Équipe 1

Reconnaissance automatique des matricules et compilation des notes pour optimiser le travail des enseignants

Document d'architecture logicielle

Version 2.0

Historique des révisions

Date	Version	Description	Auteur
2022-10-07	1.0	Première rédaction de l’architecture logicielle	Équipe 01
2022-12-07	2.0	Mise à jour de l’architecture logicielle	Équipe 01

Table des matières

Document d'architecture logicielle

1. Introduction

Dans ce document, on présente l'architecture logicielle de RMN. Tout d'abord, on commencera par présenter les objectifs et contraintes architecturaux qui possèdent un impact architectural sur notre logiciel. On met l'accent sur le serveur et le client web. Par la suite, la section 3 sera composée de diagrammes de cas d'utilisation pour montrer les différents types de cas que notre application devra prendre en compte lors de sa conception. Ensuite, la section 4 portera sur la vue logique de RMN à l'aide de diagrammes de paquetages de haut niveau et des diagrammes de classes. Puis, la section 5 sera sur la section de la vue des processus où on utilisera des diagrammes de séquences pour illustrer les différentes interactions entre les objets. À la section 6, elle sera décrite à l'aide de diagrammes de déploiement qui montrent la configuration des différentes composantes matérielles et virtuelles. Finalement, à la section 7, on présentera une description de la taille et de la performance de notre architecture logicielle.

2. Objectifs et contraintes architecturaux

L'objectif du projet est de développer une application de reconnaissance automatique des notes pour les étudiants. C'est-à-dire, il nous faudra concevoir une application pour permettre à plusieurs utilisateurs, principalement des professeurs ou évaluateurs, de compiler leurs copies d'examens en un fichier csv qui regroupera toutes les notes de leurs étudiants. Le logiciel doit être fonctionnel sur un ordinateur web. Il n'a donc pas besoin d'être fonctionnel sur une tablette ou un téléphone.

Notre architecture devra prendre en compte comment le module de reconnaissance automatique des notes a été fait. Effectivement, puisque le projet consiste à construire une application web, donc le cœur contient ce module de reconnaissance, il nous faudra modifier légèrement ce dernier en lui apportant certains changements nécessaires pour répondre aux exigences.

Quant aux contraintes, nous sommes contraints par trois échéances. Le premier étant la remise d'un prototype pour le 7 octobre. La deuxième échéance est le 23 novembre pour la remise Beta et finalement, la remise du produit final est pour le 7 décembre. Il faudra alors gérer notre temps adéquatement pour bien construire l'architecture de notre logiciel.

Le serveur occupe une place importante dans notre architecture, car il s'occupe de deux principales fonctions. La première est de gérer les données des utilisateurs de l'application. La deuxième est d'envoyer à Redis l'identifiant de la tâche et à Mongodb la tâche en tant que telle provenant de la requête http de l'application web. Pour ajouter une couche de sécurité et ne pas avoir des personnes aléatoires pouvant y accéder, nous avons ajouté un système d'authentification pour utiliser le logiciel. Le serveur sera codé en python.

Nous aurons un exécuteur de tâches, qui comme le nom l'indique, se chargera de rouler le module de reconnaissance des notes sur les copies des étudiants. Pour ce faire, il va devoir accéder au bon fichier de notes. Ce dernier sera emmagasiné dans Mongodb avec un identifiant de la tâche unique à lui (id étant généré par le serveur par la requête du web). Pour pouvoir accéder à l'identifiant de la tâche que l'on veut dans Mongodb, l'exécuteur de tâches va devoir communiquer avec Redis pour l'acquérir.

Mongodb s'occupera de recevoir une tâche du serveur, lorsqu'un utilisateur lancera la requête pour le module de reconnaissance. Il emmagasine l'identifiant de la tâche, du user, du gabarit, du fichier de notes, du fichier zip, le queued_time et le statut de la tâche.

Finalement, nous aurons Kubernetes qui se chargera de mettre l'exécuteur de tâches dans des pods pour rendre notre application plus extensible. Ceci est dans le cas où plusieurs utilisateurs font des requêtes en même temps et que l'on ne souhaite pas faire attendre les autres pour chaque traitement de requête fait avant eux.

Nous avons comme objectif architectural de réaliser une architecture robuste, structurée et modulable afin de favoriser la modification de l'application web dans le futur par d'autres personnes.

Parmi les différents points mentionnés ci-haut, la partie interface du serveur sera habillée d'un décorateur REST API

pour recevoir des requêtes de l'application web et répondre à ceux-ci selon des codes de réponses définies et structurées.

