TEAM D
Nassim Bounouas
Gaetan Duminy
Rudy Meersman
Nikita Rousseau

Rapport de projet: Plateforme Uberoo

Rapport final - 11/11/2018

Introduction

Dans le cadre du cours de *Service Oriented Architecture*, nous sommes amenés à réaliser une solution technologique pour le client *Uberoo*. Au travers du problème que propose cette entreprise factice, nous allons être confrontés au problème d'**intégration** de services.

Uberoo est une plateforme web qui permet à des restaurateurs d'afficher leurs cartes, permettant à des consommateurs de les commander et de les faire livrer à domicile. Dès lors, plusieurs acteurs de systèmes tiers sont mis en relation pour permettre ce service. Il faut donc permettre aux différents systèmes de collaborer entre eux.

Nous avons étudié en cours plusieurs paradigmes permettant à différents services de communiquer, tout en maximisant un couplage lâche entre eux.

Nous allons présenter notre système avec nos choix de conception pour les services ainsi que le bus utilisé pour la communication entre les différents services.

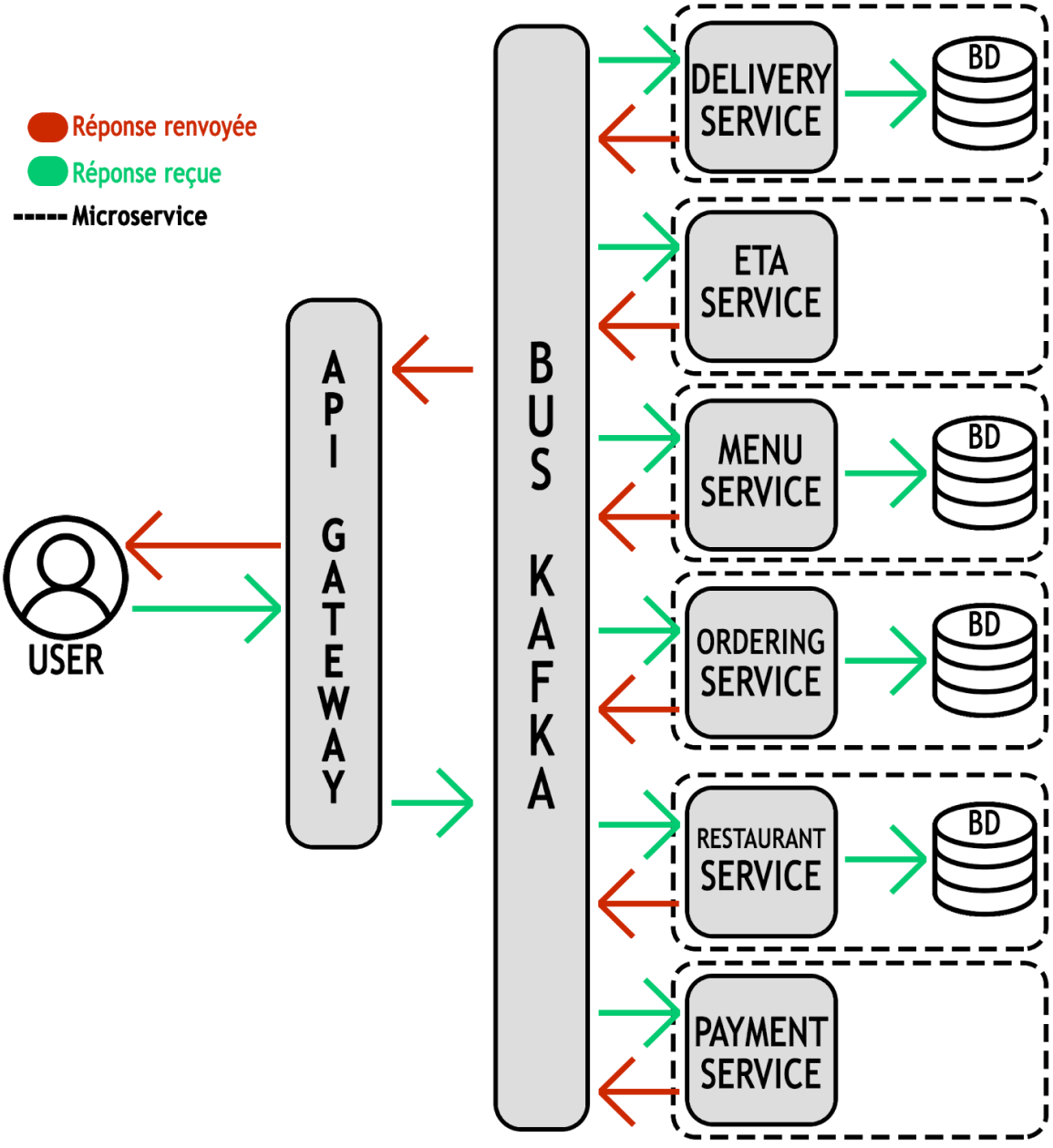
Scénarios Implémentés

Pour ce projet, nous avons implémentés les users stories présentes dans l'annexe nommée ***workflow.pdf.*** Ce document se trouve à la racine du projet et associe les users stories proposées par notre client à un flux d'opérations métier de notre projet (Services impliqués, Messages échangés)

TEAM D
Nassim Bounouas
Gaetan Duminy
Rudy Meersman
Nikita Rousseau

Choix de conception

Architecture globale



Notre architecture suit un découpage *n-tiers*. Chaque micro-service a été construit en fonction des workflows que nous avons définis en commun. Ce processus a été long et nous

a forcé à adopter une convention de description afin de représenter les workflows au format texte.

Plusieurs problèmes sont apparus :

- Quelle était la taille raisonnable d'un micro-service ?
- Quels topics devaient être créés ?
- Quels messages devaient être échangés ?

Nous avons effectué plusieurs itérations sur un document partagé afin de nous assurer qu'il était possible de réaliser un scénario donné. **Ce scénario était généralement un use case définis par notre client.**

Une fois stabilisé, ce document nous donnait plusieurs indications :

- Quels étaient les micro-services
- Quels sont les points de communications (topics)
- Messages échangés et ordre de traitement

Nous avions ainsi un éco-système de micro-services capables de communiquer entre eux au travers d'un bus Kafka. La communication était asynchrone aux travers de messages qui suivent un pattern COMMAND/RESPONSE.
