|  |  |
| --- | --- |
|  | **2015-16** |
|  | HEIG-VD, section TIC  Auteurs: Baehler Simon,  Moret Jérôme,  Purro Jan,  Berney Léonard &  Roubaty Anthony  Date du rendu finale du projet : 4 janvier 2016  Professeur: René Rentsch |

C:\Git\PDG_Project\Rapport\img\logoHeigVd.png

|  |
| --- |
| **[Documentation Flat5]** |
| Rapport du projet de semestre nommé Flat5 |



Table des matières

[1 Introduction 2](#_Toc434240037)

[1.1 Choix de projet 2](#_Toc434240038)

[1.2 Objectif de Flat 5 2](#_Toc434240039)

[1.3 Utilisation de l’application 2](#_Toc434240040)

[2 Mise en place 3](#_Toc434240041)

[2.1 Gestion des fichiers 3](#_Toc434240042)

[3 Analyse 3](#_Toc434240043)

[3.1 Fonctionnement de base 3](#_Toc434240044)

[3.2 Base de données 3](#_Toc434240045)

[3.3 Lecteur vidéo 3](#_Toc434240046)

[3.4 Lecteur audio 3](#_Toc434240047)

[3.5 API web 3](#_Toc434240048)

[3.6 Interface 4](#_Toc434240049)

[4 Réalisation 5](#_Toc434240050)

[4.1 Lecteur vidéo 5](#_Toc434240051)

[4.2 Lecteur audio 5](#_Toc434240052)

[4.3 API web 5](#_Toc434240053)

[4.4 Synchronisation 5](#_Toc434240054)

[5 Points non-réalisés 5](#_Toc434240055)

[6 Problèmes connus 5](#_Toc434240056)

[7 Planification 5](#_Toc434240057)

[7.1 Planification initial 5](#_Toc434240058)

[7.2 Re-planification 5](#_Toc434240059)

[8 Conclusion 5](#_Toc434240060)

[9 Table des illustrations 5](#_Toc434240061)

[10 Signatures 6](#_Toc434240062)

[11 Annexes 8](#_Toc434240063)

[11.1 Journal de travail 8](#_Toc434240064)

[11.2 Cahier des charges 8](#_Toc434240065)

[11.3 Manuel d’utilisation 8](#_Toc434240066)

# Introduction

Le but de ce projet est de concevoir et de développer un logiciel fonctionnel de toute pièce, et par groupe devant être composé de 4 ou 5 personnes constitué aléatoirement. Cela permet de mieux comprendre le travail en équipe et l'importance d'avoir une bonne coopération de chaque collaborateur au sein d'un projet.

Pour la réalisation d'un tel projet, certains points ont dû être mis en place tels que:

* Une planification raisonnable du projet.
* Découpe efficace des différentes tâches.
* Transmettre et communiquer correctement les informations au sein du groupe.

## Choix de projet

A la différence avec PRO, dans le cours PDG nous n’avons pas de restriction de langage. La seul restriction imposé c’est le fait de ne pas pouvoir faire de jeux, autrement nous avons carte blanche.

A partir de cela, et après un moment de réflexion une idée nous est venue. Nous voulons créer un programme permettant à deux, puit à plusieurs personnes de pouvoir regarder un film ou écouter de la musique « comme si il était à côté ».C’est-à-dire, en ayant un système de synchronisation de média.

En poussant un peu le concept, nous sommes arrivés à une idée générale regroupant, justement, ce concept de synchronisation avec une bibliothèque de médias. Notre application proposera alors une gestion facilité de divers médias (musiques, films, séries), tout en ajoutant une fonctionnalité de synchronisation. Cette dernière va justement permettre d’envoyer les commandes qui sont effectué sur un lecteur aux autres. Par exemple, il y a deux personnes qui regarde un film ensemble, et que l’une d’entre elle appuie sur « pause », alors cela mettra pause aussi chez l’autre personne.

En termes de technologie, les différents lecteurs (sons et vidéos) seront entièrement créés par nous. Nous allons coder notre application en Java avec une interface graphique en JavaFX.

**Remarque**: Les choix des outils utilisés sont expliqués plus en détails dans les chapitres de ce document.

## Objectif de Flat 5

Proposé une interface facile à prendre en main pour gérer efficacement une collection de média audiovisuel. Tant en proposant la possibilité de partager son expérience en temps réel avec d’autres personnes.

## Utilisation de l’application

Flat 5 s’adresse à toutes les personnes voulant gérer et partager lors médias.

Pour plus d’informations concernant l’utilisation de l’application, référez-vous au manuel d’utilisation.

# Mise en place

Pour pouvoir réaliser tout projet, il est préférable de définir certains points et règles afin que le déroulement du projet se fasse de la manière la plus cadrée et optimale possible. Nous avons donc séparé les différentes tâches.

* Interface : il s’agit d’un des points central de notre application. Elle sera réalisé M. Baehler Simon.
* Lecteur audio : cela prend en compte la gestion des différents événements du lecteur (play, pause, etc…). Il sera réalisé par M. Moret Jérôme.
* Lecteur vidéo : Même chose que le lecteur audio seulement pour les films et les séries. Il sera réalisé par M. Berney Léonard.
* Synchronisation : la synchronisation fera appel à une partie réseau, et surtout la mise en place d’un protocole sommaire. Elle sera réalisée par M. Roubaty Anthony.
* Base de données : elle permettra de stocker diverses informations de configuration, et aussi de stocker les informations relatives à la bibliothèque de médias. Elle sera réalisée par M. Purro Jan
* API audio : elle permettra de récupérer diverses informations sur les morceaux de musique. Elle sera réalisée par M. Roubaty Anthony
* API vidéo : idem que pour l’API audio, mais adapté pour les films et les séries. Elle sera réalisée par M. Purro Jan.

Si deux personnes ont 2 tâches différentes à réaliser, cela vient du fait que la somme de travail varie d’une tâche à l’autre. Certaines tâches, comme la rédaction de la documentation, sont considérer comme une tâche commune, il est du devoir de la personne responsable d’une tâche de documenter son travail dans le rapport.

## Gestion des fichiers

Pour pouvoir travailler le plus efficacement possible au cours de projet, nous avons utilisé la plateforme Github. Cette dernière nous permet de garder une trace de tous les changements, autant au niveau du code que du rapport.

Le répertoire Git étant public, il est accessible à cette adresse.

<https://github.com/snup482/PDG_Project>

# Analyse

Au cours de ce chapitre, nous allons expliquer les différentes recherches qui ont été mené. Cela afin de mieux comprendre les choix finaux.

## Fonctionnement de base

## Base de données

## Lecteur vidéo

## Lecteur audio

Pour concevoir un lecteur audio, il est nécessaire d’utiliser un framework efficace permettant la lecture d’un grand nombre de formats.

