Équipe 3

Turb0_P41nt_F1v3_Th0u54nd.exe
Document d'architecture logicielle

Version 1.3

Historique des révisions

Date	Version	Description	Auteur
2019-09-14	1.0	Vue des processus	Justine Lambert
2019-09-26	1.1	Ajout des cas d'utilisations et des objectifs	Marie-Elaine Bérubé
2019-09-27	1.2	Ajout des diagrammes de paquetage	Christophe Carreau-Lacasse
2019-09-27	1.3	Vue du déploiement et taille et performance	Olivier Naud-Dulude

Table des matières

Document d'architecture logicielle

1. Introduction

L'élaboration d'un jeu multijoueur en réseau de type *Fais-moi un dessin* fait appel à plusieurs concepts et architectures logicielles différents. Ce document explore les choix architecturaux que notre équipe a faits afin de répondre aux exigences du projet. Dans un premier temps, nous allons explorer les objectifs et contraintes liées à l'architecture du système. Puis, nous allons représenter les principaux cas d'utilisation, avant de présenter une vue logique de l'architecture. Nous allons ensuite représenter les processus les plus pertinents du système, puis terminer avec une vue du déploiement ainsi les principales considérations de taille et de performance.

2. Objectifs et contraintes architecturaux

L'objectif de ce projet est de construire une application qui permet à plusieurs joueurs de s'affronter au jeu *Fais-moi un dessin*. L'architecture du projet comporte un client lourd et un client léger, qui communiquent entre eux avec un serveur. Les détails de la communication entre les clients et le serveurs sont détaillés dans l'artéfact *Protocole de Communication*. Les communications seront faites de manière asynchrone afin d'avoir une architecture non bloquante, par exemple pour le service de messagerie et pour le jeu lui-même.

Nous utiliserons un système d'authentification externe, le système Auth0, afin de sécuriser les informations personnelles des utilisateurs via leur système de cryptage. Les routes du serveur seront cryptées par le protocole HTTPS, les données seront donc protégées lors de la communication. Les données seront stockées dans une base de données sécurisée sur Microsoft Azure. Les identifiants de cette base de données ne sont connus que des administrateurs du projet.

Une contrainte importante du logicielle est qu'il devra avoir une portabilité adéquate afin d'être utilisé sur différentes plateformes. Au minimum, le client lourd devra être disponible sur les ordinateurs possédant Windows comme système d'exploitation et le client léger devra être disponible sur une tablette Samsung Galaxy Tab A ayant Android comme système d'exploitation. Nous référons aux dernières versions stables de ces systèmes d'exploitation, soit *Windows 10* et *Android Pie*. L'architecture séparée en client lourd, client léger et serveur permettra de résoudre cette contrainte architecturale.

Nous devrons réutiliser certaines parties du logiciel *PolyPaint*, qui possède sa propre architecture. Afin d'éviter les problèmes d'intégration, nous minimiserons les changements à l'architecture du logiciel *PolyPaint*, autant que possible.

Nous avons choisi d'utiliser C# avec WPF comme langage de développement pour le client lourd et Kotlin pour le client léger. Le serveur sera fait en .NET Core. Les outils de développement pour le client lourd et le serveur sont Visual Studio 2019 et pour le client léger, Android Studio sera utilisé.

3. Vue des cas d'utilisation

Les cas d'utilisation les plus pertinents sont présentés dans cette section. Ils s'appliquent au client ou clients mentionnés dans la légende du cas d'utilisation.

Deux acteurs principaux interagissent avec le logiciel *TurboPaint 5000*:

-Utilisateur : Personne qui utilise le logiciel.

-Système d'authentification Auth0 : Le système d'authentification utilisé par le logiciel.

