Projet SOA – Uberoo

Alexis Couvreur
Julien Lemaire
Johann Mortara
Ivan Picard-Marchetto

Introduction

Ce rapport couvre le travail réalisé par notre équipe dans le cadre du projet Uberoo du module « SOA & Microservices ». Le but était de réaliser une architecture microservices autour d'un sujet traitant de la commande de plats livrés à domicile. Le projet consistait à délivrer un produit minimal et viable répondant aux critères et aux scénarios d'utilisation fournis.

Nous allons présenter ici le travail réalisé pendant la durée du projet, ainsi que justifier nos choix d'architecture pour la réalisation de celui-ci.

Travail réalisé

Sur l'ensemble des stories données par le product owner tout au long du projet, voici celles que nous avons réalisées :

Week 41: Initial Story set

- As Gail or Erin, I can order my lunch from a restaurant so that the food is delivered to my place;
    - En envoyant une requête POST au service uberoo-orders, le service communique à uberoo-deliveries par message l'ID de la commande à livrer.
- As Gail, I can browse the food catalogue by categories so that I can immediately identify my favorite junk food;
    - En envoyant une requête GET au service uberoo-meals en spécifiant un tag en paramètre, la liste des plats possédant ce tag s'affiche.
- As Erin, I want to know before ordering the estimated time of delivery of the meal so that I can schedule my work around it, and be ready when it arrives.
- As Erin, I can pay directly by credit card on the platform, so that I only have to retrieve my food when delivered;
    - En envoyant une requête PATCH au service uberoo-orders pour confirmer la commande, le service communique à uberoo-payment via kafka pour regler la commande.
- As Jordan, I want to access to the order list, so that I can prepare the meal efficiently.
- As Jamie, I want to know the orders that will have to be delivered around me, so that I can choose one and go to the restaurant to begin the course.

- As Jamie, I want to notify that the order has been delivered, so that my account can be credited and the restaurant can be informed.

Week 43: First evolution

New personna: Terry, restaurant owner.

- ~~As Jordan, I want the customers to be able to review the meals so that I can improve them according to their feedback;~~
- As a customer (Gail, Erin), I want to track the geolocation of the coursier in real time, so that I can anticipate when I will eat.
- ~~As Terry, I want to get some statistics (speed, cost) about global delivery time and delivery per coursier.~~

Week 44: Second evolution

- As Terry, I can emit a promotional code so that I can attract more customer to my restaurant.
- ~~As Jamie, I want to inform quickly that I can't terminate the course (accident, sick), so that the order can be replaced.~~

Week 45: Last evolution

- As Terry, I can emit a promotional code based on my menu contents (e.g., 10% discount for an entry-main course-dessert order), so that I can sell more expensive orders.
- ~~As Gail or Erin, I can follow the position of Jamie in real time, so that the food ETA can be updated automatically~~

