



Introduction à Apache Kafka

TP : Développement d'une application avec Java et Kafka

Architecture des composants d'entreprise

Table des matières

I.	Objectif du TP.....	2
II.	Prérequis	2
III.	L'architecture d'Apache Kafka	2
a.	C'est quoi Kafka ?.....	2
b.	Les cas d'utilisation.....	2
c.	L'architecture de Kafka.....	3
d.	Les 5 principales API d'Apache KAFKA.....	8
e.	Cas d'utilisation.....	8
IV.	Préparation de l'environnement Kafka	9
a.	Sans Docker.....	9
b.	Avec Docker	10
V.	Client Kafka avec Kafka CLI.....	10
a.	Créer une session Bash interactive	10
b.	Créer un Topic.....	10
c.	Envoyer un message au Topic	10
d.	Consommer les messages du Topic.....	11
VI.	Développer une application avec Java et Kafka	11
	Conclusion	Erreur ! Signet non défini.

I. Objectif du TP

- ✓ Expliquer l'architecture d'**Apache Kafka**.
- ✓ Comment démarrer Kafka avec ou sans Docker.
- ✓ Comment démarrer Kafka avec **Kraft** (à partir de Kafka version 2.8.0).
- ✓ Comment démarrer Kafka avec **Zookeeper**.
- ✓ Comment interagir avec Kafka moyennant **Kafka CLI**.
- ✓ Développer une application avec Java pour publier et consommer des messages avec Kafka.

II. Prérequis

- IntelliJ IDEA ;
- JDK version 17 ;
- Une connexion Internet pour permettre à Maven de télécharger les librairies.

NB : Ce TP a été réalisé avec IntelliJ IDEA 2023.2.3 (Ultimate Edition).

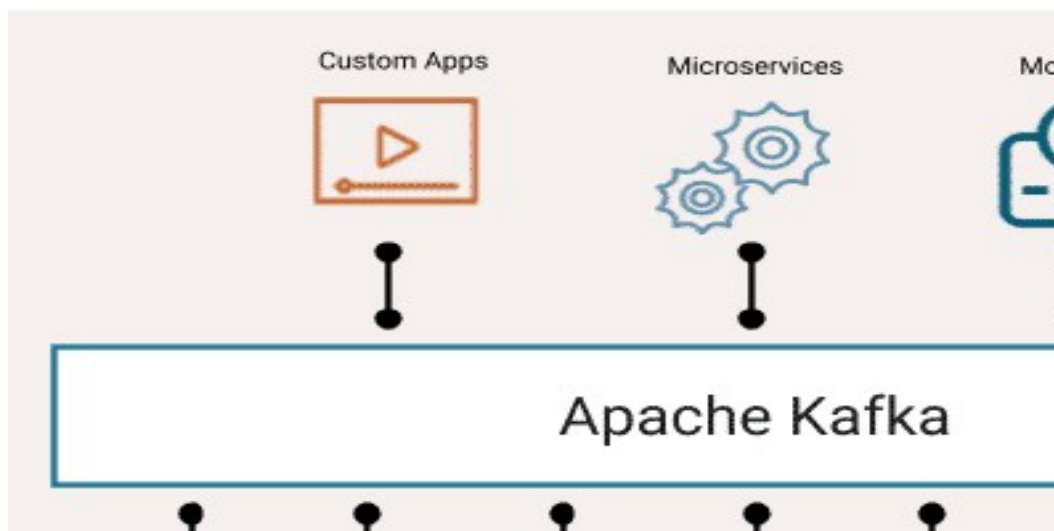
III. L'architecture d'Apache Kafka

a. C'est quoi Kafka ?

Apache Kafka est une plateforme de messagerie distribuée conçue pour gérer **des flux de données en temps réel (Data Streaming)**. Elle permet de publier, stocker et consommer des données sous forme de flux de messages. Kafka est souvent utilisé pour construire des pipelines de données et des systèmes de traitement de données en temps réel.