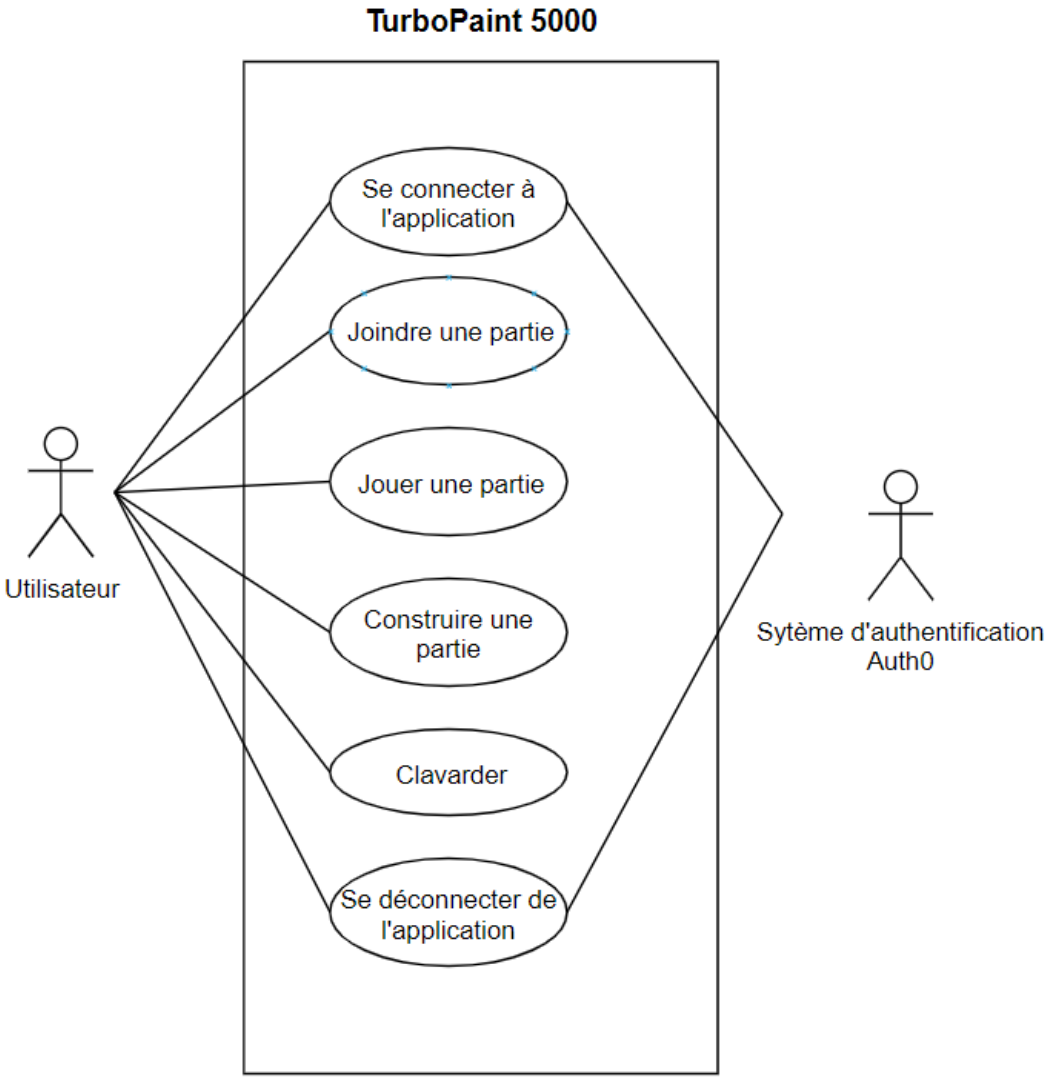

Figure 3.1 : Diagramme des cas d'utilisations principaux des clients lourd et léger

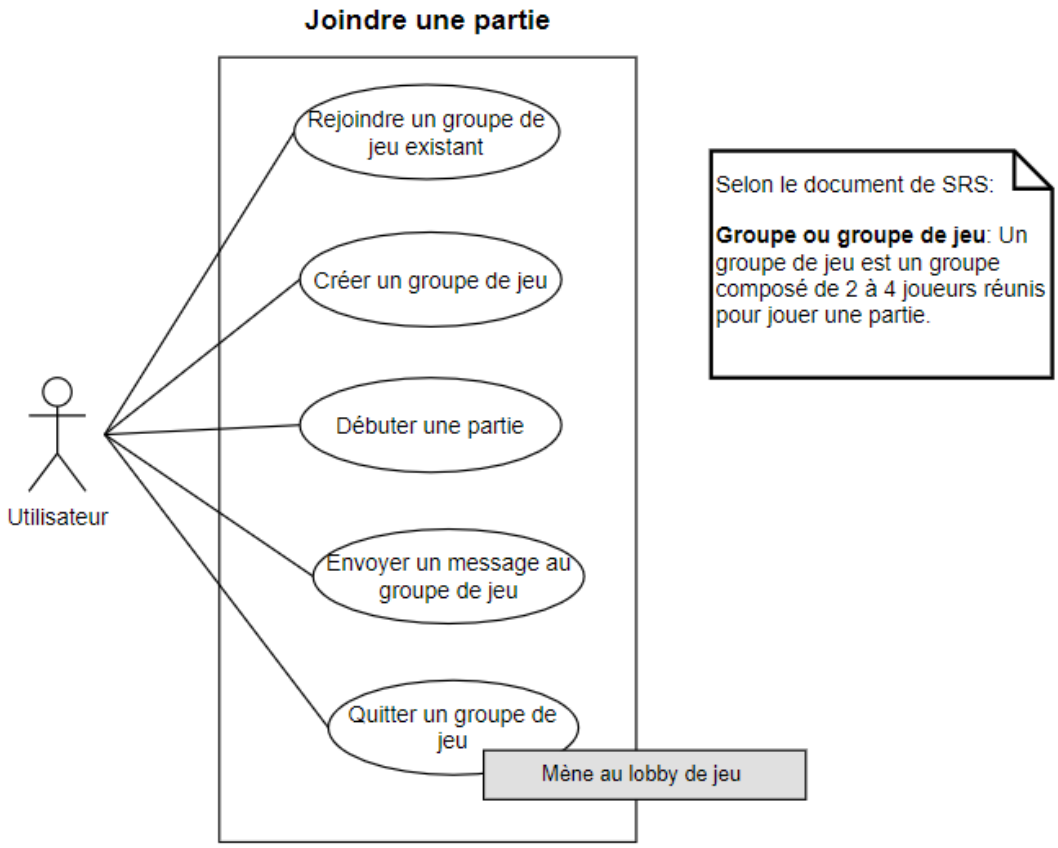

Figure 3.2: Diagramme du cas d'utilisation *Joindre une partie* des clients lourd et léger

