**Département :** TIC

**Filière :** Informatique et systèmes de communication

**Orientation :** Informatique logicielle

**Travail de Bachelor**

Gryffinium

**Non confidentiel**

**)**

|  |  |
| --- | --- |
| **Étudiant :** | Quentin Forestier |
| **Travail proposé par :** | Pier Donini  HEIG-VD  Route de Cheseaux 1  1400 Yverdon-les-Bains |
| **Enseignant responsable :** | Pier Donini |
| **Année académique :** | 2021-2022 |

Yverdon-les-Bains, le 27 juillet 2022

**Département :**TIC

**Filière :**Informatique et systèmes de communication

**Orientation :** Informatique logicielle

**Étudiant :** Quentin Forestier

**Enseignant responsable :** Pier Donini

Travail de Bachelor 2021-2022

Gryffinium

**Résumé publiable**

**Diagrammes de classes UML**

Les diagrammes de classes UML ont pour objectif de représenter graphiquement la structure d'une architecture orientée objet. Pour ce faire, des éléments structurels sont mis en place, comme les classes et les énumérations, ainsi que des liens qui les relient.

Ces diagrammes sont principalement utilisés en génie logiciel, et principalement en amont d'un projet. Il permet d'établir des bases solides, indépendantes d’un quelconque langage de programmations. Ils permettent également de pouvoir travailler en équipe, car tous les membres de l'équipe peuvent implémenter leur partie en sachant comme les autres parties vont réagir.

De plus, cela permet d'avoir une vue d'ensemble sur le projet pour directement réfléchir à la meilleure implémentation possible.

**Éditeurs existants**

Sur le marché, il existe plusieurs logiciels permettant de créer des diagrammes de classes, cependant la plupart ne sont pas uniquement spécialisés dans ce type de diagrammes. Contrairement à ces derniers, Slyum est logiciel dédié à la création de diagrammes de classes. En plus, il est développé par la Haute École d'Ingénierie et de Gestion du canton de Vaud.

Toutefois, ce logiciel n'est plus au goût du jour. Le but de ce projet est donc de créer une nouvelle version plus adaptée aux besoins actuels, par exemple, en le rendant disponible sur internet.

**Améliorations significatives**

Étant disponible depuis le web, cela signifie que les diagrammes sont stockés sur un server. Ceci les rend disponible en tout temps à condition d'avoir une connexion internet.

L'application propose une gestion de projet permettant à ses membres de collaborer en simultané sur un même diagramme.

Le changement de nom d'un des éléments du diagramme déclenchera une modification en cascade sur tous les éléments qui le référence. De cette manière, la cohérence est garantie.

|  |  |  |
| --- | --- | --- |
| Étudiant :  Quentin Forestier | Date et lieu :  Yverdon-les-Bains, le 29 juillet 2022 | Signature :  …………………………………… |
| Enseignant responsable :  Pier Donini | Date et lieu :  …………………………………………… | Signature :  …………………………………… |
| Nom de l’entreprise/institution :  Pier Donini | Date et lieu :  …………………………………………… | Signature :  …………………………………… |

**Préambule**

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d’études, en vue de l’obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d’auteur.

HEIG-VD

Le Chef du Département

Yverdon-les-Bains, le

**Authentification**

Le soussigné, Quentin Forestier, atteste par la présente avoir réalisé seul ce travail et n’avoir utilisé aucune autre source que celles expressément mentionnées.

Yverdon-les-Bains, le



Quentin Forestier

**Table des matières**

[1 Introduction 7](#_Toc109864262)

[1.1 Cahier des charges 7](#_Toc109864263)

[2 Unified Modeling Language (UML) 10](#_Toc109864264)

[2.1 Structures de données 10](#_Toc109864265)

[2.2 Lien entre les structures de données 11](#_Toc109864266)

[3 Métaschéma de diagrammes de classes 13](#_Toc109864267)

[3.1 Schéma complet 13](#_Toc109864268)

[3.2 Entités 14](#_Toc109864269)

[3.3 Liens 15](#_Toc109864270)

[3.4 Interactions liens – entités 16](#_Toc109864271)

[3.5 L’objet ClassDiagram 17](#_Toc109864272)

[4 Architecture de l’application 18](#_Toc109864273)

[4.1 Description globale 18](#_Toc109864274)

[4.2 Stockage de données 19](#_Toc109864275)

[5 Librairies graphiques 22](#_Toc109864276)

[5.1 Design de l’interface global 22](#_Toc109864277)

[5.2 Design de diagramme de classes 23](#_Toc109864278)

[6 Fonctionnement de l’application 23](#_Toc109864279)

[6.1 Login / Register 23](#_Toc109864280)

[6.2 Gestion de projets 25](#_Toc109864281)

[6.3 Collaboration sur les projets 34](#_Toc109864282)

[7 Interface utilisateur 36](#_Toc109864283)

[7.1 Page d’accueil pour utilisateur non connecté 36](#_Toc109864284)

[7.2 Page d’accueil pour utilisateur connecté 37](#_Toc109864285)

[7.3 Modal d’authentification 38](#_Toc109864286)

[7.4 Modal d’inscription 38](#_Toc109864287)

[7.5 Page d’accueil avec la liste des projets 39](#_Toc109864288)

[7.6 Modal de création et mise à jour de projet pour le propriétaire 40](#_Toc109864289)

[7.7 Modal de visualisation des détails de projet 41](#_Toc109864290)

[7.8 Modal de chat des projets 42](#_Toc109864291)

[7.9  Éditeur de diagrammes 43](#_Toc109864292)

[8 Conclusion 45](#_Toc109864293)

[8.1 Problèmes rencontrés 45](#_Toc109864294)

[8.2 Améliorations et problèmes 46](#_Toc109864295)

[9 Table des illustrations 47](#_Toc109864296)

[10 Annexes 48](#_Toc109864297)

# Introduction

Ce travail de Bachelor vise à concevoir une application web permettant l’édition de diagrammes de classes UML. Ces derniers permettent de représenter graphiquement la structure d’une application. Ces diagrammes sont utilisés en génie logiciel lors de la conception d’applications orientées objet. De plus, les diagrammes UML permettent de s’abstraire d’un quelconque langage de programmation.

Le but étant d’améliorer les fonctionnalités de Slyum, un éditeur de diagramme de classes développées à la HEIG-VD. L’éditeur permettra l’édition facilitée de diagrammes, une disponibilité web afin de garantir sa simplicité d’accès, d’empêcher les erreurs de conceptions et de simplifier la collaboration et l’accès aux diagrammes pour les membres d’une équipe.

## 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

Voici une liste non exhaustive des solutions existantes :

* Slyum
* StarUML
* Umletino
* Visual Paradigm

### 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

#### Fonctionnalités supplémentaires 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

#### Fonctionnalités 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 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

#### 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 |

# Unified Modeling Language (UML)

L’UML permet l’édition de différents types de diagrammes. Ce projet se concentre uniquement sur les diagrammes de classes, qui permettent la représentation graphique d’une application orientée objet.

## Structures de données

Les structures de données comprennent les classes, les interfaces, les énumérations ainsi que tout ce qui compose ces dernières, c’est-à-dire les attributs, les opérations, les valeurs et les paramètres. Chacun de ces éléments à une représentation bien particulière que je vais décrire dans les prochaines parties-

### Les classes

Une image contenant table

Description générée automatiquementLes classes sont représentées sous forme de rectangle, séparé en 3 parties. Ces 3 parties correspondent respectivement au nom de la classe, ses attributs, et enfin ses opérations. Les opérations sont constituées des constructeurs et des méthodes de la classe.

Figure 1 Représentation d'une classe

### Les énumérations

Une image contenant texte

Description générée automatiquementLes énumérations sont représentées sous forme de rectangle, séparé en 4 parties. On retrouve son nom, ses valeurs, ses attributs et ses opérations. Tout comme pour les classes, les opérations regroupent les constructeurs et les méthodes.

Figure 2 Représentation d'une énumération

### Les interfaces

Une image contenant texte

Description générée automatiquementLes interfaces sont représentées sous forme de rectangle, séparé en 3 parties. Tout comme pour les classes, on retrouve le nom, les attributs et les opérations. À noter que les opérations sont uniquement des méthodes, étant donné qu’une interface ne peut pas avoir de constructeur.

Figure 3 Représentation d'une interface

### Les attributs

Une image contenant table

Description générée automatiquementLes attributs sont représentés sous forme de texte ayant une grammaire particulière. Tout d’abord on peut y voir un symbole qui représente sa visibilité.

* + correspond à public
* - correspond à private

Figure 4 Représentation d'attributs

* # correspond à protected
* ~ correspond à package

Vient ensuite le nom de l’attribut et de son type, séparé par « : ». Si un attribut est constant, la notation {const} se trouve à la fin de sa signature. Si l’attribut est static, celui-ci est alors souligné.

### Une image contenant texte Description générée automatiquementLes opérations

Les constructeurs et méthodes sont représentés sous forme de texte ayant également une grammaire particulière.

Tout comme les classes, on retrouve d’abord le symbole de visibilité ("+","-","#","~"). Puis vient le nom, auquel on ajoute des parenthèses ou l’on décrit les paramètres. Les paramètres ont un nom et un type, également séparé par « : ».

Vient ensuite le type de retour de la fonction. Ce type de retour n’est pas présent pour les constructeurs.

Figure 5 Représentation des opérations

## Lien entre les structures de données

Les liens représentent comment les structures de données interagissent entre elles. Chaque type de lien a une signification particulière.

### Héritage

L’héritage est caractérisé par une flèche blanche. La flèche ayant le trait continu représente la généralisation. Un traitillé représente la réalisation.

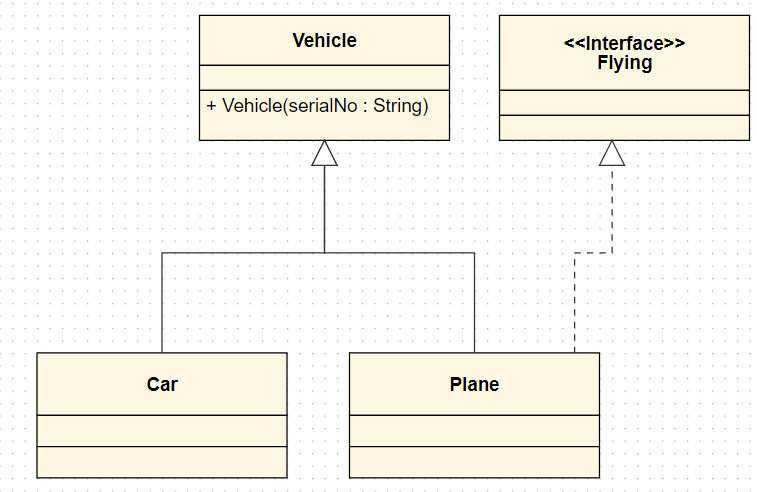


Figure 6 Représentation des héritages

### Dépendance

La dépendance est caractérisée par une flèche ouverte et un traitillé. Elle peut avoir un label qui permet de spécifier l’utilité de la dépendance.

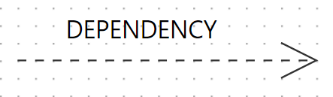


Figure 7 Représentation des dépendances

### Lien de classe interne

Le lien représentant une classe interne est caractérisé par un rond ayant une croix à l’intérieur. Le côté du rond correspond à la classe parente.

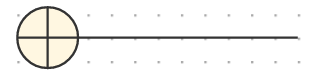


Figure 8 Représentation du lien de classe interne

### Les associations

Les associations regroupent les associations binaires, les agrégations et les compositions.

