Équipe 03

Cégep à distance

Document d'architecture logicielle

Version 2.0

Historique des révisions

Date	Version	Description	Auteur
2022-01-26	1.0	Rédaction initiale du document	Laura Beaudoin, Sandu Babira, Jasmine Roy, Guillaume Rompré, Anoir Dhaoui
2022-02-17	2.0	Modification des diagrammes de paquetage de la Vue Logique	Sandu Babira

Table des matières

Document d'architecture logicielle

1. Introduction

Ce document présente une modélisation de l'architecture de notre application. D'abord, on y définit les objectifs et contraintes architecturales de notre application. Ensuite, les différentes vues architecturales seront présentées, telles que la vue des cas d'utilisation, la vue logique, la vue des processus et la vue de déploiement. Enfin, les caractéristiques de taille et de performance ayant un impact sur le design de l'application seront abordées.

2. Objectifs et contraintes architecturaux

2.1 Objectifs

2.1.1 Sécurité et confidentialité

Pour garantir la confidentialité des informations personnelles de l'utilisateur (telles que son mot de passe, son courriel, ses statistiques, etc.), celles-ci doivent être privées.

Afin d'assurer une sécurité, les informations de connexion de l'utilisateur devront être confidentielles et protégées. Pour ce faire, un niveau de protection supplémentaire devra être appliqué sur le mot de passe à l'aide d'une fonction de hachage dans le but d'assurer une protection supplémentaire dans le cas d'une fuite de données. Ainsi, cela assure la confidentialité de ses données accessibles depuis son compte.

De plus, les fichiers contenant des données sensibles (comme les clés publiques ou privées) ne devront pas être accessibles par les interfaces externes du système et devront être stockés en conséquence.

2.1.2 Performance

Afin d'assurer une bonne qualité de l'expérience utilisateur, le système devra être conçu avec une architecture pouvant supporter plusieurs connexions simultanément sans compromettre la performance du système (surtout en termes de temps de réponse). Toutefois, comme le système de comporte pas de composantes nécessitant une communication synchrone, la performance n'est pas un élément critique à considérer.

2.1.3 Portabilité

Comme le système sera potentiellement exécuté sur diverses plateformes (ordinateurs et appareils mobiles, ainsi que des différents fureteurs web), ce dernier devra être conçu avec une architecture compatible avec la majorité des technologies couramment utilisées.

2.1.4 Testabilité

Considérant que le système développé comporte trois composantes logiques (parties *frontend* et *backend* du Tableau de bord, ainsi que le module de prédiction des abandons), celui-ci devra être développé en suivant cette modularité dans le but d'assurer l'indépendance de chaque composante. Ainsi, chaque module pourra être testé individuellement avant d'être intégré au système, ce qui facilitera grandement la discipline de tests. De plus, cette modularité améliorerait la maintenabilité du système dans le cas ou de futures modifications seraient nécessaires.

2.2 Contraintes

2.2.1 Outils et langages de développement

Outils :

- Angular : utilisé pour l'interface utilisateur du tableau de bord.

- MongoDB : utilisé comme base de données.
- Express et Node.js : utilisé pour implémenter le serveur.

Langages :

- HTML, SCSS : utilisé pour l'interface utilisateur.
- Typescript : utilisé pour l'interface utilisateur et le serveur.
- Python : utilisé pour le développement de l'outil de prédiction des abandons.

2.2.2 Échéancier

Étant donné les échéances serrées et le temps de développement limité (livraison de la version bêta prévue pour le 5 avril 2022), l'architecture du système devrait tenir compte de ces restrictions. Ainsi, celle-ci devrait être assez rapide à implémenter et pouvoir être adaptée facilement en cas de besoin.

