APGjs

Une application pour la gestion d'une société de jeunesse

TRAVAIL DE BACHELOR

ARNAUD SALVADORI

Mai 2019

Supervisé par :

Prof. Dr Jacques PASQUIER–ROCHA

et Pascal GREMAUD

Software Engineering Group

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Software Engineering Group
Department of Informatics
University of Fribourg
(Switzerland)

Résumé

La gestion d'une société de jeunesse est quelque chose de complexe. On présente, ci-joint à ce travail écrit, une application appelée "APGjs" qui a pour but de faciliter la gestion d'une société de jeunesse. Elle offre des fonctionnalités telles que la consultation des cotisations, l'ajout et la modification de membres, ou encore l'envoi de mails de rappel. L'application regroupe une base de données MongoDB, une RESTful API et un client développé avec VueJS.
Ce rapport traitera dans un premier temps d'une société de jeunesse au sens large avant de présenter en détail l'application APGjs.

Table des matières

Liste des figures

Liste des codes source

1
Introduction

1.1 Motivations et objectifs

Bien souvent, en sortant du cycle d'orientation (CO), beaucoup de jeunes s'en vont étudier ou travailler ailleurs. Bien que certaines personnes gardent contact avec des amis, d'autres oublient ou n'ont plus l'occasion de les rencontrer, perdant ainsi contact. De plus, force est de constater qu'en déménageant dans un autre endroit, il est parfois difficile de faire de nouvelles connaissances ou encore de s'habituer à de nouvelles coutumes. Il est dès lors primordial de donner la possibilité à ces jeunes de garder un ancrage dans leur village d'origine ou encore d'en faciliter l'accoutumance, ce à travers une entité référente. Une société de jeunesse permet de répondre à tous ces besoins.

D'un point de vue plus formel, une société de jeunesse se doit d'être bien gérée. Chose qui paraît simple au début, mais dès qu'elle gagne en importance, sa gestion peut devenir très complexe et demander beaucoup de temps. Ainsi, pour pallier à ce problème, celle-ci doit penser à mettre en place un système de gestion efficace et simple d'utilisation qui inspire la confiance de ses membres. De plus, une certaine automatisation des processus fonctionnels permettrait de soulager les tâches des bénévoles au service d'une société de jeunesse.

C'est dans cette optique-là que la partie pratique de ce travail de bachelor s'inscrit : créer un système de gestion fiable, efficace, pratique et automatisé au service d'une jeunesse.

1.2 Organisation du rapport

Pour commencer, ce rapport définira formellement ce qu'est une jeunesse avec l'exemple de celle de Savièse. Il tâchera ensuite d'expliquer les différentes relations entre un individu et une jeunesse avec l'utilisation de schémas entité-relations et de diagrammes relationnels. Dans le premier chapitre, le rapport mettra en évidence les besoins tant bien d'un utilisateur que d'un *admin* pour la gestion d'une société de jeunesse. Finalement, après avoir traité les aspects d'un point de vue plus global, il s'agira de recentrer ce rapport en donnant le cadre de la partie pratique.

Le troisième chapitre présente une à une les technologies utilisées dans le travail pratique. Il commencera par la base de données et l'utilisation de mongoDB. Nous verrons notamment pourquoi celle-ci a été préférée à SQL et comment elle est structurée. Ensuite, les technologies du serveur comme NodeJS et express seront passées en revue avant de terminer avec celles utilisées pour le client de l'application : vueJS et axios.

Le chapitre 4 traitera du serveur de manière plus détaillée. La structure des fichiers de l'API ainsi que mongoose, le module qui connecte une API à une base de données mongoDB, seront expliqués dans un premier temps. En clôture de ce chapitre, il sera démontré quelques éléments de programmation du serveur.

Le cinquième chapitre sera axé essentiellement sur le client APGjs. Nous passerons en revue dans un premier temps la structure des fichiers du client. Puis, dans un second temps, nous détaillerons ce qu'il est possible de faire avec le client de l'application. Pour clôturer, quelques éléments de programmation du client seront illustrés et expliqués.

Finalement, à travers une brève conclusion, un résumé du travail, une conclusion personnelle et un glossaire des éventuelles améliorations et fonctionnalités envisageables seront faits.

2

Définitions et modélisation

Nous allons d'abord essayer de comprendre ce qu'est une société de jeunesse et quels en sont ses objectifs. Ce premier point sera notamment illustré par une brève présentation de celle de Savièse. Nous tâcherons ensuite de modéliser d'un point de vue plus global les différentes relations entre une personne et une société de jeunesse. Finalement, nous mettrons en évidence les besoins en terme de fonctionnalités de gestion nécessaires au bon fonctionnement d'une société jeunesse avant de recentrer le cadre de la partie pratique à un domaine moins élargi.

2.1 Définition d'une société de jeunesse

Une société jeunesse est une association à but non lucratif qui regroupe les jeunes d'un village souhaitant s'engager pour la collectivité. Elle donne un cadre à ces derniers et les accompagne dans la vie adulte. Pour ce faire, elle organise des activités et des rencontres culturelles, festives ou encore sportives.

Outre son implication pour les jeunes, elle travaille en étroite collaboration avec la commune et l'office du tourisme en leurs rendant divers services. En contrepartie, elle se voit privilégiée dans certaines situations comme par exemple pour louer ou réserver une salle. Elle est ainsi une entité importante au sein de sa commune de résidence.

Dans le cas de la société de jeunesse de Savièse, elle est, comme son nom l'indique, la jeunesse qui représente la commune de Savièse en Valais. Fondée en 2017, elle compte à ce jour plus de 100 membres, allant de 16 à 30 ans. Il est à relever qu'elle fait partie du réseau de sociétés de jeunesses du Bas-Valais et Valais central, qui en regroupe plus de 56. Celui-ci, est illustré à

travers la Figure 1. Il s'étend du lac Léman avec la jeunesse de Saint-Gingolph jusqu'à Sierre avec celle de la Noble-Contrée.

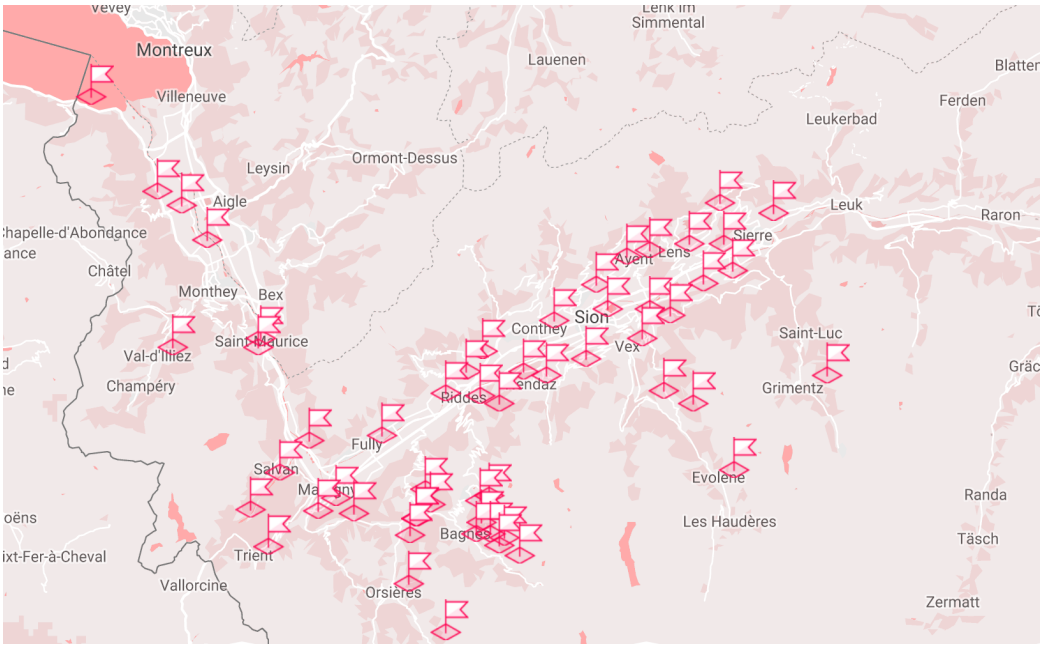

Figure 1 : Carte des jeunesses du Valais

Chaque année, une société de jeunesse valaisanne est invitée à organiser les FJVs (rassemblement des jeunesses valaisannes). Il s'agit d'un grand rassemblement où toutes les jeunesses du Valais sont amenées à s'affronter entre-elles à travers diverses joutes et activités, dans une ambiance joviale et festive.

D'un point de vue plus économique, une société de jeunesse existe, notamment à travers ses membres cotisants. En effet, chaque membre est amené à payer une cotisation annuelle dont le montant varie entre les sociétés de jeunesses. En contrepartie, celle-ci s'engage à organiser toutes sortes d'évènements et d'activités. De plus, dans l'exemple de la société de jeunesse de Savièse, une sortie est organisée chaque année pour remercier les bénévoles qui se sont engagés au moins une fois durant l'année.

Par mesure de simplification, une société de jeunesse est abrégée ci-après : jeunesse.

2.2 Modélisation d'un système de société de jeunesse

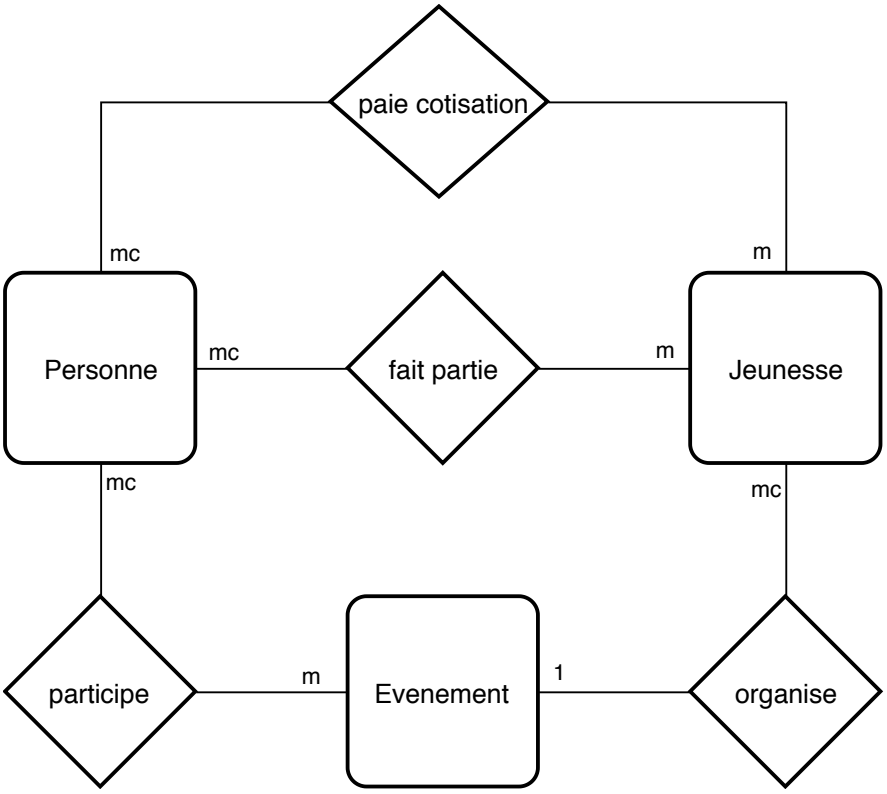

Figure 2 : Schéma entité relation d'une jeunesse

La Figure 2, élaborée à l'aide du site "draw.io", suit la notation d'Andreas Meier [1]. Celle-ci montre les différentes relations possibles entre une personne, une jeunesse et un événement :

- Une personne fait partie de 0, à 1 ou plusieurs jeunesses et une jeunesse est composée d'au moins une personne.
- Une personne paie 0, 1 ou plusieurs cotisations à une jeunesse et une jeunesse encaisse au moins une cotisation d'une personne.
- Une personne participe à 0, à 1 ou plusieurs événements et il y a au moins un participant à un événement.
- Une jeunesse organise 0, à 1 ou plusieurs événements et un événement est organisé par une seule jeunesse.

Afin de mieux comprendre ces relations, une traduction en tables relationnelles est pertinente (voir Figure 3). Nous pouvons apercevoir que les relations " *paie cotisation* ", " *participe* " et

"*fait partie*" ont été transformées en tables relationnelles. Cela est dû à leur caractère multiple-multiple.

