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

Outils utilisés :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux, MacOs, Windows 10
- JDK17+
- Spring Tools Suite 4.x avec Lombok
- Git
- Docker
- JMeter

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

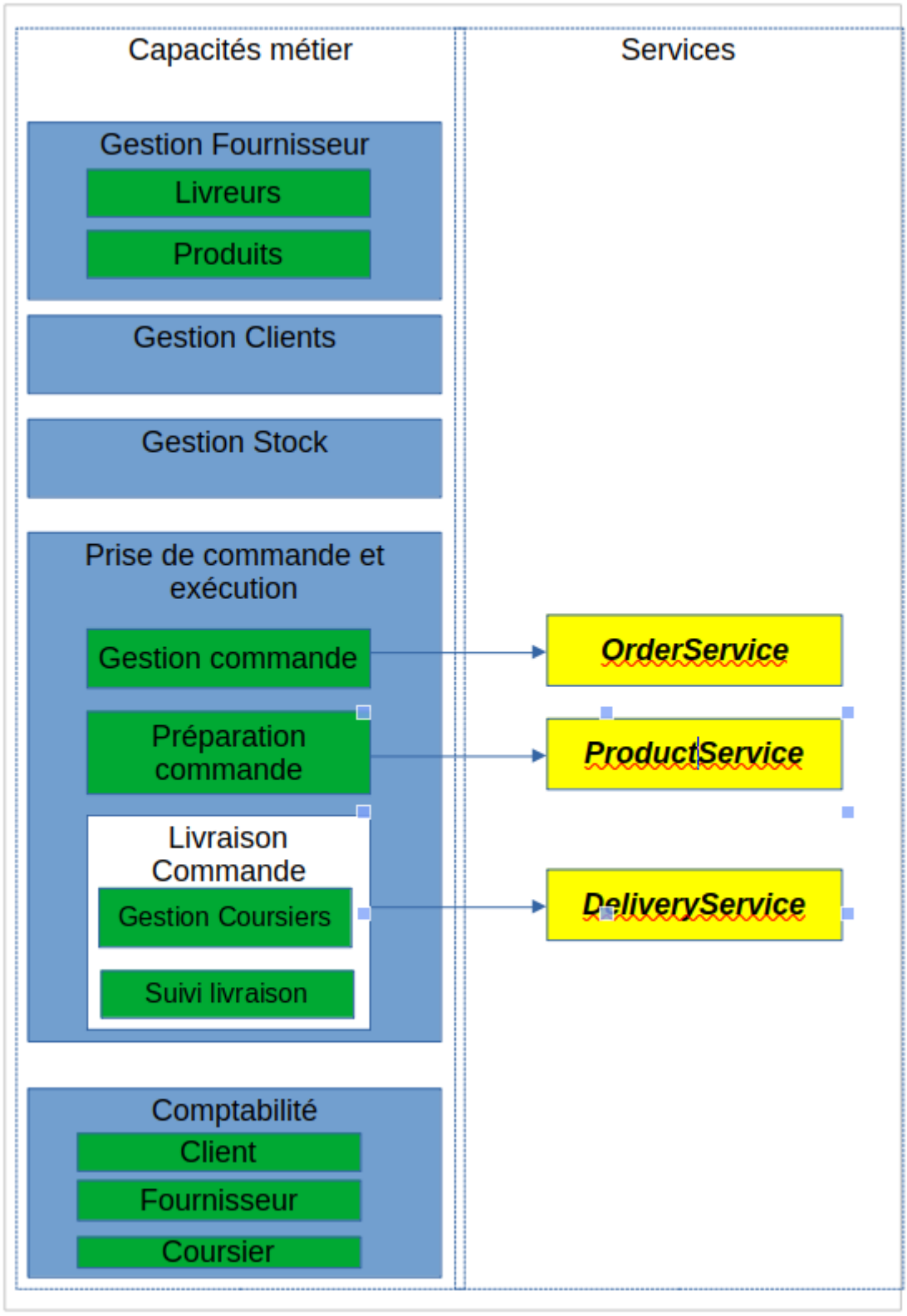
Table des matières

Atelier 1: Décomposition

Objectifs : Illustrer le pattern « *Decompose by business capability »*

Notre système représente un magasin permettant de commander des produits en ligne.
Les produits sont ensuite déstockés et livrés au domicile du client.

