Cahier de TP
« Architecture micro-services avec Spring Cloud»

Pré-requis :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux, MacOs, Windows 10
- JDK17+
- IDE Recommandés : STS 4, IntelliJIDEA, VSCode
- JMeter 5.x
- Docker / Podman

Solutions des ateliers :
https://github.com/dthibau/springcloud-solutions

Table des matières

Atelier 1: Serveur Configuration et service de Discovery....3
1.1. Mise en place d'un serveur de Discovery Eureka....3
1.2. Enregistrement d'un 1er micro-service....3
1.3. Mise en place d'un serveur de configuration....3
1.4. Changement dynamique du niveau de trace....4
1.5. Configuration Eureka en paire (Optionnel)....4
Atelier 2: Supports pour les clients REST....5
2.0. Mise en place....5
2.1. Répartition de charge....5
2.2. Disjoncteur avec Resilience4J....6
2.3. Mise en place d'un client Feign....6
2.4. Spring Cloud Gateway....7
2.5. Consumer Driven Contract avec Spring Cloud Contract....7
Atelier 3 : Monitoring....9
3.1 Métriques CircuitBreaker....9
3.2 Tracing avec Sleuth et Zipkin....9
3.3 Centralisation des traces....9
4.1 Saga Pattern....11
4.2 Streaming temps réel avec Spring Boot Stream et Kafka....11
4.2.1 Stack Kafka + ELK....11
4.2.2 Importation 1er micro-service....12
4.2.3 Mise en place second micro-service....12
4.2.4 Remontée des informations dans ElasticStack....12
4.3 SpringCloud DataFlow....13
4.3.1 Installation....13
4.3.2 Création du stream....13
4.3.3 Déploiement....13
4.3.4 Vérification....13
Atelier 5 : Spring Cloud avec oAuth2....14
5.1. Mise en place du serveur d'autorisation (KeyCloak)....14
5.2. OAuth2 login et sécurité stateful sur la Gateway....15
5.3. ACLS oAuth2 : Gateway as resource server....16

5.4. Relai de jeton.....16
Ateliers 6 : Déploiement via conteneurs.....18
6.1 Docker et docker-compose.....18
6.2 Kubernetes.....18

Atelier 1: Serveur Configuration et service de Discovery

1.1. Mise en place d'un serveur de Discovery Eureka

Créer une application SpringBoot ***annuaire*** avec une dépendance sur le starter Eureka-Server

Annoter la classe principale afin d'obtenir un serveur Eureka

Le faire écouter sur le port 1111

Faire une configuration « standalone » :

```
server.port: 1111

eureka:
  instance:
    hostname: localhost
  client:
    register-with-eureka: false
    fetch-registry: false
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

Démarrez l'application et accéder à la page d'accueil

1.2. Enregistrement d'un 1er micro-service

Récupérer le micro-service fourni *delivery-service*

Choisir le port d'écoute 3333

Démarrer le et vérifier son bon fonctionnement en accédant aux points d'accès : ***/swagger-ui.html*** et ***/actuator***

Annotez et configurez afin que le service s'enregistre sur le serveur Eureka

Visualisez la page d'accueil du service Eureka

1.3. Mise en place d'un serveur de configuration

Écrire une application SpringBoot démarrant un serveur de configuration écoutant sur le port 8888

Configurer le afin qu'il soit basé sur un système de fichiers

Vérifier l'URL : */application/default*

Placer la configuration des 2 micro-services (annuaire et *delivery-service*) dans le serveur de config

Configurer les micro-service afin qu'il charge leur configuration à partir du serveur de config

Démarrer la stack

Vérifier la bonne configuration en accédant :

- Sur le serveur de config
  - ***/annuaire/default***
  - ***/delivery-service/default*** et ***/delivery-service/prod***
- Sur annuaire et delivery-service
  - ***/actuator/env***

Astuce : Une fois la configuration

1.4. Changement dynamique du niveau de trace

Modifier la configuration pour le niveau de trace de `logging.level.root` et répercuter dynamiquement le changement sur le service ***delivery-service***

1.5. Configuration Eureka en paire (Optionnel)

Déclarer un profil dans le projet annuaire afin de démarrer 2 serveurs Eureka fonctionnant en paire

Atelier 2: Supports pour les clients REST

2.0. Mise en place

Ajouter dans le projet global les projets :

- ***notification-service :*** Micro-service de notification incluant un ficher docker-compose démarrant un fake SMTP serveur sur le port 2525
- ***order-service :*** Micro-service de gestion de Client et commandes

Récupérer la configuration des applications spring-boot et les placer dans le serveur de config

Démarrer les différents services, vérifier leur inscription dans l'annuaire Eureka. Accéder aux points d'accès */actuator*

Vous pouvez également démarrer le serveur Eureka en dehors de l'IDE, il ne sera pas modifié par la suite et cela permet de ne pas être gêné par les messages de sa console.

2.1. Répartition de charge

Dans ***order-service***, lors de la création d'une commande, nous voulons envoyer un mail de notification au client concerné via le service *notification-service*.

Nous voulons implémenter une répartition de charge si plusieurs instances de ***notification-service*** sont disponibles

Implémentation

Ajouter la dépendance sur *Spring Cloud LoadBalancer*

Utiliser un ***RestTemplate*** annoté par *@LoadBalanced* pour implémenter la répartition de charge.

Premier test

Vous pouvez tester via l'interface swagger de Order-service ***localhost:2222/swagger-ui.html***

POST /api/orders

