Quentin Forestier

TB – 16.05.2022

Gryffinium

Travail de bachelor



Table des matières

[Cahier des charges 3](#_Toc103532488)

[Problématique 3](#_Toc103532489)

[Solutions existantes 3](#_Toc103532490)

[Objectif 3](#_Toc103532491)

[Jalon 3](#_Toc103532492)

[Fonctionnalités de l’application 3](#_Toc103532493)

[Fonctionnalité principale du diagramme 3](#_Toc103532494)

[Fonctionnalité supplémentaire du diagramme 4](#_Toc103532495)

[Fonctionnalité de gestion 4](#_Toc103532496)

[Échéance 4](#_Toc103532497)

[Livrables 4](#_Toc103532498)

[Planning 4](#_Toc103532499)

[Méta schéma 6](#_Toc103532500)

[Description du méta schéma 7](#_Toc103532501)

[Framework 7](#_Toc103532502)

[Analyse de l’existant 7](#_Toc103532503)

[Spring 7](#_Toc103532504)

[Play ! 7](#_Toc103532505)

[Choix effectué 8](#_Toc103532506)

[Base de données 8](#_Toc103532507)

[SGBD 8](#_Toc103532508)

[ORM 8](#_Toc103532509)

[Analyse de l’existant 8](#_Toc103532510)

[Choix effectué 9](#_Toc103532511)

[Libraires graphiques 9](#_Toc103532512)

[Design de l’interface global 9](#_Toc103532513)

[Design de diagramme de classes 9](#_Toc103532514)

[Fonctionnement de l’application 9](#_Toc103532515)

[Login / Register 9](#_Toc103532516)

[Sécurités et validations mises en place 9](#_Toc103532517)

[Diagramme de séquences 10](#_Toc103532518)

[Gestion de projets 11](#_Toc103532519)

[Projet 11](#_Toc103532520)

[Collaborateurs 17](#_Toc103532521)

[Collaboration sur les projets 20](#_Toc103532522)

[Websocket 20](#_Toc103532523)

[Commandes 21](#_Toc103532524)

[Interface utilisateur 22](#_Toc103532525)

[Page d’accueil pour utilisateur non connecté 22](#_Toc103532526)

[Page d’accueil pour utilisateur connecté 23](#_Toc103532527)

[Modal d’authentification 24](#_Toc103532528)

[Modal d’inscription 24](#_Toc103532529)

[Page d’accueil avec la liste des projets 25](#_Toc103532530)

[Modal de création et mise à jour de projet pour le propriétaire 26](#_Toc103532531)

[Modal de visualisation des détails de projet 27](#_Toc103532532)

[Modal de chat des projets 28](#_Toc103532533)

[Conclusion 29](#_Toc103532534)

[Problèmes rencontrés 29](#_Toc103532535)

[Compréhension des Websocket et de la librairie Akka 29](#_Toc103532536)

[Utilisation d’Hibernate 29](#_Toc103532537)

[Améliorations et problèmes 29](#_Toc103532538)

[Gestion de la panne 29](#_Toc103532539)

[Abstraction de la logique dans une couche de service 29](#_Toc103532540)

[Mise en place de DTO pour les échanges clients-serveurs 29](#_Toc103532541)

[Table des illustrations 30](#_Toc103532542)

[Annexes 30](#_Toc103532543)

# Cahier des charges

## Problématique

Le but de ce travail de Bachelor est de répliquer et améliorer les fonctionnalités de Slyum, un éditeur de diagramme de classes UML développé en Java à la HEIG-VD. Il est notamment souhaité que l’application puisse proposer une collaboration sur les diagrammes, sans devoir passer le fichier entre les différents utilisateurs.

## Solutions existantes

* Slyum
* StarUML
* Umletino
* …

## Objectif

L’objectif de ce travail est de permettre d’avoir un éditeur de diagramme de classe sur le web, avec une collaboration simplifiée. Une collaboration instantanée est envisagée, mais dépendra du temps à disposition.

## Jalon

Dans un premier temps, il faudra définir un métaschéma au moyen d’un diagramme de classes.

Dans un deuxième temps, il s’agira de se familiariser avec le Framework Play!, et de voir dans quelle mesure il est possible d’utiliser les websockets afin de pouvoir collaborer. Une fois cela fait, il faudra choisir si oui ou non, il est possible d’avoir une coopération en direct sur l’édition de diagrammes. Pour se faire, une application de chat simplifiée sera mise en place.

La gestion d’utilisateur sera également implémentée dans l’application, ainsi que la gestion des droits. Ces informations seront stockées dans une base de données PostgreSQL, et les échanges seront mis en place grâce à un ORM.

Une fois que les tous les mécanismes de communication seront mis en place, l’implémentation du métaschéma, ainsi que de la partie graphique sera à faire. Pour la partie graphique, il faudra effectuer une recherche sur les différentes librairies JavaScript permettant un affichage et des modifications simples et intuitives du diagramme.

Une fois la conception du métaschéma, ainsi que la familiarisation avec Play! effectué, l’application et ses fonctionnalités seront codés en Java pour la majeure partie, et en JavaScript pour l’affichage du diagramme. L’évolution de l’application suivra les jalons, de sorte que chaque étape soit déjà un résultat.

## Fonctionnalités de l’application

### Fonctionnalité principale du diagramme

* Création de classes, interfaces, énumération, classes abstraites
* Création d’attributs et de méthodes (abstraites ou non)
* Création d’associations (binaire, n-aire, compositions, agrégation, classes d’association)
* Création de relation d’héritage
* Création de relation de dépendances
* Affichage des éléments sous format graphique
* Possibilité de modifier graphiquement le diagramme
* Exportation sous format graphique
* Sérialisation/Désérialisation des diagrammes

### Fonctionnalité supplémentaire du diagramme

* Possibilité d’effectuer une annulation de la dernière action
* Possibilité d’avoir différentes vues du diagramme
* Possibilité de dupliquer une entité du diagramme
* Mise en place de raccourcis clavier
* Possibilité d’écrire les attributs/fonctions sous forme de texte, qui sera ensuite transformé en entité

### Fonctionnalité de gestion

* Inscription/Connexion
* Création de projets
* Création de groupes d’utilisateurs
* Gestion des droits dans le groupe
* Gestion des membres du groupe
* Quitter un groupe
* Ouverture des diagrammes de classes
* Coopération directe / indirecte sur un diagramme
  + Dans le cas d’une coopération instantanée, l’accès aux autres utilisateurs sera bloqué sur l’élément du diagramme sélectionné.
  + Dans le cas d’une coopération indirecte, l’accès au diagramme sera bloqué si un autre utilisateur travaille déjà dessus

## Échéance

|  |  |
| --- | --- |
| Date | Tâche |
| 14 avril 2022 | Cahier des charges |
| 16 mai 2022 | Rapport intermédiaire |
| 29 juillet 2022 | Rendu des livrables |
| 26 août 2022 | Résumé publiable |

## Livrables

* Une application Java utilisant le framework Play! dans un container Docker
* Un protocole de tests afin de garantir la fonctionnalité de l’application
* Un rapport comprenant :
  + Les choix de conception
  + Les problèmes rencontrés
  + Les solutions envisagées
  + La synthèse du résultat obtenu
* Un manuel utilisateur décrivant les fonctionnalités de l’application

## Planning

|  |  |
| --- | --- |
| Tâches | Échéance |
| Métaschéma | 1er avril 2022 |
| Familiarisation avec Play! et application de chat basique | 22 avril 2022 |
| Mise en place des utilisateurs et des droits sur les projets avec toutes les fonctionnalités de gestion | 13 mai 2022 |
| Recherche de la librairie graphique pour les diagrammes | 27 mai 2022 |
| Implémentation de l’UML et choix de la méthode pour sauvegarder les diagrammes (XML ou relationnel) | 17 juin 2022 |
| Fonctionnalités principales du diagramme | 15 juillet 2022 |
| Fonctionnalités supplémentaires du diagramme | 29 juillet 2022 |

Les tests et la documentation seront en accord avec les différentes étapes mentionnées ci-dessus.

# Métaschéma

Une image contenant texte, signe

Description générée automatiquement

Figure 1 : Métaschéma

## Description du métaschéma

#### En orange

Toutes les représentations de types et entités possibles.

PrimitiveType représente les types primitifs, les types sont donc exhaustifs. Elle fonctionne comme une enum.

SimpleType permet d’ajouter un type, qui est simplement une String, ce qui permet donc d’avoir un type qui n’est pas représenté par une entité (utile dans le cadre d’une utilisation de librairie).

L’interface Implementor sert uniquement pour le lien de réalisation. Elle permet de spécifier quelles entités peuvent implémenter une interface.

L’interface InnerEntity sert uniquement pour le lien Inner. Elle permet de spécifier quelles entités peuvent être dans une autre entité.

#### En vert

Tous les attributs et méthodes disponibles dans les entités.

#### En bleu

Tous les liens possibles d’un diagramme de classe.

Role permet, pour chaque entité de l’association, d’avoir un nom, une visibilité et une multiplicité.

#### En blanc

ExistingTypes est un singleton qui permet d’avoir une liste des types existants. Le changement de nom d’un type se répercutera sur toutes les variables et opérations l’ayant pour type.

# Framework

## Analyse de l’existant

J’ai tout d’abord effectué une recherche des frameworks existants me permettant de travailler sur le Web et qui fonctionne avec Java.

Je n’ai pas recherché à approfondir tous les frameworks existants, simplement les plus connus et utilisés.

### Spring

Spring permet de définir des routes accessibles en HTTP, d’effectuer un traitement en Java et de rendre une page Web, ou toutes autres réponses souhaitées. Il possède également un moteur de templating afin de construire les pages Web côté serveur.

Ayant déjà eu une expérience avec Spring, j’ai pu expérimenter son côté très strict. Il est dirigiste envers le choix de la structure, du code et de la communication avec la base de données.

Spring possède tout de même une grande popularité et est très souvent utilisé. La documentation est large, même s’il arrive que beaucoup de choses se fassent dans le dos du développeur, sans explications réelles dans la documentation.

### Play !

Play ! permet également de définir des routes accessibles en HTTP, d’effectuer le traitement en Java et de rendre la réponse sous forme souhaitée (page web, json, xml, …).

