CS301 Software Engineering
Project

Software Requirements Specification (SRS) Document
For
Music Recommendation System

.

v1.0
15/09/2016
Saumya Gupta
NIIT University

Table of Contents

Revision History

Name	**Date**	**Reason For Changes**	**Version**
Final_Updation	17-09-16	Document accuracy development	v1.0

1. Introduction

This software requirement specification (SRS) report expresses complete description of our recommendation project. This document includes all the functions and specifications with their explanations to solve related problems as a CS301 course project of a team at NIIT University studying CSE.

1.1 Purpose

We are working on some common failings in most of the current music playing *System*s, like

- Insufficient or incorrect *Metadata* of one's offline music *Library* files and
- Lack of availability of recommendations of similar music by the player itself, with recommendations not only from the offline but the online music *Library* as well.

We are trying to understand and improve the current solutions on these problems and maintain the accuracy of the recommendations. The main issues would be wrong or sparse data sizes. Wrong data will lead to wrong calculations and ultimately wrong recommendations and sparse data will lead to no actions i.e. no recommendations by the player because of no data available to process. To make the picture clearer, if we are listening to some particular song, we are in a particular

mood, we would obviously want suggestions of the songs or music similar to what we are listening. Therefore we will be working on a music player that will not only recommend the *user* music similar to what he/she is listening to but also update his/her offline music *Library* by updating the *Metadata* of individual music files. We will deal with all the particular problems that come in way.

1.1 Document Conventions

No particular standards or typographical conventions as such in this document. It has some common conventions though like

- TBD - To be done i.e. not yet completed and left for later.
- Italics has been used for terminologies.
- Bold has been used for emphasising on specific features or data with special significance.
- No unique abbreviations used in this document.

1.2 Intended Audience and Reading Suggestions

- This SRS document intended for all the developers involved in the making of this music recommendation software i.e. all the members of the ProjectRecommendTeam, including the team documentation writer. This is for the unity in the ideas of the development team.
- One obvious intended reader is the course in charge of CS301 who is expected to read this carefully and deliver helpful comments.
- It is also for the possible future readers such as marketing staff etc.

This SRS contains the detailed description of what our music recommendation *System* is expected to do, starting with the overall description, followed by the external interface requirements, *System* features and other non-functional requirements.

1.3 Product Scope

The goal of the product is to access the offline music *Library* of the *user*, and while the *user* is listening to a particular song, this software will access the **MusicBrainz** music *Database* and give suggestions based on similarities of the *Metadata* of that file with the file, selected from the external *Database*. Also the product is supposed to check and update the *Metadata* of all the *user* offline music files for whenever the first time that file is played. The *user* can update the data without playing the music file too. In this way the *user* will have a fully updated music *Library*, **MusicBrainz** being the most extensively used and reliable music *Metadata Database* available, also the *user* will enjoy suggestions of similar music and then can do whatever he wants with that suggestion.

In recent years, great amounts of work have been done in music perception, psychology and music's effects on human behaviour. Undoubtedly, music always has been an important role in people's life and people have greater access to it. All of these highlight that music recommender is an effective tool for saving our time to find music for our personal interests.