Cependant, il n'y avait pas d'interface à proprement parler pour communiquer avec les micro-services. Nous avions alors pris la décision de construire une API Gateway, un point d'accès unique, qui agrège l'ensemble des actions possibles sur tous les micro-services du système. C'est notre point d'entrée pour interagir avec le système. Celle-ci est sans-état et peut être dupliquée *n-fois* (un répartiteur de charge permettrait ainsi de répartir les requêtes sur plusieurs API-Gateway, chacune étant capable de répondre à l'appel de manière asynchrone).

Une alternative possible à celle-ci est d'avoir chaque micro-service qui expose sa propre API. Notre choix d'unifier les interfaces s'est justifié pour les raisons suivantes :

1) Un unique fichier de documentation à maintenir
2) Une seule application combo serveur web capable d'aussi faire producteur/consommateur sur le bus Kafka à maintenir
3) Abstraction pour le client du coeur du système. Les clients de notre système sont "bornés" aux actions que propose notre API Gateway. On peut ainsi filtrer les entrées et effectuer plusieurs actions sur une unique requête.

API Gateway

Le point d'entrée de notre architecture est une **API GATEWAY** exposant des routes HTTP afin que l'utilisateur extérieur puisse facilement interagir avec notre système. Cette Gateway se charge de transformer la requête HTTP du client en un (ou plusieurs) message(s)

pouvant être publié(s) sur le bus KAKFA, dans le bon topic et ayant un sens pour les services consommateurs.

Notre **API-Gateway** fonctionne de manière purement asynchrone. C'est à dire que chaque requête soumise ne retourne pas une réponse qui contient le résultat de la requête. A défaut, celle-ci va retourner une URL de *callback*, qui permettra au client de scruter si une réponse est disponible. C'est une stratégie qui permet une implémentation simple de l'API Gateway. Cependant, plusieurs inconvénient apparaissent :

- Solution pas optimisée car oblige le client à soumettre sa requête plusieurs fois avant d'avoir la réponse
- Requiert une boucle d'attente active du côté client
- Disparité et hétérogénéité des appels sur l'API Gateway : certains appels sont orientés resources, d'autres sont des commandes. On mélange un paradigme RPC et REST.
- Obligation de faire expirer les résultats des callbacks après une certaine durée, sinon croissance infinie de la consommation mémoire de la gateway.

