DA3

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

Tableau des révisions :

Date	Auteur	Section	Commentaire
23/04/13	AudioWire	Toutes	Refonte des parties, réattribution des tâches aux différents membres du groupe.
15/05/13	AudioWire	Toutes	Merge de toutes les parties du DA3, premier bilan.
19/05/13	Hugo – Vivien	3-A / 8-A / 9	Mise a jour des parties concernant l'API + 2/3 fautes trouvées
22/05/2013	Guillaume	Toutes	Relecture et correction des fautes d'orthographes
29/05/2013	Stéphane	Toutes	Mise en page et relecture générale
31/05/2013	Stéphane, Hugo, Max	Toutes	Finalisation du rendu finale
12/11/2013	Stéphane	Partie client lourd	Mise à jour des définitions du client lourd

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 plutôt 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, voire 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 son profil.

AudioWire fonctionnera sera bien évidemment sur toutes les plateformes modernes : Windows, Mac OS, et Linux. De plus, sachant qu'il y a aujourd'hui plus d'un milliard d'utilisateurs de smartphone, nous avons décidé d'être également présent sur les deux plateformes les plus populaires : Android et iOS.

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

C – Définitions, 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 textuelles permettant de représenter ces données de manière générique et structurée.

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

[5]*MusicBrainz* : Base de données musicales, collaborative, universelle, librement diffusable référençant des enregistrements d'œuvres, et non des œuvres en elles-mêmes diffusables.

Site officiel : http://musicbrainz.org/

[6]*Titanium* : Kit de construction de logiciels libres de droits permettant de développer des applications natives pour iOS et Android avec un code commun. Site officiel : http://www.appcelerator.com/platform/titanium-sdk/

2. Vue globale du projet

A. Le serveur

Tous les clients du projet (Android, IPhone, application PC/MAC et site web) communiqueront avec le serveur principal grâce à une API[1] de type REST[2]. Celle-ci permettra notamment d'accéder à la base de données principale du projet, sera chargé de l'authentification des utilisateurs et des fonctionnalités sociales.

Cette API[1] utilisera le protocole HTTPS afin de garantir un maximum de flexibilité et de sécurité dans son utilisation. En effet, la majorité des langages de programmation moderne (tel que Java ou Ruby ou Python par exemple) implémentent ce protocole nativement et permet de faire des requêtes de manière rapide et sécurisé. Cette API[1] fonctionnera à l'aide d'un serveur en Ruby on Rails et la plupart des base de données moderne pourront être utilisé pour stocker les données. Ceci dit, nous recommandons l'utilisation de MySQL.

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.

Voici les requêtes qui seront disponibles au travers de l'API[1] :

Utilisateur:

URL	Méthode HTTP	Paramètres	Fonction
/users/sign-in/	POST	email : String, password : String	Se connecter / créer un token
/users/sign-out/	POST	id : Integer (Token)	Se déconnecter / invalider le token
/users/id/password/	PUT	id : Integer, oldPassword : String, newPassword : String	Mettre à jour le mot de passe
/users/sign-up/	POST	email : String, password : String	S'enregistrer
/users/id/	PATCH	id : integer, Les champs mis à jour	Mettre à jour les informations d'un compte utilisateur
/users/list/	GET	id : integer	Retourne la liste d'utilisateurs enregistrés

Liste d'amis :

URL	Méthode HTTP	Paramètres	Fonction
/friendlist/	POST	name : String	Créer une liste d'amis
/friendlist/	GET	Aucun	Retourne la liste des listes d'amis
/friendlist/id/	GET	Aucun	Retourne des informations sur la liste d'amis
/friendlist/id/	DELETE	Aucun	Supprime la liste d'amis
/friendlist/id/friend/	POST	friend : id	Ajoute un ami dans la liste d'amis
/friendlist/id/friend/	DELETE	friend : id	Supprime l'ami en paramètre de la liste d'amis

Musique:

URL	Méthode HTTP	Paramètres	Fonction
/songs/	GET	Aucun	Retourne la liste des musiques
/songs/	POST	band : String, title : String, track : String	Envoyer les informations sur une musique

/songs/id/	GET	id:Integer	Obtenir les informations sur une musique
/songs/id/	PATCH	band : String and/or title : String and/or track : String	Mettre à jour une musique partiellement
/songs/id/	DELETE	Aucun	Supprime une musique de la liste de musique d'un utilisateur

Image:

URL	Méthode HTTP	Paramètres	Fonction
/images/	POST	Fichier	Envoi d'une image
/images/id/	GET	id : Integer	Récupérer les informations d'une image
/images/id/	DELETE	id : Integer	Supprimer une image

Message:

URL	Méthode HTTP	Paramètres	Fonction
/messages/	GET	Aucun	Liste tous les messages qu'un utilisateur a reçus ou envoyé
/messages/id/	GET	Aucun	Retourne un message
/messages/id/	PUT	Content : String	Met à jour un message
/messages/id/	DELETE	Aucun	Supprime un message

/messages/sent/	GET	Aucun	Liste tous les messages envoyé
/messages/received/	GET	Aucun	Liste tous les messages reçu
/messages/	POST	User : ID, content:String	Envoie un message à un autre utilisateur
/messages/<user>	GET	Id de l'utilisateur à ajouter dans l'url	Liste tous les messages envoyé et reçu à l'utilisateur passé en paramètre

Afin d'authentifier les utilisateurs, toutes les requêtes (sauf la requêtes de création de compte et de connexion) devront contenir le paramètre GET « auth_token » contenant le token. Les requêtes auront donc le format : « /<request>/auth_token=<token> ».

Toutes les requêtes répondront en utilisant les codes de réponse HTTP standard et contiendront éventuellement des données encodées en JSON[3].

Voici les codes d'erreurs utilisés :
200 => Ok
201 => Created
204 => No Content
400 => Bad Request
401 => Unauthorized
403 => Forbidden
500 => Internal Server Error

B. Le Client

a. Le client lourd

Le client lourd également appelé application de bureau, sera l'application principale d'AudioWire car c'est elle qui contiendra toutes les fonctionnalités du projet (mis-à-part les fonctions propres aux téléphones mobiles tels que la télécommande, etc.). Il sera entièrement développé en C++ car

d'une part c'est un langage qui est maîtrisé par la totalité des membres du groupe mais surtout car c'est un langage objet de bas niveau, ce qui nous permettra d'optimiser au maximum les performances du lecteur audio.

De plus, l'intelligence artificielle sera elle aussi principalement développée en c++, ce qui minimisera les risques d'incompatibilités entre IA[4] 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.

