**BiG Data**

**TP3 - La Collecte de Données avec le Bus Kafka**

Annassiri Fatima Zahra \_ MBisd2

**Introduction : les données distribuées**

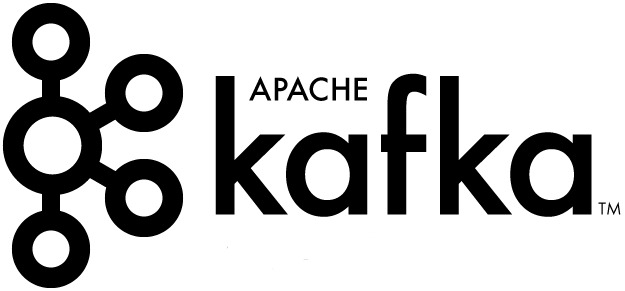
En informatique, une base de données distribuée ou BDD (en anglais : Distributed database) est une base de données dont la gestion est traitée par un réseau d'ordinateurs interconnectés qui stockent des données de manière distribuée. Ce stockage peut être soit partitionné entre différents nœuds du réseau, soit répliqué entièrement sur chacun d'eux, ou soit organisé de façon hybride1.( Wikipédia).

**Objectif du TP :**

Utilisation de Kafka pour une collecte de données distribuée, et intégration avec Spark.

**Apach Kafka :**

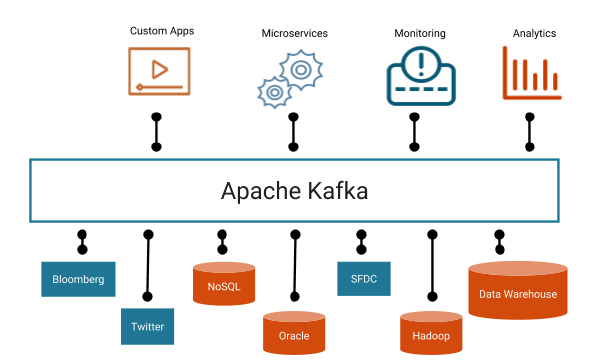
Apache Kafka est un projet à code source ouvert d'agent de messages développé par l'Apache Software Fondation et écrit en Scala. Le projet vise à fournir un système unifié, en temps réel à latence faible pour la manipulation de flux de données. Sa conception est fortement influencée par les journaux de transactions.



Kafka est un système distribué composé de **serveurs** et de **clients** qui communiquent via un protocole réseau TCP haute performance . Il peut être déployé sur du matériel nu, des machines virtuelles et des conteneurs dans des environnements sur site ou cloud.

**Serveurs** : Kafka est exécuté en tant que cluster d'un ou plusieurs serveurs pouvant couvrir plusieurs centres de données ou régions cloud. Certains de ces serveurs forment la couche de stockage, appelée les courtiers. D'autres serveurs exécutent Kafka Connect pour importer et exporter en permanence des données sous forme de flux d'événements afin d'intégrer Kafka à vos systèmes existants tels que les bases de données relationnelles ainsi que d'autres clusters Kafka. Pour vous permettre de mettre en œuvre des cas d'utilisation critiques, un cluster Kafka est hautement évolutif et tolérant aux pannes: si l'un de ses serveurs tombe en panne, les autres serveurs prendront en charge leur travail pour assurer des opérations continues sans aucune perte de données.

**Clients** : ils vous permettent d'écrire des applications et des microservices distribués qui lisent, écrivent et traitent des flux d'événements en parallèle, à grande échelle et de manière tolérante aux pannes, même en cas de problèmes de réseau ou de pannes de machine. Kafka est livré avec certains de ces clients inclus, qui sont augmentés par des dizaines de clients fournis par la communauté Kafka: les clients sont disponibles pour Java et Scala, y compris la bibliothèque Kafka Stream de niveau supérieur , pour Go, Python, C / C ++ et bien d'autres programmes langues ainsi que les API REST.