Cependant, c'est pas gênant dans notre architecture. En effet, l'API Gateway peut être dupliquée n-fois pour absorber la charge. Elle est sans état. Une requête de callback peut être répondue par une autre API-Gateway que celle qui a généré celle-ci.

Persistance des micro-services

L'implémentation de la persistance s'est faite selon le modèle du *"Database per service".* Chaque micro-service nécessitant une persistance des informations se voit doté de sa propre base de données. De cette manière, nos services sont indépendants et ne sont pas liés par une base de données ; cette pratique insiste sur la notion de contexte borné. Deux microservices traitant des données similaires n'ont pas forcément une même représentation de cette dernière. A titre d'exemple : du point de vue restaurateur (service ordering) une commande est l'association d'un client, d'un plat, et d'une heure de récupération alors que du point de vue d'un livreur (service delivery) une commande est l'association d'un restaurant, d'un identifiant de commande, d'un lieu de livraison, d'une heure de prise du repas et d'une heure de livraison.

Ce choix de paradigme est dépendant de notre organisation interne pour le développement du projet. Chaque membre de l'équipe était responsable globalement d'un micro-service en particulier. Ainsi, chaque micro-service est responsable de la modélisation de son sous-système. Cela limite aussi les effets de bord lorsqu'une nouvelle fonctionnalité arrive, qu'une base de donnée est partagée, mais que le schéma sous-jacent est mal géré par les parties coopérantes sur la même base de données.

Delivery Service

Fonction

Le **Delivery Service** va pouvoir gérer les diverses commandes à livrer ainsi que les livreurs. Les livreurs peuvent savoir où se trouve une commande et mettre à jour leur statut.

Justification de la création

Nous avons eu la volonté de découpler l'aspect livraison de l'aspect commande.

Lors de l'arrivée du persona correspondant au livreur, notre choix s'est avéré pertinent puisque nous étions alors capables de gérer le livreur indépendamment du reste du système.

ETA Computer Service

Fonction

Le **ETA Computer Service** calcule le délai estimé pour que la commande soit préparée et livrée. Ce service est aussi interrogé pour mettre à jour le délai de livraison lors du déplacement du livreur.

Justification de la création

Nous souhaitions, ici, éviter une duplication de code ou une mauvaise répartition des fonctionnalités. En effet les services *ordering* et *delivery* ont tous les deux besoin de l'estimation de l'heure d'arrivée. Il nous a donc semblé pertinent de donner cette responsabilité à un service dédié utilisé par les deux services cités précédemment. De plus, si la méthode de calcul de l'estimation venait à changer, la modification serait restreinte à ce microservice sans forcément impacter les autres services.

Menu Service

Fonction

Le **Menu Service** gère la liste de plats en fonction d'une catégorie, donne la liste des catégories et permet d'obtenir les menus associés à un restaurant.

Justification de la création

Lorsqu'un utilisateur utilise la plateforme Uberoo, en premier lieu il doit lister les menus et plats disponibles. C'est même la fonction qui sera la plus appelée et l'une des plus

importantes. Il a nous a donc paru naturel que cette grosse fonctionnalité se voit attribuer un service qui lui est propre.

Ordering Service

Fonction

Le **Ordering Service** reçoit la commande du client et va mettre à jour celle-ci en fonction de l'étape où elle se trouve. De plus, un restaurateur peut demander à ce service la liste des plats qu'il va devoir préparer.

Justification de la création

L'ordering permet de concentrer les commandes des utilisateurs ayant passé la première phase de recherche. Une fois que le client a choisi le restaurant ainsi que son plat préféré, il n'a plus à opérer directement avec le menu. Le but de ce service est donc de gérer la machine à état qu'est une commande passant par les étapes : CREATED - ACCEPTED (Payée) - WAITING (Commande en attente de livraison) - DELIVERING - DELIVERED

Payment Service

Fonction

Le **Payment Service** permet de valider un paiement ou de créditer le compte d'un livreur.

Justification de la création

La création de ce service est arrivée avec l'idée de n'avoir qu'un seul point de communication avec la banque (cette communication est d'ailleurs simulée). Ce service est utilisé à chaque fois qu'une transaction monétaire a lieu. De plus, si on souhaite changer le système de paiement (par exemple des points) celui-ci sera facilement remplaçable par un autre microservice dédié à cela.

