DA3

Brunel_m, Feraud_g, Xie_h, Strzel_a,
Meilhva_v, Rose_s, Pasqui_c, Defran_h

Revision table:

Date	Author	Section	Comment
23/12/12	AudioWire	All	DA1
22/03/13	AudioWire	All	DA2
19/05/2013	rose_s, brunel_m, defran_h, xie_h	All	Redesign parts and correcting the mobile part
22/05/2013	pasqui_c	All	Finalization of the DA3 translation
29/05/2013	feraud_g	All	Correction of general spelling
31/05/2013	Defran_h	All	Review all spelling, updated some parts and review presentation
12/11/2013	Rose_s	Desktop Player	Updated AudioWire player definition and diagrams

Table of contents

1. About AudioWire

A. Reminder of the EIP

Epitech is an Information Technology school created in 1999, following the spirit of Epita. It holds passionate students that wish to learn computing through practical exercises instead of theory. It's a 5 years course that supplies an IT degree, recognized as a level one diploma by the CNCP.

The strength of this school is mainly based on the fact that students are brought to learn the concepts by themselves. Notions are acquired by working in teams on a lot of small projects and on some more ambitious ones like the PFA and finally the EIP.

Contrary to the regular teaching methods, where knowledge is shared through theory, Epitech students are faced to problems without the knowledge to solve them, in order to let them create there own solutions. This method of teaching gives the students the ability to adapt to any kind of situation and teach them how to work in teams.

B. Background and scope of the project

Nowadays, a lot of very powerful music players exist but they usually all have more or less the same functionalities. Because our goal is innovation, our goal is to create something new and not to compete with other music players. Therefore, we decided to create AudioWire, a music player with an artificial intelligence (AI) capable of suggesting playlists depending on the user's behavior. Indeed, AudioWire will be able to suggest music according to the tastes, habits and mood of the user. A social network platform will also be developed to allow users to share their music with their friends, or check their friends' profile.

Of course, AudioWire will work on all the modern platforms: Windows, Mac OS, and Linux. Considering that there are more than one billion smartphone users nowadays, we have also decided to be present on the most popular mobile operating systems: Android and iOS. Finally, we will also create a website where users will be able to manage their account, talk with friends, and control the player on their desktop thanks to the remote functionality, which will be the main functionality of the web interface. Finally, it will of course also be possible to download the desktop client or the mobile client from the website.

C. Definitions, Acronyms and Abbreviations

[1]**API**: "Application Programming Interface", it is an interface provided by computer programs to interact with external devices.

[2]**REST**: "Representational State Transfer", it is an architectural style allowing data exchange with a structured data system.

[3]**JSON**: "JavaScript Object Notation", it is a textual data format for representing data in generic and structured way.

[4]**AI**: "Artificial Intelligence", it is a program or algorithm whose purpose is to adapt its behaviors trying to act as a human being would.

[5]**MusicBrainz**: Music database, collaborative, universal, freely distributable, referencing recordings of works, not works in themselves disseminated. Official website: http://musicbrainz.org

[6]**Titanium**: Construction free software kit-tools, allowing developers to create a native application for iOS and Android with a common code. Official website: http://www.appcelerator.com/platform/titanium-sdk/

2. Project overview

A. The server

Every client of the project (Android, IPhone, PC/MAC applications and website) will communicate with the website through a REST[2] API[1]. The API[1] will be in charge of getting the information from the main database, manage the users' authentication and the social features.

The API[1] will use the protocol HTTPS (HTTP + SSL) to ensure security as well as flexibility when using. Indeed, a large range of the new languages (such as Ruby, Python, Java, etc.) has implemented this protocol natively, giving the opportunity to do secure and fast requests easily. The API[1] will run thanks to a Ruby on Rails server; furthermore, it will be possible to use most of the modern database to store the data. However, we recommend the use of MySQL.

Finally, by using a REST[2] API[1], other developers will be able to create their own AudioWire client. A fully detailed documentation listing all the functionalities of the API[1] will be provided.

User:

URL	HTTP Method	Parameters	Usage
/users/sign-in/	POST	email: String, password: String	Sign in / create a token
/users/sign-out/	POST	id: Integer (Token)	Sign out / destroy a token
/users/id/password/	PUT	id: Integer, oldPassword: String, newPassword: String	Update the password
/users/sign-up/	POST	email: String, password: String	Sign up
/users/id/	PATCH	id: integer, Fields to be updated	Update user's information
/users/list/	GET	id: integer	Return signed up users list

Friend list:

URL	HTTP Method	Parameters	Usage
/friendlist/	POST	name: String	Create a friend list
/friendlist/	GET	None	Return friend list
/friendlist/id/	GET	None	Return information on friend list.
/friendlist/id/	DELETE	None	Delete the friend list
/friendlist/id/friend/	POST	friend: id	Add a friend to the list
/friendlist/id/friend/	DELETE	friend: id	Delete a friend from the list

Music:

URL	HTTP Method	Parameters	Usage
/songs/	GET	None	Return the playlists
/songs/	POST	band: String, title: String, track: String	Send information about a soundtrack
/songs/id/	GET	id: Integer	Get information about a soundtrack
/songs/id/	PATCH	band: String and/or title: String and/or track: String	Update partially a sound track
/songs/id/	DELETE	None	Remove a soundtrack from the user's playlist

Image:

URL	HTTP Method	Parameters	Usage
/images/	POST	File	Send an image
/images/id/	GET	id: Integer	Get information about an image
/images/id/	DELETE	id: Integer	Delete an image

Message:

URL	HTTP Method	Parameters	Usage
/messages/	GET	None	List all the messages a user received or sent
/messages/id/	GET	None	Return a message
/messages/id/	PUT	Content: String	Update a message
/messages/id/	DELETE	None	Delete a message
/messages/sent/	GET	None	List all the sent messages
/messages/received/	GET	None	List all received messages
/messages/	POST	User: ID, content: String	Send a message to another user
/messages/<user>	GET	Id user to add in the URL	List all the messages sent and received from the user passed in parameter

In order to authenticate the users, every request (except those to create an account and to connect) must contain the token within the GET parameter "auth_token".
The requests will have this format: "<request>/?auth_token=<token>".

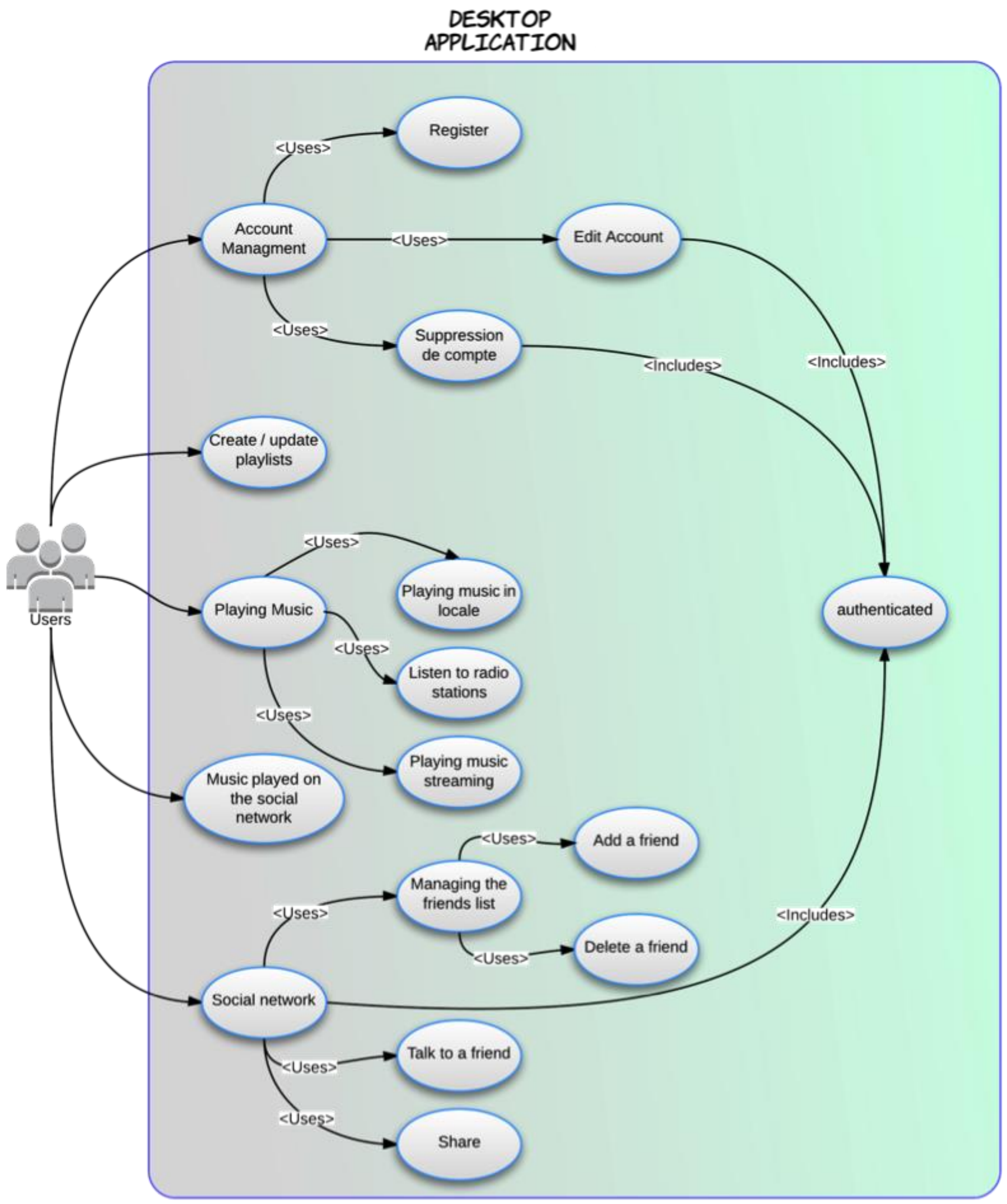
Every request will follow the HTTP response standard. In addition, the response might contain some additional data encoded in JSON.
The following error codes will be used:
200 => Ok
201 => Created
204 => No Content
400 => Bad Request
401 => Unauthorized
403 => Forbidden
500 => Internal Server Error

