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
- JMeter

Dépôt des solutions :
http://github.com/dthibau//dp-microservices-solutions.git

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 »

Opération système :

OrderService

- ***createOrder() :*** Creation d'une commande

Collaboration entre services

ProductService

- ***acceptOrder()***=> création de ticket pour rassembler les produits de la commande
- ***noteTicketReadyToPickpup()*** => Marquer la commande comme prête à être livrée

DeliveryService

- ***noteDeliveryPickup()*** => La livraison a démarrée
- ***updatePosition()*** => Met à jour la position du livreur

- ***noteDeliveryDelivered()*** => La commande est livrée

Récupérer le tag *Atelier1* des solutions

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 réinitialisée 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 Atelier2.1 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

Pour la suite des ateliers, il est conseillé de démarrer Eureka en ligne de commande pour de pas polluer la console de l'IDE via des messages de traces :

```
cd eureka
./mvnw clean package
java -jar target/eureka-0.0.1-SNAPSHOT.jar
```

2.2 Communication synchrone et Load-balancing

Récupérer le tag Atelier2.2 des solutions

Visualiser le code de *OrderService* et en particulier la classe *DependenciesConfiguration*

Utiliser le script JMeter fourni pour visualiser la répartition de charge

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.

Comparer avec le tag Atelier2.3

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

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

Déclarer kafka dans le fichier host

```
127.0.0.1 kafka
```

3.2 Mise en place du producer

Récupérer le tag Atelier 3.2

Visualisez les dépendances apportées par ***spring-kafka*** sur les projets *ProductService*

Visualiser la classe Service comportant 2 méthodes

- **`public Ticket createTicket(Long orderId, List<ProductRequest> productsRequest)`**
  Crée un ticket avec le statut *TicketStatus.CREATED*
- **`public Ticket readyToPickUp(Long ticketId)`**
  Modifie le statut du ticket en *TicketStatus.READY_TO_PICK*
  Publie un événement *READY_TO_PICK* sur le topic Kafka ***tickets***, la clé du message et l'id du Ticket, le corps du message est une instance de la classe *ChangeStatusEvent*

Noter l'annotation *@Transactional* sur la classe

3.3 Mise en place du consommateur

Reprendre le tag Atelier 3.3

Les mêmes dépendances ont été ajoutées sur *DeliveryService*

Visualiser la classe Service qui écoute les messages du topic ***ticket***

Vous pouvez tester le tout avec le script JMeter fourni

3.4 Messagerie transactionnelle

Implémenter une messagerie transactionnelle via le pattern ***Transaction Outbox***

Comparer avec le tag Atelier3.4

Atelier 4: Saga Pattern

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-status***, il attend une réponse sur le topic ***ticket-status***

2. A la réception ***d'un*** *TicketStatusEvent* sur le topic « ticket-status », si le statut est TICKET_PENDING, envoie d'une demande de paiement

3. A l'envoi d'u*n PaymentRequestEvent* sur le topic ***payment-request*** , il s'attend à la réception d'un *PaymentResponseEvent* (contenant l'information si le paiement est accepté ou rejeté) sur le topic ***payment-response***

4. A la réception de *PaymentResponseEvent,* approbation ou rejet de la commande et envoi le changement de status sur le topîc ***order-status***

Les services impliqués implémentent les réponses aux requêtes de l'orchestrateur.

Reprendre le tag Atelier4

Visualisez le code de la solution

Vous pouvez tester soit via l'interface Swagger de *order-service*, soit en utilisant le fichier JMeter *CreateOrder.jmx*

PayementService refuse la transaction si le *paymentToken* ne commence pas par le caractère « A »

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.

Comparer avec le tag Atelier5

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

Comparer avec la solution Atelier6.1

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

Comparer avec la solution Atelier6.2

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

Atelier 8: Tests

8.1 Test d'intégration

Reprendre le tag Atelier8.1

Visualiser la classe de test *LivraisonRepositoryTest*

L'exécuter et regarder les beans présents lors du test

8.2 Design By Contract et Test de composants

Reprendre le tag Atelier8.2

Côté producteur

Visualiser les contrats des projets ***OrderService*** et ***DeliveryService***

Générer les classes de test via ***./mvnw test-compile*** par exemple

Visualiser la classe de test puis exécuter les tests.
Les serveurs de *config,Eureka* et le broker Kafka sont démarrés lors de leur exécution

Une fois que les tests passés, installés l'artefact dans votre dépôt Maven :
./mvnw install

Effectuer le même processus pour les 2 projets producteur

Côté consommateur

Dans le projet ***order-query-service***

Visualiser la classe de test et les références aux 2 précédents contrats publiés

Effectuer le test en isolation.

Atelier 9: Observabilité

9.1 Actuator

Visualiser les URLs exposés par Actuator , en particulier l'Health Check API

9.2 Métriques CircuitBreaker

Ajouter les dépendances suivantes au projet *Gateway* :

- `io.github.resilience4j :resilience4j-micrometer`
- `io.micrometer :micrometer-registry-prometheus`

Vérifier celle *d'actuator*

Démarrer le micro-service et le solliciter par le script JMeter

Visualiser les métriques à l'URL ***http://<server>/actuator/metrics***

Récupérer le répertoire ***grafana*** fourni et y exécuter *docker-compose up -d*

Cela doit démarrer un serveur Prometheus et Grafana

Se connecter sur Grafana à http://localhost:3000 et se connecter avec admin/admin

La source de donnée Prometheus et le Dashboard Grafana sont déjà configurés dans les containers.

