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

Outils utilisés :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux, MacOs, Windows 10
- JDK17+
- IDE de votre choix IntelliJ, Eclipse, VSCode
- Git
- Docker
- JMeter

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

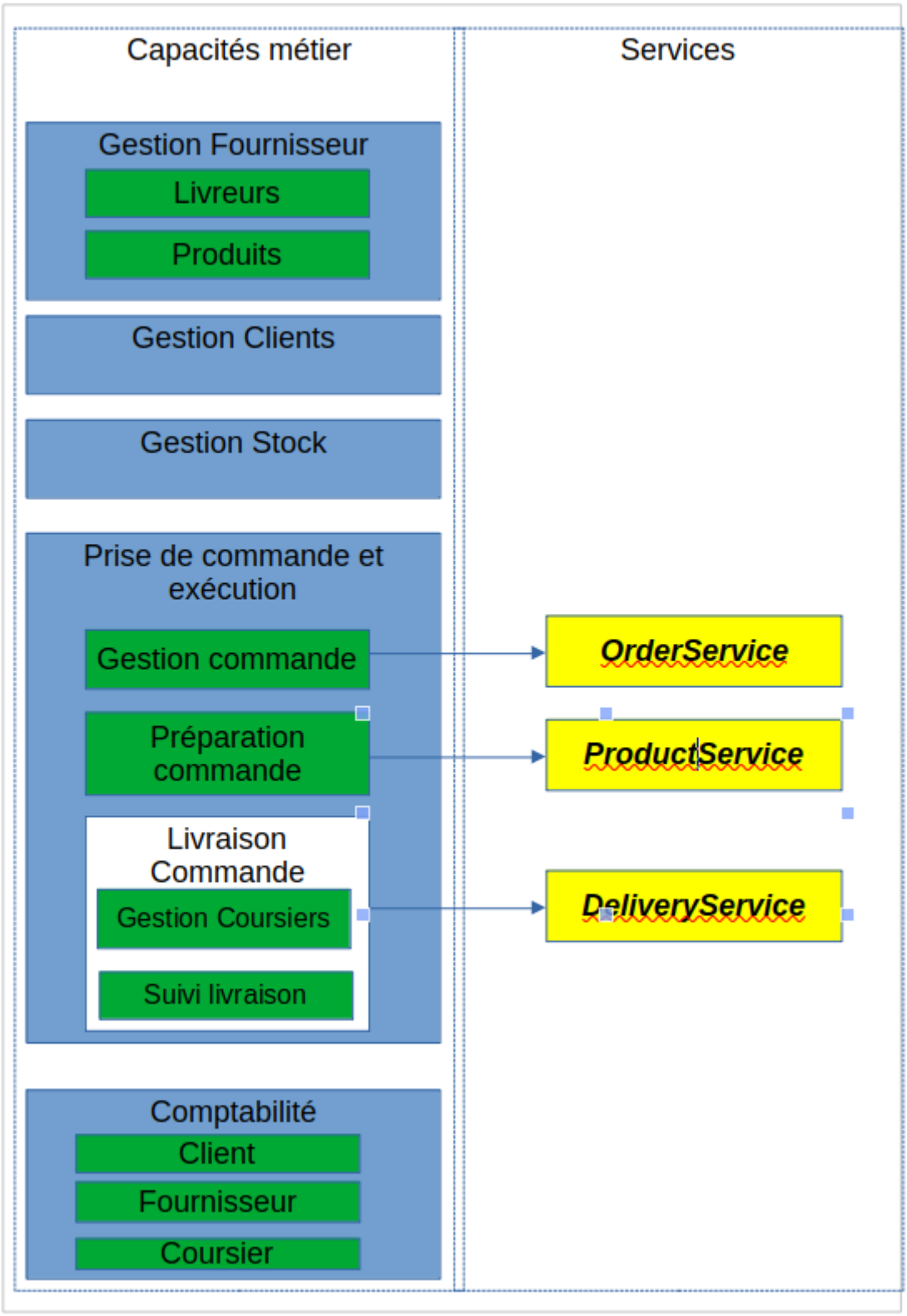
Table des matières

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()*	

<u>Commandes</u>

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 : Configuration centralisée

2.1 Mise en place de Config Server

Créer un nouveau projet Spring nommé ***config***:

- Maven / Java 21
- Group : org.formation
- Artifcat : config

Ajouter le starter :

- spring-cloud-starter-config-server

Annoter la classe principale avec ***@EnableConfigServer***

Configurer le service afin qu'il écoute sur le porte 8888.

