État d'architecture

I) Architecture courante

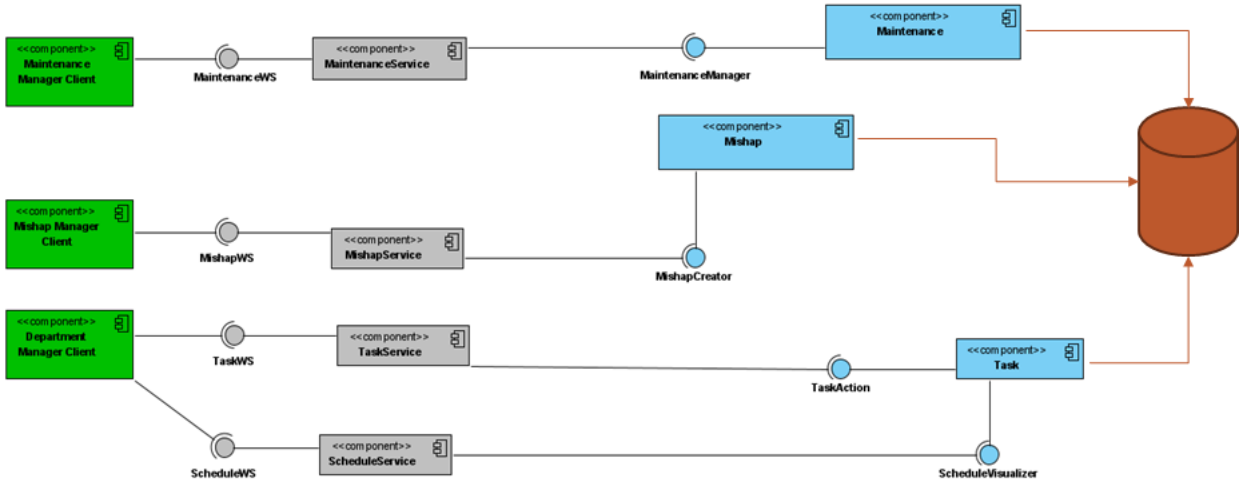

Figure 1 : Schéma d'architecture courante

Note : Les composants gris sont des WebServices.

1) Résumé rapide

Notre architecture actuelle (fig. 1) présente trois points d'entrée principaux. Le premier est destiné à l'interface *MaintenanceManager* qui va pouvoir gérer des maintenances grâce au composant *Maintenance*. Le second est destiné à l'interface *MishapManager* qui va gérer des maintenances grâce au composant *Mishap*.

Enfin le 3e point d'entrée est destiné aux chefs des département qui vont pouvoir visionner leur planning de tâches et confirmer la réalisation d'une tâche, c'est le composant *Task* qui s'occupe de cette logique. Les maintenances et incidents sont directement stockés en base de données et sont tout le temps récupérés de la base, les composants ne gardent pas d'états.

2) Rappel des méthodes des interfaces

MaintenanceManager

```
CreateMaintenance(String name, String type);
GetMaintenance(): Maintenance;
GetMaintenanceById(int maintenanceId): Maintenance;
UpdateMaintenance(int MaintenanceId, String name, String type): Maintenance;
DeleteMaintenance(int MaintenanceId): void;
```

MishapManager

```
createMishap(String name, String type, TaskPriority priority) : Mishap
getMishaps() Mishap[]
getMishapById(Long id) : Mishap
updateMishap(Long id, String name, String type, TaskPriority priority)
deleteMishap(Long id);
```

TaskAction

endTask(Long id) : Task

ScheduleVisualizer

getPlanning() : Task[]

II) Forces et faiblesses

1) Contextualisation à travers les besoins

Afin d'identifier les forces et faiblesses du projet, nous avons tout d'abord identifié les besoins propres au métier :

- **Manipuler des données qui doivent être cohérentes.** Le planning doit toujours être cohérent et exact. C'est-à-dire que les données du planning doivent rester les mêmes sur tous les nœuds du système pour assurer que les maintenances soient coordonnées entre elles.
- **Le système d'alerte concernant les incidents ne doit pas tomber en panne.** Nous devons nous assurer de la disponibilité du système de notification. Des incidents peuvent en effet survenir n'importe quand et le système dit être capable de d'informer les bons départements dans un court laps de temps. Il est aussi important de garder une trace de l'incident dont le département a été notifié.
- **Nous souhaitons des alertes rapides.** L'alerte ne doit donc pas être reçue avec trop de latence et il est important d'être certain que l'information a bien été reçue par le département concerné.
- **Avec le système d'incidents, la planification est au cœur du système**. Le système de planification permet non seulement de placer des maintenances prévues sur un planning mais doit aussi permettre de placer le traitement des incidents par les départements concernés sur ce même planning. Ce système va donc détenir beaucoup de logique puisqu'il doit être capable de placer des incidents non prévus sur le planning des en tenant en compte de leur priorité.