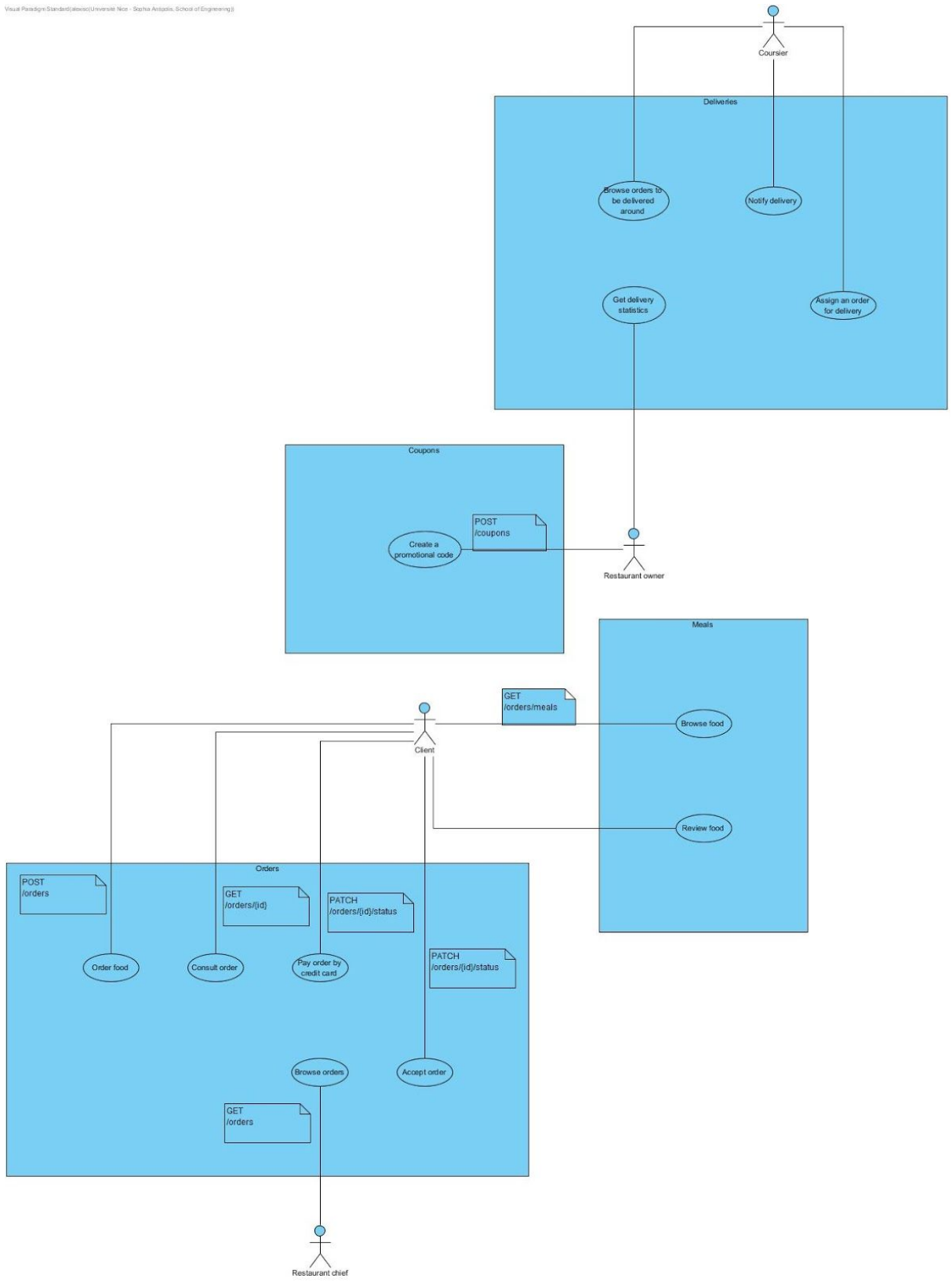
Le choix des différentes stories réalisées est essentiellement basé sur leur criticité apparente en terme métier pour l'utilisateur final (personas Gail et Erin). Le but aura donc été d'accorder davantage de priorité aux stories permettant le bon déroulement de la commande, du paiement et de la livraison d'un plat au client, laissant ainsi de côté les fonctionnalités

Choix d'architecture

Nous avons décidé de découper notre architecture selon les services suivants :

- `uberoo-meals` qui sert à la récupération de plats, potentiellement avec un filtre par tag sur les plats. Ce même service gère également le calcul du prix des plats, en prenant en compte l'application (ou non) de coupons de réduction.
- `uberoo-orders` qui sert la création et la validation ou non des commandes. Ce service permet ainsi à l'utilisateur de passer commande d'un plat et de valider sa commande.
- `uberoo-payment` (non fonctionnel) qui gère le règlement des commandes validées.
- `uberoo-deliveries` qui gère les livreurs, l'attribution des commandes à un livreur et la finition des commandes.
- `uberoo-geolocation` qui permet de se renseigner quant à l'emplacement des différents coursiers à un instant donné.

Nous avons conçu de sorte à minimiser au maximum la dépendance entre les différents services afin d'éviter une surcharge de messages qui arrivera avec un grand nombre d'utilisateurs.



Styles de services

Nos services exposent tous une interface orientée ressources, se soumettant aux contraintes définies par REST. Le principe était d'exposer des ressources définies de manière unique par des URI et d'offrir des opérations basées sur les opérations de base du protocole HTTP (GET, POST, PUT, PATCH, DELETE).

Les ressources étaient ainsi accédées en écriture et/ou en lecture (en fonction des fonctionnalités proposées par la ressource en question). Le but n'était pas ici d'appliquer le modèle CRUD sur chacunes des ressources disponibles, mais de ne réellement rendre accessibles que des opérations ayant un sens métier en accord avec les besoins utilisateurs.

