Faculdade de Engenharia da Universidade do Porto

Spotify-ed: Music Recommendation and Discovery in Spotify

José Lage Bateira

Working version

U. PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

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 users 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 and/or generation of playlists, others try to expand their music catalogue even further, or focus on a more personalized music recommendation. And these systems 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. A possible solution to this problem is to represent the artist's similarities 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 can catch 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. The use of Youtube stream for audio thus needs to be replaced, and Spotify can provide a quality stream and an accurate music catalogue. But how can RAMA and Spotify be integrated?

To address this question, this thesis proposes a Spotify App for RAMA. 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 grids)? This is the main question that this dissertation seeks 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.

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

List of Tables

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 users 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.

There is always something that makes a music streaming service different from the others. Some services focus on creation and/or generation of playlists (8tracks [1]), others try to expand their music catalogue even further (Spotify [2], Rdio [3]), while others focus more on personalized music recommendations (Pandora [4]). 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 [5].

A possible solution to this problem is to represent the artist's similarities 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)[2], a project developed at INESC Porto, uses [6] [7] [8] [9].

1.2 Motivation

From a single search, the original RAMA system 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

[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] RAMA: http://rama.inescporto.pt

stream needs to be replaced. From the available services that provide a vast music catalogue, Spotify[3] provides a good quality stream and a good developer support for creating Spotify powered Applications.

But how can RAMA and Spotify be integrated?

1.3 Goals

There are several possibilities that Spotify has made available for developers[4] 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 the existence of some restrictions when using some APIs[5], there are several aspects to take into account when choosing which API to use.

The initial proposal was to develop a software 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

All possibilities will be explored with further detail in Chapter 3. In the end, this dissertation proposes a Spotify Application [11] (module 4) that works like a plugin to the Spotify's Desktop Client, i.e., it should add something to Spotify. This is a very appealing solution: Spotify Users will have the chance to continue using Spotify as they would normally do, 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 like RAMA.

After specifying the requirements, a prototype will be developed. This approach requires a solution to 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? To answer that question and to evaluate and validate the final prototype, end-user testing will be done to compare the user experience of discovering new music with or without the developed application.

1.4 Methodologies

The following work methods consist of four phases: initial research on the state of the art of the current applications that are similar to RAMA's approach; contextualization

[3] http://spotify.com

[4] https://developer.spotify.com

[5] for example, LibspotifySDK [10] requires the developer and the user of the application to have a premium account.

of the Spotify's environment given the tools available for users and developers; definition and implementation of the prototype's user requirements and validation of the developed prototyped by users.

The initial state of the art research was done to assess the main features of the music services that offer any sort of music discovery interface. There are an uncountable number of them, and so, focus was given to the ones that use visual tools. Their main features were analysed, as well their pros and cons.

Next, Spotify's environment was introduced from the user's perspective and the developer's perspective. At first, it was important to establish how a Spotify user goes about discovering new music, i.e., what tools are available. After that, the available tools for developers of Spotify-powered applications were analysed in detail. The goal for the end of this phase was to determine what type of application to be developed as the prototype.

By this point the user requirements of the application were defined and implemented. The tools used in the prototype's development were important to help automate most of the repetitive tasks: source code version management, testing, package managing, building and deploying. All of the development processes were important to keep a maintainable development environment. After each feature implementation, a small constant group of alpha-testers gave feedback and suggestions to be introduced in the following version of the prototype - this was the main development cycle that continuously made the prototype evolve into the defined user requirements.

After the prototype implemented the defined user requirements, it was submitted for user evaluation. The beta-testers were asked to discover new music with the developed application and then fill a short questionnaire that was meant to capture their first hands-on experience with the prototype. The results of the experiment were analysed to interpret the success to the application in meeting its goals.

1.5 Report's Structure

Thesis report presents 4 additional chapters:

Chapter 2: State of the Art

Initial research on the current state of the art. Detailed analysis on the services that provide a platform for users to listen and discover new music. Although, focus will be given to the ones that use visual tools.

Chapter 3: Contextualization

Detailed analysis of the Spotify environment from the users perspective (applications available, e.g.) and from the developers perspective (available APIs, e.g.) in order to give a much more insightful view when determining the feasibility of the prototype's requirements.

Chapter 4: Implementation and Validation

Definition and implementation of: the prototype's main features/requirements; the development processes and the user validation processes.

Chapter 5: Discussion and Future Work

Discussion of the results and definition of future work to be done in the prototype (improvements, features, etc).

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 will 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[1]) 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*[2] application that not only allows users to see a graph of music artists, but also of books and movies. The interaction with the graph is very limited: 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 Figure 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 Figure 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 tool has two interesting aspects to it:

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

[1]Last.fm: http://last.fm/api
[2]http://get.adobe.com/flashplayer

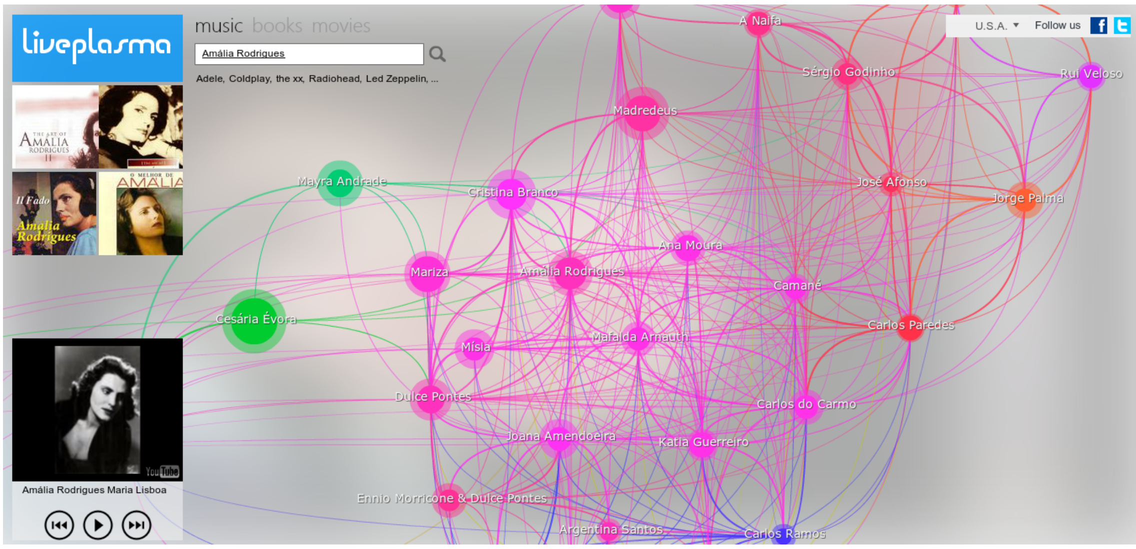


Figure 2.1: liveplasma: search result for "Amália Rodrigues"; upper left corner: artist albums; lower left corner: youtube's *mini-player*

2.2.1.2 Cons

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

Different colours are used for the nodes and edges, 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 (Figure 2.1).

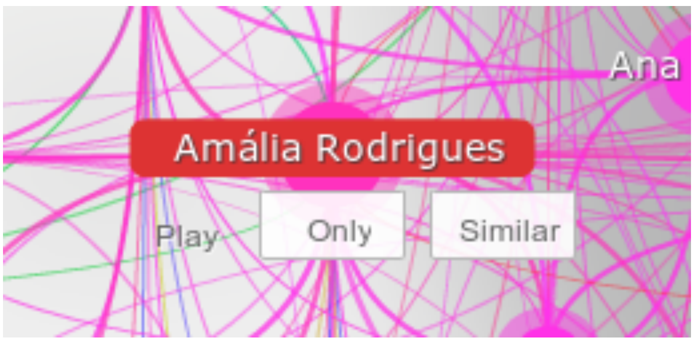


Figure 2.2: liveplasma: interface to start playing tracks. *Similar* button plays tracks from similar artists, whereas, the *only* button only plays tracks from the specified artist.

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 it would be, if using a more standard mode of discovery.

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 a user starts Tuneglue and searchs for an artist, say "Mariza", they are presented with a single-node graph. By clicking the node, the user has four options (Figure 2.3): expand, releases, lock position and delete.

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

When expanding a node, the user sees the root node with six child nodes in Figure 2.4.

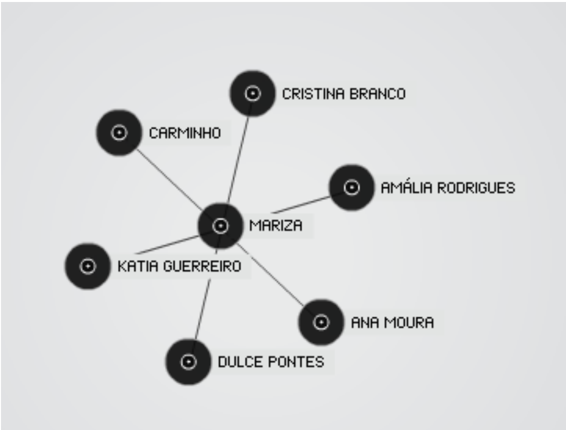


Figure 2.4: Tuneglue: graph after expanding the root node.

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, lock and delete every single node in the graph.

2.2.2.1 Pros

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

2.2.2.2 Cons

However, 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/she wants. But with no limit, the user can easily create a very complex graph that deteriorates the user experience.

