# Treemily

## Mémoire de projet

Henrik Akesson, juillet 2017

Encadré par : Prof. Pier Donini

Sujet proposé par : Prof. Pier Donini

# Contents

# 0. Remerciements

J'adresse mes remerciements aux personnes qui m'ont aidé dans la réalisation de ce projet.

En premier lieu, je souhaite remercier le Professeur Pier Donini pour m'avoir guidé dans mon travail, m'avoir aidé à me focaliser sur les aspects les plus essentiels et complexes du projet, et de manière générale pour avoir imposé une exigence et un rythme de travail stricts.

En second lieu, je souhaite remercier Jérôme Varani, ingénieur informaticien, pour m'avoir aidé à repérer et prioriser les tâches les plus difficiles du projet.

En troisième lieu, je souhaite remercier Anne-Catherine Akesson pour avoir relu et corrigé ce mémoire.

Enfin, je souhaite remercier Imane Louafa pour m'avoir apporté ses conseils et pour avoir trouvé un nom au projet.

# 1. Cahier des charges

## 1.1. Utilité

Le cahier des charges proposé au début du travail visait à décrire et prévoir la manière dont le projet serait développé, ainsi qu'à définir les modes d'utilisation, les critères d'acceptation et les fonctionnalités jugés essentiels pour considérer le logiciel comme accompli.

## 1.2. Description du projet

Le but du projet est de fournir à des utilisateurs une plateforme pour décrire et partager leur généalogie. Les utilisateurs pourront d'une part décrire leur arbre généalogique, les personnes impliquées et les évènements et média associés, et d'autre part les détails devront pouvoir être à visibilité variable, du privé au public en passant par la visibilité limitée.

D'un point de vue utilisateur, trois manières d'utiliser le logiciel seront possibles.

- Utilisateur non enregistré : rechercher une personne, voir l'arbre généalogique de cette dernière, ses évènements et les détails associés dans les limites de ce qui est visible publiquement.
- Utilisateur nouveau et inconnu : créer un arbre généalogique, y ajouter des personnes (créant au passage des profils fantômes dans le cas où une personne ajoutée n'est pas déjà enregistrée dans le logiciel), ajouter des évènements selon la visibilité souhaitée.
- Utilisateur nouveau mais dont un profil fantôme est déjà enregistré : associer le profil au nouveau compte (l'utilisateur créé un compte, saisit les informations de son profil et se voit proposer des profils qui pourraient être le sien), contribuer à l'arbre duquel il est membre, y ajouter des évènements et média.

## 1.3. Critères d'acceptation

Afin de considérer le projet comme achevé, les critères suivants doivent être respectés :

- Serveur :
  - La base de données est mise en place, la modélisation est étudiée, intuitive, propre, réutilisable et peut être développée et étendue selon de nouveaux cas d'utilisation.
  - Le serveur gère toutes les opérations CRUD associées aux cas d'utilisation et à la base de données
  - Le serveur est robuste et scalable.
  - Le serveur transmet au client toutes et seulement les données dont il a besoin selon tous les cas d'utilisation.
  - Le serveur gère les enregistrements et les connexions de manière sécurisée.
- Client :
  - Le client offre une interface graphique propre, moderne et intuitive.
  - Le client permet toutes les opérations nécessaires à l'utilisation complète du logiciel.
  - L'utilisateur est tenu au courant de l'état de ses opérations (succès, erreur, nouveaux évènements qui lui sont associés…).
  - Une section d'aide permet à un utilisateur d'apprendre les possibilités du logiciel et comment les exploiter.
  - L'application cliente permet de parcourir l'arbre d'une famille et d'avoir une vue détaillée des membres. La vue détaillée comprend les informations générales et une chronologie des évènements.

## 1.4. Fonctionnalités prévues

- Créer un compte
- Créer un arbre généalogique
    - Ajouter un profil, existant au préalable ou non (date de naissance, nom complet, sexe, lieu de naissance, adresse courante, anciennes adresses, date de décès, lieu de décès, informations de contact, photo de profil)
    - Ajouter des liens entre les profils (mariage, divorce, séparation, unions civiles, enfants biologiques, adoption, PACS, partenaire, fiançailles)
- Ajouter des évènements et leur associer des média (vidéo, photo, audio, lien, document) et profils associés
    - Les évènements peuvent être d'ordre personnel (privé, limité à certains profils, public).
    - Les évènements sont associés à un ou plusieurs profils et sont visibles sur la vue détaillée de chacun après validation.
    - La définition générale d'un évènement est libre. Il peut aussi bien s'agir d'un évènement « quelconque » comme des vacances, que d"un évènement important (changement de sexe, de nom, d'emploi, de lieu de vie, naissance, adoption).
    - Une modification du profil pourra être enregistrée comme un évènement, permettant de conserver un historique du profil. Par exemple si un utilisateur modifie sa photo de profil, un évènement décrivant cette action avec des liens vers l'ancienne et la nouvelle photo sera enregistré.
- Modifier un arbre
- Modifier un évènement
- Modifier un profil
- Parcourir un arbre généalogique : Partir d'une vue générale d'un arbre ou d'une vue détaillée d'un membre ou d'évènements.
- Accepter ou non de faire partie d'un évènement avant d'y être affiché : Un utilisateur A peut ajouter un utilisateur B dans un évènement, mais le nom de B ne sera pas affiché tant qu'il ne donne pas son accord.

# 2. Résumé

Depuis l'essor des applications web, créer son arbre généalogique de manière collaborative avec d'autres utilisateurs est maintenant possible.

Plusieurs applications existent déjà mais aucune n'apporte de grandes possibilités, aussi bien pour la temporalité et variété des relations que pour l'association d'évènements et média aux personnes et relations.

Treemily est une application web qui offre aux utilisateurs la possibilité de créer des arbres généalogiques de manière collaborative et d'associer des évènements et média (images, vidéos, fichiers audio) aux personnes et aux relations y étant présentes.

Cette application permet également à un nouvel utilisateur de récupérer un profil déjà existant et l'associer à son compte et ainsi obtenir les pleins droits sur le profil et sur ses relations.

De plus, une timeline est à disposition de l'utilisateur pour pouvoir observer l'évolution de l'arbre généalogique dans le temps.

Le projet a été développé en Java (à l'aide du framework Play !), en TypeScript (à l'aide du framework Angular) et en PostgreSQL.

# 3. Introduction

## 3.1. Contexte

Ce projet est développé dans le cadre du travail de bachelor, réalisé lors du 6ème semestre de la formation en ingénierie logicielle à la HEIG-VD.

Le but de ce travail est d'appliquer les concepts et processus de développement enseignés lors de la formation.

## 3.2. Énoncé

Initialement proposé en 2016 par le Professeur Pier Donini, ce projet a comme énoncé :

« Le but de ce projet est de permettre la création collaborative non seulement d'arbres généalogiques mais aussi d'enregistrer les évènements marquants survenant à un individu ou à un groupe d'individus (p.ex. mariages, séparations, déménagements, naissances, décès, changements de profession...).

Dans un premier temps il s'agira d'établir un schéma de bases de données permettant de stocker les individus et les relations familiales les liant ainsi que les événements qui leur sont associés. Ce schéma devra également permettre aux utilisateurs de définir des droits sur leurs données (privé, public, restreint à un groupe...).

Dans un deuxième temps, il faudra concevoir et développer une application web multi utilisateurs permettant de saisir de manière ergonomique ces données, de définir leurs droits et de pouvoir les visualiser sous la forme d'arbres interactifs (p.ex. click sur un événement pour en obtenir la description textuelle ainsi que les photos ou vidéos associées).

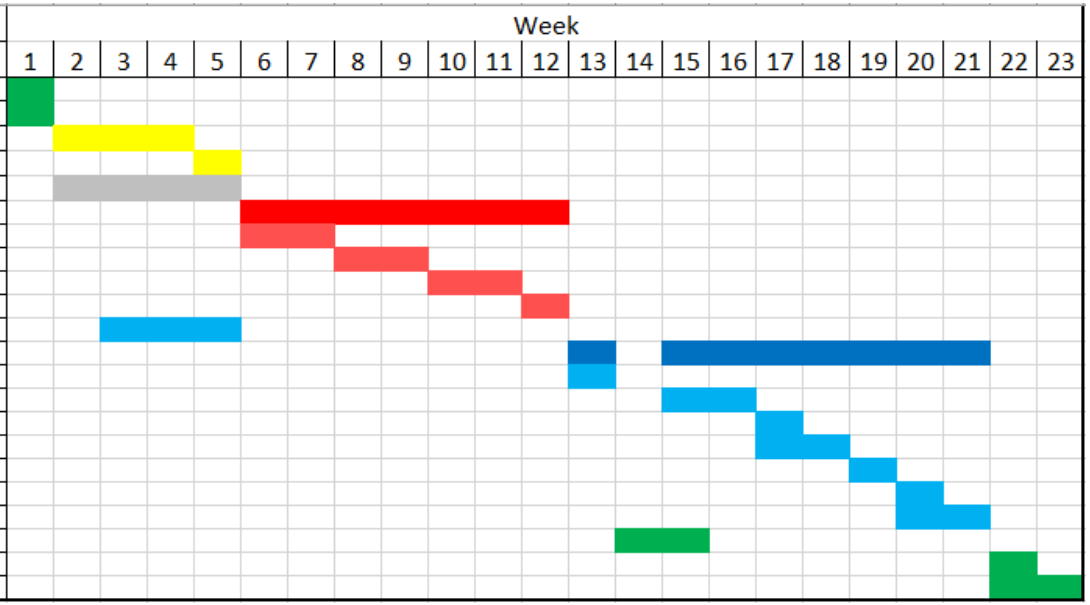
Ce projet sera réalisé en Java au moyen du framework *Play!* en HTML5 et CSS3. »

# 4. Préparation et déroulement

## 4.1. Planning

L'échéancier du projet a été défini comme suit :

| ID | Task | Duration (days) |
|---|---|---|
| 1 | Cahier des charges | 2 |
| 2 | Echéancier | 2 |
| 3 | Modélisation DB | 6 |
| 4 | Implémentation DB | 2 |
| 5 | Apprentissage Play | 8 |
| 6 | Développement serveur | 14 |
| 6,1 | Manipulation DB (CRUD) | 4 |
| 6,2 | API Rest (specification + développement) | 4 |
| 6,3 | Phase de tests - serveur + DB | 4 |
| 6,4 | Révision, corrections | 2 |
| 7 | Mockups | 6 |
| 8 | Développement client | 34 |
| 8,1 | Formulaires d'inscription, login | 2 |
| 8,2 | Création, gestion de profils et liens généalogiques | 7 |
| 8,3 | Recherche d'un profil | 2 |
| 8,4 | Création et gestion d'évènements | 8 |
| 8,5 | Vue d'un profil: chronologie, évènements, média et détails | 5 |
| 8,6 | Phase de tests - client | 3 |
| 8,7 | Révision, corrections | 7 |
| 9 | Rapport + présentation intermédiaire | 4 |
| 10 | README, description et manuel d'utilisation | 1 |
| 11 | Rapport final | 9 |




| Code couleurs | | | | | |
|---|---|---|---|---|---|
| **Couleur** | **Vert** | **Bleu** | **Rouge** | **Jaune** | **Gris** |
| **Phase associée** | Documentation, rapports | Développement client | Développement serveur | Développement DB | Apprentissage |

## 4.2. Respect des consignes et du planning

Les fonctionnalités prévues et les modes d'utilisation sont restés inchangés. Dans l'état actuel du logiciel, il est possible de voir l'arbre généalogique d'une personne (ses parents, ses enfants, ses relations). Il est également possible d'ajouter des personnes, relations et parents directement depuis la vue de l'arbre. La vue détaillée d'un profil est implémentée : Il est possible de voir les informations du profil, les évènements de sa vie et les médias associés. Il est également possible de modifier et effacer les profils, relations, parents et évènements.

