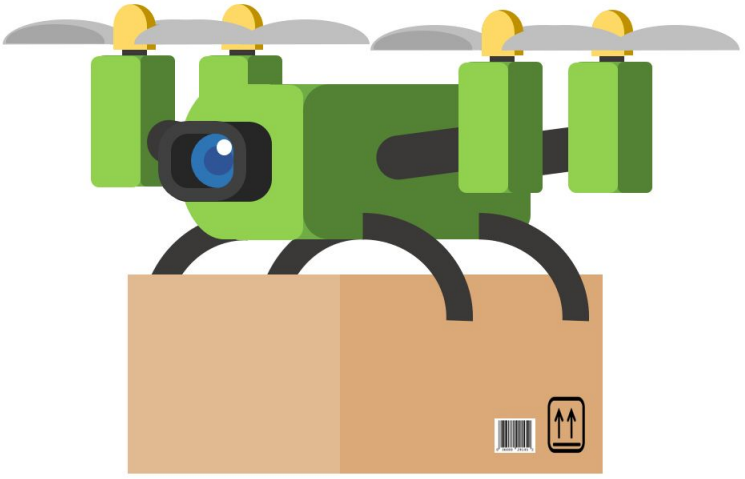


Livr'air

Introduction to Software Architecture

Team H

Focas Florian

Montoya Damien

Quang Minh Doan

Rigaut François

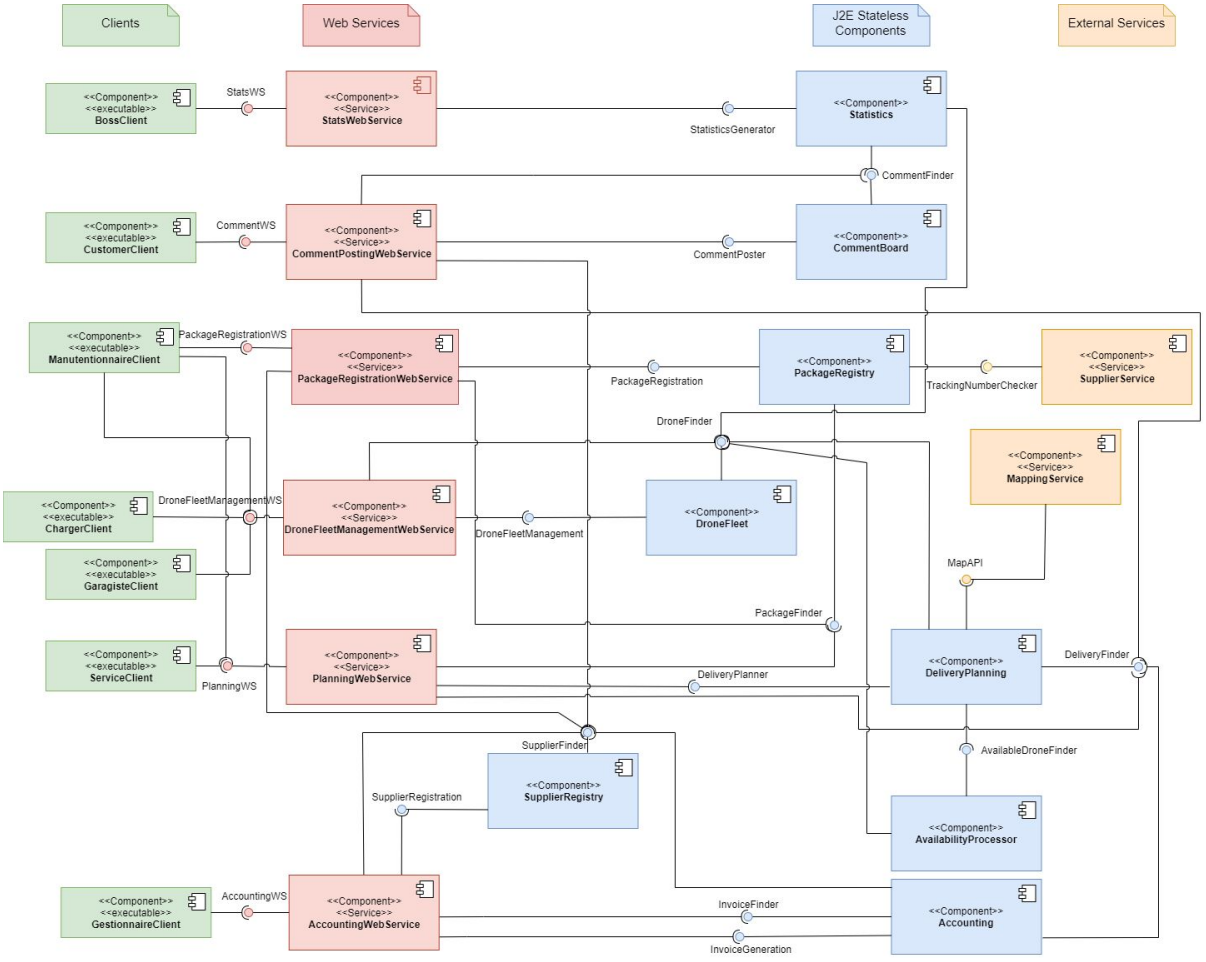
Todesco Gabin

Sommaire

Architecture

Diagramme de composants

Suite aux retours plutôt négatifs sur notre architecture de composants lors du premier rapport, nous avons décidé de reprendre celle-ci à zéro. Nous avons bien réfléchi et sommes finalement arrivés à une meilleure architecture, mieux pensée pour un développement par composants. Voici donc le diagramme de composants qui en a découlé :



Tous les composants présents dans ce diagramme ont bien étés implémentés hormis le service externe “TrackingNumberService”. En effet, lors de la phase d’analyse nous avions imaginé un service externe permettant de récupérer auprès des fournisseurs les informations d’un colis grâce à son identifiant comme par exemple l’adresse de livraison, le numéro de téléphone du destinataire etc.... Cependant, nous n’avons pas eu le temps d’implémenter ce service, n’étant pas une priorité puisque nous avons déjà un service externe pour démontrer la chaîne CLI—backend—Service externe, avec l’implémentation du MappingService.

Parmi ces composants, plusieurs servent à gérer l'accès et l'édition à des objets persistés: SupplierRegistry, DroneFleet, PackageRegistry et CommentBoard notamment font partie de ces composants "simples" de gestion d'objets. Ils ont été séparés par souci de séparation des responsabilités.

Ensuite, le composant DeliveryPlanning sert à planifier les livraisons. Pour ce faire, il contacte dans un premier temps le MappingService, service externe qui lui permet de retrouver la distance nécessaire pour la livraison en lui envoyant l'adresse du colis. Ensuite, il contacte le composant AvailabilityProcessor, qui se charge de trouver le drone le plus optimisé pour effectuer la livraison au moment voulu si cela est possible, ce qui permet au DeliveryPlanning de créer la livraison.

Le composant Accounting sert à générer et retrouver les factures. Une facture reprend toutes les livraisons effectuées durant le dernier mois, et stocke ces objets en mémoire pour chaque supplier.

Enfin, le composant Statistics s'occupe de la génération des données statistiques correspondant au taux d'utilisation des drones et à la satisfaction client.

Nous avons créé différentes CLI, une pour chaque poste de l'entreprise, afin de séparer les responsabilités et tester l'efficacité de notre modèle de web services. Nous avons également ajouté un client les regroupant tous, appelé client d'intégration, qui ne figure pas sur le diagramme par souci de lisibilité. Ce client nous permet de tester toutes les commandes dans un seul système, et en plus de faire des tests d'intégration.