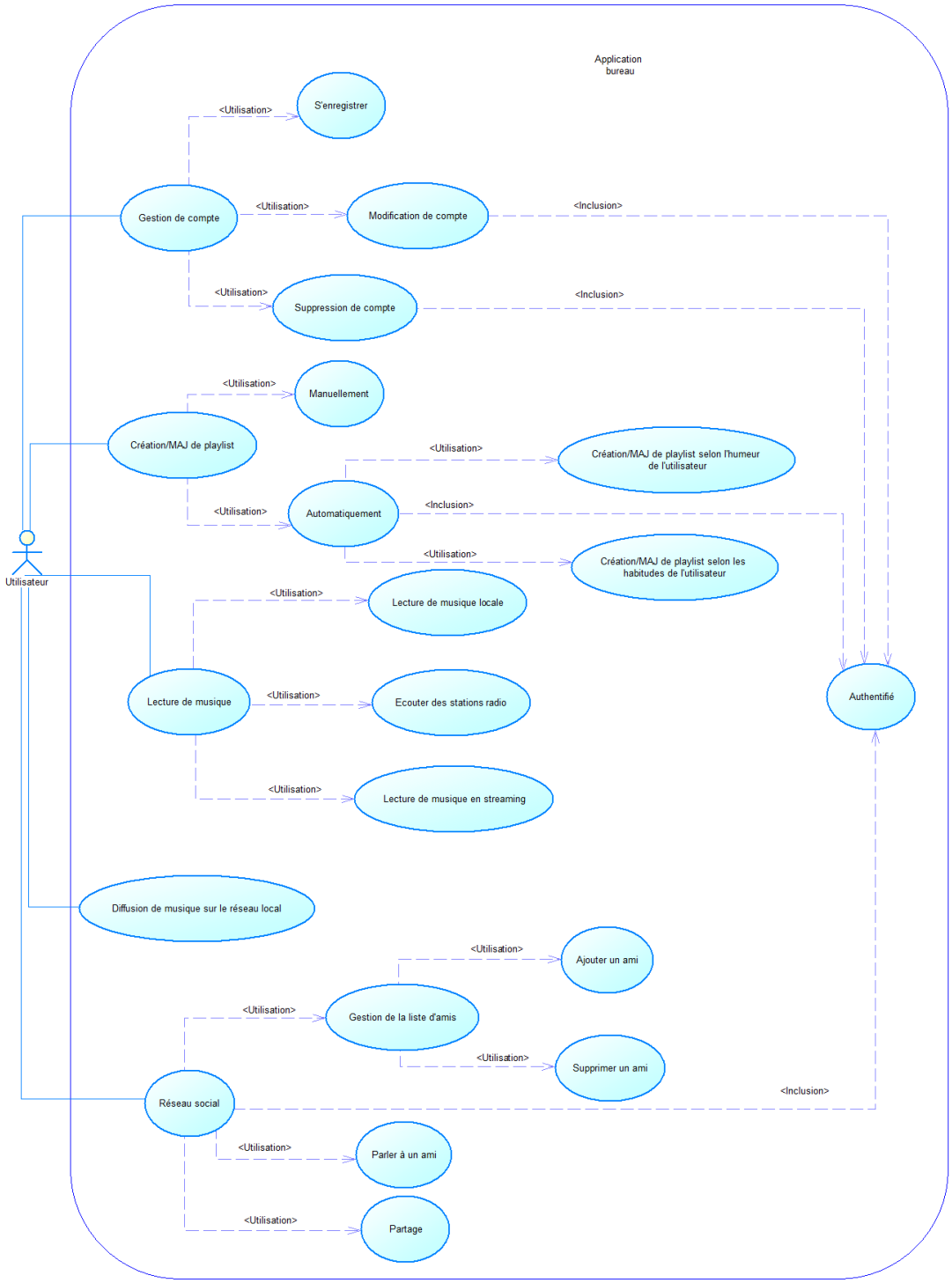


Diagramme de cas d'utilisations du client lourd

b. 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.

Nous allons utiliser AVFoundation pour l'application iOS. AVFoundation est un framework Objective-C (langage utilisé pour le développement iOS) créé par Apple. Ce framework sert essentiellement aux traitements des contenus multimédia. Ce kit de développement a vu jour sous iOS 4 et a été grandement amélioré pour l'iOS 5 et iOS 6.