En revanche, l'échéancier n'a pas été respecté :

- La modélisation de la base données (modèle EA, modèle relationnel et implémentation) a pris trois semaines de plus afin de garantir un modèle approprié, complet et adapté à ce que le logiciel vise à proposer aux utilisateurs.
- La rédaction du cahier des charges et de l'échéancier a pris une semaine de plus qu'initialement prévu.
- L'échéancier prévoyait d'implémenter le serveur avant de développer l'application client. Dû au fait que le serveur sert principalement à effectuer des opérations CRUD à la base de données, son implémentation a été mise en attente afin de pouvoir consacrer une partie du semestre au développement du client, plus compliqué et nécessitant plus de discussion avec les encadrants du travail. Le développement du client, étant toujours en cours, a également été poursuivi lors des semaines 14 et 15 en parallèle à la rédaction de ce rapport.
- Durant la seconde moitié du développement, les implémentations du serveur et du client se sont faites en parallèle, en suivant une approche « Top-down ». Ceci a permis de développer chaque fonctionnalité dans son integralité.

# 5. Structure du projet

Cette partie du rapport sert à décrire en premier lieu les technologies qui sont utilisées pour développer ce projet, puis la manière dont il fonctionne et est structuré et enfin la modélisation de la base de données, dont on présentera le modèle EA ainsi que le modèle relationnel.

Il est à préciser, dû au fait que le serveur n'est pas encore concrètement utilisable depuis le client, que la communication entre les deux logiciels ainsi que la structure du serveur pourront être étendues et modifiées

## 5.1. Technologies utilisées

### 5.1.1. Client

L'application client est développée avec HTML5, CSS3 et les technologies suivantes :

#### 5.1.1.1. Angular & TypeScript

Angular[1] est un Framework développé par Google, open-source, permettant de développer des applications web. Contrairement à son prédécesseur AngularJS, Angular (ou Angular2) est développé pour être utilisé en TypeScript[2].

TypeScript est un langage développé par Microsoft, défini comme un sur-ensemble de JavaScript, y ajoutant les concepts de classes, interfaces, et typage. Le langage est ensuite transcompilé en JavaScript pour pouvoir être interprété par les moteurs JavaScript des navigateurs[3]. Un avantage considérable acquis grâce à la compilation du code est le fait que, contrairement à JS, l'application ne sera pas rendue tant que des erreurs figurent dans le code. De plus, le typage des variables (qui reste optionnel) permet de rendre le logiciel plus robuste et sécurisé.

Angular permet la programmation orientée objets, l'injection de dépendances, l'utilisation de directives (attributs personnalisés d'éléments du DOM), l'animation de composants et la création de classes injectables.

#### 5.1.1.2. Angular-cli

Angular-cli[4] est une interface en ligne de commande permettant de gérer un projet Angular. Il sert à compiler, créer des composants, interfaces, services, types

[1] https://angular.io/
[2] https://www.typescriptlang.org/
[3] https://fr.wikipedia.org/wiki/TypeScript
[4] https://github.com/angular/angular-cli

énumérés, modules et les inclure dans les modules Angular automatiquement, à servir l'application et à créer des fichiers de test.

#### 5.1.1.3. D3js

D3js[5] est une librairie JavaScript qui permet de créer, d'attribuer des comportements et d'interagir avec des éléments du DOM de manière fluide. Son API liée aux éléments SVG s'avère particulièrement pratique dans ce projet, notamment pour adapter dynamiquement la vue de l'arbre généalogique.

#### 5.1.1.4. Angular2 Material

Angular2 Material[6] fournit des composants Angular prédéfinis qui respectent les spécifications de Material Design[7] défini par Google, tels que les barres de navigation, menus, dialogues, boutons, formulaires, icônes, tableaux et bien d'autres.

[5] https://d3js.org/
[6] https://material.angular.io/
[7] https://material.io/guidelines/

### 5.1.2. Serveur

Le serveur est développé en Java 1.8 et utilise les technologies suivantes :

#### 5.1.2.1. Play Framework

Le Framework Play[8] est un Framework open-source développé en Scala[9], mais compatible avec d'autres langages compilés en Bytecode[10].

Conçu pour être utilisé dans le cadre du développement d'applications web, Play a la particularité d'être :

- RESTful[11]
- Stateless[12]
- Convention plutôt que configuration[13]

Ces particularités le rendent adapté à ce projet, puisque le serveur ne répondra qu'à des évènements enclenchés par l'application client.

Lorsqu'une requête http est reçue sur le port d'écoute de l'application, un fichier de configuration lui permet de la rediriger vers un contrôleur adapté qui se chargera de traiter la requête puis de formuler une réponse.

#### 5.1.2.2. SBT

SBT[14], acronyme de Scala Build Tool, est un outil permettant de compiler du code Scala ou Java, gérer des dépendances et télécharger les librairies nécessaires. Similaire à Maven, SBT est le moteur de production préconisé par Play. Les fichiers de configuration de SBT sont écrits en Scala plutôt qu'en XML pour Maven.

#### 5.1.2.3. Hibernate

L'ORM[15] utilisé pour ce projet est Hibernate[16], un framework implémentant la spécification JPA[17] permettant de lier des classes Java à une base de données relationnelle. Ayant son propre langage de requêtes, HQL, basé sur SQL, Hibernate gère les communications à la base de données. Dans le cadre de Treemily, la base

[8] https://www.playframework.com/
[9] https://www.scala-lang.org/
[10] https://en.wikipedia.org/wiki/Bytecode
[11] https://en.wikipedia.org/wiki/Representational_state_transfer
[12] https://en.wikipedia.org/wiki/Stateless_protocol
[13] https://en.wikipedia.org/wiki/Convention_over_configuration
[14] http://www.scala-sbt.org/
[15] https://en.wikipedia.org/wiki/Object-relational_mapping
[16] http://hibernate.org/
[17] https://en.wikipedia.org/wiki/Java_Persistence_API

de données est implémentée séparément ; le schéma doit donc être importé et interprété par le framework, contrairement à d'autres cas d'utilisation où les schémas et tables sont générés depuis des POJO[18] codés en Java.

### 5.1.3. Base de données

La base de données utilisée est une base de données relationnelle SQL.

#### 5.1.3.1. PostgreSQL

PostgreSQL[19] est un système de base de données relationnelle, basé sur le langage SQL en y ajoutant des opérations supplémentaires. Il propose de nombreuses fonctionnalités pratiques, telles que les clés étrangères qui réfèrent automatiquement aux clés primaires des tables associées.

### 5.1.4. Rédaction, codage, modélisation

#### 5.1.4.1. Modèle EA

Le modèle entités-associations a été dessiné avec StarUML[20], un logiciel de modélisation

#### 5.1.4.2. Serveur et base de données

Le serveur est développé avec l'IDE Intellij de Jetbrains[21], proposant un support pour de nombreux frameworks dont Play et Hibernate. On peut également accéder à la base de données depuis le logiciel, aussi bien pour parcourir les tables que pour effectuer des requêtes.

#### 5.1.4.3. Client

Le client est développé avec Visual Studio Code[22], un éditeur de code open-source développé par Microsoft qui a de nombreux plugins et est particulièrement adapté au développement web.

#### 5.1.4.4. Rapport, cahier des charges et échéancier

Le rapport, le cahier des charges et l'échéancier sont rédigés avec la suite Microsoft Office.

#### 5.1.4.5. Github

Github[23] et Git sont utilisés pour gérer le versioning et le backup du code source du projet.

---

[18] https://en.wikipedia.org/wiki/Plain_old_Java_object
[19] https://www.postgresql.org/
[20] http://staruml.io/
[21] https://www.jetbrains.com/idea/
[22] https://code.visualstudio.com/
[23] https://github.com/

### 5.1.5. Tests unitaires

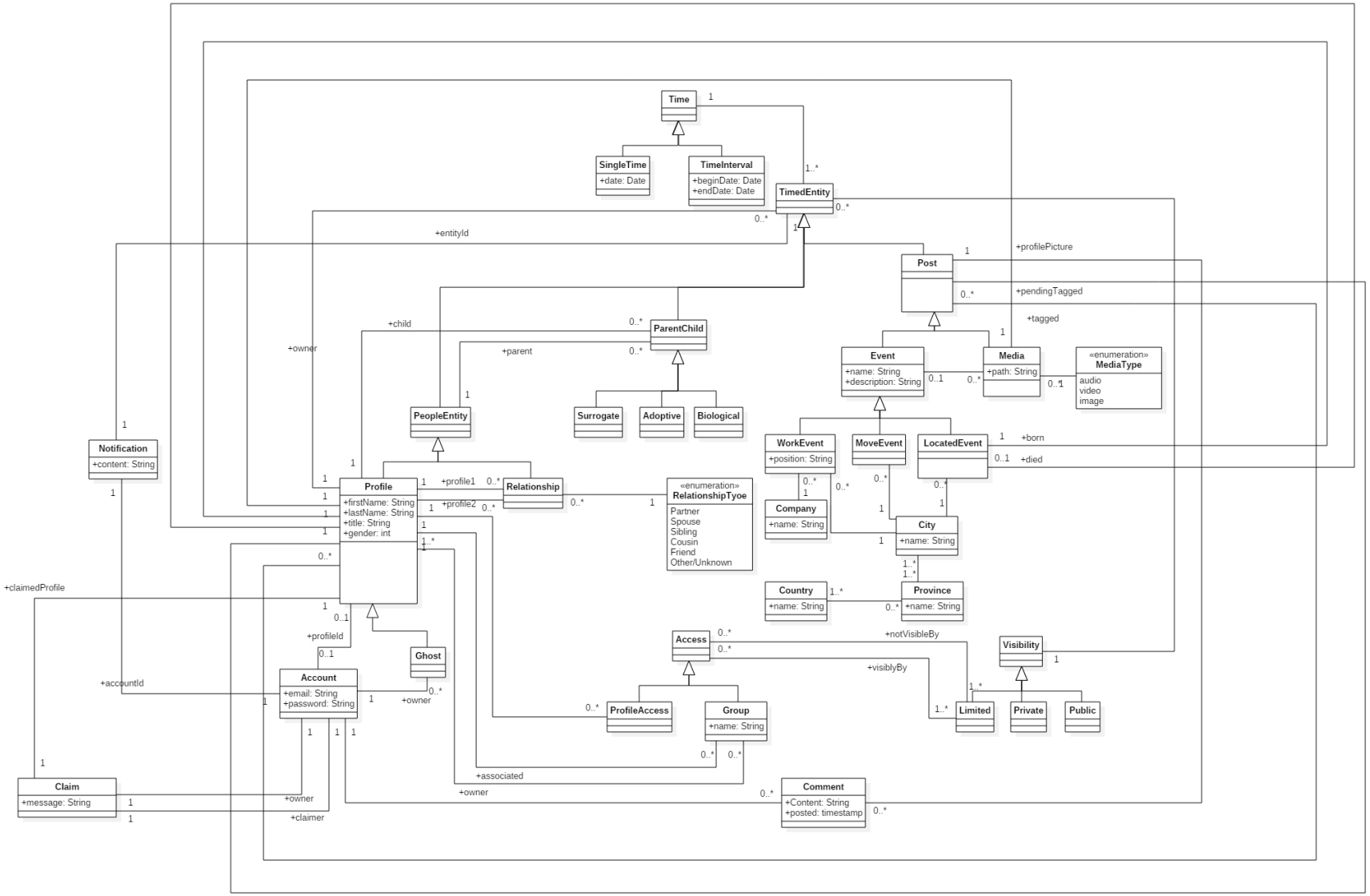
Les tests unitaires sont développés à l'aide du framework de tests Mocha[24] et des librairies Chai[25] et Chai-as-promised[26]. Ces utilitaires sont conçus pour être utilisés en JavaScript et avec Node.js[27].

Ensemble, ils permettent de tester des comportements asynchrones, notamment les requêtes http vers le serveur.