3. Vue des cas d'utilisation

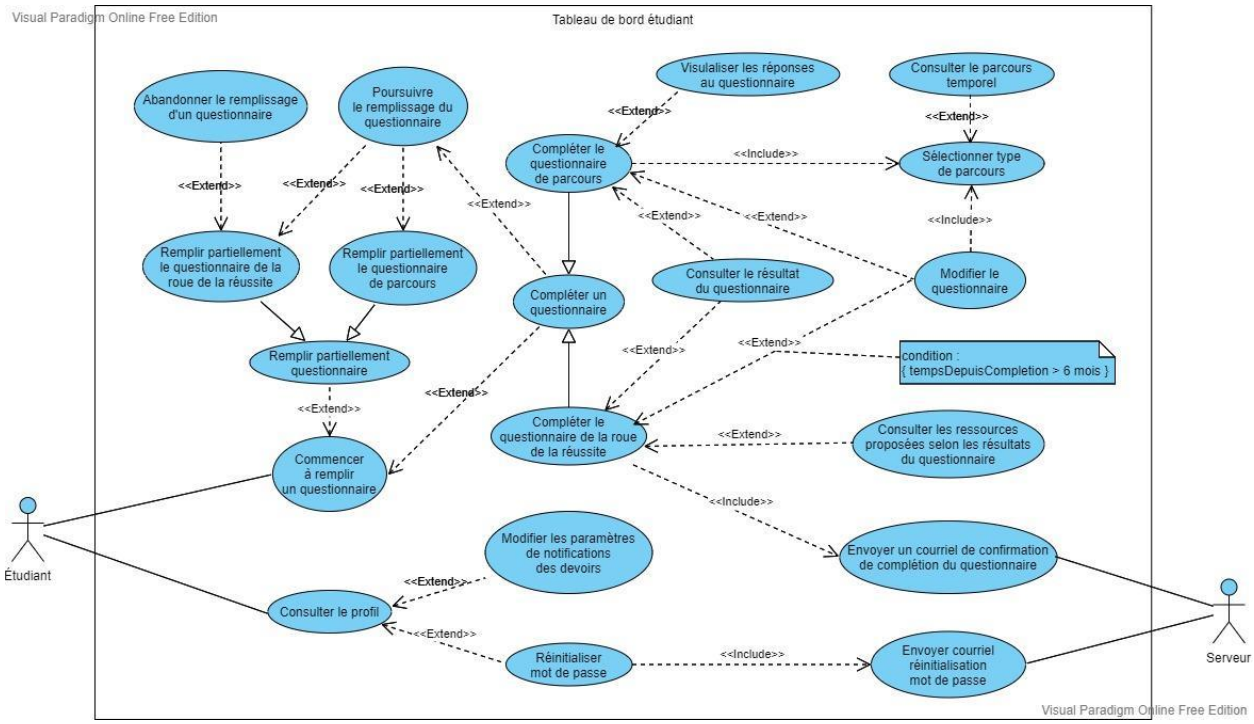

Figure 1. Diagramme des cas d'utilisation - Tableau de bord pour un étudiant

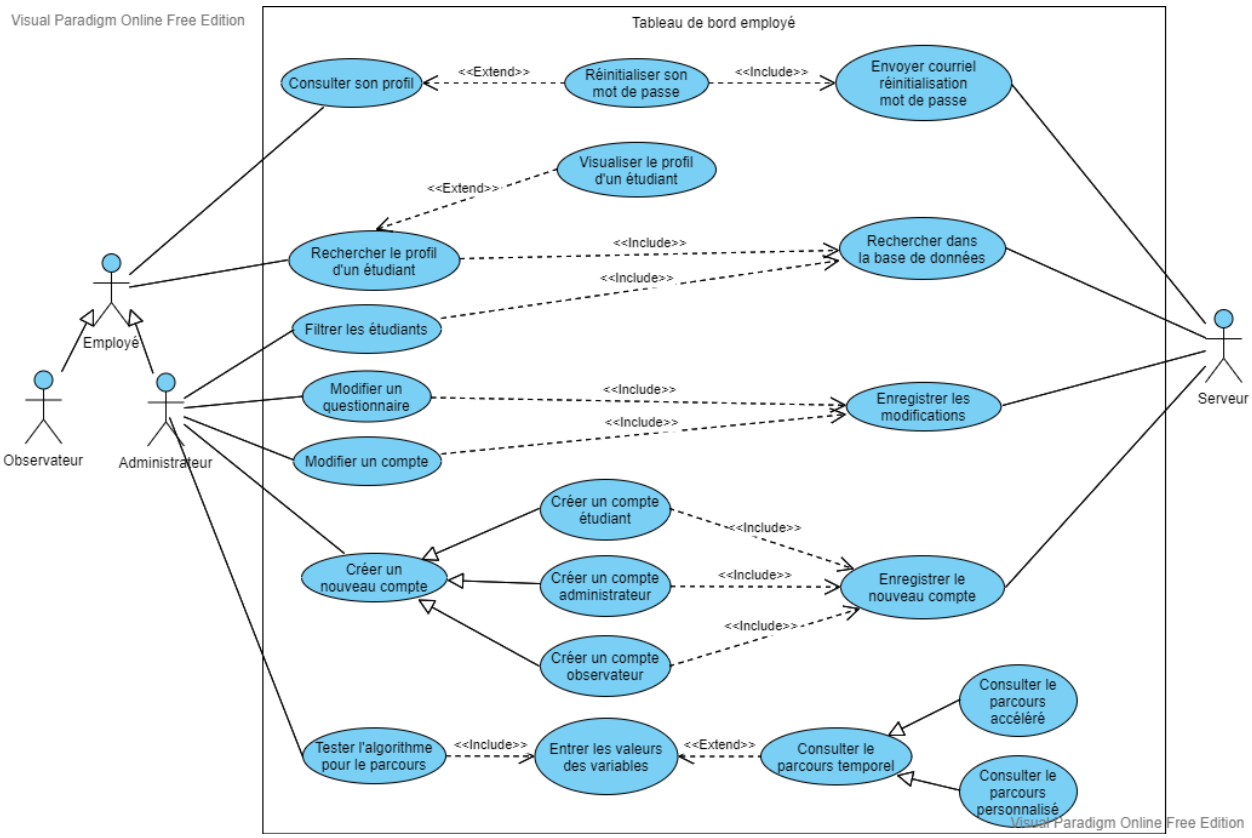

Figure 2. Diagramme des cas d'utilisation - Tableau de bord pour un employé

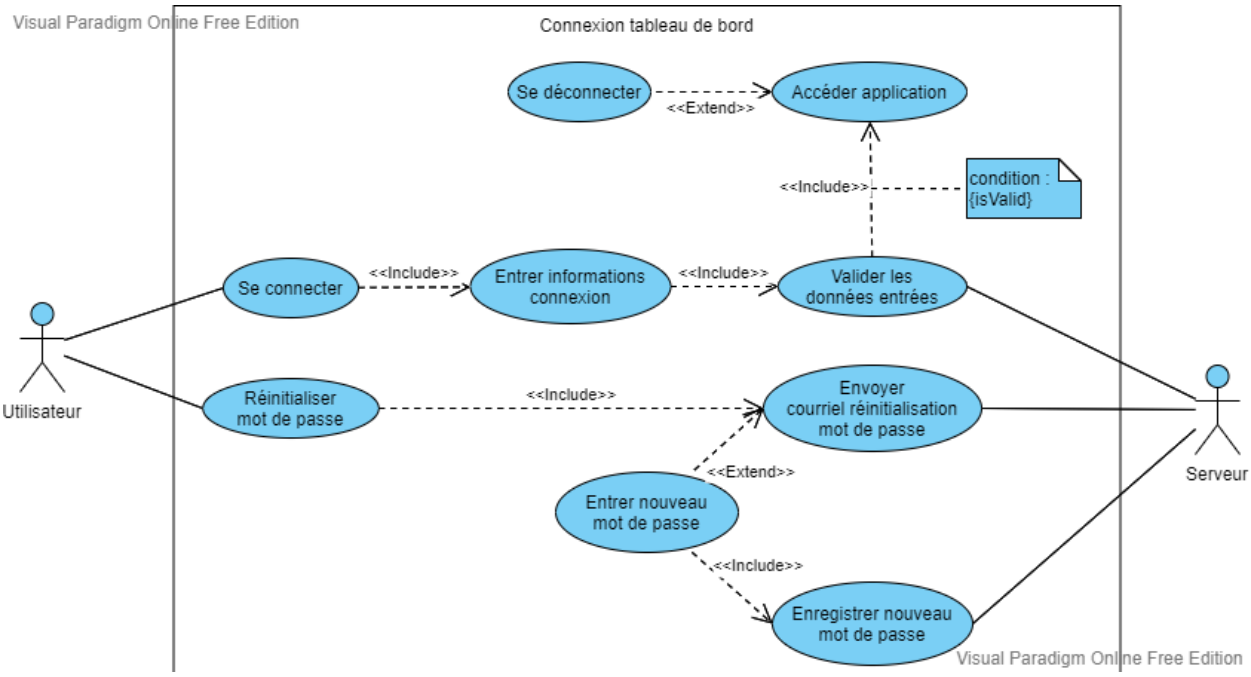

Figure 3. Diagramme des cas d'utilisation - Connexion au tableau de bord

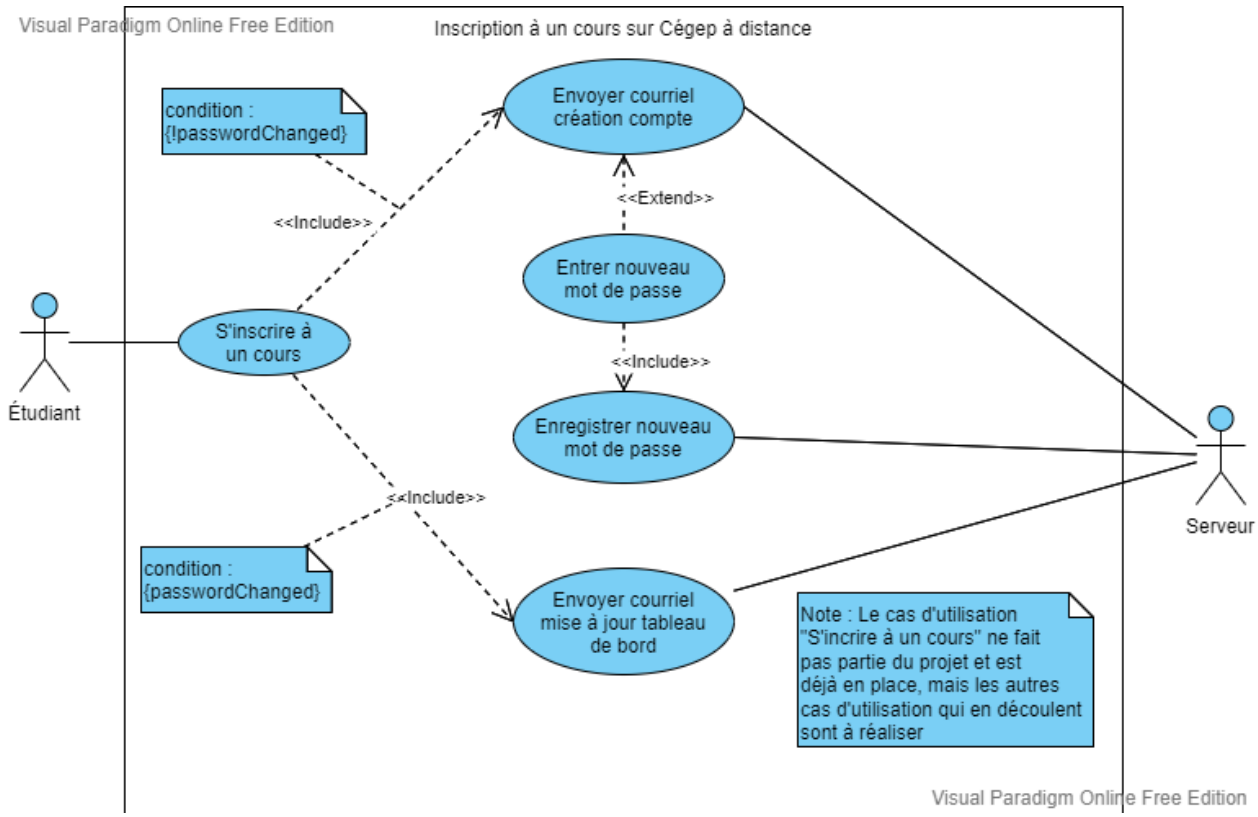

Figure 4. Diagramme des cas d'utilisation - Inscription à un cours

4. Vue logique

L'application développée sera composée de 3 éléments principaux : le client Angular (ici identifié comme le *frontend*), le serveur Express (ou le *backend*) et le module IA. Chacune de ces parties sera indépendante et communiquera avec les composantes concernées à travers des requêtes REST. Plus de détails concernant l'architecture globale du système seront présentés à la *Figure 24* dans le diagramme de déploiement. Dans cette section, les diagrammes de paquetage pour chacune de ces composantes seront présentés, ainsi que les diagrammes de classes pour certaines structures, lorsque ceux-ci seront jugés pertinents.

4.1 Frontend

Cette partie du système sera basée sur une structure plutôt classique d'un projet Angular : on y retrouvera des *Modules*, des *Components*, des *Services*, etc. Ceci dit, l'application du client sera composée de trois paquetages principaux : *Root*, *Modules* et *Shared*, présentés à la *Figure 5*.

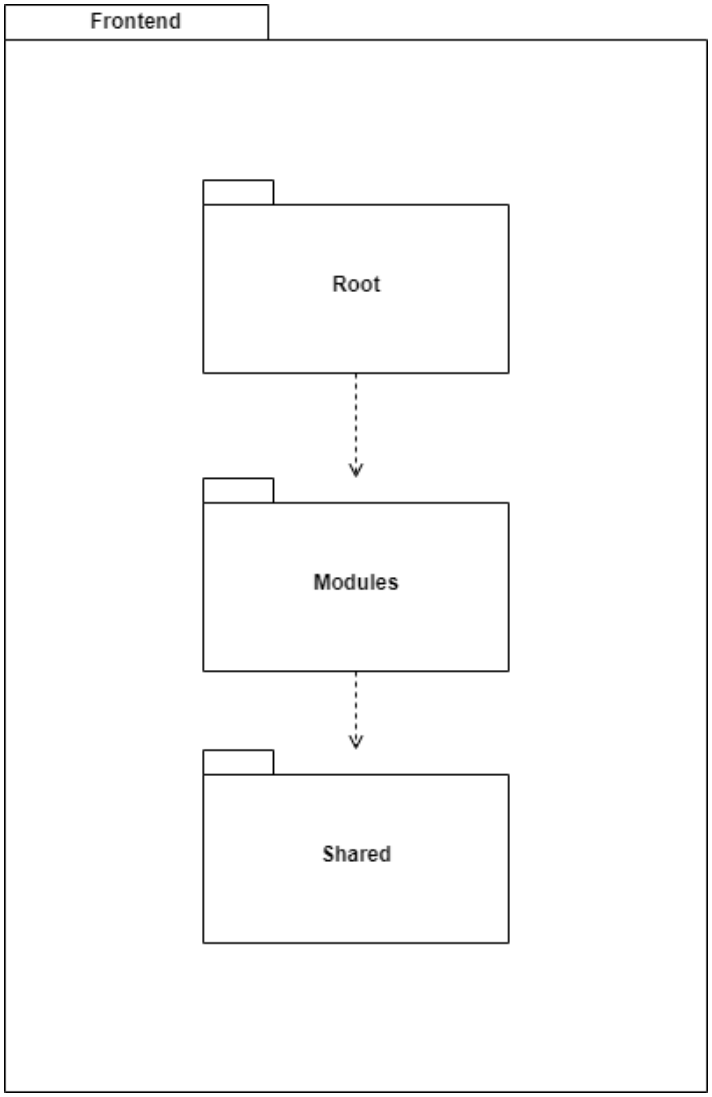

Figure 5. Diagramme de paquetage de haut niveau de la structure du client.

