Cahier de TP
«Design Pattern pour les micro-services»

Outils utilisés :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux, MacOs, Windows 10
- JDK11+
- Spring Tools Suite 4.x avec Lombok
- Git
- Docker

Atelier 1: Décomposition

Les capacités métier de notre Magasin en Ligne :

- Prise et Gestion de commande
- Gestion du catalogue et stock produit
- Gestion des livraisons de commande

La décomposition a donc donné lieu à 3 services

- ***OrderService***
- ***ProductService***
- ***DeliveryService***

Les APIs ont été déduites de la Story « Passer une commande »

Récupérer les projets fournis et les importer dans vos IDE.

Démarrer chacun des services et accéder à la documentation Swagger :

http://localhost:<port>/swagger-ui.html

Tous les projets utilisent une base de donnée mémoire H2 qui est réinitialiée à chaque démarrage.

La console permettant de voir la base de données est disponible sur l'URL
http://localhost:<port>/h2-console

Atelier 2: Communication synchrone, Discovery et Circuit Breaker

2.1 Mise en place serveurs de discovery et de config

Récupérer le tag 2.1 des solutions

Importer les nouveaux projets ***eureka*** et ***config*** fournis

Déclarer dans votre fichier hosts, les lignes suivantes

```
127.0.0.1 annuaire
127.0.0.1 config
```

Les démarrer dans l'IDE, démarrer *config* puis *eureka*

Vérifier sur le serveur de config les URLS suivantes :

- http://config:8888/application/default
- http://config:8888/ProductService/replica

Vérifier le serveur eureka :

- http://localhost:1111

Démarrer les services applicatifs

Démarrer *product-service* 2 fois dont une fois en activant le profile ***replica***

Vérifier les bonnes inscriptions des 2 services dans Eureka

2.2 Communication synchrone et Load-balancing

Récupérer le tag 2.2 des solutions

Commentaire du code

2.3 Circuit Breaker Pattern

Dans le projet ***OrderService***, vérifier la dépendance sur *Resilience4j*

Encapsuler le code de l'appel du service *ProductService* dans un circuit breaker.

Atelier 3: Communication asynchrone

Nous mettons en place une communication de type Publish/Subscribe

ProductService publie l'événement TICKET_READY vers un topic

DeliveryService réagit en créant une livraison

Solution : tag 3

3.1 Démarrage du Message Broker

Récupérer le fichier docker-compose.yml permettant de démarrer le message broker Kafka et le processus ZooKeeper nécessaire

docker-compose up -d

3.2 Mise en place du producer

Ajouter les starters spring-kafka sur les projets ProductService

Implémenter une classe Service qui

- Crée charge un ticket
- Publie un événement *READY_TO_PICK* sur le topic Kafka ***tickets-status***, la clé du message et l'id du Ticket, le corps du message est une instance de la classe *ChangeStatusEvent*

3.3 Mise en place du consommateur

Ajouter les starters spring-kafka sur les projets DeliveryService

Implémenter une classe Service qui

- Ecoute le topic ticket-status
- Si l'événement est du type READY_TO_PICK, créer une Livraison

Vous pouvez tester le tout avec le script JMeter fourni

3.4 Messagerie transactionelle

Implémenter une messagerie transactionelle via le pattern Transaction Outbox

Atelier 4: Saga Pattern

Solution : Atelier4

Nous voulons implémenter la saga suivante :

Étape	Service	Transaction	Transaction de compensation
1	*OrderService*	*createOrder()*	*rejectOrder()*
2	*ProductService*	*createTicket()*	*rejectTicket()*
3	*PaymentService*	*authorizePayment()*	
4	*ProductService*	*approveTicket()*	
5	*OrderService*	*approveOrder()*	

Nous implémentons le pattern via un orchestrateur CreateOrderSaga de OrderService.

L’orchestrateur envoie des messages commande dans le style request/response :

- 1. *OrderCreatedEvent* sur le topic « order-staus »
- 2. *TicketCreatedEvent* sur le topic « ticket-status »
- 3. *PaymentRequestEvent* sur le topic « payment-request »et PaymentResponseEvent (contenant l’information si le paiement est accepté ou rejeté) sur le topic « payment-response »

Atelier 5: Logique métier

Reprendre le service *ProductService* qui implémente plutôt un *TransactionScriptPattern* (classe org.formation.service.TicketService) et appliquer les principes de ***DomainModel Pattern***

Définir une classe ResultDomain encapsulant :

- Un agrégat Ticket
- Un événement : TicketStatusEvent

Implémenter les méthodes createTicket, approveTicket et rejectTicket dans la classe Ticket.

Les règles sur les agrégats dans les différents services sont-elles respectées ?

Atelier 6: Service de Query

Point de départ de l'atelier, tag ***Atelier6***

Vous pouvez utilisez le script JMeter fourni pour initialiser les bases avec des données (Order, Ticket, Livraison)

6.1 API Composition

Nous voulons pouvoir visualiser toutes les informations d'un livreur affecté à une commande.

Créer un nouveau micro-service *OrderQueryService* qui applique le pattern API Composition pour implémenter cette fonctionnalité.

Ensuite, implémenter un point d'accès qui affiche toutes les commandes en cours

6.2 CQRS View Pattern

Implémenter le pattern CQRS, le service s'abonne aux événements métier ***OrderEvent*** et ***DeliveryEvent*** pour mettre à jour une table locale stockant la jointure entre Order et Livraison

Atelier 7: API Gateway

Création d’un nouveau service Gateway qui route les URLs exetrnes vers :

- Le endpoint permettant de créer une Commande
- Le endpoint permettant d’indiquer qu’un ticket est prêt
- Le endpoint permettant au livreur de prendre une Livraison
- Le endpoint permettant de faire des requêtes