Chaque retour de commande, s'il est correct, apporte des liens supplémentaires dans un attribut `_links`, afin de pouvoir explorer le reste des opérations associées à une ressource. Cet ajout est réalisé dans le but de correspondre à la contrainte REST d'interface uniforme, et plus particulièrement d'*Hypermedia As The Engine Of the Application State* (HATEOAS). Un service est donc explorable à l'aide des liens hypermédias fournis par les différentes ressources accédées. Ce choix permet de rendre l'accès aux ressources moins dépendants de l'utilisation d'URI fixes et par une exploration se rapprochant davantage d'une logique métier (exemple : accès à un lien enregistré sous un nom `meals` plutôt qu'à une URI fixée pointant sur la collection type `http://localhost:8080/meals`).

Communication entre services

Le service `uberoo-meals` ne communique que par REST avec l'utilisateur sans interaction avec les autres.
Le service `uberoo-orders` communique via kafka au service `uberoo-payment` (non fonctionnel) et communique par REST avec l'utilisateur.
Le service `uberoo-deliveries` communique via kafka au service `uberoo-orders` pour confirmer que la commande est livrée et communique par le biais d'une API REST avec l'utilisateur.
Le service `uberoo-payment` (non fonctionnel) communique via kafka pour recevoir les données de paiement et confirmer le paiement en envoyant un reçu.

Les services communiquent entre eux par le biais de messages asynchrones, à l'aide du bus de messages Kafka. Le bus permet une communication entre services asynchrone, basées sur l'abonnement à un type de messages et l'envoi de messages avec un type associé. On obtient ainsi une meilleure indépendance avec les services : un service ne réagit à la réception de messages provenant d'autres services, qui eux-mêmes n'émettent des messages qu'à l'exécution d'une action particulière.

Dans l'implémentation actuelle du système Uberoo, ce principe d'envoi de messages est par exemple utilisé par `uberoo-orders` pour signaler aux autres services qu'un client a validé une commande. Ce message est alors consommé par `uberoo-deliveries` pour l'enregistrement de ladite commande en tant que commande assignable à un coursier. Cet exemple est généralisé par l'envoi d'un message à chaque changement de statut d'une

commande par `uberoo-orders`. L'envoi d'un message inutile sur le bus n'est pas coûteux, car il n'est juste écouté par aucun autre service.

Pistes d'amélioration

Il était prévu de mettre en place une gateway, telle que présentée en cours, afin de centraliser les différents points d'entrée du service. Le principe aurait alors été de réaliser un service exposant une API REST dans le même style que décrit précédemment, qui aurait été un agglomérat de l'ensemble des endpoints des autres services. Le service gateway aurait alors agi à la fois en tant que serveur (pour recevoir ces requêtes HTTP) mais aussi comme client, pour rediriger ces requêtes sur les services correspondants avec les bons endpoints.

Au premier abord, la mise en place de ce service peut paraître être à la fois un point critique du système et un véritable goulot d'étranglement en termes de performances. Cependant, il aurait été possible, dans une optique de passage à l'échelle, de créer plusieurs instances de ce service gérées par un *load balancer*. Il aurait alors réellement été possible d'avoir un point d'entrée unique pour le système, ce qui aurait grandement simplifié les différentes communications vers celui-ci dans un contexte réel.

Évidemment, il aurait été souhaitable d'apporter davantage de fonctionnalités sur les différents services actuellement présentés, afin de compléter l'ensemble des stories. Nous aurions ainsi aimé offrir la possibilité d'un suivi en temps réel d'un coursier livrant un plat, ou encore la possibilité de récolter des statistiques sur les coursiers.

Résultats du test de charge

Nous avons, comme demandé dans le sujet, pu réaliser un test de charge du système livré à l'aide de Gatling.

Le scénario est le suivant :

- 730 utilisateurs sont injectés suivant une rampe linéaire sur une durée de 2 minutes.
- Chaque utilisateur affiche la liste des plats, et demande l'ETA de 5 plats, avec 5 secondes d'attente entre chaque demande d'ETA.

Toutes les requêtes sont acceptées jusqu'à 730 utilisateurs injectés. Dès 740 utilisateurs, le service refuse certaines connexions. Nous pouvons estimer que le système reste résilient pour une charge d'environ 730 utilisateurs sur une fonctionnalité de base (à savoir l'affichage des plats suivi d'une commande). Nous pouvons estimer que ce nombre peut être étendu par la multiplication des instances des différents services composant le système, le tout géré par un load balancer, redirigeant les différentes requêtes en fonction de la surcharge d'utilisation des différentes instances.

Conclusion

En conclusion, nous pouvons confirmer la difficulté de travailler sur une architecture de services. Il est nécessaire de se coordonner davantage que dans une application monolithique et de s’accorder sur les communications inter-services.