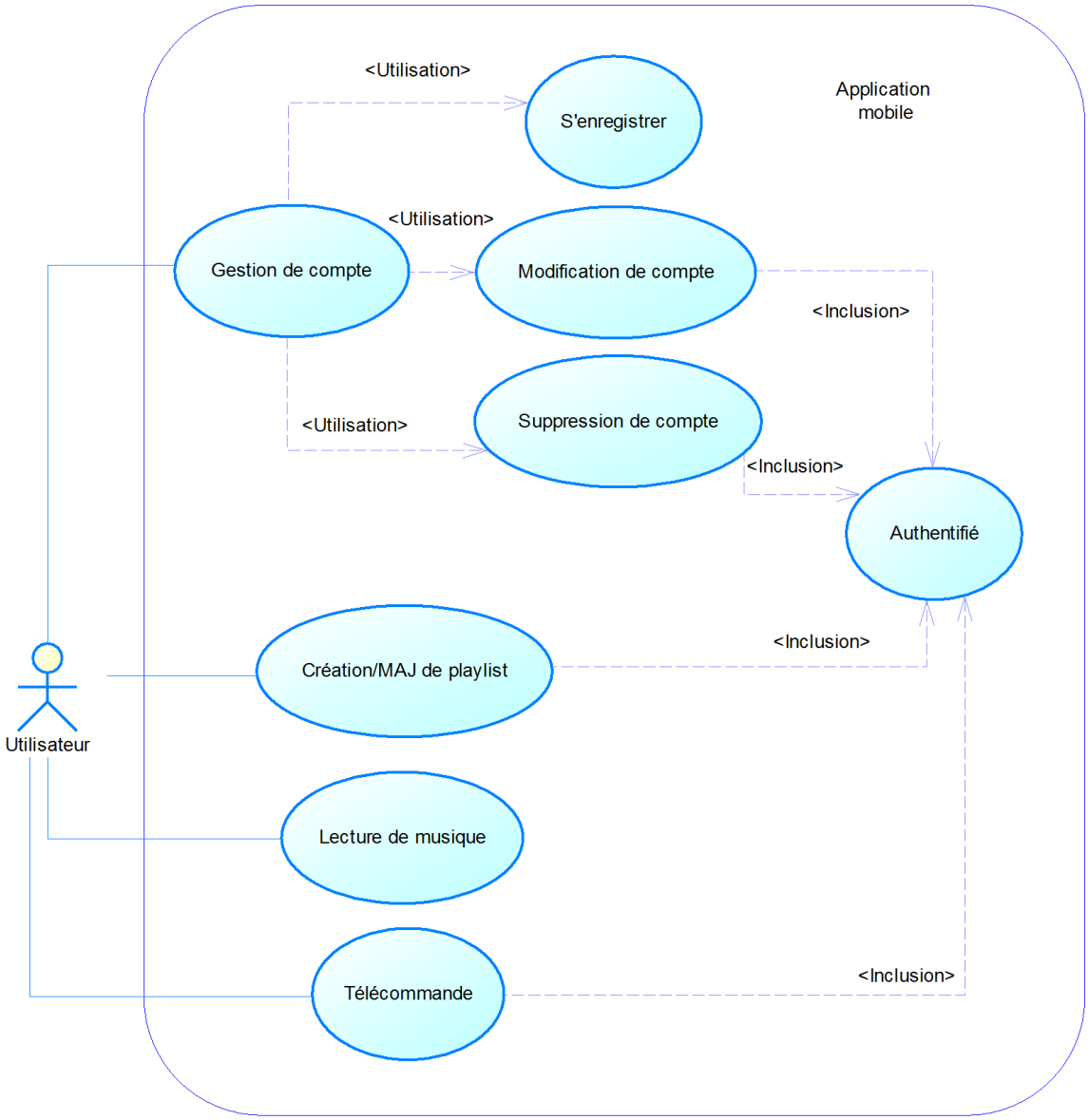


Diagramme de cas d'utilisation des applications mobiles d'AudioWire

3 – Site web

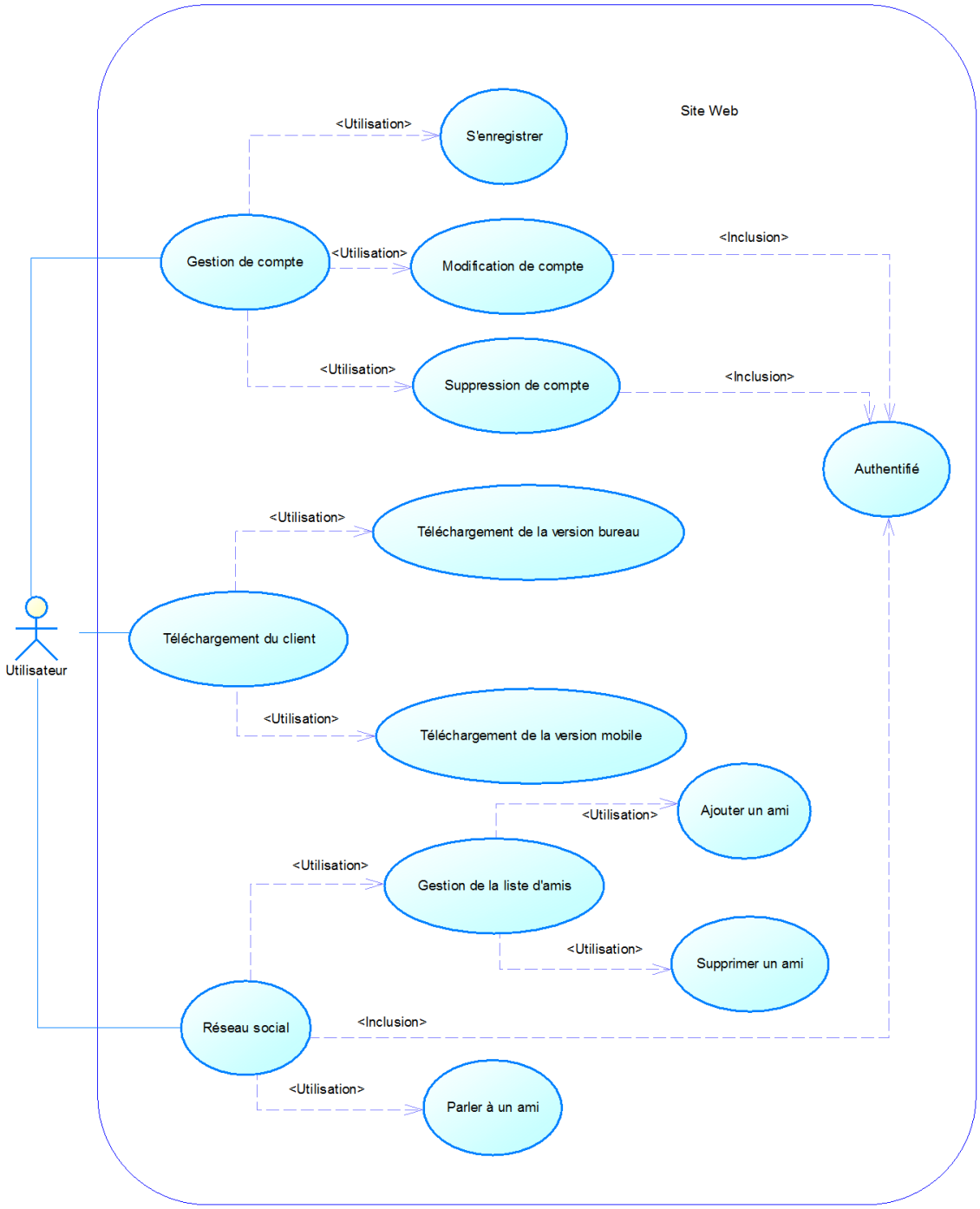


Diagramme de cas d'utilisation de l'application Web

3. Le lecteur audio

Le lecteur audio sera disponible sur toutes les plates-formes connues à savoir Mac, Linux et Windows. Il contiendra non seulement les fonctionnalités classiques d'un lecteur à savoir la gestion de playlist, la gestion de bibliothèque, lecture aléatoire mais également l'intelligence artificielle, la synchronisation musicale et la partie sociale.

Le rôle du client lourd est important car outre son aspect purement technique c'est également lui qui devra attirer l'utilisateur. Il faudra donc que visuellement et ergonomiquement le logiciel soit adapté aux gouts d'un maximum d'utilisateurs.

Les choix techniques

Pour le développement du client, nous avons décidé d'utiliser le c++ pour deux raisons. La première est tout simplement que la majorité d'entre nous ayant étudié du c++ pendant nos années d'études, il nous a paru logique de choisir ce langage. De plus le c++ est un langage objet puissant nous permettant ainsi d'avoir une optimisation des performances non négligeables notamment en ce qui concerne l'intelligence artificielle.

Pour le choix de l'environnement de développement nous avons décidé d'utiliser QT, une API orientée objet qui utilise c++ car cette dernière est entièrement portable et la version (5.1) contient des librairies très pratiques pour le développement de lecteur multimédia.

Les choix stratégiques ont été faits au niveau de l'interface graphique. C'est-à-dire qu'il fallait penser le développement de l'application de manière intuitive et ergonomique. C'est pourquoi nous avons décidé de piocher différentes idées dans des logiciels déjà existants tels que Deezer ou Spotify. En effet ces deux grosses plates-formes de musiques en ligne ont déjà fait leurs preuves et étant donné la période de temps limité dans laquelle le projet doit se construire, il nous a paru plus simple de s'inspirer de ces deux logiciels.

Il faut savoir également que les principaux futurs utilisateurs visé sont en grande partie les amateurs de musique en ligne. C'est pourquoi il faut donc que la période d'adaptabilité d'un logiciel à un autre soit la plus courte possible.

Pour le reste des fonctionnalités tel que la synchronisation musicale et l'intelligence artificielle, nous avons pensé l'interface de l'application en conséquence afin de garantir une utilisation intuitive et efficace.

4. L'IA⁴

A. Rôle de l'intelligence artificielle

L'intelligence artificielle présente dans le projet AudioWire a pour objectif de proposer des musiques à l'utilisateur. Ces musiques devront bien sûr correspondre aux goûts et humeurs de l'utilisateur.

B. Outils utilisés

Afin de d'avoir un maximum de données à analyser pour chaque chanson, nous utiliserons les outils Aubio et Vamp. Ceux-ci permettent d'extraire, pour un fichier audio donné, un grand nombre d'informations telles que le tempo, les fréquences sonores, notes, spectres audio…

L'utilité principale de ces informations sera de nous permettre de classifier les musiques de l'utilisateur. Puis, nous utiliserons ces informations afin de définir ce que l'utilisateur sera susceptible d'apprécier dans d'autres chansons.

C. Description de l'algorithme

a. Entrées

Entrées obtenues par les tags musicaux et par le fichier :

- Genre musical
- Durée
- Métadonnées (artiste, album, titre, année, etc.)

Données générées par les outils AUBIO et VAMP :

- Beat / tempo
- Fréquences sonores (Hz / MIDI)
- Changements harmoniques
- Segments/blocs musicaux similaires
- Différents spectres audio
- Etc.

