Rapport final SOA

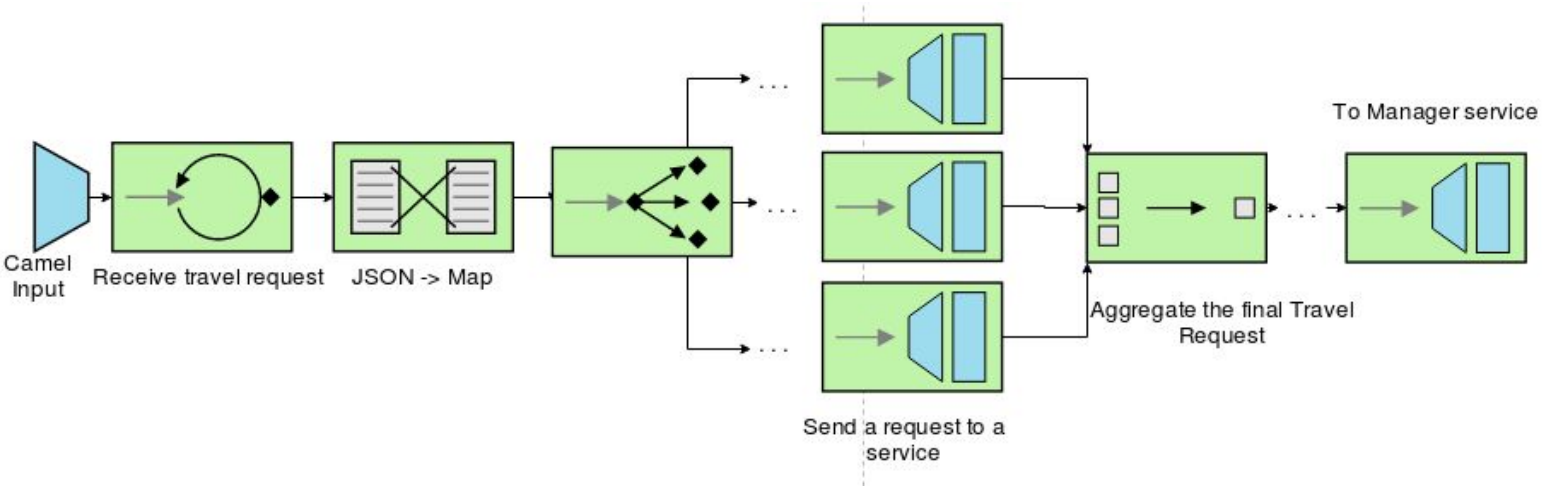
Thierno Balde, Anthony Fusco, Florent Pastor, Thomas Suignard

Introduction :

Le but de ce projet était de réaliser un outils de gestion de voyages dans lequel il fallait intégrer des services externes pour satisfaire à une demande de voyage et de remboursement de voyage.
Nous décrivons dans ce rapport tout d'abord l'intégration des services pour requêter les voyages.
Nous parlons ensuite de la partie intégration pour le remboursement des dépenses.
Enfin, nous décrivons nos tests d'intégration et le déploiement.

Description de l'intégration de la partie "Création de la requête de voyage" :

Voici un diagramme haut niveau du flot d'intégration :



Le bus d'intégration reçoit un fichier json contenant au maximum une demande de vol, d'hôtel et de location de voiture, chaque type de demande est séparé puis dirigé vers une route capable de la traiter. Chaque route retourne ensuite un objet correspondant au résultat de la requête de l'employé (ou un objet nul) et les résultats sont agrégés en un seul objet, puis envoyés vers le service d'acceptation des requêtes par les managers.