Tester le démarrage

Configurer *config* afin que les configurations soient gérées par le FileSystem dans le dossier *classpath:/shared*

```yaml
spring:
  cloud:
    config:
      server:
        native:
          search-locations: classpath:/shared
  profiles:
    active: native
```

Créer ensuite le répertoire ***src/main/resources/shared***

2.2 Configuration des clients

Sur les 3 micro-services ajouter les straters :

- ***spring-cloud-starter-bootstrap***
- ***spring-cloud-starter-config-client***

Créer un fichier ***src/main/resources/bootstrap.yml***

Définir les propriétés suivantes :

```yaml
spring:
  application:
    name: <nom_du_service>

  cloud:
    config:
      uri:
      - http://localhost:8888
      fail-fast: true
```

La propriété ***spring.application.name*** ne doit pas contenir de caractère spéciaux. Utiliser *product,order* et *delivery*

Sur le serveur de config, créer le fichier ***src/main/resources/shared/application.yml*** et positionner les valeurs de configuration communes à tous les micro-services.
Par exemple, la configuration des endpoints de Actuator

```yaml
management:
  endpoints:
    web:
      exposure:
        include:
        - env
        - beans
        - health
- mappings
```

Créer ensuite un fichier par service ***src/main/resources/shared/<nom_du_service>.yml*** et indiquer les propriétés spécifiques à chaque service.

Démarrer le serveur de configuration et vérifier les URLS suivantes :

- http://localhost:8888/application/default
- http://localhost:888/product/default

Vérifier le démarrage des 3 services en particulier la trace :
`Fetching config from server at :` http://localhost:8888

Atelier 3: RPC, Service Discovery et Circuit Breaker

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

3.1 Mise en place du services de discovery Eureka

Créer un projet eureka avec les starters suivants :

- ***config client***
- ***bootstrap***
- ***eureka server***

Annoter la classe principale avec ***@EnableEurekaServer***

Définir un fichier bootstrap comme précédemment :

```
spring:
  application:
    name: eureka

  cloud:
    config:
      uri:
      - http://config:8888

      fail-fast: true
```

Configurer eureka en mode standalone :

Dans les propriétés communes des micro-services ***shared/application.yml***

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/
    healthcheck:
      enabled: true
    preferSameZoneEureka: true
  instance:
    metadataMap:

      zone: zone1
```

Dans les propriétés spécifiques au service eureka ***shared/eureka.yml***

```
server:
  port: 1111


eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

Vérifier le démarrage du serveur eureka :

- http://localhost:1111

Dans les services applicatifs ajouter la dépendance :

- `spring-cloud-starter-netflix-eureka-client`

Démarrer tous les services et vérifier leur inscription dans l'annuaire.

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.

3.2 Interaction synchrone REST

Dans le projet ***order-service***, ajouter une classe de configuration contenant :

```
@Configuration
public class DependenciesConfiguration {

	@Autowired
	RestTemplateBuilder builder;

	@Bean
	@LoadBalanced
	RestTemplate productRestTemplate() {
		return builder.rootUri("http://product").build();
	}

}
```

Ensuite, dans la classe ***org.formation.service.OrderService***, créer une méthode permettant de créer une commande dans la base puis d'effectuer une requête REST vers ***product-service*** sur le endpoint suivant :

POST /api/tickets/{orderId}

Tester l'implémentation via l'interface Swagger de ***order-service*** :

```
POST /api/orders
{
  "restaurantId": 1,
  "consumerId": 1,
  "lineItems": [
   {
    "refProduct": "REF",
    "price": 1000,
```

```
      "quantity": 1
    }
  ],
  "deliveryAddress": {
    "rue": "RUE",
    "ville": "VILLE",
    "codePostal": "75000"
  },
  "paymentInformation": {
    "paymentToken" : "AAAAAA"
  }
}
```

