DA1

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

Table des matières

1 - A Propos d'AudioWire

A – Rappel de l'EIP

L'Epitech est une école d'expertise en informatique créée en 1999 dans la mouvance de l'Ecole pour l'informatique et les Techniques Avancées (Epita) pour accueillir les bacheliers passionnés qui souhaitent apprendre l'informatique par la technique et non par la théorie. Le cursus se déroule sur 5 ans et fourni un diplôme de niveau 1 par la Commission nationale de la certification professionnelle (CNCP).

La particularité et la force de cette école repose principalement sur le fait que les étudiants sont amenés à apprendre par eux-mêmes les notions recherchées dans le cadre de mini-projets de groupes et de projets plus ambitieux comme le PFA (Projet de fin d'année) et l'EIP (Epitech Innovative Project).

En effet, contrairement à l'enseignement scolaire classique où l'on amène la connaissance à l'élève afin qu'il puisse résoudre des problèmes, un étudiant à l'Epitech devra rechercher de lui-même les notions dont il a besoin avec son groupe de travail pour résoudre un problème ambiguë dans lequel il n'aura pas à l'origine les bases nécessaires. C'est de ce principe que naitra l'émergence de comportement que recherche l'Epitech, à savoir la capacité d'adaptabilité par rapport à une problématique et des concepts ainsi que la gestion d'une équipe de travail.

B – Contexte et périmètre du projet

De nos jours, il existe déjà un nombre important de lecteur audio très performant mais qui, d'une manière générale, ont plus ou moins les mêmes fonctionnalités. Nous ne voulons pas faire concurrence avec les lecteurs existant déjà, mais de créer quelque chose de nouveau. Etant donné que notre but premier est l'innovation, nous avons donc choisi de créer un lecteur audio équipé d'une intelligence artificielle capable de proposer une liste de lecture en fonction du comportement de l'utilisateur. En effet, Audiowire sera capable de proposer des musiques selon les goûts, les habitudes, voir l'humeur de l'utilisateur. Un mini réseau social sera aussi mis en place : chaque utilisateur pourra partager ses musiques avec ses contacts, ou aller voir leurs profils.

Audiowire sera bien évidemment multiplateforme, Windows, Mac OS, et Linux, nous avons aussi choisi des toucher un publique encore plus large : sachant qu'il y a aujourd'hui plus d'un milliard d'utilisateur de Smartphone. Les deux systèmes d'exploitation mobiles les plus populaires étant Android de Google et iOS d'Apple, nous avons donc choisi de nous y intéresser. Avec l'arrivée récente de Windows phone sur le marché, des experts et des chercheurs pensent que ce système d'exploitation mobile de Microsoft sera un concurrent direct d'Android et d'iOS. Nous envisageons peut-être de nous y investir dessus aussi, mais cela dépendra de l'avancement de notre projet.

Nous allons également créer une interface Web, sur laquelle l'utilisateur pourra gérer son compte, créer de nouvelles listes de lectures, et aussi garder contact avec ses amis. Il pourra bien évidemment télécharger le client version bureautique et mobile.

C – Définition, Acronymes et Abréviations

[1] *API* : « Application Programming Interface » ou « Interface de programmation » en français. Il s'agit d'une interface fournit par un programme informatique permettant une interaction avec des composants externes.

[2] *REST* : « REpresentational State Transfer ». Il s'agit d'un style d'architecture permettant s'échanger des données avec un système de manière structurée.

[3] *JSON* : « JavaScript Object Notation » est un format de données textuel permettant de représenter ces données de manière générique et structurée.

[4] *XML* : « Extensible Markup Language » ou « langage de balisage extensible » en français. Il s'agit d'un format de données textuel permettant de représenter ces données de manière générique et structurée.

[5] *IA* : « Intelligence Artificielle ». Programme ou algorithme dont le but est de rapprocher le plus possible son comportement s'un comportement humain.

[6] *Foreign Key* : « Contraintes de clés étrangères » en français. Dans le contexte d'une base de donnée relationnel telle que MySQL, il s'agit d'un champ représentant une référence d'une clé primaire d'une table dans une autre. Pour plus d'information, se référer à la documentation de MySQL.

2 - Représentation de l'architecture globale

3 - Vue globale du projet

A – Le serveur

1 - La Base de données

Le projet AudioWire requiert une base de données ayant un double objectif. Celle-ci contiendra d'une part les informations concernant les utilisateurs et d'autre part les informations recueillis sur les musiques de nos utilisateurs.