Restaurant Service

Fonction

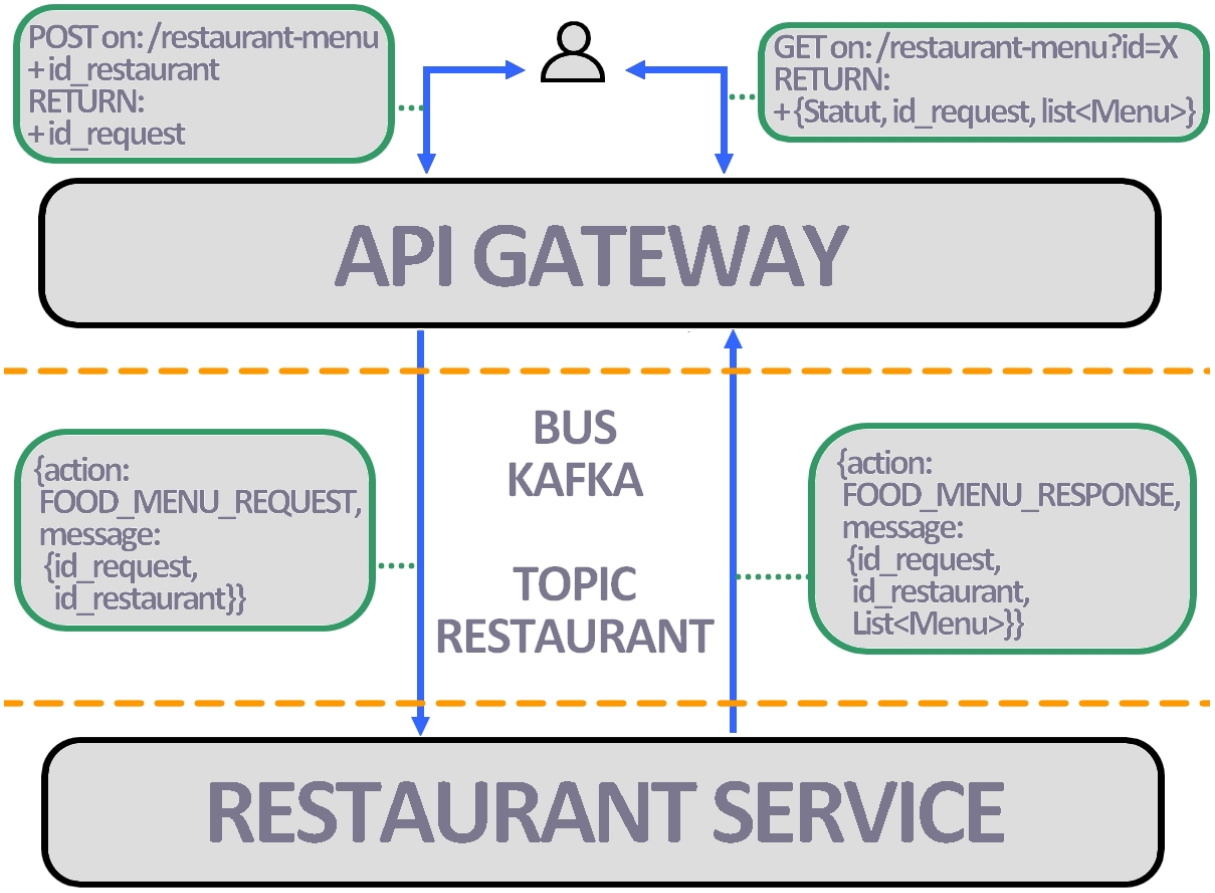
Le **Restaurant Service** qui permet d'obtenir la liste des restaurants associés à Uberoo.

Justification de la création

Lorsqu'un client veut accéder au catalogue d'un restaurant, il passe par ce service. Suivant le même raisonnement qu'en ce qui concerne le Menu service, il nous a paru judicieux de dédier la navigation au travers des différents restaurants à un service dédié. En effet, lors d'une éventuelle croissance de la plateforme, le nombre de restaurant risque de devenir conséquent et la recherche d'un restaurant par zone géographique et/ou type de menu / plat constituera alors un point lourd en terme de complexité.

