Uberoo – Rapport

Toutain - Fornali - Swiderska – Benza

Master 2 IFI - SOA - Novembre 2018

♦ Sommaire :

I. Approche et difficultés

Afin de pouvoir couvrir le set de story initial tout en prévoyant une évolutivité suffisante, nous avons choisi d'utiliser des technologies populaires sur lesquelles nous pourrions trouver beaucoup de documentation en cas de problème (voir plus bas). Ce fût un choix pertinent puisque, comme vous le verrez plus bas, nous avons rencontré de nombreuses difficultés techniques.

Nous avions tout d'abord mis en place un grand nombre de micro-services où chacun devait réaliser une tâche de granularité très fine. Par exemple nous pensions créer un micro-service dédié à l'ajout de nourriture, un autre à l'édition d'utilisateur et ainsi de suite.
Nous avons dans un premier temps naïvement mis en place une trentaine de micro-services (dont plusieurs agrégateurs). Malheureusement nous ne nous sommes rendu compte après cette mise en place (plusieurs semaines après le début de projet) que déployer autant de services, aussi micro soient-ils, avec la technologie que nous avions choisie rendait l'application beaucoup trop lourde (ici 7 Gigas de mémoire vive).

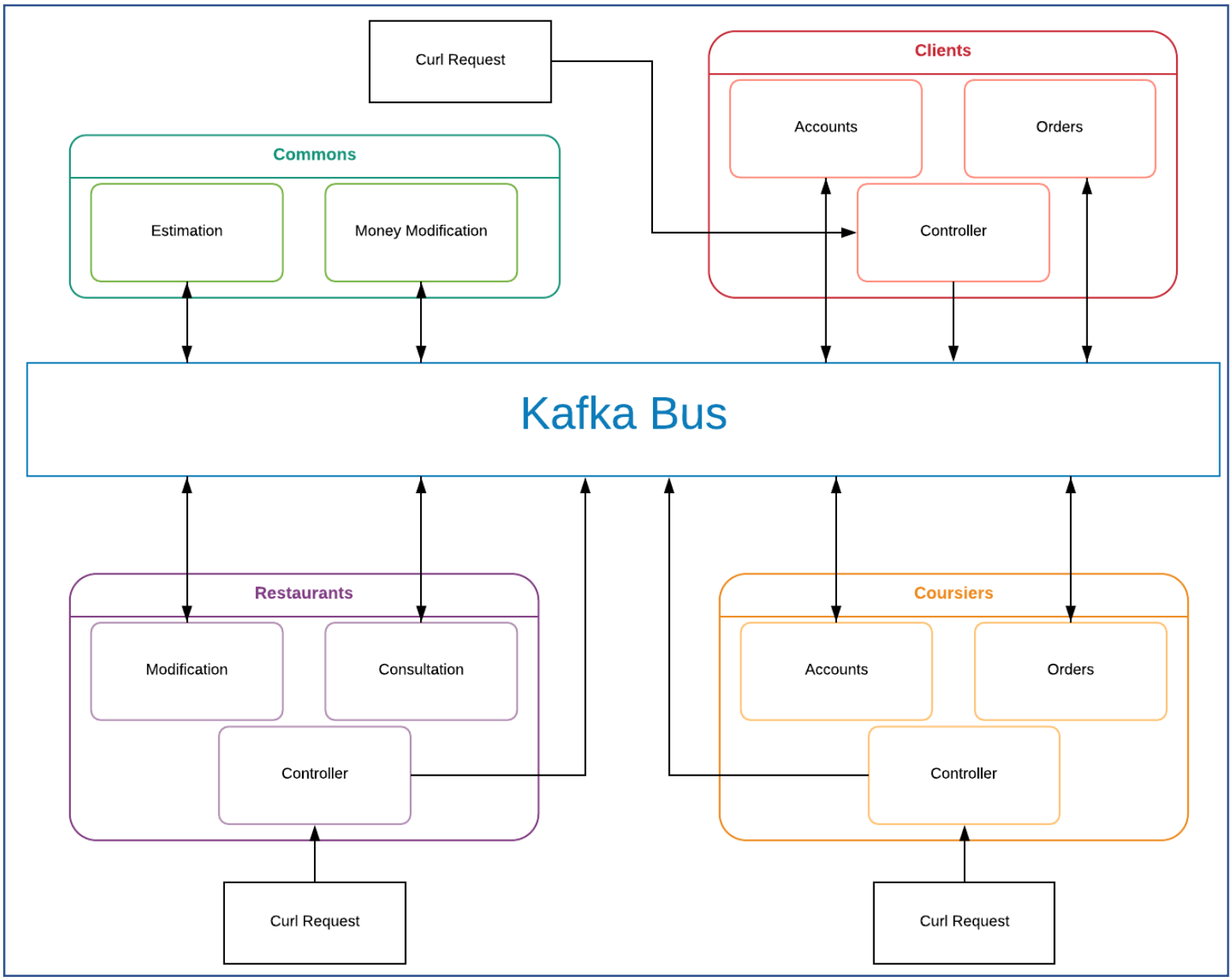
Pour régler ce problème nous avons choisi de réduire le nombre de micro-services ainsi que le nombre de serveurs Web qui seraient exposés aux différents utilisateurs.
Nous avons au final passé énormément de temps à configurer ainsi qu'à mettre en place nos micro-services. Cela a été un frein au plein développement de la logique métier permettant de réaliser les stories demandées.