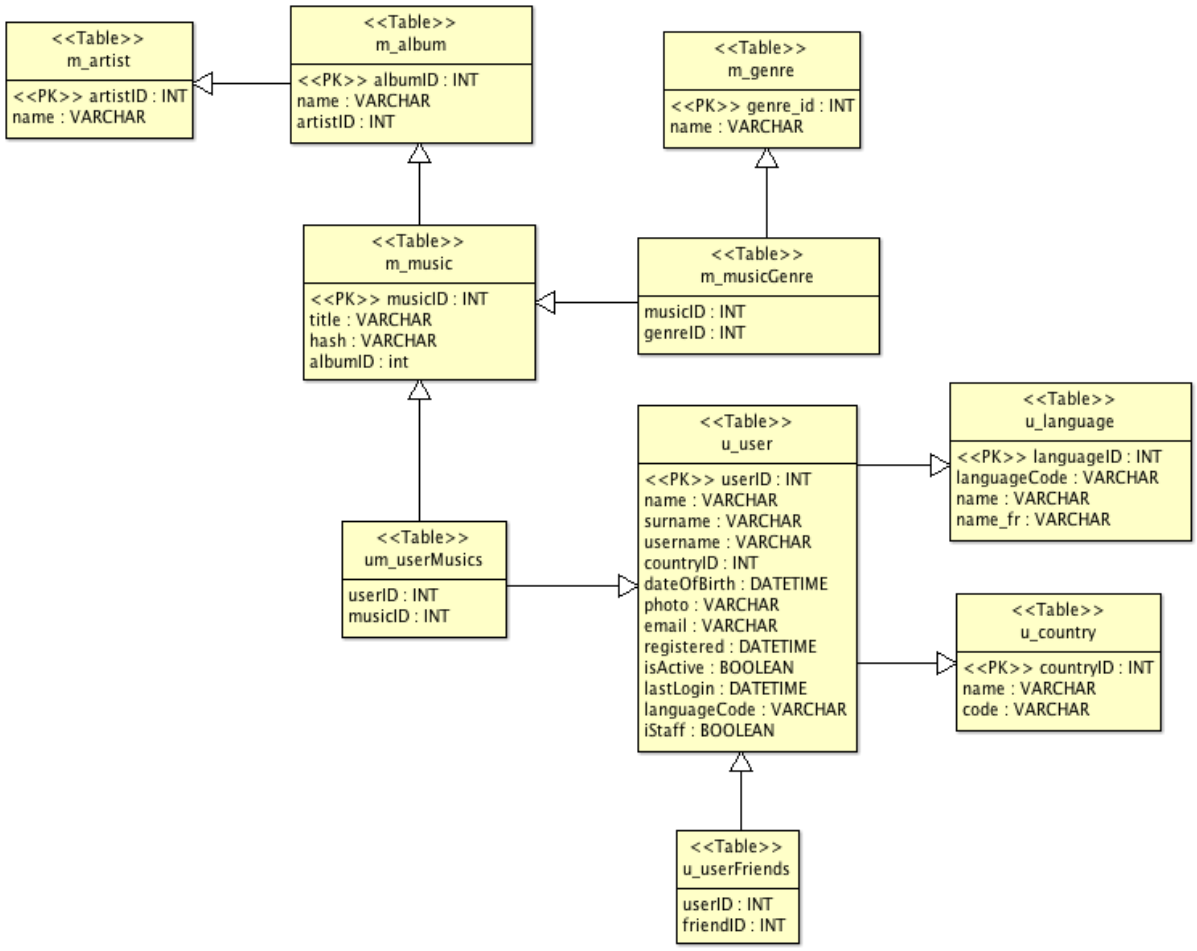
Premièrement, les tables ayant le suffixe « u_ » concernent les utilisateurs. La table u_langage contient une liste de tous les langages existants afin de permettre de s'assurer que l'utilisateur nous donnera un langage qui existe vraiment. Cette information pourra notamment servir à la gestion de l'internationalisation, d'où la nécessité d'avoir des langues qui existent vraiment. La table u_country contient une liste des pays du monde avec leur code pays respectifs. Tout comme la table u_language, cette dernière nécessite de contenir que des pays qui existent réellement car cette information pourra rentrer dans les données prises en compte par l'intelligence artificielle. La table u_user représente un utilisateur. Elle contient donc ses informations comme son nom, son prénom, son âge etc. Elle contient également des foreign keys vers la table u_language et u_country. Enfin, la table u_userFriends permet de sauvegarder les listes d'amis de nos utilisateurs, permettant ainsi de synchroniser celle-ci entre les différentes plateformes.

Deuxièmement, la base de données AudioWire contient des tables permettant de stocker les informations sur les musiques des utilisateurs. Ces données seront utilisées dans le cadre de l'intelligence artificielle et ont le préfixes « m_ ». La table m_artist permet de stocker les différents artistes ainsi que les différentes informations les concernant. La table m_album contient les informations sur un album comme son artiste par exemple. La table m_music représente une musique et contient toutes les informations la concernant ainsi que sa signature. Le fait de stocker la signature des musiques à l'énorme

avantage de permettre aux utilisateurs de ne pas avoir à générer de signature si un autre utilisateur l'a déjà fait ou ne pas avoir à interroger la base de données contenant les informations sur toutes les musiques si un autre utilisateur l'a déjà fait. La table m_genre permet de stocker une liste de tous les genres de musique écoutée par les utilisateurs. Afin de pouvoir associer plusieurs genres à une musique, la table m_musicGenre a été créée. Celle-ci permet d'associer un id de musique à différents ids de genre.

Enfin, les tables portant les préfixes « um_ » regroupent les tables qui concernent à la fois les musiques et les utilisateurs. Pour le moment, il n'existe qu'une seule table appelée um_userMusics. Celle-ci permet d'associer les ids de musique à des utilisateurs afin de savoir quelles musiques possède un utilisateur.

Pour conclure, la base de données AudioWire contient deux « sous base de données » identifiables par le préfixe des tables. Ces deux sous base de données servent d'une part à stocker des informations sur les utilisateurs et d'autre part à stocker des informations sur la musique. Les tables n'ont pas été séparées dans des bases de données différentes car certaines tables nécessitent d'accéder aux deux en même temps. Ces dernières portent les préfixes « um_ ». Les tables contenues dans cette base de données sont évidemment vouées à évoluer en fonction des besoins lors de l'avancement du projet.



Représentation de la base de données d'AudioWire

2 – L'API[1]

Afin de permettre à tous les clients du projet (Android, IPhone, application PC/MAC et site web) de communiquer avec le serveur principal, une API[1] de type REST[2] sera mise à disposition. Cette API[1] supportera les formats JSON[3] et XML[4], et répondra dans le même format que la requête (soit XML[4], soit JSON[3]). Cette API[1] sera disponible sur le serveur principal d'AudioWire et permettra

notamment d'accéder à la base de données ainsi qu'aux différentes options de l'intelligence artificielle.