==> La logique globale du système est donc assez concentrée dans la **planification** et le **déclanchement d'alertes non prévues.**

2) Forces

Cohérence et disponibilité des données. Nous traitons des incidents et des plannings, nous avons donc besoin d'une cohérence des données et d'une disponibilité de celle-ci, notre base de données respecte les propriétés ACID qui permettent de répondre à nos besoins. La technologie MySQL mise en place nous permet de correspondre à ces propriétés et donc de répondre correctement au besoin.

Rapidité de la circulation d'information dans le système. Notre système doit gérer des incidents, ce qui implique un besoin en termes de rapidité de traitement par le système. Notre architecture monolithique permet un traitement complet des incidents en limitant beaucoup les passages sur le réseau (contrairement à une architecture en services).

Architecture monolithique adaptée au besoin. L'architecture n-tiers mise en place a été pensée en fonction du besoin

Les besoins métiers que nous avons défini ne nécessitent pas l'implémentation d'une architecture en services car cela aurait trop peu d'intérêt.

Les principaux avantages qu'apporterait une architecture en services pour notre besoin sont :

- La mise à échelle facile du système et la résilience en cas de forte affluence (ce qui pourrait être utile dans le cas où beaucoup d'alertes sont levées avec les incidents) ;
- La facilité de déploiement des services indépendamment des autres (cela serait utile pour déployer le service de planning indépendamment du service qui gère les incidents)

Cependant, un tel type d'architecture implique la complexification de l'implémentation puisqu'on manipule de nombreux services. On repose également beaucoup sur le réseau, or notre système d'alerte doit à tout prix éviter cela.

Concernant la mise à l'échelle du système, cela ne constitue pas un point essentiel car il faudrait un très grand nombre d'alertes d'un coup pour que nous ayons à mettre à l'échelle ce qui n'est pas censé arriver.

Concernant le déploiement de services indépendants, cela serait utile, mais la complexification de l'implémentation n'en vaut pas le coût.

Ainsi, l'architecture monolithique que nous avons mis en place nous permet de faciliter grandement l'implémentation et la maintenance du système tout en répondant aux besoins énoncés plus haut.

Enfin, le Framework a déjà été mis en place et nous comptons poursuivre avec notre choix d'une architecture monolithique. Cela nous permet de nous concentrer uniquement sur l'implémentation de la logique métier et plus sur la configuration.

3) Faiblesses

Manque d'interaction entre les composants. Il y a un manque de communication entre nos composants au sein de notre architecture. En effet comme on peut le voir sur le schéma actuel, les composants ne communiquent pas entre eux, leur comportement est isolé. Cela cause un manque d'implication de notre système et donc une délégation de la logique côté client. Il faudra par la suite donner plus de responsabilité au système car la logique est trop déportée vers le client.

Pas de comportement interne au backend. Le *proof of concept* que nous avons présenté ne contient pas de logique de planification. Les composants ne font que stocker et récupérer des données sans traitement. Cette logique va être ajoutée par la suite à travers l'assignation de tâche au bon département (cf. partie III).

Une gestion ralentie des incidents. Notre architecture actuelle n'est pas assez réactive au niveau des incidents. En effet lorsqu'un incident est créé il passe en base de données avant d'être récupéré depuis celle-ci, cela cause un ralentissement dans l'alerte d'incident.

III) Chantiers techniques

Comme dis plus haut, le cœur du besoin se trouve dans les alertes dues aux incidents et dans la planification des maintenances. Nous prévoyons donc de renforcer ces besoins dans notre architecture.

Les alertes seront mises en place durant le second bimestre. Un composant *Notification* sera chargé de communiquer à certains clients qu'un incident a été émis. Le protocole de communication entre le serveur et le client est encore à déterminer.