Chacun de ses liens est composé d’un label décrivant l’utilité du lien, un label par extrémité ainsi qu’un label par extrémité exprimant la multiplicité. Une flèche marque si le lien est dirigé ou non

#### Association

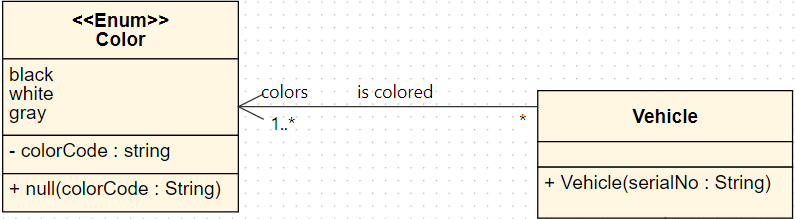


Figure 9 Représentation d'une association

#### Agrégation

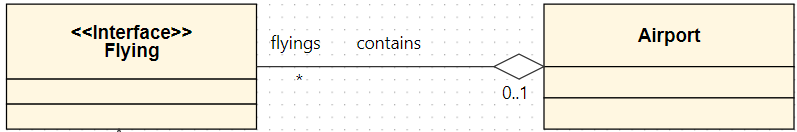


Figure 10 Représentation d'une agrégation

#### Composition

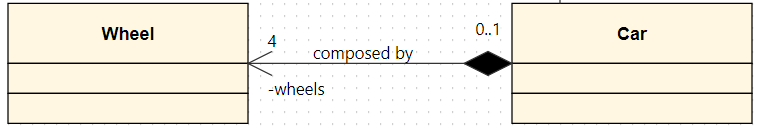


Figure 11 Représentation d'une composition

#### Les associations multiples

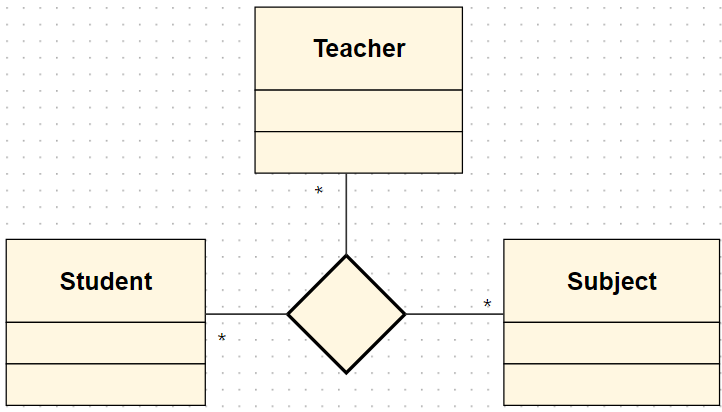


Figure 12 Représentation d'une association multiple

#### Les classes d’associations

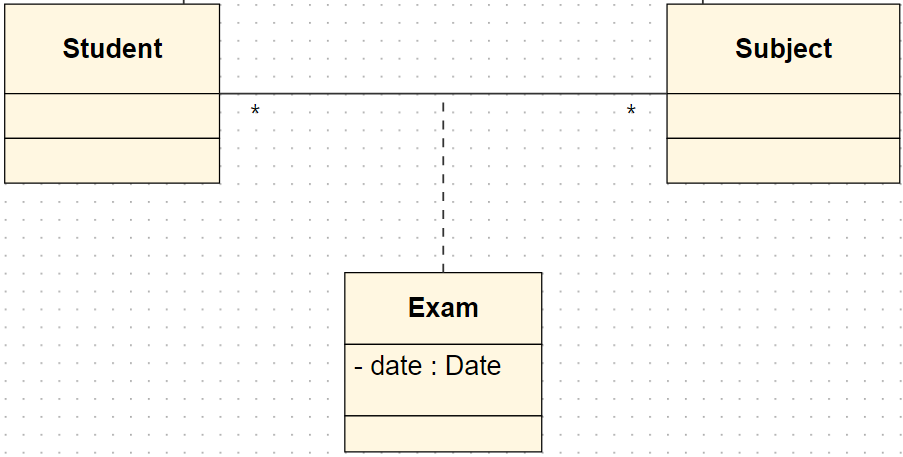


Figure 13 Représentation d'une classe d'association

# Métaschéma de diagrammes de classes

## Schéma complet

Le schéma complet est disponible en annexe.

## Entités

Figure 14 Diagramme de classes des entités

### Description

Toutes les entités sont des types. De ce fait, les variables et opérations peuvent leur faire référence pour leur type et type de retour.

*PrimitiveType* représente les types primitifs, ils sont donc exhaustifs. Cette classe fonctionne donc comme énumération. Le constructeur est privé afin de ne pas pouvoir créer d’autre type primitif.

*SimpleType* permet d’ajouter un type, qui est simplement composé d’un nom, ce qui permet donc d’avoir un type qui n’est pas représenté par une entité. Cela est utile si l’on utilise des classes appartenant au langage.

L’interface *Implementor* sert uniquement pour le lien de réalisation. Elle permet de spécifier quelles entités peuvent implémenter une interface. De ce fait, il est possible de vérifier si une entité peut implémenter une interface.

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

## Liens

Figure 15 Diagramme de classes des liens

### Description

La classe *Role* permet de décrire la multiplicité ainsi que le nom du rôle.

Afin de faciliter la modification des multi associations, j’ai décidé de créer des associations unaires, qui fonctionnent comme des associations binaires, mais en ayant un unique rôle de destination.

Tous les textes affichables sont représentés par des labels. De ce fait, il est alors possible d’avoir également leur position et donc de les afficher aux endroits sur le diagramme.

## Interactions liens – entités

Figure 16 Diagramme de classes des interactions

### Description

Chaque lien a son propre type d’entité source et destination. De cette manière, on peut vérifier que le schéma réalisé par l’utilisateur est conforme aux normes.

## L’objet ClassDiagram

Figure 17 Diagramme de classes de l'éléments racines

### Description

Cet objet est l’entité source. C’est elle qui est la racine de la sérialisation. Elle contient tous les entités et liens du diagramme.

*ExistingTypes* permet de référencer tous les types existants dans le diagramme, afin de ne pas avoir deux types portant le même nom. De plus, cela permet d’obtenir la référence vers le bon type lors de la création d’un attribut ou paramètre, et donc de pouvoir mettre à jour ce dernier automatiquement si le nom du type change.

# Architecture de l’application

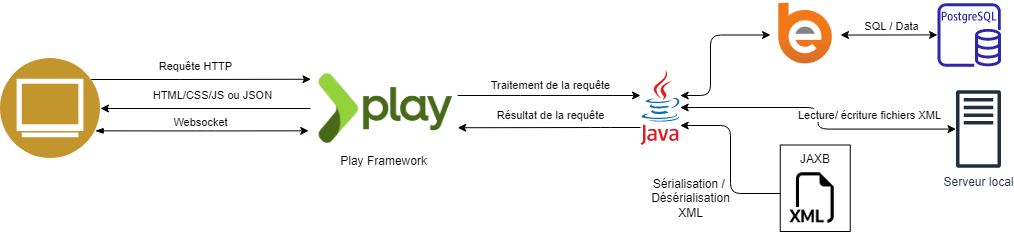


Figure 18 Architecture de l'application

## Description globale

L’application est multi tiers. On y retrouve donc une couche présentation, une couche logique et une couche donnée.

La couche présentation comprend des pages HTML/CSS/JS créées à partir de Twirl, le moteur de Template par défaut de Play. Ces pages peuvent également effectuer des requêtes http en asynchrone, et recevoir du JSON comme réponse.

Toute la logique de l’application est faite en Java. Les accès à la base de données sont faits via un ORM, Ebean, qui permet d’abstraire les requêtes SQL en fonction Java. La sérialisation / désérialisation XML est faite grâce à JAXB, une librairie spécialisée dans ce domaine.

Les données des projets sont stockées sur une base de données PostgreSQL à l’exception des diagrammes, qui sont stockés en fichier .xml sur le serveur.

### Websocket

Les websockets méritent une attention particulière car ils sont le cœur de l’interactivité de l’application. Chaque action que l’utilisateur fait sur l’éditeur de diagrammes est envoyée sous forme de commande via un websocket.

Chaque commande sera parsée, vérifiera que les droits d’exécution sont bons, exécutera la commande et retournera la réponse. Les cibles de la commande peuvent diverger en fonction de la nature de la commande.

Par exemple, la commande *Select* ne renverra la réponse qu’à l’utilisateur ayant envoyé la commande, alors que la commande *Update* renverra la réponse à tous les utilisateurs connectés.

Il faut également noter que les Websocket se ferment si aucune information n’est échangée pendant un certain temps. Une commande de Ping est donc à mettre en place

## Stockage 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

#### Utilité de l’ORM

Les ORM accélèrent la vitesse de développement en éliminant le besoin d’écrire des requêtes SQL. Il ne soucie également pas du SGBD, ce qui permet de pouvoir le changer plus simplement.

Voici un exemple qui illustre bien la rapidité à laquelle les interactions avec la base de données peuvent être mise en place :

Une image contenant texte

Description générée automatiquement

Figure 19 Exemple d'utilisation d'un ORM

Ici, il est question de retourner un objet *Project* ayant l’id passé en paramètre. Une requête SQL « classique » nécessiterait de récupérer la connexion à la base de données, créer la requête, l’exécuter pour enfin obtenir les données, qu’il faudrait alors utiliser pour créer un objet *Projet*. L’ORM nous permet donc de passer par des fonctions représentant le SQL, et faire des requêtes rapidement.

#### 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 gère l’héritage à un seul niveau, ce qui n’est pas suffisant pour le projet. Il est dès lors impossible de stocker le métaschéma, fait précédemment, de manière relationnelle. Il faudrait alors stocker le diagramme sous forme d’XML.

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. De plus, il n’est pas forcément pertinent de stocker les diagrammes directement dans la base, car ceci l’alourdirait sans réel bénéfice.

#### Mise en place

Les *entities* seront les classes qui seront représenté comme des tables dans la base de données. Il est nécessaire de les annoter avec certain mot clé afin d’exprimer à l’ORM ce que l’on souhaite. Elles doivent également hériter de la classe *Model* mise à disposition par Ebean.

Voici un exemple de mise en place d’une entité :

Une image contenant texte

Description générée automatiquement

Figure 20 Mise en place d'une entité Ebean

On retrouve donc l’annotation *@Entity* qui permet de signifier à Ebean que cette classe est à créer dans la base de données. L’annotation *@Id* décrit le champ comme identifiant de la table, *@NotNull* oblige que le champ soit rempli dans la base de données, *@Transient* permet d’ignorer le champ, car dans le cadre de ce projet, le diagramme n’est pas sauvegardé dans la base de données. Enfin, *@OneToMany* permet de gérer les jointures, et donc dans le cas ci-dessus de récupérer tous les utilisateurs participant au projet. [[1]](#footnote-1)

### XML

Les diagrammes sont sauvegardés sous format XML. J’ai eu la possibilité de choisir entre les sauvegarder dans la base de données, ou alors en fichier .xml. J’ai opté pour la deuxième option, car les fichiers peuvent très rapidement devenir lourds et cela alourdirait la base de données sans réel intérêt.

Cependant, il pourrait être intéressant de mettre cet XML dans la base de données, car PostgreSQL permet d’utiliser XPath, et on pourrait donc imaginer pouvoir rechercher un diagramme par une entité qui se trouve à l’intérieur.

#### Mise en place

J’ai utilisé JAXB, fourni avec Play afin de sérialiser et désérialiser les diagrammes. JAXB permet d'annoter les différents attributs à sérialiser. Il permet également la sérialisation avec héritage, en ajoutant une annotation sur la classe parente qui référence sur les classes enfants.

Tout comme pour l’ORM, il est nécessaire d’annoter les champs afin de spécifier à la librairie comment gérer la donnée.

Voici un exemple d’utilisation :

Une image contenant texte

Description générée automatiquement

Figure 21 Exemple d'annotations pour JAXB

L’annotation *@XMLAttribute* permet de créer un attribut XML avec les informations du champ souhaité.

L’annotation *@XMLElement* permet de créer un élément XML. Il est alors possible des éléments plus complexes, comme c’est le cas sur l’image ci-dessus, où une entité possède une liste d’attribut, qui sont eux aussi des objets.

L’annotation *@XmlID* permet de créer un Id XML qui sera alors référençable avec l’annotation *@XMLIDREF*.

Et encore plus important, l’annotation *@XmlSeeAlso* qui se trouve en haut des classes. Cette annotation permet de référencer des classes héritant de celle-ci. Combiner avec l’annotation *@XmlType*, également à placer en haut d’une classe, la sérialisation/désérialisation d’un objet pourra être effectué avec le bon type.