Les capacités métier de notre magasin



Nous nous intéressons principalement à la prise et l'exécution de la commande qui a donc identifié 3 micro-services

- ***OrderService :***
  Création, Mise à jour, historique de commande
- ***ProductService :***
  Récupération des produits, empaquetage
- ***DeliveryService :***

Disponibilité des coursiers, suivi de livraison

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

Les opérations identifiées et leurs collaborateurs pour ce UserStory sont résumés dans le tableau suivant :

Service	**Opération**	**Collaborateur**
OrderService	*createOrder()* *findOrderDetail()*	***ProductService*** *createTicket()* ***AccountingService*** *executePayment()*
ProductService	*createTicket()* *noteTicketReadyToPickpup()*	***DeliveryService*** *createDelivery()*
DeliveryService	*noteDeliveryPickUp()* *updatePosition()* *noteDeliveryDelivered()* *findAvailableCourier()* *findPendingDeliveries()*	

Commandes

OrderService

- *createOrder() :*
  - Entrée *: consumerId, orderLineItems, paymentInfo, deliveryInfo*
  - Retour *: orderId*
  - Post-conditions *:* Paiement effectué

ProductService

- *createTicket()*
  - Entrée : *OrderId, orderLineItems*
  - Retour : *TicketId*
  - Pré-conditions : Les produits demandés sont disponibles en stock
- *noteTicketReadyToPickup()*
  - Entrée : *ticketId*
  - Retour : *Void*
  - Post-conditions : La livraison correspondante est créée

DeliveryService

- *createDelivery()*
  - Entrée : *ticketId, orderId*
  - Retour : *deliveryId*
  - Post-conditions: Création d’une livraison dans l’état *TOPICK_UP*
- *noteDeliveryPickUp()*

    - Entrée *: deliveryId, courierId*
    - Retour *: Void*
    - Post-conditions *:* Livraison dans l'état *IN_PROGRESS*, coursier affecté
- *updatePosition()*
    - Entrée *: courierId*
    - Retour *: Void*
    - Post-conditions *:* Mise à jour position coursier
- *noteDeliveryDelivered()*
    - Entrée *: deliveryId,*
    - Retour *: Void*
    - Post-conditions *:* Livraison dans l'état *DELIVERED*, Coursier disponible

Requêtes

OrderService

- *findOrderDetail(orderId) :* Toutes les informations d'une commande

ProductService

- *findTicket(orderId)* : Le ticket associé à une commande

DeliveryService

- *findAvailableCouriers()* : Les coursiers disponibles
- *findDeliveries(deliveryStatus)* : Les livraisons en fonction de leurs statuts
- *findDelivery(orderId)* => La livraison associée à la commande

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: Interaction synchrone, Discovery et Circuit Breaker

Objectifs : Comprendre les pré-requis à une interaction synchrone dans un contexte de réplication de services et de résilience

2.1 Services de discovery et de configuration centralisée

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 ***ProductService*** 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
```

Cette méthode peut également être appliqué à tous les projets qui peuvent être démarrés sans l'IDE.

2.2 Interaction 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 ***TPS/2.2/CreateOrder.jmx*** fourni pour visualiser la répartition de charge

Arrêter une réplique pendant le tir et observer le comportement de ***OrderService***

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 votre code avec le tag Atelier2.3

Atelier 3: Interaction asynchrone

Objectifs : Comprendre les pré-requis aux interactions asynchrones, le découplage producteur / consommateur et le rôle du *Transactional Outbox Pattern*

Description

Nous mettons en place une communication de type Publish/Subscribe

ProductService publie l'événement TICKET_READY vers le topic *tickets*

DeliveryService réagit en créant une livraison

3.1 Démarrage du Message Broker

Récupérer le fichier ***TPS/3.1/docker-compose.yml*** permettant de démarrer le message broker Kafka et d'une console d'administration accessible à *http://localhost:9090*

docker-compose up -d

Déclarer kafka dans le fichier */etc/hosts*

```
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 ***org.formation.service.TicketService*** 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*

L'annotation *@Transactional* sur les classes services englobe les méthodes dans une transaction BD. Cependant si une erreur survient lors de l'envoi du message la transaction n'est pas rollbackée.

