

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

Travail de Bachelor 2021-2022

Gryffinium

Résumé publiable

Diagrammes de classes UML

Les diagrammes de classes UML (Unified Modeling Language) 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 entre elles.

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 programmation. Ils permettent également 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 serveur. Ceci les rend disponibles 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

Enseignant responsable :

Pier Donini

HEIG-VD :

Pier Donini

Date et lieu :

Yverdon-les-Bains, le 29 juillet 2022

Date et lieu :

.....

Date et lieu :

.....

Signature :

...*Q. Forestier*.....

Signature :

.....

Signature :

.....

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 27 juillet 2022


Quentin Forestier

Table des matières

1	INTRODUCTION.....	6
1.1	CAHIER DES CHARGES	6
2	UNIFIED MODELING LANGUAGE (UML).....	9
2.1	STRUCTURES DE DONNEES	9
2.2	LIEN ENTRE LES STRUCTURES DE DONNEES.....	10
3	METASHEMA DE DIAGRAMMES DE CLASSES.....	13
3.1	SCHEMA COMPLET	13
3.2	ENTITES	13
3.3	LIENS	14
3.4	INTERACTIONS LIENS – ENTITES.....	15
3.5	L’OBJET CLASSDIAGRAM	16
4	ARCHITECTURE DE L’APPLICATION.....	17
4.1	DESCRIPTION GLOBALE.....	17
4.2	FRAMEWORK WEB	18
4.3	STOCKAGE DE DONNEES	19
5	LIBRAIRIES GRAPHIQUES.....	24
5.1	DESIGN DE L’INTERFACE GLOBAL	24
5.2	DESIGN DE DIAGRAMME DE CLASSES.....	24
6	GESTION DE LA CONCURRENCE.....	26
6.1	MECANISME EN PLACE	26
6.2	PROBLEME DETECTE	26
6.3	AMELIORATIONS POSSIBLES	26
7	INTERFACE UTILISATEUR.....	27
7.1	PAGE D’ACCUEIL POUR UTILISATEUR NON CONNECTE	27
7.2	PAGE D’ACCUEIL POUR UTILISATEUR CONNECTE	28
7.3	MODAL D’AUTHENTIFICATION	29
7.4	MODAL D’INSCRIPTION	29
7.5	PAGE D’ACCUEIL AVEC LA LISTE DES PROJETS.....	30
7.6	MODAL DE CREATION ET MISE A JOUR DE PROJET POUR LE PROPRIETAIRE	31
7.7	MODAL DE VISUALISATION DES DETAILS DE PROJET	32
7.8	MODAL DE CHAT DES PROJETS	33
7.9	ÉDITEUR DE DIAGRAMMES.....	34
8	CONCLUSION	37
8.1	ÉTAT ACTUEL DU PROJET	37
8.2	PROBLEMES RENCONTRES.....	37
8.3	AMELIORATIONS POSSIBLES	38
8.4	POUR ALLER PLUS LOIN	40
8.5	TESTS DE L’APPLICATION	40
8.6	CONCLUSION PERSONNELLE	40
8.7	REMERCIEMENT	40
9	TABLE DES ILLUSTRATIONS	41
10	ANNEXES.....	41
11	BIBLIOGRAPHIE.....	42

1 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.

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 bibliothèques 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 codées 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
- Sérialisation/Désérialisation 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
- 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

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 :

- Les choix de conception
- Les problèmes rencontrés
- Les solutions envisagées
- 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 a une représentation bien particulière qui sera décrite 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.

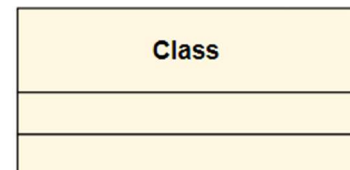


Figure 1 Représentation d'une classe

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.

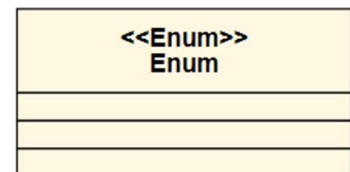


Figure 2 Représentation d'une énumération

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.

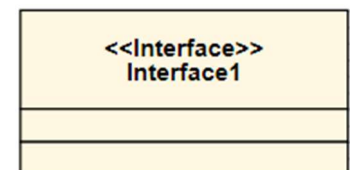


Figure 3 Représentation d'une interface

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

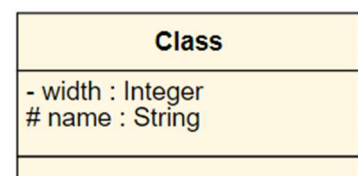


Figure 4 Représentation d'attributs

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 / Implémentation

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

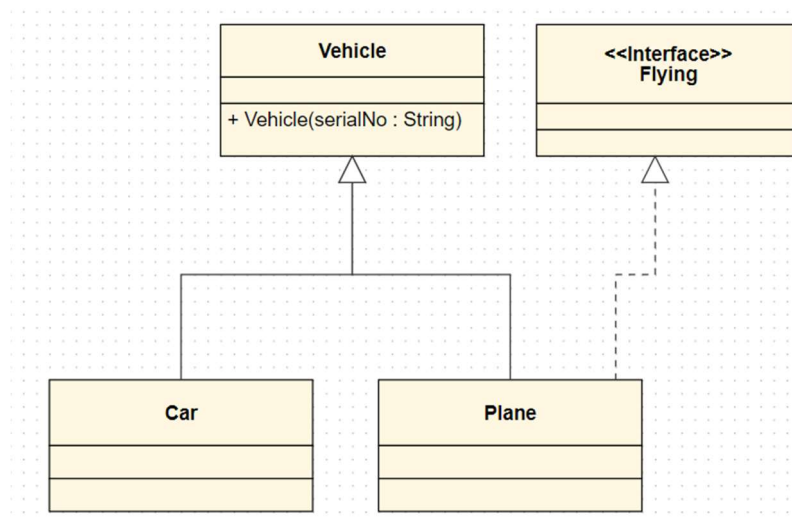


Figure 6 Représentation des héritages

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

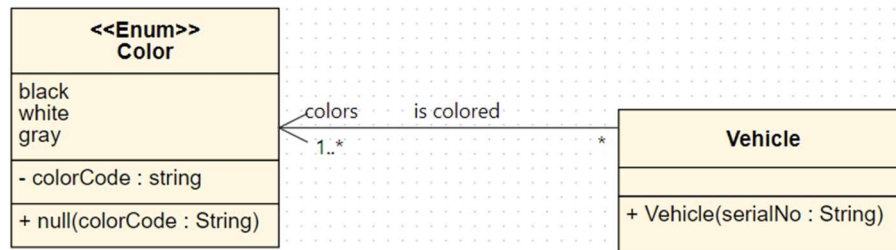


Figure 9 Représentation d'une association

2.2.4.2 Agrégation



Figure 10 Représentation d'une agrégation

2.2.4.3 Composition



Figure 11 Représentation d'une composition

2.2.4.4 Les associations multiples

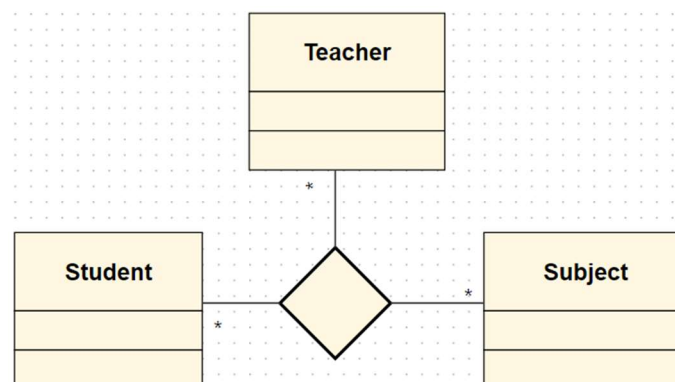


Figure 12 Représentation d'une association multiple

2.2.4.5 Les classes d'associations

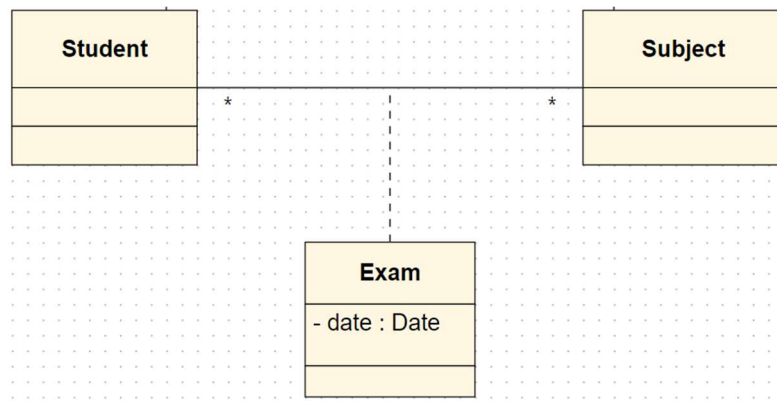


Figure 13 Représentation d'une classe d'association

3 Métaschéma de diagrammes de classes

3.1 Schéma complet

Le schéma complet est disponible en annexe.