B. The Client

a. The desktop application

The desktop application will be the main application of AudioWire; it will contain all the main functionalities of the project (apart from some functionalities that are specific to smartphones or the website). This client will be written in C++ because it is a programming language mastered by all members of the team, and most of all, it is a low level object oriented language that allows us to optimize the audio player's performances.

In order to minimize any incompatibility risks between the AI[4] and the desktop application, the artificial intelligence will also be written in C++. All the basic functionalities of an audio player will be implemented, such as a library management system and playlists playback. We will also implement some social features such as contact management, playlists and library sharing between friends. The user will also be able to show his mood and his thoughts; that will allow the AI[4] to retrieve information about the user. Finally, the user's contacts will be able to see what kind of music he is listening to according to his behavior. You can see below a use case diagram that represents the desktop application:



Desktop application use case diagram

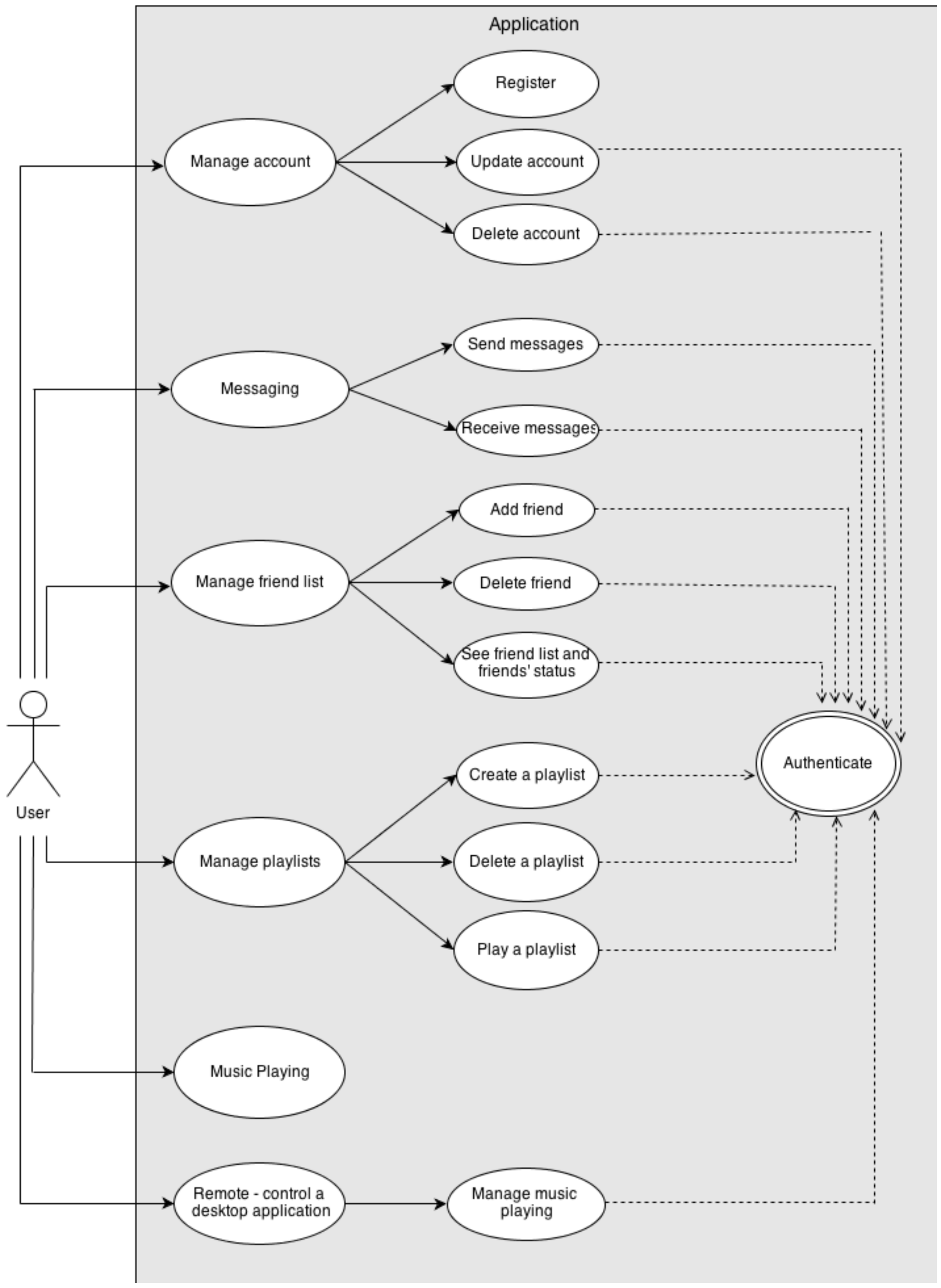
b. Mobile Application

We have decided to focus our mobile development on the two most popular mobile operating systems: iOS and Android. Due to the limited performance of smartphones compared to desktop computers, some features of the desktop application will not be available in the mobile applications. The users will be able to manage their playlists and contacts list via their profile page, change their personal information and, of course, they will also be able to listen to local music.

All features that require an excessive amount of resources will not be implemented. For example, some AI[4] algorithms will not be present. It will also be able to synchronize playlists between the desktop and mobile application.