3. Vue des cas d'utilisation

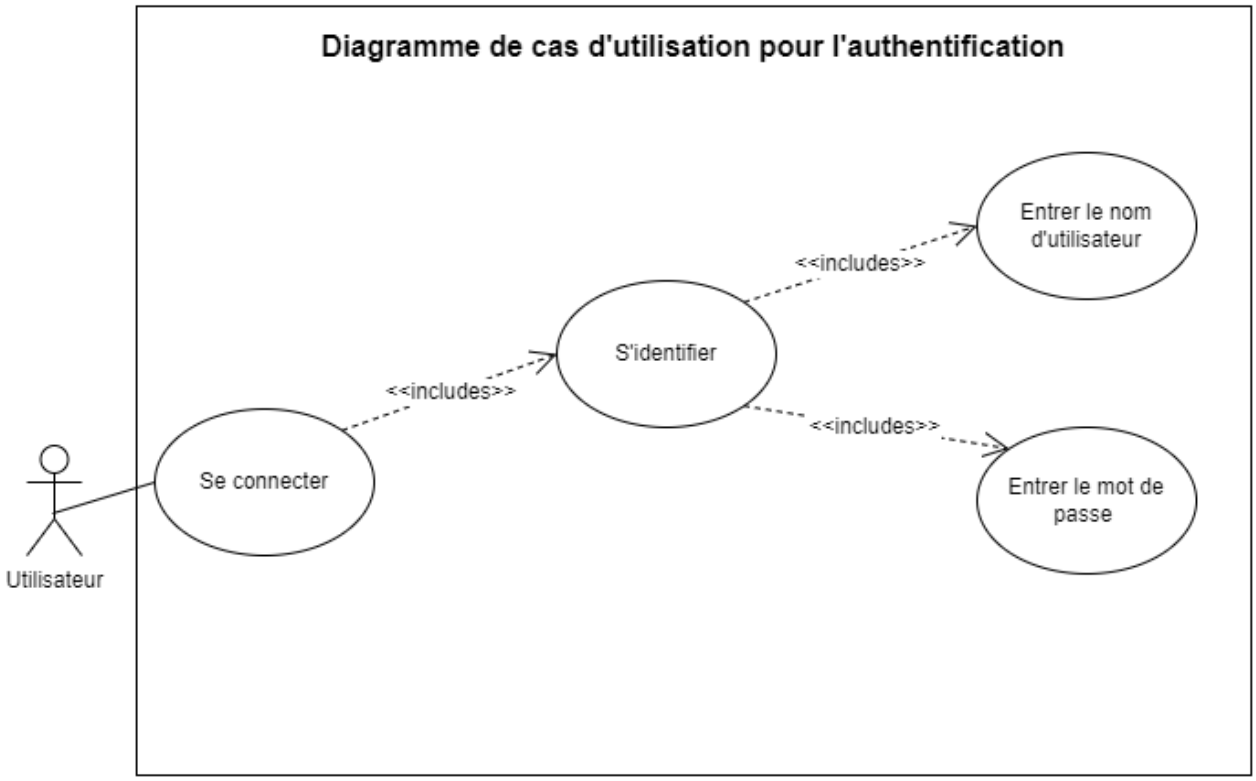

Figure 1: Diagramme de cas d'utilisation pour l'authentification

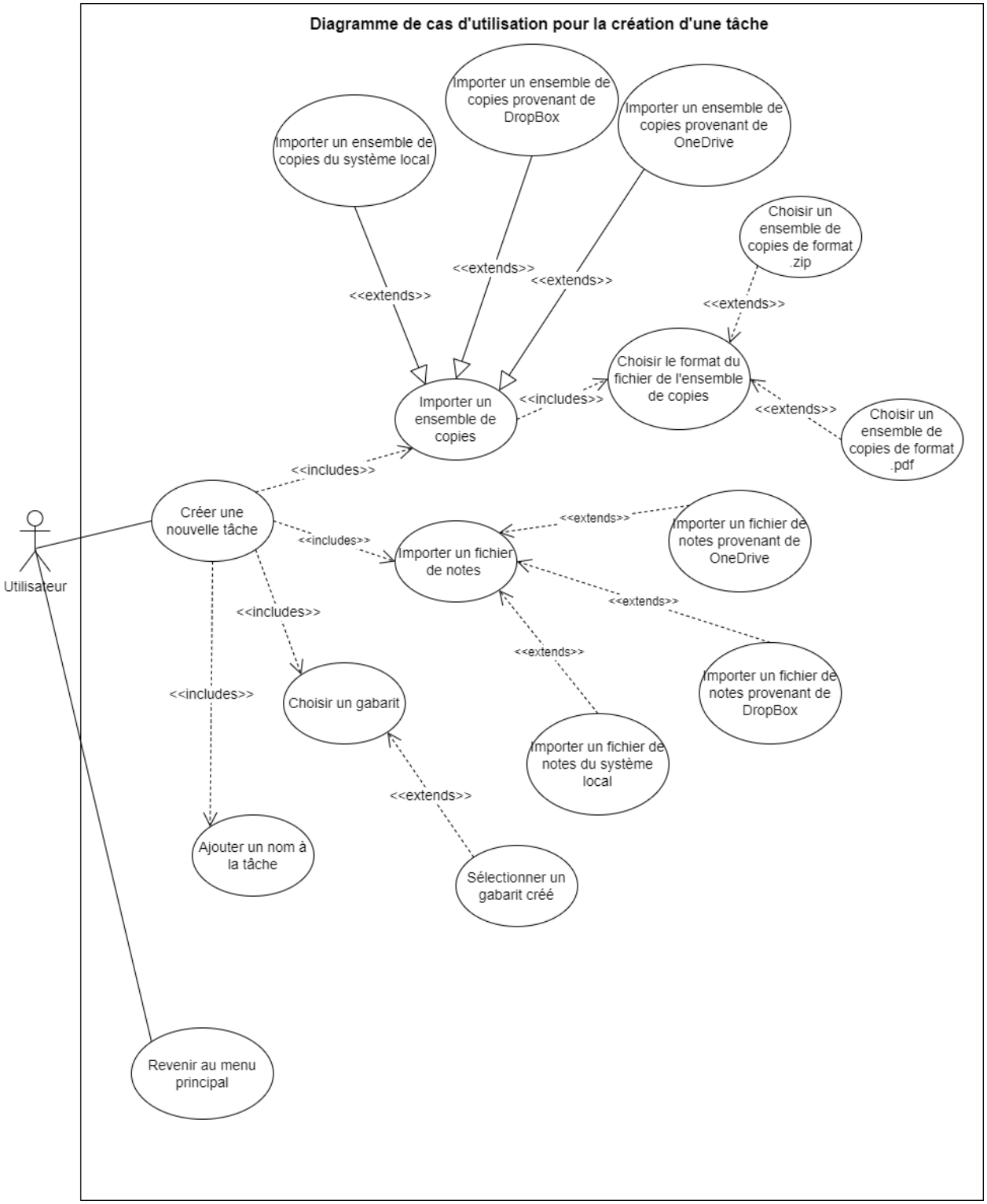

Figure 2: Diagramme de cas d'utilisation pour la création d'une tâche

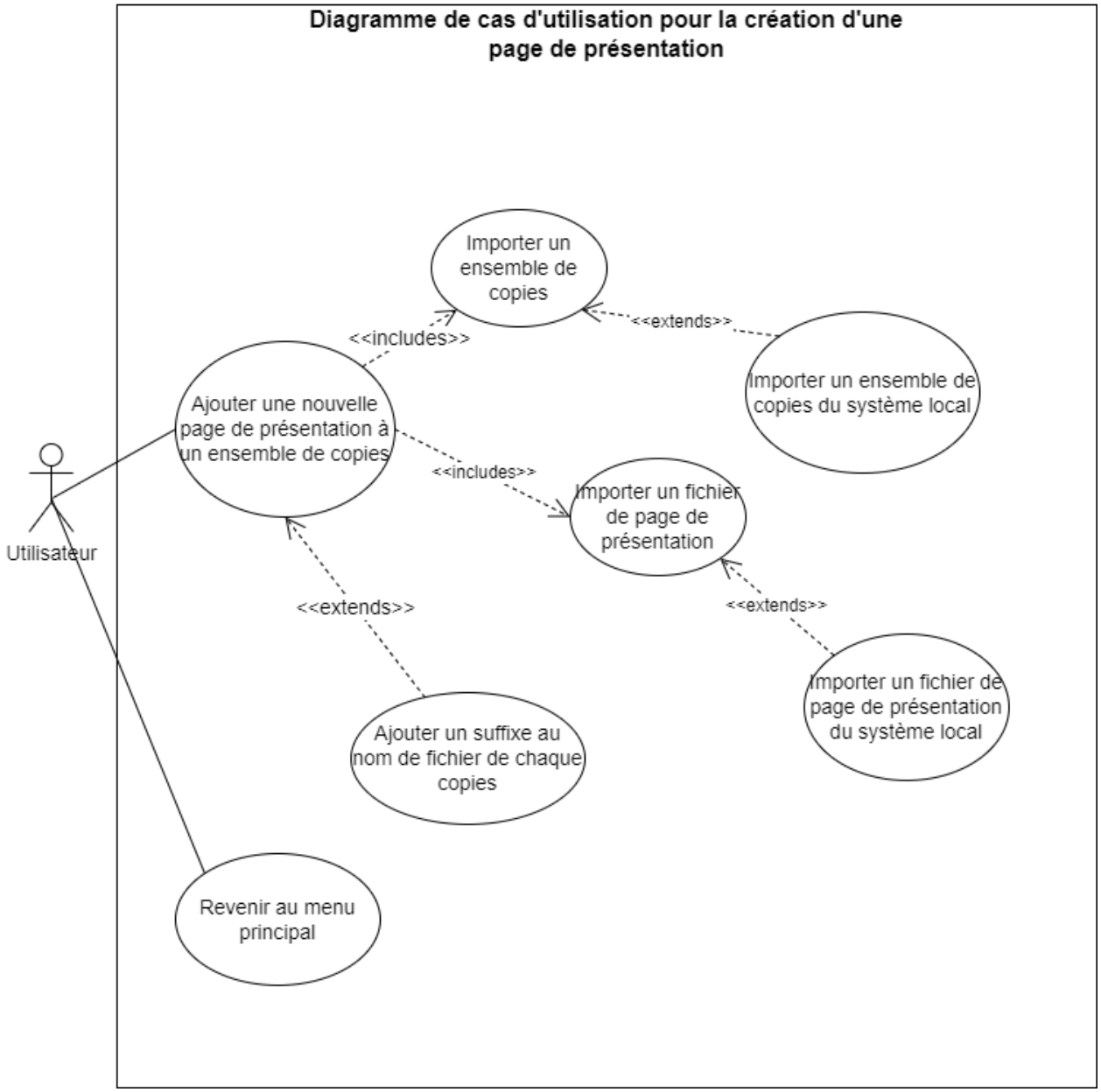

Figure 3: Diagramme de cas pour la création d'une page de présentation

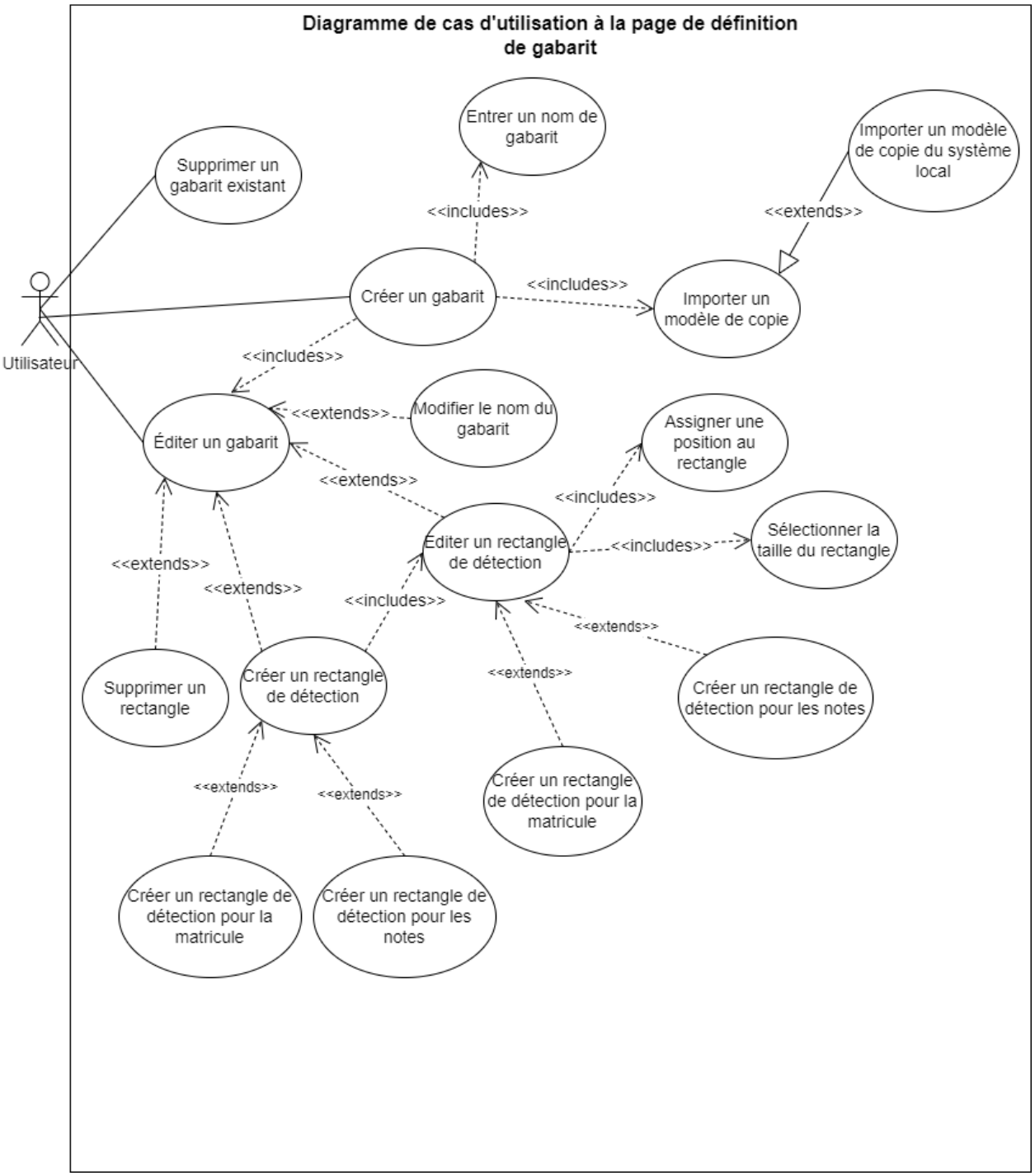

Figure 4: Diagramme de cas d'utilisation à la page de définition de gabarit

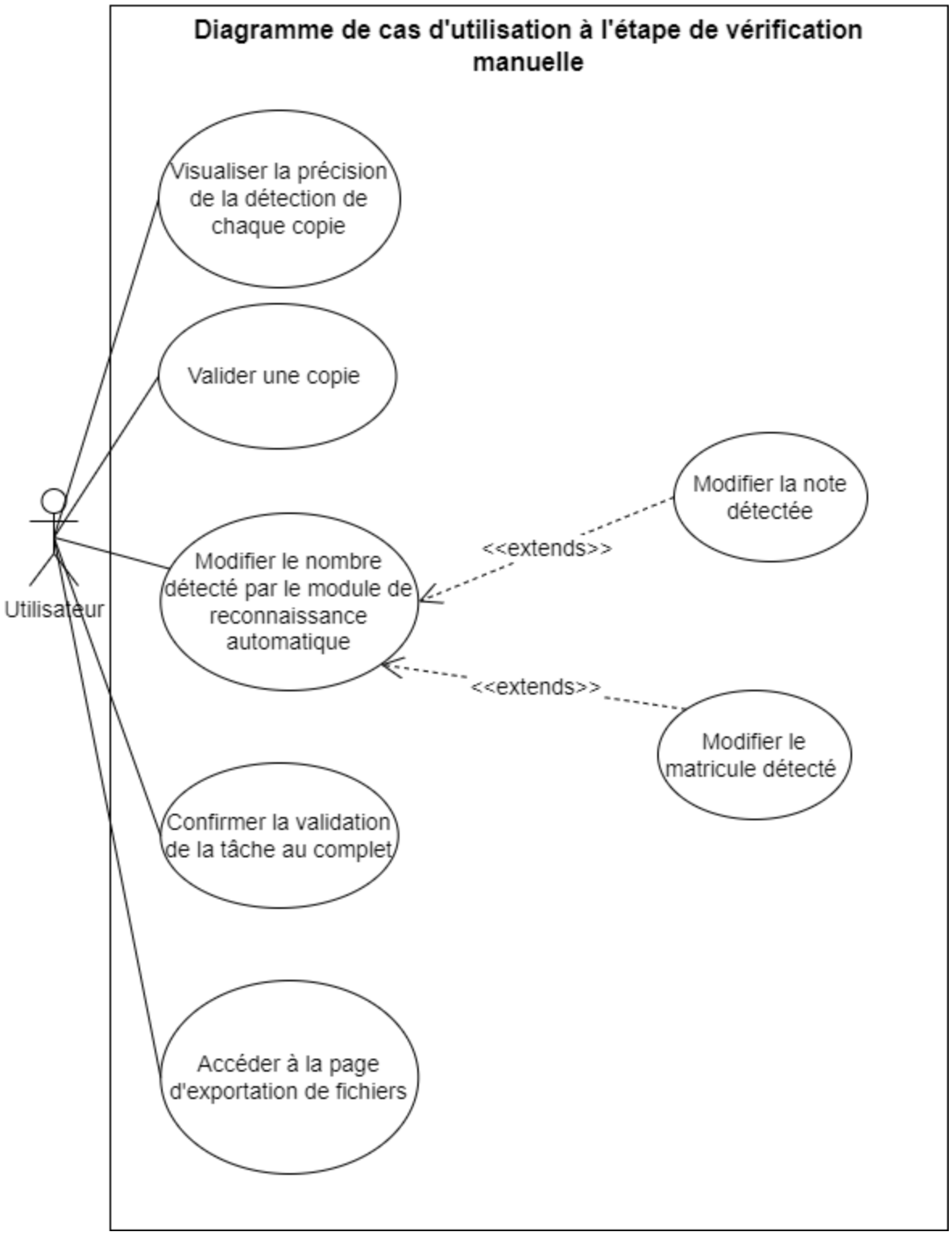

Figure 5: Diagramme de cas d'utilisation à l'étape de vérification manuelle

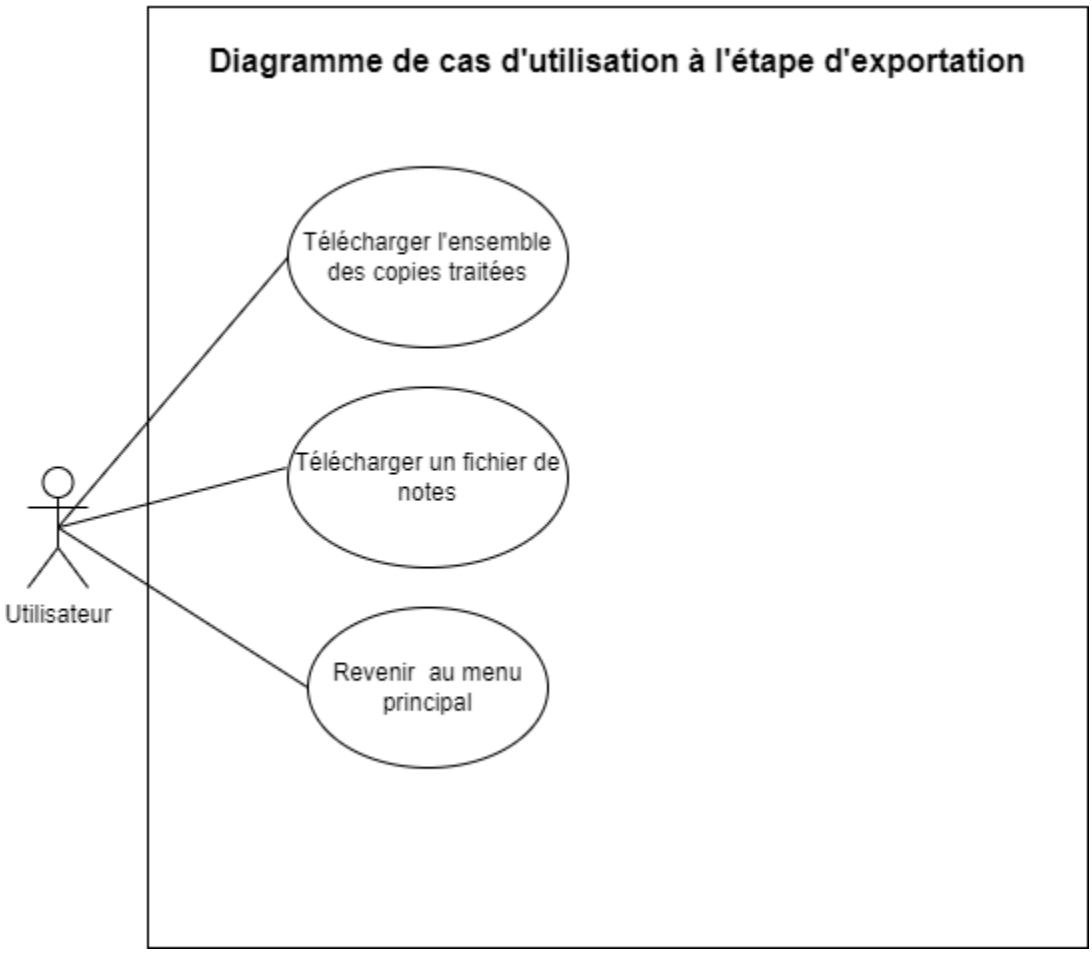

Figure 6: Diagramme de cas d'utilisation à l'étape d'exportation

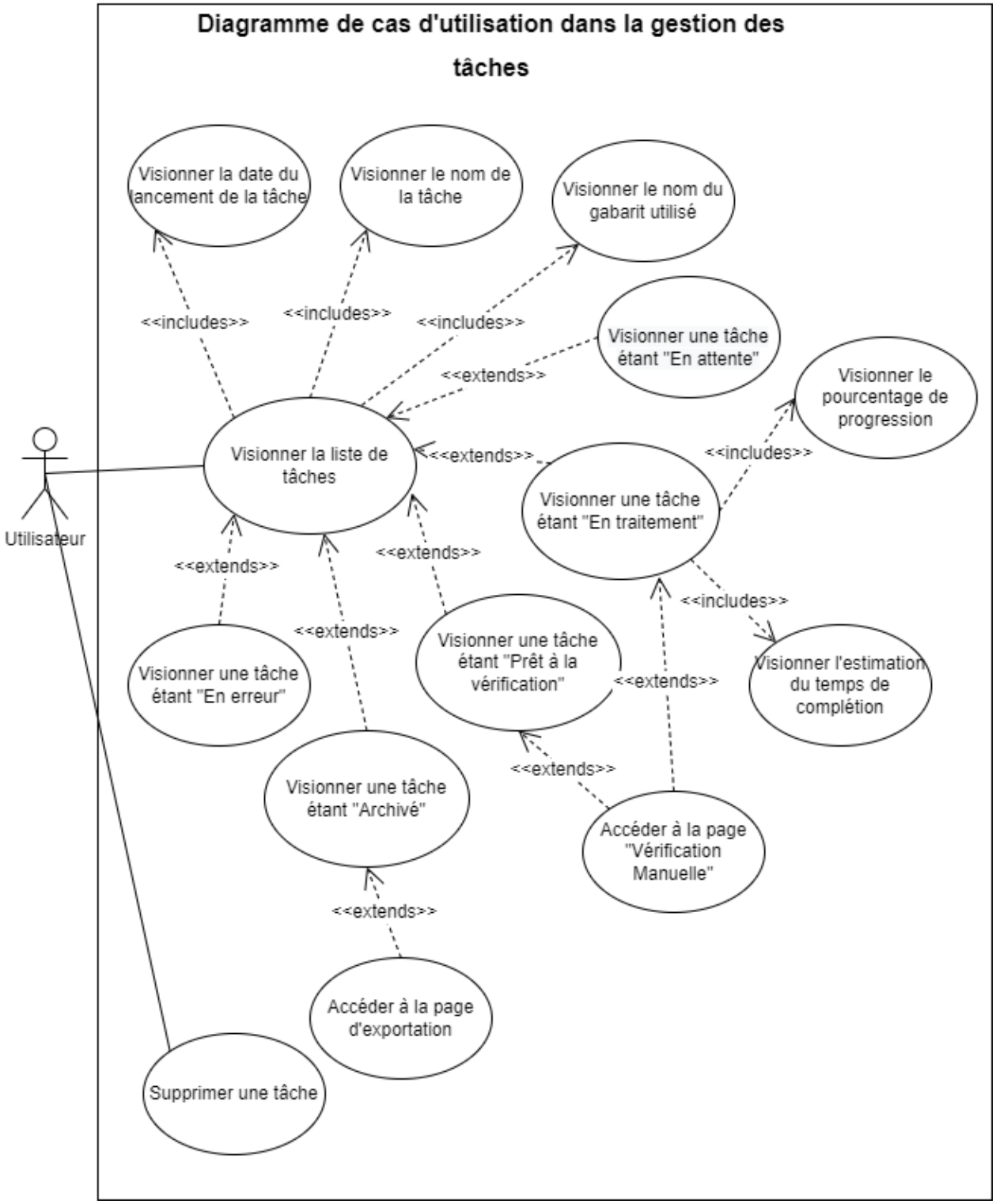

Figure 7: Diagramme de cas d'utilisation dans la gestion des tâches