```
{
  "date": "2021-12-13T15:24:14.226Z",
  "discount": 0,
  "client": {
    "id": 1
  },
  "orderItems": [
    {
      "refProduct": "string",
      "price": 100,
      "quantity": 1
    }
  ]
```

}

Répartition de charge

Ajouter un profil dans *notification-service* permettant de démarrer 2 services sur votre poste

Installer JMeter 5.x et charger le fichier *CreateOrder.jmx*

Lancer le test de charge

Observer la répartition de charge dans les différentes instances de *notification-service*

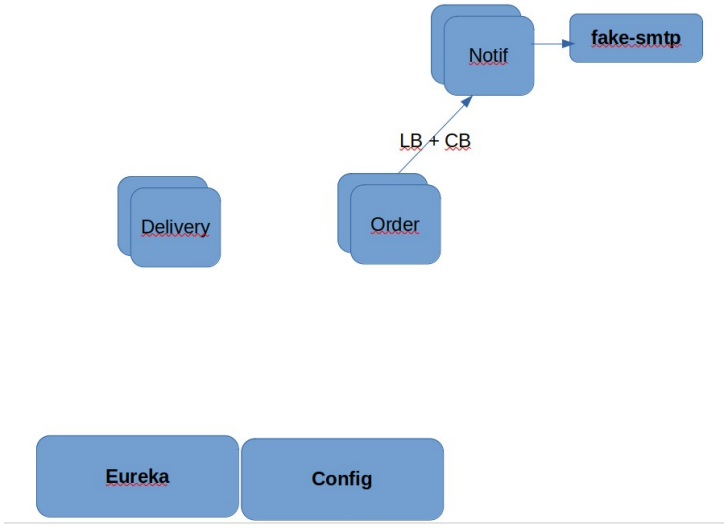
2.2. Disjoncteur avec Resilience4J

Pour cet atelier il est préférable de :

- Enlever le support pour docker-compose dans notification-service
  `spring.docker.compose.enabled = false`
- `Démarrer manuellement` ***`fake-smtp`***
  `docker run -p 2525:2525 dthibau/fake-smtp:1.0`

Dans ***order-service***, Ajouter une dépendance vers Resilience4J

Implémenter un pattern disjoncteur lors de l'appel à notification-service en utilisant l'usine ***CircuitBreakerFactory***



2.3. Mise en place d'un client Feign

Dans ***account-service***, ajouter une dépendance vers *OpenFeign*

Définir une interface permettant d'accéder à l'envoi de mail via ***notification-service***

Implémenter l'interface afin de fournir un fallback

Utiliser le script JMeter ***Register.jmx***

Arrêter les services de notification et observer le passage par la méthode de fallback

2.4. Spring Cloud Gateway

Créer un projet ***gateway*** avec les starters suivants :

- *Spring-cloud-gateway*
- *Discovery-client*
- *Config*
- *Actuator*

Configurer le port d'écoute à 8000

Configurer les routes vers les micro-services ***delivery-service,order-service*** et ***account-service*** en accord avec les URLs spécifiées dans le script JMeter

Configurer ***actuator*** et Vérifier le point d'accès */actuator/gateway/routes*

2.5. Consumer Driven Contract avec Spring Cloud Contract

Côté producteur

Ajouter le starter *Contract Verifer* au projet ***delivery-service***

Utiliser la classe de base de test fournie

Mettre en place dans `src/test/resources/contracts des spécifications` de l'API fournie par *LivraisonResource*
Vous pouvez vous inspirer des fichiers fournis

Mettre à jour le *pom.xml* pour inclure le plugin de SpringCloudContract :

