Équipe F

CANAVA Thomas - GARDAIRE Loïc - JUNAC Jérémy - MORTARA Johann

BlablaMove F

Projet AL SI5 V5

Scope initial

- Recherche d'une solution
  - Appel d'un service minimum et éventuellement un service externe
- Pricing
  - Appel d'un service externe
- Possibilité de signaler un problème / une déviation
- Notification de problème / reprise du parcours
- Gestion d'erreur en cas de non-disponibilité
- Contact de l'assurance
- Gestion des retards
- Notre service est appelé par le reste de l'application et ne gère que ce qui est décrit dans ce document.

Utilisateurs

- L'étudiant Norbert voulant déplacer ses cartons
- Le conducteur Gérard du véhicule
- Deux autres conducteurs, Jacqueline et Camille, arrivant sur la même route et prêts à prendre le carton
- Une étudiante Georgette prête à accueillir un carton

Scenarii Cockburn

Scénario : Le conducteur a un accident

1. La voiture tombe en panne.
2. Le colis n'a pas de dégâts.
3. Le conducteur signale le problème.
4. Le propriétaire est notifié.
5. L'application recherche une solution.
6. L'application trouve un conducteur proche pouvant reprendre le trajet pour le colis.
7. Lorsque Jacqueline récupère la marchandise, elle le signale.
8. Le propriétaire est notifié du départ du carton.

- Variantes

- 2-3.a Dommage sur le colis
  - 2. Le colis est endommagé.
  - 3. Le conducteur signale les dégâts et l'application envoie les données à l'assurance.
- 6-9.a Il n'y a pas de conducteur
  - 6. L'application trouve un étudiant proche pouvant garder le colis jusqu'à ce qu'un conducteur puisse reprendre le colis.
  - 7. Georgette notifie la réception du colis.
  - 8. Le système redirige Camille vers l'appartement de Georgette.
  - 9. L'application notifie le propriétaire lors du départ du carton.

Scénario : Le conducteur change de direction

1. Le conducteur Gérard souhaite changer de direction
2. L'application recherche une solution.
3. L'application trouve un conducteur proche pouvant reprendre le trajet pour le colis.
4. Lorsque Jacqueline récupère la marchandise, elle le signale.
5. Le propriétaire est notifié du départ du carton.

- *Variantes*
  - 3-6a. Il n'y a pas de conducteur
    - 3. L'application trouve un étudiant proche pouvant garder le colis jusqu'à ce qu'un conducteur puisse reprendre le colis.
    - 4. Georgette notifie la réception du colis.
    - 5. Le système redirige Jacqueline vers l'appartement de Georgette.
    - 6. Jacqueline notifie le propriétaire du départ du carton.

Scénario : Un conducteur a du retard

1. La personne qui a actuellement le colis souhaite s'arrêter sur le chemin.
2. Le système notifie le propriétaire du colis et éventuellement le transporteur suivant.
3. Le retard ne dérange pas le propriétaire.
4. Le conducteur notifie le propriétaire lors du départ de l'escale.

- *Variantes*
  - *2-5a. Le retard dérange le propriétaire ou le transporteur suivant*
    - 2. Le système recalcule un trajet depuis le point d'arrêt actuel.
    - 3. Le système notifie Camille du nouveau colis à récupérer.
    - 4. Lorsque Jacqueline récupère la marchandise, elle le signale.
    - 5. Le propriétaire est notifié du départ du colis.

Roadmap

Stratégie pour livrer un POC

Pour réaliser notre POC, ce qui nous semble le plus important à faire est de réaliser le scénario "*Le conducteur a un accident*", sans les variantes. Ceci nous permet de passer à travers toutes les couches de notre projet et donc d'en avoir une version minimale.

Les composants utilisés pour ce scénario sont les composants qui sont nécessaires à la réalisation de tous nos scénarii. Il semble donc intéressant de les intégrer à notre POC.

Pour réaliser cette preuve de concept, nous avons comme objectif de coder les composants nécessaires et de leur appliquer des scénarii d'acceptation permettant de démontrer les fonctionnalités.