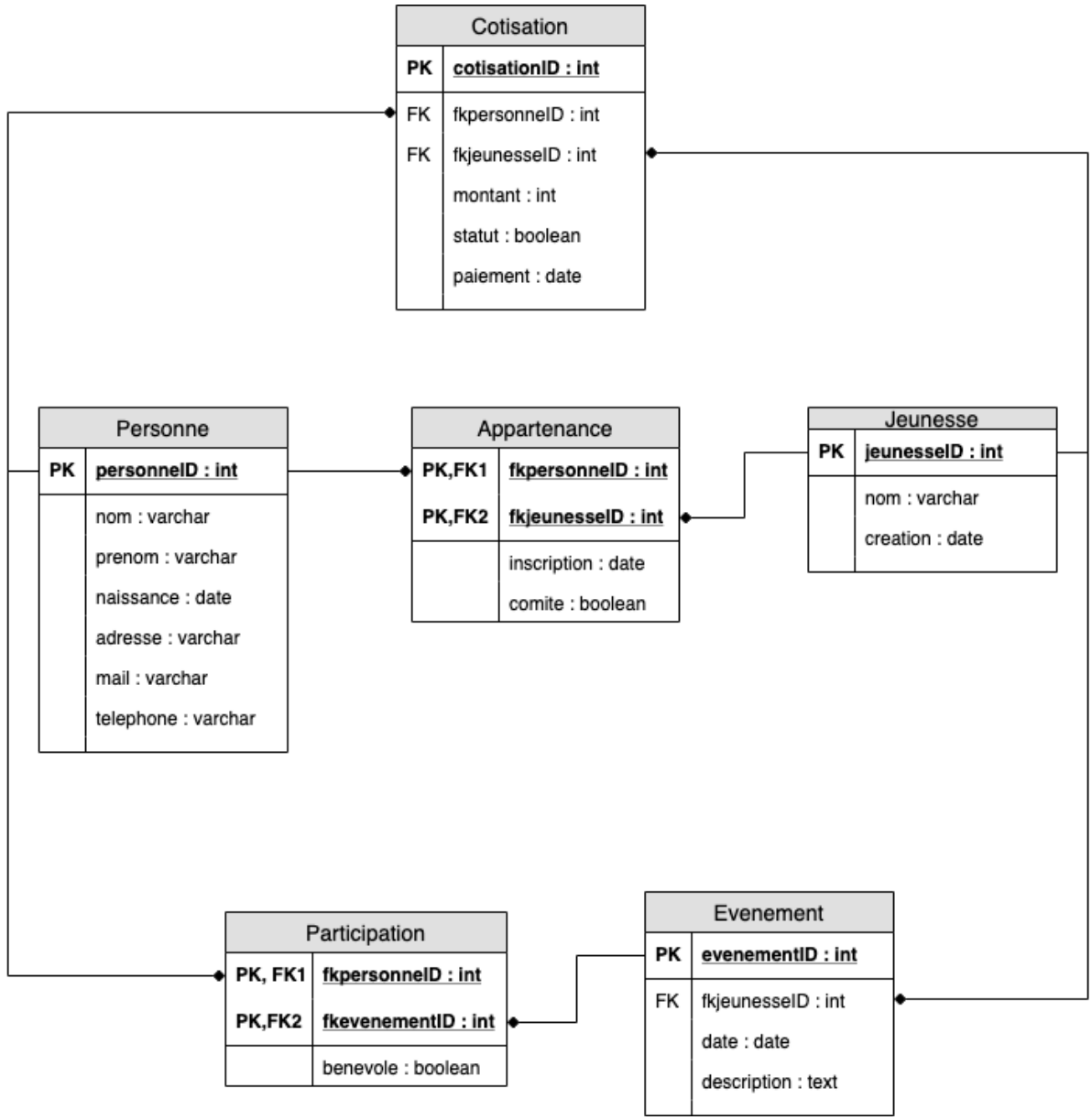

Figure 3 : Diagramme relationnel

Si nous prenons comme exemple la table « Participation », nous pouvons voir que les deux clés primaires sont toutes deux des clés étrangères : "*fkpersonneID*" et "*fkjeunesseID*". Il est dès lors nécessaire dans ce cas précis d'avoir deux clés primaires afin d'éviter la redondance, car une personne peut participer à plusieurs évènements et plusieurs personnes participent à un événement. Elle traduit donc la relation multiple – multiple qui existait dans le schéma entité

relation de la Figure 2. Par ailleurs, on ajoute l'attribut bénévole (boolean) qui indique si la personne était bénévole (= 1) ou simple participante (= 0) pendant l'événement.
Il est aussi important de préciser que la même démarche n'est pas applicable pour l'entité cotisation car, une personne paie plusieurs cotisations à la même jeunesse (une par année). Il est donc essentiel d'ajouter une clé primaire " *cotisationID* " qui garantit le caractère unique de la relation.

2.3 Cas d'utilisations

Nous allons étudier dans ce chapitre quels sont les besoins et fonctionnalités recherchés par un membre, un non-membre et ceux nécessaires à un admin (membre du comité) pour la gestion d'une jeunesse.
Ce chapitre tâchera de retranscrire les relations vues au chapitre [2.2] en fonctionnalités plus concrètes qui pourraient, quant à elles, être implémentées directement dans l'application APGjs.
Pour ce faire, des diagrammes UML de cas d'utilisation seront utilisés pour illustrer chaque système. Nous commencerons par définir les acteurs puis verrons chaque système en détails.

2.3.1 Description des acteurs

Pour faciliter la compréhension des cas d'utilisations, les acteurs suivants sont à considérer :

- Un non-membre : personne non cotisante qui ne fait pas partie de la jeunesse locale. Peut-être un simple visiteur ou membre d'une autre jeunesse qui vient s'informer sur les évènements de la jeunesse en question.
- Un membre :
    - Sans mot de passe : une personne qui fait partie de la jeunesse mais qui ne s'est pas encore enregistrée dans le système.
    - Avec mot de passe : un membre de la jeunesse ayant déjà associé un mot de passe à son compte.
- Un utilisateur non-authentifié : un acteur qui ne s'est pas logué dans le système. Il peut s'agir soit d'une personne non-membre soit d'un membre sans mot de passe ou soit d'un membre avec mot de passe mais pas logué dans le système. Ce dernier n'aura accès qu'à un nombre limité de fonctionnalités.

- Un utilisateur authentifié : membre de la jeunesse avec mot de passe, étant logué dans le système.
- Un admin : membre avec mot de passe, faisant partie du comité de la jeunesse avec pouvoirs d'administration.

2.3.2 Système d'authentification

Avant de pouvoir accéder aux fonctionnalités fournies par le site/plateforme de gestion, chaque utilisateur, qu'il soit *admin* ou simple membre, doit s'authentifier dans le système via une page de *login*. La Figure 4 montre comment fonctionnent l'enregistrement et l'authentification dans le système.

Dans le cas où la personne est un non-membre de la jeunesse, cette dernière a la possibilité de remplir un formulaire d'adhésion qui sera ensuite consulté et éventuellement confirmé par un *admin* du système.

Un membre a, quant à lui, deux possibilités : ajouter un mot de passe ou se loguer (si un mot de passe a été auparavant ajouté dans le système). Les informations à fournir sont une adresse mail et un mot de passe. En cas d'oubli de celui-ci, le membre aura la possibilité de le réinitialiser.

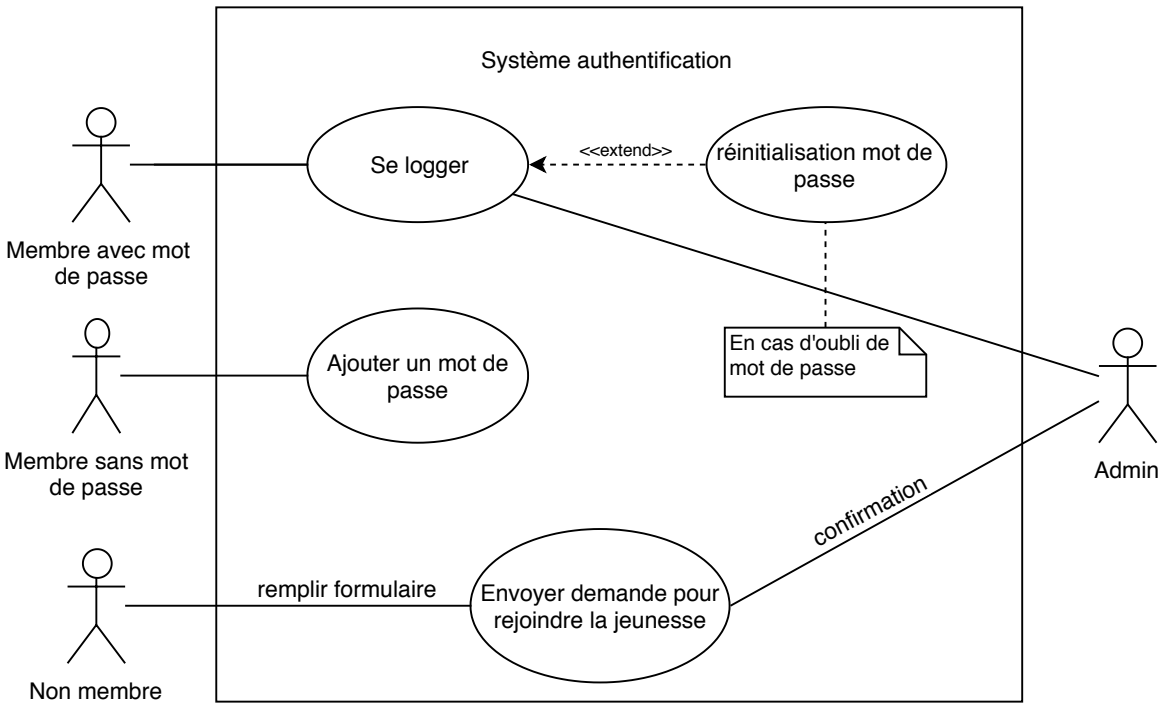

Figure 4 : Cas d'utilisation authentification

2.3.3 Système de gestion des membres

Une fois logué dans le site, le membre devient un utilisateur authentifié et accède ainsi à un nombre plus important de fonctionnalités. Afin d'éviter l'encombrement et faciliter la compréhension, le système a été séparé en trois parties : gestion des membres, gestion des événements et gestion des cotisations. Pour commencer, nous verrons celui de la gestion des membres qui est illustrée à travers la Figure 5.

Dans le système de gestion de membres, un utilisateur authentifié a accès à ses informations personnelles comme par exemple son adresse, son âge ou son mot de passe. Dans la mesure où il y aurait eu un changement quelconque ou une faute de frappe lors de la saisie du formulaire *online*, l'utilisateur peut à tout moment modifier ses coordonnées. Celui-ci peut aussi choisir de se désinscrire de la jeunesse. Un *admin* recevra alors une notification et devra confirmer sa désinscription qui conduira à sa suppression de la base de donnée.

En sus des fonctionnalités de l'utilisateur lambda, l'administrateur peut ajouter un nouveau membre, un nouvel *admin* ou encore rechercher un membre puis modifier ses informations. À l'exception du mot de passe, il a un accès total aux informations de chaque membre. Il existe bien évidemment un service de *back-up* qui pourrait être utilisé en cas de fausse manipulation ou d'erreur grave.