Cependant, l'architecture mise en place par la suite et présentée ci-dessous permettrait cela sans aucun soucis (voir comment ajouter une fonctionnalité au point *V*).

À cause des problèmes rencontrés, ce que nous pouvons proposer aujourd'hui, est une base solide et évolutive d'**Uberoo** couvrant seulement une partie des cas d'utilisation.

II. Description de notre architecture

Vous trouverez ci-dessous un schéma représentant une vue d'ensemble de notre architecture.



Notre architecture est composée de 15 micro-services, dont 4 agrégateurs :

- **Clients :**
  - Controller : Le premier micro-service référencé par l'agrégateur client reçoit les requêtes faites par un utilisateur (Gail /Erin) et permet d'envoyer des messages au bus Kafka, qui va leur permettre d'être récupérés par les micro-services concernés.

- Accounts : Ce micro-service permet de stocker toutes les informations relatives aux utilisateurs, comme leur adresse, nom, mail, etc... Il permet d'y accéder et de les modifier si besoin.
- Orders : Ce micro-service permet de créer / modifier les informations relatives à l'utilisateur dans une commande.

➢ **Restaurants :**
- Controller : À l'instar du micro-service Controller gérant les requêtes faites au niveau des clients, celui-ci va recevoir celles liées aux restaurants. Il va ensuite les traiter pour faire passer des messages au bus Kafka. Les services écoutants le bus pourront alors récupérer les messages les concernant.
- Consultation : Ce micro-service permet au client (Gail /Erin) de parcourir le catalogue de nourriture proposés par le restaurant, ainsi que d'y accéder par catégories. Il permet également au restaurateur (Jordan) d'avoir accès à la liste des commandes passées à son restaurant, afin qu'il puisse s'organiser.
- Modification : Enfin, ce micro-service permet de mettre à jour le contenu lié au domaine métier restaurant. (Nourriture, informations du restaurant...)

➢ **Coursiers :**
- Controller : Comme les micro-services Controller cités précédemment, celui-ci s'occupe des requêtes liées aux coursiers. Il va recevoir ces dernières puis les traiter afin d'envoyer au bus Kafka les messages appropriés. Ce bus permettra ensuite de faire passer les messages entre les differents micro-services.
- Accounts : Ce micro-service permet de stocker toutes les informations relatives aux coursiers (Jamie), comme leur nom, mail, téléphone, etc... Il permet d'y accéder et de les modifier si besoin.
- Orders : Ce micro-service permet d'accéder et de gérer les informations relatives au coursier dans une commande.

➢ **Commons :**
- Estimation : Ce micro-service a pour rôle de calculer et d'envoyer 2 estimations. La première est une distance : elle permet aux coursiers (Jamie) de savoir quels sont les restaurants à proximité afin de pouvoir sélectionner les plus proches pour effectuer leur prochaine livraison.
- Money Modification : Ce micro-service, quant à lui, permet de prélever l'argent d'un client venant d'effectuer une commande (Erin) pour qu'il soit ensuite reversé au restaurant préparant le plat. (Nous sommes ici partis du postulat que le restaurant s'occupait de payer le coursier pour la livraison, et que ce paiement ne dépendait pas de la plateforme.)