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 making the graph too confusing.

2.2.3.1 Pros

There are three types of search (Figure 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

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 Figure 2.6 one can see the search result for "Mariza".

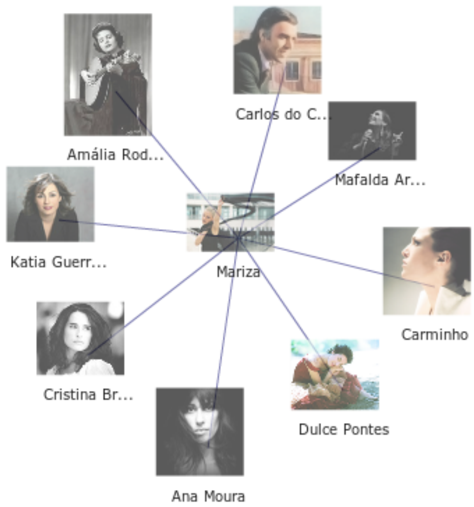


Figure 2.6: MusicRoamer: Visual representation of the artist graph

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 are also some parameters (Figure 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 (Figure 2.8), the edges are drawn over the images, the artist's names start to get overlap other artist's images.

2.2.3.3 Summary

Although the MusicRoamer user has a lot freedom when creating the graph, the graph presentation is weak and not very aesthetically pleasing.

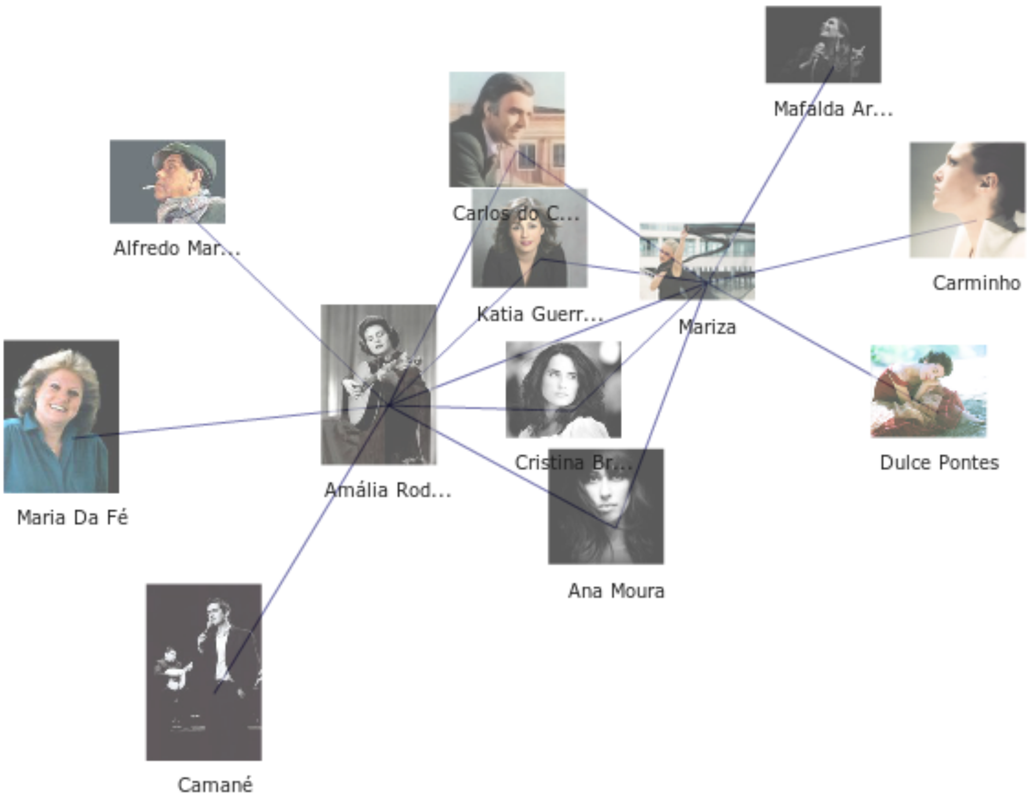


Figure 2.8: MusicRoamer: The graph after expanding one node

2.3 Summary

There are 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 somewhat 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
- 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 greater the visualization parameters (branching and depth) 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

Contextualization

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. The aim of this chapter is to compare and contrast these proposed features, towards determining which among them will be pursued in this thesis, either: Spotify Application, Mobile Application, or RAMA improvements.

At first, Spotify's user environment will be introduced (3.1), followed by Spotify's Development Tools (3.2) in order to assess which tools are available for developers. Next, the available tools will be evaluated, through experiments (3.3), in order to determine which ones fit the proposed modules better.

By the end of this chapter the modules developed should be clearly stated, as well as which development tools will be used in the prototype. The prototype should pursue the objective of contributing to an improved user experience of discovering new music by 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, although spotify still advises the use of the native application for a better user experience.

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

3.2 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.

3.2.1 Spotify Apps

Spotify Applications [11] are a special case in the whole set of tools provided by Spotify. These applications are designed to run *inside* the Desktop Client. Spotify users can run and install applications from the store called "App Finder". All the applications are free. In Figure 3.1 one 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 Figure 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 [12]. 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[2] to help developers create these applications: the API 1.x Framework[3] and the Views Framework[4]. 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.

3.2.2 Spotify Widgets

Spotify Widgets [13] are small web components that can be embedded in external websites. Spotify provides two components: *Play Button* (Figure 3.3) and a *Follow Button* (Figure 3.4)

[1] http://developer.spotify.com/technologies
[2] https://developer.spotify.com/technologies/apps/reference
[3] https://developer.spotify.com/docs/apps/api/1.0/
[4] https://developer.spotify.com/docs/apps/views/1.0/

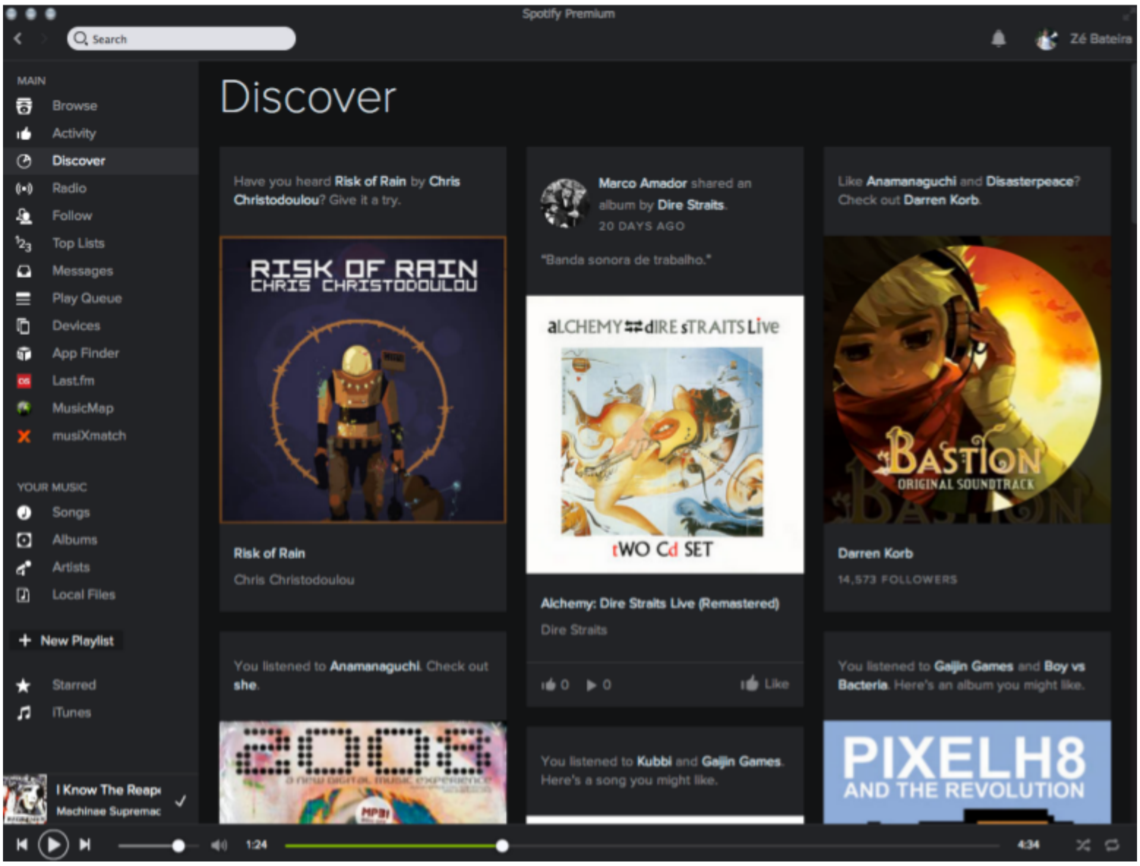


Figure 3.1: Spotify: desktop client's discovery mode interface.

