



## Table des matières

1. Introduction .....	4
2. Présentation de l'entreprise : [Titre perso] .....	5
2.1. Amiltone .....	5
2.1.1. Chiffres clés .....	5
2.1.2. La transformation digitale .....	5
2.1.3. Les Factories.....	6
2.2. Bluck Studio .....	8
2.3. Data New Road.....	9
2.3.1. Des projets ambitieux.....	9
2.4. Mon point de vue.....	10
2.4.1. Une entreprise « jeune » .....	10
<b>2.4.2. Une entreprise impliquée envers ses collaborateurs</b> .....	10
2.5. Mon alternance .....	11
2.5.1. L'équipe sur le projet.....	11
2.5.2. Utilisation de la méthode Agile .....	12
2.5.3. Mes missions .....	13
3. Analyse du contexte : Le projet AmilApp .....	14
3.1. AmilApp.....	14
3.2. Firebase .....	16
3.2.1. Pourquoi l'avoir choisi ? .....	16
3.2.2. Avantages et inconvénients de Firebase .....	16
3.3. Un projet basé sur l'innovation technologiques.....	17
3.3.1. Préparé pour le cloud .....	17
<b>3.4. Basé sur le partage de connaissances</b> .....	19
3.5. Analyse personnelle du contexte de stage de fin d'étude .....	19
4. Problématique : Comment mettre en place une solution web réutilisable, modulaire et évolutive ? .....	20
4.1. Le démarrage d'un projet, une perte de temps ? .....	20

4.2.	Les contraintes de l'architecture monolithique .....	20
4.3.	Les micro-services .....	21
4.3.1.	Définition .....	21
4.3.2.	Pourquoi les avoir choisis ?.....	22
4.4.	Le starter-kit .....	24
4.4.1.	Définition .....	24
4.4.2.	Avantages .....	24
5.	Méthodes habituellement utilisées pour une situation présentant des similitudes : [Titre perso] .....	25
5.1.	Spring Boot .....	25
5.1.1.	Fonctionnalités .....	25
5.1.2.	Avantages et inconvénients par rapport au projet.....	25
5.2.	Hackathon-starter.....	26
5.2.1.	Fonctionnalités .....	26
5.2.2.	Avantages et inconvénients par rapport au projet.....	27
5.3.	Autres solutions.....	27
6.	Exposé des décisions prises et des interventions menées par le stagiaire pour résoudre le problème : [Titre perso].....	28
6.1.	Objectifs et contraintes du projet .....	28
6.2.	La mise en place de l'architecture .....	28
6.2.1.	Le choix de l'API Gateway.....	28
6.2.2.	La communication « interservices » .....	32
6.2.3.	Le modèle CQRS .....	35
6.3.	De l'ancien starter-kit aux micro-services .....	36
6.3.1.	Un nouveau framework : NestJS.....	36
6.3.2.	La séparation en « packages » .....	39
6.4.	Le module de connexion .....	41
7.	Démonstration d'une originalité dans l'élaboration et la mise en œuvre de la solution : [Titre perso] .....	43
7.1.	Une solution modulable et évolutive .....	43

7.2.	Des équipes indépendantes .....	44
7.3.	Une interface en ligne de commande ? .....	44
8.	Analyse de l'approche choisie : [Titre perso] .....	44
8.1.	Résultats obtenus .....	44
8.2.	Analyse du champ d'application de la solution élaborée.....	44
8.3.	Mise en perspective avec d'autres contextes .....	44
9.	Réflexion sur le stage et le mémoire : [Titre perso].....	45
9.1.	Auto-évaluation du travail réalisé.....	45
9.2.	Bilan des acquis sur les aspects techniques, stratégiques et managériaux .....	46
9.3.	Perspectives professionnelles en relation avec les compétences acquises.....	46
10.	Conclusion.....	46

## 1. Introduction

## 2. Présentation de l'entreprise : [Titre perso]

### 2.1. Amiltone

Amiltone est une société de **services numériques** (ESN) de taille humaine. L'écosystème d'expertise de la **Digital Factory**, que j'expliquerai un peu plus tard, permet à Amiltone d'accompagner ses clients dans leurs problématiques de développement et de compétitivité grâce à une approche globale des systèmes d'informations. Combinant la valeur ajoutée, l'innovation et la performance des services fournis, Amiltone propose plusieurs solutions : le conseil, l'intégration de systèmes et le développement de solutions métiers adaptées aux besoins spécifiques de chacun de ses clients.

L'entreprise recrute en permanence mais a à cœur de garder une croissance organique, c'est-à-dire qu'elle grandit sans racheter d'autres entreprises et sans expansion à l'international. Les dirigeants souhaitent garder une certaine proximité avec leurs employés et c'est pour cela que chaque nouveau collaborateur rencontre le directeur ainsi que le président d'Amiltone.

Ajouter l'organigramme, les clients et les partenaires

#### 2.1.1. Chiffres clés

Amiltone a été fondée en 2012 à Lyon, où se trouve actuellement le siège social. L'entreprise comptabilise 270 employés en 2019 et a pour objectif d'atteindre les 300 collaborateurs en 2020. Le chiffre d'affaire de l'entreprise était de 2.7 millions d'euros en 2014 et était de 16 millions d'euros en 2019. Aujourd'hui, Amiltone c'est sept agences en France : Lyon, Paris, Niort, Nantes, Bordeaux, Grenoble et Aix-en-Provence. Chacune de ces agences a ses spécificités dans les technologies utilisées, elles sont en général adaptées au mieux à la région. L'entreprise ne souhaite pas se déployer à Paris car c'est un secteur trop concurrentiel.

#### 2.1.2. La transformation digitale

La transformation digitale, gérée chez Amiltone par la Digital Factory que je présente ci-dessous, est devenue un enjeu déterminant pour le secteur du commerce dans son ensemble. Parfois appelée transformation numérique, la transformation digitale désigne le processus qui

permet à une entreprise de comprendre et d'incorporer au maximum les technologies digitales dans l'ensemble de ses activités.

### 2.1.3. Les Factories

Amiltone a créé plusieurs pôles nommés Factory qui se focalisent sur des technologies précises.

#### ♦ *La Digital Factory*

La Digital Factory est le pôle d'expertise qui accompagne les clients d'Amiltone dans leur transformation digitale. L'objectif de cette entité est d'analyser l'entreprise, le fonctionnement et les outils déjà en place chez le client pour proposer une solution adéquate, stable et évolutive permettant de répondre aux besoins et d'anticiper les évolutions et les changements futurs, des grands groupes comme des petites structures pour que leurs systèmes d'information deviennent l'un des moteurs de leur croissance.

#### ♦ *La Mobile Factory*

La Mobile Factory est l'équipe experte en développement d'applications mobiles natives sur **Android et iOS**. Amiltone fait le choix de développer en langage natif. En effet, seule une application développée nativement permet d'obtenir une vitesse d'exécution optimale sur n'importe quel terminal. La réactivité est l'atout majeur du natif sur tous les autres types de développement comme le web embarqué ou le cross-platform.

Le natif permet également d'utiliser l'ensemble des éléments d'interface et d'ergonomie propres au système tout en assurant leur évolutivité. L'utilisateur retrouvera, selon son système, une interface et des usages qu'il maîtrise et qu'il retrouve dans les applications livrées de base avec son smartphone.

À noter que plusieurs fonctionnalités ne sont pour l'instant réalisables qu'avec du code natif et que maintenir une application hybride avec des parties de code natif augmente considérablement la charge de travail.

La maintenabilité de l'application à moyen et long terme est donc plus sûre et plus simple en utilisant le natif, puisque l'évolution conjointe des Software Development Kit (SDK) et du système permet d'utiliser directement l'ensemble des nouveaux éléments.

Le choix des applications cross-platform ajoute également des risques supplémentaires, comme l'arrêt du support de la technologie utilisée, ou un retard de mise à jour par rapport aux dernières versions des systèmes mobiles, ces derniers évoluant très rapidement.

Partir sur du natif permet d'assurer la longévité de l'application et une maintenance simplifiée.

Amiltone estime que le coût initial d'un projet, bien que plus long à développer en phase de conception, rejoint sur le moyen terme avec la maintenance et les évolutions le coût d'un projet cross-platform.

### ◆ *La Web Factory*

La Web Factory est la pierre angulaire de l'offre globale d'Amiltone. En effet, l'utilisation de technologies dédiées à la création d'un backend d'administration des données permet de connecter entre toutes les briques d'un système d'information.

Pour chaque nouveau projet, un comité d'experts est chargé d'identifier les technologies Web les plus appropriées. Les réponses aux questions de langages et de base de données permettent de créer une fondation solide basée sur des technologies adaptées et spécifiquement sélectionnées pour leur intérêt sur un sujet spécifique.

En outre, l'expertise de la Web Factory ne se limite pas au développement d'un système qui répond parfaitement à un besoin métier. Son rôle est également de respecter les normes de développement afin d'améliorer le référencement naturel, l'interconnexion simplifiée avec les API ou l'intégration de standards de communication multi-canaux tels que Facebook ou Twitter.

### ◆ *La Data Factory*

La Data Factory permet de répondre à une problématique commune à de nombreux clients d'Amiltone, quels que soient leur taille et leur champ d'activité : comment gérer leurs données ? Cette question complexe doit être résolue à plusieurs niveaux. Tout d'abord, le but de la Data Factory est de comprendre les activités du client afin de déterminer avec lui les indicateurs pertinents qui lui permettront de contrôler plus facilement son activité. La deuxième étape consiste à extraire des données pour répondre à ce besoin. Très souvent, les données sont stockées à différents emplacements du Service Informatique (SI) et sur différentes technologies (Base de données, email, Excel par exemple). L'étape suivante consiste à consolider ces données dans un Datawarehouse, avec éventuellement l'ajout de Datamart pour différencier plusieurs secteurs, pour permettre leur utilisation simple et rapide dans la dernière étape qui est l'affichage d'indicateurs pertinents et visuellement percutants pour que les décideurs puissent réagir en un coup d'œil.

La Data Factory offre également des services de surveillance de log serveurs et une surveillance complète d'un flux métier.

Afin d'aller toujours plus loin dans l'analyse des données, la Data Factory est accompagnée par un expert de renommée internationale spécialisé dans l'intégration de briques algorithmiques pour des usages tels que l'analyse prédictive ou le Machine Learning.

## ◆ Expertise et qualité

La qualité logicielle et l'expertise technique étant les principaux axes de la stratégie d'Amiltone, la Digital Factory a mis en place une veille particulière sur des problématiques telles que :

- La maintenabilité du code
- L'objectif zéro défaut
- Le contrôle du cycle de vie du logiciel
- L'amélioration continue
- L'organisation projet

Les équipes techniques sont impliquées dans ce processus de veille et scrutent en permanence les outils et techniques de demain afin d'améliorer la qualité des projets et la satisfaction des clients.

## 2.2. Bluck Studio

Depuis le début, Amiltone a toujours proposé à ses clients des services et des conseils sur le design et l'ergonomie (Interface Utilisateur, Expérience Utilisateur) sur leurs applications pour améliorer l'expérience utilisateur. L'entreprise s'est rendu compte que cette étape de conception de maquettes devenait de plus en plus importante et a alors décidé d'agrandir l'équipe design et de créer récemment **Bluck Studio**.

Bluck Studio a donc été créé en 2019 et compte actuellement 8 collaborateurs plus un ingénieur d'affaires. Le studio n'est pas une entreprise mais plutôt une branche d'Amiltone comme le sont les différentes Factories. Le studio est divisé en 3 pôles différents, le pôle **UI/UX**, le pôle **Illustration** et le pôle **Communication**.

**Le pôle UI/UX** : réalise les différentes interfaces web et mobile tout en répondant aux contraintes du client.

**Le pôle Illustration** : la partie création du studio, il réalise les différents affichages, illustrations ou vidéos d'Amiltone.

**Le pôle communication** : s'occupe de la communication interne d'Amiltone ainsi que la communication de l'entreprise sur les réseaux sociaux. Il réalise aussi les événements interne ainsi que la préparation des événements externes, comme le SIDO (Salon des objets connectés).

Le studio a été créé dans le but de répondre aux clients qui souhaitaient n'avoir que des maquettes. Mais Bluck Studio reste avant tout le studio design d'Amiltone et collabore sur tous les projets de l'entreprise.



Bluck Studio travaille actuellement avec un client sur une application mobile permettant de commander des recharges de batterie pour les voitures électriques. Le pôle UI/UX travaille sur les différentes contraintes d'accessibilité de la solution.

En parallèle, le studio est en train de concevoir ses propres lignes directrices graphiques pour rendre les applications internes plus cohérentes entre elles. La charte sera amené à être réutilisée sur les nouveaux projets internes mais aussi les anciens.

