Drone Delivery

RAPPORT DE PROJET

Introduction to **S**oftware **A**rchitecture

>_Team D

DJEKINOU Paul-Marie

KOFFI Paul

MESSAN Aurore

MONIN Donélia

NABAGOU Djotiham

Dimanche, 10 Mai 2020

Sommaire

Introduction

Drone Delivery est un système de livraison de colis par drones, faisant interagir trois parties principales à savoir le client, le serveur backend et le serveur externe. Telle est l'architecture dans sa globalité que nous devrions modéliser dans la première partie de ce projet. Le rendu ci-contre fait mention de la mise en œuvre de l'architecture proposée grâce aux technologies apprises en cours, des éventuelles évolutions ou améliorations basées sur nos justifications de conception, la présentation technique du code produit et la cohérence des interactions entre les trois parties développées, chacune étant sur sa propre couche technologique différente des autres.

I. Evolution d'Architecture

Nous allons présenter dans cette partie une architecture plus complète et détaillée que celle présentée lors du premier rendu tout en expliquant les motivations des modifications apportées.

1. Diagramme de composants

Comparé au rendu précédent notre diagramme de composant n'a pas reçu beaucoup de changement. Avant d'énoncer et d'expliquer nos changements de conceptions. Voici un bref récapitulatif de ce qu'était notre diagramme de composant du départ.

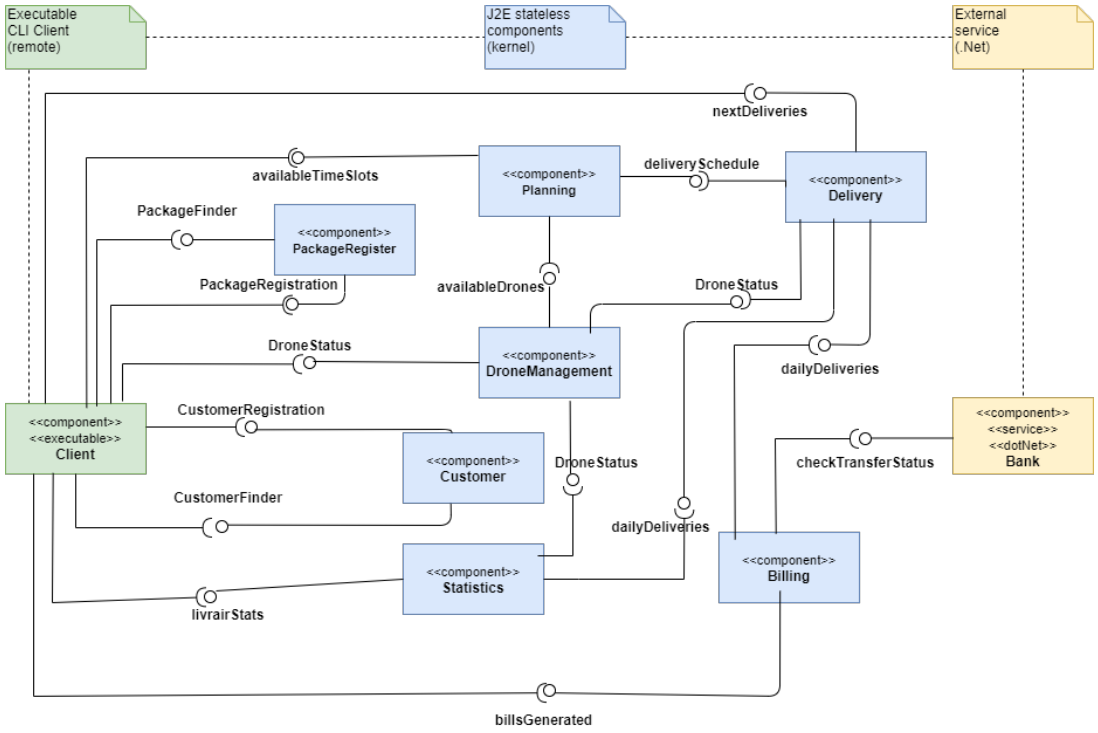

Figure 1 : Ancien diagramme de composants

Par rapport à l'ancien diagramme, nous avons ajouté les web services entre les clients et les composants de notre système j2e. La raison pour laquelle nous l'avons fait est qu'avec ce diagramme notre architecture était orienté base de données et de plus nous cachions une certaine logique dans certains composants ce qui faisait que notre système ne se ramenait qu'à demander des valeurs sans rien faire. Nous l'avons donc changé pour passer plus sur une architecture qui offrait des fonctionnalités, de calcul, de logique et non plus juste des récupérations de données. Avec l'ajout des web services, nous conceptualisions mieux cette nouvelle architecture.