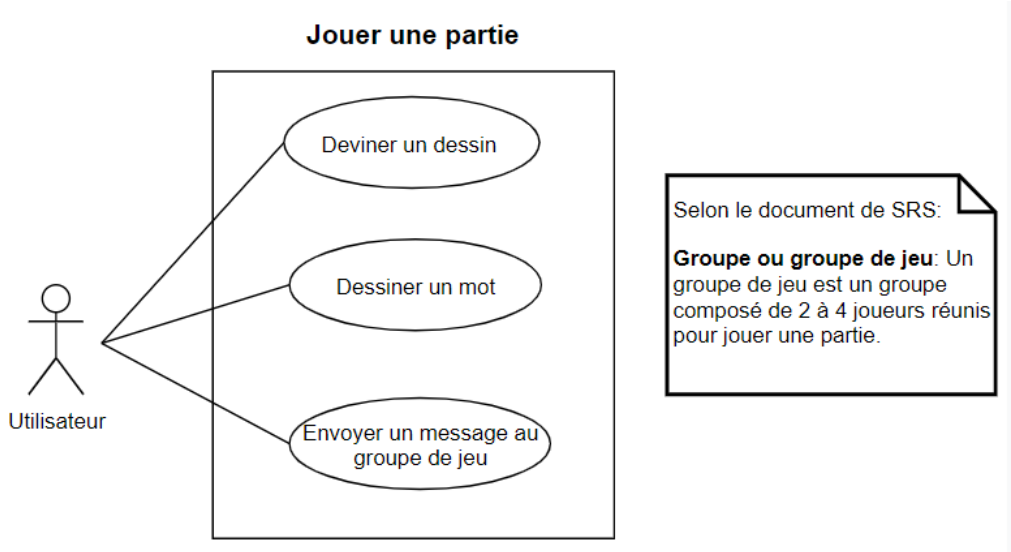

Figure 3.3: Diagramme du cas d'utilisation *Jouer une partie* des clients lourd et léger

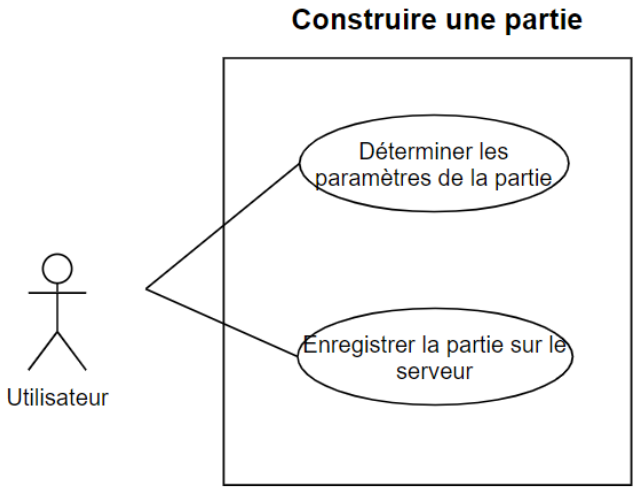

Figure 3.4: Diagramme du cas d'utilisation *Construire une partie* du client lourd

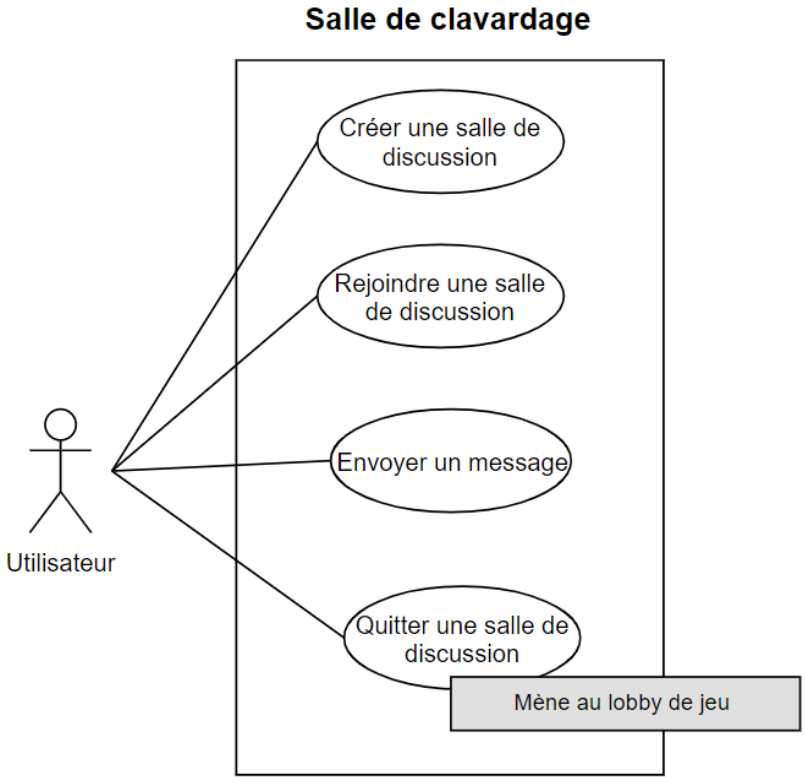

Figure 3.5: Diagramme du cas d'utilisation *Clavarder* des clients lourd et léger

4. Vue logique

Cette section présente l'architecture du logiciel, sous forme de différents diagrammes de paquetages. Le premier diagramme représente le logiciel dans son entièreté et est suivi d'un diagramme pour chacun des ses paquetages, soit le serveur, le client léger et le client lourd. À la suite de chacun de ces diagrammes, une description de chaque paquetage est donnée.