### 2.3. Data New Road

Data New Road (DNR) est le fruit d'une réflexion entre les entreprises Amiltone et APRR, un des plus gros partenaires d'Amiltone, et démarre son activité en 2019 après six mois de travail sur la prévision long terme et la construction des serveurs. L'objectif de DNR est de valoriser les données trafic via des algorithmes de dernières générations afin d'obtenir des prévisions et des analyses apportant une plus-value. L'équipe de développement est localisée dans les locaux d'Amiltone à Villeurbanne, tout proche de l'INSA avec qui a été signé un contrat de recherche.

Data New Road reste encore une petite entreprise avec pour Directeur Général Damien Corbi, quatre Data Scientist qui opèrent sur du traitement d'image, des réseaux de neurones et du Machine Learning, deux développeurs web pour le développement des différents Dashboard et API, un chef de projet et un ingénieur d'affaires. La plupart des collaborateurs de DNR viennent de chez Amiltone et APRR finance et utilise les projets.

#### 2.3.1. Des projets ambitieux

Je présente dans cette partie les différents projets sur lesquels travaille l'équipe de Data New Road.

##### ◆ Flow

Un des premiers sujets a été la prévision du trafic routier avec Flow. L'objectif est de prévoir le niveau de trafic sur les autoroutes sur le long terme, plus d'un an, et le court terme, de quinze minutes jusqu'à une heure par exemple, grâce à dix ans d'historique qu'APRR fournit à Data New Road, afin d'accompagner les gestionnaires et dirigeants publics et privés d'infrastructures routières dans leur processus de décision. Au départ, c'est le laboratoire INSAVALOR, spécialisé dans l'analyse de données, qui a créé l'algorithme pour la prévision à long terme.

Avant de vouloir utiliser Flow, APRR utilisait une autre application qui provient de l'entreprise Phoenix ISI, qui est toute seule sur le marché français dans ce domaine. Flow entre donc en concurrence avec cette entreprise et a vraisemblablement un tableau de bord plus complet, plus ergonomique et plus intuitif. Le but ici est de faire en sorte que ATMB, un autre gros client

et partenaire d'Amiltone, utilise Flow et qu'ainsi, avec une action en chaîne, Amiltone récupère le marché.

### ◆ Safe

Le deuxième projet le plus avancé de Data New Road est **Safe**, un outil qui va analyser les données des accidents sur les autoroutes, toujours grâce à l'historique d'APRR, pour mettre en avant les sections les plus dangereuses et les plus accidentogènes en fonction de certains critères sélectionnés.

### ◆ Classif

**Classif est un algorithme** qui va permettre de classer les différents véhicules qui traversent les péages. C'est un projet qu'APRR et Amiltone testent ensemble tout en sachant qu'ils ne sont pas en avance par rapport au marché. Potentiellement, seul APRR utiliserait cette application.

### ◆ Boards

**Boards un outil** qui analyse les données des différentes zones géographiques d'APRR sur lesquelles il peut y avoir une ou plusieurs autoroutes et qui va permettre de faire des simulations de coût en fonction d'un budget et de dépenses en cas de nouvel appel d'offres.

Ces deux derniers projets sont beaucoup moins avancés que Flow et Safe, ils sont prêts à être utilisés, mais sont encore en attente d'améliorations en partenariat avec APRR.

## 2.4. Mon point de vue

### 2.4.1. Une entreprise « jeune »

Amiltone est une entreprise assez **jeune** dans son état d'esprit et qui a à cœur d'innover dans les **nouvelles technologies**, ce qui m'a permis de découvrir et d'apprendre énormément. Effectivement, l'entreprise a créé différents départements, que j'ai déjà expliqué ci-dessus, qui utilisent des technologies différentes et j'ai pu en expérimenter plusieurs grâce à eux, tel que le Java, le C++ ou même le CUDA, un langage créé par NVIDIA optimisé pour le calcul sur carte graphique. Plus tard, je pourrais potentiellement m'essayer sur différents langages mobiles.

### 2.4.2. Une entreprise impliquée envers ses collaborateurs

Depuis mon arrivée chez Amiltone, j'ai toujours travaillé sur des projets internes. J'ai pu observer les différentes méthodes de travail en équipe et elles n'ont pas toujours été optimales. La priorité étant pour les projets clients, les projets internes étaient largement mis au second plan, qu'il s'agisse des maquettes ou des présentations. J'ai eu l'occasion de travailler sur une application sans chef de projet, sans référent technique et tous les collaborateurs qui

travaillaient avec moi, ainsi que moi-même, étions débutant sur les technologies utilisées. Nous nous faisions les merge request entre nous et finalement, nous ne pouvions pas réellement progresser puisque personne ne nous indiquait nos erreurs. Ce fut probablement le dernier projet sans référent technique, sans chef de projet.

Amiltone a su donner de l'importance aux applications internes et améliorer les différents process sur ces projets. Chacun d'eux est maintenant pris en charge par un chef de projet et un référent technique y est assigné pour traiter les revues de code.

## 2.5. Mon alternance

Cela fait maintenant deux années que je suis en alternance chez Amiltone à Lyon sur le projet AmilApp.

### 2.5.1. L'équipe sur le projet

Depuis le début de projet, un grand nombre de collaborateurs ont pu travailler sur AmilApp. Nous avons commencé à trois développeurs, **tous alternants**, chacun sur une plateforme différente : un sur iOS, le système d'exploitation de l'iPhone, un sur Android, celui de Google, et moi-même sur la partie administrateur de l'application, en web, et chacun supervisé par le responsable de chaque factory, pour relire nos merge request.

Une fois la première version d'AmilApp sortie, au bout d'un an, je me suis retrouvé seul sur le projet le temps de développer la partie utilisateur du projet sur le web. Puis, dès le mois de Novembre, l'idée de développer notre propre backend pour remplacer Firebase, que j'explique plus tard, florissait et je commençais alors à développer l'API avec un autre collaborateur.

L'équipe s'agrandit assez rapidement puisque beaucoup de nouveaux collaborateurs arrivent chez Amiltone, nous sommes sept sur le projet au mois de Juillet 2020 : deux développeurs sur le backend, deux autres sur les fronts utilisateur et administrateur, deux sur l'application Android et un dernier sur l'application iOS.

Dans l'ensemble, la taille de l'équipe varie beaucoup en fonction de la charge de travail à effectuer sur le projet et des missions **clientes**. En effet, tous les développeurs peuvent être envoyés en **mission** à n'importe quel moment, l'équipe d'AmilApp n'est jamais fixe. De mon point de vue, c'est une situation qui peut être parfois compliquée à gérer surtout que généralement, les personnes qui arrivent sur le projet sont débutantes sur les technologies utilisées. En effet, un autre projet s'est déroulé dans un cadre similaire : AmilNote, l'application de gestion des collaborateurs **destiné** aux ressources humaines. À cause du grand nombre de **développeur** ayant codé sur l'application, celle-ci est devenue difficilement maintenable et elle a été recodée depuis le début. **J'essaye** donc de faire en sorte que l'application reste le plus maintenable possible.

## 2.5.2. Utilisation de la méthode Agile

**La méthode Agile est une méthode de travail qui place le client au centre des priorités du projet.** À l'origine créée pour les projets de développement web et informatique, **elle** **aujourd'hui** de plus en plus utilisée car elle s'adapte à de nombreux type de projets, tous secteurs confondus.

Au début des années 2000, des experts logiciels se réunissent afin de mettre en commun leur méthode de travail et **créé** le « **Manifeste Agile** ». Une meilleure implication de la part du client et une plus grande **réactivité des équipes** face à ses demandes font partie des principes fondamentaux de la méthode Agile. Selon cette méthode, planifier tout un projet à l'avance serait contre-productif, et il est recommandé de se fixer des objectifs à court terme pour être plus réactif aux potentiels aléas qui peuvent survenir. Une fois l'objectif, on passe au suivant jusqu'à l'accomplissement de l'objectif final.

Malgré le fait qu'AmilApp soit un projet interne et qu'il n'y **a** pas réellement de client final, l'équipe essaye au maximum de fonctionner en mode **agile**. Les « **sprint** » sont renouvelés toutes les deux semaines afin de permettre une meilleure réactivité aux possibles évolutions et améliorations. Comme AmilApp est un projet interne, il n'y a **à priori** pas de deadline, mais plutôt des objectifs, comme par exemple, faire en sorte que l'application soit présentable à certains salons auxquels Amiltone a participé.

Plusieurs fois dans l'histoire du projet, de nouvelles évolutions conséquentes ont été demandées, comme de nouvelles pages sur le site web ou de nouveaux écrans sur les applications mobiles. **À chacune d'elle**, l'équipe **organisait** plusieurs réunions avec le chef de projet, les collaborateurs qui se sont occupés des maquettes et les développeurs, **qui nous ont permis de discuter des nouvelles maquettes, de ce qui était faisable ou non, de l'ergonomie de celles-ci, ou même parfois des fonctionnalités qui peuvent manquer de cohérence de temps en temps.**

D'autres réunions sont ensuite **organisées** pour que les développeurs puissent estimer les nouvelles tâches. Etant sur le projet depuis le début, **c'est moi qui** suis en charge de créer les différents tickets du projet web sur le tableau Jira.

De temps en temps, trop peu à mon avis, des revues de sprint sont organisées avec le « product owner », qui n'est autre que mon maître de stage Alexandre Buffy, pour lui présenter les avancées du projet. Mais effectivement, ces réunions sont surtout organisées lorsqu'il y a de grosses avancées sur le projet.

Le chef de projet permet ici d'harmoniser la communication entre les différentes équipes de développement ainsi que l'équipe en charge des maquettes et le « product owner ». **Mais l'équipe utilise l'outil** de communication professionnel Teams, créé par Microsoft, ce qui simplifie grandement la communication globale sur le projet.

### 2.5.3. Mes missions

Comme dit précédemment, ma mission sur le projet AmilApp **a été dans un premier temps de développer l'application web**. C'est dans cette année de développement que j'ai pu réellement progresser, grâce au développeur leader de la Web Factory qui a fait mes revues de code. Malgré le côté frustrant que ces revues de code peuvent avoir lorsqu'il y a beaucoup de choses qui ne vont pas techniquement dans le code, je me suis vite rendu compte que c'est grâce à celles-ci j'ai pu m'améliorer. Rapidement, après le début du projet, il n'y avait déjà plus beaucoup de code à corriger pendant les revues.

Une fois la première version d'AmilApp déployée, j'ai commencé le développement du front utilisateur web. J'étais alors seul sur le projet et j'ai pu prendre la décision de repartir de zéro afin de mettre à jour les différentes versions des paquets utilisés sur le projet. Effectivement, **j'ai trouvé** plus simple de prendre cette option plutôt que de continuer sur l'existant, sachant que nous étions partis sur l'ancien starter-kit maison en Angular. **Il m'a permis de rapidement commencer le projet la première année**, mais je me suis vite rendu compte que beaucoup de code n'était pas utilisé et pour moi, c'était perdre du temps de nettoyer un projet qui, de toute façon, n'était pas à jour avec Angular.

J'ai alors pu expérimenter certaines **choses** sur le projet pour essayer de le rendre le plus propre et le plus générique possible, c'est-à-dire que plusieurs fonctionnalités utilisent le même code, les mêmes fonctions, les mêmes classes, indépendamment du type de **donnée**. **Après plus d'un an passé**, ce code est toujours utilisé et une fois maîtrisé par les nouveaux collaborateurs, permet d'ajouter de nouvelles pages web similaires aux précédentes très rapidement et avec un minimum de code.

J'ai ensuite pris la décision de reprendre le code de la partie administrateur pour l'intégrer complètement à la partie front utilisateur afin d'être cohérent sur les projets et surtout ne pas

avoir à déployer deux projet web différents pour une même application. L'intégration s'est faite rapidement grâce au code générique dont j'ai parlé plus haut.

Le front utilisateur maintenant déployé, grâce à mon expérience acquise sur le projet, j'ai pu exprimer mon ressenti par rapport à Firebase ainsi que mon envie de nous en séparer à mon chef de projet. La décision est alors prise de nous détacher complètement de Firebase et de créer notre propre API en micro-services. Grâce à celle-ci, il sera éventuellement possible de déployer l'application pour d'autres client en mode édition logicielle SaaS (Software as a Service, j'explique plus tard ce qu'est le SaaS).

Etant la personne la plus ancienne sur le projet et celle ayant le plus d'expérience, j'ai maintenant la responsabilité de relire et de valider, ou non, le code Angular de mes collègues sur le projet AmilApp. En général, chaque personne de l'équipe peut choisir le ticket, la fonctionnalité sur laquelle il veut travailler, mais régulièrement, mes collègues viennent me demander ce qu'ils peuvent commencer lorsque les tâches sont plus ardues. Dans ce cas-là, je n'hésite pas à régulièrement demander si tout se passe bien et fourni de l'aide si besoin tout en essayant de faire en sorte qu'ils comprennent le code du mieux possible.

