# Cahier de TP
# « Messagerie distribuée avec Kafka »
# C# - .NET

Prérequis :

- Bonne connexion Internet
- Système d'exploitation : Windows
- IDE Recommandés : Visual Studio
- Docker, Git
- JDK21+

Solutions :
***https://github.com/dthibau/kafkac-solutions***

Images docker 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
- confluentinc/cp-kafka-connect:7.8.0

**TABLE DES MATIERES**

# Ateliers 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
```

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

```
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

## Ateliers 2 : Producer API

Créer un nouveau projet de type ***ApplicationConsole*** avec les packages ***Confluent.Kafka***

Récupérer les sources fournis. La classe principale ***Program.cs*** qui prend en arguments :

- ***nbThreads*** : Un nombre de threads/task
- ***nbMessages*** : Un nombre de messages
- ***sendMode*** : Le mode d'envoi : 0 pour Fire_And_Forget, 1 pour Synchrone, 2 pour Asynchrone

L'application instancie *nbThreads ProducerThread* et leur demande d'envoyer *nbMessages* dans le mode d'envoi spécifié en ligne de commande.

Lorsque toutes les tâches sont terminées. Elle affiche le temps d'exécution

Le projet est constitué des classes suivantes :

- Une classe ***ProducerThread*** que vous devez implémenter en particulier la construction du producteur Kafka et les méthodes d'envoi de messages.
  Les messages sont constitués d'une clé au format string (*courier.id*) et d'une valeur au format JSON (classe *Position)*
- Un sérialiseur Custom capable de sérialiser en Json n'importe quelle classe.
- 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

Implémenter la classe *ProducerThread* en implémentant les 3 modes d'envoi.

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

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

Construire un exécutable

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

Envoyer 100 000 de messages, par exemple :
***bin/Debug/KafkaProducer.exe 100 1000 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*

Créer un nouveau projet avec les packages ***Confluent.Kafka*** et ***Npgsql***

Récupérer le code fourni

Le projet est composé de :

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

L'application instancie *nbThreads KafkaConsumerThread* et leur demande de consommer les messages sous le même nombre de groupe. Le programme s'arrête avec un Ctrl+C.

- ***KafkaConsumerThread*** lors de la réception de message insère en base dans la table coursier l'id du coursier et l'offset du message
- 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 tester la réception, vous pouvez utiliser le programme précédent et le lancer afin qu'il exécute de nombreux messages.

Positionner des handler écoutant la réaffecation de partitions

## *3.2 Tests*

Une fois le programme mis au point, construire

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

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

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. Schema Registry et sérialisation Avro

## *4.1 Ajout de Confluent Schema Registry* et outils Avro

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

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 ***KafkaProducerAvro*** avec les packages ***Confluent.Kafka, Confluent.SchemaRegistry.Serdes.Avro***,

Mettre au point un **schéma Avro : Coursier.avsc**

Installer globalement le package ***Apache.Avro.Tools*** :

```
dotnet tool install --global Apache.Avro.Tools
```

Dans le répertoire du fichier Schema, exécuter :

```
avrogen -s Coursier.avsc .
```

Reprendre les classes du projet ***producer*** sans les classes du modèle ni le serialiseur

Dans la classe principale, enregistrer le schéma dans le serveur registry :

Dans le producteur de message modifier la classe *KafkaProducerThread* afin

- Qu'elle compile
- qu'elle utilise un sérialiseur de valeur de type **AvroSerializer**

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

Accéder à http://localhost:8081/subjects

Puis à Accéder à http://localhost:8081/schemas/

Vérifier que ***akhq, redpanda*** puissent lire les messages.

## *4.3 Consommateur de message*

Créer un nouveau projet *KafkaConsumerAvro* avec les packages ***Confluent.Kafka, Confluent.SchemaRegistry.Serdes.Avro***,

Reprendre les sources du projet *KafkaConsumer* sans les classes du modèle ni le désérialiseur

Modifier le code fin d'utiliser la classe ***GenericRecord***
Modifier le désérialiseur de la thread de consommation :

## *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 champ 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*** : 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 Producer***

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** 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***

Vérifier que le consommateur ne consomme que 100 messages

## *6.2 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

# Atelier 7 : Streamiz

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
***Streamiz.Kafka.Net.SchemaRegistrySerdesAvro***

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

```
avrogen -s Coursier.avsc .
```

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.

Nous voulons dorénavant avoir en temps-réel, la liste des coursiers associés à une position.

Voici une proposition de solution mais vous pouvez implémenter votre propre logique.

- Utilisation d'un Dictionary pour stocker les positions précédentes d'un coursier,
- Fonction d'agrégation construisant une liste de coursier par position
- *FlatMap* qui détecte les changements de position d'un coursier pour l'enlever de la liste de son ancienne 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

## *9.0 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

## *9.1 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 Producer afin qu'il utilise le protocle SSL pour communiquer

```
SecurityProtocol = SecurityProtocol.Ssl,
SslCaLocation = "/path/to/ca-cert.pem",
```

Vérifier la production de message

## *9.2 Authentification avec SASL/PLAIN*

### *9.2.1 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

### 9.2.2 Authentification client

Configurer le projet Producer pour qu'il s'authentifie avec l'utilisateur alice

```
SecurityProtocol = SecurityProtocol.SaslSsl, // Protocole sécurisé SASL_SSL
SaslMechanism = SaslMechanism.Plain, // Mécanisme d'authentification PLAIN
SaslUsername = "alice", // Nom d'utilisateur
SaslPassword = "alice-secret", // Mot de passe
SslCaLocation = "<path-to>\\ca-cert.pem",
```

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

## 9.3 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

## 9.4 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 le Producer 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

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