POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

LOG8430 : TP3

Mise en œuvre d'une architecture logicielle – Applications distribuées

Option 1 : Gestion de fichiers

Alexandre Chenieux - Thomas Neyraut - Alexandre Pereira
POLYTECHNIQUE MONTREAL

Table des matières

Introduction

Ce document présente l'architecture du logiciel que nous avons implémenté afin de répondre à notre demande client. Il nous a été demandé de concevoir une application serveur permettant d'appliquer un ensemble de commandes sur des fichiers et des dossiers. Les fichiers et les dossiers peuvent être ceux présents sur le serveur hébergeant l'application ou provenir du compte Google Drive ou du compte Dropbox de l'utilisateur. Le programme doit afficher une fenêtre graphique permettant son utilisation. Pour des soucis d'organisation et de développement, nous avons utilisé un dépôt GitHub pour faciliter le développement collaboratif du logiciel (https://github.com/PolymtlAC/LOG8430-TP3). La partie ci-dessous de ce document présente l'architecture du logiciel, son fonctionnement et ses fonctionnalités.

Architecture du logiciel

Dans cette partie, un ensemble de diagrammes et de figures présentent l'architecture, le fonctionnement et les fonctionnalités du logiciel.

a) Cas d'utilisation

Dans un premier temps, nous avons listé les acteurs et les différentes fonctionnalités (actions effectuables par un acteur) que devrait comporter notre logiciel. En effet, il est important dans un processus de développement de commencer par lister les fonctionnalités afin d'être certains de ne rien oublier, et de faire valider cette liste par le client. Nous avons réalisé le diagramme UML des cas d'utilisation suivant (**figure 1**) :