### 3. Analyse du contexte : Le projet AmilApp

#### 3.1. AmilApp

Dans cette partie, je vais présenter le projet sur lequel je travaille depuis bientôt deux ans. Comme je ne pars pas en mission chez les clients, je suis le seul développeur à être resté sur le projet depuis le début et les nouvelles personnes qui arrivent sur le projet sont, soit de nouveaux Amiltoniens qui vont se former aux technologies utilisées sur l'application, soit des Amiltoniens en inter-contrat.

La bonne entente ainsi que la bonne ambiance étant des valeurs fortes d'Amiltone, l'entreprise se doit d'avoir un moyen pratique de communiquer avec ses collaborateurs. AmilApp permet au département communication d'Amiltone de partager différentes informations avec les collaborateurs. L'application est divisée en plusieurs parties : les News (actualités), les Événements, les Sondages et les Alertes. L'avantage de cette application est de pouvoir centraliser toutes les informations en lien avec Amiltone et permet de rendre les différents posts publiés sur l'application, habituellement communiqués via Teams ou par courriel, plus visibles pour tous les collaborateurs d'Amiltone. Ceux-ci sont censé pouvoir se connecter avec le même compte qu'ils utilisent en entreprise. Mais à cause de la complexité du développement

d'une telle fonctionnalité compatible avec Firebase, elle a été repoussée et c'est l'authentification intégrée de Firebase qui a été utilisée.

Après un développement de plus d'un an, AmilApp sort sur iOS, Android et web en version 1.0. Le site web reprend les fonctionnalités des applications mobiles. En plus de celles-ci, AmilApp dispose d'un backoffice administrateur pour pouvoir ajouter du contenu sur l'application. Pendant toute cette année de développement, j'étais en charge des parties backoffice administrateur et site web, codées en Angular 8. AmilApp ne dispose pas de backend à proprement parlé puisque nous utilisons la solution cloud de Google, Firebase. J'expliquerai un peu plus tard pourquoi nous l'avons choisi ainsi que ces avantages et inconvénients.

Je vais présenter ci-dessous les fonctionnalités principales d'AmilApp.

### ◆ *Les news*

Les News, ou Actualité, permettent d'informer les utilisateurs des dernières nouvelles chez Amiltone ou en lien avec les nouvelles technologies. Actuellement, une News peut afficher une image de présentation, des pièces jointes au format PDF et peut être reliée à un sondage ou à un événement. Plus tard, il sera possible d'y ajouter une galerie d'images, de cibler les personnes qui recevront la News, et même la possibilité de « liker » celle-ci. À la publication d'une News, les collaborateurs d'Amiltone ayant téléchargé l'application reçoivent une notification sur leur téléphone.

### ◆ *Les événements*

C'est dans la partie Événement qu'Amiltone communique sur les soirées d'agence, sorties ski, ou, plus local, les sessions sportives ou cinéma. Les administrateurs ont la possibilité de sélectionner une liste d'invités qui recevront une notification sur leur téléphone. Que ce soit l'application mobile ou le site web, les collaborateurs ont la possibilité d'indiquer leur présence ou non, tant que la date limite de réponse n'est pas dépassée.

### ◆ *Les sondages*

Les Sondages permettent à la Communication de l'entreprise de poser des questions aux collaborateurs pour ensuite avoir accès à des statistiques. Un sondage peut être de type ouvert, les utilisateurs répondent avec leurs propres mots ce qu'ils veulent ; de type multiple, les utilisateurs doivent choisir une ou plusieurs réponses prédéfinies ; ou de type date, les utilisateurs ont la possibilité de sélectionner une ou plusieurs dates prédéfinies, encore une fois. Les administrateurs ont la possibilité de cibler le sondage en sélectionnant une liste de personnes, ou d'agence, qui pourront participer à celui-ci.



## ◆ Les alertes

Les Alertes permettent de lancer des rappels aux collaborateurs. Les administrateurs doivent sélectionner une liste de personnes qui la recevront.

## ◆ Les améliorations futures

Il y a encore beaucoup d'améliorations de prévues pour AmilApp : les activités, les enquêtes, la possibilité pour les collaborateurs d'Amiltone de remplir leur rapport d'activité mensuel, une refonte complète du design de la partie administrateur du site web. Mais pour le moment, la prochaine grosse amélioration prévue est la création d'une API pour remplacer Firebase. J'explique ci-dessous pourquoi la décision de se séparer du service de Google a été prise.

### 3.2. Firebase

Firebase est une solution proposée par Google qui permet la création de backend simplement et rapidement, le tout scalable et performant. Il permet aussi de gérer le stockage de fichier dans le cloud et prend en charge la gestion de comptes utilisateur.

Firestore est le service qui sert de base de données pour AmilApp et propose une actualisation des données en temps réel. Il est basé sur du NoSQL et ressemble beaucoup à MongoDB.

#### 3.2.1. Pourquoi l'avoir choisi ?

AmilApp n'était pas le premier projet à utiliser Firebase chez Amiltone. AmilCar, une petite application permettant la gestion du parc automobile d'Amiltone, a été le premier essai avec Firebase et a été développé en une semaine. Firebase a donc été choisi pour gagner du temps. Au début du projet, nous étions trois développeurs en alternance à travailler dessus et personne n'était là en même temps. S'il avait fallu qu'on développe le backend, le projet aurait été beaucoup plus long, sachant qu'à trois, on comptabilisait 26 jours de travail par mois au lieu de 60 si on avait été à temps plein.

#### 3.2.2. Avantages et inconvénients de Firebase

Firebase permet la mise en place d'un backend très rapidement et prend en charge l'authentification, qui est une partie souvent redondante et lourde à développer dans n'importe quelle application. De plus, l'utilisation de Firestore est assez simple et la documentation sur internet est plutôt bien faite.

Mais bien que Firebase soit très pratique pour une petite application, la solution proposée par Google ne semble pas assez mature lorsqu'il s'agit d'avoir des interactions plus complexes avec Firestore. En effet, sur AmilApp, on a besoin de récupérer certaines données avec des tris assez précis, et avec une API normale, lorsque que l'on fait une requête, les données reçues sont déjà ordonnées comme on le souhaite. Or, avec Firebase, certaines opérations,



pourtant assez basiques dans un langage comme le SQL ou le NoSQL, n'existent tout simplement pas sur Firestore. J'ai donc été obligé de lancer plusieurs requêtes en même temps avec des tris différents pour avoir accès aux données que je souhaitais recevoir précisément. Le code s'en retrouve beaucoup plus complexe dès lors que l'on souhaite intégrer l'actualisation des données en temps réel ou le « lazy-loading », pratique qui consiste à attendre une action de l'utilisateur pour charger des données et qui permet de grandement accélérer le chargement d'une application.

Enfin, la gestion du cache serveur de Firebase ne fonctionne correctement seulement si l'application est hébergée sur leur cloud, ce qui n'est pas le cas d'AmilApp. J'ai donc dû gérer moi-même le cache côté navigateur, mais certains bogues persistent à cause du lazy-loading, des requêtes envoyées simultanément pour récupérer les données correctement et l'actualisation des données en temps réel.

Avec tous ces inconvénients, le plus simple d'après moi est de se séparer de Firebase et de créer notre propre API.



### 3.3. Un projet basé sur l'innovation technologiques

Avec AmilApp, il a été possible pour Amiltone et les équipes de développement de tester et d'expérimenter les dernières innovations informatiques sur le marché, afin que chacun puisse avoir son propre avis sur celles-ci.

#### 3.3.1. Préparé pour le cloud

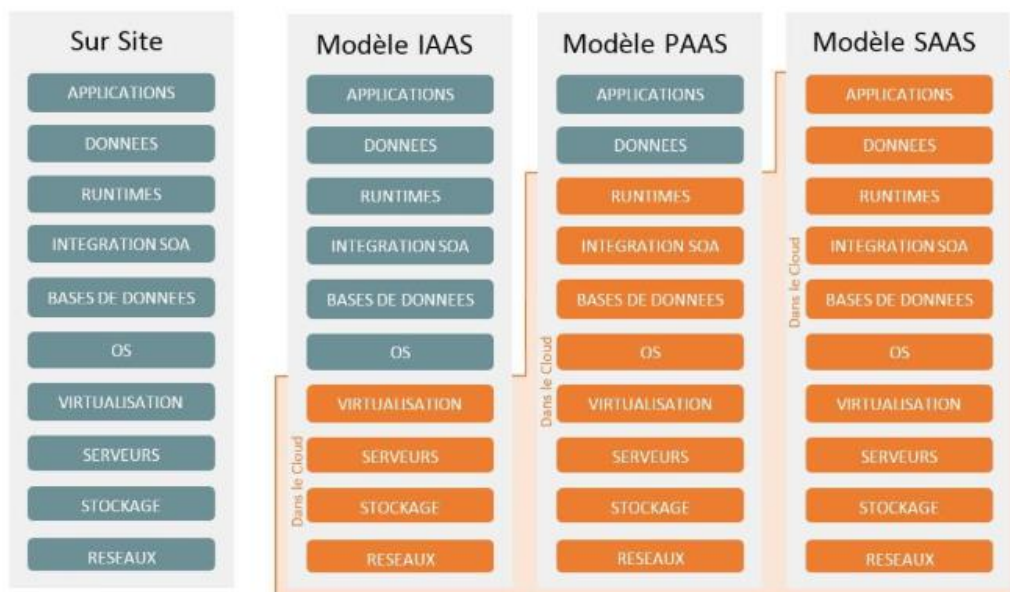
En informatique, le « Cloud », nuage en français, ou « Cloud Computing » désigne la mise à disposition de services ou de ressources accessible via les différents réseaux de communication. Les services Cloud les plus connus sont ceux de Google avec Google Drive, Microsoft avec OneDrive ou encore Apple avec iCloud. Ces services servent surtout aux particuliers pour stocker leurs documents. Mais le cloud ne sert pas seulement à cela, il peut aussi permettre l'utilisation de logiciel directement depuis le navigateur internet par exemple, sans installation requise. Prenons un exemple : Shadow<sup>1</sup>. Lorsque l'on souhaite jouer à un jeu vidéo sur ordinateur, il faut acheter tout le matériel : processeur, carte graphique, barrettes de mémoire vive, espace de stockage, alimentation, etc. Le tout coûte plus ou moins cher en fonction de la configuration choisie et a une durée de vie limitée, ou devient obsolète à cause

---

<sup>1</sup> 11 août 2020 : <https://shadow.tech/frfr>

des évolutions technologiques en termes de graphismes. Shadow propose de louer un ordinateur à travers le cloud, avec une configuration minimum suffisamment puissante pour pouvoir jouer tranquillement, le tout directement accessible depuis n'importe quel navigateur web et même depuis un smartphone, en utilisant le cloud. Les configurations disponibles sont mises à jour à chaque nouveauté technologique, l'obsolescence est maîtrisée. Dans ce cas-là, le cloud permet de nous séparer des contraintes matérielles que sont les équipements informatiques, souvent onéreux.

Il existe plusieurs modèles de cloud, les trois principaux sont **SaaS**, **PaaS** et **IaaS**.



Annexe 1

Le **SaaS**, abréviation de Software as a Service, désigne l'utilisation d'un logiciel comme d'un service. Pas besoin d'installer le logiciel sur l'ordinateur pour l'utiliser, il suffit en général de passer via un navigateur web.

Le **PaaS**, abréviation de Platform as a Service, désigne l'utilisation d'une plateforme, c'est-à-dire d'une machine avec un système d'exploitation, le tout prêt à être utilisé. L'exemple du Shadow ci-dessus utilise le modèle **PaaS**.

Le **IaaS**, abréviation de Infrastructure as a Service, désigne l'utilisation d'une infrastructure comme d'un service, c'est-à-dire qu'il n'est pas nécessaire d'acheter le matériel pour installer l'infrastructure. Il suffit d'installer les serveurs que l'on souhaite utiliser et c'est nous qui gérons les systèmes d'exploitation installés dessus. L'avantage est qu'il n'est pas nécessaire d'avoir un emplacement pour placer le matériel.

En modifiant l'architecture du projet, AmilApp devient une application cloud-native, c'est-à-dire une application utilisant des services indépendant et faiblement couplé, ce que sont les micro-services, et donc prête pour être intégrée dans un cloud.

Ajouter un 3.3.2

### 3.4. Basé sur le partage de connaissances

#### 3.5. Analyse personnelle du contexte de stage de fin d'étude

Le sujet choisi a pour cadre AmilApp, un projet important aux yeux d'Amiltone car il représente la vitrine technologique de l'entreprise sur les technologies du web. Un des avantages de celui-ci est le fait qu'il ne soit pas destiné à un client, pour le moment, et n'a donc pas de deadline.

