# Lab4: Apache kafka

L'objectif de ce TP est de :

- Installation d'apache kafka
- Première utilisation d'apache Kafka
- création d'une application word count kafka

**Apache Kafka** est un système de messagerie distribué basé sur le pattern publish /subscribe. Il combine trois fonctionnalités :

- Publier et s'abonner à des flux d'événements en important/exportant des données en continue depuis d'autres systèmes.
- Stocker des flux d'événements de manière durable et fiable aussi longtemps que vous le souhaitez.
- Traiter des flux d'événements au fur et à mesure qu'ils se produisent ou rétrospectivement

**Zookeeper** est un service centralisé permettant de maintenir l'information de configuration, de nommage, de synchronisation et de services de groupe. Ces services sont utilisés par les applications distribuées en général, et par Kafka en particulier

## 1. Installation kafka

- Kafka a été installé sur le cluster créé dans le TP précédent.
- Démarrer vos conteneurs en utilisant la commande suivante :

```
docker start hadoop-master hadoop-slave1 hadoop-slave2
```

- Accéder maintenant au conteneur master:

docker exec -it hadoop-master bash

Lancer ensuite les démons yarn et hdfs:

```
./start-hadoop.sh
```

- Lancer Kafka et Zookeeper en tapant :

```
./start-kafka-zookeeper.sh
```

vérifier que les deamons zookeeper et hadoop sont démarrés en utilisant la commandes

```
jps
```

## 2. Première utilisation d'apache Kafka

### a) Création d'un topic

Pour gérer les topics, Kafka fournit une commande appelée kafka-topics.sh.

- Dans un nouveau terminal du master, créer un nouveau topic appelé "Hello-Kafka" en utilisant la commande suivante

```
kafka-topics.sh --create --bootstrap-server localhost:9092 \
--replication-factor 1 --partitions 1 \
--topic Hello-Kafka
```

- Pour afficher la liste des topics existants, il faudra utiliser:

```
kafka-topics.sh --list --bootstrap-server localhost:9092
```

**Remarque** : Depuis Kafka 2.x, il est recommandé d'utiliser --bootstrap-server et non --zookeeper.

### b) description d'un topic

montrer des détails tels que le nombre de partitions du nouveau sujet

```
kafka-topics.sh --describe --topic Hello-Kafka –bootstrap-server \ localhost:9092
```

### c) Ecrire des évènements dans un topic

Exécutez le client **producer** de la console pour écrire quelques événements dans le topic créé. Par défaut, chaque ligne saisie entraînera l'écriture d'un événement distinct dans le topic.

```
kafka-console-producer.sh --bootstrap-server localhost:9092 --topic Hello-Kafka
```

**remarque** : ctl+c pour quitter la saisie

### d) Lire des événements

Ouvrir une autre session de terminal et exécuter le client consommateur de la console pour

```
kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic Hello-Kafka --
from-beginning
```

## 3. Création d'une application kafka

L'objectif de cette partie est de créer une application pour publier et consommer des événements de Kafka. Pour cela, nous allons utiliser les API KafkaProducer et KafkaConsumer .

- **Création du Producer**

Créer un producteur Kafka nommé EventProducer

```
import java.util.Properties;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

public class EventProducer {

  public static void main(String[] args) throws Exception{


   // Verifier que le topic est fourni comme arg
     if(args.length == 0){
        System.out.println("Entrer le nom du topic");
        return;
     }

     String topicName = args[0].toString(); // lire le topicName fourni comme param
     /
     Properties props = new Properties(); / acceder aux configurations du producteur

     props.put("bootstrap.servers", "localhost:9092"); // spécifierle serveur kafka

     // Definir un acquittement pour les requetes du producteur
     props.put("acks", "all");

     // Si la requete echoue, le producteur peut reessayer automatiquement
     props.put("retries", 0);

// Specifier la taille du buffer size dans la config
props.put("batch.size", 16384);
```

```
      // controle l espace total de mem dispo au producteur pour le buffering
      props.put("buffer.memory", 33554432);

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer","org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<String,String>(props);

for(int i = 0; i < 10; i++)
  producer.send(new ProducerRecord<String, String>(topicName,
            Integer.toString(i), Integer.toString(i)));
  System.out.println("Message envoye avec succes");
  producer.close();
   }
 }
```

*ProducerRecord* est une couple <cle,valeur> envoyé au cluster Kafka. Il possède plusieurs constructeurs.