Nous avons décidé d’utiliser **vlcj** qui est un framework Java offrant l’intégration d’une instance de VLC dans notre projet. L’avantage de cette approche est que nous découplons la partie visuelle de la partie fonctionnelle. Nous codons notre propre fenêtre pour le lecteur, celle-ci utilisant les services de VLC pour l’étape finale qu’est la lecture du média. Rappelons que VLC supporte, pour la musique, les formats suivants : MPEG Layer ½, **MP3 - MPEG Layer 3**, AAC - MPEG-4 part3, Vorbis, AC3 - A/52 (Dolby Digital), E-AC-3 (Dolby Digital Plus), MLP / TrueHD">3, DTS, WMA ½, WMA 3, **FLAC**, ALAC, Speex, Musepack / MPC, ATRAC 3, **Wavpack**, Mod (.s3m, .it, .mod), TrueAudio (TTA), APE (Monkey Audio), Real Audio, Alaw/µlaw, AMR (3GPP), MIDI, LPCM, ADPCM, QCELP, DV Audio, QDM2/QDMC (QuickTime), MACE.

L’instance de VLC est jointe dans les ressources de notre projet. Nous nous sommes basé sur une instance 64 bits car nous utilisons la JDK 1.8 64 bits de Java pour la compilation de notre projet.

D’une fois que nous avons notre partie fonctionnelle, nous pouvons passer à la partie visuelle. Généralement en Java il est difficile de réaliser une belle interface car on doit bien souvent s’appuyer sur la bibliothèque **Swing** ou **AWT** qui peine à sortir un design actuel.

Pour ce projet, nous nous sommes lancé le défi de réaliser un design flat et pour cela nous avons utilisé **JavaFX,** nouvelle technologie d’Oracle promettant du renouveau en termes d’interface.

Pour la conception d’un lecteur audio, il suffit d’utiliser une liste, des boutons Play / Suivant / Précédent et une barre de progression.

## API web

Les APIs web sont utilisées dans l’application pour retrouver les différentes informations sur les médias exporté. Il faudra en utilisé deux différentes, une pour les films, et une pour les musiques.

Pour les films, TODO FILM API

Et maintenant concernant l’API choisi pour les musiques, l’idée de base été de pouvoir reconnaître un fichier audio juste le lisant. Nous nous sommes donc penchés sur des algorithmes d’ « acoustic fingerprint ».

Le principe de cet algorithme est assez simple. Cela consiste à détecter, sur base d’un échantillon, les différentes caractérise d’un morceau (tempo, etc…), et ensuite de recherche dans une base de données si cela correspond à un fichier déjà connu. Le problème principal de cette méthode c’est justement la dernière partie. Car posséder une telle base de données n’est pas une chose facile.

Après plusieurs heures recherches, nous avons trouvé différents outils.

**Musicbrainz** : site mettant à disposition un large catalogue gratuit de musique. Mais attention ce catalogue ne contient que les informations textes, et la pochette. De plus, il ne propose pas un moyen de reconnaître un fichier audio.

**Echonest** : Entreprise mettant à disposition différent outils concernent les fichiers audio. Notamment un système de « fingerprint ». Malheureusement, ces dernières ne mettent pas leur base de données à disposition gratuitement (par contre la librairie est disponible gratuitement).

**Audiotag** : site permettant de faire une reconnaissance en ligne de musique. Malheureusement il n’existe pas d’API.

En d’autres termes, vous l’aurez compris, il n’existe actuellement pas d’entreprise mettant à disposition une telle base de données gratuitement.

Il faut maintenant passer au plan B, c’est-à-dire, reconnaître un fichier audio grâce à ses métadonnées.

Pour faire cela, il faut pouvoir les lire. A première vue, cela semble facile, mais elle dépende du format du fichier. Et comme nous avons dit, dans le cahier des charges, que nous supportons plusieurs types de format, il faut être capable de tout lire sans avoir une librairie par format.

Pour réussir cela, nous avons utilisé la librairie jaudiotagger qui permet de lire les métadonnées de tous les formats qui nous intéresse.

A partir de là, il nous faut encore un moyen de récupérer les informations d’une musique. Pour cela, il existe une API, qui a déjà fait ces preuves, l’API de Spotify. Il faut juste trouver un moyen de l’intégrer efficacement dans notre application.

Après un peu de recherche, nous sommes tombé sur la repertoire Github suivant : <https://github.com/thelinmichael/spotify-web-api-java>

Ce dernier met à disposition une librairie pour faciliter l’utilisateur de l’API, pas d’appel HTTP à faire.

Donc si on résume, nous allons utiliser jaudiotagger pour lire les différentes métadonnées des fichiers audio, et ensuite, nous allons utiliser l’API Spotify (à travers la librairie trouvée) pour récupérer toutes les informations qui nous intéresse (pochette album, etc…).

## Interface

### Introduction

L'interface a été réalisé au moyen de la librairie graphique FXML et grâce au logiciel « SceneBuilder » pour une construction WYSIWYG de nos vues. L'avantage principal de faire nos interfaces graphiques avec FXML et SceneBuiler est que nous allons utiliser du css (css légèrement différent que celui qu'on utilise pour les Web), nous allons donc pouvoir réaliser des interfaces relativement facilement, et qui plus est, des interfaces esthétiquement bonnes. Le problème en revanche avec FXML, est qu'il est relativement jeun et certaines fonctionnalités ne sont pas disponible ou mal fonctionnelle, notamment la lecture de vidéo qui a dû, elle être faite en Swing.

Dans cette partie nous allons détailler chaque vue du programme ainsi que le comparer avec les mockups réalisés sur Photoshop au début du projet.

D'un point de vue global, l'interface n'a que très peu changée, elle est restée très proche de ce qui avait été désigné au début. La plus part des modifications apportées dans le but de simplifier l'interface et d'éviter la redondance, c'est points seront détaillé plus bas quand nous aborderons les comparaisons des vue une à une.

### Accueil

Nous allons commencer par détailler la vue d'accueil de notre programme, la première image représente l'interface désignée sur Photoshop, la seconde l'interface réelle. Globalement les deux vue sont similaire, dans les deux cas nous avons notre menu en haut à gauche (nous l'appellerons root) ce menu sera présent tout au long de la navigation. Nous avons également le menu d'accueil qui reprend le menu du haut sans le bouton d'accès à l'accueil. L'unique différence entre l'interface fait sur Photoshop et celle réalisé est le style css des boutons. La raisons à cela est dû au fait que nous avions pris des css déjà existant pour ne pas nous attardé sur quelque chose de si futile, de plus les css trouvés allaient bien avec notre interface.