Kafka est capable de monter en charge (scalabilité horizontale), de supporter des débits de données très importants et d'assurer la persistance des données pendant une période déterminée.

b. Les cas d'utilisation

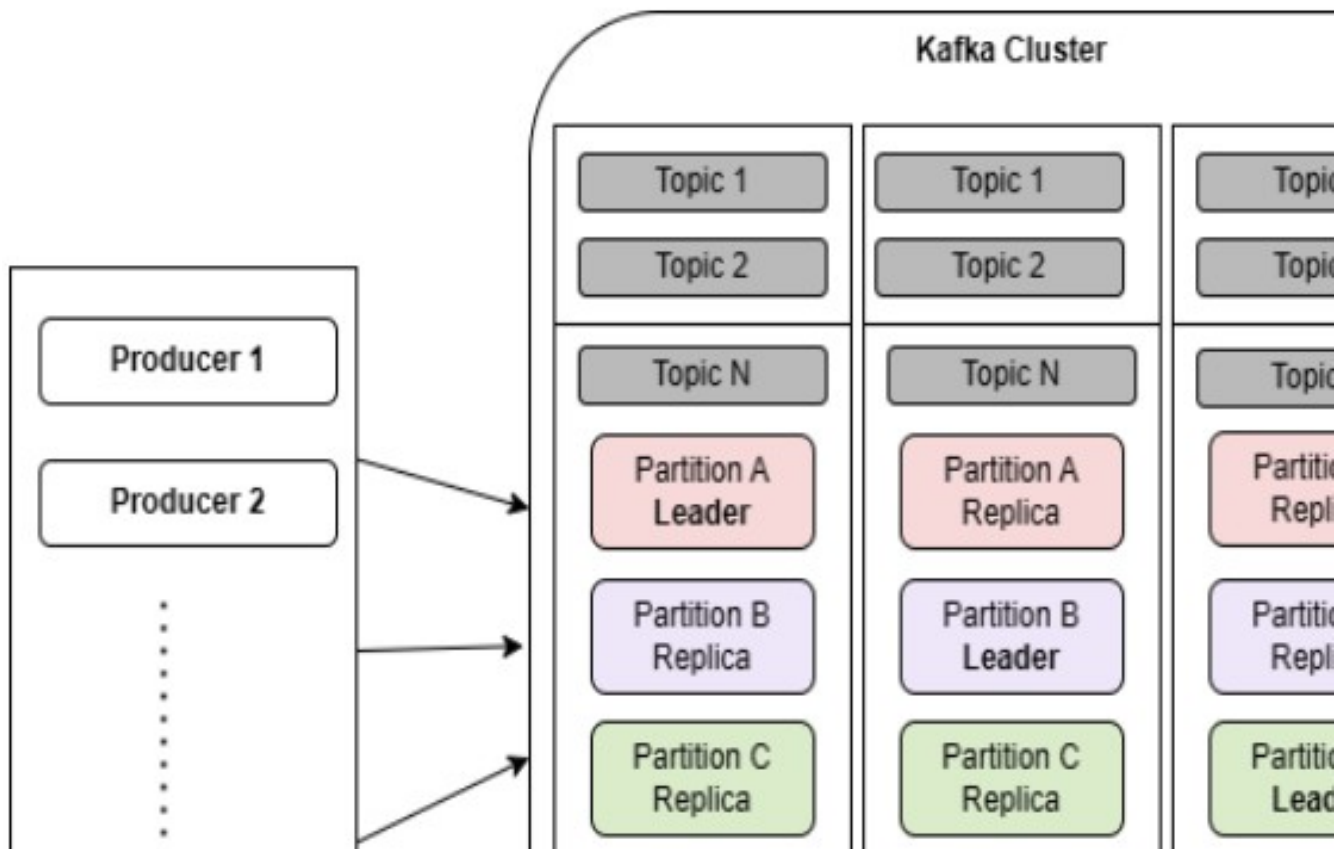


Kafka peut être utilisé dans les cas suivants :

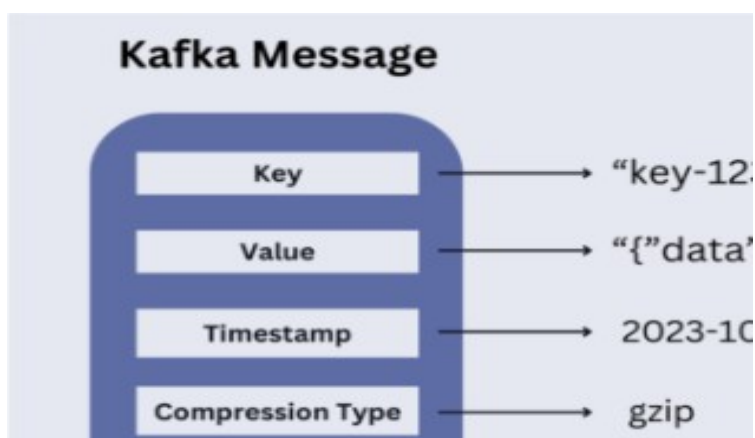
- **Streaming en temps réel** : Analyse des flux de données provenant de capteurs IoT, de réseaux sociaux ou de systèmes financiers.
- **Intégration de données** : Collecte et synchronisation des données entre différents systèmes, comme les bases de données ou les microservices.

