Cahier de TP
« Gestion centralisée de la sécurité avec KeyCloak»

Pré-requis :

- Bonne connexion Internet
- NodeJS
- Docker
- Git

Atelier 1: Installation et sécurisation d'une première application

1.1 Installation avec Docker

```
docker run -p 8080:8080 -e KEYCLOAK_ADMIN=admin -e KEYCLOAK_ADMIN_PASSWORD=admin quay.io/keycloak/keycloak:20.0.2 start-dev
```

Cela démarre un serveur KeyCloak en mode dev écoutant sur le port 8080

1.2. Création d'un realm

Accéder à la console d'administration

Créer un realm nommé ***formation***

Pour le realm *formation* :

- Créer un user avec comme login mot de passe ***user/secret***
- Créer un groupe ***userGroup***
- Créer un rôle ***USER*** et l'affecter au groupe précédent

Se connecter avec *user/secret* à Keycloak, et parcourir l'interface.

1.3. Sécurisation d'une application

Récupérer les sources fournis

Démarrer l'application front-end :

```
cd frontend
npm install
npm start
```

Puis l'application back-end

```
cd backend
npm install
npm start
```

Accéder à *http://localhost:8000* et vérifier la présence d'un bouton login

Enregistrement du client ***frontend***

Dans la console d 'admin de Keycloak, créer un nouveau client :

- Client ID: ***frontend***
- Client Protocol: openid-connect
- Root URL: http://localhost:8000
- Décocher Client Access Grant

Après la sauvegarde, compléter les informations du client :

- Valid Redirect URI : http://localhost:8000/
- Logout URL : + (cela inclut toutes les Valid Redirect URIs)
- Web origins : +

Activer ensuite le bouton login dans l'application front-end, vous devriez vous logger et accéder aux boutons permettant de voir les jetons obtenus.

Vous pouvez également renseigner l'attribut ***picture*** de l'utilisateur avec une URL pointant sur une image afin de voir comment une application peut utiliser les attributs stockés dans *KeyCloak*

Accéder directement aux URLs du backend :

- *http://localhost:3000/public*
- *http://localhost:3000/secured*

Accéder ensuite à l'URL sécurisé via l'application de frontend

Atelier 2: OpenID Connect

Récupérer les sources fournis

Démarrer l'application :

```
cd TP2
npm install
npm start
```

Accéder à http://localhost:8000

2.1 Endpoint de découverte

Activer le bouton 1 – Discovery

Vérifier l'URL de l'issuer (il contient le nom du realm Keycloak)

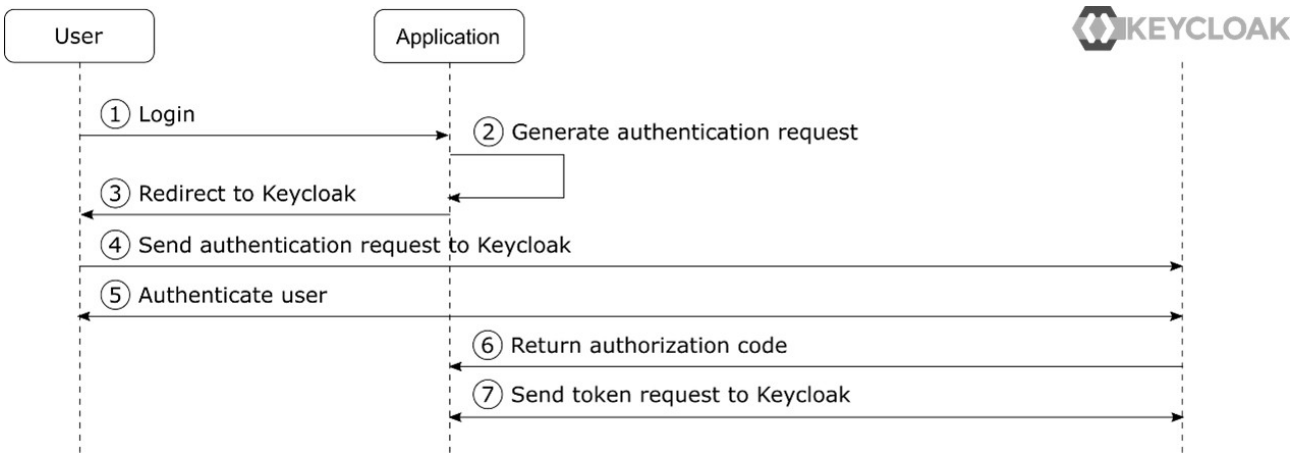
Activer ensuite le bouton « *Load OpenID Provider Configuration* »