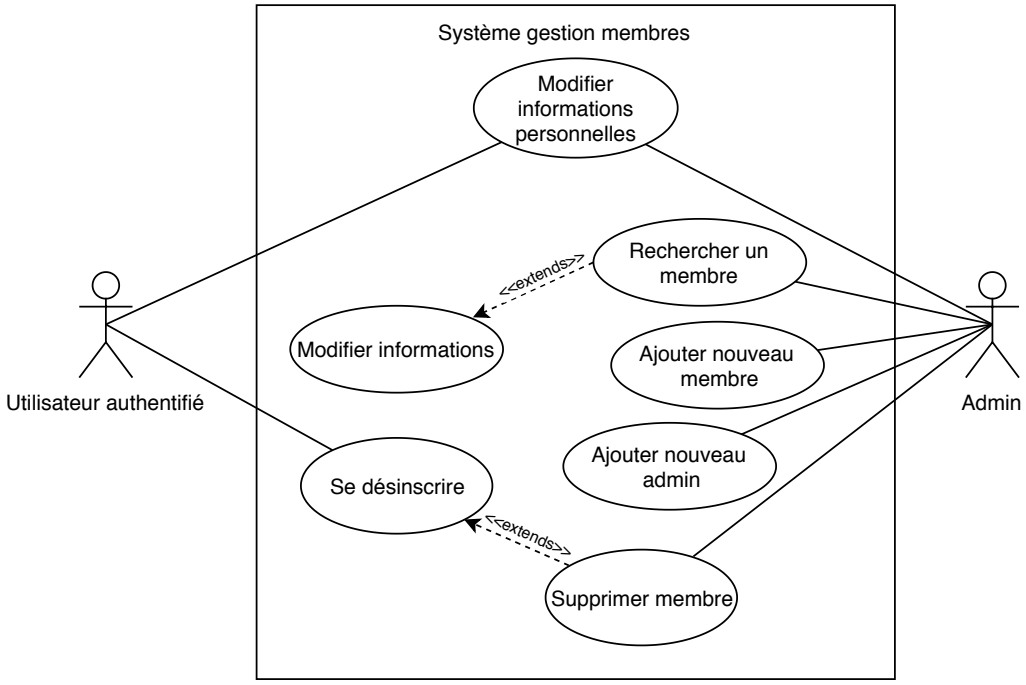

Figure 5 : Cas d'utilisation membre

2.3.4 Système de gestion des événements

La Figure 6 montre toutes les fonctionnalités concernant la gestion d'un événement. Contrairement aux autres cas d'utilisations, celui-ci implique un nouvel acteur : un utilisateur non-authentifié. Ce dernier aura uniquement la possibilité de consulter les événements organisés par la jeunesse.

L'utilisateur authentifié, quant à lui, peut consulter les évènements, s'y inscrire comme bénévole ou bien encore remplir un formulaire de proposition d'idée d'évènement. Cette idée sera ensuite consultée par un *admin* qui en jugera de sa pertinence et de sa faisabilité.

Finalement, un *admin* a la possibilité, en plus des fonctionnalités citées plus haut, de consulter la liste des tous les bénévoles inscrits aux différents événements, de créer un nouvel événement ou encore d'en modifier les informations.

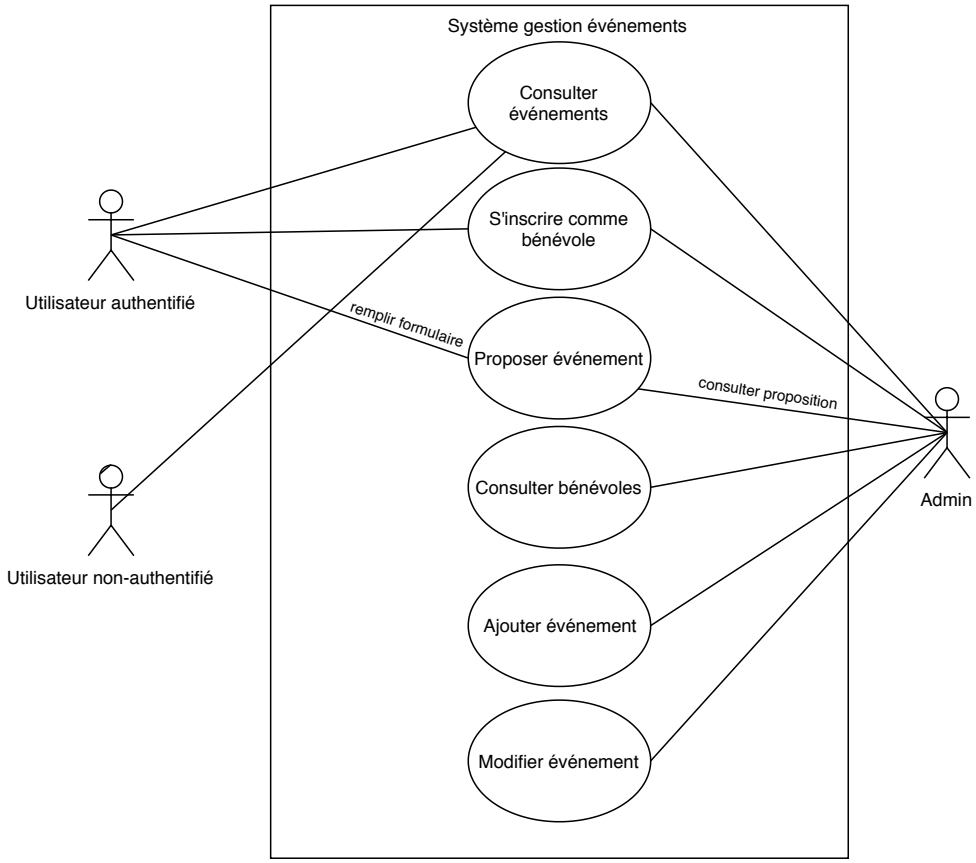

Figure 6 : Cas d'utilisation événement

2.3.5 Système de gestion des cotisations

Le système, illustré avec la Figure 7, met en lumière le système de gestion des cotisations. En effet, chaque membre est amené à payer au moins une cotisation à sa jeunesse (Figure 2), sous peine de ne pas pouvoir être admis. Ainsi, à travers le site, les utilisateurs authentifiés (membres de la jeunesse) pourront directement procéder au paiement de leurs cotisations avec possibilité de recevoir une confirmation de paiement. En outre, ils se voient aussi octroyer l'accès à l'historique de leurs paiements et aux informations de la cotisation en cours : par exemple son montant ou la date limite de paiement de la cotisation en cours.

L'*admin* peut, bien évidemment, consulter l'état de paiement de tous les membres mais aussi leurs envoyer des notifications de rappels s'ils n'ont pas encore payé leur cotisation annuelle. Dans le cas d'espèce, il est important de ne pas automatiser ce processus d'envoi de rappels car, souvent, au cours de l'année, des circonstances non prévues surviennent, ne permettant pas de fixer des délais fixes aux membres de la jeunesse. De plus, le montant de certaines cotisations peut changer au cours de l'année (frais de rappels, etc.). Naturellement, un *admin* peut aussi créer une nouvelle cotisation annuelle au moment jugé opportun et avec le montant voulu.

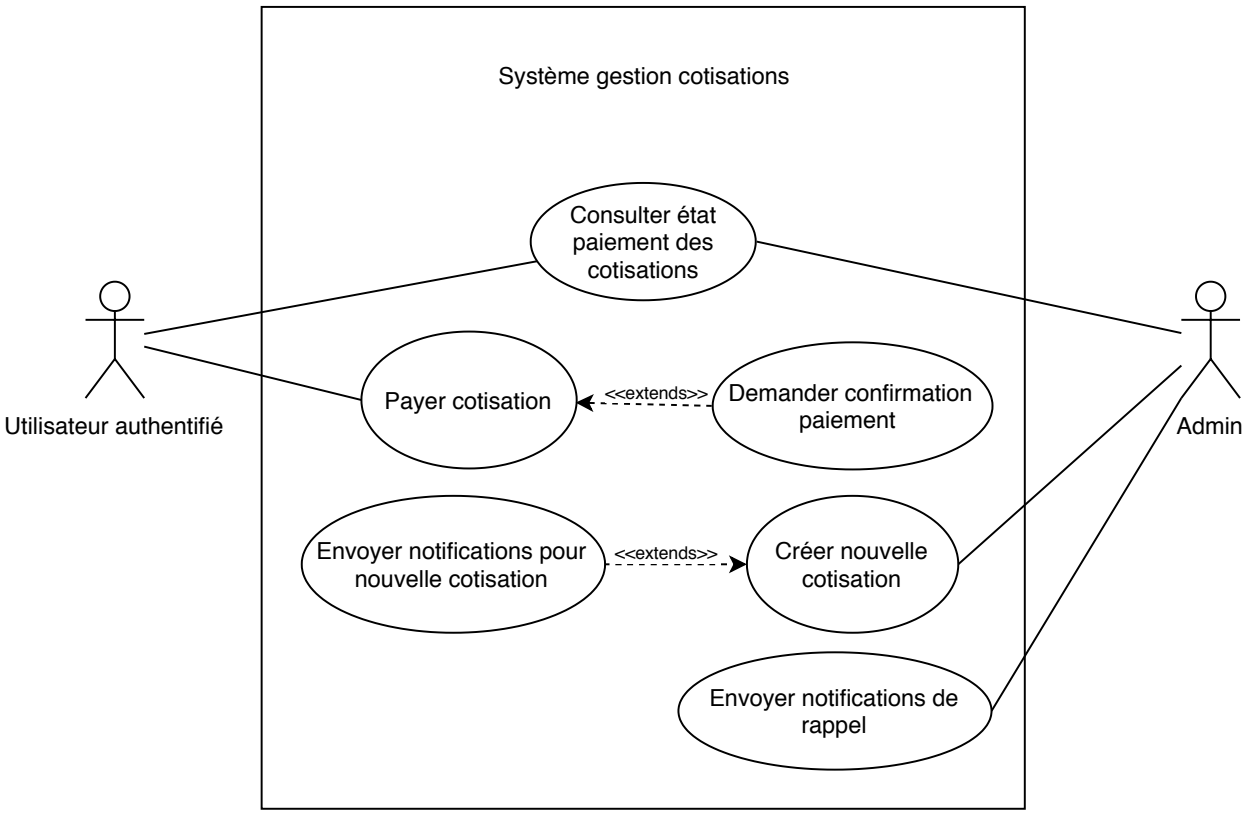

Figure 7 : Cas d'utilisation cotisation

2.4 Cadre du travail pratique

Ce chapitre avait avant tout pour but de penser le système de gestion d'une jeunesse au sens large. Cependant, dans le cas du présent du présent rapport, l'implémentation en terme de fonctionnalités se limitera à la dimension de gestion du point de vue d'un admin. En d'autres termes, le système pourra seulement être utilisé par une personne du comité (*admin*) et non pas par tous les membres de la jeunesse. De plus, toute la dimension qui traite de la gestion des évènements sera aussi mise de côté pour le moment. Le chapitre 6.3 mettra cependant en lumière toutes ces fonctionnalités manquantes qui pourraient être réalisables par la suite.

3
Technologies

Ce chapitre explicite les différentes technologies utilisées dans le cadre de la partie pratique de ce travail. Nous verrons dans un premier temps tout ce qui concerne la base de données et dans un deuxième temps les technologies utilisées au niveau du serveur et celles du côté client.

3.1 Base de données

La gestion des membres, des cotisations ou encore celle des évènements engendrent une grande quantité de données. Il est dès lors indispensable de pouvoir les stocker centralement et d'en gérer l'accès.

3.1.1 Mongodb

Le *back-end*, expliqué à travers le chapitre 3.2, utilise MongoDB [2] pour stocker de manière non-volatile les données des membres. MongoDB est une base de données NoSQL, orientée documents, complètement gratuite et *open source*. En d'autres termes, elle ne nécessite aucun schéma prédéfini des données et stocke celles-ci dans des documents au format JSON [3].
Le JSON ou la JavaScript Object Notation, est un format de données textuelles dérivé de la notation des objets du langage JavaScript. Un document JSON comprend un ensemble de paires clé/valeurs et de listes ordonnées de valeurs. Le principal avantage du JSON est son côté générique et abstrait qui permet de le représenter à travers n'importe quel langage de programmation. De plus, il est très simple à mettre en œuvre tout en gardant un niveau de complexité suffisant. [4]

De par le fait que MongoDB soit une base de données orientée documents, elle répond aux besoins suivants : flexibilité, richesse de la structure et autonomie.
Dans notre cas, MongoDB a été préféré à la base de données MySQL [5] pour une raison majeure : comme l'application est en construction, sa structure pourrait avoir à changer dans le temps. Il serait donc plus simple de la modifier à travers MongoDB de par sa flexibilité. Si nous envisageons par exemple d'ajouter la dimension d'événement à notre application, il faudrait pour cela ajouter une nouvelle entité en base de données. Chose qui peut paraître simple à première vue, mais qui peut s'avérer très complexe en fonction du choix de la base de données. Dans le cas de la base de données MySQL, il faudrait en effet créer et modifier toutes les relations existantes entre les acteurs. Tandis qu'avec mongoDB, il suffirait de créer la nouvelle ressource sans forcément avoir à modifier toutes les relations existantes (*Dynamic Schema*). [6]

3.1.2 Structure de la base de données

Afin d'avoir une vue d'ensemble de la base de données MongoDB de mon travail, le site MLab [7] a été choisi. Il permet d'une part, d'héberger en ligne les données générées automatiquement par l'application et d'autre autre part d'en donner un aperçu avec possibilité éventuelle d'en modifier les valeurs manuellement.