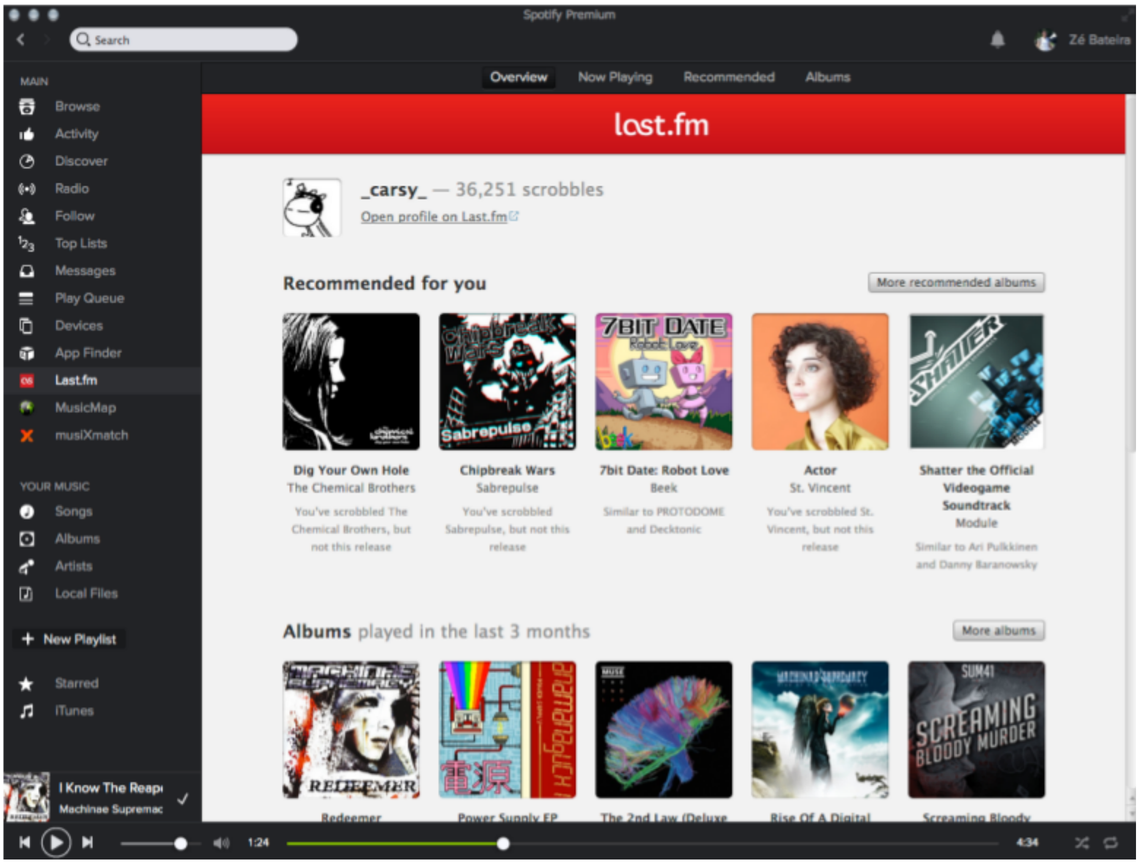


Figure 3.2: Spotify: Last.fm's Spotify Application opened.

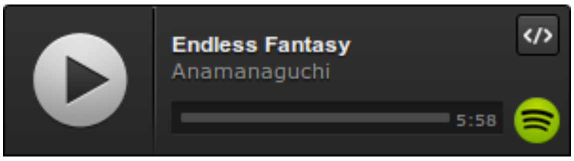


Figure 3.3: Spotify: *Play Button.*

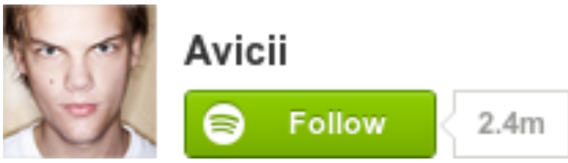


Figure 3.4: Spotify: *Follow Button* Allows the user to follow the music artist.

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 plays 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:

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

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

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

3.2.3 Libspotify SDK

Libspotify SDK [10] 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 signed up to.

There are three different types of subscriptions [14], 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, only Spotify users can interact with Spotify Widgets (Figure 3.2.2). 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 do 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 [15].

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

3.2.4 Metadata API

The *Metadata API*[5] [16] 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*[6] e *lookup*[7]. 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 [17].

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:

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:

```
{
    "info": {
        "num_results": 2,
        "limit": 100,
```

[5]This API was very recently deprecated (June, 2014) and was replaced by the *Web API*. It follows the same principles of the previous one, and so, for the purposes of this report, the differences are not relevant.

[6]https://developer.spotify.com/technologies/web-api/search

[7]https://developer.spotify.com/technologies/web-api/lookup

```
        "offset": 0,
        "query": "camane",
        "type": "artist",
        "page": 1
    },
    "artists": [
        {
            "href": "spotify:artist:3MLPFTe4BrpEV2eOVGOgLK",
            "name": "Camane",
            "popularity": "0.27"
        },
        {
            "href": "spotify:artist:5Gwulm1LfURW7dbZD1V3zX",
            "name": "Sergio Godinho/Camane/Carlos Do Carmo",
            "popularity": "0"
        }
    ]
}
```


Listing 3.2: Results ordered by "popularity"

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:3MLPFTe4BrpEV2eOVGOgLK&extras=album

The server responds with:


```
{
  "info": {
    "type": "artist"
  },
  "artist": {
    "href": "spotify:artist:3MLPFTe4BrpEV2eOVGOgLK",
    "name": "Camane",
    "albums": [{
        "album": {
          "artist-id": "spotify:artist:3MLPFTe4BrpEV2eOVGOgLK",
          "name": "O Melhor 1995 -2013",
          "artist": "Camane",
          "href": "spotify:album:1YksI1vvItkdH95NBfjTvX",
          "availability": {
            "territories": "AD AR AT AU BE BG BO BR CA CH CL CO CR
                CY CZ DE DK DO EC EE ES FI FR GB GR GT HK HN HR HU
                IE IS IT LI LT LU LV MC MT MX MY NI NL NO NZ PA PE
                PH PL PT PY RO SE SG SI SK SV TR TW UY"
          }
        },
        "info": {
          "type": "album"
        }
      }, {
        "album": {
          "artist-id": "spotify:artist:3MLPFTe4BrpEV2eOVGOgLK",
          "name": "Do Amor E Dos Dias",
          "artist": "Camane",
```

```
              "href": "spotify:album:0N4aahnnaGZfLRFSQaqqUe",
              "availability": {
                "territories": "AD AR AT AU BE BG BO BR CA CH CL CO CR
                     CY CZ DE DK DO EC EE ES FI FR GB GR GT HK HN HR HU
                     IE IS IT LI LT LU LV MC MT MX MY NI NL NO NZ PA PE
                     PH PL PT PY RO SE SG SI SK SV TR TW UY"
              }
            },
            "info": {
              "type": "album"
            }
          },

          {
              ...
          }

        ]
    }
}
```

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

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

3.2.5 iOS SDK (beta)

The iOS SDK supports iOS Application developers. Although still in beta [18], 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

3.3 Experiments

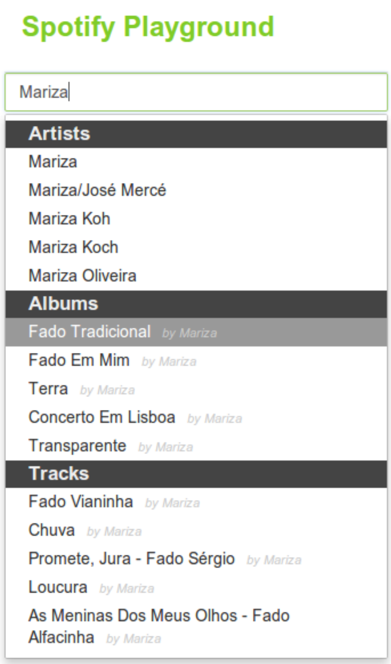
As a first hands-on experience with these tools, a single-page website was developed which allows the users to search and listen to music using Spotify's *Metadata API* and *Widgets*:

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

In Figure 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:



(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.

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

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

The final result can be seen in Figure 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 Figure 3.7 one can see the 100% supported canvas element.

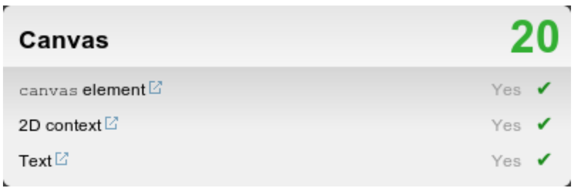


Figure 3.7: Test result for the canvas element.

3.4 Summary

Based on the analysis conducted, the most appropriate solution was to develop a Spotify Application. Although the other proposals were also feasible, 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 then implement the proposed modules 4 and 5:

4. Integrate the RAMA concept into a Spotify Application

5. Integrate RAMA's playlist generation into a Spotify Application

Chapter 4

Implementation and Validation

In this chapter, further details about the prototype's main features and the methodologies used when developing the prototype, as well as in the the validation process, will be explored.

The Implementation section will go into details about the processes used to have a maintainable development environment by taking advantage of several tools that automated most of the common tasks. The main implemented features will also be analysed. 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 Implementation

4.1.1 Technologies used

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

4.1.1.1 Git

The chosen version control system to manage the source code was Git[1].
Although not strictly followed, the model described in [19] was an inspiration. The use of branches [20] and tags [21] allowed for a manageable code development environment, e.g., marking release commits or working on experimental features.