Elle provoque la requête GET :
http://localhost:8080/realms/<realm-name>/.well-known/openid-configuration

Visualiser les informations accessibles.

2.2 Authentification

Le processus détaillé de l'authentification est le suivant :



Pour envoyer la requête d'authentification, cliquer sur le bouton correspondant.

Un formulaire est proposée par l'application :

- *client_id* : Le client enregistré chez Keycloak
- *scope* : La valeur par défaut est openid , ce qui signifie une requête OpenID
- *prompt* : Ce champ peut être utilisé pour différents buts :

    - *none* : Keycloak n’affiche pas d’écran de login mais authentifie l’utilisateur seulement si l’utilisateur est déjà loggé
    - *login* : Demande à l’utilisateur de s’authentifier sur Keycloak même si il est déjà loggé
- *max_age* : Maximum en seconds de la dernière authentification valide
- *login_hint* : Si l’application connaît la login de l’utilisateur, il peut pré-remplir le champ login de Keycloak

Générer la requête et l’envoyer. Le point d’accès est
GET http://localhost:8080/realms/<realm-name>/protocol/openid-connect/auth

La réponse doit contenir le code d’autorisation

Essayer d’autres combinaisons de paramètres

2.3 Obtention des jetons

La 3ème étape consiste à obtenir les jetons avec le code d’autorisation.

Utiliser les boutons adéquats

Observer la réponse et ses différents champs

Observer le jeton d’identification déchiffré.

2.4 Création de scope et personnalisation des revendications

Créer un client scope ***custom_scope***

Dans l’onglet Mappers, créer un mapper :

- Type : ***User Attribute***
- Name : ***picture***
- User Attribute : ***picture*** (l’attribut créé à l’atelier 1)
- Token Claim Name: ***picture***
- Claim JSON Type: ***String***

S’assurer que cet attribut sera ajouté au jeton d’identification.

Sélectionner ensuite le client *frontend* et ajouter *custom_scope* comme scope optionnel du client

Réauthentifier vous en précisant dans le paramètre *scope* : ***openid custom_scope***

Visualiser le jeton, l’attribut *picture* doit être présent

Sélectionner ensuite le scope ***roles***, puis l’onglet *Mappers* et *realm roles*, ajouter les ream roles au jeton d’identification.

Se réauthentifier et vérifier que les rôles utilisateurs sont dans le jeton.

Accéder ensuite au endpoint de UserInfo.

Sélectionner le client *frontend*, dans client scopes, sélectionner ***frontend-dedicated*** et y créer un

nouveau Mapper avec les informations suivantes :

- Name : ***custom_claim***
- Mapper Type : ***Hardcoded claim***
- Token Claim Name: ***custom_claim***
- Claim value: ***A custom value***
- Claim JSON Type: ***String***

S’assurer que cette revendication est incluse dans le point d’accès *UserIinfo*

Vérifier

2.5 Logout

Configurer une URI de Front-channel logout à : http://localhost:8000/logout
Effectuer un logout et observer les requêtes effectuées par la navigateur

Configurer une URI de Back-channel logout à : http://localhost:8000/logout
Effectuer un logout et observer la console de Keycloak