Le nouveau backend d'AmilApp est un projet complexe sur le plan technologique car il demande aux développeurs de nouvelles compétences encore inexplorées sur un projet interne chez Amiltone. J'ai, pour ma part, pu être assez autonome sur le projet et j'ai pu apprendre de nouveaux concepts novateurs et passionnants. Je me suis beaucoup investi au niveau de la qualité de code sur l'application, et la confiance que m'a donné le chef de projet en charge d'AmilApp en me donnant la responsabilité de faire la revue de code sur le projet m'a permis d'y veiller du mieux possible. Et étant sur le projet depuis ses débuts, j'ai essayé de garder une certaine rigueur et ai fait en sorte qu'elle soit partagée par mes collègues qui se forment sur celui-ci.

Travailler dans une équipe comme celle d'AmilApp, une équipe qui évolue tout le temps, a été très enrichissant car j'ai pu « former » de nouvelles personnes sur des technologies qui me passionnent et j'ai pu leur transmettre mon enthousiasme pour le projet, ce qui s'est ressenti lors des brainstormings où tout le monde proposait de nouvelles idées pour améliorer l'application.

Au vu du nombre de technologies qu'utilise AmilApp, ce fut pour moi la possibilité d'acquérir ou de progresser sur de nouvelles compétences comme Docker, Firebase ou les micro-services. Comme je formais les nouveaux développeurs sur le projet, je pense avoir non seulement progressé sur le plan technique, mais aussi sur le plan relationnel.

L'utilisation des dernières technologies et concept en vogue sur le marché de l'informatique a été pour moi une source de motivation car la plupart du temps, elles sont un gage de qualité pour un projet et je pense que c'est ce qu'il me plaît le plus dans le développement informatique.

## 4. Problématique : Comment mettre en place une solution web réutilisable, modulaire et évolutive ?

### 4.1. Le démarrage d'un projet, une perte de temps ?


Il n'est pas rare de retrouver les mêmes fonctionnalités sur différentes applications, telles que l'authentification, la gestion d'envoi de mail, la gestion des utilisateurs ou encore, plus technique, les services, ou les morceaux de code, qui permettent la connexion à la base de données, ceux-ci sont souvent génériques et similaire entre les projets utilisant les mêmes types de stockage de données. Pour les développeurs, tout ce code est long à produire et est redondant entre les applications qui utilisent les mêmes langages. Et comme nous le savons tous, un bon développeur est un développeur « paresseux », son objectif est de réécrire le moins de code possible. Il doit donc produire un code plus générique, plus factorisé.

AmilApp est la première application interne web ayant pour base l'architecture micro-services. Le projet sert surtout d'expérimentation, de voir ce qu'il est possible de faire ou non, et d'apprendre de nos potentielles erreurs. Mais une fois terminé, il sera très intéressant pour les futurs et autres projets internes d'utiliser une architecture micro-service, en fonction de la taille de l'application. Et comme AmilApp sert de base de lancement, j'ai pour objectif de faire en sorte que l'application soit la plus générique et évolutive possible, donc réutilisable sur d'autres projets.

### 4.2. Les contraintes de l'architecture monolithique

L'architecture monolithique représente la façon dont sont associés les différentes fonctionnalités d'une application. En effet, dans cette architecture, les services et composants sont interconnectés et interdépendants. Ils communiquent en général en appelant directement les classes et les fonctions des services dont ils ont besoin. De cette manière, tous les composants doivent être présents et fonctionnels pour permettre l'exécution ou la compilation du code, ils forment un tout. De fait, lors de la modification d'une classe ou d'une fonction, il est tout à fait possible qu'une autre partie du code utilisant cette classe ne fonctionne plus correctement, et donc cela implique de retester toute l'application.

Avec le temps, les applications deviennent de plus en plus complexes au fur et à mesure que de nouvelles fonctionnalités sont ajoutées aux existantes. Avec cette complexité arrivent plusieurs désavantages :

- Il devient plus difficile de faire évoluer le projet proprement. Chaque évolution apportant son lot de nouvelles interactions avec le code existant, même avec des tests de code solides, le projet devient de moins en moins lisible et maintenable.
- Le projet perd en fiabilité à cause des potentiels « fix » qui rendent les fonctionnalités assez instables.
- Lors de l'ajout d'une nouvelle fonctionnalité, il se peut qu'il soit plus pratique de changer  un morceau de la structure du projet. Mais plus le projet est gros, plus il est difficile de changer cette structure sans risquer de casser d'anciennes fonctionnalités.

### 4.3. Les micro-services

Le but de cette partie est de faire un tour des différentes définitions et structurations de l'architecture micro-services.

#### 4.3.1. Définition

Le but des micro-services est de diviser un gros projet en plusieurs petits projets, que l'on appelle **micro-services**. Voici une définition simple : « Microservices are small, autonomous services that work together »<sup>2</sup> (les micro-services sont des petits services autonomes qui travaillent ensemble). Elle permet, sans être trop technique, de mettre en avant les deux principes fondateurs des micro-services :

- L'existence de services : c'est-à-dire de modules dont les finalités sont différentes. Un service peut travailler avec un autre service, mais chacun aura son propre objectif.
- Les services sont petits. De manière générale, un micro-service a une seule fonction.

#### A compléter ou modifier

Voici les principes fondamentaux des micro-services :

- Un service n'est responsable de l'exécution que d'un nombre restreint de tâches, axées sur une seule fonctionnalité technique ou métier, comme par exemple la gestion du panier de l'utilisateur sur un site de vente en ligne.
- Un service est complètement autonome et possède son propre système d'exploitation et ses propres données.
- Un service est développé avec les technologies (langage et base de données) qui répondent le mieux au besoin.
- En général, les micro-services sont des API REST (Application Programming Interface Representational State Transfer) et utilisent le protocole HTTP.

---

<sup>2</sup> 18 juillet 2020 : <https://www.oreilly.com/library/view/building-microservices/9781491950340/ch01.html>

### 4.3.2. Pourquoi les avoir choisis ?



Aujourd'hui, le « Cloud Computing » est devenu un standard dans l'utilisation de nos applications ainsi que dans leur développement. Celles-ci doivent être **disponible** pour tous les utilisateurs à tout moment et depuis n'importe où. Pour une grosse application, choisir le cloud c'est permettre de l'adapter rapidement en fonction de l'utilisation qu'en font les utilisateurs et c'est aussi permettre aux équipes de développement de se focaliser sur le produit final. Ne pas l'utiliser, c'est devoir prendre le temps **pour** concevoir une architecture serveur robuste, mais c'est aussi prendre le risque de voir augmenter les factures du matériel informatique, qui peut potentiellement ne pas être amorti en fonction des utilisateurs encore une fois.



Pour utiliser le cloud, il vaut mieux s'y préparer en développant directement des projets adaptés au cloud computing. C'est ici que les micro-services nous intéressent.



Une fois **déployée**, les micro-services permettent à une application d'être plus scalable, c'est-à-dire qui s'adapte d'un point de vue dimensionnel, tant vers des tailles inférieures que vers des tailles supérieures, en fonction des pics d'utilisation de celle-ci, et d'être plus tolérante à la panne.



Mais les micro-services permettent aussi aux entreprises d'optimiser les ressources consacrées au développement des applications. En effet, utiliser ce type d'architecture permet de diviser une grosse équipe en plusieurs équipes de tailles réduites, qui se consacrent seulement à un seul service, indépendamment **des autres équipes**, et est plus adapté au fonctionnement en mode agile. « *Les organisations qui conçoivent des systèmes [...] tendent inévitablement à produire des designs qui sont des copies de la structure de communication de leur organisation* » (Loi de Conway), autrement dit, l'organisation des équipes découpées en sous-équipes indépendantes représente l'architecture globale de l'application, l'architecture micro-services. Comme chaque service est indépendant, chaque équipe peut



prendre les décisions les plus adaptées et optimisées pour répondre au cahier des charge imposé. Enfin, la taille relativement petite de chaque service permet de **les** réécrire aisément avec des technologies plus récentes et plus pertinentes : la complexité globale du projet s'en retrouve réduite.

Contrairement à l'architecture monolithique, les micro-services sont optimisés pour la mise à l'échelle horizontale, c'est-à-dire que lorsque les capacités de la machine qui héberge l'application sont trop limitées pour supporter le nombre de requêtes utilisateurs, l'application est facilement déployable sur plusieurs serveurs distincts de façon à réduire la charge de travail entre eux. Et, contrairement à la mise à l'échelle verticale, l'application reste toujours

disponible pendant l'ajout de ressources supplémentaires : si **un** des serveurs tombe en panne, l'application reste disponible.

Globalement, le projet devient plus facile à appréhender pour les développeurs puisqu'ils **se focalisent** sur une seule fonctionnalité. Chaque service est donc plus **petit** en termes de code et **est** plus rapidement testable. **De plus, étant donné qu'ils peuvent être déployés indépendamment des autres, les équipes n'ont pas besoin de coordonner le déploiement des modifications spécifiques à leur service.** Le déploiement continu devient donc possible là où il ne l'est pas avec une architecture monolithique.

Malgré des avantages indéniables, l'architecture micro-services n'est pas à la portée de toutes les entreprises. Elle implique de lourds changements au niveau de l'organisation des équipes d'un projet et une entreprise de petite taille, ou un projet avec une équipe déjà réduite, aura plus de mal à diviser ses équipes. Celles-ci devront apprendre de nouvelles pratiques de **communications, voir même** un changement de culture **lorsque** la méthode agile n'était pas du tout appliquée.

Etant encore une architecture récente et évoluant très rapidement, les développeurs devront faire une veille technologique plus régulière et plus importante que pour une architecture monolithique. Les micro-services restent complexes à mettre en œuvre et **demande** aux développeurs de nouvelles connaissances, de nouvelles bonnes pratiques, **et est** donc plus **couteuse** en temps et en conception.

#### Parler du changement dans l'interface d'un service qui est lourd à gérer

Enfin, tous les projets n'ont pas l'utilité d'être **développée** en micro-services. Par exemple, une application CRUD (Create, Read, Update, Delete, autrement dit, une application de gestion de données) spécifique à un seul type de données n'aura pas besoin de ce type d'architecture et pourra largement se contenter d'une architecture monolithique qui sera beaucoup plus simple et rapide à mettre en place.

Netflix, application dont la popularité n'est plus à prouver, a décidé en 2008 d'utiliser les Amazon Web Service, solution cloud d'Amazon, et a dû adapter son architecture pour mieux correspondre au cloud. Ce sont ces deux décisions qui ont façonné le succès de l'entreprise aujourd'hui. Celle-ci est même devenue un acteur majeur dans la définition des micro-services et des bonnes pratiques à mettre en place.

Amiltone a choisi d'utiliser les micro-services pour son application AmilApp afin de pouvoir potentiellement déployer la solution pour d'autres clients en mode édition logicielle Saas, **Logiciel as a Service**. **En effet, chaque client n'a pas les mêmes besoins et peut n'avoir envie que d'une partie des fonctionnalités que propose AmilApp.** Enfin, l'utilisation des micro-



services permettra la création de services réutilisables sur d'autres projets internes tels que l'authentification.

## 4.4. Le starter-kit

### 4.4.1. Définition

« En traitement de texte, un boilerplate est un segment de texte mis en mémoire pour être fréquemment utilisé et pouvant être combiné avec d'autres textes pour créer un nouveau document »<sup>3</sup>, voici la définition d'un « boilerplate ». Un starter-kit n'est pas un boilerplate, mais s'en rapproche beaucoup. Le principe d'un starter-kit est de créer du code qui sera réutilisable ou facilement intégrable dans n'importe quelle application. La différence avec un boilerplate, c'est que le starter-kit contient des fonctionnalités de base d'un projet, qui sont souvent communes à énormément d'applications, comme par exemple toute la partie authentification d'utilisateur, la gestion de mail, la gestion de tâches planifiées. Un starter-kit n'est pas propre à un seul langage et peut être créé pour n'importe quelle technologie avec n'importe quelle fonctionnalité, le but étant d'éviter aux développeurs de passer du temps sur du code qui existe déjà dans d'autres applications.

Chez Amiltone, la Web Factory a créé un starter-kit contenant déjà énormément de fonctionnalité de base, actuellement utilisé sur tous les projets internes nécessitant une API. L'objectif est d'éviter de repartir de zéro à chaque nouveau projet.

### 4.4.2. Avantages

Les avantages semblent assez évidents ici, le principal étant d'accélérer le développement d'un projet. Les starter-kits sont souvent utilisés dans les compétitions de code où l'objectif est de créer un projet innovant le plus rapidement possible. Dans ce genre de cas, le starter-kit permet de gagner de nombreuses heures sur des fonctionnalités basiques et répétitives.