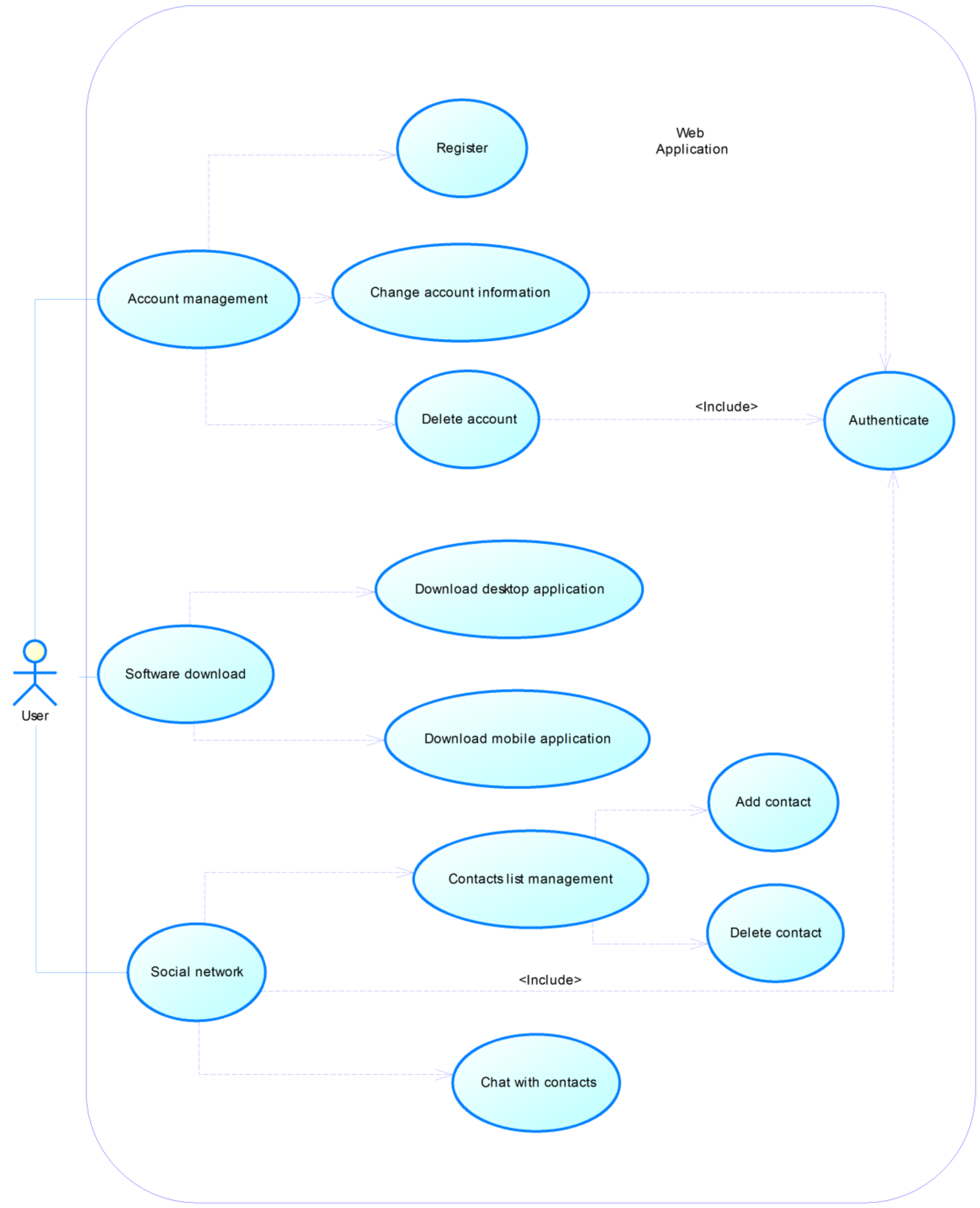
We will use an Objective-C framework called AVFoundation for the iOS application. This framework was made by Apple and is essentially used for the treatment of multimedia files. This development kit was made with iOS 4 but was improved a lot for iOS 5 and 6.

Mobile Application

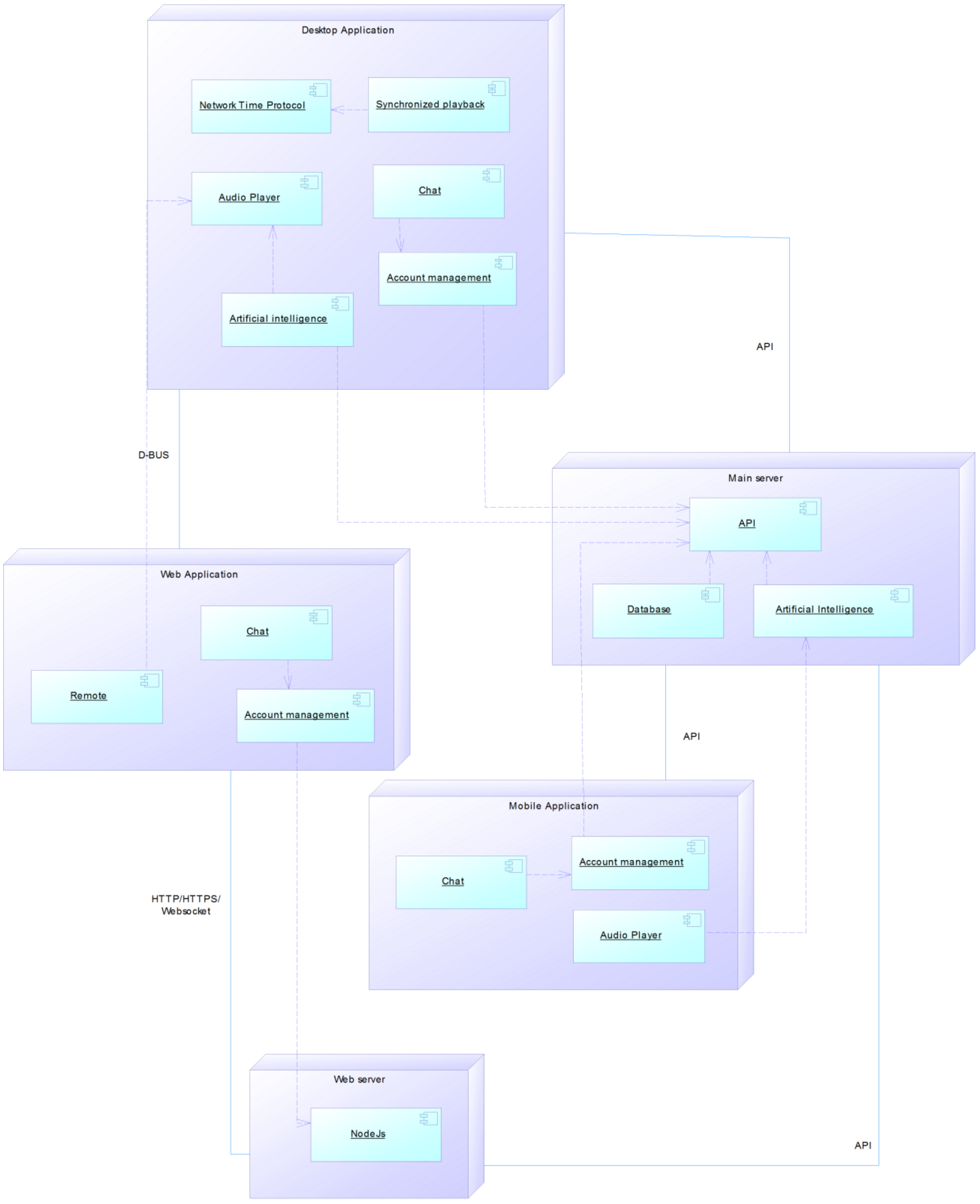


Mobile applications use cases diagram

c. Website



Web application's use cases diagram


Desktop Application
Network Time Protocol
Synchronized playback
Audio Player
Chat
Account management
Artificial intelligence
API
D-BUS
Main server
API
Database
Artificial Intelligence
Web Application
Chat
Remote
Account management
API
Mobile Application
Account management
Chat
Audio Player
HTTP/HTTPS/
Websocket
Web server
NodeJs
API

Physical diagram of AudioWire

3. Audio player

The music player will be available on all known platforms such as Mac, Linux and Windows. It will contain not only all the basic features from a classic player but also features like artificial intelligence, musical synchronizing and the social part.

The role of the client is important because apart the technical aspect, it will attract the user. Therefore, the software must be as visually and functionally adapted as possible in order to gather a lot of people.

The technical choices

For the development of the client, we decided to use c++ language for two reasons. The first one is simply because we all studied this language during our years of study and then it seemed logical to choose this one. The second one is that c++ is a very powerful object-oriented language allowing us to optimize a lot our software especially in term of artificial intelligence.

For the environment of development we decided to use QT platform, an object-oriented API that is used with c++. The pros of this API is that we can develop multiplatform graphical interfaces very quickly in cpp and the 5.1 version of QT implement a lot of features in term of media player.

The strategic choices

Strategic choices have been made in the GUI. That is to say, we oriented the development of the interface in the way that people can understand and use it easily. This is why we wanted to draw out different ideas into existing software such as Spotify and Deezer. Indeed these two large platforms of online music are very famous in virtual music world and because of the limited time we have to develop our project, it seems easier to learn from these two programs instead of redevelop another concept.

You should also know that the main target of AudioWire is mainly people who often listening music on the web or on their smartphones and this is why it's important to us to develop our software in the same way, in order to minimize the adaption period from one software to another.