[24] https://mochajs.org
[25] http://chaijs.com
[26] https://github.com/domenic/chai-as-promised
[27] https://nodejs.org/en/

# 5.2. Modélisation

## 5.2.1. Modèle Entité-Association



### 5.2.2. Description des choix

#### 5.2.2.1. TimedEntity

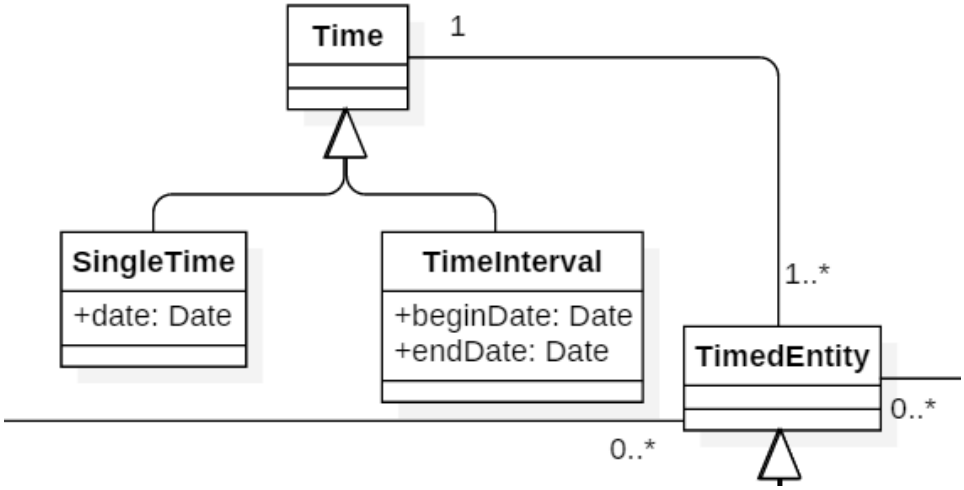



*Figure 1- TimedEntity*

La base de ce modèle est l'entité TimedEntity. Les relations, personnes et évènements sont des généralisations de cette entité.

TimedEntity décrit un objet qui est délimité dans le temps, soit par une date fixe, SingleTime (qui peut indiquer un jour unique ou un intervalle de temps dont la date de fin n'est pas connue).

#### 5.2.2.2. Profils, relations et parents

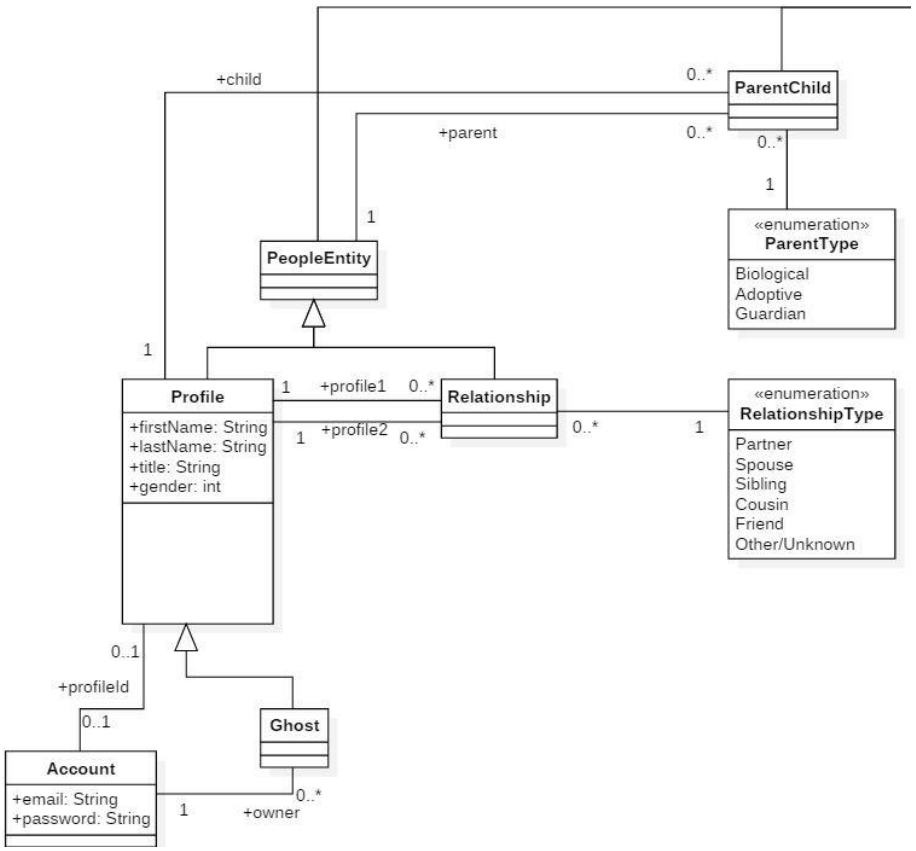



*Figure 2- Relations, profils, comptes et parents*

Les profils et relations héritent d'une entité nommée PeopleEntity, dû au fait que l'entité ParentChild peut être liée soit à une personne, soit à une relation entre deux personnes. Ce choix a été fait pour que l'utilisateur puisse préciser s'il le souhaite un ou deux parents pour une personne, que ce soit pour des parents biologiques ou pour des parents adoptifs.

Une relation lie deux personnes. Une personne peut être dans plusieurs relations ou dans aucune.

Un ParentChild n'admet qu'un seul parent, que ce soit une relation ou une personne seule.

Initialement, un Account représentait un utilisateur enregistré héritant de Profile, le profil duquel le compte héritait étant le profil de l'utilisateur. Finalement, j'ai opté pour dissocier les comptes des profils, afin de gérer les réclamations.

En effet, lorsqu'un utilisateur créé un compte et qu'il souhaite récupérer un profil déjà existant comme étant le sien, il faut que le propriétaire approuve la demande. En attendant une confirmation ou un refus, le compte doit tout de même exister. Si

le propriétaire du profil réclamé accepte, ce dernier devient le profil principal du compte. Sinon, le réclamateur reçoit une notification lui demandant de créer son profil.

De plus, si un utilisateur supprime son profil, son compte ne sera pas supprimé en conséquence.

Un Account peut posséder des Ghosts, qui représentent des profils qui ne sont pas associés à un compte, qu'un utilisateur différent a créés. Jusqu'à ce qu'un compte soit créé pour récupérer ce profil, son owner a tous les droits sur ce profil.

#### 5.2.2.3. Posts, Events, Media

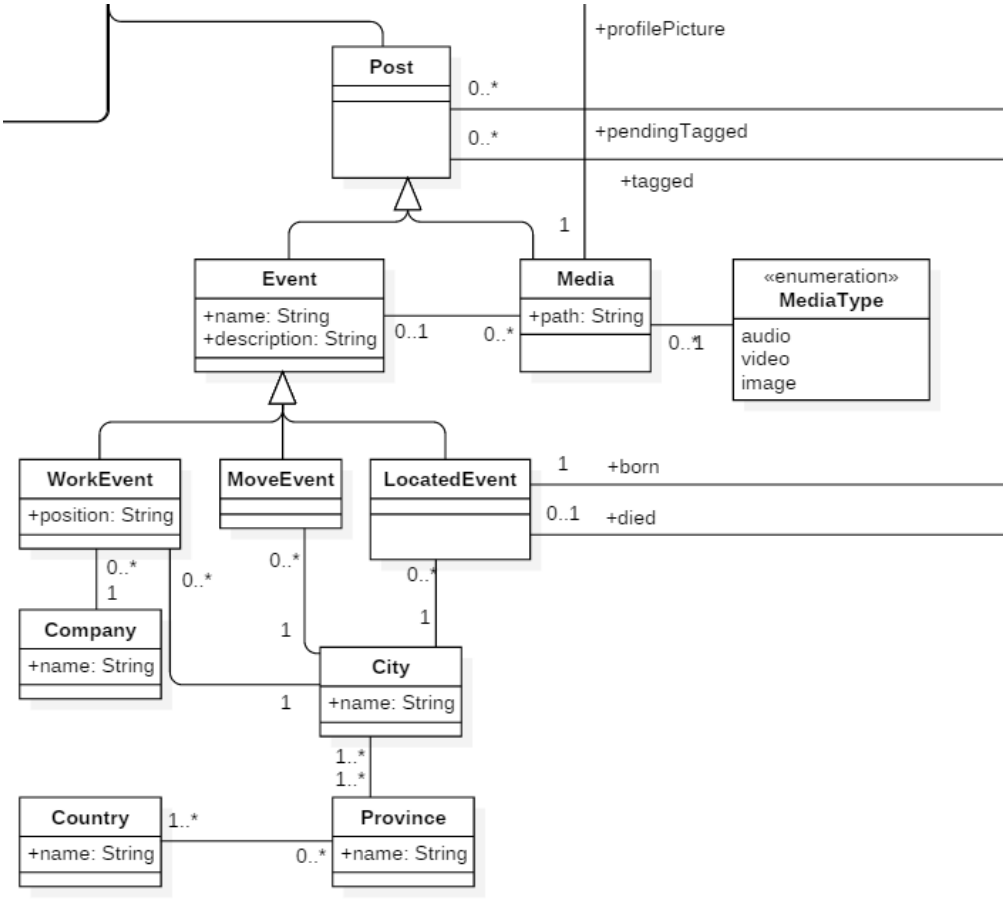



*Figure 3- Posts, Events, Media, Location*

Le choix de faire hériter de Post les entités Event et Media vient de la nécessité de pouvoir associer des commentaires, profils et une visibilité aux deux.

Tagged et pendingTagged sont des associations à des profils. Elles représentent les personnes qui sont associées à un Post. Lors de l'ajout d'une personne associée,

elle est automatiquement enregistrée comme une relation pendingTagged, jusqu'à ce que la personne en question (ou son propriétaire) accepte d'y être associé avant d'être migrée vers une association tagged.

Le propriétaire d'un évènement ou média est hérité de l'association Profile – TimedEntity nommée owner.

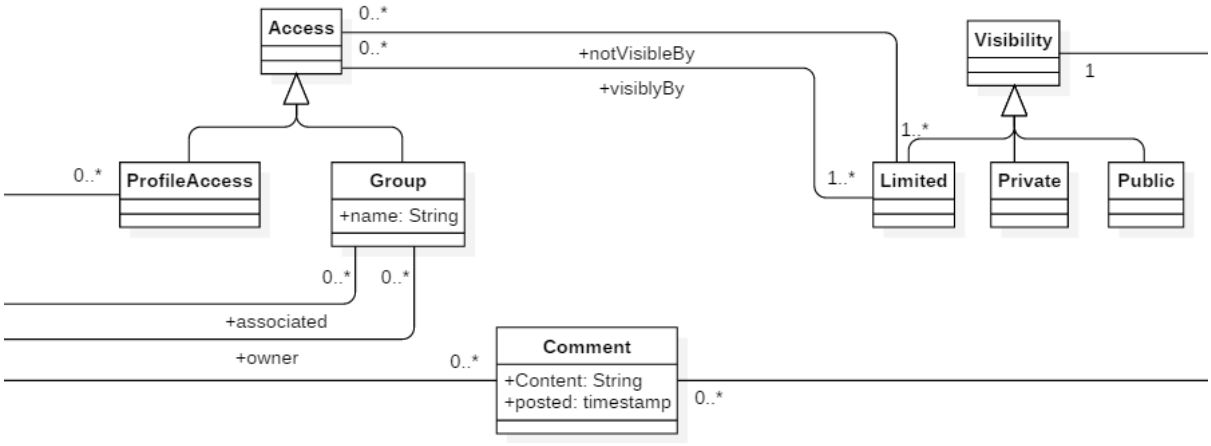
Un média est soit lié à un évènement soit une photo de profil. Des médias ne peuvent pas être créés indépendamment de ces deux cas. Trois types de média sont reconnus : audio, image ou vidéo.