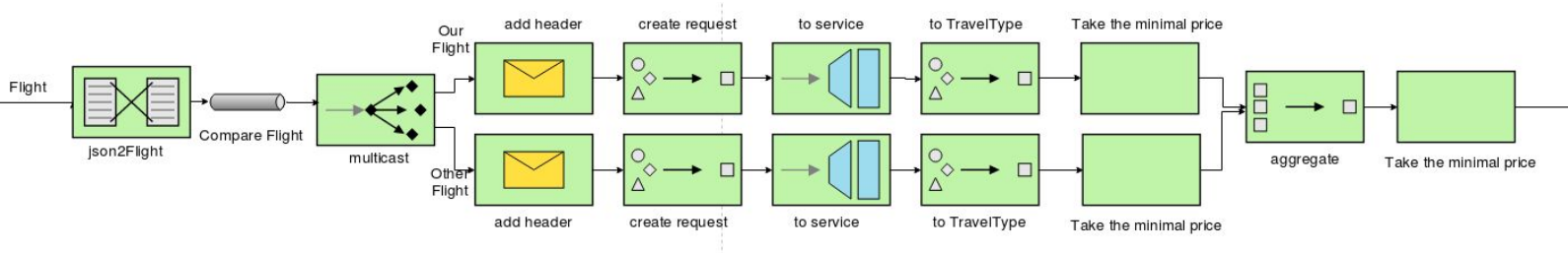
On distingue plusieurs choix de design :

Le "channel adapter" Camel Input qui constitue l'entrée du bus nous semble être la meilleure solution. Il est d'abord facile d'adapter un service externe pour qu'il redirige ses requêtes vers cette entrée. Ensuite, dans le cas ou le bus est éteint les requêtes peuvent s'accumuler et seront traités au réveil du bus, les requêtes ne sont alors pas perdues.

Le “split” et le routage du message d’entrée vers plusieurs routes différentes nous permet de séparer les responsabilités de traitement de chaque type de requêtes. De plus cette partie est facilement extensible, l’ajout d’une nouvelle branche n’impacte pas les autres.
Les messages non reconnus par nos POJO sont dirigés vers une queue spécifique.
Il est important que des erreurs dans le fichier d’entrée ne fassent pas tomber le bus, un doTry/doCatch est utilisé pour catcher les erreurs de dé-sérialisation et rediriger les messages vers une queue réservée.
Ce design nous permet aussi d’effectuer toute les opérations spécifiques à chaque type de requêtes en parallèle.

L'agrégation des réponses est déterminé complète par le bus lorsque le processus a reçu autant de messages qu’il y a eu de requêtes en entrée (e.g. si la requête contient un hotel et un vol, le processus d’agrégation attend deux messages). Cela nous permet d’avoir un nombre variable de requête en entrée et d’être sûr de toutes les recevoirs en sortie. Il est par contre possible que ce système bloque le bus si un problème se passe dans une des routes de traitements des requêtes et qu’un message reste bloqué. Nous verrons plus tard comment nous avons géré les exceptions pour éviter ce scénario. Malgré tout, ils nous semble y avoir là un risque pour la résilience du bus, et d’autres alternatives doivent être étudiées.

Nous allons maintenant présenter les routes de traitement des requêtes :



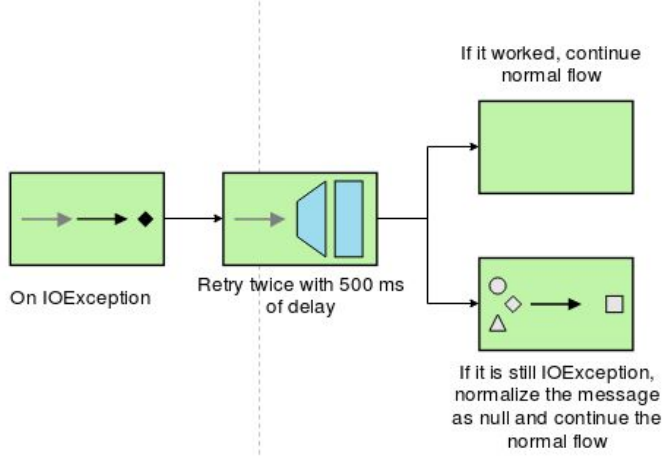
La requête est d’abord traduite du json vers un objet et placée dans une queue. Le message est dupliqué entre deux routes qui seront traitées en parallèle, une pour notre service et une pour le service d’un autre groupe. L’objet est transformé en requête HTTP que le service peut interpréter. La réponse du service est ensuite traduite et l’objet avec le prix minimum est envoyé à l'agrégateur. Finalement, le prix minimal entre les deux service est sélectionné.

Ce design est semblable au design précédent et possède les mêmes caractéristiques. La queue à l’entrée de la route permet d’accumuler les messages en cas de temps de réponse lent des services externes et le pattern multicast nous permet de traiter les messages en parallèle. Des détails d’intégration sont masqués par les patterns “normalize”, en effet il nous faut toujours implémenter une logique pour adapter les entrées/sorties des différents services avec les entrées/sorties du bus.