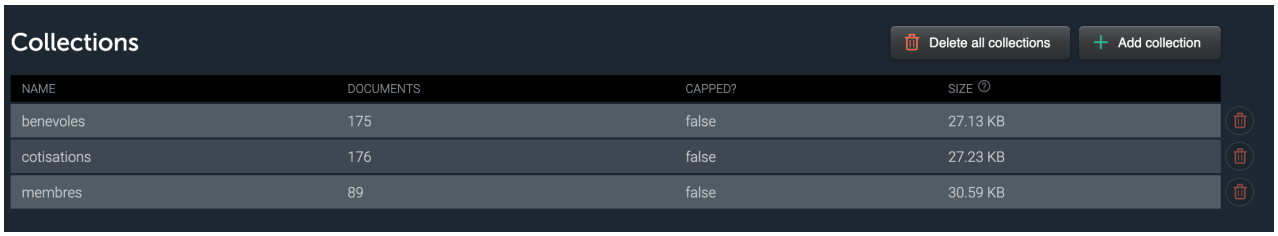

Figure 8 : Collections

Comme le montre la Figure 8, les données sont classées en 3 collections : "*bénévoles*", "*cotisations*" et "*membres*". Ces dernières regroupent un ensemble de documents. Pour la collection "*membres*", on peut voir qu'il y aura 89 documents, soit un document par membre, donc 89 membres.

```
1     {
2         "_id": {
3             "$oid": "5bf7c6debefc65651da6cf67"
4         },
5         "comite": true,
6         "nom": "Bourgeon",
7         "prenom": "Nicolas",
8         "adresse": "route de l'école 4",
9         "mail": "bourgeon.nicolas@unifr.ch",
10        "telephone": "079 304 03 91",
11        "inscription": {
12            "$date": "2017-11-23T00:00:00.000Z"
13        },
14        "naissance": {
15            "$date": "1997-08-12T00:00:00.000Z"
16        },
17        "password":
      "$2a$10$S/xwBpWs17P9lU/eG6fg6udTPrah3H.N2F6nWX50pnDRQ5PCDEK",
18        "__v": 0
19    }
```

Code 1 : Exemple de document

À travers le Code 1 au format JSON, nous pouvons voir un exemple de document de la collection membre rempli avec des données fictives. Prenons par exemple la ligne 7 du code : la clé est : "*prénom*" et la valeur associée à celle-ci : "*Nicolas*". Il est à souligner que le JSON n'utilise que du texte pour représenter les informations. Ainsi, il n'existe pas de format prédéfini pour une date.

Nous nous intéresserons notamment, à travers le chapitre 4, à étudier comment ce document a été généré automatiquement depuis le serveur.

3.2 Serveur

3.2.1 API REST

Avant de se pencher sur les technologies utilisées pour la conception du serveur, il est important d'expliquer brièvement ce qu'est une API REST. Ainsi, nous verrons ce qu'est une API et comment elle devient REST.

Une API *(Application Programming Interface*) est un ensemble normalisé de classes et de fonctions, servant de façade par laquelle un logiciel offre ses services à un autre. Son objectif est de donner accès à des fonctionnalités tout en cachant les détails d'implémentation [8]. Il est important de ne pas confondre une API avec un service *Web*. Ils sont tous deux des services de communication mais diffèrent dans le fait qu'un service *Web* facilite la communication entre deux machines sur un réseau tandis qu'une API agit comme une interface entre deux

applications différentes afin qu'elles puissent communiquer entre elles. Ainsi, tous les *Web services* sont des APIs, mais toutes les APIs ne sont pas des *Web services*. [22]
On dit d'une API qu'elle est RESTful si elle suit le standard REST, crée en 2000 par la personne de Roy Fielding. Ce dernier établit un style d'architecture logiciel à respecter pour qu'une API soit REST. Elle doit par exemple suivre le standard URI (*Uniform Resource Identifier*) pour adresser les ressources ou suivre le standard HTTP (*Hypertext Transfer Protocol*) en utilisant uniquement des requêtes de type GET, POST, PUT ou encore DELETE. De plus, une architecture REST se doit d'être *stateless*. En d'autres termes, elle ne doit pas définir un flux d'action [9].

3.2.2 Node.js et express

Après avoir brièvement expliqué ce qu'est une API REST, il convient de présenter maintenant les technologies utilisées à son développement.
Développé en 2009 par Ryan Lienhart Dahl, NodeJS [10] est un environnement d'exécution bas niveau permettant l'exécution de JavaScript côté serveur. Il est notamment utilisé comme plateforme de serveur web par des entreprises comme Yahoo, LinkedIn ou encore Microsoft. Parmi les nombreux avantages de NodeJS, citons ses excellentes performances (application en temps réel) et l'utilisation du langage JavaScript qui est le même qu'en front-end, permettant une très bonne compatibilité entre le *front-end* et le *back-end*.
Si nodeJS peut être utilisé sans aucun *framework*, nous avons quand même opté pour celui de *express* [11]. En d'autres termes, ce dernier donne un cadre à l'organisation de notre serveur et fournit d'importantes fonctionnalités simplifiant considérablement le code. Il est notamment très utile pour développer des APIs. De plus, celui-ci offre un grand nombre de modules facultatifs, mis à jour par une communauté très active, qu'il suffit d'importer au début de notre projet si nécessaire [12].

3.2.3 Endpoints

Pour une API, un *endpoint* est simplement un URL (*Uniform Resource Locator*) unique qui représente un objet ou une collection d'objets. C'est celui-ci qui nous permet de communiquer à travers un client HTTP avec le serveur. En d'autres termes, un client envoie des requêtes HTTP à des URL définis au préalable (*endpoints*). Ces requêtes sont traitées via l'API (serveur) et, dans certaines situations, retournées au client sous forme de donnée JSON. Pour qu'une API

soit REST, il est nécessaire que toutes les ressources soient accessibles de manière unique avec des URL qui leurs sont propres.

Afin de visualiser les *endpoints* de l'application, l'outil Swagger [14] a été choisi. Celui-ci est un *framework* qui offre des outils permettant de générer de la documentation pour une API ainsi qu'une interface permettant d'explorer et tester les différentes requêtes offertes par le serveur.

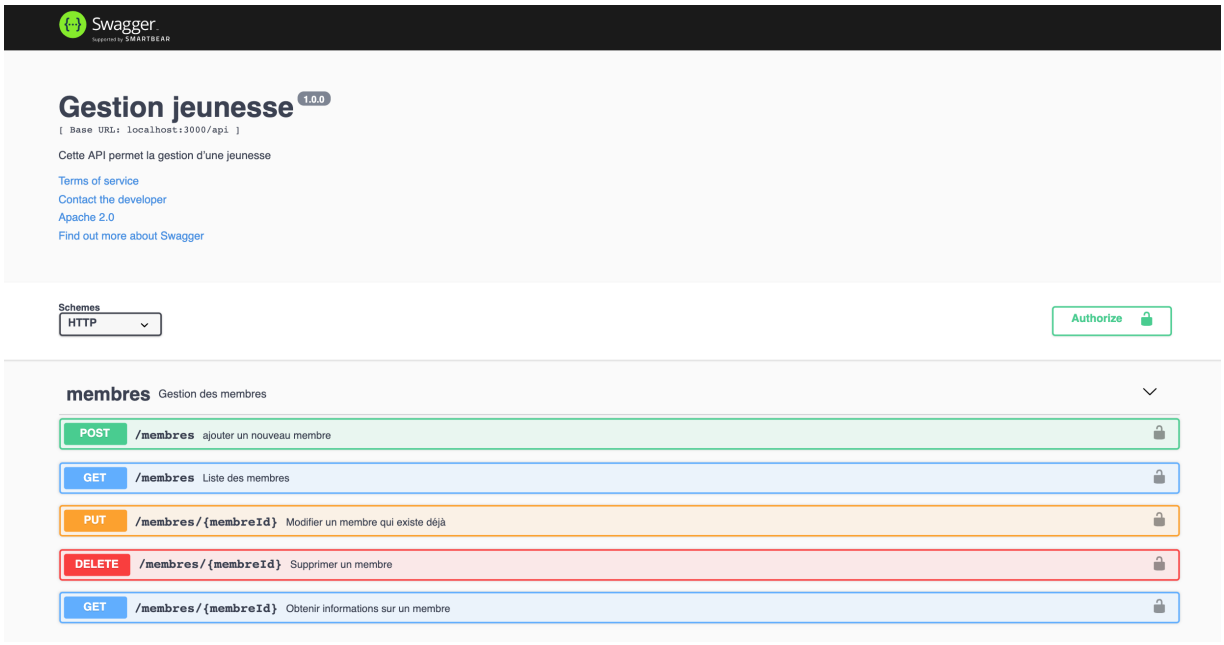

Figure 9 : Documentation Swagger

La Figure 9 illustre la collection membres avec ses différentes requêtes. Il est à souligner que chaque requête est unique : "*POST /membres*" ou "*GET /membres*" ne représentent pas la même chose. "*POST /membres*" demande pour ajouter une nouvelle ressource (*WRITE*) alors que "*GET /membres*" ne fait que de récolter des informations sans changer l'état (*READ*). Ainsi, deux requêtes peuvent avoir la même URL mais différer au niveau de la requête HTTP qui est envoyée au serveur. Prenons l'exemple du "*GET /membres*" de la Figure 9. Dans ce cas, on s'adresse à la ressource membres en lui envoyant une requête HTTP de type GET.

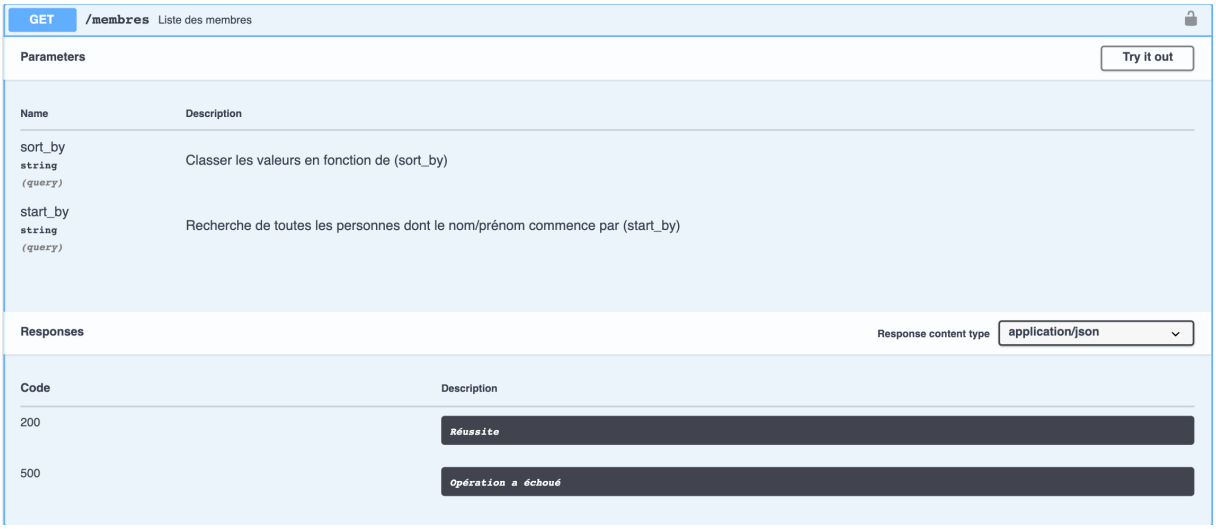

Figure 10 : Endpoint GET /membres

En examinant cette requête de manière plus détaillée (voir Figure 10), il serait important de notifier qu'une requête peut être composée de paramètres. Il existe 4 types de paramètres : les *Header paramater* (abordés plus tard), les paramètres *path*, les *query string paramaters et* les *request body parameters.* Les *query parameter* diffèrent des *path* de deux manières : ils ne sont pas des attributs directs de la ressource sur laquelle ils portent et ne sont pas obligatoires. Par exemple, "*sort_by*" et "*start_by*" ne font pas partie de la ressource membres ; ce sont des filtres dans notre cas. Si l'on voulait demander au serveur toutes les informations des membres dont le nom ou prénom commence par A et les classer par nom, la requête serait la suivante : "*/membres/?start_by=nom&sort_by=A*". Cependant, si l'on voulait récupérer les informations d'uniquement un membre, on utiliserait plutôt un paramètre de type *path* avec "*membreID*". En effet, "*membreID*" fait partie intégrante de la ressource Membre. On aurait donc un GET avec l'url suivant : "*/membres/{membreID}*". Concernant les paramètres *request body*, ils sont, la plupart du temps, utilisés dans des requêtes de type POST (lorsque l'on crée quelque chose). Contrairement aux *query* et *path parameters,* les *request body parameter* ne sont pas directement visibles à travers l'URL du *endpoint* mais, comme l'indique son nom, à travers le corps de la requête, la plupart du temps en format JSON.

