

Travail de Bachelor

Gryffinium

Non confidentiel



Étudiant : Quentin Forestier

Travail proposé par : Pier Donini

Nom de l'entreprise/institution

Adresse

NPA Ville

Enseignant responsable : Pier Donini

Année académique : 2021-2022

Yverdon-les-Bains, le 21 juillet 2022

Travail de Bachelor 2021-2022

Gryffinium

Résumé publiable

Slyum est un éditeur de diagramme de classes UML développé en Java à la HEIG-VD.

Le but de ce projet est de réaliser une application web permettant l'édition collaborative et intuitive de diagrammes de classes se conformant à la norme UML, répliquant et améliorant les fonctionnalités de Slyum :

- Définition de groupes d'utilisateurs
- Définition de projets
- Définition de classes, interfaces, attributs, méthodes, associations, classes d'associations et dépendances
- Enregistrement et chargement
- Exportation sous format graphique.

Étudiant :	Date et lieu :	Signature :
Quentin Forestier
Enseignant responsable :	Date et lieu :	Signature :
Pier Donini
Nom de l'entreprise/institution :	Date et lieu :	Signature :
Pier Donini

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 21 juillet 2022

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 21 juillet 2022

Quentin Forestier

Table des matières

Table des matières

1	Introduction.....	7
1.1	Cahier des charges.....	7
1.1.1	Problématique.....	7
1.1.2	Solutions existantes.....	7
1.1.3	Objectif	7
1.1.4	Jalon.....	7
1.1.5	Fonctionnalités de l'application	8
2	Unified Modeling Language (UML).....	10
2.1	Structures de données	10
2.1.1	Les classes.....	10
2.1.2	Les énumérations	10
2.1.3	Les interfaces.....	10
2.1.4	Les attributs.....	10
2.1.5	Les opérations	11
2.2	Lien entre les structures de données	11
2.2.1	Héritage	11
2.2.2	Dépendance.....	11
2.2.3	Lien de classe interne	11
2.2.4	Les associations	12
2.2.5	Les associations multiples	12
3	Métaschéma de diagrammes de classes	13
3.1	Première version	13
3.1.1	Description de la première version du métaschéma	14
3.2	Deuxième version.....	0
3.2.1	Entités.....	0
3.2.2	Liens.....	1
3.2.3	Interactions liens – entités	2
3.2.4	L'objet ClassDiagram	3
4	Stockage de données.....	3
4.1	Base de données.....	3
4.1.1	SGBD	3
4.1.2	ORM.....	3
4.2	XML.....	4
4.2.1	Mise en place.....	4

5	Librairies graphiques	4
5.1	Design de l'interface global	4
5.2	Design de diagramme de classes.....	4
6	Fonctionnement de l'application.....	5
6.1	Login / Register.....	5
6.1.1	Sécurités et validations mises en place	5
6.1.2	Diagramme de séquences	6
6.2	Gestion de projets	7
6.2.1	Projet.....	7
6.2.2	Collaborateurs	13
6.3	Collaboration sur les projets	16
6.3.1	Websocket.....	16
6.3.2	Commandes.....	17
7	Interface utilisateur	18
7.1	Page d'accueil pour utilisateur non connecté	18
7.2	Page d'accueil pour utilisateur connecté	19
7.3	Modal d'authentification.....	20
7.4	Modal d'inscription	20
7.5	Page d'accueil avec la liste des projets	21
7.6	Modal de création et mise à jour de projet pour le propriétaire.....	22
7.7	Modal de visualisation des détails de projet.....	23
7.8	Modal de chat des projets.....	24
7.9	Éditeur de diagrammes	25
7.9.1	ToolBox.....	26
7.9.2	Toolbar.....	26
8	Conclusion	27
8.1	Problèmes rencontrés	27
8.1.1	Compréhension des Websocket et de la librairie Akka	27
8.1.2	Utilisation d'Hibernate	27
8.2	Améliorations et problèmes.....	28
8.2.1	Gestion de la panne.....	28
8.2.2	Abstraction de la logique dans une couche de service	28
8.2.3	Vérification de l'email.....	28
8.2.4	Confirmation d'action irréversible	28
8.2.5	Connexion interdite à un websocket.....	28
8.2.6	Suppression de projets ou de collaborateurs.....	28
8.2.7	Multiassociation	28
8.2.8	Association Class.....	28

8.2.9	Édition sur le diagramme	28
8.2.10	Annulation de la dernière action.....	29
9	Table des illustrations.....	29
10	Annexes	30

Bibliographie

1 Introduction

Ce travail de Bachelor a pour but de 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.

Ce projet a pour but 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.

1.1 Cahier des charges

1.1.1 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.

1.1.2 Solutions existantes

Voici une liste non exhaustive des solutions existantes :

- Slyum
- StarUML
- Umletino
- Visual Paradigm

1.1.3 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.

1.1.4 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 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! effectuée, 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.

1.1.5 Fonctionnalités de l'application

1.1.5.1 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
- Srialisation/Désrialisation des diagrammes

1.1.5.2 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

1.1.5.3 Fonctionnalités de gestion

- Inscription/Connexion
- Créeation de projets
- Créeation 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

1.1.5.4 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

1.1.5.5 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 :
 - o Les choix de conception
 - o Les problèmes rencontrés
 - o Les solutions envisagées
 - o La synthèse du résultat obtenu

1.1.5.6 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

2 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.

2.1 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-

2.1.1 Les classes

Les 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.

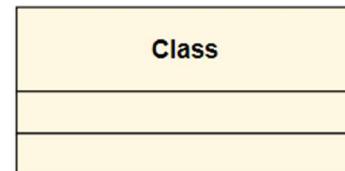


Figure 1 Représentation d'une classe

2.1.2 Les énumérations

Les 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.

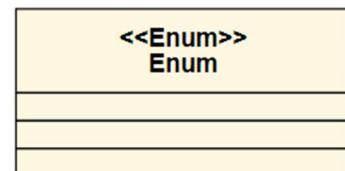


Figure 2 Représentation d'une énumération

2.1.3 Les interfaces

Les 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.

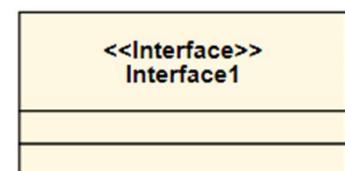


Figure 3 Représentation d'une interface

2.1.4 Les attributs

Les 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
- # correspond à protected
- ~ correspond à package

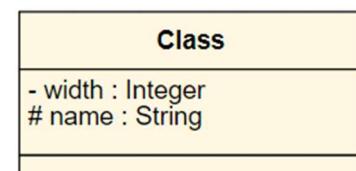


Figure 4 Représentation d'attributs

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.

2.1.5 Les 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.

Class
- width : Integer
name : String
+ Class(width : Integer, name : String)
+ toString() : String
+ setWidth(width : Integer) : void

Figure 5 Représentation des opérations

2.2 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.

2.2.1 Héritage

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

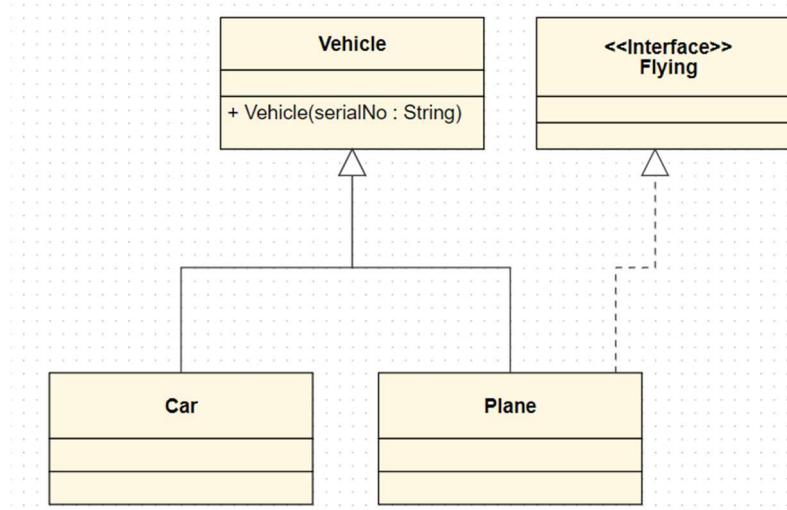


Figure 6 Représentation des héritages

2.2.2 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.

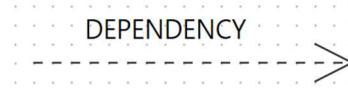


Figure 7 Représentation des dépendances

2.2.3 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.