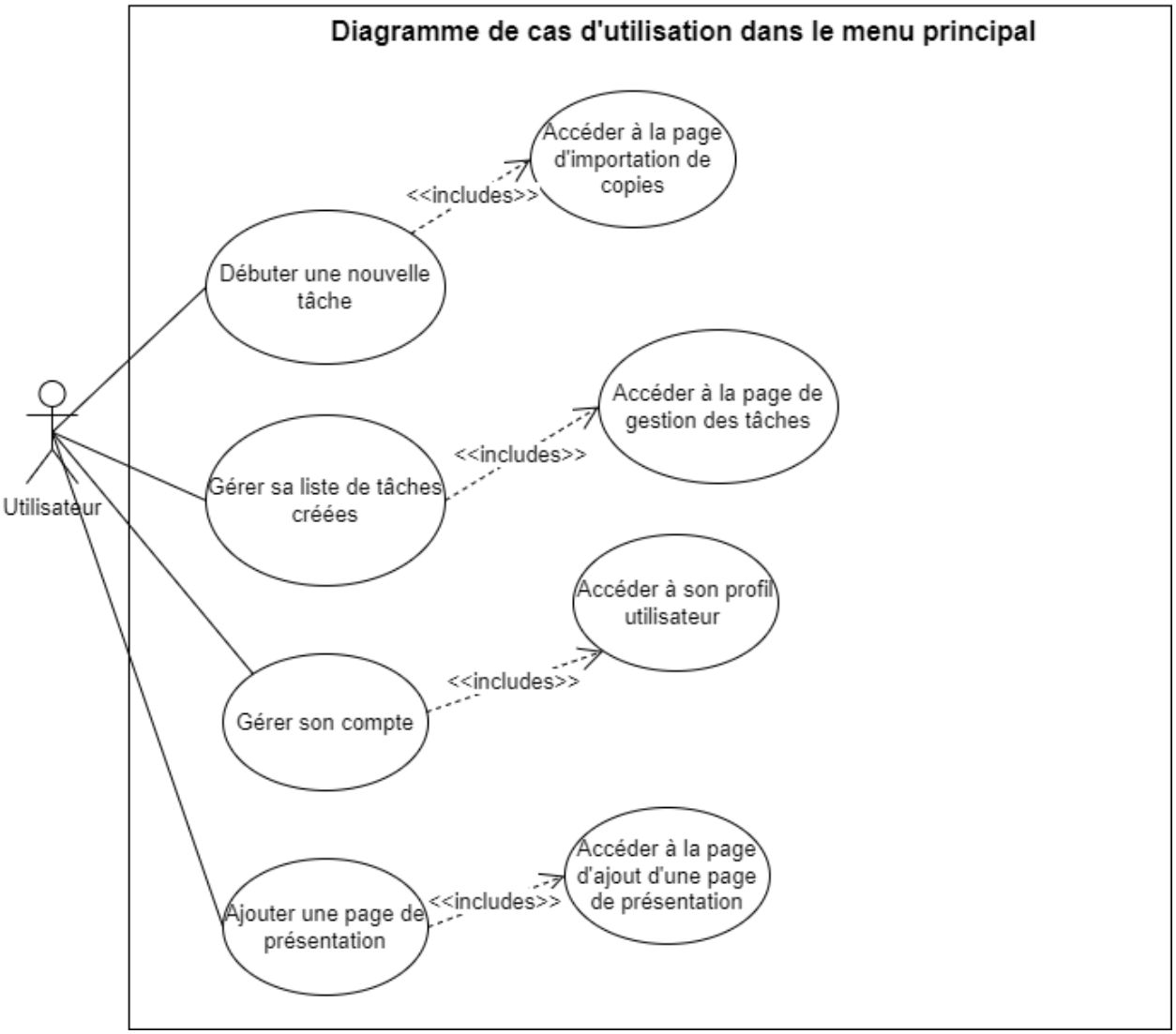

Figure 8: Diagramme de cas d'utilisation dans le menu principal

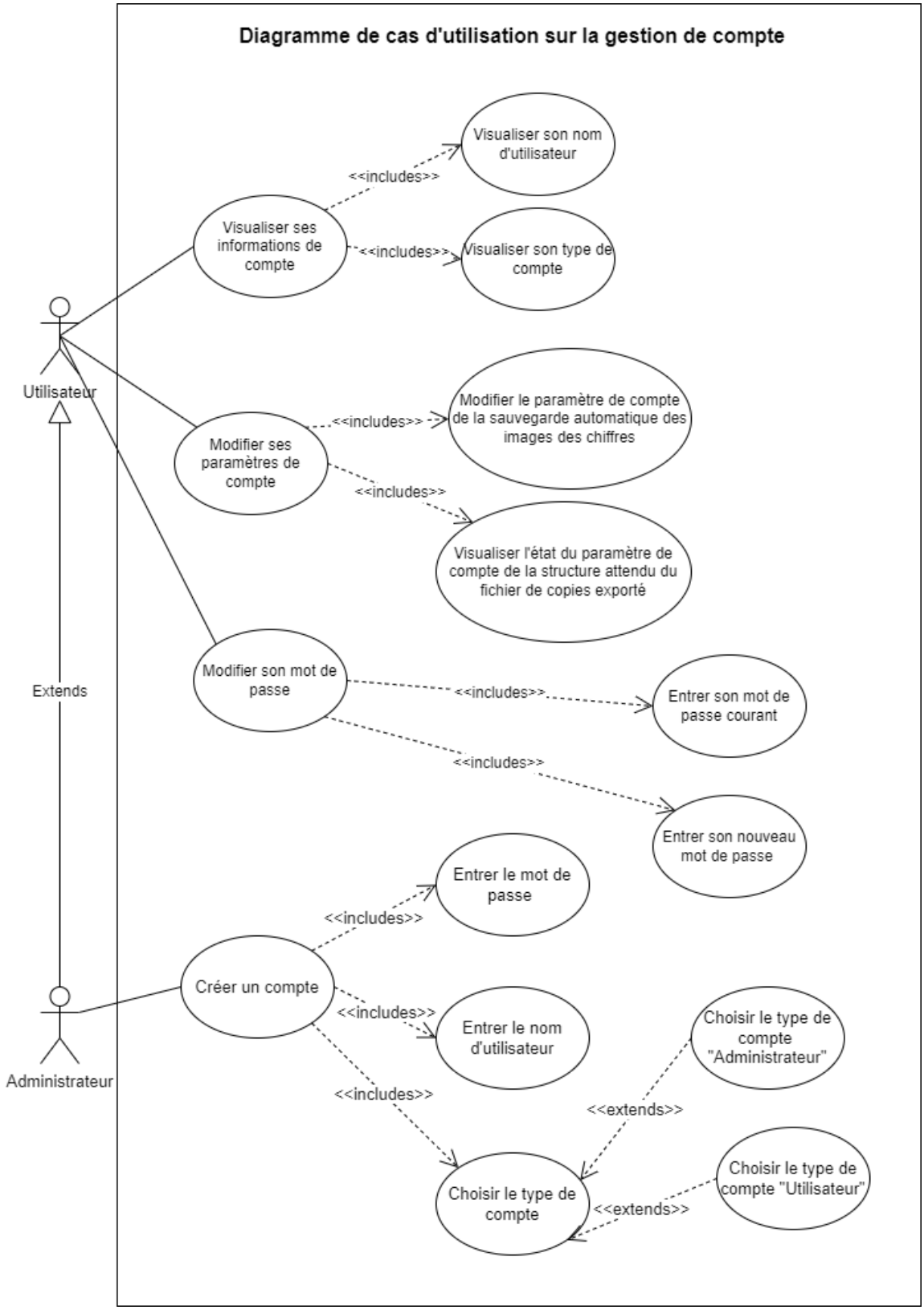

Figure 9: Diagramme de cas d'utilisation sur la gestion de compte

4. Vue logique

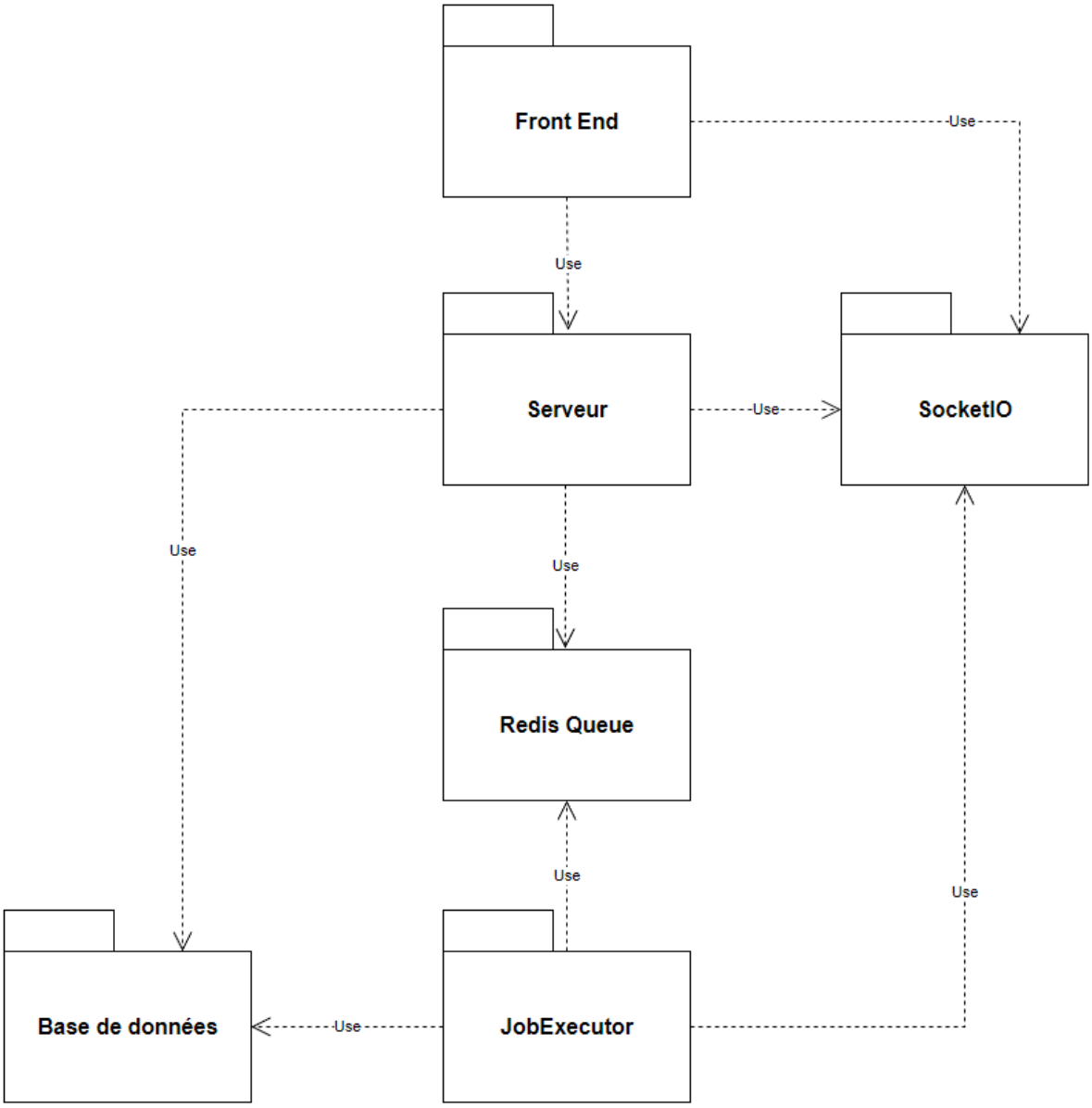

Figure 10: Diagramme de paquetage global du système

Front End
Ce paquetage contient les paquetages du client.

Serveur
Ce paquetage contient les paquetages du serveur.

JobExecutor
Ce paquetage contient les paquetages du module qui permet d'effectuer la

reconnaissance.

SocketIO
Ce paquetage contient les paquetages du serveur SocketIO.

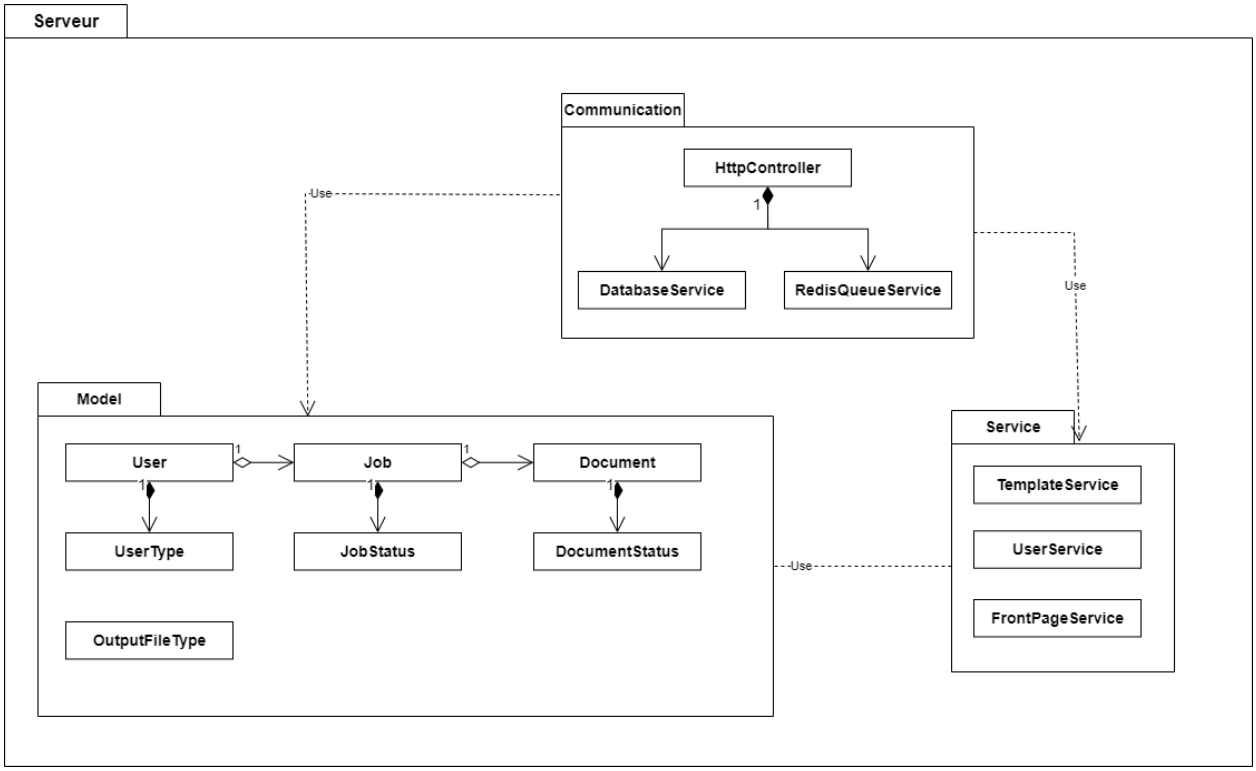

Figure 11: Diagramme de paquetage et de classe du serveur

Communication
