Appappa

Une application web pour les colocataires

Travail de Bachelor

Zeno Bardelli

Janvier 2018

Supervisé par :

Prof. Dr. Jacques Pasquier-Rocha
et
Pascal Gremaud
Software Engineering Group

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Groupe Génie Logiciel
Département d'Informatique
Université de Fribourg (Suisse)

Software Engineering Group

Remerciements

Tout d'abord j'aimerai remercier le Professeur Jacques Pasquier-Rocha de sa disponibilité et de m'avoir permis d'acquérir un grand nombre de connaissances à travers ce travail. Merci aussi à l'assistant Pascal Gremaud pour ses explications toujours claires et complètes.
Je tiens également à remercier mon ancien coloc et tous ces étudiants qui, en vivant en colocation, m'ont inspiré pour ce travail.
Enfin un grand merci à Chiara qui a corrigé mes erreurs en français.

Résumé

Les personnes qui vivent en colocation ont des besoins particuliers. On présente une application web appelée "Appappa" qui permet de satisfaire certaines nécessités telles qu'échanger des messages, régler les dépenses en commun, gérer une liste des courses commune et planifier les travaux ménagers. L'application a été entièrement realisée avec la stack MEAN. Cela signifique que l'application se compose d'une base des données en MongoDB, d'une RESTful API développée avec Node.js et Express et d'un client développée avec Angular.

Table des matières

Liste des figures

Liste des codes source

1
Introduction

1.1. Motivations et objectifs

Dans le cadre de ce travail, on développe une application web en utilisant le stack MEAN. En d'autres termes, il s'agit de développer une base de données MongoDB, une API REST Node.js et un client Angular pour obtenir une application web. Afin de réaliser l'objectif du travail, il était nécessaire de projeter une application pratique.

L'application qu'on a choisie de développer se veut donc comme un support pour les besoins organisationnels des colocataires. Le nom de l'application est "Appappa", une fusion entre les mots "app" et "appartement".

Il est très commun pour les étudiants de quitter le nid familial pour aller vivre proche de l'université. Soit-il pour des raisons économiques ou pour le plaisir d'être en bonne compagnie, très souvent les étudiants se retrouvent à vivre en colocation. En tant qu'étudiant, on a également vécu cette expérience et on a pu écouter de nombreux témoignages d'amis. Dans certaines colocations, tout le monde s'entend bien, alors que dans d'autres les personnes se parlent à peine et dans autres encore elles se disputent sans arrêt. Il existe toutefois un élément essentiel dont toutes les colocations ont besoin : un minimum d'organisation. Régler les dépenses en commun, planifier les travaux ménagers et organiser les courses ne représentent qu'une petite partie des tâches pour la répartition desquelles les colocataires essayent de trouver une méthode.

La dernière partie de ce travail consiste, dans la rédaction de ce rapport que vous êtes en train de lire, dont le contenu est détaillé dans la section qui suit

1.2. Organisation du rapport

En premier lieu, le rapport analyse en profondeur les besoins des personnes qui vivent en colocation. Ensuite, les situations dans lesquelles l'application en question pourrait être utilisée seront mises en évidence avec des diagrammes des cas d'utilisation à l'appui. Ces situations déterminent ce qu'il doit être possible de faire avec l'application.

Le chapitre suivant résume la mise en œuvre de l'application. On va expliquer ce qu'est le stack MEAN et ce qu'est une API REST. Ensuite on présentera MongoDB et la manière dont la base de données est construite, Node.js et le fonctionnement des endpoints de l'API, Angular et les élélments qui ont été implémentés dans le client.

La troisième partie consiste en une explication de ce qu'il est possible de réaliser avec le client de l'application. Le contenu du chapitre est illustré par de nombreuses captures d'écran qui montrent les pages du client telles qu'elles apparaissent dans son navigateur.

Ensuite une brève conclusion commente le développement avec le stack MEAN, et énumère les difficultés rencontrées ainsi que les points améliorables.

En annexe au travail principal sont fournies les trois appendices qui contiennent le code de l'implémentation de la base de données, les descriptions des endpoints et la description du fichier, en plus des instructions pour lancer l'application.

2
Analyse des besoins

Ce chapitre analyse les besoins sous-jacents à l'utilisation de cette application. En premier lieu, on décrit les problèmes organisationnels récurrents dans les collocation. Ensuite on montre à travers les cas d'utilisation comment l'application correspond aux besoins du public cible.

2.1. Les besoins des colocataires

Vivre en colocation est très commun parmi les jeunes, sourtout dans le cas d'étudiants universitaires. À travers l'expérience d'amis et connaissances et grâce à ma propre expérience on a remarqué qu'il y existe des discussions typiques. Comment peut-on administrer les dépenses en commun ? Comment peut-on établir la répartition des travaux ménagers ? L'application se propose comme un outil pour répondre à ces questions et à d'autres.

Si l'on part du principe que le potentiel utilisateur de cette application est une personne qui vit en colocation avec une ou plusieurs autres personnes, on peut tenter d'identifier ses besoins. Le premier est celui de disposer d'un compte personnel contenant ses données. De plus, il est nécessaire que ce compte puisse être rattaché à ceux des autres colocataires. Une autre fonction utile serait la possibilité d'envoyer des messages aux colocataires.

En ce qui concerne les problèmes concrets qui peuvent surgir dans le cadre d'une colocation, on estime qu'il est essentiel que des personnes vivant sous le même toit partagent certaines dépenses. Parfois, des dépenses communes sont faites pour des biens tels que la nourriture, les produits de nettoyage ou l'abonnement à internet. Le but serait donc de

faire en sorte que ce genre de dépenses soient partagées équitablement. Dans la pratique courante au sein des colocations, plusieurs méthodes sont utilisées. Dans certains cas, on tient compte de tous les reçus dont le montant est partagé à la fin du mois. Dans d'autres, on tient un fond de caisse commun. Dans une collocation, il est donc essentiel de savoir quels montants l'on doit recevoir ou donner aux autres personnes et de maintenir un historique des dépenses.

L'un des avantages de la vie en colocation est que l'on peut compter les uns sur les autres. Par exemple, si le papier sulfurisé est fini ou une envie d'un yogourt fait surface l'un des colocataires peut passer au magasin en rentrant de l'école. La question est donc de savoir comment communiquer en temps réel les nécessités pour faire en sorte que le colocataire qui a la possibilité de se rendre en premier au magasin puisse se servir en temps réel de ces informations.

Le dernier besoin qu'on a pris en considération est celui de planifier les tâches ménagères telles que passer l'aspirateur, nettoyer la salle de bains ou faire la vaisselle. Afin d'éviter que de telles tâches ne soient pas exécutées, il est toujours utile d'établir des tournus. Souvent, l'on accroche sur le réfrigérateur un planning illustrant un tournus qui implique chaque membre de la colocation. Un utilisateur de l'application apprécierait donc la possibilité de visualiser et modifier un planning de ce genre. De plus, il devrait être en mesure d'attribuer les tâches dans un système de tournus.

2.2. Cas d'utilisation

Dans cette section, on décrit l'utilisation d'un service client générique engendré par la consommation de l'API. On remarque que le client élaboré pendant ce projet ne fournit pas toutes les fonctionnalités présentées ci-dessus car il s'agit d'une version incomplète. Plus précisément, avec ce client il n'est pas possible de gérer les images, la liste des courses et les tâches ménagères. En général, le client se veut un outil qui aide les colocataires dans la gestion de certains aspects de leur vie en commun. On fournit les descriptions détaillées des utilisateurs et des fonctionnalités disponibles. Pour illustrer notre explication, on inclut des diagrammes des cas d'utilisation qui indiquent les différents éléments qui peuvent être gérés.

2.2.1. Description des utilisateurs

L'utilisateur idéal de l'application est comme on l'a vu une personne qui vit dans une colocation, dont les besoins sont ceux décrit dans la Section 2.1. Dans la pratique, on peut distinguer trois catégories d'utilisateurs. La première comprend les utilisateurs qui ne sont pas connectés à un compte. Il faut préciser à ce propos que la plupart des fonctionnalités de l'application sont disponibles seulement aux utilisateurs munis d'un compte et connectés. Les autres catégories incluent les utilisateurs connectés au compte. Le critère discriminant est donc l'appartenance à un groupe appelé par "colocation" dans l'application. Les participants seraient uniquement les colocataires du monde réel, alors que d'autres potentiels utilisateurs devraient d'abord rejoindre le groupe ou en créer un pour se servir des fonctions permettant de gérer la vie en commun.

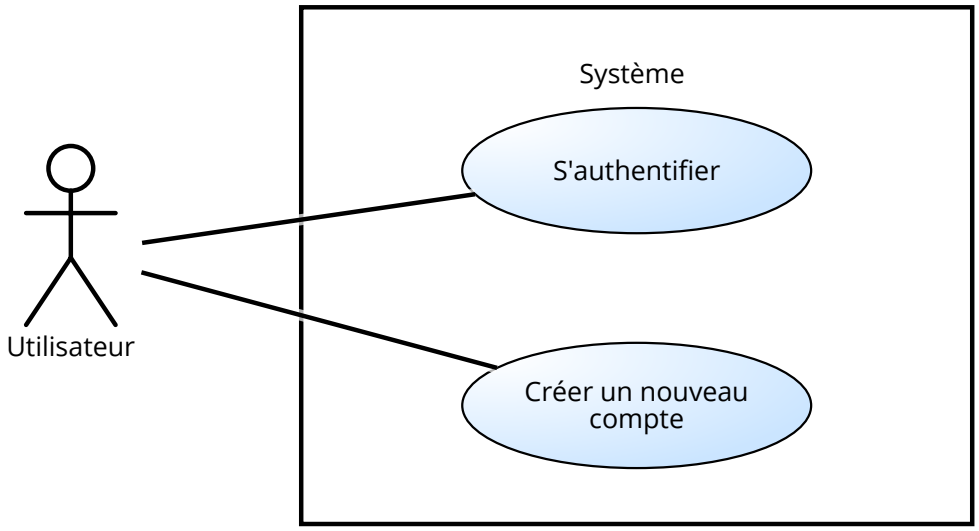

Figure 2.1. – Cas d'utilisation pour un utilisateur non connecté.

2.2.2. Connexion ou création du compte

La Figure 2.1 montre les actions réalisables dans le système par un utilisateur qui n'est pas connecté. Celui-ci peut s'authentifier en fournissant son adresse mail et son mot de passe. Par contre, si l'utilisateur n'a pas encore de compte il peut en créer un. Les informations à fournir sont une adresse mail, ses nom et prénom et le mot de passe. Une fois que l'utilisateur s'est authentifié, il peut accéder aux autres fonctionnalités du service.

2.2.3. Gestion du compte

Comme le montre la Figure 2.2 la gestion du compte est très simple. On peut modifier nom, prénom et mot de passe et ajouter une image de profil. De plus, il est possible d'afficher une seule image à la fois et si on veut la changer alors la nouvelle image écrase la précédente. Le service n'est pas un réseau social, donc on ne voit pas l'intérêt de fournir la possibilité de charger plusieurs images en même temps. Le client associe l'image au nom de l'utilisateur de façon similaire à ce qui se passe avec des applications comme WhatsApp afin de rendre plus intuitive l'interface.

2.2.4. Gestion de la colocation

En considérant l'objectif de l'application, il n'est pas surprenant qu'il soit nécessaire de créer une sorte de groupe réunissant les colocataires. Il est possible de faire partie seulement d'un groupe à la fois et toutes les données associées au groupe sont accessibles seulement par les membres du groupe. Cela signifie par exemple que quand on abandonne le groupe, on ne peut plus visualiser les messages envoyés et reçus. Le groupe en question s'appelle "colocation" et la Figure 2.3 montre comment le gérer.