Il est cependant bien moins strict, il possède un moteur de templating par défaut, Twirl, cependant il est changeable facilement. Il ne possède pas un ORM en particulier, mais supporte une liste directement intégrée, et à en plus la possibilité d’étendre grâce à des extensions les ORM utilisable.

## Choix effectué

J’ai décidé d’utiliser Play! car son côté permissif et simple à mettre en place me sera bénéfique. Il répond amplement à mes attentes, la documentation est suffisante, bien que majoritairement faite pour Java et Scala, mais décrite et expliquée uniquement pour Scala.

Ce qui a également fait pencher la balance est les conseils du responsable, M. Donini, et son entourage. En effet, étant habitué à utiliser Play!, il sera alors possible d’obtenir des conseils plus facilement.

# Base de données

## SGBD

Ayant principalement eu des expériences avec des SGBD relationnelles et n’ayant pas trouvé d’avantages importants aux SGBD non relationnelles, j’ai décidé de ne pas trop approfondir ce sujet.

Ayant déjà utilisé MySQL et PostgreSQL, j’ai décidé de choisir PostgreSQL car plus récent, plus modulable et qu’il est largement utilisé. Il possède également une architecture orientée objet, ce qui match avec le Framework Play ! qui est aussi orienté objet.

# ORM

## Analyse de l’existant

Play ! met à disposition des plug-ins pour une liste d’ORM. Ebean est l’ORM par défaut lorsque l’on utilise Java. La documentation de Play ! spécifie qu’il n’existe pas d’implémentation de JPA directement, mais qu’il est possible d’ajouter la dépendance au projet.

Cela dit, j’ai tout d’abord essayé d’implémenter Ebean, qui supporte PostgreSQL. L’ORM permet d’annoter les classes représentant les tables, ainsi que les différents attributs (NotNull, ManyToOne, OneToMany, etc…). Il est possible de communiquer avec la base de données aux moyens de fonction, sans écrire la moins ligne de SQL.

Ebean a malheureusement un point faible. Il ne gère pas l’héritage. Il est dès lors impossible de stocker le métaschéma, fait précédemment, de manière relationnelle. Il faudrait alors stocker le diagramme sous forme d’XML.[[1]](#footnote-1)

J’ai donc essayé de changer d’ORM et de passer à Hibernate. Comme dit plus haut, JPA n’est pas directement supporté, dès lors j’ai dû ajouter la dépendance au projet. J’ai cependant aperçu une limite très rapidement. Dès qu’une requête n’est plus basique (comme un simple select/update/delete avec id), ou qu’une jointure de table est à faire, il est nécessaire d’écrire le SQL à la main.

Spring met en place une JpaRepository, qui génère automatiquement la requête SQL à faire en fonction du nom de la méthode. Play ! ne possède pas ce mécanisme, et dès lors, je trouve qu’utiliser Hibernate n’apporte pas énormément, voir complique même les choses.

## Choix effectué

Étant donné que l’héritage peut être non requis en enregistrant le diagramme sous format XML, j’ai décidé de continuer à travailler avec Ebeans. J’ai trouvé bien plus simple d’utilisation et à mettre en place, et il est supporté par défaut par Play!.

# Libraires graphiques

## Design de l’interface global

Toute l’interface des pages Web est faite avec le moteur de template par défaut de Play!, Twirl. Il me permet de gérer les erreurs de formulaires très simplement, d’utiliser la librairie graphique que je souhaite, Bootstrap.

## Design de diagramme de classes

A rechercher

# Fonctionnement de l’application

## Login / Register

Le login et le register se font de manière synchrone. Le formulaire est envoyé, et le serveur renvoie une nouvelle page. Twirl et Play! ayant un système de gestion d’erreurs pour les formulaires, c’était la façon la plus simple d’arriver à rendre un feedback précis à l’utilisateur.

Les conditions sur chaque input sont gérées grâce à Twirl et Play !, tandis que l’unicité d’une adresse email est gérée à la main, en questionnant la base de données.

### Sécurités et validations mises en place

Le mot de passe de l’utilisateur est hashé et salé grâce à l’utilitaire BCrypt. [[2]](#footnote-2)

Validation des entrées utilisateurs :

* Email : L’email est validé à l’inscription en utilisant le Constraint.Email de Play !
* Mot de passe : Le mot de passe est validé à l’inscription avec un regex (8 caractères, un chiffre, une majuscule, une minuscule et un caractère spécial)
* Nom : Le nom est valide s’il a plus de 2 caractères
* Confirmation du mot de passe : Vérification que le mot de passe est bien égal à la confirmation

Toutes ces entrées utilisateurs sont requises, autant pour l’authentification que pour l’enregistrement d’un utilisateur.

### Diagramme de séquences

#### Login

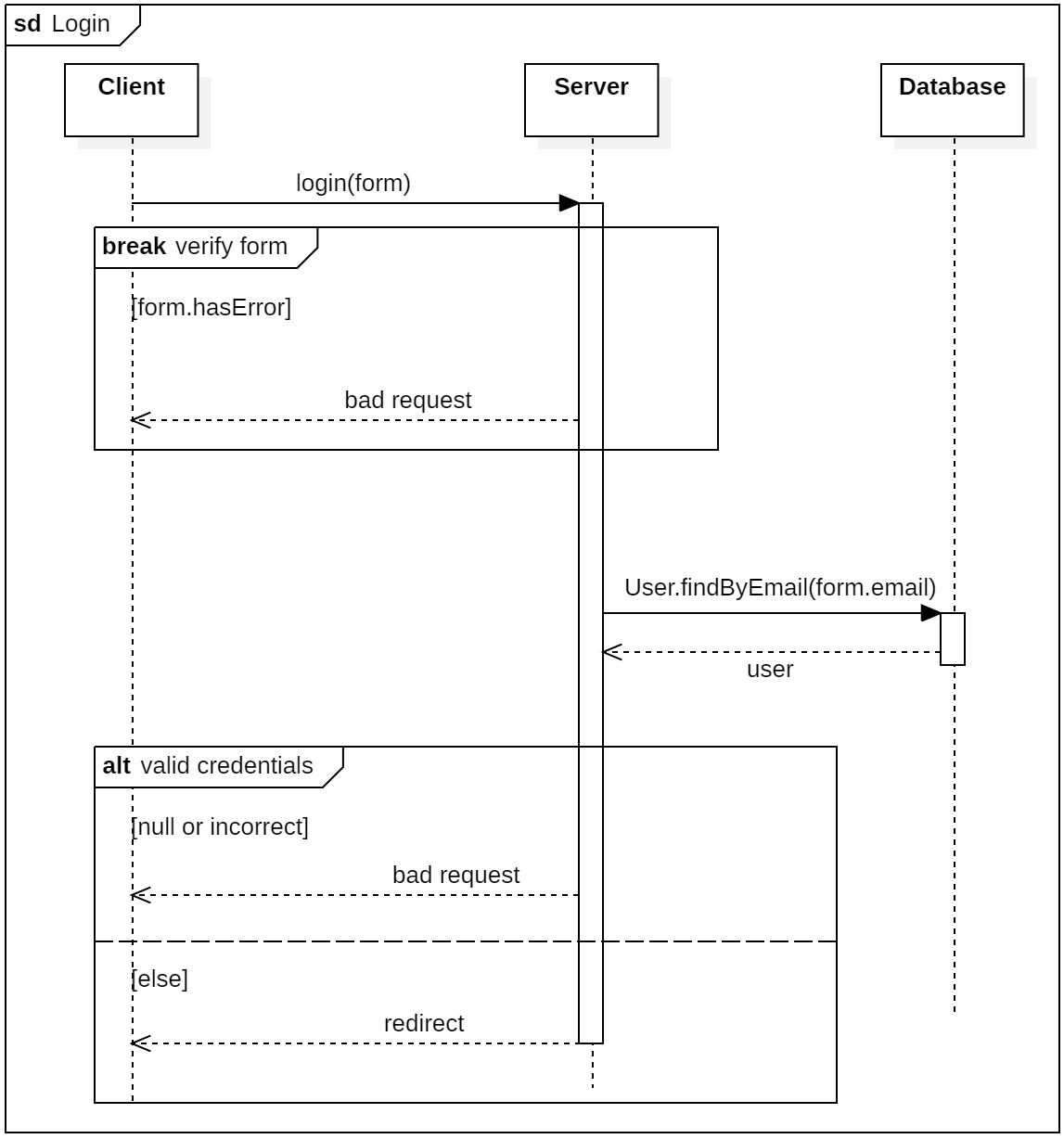


Figure 2 : Diagramme de séquence de l’authentification

Le paramètre « form » représente le formulaire envoyé, avec tous les champs nécessaires à l’authentification.