Cette API[1] utilisera les protocoles HTTP et HTTPS afin de garantir un maximum de flexibilité et de simplicité dans son utilisation. En effet, la majorité des langages de programmation moderne (tel que Java ou Python par exemple) implémentent ces protocoles nativement et permettent de faire des requêtes de manière très simple et rapide.

De plus, le fait d'utiliser une API[1] REST[2] permettra également à des développeurs tiers de créer leur propre client AudioWire de manière très simple. Une documentation complète de l'API[1] détaillant ses fonctionnalités devra être fournie à cet effet.

B – Le Client

1 – le client lourd

Le client lourd également appelé application de bureau, sera l'application principale d'AudioWire car c'est lui qui contiendra toutes les fonctionnalités du projet (Hors mis les fonctions propres aux téléphones mobiles tels que la télécommande, etc). Il sera entièrement développer en c++ car d'une part c'est un langage qui est maîtrisé par la totalité des membres du groupes mais surtout car c'est un langage objet de bas niveau, ce qui nous permettra d'optimiser au maximum les performances du lecteur.

De plus, l'intelligence artificielle sera elle aussi principalement développée en c++, ce qui minimisera les risques d'incompatibilités entre IA et lecteur audio. Il implémentera toutes les fonctionnalités de bases d'un lecteur audio à savoir, la gestion de bibliothèques et la lecture de liste de lectures mais il intègrera également des notions sociales telles que la gestion de contacts, le partage entre amis de ces listes de lectures, de ces bibliothèques. Il pourra également indiquer son humeur, ses envies du moment ou alors ses pensées. Cela permettra de récupérer les informations sur certains utilisateurs par le biais de ses amis ou de voir également de quelles manières ils utiliseront les bibliothèques

partagées. On peut voir ci-dessous un diagramme global de cas d'utilisations qu'implémentera le client de bureau d'AudioWire.

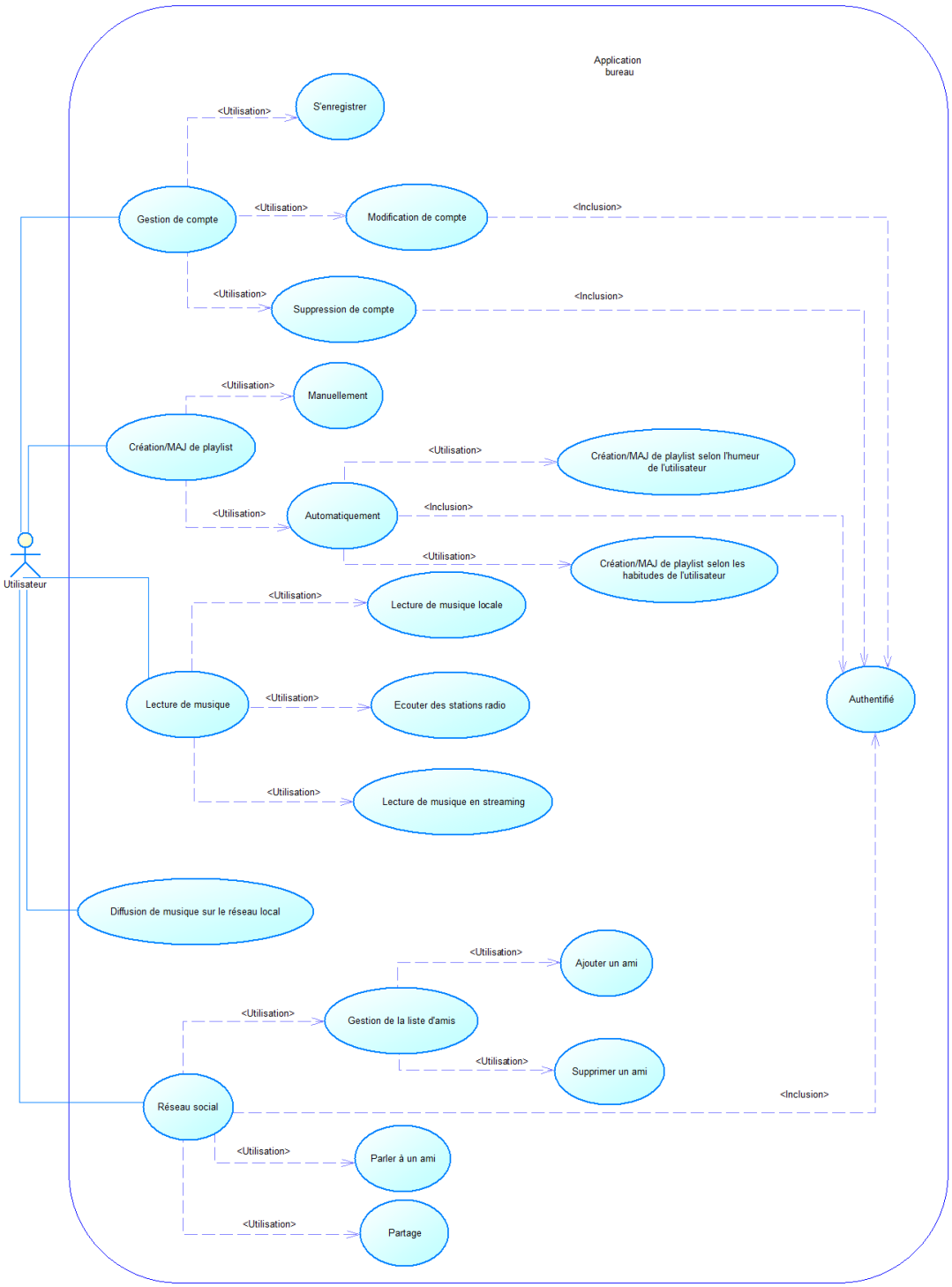


Diagramme de cas d'utilisations du client lourd

2 – Les clients mobiles

Concernant les applications mobiles, nous avons choisi de nous intéresser aux deux plateformes mobiles les plus populaires, iOS et Android. En raison des performances limitées des téléphones portables en comparaison aux ordinateurs de bureau, nos applications mobiles n'implémenteront pas certaines des fonctionnalités des versions de bureau. L'utilisateur pourra bien évidemment gérer ses listes de lectures via sa page de profil, changer ses informations personnelles, accéder à sa liste d'amis, et bien sûr écouter la musique stockée sur son téléphone.

