RAPPORT SOA

Filipe Costa de Sousa
Cyril Marilier
Romain Garnier
Valentin Ah-Kane

Année 2019-2020

Sommaire

1. Le contexte du projet :

1.1. Rappel des users story

- US#1: Commander un article

- US#2: Empaquetage d'une commande

- US#3: Envoi d'une notification lorsque le colis arrive

- US#4: Traquer le niveau de batterie des Drones pour les mettre de côté et les recharger.
  Dans notre implémentation, c'est à l'utilisateur (Elena) de faire la demande de cette information à un Drone (*/fleet/drones/[id]/battery* et */*[*de*]*activate*).
  Compréhension
  Il est évident qu'en situation réelle, ce qui serait désiré est un monitoring constant des batteries des Drones (cette requête pourrait être faite régulièrement mais il serait moins coûteux d'avoir les Drones eux-mêmes informer de leurs niveaux de batteries à intervalles réguliers), voire une gestion automatique, de la part des Drones, de leur besoin de se recharger, tout en laissant la possibilité à un acteur externe (Elena) de prendre cette décision (potentiellement bien avant que le Drone ait peu de batterie).

- US#5: Rappeler tous les Drones en pleine livraison en cas d'intempéries.
  Fait via la route */recall*, il est également envisageable, comme pour l'US#4, d'automatiser cette détection de mauvais temps en échangeant avec un Service météo et cette route HTTP pourrait être améliorée afin de ne rappeler que les Drones réellement susceptibles d'être pris dans les intempéries (géolocalisation).

- US#6: Envoi d'une notification si le colis n'arrivera pas

- US#8: Accéder à la position GPS d'un Drone en temps réel
  Le Drone envoie sa position à la BD centrale à intervalles réguliers et il est possible de lui demander directement.

- US#9: Historique de télémétrie de Drone
  La télémétrie envoyée par les Drones à intervalles réguliers à la BD est sauvegardée et peut-être consultée à n'importe quel moment, à partir de l'ID du Drone.
  Actuellement, toutes les données sont renvoyées mais, en pratique, un système de pagination serait à mettre en place pour des requêtes plus légères.

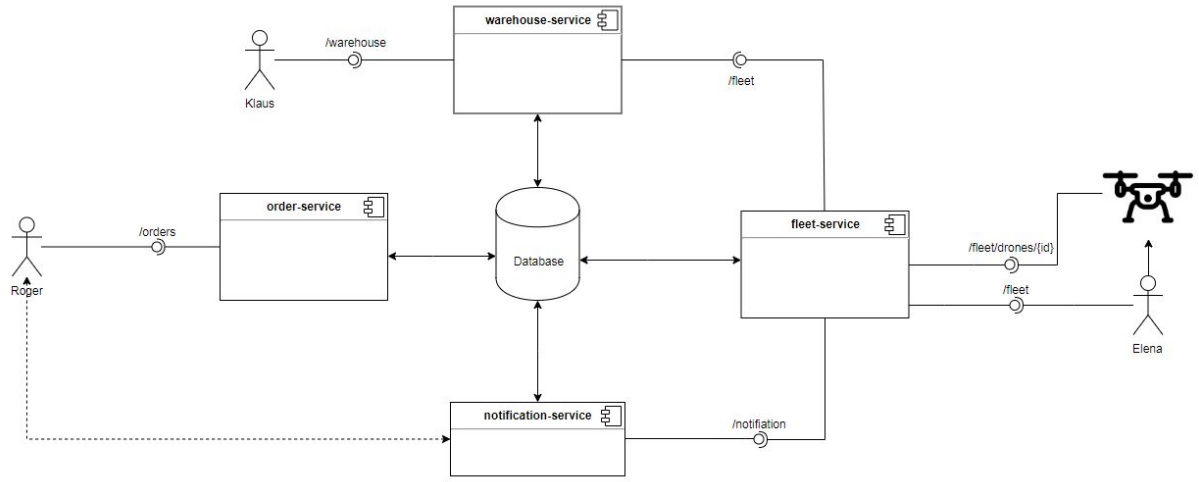
2. L’architecture

2.1. Présentation globale

Notre architecture est composée de 4 services et d’une base de données centrale. 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 seule base de données, les données sont donc toutes centralisées dans un même endroit.
Même si les services sont responsables d’une partie métier, ils peuvent aussi modifier un objet ne faisant pas partie de leur périmètre. 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.



