Application client-serveur servant à la gestion de cours extra-scolaires intercantonaux

Travail de Bachelor

Sophie Crevoiserat
Septembre 2021

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)

Table des matières

Liste des figures

Liste des codes sources

1
Introduction

1.1. Motivations et objectifs

L'offre de cours extra-scolaires est vaste. L'organisation de ceux-ci peut énormément varier suivant les prestataires et cela se fait parfois de façon très informelle et déstructurée. Il est d'autant plus compliqué de s'organiser lorsqu'un cours est donné à grande échelle, comme par exemple à un niveau intercantonal. Harmoniser le tout permet donc un gain d'efficacité et de temps, puisque toute l'information nécessaire est regroupée au même endroit. En ce sens, un service permettant de gérer l'organisation de ce type de cours est utile et permet de garder un cadre bien ordonné. Ce travail va donc montrer la mise en oeuvre d'une application client-serveur de gestion de cours extra-scolaires communs à plusieurs cantons suisses.

L'objectif principal est donc de créer et de comprendre comment fonctionne un service client-serveur basé sur une architecture REST. Pour cela, il faut :

1. Définir les besoins nécessaires au service
2. Créer un serveur permettant de répondre aux différentes requêtes
3. Créer un client permettant aux utilisateurs d'interagir avec le service

1.2. Structure du rapport

Chapitre 1 : Introduction

L'introduction commence par présenter les motivations qui poussent à réaliser ce travail et les objectifs à atteindre pour y arriver. Ensuite, elle décrit la structure du rapport, c'est-à-dire un bref résumé de chaque chapitre qui le compose. Finalement, quelques mots sur les différentes conventions utilisées pour la rédaction.

Chapitre 2 : Contexte

Ce chapitre expose ce que sont les cours extra-scolaires et pose les bases de leur fonctionnement. Le tout est modélisé pour une vision d'ensemble. Il détaille également les différentes fonctionnalités souhaitées pour le service. Des use cases et schémas UML sont utilisés pour faciliter la compréhension de ce chapitre.

Chapitre 3 : Présentation du point de vue utilisateur final

La première partie de ce chapitre est consacrée à la présentation du client du point de vue de l'utilisateur final. En d'autres termes, comment est construit l'interface et comment l'utiliser. La deuxième partie explique le déroulement de quelques scénarios possibles avec ce client. A savoir que ce qui concerne la programmation du client à proprement parler sera présenté dans le chapitre 5. Ici, il est plutôt question du guide d'utilisation.

Chapitre 4 : Programmation du serveur

Ce chapitre expose l'idée générale du code concernant le serveur et des différentes technologies utilisées pour le faire. Les principales bases de l'architecture REST y sont également détaillées avec notamment les différents endpoints de l'API Swagger UI.

Chapitre 5 : Programmation du client

Comme évoqué précédemment, ce chapitre développe ce qui concerne la partie programmation relative au client ainsi que les différentes technologies utilisées pour le faire.

Chapitre 6 : Conclusion

Finalement, une conclusion est faite en commentant les résultats obtenus et en commentant l'atteinte ou non des objectifs. Plusieurs améliorations possibles sont également exposées afin de faire évoluer l'application pour un potentiel déploiement et quelques mots personnels terminent le travail.

1.3. Conventions

— Le travail est divisé en 6 chapitres. Chaque chapitre est divisé en plusieurs sections et, pour le chapitre 3, encore en sous-sections.

— Les deux genres, féminin et masculin, sont égaux, mais pour une raison de simplicité le genre masculin est systématiquement utilisé.

— Les figures et codes sources (listings) sont numérotés séparément par ordre d'apparition.

— Les codes sources sont formatés de la sorte :

```
division(x, y) {
    try{
        result = x / y;
        return result;
    }catch(err){
        return false;
    }
}
```

2
Contexte

2.1. Principes des cours extra-scolaires intercantonaux

Tout d'abord, un cours extra-scolaire est un cours qui sort du cadre classique de l'enseignement. Cela peut être un cours de danse, d'informatique, de langue, et bien d'autres encore. Une inscription à un cours n'empêche pas l'inscription à un autre cours, du temps qu'ils n'aient pas lieu au même moment. Généralement, pour éviter cette situation, un type de cours a lieu chaque semaine au même jour et à la même heure. Les cours extra-scolaires se situent également à différents niveaux d'âge, variant des tout petits entrant à l'école enfantine jusqu'à des étudiants universitaires. Cela permet d'avoir des cours adaptés que ce soit par rapport à la difficulté ou au sujet traité.

Ensuite, le déroulement et l'organisation des cours dépendent de qui s'occupe de cela. Un particulier, une école ou une institution spécialisée ne travailleront pas de la même façon. Plusieurs écoles peuvent également travailler ensemble à un niveau régional ou intercantonal. C'est ce dernier cas qui est développé pour ce travail.

Dans chaque canton qui participe, généralement un seul établissement est mis à disposition pour les multiples cours qui parleront d'informatique, mais il y a la possibilité d'en avoir plusieurs. Les différents élèves souhaitant participer à ces cours sont répartis dans les classes du canton dans lequel ils vont à l'école habituellement. Un cours extra-scolaire n'est pas équivalent à une leçon d'école, il dure généralement plus longtemps et puisque les cours ont lieu régulièrement, il y a pour chaque établissement une classe associée au matin et une classe associée à l'après-midi. Évidemment, si un canton a peu d'enfants inscrits pour les cours, un seul bâtiment est mis à disposition et il peut aussi n'y avoir qu'une seule classe, soit le matin, soit l'après-midi.

L'application, qui sert donc à la gestion de ces cours, est destinée uniquement aux chefs et assistants, puisque les enfants n'ont pas besoin d'avoir accès à d'autres informations que les dates et lieu des cours ce qui peut très bien être communiqué au préalable. Dans l'application par exemple, les différents cours y sont mis à disposition pour que

tous les assistants qui enseignent puissent les consulter. Les assistants, tout comme les enfants, sont affiliés à un canton et travailleront à chaque fois pour le même canton. Ils ne peuvent pas donner cours dans deux cantons différents, mais s'ils le souhaitent, ils peuvent faire un changement de canton de travail. Cela peut par exemple être utile en cas de déménagement.

2.2. Modélisation

Il est donc question de cours d'informatique donnés par des assistants à des enfants dans plusieurs cantons suisses. La Figure 1 modélise les explications qui suivent.

Tout d'abord, il y a des personnes qui sont soit un assistant, soit un chef, soit un enfant. Il ne peut pas y avoir d'autre type de personnes et chaque personne occupe un unique rôle. Un chef est une personne qui ne donne pas et qui ne suit pas de cours, mais qui supervise l'ensemble. Il a le rôle d'admin dans l'application. Un assistant est une personne qui donne des cours et qui travaille pour un seul canton. Il a le rôle de simple utilisateur dans l'application. Un enfant est une personne qui appartient à une seule classe d'un seul canton. Les cours suivis se font en fonction de la classe et il n'a pas accès à l'application.

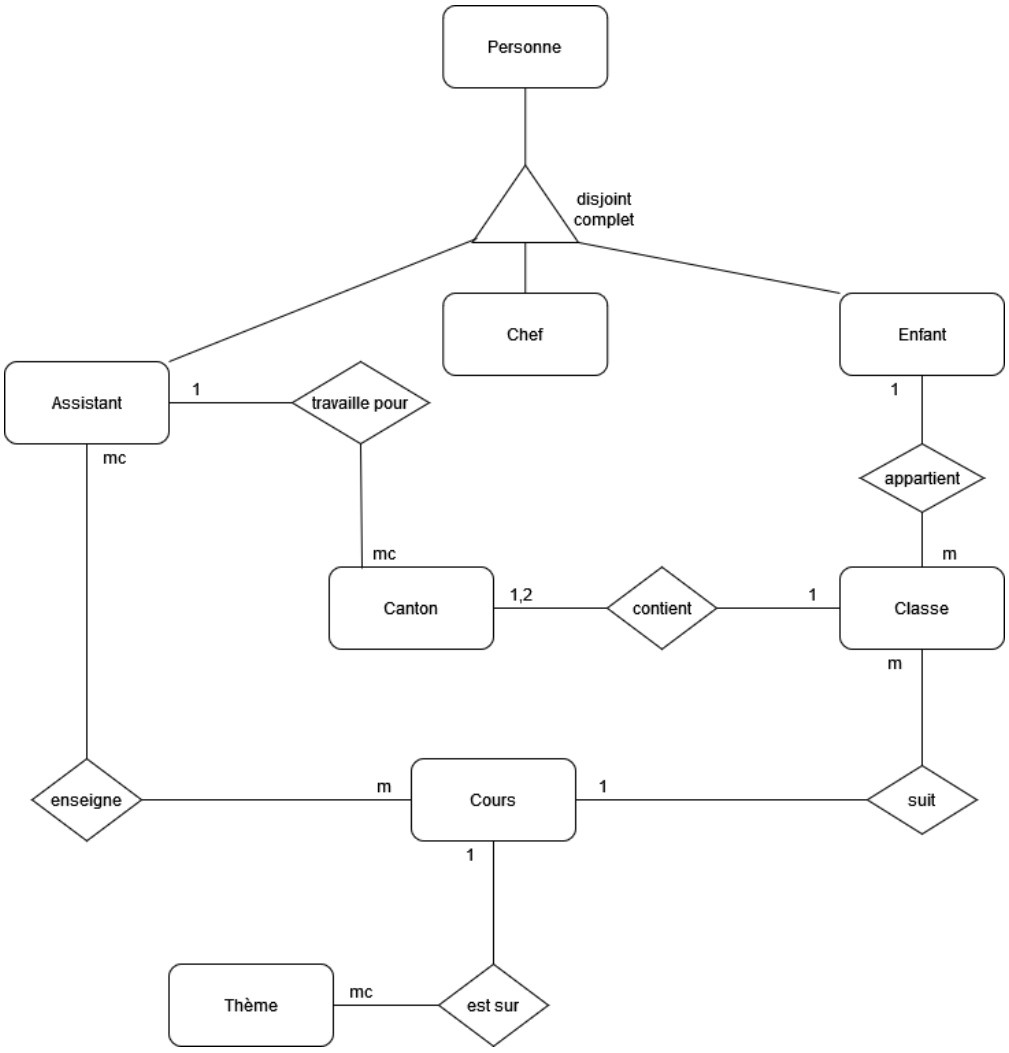

Figure 1. – Schéma entité-relation des cours extra-scolaires intercantonaux

Ensuite, une classe va suivre plusieurs cours et est affiliée à un canton. Étant donné que chaque établissement par canton possède une ou deux classes, une le matin et/ou une l'après-midi, les cours que suivent les classes en question sont donnés par les assistants associés à ce même canton. En d'autres termes, un étudiant travaillant pour le canton de Fribourg donne uniquement des cours à des classes du canton de Fribourg. Il ne peut pas donner de cours pour une classe du canton du Jura par exemple.

Finalement, un cours traite d'un thème. Les différents thèmes sont traités dans plusieurs cours pour pouvoir être donnés à de multiples classes dans les différents cantons.

La Figure 2 décrit un exemple fictif pour aider à la compréhension. Pour que cela reste lisible, une simplification est faite et l'entièreté des liens n'est pas représentée. Il y est montré le chef, Jean Simon, qui possède une vue d'ensemble sur les participants et les différents cours et classes. Il y a également la liste des enfants inscrits dans le Jura. Par conséquent, ces enfants sont répartis dans les deux classes du canton. L'assistant Yves Dupont étant affilié au même canton, il peut donner cours aux deux classes.

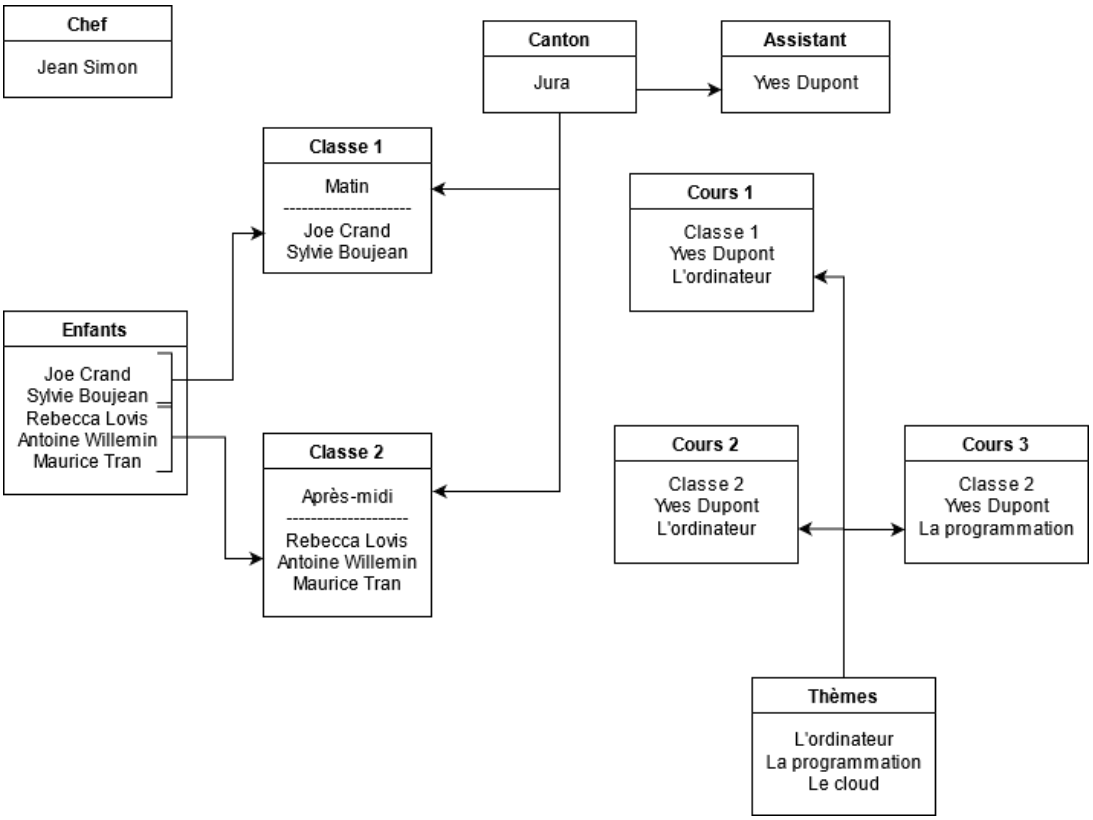

Figure 2. – Exemple fictif de vue d'ensemble des cours extra-scolaires

La première classe suit un cours sur l'ordinateur avec par conséquent Yves Dupont, puisque c'est le seul assistant travaillant pour le canton du Jura. La deuxième classe suit deux cours, un sur le même thème et un sur la programmation, toujours avec le même assistant. Cet exemple peut évidemment être étendu, à plus d'enfants, de cantons, de classes, etc.

La base de données relationnelle peut désormais être construite. Il en résulte la Figure 3 qui ressemble fortement à la Figure 1, puisqu'elle a été développée à partir de celle-ci. Chaque entité représente une table, à la différence près que la table "Enseignement" est

là en plus, car il y a une relation multiple-multiple entre assistant et cours. En ajoutant cette table, la redondance d'attributs est éliminée.