3.2 Entités

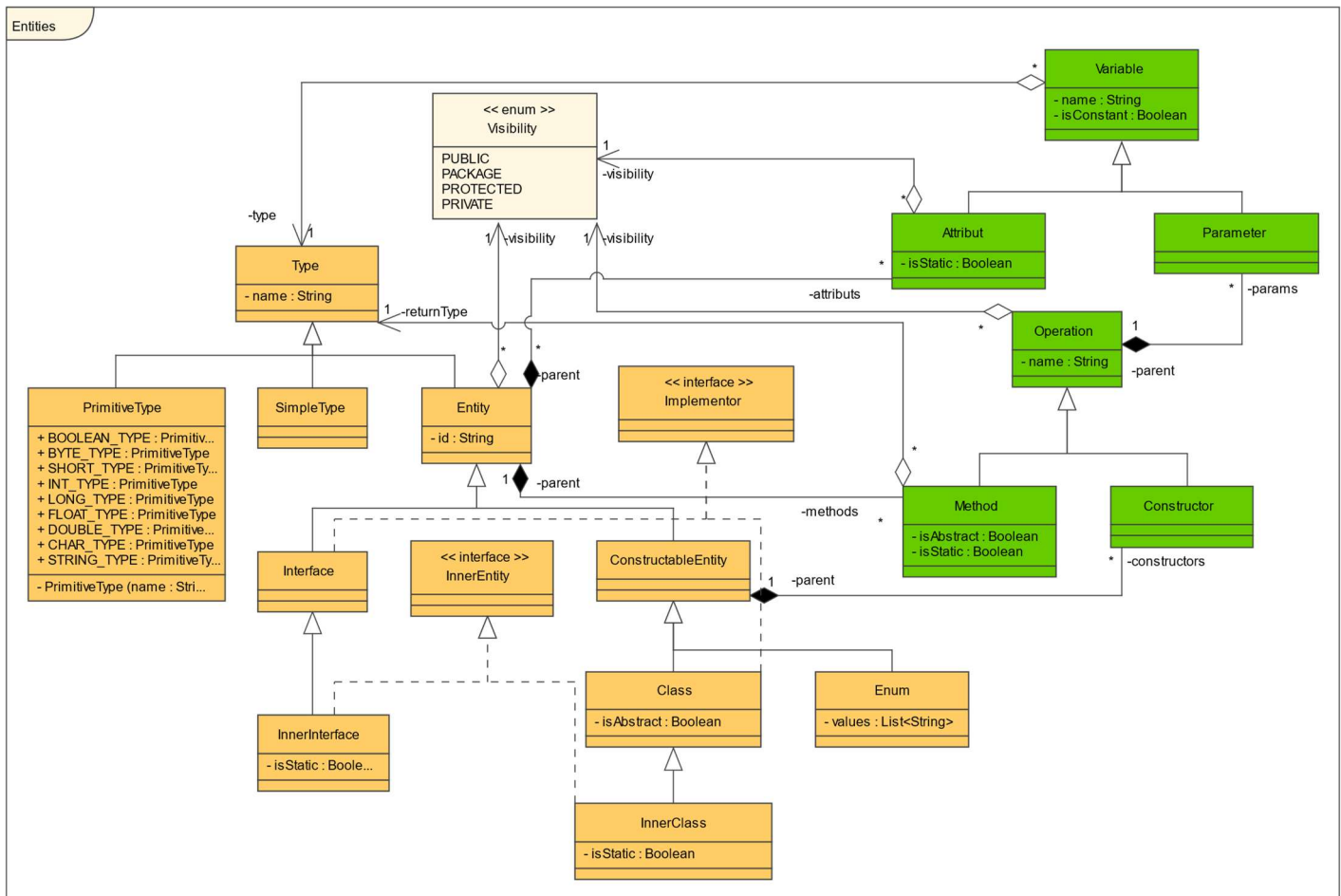


Figure 14 Diagramme de classes des entités

3.2.1 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.

3.3 Liens

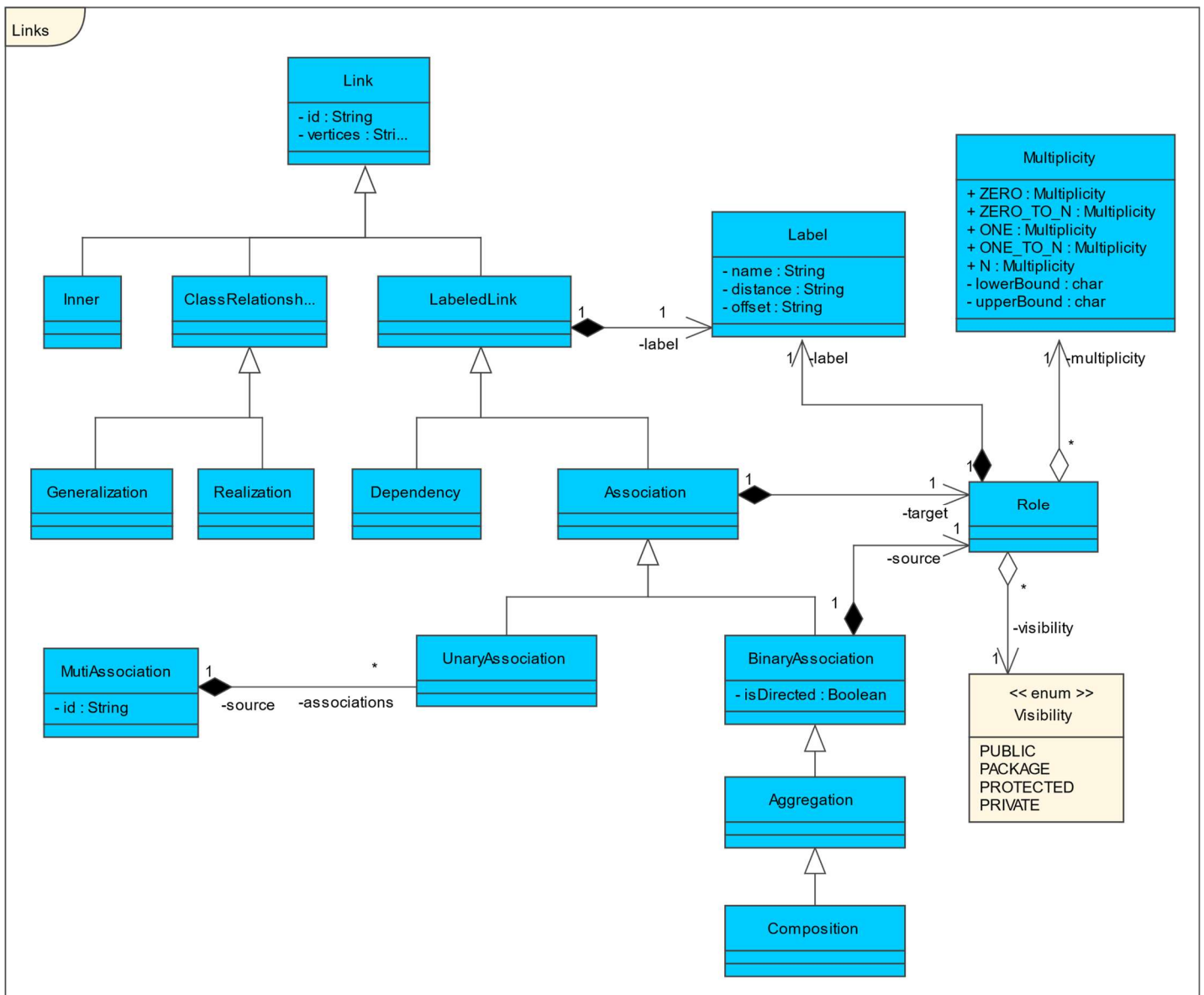


Figure 15 Diagramme de classes des liens

3.3.1 Description

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

Afin de faciliter la modification des multiassociations, 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.