- **Analyse des journaux** : Centralisation et traitement des journaux des applications.
- **Notifications en temps réel** : Systèmes d'alertes et de messagerie.

c. L'architecture de Kafka



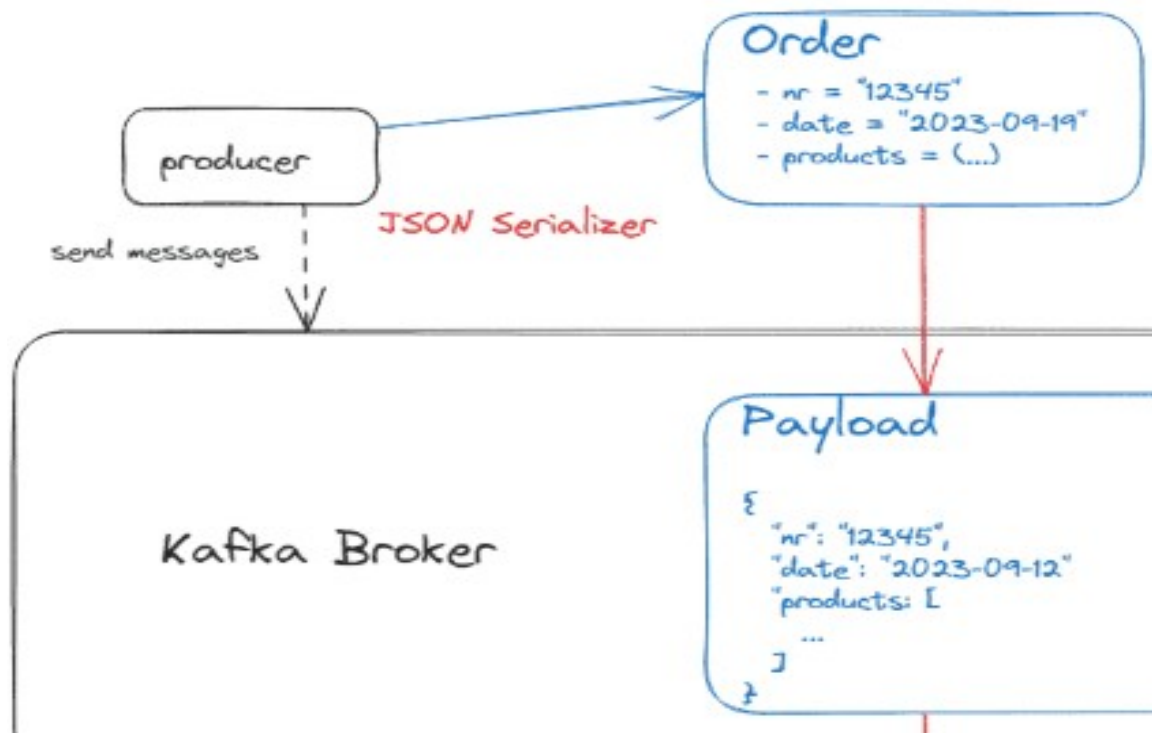
- **Les messages :**



Les messages sont organisés dans des catégories appelées « Topics », qui se constituent de partitions ordonnées. Chaque message au sein d'une partition reçoit un identifiant incrémentiel. Généralement chaque partition a plusieurs répliques sur un ou plusieurs brokers Kafka.