Figure : MockUp de l'accueil

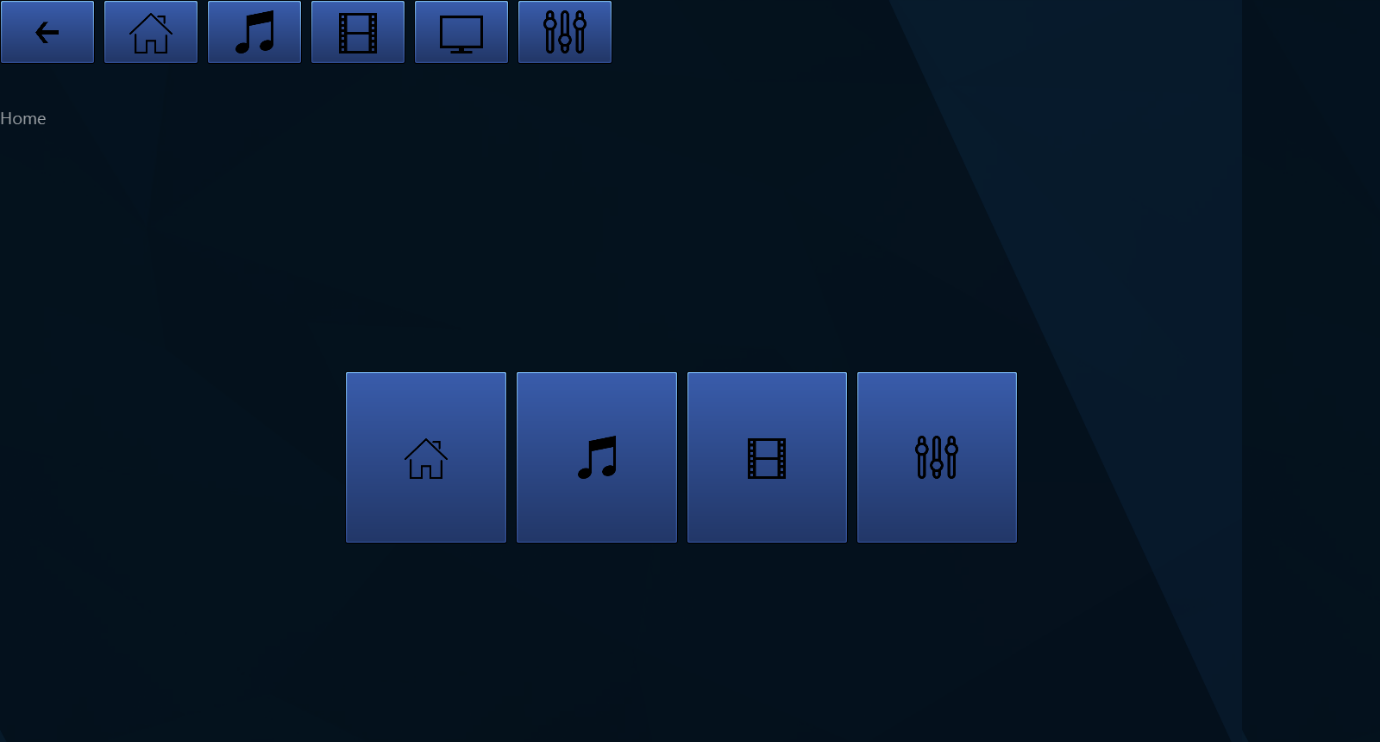


Figure : Réalisation de l'accueil

### Vue film

Nous allons désormais passer à la vue des films, cette vue a été considérablement simplifié, lors de la réalisation sur Photoshop nous avions eu les yeux plus grand que le ventre et nous avons réalisé une maquette avec une interface relativement poussée, nous nous sommes très vite rendu compte qu'une telle interface allait être trop chronophage et nous avons donc dû faire des concessions et réalisé une interface plus simple. Nous avons opté pour un simple tableau listant les différents films.



Figure : MockUp vue film



Figure : vue des films

### Vue film détaillée

Pour la vue détaillée pour les films nous avons fait quelque chose de relativement similaire à ce que nous avions fait comme maquette, nous avons simplement supprimé les onglets casting et critiques. Nous les avons jugés peu pertinentes.