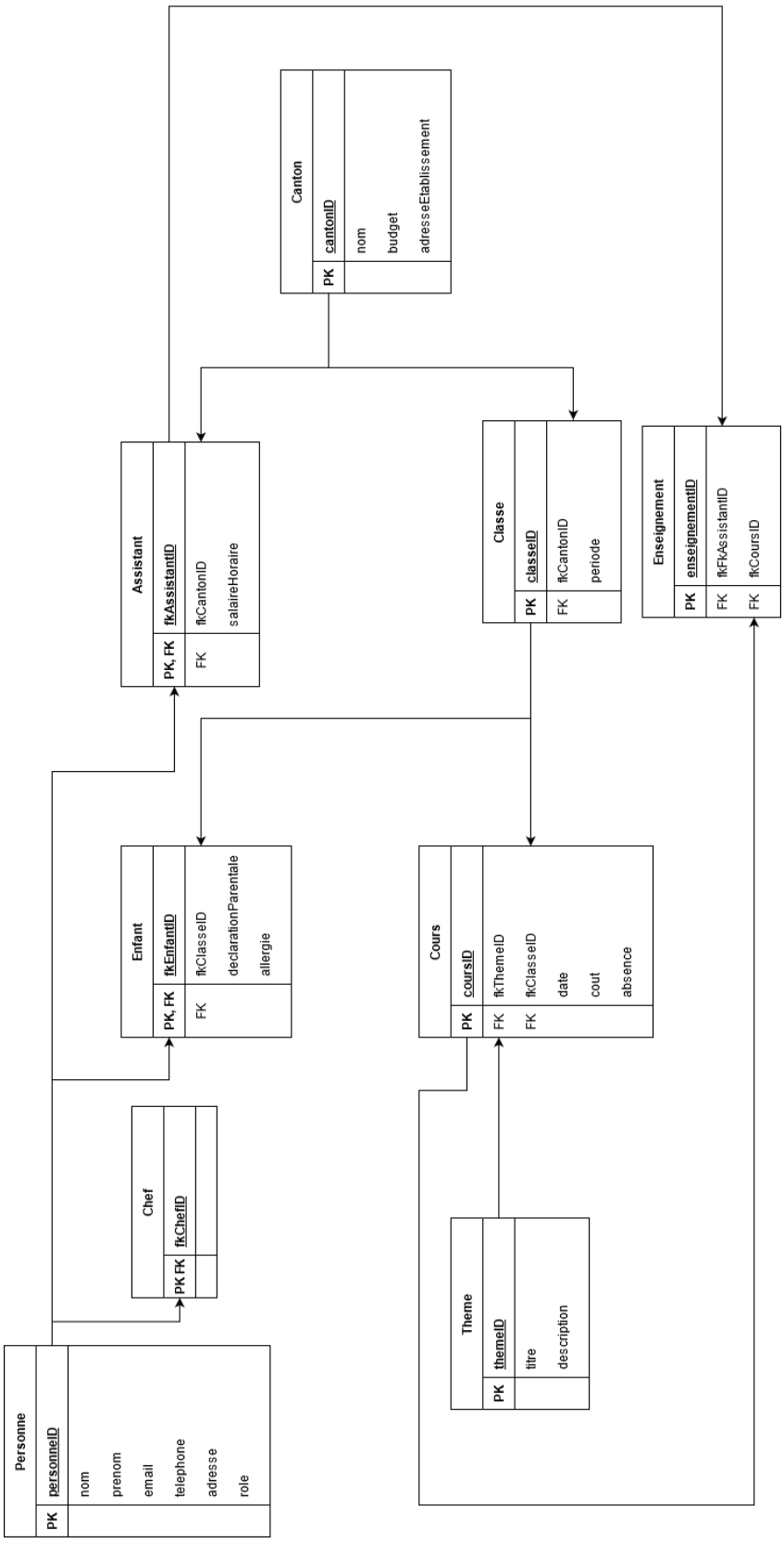

Figure 3. – Schéma base de données relationnelle

Les clés primaires sont représentées par PK (*primary key*) et les clés étrangères par FK (*foreign key*). Les tables *Chef*, *Enfant* et *Assistant* ont toutes comme clé primaire la clé étrangère de l'ID de la personne. En faisant cela, toutes les informations générales sur la personne sont retrouvées. A noter que l'attribut *adresse* dans la table *Personne* et *Canton* contient plusieurs autres attributs qui sont le *code postal*, la *ville*, le *nom de rue* et le *numéro de rue*. Cela n'a pas été explicité ici pour des raisons de lisibilité.

Toujours dans la table *Canton*, il y a un attribut *budget*. En effet, à chaque établissement par canton est allouée une certaine somme d'argent qui pourra être utilisée comme bon leur semble par les assistants. Lors de chaque cours, une dépense est possible et est visible dans la table *Cours* avec l'attribut *cout*. Ces coûts peuvent être de plusieurs natures comme l'achat de goûters pour les enfants ou de matériels supplémentaires pour rendre le cours plus attractif par exemple.

2.3. Fonctionnalités désirées

Un certain nombre de fonctionnalités est souhaitable pour que le service aide à l'organisation des cours extra-scolaires. Cette partie détaille celles qui sont idéales dans une première analyse. Dans tous les diagrammes de cas d'utilisation qui suivent, les acteurs sont le chef et l'assistant étant donné que l'application client-serveur n'est destinée à être employée que par eux. L'application est séparée en plusieurs systèmes : gestion des utilisateurs, gestions des enfants, gestions des cantons, gestion des classes, gestions des cours et gestion des thèmes. Seuls les systèmes de gestion des utilisateurs et de gestion des cours sont développés dans cette section, les quatre autres diagrammes de cas d'utilisation étant en annexes, car construits de façon similaire avec des fonctionnalités semblables.

Pour commencer, la Figure 4 représente le diagramme UML de gestion des utilisateurs. Comme mentionné précédemment, un utilisateur est un assistant ou un chef, mais pas un enfant. Le chef est le seul à pouvoir créer, supprimer et modifier n'importe quel compte utilisateur. L'assistant ne peut modifier que son compte. En revanche, les deux acteurs peuvent s'authentifier sur leur compte, ce qui permet de gérer les accès en conséquence. Les deux peuvent également consulter tout compte utilisateur.

Pour ce diagramme de cas d'utilisation du système de gestion des utilisateurs, uniquement un des cas d'utilisations est explicité. Le cas d'utilisation "Modifier les données personnelles" est donc spécifié dans le tableau juste en dessous de la Figure 4 concernée. En plus du déroulement basique, deux déroulements alternatifs sont possibles. Le premier étant qu'un assistant ne réussira pas à modifier les informations d'un compte qui ne lui appartient pas et le second étant que l'enregistrement demande une confirmation, ce qui implique que si l'utilisateur modifie par erreur une donnée, cette dernière n'est pas automatiquement sauvegardée. Une action manuelle est nécessaire à la validation de celle-ci.

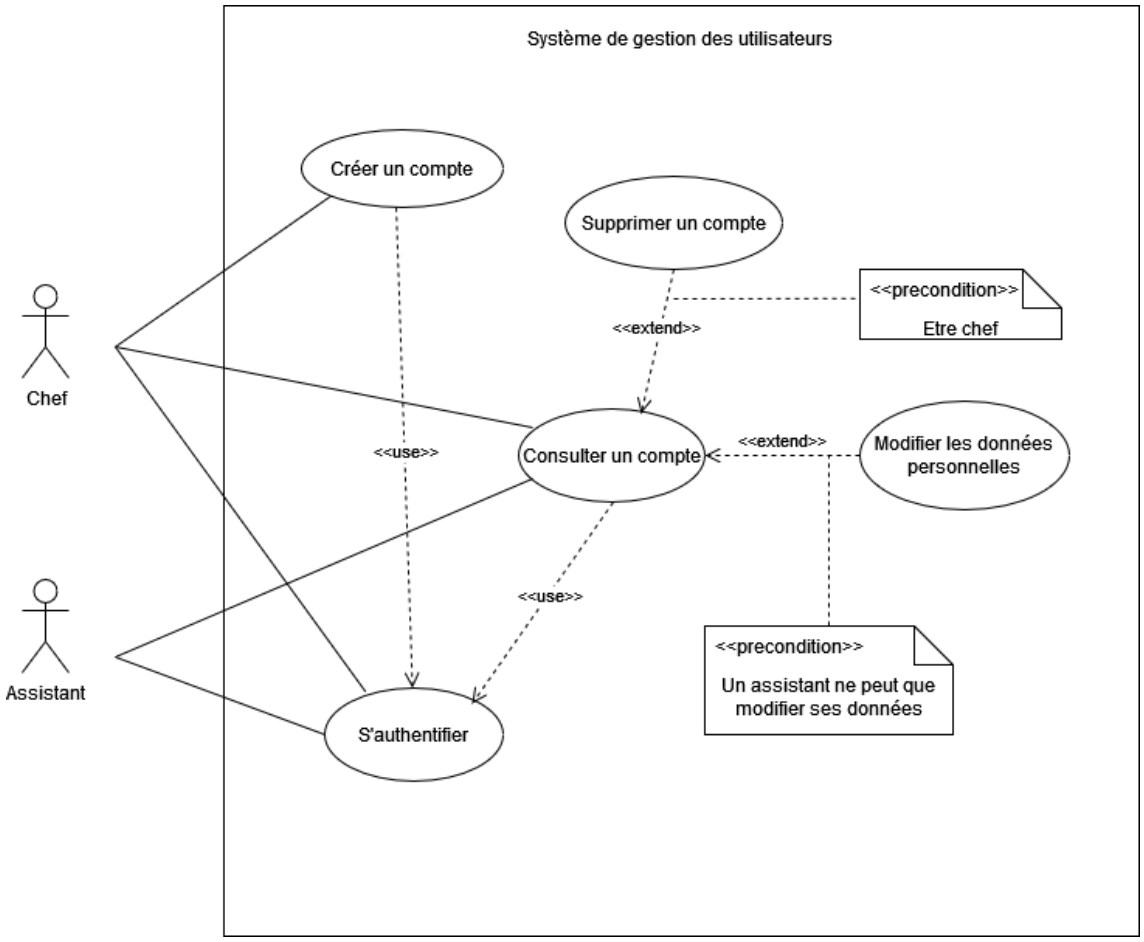

Figure 4. – Diagramme UML de cas d'utilisation du système de gestion des utilisateurs

Titre	Modification des données (d'un compte utilisateur)	
Description et but	Modifier un compte utilisateur et ses données personnelles afin de les compléter ou les mettre à jour	
Acteurs	Chef, Assistant	
Préconditions	L'acteur s'est authentifié	
Postconditions	Le compte utilisateur est mis à jour	
Déroulement basique	Étape	Action
	1	L'acteur consulte le compte à modifier
	2	L'acteur clique sur modifier le compte | Alternative 1
	3	L'acteur modifie les données
	4	L'acteur enregistre les modifications | Alternative 2
	5	L'acteur confirme son action
Déroulement alternatif	Alternative 1 : Un assistant ne peut modifier que son compte	
	Étape	Action
	1	L'option modification n'est pas disponible sur les autres comptes que celui de l'assistant
	Alternative 2 : Enregistrement de données erronées	
	Étape	Action
	1	L'acteur ne confirme pas la modification
	2	Retour à l'étape 3 du déroulement basique

Concernant le second diagramme UML qui est présenté, il décrit la gestion des cours et est modélisé avec la Figure 5. Comme pour la Figure 4, le chef peut créer, supprimer et modifier n'importe quel cours. Cette fois-ci, l'assistant peut aussi modifier un cours et autant ce dernier que le chef peuvent consulter un cours. L'assistant a la possibilité de s'inscrire pour donner un cours ou se désinscrire pour ne plus donner un cours. Cela laisse la liberté aux assistants de s'organiser entre eux, par exemple si pour un cours ils estiment qu'un seul assistant est suffisant ou si au contraire il en faudrait au moins trois.

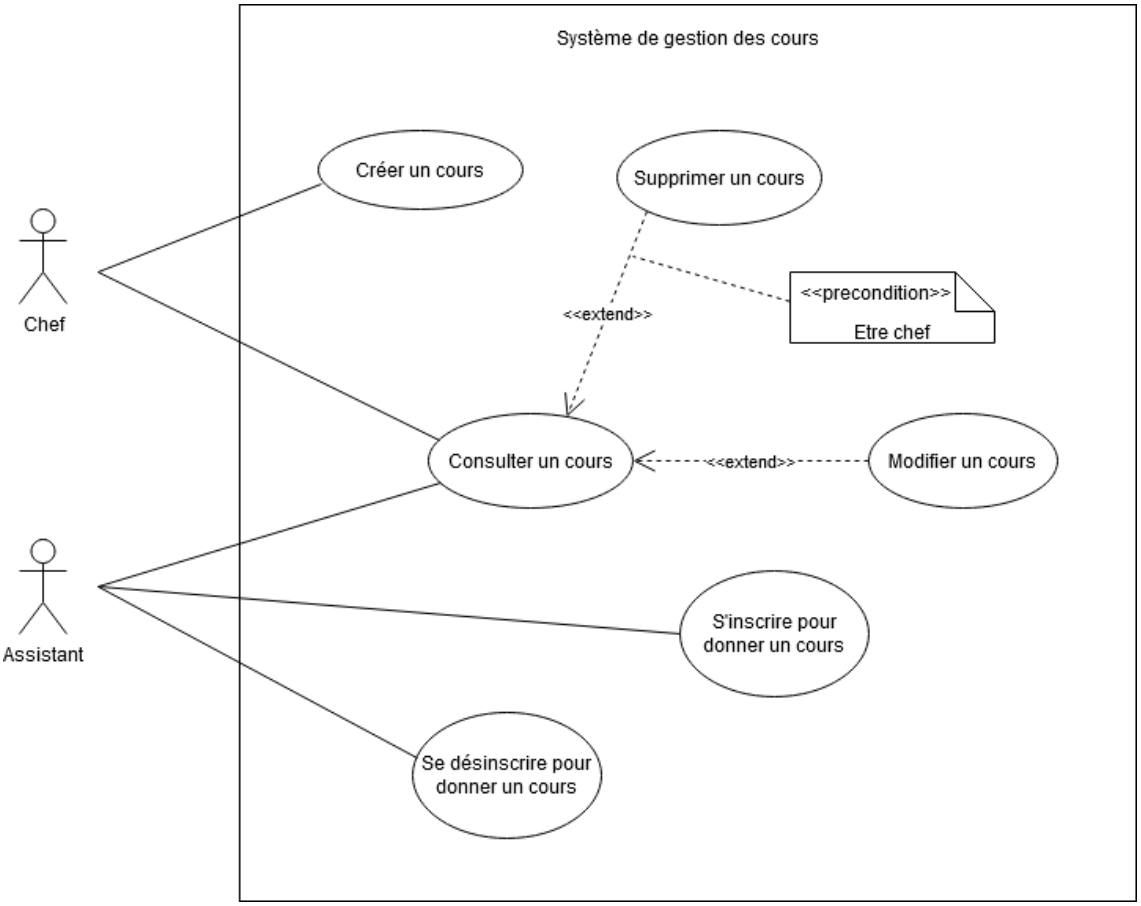

Figure 5. – Diagramme UML de cas d'utilisation du système de gestion des cours

Cette fois-ci, le cas d'utilisation "Créer un cours" est détaillé dans le tableau qui suit. Il ne possède qu'un déroulement alternatif lorsque l'application trouve déjà un cours similaire, c'est-à-dire avec la même classe et le même thème.