- ✓ **Key (Clé de message, facultatif)** : les clés sont couramment utilisées lors de l'ordre ou du regroupement de données associées. Par exemple, dans un système de traitement des journaux, nous pouvons utiliser l'ID utilisateur comme clé pour nous assurer que tous les messages de journal d'un utilisateur spécifique sont traités dans l'ordre dans lequel ils ont été générés.
- ✓ **Value** : Il contient les données réelles que nous voulons transmettre. Kafka n'interprète pas le contenu de la valeur. Il est reçu et envoyé tel quel. Il peut s'agir de XML, JSON, String ou n'importe quoi. De nombreux développeurs Kafka préfèrent utiliser **Apache Avro** (<https://howtodoinjava.com/kafka/kafka-with-avro-and-schema-registry>), un cadre de sérialisation initialement développé pour Hadoop.
- ✓ **Timestamp (Horodatage, facultatif)** : Nous pouvons inclure un horodatage facultatif dans un message indiquant son horodatage de création. Il est utile pour suivre le moment où les événements se sont produits, en particulier dans les scénarios où le temps de l'événement est important.
- ✓ **Compression Type (Type de compression, facultatif)** : les messages Kafka sont généralement de petite taille et envoyés dans un format de données standard, tel que **JSON**, **Avro** ou **Protobuf**. De plus, nous pouvons les compresser davantage dans d'autres formats (par exemple : gzip, lz4, snappy, zstd).
- ✓ **Headers (en-têtes, facultatif)** : Kafka permet d'ajouter des en-têtes qui peuvent contenir des métadonnées supplémentaires liées au message.
- ✓ **Offset** : une fois qu'un message est envoyé dans une rubrique Kafka, il reçoit également un numéro de partition et un ID Offset stockés dans le message.

Le schéma suivant montre les opérations de sérialisation et désérialisation du message :



▪ Les producteurs :

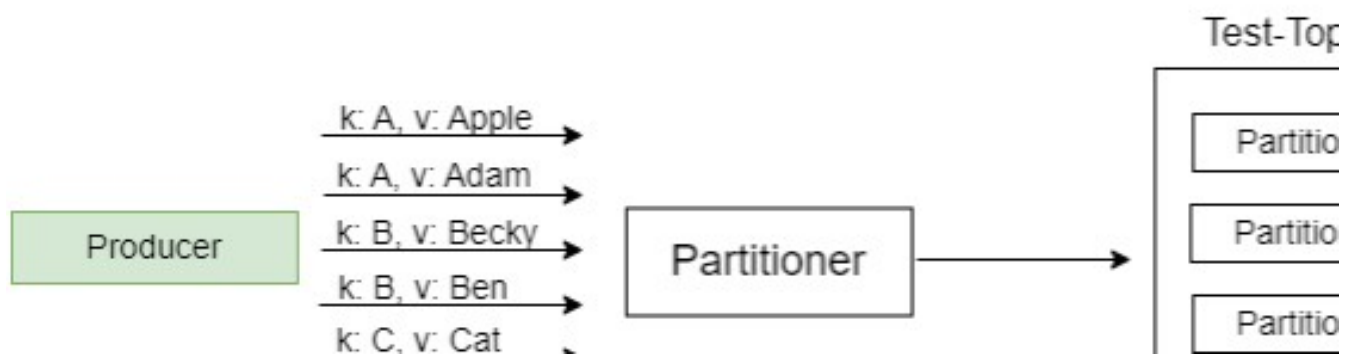
Les producteurs transmettent, via des messages, les données aux Brokers. Lorsque le nouveau broker est démarré, tous les producteurs le recherchent et lui envoient automatiquement un message. Dans ce système, le producteur n'attend pas les « accusés de réception » du broker et envoie les messages aussi rapidement que le broker peut les gérer.

▪ Les Brokers :

Le cluster Kafka se compose généralement de plusieurs brokers (nœuds du cluster) pour maintenir l'équilibre de charge (Load Balancing). Ces brokers sont sans état (Stateless), ils utilisent donc Zookeeper ou KRAFT afin de maintenir l'état de leur cluster. Une instance de broker Kafka peut gérer des centaines de milliers de lectures et d'écritures par seconde, et chaque broker peut gérer des TO de messages sans impacter les performances.

▪ Les partitions :

Un **Topic** dans Kafka est une catégorie ou un flux de messages. Ce topic est divisé en une ou plusieurs partitions. Chaque **partition** est une séquence ordonnée et immuable de messages, identifiée par un numéro (offset). Les partitions sont stockées de manière indépendante sur les différents **brokers**.