3.3 Load-balancing

Dans la configuration de ***product-service***, configurer un profil ***replica*** faisant écouter le service sur un autre port

Modifier *TicketController* afin qu'il affiche le n° de port dans le message de log de création de Ticket

Démarrer 2 instances de ***product-service***

Dans le projet ***order-service***, vérifier l'annotation ***@LoadBalanced*** sur la création du bean RestTemplate

Utiliser le script JMeter ***TPS/3.1/CreateOrder.jmx*** fourni pour visualiser la répartition de charge

Arrêter une réplique pendant le tir et observer le comportement de ***order-service***

3.4 Circuit Breaker Pattern

Dans le projet ***order-service***, ajouter la dépendance

- `spring-cloud-starter-circuitbreaker-resilience4j`

Encapsuler le code de l'appel du service *product-service* dans un circuit breaker avec une méthode de fallback

Réexécuter le script JMeter et observer le comportement lors d'arrêt de product-service

Atelier 4: Messaging

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

product-service publie l'événement TICKET_READY vers le topic *tickets*

delivery-service réagit en créant une livraison

4.1 Démarrage du Message Broker

Récupérer le fichier ***TPS/4.1/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

Configuration Kafka, propriétés communes à tous les micro-services :

```
spring:
  kafka:
    producer:
      key-serializer: org.apache.kafka.common.serialization.LongSerializer
      value-serializer:
org.springframework.kafka.support.serializer.JsonSerializer
    consumer:
       key-deserializer: org.apache.kafka.common.serialization.LongDeserializer
       value-deserializer:
org.springframework.kafka.support.serializer.JsonDeserializer
       properties:
        spring.json.trusted.packages: '*'
```

4.2 Mise en place du producer

Ajouter la dépendance suivante dans product-service :

- ***spring-kafka***

Créer une classe DTO qui encapsule le message org.formation.service.TicketEvent :

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class TicketEvent {

      private TicketStatus oldStatus;
      private TicketStatus newStatus;

      private Ticket ticket;


}
```

Créer une classe ***org.formation.service.EventService*** qui publie un TicketEvent sur le topics tickerts. Ce bean service contiendra une méthode :
public void publish(TicketEvent ticketEvent) {

Créer une classe ***org.formation.service.TicketService*** comportant 2 méthodes

- **`public Ticket createTicket(Long orderId, List<ProductRequest> productsRequest)`**
  Crée un ticket avec le statut *TicketStatus.CREATED, publier l'évènement*
- **`public Ticket readyToPickUp(Long ticketId)`**
  Modifie le statut d'un 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 *TicketEvent*

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 des messages, commencer par créer un ticket via swagger

```
POST /api/tickets/12
[
  {
    "reference": "REF",
    "quantity": 2
  }
]
```

Puis provoquer l'envoi du message via

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

Visualisez le topic créé et les message envoyés via la console d'administration Kafka

4.3 Mise en place du consommateur

Sur le projet ***delivery-service***, *ajouter le starter spring-kafka*

Lors de la réception d'un message, le projet doit créer un objet Livraison dans sa `base`.

Implémenter les classes services et DTO nécessaires.

Vous pouvez tester le tout avec le script JMeter fourni ***TPS/4.3/ReadyToPickUp.jmx***

Lors de l'exécution du test, vous pouvez arrêter ***delivery-service*** puis le redémarrer tous les messages sont consommés.

Arrêter également le service Kafka pendant le test et observer les conséquences.

4.4 Messagerie transactionnelle

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

Au lieu de directement publier le message le stocker dans une table.

Écrire un service schédulé qui consulte ces tables et essaient de publier vers Kafka. (Vous devez ajouter l'annotation ***@EnableScheduling*** sur la classe principale et ***@Scheduled*** sur la méthode à scheduler)

Refaire ensuite le test d'arrêt du service Kafka . Assurer vous qu'aucun message n'est perdu

Atelier 5: 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*** dans le projet ***order-service***.

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 ***product-service*** et ***payment-service*** implémentent les réponses aux requêtes de l'orchestrateur.

Le projet ***payment-service*** est fourni. Une fausse logique a été implémentée qui refuse la demande de paiement si le *paymentToken* ne commence pas par le caractère « A »

Vous pouvez configure les channels suivant dans la configuration générale des micro-services :