Titre	Création d'un cours	
Description et but	Créer un cours avec toutes les données correspondantes afin d'aider à l'organisation	
Acteurs	Chef	
Préconditions	Le chef s'est authentifié	
Postconditions	Le cours est créé	
Déroulement basique	Étape	Action
	1	Le chef clique sur créer un nouveau cours
	2	Le chef remplit les données du nouveau cours
	3	Le chef enregistre le nouveau cours | Alternative 1
Déroulement alternatif	Alternative 1 : Un cours possède le même thème et la même classe	
	Étape	Action
	1	Le cours déjà existant possédant le même thème et la même classe s'affichent à titre de comparaison
	2	Demande de confirmation de création du nouveau cours

Les fonctionnalités présentées ici et en annexe sont celles qui ressortent comme idéales. En revanche dans l'application créée, l'authentification n'a pas été réalisée et par conséquent les préconditions concernant cette authentification dans les différents cas d'utilisation ne sont pas vérifiées. Les autres fonctionnalités et étapes restent cependant globalement les mêmes.

3

Présentation du point de vue utilisateur final

3.1. Concept global

L'architecture de l'application est établie à partir d'un serveur, d'un client et d'une base de données. Le serveur reçoit de multiples requêtes venant du client et son rôle est d'y répondre. Le serveur fait par conséquent office de service au client. Pour satisfaire une requête, le serveur a besoin de certaines données qui sont stockées dans une base de données contenant plusieurs tables, comme vu précédemment à la Figure 3. Le point de vue de l'utilisateur final correspond donc à l'application côté client utilisée soit par un assistant soit par un chef, ou en d'autres termes à l'interface utilisateur. Étant donné que l'application n'est pas déployée, la page d'accueil se trouve à l'adresse `http://localhost:8080/`. Toutes les autres pages de l'application commencent par la même URL et sont étendues, par exemple `http://localhost:8080/cantons/list` qui affiche l'ensemble (des établissements) des cantons. A noter qu'une page d'erreur s'affiche lorsqu'une URL est fausse. De plus, l'application a été conçue en anglais, par conséquent les figures qui sont présentées contiennent les termes équivalents du français de ce qui est expliqué dans le travail.

Tout d'abord, sur chacune des pages, il y a constamment une barre de navigation située tout en haut. Cette dernière permet d'accéder aux différentes rubriques concernant les cantons, les personnes, les cours, les classes, les enseignements, les thèmes ou tout simplement la page d'accueil. Cette barre sert donc de menu et permet de naviguer d'une section à l'autre sans passer par une page spécifique. Cette barre de navigation est visible sur la Figure 6 en couleur brun-rouge, plus foncée que le reste. Étant donné que chaque rubrique est construite de façon similaire, uniquement la partie concernant les enseignements est explicitée. Un aperçu des autres pages est visible dans la section suivante, lors de l'exposition de scénarios possibles. A noter également que le contenu s'adapte en fonction de la taille de la fenêtre et qu'une barre de défilement verticale apparaît si nécessaire.

Ensuite, chaque page commence par un titre qui résume ce qu'elle contient. Lorsqu'une rubrique est sélectionnée, la page affichée comporte la liste des objets non détaillés. En reprenant donc la Figure 6 ci-dessous, la page est titrée comme "Liste de tous les enseignements". Lorsque la page de modification est sélectionnée, le titre est alors "Mise à jour de l'enseignement", lorsque la page de vue est sélectionnée, le titre est "Détails de l'enseignement" et lorsque la page d'ajout est sélectionnée, le titre est "Ajout d'un enseignement". Les pages des autres rubriques sont appelées de façon analogue : "Liste de tous les cours", "Liste de tous les thèmes", "Détails du thème", etc.

Sur cette page de liste, juste en dessous du titre se trouve un bouton permettant d'ajouter un objet à la liste en question, ici un enseignement, qui redirigera l'utilisateur vers une nouvelle page visible à la Figure 8. Juste en dessous, un ensemble de filtres est disponible pour trouver plus rapidement un ou des enseignements spécifiques. Le premier filtre permet de rechercher par le nom et/ou le prénom d'un assistant qui donne un cours. A noter que l'utilisateur peut ou non mettre une ou des majuscules dans le nom. Par exemple, écrire "Dupond", "dupond" ou "dUPONt" donnera le même résultat de recherche. Ensuite, le deuxième filtre permet de sélectionner un thème parmi l'ensemble des thèmes disponibles sous forme de liste, le troisième filtre fonctionne pareillement avec la liste des cantons et le quatrième filtre permet de chercher par date où a lieu un enseignement. A noter que si plusieurs établissements sont mis à disposition pour un seul canton, le nom du canton apparaîtra une seconde fois ce qui permet de différentier les établissements. Pour lancer une recherche, il suffit de cliquer sur le bouton recherche. Tous les filtres n'ont pas besoin d'être remplis pour la lancer. Le bouton d'à côté sert à réinitialiser les filtres, c'est pourquoi la liste complète des enseignements est à nouveau affichée.

Finalement, les enseignements résumés sont listés avec à leur droite trois boutons. Le premier (*View Teaching*) sert à visualiser en détail l'enseignement en question, le deuxième (*Update*) à le mettre à jour et le dernier (*Delete*) à le supprimer. Ce dernier bouton n'amène pas à une autre page comme les deux autres, mais affiche uniquement une demande de confirmation de suppression. Lorsqu'il y a confirmation de suppression, l'enseignement disparaît de la liste.

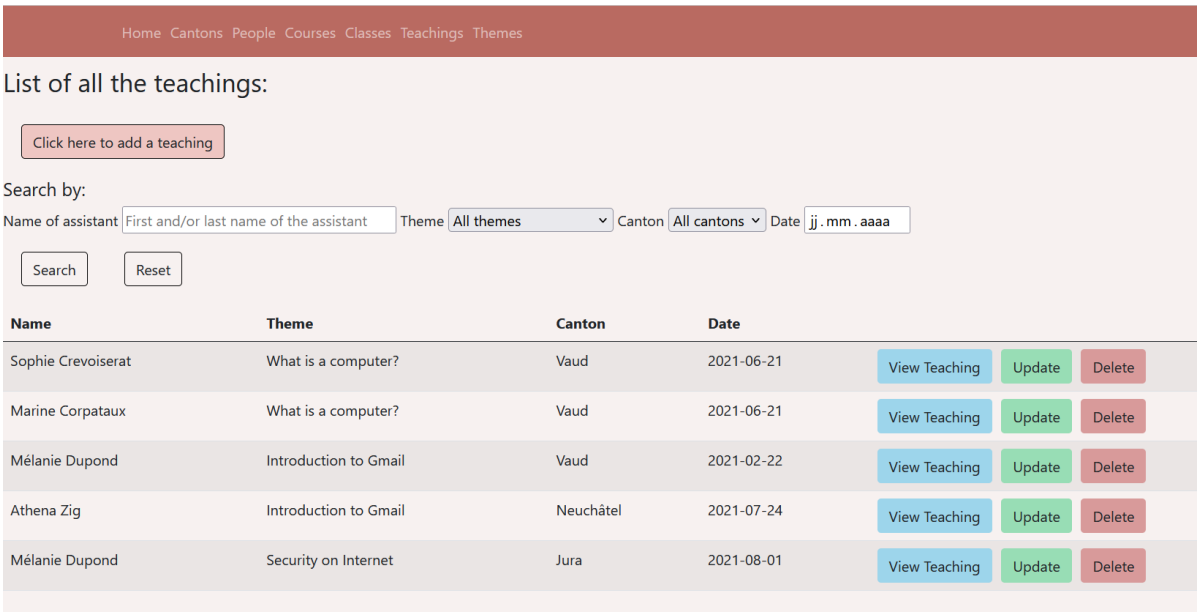

Figure 6. – Page client de la liste de tous les enseignements

Lorsqu'un utilisateur souhaite connaître les détails d'un enseignement et qu'il clique sur le bouton en question, il est redirigé vers la page montrée à la Figure 7 ci-dessous. Dans ce cas, il n'y a que peu d'information supplémentaire, contrairement au cas d'un canton ou d'une personne. Les mêmes boutons de modification et de suppression sont à nouveau visibles. De plus, la page de détails d'un enseignement ne contient pas tous les assistants qui donnent le même cours, puisque cela correspond en quelques sortes à la table *Enseignement* de la Figure 3 avec certains détails des clés étrangères à la place des clés elles-mêmes. Si un utilisateur souhaite trouver tous les assistants qui donne un même cours, il lui suffit d'utiliser les filtres *Theme*, *Canton* et *Date* à sa disposition.

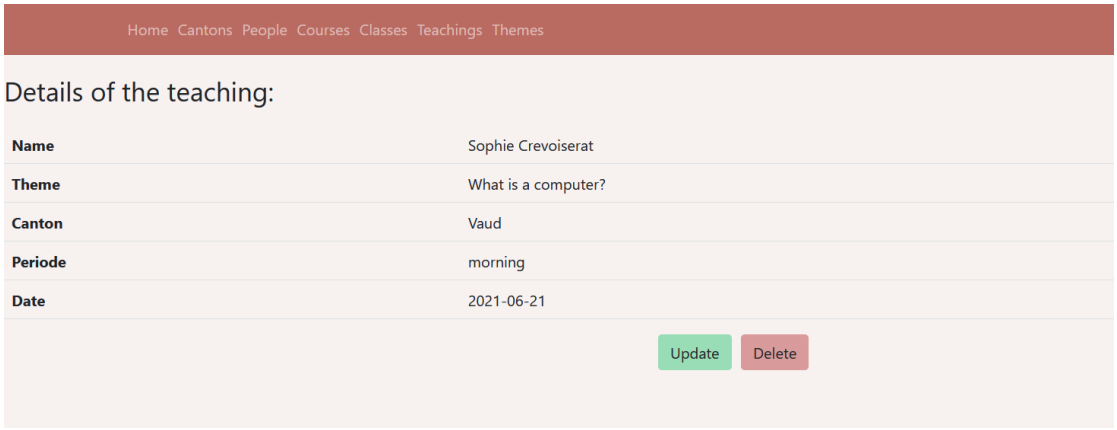

Figure 7. – Page client de détails d'un enseignement

L'ajout d'un enseignement est légèrement différent de l'ajout d'un autre objet, puisqu'il dépend du canton. Pour rappel, un assistant ne peut que donner cours dans le canton dans lequel il travaille. Par conséquent, lorsque l'assistant est sélectionné en premier, il ne reste que les cours appartenant au canton en question dans la seconde liste déroulante. Inversement, si un cours est d'abord sélectionné, il ne sera affiché que les assistants appartenant au canton en question dans la première liste déroulante. C'est pourquoi un bouton de réinitialisation est là, apparent sur la Figure 8. Pour savoir exactement quel cours l'utilisateur veut choisir, un résumé est affiché dans la liste déroulante, comme montré à la Figure 9. Concernant les ajouts des autres objets, la plupart des champs sont des zones de texte. Certaines zones sont également des champs numériques ou des listes déroulantes.

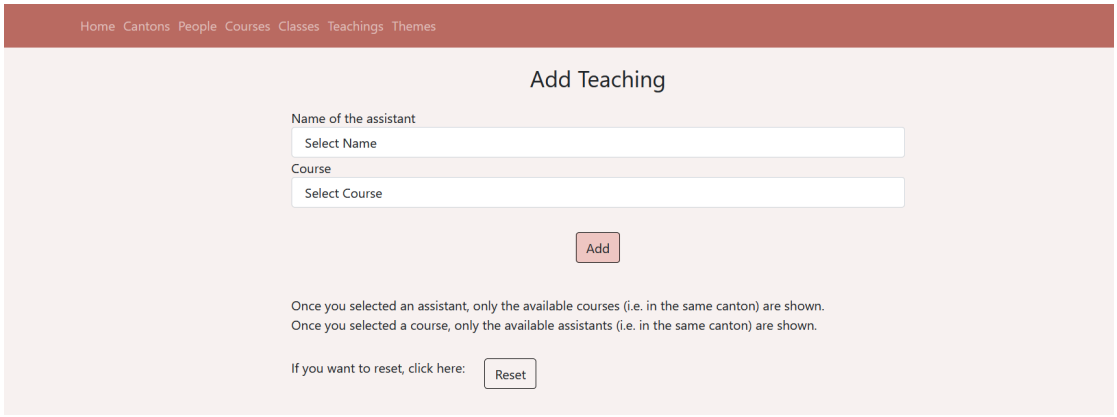

Figure 8. – Page client d'ajout d'un enseignement

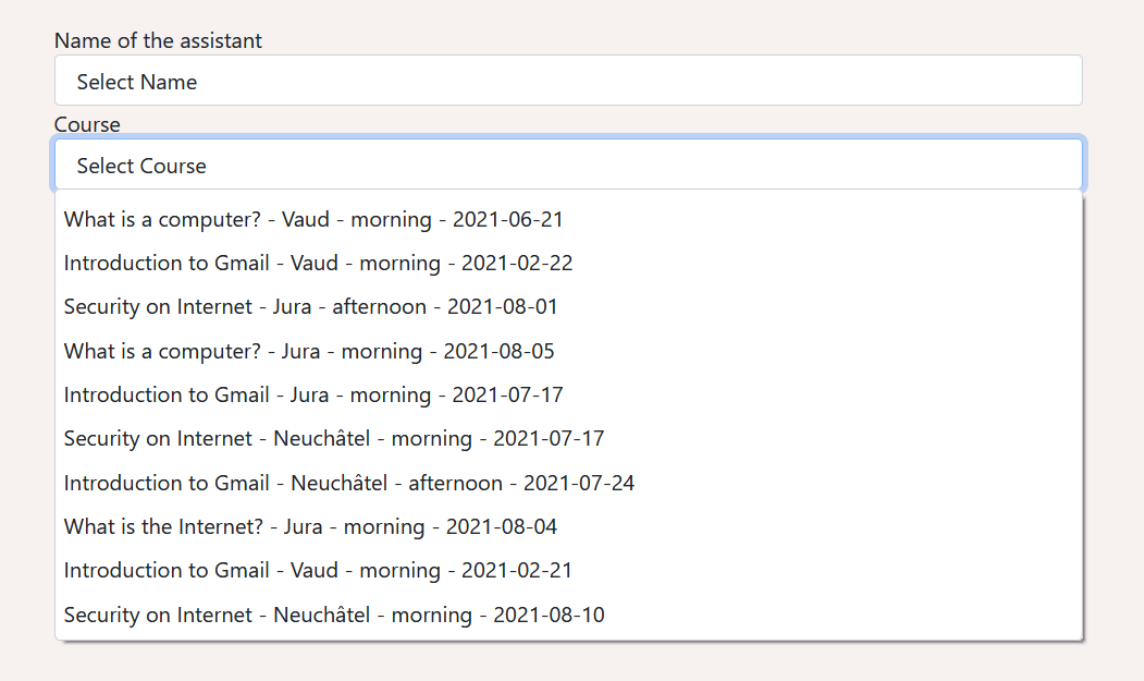

Figure 9. – Détail de la page client d'ajout d'un enseignement

Il reste à présenter la page de modification d'un enseignement. De manière générale, une page qui met à jour un objet reprend les données modifiables et les affiche directement dans le formulaire de modification. Dans la Figure 10, étant donné qu'un enseignement représente quel assistant donne quel cours et qu'un cours a déjà sa page de mise à jour, ici il n'est que possible de modifier l'assistant donnant le cours. Si le cours concerné a besoin d'être mis à jour, il suffit de cliquer sur le bouton en bas de la page qui renvoie vers le bon formulaire. Dans le cas où l'assistant s'est inscrit pour donner le mauvais cours, il suffit de supprimer l'enseignement incorrect et d'ajouter le bon.