### Kafka et Zookeeper :

## Zookeeper est nécessaire pour Apache Kafka et voila le raisonnement :

### Élection du contrôleur

Le contrôleur est l'une des entités de courtage les plus importantes d'un écosystème Kafka, et il a également la responsabilité de maintenir la relation leader-suiveur sur toutes les partitions. Si un nœud s'arrête pour une raison quelconque, il est de la responsabilité du contrôleur de dire à tous les répliques d'agir en tant que chefs de partition afin de remplir les fonctions des chefs de partition sur le nœud qui est sur le point d'échouer. Ainsi, chaque fois qu'un nœud s'arrête, un nouveau contrôleur peut être élu et il peut également être assuré qu'à un moment donné, il n'y a qu'un seul contrôleur et tous les nœuds suiveurs sont d'accord sur cela.

### Configuration des sujets

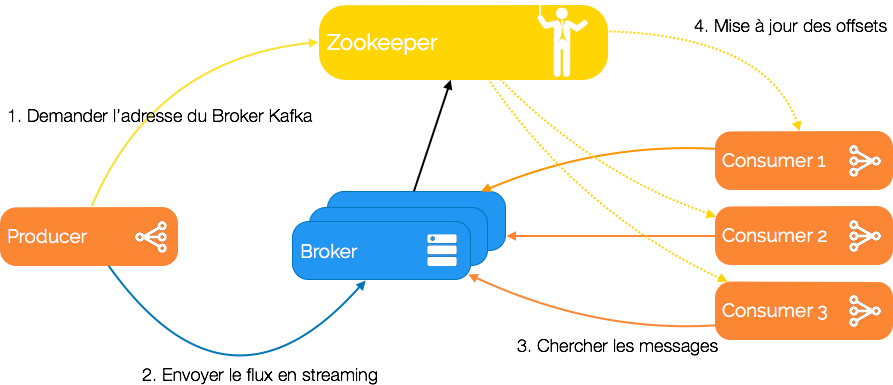
La configuration concernant tous les sujets, y compris la liste des sujets existants, le nombre de partitions pour chaque sujet, l'emplacement de tous les répliques, la liste des remplacements de configuration pour tous les sujets et quel nœud est le leader préféré, etc.

### Listes de contrôle d'accès

Les listes de contrôle d'accès ou ACL pour tous les sujets sont également maintenues dans Zookeeper.

### Appartenance au cluster

Zookeeper maintient également une liste de tous les courtiers qui fonctionnent à un moment donné et qui font partie du cluster.



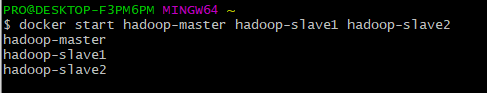
## **Outils et Versions :**

* [Apache Kafka](https://kafka.apache.org/) Version 2.11-0.8.2.1
* [Apache Hadoop](http://hadoop.apache.org/) Version: 2.7.2
* [Apache Spark](https://spark.apache.org/) Version: 2.2.1
* [Docker](https://www.docker.com/) Version 17.09.1
* [IntelliJ IDEA](https://www.jetbrains.com/idea/download/) Version Ultimate 2016.1 (ou tout autre IDE de votre choix)
* [Java](http://www.oracle.com/technetwork/java/javase/downloads/index.html) Version 1.8
* Unix-like ou Unix-based Systems (Divers Linux et MacO

**Installation**

Comme le téléchargement de l’image docker est faite dans le dernier tp, il suffit donc de lancer la machine via la commande :

docker start hadoop-master hadoop-slave1 hadoop-slave2

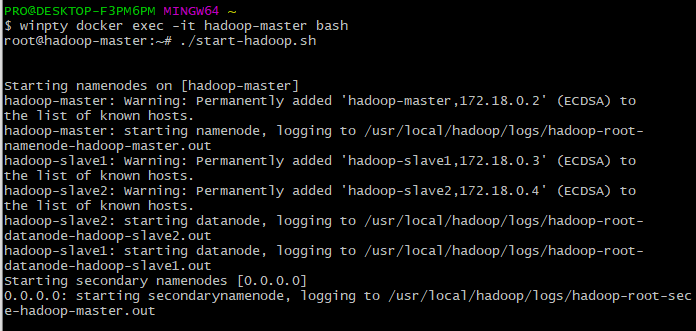


Puis d'entrer dans le centenaire master via la commande :

docker exec -it hadoop-master bash

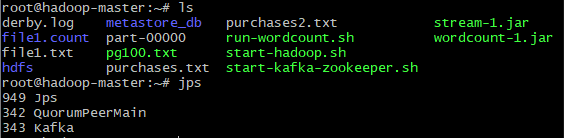
et Lancer ensuite les démons yarn et hdfs via la commande :