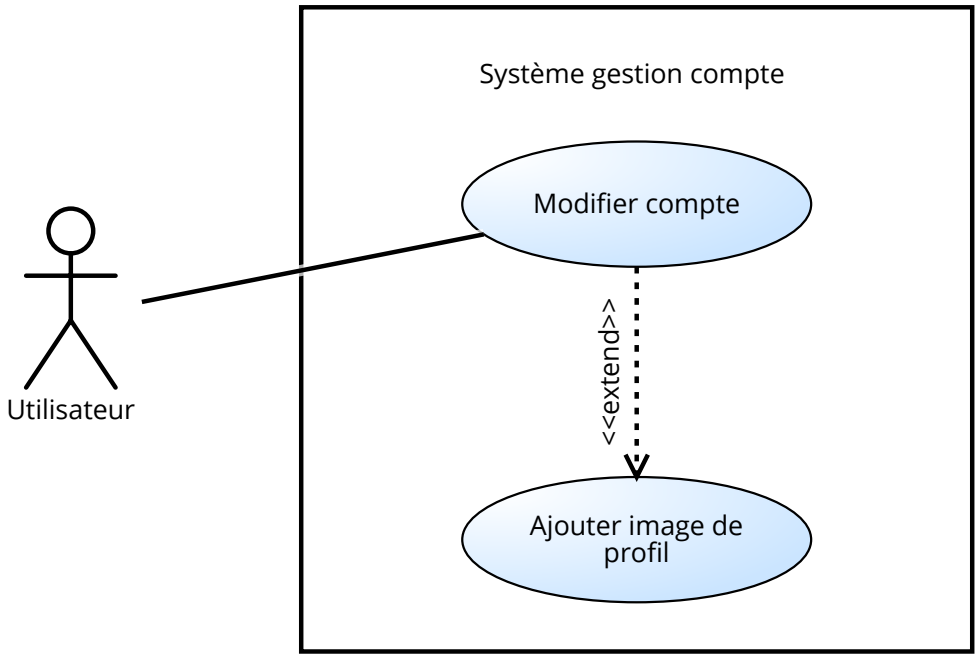

Figure 2.2. – Cas d'utilisation pour la gestion du compte.

On voit que l'utilisateur peut faire des choses différentes selon sa situation. S'il n'est pas dans une colocation, il existe deux façons pour en rejoindre une : créer une nouvelle colocation en choisissant un nom ou attendre qu'un autre utilisateur qui est déjà dans une colocation l'invite. Si par contre il est dans une colocation, il peut y ajouter d'autres utilisateurs grâce à une fonction de recherche qui marche avec les noms, prénoms ou adresses mail, abandonner la colocation ou ajouter une image pour la colocation. Comme pour l'image de profil, l'image de la colocation est aussi unique et son but et de permettre un minimum de personnalisation.

On souligne que toutes les fonctionnalités présentées dans le reste du chapitre concernent des données associées à une colocation qui sont donc accessibles uniquement à des utilisateurs qui font partie d'un groupe.

2.2.5. Gestion de messages

Si les colocataires ont besoin de communiquer quelque chose, ils peuvent utiliser un simple système de messagerie tel que décrit dans la Figure 2.4. Les messages contiennent seulement du texte. Ce qui est important à retenir est que les messages sont associés à la colocation et qu'ils sont lisibles par tous les colocataires. On ne peut pas envoyer de message privé ou visualiser les messages une fois qu'on a abandonné le groupe.

2.2.6. Gestion des dépenses

En ce qui concerne le traçage et le partage des dépenses communes, le service offre une solution illustrée dans le diagramme représenté dans la Figure 2.5.

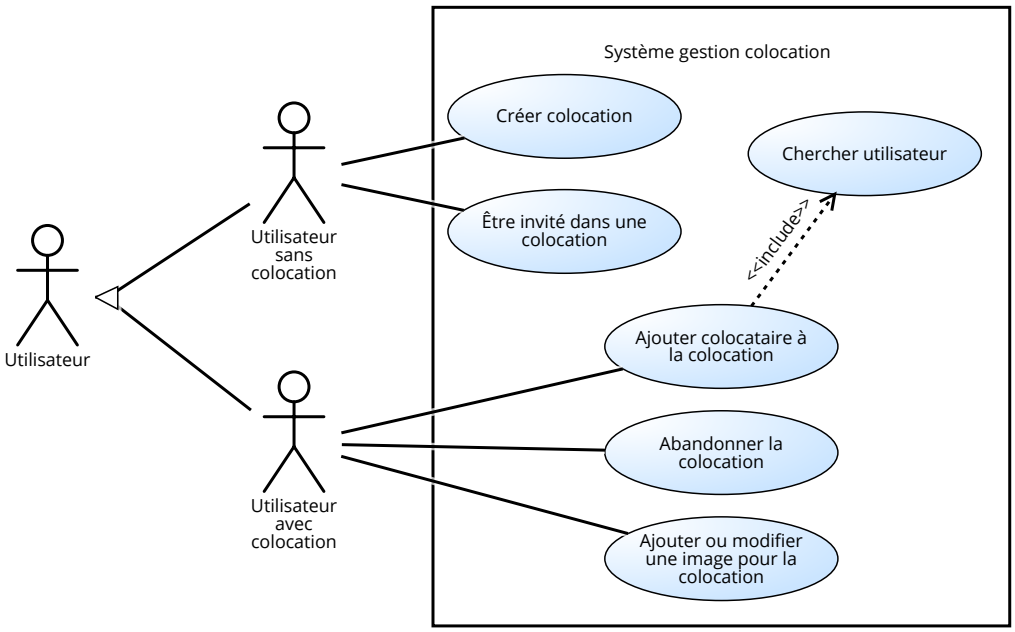

Figure 2.3. – Cas d'utilisation pour la gestion de la colocation.

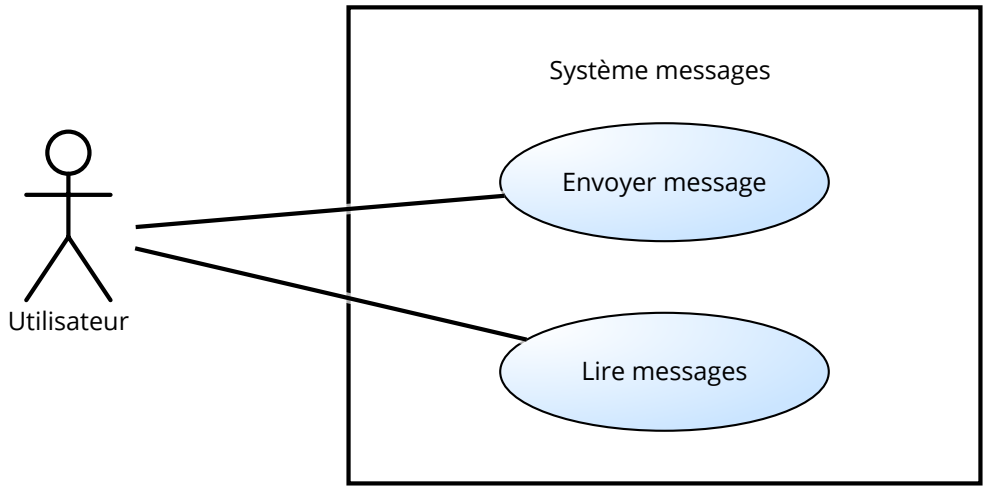

Figure 2.4. – Cas d'utilisation pour la gestion des messages.

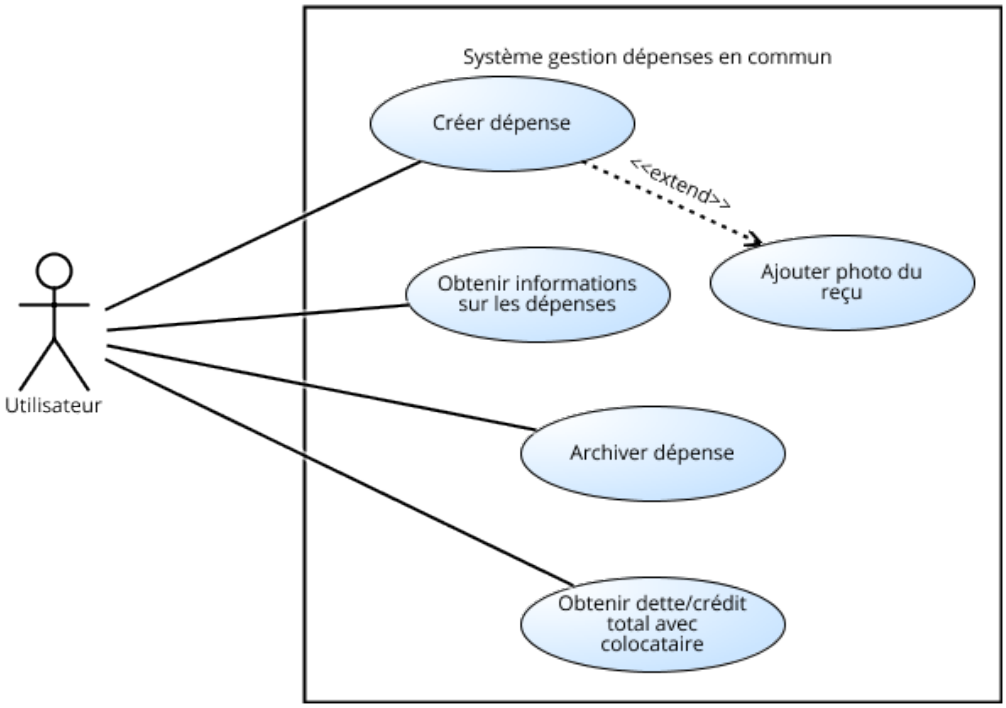

Figure 2.5. – Cas d'utilisation pour la gestion des dépenses.

L'idée est que chaque fois que l'un des colocataires fait un achat qui devrait être partagé avec les autres, il crée une "dépense" dans l'application. Pour créer une dépense, il faut fournir le coût, une description de l'article acheté et les noms des colocataires qui doivent contribuer. Il est très important que le colocataire qui a payé soit celui qui créé la dépense. Il a le choix de créer une dépense qui contient le total des courses ou de créer une dépense pour chaque article.

Le programme divise ensuite le montant total par le nombre de participants à la dépense et trace combien d'argent les contributeurs doivent payer. De plus, il est possible d'ajouter une - seule - image, idéalement une photo du reçu comme preuve d'achat. Ensuite, chaque colocataire peut accéder aux informations des dépenses qui le concernent selon différents critères : les dépenses où il doit de l'argent à quelqu'un, celles où quelqu'un lui doit de l'argent ou encore celles qui ont été archivées. Un utilisateur doit en effet archiver une dépense quand la dette a été payée. Il est possible de marquer comme payées les dettes correspondant seulement à une partie des colocataires concernées par la dépense et dans ce cas, on peut indiquer que la dépense est archivée seulement pour certains colocataires. Il est important de savoir que l'utilisateur qui a créé la dépense est le seul qui peut l'archiver, soit partiellement, soit entièrement.

Le dernier élément de la gestion des dépenses est la possibilité d'obtenir un bilan pour chaque utilisateur. Le programme peut en effet calculer automatiquement si l'utilisateur a une dette ou un crédit en précisant les montants en question. Si deux colocataires aplanissent les comptes en utilisant le bilan, il faut qu'ensuite les deux archivent toutes leurs dépenses en même temps.