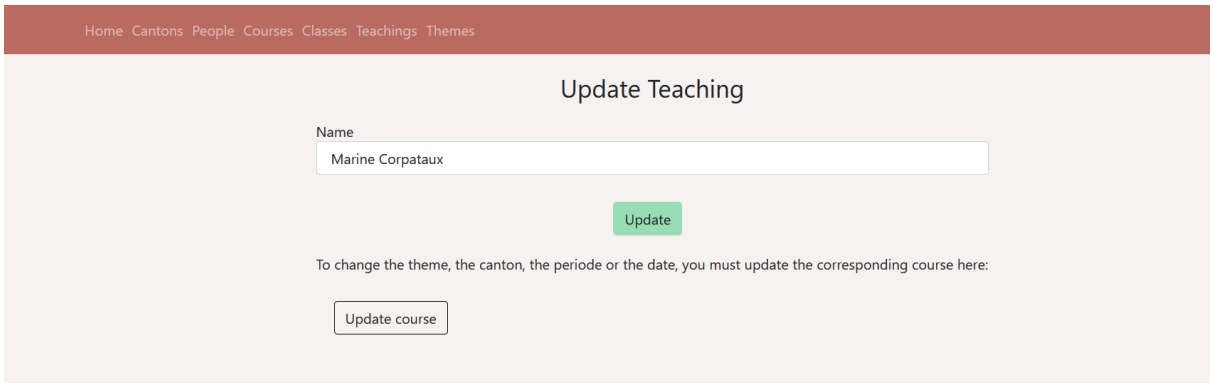

Figure 10. – Page client de modification d'un enseignement

3.2. Scénarios

3.2.1. Ajout d'un assistant

Cette sous-section montre comment ajouter concrètement un assistant dans le système. Pour commencer, il faut se rendre dans la rubrique "Personnes" visible dans la barre de navigation. Une fois dans cette rubrique, il suffit d'appuyer sur le bouton d'ajout d'une personne situé en dessus des filtres disponibles pour la recherche de personnes, comme c'est le cas pour un enseignement à la Figure 6. Ensuite, l'utilisateur arrive sur une page contenant un formulaire visible à la Figure 11. Pour pouvoir valider le formulaire, et donc ajouter une personne, il faut que tous les champs soient renseignés. Si ce n'est pas le cas, un message apparaît près du champ non complété pour demander à l'utilisateur de le faire. De plus, le formulaire vérifie que les zones concernant le code postal et le numéro de rue sont des nombres. Comme pour la complétion, si une de ces deux zones n'est pas un nombre, le formulaire ne se validera pas. A noter que le dernier champ est une liste déroulante dans lequel les rôles autorisés sont montrés.

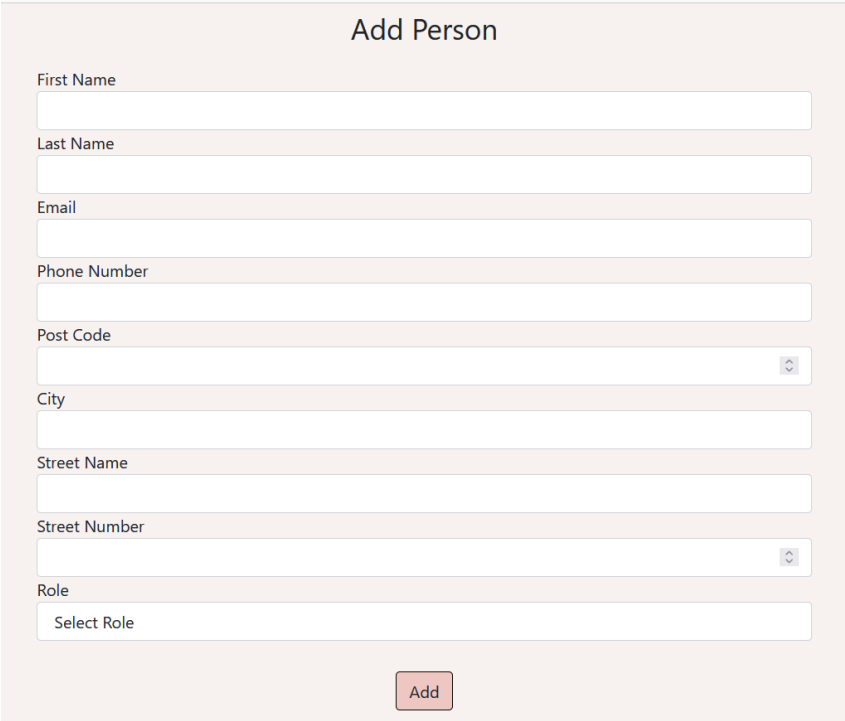

Figure 11. – Page client d'ajout d'une personne

Une fois le formulaire d'ajout d'une personne complété et validé, l'utilisateur est redirigé sur une page différente suivant le rôle attribué à la personne. Dans le cas du rôle d'assistant, l'utilisateur arrive sur la page montrée à la Figure 12. Il doit à ce moment spécifier le salaire qui sera attribué à l'assistant et pour quel canton ce dernier travaille. Dans le cas d'un enfant, l'utilisateur arrive sur une page similaire qui demande si l'enfant en question a des allergies, si la déclaration parentale a déjà été donnée ou non et dans quel canton et quelle période il suivra les cours. Dans ces deux premiers cas, une fois le formulaire validé, l'utilisateur est redirigé vers la page qui liste toutes les personnes qui sont dans le système avec cette nouvelle personne en plus. Finalement dans le cas du chef, il n'y a pas

de donnée plus spécifique à compléter et donc l'utilisateur retourne automatiquement sur cette page après la validation de la personne.

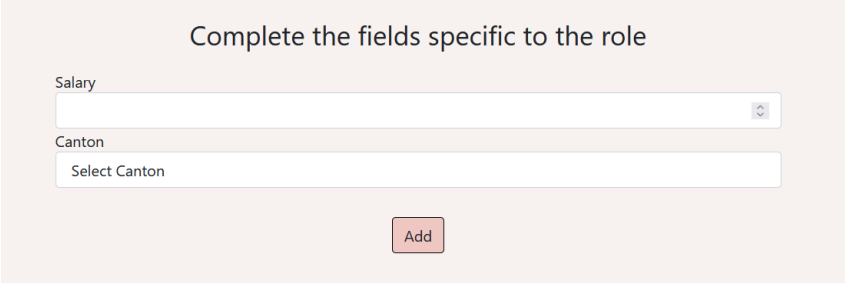

Figure 12. – Page client d'ajout d'un assistant

3.2.2. Modification d'un cours

Cette fois-ci, le scénario de modification d'un cours est décrit. Pour qu'un utilisateur puisse modifier un cours, il doit tout d'abord aller dans la rubrique correspondante, c'est-à-dire la rubrique "Cours". Ensuite, il peut soit directement appuyer sur le bouton vert "Mise à jour" du cours qu'il souhaite modifier, soit d'abord regarder les détails de celui-là pour être sûr qu'il s'agisse du bon cours. Pour rappel, l'utilisateur peut voir les détails d'un cours en cliquant sur le bouton bleu "Voir cours". Une fois le cours choisi pour la mise à jour, l'utilisateur est envoyé sur la page correspondant à la Figure 13. Toutes les informations concernant le cours sont reportées dans un formulaire. Cela permet à l'utilisateur de voir les informations du cours et de mettre à jour uniquement le nécessaire. La particularité de cette page est que les enfants qui suivent le cours en question sont listés en bas juste avant le bouton de mise à jour. Si un enfant a sa case cochée, comme c'est le cas pour Olivia Grosjean sur la Figure 13, c'est que cette personne est absente durant ce cours. Une fois la modification validée, l'utilisateur est renvoyé sur la page qui liste l'ensemble des cours et la mise à jour a été appliquée.

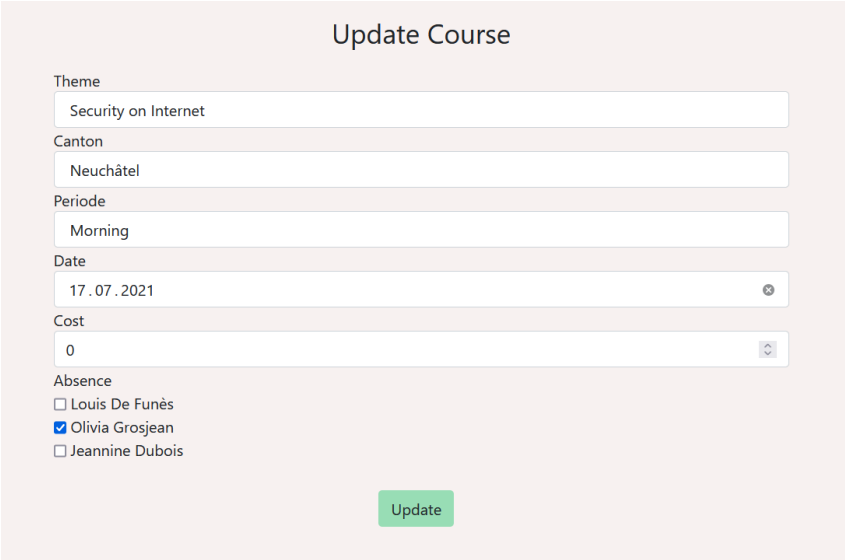

Figure 13. – Page client de mise à jour d'un cours

4

Programmation du serveur

4.1. Présentation générale

Le serveur correspond au service utilisé par le client pour obtenir des réponses à ses requêtes, ces dernières étant principalement les fonctionnalités définies dans le chapitre 2. Pour ce faire, il faut tout d'abord déterminer quelles technologies et quel type de structure utilisés. Puisque l'application est un service web client-serveur qui n'a pas besoin de conserver l'état du client et qui effectue des opérations CRUD (Create, Read, Update et Delete), utiliser une architecture REST correspond parfaitement. Cette architecture est détaillée dans la section suivante. Concernant les technologies, le serveur est codé à l'aide de Node.js [14], qui est un moteur d'exécution JavaScript. Il a la particularité d'être asynchrone et axé sur les évènements.

Ce dernier point est primordial. En effet, un langage asynchrone permet de continuer une opération sans attendre la fin d'une autre opération. Cela permet dans le cas d'une application comme celle-ci d'afficher une page même si tout n'a pas encore été obtenu. Si une réponse est nécessaire au bon fonctionnement de la suite du script, des méthodes particulières permettant d'attendre ladite réponse sont utilisées. Les exemples qui suivent sont tirés de [7] et [8] et contiennent des explications plus détaillées.

Tout d'abord, une fonction peut gérer cet asynchronisme en étant *callback* (de rappel). C'est-à-dire que la fonction de rappel contient en paramètre une autre fonction qui sera appelée lorsque la fonction principale aura fini son exécution. Une fonction *callback* est illustrée avec la Listing 1. La fonction *callback* est donc *setTimout* et prend en paramètre une fonction anonyme. Dans ce cas, la fonction anonyme est exécutée une fois que 50 millisecondes se sont écoulées et pas avant.

```
1 setTimeout(() => console.log('a'), 50);
```

Listing 1 – Exemple fonction callback

Une fonction peut aussi gérer l'aspect asynchrone en retournant explicitement une promesse (*Promise*). Une promesse est en fait un objet qui représente l'état d'une opération asynchrone. Dans certains cas, il n'est pas nécessaire de créer explicitement la promesse en utilisant le constructeur associé, puisque quelques fonctions asynchrones prédéfinies la retournent directement en tant que telle. La promesse peut être en cours (*pending*), résolue (*resolve*) ou rejetée (*reject*). La fonction retournant la promesse est généralement suivi de "*.then()*", qui est appelé si la promesse est résolue, puis "*.catch()*", qui est appelé si la promesse est rejetée. En d'autres termes ces deux fonctions sont bloquées en attendant le résultat de la promesse. Un exemple de l'utilisation de cette méthode avec la construction explicite d'une promesse est présent avec la Listing 2.

```
function asyncFunct() {
    return new Promise(function(resolve, reject) {
        //script that leads either to
        ...
        resolve(message);
        //or to
        ...
        reject(new Error(message));
    });
}

let promise = asyncFunct();
promise.then(message => {
    console.log(message);
}).catch(err => {
    console.error(err.message);
});
```

Listing 2 – Exemple fonction avec promesse

Finalement, l'usage des mots-clés *async* et *await* est également une solution. C'est notamment cette dernière possibilité qui est employée dans le programme du serveur. En employant le mot clé *async* devant une fonction, cette dernière va automatiquement retourner une promesse sans avoir à le spécifier autrement. Pour intercepter le résultat de cette promesse, et donc l'attendre, il faut utiliser le mot clé *await* à l'intérieur de la fonction concernée. De plus, *await* doit toujours être englobé dans un bloc de type "*try{} catch(err){}*" puisqu'il est nécessaire de gérer le cas de rejet de la promesse. Dans la Listing 3, il est question de rechercher un élément dans une base de données MongoDB. Le résultat de la promesse est donc le résultat de la recherche dans la base de données.

```
async asyncFunction() {
    try {
        const res = await collection.findOne({ isAsynchronous: 'true' });
        return res.answer;
    }catch(err) {
        throw new Error(err.message);
    }
}
```

Listing 3 – Exemple fonction avec async-await

Après avoir introduit le type de langage choisi, la structure du code et le code en lui-même vont être présentés. Le code est divisé en plusieurs dossiers : les modèles (*models*), les modules node (*node_modules*), les routes (*routes*) et les middlewares annexes

(*someMiddlewares*). Un premier fichier *app.js* servant à créer le serveur en tant que tel et deux autres fichiers *package-lock.json* et *package.json* servant à gérer les dépendances et les versions sont également présents à la racine du serveur. Le code complet, sans le dossier *node_ modules*, peut être trouvé sur le repository Github à l'adresse `https://github.com/SophieLydia/TB_Server`. Ce dossier n'y a pas été déposé, car il est lourd en termes d'espace de stockage et les deux fichiers *package-lock.json* et *package.json* contiennent en quelque sorte un équivalent ce dossier. En effet, *node_ modules* contient tous les modules desquels le projet dépend. Package.json contient la liste des modules nécessaires (dépendances) au serveur et donc tout ce qui est utile pour l'installation. Package-lock.json quant à lui contient un arbre de dépendance versionné qui est généré à chaque modification faite dans le fichier *package.json* ou dans le dossier *node_ modules*.

Le fichier *app.js*, comme dit dans le paragraphe précédent, sert principalement à la création du serveur, mais pas seulement. Il spécifie également l'utilisation des technologies Express.js, Mongoose, Cors et Swagger UI. Swagger UI a été utilisé pour créer rapidement une API afin de tester les différentes fonctionnalités au fur et à mesure de leurs développements et est détaillé dans la section 3 de ce chapitre. Mongoose aide à faire le lien entre la base de données et le service et Cors à utiliser le service depuis le client final. En effet, le serveur doit d'abord accepter les requêtes venant d'un autre domaine avant de les traiter. Mongoose et Swagger UI vont également être discutés plus loin, dans la section 4 de ce chapitre. Finalement, Express.js [7] est un framework pour Node.js qui apporte un ensemble de fonctionnalités afin d'aider et de simplifier la construction d'une infrastructure web comme celle-ci. La Listing 4 montre comment démarrer le serveur sur le port 8000 du localhost. La première ligne sert à importer le package Express.js et la seconde à créer une application Express. Les dernières lignes servent quant à elles à lancer le serveur sur le bon port.

```
1 const express = require('express');
2 const app = express();
3
4 app.listen(8000, () => {
5   console.log("App listening on port 8000")
6 });
```

Listing 4 – Démarrage du server

C'est également dans ce fichier que les routes nécessaires doivent être importées et utilisées par l'application (*const app*). Dans le cadre de ce travail, une route est un fichier contenant tout ce qui correspond aux routages relatifs à un objet. Autrement dit, quelle est la manière de répondre aux requêtes nécessaires suivant un chemin (*path*) et une méthode de requête HTTP spécifique. Ce routage prend la forme "*router.method(path, function)*", *method* étant la méthode de requête HTTP, *path* le chemin sur le serveur et *function* la fonction exécutée lorsque le routage est effectué. C'est bel et bien *router.method* et non *app.method* qui est utilisé dans ce travail, car il s'agit d'un middleware niveau router, ce qui apporte plus de modularité qu'un routage avec un middleware niveau application.