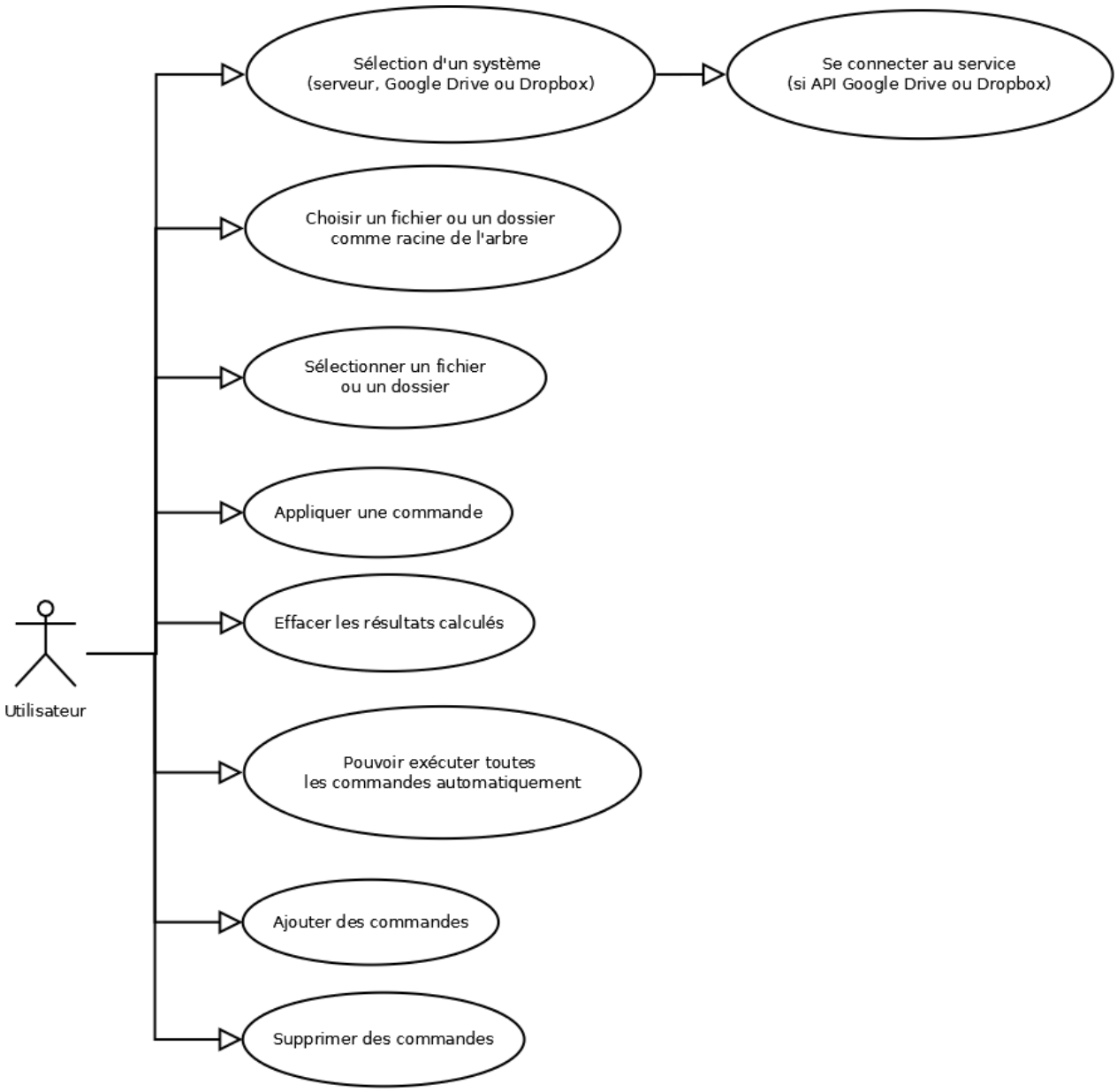

Figure 1. Diagramme de cas d'utilisation

b) Processus du logiciel

Une fois après avoir réfléchi sur les différentes fonctionnalités que notre logiciel devra intégrer, nous avons réfléchi sur les suites d'actions pouvant s'offrir à un utilisateur au cours de son expérience. L'objectif ici est de définir si certaines actions peuvent être effectuées uniquement dans certains contextes (par exemple : après une ou plusieurs actions), et si certaines actions doivent être réalisées avant que l'utilisateur puisse utiliser le logiciel librement. La **figure 2** ci-dessous présente les différentes actions pouvant être effectuées par l'utilisateur. Au lancement du logiciel, ce-dernier va sélectionner automatiquement un dossier racine pour l'arborescence du serveur. Ce dossier est par défaut le dossier « root » présent sur le serveur. Par la suite, l'utilisateur va pouvoir effectuer d'autres actions de son choix en sélectionnant un fichier ou un dossier dans l'arborescence avant de pouvoir exécuter une commande.