Atelier 3 : Consumer API

3.1 Implémentation

Le projet est composé de :

- Une classe principale ***KafkaConsumerApplication*** qui prend en arguments :
  - ***nbThreads*** : Un nombre de threads
  - ***sleep*** : Un temps de pause

    L'application instancie *nbThreads KafkaConsumerThread* et leur demande de s'exécuter. Le programme s'arrête au bout d'un certains temps.
- Une classe ***KafkaConsumerThread*** qui une fois instanciée poll le topic position tout les temps de pause.
  A la réception des messages, il affiche la clé, l'offset et le timesatmp de chaque message. Il met également à jour une Map qui contient le nombre de mise à jour pour chaque coursier
- Le package ***model*** contient les classes modélisant les données
  - ***Position*** : Une position en latitude, longitude
  - ***Courier*** : Un coursier associé à une position

Compléter la boucle de réception des messages

Pour cela, vous devez

- Initialiser un *KafkaConsumer*
- Fournir un Deserialiseur
- Implémenter la boucle de réception

Pour tester la réception, vous pouvez utiliser le programme précédent et le lancer afin qu'il exécute de nombreux message :

Par exemple :

```
producer_home$ java -jar target/producer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 10 100000 500 0
```

3.2 Tests

Une fois le programme mis au point, effectuer plusieurs tests

Tester qu'aucun message n'est perdu :
Démarrer le programme avec 1 thread arrêter puis redémarrer avec la même configuration

Tester la réaffectation de partitions :
Démarrer avec 2 threads puis 3 threads et visualiser la répartition des partitions

Démarrer également le programme avec 5 threads

Atelier 4. Sérialisation Avro

4.1 Démarrage de Confuent Registry

Télécharger une distribution de la Confluent Platform version communautaire :
curl -O http://packages.confluent.io/archive/7.2/confluent-community-7.2.1.zip
Dézipper

Démarrer le seveur de registry via :
./schema-registry-start ../etc/schema-registry/schema-registry.properties

Accéder à localhost:8081/subjects

4.2 Producteur de message

Récupérer le **pom.xml** fourni
Mettre au point un **schéma Avro**
Effectuer un ***mvn compile*** et regarder les classes générées par le plugin Avro

Reprendre les classes du projet producer sans les classes du modèle
Dans la classe main, poster le schéma dans le serveur registry :

```
String schemaPath = "/Courier.avsc";
// subject convention is "<topic-name>-value"
String subject = TOPIC + "-value";


InputStream inputStream =
KafkaProducerApplication.class.getResourceAsStream(schemaPath);

Schema avroSchema = new Schema.Parser().parse(inputStream);

CachedSchemaRegistryClient client = new
CachedSchemaRegistryClient(REGISTRY_URL, 20);

client.register(subject, avroSchema);
```

Dans le producteur de message modifier la classe producer afin

- qu'il compile
- qu'il utilise un sérialiseur de valeur de type **io.confluent.kafka.serializers.KafkaAvroSerializer**
- Qu'il renseigne la clé **AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG**

Modifier le topic d'envoi et tester la production de message.

Éventuellement, configurer *akhq* afin qu'il puisse décoder les messages. Vous devez indiquer l'URL du registre de schéma dans la configuration.

4.3 Consommateur de message

Reprendre le même ***pom.xml*** que le projet producer
Ne plus utiliser les classes de modèle mais la classe d'Avro ***GenericRecord***
Modifier les propriétés du consommateur :

- Le désérialiseur de la valeur à :

"io.confluent.kafka.serializers.KafkaAvroDeserializer"

- La propriété ***schema.registry.url***

Consommer les messages du topic précédent

4.4 Mise à jour du schéma

Metter à jour le schéma (ajout du champ ***firstName*** dans la structure ***Courier*** par exemple)

Relancer le programme de production et visualiser la nouvelle version du schéma dans le registre

Consommer les messages sans modifications du programme consommateur

Atelier 5. Frameworks

Objectifs : Utiliser les frameworks Spring et Quarkus pour consommer les enregistrements du topic *position* précédent

5.1 Spring Cloud Stream

Récupérer le projet Maven/SpringBoot fourni.

