

## **LOG 8430:** [**Architecture logicielle et conception avancée**](https://moodle.polymtl.ca/course/view.php?id=1766)

**TP3 – Mise en œuvre d’une Architecture Logicielle Application distribuée**

**Option 2 - Intégration de services de streaming de musique**

Professeur : M. Yann-Gaël Guéhéneuc

**Étudiants:** Aymen Zalila (1746279)

Marcial Lopez-Ferrada (1609773)

Youssef Zemmahi (1665843)

Groupe : 07

Soumis à : Fábio Petrillo

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

01 Mai 2016

**Introduction:**

Ce document est un rapport qui détaille l’architecture logicielle mise en place par l’équipe de développement de l’application “Iouseph”. Ce projet s’est déroulé en trois étapes et celle-ci représente la première version de la release finale. La première étape consistait à développer un client pouvant communiquer avec les APIs de différents services de streaming de musique. Le second de créer un plug-in Eclipse qui reproduit les mêmes fonctionnalités. Cette dernière étape veut que l’on en implémente un service REST.

Nous recommandons de prendre connaissance des guides développeurs et utilisateurs des anciennes releases pour des informations complémentaires, ainsi que du guide utilisateur fournit avec ce document.

Les trois projets sont accessibles sur github par les liens suivants:

* TP1: https://github.com/elKokito/iouseph
* TP2: https://github.com/EyMenZ/Iouseph4Eclipse
* TP3: https://github.com/EyMenZ/IousephAsAService

**Cahier des charges:**

Dans ce travail, il nous est demandé de réaliser un service REST qui fournit la possibilité de :

* Communiquer avec trois services de streaming musical de notre choix
* Permettre la recherche et l’écoute de musique
* Permettre la création et la gestion de playlists

**Proposition:**

Notre proposition pour répondre à ce besoin est de développer deux applications indépendantes:

* IousephFX: une application JavaFX qui joue le rôle de client, elle permet de consommer le service qui sera mis à disposition par la seconde application.
* IousephAsAService: est un serveur qui gère la communication avec les différentes APIs, de gérer des comptes utilisateurs et de stocker les playlists.

Cette proposition nous permet de présenter une architecture déployée sur trois couches: une couche Client, une couche Service et une couche APIs.

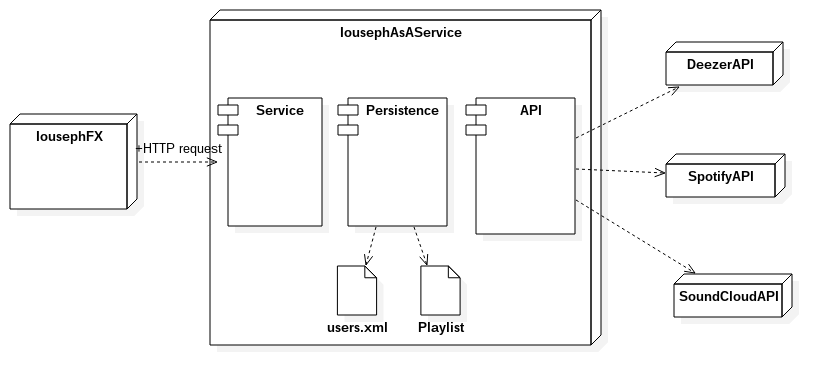


Diagramme de déploiement

**Acronymes:**

REST: REpresentational State Transfer

MVC: Modèle Vue Contrôleur

**Aperçue:**

Ce document comporte 4 sections, qui sont développées comme suit:

* Choix architecturaux: détaille les choix pris pas l’équipe et les patrons de conception utilisés, elle représente l’architecture idéale que nous voulons fournir.
* Vue logique: contient des diagrammes de classe qui détaillent l’implémentation réelle du projet.
* Vue processus: contient des diagrammes de séquences expliquant le déroulement de l’exécution.
* Critiques et travaux futurs: explique les manquements faits par choix ou par nécessité et le contenu de la future version.

**I-Choix Architecturaux:**

Comme précisé précédemment, le projet “Iouseph” suit une architecture trois tiers; Dans ce qui suit nous détaillerons seulement les couches client et service, la dernière couche n’étant pas développé par notre équipe.