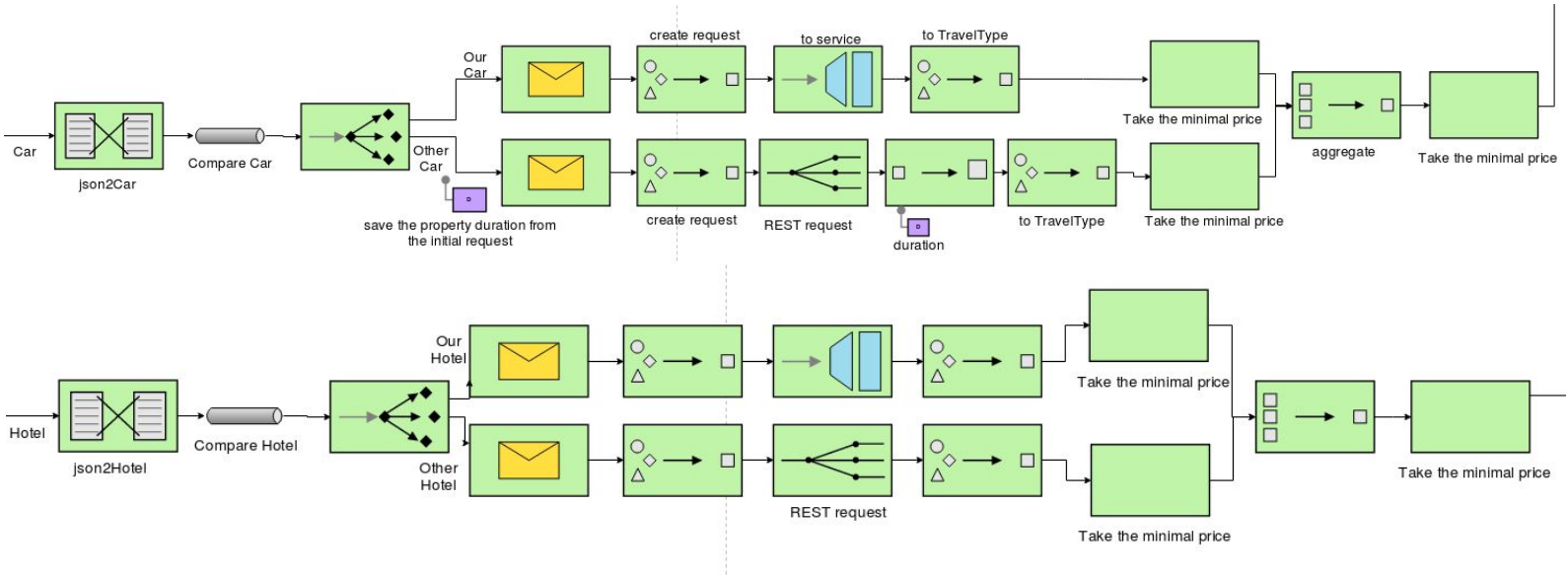
Il se pose maintenant des problématiques de disponibilité des services externes.

Tout d'abord, au début du projet nous avions pris la décisions de retourner un message nul si aucun service ne peut être atteint ou si aucune ressource des services ne correspond à la requête de l'employé. Ce message signifie donc que "rien n'a été trouvé" ainsi que "rien n'est disponible". Ceci est problématique pour le feedback final retourné à l'utilisateur et devrait être modifié. Néanmoins cette propriété est prise en compte dans tout le code pour s'assurer de toujours retourner un messages "au pire vide" et permettre au bus de continuer à traiter les requêtes.