./start-hadoop.sh

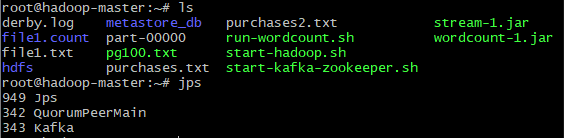


Lancement de Kafka et Zookeeper à l’aide de la commande :

./start-kafka-zookeeper.sh



La commande jsp affiche les processus java qui sont en cours de l’exécution :

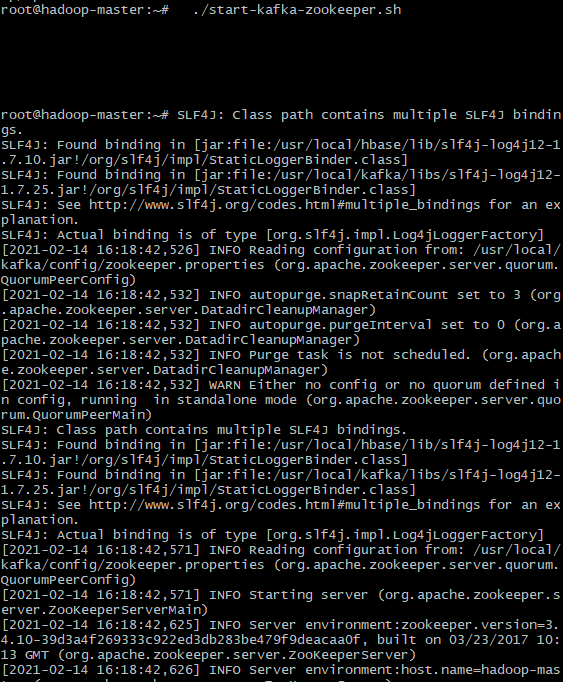


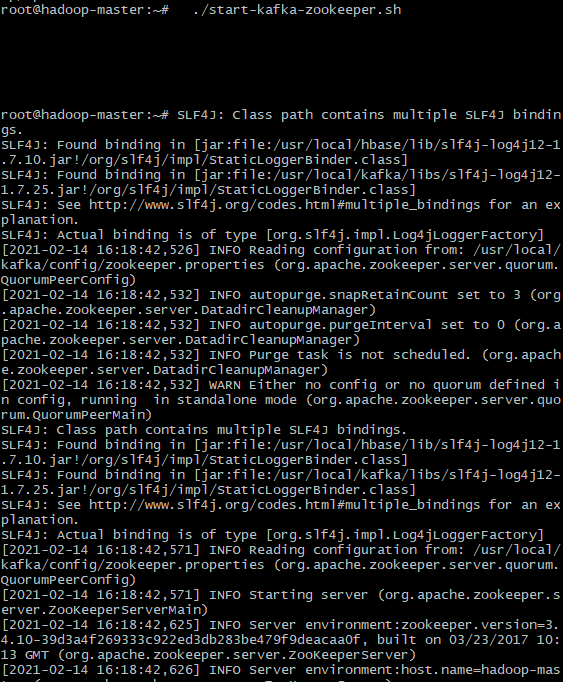
**Première utilisation de Kafka**

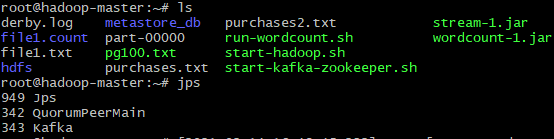
### Création d'un topic

Pour gérer les topics, Kafka fournit une commande appelée :

 kafka-topics.sh.





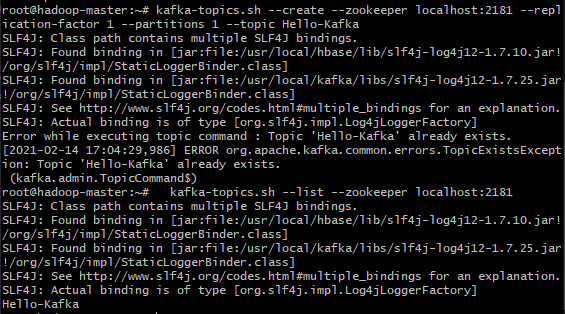


Dans un nouveau terminal, taper la commande suivante pour créer un nouveau topic appelé "Hello-Kafka".