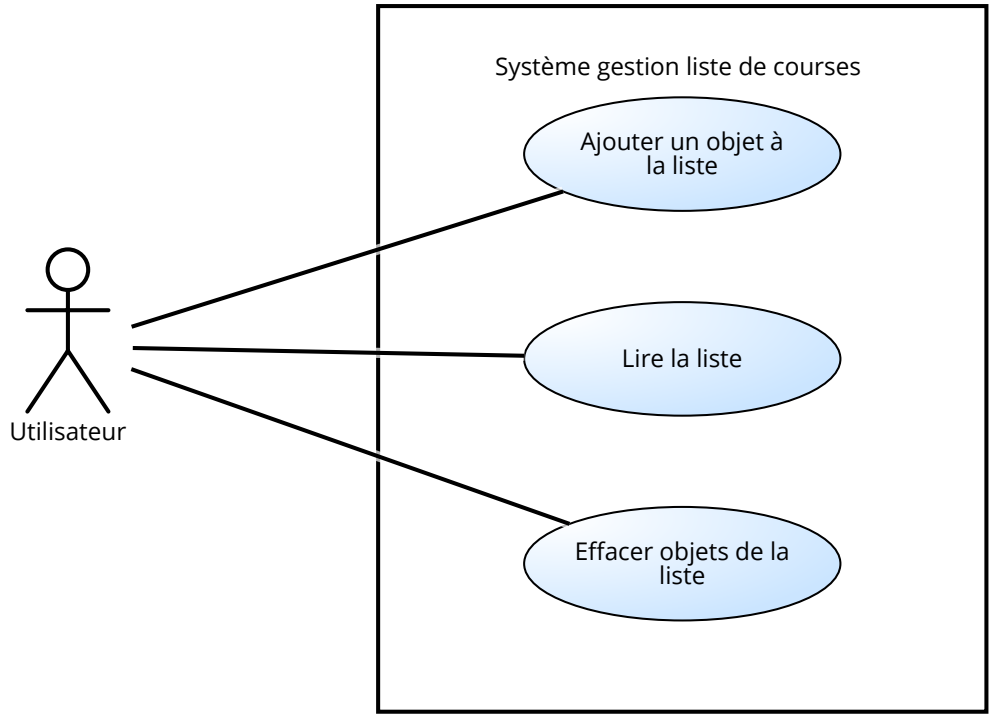

Figure 2.6. – Cas d'utilisation pour la gestion des liste de courses.

2.2.7. Gestion de la liste des courses

Une liste des courses commune fait partie des fonctionnalités du service. L'avantage est que chaque fois que l'un des colocataires fait des courses, il peut regarder s'il existe des requêtes de la part des autres, soient-elles pour des nécessités communes ou pour des besoins personnels. Le fonctionnement est très simple. Une seule liste par colocation contient tous les objets qui ont été ajoutés par ses membres. Un objet se compose d'une description de ce que l'on veut acheter et du nom du colocataire qui a déposé la requête.

Quand l'article souhaité a été acheté et que l'on n'a plus besoin qu'il apparaisse dans la liste, on peut l'éliminer. Il existe plusieurs manières d'éliminer les articles : tous ensemble, un à la fois, tous ceux ajoutés par soi-même, ou encore tous ceux ajoutés à un moment précédant une date donnée.

Un diagramme de cas d'utilisation pour les listes des courses peut être observé dans la Figure 2.6.

2.2.8. Gestion des tâches ménagères

On offre également un outil utile à l'organisation des tâches ménagères, un autre élément dont l'organisation pose souvent problème dans les colocations. Comme on peut le voir dans la Figure 2.7, le but est de créer des tâches et de déterminer ensuite les dates auxquelles il faut les réaliser.

Plusieurs points peuvent être soulevés en ce qui concerne la création de ces tâches. Il est possible de créer une tâche ponctuelle, qui ne se répète jamais, ou de choisir entre

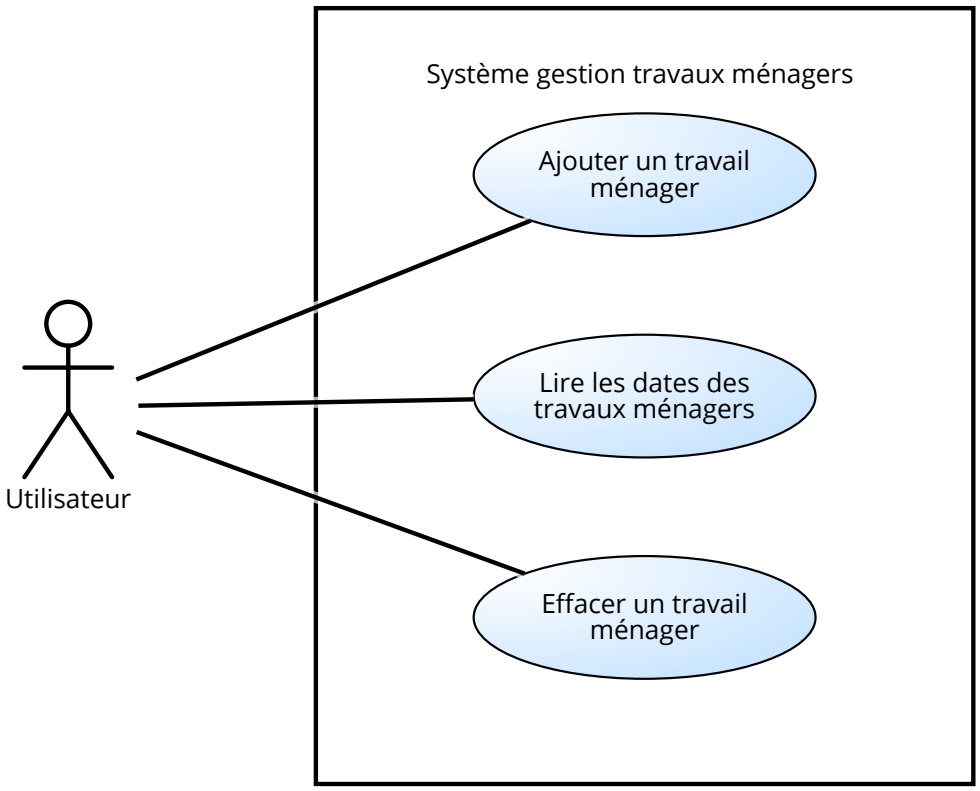

Figure 2.7. – Cas d'utilisation pour la gestion des tâches ménagers.

plusieurs types de récurrence. Dans les deux cas, il faut fournir une description de la tâche, et pour une tâche unique, il suffit d'insérer la date et le nom du colocataire à qui elle est attribuée.

Les tâches récurrentes se créent de manière légèrement différente. Il est nécessaire d'établir le type de récurrence entre ces quatre possibilités : chaque jour, chaque semaine (par ex.. chaque mardi), chaque énième semaine du mois (par ex. chaque mardi de la troisième semaine du mois) et chaque mois (par ex. le 3 de chaque mois). En outre, on peut également programmer des plages vides entre ces récurrences (par ex. chaque trois jours à la place de chaque jour, chaque mardi toute les deux semaines à la place de chaque mardi). Il est nécessaire de donner la date du début de la récurrence et il est possible aussi de donner une date de fin.

Le dernier pas est d'attribuer les tâches aux colocataires. En insérant plusieurs colocataires, le programme leur attribue la tâche selon un tournus. Ensuite, les utilisateurs peuvent visualiser les tâches dont ils doivent s'occuper entre deux dates. On remarque que chaque tâche est associée seulement à une date (et pas à un horaire), et que l'on reçoit seulement un calendrier des tâches. Il ne faut donc pas marquer une tâche comme ayant été exécutée. Enfin, il est possible d'effacer une tache si elle n'est plus nécessaire.

3
Mise en oeuvre

Dans ce chapitre, on montre comment et avec quels outils l'application a été développée. En premier lieu, on définit ce qu'est un stack MEAN et de quoi il se compose, en s'arrêtant brièvement sur l'API REST. Ensuite, on présente la base de données et sa structure, l'API REST et l'idée générale derrière ses endpoints, et le client à partir des technologies utilisées. On décrira également la structure de la base de données et on présentera l'idée générale derrière les endpoints de l'API REST. Enfin, on décrit ce qui a été implémenté dans le client et quels sont ses faiblesses.

3.1. Stack MEAN

MEAN est un ensemble de composants Open Source qui, ensemble, fournissent un framework end-to-end pour construire des applications Web dynamiques [5]. Avec ce framework il est possible de construire l'application de la base (base de données) jusqu'au sommet (client). Le mot MEAN est un acronyme indiquant les technologies qui composent le framework :

— MongoDB est une base de données orientée documents ;
— Express est un back-end web application framework ;
— Angular est un front-end web application framework ;
— Node.js est un environnement d'exécution pour JavaScript, utilisé pour le backend.

Dans la pratique on utilise donc MongoDB comme base de données persistante, Node.js et Express pour le back-end et Angular pour le client.

En particulier, comme backend on a développé une API REST, autrement dit RESTful API. L'acronyme API désigne une Application Programming Interface. Une API est une spécification des interactions possibles avec un composant logiciel [8].

L'acronyme REST indique un type d'architecture pour les systèmes hypermédia distribués [7]. La signification précise est "REpresentational State Transfer". L'architecture REST est définie par six principes directeurs.

1. Client-serveur : l'interface utilisateur est séparée du backend. Grâce à ça, il est facile d'améliorer la portabilité de l'interface et le serveur est plus simple.
2. Sans état : les requêtes du client sont complètes, dans le sens qu'elles n'ont pas besoin de contextes stockés sur le serveur pour être comprises.
3. Cachable : les données obtenues comme réponse à une requête peuvent être cachables ou pas. Si elles le sont, alors le client peut les utiliser pour une même requête plus tard.
4. Interface uniforme : une interface uniforme est utile pour simplifier l'architecture et améliorer la visibilité des interactions. Afin de l'obtenir, il est nécessaire d'appliquer quatre contraintes à l'architecture de l'interface : identification des ressources ; manipulation des ressources par des représentations ; messages autodescriptifs ; hypermédia comme moteur de l'état d'application.
5. Système stratifié : l'architecture est composée de couches hiérarchiques telles qu'elles ne peuvent interagir qu'avec les couches directement proches.
6. Code sur demande (facultatif) : le client peut étendre ses fonctionnalités en téléchargeant des codes.

En REST, toutes les informations sont considérées comme des ressources, qu'elles soient un document, une image ou un service temporel. En particulier, pour le projet on a utilisé surtout des ressources du type JSON.

3.2. Base de données

3.2.1. MongoDB

Le backend se sert d'une base de données pour stocker de manière permanente les données des utilisateurs. On a utilisé MongoDB qui est une base de données orientée documents opensource et gratuite [6]. Concrètement, cela signifie que les données sont stockées en toute flexibilité dans des documents de type JSON. Pour cette raison, il est simple de manipuler les données puisqu'elles sont mappées aux objets du code applicatif. Ces documents sont regroupés en collections. Une collection est l'équivalent d'une table RDBMS, mais il n'est pas strictement nécessaire que ses documents aient un schéma précis [4]. La version de MongoDB utilisée est v3.4.18.

3.2.2. Structure de la base de données

La base de données est structurée de façon telle que chaque élément qui peut-être géré par l'utilisateur dispose d'une collection. Les collections disponibles sont les suivantes :

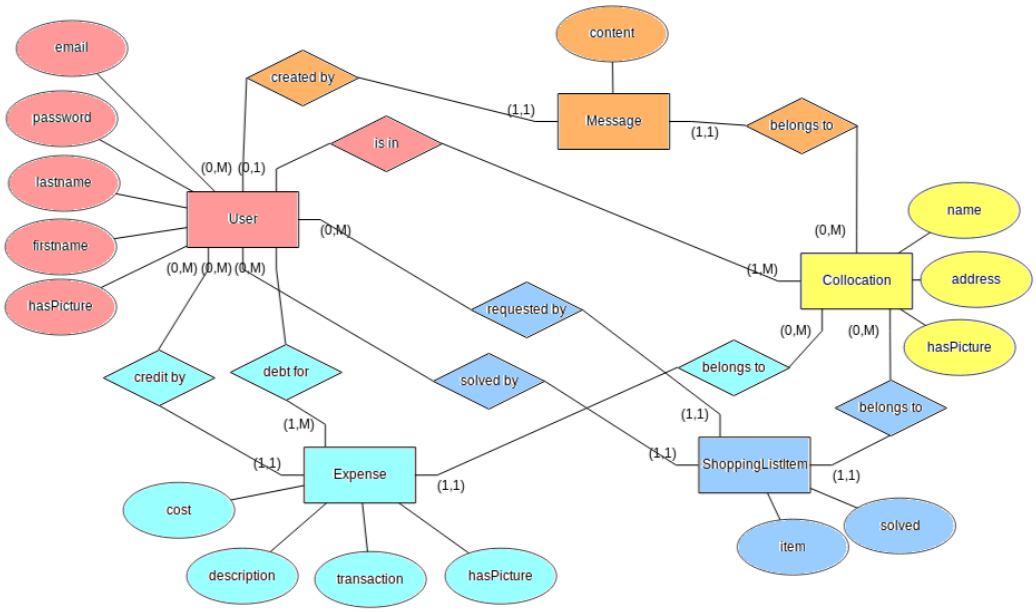

Figure 3.1. – Schéma ERM des collections User, Message, Collocation, Expense et ShoppingListItem.

utilisateurs, colocations, messages, dépenses, éléments des listes des courses et travaux ménagers.