Un emplacement (Location) est décrit par une ville (City), une région ou Province, et un pays (Country). Dû à la possibilité d'avoir des profils datant d'époques où les pays et régions avaient des noms différents ou n'étaient pas dans la même entité (par exemple : Leipzig, qui se trouve actuellement en Saxonie, RFA se trouvait précédemment en RDA).

Il y a quatre sortes d'évènements :

- Event : Un évènement « classique », auquel peuvent être associés des médias, des commentaires, des personnes. Possède un nom, une description, un propriétaire et une date ou un intervalle de temps.
- LocatedEvent : Un évènement situé. En plus des attributs et associations hérités d'Event, est lié à un emplacement. De plus, chaque Profile a une association vers un LocatedEvent, nommé « born » et peut ou non avoir une autre relation nommée « died ».
- MoveEvent : Un MoveEvent représente un déménagement, et possède une association à un emplacement, la nouvelle ville de résidence de la personne.
- WorkEvent : Un WorkEvent représente un emploi qu'occupe une personne. Il possède un attribut « position » et est associé à une entreprise (Company). Une entreprise peut évidemment être associée à plusieurs WorkEvents.

#### 5.2.2.4. Vibilité, accès, commentaires et groupes




Chaque TimedEntity a une visibilité. Ainsi, des profils, évènements, média, relations et liens de type parent peuvent avoir une visibilité limitée selon le choix du créateur de l'entité en question.

Il y a trois sortes de visibilité définies dans Treemily :

- Une visibilité publique signifie que tout utilisateur, enregistré ou non, a accès au contenu.
- Une visibilité privée signifie que seulement le créateur et les personnes associées ont accès à l'entité. Exemples :
    - Une relation avec une visibilité privée sera visible par les deux membres de la relation.
    - Un évènement avec une visibilité privée sera visible par son créateur et les personnes associées (aussi bien pendingTagged que tagged).
- Une visibilité limitée signifie qu'en plus de la visibilité de base (créateur et associés), celle-ci est étendue à une entité de type Access. Un accès peut être accordé ou empêché (visibleBy et notVisibleBy) à des personnes spécifiées individuellement ou à des groupes. Un groupe est un ensemble de personnes, ayant un propriétaire et un nom, qui est sauvegardé afin de pouvoir avec facilité le réutiliser à plusieurs reprises.

De manière générale, un TimedEntity à visibilité limitée sera visible par :

$$Soit\ V\ l'ensemble\ des\ personnes\ pouvant\ voir\ l'entité$$
$$Soit\ U\ l'ensemble\ de\ tous\ les\ utilisateurs, inscrits\ ou\ non$$
$$Soit\ E_V\ l'ensemble\ des\ profils\ spécifiés\ dans\ visibleBy$$
$$Soit\ E_{NV}\ l'ensemble\ des\ profils\ spécifiés\ dans\ notVisibleBy$$
$$Soit\ E_{T,PT}\ l'ensemble\ des\ personnes\ tagged\ et\ pendingTagged\ s'il\ s'agit\ d'un\ énènement$$
$$Soit\ O\ le\ propriétaire\ de\ l'entité$$

$$Si\ E_V = \emptyset => V := (U \setminus E_{NV}) \cup O \cup E_{T,PT}$$
$$Sinon => V := (E_V \setminus E_{NV}) \cup O \cup E_{T,PT}$$

Ces formules garantissent que les personnes tagged et pendingTagged (s'il s'agit d'un évènement) et le propriétaire du TimedEntity auront toujours le droit de le voir. Dans le cas où l'ensemble visibleBy est défini, l'ensemble notVisibleBy lui sera soustrait. Ainsi, un utilisateur pourrait inclure un groupe tout en excluant un sous-ensemble de ce groupe sans produire d'erreur logique.

Sachant qu'un média peut avoir sa propre visibilité, la priorité des visibilités d'un média est gérée comme suit : si un média a une visibilité plus limitée que l'évènement le contenant, il conserve sa visibilité. En revanche, si un média est défini comme ayant une visibilité plus large que son évènement, la visibilité du média sera « écrasée » par celle de son évènement.

Un média ne peut pas avoir des profils tagged qui ne sont pas tagged à l'évènement le contenant. L'utilisateur aura donc le choix entre étendre les tagged de l'évènement en y ajoutant ceux du média en question ou retirer ces mentions du média.

Une contrainte d'intégrité à prendre en considération lors de l'implémentation est que toute personne d'une relation ne peut pas être exclue de la visibilité de cette relation.

Enfin, un Post peut avoir des commentaires, qui sont associés à un Profile. Un commentaire a un contenu et une estampille de la date à laquelle il a été soumis.

#### 5.2.2.5. Notifications et réclamations

Durant la seconde moitié du développement de Treemily, deux entités supplémentaires ont été ajoutées, servant à gérer les réclamations de profil et les notifications concernant les droits des entités.

- Notifications

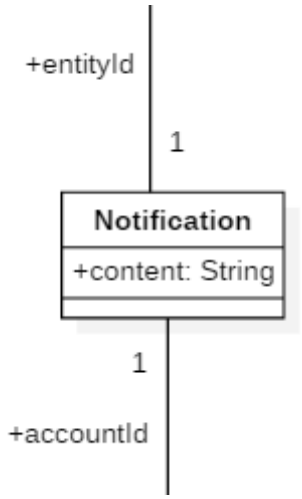



*Figure 4- Notifications*

Une notification est enregistrée lorsqu'un compte ajoute des relations et des parents utilisant un ou plusieurs profils ne lui appartenant pas. Lors de l'enregistrement, une notification est créée, où entityId référence l'entité à approuver ou refuser par le compte, référencé par accountId. Lorsque l'utilisateur dont les profils ont été utilisés se connecte sur l'application, il recevra une notification au sujet des nouveaux liens. Il peut ensuite approuver, ce qui effacera la notification et conservera la nouvelle relation, ou refuser, ce qui efface la relation et la notification. Le contenu de la notification est fixé par le serveur, par exemple :

"john.doe@email.com set Jane Doe as spouse of John Doe"

- Réclamations

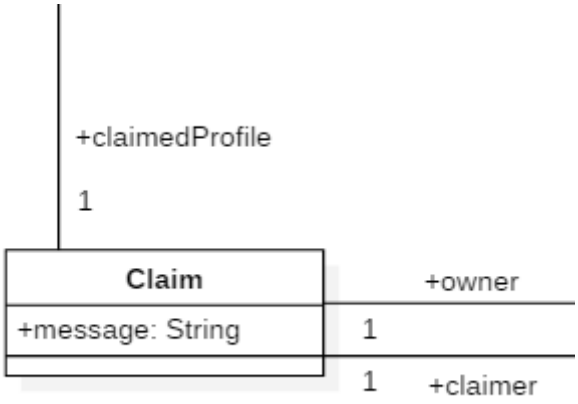



*Figure 5- Réclamations*

Une réclamation est enregistrée lorsqu'un nouveau compte réclame un profil déjà existant ne lui appartenant pas. L'association claimedProfile lie la réclamation au profil réclamé. L'association owner lie la réclamation au compte propriétaire du profil et claimer au compte réclamant le profil. Le contenu, message, est fixé par le compte réclamant le profil, pour que ce dernier puisse éventuellement laisser un message expliquant les raisons de la réclamation.

Exemple :

john.doe@email.com claims John Doe: "Hey, it's me! Can you give me my profile?"

# 6. Implémentation

## 6.1. Base de données

La base de données a été créée en utilisant un ensemble de requêtes en PostgreSQL. Les entités définies dans l'UML sont conservées, tout en ajoutant des tables d'association pour représenter les associations « many-to-many ».

## 6.2. Serveur

Le serveur est organisé en 4 packages[28] Java et des fichiers de configuration :

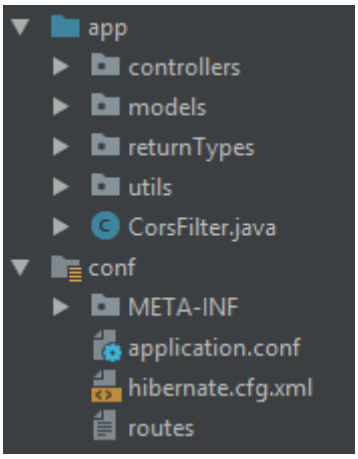



*Figure 6- Structure du serveur*

### 6.2.1. Routes

Le fichier **routes** spécifie les points d'entrée du serveur. Il indique, pour un URL et un type de requête spécifique[29], vers quel contrôleur et quelle méthode transférer la requête. Si aucune route n'est trouvée pour une requête, une erreur 404[30] est retournée à l'émetteur.

Exemple d'entrée dans le fichier routes :

```
POST        /auth                       @controllers.AccountController.login
```

[28] https://docs.oracle.com/javase/tutorial/java/concepts/package.html
[29] http://www.restapitutorial.com/httpstatuscodes.html

Cette ligne indique que lors d'une requête de type POST, à l'url relatif « /auth », le contenu de la requête doit être redirigé vers la méthode « login » du contrôleur « AccountController ».

À mon humble avis, cette convention représente le principal avantage du framework Play, permettant au développeur de ne pas se soucier des requêtes dont le traitement n'est pas encore implémenté.

### 6.2.2. Contrôleurs

Les contrôleurs ont été développés comme des classes mettant à disposition un ensemble de méthodes, ne nécessitant donc pas l'instanciation des contrôleurs durant l'exécution.

Huit contrôleurs sont utilisés par l'application :

#### 6.2.2.1. AccountController

Ce contrôleur gère les requêtes liées spécifiquement aux comptes, soit la création et la mise-à-jour d'un compte, le login, l'association d'un compte à un profil principal ainsi que la gestion des réclamations et notifications.

Exemple pour l'authentification :

```
public Result login() {
    JsonNode json = request().body().asJson();
    Session session =
SessionHandler.getInstance().getSessionFactory().openSession();
    String queryString = "from Account where email = :emailParam";
    Query query = session.createQuery(queryString);
    query.setParameter("emailParam", json.get("email").asText());
    try {
        Account result = (Account) query.list().get(0);

        session.close();
        if (result.getPassword().equals(json.get("password").asText())) {
            result.setPassword("");
            JsonNode resultJson = Json.toJson(result);
            return ok(resultJson);
        } else {
            return badRequest("Wrong password");
        }
    } catch (Exception e) {
        return notFound();
    }
}
```

Ici, la première ligne de code récupère le « payload » de la requête.

Ensuite, une intance de Session hibernate est ouverte, par le biais d'un singleton nommé SessionHandler (décrit dans la partie « Utilitaires »).

Une requête à la base de données est créée, cherchant un compte associé à l'adresse email spécifiée dans le corps de la requête. Si aucun compte avec cette

adresse n'existe, une erreur 404 est retournée. Si un compte est trouvé, l'instance du compte est retournée, après avoir effacé le mot de passe (cette délétion n'est pas enregistrée dans la base de données, mais sert uniquement à ne pas transmettre le mot de passe dans la réponse).

La librairie Json[31] proposée par le framework est beaucoup utilisée dans ce projet, servant à aisément convertir des objets java habituels en format JSON, de manière similaire à la librairie Gson[32] de Google, plus populaire.

Lorsqu'un Result est retourné, le framework se charge de le retransmettre à l'émetteur.

[31] https://playframework.com/documentation/2.0/api/java/play/libs/Json.html
[32] https://github.com/google/gson

#### 6.2.2.2. CommentController

Le CommentController sert à gérer la création et la récupération de commentaires liés à un Post, superclasse des évènements et média dans la modélisation. Même si le serveur et la base de données le permettent, l'addition de commentaires à un media n'est pas implémenté côté client.