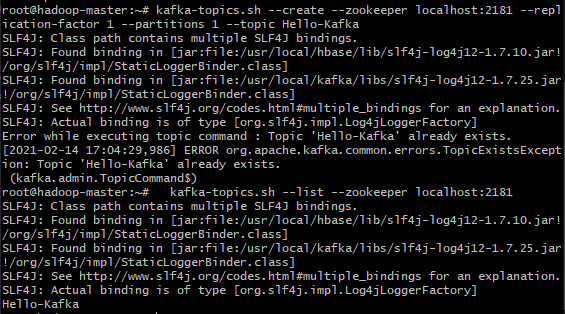
kafka-topics.sh --create --zookeeper localhost:2181

--replication-factor 1 --partitions 1

--topic Hello-Kafka



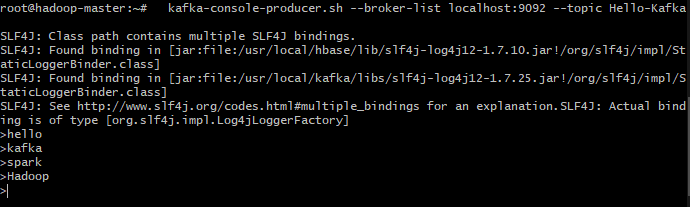
Le résultat devrait être (parmi un grand nombre de lignes d'INFO):



**Exemple Producteur Consommateur**

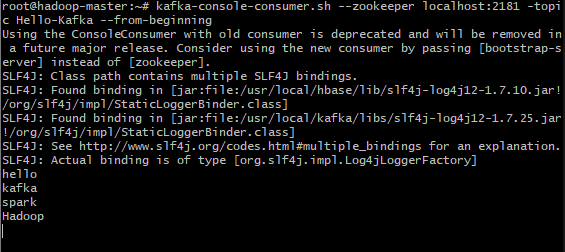
Kafka fournit un exemple de producteur standard que vous pouvez directement utiliser. Il suffit de taper:

kafka-console-producer.sh --broker-list localhost:9092 --topic Hello- Kafka



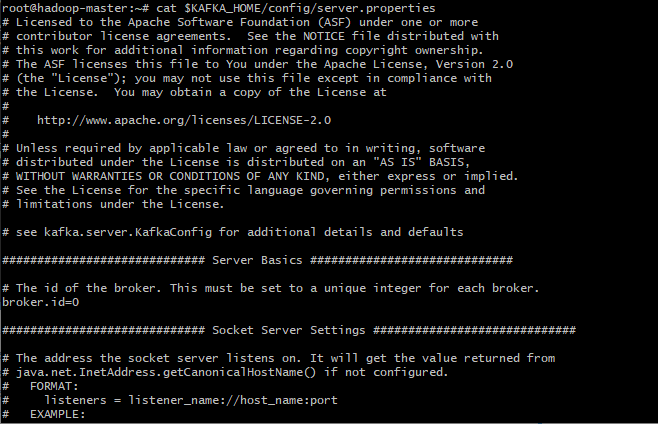
Lancer un consommateur sur le même Zookeeper sur le même topic : Hello Kafka

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



**Configuration de plusieurs brokers**

Pour créer plusieurs brokers, il suffit de dupliquer ce fichier :



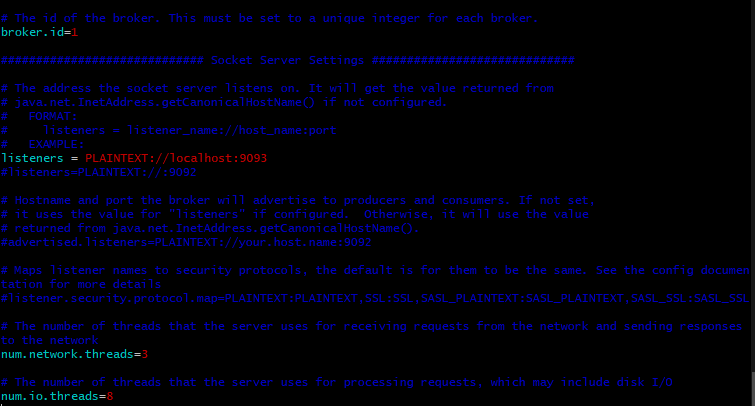
On va  créer deux autres fichiers: server1.properties et server2.properties :