Voici un bref descriptif de chacun de ces paquetages :

- Root : ce paquetage contient les modules de racine de l'application. C'est le point d'entrée, à partir duquel les appels seront faits aux modules nécessaires.

- Modules : ce paquetage contient tous les modules associés aux différentes fonctionnalités de l'application. L'utilisation de modules permet d'encapsuler chacune des parties de l'application et de charger seulement ceux qui sont nécessaires à l'utilisateur. Ainsi, de manière générale, chacun de ces modules aura une structure similaire, ayant les éléments suivants : *routing*, *pages*, *components* et *services*. Certains modules possèdent également une composante nommée *interfaces*, qui contient naturellement les interfaces spécifiques utilisées par le module en question. Tous ces éléments communiqueront entre eux comme décrit dans la *Figure 6* pour assurer les fonctionnalités attendues. Ceci résulterait théoriquement en une meilleure performance de l'application et une modification plus facile de cette dernière en cas de besoin.

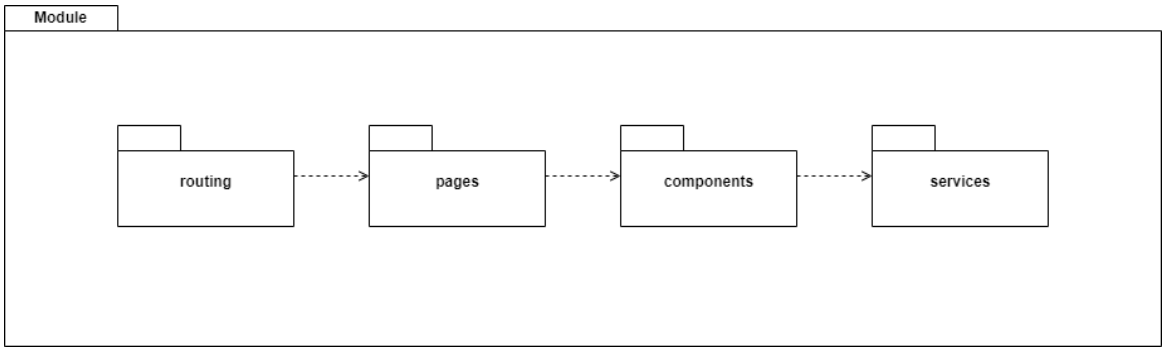

Figure 6 : Un exemple de diagramme de paquetage pour les possibles modules du *fontend* de l'application.

- Shared : ce paquetage contient les structures qui seront partagées par tous les modules. En effet, comme les modules sont des entités complètement séparées, chacun possède son propre *scope*. Pour ce fait, un paquetage séparé contenant des éléments communs à tous les modules du *frontend* a été conçu. L'accès à ces éléments se fait à travers une interface commune : *SharedModule*.

Pour mieux visualiser la structure de l'application Angular, la *Figure 7* présente un diagramme de paquetage plus détaillé de la structure du *frontend*

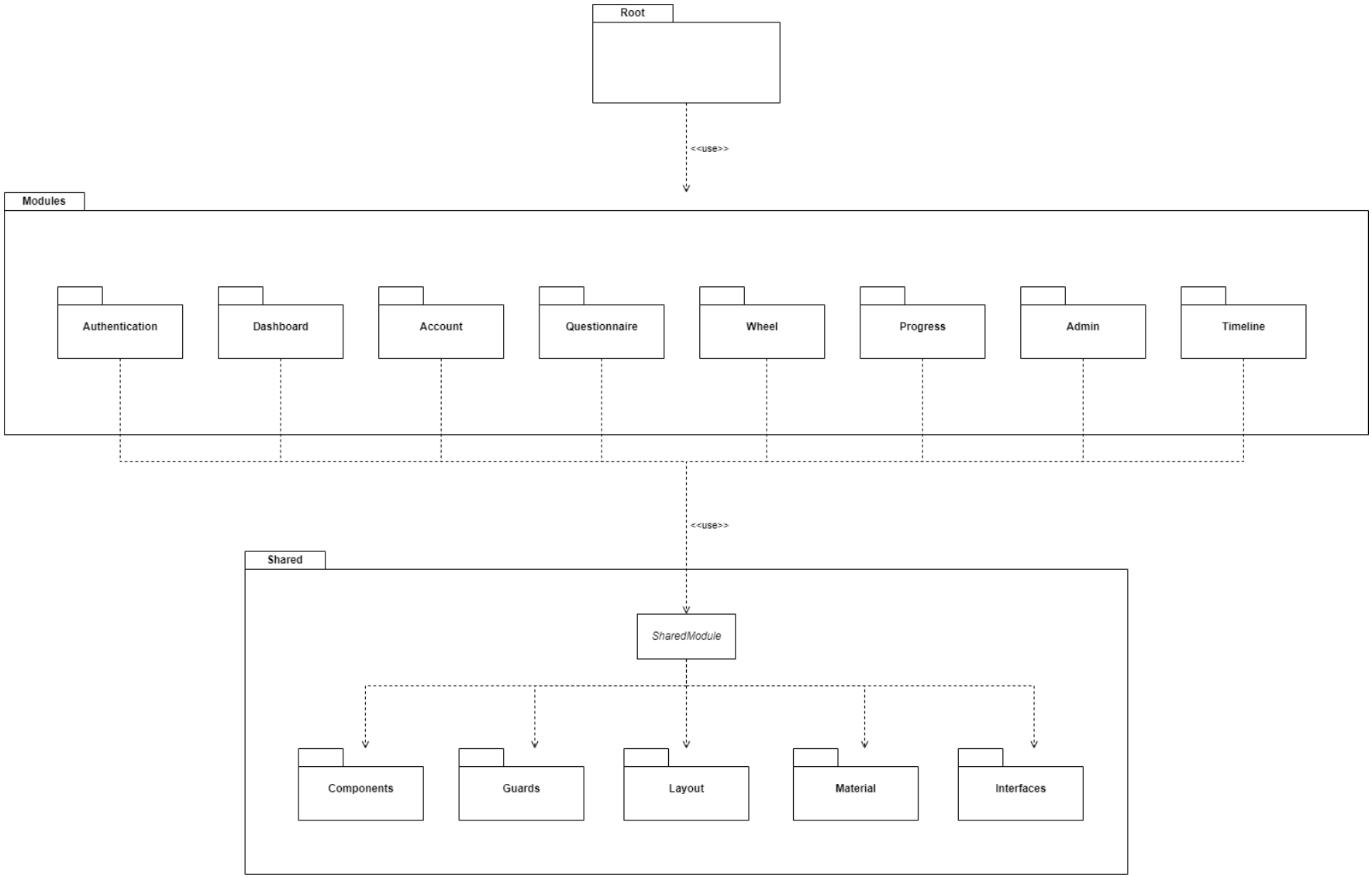

Figure 7. Diagramme de paquetage détaillé de la structure du *frontend*.

4.2 Backend

Le serveur de l'application sera implémenté sous la forme d'une architecture en couches. En effet, ce dernier sera composé de quatre couches distinctes (*Controllers, Managers, Services et Repositories*), qui ne vont interagir qu'avec des couches adjacentes. La *Figure 8* présente un aperçu de haut niveau de cette structure. On note également l'ajout de nouvelles structures de support pour les couches présentées ci-dessous, comme les paquetages *Middlewares*, *Data-Files* et *Classes*. La *Figure 9* présente un diagramme de paquetage étendu incluant ces nouveaux éléments.