2.2. service “order”

Le service de commande est le service qui sera le point d’entrée pour un client (Roger) lorsqu’il voudra commander un produit de dronizone.
Ce service n’a plus vraiment d'interaction avec d’autres services de notre système après l’introduction d’une base de données commune, mais utilisera des données entrés par le service warehouse tels que les produits disponibles afin de créer une nouvelle commande et l’ajouter à la base de données.
Donc actuellement, ce service permet à un client d’obtenir la liste des produits disponibles dans l'entrepôt et de créer une commande en spécifiant le produit qu’il veut acheter.

2.3. service “warehouse”

Ce service est responsable de la récupération de la gestion des articles. Il permet donc de récupérer la liste des articles, d’en créer et d’en supprimer, grâce à la route “warehouse/items”.
Un article est défini par son identifiant et son prix, nous n’avons pas implémenté la notion d’une quantité d’articles pour une référence.
Il permet aussi à l’utilisateur de renseigner le paquetage d’une commande. Actuellement, l’empaquetage d’une commande n’implique pas de modification du côté des articles étant donné que les articles n’ont pas de quantités associée. Le service warehouse change donc le statut de la commande qui lui a été passé.

2.4. service “fleet”

Manipulé par un “gérant” de drones, tel qu’Elena, dans les User Story (drone management et télémétrie).
Dans notre cas, le service “warehouse” envoie une requête au service fleet pour indiquer qu’un colis est près d’être livré.

2.5. service “notification”

Le service de notification (« NotificationService »), exposé par le composant (« module », au sens de Maven) du même nom, est au sens logique (ici, le sens de Java) une interface qui fournit (actuellement) au composant de gestion (de la flotte) des drones (et pour autant de sortes de requêtes HTTP) deux méthodes :
- notifyThatOrderWillShortlyBeDelivered(String orderId) ;
- notifyThatOrderWillNotBeDeliveredOnTime(String reason, String orderId)...
En effet, voici des scénarios qui, mettant en jeu ces (deux) méthodes, me permettent d’implémenter les « user story » 3, 6 et 7, qui définissent, jusqu’à présent, l’ensemble de toutes les « user story » relatives au service de notification.

2.6. Description des interfaces

La communication avec le service “fleet” se fait par requêtes HTTP en respectant (à peu près) le style REST (*tous les endpoints ne sont pas forcément des collections, par exemple*).
FleetService expose une route /fleet/drones*/assign* pour signaler qu’un colis est prêt pour livraison. Au-delà de ça, les routes sont destinées à manager les Drones et leurs télémétries.

3. Analyse de l'architecture

3.1. Les forces du système

- La force principale que nous avons pu identifier pour notre système est sa simplicité. Etant donné que la complexité de l'architecture est assez réduite, elle reste facilement compréhensible et maintenable si l'on veut par exemple ajouter d'autres routes, communiquer d'un service à un autre ou juste ajouter un nouveau service.

3.2. Les faiblesses du système

- En l'état actuel du projet, étant donné que nous n'avons pas de "microservices" à proprement parler, chaque service n'est pas le seul responsable de ses propres données.
- Les échanges de données sont étroitement liés à la description des objets partagés. Il est nécessaire de connaître la description d'une commande pour pouvoir créer une commande au format Json et l'envoyer au service order pour créer une nouvelle commande.
- Les services sont très fortement couplés entre eux par leurs objets. L'objet "order" est partagé par tous les services. L'objet "item" est décrit par les services order et warehouse, et l'objet "notification" est décrit dans les services fleet et notification.
  Ce couplage a entraîné des problèmes d'intégrations causés notamment par le manque d'une description précise des objets et de l'accord de toute l'équipe.
- La BD centrale peut être source de ralentissements (du fait qu'elle soit centrale, et du fait que plusieurs services l'utilisent pour accéder aux mêmes ressources par moments, ce qui peut entraîner des DeadLock, en plus des ralentissements dus aux Transactions). La BD est donc un single point of failure.
- Les services n'ont pas de gestion d'erreurs. Si une modification dans la base de données remonte une erreur, les services ne gèrent pas l'erreur et ne font que la renvoyer. Par exemple, lorsque le service warehouse demande au service fleet d'assigner un drone à une commande. Si le service fleet ne parvient pas à accéder à la base de données, il renvoi l'erreur au service warehouse qui ne fait qu'afficher l'erreur en console.

