RAPPORT D'ARCHITECTURE

Le sujet est le développement d'une API de services pour la livraison par drone : les drones ont pour avantages, d'une part, de livrer rapidement (car ils n'ont, par exemple, pas de problème d'embouteillage) et, d'autre part, de livrer où l'on veut. (Un drone peut, en effet, accéder à des lieux auxquels un camion, par exemple, ne peut pas accéder.)

Afin de satisfaire au mieux Dronazon, l'entreprise de livraison par drone, c'est-à-dire le client, nous avons commencé par nous intéresser aux "user story". En effet, n'ayant pas trouvé d'autre information sur les besoins des utilisateurs, nous ne voyions pas d'autre manière d'entrer dans le sujet.

1 Les "user story" à un point de vue de haut niveau : première lecture

Pour des questions de temps, notamment, nous avons préféré nous concentrer tout d'abord sur les "user story" 1, 2, 3 et 4. Puis, assez rapidement, nous nous sommes rendus compte que les deux autres "user story", 5 et 6, données en cours de route, ne changeraient pas notre architecture globale.

En effet, comme nous allons le voir, chacune des "user story" 1, 2, 3 et 4 inaugure un service, différent de chacun des trois autres ; en revanche, la "user story" 5, qui met en scène la personne de la "user story" 4, Elena, une responsable de flotte de drones, semble mettre en jeu le même service que le service de la "user story" 4 et la "user story" 6, qui met en scène la personne de la "user story" 3, Roger, un client de Dronazon, semble mettre en jeu le même service que le service de la "user story" 3.

1.1 La "user story" 1

La "user story" 1 met en scène l'utilisation du service de commande (que nous avons choisi d'appeler, alors, "order service") ; nous supposons que ce service sera enrichi par la suite. (Actuellement, nous n'avons pas le cas, par exemple, où le paiement (de la commande) n'est pas effectif.)

1.2 La "user story" 2

La "user story" 2 met en scène l'utilisation d'un service que nous avons nommé, pour le moment, "warehouse service", bien que nous pensions que ce service ne permet de jouer que l'un des nombreux rôles du "warehouse manager". D'une part, ce service permet, en effet, de ne réaliser que l'action qui consiste à prendre connaissance de commandes. (Cette action précède l'opération de préparation de commande, dont la dernière opération est l'emballage.) D'autre part, un "warehouse manager" ("responsable d'entrepôt") est

également responsable de la gestion des stocks (leur réapprovisionnement, en particulier), qui représente une grande partie de ce qu'il a en charge, mais aussi de la qualité des produits, avec la gestionnaire de la flotte (de drones) dans le cadre de notre projet, de l'expédition des marchandises dans les temps voulus, etc.

1.3 La “user story” 3

La "user story" 3 met en scène l'utilisation d'un service de notification au client : “Grâce à ce service, Roger est notifié de ce que le drone (chargé d’effectuer sa livraison) arrivera bientôt.”

1.4 La “user story” 4

La "user story" 4 met en scène l'utilisation d'un service de suivi de la charge de batterie des drones, service que nous avons choisi d’appeler “fleet service” car nous pensons qu’il est l’un des principaux services de contrôle (de la flotte) des drones. Plus précisément, nous imaginons que la flotte est en majeure partie autonome, notamment dans le déplacement des drones ; par conséquent, on devrait n’avoir qu’à se préoccuper du contrôle comme c’est le cas avec la tour de contrôle d’un aérodrome…

1.5 La “user story” 5

La "user story" 5 met en scène l'utilisation d'un service de rappel des drones (au Droniport). Il nous a alors semblé qu’elle pourrait s’inscrire à la suite de la “user story” 4, bien que son énoncé exact parle de rappeler *tous* les drones ; il nous a semblé, en effet, qu’il y avait peu de différence entre le rappel d’un drone et le rappel de tous les drones. En tout cas, cette “user story” est le récit d’un autre cas de gestion des drones et c’est pourquoi nous avons considéré qu’elle mettait aussi en scène l’utilisation du “fleet service”.

1.6 La “user story” 6

La "user story" 6 met clairement en scène l'utilisation du service de notification au client, c’est-à-dire le même service que dans le cas de la “user story” 3 ; mais, cette fois, plutôt qu’une information positive c’est une information négative qui est envoyée au client : “Grâce à ce service, Roger est notifié de ce que le drone (chargé d’effectuer sa livraison) n’est plus en mesure de remplir sa mission.”

