Rapport de stage

Développement d'une plateforme de réseau social

Alexandre DEMODE

Licence Professionnelle SIL DA2I
IUT A
Université Lille 1
Avril - Juin 2014

Remerciements

Je tiens à remercier MM. Benoît Thieffry et Mathieu Froidure de m'avoir permis d'effectuer mon stage au sein de leur équipe. Equipe que je remercie également pour son accueil, sa bonne humeur et ses conseils durant ces trois mois.

Merci également à M. Michel Hoël, maître de stage, et M. Bruno Beaufils, tuteur de formation, pour leurs retours et leur temps accordé au cours du stage.

Résumé

Mon stage de fin de Licence Professionnelle SIL DA2I (Systèmes Informatiques et Logiciels, option Développement et Administration de site Internet et Intranet) s'est déroulé au sein de l'entreprise *Urbilog* à Roubaix (puis Villeneuve d'Ascq, après déménagement). L'entreprise a fait de l'accessibilité son coeur de métier et développe, entre autres, le logiciel *expertfixer*. En version 1 lors de mon stage, c'est un outil d'analyse et d'audit de site Web qui permet d'obtenir un rapport des points à améliorer pour arriver à un niveau d'accessibilité correspondant au référentiel souhaité. Durant ces trois mois de stage, j'ai intégré l'équipe de recherche et développement (R&D) qui met au point *expertfixer* depuis plusieurs années.

Ma mission : développer une plateforme de réseau social autour d'*expertfixer*. Le réseau aurait, à terme, pour but de permettre la collaboration sur les audits d'*expertfixer* ainsi que le partage d'expérience entre entreprises autour de la problématique de l'accessibilité numérique et tout ce que cela engendre.

Mots-clés : API, REST, composant, accessibilité, ACL, collaboration, R&D.

Abstract

My internship Professional License SIL DA2I (Computer Systems and Software Development and Administration of Internet and Intranet option) took place within the *Urbilog* company at Roubaix (Villeneuve d'Ascq after removal). The company has made access its core business and development, and especially, *expertfixer* software. In first version during my internship, it is an analysis and audit website tool to get a report with points to improve to reach an accessibility level corresponding to the desired reference tool. During these three months of internship, I joined the research and development team which develops *expertfixer* for several years.

My mean mission has been to develop a social networking platform around *expertfixer*. The network would, eventually, to enable collaboration on *expertfixer* audits and sharing of experiences between companies around the issue of digital accessibility and all it engenders.

Keywords : API, REST, component, accessibility, ACL, collaboration, R&D.

Table des matières

1 Introduction

Le sujet de stage étudié vise à développer une plateforme de réseau social qui sera proposée sous forme d'un SaaS (Software as a Service, logiciel en tant que service) aux entreprises utilisant *expertfixer*. Le fait de savoir que la plateforme sera uniquement hébergée et gérée par *Urbilog* a laissé le champ libre en terme de choix de technologies et de langages. C'est d'un commun accord que *node.js* a été choisi comme base du système.

Les objectifs initiaux de ce projet pour le stage étaient les suivants :

- me familiariser avec l'éco-système *node.js* qui m'était totalement inconnu lors de mon arrivée ;
- appliquer les connaissances acquises au cours de ma formation à un projet réel de façon à mieux comprendre leurs intérêts et bénéfices ;
- créer une application accessible.

J'ai également souhaité découvrir les autres activités de l'entreprise pour mieux comprendre son fonctionnement et ses principes. J'imaginais relativement bien ce qu'était le développement mais ne voyais pas exactement ce qu'était la R&D. Ce stage m'aura permis de le découvrir.

Je commencerai par présenter *Urbilog*, son organisation, ses activités et ses produits.

J'aborderai ensuite le projet, l'environnement et les outils de travail que j'ai utilisé pour construire cette application.

Enfin, j'exposerai plus en détails la réalisation même du projet, ainsi que les difficultés rencontrées m'amenant à résumer les compétences acquises durant ce stage.

2 L'entreprise : Urbilog

2.1 Présentation

Histoire de l'entreprise

Urbilog est une SAS[1] créée en 1995. D'un petit restaurant proche de la cité universitaire de Lille 2, il a évolué au fil des années jusqu'à se reconvertir, en 1998, en une société concentrée et spécialisée autour de l'accessibilité numérique qui est alors très peu en vogue. À cette époque, l'Internet n'est que balbutiant et les investisseurs ne croient pas aux projets qui y sont liés, et c'est notamment le cas d'*Urbilog* qui a eu bien du mal à débuter financièrement parlant.

Aujourd'hui, *Urbilog* est une entreprise reconnue dans son domaine de prédilection et travaille avec les plus grands groupes français : *Vivendi*, *Bouygues*, *Orange*, *Auchan* ou encore des organisations publiques telles que la *Bibliothèque nationale de France* (*BnF*).

Composition

L'entreprise est composée de 12 salariés, dont voici l'organisation :

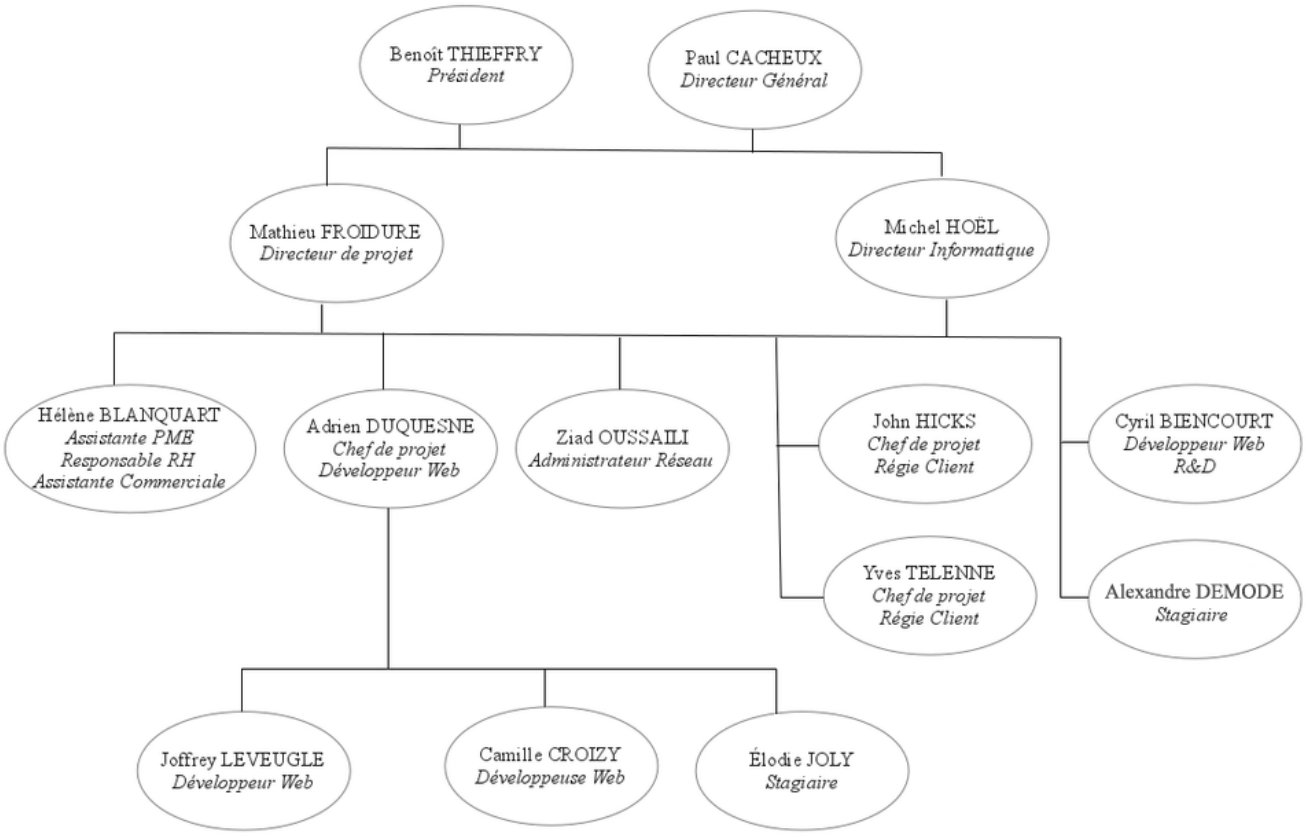

Fig. 1 : Organigramme de l'organisation d'Urbilog

[1]SAS : Société par Actions Simplifiées

Chiffres clés

Partant d'un capital de 120 000 € peu après leur reconversion de 1998, *Urbilog* atteint aujourd'hui un capital estimé à 220 000 €. L'entreprise réalise un chiffre d'affaire annuel d'environ un million d'euros.

2.2 Activités

Urbilog a su évoluer au fil des années, a grandi et propose aujourd'hui plusieurs services ayant tous l'accessibilité comme centre d'intérêt.

2.2.1 Accessibilité numérique

Qu'est-ce que l'accessibilité numérique ?

L'accessibilité numérique est le fait de permettre à toute personne d'utiliser, comprendre et interagir avec le support numérique en question. Il peut s'agir d'un site Web, une application sur *smartphone* ou *tablette*, d'un *ebook* ou encore de simples documents textes qui doivent être enrichis pour être compris par certaines personnes ayant perdu la vue ou l'ouïe.

Le rôle de l'accessibilité numérique au sein Urbilog

C'est le cœur de métier, mais surtout la grande spécialité et l'expertise d'*Urbilog*. Il s'agit d'étudier les problématiques d'accessibilité au niveau technique (contrastes des couleurs, tailles de polices d'écriture, textes alternatifs, etc.) qui sont quelques critères parmi des centaines d'autres. Ces critères permettent d'évaluer la qualité d'un site Web ou une application en terme d'accessibilité afin de rédiger des rapports aux équipes techniques concernées. Ces dernières devront alors mettre à jour leur production.