```
channels:
  ticket-command: 'tickets-command'
  payment-command: 'payments-command'
  order-response: 'order-response'
```

Les données échangées entre les services pourront être :

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class TicketCommand {

	Long orderId;

	String commande;

	List<ProductRequest> productRequest;

}

@Data
@AllArgsConstructor
public class PaymentCommand {

		private long orderId;

		private Float amount;

		PaymentInformation paymentInformation;

}

@Data
public class CommandResponse {

	private long orderId;

	// 0 Ok et -1 KO
	private int status;

	private String command;

	public boolean isOk() {
		return status == 0;
	}
}
```

Vous pouvez tester soit via l'interface Swagger de *order-service*, soit en utilisant le fichier JMeter ***TPS/3.3/CreateOrder.jmx***

Atelier 6: Domain Model et Domain Event Pattern

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

Définir une classe ResultDomain encapsulant :

- L’agrégat *Ticket*
- L’événement : *TicketEvent*

Réorganiser le code afin que la logique métier soit encapsulée 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 rôle de la lasse 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)

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

Atelier 7: Service de Query

Reprendre les projets ***config, payment-service, delivery-service, order-service et product-service***

Vous pouvez utilisez le script JMeter ***TPS/7/CreateAndDeliverOrder.jmx*** fourni pour initialiser les bases avec des données (Order, Ticket, Livraison).

Après l'exécution du script, visualiser les topics Kafka

7.1 API Composition

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

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

Utiliser les starters :

- reactive web
- resilience 4j
- config client
- Eureka Client
- Starters de développement

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

L'objet DTO retourné sera :

```
@Data
public class OrderDto {

	long orderId;
	Instant date;
	Address address;
	String nomLivreur;
	String telephoneLivreur;


}
```

7.2 CQRS View Pattern

Ajouter les starters :

- PostgreSQL
- R2DBC

- Kafka

Démarrer postgres et pgAdmin via docker

```
cd TP/7.2
docker-compose -f postgres-docker-compose.yml up -d
```

Créer un base postgres gateway et la déclarer dans config/gateway.yml

```yaml
spring:
  r2dbc:
    url: 'r2dbc:postgresql://localhost:5432/gateway'
    username: postgres
    password: postgres
```

Dans le projet gateway, 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 8: API Gateway

Dans le projet gateway, ajouter le starter :

- ***spring-cloud-starter-gateway***

Définir via la configuration les routes suivantes :

- Le endpoint permettant de créer une Commande
- Les endpoint pour la gestion des tickets
- Les endpoints de ***delivery-service***

Atelier 9 : Sécurité

Démarrer un serveur KeyCloak en important le realm store:

cd TPS/9

```
docker run -p 8180:8080 \
  -e KEYCLOAK_ADMIN=admin \
  -e KEYCLOAK_ADMIN_PASSWORD=admin \
  -v .:/opt/keycloak/data/import \
  quay.io/keycloak/keycloak start-dev --import-realm
```

Vérifier que vous pouvez obtenir un jeton via le grant type password pour le client frontend

```
curl -XPOST http://localhost:8180/realms/store/protocol/openid-connect/token -d grant_type=password -d client_id=frontend -d username=alice -d password=secret
```

9.1 Sécurisation de la gateway

Ajouter les starters :

- ***SpringSecurity***
- ***spring-boot-starter-oauth2-resource-server***

Configurer la sécurité comme suit :

```
@Bean
public SecurityWebFilterChain securitygWebFilterChain(ServerHttpSecurity http) {

      return http.authorizeExchange(acl -> acl.anyExchange().authenticated())
                  .oauth2ResourceServer((v -> v.jwt(Customizer.withDefaults())))
                  .csrf(csrf -> csrf.disable())
                  .build();
}
```