Une fois qu'une fonctionnalité a été développée, testée et est mise en place sur un projet, on peut être sûr de sa qualité, fonctionnellement parlant, et en termes de code. En la redéveloppant une seconde fois depuis le début sur un autre projet, on ne peut assurer la même qualité de code bien que fonctionnellement ce soit la même chose. De plus, des bugs peuvent potentiellement survenir là où il n'y en a pas sur la première application. Enfin, il faut à nouveau tester la fonctionnalité ce qui prend encore du temps. En développant un starter-kit, on s'assure que le code est de bonne qualité en le testant, et en l'utilisant dans de nouveaux projets, non seulement on gagne du temps en ne le redéveloppant pas depuis le début, mais on gagne aussi du temps sur la partie de tests.

---

<sup>3</sup> 8 août 2020 : <http://dircomleblog.canalblog.com/archives/2011/07/29/21695781.html>



Partie pas terminée

## 5. Méthodes habituellement utilisées pour une situation présentant des similitudes : [Titre perso]



Dans cette **partie**, je présente deux solutions : Spring Boot et Hackathon-starter, qui sont largement utilisées au sein de la communauté de développeurs pour démarrer un projet dans de plus brefs délais.

### 5.1. Spring Boot

Spring est un framework open source bien connu et très utilisé dans le monde du Java pour les fonctionnalités qu'il propose : l'injection de dépendances, une gestion des instances de classes, des outils pour les applications web et encore beaucoup d'autres. Mais sa configuration reste très complexe et peut prendre beaucoup de temps. Les équipes de Spring ont alors créé Spring Boot.

#### 5.1.1. Fonctionnalités

Spring Boot est donc un framework qui rend la configuration de Spring plus rapide et beaucoup plus simple, en fournissant différentes annotations à placer dans le code, comme **@EnableAutoConfiguration**, qui, comme son nom l'indique, active l'auto-configuration pour toutes les dépendances présentes dans l'application.

Spring Boot est utilisé pour créer des applications micro-services et permet donc d'obtenir une application packagée, complètement autonome, comme le sont les micro-services, et prête pour la production (« production ready ») dès la création du projet. De plus, le framework intègre directement un serveur Web dans l'application, Apache Tomcat par exemple, qui sera démarré automatiquement au lancement de celle-ci.

Pour la création d'une application via Spring Boot, le site web Spring Initializr (<https://start.spring.io>) permet de générer très rapidement la structure du projet tout en y incluant les dépendances que vous aurez sélectionné sur le site.



Enfin, l'application contiendra automatiquement différentes routes pour afficher des statistiques (**métrique**) sur celle-ci, telles que l'utilisation de la mémoire, la liste des requêtes http, l'état de santé de l'application.

#### 5.1.2. Avantages et inconvénients par rapport au projet

Grâce à Spring Boot, le développement d'application Java est très simplifié et beaucoup plus rapide. En évitant aux développeurs la configuration XML complexe, le code *boilerplate*, c'est-à-dire du code qui doit être **inclus** de façon répétitive dans un projet, et en fournissant des



paramètres par défaut pour démarrer un nouveau projet rapidement, le framework augmente grandement la productivité des équipes. La possibilité de créer un projet fonctionnel et « production ready » en très peu de temps grâce à **Spring Initializr** simplifie encore une fois la tâche des développeurs. De plus, celui-ci est axé sur les micro-services et est extrêmement modulable grâce à l'auto-configuration des dépendances. La présence d'un outil CLI (Command Line Interface) permet de développer et tester les applications Spring à partir d'une invite de commande facile à prendre en main. Enfin, l'utilisation de Spring Boot pour une grosse application monolithique est largement déconseillée.

Globalement, le framework Spring Boot aurait pu correspondre à notre projet, mais le langage n'est pas celui recherché malheureusement.

## 5.2. Hackathon-starter

Le hackathon est un événement pendant lequel des équipes de développeur se réunissent pour créer des projets informatiques dans un temps imparti souvent très court : un weekend, une journée ou même une nuit. L'objectif est alors de coder le plus rapidement possible un logiciel ou une application qui sorte du lot. C'est donc tout naturellement que j'ai recherché l'existence d'un starter-kit destiné aux hackathons, qui allait permettre de gagner un temps précieux pendant ces événements. Mon choix s'est porté sur Hackathon-starter.

### 5.2.1. Fonctionnalités

Hackathon-starter est un projet starter-kit open-source, c'est-à-dire un projet libre d'accès et de droits, écrit en JavaScript et NodeJS, hébergé sur la plateforme Github. Avec ses trente milles étoiles, il fait partie des projets qui en recueille le plus sur la plateforme. De base, il inclut énormément de fonctionnalités déjà codées :

- Toute la partie authentification, que ce soit en local, avec l'adresse email et le mot de passe, ou avec OAuth 1.0a et OAuth 2.0 pour s'authentifier via différents réseaux sociaux.
- Une gestion des notifications
- Une gestion des comptes utilisateurs
- Des pages web prédéfinies comme la page de profil par exemple

Le starter-kit utilise MongoDB pour la base de données et est compatible avec Docker.

A compléter

### 5.2.2. Avantages et inconvénients par rapport au projet

Les avantages à utiliser ce starter-kit sont assez nombreux, tout est déjà codé : la connexion avec la base de données, la partie authentification et même quelques pages web !

Malgré toutes ces fonctionnalités, ce projet ne conviendra pas à mon problème. Tout le code est écrit en JavaScript et je souhaite utiliser le TypeScript pour son typage très pratique et qui permet de garder un code plus propre et plus facilement maintenable lorsque de nouvelles personnes arrivent sur le projet. De plus, Hackathon-starter est très complet, trop complet même pour mon problème. Je n'ai pas besoin des pages web prédéfinies puisque chaque micro-service contiendra seulement une API Rest. Qui veut dire trop complet dit code inutile et donc suppression de code, ce qui prend énormément de temps. De plus, il faut faire attention à ne pas casser des fonctionnalités que l'on souhaite garder.

### 5.3. Autres solutions



J'ai pu trouver beaucoup de **starter-kit** différents sur Github, mais quasiment aucun ne respectait les conditions requises pour mon problème. Beaucoup ne sont pas **écrit** en TypeScript, et pour ceux qui le sont, ce sont surtout des starter-kits pour Angular ou React. Le type de base de données utilisé ne me convient pas forcément non plus.

## 6. Exposé des décisions prises et des interventions menées par le stagiaire pour résoudre le problème : [Titre perso]

### 6.1. Objectifs et contraintes du projet

### 6.2. La mise en place de l'architecture

#### 6.2.1. Le choix de l'API Gateway

Une API Gateway, ou passerelle API en français, est **le point d'entrée** pour les APIs et les micro-services définis. Elle permet d'assurer la protection, l'évolutivité et la haute disponibilité de **celles-ci**. C'est par la passerelle API que transitent toutes les requêtes utilisateurs.

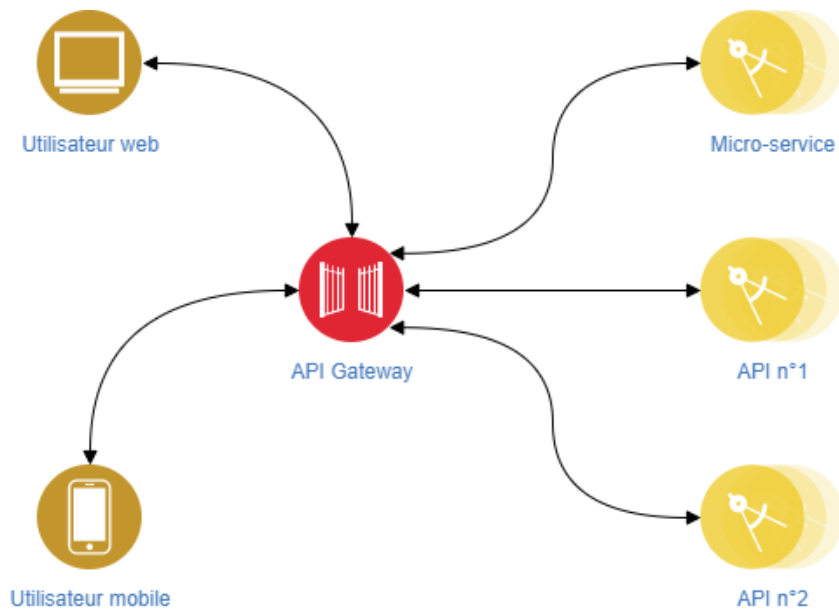


Schéma représentant le fonctionnement d'une passerelle API

L'API Gateway « **cache** » la complexité d'un backend au client en lui permettant d'appeler seulement la passerelle au lieu des multiples APIs et micro-services. Dans une architecture micro-services, la passerelle API permet d'**externaliser** toute la configuration de chaque API directement sur celle-ci.

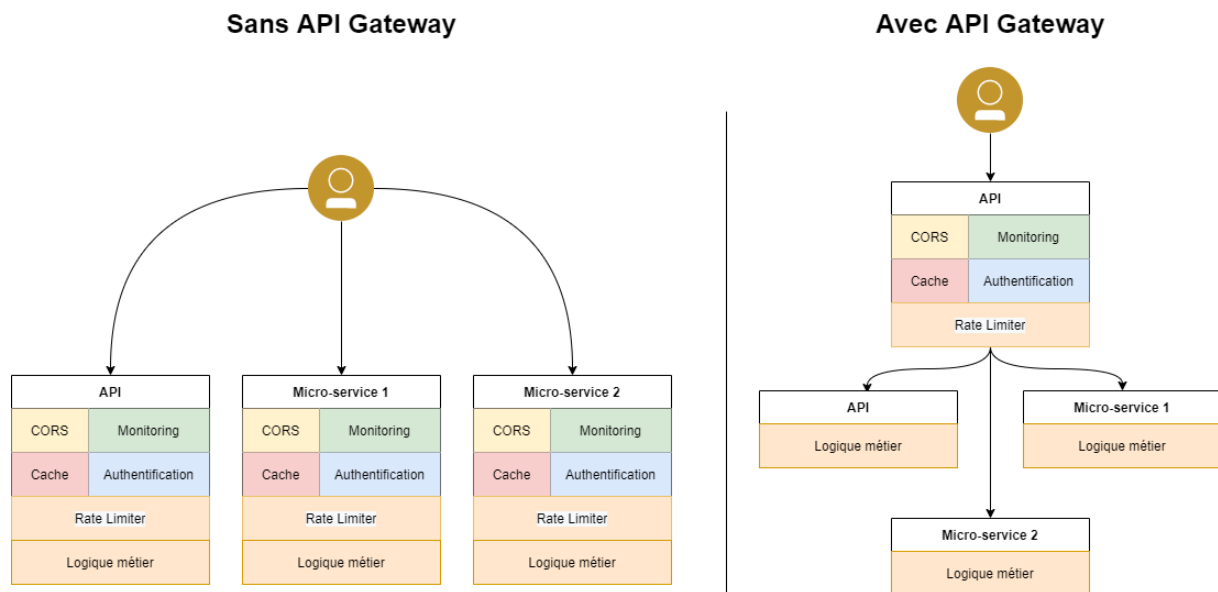


Schéma représentant l'externalisation de la configuration des APIs

Je vais présenter ci-dessous plusieurs solutions API Gateway toutes open-source.

#### ◆ *Gravitee*

Gravitee est une passerelle API développée en Java par l'entreprise Gravitee Source depuis 2015. Celle-ci dépend de 3 applications : un dashboard, MongoDB et la passerelle en question. Les deux premières permettent de gérer et de configurer l'API Gateway. La configuration est enregistrée sur MongoDB et est utilisée par l'API Gateway.

L'avantage de Gravitee est son dashboard, celui-ci est clair, facile à comprendre et pratique pour les développeurs. Il permet d'entièrement configurer la passerelle (les routes, l'authentification, etc...).

Mais l'installation de cette solution est complexe et la documentation est trop légère et devient le point noir de Gravitee. De plus, la communauté n'est pas très grande (à peu près 1000 étoiles sur GitHub) et il n'est pas simple de trouver des tutoriels sur internet.



Au niveau des performances, Gravitee a besoin de beaucoup de mémoire vive (RAM). **Il est conseillé d'utiliser une machine par instance de la passerelle puisque celles-ci utilisent au minimum 4GB de mémoire vive chacune.** La solution est scalable horizontalement et verticalement. Enfin, une fois le tout mis en place, les performances sont impressionnantes.

En conclusion, Gravitee est difficile à installer mais offre de bonnes performances et une interface complète et pratique.

## ◆ *HAProxy*



HAProxy est une solution open-source développée en C et Lua. Elle est particulière par rapport aux autres solutions car **celle-ci** est d'abord un Load-Balancer (répartiteur de charge en français) avant d'être une API Gateway. C'est-à-dire qu'elle est faite avant tout pour garantir la haute disponibilité des APIs. Mais elle peut être utilisée comme passerelle API.