Services proposés autour de l'accessibilité

Les audits sont une part importante du travail d'*Urbilog*. Les clients font souvent appel à eux pour expertiser et auditer les systèmes développés par d'autres entreprises. Il est généralement question de mettre à niveau le système audité par rapport à un des nombreux référentiels et de leurs degrés de validation. L'analyse peut être plus ou moins précise, conduisant à la rédaction d'une sythèse, d'un cahier de route ou d'un cahier technique.

Urbilog sensibilise ses partenaires et ses clients à l'accessibilité au niveau technique mais aussi en ce qui concerne la compréhension de l'utilité de rendre les outils numériques accessibles.

Au delà de la sensibilisation, *Urbilog* propose trois types de formations :

- accessibilité : forme les graphistes, les développeurs et les rédacteurs à l'accessibilité afin de les rendre au maximum autonomes ;
- déficients visuels : formation aux outils fréquemment utilisés par les malvoyants afin de pouvoir faire des tests en direct pendant le développement et gagner ainsi en productivité en réduisant le nombre d'appels aux experts d'*Urbilog* ;
- 360 degrés : elle reprend tous les modules de la formation à l'accessibilité numérique, segmentés sur un tronc commun puis découpés par métier, de manière à permettre aux chefs de projets d'acquérir une hauteur de vue nécessaire, sur les impacts et les recommandations auprès de leurs équipes.

2.2.2 Développement Web

Au delà de l'analyse du produit d'autres entreprises, *Urbilog* développe des sites Web, accessibles, pour différents clients. Tous types de projets Web sont pris en charge, allant de la conception à la réalisation d'outils de communication interactive, de l'application Web la plus simple à la plus évoluée : Internet, intranet ou extranet. Grâce à ses partenaires, *Urbilog* propose également des services connexes au développement Web : hébergement, noms de domaines, …

2.2.3 Applications mobiles et Web mobile

Le numérique est mobile aujourd'hui, *Urbilog* développe des applications mobiles et des sites Web *responsives* (à largeur fluide) adaptés aux supports mobiles que sont les smartphones et tablettes. On pourra trouver des applications pour *iOS* ainsi que pour *Android* ; mais aussi et surtout une certaine expertise dans l'analyse d'applications mobiles développées par des entreprises dont c'est le cœur de métier.

2.2.4 Réseau

Le pôle réseau d'*Urbilog* propose des solutions d'installation et de configuration réseau dans le nord de la France. Il propose une élaboration et une mise en œuvre d'infrastructures réseaux, mais également l'installation et la configuration de serveurs de données et de sauvegarde. Les entreprises peuvent également demander auprès d'*Urbilog*, une élaboration de stratégies de sauvegarde client-serveur, ou encore la mise en place et la configuration de *VPN*[2]. Il est également possible d'installer des petits parcs informatiques, de mettre en place un proxy ou de configurer le réseau *Wi-Fi*.

2.2.5 Recherche et développement

C'est dans ce service qu'est développé *expertfixer*, mais aussi d'autres produits en partenariat avec d'autres entreprises soucieuses de l'accessibilité de leurs applications numériques. D'autres outils sont développés dans ce pôle, notamment un clavier virtuel accessible, une participation active au projet *CorrecT* avec la *BnF* (Bibliothèque nationale de France), des outils de validation et de contrôle de flux HTML, ainsi que d'autres outils qui ne sont plus maintenus à l'heure actuelle (notamment *OCAWA*, cf. ci-dessous).

2.3 Produits

Les produits développés par *Urbilog* sont diversifiés. Les plus importants sont explicités rapidement ici.

OCAWA

Développé en partenariat avec *Orange*, *OCAWA* avait pour but d'analyser les pages Web et de fournir un rapport technique pour augmenter le niveau d'accessibilité du site concerné.

AWADAO !

Une application mobile qui permet de localiser les lieux accessibles les plus proches. Principalement dédiée aux handicapés, cette application est utilisable par tous. L'application fonctionne sur les *smartphones* et sur les *tablettes* pour permettre aux personnes handicapées de trouver les lieux et les services accessibles plus facilement. Les informations sont maîtrisées et contrôlées par la structure (collectivité, entreprise) offrant le service.

[2]VPN : *Virtual Private Network*, littérallement *réseau privé virtuel*

Clavier virtuel

C'est un moyen d'accès pour toute personne à ses comptes bancaires de façon sécurisée via Internet. *Urbilog* a développé une solution accessible aux personnes handicapées et aux seniors. Ce clavier virtuel peut être intégré dans la plupart des sites web des organismes bancaires sans altérer le niveau de sécurité initial de l'application.

SOURDLINE Développement

Le 1er centre d'appels spécialisé à destination des sourds et malentendants. Ce centre d'appels a la caractéristique de communiquer en langue des signes. C'est un service très particulier qui est assuré par cette filière pour de grandes enseignes françaises.

expertfixer

C'est un logiciel permettant de faire des analyses de pages web par rapport à un référentiel donné (référencement - *SEO*, accessibilité - *RGAA*, qualité - *Opquast*, …).

3 Le projet

3.1 Présentation

expertfixer est un client lourd. Il n'est pas possible de collaborer lors de la réalisation d'un audit. C'est pour palier à ce manque que le projet est né. Il doit permettre la réalisation d'audits collaboratifs et donner la possibilité d'échanger via des outils communs tels qu'une messagerie instantannée, le partage et le commentaire de messages publics sur la plateforme, voir l'état des audits et la possibilité d'étendre la plateforme au moyen de greffons.

3.2 Objectifs

Le projet est avant tout une preuve de concept (POC[3]).

L'objectif principal, au delà de la réalisation du projet, était de valider sa faisabilité, ainsi que la possibilité d'utiliser les outils récents que sont *node.js* et *MongoDB* qui m'ont été imposés. Ces outils sont puissants dans leurs domaines respectifs. Il s'agissait donc de les faire fonctionner ensemble de façon à proposer une première piste fonctionnelle au projet global. Cette piste étant très légère, il faudra également trouver les outils ou composants qui pourraient fonctionner ensemble pour réutiliser un maximum de modules existants tout en ayant un système parfaitement adapté aux besoins.

Il fallait également permettre l'intégration par composant de la plateforme dans *expertfixer* ainsi que la connexion et les interactions directes du client lourd vers la plateforme du réseau social. C'est ainsi qu'il a été décidé de s'orienter vers la mise en place d'une *API RESTful*[4].

3.3 Outils et technologies

Au delà des technologies et outils imposés, il aura fallu construire tout l'environnement de développement pour réaliser le projet dans de bonnes conditions.

3.3.1 Environnement

Le poste de travail est une chose à prendre au sérieux et à adapter en fonction de ses besoins et ceux du projet. Certains outils ou logiciels m'ont été imposés, d'autres ont été choisis par habitude ou suite à des recherches durant le développement du projet.

Fedora 20

Lorsque je suis arrivé, la machine qui m'a été fournie était sur *Fedora 20*. Etant habitué à la famille *Debian*, j'ai du m'y acoutumer ce qui fût relativement rapide. J'ai pu découvrir et utiliser un autre gestionnaire de paquet : *YUM*.

Sublime Text 2

Certainement mon éditeur de texte favori lorsqu'il s'agit de développer des projets relativement lourds. *Emacs* ou *Vim* auraient également pu être utilisés, mais pour des raisons d'habitude j'ai choisi de rester sur *Sublime Text*. Comme tout bon éditeur, une multitude de modules est disponible et j'ai principalement utilisé ceux-ci :

- *Emmet* : auto-complétion avancée pour *HTML* et *CSS* ;

[3] POC : *Proof Of Concept*, littéralement *preuve de concept*
[4] API REST : Application Programming Interface, REpresentational State Transfer

- *Emmet LiveStyle* : outil permettant d'avoir le rendu synchronisé en direct entre *Chromium* et *Sublime Text* ;
- Modules de coloration et d'auto-complétion pour : *Twig*, *Jade* et *CoffeeScript*.

Chromium

Chromium est un navigateur web très complet, notamment en terme d'outils de développement. J'ai pu profiter de beaucoup d'extensions qui seront détaillées un peu plus loin, mais on peut mentionner la qualité de la barre pour développeurs de *Chromium* qui est vraiment une très bonne base pour le débogage des requêtes, vérifier la performance du rendu des pages et les tests à la volée sur le contenu de la page.

SVN

À l'heure actuelle, tous les projets se doivent d'être versionnés. L'entreprise utilise *SVN*[5] et j'ai donc du m'y plier malgré une préférence pour son homologue *git*. *SVN* m'aura permis de garantir une sauvegarde régulière de mon travail avec un historique qui donne la possibilité de retrouver les modifications effectuées. Cela permet parfois de comprendre des bogues ou de revoir son code dans une ancienne version.

3.3.2 Côté serveur (back-end)

node.js et npm

node.js était une technologie qui m'était quasiment inconnue à mon arrivée. J'ai du apprendre à la découvrir avec son gestionnaire de paquets : *npm*. J'ai pu voir l'ampleur déjà prise par *node.js* malgré son jeune âge et la quantité de paquets qui permettent de démarrer un projet très rapidement. La plupart des éléments qui vont suivre sont en réalité des paquets *npm*.

Express v4

Express est le *framework* le plus répandu pour développer des sites web dynamiques avec *node.js*. *Express* était en version 3 au début de mon stage, mais j'ai fait le choix de migrer vers la version 4 durant sa beta de façon à avoir la dernière version du *framework* et pouvoir profiter de certains modules qui vont suivre qui n'étaient compatibles qu'avec la version 4. *Express* englobe également un serveur *HTTP* bien construit qui permet d'avoir beaucoup de souplesse sur la gestion des ressources statiques entre autres.

PassportJS

Une grande partie du projet consiste à gérer des membres et leurs connexions. J'ai donc recherché un module qui permettait d'avoir un large panel de modes de connexion. Ainsi, *PassportJS* permet aisément de se connecter via *OAuth* (1.0 & 2.0), *OpenID*, possède des extensions pour s'adapter à l'*ORM* (Object-Relational Mapping) que l'on souhaite ou encore de gérer la connexion via des services tiers tels que *Facebook*, *Twitter* et *Google*. J'ai choisi d'utiliser plusieurs couches du module :

- *passport* : le noyau ;
- *passport-local* : la surcouche qui permet de gérer la connexion en interne, sans service tiers ;
- *passport-local-mongoose* : l'interface entre *passport-local* et la base *MongoDB*.