**Architecture de IousephFX:**

L’application client IousephFX est une amélioration du projet initial “Iouseph”, elle comprend un ajout de fonctionnalités et un allégement du traitement, qui est délégué à IousephAsAService. Elle reprend le modèle MVC; soit une division des taches de chaque paquetage de cette architecture afin que chacune ait un rôle suivant l’abstraction appropriée. Dans aux implémentations réalisées dans le cadre des travaux antérieurs, celle du présent travail a été quelque peu adapté afin de respecter au mieux les exigences non fonctionnelles imposées soit l'utilisation d’une architecture orientée services ainsi que des services Web RESTful.

L’interface graphique est développée à l’aide de JavaFX. Nous avons choisi d’utiliser la dernière bibliothèque graphique offerte par Oracle dans java 8 parce qu’elle permet une parfaite implémentation du modèle MVC grâce à l’utilisation des fichiers FXML pour les vues reliées aux contrôleurs qui décrivent la logique.

Le contrôleur se compose principalement de deux modules, Le premier permet la gestion des interactions de l’utilisateur. Le second gère tous les aspects reliés à la communication avec le serveur, et ce, en implémentant un client dont le rôle est l’échange d’informations avec le serveur, un parseur permettant de permet des instances d’objets en format JSON reconnaissable par le serveur, ainsi que l’opération inverse, soit la conversion d’un json en une instance utilisable par le client.

**Architecture de IousephAsAService:**

La partie serveur du projet, comme précisé ultérieurement, reprend aussi une partie du projet initial. Les paquetages responsables de la communication avec les APIs, le parsing du Json et le modèle sont repris avec quelques améliorations évolutives ou requises à un environnement de service. Ces parties ont était développées d’une façon flexible pour pouvoir être importées dans d’autres projets.

IousephAsAService est essentiellement composée de trois parties. Une couche de communication Client/Serveur (ou Service), une couche persistance et une couche de communication Serveur/APIs.

L’utilisation d’un Framework pour l’implémentation du service web a été utilisé (<http://sparkjava.com/>). Le choix de ce Framework s’explique par le fait qu’il est simple à utiliser (ce qui réduit notre temps consacré sur cette tache). Un choix différent aurait été justifié si un service plus complexe avait été demandé.

La sauvegarde persistent suit les principes RESTful. Encore une fois, le choix d’un fichier xml comme sauvegarde s’explique par sa simplicité. Dans une situation plus complexe, l’utilisation d’une base de données et l’établissement d’un schéma serait justifié. La persistance de donnée est présentement utilisée pour les usernames et les passwords.

L’utilisation de l’API est détaillée dans le README.md du projet IousephAsAService.

**Architecture Générale:**

Comme cité précédemment l'architecture globale de l’application est une architecture MVC. Les parties Vue et contrôleur sont implémentées au niveau local (Client), Nous avons procédé, comme lors de nos précédents travaux à respecter certaines normes qui nous ont permis d’assurer à nos classe une forte cohésion, une haute lisibilité et compréhensibilité aux systèmes implémentées, une uniformité des méthodes de développement pour chacun des membres de notre équipe. Les normes sont les suivantes :

* La nomenclature associée aux variables, objets, méthodes et classes doivent refléter son rôle. (Favoriser une nomenclature tel que “User utilisateurTemp;” que “ User”)
* Les méthodes dont les fonctions sont les mêmes doivent avoir le même nom, seul les arguments diffèrent. (exemple : “JSONObject ParseToJson(User user)” et “JSONObject ParseToJson(Track)” ) .
* Les classes utilitaires doivent être déclarées statiques.
* Favoriser la création de classe de constantes que l’usage de “HardCoding”. (Favoriser un “return Constants.userAdded;” qu’un “return “UserAdded”.
* Les messages associées au commit se doivent d’être significatifs.
* Des javadocs présents et à jour.
* L’usage des TODO d’Eclipse, afin de facilement repérer les taches restantes et diminuer les probabilités d’oubli ou d’incomplétude.
* L’utilisation de patron de conception que lorsque le contexte nécessite réellement leur usage.

Nous avons remarqué lors de nos évaluations précédentes ainsi que dans les rétroactions des équipes nous ayant évalué qu’il arrive, compte tenu que les travaux présents sont donnés dans le cadre d’un cours d’architecture, que certaines personnes s’efforcent à utiliser ou indique la nécessité d’utiliser un patron de conception dans des cas ou leurs usages n’est pas nécessaire.