Pour montrer les relations entre les collections, on a dessiné des diagrammes ERM [10]. Dans les diagrammes, les rectangles et les ellipses représentent respectivement les collections et leurs attributs. Les losanges représentent les relations entre les collections. Les paires proches d'une collection sont le nombre de relations qu'elle peut avoir. En particulier, le nombre de gauche est le nombre minimal et le nombre de droite est le nombre maximal. La lettre M indique une quantité non mieux précisée. Enfin, on a réuni visuellement chaque collection et ses champs en utilisant des couleurs.

Il peut paraitre étrange d'utiliser un diagramme ERM pour une base de données qui n'est pas relationnelle et la raison de ce choix est que ce diagramme est très efficace pour décrire la base de données. Dans l'implémentation, chaque attribut et chaque relation sont des champs de la collection. En particulier, les champs qui décrivent une relation contiennent l'identificateur (ou plusieurs identificateurs dans certains cas) du document de la collection en relation.

La Figure 3.1 représente les collections utilisateurs, colocations, messages, dépenses et éléments des listes des courses. Dans la Figure 3.2 on peut observer les collections utilisateurs, colocations et travaux ménagers. La raison d'une telle division est d'améliorer la lisibilité.

Les collections utilisateurs et colocations ont des relations avec toutes les autres collections. En général, chaque document d'une collection appartient à une colocation et est lié à un ou plusieurs utilisateurs. Dans la suite on va décrire les collections et leurs attributs. L'Appendice A.1 contient les détails de l'implémentation de la base des données. De plus on rappelle que chaque document a son propre identificateur.

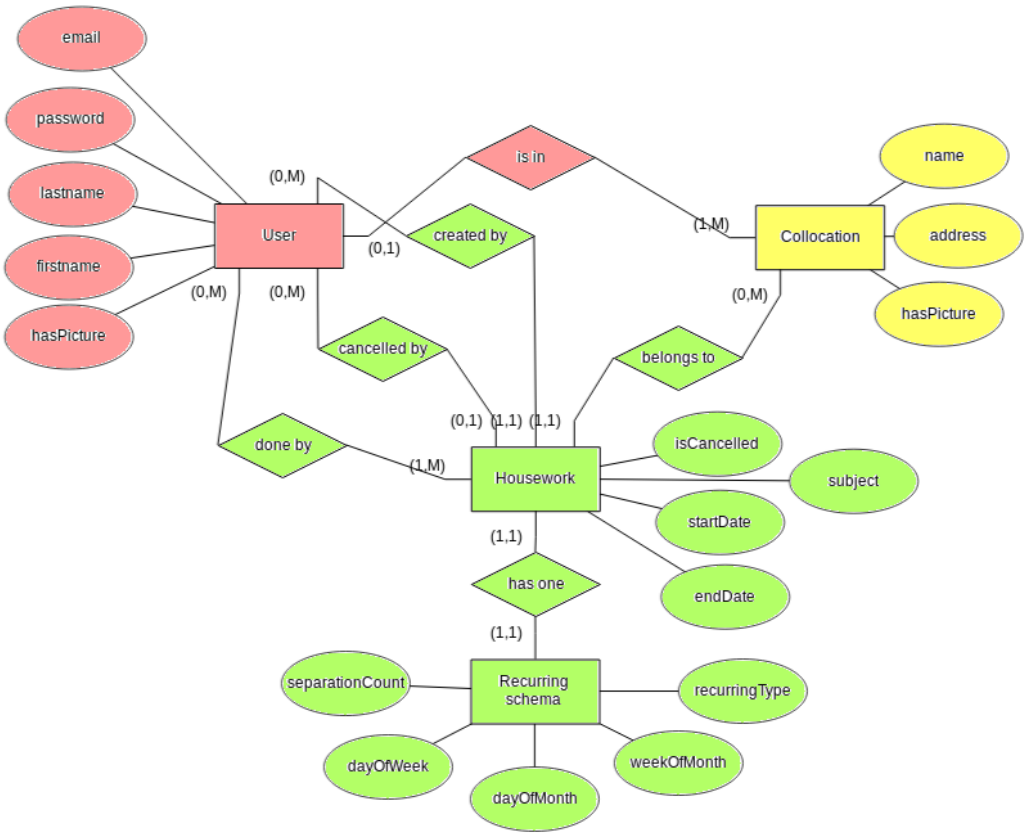

Figure 3.2. – Schéma ERM des collections User, Message et Housework.

L'utilisateur a les attributs suivants : une adresse mail, qui est unique et identifie l'utilisateur, un mot de passe pour gérer l'accès au compte, un nom et un prénom. Le mot de passe est sauvegardé de manière encryptée. De plus, le champ "hasPicture" indique si l'utilisateur a ou n'a pas chargé une image profil. Les collections colocations et dépenses ont aussi ce champ. Comme déjà expliqué plus haut, il n'est pas possible d'avoir plusieurs images par utilisateurs. L'attribut "hasPicture" est donc un boolean et la route pour retourner l'image utilise l'id de l'objet. La seule relation contenue directement dans la collection utilisateur indique la colocation à laquelle l'utilisateur appartient.

La collection colocations est plutôt simple. Elle contient un nom que ses membres peuvent donner, l'adresse et un attribut "hasPicture". Toutes les collections suivantes sont en relation avec les colocations. En effet, chaque élément appartient à exactement une colocation. de façon telle que seulement ses membres peuvent y accéder.

Un message a un contenu textuel et un utilisateur qui l'a créé.

Un document de la classe dépense est en relation avec plusieurs utilisateurs : un utilisateur qui a payé pour la dépense et les autres qui sont les débiteurs. Les champs de la collection sont donc le prix total payé, une description de la dépense, le dernier attribut "hasPicture" et un attribut "transaction" qui peut être "Open" ou "Closed" et indique si la dette a été entièrement remboursée. La relation avec les débiteurs possède elle-même un attribut "transaction" pour préciser si l'utilisateur a remboursé sa dette.

Un élément de la liste de courses est caractérisé par un champ qui est sa description et un autre champ qui indique si l'objet a été acheté. En ce qui concerne les relations, un utilisateur crée l'élément et un autre éventuel utilisateur l'achète.

Enfin, pour ce qui est de la collection des travaux ménagers, les relations sont créées ou effacées par un utilisateur et menées par un groupe d'utilisateurs. Le schéma est telle qu'il permet que les travaux soient récursifs sans se répéter plusieurs fois dans la base de données. Pour construire le schéma, on s'est basé sur un article qui explique comment gérer les évènements récursifs dans une base de données [9]. Une tâche ménagère a un sujet, une date de début et une date après laquelle elle ne serait plus répétée en cas de récursivité. Elle a donc un sous-schéma pour gérer la récursivité. Le premier attribut du sous-schéma est le type de récursivité. Pour une description de ces types, on peut consulter les explications dans la Section 2.2. Concrètement, le champ "type" peut prendre une des valeurs suivantes : "No", "Daily", "Weekly", "MonthlyDay", "MonthlyWeek". Le champ "separtionCount" indique combien de pauses il y a entre les répétitions. Enfin, les trois valeurs "dayOfWeek", "weekOfMonth", "dayOfMonth" correspondent respectivement au jour de la semaine (0 correspond au dimanche), à la semaine du mois (0 est la première semaine) et au jour du mois où la tâche doit être réalisée.

3.3. RESTful API

3.3.1. Node.js et Express

L'application back-end a été construite avec Node.js et Express.

Le premier est un environnement d'exécution pour JavaScript [1]. Il est asynchrone et événementiel et il est conçu pour construire des applications évolutives. Avec Node.js il est facile de gérer de multiples connections et on ne doit pas utiliser de threads. De plus,

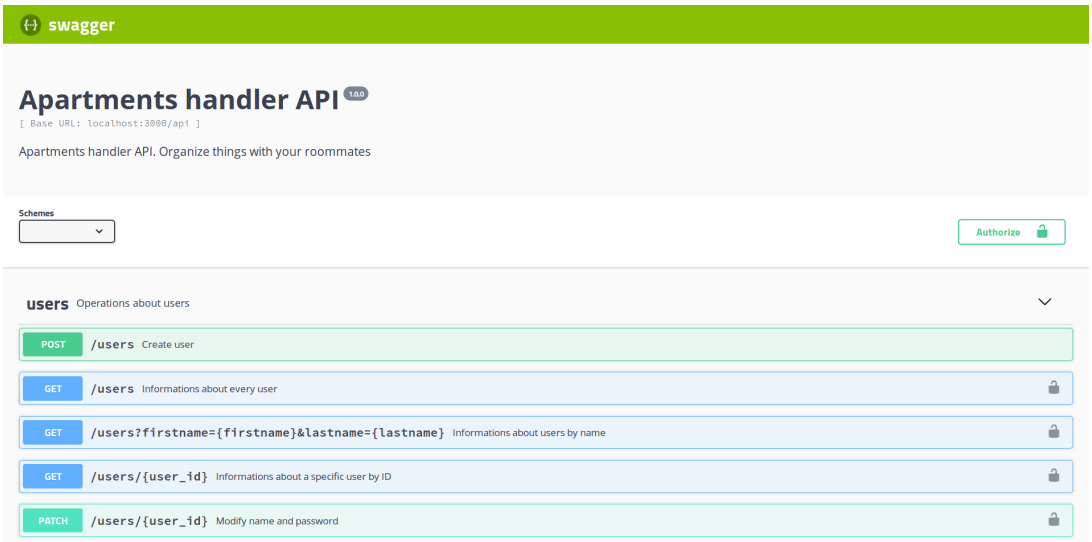

Figure 3.3. – Documentation interactive Swagger telle qu'elle apparaît dans le navigateur.

Node.js n'utilise presque pas de fonctions l'I/O, donc le processus ne s'arrête jamais. La version utilisée est la v10.8.0.

Le deuxième est un framework d'application web Node.js minimal et flexible qui fournit un ensemble robuste de fonctionnalités pour les applications web et mobiles [3]. Express est très utile pour développer des API. La version utilisée est la v4.16.3.

De plus on a utilisé Swagger qui est une suite d'outils de développement API permettant de concevoir, documenter et tester une API [2]. Il est possible de visualiser la documentation interactive dans un navigateur. On a utilisé la version v2.0 de Swagger et pour son implémentation on a écrit un document JSON. Dans la Figure 3.3, on peut voir à quoi cela ressemble. Tester l'API avec Swagger est plutôt intuitif. En cliquant sur l'un des endpoints, on ouvre une fenêtre qui décrit ce qu'il va faire, quels sont les paramètres et quelle est la réponse du serveur. Le bouton "Try it out" permet d'insérer les éventuels paramètres et d'exécuter la fonction. L'icône du cadenas indique que la fonction nécessite d'être un utilisateur autorisé. Il suffit de cliquer sur le bouton "Authorize" pour se connecter.

3.3.2. Endpoints

En général, un endpoint est une extrémité d'un canal de communication. Pour une API, les endpoints sont habituellement des URL qui permettent au client d'intéragir. En particulier, pour le cas REST, toutes les ressources sont accessibles par un URL unique. Pour spécifier le type d'intéraction avec la ressource, on utilise les méthodes HTTP GET, POST, PATCH et DELETE. Les méthodes sont respectivement pour lire, créer, modifier ou effacer une ou plusieurs ressources.