Forces et faiblesses

Lorsque nous avons conçu le diagramme de classe pour le premier rapport, nous avions une approche trop "base de données". En apprenant de ces erreurs, nous avons ajouté le composant AvailabilityProcessor, composant plutôt axé sur des calculs prévisionnels que de la gestion d'objet. Nous pensons que cette séparation enlève de la responsabilité au DeliveryPlanning qui en porte déjà beaucoup, et permet de séparer la logique de prévision des livraisons du stockage des objets.

Le fait que tous les autres composants s'occupent de la gestion d'un objet en mémoire nous semble être la bonne option dans une optique de traçabilité de l'historique de l'entreprise. La séparation en un composant par type d'objet permet d'avoir une vue plus claire sur le système et des responsabilités mieux réparties. Nous avons hésité à combiner les composants SupplierRegistry et Accounting, le rôle du second étant assez proche de la gestion des fournisseurs. Nous avons cependant décidé de laisser ces composants séparés par souci de lisibilité du système.

En outre, nos web services utilisent des méthodes prenant des objets primitifs en paramètres. L'avantage de cette solution est d'avoir une utilisation plus simple des fonctionnalités du système, car nous sommes sûr qu'un identifiant référence un seul et unique objet persisté alors que passer un objet complexe en paramètre pourrait causer des

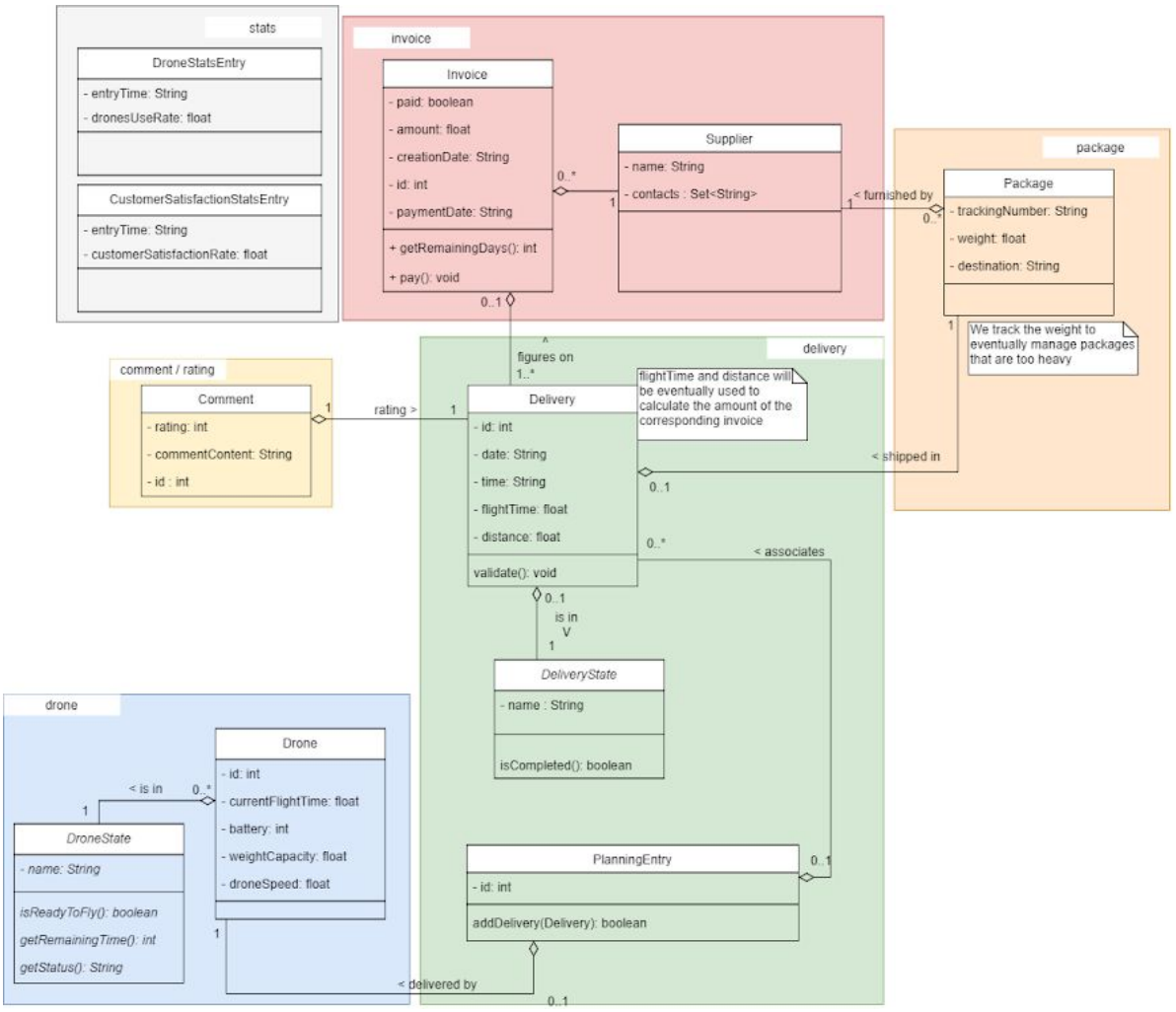
soucis de cohérence et synchronisation des objets. D'autre part, cela rend l'utilisation des commandes de notre CLI beaucoup plus simples, puisque l'on entre l'identifiant de l'objet en ligne de commande, et non l'objet lui-même.