Exemple de récupération des commentaires d'un Post :

```
public Result getComments(Integer postid) {
    Integer requesterId = -1;
    if (request().hasHeader("requester"))
        requesterId = Integer.parseInt(request().getHeader("requester"));
    if (Util.isAllowedToSeeEntity(requesterId, postid)) {
        Session session = SessionHandler.getInstance().getSessionFactory().openSession();

        Query query = session.createQuery(
                "select p.firstname, " +
                        "p.lastname, " +
                        "c.content, " +
                        "c.postedon " +
                        "from Comment c " +
                        "inner join Profile p on c.commenter = p.peopleentityid " +
                        "where c.postid = :postid");

        query.setParameter("postid", postid);
        List<Object[]> comments = query.list();
        List<CommentResult> results = new ArrayList<>();
        for (Object[] comment : comments) {
            String name = comment[0].toString() + " " + comment[1].toString();
            String content = comment[2].toString();
            String date = new Date(((Timestamp) comment[3]).getTime()).toString();
            results.add(new CommentResult(date, content, name));
        }
        session.close();
        return ok(Json.toJson(results));
    } else {
        return forbidden();
    }
}
```

Le paramètre postid donné à la méthode est un paramètre URL et spécifie l'entité dont on souhaite récupérer les commentaires.

L'entête de la requête nommé « requester » est spécifié à chaque requête effectuée depuis le client au serveur, et représente l'identificateur du compte. Si l'utilisateur n'est pas enregistré, l'entête n'est pas présent.

Il faut préciser qu'un utilisateur non enregistré de l'application a uniquement accès aux entités publiques.

La méthode isAllowedToSeeEntity(Integer requesterId, Integer timedEntityId) de la classe Util, retourne une valeur vraie ou fausse représentant si l'utilisateur a le droit de voir l'entité (voir partie « Utilitaires »).

Si la méthode retourne faux, une erreur « 403 forbidden » est retournée, sinon tous les commentaires sont récupérés, en incluant également le prénom et nom de famille de l'utilisateur ayant commenté.

#### 6.2.2.3. EventController

L'EventController sert à créer, récupérer et modifier des évènements.

```
public class EventController {
    @Transactional
    public Result addEvent() {...}

    public Result getEvent(Integer id) {...}

    private Result getWorkEvent(Integer id) {...}

    private Result getLocatedEvent(Integer id) {...}

    private Result getMoveEvent(Integer id) {...}

    @Transactional
    public Result updateEvent(Integer eventid) {...}
}
```

Dû à la longueur des corps de ces méthodes, seulement leurs entêtes sont précisés ici.

La méthode addEvent enregistre l'évènement si le corps de la requête est valide.

Le corps précise le type d'évènement à enregistrer. Premièrement, la date ou l'intervalle de temps est enregistré, suivi du timedEntity, suivi du Post puis de l'Event. Ensuite, si un type est précisé, par exemple LocatedEvent, un sous-évènement est enregistré ainsi que sa localisation.

Lors de la récupération d'un évènement (getEvent()), une vérification a lieu : S'il existe une entrée avec l'identificateur fourni dans une des tables des sous-évènements, la méthode retourne le résultat de la méthode spécifique à ce type d'évènement.

La mise à jour d'un évènement permet de changer son type, à l'exception des évènements étant également associés comme born ou died à un profil. Un évènement localisé (LocatedEvent) peut par exemple être converti en déménagement (MoveEvent), en conservant les données communes si des nouvelles données ne sont pas précisées. Typiquement, dans le cas ci-dessus, si une nouvelle localisation n'est pas précisée, l'ancienne sera migrée dans le nouvel évènement.

Ici, toutes les méthodes vérifient les droits de l'émetteur avant d'effectuer des insertions, mises-à-jour ou lectures. De même que pour les commentaires, un code 403 est retourné si les droits ne sont pas suffisants.

Un évènement peut être supprimé, mais cette requête n'est pas gérée dans ce contrôleur, puisqu'il s'agit des mêmes opérations que pour un profil, une relation ou un lien de type parent (dû à l'héritage depuis timedEntity et à la suppression en cascade de ses sous-types).

#### 6.2.2.4. GroupController

Le GroupController permet de gérer la création, la récupération et la suppression de groupes. La modification d'un groupe n'a pas été implémentée.

Voici le code permettant de créer un groupe :

```
@Transactional
public Result createGroup() {
    Session session = SessionHandler.getInstance().getSessionFactory().openSession();
    session.getTransaction().begin();


    Group group = new Group();
    JsonNode jsonNode = request().body().asJson();
    group.setName(jsonNode.get("name").asText());
    group.setOwner(jsonNode.get("owner").asInt());
    session.save(group);

    for (int i = 0; i < jsonNode.get("people").size(); i++) {
        Grouppeople grouppeople = new Grouppeople();
        grouppeople.setGroupid(group.getId());
        grouppeople.setProfileid(jsonNode.get("people").get(i).asInt());
        session.save(grouppeople);
    }

    session.getTransaction().commit();
    session.close();
    return ok();
}
```

Supprimer un groupe :

```
@Transactional
public Result deleteGroup(Integer groupid) {
    if (!request().hasHeader("requester")) {
        return forbidden("No requester specified");
    }
    Integer requesterId = Integer.parseInt(request().getHeader("requester"));
    Session session = SessionHandler.getInstance().getSessionFactory().openSession();
    Group group = session.get(Group.class, groupid);
    if(group == null) {
        session.close();
        return notFound();
    }

    if (group.getOwner() == requesterId) {
        session.getTransaction().begin();
        session.delete(group);
        session.getTransaction().commit();
    } else {
        session.close();
        return forbidden();
    }
    session.close();
    return ok();
}
```

Il est indispensable que le fait d'effacer un groupe ne soit possible qu'au propriétaire de ce dernier.

Des vérifications ont donc lieu :

- Premièrement, si le groupe en question n'existe pas, l'émetteur de la requête en est informé.

- Deuxièmement, un groupe ne peut être effacé que par son propriétaire. S'il y a conflit entre l'identificateur fourni et le propriétaire du groupe, la suppression n'a pas lieu.

#### 6.2.2.5. ParentController

Ce contrôleur gère la création, la mise-à-jour et l'association à un évènement d'un Parent.

Un évènement associé à un parent est un Event dont le timedEntity associé est le même que celui du parent. Ceci permet notamment depuis le client de pouvoir cliquer sur un lien de type parent et obtenir les média, commentaires et détails associés. Cet évènement a donc la même visibilité que le parent, sachant que la visibilité est fixée au niveau du timedEntity.

Lorsqu'un parent est créé, plusieurs insertions dans timedEntityOwner ont lieu :

- Premièrement, l'enfant est propriétaire de l'entité
- Deuxièmement, si le parent est un Profil, il est propriétaire. Si le parent est une relation, les deux profils de cette relation sont propriétaires.

Enfin, si les profils associés à la nouvelle entité Parentsof n'appartiennent pas tous au même compte, une notification est enregistrée pour chaque compte dont le profil ou la relation a été utilisé.

#### 6.2.2.6. RelationshipController

Le RelationshipController, de manière similaire à ParentController, gère la création, la modification et la suppression d'une relation.

Si un ou deux des profils de la relation n'appartiennent pas au créateur de cette dernière, une notification est enregistrée pour les comptes propriétaires des profils en question.

#### 6.2.2.7. FamilyController

Ce contrôleur gère la récupération de l'arbre généalogique d'une personne. La méthode en question, getFamily(Integer id), suit le fonctionnement suivant :

```
1. Vérifier que celui qui émet la requête a le droit de voir le profil dont
   l'identifiant a été donné en paramètre
2. Récupérer le profil simplifié du profil en question (identificateur, prénom, nom,
   photo de profil, sexe)
3. Récupérer les relations qu'a le profil
4. Pour chaque relation
      4.1.  Si l'émetteur de la requête n'a pas le droit de voir la relation, la
            retirer
      4.2.  Sinon, récupérer les informations supplémentaires de la relation
            (notamment ses dates)
      4.3.  Récupérer le profil simplifié de la personne n'étant pas le profil dont
            on récupère l'arbre de la relation
      4.4.  Si le profil n'est pas visible par la personne effectuant la requête, on
            retire ce dernier ainsi que la relation
5. Récupérer les liens de type Parent eus par le profil, depuis ses relations et en
   tant que parent célibataire
6. Pour chaque parent
      6.1.  Si l'émetteur n'a pas le droit de le voir, le retirer
      6.2.  Sinon, ajouter l'enfant
      6.3.  Si l'émetteur n'a pas le droit de voir l'enfant, retirer l'enfant et le
            Parent
7. Récupérer les parents du profil
8. Pour chaque parent :
      8.1.  Si l'émetteur de la requête n'a pas le droit de voir le parent, celui-ci
            est retiré de la liste des parents
      8.2.  Sinon, récupérer les données relatives à ce parent (profils et relations)
      8.3.  Si l'utilisateur n'a pas le droit de voir un des profils associés à ce
            parent, retirer le parent ainsi que les profils associés
      8.4.  Récupérer les autres enfants eus par le ou les parents du profil demandé.
```

#### 6.2.2.8. ProfileController

Le ProfileController gère la création de profil, la création de Ghost, la suppression de profil et la récupération du profil complet d'une personne (incluant ses évènements et média).

Lors de la récupération d'un profil, les droits sont vérifiés. Si l'émetteur de la demande a le droit de voir le profil, on vérifie ensuite s'il a le droit de voir chacun des évènements avant de les inclure dans la réponse.

#### 6.2.2.9. SearchController

Le SearchController sert uniquement à retourner une liste de profils selon les données qui lui sont fournies, tout en vérifiant si l'émetteur de la recherche a le droit de voir les profils.

La recherche de profils dans Treemily est à ce stade relativement rudimentaire. Il faut fournir un prénom et/ou un nom de famille séparément. Une requête est ensuite effectuée à la base de données pour récupérer les profils dont le prénom contient le prénom fourni et où le nom de famille contient le nom de famille fourni.

#### 6.2.2.10. UploadController

L'UploadController sert uniquement à gérer le téléchargement de fichiers vers le serveur et retourne son type et son chemin relatif à l'adresse du serveur.

### 6.2.3. Types de retour

Les types de retour sont des classes représentant les données qui doivent être retournées au client lors de requêtes.

Par exemple, un profil dans un arbre généalogique ou un résultat de recherche ne nécessite pas plus d'informations que le prénom, le nom de famille, le sexe, la photo de profil et les dates de naissance et de mort. La classe ProfileResult a été implémentée en n'ayant que ces attributs. Ceci permet ensuite de retourner un tableau de ProfileResults que le client pourra interpréter.

L'utilité des types de retour est due au fait que la modélisation sépare beaucoup des données.

Les types de retour suivants ont été créés :

- ProfileResult
- CommentResult (ajoutant au contenu du commentaire le prénom et le nom de famille du profil ayant commenté et l'estampille du commentaire)

- ClaimResult, contenant le message, l'email du réclamateur, le prénom et nom de famille du profil réclamé.
- LocationResult : contient le nom de la ville, de la province et du pays d'un lieu (séparés dans la base de données)
- EventResult : contient l'identificateur, le nom, la description, les dates sous forme de chaînes de caractères et les média de l'évènement.
    - LocatedEventResult : sous-classe d'EventResult, contenant également un LocationResult et une chaîne de caractères contenant le type de l'évènement
    - MoveEventResult : identique à LocatedEventResult mais ayant un type d'évènement différent
    - WorkEventResult : contient également un LocationResult et le nom de l'entreprise et le poste
- ParentResult : contient l'identificateur du Parentsof, les identificateurs de l'enfant et du parent, que ce soit une relation ou un profil, ainsi que ses dates.
- FullProfile : contient un ProfileResult, une liste d'EventResult et un ou deux LocatedEventResult, représentant l'évènement de naissance et de mort du profil. Ce type est retourné lors de la récupération du profil complet.
- RelationshipResult : contient un identificateur, les identificateurs des profils, les dates de début et de fin, le type de la relation.
- FamilyResult : contient une liste de RelationshipResult, une liste de ParentResult et une liste de ProfileResult. Permet de retourner l'arbre généalogique complet d'une personne dans un seul objet.
- UploadFile : Représente le type et le chemin relatif à l'adresse du serveur d'un fichier.