Les endpoints doivent respecter les principes REST décrits dans la Section 3.1. Puisque l'interface doit être uniforme, on a des règles précises aussi pour la structure de l'URL. Normalement, on met dans l'URL le nom de la ressource qu'on veut obtenir (par exemple : `/ressource`). De cette manière, on peut accéder à toutes les ressources portant un tel nom. Pour accéder à une ressource précise, il est nécessaire de spécifier son iden-

tificateur (par exemple : `/ressource/1`). L'URL pour la méthode POST pour créer une ressource n'a pas d'identificateurs. Il y a aussi la possibilité d'implémenter des filtres. Dans ce cas, l'URL doit contenir un point d'interrogation après le nom de la ressource et les noms des attributs filtrés, separés par un caractère "&" (par exemple : `/ressource?attribut1=toto&attribut2=titi`).

Pour en venir à l'application, toutes les collections de la base de données (voir Section 3.2.2) correspondent à une ressource. Les ressources existantes sont donc utilisateurs, colocations, messages, dépenses, éléments de liste de courses et travaux ménagers. De plus, on peut considérer également comme étant une ressource le bilan avec les colocataires et les images des utilisateurs, des colocations et des dépenses. Les interactions avec les ressources à travers les endpoints sont plutôt classiques et tombent dans les catégories lire, créer, modifier et effacer. Les interactions possibles avec l'API satisfant les cas d'utilisation montrés dans la Section 2.2. Pour connaître les détails de chaque endpoint, on renvoie à l'Appendice A.2.

Du fait que beaucoup de ressources sont forcément liées à une colocation précise, on a opté pour une structure arborescente des URLs. Par exemple, les messages sont associés à une colocation et donc l'URL pour la méthode GET pour les messages est : `/collocations/{id}/messages`. De même, les images dépendent d'une autre ressource et fonctionnent de manière similaire (par exemple : `/user/{id}/picture`). Le schéma dans la Figure 3.4 synthétise la structure des URLs. Les flèches indiquent la ressource parente qui est unique et donc possède aussi un identificateur. Par "Root", on désigne le début de l'URL. On remarque que les images sont aussi strictement dépendantes de la ressource parente et donc on peut les considérer comme ressources séparées (les images des utilisateurs sont une ressource différente des images des colocations).

La plupart des endpoints nécessitent aussi une authentification. On a utilisé la "basic authentication". Ce type d'authentification implique qu'à chaque requête on renvoie les informations d'accès, à savoir l'adresse mail et le mot de passe. L'authentification, en plus de permettre certaines actions seulement aux utilisateurs, permet également de délimiter l'accès aux ressources d'une colocation exclusivement à ses membres.

Il est nécessaire de préciser qu'un petit nombre d'endpoints ne respecte pas le principe REST "sans état". Il s'agit des endpoints qui donnent des résultats différents selon l'identité de l'utilisateur. Par exemple, l'URL `/users/me` est une méthode utile pour obtenir des informations sur soi-même. est une méthode utile pour obtenir des informations sur soi-même. Les autres URLs "coupables" sont ceux nécessaires à obtenir des informations sur les colocataires pour obtenir le bilan. Enfin, on signale la présence d'un endpoint utilisé pour les tests. Il s'agit d'une fonction qui réécrit la base de données, que l'on conseille de manipuler avec soin.

3.4. Client

3.4.1. Angular

Angular est un framework de JavaScript pour construire des applications [11]. Il met à disposition une bibliothèque riche visant la simplification d'aspects complexes comme la liaison des données, les routages et les animations. Le framework est idéal pour des applications qui utilisent beaucoup des données, que ce soit pour les collecter avec des

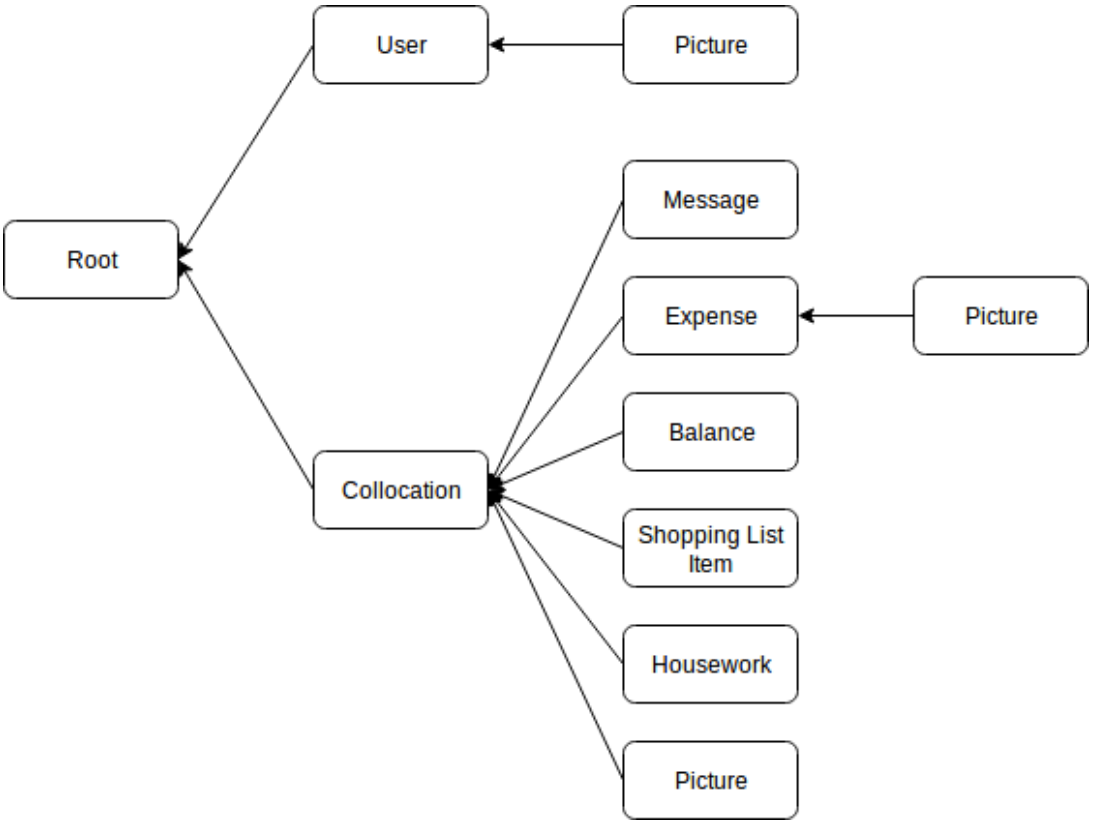

Figure 3.4. – Structure des URLs des endpoints.

formes ou pour les afficher et manipuler. Angular suit une approche par composantes. Il est donc facile de créer des composantes et de s'en servir plusieurs fois dans l'application. La version d'Angular utilisée est la v6.1.0.

3.4.2. Implémentation du client

Le focus du projet était le développement de l'API. Pour cette raison, le client dispose seulement de certaines fonctionnalités. Le client implémenté satisfait les cas d'utilisation illustrés dans les Sections 2.2.2, 2.2.3, 2.2.4, 2.2.5 et 2.2.6, sauf pour la gestion des images. En résumant, il s'agit de créer un compte et de s'y connecter, et de gérer ce compte, la colocation, les messages et les dépenses. Donc, en plus des images, le client ne permet pas de gérer une liste des courses (Section 2.2.7) et les tâches ménagères (Section 2.2.8) même si l'API est disponible. L'implémentation complète du client aurait été plus longue sans vraiment approfondir les connaissances d'Angular. Plus de détails sur l'utilisation du client sont décrits dans le chapitre 4.

D'un point de vue pratique, lorsque l'on accède au serveur, on télécharge l'application web du client dans le navigateur. À partir de ce moment, il est possible de visiter les pages de l'application qui affichent les données et contiennent des formes. Les données sont contenues dans des variables. Grâce à Angular, quand les variables changent, la page est toujours mise à jour, sans qu'il y ait besoin de la rafraîchir.

3.4.3. Faiblesses du client

Le client présente des faiblesses importantes qui ne permettent pas, actuellement, la diffusion commerciale.

Le problème le plus grave concerne la sécurité. Comme indiqué dans la section, on a utilisé un type d'authentification qui exige que les informations d'accès soient fournies à chaque requête. Ces informations sont donc stockées en base 64 dans une variable du type String. De plus, le site n'utilise pas le protocole HTTPS. Il est donc évident que les données d'accès sont facilement récupérables à travers des attaques comme Man in the Middle.

L'autre défaut est que l'interface est peu réactive : son adaptation est limitée aux dimensions des fenêtres et elle n'est pas vraiment adaptée aux smartphones et aux tablettes.

4

Client Appappa

Dans la première section, on explique comment est composée l'interface du client Appappa et quelles idées sont à la base de sa conception. Dans la deuxième, on propose un aperçu global du client à l'aide de nombreuses images. Page par page, on explique quelles sont les données montrées et ce que permettent les différentes formes et boutons.

La Figure 4.1 montre le logo d'Appappa.

Figure 4.1. – Le logo du client Appappa.

4.1. Interface

Le concept de base de l'interface est d'avoir acès à tout en même temps, sans confondre différents éléments. Par exemple, la page d'accueil contient deux cases séparées, une pour le login et l'autre pour la création d'un compte. Les autres pages ont une structure qui correspond aux nombre de choses contenues. Par exemple, la page des messages est simple et contient les messages et une forme pour les envoyer. Par contre, la page des dépenses contient plusieurs éléments : la création d'une nouvelle dépense, les bilans et une liste de dépenses. Pour améliorer la lisibilité, les différents composants sont contenus chacun dans une case distincte.

Une fois connecté, une barre apparait en haut de la page. Elle contient le logo, qui n'est pas interactif, un bouton pour se déconnecter et des icônes pour se déplacer entre les pages de l'application. Concrètement, il s'agit des liens vers les différents éléments à gérer, comme les messages ou les dépenses. On a choisi des icônes universelles pour permettre à l'utilisateur de comprendre facilement quelle icône correspond à quoi. Les deux bonshommes indiquent la gestion du compte et de la colocation, le bulle indique les messages, le symbole du dollar indique les dépenses, la liste indique la liste de courses et le calendrier indique les tâches ménagères. Dans un souci de clarté, la première page contient une case qui indique quelle icône correspond à quelle activité.

Enfin, l'application contient de nombreux boutons. S'il est possible de cliquer sur un bouton, celui-ci s'affiche en bleu, dans le cas contraire il est gris.

4.2. Pages du client

4.2.1. Page d'accueil

La Figure 4.2 illustre la page d'accueil qui est la seule partie accessible à des utilisateurs non connectés. Elle est très simple et se compose du logo de l'application en haut et de deux cadres. Le cadre de gauche est destiné à accueillir le login et le cadre de droite est destiné à la création du compte. La zone de texte pour l'email force l'insertion d'une adresse grâce à une expression régulière et celle pour le mot de passe cache l'input. La création d'un compte enregistre automatiquement le nouvel utilisateur. D'éventuelles tentatives de connection avec des valeurs incorrectes ou de création de compte avec une adresse email déjà existante renvoie vers une page d'erreur.

4.2.2. Page centrale

La page centrale est celle à partir de laquelle on démarre l'expérience comme utilisateur. Sur cette page et sur toutes les autres qui suivent la barre en haut, décrite dans la Section 4.1, est toujours présente.

Sur cette page est fourni un aperçu de son propre compte et de la colocation, ainsi que les boutons pour accéder aux activités de l'application. Puisque l'utilisateur n'est pas forcément dans une colocation, la page peut présenter deux aspects différents. La Figure 4.3 montre la page pour un utilisateur sans colocation, alors que la Figure 4.4 montre la page pour un utilisateur avec colocation.

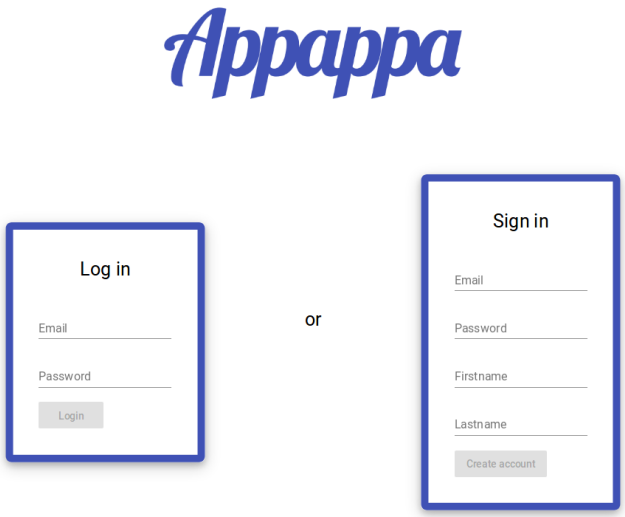

Figure 4.2. – Page d'accueil.