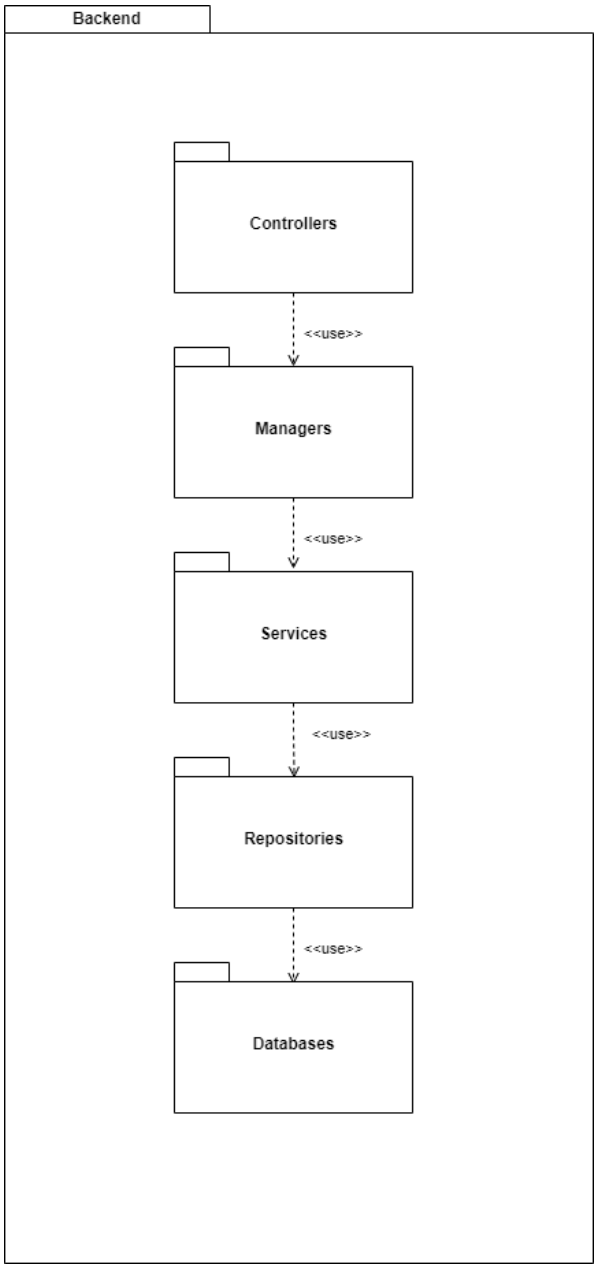

Figure 8. Diagramme de paquetage de haut niveau de la structure du serveur

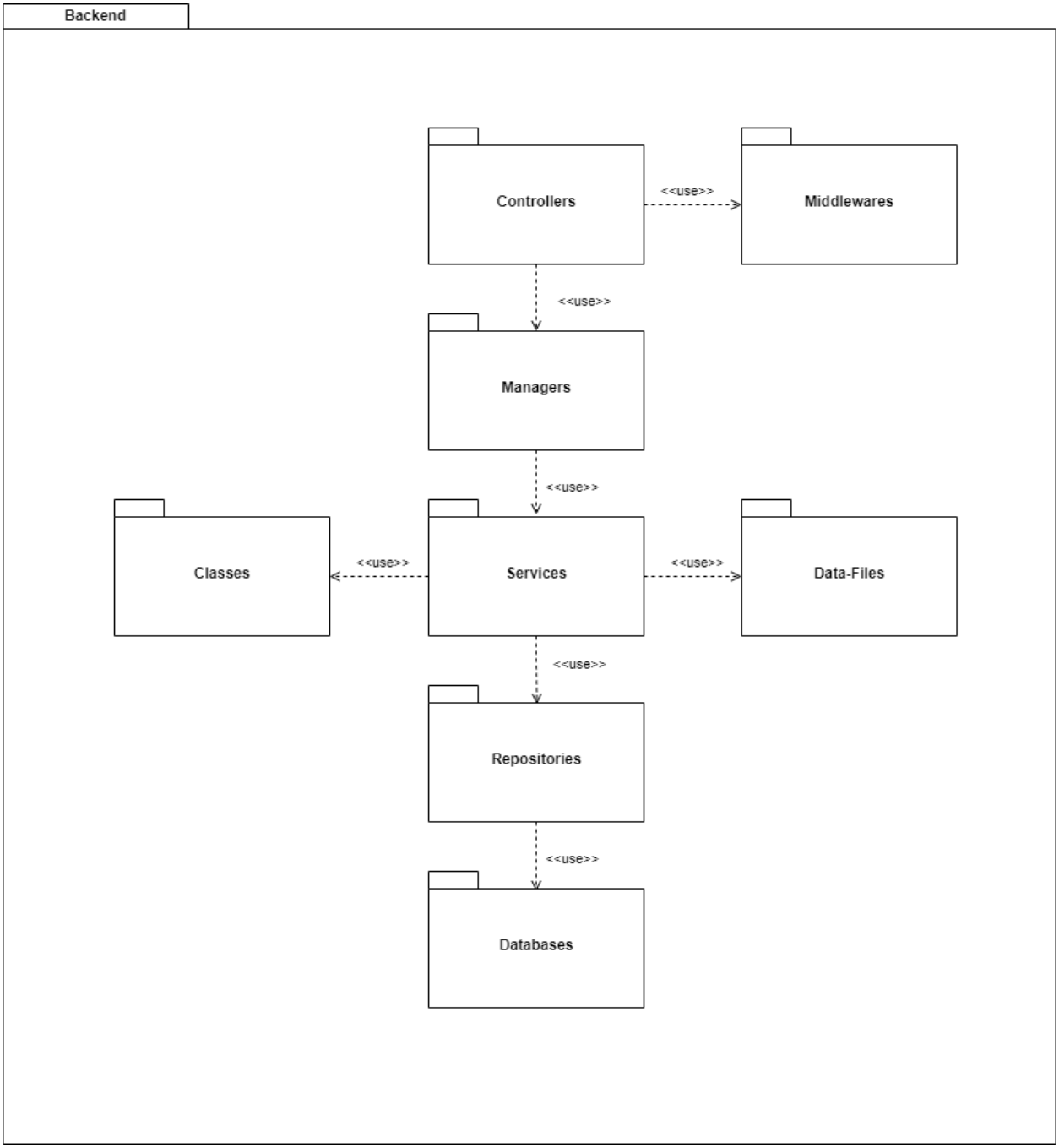

Figure 9. Diagramme de paquetage étendu de haut niveau de la structure du serveur

Chacune de ces couches possède une fonction précise, permettant de respecter les principes de responsabilité unique (ou du moins limitée et précise) et d'encapsulation. Ceci aura pour effet d'assurer une meilleure clarté et maintenabilité du code. Voici la description des rôles de chacune de ces couches :

- Controllers : cette couche s'occupe de la gestion des requêtes reçues par le serveur en les relayant aux structures appropriées (*Managers*). Elle comprend les fonctions associées au *Router* et les complète en permettant de regrouper les routes par thème. La liste des différents controllers est présentée dans le diagramme de paquetage détaillé à la *Figure 11*. Ces différents types de contrôleurs héritent d'une structure de base établie par la classe abstraite *Controller* (voir *Figure 10* ci-dessous). De plus, comme l'application requiert une gestion des droits d'accès, les *controllers* feront usage du paquetage *Middleware*, qui contient des classes permettant de valider ces droits d'accès avant de faire appel aux structures sous-jacentes.

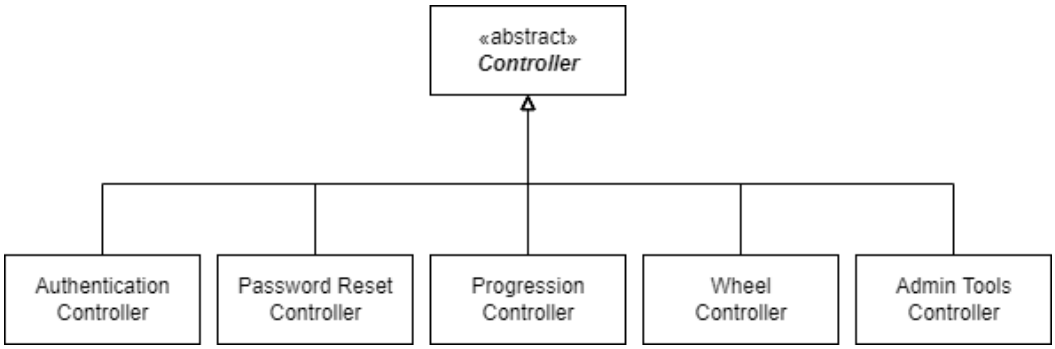

Figure 10. Diagramme de classe des *Controllers*.

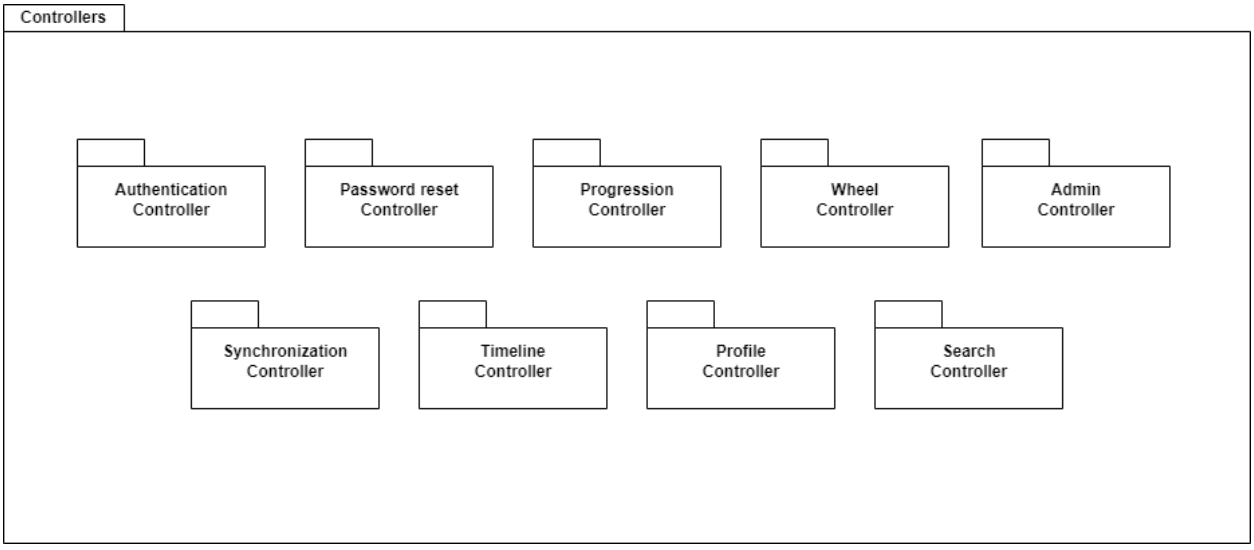

Figure 11. Diagramme de paquetage détaillé pour les *Controllers*.

- Managers : comme mentionné ci-haut, les éléments de cette couche se font appeler directement par les *Controllers* et ont comme tâche principale de coordonner les services nécessaires afin d'accomplir une tâche. Ils agissent comme des interfaces, offrant une liste d'opérations reliées à chaque module du système et font appel aux services individuels pour les accomplir. Contrairement aux *Controllers*, les *Managers* n'ont pas de structure commune puisqu'ils coordonnent des services différents d'un module à l'autre. La liste de *Managers* est présentée à la *Figure 12*.