Le planning pour les équipes des départements sera également mis en place, un algorithme sera en charge d'optimiser un maximum les plannings des différents départements, et cela, en réagissant lorsque des évènements imprévus surviennent.

Nous souhaitons également automatiser la planification de certaines maintenances. Étant donné que certaines maintenances sont faites à partir de règles strictes (nombre de kilomètres atteint, date de dernière maintenance …) ou de données de capteurs (freins, moteur …). Un composant *Clock* va être ajouté il sera en charge de déclencher (pendant une période où le système est le moins demandé) un appel à un service externe via le nouveau composant *Equipment.*

Enfin nous voudrions permettre une assignation automatique d'un département à une tâche. Ainsi une fois un incident créée le système va se charger de trouver le meilleur département (proximité, disponibilité …) pour lui assigner la tâche.

La figure 2 représente le diagramme d'architecture final vers lequel nous voulons tendre.

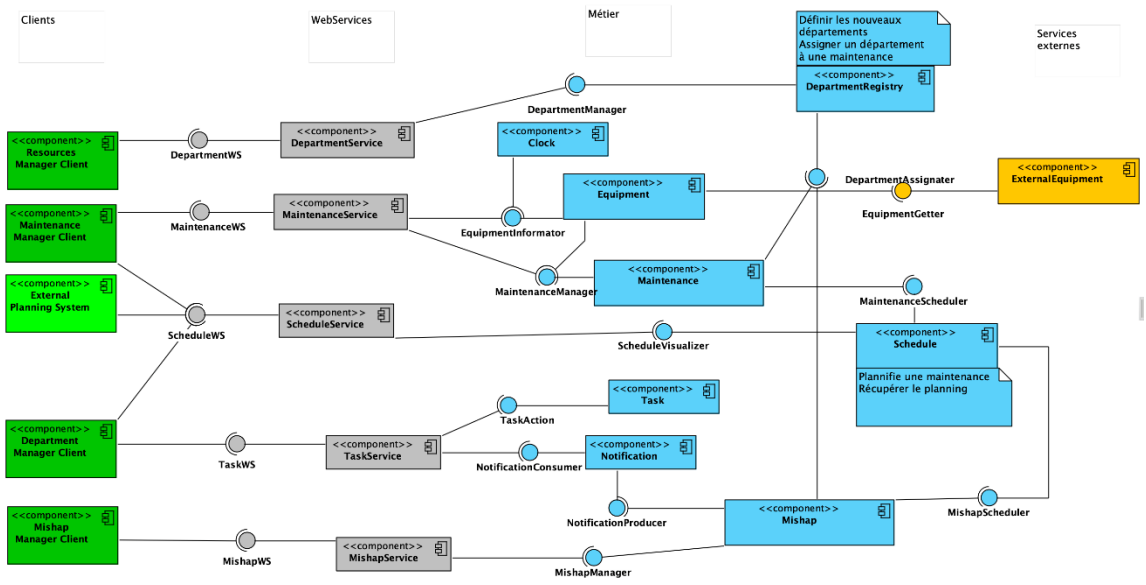

Figure 2 : Schéma d'architecture prévisionnel

IV) Si j'avais su

La première chose que nous aurions dû faire lors du premier bimestre est la définition précise du métier. Le métier étant très peu connu, il aurait été préférable de réfléchir aux besoins principaux sur lesquels concentrer notre réflexion pour définir une architecture bien adaptée au cœur du besoin. Ne pas avoir précisément cerné le cœur du besoin auparavant a engendré un manque de logique interne dans le système et une déportation de la logique vers l'utilisateur.

Un bon composant à implémenter aurait été le composant Schedule qui détient beaucoup de logique métier et qui est essentiel à avoir pour démontrer que notre système est capable de faire communiquer les différents composants métier.

Le framework Spring Boot a été choisi en fonction de nos connaissances en Java et non en fonction du beosin. Il aurait fallu changer notre démarche et partir du besoin architectural pour choisir le

framework. Celui-ci s'est finalement avéré être un bon choix puisque Spring Boot est le seul framework que nous maitrisons qui permet l'injection de dépendance nativement (contrairement à Node JS par exemple) et permet d'utiliser un ORM intégré.