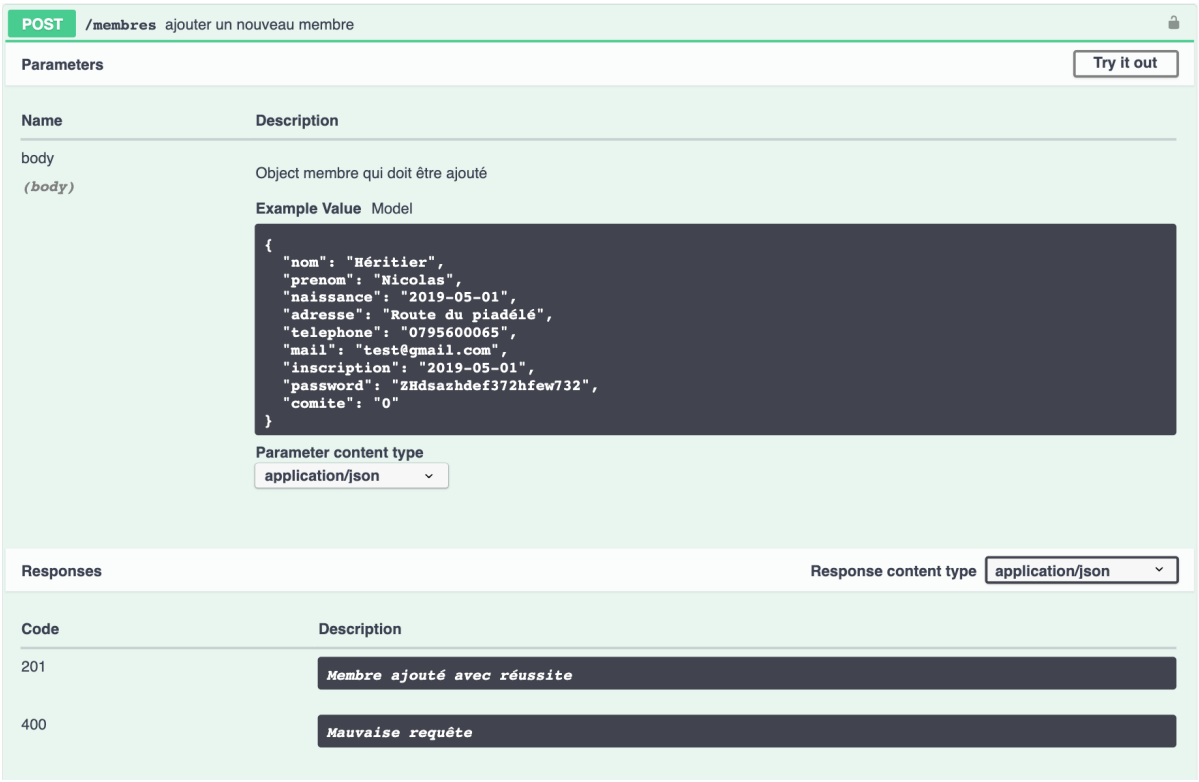

Figure 11 : Endpoint POST /membres

Par le biais de la Figure 11, on peut voir que pour chaque requête, des codes pour différents types de réponses sont prévus. Ainsi, si la réponse retournée par le serveur, a comme code 201, cela veut dire que le membre a été ajouté avec succès. Dans le cas où la requête a échoué, le code sera de 400, indiquant que le modèle de la requête n'a pas été respecté.

3.3 Client

Ce dernier sous-point de la présentation des technologies présentera VueJS [15] puis expliquera comment le client communique avec le serveur.

VueJS est un *framework* JavaScript évolutif pour construire des interfaces utilisateur. À la différence des autres *framework* monolithiques, VueJS a été conçu et pensé pour pouvoir être adopté de manière incrémentale. Par ailleurs, grâce à son intégration fluide et transparente avec d'autres outils de développement et bibliothèques, il se trouve être un excellent choix pour le développement d'applications mono-page.

Avoir un client qui fonctionne est quelque chose, mais le faire communiquer avec l'API en est une autre. Ainsi, afin d'envoyer des requêtes et d'en recevoir des réponses, nous avons utilisé le module Axios [17]. Celui-ci est une librairie qui offre des outils pour permettre la connexion et l'échange d'informations entre le client et l'API.

```
20    static postMembre(nom, pre, adr, mail, tel, naiss) {
21        return axios.post(
22          url,
23          {
24            nom: nom,
25            prenom: pre,
26            mail: mail,
27            telephone: tel,
28            adresse: adr,
29            naissance: naiss
30          },
31          {
32            headers: {
33              Authorization: "Bearer " + sessionStorage.getItem("jwt")
34            }
35          }
36        );
37      }
```

Code 2 : Exemple de requête axios

Le Code 2 montre un exemple de requête avec axios envoyée depuis le client. La méthode "*postmembre()*" utilise une fonction "*axios.post()*" qui se charge d'envoyer une requête HTTP de type POST au serveur. Pour ce faire, on indique premièrement l'URL vers lequel la requête sera envoyée. On définit ensuite le body de la requête en JSON ; dans notre cas, les informations du nouveau membre avec comme par exemple son nom, prénom ou encore son adresse mail. Finalement, afin que l'API accepte de traiter la requête, il faut encore ajouter un token de type "*Bearer*", obtenu plus tôt lors de l'enregistrement dans le système, dans le *header* de la requête.

4

Serveur

Ce chapitre traitera du serveur en détail. Nous commencerons par montrer la structure des fichiers de celui-ci avant de voir le fonctionnement de mongoose ; un module permettant de gérer l'accès à la base de données. Finalement, nous développerons quelques éléments de programmations du serveur jugés intéressants.

4.1 Structure

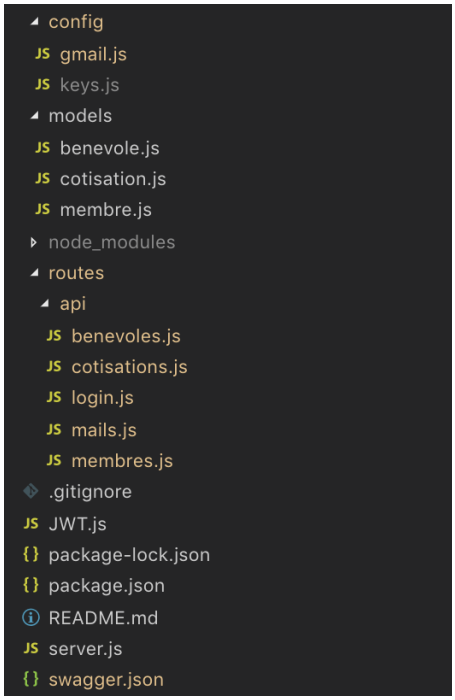

Figure 12 : Fichiers du serveur

La Figure 12 définit la structure des fichiers du serveur. Ceux-ci sont expliqués ci-après :

- Le répertoire "*config*" contient les identifiants de connexion à l'API gmail, ceux de la base de données du site mlab ainsi que le secret du token JWT. Il est donc primordial que ce répertoire ne soit pas divulgué ; il doit uniquement être accessible à des personnes de confiance, dûment autorisées.
- Le répertoire "*Models*" contient les schémas des collections utilisés par le framework mongoose. Ceux-ci sont notamment exploités lorsqu'une requête POST est envoyée à l'API et qu'une ressource doit être créée dans la base de données. Dans notre cas, seulement trois modèles sont nécessaires : membre, bénévole et cotisation.
- Le répertoire "*node_modules*" contient tous les modules téléchargés et utilisés par le serveur node.
- Le répertoire "*routes*" contient toutes les routes de l'API. C'est à travers ces dernières que les *endpoints* sont définis.
- Le fichier "*JWT*" détermine les règles de sécurité de l'API. Le sous-chapitre 4.3, traitera plus spécifiquement de la sécurité de celle-ci.
- Le fichier "*package.json*" contient toutes les informations basiques au sujet du projet, notamment le nom de l'API et son auteur.
- Le fichier "*server.js*" n'est rien d'autre que le serveur lui-même. C'est à travers ce fichier que les différentes connexions sont effectuées. Nous verrons certaines parties de ce fichier d'une manière plus détaillée à travers le sous-chapitre 4.3.
- Le fichier "*swagger.json*" est utilisé pour la documentation Swagger de l'API. Ce fichier décrit en format JSON les endpoints et modèles de l'API.

Il est à préciser que le modèle et les routes évènements n'ont pas été décrit ici car ils ne concernent pas le cadre du travail pratique. Cependant, de par le fait qu'une API se veut extrêmement modulable et flexible, on pourrait facilement imaginer de créer par la suite un nouveau modèle pour un événement dans le répertoire "*Models*" et un nouveau fichier contenant les routes pour les endpoints évènements dans le répertoire "*routes*".

4.2 Mongooose

Le *framework* mongoose [18] est indispensable si l'on utilise une base de données mongoDB et que l'on souhaite y accéder directement depuis le serveur node.js. Selon la documentation de

mongoose : ["*Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment* "] [19].

Il s'agit tout d'abord de créer une connexion mongoose avec la base de données.

```
38    const db = config.mongoURI;
39
40      mongoose
41        .connect(
42          db,
43          { useNewUrlParser: true }
44        )
45        .then(() => console.log("MongoDB connected..."))
46        .catch(err => console.log(err));
```

Code 3 : Connexion base de données

Comme montré par le Code 3, on ajoute toutes les informations concernant la base de données dans la variable "*db*". On crée ensuite la connexion avec la méthode "*mongoose.connect()"*. Cette dernière retourne une promesse. En cas d'erreur, celle-ci est écrite dans la console.

Il s'agit ensuite de définir les modèles de notre base de données avec l'interface Schéma proposé par le *framework* mongoose.

```
47     const CotisationSchema = new Schema({
48     annee: { type: Number, required: true },
49     montant: { type: Number, required: true },
50     statut: { type: Boolean },
51     membre: { type: Schema.Types.ObjectId, ref: "membre", required :
      true },
52       paiement: { type: Date }
53    });
```

Code 4 : Schéma mongoose

Le Code 4 montre le modèle pour une cotisation. Si on prend par exemple la clé "*membre*", on voit que son type est en fait une référence à un autre Schéma "*membre*" et qu'une cotisation appartient obligatoirement à un membre ("*required : true")*. À chaque fois qu'une requête de type POST veut ajouter une nouvelle cotisation, le *body* de celle-ci doit obligatoirement respecter le schéma du Code 4 sous peine de ne pas être validée.

```
54    router.put("/:id", async (req, res) => {
55      try {
56        await Cotisation.findByIdAndUpdate(
57          req.params.id,
58          req.body,
59          {new: true}
60        ).exec();
61          return res.status(200).json(data);
62      } catch (err) {
63          return res.status(400).send(err);
64      }
65    });
```

Code 5 : Exemple de route

Le Code 5 nous donne un exemple de route pour une requête HTTP de type PUT. On voit que cette dernière utilise la fonction "*Model.findByIdAndUpdate()*" pour modifier les données dans la base de données. Il suffit de lui indiquer quelle document, à travers "r*eq.params.id*", et quelles clé/valeurs, à travers "*req.body*", sont à modifier. Mongoose se charge ensuite d'établir une connexion avec Mlab et d'y modifier les données comme indiqué dans la route PUT.
Outre "*Model.findByIdAndUpdate()*", *mongoose* propose un grand nombre d'autres fonctions comme par exemple "*Model.deleteOne()*", "*Model.findOne()*", "*Modelfind()*" ou encore "*Model.replaceOne()*". En définitive, grâce au large éventail de fonctionnalités offert par le client *mongoose*, le serveur peut communiquer facilement avec la base de données.