Pour générer un message, commencer par créer un ticket via swagger

```
POST /api/tickets/1
[
  {
    "reference": "REF",
    "quantity": 2
  }
```

```
]
```

Puis provoquer l’envoi du message via

```
POST /api/tickets/1/pickup
```

Visualisez le topic créé et le message envoyé via la console d’administration Kafka

3.3 Mise en place du consommateur

Reprendre le tag Atelier 3.3

Les mêmes dépendances ont été ajoutées sur *DeliveryService*

Visualiser la classe ***org.formation.service.DeliveryService*** qui écoute les messages du topic ***tickets***

Vous pouvez tester le tout avec le script JMeter fourni

Lors de l’exécution du test, vous pouvez arrêter *DeliveryService* puis le redémarrer tous les messages sont consommés

3.4 Messagerie transactionnelle

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

Comparer avec le tag Atelier3.4

Atelier 4: Saga Pattern

Objectifs : Implémenter le pattern Saga via un orchestrateur

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. ***TicketCommande/TICKET_CREATE*** , sur le channel ***TICKET_REQUEST_CHANNEL***
Il attend une réponse sur le channel ***ORDER_SAGA_CHANNEL***

2. A la réception d'une réponse à TICKET_CREATE:
    - Si OK : Envoi d'une commande de paiement sur le channel ***PAYMENT_REQUEST_CHANNEL***
    - Si NOK : Transaction locale compensatoire : la commande est annulée

3. A la réception d'une réponse à PAYMENT_AUTHORIZE:
    - Si OK :
        - Envoi d'une commande TICKET_APPROVE via le channel ***TICKET_REQUEST_CHANNEL***
        - Transaction locale pour valider la commande
    - Si NOK :
        - Envoi d'une commande TICKET_REJECT via le channel ***TICKET_REQUEST_CHANNEL***
        - Transaction locale compensatoire : la commande est annulée

Les services impliqués ProductService et PaymentService 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 ***TPS/2.2/CreateOrder.jmx***

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

Reprendre la branche *sagaRest*

Visualiser le code de ***CreateOrderSaga*** qui implémente le pattern via des interactions RestFul.

Vous pouvez également tester cette implémentation avec le même script JMeter

Atelier 5: Logique métier

5.1 Domain Model et Domain Event Pattern

L'objetif est d'appliquer ces 2 patterns au service *ProductService* (qui pour l'instant implémente un *TransactionScriptPattern* ou la classe *org.formation.service.TicketService* contient la logique métier)**.**

Définir une classe ***ResultDomain*** encapsulant :

- Un agrégat ***Ticket***
- Un événement : ***TicketStatusEvent***

La logique métier relatif au Ticket est implémentée dans la classe Ticket. Celle ceci expose :

- `public static ResultDomain createTicket(Long orderId, List<ProductRequest> productRequest) throws MaxWeightExceededException`
- `public ResultDomain approveTicket()`
- `public ResultDomain rejectTicket()`
- `public ResultDomain readyToPickUp()`

Le code de classe service est donc de déléguer la logique métier à la classe Ticket, récupérer le *ResultDomain* résultant et systématiquement :

- stocker l'objet du domaine en base
- publier l'événement sur la channel *ticket-event* (Transactional Outbox Pattern)

Récupérer le tag Atelier5

Visualiser le code

Vous pouvez tester soit en utilisant le fichier JMeter ***TPS/2.2/CreateOrder.jmx***

Atelier 6: Service de Query

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

Vous pouvez utilisez le script JMeter ***TPS/6/CreateAndDeliverOrder.jmx*** 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 Métriques 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/prometheus`

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
```

11.2 : 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

11.3 : Maillage de service, 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
```

11.4 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