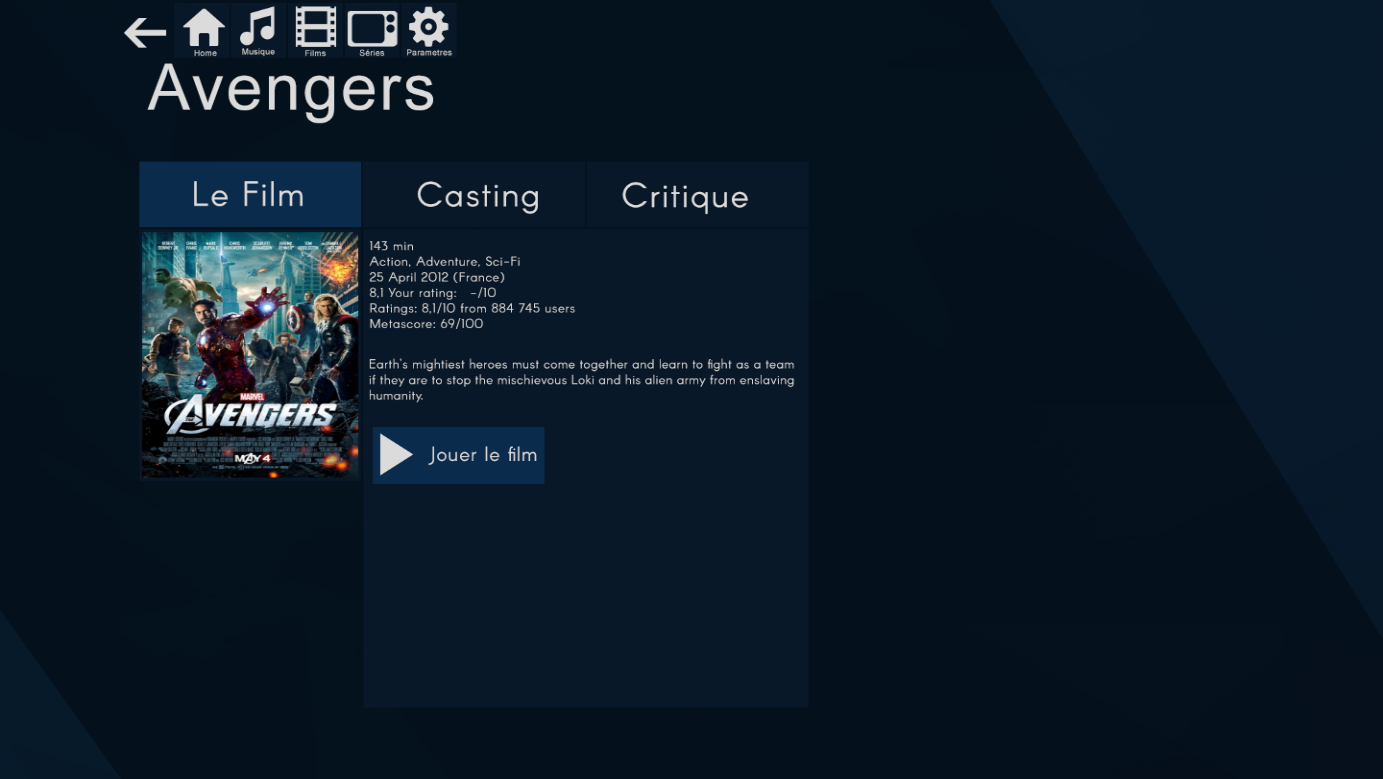


Figure MockUp Film détaillée

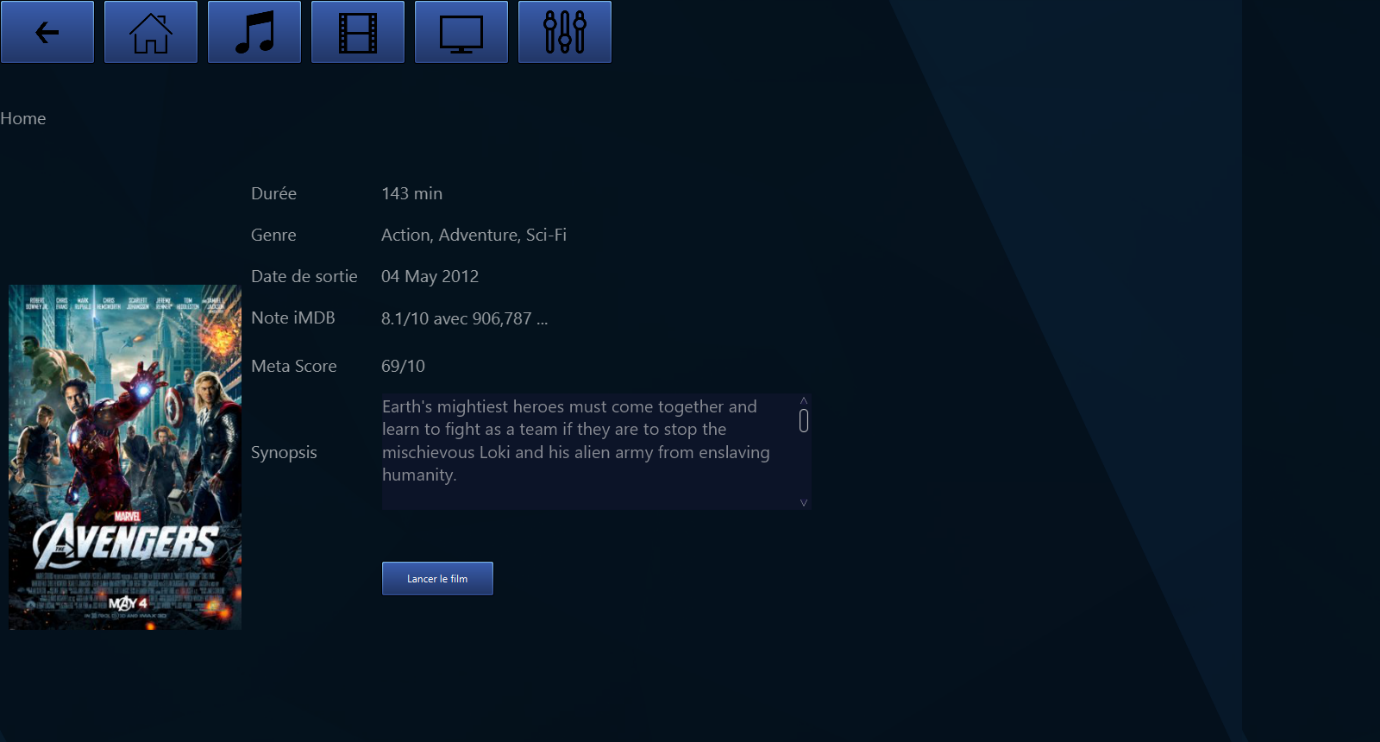


Figure vue Film détaillé

### Vue musique

La vue musique à elle aussi été grandement simplifié, le fait que dans un tableau nous pouvions trier nos musique en cliquant sur les têtes de colonne (Titre, Artiste, Album etc…) nous avons jugé inutile de faire ces onglets, et tout comme ces homologues « film » et « série » son interface réalisée sur Photoshop était là aussi beaucoup trop chronophage. Nous n’avions aussi pas prévue d’espace pour la partie synchronisation, ce qui nous a pousser vers une restructuration complète de cette vue. Nous avons donc pris la décision de merger la partie navigation et la partie player

