Introduction

Dans le laboratoire 6, je propose de mettre en oeuvre une Saga pour gérer les transaction impliquant plusieurs microservices. Il existe en deux approches principales : la Saga orchestrée et la Saga chorégraphiée.

- La Saga orchestrée repose sur un service central qui coordonne les différentes étapes de la transaction. Il appelle chaque microservice dans un ordre défini et prend les décisions en fonction des réponses obtenus. En cas d'échec à une étape, il peut déclencher des actions de compensation et interrompre le processus global.
- La Saga chorégraphiée fonctionne de manière décentralisée. Chaque microservice effectue sa tâche et déclenche un événement que les autres services peuvent écouter pour exécuter la prochain étape.

Dans ce laboratoire, j'utilise l'approche orchestrée et synchrone, afin de coordonner une série d'actions dépendantes entre plusieurs microservices.

Scénario métier

Le scénario proposé dans l'énoncé est idéal pour mon application. La création d'une commande client implique plusieurs microservices, chacun responsable d'une étape :

- Vérification de la commande
- Réservation des produits
- Paiement
- Confirmation ou annulation de commande
- Génération et envoi du reçu

Les microservices impliqués dans ce scénario sont :

- Service Commande
- Service Inventaire
- Service Paiement
- Service Produit
- Service Transaction

Architecture proposée

Pour commencer, il faut un microservice principal qui soit déclencé lorsque l'utilisateur clique sur le bouton pour démarrer sa commande. Ce microservice jouera le rôle d'orchestrateur. Il sera responsable de coordonner l'appel aux autre microservices nécessaire à la création d'une commande complète.

Pour ce faire, il convient de créer un microservice nommé OrchestratorService, qui reçoit un événement de début de transaction. Dans ce schénario, il attend l'événement `CommandeCreate`, déclenché par l'action de l'utilisateur.

L'orchestrateur exécutera ensuite les appels aux microservices concernés et dans l'ordre suivant :

- CommandeService : crée et initialise la commande.
- ProduitService : vérifie si le magasin possède les produits demandés.
- InventaireService : vérifie la disponibilité des produits et réserve les articles.

- PaiementService : effectue la transaction de paiement
- TransactionService: génère le reçu de la commande.

À chaque étape, l'orchestrateur évalue le résultat de l'appel :

- En cas de succès, il continue le processus jusqu'à la confirmation finale de la commande. Une fois toutes les étapes validées, il publie l'état `CommandeConfirm`.
- En cas d'échec, il déclenche des actions de compensation pour assurer la cohérence du système, puis publie un état de `CommandeCancel`. Par exemple, si le paiement échoue, il libère les produits réservés et annule la commande.

Gestion des événements et de l'état

Dans cette étape, chaque microservice impliqué dans la saga publiera un événement indiquant le succès ou l'échec de son traitement. Ces événements sont généralement représentés par des constantes ou des statuts propres à chaque microservice, correspondant aux résultats possibles de leurs opérations.

Cependant, seul le microservice `OrchestratorService` doit posséder une énumération définissant les états globaux de la saga.

Liste d'événements par chaque microservice :

- CommandeService : CommandeCreate, CommandeCancel et CommandeConfirm
- ProduitService : ProduitNotFound et ProduitExist
- InventaireService : InventaireReserve et InventaireFail
- PaiementService : PaiementCancel et PaiementConfirm
- TransactionService: TransactionFail et TransactionConfirm

Le fichier Enum de l'orchestrateur pourra ressembler à ceci :

```
enum SagaOrchesState {
    CommandeCreate,
    InventaireReserve,
    PaiementConfirm,
    TransactionGenerate,
    CommandeConfirm,
    CommandeCancel
}
```

Simulation de cas d'échec

Dans cette étape, il faut ajouter utiliser un mécanisme de gestion d'erreur `try... catch`. Puisque j'utilise Rust, ce sera géré avec `Ok et Err` pour traiter les succès et les erreurs lors de l'exécution du microservice `OrchestratorService`. En cas d'erreur `Err`, le service déclenchera un rollback via des actions de compensation.

Observabilité

Pour le suivi des sagas, Prometheus fournit plusieurs outils permettant de mesurer les indicateurs suivants :

- Durée moyenne d'exécution : Utiliser `HistogramVec` pour enregistrer la durée des sagas.
- Nombre d'échecs : Utiliser `IntCounter` qui incrémente à chaque fois que Saga échoue.
- Étapes atteintes : Utiliser `IntCounterVec` qui permet de savoir combien de fois chaque étape est réussie.

Après avoir créés, il faut intégrer ces métriques dans les microservices qui font partie du Saga Orchestrateur. Si l'intégration fonctionne correctement, la route `/metrics` exposera les données.