Cependant, cette méthode n'exploite pas les capacités de la communication SOAP à son plein potentiel, mais surtout elle a pour effet de rajouter des dépendances entre composants, puisque les web services doivent parfois retrouver un objet persisté via son identifiant à l'aide d'un composant Finder pour le passer en paramètre à un autre composant.

En effet, nos composants, eux, prennent des objets complexes en paramètres, afin de faciliter l'interaction entre composants et la gestion d'erreur dans ces cas-là, en regardant si un objet avec un certain identifiant est exactement identique à l'objet avec le même identifiant qui est persisté, et gérer les erreurs de cohérence.

Globalement, notre système est assez fragmenté et comporte de nombreux composants et beaucoup de liaisons entre eux, mais nous pensons que cela rend chaque fonctionnalité du système précise, claire, et répartit les responsabilités d'une manière qui nous semble correcte.

Diagramme de classes



Pour commencer, nous avons identifié les entités principales du projet : les drones, effectuant la livraison, les fournisseurs mettant à disposition les colis à livrer, les factures et les commentaires.

À cela s'ajoutent certaines entités moins évidentes telles que les PlanningEntry, qui sont des entrées associant chaque drone à une liste de ses livraisons passées, présentes et futures, les Drone et Customer StatsEntry pour gérer la satisfaction client et l'utilisation globale des drones en stockant leur date de génération pour pouvoir suivre l'évolution de ces indicateurs, et enfin les Drone et Delivery State pour différencier l'état présentiel d'une livraison et d'un drone (en cours de livraison, prêt à la livraison, en maintenance...).

Les livraisons (classe Delivery) sont fortement couplées aux packages. En effet, cela n'a pas de sens d'effectuer une livraison sans colis. De même, impossible de commenter sans Delivery.

Les Drone et Delivery State sont des classes abstraites, qui seront héritées par leurs enfants. Ces dits enfants sont créés par une factory grâce à un mot clé (exemple : “ready”, “completed”...) ce qui permet de facilement ajouter de nouveaux états si nécessaires, sans modifier quoi que ce soit d’autre, et également de récupérer ces mots-clés pour les utiliser en lignes de commandes dans la CLI. Nous avons décidé de partir sur une conception d’héritage plutôt que d’énumération car certains états impliquent des variables différentes (exemple : “not-sent” n’a pas besoin de variable de date, alors que “completed” peut en avoir besoin). Il aurait donc été possible de partir sur une énumération, simplifiant le code au prix de la précision sur les statuts.

Les variables flightTime et distance dans Delivery ont pour but de permettre à l’algorithme de sélection de drone de choisir plus facilement le drone qui va effectuer la livraison, ainsi que de permettre (ultérieurement) une révision du calcul du paiement.