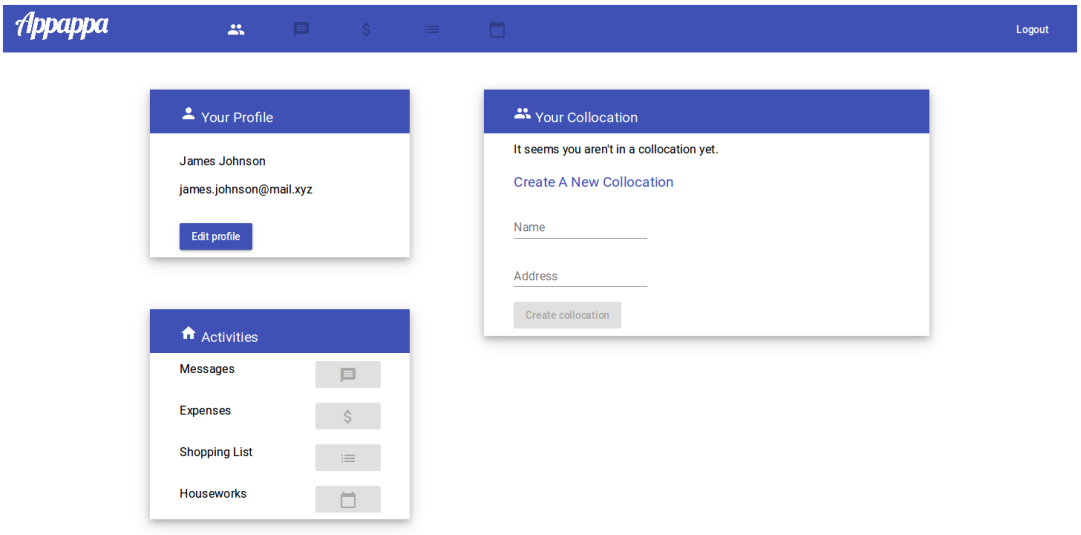

Figure 4.3. – Page centrale pour un utilisateur sans colocation.

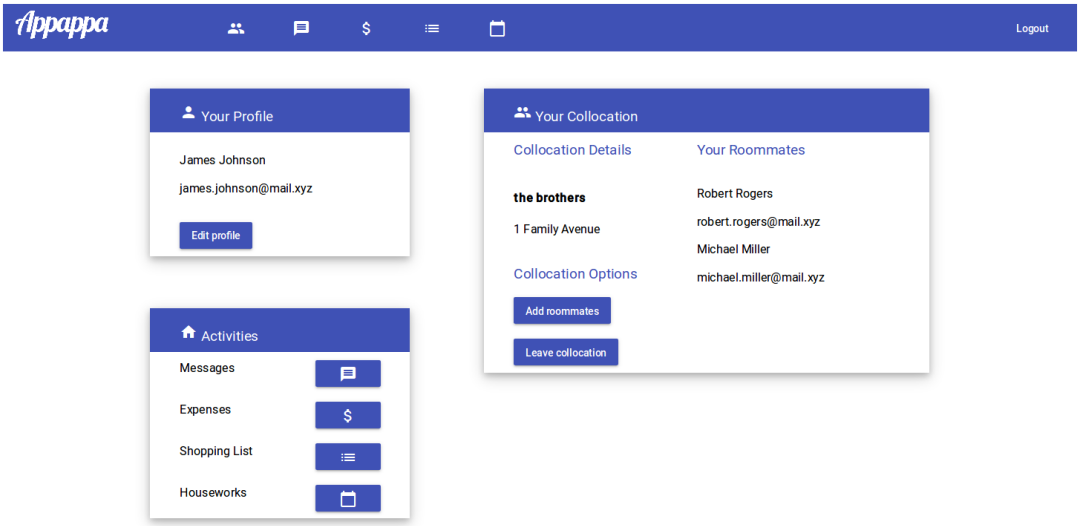

Figure 4.4. – Page centrale pour un utilisateur avec colocation.

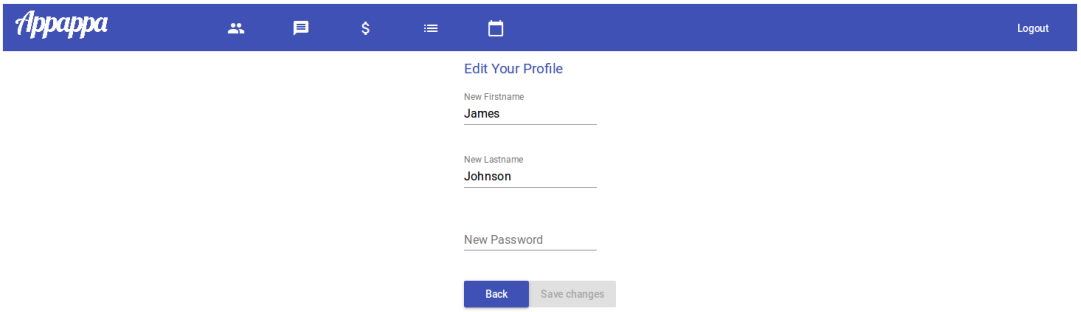

Figure 4.5. – Page de modification de l'utilisateur.

À partir de cette page, on peut modifier le profil d'utilisateur, créer ou laisser une colocation, y rajouter des colocataires ou encore aller aux pages des activités. Enfin, on remarque que si l'on n'est pas dans une colocation, les boutons des activités sont gris. Cela signifie qu'ils ne sont pas accessibles.

4.2.3. Page de modification de l'utilisateur

La Figure 4.5 montre la page qui permet de modifier certains paramètres de son propre profile. Les paramètres qui peuvent être modifiés sont le nom, le prénom et le mot de passe.

4.2.4. Page pour ajouter les colocataires

Pour ajouter des colocataires à son propre groupe, il est nécessaire de passer par cette page, présentée dans la Figure 4.6. On y trouve une liste qui contient tous les utilisateurs. Des filtres sont applicables selon le nom, prénom et/ou adresse mail. Quand un utilisateur n'appartient pas à une colocation, on peut l'ajouter à sa propre colocation à travers le bouton "Add Roommate".

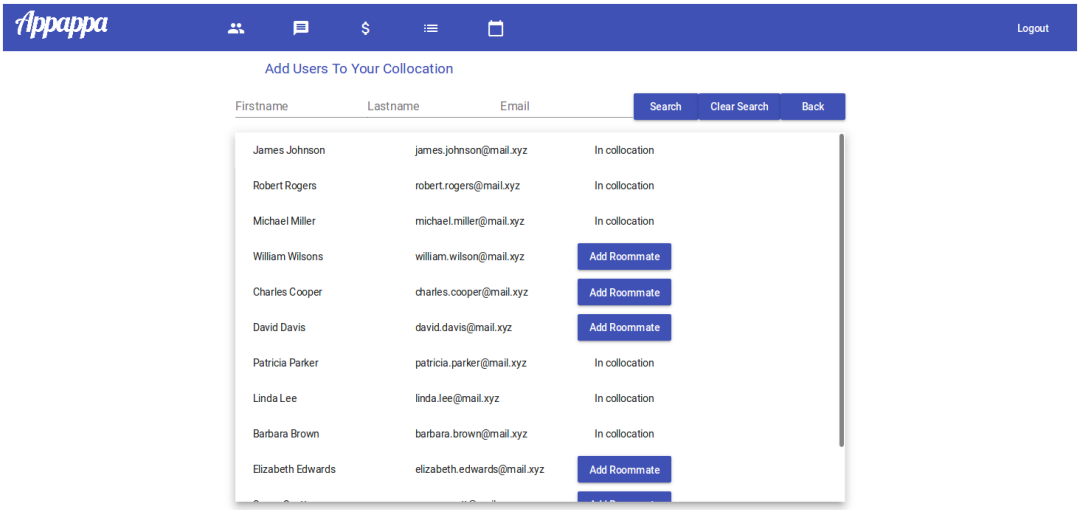

Figure 4.6. – Page pour ajouter les colocataires.

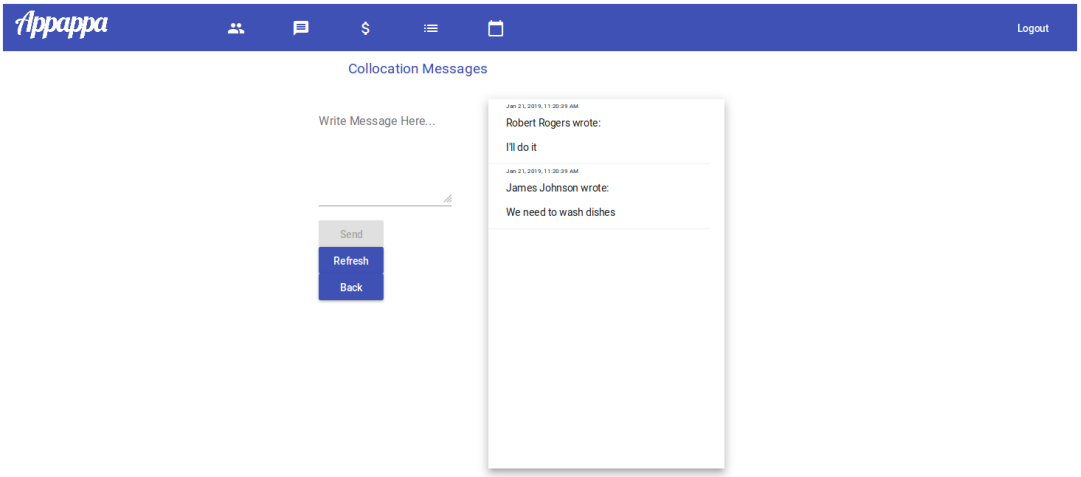

Figure 4.7. – Page pour envoyer et lire des messages.

4.2.5. Page des messages

La page pour les messages (Figure 4.7) présente un layout similaire à celui d'un chat. À gauche se trouve une zone de texte pour écrire les messages et un bouton pour les envoyer. À droite, se trouve l'historique des messages de la colocation, avec les messages les plus récents en haut. Puisqu'il ne s'agit pas vraiment d'un chat, on a aussi mis un bouton "Refresh" pour mettre à jour les messages.

4.2.6. Page des dépenses

Dans la Figure 4.8 est illustrée la page des dépenses, construite sur la base de trois cadres.