HAProxy est simple à mettre en place avec seulement une seule application. **La solution ne possède pas encore de dashboard pour l'API Gateway mais les équipes en charge de celle-ci sont en train d'en développer un afin de corriger cette faiblesse.** HAProxy ne possède pas autant de fonctionnalités que les autres solutions (pas de connexion OpenID Connect par exemple) mais il est possible d'en ajouter à travers des plugins disponibles sur leur site. De plus, s'il manque toujours une fonctionnalité, il est possible de développer soit même un plugin personnalisé en Lua.

Enfin, la solution s'installe et se configure très facilement grâce à sa documentation complète. La communauté et le support qui entourent le projet sont aussi très réactifs. HAProxy utilise très peu de ressources (processeurs et mémoire vive) par rapport aux autres solutions pour avoir de meilleures performances.

Dans l'ensemble, HAProxy est simple d'utilisation et performant. Il est fait pour les projets qui utilisent déjà le Load-Balancer.

## ◆ *Tyk*

Tyk est une passerelle API codée en langage Go par les équipes de Tyk Technologies depuis 2014. Comme pour Gravitee, la solution contient trois applications mais a aussi besoin de Redis et de MongoDB. Les trois applications sont la passerelle, le dashboard pour la configurer, et la Pompe qui extrait les données de Redis pour les injecter dans MongoDB afin de les récupérer sur le dashboard.

La solution est assez longue à installer mais grâce à sa documentation, elle ne pose pas de problèmes. Le dashboard permet de la configurer facilement puisqu'il est clair et ergonomique. La communauté n'est pas énorme mais est très réactive (plus de 5000 étoiles sur GitHub).



**Les performances** sont le point faible de Tyk. En effet, malgré une consommation de RAM assez faible (moins de 250MB), la solution utilise beaucoup le processeur et a du mal à gérer plus de 4000 requêtes utilisateurs par seconde.



Globalement, Tyk est une solution pratique grâce à son dashboard mais **est faite pour les applications qui n'ont pas énormément de trafic.**

## ◆ Kong

Kong est une API Gateway open-source développée en Lua et Perl par l'entreprise Kong Inc depuis 2017. Il existe une version « Entreprise » payante qui offre plus d'outils pour la solution comme un dashboard ou des outils de surveillance.



La solution a juste besoin d'une base de données PostgreSQL pour enregistrer sa configuration et peut être installé sur énormément de support et de systèmes d'exploitation.



Kong permet l'ajout de plugins, ce qui en fait son avantage le plus important. En effet, de nombreux plugins sont développé par l'énorme communauté (plus de 26 000 étoiles sur GitHub) et peuvent satisfaire toutes nos envies : authentification OpenID Connect, surveillance, limiteur de débit (rate limiter), etc... La communauté est très active et la solution évolue rapidement. De plus, Kong s'installe très rapidement et sa documentation est complète et bien faite.






Au niveau des performances, Kong utilise moins de 300MB de mémoire vive et est scalable horizontalement et verticalement.

Pour récapituler, Kong est une solution largement utilisée à travers le monde et permet de répondre à n'importe quel besoin grâce à sa communauté très active.

Après avoir comparé toutes ces solutions, les équipes d'Amiltone ont choisis d'utiliser Kong pour sa communauté, ses plugins et sa simplicité d'installation. J'ai totalement approuvé ce choix après avoir découvert des plugins pour l'authentification des utilisateurs ainsi que pour la redirection d'une requête vers un service d'autorisation (propre à AmilApp et dépendant de ses données, à ne pas confondre avec l'authentification). A compléter

### 6.2.2. La communication « interservices »

Avec l'architecture micro-services, chaque service doit être totalement **indépendant** des autres. Mais il arrive forcément un moment où un service doit utiliser une fonctionnalité d'un autre service. Par exemple, sur AmilApp, lorsque je publie un sondage, je dois uploader une image sur un autre service. Actuellement, le modèle de données des sondages contient les informations de chaque image. Voici les étapes :

-  - Le service des sondages reçoit la requête utilisateur et la traite
-  - Envoie de l'image reçue au service de gestion d'images pour qu'il la traite
-  - Envoie des informations de l'image au service des sondages
-  - Le sondage est ajouté en base de données
-  - L'utilisateur reçoit une réponse « OK, le sondage a été créé »

Tout cela de manière **synchrone**. Le service qui gère les sondages est alors dépendant du service de gestion d'images. Le problème en reproduisant ce schéma sur des micro-services est que si **un des micro-service** qui **forme** la chaîne d'appel HTTP plante, c'est la requête complète qui est annulée.

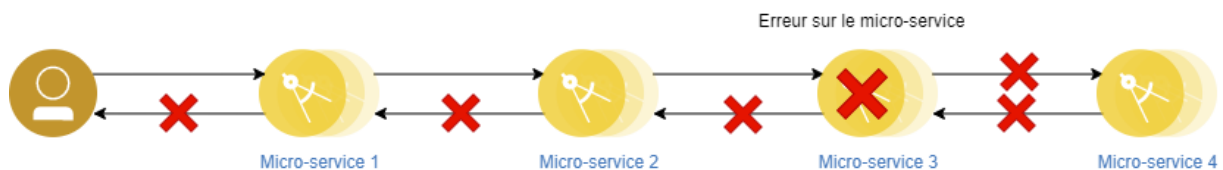



Schéma représentant les conséquences de l'appel en chaîne des micro-services

La solution est alors de rendre toutes ces opérations **asynchrones**. Si on reprend l'exemple des sondages ci-dessus : il faut faire en sorte que le sondage soit créé en base de données dès que le service éponyme reçoit la requête utilisateur pour lui répondre « OK » le plus vite possible. Le service des sondages a alors déjà notifié le service de gestion d'images pour uploader l'image. Cela implique un changement dans le modèle de données : le sondage ne doit plus contenir les informations de l'image. Cette façon de faire se nomme l'**Event-Driven Architecture** ou l'architecture orientée événements.

Dans un système orienté événements, la communication et le traitement des événements devient la structure centrale de notre application. Une telle solution peut être créée dans n'importe quel langage puisqu'elle est une approche et non un langage de programmation.

 Celle-ci permet de garder des micro-services faiblement couplés. En effet, lorsqu'un micro-service publie un événement, il ne connaît pas les consommateurs et ignore les conséquences de son apparition.



Un micro-service qui souhaite réagir à un événement doit d'abord écouter un service de message (*Message Broker* ou *Message Queue*). Un Message Broker permet de gérer des files d'attente de messages. Celui-ci recevra alors tous les événements qui l'intéresse et pourra soit les traiter, soit seulement être affecter par ce dernier, de manière **asynchrone**.



Schéma représentant un système orienté événements

Après avoir défini l'architecture **orienté** événements, il faut choisir une solution de Message Broker. Je vais présenter ci-dessous trois solutions que j'ai pu expérimenter : **Nats**, **RabbitMQ** et **ActiveMQ**.

#### ◆ Nats

Nats est un service de **message** open-source développé en langage Go. Il utilise le protocole TCP et permet de transférer un nombre très important de messages par seconde (entre 8 et 11 millions) tout en étant très léger (l'image Docker ne fait que 3Mo) et peu consommateur en ressources. Il n'en reste pas moins une solution très robuste, scalable et hautement disponible.

En général, Nats est utilisé pour le transfert de **message** très légers : moins d'un mégaoctet.

Il fonctionne en mode « **Publish/Subscribe**<sup>4</sup> » (Pub/Sub) et ne permet donc pas la persistance **de message** : s'il n'y a aucun consommateur sur un topic lorsqu'un message arrive, **celui-ci** est perdu et ne peut pas être récupéré. Enfin, Nats ne supporte pas la réplication des messages.

---

<sup>4</sup> Comme le mode « Message Queuing », les messages sont transférés d'un producteur (**de message**) vers un consommateur. Mais contrairement au Message Queuing où le message n'est reçu que par un seul consommateur puis supprimé, le mode Pub/Sub permet à tous les consommateurs de recevoir le message.

## ◆ RabbitMQ



RabbitMQ est un des services de messages open-source les plus populaires. Il utilise la norme **AMQP 0.9**<sup>5</sup> et fonctionne en mode **Message Queuing** mais supporte aussi le Pub/Sub. Grâce à l'ajout de plugins, RabbitMQ peut aussi utiliser d'autres protocoles de **message** comme **MQTT**<sup>6</sup>. Les messages et les queues peuvent être persistants, ils sont répliqués soit en mémoire vive, soit sur le disque dur, de manière **synchrone**. De cette manière, si RabbitMQ tombe, les messages ne sont pas perdus. De plus, si aucun consommateur n'écoute sur un topic lorsqu'un message arrive, celui-ci est alors conservé dans la queue jusqu'à ce qu'il soit consommé. Grâce à un système de confirmation de réception, si après avoir un reçu un message un service plante, le message sera alors remis dans la queue en attente d'être consommé.

## ◆ ActiveMQ



Apache ActiveMQ est le service de **message** open-source et multi-protocole le plus populaire. Il supporte nativement les protocoles **AMQP 1.0**, **MQTT** et STOMP. Les messages peuvent être persistants, ils sont répliqués de manière **synchrone ou asynchrone**. Globalement, ActiveMQ reprend les mêmes fonctionnalités que RabbitMQ.

L'assemblée technique d'Amilstone a décidé de créer une cohérence technologique entre les différents projets internes, et a donc convenu qu'**ActiveMQ** serait la norme chez Amilstone. Son gros avantage est le fait de pouvoir s'interconnecter avec n'importe quelle technologie grâce au large choix de protocoles proposés nativement. La prise en charge native des différents protocoles de communication en fait un choix de marque par rapport aux autres solutions. En effet, Nats ne prend pas encore en charge le MQTT et RabbitMQ propose d'autres **protocole** mais il faut installer des plugins pas forcément très maintenus. ActiveMQ est développé par les équipes d'Apache, un acteur reconnu dans le monde de l'informatique, ce qui nous garantit une **certaines** fiabilité et maintenabilité sur le long terme. ActiveMQ est en train d'être mis à jour en version 6, ce qui prouve encore une fois la réactivité des équipes de développement. Enfin, le service est nativement disponible sur les Amazon Web Service avec AmazonMQ, ce qui nous permettrait de potentiellement utiliser la solution cloud d'Amazon.



---

<sup>5</sup> AMQP, qui signifie Advanced Message Queuing Protocol, est un protocole de **message** ayant pour but de standardiser la communication entre les serveurs. Il fonctionne en mode Message Queuing. Actuellement, AMQP est en version 1.0 et n'est pas compatible avec les versions antérieures.

<sup>6</sup> MQTT, qui signifie Message Queue Telemetry Transport, est un protocole de **message** plus simple qu'AMQP et, comme son nom ne l'indique pas, ne fonctionne pas en mode Message Queuing mais en mode Pub/Sub. C'est un protocole très léger et donc très utilisé pour l'IoT (Internet of Things)

### 6.2.3. Le modèle CQRS

Dans une architecture monolithique, comme il n'y a qu'une seule base de données, il est facile de faire une jointure entre différentes collections afin d'agréger les données entre elles. Cependant, il n'est pas possible d'utiliser les jointures dans une architecture micro-services puisque chaque service possède sa propre base de données, qui peuvent toutes être différentes (SQL d'un côté, NoSQL d'un autre).

Plusieurs options s'offrent alors à nous, mais toutes ne se valent pas. Je vais les détailler en reprenant l'exemple du sondage utilisé précédemment. Pour rappel, le modèle de celui-ci ne contient plus les informations de l'image. En tant qu'utilisateur, je souhaite récupérer toutes les informations liées à un sondage : les images et les utilisateurs concernés.

**Première solution** : depuis le micro-service des sondages, je fais directement un appel au service de gestion d'images afin de récupérer les données pour les fusionner dans mon sondage et ensuite renvoyer le tout à mon utilisateur. De cette manière, je suis sûr que mes données sont bien à jour. Mais en faisant cela, nos micro-services sont fortement couplés, le service des sondages dépend du service des images pour renvoyer le modèle de données complet. Cette méthode ne convient donc pas.



**Deuxième solution** : Les données des images sont répliquées sur le service des sondages afin de rendre **disponible** les jointures. Cette option peut amener à des problèmes de données désynchronisées et, personnellement, je n'aime pas le concept de réplication de données. Toutes les bases de données contiennent toutes les données et je trouve que le principe des micro-services n'est plus vraiment respecté.



**Troisième solution** : La partie frontend fait plusieurs requêtes HTTP afin d'agréger elle-même les données. Le front fait alors trop de traitement de données et, sur un mobile, cela entraîne une perte de performance. Cette solution n'est donc pas **possible**.