La technologie Kafka pour le Bus

Exemple d'utilisation de Kafka dans un workflow



WORKFLOW: DEMANDE DES MENUS

Lorsqu'un utilisateur demande la liste de menu d'un restaurant, l'api gateway va envoyer un message au service restaurant et retourner à l'utilisateur le numéro de sa requête. Pour le message destiné au service restaurant, il va créer le json adéquate puis l'envoyer sur le topic restaurant du bus Kafka. Le service va, de son côté, lire le topic et lorsqu'il réceptionne un message, il l'analyse. Dans ce cas, il va lire le message puis créer à son tour un json pour répondre. Ce json est remis au même topic et sera récupérer par l'Api Gateway. Le client va faire un get avec l'identifiant de sa requête afin d'obtenir une réponse sur le statut de celle-ci.

Plusieurs topics pour une meilleure gestion ?

Afin de fluidifier l'information et de pouvoir la gérer de manière plus constructive, il existe plusieurs topics en fonction du type de demande:

- Le topic **restaurant** va pouvoir gérer les messages en rapport direct avec le restaurant tel que les menus, les catégories ou encore les établissements.

- Le topic **delivery** permet la gestion des messages pour la livraison, que ce soit du côté livreur pour chercher une offre, en choisir une et indiquer l'évolution de la commande lors de la livraison, du point de vue client ce topic permet d'obtenir la demande de localisation du livreur.

- Le topic **ordering** s'occupe de transmettre les messages concernant la validation d'une commande par un client ou encore, pour un restaurateur, de récupérer la liste des plats à préparer.

- Le topic **payment** n'est utilisé que lorsque l'on doit débiter l'argent du client en contactant sa banque.

- Le topic **eta** est utilisé lors d'une demande de calcul ou bien de mise-à-jour de celui-ci.

Choix technologiques

Initialement, avant l'introduction du bus Kafka nous avions un ensemble de microservices polyglotte. En effet, nous étions partis sur les langages : Python, Java et Go afin de produire le MVP demandé lors de la première livraison. Du point de vue des différentes bases de données nous avions fait le choix de n'utiliser que Mysql.

Cette seconde livraison présente un grand changement, en effet à la moitié du projet nous avons fait le lourd choix de nous séparer des services codés en Java et Go. Ce choix implique une régression considérable en terme de bases de code et de fonctionnalitées présentes. Ce choix a cependant eu ses avantages ; maintenir un écosystème sur plusieurs langages sur une durée de projet aussi courte (~ 6 semaines) et avec un effectif de 4 membres est une très mauvaise idée mise à part dans le cas où les membres sont assignés à un service et en assure la direction.

Cette approche était la nôtre au début du projet et il s'est avéré qu'après quelques semaines tout le monde touchait à l'ensemble des services compte tenu des délais à respecter et le maintien de l'ensemble des services s'est avéré très chronophage.

Arrivé à un niveau tel d'étranglement (User story s'ajoutant chaque semaine, problèmes liés à kafka, problèmes liés à l'intégration) que nous savions plus par quoi reprendre notre solution, nous avons fait le choix de faire table rase sur une grande partie du projet.

Un point très important de ce projet a été la documentation. Nous regrettons qu'il n'existe pas d'outil dédié à la documentation des évènements kafka. Nous avons néanmoins pris soin de documenter ces derniers via un traitement de texte, nous permettant de tracer les fluxs des messages en fonction des users stories.

En ce qui concerne l'API Gateway, celle-ci a été documentée au moyen de l'outil swagger et vient lors de la livraison avec un fichier permettant dans postman de d'exécuter les différents contextes d'utilisation de la plateforme.

Les Tests

Notre projet souffre de plusieurs maux :

- Manque de tests unitaires
- Pas d'intégration continue
- Tests fonctionnels tardifs et imparfaits

Plusieurs raisons sont responsables de ces problèmes. D'une part, nous avions beaucoup de difficultés à créer le code métier. Nous devions utiliser des bibliothèques et de nouvelles technologies (introduction à Docker pour certains membres de l'équipe, Kafka pour l'ensemble de l'équipe, mais aussi Flask et les clients Mysql/Kafka).

L'absence de tests unitaires faisait que le code n'était pas fiable. Nous ne pouvions pas faire confiance au code des différents micro-services, que ce soit celui dont nous avions la maintenance, tout comme ceux de nos camarades.