Vu que la clé de message est facultative, donc si aucune clé n'est fournie, Kafka partitionnera les données de manière circulaire (round robin) :



▪ Le Zookeeper :

Le logiciel **Zookeeper** est utilisé pour gérer et coordonner les brokers Kafka. Il élit le broker leader du cluster et informe le producteur et le consommateur de la présence de tout nouveau broker ainsi que de la défaillance d'un broker dans le système Kafka. C'est après avoir reçu la notification par le ZooKeeper, que le producteur et le consommateur commencent à coordonner leur tâche avec un autre broker.

Apache Kafka utilisait **ZooKeeper** comme service de coordination pour gérer le stockage des métadonnées du cluster, comme les informations sur les topics, les partitions et les offsets.

ZooKeeper n'est plus un prérequis à partir de Kafka 2.8.0, grâce à l'introduction de KRaft.

Kafka 3.3.0 et versions ultérieures marquent une étape importante dans la transition complète vers un Kafka sans ZooKeeper.

KRaft utilise le protocole Raft pour gérer les métadonnées directement dans le cluster Kafka, éliminant ainsi le besoin de ZooKeeper.

❖ Les consommateurs :

- **Groupes de Consommateurs (Consumer Groups)**
 - ✓ Les consumers appartiennent à un **consumer group** identifié par un identifiant unique (**group.id**).
 - ✓ Un consumer group permet à plusieurs consumers de collaborer pour consommer des messages d'un topic.
 - ✓ **Règle de répartition** : Kafka garantit que chaque partition d'un topic est lue par un seul consumer dans un consumer group. Cela permet un traitement parallèle sans duplication.
- **Attribution des Partitions (Partition Assignment)**
 - ✓ Kafka utilise des algorithmes d'attribution pour distribuer les partitions entre les consumers d'un consumer group.
 - ✓ Chaque partition est assignée à un seul consumer dans le groupe.
 - ✓ Si un consumer quitte le groupe ou si un nouveau consumer rejoint le groupe, un **rééquilibrage (rebalance)** est déclenché pour réattribuer les partitions.
- **Lecture des Offsets**
 - ✓ Chaque consumer suit sa position dans la partition via un **offset**.
 - ✓ Kafka stocke les offsets dans un topic spécial nommé **__consumer_offsets**.
 - ✓ Le consumer peut :
 - **Auto-commiter** les offsets (configuration `enable.auto.commit=true`).
 - **Manuellement** gérer les commits (`enable.auto.commit=false`) pour plus de contrôle.
- **Traitement des Messages**
 - ✓ Un message lu par un consumer peut être :
 - **Traitement réussi** : Le consumer doit alors commiter l'offset pour indiquer que le message a été traité.
 - **Traitement échoué** : Le consumer peut réessayer de traiter le message en relisant l'offset.
- **Ordre des Messages**
 - ✓ Kafka garantit que les messages dans une partition spécifique sont livrés dans l'ordre où ils ont été produits.
 - ✓ Si plusieurs consumers appartiennent au même consumer group, l'ordre global des messages peut ne pas être préservé car les partitions sont consommées indépendamment.
- **Tolérance aux Pannes**
 - ✓ Si un consumer d'un groupe échoue ou devient inactif, Kafka réattribue automatiquement ses partitions à d'autres consumers actifs du groupe.
- **Durée de Rétention des Messages**
 - ✓ Kafka ne supprime pas immédiatement les messages après qu'ils ont été consommés.
 - ✓ Les messages restent dans les partitions selon la **durée de rétention configurée** (`log.retention.hours`), permettant à d'autres consumers de relire les données si nécessaire.
- **Consommation Parallèle**
 - ✓ Pour maximiser les performances :

- Le nombre de partitions doit être **supérieur ou égal** au nombre de consumers dans un groupe.
- Si le nombre de consumers dépasse le nombre de partitions, certains consumers resteront inactifs.

d. Les 5 principales API d'Apache KAFKA

Pour offrir aux applications un accès à Apache Kafka, le logiciel propose 5 principales API, à savoir :

- ✓ **L'API Producer** permet aux applications d'envoyer des flux de données aux rubriques du cluster Kafka.
- ✓ **L'API Consumer** permet aux applications de lire des flux de données à partir de rubriques dans le cluster Kafka.
- ✓ **L'API Streams** permet de transformer des flux de données de rubriques d'entrée en rubriques de sortie.
- ✓ **L'API Connect** permet d'implémenter des connecteurs qui récupèrent continuellement des données d'un système ou d'une application source vers Kafka ou de Kafka vers un système ou une application réceptrice.
- ✓ **L'API Admin** permet de gérer et d'inspecter les brokers et d'autres objets Kafka.