[5] SVN : Apache Subversion

async

C'est *la* librairie *JavaScript* à connaître : elle permet de gérer la parallélisation de tâches. Très simple à appréhender, *async* s'utilise côté client ou serveur, mais je n'en n'ai eu besoin que côté serveur. Avec celle-ci, on peut alors gérer la façon dont est exécuté notre code : plusieurs blocs en parallèle, à la suite, ou même combiner les deux. *async* permet d'autres possibilités sur la base du parallèle/chaîné.

CoffeeScript

Côté serveur, j'ai choisi de tenter l'expérience *CoffeeScript* pour alléger l'écriture du *JavaScript* qui peut parfois se révéler relativement lourde. Tout le côté serveur sera finalement codé en *CoffeeScript* ce qui m'a amené à chercher quelques solutions pratiques, notamment pour éviter de devoir compiler à chaque fois en *JavaScript* avant de relancer le serveur *node.js* et c'est là qu'intervient le prochain outil.

nodemon

Très pratique, cet outil surveille les modifications éventuelles des fichiers et de relance automatiquement le serveur en prenant les modifications en compte. C'est donc un démon pour *node.js*. Le gros avantage est que *nodemon* se charge de la transcription de *CoffeScript* en *JavaScript*.

Jade et TwigJS

Lors de mes premiers essais, afin d'assimiler le fonctionnement de *node.js* et surtout d'*Express*, j'ai commencé à développer le système côté serveur uniquement. Je n'ai pas tout de suite commencé à faire du *REST*. J'ai donc du opter pour un moteur de templates fonctionnant avec *Express*. Par défaut, c'est *Jade* qui est proposé, qui nous viens du monde *Ruby*. La syntaxe déplaisait à mon maître de stage, Michel Hoël et était parfois un peu complexe de par sa philosophie qui veut rendre son écriture légère. *Jade* ne se base pas sur une syntaxe surchargée du *HTML* avec du remplacement de variable et un peu d'algorithmique, mais propose tout un tas de raccourcis qui permettent de générer les balises *HTML* ainsi que leurs attributs. De plus, l'écriture est basée sur l'indentation ce qui peut être parfois difficile à lire. De par mon expérience avec *Symfony*[6] j'ai cherché un équivalent de *Twig*[7] en *JavaScript* et c'est alors que j'ai trouvé *TwigJS* qui est un portage du moteur en *JavaScript*. Ce module importait peu, il n'était là que pour appréhender *Express* et *node.js*. Les deux modules seront supprimés dès le début du développement de l'*API REST*.

enrouten

Une part importante du travail a été de gérer le routage proprement, ainsi que les droits. Pour des raisons de pratique et de maintenance, j'ai souhaité profiter d'un système de routage par routes nommées. Ainsi, on peut générer l'*URL* à partir de son nom et en lui passant les paramètres nécessaires. *Express* ne permet pas de base de nommer les routes, j'ai donc recherché un module qui le faisait, j'ai alors trouvé *enrouten* qui fait très bien son travail et même au delà puisqu'il permet de générer l'*URL* par rapport à la position du fichier du contrôleur dans l'arborescence des fichiers. Il ne gère par contre pas la possibilité de filtrer par rapport à un système de droits, j'expliquerai ci-dessous comment j'ai greffé la gestion des droits à *enrouten*.

[6] *Symfony* est un *framework PHP*.

[7] Twig est le moteur de template de *Symfony*, codé en *PHP*.

MongoDB

Comme beaucoup de sites dynamiques, celui-ci a besoin d'une base de données. La plupart des projets utilisent une base de données relationnelle tandis que pour des raisons évidentes de manipulation de documents et éviter d'avoir deux bases de données (une pour les utilisateurs et l'autre pour les documents) il m'a été imposé d'utiliser *MongoDB*. J'ai pu découvrir ce que l'on appelle le *NoSQL*[8] et comprendre la logique de la manipulation de documents.

Mongoose

Utilisant à la fois *node.js* et *MongoDB*, je n'ai pas eu à chercher longtemps pour trouver le module qui m'a permis simplement de faire des requêtes depuis *node.js* vers *MongoDB* : *Mongoose ODM*[9].

Robomongo

Ce petit utilitaire graphique offre davantage de confort de lecture, en proposant une vue en arbre, très pratique lorsque les objets ont beaucoup de niveaux de profondeur.

Mocha

Voyant l'application grossir peu à peu, j'ai pris la décision d'implémenter des tests unitaires. Plusieurs outils s'offraient à moi : *node-unit*, *Expresso*, et bien d'autres. Mon choix s'est naturellement orienté vers *Mocha* car développé par les mêmes personnes qu'*Express* et très adapté aux besoins de l'application. Les tests au travers de l'*API REST* sont très simples à réaliser. J'ai couplé *Mocha* à *should.js* pour tester le résultat des réponses *JSON* de l'*API*.

3.3.3 Côté client (front-end)

Le côté client ayant pour but d'être modulable et orienté **composant**, il m'a fallu utiliser des outils relativement imposants et puissants.

AngularJS

Au cœur de la dynamique client, *AngularJS* est complet, puissant et extensible. Il est construit selon l'architecture *MVC*[10]. Son principal avantage qui peut parfois être un défaut est l'écoute réciproque des événements entre les différentes couches, les écouteurs d'événements sont gérés automatiquement. C'est exactement ce qu'il fallait pour pouvoir faire une interface qui interagisse avec le côté serveur via l'*API REST* ou encore faire des filtres de recherche très rapidement et simplement.

AngularUI Router

Tout comme pour le côté serveur, le routage côté client de base n'est pas nommé. J'ai opté pour le routeur avancé proposé par *AngularUI* qui offre davantage de possibilités que la *directive ngRouter* proposée de base.

3.3.4 Les extensions Chromium

Pour m'aider tout au long de ce projet, j'ai utilisé des outils utiles au débogage et au développement.

[8] NoSQL : Not only SQL
[9] ODM : Object Document Mapper
[10] MVC : *Model View Controller*, littéralement *Modèle Vue Contrôleur*

Barre d'outils

La grande force de *Chromium* pour les développeurs frontaux (*front-end*) est certainement la barre d'outils du navigateur qui est très complète et qui inclut, entre autres, une console *JavaScript*, une navigation dans le *DOM*[11], dans le style *CSS* ainsi que la modification à la volée de tous les attributs ou propriétés que l'on souhaite.

AngularJS Batarang

Extension qui rajoute un onglet à la barre d'outils et propose une grande quantité d'assistants au débogage du côté client construit avec *AngularJS*. On y trouvera un explorateur de *scopes*, un graphe de dépendances entre les divers modules : pratique pour voir rapidement quelles dépendances ne sont plus utiles après un réusinage de code (*refactoring*).

Postman

C'est une application à part entière, construite en tant qu'une extension *Chromium* mais qui pourrait être une application à part entière. Elle permet d'effectuer des requêtes *HTTP* et de consulter les réponses, et surtout d'enregistrer des requêtes et leurs paramètres pour pouvoir les exécuter rapidement. J'aurais pu utiliser *cURL* et faire des scripts, mais *Postman* offre un plus grand confort d'utilisation : visualisation des *JSON* sous forme d'arbre, détail des en-têtes *HTTP* de la réponse, …

LiveReload et Emmet LiveStyle

Bien que secondaire dans ce projet, j'ai utilisé *LiveReload* et expérimenté *Emmet LiveStyle* qui sont deux outils de rechargement synchronisés avec l'éditeur de code. *LiveReload* se contente de recharger les fichiers lors de leur sauvegarde tandis que *Emmet LiveStyle* permet d'éditer le style *CSS* en temps réel dans l'éditeur de code ou dans le barre d'outils de *Chromium*, les deux se synchronisant dans un sens ou dans l'autre.

[11] DOM : *Document Object Model*

4 Réalisation du projet

Bien que l'analyse soit ici séparée du développement, cela n'est pas en lien avec le déroulement chronologique du stage. Etant en recherche et développement, le développement a évolué au fur et à mesure des recherches autour du projet.

4.1 Analyse

4.1.1 Etude de l'existant : expertfixer

Le premier jour du stage était le jour du lancement de la première version d'*experfixer*. La machine qui m'était dédiée par la suite étant alors utilisée pour compiler l'application. J'ai suivi avec attention les diverses procédures de Cyril et j'ai appris les fondements du fonctionnement d'*experfixer*. Cette simple journée m'aura suffit à comprendre qu'*experfixer* était une application basée sur *QtWebKit*, la vue est donc construite en *HTML*, *CSS* et *JavaScript*.

Dans un premier temps, l'application utilisait *BackboneJS*, puis suite à des problèmes de performance, cela a été migré vers *AngularJS* qui posait moins de problèmes et qui proposait davantage de fonctionnalités.

J'ai pu également découvrir le moyen de stockage des audits qui n'était autre qu'un fichier *JSON* accompagné d'un *manifest* et d'une capture haute définition, tout cela pour chaque page de l'audit. Cette partie de la structure était intéressante dans l'optique d'une réflexion à un outil de gestion des versions adapté à cette structure. C'est pour cela que le choix de *MongoDB* s'est révélé être un plus, le *JSON* étant un document manipulable simplement par *MongoDB*, cela simplifiait grandement la suite du développement, auquel je n'ai malheureusement pas pu arriver.

4.1.2 Réunions

Au début du projet, nous avons fait une première réunion avec Michel et Cyril afin de poser les bases et la direction à prendre dans le développement de ce projet. C'est lors de cette première réunion que j'ai pris connaissance des objectifs, mais également de la construction globale d'*experfixer* et de son fonctionnement. Les objectifs alors énoncés sont ceux cités dans la partie dédiée de ce rapport.