Un objet *Class*, qui hérite donc de *Entity* sera représenté tel que voici :

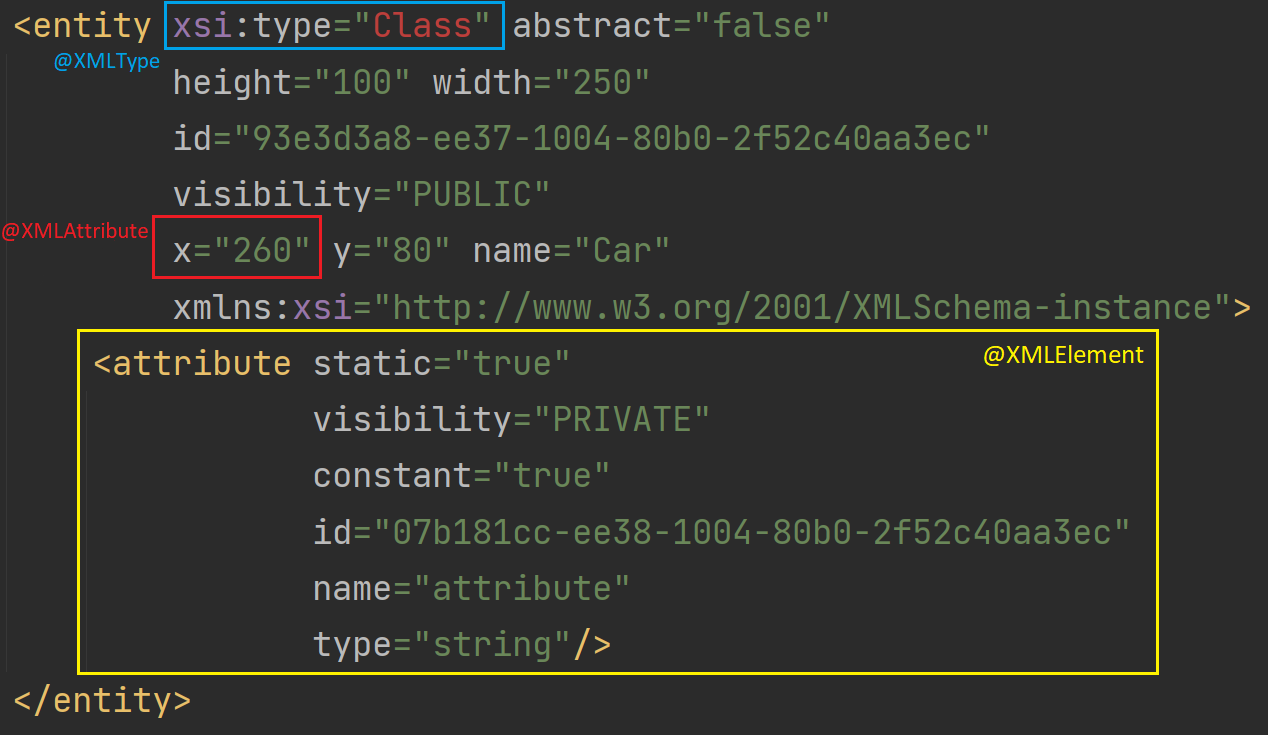


Figure 22 Exemple de résultat de sérialisation

On retrouve donc une entité, ayant pour réel type le type *Class*, ainsi que toutes les informations de la classe *Class*.

On retrouve également la liste d’attribut, qui sont eux aussi des éléments XML.

# Librairies 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

J’ai décidé d’utiliser la librairie JointJS, car lors de mes recherches, une bonne documentation était à disposition. Les premiers problèmes que j’ai eus en l’essayant trouvaient rapidement solution avec une recherche sur internet.

JointJS mettait à disposition une librairie pour les diagrammes UML, ce qui m’a permis de l’utiliser comme base, et d’ajouter et modifier les éléments manquants pour mon travail. Cette librairie proposait une implémentation qui se rapproche de l’orienté objet. Pour chaque « classe », il est possible de spécifier ses attributs et méthodes, d’ajouter des événements et un markup SVG pour l’affichage.

Cependant, l’utilisation de cette librairie m’a fait perdre un temps précieux, car la documentation était bonne en surface, mais une fois plongé dans les détails, j’ai souvent dû regarder le code source afin de comprendre au mieux, et d’envisager une solution par-dessus.

Je pense qu’utiliser la librairie à disposition qui mettait déjà à disposition les différents éléments d’UML m’a aidé au début, mais qu’elle m’a ensuite mis un frein, car je n’avais pas totalement saisi tout ce que pouvait faire JointJS. J’ai donc effectué un refactor pour les liens, car les liens proposés par la librairie ne me convenaient pas.

Un refactor des autres éléments m’auraient permis une meilleure compréhension de chaque attribut, et cela aurait pu m’aider à pouvoir éditer le diagramme directement sur le schéma.

# 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

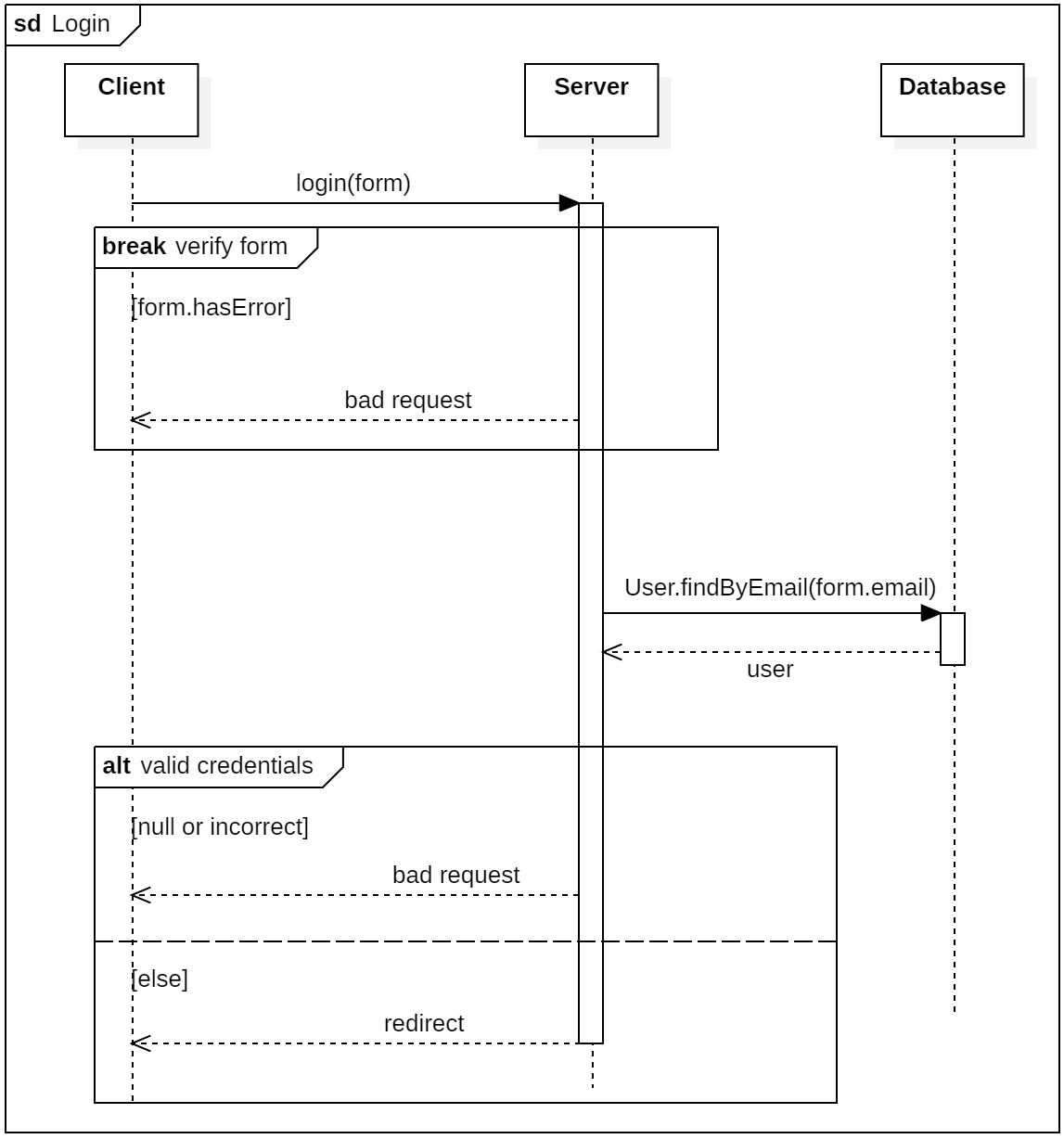


Figure 23 Diagramme de séquence de l'authentification

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

#### Register

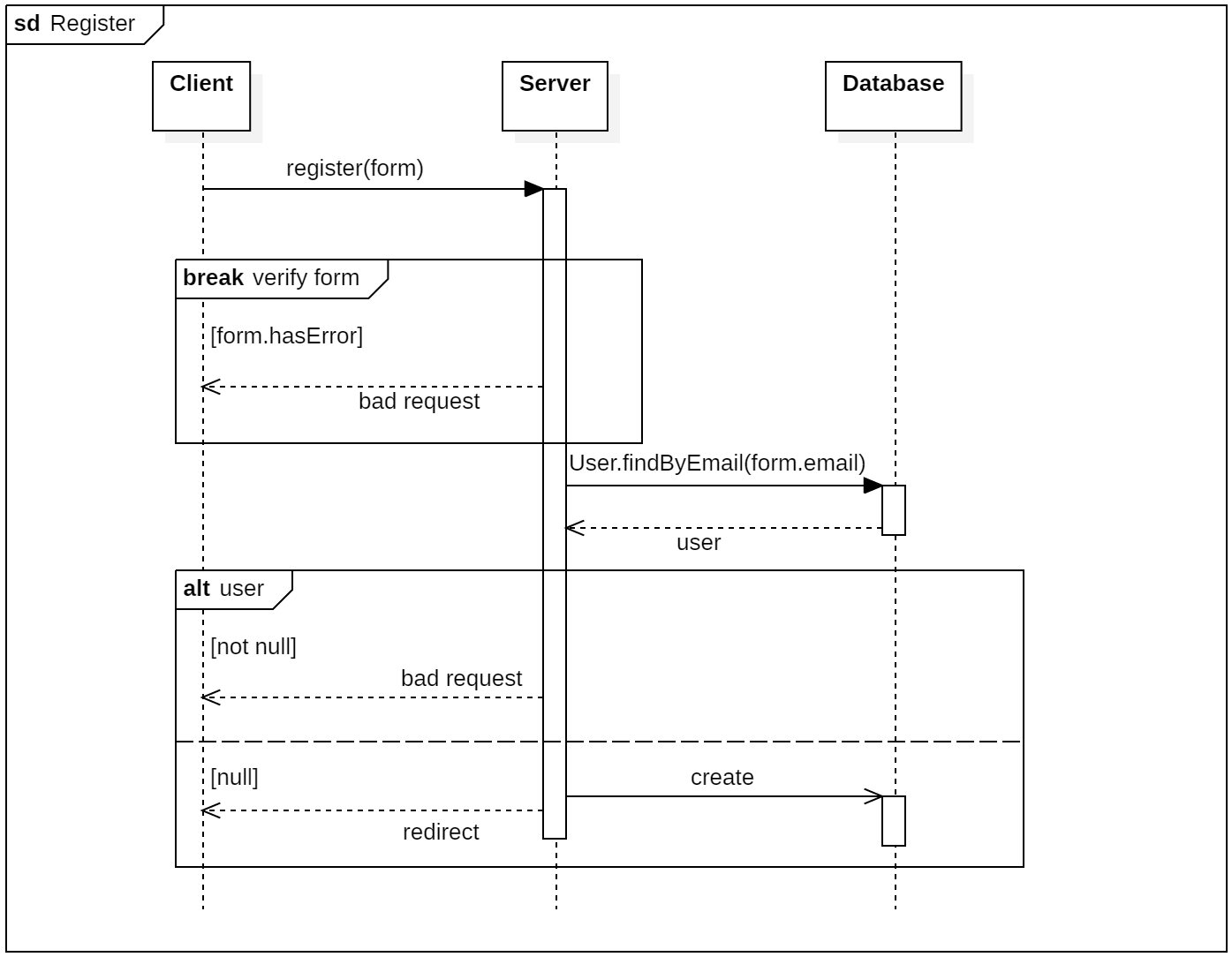


Figure 24 Diagramme de séquence de l'inscription

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.