e. Cas d'utilisation

- **Monitoring en temps réel** : Analyser les logs d'une application pour détecter des anomalies.
- **Traitement d'événements** : Calculer des agrégats en temps réel, comme le nombre total de transactions par minute.
- **Systèmes de recommandation** : Générer des recommandations personnalisées en temps réel en fonction du comportement de l'utilisateur.
- **ETL en temps réel** : Extraire des données de Kafka, les transformer et les charger dans un autre système.
- **Alertes** : Déclencher des notifications en temps réel en fonction d'événements spécifiques (exemple : détection de fraudes).

IV. Préparation de l'environnement Kafka

Dans ce qui suit, vous pouvez choisir la solution qui vous convient pour démarrer Kafka :

- ✓ Démarrer Kafka avec Kraft sans Docker.
- ✓ Démarrer Kafka avec Zookeeper sans Docker.
- ✓ Démarrer Kafka avec Docker.

a. Sans Docker

- Télécharger la dernière version de Kafka. Dans cet atelier, nous allons utiliser **la version 3.9.0**. Voici le lien de téléchargement :

https://www.apache.org/dyn/closer.cgi?path=/kafka/3.9.0/kafka_2.13-3.9.0.tgz

- Décompresser le fichier :

```
$ tar -xzf kafka_2.13-3.9.0.tgz
$ cd kafka_2.13-3.9.0
```

Configurer Kafka avec Kraft

- Créer un identifiant du cluster Kafka :

```
$ cd kafka_2.13-3.9.0
$ KAFKA_CLUSTER_ID="$(bin/kafka-storage.sh random-uuid)"
```

- Initialiser le cluster Kafka :

```
$ bin/kafka-storage.sh format --standalone -t $KAFKA_CLUSTER_ID -c config/kraft/reconfig-server.properties
```

- Démarrer Kafka :

```
$ bin/kafka-server-start.sh config/kraft/reconfig-server.properties
```

Configurer Kafka avec Zookeeper

- Commencer par démarrer Zookeeper :

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

- Ouvrir un nouveau terminal et lancer la commande suivante pour démarrer Kafka :

```
$ bin/kafka-server-start.sh config/server.properties
```

b. Avec Docker

Lancer la commande suivante pour démarrer le conteneur Kafka :

```
$ docker run -p 9092:9092 apache/kafka:3.9.0
```

V. Client Kafka avec Kafka CLI

a. Créer une session Bash interactive

- Commencer par démarrer Kafka (par exemple utiliser Docker comme expliqué ci-dessus).
- Lancer la commande suivante pour identifier le nom du conteneur Kafka :

```
docker ps
```

- Ouvre une session Bash interactive à l'intérieur du conteneur Docker

```
docker exec -it [A REMPLACER PAR LE VOTRE] /bin/bash
```

b. Créer un Topic

- Dans la session Bash, lancer les commandes suivantes pour créer le Topic « my-first-topic » :

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

```
sh kafka-topics.sh --bootstrap-server localhost:9092 --create --topic my-first-topic --partitions 1 --replication-factor 1
```



- Cette commande permet de créer le Topic «my-first-topic» au niveau du cluster Kafka démarré au port 9092 avec une seule partition et sans réplication des données.
- Il est recommandé d'avoir plusieurs partitions (pour le parallélisme) et un facteur de réplication supérieur à 1 (pour la tolérance aux pannes). Par exemple :

```
sh kafka-topics.sh --bootstrap-server localhost:9092 --create --topic important-topic --partitions 3 --replication-factor 2
```

- Afficher la liste des Topic en exécutant la commande suivante :