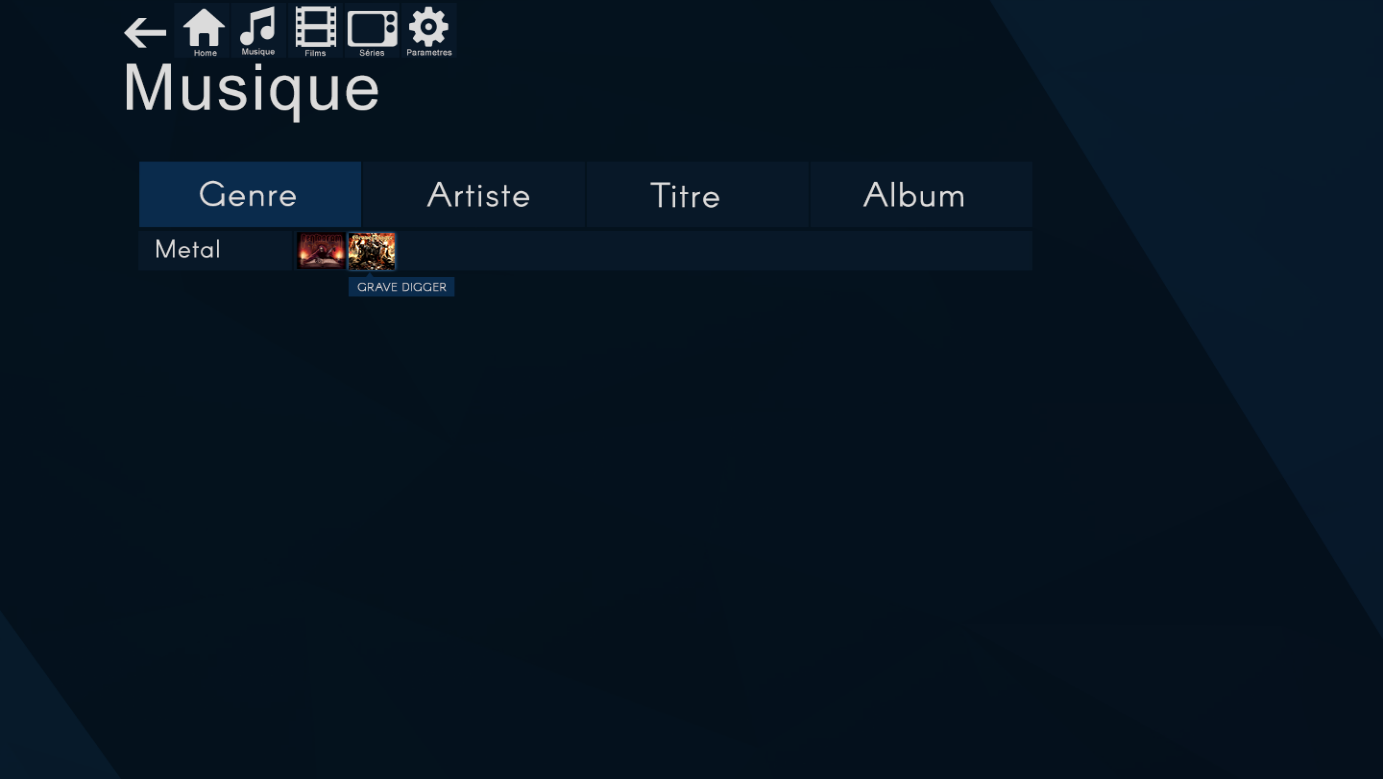


Figure MockUp musique

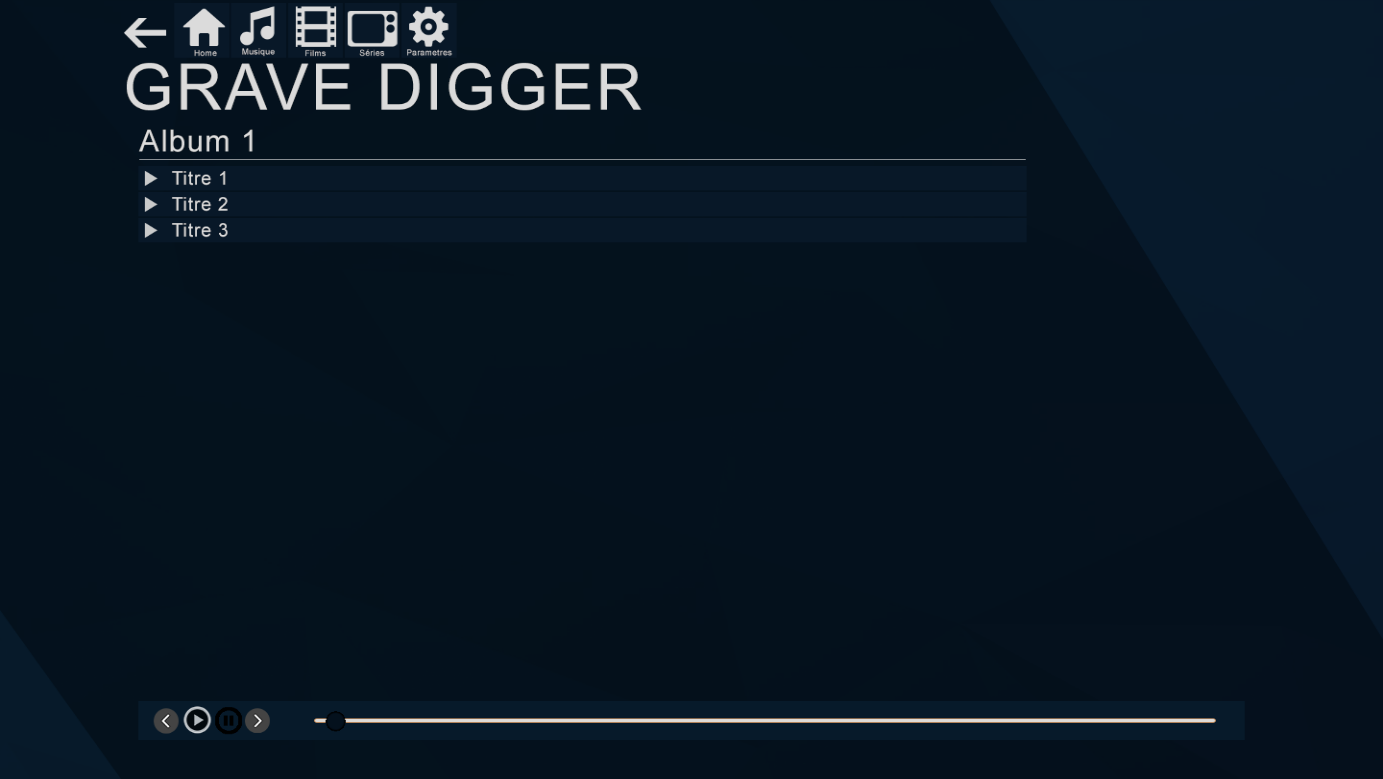


Figure MockUp player

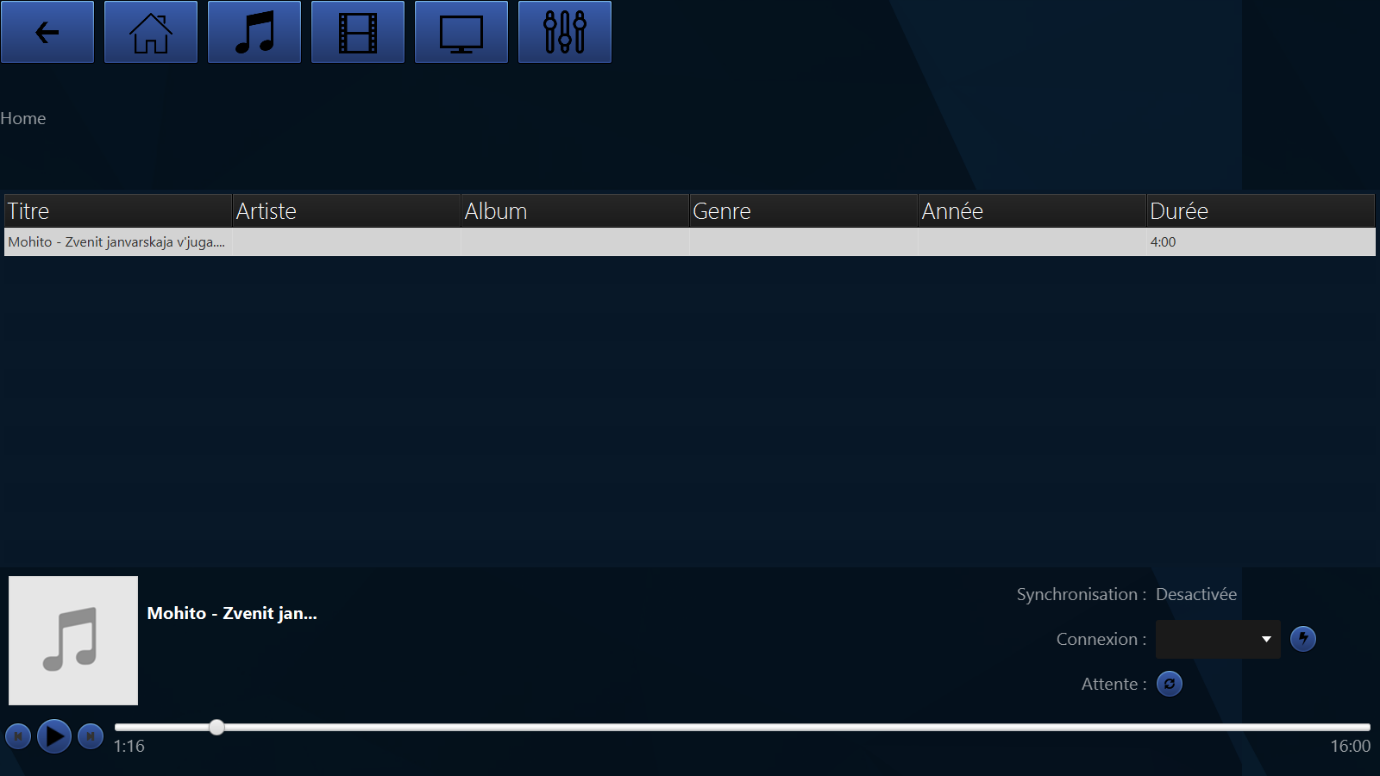


Figure Player réalisé

### Vue série

Tout comme les films et la musique la vue des séries a été simplifiée à un simple tableau rempli par le nom des séries que nous possédons (un simple dossier avec un nom correcte suffi pour récupérer les informations relative à la série)

