

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

Images utilisées :

- *docker.io/bitnami/kafka:3.7*
- *tchiotludo/akhq*
- *docker.redpanda.com/redpandadata/console:latest*
- *confluentinc/cp-schema-registry*
- *postgres:15.1*
- *dpage/pgadmin4*
- *docker.elastic.co/elasticsearch/elasticsearch:7.7.1*
- *docker.elastic.co/kibana/kibana:7.7.1*

Table des matières

Atelier 1: Le cluster kafka.....	3
1.1 Premier démarrage et logs.....	3
1.2 Utilisation des utilitaires.....	3
1.3 Outils graphiques.....	3
Atelier 2: Producer API.....	4
Atelier 3 : Consumer API.....	5
3.0 Mise en place d'une base Postgres.....	5
3.1 Implémentation.....	5
3.2 Tests.....	6
Atelier 4. Sérialisation Avro.....	7
4.1 Ajout de Confluent Schema Registry.....	7
4.2 Producteur de message.....	7
4.3 Consommateur de message.....	8
4.4 Mise à jour du schéma.....	8
4.4.1 Evolution du schéma compatible.....	8
4.4.2 Evolution du schéma incompatible.....	8
Atelier 5. Kafka Connect.....	9
5.1 Installations ElasticSearch.....	9
5.2 Installation connecteur ElasticSearch.....	9
5.3 Configuration du connecteur.....	9
Atelier 6 : Garanties Kafka.....	11
6.1. Transaction et ISOLATION_LEVEL.....	11
6.1.1 Producteur transactionnel.....	11
6.1.2 Consommateur.....	11
6.2 Transfert Exactly Once.....	11
6.3 Stockage et rétention.....	11
Ouvrir un bash sur un des nœuds.....	11

Visualiser les répertoires de logs sur les brokers :.....	11
Visualiser les segments du topic position et apprécier la taille.....	11
Pour le topic position modifier le <i>segment.bytes</i> à 1Mo.....	11
Diminuer le <i>retention.bytes</i>	11
Produire de nombreux messages afin de voir des segments disparaître.....	11
Atelier 7 : Introduction à KafkaStream.....	12
7.1 Opérateurs stateless.....	12
7.2 Opérateurs stateful.....	12
Atelier 8 : ksqlDB.....	13
8.1 Getting started.....	13
Ateliers 9: Sécurité.....	15
9.1 Séparation des échanges réseaux.....	15
9.2 Accès via SSL au cluster.....	15
9.3 Authentification avec SASL/PLAIN.....	15
9.3.1 Authentification inter-broker.....	15
9.3.2 Authentification client.....	16
9.4 ACL.....	16
9.5 Quotas.....	16
Atelier 10. Frameworks.....	18
10.1 MP Messaging avec Quarkus.....	18
10.2 Production et Consommation de message avec Spring Kafka.....	18
10.3 Spring Cloud Stream.....	18
Atelier 11 : Annexes.....	19
11.1 Métriques JMX et mise en place monitoring Prometheus, Grafana.....	19

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 nœud

```
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

Accéder à la description détaillée du topic

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

Consommer les messages depuis le début

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

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 en ms
 - ***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

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
Le traitement d'un message consiste à un insérer un enregistrement en BD
- 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 Installation du plugin ElasticSearch dans kafka connect

Visualiser le fichier Dockerfile fourni et construire une image *my-kafka-connect-with-elasticsearch* via :

```
docker build -t my-kafka-connect-with-elasticsearch .
```

5.2 Démarrage de la stack

Démarrer la nouvelle stack qui ajoute l'image précédemment construite, ElasticSearch et Kibana

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

Vérifier l'installation du plugin via :

<http://localhost:8083/connector-plugins>

Se connecter à kibana localhost :5601 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

5.3 Configuration du connecteur

Définir un connecteur *elasticsearch-sink* qui déverse le topic position dans l'index position :

```
curl -X POST http://localhost:8083/connectors \
-H "Content-Type: application/json" \
-d '{
"name": "elasticsearch-sink",
"config": {
"connector.class":
```

