Cahier de TP
« Messagerie distribuée avec Kafka »

Pré-requis :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux
- JDK21+, Maven
- IDE Recommandés : STS 4, IntelliJIDEA, VSCode
- Docker, Git

Table des matières

Atelier 1: Le cluster kafka

1.1 Premier démarrage et logs

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

Il permet de démarrer un cluster 3 nœuds ainsi que 2 containers utiles pour l'administration du cluster

Démarrer cette stack avec :

```
docker compose up -d
```

Observer les logs de démarrages d'un noeud

```
docker logs -f kafka-0
```

1.2 Utilisation des utilitaires

Ouvrir un bash sur un des nœuds

```
docker exec -it kafka-0 bash
```

Se positionner dans les répertoires des utilitaires :

```
cd /opt/bitnami/kafka/bin/
```

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***

1.3 Outils graphiques

Parcourir l'UI des 2 outils graphiques fournis :

akhq : http://localhost:8080

Redpanda Console : http://localhost:9090

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

Tester l'envoi de messages dans les différents dans l'IDE

Via les commandes utilitaires ou l'interface d'admin, vérifier la création du topic et le contenu des messages

Construire un jar exécutable avec :

```
mvn package
```

Supprimer le topic et le recréer avec un nombre de *partitions=5* et un *replication-factor=3*

Envoyer 1M de messages, par exemple :

```
java -jar target/producer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 500 2000 10 1
```

Atelier 3 : Consumer API

3.0 Mise en place d'une base Postgres

Démarrer un serveur postgres et sa console d'administration
`docker compose -f postgres-docker-compose.yml up -d`

Accéder à *localhost:81* et se logger avec ***admin@admin.com/admin***
Enregistre un serveur avec comme paramètres de connexion :

- host : ***consumer-postgresql***
- user : ***postgres***
- password : ***postgres***

Créer une base consumer et y exécuter le script de création de table fourni : ***create-table.sql***

3.1 Implémentation

Le projet est composé de :

- Une classe principale ***KafkaConsumerApplication*** qui prend en arguments :
  - ***nbThreads*** : Un nombre de threads
  - ***sleep*** : Le temps de traitement d'un message (Simuler par un Thread.sleep)

  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 avec la propriété* ***ConsumerConfig.AUTO_OFFSET_RESET_CONFIG*** à « ***earliest*** »
- Fournir un Deserialiseur
- Implémenter la boucle de réception, pour chaque enregistrement reçu stocker dans la base postgres id du coursier et l'offset kafka en utilisant la classe ***ConsumerDao***

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
```

Implémenter un ***ConsumerRebalanceListener*** et l'associer au moment du *subscribe()*

3.2 Tests

Une fois le programme mis au point, construire l'application avec :

```
mvn clean package
```

effectuer plusieurs tests, entre chaque démarrage effacer les offsets du groupe et nettoyer la base de données

Tester qu'aucun message n'est perdu :
Démarrer le programme avec 3 thread

```
java -jar target/consumer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 3 10
```

Visualiser la répartition des partitions

Arrêter puis redémarrer avec la même configuration

Tester la réaffectation de partitions :
Démarrer 2 fois l'application : 1 avec 2 threads l'autre avec 3 threads

```
java -jar target/consumer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 2 10
```

```
java -jar target/consumer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 3 10
```

visualiser la répartition des partitions

Arrêter et redémarrer un processus pendant la consommation

Test redémarrage broker

Arrêter un broker :

```
docker stop kafka-0
```

Redémarrer le, Dans le répertoire du fichier docker-compose