1.4 References

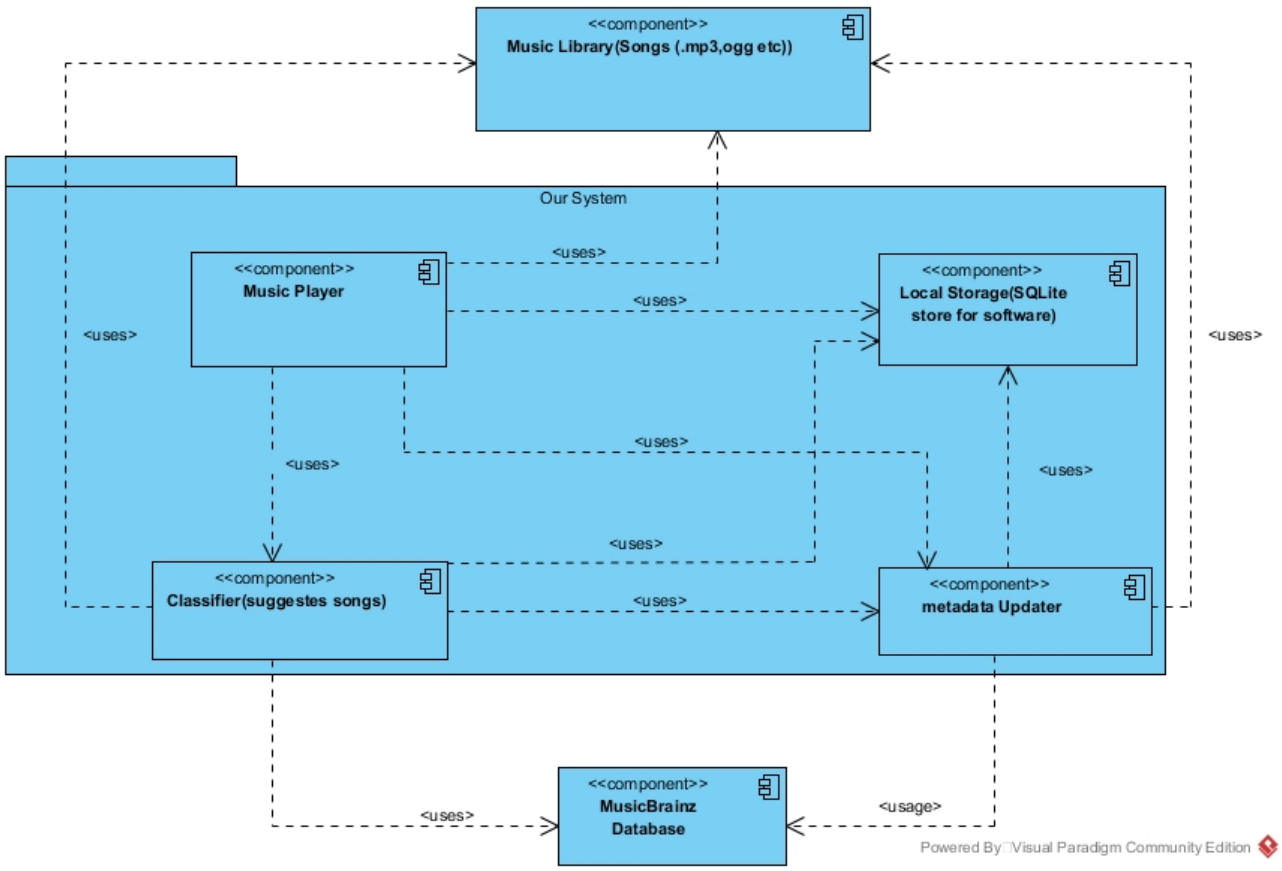
This document is written with Google docs, downloaded as a .docx file and finally converted to .md file so as to be submitted to GitHub. The basic template or scope document was provided by the course in charge.

- IEEE Std 830-1998 - IEEE Recommended Practice for Software Requirements Specifications. IEEE Computer Society, 1998.

2. Overall Description

2.1 Product Perspective

- Our recommendation *System* depends on data coming from the *user*'s offline music *Library* file and the **MusicBrainz** song *Metadata Database*. This program has different type of *user*s, so there will be functionality differences between *user*s that will occur with respect to item data. Our recommendation *System* should work efficiently according to music data. So, there exists a *user* interface that is suitable for music recommendations and our *System* will be working in background. Once this *System* finds an accurate result, it will be shown on the interface.
- In terms of software, our recommendation *System* will run on personal computers i.e. Desktop as for now and later we can extend it to smart phones etc. That is, it will run on any device with internet connection. Moreover, it will be implemented making use of *Database* **SQLite**. This brief information of interfaces and components is explained in more detail in the below diagram.



2.2 Product Functions

This product is basically a music player software that will run on Desktop as for now. The product will act as a very basic music player and have features like play/pause music, but in addition to these, it will also provide *user*s with recommendations. To be very clear, when the *user* will be listening to a particular song, the player will display suggestions of music related to what *user* is listening, based on its *Metadata*, i.e. *Tags* like, genre, singer, rating etc. For suggesting it will use the *user*'s offline *Database* as well as the **MusicBrainz** *Database*. Also it will update *Metadata* of *user*'s offline music files. Functions in detail are given below.

- Manage songs (adding/removing songs to/from the application).
- Control music (play/pause/stop/seek music, go to previous/next track, increase /decrease/mute music volume).
- Manually update *Metadata*.
- Manually recommend music.
- Edit fields in the song info

2.3 *User* Classes and Characteristics

- A music player is one software which has the maximum number of types of *user*s. Every *user*, be him a non-working aged guy or a busy, working middle-aged guy, or a school boy, is at least an average music listener who would once in a while also like to get suggestions on similar music, based on the mood.
- As far as music lovers who want to listen to new music and experiment with their choices are concerned, this product is a gem for them.
- Also the *user*s who struggle with updating the *Metadata* of their music or face embarrassment while listening to great music with cheap *Metadata*, especially cover art, since they have downloaded it from a pirated site, will have a solution for them now.

