Quentin Forestier

TB – 06.05.2022

Rapport

Travail de bachelor



Table des matières

[Cahier des charges 3](#_Toc102722829)

[Problématique 3](#_Toc102722830)

[Solutions existantes 3](#_Toc102722831)

[Objectif 3](#_Toc102722832)

[Jalon 3](#_Toc102722833)

[Fonctionnalités de l’application 3](#_Toc102722834)

[Fonctionnalité principale du diagramme 3](#_Toc102722835)

[Fonctionnalité supplémentaire du diagramme 4](#_Toc102722836)

[Fonctionnalité de gestion 4](#_Toc102722837)

[Échéance 4](#_Toc102722838)

[Livrables 4](#_Toc102722839)

[Planning 4](#_Toc102722840)

[Métaschéma 6](#_Toc102722841)

[Description du métaschéma 7](#_Toc102722842)

[En orange 7](#_Toc102722843)

[En vert 7](#_Toc102722844)

[En bleu 7](#_Toc102722845)

[En blanc 7](#_Toc102722846)

[Framework 7](#_Toc102722847)

[Analyse de l’existant 7](#_Toc102722848)

[Spring 7](#_Toc102722849)

[Play ! 7](#_Toc102722850)

[Choix effectué 8](#_Toc102722851)

[Base de données 8](#_Toc102722852)

[ORM 8](#_Toc102722853)

[Analyse de l’existant 8](#_Toc102722854)

[Choix effectué 8](#_Toc102722855)

[Libraires graphiques 9](#_Toc102722856)

[Design de l’interface global 9](#_Toc102722857)

[Design de diagramme de classes 9](#_Toc102722858)

[Fonctionnement de l’application 9](#_Toc102722859)

[Login / Register 9](#_Toc102722860)

[Gestion de projets 9](#_Toc102722861)

[Collaboration sur les projets 9](#_Toc102722862)

# Cahier des charges

## Problématique

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

## Solutions existantes

* Slyum
* StarUML
* Umletino
* …

## Objectif

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

## Jalon

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

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

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

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

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

## Fonctionnalités de l’application

### Fonctionnalité principale du diagramme

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

### Fonctionnalité supplémentaire du diagramme

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

### Fonctionnalité de gestion

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

## Échéance

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

## Livrables

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

## Planning

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

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

# Métaschéma

## Description du métaschéma

### En orange

Toutes les représentations de types et entités possibles.

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

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

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

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

### En vert

Tous les attributs et méthodes disponibles dans les entités.

### En bleu

Tous les liens possibles d’un diagramme de classe.

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

### En blanc

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

# Framework

## Analyse de l’existant

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

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

### Spring

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

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

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

### Play !

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

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

## Choix effectué

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

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

# Base de données

## SGBD

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

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

# ORM

## Analyse de l’existant

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

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

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

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

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

## Choix effectué

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

# Libraires graphiques

## Design de l’interface global

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

## Design de diagramme de classes

A rechercher

# Fonctionnement de l’application

## Login / Register

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

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

### Sécurités et validations mises en place

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

Validation des entrées utilisateurs :

