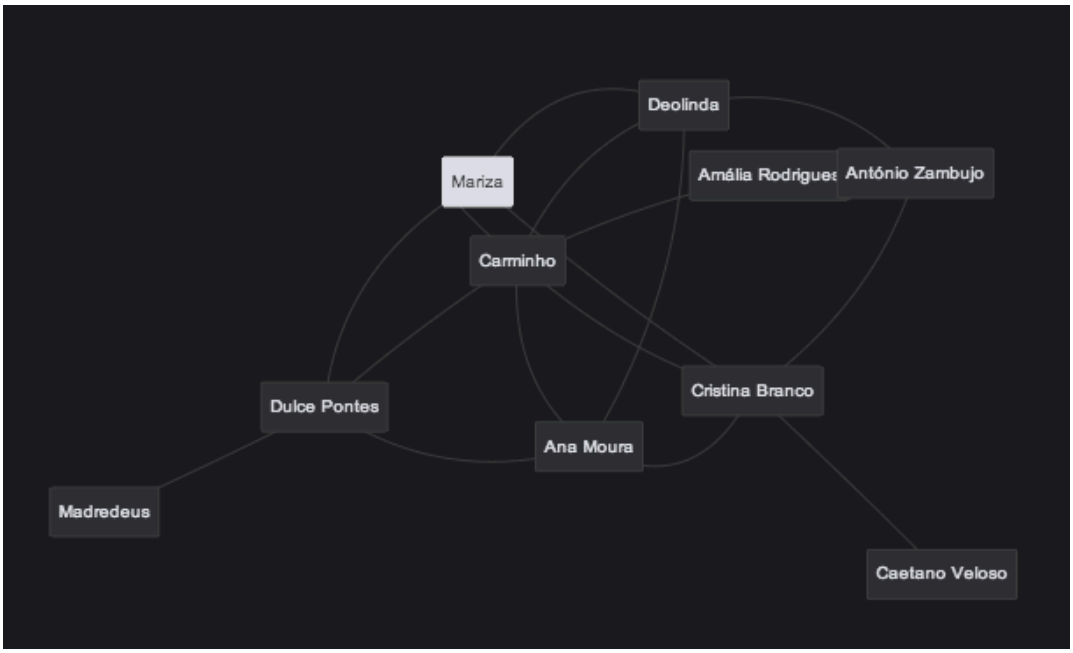


Figure 4.4: Graph created with all the connections with "Mariza" as the root node.

Figure 4.5: Graph with depth value of 3

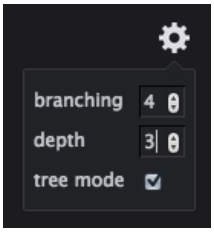


Figure 4.6: The settings menu that allows the user to change the visualization parameters.

Figure 4.7: "Dispatch" artist node expanded.

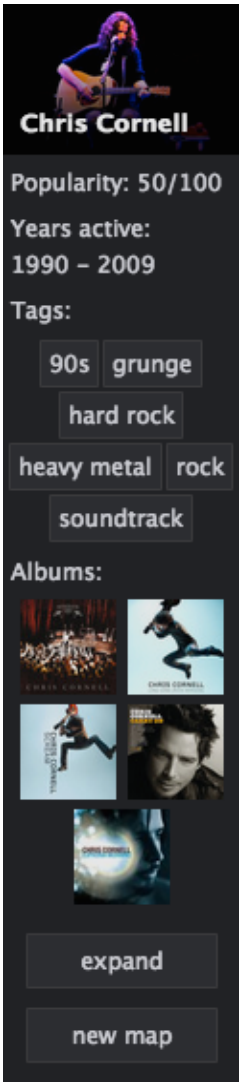


Figure 4.8: Artist Menu with information about "Chris Cornell"

Figure 4.9: Tags overlay for the displayed graph. When the tag "grunge" is selected the corresponding artist nodes are selected