Ainsi ce qui changeait était le fait que les interfaces fournies par les composants j2e vers les clients étaient consommées par les web services qui eux offraient d'autres interfaces aux clients.

Nous avons gardé tous nos composants comme sur la figure précédente, sauf que nous avons ajouté un composant : ***ProviderComponent***. (Voir figure 2). Nous en avions besoin pour pouvoir donner le contrôle au client d'ajouter, modifier ou encore de supprimer un fournisseur. Cela nous est paru évident de pouvoir stocker les informations de nos fournisseurs dans notre système afin d'avoir une meilleure gestion, facilité d'utilisation du système pour le client surtout lors de l'enregistrement d'un colis et la planification d'une nouvelle livraison, mais aussi pour la génération des factures.

ProviderComponent fournit deux interfaces : *ProviderRegister* et *ProviderFinder* à l'instar de nos deux autres composants d'enregistrement (de colis et de client). En effet, la responsabilité pour ces composants est similaire à la différence des données qu'elle manipule qui peut nécessiter un comportement différent selon la donnée à enregistrer.

Pour ce qui est du reste, le diagramme a reçu quelques modifications. En effet, le composant « DroneManagement » offre son interface « AvailableDrone » non plus au composant « PlanningComponent » mais plutôt à « Delivery ». La raison est qu'au départ nous pensions que c'était lors de la planification d'une livraison qu'il fallait à la fois regarder si le créneau choisi par le client était disponible mais aussi s'il y'avait des drones disponibles. Mais elle a été rectifiée car nous avons remarqué qu'entre la date où la livraison a été planifiée et le jour de la livraison il se peut que le drone chargé de cette dernière ne soit plus disponible au moment (soit en charge soit en réparation). Nous avons choisi alors de ne pas affecté de drone à une livraison lors de la planification mais au moment de la livraison. Ainsi le système donnera toujours un drone libre au moment d'une livraison (moyennant le nombre de drones que possède l'entreprise). On n'a donc plus à se soucier d'un potentiel changement d'état pour un drone au moment de la livraison.

De plus, il nous est paru aussi nécessaire de nous occuper de l'enregistrement d'un drone. Dans notre première architecture, nous ne considérons pas cet aspect. Ainsi, nous choisissons maintenant de donner la possibilité au client d'enregistrer/supprimer un drone. Ayant déjà un composant dont le rôle est la gestion du cycle de vie d'un drone (en cours de livraison, déchargé, en réparation…), nous avons décidé de permettre à ce composant de fournir une interface au webService (*DroneRegister*) pour l'enregistrement d'un drone.

Par ailleurs, nous n'avons eu le temps d'implémenter le composant « statistics » défini dans l'ancien diagramme. Nous avons choisi de mettre l'accent sur les composants métiers de notre architecture.

Enfin, nous avons effectué quelques modifications dans les interfaces en ajoutant de nouvelles méthodes :

- Interface Deliverie

Nous avons rajouté une méthode getAllDeliveriesOfAProvider(int id): List<Delivery> qui permet de récupérer toutes les livraisons effectuées pour un fournisseur donné.

- Interface DeliverySchedule

Nous avons rajouté la méthode *findDeliveryByDateAndHour(string deliverydate, string deliveryhour): Delivery*. Elle nous permet d'obtenir une livraison à une heure donnée. Nous avons décidé de filtrer sur l'heure car avec l'Id, étant donnée deux fournisseurs différents, l'on peut avoir le même Id pour la livraison. Selon notre planning, il ne peut avoir qu'une seule livraison à une heure donnée, donc nous avons choisi de faire la recherche par rapport à l'heure de la livraison.
En plus de cette méthode, nous avions besoin d'une méthode pour obtenir les livraisons effectuées un jour donné :
all_deliveries_oftheDate(string date): List<Delivery>

- Interface DeliveryRegistration

Nous avons la méthode : reprogragmminDelivery () :
Cette nouvelle méthode nous permet lorsque le client a un empêchement quelconque de trouver une autre plage horaire disponible pour reprogrammer cette livraison.