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

c. Envoyer un message au Topic

- Lancer la commande suivante pour envoyer un message :

```
sh kafka-console-producer.sh --bootstrap-server localhost:9092 --topic my-first-topic  
>Hello World
```

```
>The weather is fine
>I love Kafka
```

d. Consommer les messages du Topic

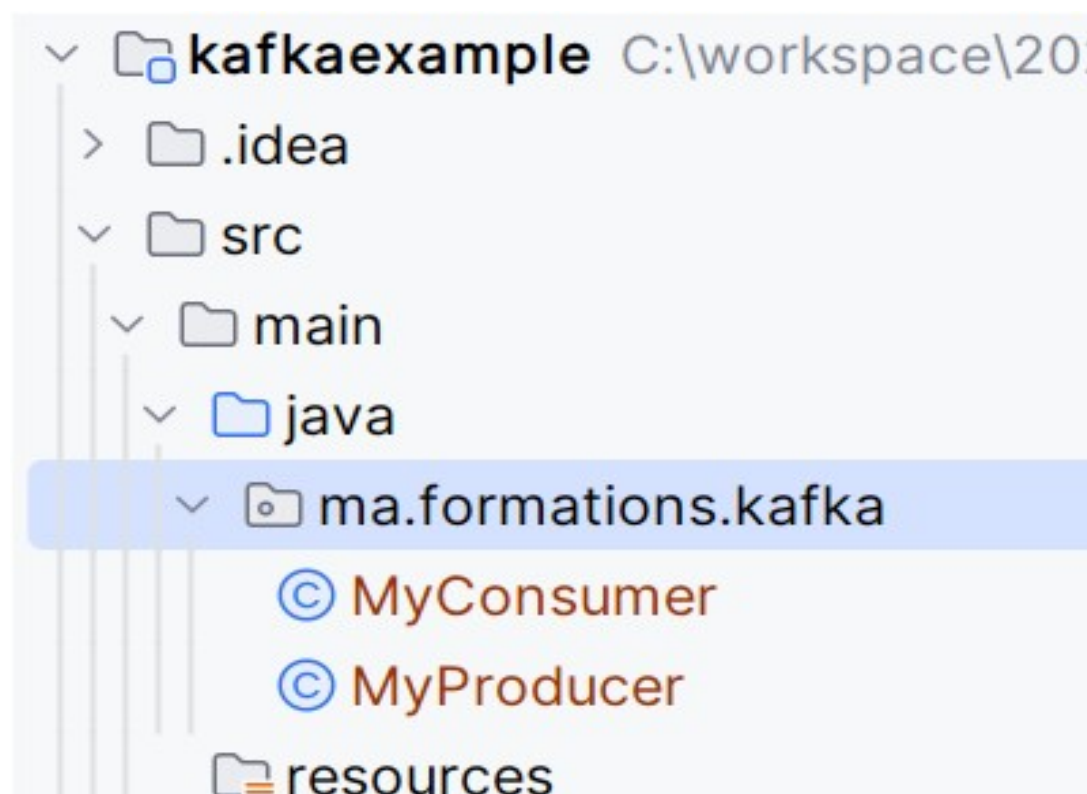
- Lancer la commande suivante pour consommer les messages :

```
sh kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic my-first-topic --from-beginning
>Hello World
>The weather is fine
>I love Kafka
```

VI. Développer une application avec Java et Kafka

- Créer un projet Maven (par exemple kafkaexample).

- Créer l'arborescence suivante :



- Ajouter les dépendances suivantes au niveau du fichier pom.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
```

```

4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>ma.formations.kafka</groupId>
  <artifactId>kafkaexample</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>3.9.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-slf4j-impl</artifactId>
      <version>2.20.0</version>
    </dependency>
  </dependencies>
</project>

```

- Créer la classe MyConsumer suivante :

```

package ma.formations.kafka;

import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.time.Duration;
import java.util.List;
import java.util.Properties;

public class MyConsumer {
  public static void main(String[] args) {
    Properties props = new Properties();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "MyFirstConsumer");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
    // receive messages that were sent before the consumer started
    props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
    // create the consumer using props.
    try (final Consumer<Long, String> consumer = new KafkaConsumer<>(props)) {

```

