Rapport Lab Uberoo — EIINA905 SOA & Microservices — Équipe H

Alexis COUVREUR, Julien LEMAIRE, Johann MORTARA, Ivan PICARD-MARCHETTO

Introduction

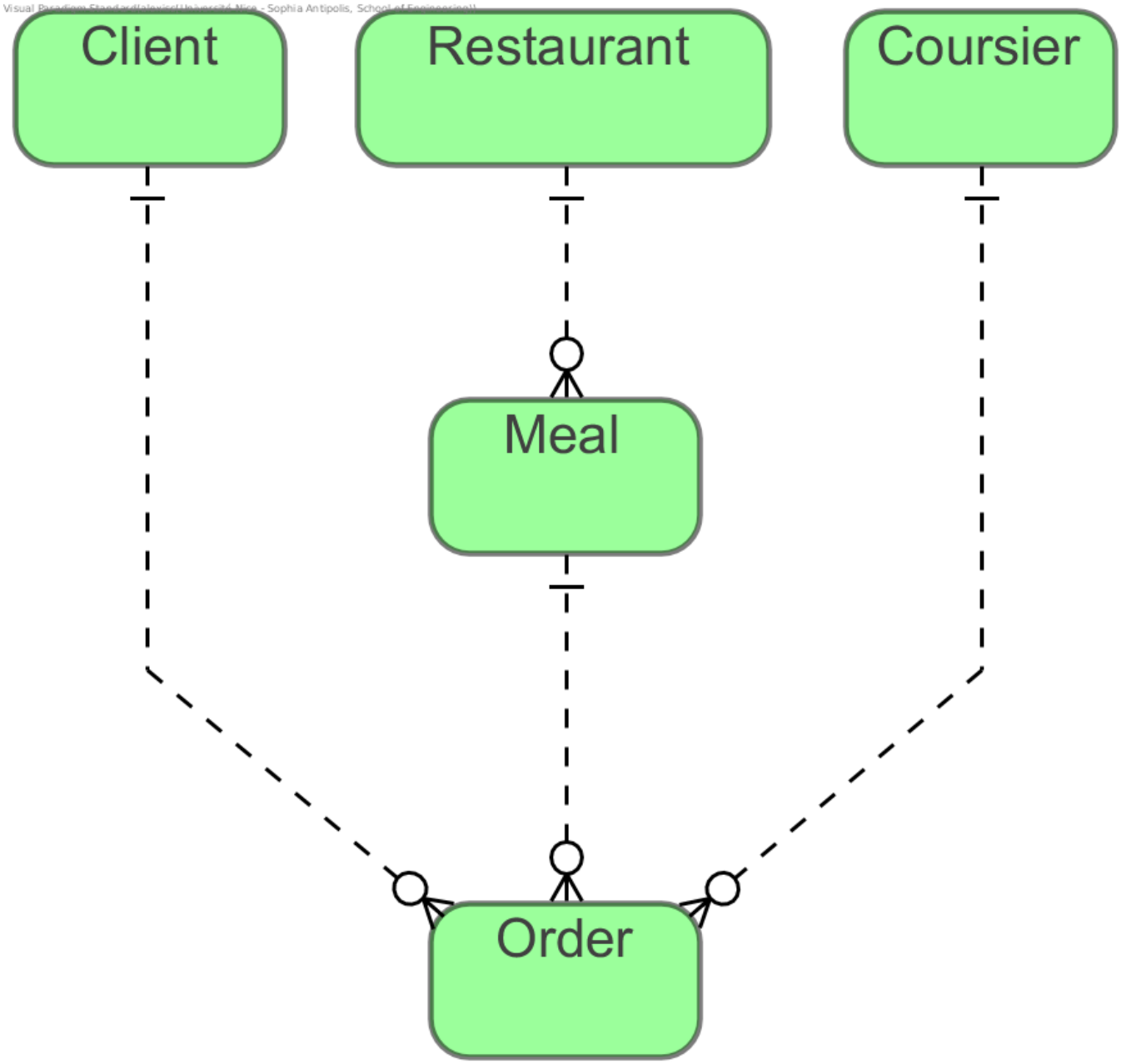
Ce rapport couvre le travail réalisé par notre équipe pendant le travail d'introduction du module « SOA & Microservices ». Le but était de réaliser une architecture orientée services autour d'un sujet traitant de la commande de plats livrés à domicile, sur une plateforme du nom d'Uberoo. Le projet consistait à délivrer un produit minimal et viable répondant aux critères et au scénario utilisateur fourni.

Nous allons présenter et justifier ici nos choix en termes de style d'API ainsi qu'en représentation du modèle métier, avant d'envisager les différences architecturales qu'auraient pu apporter des choix différents.

