Cahier de TP
« Messagerie distribuée avec Kafka »

Pré-requis :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux
- JDK11+, Maven
- IDE Recommandés : STS 4, IntelliJIDEA, VSCode
- Docker, Git

Table des matières

Atelier 1: Installation

1.1. Installation broker Kafka

1.1.1 Installation à partir d'une archive

Récupérer une distribution de *Kafka*

Démarrer un broker via :

./kafka-server-start.sh [-daemon] ../config/server.properties

Vérifier le bon démarrage via la console

Faites des vérification en créant un topic et envoyant des messages via les utilitaires.

Optionnel Installer l'outil graphique akhq (*https://akhq.io/)*

- *Télécharger une archive (.zip) et décompresser dans un répertoire*

1.1.2 Installation à partir de docker

On utilise l'image fournie par bitnami :
https://github.com/bitnami/containers/tree/main/bitnami/kafka

Visualiser le fichier ***docker-compose-dev.yml***

Démarrer le container via :
docker-compose -f docker-compose-dev.yml up -d

Visualiser les logs.

Obtenir un shell sur le container.

docker exec -it 1_installation_kafka_1 /bin/bash

Se placer dans le répertoire ***/opt/bitnami/kafka/bin/***

Faites des vérification en créant un topic et envoyant des messages via les utilitaires.

1.2. Mise en place cluster Kafka

Etape 1 : Création du cluster ID et formattage des répertoires de logs

Télécharger et dézipper la distribution de Kafka dans ***$KAFKA_DIST***

Créer un répertoire ***$KAFKA_LOGS*** qui stockera les messages des 3 brokers

Créer un répertoire ***$KAFKA_CLUSTER*** et 3 sous-répertoires : ***broker-1***, ***broker-2***, ***broker-3***

Copier le fichier de la distribution *$KAFKA_DIST/config/kraft/server.properties* dans les 3 sous-répertoires de $***KAFKA_CLUSTER***

Editer les 3 fichier server.properties afin de modifier les propriétés :

- ***node.id***
- ***listeners***
- ***advertised.listeners***
- ***controller.quorum.voters***
- ***log.dir*** à ***$KAFKA_LOGS/broker-<id>***

Générer un id de cluster :

$KAFKA_DIST/bin/kafka-storage.sh random-uuid

Utiliser l'ID du cluster pour formatter les 3 répertoires

Pour chaque broker, formatter son répertoire de log avec :

bin/kafka-storage.sh format -t <cluster_id> -c $KAFKA_CLUSTER/broker-<id>/server.properties

Etape 2 : Mise au point d'un script de démarrage/arrêt

Mettre au point un script sh permettant de démarrer les 3 brokers en mode daemon :

Exemple :

```
#!/bin/sh


export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64/
export KAFKA_DIST=/home/dthibau/Formations/Kafka/MyWork/kafka_2.13-3.3.1
export KAFKA_CLUSTER=/home/dthibau/Formations/Kafka/github/solutions/kafka-cluster
export KAFKA_LOGS=/home/dthibau/Formations/Kafka/MyWork/kafka-logs


$KAFKA_DIST/bin/kafka-server-start.sh -daemon $KAFKA_CLUSTER/broker-1/server.properties
$KAFKA_DIST/bin/kafka-server-start.sh -daemon $KAFKA_CLUSTER/broker-2/server.properties
$KAFKA_DIST/bin/kafka-server-start.sh -daemon $KAFKA_CLUSTER/broker-3/server.properties
```

Visualiser les traces de démarrages :

tail -f $KAFKA_HOME/logs/server.log

Mettre au point un script d'arrêt

```
#!/bin/sh


export KAFKA_DIST=/home/dthibau/Formations/Kafka/MyWork/kafka_2.12-2.4.1

$KAFKA_DIST/bin/kafka-server-stop.sh
```

Effectuer les vérifications de création de topic et envoi/réception de messages

```
bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1
--topic test

bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic test

bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning
```

Option archive : Installation akhq

Télécharger une distribution d'akhq (akhq-all.jar)

Récupérer le fichier de configuration fourni ***application-basic.yml***

Exécuter le serveur via :
java -Dmicronaut.config.files=./application-basic.yml -jar akhq-0.21.0-all.jar

1.2.2 *Option Docker*

Visualiser le fichier ***docker-compose.yml***

Démarrer la stack et observer les logs de démarrages

1.2.3 Toutes options

Ensuite soit avec les utilitaires Kafka soit avec l'UI de *akhq*

Créer un topic ***testing*** avec 5 partitions et 2 répliques :

./kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 2 --partitions 5 --topic testing