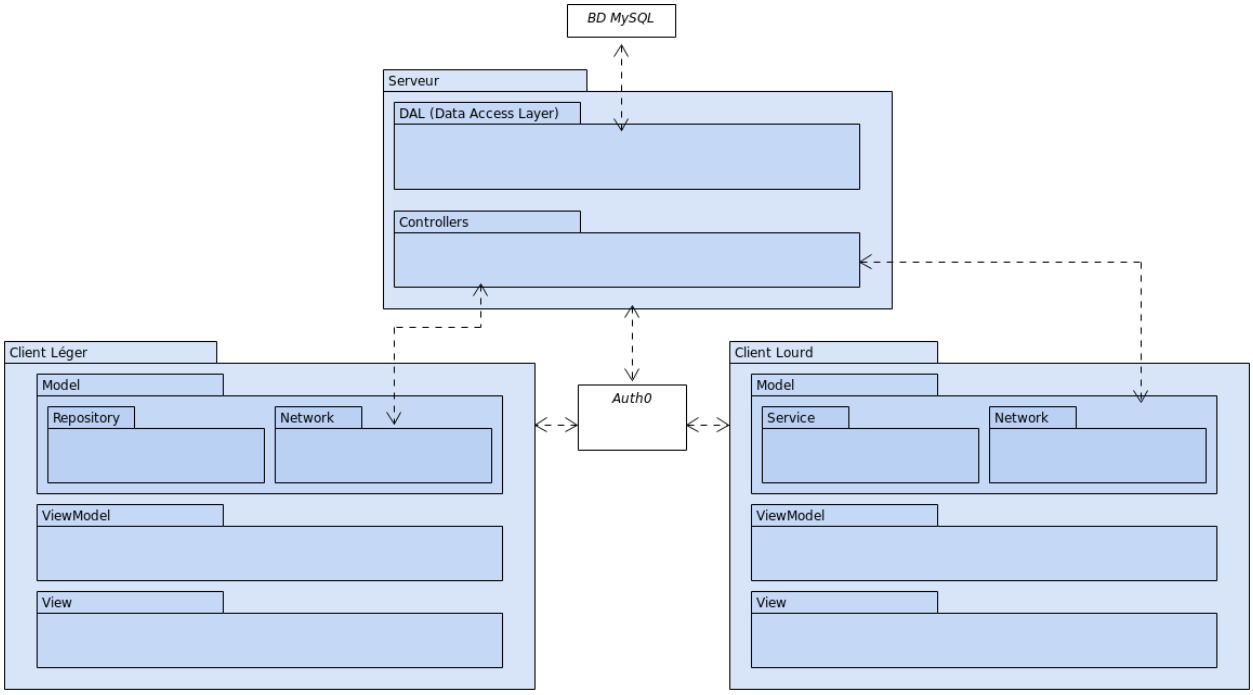

Figure 4.1: Diagramme de paquetages de l'application entière

Serveur
Ce paquet représente le serveur

Client léger
Ce paquet représente le client léger. Le client léger est séparé en 3 paquets principal car il suit le modèle MVVM pour *Model-View-ViewModel*. Ce modèle a pour but de séparer le code selon son utilité.

Client lourd
Ce paquet représente le client lourd. Le client lourd suit aussi le modèle MVVM et est donc séparé en 3 paquets principal.

BD MySQL
Ce paquet représente la base de données MySQL qui contient toutes les données de notre application.

Auth0
Ce paquet représente le service *Auth0,* qui gère l'authentification des utilisateurs de notre application.

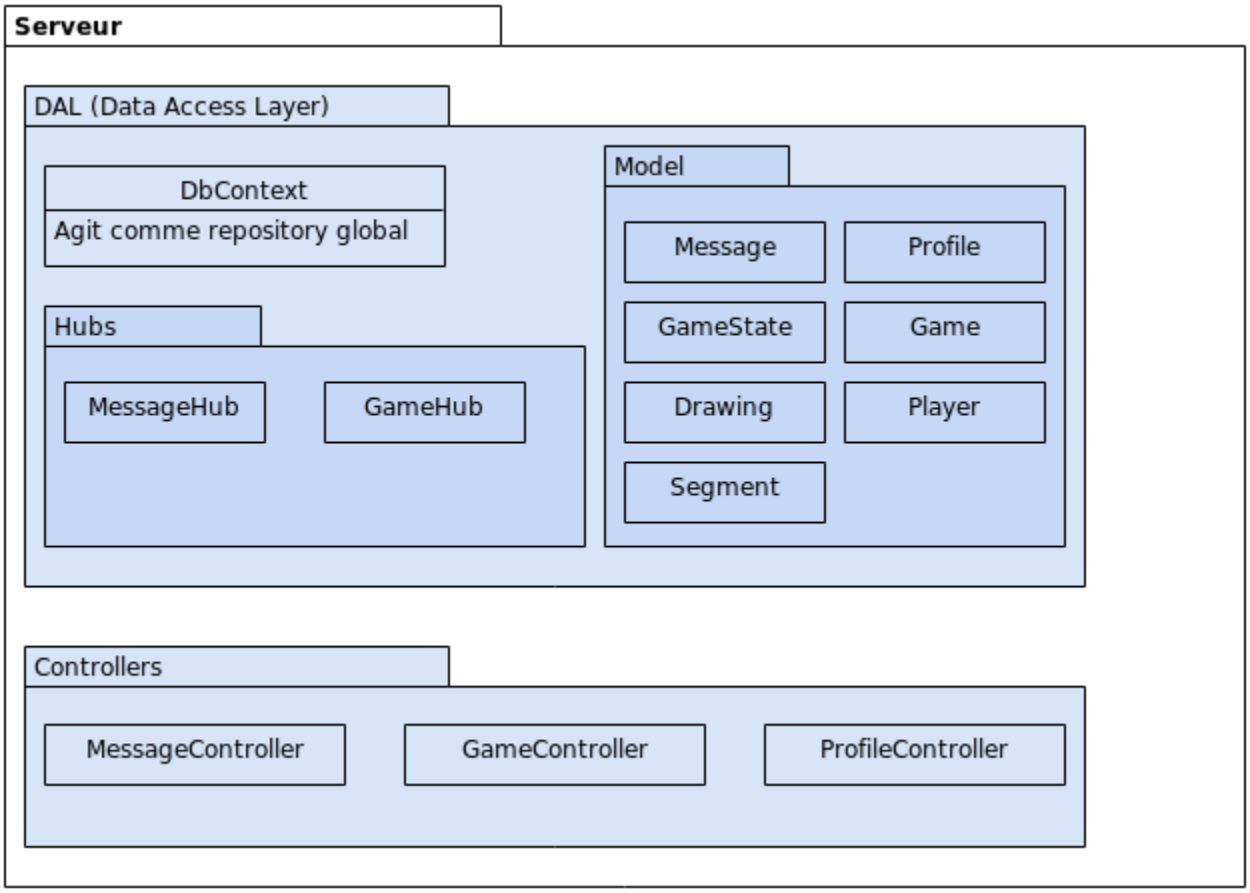

Figure 4.2: Diagramme de paquetages du serveur

DAL (Data Access Layer)
Ce paquet contient tout ce qui a trait au modèle et à l'accès à la base de données. La classe DbContext initialise *EntityFramework*[1] et donc contient une référence à tous les *repository* de la base de données. Le sous-paquet *Hubs* contient les actions et la logique lié au *WebSocket SignalR*. Le sous-paquet Model contient les classes de modèle.