2.4 Operating Environment

As mentioned above, that, as for now, the software will work on Desktop. Any operating *System* with hardware support for playing sound is capable of running this software on it.

	GNU/Linux	Windows
Minimum CPU clock speed	2GHz	1GHz
Minimum RAM space	1 GB	1GB(32-bit)/ 2GB(64-bit)
Minimum hard disk space	4GB	16 GB free
Graphics	X Window server etc.	Microsoft DirectX 9 graphics device with WDDM driver

2.5 Design and Implementation Constraints

The implementation constraints can be encountered while choosing a suitable method for the filtering purpose. This is important as this will result in either wrong or accurate recommendations for the *user*, and based on this, the product usability is decided. So the 3 possible methods for filtering purpose are

- **Content based filtering**

1. Grouping into views based on already available *Metadata*.
2. *User* behaviour not analysed, may lead to less accurate results.
3. Requires a standard *Metadata* schema, here, provided by **MusicBrainz**.

- **Collaborative filtering**

1. Generates more accurate result, requires *user* data,
2. We don’t have *user* data available with us, so we are not implementing this.

- **Waveform analysis**

1. Grouping into genres based on the track’s waveform.

2. It's found inaccurate and still under research & development, so we are not implementing this.

2.6 *User* Documentation

In the *user* documentations we will provide 2 things

- *User* manual: Detailed description of the functionalities of the player, i.e. what all the *user* can do and steps for how they can do it.
- 'Help' icon in the menu bar: Like other software, clicking on this option will redirect the *user* to the help manual page on the website.

2.7 Assumptions and Dependencies

Dependencies include various open source material available online and components from other projects which we integrated with ours. Following are the open source material

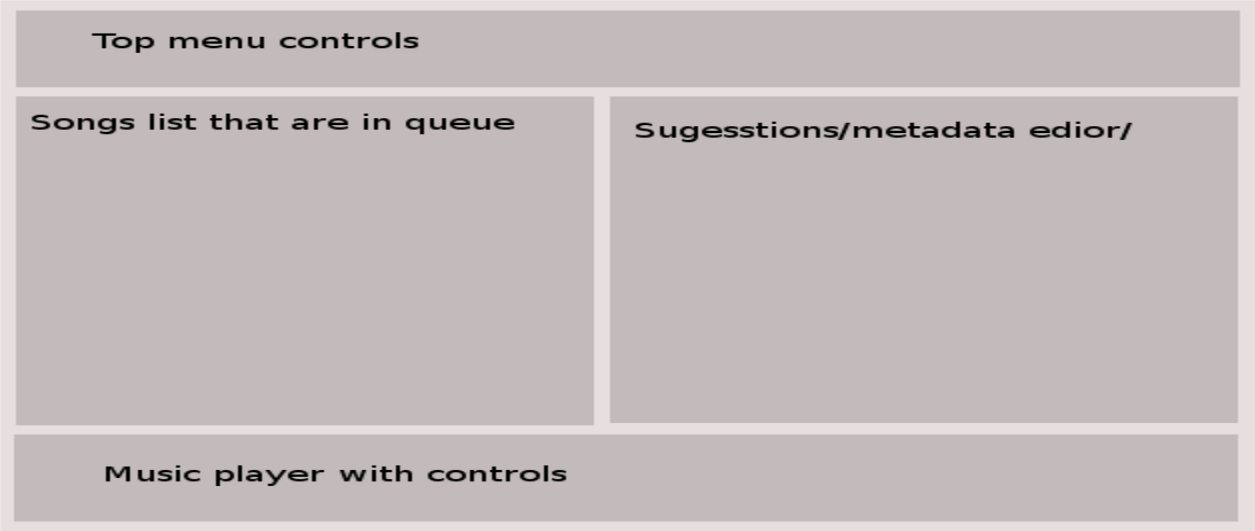
• **MusicBrainz**	For the music *Database* schema
• **MusicBrainz Picard**	For tagging music
• mutagen	For handling *Metadata* of audio file
• cherrymusic	Music player
• **PyQT**	Python cross platform GUI toolkit
• Sci-kit learn	For the Classifier

Our recommendation *System* will work accurately i.e. recommend correct tracks, if the *Metadata* is correct. So we are assuming that **MusicBrainz** *Database* is correctly updated and also when the *user* edits the data of the file manually, he does it correctly.