Toutes les fonctionnalités qui demandent une quantité trop importante de ressources ne seront pas implémentées. Par exemple les algorithmes d'intelligence artificielle ne se seront pas présents. Cependant, certaines fonctionnalités seront spécifiques aux applications mobiles, comme une fonction télécommande qui permettra de contrôler un client lourd à partir d'un téléphone. Il sera également possible de synchroniser les listes de lectures des clients lourds avec les clients mobiles.

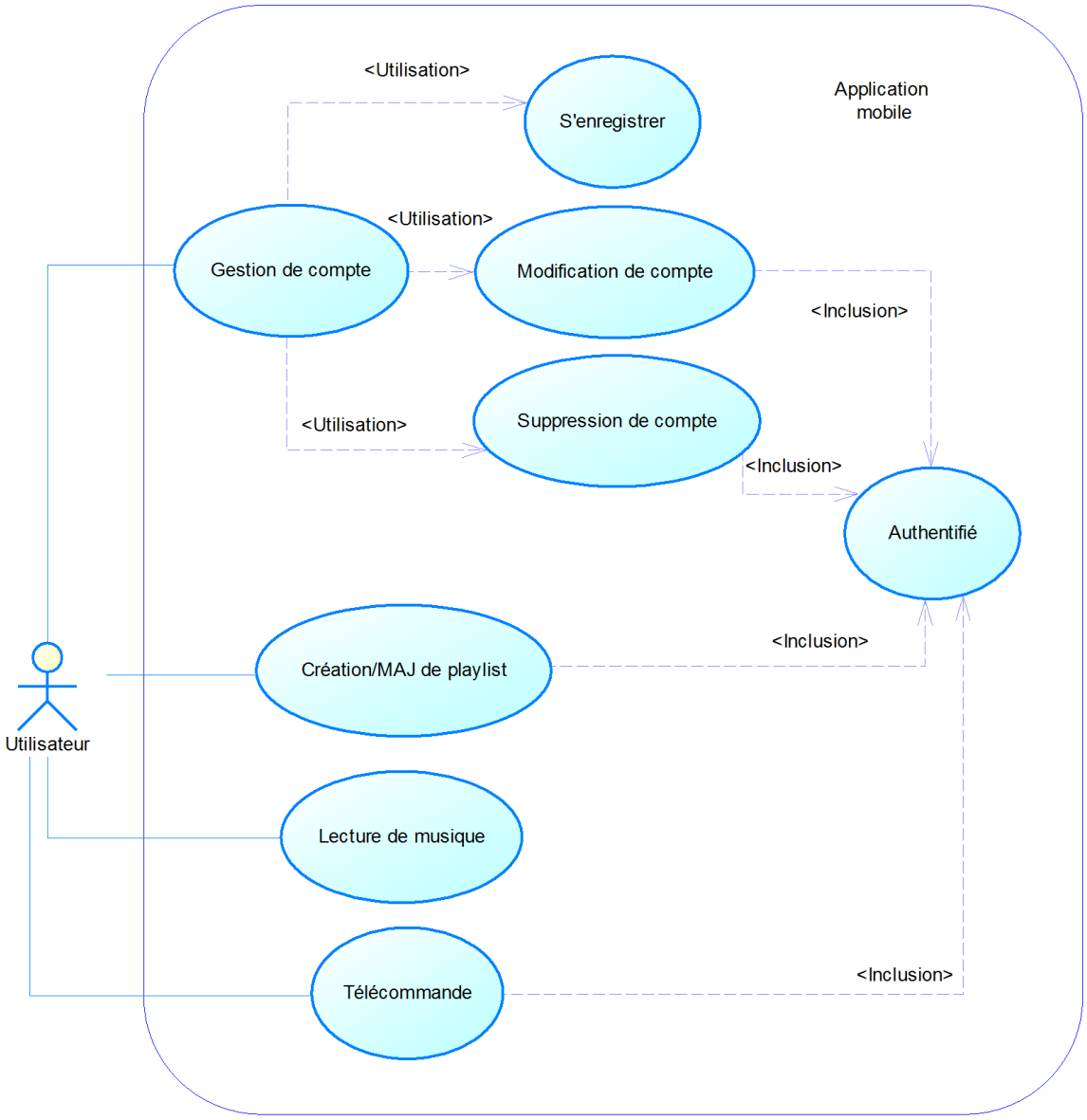


Diagramme de cas d'utilisation des applications mobiles d'AudioWire

3 – Site web

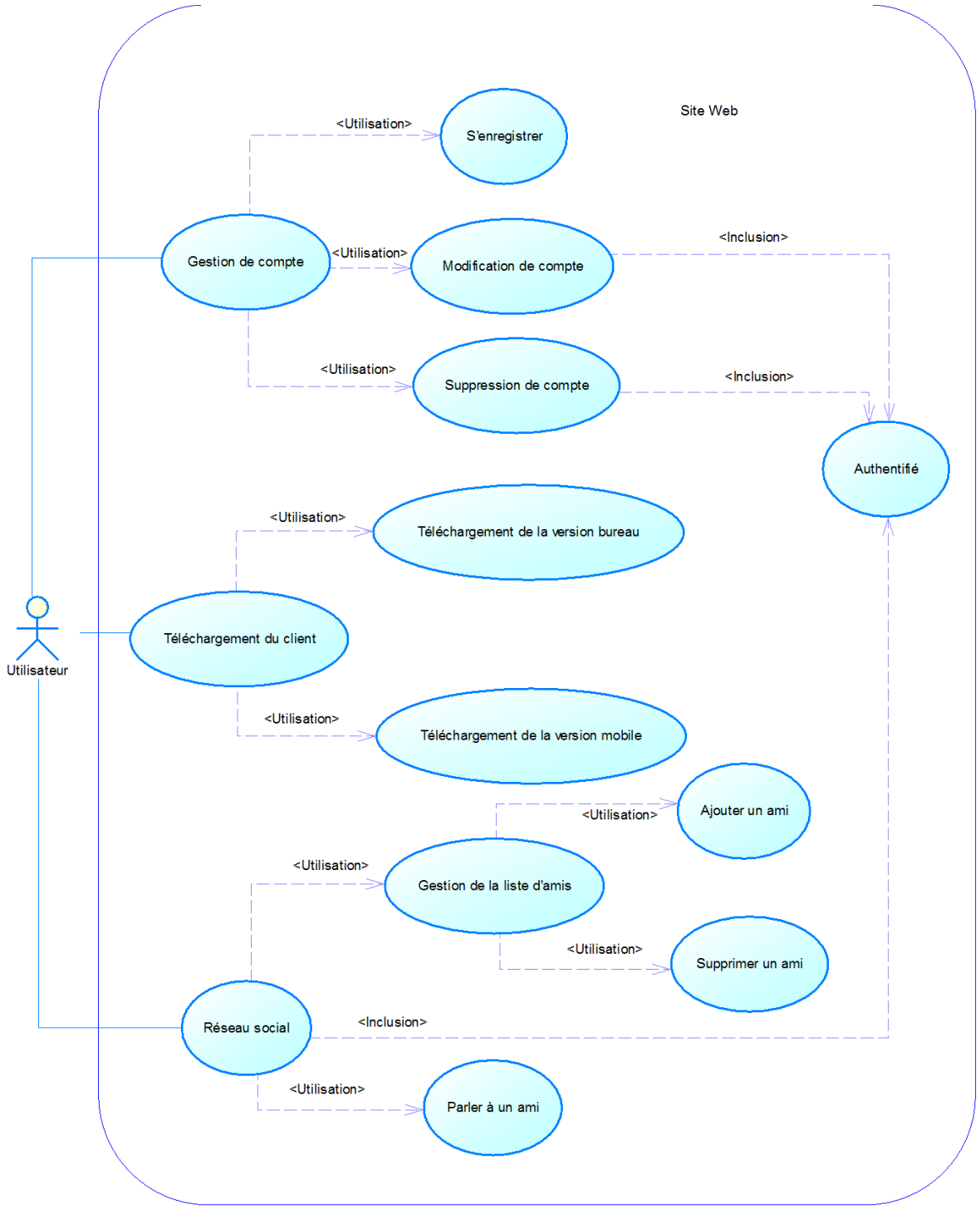


Diagramme de cas d'utilisation de l'application Web

4 – Le lecteur audio

Clémentine est un projet libre basé sur le lecteur musical Amarok, un lecteur puissant et connu du monde du logiciel libre. Dans un premier temps, nous expliquerons en quelques lignes ce qu'est exactement Clémentine à savoir comment il est codé, quelle sont ses propriétés puis nous détaillerons sous quelles formes nous l'adapterons à AudioWire.

A – Pourquoi Clémentine ?

Le lecteur Clémentine est développé en c++. Il correspond donc à nos critères de développement. De plus, son architecture est très générique et son système de « plug-in » en fait un avantage considérable car il nous permettra de nous intégrer rapidement et parfaitement avec notre client lourd. L'interface graphique de Clémentine est quant à elle faites avec QT, une API[1] graphique multiplateforme et très facile d'utilisation.

B – Implémentation avec AudioWire