L'intégration continue devait être une priorité dans ce projet. Cependant, par manque de temps et par soucis logistique purement matériels (serveur dédié instable avec accès limité), cet axe n'a jamais été implémenté dans le projet. Cela a constitué une grosse perte de temps car nous avons dû réaliser celle-ci à la main dans la dernière semaine de projet.

L'absence de tests unitaires a été pointé mais en réalité, le retard dans les tests fonctionnels nous a le plus pénalisant. Les tests fonctionnels permettent dans le cas de microservices de remonter des problèmes relevant à la fois du fonctionnel et du non fonctionnel. De part leur nature ces tests font office de test d'intégration et permettent dès lors de détecter des problèmes relatifs à ladite intégration.

A titre d'exemple, nous avons durant très longtemps fait appel à la fonctionnalité de validation des commandes à la main alors que ceci devait être automatiquement fait par le

paiement d'une commande. Ce n'est que tardivement que nous avons pris conscience de ce genre d'erreurs et il était déjà trop tard pour le résoudre (delta entre les scénarios validés et les scénarios à implémenter trop important).

Test de Charge du système

Nous avons réalisé un test de charge sur la récupération des catégories ainsi que sur le passage de commande. En effet, ces deux points sont pour nous les plus critiques, l'afflux des clients se fera sur la recherche des plats disponibles et la charge sur les requêtes suivant ce point sera inférieur.

Il aurait été intéressant de charger un scénario complet jusqu'à livraison. Cependant notre API GATEWAY est un réel goulot d'étranglement : pour offrir un contexte de test de charge crédible, il nous aurait fallu remplacer son serveur web (flask) par un réel serveur WSGI de production comme **Gunicorn**.

Les problèmes rencontrés

Nous avons été confrontés à de nombreux problèmes ayant induit beaucoup de retard dans le développement du projet.

Au sein de l'équipe, nous avions des connaissances communes en Python. Cependant, comme cité ci-dessus, une partie de nos micro-services était écrite en Go. Or, uniquement un membre de l'équipe était capable de maintenir ce code. Nous avons ainsi pris le choix de réécrire ces services.

D'autre part, l'API Gateway était à l'origine écrite en Sprint-Boot (Java-EE). Nous avions rencontré des problèmes de format de message (Marshalling / Unmarshalling) JSON. Étant encore petit, il a été raisonnable de le réécrire. Cependant, c'était de nouveau du temps et de l'énergie perdue.

Enfin, le travail fourni par les membres de l'équipe a été inégal dans le temps. La paternité du code a beaucoup évolué, suite aux nombreuses réécritures. Les deux dernières semaines de projets ont été globalement portées par un effectif d'équipe réduit de moitié.

Un membre dédié de l'équipe a effectué l'inception (faisabilité technique d'une idée), puis proposé une intégration dans la pile technologique existante avec documentation et exemple. Cette tâche dédiée est chronophage et ne permet pas de faire grandir fonctionnellement le projet.

Conclusion

L'évolution de l'architecture à micro-services de ce projet nous a mené à repenser nos services et d'en ajouter un nouveau afin de satisfaire les besoins du client. L'intégration de kafka a constitué un réel obstacle dans notre avancée (et se trouve être une des grandes motivations de la réécriture des services dans un même langage). Le découpage initial du projet nous a donné une bonne base cependant celle-ci ne nous a pas épargnés de nombreuses remises en question au fur et à mesure de l'ajout des nouvelles users stories.

Ce que nous retenons de ce projet est que la partie la plus contraignante d'une architecture sous formes de micro-services est la partie non fonctionnelle. La documentation est le pilier d'une architecture telle que celle là et le laxisme concernant cette documentation amène vite à un projet où il est très compliqué de se repérer. Nous avons ainsi concentré beaucoup d'effort dans son élaboration au fur et à mesure. C'était notre seul garde-fou dans l'avancé du projet.

D'autre part, afin de garantir une croissance rapide, amenée par un couplage lâche grâce au bus de message et les micro-services, il est capital de construire une suite de teste rigoureuse. Celle-ci doit garantir le code métier du micro-service, son intégration avec le système global et enfin, vérifier que les fonctionnalités clientes sont présentes et suivent un workflow cohérent.