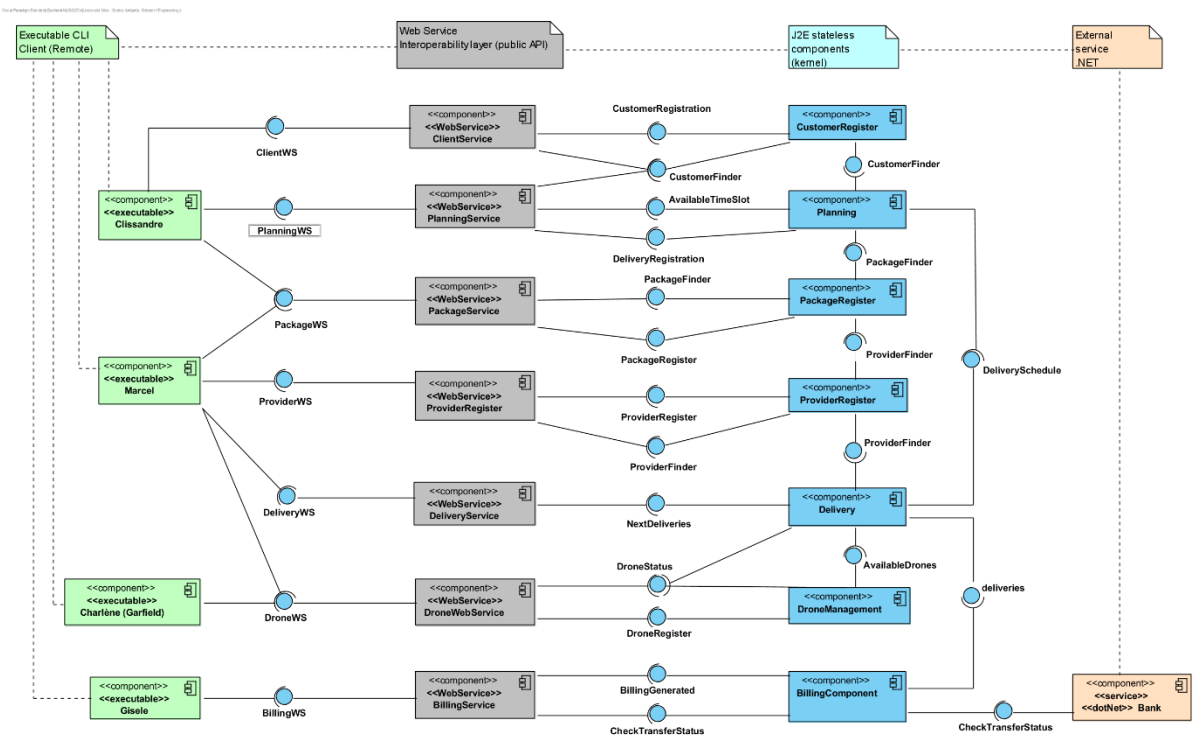

Figure 2 : Nouveau diagramme de composants

2. Diagramme de classes

Pour ce qui est du diagramme de classes, nous n'avons pas apporté de nombreuses modifications. Nous avons juste ajouté une classe « My_Date » qui permettait de manipuler les dates et les heures dans le système.

II. Mise en œuvre et choix de conception

1. Evolution des choix de conception

Notre architecture niveau emplacement a évolué dans le sens où nous avons mis sur des dépôts différents nos différents composants. En effet cela apportait de la souplesse, de la modularité et d'agilité dans notre système.

Chaque composant dans notre projet se trouve sur son dépôt propre à lui. Cette approche nous permet de faire le plus faible couplage possible entre tous les composants. Les composants cœurs du système : Planning et Delivery. Ces 2 composants interagissent entre eux pour implémenter toute la logique qui entoure une livraison de colis. On a pensé à les dissocier car toute la logique pour programmer une livraison, reprogrammer une livraison , valider les plages horaires disponibles se référait plus à du planning et non pas directement à une livraison. Ce faisant , On a donc décider de les implémenter dans Planning.