Le premier sert à créer une nouvelle dépense dans la base de données. Lorsque l'on a payé quelque chose pour les autres, on doit utiliser cette fonctionnalité. Il est possible de marquer, grâce aux cases à cocher, quels colocataires ont bénéficié de la dépense et doivent

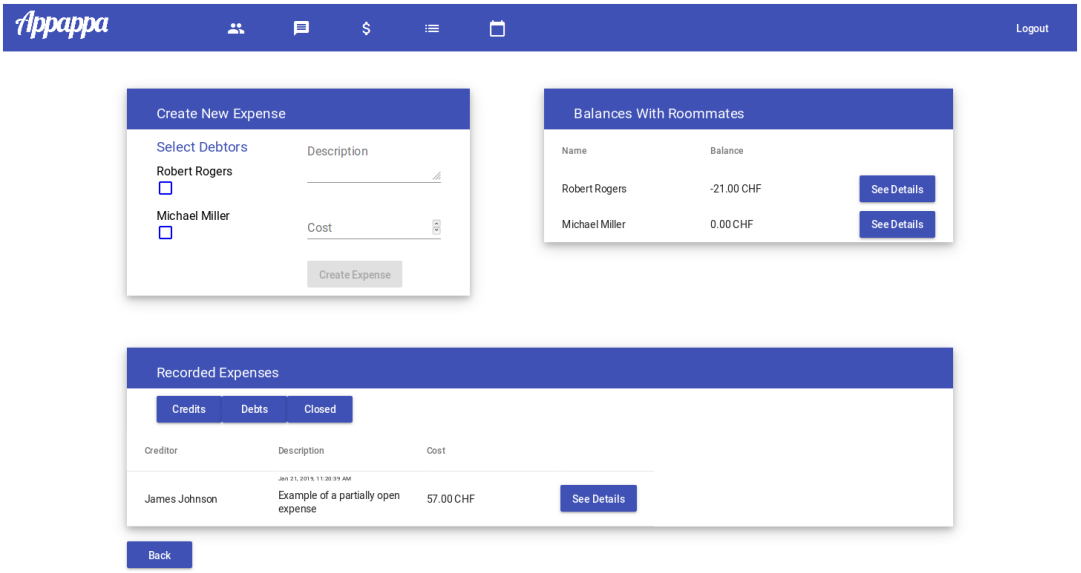

Figure 4.8. – Page des dépenses.

donc y contribuer. Il faut aussi spécifier le prix total payé et fournir une description pour justifier l'élément (par ex. :"achat du lait").

Le deuxième montre la balance avec les colocataires. Si un chiffre positif est indiqué, le colocataire nous doit cette somme. Vice-versa, un chiffre négatif indique une dette envers lui. En cliquant sur le bouton "See details", on accéde à une autre page pour voir les détails des dettes et crédits avec le colocataire.

Le dernier cadre montre les dépenses créées dans la colocation. Il est possible de visualiser les crédits , les dettes et les dépenses déjà reglés et archivés par l'utilisateur. Le bouton "See details" permet d'accéder aux détails de chaque dépense.

4.2.7. Page des détails d'une dépense

Dans cette page, illustrée par la Figure 4.9, on peut voir les détails d'une dépense. Elle contient le nom du créditeur, la date de création, la description, les coûts totaux et individuels, les noms des débiteurs et s'ils ont déjà payé leur partie. Si on est le créditeur de la dépense qu'on visualise, on a aussi accès à des boutons pour confirmer qu'un des débiteurs à remboursé ("Payment Confirmation") ou que tout le monde a remboursé ("Close Expense").

4.2.8. Page des détails d'un bilan

Dans les détails d'un bilan (Figure 4.10) sont résumés les dettes et crédits d'un colocataire et le bilan total. Grâce au bouton "See details", on accède à la page des détails des dépenses. Si on veut marquer comme ayant été réglés tous les crédits que le colocataire à envers nous, on peut utiliser le bouton "Close Expenses". On rend attentif au fait que l'utilisation de cette option ne réduit pas forcément le bilan à zéro, puisque les dettes qu'on a envers le colocataire sont encore ouvertes. Pour les réduire à zéro, il faut que le colocataire clique sur le même bouton dans sa page.

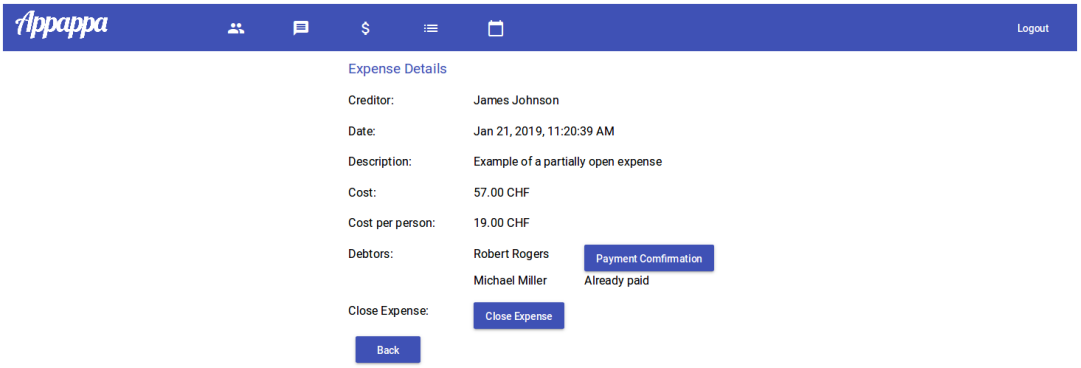

Figure 4.9. – Page des détails d'une dépense dont on est créditeur.

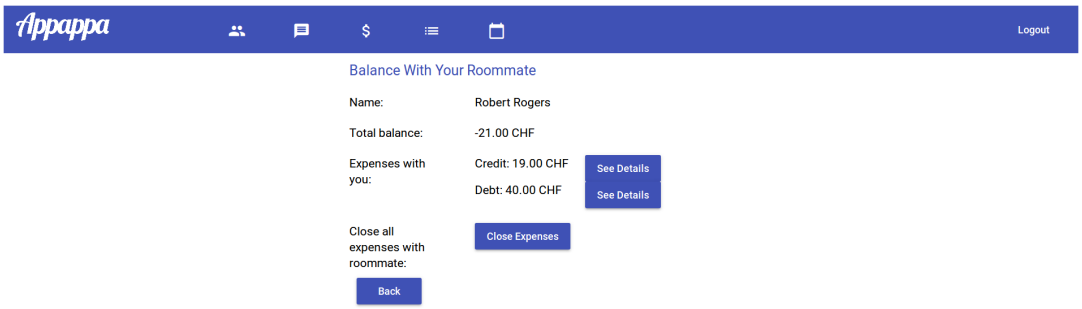

Figure 4.10. – Page des détails d'un bilan.

4.2.9. Page pour liste de courses et tâches ménagères

Comme anticipé dans la Section 3.4.2, on n'a pas implémenté les fonctionnalités pour la liste de courses et les tâches ménagères. Les pages sont accessibles mais vides et la Figure 4.11 montre à quoi elles ressemblent.

4.2.10. Page d'erreur

La dernière page du client est la page d'erreur. Elle apparaît si on se trompe en essayant de faire le login ou si on veut créer un utilisateur avec une adressemail déjà existante. La page indique le statut de l'erreur et un message obtenu comme réponse de l'API, plus un bouton pour revenir en arrière. La Figure 4.12 montre un exemple de page d'erreur. La page d'erreur a été très utile pendant le développement pour le debugging.

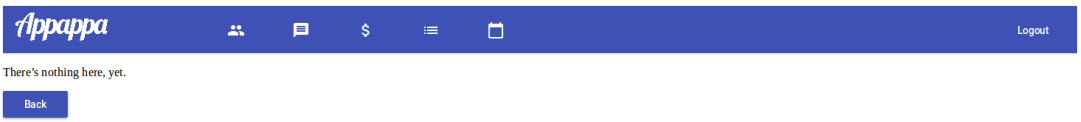

Figure 4.11. – Page vide pour liste des courses et tâches ménagères.

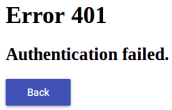

Figure 4.12. – Page d'erreur pour un erreur d'accès.

5 Conclusion

5.1. Développer avec le stack MEAN

Le but ultime du projet était de développer une application web en utilisant le stack MEAN. Au cours de ce travail, on a pu apprécier les avantages d'une telle approche. Avec ce framework, il est très intuitif de séparer base de données, RESTful API et client et de construire le logiciel à partir de rien. En même temps, on a pu apprécier le fait que les différentes couches utilisent les mêmes formats document JSON pour les données, ce qui aide la cohérence du programme et la communication entre ses parties.

On a pu également constater que MongoDB est une base de données extrêmement flexible et simple. L'utilisation de Node.js pour démarrer un serveur est rapide. De plus, en combinaison avec le gestionnaire de paquets npm, Node.js peut accéder à un nombre très élevé de modules . Express, en tant que frameworks parmi les plus populaires pour Node.js, peut compter sur beaucoup de support dans la documentation et dans le web. Angular est un outil très puissant mais aussi un peu lourd. Une fois qu'on a appris les bases, on a apprécié le système des composantes qui simplifie la réutilisation d'éléments dans une page web.

En conclusion, l'expérience tirée de l'utilisation du stack MEAN est positive. On a construit une application complète même si les connaissances préalables étaient presque nulles.

5.2. Difficultés rencontrées

La principale difficulté rencontrée au cours de ce projet relève de l'apprentissage. En effet, pour développer l'application full stack on a dû apprendre à maîtriser MongoDB, Node.js et Angular sans aucune connaissance préalable. Pour ce faire, on a commencé à développer de petites applications, à suivre des tutoriels et à faire de l'ingénierie inversée

sur des exemples déjà existants. Stack Overflow a été un important outil de référence tout au long de ce parcours.

Cette approche basée sur l'apprentissage en cours d'usage nous a permis d'acquérir beaucoup de connaissances mais nous a aussi mené à certaines erreurs sur le plan conceptuel. Par exemple, on a développé des endpoints qui n'étaient pas REST ou on a fait tourner l'API et le client dans la même application.

L'élaboration de ce projet nous a poussé a surmonter avec persévérance la plupart des obstacles et à développer une grande indépendance dans la résolution des problèmes.

5.3. Améliorations possibles

Ce travail étant le fruit d'une première expérience avec beaucoup de technologies et de notions différentes, la marge d'amélioration possible reste assez grande, soit au niveau de la base de données, soit au niveau de l'API ou du client.

L'élément principal à relever à ce propos est la sécurité. En effet, non seulement on n'utilise pas le protocole HTTPS, mais les données d'accès sont gérées de manière non sûre. Pour combler ce manque, il faudrait utiliser JSON Web Token, un format standard pour jetons d'authentification.

Un autre aspect améliorable concerne les mots de passe. Quand on crée un mot passe ou on essaye de le modifier, il faut l'insérer une seule fois et non deux comme il est couramment d'usage. Les données contenues dans la base de données, à l'exception des mots de passe, ne sont pas encryptées et on est donc face à un problème de confidentialité. Toujours en ce qui concerne la confidentialité, tous les noms et adresses mail des comptes inscrits sont visibles par tous les usagers.

De plus, la façon d'inviter des participants dans une colocation n'est pas acceptable pour une application professionnelle car il ne s'agit pas d'une réelle invitation à laquelle on peut répondre et l'utilisateur est directement intégré dans le groupe Afin de palier cela, on suggère par exemple un système similaire à celui de demandes d'amitié de Facebook.

Enfin, l'interface du client pourrait être plus réactive si elle était développée avec bootstrap.