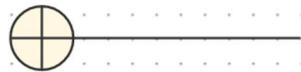


Figure 8 Représentation du lien de classe interne

2.2.4 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

2.2.4.1 Association

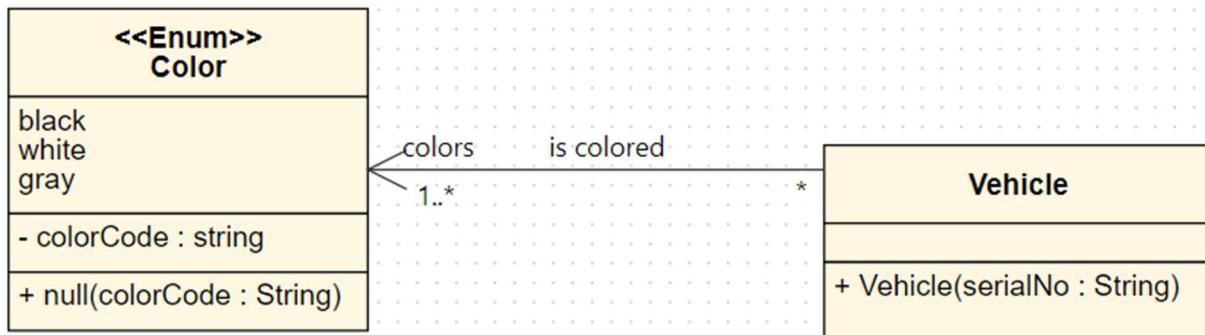


Figure 9 Représentation d'une association

2.2.4.2 Agrégation

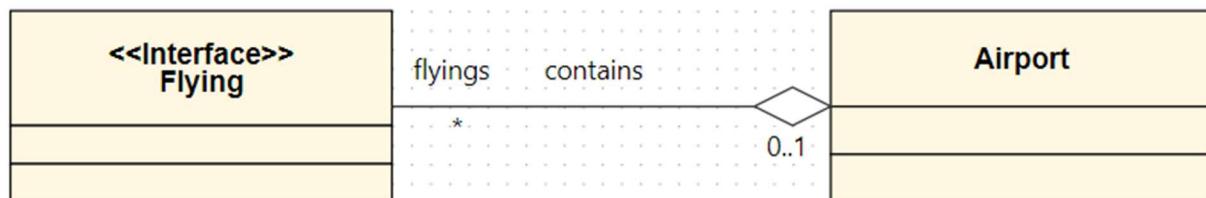


Figure 10 Représentation d'une agrégation

2.2.4.3 Composition

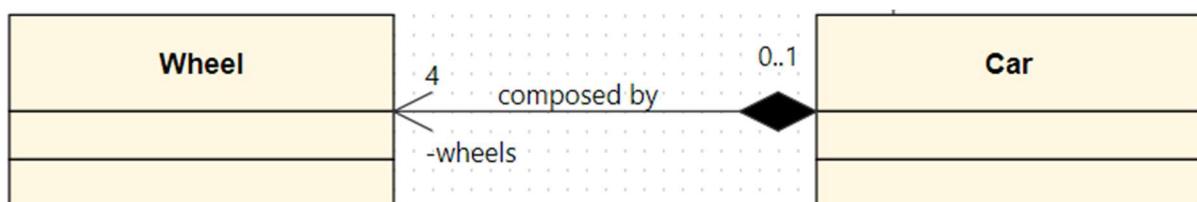


Figure 11 Représentation d'une composition

2.2.5 Les associations multiples

A rédiger

3 Méta-schéma de diagrammes de classes

3.1 Première version

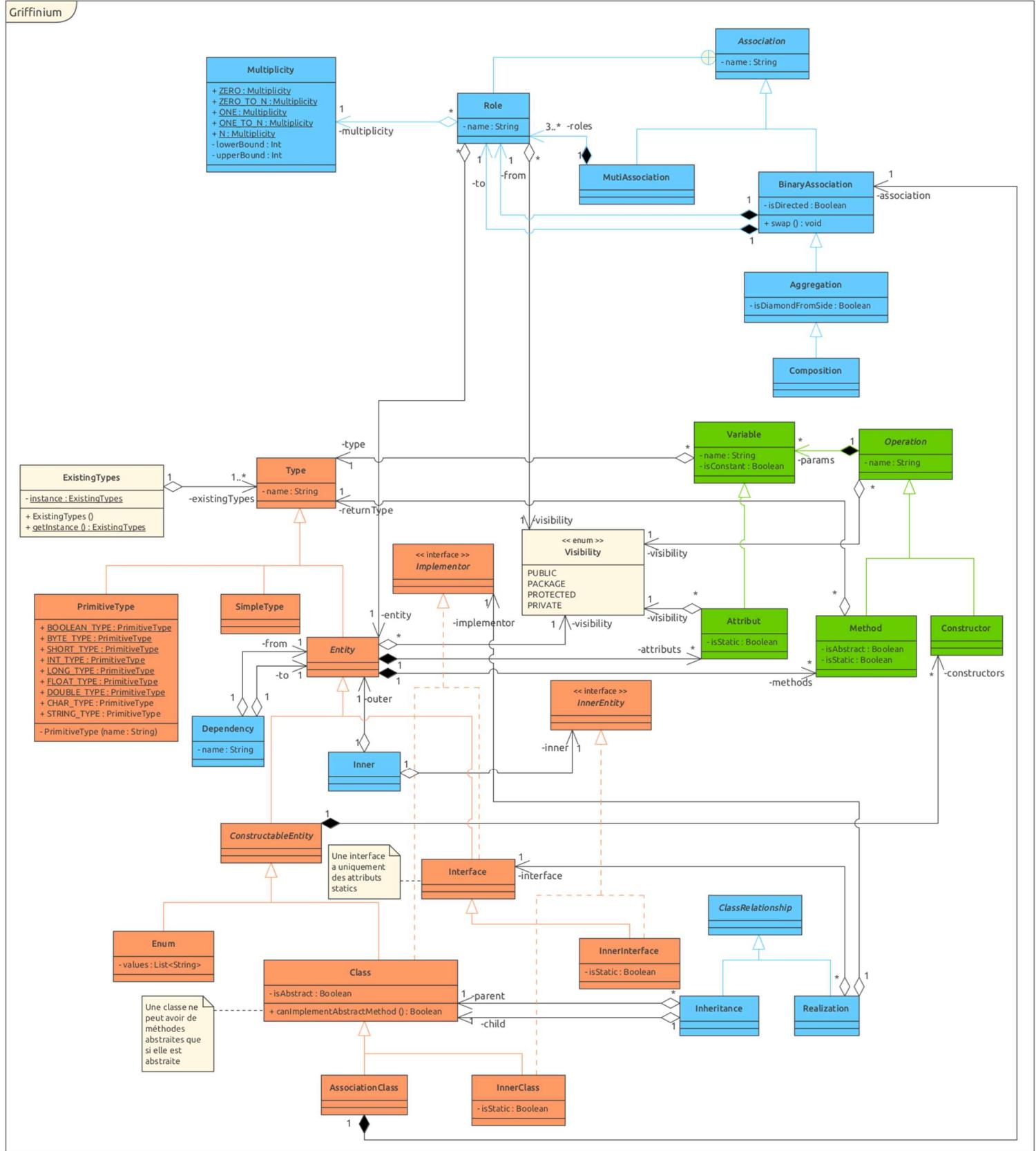


Figure 12 Méta-schéma de l'application

3.1.1 Description de la première version du métaschéma

3.1.1.1 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.

3.1.1.2 En vert

Tous les attributs et méthodes disponibles dans les entités.

3.1.1.3 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.

3.1.1.4 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.

3.2 Deuxième version

3.2.1 Entités

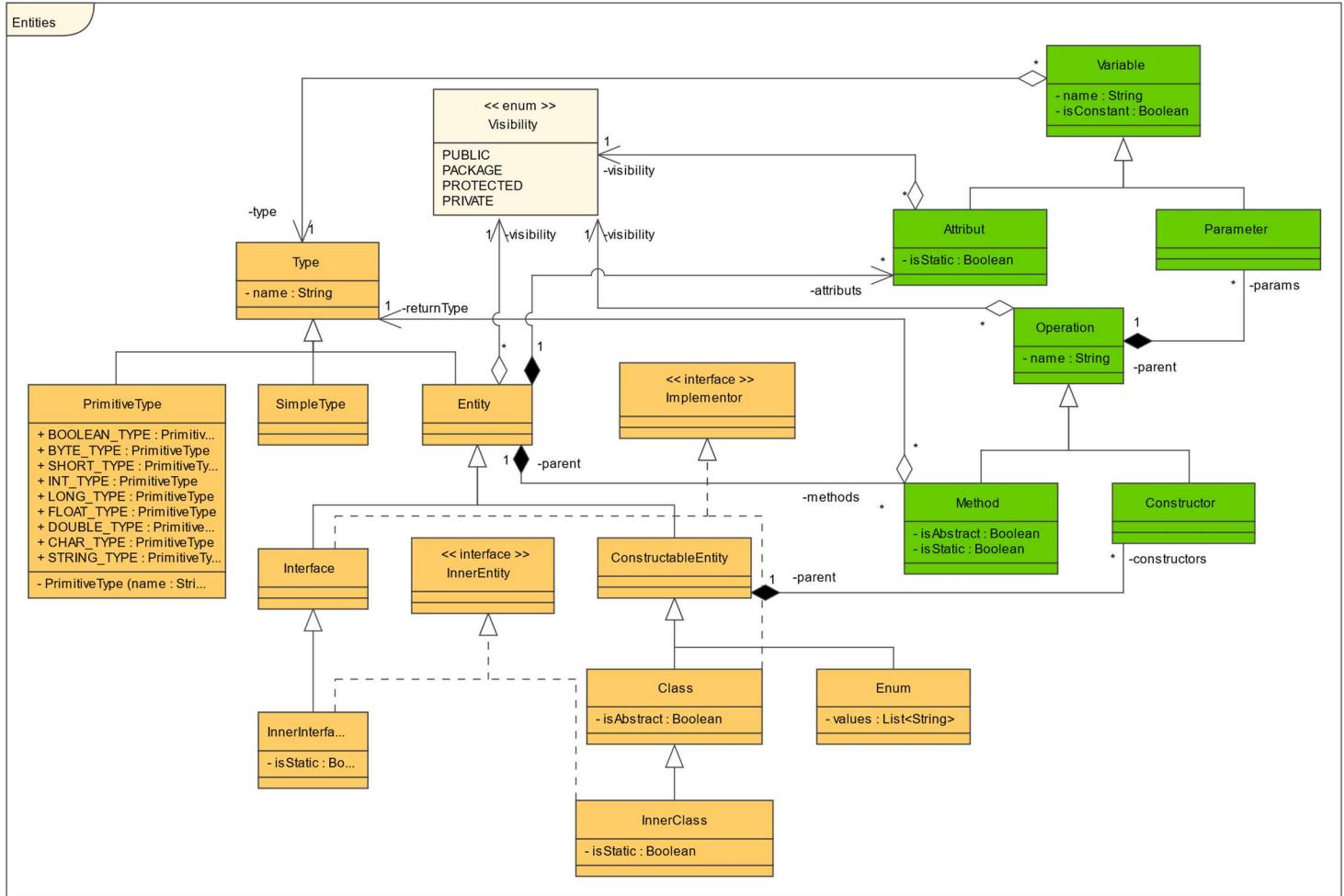


Figure 13 Diagramme de classes des entités - 2ème version

3.2.1.1 Changement depuis la première version

Les entités n'ont pas changé depuis la version un, mis à part qu'`« AssociationClass »` n'hérite plus de `« Class »`. Je me suis rendu compte que c'était plutôt une classe qui devait englober une classe et une association, plutôt que d'être une classe qui hérite de `« Class »` qui a une association.

3.2.2 Liens

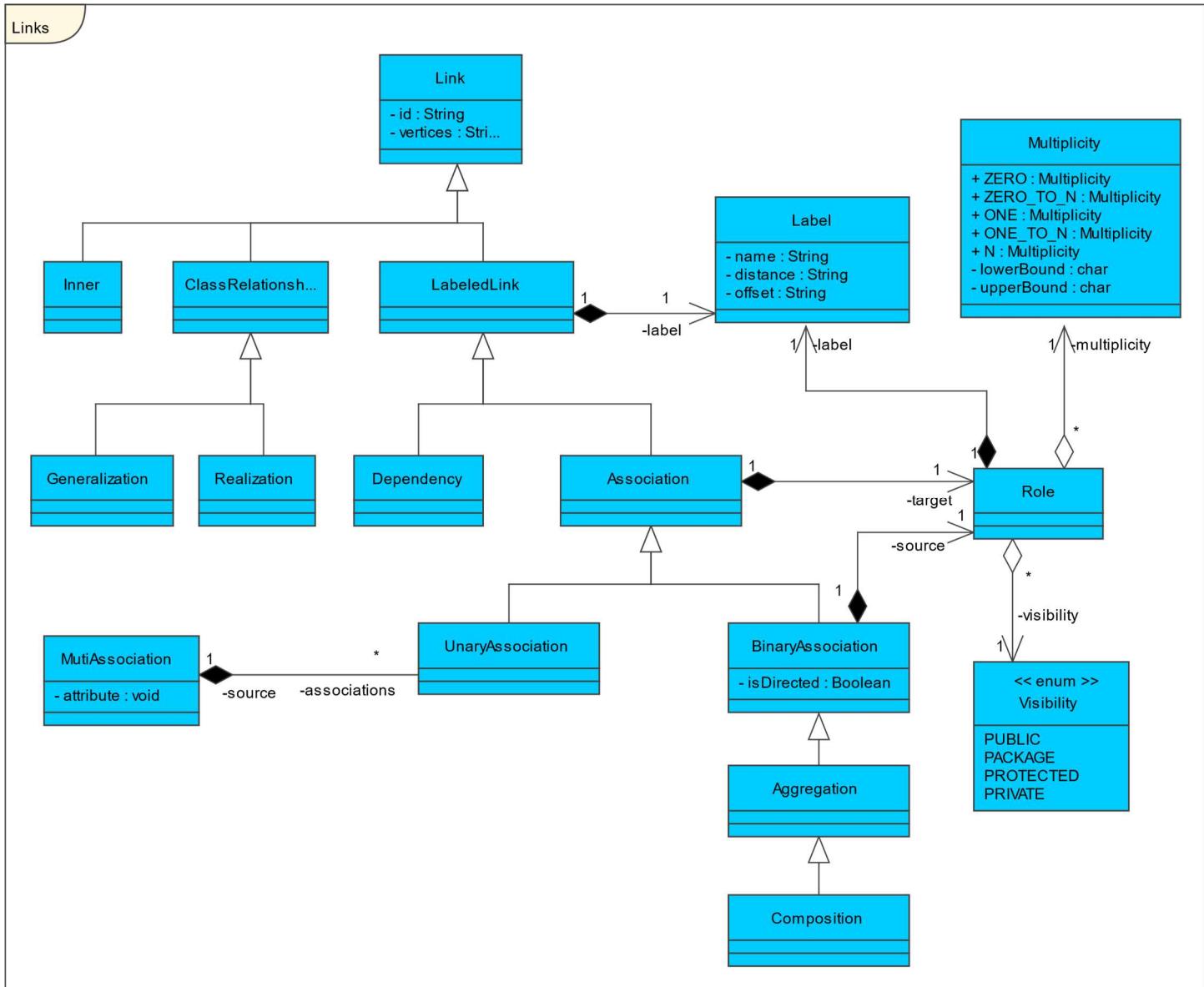


Figure 14 Diagramme de classes des liens - 2ème version

3.2.2.1 Changement depuis la première version

Après avoir commencé à implémenter les liens, autant dans la partie graphique que logique, je me suis rendu compte que :

- Tous les liens avaient une liste de « vertices » (des points balisant le passage du lien).
- Tous les liens ont un id

Dans la première implémentation, il aurait fallu avoir beaucoup de doublons afin que tous les types de liens puissent avoir ses informations.

J'ai également réfléchi à la multiassociation. Il m'a paru judicieux de prendre une multiassociation pour un groupe d'association ayant uniquement une cible. C'est pourquoi j'ai créé la classe « UnaryAssociation » qui est une classe ayant une entité source sans rôle, et une entité cible avec rôle.

J'ai également rencontré d'autres similitudes, telles que les labels, qui avaient besoin d'informations supplémentaires que je suis le contenu textuel.