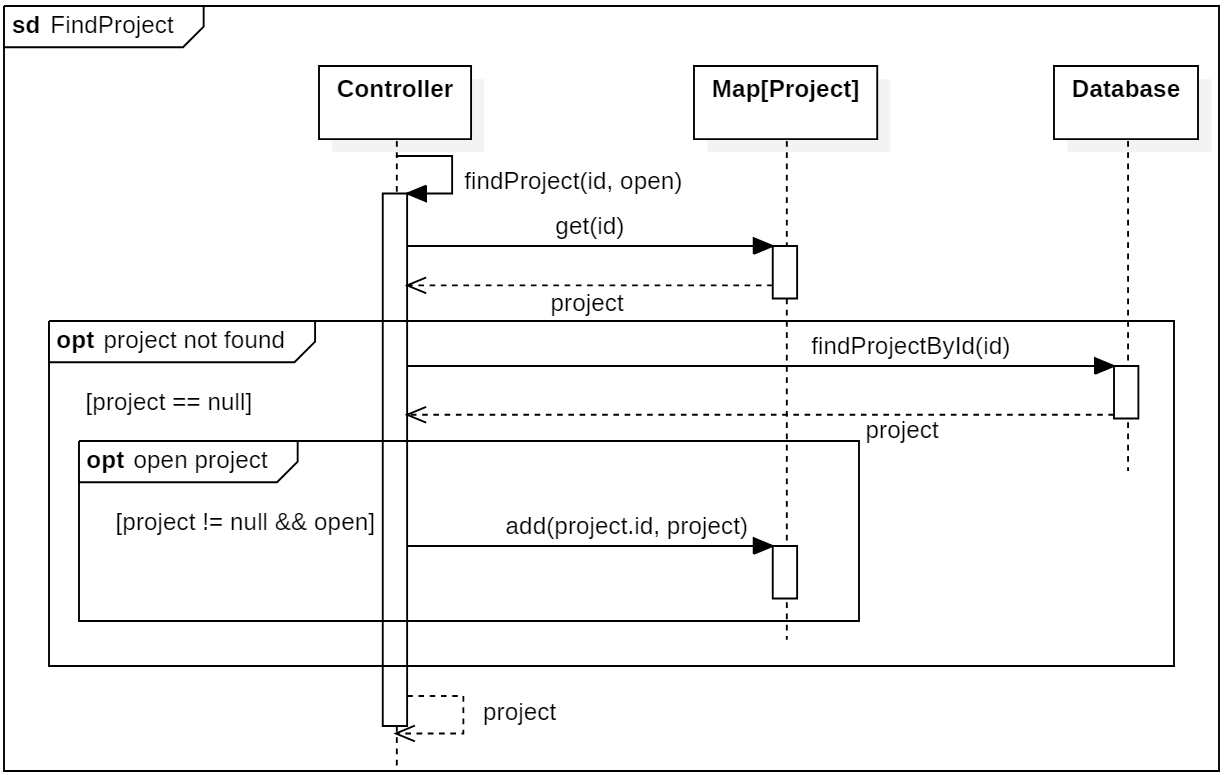


Figure 25 Diagramme de séquence de la recherche/ouverture d'un diagramme

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. Le diagramme du projet est également chargé lors de son ouverture, et donc cela est fait une unique fois.

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)

De plus, lorsqu’un projet s’ouvre, un timer est lancé afin d’effectuer des actions à un certains intervalles. Ces interactions consistent à :

* Sauvegarder le diagramme dans un fichier XML
* Vérifier si des utilisateurs sont connectés
  + Si aucun utilisateur n’est connecté, le projet se sauvegarde et se ferme.