4.1.1.2 Spotify Desktop Client

Spotify Applications are developed in its runtime environment - the Spotify Desktop Client. To open a Spotify Application, in the Desktop Client, one writes the following in the search bar: spotify:app:rama, where *rama* is the application identifier declared in the *manifest.json* file[2]. Example:

```
{
  "AppName": {
    "en": "RAMA"
  },
```

[1]Git: http://git-scm.com

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

```
  "BundleIdentifier": "rama",
  "AppDescription": {
    "en": "RAMA: Relational Artist MAps"
  },
  "AcceptedLinkTypes": [
    "playlist"
  ],
  "BundleType": "Application",
  "BundleVersion": "1.0",
  "DefaultTabs": [{
    "arguments": "index",
    "title": {
      "en": "Home"
    }
  }],
  "Dependencies": {
    "api": "1.10.2",
    "views": "1.18.1"
  },
  "SupportedDeviceClasses": ["Desktop"],
  "SupportedLanguages": [
    "en"
  ],
  "VendorIdentifier": "pt.inescporto"
}
```

Listing 4.1: manifest.json: *BundleIdentifier* is the application's identifier; *Dependencies* declares the Application's API dependencies.

There are useful options for development located in the "Develop" tab (Figure 4.1).

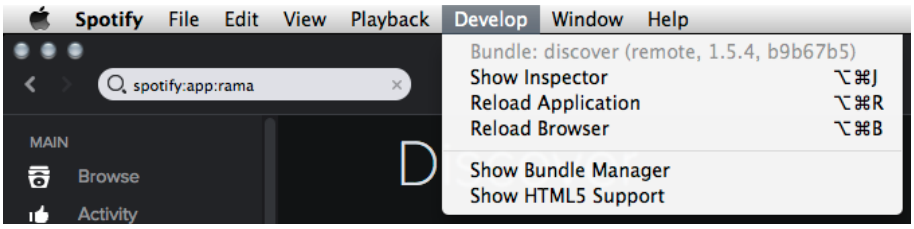


Figure 4.1: *Develop* Tab

The "Show Inspector" option opens the *Webkit Development Tools* (4.1.1.3) window.

4.1.1.3 Webkit Development Tools - webkit.org

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

Inspector Allows the inspection of the resulting Html and CSS and edit the code and see the application automatically reflect those changes (Figure 4.2).

Network Shows a timeline list of resources that where loaded from external sources (sometimes local) (Figure 4.3).

Profile Allows the identification of which parts of the javascript code are being executed frequently, and which ones might be creating a performance issue (Figure 4.4).

Audit Helps clarify which CSS rules are not being used (Figure 4.5).

Console Javascript interpreter that also works as the log output for the application (Figure 4.6).

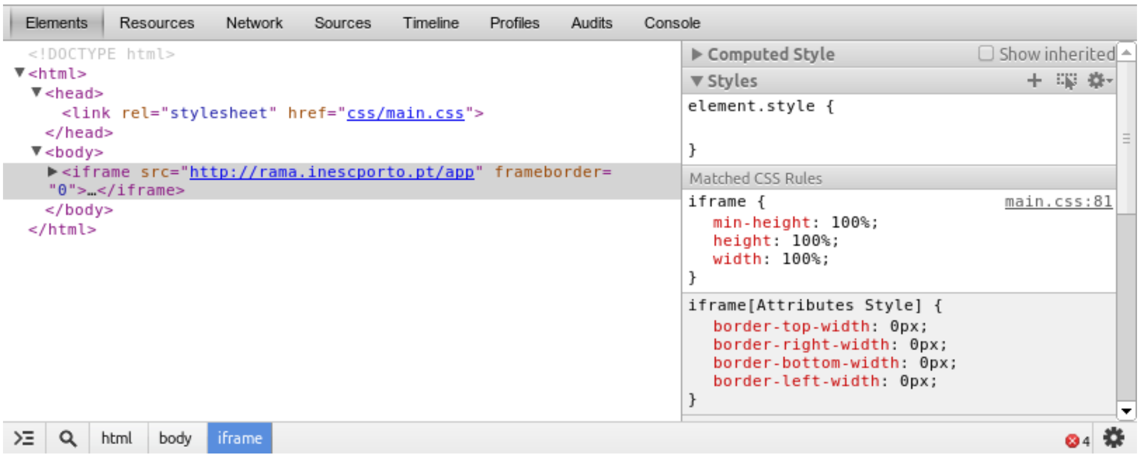


Figure 4.2: Webkit: *Inspector* tab view. Other tools available: *Resources, Network, Sources, Timeline, Profiles, Audits* and *Console*.

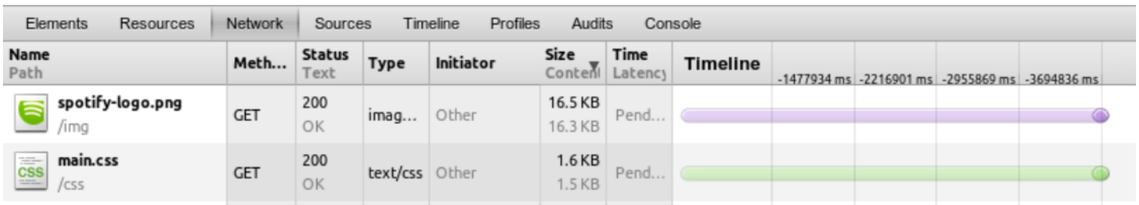


Figure 4.3: Webkit Network

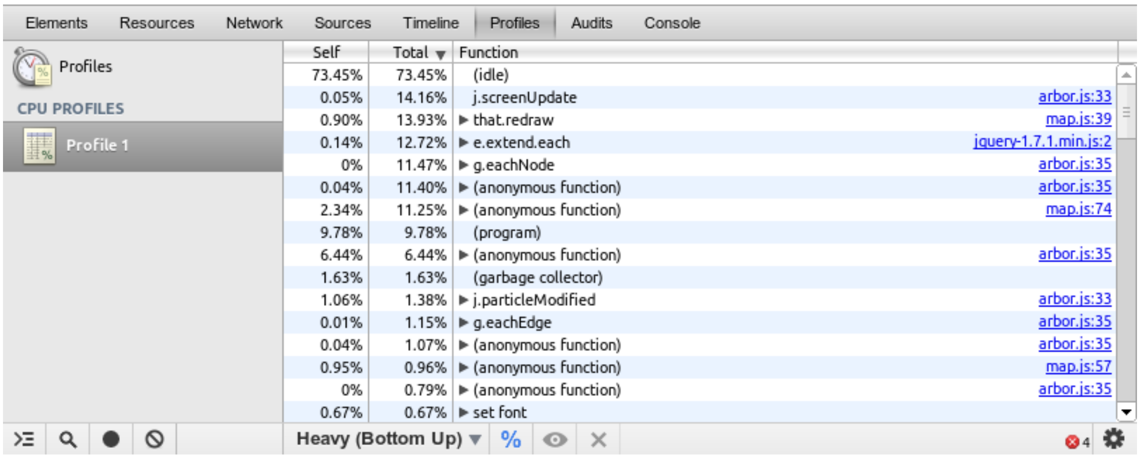


Figure 4.4: Webkit Profile: Canvas render functions are the ones taking up most of the processing cycles. However there is a JQuery function that used 12.75% of processing time, which might indicate a performance issue to be improved.

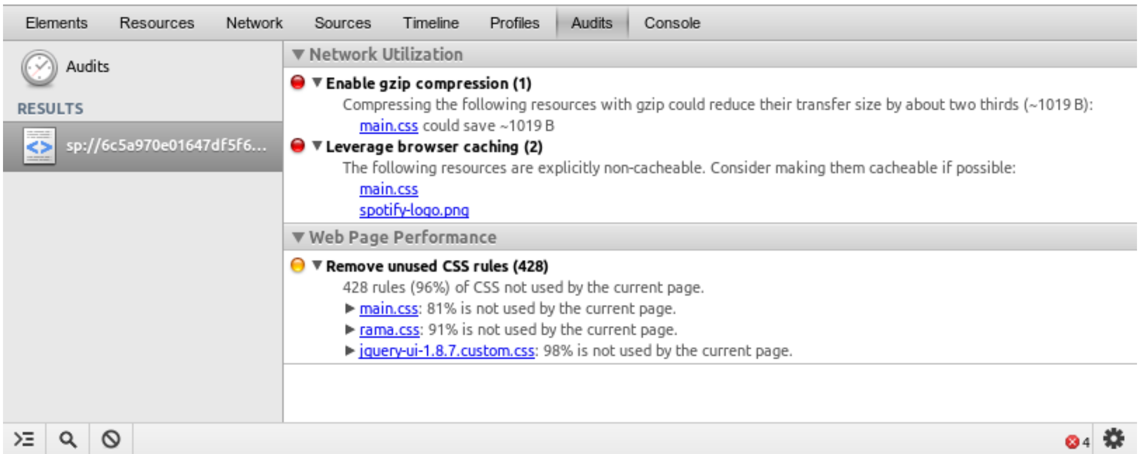


Figure 4.5: Webkit Audit: 96% of the CSS code is not being used indicates a issue to be solved.