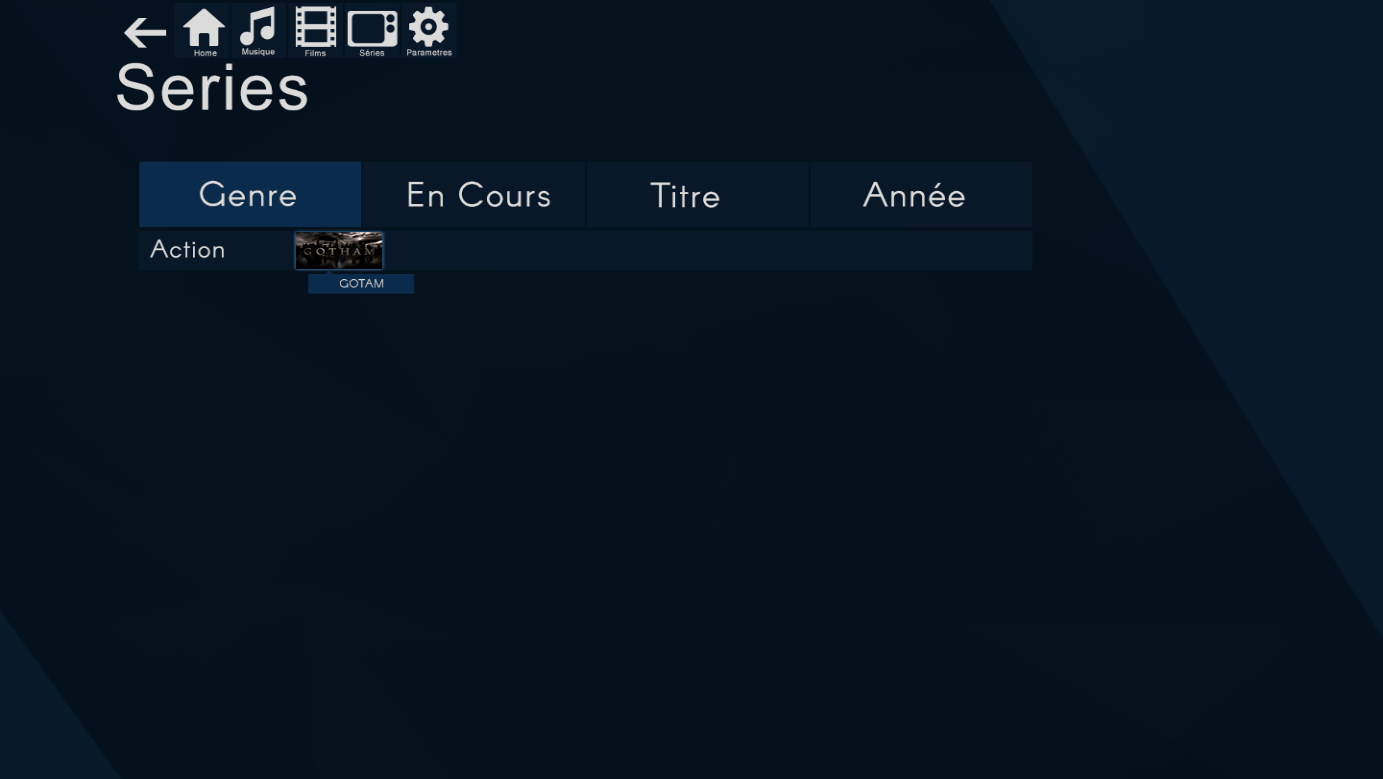


Figure : MockUp serie

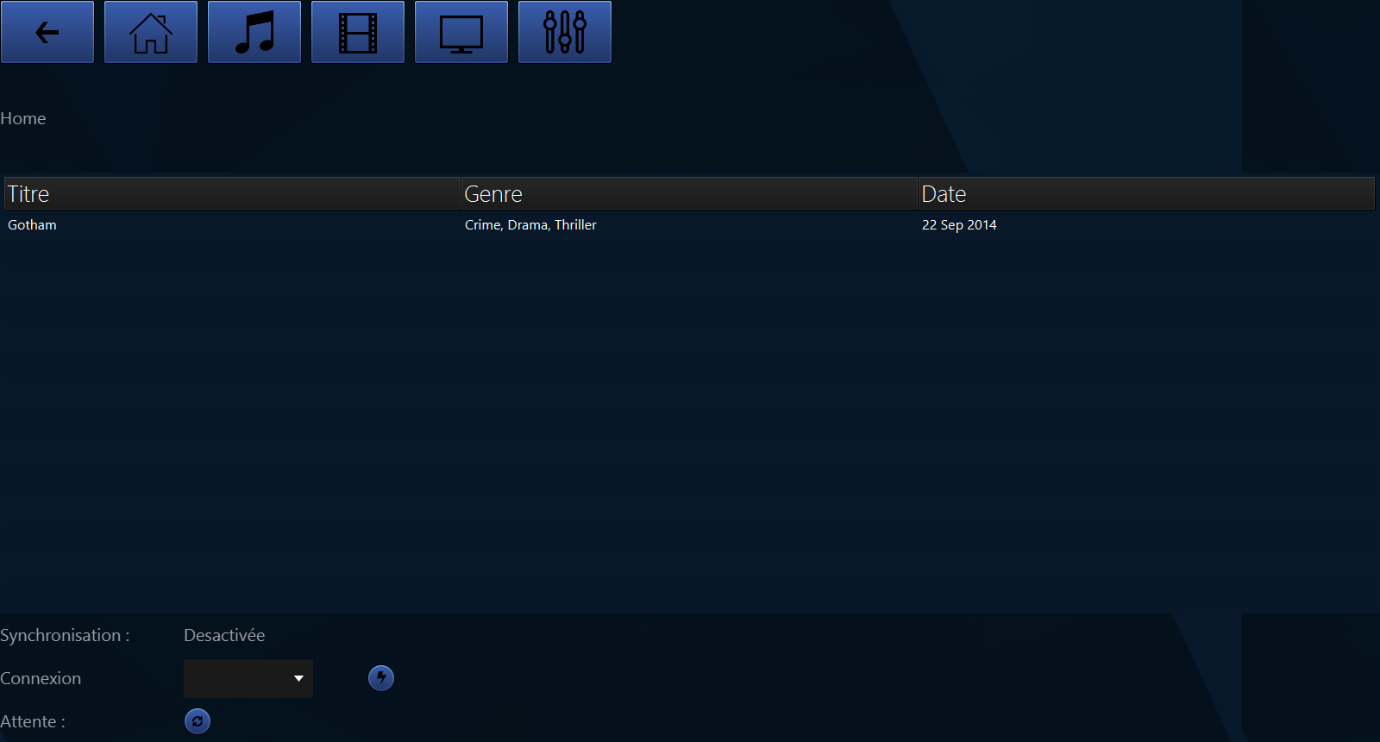


Figure : Vue des séries

### Vue série détaillée

Tout comme la partie détaillée sur le film, la partie détaillée des séries et plutôt proche de l’originale. Les onglets casting et critique ont été supprimer et un onglet « épisode » a été ajouté.