#### Création

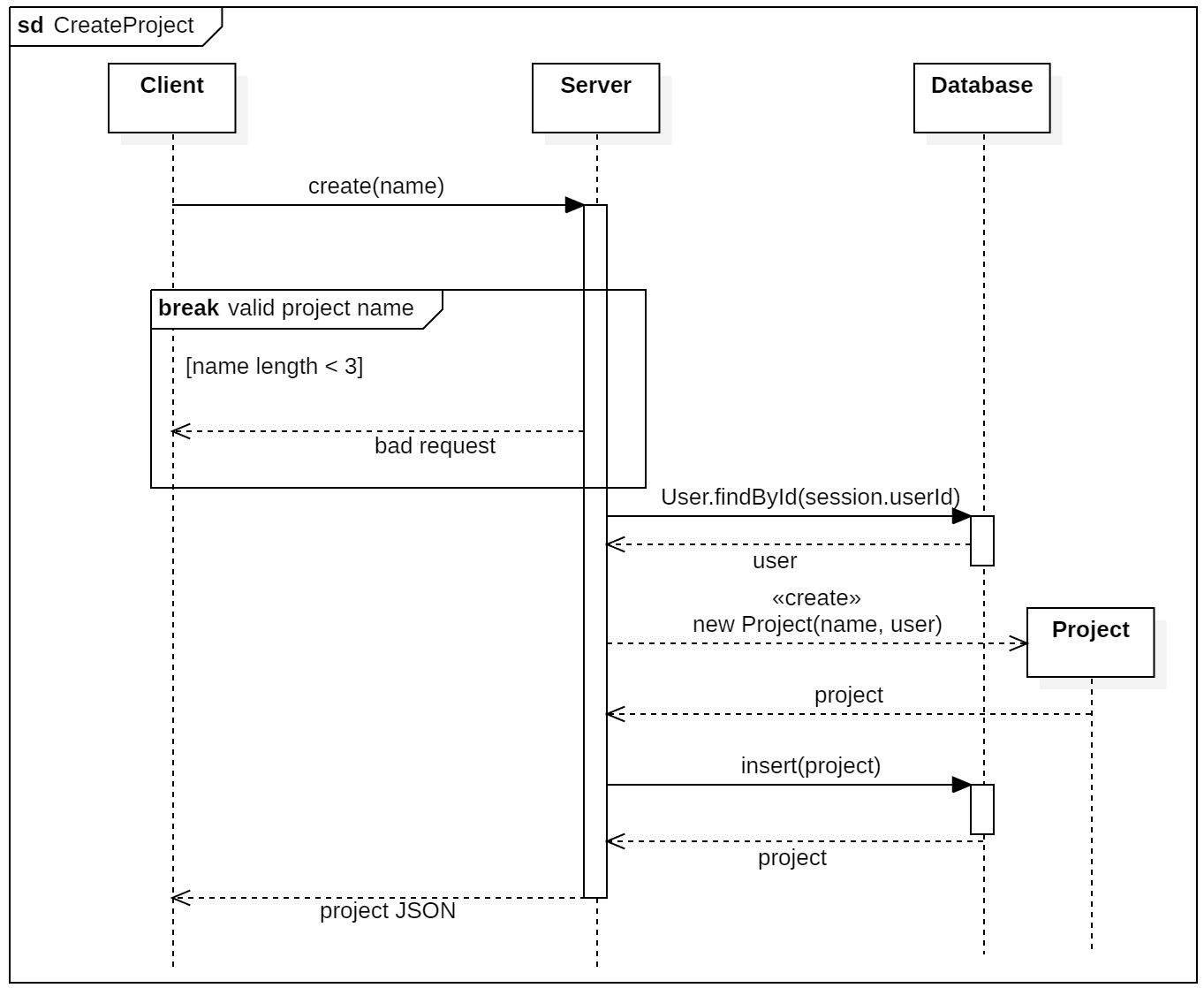


Figure 26 : Diagramme de séquence de la création de projets

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

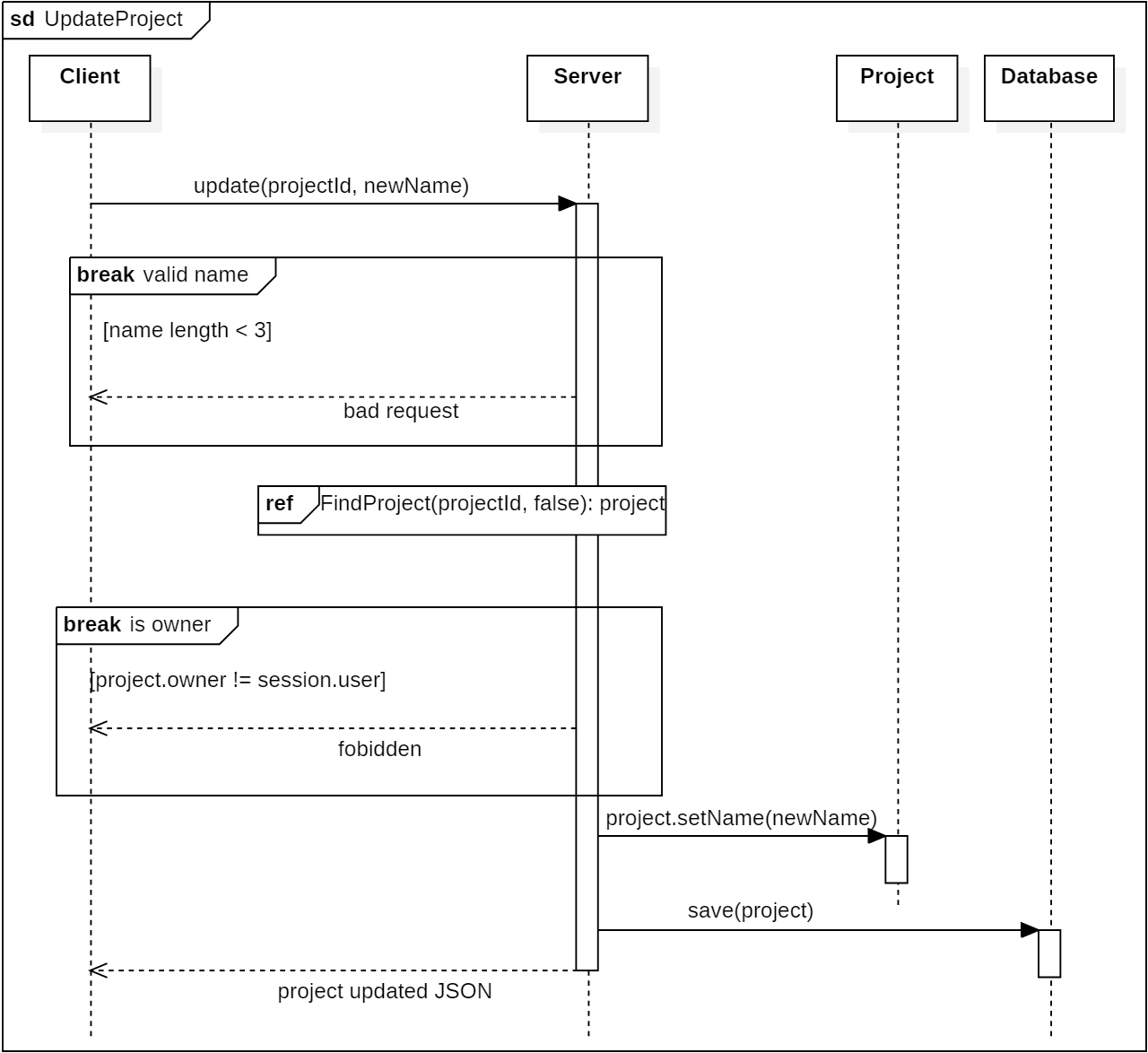


Figure 27 : Diagramme de séquence de la mise à jour d'un projet

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

#### Suppression

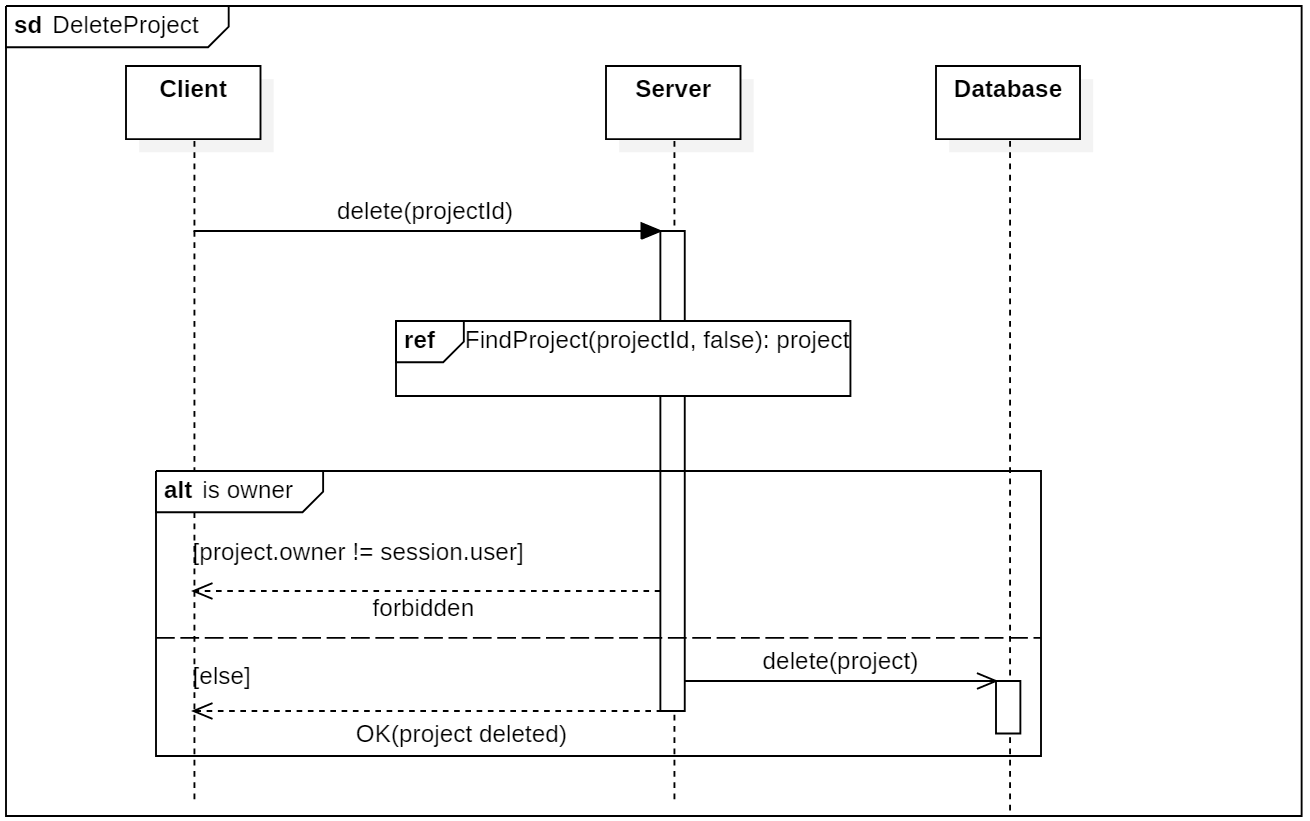


Figure 28 : Diagramme de séquence de la suppression d'un projet

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

#### Récupération des projets d’un membre

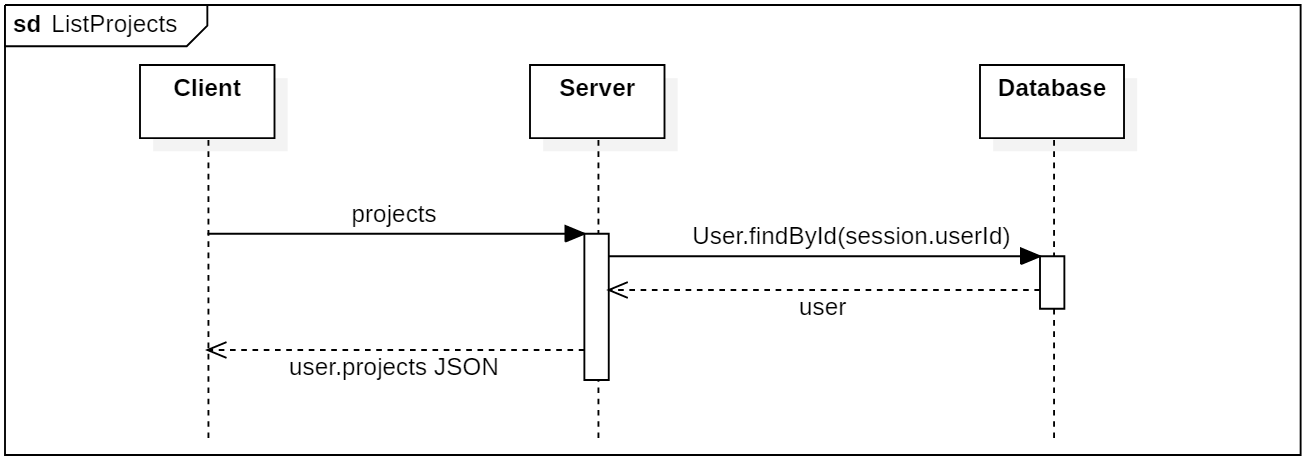


Figure 29 : Diagramme de séquence de la récupération des projets de l'utilisateur connecté

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

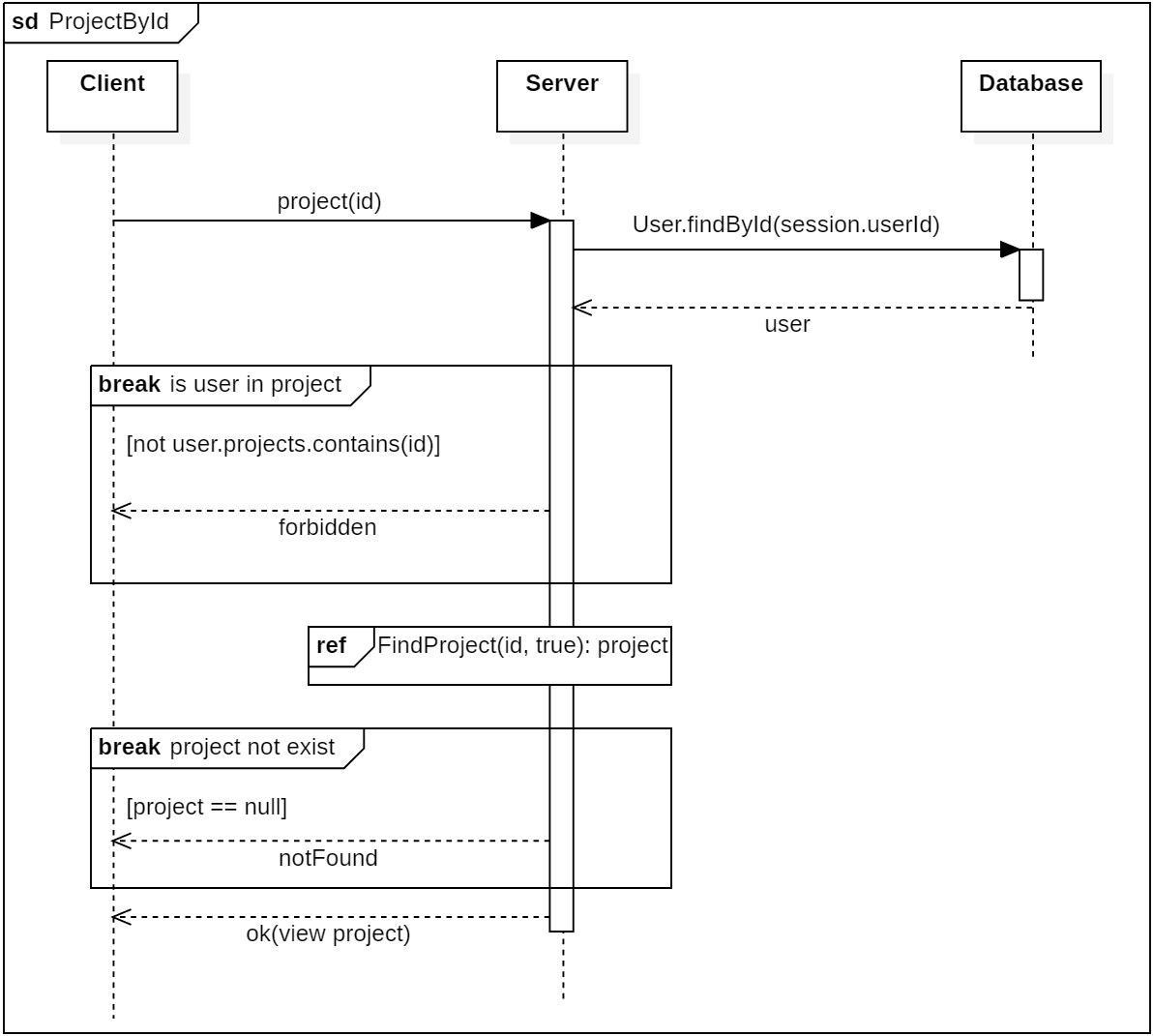


Figure 30 : Diagramme de séquence de la récupération d'un projet spécifique et de sa vue