3.2.3 Interactions liens – entités

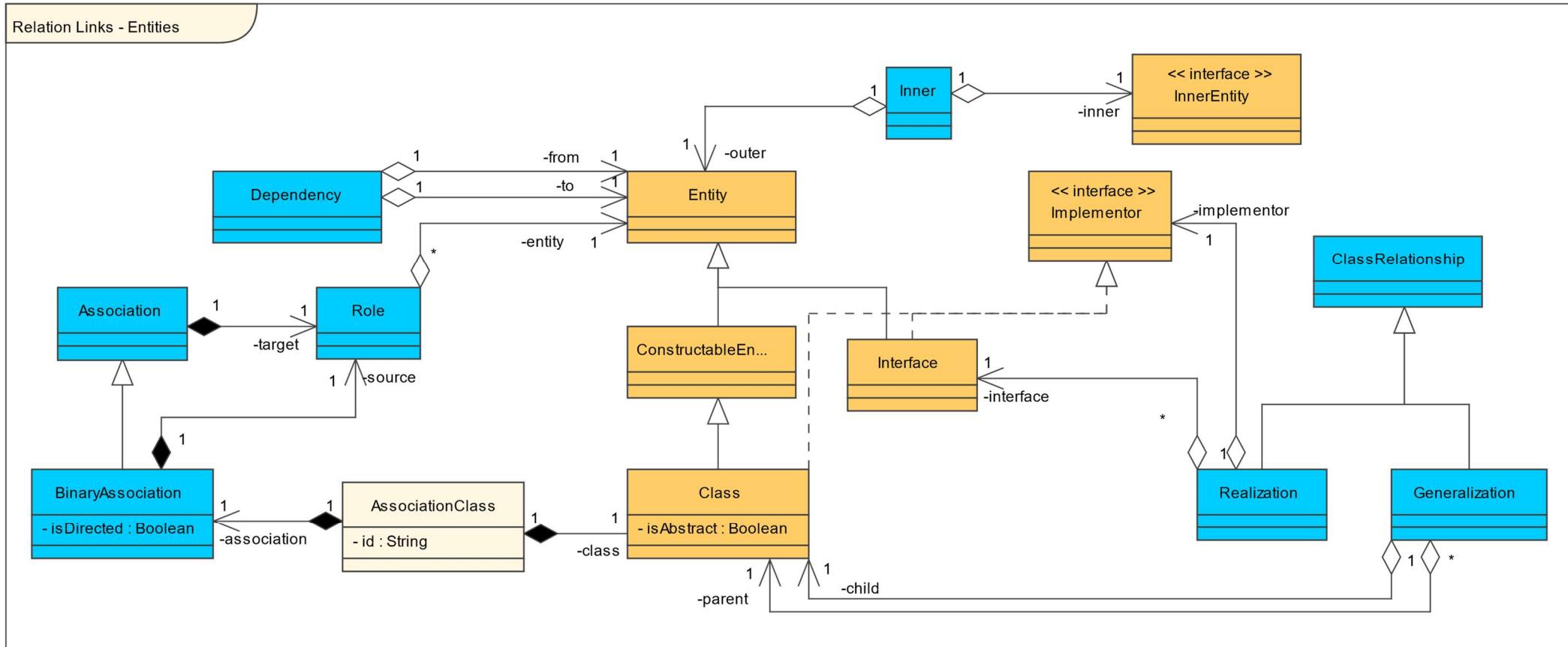


Figure 15 Diagramme de classes des interactions

3.2.4 L'objet ClassDiagram

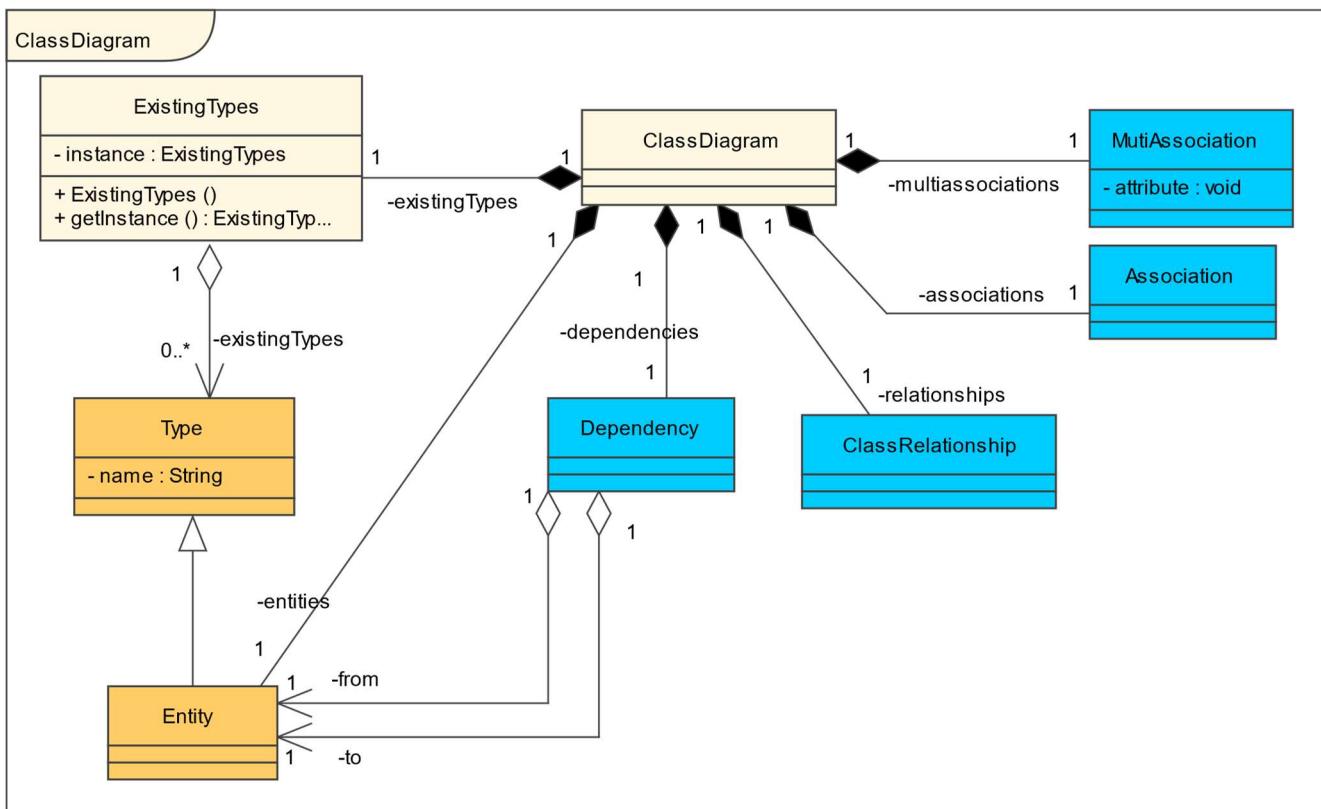


Figure 16 Diagramme de classes de l'éléments racines - 2ème version

3.2.4.1 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.

4 Stockage de données

4.1 Base de données

4.1.1 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.

4.1.2 ORM

4.1.2.1 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.

4.1.2.2 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.

4.2 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 Postgres permet d'utiliser XPath, et on pourrait donc imaginer pouvoir rechercher un diagramme par une entité qui se trouve à l'intérieur.

4.2.1 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.

5 Librairies graphiques

5.1 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.

5.2 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.

6 Fonctionnement de l'application

6.1 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.

6.1.1 Sécurités et validations mises en place

Le mot de passe de l'utilisateur est hashé et salé grâce à l'utilitaire BCrypt.¹

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.

¹ <https://www.mindrot.org/projects/jBCrypt/>

6.1.2 Diagramme de séquences

6.1.2.1 Login

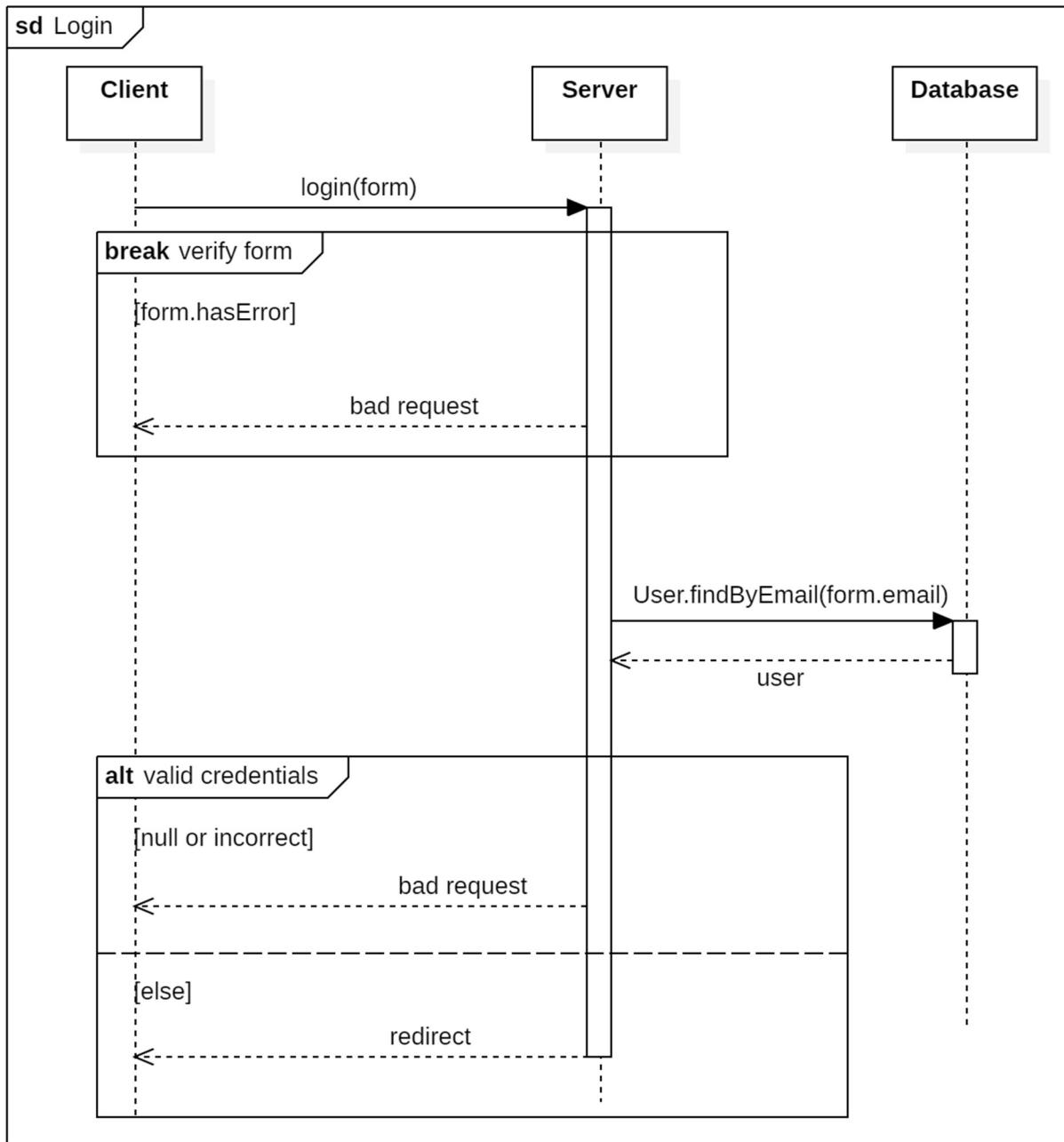


Figure 17 Diagramme de séquence de l'authentification

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

6.1.2.2 Register

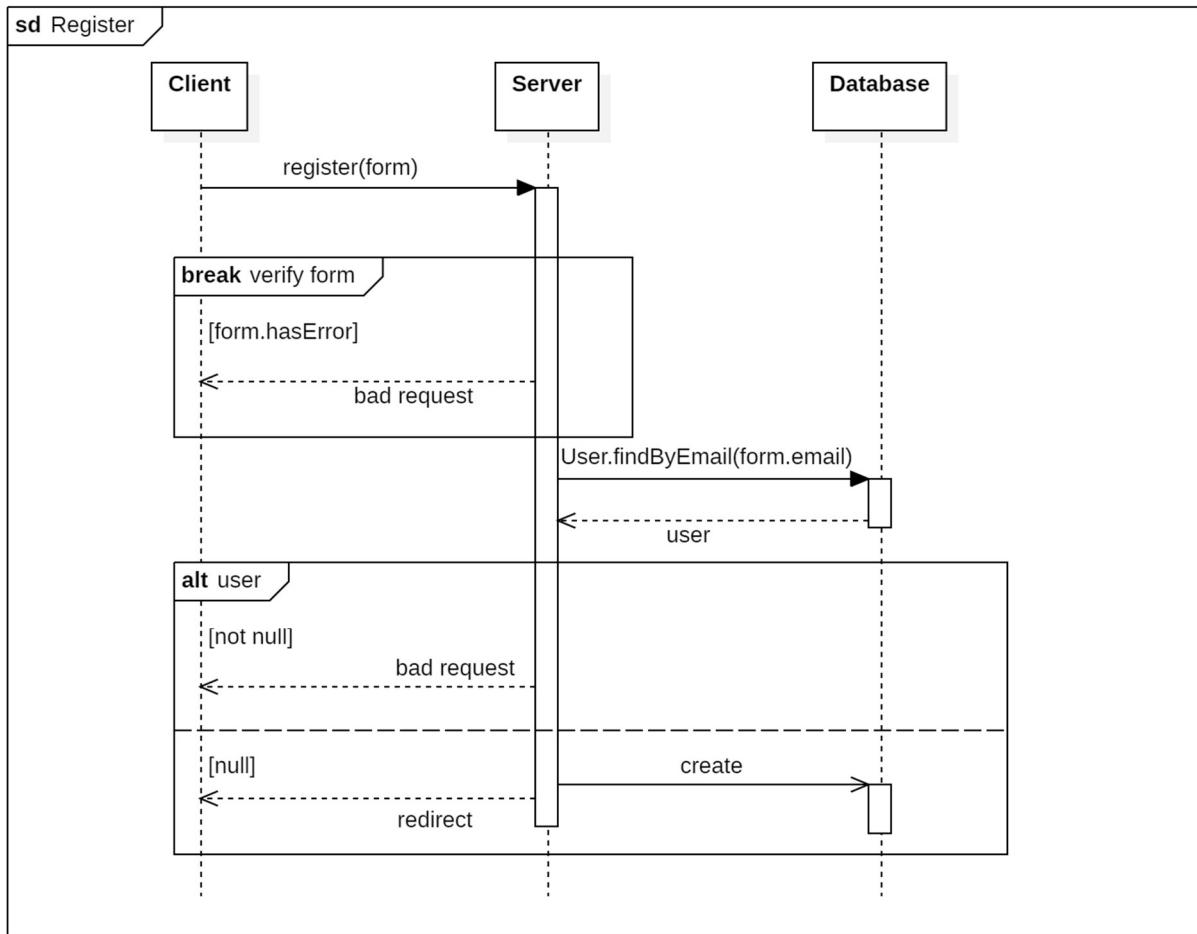


Figure 18 Diagramme de séquence de l'inscription

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

6.2 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.