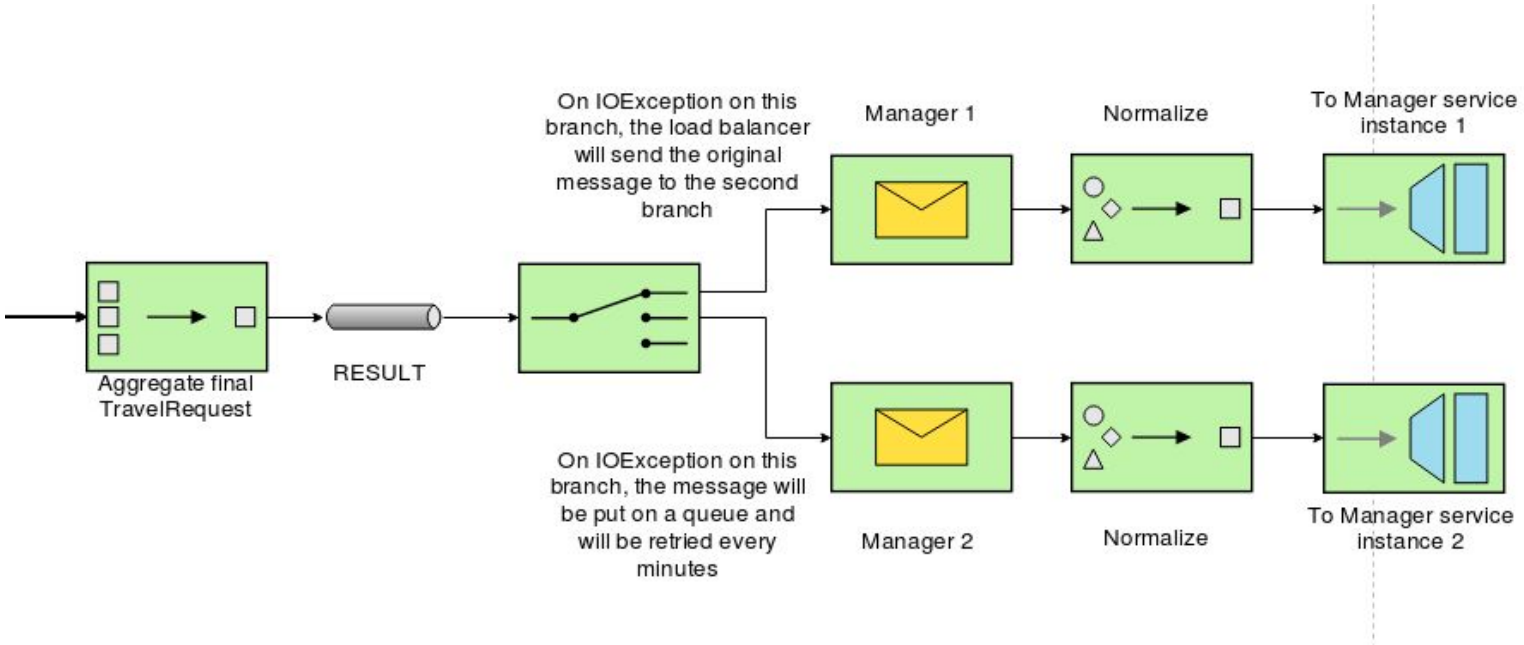
Pour toutes les routes contenant un appel vers un service externe, nous avons déclaré un gestionnaire d'exceptions (onException) qui va catcher les IOException et réessayer la requête au plus deux fois avec 500 ms de délai. Cela nous permet en cas d'interruption temporaire de la connection avec le service d'avoir une chance de bien envoyer la requête. Si cela suffit à recevoir une réponse du service, la réponse est propagée et le bus reprend son fonctionnement normal. Sinon, un message nul est propagé à la place et le bus reprend son fonctionnement normal. Nous avons donc préféré la vitesse de réponse du bus plutôt qu'un grand nombre de renvoi de la requête.



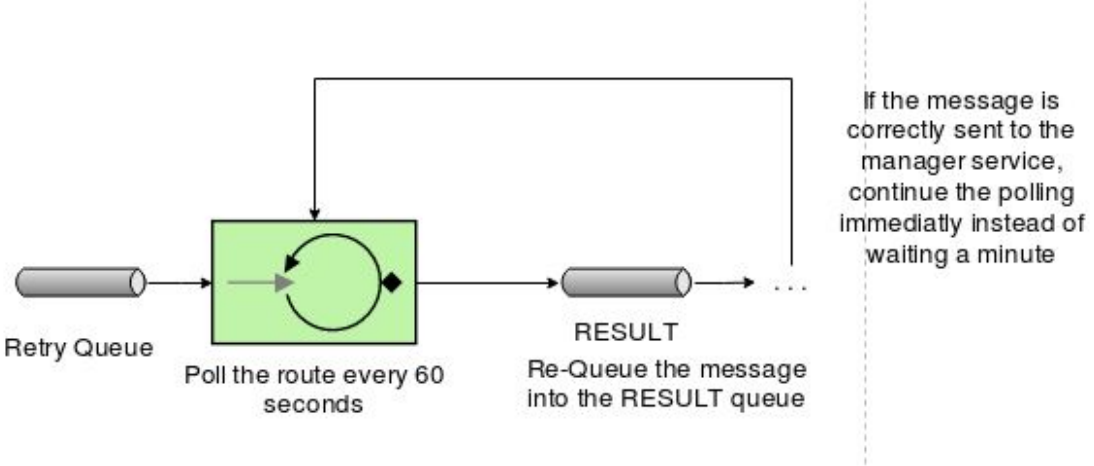
Les routes des autres type de requêtes sont similaires :