Entrées de l'algorithme données par l'utilisateur :

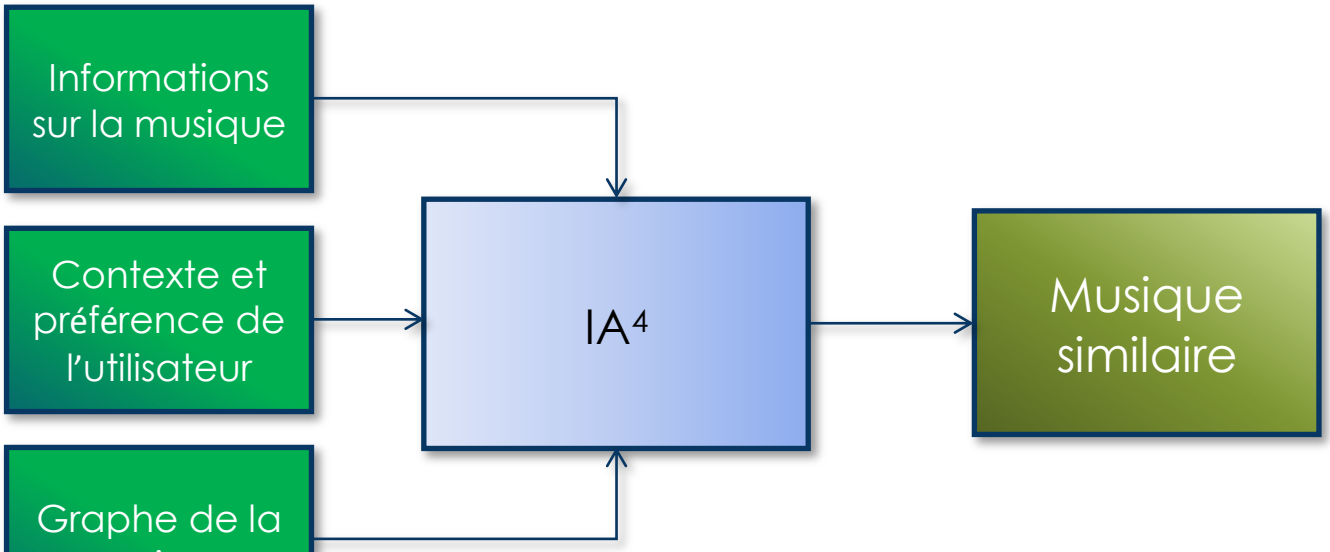
- Humeur
- Contexte (détente, soirée, ménage, etc.)
- Appréciation des musiques écoutées (grâce à 2 boutons : « J'aime » / « J'aime pas »)

b. Fonctionnement

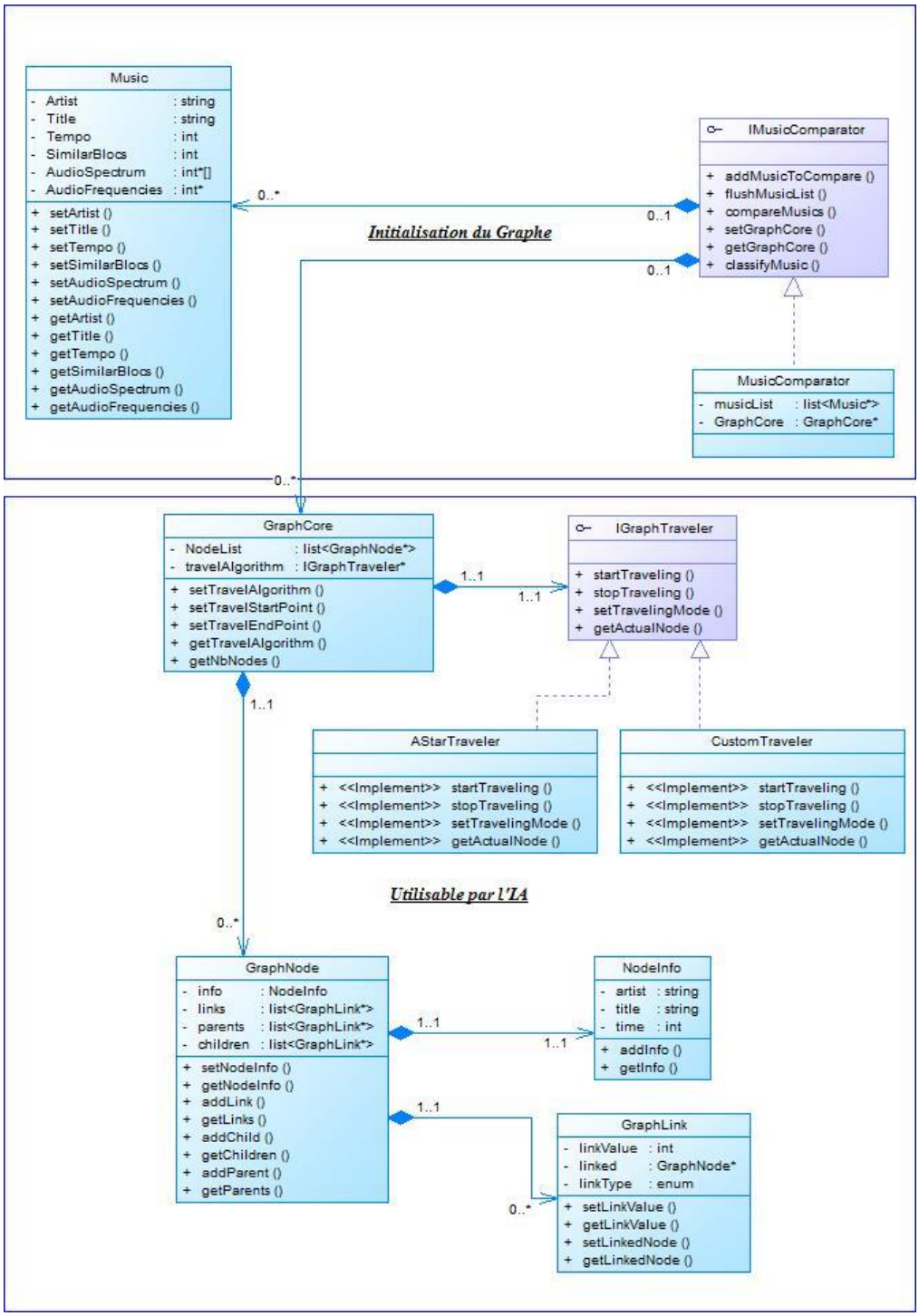
La première étape que devra réaliser l'IA⁴ sera de connecter chaque musique par similarité. Les artistes, genres musicaux, ainsi que les informations extraites telles que le tempo ou encore les fréquences sonores seront utilisées pour remplir cette tâche.

Afin de trouver les musiques qui sont en adéquations avec les goûts de l'utilisateur, l'algorithme mettra en valeur les musiques qui ont des attributs (tempo, fréquences, etc.) semblables à celles que l'utilisateur apprécie.

De plus, celui-ci pourra créer des listes de lecture « contextuelles » dans lesquelles il devra ajouter quelques musiques et ainsi permettre à AudioWire de chercher des sons correspondants.



voir ci-dessous le diagramme de classe du graphe qui sera
es musiques par similarité:


Music
- Artist : string
- Title : string
- Tempo : int
- SimilarBlocs : int
- AudioSpectrum : int*[]
- AudioFrequencies : int*
+ setArtist ()
+ setTitle ()
+ setTempo ()
+ setSimilarBlocs ()
+ setAudioSpectrum ()
+ setAudioFrequencies ()
+ getArtist ()
+ getTitle ()
+ getTempo ()
+ getSimilarBlocs ()
+ getAudioSpectrum ()
+ getAudioFrequencies ()
0..*
0..1
Initialisation du Graphe
IMusicComparator
+ addMusicToCompare ()
+ flushMusicList ()
+ compareMusics ()
+ setGraphCore ()
+ getGraphCore ()
+ classifyMusic ()
0..1
MusicComparator
- musicList : list<Music*>
- GraphCore : GraphCore*
0..*
GraphCore
- NodeList : list<GraphNode*>
- travelAlgorithm : IGraphTraveler*
+ setTravelAlgorithm ()
+ setTravelStartPoint ()
+ setTravelEndPoint ()
+ getTravelAlgorithm ()
+ getNbNodes ()
1..1
1..1
IGraphTraveler
+ startTraveling ()
+ stopTraveling ()
+ setTravelingMode ()
+ getActualNode ()
1..1
AStarTraveler
+ <<Implement>> startTraveling ()
+ <<Implement>> stopTraveling ()
+ <<Implement>> setTravelingMode ()
+ <<Implement>> getActualNode ()
CustomTraveler
+ <<Implement>> startTraveling ()
+ <<Implement>> stopTraveling ()
+ <<Implement>> setTravelingMode ()
+ <<Implement>> getActualNode ()
Utilisable par l'IA
0..*
GraphNode
- info : NodeInfo
- links : list<GraphLink*>
- parents : list<GraphLink*>
- children : list<GraphLink*>
+ setNodeInfo ()
+ getNodeInfo ()
+ addLink ()
+ getLinks ()
+ addChild ()
+ getChildren ()
+ addParent ()
+ getParents ()
1..1
1..1
NodeInfo
- artist : string
- title : string
- time : int
+ addInfo ()
+ getInfo ()
1..1
GraphLink
- linkValue : int
- linked : GraphNode*
- linkType : enum
+ setLinkValue ()
+ getLinkValue ()
+ setLinkedNode ()
+ getLinkedNode ()
0..*

5. Synchronisation

Cette partie explique comment AudioWire permet aux utilisateurs de jouer de la musique sur plusieurs postes dans un même lieu ou d'écouter un morceau avec un contact en même temps grâce à un réseau.