#### Register

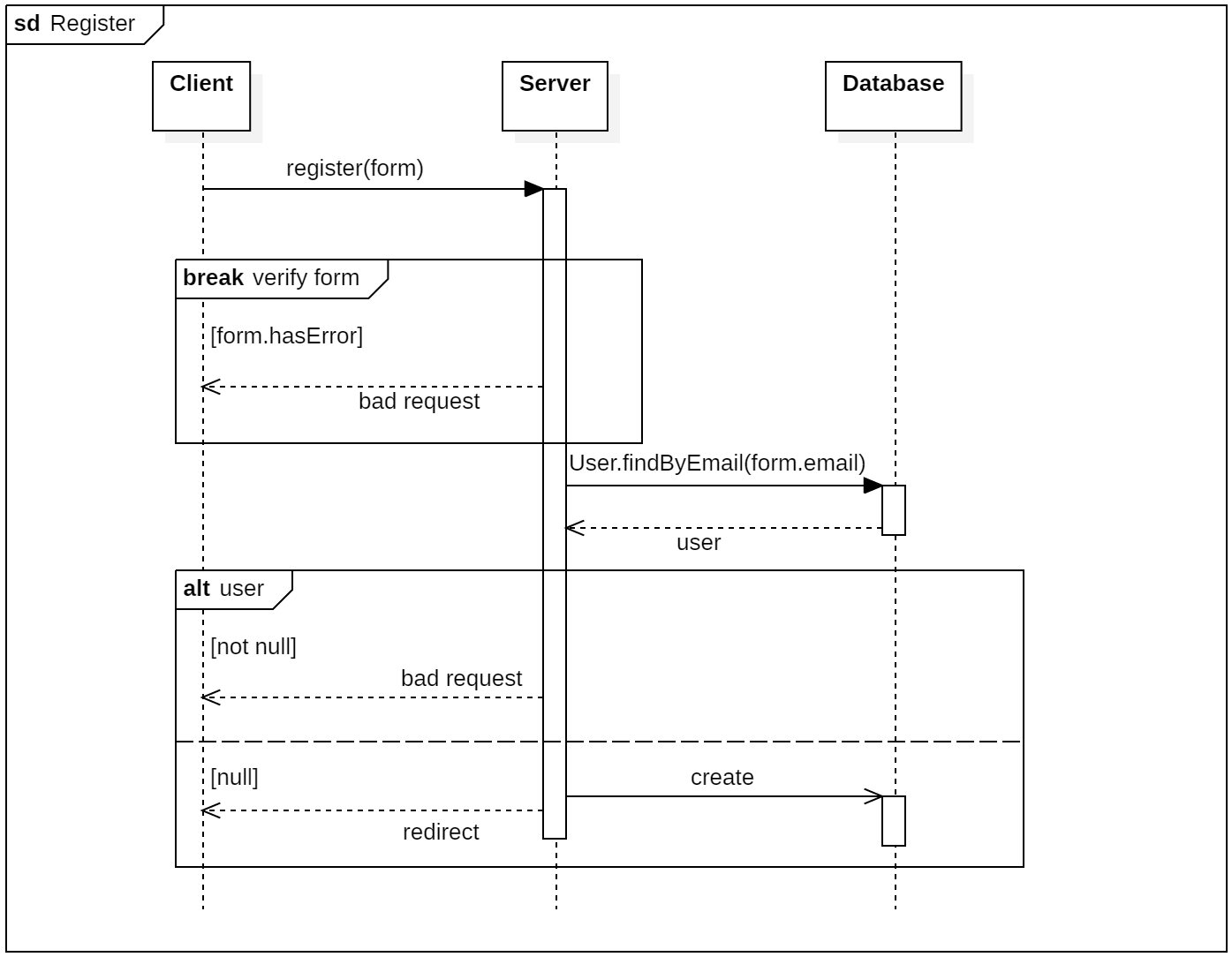


Figure 3 : Diagramme de séquence Register

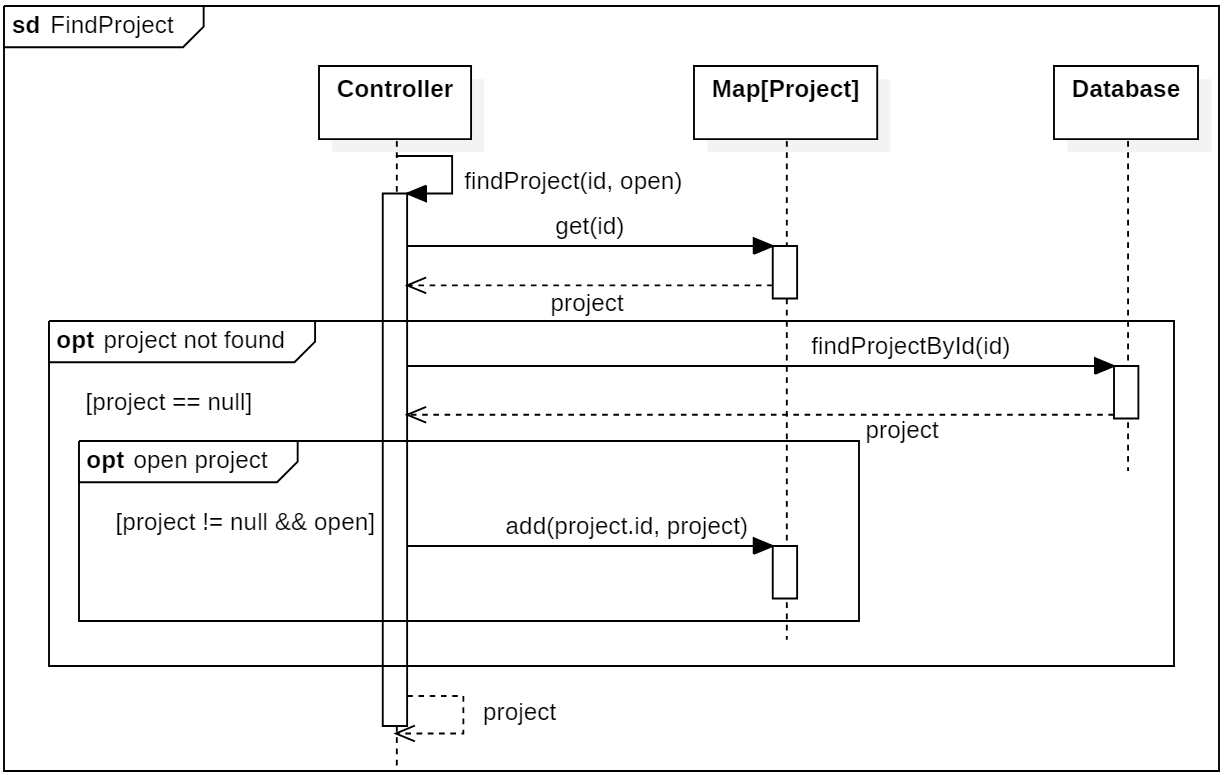
Le paramètre « form » représente le formulaire envoyé, avec tous les champs nécessaires à l’authentification.

## Gestion de projets

Toutes les routes concernant les projets sont protégées. Il faut être authentifié afin d’avoir accès à la route. La vérification d’authentification n’est donc pas mise sur les diagrammes qui suivent afin de les alléger.

### Projet

Afin de favoriser l’expérience utilisateur, toutes les interactions de gestion de projets sont faites en asynchrones, depuis du code JavaScript présent dans les vues. Cela permet de ne pas recharger la page à chaque modification effectuée.



Une map de projets ouverts permet d’effectuer les modifications sur le projet correspondant et que la modification se répercute pour tous les utilisateurs en instantané. Ceci est tout d’abord utilisé pour le websocket, mais afin de limiter les appels à la base de données, cette liste est également utilisée pour les autres fonctionnalités.

Un projet n’étant pas déjà ouvert, mais existant dans la base de données peut être ajouté à la map ou non en fonction d’un paramètre.

Afin d’arriver à cette solution, je me suis inspiré d’un tutoriel qui utilise les Akka Actor pour travailler avec les websockets, afin de créer un serveur de messagerie en Scala.[[3]](#footnote-3)

#### Création

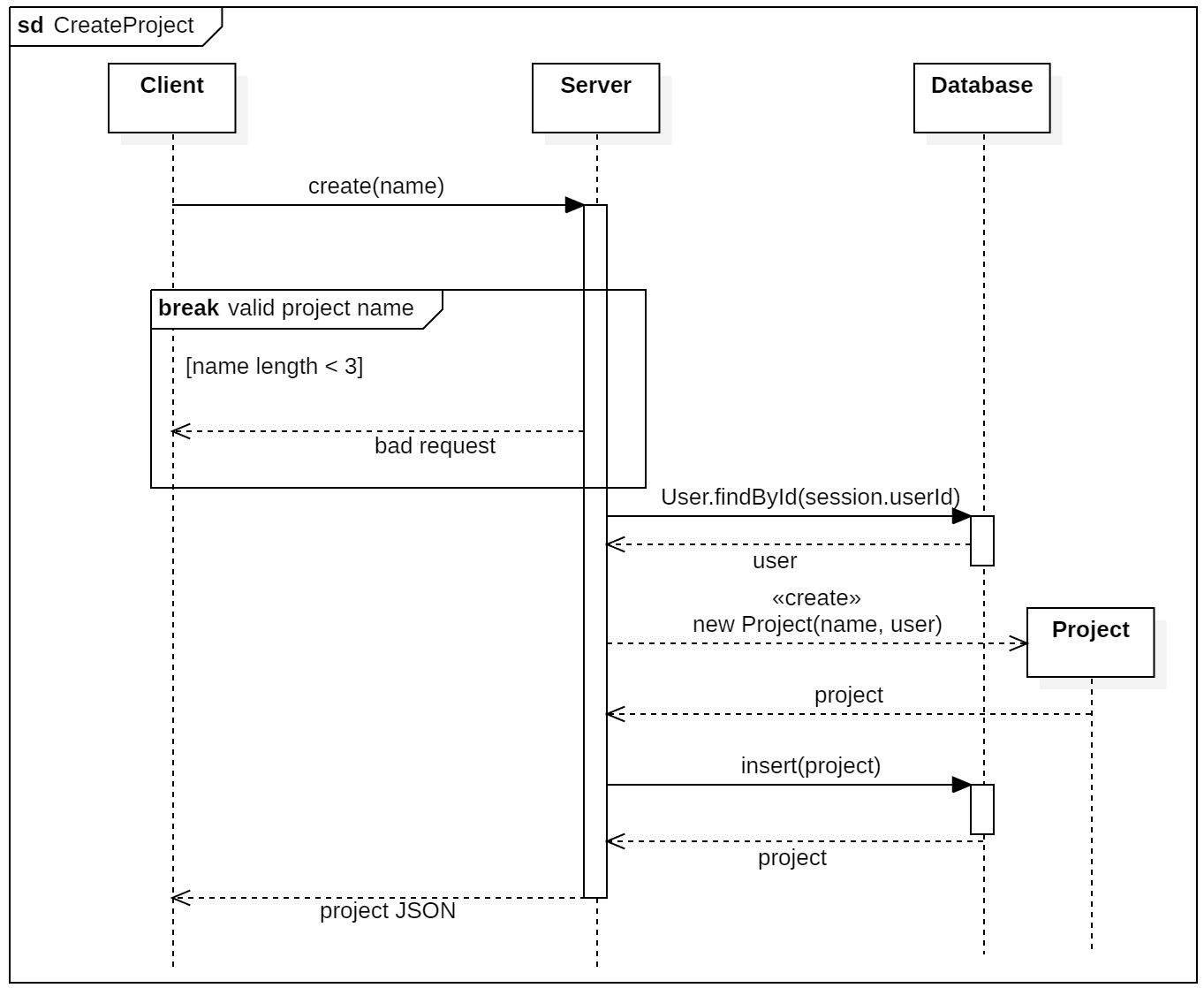


Figure 4 : Diagramme de séquence de la création de projets

Une validation est effectuée afin que le nom d’un projet ne soit pas plus petit que 3 caractères.