6.2.1 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.

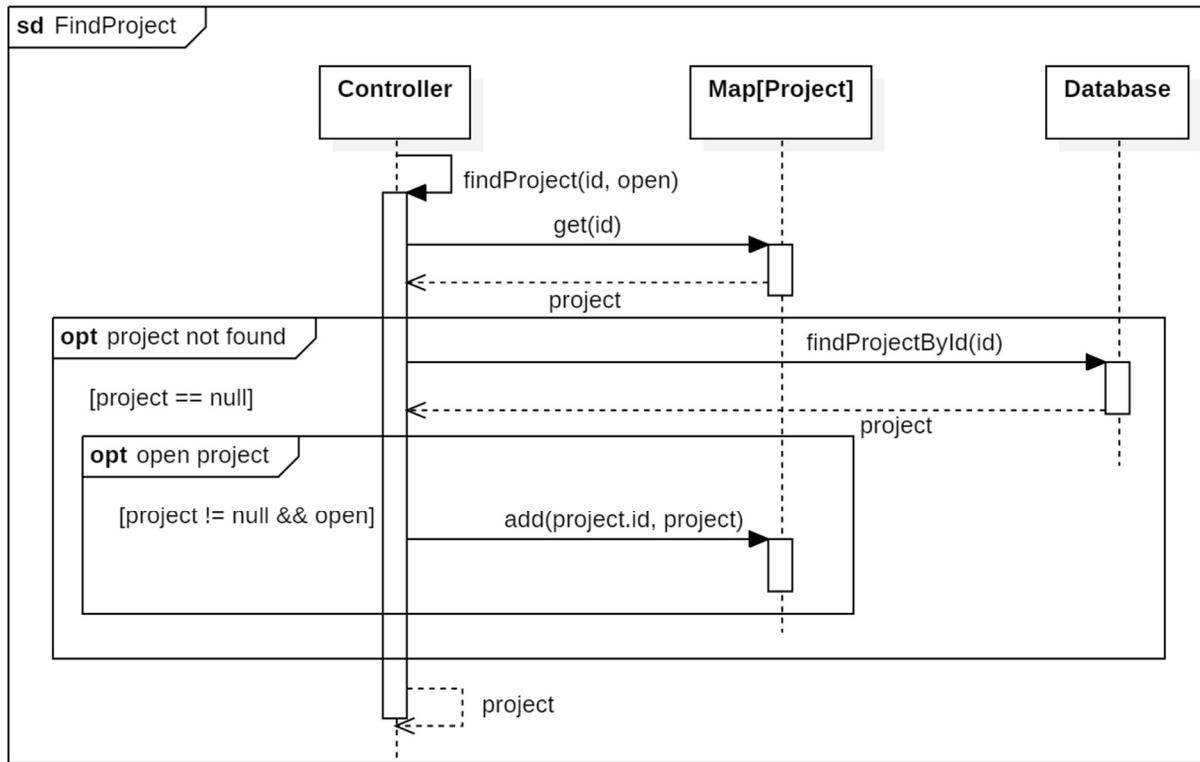


Figure 19 Diagramme de séquence de la recherche/ouverture d'un diagramme

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.²

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
 - o Si aucun utilisateur n'est connecté, le projet se sauvegarde et se ferme.

² <https://medium.com/@nnnsadeh/building-a-reactive-distributed-messaging-server-in-scala-and-akka-with-websockets-c70440c494e3>

6.2.1.1 Crédation

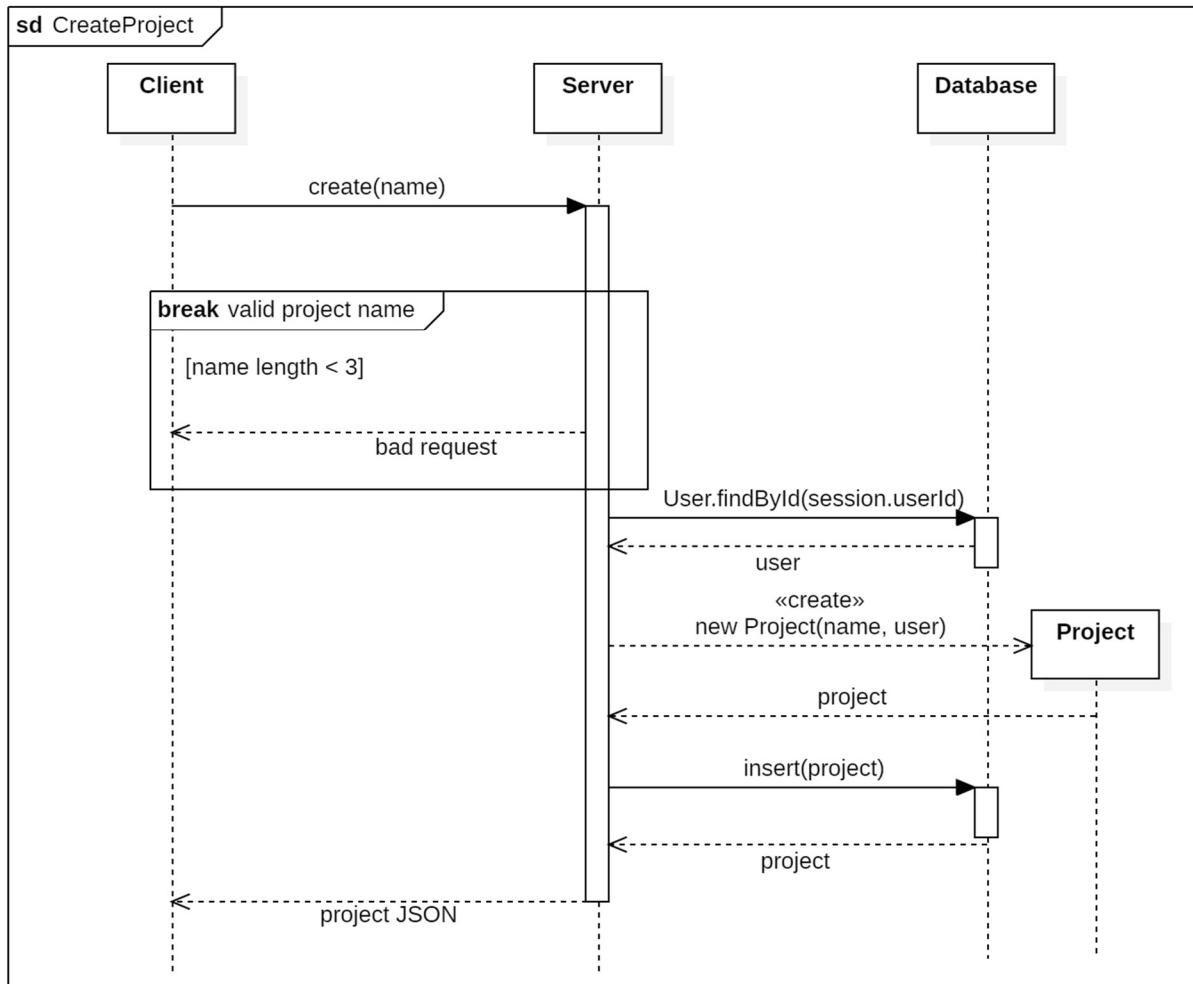


Figure 20 : Diagramme de séquence de la création de projets

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.