Lister les topics du cluster

Démarrer un producteur de message

./kafka-console-producer.sh --broker-list localhost:9092 --topic testing --property "parse.key=true" --property "key.separator=:"

Saisir quelques messages

Accéder à la description détaillée du topic

Visualiser les répertoires de logs sur les brokers :

./kafka-log-dirs.sh --bootstrap-server localhost:9092 --describe

Consommer les messages depuis le début

Dans une autre fenêtre, lister les groupes de consommateurs et accéder au détail du groupe de consommateur en lecture sur le topic testing

Atelier 2: Producer API

Importer le projet Maven fourni

Le projet est composé de :

- Une classe principale ***KafkaProducerApplication*** qui prend en arguments :
  - ***nbThreads*** : Un nombre de threads
  - ***nbMessages*** : Un nombre de messages
  - ***sleep*** : Un temps de pause
  - ***sendMode*** : Le mode d'envoi : 0 pour Fire_And_Forget, 1 pour Synchrone, 2 pour Asynchrone

    L'application instancie *nbThreads KafkaProducerThread* et leur demande de s'exécuter ; quand toutes les threads sont terminées. Elle affiche le temps d'exécution
- Une classe ***KafkaProducerThread*** qui une fois instanciée envoie *nbMessages* tout les temps de pause selon un des 3 modes d'envoi.
  Les messages sont constitués d'une au format String (*courier.id*) et d'une payload au format JSON (classe *Courier)*
- Le package ***model*** contient les classes modélisant les données
  - ***Position*** : Une position en latitude, longitude
  - ***Courier*** : Un coursier associé à une position
  - ***SendMode*** : Une énumération des modes d'envoi

Compléter les méthodes d'envoi de *KafkaProducerThread*.

Pour cela vous devez :

- Initialiser un *KafkaProducer<String,Courier>* et y positionner des sérialiseurs JSON pour la classe *Courier*
- Construire un *ProducerRecord* pour chaque messages
- Implémenter les 3 méthodes d'envoi

Via les commandes utilitaires de Kafka, vérifier la création du topic et consommer les messages

Supprimer le topic et le recréer avec un nombre de *partitions=3* et un *replication-factor=2*

Envoyer des messages

Construire un jar exécutable avec :
mvn package

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

Option archive
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***

Option Docker

Visualiser le fichier ***docker-compose.yml*** fournis

Démarrer la stack

Accéder à *localhost:8081/subjects*

4.2 Producteur de message

Créer un nouveau projet Maven ***producer-avro***
Récupérer le **pom.xml** fourni
Mettre au point un **schéma Avro : Courier.avsc**
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 *KafkaProducerThread* afin

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

AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG

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

Accéder à localhost:8081/subjects

Puis à Accéder à localhost:8081/schemas/

Vérifier que ***akhq*** puisse lire les messages. (Si option non docker, vérifier la configuration, 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-avro*
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

4.4.1 Evolution du schéma compatible

Mettre à jour le schéma en ajoutant les champs optionnels : ***firstName*** dans la structure ***Coursier***

```
{
      "name": "first_name",
      "type": "string",
      "default": "undefined"
},
```

Fixer les problèmes de compilation

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

Consommer les messages sans modifications du programme consommateur

4.4.2 Evolution du schéma incompatible

Mettre à jour le schéma en ajoutant un champs obligatoire : ***vehicle_id*** dans la structure ***Coursier***

```
      {
            "name": "vehicle_id",
            "type": "int"
      },
```

Fixer les problèmes de compilation

Relancer le programme de production et visualiser l’exception au moment de l’enregistrement du nouveau schéma.

Visualiser les nouveaux messages publiés dans le *topic*

Atelier 5. Frameworks

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

5.1 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

5.2 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

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"
         }
     }
 }
}
```

La commande crée un index elasticsearch ***position*** avec pour l'instant un seul champ *@timestamp*

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 ***libs*** de Kafka

6.2 Configuration KafkaConnect 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, un mode de réplication de 2 et un *min.insync.replica* de 1.

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 le *min.insync.replicas*

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 position-consumer 1000 >> log1.csv
```

```
java -jar target/consumer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 1 position-consumer 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.

Visualisation résultat :

Concaténer les fichiers résultas :