Dans le dossier routes, il y a par exemple le fichier *themes.js* qui va gérer tout ce qui est en rapport avec les thèmes d'un cours. Le premier routage dans ce fichier figure sur la Listing 5. Dans ce cas, la méthode de requête utilisée est GET, le chemin est "/" et la fonction qui s'active lorsque cela est déclenché est une fonction asynchrone qui va rechercher tous les thèmes présents dans la base de données et, si un filtre est donné en paramètre, va rechercher tous les thèmes contenant dans son titre ledit paramètre.

```
router.get('/', async (req, res) => {
    try{
        const themes = await Theme.find(
            req.query.title ? {"title": {"$regex": req.query.title, "$options": 'i'}} :
                {},
        )
        .select({"title": 1, "_id":1});
        res.json(themes);
    }catch(err){
        res.status(404).send("Themes not found");
    }
});
```

Listing 5 – Routage GET pour tous les thèmes

Il a été précisé que "/" est le chemin d'accès, mais ce n'est pas tout à fait correct. C'est le chemin d'accès lorsque la route correspondant au fichier *themes.js* est sélectionnée au préalable. Lors de l'importation de toutes les routes, chacune obtient un chemin. Dans le cas de nos thèmes, cela correspond au chemin "*/themes*". De plus, le serveur tourne à l'adresse "`http://localhost:8000/`", ce qui correspond au nom de domaine. Au final, la fonction retournant tous les thèmes s'activera lorsque la requête GET est lancée à l'adresse `http://localhost:8000/themes/`, soit lorsque les trois morceaux de chemin sont mis côte à côte. Si un filtre est utilisé, un "?" suivi du nom du filtre et de la recherche sont ajoutés à la fin. Par exemple `http://localhost:8000/themes/?title=Introductionto`, va retourner tous les thèmes ayant "Introduction to" dans le titre. Les notions de chemin et de *endpoint* sont développées dans la section 3 de ce chapitre.

Tous les fichiers de routes sont similaires. Il y a un routage qui retourne tous les objets contenus dans la base de données avec de potentiels filtres à disposition, un routage qui retourne un objet en fonction de son identifiant, un routage permettant de créer, et donc d'ajouter à la base de données un objet, un routage permettant de supprimer un objet et un routage permettant de modifier un objet. Comme toutes les routes fonctionnent sur le même principe, les routages présentés concernent uniquement l'objet thème. Le premier routage a déjà été montré avec la Listing 5. La Listing 6 est similaire et retourne donc, suivant le chemin "`/:id`" et la méthode HTTP GET, le thème ayant comme ID le paramètre donné lors de la requête. La route est dans ce cas dynamique. En effet, "`/:id`" est remplacé par l'identifiant de l'objet en question.

```
router.get('/:id', async (req, res) => {
    try{
        const theme = await Theme.findById(req.params.id);
        res.json(theme);
    }catch(err){
        res.status(404).send("Theme not found");
    }
});
```

Listing 6 – Routage GET pour un thème suivant son ID

Ces deux routages sont très semblables, puisque dans ce dernier cas c'est comme si un filtre correspondait à l'identifiant du thème. Lorsqu'aucun thème n'est trouvé, que ce soit dans le cas de la Listing 5 ou 6, une erreur est lancée en précisant que le ou les thèmes n'ont pas été trouvés. Si un thème n'est pas trouvé, il est possible de l'ajouter à la base de données grâce au code se trouvant à la Listing 7. Cette fois, la méthode

utilisée est POST avec le chemin "/" et un corps (*req.body*) contenant les éléments de l'objet est envoyé. Chaque élément passé en paramètre est associé à un élément de l'objet. Comme vu précédemment avec la Figure 3 de la base de données, un thème comporte deux attributs, un titre et une description. Une fois que le thème est créé avec les lignes 2 à 5, il est sauvegardé à l'aide d'une méthode asynchrone *save()* dans la base de données grâce à la ligne 7. Si l'ajout de l'objet s'est bien passé, il apparaît désormais lorsqu'une requête retournant tous les thèmes est lancée.

```
1  router.post('/', async (req, res) => {
2      const theme = new Theme({
3          title: req.body.title,
4          description: req.body.description
5      });
6      try{
7          const savedTheme = await theme.save();
8          res.status(201);
9          res.json(savedTheme);
10     }catch(err){
11         res.status(400).send("Bad request");
12     }
13 });
```

Listing 7 – Routage POST pour un thème

Maintenant, il est également intéressant de voir comment supprimer un thème, soit parce qu'il n'est plus utilisé soit parce que l'ajout n'a pas été fait avec les bons attributs et n'est donc pas utilisable. Le code est visible avec la Listing 8. La méthode ne peut supprimer qu'un thème à la fois et nécessite que l'identifiant du thème lui soit passé en paramètre. La méthode est donc DELETE et le chemin est l'identifiant, soit "*/ :id*".

```
1  router.delete('/:id', async (req, res) => {
2      try{
3          const removedTheme = await Theme.deleteOne({_id: req.params.id});
4          res.json(removedTheme);
5      }catch(err){
6          res.status(404).send("Theme not found");
7      }
8  });
```

Listing 8 – Routage DELETE pour un thème suivant son ID

Il ne reste qu'une méthode à présenter concernant les thèmes et c'est la méthode PATCH qui sert à la modification. Elle évite par exemple à un utilisateur de supprimer un thème si l'ajout n'a pas été fait correctement. Il peut tout simplement modifier les éléments nécessaires. Avec la Listing 9, la méthode est PATCH et le chemin est à nouveau l'identifiant. Il faut donner en paramètre l'identifiant permettant de retrouver le thème en question à modifier et tous les éléments que l'utilisateur souhaite modifié dans un corps.

```
9  router.patch('/:id', async (req, res) => {
10     try{
11         const updatedTheme = await Theme.updateOne(
12             {_id: req.params.id},
13             {$set: req.body
14         });
15         res.json(updatedTheme);
16     }catch(err){
```

```
17        res.status(404).send("Theme not found");
18    }
19 });
```

Listing 9 – Routage PATCH pour un thème suivant son ID

Maintenant que le fichier principal *app.js* et tous les types de routes ont été présentés, il ne reste que ce qui concerne les modèles et middlewares annexes. Dans le dossier contenant les modèles, ce sont les schémas des tables pour la base de données. Ce dossier est détaillé dans la dernière section de ce chapitre lorsque Mongoose est abordé. Concernant les autres middlewares, il s'agit tout simplement de fonctions utilisées lors de routages GET qui demandent plus de travail. Elles servent en fait à filtrer les résultats au besoin. Un exemple est présenté au paragraphe qui suit.

Avec le routage qui permet d'obtenir toutes les personnes de la base de données (méthode GET avec le chemin "*/people/*"), il existe plusieurs filtres dont celui qui retourne les personnes en fonction d'un canton. Ce filtre, pour éviter que la fonction de base ne soit trop grande et reste lisible, est situé dans le premier fichier du dossier *someMiddlewares* sous le nom de *queryByCanton.js*. Ce middleware, lorsqu'il est appelé, retourne tous les identifiants des personnes en fonction du nom de canton passé en paramètre. Pour cela, il faut tout d'abord trouver les identifiants correspondants au nom de canton qui a été donné. Cela est visible aux lignes 3 à 10 de la Listing 10. Une fois la liste d'identifiants obtenue, généralement composée d'un seul élément puisque dans la plupart des cas, chaque canton à un seul établissement, il faut trouver les assistants et enfants correspondants à ce résultat. Les assistants ont directement un attribut correspondant à l'ID du canton pour lequel ils travaillent. Il est donc facile d'obtenir ceux-ci et cela se fait en une seule ligne, la 13. En revanche, concernant les enfants, ils n'ont pas directement l'attribut nécessaire à la recherche, mais à un "niveau" au-dessus. Ils ont un attribut *classId* qui est l'identifiant de la classe dans laquelle ils suivent les cours et la classe associée à cet ID possède un attribut correspondant à l'ID d'un canton. Pour les mêmes raisons que le middleware *queryByCanton.js* existe, le middleware *queryByCantonClasses.js* existe également. Il va donc aider à trouver les identifiants des classes qui correspondent aux identifiants de canton trouvés précédemment et par conséquent trouver les enfants filtrés correctement. A ce stade, les enfants et assistants voulus sont trouvés, ce qui équivaut à la fin de la ligne 15 de la Listing 10. Il ne reste qu'à extraire tous leurs identifiants dans une liste (finalIds) qui est donc la valeur retournée lors de l'appel à ce middleware.

```
1  var QueryByCanton = async(query) => {
2      try{
3          //Find cantons by the given name
4          const cantons = await Canton.find({name: query});
5
6          //Get the ids of cantons
7          var arrayId = [];
8          for (i=0; i<cantons.length; ++i){
9              arrayId[i] = cantons[i]._id;
10         }
11
12         //Find the corresponding assistants and children
13         const assistant = await Assistant.find({cantonId: {"$in": arrayId}});
14         const classIds = await QueryByCantonClasses(arrayId);
15         const child = await Child.find({classId: {"$in": classIds}});
16
```

```
17        //Get the corresponding ids
18        var finalIds = [];
19        for (i=0; i<assistant.length; ++i){
20            finalIds[i] = assistant[i]._id;
21        }
22        for(i=0; i<child.length; ++i)
23            finalIds[i+assistant.length] = child[i]._id;
24        return finalIds;
25    }catch(err){
26        return err;
27    }
28 }
```

Listing 10 – Middleware pour filtrer les personnes en fonction d'un nom de canton

Étant donné que les autres middlewares se base sur le même principe, c'est-à-dire de retourner des identifiants en fonction d'un paramètre de recherche, ils ne sont pas développés plus ici. Le tour du programme du serveur peut donc être conclu.

4.2. Principes généraux de l'architecture REST

Ce travail est basé sur une architecture REST, ce qui est l'abréviation de REpresentational State Transfer. Cette architecture sert à créer un service web [5]. Un service web qui utilise une architecture de ce type est également appelé service web RESTful. Un service web est une application qui permet d'échanger des données et de communiquer avec d'autres applications web. Cela fonctionne en trois étapes. Tout d'abord le client réalise une requête, ensuite la requête est envoyée au serveur et finalement, une fois reçue, le serveur répond à la requête.

Pour en revenir à l'architecture REST, il définit un ensemble de règles de design dont les suivantes :

- Ressource : Tout est considéré comme ressource autant les données que les fonctionnalités. Elles sont accessibles via des URI, Uniform Resource Identifier, qui sont dans la majorité des cas des URL, Uniform Resource Locator, mais peuvent aussi être des URN, Uniform Resource Name.
- Interface uniforme : L'interaction avec les ressources se fait à travers une interface uniforme. Cela signifie que l'évolution du serveur et du client sont indépendantes l'une de l'autre puisque l'interface entre les deux est définie selon certaines règles.
- Client-serveur : Grâce à l'interface uniforme, le serveur et le client sont deux aspects qui ne dépendent pas de l'autre dans leur évolution. En conséquence, le client ou le serveur peut être remplacé ou peut évoluer séparément du temps que l'interface entre les deux ne change pas. Cela améliore la portabilité de l'interface utilisateur sur plusieurs plateformes.
- Sans état (*stateless*) : Lors de la communication entre client et serveur, le message envoyé doit contenir tout ce qui est nécessaire pour que le serveur puisse répondre à la requête. Le serveur est donc dit *stateless* car il ne conserve rien des précédentes requêtes, chacune est traitée indépendamment. Par conséquent, c'est au client de faire le nécessaire pour envoyer toutes les informations indispensables au traitement de sa requête.

D'autres principes existent mais ceux-là sont les principaux. Ils proviennent des références [10], [16] et [21]. Ces mêmes références expliquent plus en profondeur ce qu'est l'architecture REST et sont une bonne source d'information complémentaire, en particulier [16] qui provient de la thèse de Roy T. Fielding qui a posé les bases de cette architecture.

Il a été dit que la communication doit être sans état et c'est le protocole de communication HTTP, HyperText Transfert Protocol, qui est celui utilisé dans l'application de ce travail. Il possède plusieurs méthodes intéressantes dont :

- GET : Cette méthode sert à obtenir ou lire (*read*) la représentation d'une ressource. A chaque appel sur la même ressource, elle retournera la même chose. Elle ne doit en aucun cas être dérivée afin de pouvoir modifier la ressource.
- POST : Cette méthode sert à créer (*create*) une ressource. C'est à ce moment que la ressource obtient un identifiant unique.
- PATCH : Cette méthode sert à modifier (*update/modify*) un ou plusieurs champs d'une ressource. Les autres champs restent quant à eux inchangés.
- PUT : Cette méthode est très similaire à la méthode PATCH. Elle sert également à modifier (*update/replace*) une ressource, mais dans ce cas tous les champs sont modifiés. Étant donné que tout est remplacé, c'est comme si une nouvelle ressource est créée. Il arrive donc que cette méthode soit utilisée comme équivalent à la méthode POST car si la ressource sur laquelle cette méthode est utilisée n'existe pas encore, la ressource est alors créée à ce moment-là.
- DELETE : Cette méthode sert à supprimer (*delete*) une ressource. En conséquence, la ressource supprimée n'est plus ni accessible ni modifiable.

D'autres méthodes HTTP existent, telles que HEAD, CONNECT, OPTIONS, TRACE, qui ne seront pas détaillées ici. Dans le serveur de l'application, seules les méthodes GET, POST, PATCH et DELETE sont employés.

4.3. Endpoints avec Swagger UI

Comme vu dans la section précédente, l'application est un service web RESTful. Pour cela, il est nécessaire d'avoir une API, Application Programming Interface, qui suit donc l'architecture REST, car tout service web est en réalité une API. L'inverse n'est pas nécessairement vrai. Les références de la bibliographie concernant les API sont [1], [6] et [15]. En effet, une API permet la communication entre plusieurs systèmes sans connaître les détails de son implémentation. En d'autres termes, elle établit un contrat décrivant comment l'interaction doit se faire, dans ce cas-ci entre le client et le serveur. Les différents routages dans le serveur correspondent aux routages de l'API et sont également nommés *endpoints* [2]. Un *endpoint* (point de terminaison) est donc l'emplacement à partir duquel les ressources sont accessibles. En reprenant la Listing 5, le chemin `http://localhost:8000/themes/` est en réalité un *endpoint* rendant possible l'accès à la liste des thèmes.

Swagger [18] est une technologie permettant de documenter une API et Swagger UI est l'interface utilisateur correspondant. Pour apercevoir tous ces *endpoints* et pouvoir tester toutes les fonctionnalités offertes par le serveur afin de détecter d'éventuelles erreurs de programmation, cette technologie a donc été utilisée. Sa mise en place a été aidée grâce à la référence [9]. Un aperçu ne contenant pas les détails de cette interface est montré à la

Figure 14. Chacune des huit sections contient le détail des différents *endpoints* associés à chacune des huit routes du serveur.