```
docker compose up -d
```

Essayer de vérifier le traitement de tous les messages. (Il se peut qu'il y ait des doublons de traitement)

Atelier 4. Sérialisation Avro

4.1 Ajout de Confluent Schema Registry

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

Démarrer la stack

Accéder à *localhost:8081/subjects*

Visualiser dans la Redpanda Console le schema registry

4.2 Producteur de message

Créer un nouveau projet Maven ***producer-avro***
Récupérer le **pom.xml** fourni
Mettre au point un **schéma Avro : src/main/resources/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, redpanda*** puissent lire les messages.

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. Kafka Connect

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

5.1 Installations ElasticSearch

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*

5.2 Installation connecteur ElasticSearch

Télécharger une distribution de Kafka et copier toutes les librairies du répertoire ***kafka-connect-elasticsearch***[1] fourni dans le répertoire ***libs*** de la distribution kafka

5.3 Configuration du connecteur

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 Connect via la commande :

$KAFKA_HOME/bin/connect-standalone.sh <path-to-connect-standalone.properties> <path-to-elasticsearch-connect.properties>

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 Kafka

Atelier 6 : Introduction à KafkaStream

Objectifs :
Écrire une mini-application Stream qui prend en entrée le topic ***position*** et écrit en sortie dans 2 topic

- ***distance :*** Un information de distance par rapport à un point d'origine est ajouté
- ***average :*** La position moyenne d'un coursier est calculée

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

- Les champs distance et average ont 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 en commençant par le topic ***distance***

Atelier 7. Frameworks

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

7.1 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

7.2 Production et Consommation de message avec Spring Kafka

Reprendre les 2 projets SpringKafka fourni et les comprendre.

Supprimer le topic position, nettoyer la base PostgreSQL et exécuter SpringProducer et SpringConsumer

7.3 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 8 : Garanties Kafka

8.1. Transaction et ISOLATION_LEVEL

8.2.1 Producteur transactionnel

Modifier le code du producer afin d'englober plusieurs envois de messages dans une transaction.

Commiter tous les 10 envois

Lancer ensuite le programme avec comme arguments 1 **105** 100 0

Ce qui signifie que 1 threads envoie 105 messages, les 100 premiers messages font partie de 10 transactions validées. Les 5 derniers messages ne sont pas validés.

Visualiser les messages dans une console d'administration.

Noter le nombre et les offsets, Visualiser le flag transactionnel

8.2.2 Consommateur

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

8.2 Transfert Exactly Once

Modifier le consommateur afin qu'il transfert exactement une fois les messages produits en amont vers un autre topic de sortie

Éventuellement on peut effectuer un traitement du message comme calculer la distance à un point d'origine.

Vérifier les messages produits et les offsets consommés

Ateliers 9: Sécurité

9.1 Séparation des échanges réseaux

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

Visualisez la configuration bitnami des listeners

9.2 Accès via SSL au cluster

Redémarrer le cluster en utilisant la nouvelle version de docker-compose, le port EXTERNAL utilise dorénavant SSL

Récupérer le répertoire SSL qui contient un keystore et un trustore self-signed pour localhost

Modifier ***docker-compose.yml*** afin que le répertoire ***ssl/mount*** soit correctement monté sur chaque broker.

Démarrer le cluster et vérifier le certificat produit par le serveur :

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

Modifier les propriétés du client SpringProducer afin qu'il utilise le protocle SSL pour communiquer

```
spring:
  kafka:
    bootstrap-servers: localhost:19092
    ssl:
      trust-store-location:
file:///home/dthibau/Formations/Kafka/github/kafka-solutions/ssl/mount/kafka.truststore.jks
      trust-store-password: secret
```

Vérifier la production de message

9.3 Authentification avec SASL/PLAIN

9.3.1 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";
};

Récupérer la nouvelle version de docker-compose et vérifier le montage de volume vers votre fichier ***kafka_server_jaas.conf***

Redémarrer le cluster et vérifier son bon démarrage

9.3.2 Authentification client

Mettre à jour un 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 SpringProducer pour qu'il s'authentifie avec l'utilisateur alice

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

9.4 ACL

Visualiser la nouvelle configuration des brokers dans le nouveau docker-compose.yml

Démarrer le cluster.

Vérifier via Redpanda Console que le lien sécurité permet de créer des ACLs

Interdire à Alice la production de message et tester

9.5 Quotas

Se créer un ficier ***client-admin.properties*** permettant de se logger avec l'utilisateur admin

Définition d'un quota pour l'utilisateur alice

```
$KAFKA_DIST/bin/kafka-configs.sh --bootstrap-server localhost:9094 --alter --add-config 'producer_byte_rate=1024,consumer_byte_rate=1024' --entity-type users --entity-name alice --command-config client-admin-ssl.properties
```

Relancer SpringProducer et observer les messages d'erreurs produits

Supprimer les quotas :

```
$KAFKA_DIST/bin/kafka-configs.sh --bootstrap-server localhost:9094 --alter --delete-config producer_byte_rate --entity-type users --entity-name alice --command-config client-admin-ssl.properties
```

```
$KAFKA_DIST/bin/kafka-configs.sh --bootstrap-server localhost:9094 --alter --
delete-config consumer_byte_rate --entity-type users --entity-name alice --
command-config client-admin-ssl.properties
```

Atelier 10 : Annexes

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

10.1 Retention

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

10.2 Métriques JMX et 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 ***monitoring***

```
Dans un répertoire de travail monitoring
wget
https://repo1.maven.org/maven2/io/prometheus/jmx/jmx_prometheus_ja
vaagent/0.20.0/jmx_prometheus_javaagent-0.20.0.jar
wget
https://raw.githubusercontent.com/confluentinc/jmx-monitoring-
stacks/main/shared-assets/jmx-exporter/kafka_broker.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_broker.yml"
```

Visualiser le nouveau docker-compoe.yml qui intègre les services Prometheus et Grafana

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