```
<!-- Generate test classes for SC Contract -->
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>2.1.1.RELEASE</version>
  <extensions>true</extensions>
  <configuration>
    <baseClassForTests>
      org.formation.BaseTestClass
    </baseClassForTests>
  </configuration>
</plugin>
```

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

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

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

Côté Client

Dans le projet ***order-service*** ajouter le starter *Contract-Stub-Runner*

Modifier la méthode ***processOrder()*** de *order-service* afin qu'elle fasse un appel REST à *delivery-service* POST /api/livraison afin de créer une livraison associé à la commande

Récupérer la classe de test fournie et adapter la à votre environnement.

Exécuter le test et s'assurer qu'il passe

Atelier 3 : Monitoring

3.1 Métriques CircuitBreaker

Ajouter les dépendances suivantes au projet *order-service* :

- `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.

3.2 Tracing

Démarrer un serveur Zipkin

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

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

Déprécié en SB3

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
```

Spring Boot 3 Observability

Dépendances :

```
        <dependency>
            <groupId>io.micrometer</groupId>
            <artifactId>micrometer-observation</artifactId>
        </dependency>
        <dependency>
```

```
            <groupId>io.micrometer</groupId>
            <artifactId>micrometer-tracing-bridge-brave</artifactId>
        </dependency>
        <dependency>
            <groupId>io.zipkin.reporter2</groupId>
            <artifactId>zipkin-reporter-brave</artifactId>
        </dependency>
```

Configuration :

```
management:
  tracing:
    sampling:
      probability: 1.0
```

Effectuer cette configuration pour tous les micro-services métier et la gateway

Visualiser les traces dans Zipkin

3.3 Centralisation des traces

Récupérer l'archive ***elk.zip*** fournie et la décompresser dans son workspace

Visualiser la pipeline logstash

Mettre en place la configuration logback fourni dans les projets que vous voulez tracer. Cela nécessite la dépendance :

```
<dependency>
      <groupId>net.logstash.logback</groupId>
      <artifactId>logstash-logback-encoder</artifactId>
      <version>6.4</version>
</dependency>
```

Démarrer la stack ELK, puis démarrer la stack applicative

Vérifier :

- http://localhost:9600/_node/stats/pipelines : Visualisation des évènements traités par logstash
- http://localhost:9200/_cat/indices : Index ElasticSearch, doit contenir un index logstash-yyyy-MM-dd
- http://localhost:9200/logstash*/_search : Doit retourner le total de documents dans l'index logstash

Accéder à Kibana (http://localhost:5601) et créer un index pattern ***logstash-****.
Visualiser les logs dans Kibana

4.Messaging

4.1 Saga Pattern

Nous voulons implémenter la saga suivante :

Étape	Service	Transaction	Transaction de compensation
1	*OrderService*	*createOrder()*	*rejectOrder()*
2	*AccountService*	*handleCreateOrder()*	
3	*DeliveryService*	*createDelivery()*	
4	*OrderService*	*approveOrder()*	

Nous implémentons le pattern via un chorégraphie de messages.

Chaque micro-service envoie des messages sur des topics Kafka:

- 1. *PaymentRequest* sur le topic « order »
- 2. Transaction Pivot : *PaymentResponse* sur le topic « account »
- 3,4 *Réactions à l'issue précédente, transaction rééessayable ou compensation*

Démarrage du cluster Kafka via *docker-compose*

Récupération des projets *account-service* et *delivery-service* et de la configuration kafka

Développement order-service

4.2 Streaming temps réel avec Spring Boot Stream et Kafka

Objectifs :

1. Recevoir un message via une API REST
2. L'écrire sur un topic Kafka
3. Développer un second micro-service à posteriori
4. Traiter les précédents messages

4.2.1 Stack Kafka + ELK

Un fichier *docker-compose* permettant de démarrer kafka et zookeeper est fourni.