Tester avec le script JMeter fourni

Extraction des rôles dans le claims group

Implémenter un Converter JWT de la façon suivante :

```
@Bean
public ReactiveJwtAuthenticationConverter jwtAuthenticationConverter() {
      JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter = new JwtGrantedAuthoritiesConverter();
    grantedAuthoritiesConverter.setAuthoritiesClaimName("groups");
    grantedAuthoritiesConverter.setAuthorityPrefix("ROLE_");

    ReactiveJwtAuthenticationConverter jwtAuthenticationConverter = new ReactiveJwtAuthenticationConverter();
    jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(new ReactiveJwtGrantedAuthoritiesConverterAdapter(grantedAuthoritiesConverter));
    return jwtAuthenticationConverter;
}
```

Changer les ACLs de la gateway afin qu'elle autorise les accès aux seuls rôles CLIENT et MANAGER

9.2 ACLs sur services métier et propagation de jeton

Sur le projet ***product-service***, ajouter les starters :

- ***SpringSecurity***
- ***spring-boot-starter-oauth2-resource-server***

Protéger les endpoints afin qu'il ne soit accessible qu'avec le profil *MANAGER*

Vérifier que la gateway transfert les entêtes http et en particulier l'entête ***Authorization***

9.3 Remplacement de jeton

9.3.1 Appel micro-service interne

Récupérer le projet ***notification-service*** et utiliser son API pour envoyer un mail lors de la création d'une commande.

9.3.2 Obtention d'un jeton et ajout du jeton dans un RestTemplate

Vérifier que vous pouvez obtenir un jeton via le grant type client_credentials pour le client order

```
curl -XPOST http://localhost:8180/realms/store/protocol/openid-connect/token -d grant_type=client_credentials -d client_id=order -d client_secret=secret
```

Dans le projet ***order-service***, ajouter les starters :

- ***SpringSecurity***
- ***spring-boot-starter-oauth2-client***
- ***spring-boot-starter-oauth2-resource-server***

Configurer la sécurité comme suit :

```
@Bean
public SecurityFilterChain securityFilterChain(     HttpSecurity http) throws Exception {

return http.authorizeHttpRequests(acl -> acl.anyRequest().hasAnyRole("CLIENT"))
        .oauth2ResourceServer(r -> r.jwt(jwt ->
jwt.jwtAuthenticationConverter(jwtAuthenticationConverter())))
    .csrf(csrf -> csrf.disable())
    .build();

}

@Bean
```

```
public JwtAuthenticationConverter jwtAuthenticationConverter() {
JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter = new JwtGrantedAuthoritiesConverter();
    grantedAuthoritiesConverter.setAuthoritiesClaimName("groups");
    grantedAuthoritiesConverter.setAuthorityPrefix("ROLE_");

    JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();

jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(grantedAuthoritiesConverter);
    return jwtAuthenticationConverter;
}

@Bean
public AuthorizedClientServiceOAuth2AuthorizedClientManager authorizedClientManager(
        ClientRegistrationRepository clientRegistrationRepository,
        OAuth2AuthorizedClientRepository authorizedClientRepository,
        OAuth2AuthorizedClientService authorizedClientService) {

OAuth2AuthorizedClientProvider authorizedClientProvider =
              OAuth2AuthorizedClientProviderBuilder.builder()
                            .refreshToken()
                            .clientCredentials()
                            .build();

AuthorizedClientServiceOAuth2AuthorizedClientManager authorizedClientManager =
             new AuthorizedClientServiceOAuth2AuthorizedClientManager(
                          clientRegistrationRepository, authorizedClientService);
        authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);

return authorizedClientManager;
}
```

Et dans les propriétés de configuration :

```
spring :
  security:
    oauth2:
      client:
        provider:
          keycloak:
            token-uri: http://localhost:8180/realms/store/protocol/openid-
connect/token
        registration:
          store:
            provider: keycloak
            client-id: order
            client-secret: secret
            client-authentication-method: client_secret_basic
            authorization-grant-type: client_credentials
            scope:
            - openid
            - service
```