L’utilisateur authentifié est alors ajouté en tant que propriétaire du projet avec tous les droits.

#### Mise à jour

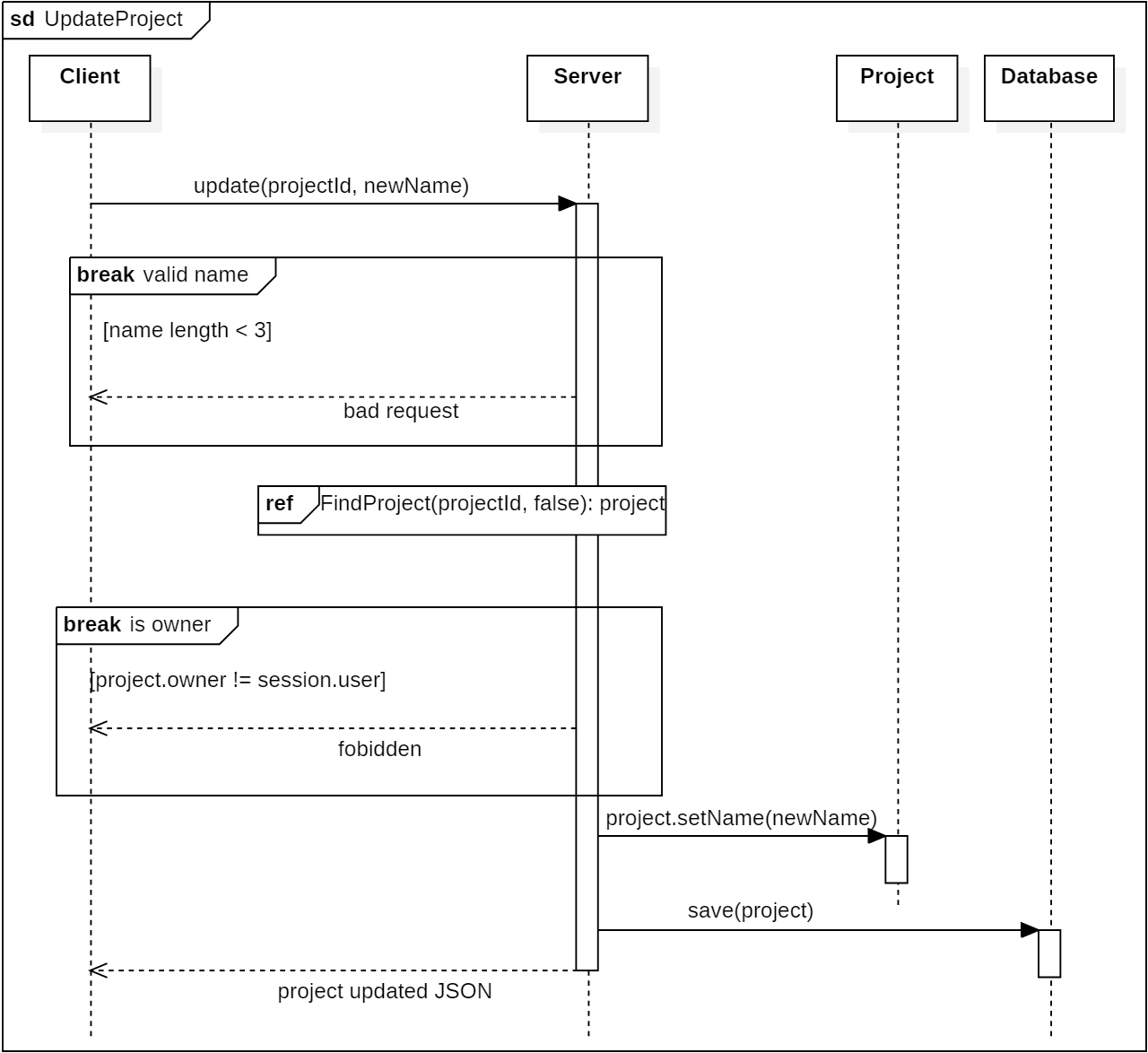


Figure 5 : Diagramme de séquence de la mise à jour d'un projet

On vérifie que le nom est correct, et que le projet a modifié appartient bien à l’utilisateur connecté avant de le modifier.

#### Suppression

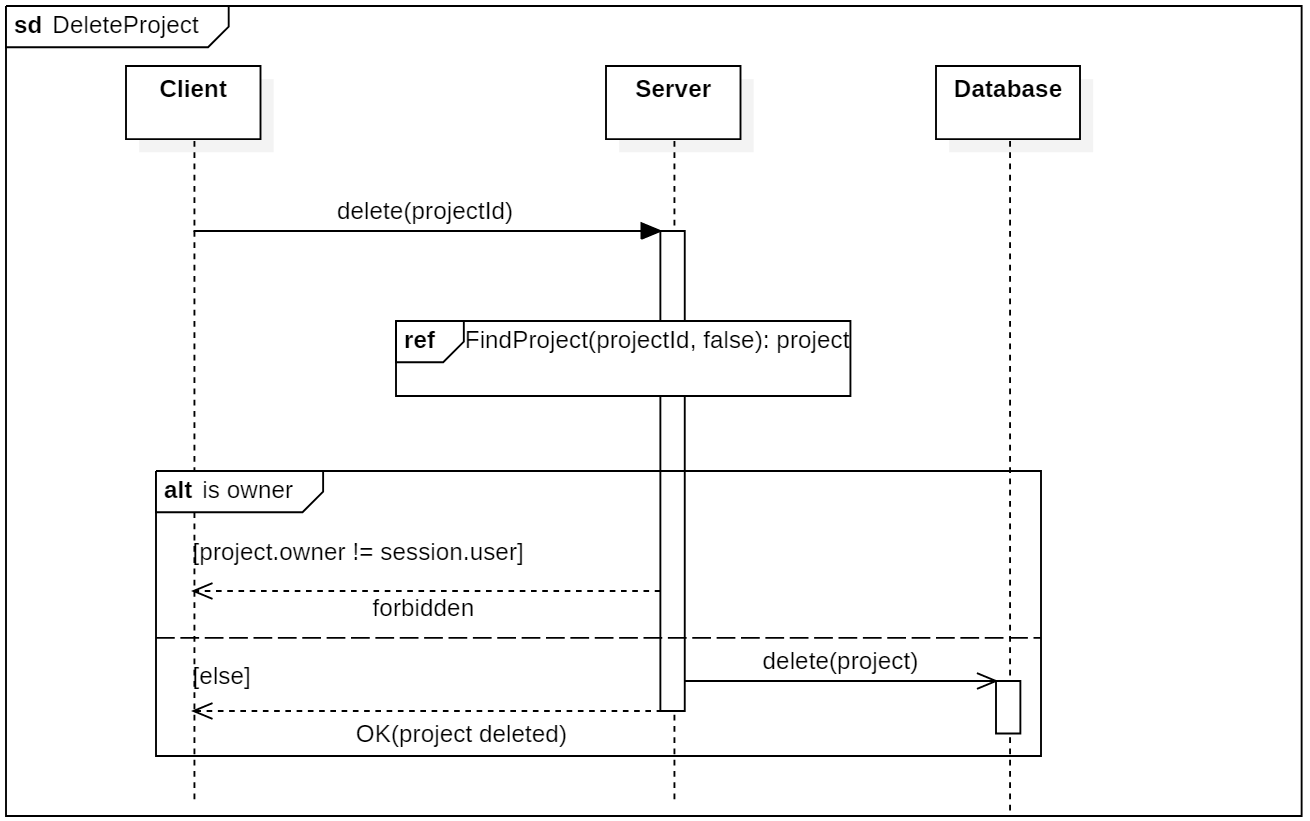


Figure 6 : Diagramme de séquence de la suppression d'un projet

On vérifie que le projet appartient bien à l’utilisateur connecté avant de le supprimer.

#### Récupération des projets d’un membre

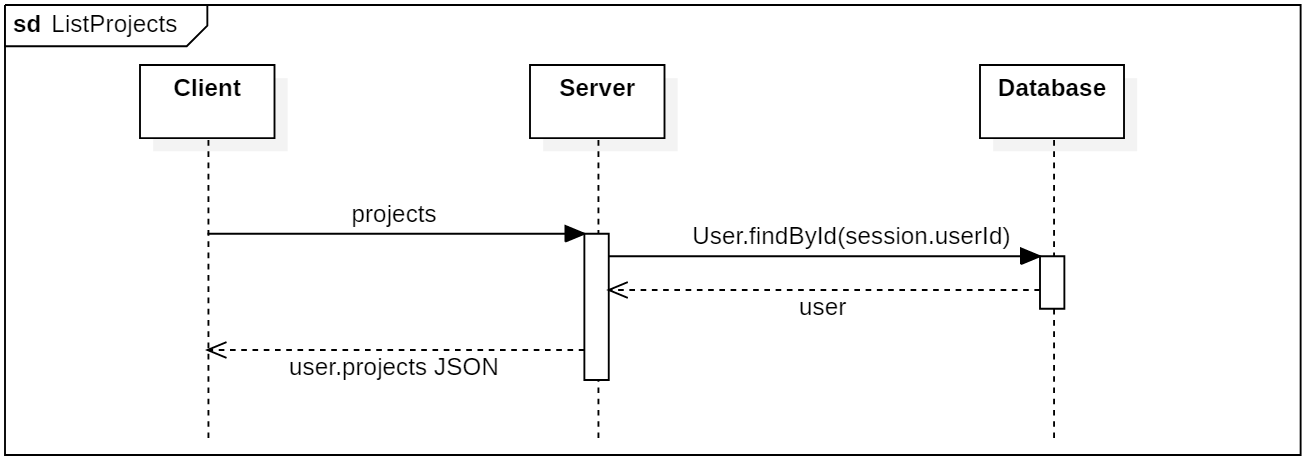


Figure 7 : Diagramme de séquence de la récupération des projets de l'utilisateur connecté