3. External Interface Requirements

3.1 *User* Interfaces

PyQT, which is a Python cross-platform GUI toolkit, is being used for implementing the *user* interface. The following is a basic UI mock-up of what the screen layout will look like once the product is completed.



The above wireframe suggest that there will be four frames on the screen

- Top frame will consist of menu control options like ‘Help’, ‘Add/Remove’ music, etc.
- In the middle, we will have two frames, the left one will show the songs queued, and the right one will show suggestions of the song playing, or suggestion for a particular selected file.
- At the bottom frame we will have various options for controlling music, like play/pause/seek/next/previous.

3.2 Hardware Interfaces

- The *user* needs an input device to interact with the *System*.
- The software needs an output device to interact with the *user*.
- Software need a playback device for the sound output.
- A working **NIC** is needed for internet connectivity while giving suggestions and using *Metadata* updater.

3.3 Software Interfaces

Declared TBD by the whole team, due to insufficient information at present.
<Describe the connections between this product and other specific software components (name and version), including *Database*s, operating *System*s, tools, libraries, and integrated commercial components. Identify the data items or messages coming into the *System* and going out and describe the purpose of each. Describe the services needed and the nature of communications.

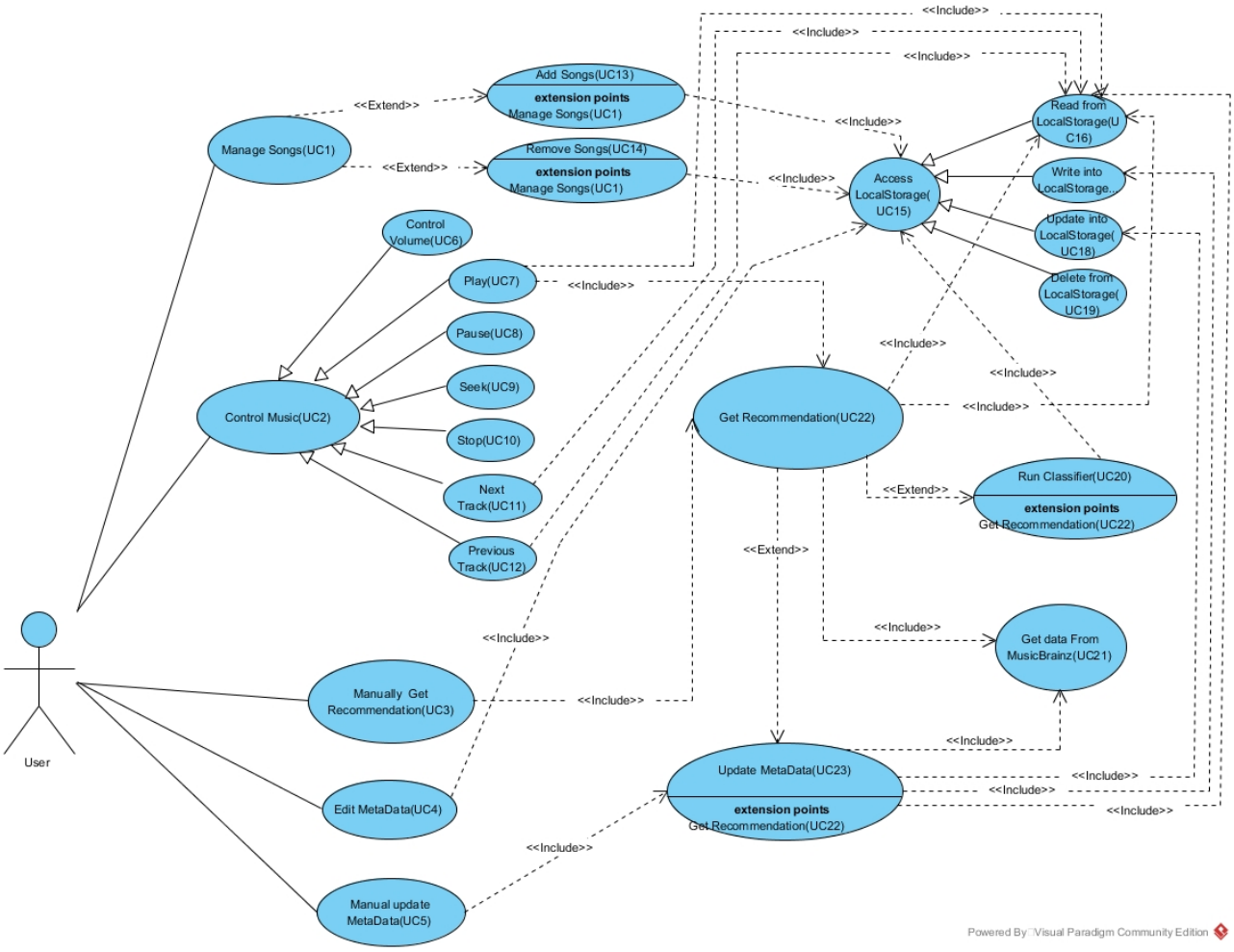
Refer to documents that describe detailed application programming interface protocols. Identify data that will be shared across software components. If the data sharing mechanism must be implemented in a specific way (for example, use of a global data area in a multitasking operating *System*), specify this as an implementation constraint.>