4.3 Éléments de programmation

Ce sous-chapitre traitera dans un premier temps d'une partie du fichier server.js en expliquant comment est effectué le routage au sein du serveur. Nous verrons dans un deuxième temps le hachage des mots de passe ainsi que la vérification des hashs avec le module bcrypt et, finalement, le fonctionnement de la connexion à l'API Gmail de Google.

```
66    app.use("/api/login", login);
67    app.use("/api/membres", middleware.checkToken, membres);
68    app.use("/api/cotisations", middleware.checkToken, cotisations);
69    app.use("/api/mails", middleware.checkToken, mails);
70    app.use("/api/benevoles", middleware.checkToken, benevoles);
71    app.use("/api-docs",                                swaggerUi.serve,
      swaggerUi.setup(swaggerDocument));
72
73    const port = process.env.PORT || 3000;
74    app.listen(port,  ()  =>  console.log(`Server  started  on  port
      ${port}`));
```

Code 6 : Enregistrement des routes

Le Code 6 démontre comment est effectué le routage par le serveur. Pour chaque route (login, membres, cotisations, etc.), on utilise la fonction "*express().use()*" afin d'indiquer les endpoints possibles. Cette fonction prend tout d'abord comme argument le nom qui sera utilisé par le client pour communiquer avec l'API. Elle prend ensuite, à l'exception de la route "*login*" et "*api-docs*", une fonction "*checkToken()*". Cette dernière est une fonction qui oblige les différents clients à s'authentifier à l'API ou, en d'autres termes, à envoyer un *token* dans le *header* de la requête, avant de pouvoir communiquer avec. Nous verrons notamment, à travers le Code 7 et le Code 8, comment un *token* est créé et comment il est contrôlé lors de chaque requête. Comme énoncé plus haut, les routes "*login*" et "*api-docs*" n'ont pas de contrôle de *token*. En effet, c'est à travers la route "*login*" que le client pourra s'authentifier et recevoir un token d'accès à l'API. Cependant, concernant la route" *api-docs*", une authentification est tout de même nécessaire car celle-ci, utilisée par Swagger, permettra d'envoyer des requêtes à toutes les autres routes. Cette authentification se fera aussi par le biais de la route "*login*", mais directement par le biais de l'interface Swagger.

Ensuite, après avoir instancié le routage, on définit par le biais de la constante "*port*" sur quel port le serveur tournera. Dans notre cas, il tournera sur le port 3000 mis à part si un autre port est spécifié comme argument au lancement du serveur comme par exemple avec la commande : "PORT=4444 node server.js". Finalement, on permet à notre serveur *express* d'écouter au port notifié avec la méthode "*app.listen()*".

```
75    if (bcrypt.compareSync(req.body.password, data.password)) {
76            let   token   =   jwt.sign({   mail:   req.body.mail   },
      config.secret, {
77              expiresIn: "24h" // expires in 24 hours
78            });
79            // return the JWT token for the future API calls
80            return res.status(202).json({
81              success: true,
82              message: "authentification réussie",
83              token: token
84            });
```

Code 7 : Création d'un JWT

Bcrypt.js [20] est une librairie conçue pour nodejs qui aide à hacher les mots de passe. Elle fournit dans un premier temps une méthode "*compareSync()*" qui compare un nouveau mot de passe envoyé par le client avec le hash stocké dans la base de données. On voit notamment avec l'exemple du Code 7 qu'en cas de succès, l'API retourne un JWT *token*, créé avec la méthode "*jwt.sign()*", valide pendant 24h. Celui-ci sera ajouté à chaque nouvelle requête HTTP de ce client vers l'*API*.

```
85    const hashedPassword = bcrypt.hashSync(req.body.password, salt);
86    data.password = hashedPassword;
87    data.save().then(
88      res.status(201).json({
89         success: true,
90         message: "Mot de passe correctement enregistré"
91      })
92     );
```

Code 8 : Hashage d'un mot de passe

Dans un deuxième temps, elle donne accès à une autre méthode qui permet, quant à elle, de créer un hash pour un nouveau mot de passe. Nous pouvons voir à travers le Code 8 que cette fonction prend, en plus du nouveau mot de passe, un argument "*salt*". Celui-ci est en fait, un "code secret", définit dans un fichier confidentiel, que l'on ajoute à chaque mot de passe avant que celui soit haché afin que son *hash* soit unique. En d'autres termes, si une personne accède d'un moyen ou d'un autre à la liste des *hash* des mots de passe, il ne pourra pas essayer de comparer ceux-ci avec un dictionnaire de mot de passe car ceux stockés sont différents des mots de passe initiaux.

```
93    const transporter = nodemailer.createTransport({
94      service: "gmail",
95      auth: {
96        type: "OAuth2",
97        user: gmail.client_user,
98        clientId: gmail.client_id,
99        clientSecret: gmail.secret,
100       refreshToken: gmail.refresh_token,
101       accessToken: gmail.access_token
102     }
103   });
```

Code 9 : Connexion API gmail

Étant donné que cette API doit être en capacité de traiter l'envoi d'un nombre considérable de mails (rappels, informations, etc.), la connexion à une autre API de gestion de mails est un élément essentiel dans ce travail pratique. Nous avons préféré l'API gmail en raison de sa facilité d'implémentation. Il était néanmoins encore nécessaire de passer par l'intermédiaire d'un module nodeJS appelé nodemailer [21], pour permettre l'envoi physique de mails. Le Code 9 présente la connexion par le biais de nodemailer à l'API de gmail. On y crée un nouvel objet de type nodemailer, en lui indiquant toutes les informations nécessaires à l'authentification de l'API gmail. Ces dernières sont fournies via la plateforme oauthplayground de Google.

```
104   let mailOptions = {
105             from: "Cotisations <jeunesse.cotisation@gmail.com>",
106             subject:        `Paiement        Cotisation        ${new
      Date().getFullYear}`,
107             text: `exemple`
108           };
109           mailOptions.to = mails;
110
111           await       transporter.sendMail(mailOptions,       async
      function(err, res) {
```

Code 10 : Envoi d'un mail

L'envoi d'un mail est ensuite très simple. Si l'on se fie au Code 10, il suffit d'indiquer les options du mail comme l'expéditeur, le sujet (titre), le texte (contenu du mail) et le ou les destinataires. On invoque ensuite la fonction "*transporter.sendMail()*" avec les options définies plus haut qui se chargera d'envoyer les différents mails. Il est à relever que la valeur "*text*" du mail accepte du HTML ainsi que du CSS. Un exemple de mail envoyé aux membres de la jeunesse, pour le paiement de la cotisation de l'année 2019, est visible à travers à la Figure 13.

Salut !

Deuxième rappel : tu as jusqu'au **28 février** pour payer ta cotisation 2019.

- Si tu as déjà payé cette cotisation ou que tu n'es pas là cette année (p.exemple à l'étranger), tu peux envoyer un mail à un membre du comité et on essaiera de s'arranger.
- Si tu souhaites sortir de la jeunesse, et donc plus recevoir ces mails, tu peux aussi envoyer un message à un membre du comité.

Montant :	**50.-**
Numéro du compte :	[redacted]
Titulaire :	**Jeunesse de Savièse** **1965 Savièse**

Merci de jouer le jeu en respectant le délai fixé, on ne voudrait pas avoir besoin de courir après les gens qui ne paient pas..

Cordialement, votre comité :)

Figure 13 : Exemple de mail de rappel de cotisation

5 Client APGjs

Après un chapitre consacré au serveur, il reste encore à passer en revue le client de celui-ci. Nous commencerons par inspecter comment ce dernier est structuré à l'aide d'une analyse de la hiérarchie des fichiers. Nous continuerons ensuite avec un sous-chapitre montrant le rendu visuel du client et nous finirons avec quelques exemples intéressants de programmation.

5.1 Structure

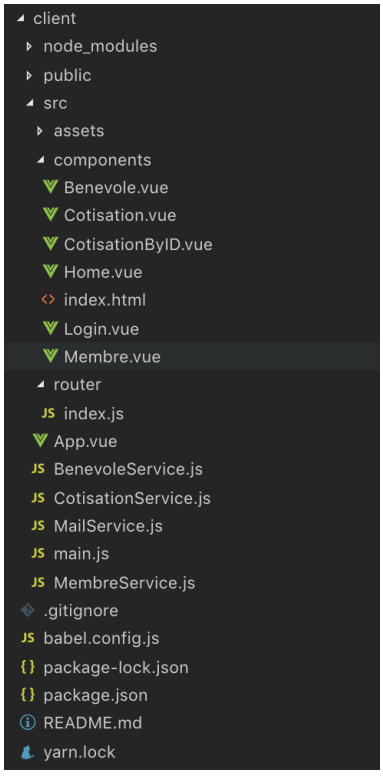

Figure 14 : Structure des fichiers du client

La structure des fichiers du client est définie dans la Figure 14. Afin de mieux comprendre de quoi il s'agit, chaque fichier/répertoire est expliqué ci-après :

- Le dossier "*components*" contient toutes les instances VueJS réutilisables de l'application. Celles-ci représentent dans notre cas, des pages entières de rendu visuel qui seront routées entre-elles pour former l'application. Il est par ailleurs facilement envisageable de créer une page composée de multiples composants emboités l'un dans l'autre.
- Le fichier "*index.js*" du répertoire router définit les règles de routage de l'interface.

```
112   {
113         path: "/login",
114         name: "login",
115         component: Login
116       },
```

Code 11 : Exemple de routage VueJS

Le Code 11 présente un exemple de routage de type vue-router, un module pouvant être importé. On associe une route à un composant et on lui choisit un nom. Dans cet exemple, on associe le composant "*Login*", définit dans le répertoire "*component* ", à la route "*/login*".

- Le fichier "*App.vue*" est le composant le plus générique de l'interface. On y définit par exemple une barre de navigation qui apparaîtra dans toutes les pages ou bien l'authentification/accès aux composants (*token*).
- Les divers fichiers "*...Service.js*" s'occupent quant à eux des relations entre l'interface de gestion et l'API. Ceux-ci définissent les requêtes nécessaires HTTP qui seront envoyées et traitées par l'API par le biais de axios (Code 2 : Exemple de requête).
- Le fichier "*main.js*" s'occupe principalement de créer une nouvelle Vue représentant l'application en général.

5.2 Rendu visuel

Nous allons à présent nous intéresser au rendu visuel de l'interface de gestion. Pour ce faire, nous passerons en revue toutes les pages du *front-end* en expliquant brièvement leurs fonctionnalités.

Commençons par la page de *login* à la Figure 15, une étape indispensable afin d'accéder au reste des fonctionnalités. L'utilisateur est amené à s'y enregistrer ou à s'y logger si son enregistrement a déjà été effectué. Force est de constater que seul un admin ou membre du comité dont la clé comité contient la valeur "*true*" dans la base de données pourra s'y enregistrer. Pour le moment, le changement de celle-ci se fait manuellement avec Mlab.

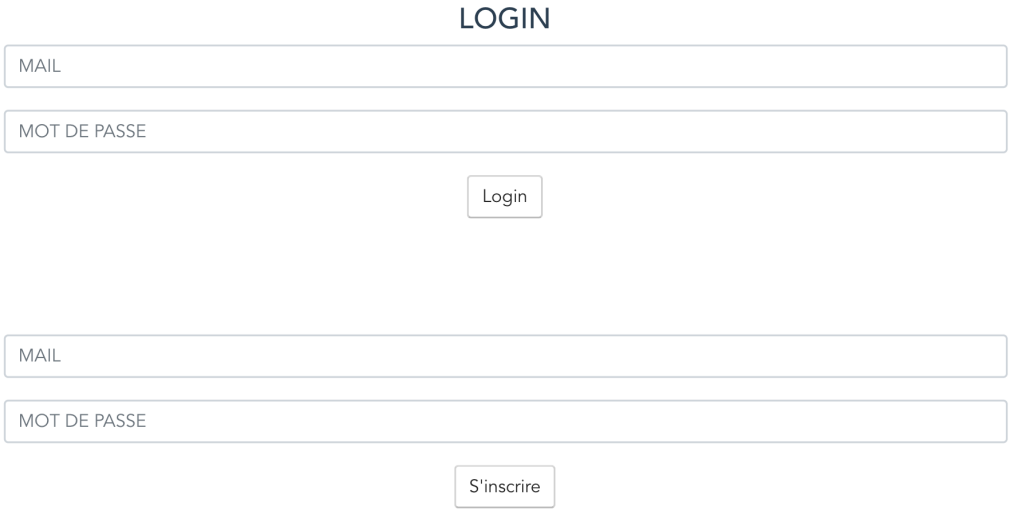

Figure 15 : Page de login

La gestion des erreurs est illustrée avec la Figure 16. Si un utilisateur oublie accidentellement de rentrer un mot de passe ou mail, il sera directement informé par un message d'erreur. Si un utilisateur entre une mauvaise adresse mail ou qu'il n'est pas membre du comité, un autre message d'erreur s'affiche : " *Ce membre n'existe pas ou n'est pas un admin* "

Figure 16 : Gestion des erreurs lors de l'enregistrement

Après avoir entré les bons identifiants sur la page de *login*, l'utilisateur recevra un *token* qui sera ensuite stocké dans son *local storage*. Celui-ci lui permettra d'accéder à toutes les pages de l'interface de gestion. La page d'accueil, visible à travers la Figure 17, s'affiche directement après l'authentification de l'admin. La personne peut alors choisir entre trois types de gestion : celle des membres, des cotisations ou des bénévoles. De plus, une barre de navigation est disponible au sommet de l'écran. Celle-ci reste visible tout au long de la navigation entre les pages mais disparaît lors du *logout* de l'utilisateur.