Après ma découverte des outils, notamment *node.js*, nous avons fait une nouvelle réunion pour réévaluer certains points et éclaircir la notion de groupe et les différentes couches d'autorisation à gérer. C'est alors que nous avons défini 3 couches :

- **réseau**, qui correspondrait à une entreprise ou une filiale d'une grande entreprise ;
- **groupe**, qui serait un groupe de travail pour certains, un projet pour d'autres, l'usage est libre, le tout étant d'avoir la possibilité de gérer ce niveau de droits ;
- **audit**, qui est le plus bas niveau et qui se limite à la gestion d'un audit et de toutes ses pages, sa gestion des versions, ses commentaires et autres discussions.

Aucun cahier des charges prédéfini n'a été fourni. Aucun cahier des charges n'a été rédigé pendant la période du stage. En effet, le projet évoluant de jour en jour à mesure de son développement et des problèmes rencontrés, il était impossible de rédiger quelque chose de définitif.

4.1.3 Prototypage, méthode agile

La procédure suivie relève typiquement de la recherche et développement, c'est-à-dire que le développement et la réflexion sont entremêlés et les deux notions permettent de faire avancer l'autre. Ainsi, la théorie lance la pratique et on peut alors valider ou non la théorie lorsqu'elle est mise en pratique. Si cela s'est révélé infructueux, la théorie est revue et le développement reprend alors sur cette nouvelle piste.

Chaque fonctionnalité n'a pas été développée telle qu'elle l'aurait été pour un livrable, mais comme un prototype. Ainsi, le côté client n'aura qu'un design simpliste, l'accessibilité n'a pas été réellement travaillée, mais il est possible de rendre accessible ce qui est en place. Par exemple, la gestion des droits se fait via des cases à cocher et non via du glisser-déposer, ceci rendant très facilement accessible de tels outils, bien qu'un outil graphique serait bien plus pratique pour un utilisateur lambda : mais là n'était pas l'intérêt de cette partie du projet.

4.1.4 RESTful

Une composante majeure du projet était de rendre la plateforme compatible avec *expertfixer* via des composants, mais également au niveau de l'interrogation et des réponses au serveur. La technique la plus simple et la plus répandue est la création d'une *API REST* dite *RESTful* car elle respecte complètement les standards :

- format du type de données : *JSON* ;
- respecte les méthodes *HTTP* standards (*POST*, *GET*, *PUT*, *DELETE* qui forment ce que l'on appel *CRUD* : Create-Read-Update-Delete) ;
- le lien de base de chaque collection cible tous les éléments de la collection (exemple : http ://hote/api/users/ pour cibler tous les utilisateurs) ;
- le lien de base suivi d'un identifiant cible un élément de la collection (exemple : http ://hote/api/users/toto pour cibler l'utilisateur toto).

REST est un modèle de conception et non un protocole, les points su-cités sont donc bien des standards et non des spécifications. Aussi, il n'est pas lié à *HTTP* bien que très souvent utilisé au travers de celui-ci.

4.2 Développement

4.2.1 Architecture

Contraintes

La seule contrainte imposée est de rendre modulable la plateforme via une *API RESTful*.

Réalisation et choix

Pour rendre la plateforme la plus modulable possible, j'ai choisi de séparer les différentes couches afin d'arriver à la répartition suivante :

- base de données ;
- logique serveur : l'*API REST* ;
- interface client : l'application visible par l'utilisateur.

Problèmes et solutions

Il a été difficile de choisir l'endroit où serait géré l'application visible du client. Il y avait deux choix possibles :

- Créer un deuxième serveur *Express* en faisant des appels à l'autre serveur sous *Express* qui gère l'*API REST*, ainsi la seconde couche produirait des fichiers HTML délivrés classiquement au navigateur à chaque changement page ;
- Gérer la dynamique de l'application côté client en interrogeant l'*API REST* directement depuis le poste utilisateur.

C'est cette seconde solution qui a été retenue, car elle s'est révellée plus légère et plus rapide côté client, tout en allégeant la charge du serveur.

Améliorations possibles

C'est une architecture tri-tiers : un classique à l'heure actuelle. Il est très certainement possible de l'améliorer, mais elle me semble suffisante à la réalisation de ce projet.

4.2.2 Base de données

Contraintes

Le moteur de la base de données m'a été imposé, il s'agit de *MongoDB*. Il a été choisi pour sa capacité à gérer des documents très rapidement ainsi que la compatibilité totale offerte avec *node.js*.

Réalisation et choix

Je ne suis pas intervenu dans le choix du *SGBD*, mais il m'a été fait part de la réflexion autour de cette contrainte qui m'a été donné. *MongoDB* est donc un gestionnaire de base de données par documents, et non-relationnel. Aussi, son langage d'interrogation n'est pas le *SQL* mais un ensemble de méthodes dont la sémantique est proche. Celles-ci se base sur les deux opérations principales de l'architecture *MapReduce*. Les langages d'interrogation de ce type sont classifiés comme étant *NoSQL*[12]. En effet, les requêtes vont au delà des fonctionnalités proposés par le *SQL*, notamment le traitement des propriétés de chaque élément d'une collection qui peut aller très loin.

L'avantage majeur du choix d'un moteur *NoSQL* est que chaque élément d'une même collection peut avoir un schéma différent et il devient ainsi possible de stocker ce que l'on souhaite à n'importe quel moment, sans devoir mettre à jour le schéma d'une table comme avec un *SGBDR*. Ainsi, les documents stockés à terme, que sont les audits, pourront évoluer sans avoir à retoucher au code source de la plateforme supportant le réseau social, à moins qu'une mise à jour majeure intervienne et change des fichiers clés tels que le *manifest*[13] d'un audit.

Pour simplifier la liaison entre *node.js* et *MongoDB*, j'ai choisi d'utilser *Mongoose ODM*. Cet outil permet de créer et gérer très simplement des collections *MongoDB* au travers de schémas, qui contiennent à la fois les différents attributs de l'objet, mais aussi les méthodes associées au schéma. On obtient alors quelque chose de comparable à une entitié sur les *ORM*.

[12] NoSQL : Not only SQL

[13] Fichier énumérant de façon organisée l'ensemble des fichiers liés à l'audit

Problèmes et solutions

Bien que cela ne soit pas des plus efficaces, il est tout de même possible de faire du relationnel avec *MongoDB*. Tous les bénéfices des clés étrangères et des diverses vérifications de correspondances sont, par contre, perdus : *MongoDB* se moque de l'existence ou non de l'élément lié. Il est plus courant de dé-normaliser la base pour dupliquer l'information à plusieurs endroits que de faire des jointures telles qu'en *SQL*. Cela a été problématique pour moi en ce sens où j'étais habitué à ne pas dupliquer les informations pour éviter de perdre la cohérence des données. J'ai donc pris le parti de minimiser les "jointures" nécessaires en utilisant au maximum l'aspect document de *MongoDB*. On pourra, par exemple, retrouver les rôles de l'utilisateur relatif à chaque réseau/groupe/audit dans chacun des utilisateurs, tandis que dans une base relationnelle il aurait été préférable de faire une table liée d'une part à l'utilisateur et de l'autre au réseau/groupe/audit concerné.

Améliorations possibles

Les différents schémas sont sans doute perfectibles, notamment sur la validation des données. Certains champs profitent de la souplesse de *MongoDB* et sont donc relativement libres et non validés ce qui crée parfois des incohérences lors des mises à jour de la structure.

Il y a certainement des dé-normalisations possibles de façon à améliorer notablement les performances de la plateforme par rapport aux requêtes effectuées, mais mes connaissances acquises au cours de ce stage à propos de *MongoDB* ne m'ont pas permis d'améliorer davantage le système en place avant la fin de ce stage.

4.2.3 Gestion des utilisateurs

Contraintes

Peu de contraintes m'ont été imposées à ce niveau, la principale étant que les données de certains clients étant sensibles, il m'a été demandé de les protéger.

Réalisation et choix

La gestion des utilisateurs était commune : inscription et connexion, le tout accessible via l'interface *REST*. J'ai opté pour la bibliothèque *PassportJS* pour sa souplesse et son extensibilité détaillée dans la deuxième grande partie au sujet des outils utilisés. La possibilité d'avoir une connexion via *OAuth* dans le futur était plutôt séduisante par rapport à la modularité souhaitée de la plateforme.

Problèmes et solutions

Le choix de *PassportJS* est un choix réfléchi, qui implique le développement d'une surcouche pour gérer l'authentification en *REST*. J'ai donc créé un composant nommé *security* légèrement inspiré du composant *Security* de *Symfony 2* qui gère à la fois l'inscription, la connexion et qui pourra être étendu dans le futur assez simplement.

Améliorations possibles

Il n'y actuellement aucun moyen de changer son adresse courriel, ni son mot de passe. Il n'y a pas non plus de système de récupération de mot de passe. Tout cela est donc améliorable du point de vue des fonctionnalités.

Il serait également envisageable de faire un module *passport-rest* déposé sur *npm* de façon à partager et profiter des retours et corrections d'autres développeurs.

4.2.4 Routage

Contraintes

La seule contrainte est indirectement imposée par le sujet du stage du fait de devoir développer une *API RESTful* ce qui oblige à respecter certaines conventions de construction des *URL* citées en amont dans ce rapport.

Réalisation et choix

Pour gérer le routage côté serveur, j'ai cherché un module qui serait une surcouche à *Express* en y ajoutant la gestion du nommage des routes. C'est donc *enrouten* qui a été mon choix, par rapport à sa grande quantité d'options qui étaient adaptées aux besoins du projet. On y trouvera, par exemple, la possibilité de construire les *URL* par rapport à l'arborescance des dossiers conteneurs du contrôleur concerné, la possibilité de l'étendre et d'accéder simplement à la liste des routes par leurs noms.

Problèmes et solutions

Le paquet n'incluant pas de fonction qui permette de reconstruire l'*URL* à partir du nom et des paramètres de la route, j'ai du la développer en créant une surcouche à *enrouten* (qui lui-même est déjà une surcouche au routeur d'*Express*). Cela a très bien fonctionné et s'est révélé efficace.