3.4 Communications Interfaces

- **Internet connection requirement**
  1. When *Metadata* updater accesses the **MusicBrainz** *Database*.
  2. When Classifier communicates with **MusicBrainz** *Database*.
- **Network communications**
  1. Will use HTTPS/TLS.
  2. Will be encrypted.

4. *System* Features

The *System* feature will be more elaborative after this use case diagram for the software.



Our *Music Recommendation System* comes with the following set of *System* component features.

4.1 Music Player

It is the main component of the *System* that can play music with a number of controls including play, pause, stop, play next or previous track, and seek and volume control. It has high priority, w/o this, the basic requirement of playing song can't be fulfilled.

- The *user* will be able to import music from his offline music collection, which will put that music file in the player queue.
- *User* can remove a track from imported music, which will make it vanish from the player queue.
- Rest *user* operations on music control will work as expected.

4.2 Local Store

Local Store is a resolve layer of the application, which will be implemented as the **SQLite** *Database* and will contain the cache data of tracks. It will Store all the information required for the software to work, showing high priority. This info will be used by other component features internally and it will not be directly available for *user* interaction as such.

- On adding a track or importing a track from the offline music *Library* of the *user* to the application the track will be added into the local Store.
- The Local Store will be storing the file path of the track, whether the track has been correctly tagged or not, the **MusicBrainz** track id and the id of the track.
- On removing the track from the application the track will also be removed from the Local Store.

4.3 Music Metadata Updater

This component feature will update the *Metadata*/*Tags* of music optionally from the **MusicBrainz** *Database*. Although *System* will work w/o this component but it fulfils a lot of functional requirements, hence it is a feature of high priority.

- Work in 2 different situations:
    a. Firstly, if the *user* plays a track, Classifier gets triggered and if *Metadata* for that track is not updated, this component will be triggered by Classifier.
    b. Secondly, if the *user* manually triggers *Metadata* updating.

4.4 Classifier

It is the core of the *System* and has high significance as it fulfils the basic purpose of the project, i.e. recommendations. In order to fulfil the basic functionality of recommendations, we need this.

- Work in 2 different situations:
    a. Firstly, when the *user* chooses to play the music, the currently playing track will automatically be used for getting suggestions, play process will trigger the Classifier.
    b. Secondly, the *user* can get suggestions on a track that he/she is not playing manually. In that case also the Classifier will be triggered.

4.5 MusicBrainz Database

The **MusicBrainz** *Database* is the external *Database* which is primary *Database* for collection of correct music *Metadata*. It is the most reliable source, since it contains almost every single track, with the correct *Metadata* info.

- This *Database* is used during updating *Metadata* and getting data for classifier to generate suggestions, which proves that though we can build a player that will not have an external music *Database* but, it can’t ever be used for recommendations.

***User* Actions**

Our Music Recommendation *System* comes with the following set of use case specifications/functional requirements/*user* actions i.e. what the *user* will be able to do with the application.