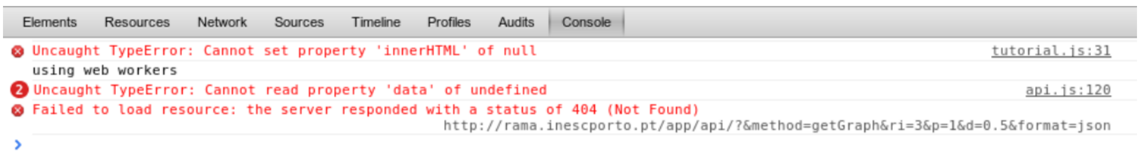


Figure 4.6: Webkit Console: Javascript errors are reported there (and highlighted in red as well).

4.1.1.4 Gruntjs - gruntjs.com

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

4.1.1.5 Npmjs - npmjs.org

Npmjs is the package dependency manager for nodejs[3] - Node Packaged Modules. Node packages will be used, since Gruntjs plugins are all nodejs packages (as well as Grunt itself). An npm configuration file (*package.json*) allows the identification of the packages that the application depends upon, as well as its versions. Example:

```
{
  "name": "RAMA",
  "devDependencies": {
    "grunt": "~0.4.2",
    "grunt-contrib-jshint": "*",
    "grunt-contrib-jasmine": "*",
    "grunt-contrib-watch": "*"
  },
  "version": "0.1.0"
}
```

Listing 4.2: *package.json*: "*" means that npm should always install the latest version of that package.

Note that these software packages are for development only. They are not needed at all when running production code.

4.1.1.6 Bower - bower.io

Bower is also a package manager, but oriented for web front-end packages. It mainly supports runtime software packages, which are needed for production code.

4.1.1.7 vis.js - visjs.org

Vis.js is a javascript framework for visualization. It provides a few visual components, including graphs.

4.1.2 Implemented Features

The requirements for the prototype are as following:

- Visualization of relations between artists by means of a visual tool;
- Editing of the visualization using several parameters;
- Editing of the graph by allowing to remove and add new nodes;
- Visualizing the tags/genres (that describe an artist) in the graph representation.

All of these features were implemented.

[3] nodejs: http://nodejs.org

4.1.2.1 Visulization of the Artists Map

The application automatically draws the map with the current playing artist as the main node, as seen in Figure 4.7.

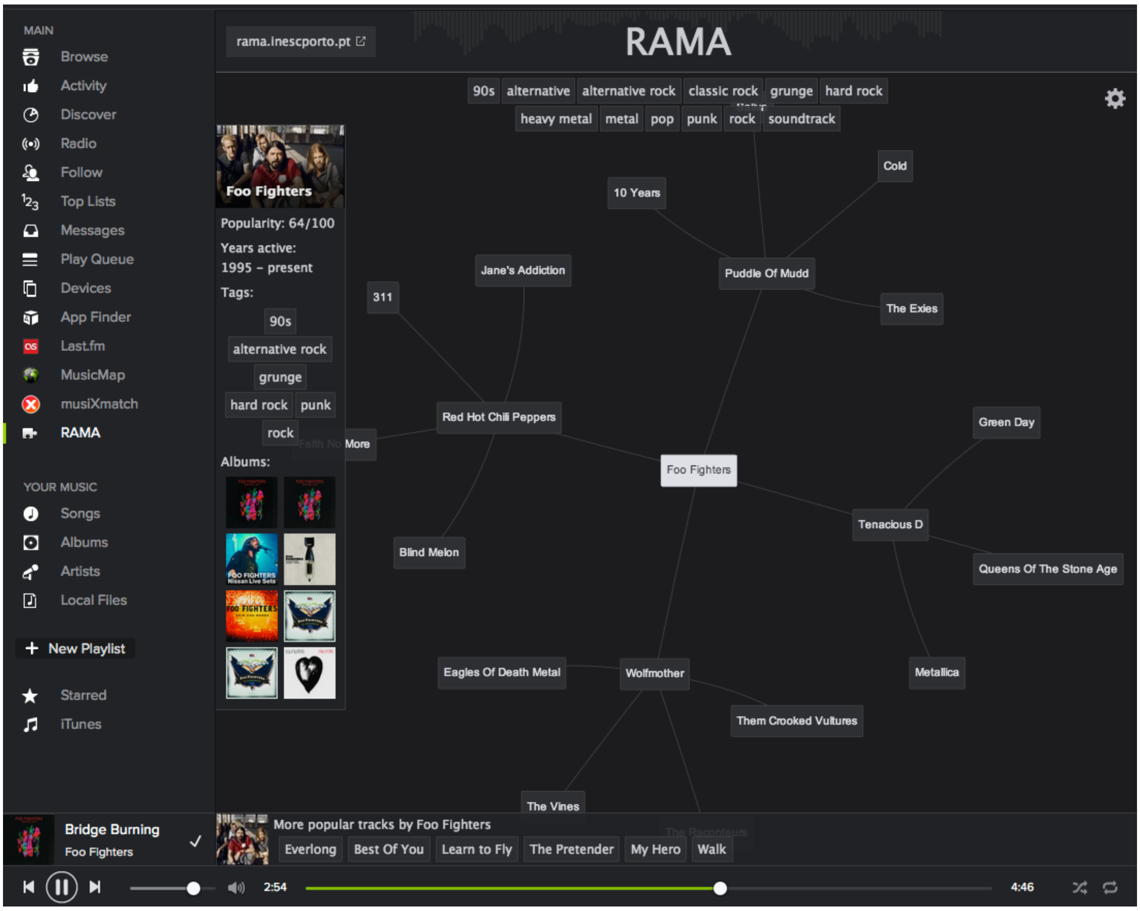


Figure 4.7: The first drawn graph uses the current playing artist (lower left corner) as the root node.

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[4], the algorithm stops. The graph creation algorithm is as follows:

```
function buildGraph() {
  // create a node with the root artist and insert it into the graph
  this.insertNode(this.rootArtist);

  // start constructing the graph recursively
  this.expandNode(
    this.depth - 1,
    this.rootArtist
  );
}

// Expands the node of the parent artist by this.branching.
```

[4] depth value of a graph

```
// It recursively decreases the depth parameter.
function expandNode(depth, parentArtist) {
  var node = this.getNode(parentArtist);

  // after expanding, the node will stop being a leaf
  node.isLeaf = false;

  // retrieve this.branching number of childs of the parent artist
  var relatedArtists = parentArtist.getRelatedArtists(this.branching);

  // for each child artist, insert and create a node into the graph
  // and do the recursive call for the child, but with decreased depth.
  for (var childArtist in relatedArtists) {
    this.insertNode(childArtist);

    // note that the stop condition of the recursion is depth <= 0
    if (depth > 0)
      expandNode(depth - 1, childArtist);
  }
}
```

Listing 4.3: Simplified graph creation algorithm in Javascript (duplicate nodes checking is encapsulated in the insertNode function)

The maximum number of nodes that the graph might have, can be calculated as follows:

$$\sum_{i=0}^{d} b^i \tag{4.1}$$

Where d is the depth value and b is the branching value of the graph.

This algorithm, albeit simplified, represents the basic flow when constructing a graph, or more specifically, a tree. Since, 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 4.8.

This approach, however, is not showing all of the available information. Given this same example (Figure 4.8), the artist node "Stone Temple Pilots" is a child of the node artist "Faith No More". The algorithm inserted the latter into the graph first. After that, when retrieving the childs of "Jane's Addiction", "Stone Temple Pilots" is contained in that list, and so the insertNode() function discards the node since it is a duplicate. This means that there is a connection between both of them that are 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 4.9.

From now on, if the graph is a tree, it will be said that the algorithm is using the Tree Mode. *So the example in Figure 4.8 is a* Tree Mode *example, and the examples in*

Figure 4.8: Graph created like a tree with "Red Hot Chilli Peppers" as the root node.

Figure 4.9 and Figure 4.10 are not Tree Mode *examples*[5].

Note (Figure 4.9) 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 current user's perception of the artist "Stone Temple Pilots" might be different from the previous example (Figure 4.8). 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 understanding of the artist's network by making sure the user knows that those specific artists are more connected between them, than any others. Since only three extra edges where added, this means 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

[5] 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.