- Créer un projet Maven (no archtype) dans VSCode (ajouter les extensions nécessiares ***Maven for Java*** *et* ***Extension Pack for Java)***
  - choisir **no archetype / groupId :** *edu.supmti.kafka* / **artifactId kafka**_*lab*
  - ajouter le projet dans le répertoire ***BigdataLabs*** créé précédemment
  - ajouter les dépendances au fichier **pom.xml**

```
<groupId>edu.supmti.kafka</groupId>
<artifactId>lab_kafka</artifactId>
<version>1.0-SNAPSHOT</version>
<properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <kafka.version>3.5.1</kafka.version>
</properties>
<dependencies>
        <dependency>
                <groupId>org.apache.kafka</groupId>
                <artifactId>kafka-clients</artifactId>
                <version>${kafka.version}</version>
        </dependency>
        <dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-api</artifactId>
                <version>1.7.36</version>
        </dependency>
        <dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-simple</artifactId>
                <version>1.7.36</version>
        </dependency>
</dependencies>
```

```
    <build>
    <plugins>
        <!-- Compiler plugin -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.11.0</version>
            <configuration>
                <source>${maven.compiler.source}</source>
                <target>${maven.compiler.target}</target>
            </configuration>
        </plugin>
        <!-- Assembly plugin for fat JAR -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>3.6.0</version>
            <configuration>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
                <archive>
                <manifest>
                    <mainClass>edu.supmti.kafka.EventConsumer</mainClass>
                </manifest>
                </archive>
                <finalName>kafka-consumer-app</finalName>
            </configuration>
            <executions>
                <execution>
                    <id>make-assembly</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
</project>
```

- créer le jar du producer

```
mvn clean package
```

- Copier le jar créé dans le dossier hadoop_project/kafka et sur la console du hadoop-master, lancer le jar en spécifiant le nom du topic

```
java -jar /shared_volume/kafka/kafka-producer-app-jar-with-dependencies.jar
Hello-Kafka
```

- Pour voir le résultat saisi dans Kafka, utiliser le consommateur prédéfini de Kafka:

```
  kafka-console-consumer.sh --zookeeper localhost:2181 --topic Hello-Kafka --from-
beginning
```

- **Création du consumer**

Ecrire la classe EventConsumer qui permettra de lire les enregistrements envoyés précédemment

```
import java.util.Properties;
import java.util.Arrays;
import java.time.Duration;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.ConsumerRecord;

public class EventConsumer {
  public static void main(String[] args) throws Exception {
    if(args.length == 0){
       System.out.println("Entrer le nom du topic");
       return;
    }
    String topicName = args[0].toString();
    Properties props = new Properties();

    props.put("bootstrap.servers", "localhost:9092");
    props.put("group.id", "test");
    props.put("enable.auto.commit", "true");
    props.put("auto.commit.interval.ms", "1000");
    props.put("session.timeout.ms", "30000");
    props.put("key.deserializer",
       "org.apache.kafka.common.serialization.StringDeserializer");
    props.put("value.deserializer",
       "org.apache.kafka.common.serialization.StringDeserializer");

    KafkaConsumer<String, String> consumer = new KafkaConsumer
       <String, String>(props);

    // souscription du consumer a la liste de topics
    consumer.subscribe(Arrays.asList(topicName));

    // Afficher le nom du topic
    System.out.println("Souscris au topic " + topicName);
    int i = 0;

    while (true) {
       ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));
       for (ConsumerRecord<String, String> record : records)

       // Afficher l'offset, clef et valeur des enregistrements du consommateur
       System.out.printf("offset = %d, key = %s, value = %s\n",
          record.offset(), record.key(), record.value());
    }
  }
}
```

- créer le jar du consumer de la même manière (n'oublier pas de modifier le pom.xml)

```
mvn clean package
```

- Copier le jar créé dans le dossier hadoop_project/kafka et sur la console du hadoop-master, lancer le jar en spécifiant le nom du topic

```
java -jar /shared_volume/kafka/kafka-consumer-app-jar-with-dependencies.jar
Hello-Kafka
```

## 4. Ingestion des données d'une source (fichier) vers une destination(sink) HDFS avec Kafka Connect

Dans cette section nous allons utiliser Kafka Connect avec des connecteurs simples qui importent des données d'un fichier vers une topic Kafka et exportent des données d'un topic Kafka vers un fichier.


- Tout d'abord, modifier le fichier `config/connect-standalone.properties` pour ajouter les connecteurs nécessaires

```
echo "plugin.path=/usr/local/kafka/libs/" >> $KAFKA_HOME/config/connect-
standalone.properties
```

- Créer les fichiers de configuration des connecteurs

Par défaut des fichiers exemples existent dans `$KAFKA_HOME/config/`

- Fichier de configuration source : *`connect-file-source.properties`*

```
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=/tmp/test-source.txt
topic=connect-topic
```

- Fichier de configuration destination : *`connect-file-sink.properties`*

```
name=local-file-sink
connector.class=FileStreamSink
tasks.max=1
file=/tmp/test-sink.txt
topics=connect-topic
```

- Créer un nouveau topic kafka nommé ***connect-topic***
- Créer le fichier source *test-source.txt*

```
echo "Bonjour Kafka" > /tmp/test-source.txt
```

```
echo "Bienvenue dans le monde du streaming" >> /tmp/test-source.txt
```