3.4 Interactions liens – entités

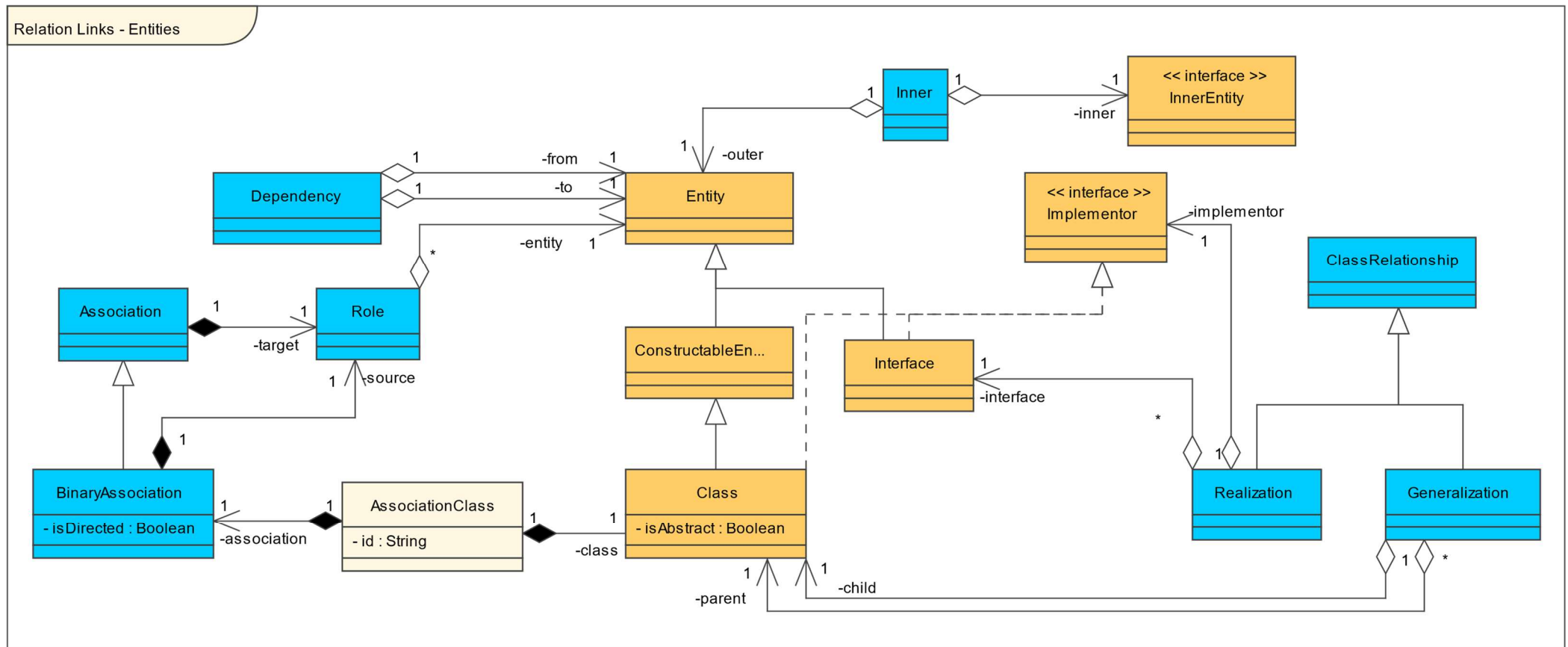


Figure 16 Diagramme de classes des interactions

3.4.1 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.

3.5 L'objet ClassDiagram

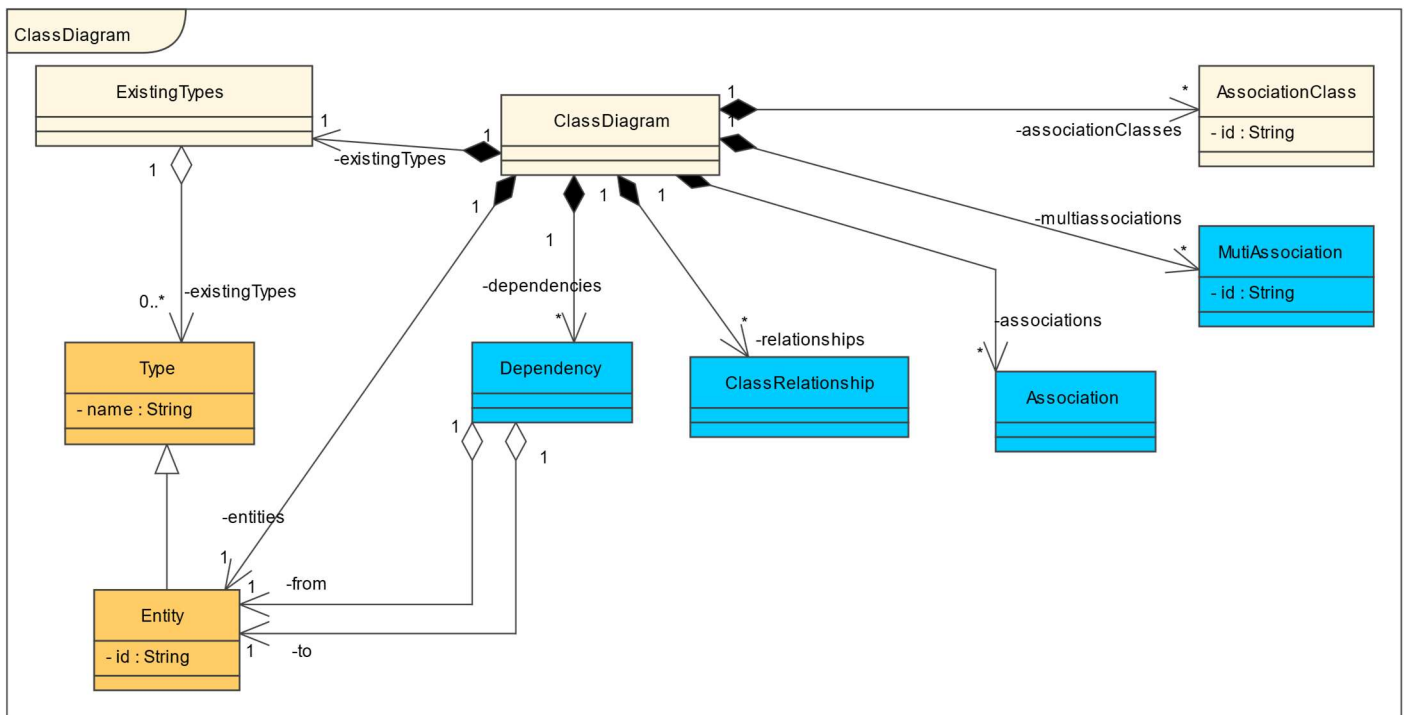


Figure 17 Diagramme de classes de l'élément racine

3.5.1 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.

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.

4 Architecture de l'application

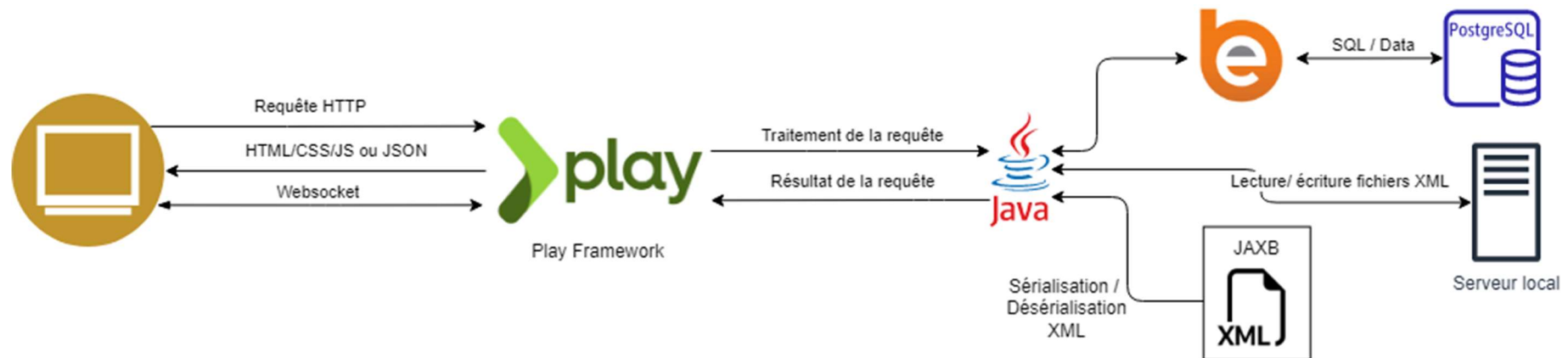


Figure 18 Architecture de l'application

4.1 Description globale

L'application est multitière. On y retrouve donc une couche présentation, une couche logique et une couche donnée.

La couche présentation comprend des pages HTML/CSS/JS créée 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.

4.1.1 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 DTO via un websocket.

Chaque DTO sera parsée en commande, qui vérifiera que les droits d'exécution sont bons, s'exécutera 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.