Modèle métier

Notre modèle métier est composé de 5 entités principales : * Restaurant * Client * Coursier * Plat * Tag * Commande * Statut

Deux acteurs interviennent dans le scénario, Bob, qui veut commander un plat, et le restaurant qui souhaite consulter les commandes à préparer. Lorsque Bob veut commander un plat, il crée donc une commande où sont enregistrés son identifiant client, l'identifiant du restaurant et celui du plat. Lorsqu'il accepte la commande le système peut donc assigner un coursier à la commande.



Choix de style des services

L'application est divisée en deux services : un pour la gestion des repas proposés par les différents restaurants partenaires et l'autre pour la gestion des commandes des clients. Chaque service délivre donc un point essentiel du modèle métier (commandes et plats respectivement). De là est venue la décision de faire des services RESTful, orientés ressources.

Ce choix nous permet d'exposer des opérations simples et explicites basées sur le modèle des requêtes HTTP (`GET` pour la lecture d'éléments, `POST` pour la création d'éléments, `PUT` pour leur mise à jour et `DELETE` pour leur suppression). La sémantique est donc simple à comprendre. De plus, le respect de la contrainte REST d'interface uniforme permet de rendre plus facile la découverte des différentes opérations offertes par le service.

Ce style de service rend ainsi l'architecture du système plus extensible, en minimisant le couplage et en fonctionnant par contrats faibles. Par exemple, les messages auto-descriptifs et le principe de

HATEOAS (*Hypermedia As The Engine Of Application State*) assurent qu'un client de l'application puisse communiquer avec notre architecture de services sans aucune connaissance préalable. Il découvre le système au fur et à mesure qu'il l'utilise, à partir du moment où il sait interpréter de l'hypermédia.

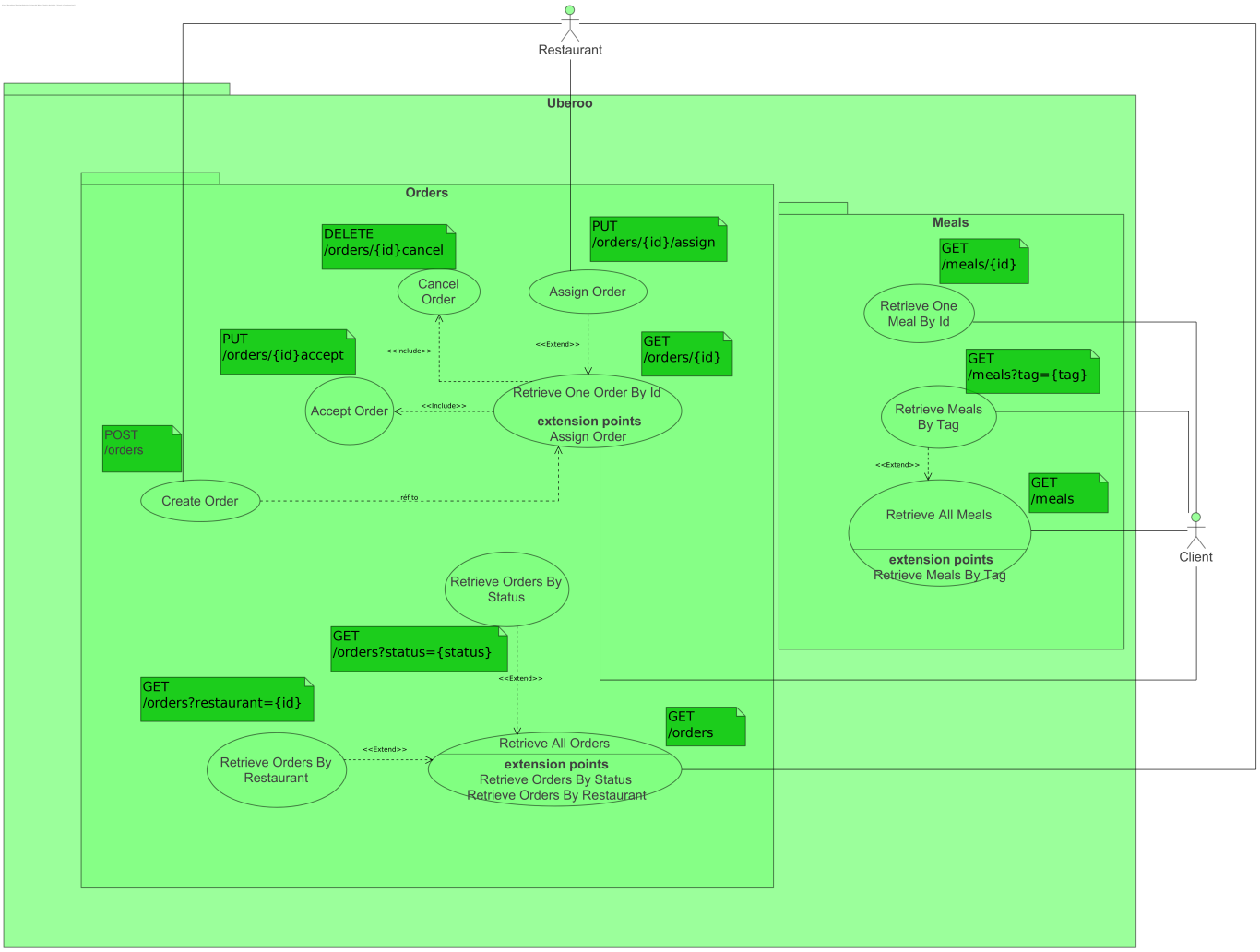
Architecture de services et choix de conception