4. AI[4]

A. Purpose of the artificial intelligence

The Artificial intelligence in AudioWire aims to select music for the user. This music will of course reflect his tastes and moods.

B. Tools

In order to have as much data as possible for each song, we will use the two tools called Aubio and Vamp. They can extract a large amount of data such as tempo, sound frequencies, notes, audio spectrum, etc. for a given audio file.

This information will allow us to rank the music of the user and to identify what other song the user may like.

C. Algorithm description

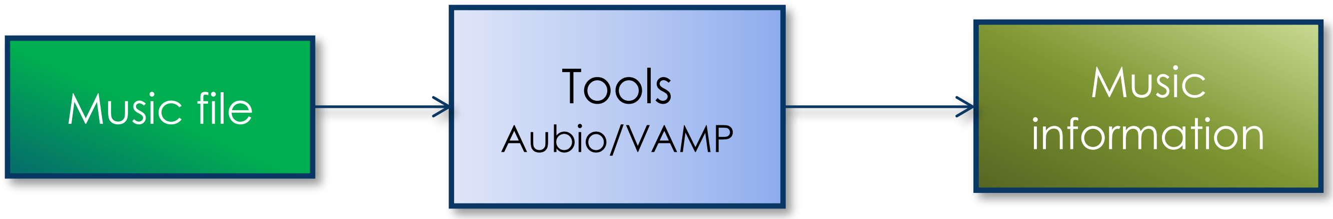
a. Inputs

Data obtained thanks to the musical styles and the file:

- Type of music
- Length
- Meta-data (artist, album, title, etc.)

Data obtained thanks to the AUBIO/VAMP tools:

- Beat / tempo
- Sound frequency (Hz / MIDI)
- Harmonical changes
- Musical segments/blocs similar



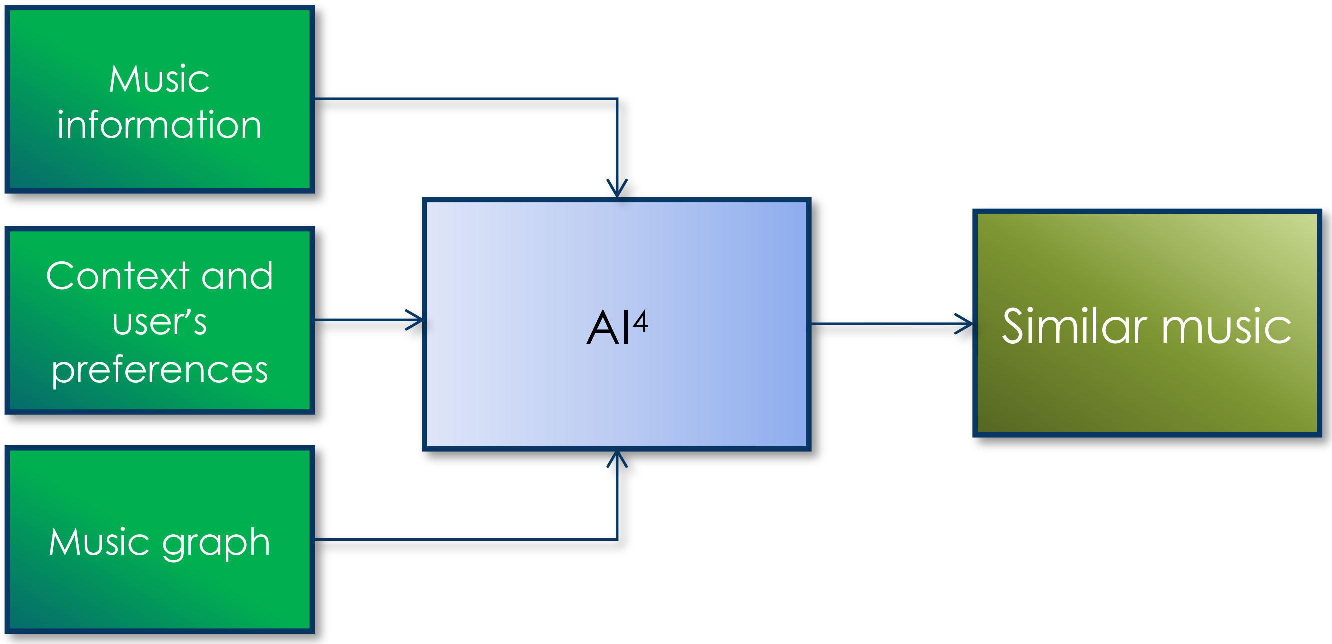
- Different audio spectrum
- Etc.

Data obtained thanks to the algorithm:

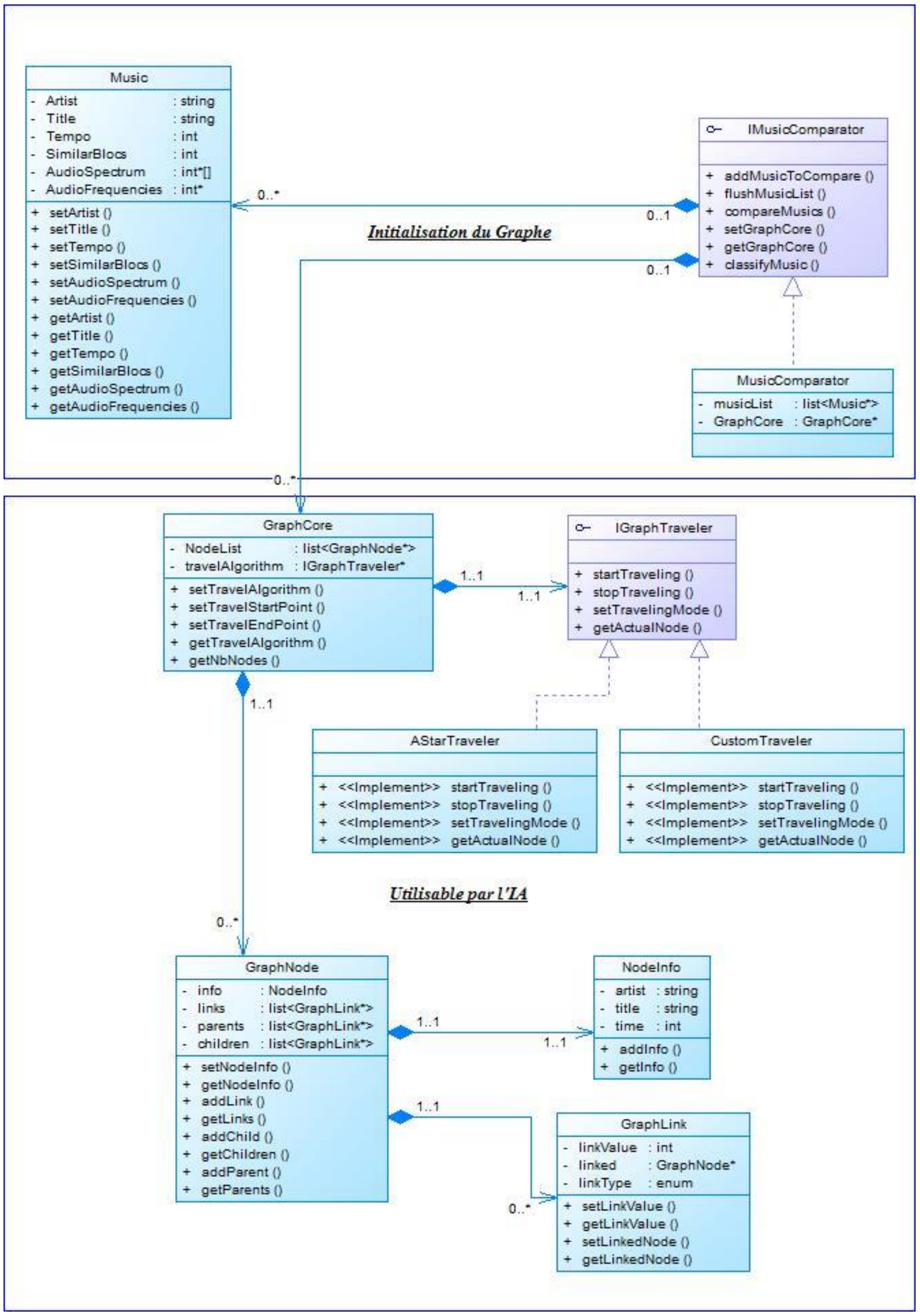
- User's mood
- Context (relaxing, partying, cleaning, etc.)
- Like of the listened songs (thanks to 2 buttons: "Like" "Dislike).

b. Operation

The first step that the AI⁴ will have to take will be to group songs by type. To do so, meta-data, musical genre, but also the extracted information such as the tempo or the sound frequencies will be used. In order to find the songs matching the user's tastes, the algorithm will show off songs that have the attributes (tempo, frequency…) that the user seems to like. Moreover, the user will be able to create contextual playlists, where he will add a few songs and let the AI⁴ looks for matching songs.