```
cat log1.csv >> cat log2.csv >> log.csv
```

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

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

8.2. Transaction et Exactly-Once

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

8.2.2 Transfert Exactly Once

Modifier le consommateur afin qu'il transfert exactement une fois les messages produits en amont

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.

Option Docker : un docker-compose est fourni

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

Ateliers 10 : Sécurité

10.1 Séparation des échanges réseaux

Installation à partir de l'archive

Modifier les fichiers ***server.properties*** des 3 brokers afin de créer 3 listeners *PLAIN_TEXT* dénommé PLAIN_TEXT, CONTROLLER et EXTERNAL en leur affectant des ports différents

Pour l'instant utiliser des communications en clair pour les 3

3 propriétés de configuration doivent être mises à jour :

- listeners
- advertised.listeners
- listener.security.protocol.map

Avec un programme précédent utiliser le listener EXTERNAL

Installation docker-compose

Le fichier *docker-compose* sépare déjà sur des ports différents la communication contrôleurs, brokers et client externe .

Visualisez la configuration bitnami des listeners

10.2 Mise en place de SSL pour crypter les données

10.2.1 Génération keystore et truststore

Créer nouveau répertoire ***ssl*** permettant de stocker les keystore

Y déposer le script ***TPS/10_Security/10.2_TLS/generate-ssl.sh*** fourni (Provient de Bitnami)

Faites attention lors des réponses aux questions de l'assistant :

- Lorsque vous êtes invité à entrer un mot de passe, utilisez le même pour tous.
  Par exemple : *secret*
- Définissez les valeurs Common Name ou FQDN sur le nom d'hôte de votre conteneur Apache Kafka, par ex. *localhost*. Après avoir saisi cette valeur, lorsque vous êtes invité "Quel est votre nom et prénom ?", saisissez également cette valeur.

A la fin de l'opération, vous devez avoir 2 fichiers *.jks* :

- ***keystore/kafka.keystore.jks*** : Certificats utilisé par les brokers
- ***truststore/kafka.truststore.jks*** : Truststore pour les serveurs

Vous pouvez vérifier les contenus des store avec :

```
keytool -v -list -keystore keystore/kafka.keystore.jks
```

10.2.2 Configuration pour SSL appliqué aux communications inter-broker

Option Archive

Configurer les fichiers *server.properties afin d'a*jouter un nouveau SSL, mappé sur le protocole SSL et l'utiliser pour la communication inter-broker.

Les propriétés à modifier sont :

- `listeners`
- `inter.broker.listener.name`
- `advertised.listeners`
- `listener.security.protocol.map`

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

```
ssl.keystore.location=<your-env>/ssl/server.keystore.jks
ssl.keystore.password=secret
ssl.key.password=secret
ssl.truststore.location=<your-env>/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 démarrage

Dans les traces doivent apparaître :

```
[2023-08-14 10:10:13,308] INFO [SocketServer listenerType=BROKER,
nodeId=2] Started data-plane acceptor and processor(s) for
endpoint : ListenerName(SSL) (kafka.network.SocketServer)
```

Option Docker

Dans l'environnement docker les noms DNS ne sont pas *localhost*. Les certificats générés précédemment ne sont valable.

Donc rester en plaintext pour la communication inter-broker

Solution possible :

A partir du docker-compose, il faut créer 3 certificats différents pour ***kafka-0, kafka-1, kafka-2***

Et les importer dans le truststore

Déposer les keystore et trustore dans `/opt/bitnami/kafka/config/certs/` en faisant un montage de répertoire

Puis effectuer une configuration par variable d'environnement comme suit :

```
environment:
- KAFKA_CFG_NODE_ID=0
- KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=0@kafka-0:9093,1@kafka-1:9093,2@kafka-2:9093
- KAFKA_KRAFT_CLUSTER_ID=abcdefghijklmnopqrstuv
- KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,CONTROLLER://:9093,EXTERNAL://:19092,SSL://:9095
- KAFKA_CFG_INTER_BROKER_LISTENER_NAME=SSL
```

- KAFKA_CFG_CONTROLLER_LISTENER_NAMES=CONTROLLER
- KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka-0:9092,EXTERNAL://localhost:19092,SSL://kafka-0:9095
- KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=CONTROLLER:PLAINTEXT,EXTERNAL:PLAINTEXT,PLAINTEXT:PLAINTEXT,SSL:SSL
- KAFKA_CFG_SSL_KEYSTORE_LOCATION=/opt/bitnami/kafka/config/certs/kafka.keystore.jks
- KAFKA_CFG_SSL_KEYSTORE_PASSWORD=secret
- KAFKA_CFG_SSL_KEY_PASSWORD=secret
- KAFKA_CFG_SSL_TRUSTSTORE_LOCATION=/opt/bitnami/kafka/config/certs/kafka.truststore.jks
- KAFKA_CFG_SSL_TRUSTSTORE_PASSWORD=secret
- KAFKA_CFG_SSL_CLIENT_AUTH=none
- KAFKA_CFG_SSL_ENDPOINT_IDENTIFICATION_ALGORITHM=