Les PlanningEntry sont des entrées telles qu’on pourrait en trouver dans une base de données qui assignent à chaque drone (Drone) de la flotte la liste des livraisons (Delivery) sous sa responsabilité (passées, en cours, futures). L’attribut id de cet objet n’est autre que l’id du drone responsable. Nous avons en effet conclu que, pour mesurer l’activité d’un drone, il était plus évident de regarder la liste des livraisons effectuées par chaque drone, plutôt que de regarder pour chaque livraison effectuée depuis une date donnée l’id du drone associé.

Les attributs date des différentes classes sont représentés sous forme de String afin de permettre une persistance et sérialisation des données. En effet, les objets LocalDate et LocalDateTime ne sont pas sérialisables, mais il est cependant possible de les convertir en String et de recréer l’objet à partir de ce String grâce au parseur de la classe.

Force et faiblesses

Les PlanningEntry sont un des atouts de notre projet, car en mettant à jour cette liste de livraisons avec pour l’id du drone responsable, on peut retrouver aisément toutes les livraisons avec un minimum de calcul de la part du serveur.
Cependant, cette économie de calcul signifie que, de l’autre côté, nous avons une perte en mémoire proportionnelle au nombre de drones et livraisons. De plus, ce modèle ne permet pas d’avoir une liste ordonnée dans le temps de toutes les livraisons associées à leur drone, ce qui rend l’algorithme de prévision un peu plus complexe. On pourrait pallier ce problème en créant une classe additionnelle Planning comme notre structure le suggérait dans le premier rapport, mais cette classe s’est avérée assez peu utile et nous avons décidé de ne pas l’implémenter, même s’il est possible de le faire.

Interfaces

Accounting

InvoiceFinder

```
Set<Invoice> findAllUnpaidInvoices();
Set<Invoice> findInvoicesForSupplier(Supplier s);
Set<Invoice> findAllInvoices();
Optional<Invoice> findInvoiceWithId(int id);
```

InvoiceGeneration

```
Invoice generateInvoiceFor(Supplier s);
void generateInvoicesForAllSuppliers();
boolean deleteInvoicesForSupplier(Supplier supplier);
boolean setInvoicePaid(int invoiceId) throws UnkownInvoiceException;
```

AvailabilityProcessor

InvoiceFinder

```
Optional<Drone> getAvailableDroneAtTime(Set<PlanningEntry>
alreadyPlannedDeliveries, LocalDateTime timeToDeliverThePackage, float
packageWeight, float packageDistance) throws CorruptedPlanningException;
```

CommentBoard

CommentFinder

```
Optional<Comment> findCommentForPackage(String packageId);
Set<Comment> findAllComments();
Set<Comment> findCommentsForSupplier(Supplier s);
```

CommentPoster

```
Comment postComment(Delivery d, int rating, String comment);
boolean deleteComment(Delivery d) throws UnknownCommentException;
```

DeliveryPlanning

DeliveryFinder

```
Optional<Delivery> findDeliveryById(String id);
Set<PlanningEntry> findAllPlannedDeliveries();
Set<PlanningEntry> findCompletedDeliveriesSince(LocalDateTime time);
Set<PlanningEntry> findCompletedDeliveriesSince(LocalDateTime time, Supplier
s);
Optional<PlanningEntry> findPlanningEntryByTrackingId(String trackingId);
Optional<PlanningEntry> findPlanningEntryByDroneId(int droneId);
Set<PlanningEntry> findAllPlannedDeliveriesBeforeAfterNow();
DeliveryState checkAndUpdateState(String name) throws
UnknownDeliveryStateException;
```