Les synchronisations audio sur réseau local et via internet seront implémentées grâce au Network Time Protocol (NTP). Le NTP permet la synchronisation d'horloges avec des serveurs sur internet. En ajustant les horloges du serveur ainsi que celles des clients sur la même base temps, il nous sera possible de lancer les lectures de fichiers audio simplement, en gardant une latence de moins de 12ms. En effet, après avoir fait des recherches, nous avons trouvé que l'oreille humaine était capable de détecter un décalage entre deux sources audio à partir de 12ms d'écart.

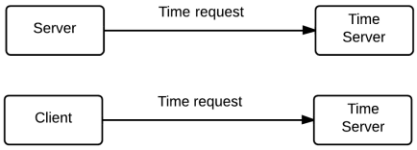
Afin que la fonctionnalité synchronisation soit viable nous avons donc fait le choix d'utiliser le NTP car ce protocole nous permet d'avoir des temps de latence entre plusieurs machines de moins de 10ms.

Pour illustrer le fonctionnement de cette partie, voici un schéma montrant les étapes à réaliser avant de lancer les lectures sur les différents

postes :

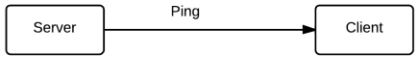
Comment Synchroniser la Lecture Audio

#1



Les horloges du serveur ainsi que des clients sont synchronisées afin d'avoir la même base temps.
Ceci est réalisé avec le Network Time Protocol, qui permet la synchronisation d'horloge avec des serveurs a distance sur internet.

#2



Le server ping les clients cinq fois afin de récuperer le temps de latence le plus haut.

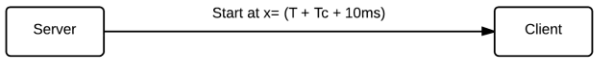
#3

L / 2 + 10ms = Tc

Where L = Highest latency
Tc = Time to client

Le serveur calcule le temps qu'il faut a une commande pour qu'elle soit executée basé sur le temps de latence.
L est divisé par deux car la commande ping inclut le temps de réponse au serveur. Il faut donc moitié moins de temps pour envoyer une instruction.

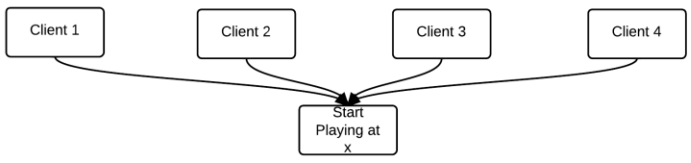
#4



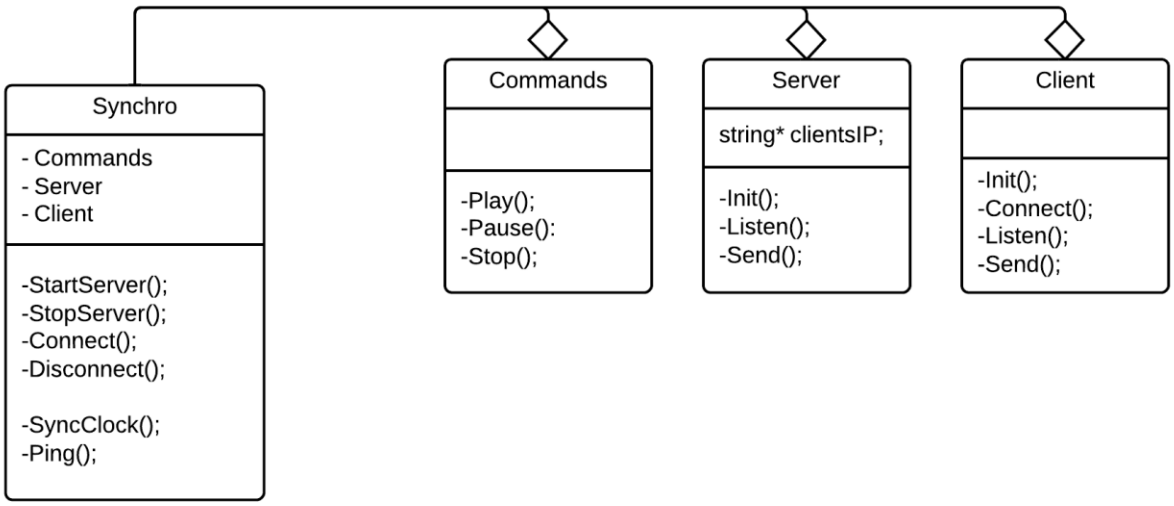
Where T = Time
Tc = Time to client

Le serveur demande au client de commencer la lecture à un temps définit (x) et ajoute le temps qu'il faut à la commande pour atteindre le client. Nous ajoutons 10ms par mesure de précaution.

#4



Ci-dessous le schéma UML de la partie synchronisation.
La classe "Synchro" sera présente de la partie principale de notre application de bureau.



6. Architecture, buts et contraintes techniques

A. Contraintes techniques

a. Puissance des téléphones mobiles

La majorité des téléphones portables d'aujourd'hui ne peuvent pas rivaliser en termes 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.).

b. Bande Passante

Le procédé de streaming implique une 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 satisfaisants, le logiciel préviendra l'utilisateur et le streaming sera automatiquement désactivé.

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 une évolutivité simplifiée des ressources. 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.

B. Communication

Voici ci-dessous quelques précisions sur les méthodes de communication entre les différentes parties du projet, aidées d'un diagramme de communication.

Les différentes parties sont :

- Un serveur principal qui contiendra le cerveau de notre API[1] développée en REST[2]. Cette API[1] utilisera le protocole HTTPS afin de garantir un maximum de flexibilité et de simplicité dans son utilisation. C'est par cet unique serveur que toutes les applications du projet devront passer pour communiquer. Il contiendra aussi une intelligence artificielle allégée pour gérer les musiques des applications mobiles.
- Un serveur web, écris en NodeJS, communique grâce aux protocoles HTTP, HTTPS et Websocket avec les clients web. Le serveur web communiquera avec le client lourd, via D-BUS. D-BUS est un logiciel de communication interprocessus permettant à des applications de communiquer entre elles.
- Une base de données relationnelle contenant les informations des utilisateurs et des musiques traitées par l'IA[4] contenus directement dans le serveur principal. Etant interne au serveur contenant l'API[1], elle répondra directement aux requêtes du serveur. Nous recommandons d'utilisation de MySQL, bien que toutes les bases de données relationnelles modernes soient supportées.
- Une base de données de musiques qui sera le support de toutes nos applications et de notre IA[4]. Cette dernière est clonée depuis une base de données libre de droits, MusicBrainz[5]. Cette base de données sera basée sur MySQL, dont les requêtes communiqueront directement avec la partie IA[4] du projet, contenue dans AudioWire.
- Deux applications mobiles, Android et iOS. Un outil tel que Titanium[6] sera utilisé afin de pouvoir développer une application qui sera portable sur les deux systèmes. Le protocole HTTPS sera utilisé pour communiquer avec le serveur principal.