Nous réutiliserons donc une partie du code source de Clementine notamment pour son architecture qui est très intéressante pour nous. En effet, elle utilise le système de plug-in, qui est une manière de programmer qui nous permet de rajouter facilement des bouts de codes pour rajouter des fonctionnalités au projet. Ce qui nous permettra de tester au fur et à mesure les améliorations apportées à AudioWire de manière pertinente.

5 – L'IA[5]

6 – Architecture, buts et contraintes techniques

A - Contraintes techniques

1 – Puissance des téléphones mobiles

La majorité des téléphones portables ne peuvent rivaliser en terme de puissance avec des ordinateurs de bureau. C'est pour cette raison que nous avons décidé que nos applications mobiles n'offriraient pas certaines fonctionnalités lourdes en ressources qui seront donc spécifiques aux versions de bureau. Les algorithmes d'intelligence artificielle, par exemple, ne seront pas présents sur les mobiles car ils risqueraient de mal fonctionner et d'utiliser trop de ressources du téléphone (mémoire, processeur, batterie, etc.).

2 – Bande Passante

Le procédé de streaming implique une bonne bande passante avec un temps de latence relativement faible. Il sera donc impossible, si la qualité de la connexion réseau n'est pas suffisante, d'utiliser les options de streaming. Notre logiciel effectuera des tests de bande passante avant d'établir des connexions inter-utilisateurs pour le streaming. Si ceux-ci ne s'avèrent pas satisfaisant, le logiciel préviendra l'utilisateur et le streaming sera automatiquement désactivé.