Définir un service responsable de l'obtention des jetons et gérant le rafraîchissement :

```
@Service
public class TokenService {

@Autowired
private AuthorizedClientServiceOAuth2AuthorizedClientManager authorizedClientServiceManager;

OAuth2AuthorizedClient oAuth2AuthorizedClient;

public OAuth2AccessToken getToken() {
        if (oAuth2AuthorizedClient == null) {
                oAuth2AuthorizedClient = authorizedClientServiceManager.authorize(initialRequest());
        } else if
(oAuth2AuthorizedClient.getAccessToken().getExpiresAt().minusSeconds(60).isBefore(Instant.now())) {
                        oAuth2AuthorizedClient =
authorizedClientServiceManager.authorize(initialRequest());
        }
        return oAuth2AuthorizedClient.getAccessToken();
```

```
        }

OAuth2AuthorizeRequest initialRequest() {
        return OAuth2AuthorizeRequest.withClientRegistrationId("store").principal("Order
Service").build();
}


}
```

Définir un intercepteur dans le bean RestTemplate utilisé pour communiquer avec le micro-service interne notification-service

```
@Bean
@LoadBalanced
RestTemplate notificationRestTemplate(OAuth2AuthorizedClient oAuth2AuthorizedClient) {


        RestTemplate rest = builder.rootUri("http://notification").build();
        rest.getInterceptors().add((request, body, execution) -> {
                OAuth2AccessToken accessToken  = tokenService.getToken();
                request.getHeaders().setBearerAuth(accessToken.getTokenValue());
                return execution.execute(request, body);
        });

        return rest;
}
```

Tester via le script JMeter pour l'utilisateur bob

9.3.2 Protection notification-service via le scope services

Dans Keycloak ajouter un client-scope ***service,*** l'affecter par défaut à tous les clients ***« backend »***

Sur ***notification-service***, ajouter les starters :

- ***SpringSecurity***
- ***spring-boot-starter-oauth2-resource-server***

Configurer la sécurité afin que les enpoints soient accessible avec le scope *service*

Tester via le script JMeter pour l'utilisateur bob

Atelier 10 : Observabilité

10.1 Health API et actuator

Configurer les micro-services afin qu'ils affichent le détail de leur état santé ainsi que les liveness probes et readiness probes :

```yaml
management:
  endpoint:
    health:
      show-details: always
      probes:
        enabled: true
```

Accéder à *http://<service>/actuator/health*

10.2 Métriques micrometer

10.2.1 Métriques automatiques

Ajouter les dépendances suivantes aux projets Gateway, Order et notification :

- ***io.micrometer :micrometer-registry-prometheus***

Vérifier celle d'actuator et autoriser les URLs d'actuator dans la sécurité

Démarrer les micro-services et le solliciter par le script JMeter

Visualiser les métriques à l'URL
`http://<server>/actuator/prometheus`

Dans order-service, encapsuler l'envoi de notification dans un circuit-breaker

Récupérer le répertoire grafana fourni et y exécuter :
`docker-compose up -d`
Cela doit démarrer un serveur Prometheus et Grafana

Se connecter sur Prometheus :
`http://localhost:9090`
Et vérifier que les cibles sont bien atteintes : *Status → Target*

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.

10.2.2 Métrique custom

Nous voulons exporter 2 métriques personnalisés :

- Le nombre de commandes approuvées

- Le nombre de commandes rejetées

Dans la classe ***CreateOrderSaga***, s'injecter un *MeterRegistry* et construire 2 compteurs durant la construction :

```
this.orderApproved = Counter.builder("order.approved").
         tag("type", "métier").
         description("Nombre de commande approuvée").
         register(meterRegistry);
this.orderRejected = Counter.builder("order.rejected").
          tag("type", "métier").
          description("Nombre de commande rejetée").
          register(meterRegistry);
```

Incrémenter ses compteurs lors de l'approbation d'une commande ou lors de son rejet.

Solliciter l'architecture via le script JMeter et visualiser les compteurs dans l'URL *http://localhost:8082/actuator/prometheus*

10.3 Tracing avec Brave et Zipkin

Démarrer un serveur Zipkin

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

Accéder à

```
http://localhost:9411/zipkin/
```

Pour tous les services, ajouter les dépendances :

- *io.micrometer :micrometer-tracing-bridge-brave*
- *io.zipkin.reporter2:zipkin-reporter-brave*

Ajouter dans la configuration centralisée :

```
management:
  tracing:
    enabled: true
    sampling.probability: 1.0
```

Les redémarrer et solliciter l'architecture avec le script JMeter fourni

Visualiser ensuite les traces dans zipkin