Le fichier *pom.xml* déclare en particulier les dépendances suivantes :

- `spring-cloud-stream`
- `spring-cloud-stream-binder-kafka`

Déclarer un Bean Spring ayant pour nom ***position*** de type ***Consumer<Message<String>>***

Dans le fichier de configuration ***src/main/resources/application.yml*** :

- Utiliser le nom de la méthode annotée Bean pour binder le topic *position*
- Déclarer les *bootstrap-servers* Kafka

Tester en alimentant le topic

5.2 MP Messaging avec Quarkus

Récupérer le projet Maven/Quarkus fourni.

Le fichier *pom.xml* déclare en particulier les dépendances suivantes :

- `quarkus-smallrye-reactive-messaging-kafka`
- `quarkus-resteasy-reactive-jackson`

Déclarer un Bean ***PositionService*** déclarant un méthode de réception de message

Dans le fichier de configuration ***src/main/resources/application.properties*** :

- Déclarer les bootstrap-servers Kafka

Pour démarrer l'application :
mvn quarkus:dev
Tester en alimentant le topic

Atelier 6. Kafka Connect

Objectifs : Déverser le topic *position* dans un index ElasticSearch

6.1 Installations ElasticSearch + Connecteur

Démarrer *ElasticSearch* et *Kibana* en se plaçant dans le répertoire du fichier *docker-compose.yml* fourni, puis :

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

Se connecter à kibana et dans la DevConsole exécuter :

```
PUT /position
{
  "mappings":{
     "properties": {
         "@timestamp": {
             "type": "date",
             "format": "epoch_millis"
         }
     }
 }
}
```

Récupérer le projet OpenSource ElasticSearchConnector de Confluent et se placer sur une release puis builder.

```
git clone https://github.com/confluentinc/kafka-connect-elasticsearch.git
```

```
cd kafka-connect-elasticsearch
```

```
git checkout v11.0.3
```

```
mvn -DskipTests clean package
```

Copier ensuite toutes les librairies présentes dans ***target/kafka-connect-elasticsearch-11.0.3-package/share/java/kafka-connect-elasticsearch/*** dans le répertoire ***lib**s* de Kafka

6.1 Configuration Kafka Standalone

Mettre au point un fichier de configuration ***elasticsearch-connect.properties*** contenant :

```
name=elasticsearch-sink
connector.class=io.confluent.connect.elasticsearch.ElasticsearchSinkConnector
tasks.max=1
topics=position
topic.index.map=position:position_index
connection.url=http://localhost:9200
type.name=log
key.ignore=true
```

```
schema.ignore=true
```

Démarrer ***kafka-standalone*** et alimenter le topic ***position***

Vous pouvez visualiser les effets du connecteur

- Via ElasticSearch : http://localhost:9200/position/_search
- Via Kibana : http://localhost:5601

Optionnel : Améliorer le fichier de configuration afin d'introduire le timestamp

Atelier 7 : KafkaStream

Objectifs :
Écrire une mini-application Stream qui prend en entrée le topic ***position*** et écrit en sortie dans le topic ***position-out*** en ajoutant un timestamp aux valeurs d'entrée

Importer le projet Maven fourni, il contient les bonnes dépendances et un package *model* :

- Un champ timesatmp a été ajouté à la classe Courier
- Une implémentation de *Serde* permettant la sérialisation et la désérialisation de la classe *Courier* est fournie

Avec l'exemple du cours, écrire la classe principale qui effectue le traitement voulu

Atelier 8 : Fiabilité

8.1. At Least Once, At Most Once

Objectifs :

Explorer les différentes combinaisons de configuration des producteurs et consommateurs vis à vis de la fiabilité sous différents scénarios de test.