Actor - *User*

***User* Requirements**	**Includes use case(s)/Functional Requirements**	**Description**
Manage songs R_01	add music UC_13/**RQ_11**	The *user* will be able to import music from his external music collection, to the application
	remove music UC_14/**RQ_12**	The *user* will be able to remove any track from the playlist
Control music R_02	play music UC_07/**RQ_05**	The *user* will be able to play the track by selecting it or clicking on Play
	seek track UC_09/**RQ_07**	The *user* will be able to move anywhere in the timeline of the track
	pause music UC_08/**RQ_06**	The *user* will be able to pause the track being able to play it again from the same timeline
	stop music UC_10/**RQ_08**	The *user* will be able to stop the track which will close the track, in order for the *user* to play another track or exit software
	go to the next track UC_11/**RQ_09**	The *user* will be able to play the next track
	go to the previous track UC_12/**RQ_10**	The *user* will be able to play the previous track
	volume control UC_06/**RQ_04**	The *user* will be able to increase or decrease or mute the volume of the playing track
Manually update *Metadata* R_03	Manually update *Metadata* UC_05/**RQ_03**	The *user* will be able to update the *Metadata* of any track manually, i.e., simply by right clicking
Manually recommend music R_04	Manually recommend music UC_03/**RQ_01**	The *user* will be able to get recommendations of any track manually, i.e., simply by right clicking
Edit fields in the song info R_05	Edit fields in the song info UC_04/**RQ_02**	The *user* will be able to edit info of any track manually, i.e., simply by right clicking

***User* Interactions**

Functional Requirement: Manage songs
User interaction:

- Import tracks from offline music collection.
- Add directories, subdirectories and files to the music player.
- The *user* can remove the track from the application.
- The local Store updates itself deleting the track.

Features involved: Music Player, Local Store

Functional Requirement: Control music
User interaction:

- Click events trigger all controlling of music operations.
- On playing a track, the play process triggers an event that gets recommendations for the track.
- The *System* checks whether the *Metadata* is already updated or not, if not then the *Metadata* is updated in the track.
- The *System* connects to the **MusicBrainz** *Database* to update the track *Metadata*.
- The *System* fetches *Tags* of the track.
- After updating *Metadata*, the local Store is updated for the current track.
- The *System* then runs Classifier again on the track and gets all suggestions for that track.
- The suggestions are updated into the local Store.

Features involved: Music Player, Local Store, Music Metadata Updater, Classifier and **MusicBrainz** *Database*

Functional Requirement: Manually get recommendation
User interaction:

- The *user* triggers event of getting recommendation of a track. The *System* checks whether the *Metadata* is already updated or not, if not then the *Metadata* is updated in the track.
- The *System* connects to the **MusicBrainz** *Database* to update the track *Metadata*.
- The *System* fetches *Tags* of the track.
- After updating *Metadata*, the local Store is updated for the selecting track.
- The *System* then runs Classifier on the track and gets all suggestions for that track.
- The suggestions are updated into the local Store.

Features involved: Local Store, Music Metadata Updater, Classifier and **MusicBrainz** *Database*

Functional Requirement: Edit *Metadata*
User interaction:

- The *user* defined *Metadata* of a track is going to be updated in the local Store.
- Here, the *user* is going to type the data new data himself.

Features involved: Local Store

Functional Requirement: Manually update *Metadata*
User interaction:

- The *user* triggers an event of manually updating the *Metadata* of a track.
- The *System* connects to the **MusicBrainz** *Database*.
- The *System* fetches the *Metadata* of a track from the *Database*.
- The *System* then updates the *Metadata* into the local Store.

Features involved: Local Store, Music Metadata Updater, and **MusicBrainz** *Database*

5. Other Non-functional Requirements

The non-functional requirements of the *System* are explained below.

Non-Functional Requirements	Name	Description
5.1 Performance Requirements		
NR_01	Quickness	*System* should be fast enough to play music and respond to any of the *user* action in any way without any shattering or buffering, else it will be not be a good experience.
NR_02	Robustness	*System* should be robust to deal and act accordingly with common error scenarios like no internet connection, unavailable *Metadata*, unsupported file types.
NR_03	Failure Handling	In case of failures it should be able to fail or recover quickly.
5.2 Safety Requirements		
NR_04	Exception Handling	The software should be able to restrict or warn (in the first place) the *user* from doing things not suitable, like, increasing volume beyond threshold, or exiting the software w/o saving the changed data.
5.3 Security Requirements		
NR_05	Encrypted Connection	Connection between *user* and **MusicBrainz** servers should be Encrypted (HTTPS/TLS).

5.4 Software Quality Attributes

- It should be designed to be easily adopted by a *System*, so that it is usable.
- It should interfere with the working of other important software.
- The *System* should have accurate results and should respond correctly to the *user*'s changing habits.
- There is no way it can harm the operating system.

5.5 Business Rules

- This software is an Open Source software.
- Unless required by applicable law or agreed to in writing, software distributed is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

6. Other Requirements

TBD

Appendix

- System Requirement Document source : *Wikipedia*