La dernière tâche de ce flot est de notifier les managers qu'une nouvelle requête est en attente. Cela nous emmène après l'agrégation des messages venant des services. :



Pour ce flot, nous avons voulu aller plus loin dans la gestion de la disponibilité du service. En effet cette partie du flot et spécifiquement ce service étant plus simple à intégrer, nous avons voulu ajouter de la valeur en augmentant la résilience autant que possible.
Pour cela, nous avons dupliqué le container du service et utilisé le composant "load balancer" de Camel avec comme stratégie de décision "failOver". Le load balancer commence par envoyer le message vers le premier container et à la levée d'une IOException va nettoyer la branche et envoyer le message d'origine vers le deuxième container. L'erreur du premier container est loggé et si le deuxième container répond le flot est terminé.
Si le deuxième container est lui aussi éteint, le bus réessaiera la requête deux fois comme pour les autres services, si il n'y as toujours pas de réponse alors le message sera mis dans une queue réservée.

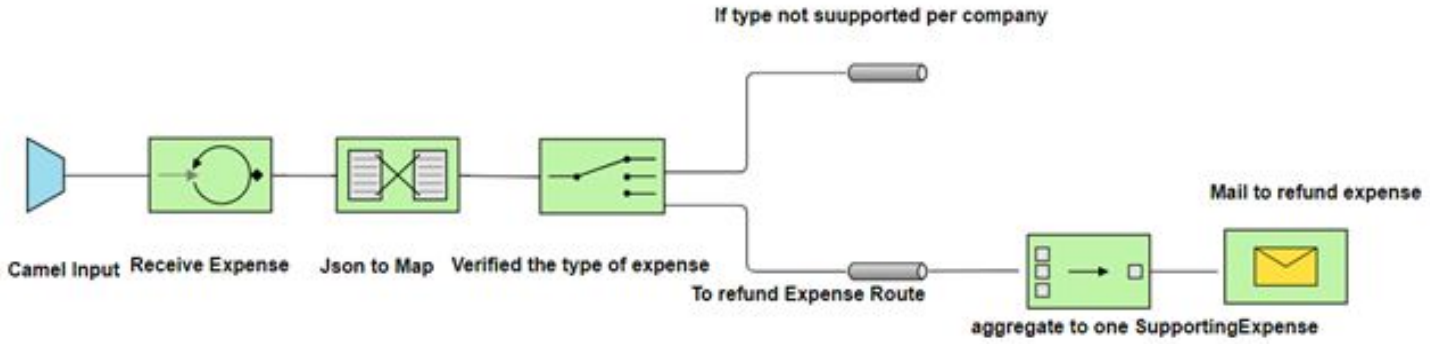


Une route de type “timer” va alors toute les minutes essayer de récupérer le premier message de la queue et le réinjecter dans la queue RESULT qui se trouve avant le load balancer. A partir de là, le flot normal pour ce message est repris.
La route videra la queue tant que les messages sont correctement envoyés au service.

On peut donc dire qu’une requête d’un employé finira toujours par atteindre la fin du flot. Le load balancer nous permet d’être plus réactif sur l'indisponibilité d’un service et le système de timer nous garantit que la requête sera éventuellement envoyée. Les deux instances pointent sur la même database, les même données sont donc toujours accessible.