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

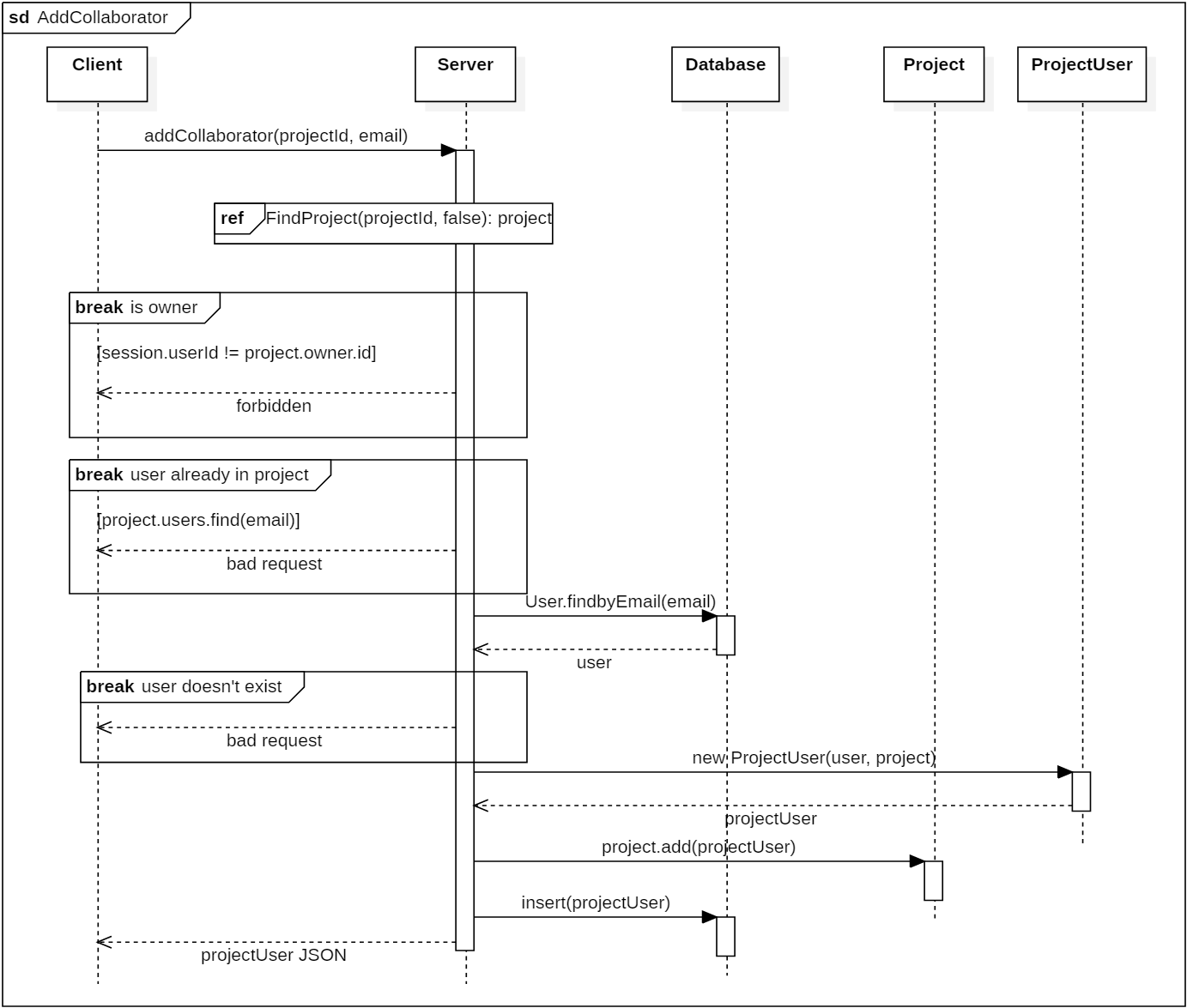


Figure 31 : Diagramme de séquence de l'ajout d'un collaborateur

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

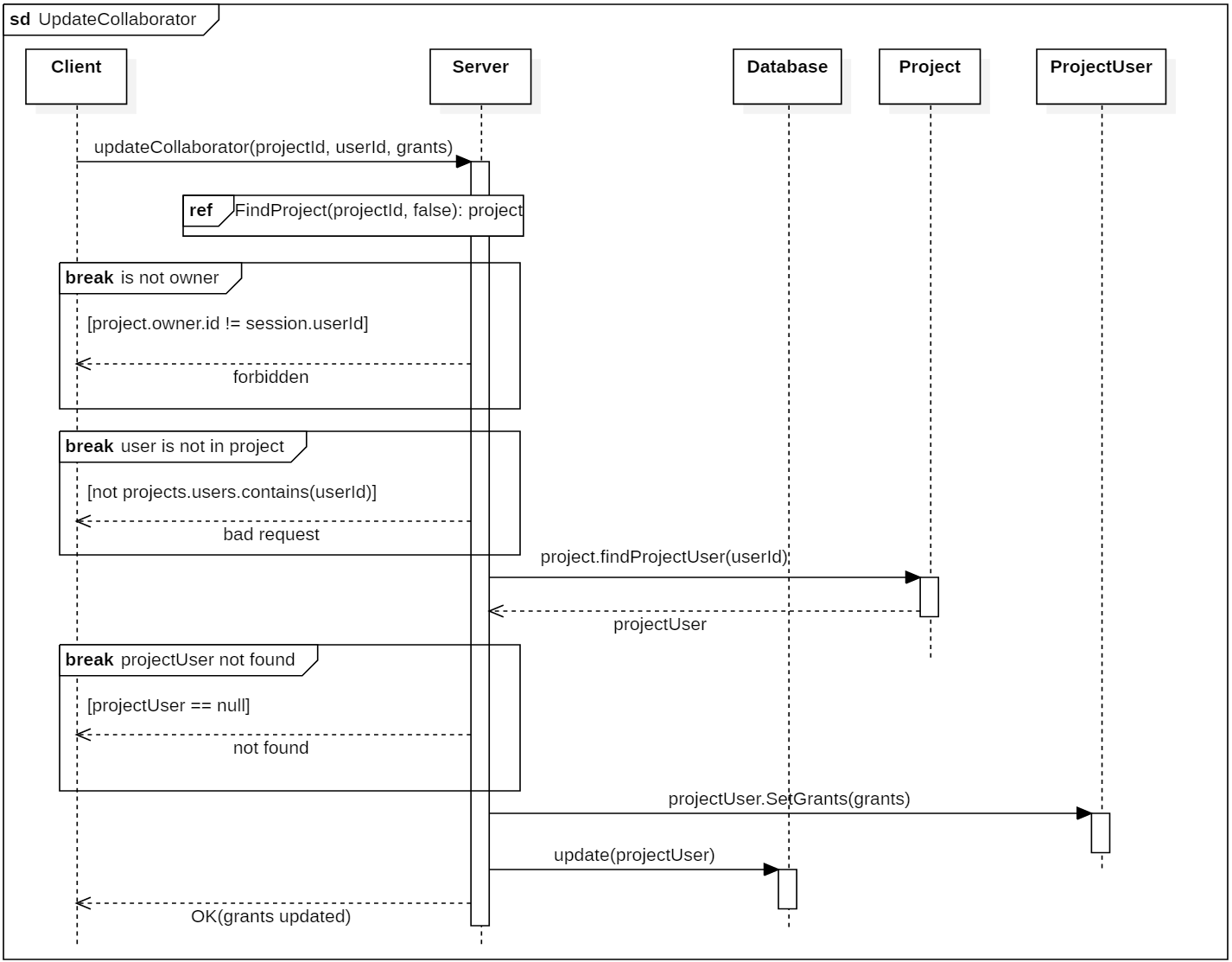


Figure 32 : Diagramme de séquence de la modification des droits d'un utilisateur

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

#### Suppression d’un collaborateur

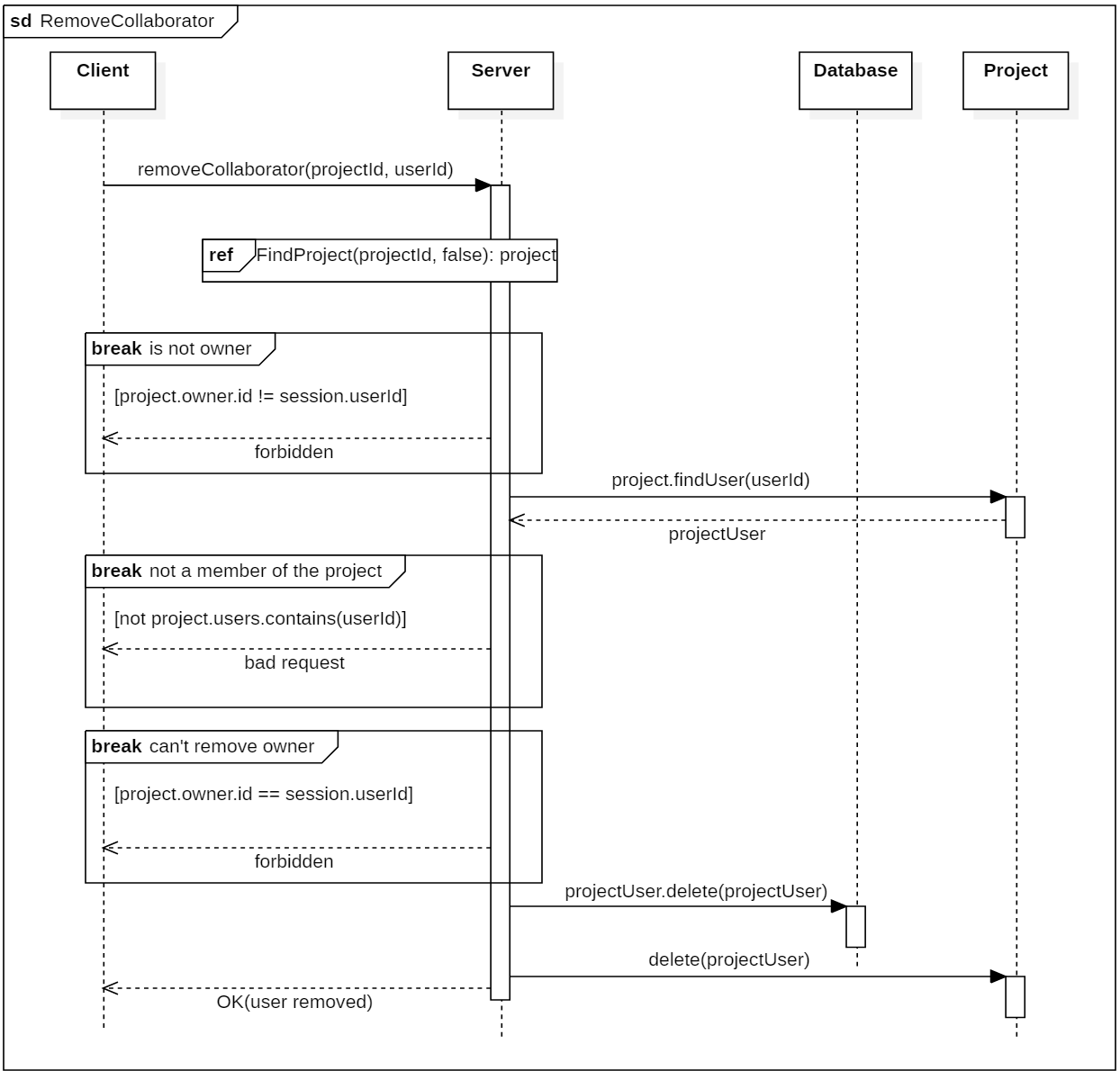


Figure 33 : Diagramme de séquence de la suppression d'un collaborateur

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

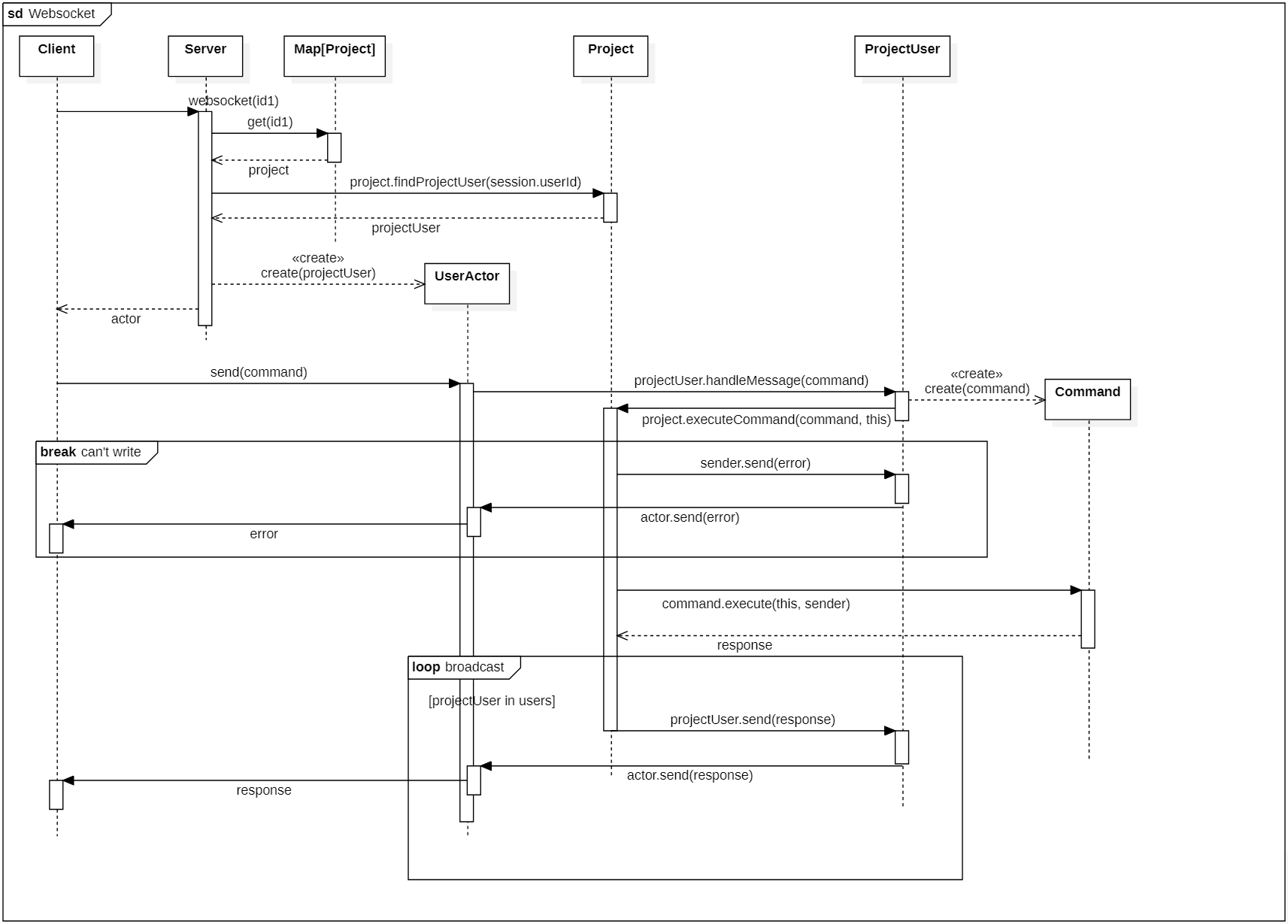


Figure 34 : Diagramme de séquence du websocket

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.