Controllers
Ce paquet contient les contrôleurs de l'API et s'occupe de faire le lien entre les clients et la couche d'accès aux données (*DAL*).

[1] https://docs.microsoft.com/en-us/ef/

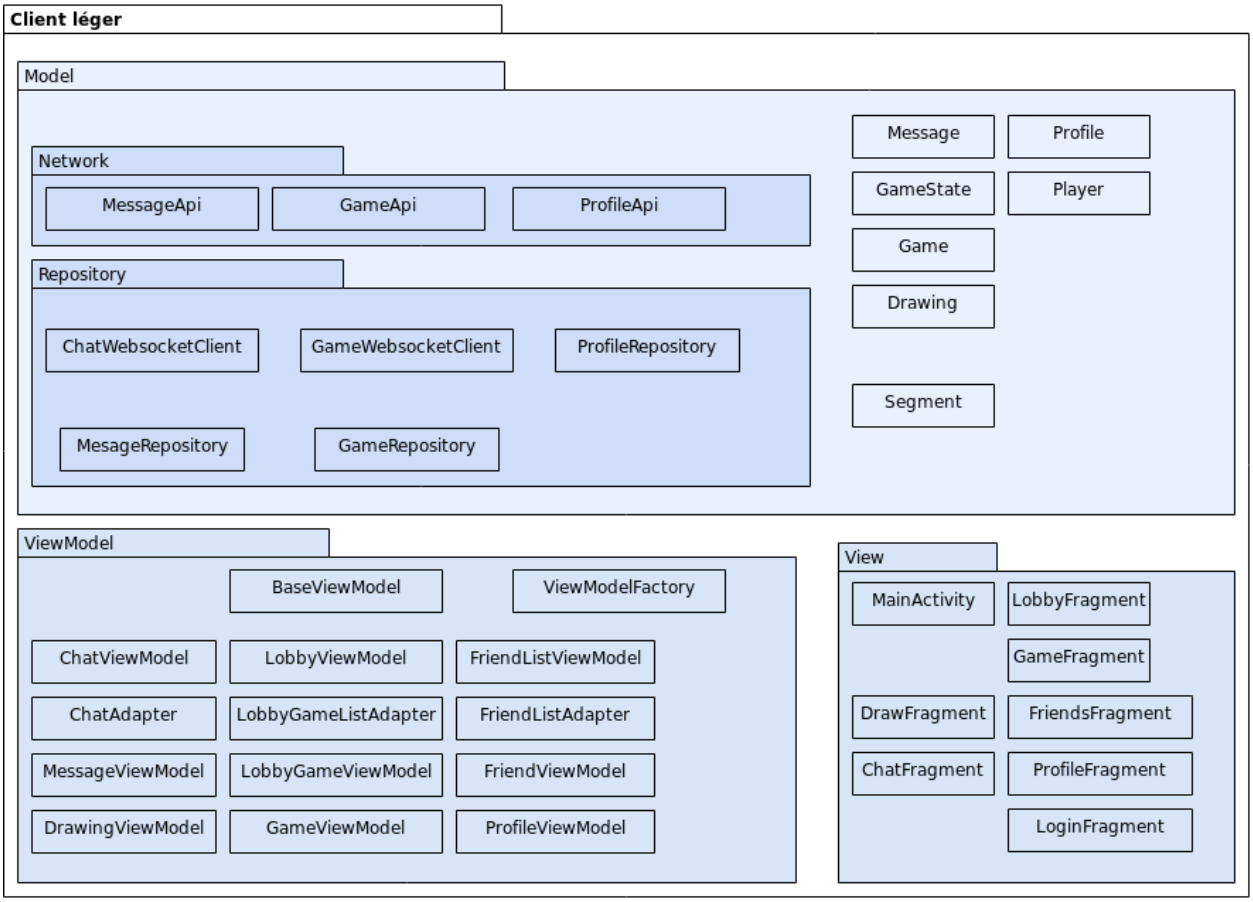

Figure 4.3: Diagramme de paquetages du client léger

Model
Ce paquet contient tout ce qui à trait au modèle, c'est-à-dire les classes du modèle, les *repository* qui s'occupe d'aller récupérer les données du serveur via l'API et le *WebSocket SignalR*, et les clients de l'API (dans la couche *network*). Les clients de l'API sont générés automatiquement par l'outil *Retrofit*[2].

ViewModel
Ce paquet est la couche intermédiaire entre le *Model* et la *View*, c'est elle qui transforme l'information obtenue du serveur en valeur adapté à la vue. Les classes finissant par *Adapter* sont des classes nécessaires afin d'associer un fichier XML aux éléments d'une liste.

View
Ce paquet contient l'activité principale de l'application ainsi que tous les fragments qui la compose.