Comme expliqué plus tôt, le service `meals` sert donc de répertoire pour les différents plats proposés par le service Uberoo. Ces plats sont associés à des restaurants mais également à des étiquettes permettant de les classer dans une certaine catégorie. Dans le cadre du MVP, il n'est possible que de récupérer des informations sur ces plats et non pas de les modifier.

D'un autre côté, le service `orders` permet, grâce aux informations sur le client et le plat que celui-ci désire commander, de créer une commande, de recevoir des informations relatives, par exemple, au temps de livraison et enfin de l'accepter. C'est aussi ce service qui permet à un restaurant de connaître les commandes qu'il doit satisfaire à un instant donné.

Ces services sont articulés autour des deux piliers de la logique métier (repas et commande). Nous nous sommes limités à cette architecture pour le MVP, pensant qu'elle puisse être suffisante pour le scénario demandé. Il est toutefois possible de davantage modulariser le système dans une vision à plus long terme, par exemple en séparant le système de coursiers dans un service à part entière (il est actuellement dans le système de commandes) ou encore faire de même avec les différents restaurants. Une telle découpe permettrait de modulariser davantage le système avec des services potentiellement interchangeables, à partir du moment où ils s'intègrent dans une logique RESTful.

Les deux services n'ont pas besoin de communiquer entre eux pour fonctionner. Ils sont suffisamment indépendants pour que l'on puisse intéragir avec les deux simultanément et ne faire transiter que les informations qui sont strictement nécessaires. C'est par ailleurs comme ça que notre scénario utilisateur est orchestré : par communication alternée entre les deux services.



Impact d'un choix différent de style d'API

Style RPC

Si nous avions choisi de concevoir nos services dans un style RPC, nos services se centreraient davantage sur des opérations de comportements à petit grain. Ce style d'API possède l'avantage d'être beaucoup plus simple à concevoir, mais peut aussi vite tomber dans le piège de la multiplication du nombre d'appels transitant sur le réseau à cause de la granularité fine des opérations.

De plus, ce style d'API impose l'utilisation de contrats forts. Ainsi, les différents clients d'un service exposant une API RPC sont davantage couplés aux méthodes de celle-ci. En d'autres termes, une modification de l'interface publique de ces services induit un plus gros effort de développement, pouvant aller de la simple mise à jour de contrat à l'adaptation complète du client aux nouvelles méthodes exposées.

Cela aurait donc impacté nos services dans la nature des opérations qu'ils exposent et dans leur manque d'extensibilité. Il serait cependant assez simple de convertir les différentes opérations exposées par nos services actuellement en méthodes. Par exemple, la liste des plats à récupérer serait une méthode, tandis que cette même liste filtrée selon un tag précis en serait une autre.

Style Document

Le style document impose des opérations exposées à plus gros grain. Le but est que le client fournisse un certain nombre d'informations nécessaires à l'exécution d'une fonctionnalité globale du système, basés directement sur des besoins utilisateurs. Ce style permet de davantage déléguer le fonctionnement même des opérations au service, contrairement à un modèle RPC par exemple où l'ensemble des appels de méthodes doit être fait par le client.

Notre conception de service aurait été peu impactée dans sa logique par ce choix de style d'API. Le nom des différentes opérations aurait cependant été plus explicite et plus orienté métier (par exemple `CREATE_ORDER` au lieu d'une requête `POST` sur la collection de commandes), ce qui aurait été un des principaux avantages de ce choix. De plus, les requêtes vers un même service seraient envoyées vers un unique point d'entrée, qui se chargerait de rediriger les opérations à plus petit grain vers les bonnes méthodes des bons objets.

On pourrait imaginer également d'implémenter la découverte d'opérations par un type de requête spécifique listant les opérations disponibles. Cela aurait permis de garder l'aspect découverte d'un service RESTful là où un service RPC induit par construction un couplage fort aux méthodes.