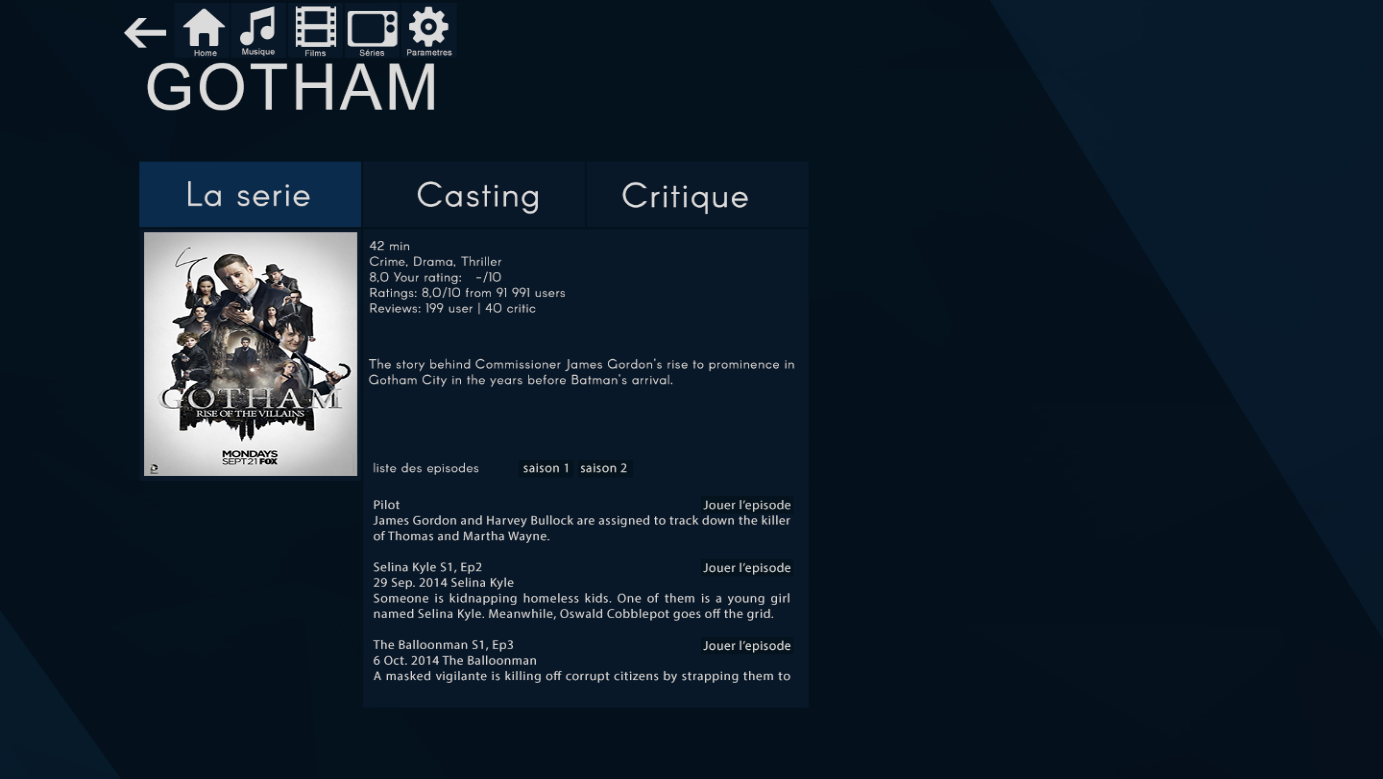


Figure : MockUp série détaillée



Figure vue Film détailé

### Settings

Cette dernière vue ne figurait pas dans nos mockUp, car au debut du projet nous ne savions pas vraiment ce que nous allions y mettre et surtout comment. Nous avions une vague idée des éléments à y mettre, comme les répertoires de musique, film et vidéo.



### Note finale

Pour terminer ce chapitre nous pouvons dire que la réalisation d’une interface sur un outil telle que Photoshop, puis la reproduire via du code n’est pas une mince affaire, surtout quand nous avons les yeux plus grand que le ventre et nous réalisons un mockUp bien travaillé et où l’on a plus pensé artistiquement que pratiquement.

Dans l’ensemble nous avons quand même pu avoir une interface convenable et agréable à utiliser même si elle diffère en pas mal de points de l’originale.

# Réalisation

## Lecteur vidéo

## Lecteur audio

### Technologies

Le lecteur audio a été réalisé à l’aide de JavaFX en respectant le modèle de programmation **MVC**. La *vue* est caractérisé par le fichier fxml décrivant les composants de la fenêtre, leurs positions, leurs tailles, leurs contraintes ainsi que d’éventuels id. Le *contrôleur* est représenté par une classe interagissant avec les composants de la fenêtre. Le *modèle* est défini par une classe Java simple, appelée aussi POJO, représentant une musique elle-même caractérisée par ses tags (titre, artiste, album, …) et autres informations utiles.

### Structure

Son implémentation se trouve dans le package **music**.

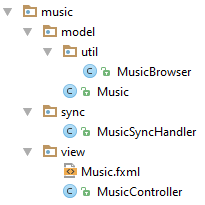


Figure : Package music

Nous avons à l’intérieur un package **model** fournissant la classe modèle Music.java ainsi qu’un autre package **util** offrant une classe MusicBrowser.java permettant la récupération de musique depuis la base de données puis la conversion vers notre format modèle.

Nous avons un package **sync** et sa classe MusicSyncHandler.java implémentant l’interface SyncHandler.java détaillée dans le chapitre 4.4. Cette classe recevra toutes les actions effectuées par l’ami distant lors d’une synchronisation.

Nous avons un package **view** possédant le fichier fxml pour la vue et la classe pour le contrôleur MusicController.java. Nous aurions pu séparer le contrôleur de la vue en l’insérant dans un autre package mais cela aurait empêché le SceneBuilder de le détecter directement.

### Programmation

#### vlcj

Pour la réalisation d’un lecteur audio se basant sur vlcj, nous avons besoin en premier lieu d’informer le système de la location de **VLC.**

Comme VLC est integré directement dans nos ressources, nous pouvons lui passer directement le chemin à l’aide de la commande suivante :



Celle-ci cherchera dans le dossier NATIVE\_LIBRARY\_SEARCH\_PATH la présence d’une instance VLC.

Dans un second temps, nous avons besoin des deux classes suivantes :

* uk.co.caprica.vlcj.component.AudioMediaPlayerComponent

Classe de base à instancier encapsulant le lecteur audio. Tous les détails d’implémentation telle que la création de factory sont intégrés.

* uk.co.caprica.vlcj.player.MediaPlayer

Spécification pour un composant lecteur multimédia.

Afin d’avoir un lecteur audio prêt en arrière-plan, il est nécessaire d’effectuer 2 étapes :

* Instancier un AudioMediaPlayerComponent



* Récupérer la référence sur le MediaPlayer embarqué



Une fois le lecteur prêt en arrière-plan nous n’avons plus qu’à lui passer différentes commandes au travers son interface MediaPlayer tel que :

* playMedia(String mrl, String... mediaOptions)

Lecture du média X.

* pause()

Mise en pause du lecteur.