De plus, nous souhaitons mettre en place l'intégration continue dès le début pour s'assurer qu'il n'y a pas de régression à chaque modification du code.

Prévisions après le POC

- Nous continuerons en implémentant les variantes du scénario principal.
- Pour la suite, nous comptons développer en priorité les retards, car ils sont plus complexes fonctionnellement que les déviations. En effet, il existe toute une partie *"Est-ce que le retard dérange le propriétaire ?"* qui n'est pas présente dans la déviation.
- La dernière fonctionnalité sera l'ajout des déviations comme cas de problème.

Rôles

- Thomas : Integrator, user experience expert
- Loïc : Team leader, architecture consultant
- Jérémy : Tech leader, test expert
- Johann : Scrum master, Product owner

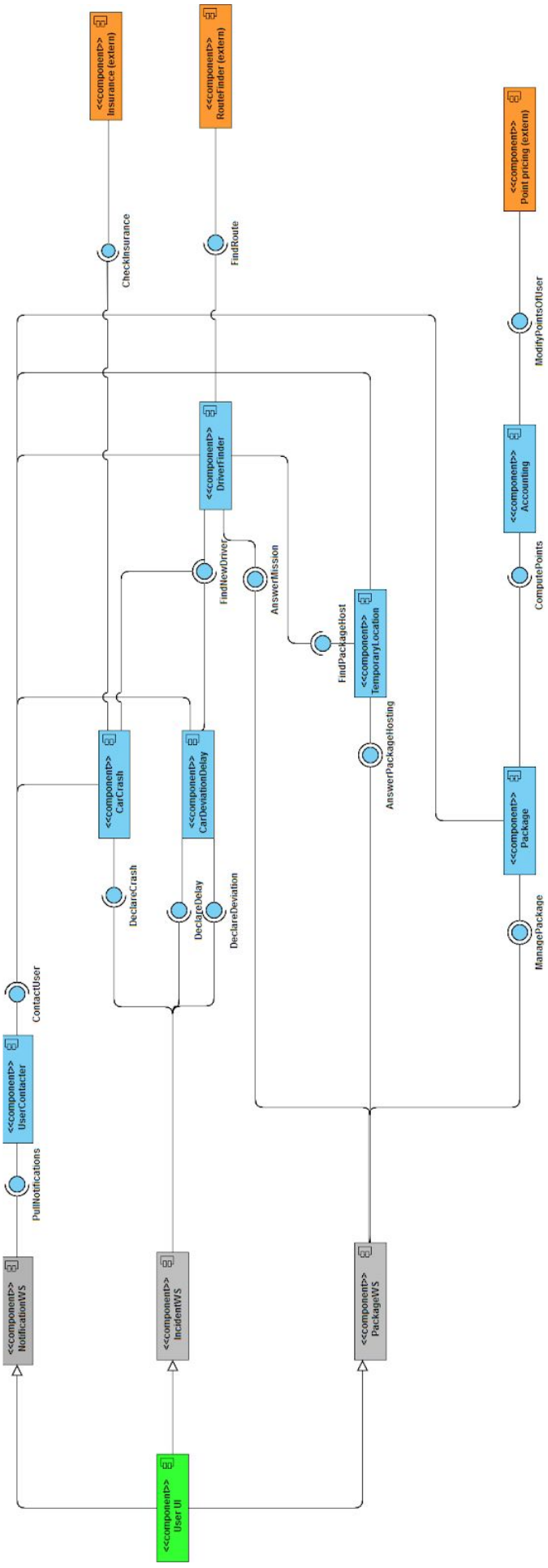
Situation à la fin du POC

Arrivés au terme de notre POC, nous avons fait face à un problème. Tous les composants faisaient appel au composant responsable de l'envoi de messages aux différents utilisateurs du système, comme visible sur le schéma de l'architecture à la fin du POC ci-dessous. Tous les services étaient liés au service *UserContacter*.

De plus, nous n'avions aucune sécurité au niveau de la communication avec les services externes. Si l'un d'eux ne pouvait plus répondre, le système complet était impacté.