#### ChatMessage

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

Tous utilisateurs, même ceux n’ayant pas les droits de modifications peuvent utiliser cette commande.

#### Select

Cette commande sert à récupérer des informations sur le diagramme. Notamment, lors d’une connexion au websocket, il est nécessaire de récupérer le diagramme dans son entièreté.

Tous les utilisateurs peuvent utiliser cette commande.

#### Create

Cette commande crée une entité sur le serveur. Si plus d’une entité doit être créée, la commande de création de chaque entité est envoyée à chaque utilisateur connecté.

Seuls les utilisateurs ayant le droit de modification peuvent utiliser cette commande.

#### Update

Cette commande est la plus complexe. Elle modifie non seulement l’entité souhaitée, mais également les entités lui faisant référence. Par exemple, une méthode ayant un paramètre de type « Person », « Person » étant une classe du diagramme, si le nom de la classe « Person » change, la modification du paramètre doit se faire également. Cela est fait automatiquement sur le serveur, ce qui veut dire qu’on recharge la page, le nom est bien mis à jour à tous les endroits.

J’ai mis en place un mécanisme qui permet que le nom soit directement modifié, sans avoir à recharger la page. Pour cela, j’ai utilisé une liste de « Subscribers », qui lors d’une modification du nom d’un Type, on crée également une commande signifiant leur mise à jour à envoyé à tous les utilisateurs.

Seuls les utilisateurs ayant le droit de modification peuvent utiliser cette commande

#### Remove

Cette commande sert simplement à retirer une entité du diagramme.

Seuls les utilisateurs ayant le droit de modifications peuvent utiliser cette commande.

