RAPPORT D'ARCHITECTURE

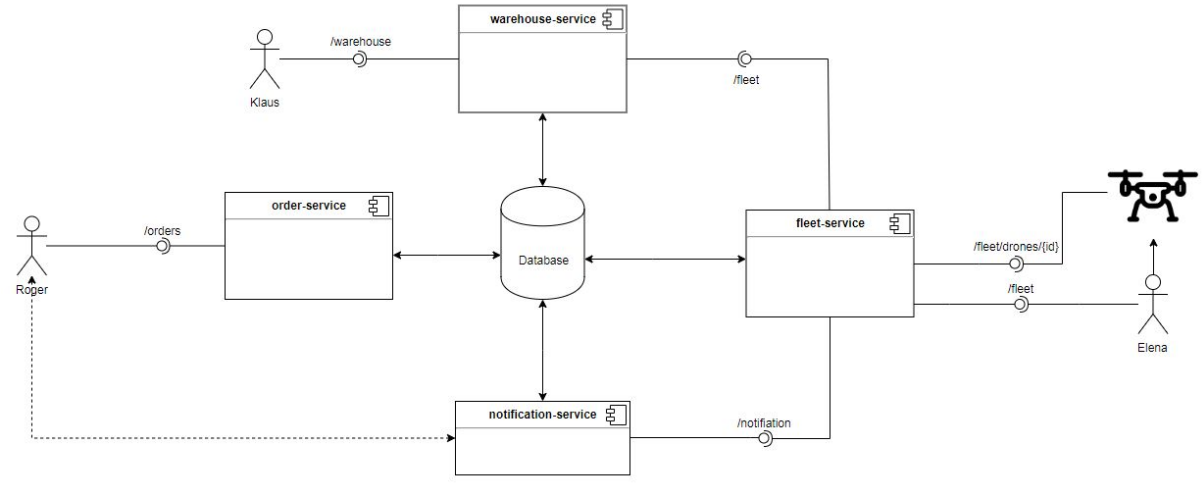
1. L'architecture

1.1. Présentation globale

Notre architecture est composé de 4 services et d'une base de données. Nous avons créé les services en fonction de certains besoins métier :

- service "order" : en charge de la gestion des commandes
- service "warehouse" : en charge de la gestion des articles
- service "fleet" : en charge de la gestion des drones
- service "notification" : en charge de la gestion des notifications

Ces quatres services pointent tous sur une seul base de données, les données sont donc toutes centralisé dans un même endroit.
Même si les services sont responsables d'une partie métier, il peuvent aussi modifier un objet ne faisant pas partie de leur scope. Par exemple les services warehouse et fleet sont en charge de modifier le statut de la commande.
Les échanges entre les services sont effectués par le biais de routes REST.
Nous avons choisi une base de données MongoDB, dont les données sont sauvegardées sous format Json.



1.2. Avantages de l'architecture

Limitations des dépendances directes entre les services. Si le service order n'est plus fonctionnel, le service warehouse peut quand même modifier l'état de la commande. Les échanges entre les services sont limités, ce qui limite l'utilisation des ressources.
L'utilisation d'une seule base de données simplifie le partage des données entre les services. On limite le risque de désynchronisation entre les services.
La base de données de types documents permet de gagner en souplesse, les enregistrement sont simplifiés. De plus, les données échangés entres les services sont au format Json qui est le format utilisé par la base de données pour enregistrer les données.

1.3. Désavantages de l'architecture:

La logique métier lié à la gestion d'une commande n'est pas centré au niveau du service order. Pour faire modifier la gestion d'une commande il faudrait modifier plusieurs services en plus du service order.
Le gain de souplesse fournit par la base de données implique un travail supplémentaire au niveau des services pour éviter d'insérer des données incohérentes.

2. Développement

Nous avons répartis nos tâches par services. Cette répartition a eu pour incidence que malgré l'implémentation des services nous n'étions pas en mesure de les faire communiquer. Nous avons justement eu des problèmes de conception par rapport à la description des objets utilisés par chacun de nos services.
Nous avons eu de nouveaux problèmes lors de l'intégration de la base de données. La base de données MongoDB NoSQL de type Document. Nous avons choisi cette base de données pour simplifier la connexion et la sauvegarde des données. De plus c'était une bonne occasion de pour nous de découvrir l'utilisation de ce type de base de données. Le problème est que la souplesse offerte par ce type de base de données peut être handicapante s'il on est pas rigoureux. Dans notre cas, l'absence d'une description précise de nos objets et de la manière de les sauvegarder a engendré de nombreux problèmes d'intégration. Ce dernier point peut aussie être expliqué par le fait qu'au début du projet nous avions choisi d'utiliser un framework pour gérer la persistance des données et la connexion à la base de données. Nous avons abandonnée ce choix mais nous ne sommes pas revenu sur la manière de gérer la persistance à ce moment là.

3. Etat actuel

Nous avons réussi à implémenter les users story 1,2,3,4,5,6,8 et 9.
Nous avons eu des problèmes pour tester nos services après l'intégration de la base de données. Actuellement, seul le service "notification" est testable par cucumber. Les tests d'intégrations du service warehouse ne sont pas fonctionnel car nous n'avons pas réussi à modifier la base de données à l'intérieur des tests. Le service "fleet" est testable avec l'application Postman par le biais de ses routes.
Tous nos services sont dockerisé et peuvent être lancé à l'aide de docker-compose

3.1. Scénarios

A l'heure actuel nous avons un scénario fonctionnel couvrant les users-story 1,2 et 3.
Le scénario couvre en fait la création d'une commande pour un client jusqu'à la livraison de la commande en passant par chacun des services. Nous avons choisi de dockerisé ce scénario pour pouvoir le connecter aux service lancé par le biais de docker-compose.

4. Rétrospective

Nous aurions préférés implémenter les services à la manière des micro-services, c'est à dire que chaque services est en charge d'un seule aspect métier. Les services warehouse et fleet n'aurait pas à modifier le statut d'une commande dans la base de données mais auraient juste à signaler au service order qu'il doit modifier le statut. Le problème est qu'en choisissant cette architecture nous aurions dû gérer de nouveaux problèmes plus complexes, ce qui aurait entraîné une perte de temps. Le choix de laisser à certains services la possibilité de "déborder" de leur responsabilité nous a paru le plus judicieux par rapport au temps que nous avions pour réaliser l'application.

5. Evolution future

Dans l'état actuel notre architecture présente un défaut majeure au niveau de la gestion de la persistance des données. Il nous faut donc en priorité définir une description précise de nos objets et la manière dont ils seront enregistrés dans la base de données. Ensuite le mieux serait de bien déterminer le scope de chacun des services et éviter que certains modifient des objets dont ils ne sont pas responsable dans la base de données.