6.2.1.2 Mise à jour

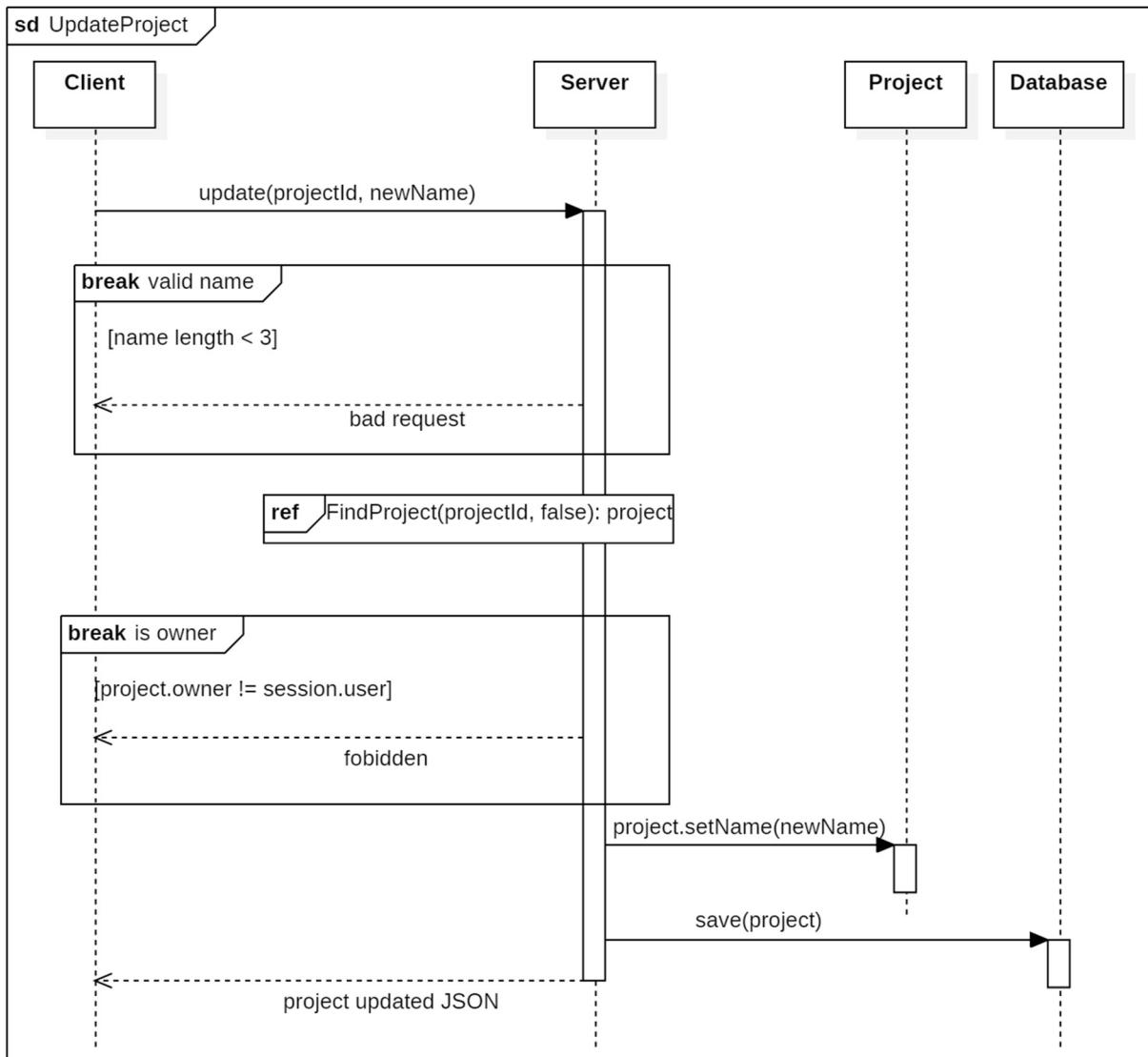


Figure 21 : Diagramme de séquence de la mise à jour d'un projet

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

6.2.1.3 Suppression

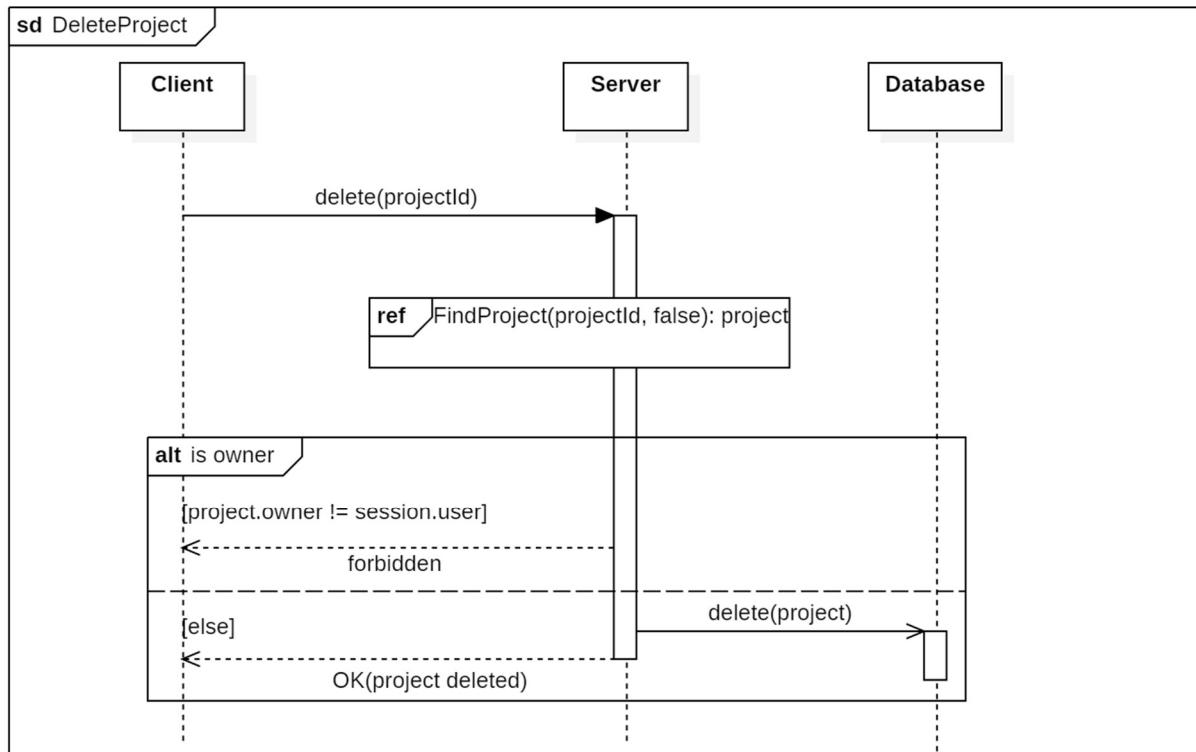


Figure 22 : Diagramme de séquence de la suppression d'un projet

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

6.2.1.4 Récupération des projets d'un membre

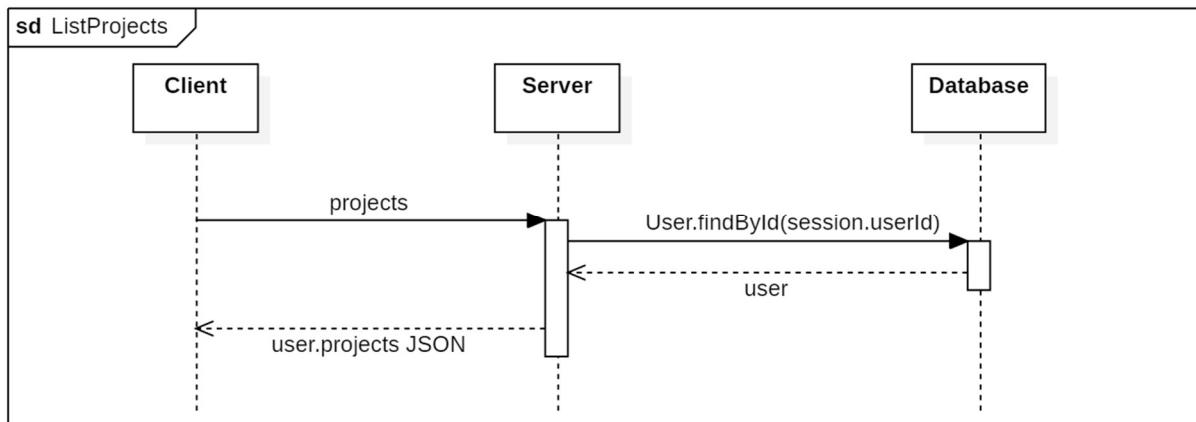


Figure 23 : Diagramme de séquence de la récupération des projets de l'utilisateur connecté

6.2.1.5 Récupération d'un projet spécifique et accès à la page du projet

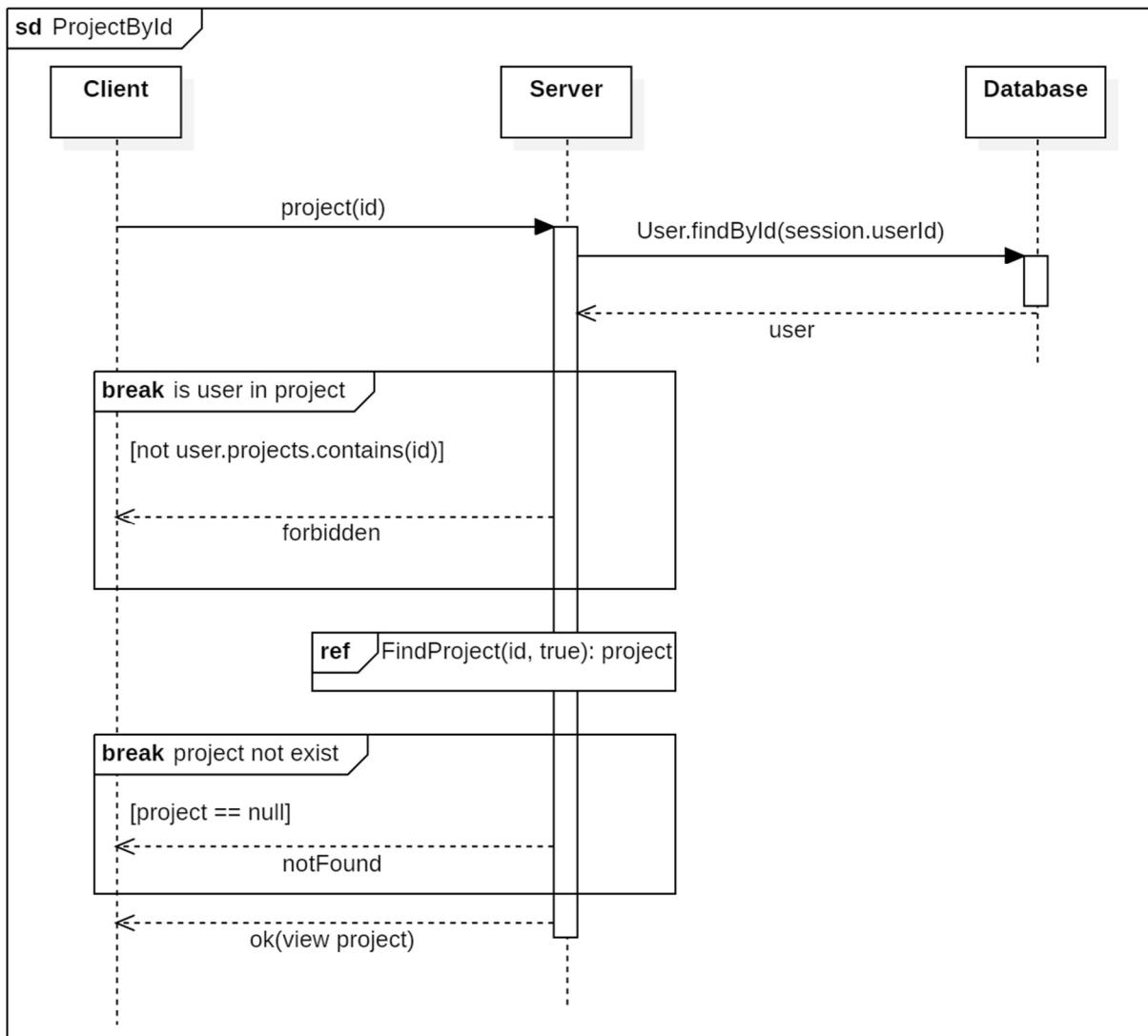


Figure 24 : Diagramme de séquence de la récupération d'un projet spécifique et de sa vue

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.