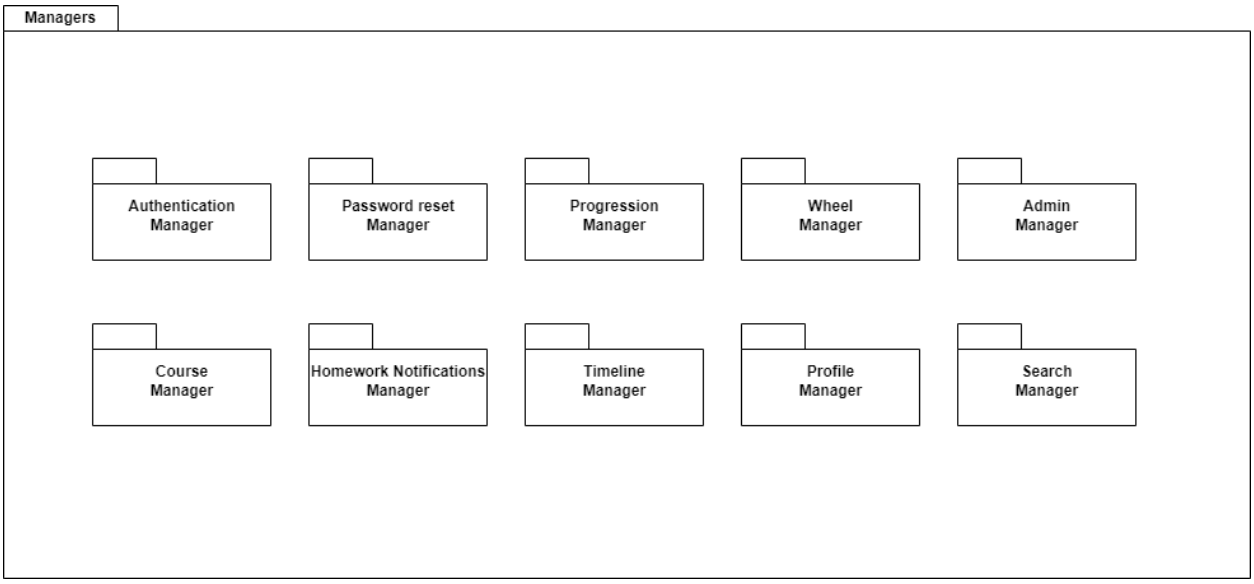

Figure 12. Diagramme de paquetage détaillé pour les *Managers*.

- Services : le paquetage des services comprend la majeure partie de la logique du serveur. Cette dernière est séparée en services individuels par fonctionnalité, servant à accomplir un but précis. Il est également fréquent que des services de cette couche communiquent entre eux pour accomplir certaines tâches. Certains de ces services font appel aux éléments de la couche inférieure, les *Repositories*, afin d'interagir avec la base de données.

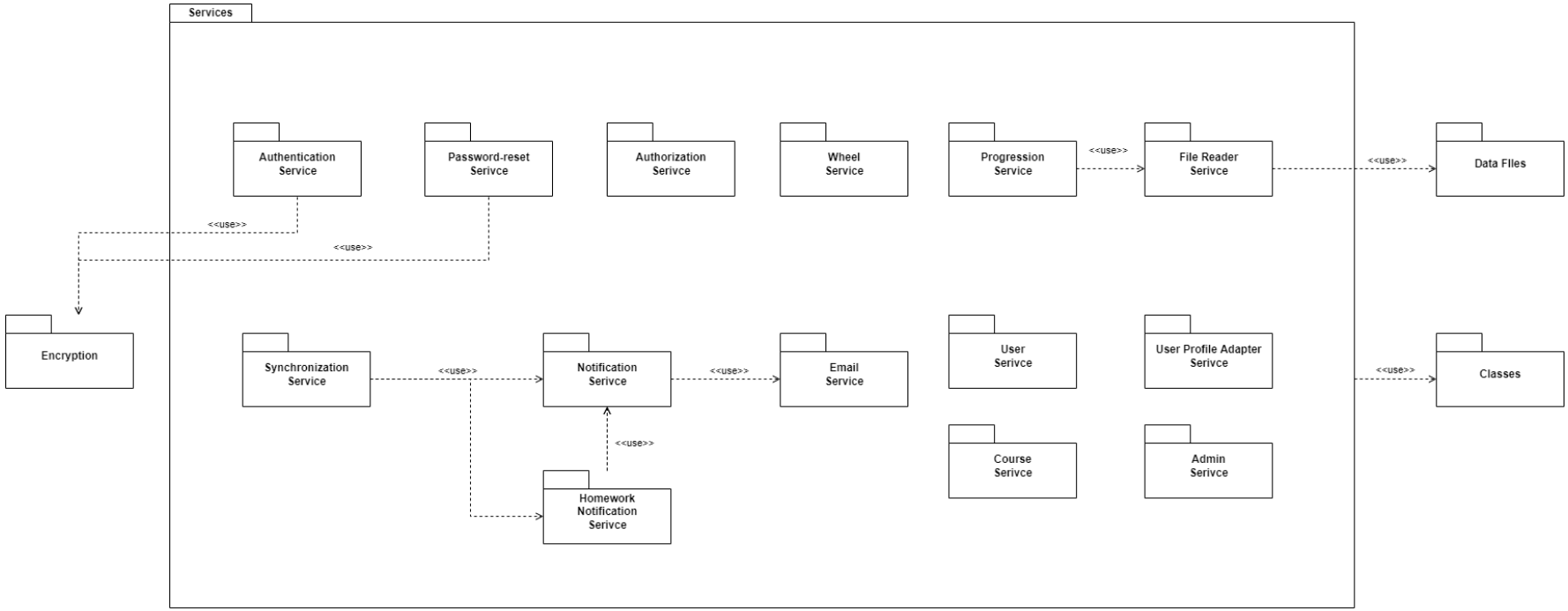

Figure 13. Diagramme de paquetage détaillé pour les *Services*.

- Repositories : cette couche représente l'un des niveaux les plus bas pour la gestion de données, avant la base de données elle-même et la structure qui en fait la connexion. Cette couche comprend les modèles décrivant les structures de la BD, ainsi que les entités gérant l'accès exclusif à ces structures. Autrement dit, c'est le seul point d'accès à la BD à partir de l'application. Cette encapsulation permet d'assurer une gestion plus efficace et sécuritaire des données.

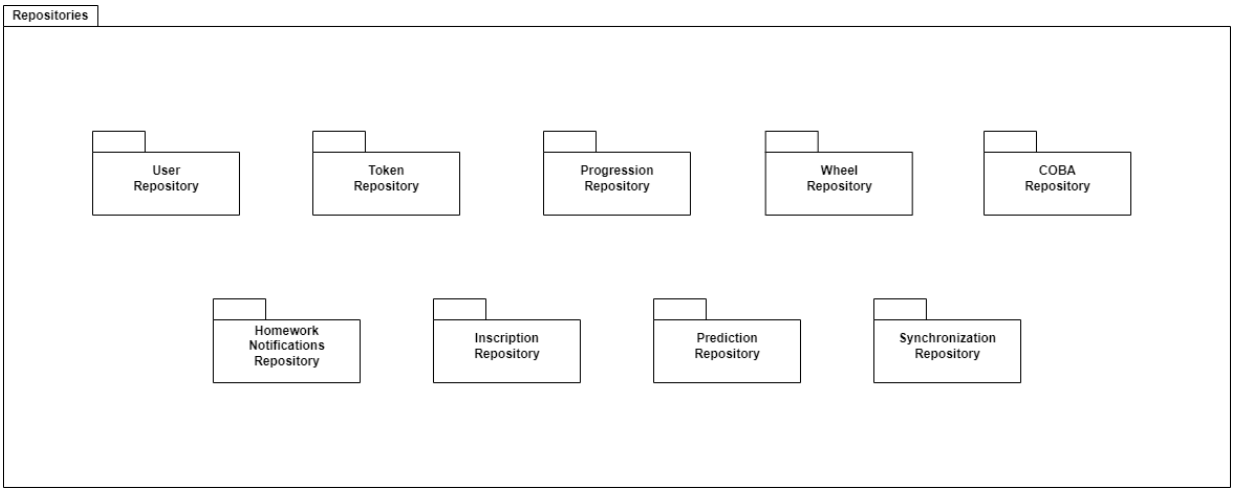

Figure 14. Diagramme de paquetage détaillé pour les *Repositories*.

- Databases : cette dernière couche s'assure simplement de faire la connexion avec les bases de données respectives afin de permettre aux *Repositories* de faire les opérations nécessaires.

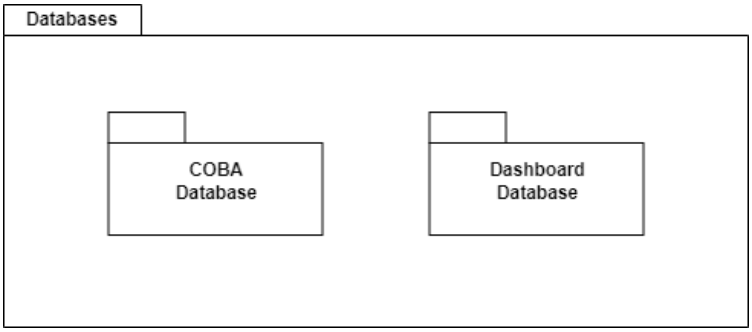

Figure 15. Diagramme de paquetage détaillé pour les *Databases*.

4.3 Prévision des abandons

Cette composante du système prendra la forme d'une intelligence artificielle responsable du calcul de la probabilité d'abandon pour les étudiants du Cégep à distance. Cette dernière sera alimentée directement par la base de données de l'institution selon un horaire préétabli ou par demande d'un utilisateur ayant des droits exclusifs (admin). Le calcul des résultats des prédictions sera également déclenché selon un horaire et enregistrera les données obtenues sur une base de données Mongo. Voici un diagramme de haut niveau représentant la structure générale de cette composante.