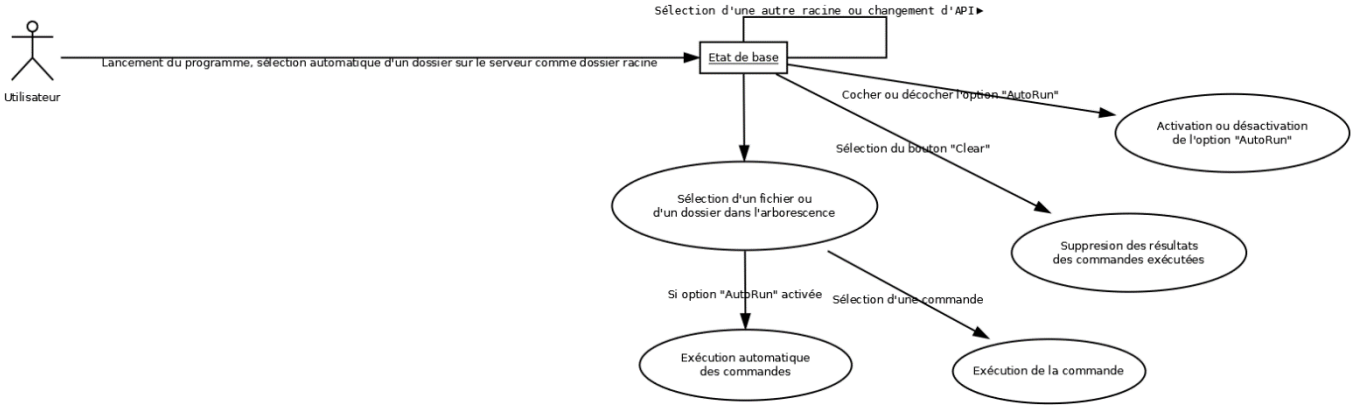

Figure 2. Diagramme de processus

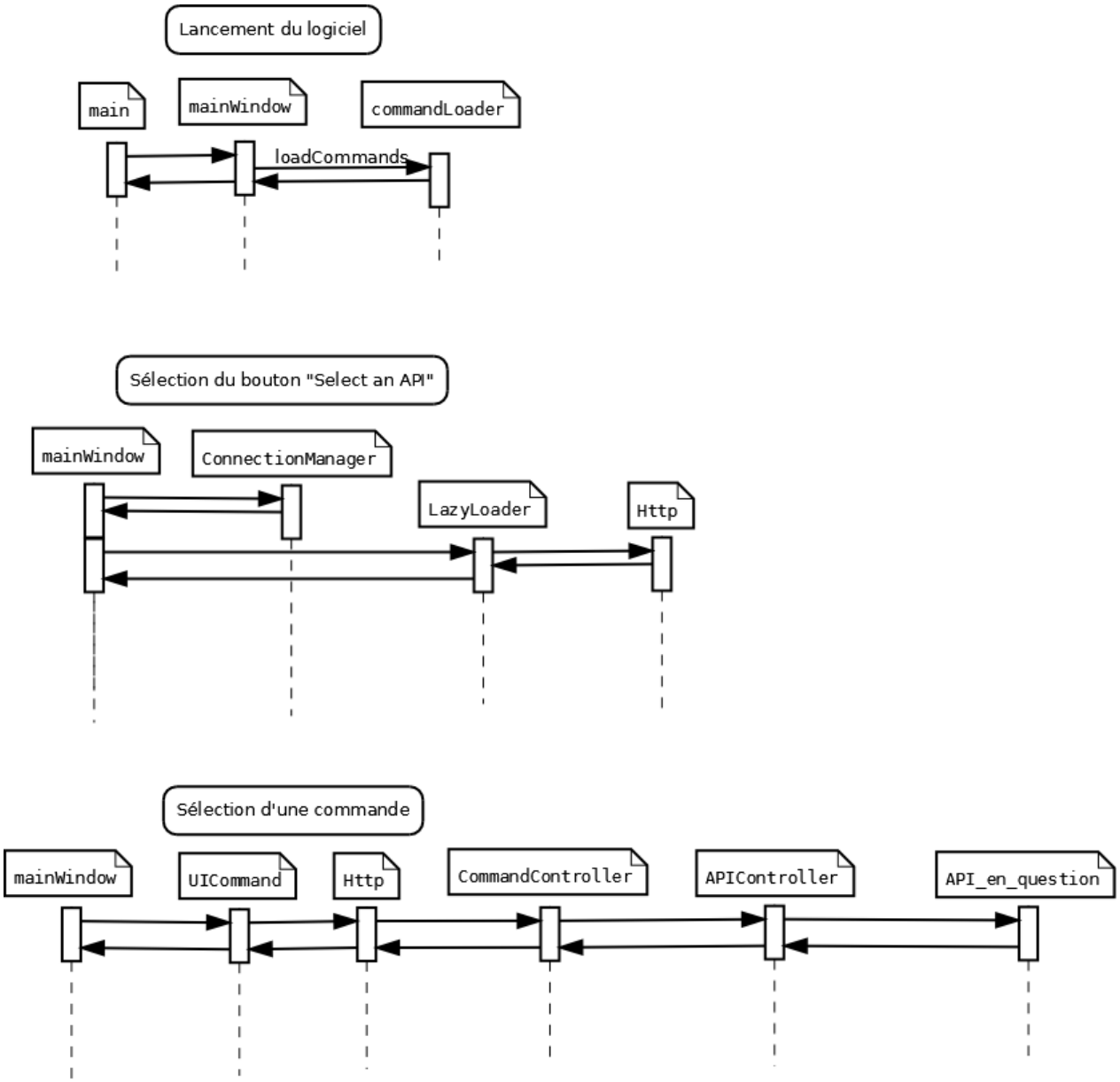

Figure 3. Diagramme de séquences

c) Diagramme de classes

Avant de commencer l'implémentation de notre logiciel, nous avions commencé à réfléchir sur sa structure à l'aide de deux diagrammes de classes. Notre logiciel se décompose en deux parties. La première est la partie client qui offre à l'utilisateur une interface graphique lui permettant d'utiliser le logiciel. Cette partie client de notre logiciel comporte les classes suivantes présentées dans le diagramme de classes ci-dessous (**figure 4**) :

- Command : Cette classe est une classe interface permettant de définir les commandes applicables à un dossier ou/et à un fichier,
- FolderNameCommand, FileNameCommand, AbsolutePathCommand : Ces classes implémentent la classe Command et permettent de définir différentes commandes,

- MainWindow : Cette classe permet de définir et d'afficher la fenêtre graphique principale du logiciel, elle permet aussi de gérer certains évènements provenant de cette fenêtre,
- UICommand : Cette classe permet de définir l'élément graphique complet permettant à l'utilisateur de lancer une commande en particulier parmi celles proposées, et permettant d'afficher les résultats des différentes commandes,
- CommandLoader : Cette classe permet comme son nom l'indique de charger les commandes (fichiers en .class présents dans le dossier « commands »),
- CommandWatcher : Cette classe sert d'observer pour le dossier contenant les commandes et d'en informer l'algorithme maître lorsqu'il y a une modification,
- LazyLoader : Cette classe permet de récupérer l'arborescence sur le serveur, ou bien sur le compte Google Drive ou Dropbox de l'utilisateur. Cette récupération est « fainéante », on fait une requête pour développer un dossier au moment du clic et non tout au début,
- FileNode : Cette classe permet de définir les instances représentant les fichiers et les dossiers dans l'arborescence,
- ConnectionManager : Cette classe permet de gérer la connexion avec le serveur de récupérer l'arborescence du serveur ou de Google Drive ou de Dropbox, transmettre les requêtes des commandes et récupérer les résultats,
- Http : Cette classe utilitaire permet de gérer la connexion avec le serveur.

