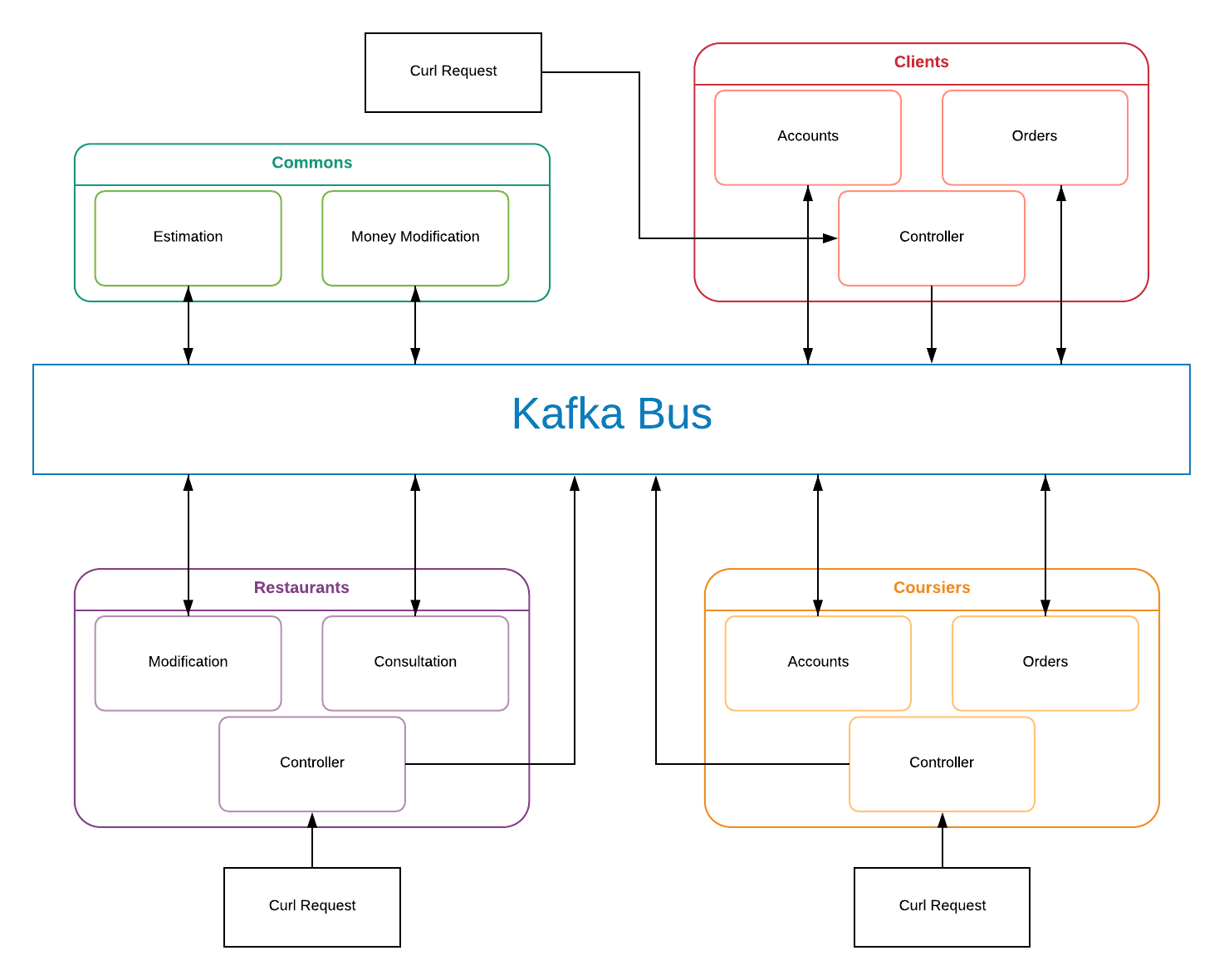
Uberoo - Rapport

***Toutain - Fornali - Swiderska - Benza***

*Master 2 IFI - SOA - Novembre 2018*

* **Sommaire :**

1. **Description de notre Architecture**
2. **Design de notre Architecture**
3. **Technologies Utilisées**
4. **Choix de design**
5. **Tests de charges**
6. **Étapes à suivre pour enrichir l’application**
7. **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 lui-même les faire passer aux services concernés.
* **Accounts :** Ce micro-service permet de stocker toutes les informations relatives à l’utilisateur, 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 une commande passée par l’utilisateur.
* **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 qui va ensuite se charger de les envoyer aux services appropriés.
* **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 :**
* **Accounts :**
* **Orders :**
* **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 sa 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 payait le coursier pour la livraison, et non pas la plateforme.)

1. **Design de notre architecture**
2. **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.

1. **Choix de design**

* Pourquoi cette architecture :
* Pourquoi ces MS ?
* Pourquoi agencés comme ça ?
* Comment les scénarios sont intégrés dans nos MS ? (Qui parle à qui ? Diagramme séquence pour certaines interactions ?)

Nous avons, dans notre architecture, mis en place un bus de messages Kafka permettant de communiquer entre nos différents services.

Ces derniers sont divisés en domaines métiers (restaurant, client…) et utilisent les mêmes verbes de communication Kafka afin de pouvoir s’écouter entre eux.

Le fait de faire communiquer nos micro-services via un tel bus permettrait d’enrichir l’application globale. 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.

1. **Tests de Charge**
2. **Étapes à suivre pour enrichir l’application**

Pour ajouter un nouveau micro-service dans notre écosystème, veuillez s’il vous plait suivre les indications suivantes :

1. Créez un module Maven dédié dont l’identifiant doit suivre la convention suivante :
   1. S’il étend une fonctionnalité existante (fe), il doit être identifié par :

**{fe}-{nom}**.

* 1. 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.

1. Le module doit également être nommé de la façon suivante: **MSE – {fe} – {nom}.**
2. Le nommage de packages doit suivre la convention suivante : **com.lama.mse.{nomModule}.{nomPackage}.**
3. Si la fonctionnalité ajoutée n’existait 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.
4. 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.