Description de la partie “Remboursement des dépenses de voyage”

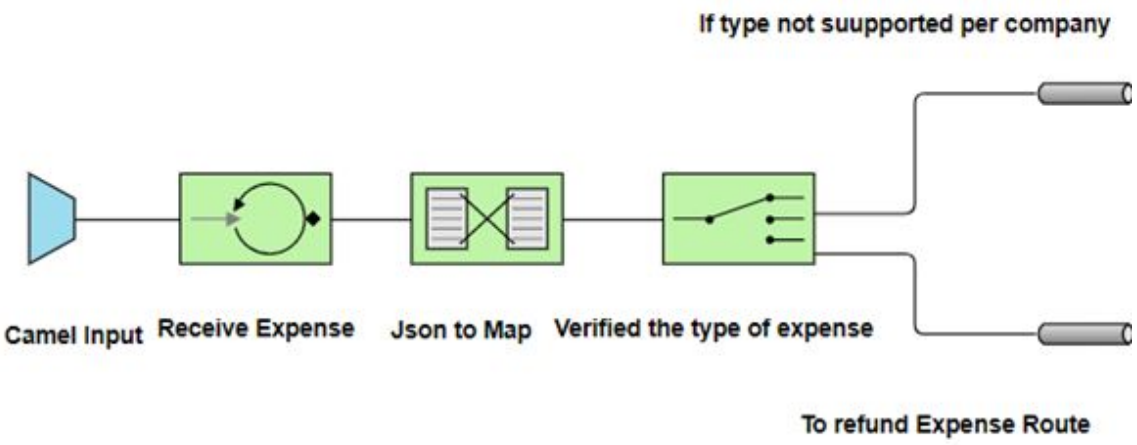
Ci-dessous un diagramme générique du flow



Le flow de « Remboursement des dépenses de Voyage » se déroule de la façon suivante :
Dès que nous recevons un mail de remboursement en format json, nous vérifions le type de dépense en question pour savoir si la compagnie le prend en charge, si oui nous regroupons tous les dépenses en un seul pour effectuer le remboursement en tenant compte du seuil de référence fixé par jour.

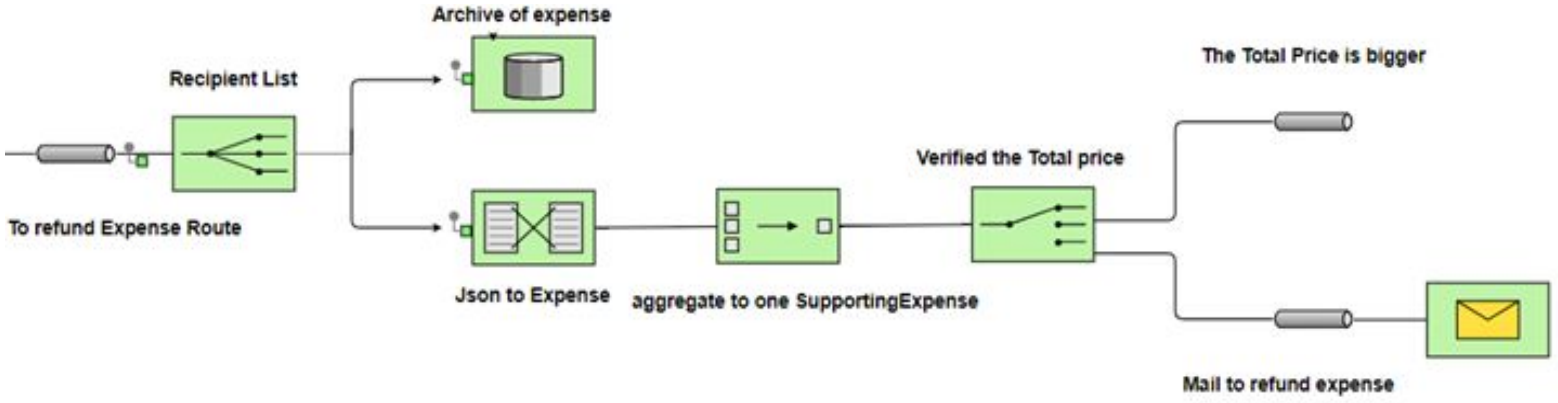
Pour cette seconde partie nous avons choisi dès la réception d’un message d’utiliser un content based router qui va nous permettre d’éliminer tous les types de dépenses que notre entreprise ne prend pas en charge. Ceci nous apporte l’avantage d’archiver par la suite que les dépenses que nous prenons en charge dans une base de données (dossier archive dans le répertoire camel).
Ensuite nous utilisons un agrégateur pour fusionner toute les dépenses que nous avons reçu une par une en seul objet de remboursement, ça nous permet de calculer le prix total afin de faire une vérification pour savoir si l'employé n’a pas dépassé le seuil de référence du lieu.
Maintenant nous allons détailler la route.