1.7 Conclusion de cette première lecture

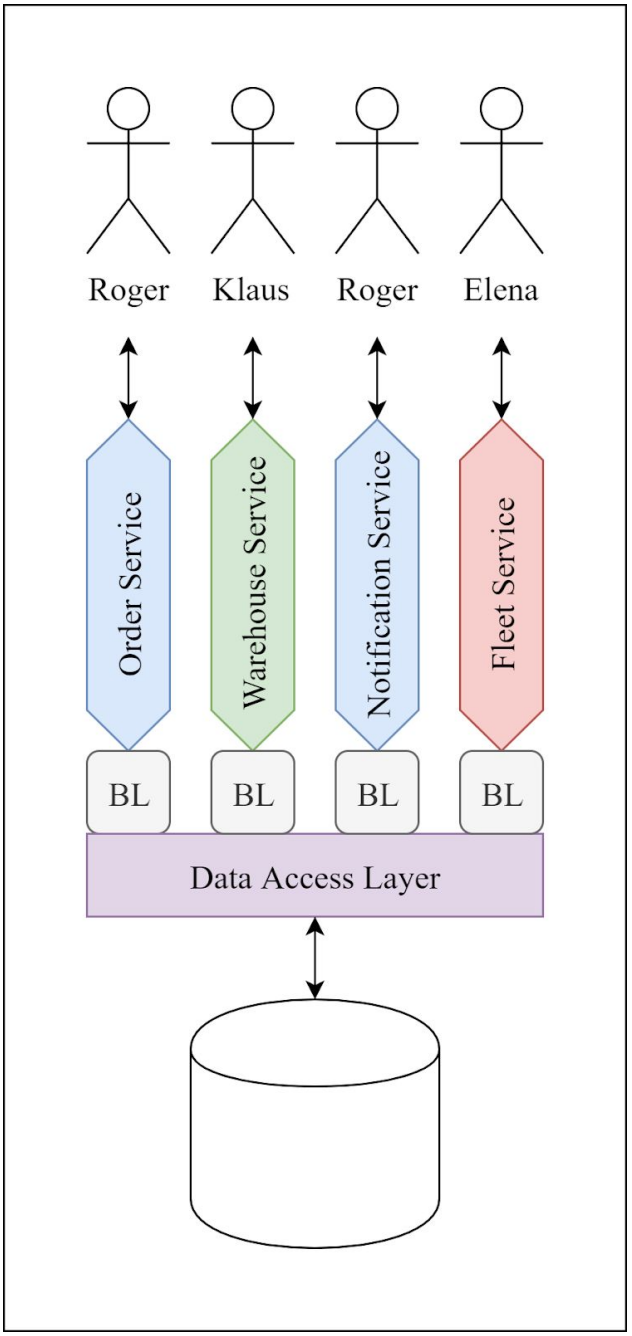
Nous aurions donc quatre services :

1) un service de commande, nommé “Order Service”,
2) un service d’entrepôt, nommé “Warehouse Service”,
3) un service de notification, nommé “Notification Service”,
4) un service de flotte, nommé “Fleet Service”.

1.7.1 Notre architecture dans sa globalité

Suit un premier diagramme de notre architecture, vue à un point de global. Comme nous pouvons le voir, notre architecture est une architecture “trois tiers” ; elle compte en effet :

1) une couche d’UI (au niveau des utilisateurs : Roger, Klaus et Elena) ;
2) une couche “métier” (qui contient les services et leur logique “métier”, désignée par “BL”) ;
3) une couche d’accès aux données.



1.7.2 Des avantages de notre architecture : quelques aspects techniques

Le premier avantage que nous pensons tirer de cette architecture est une grande capacité à tolérer des UI quelquefois très différentes. Par exemple, l’UI de Roger serait, *a priori*, un

simple ordinateur, tandis que celle d'Elena pourrait incorporer un casque virtuel. Nous expliquons cette capacité par le fait que l'interaction, des utilisateurs avec le système, se fait par des services, chaque service étant indépendant des trois autres.

Le second avantage est apparenté au premier : étant donné que la couche d'accès aux données est commune à tous les services et que chaque service est indépendant des autres, jusqu'à sa logique "métier", s'il fallait ajouter (ou supprimer) un service, cela ne devrait pas être problématique pour le reste du système.