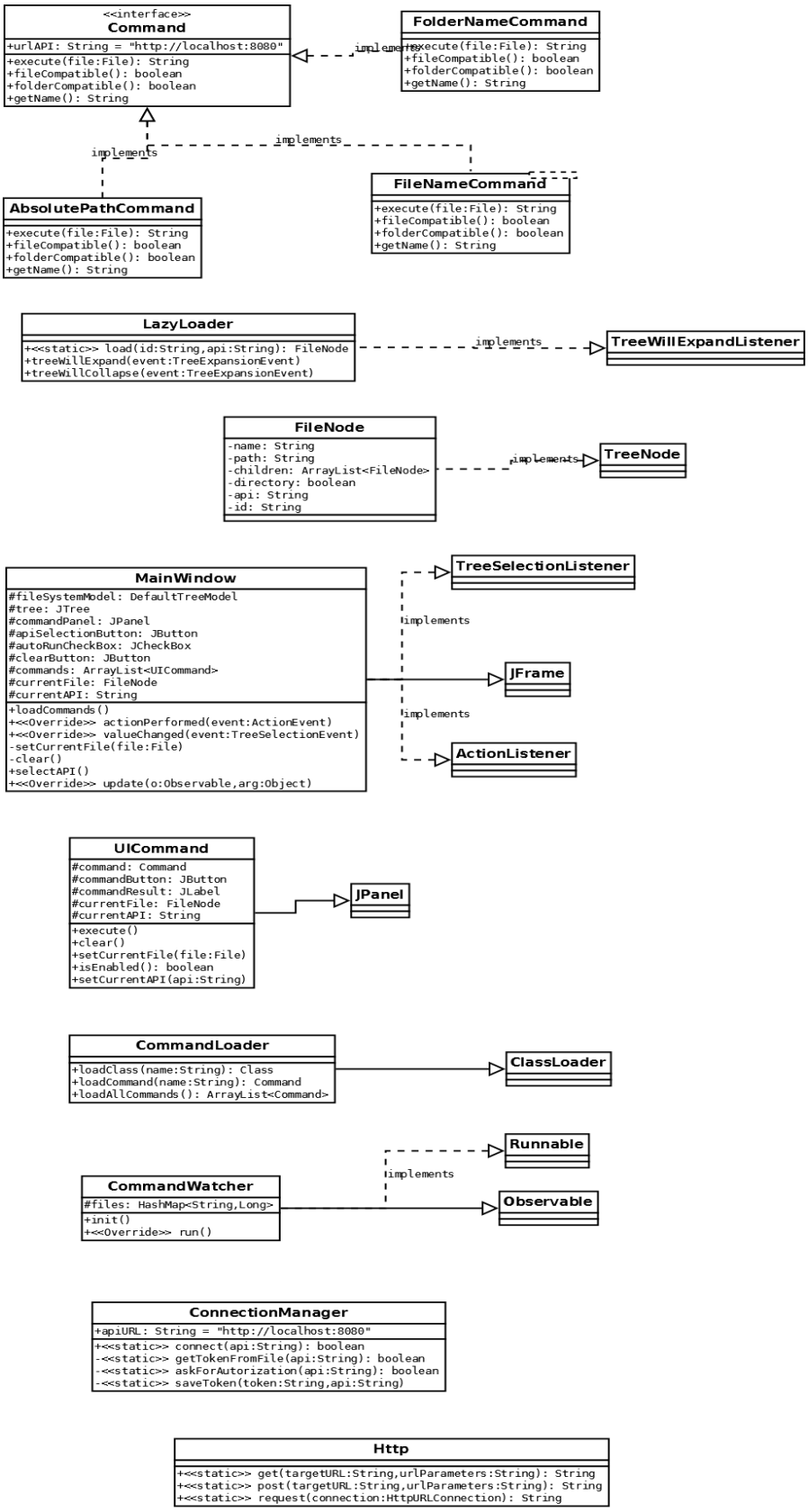

Figure 4. Diagramme de classes de la partie client du logiciel

La seconde partie de notre logiciel est la partie serveur. Cette partie permet de définir un serveur, de communiquer avec la partie client et de communiquer avec les API de Google Drive et de Dropbox. Cette partie serveur de notre logiciel comporte les classes suivantes présentées dans le diagramme de classes ci-dessous (**figure 5**) :

- API : Cette classe est une interface permettant d'implémenter les différentes classes nécessaires pour communiquer avec les différentes API (APIGoogleDrive, APIDropbox ou le serveur lui-même),
- AbstractAPI : Cette classe est une classe abstraite utile dans la définition de l'interface API, qui permet de gérer les requêtes aux API de manière factorisée,
- APIDropbox : Cette permet de communiquer avec l'API de Dropbox pour envoyer les requêtes et récupérer leur réponse,
- APIGoogleDrive : Cette classe permet de communiquer avec l'API de Google Drive pour envoyer les requêtes et récupérer leur réponse,
- APIServer : Cette classe est l'API pour les fichiers locaux du serveur,
- APIFactory : Cette classe permet d'obtenir les instances uniques des « classes API » par leur nom pour les utiliser,
- CommandController : Cette classe « Controller » permet de gérer les différents envois et réceptions des commandes via des requêtes REST en utilisant le framework Spring,
- APIFile : Cette classe « Model » permet de générer, représenter et gérer l'arborescence envoyée au client,
- APIController : Cette classe « Controller » permet de gérer l'authentification de l'utilisateur aux différents services (ici Google Drive ou Dropbox).