#### Récupération d’un projet spécifique et accès à la page du projet

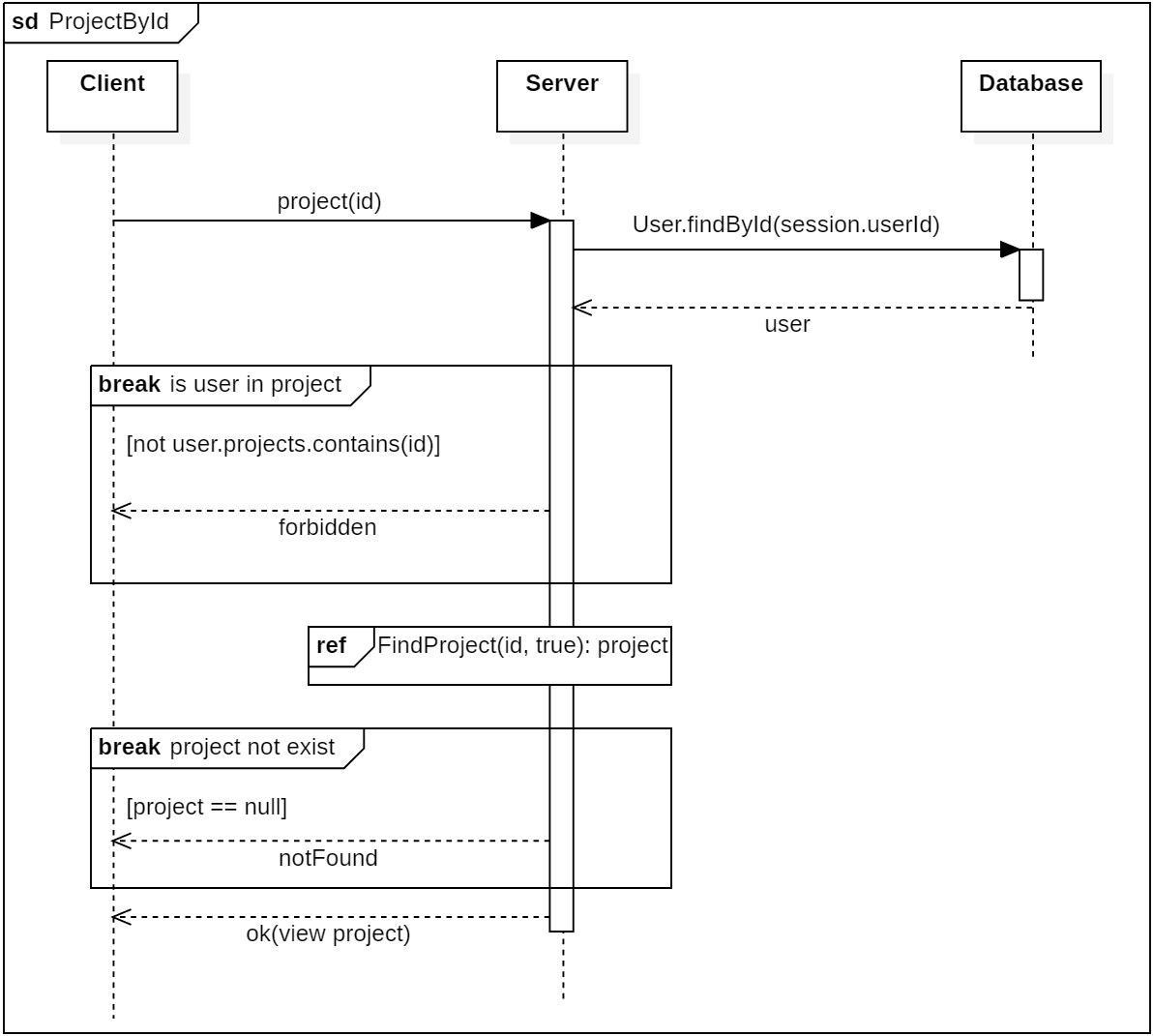


Figure 8 : Diagramme de séquence de la récupération d'un projet spécifique et de sa vue

On vérifie que l’utilisateur connecté appartient bien au projet. Si le projet n’est pas dans la map des projets ouverts, il y est ajouté.

### Collaborateurs

#### Ajout d’un collaborateur

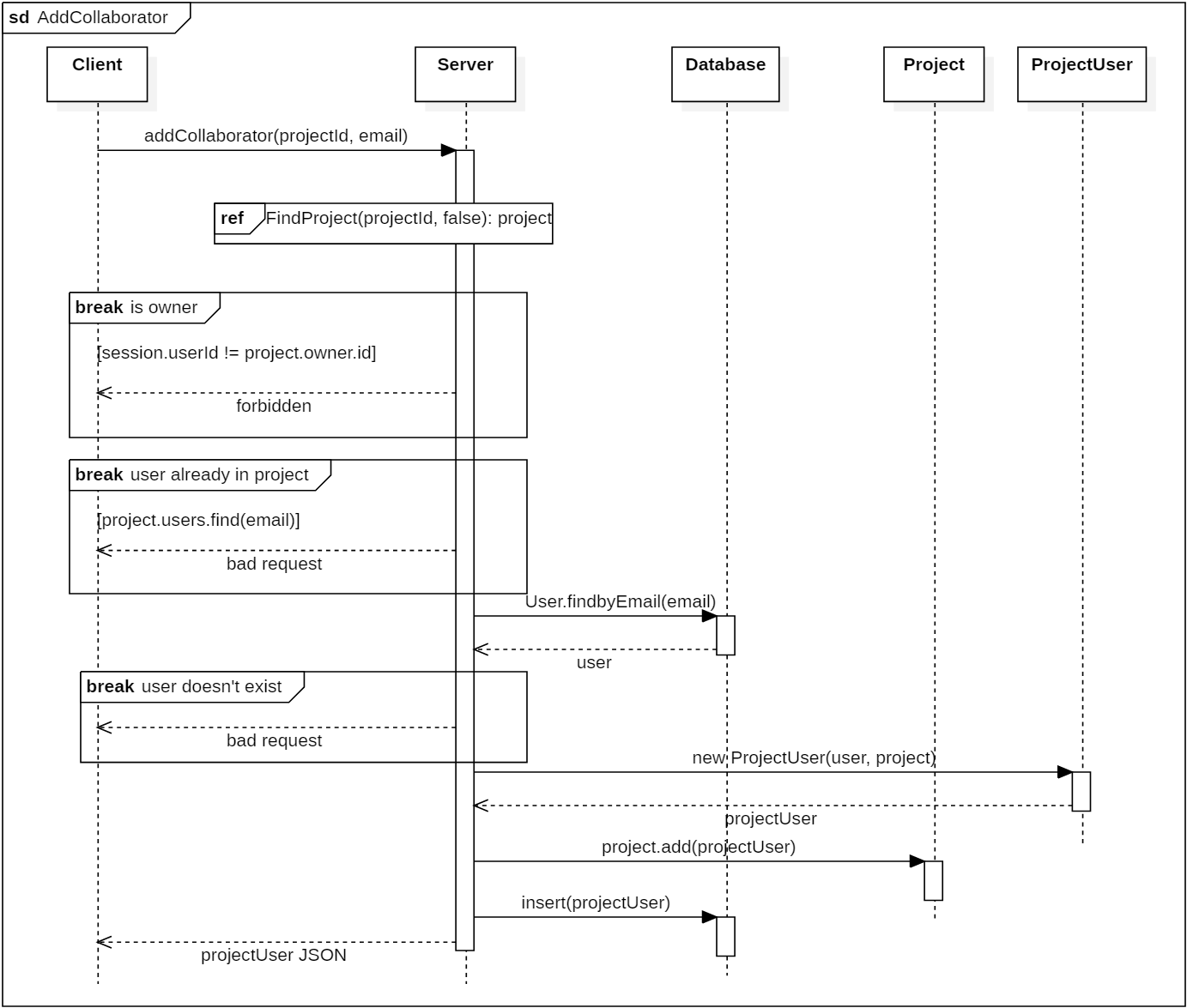


Figure 9 : Diagramme de séquence de l'ajout d'un collaborateur

Seul le propriétaire d’un projet peut ajouter un collaborateur.

Un collaborateur déjà dans le projet ne peut pas être ajouté une seconde fois.

#### Modification des droits d’un utilisateur

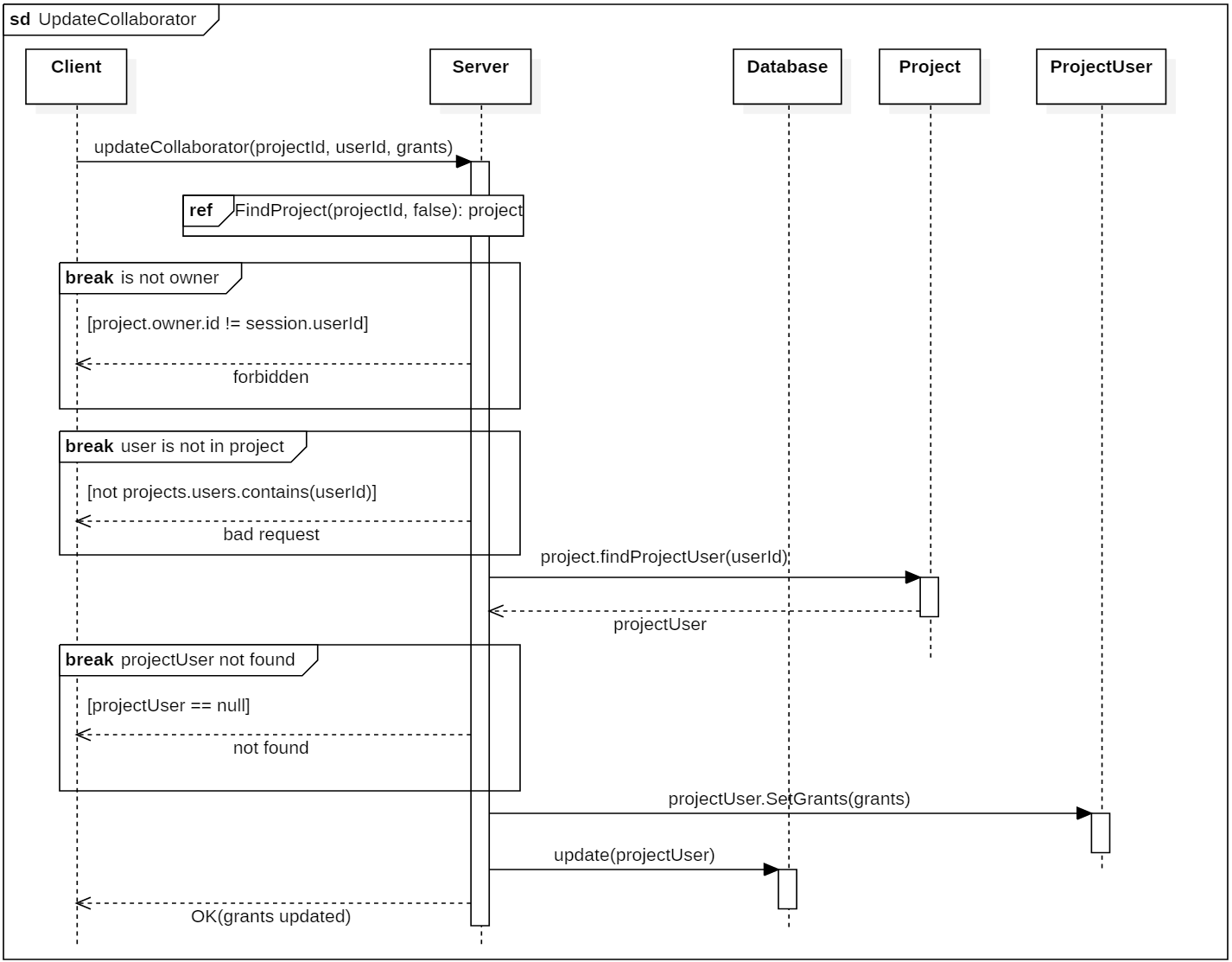


Figure 10 : Diagramme de séquence de la modification des droits d'un utilisateur

On vérifie que l’utilisateur connecté est bien le propriétaire, et que l’utilisateur a modifié appartient bien au projet.