4.2 Framework Web

4.2.1 Utilité

L'utilisation d'un Framework Web permettrait de pouvoir simplifier l'utilisation d'Endpoint HTTP. En effet, il est possible d'avoir la notion de « route », qui est en fait une URL spécifique qui est liée par le Framework à une fonction. Ce qui signifie que si une requête arrive sur une certaine route, la fonction associée sera exécutée. Il met également à disposition des dispositifs de sécurité, tels que la gestion des « Cross-Site Request Forgery »¹.

4.2.2 Analyse de l'existant

J'ai tout d'abord effectué une recherche des Frameworks existants me permettant de travailler sur le Web et qui fonctionne avec Java. Je n'ai pas recherché à approfondir tous les Frameworks existants, simplement les plus connus et utilisés.

4.2.2.1 Spring

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

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

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

4.2.2.2 Play

Play ! permet également de définir des routes accessibles en HTTP, d'effectuer le traitement en Java et de rendre la réponse sous forme souhaitée (page HTML, JSON, XML ...).

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

4.2.3 Choix effectué

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

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

4.2.4 Mise en place

Play propose un fichier « routes » dans le dossier « conf ». Ce fichier répertorie la liste d'URL associées aux fonctions des divers contrôleurs.

```
# Application Controller
GET / controllers.ApplicationController.index(request : Request)
```

Figure 19 Exemple de routes avec Play

¹ **Mozilla Foundation. 2022.** CSRF. *developper.mozilla.* [En ligne] 28 05 2022. <https://developer.mozilla.org/fr/docs/Glossary/CSRF>.

Cette route lie donc l'URL "{hostname}/" à la fonction index du contrôleur *ApplicationController*.

Il est également possible d'avoir des routes prenant en compte des paramètres.

```
# Project Controller
GET /projects/:uuid controllers.ProjectController.project(request: Request, uuid : String)
```

Figure 20 Exemple de route Play avec paramètres

Pour la gestion de sécurité des routes, il est possible d'annoter les fonctions, par exemple pour autoriser uniquement les utilisateurs connectés d'accéder à la fonction.

```
// DELETE
@Security.Authenticated(Secured.class)
public Result delete(Http.Request request, String id)
{
```

Figure 21 Annotation Play pour la sécurité

4.3 Stockage de données

4.3.1 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.

J'ai créé un container Docker pour la base de données. De ce fait, il suffit de lancer la commande docker-compose afin d'avoir une base de données PostgreSQL prête.

4.3.2 ORM

4.3.2.1 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 mises en place :

```
public Project findById(UUID id)
{
    return supplyAsync(() -> DB.find(Project.class) Query<Project>
        .where() ExpressionList<Project>
        .eq( propertyName: "id", id)
        .findOne(), executionContext).join();
}
```

Figure 22 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.

4.3.2.2 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 de 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.

4.3.2.3 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.

4.3.2.4 Mise en place

Les *entities* seront les classes qui seront représentées comme des tables dans la base de données. Il est nécessaire de les annoter avec certains mots clés 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 :

```
@Entity
public class Project extends Model
{
    @Id
    public UUID id;

    @NotNull
    public String name;

    @Transient
    public ClassDiagram diagram = new ClassDiagram();

    @OneToMany(mappedBy = "project", cascade = CascadeType.ALL)
    public List<ProjectUser> projectUsers;
```

Figure 23 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 à ce 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.²

4.3.3 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.

Les fichiers XML sont directement sauvegardés sur le serveur.

Cependant, il pourrait être intéressant de mettre ce 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.

4.3.3.1 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.

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 :

² **Ebean ORM. 2022.** Ebean. *Ebean*. [En ligne] 24 04 2022. <https://ebean.io/docs/mapping/>

```
@XmlSeeAlso({ConstructableEntity.class, Interface.class})
public abstract class Entity extends Type
{

    @XmlID
    @XmlAttribute
    public String getId()
    {
        return id;
    }

    @XmlAttribute
    public int getX()
    {
        return x;
    }

    @XmlAttribute
    public int getY()
    {
        return y;
    }

    @XmlElement(name="attribute")
    public ArrayList<Attribute> getAttributes()
    {
        return attributes;
    }

    private String id;
```

Figure 24 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 possible de référencer 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 :

```
<entity xsi:type="Class" abstract="false"
  @XMLType height="100" width="250"
  id="93e3d3a8-ee37-1004-80b0-2f52c40aa3ec"
  visibility="PUBLIC"
  @XMLAttribute x="260" y="80" name="Car"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <attribute static="true" @XMLElement
    visibility="PRIVATE"
    constant="true"
    id="07b181cc-ee38-1004-80b0-2f52c40aa3ec"
    name="attribute"
    type="string"/>
</entity>
```

Figure 25 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 est eux aussi des éléments XML.

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.

Bootstrap³ permet d'ajouter des classes à des éléments, qui ont chacune un style et des fonctions associées. Dès lors, les éléments HTML deviennent responsive, ont des animations ou d'autres genres d'améliorations visuelles.

5.2 Design de diagramme de classes

5.2.1 Utilité

Dans l'optique de ne pas réinviter la roue et de ne pas perdre de temps, une librairie graphique pouvant gérer des éléments déplaçable, cliquable ou ayant d'autres événements me paraît être la bonne solution.

La solution est donc de trouver une librairie permettant ceci, afin de la mettre en place dans le projet et éventuellement rajouter certaines fonctionnalités si elle ne les possède pas.

5.2.2 Choix de la librairie

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.

JointJS⁴ 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.

5.2.3 Mise en place

JointJS fonctionne sur un système de *Graph* et de *Paper*. Les *Graph* contiennent l'entièreté des données, alors que les *Paper* sont pour l'affichage. Chaque élément a donc une vue bien à lui, qui permet le rendement graphique sur un Paper. Chaque vue possède un nombre d'événements, comme un clic de souris, ou un mouvement de l'élément en question.

Dans le cadre de ce projet, j'ai donc créé les éléments correspondant à chaque élément présent sur le diagramme de classes présenté plus haut.

³ **Bootstrap. 2022.** Bootstrap. *getbootstrap*. [En ligne] 10 04 2022. <https://getbootstrap.com/>.

⁴ **client.IO. 2022.** JointJS opensource. *JointJS*. [En ligne] 21 05 2022. <https://www.jointjs.com/opensource>.


```

let Class = basic_mjs.Generic.define('uml.Class', {
  attrs: {...}, Attributs
  name: [],
  attributes: [],
  methods: [],
  constructors: [],
  id: undefined,
  height: undefined,
  alreadyDeleted: false,
  visibility: 'public',
  toolbox: undefined,
}, {
  markup: [
    '<g class="rotatable">', Élémt
    '<g class="scalable">', d'affichag
    '<rect class="uml-class-name-rect"/>',
    '<rect class="uml-class-values-rect"/>',
    '<rect class="uml-class-attrs-rect"/>',
    '<rect class="uml-class-methods-rect"/>',
    '</g>',
    '<text class="uml-class-name-text"/>',
    '<text class="uml-class-values-text"/>',
    '<text class="uml-class-attrs-text"/>',
    '<text class="uml-class-methods-text"/>',
    '</g>'
  ].join(''),
  initialize: function () {...}, Foncti
  selected: function (flag) {...},

```

Figure 26 Exemple d'entité JointJS

En prenant cet exemple, on peut voir qu'une entité JointJS se caractérise par des attributs, un markup SVG pour l'affichage, et des fonctions.

Je me suis basé sur le plug-in⁵ UML mis à disposition par la librairie et je l'ai modifié afin d'arriver au résultat attendu.

La gestion des événements se fait en grande partie au niveau du Paper, cependant il peut arriver que certains événements soient à rajouter sur les éléments.

```

paper.on('element:mouseleave', function (elementView) {
  if (elementView.model !== selectedElement && !selectMultiAssociationMode) {
    paper.removeTools();
  }
});

```

Figure 27 Exemple d'événement sur un élément

⁵ client.IO. 2022. Plugin UML JointJS. *GitHub*. [En ligne] 23 05 2022. <https://github.com/clientIO/joint/blob/master/dist/joint.shapes.uml.js>.

6 Gestion de la concurrence

6.1 Mécanisme en place

La modification d'un élément envoie uniquement le champ modifié. De telle manière que si plusieurs personnes travaillent sur même élément, mais pas le même attribut, aucun conflit n'aura lieu.

6.2 Problème détecté

Après avoir utilisé l'application en collaboration avec des amis pour tester la concurrence, nous sommes arrivés à la conclusion que les problèmes de concurrences ne sont pas gênants dans la plupart des cas.

Le cas le plus gênant est la modification d'attribut, car c'est l'interface qui est régénérée entièrement, mais la donnée reçue pourrait être simplement mise à jour. Ce n'est donc pas un problème de concurrence, mais plutôt de gestion d'interface.

Un problème survient lorsque plusieurs utilisateurs éditent en même temps un même champ. Cela paraît évident, mais il est nécessaire de le citer.