Il nécessite la déclaration de la machine kafka vers localhost (/etc/hosts)

4.2.2 Importation 1er micro-service

Importer le projet Maven ***position-service*** dans Spring Tools Suite, visualiser les starters, la configuration et démarrer l'application. Vérifier la bonne connexion à Kafka.

Démarrer JMeter et ouvrir le fichier JMX fourni.

Exécuter le script et vérifier le bon envoi de message dans le topic Kafka

4.2.3 Mise en place second micro-service

Créer un projet ***average-service*** avec les bons starters dans Spring Tools Suite.

Écrire un miro-services prenant en entrée le topic précédent et écrivant dans un topic de sortie les positions moyennes par minute.

`Le code calculant les moyenne est fourni`

Configurer l'application pour quel traite les messages à posteriori

4.2.4 Remontée des informations dans ElasticStack

Visualiser et Placer le fichier de configuration *logstash-elk.conf* fourni au bon endroit

Accéder à *localhost:9200* pour vérifier le bon fonctionnement de ES

à ***localhost:9200/_cat/indices*** pour vérifier les index créés

à ***localhost:9600/_node/stats/pipeline*** pour surveiller les flux de messages de la pipeline

Ouvrir Kibana et aller sur la devConsole Kibana

Exécuter les commandes suivantes dans la console Kibana, ces commandes créen un index positions contenant un champ location de type geo_point :

```
DELETE /positions

PUT /positions

PUT /positions/_mapping
{
  "properties": {
    "location" : {
            "type" : "geo_point"
          }
  }
}
```

Alimenter l'index en utilisant JMeter et les 2 micro-services précédents

Visualiser les données dans Kibana (vous devez auparavant créer un IndexPattern)

Afficher la répartition des coursiers sur une carte Kibana

4.3 SpringCloud DataFlow

Objectifs :

- Aperçu de SpringCloud DataFlow
  https://dataflow.spring.io/docs/stream-developer-guides/getting-started/stream/

4.3.1 Installation

Un fichier docker-compose permettant de démarrer ;

- Spring Cloud Data Flow Server
- Spring Cloud Skipper Server
- MySql
- Kafka

4.3.2 Création du stream

Accéder au tableau de bord du serveur : http://localhost:9393/dashboard

Streams → Create Stream

Dans la zone de texte, saisir : ***http | log***

Sauvegarder et donner un nom au stream : ***http-ingest***

4.3.3 Déploiement

Cliquer sur le bouton *Deploy*

Ajouter une propriété à la source ***http*** en fixant un port ne rentrant pas en conflit dans votre environnement par exemple 9000, Garder les autres valeurs par défaut

4.3.4 Vérification

Après avoir obtenu le statut déployé, envoyer une requête sur le port http :

curl http://localhost:9000 -H "Content-type: text/plain" -d "Happy streaming"

Dans l'onglet ***Runtime***, sélectionner le micro-services ***log*** et copier le chemin de ***stdout***
Visualiser la console de ce conteneur via :
docker exec -it skipper tail -f /path/from/stdout/textbox/in/dashboard

Vous pouvez également observer les processus s'exécutant sur le container skipper :
docker exec -it bin/bash
ps -aux
opt/openjdk/bin/jstack <pid_of_log_process>

Atelier 5 : Spring Cloud avec oAuth2

Objectifs :

- Architecture où chaque micro-service peut préciser des ACLs par rapport à l'identité du client (scope) et l'utilisateur final
- Utilisation de JWT avec un secret partagé pour limiter le trafic réseau

5.1. Mise en place du serveur d'autorisation (KeyCloak)

Démarrer un serveur KeyCloak via Docker :

```
docker run -e KEYCLOAK_USER=admin -e KEYCLOAK_PASSWORD=admin -p
8089:8080 --name keycloak jboss/keycloak
```