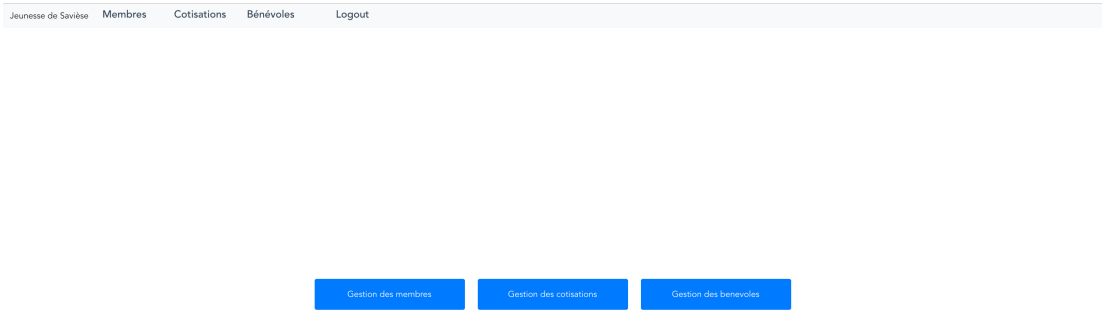

Figure 17 : Page d'accueil

Commençons d'abord avec la gestion des membres de la jeunesse. Plusieurs fonctionnalités sont disponibles sur la page de la Figure 18 : on peut tout d'abord ajouter un nouveau membre par le biais d'un formulaire. Il est ensuite possible d'effectuer une recherche par nom ou prénom. À chaque fois qu'un nouveau caractère est entré dans la barre de recherche, l'interface de gestion envoie une nouvelle requête HTTP de type GET avec un filtre : "*start_by : la valeur inscrite dans la barre de recherche* ". Le tableau est ainsi initialisé à chaque fois qu'un nouveau caractère est entré. De plus, dans l'éventualité où une valeur a été mal notée dans le formulaire d'ajout ou qu'une devait être modifiée (par exemple l'adresse d'une personne), l'admin pourrait directement changer celle-ci à travers le tableau des membres en cliquant simplement dessus. Au moment où ce dernier presse la touche *enter* pour confirmer sa modification, une requête PUT est envoyée au serveur. Le serveur traite ensuite la requête et notifie la base de données de modifier cette valeur. Pour continuer, il est encore possible de supprimer un membre en cliquant sur le bouton "*delete*", tout à droite de la Figure 18. Force est de constater qu'en supprimant un membre, toutes ses cotisations et bénévolats seront automatiquement supprimés de la base de données. Finalement, en cliquant sur "*lien*" dans la colonne Cotisations, l'*admin* est redirigé vers une autre page, celle de la Figure 19.

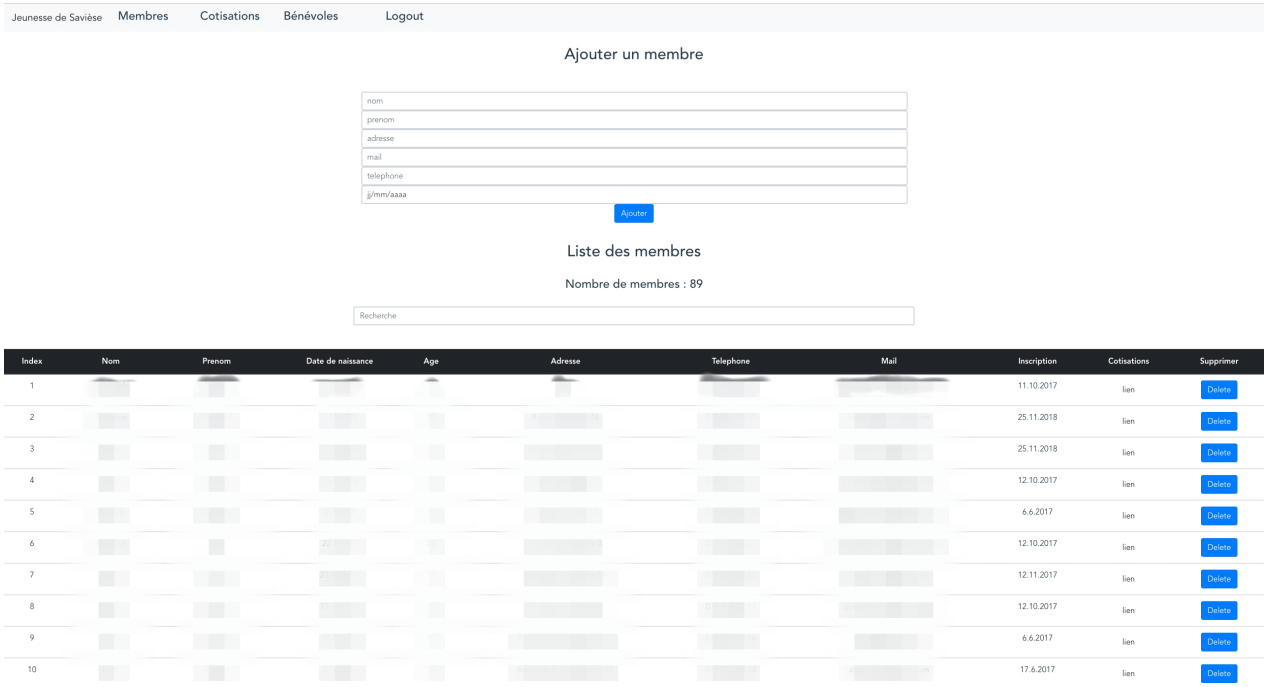

Figure 18 : Page de gestion des membres

Celle-ci montre, pour la personne en question, l'état de ses cotisations et de ses jours de bénévolat (combien de fois a-t-il été bénévole au long de l'année). De plus, l'utilisateur peut modifier directement depuis cette page l'état des paiements ou le nombre de bénévolats.

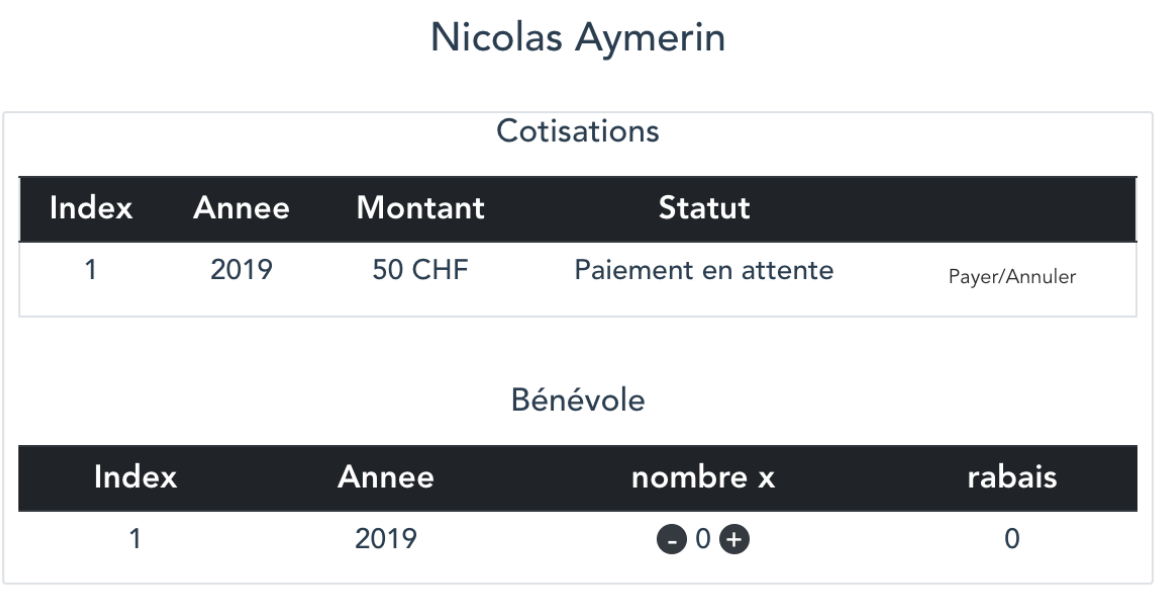

Figure 19 : Page de gestion pour un membre

En cliquant sur la rubrique cotisations dans la barre de navigation, l'utilisateur accède à une page recensant toutes les ressources cotisations. Cette page, illustrée à travers les figures 20 et 21, fournit une liste des personnes qui ont payé ou non la cotisation de l'année choisie dans le menu déroulant au sommet de la page. Dès lors, il est possible de modifier le statut de paiement "*payer/annuler*" ou encore d'envoyer des mails de rappel. Cette dernière fonctionnalité envoie une requête de type POST avec comme filtre l'année en question et le type (test ou rappel) à l'API de l'application qui se chargera d'envoyer un mail à toutes les personnes qui n'ont pas encore payé. Le *template* du mail est, quant à lui, traité au niveau du back-end (voir chapitre 4.3).