6.3 Améliorations possibles

Il est évident que régler le problème paraît être une priorité. Cependant, il n'existe pas de réelle solution pour le deuxième problème. On pourrait imaginer que l'on voit ce que les autres utilisateurs sont en train de faire au moyen de surlignement. Cela permettrait donc de ne pas éditer en même temps le même champ.

7 Interface utilisateur

7.1 Page d'accueil pour utilisateur non connecté

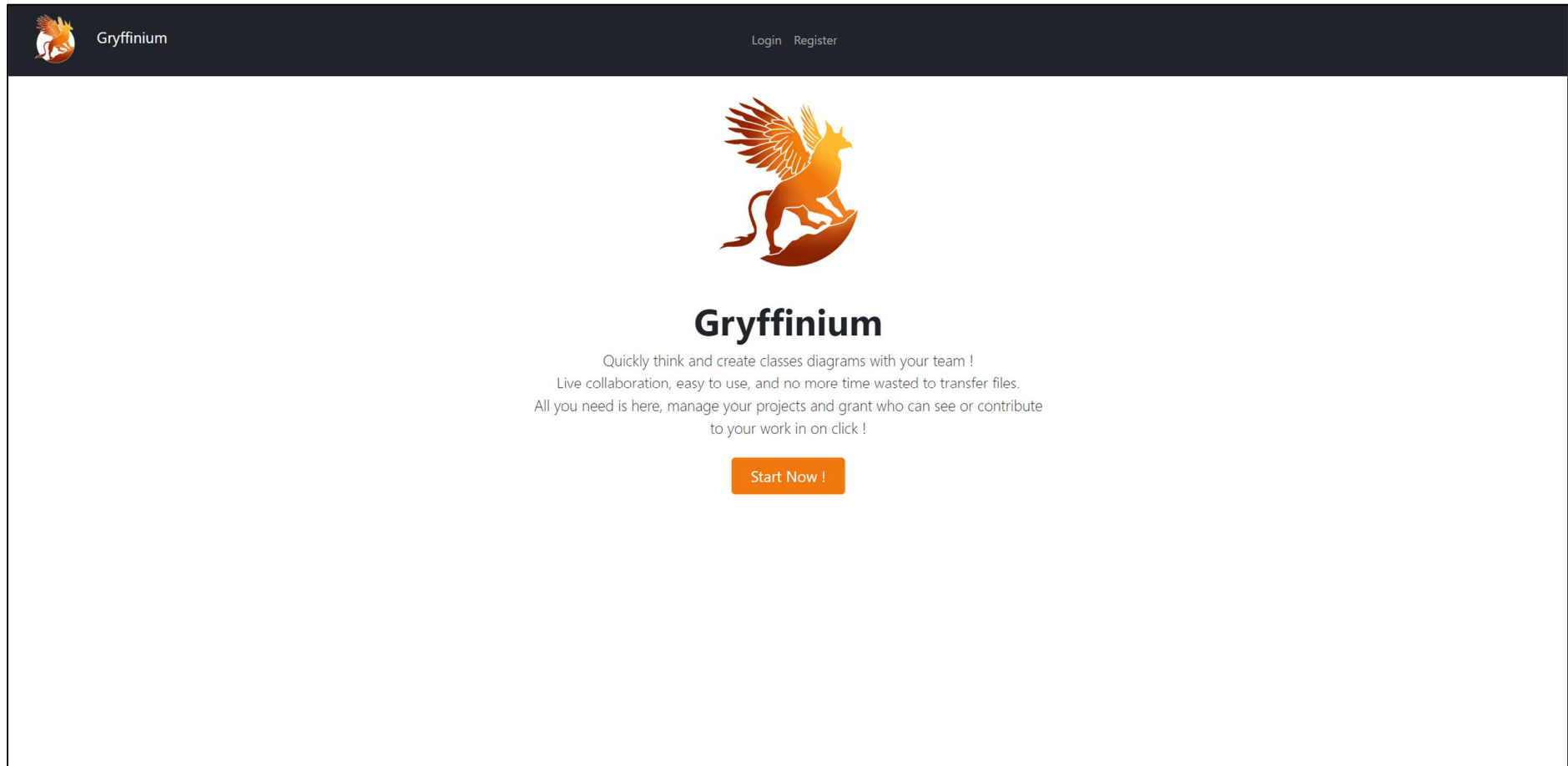


Figure 28 : Page d'accueil pour utilisateur non connecté

7.2 Page d'accueil pour utilisateur connecté

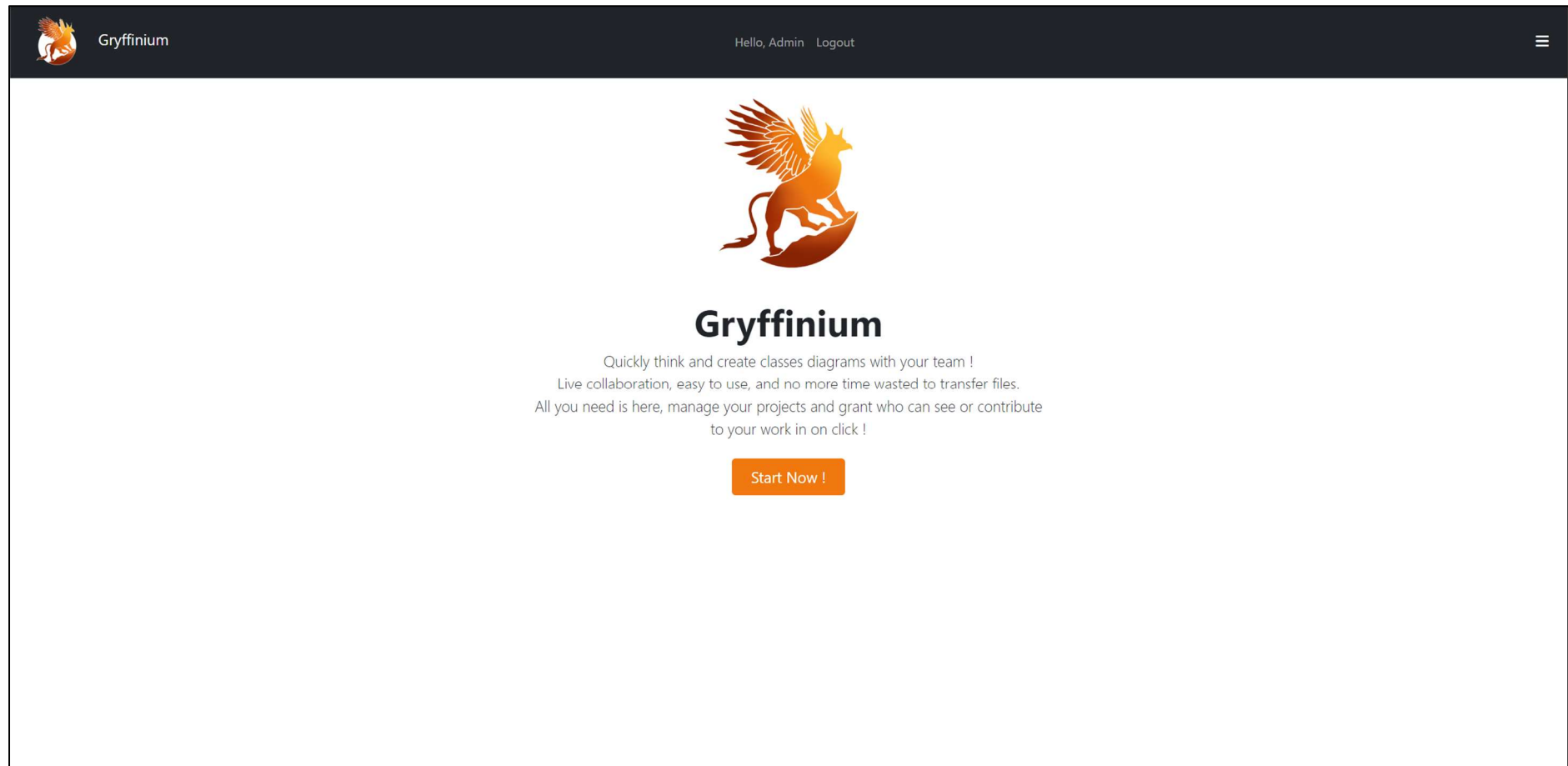
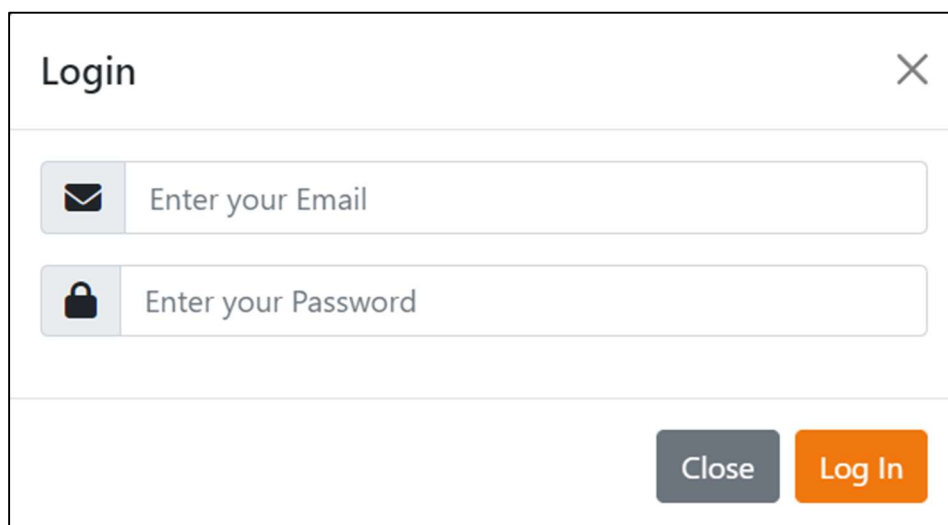