En outre, notre projet est un projet avec de multiples "modules" (au sens d'Apache Maven) : un module par service et un module commun (nommé "common"). Le module commun correspond à la couche d'accès aux données : il devrait être entièrement développé avec Hibernate. Hibernate nous permet de développer les accès aux données au niveau "objet" : pour nous, développeurs, il rend la tâche plus facile. Par ailleurs, nous avons fait le choix d'utiliser Spring dans l'ensemble du projet : ce framework nous permet, notamment, de développer l'API d'une manière plus condensée et, dès lors, plus rapide ; cependant, l'utiliser n'est pas toujours évident, une partie non négligeable des mécanismes de Spring étant implicites.

Mais montrons un exemple de ce qui se passe, dans les grandes lignes, au point de vue technique, lors de l'utilisation d'un service. (Nous aborderons cette utilisation en détail, et pour chaque service, dans la prochaine partie.) Par exemple, voyons ce qui se passe dans le cas de la "user story" 1.

1.7.3 Cas de la "user story" 1

Roger consomme *le service de commande* pour commander l'article de son choix :

1) Grâce à la logique "métier" ("BL", pour "Business Logic" en anglais) du service de commande, une instance de la classe (au sens de Java) "Order" est "construite".
2) Cette "construction" est aussi une préparation à la mise en persistance des informations de la commande en question. En effet, nous utilisons Hibernate ; or ce dernier nous permet, en premier lieu de la couche d'accès aux données, un "mapping" entre les attributs de la classe "Order" et les champs du tableau correspondant à cette classe.[1]
3) La couche d'accès aux données, grâce à du DAO, permet, en second lieu, la mise en persistance.[2]
4) Enfin, une fois que cette mise en persistance est terminée, une réponse est retournée à Roger.

2 Les "user story" à un point de vue de plus bas niveau : deuxième lecture

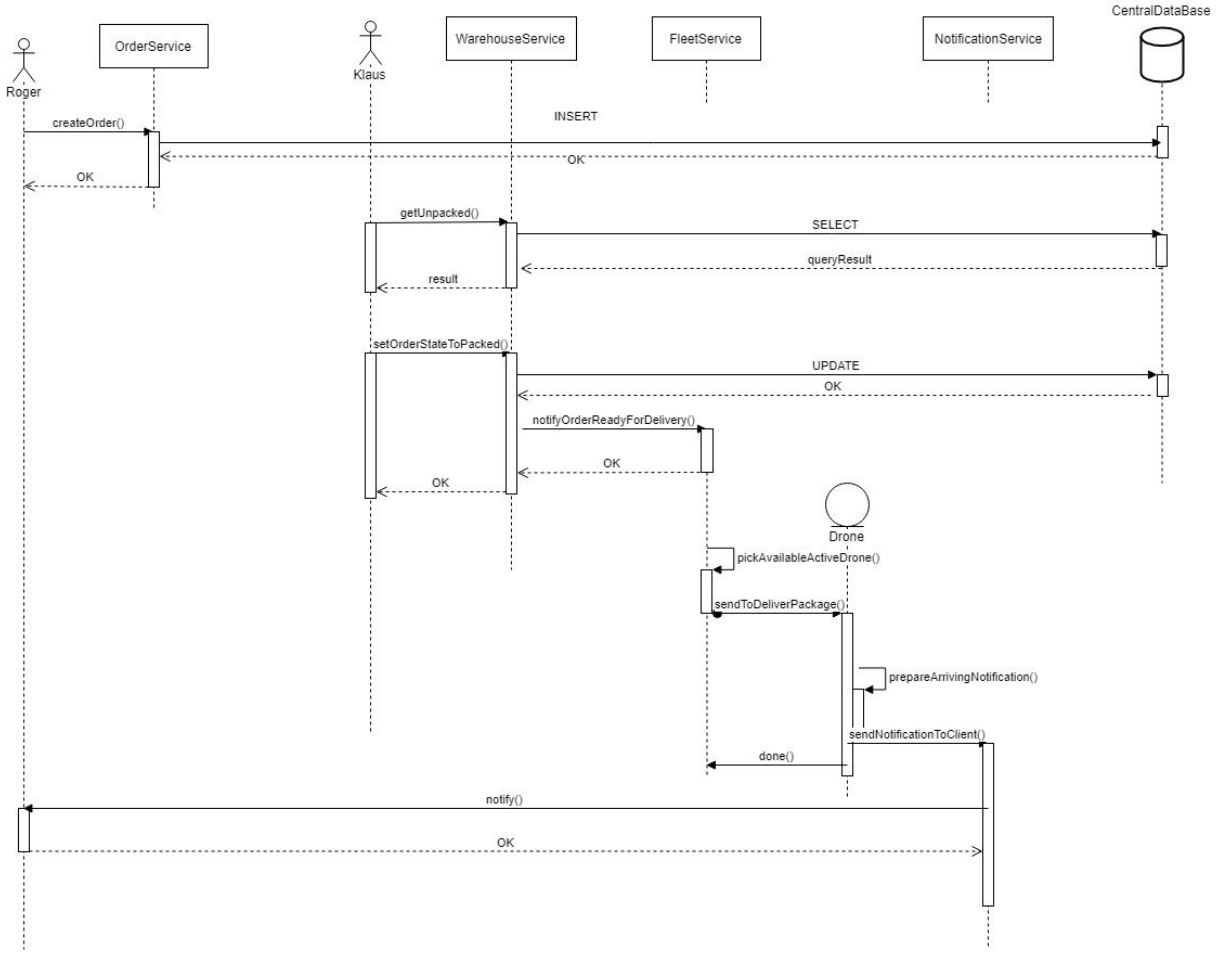
Pour cette deuxième lecture, nous avons privilégié les aspects fonctionnels de chacune des "user story" ; dès lors, pour mieux présenter nos vues, nous avons choisi d'utiliser des diagrammes de séquences.