Here is the class diagram showing how the songs may be linked by similarities:



5. Synchronization

This part will explain how AudioWire allow users to play a song on several devices at the same time and in a same area, or to listen a song with a friend at the same time over a network.

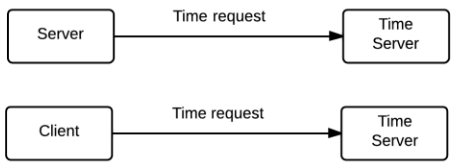
The audio synchronization, on a local network and on the Internet, will be implemented thanks to the Network Time Protocol (NTP). This protocol allows the synchronization of clocks with an online server.

By synchronizing the clients' clocks with the server, it will be possible to launch audio files at the same time easily, while keeping a latency of less than 12ms. Indeed, after doing some research, we realized that the human ear could only spot a lag between two different audio sources only if the delay is more than 12ms. To implement the synchronization functionality, we chose to use the NTP as it helps synchronizing different devices' clock with a latency of no more than 10ms.

To illustrate this, here is a diagram showing off the steps to realize before launching the music on several devices:

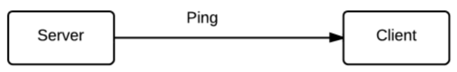
How to Synchronize Audio Playback

#1



The server and client(s) clocks are synced with time servers so that server and client(s) have the same time base.
This is done using the Network Time Protocol, which allows clock synchronization over the internet.

#2



Server pings client five times to get multiple latency times.

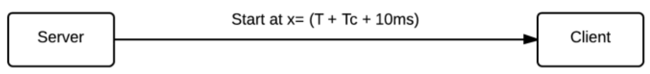
#3

L / 2 + 10ms = Tc

Where L = Highest latency
Tc = Time to client

Server Calculates the time a command will take to be executed based on highest latency time.
L is divided by two because the ping command is round-trip which means that it takes half the time to send an instruction.

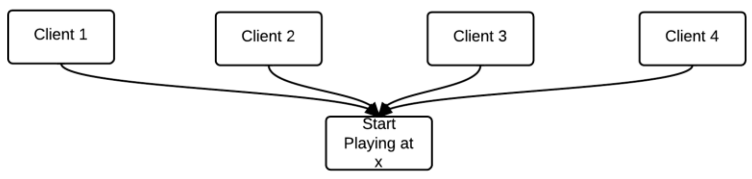
#4



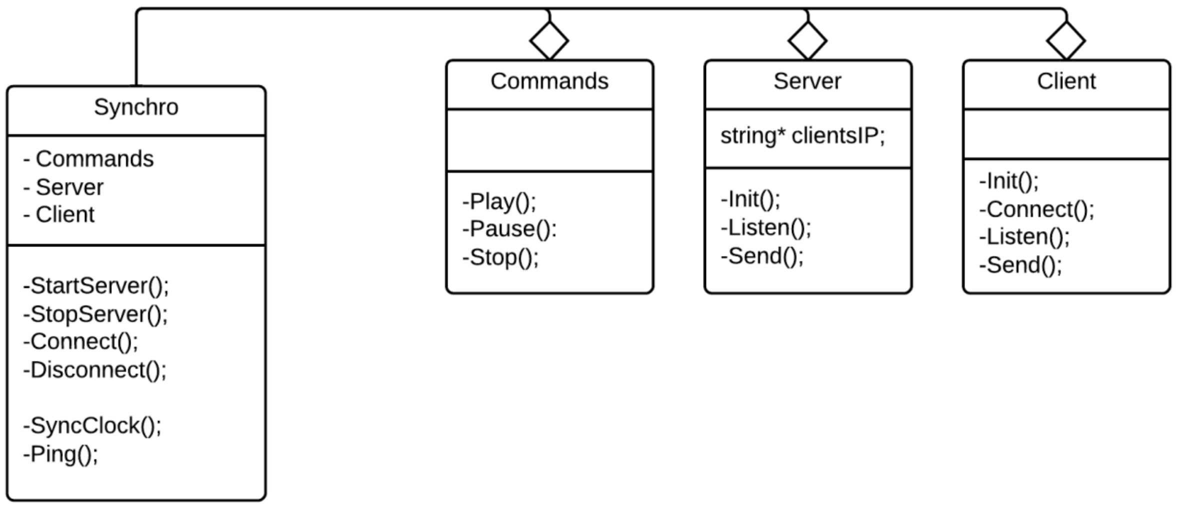
Where T = Time
Tc = Time to client

Server asks client to start playback at a defined time and adds the time it takes to reach the client. 10ms are added for safety.

#4



The diagram bellow is a UML representing the synchronization part present in the desktop applications:



6. Architecture, goals and technical constraints

A. Technical constraints

a. Smartphones' resource

Most of today's smartphones can't compete in term of resources with desktop computers. For that reason, we decided that the mobile applications would not offer some heavy functionality, which will therefore become exclusive to desktop clients. The AI[4] algorithm, for instance, will not be available in the mobile applications because it would use too much resource for a smartphone (memory, processor, battery, etc.).

b. Bandwidth

The streaming process implies a bandwidth with a very small latency. It won't be possible to use the streaming option if the network quality isn't good enough. If so, the application will let the user know and the streaming will be disabled automatically.

7. Application logical view

A. Global project view

Like many projects, AudioWire is composed of big two parts: a server and clients. However, we decided to split the server that contains the algorithm part from the AI[4] part. The AI[4] will have a direct access to the web server database available to clients through the API[1].

All the clients (including the web server which is handled as a normal client) communicate with the server through a REST[2] API[1]. This choice was made for several reasons:

- First, it ensures a better security of the data in our database, as the web server doesn't have a direct access to it.
- In addition, separating the AI[4] and API[1] allows a very good scalability as it allows us to put them on different servers when the traffic grows.

B. Communication

We will detail below the communication methods used between the different parts of the project.

The different parts are:

- A main server that will contain our REST[2] API[1]. This API[1] will use the HTTPS protocol in order to ensure a maximum of flexibility and simplicity using it. All the applications of the project will communicate through this unique server. This server will also contain a light version of the AI[4] to serve the mobile applications.

- A web server, written in NodeJS. This server communicates through HTTP, HTTPS and Websocket with the web clients. The web server will communicate with the desktop client using a D-BUS. D-BUS is an inter-process communication software allowing applications to communicate.
- A relational database, which contains users' data on the main server. As part of the server containing the API, this database will directly answer to the server's queries. It will also contain information about the music studied by the AI[4], and messages sent between the users.
- A music database, which will be used by our AI[4]. This database is a clone of an open source one, MusicBrainz[5]**.** It will be MySQL based, and the request will communicate directly with the AI[4], contained in AudioWire.
- Two mobile applications, Android and iOS. A tool like Titanium[6] will be used in order to write only one application working on both platforms. The HTTPS protocol will be used to communicate with the main server.
- A desktop client that will have, like all the project parts, to go through the API[1] (HTTPS protocol) in order to modify status for example, or do some music operations, etc. This client will contain an advanced AI[4]. It will also to communicate to the web clients via D-Bus.

C. Physical view of the application

In this section, we will present the physical part of AudioWire or at least what the project should look like in term of communication process and physically speaking.

First of all, it is important to notice that it is still difficult at this part of the development to establish the communication protocol precisely. That is why some terms may be too generic or fuzzy. However, we tried to be as clear as possible despite of these few constraints.