Ce paquetage contient les services qui permettent de communiquer avec des services externes (client, base de données et la queue Redis).

Model
Ce paquetage contient toutes les classes permettant de structurer les données.

Service
Ce paquetage contient tous les services.

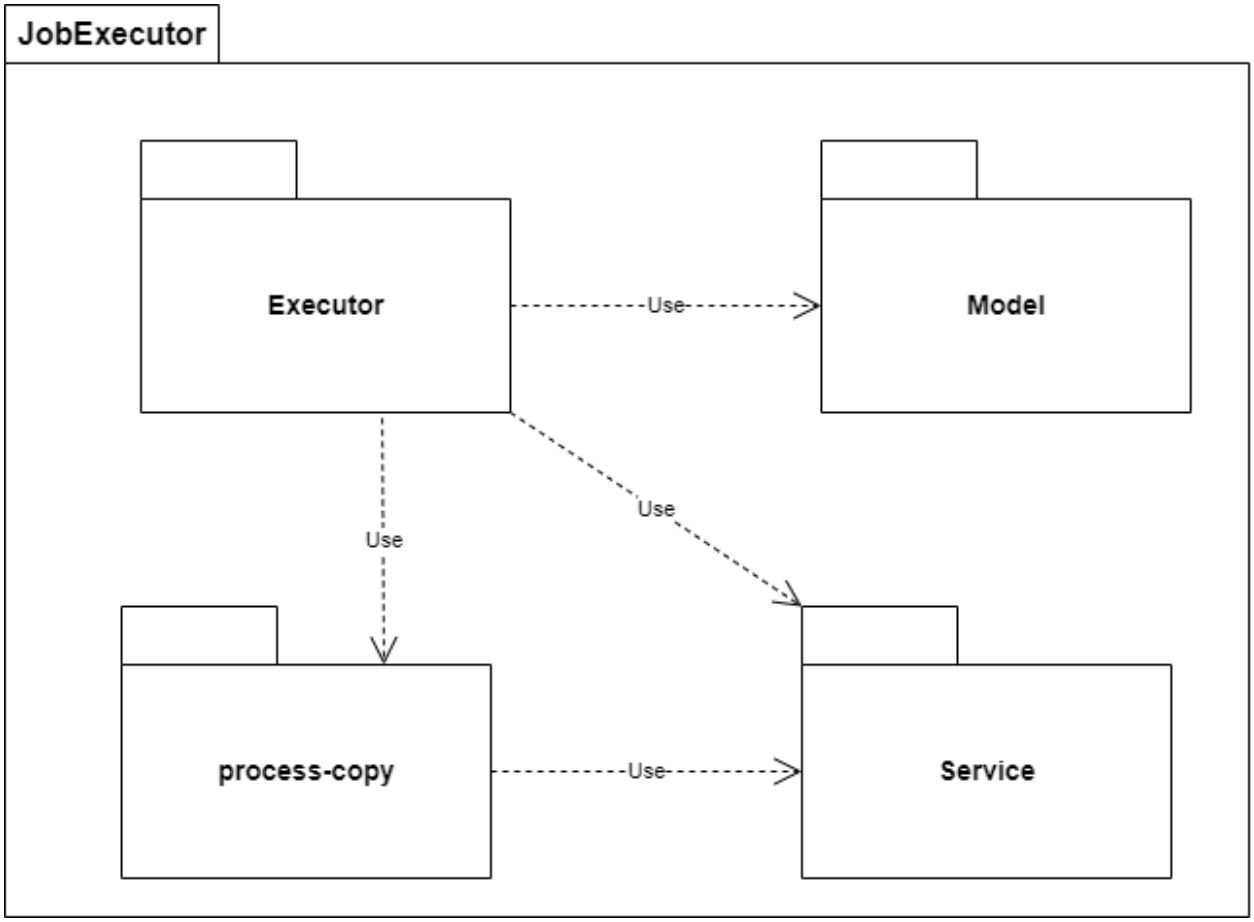

Figure 12: Diagramme de paquetage du JobExecutor

Executor
Ce paquetage contient l'ensemble des étapes de l'exécution d'un traitement d'une tâche.

Service
Ce paquetage contient les services qui permettent de communiquer avec des entités externes (base de données et la queue Redis).

Model
Ce paquetage contient toutes les classes permettant de structurer les données.

process-copy
Ce paquetage contient toutes les classes du module de reconnaissance de matricule et de notes développé par Antoine Legrain.