- Un client dit « lourd » PC / MAC, qui devra, tout comme les autres parties du projet, passer par l'API[1] (via le protocole HTTPS) afin d'effectuer des modifications de statut par exemple, ou des opérations sur les musiques, etc. Il contiendra l'intelligence artificielle, qui sera intégrée dans le lecteur audio libre de droits. Il pourra communiquer avec le serveur web grâce à D-BUS.

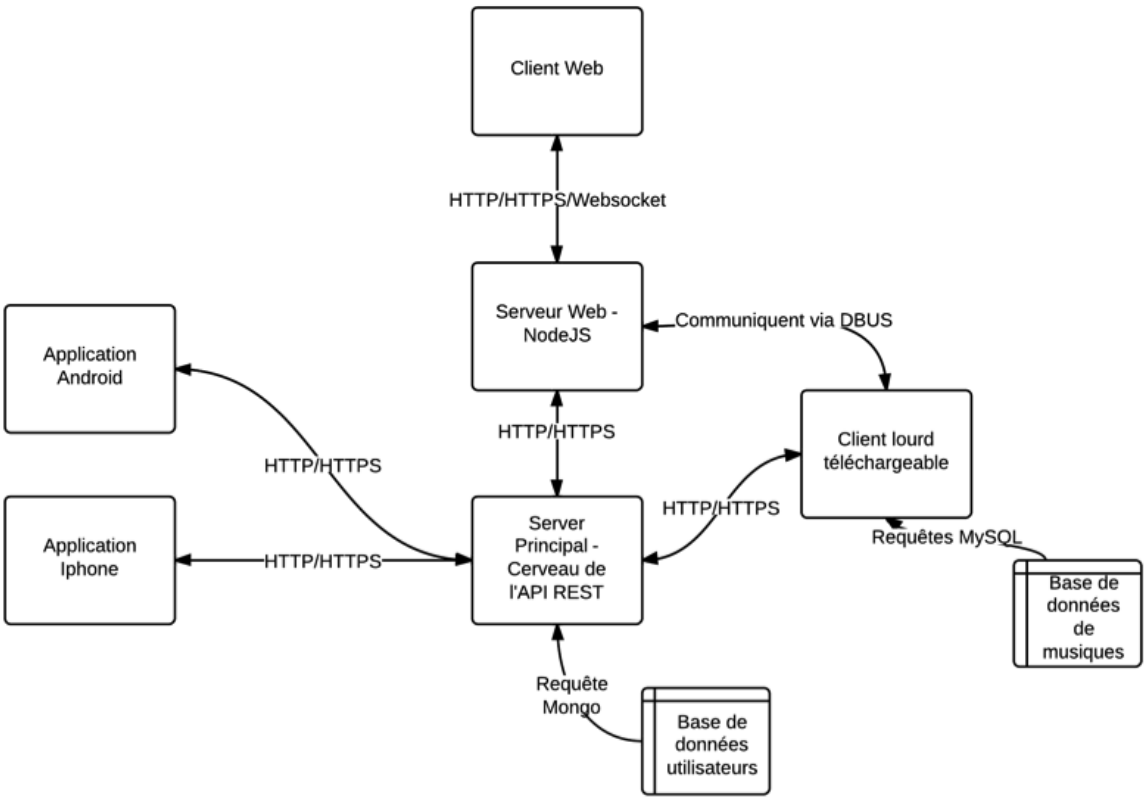


Diagramme de communication d'AudioWire

C. Vue physique de l'application

Dans cette partie nous allons présenter la partie physique d'AudioWire ou, du moins, ce à quoi le projet devrait ressembler une fois terminé d'un point de vue communication et composante physique. Il faut savoir qu'il est encore difficile à ce stade du développement d'établir les protocoles de communication avec précision. C'est pourquoi certains termes peuvent encore paraitre trop génériques ou des protocoles trop peu détaillés. Cependant nous avons essayé d'être le plus clair et précis possible malgré ces quelques contraintes.

D'après le diagramme de déploiement ci-dessous, nous pouvons voir que le projet AudioWire sera principalement composé de cinq parties distinctes. A savoir, l'application de bureau, le serveur principal, l'application Web, le serveur Web et les applications mobiles qui pour l'instant ne forment qu'une seule et même partie car nous ne sommes pas encore en mesure de vraiment développer cette partie précisément (CF : diagramme de Gantt).

a. L'application de bureau

L'application bureautique (« applications lourde ») regroupe les clients sur les plateformes Windows, Mac OS X, et Linux. Ce client sera le client le plus développé et contiendra l'intégralité des caractéristiques d'AudioWire. Il contiendra notamment les fonctionnalités d'intelligence artificielle (qui nous permettront d'adapter la musique aux goûts des utilisateurs) et la gestion du compte utilisateur. La lecture simultanée sera directement implémentée côté client avec le Network Time Protocol (NTP) pour avoir un meilleur résultat synchronisé. En d'autres termes, cela signifie que la communication pour l'écoute synchronisée utilisera son propre protocole de communication et n'interfèrera pas avec les autres protocoles de communications.

b. Le serveur principal

Le serveur principal nous servira à héberger à la fois la base de données pour les comptes utilisateurs, la base de stockage

pour les métadonnées et les spectres des différents morceaux de musique. Il contiendra également une Intelligence Artificielle plus petite pour la gestion des applications mobiles. La communication entre le serveur et les supports externes sera effectuée grâce à l'API[1].

c. L'application Web

L'application web, qui est la toute nouvelle fonctionnalité d'AudioWire, sera disponible sur tous les navigateurs modernes. Ses deux principales fonctionnalités seront la fonction de télécommande qui contrôlera des clients lourds et la partie sociale du projet (consultation de son profil, gérer ses listes d'amis, etc.). Il faut savoir que la partie sociale d'AudioWire n'est pas encore totalement mise au point car ce n'est pas la priorité à ce stade du projet.

La communication entre l'application Web et le client lourd sera effectuée grâce à D-BUS qui est un logiciel de communication informatique permettant la communication interprocessus.