Figure 29 : Page d'accueil pour utilisateur connecté

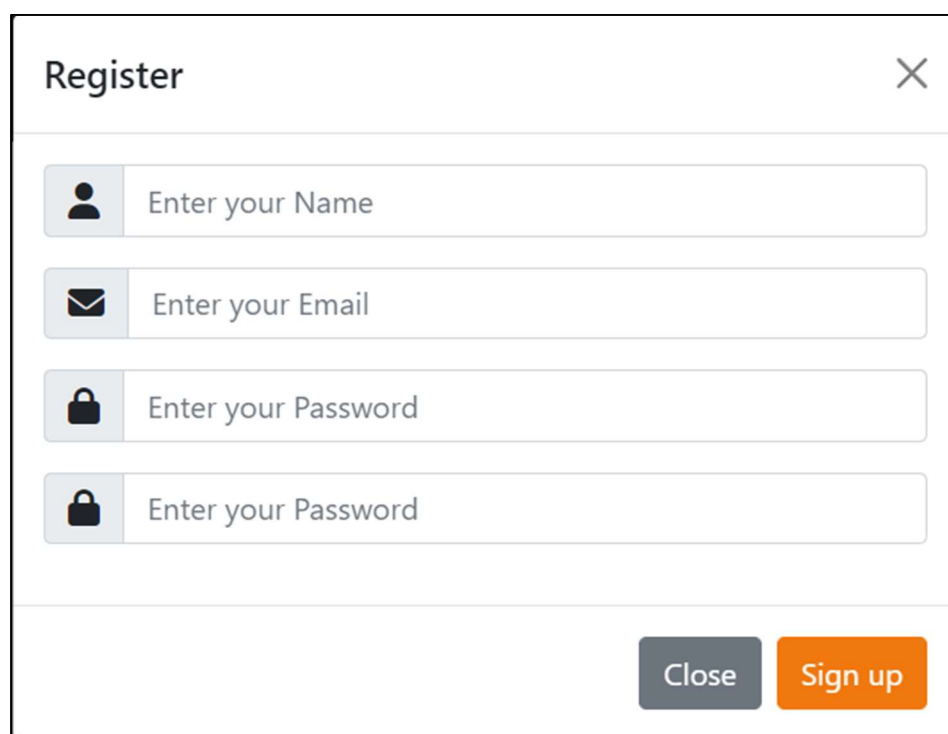
7.3 Modal d'authentification



A login modal window titled "Login" with a close button (X) in the top right corner. It contains two input fields: "Enter your Email" with an envelope icon and "Enter your Password" with a padlock icon. At the bottom right, there are two buttons: "Close" (grey) and "Log In" (orange).

Figure 30 : Modal d'authentification

7.4 Modal d'inscription



A register modal window titled "Register" with a close button (X) in the top right corner. It contains four input fields: "Enter your Name" with a person icon, "Enter your Email" with an envelope icon, and two "Enter your Password" fields with padlock icons. At the bottom right, there are two buttons: "Close" (grey) and "Sign up" (orange).