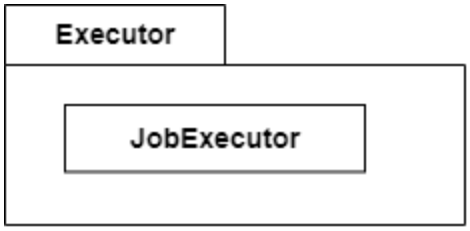

Figure 13: Diagramme de classe du paquetage « Executor

Figure 14: Diagramme de classe du paquetage Service

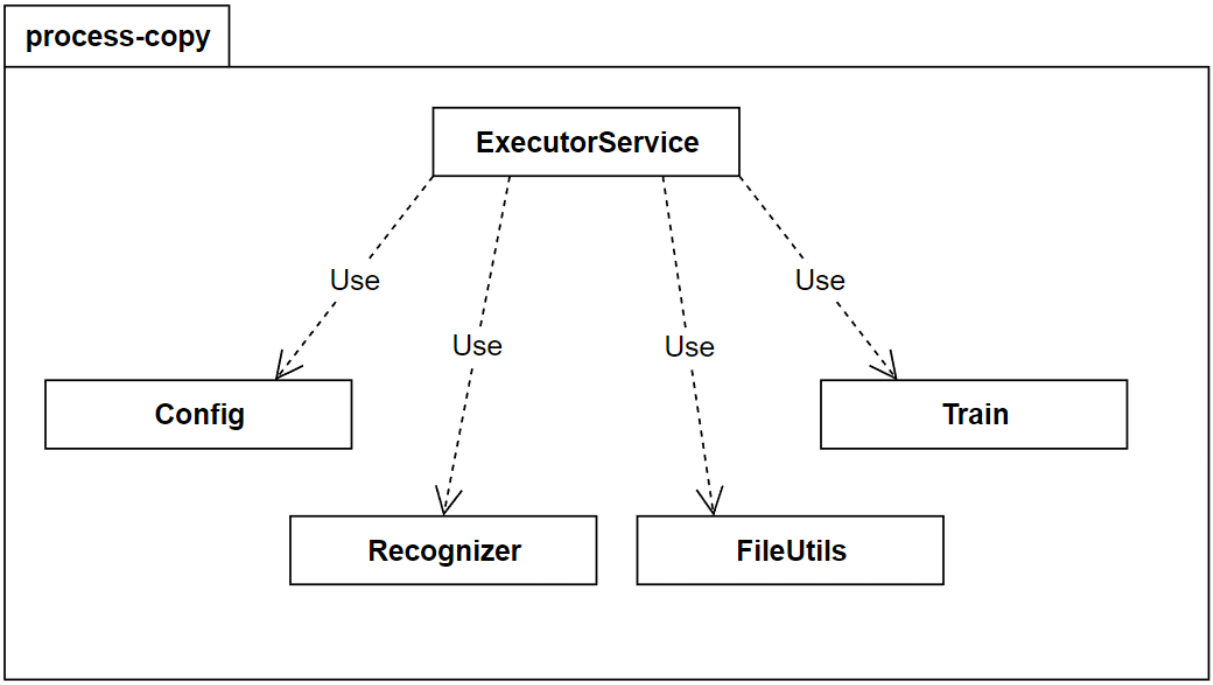

Figure 15: Diagramme de classe du paquetage « process-copy »

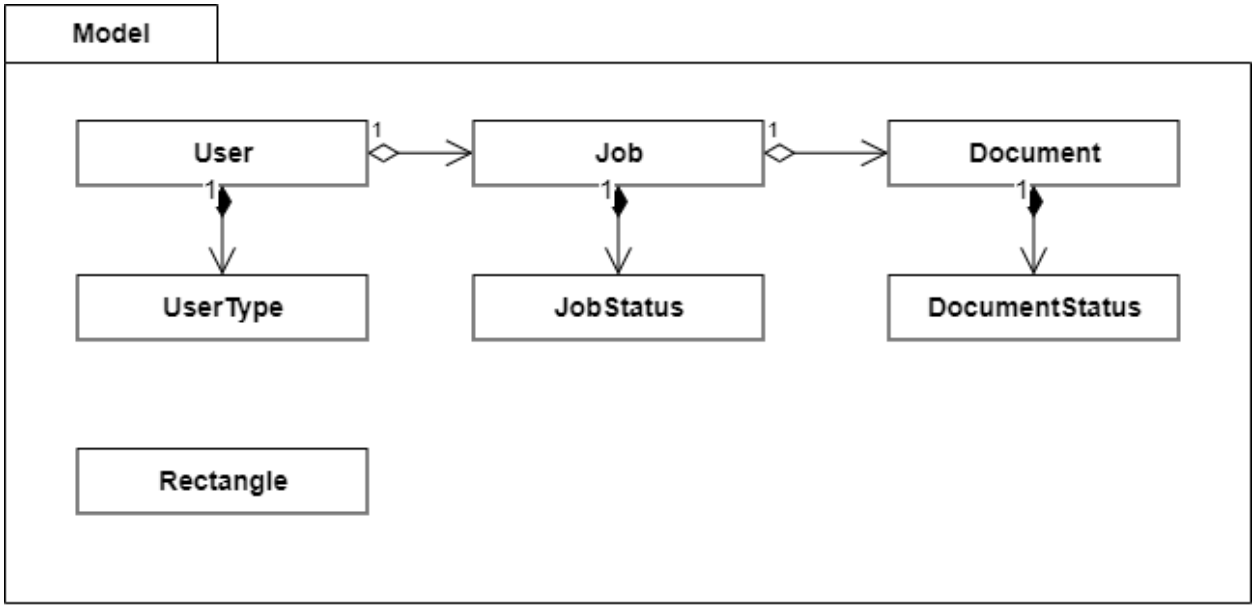

Figure 16: Diagramme de classe du paquetage « Model »

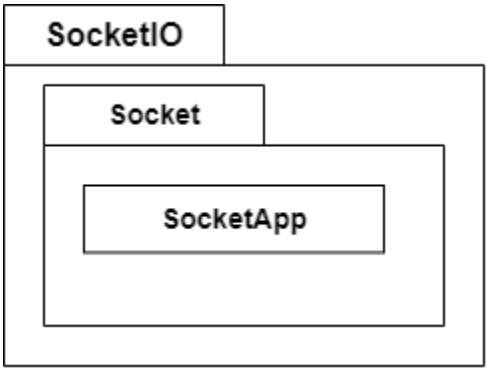

Figure 17: Diagramme de classe du paquetage « Model »

Socket
Ce paquetage contient l'ensemble de la logique du serveur SocketIO.

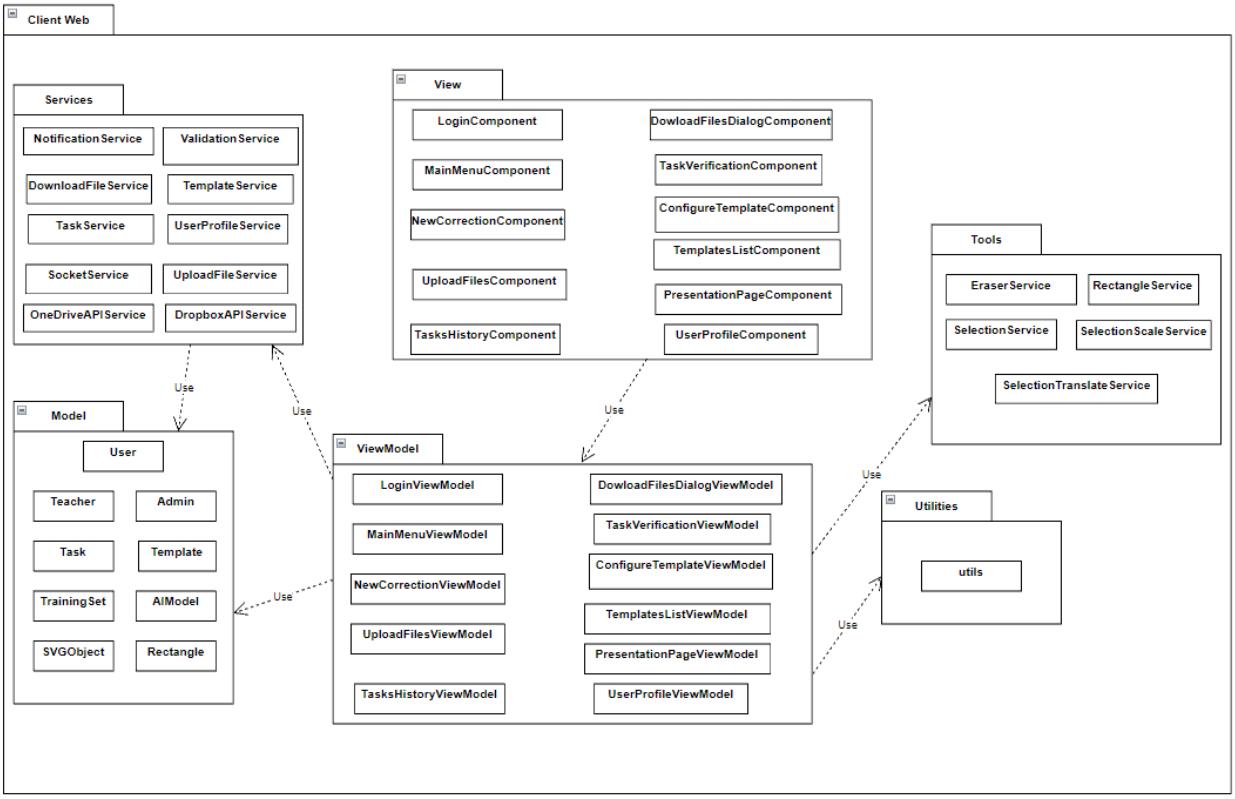

Figure 18: Diagramme de paquetage pour le client web

View
Ce paquetage contient tous les éléments de vue du client web.

ViewModel
Ce paquetage contient les classes qui s'occupent de la logique des vues.

Model
Ce paquetage contient les modèles utilisés pour sauvegarder les données provenant du serveur pour chaque utilisateur.

Service
Ce paquetage contient les classes qui fournissent un service tel que les informations de l'utilisateur ou téléverser/télécharger des fichiers.

Tools
Ce paquetage contient la logique des outils utilisés durant la configuration d'un gabarit.

Utilities
Ce paquetage contient la logique des interfaces et des constantes utiles pour client web.