Équipe F

CANAVA Thomas - GARDAIRE Loïc - JUNAC Jérémy - MORTARA Johann

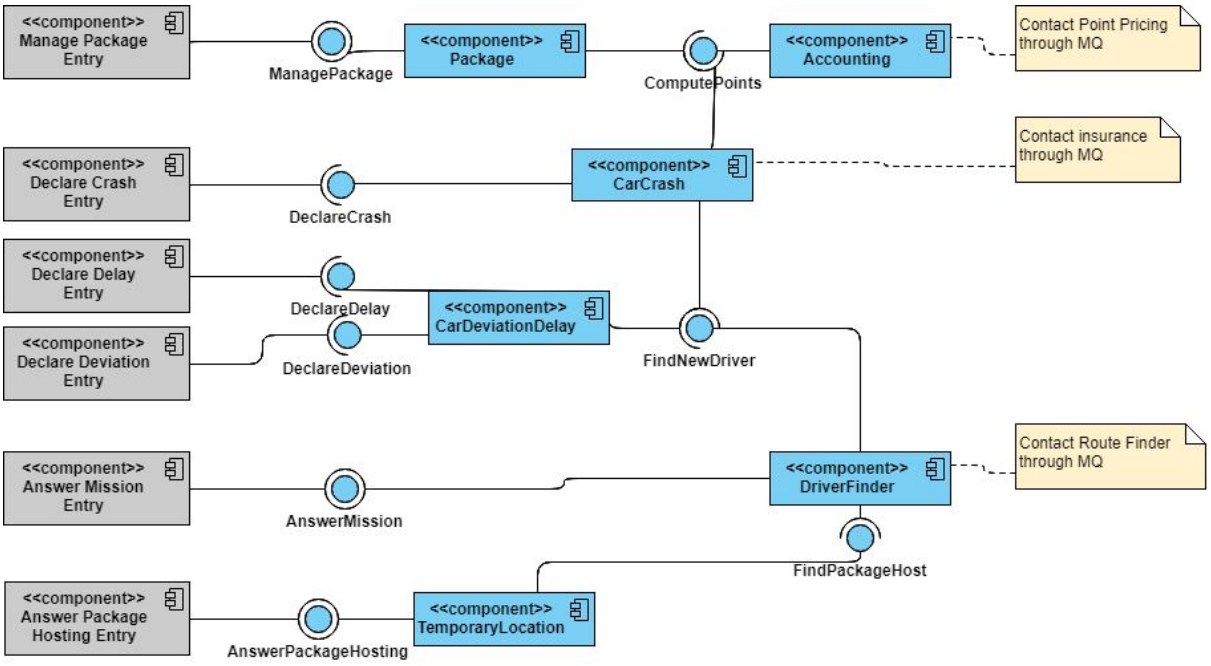


Nouvelle fonctionnalité: Chaos Monkey

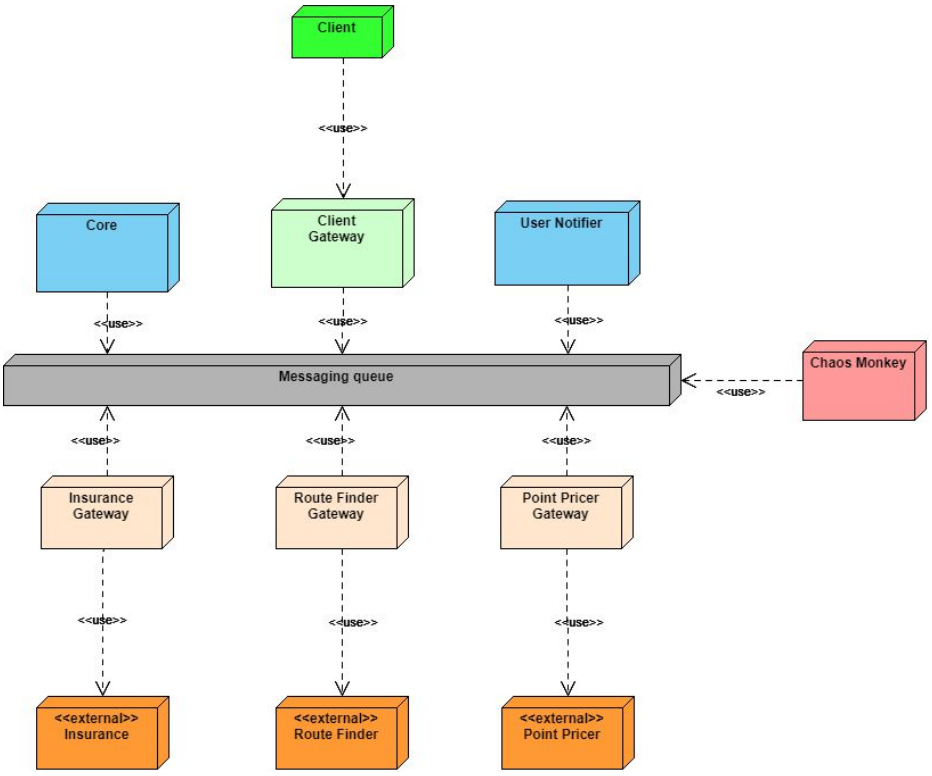
En plus du scope initial, nous devons maintenant ajouter un Chaos Monkey externe à notre application. Il aurait pour but de faire échouer une ou des opérations de notre application de temps en temps. Cette chance d'erreur doit être réglable, et nous devons ensuite simuler des trajets, avec ce système en fonctionnement. Le principal challenge est qu'il doit être externe à notre système. Nous devons donc pouvoir modifier le système de l'extérieur, sans pour autant modifier le métier présent ou surcharger notre monolithe (sans faire des appels externes à chaque requête par exemple).

Notre architecture

L'impact des modifications expliquées ci-dessus est visible dans les schémas de l'architecture ci-dessous.



Ce schéma représente le contenu du *Core* visible sur la première représentation. Ce dernier contient toutes les fonctionnalités présentes lors du POC, avant l'intégration du *Chaos Monkey*.



Ce schéma présente notre nouvelle architecture. Nous avons fait le choix de ne pas faire communiquer directement les services externes avec la file de messages, car cela aurait impliqué de devoir modifier leur implémentation, ce qui n'est pas concevable. Par conséquent, nous avons décidé d'extraire du monolithe la communication avec les services externes dans des *gateways*, une pour chaque service. En recevant le message