Un autre souci a été de gérer la page d'accueil (l'application) et l'*API* séparément. Il faut renvoyer du *HTML* d'un côté et du *JSON* de l'autre. Une personne qui arrive depuis une page externe doit être renvoyée vers la page d'accueil qui se chargera via le code *JavaScript* de reconstruire et de rediriger vers la bonne page. J'ai donc du adapter le routeur côté serveur de telle façon qu'il retourne la page principale de l'application si l'adresse ne fait pas partie des *URL* de l'*API*.

Côté client cette fois, le routage devait donc gérer l'éventualité que l'utilisateur venait d'ailleurs. Le serveur renvoyait alors la page d'accueil (qui n'est autre que l'application complète) et le côté client se chargeait de construire la page attendue. Il a également fallu chercher un module pour gérer les routes nommées de ce côté, et c'est ainsi que j'ai trouvé *AngularUI* et sa composante *Router* qui aide, entre autres, à nommer les *URL* et de les reconstruire depuis leur nom et paramètres, le tout très simplement (beaucoup plus simple que le côté serveur).

Pour gérer les différentes routes, j'ai créé un fichier `config/routing` qui regroupe les contrôleurs à gérer en fonction des préfixes, qui sont hérités de niveau en niveau.

Améliorations possibles

La philosophie de gestion des routes que j'ai souhaité employer émane de mon passé avec *Symfony 2* qui est très pointu à ce niveau. Il y a très certainement des conventions que je n'ai pas vu et qu'il serait préférable d'appliquer. J'ai donc appliqué la structure qui n'est probablement pas la meilleure, mais qui a le mérite d'être modulable et de fonctionner.

Encore une fois, créer un paquet *npm* pour la surcouche à *enrouten* serait un plus. Cela permettrait de sortir ce code du corps de l'application et de pouvoir le réutiliser dans d'autres projets. Le module est déjà conçu comme un module *node.js*. Il n'y a donc qu'à en faire un paquet.

4.2.5 Droits

Contraintes

Les droits devaient être assez fins pour gérer la lecture ou l'écriture d'un utilisateur pour chaque niveau de segmentation : réseau, groupe, audit. Par exemple, il devait être possible d'ajouter un prestataire externe en

lecture seule sur un audit pour qu'il puisse le regarder sans pour autant le modifier mais avoir accès en écriture sur le groupe pour pouvoir faire un retour au groupe au sujet de cet audit. Il y a de nombreux cas étranges à gérer qui ont rendu la gestion des droits plus complexe de façon globale. Enfin, la gestion des droits devait se faire au travers de l'*API* et donc retourner les bonnes erreurs quand il fallait signaler que l'utilisateur n'avait pas les droits.

Réalisation et choix

La gestion des droits est faite au niveau du routeur, en testant à chaque requête si l'utilisateur a parmi sa liste blanche le réseau, groupe ou audit auquel il tente d'accéder. Cette liste blanche est stockée dans l'objet *user* en tant que surcouche au schéma de *PassportJS* via un attribut que j'ai nommé *roles*. J'ai opté pour une solution "fait-maison" car les modules d'*ACL*[14] ne correspondaient pas aux besoins et aux contraintes imposés.

Pour gérer la correspondance droit/route, j'ai fait en sorte d'avoir un fichier de configuration dédié à la gestion des droits, nommé `config/security`. Il permet de paramétrer la liste des rôles ainsi que la liste de contrôle d'accès. On retrouvera un système de préfixe et des paires *path/role* mais également une gestion du contexte (réseau, groupe, audit).

Cet exemple écrit en *CoffeeScript* est extrait du fichier `config/security.coffee`.

```
prefix : '/network'
rules : [
    path : '/register'
    role : 'ROLE_AUTHENTICATED'
,
    prefix : '/ :network'
    rules : [
        path : '/admin'
        context : 'network'
        value : ' :network'
        role : 'ROLE_ADMIN'
    ]
]
```

On peut y voir le support du couple *path/role* simple avec le premier objet. Une gestion des préfixes très simple est disponible. Dans l'exemple, elle est couplée à l'utilisateur d'un paramètre qui sera utilisé dans les règles du tableau `rules`.

Dans cet exemple, on trouve deux règles finales qui gèrent les droits d'accès aux *URL* `/network/register` et `/network/<nom-du-réseau>/admin`. Dans le premier cas, l'accès nécessite simplement d'être authentifié. Dans le second cas, il faut le rôle administrateur (*ROLE_ADMIN*) sur le réseau dont le *slug* est capturé par l'expression `:network`.

Les règles sont analysées dans l'ordre de leur inscription dans ce tableau de gestion des droits d'accès. Ainsi, la première règle sera prise en compte en priorité si elle est avant une autre qui correspond également à l'*URL* de la requête ; cela peut être assez déroutant parfois car il faut bien faire attention à l'ordre des règles et mettre les règles les plus précises en premier (généralement les *URL* les plus longues) pour finir par les règles plus généralistes. Seule la première règle trouvée est prise en compte.

Ce fichier de configuration est très inspiré de celui que l'on trouvera sur *Symfony2* car il autorise une grande finesse et une grande souplesse à la fois.

[14] ACL : *Access Control List*, littéralement *liste de contrôle d'accès*

Problèmes et solutions

La gestion des préfixes a été une première difficulté, nécessitant de la récursivité et la mise en place d'une structure particulière dans le fichier de configuration, il m'aura fallu remanier le code pour les prendre en compte. J'ai eu à un moment un soucis qui faisait que seul le préfixe de premier niveau était pris en compte. J'ai alors revu le code et j'en ai profité pour corriger d'autres bogues tels que la non prise en compte des caractères `^` et `$` — qui permettaient respectivement de détecter le début et la fin de l'*URL* — dans le fichier de configuration, j'ai donc retiré leur support dont le manque ne s'est pas fait sentir.

Améliorations possibles

Une amélioration notable serait d'offrir la possibilité d'avoir plusieurs règles qui sont validées pour une même requête, en héritant les règles du préfixe par exemple. Ainsi on pourrait cumuler les rôles nécessaires pour accéder à certaines pages, ce qui n'est pas le cas actuellement.

Il serait intéressant également d'avoir la possibilité de rendre dynamique la liste des rôles : un administrateur pourrait ainsi créer des rôles par audit, groupe ou réseau et donner les droits qu'il veut pour chaque rôle et ainsi obtenir un résultat plus proche de ce qu'il souhaite.

Une autre chose possible serait de créer un paquet *npm* pour le module de gestion des droits, qui est déjà, comme les autres, un module *node.js*.

4.2.6 Tests unitaires

Contraintes

Aucune contrainte ne m'a été imposé, les tests ont été mis en place durant le développement de mon plein gré.

Réalisation et choix

J'ai choisi *Mocha* qui est un *framework* de tests *JavaScript*, que j'ai couplé à *should.js* pour les assertions. J'ai tenté de diriger mon développement par les tests, mais cela s'est révélé plus long et complexe que de développer les tests après le module en question. C'est donc une trentaines de tests qui ont été développés pour surveiller les éventuels régressions produites par les derniers changements effectués.

Problèmes et solutions

La gestion des sessions de connexion a été le point le plus gênant lors du développement des tests. Il m'a fallu apprendre à utiliser la librairie *async* pour les rendre à la fois performants et lisibles. En effet, le chaînage des actions et requêtes est garanti par des fonctions (souvent anonymes) de retour, et on arrive très vite à une indentation très large et illisible. J'ai pu apprendre la différence entre les diverses gestions de flux d'*async* et principalement les entre les modes série, parallèle et en cascade. Il y a notamment certains bogues qui étaient dûs à l'utilisation du mode série plutôt que le mode cascade.

Améliorations possibles

La batterie de tests est largement extensible, notamment sur les dernières fonctionnalités ajoutées dont la gestion des droits. Il pourrait être intéressant de développer des tests pour valider le côté client bâti avec *AngularJS*, en utilisant *Mocha* également.

4.2.7 Interface de l'application

Contraintes

Aucune contrainte au niveau du côté client, mais on m'a conseillé d'utiliser *AngularJS* plutôt que *Backbone* pour des raisons de performance. J'ai suivi ce conseil, notamment par curiosité de découvrir ce *framework* dont on entend parler partout.

Réalisation et choix

L'utilisateur est stocké côté client dès sa connexion. La connexion est donc automatique, car à chaque retour sur le site, les informations sont disponibles localement et tant que le serveur considèrera la session comme active, l'utilisateur sera connecté. Un bouton de déconnexion est néanmoins disponible pour quitter et vider la session à tout instant.

J'ai créé plusieurs *directives* :

- fil d'Ariane : garantit la construction automatique et la cohérence durant la navigation ;
- boîte modale : gère l'ensemble des boîtes modales du site et leur possibilités d'action ;
- messages d'alerte : gère la pile des message d'erreur, d'information ou d'alerte.

J'ai également créé un service pour gérer l'autentification, côté client. Ce service gère la connexion, la déconnexion, permet de mettre à jour le statut de l'utilisateur. Par exemple, du fait de la dénormalisation de la base, la liste des réseaux auxquels l'utilisateur a accès est dans le document de l'utilisateur. Il faut donc mettre à jour la copie côté client régulièrement, notamment au moment où de nouveaux réseaux, groupes et audits sont créés.

Problèmes et solutions

La gestion de l'utilisateur a été quelque peu complexe, notamment la gestion de l'état de la connexion : est-ce que l'utilisateur est connecté ou non ? Pour cela j'ai du développer un façade supplémentaire dans l'*API REST* qui permet de récupérer les informations de l'utilisateur, qui retourne une erreur *403* si celui-ci n'est pas connecté.

Aussi, la notion de session n'existe pas côté client. J'ai tenté d'utilise *sessionStorage* mais cette variable n'est accessible que pour la session courante de l'onglet du navigateur en question, ce qui empêche notamment l'utilisation de plusieurs onglets. J'ai donc opté pour une gestion via *localStorage* en y stockant les données utilisateur non-sensibles.