DeliveryPlanner

```
Delivery planDelivery(Package p, String date, String time) throws
DeliveryDistanceException, UnknownDeliveryStateException, NoReadyDroneException,
DeliveryPastTimeException, CorruptedPlanningException;
boolean editDeliveryStatus(Delivery delivery, String state) throws
UnknownDeliveryStateException;
boolean startDelivery(Drone drone, Delivery delivery);
Set<PlanningEntry> getCompleteDeliveryPlanning();
boolean deleteDelivery(String trackingNumber) throws
UnknownDeliveryException;
boolean deletePlanningEntry(String trackingNumber) throws
UnknownPlanningEntryException;
boolean deletePlanningEntry(int droneId) throws
UnknownPlanningEntryException;
```

DroneFleet

DroneFinder

```
Optional<Drone> findDroneById(int id);
Set<Drone> findReadyDrones();
DroneState checkAndUpdateState(String name) throws
UnknownDroneStateException;
Set<Drone> findAllDrones();
```

DroneFleetManagement

Drone addDrone(int id, float weightCapacity, float speed) throws AlreadyExistingDroneException;
boolean editDroneStatus(int id, String newStatus) throws UnknownDroneException, UnknownDroneStateException;
Drone editDrone(int id, int battery, float flightTime) throws UnknownDroneException;
boolean deleteDrone(int id) throws UnknownDroneException;

PackageRegistry

PackageFinder

Optional<Package> findPackageByTrackingNumber(String trackingId);
Set<Package> findPackagesBySupplier(Supplier s);
Set<Package> findAllPackages();

PackageRegistration

Package register(String trackingNumber, Supplier s, float weight, String destinationAddress) throws AlreadyExistingPackageException;
boolean edit(String trackingNumber, Supplier s, float weight, String destinationAddress) throws UnknownPackageException;
boolean delete(String trackingNumber) throws UnknownPackageException;

Statistics

StatisticsGenerator

float getAverageCustomerSatisfaction();
float getAverageDroneUseRate();
Set<DroneStatsEntry> getDroneStatsEntries();
Set<DroneStatsEntry> getDroneStatsEntriesFrom(LocalDateTime dateTime);
Set<CustomerSatisfactionStatsEntry> getCustomerStatsEntries();
Set<CustomerSatisfactionStatsEntry> getCustomerStatsEntriesFrom(LocalDateTime dateTime);
DroneStatsEntry generateNewDroneStatsEntry();
CustomerSatisfactionStatsEntry generateNewCustomerSatisfactionEntry();

SupplierRegistry

SupplierFinder

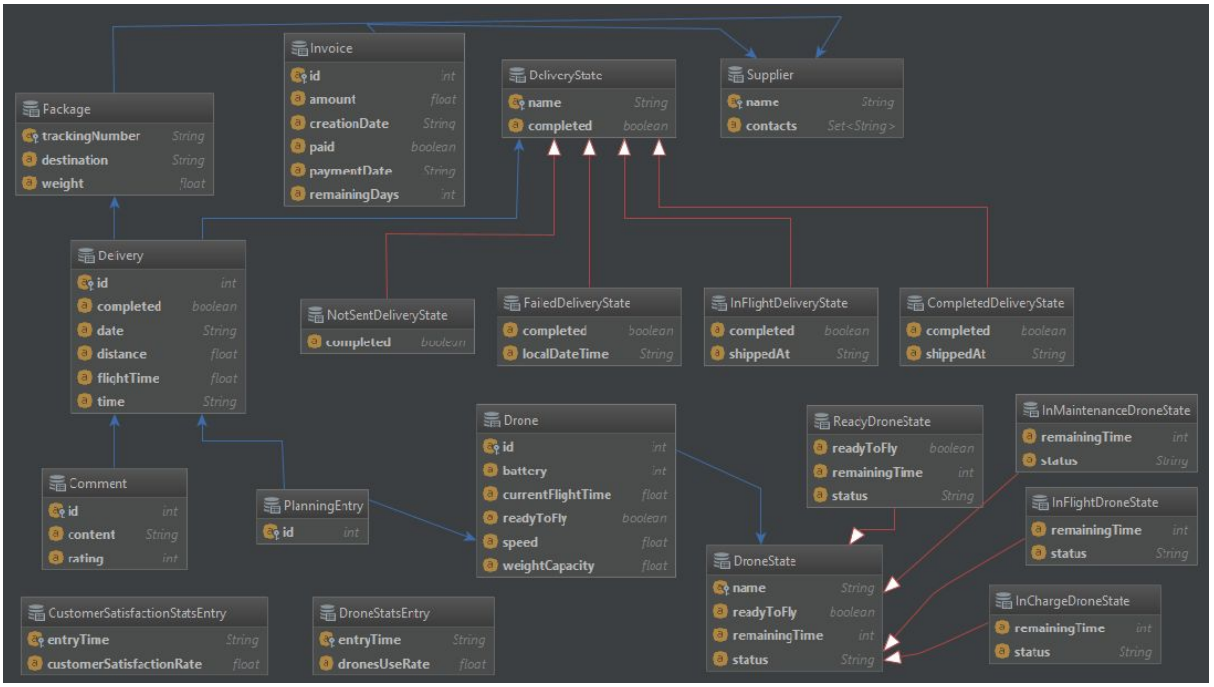
```
Optional<Supplier> findByName(String name);
Set<Supplier> findAll();
```