Based on the deployment diagram below, we can see that AudioWire is mainly composed of five distinct parts: the desktop application, the main server, the web application, the web server and the mobile applications which is a bit empty because we are not able to develop this part specifically (please take a look at the Gantt diagram for more information).

a. The desktop application

The desktop application regroups the clients on the Windows, Mac OS X and Linux platforms. These clients will contain all the features from AudioWire such Artificial Intelligence (which will allow us to adapt the music to the tastes of the users), and also user account management. Synchronized playback between multiple devices will be directly implemented on the client side. The devices will be synchronized thanks to the Network Time Protocol (NTP) to avoid latency between the devices. In other words, this means that the communication for the synchronized playback will use its own communication protocol and will not interfere with any other communication protocols.

b. The main server

The main server will host the database for the users account, the meta-data database and the spectra of the music. It will also contain a light AI[4] for mobile applications. Communication between the server and the external media will be made through the API.

c. Web application

The web application, which is the newest AudioWire feature, will be available on the most modern browsers. This client will integrate a remote control to control the desktop clients, the profile and friends list management and so on. You should notice that the social part of AudioWire is not fully implemented yet because it is not our priority at this time of the development.

Communication between the website and desktop application will be made with D-BUS, which is a communication system for computing inter-process communication.

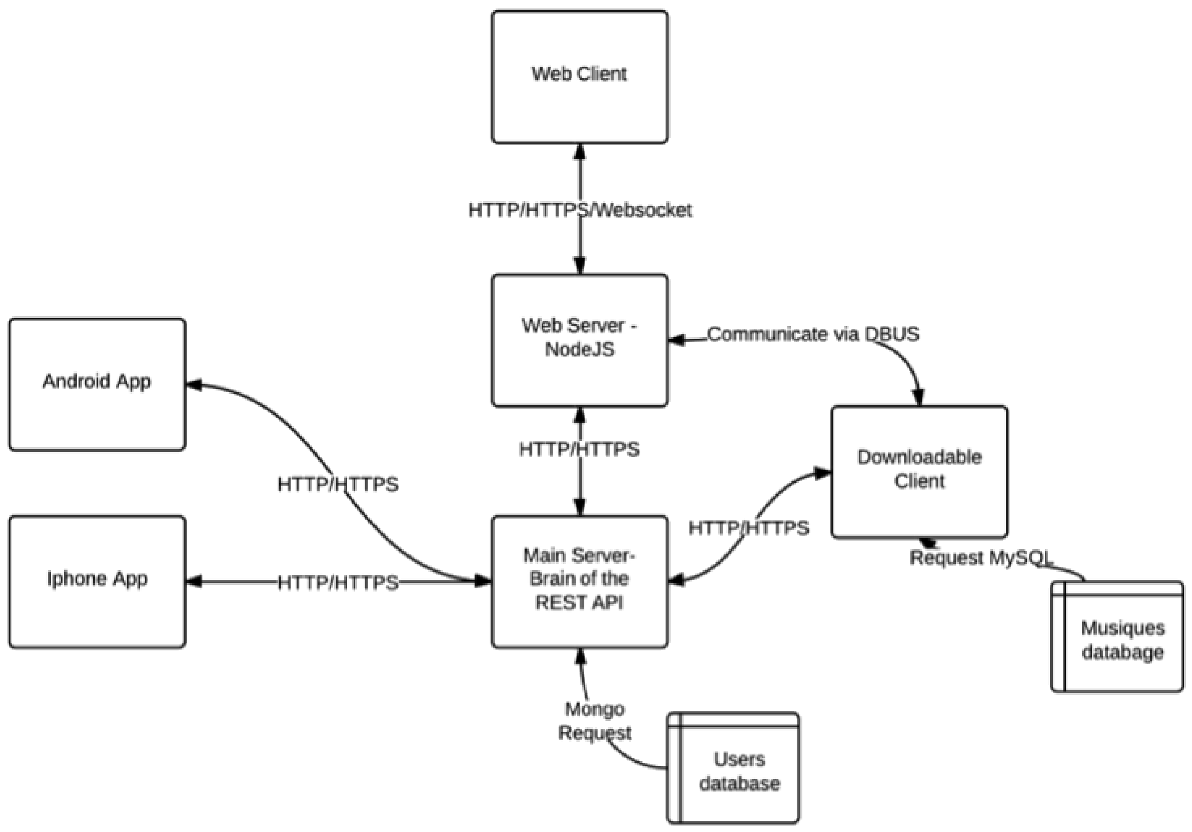
d. Web server

The web server will be written in Node.js. It is an event framework for writing network applications in JavaScript. Communication between the web server and the web application will be made through HTTPS / Websockets.

e. Mobile application

As said before, the smartphones will communicate with the API using the HTTPS protocol. These applications will contain all the classic features that can be found in the common music players plus a lightened AI[4] that will be running on the server side.

Here is the communication diagram, illustrating the way parts of the project will communicate and interact:



8. Client implementation

The development of the client covers two main areas: the development of an audio player on one hand and the development of an artificial intelligence on the other hand.

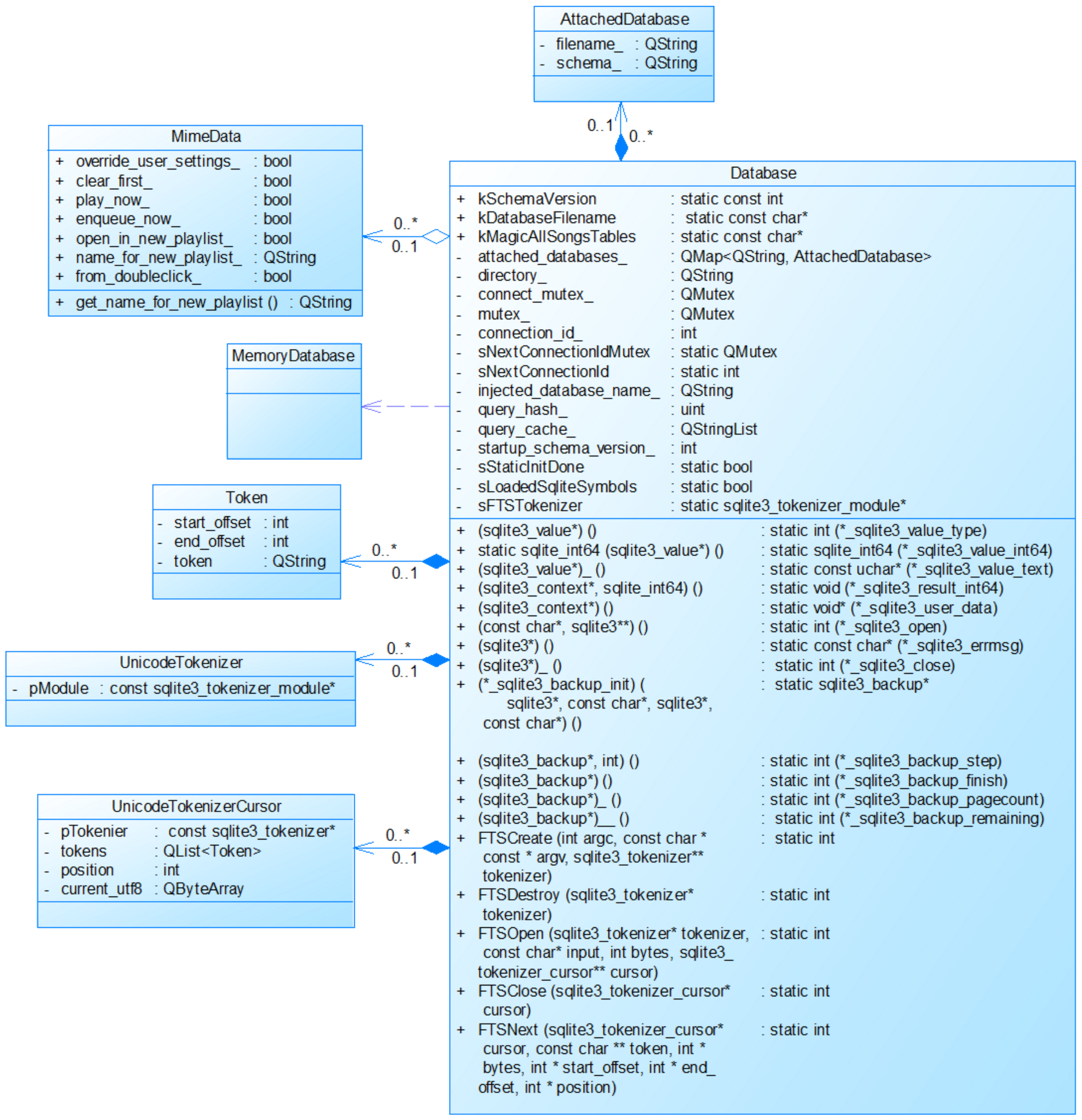
For now, we are implementing a “MVC” (Model-View-Controller) design pattern. It’s a way to build a program by separating its 3 main parts:

- The model is in charge of the data retrieval
- The view is in charge of presenting the data to the user and interacting with him
- The controller is a sort of event manager, linking the two previous parts.