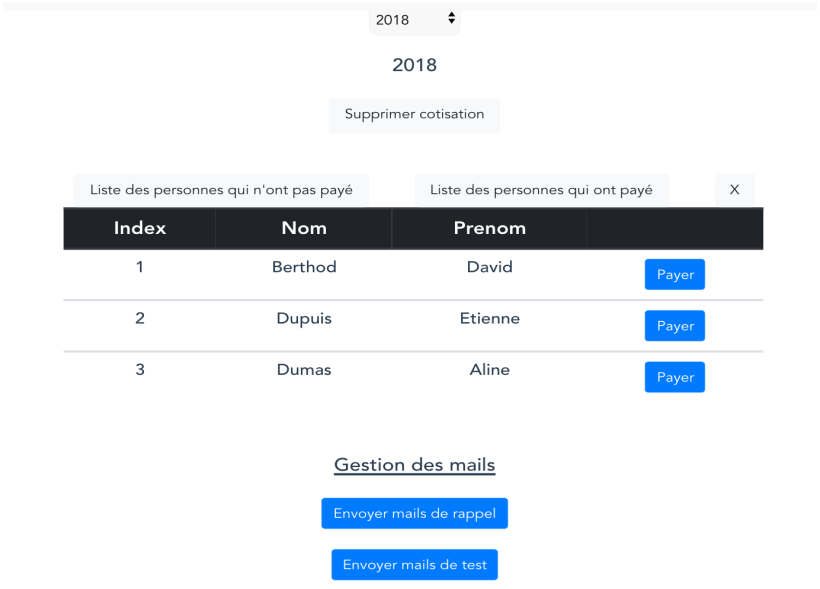

Figure 20 : Page de gestion des cotisations 1

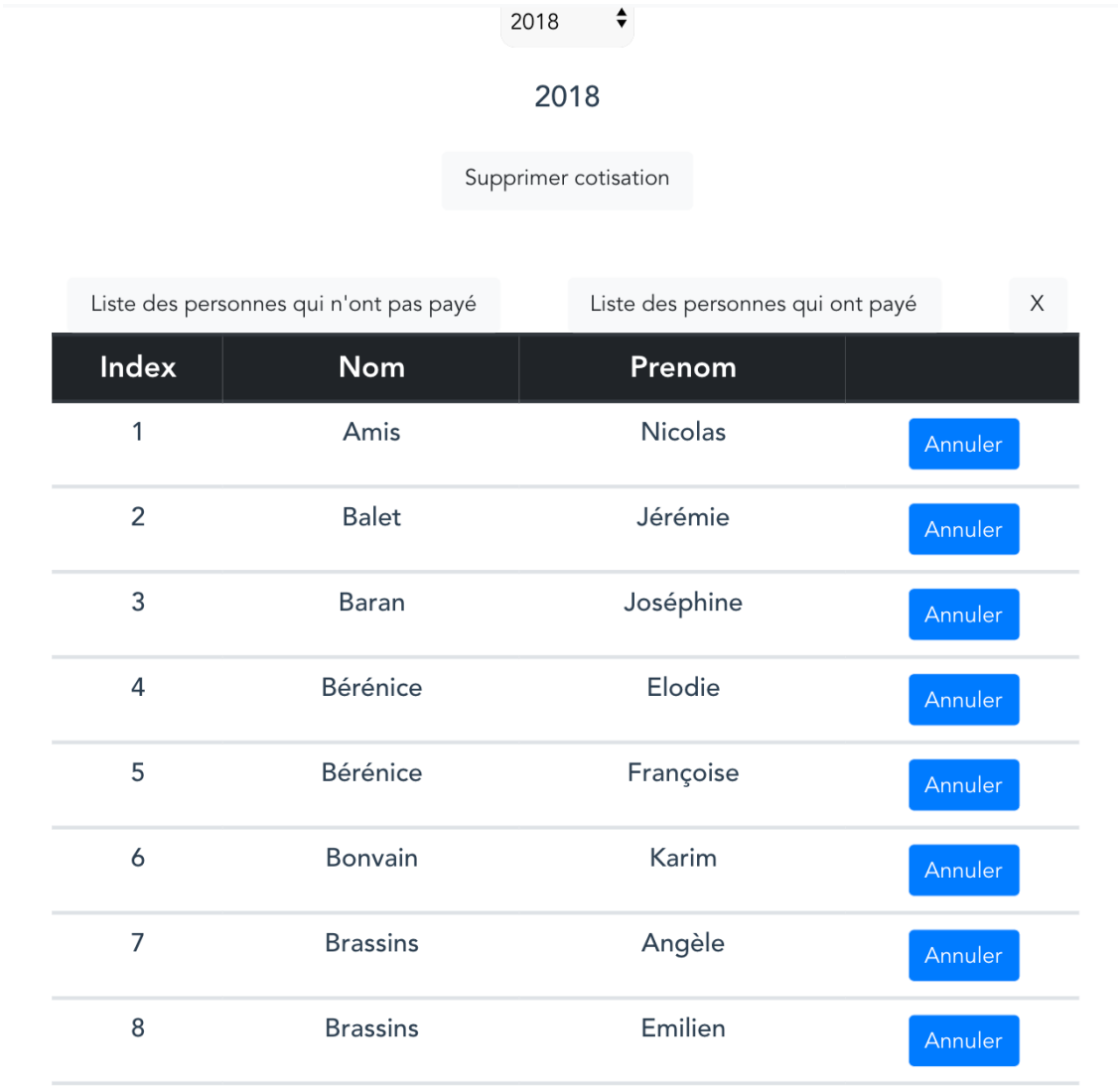

Figure 21 : Page de gestion des cotisations 2

Finalement, la gestion des bénévoles, fournit une liste structurée d'une manière analogue à celle des cotisations. Il est aussi possible d'y modifier (+ ou -) le nombre de fois que la personne fut bénévole pendant l'année choisie. À souligner que la personne obtient un rabais de 5.- par jour de bénévolat sur sa cotisation annuelle ; rabais total maximal limité à 15.- par année.

5.3 Éléments de programmation

Ce sous-chapitre tâchera d'expliquer deux méthodes intéressantes du client APGJs. Nous verrons dans un premier temps la méthode "*checkLogin()*" puis, dans un second temps, celle qui permet de créer un nouveau membre : "*createMembre()*".

Commençons donc avec la méthode "*checkLogin()*" qui s'occupe de l'authentification de l'utilisateur. Cette dernière, définie sous le Code 12, est spécifiquement invoquée par la page

de *login* (Figure 15 : Page de login) lorsqu'un admin clique sur le bouton "*login*" après avoir entré son adresse mail et son mot de passe.

Elle adresse dans un premier temps une requête HTTP de type POST à l'API avec comme *body* le mail et mot de passe de l'utilisateur. Le cas échéant, si la promesse n'est pas respectée, la méthode retourne le message d'erreur dans la réponse de l'API. En cas de respect de la promesse, le *token* JWT de la réponse est stocké dans le *sessionStorage* de l'utilisateur. Ce dernier sera ajouté dans le *headers* de chaque nouvelle requête à destination de l'API et permettra de l'autoriser à être traitée par l'API. De plus, le *boolean* "*authenticated*" est mis à la valeur "*true*". Celui-ci est contrôlé par le routeur VueJS et doit indispensablement être à "*true*" afin de pouvoir accéder aux différentes pages de l'interface. Finalement, la méthode "*replace()*" du routeur est invoquée, remplaçant la page actuelle par la page "*home*".

```
117   checkLogin() {
118         const type = "login";
119         axios
120           .post(
121             `${url}?type=${type}`,
122             {
123               mail: this.login.mail,
124               password: this.login.password
125             },
126             { validateStatus: false }
127           )
128           .then(res => {
129             if (res.data.success == false) {
130               return (this.login.err = res.data.message);
131             }
132             sessionStorage.setItem("jwt", res.data.token);
133             sessionStorage.setItem("authenticated", true);
134             this.$emit("authenticated", true);
135             this.$router.replace({ name: "home" });
136           });
137       }
```

Code 12 : Méthode checkLogin

"*CreateMembre()*" est, comme l'indique son nom, une méthode qui permet d'ajouter un nouveau membre à la base de données de la jeunesse. Celle-ci est définie à travers le Code 13. Elle commence donc par appeler la fonction "*postMembre()*" définie dans le fichier "*MembreService*" qui s'occupera d'envoyer une requête HTTP de type POST avec comme *body* toutes les informations du nouveau membre. À noter que ces informations sont récoltées au moyen du formulaire de la page de gestion des membres (voir Figure 18 : Page de gestion des membres). Dans un second temps, si le membre a bien été ajouté, une nouvelle cotisation pour l'année en cours est générée pour le nouveau membre par le biais de la méthode

"*postOneCotisation()*". De manière analogue, une nouvelle ressource bénévole est aussi créée pour celui-ci. Finalement, les informations sur les membres stockées dans l'application sont actualisés à l'aide de la méthode "*getMembres()*".

```
138   async createMembre() {
139         await MembreService.postMembre(
140           this.membre.nom,
141           this.membre.pre,
142           this.membre.adr,
143           this.membre.mail,
144           this.membre.tel,
145           this.membre.naiss
146         );
147         await CotisationService.postOneCotisation(
148           this.membre.nom,
149           this.membre.pre,
150           50
151         );
152         await      BenevoleService.postOneBenevole(this.membre.nom,
      this.membre.pre);
153
154         this.getMembres();
```

Code 13 : Méthode createMembre

6
Conclusion

6.1 Synthèse

En définitive, deux grandes parties ont composé ce travail de bachelor : une partie pratique et une partie théorique.

Cette première consistait en l'élaboration d'une application de gestion pour une jeunesse. Il fallait tout d'abord déterminer le cadre général de l'application : acteurs impliqués, fonctionnalités nécessaires, etc. Cette première étape nous a permis d'élaborer une base de données concrète dans laquelle les différentes informations des utilisateurs seraient stockées. Nous avons ensuite construit un serveur NodeJS sous forme d'API REST, répondant aux besoins en terme de fonctionnalités pour la gestion d'une jeunesse, avant d'implémenter une interface vueJS dont les fonctionnalités sont puisées depuis l'API.

La deuxième grande partie fut consacrée à l'écriture d'un rapport expliquant l'application. Celui-décrit d'abord ce qu'est une jeunesse et comment la modéliser de manière général. Nous avons ensuite recentré le rapport au cadre du travail pratique et vu, une à une, les technologies utilisées. Deux chapitres, respectivement un sur le serveur puis un sur le client, présentaient ensuite ceux-là en détails en donnant quelques exemples de programmations ou encore de rendus visuels.

6.2 Bilan personnel

N'étant pas spécialiste dans le domaine de l'informatique et de la programmation, une grande partie du travail a été composée de l'apprentissage des technologies. En effet, mis à part quelques bases en JavaScript, tous les autres langages comme NodeJS, mongoDB ou vueJS m'étaient inconnus. Bien que l'apprentissage ne fut pas évident au début, j'ai acquis rapidement un grand nombre de notions et découvert de nouvelles technologies qui me seront sans doutes fort utiles par la suite.

De plus, j'ai beaucoup apprécié le côté pratique de ce travail, plus précisément le fait de pouvoir concevoir une application de "A à Z", de la voir s'améliorer de jours en jours au gré des nouvelles fonctionnalités.

Finalement, un dernier point essentiel est celui de l'utilité de l'application. Je l'utilise en effet toutes les semaines lorsqu'il s'agit d'ajouter des nouvelles personnes ou d'envoyer des mails de rappel. Tout est désormais situé au même endroit, m'évitant la perte de feuilles ou de fichiers. De plus, l'application simplifie énormément tout le travail de gestion : les lettres écrites à la main ont par exemple été remplacées par des mails automatiques.

6.3 Améliorations possibles

Concernant les améliorations possibles de l'application, elles sont au nombre de trois. Premièrement, il s'agirait d'ajouter toute la dimension des évènements. Ainsi, un administrateur pourrait créer de nouveaux événements à une date précise avec leur propre description.

Une deuxième amélioration possible serait celle de remplacer l'interface de gestion par un site web. C'est à dire, que ce dernier ne serait plus seulement accessible aux administrateurs (membres du comité) mais aussi à tous les membres et non membres de la jeunesse. Les non-membres pourraient alors consulter les prochains évènements organisés par la jeunesse ou tout simplement se renseigner sur elle, tandis que les membres auraient accès à un espace personnel par le biais d'une page de *login*. Dès lors, ils pourraient consulter l'état de leurs paiements, proposer de nouveaux évènements ou encore procéder au paiement de la cotisation en cours.

Finalement et afin d'améliorer la portabilité de l'application, respectivement le site, une application Android/iPhone serait d'une grande utilité. En effet, ces dernières sont d'actualité et connaissent un succès important auprès des jeunes. De plus, on augmenterait la taille de l'éventail des fonctionnalités possibles.

Bibliographie

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

[2] The most popular database for modern apps | MongoDB. https://www.mongodb.com (accédé le 27 mars 2019)

[3] Introducing JSON. https://json.org/ (accédé le 27 mars 2019)

[4] The JSON DATA. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (accédé le 27 mars 2019)

[5] MySQL. https://www.mysql.com/ (accédé le 26 mars 2019)

[6] MySQL vs MongoDB. https://medium.com/xplenty-blog/the-sql-vs-nosql-difference-mysql-vs-mongodb-32c9980e67b2 (accédé le 28 mars 2019)

[7] MongoDB Hosting: Database-as-a-Service by mLab. https://mlab.com (accédé le 12 mars 2019)

[8] Utilisez des API REST dans vos projets web. https://openclassrooms.com/fr/courses/3449001-utilisez-des-api-rest-dans-vos-projets-web/3501901-pourquoi-rest (accédé le 08 mars 2019)

[9] Les API REST. http://blog.pilotsystems.net/2012/septembre/les-api-rest (accédé le 06 avril 2019)

[10] Nodejs. https://nodejs.org/en/ (accédé le 02 mars 2019)

[11] Express - Node.js web application framework expressjs.com (accédé le 03 mars 2019)

[12] Introduction à Express NodeJs. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction (accédé le 05 avril 2019)

[13] Documentating APIs : a guide for technical writers. https://idratherbewriting.com/learnapidoc/docapis_doc_parameters.html (accédé le 08 avril 2019)

[14] The Best APIs are Built with Swagger Tools | Swagger. https://swagger.io (accédé le 10 avril 2019)

[15] Introduction à Vue.js. https://fr.vuejs.org/v2/guide/index.html (accédé le 12 mars 2019)

[16] Pourquoi avoir choisi Vue.js. https://medium.com/@bluecoders/interview-cto-i-pourquoi-avoir-choisi-vuejs-9ef0b03dd1a5 (accédé le 18 mars 2019)

[17] Promise based HTTP client for the browser and node.js. https://github.com/axios/axios (accédé le 10 avril 2019)

[18] Mongoose ODM v5.5.1. https://mongoosejs.com (accédé le 11 avril 2019)

[19] GitHub - Automattic/mongoose. https://github.com/Automattic/mongoose (accédé le 05 avril 2019)

[20] bcrypt-nodejs - npm. https://www.npmjs.com/package/bcrypt-nodejs (accédé le 25 février 2019)

[21] Node mailer : module for nodejs to send mails. https://nodemailer.com (accédé le 10 mars 2019)

[22] Différence entre api et web service. https://waytolearnx.com/2018/11/difference-entre-api-et-web-service.html (accédé le 01 mai 2019)