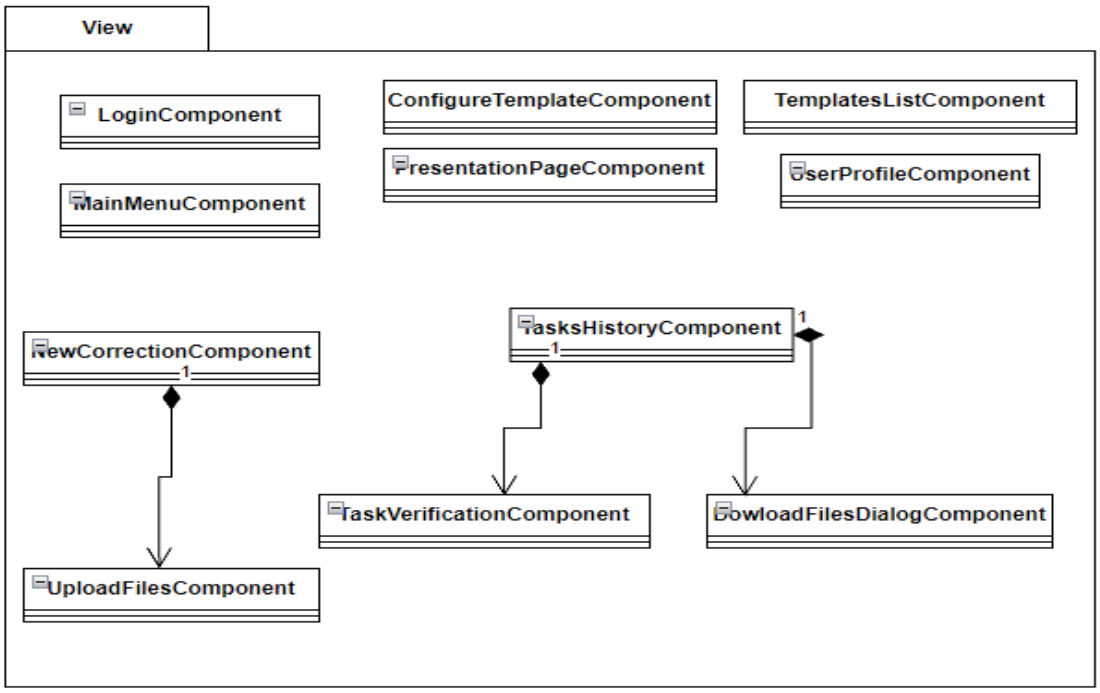

Figure 19: Diagramme de classe du paquetage View

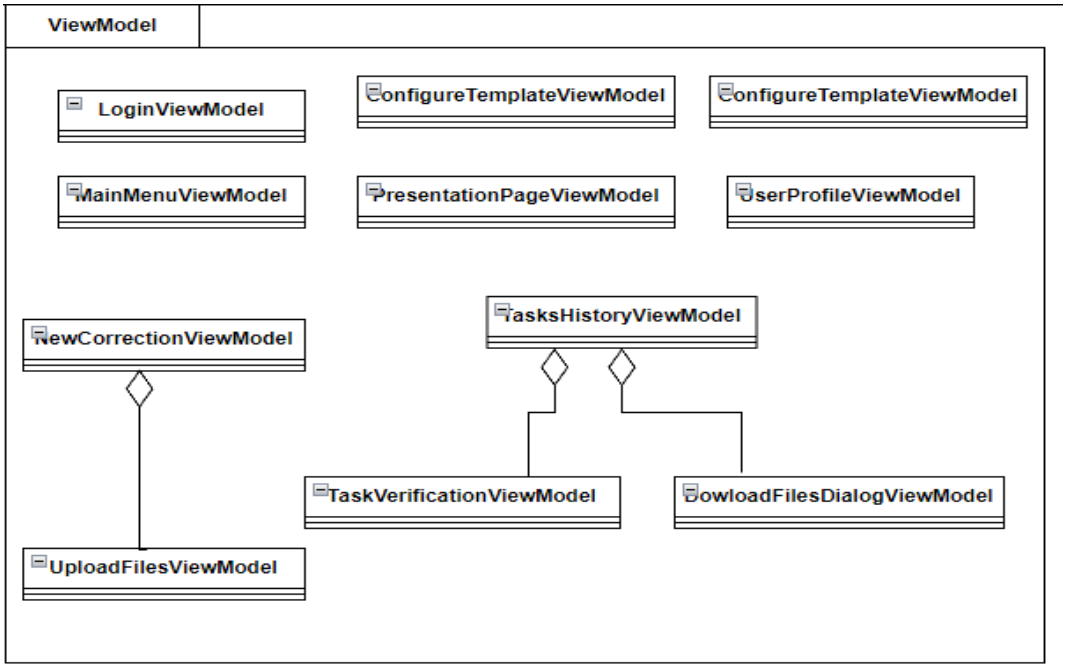

Figure 20: Diagramme de classe du paquetage ViewModel

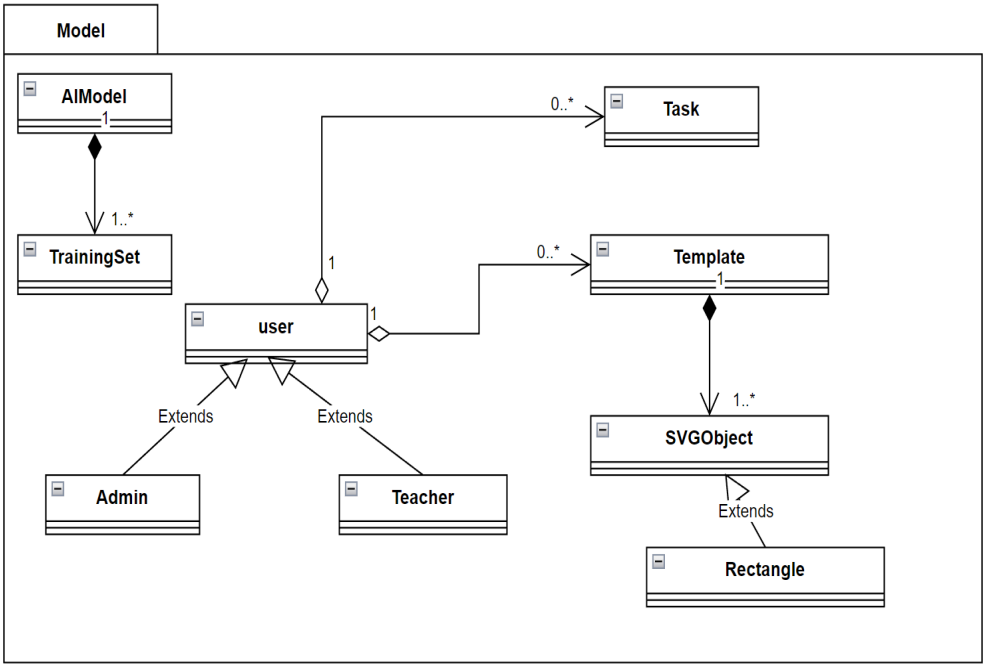

Figure 21: Diagramme de classe du paquetage Model

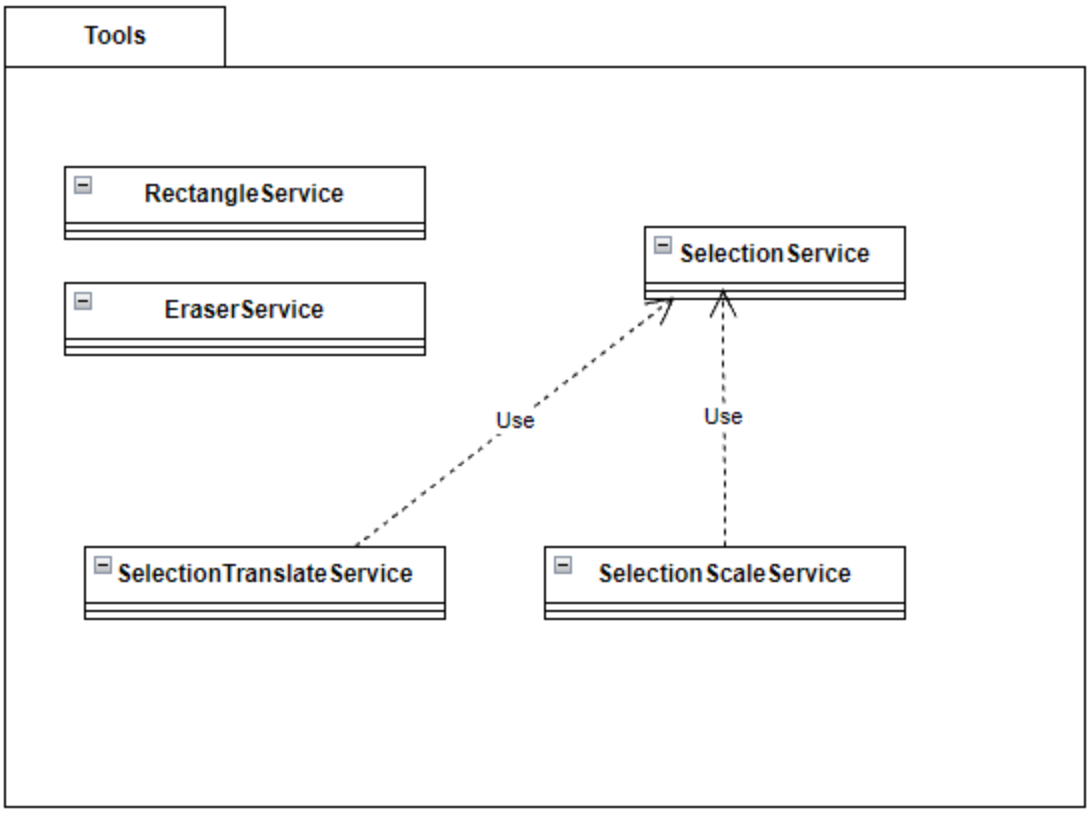

Figure 22: Diagramme de classe du paquetage Tools

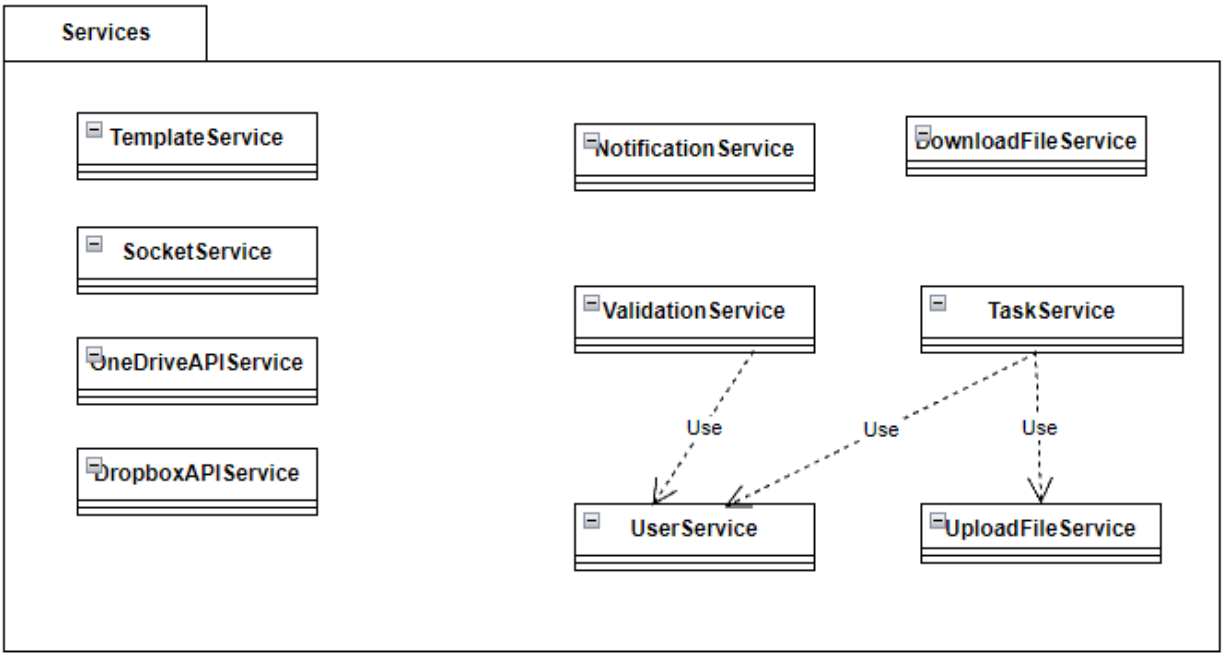

Figure 23: Diagramme de classe du paquetage Services

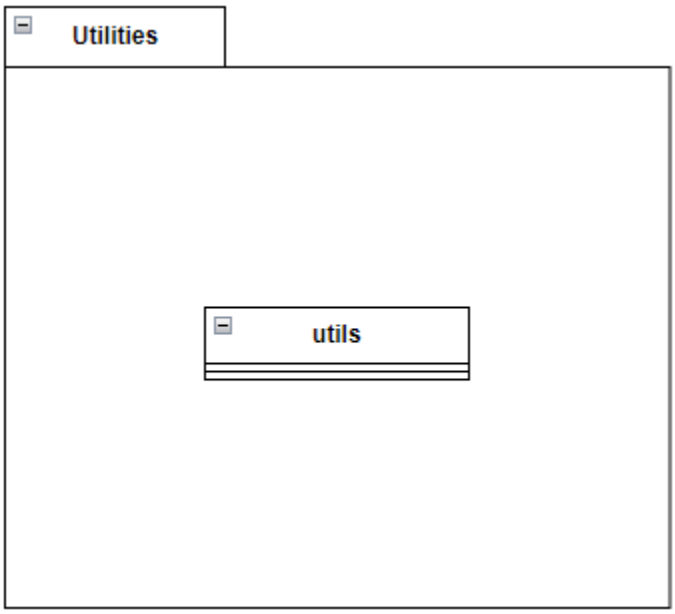

Figure 24: Diagramme de classe du paquetage utilities

5. Vue des processus

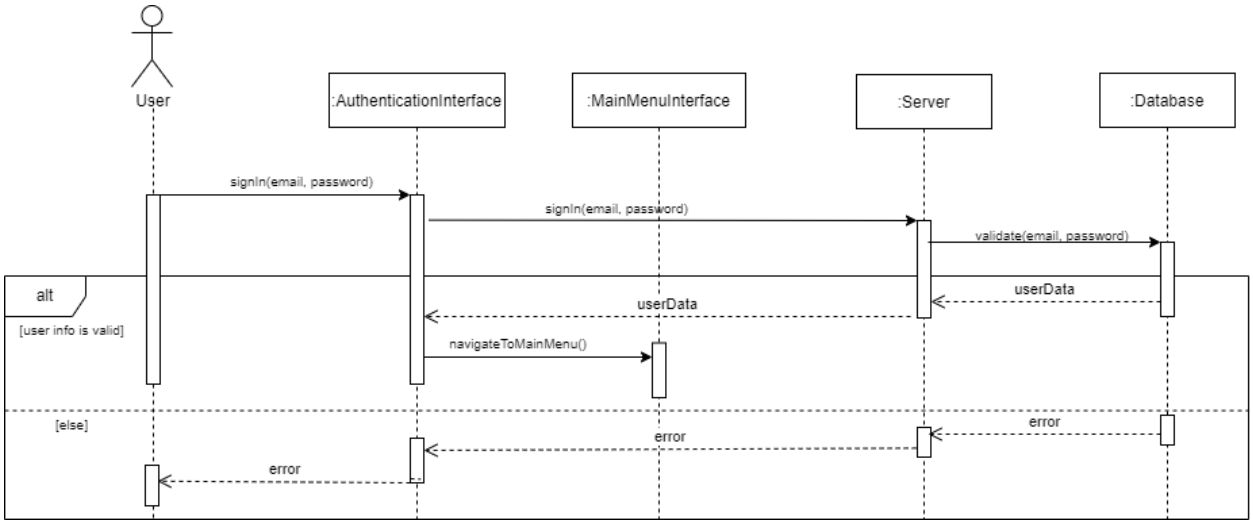

Figure 25: Diagramme de séquence pour la connexion à la plateforme

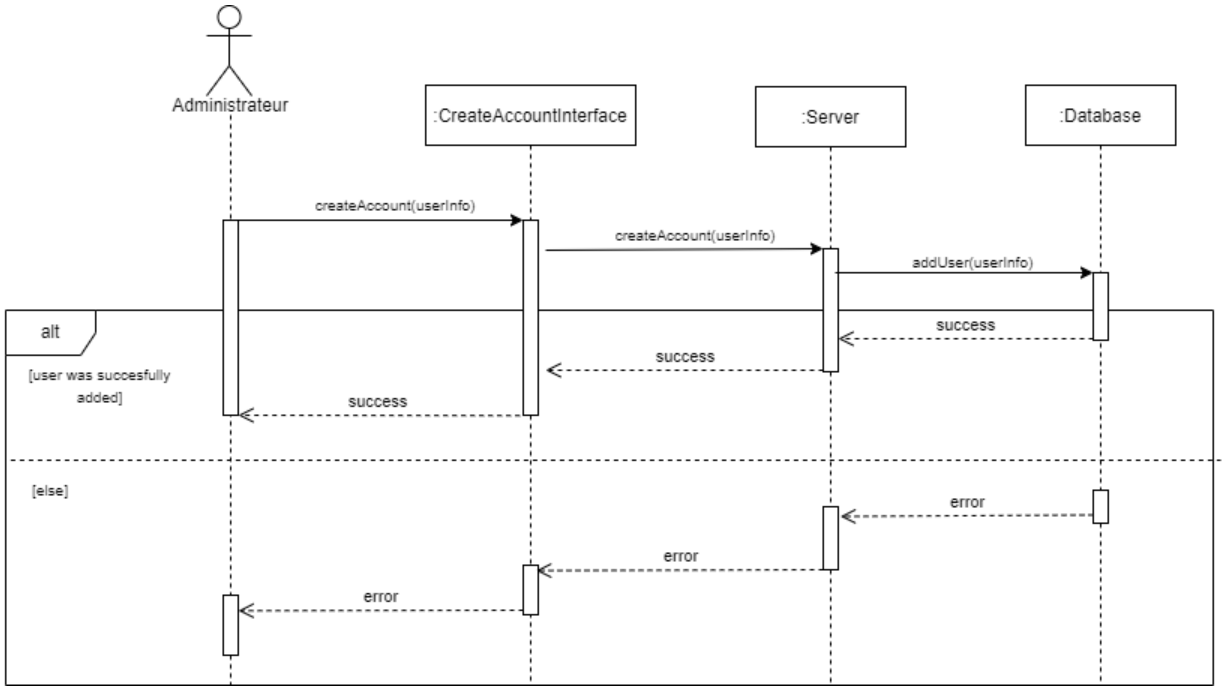

Figure 26: Diagramme de séquence pour la création d'un compte

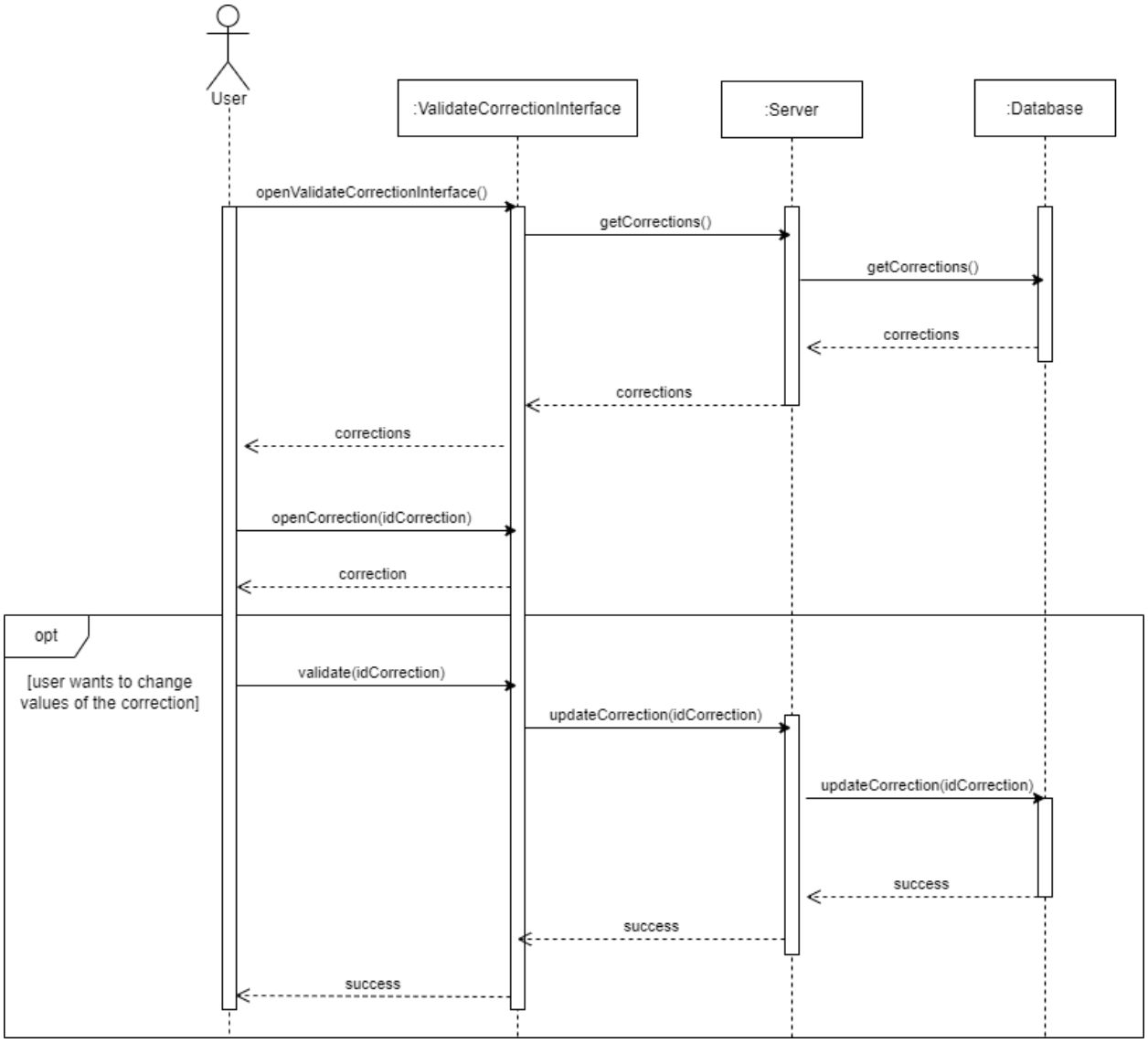

Figure 27: Diagramme de séquence pour la validation d'une correction

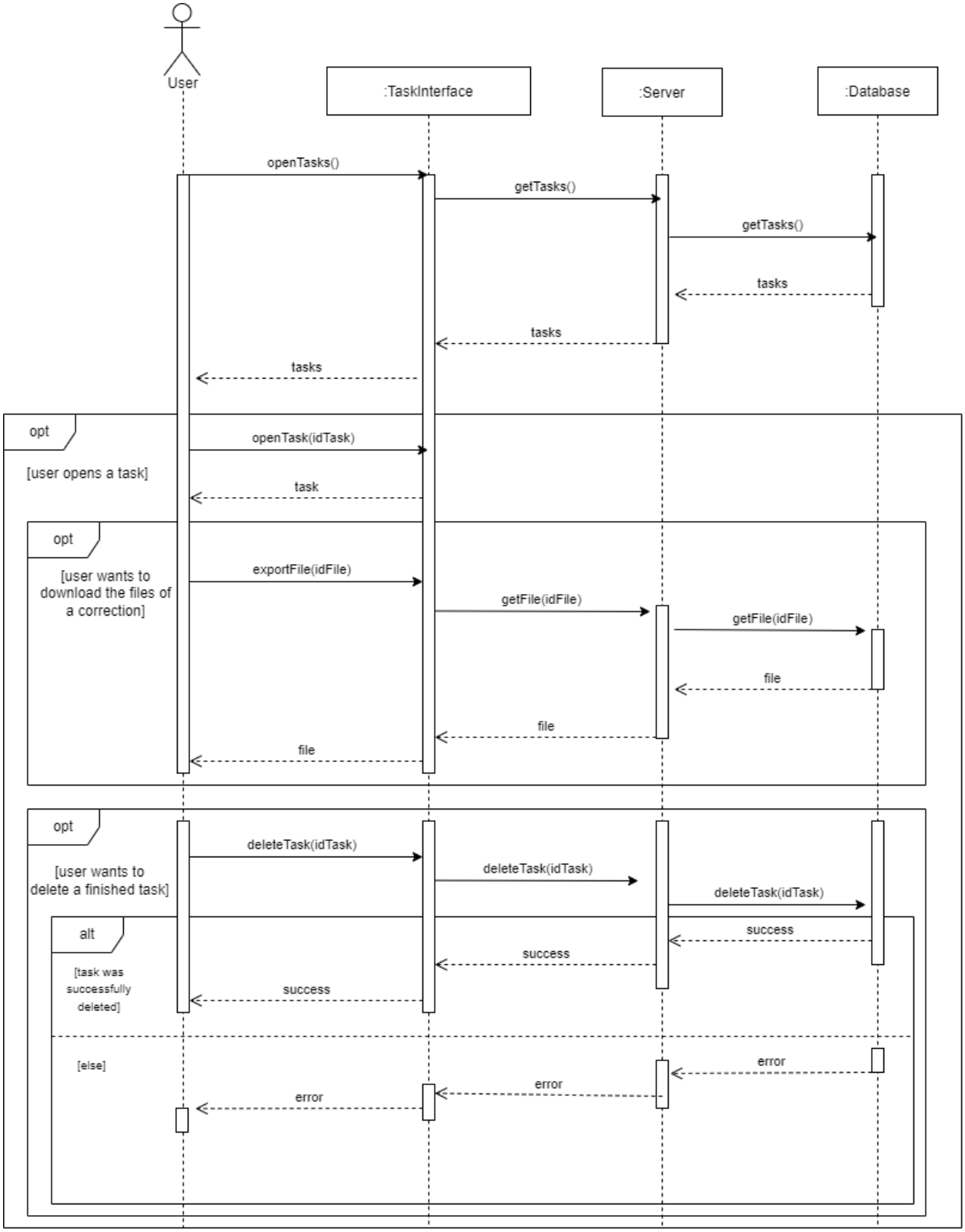

Figure 28: Diagramme de séquence pour la gestion des corrections