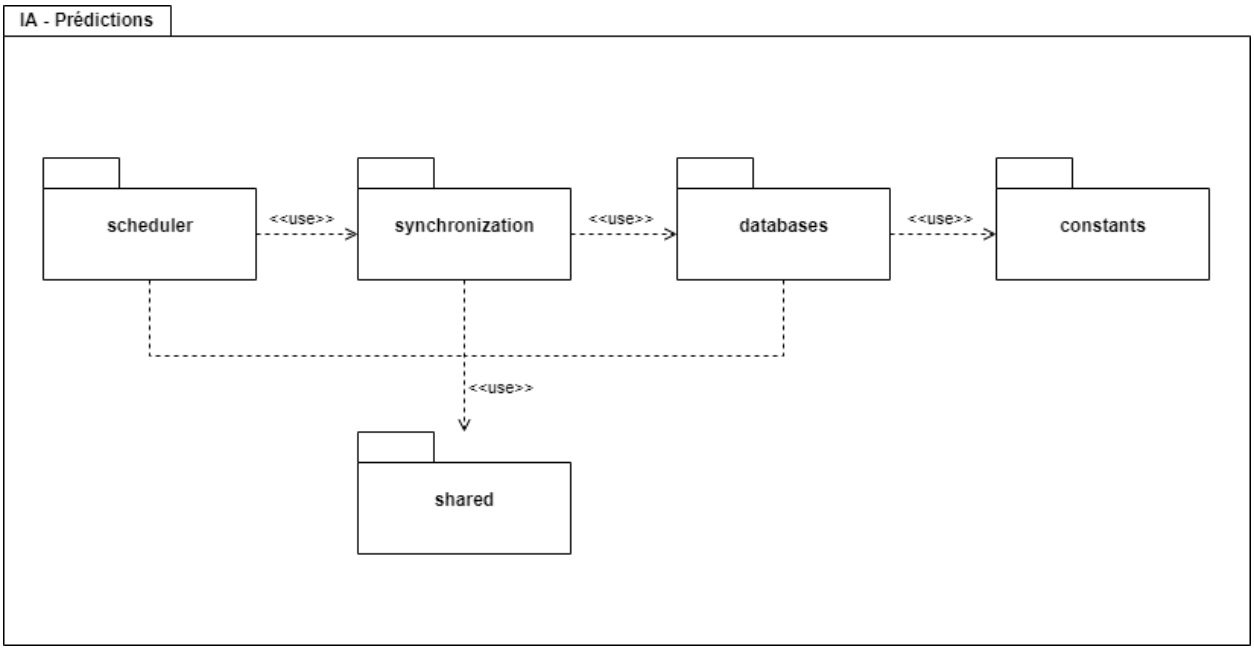

Figure 16. Diagramme de paquetage de la structure de l'IA pour la prédiction des abandons.

Voici une brève description des rôles de chacun des paquetages identifiés ci-haut :

- Scheduler : comme la composante de l'IA va devoir se mettre à jour régulièrement en fonction des interactions continues des étudiants avec le système, ce paquetage va s'assurer d'exécuter certaines tâches régulièrement afin d'assurer la pertinence des données calculées. Son but sera donc de déclencher les tâches (*jobs*) à une heure prédéterminée en appelant le paquetage *Synchronization* (et plus particulièrement le *SynchronizationCoordinator*).

- Synchronization : ce paquetage a pour but d'assurer, dans un premier lieu, la synchronisation des données analysées par l'IA entre le système et les bases de données du CAD. En effet, on s'assure, à chaque synchronisation planifiée quotidiennement, de retrouver les dernières entrées d'inscription en provenance de COBA (à travers le *Inscription Synchronization*) et celles des actions provenant de Moodle (à travers *Action Synchronization*). Par la suite, un autre élément de ce paquetage s'occupe de mettre à jour les données de prédiction calculées par l'IA : il s'agit du *Prediction Synchronization*. Finalement, un coordonnateur s'occupe de faire les appels nécessaires selon la logique de fonctionnement du module. La *Figure 17* décrit la structure de ce paquetage.

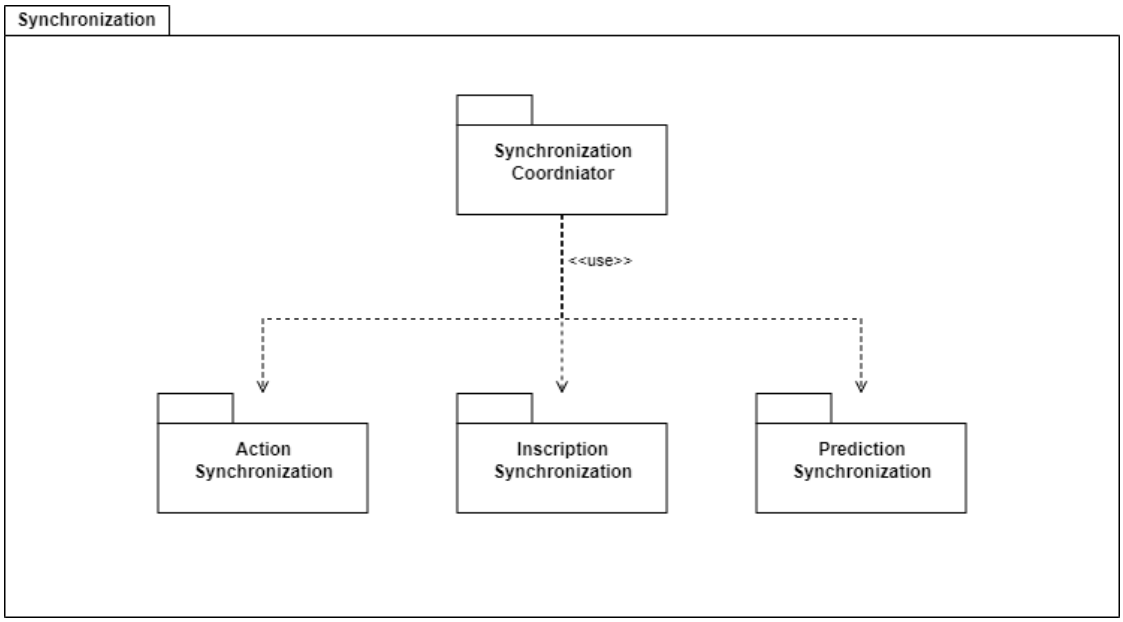

Figure 17. Diagramme de paquetage détaillé du paquetage *Synchronization*.

- Databases : ce paquetage contient les structures nécessaires à la connexion aux différentes bases de données (COBA, Moodle et le tableau de bord) ainsi que les *Repositories* qui se chargent des opérations sur ces dernières. Il contient un sous-paquetage pour chacune des bases de données nommées, soit *Coba*, *Moodle*, et *Dashboard*. Voici un schéma qui représente ce paquetage.

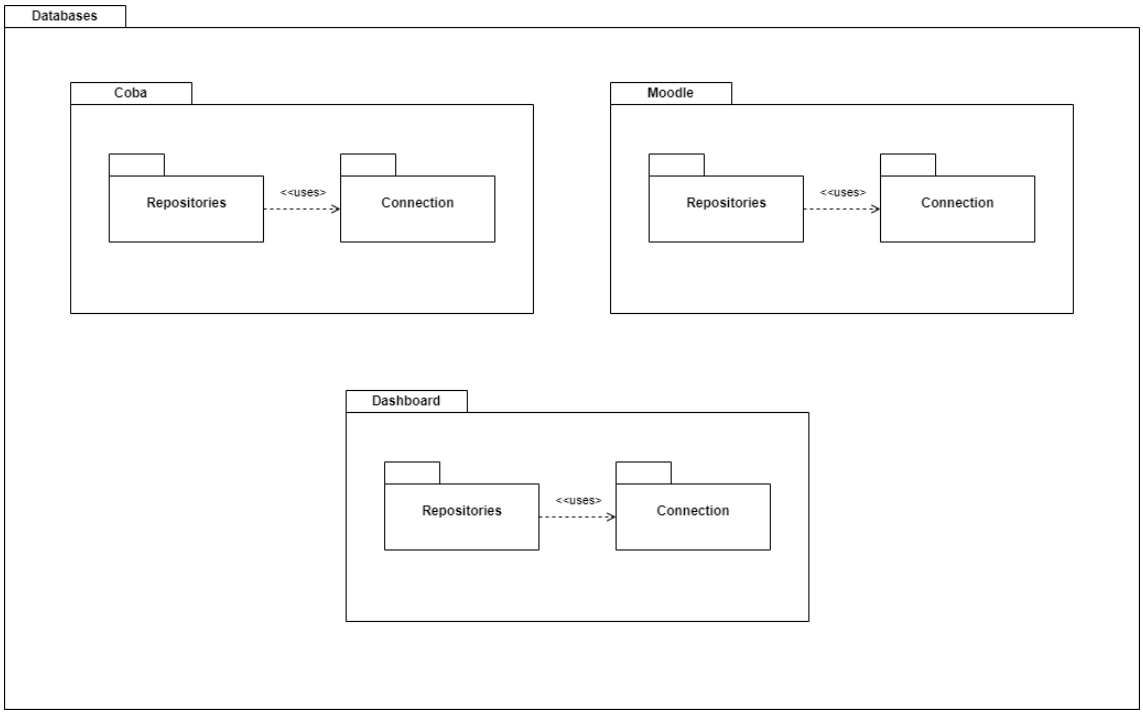

Figure 18. Diagramme de paquetage détaillé du paquetage *Databases*.

- Constants : ce paquetage contient toutes les constantes nécessaires pour les accès aux bases de données. Ces dernières ont été extraites du paquetage précédent (*Databases*) afin de respecter l'encapsulation et d'augmenter la maintenabilité du code en permettant de modifier certains paramètres plus facilement. La *Figure 19* représente les contenus de ce paquetage.