#### Suppression d’un collaborateur

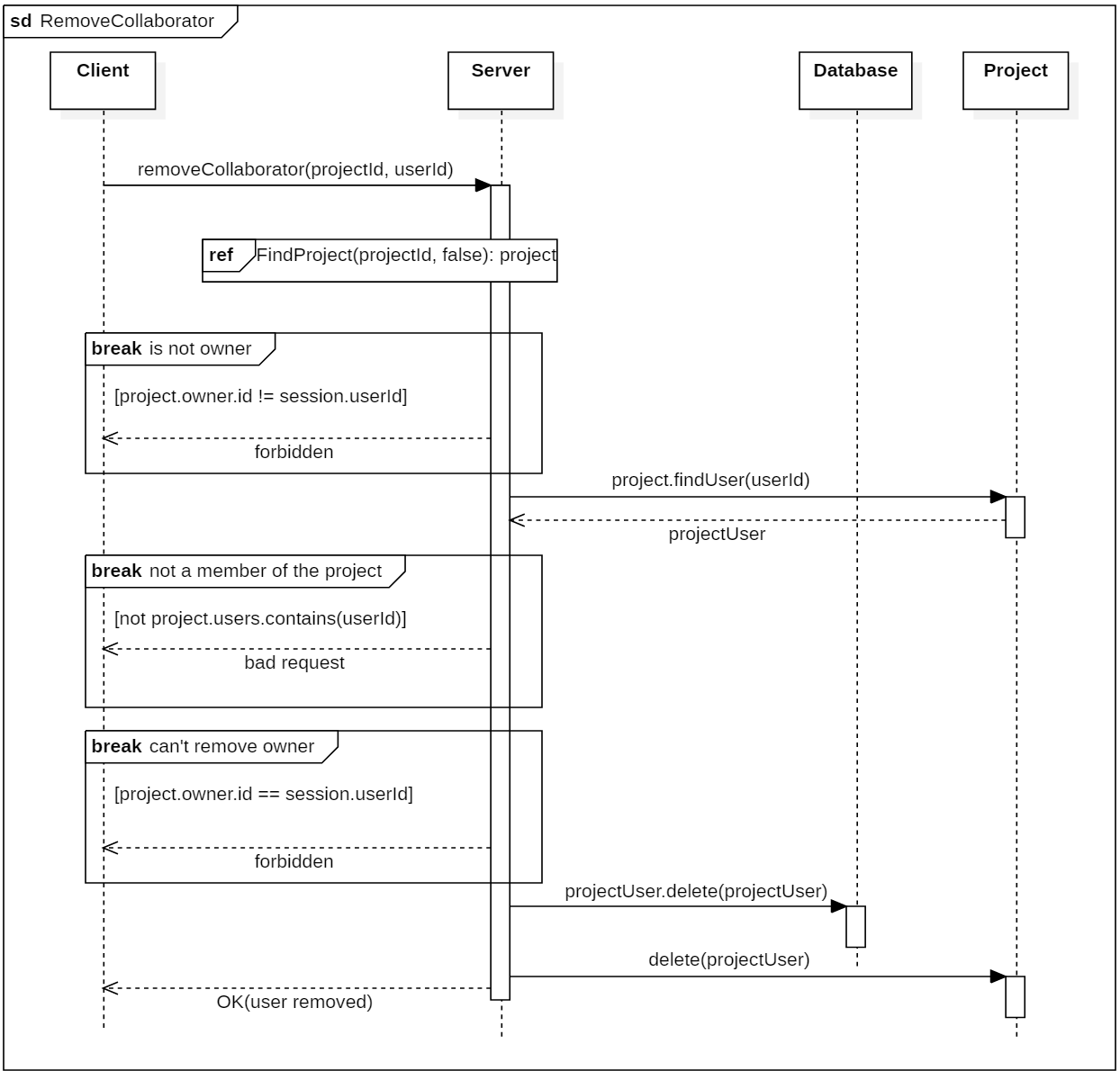


Figure 11 : Diagramme de séquence de la suppression d'un collaborateur

On vérifie que l’utilisateur connecté est le propriétaire du projet, et que le collaborateur a supprimé est bien un membre du projet et qu’il n’est pas le propriétaire du projet.

## Collaboration sur les projets

### Websocket

#### Connexion

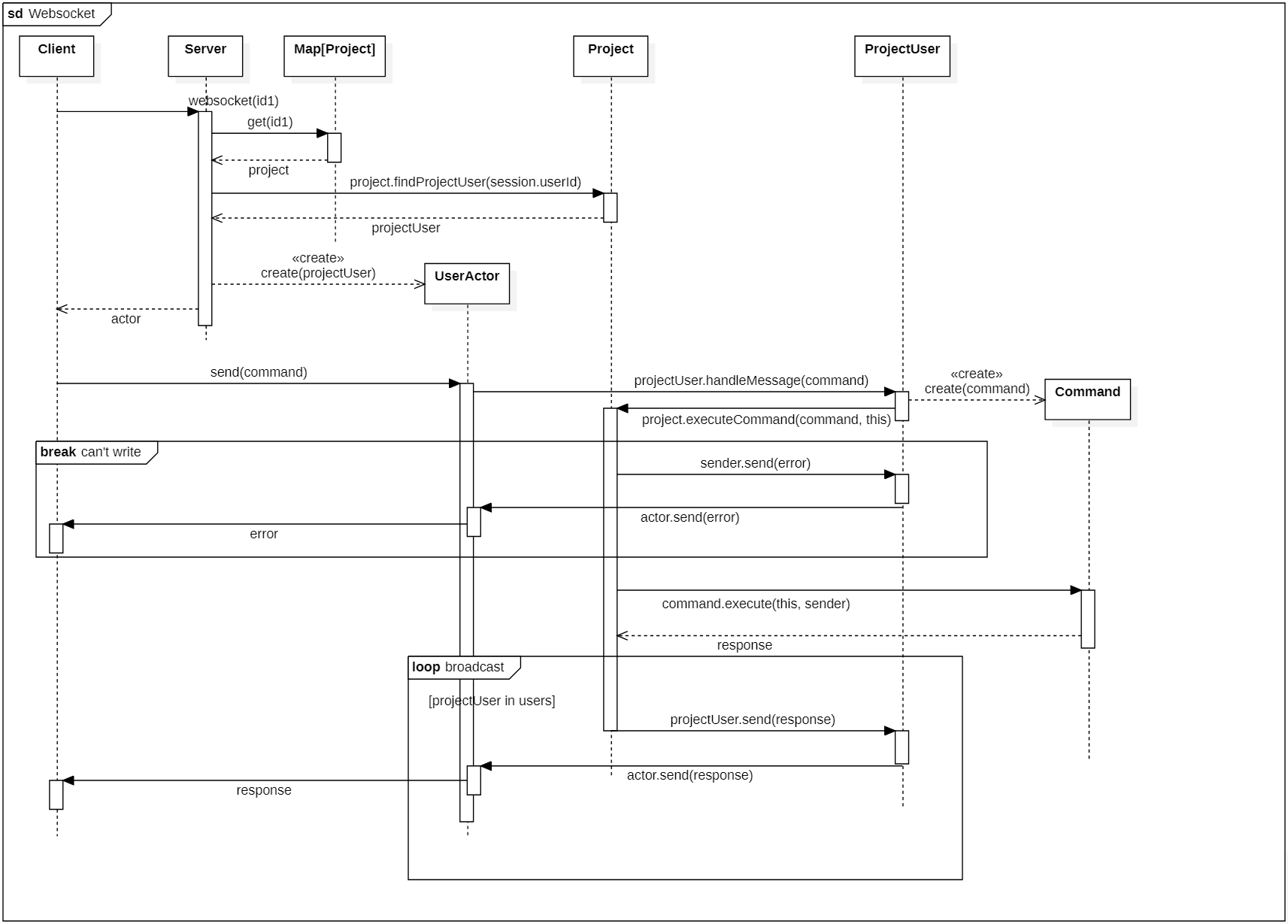


Figure 12 : Diagramme de séquence du websocket

Afin de pouvoir collaborer en instantané, il est nécessaire d’utiliser les websockets.

Les websockets sont gérés par des Akka.Actor. Ils permettent d’avoir des fonctionnalités, tels que la détection de déconnexion ou encore d’associer des objets métiers à un Actor.[[4]](#footnote-4)

Dans le cadre de ce projet, lors de la création de l’Actor, on lui associe un ProjectUser, symbolisant un utilisateur d’un certain projet. De cette manière, il est possible d’ouvrir différents websockets sans pour autant avoir des conflits. Chaque connexion aura son propre ProjectUser, qui aura une référence sur le bon projet.

#### Déconnexion

Une déconnexion d’un websocket peut arriver dans 2 cas.

##### Le client se déconnecte

L’Akka Actor va exécuter une fonction, qui permettra de notifier le projet qu’un utilisateur s’est déconnecté.