Ces types de retour sont localisés dans le package returnTypes. Leur but est d'être sérialisé (d'où la visibilité publique de ses attributs) puis retourné au client.

### 6.2.4. Hibernate

Hibernate a été utilisé pour gérer les requêtes à la base de données. Le singleton **SessionHandler** s'occupe de gérer le « SessionFactory[33] ».

```
public class SessionHandler {

    private static SessionHandler instance = null;

    SessionFactory sessionFactory;

    public static SessionHandler getInstance() {
        if (instance == null) {
            instance = new SessionHandler();
        }
        return instance;
    }

    protected SessionHandler() {
        Configuration configuration = new Configuration();
        this.sessionFactory = configuration.configure().buildSessionFactory();
    }

    public SessionFactory getSessionFactory() {
        return this.sessionFactory;
    }

}
```

Une session peut ensuite être instanciée ainsi :

```
Session session = SessionHandler.getInstance().getSessionFactory().openSession();
```

Le package **models** contient les POJOs[34] relatifs aux tables de la base de données.

Par exemple, pour la table **City** :

```
@Entity
public class City {
    private int id;
    private String name;

    @Id
    @Column(name = "id")
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @Basic
    @Column(name = "name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

[33] https://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/SessionFactory.html
[34] https://en.wikipedia.org/wiki/Plain_old_Java_object

```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        City city = (City) o;

        if (id != city.id) return false;
        return name != null ? name.equals(city.name) : city.name == null;
    }

    @Override
    public int hashCode() {
        int result = id;
        result = 31 * result + (name != null ? name.hashCode() : 0);
        return result;
    }
}
```

Les classes Java correspondantes aux tables de la base de données ont été autogénérées par l'IDE, en se basant sur le schéma PostgreSQL « familytree ». Certaines classes ont ensuite dû être modifiés pour corriger les imperfections de l'importation. Notamment, le mot-clé SERIAL en PostgreSQL n'est pas interprété de la bonne manière par l'IDE : Il a donc fallu ajouter une annotation signifiant que le champ en question est fixé par la base de données et non pas par le développeur.

Le fichier de configuration d'hibernate est situé dans le répertoire relatif conf/hibernate.cfg.xml

Une fois une session obtenue, utiliser hibernate pour effectuer des requêtes est très simple d'utilisation.

### 6.2.5. Utilitaires

Les utilitaires représentent des classes et méthodes utilisées par plusieurs contrôleurs et sont situés dans le package utils.

La classe SessionHandler, décrite dans le chapitre 6.2.4, en fait partie puisqu'elle est utilisée dans tous les contrôleurs.

La classe Util, dans laquelle le plus de méthodes utilitaires sont situées, permet notamment de :

- Créer ou récupérer l'identificateur d'un Location, selon un nom de ville, de province et de pays. S'il n'y a pas une entrée dans la base de données correspondant spécifiquement aux paramètres donnés, elle est créée puis son identificateur est retourné. Si le lieu existe déjà, l'identificateur du lieu est retourné.
- Récupérer un objet de type sql.Date depuis une chaîne de caractères au format « yyyy-mm-dd ».
- Récupérer tous les médias associés à un évènement.
- Créer ou récupérer un Singletime ou Timeinterval depuis un tableau d'une ou deux dates.
- Ajouter une visibilité à une entité, en passant en paramètre l'entité et la visibilité sous format Json.
- Vérifier les droits d'accès d'un utilisateur à une entité
- Récupérer le type d'une entité (Profil, Evenement, Relation, Parent, Media, Compte)
- Récupérer le propriétaire d'un profil

### 6.2.6. Téléchargement de fichiers vers le serveur

Le téléchargement de fichiers vers le serveur depuis le client se fait également par le biais du protocole http, sous le type multipart/form-data[35]

Lorsqu'un fichier est téléchargé vers le serveur, il est automatiquement enregistré par le framework dans un dossier de fichiers temporaires. Il a donc fallu implémenter l'enregistrement des fichiers dans le bon dossier afin qu'ils soient accessibles depuis le client.

Le type du fichier est détecté depuis les métadonnées du fichier, puis il est copié avec la bonne extension dans le dossier « public ». Le nom du fichier est généré par le code, comme étant le hashcode[36] du fichier suivi de son extension.

Après que le fichier soit enregistré, la réponse que l'émetteur reçoit contient le chemin vers le fichier ainsi que son type.

Par exemple, lorsqu'un nouvel évènement va être créé depuis le client, tous les fichiers sont d'abord téléchargés. Ensuite, le client ajoutera à l'évènement la liste de media, contenant le format et le chemin vers ceux-ci.

La configuration du téléchargement limite la taille des fichiers à 1GB et supporte les images, vidéos et fichiers audio.

[35] https://www.ietf.org/rfc/rfc2388.txt
[36] https://fr.wikipedia.org/wiki/Java_hashCode()

## 6.3. Client

### 6.3.1. Classes et Components

Plusieurs classes et « components », c'est-à-dire des objets ayant une vue, un modèle et des méthodes, sont utilisés. La particularité qu'a un component est qu'il peut être affiché en utilisant une balise html qui lui sera spécifique :

Par exemple, un component nommé « A » peut définir son sélecteur comme étant « A », signifiant que sa création depuis le DOM sera de la forme « <A></A> ».

Entre ces balises sera ensuite rendue la vue du component en question. L'utilité unique qu'offre un component est de pouvoir être créé depuis le DOM.

Pour l'instant, les profils, les relations et les parents sont séparés en une classe et un component, possédant à son tour sa propre classe.

Pour illustrer le concept de component d'Angular2, attardons-nous sur l'instanciation et l'affichage d'un nœud.

#### 6.3.1.1. Node & NodeVisual

Un Node représente un profil et contient les données suivantes :

```
import * as globals from '../../globals'

export class Node {
  // Node x coordinate
  x: number;
  // Node y coordinate
  y: number;
  // Image path
  image: string;
  // Node id, obtained from server data
  id: number;

  firstname: string;
  lastName: string;

  constructor(id: number, image, firstName: string, lastName: string) {
    this.id = id;
    this.image = !image ? globals.defaultAvatar : image;
    this.firstname = firstName;
    this.lastName = lastName;
    this.x = this.y = 0;
    this.width = 60;
    this.height = 60;
  }
}
```

Un Node peut être instancié de manière « classique » de la programmation orientée objet :

```
        let jack: Node = new Node(1, null,"Jacques", "Chirac")
```

Le component NodeVisual est défini comme suit :

```
import { Inject, Component, Input, Output,
         EventEmitter, ChangeDetectorRef,
         HostListener, ChangeDetectionStrategy
        } from '@angular/core';
import { Node } from '../../../d3';
import { TreeComponent } from '../../../tree/tree.component'
import { MdDialog, MdDialogRef, MD_DIALOG_DATA } from '@angular/material';
import { ProfileDialog } from './profileDialog'
import * as globals from '../../../globals';
@Component({
  selector: '[node]',
  templateUrl: './node-visual.component.html',
  styleUrls: ['./node-visual.component.css']
})
export class NodeVisualComponent {
  @Input('node') node: Node;
  globals;
  constructor(public dialog: MdDialog) {
    this.globals = globals;
  }

  openDialog() {
    let dialogRef = this.dialog.open(ProfileDialog, {
      data: this.node
    });
  }
}
```

La notation @Component permet d'attribuer des attributs propres à un Component, soit son sélecteur, le chemin vers sa vue et son style.

Ici, le sélecteur étant défini comme 'g[node]' indique que la balise à utiliser devra être

```
        <g [node]></g>
```

Le paramètre d'entrée du constructeur, « dialog » est un objet qui sera injecté par Angular2 lors de l'exécution.

La notation '@Input' indique qu'un paramètre d'entrée peut être donné lorsque le component est créé depuis le DOM. Ici,

```
  @Input('node') node: Node;
```

Indique que le paramètre d'entrée sera identifié par un attribut html nommé 'node', qu'il est de type Node et que le modèle l'enregistre dans un attribut nommé 'node'.

Utilisation :

```
        <g [node]="jack"></g>
```

La raison pour laquelle la balise commence par la lettre g est due à l'utilisation d'éléments SVG et sera expliqué en détail à la partie 6.3.2.1

Le template de la vue d'un NodeComponent est défini comme suit :

```
<svg:g>
    <defs>
        <pattern [attr.id]="node.id" height="100%" width="100%" viewBox="0 0
60 60">
            <image [attr.x]="0" [attr.y]="0"
[attr.width]="globals.nodeImageWidth" [attr.height]="auto"
[attr.xlink:href]="node.image"></image>
        </pattern>
    </defs>
    <svg:circle class="node" [attr.fill]="'url(#' + node.id+')'"
[attr.cx]="node.x" [attr.cy]="node.y" [attr.r]="globals.nodeRadius">
    </svg:circle>
</svg:g>
<svg:text class="node-name" [attr.x]="node.x" [attr.y]="node.y +
globals.nodeRadius + node.fontSize/2" [attr.font-size]="node.fontSize">
    {{node.firstname[0]}}
</svg:text>
```

La notation entre doubles accolades à l'avant dernière ligne signifie que l'on accède à un attribut du modèle. Ici, {{node.firstname[0]}} fait référence à la première lettre de l'attribut firstname du Node qui a été donné en paramètre d'entrée à l'instanciation du NodeVisualComponent. L'utilité de séparer une classe Node d'un component NodeVisualComponent vient du fait que des objets Node peuvent être créés depuis le code d'une fonction et maintenus séparés d'une vue.

En revanche, il n'est pas possible de créer dans du code TS une instance d'un component et de l'afficher individuellement en utilisant son sélecteur.

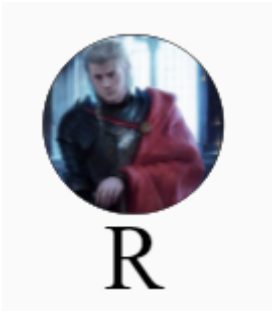



*Figure 7- NodeVisualComponent créé à partir d'un objet de type Node*

### 6.3.2. Visualisation

Nous traiterons ici de la visualisation de l'arbre généalogique, axe principal de ce projet. Nous développerons ensuite les dialogues mis à disposition de l'utilisateur pour interagir avec l'arbre généalogique.

#### 6.3.2.1. Composants & SVG

Afin d'afficher un arbre généalogique, le choix s'est porté vers la librairie JavaScript D3.js et une représentation de l'arbre en format SVG. L'avantage du format SVG est qu'il apporte la possibilité pour l'utilisateur d'adapter la vue des éléments graphiques sans affecter leur résolution.

Un tutoriel sur la visualisation et l'utilisation de graphes non-orientés utilisant D3.js en Angular2 a été suivi[37]. Proposant de développer un graphe où des forces s'appliquent entre les nœuds et où ils sont déplaçables librement dans la vue, seules les fonctionnalités nécessaires au projet (zoom et déplacement de l'arbre entier) ont été retenues.

L'astuce qu'apporte la méthode de l'auteur est de définir les sélecteurs des components de l'arbre dans un format interprétable en SVG

```
  selector: '[node]',
```

Entourer le nom du sélecteur de crochets permet de confondre sélecteur et attribut du component Angular.

[37] https://medium.com/@lsharir/visualizing-data-with-angular-and-d3-209dde784aeb

Créer ensuite une instance de NodeVisualComponent se fait de la manière suivante :