SupplierRegistration

```
Supplier register(String name, String contact) throws
AlreadyExistingSupplierException;
boolean delete(String name) throws UnknownSupplierException;
boolean addContact(String name, String contact) throws
UnknownSupplierException, AlreadyExistingContactException;
```

Persistance

Diagramme ER



Forces et faiblesses

Dans notre code, deux entités possèdent des attributs de type Collection, PlanningEntry et Supplier. Dans le cas de Supplier, la liste de contacts est un Set de String, donc nous n'avons pas eu trop de problèmes pour l'implémenter, si ce n'est que nous avons mis le comportement FetchType.EAGER (au lieu de FetchType.LAZY par défaut) afin que les contacts soient toujours récupérés lorsque l'on va chercher les Supplier dans la base de données. Nous avons fait de même dans la classe PlanningEntry pour le Set de Delivery. Qui plus est, nous avons dû rajouter un attribut ID dans la classe afin de pouvoir l'identifier de manière unique. Cet attribut est initialisé à la création de l'objet, et est égal à l'ID du drone associé, ce qui peut causer de la redondance, mais nous avons employé cette solution pour sa simplicité d'implémentation.

Toutes nos références à d'autres objets du système dans chaque classe est en CascadeType.MERGE uniquement. En effet, notre manière d'aborder la gestion des objets persistés se focalise sur l'indépendance du comportement de chaque objet. CascadeType.PERSIST nous posait des problèmes car l'objet en attribut était toujours déjà persisté, CascadeType.REMOVE posait également des problèmes lorsque l'on supprimait des objets. Nous ne nous sommes pas servis de REFRESH et DETACH dans ce projet.