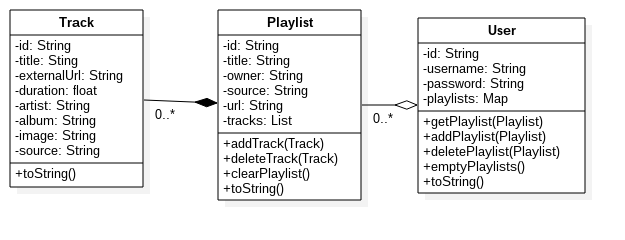
Les normes présentées précédemment ont permis une homogénéité dans le code, un suivi et un débogage facile, et permettent une facilité lors de la réutilisation et l’évolution de paquetages et de classes.

Dans les parties suivantes, nous présenterons la structure des deux systèmes implémentées (Client et service), et y décrirons les différents patrons utilisés, structures en expliquant les raisons derrière les différentes décisions prises.

**II-Vue Logique:**

À l’aide de diagrammes de classes, nous détaillons l’implémentation du projet dans cette section.

Les paquetages “com.iouseph.model” des deux applications sont décrits par la figure suivante. Ils contiennent aussi des getters et des setters pour tous les attributs (nous les avons retirés du diagramme par souci de clarté). La seule différence entre celui de “IousephAsAService” (représente ici) et celui de “IousephFX” est que le second import “javafx.beans.property” au lieu de “java.util”.

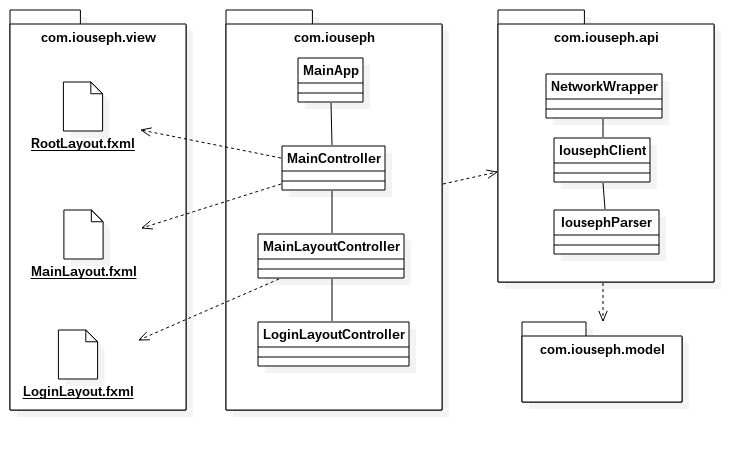


Les besoins du système requièrent uniquement 3 classes; une classe “Track” qui représente la musique, une classe “Playlist” pour créer et gérer des listes de chansons (une playlist contient une List de Track) et une classe user pour gérer les comptes utilisateurs.

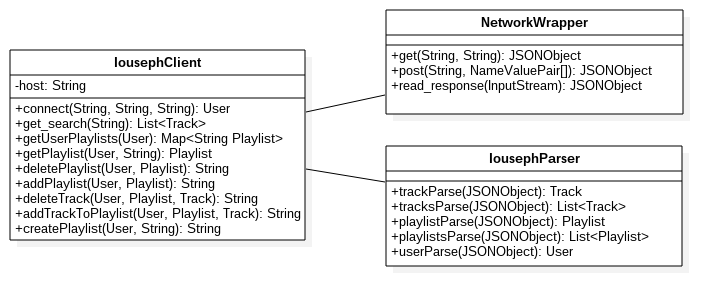
**IousephFX:**

Dans cette partie, nous décrivons la conception de l’application cliente IousephFX. Les deux diagrammes suivants représentent la totalité du projet, le premier est un diagramme général représentant les dépendances entre les paquets, le second détaille le paquetage “com.iouseph.api”.

La figure 3 démontre la présence du modèle MVC, “com.iouseph.model” représente la partie modèle, “com.iouseph.view” la partie vue, et “com.iouseph” la partie contrôleur.



“MainApp” est la classe de lancement de l’application. “MainController” fait appel aux premiers fichiers .fxml, qui contiennent l’interface de l’application, et instancie le premier contrôleur “MainLayoutController”, qui instancie à son tour le contrôleur de l’interface de connexion “LoginLayout” si elle est appelée.