On utilisera un cluster de 3 nœuds avec un topic de 3 partitions et un mode de réplication de 2.

Le scénarios de test envisagé (Choisir un scénario parmi les 2):

- Redémarrage de broker(s)
- Ré-équilibrage des consommateurs

Les différentes combinaisons envisagées

- Producteur : *ack=0* ou *ack=all*
- Consommateur : auto-commit ou commits manuels

Les métriques surveillés

- Producteur : Trace WARN ou +
- Consommateur : Trace Doublon ou messages loupés

Méthodes :

Supprimer le topic ***position***

Le recréer avec

```
./kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 2 --partitions 3 --topic position
```

Vérifier l'affectation des partitions et des répliques via :

```
./kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic position
```

Récupérer les sources fournis et construire pour les 2 clients l'exécutable via
`mvn clean package`

Dans 2 terminal, démarrer 2 consommateurs :

```
java -jar target/consumer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 1 1000 >> log1.csv
```

```
java -jar target/consumer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 1 1000 >> log2.csv
```

Dans un autre terminal, démarrer 1 producteur multi-threadé :

```
java -jar target/producer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 20 5000 10 <0|1|2> <0|all>
```

Pendant la consommation des messages, en fonction du scénario : arrêter et redémarrer un broker ou

un consommateur.

Un utilitaire ***check-logs*** est fourni permettant de détecter les doublons ou les offsets.

```
java -jar check-logs.jar <log.csv>
```

8.2. Exactly-Once et transaction

8.2.1 Producteur transactionnel

Modifier le code du producer afin d'englober plusieurs envois de messages dans une transaction. Certaines transactions sont validées d'autres annulés

8.2.2 Consommateur

Modifier la configuration du consommateur afin qu'il ne lise que les messages committés

Atelier 9 : Administration

Exécuter les producteurs et les consommateurs pendant les opérations d'administration

9.1 Reassign partitions, Retention

Extension du cluster et réassignement des partitions

Modifier le nombre de partitions de position à 8

Vérifier avec

`./kafka-topics.sh --bootstrap-server localhost:909 --describe --topic position`

Ajouter un nouveau broker dans le cluster.

Réexécuter la commande

`./kafka-topics.sh --bootstrap-server localhost:909 --describe --topic position`

Effectuer une réaffectation des partitions en 3 étapes

Rétention

Visualiser les segments et apprécier la taille

Pour le topic ***position*** modifier le ***segment.bytes*** à 1Mo

Diminuer le ***retention.bytes*** afin de voir des segments disparaître

9.2 Rolling restart

Sous charge, effectuer un redémarrage d'un broker.

Vérifier l'état de l'ISR

9.3 Mise en place de SSL

Travailler dans un nouveau répertoire ***ssl***

```
#  Créer son propre CA (Certificate Authority)
openssl req -new -newkey rsa:4096 -days 365 -x509 -subj "/CN=localhost" -keyout ca-key -out ca-cert -nodes
# Générer une paire clé publique/privé pour chaque serveur
keytool -keystore server.keystore.jks -alias localhost -validity 365 -genkey -keyalg RSA -storetype pkcs12
# Create Certificate signed request (CSR):
keytool -keystore server.keystore.jks -certreq -file cert-file -storepass secret -keypass secret -alias localhost
# Générer le CSR signé avec le CA
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-file-signed -days 365 -CAcreateserial -passin pass:secret
# Importer le certifcat CA dans le KeyStore serveur
keytool -keystore server.keystore.jks -alias CARoot -import -file ca-cert -storepass secret -keypass secret -noprompt
# Importer Signed CSR dans le KeyStore
keytool -keystore server.keystore.jks -import -file cert-file-signed -storepass
```

```
secret -keypass secret -noprompt -alias localhost
# Importer le certificat CA dans le TrustStore serveur
keytool -keystore server.truststore.jks -alias CARoot -import -file ca-cert -
storepass secret -keypass secret -noprompt


#Importer le CA dans le client
keytool -keystore client.truststore.jks -alias CARoot -import -file ca-cert -
storepass secret -keypass secret -noprompt
```