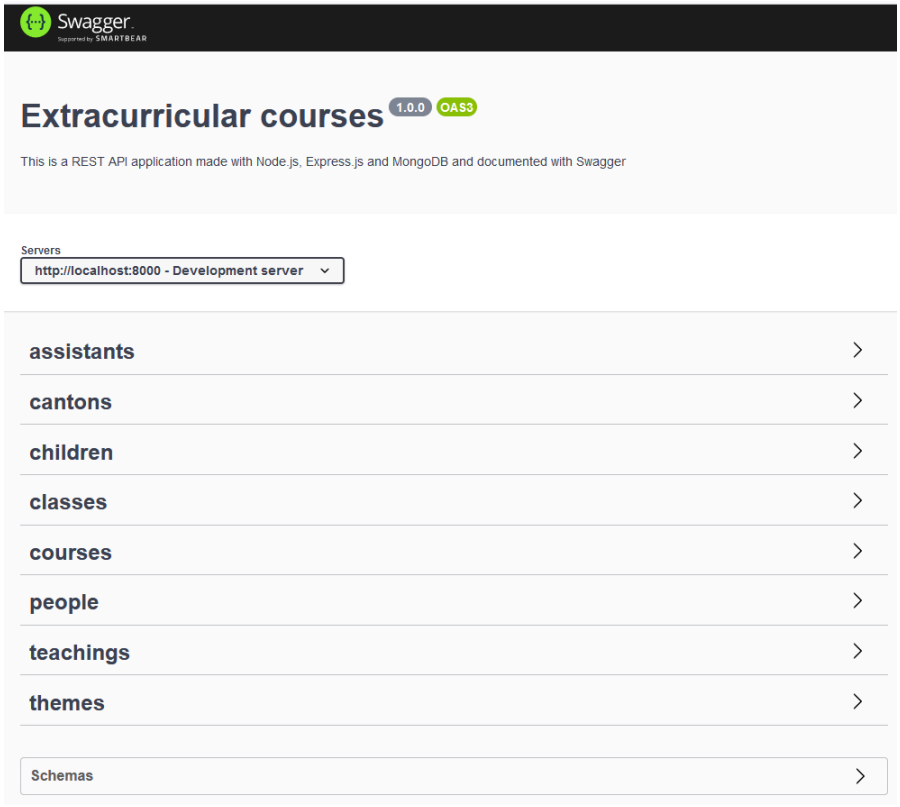

Figure 14. – Interface utilisateur de Swagger UI

La section "Schemas" quant à elle contient un ensemble de structures de données utilisées dans les sections au-dessus. Ces structures sont constituées du nom des attributs, de leur type, d'un exemple et d'une description de l'attribut. Cela rend la compréhension des objets plus simple. Un exemple est donné avec la Figure 15 lorsque la section est déroulée et que le schéma *NewClass* est sélectionné. Dans ce schéma est visible ce qui est affiché à titre d'exemple lors de la création d'une classe, autrement dit quelles sont les informations à donner. Il faut donc un attribut *cantonId* qui est une chaîne de caractères (*string*) et qui représente l'identifiant du canton auquel sera affiliée la classe et un attribut *periode* qui est également une chaine de caractères et qui représente la période durant laquelle la classe aura cours (matin ou après-midi).

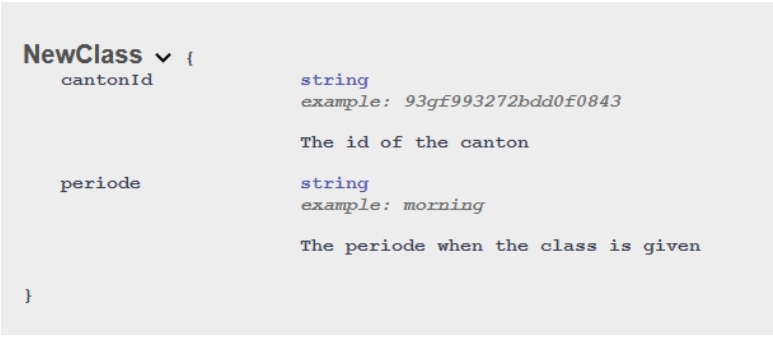

Figure 15. – Schéma Swagger d'une nouvelle classe

Lorsqu'une section est sélectionnée, par exemple *cantons* avec la Figure 16, un premier niveau de détail est visible. Les différents *endpoints* sont affichés et, suivant les fonctionnalités, un code couleur est appliqué. Les requêtes GET sont en bleu, POST en vert, DELETE en rouge et PATCH en vert clair. Dans l'affichage des *endpoints*, le nom de domaine n'est pas représenté puisqu'il ne change jamais. A noter qu'un même *endpoint* peut être utilisé de plusieurs manières suivant la méthode HTTP employée. Par exemple GET/cantons/id permet d'obtenir les informations d'un canton particulier et DELETE/-cantons/id permet de supprimer un canton particulier.

Figure 16. – Swagger UI concernant les cantons

Pour un deuxième niveau de détail, il suffit de cliquer sur le *endpoint* voulu. Chacun des *endpoints* est exposé de la même façon avec Swagger UI, c'est-à-dire en deux parties.

Dans un premier temps, il y a la description du *endpoint* avec éventuellement des paramètres obligatoires ou non (*path* ou *query*) pour pouvoir effectuer la requête et en fonction de la méthode un corps de la demande (*request body*). Lors de la présence d'un ou de plusieurs paramètres, une définition de chaque paramètre est donnée, le type est spécifié, chaîne de caractères, entier, etc., ainsi que le genre, *query* ou *path*. Lorsque c'est un paramètre *path*, celui-ci est indispensable pour pouvoir lancer la requête puisqu'il va compléter dynamiquement le chemin pour trouver le *endpoint*. Lorsque c'est un paramètre *query*, à moins que ce ne soit signalé, il n'est pas obligatoire de remplir le champ. Lorsqu'un corps de la demande est présent, il faut le compléter avant d'envoyer la requête. Il est présent dans les méthodes POST et PATCH. En effet, lors de la création d'une ressource, le contenu de la ressource doit être spécifié pour pouvoir être créée. De même lors de la modification d'une ressource, si les attributs à mettre à jour ne sont pas spécifiés, la requête ne fonctionnera pas correctement.

Dans un deuxième temps, les différentes réponses possibles sont exposées avec une description de celle-ci et le code d'état HTTP associé. Un code d'état HTTP représente un type de réponse. Ceux utilisés dans l'application sont 200, 201, 400 et 404, mais il en existe encore bien d'autres. Le code 200 signifie que la requête a été traitée avec succès, 201 que la création d'une ressource a été faite avec succès, 400 que la requête comporte une syntaxe invalide et 404 que la ressource n'a pas été trouvée. De manière générale, les codes commençant par un 2 sont des codes de succès et les codes commençant par un 4 des codes d'erreur.

En commençant par le premier cas de la Figure 16, le deuxième niveau de détail est la Figure 17 et 18. Sur la Figure 17, il y a tout d'abord la description de ce qui se passe quand

GET est appelé sur le *endpoint*, c'est-à-dire que la liste de tous les cantons se trouvant dans le système est retournée. Ensuite, soit la requête peut être lancée telle quelle, soit il est possible d'ajouter un paramètre *query* qui, dans ce cas, sert à filtrer cette liste en fonction d'une chaîne de caractères donnée représentant le nom d'un canton.

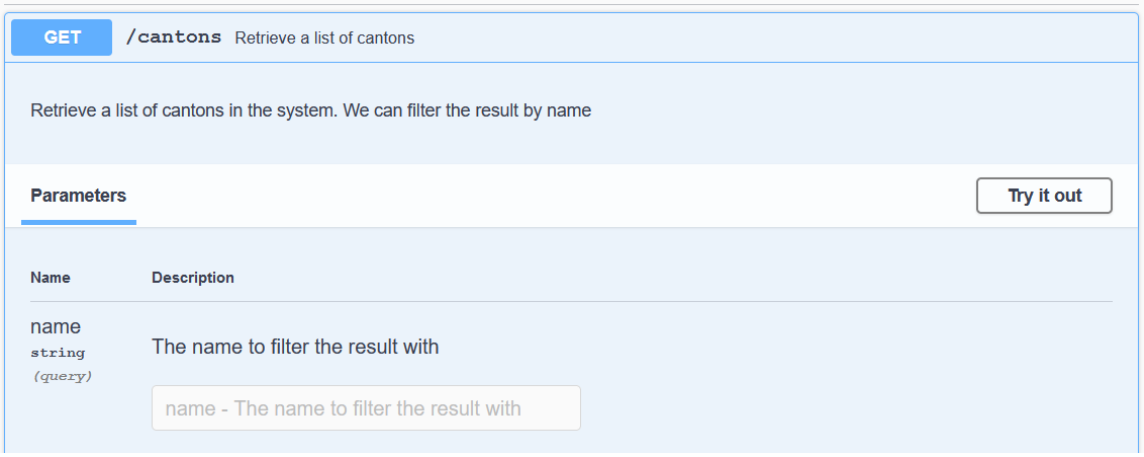

Figure 17. – Swagger UI requête GET pour tous les cantons

Dans la Figure 18 qui compose la deuxième partie, un aperçu des réponses possibles est affiché. Soit un code d'état HTTP 200 est renvoyé, ce qui signifie que tout c'est bien passé, soit un code d'état HTTP 404 est renvoyé, ce qui signifie qu'il y a eu un problème et que le serveur n'a pas trouvé de canton à envoyer comme réponse. Dans le cas où tout se passe correctement, une valeur d'exemple est affichée et montre à quoi ressemble la réponse. Cette réponse est structurée à l'aide des schémas, comme expliqué plus haut. Cette réponse va donc être une liste d'objets, qui sont des cantons, composé d'un identifiant et du nom du canton associé.

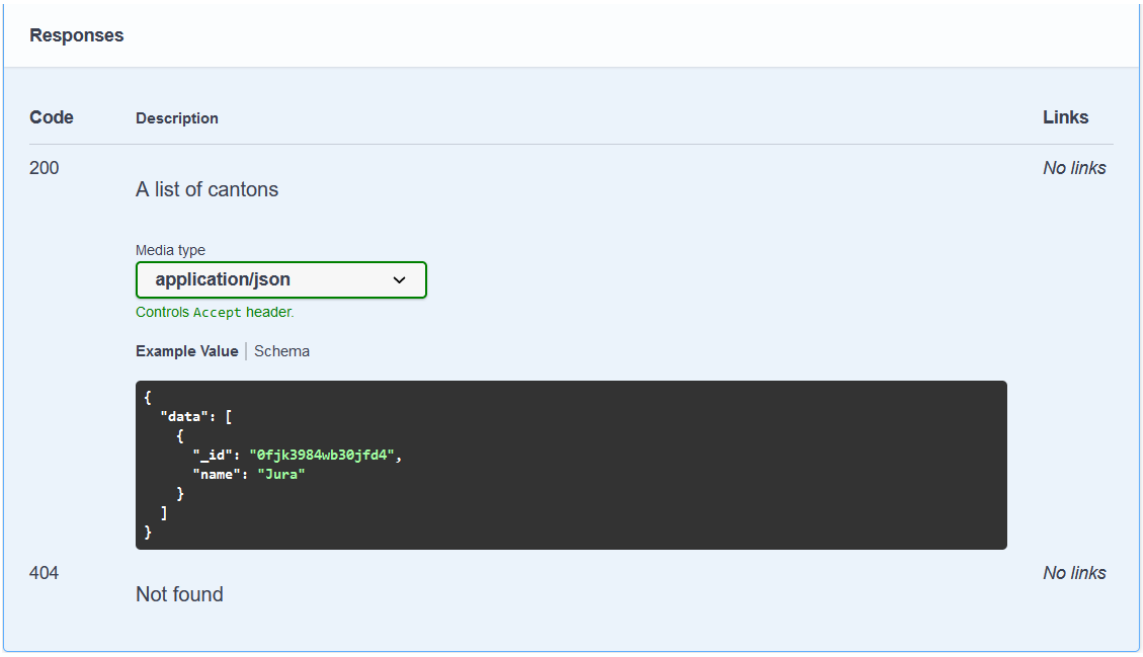

Figure 18. – Swagger UI réponse GET pour tous les cantons

Les autres méthodes HTTP ainsi que les différents *endpoints* sont construits de la même façon et ne sont donc pas plus développés, uniquement une brève liste est présentée.

Concernant les assistants
- POST/assistants
- GET/assistants/{id}
- DELETE/assistants/{id}
- PATCH/assistants/{id}

Concernant les cantons
- GET/cantons
- POST/cantons
- GET/cantons/{id}
- DELETE/cantons/{id}
- PATCH/cantons/{id}

Concernant les enfants
- POST/children
- GET/children/{id}
- DELETE/children/{id}
- PATCH/children/{id}

Concernant les classes
- GET/classes
- POST/classes
- GET/classes/{id}
- DELETE/classes/{id}
- PATCH/classes/{id}

Concernant les cours
- GET/courses
- POST/courses
- GET/courses/{id}
- DELETE/courses/{id}
- PATCH/courses/{id}

Concernant les personnes
- GET/people
- POST/people
- GET/people/{id}
- DELETE/people/{id}
- PATCH/people/{id}

Concernant les enseignements
- GET/teachings
- POST/teachings
- GET/teachings/{id}
- DELETE/teachings/{id}
- PATCH/teachings/{id}

Concernant les thèmes
- GET/themes
- POST/themes
- GET/themes/{id}
- DELETE/themes/{id}
- PATCH/themes/{id}

Pour consulter la page Swagger UI de l'application et avoir plus de détails, il suffit de mettre en marche le serveur et de consulter la page `http://localhost:8000/docs/`.

4.4. Autre technologie utilisée

Pour la programmation du serveur, il est nécessaire de faire le lien avec une base de données. Le système de gestion de base de données MongoDB [12] a été choisi et le framework JavaScript associé, soit Mongoose [13], a été utilisé. Étant donné que MongoDB fonctionne avec des documents structurés, il est primordial de définir le format de ces objets, en l'occurrence d'objet en format JSON, signifiant JavaScript Object Notation. Avec MongoDB, un document est tout simplement un élément de la base de données qui contient un ensemble de paires clés-valeurs. Des informations supplémentaires se trouvent à [19].

Pour effectuer le lien du serveur à la base de données, il suffit de spécifier la connexion à la base voulue dans le fichier *app.js* du serveur. Une fois la connexion faite, il faut spécifier les modèles, et donc la structure, des documents. Chaque modèle correspond donc à une table de la Figure 3. Tous les modèles se trouvent dans le dossier modèle du serveur. Avec

la Listing 11, la structure (schéma) d'un assistant est montrée. Lors de la création d'un élément dans la base de données, même si cela n'est pas spécifié, un identifiant est donné avec l'attribut *_id*. Dans le cas de l'assistant, l'identifiant est une référence à la personne et donc à un autre document d'un autre schéma. L'attribut *cantonId* est lui aussi une référence. Dans ces deux cas, il est précisé que ces attributs sont requis pour la création d'un document avec les lignes 6 et 11 de la Listing 11. Finalement, il y a un attribut qui correspond au salaire. Si la valeur de cet attribut n'est pas spécifiée lors de la création, il prendra automatiquement la valeur de 20. Avec Mongoose, chaque attribut possède un type et ils en existent plusieurs prédéfinis comme les chaînes de caractères, les nombres, les booléens, etc. Il en existe des plus particuliers comme les *ObjectId* qui servent lors de référence à un autre document, ou des objets emboîtés (*nested*) comme l'adresse dans les modèles *Canton* et *Person*. Finalement, chaque modèle utilise un *timestamps*. Cela permet de voir quand un document est créé avec *createdAt* et quand il a été mis à jour la dernière fois avec *updatedAt*.

```
1  const assistantSchema = mongoose.Schema (
2      {
3          _id: {
4              type: mongoose.Schema.Types.ObjectId,
5              ref: "Person",
6              required: true
7          },
8          cantonId: {
9              type: mongoose.Schema.Types.ObjectId,
10             ref: "Canton",
11             required: true
12         },
13         salary: {
14             type: Number,
15             default: 20.00
16         }
17     },
18     {timestamps: true}
19 );
```

Listing 11 – Schéma mongoose assistant

Lorsqu'un schéma comme celui-ci est défini, il faut le compiler pour que cela devienne un modèle. Un modèle est en fait une classe depuis laquelle des documents peuvent être construits et dont les propriétés et le comportement suivent le schéma correspondant. Certaines fonctions, telles que *findById()* ou *save()* dans les routages, sont des méthodes de la librairie Mongoose. Pour les utiliser, il suffit de préciser à quel modèle la fonction doit être associée.