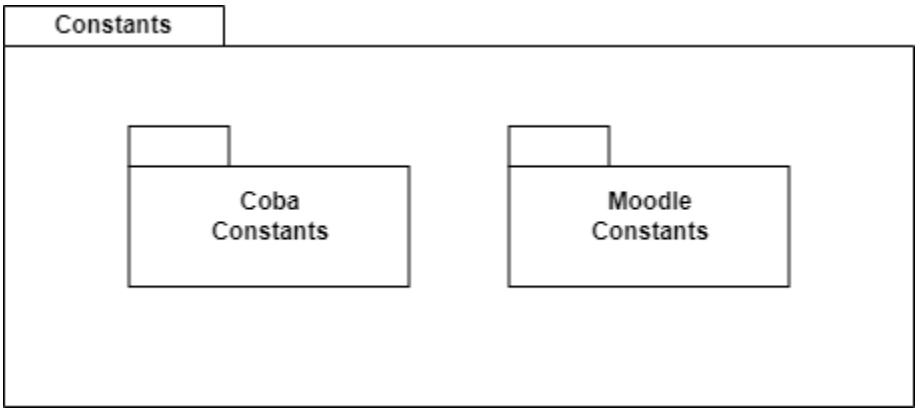

Figure 19. Diagramme de paquetage détaillé du paquetage *Constants*.

- Shared : ce paquetage ne contient qu'une seule structure, qui est utilisée par plusieurs composantes du système de l'IA : l'implémentation du patron Singleton. La *Figure 20* représente un diagramme avec les éléments qui héritent de cette classe afin d'assurer la création d'une seule instance respective.

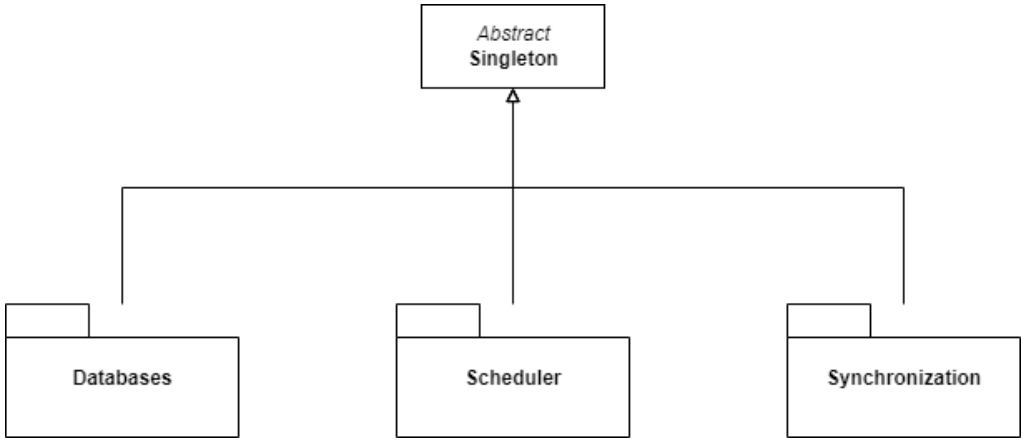

Figure 20. Diagramme représentant l'héritage de la classe *Singleton* par les paquetages du module IA.

5. Vue des processus

5.1 Authentification

Le processus d'authentification est chargé de recevoir les requêtes de connexion des usagers lorsqu'ils veulent enregistrer leur compte ou se connecter à un compte déjà créé.

5.1.1 Processus d'enregistrement d'utilisateur

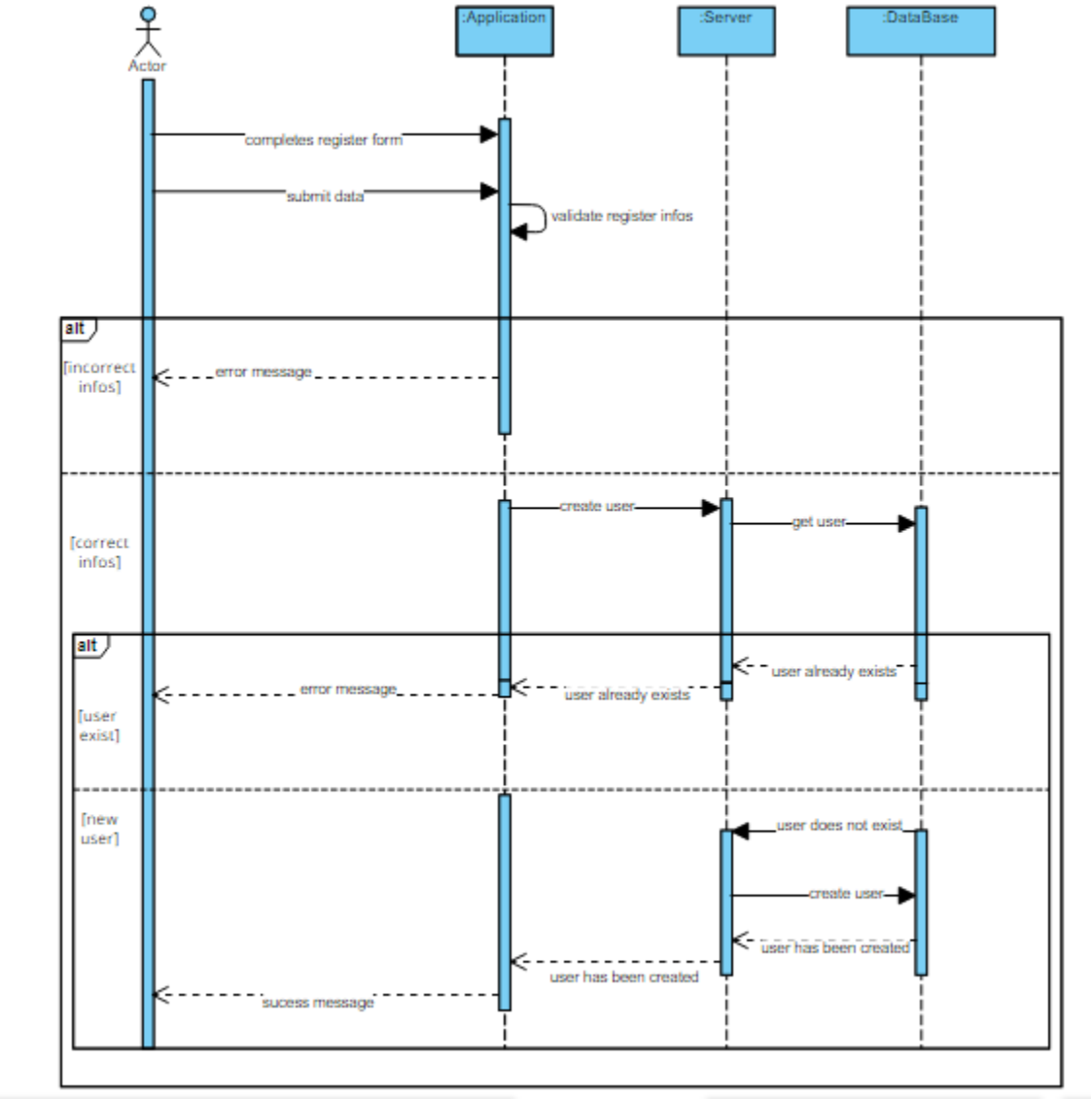

Figure 21. Vue de processus concernant l'enregistrement de l'utilisateur

5.1.2 Processus de connexion

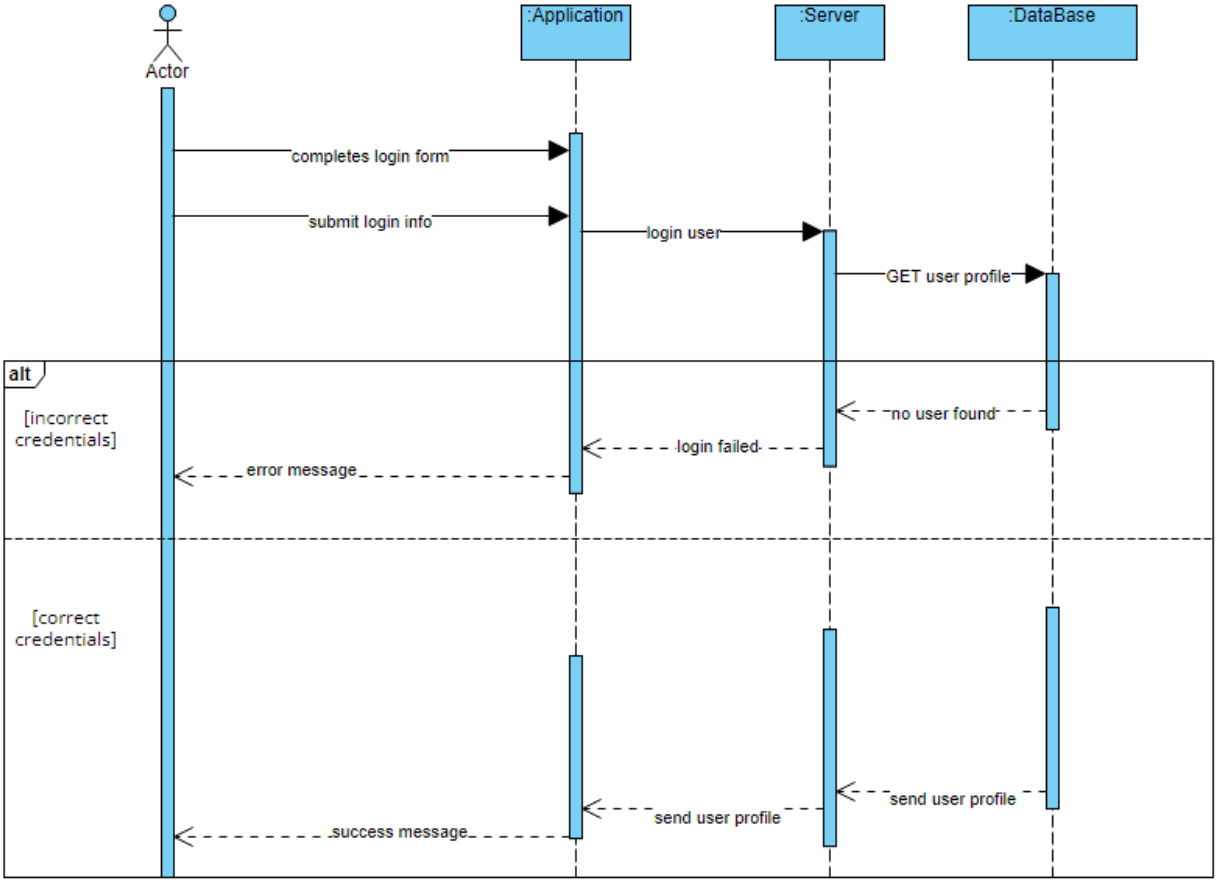

Figure 22. Vue de processus concernant la connexion de l'utilisateur.