Figure 31 : Modal d'inscription

7.5 Page d'accueil avec la liste des projets

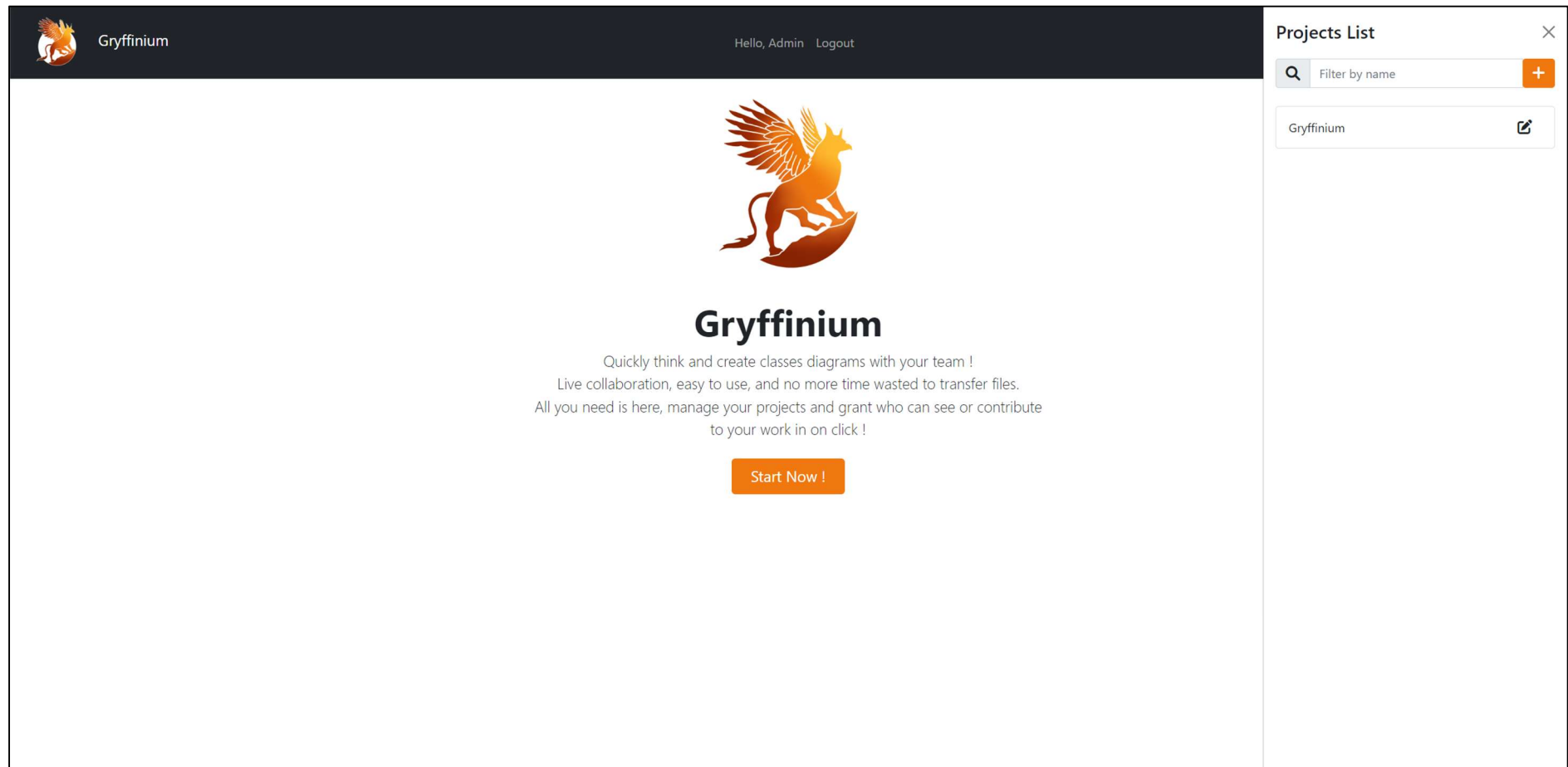


Figure 32 : Page d'accueil avec la liste des projets

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

Project informations

Name

Gryffinium

Collaborators

Invite a collaborator

+

admin@admin.ch

quentin@forestier.ch

Close

Delete

Figure 33 : Modal de création et mise à jour de projet pour le propriétaire

7.7 Modal de visualisation des détails de projet

Project informations

Name

Gryffinium

Collaborators

admin@admin.ch

quentin@forestier.ch

Close

Figure 34 : Modal de visualisation des détails de projet

7.8 Modal de chat des projets



Figure 35 : Modal de chat des projets

7.9 Éditeur de diagrammes

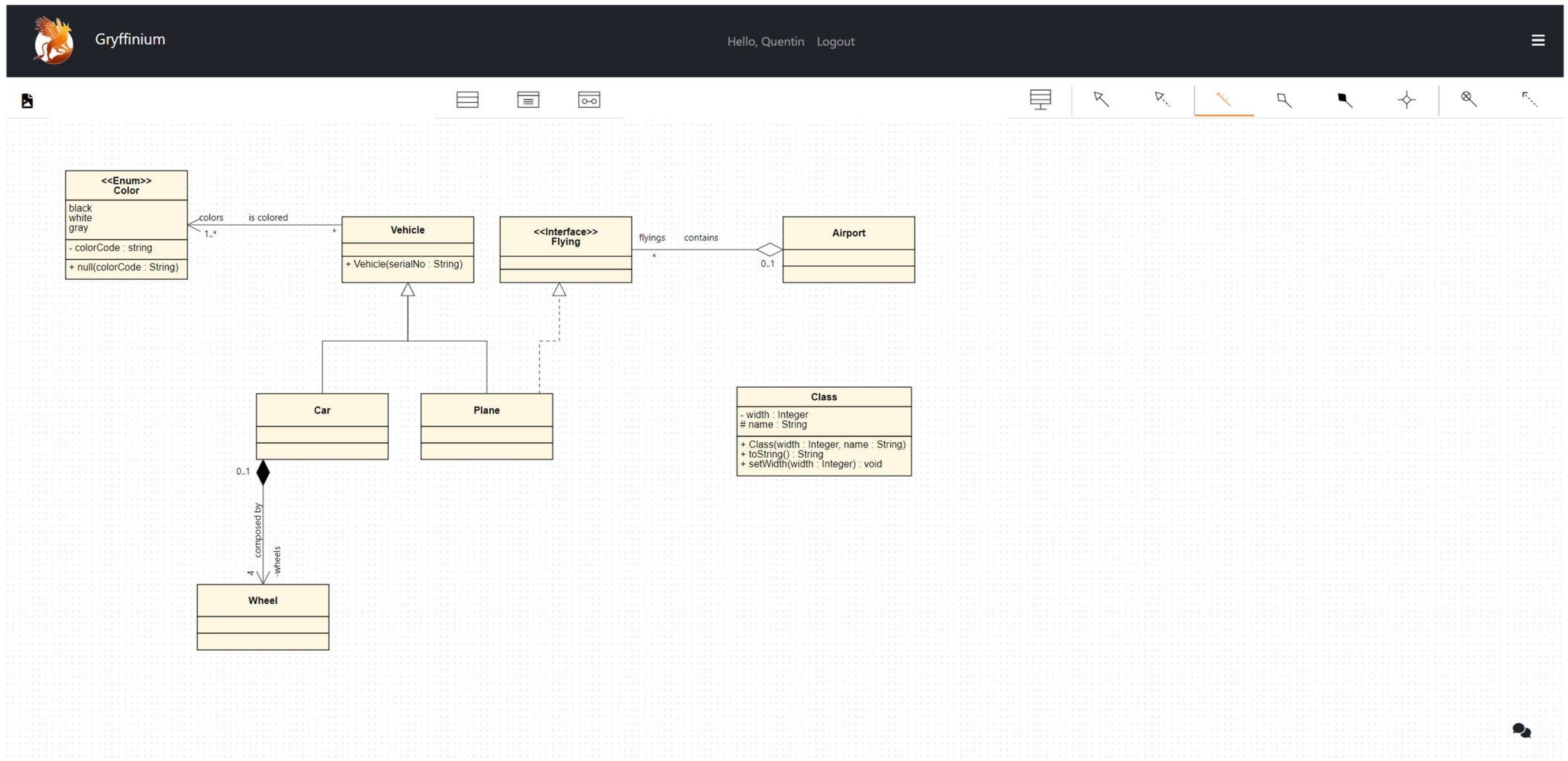


Figure 36 Interface de l'éditeur de diagrammes

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.

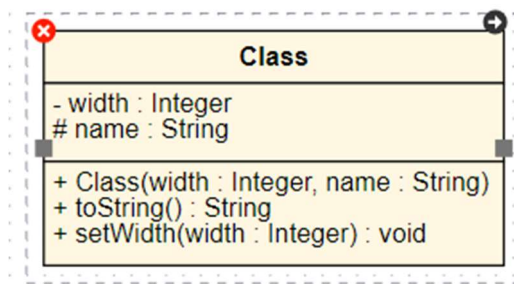


Figure 37 Toolbox d'une entité

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

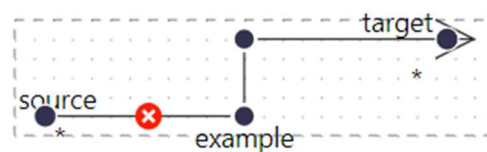









Figure 38 Toolbox d'un lien

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.
	Sélectionne la généralisation comme lien créé via la Toolbox. La classe d'association sera créée une fois que le lien est en place.
	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
	Active le mode « Sélection » qui permet à l'utilisateur de cliquer sur les entités à liées.
	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 État actuel du projet

Dans l'état actuel, le projet permet de :

- S'inscrire et se connecter
- Créer / Modifier / Supprimer des projets
- Ajouter / Modifier / Supprimer des utilisateurs sur les projets, et leurs droits
- Créer / Modifier / Supprimer tous les éléments UML d'un diagramme de classes
- Collaborer en instantané avec les autres membres du projet
- Envoyer des messages aux autres membres du projet
- D'exporter un diagramme sous format PNG et SVG
- De sauvegarder un diagramme dans un fichier XML
- De charger un diagramme sauvegardé dans un fichier XML

Certaines fonctionnalités supplémentaires sont également en place :

- Changement de nom d'une classe/enum/interface change également le nom des éléments leur faisant référence
- Ajustement de la taille automatique des éléments
- Possibilité de se déplacer sur le diagramme, qui est de taille infinie
- Possibilité de zoomer sur le diagramme

Le projet ne permet pas de :

- Modifier les éléments directement sur le diagramme
- D'annuler une action effectuée
- D'avoir différentes vues ayant les mêmes entités
- D'utiliser des raccourcis clavier pour être plus efficace
- De dupliquer une entité

8.2 Problèmes rencontrés

8.2.1 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.

8.2.1.1 Résolution

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

J'ai compris que l'on pouvait passer en paramètre un objet lors de la création d'un Actor. Une fois cela fait, j'ai pu créer les sessions différentes pour chaque projet avec simplicité.

8.2.2 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 contraint à stocker le diagramme de classes sous forme de XML.

8.2.2.1 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 de XML, afin que les utilisateurs puissent le télécharger s'ils le souhaitent.

8.2.3 Compréhension plus approfondie de la librairie JointJS

La documentation était en surface très bonne, cependant il manquait énormément de documentation sur tous les événements existants. Par exemple, la gestion des points qu'il est possible d'ajouter le long d'un lien n'est pas documentée.

8.2.3.1 Résolution

Afin de pallier à ce problème, je suis allé rechercher dans le code source les différentes possibilités. Souvent, les événements n'étaient simplement pas référencés dans la documentation, mais existaient bel et bien. Cela a été une grande perte de temps, car le code source était très complexe et difficile à lire.

8.2.4 Mise en place du projet dans un docker

Je n'ai pas réussi à mettre le projet dans un container Docker. En effet, obtenir les sources et exécuter comme indiqué dans la documentation⁶ à générer un problème que je n'ai malheureusement pas eu le temps de résoudre.

8.3 Améliorations possibles

8.3.1 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.

8.3.2 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.

8.3.3 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.

8.3.4 Confirmation d'action irréversible

Afin de favoriser l'expérience utilisateur, il serait bien d'avoir un 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.

8.3.5 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.

8.3.6 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.

⁶ **LightBend.** 2022. Play Framework - Deploying. *Play Framework*. [En ligne] 25 07 2022. <https://www.playframework.com/documentation/2.8.x/Deploying>.

8.3.7 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 possible⁷.

8.3.8 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.

8.3.9 Réimaginer la partie édition des éléments du diagramme

Idéalement, il faudrait que toutes les modifications soient faites sur le diagramme, même pour les attributs tels que « constant » ou « static ». S'il était possible de s'abstraire entièrement de l'interface qui s'affiche lors d'un clic sur un élément, cela serait bien plus ergonomique et simple d'utilisation.

8.3.10 Refonte du code JavaScript

Le déplacement dans le canevas JointJS n'est pas possible nativement. Il existe cependant dans la version payante de la librairie. J'ai donc réussi à l'implémenter à ma manière, cependant il serait judicieux séparer ce genre de fonction dans un objet qui pourrait avoir ses propres événements, afin de séparer les tâches et rendre le code plus compréhensible.