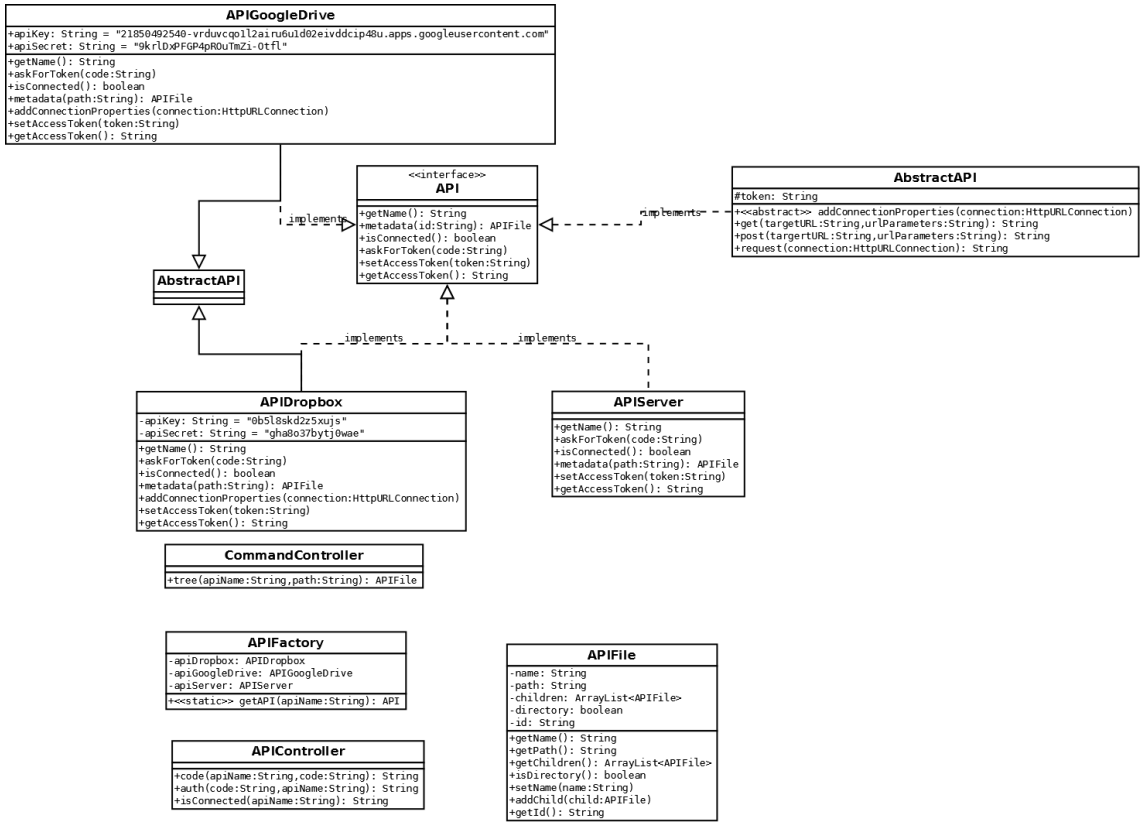

Figure 5. Diagramme de classes de la partie serveur du logiciel

d) Diagramme de paquetage

Afin de présenter sommairement les différentes librairies Java que nous avons utilisées dans nos différentes classes, nous avons réalisé deux diagrammes de paquetage (**figure 6 et figure 7**) pour la partie serveur et pour la partie client.