6.2.2 Collaborateurs

6.2.2.1 Ajout d'un collaborateur

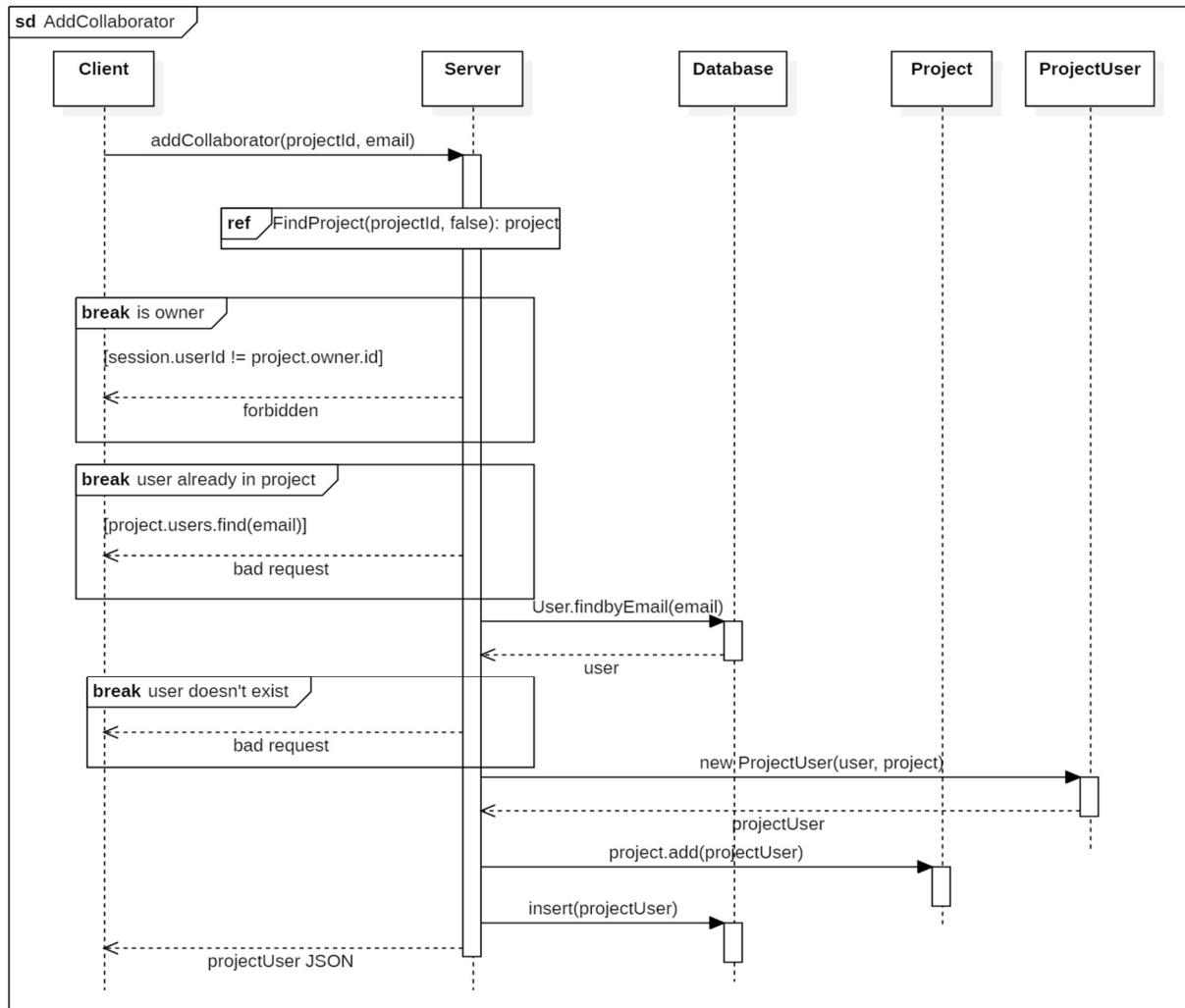


Figure 25 : Diagramme de séquence de l'ajout d'un collaborateur

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

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

6.2.2.2 Modification des droits d'un utilisateur

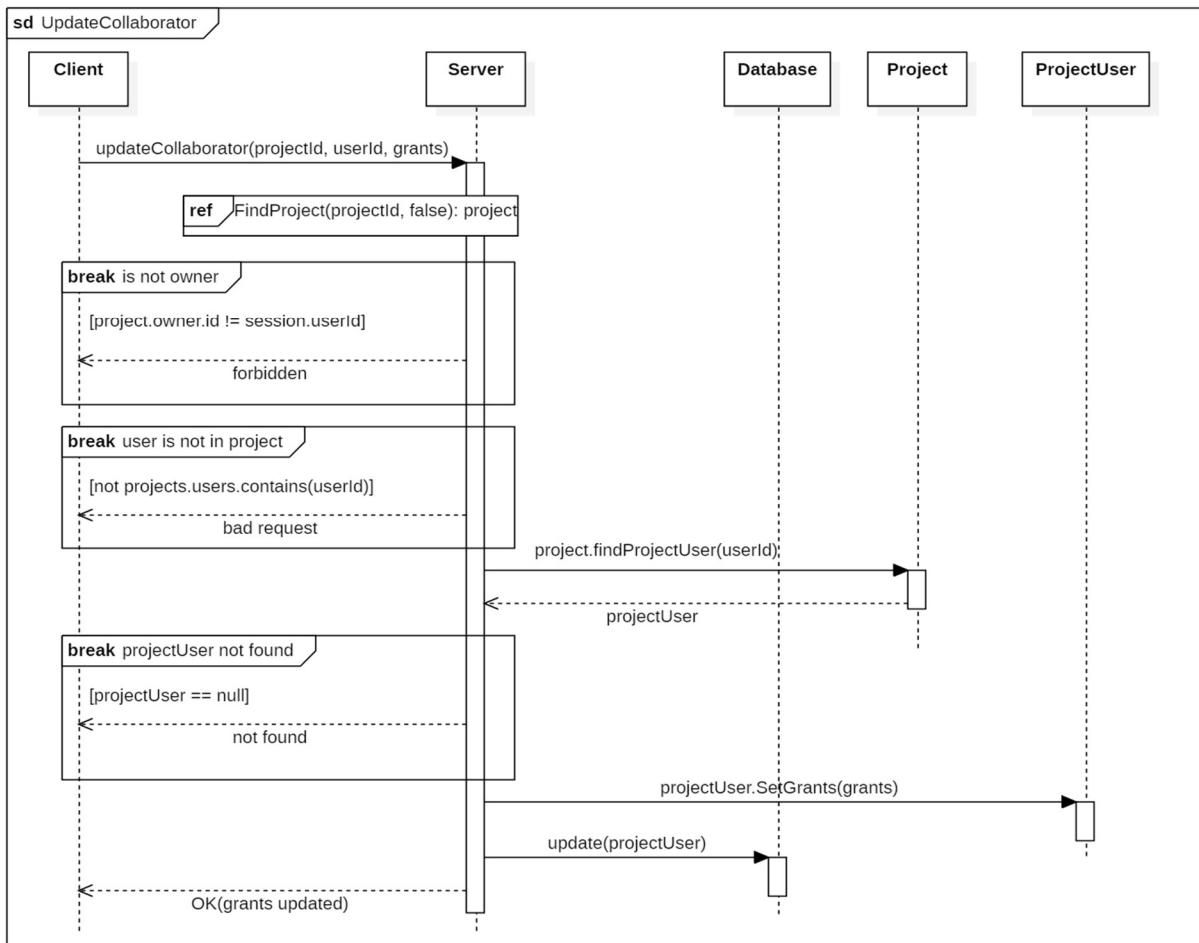


Figure 26 : Diagramme de séquence de la modification des droits d'un utilisateur

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

6.2.2.3 Suppression d'un collaborateur

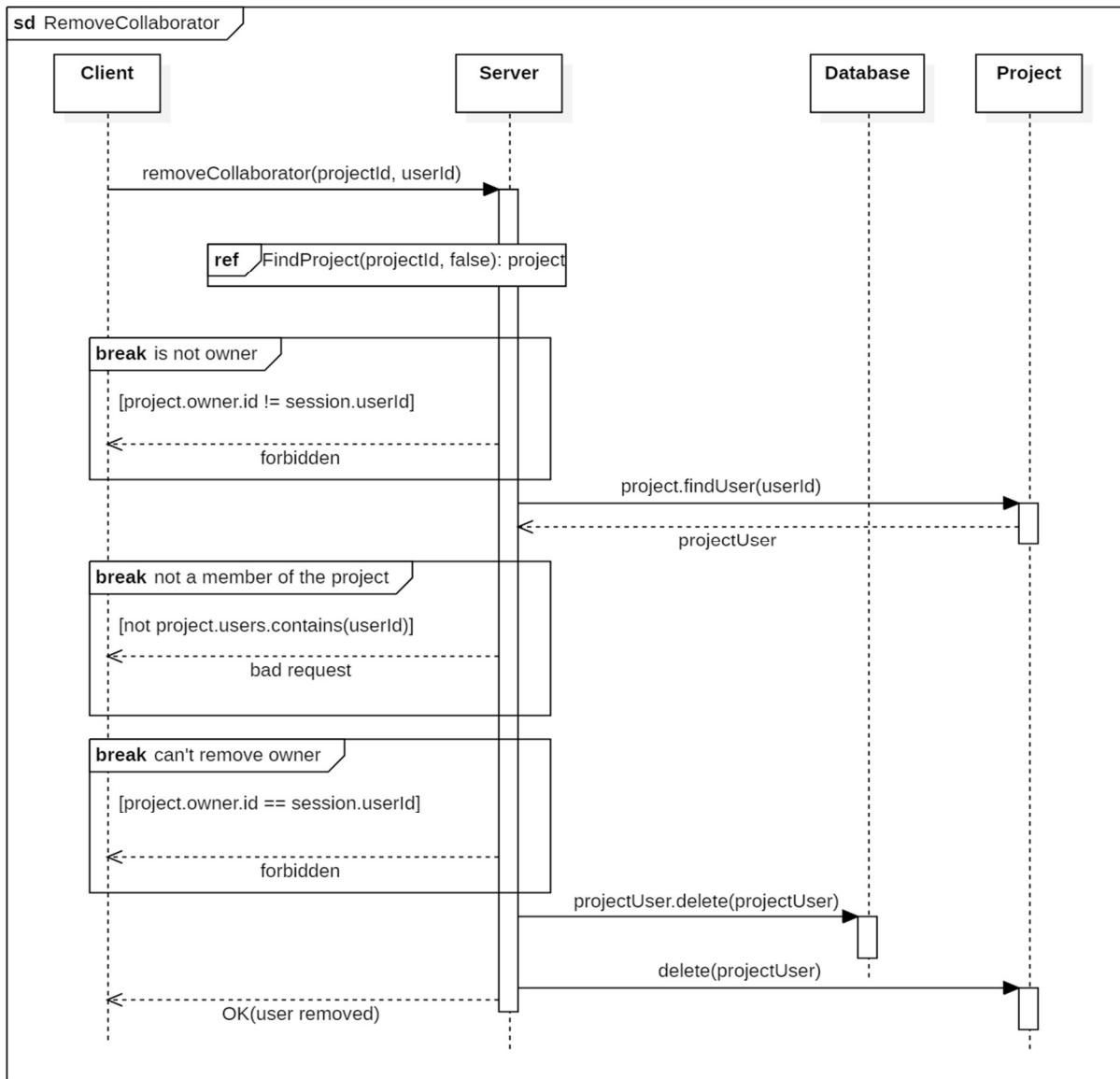


Figure 27 : Diagramme de séquence de la suppression d'un collaborateur

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.

6.3 Collaboration sur les projets

6.3.1 Websocket

6.3.1.1 Connexion

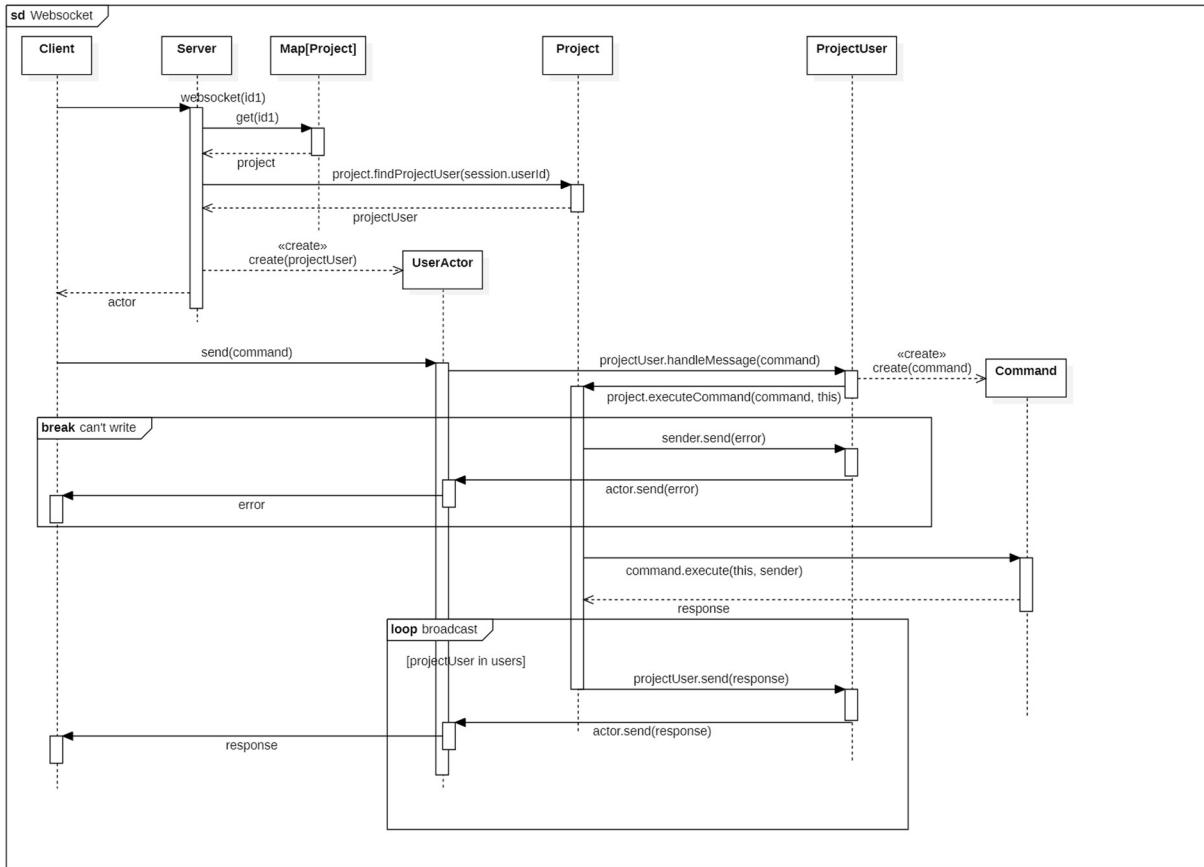


Figure 28 : Diagramme de séquence du websocket

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.³

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.

6.3.1.2 Déconnexion

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

6.3.1.2.1 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.

³ <https://www.playframework.com/documentation/2.8.x/JavaWebSockets>

6.3.1.2.2 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.

6.3.2 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.

6.3.2.1 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.

6.3.2.2 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.

6.3.2.3 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.

6.3.2.4 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

6.3.2.5 Remove

Cette commande sert simplement à retirer une entité du diagramme.

