

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.....	3
1.1. Installation broker Kafka.....	3
1.1.1 Installation à partir d'une archive.....	3
1.1.2 Installation à partir de docker.....	3
1.2. Mise en place cluster Kafka.....	3
1.2.1 <i>Option archive</i>	3
1.2.2 <i>Option Docker</i>	4
1.2.3 Toutes options.....	4
Atelier 2: Producer API.....	5
Atelier 3 : Consumer API.....	6
3.1 Implémentation.....	6
3.2 Tests.....	6
Atelier 4. Sérialisation Avro.....	7
4.1 <i>Démarrage de Confluent Registry</i>	7
4.2 Producteur de message.....	7
4.3 Consommateur de message.....	8
4.4 <i>Mise à jour du schéma</i>	8
Atelier 5. Frameworks.....	9
5.1 Spring Cloud Stream.....	9
5.2 MP Messaging avec Quarkus.....	9
Atelier 6. Kafka Connect.....	10
6.1 Installations ElasticSearch + Connecteur.....	10
6.2 Configuration Kafka Standalone.....	10
Atelier 7 : KafkaStream.....	12
Atelier 8 : Fiabilité.....	13
8.1. At Least Once, At Most Once.....	13
8.2. Transaction et <i>Exactly-Once</i>	14
8.2.1 Producteur transactionnel.....	14
8.2.2 Consommateur.....	14
8.2.2 Transfert exactly Once.....	14
Atelier 9 : Administration.....	15
9.1 Reassign partitions, Retention.....	15
9.2 Rolling restart.....	15
Atelier 10 : Sécurité.....	16
10.1 Mise en place de SSL pour crypter les données.....	16
10.2 <i>Authentication via SASL/PLAIN</i>	17
10.3 <i>Autorisation</i>	17

Atelier 11 : *Monitoring*.....18

11.1 Mise en place monitoring Prometheus, Grafana.....18

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

1.2.1 Option archive

Réinitialiser les données de Zookeeper en arrêtant l'ensemble, supprimant les données dans le répertoire ***data*** et redémarrer

Créer un répertoire ***kafka-cluster*** et 3 sous-répertoires : ***broker-1***, ***broker-2***, ***broker-3***

Copier le fichier *server.properties* dans le répertoire ***kafka-cluster***

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

propriétés :

- **broker.id**
- **logs.dir**
- **listeners**

Visualiser les traces de démarrages :

```
tail -f $KAFKA_HOME/logs/server.log
```

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 AsynchroneL'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 pauseL'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 Confluent 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 seueur 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

Mettre 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 une 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 **libs** de Kafka

6.2 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 timestamp 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 Doubleton 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 résultats :

```
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 transfère 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

Atelier 10 : Sécurité

10.1 Mise en place de SSL pour crypter les données

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 3650 -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 certificat 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
```

Option Archive

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,

Option Docker

Monter le répertoire ssl sur */opt/bitnami/kafka/config/certs/kafka.truststore.jks*

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
```

10.2 Authentication via SASL/PLAIN

<https://hub.docker.com/r/bitnami/kafka/>

10.3 Autorisation

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