[2] https://square.github.io/retrofit/

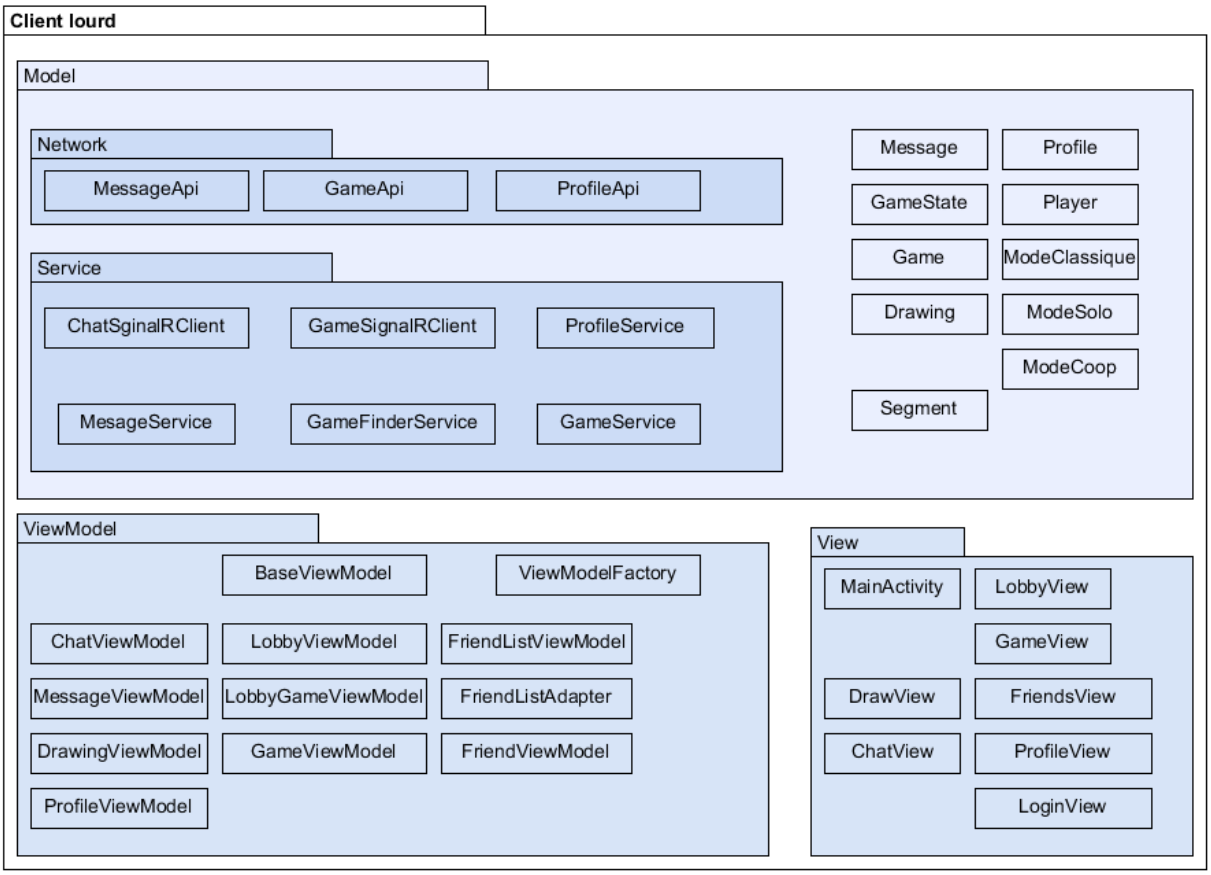

Figure 4.4: Diagramme de paquetages du client lourd

Model
Le Model contient les squelettes de chacuns des objets que nous allons avoir dans le client lourd. Son rôle principal est de communiquer avec le serveur par le biais de requêtes HTTP et SignalR afin d'obtenir de l'information et d'envoyer celle-ci au ViewModel.

ViewModel
Le ViewModel est l'endroit où l'on reçoit l'information des services (Model) et on transforme celle-ci en modèle pour qu'elle soit utilisable par la vue. On peut aussi faire du *Data binding* entre les variables du ViewModel et la vue pour que les informations dans la vue soient mises à jour directement.

View
Ce paquet contient l'interface utilisateur de l'application séparés en plusieurs composants qui sont chacun relié à un ViewModel

5. Vue des processus

Cette section présente l'interaction entre les différents processus de l'application pour trois des principaux cas d'utilisation du système, soit échanger des messages dans le clavardage, jouer une partie en mode classique et construire un jeu en mode assisté I.