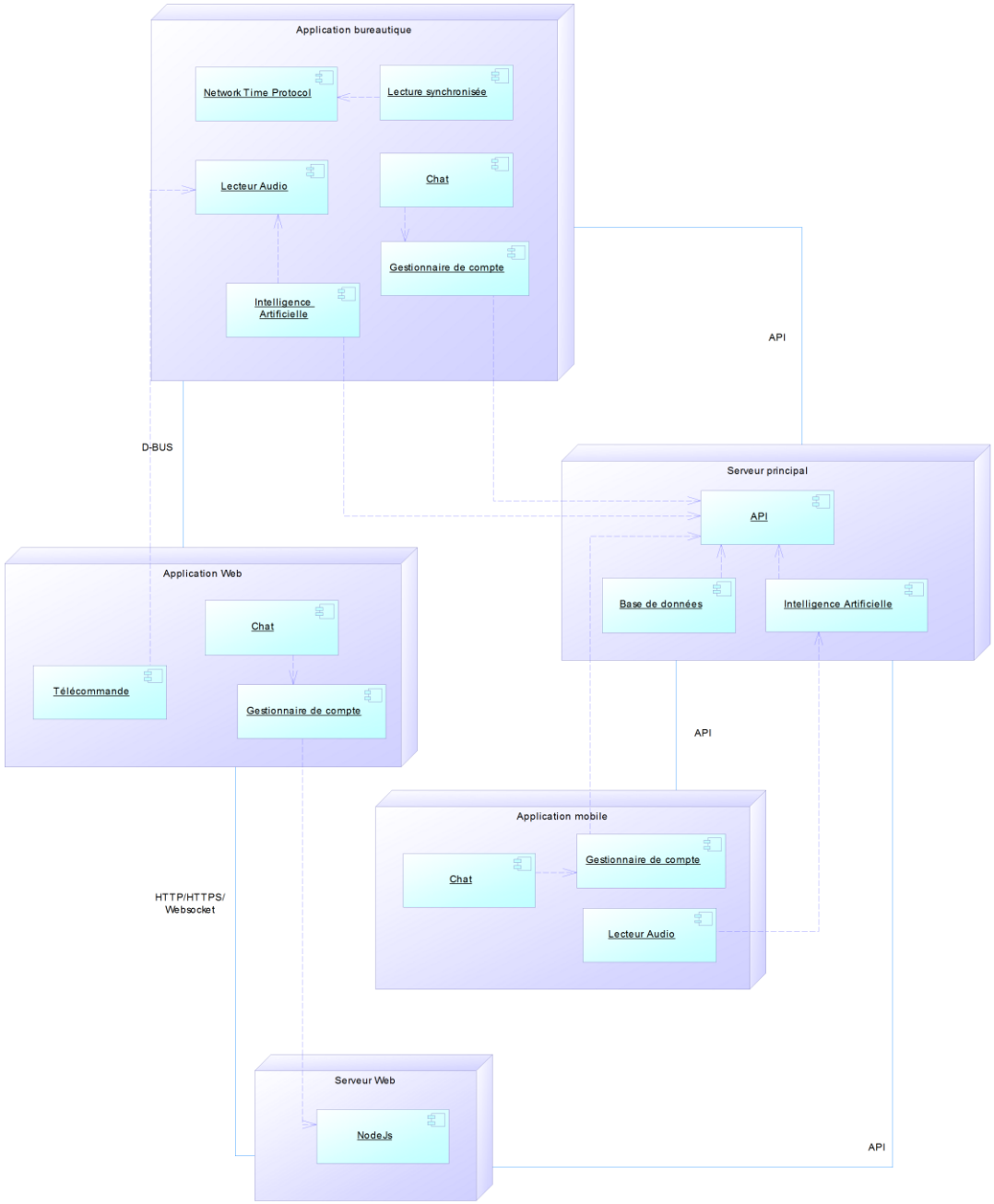
d. Le serveur web

Le serveur web, quant à lui, sera écris en Node.js, un Framework événementiel permettant d'écrire des applications réseaux en JavaScript. Le serveur web servira donc l'application web. La communication entre le serveur web et l'application web sera effectuée grâce au protocole HTTP/HTTPS et Websocket.

e. L'application mobile

Comme annoncé précédemment, les téléphones communiqueront avec le serveur principal grâce à l'API[1]. Ses

applications mobiles contiendront les fonctionnalités classiques d'un lecteur audio que l'on peut trouver sur les mobiles aujourd'hui ainsi qu'une intelligence artificielle allégée qui fonctionnera au travers de l'API[1].



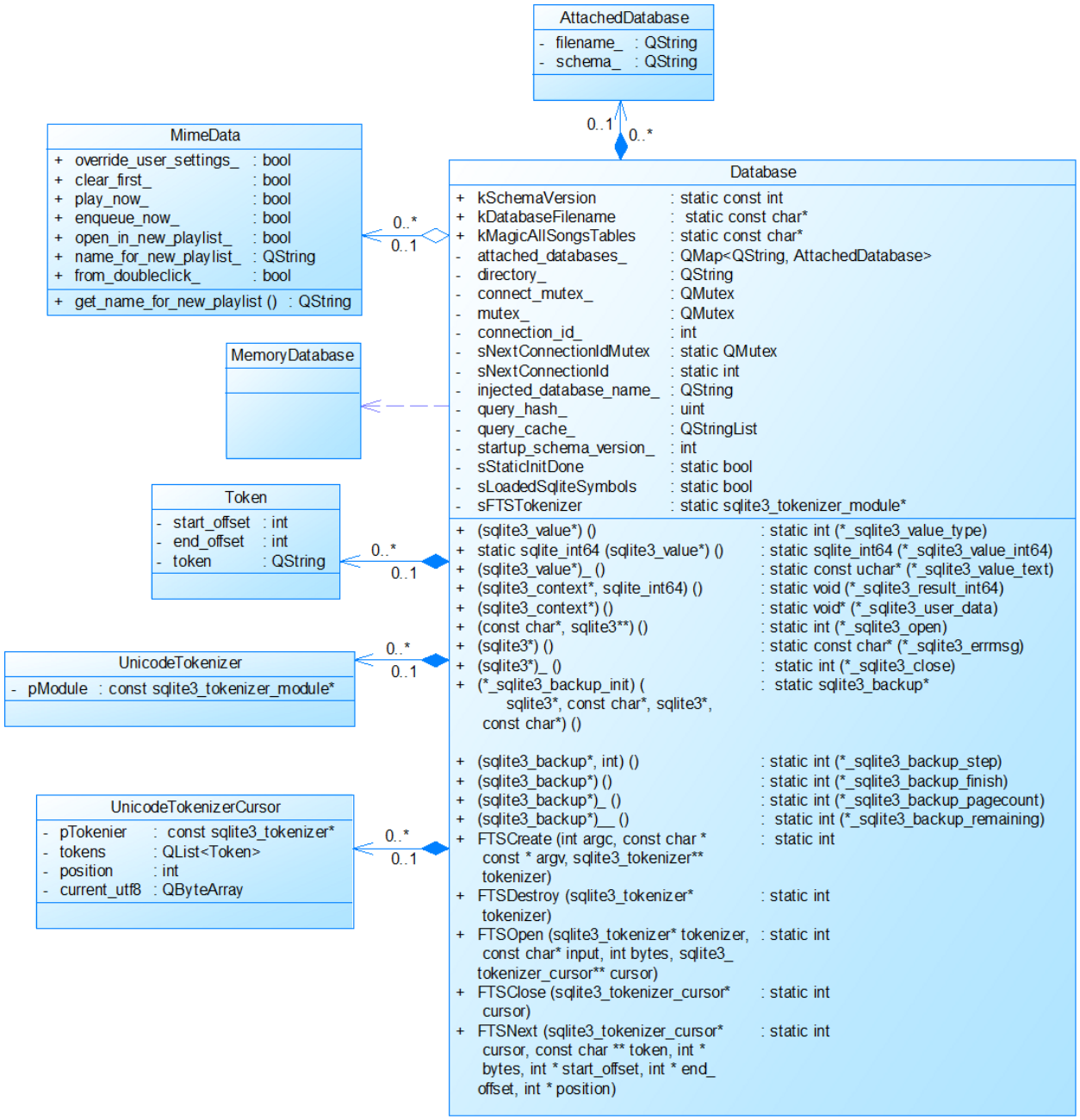
8. Implémentation du client

Le développement s'étend sur deux points principaux : le développement du lecteur audio d'un côté et sur l'intelligence artificielle et les fonctionnalités associées de l'autre.