Configurer le listener SSL et les propriétés SSL suivante dans *server.properties*

```
ssl.keystore.location=/home/dthibau/Formations/Kafka/github/solutions/ssl/server.keystore.jks
ssl.keystore.password=secret
ssl.key.password=secret
ssl.truststore.location=/home/dthibau/Formations/Kafka/github/solutions/ssl/server.truststore.jks
ssl.truststore.password=secret
security.inter.broker.protocol=SSL
ssl.endpoint.identification.algorithm=
ssl.client.auth=none
```

Démarrer le cluster kafka et vérifier son bon démarrages

Dans les traces doivent apparaître :

```
Registered broker 1 at path /brokers/ids/1 with addresses:
SSL://localhost:9192,
```

Vérifier également l'affichage du certificat avec :

```
openssl s_client -debug -connect localhost:9192 -tls1_2
```

Mettre au point un fichier ***client-ssl.properties*** avec :

```
security.protocol=SSL
ssl.truststore.location=/home/dthibau/Formations/Kafka/github/solutions/ssl/client.truststore.jks
ssl.truststore.password=secret
```

Vérifier la connexion cliente avec par exemple

```
./kafka-console-producer.sh --broker-list localhost:9192 --topic ssl
    --producer.config client-ssl.properties
```

9.3.2 (Optionnel): Authentification et ACL avec SCRAM

Voir *https://medium.com/egen/securing-kafka-cluster-using-sasl-acl-and-ssl-dec15b439f9d*

9.4 Mise en place monitoring Prometheus, Grafana

Dans un premier temps, démarré une ***JConsole*** et visualiser les Mbeans des brokers, consommateurs et producteurs

Dans un répertoire de travail ***prometheus***

```
wget https://repo1.maven.org/maven2/io/prometheus/jmx/jmx_prometheus_javaagent/0.6/jmx_prometheus_javaagent-0.6.jar
wget https://raw.githubusercontent.com/prometheus/jmx_exporter/master/example_configs/kafka-2_0_0.yml
```

Modifier le script de démarrage du cluster afin de positionner l'agent Prometheus :

```
KAFKA_OPTS="$KAFKA_OPTS -javaagent:$PWD/jmx_prometheus_javaagent-0.2.0.jar=7071:$PWD/kafka-0-8-2.yml" \
  ./bin/kafka-server-start.sh config/server.properties
```

Attention, Modifier le port pour chaque broker

Redémarrer le cluster et vérifier `http://localhost:7071/metrics`

Récupérer et démarrer un serveur prometheus

```
wget https://github.com/prometheus/prometheus/releases/download/v2.0.0/prometheus-2.0.0.linux-amd64.tar.gz
tar -xzf prometheus-*.tar.gz
cd prometheus-*
cat <<'EOF' > prometheus.yml
global:
 scrape_interval: 10s
 evaluation_interval: 10s
scrape_configs:
 - job_name: 'kafka'
   static_configs:
    - targets:
      - localhost:7071
      - localhost:7072
      - localhost:7073
      - localhost:7074
EOF
./prometheus
```

Récupérer et démarrer un serveur Grafana

```
sudo apt-get install -y adduser libfontconfig1
wget https://dl.grafana.com/oss/release/grafana_7.4.3_amd64.deb
sudo dpkg -i grafana_7.4.3_amd64.deb

sudo /bin/systemctl start grafana-server
```

Accéder à http://localhost:3000 avec ***admin/admin***

Déclarer la datasource Prometheus

Importer le tableau de bord : https://grafana.com/grafana/dashboards/721

Les métriques des brokers devraient s’afficher