3 – Stabilité et puissance des serveurs

Dans le cadre du projet AudioWire, le serveur est chargé de récupérer des informations concernant les utilisateurs et leurs musiques. Toutes ses

informations seront sauvegardées dans notre base de données et permettront aux algorithmes de l'intelligence artificielle de générer des résultats pertinents. C'est pour cela que la stabilité du serveur est primordiale que ce soit au niveau matériel ou au niveau logiciel.

Les tests de premiers niveaux que nous ferons après avoir développé la partie du serveur nous permettront de tester la partie logicielle et de nous assurer de sa stabilité. Sur le plan matériel, le laboratoire EIP nous a autorisés à utiliser un de leur serveur, si celui-ci est suffisamment puissant pour les besoins de l'intelligence artificielle, nous ne devrions pas avoir de soucis majeurs pendant la période de développement. Nous verrons une fois le projet terminé si nous devons investir dans de nouvelles infrastructures plus performantes.

7 - Vue logique de l'application

A – Vue globale du projet

Comme beaucoup de projets, le projet AudioWire est composé de deux grandes parties qui sont un serveur et des clients. Cependant, dans le cadre d'AudioWire, nous avons décidé de séparer le serveur qui contient la partie algorithmique de l'intelligence artificielle et qui a un accès direct à la base de données du serveur web disponible aux clients. Ainsi, le serveur web est traité comme un client « normal » en ce qui concerne le serveur principal, au même titre que les clients mobiles (iOS et Android) et les clients lourds (PC/MAC). Ainsi, tous les clients communiqueront avec le serveur principal à travers une API[1] de type REST[2].

Ce choix a été réalisé pour plusieurs raisons. Tout d'abord, cela permet de garantir une plus grande sécurité des données contenues dans notre base de données en ne donnant pas un accès direct à la base de données depuis le serveur web qui est le plus vulnérable aux attaques extérieures. De plus, le fait de séparer la charge du serveur web de celle de l'intelligence artificielle sur deux serveurs différents garantira des performances constantes du service car par exemple, si le serveur web reçoit une forte charge à un moment donné, cela ne doit en aucun cas impacter les performances du serveur principal, et vice-versa. Enfin, cela permet une implémentation simplifiée et plus rapide des

services sur les différentes plateformes car la même API[1] REST[2] pourra être utilisée à la fois pour les clients mobiles, lourds et web. Cela évite par exemple de dupliquer le code qui permet d'accéder à la base de données.