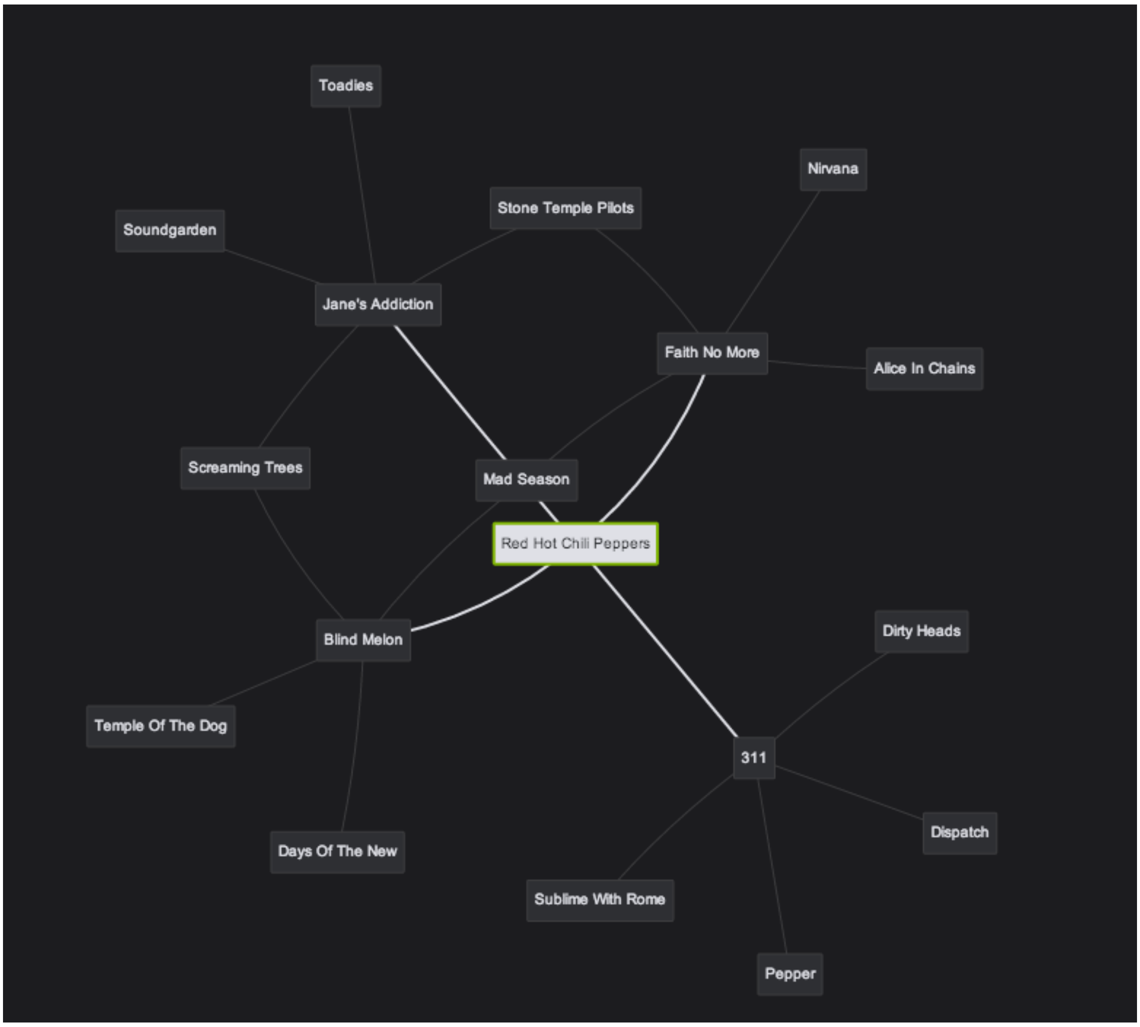


Figure 4.9: Graph created with all the connections with "Red Hot Chilli Peppers" as the root node.

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 4.10.

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 desirable. Instead of creating a visual map of the artists' network for the user to explore and change in his/her 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 more complete understand of the underlying 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:

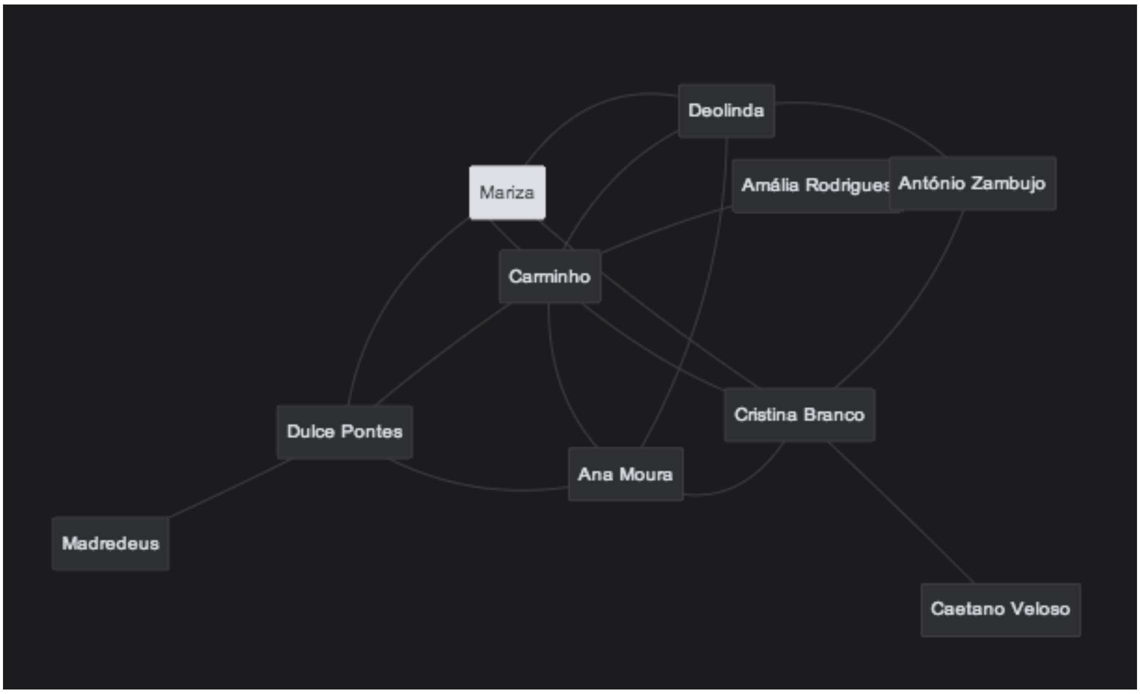


Figure 4.10: Graph created with all the connections with "Mariza" as the root node.

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 his/her needs.

Depth and Branching values
The depth and branching values have been mentioned before in this chapter, but not yet fully explained.

The **depth** value of a graph determines how deep the recursive algorithm is. The previously presented example (Figure 4.8) has a depth value of 2. The example in Figure 4.11 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 in Figure 4.8 has a branching value equal to 4.

4.1.2.2 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 4.12. When the user changes the parameters, the graph refreshes accordingly.

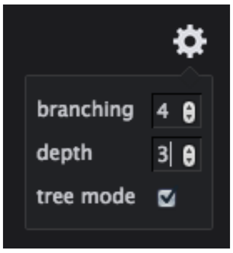


Figure 4.12: The settings menu that allows the user to change the visualization parameters.

4.1.2.3 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 (4.1.2.4).

Expand node
This option allows the user to expand a node further (ignoring the graph's branching value). Given the previous example in Figure 4.8, if the "Dispatch" node gets expanded, it results in Figure 4.13. 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. This is useful for when the user wants to construct the graph to his/her 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.

4.1.2.4 Artist Info

The user is able to see additional information about artists in the Artist Menu (Figure 4.14) such as its popularity value, albums and tags, as well as perform the *expand* and *new map* functions described in 4.1.2.3. When the user selects a node by clicking on it, the artist menu updates the displayed information.

The *popularity*, the *albums* and the *tags* are metadata information retrieved from Spotify's API framework (3.2.1), although Echonest's API[6] is also used as a fallback source for the *tags* (sometimes Spotify's API responds with a small number of results).

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 [17]. For example, to retrieve the terms[7] of an artist from Echonest's metadata API using the

[6] Echonest's API: http://developer.echonest.com/docs/v4

[7] Echonest calls it terms. From now on, terms and tags will be used interchangeably

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 (3.2.4), and this allows for retrieving extra information about the artist that Spotify's API does not provide.

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[8] explains [22] 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 Figure 4.14, the tags shown are very descriptive of the artist (grunge, 80s, hard rock).

4.1.2.5 Tags Overlay

The tags overlay menu (Figure 4.15) is meant to enhance the user's understanding on the displayed artists' nodes regarding their tags. These tags are the same ones used in the Artist Menu (Figure 4.14). They are selectable, and so, when clicked, the respective artist nodes that are described by those tags, are highlighted (as seen in Figure 4.15).

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 4.16, 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.

[8]Paul Lamere is Director of Developer Platform at the.echonest.com (http://the.echonest.com/company). He also blogs here: http://musicmachinery.com

Figure 4.11: Graph with depth value of 3

Figure 4.13: "Dispatch" artist node expanded.

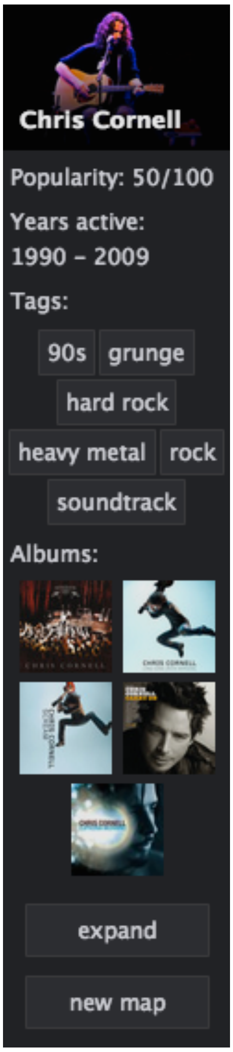


Figure 4.14: Artist Menu with information about "Chris Cornell"

Figure 4.15: Tags overlay for the displayed graph. When the tag "grunge" is selected the corresponding artist nodes are selected

Figure 4.16: Graph for "Anamanaguchi". The tags shown above are only but a small sample of all the tags of all the artists in the graph

4.1.3 Development Processes

The main objectives for the RAMA Spotify Application were to implement the basic functionalities that RAMA's website [23] 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

An overview of pushed commits can be viewed in Figure 4.17.