A
Annexes

A.1. Implémentation de la base de données

```
var userSchema = new Schema(
  {
    email: {
      type: String,
      required: true
    },
    password: {
      type: String,
      required: true
    },
    firstname: {
      type: String,
      required: true
    },
    lastname: {
      type: String,
      required: true
    },
    collocation_id: ObjectId,
    hasPicture: {
      type: String,
      default: 'False'
    }
  }
);
```

Listing A.1 – Implementation de la collection User

```
var collocationSchema = new Schema(
    {
        name: {
          type: String,
          required: true
        },
        address: {
          type: String,
          required: true
        },
        hasPicture: {
          type: String,
          default: 'False'
        }
    }
);
```

Listing A.2 – Implementation de la collection Collocation

```
var messageSchema = new Schema(
    {
        collocation_id: ObjectId,
        createdBy_id: ObjectId,
        content: {
          type: String,
          required: true
        }
```

```
    }
);
```

Listing A.3 – Implementation de la collection Message

```
var debtorSchema = new Schema(
  {
    user_id: ObjectId,
    transaction: {
      type: String,
      default: 'Open'
    }
  }
);

var expenseSchema = new Schema(
  {
    description: {
      type: String,
      required: true
    },
    collocation_id: ObjectId,
    creditor_id: ObjectId,
    debtorsList: [debtorSchema],
    cost: {
      type: Number,
      required: true
    },
    transaction: {
      type: String,
      default: 'Open'
    },
    hasPicture: {
      type: String,
      default: 'False'
    }
  }
);
```

Listing A.4 – Implementation de la collection Expense

```
var shoppingListItemSchema = new Schema(
    {
        requestBy_id: ObjectId,
        collocation_id: ObjectId,
        item: {
          type: String,
          required: true
        },
        solved: {
          type: String,
          default: 'False'
        },
        solvedBy_id: ObjectId
    }
);
```

Listing A.5 – Implementation de la collection ShoppingListItem

```
var userIdSchema = new Schema(
  {
    user_id: ObjectId
  }
);

var recurringSchema = new Schema(
  {
    recurringType: {
      type: String,
      required: true
    },
    separationCount: Number,
    dayOfWeek: Number,
    weekOfMonth: Number,
    dayOfMonth: Number
  }
);

var houseworkSchema = new Schema(
  {
    createdBy_id: ObjectId,
    collocation_id: ObjectId,
    subject: {
      type: String,
      required: true
    },
    usersList: [userIdSchema],
    startDate: {
      type: Date,
      required: true
    },
    endDate: {
      type: Date
    },
    isCancelled: {
      type: String,
      default: 'False'
    },
    cancelledBy_id: ObjectId,
    recurring: recurringSchema
  }
);
```

Listing A.6 – Implementation de la collection Housework

A.2. Endpoints

Dans ce chapitre, on décrit tous les endpoints de l'application. Pour ce faire, on dessin des tableaux. La première ligne de chaque table est une capture d'écran de la documentation Swagger qui contient beaucoup d'informations : la méthode HTML utilisée, l'URL, une brève description de l'utilisation de l'endpoint en anglais et enfin, la présence du cadenas indique s'il faut se connecter au compte. Souvent, dans l'URL il y a des mots entourés d'accolades. Ces mots sont appelés techniquement "paramètres" et doivent être substitués avec la valeur qui, dans la majorité des cas, indiquent des identifieurs. Ensuite les autres lignes des tables décrivent quel type de document JSON il faut envoyer (body), les filtres possibles pour l'endpoint et le contenu de la réponse du serveur (dans le cas où celle-ci est positive). Pour une description plus précise des documents JSON utilisé comme body ou réponse par le serveur, voir la documentation Swagger (setup : Appendix A.3).

A.2.1. Utilisateurs

POST /users Create user	
Body	Données base pour l'utilisateur.
Filtres	-
Réponse	Utilisateur crée.

GET /users Informations about every user	
Body	-
Filtres	Il est possible de filtrer la requête pour n'importe pas quel attribut de l'utilisateur.
Réponse	Liste de utilisateurs.

GET /users/{user_id} Informations about a specific user by ID	
Body	-
Filtres	-
Réponse	Un utilisateur.

PATCH /users/{user_id} Modify name and password	
Body	Les informations que on veut changer.
Filtres	-
Réponse	L'utilisateur modifié (il faut être l'utilisateur indiqué).

GET /users/me Informations about themselves	
Body	-
Filtres	-
Réponse	L'utilisateur qui a envoyé la requête.

POST /users/{user_id}/picture Uploads a profile picture.	
Body	L'image.
Filtres	-
Réponse	Message de succès (il faut être l'utilisateur indiqué).

GET /users/{user_id}/picture See the picture of a specific user by ID	
Body	-
Filtres	-
Réponse	L'image profile de l'utilisateur indiqué.

DELETE /users/{user_id}/picture Remove profile picture	
Body	-
Filtres	-
Réponse	Message de succès (il faut être l'utilisateur indiqué).

A.2.2. Colocations

POST /collocations Create collocation	
Body	Données base pour la colocation.
Filtres	-
Réponse	Colocation crée.

DELETE /collocations Leave collocation	
Body	-
Filtres	-
Réponse	Message de succès.

GET	`/collocations/{collocation_id}` Informations about the collocation
Body	-
Filtres	-
Réponse	Une colocation.

PATCH	`/collocations/{collocation_id}` Add roommate to the collocation
Body	Utilisateur qu'on veut ajouter.
Filtres	-
Réponse	La colocation.

GET	`/collocations/{collocation_id}/roommates` Informations about the roommates
Body	-
Filtres	-
Réponse	Une liste d'utilisateurs.

POST	`/collocations/{collocation_id}/picture` Uploads a collocation picture.
Body	L'image.
Filtres	-
Réponse	Message de succès.

GET	`/collocations/{collocation_id}/picture` See the picture of your collocation
Body	-
Filtres	-
Réponse	L'image profile de la colocation.

DELETE	`/collocations/{collocation_id}/picture` Remove collocation picture
Body	-
Filtres	-
Réponse	Message de succès.

A.2.3. Messages

POST /collocations/{collocation_id}/messages Send message	
Body	Contenu du message
Filtres	-
Réponse	Le message envoyé.

GET /collocations/{collocation_id}/messages Read messages	
Body	-
Filtres	-
Réponse	Liste de messages.

A.2.4. Dépenses

POST /collocations/{collocation_id}/expenses Create expense	
Body	Données base pour un dépense.
Filtres	-
Réponse	La dépense crée.

GET /collocations/{collocation_id}/expenses Show all expenses	
Body	-
Filtres	On peut sélectionner uniquement les dépenses ouvertes ou archivées.
Réponse	Liste de dépenses.

PATCH /collocations/{collocation_id}/expenses Close all expenses with all users	
Body	-
Filtres	On peut choisir de fermer les dépenses ouvertes uniquement avec un colocataire précis.
Réponse	Liste de dépenses fermées.

GET `/collocations/{collocation_id}/expenses/{expense_id}` Show one expense	
Body	-
Filtres	-
Réponse	La dépense.

PATCH `/collocations/{collocation_id}/expenses/{expense_id}` Close one expense with all users	
Body	-
Filtres	On peut choisir de fermer la dépenses uniquement avec un colocataire précis.
Réponse	La dépense fermée.

GET `/collocations/{collocation_id}/expenses_credit` Show open credits	
Body	-
Filtres	-
Réponse	Liste de dépenses.

GET `/collocations/{collocation_id}/expenses_debt` Show open debts	
Body	-
Filtres	-
Réponse	Liste de dépenses.

GET `/collocations/{collocation_id}/expenses_balance` Show balances with every roommate	
Body	-
Filtres	-
Réponse	Liste de bilans.

POST `/collocations/{collocation_id}/expenses/{expense_id}/picture` Uploads a receipt picture.	
Body	L'image.
Filtres	-
Réponse	Message de succès.

GET /collocations/{collocation_id}/expenses/{expense_id}/picture — See the picture of your expense	
Body	-
Filtres	-
Réponse	L'image de la dépense.

DELETE /collocations/{collocation_id}/expenses/{expense_id}/picture — Delete receipt picture.	
Body	-
Filtres	-
Réponse	Message de succès.

A.2.5. Liste des courses

POST /collocations/{collocation_id}/shoppinglists — Create an item for the shopping list	
Body	L'objet que on veut ajouter à la liste.
Filtres	-
Réponse	L'objet crée.

GET /collocations/{collocation_id}/shoppinglists — Shows the whole shopping list	
Body	-
Filtres	On peut filtrer la liste pour exclure les objets qui ont déjà été marqué comme achetés ou les objets insérés avant une certaine date.
Réponse	La liste des courses.

PATCH /collocations/{collocation_id}/shoppinglists — Marks as solved the whole shopping list	
Body	-
Filtres	On peut filtrer la liste pour marquer seulement les objets insérés avant une certaine date.
Réponse	La liste des objets marqués.

PATCH	/collocations/{collocation_id}/shoppinglists/{shoppingListItem_id}	Marks as solved one item of the shopping list
Body	-	
Filtres	-	
Réponse	L'objet marqué.	

A.2.6. Tâches ménagères

POST	/collocations/{collocation_id}/houseworks	Create an housework
Body	Les données base pour une tâche ménagere.	
Filtres	-	
Réponse	La tâche crée.	

DELETE	/collocations/{collocation_id}/houseworks/{housework_id}	Delete an housework
Body	-	
Filtres	-	
Réponse	La tâche effacé.	

GET	/collocations/{collocation_id}/houseworks?dateFrom={dateFrom}&dateTo={dateTo}	Shows houseworks between dates
Body	-	
Filtres	Le filtre est obligatoire pour cet endpoint. Il sert à specifier l'intervalle de dates.	
Réponse	La liste des tâches à faire entre les dates données.	

A.2.7. Testing

GET	/test/setDatabase	Set database for testing purposes
Body	-	
Filtres	-	
Réponse	La base de données est réécrite.	

A.3. Fichier zip en annexe

A.3.1. Contenu du fichier

Le présent document est accompagné d'un fichier zip "bardelliz_bachelorthesis.zip" en annexe qui contient la version digitale de ce rapport ("rapport.pdf") et le code source de l'application web Appappa, RESTful API inclue (web-app-apartments).

A.3.2. Lancer et tester l'application web

Configuration

L'application utilise Python 2.7 et marche dans un système d'exploitation Linux. De plus il est nécessaire que Node.js et MongoDB soient installés.

Pour démarrer le serveur, il est nécessaire de se rendre dans le répertoire. La commande

```
$ npm install
```

prépare les modules nécessaires pour l'application décrits dans "package.json". Avec

```
$ sudo service mongod start
```

on démarre la base de données et avec

```
$ npm start
```

le serveur. Il est donc possible d'observer et tester la documentation Swagger à l'adresse `http://localhost:3000/api-doc` vec l'aide d'un navigateur quelconque. Le client est accessible à l'adresse `http://localhost:3000`.

Environnement de test

Pour simplifier les tests, l'application fournit une fonction qui remplace la base de données avec un environnement "préemballé". Pour préparer l'environnement, il faut utiliser l'endpoint "GET /test/setDatabase". Il est nécessaire d'être prudent avec cette fonction parce qu'elle écrase la base de données. Concrètement, il y aurait des utilisateurs, des colocations et des autres élément déjà prêts. Dans la Table A.1 sont listées les colocations et leurs utilisateurs.

Afin de réaliser ce test, il est recommandé de se logger avec les utilisateurs de la première collocation. Pour chacun des utilisateurs, l'adresse mail pour l'authentification suis le modèle "prénom.nom@mail.xyz" et le mot de passe est "string".

Colocation	Utilisateurs
Colocation 1	James Johnson Robert Roger Michael Miller
Colocation 2	Patricia Parker Linda Lee
Colocation 3	Barbara Brown
Sans colocation	William Wilson David Davis Charles Cooper Elizabeth Edwards Susan Scott Abigail Adams

Table A.1. – Comme les utilisateurs sont distribués entre les colocations dans l'environnement de test.

Bibliographie

[1] About node.js. https://nodejs.org/en/about/ (dernière consultation le Janvier 19, 2019). 16

[2] About swagger. https://swagger.io/about/ (dernière consultation le Janvier 19, 2019). 17

[3] Express - fast, unopinionated, minimalist web framework for node.js. https://expressjs.com/ (dernière consultation le Janvier 19, 2019). 17

[4] Glossary - MongoDB Manual. https://docs.mongodb.com/manual/reference/glossary/ (dernière consultation le Décembre 7, 2018). 13

[5] Welcome to the mean stack. https://mean.io (dernière consultation le Janvier 19, 2019). 12

[6] What is MongoDB ? https://www.mongodb.com/what-is-mongodb (dernière consultation le Décembre 7, 2018). 13

[7] What is REST ? https://restfulapi.net/ (dernière consultation le Janvier 13, 2019). 13

[8] Johnatan Freeman. What is an API ? Application programming interfaces explained. https://www.infoworld.com/article/3269878/apis/what-is-an-api-application-programming-interfaces-explained.html (dernière consultation le Janvier 13, 2019). 13

[9] Shantanu Kher. Again and Again ! Managing Recurring Events In a Data Model. https://www.vertabelo.com/blog/technical-articles/again-and-again-managing-recurring-events-in-a-data-model (dernière consultation le Mars 10, 2018). 16

[10] Andreas Meier. *Relationale und postrelationale Datenbanken*. eXamen.press. Springer, Berlin [u.a.], 6., überarb. und erw. edition, 2007. 14

[11] TJ VanToll. What is angular ? https://developer.telerik.com/topics/web-development/what-is-angular/ (dernière consultation le Janvier 19, 2019). 18