**Quatrième et ultime solution** : On applique le modèle **CQRS** (Command Query Responsibility Segregation) à l'architecture micro-services. Ce modèle consiste à séparer en micro-services les opérations qui influent sur les données (Command) de celles qui les récupèrent (Query). Reprenons notre sondage, voici les étapes de sa création jusqu'à la récupération de ses données pour les afficher sur le front :

- Première requête sur le micro-service sondage. Celui-ci envoie un événement « sondage créé » avec les données du sondage créé ainsi que celles des images. Enfin, il renvoie un message « OK » à l'utilisateur.
- Le service de gestion d'image reçoit l'événement « sondage créé » et uploadé l'image en base de données. Une fois l'image uploadée, il envoie un événement « image uploadée ».

- En parallèle, le service d'agrégation de données reçoit l'événement « sondage créé » et crée le sondage dans sa base de données avec le format voulu.
- Plus tard, le service d'agrégation de données reçoit l'événement « image uploadée » et met à jour le sondage créé précédemment.
- Enfin, l'utilisateur envoie une requête sur le service d'agrégation de données qui lui renvoie le sondage au complet.

De cette manière, on peut imaginer plusieurs services d'agrégation de données qui renvoient différents formats d'un même sondage, si la requête vient d'un mobile ou d'un navigateur web par exemple.

**A compléter**

## 6.3. De l'ancien starter-kit aux micro-services

### 6.3.1. Un nouveau framework : NestJS



Le choix d'un framework peut être déterminant pour la réussite, ou non, d'un projet dans les temps. Si le but **de celui-ci** est de simplifier et d'uniformiser le travail des développeurs, un mauvais choix peut avoir de lourdes conséquences : devoir changer de framework en plein milieu de développement peut vite devenir très coûteux. De plus, choisir un framework inapproprié entraînera un développement plus long à terme, ainsi qu'une maintenance plus difficile.



Pour choisir le bon framework, il est important de cerner du mieux possible les besoins de l'application. Un framework trop spécialisé impliquera un manque de **fonctionnalité** et donc une perte de temps sur l'implémentation de celles-ci. À l'inverse, un framework trop généraliste sera trop lourd et contiendra beaucoup d'éléments inutiles à notre projet.




La popularité d'un framework est l'un des critères les plus déterminants malgré le fait qu'il ne soit pas technique. En effet, sa popularité **garanti** un meilleur suivi par les développeurs, une communauté plus grande et plus active, plus de tutoriels et une plus longue durée de vie. De plus, plus un framework est populaire, plus les chances pour que les développeurs d'une équipe l'aient déjà utilisé sont grandes, ce qui permettra de gagner du temps sur la prise en main de celui-ci.





La documentation d'un framework est un facteur important à prendre en compte et tous ne sont pas **logés** à la même enseigne. Il est donc, selon moi, essentiel de lire une bonne partie de la documentation disponible pour comprendre les différentes fonctionnalités et limitations d'un framework.

L'objectif étant de partir du starter-kit existant, il a fallu à un moment donné se poser la question du framework : est-il toujours adapté au besoin, à l'architecture micro-services ?

## ◆ Ts.ED

 Le starter-kit actuel utilise le package Ts.ED, TypeScript Express Decorator. Express est un package qui permet la création d'infrastructure web minimaliste, souple et rapide pour NodeJS.


 Ts.ED permet d'uniformiser le développement d'API en mettant à disposition des annotations et des décorateurs simplifiant le code et sa lecture. Ceux-ci permettent de définir les différentes routes de notre application d'une manière simplifiée, tout en respectant la structure **Modèle Vue Contrôleur** (MVC) grâce au système de service et d'injection de dépendances se rapprochant du modèle que propose Angular. Un développeur Angular, technologie très répandue chez Amiltone, sera alors à l'aise plus rapidement sur un projet backend.

 Un des points important pour le choix de Ts.ED à l'époque est le fait qu'il intègre une génération automatique de la documentation Swagger, un outil permettant de générer la documentation d'une API web et offrant une interface permettant de tester les différentes routes et méthodes de l'application. L'avantage de Swagger est qu'il maintient une synchronisation du code avec la documentation. Comme les équipes sur un projet évoluent souvent chez Amiltone, il est important de garder une documentation à jour, et Swagger permet de répondre à cette problématique chronophage.

Ts.ED intègre également des fonctionnalités importantes telles qu'un middleware authentification ou la génération d'erreurs pour les requêtes en entrée et sortie. Le projet est codé en TypeScript et permet une intégration simple dans notre backend. Ce framework a été choisi à l'époque car Data New Road s'était également orienté dessus de son côté sans concertation avec la Web Factory, ce qui a permis, après discussion, un réconfort dans le choix du framework, ainsi qu'une opportunité de conserver une certaine cohérence technologique entre les projets.

Mais Ts.ED souffre d'une documentation assez légère et peu informative sur l'injection de dépendances, un concept qui reste assez complexe à appréhender et à maîtriser. De plus, le framework ne prends pas encore en charge la dernière version d'OpenAPI/Swagger, ce qui est bloquant pour certains modèles de données complexes.

## ◆ NestJS

 Malgré de nombreux avantages que possède Ts.ED, je n'ai pas pu m'empêcher de regarder s'il n'existait pas de solution plus adapté à notre problème, qui intègre, d'une certaine manière, l'architecture micro-services et simplifie son utilisation. Il fallait tout de même que la solution intègre la génération automatique de la documentation Swagger et fonctionne avec Express ou un package similaire. J'ai alors trouvé **NestJS** et après des recherches poussées sur ses fonctionnalités, je l'ai proposé au référent technique ainsi qu'au chef de projet.

NestJS reprends toutes les différentes fonctionnalités que propose Ts.ED en les améliorant. Il propose l'injection de dépendances ainsi qu'une architecture basée sur des modules, exactement comme le fait Angular. Ce point en fait un atout majeur pour l'équipe puisqu'à ce moment-là, deux développeurs juniors rejoignent le développement du backend et tout d'eux n'ont jamais fait de NodeJS, mais ont l'habitude d'Angular grâce au backoffice d'AmilApp. Ils n'auront alors aucun mal à utiliser le framework. De fait, ce principe fonctionnera aussi dans l'autre sens, un développeur qui n'a utilisé que NestJS pourra rapidement prendre en main un projet Angular puisqu'il maîtrise déjà l'injection de dépendances.

Le framework propose aussi des annotations pour la définition des routes de l'API ainsi que pour la génération automatique de la documentation Swagger. Celui-ci est, tout comme Ts.ED, codé en TypeScript.

La documentation du framework est extrêmement complète et jouit d'une popularité beaucoup plus importante que celle de Ts.ED. En effet, NestJS est téléchargé en moyenne plus de 300 000 fois d'après le site **npmjs.com** et presque 30 000 étoiles sur **GitHub**, contre respectivement, un peu plus 6 000 fois et 1200 étoiles pour Ts.ED. Il est donc beaucoup plus simple de trouver de l'assistance pour NestJS.

Mais NestJS intègre également des fonctionnalités que la Web Factory a déjà développé grâce à Ts.ED. Je pense particulièrement aux tâches planifiées. Selon moi, l'implémentation de cette fonctionnalité dans le starter-kit n'est pas optimale et n'exploite pas correctement l'injection de dépendances proposées par Ts.ED. NestJS propose donc cette fonctionnalité grâce aux décorateurs et aux annotations et leur utilisation est largement simplifiée par rapport au starter-kit.

Enfin, j'ai choisi NestJS parce qu'il propose une intégration des micro-services en simplifiant la gestion des échanges par messages ou événements entre micro-services grâce aux annotations et aux décorateurs. De plus, il propose un large choix de technologies pour la gestion de ceux-ci telles que **Nats**, **RabbitMQ**, **MQTT** ou même **Kafka**.

Avant de passer définitivement au nouveau framework, il a d'abord fallu que j'estime le temps que la conversion de Ts.ED à NestJS prendrait. Comme c'était la première fois que j'utilisai ce framework, j'ai alors simplement repris un des micro-services existants pour y intégrer NestJS. Grâce aux ressemblances des deux packages, la conversion fût très rapide et ce fût un nouvel argument pour convaincre le référent technique et le chef de projet.

### 6.3.2. La séparation en « packages »

Le starter-kit actuelle implémente beaucoup de fonctionnalité de base. Voici lesquelles :

- Un middleware qui autorise, ou non, un utilisateur à accéder à une route et qui décrypte un JsonWebToken afin de récupérer ses informations.
- Un service permettant de gérer la création, la modification et la suppression des utilisateurs.
- Un service permettant la gestion des bases de données MongoDB ainsi que des classes prêtes à être utilisées pour les différents modèles de données.
- Une gestion d'envoi de mail.
- Un service permettant de créer et gérer des tâches planifiées.
- Une gestion de version pour les APIs.
- Un service fait maison qui « log » des informations dans la console de développement, un « logger ».

L'objectif est donc de séparer toutes ces fonctionnalités, soit en packages, soit en micro-services directement. Je vais m'attarder ici sur les packages.

Un package, ou un module, est une petite « application » contenant des fonctionnalités que l'on peut réutiliser dans n'importe quel projet. En général, tous ces modules sont disponibles en téléchargement sur le site **npmjs.com** ou alors en utilisant le gestionnaire de packages installé par défaut avec NodeJS : le **Node Package Manager** (npm). Une simple commande « `npm install <nom du package>` » installe le module que l'on souhaite, dans la version de notre choix. Par défaut, les modules sont téléchargés et installés depuis le repository (dépôt) global de NodeJS. Mais il est possible de créer notre propre repository interne afin que seuls les utilisateurs ayant accès à ce dépôt puissent utiliser les modules. C'est ce qu'Amiltone a fait et j'ai eu la responsabilité de publier les premiers packages.

J'ai alors dû différencier les fonctionnalités qui devait être exportées dans des modules à part de celles qui allaient être convertie en micro-services. De plus, en changeant d'architecture, beaucoup de fonctionnalités sont amenées à disparaître des micro-services et c'est l'API Gateway Kong qui en aura la charge. Voici ma démarche.

J'ai d'abord identifié les différentes caractéristiques de chaque fonctionnalité pour ensuite les catégoriser en fonction celles-ci :

#### ◆ *Gestion de base de données MongoDB*

- **Réutilisée** dans quasiment toutes les fonctionnalités
- Un micro-service qui utilise MongoDB en est **dépendant**, c'est-à-dire qu'il ne fonctionnera pas sans la fonctionnalité



- **Ne stocke pas de données** sur une base de données
- Dans le code, ce sont juste **des classes et des fonctions utiles**

#### ◆ *Gestion d'envoi de mail*

- **Réutilisée** par quelques fonctionnalités
- Les micro-services qui ont besoin d'envoyer des mails **ne sont pas dépendant** de cette fonctionnalité. Un exemple : si après la création d'un nouvel utilisateur celui-ci ne reçoit pas de mail pour l'en informer, techniquement il pourra quand même accéder à son application
- **A besoin d'une base de données** pour stocker le journal des événements (savoir si un mail a déjà été envoyé ou non)

#### ◆ *Le logger*

- **Réutilisé** dans tous le code
- Les micro-services qui l'utilisent ne sont **pas forcément dépendant** du logger puisqu'il a pour but de faire du débogage pendant le développement
- **Ne stocke pas de données**
- Dans le code, ce sont juste **des classes et des fonctions utiles**

#### ◆ *La gestion des utilisateurs*

- **N'est pas réutilisée** dans les autres fonctionnalités, celles-ci n'ont pas besoin de la gestion des utilisateurs pour fonctionner
- Dans le code, possède des contrôleurs, des services et des modèles (pattern MVC) : **est donc une API à elle toute seule**
- Est **complètement indépendante** des autres fonctionnalités
- **A besoin d'une base de données** pour stocker les informations des utilisateurs

#### ◆ *L'authentification*

- **N'est pas réutilisée** dans les autres fonctionnalités, celles-ci ne dépendent que d'un token que l'authentification fourni
- **Indépendant** des autres fonctionnalités
- Dans le code, **est un service** à part entière

#### ◆ *Les tâches planifiées*

- **Réutilisées** dans quelques fonctionnalités
- **A compléter**



L'objectif d'un package est d'éviter la redondance de code. Celui-ci ne possède en général que des fonctions et des classes utiles souvent utilisées dans le code. Il n'a pas besoin de stocker de données dans une base de données et chaque package est indépendant des autres<sup>7</sup>. D'après cette explication, j'ai pu facilement sélectionner les nouveaux modules qui seront le **logger** et la **gestion de base de données MongoDB**. Toutes les autres fonctionnalités deviendront des micro-services à part entière.

#### 6.4. Le module de connexion

Anciennement liée à Firebase, il a fallu trouver une solution pour répondre à notre problème d'authentification à l'Active Directory d'Amiltone. Après avoir passé presque six mois sur une solution CAS<sup>8</sup>, le *lead developer* est envoyé en mission sans avoir pu terminer le projet. En tant que développeur junior, il semblait compliqué pour moi de reprendre son travail là où il en était, et de toute façon, il n'était pas concevable que je repasse plus de six mois à comprendre ce qui a été fait.