- Première Partie de la Route

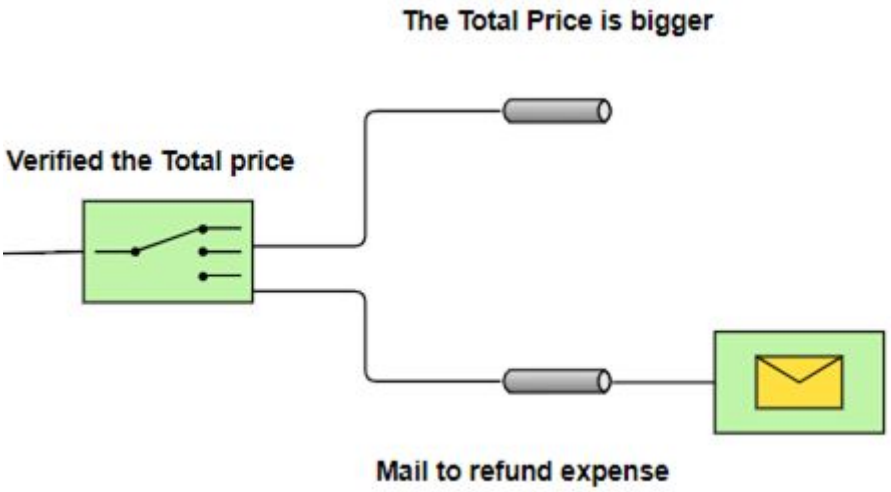


A chaque réception d'une dépense, nous la transformons en une map pour pouvoir faire un traitement sur le type de dépense, on regarde si le type fait partie des dépenses que nous remboursons, si oui on l'envoie sur une queue de messages de remboursement, sinon on l'envoie sur une queue de messages de non remboursement tout en affichant un message d'information.

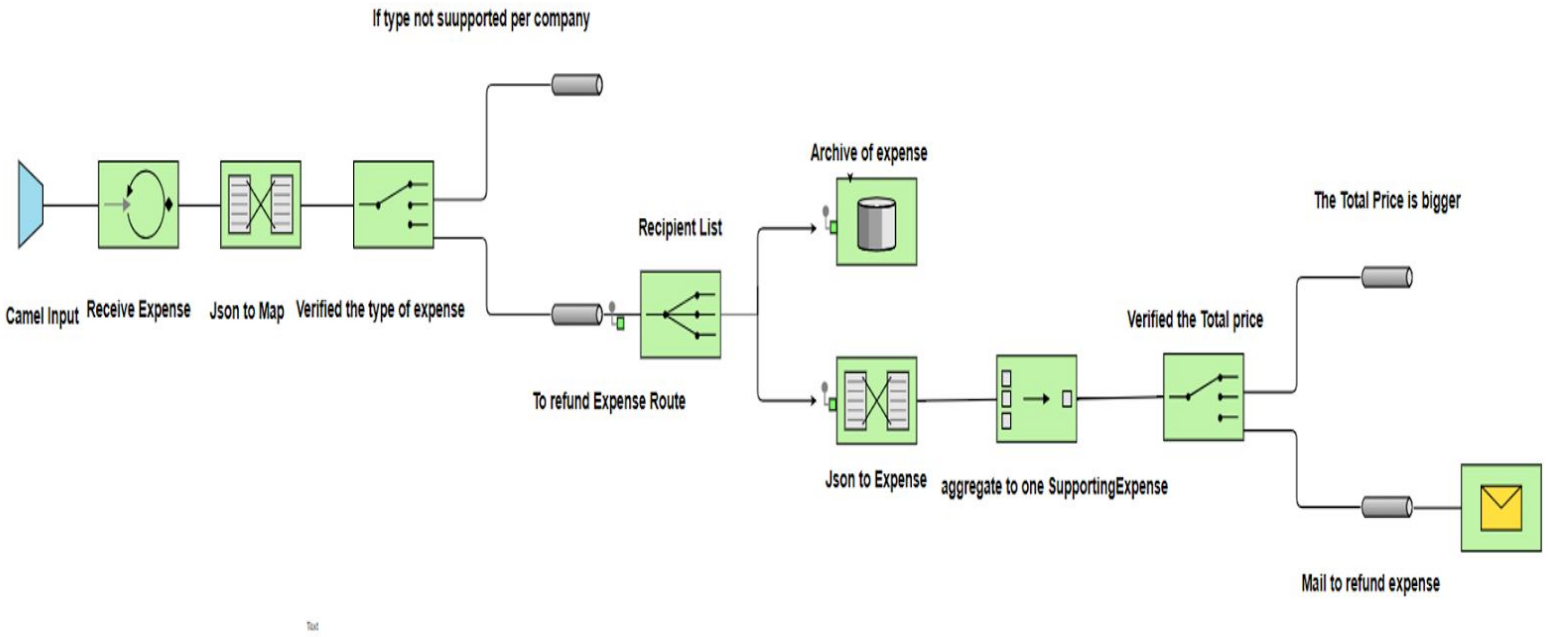
- Seconde Partie :