```

"io.confluent.connect.elasticsearch.ElasticsearchSinkConnector",
"tasks.max": "1",
"topics": "position",
"topic.index.map": "position:position_index",
"connection.url": "http://elasticsearch:9200",
"type.name": "log",
"key.ignore": "true",
"schema.ignore": "true",
"key.converter": "org.apache.kafka.connect.json.JsonConverter",
"key.converter.schemas.enable": "false",
"value.converter": "org.apache.kafka.connect.json.JsonConverter",
"value.converter.schemas.enable": "false"
}
}'

```

Vous pouvez vérifier la bonne installation du connecteur :

- Via les logs de KafkaConnect
- Via [http://localhost:8083 /connectors](http://localhost:8083/connectors) : Liste des connecteurs actifs
- Via l'interface de RedPanda

Alimenter le topic position

Vous pouvez visualiser les effets du connecteur

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

5.3 Améliorations

Améliorer le fichier de configuration afin d'introduire que le timestamp Kafka soit renseigné dans le champ @timestamp de l'index position d'ElasticSearch

Déverser le topic ***position-avro*** dans un index ***position-avro***

Atelier 6 : Garanties Kafka

6.1. Transaction et ISOLATION_LEVEL

6.1.1 Producteur transactionnel

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

Committer 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

6.1.2 Consommateur

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

6.2 Transfert Exactly Once

Modifier le consommateur afin qu'il transfère 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

6.3 Stockage et rétention

Ouvrir un bash sur un des nœuds

docker exec -it kafka-0 bash

Visualiser les répertoires de logs sur les brokers :

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

Visualiser les segments du topic position et apprécier la taille

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

Diminuer le *retention.bytes*

Produire de nombreux messages afin de voir des segments disparaître

Atelier 7 : Introduction à KafkaStream

Objectifs :

Écrire une application Stream qui prend en entrée le topic *position-avro*

7.1 Opérateurs stateless

Créer un projet *PositionStream* et installer le package

io.confluent :kafka-avro-serializer

Reprendre le schéma Avro initial du producteur Avro et générer les classes du modèle

- Dans un premier temps, transformer les informations de la position d'un coursier en arrondissant la longitude et la latitude à 1 valeur entière.

Tester, regarder les timestamp des messages

Supprimer le topic de sortie

- Compléter le stream en inversant les clés et Valeurs (La position du coursier devient la clé, l'id du coursier la valeur)

Tester

- Utiliser branch pour créer 2 topic de sortie un contenant les positions dont la latitude est supérieure à 45.0 et l'autre le reste

Tester

7.2 Opérateurs stateful

Sur les branches précédentes, effectuer une agrégation de type Count(), i.e compter le nombre de coursier ayant passé à une position donné.

Tester

Modifier en utilisant une fenêtre temporelle de 1 seconde.

Utiliser l'opération reduce pour avoir en temps-réel, la liste des coursiers associés à une position.

Atelier 8 : ksqlDB

8.1 Getting started

Démarrer la stack avec le docker compose fourni.

Ensuite, démarrer une console interactive ksql-cli

```
docker exec -it ksqldb-cli ksql http://ksqldb-server:8088
```

Créer un stream associé à un topic

```
CREATE STREAM riderLocations (profileId VARCHAR, latitude DOUBLE, longitude DOUBLE) WITH (kafka_topic='locations', value_format='json', partitions=1);
```

Créer une table contenant les derniers emplacements des riderLocation

```
CREATE TABLE currentLocation AS  
SELECT profileId,  
LATEST_BY_OFFSET(latitude) AS la,  
LATEST_BY_OFFSET(longitude) AS lo  
FROM riderLocations  
GROUP BY profileId  
EMIT CHANGES;
```

Créer une table contenant les données agrégées (liste de coursier, nombre) par distance par rapport à un point d'origine

```
CREATE TABLE ridersNearMountainView AS  
SELECT ROUND(GEO_DISTANCE(la, lo, 37.4133, -122.1162), -1) AS distanceInMiles,  
COLLECT_LIST(profileId) AS riders,  
COUNT(*) AS count  
FROM currentLocation  
GROUP BY ROUND(GEO_DISTANCE(la, lo, 37.4133, -122.1162), -1);
```

Exécuter une PUSH QUERY

```
-- Mountain View lat, long: 37.4133, -122.1162
```

```
SELECT * FROM riderLocations WHERE GEO_DISTANCE(latitude, longitude, 37.4133, -122.1162) <= 5 EMIT CHANGES;
```

Démarrer une autre session :

```
docker exec -it ksqldb-cli ksql http://ksqldb-server:8088
```

```
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES ('c2309eec', 37.7877, -122.4205);
```

```
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES ('18f4ea86', 37.3903, -122.0643);
```

```
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES ('4ab5cbad', 37.3952, -122.0813);
```

```
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES ('8b6eae59', 37.3944, -122.0813);
```

```
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES ('4a7c7b41',  
37.4049, -122.0822);
```

```
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES ('4ddad000',  
37.7857, -122.4011);
```

Puis exécuter une PULL QUERY

```
SELECT * from ridersNearMountainView WHERE distanceInMiles <= 10;
```

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:19092 -tls1_2
```

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

```
ssl:
  trust-store-location: <path-to>/ssl/mount/kafka.truststore.jks
  trust-store-password: secret
security:
  protocol: SSL
```

Vérifier la production de message

9.3 Authentification avec SASL/PLAIN

9.3.1 Authentification inter-broker

Mettre au point un fichier *kafka_jaas.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_jaas.conf**

Redémarrer le cluster et vérifier son bon démarrage

9.3.2 Authentication 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 fichier **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. Frameworks

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

10.1 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 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

10.2 Production et Consommation de message avec Spring Kafka

Reprendre les 2 projets SpringKafka fournis et les comprendre.

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

10.3 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 11 : Annexes

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

11.1 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

Visualiser le nouveau ***docker-compose.yml*** qui :

- Attache l'agent `jmx_prometheus_javaagent-0.20.0.jar` aux processus Java des brokers
- Intègre et configure les services Prometheus et Grafana

Produire et consommer

Vérifier la production de métriques par chaque broker :

`http://localhost:7071`

Vérifier la récupération des métriques dans Prometheus :

`http://localhost:9091`

Connecter-vous à Grafana

`http://localhost:3000` avec `admin/admin`

Vérifier la datasource Prometheus

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

Les métriques des brokers devraient s'afficher