9.3 Tracing avec Sleuth et Zipkin

Démarrer un serveur Zipkin

```
docker run -d -p 9411:9411 –name zipkin openzipkin/zipkin
```

Accéder à http://localhost:9411/zipkin/.

Ajouter pour tous les services la dépendance sur le starter ***zipkin*** et ***sleuth***

Les redémarrer et visualiser la différence dans les traces

Ajouter pour chaque micro-services utilisant ***sleuth*** et ***zipkin*** la configuration suivante :

```
spring.zipkin.base-url=http://localhost:9411/
spring.sleuth.sampler.probability=1
```

Visualiser les traces dans Zipkin

Atelier 10: Démarrage de la stack via docker-compose

Tag Atelier10

Construire le projet config :

```
cd config
./mvnw clean package
```

Construire les images docker :

```
cd <workspace>
./mvnw spring-boot:build-image
docker-compose build
```

Démarrer la stack

```
docker-compose up -d
```

Vérifier les logs :

- Inscription à l'annuaire Eureka
- Connexion avec le broker

Atelier 11: Kubernetes

Démarrage *minikube* ou *kind*

Accéder au dashboard

11.1 : Déploiements à partir d'une image

Créer un déploiement à partir d'une image docker
```
kubectl create deployment delivery-service
--image=dthibau/delivery-service:0.0.1-SNAPSHOT
```
Exposer le déploiement via un service
```
kubectl expose deployment delivery-service --type LoadBalancer \
  --port 80 --target-port 8080
# Vérifier exécution des pods
kubectl get pods
# Accès aux logs
kubectl logs <pod_id>

kubectl get service delivery-service
#Forwarding de port
kubectl port-forward service/delivery-service 8080:80
```

Accès à l'application via l*ocalhost:8080*

Mise à jour du déploiement
```
kubectl set image deployment/delivery-service delivery-
service=dthibau/delivery-service:0.0.3-SNAPSHOT
```

Statut du roll-out
```
kubectl rollout status deployment/delivery-service
```

Accès à l'application : *http:<IP>/actuator/info*

#Visualiser les déploiements
```
kubectl rollout history deployment/delivery-service
```

#Effectuer un roll-back
```
kubectl rollout undo deployment/delivery-service
```
#Scaling
```
kubectl scale deployment/delivery-service --replicas=5
```

11.1 : Déploiements de la stack

Tag : Atelier11

Installation de Kafka via Helm

Installation de Helm

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install my-release bitnami/kafka
```

Exposition du service kafka sur l'adresse kafka:9092

```
kubectl apply -f k8s/kafka-micro.yml
```

Déploiement de la stack

Les images du TP précédent ont été poussées vers DockerHub
Visualiser le profil ***kubernetes*** de ***Gateway.yml***
Visualiser le répertoire ***k8s*** et les différents manifestes Kubernetes
Utiliser les scripts ***deploy.sh*** et ***undeploy.sh***

Effectuer un déploiement

Exposer la Gateway :

```
kubectl port-forward service/gateway 8080:8080
```

Accéder aux URLs de la gateway, par exemple :

- *http://localhost:8080/actuator/gateway/routes*
- *http://localhost:8080/order/actuator*

Utiliser le script JMeter fourni pour solliciter la stack

11.2 : Istio

Installation Istio :
Voir https://istio.io/docs/setup/getting-started/#download

Dans un premier temps désactiver Istio

```
kubectl edit namespace default
```

Mettre au point un script /deployment.sh pour déployer les micro-services sans la gateway

Vérifier le bon déploiement et le nombre de pods dans un contexte sans-istio

Activer istio dans le namespace par défaut

```
kubectl label namespace default istio-injection=enabled
```

Déployer la stack et regarder les pods

Définir une gateway istio permettant le routage vers les différents micro-services :

- *LivraisonService*
- *OrderService*

- *OrderQueryService*
- *ProductService*

```
kubectl apply -f gateway.yaml
```

Vérifier le tout avec :

```
istioctl analyze
```

Accéder à des micro-services via la gateway :

```
export INGRESS_PORT=$(kubectl -n istio-system get service istio-
ingressgateway -o jsonpath='{.spec.ports[?
(@.name=="http2")].nodePort}')
export SECURE_INGRESS_PORT=$(kubectl -n istio-system get service
istio-ingressgateway -o jsonpath='{.spec.ports[?
(@.name=="https")].nodePort}')
export INGRESS_HOST=$(minikube ip)
```

Dans un autre terminal :

```
minikube tunnel
```

Puis dans un navigateur :

```
http://$INGRESS_HOST:$INGRESS_PORT/order-service
```

11.3 Dasboard Istio/Kiali

Installer les add-ons istio : dans le répertoire d'istio :

```
kubectl apply -f samples/addons
```

Accès au tableau de bord kiali

```
istioctl dashboard kiali
```

Se logger avec admin/admin

Vérifier les connexions entre micro-services et en particulier avec zipkin.

Solliciter via un script JMeter