5.1.3 Réponse à un questionnaire

Dès sa première connexion à l'application, un étudiant du CAD doit remplir un questionnaire qui permet, après une évaluation, de suggérer une option temporelle de parcours.

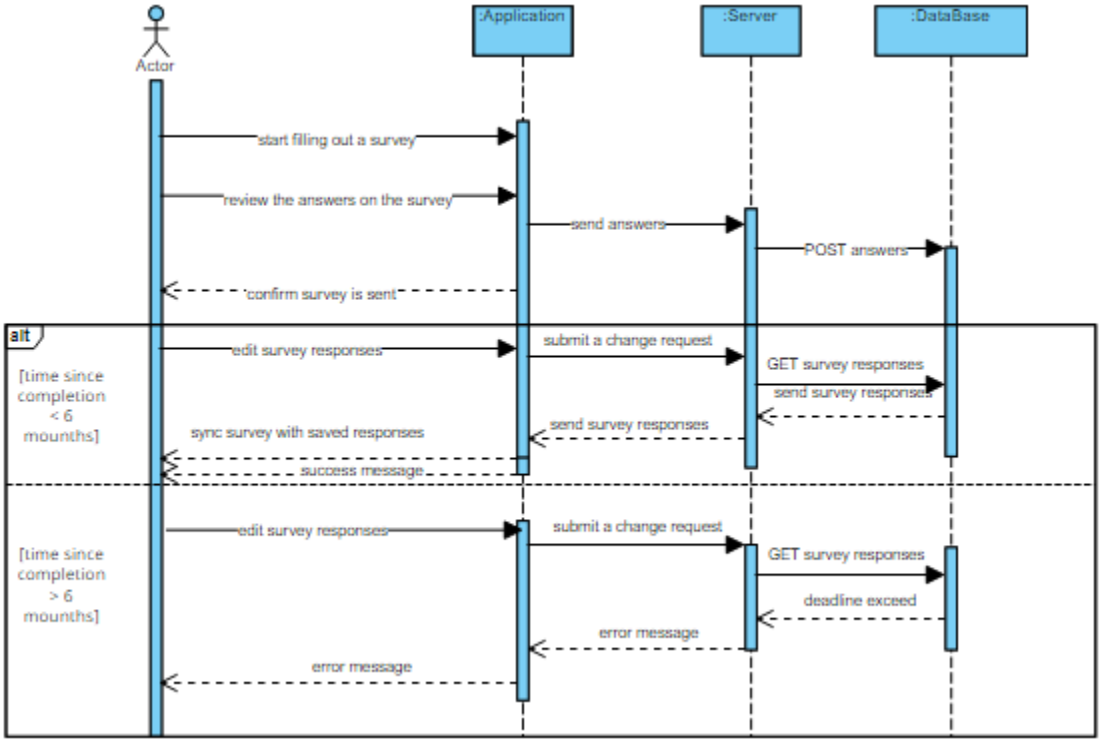

Figure 23. Vue de processus concernant la réponse de l'utilisateur à un questionnaire.

6. Vue de déploiement

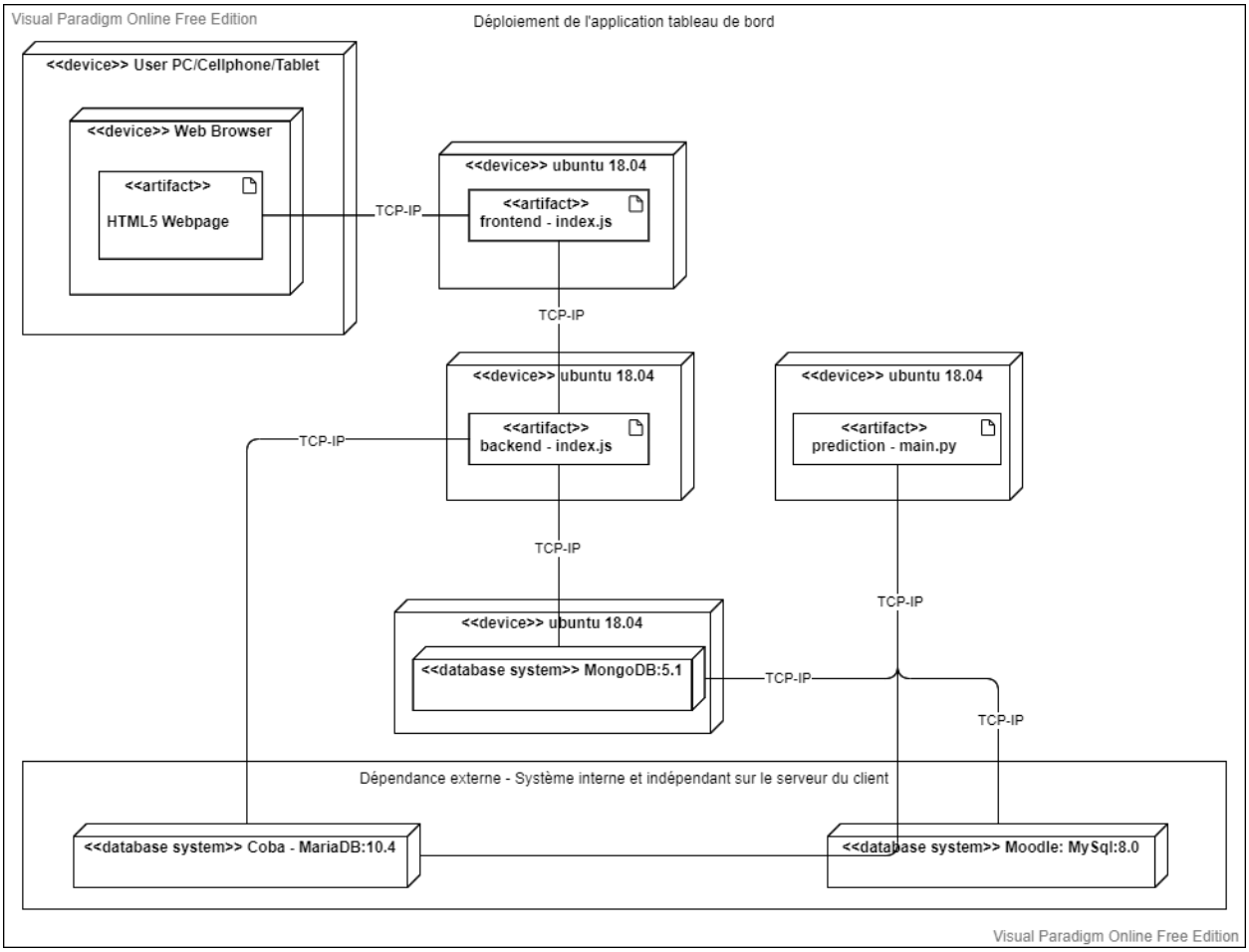

Figure 24. Diagramme de déploiement - Tableau de bord

La *Figure 24* présente le diagramme de déploiement de l'application. L'application est divisée en plusieurs systèmes. Tous les systèmes communiqueront par TCP-IP. Le client peut accéder à l'application à partir d'un fureteur. Ce fureteur sera connecté à une machine linux-debian:11 dans laquelle une instance de alpine:3.15.0 qui contiendra le serveur *frontend*. Le serveur *frontend* sera connecté à une instance alpine:3.15.0 qui contiendra le serveur *backend*. Le serveur *backend* et le serveur de prédiction seront connectés à une instance de MongoDB:5.1 qui servira de base de données pour l'application. Les dépendances externes administrées par le client ont été ajoutées dans un objectif de clarification. L'équipe de développement n'a aucun impact sur ces systèmes et l'application ne fait que consommer les données produites par ses systèmes.

7. Taille et performance

7.1 Contrainte de performance

La principale contrainte de performance est de supporter un grand nombre d'utilisateurs simultanément. Le client sait déjà que le nombre total d'utilisateurs présents simultanément sur le tableau de bord dépasse rarement 100. Cependant, il faut s'assurer que la performance restera adéquate même lorsque ce nombre est atteint. La marge de sécurité sera donc de 500 utilisateurs simultanément avec une latence maximale de 500 ms pour le TTFB. Cette contrainte sera testée avec un test de performance. Comme le tableau de bord sera déployé sur le serveur du client, l'équipe de développement n'a pas de contrôle sur la performance du matériel. Toutefois, l'application sera divisée en conteneurs et il sera facile pour l'administrateur de système du Cégep à distance de déployer de nouvelles instances de services si le réseau devient surchargé. La recommandation d'utiliser Kubernetes pour que cette gestion soit automatisée sera faite au client.

7.2 Contrainte de taille

La principale contrainte de taille est d'enregistrer les données relatives à l'application de la totalité des étudiants qui sont inscrits au Cégep à distance. Le client a indiqué que le nombre moyen d'étudiants par année est de 16000 et qu'ils suivent en moyenne 2 cours. Le client a aussi annoncé qu'il possède l'infrastructure matérielle nécessaire pour enregistrer ces données sur son serveur local.