- Connecter vous à sa console d'administration avec admin/admin
- Créer un Realm ***StoreRealm***
- Sous ce realm, créer 2 clients
  - ***store-app***
  - ***monitor***

    Pour ces 2 clients, positionner ***access-type*** à ***confidential*** et Valid Redirect Uri à *******

    Donner un scope ***MONITOR*** au client *monitor*
- Créer ensuite un utilisateur ***user/secret***

Obtention des jeton :

Grant type *password* pour le client *store-app*

```
curl -XPOST http://localhost:8089/auth/realms/StoreRealm/protocol/openid-connect/token -
d grant_type=password -d client_id=store-app -d client_secret=<client_secret> -d
username=user -d password=secret
```

Obtenir un jeton de rafraîchissement en utilisant une requête POST avec :

```
client_id:store-app
client_secret:<client_secret>
'refresh_token': refresh_token_requete_précédente,
grant_type:refresh_token
```

Grant type *client_credentials* pour le client monitor

Dans KeyCloak, autoriser le grant type ***client_credentials***

Settings → Access Type = Confidential

Service Accounts Enabled

Tester avec un requête curl :
curl -XPOST http://localhost:8089/auth/realms/StoreRealm/protocol/openid-connect/token -d grant_type=client_credentials -d client_id=monitor -d client_secret=<client_secret>

5.2. OAuth2 login et sécurité stateful sur la Gateway

Ajouter les starters ***`security`*** ***et*** ***`oauth2-client`*** au projet gateway

Configurer a sécurité de la gateway (Bean étendant *SecurityFilterChain*) comme suit :

```
@Bean
public SecurityWebFilterChain securitygWebFilterChain(
  ServerHttpSecurity http) {
    return http.authorizeExchange()
            .pathMatchers("/actuator/**").permitAll()
            .pathMatchers("/auth/**").permitAll()
      .anyExchange().authenticated()
      .and().
      oauth2Login().csrf().disable().build();
}
```

Configurer le client oauth2 de Keycloak comme suit :

```
spring :
  security:
    oauth2:
      client:
        provider:
          keycloak:
            token-uri: http://localhost:8089/auth/realms/StoreRealm/protocol/openid-
connect/token
            authorization-uri:
http://localhost:8089/auth/realms/StoreRealm/protocol/openid-connect/auth
            user-info-uri: http://localhost:8089/auth/realms/StoreRealm/protocol/openid-
connect/userinfo
            user-name-attribute: preferred_username
        registration:
          store-app:
            provider: keycloak
            client-id: store-app
            client-secret: 57abb4f6-5130-4c73-9545-6d377dd947cf
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/keycloak"
```

Accéder ensuite à l'API via la gateway via un navigateur

Pour mieux comprendre, vous pouvez voir le filtre configuré dans la console et également augmenter les traces via :

```
logging:
  level:
    org.springframework.security: debug
```

5.3. ACLS oAuth2 : Gateway as resource server

Ajouter les starters ***`oauth2-resource-server`*** ***et*** ***`oauth2-jose`*** au projet gateway.

Indiquer la propriété ***`spring.security.oauth2.resourceserver.jwr.issuer-uri`***

Modifier la configuration de la sécurité pour s'adapter au resource server et enlever le login oauth.

```
@Bean
public SecurityWebFilterChain securitygWebFilterChain(
  ServerHttpSecurity http) {
    return http.authorizeExchange()
            .pathMatchers("/actuator/**").permitAll()
            .pathMatchers("/auth/**").permitAll()
      .anyExchange().authenticated()
      .and().
      oauth2ResourceServer(v -> v.jwt()).csrf().disable().build();
}
```

Configurer les routes de la gateway de telle façon que le serveur KeyCloak soit accessible via la gateway.