# Interface utilisateur

## Page d’accueil pour utilisateur non connecté

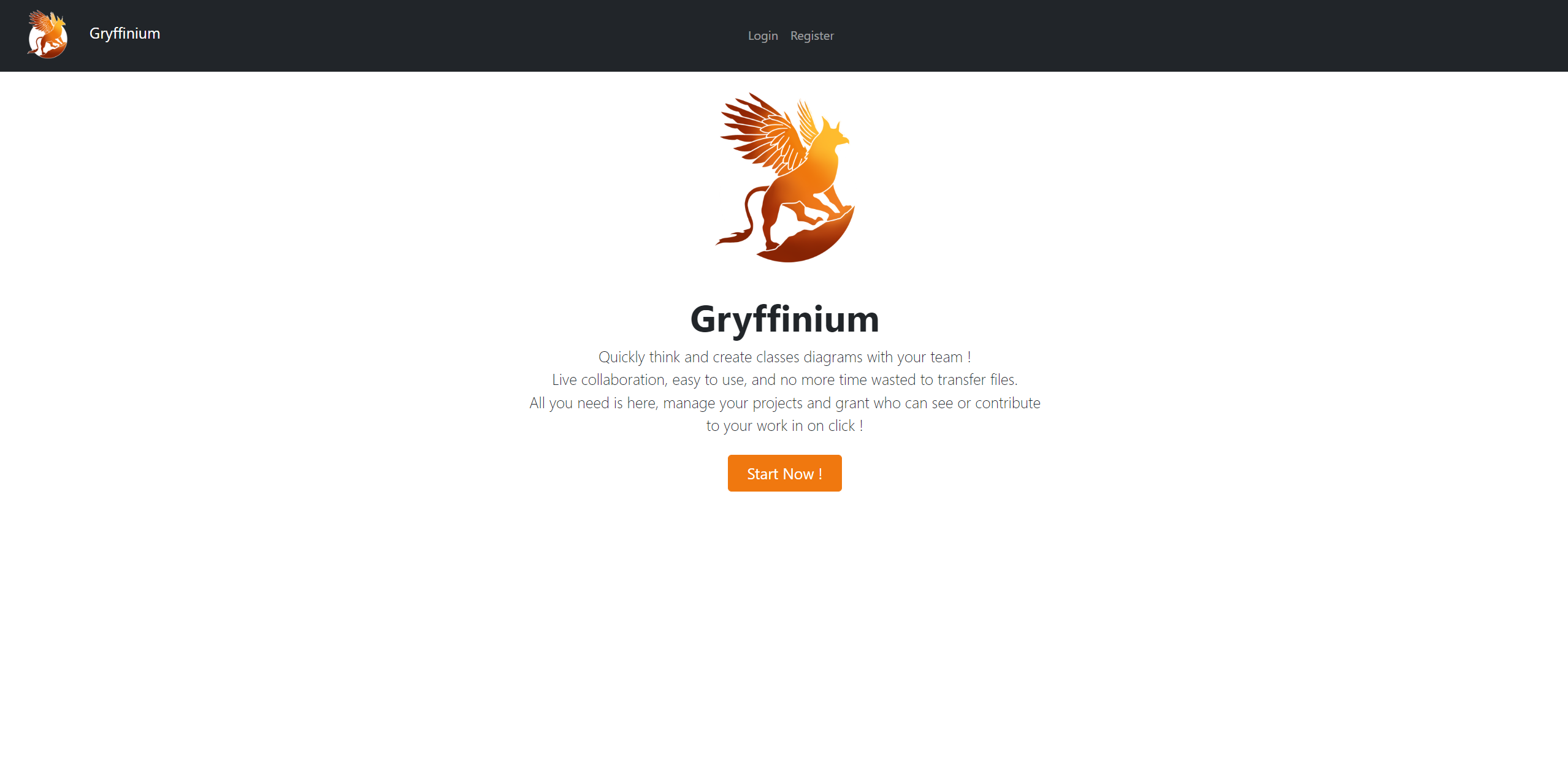


Figure 35 : Page d'accueil pour utilisateur non connecté

## Page d’accueil pour utilisateur connecté

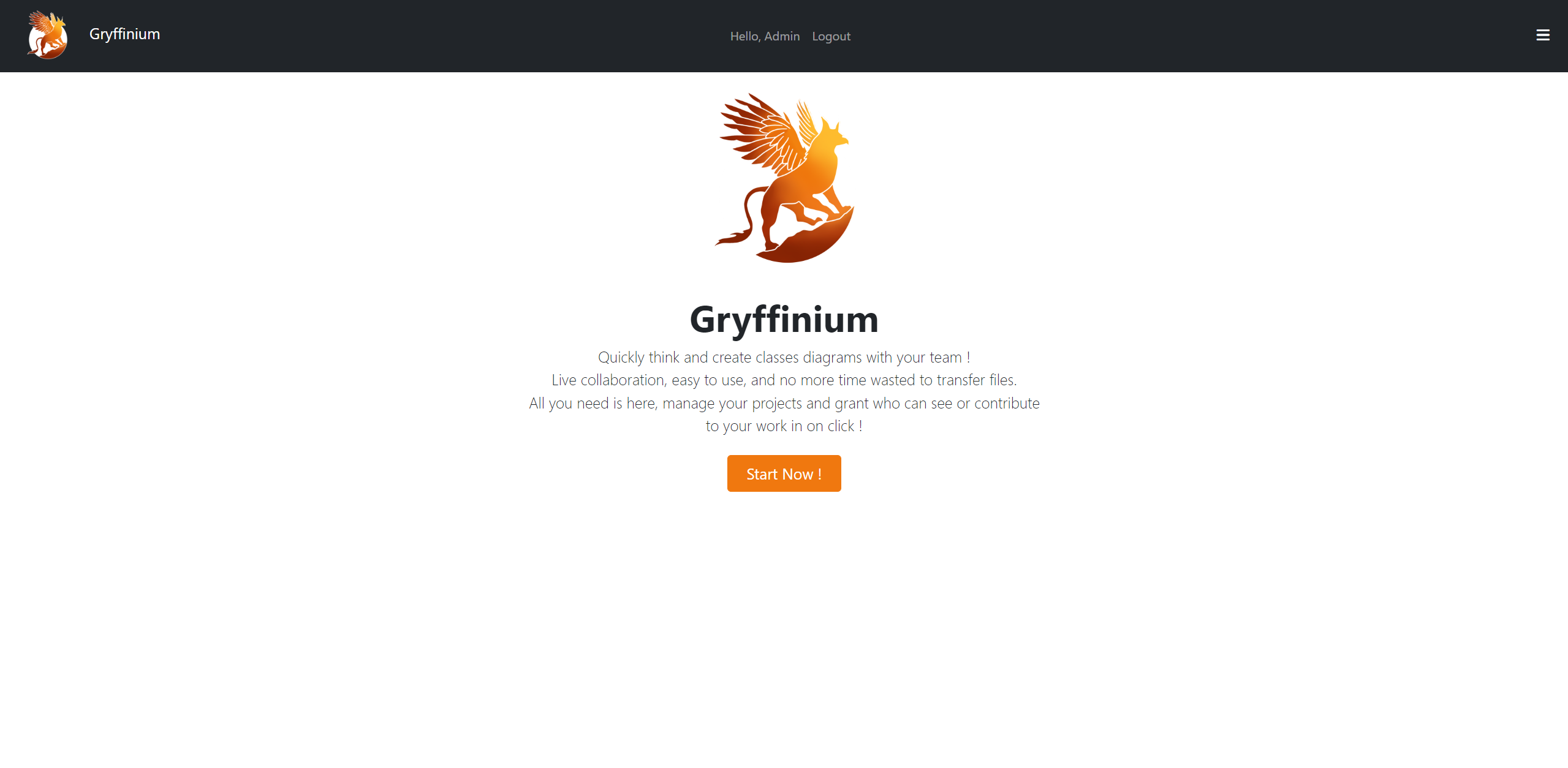


Figure 36 : Page d'accueil pour utilisateur connecté

## Modal d’authentification

Une image contenant texte

Description générée automatiquement

Figure 37 : Modal d'authentification

## Modal d’inscription

Une image contenant texte

Description générée automatiquement

Figure 38 : Modal d'inscription

## Page d’accueil avec la liste des projets

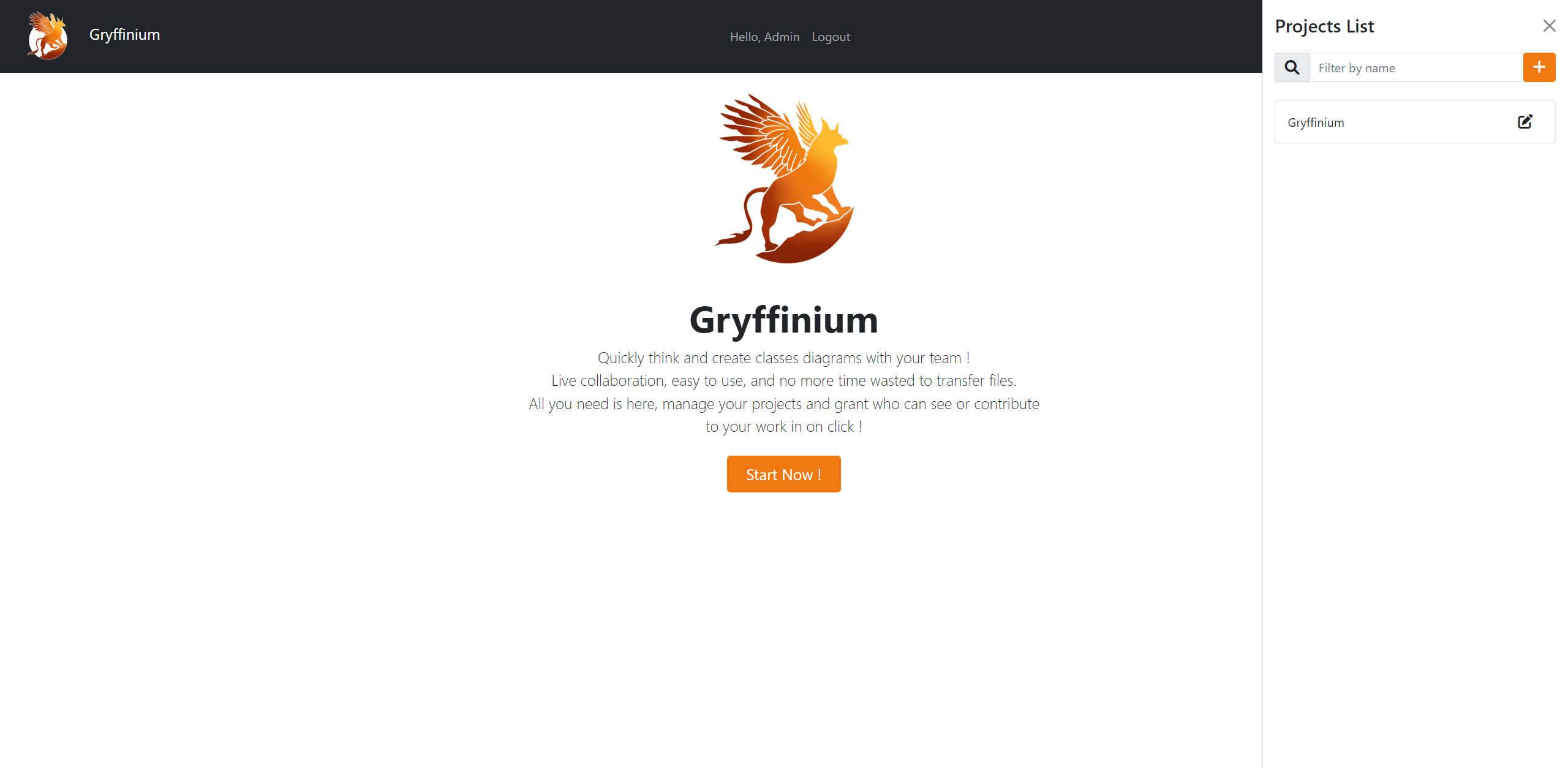


Figure 39 : Page d'accueil avec la liste des projets

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

Une image contenant texte

Description générée automatiquement

Figure 40 : 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 41 : Modal de visualisation des détails de projet

## Modal de chat des projets

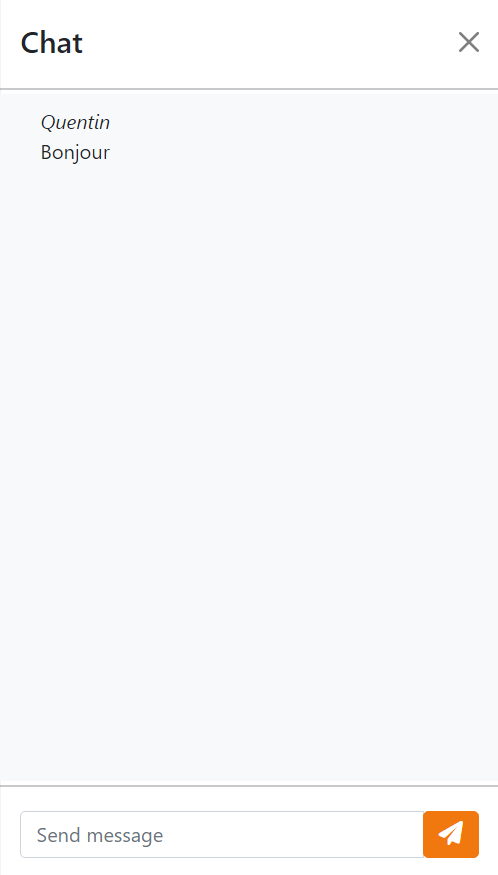


Figure 42 : Modal de chat des projets

## Une image contenant table Description générée automatiquement Éditeur de diagrammes

Figure 43 Interface de l'éditeur de diagrammes

### ToolBox

#### Entités

Une image contenant table

Description générée automatiquementOn peut retrouver un bouton pour supprimer l’entité, 2 grips sur les côtés pour le redimensionnement de l’entité, ainsi qu’un bouton qui démarre un lien ayant pour source l’entité sélectionnée.

Figure 44 ToolBox d'une entité

#### Liens

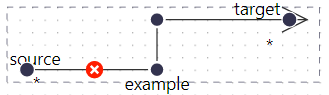
On peut retrouver le bouton permettant de supprimer le lien, mais également tous les vertex permettant de rediriger le lien

Figure 45 ToolBox d'un lien

### Toolbar

|  |  |
| --- | --- |
| Icône | Utilité |
|  | Permets l’exportation du diagramme en SVG |
|  | Active le mode « Ajout » qui permet d’ajouter une classe au prochain clic sur le diagramme. |
|  | Active le mode « Ajout » qui permet d’ajouter une énumération au prochain clic sur le diagramme. |
|  | Active le mode « Ajout » qui permet d’ajouter une interface au prochain clic sur le diagramme. |
|  | Classe d’association pas implémentée |
|  | Sélectionne la généralisation comme lien créé via la ToolBox |
|  | Sélectionne la réalisation comme lien créé via la ToolBox |
|  | Sélectionne l’association comme lien créé via la ToolBox |
|  | Sélectionne l’agrégation comme lien créé via la ToolBox |
|  | Sélectionne la composition comme lien créé via la ToolBox |
|  | Multiassociation pas implémentée |
|  | Sélectionne le lien de classe interne comme lien créé via la ToolBox |
|  | Sélectionne la dépendance comme lien créé via la ToolBox |

# 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.

### 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.

### Multiassociation

Les multiassociations sont en partie implémentées. Toute la partie sur serveur est prête, il manque uniquement la représentation sur le navigateur. En d’autres termes, il faut créer la « classe » JointJS qui s’occupera de l’affichage de la multiassociation. Il faut également implémenter la sélection d’entités qui composeront la multiassociation.

### Association Class

L’association classe est à implémenter. Elle est pensée sur les diagrammes, mais aucune implémentation n’est encore faite.

### Édition sur le diagramme

L’édition sur le diagramme n’est pas possible. Cependant, j’ai trouvé toutes les informations permettant d’y arriver. Malheureusement, la contrainte de temps m’empêche d’arriver à réaliser ce point-ci.

Pour implémenter cette fonctionnalité, il faut utiliser le foreignObject du SVG. Il faudrait donc changer l’attribut « markup » des éléments, ainsi que l’attribut « markupLabel » des liens.

En modifiant cela par un foreignObject, combiné avec un input HTML, il serait alors possible d’éditer sur le diagramme directement. J’ai trouvé une référence prouvant que cela est possible[[5]](#footnote-5).

### Annulation de la dernière action

Chaque action étant déjà une commande, il faudrait simplement créer la méthode « undo » dans chaque commande. L’architecture est donc en place, mais il manque l’implémentation de la fonction.

J’imagine la solution en stockant l’ancien statut dans un dto, et que dans le cas d’un retour en arrière, il suffirait de remodifier l’entité en fonction du dto sauvegardé.

# Table des illustrations

[Figure 1 Représentation d'une classe 10](file:///D:\HEIG-VD\TB\Gryffinium\docs\Rapport_Forestier.docx#_Toc109864217)

[Figure 2 Représentation d'une énumération 10](file:///D:\HEIG-VD\TB\Gryffinium\docs\Rapport_Forestier.docx#_Toc109864218)

[Figure 3 Représentation d'une interface 10](file:///D:\HEIG-VD\TB\Gryffinium\docs\Rapport_Forestier.docx#_Toc109864219)

[Figure 4 Représentation d'attributs 10](file:///D:\HEIG-VD\TB\Gryffinium\docs\Rapport_Forestier.docx#_Toc109864220)

[Figure 5 Représentation des opérations 11](file:///D:\HEIG-VD\TB\Gryffinium\docs\Rapport_Forestier.docx#_Toc109864221)

[Figure 6 Représentation des héritages 11](#_Toc109864222)

[Figure 7 Représentation des dépendances 11](#_Toc109864223)

[Figure 8 Représentation du lien de classe interne 11](#_Toc109864224)

[Figure 9 Représentation d'une association 12](#_Toc109864225)

[Figure 10 Représentation d'une agrégation 12](#_Toc109864226)

[Figure 11 Représentation d'une composition 12](#_Toc109864227)

[Figure 12 Représentation d'une association multiple 12](#_Toc109864228)

[Figure 13 Représentation d'une classe d'association 13](#_Toc109864229)

[Figure 14 Diagramme de classes des entités 14](file:///D:\HEIG-VD\TB\Gryffinium\docs\Rapport_Forestier.docx#_Toc109864230)

[Figure 15 Diagramme de classes des liens 15](file:///D:\HEIG-VD\TB\Gryffinium\docs\Rapport_Forestier.docx#_Toc109864231)

[Figure 16 Diagramme de classes des interactions 16](file:///D:\HEIG-VD\TB\Gryffinium\docs\Rapport_Forestier.docx#_Toc109864232)

[Figure 17 Diagramme de classes de l'éléments racines 17](file:///D:\HEIG-VD\TB\Gryffinium\docs\Rapport_Forestier.docx#_Toc109864233)

[Figure 18 Architecture de l'application 18](#_Toc109864234)

[Figure 19 Exemple d'utilisation d'un ORM 19](#_Toc109864235)

[Figure 20 Mise en place d'une entité Ebean 20](#_Toc109864236)

[Figure 21 Exemple d'annotations pour JAXB 21](#_Toc109864237)

[Figure 22 Exemple de résultat de sérialisation 22](#_Toc109864238)

[Figure 23 Diagramme de séquence de l'authentification 24](#_Toc109864239)

[Figure 24 Diagramme de séquence de l'inscription 25](#_Toc109864240)

[Figure 25 Diagramme de séquence de la recherche/ouverture d'un diagramme 26](#_Toc109864241)

[Figure 26 : Diagramme de séquence de la création de projets 27](#_Toc109864242)

[Figure 27 : Diagramme de séquence de la mise à jour d'un projet 28](#_Toc109864243)

[Figure 28 : Diagramme de séquence de la suppression d'un projet 29](#_Toc109864244)

[Figure 29 : Diagramme de séquence de la récupération des projets de l'utilisateur connecté 29](#_Toc109864245)

[Figure 30 : Diagramme de séquence de la récupération d'un projet spécifique et de sa vue 30](#_Toc109864246)

[Figure 31 : Diagramme de séquence de l'ajout d'un collaborateur 31](#_Toc109864247)

[Figure 32 : Diagramme de séquence de la modification des droits d'un utilisateur 32](#_Toc109864248)

[Figure 33 : Diagramme de séquence de la suppression d'un collaborateur 33](#_Toc109864249)

[Figure 34 : Diagramme de séquence du websocket 34](#_Toc109864250)

[Figure 35 : Page d'accueil pour utilisateur non connecté 36](#_Toc109864251)

[Figure 36 : Page d'accueil pour utilisateur connecté 37](#_Toc109864252)

[Figure 37 : Modal d'authentification 38](#_Toc109864253)

[Figure 38 : Modal d'inscription 38](#_Toc109864254)

[Figure 39 : Page d'accueil avec la liste des projets 39](#_Toc109864255)

[Figure 40 : Modal de création et mise à jour de projet pour le propriétaire 40](#_Toc109864256)

[Figure 41 : Modal de visualisation des détails de projet 41](#_Toc109864257)

[Figure 42 : Modal de chat des projets 42](#_Toc109864258)

[Figure 43 Interface de l'éditeur de diagrammes 43](file:///D:\HEIG-VD\TB\Gryffinium\docs\Rapport_Forestier.docx#_Toc109864259)

[Figure 44 ToolBox d'une entité 44](file:///D:\HEIG-VD\TB\Gryffinium\docs\Rapport_Forestier.docx#_Toc109864260)

[Figure 45 ToolBox d'un lien 44](file:///D:\HEIG-VD\TB\Gryffinium\docs\Rapport_Forestier.docx#_Toc109864261)

# 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)
5. https://jsfiddle.net/kumilingus/rpw5u8mq/ [↑](#footnote-ref-5)