```
// subscribe to the topic.
final String topic = "my-first-topic";
consumer.subscribe(List.of(topic));
// poll messages from the topic and print them to the console
while (true) {
    consumer
        .poll(Duration.ofMinutes(1))
        .forEach(System.out::println);
}
}
}
```



- Concernant les deux propriétés **`ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG`** et **`ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG`** : Lorsque vous consommez des messages de Kafka, chaque message a une **clé** et une **valeur**. Kafka envoie ces éléments sous forme de **bytes**, et vous devez spécifier comment les **désérialiser** en un format compréhensible par votre application. Ici, nous avons utilisé la classe de Kafka : **`org.apache.kafka.common.serialization.StringDeserializer`**
- Concernant la propriété **`ConsumerConfig.AUTO_OFFSET_RESET_CONFIG`** : est utilisée pour configurer le comportement du **Kafka Consumer** lorsqu'il consomme des messages d'un **topic** Kafka et qu'il n'a pas d'offset **commencé** ou si l'offset précédemment consommé est **inaccessible** (par exemple, si les messages ont été supprimés du topic).
- **"earliest"** : Cette valeur indique au consommateur de commencer à consommer les messages **à partir du plus ancien** (c'est-à-dire le message le plus ancien disponible dans le topic) si l'offset demandé est invalide. Cela signifie que le consommateur consommera tous les messages existants dans le topic depuis le début.
- Valeurs possibles pour **`AUTO_OFFSET_RESET_CONFIG`** :
 - ✓ **latest** : Le consommateur commencera à consommer les messages **à partir du plus récent** (c'est-à-dire le message le plus récemment produit) si aucun offset n'existe ou si l'offset est invalide. C'est le comportement par défaut, et il est utilisé lorsque vous voulez seulement consommer les messages produits après que le consommateur ait commencé à s'abonner.
 - ✓ **none** : Si aucune information d'offset valide n'est trouvée (par exemple, si le topic n'existe pas), une exception sera lancée. Cela empêche le consommateur de commencer s'il ne trouve pas d'offset valide à utiliser.

- Créer la classe MyProducer suivante :

```
package ma.formations.kafka;

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

```

import java.util.Properties;

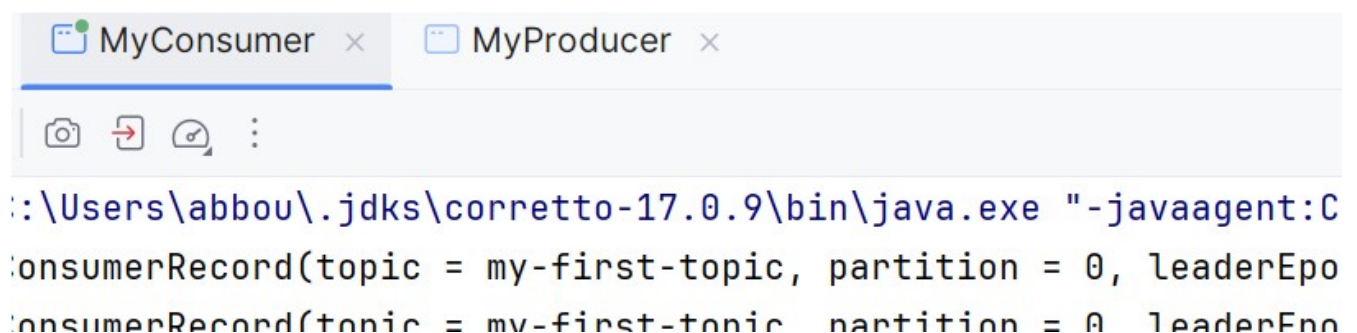
public class MyProducer {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092"); // Adresse du cluster Kafka
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("acks", "all"); // Garantir que les données sont confirmées par Kafka

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

        try {
            int i = 1;
            String message = "Message " + (i + 1);
            // Création du message à envoyer
            ProducerRecord<String, String> record = new ProducerRecord<>("my-first-topic", Integer.toString(i),
message);
            // Envoi du message
            producer.send(record);
            System.out.println("Message envoyé : clé = " + i + ", valeur = " + message);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // Fermeture du producteur
            producer.close();
        }
    }
}

```

- Pour tester l'application, démarrer la méthode main de votre Consumer et exécuter ensuite la méthode main de votre Producer et remarquer que les messages sont bien consommés par le consumer :



```

C:\Users\abbou\.jdk\corretto-17.0.9\bin>java.exe -javaagent:C:\Users\abbou\.jdk\corretto-17.0.9\bin\javaagent.jar -jar MyConsumer.jar
ConsumerRecord(topic = my-first-topic, partition = 0, leaderEpoch = 0, value = 1)
ConsumerRecord(topic = my-first-topic, partition = 0, leaderEpoch = 0, value = 2)

```