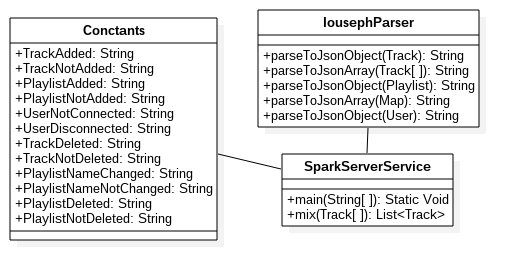


Le paquetage “com.iouseph.api” contient les classes qui implémente le cœur du projet “IousephFX”. “IousephClient” contient les appels au service, “IousephParser” fait le parsing du Json récupéré par les messages envoyés au serveur et “NetworkWrapper” construit les requêtes HTTP (get et post).

**IousephAsAService:**

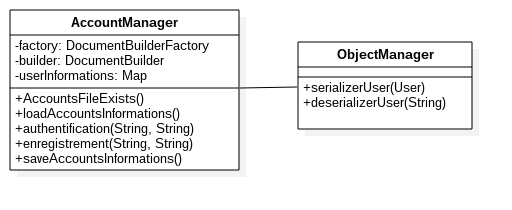
L’application IousephAsAService est composée de trois grandes parties. Dans la section qui suit, nous décrivons chaque partie individuellement.

La partie “Service” représentée par la figure suivante, contient l’implémentation du service REST à l’aide du Framework Spark (d’où le non de la classe “SparkServerService”) et d’une classe “Constants” qui définit les messages à retourner à chaque requête reçue par le serveur.



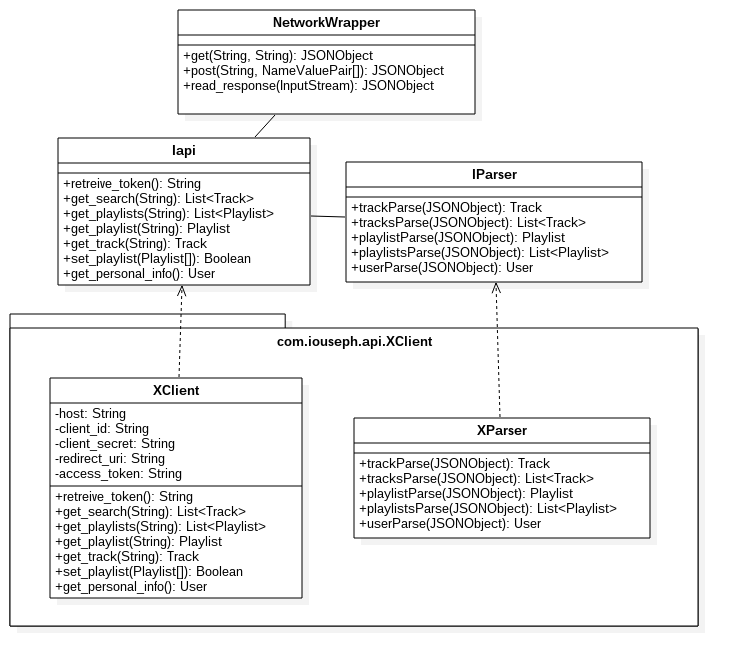
La classe “IousephParser” est appelée par SparkServerService pour faire le parsing des objets avant de les envoyés sous format Json en réponse au client.

La couche “Persistance” est la partie qui gère les enregistrements des informations relatives aux utilisateurs; le nom d’utilisateur, le mot de passe et sa liste de playlists.



Chacune des classes de la figure ci-dessus s’occupe d’une partie des enregistrements. “AccountManager” gère la connexion et l’inscription, elle écrit et lit l’id, le nom d’utilisateur et le mot de passe sur un fichier .xml. “ObjectManager”, quant à elle gère la sérialisation des classes du paquetage “com.iouseph.model” (toutes les classes de ce dernier implémentent l’interface java.io.Serializable), elle enregistre les playlists de l'utilisateur sur un fichier nommé par son id.

La dernière partie représente la communication avec les APIs de services de streaming (Deezer, SoundCloud et Spotify). Dans la figure suivante, un seul paquetage “com.iouseph.api.XClient” est présenté, mais en réalité un paquetage par client est implémenté. Tous les “XClient” implémentent l’interface “Iapi” et “Xparser” l’interface “IParser”.



“NetworkWrapper” représente une façade d’adaptation entre le système et les APIs consommés. Ce patron de conception est couramment utilisé dans la conception des applications distribuées, elle permet de regrouper les appels faits aux différentes APIs.

**III-Vue Processus:**

Dans cette partie, nous expliquerons le fonctionnement des systèmes et décrirons les interactions entre eux. Les diagrammes de classes suivants présentent les principales interactions présentes, ces dernières sont les mêmes que celles présentes dans les autres fonctionnalités des systèmes, seuls les noms des méthodes appelées diffèrent. (Les diagrammes de séquences en format .png sont joints au rapport).