3.3. 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 enregistrements sont simplifiés. De plus, les données échangées entres les services sont au format Json qui est le format utilisé par la base de données pour enregistrer les données.

3.4. **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 fourni par la base de données implique un travail supplémentaire au niveau des services pour éviter d'insérer des données incohérentes. De plus le type document ne respecte pas les propriétés ACID d'une base de données relationnelle. Il est donc très compliqué de gérer des transactions au sein de la base de données. Cela augmente le risque d'incohérence entre les données.

3.5. Justification de nos choix d'implémentations

la création d'une commande : nous avons commencé par créer une route permettant de créer une commande en passant une commande en paramètre. Ce choix avait pour conséquence un couplage important avec la description d'une commande au format Json. Nous avons donc ajouter une route permettant de créer une commande en passant un item en paramètre. De cette manière il suffisait de récupérer un item au format Json depuis le service warehouse pour créer une commande sans avoir besoin de connaître son détail. La responsabilité de la création d'une commande revenait entièrement au service order. Malheureusement, nous avons remarqué que cette méthode ne permettait pas de passer l'identifiant du client et son adresse sans multiplier les paramètres. Il aurait fallu remettre en question la description de l'objet order dans le service notification. Dans un souci de temps, nous avons décidé de suivre la description du service notification pour ne pas détruire ce qui avait déjà été implémenté.

4. Développement

Nous avons réparti 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 si l'on n'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 aussi 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 abandonné ce choix mais nous ne sommes pas revenu sur la manière de gérer la persistance à ce moment-là. De plus nous avons eu des désaccord au sein du groupe sur la manière de décrire les objets au sein de la base de données.

5. 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'intégration du service warehouse ne sont pas fonctionnels 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 lancés à l'aide de docker-compose

6. Scénarios

À l'heure actuelle nous avons deux scénarios fonctionnels.
Le premier scénario couvre les users-story 1,2,3 et 4. Il commence par la création d'une commande pour un client jusqu'à la livraison de la commande en passant par chacun des services.
Le deuxième scénario couvre les users story 5,6,8 et 9.
Nous avons choisi de dockeriser les scénarios pour pouvoir le connecter aux services lancés par le biais de docker-compose.

7. Rétrospective

7.1. Ce que nous aurions voulu implémenter

Nous aurions préféré implémenter les services à la manière des micro-services, c'est-à-dire que chaque service est en charge d'un seul aspect métier. Les services warehouse et fleet n'auraient ainsi 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.
Bien sûr nous aurions voulu aborder Kafka, pour avoir au moins une idée générale de son fonctionnement et les gestions des événements. Cela aurait pu nous permettre de remettre en question notre architecture et réfléchir sur comment l'adapter à un fonctionnement orienté événement.

8. Evolution futur

8.1. Pour une application réel

Dans l'état actuel notre architecture présente un défaut majeur 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 périmètre pour chacun des services et éviter que certains modifient des objets dont ils ne sont pas responsable dans la base de données.

Sécurité
Il y a également l'aspect sécurité à prendre en compte. Toutes nos requêtes se font en REST (http) sans aucune forme de sécurité (par d'identification, etc…).

De plus, certaines de nos requêtes ne respectent pas complètement les standards REST et pourraient, pour certaines, gagner à passer en mode RPC/Document (manipulation des Drones / MàJ de la BD / …) —> une requête de tracking peut être en REST, mais quelque chose de plus sécurisé (comme un paiement) sera plutôt fait en respectant un contrat fort, en RPC par exemple.

8.2. Cas à la marge/erreurs

- File d'attente (aucun drone disponible pour livraison)
- Faire remonter le niveau de batterie lors de la recharge / Faire descendre le niveau de batterie lors de livraisons
- Out of stock avant la validation d'une commande (il allait payer mais quelqu'un a commandé le dernier stock) —> annulation de commande car paiement impossible/invalide

Annexe

Diagramme de séquence pour notre scénario