Pour consulter la prochaine livraison, avoir la liste des livraisons du jour ou des livraisons d'un jour, avoir la liste des livraisons par fournisseur etc … , on a juger utile d'implémenter cette logique là alors dans le composant Delivery.

Nous avons débuter le projet avec une architecture Web où le WebService d'un composant se trouvait directement dans ce composant. Par la suite, il nous est donc apparu évident que nous devions réduire le couplage fort d'une part et de l'autre il nous fallait lancer un seul serveur tomee run qui fournirait les routes disponibles vers tous les WebServices de notre Système. Nous

avons donc créer un module Web qui se chargerait de déployer tous les autres WebServices. Pour ce faire , on a donc migrer les WebServices qui sont devenus individuellement des projets à part entière, ce qui nous permettait d'améliorer la modularité et de l'autre cela nous permettait donc à partir du seul module Web qui serait déployé d'avoir accès aux autres WebServices disponibles. Ce fut l'architecture idéale pour nous.

2. Persistance

Nous avons implémenté la persistance dans tout le système j2e en utilisant Java Persistence API (JPA, OpenJPA) et HyperSQL Databse (HSQLDB). Tout d'abord, nous avons faire correspondre les attributs des différentes classes du diagramme de classes en des attributs d'une base de données c'est-à-dire que les différentes colonnes d'une table sont en type et en nom identiques aux attributs de la classe car la table correspond à la classe qui possède l'annotation « @Entity ». Donc la base de données contient autant de table qu'il y a de classe avec cette annotation. Dans ces classes, nous avions dû ajouter un attribut id qui correspondait à la clef primaire de la classe et qui était généré par la base de données. Au vu de cela les classes mappées ont un autre id qui est l'id logique. Et pour terminer avec les classes pour la persistance, il fallait établir les différentes relations / associations entre les différentes classes. Nous avons donc utilisé les annotations @ManyToMany ou @OneToMany, OneToOne. Dans le cas où les relations sont autres que OneToOne, il faut préciser un référent qui donne le sens de cette relation et cela en ajoutant mappebBy= 'nomdelaclasse' ». L'application de ces annotations représentent les relations dans notre diagramme de classes. Nous avons aussi annoté certains attributs dans les classes @NotNull pour contraindre le fait que cet attribut ne doit point être nul lors de l'ajout d'une instance de la classe dans la base de données. Evidemment nous n'avons pas mis toutes les classes persistantes, puisque la classe « Drone Status » a été mise en « @Embeddable » c'est pour indiquer que cette classe est persistante mais pas avec une clef primaire ; elle peut être embarquée dans une classe « @Entity ». Nous l'avons utilisé dans le cas la relation entre Drone et Drone_Status, car nous n'avions aucune raison de d'enregistrer une clef pour cette classe car l'on en a aucune utilité et nous n'avons pas besoin d'assurer l'unicité des instances de cette classe. Pour terminer pour que tout cela fonctionne, nous avons ajouté un fichier « persistence.xml » dans lequel nous avons renseigné les classes qui étaient persistantes. Enfin pour utiliser la persistance dans les autres composants, il faut l'« entityManager » grâce à l'annotation @PersistenceContext qui est celui se trouvant de le fichier « persitence.xml ». Puisque c'est l'entityManager qui se charge d'effectuer des actions avec la base de données ici qui est HSQLDB.

3. Type de composants implémentés : **STATELESS**

Dans notre projet, nous avons fait le choix d'implémenter tous nos composants en type « stateless ». Ce choix se justifie par tout d'abord qu'en terme de performance une architecture avec le plus de composants stateless est bonne dans le sens où l'on ne conserve pas l'état pour l'action d'après. C'est-à-dire que les actions sont complètement indépendantes entre elles. En effet lorsque l'on enregistre un objet nous n'avons pas besoin de connaître ce qui s'est passé lors de la transaction précédente. Et cela est valable pour la recherche d'une instance. Il en va de même pour la recherche, la récupération d'une liste de valeur ou encore d'une mise à jour. Nos actions dans le système sont indépendantes entre elles donc n'ont pas besoin d'une information de la précédente.