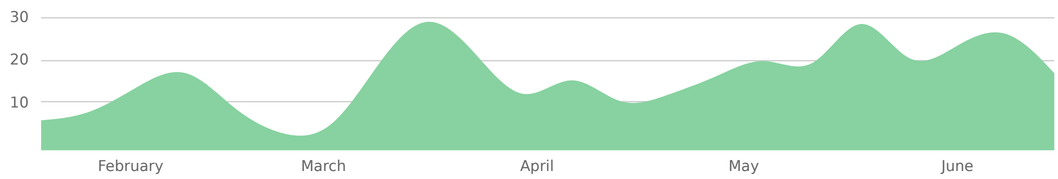


Figure 4.17: Graph showing the number of commits to the project from January to June, 2014. All of the development code was hosted in Github.com. Adapted from https://github.com/carsy/rama-spotify/graphs/contributors

Unit tests were added for each feature. The tests were run locally to ensure that previous features were still working during development. Also to ensure that every release had the tests passing, the project was integrated in Travis's Continuous Integration[9] system:
http://travis-ci.org/carsy/rama-spotify.
This way, after every push of a new release, the project gets built 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 alpha-testers (3 people) reviewed the state of the application. Their feedback would be carried over to the next release: bug fixes, improvements, etc. This was the testing cycle that allowed to validate the implemented features. To help with this iterative process the issue tracking system of Github[10] was used: https://github.com/carsy/rama-spotify/issues. Given this iterative process of implementation and feedback, the solution would be better prepared for future beta-testing.

[9] Travis: http://travis-ci.org
[10] Github: http://github.com

4.2 Validation

In order to validate the proposed method, user tests were done with regular Spotify users and non-Spotify users. They represent the second group of testers and will be referred as the beta-testers.

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 Desktop Client. 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 a few minutes, depending on the user. The next task was to open the RAMA application and do the same (again this should be completed in a short period of time). 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.

The objectives of the experiment were to assess some aspects of the application:

User Experience Was the user able to discover new music easily?

Interface's intuitiveness/design Were all of the application's components (buttons, controls, etc) easy to use? well placed?

After the test, the user filled a short questionnaire about the experiment, which contained the following questions:

1. Are you a Spotify user?
2. Did the RAMA application help you discover new interesting artists?
3. Do you think you would use the application to discover new music?
4. Was the application well integrated with Spotify?

Also in the end, the user was encouraged to leave suggestions/feedback or comments about the application.

4.2.2 Data Analysis

A total of 21 beta-testers participated in the experiment.

4.2.2.1 Question 1: Are you a Spotify user?

This question was meant to establish the user's experience with Spotify. The results can be seen in Table 4.1.

Answer	Nº of responses
Yes, I use it as my main way to listen to music	6
Yes, I use it from time to time	4
No, but I have used/tried it at least once	5
No, this is my first time using it	6

Table 4.1: Results for question 1.

This test group is relatively well distributed with 10 Spotify users, and 11 non-Spotify users.

This information is very important to better understand the user's suggestions that may or may not be related to basic Spotify interactions. For example, when a user wants to start a new map with an artist that is not at the moment visible, there two possibilities to do this. By searching for the new artist, the user can then drag and drop the artist into the application in the right side menu, or start playing a song from that artist, and then go back to RAMA and click the *new map* button in the artist menu.

Both ways require the user to search a new artist using Spotify's search bar. To regular Spotify users this is a recurrent procedure, but for new Spotify users, it might confuse them to go out of RAMA's application to fetch information.

4.2.2.2 Question 2: Did the RAMA application help you discover new interesting artists?

This was a yes or no question: 20 responded "yes", whereas, 1 responded "no". It is clear that RAMA's Spotify Application fulfilled the initial purpose of letting the user discover new music.

Most of the users did not discover new artists before using the *expand* button or the *new map* button. The users launched RAMA when listening to a known artist and this meant that, for the most cases, the user already knew most of the similar artists that appeared in the graph. Only when the user started to use these action buttons and found out how to progress in discovering new music, it started to get excited to see what new artists were coming.

4.2.2.3 Question 3: Do you think you would use the application to discover new music?

This is the principal question used to determine if users felt that RAMA was a usable way of discovering new music within Spotify.

Answer	**Nº of responses**
Yes, this is what Spotify is missing!	6
Yes, sometimes it's nice to see this visual representation	15
No, it does not add much to Spotify	0
No, I wouldn't use it. I like Spotify the way it is	0

Table 4.2: Results for question 3.

The results indicate that the visual representation is indeed a very useful feature. Although the majority of the users do not find it to be an essential feature that Spotify is missing, they still think it serves as a good helper to discover new music.

Given how different the listening habits of users can be, RAMA might be or might not be an appropriate tool for them: some users like to just have an automatic generated playlist playing in the background, while others might like to listen to an artist's albums in sequence.

4.2.2.4 Question 4: Was the application well integrated with Spotify?

In a scale of 1 to 5, given 1 to be "I felt the app was out of place" and 5 to be "I felt the app was part of Spotify", the users were asked to grade the application's integration into Spotify.

Answer	Nº of responses
1	0
2	1
3	1
4	6
5	13

Table 4.3: Results for question 4.

The majority of the users felt that the application was part of Spotify. These results indicate that the developed prototype blends very well with Spotify's Desktop Client. This is a very important indicator for evaluating the application's integration into Spotify's environment since it plays a major role in defining how good is RAMA in terms of user experience.

The user that classified with 2 points was not able to create a new map from an artist that was not visible in the graph although it was a regular Spotify user. This might indicate that the current way of performing this action is not very intuitive.

4.2.3 Summary

Overall, the results of the experiments were very positive and the users were very impressed with this new way of discovering new music within Spotify. Some functionalities still need to be polished in order to create an even more improved user experience, but RAMA's concept implemented within Spotify worked very well for the majority of the users.

Chapter 5

Conclusions

5.1 Summary

The proposed thesis focused on delivering an enhanced user experience when discovering new music in Spotify's Environment. By using RAMA's concept applied in the developed prototype, the user experience when discovering new music as been greatly increased. The users felt that RAMA's Spotify Application was natural and intuitive.

The amount of services that use visual tools for recommending music to users are not that many, although the ones shown here (2) are not representative of the whole spectrum.

Given the overview of the possibilities, creating a Spotify Application to apply RAMA's concept proved to be the best option to take.

The proposed modules 4 and 5 were implemented. The developed prototype proved to work after the beta-testing results. Although, there are a lot of improvements to do, the final result was very appealing to the users. All of the beta-testers liked the visual experience and the majority responded positively about using the application in a regular basis to discover new music.

All of the developed material (code, documentation, wiki, screenshots, demos) can be found in the project's code repository (`http://github.com/carsy/rama-spotify`) as well as in the appendix to this document.

5.2 Discussion

By introducing a visualization tool into a complete service like Spotify, the users felt that their experience with RAMA's application improved their abilities to find new music. The tests' results show that RAMA's Spotify Application is a successful approach to music discovery and recommendation. Although, the final results point in that direction, after the experiments, 3 beta-testers stated that their music listening habits are not focused on the music artists they are listening to. Instead, they simply pay attention to the songs (mostly, the popular ones), and so, their playlists are track-driven, not artist-driven. That might have had presented a problem to those users, since the focus of RAMA is the relations between the artists. However, Spotify's API's recommendation system proved to please those users, who started to pay more attention to the name of the artists they listen to.

Services like Spotify or Rdio, offer a complete set of features that range between playing every track on their catalogue, to saving albums for offline mobile listening. With such a vast music catalogue, the user might feel lost and not very motivated to find new music. Although these services continue to add features like Spotify's "Radio" [24], the user finds it hard to compute such a large world of available music.

By introducing this visual aid to music artist's relations, RAMA succeeds in letting the user understand and explore better the whole spectrum of available music.

5.3 Future Work

From the developed work so far, the need for a more segmented testing might have been called for, in order to improve the results obtained, as well to better understand the user's needs.

During the prototype's development and the beta-testing experiments, there were several additional features suggested for RAMA's Spotify Application. Most of them were not included in the prototype because of the limited time frame for development, while others seemed to stray from RAMA's focus. Nonetheless, every idea is important and might someday contribute for RAMA's future features.

Place the actions buttons (expand, new map and delete) near the artist node

Some users complained that it was hard to notice the actions buttons (one user did not even notice them). Maybe placing the buttons near the artist node and only show them while hovering it would make more sense.

Make it clear that the tags are clickable to the user

Some users noticed that it was not apparent at first that the upper tags were clickable, they thought it was just an extra information, not something they could interact with.

Allow to click-and-hold to preview listening to an artist

This Spotify feature could be applied to the nodes so that the user could browse the graph much faster.

Select multiple artist or tags

Allow the user to select multiple artists (or tags), in order to, for example, generate playlists from that selection.

Let the graph update automatically with the current playing artist

In order to keep RAMA more alive, the graph would refresh automatically when a new artist starts playing, with the new updated artist as root. Maybe an option to keep the graph "locked" would allow the user choose if he/she wants that behaviour or not.

Improve artists recommendation

The assumption that Spotify's recommendations are valid might not be so desirable. Although the results are satisfactory, without knowing the reasons behind those choices of connections, the artists recommendations become inflexible. To improve this, one could fully use Echonest's API to fetch the similar artists [25]. The amount of query personalization is much more attractive than Spotify's.

Improve the connection between the artists and the tags

As an improvement of the previous idea, the connections between the artists and the overlaying tags could become more related, and so, the intersection of tags between artists could be used to improve the recommendations (creation of connections).