---

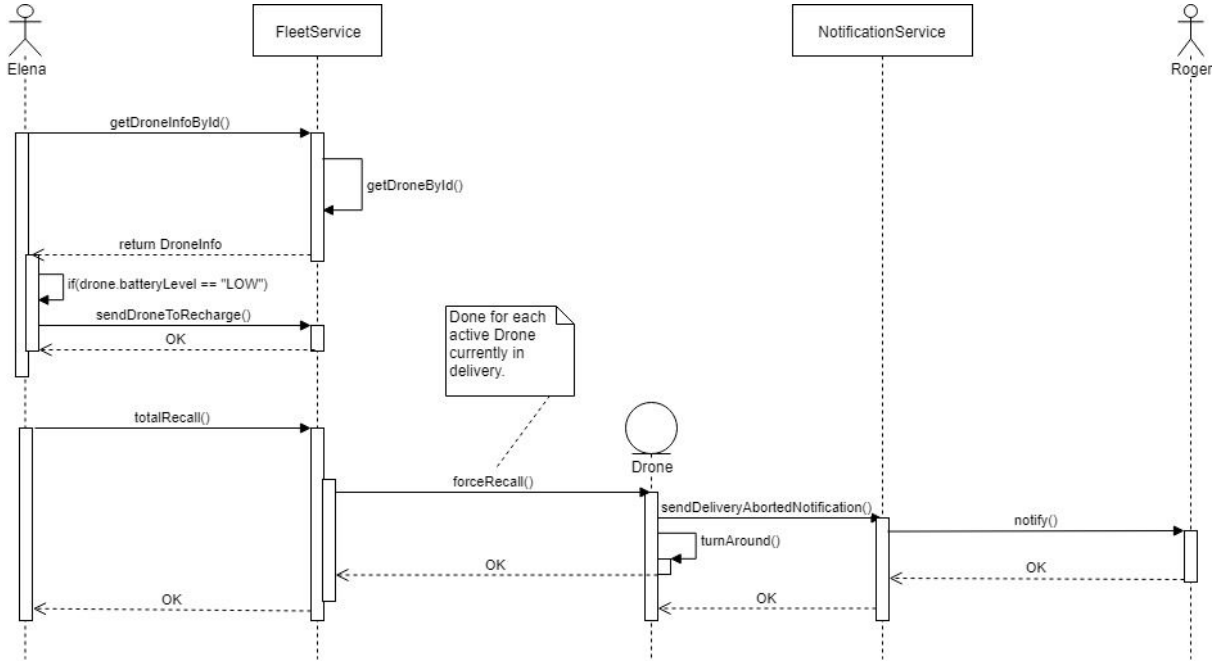
[1] En réalité, ce "mapping" porte sur l'ensemble des classes du modèle.

[2] À ce jour, aucune base de données n'ayant été créée, nous n'avons, en vérité, que la partie "mapping" de la couche d'accès aux données.