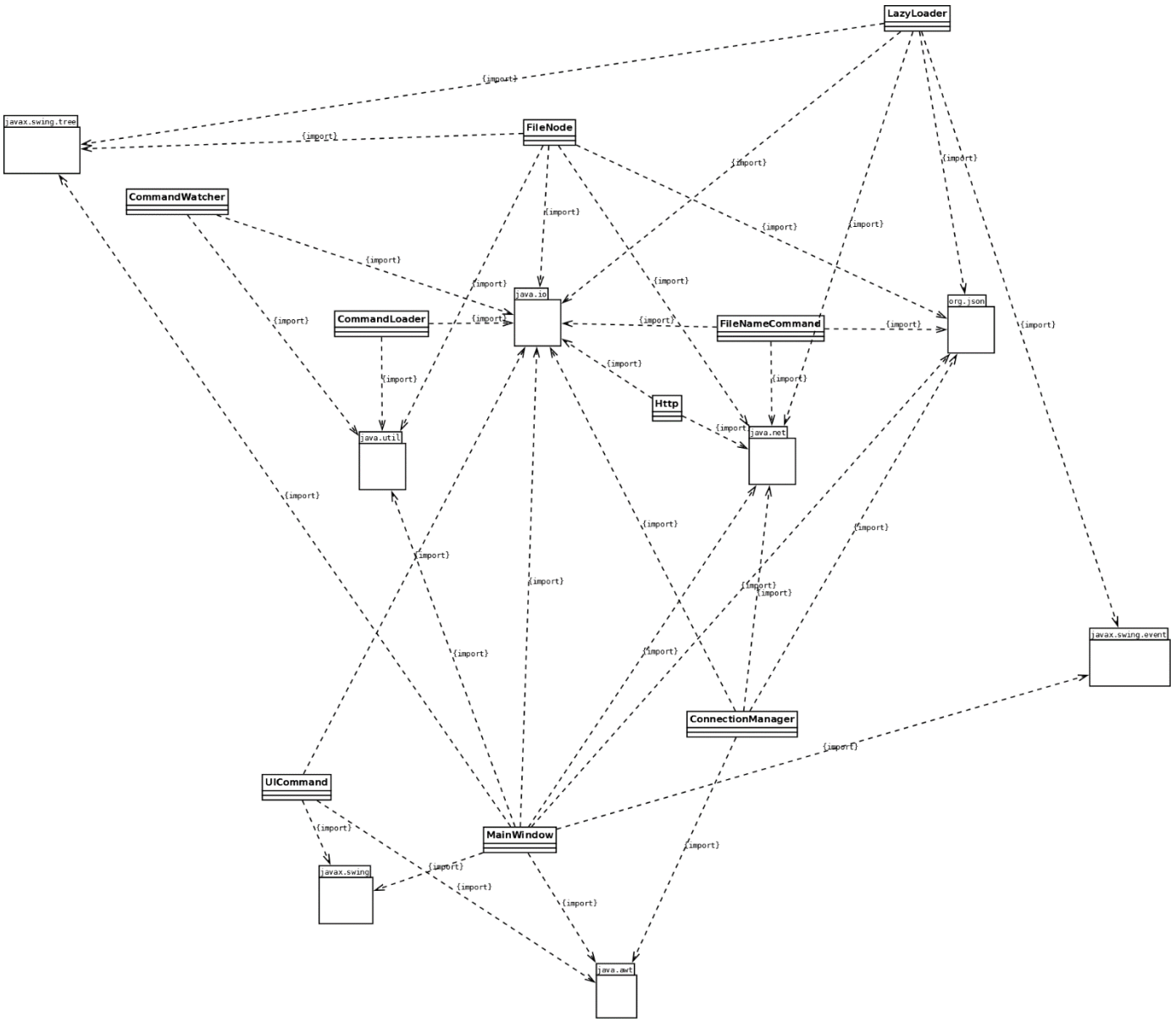

Figure 6. Diagramme de paquetage de la partie client du logiciel

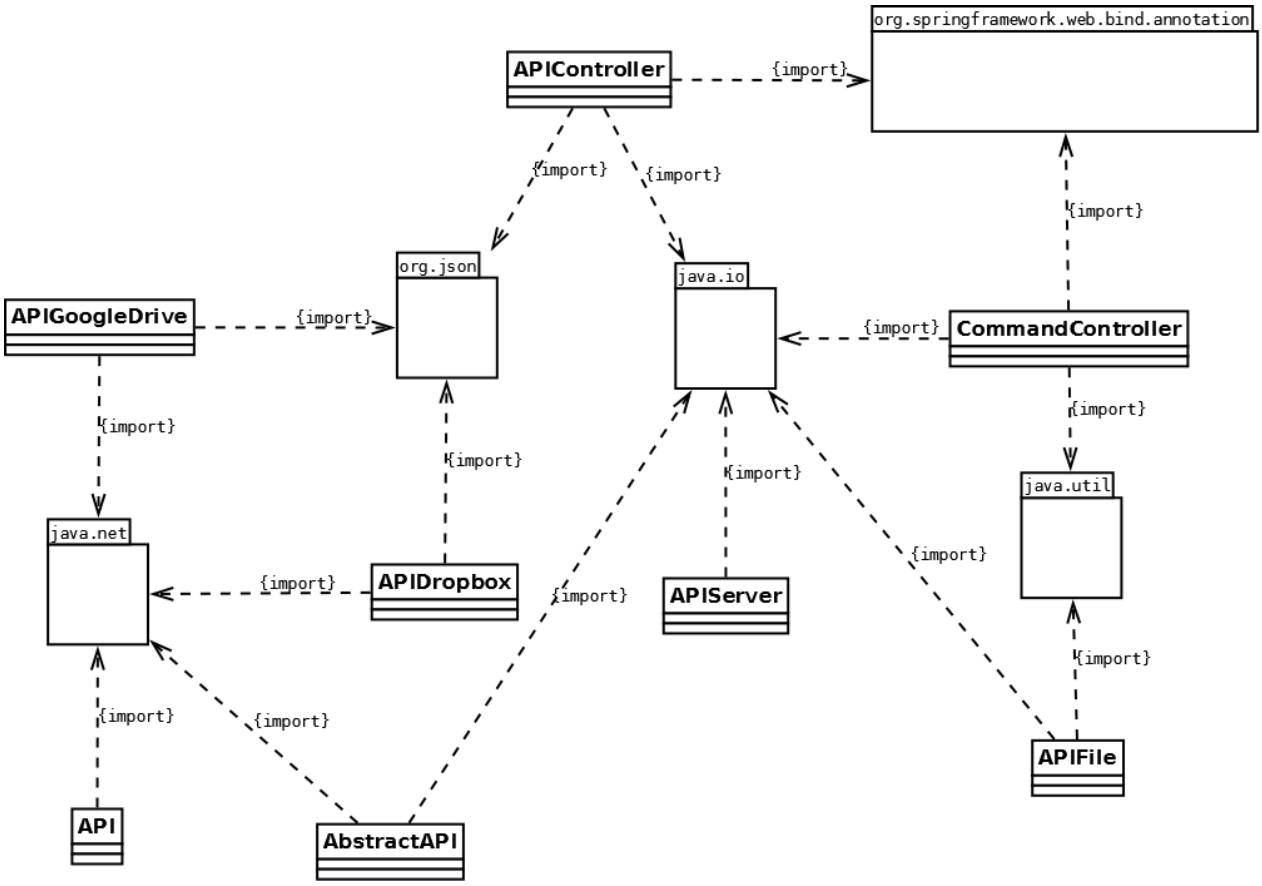

Figure 7. Diagramme de paquetage de la partie serveur du logiciel

Conclusion

Pour conclure, du fait de l'architecture de notre logiciel, il est très simple de pouvoir ajouter de nouveaux services autres que Google Drive et Dropbox. En effet, il suffit de créer une nouvelle classe implémentant « API » côté serveur et l'ajouter à « l'APIFactory » ; côté client, il suffit d'ajouter un bouton représentant la nouvelle API. De plus, il nous est aussi très facile de pouvoir ajouter de nouvelles commandes sans toucher au code du client qui charge les commandes dynamiquement ; côté serveur, il faut ajouter des « URL » au « CommandController » et les méthodes associées dans les API. Dans les prochaines semaines, nous allons continuer à améliorer notre logiciel en le commentant un peu plus et en y ajoutant de nouvelles commandes.