5
Programmation du client

5.1. Présentation générale

Le client correspond à l'interface utilisateur de l'application et s'occupe d'envoyer les requêtes au serveur qui va lui répondre. La technologie servie pour programmer cette partie est Vue.js qui est un framework JavaScript. Tout comme pour la partie serveur, l'ensemble du code est accessible sur le repository GitHub à l'adresse `https://github.com/SophieLydia/TB_Client` et, pour les mêmes raisons, le dossier *node_modules* ne s'y trouve pas. Pour que le client puisse communiquer avec le serveur, il doit pouvoir lancer des requêtes. Axios sert à cela et est développé dans la dernière section de ce chapitre.

Concernant la composition structurelle du client, les fichiers codes sont séparés en plusieurs dossiers, dont les dossiers *assets*, *components* et *services* qui seront vus plus profondément. Le dossier *assets* ne comporte que deux éléments, un fichier CSS qui décrit certains aspects esthétiques de la présentation des différentes pages de l'interface utilisateur et une image, la seule qui est utilisée sur l'interface. Le dossier *components* quant à lui contient les fichiers correspondant aux pages de l'interface. Un fichier étant l'équivalent d'une page, ils sont classés en fonction de l'objet associé. Par exemple, dans le sous-dossier *course*, il y a quatre fichiers, dont un correspondant à la page de création d'un cours, un correspondant à la page de modification d'un cours, un correspondant à la page de visualisation de la liste de tous les cours et un correspondant à la visualisation d'un cours en particulier. Tous les sous-dossiers sont construits de cette façon, à l'exception du sous-dossier *person* qui contient plus de fichiers puisqu'il rassemble en un endroit ce qui porte sur les chefs, les enfants et les assistants. Ensuite, le dossier *services* permet d'envoyer les différentes requêtes vers l'API et de réceptionner les réponses. Il y a un fichier par objet et chaque fichier s'occupe de la gestion des différentes fonctions qui y sont relatives à l'exception des objets chef, enfant et assistant qui sont à nouveau regroupés en un seul. Ces différents services sont utilisés par les fichiers du dossier *components* afin d'afficher les résultats et informations souhaités par l'utilisateur.

Finalement, un dernier fichier portant l'extension .vue qui ne se trouve dans aucun de ces dossiers, mais à leur source, est présent. Il sert à la mise en place d'une barre de navigation étant donnée que cette dernière est présente constamment sur n'importe laquelle des pages de l'application. Toujours à la source de ces différents dossiers, trois fichier JavaScript sont présents. Le premier, *http-common.js*, permet de créer une instance d'Axios, par conséquent son contenu est expliqué dans la section le concernant. Le deuxième, *main.js*, sert à créer une instance de Vue et par conséquent à initialiser l'application Vue. Le dernier, *router.js*, fait le lien entre le chemin lisible dans la barre d'adresse et le composant, donc la page, à afficher. Par exemple, `http://localhost:8080/teachings/606e03299ef6824bb414f16e` correspond au chemin */teachings/{id}* et renvoie à *./components/teaching/UpdateTeaching*. Ce dernier est le composant *UpdateTeaching* qui se trouve dans le sous-dossier *teachings* qui lui-même se trouve dans le dossier *components*.

5.2. Vue.js

Le framework JavaScript Vue.js [20] est conçu justement pour créer des interfaces utilisateur. Il est relativement simple d'utilisation, car la création d'une application se fait via la création de composants. Un composant est en fait une entité possédant ses propres attributs et sa propre logique. Grâce à ces composants, une grande modularité est présente dans la construction de l'interface. Il est en effet facile d'ajouter, de supprimer, de modifier ou même de réutiliser des composants. Un composant est généralement divisé en trois parties : le template (HTML), le script (JS) et le style (CSS).

Les Listings 12, 13 et 14 forment ensemble un composant. Ces derniers représentent la page d'ajout d'un thème. La Listing 12 concerne la partie template et est délimitée par les balises du même nom visible aux lignes 1 et 24. Le template est basé sur une syntaxe HTML et définit la manière dont est construite la page. HTML permet en partie de donner aux éléments un attribut *class*. Cet attribut, qui est très présent, offre la possibilité d'être utilisé par le style CSS et le script JavaScript dans les deux autres parties du composant. Il est donc facile de dire au programme quelle partie traiter et comment. La page contient un titre engendré par la ligne 4 suivi directement par un formulaire défini entre les balises *form* aux lignes 5 et 21. Ce formulaire est composé de deux champs d'entrée pour que l'utilisateur puisse donner les informations nécessaires à la création d'un thème et d'un bouton pour que l'utilisateur puisse soumettre le formulaire et donc puisse effectuer la création du thème. Hormis un titre et un formulaire, rien d'autre ne compose cette page si ce n'est la barre de navigation commune à toutes les pages.

```
1  <template>
2      <div class="row justify-content-center">
3          <div class="col-sm-6">
4              <h3 class="text-center">Add Theme</h3>
5              <form @submit.prevent="handleSubmitForm">
6                  <div class="form-group">
7                      <label>Title</label>
8                      <input type="text" class="form-control" v-model="theme.title"
                           required>
9                  </div>
10                 <div class="form-group">
11                     <label>Description</label>
```

```
12                  <input type="text" class="form-control" v-model="theme.description"
                        required>
13              </div>
14              <div class="row justify-content-center">
15                  <div class=" col-sm-auto">
16                      <div class="form-group">
17                          <button class="btn btn-add">Add</button>
18                      </div>
19                  </div>
20              </div>
21          </form>
22      </div>
23  </div>
24 </template>
```

Listing 12 – Composant Vue d'ajout d'un thème partie template

Concernant la deuxième partie sur le script, il faut se référer à la Listing 13 ci-dessous. Cette partie est donc du code JavaScript entouré des balises *script* aux lignes 1 et 20 comme délimitateur. C'est dans cette section que les différentes propriétés et comportements du composant sont initialisés et décrits. La propriété *data* a la particularité d'être toujours une fonction. Dans ce cas, elle retourne un objet *theme* qui contient un attribut *titre* initialement vide et un attribut *description* initialement vide aussi. Dans cet exemple, il y a aussi la présence de la propriété *methods* qui permet, comme son nom l'indique, de définir des méthodes. Une seule fonction est présente à la ligne 13 et s'appelle *handleSubmitForm()*. Cette fonction s'exécute une fois qu'elle est appelée et c'est le cas lorsqu'un utilisateur clique sur le bouton d'ajout et donc soumet le formulaire avec la ligne 5 de la Listing 12. Cette méthode va utiliser un service permettant de faire le lien avec le serveur et permettant de lui envoyer une requête lui demandant d'ajouter l'objet *theme*. En effet, une fois que l'utilisateur a correctement rempli le formulaire, les attributs de *theme* ne sont plus vides et ont pris les valeurs qui lui ont été données à l'aide dudit formulaire. Finalement, l'utilisateur est automatiquement redirigé à l'adresse */themes/list* grâce à la ligne 15. En regardant dans le fichier *router.js*, qui comme dit précédemment fait le lien entre l'adresse et la page à afficher, c'est le composant *ThemesList* qui est désormais mis à l'écran.

Il existe aussi la propriété *created* qui est une fonction tout comme *data*. Elle n'est pas présente dans ce composant, mais dans plusieurs autres comme le composant *ThemesList*. Cette fonction est appelée juste après l'instanciation de la page en question et sert à récupérer des données venant du serveur afin de les assigner à des propriétés du composant, généralement celles contenues dans la fonction *data*. C'est donc particulièrement utile lorsqu'une page décrit la liste d'un objet, par exemple la liste des thèmes, puisqu'elle va directement lancer la requête nécessaire auprès du serveur pour récupérer ce qu'il faut afin d'afficher les données voulues.

```
1 <script>
2     import Theme from "@/services/ThemeService"
3     export default {
4         data() {
5             return {
6                 theme: {
7                     title: '',
8                     description: ''
9                 }
```

```
10            }
11        },
12        methods: {
13            handleSubmitForm() {
14                Theme.create(this.theme).then(() => {
15                    this.$router.push('/themes/list');
16                });
17            }
18        }
19    }
20 </script>
```

Listing 13 – Composant Vue d'ajout d'un thème partie script

La Listing 14 concerne donc la troisième et dernière partie d'un composant sur le style. Tout ce qui se trouve entre les balises *style*, ici entre les lignes 1 à 9, sera du CSS. Cela sert à rendre plus esthétique et à afficher d'une certaine façon les différents éléments HTML d'une page. Lorsque l'attribut *scoped* est joint à la balise *style*, comme c'est le cas ici, cela signifie que le CSS ne s'applique qu'au composant en question. Uniquement de petits ajouts de style ont été effectués dans les composants puisque la majorité est commune à tous les composants et est donc réunie dans le fichier *global.css* du dossier *assets*.

La syntaxe de CSS est la suivante. Tout d'abord un sélecteur est choisi, c'est-à-dire l'élément HTML à styliser. Ensuite entre accolades, il y a la ou les propriétés à modifier suivant la ou les valeurs à prendre pour ces propriétés. A la ligne 2, le sélecteur est tous les éléments de la page ayant *btn* comme valeur pour l'attribut *class*, autrement dit, tous les boutons. Concernant cet élément, la propriété spécifiée est *margin-top* à la ligne 3, ce qui veut dire qu'un espace est créé en haut de l'élément en question. Finalement, la valeur de cet espace est de 3 pixels. Pour ce sélecteur-là, aucun autre style n'est ajouté localement.

```
1 <style scoped>
2     .btn {
3         margin-top: 30px;
4     }
5     h3 {
6         margin-top: 20px;
7         margin-bottom: 20px;
8     }
9 </style>
```

Listing 14 – Composant Vue d'ajout d'un thème partie style

Comme déjà dit, un composant peut utiliser un service et chaque service utilise Axios pour communiquer avec le serveur. Cette partie est donc discutée dans la section suivante concernant Axios.

5.3. Axios

Axios [3] sert à effectuer des requêtes HTTP vers des *endpoints* d'une API. C'est une librairie qui peut être utilisée dans une simple application JavaScript ou avec un framework particulier comme Vue.js.

Le fichier *http-common.js* sert à créer une instance d'Axios en fonction d'une certaine configuration. Dans cette configuration, il est précisé que le nom de domaine du serveur est `http://localhost:8000`. Il est également précisé dans le *headers* que l'application fonctionne avec des objets JSON.

Pour revenir au service, un fichier de ce type contient plusieurs fonctions effectuant diverses requêtes. Par exemple, dans le fichier *ThemeService*, il y a sept fonctions différentes : une obtient tous les thèmes, une obtient un thème en particulier suivant l'identifiant donné, une obtient uniquement les titres de tous les thèmes, une crée un thème, une met à jour un thème, une supprime un thème et une retourne tous les thèmes suivant un titre donné. Avec la Listing 15, cela correspond au premier cas où tous les thèmes sont retournés. A la ligne 3, *http* correspond à une instance d'Axios. Elle est suivie par la méthode HTTP à effectuer, dans ce cas GET, puis prend comme paramètre le chemin.

```
1 async getAll() {
2    try{
3        const res = await http.get("/themes");
4        return res.data;
5    }catch(err){
6        return false;
7    }
8 }
```

Listing 15 – Service retournant tous les thèmes

Dans ce cas, il n'y a pas d'autres paramètres, mais il arrive qu'il faille que certaines données soient également passées en paramètre. Lorsqu'une requête de création d'un thème est envoyée par exemple, il faut donner un titre et une description qui sont encapsulés dans un objet au préalable et correspond au paramètre *data* dans la Listing 16.

```
1  async create(data) {
2    try{
3      const res = await http.post("/themes", data);
4      return res.data;
5    }catch(err){
6      return false;
7    }
8  }
```

Listing 16 – Service créant un thème

Tous les autres services sont structurés de cette façon, c'est-à-dire un ensemble de fonctions asynchrones effectuant une requête GET, POST, PATCH ou DELETE auprès d'un *endpoint* de l'API du serveur et avec comme paramètres un chemin, qui peut-être créé dynamiquement dans le cas où un identifiant ou un filtre est nécessaire, et des données supplémentaires pour la création ou la modification d'un objet.

6 Conclusion

6.1. Résultats

En résumé, dans ce travail, il a été fait un rapport décrivant la théorie et l'analyse liées à une application et une application client-serveur qui sert à l'organisation de cours extra-scolaires intercantonaux. Une fois l'analyse des besoins effectuée, il est nécessaire de trouver une structure des données et de leurs relations ainsi que de choisir les technologies à adopter. Une base de donnée NoSQL, MongoDB, a été mise en oeuvre pour cela ainsi qu'une architecture REST avec un serveur utilisant Node.js et un client utilisant Vue.js.

La totalité des objectifs n'a malheureusement pas été atteinte, car l'authentification n'a pas été mise en place et certaines manipulations n'ont pas été sécurisées. En exemple, un enfant ne peut pas être créé si la classe qui lui sera attribuée n'est pas au préalable existante. En effet, la création de l'enfant, ou du rôle spécifique de la personne de manière générale, ne peut pas être faite après coup dans l'état actuel de l'application. Un autre exemple, la suppression de n'importe quel élément ne vérifie pas si cet élément n'est pas présent ailleurs. Si un élément est un attribut d'un autre élément, sa valeur sera tout simplement nulle après sa suppression. Malgré ces quelques points, la grande majorité des fonctionnalités désirées a bien été réalisée, fonctionne correctement et est présente dans l'application finale.

6.2. Améliorations possibles

Cette application telle quelle ne peut pas être déployée et nécessite encore du travail pour le devenir. A commencer par les points précisés dans la section précédente avec l'authentification utilisateur et une meilleure gestion de certaines manipulations qu'il faut sécuriser. Elle peut également évoluer avec l'ajout des documents de cours directement sous forme de PDF et téléchargeables.

Un autre aspect intéressant pourrait être l'utilisation de l'application par des personnes n'ayant pas de compte, comme les enfants qui suivent les cours ou leurs parents. Ces différentes pages seraient donc conçues pour des visiteurs quelconques. Les pages contenant des données personnelles ne seraient pas accessibles, mais des pages contenant différents cours avec les dates, les lieux et les classes seraient intéressantes et pratiques à avoir. Une page décrivant le principe des cours, les informations importantes et les informations de contact, telles qu'une adresse mail en cas de question d'un parent, pourrait directement

être intégrée. L'intégration d'un formulaire d'inscription d'un enfant offrirait la possibilité aux parents d'inscrire directement leur enfant. L'automatisation de la création de l'enfant dans le système serait également une continuité de ce dernier cas. Cela libérerait du temps aux chefs, car ils ne devraient plus le faire manuellement.

D'autres possibilités d'évolution et d'amélioration sont toujours imaginables, mais pour garder l'idée initiale du travail les plus importantes ont été citées.