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

7 Interface utilisateur

7.1 Page d'accueil pour utilisateur non connecté

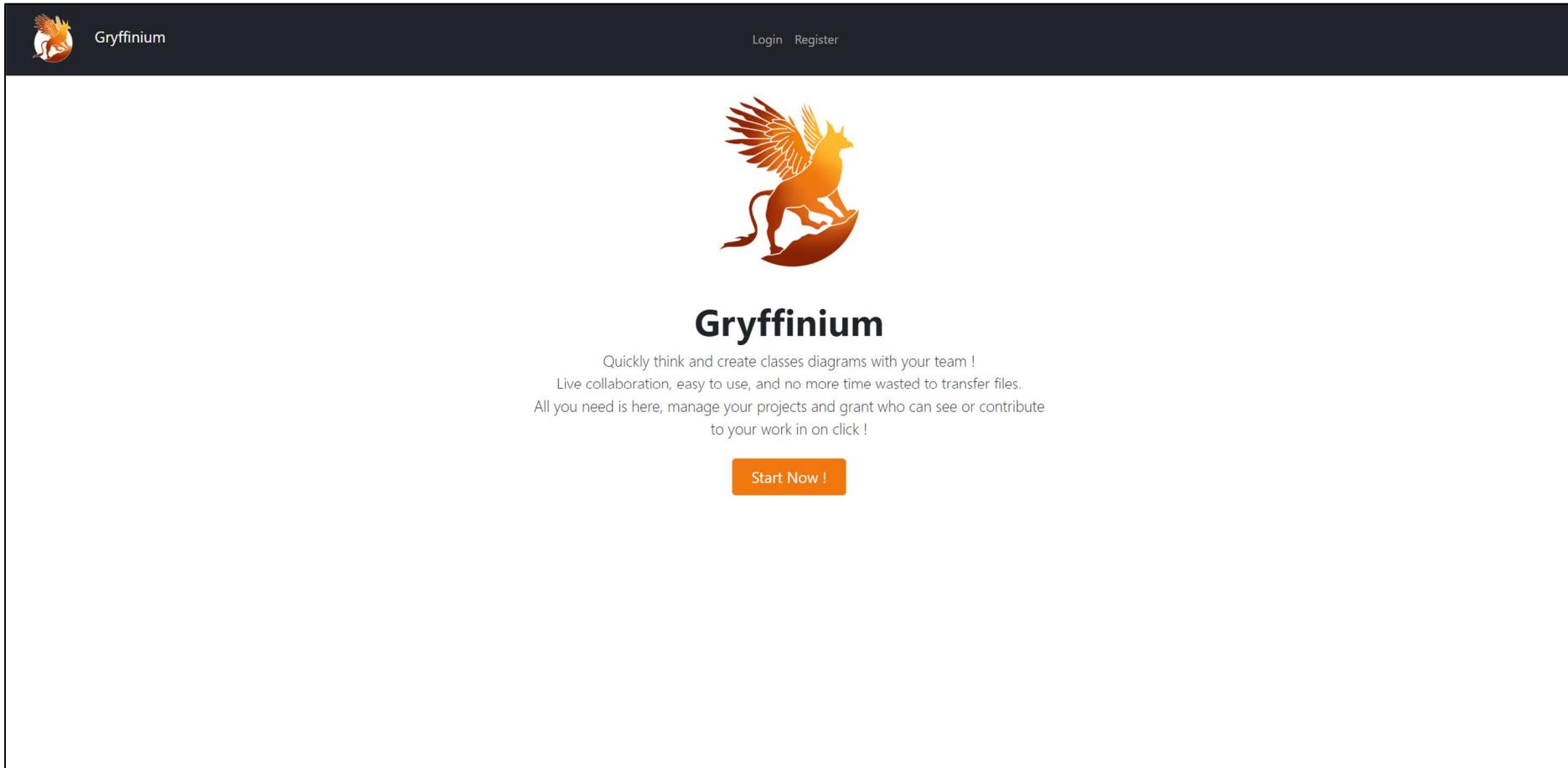


Figure 29 : Page d'accueil pour utilisateur non connecté

7.2 Page d'accueil pour utilisateur connecté

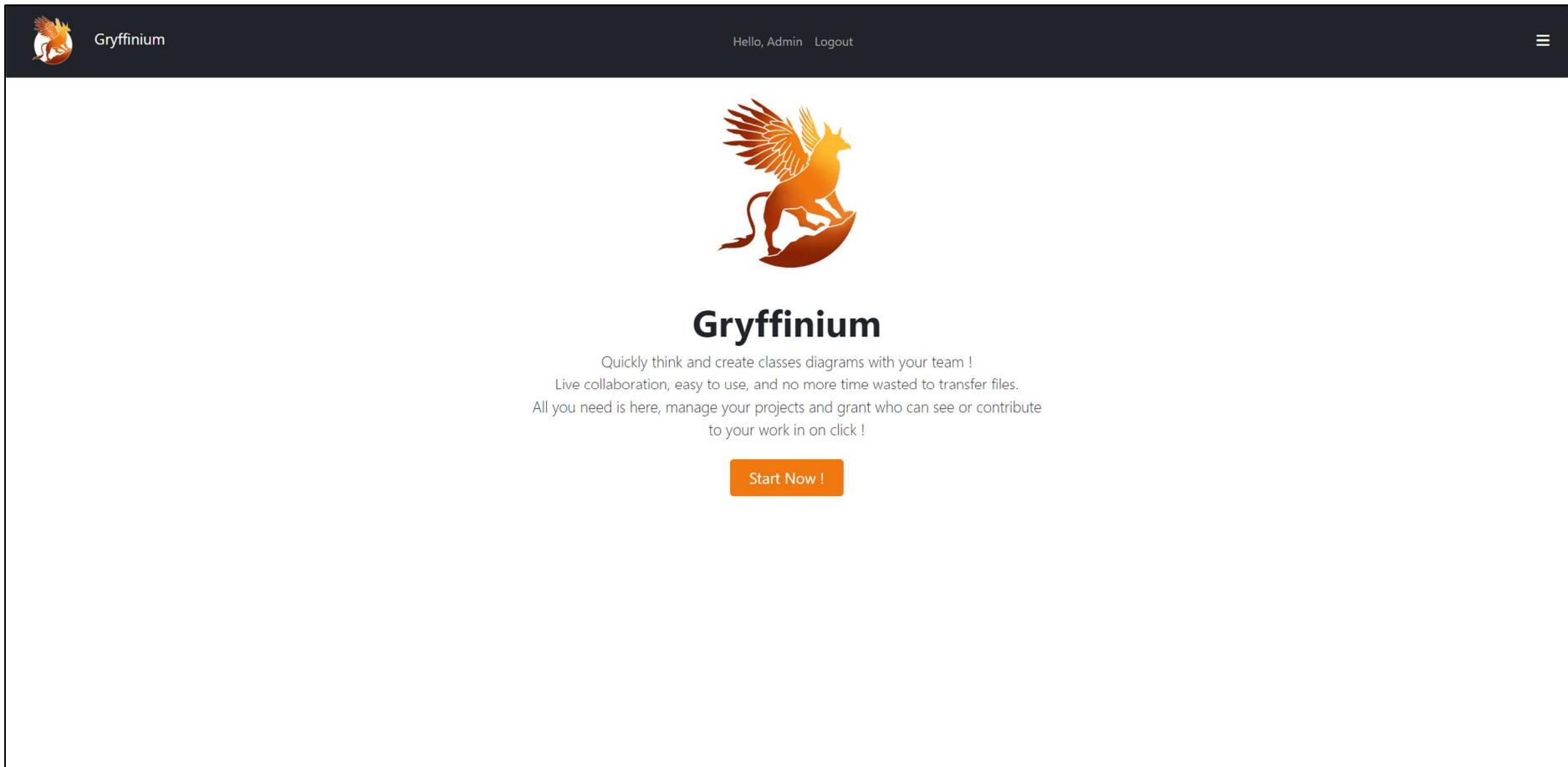
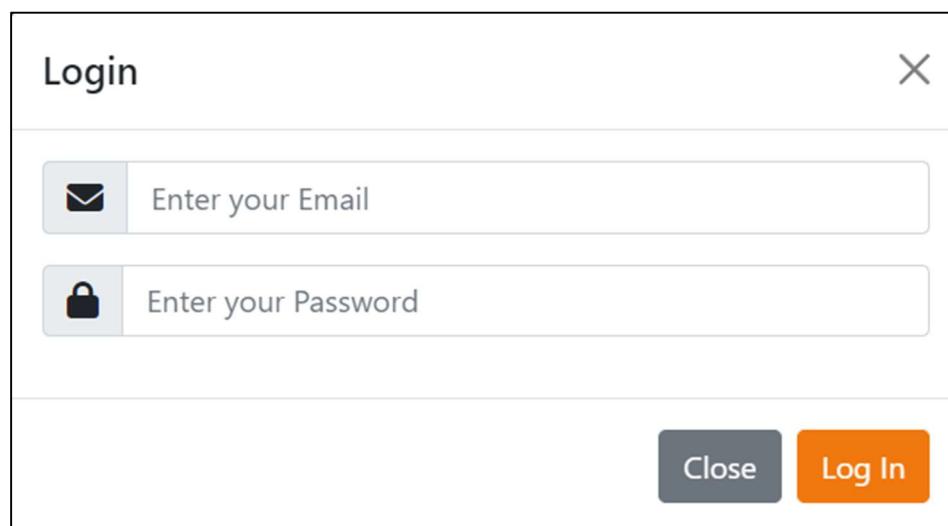


Figure 30 : Page d'accueil pour utilisateur connecté

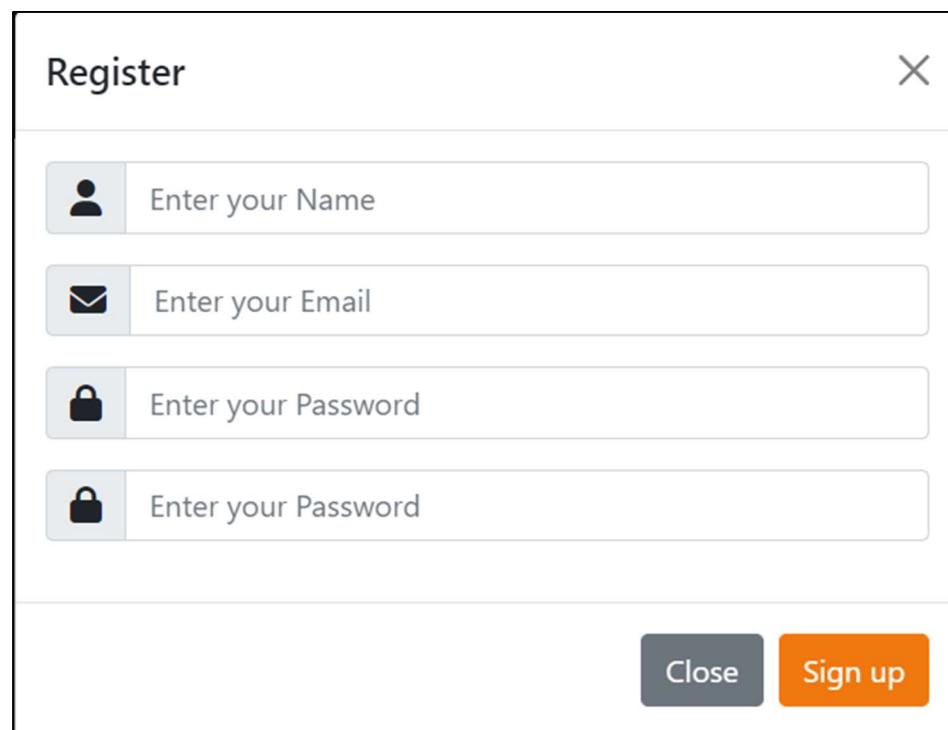
7.3 Modal d'authentification



A screenshot of a login modal window titled "Login". The window has a light gray background and a white header bar with the title "Login" and a close button (an "X"). It contains two input fields: one for "Enter your Email" with a mail icon and one for "Enter your Password" with a lock icon. At the bottom right are two buttons: "Close" (gray) and "Log In" (orange).

Figure 31 : Modal d'authentification

7.4 Modal d'inscription



A screenshot of a registration modal window titled "Register". The window has a light gray background and a white header bar with the title "Register" and a close button (an "X"). It contains four input fields: "Enter your Name" (with a person icon), "Enter your Email" (with a mail icon), and two "Enter your Password" fields (each with a lock icon). At the bottom right are two buttons: "Close" (gray) and "Sign up" (orange).

Figure 32 : Modal d'inscription

7.5 Page d'accueil avec la liste des projets

The screenshot shows the Gryffinium application interface. At the top left is the Gryffinium logo (a golden griffin) and the word "Gryffinium". At the top right are "Hello, Admin" and "Logout" links. The main content area features a large golden griffin logo, the word "Gryffinium" in bold, and a brief description: "Quickly think and create classes diagrams with your team ! Live collaboration, easy to use, and no more time wasted to transfer files. All you need is here, manage your projects and grant who can see or contribute to your work in one click !" Below this is an orange "Start Now !" button. To the right is a sidebar titled "Projects List" with a search bar, a "+" button, and a table showing one project: "Gryffinium" with an edit icon.