10.2.3 Configuration pour SSL appliqué aux communications externes

Utiliser également SSL pour le listener EXTERNAL, il suffit de modifier la propriété :

- listener.security.protocol.map

10.2.4 Accès client via SSL

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

security.protocol=SSL

ssl.truststore.location=/home/dthibau/Formations/SpringKafka/github/solutions/ssl/truststore/kafka.truststore.jks
ssl.truststore.password=secret

Vérifier la connexion cliente avec par exemple

```
$KAFKA_DIST/bin/kafka-console-producer.sh --broker-list localhost:EXTERNAL_PORT --topic ssl --producer.config client-ssl.properties
```

Configurer le projet Spring ***PositionService*** pour configurer le client avec ssl.

Dans KafkaConfig, rajouter les propriétés de configuration suivantes :

```
props.put("security.protocol", "SSL");
props.put("ssl.truststore.location", "<your-config>/kafka.truststore.jks");
props.put("ssl.truststore.password", "secret");
```

Tester un envoi de message, vérifier la bonne configuration du *KafkaProducer* et la production de message

10.3 Authentification avec SASL/PLAIN

10.3.1 Authentification inter-broker

Mettre au point un fichier ***kafka_server_jass.conf*** définissant 2 utilisateurs *admin* et *alice* et indiquant que le serveur utilise l'identité *admin* comme suit :

KafkaServer {
org.apache.kafka.common.security.plain.PlainLoginModule required
username="admin"
password="admin-secret"
user_admin="admin-secret"
user_alice="alice-secret";
};

Modifier le script de démarrage afin qu'il utilise le fichier :

```
export KAFKA_OPTS="-Djavax.net.debug=ssl:handshake:verbose
-Djava.security.auth.login.config=<your-env>/kafka_server_jaas.conf"
```

Modifier les fichiers *server.properties* afin que la communication inter-broker utilise le listener ***SASL_SSL***.

Vous devez modifier les propriétés de configuration :

- *listeners*
- *inter.broker.listener.name*
- *advertised.listeners*

Ajouter également les configurations :

sasl.mechanism.inter.broker.protocol=PLAIN

sasl.enabled.mechanisms=PLAIN

Redémarrer le cluster et vérifier son bon démarrage

10.3.2 Authentification client

Configurer le listener EXTERNAL pour qu'il utilise également SASL_SSL comme protocole

Mettre à jour le fichier ***client-ssl.properties*** comme suit :

security.protocol=SASL_SSL
sasl.mechanism=PLAIN
*sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required *
*username="alice" *
password="alice-secret";

Tester avec :

```
$KAFKA_DIST/bin/kafka-topics.sh --bootstrap-server localhost:19092 --list --
command-config ssl/client-ssl.properties
```

Configurer le projet Spring ***PositionService*** pour configurer le client avec ssl.

Dans KafkaConfig, rajouter les propriétés de configuration suivantes :

```
props.put("security.protocol", "SASL_SSL");
props.put("ssl.truststore.location", "<your-config>/kafka.truststore.jks");
props.put("ssl.truststore.password", "secret");
props.put("sasl.jaas.config","org.apache.kafka.common.security.plain.PlainLoginModule
required username=\"alice\" password=\"alice-secret\";");
props.put("sasl.mechanism", "PLAIN");
```

Tester un envoi de message, vérifier la bonne configuration du *KafkaProducer* et la production de message

10.4 ACL

10.4.1 Configuration brokers

Activer les ACL avec la classe ***org.apache.kafka.metadata.authorizer.StandardAuthorizer***

Définir les super users et la règle allow.everyone.if.no.acl.found=true

Indiquer que le contrôleur doit s'identifier

listener.name.controller.ssl.client.auth=required

authorizer.class.name=org.apache.kafka.metadata.authorizer.StandardAuthorizer
super.users=User:ANONYMOUS,User:admin,User:alice
allow.everyone.if.no.acl.found=true

Démarrer le cluster et vérifier son bon démarrage

Tester l'envoi d'un message par ***PositionService***

A Voir également les liens suivants :

https://github.com/bitnami/containers/issues/23237

https://stackoverflow.com/questions/74671787/kafka-cluster-using-kraft-mtls-and-standardauthorizer-not-starting-up-getting

10.4.2 Définition ACLs

Définir une ACL via Redpanda Console interdisant à l'utilisateur alice d'écrire sur les topics.

Tester le refus d'envoi de message par ***PositionService***

Atelier 11 : *Monitoring*

11.1 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