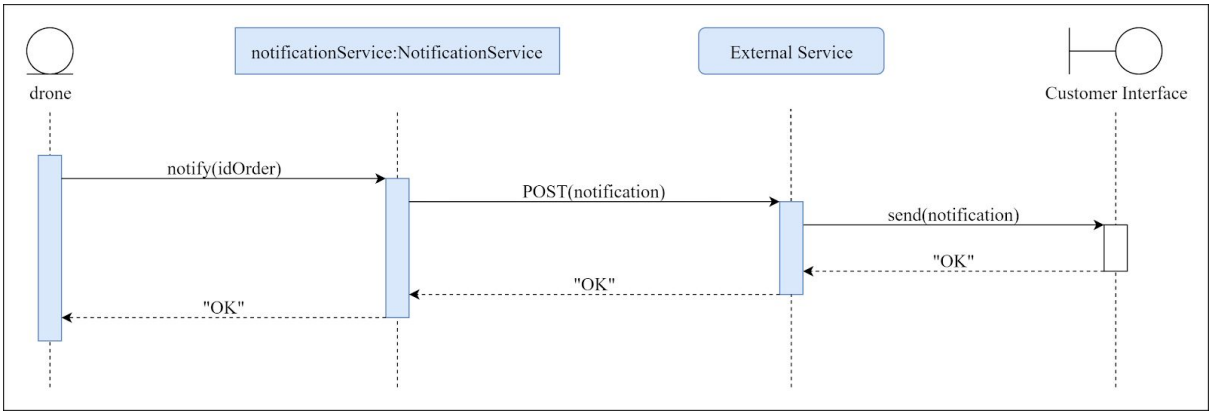
Il nous est apparu tout de suite que les “user story” 1, 2 et 3 peuvent fournir un scénario :



De même, les “user story” 4, 5 et 6 le scénario qui suit :



Plus précisément, voici ce qui se passe dans le cas de la “user story” 3 (ou le cas de la “user story” 6), par exemple :



Admettons que les “user story” 1 et 2, notamment, se soient correctement passée : Roger a pu commander l’article de son choix et Klaus a pu prendre connaissance de la commande de Roger ; par ailleurs, un drone est en chemin pour effectuer la livraison. Puis un événement déclenche la consommation du service de notification : le drone applique la méthode “notify(idOrder)”, où “idOrder”, étant l’identifiant de la commande, suffit à générer un message de notification (avec l’adresse postale de Roger, par exemple). Cette application est accompagnée d’une consommation du service de notification (interne) : une requête HTTP utilisant la méthode POST est envoyée au service externe, qui, à son tour, envoie la notification à Roger…

3 Conclusions

L’idéal aurait été, sans doute, d’avoir une image Docker de chaque service. Chaque service aurait été alors vu comme une fonction à part et que l’on peut déployer indépendamment du reste (un peu comme avec Google Cloud Functions) ; un avantage de cela et que, s’il fallait ne déployer qu’une partie des services, déployer un nouveau service, supprimer un service ou modifier un service, nous n’aurions pas eu à agir sur la totalité de notre produit.

L’idéal aussi aurait été d’utiliser Cucumber pour des tests d’intégration : une fois que tous les services ont été démarrés, les tests Cucumber sont lancés ; chaque test reproduit les comportements d’un *persona* dans les cas d’utilisation d’un service. Le langage, Gherkin, très proche d’un langage naturel, permet précisément de coller à l’énoncé d’une “user story”. Cela aurait dû, par ailleurs, nous inciter à faire du BDD ; mais, pour faire du BDD, il aurait fallu mieux travailler notre conception.

Le manque d’accord entre les membres, enfin, apparaît, au moins partiellement, dans notre “implémentation” ; celle-ci est en effet très hétérogène par endroits : par exemple, actuellement, seul le service de notification a été “dockerisé”. L’écriture des scripts shell d’installation et d’exécution nous a permis de repenser certains points, afin de parvenir à une cohérence suffisante de l’ensemble, pour que notre produit fonctionne.

Le résultat obtenu est un produit encore éloigné de l’architecture que nous avons conçue et qui présente un certain nombre de défauts importants : l’accès aux données, par exemple,

s’inscrit dans un couplage fort, puisqu’il est commun à tous les services. L’idéal aurait été que chaque service ait son propre accès aux données : jusqu’à présent, nous n’étions pas d’accord entre nous sur ce point ; toutefois, nous pourrions profiter, prochainement, de créer la base de données de notre “solution” pour essayer de mettre en place un tel accès.