Diagramme de séquences 1 : Authentification de l’utilisateur au système

Lors du processus d'authentification, les valeurs des variables, username et password sont transmises à partir de la classe “LoginLayoutController” à la classe “MainLayoutController” qui possède l’instance du client “IousephClient” qui effectue par le biais de la classe statique “NetworkWrapper” une requête POST transmettant les valeurs saisies par l’utilisateur.

La classe “NetworkWrapper” est une classe utilitaire permettant d’effectuer des requêtes Http (GET et POST dans notre cas).

Une fois la requête reçu par le serveur “SparkServerService”, ce dernier vérifie grâce au singleton “AccountsManager” la validité des informations de connexion, si ces informations sont valides le serveur parse l’instance de l’utilisateur en format Json puis le renvoi au client. Sinon refuse la connexion et transmet une instance vide au client.



Diagramme de séquences 2: recherche de Tracks

Le diagramme de séquences ci-dessus, présente la série d’appels effectuée lors du processus de recherche de Track. Le mot clé saisi par l’utilisateur est transmis à la classe “IousephClient” qui effectue une requête GET au serveur.

Une fois l’information reçue, le serveur transmet le mot clé, aux classes SoundCloundClient, DeezerClient et SpotifyClient. Ces dernières interagissent avec les systèmes de streaming distants afin de récupérer les résultats de la recherche. Ces résultats sont ensuite parser par la classe utilitaire “IousephParser” en format JSONArray puis retournés au client qui les affiche dans la vue.



Diagramme de séquences 3: Ajout d’un Track à une playlist

Le diagramme ci-dessus décrit la séquence d’appels lors de l’ajout d’une Track à une playlist.

Les informations présentes dans l’instance de la classe “Track” à ajouter à la playlist de l’utilisateur actuel sont transmises au serveur via une requête POST en indiquant l’identifiant de l’utilisateur et celui de la playlist dans laquelle sera ajouté le Track.   
Le serveur instancie un objet de la classe “Track” et y ajoute les informations reçues. Il récupère ensuite l’instance de l’utilisateur associé à l’identifiant reçu puis ajoute à la playlist correspondante le Track instancié. Le serveur renvoie ensuite un message au client indiquant le succès ou l’échec de l’opération. Si l’ajout est réussi les nouvelles informations sont ensuite enregistrées en utilisant la sérialisation.   
Les différents messages envoyés sont récupérer à partir de la classe “Constants”, Cette classe contient les différents messages à envoyer, elle est utilisé afin d’uniformiser les messages réduisant les possibles erreurs de frappe pouvant causer des problèmes dans le système.

**IV-Dette Technique:**

Nous énumérons dans cette partie les différents défauts de notre implémentation actuelle du projet:

* Manque de contrôle de saisie et de vérifications des actions non conformes aux scénarios nominaux, (manque de temps)
* Le lecteur de musique ne parvient pour l’instant à lire que les chansons avec un lien http. Celles en https lancent une exception. (manque de maitrise de JavaFX)
* La classe com.iouseph.MainLayoutController.java constitue un blob pour l’instant. Elle devra être décomposée en plusieurs classes.
* Le système de recherche mélange les chansons récupérées en résultat de la recherche d’une façon aléatoire et non pertinente.

**V-Travaux Futurs:**

Cette section énumère les manquements qui pénalisent notre travail à l’heure présente. Ces manquements sont présentent de manière volontaire principalement par manque de temps pour développer un système de haute qualité.

Nous essayons donc de critiques plus que possible notre travail et de déterminer des évolutions à apporter au projet dans le court et long terme.

* Implémenter un service Oauth du coté serveur pour améliorer la sécurité.
* Ajouter un système de chargement dynamique les services de streaming source.
* Améliorer l’architecture du projet et la rendre extensible pour l’ajout de nouveaux services plus facilement.

**Conclusion:**

Ce travail répond en grande partie aux besoins exprimés dans le cahier des charges du TP3. Nous fournissons un service REST structuré, qui fonctionne correctement pour les requêtes implémentées jusqu’à maintenant. Nous fournissons aussi une application cliente avec une interface graphique nette et facile à utiliser. Nous sommes conscients, malgré tout, que plusieurs améliorations peuvent être apportées au système et à son architecture.