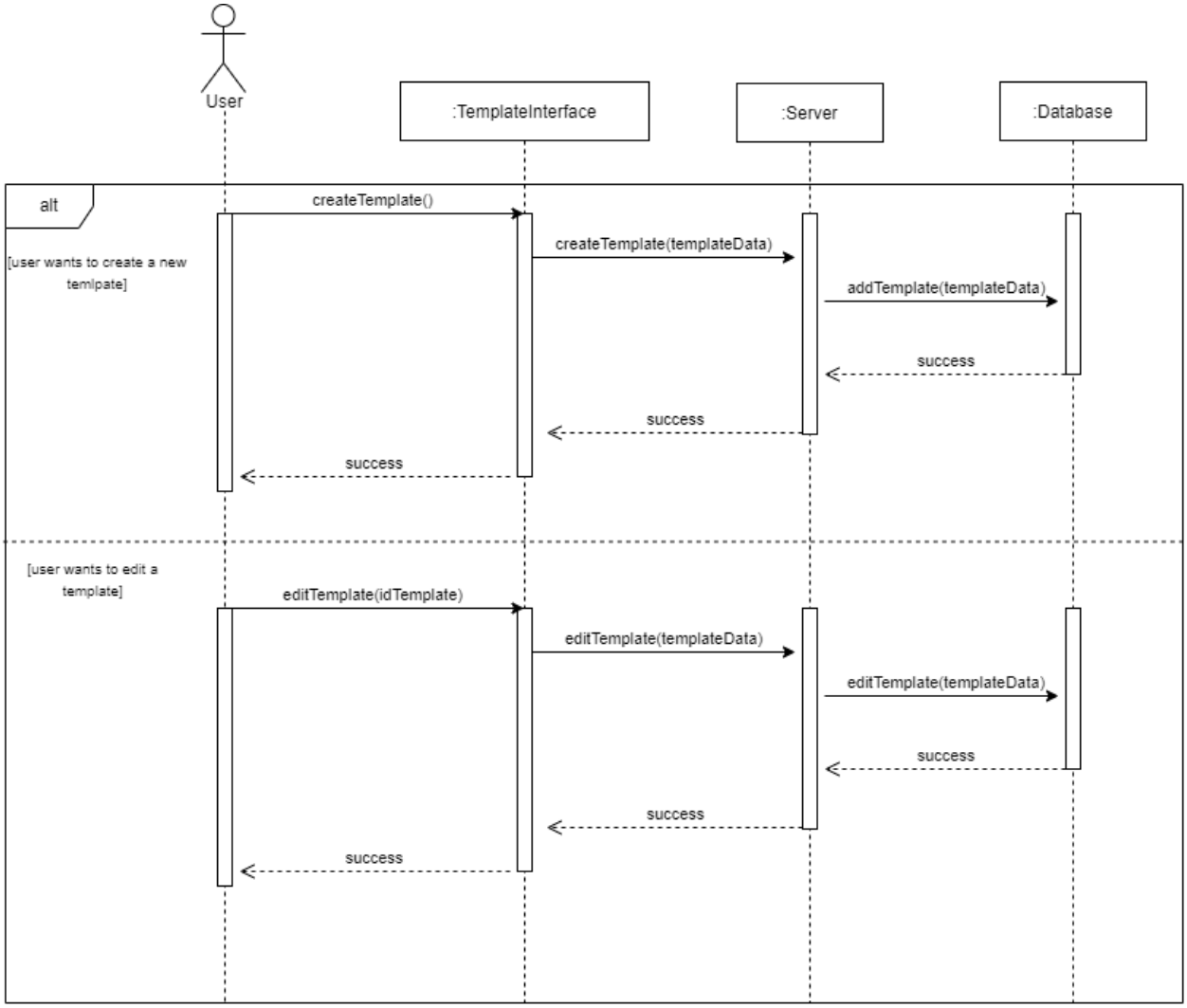

Figure 29: Diagramme de séquence pour la création et la modification d'un gabarit

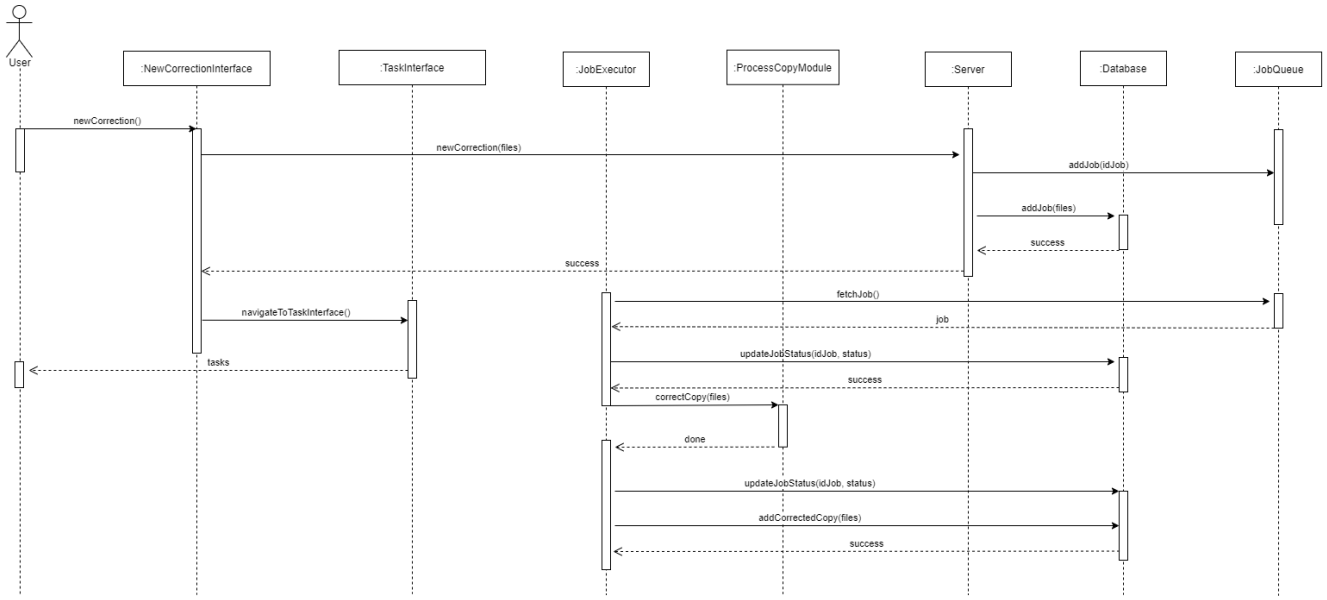

Figure 30: Diagramme de séquence pour traiter une nouvelle correction

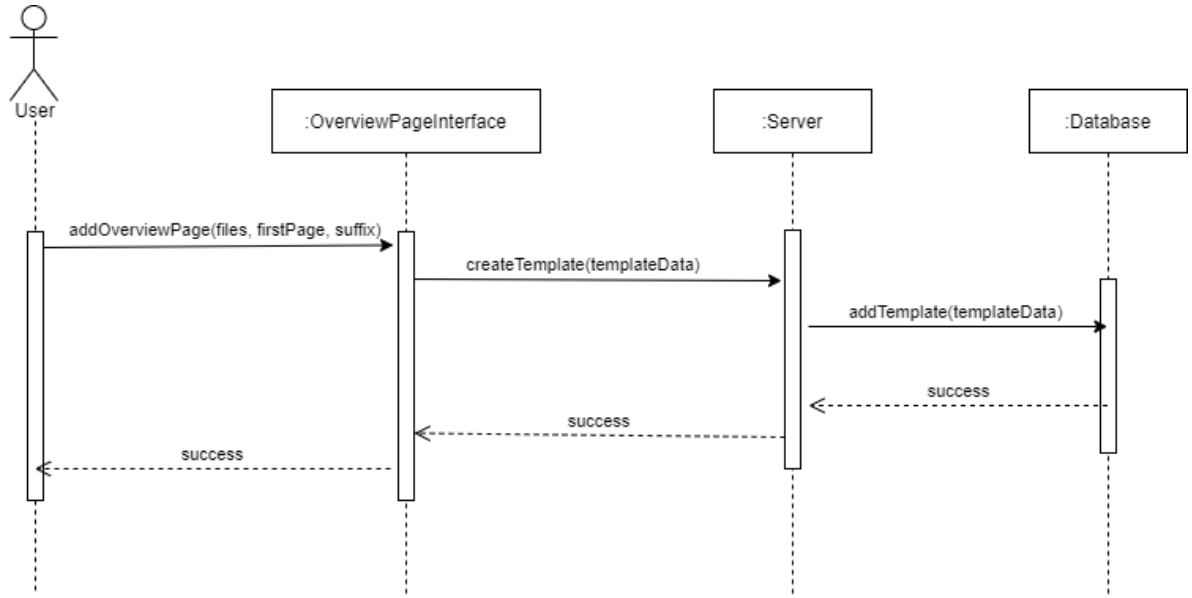

Figure 31: Diagramme de séquence pour traiter une nouvelle correction

6. Vue de déploiement

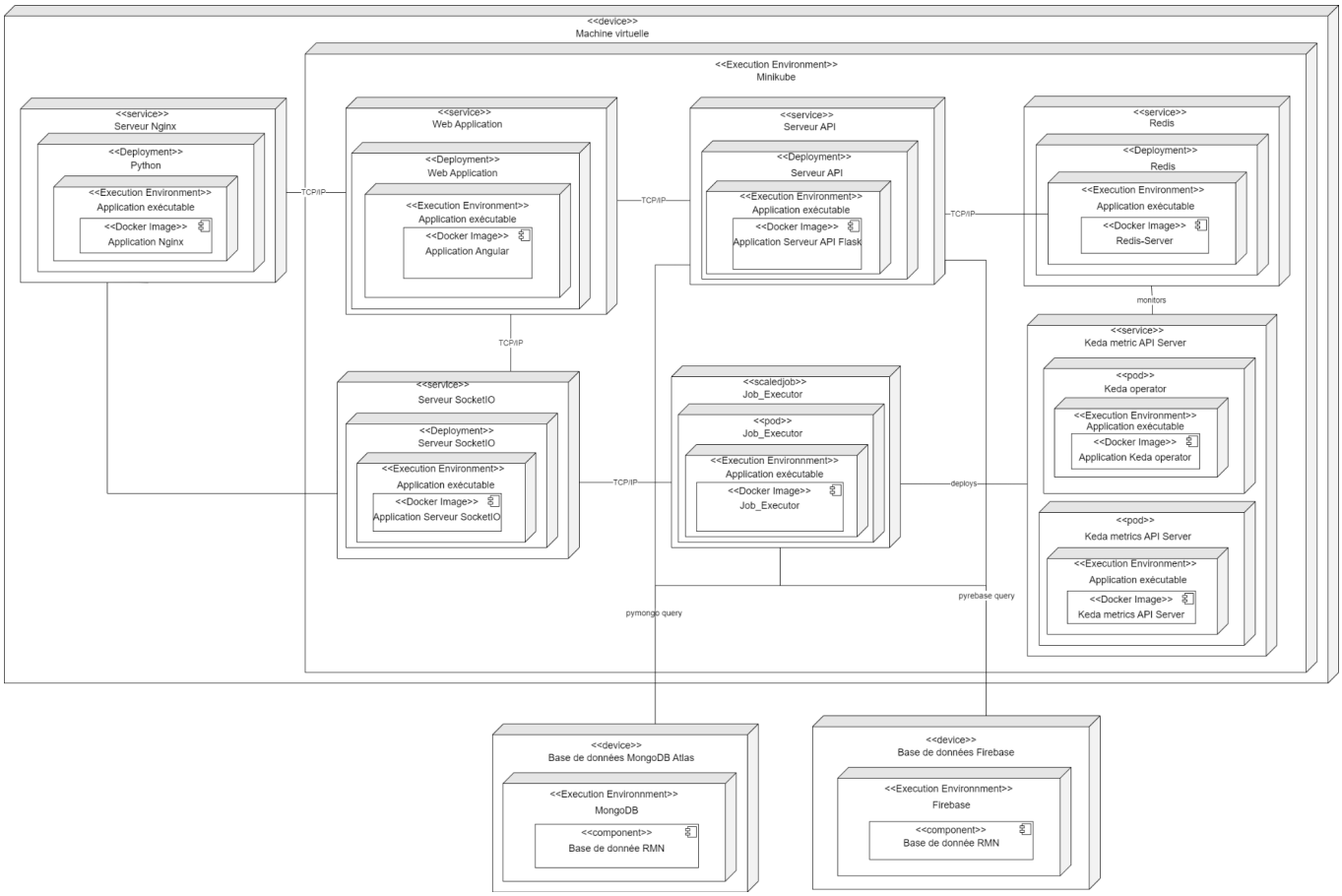

Figure 32: Diagramme de déploiement du système

7. Taille et performance

Afin que l'architecture puisse satisfaire l'utilisateur, plusieurs contraintes de taille et de performance devront être prises en considérant.

Premièrement, nous avons une application web à notre disposition, donc elle ne devrait pas prendre d'espace sur le disque dur de l'usager. Aussi, en supposant que l'ordinateur de l'usager est moderne, l'application ne devrait pas consommer plus de 1 Go de mémoire vive.

Les fichiers notes que l'utilisateur procurera à l'application web seront hébergés dans Mongodb, où ce dernier a une limite maximale de 512 Mo.

Vu que la machine virtuelle va héberger la grande majorité de notre application, son système devra pouvoir supporter les charges de plusieurs clients qui envoient des demandes simultanément. Ainsi, pour ne pas être en déficit de performance, elle devra occuper un maximum de 50 Go de RAM pour sa mémoire vive.

Par ailleurs, vu que notre application pourra avoir plus qu'un utilisateur en même temps qui fait une requête, nous avons opté pour un système de Pods où chacun d'entre eux pourra exécuter le module de reconnaissance. Ceci permettra de grandement améliorer la performance de notre application, car dans le cas contraire, on aura à faire attendre chaque utilisateur qui fera une opération après celle en cours d'exécution.

En résumé, notre architecture devra prendre en considération ces différents aspects pour satisfaire l'utilisateur. Ce dernier ne devrait pas avoir une expérience d'utilisation négative en raison de la performance du système.