Le référent technique m'a alors demandé d'utiliser la norme OpenID Connect. OpenID Connect est un standard d'identification se positionnant au-dessus d'OAuth 2.0<sup>9</sup>. La solution peut être mise en place dans beaucoup de langages puisqu'il existe énormément de packages certifiés par les équipes d'OpenID elles-mêmes. Cette solution est donc simple à mettre en place et sécurisée pour nos APIs et pour les utilisateurs. Voici son fonctionnement :

- Notre API reçoit une requête non authentifiée et redirige l'utilisateur vers une page de connexion spécifiée dans la configuration d'OpenID Connect.
- Le service essaye d'identifier l'utilisateur en demandant à un fournisseur d'identité. Il est possible d'activer d'autres solutions de connexion telles que l'authentification à double facteurs.
- Si l'identification s'est déroulée sans problème, le service OpenID Connect peut soit renvoyer un token chiffré (JWT), soit un code. La suite des étapes par le principe que le service renvoie un code.
- Notre API envoie alors ce code au service OpenID Connect pour obtenir le token de l'utilisateur connecté.

---

<sup>7</sup> N'est pas totalement vrai, un package peut importer d'autres packages comme le font des applications classiques. Mais dans notre cas, une fonctionnalité qui sera transformée en module n'est pas dépendante des autres fonctionnalités du starter-kit.

<sup>8</sup> CAS, qui signifie Central Authentication Service, est un système d'authentification unique. C'est-à-dire qu'un utilisateur qui se connecte sur un site web sera connecté sur tous les sites web utilisant le même serveur CAS.

<sup>9</sup> OAuth 2.0 est un framework d'autorisation permettant à une application tierce d'accéder à un service web.

- Notre API retourne alors le token à notre utilisateur qui pourra maintenant faire ses requêtes sur notre API.

L'utilisation de cette norme va permettre de centraliser l'authentification sur les projets internes d'Amiltone. En effet, il ne sera plus nécessaire de s'occuper de la partie connexion des utilisateurs, une partie complexe et souvent redondante sur les projets. Enfin, Google, Microsoft et Orange utilisent OpenID Connect ce qui confirme la notion de sécurité et fiabilité de la solution.

Je vais dans un premier temps parler du fournisseur d'identité. Celui-ci est le service qui authentifie notre utilisateur et nous renvoie ses informations, informations qui seront chiffrées dans le token renvoyé à l'utilisateur. Comme expliqué dans les contraintes du projet, l'utilisateur doit pouvoir utiliser ses identifiants de l'Active Directory pour se connecter, et ses informations doivent venir de l'application Collab (**dire que Collab n'est pas encore prêt dans les contraintes du projet**). J'ai alors dû créer un fournisseur d'identité personnalisé qui permettra ces actions et j'ai décidé d'utiliser le NodeJS pour cela, avec le package **node-oidc-provider**. Celui-ci est certifié par OpenID et je l'ai trouvé dans la liste des packages proposés sur leur site.

Après avoir lu la documentation du package, je me suis rendu compte que celle-ci s'adressait à un public qui maîtrise déjà l'OpenID Connect ainsi que toutes les normes d'authentification avec les JWT, les JWK, etc... J'ai alors dû réaliser un véritable travail de recherche sur son fonctionnement. Nous avons ensuite décidé d'utiliser le mode « Implicit » d'OpenID Connect, c'est-à-dire que le fournisseur renvoie directement un token chiffré à l'utilisateur au lieu de lui renvoyer un code.

Une fois réalisé, j'ai dû travailler main dans la main avec l'équipe du Service Informatique (SI) d'Amiltone pour créer et tester l'authentification à l'Active Directory. Pour l'intégrer dans mon fournisseur d'identité, j'ai utilisé le module **ldapjs**, « ldap » étant le protocole de communication avec l'Active Directory.

Je vais maintenant parler de la mise en place de l'OpenID Connect sur le projet AmilApp. Comme expliqué plus tôt dans ce mémoire, notre architecture micro-service intègre une API Gateway avec Kong. Celle-ci prend donc en charge les plugins et il existe bien évidemment un plugin open-source pour gérer OpenID Connect à notre place : **kong-oidc**. Celui-ci redirige automatiquement l'utilisateur vers une page de connexion. Une fois authentifié, lorsque l'utilisateur nous envoie une requête avec son token chiffré, kong-oidc le déchiffre et redirige la requête vers notre API en y ajoutant les informations de l'utilisateur dans l'en-tête de celle-ci. Chaque micro-service n'a donc plus besoin de gérer l'authentification.

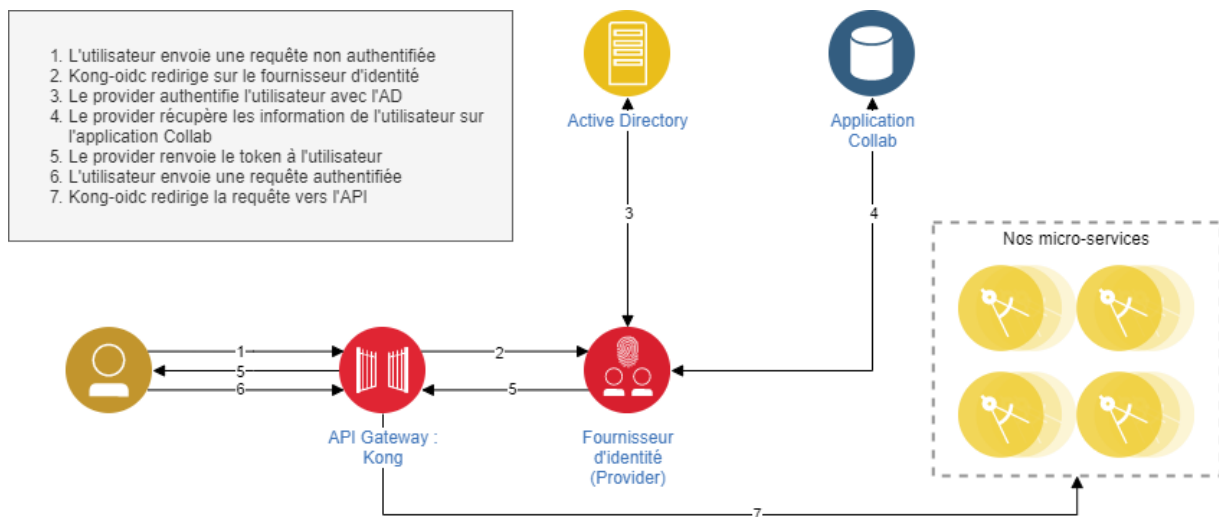


Schéma représentant les étapes de l'authentification OpenID Connect

## 7. Démonstration d'une originalité dans l'élaboration et la mise en œuvre de la solution : [Titre perso]

### 7.1. Une solution modulaire et évolutive

Le projet AmilApp est une application extrêmement **modulable** grâce à son architecture **micro-services**. Il est alors très simple d'y ajouter de nouvelles fonctionnalités sans risquer de compromettre l'intégrité de l'application. De plus, l'indépendance de chaque micro-service garantit une certaine fiabilité et les modifications des uns ou des autres ne perturbera pas le bon fonctionnement de chacun.

AmilApp possède une structure ainsi qu'une configuration **réutilisable** sur d'autres projets, nouveaux ou existants. L'API Gateway Kong préconfiguré permet aux équipes de développeurs d'être focalisées sur les fonctionnalités qu'ils doivent implémenter. De plus, les images Docker créées pour les services d'authentification et de gestion d'envoi de mails par exemple permettent la création ou l'adaptation rapide d'une nouvelle application interne chez Amiltone. La compréhension du projet est d'autant plus simple grâce à la documentation auto-générée des classes et fonctions présentées dans le code grâce à NestJS.

Cette structure est aussi très **évolutive** et **adaptable** grâce à l'injection de dépendances proposée par NestJS. Celle-ci permet d'adapter n'importe quelle classe, service ou même n'importe quel package et de l'intégrer facilement dans le projet.

AmilApp est une solution **résiliente** grâce à son architecture micro-services encore une fois car ils simplifient la surveillance de chaque service et garantissent une meilleure robustesse qu'auparavant pour l'application.

La solution est aussi **élastique**, car elle offre la possibilité d'augmenter ou de réduire le nombre d'instances disponibles pour chaque micro-services afin de répondre au nombre d'utilisateurs qui utilisent l'application simultanément.

## 7.2. Des équipes indépendantes

L'architecture micro-services oblige aux équipes de développement d'adapter leur fonctionnement ainsi que leur manière de travailler et de communiquer. Elles doivent travailler conjointement avec les équipes DevOps et **monter en compétence** pour pouvoir intervenir sur le déploiement de leur solution. Les équipes doivent également apprendre à être plus réactives aux changements puisque l'architecture micro-services permet de s'adapter et de réagir facilement aux besoins du client.

Chez Amiltone, il n'y a qu'une seule équipe sur le projet AmilApp, chaque développeur est alors responsable d'un micro-service. Malgré un « turn-over » habituel assez important chez Amiltone, surtout sur les projets internes, AmilApp fut épargné puisque l'équipe est restée fixe. C'est pour moi un avantage pour le développement d'une telle application puisqu'on aurait été contraints à effectuer des passations de connaissances très régulières.

## 7.3. Une interface en ligne de commande ?

Le développement d'une interface en ligne de commande (CLI) permettrait de rendre l'utilisation du starter-kit plus simple et plus intuitive, avec par exemple la possibilité de générer un nouveau micro-service écrit en NodeJS avec le framework NestJS. Celui-ci contiendrait alors la configuration minimale, la configuration de la base de données, du logger, et du bus de message, pour que le service soit fonctionnel et prêt à être envoyé en production.

(à déplacer peut être)

## 8. Analyse de l'approche choisie : [Titre perso]

### 8.1. Résultats obtenus

### 8.2. Analyse du champ d'application de la solution élaborée

### 8.3. Mise en perspective avec d'autres contextes

**Demander aux chefs de projets** (voir brouillon)

## 9. Réflexion sur le stage et le mémoire : [Titre perso]

### 9.1. Auto-évaluation du travail réalisé

Mes collègues ont été pour moi une réelle source d'information, que ce soit ceux d'Amiltone ou ceux de mon école, ils m'ont appris à travailler en équipe. Pendant mon cursus à SUPINFO International University, les nombreux projets de groupes que nous avons effectués nous ont forcé à réaliser un véritable partage technologique. Les cahiers des charges de ces projets étaient stricts et j'ai eu l'occasion à plusieurs reprises de faire un travail de hiérarchisation des tâches, afin d'optimiser les fonctionnalités qu'il était possible de réaliser en fonction du temps disponible pour ce projet. J'ai également eu l'occasion de donner cours aux étudiants des promotions inférieures. J'ai alors dû m'imprégner des méthodes pédagogiques de l'école afin de retranscrire au mieux le contenu du cours.

Au cours de mon alternance chez Amiltone, j'ai travaillé sur des projets très différents en termes de technologies et d'organisation. J'ai parfois été dans une équipe de taille assez conséquente, parfois de taille réduite, et quelques fois j'ai même été seul sur un projet. Grâce à AmilApp, j'ai expérimenté ces trois modes de fonctionnement **et j'ai pu comprendre le problème de la communication interne dans une entreprise de taille moyenne**. Nous étions trois alternants au début du projet, chacun supervisé par un référent technique. Etant tous étudiants, nous avions le même mode de fonctionnement et les projets de groupe de l'école m'ont permis de m'adapter facilement.

Au cours de la seconde année de mon alternance, j'ai eu la responsabilité de prendre des décisions importantes pour le projet, comme la refonte globale de celui-ci. Les chefs de projet ont su me faire confiance et mon alors confié la tâche de faire la revue de code sur d'autres projets internes.

Etant sur AmilApp depuis ses débuts, je suis en quelque sorte devenu le noyau du projet et globalement, je pense avoir permis au projet de garder une certaine stabilité sur son avancement ainsi que sur sa qualité de code.

La rédaction de ce mémoire m'a permis de prendre le temps d'étudier et d'apprécier les technologies que j'ai utilisé pendant mon alternance. Grâce à l'expérience acquise pendant celle-ci, j'ai su prendre du recul sur les technologies apprises, sur leur fonctionnement et sur mes préférences afin d'en avoir un avis plus pertinent qu'à mes débuts. **Enfin, j'ai pu comprendre le problème de la communication interne dans une entreprise de taille moyenne.**

## 9.2. Bilan des acquis sur les aspects techniques, stratégiques et managériaux

Aspect technique : maintenabilité du code, propreté, scalabilité

Stratégique : R&D, choix technique sur le projet et les argumenter, présenter une solution, documentation

Managériaux : gérer une équipe

## 9.3. Perspectives professionnelles en relation avec les compétences acquises

## 10. Conclusion