Utiliser le script *JMeter* ***oAuth2WithGateway*** pour valider votre configuration

5.4. Relai de jeton

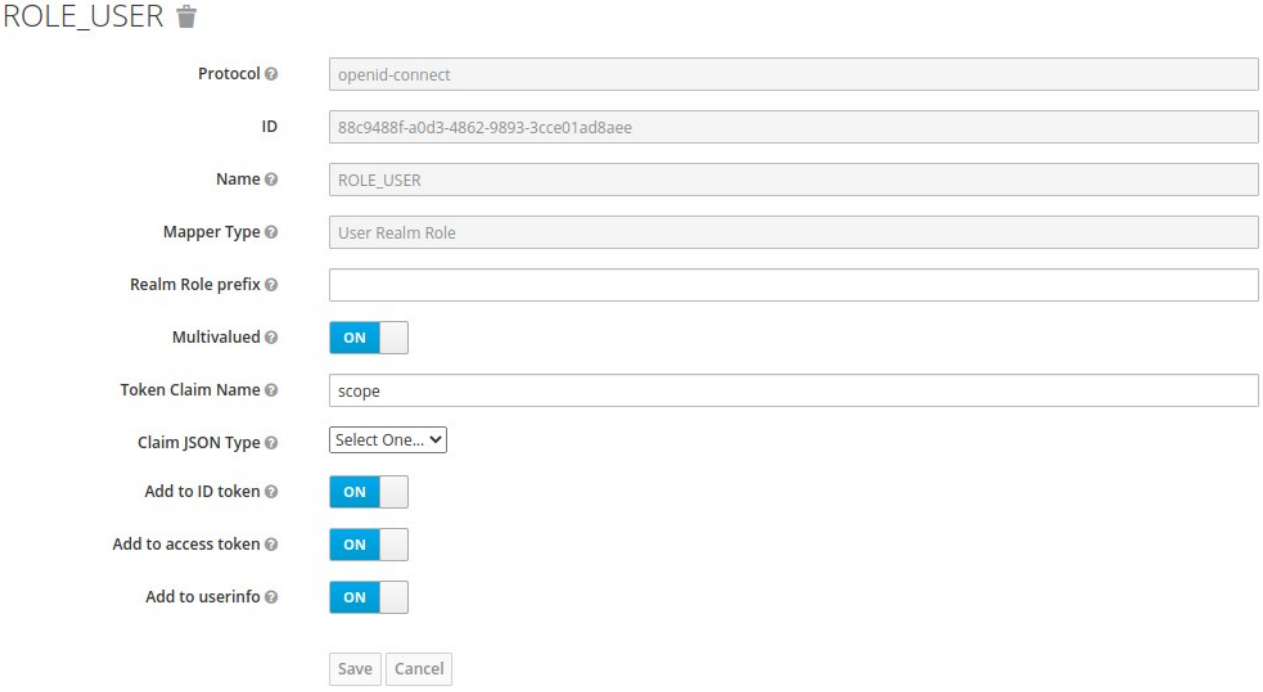
Rôle *USER* à l'utilisateur *user*

Dans KeyCloak, ajouter un rôle USER au client ***store-app***.

Affecter ce rôle à l'utilisateur user

Mapper pour inclure le rôle dans le jeton JWT sous l'attribut scope

Dans le client ***store-app***, ajouter un nouveau mapper, comme suit :



Sécurisation service order-service

Configurer *order-service* comme *ResourceServer* de la même manière que gateway

Définir les ACLs suivantes :

- Pour toutes le requête le client oAuth2 doit être dans le scope USER
- Pour accéder aux endpoints ***actuator***, le client doit être dans le scope MONITOR

Autoriser le relai des entêtes d'autorisation dans la configuration de la gateway

Utiliser le script jMeter ***oAuth2WithGateway*** pour valider votre configuration :

Le groupe d'utilisateur store-app Client valide les protections via les rôles USER

Le groupe d'utilisateur monitor client valide les protections via le scope associé à monitor

Ateliers 6 : Déploiement via conteneurs

Objectifs

- Déployer la stack sous forme de conteneurs
  - Avec docker-compose
  - Avec des manifeste Kubernetes

6.1 Docker et docker-compose

Réorganiser les projets en projet multi-modules Maven afin de pouvoir construire tous les artefacts en une seule commande

Choisir un micro-service applicatif et mettre au point son Dockerfile

Pour les micro-services en version 2.3.x utiliser le plugin Spring Boot

Compléter et corriger le fichier *docker-compose* fourni afin qu'il puisse démarrer toute la stack

Scaler dynamiquement un conteneur

6.2 Kubernetes

Mise au point des ressources Kubernetes fournies. (Voir *delivery-service/src/main/k8*)