Figure 33 : Page d'accueil avec la liste des projets

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

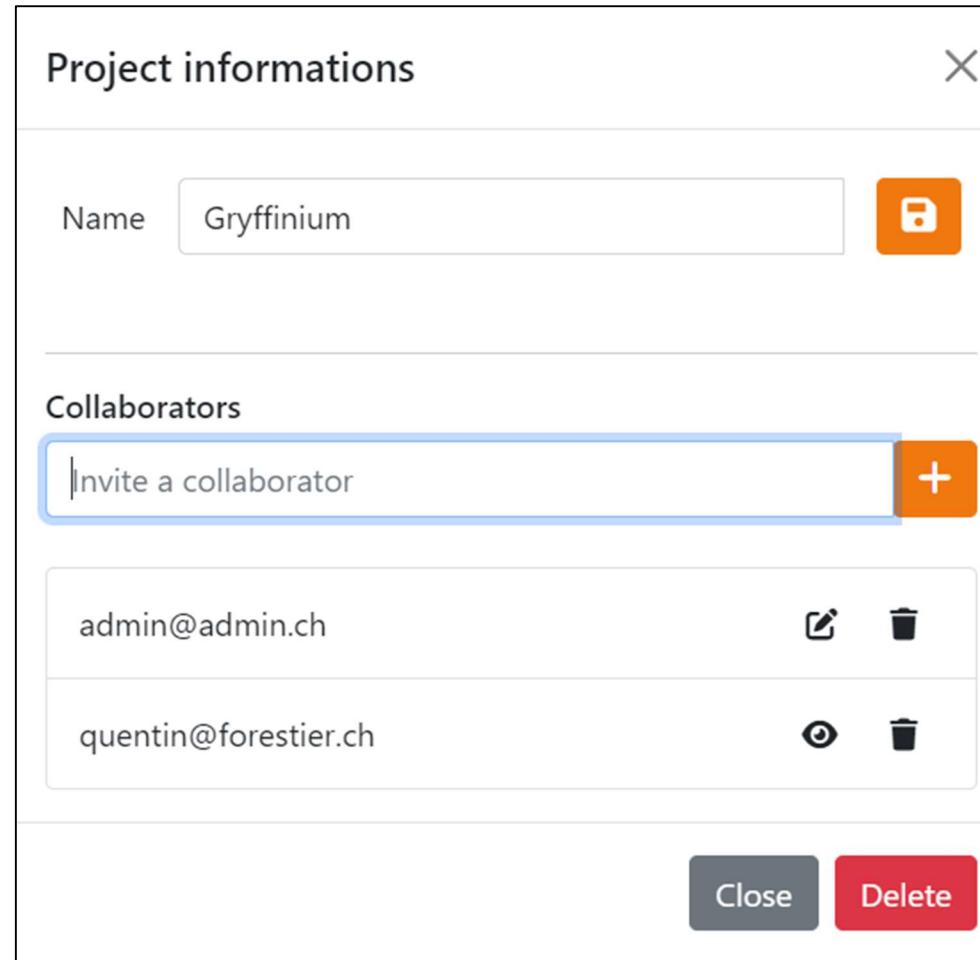


Figure 34 : Modal de création et mise à jour de projet pour le propriétaire

7.7 Modal de visualisation des détails de projet

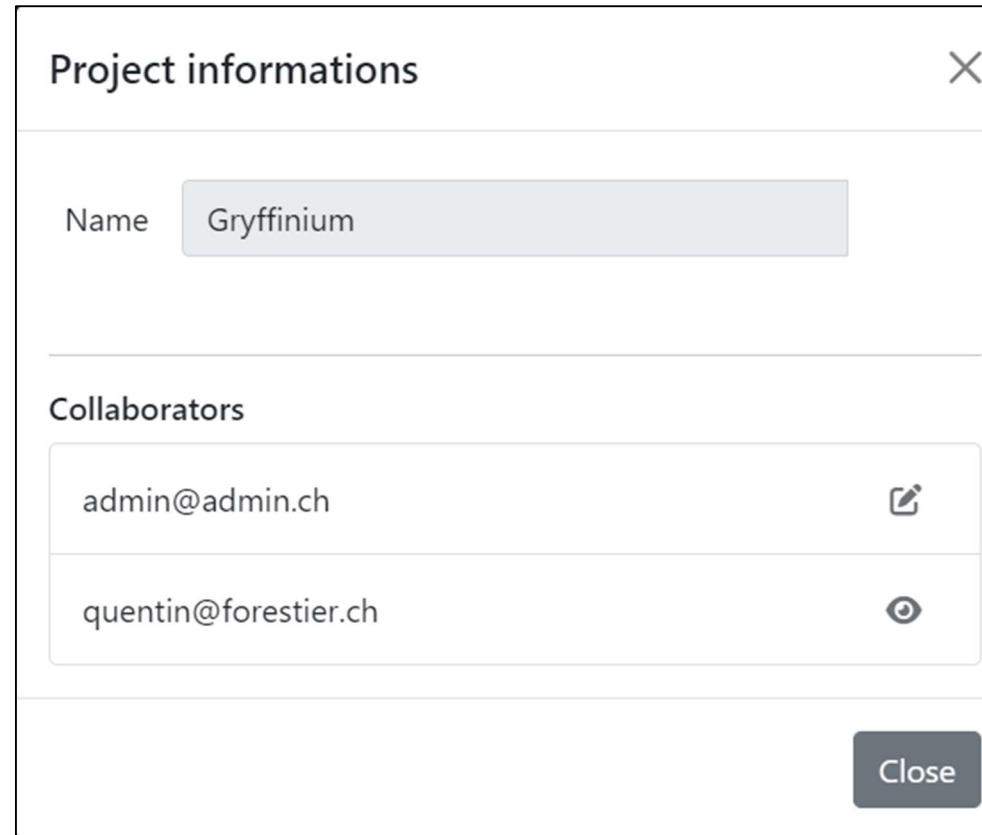


Figure 35 : Modal de visualisation des détails de projet

7.8 Modal de chat des projets

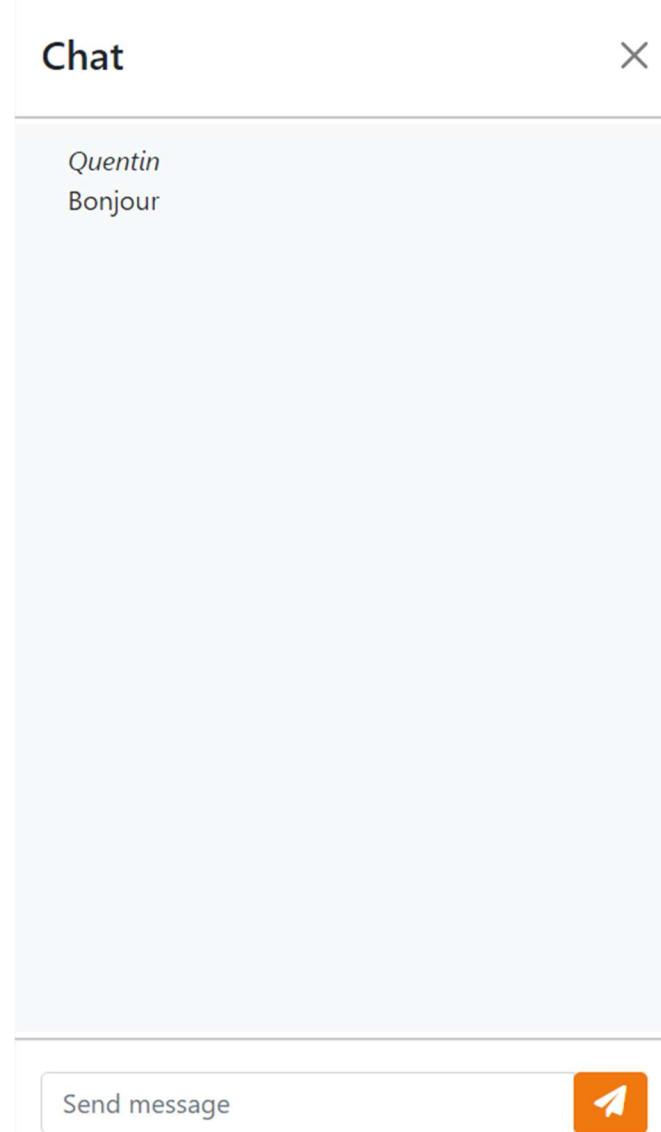


Figure 36 : Modal de chat des projets

7.9 Éditeur de diagrammes

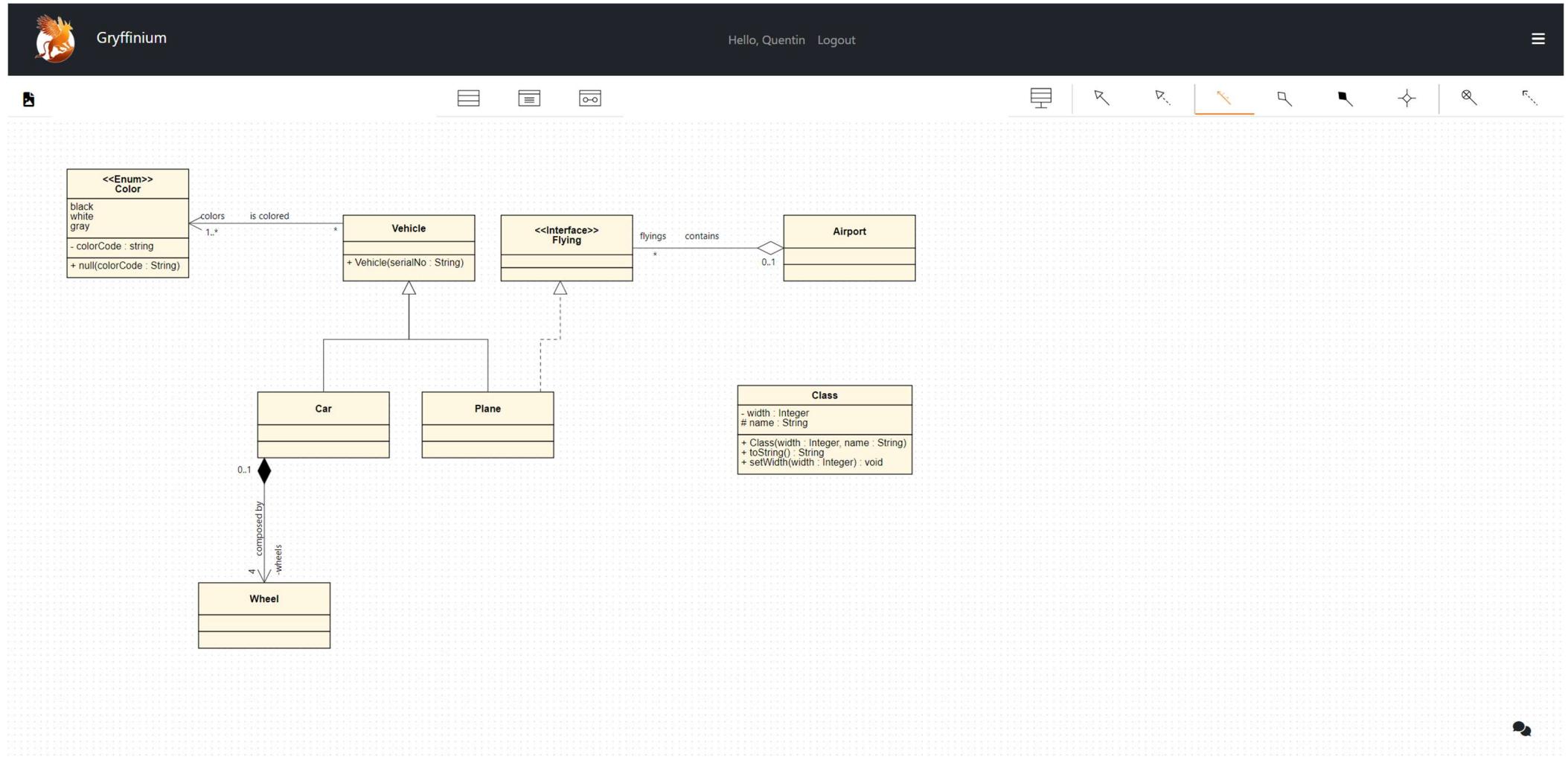


Figure 37 Interface de l'éditeur de diagrammes

7.9.1 ToolBox

7.9.1.1 Entités

On 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.

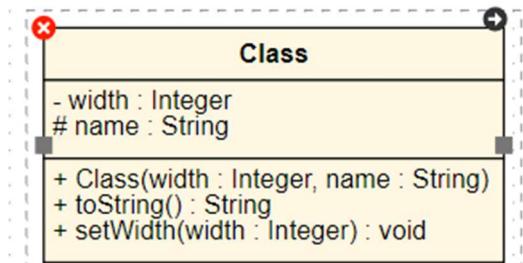


Figure 38 ToolBox d'une entité

7.9.1.2 Liens

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

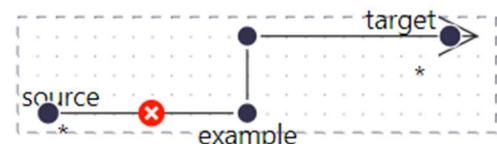


Figure 39 ToolBox d'un lien

7.9.2 Toolbar

Icône	Utilité
	Permet 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

8 Conclusion

8.1 Problèmes rencontrés

8.1.1 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.

8.1.1.1 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 apprêhender les exemples et la documentation, afin de comprendre les concepts plus avancés.

8.1.2 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 constraint à stocker le diagramme de classes sous forme d'XML.

8.1.2.1 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.

8.2 Améliorations et problèmes

8.2.1 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.

8.2.2 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.

8.2.3 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.

8.2.4 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.

8.2.5 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.

8.2.6 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.

8.2.7 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.

8.2.8 Association Class

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

8.2.9 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 possible⁴.

8.2.10 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.

9 Table des illustrations

Figure 1 Représentation d'une classe.....	10
Figure 2 Représentation d'une énumération	10
Figure 3 Représentation d'une interface.....	10
Figure 4 Représentation d'attributs	10
Figure 5 Représentation des opérations	11
Figure 6 Représentation des héritages.....	11
Figure 7 Représentation des dépendances	11
Figure 8 Représentation du lien de classe interne	12
Figure 9 Représentation d'une association	12
Figure 10 Représentation d'une agrégation.....	12
Figure 11 Représentation d'une composition	12
Figure 12 Métaschéma de l'application	13
Figure 13 Diagramme de classes des entités - 2ème version.....	0
Figure 14 Diagramme de classes des liens - 2ème version	1
Figure 15 Diagramme de classes des interactions.....	2
Figure 16 Diagramme de classes de l'éléments racines - 2ème version	3
Figure 17 Diagramme de séquence de l'authentification	6
Figure 18 Diagramme de séquence de l'inscription	7
Figure 19 Diagramme de séquence de la recherche/ouverture d'un diagramme	8
Figure 20 : Diagramme de séquence de la création de projets.....	9
Figure 21 : Diagramme de séquence de la mise à jour d'un projet.....	10
Figure 22 : Diagramme de séquence de la suppression d'un projet	11
Figure 23 : Diagramme de séquence de la récupération des projets de l'utilisateur connecté.....	11
Figure 24 : Diagramme de séquence de la récupération d'un projet spécifique et de sa vue	12
Figure 25 : Diagramme de séquence de l'ajout d'un collaborateur	13
Figure 26 : Diagramme de séquence de la modification des droits d'un utilisateur.....	14
Figure 27 : Diagramme de séquence de la suppression d'un collaborateur	15
Figure 28 : Diagramme de séquence du websocket.....	16
Figure 29 : Page d'accueil pour utilisateur non connecté	18
Figure 30 : Page d'accueil pour utilisateur connecté.....	19
Figure 31 : Modal d'authentification.....	20
Figure 32 : Modal d'inscription.....	20
Figure 33 : Page d'accueil avec la liste des projets.....	21

⁴ <https://jsfiddle.net/kumilingus/rpw5u8mq/>

Figure 34 : Modal de création et mise à jour de projet pour le propriétaire.....	22
Figure 35 : Modal de visualisation des détails de projet.....	23
Figure 36 : Modal de chat des projets	24
Figure 37 Interface de l'éditeur de diagrammes	25
Figure 38 ToolBox d'une entité.....	26
Figure 39 ToolBox d'un lien	26

10 Annexes

- Journal de travail