Un des soucis les plus délicats a été la désynchronisation client/serveur ; c'est-à-dire que la version du document décrivant l'utilisateur n'était pas la même des deux côtés. J'ai du étendre le service d'autentification pour qu'il se mette automatiquement à jour au déclenchement de certains événements suite à des actions clés telles que la création d'un réseau, sa modification ou sa suppression. Il en va de même pour la gestion des droits des utilisateurs pour un réseau donné.

Améliorations possibles

La chose la plus important serait de rendre accessible la totalité du côté client de l'application. Actuellement, l'application n'est pas accessible mais tout a été fait pour qu'elle puisse le devenir facilement.

La gestion des boîtes modales est à revoir, cela devient très complexe dès qu'elle contient un formulaire dont le comportement est défini dans la page mère (celle qui a permi l'ouverture de la boite modale).

4.3 Compétences acquises

node.js

Je n'avais jamais vraiment fait beaucoup de *JavaScript*, et encore moins côté serveur. Ce stage m'a donné l'opportunité de découvrir *node.js* ainsi que toutes les possibilités offertes par ce fameux moteur *V8* amélioré pour être indépendant.

Suite à quelques tests de charge, j'ai pu constater la puissance de *node.js* de par sa rapidité et sa tenue lors de montées en charge. Pourtant, tout semble simple à développer, les exercices ou didacticiels trouvés sur l'Internet le prouvent : on en trouvera facilement un pour créer un système gestion de liste de choses à faire (*todo-list*) ou encore un blog, et ce en quelques heures maximum.

La grande force de *node.js*, selon moi, est la richesse de sa bibliothèque de modules, nommée *npm*. Avant de coder quoi que ce soit, je suis allé chercher dans cette mine d'or si un composant ne faisait pas ce que je souhaitais, et sur un seul et même site Web, tout est regroupé, classé et expliqué.

Enfin, après cette excursion dans le très vaste monde de *node.js*, je peux affirmer que *node.js* permet de faire énormément de choses et on trouve de tout : *node-webkit* pour faire des applications de bureau basées sur *node.js* et avec le moteur de rendu *blink*, mais aussi *Node-OS* : un système d'exploitation en cours de développement basé sur un noyau *Linux* qui lancerait directement une console *node.js*. On trouvera donc des clients IRC, des jeux, des utilitaires, etc. basés sur *node.js*.

CoffeeScript

C'est un préprocesseur que j'ai voulu découvrir et j'ai profité de l'omniprésence du *JavaScript* dans ce projet pour pouvoir tenter l'expérience. J'ai donc développé tout le côté serveur en *CoffeeScript* et j'ai apprécié la légèreté du langage tant au niveau de sa syntaxe que des facilitateurs qui sont offerts.

Je le trouve par contre peu pratique dès qu'il s'agit de manipuler le *DOM* côté client, qui est donc resté en *JavaScript*.

JavaScript

J'étais plutôt à l'aise avec les notions de bases du *JavaScript* mais n'étais jamais allé aussi loin. J'ai pu comprendre et apprendre une grande partie de la logique du *JavaScript* qui m'avait échappé jusqu'ici.

API REST

Au début du projet, je trouvais relativement contraignant de passer par une couche *REST* plutôt que de produire directement les pages dynamique côté serveur. Puis, peu à peu, je me suis rendu compte du potentiel de ce type d'architecture.

Parmi les avantages, on peut noter la possibilité de faire plusieurs façades pour un même système de gestion qui reste côté serveur. On peut donc créer une application mobile, tablette ou pour ordinateur, via le Web ou non, le tout étant construit sur la même base, ce qui factorise le code à gérer et améliore la maintenabilité du projet.

Suite à ce projet, j'aurais peut-être tendance à vouloir faire une couche *REST* sur tous les sites dynamiques que j'aurais à faire par la suite.

Durant cette partie du stage, j'ai pu comprendre l'intérêt des *RFC* pour les développeurs dans des langages à haut niveau d'abstraction : j'ai pu trouver les différents codes d'erreur à renvoyer dans chacun des cas précis, ainsi que la description précise de chacune des méthodes définies par le protocole *HTTP*.

Accessibilité

Une part du projet m'a aidé à être sensibilisé à l'accessibilité numérique et de comprendre le fonctionnement des outils utilisés par les personnes atteintes d'un handicap visuel. On parle alors de conception visuelle en prenant garde aux choix des couleurs ayant un contraste suffisant, en choisissant des polices d'écriture lisibles, en permettant à l'utilisateur d'agrandir la page ou encore de jouer correctement sur les tailles des écritures ; mais aussi tout un tas de règles et conseils au niveau technique. J'ai vu également le fonctionnement d'*expertfixer* et une partie de ses règles qui m'a donc appris davantage de choses à ce niveau également.

J'ai pu découvrir tout un panel de référentiels, tantôt traitables automatiquement, tantôt nécessitant l'intervention d'un humain pour les valider.

L'expérience gagnée durant le stage m'a directement servi dans des projets extérieurs et j'ai pu ressentir la différence de la part de certains utilisateurs. J'ai pris davantage conscience du problème et me soucis désormais de l'accessibilité dans chacun de mes nouveaux projets.

HTML5

J'ai pu améliorer ma compréhension de la sémantique du *HTML5* ainsi que la notion de composant qui est en cours de développement au travers du projet *Web Components*. On peut d'ailleurs citer *Polymer*, qui est un projet qui va dans le même sens et qui est en cours de développement en ce moment même.

Cette plateforme a donc été, en plus d'être faite avec une couche *REST*, pensée pour être orientée "composants".

AngularJS

C'était ma première approche avec un *framework* riche côté client. Il en existe plusieurs autres, dont les plus répandus sont *Backbone* et *Ember.js*. N'en n'ayant testé aucun, j'ai suivi le conseil avisé de Michel et Cyril qui avaient, eux, expérimenté les deux dans *experfixer*, et je n'en suis pas déçu. J'ai saisi rapidement la puissance de l'outil, mais je reste loin de le maitriser, même après 3 mois de pratique durant ce stage, tellement il est complet et complexe.

J'ai également découvert une multitude de modules pour *AngularJS* tels que *AngularUI* que je ne connaissais pas non plus. J'ai alors compris comment fonctionnaient les animations avec *AngularJS*, le routage, les directives, le modèle *MVC* côté client, et beaucoup d'autres subtilités.

Le défaut d'*AngularJS* est sa façon étrange d'afficher les erreurs : un lien vers le site d'*AngularJS* avec le message d'erreur encodé dans l'*URL*, ce qui n'est pas du tout pratique lorsque l'on est hors ligne, et ça n'en reste pas moins pratique lorsque l'on dispose d'une connexion à l'Internet.

MongoDB et NoSQL

L'univers *NoSQL* est la plus grande découverte de ce stage. Je n'avais absolument aucune idée de ce que ça pouvait être, de comment c'était construit et encore moins de la façon d'interragir avec un tel type de moteur de base de données.

Ma première impression a été que la logique n'est pas si éloignée que celle des *SGBDR* auxquels j'étais habitué. Ce qui est déroutant c'est de ne pas avoir l'aspect "relationnel" de la chose. On se rend vite compte que ça n'est pas adapté à tous les projets, mais que dans les cas où le contenu est au centre des préoccupations, *MongoDB* (ou tout autre moteur du même genre, notamment *Redis*) est un choix judicieux.

Malgré ses différences dans la structuration de la base, un moteur *NoSQL* peut être utilisé, dans une certaine mesure, à la place d'un *SGBDR*.

Les performances sont très bonnes à ce que j'ai pu constater. J'ai pu noter que les performances en lecture étaient bien meilleures que celles en écriture.

5 Conclusion

Ce stage de trois mois au sein d'*Urbilog* a été mon premier stage en entreprise. Il aura été enrichissant, notamment pour comprendre l'organisation et le fonctionnement d'une entreprise.

Bien évidemment, j'ai beaucoup appris en terme d'accessibilité numérique, à tel point que c'est devenu un point important pour moi dans des projets communautaires extérieurs au stage. Au delà de l'ergonomie, de l'aspect visuel, des fonctionnalités, il faut toujours veiller à ce que ce que soit compréhensible par tous quelque soit la personne et ses handicapes.

La plateforme développée durant ce stage n'étant qu'une preuve de concept (POC), elle ne verra certainement jamais le jour publiquement sous sa forme actuelle ; mais je pense avoir réussi à prouver qu'il est possible de développer une telle plateforme avec les outils et technologies qui m'étaient imposées.

Bibliographie

Article qui m'a aidé à commencer avec *node.js* :
http ://fabbook.fr/developer-un-site-web-avec-node-js-et-tous-les-outils/blogarticle

Site officiel de *MongoDB* :
http ://www.mongodb.org/

Site officiel de *Mongoose ODM* :
http ://mongoosejs.com/

Documentation d'*express* 4.x :
http ://expressjs.com/4x/api.html

Documentation de *PassportJS* :
http ://passportjs.org/guide/

Site officiel d'*AngularJS* :
https ://angularjs.org/

Convertisseur *JavaScript* / *CoffeeScript* :
http ://js2coffee.org/

Glossaire

API : *Application Programming Interface*, littéralement *Interface de programmation d'application*. C'est une façade constituée de classes, méthodes ou de fonctions permettant l'interraction avec d'autres applications.

CSS : *Cascading Style Sheets*, littérallement *Feuilles de style en cascade*, est un langage qui décrit, notamment, la présentation des documents *HTML*.

directive : C'est un composant *AngularJS* autonome et réutilisable.

DOM : *Document Object Model*, standard *W3C* décrivant une interface permettant à des scripts d'accéder et de mettre à jour le contenu ou la structure de documents *HTML*.

HTML : *HyperText Markup Language*, est un format de données visant à représenter des pages *Web* en écrivant de l'hypertexte.

HTTP : *Hypertext Transfer Protocol*, littéralement *protocole de transfert hypertexte*, est un protocole de communication client-serveur développé pour le *Web*.

npm : *node.js package manager*, c'est le gestionnaire de paquets officiel de *node.js*.