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

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

### Diagramme de séquences

#### Login

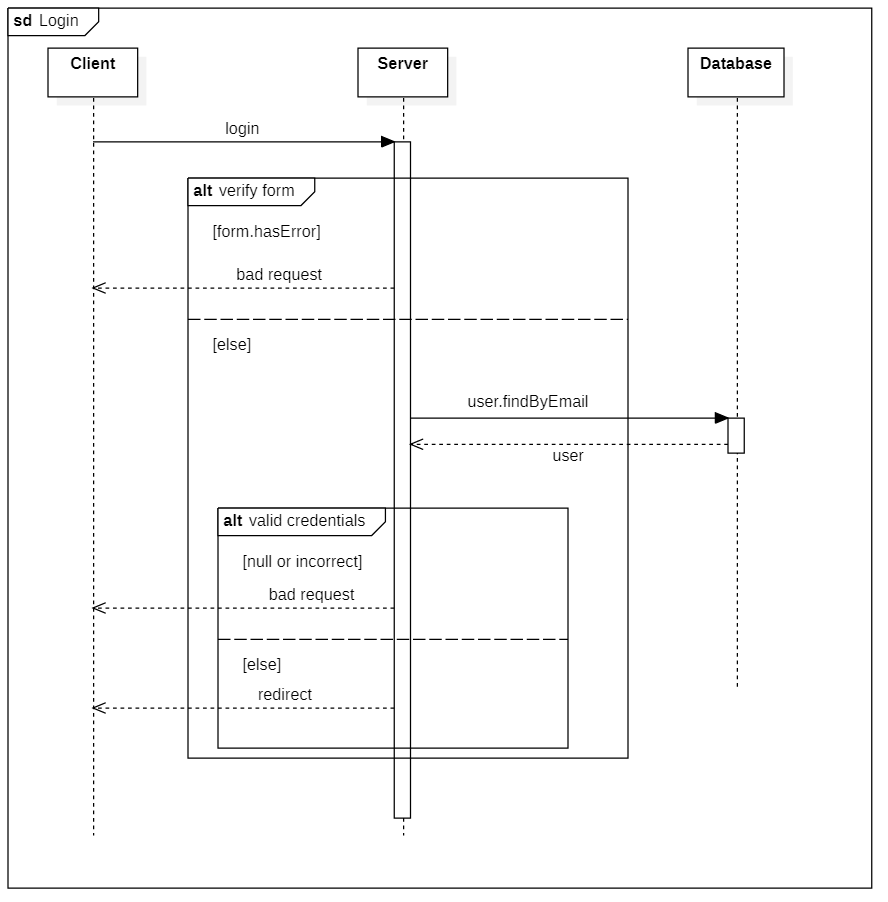


Figure : Diagramme de séquence Login

#### Register

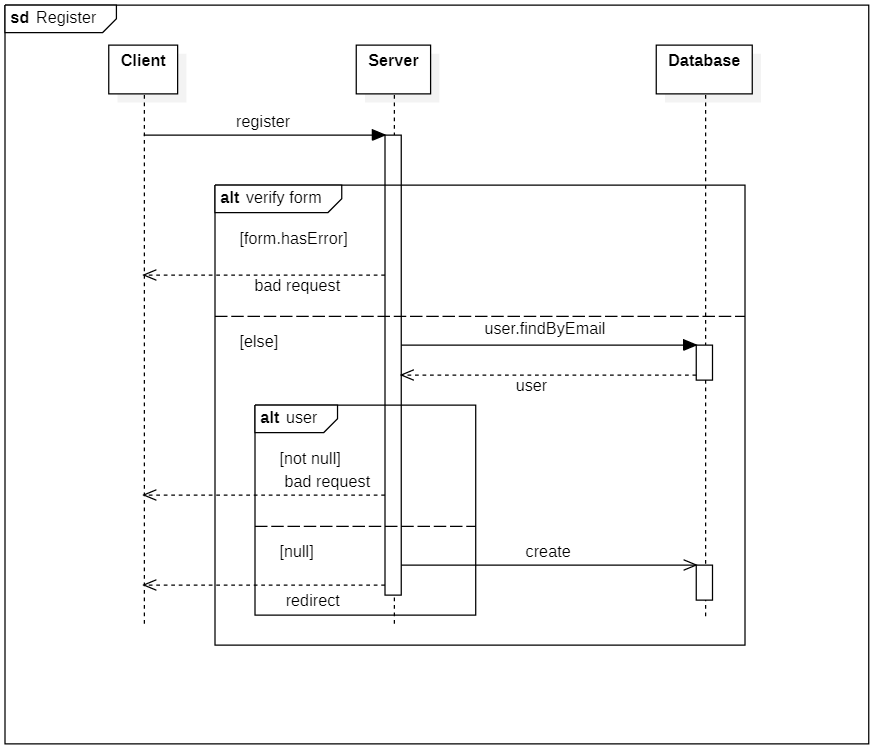


Figure : Diagramme de séquence Register

## Gestion de projets

Toutes les routes concernant les projets sont protégées. Il faut être authentifié afin d’avoir accès à la route.

### Projet

Afin de favoriser l’expérience utilisateur, toutes les interactions de gestion de projets sont faites en asynchrones.

#### Création

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.

### Collaborateurs

## Collaboration sur les projets

1. https://ebean.io/docs/mapping/ [↑](#footnote-ref-1)
2. https://www.mindrot.org/projects/jBCrypt/ [↑](#footnote-ref-2)