8.3.11 Refonte des éléments « structurels » JointJS

Les éléments structurels proposés par le plug-in UML de JointJS ont été d'une grande aide, cependant ils sont très limités et « fouillis ». Ils permettent difficilement une évolutivité, et le projet mériterait donc que cela soit refait complètement.

Si ces éléments structurels étaient mieux faits à la base, il serait très facile d'ajouter l'édition sur le schéma.

8.3.12 Implémentation Backend des classes d'association

La classe d'association est un objet composé d'une association et d'une classe. Si un de ces 2 éléments est supprimé, l'objet *AssociationClass* ne l'est pas. À cause de la contrainte temps, je n'avais pas le temps de gérer la différence au niveau de l'affichage, et donc j'ai préféré que les *AssociationClass* soient présentes, bien que pas optimale.

8.3.13 Meilleure gestion de la concurrence

Il serait possible de mettre à jour les éléments modifiés à chaque entrée utilisateur, de ce fait, si une donnée est écrasée, ce ne serait qu'un unique caractère. Il faudrait également donner un feedback visuel sur quel membre du projet édite quel élément.

⁷ **Bruckner, Roman. 2022.** Exemple d'élément JointJS avec champs HTML. *JS Fiddle*. [En ligne] 17 07 2022. <https://jsfiddle.net/kumilingus/rpw5u8mq/>.

8.4 Pour aller plus loin

8.4.1 Exportations des fichiers

Une amélioration conséquente, mais plus que bienvenue serait de générer automatiquement les fichiers dans le langage souhaité. De cette manière, il ne serait plus utile de réécrire l'entièreté des attributs et méthodes des classes.

8.4.2 Possibilité de créer d'autres diagrammes UML

Les diagrammes de classes n'étant pas les seuls diagrammes UML, il serait intéressant d'étendre le projet à tous les types de diagrammes existants. Il faudrait cependant garder à l'esprit la simplicité d'édition du diagramme, afin de ne pas perdre l'utilisateur dans une multitude de menus et d'éléments.

8.5 Tests de l'application

8.5.1 Tests automatisés

Une batterie de tests automatisée a été mise en place à l'aide de JUnit. Ces tests ont pour but de tester les différentes routes en envoyant une requête HTTP et en vérifiant la réponse.

Ces tests couvrent également les différentes commandes de l'application, permettant ainsi de vérifier que les interactions effectuées au travers du Websocket fonctionnent bien.

Pour plus d'informations sur les tests, voir le rapport des tests en annexe.

8.6 Conclusion personnelle

Je suis content du travail que j'ai pu effectuer. J'ai pu utiliser un grand nombre de compétences que j'ai acquises durant ma formation. La partie serveur est d'après bien faite et compréhensible. La partie client est cependant mon regret, étant donné les fonctionnalités manquantes. Ma plus grande frustration étant de ne pas avoir réussi à rendre le code JavaScript structuré. Le fait d'apprendre la librairie, les websockets, la gestion d'événements en même temps ne m'a pas permis de prendre suffisamment de recul pour imaginer les choses au mieux.

8.7 Remerciement

Je tiens à remercier Jérôme Varani qui m'a bien aidé dans ma découverte de Play. Étant donné son expérience, il m'a aidé à me poser les bonnes questions et à m'aiguiller dans mes priorités.

Je tiens également à remercier Magalie Bouvier pour le logo, icône qui symbolise mon travail de Bachelor.

Je remercie Guillaume Forestier pour les icônes qui représentent les entités UML sur la page d'éditions de diagrammes.

Et je remercie grandement Pier Donini de m'avoir permis d'effectuer mon travail de Bachelor sur ce sujet, qui m'a passionné et que j'espère pouvoir continuer.

9 Table des illustrations

Figure 1 Représentation d'une classe.....	9
Figure 2 Représentation d'une énumération	9
Figure 3 Représentation d'une interface.....	9
Figure 4 Représentation d'attributs	9
Figure 5 Représentation des opérations	10
Figure 6 Représentation des héritages.....	10
Figure 7 Représentation des dépendances	10
Figure 8 Représentation du lien de classe interne	10
Figure 9 Représentation d'une association	11
Figure 10 Représentation d'une agrégation.....	11
Figure 11 Représentation d'une composition	11
Figure 12 Représentation d'une association multiple.....	11
Figure 13 Représentation d'une classe d'association.....	12
Figure 14 Diagramme de classes des entités.....	13
Figure 15 Diagramme de classes des liens	14
Figure 16 Diagramme de classes des interactions.....	15
Figure 17 Diagramme de classes de l'élément racine	16
Figure 18 Architecture de l'application	17
Figure 19 Exemple de routes avec Play	18
Figure 20 Exemple de route Play avec paramètres	19
Figure 21 Annotation Play pour la sécurité	19
Figure 22 Exemple d'utilisation d'un ORM	19
Figure 23 Mise en place d'une entité Ebean	21
Figure 24 Exemple d'annotations pour JAXB.....	22
Figure 25 Exemple de résultat de sérialisation.....	23
Figure 26 Exemple d'entité JointJS	25
Figure 27 Exemple d'événement sur un élément.....	25
Figure 40 : Page d'accueil pour utilisateur non connecté	27
Figure 41 : Page d'accueil pour utilisateur connecté.....	28
Figure 42 : Modal d'authentification	29
Figure 43 : Modal d'inscription.....	29
Figure 44 : Page d'accueil avec la liste des projets.....	30
Figure 45 : Modal de création et mise à jour de projet pour le propriétaire.....	31
Figure 46 : Modal de visualisation des détails de projet	32
Figure 47 : Modal de chat des projets	33
Figure 48 Interface de l'éditeur de diagrammes	34
Figure 49 ToolBox d'une entité.....	35
Figure 50 ToolBox d'un lien	35

10 Annexes

- Journal de travail
- Documentation technique
- Source du projet
- Rapport de tests
- Diagramme de classes complet de l'application

11 Bibliographie

- Bootstrap. 2022.** Bootstrap. *getbootstrap*. [En ligne] 10 04 2022. <https://getbootstrap.com/>.
- Bruckner, Roman. 2022.** Exemple d'élément JointJS avec champs HTML. *JS Fiddle*. [En ligne] 17 07 2022. <https://jsfiddle.net/kumilingus/rpw5u8mq/>.
- client.IO. 2022.** Graph & Paper. *JointJS*. [En ligne] 28 05 2022. <https://resources.jointjs.com/tutorial/graph-and-paper>.
- . **2022.** JointJS opensource. *JointJS*. [En ligne] 21 05 2022. <https://www.jointjs.com/opensource>.
- . **2022.** Plugin UML JointJS. *GitHub*. [En ligne] 23 05 2022. <https://github.com/clientIO/joint/blob/master/dist/joint.shapes.uml.js>.
- dave. 2022.** How to make a papper draggable. *Stackoverflow*. [En ligne] 1 06 2022. <https://stackoverflow.com/questions/28431384/how-to-make-a-paper-draggable>.
- Doughan, Blaise. 2022.** JAXB and Inheritance - Using XmlAdapter. *DZone*. [En ligne] 06 18 2022. <https://dzone.com/articles/jaxb-and-inheritance-using>.
- Ebean ORM. 2022.** Ebean. *Ebean*. [En ligne] 24 04 2022. <https://ebean.io/docs/mapping/>.
- JHarley1. 2022.** Les avantages et les inconvénients de l'utilisation de l'ORM. *askcodez*. [En ligne] 28 07 2022. <https://askcodez.com/les-avantages-et-les-inconvenients-de-lutilisation-de-lorm.html>.
- LightBend. 2022.** Play Framework - Deploying. *Play Framework*. [En ligne] 25 07 2022. <https://www.playframework.com/documentation/2.8.x/Deploying>.
- Mozilla Foundation. 2022.** CSRF. *developer.mozilla*. [En ligne] 28 05 2022. <https://developer.mozilla.org/fr/docs/Glossary/CSRF>.
- Northwoods Software. 2022.** UML Class Sample. *GoJS*. [En ligne] 26 04 2022. <https://gojs.net/latest/samples/umlClass.html>.
- Sadeh, Nimrod. 2022.** Building a Reactive, Distributed Messaging Server in Scala and Akka with WebSockets. *medium*. [En ligne] 26 04 2022. <https://medium.com/@nnnsadeh/building-a-reactive-distributed-messaging-server-in-scala-and-akka-with-websockets-c70440c494e3>.
- vladimir. 2022.** joint.shapes.basic.Table. *codepen*. [En ligne] 10 07 2022. <https://codepen.io/vladimirtalas/pen/MrymjB>.
- W3Schools. 2022.** How TO - Collapse Sidepanel. *w3schools*. [En ligne] 16 04 2022. https://www.w3schools.com/howto/howto_js_collapse_sidepanel.asp.