OAuth : standard ouvert permettant l'autorisation à un site *Web* d'accéder à l'*API* d'un autre site *Web* pour le compte d'un utilisateur.

Opquast : *OPen QUAlity STandards* regroupe l'essentiel de ce qu'il faut comprendre pour obtenir des sites *Web* de qualité qui sont donc — entre autres — accessibles, référencé et ergonomiques.

POC : *Proof Of Concept*, littéralement *preuve de concept*, est une réalisation courte ou incomplète d'une idée pour démontrer sa faisabilité.

REST : *REpresentational State Transfer* : c'est une architecture client-serveur pour les systèmes hypermédia distribués.

RFC : *Request For Comments*, littéralement

RGAA : *Référentiel Général d'Accessibilité pour les Administrations*, est un référentiel visant à définir les modalités techniques d'accessibilité des services en ligne de l'État français, des collectivités territoriales et des établissements publics qui en dépendent, et notamment pour le *Web*.

SEO : *Search Engine Optimization*, littéralement *Optimisation pour les moteurs de recherche*. C'est un ensemble de techniques mises en place afin d'optimiser sa position dans les moteurs de recherche.

SGBD : *Service de Gestion de Base de Données*, est un système visant à stocker, partager et manipuler les données d'une base de données, en cachant la complexité des opérations.

SGBDR : *Service de Gestion de Base de Données Relationnelles*, est une catégorie des *SGBD* qui propose une structure matricielle via des tables. Le contenu de ce type de *SGBD* est manipulable grâce à l'algèbre relationnelle.

SVN : *Apache Subversion*, un logiciel de gestion de versions maintenu par la fondation *Apache*.

URL : *Uniform Resource Locator*, littéralement *localisateur uniforme de ressource*, plus couramment appelé *adresse Web*.

Web : c'est ce que l'on appelle moins communément le *World Wide Web* abrégé *WWW* ou *W3*, est un système hypertexte fonctionnant sur l'Internet. On pourra y consulter des pages de sites en utilisant un navigateur.

YUM : *Yellowdog Updater, Modified*, c'est le nom d'un gestionnaire de paquets notamment utilisé par *Fedora*.