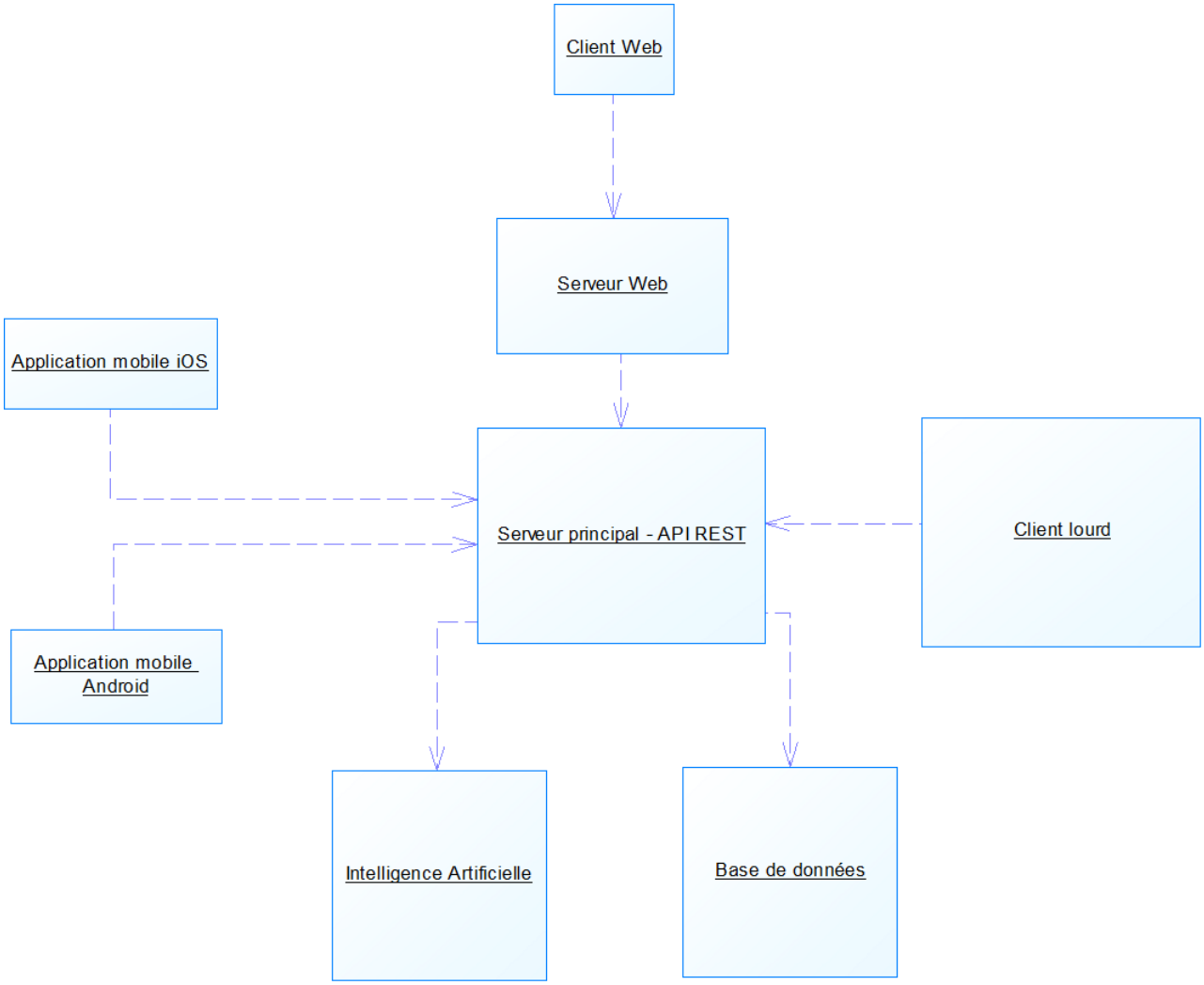
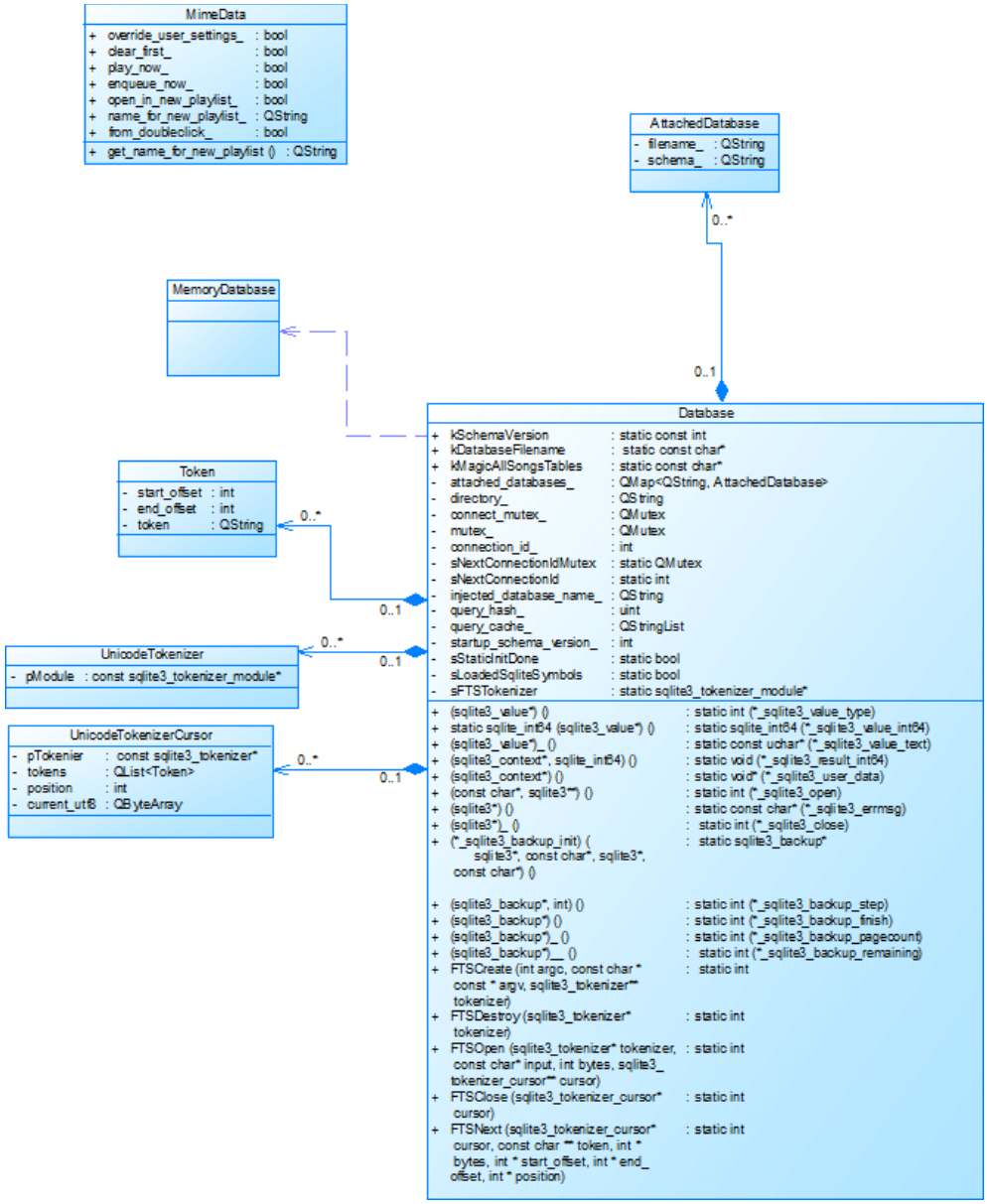