Si tous les utilisateurs sont déconnectés, le projet se ferme.

##### Le serveur déconnecte un client

Lorsqu’un projet est supprimé, ou qu’un collaborateur est retiré d’un projet, le serveur termine la connexion avec les utilisateurs concernés.

### Commandes

Les utilisateurs connectés à un websocket peuvent envoyer des messages JSON via ce dernier, qui seront traduits en Command. Ces commandes seront ensuite exécutées dans le contexte du projet en question.

Un message résultant de l’exécution de la commande est ensuite envoyé à tous les utilisateurs connectés.

Toutes les commandes qui ont été exécutées avec succès sont stockées dans une liste présente dans chaque ProjectUser. De cette manière, il est possible d’effectuer une annulation de la commande pour celles qui le permettent.

#### ChatMessage

Cette commande crée simplement une réponse contenant le nom de l’utilisateur l’ayant exécuté, ainsi que le message de ce dernier.

# Interface utilisateur

## Page d’accueil pour utilisateur non connecté

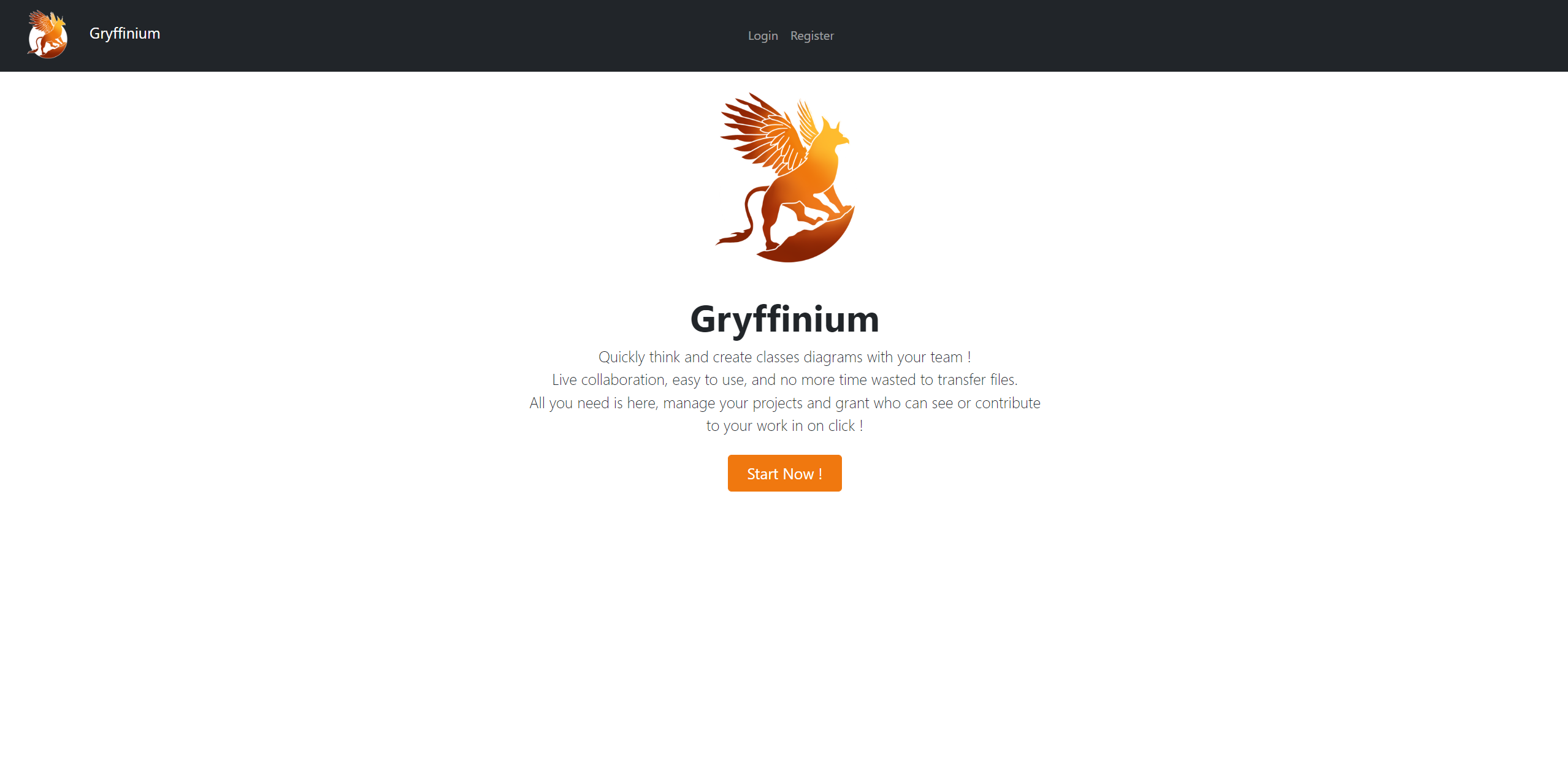


Figure 13 : Page d'accueil pour utilisateur non connecté

## Page d’accueil pour utilisateur connecté

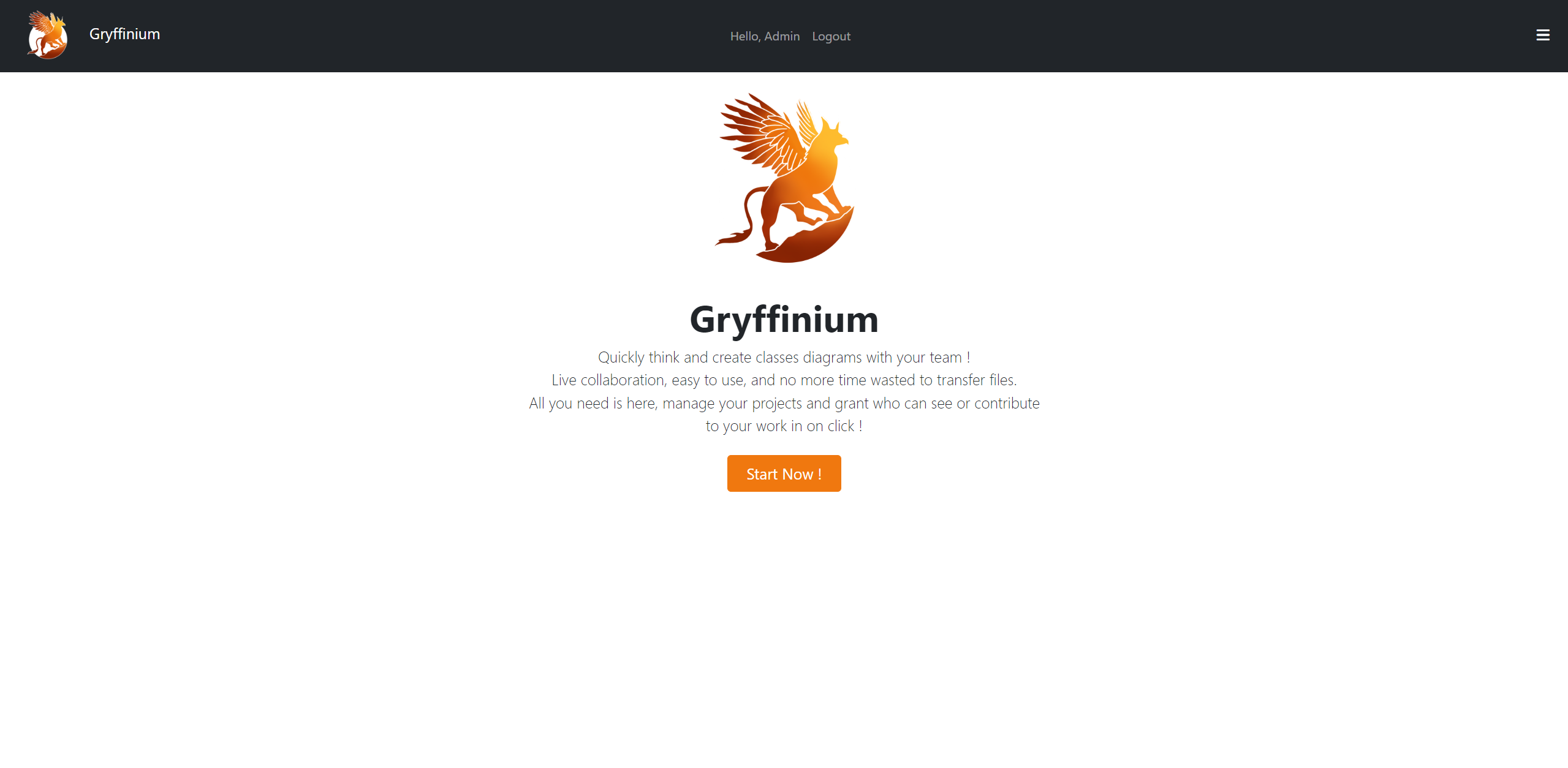


Figure 14 : Page d'accueil pour utilisateur connecté

## Modal d’authentification

Une image contenant texte

Description générée automatiquement

Figure 15 : Modal d'authentification

## Modal d’inscription

Une image contenant texte

Description générée automatiquement

Figure 16 : Modal d'inscription

## Page d’accueil avec la liste des projets

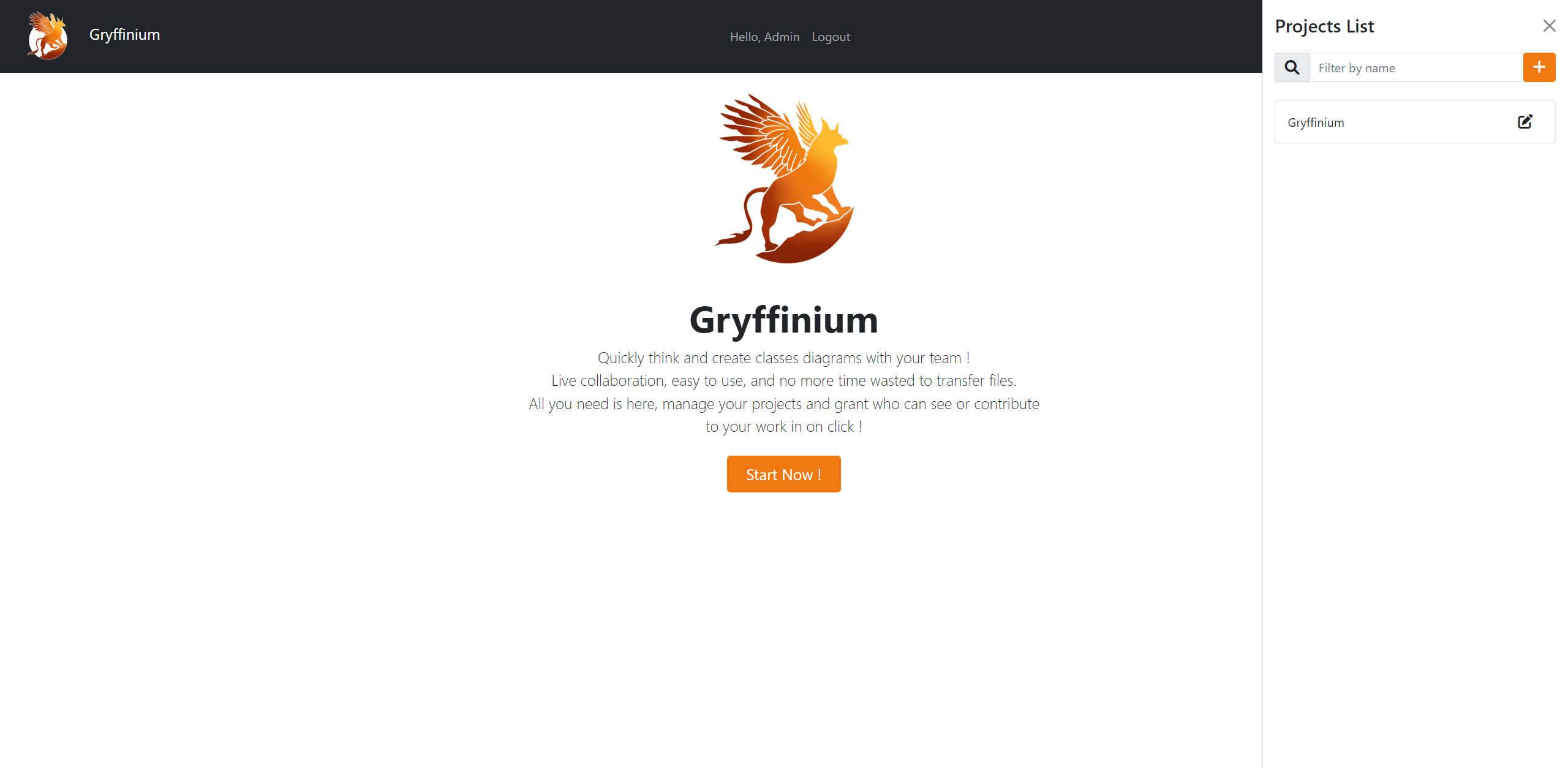


Figure 17 : Page d'accueil avec la liste des projets

## Modal de création et mise à jour de projet pour le propriétaire

Une image contenant texte

Description générée automatiquement

Figure 18 : Modal de création et mise à jour de projet pour le propriétaire

## Modal de visualisation des détails de projet

Une image contenant texte

Description générée automatiquement

Figure 19 : Modal de visualisation des détails de projet

## Modal de chat des projets

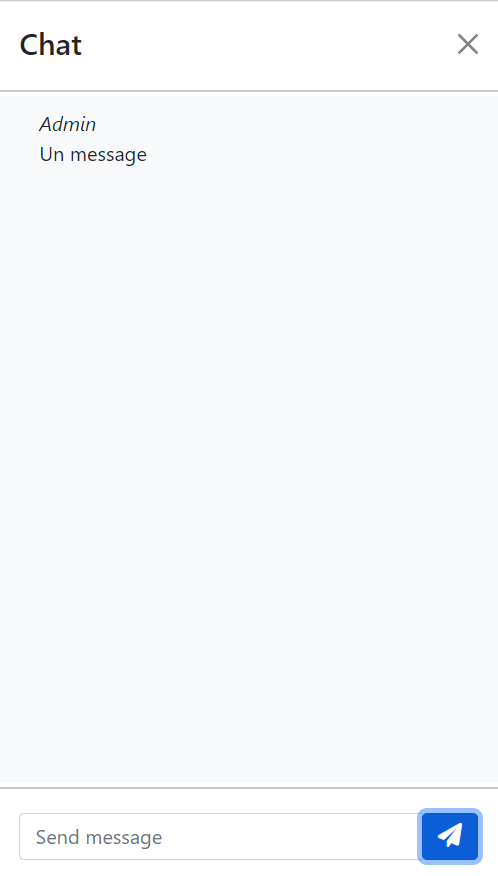


Figure 20 : Modal de chat des projets

# Conclusion

## Problèmes rencontrés

### Compréhension des Websocket et de la librairie Akka

Play ! étant plus axé sur Scala, la plupart des exemples sont dans ce langage. N’étant pas à l’aise avec le Scala, il m’était difficile de comprendre tous les concepts.

J’ai tout de même réussi à faire une première version qui permettait de stocker via des websockets, cependant cela utilisait de simples flux, et cela n’était pas suffisant pour la suite du projet.

#### Résolution

Je suis allé demander conseil à M. Jérôme Varani, qui m’a éclairé sur les concepts de bases. Suite à cela, j’ai pu mieux appréhender les exemples et la documentation, afin de comprendre les concepts plus avancés.

### Utilisation d’Hibernate

J’ai eu beaucoup de mal à mettre en place Hibernate. Nécessitant l’ajout de l’extension Hibernate pour Play!, il n’est pas directement supporté. De plus, il est très strict sur les modèles.

De plus, il ne m’a pas convaincu dû au fait que les requêtes SQL étaient en grande partie à faire à la main.

Cela pose un problème, car Ebean ne permet pas l’héritage, et donc contraint à stocker le diagramme de classes sous forme d’XML.

#### Résolution

J’ai décidé de ne pas utiliser Hibernate. Ebean permet une solution satisfaisant les besoins du projet. De plus, il peut être intéressant d’avoir le diagramme sous forme d’XML, afin que les utilisateurs puissent le télécharger s’ils le souhaitent.

## Améliorations et problèmes

### Gestion de la panne

La gestion de panne n’est pas en place. Dans l’état actuel, lorsqu’un utilisateur perd la connexion au websocket, il est obligé de récupérer tout le projet, et non uniquement les derniers messages échangés.