de la file, la *gateway* va contacter le service externe et renvoyer un message résultant de l'appel du service. De cette manière, si une modification du format des messages s'impose, les services externes ne sont pas impactés.

Justification de l'architecture

Dans notre variante, nous devons gérer les problèmes qui peuvent survenir sur la route. Nous avons donc pris la décision de séparer le cas d'un crash et les cas de retard ou de déviation. En effet, un retard ou une déviation sur le trajet vont utiliser les mêmes composants pour résoudre le problème. À l'inverse, un crash oblige un contact avec l'assurance en cas de dommages sur le colis. L'assurance étant hors de notre scope, elle est considérée comme un composant externe ici.

Nous avons un composant nommé *DriverFinder* permettant de chercher un conducteur pouvant reprendre le colis. Ce composant fait appel à un composant externe (*RouteFinder*) nous permettant de trouver la meilleure route pour le colis en partant du lieu où il a été abandonné. Encore une fois, cette fonctionnalité sort de notre scope et est donc utilisée comme un composant externe. Si aucun conducteur n'est trouvé ou qu'il indique qu'il ne prend pas le paquet, le composant *DriverFinder* contacte *TemporaryLocation* ayant pour rôle de trouver un hôte proche pouvant accueillir le colis en attendant qu'un conducteur puisse le récupérer.

Dans le cas d'un crash, comme dit précédemment, nous contactons l'assurance via le composant externe, en plus du scénario expliqué ci-dessus.