```
        <g [node]="node" *ngFor="let node of nodes"></g>
```

La syntaxe « *ngFor » est une syntaxe d'Angular2 signifiant que pour chaque élément, nommé ici « node » contenu dans le tableau « nodes », une balise de cette sorte sera créée.

La balise 'g' fait comprendre au navigateur qu'à l'intérieur des balises se trouveront des éléments SVG[38].

Sachant que le template du NodeVisualComponent contient un cercle svg, un rectangle svg, du texte svg et une balise <defs>[39] (un indicateur qui définit que l'élément à l'intérieur sera réutilisé plus tard dans le document, en l'occurrence dans le cercle svg afin d'afficher la photo de profil de l'utilisateur), aucune erreur d'affichage n'aura lieu.

Le même principe est utilisé pour l'affichage des relations (LinkVisualComponent) :

```
<svg>
      <image class="linkIcon" (click)="showMore()" [attr.x]="link.middle.x -
8" [attr.y]="link.middle.y - 18" [attr.xlink:href]="link.icon" height="16"
width="16"/>
<line class="link" [attr.x1]="link.source.x" [attr.y1]="link.source.y"
[attr.x2]="link.target.x" [attr.y2]="link.target.y"></line>
</svg>
```

[38] https://developer.mozilla.org/en/docs/Web/SVG/Element/g
[39] https://developer.mozilla.org/en-US/docs/Web/SVG/Element/defs

Et également pour l'affichage des liens parents :

```
<svg>
    <line (click)="showMore()" *ngIf="parent.type === 0"
style="stroke:rgb(255,255,255); cursor: pointer;" class="link"
[attr.x1]="parent.x1"
        [attr.y1]="parent.y1" [attr.x2]="parent.x2" [attr.y2]="parent.y2">
    </line>
    <image class="linkIcon" (click)="showMore(parent.id)"
[attr.x]="(this.parent.x1 + this.parent.x2) / 2 - 20"
[attr.y]="(this.parent.y1 + this.parent.y2) / 2"
        [attr.href]="icon.path" height="16" width="16" />
    <line *ngIf="parent.type !== 0" stroke-dasharray="5, 5"
style="stroke:rgb(255,255,255); cursor: pointer;" class="link"
[attr.x1]="parent.x1"
        [attr.y1]="parent.y1" [attr.x2]="parent.x2" [attr.y2]="parent.y2">
    </line>
</svg>
```

La visualisation de l'arbre se fait ensuite à travers le component "TreeComponent", et est définie ainsi :

```
<svg #svg width="100%" [attr.height]="height">
    <g [zoomableOf]="svg">
        <g [linkVisual]="link" *ngFor="let link of visibleRelationships"></g>
        <g [parentVisual]="parent" *ngFor="let parent of visibleParents"></g>
        <g [node]="node" (contextmenu)="onRightClickEvent($event, node)"
*ngFor="let node of visibleNodes"></g>
    </g>
</svg>
```

L'attribut « [zoomableOf] » est un attribut liant à un component dont la fonction est de permettre le zoom/dézoom de l'arbre.

visibleNodes, visibleParents et visibleRelationships contiennent les données qui doivent être rendues. Si la timeline est activée, leur contenu est défini comme étant les nœuds, liens et parents filtrés selon leurs dates.

#### 6.3.2.2. Algorithme de placement des nœuds

Un des aspects les plus complexes de l'application client est le placement des profils afin d'avoir une structure cohérente pour l'utilisateur.

Sachant que l'arbre possède trois tableaux :

- Un tableau de personnes, contenant des Nodes
- Un tableau de relations, contenant des Relationship (classe ayant pour attributs les identifiants des Nœuds qu'elle relie et les coordonnées de son milieu, les extrémités étant les coordonnées x et y des nœuds)
- Un tableau de parents, contenant des Parentcomponent (ayant pour attributs l'identifiant de l'enfant ainsi que l'identifiant d'un nœud (en cas de parent unique) ou l'identifiant d'une relation (lorsque deux personnes adoptent ou ont un enfant biologique ensemble).

Les coordonnées des relations et des parents étant héritées des coordonnées des personnes, la structure des éléments de l'arbre se basent sur les coordonnées qui seront attribuées aux personnes.

L'algorithme de placement des nœuds se fait de la manière suivante :

```
Soit personnesRestantesAPlacer = personnes (Node[])
Soit parentsRestantAPlacer = parents (Parent[])
Soit relations les relations liant les personnes (Relationship[])
niveaux: Node[][] = []
niveauCourant = 0

tant que personnesRestantesAPlacer n'est pas vide:
    pour chaque personne p de personnesRerstantesAPlacer:
        Si parentsRestantAPlacer ne contient ni un parent dont l'enfant est p,
        ni un parent dont l'enfant est une des personnes avec lesquelles p est en relation
        (cherché récursivement ensuite avec les relations des relations de p etc...):
            // p est situé au niveau courant.
            ajouter p à niveaux[niveauCourant]
            retirer p de personnesRestantesAPlacer
    Retirer de parentsRestantAPlacer les entités dont le parent se situe à ce niveau
    niveauCourant = niveauCourant + 1

Trier chaque niveau de sorte à ce que des personnes ayant une relation soient côte à côte dans le
tableau

Soit largeurMax la taille maximale d'une entrée des niveaux (l'étage avec le plus de personnes)
Soit hauteurMaximale le nombre de niveaux
Soit décalageHorizontal = 100
Soit décalageVertical = 200

Soit décalage = (largeur de la fenetre) - (largeurMax*décalageHorizontal + largeurMax * 30 (rayon d'un
noeud))