Diagramme de communication d'AudioWire

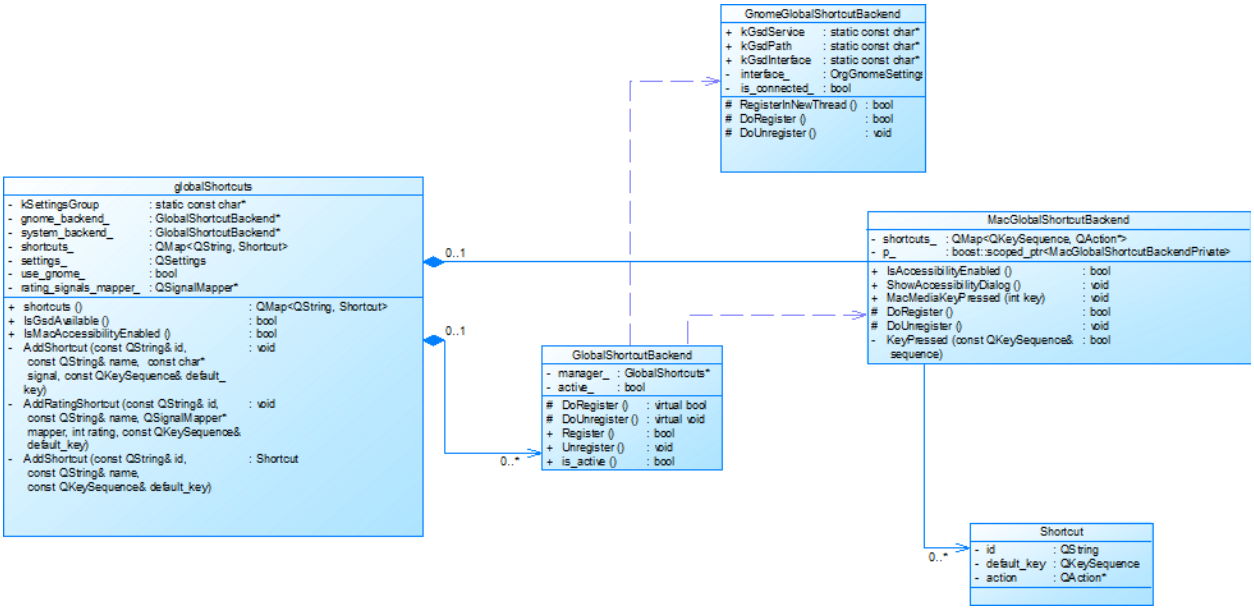
8 – Implémentation du client

Le développement s'étend sur deux points principaux : le développement du lecteur audio d'un côté et sur l'intelligence artificielle de l'autre. Pour le moment nous sommes partis sur un design pattern «Modèle-Vue-Contrôleur » classique car c'est une technique les plus utilisées pour ce type d'application. En effet, le design MVC est une manière de construire un projet, une architecture qui sépare les différentes problématiques liées aux applications interactives à savoir le modèle de données, le contrôleur (gestionnaire d'évènements) et la vue (l'interface graphique). Nous sommes actuellement en train de travailler sur la représentation des classes que nous allons utiliser en nous aidant de l'architecture de Clementine. Voici quelques diagrammes de classes de l'architecture du client d'AudioWire :

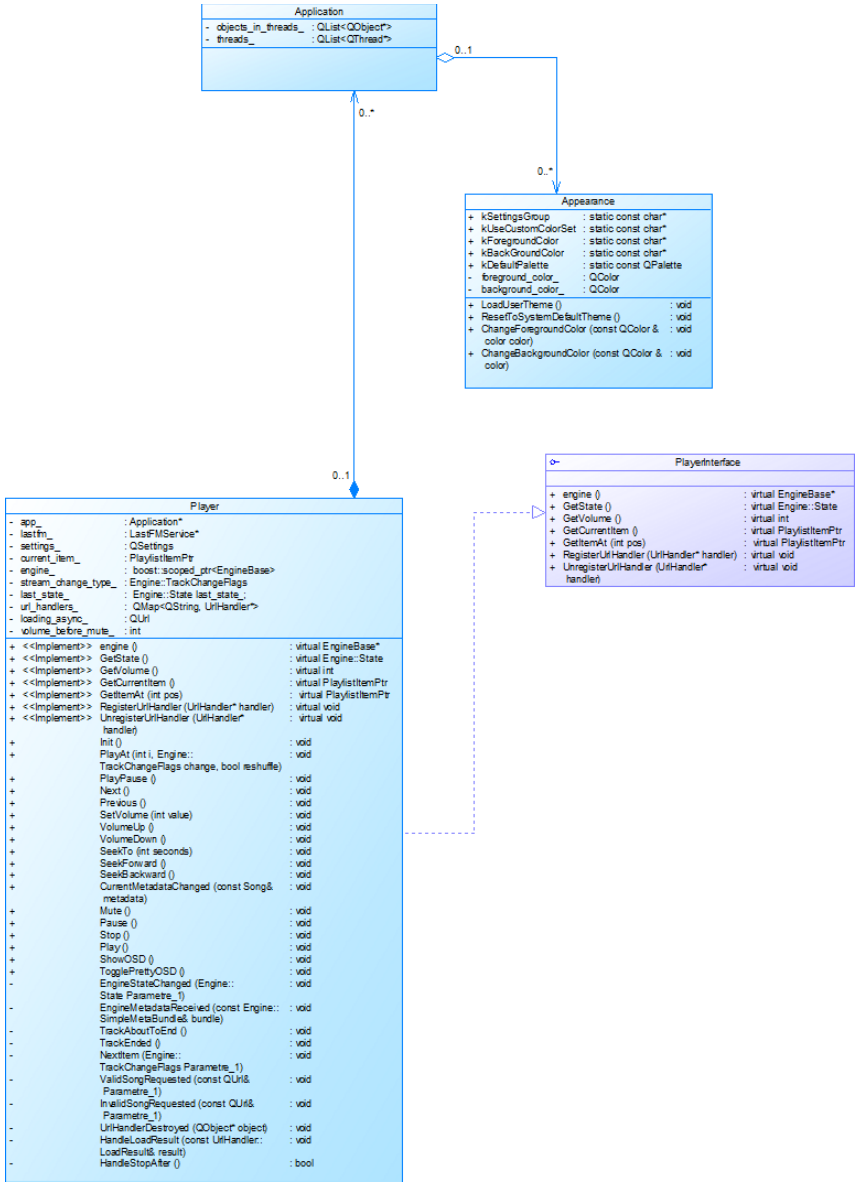
Voici une première ébauche de la manière dont la BDD sera implémentée.



La deuxième partie de l'architecture consiste à séparer les contrôles de l'application à savoir les raccourcis clavier et les différents évènements en fonction des systême d'exploitations de la machine.



Enfin La troisième partie représente la vue, celle-ci n'est pas encore très détaillée car elle est en pleine implémentation avec la partie de Clementine.


Application
- objects_in_threads_ : QList<QObject*>
- threads_ : QList<QThread*>
0..1
0..*
0..*
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
0..1
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
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