- Démarrer Kafka Connect en mode standalone. Pour ce faire, nous allons lancer les deux connecteurs en même temps

```
$KAFKA_HOME/bin/connect-standalone.sh \
$KAFKA_HOME/config/connect-standalone.properties \
$KAFKA_HOME/config/connect-file-source.properties \
$KAFKA_HOME/config/connect-file-sink.properties
```

- une fois le processus kafka connect lancé, la pipeline suivante est exécutée


- visualiser le contenu du test-sink.txt

```
more /tmp/test-sink.txt
```

- Ajouter des données au fichier et les voir se déplacer dans le pipeline :

```
echo "Exercice Kafka Connect simple" >> /tmp/test-source.txt
```

## 5. Application Word Count avec Kafka Streams

On veut développer une application **Kafka Streams** qui lit des phrases depuis un **topic Kafka (input-topic)**, compte la fréquence des mots, puis envoie les résultats vers un **autre topic(output-topic)**.
Pour ce faire, on utilisera le même projet créé précdemment. Ajouter la dépendence au fichier pom.xml

```
<dependency>
<groupId>org.apache.kafka</groupId>
<artifactId>kafka-streams</artifactId>
<version>${kafka.version}</version>
</dependency>
```

- Créez un fichier WordCountApp.java

```
package edu.supmti.kafka;

import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.*;
import org.apache.kafka.streams.kstream.*;
import java.util.*;

public class WordCountApp {

public static void main(String[] args) {
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "word-count-app");
```

```
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());

StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> textLines = builder.stream(args[0], Consumed.with(Serdes.String(), Serdes.String()));
// Word count logic goes here
textLines
        .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
        .groupBy((key, word) -> word)
        .count(Materialized.as("word-counts-store"))
        .toStream()
        .mapValues(count -> Long.toString(count))
        .to(args[1], Produced.with(Serdes.String(), Serdes.String()));
// End of word count logic
KafkaStreams streams = new KafkaStreams(builder.build(), props);
streams.start();
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}
}
```

- Créer le jar correspondant et copier le dans hadoop_project/kafka. N'oublier pas de modifier la classe principale dans le pom.xml
- Créer les topics input-topic et output-topic (kafka-topics.sh)
- Lancer le jar en spécifiant les noms des topics

```
java -jar /shared_volume/kafka/kafka-wordcount-app-jar-with-dependencies.jar
input-topic output-topic
```

- Ouvrir le terminal et saisir du texte dans le input-topic (kafka-console-producer.sh)
- Ouvrir le terminal et lire les messages du topic output-topic (kafka-console-consumer.sh)

```
kafka-console-consumer.sh --topic output-topic --from-beginning \
--bootstrap-server localhost:9092 --property print.key=true \
```

## 6. Mise en place d'un cluster Kafka à deux brokers et application WordCount interactive

L'objectif est de Comprendre la configuration d'un cluster Kafka multi-serveurs, la réplication des topics, et développer une application WordCount en Java qui lit les mots saisis au clavier, les envoie à Kafka, puis compte leur fréquence en temps réel.

### Partie 1 : Configuration de plusieurs brokers

Dans ce qui précède, nous avons configuré Kafka pour lancer un seul broker. Pour créer plusieurs brokers, il suffit de dupliquer le fichier **$KAFKA_HOME/config/server.properties** autant de fois que nécessaire.

- créer deux autre fichiers: server-one.properties et server-two.properties, et modifier les paramètres suivants comme suit:

```
### config/server-one.properties
broker.id = 1 listeners=PLAINTEXT://localhost:9093
log.dirs=/tmp/kafka-logs-1
### config/server-two.properties
broker.id = 2 listeners=PLAINTEXT://localhost:9094
log.dirs=/tmp/kafka-logs-2
```

- Démarrer les deux nouveaux brokers en appelant **$KAFKA_HOME/bin/kafka-server-start.sh** avec les nouveaux fichiers de configuration.
- Créer un topic «**WordCount-Topic**» répliqué sur les deux brokers
- Vérifier la configuration du topic

### Partie 2 : Création de l'application word count

- Développer deux classes Java :
  - **WordProducer** : lit du texte depuis le clavier et envoie chaque mot dans Kafka.
  - **WordCountConsumer** : lit les messages du topic et affiche la fréquence de chaque mot en temps réel.
- Créer un jar emballant les deux classes
- Exécuter le producteur dans un terminal
- Exécuter le consommateur dans un autre terminal

### Partie 3 : Kafka-ui

Ajouter un container Kafka-ui pour visualiser via une interface web les information du cluster (docker-compose.yml)

```
# ================================
  kafka-ui:
    image: provectuslabs/kafka-ui:latest
    container_name: kafka-ui
    hostname: kafka-ui
    networks:
      - hadoop
    ports:
      - 8081:8080
    environment:
      - KAFKA_CLUSTERS_0_NAME=local
      - KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS=hadoop-master:9092
      - KAFKA_CLUSTERS_0_ZOOKEEPER=hadoop-master:2181
networks:
  hadoop:
    driver: bridge
```