Le composant *UserContacter* présent à la fin du POC disparaît au profit d'une file de messages. Il permettait de mettre en lien tous les acteurs du système. À chaque fois qu'un composant a besoin de notifier un utilisateur, il envoyait les informations nécessaires à ce composant qui se chargeait de notifier le destinataire. Il permettait aussi de poser des questions à l'utilisateur, et de préciser la méthode qui va recevoir la réponse. Ceci n'existe plus et les notifications passent maintenant par la file de messages.

Ensuite, nous avons un composant *Package*. Comme son nom l'indique, il contient toutes les fonctionnalités métiers concernants les paquets. Il permet notamment de déposer le paquet à un hôte, d'indiquer que le nouveau conducteur récupère le paquet, etc.

Enfin, nous avons un dernier composant pour la gestion des points des utilisateurs appelé *Accounting*. Il est chargé d'adapter l'interface du service externe de la gestion des comptes à notre métiers.

Pour faire la liaison avec ces composants, nous avons fait le choix d'avoir 3 *Web Services* : un dont le seul rôle est la gestion des paquets et un autre permettant de connecter les composants résolvant les différents problèmes de nos utilisateurs. Le dernier gère les notifications entre les différents utilisateurs. Le choix a été fait de séparer ces responsabilités en 3 *Web Services* car chacun a un métier différent.

Justification de nos choix techniques

Notification avec réponse

Pour l'attribution d'une mission de transport ou l'hébergement d'un colis, nous envoyons une notification sous forme de question. Le client a juste à répondre oui ou non, et le client renvoie la réponse au serveur. Ce mécanisme de réponse est opéré grâce à la facilité de JSON-RPC. En effet, la notification possède un champ réponse, si une réponse est nécessaire, qui contient toutes les informations pour effectuer la réponse. On a donc la route, le nom de la méthode, ainsi que tous les paramètres autres que la réponse bien sûr. Bien que ceci soit possible quelque soit le paradigme (REST, RPC, Message) ou le type de RPC choisi (RMI, SOAP), le fait que JSON-RPC repose sur un contrat léger facilite grandement l'implémentation. Avec un contre-coup cependant, si on change de route ou de nom de méthode, nous devons changer la réponse à la notification.

Relation mission-colis

Nous avons tout d'abord voulu représenter une course par un objet *Mission*, qui aurait été une représentation d'une liste de colis possédés par un utilisateur et transportés par un autre utilisateur d'un point de départ à un point d'arrivée. Cependant, nous nous sommes aperçus au cours du développement que certains cas demanderaient des manipulations assez particulières au niveau des colis, par exemple :

- Un accident survient et un conducteur de remplacement ne peut prendre qu'une partie des colis transportés par le conducteur accidenté
- Un conducteur a en sa possession des colis appartenant à des utilisateurs différents

En effet, dans ces cas-là, la représentation et la manipulation des données dans le code deviendraient complexes. Nous avons donc fait le choix d'avoir un unique colis par *Mission*, ce qui permet donc d'avoir par utilisateur plusieurs listes de *Missions* (possédées, transportées) et donc de pouvoir plus facilement gérer ces cas particuliers.

Utilité de la file de messages

Pour résoudre les problèmes soulevés à la fin du POC, nous avons choisi de mettre en place une file de messages, car cette dernière permet de corriger les problèmes existants tout en intégrant les nouvelles fonctionnalités demandées.

Premièrement, la file de messages permet de sécuriser la communication avec les services externes. En effet, avec l'ajout d'une *Gateway* par service externe, si l'un d'eux ne peut plus répondre, la file de messages servira de tampon pour les messages entrants de ce service. Nous utiliserons ainsi la fonctionnalité d'essais

multiples de notre file.

Deuxièmement, la file de messages vient aussi en aide au service de messages aux utilisateurs du système. Elle permettra de gérer les flux et de mettre en file les demandes de messages si un afflux trop important arrive.

Enfin, troisièmement, la file de messages nous permet d'intégrer la nouvelle fonctionnalité demandée : le Chaos Monkey. Ce dernier sera connecté comme tous les autres composants à la file et pourra alors jouer son rôle en supprimant aléatoirement des messages par exemple, comme visible sur le diagramme de notre architecture globale présenté précédemment.