Include an alternate view with tags as nodes, instead of artists

Some users suggested that the tags overlay was an interesting point of view, and so, if the graph could be drawn using the tags instead, the resulting map could be an alternative view of the application.

Cluster groups of artists

Cluster groups of artists to create a geographical representation of what the entire network of available artists could be like.

Tag mind map

In a way to try and complement the two previous ideas, a clustering of the tags would create an interesting mind map of the current genres. And since there are indeed music genres that are similar (like metal and celtic metal) the map would be browsable.

Information

Add an information UI component to help the users understand better the visualization parameters (a tutorial-like component) - although those parameters are advanced settings (not strictly necessary for the user to interact with) it might be a good idea to give more power to the user and make these settings easy to understand.

References

[1] 8tracks' Product Vision. http://8tracks.com/about. [Online; accessed 20-June-2014].

[2] Spotify's About page. https://www.spotify.com/uk/about-us/contact/. [Online; accessed 20-June-2014].

[3] Rdio's Product Vision. http://www.rdio.com/about/. [Online; accessed 20-June-2014].

[4] Pandora's music recommendation system. http://www.pandora.com/about. [Online; accessed 20-June-2014].

[5] P. Lamere. Creating transparent, steerable recommendations. 2008. URL: http://www.mirlab.org/conference_papers/International_Conference/ISMIR%202008/latebreak/lamere.pdf.

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

[7] L Sarmento and 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.

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

[9] Fabien Gouyon, Nuno Cruz, and Luis Sarmento. A Last.fm and YouTube Mash-up for Music Browsing and Playlist Edition. 2011. URL: http://ismir2011.ismir.net/latebreaking/LB-3.pdf.

[10] Libspotify SDK. https://developer.spotify.com/technologies/libspotify. [Online; accessed 20-June-2014].

[11] Spotify Apps. https://developer.spotify.com/technologies/apps. [Online; accessed 20-June-2014].

[12] The Chromium Embedded Framework (CEF). https://code.google.com/p/chromiumembedded. [Online; accessed 20-June-2014].

[13] Spotify Widgets. https://developer.spotify.com/technologies/widgets. [Online; accessed 20-June-2014].

[14] Spotify Subscriptions Comparison Chart. http://community.spotify.com/t5/Newcomers-and-Contribution/Subscription-Comparison-Chart/td-p/666916. [Online; accessed 20-June-2014].

[15] Libspotify SDK's builds available. https://developer.spotify.com/technologies/libspotify/#libspotify-downloads. [Online; accessed 20-June-2014].

[16] Metadata API (deprecated, now Web API). https://developer.spotify.com/technologies/web-api. [Online; accessed 20-June-2014].

[17] Project Rosetta Stone. http://developer.echonest.com/docs/v4#project-rosetta-stone. [Online; accessed 20-June-2014].

[18] iOS SDK (beta). https://developer.spotify.com/technologies/spotify-ios-sdk. [Online; accessed 20-June-2014].

[19] Vincent Driessen. A successfull Git branching model. http://nvie.com/posts/a-successful-git-branching-model/. [Online; accessed 20-June-2014].

[20] Git Brancing: What a Branch Is. http://git-scm.com/book/en/Git-Branching-What-a-Branch-Is. [Online; accessed 20-June-2014].

[21] Git Basics: Tagging. http://git-scm.com/book/en/Git-Basics-Tagging. [Online; accessed 20-June-2014].

[22] Echonest's terms: What is the difference between weight and frequency. http://developer.echonest.com/forums/thread/353. [Online; accessed 20-June-2014].

[23] RAMA's original Website application. http://rama.inescporto.pt/app. [Online; accessed 20-June-2014].

[24] Spotify's Radio mode. http://news.spotify.com/us/2011/12/09/discover-the-new-spotify-radio. [Online; accessed 20-June-2014].

[25] Similar artists query using Echonest's API. http://developer.echonest.com/docs/v4/artist.html#similar. [Online; accessed 20-June-2014].

Appendix A

Appendix

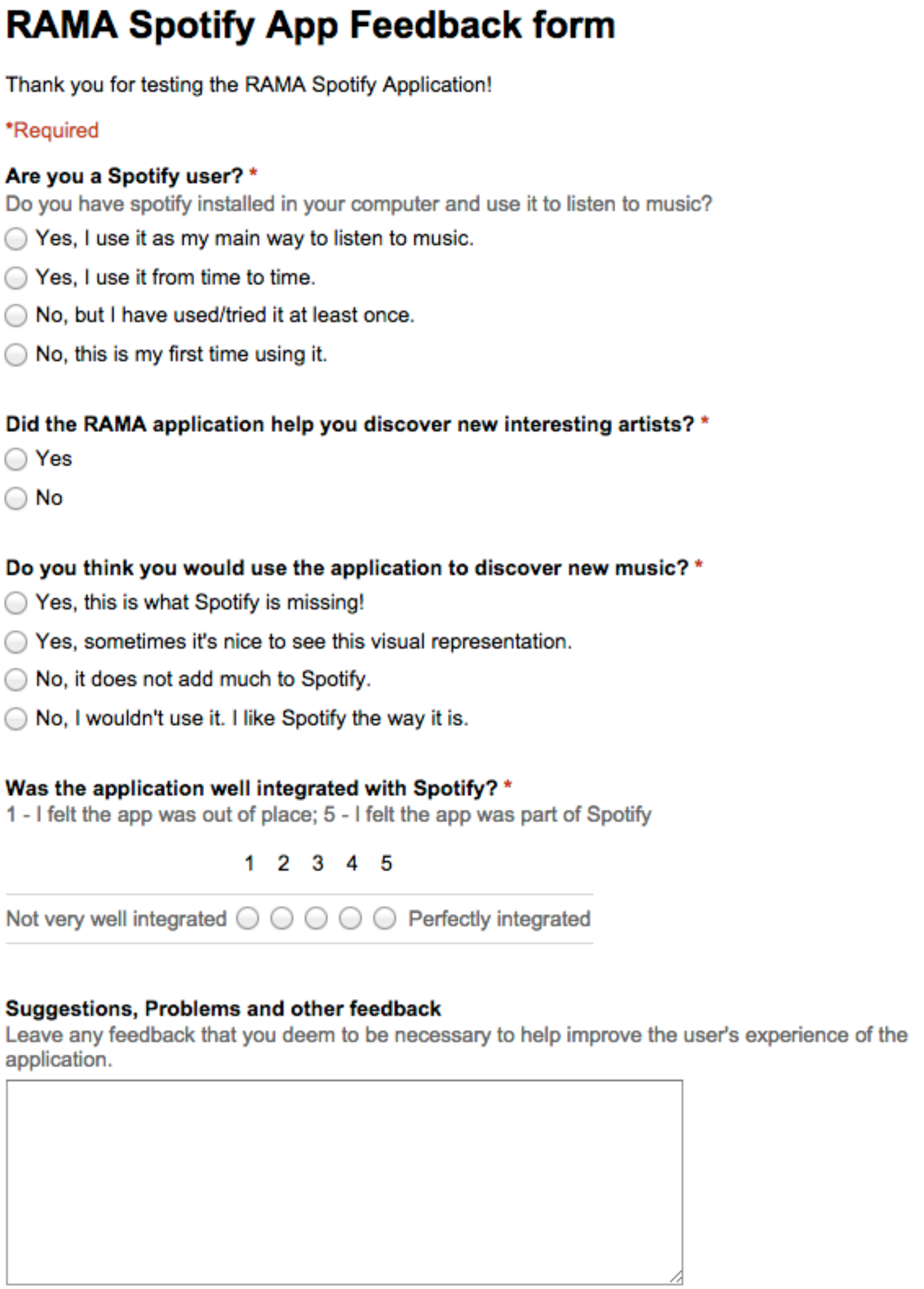
A.1 Demonstrations

Two demonstrations of the developed prototype can be seen in:
https://raw.githubusercontent.com/carsy/rama-spotify/master/demo.gif
and
https://raw.githubusercontent.com/carsy/rama-spotify/master/demo2.gif

An effort was made to show the main implemented features.

A.2 Questionnaire

The questionnaire in Figure A.1 was composed of four mandatory questions and a free-text field for suggestions and other feedback.



RAMA Spotify App Feedback form

Thank you for testing the RAMA Spotify Application!

*Required

Are you a Spotify user? *
Do you have spotify installed in your computer and use it to listen to music?

- Yes, I use it as my main way to listen to music.
- Yes, I use it from time to time.
- No, but I have used/tried it at least once.
- No, this is my first time using it.

Did the RAMA application help you discover new interesting artists? *

- Yes
- No

Do you think you would use the application to discover new music? *

- Yes, this is what Spotify is missing!
- Yes, sometimes it's nice to see this visual representation.
- No, it does not add much to Spotify.
- No, I wouldn't use it. I like Spotify the way it is.

Was the application well integrated with Spotify? *
1 - I felt the app was out of place; 5 - I felt the app was part of Spotify

	1	2	3	4	5	
Not very well integrated	○	○	○	○	○	Perfectly integrated

Suggestions, Problems and other feedback
Leave any feedback that you deem to be necessary to help improve the user's experience of the application.

Figure A.1: Questionnaire that the beta-testers were required to fill after the experiment.