4. Intercepteurs

Nous avons aussi implémenté des intercepteurs dans notre projet. Les intercepteurs permettent comme son nom l'indique d'intercepter ce que l'on désire pour appliquer une logique qui est externe à notre système. Avec comme exemple notre projet, nous avons implémenter deux intercepteurs dans le planning composant. Ces deux intercepteurs qui vérifient que la date et l'heure passés aux fonctions « registerDelivery » et « reprogragmminDelivery » et « validSlot » se trouvant dans planning web services. Ainsi lorsque l'utilisateur fait appel à une de ces fonctions les paramètres de ces fonctions sont d'abord vérifier dans les différents intercepteurs et s'ils sont validés alors il va reprendre son exécution en rentrant dans la fonction et donc dans le système mais s'ils ne sont pas validés par le système alors une exception est levée et c'est celle de la fonction de l'intercepteur. Cela permet au système de se prévenir des erreurs potentielles.

III. Bilan

L'architecture de notre projet a beaucoup évolué notamment avec les changements opérés au niveau du diagramme des composants ainsi que nos choix de conception.

Dans notre système actuel, nous pouvons effectuer une livraison, enregistrer un client, un fournisseur, des colis. Nous pouvons aussi vérifier l'état des factures ainsi que générer des factures. Nous pouvons programmer et reprogrammer des livraisons, ajouter des drones et modifier leur état (available, being repared, etc).

Quant aux points forts de notre solution, nous pouvons citer la responsabilité de nos composants. Nous avons découpé le projet de telle sorte que chaque composant ait un rôle précis. Cela permet à notre projet d'être ouvert à la modification et l'ajout de nouvelles fonctionnalités. Ajoutons à cela le respect de l'architecture trois-tiers. Nous avons essayé de respecter cette notion avec nos 3 couches bien définies (celle qui accède aux données, celle qui fait le traitement et celle qui les affiche).

Cependant, nous pouvons citer quelques points faibles dans notre solution et suggestions d'amélioration. Dans un premier temps, nous avons la mise à jour des données en base de données. Nous avons des difficultés suivant le contexte en utilisant la méthode *find()* de entityManager ou encore le *merge()*. Nous avons rencontré souvent des erreurs selon le contexte, et selon la méthode. Nous devons donc revoir la documentation de ces deux fonctions, et comprendre la différence. De plus, nous n'avons pas fait de vérification avant de re programmer une livraison. En effet, nous devons vérifier d'abord que la livraison n'a pas déjà été effectuée avant de pouvoir la reprogrammer à une date ultérieure.

Enfin, nous pouvons rajouter quelques fonctionnalités telles que le suivi du colis, et le changement de la perte du niveau de batterie selon la distance effectuée par le drone, car actuellement la valeur est statique. Nous pouvons aussi rajouter la gestion des statistiques pour Bob, et le fait de voir l'impact des composants en oriented-messages sur notre architecture.

Conclusion

Le projet Drone Delivery, nous appris les concepts d'architecture logicielles et de DevOps et à les mettre en application. La partie Introduction en Architecture Logicielle nous a appris comment modéliser la conception d'un logiciel à différents niveaux. Que ça soit l'architecture global avec les différents acteurs, la relation entre ces derniers par les contrats (les interfaces), le principe de 3-tiers architecture. Elle nous a aussi les choix technologiques et techniques liées à l'implémentation du logiciel avec les EJB, de dépendance injection, l'implémentation des 3 couches de 3-tiers architecture., l'utilisation de la persistance, de l'interopérabilité, l'implémentation de la sérialisation de nos données pour les transports à travers des clients distants et autres.

L'implémentation de Drone Delivery basée sur l'architecture initialement prévue nous a conduit à plusieurs étapes de réanalyse et de prise de recul. Nous obtenons finalement le système actuel dont les caractéristiques et motivations apparaissent dans ce rendu. Les notions d'ISA apprises en cours étaient nos outils pour l'élaboration d'une bonne architecture.