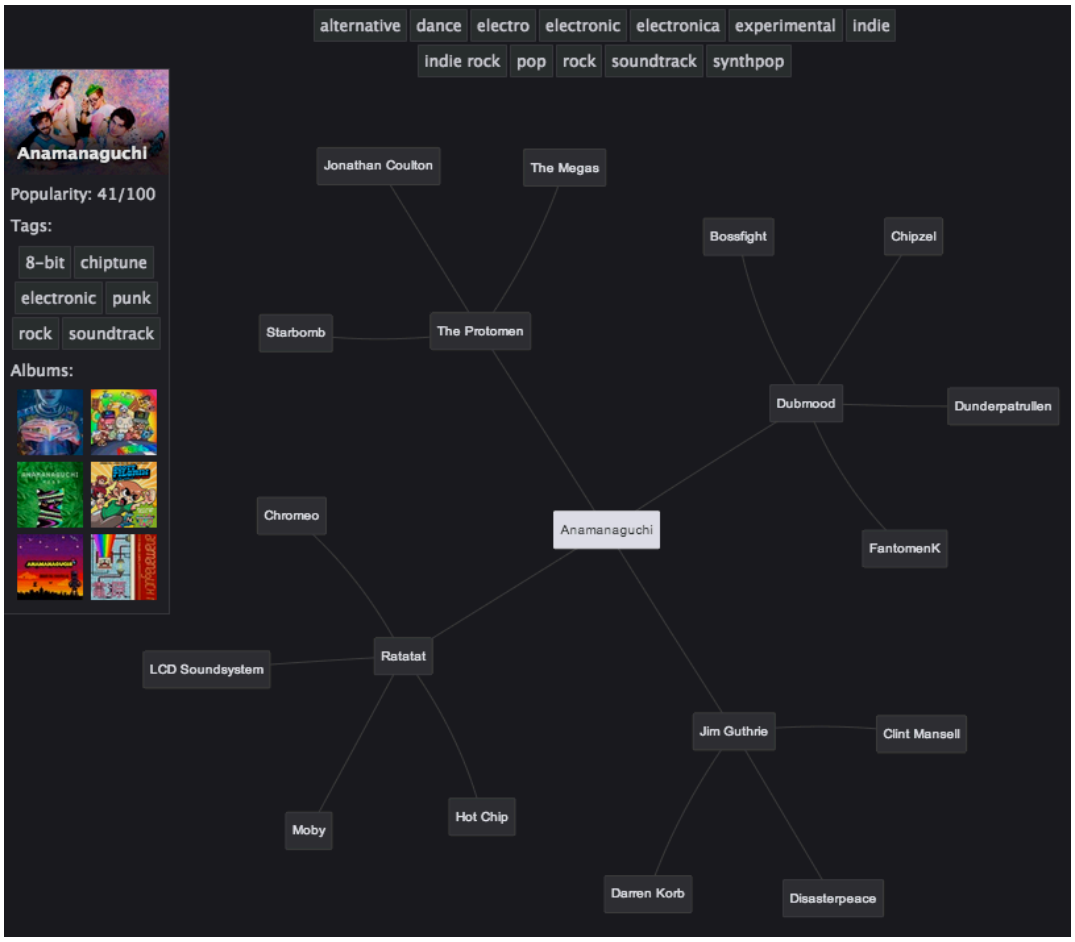


Figure 4.10: Graph for "Anamanaguchi". The tags shown above are only but a small sample of all the tags of all the artists in the graph

Chapter 5

Conclusions

References

[1] P. Lamere. Creating transparent, steerable recommendations. 2008.

[2] BG Costa, Fabien Gouyon, e L Sarmento. A Prototype for Visualizing Music Artist Networks. 2008. URL: http://www.inescporto.pt/~fgouyon/docs/CostaGouyonSarmento_ARTECH2008.pdf.

[3] L Sarmento e EC Oliveira. Visualizing networks of music artists with rama. *International Conference on Web ...*, 2009. URL: http://repositorio-aberto.up.pt/bitstream/10216/15194/2/18675.pdf.

[4] Diogo Costa, Luis Sarmento, e Fabien Gouyon. RAMA : An Interactive Artist Network Visualization Tool. (i):2, 2009. URL: http://ismir2009.ismir.net/proceedings/LBD-2.pdf.

[5] Fabien Gouyon, Nuno Cruz, e Luis Sarmento. A last.fm and youtube mash-up for music browsing and playlist edition. 2011.