5.1 Clavardage

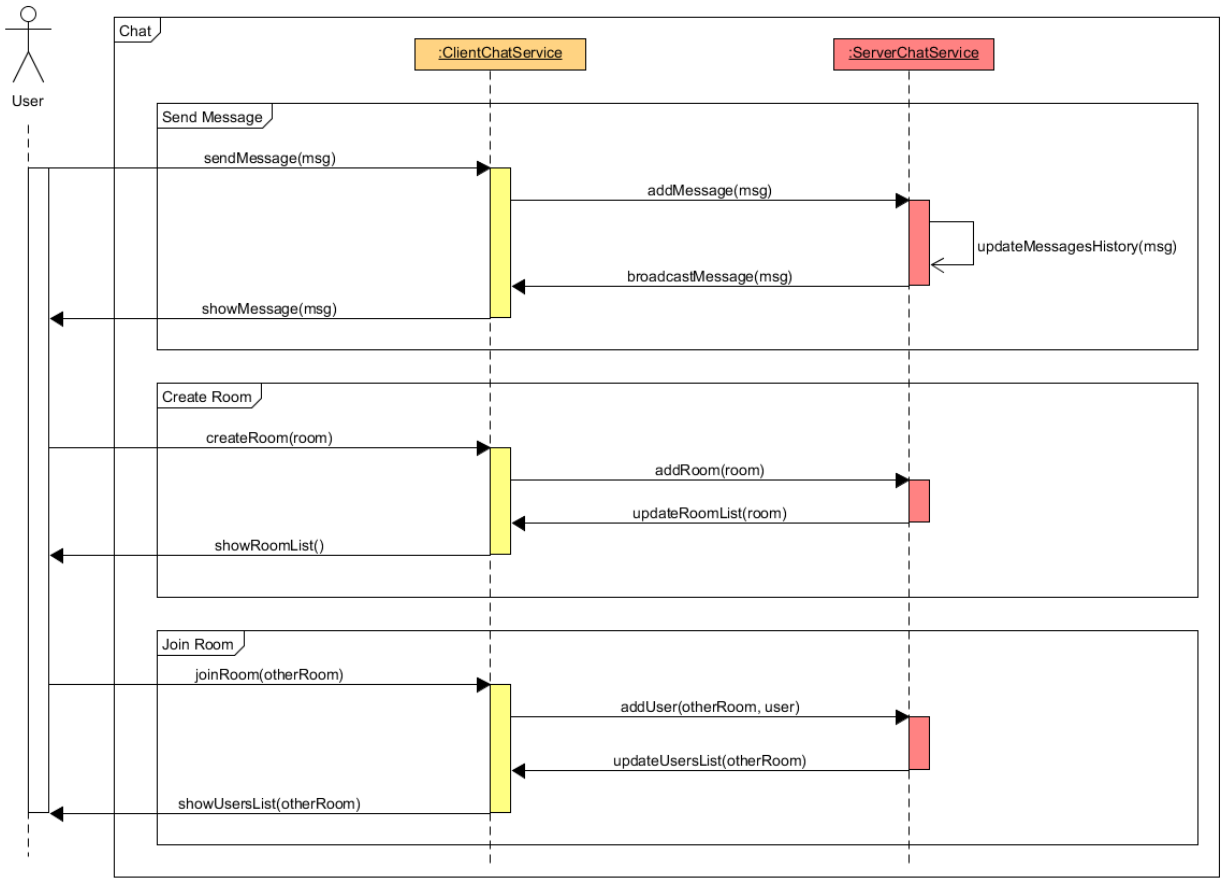

Figure 5.1: Diagramme de séquence pour le clavardage

5.2 Mode de jeu classique

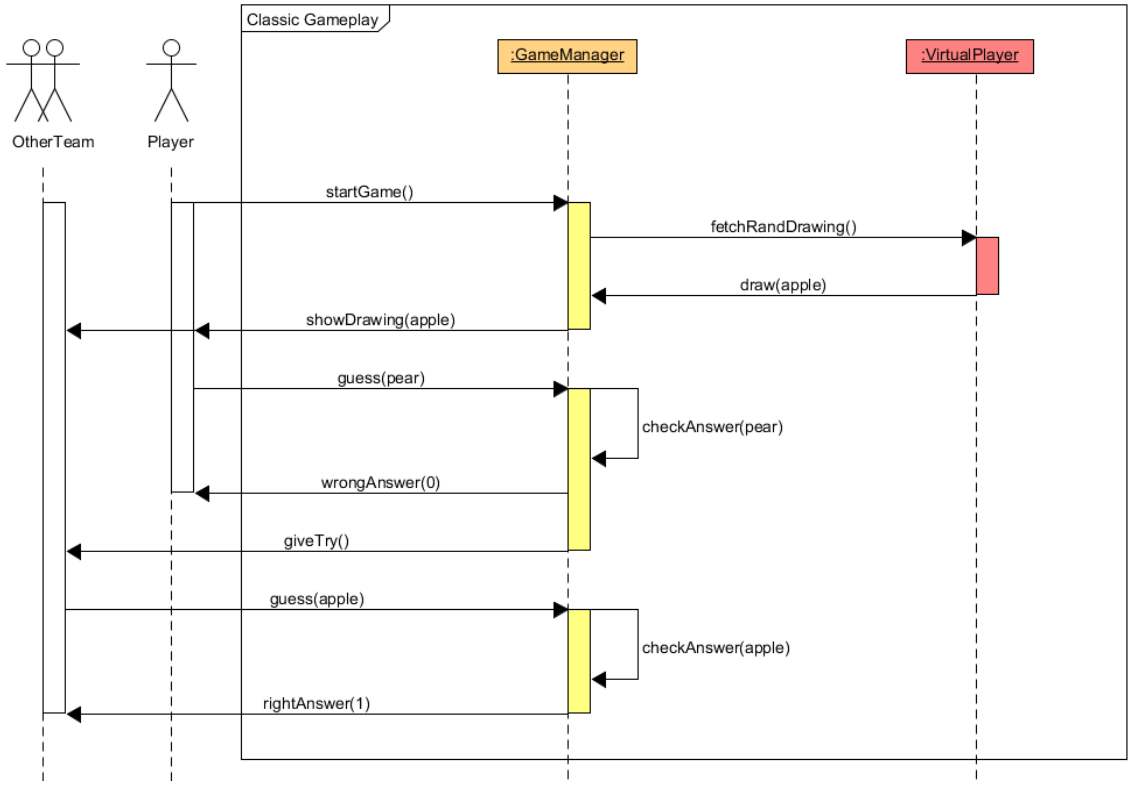

Figure 5.2: Diagramme de séquence pour le mode de jeu classique

5.3 Construction de jeu (mode assisté I)

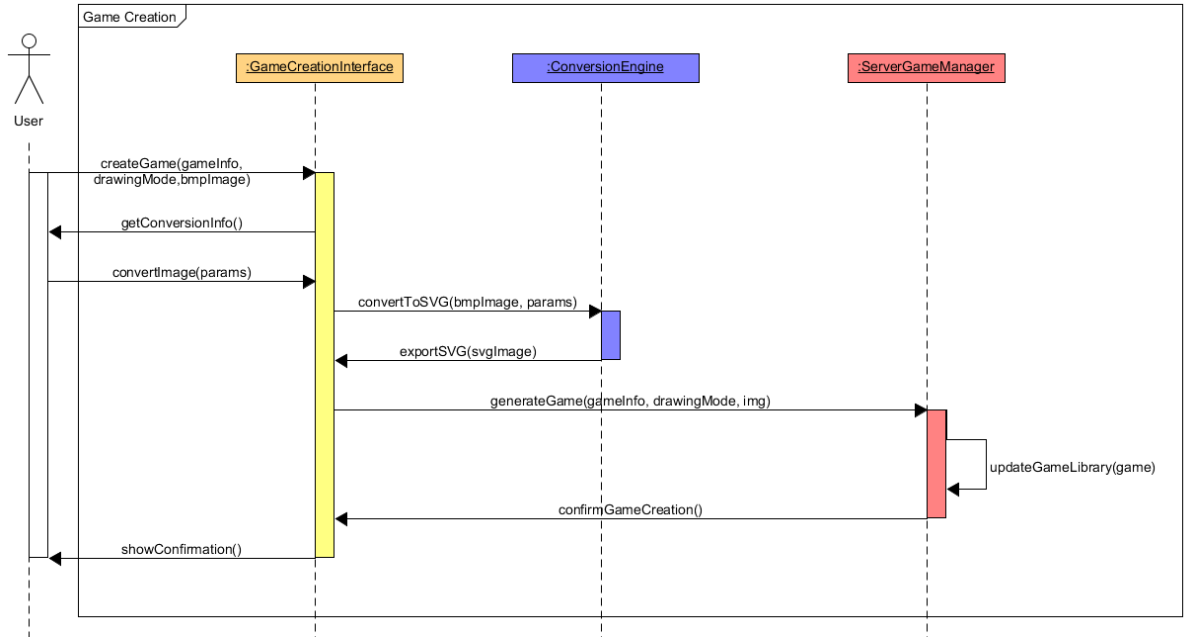

Figure 5.3: Diagramme de séquence pour la création de jeu

6. Vue de déploiement

Notre système est divisé en trois composantes physiques : le client lourd sur un ordinateur Windows 10, le client léger sur un appareil Android, et le serveur ainsi que la base de données qui sont hébergés par le service Microsoft Azure. Les deux clients communiquent de la même manière avec le serveur : par le biais de requêtes HTTP ainsi que le protocole SignalR offert par Microsoft qui permet la communication d'événements en temps réel.

Voici diagramme de déploiement qui illustre ces différentes composantes :

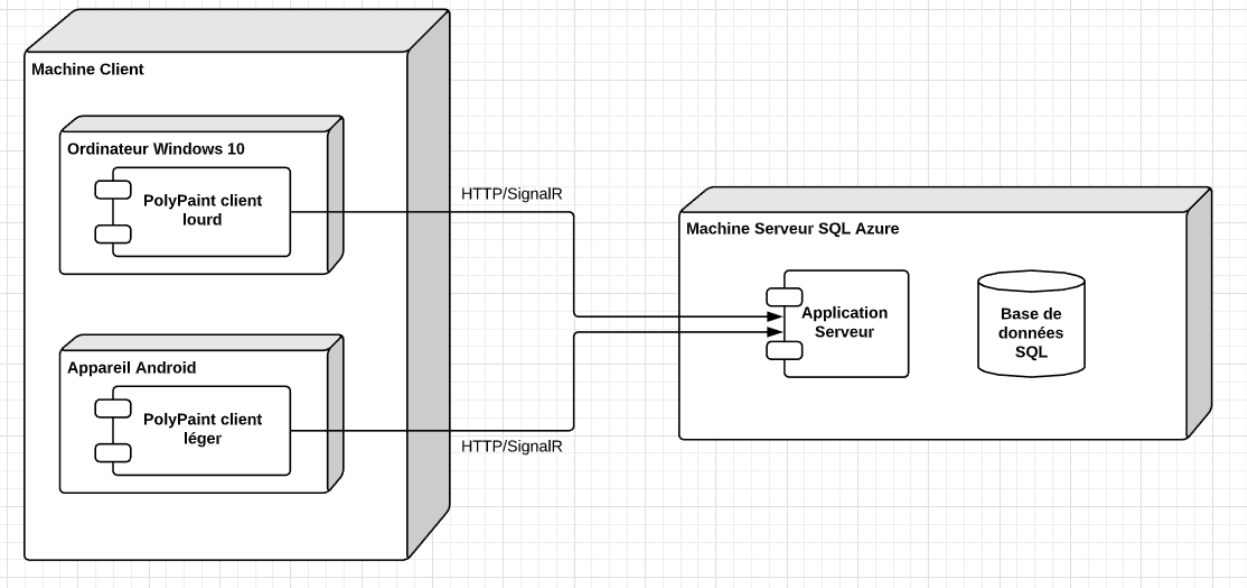

Figure 6.1: Diagramme de déploiement pour le système

7. Taille et performance

En termes de performance, notre système doit respecter minimalement les spécifications émises dans notre SRS. On doit donc s'assurer que la connexion puisse permettre un minimum de 8 joueurs, un chargement de partie inférieur à 3 secondes, l'affichage de messages lors du clavardage inférieur à 2 secondes, etc. Afin de bien respecter ces spécifications, nous utilisons des architectures logicielles et des librairies permettant la meilleure performance possible. Par exemple, étant donné que notre serveur est en .NET Core, nous utilisons la librairie SignalR de Microsoft pour notre communication en temps réel. Cette librairie est effectivement spécialement performante et compatible avec d'autres technologies Microsoft comme .NET Core et WPF.

Au niveau de la taille, c'est surtout pour le client léger que nous devons faire attention. Étant donné qu'on va utiliser celui-ci sur des appareils Android qui ont une capacité de stockage et une performance limitée, il faut mettre de l'emphase sur la légèreté de ce client afin d'assurer son bon fonctionnement.