Le projet est également sauvegardé uniquement lorsque tous les utilisateurs se sont déconnectés. Si le serveur crash entre temps, les modifications sont perdues.

### Abstraction de la logique dans une couche de service

Afin de ne pas surcharger les contrôleurs et les modèles, il est prévu qu’une couche de service soit mise en place.

### Mise en place de DTO pour les échanges client-serveur

Il est nécessaire que des DTO soient mis en place pour les échanges de données entre le client et le serveur. Notamment pour la gestion de projet, qui est actuellement des simples objets JavaScript créés à la volée, sérialisés en JSON avant d’être envoyés. Le serveur désérialise le JSON, et récupère champ par champ les données, ce qui n’est pas idéal.

### Vérification de l’email

Lors d’une création de comptes, il serait intéressant de vérifier l’email, en envoyant une confirmation à l’email mentionné. Le compte serait alors actif uniquement quand l’email est vérifié.

### Confirmation d’action irréversible

Afin de favoriser l’expérience utilisateur, il serait bien d’avoir une pop up de confirmation pour les actions irréversibles. Pour l’instant, l’alerte basique JavaScript est utilisée, cependant en créer une avec un design correspondant au reste de l’application est à implémenter.

### Connexion interdite à un websocket

Lorsqu’un client se connecte à un websocket alors qu’il n’appartient pas au projet, ou qu’il n’a pas le droit de le faire, le websocket se ferme et ne renvoie pas une erreur.

### Suppression de projets ou de collaborateurs

Si un utilisateur travaille sur un projet, et qu’il est retiré du projet ou que le projet est supprimé, une erreur au niveau des websockets se fait. Il faudrait détecter cela et déconnecter tous les utilisateurs avant de supprimer.

# Table des illustrations

[Figure 1 : Métaschéma 6](#_Toc103525612)

[Figure 2 : Diagramme de séquence de l’authentification 10](#_Toc103525613)

[Figure 3 : Diagramme de séquence Register 11](#_Toc103525614)

[Figure 4 : Diagramme de séquence de la création de projets 13](#_Toc103525615)

[Figure 5 : Diagramme de séquence de la mise à jour d'un projet 14](#_Toc103525616)

[Figure 6 : Diagramme de séquence de la suppression d'un projet 15](#_Toc103525617)

[Figure 7 : Diagramme de séquence de la récupération des projets de l'utilisateur connecté 15](#_Toc103525618)

[Figure 8 : Diagramme de séquence de la récupération d'un projet spécifique et de sa vue 16](#_Toc103525619)

[Figure 9 : Diagramme de séquence de l'ajout d'un collaborateur 17](#_Toc103525620)

[Figure 10 : Diagramme de séquence de la modification des droits d'un utilisateur 18](#_Toc103525621)

[Figure 11 : Diagramme de séquence de la suppression d'un collaborateur 19](#_Toc103525622)

[Figure 12 : Diagramme de séquence du websocket 20](#_Toc103525623)

[Figure 13 : Page d'accueil pour utilisateur non connecté 22](#_Toc103525624)

[Figure 14 : Page d'accueil pour utilisateur connecté 23](#_Toc103525625)

[Figure 15 : Modal d'authentification 24](#_Toc103525626)

[Figure 16 : Modal d'inscription 24](#_Toc103525627)

[Figure 17 : Page d'accueil avec la liste des projets 25](#_Toc103525628)

[Figure 18 : Modal de création et mise à jour de projet pour le propriétaire 26](#_Toc103525629)

[Figure 19 : Modal de visualisation des détails de projet 27](#_Toc103525630)

[Figure 20 : Modal de chat des projets 28](#_Toc103525631)

# Annexes

* Journal de travail

1. https://ebean.io/docs/mapping/ [↑](#footnote-ref-1)
2. https://www.mindrot.org/projects/jBCrypt/ [↑](#footnote-ref-2)
3. https://medium.com/@nnnsadeh/building-a-reactive-distributed-messaging-server-in-scala-and-akka-with-websockets-c70440c494e3 [↑](#footnote-ref-3)
4. https://www.playframework.com/documentation/2.8.x/JavaWebSockets [↑](#footnote-ref-4)