* play()

Lecture du lecteur en pause.

* setTime(long time)

Saut à un certain moment dans la musique.

Enfin, pour réagir à certains états du lecteur, il est encore intéressant d’écouter les évènements du lecteur médias tel que :

* error(MediaPlayer mediaPlayer)

Une erreur est arrivée.

* finished(MediaPlayer mediaPlayer)

Le média a fini d’être lue.

* playing(MediaPlayer mediaPlayer)

Le média a commencé à être lu.

* timeChanged(MediaPlayer mediaPlayer, long newTime)

Le temps du play-back a changé.

#### Concurrence des threads avec le thread de l’interface

Lorsque plusieurs threads accèdent en même temps à un composant présent dans le thread de l’interface graphique, le programme plante à l’exécution.

Pour résoudre ce problème il est nécessaire d’utiliser le code suivant :



Figure : Implémentation vide

Où l’on donne l’implémentation (ici Anonyme) d’un Runnable effectuant l’action désirée.

Le système se chargera d’effectuer dans l’ordre les actions sur le thread de l’interface.

#### Synchronisation

Pour indiquer au lecteur audio que le système est synchronisé un simple booléen a été utilisé.

Lors de la connexion à un ami ou lors de l’attente d’un ami, nous avons utilisé une stratégie garantissant que le système reste utilisable pendant la connexion et évitant tout interbloquage.

Lors de l’appui sur le bouton pour la connexion et pour l’attente, le système exécute un nouveau thread s’occupant d’appeler la méthode correspondante dans le SyncManager soit respectivement connect et accept. Ces deux méthodes attendent sur des messages (bloquant) il retourne à la fin true ou false selon la réussite ou l’échec de l’opération.

En parallèle, le système déclenche un compte à rebours de 30s à la fin duquel, une vérification si le thread est toujours en cours sera effectué. Dans le cas où le thread s’exécuterait encore, celui-ci est instantanément tué sinon cela veut dire que l’opération de connexion s’est réalisée avec succès.

##### Accept

Dans le cas d’un accept, on attend en premier lieu sur une connexion TCP. Dès qu’on reçoit une connexion, on renvoit un *OK* indiquant à cette personne qu’on a reçu sa connexion. A son tour il nous répond *OK* s’il est toujours de la partie. A ce moment-là, un dialogue apparait nous informant qu’une certaine personne souhaite se connecter. Nous pouvons alors accepter ou refuser l’invitation. Enfin dans le cas où l’on aurait accepté, on attend encore la réponse de la personne en question avant de confirmer la bonne connexion avec un return **true** et la création d’un Worker.

##### Connect

Dans le cas d’un connect, on se connecte d’abord à la personne souhaitée en instanciant un nouveau Socket (connexion TCP). Puis, on attend sur le *OK* de la personne. Lorsqu’on a reçu le OK on renvoit un nouveau OK. A ce moment-là, on attend sur la réponse au dialogue de la personne, on attend donc sur un accept ou un deny. Dans le cas d’un deny, on retourne **false** et la connexion échoue. Dans le cas d’un accept, on envoit un Accept à la personne souhaitée et on retourne **true**.

Afin de mieux comprendre le mécanisme, voici un schéma résumant une connexion à une personne en attente :



Figure : Schéma mécanisme synchronisation musique

## API web

Concernant l’implémentation des API, nous avons pris le choix de créer un package « api » avec 2 sous-package « sound » et « video ». Les différentes dépendances, nécessaire pour le fonctionnement des API, sont ajoutées dans le fichier Maven.

Concernant la partie son, nous avons créé 2 classes. La première classe « GetSoundInfo » qui contient une méthode statique qui prend en paramètre le chemin vers le fichier audio. Cette dernière va lire les métadonnées, et ensuite en se basant sur le nom du morceau, on va faire une recherche sur l’API Spotify. Une fois le résultat trouvé, la méthode va nous retourner un objet de type « TrackInfos » qui contient toutes les informations trouvés sur Spotify, avec notamment l’URL de la pochette.

Nous avons appliqué la stratégie suivante pour la définition des tags d’une musique quelconque :

* On récupère dans tous les cas la longueur des médias
* On récupère le **titre** de la musique par tag ID3
  + Si on ne le trouve pas, on affichera simplement le nom du fichier mp3 comme titre à l’affichage
* On récupère un maximum de tags ID3 (artiste, album, genre, année)
  + Pour chaque tag qu’on ne trouve pas, on tente une récupération via l’API Spotify
* Finalement, on récupère une image de la prochette de l’album via l’API Spotify

## Synchronisation

Pour réaliser la synchronisation de médias, il faut faire attention à 2 points principalement :

* L’exécution des commandes sur le lecteur
* Le passage des commandes via le réseau

Pour passer les commandes via le réseau, nous allons utiliser TCP, car c’est le moyen le plus rapide et le plus simple de mettre en œuvre une communication réseau.

Dans notre application, nous avons implémenté cela sous format d’une seule classe et d’une interface. La classe singleton « SyncManager » permet de créer un serveur TCP, et ensuite d’utiliser le protocole mis en œuvre dans notre application. Tandis que l’interface « SyncHandler » est utilisé pour exécuté les commandes reçus.

Voici maintenant le protocole de communication que nous avons implémenté pour la synchronisation des médias (un « #@ » sépare la commande du paramètre):

* **begin#@NomDuMedia** : permet de donner le nom du fichier afin de comparer et d’informer l’utilisateur le cas où c’est différent.
* **pause** : indique que l’utilisateur a appuyé sur pause.
* **play** : demande au lecteur de lire le média.
* **setAt#@Seconde** : demande au lecteur de se positionner à N seconde.
* **bye** : fin de la synchronisation.

A partir de ces 2 fichiers créés, il s’agit de les « lier » aux différents lecteurs (audio et vidéo), Et cela fonctionne.

Aux niveaux de la robustesse, étant donné que nous utilisons la pile TCP, on est assuré de recevoir les messages dans l’ordre.

# Points non-réalisés

# Problèmes connus

# Planification

La planification recense les différents diagrammes de Gantt que nous avons utilisé lors de la gestion du projet.

## Planification initial

## 

Nous avons structuré les différentes tâches de telle manière à avoir une partie de conception, à travers laquelle nous allons mener divers étude de faisabilité afin de pouvoir identifier les problèmes à l’avance, pour mieux les gérer.

A partir du mois de Novembre, nous rentrons dans la phase d’implémentation dans laquelle nous allons utiliser les différentes méthodes testées pendant la première phase, afin de produire l’application finale.

Au cours du projet, nous avons placé des tâches récurrentes qui doivent être exécuté chaque semaine, comme notamment la séance hebdomadaire et la rédaction de la documentation.

Nous avons aussi ajouté en rouge les différentes dates importantes du projet (état d’avancement, rendu final, et la présentation).

## Re-planification

# Conclusion

# Table des illustrations

# Signatures

Ce document imprimée est validé et certifiée par les auteurs de Flat 5, daté du 4 Janvier 2016.

Baehler Simon

Moret Jérôme

Purro Jan

Berney Léonard

Roubaty Anthony

# Annexes

## Journal de travail

## Cahier des charges

## Manuel d’utilisation