Annexes

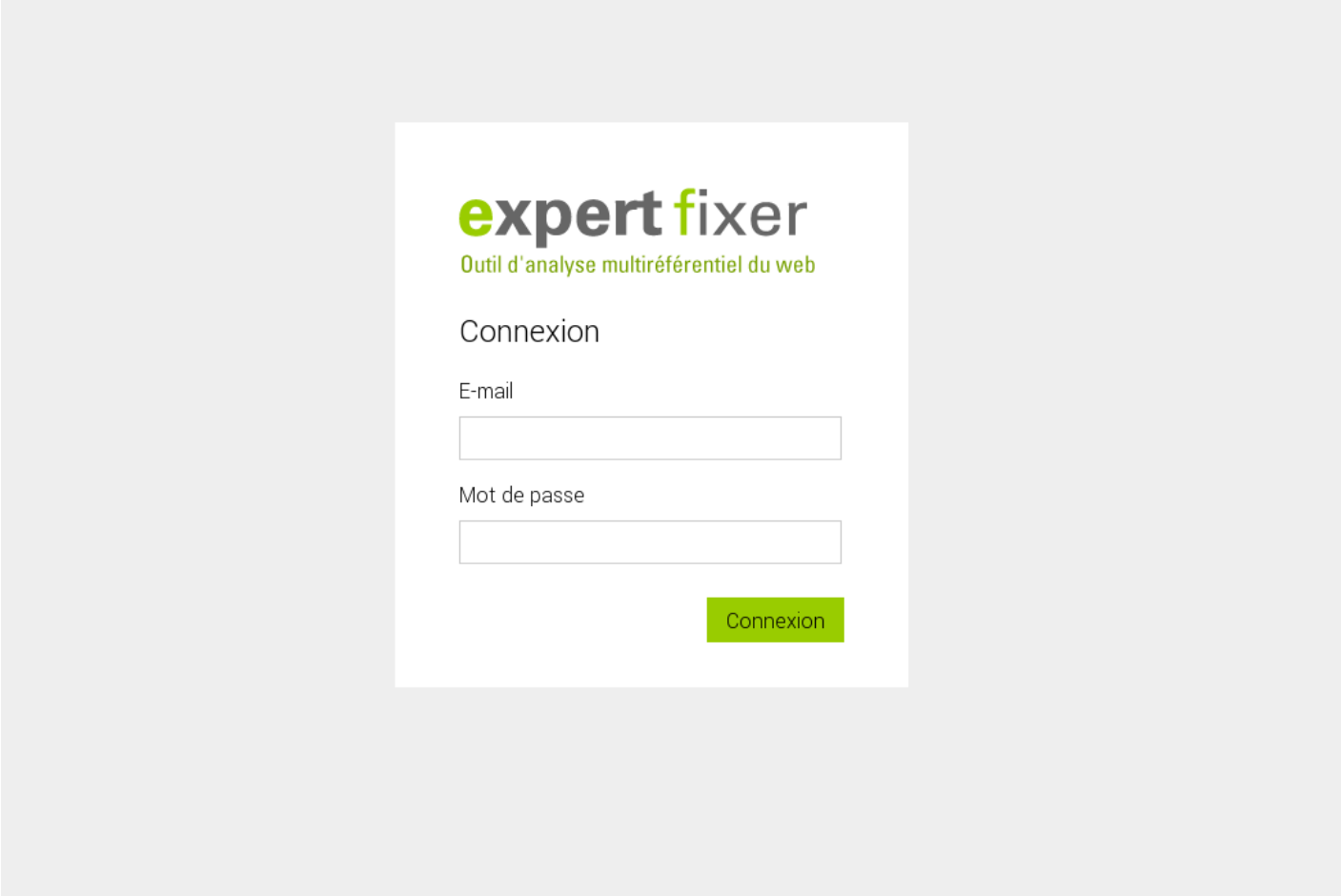

Fig. 2 : Page de connexion à la plateforme

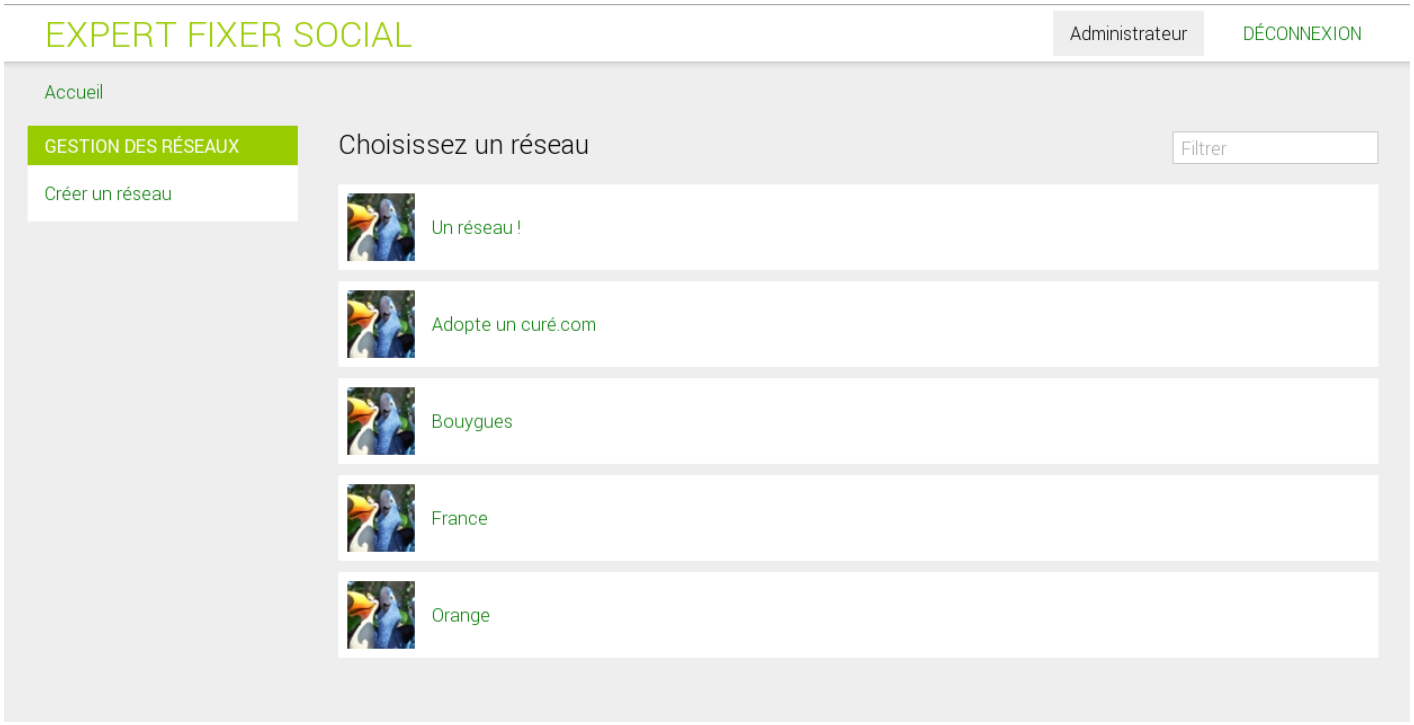

Fig. 3 : Accueil visible par un utilisateur connecté

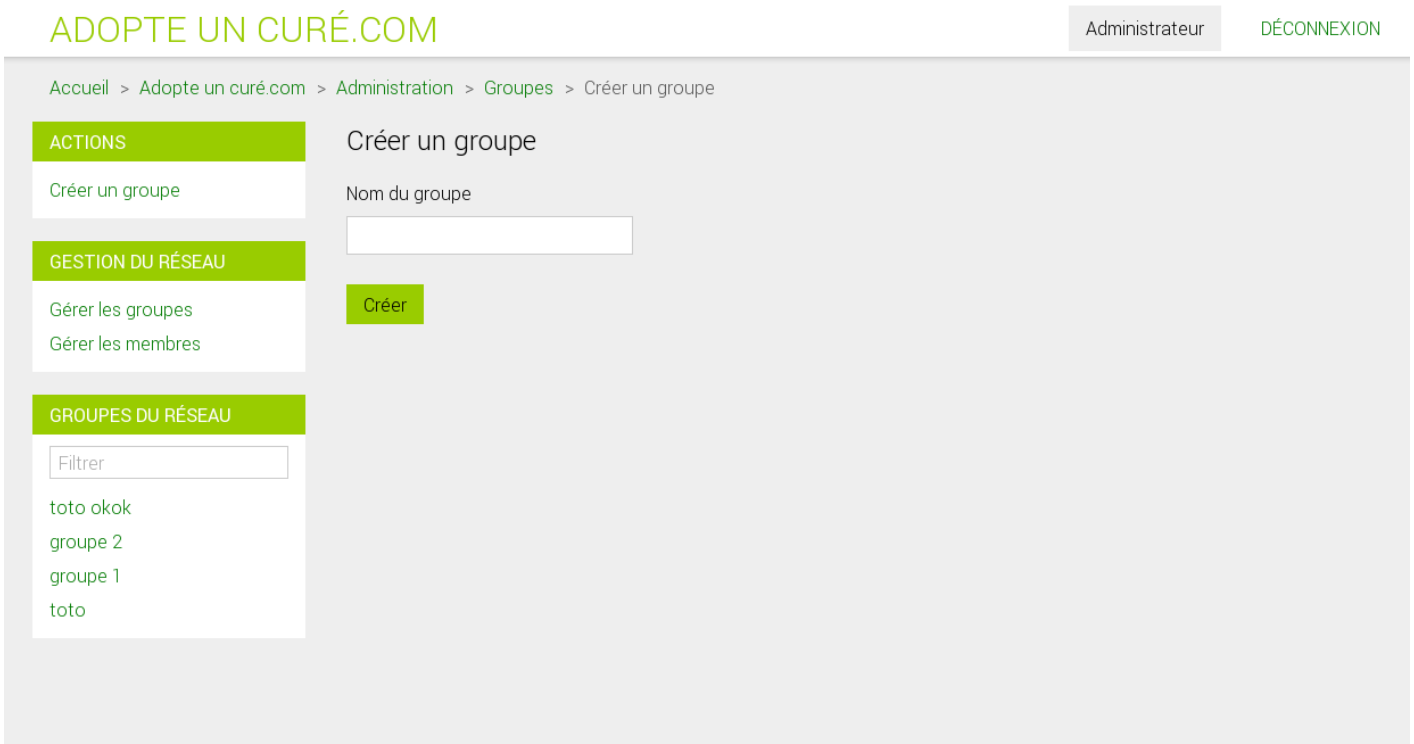

Fig. 4 : Création d'un réseau

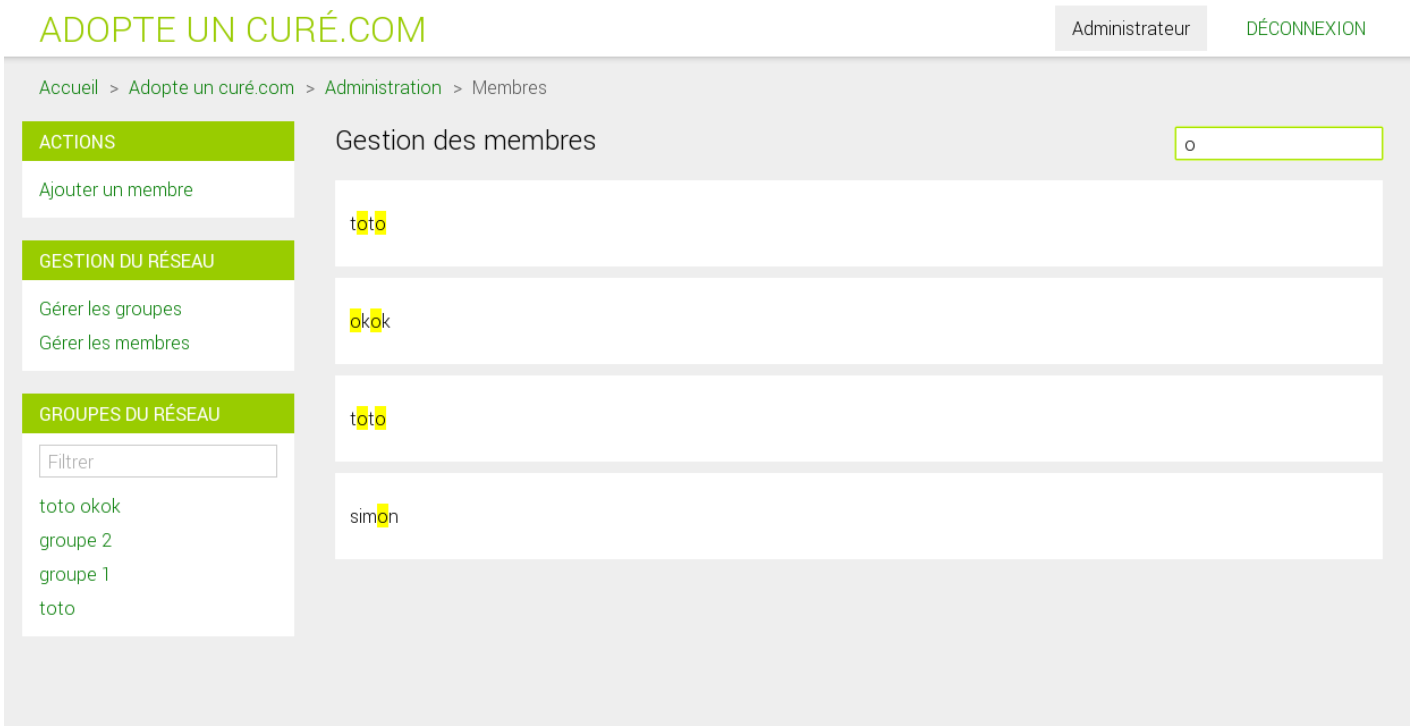

Fig. 5 : Il est possible de filtrer les listes instantannément

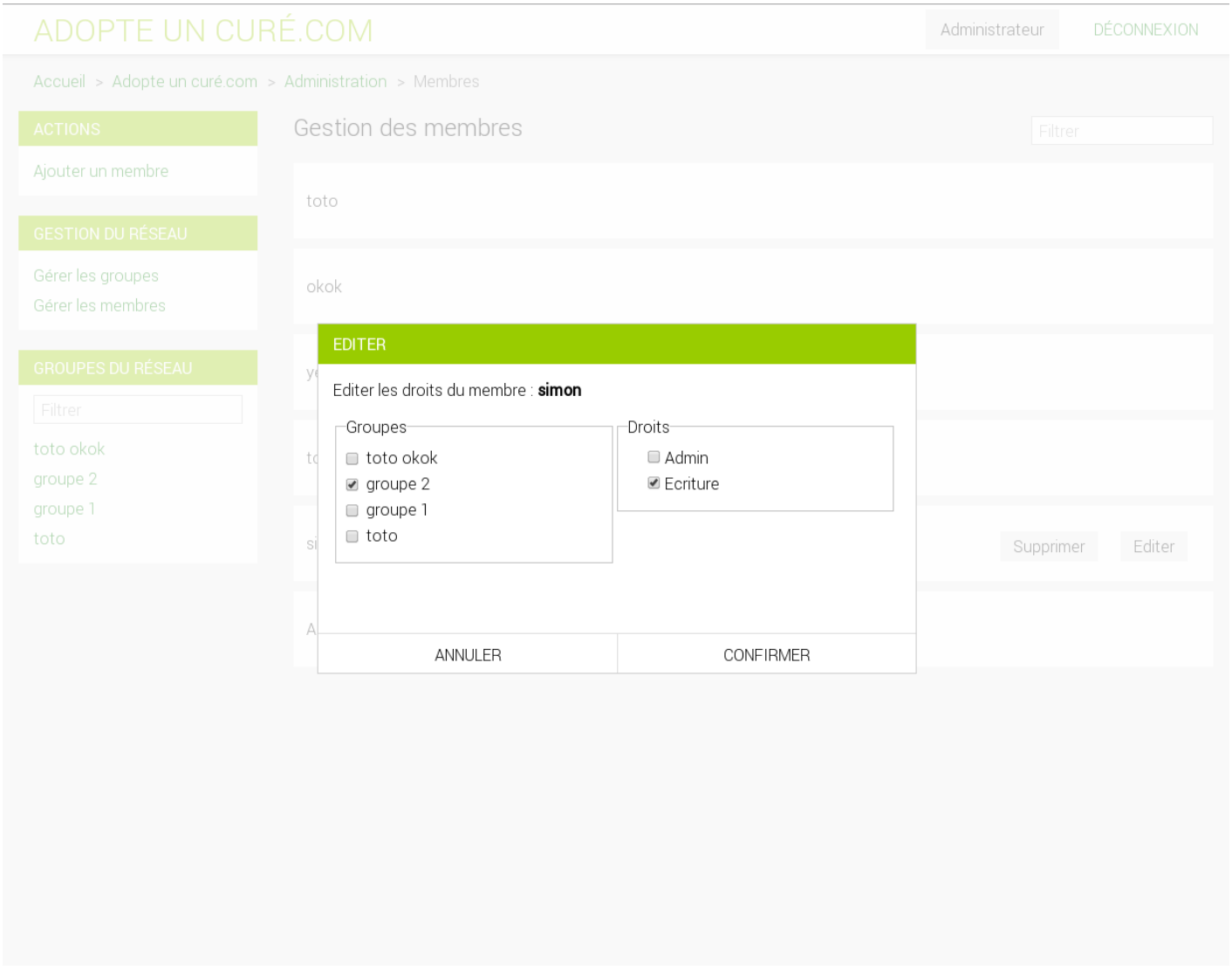

FIG. 6 : La gestion des droits est faite avec une boite modale