Pour i de 0 à niveaux.taille
    Pour j de 0 à niveaux[i].taille
        niveaux[i][j].x = décalage + largeurMax / niveaux[i].taille * j * 100 + 100
        niveaux[i][j].y = i * décalageHorizontal + 50
```

#### 6.3.2.3. Lazy-loading de l'arbre généalogique

Lorsque l'arbre est initialement chargé, sa configuration sera récupérée du serveur dépendant du nœud que l'utilisateur souhaite observer (lui-même par exemple).

Le serveur envoie au client les informations de la personne, ses relations, ses parents, ses enfants et ses frères et sœurs.

Lorsque l'utilisateur souhaite étendre l'arbre d'un des membres actuels et l'inclure, une nouvelle requête sera formulée vers le serveur en indiquant l'identifiant de la personne dont l'arbre souhaite être récupéré. Ensuite, les données qui ne sont pas actuellement incluses dans l'arbre sont ajoutées, et l'algorithme de placement est exécuté à nouveau.

A ce stade, un service Angular2 injectable se charge de récupérer les données relatives à une personne depuis un fichier JSON contenant les données de personnes, de relations et de parents.

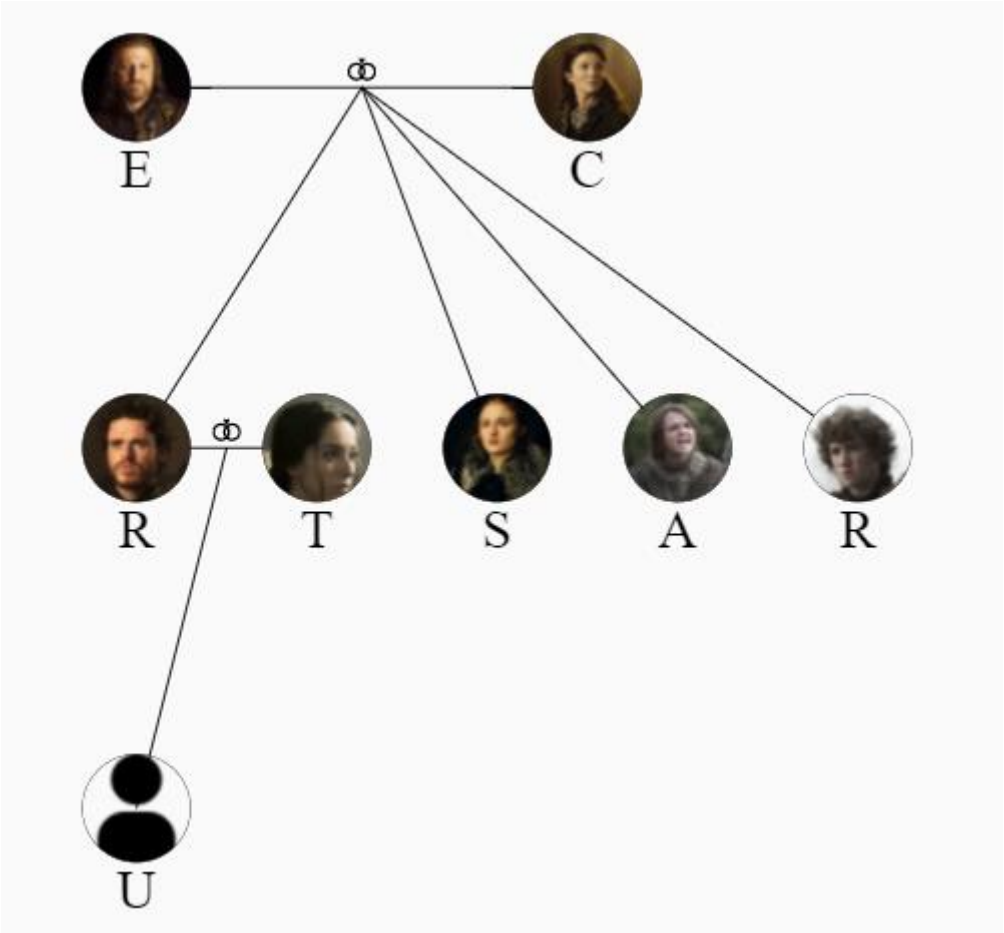



*Figure 8- Exemple d'arbre récupéré*

#### 6.3.2.4. Timeline

Une fonctionnalité supplémentaire a été ajoutée, dont le but est de permettre à l'utilisateur de visualiser l'évolution de l'arbre à travers le temps.

Lorsque des données sont ajoutées par le client ou récupérées par le serveur, leurs dates sont insérées dans un tableau.

Lorsque l'utilisateur active la timeline, un « slider » apparaît, et permet à l'utilisateur de le glisser entre la date la plus ancienne et la date la plus récente de tous les éléments de l'arbre. Sans perdre les données, la vue se met à jour selon les éléments qui sont présents à une date précise (par exemple, une relation ne sera visible que lorsque le slider est fixé à une date à laquelle la relation existe).

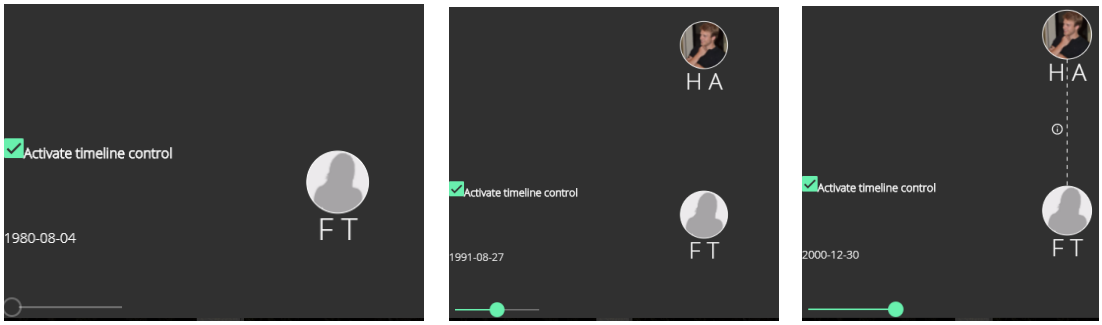



*Figure 9- Evolution de la timeline*

Dans cet exemple, la première date est le 04 août 1980. A cette date, seulement une personne existe. A la deuxième, les deux profils existent et à la troisième leur lien apparaît.

L'idée de cette fonctionnalité est venue lorsque les arbres développés devenaient saturés par le nombre de relations.

#### 6.3.2.5. Visualisation d'un profil et autres dialogues

Des interactions avec l'arbre sont possibles :

- Lorsqu'un clic droit est effectué sur une personne, son arbre est importé.
- Lorsqu'un clic gauche est effectué sur une personne, une relation ou un lien de type parent, ses détails et évènements sont chargés et affichés dans un dialogue :

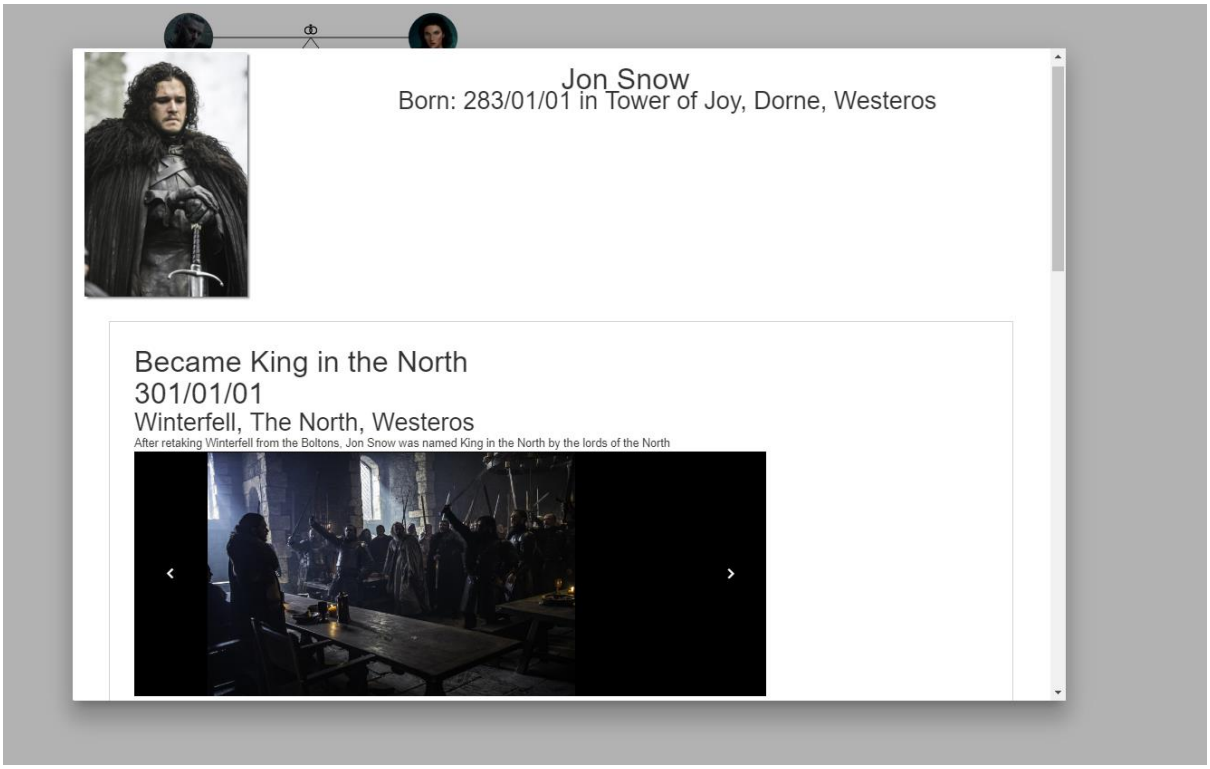



*Figure 10- Exemple de la vue détaillée d'une personne*

Si l'utilisateur est propriétaire du profil, de la relation ou du lien parent affiché dans le dialogue, un bouton supplémentaire est affiché afin de permettre son édition ou suppression. Il est à préciser que l'édition d'un profil se fait depuis une page spécifique aux profils, qui permet également d'ajouter des évènements à celui-ci :

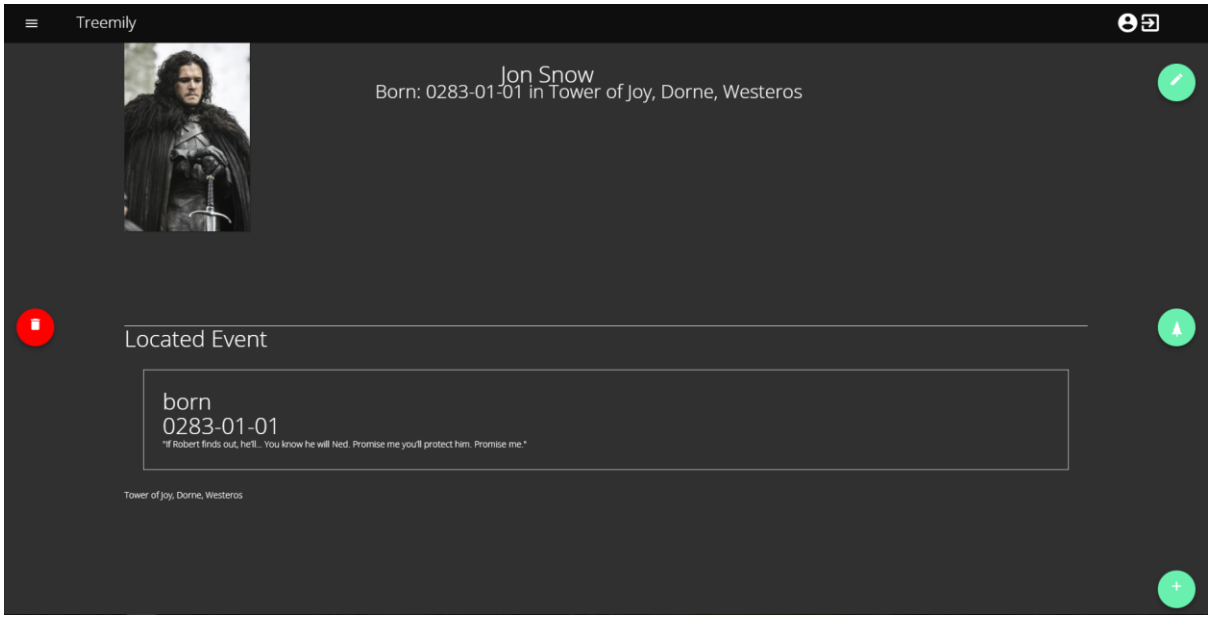



*Figure 11- Vue administrateur d'un profil*

- Des dialogues permettent également d'ajouter des personnes, relations et parents depuis l'interface graphique. A chaque nouvelle création, les coordonnées de tous les membres de l'arbre sont rechargées.

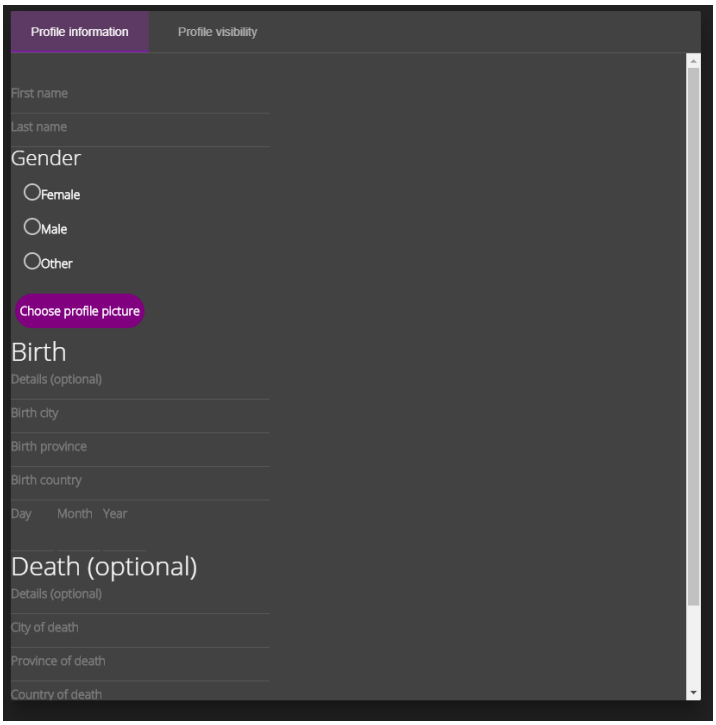



*Figure 12- Formulaire pour créer un profil*

La visibilité d'un profil, d'une relation ou d'un parent se fixe dans le formulaire de création. Lors de la modification d'une de ces entités, il est également possible de redéfinir sa visibilité.


*Figure 13- Configuration de la visibilité*

La création de groupe se fait depuis une page spécifique :


*Figure 14- Gestion des groupes*

Lorsque l'utilisateur entre le prénom d'une personne dans le champ « Include person », une recherche s'effectue et la liste des profils correspondants à la requête s'affiche. L'utilisateur doit ensuite sélectionner celui qu'il souhaite.

Les pages dédiées aux réclamations et aux notifications sont formées de manière similaire :

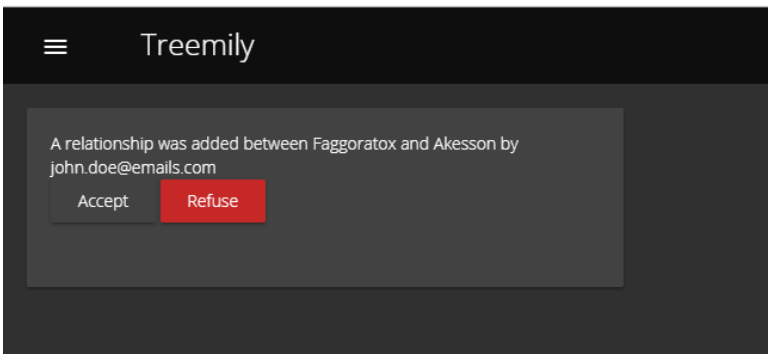



*Figure 15- Notifications*

Concernant les réclamations, deux cas sont à gérer :

- Si le propriétaire du profil réclamé refuse la demande, la notification propose de créer un nouveau profil directement depuis un dialogue qui apparaît.
- Si le propriétaire a accepté la demande, l'utilisateur doit se reconnecter pour mettre à jour ses données.

# 7. Problèmes existants, améliorations

Il reste plusieurs aspects qui peuvent être améliorés :

- La gestion des tags et pendingTagged n'a pas été implémentée
- Les commentaires ne peuvent être créés que pour des évènements.
- L'algorithme de placement des nœuds n'est pas satisfaisant pour une mise en production.
- Une photo de profil n'est pas toujours bien cadrée dans le nœud. Un moyen de pouvoir rogner l'image avant le téléchargement permettrait de résoudre cet aspect.
- L'application n'est pas responsive.
- Le rendu graphique de l'arbre n'affiche pas les photos de profil dans les nœuds sur les navigateur Mozilla Firefox et Microsoft Edge.
- Le style peut être amélioré, notamment les évènements.
- Les mots de passe sont stockés en clair dans la base de données
- Un système de jetons comme JWT[40] permettrait de gérer la connexion et assurerait qu'une tierce personne ne puisse pas effectuer des requêtes au nom d'un autre utilisateur.

[40] https://jwt.io

# 8. Conclusion

Le projet en son état actuel est relativement abouti. Le client, le serveur et la base de données communiquent correctement et la majorité des fonctionnalités prévues sont incluses. Je pense que quelques semaines supplémentaires permettraient de le finaliser, remédier aux problèmes de sécurité et améliorer le design de l'application client.

Le sujet est très intéressant, exigeant et m'a permis de mettre en œuvre beaucoup de concepts appris lors de la formation : modélisation de données, paradigmes de programmation divers, théorie des graphes et de manière générale l'ingénierie logicielle.

Au fur et à mesure de l'avancement, ma motivation et mon implication dans le projet n'ont fait qu'augmenter et poursuivre son développement dans un cadre open-source serait envisageable.

# 9. Bibliographie et références


- Angular2 : https://angular.io/
- Angular Material : https://material.angular.io/
- Play framework: https://www.playframework.com/
- D3js: https://d3js.org/
- Sharir, Liran : Visualizing Data with Angular and D3: https://medium.com/@lsharir/visualizing-data-with-angular-and-d3-209dde784aeb
- Documentation PostreSQL: https://docs.postgresql.fr/9.6/
- Documentation hibernate : http://hibernate.org/