6.3. Bilan personnel et remerciements

Ce travail m'a appris énormément de choses, principalement la programmation à l'aide des technologies Node.js et Vue.js qui m'étaient jusque-là inconnues. Je n'avais jamais créé d'application client-serveur, ni même uniquement un serveur ou uniquement un client de sa conception jusqu'à son fonctionnement. C'était très enrichissant de voir les multiples possibilités de création et d'obtenir une application créée sur mesure en fonction de ce que je souhaitais intégrer. Avec la partie programmation, l'outil "inspecteur" et l'affichage des bugs dans la console de mon environnement de développement m'ont été d'une aide infinie pour comprendre des erreurs de toutes sortes. Le site `https://stackoverflow.com/` m'a également fait comprendre beaucoup d'erreurs et m'a aidé à comprendre le fond de certains problèmes. Comme ils le disent si bien eux-mêmes : "*Every developer [...] has a tab open to Stack Overflow*". La mise en place d'une base de données était également nouvelle pour moi et MongoDB m'a donné quelques difficultés, puisque je n'étais pas familière avec ce genre de système de gestion.

L'aspect d'analyse m'a aussi beaucoup plu et n'est pas à mettre de côté, car commencer à coder sur la base d'une mauvaise analyse ne va rien amener de bon. En effet, si l'idée des structures et des fonctionnalités n'est pas bonne et clairement définie, des bouts de codes seront très probablement inutiles et il faudra dans tous les cas effectuer des modifications si ce n'est tout revoir. Heureusement, j'ai dû uniquement apporter quelques petites modifications à l'analyse de base, ce qui n'a pas beaucoup impacté mon code.

Je clos ce travail en tenant à remercier le Prof. Dr. Jacques Pasquier-Rocha qui m'a suivi tout au long de ce travail, qui a répondu à mes questions et m'a aidé à garder le cap. Je tiens également à remercier M. Pascal Gremaud pour avoir répondu à mes questions plus techniques et m'avoir aidé avec le choix de certaines technologies. Finalement je remercie également ma famille, mes amis et mon copain pour m'avoir épaulé, conseillé et aidé à la relecture du travail.

A
Annexes

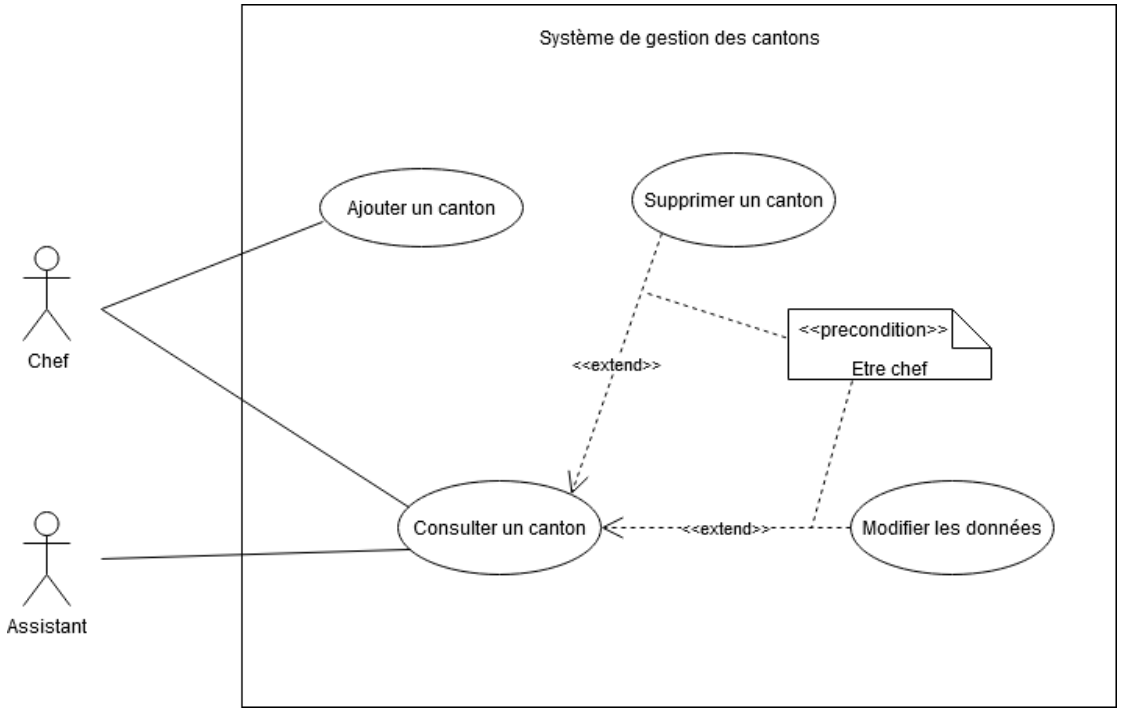

Figure 19. – Diagramme UML de cas d'utilisation du système de gestion des cantons

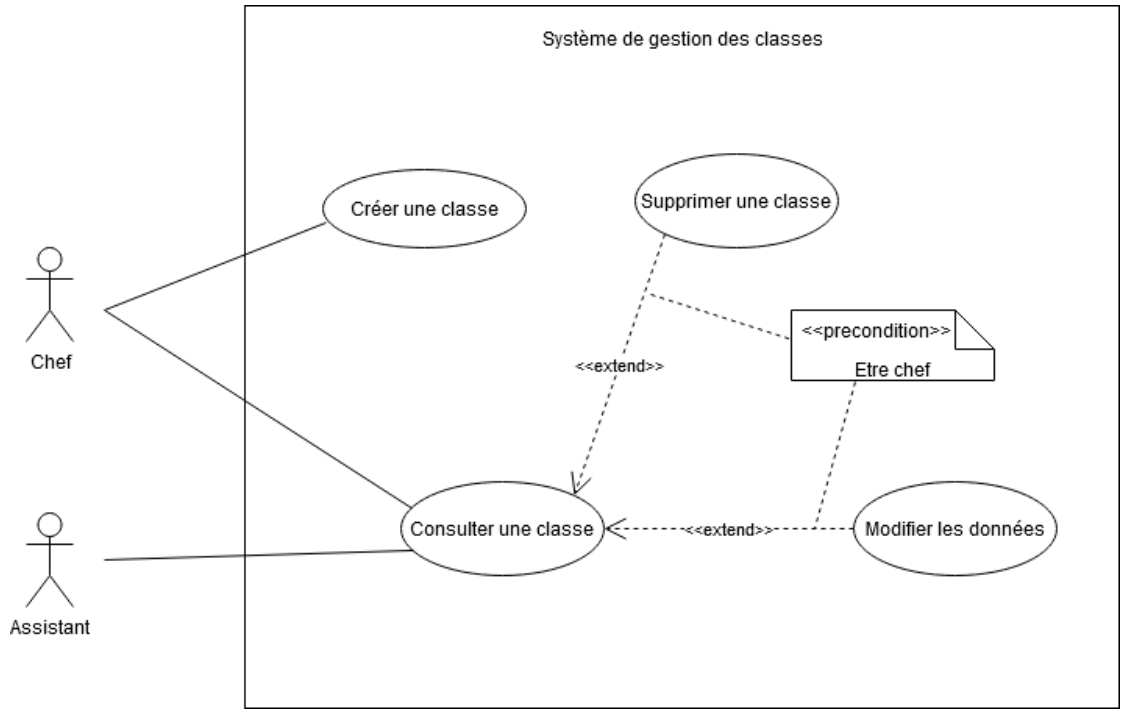

Figure 20. – Diagramme UML de cas d'utilisation du système de gestion des classes

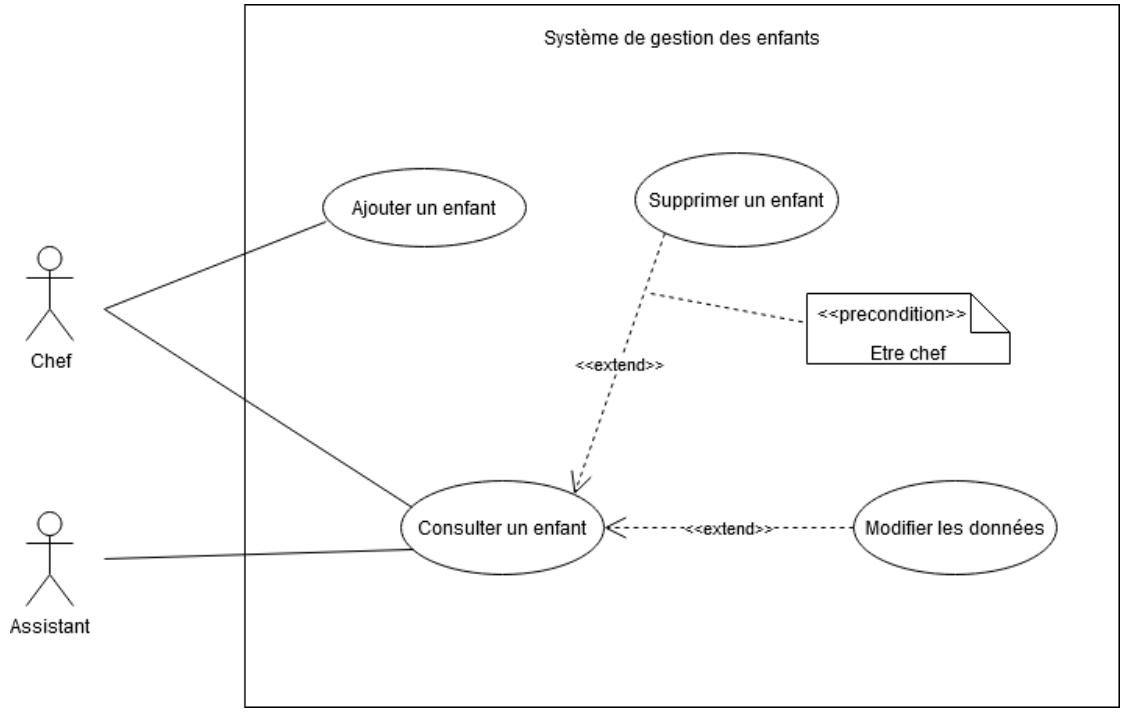

Figure 21. – Diagramme UML de cas d'utilisation du système de gestion des enfants

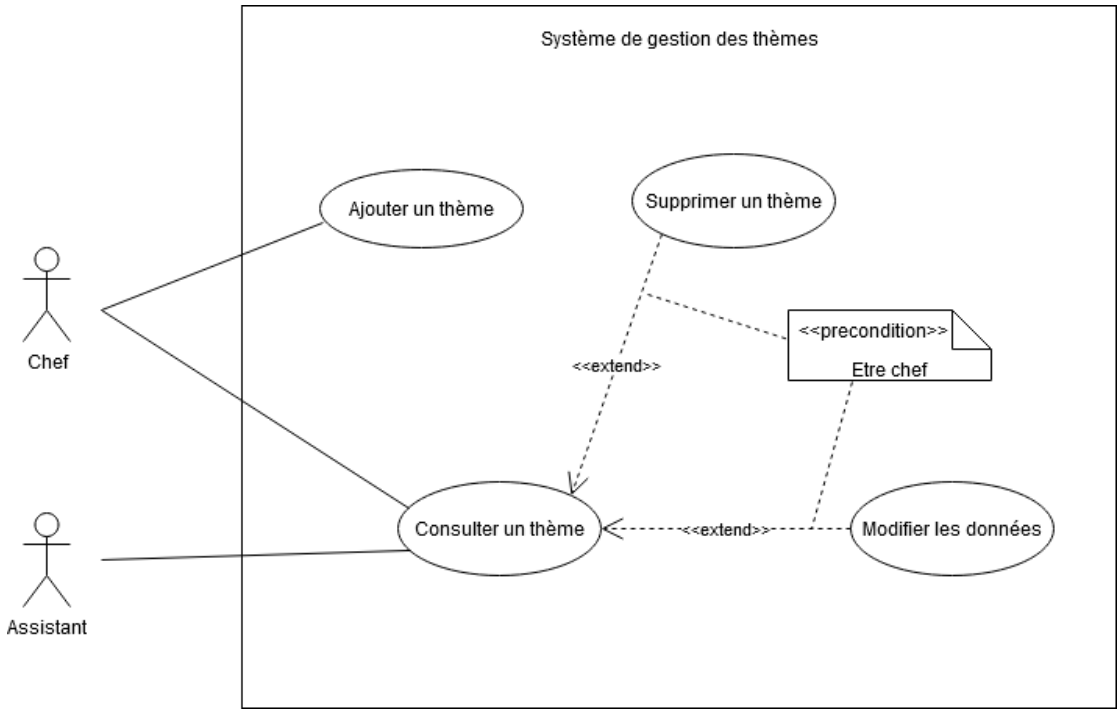

Figure 22. – Diagramme UML de cas d'utilisation du système de gestion des thèmes

Bibliographie

[1] Adoptez les API REST pour vos projets web. https://openclassrooms.com/fr/courses/6573181-adoptez-les-api-rest-pour-vos-projets-web (dernière consultation le 03 septembre, 2021). 24

[2] API Endpoints - What Are They ? Why Do They Matter ? https://smartbear.com/learn/performance-monitoring/api-endpoints/ (dernière consultation le 03 septembre, 2021). 24

[3] Axios. https://axios-http.com/ (dernière consultation le 06 septembre, 2021). 33

[4] Build A Restful Api With Node.js Express MongoDB | Rest Api Tutorial. https://www.youtube.com/watch?v=vjf774RKrLc (dernière consultation le 05 août, 2021).

[5] Comment fonctionne un Web Service - Qu'est-ce qu'un Web Service ? https://www.oracle.com/fr/cloud/definition-web-service/ (dernière consultation le 31 août, 2021). 23

[6] Différence entre API et Web service. https://waytolearnx.com/2018/11/difference-entre-api-et-web-service.html (dernière consultation le 03 septembre, 2021). 24

[7] Express.js. https://expressjs.com/fr/ (dernière consultation le 27 août, 2021). 17, 19

[8] Fonctions Synchrones VS Asynchrones. https://adrienjoly.com/cours-nodejs/sync-vs-async.html (dernière consultation le 26 août, 2021). 17

[9] How to add Swagger UI to an existing Node.js and Express.js project. https://levelup.gitconnected.com/how-to-add-swagger-ui-to-existing-node-js-and-express-js-project-2c8bad9364ce (dernière consultation le 03 septembre, 2021). 24

[10] Learn REST : A RESTful Tutorial. https://www.restapitutorial.com/ (dernière consultation le 31 août, 2021). 24

[11] L'asynchrone en JavaScript. https://www.pierre-giraud.com/javascript-apprendre-coder-cours/introduction-asynchrone/ (dernière consultation le 26 août, 2021).

[12] MongoDB. https://www.mongodb.com/fr-fr (dernière consultation le 31 août, 2021). 28

[13] Mongoose. https://mongoosejs.com/ (dernière consultation le 31 août, 2021). 28

[14] Node.js. https://nodejs.org/en/ (dernière consultation le 23 août, 2021). 17

[15] Qu'est-ce qu'une API ? https://www.redhat.com/fr/topics/api/what-are-application-programming-interfaces (dernière consultation le 03 septembre, 2021). 24

[16] Representational State Transfer (REST). https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (dernière consultation le 31 août, 2021). 24

[17] Stack Overflow. https://stackoverflow.com/ (dernière consultation le 05 septembre, 2021).

[18] Swagger. https://swagger.io/ (dernière consultation le 03 septembre, 2021). 24

[19] Une introduction à Mongoose pour MongoDB et Node.js. https://fr.accentsconagua.com/articles/code/an-introduction-to-mongoose-for-mongodb-and-node-js.html (dernière consultation le 23 août, 2021). 28

[20] Vue.js. https://vuejs.org/ (dernière consultation le 05 septembre, 2021). 31

[21] What is REST. https://restfulapi.net/ (dernière consultation le 31 août, 2021). 24