A ce stade nous possédons que les dépenses prise en charge par notre société, nous récupérons dans la queue de messages toutes les dépenses qui s'y trouvent, nous les archivons dans une base de données (le répertoire archive) par type de dépense.
Ensuite nous les agrégeons tous dans un seul objet de remboursement tout en calculant le prix total des dépenses.
Après nous récupérons le prix total calculé lors de l'agrégation, nous faisons appel à une route qui nous permet de vérifier si les dépenses de voyages de l'employé dépassent le prix référentiel.



Si les dépenses totales sont inférieures ou égales au prix référentiel, nous remboursons automatiquement l'employé en envoyant un mail d'informations résumant la somme totale qu'il va percevoir et les détails des dépenses qui correspondent à son remboursement.
Ci-dessous le flow détaillé dans son intégralité



A chaque demande de remboursement (route) nous avons 2 chances sur 3 de ne pas être remboursé automatiquement, la première est le cas où tous les types de dépenses envoyées ne sont pas prises en charge, la seconde est le cas où le prix total dépasse le seuil référentiel correspondant à son lieu de séjour.
NB :
Mail : Camel/mai/
Archive : Camel/archive/
Mail de remboursement : Camel/ refundEXP+id

Tests:

Sous /integration/flows/src/main/test
De multiples tests s'assurent que les communications avec les services intégrés sont corrects. Que les prix minimums sont toujours choisis, ainsi que plusieurs cas spéciaux soient gérés correctement.
Plusieurs tests de résilience autours de la gestion d'erreur sont aussi présents.
La plupart des tests/classes de tests sont documentés pour plus de précision.

Deployment :

Pour lancer l'application, il faut tout d'abord créer l'image Docker en lançant le script build.sh.
L'application peut être lancée en exécutant le script start.sh.
Le script run.sh permet de lancer des requêtes dans nos services et le script chaos.sh permet de stopper et de lancer aléatoirement des containers Docker à l'exception de la base de données et de l'ESB.
Afin de pouvoir utiliser les services des autres groupes, nous avons décidé de les mettre sur le Docker cloud et de mettre au préalable des données qui correspondent aux données de nos services pour pouvoir après faire des tests plus facilement. Nous avons aussi mis nos propres services sur le Docker cloud pour avoir des images figée et un environnement de déploiement plus propre.

Répartition du travail :

1. Création de la requête de voyage 70 %
   Anthony Fusco : 40%
      - Intégration des services
      - Tests
      - Gestion du bus avec Docker
      - Mise en place de la démo run.sh/chaos.sh

   Florent Pastor : 30%
      - Tests
      - Ajout des données dans les services
      - Gestion des services avec Docker
      - Script chaos.sh
2. Remboursement des dépenses de voyage 30%
   Balde Thierno
      - Remboursement des dépenses
      - Tests

   Thomas Suignard
      - Script chaos.sh

Remerciements :

Nous tenons à remercier nos professeurs Sébastien Mosser et Mireille Blay-Fornarino pour nous avoir apporté des conseils durant nos cours.