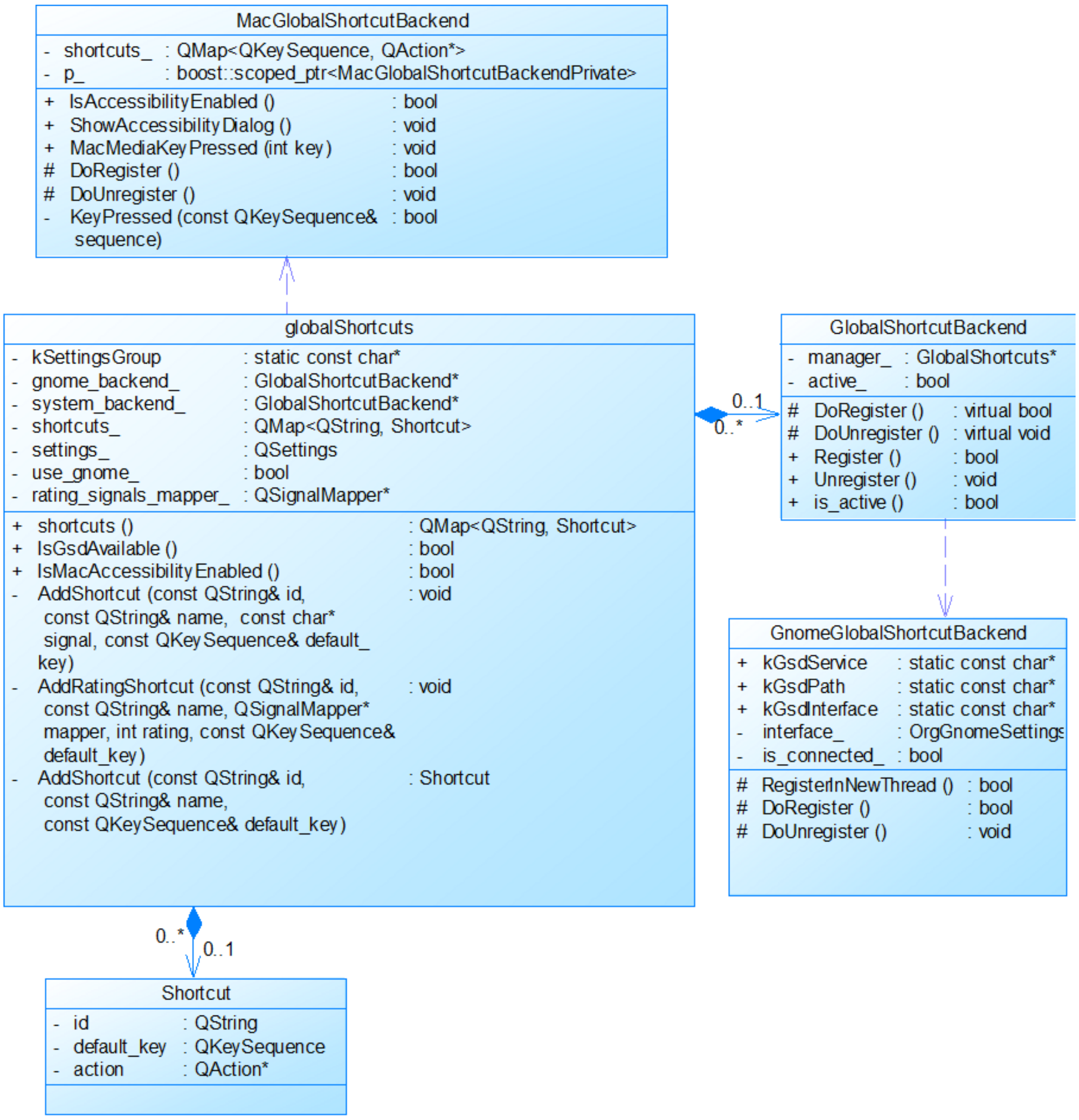
This architecture is really common for that kind of application because it is pretty simple to implement and easy to use. We are currently working on the representation of classes that we will use with AudioWire.

Here are some class diagrams representing this architecture:

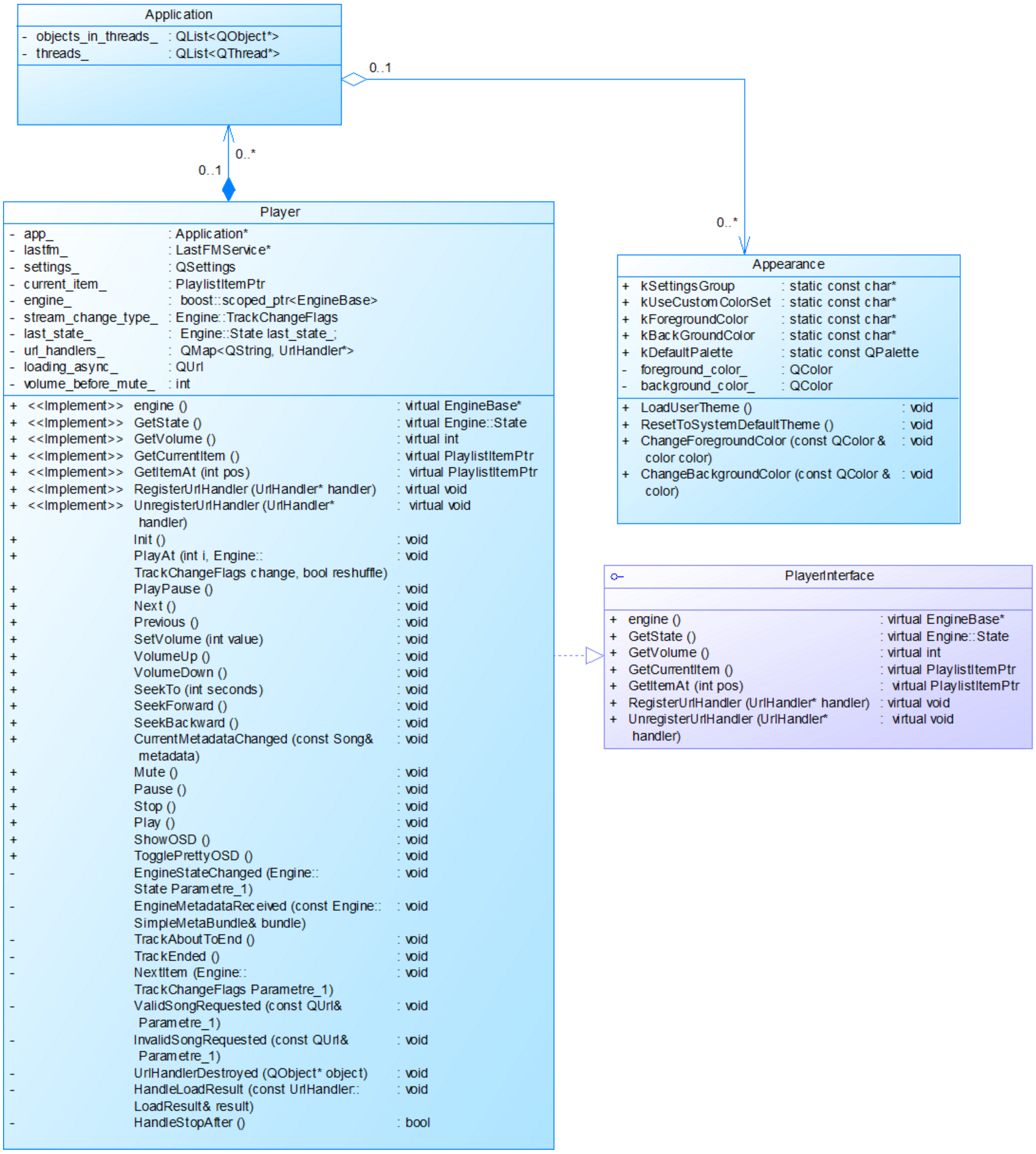
The first part is a representation of the AudioWire Database:


AttachedDatabase
- filename_ : QString
- schema_ : QString
0..1
0..*
MimeData
+ override_user_settings_ : bool
+ clear_first_ : bool
+ play_now_ : bool
+ enqueue_now_ : bool
+ open_in_new_playlist_ : bool
+ name_for_new_playlist_ : QString
+ from_doubleclick_ : bool
+ get_name_for_new_playlist () : QString
0..*
0..1
MemoryDatabase
Token
- start_offset : int
- end_offset : int
- token : QString
0..*
0..1
UnicodeTokenizer
- pModule : const sqlite3_tokenizer_module*
0..*
0..1
UnicodeTokenizerCursor
- pTokenier : const sqlite3_tokenizer*
- tokens : QList<Token>
- position : int
- current_utf8 : QByteArray
0..*
0..1
Database
+ kSchemaVersion : static const int
+ kDatabaseFilename : static const char*
+ kMagicAllSongsTables : static const char*
- attached_databases_ : QMap<QString, AttachedDatabase>
- directory_ : QString
- connect_mutex_ : QMutex
- mutex_ : QMutex
- connection_id_ : int
- sNextConnectionIdMutex : static QMutex
- sNextConnectionId : static int
- injected_database_name_ : QString
- query_hash_ : uint
- query_cache_ : QStringList
- startup_schema_version_ : int
- sStaticInitDone : static bool
- sLoadedSqliteSymbols : static bool
- sFTSTokenizer : static sqlite3_tokenizer_module*
+ (sqlite3_value*) () : static int (*_sqlite3_value_type)
+ static sqlite_int64 (sqlite3_value*) () : static sqlite_int64 (*_sqlite3_value_int64)
+ (sqlite3_value*)_ () : static const uchar* (*_sqlite3_value_text)
+ (sqlite3_context*, sqlite_int64) () : static void (*_sqlite3_result_int64)
+ (sqlite3_context*) () : static void* (*_sqlite3_user_data)
+ (const char*, sqlite3**) () : static int (*_sqlite3_open)
+ (sqlite3*) () : static const char* (*_sqlite3_errmsg)
+ (sqlite3*)_ () : static int (*_sqlite3_close)
+ (*_sqlite3_backup_init) (sqlite3*, const char*, sqlite3*, const char*) () : static sqlite3_backup*
+ (sqlite3_backup*, int) () : static int (*_sqlite3_backup_step)
+ (sqlite3_backup*) () : static int (*_sqlite3_backup_finish)
+ (sqlite3_backup*)_ () : static int (*_sqlite3_backup_pagecount)
+ (sqlite3_backup*)__ () : static int (*_sqlite3_backup_remaining)
+ FTSCreate (int argc, const char * const * argv, sqlite3_tokenizer** tokenizer) : static int
+ FTSDestroy (sqlite3_tokenizer* tokenizer) : static int
+ FTSOpen (sqlite3_tokenizer* tokenizer, const char* input, int bytes, sqlite3_tokenizer_cursor** cursor) : static int
+ FTSClose (sqlite3_tokenizer_cursor* cursor) : static int
+ FTSNext (sqlite3_tokenizer_cursor* cursor, const char ** token, int * bytes, int * start_offset, int * end_offset, int * position) : static int

The second part of this architecture separates the application controls to know the keyboard shortcuts and different events depending on different Operating systems:


MacGlobalShortcutBackend
- shortcuts_ : QMap<QKeySequence, QAction*>
- p_ : boost::scoped_ptr<MacGlobalShortcutBackendPrivate>
+ IsAccessibilityEnabled () : bool
+ ShowAccessibilityDialog () : void
+ MacMediaKeyPressed (int key) : void
# DoRegister () : bool
# DoUnregister () : void
- KeyPressed (const QKeySequence& sequence) : bool
globalShortcuts
- kSettingsGroup : static const char*
- gnome_backend_ : GlobalShortcutBackend*
- system_backend_ : GlobalShortcutBackend*
- shortcuts_ : QMap<QString, Shortcut>
- settings_ : QSettings
- use_gnome_ : bool
- rating_signals_mapper_ : QSignalMapper*
+ shortcuts () : QMap<QString, Shortcut>
+ IsGsdAvailable () : bool
+ IsMacAccessibilityEnabled () : bool
- AddShortcut (const QString& id, const QString& name, const char* signal, const QKeySequence& default_key) : void
- AddRatingShortcut (const QString& id, const QString& name, QSignalMapper* mapper, int rating, const QKeySequence& default_key) : void
- AddShortcut (const QString& id, const QString& name, const QKeySequence& default_key) : Shortcut
0..1
0..*
GlobalShortcutBackend
- manager_ : GlobalShortcuts*
- active_ : bool
# DoRegister () : virtual bool
# DoUnregister () : virtual void
+ Register () : bool
+ Unregister () : void
+ is_active () : bool
GnomeGlobalShortcutBackend
+ kGsdService : static const char*
+ kGsdPath : static const char*
+ kGsdInterface : static const char*
- interface_ : OrgGnomeSettings
- is_connected_ : bool
# RegisterInNewThread () : bool
# DoRegister () : bool
# DoUnregister () : void
0..*
0..1
Shortcut
- id : QString
- default_key : QKeySequence
- action : QAction*

Finally, the third part is the view of the MVC design pattern:


Application
- objects_in_threads_ : QList<QObject*>
- threads_ : QList<QThread*>
0..1
0..*
0..1
0..*
Player
- app_ : Application*
- lastfm_ : LastFMService*
- settings_ : QSettings
- current_item_ : PlaylistItemPtr
- engine_ : boost::scoped_ptr<EngineBase>
- stream_change_type_ : Engine::TrackChangeFlags
- last_state_ : Engine::State last_state_;
- url_handlers_ : QMap<QString, UrlHandler*>
- loading_async_ : QUrl
- volume_before_mute_ : int
+ <<Implement>> engine () : virtual EngineBase*
+ <<Implement>> GetState () : virtual Engine::State
+ <<Implement>> GetVolume () : virtual int
+ <<Implement>> GetCurrentItem () : virtual PlaylistItemPtr
+ <<Implement>> GetItemAt (int pos) : virtual PlaylistItemPtr
+ <<Implement>> RegisterUrlHandler (UrlHandler* handler) : virtual void
+ <<Implement>> UnregisterUrlHandler (UrlHandler* handler) : virtual void
+ Init () : void
+ PlayAt (int i, Engine::TrackChangeFlags change, bool reshuffle) : void
+ PlayPause () : void
+ Next () : void
+ Previous () : void
+ SetVolume (int value) : void
+ VolumeUp () : void
+ VolumeDown () : void
+ SeekTo (int seconds) : void
+ SeekForward () : void
+ SeekBackward () : void
+ CurrentMetadataChanged (const Song& metadata) : void
+ Mute () : void
+ Pause () : void
+ Stop () : void
+ Play () : void
+ ShowOSD () : void
+ TogglePrettyOSD () : void
- EngineStateChanged (Engine::State Parametre_1) : void
- EngineMetadataReceived (const Engine::SimpleMetaBundle& bundle) : void
- TrackAboutToEnd () : void
- TrackEnded () : void
- NextItem (Engine::TrackChangeFlags Parametre_1) : void
- ValidSongRequested (const QUrl& Parametre_1) : void
- InvalidSongRequested (const QUrl& Parametre_1) : void
- UrlHandlerDestroyed (QObject* object) : void
- HandleLoadResult (const UrlHandler::LoadResult& result) : void
- HandleStopAfter () : bool
Appearance
+ kSettingsGroup : static const char*
+ kUseCustomColorSet : static const char*
+ kForegroundColor : static const char*
+ kBackGroundColor : static const char*
+ kDefaultPalette : static const QPalette
- foreground_color_ : QColor
- background_color_ : QColor
+ LoadUserTheme () : void
+ ResetToSystemDefaultTheme () : void
+ ChangeForegroundColor (const QColor & color color) : void
+ ChangeBackgroundColor (const QColor & color) : void
PlayerInterface
+ engine () : virtual EngineBase*
+ GetState () : virtual Engine::State
+ GetVolume () : virtual int
+ GetCurrentItem () : virtual PlaylistItemPtr
+ GetItemAt (int pos) : virtual PlaylistItemPtr
+ RegisterUrlHandler (UrlHandler* handler) : virtual void
+ UnregisterUrlHandler (UrlHandler* handler) : virtual void