III. Design de notre architecture

1. Technologies utilisées

Nous avons choisi de développer **Uberoo** en utilisant Java/Springboot. Ces deux technologies étant très utilisées et fournies en librairies, elles rendent assez facilement possible l'enrichissement de notre système avec du contenu externe.
De plus, étant donné le faible temps de développement dont nous disposions pour réaliser ce projet, nous avons fait le choix d'utiliser des technologies déjà familières aux membres de l'équipe. Ce choix nous a permis d'économiser du temps que nous avons notamment pu consacrer à l'apprentissage de Kafka.
Afin de stocker les données relatives à **Uberoo**, nous avons fait le choix d'utiliser une base de données NoSQL car le fait d'avoir des données non cohérentes n'était pas critique pour notre application.
La majorité des membres de notre groupe disposant déjà d'une expérience avec Mongo, ce choix nous a semblé évident et le plus simple à mettre en place.

2. Choix de design

Comme déjà mentionné dans les parties précédentes, nous avons fait le choix de découper notre architecture en une dizaine de micro-services communiquant entre eux en passant par un bus de messages Kafka.
Ces derniers sont divisés en differents domaines métiers (restaurants, clients…) afin de permettre la réalisation des cas d'utilisation fournis à la base, tout en permettant une bonne évolutivité.
Par exemple, dans le cas de Gail qui souhaite consulter le catalogue par catégorie de nourriture :

1) Le micro-service **restaurant-controller** va tout d'abord écouter la requête de Gail.
2) Il va ensuite envoyer le message composé du verbe *"consult-category-food"* ainsi que la catégorie à regarder au bus Kafka.
3) Le micro-service **restaurant-consultation** va écouter le bus et entendre cet évènement.
4) Il va à son tour renvoyer l'information demandée par Gail au bus Kafka.

Les micro-services utilisent donc les mêmes verbes de communication Kafka afin de pouvoir s'écouter entre eux.

Le fait de faire communiquer nos micro-services via Kafka permet d'enrichir l'application globale assez simplement. En effet, l'ajout d'un nouveau micro-service ne changerait en rien l'architecture existante puisqu'il suffirait de connaitre et/ou d'ajouter de nouveaux verbes de communications pour permettre des échanges de données entre micro-services.

IV. Tests de Charge

Malheureusement, suite aux difficultés mentionnées plus haut, nous n'avons pas eu le temps de mettre en place des tests de charge.

Si nous en avions, nous aurions tout d'abord cherché à définir les cibles de nos tests. Nous aurions donc sélectionné les modules les plus utilisés (les points chauds de notre écosystème de micro-services). Ici nous aurions notamment privilégié le micro-service servant à la consultation de nourriture, puisqu'il aurait toujours été utilisé en amont de la création d'une commande.
Nous aurions ensuite simulé un grand nombre d'activités afin de vérifier le comportement de notre système en situation de stress.

Afin de réaliser ces tests, nous aurions utilisé **Gatling** un outil performant et fiable, idéal pour le genre de tests que nous souhaitions effectuer.

V. Étapes à suivre pour enrichir l'application

En suivant les indications suivantes, il est simple d'ajouter un nouveau micro-service dans notre écosystème :

1. Tout d'abord, il est nécessaire de créer un module Maven dédié dont l'identifiant doit suivre la convention suivante :
    a. S'il étend une fonctionnalité existante (fe), il doit être identifié par : {fe}-{nom}.
    b. Sinon, un nommage clair indiquant sa fonctionnalité doit être ajouté. Par exemple : clients-statistics ou bien userInterface.
    Le caractère « - » exprimant ici une profondeur de module dans l'arborescence.
2. Le module doit également être nommé de la façon suivante: MSE – {fe} – {nom}.
3. Le nommage de packages doit suivre la convention suivante : com.lama.mse.{nomModule}.{nomPackage}.
4. Si la fonctionnalité ajoutée n'existe pas déjà, il est nécessaire de mettre à jour le pom principal *(module mse -> pom.xml).* Sinon, il faut mettre à jour le pom agrégateur dont le nouveau module doit hériter.
5. Si la fonctionnalité ajoutée souhaite utiliser celles existantes, elle doit connaitre les verbes de communication de ces dernières afin de pouvoir écouter les bons messages transportés entre les modules par le bus Kafka. Sinon, elle doit ajouter ses propres verbes et venir enrichir les fonctionnalités existantes.