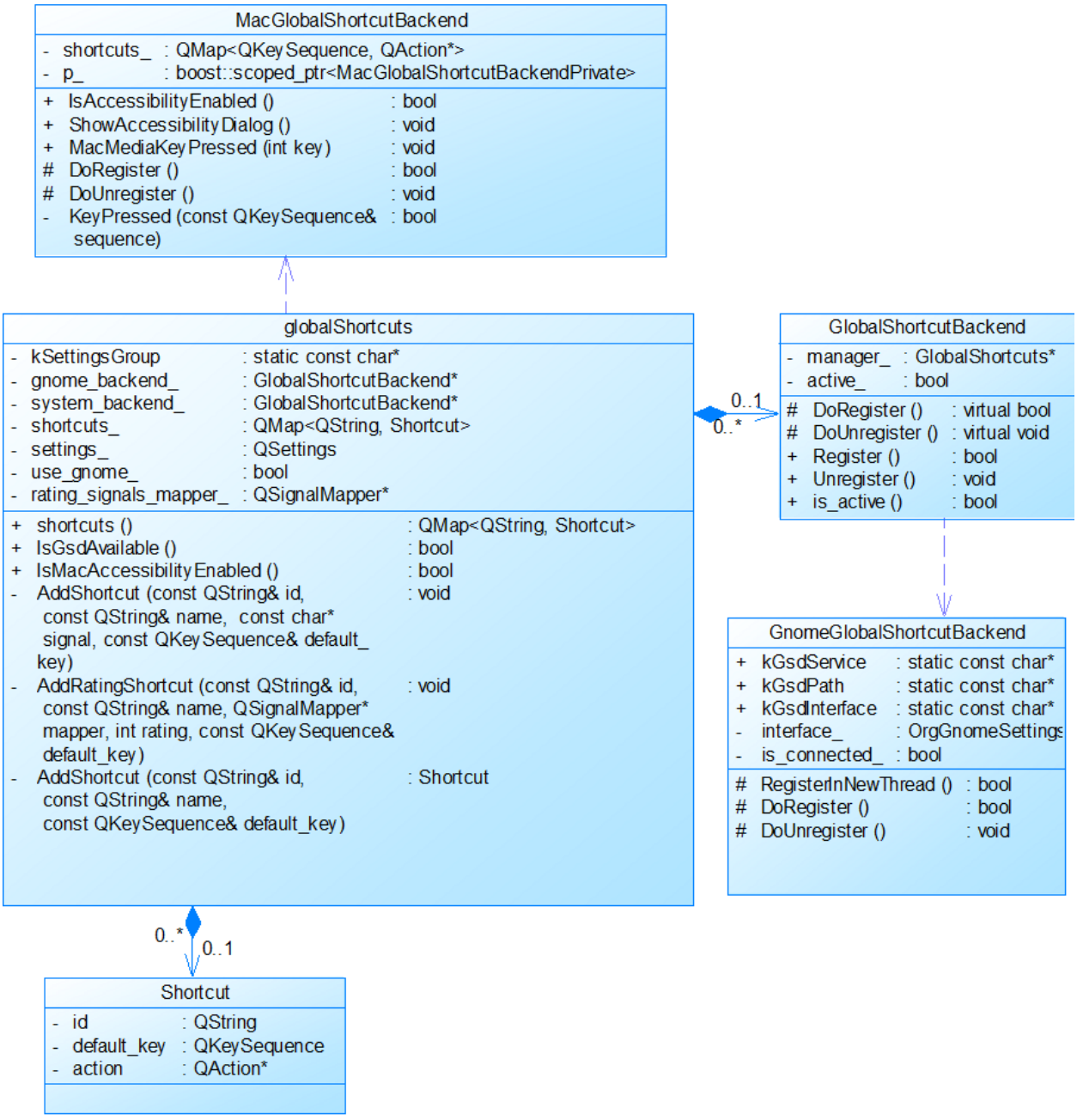
Pour le moment nous sommes partis sur un design pattern «Modèle-Vue-Contrôleur » (MVC) classique car c'est l'une des techniques les plus utilisées pour ce type d'application.

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 cette méthode. Les schémas qui vont suivre sont les parties essentielles que nous avons reprises pour le développement d'AudioWire et du design MVC.

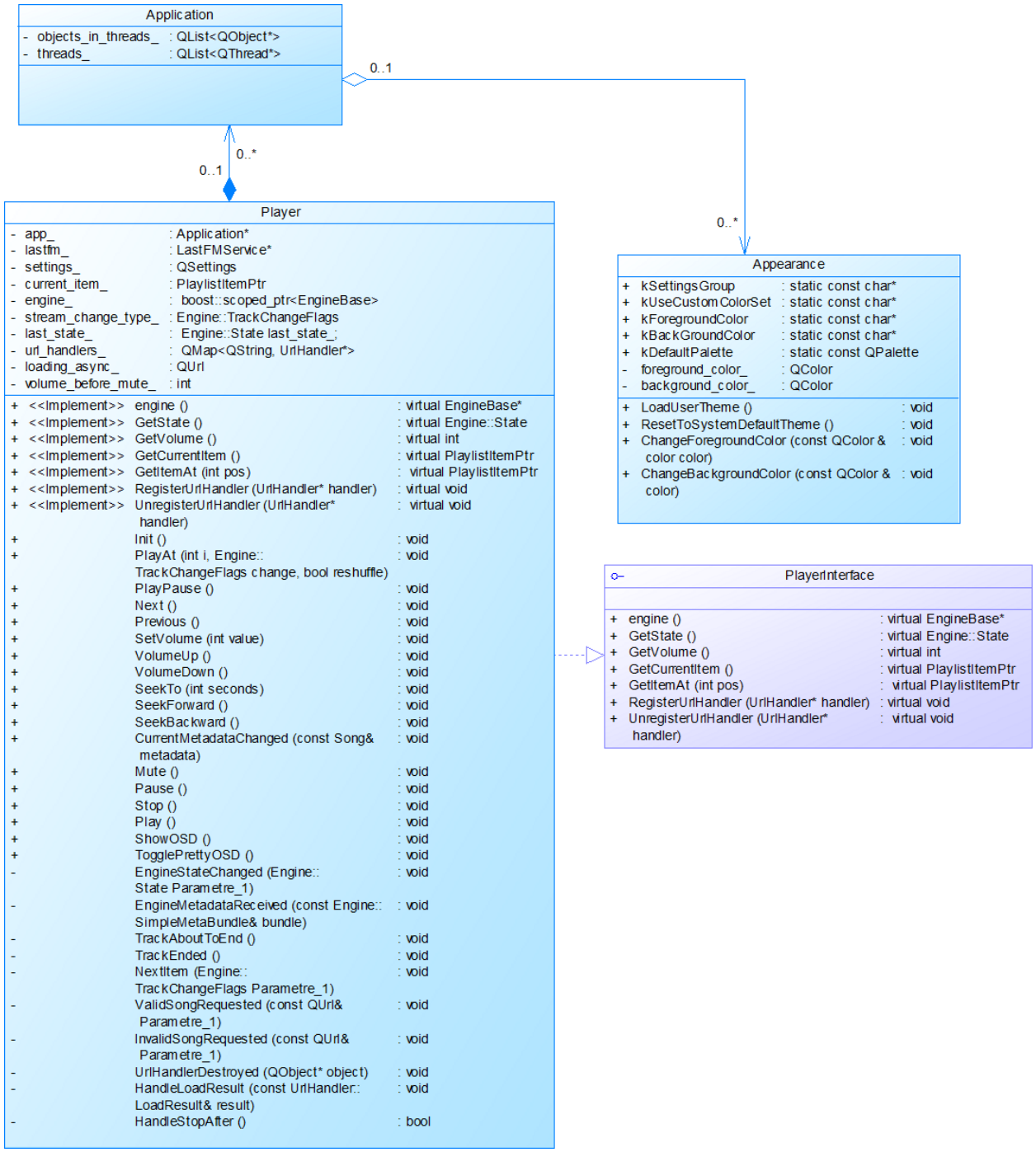
Voici le diagramme de classe de la partie base de données de d'AudioWire :



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 systèmes 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.


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