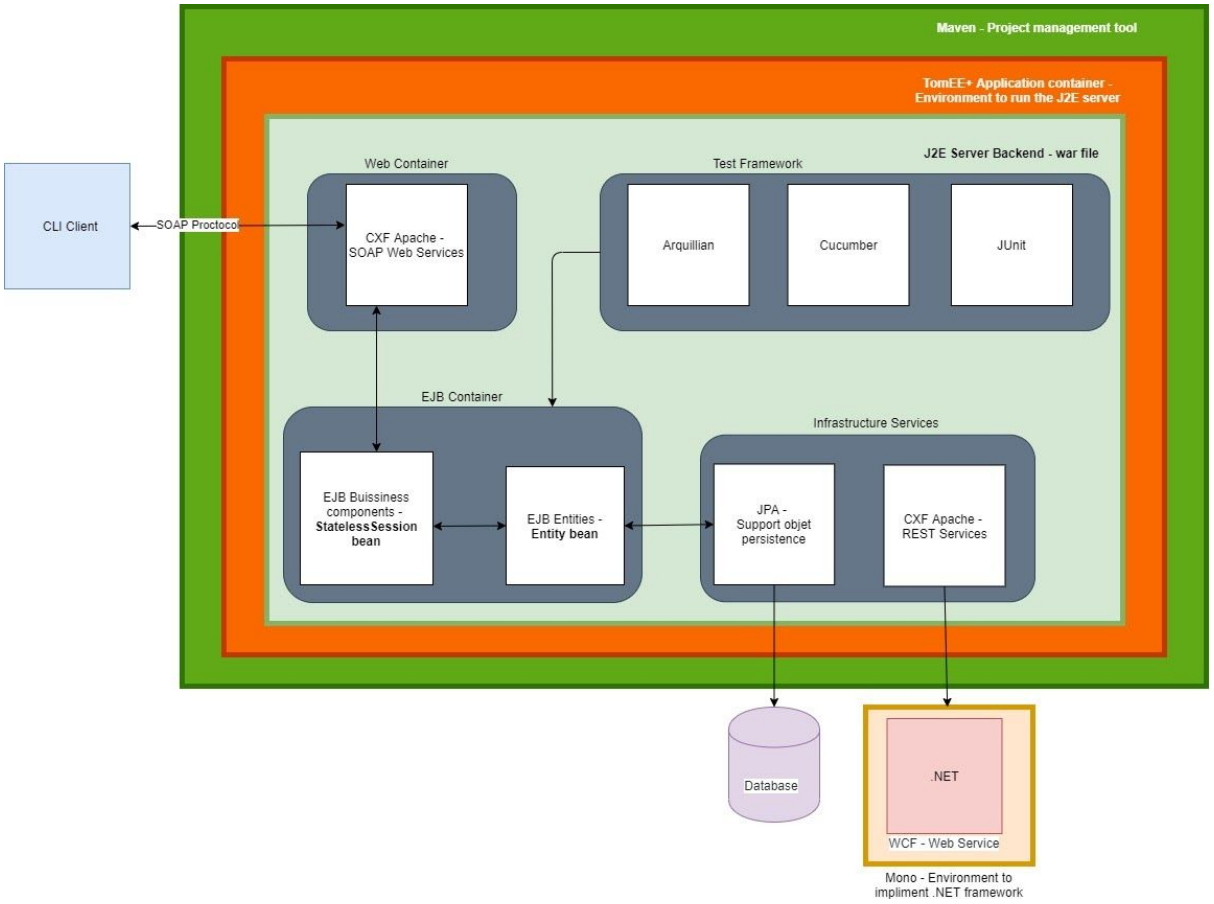
Améliorations possibles du projet

Notre projet pourrait être amélioré de plusieurs manières:

- Ajouter des intercepteurs pour incrémenter des compteurs ou déclencher des méthodes lors du lancement d'une livraison ou de la création d'un commentaire afin de générer des statistiques de manière plus dynamique.
- Implémenter un message-driven bean pour pouvoir ajouter des factures dans une file d'attente pour impression.
- Améliorer l'algorithme de choix de drone, pour l'instant incomplet.
- Créer des factures en fonction des distances ou des temps de vol des livraisons.

Nous n'avons pas trouvé de possibilité d'ajout pertinent de stateful bean. En effet, les différents utilisateurs ont tous pour but d'apporter des modifications à un système commun, et garder des données spécifiques à une session ne nous semble pas intéressant.

Workflow


Maven - Project management tool
TomEE+ Application container - Environment to run the J2E server
J2E Server Backend - war file
CLI Client
SOAP Proctocol
Web Container
CXF Apache - SOAP Web Services
Test Framework
Arquillian
Cucumber
JUnit
EJB Container
EJB Buissiness components - StatelessSession bean
EJB Entities - Entity bean
Infrastructure Services
JPA - Support objet persistence
CXF Apache - REST Services
Database
.NET
WCF - Web Service
Mono - Environment to impliment .NET framework