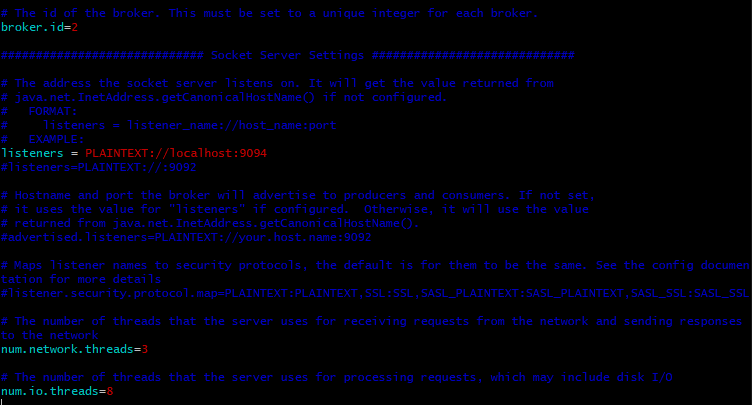
A l’aide de la commande suivante :

vi $KAFKA\_HOME/config/server1.properties

On a le droit de modifier les champs suivant :



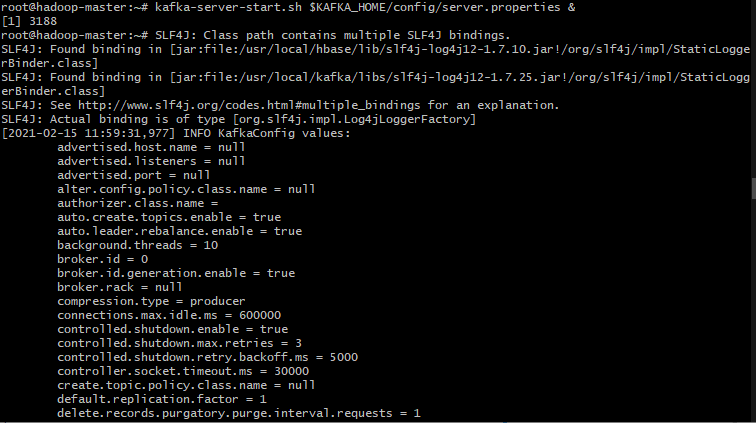
Ensuite la deuxième fichier: vi $KAFKA\_HOME/config/server2.properties



Pour démarrer les différents brokers, il suffit d'appeler kafka-server-start.sh avec les nouveaux fichiers de configuration.

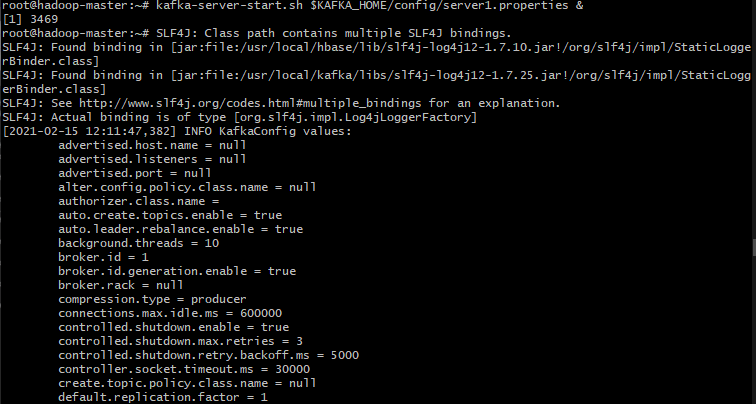
* La commande:

kafka-server-start.sh $KAFKA\_HOME/config/ server.properties &



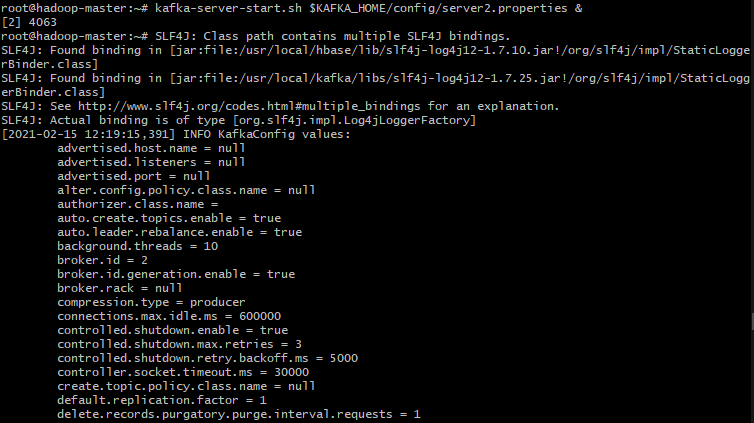
La commande :

kafka-server-start.sh $KAFKA\_HOME/config/server1.properties &

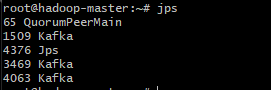


La command :

kafka-server-start.sh $KAFKA\_HOME/config/server2.properties &



Pour voir les trois serveurs s'exécuter, on lance la commande jps:

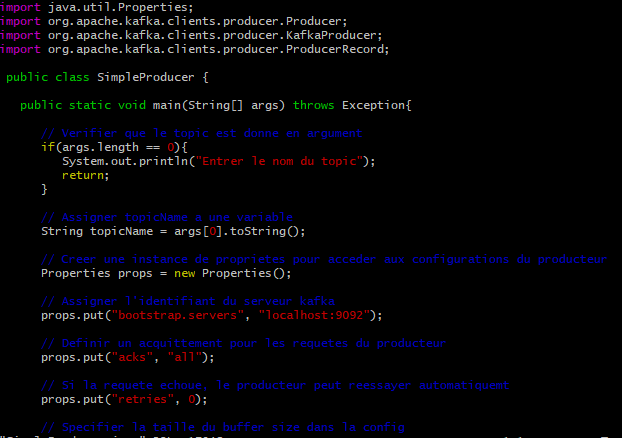


**Création d'une application personnalisée**

On va créer un docier kafka-code et un fichier de type java: SimpleProducer.java :



Le code est:



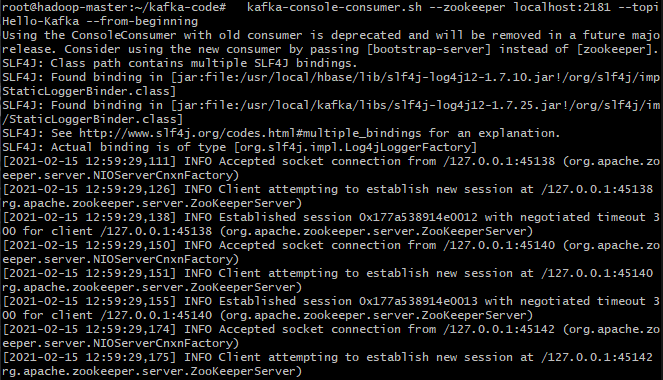
Pour exécuter le code :



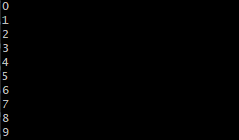
Exécuté le code avec le toppic Hello-kafka :



Pour voir le résultat saisi dans Kafka, il est possible d'utiliser le consommateur prédéfini de Kafka, à condition d'utiliser le même topic:



Et comme résultat est:

### Consommateur

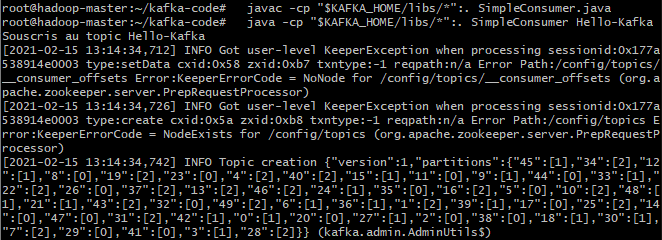
On a va créer un fichier s’appelle SimpleCosumer.java :

### 

Avec le code suivant :

### 

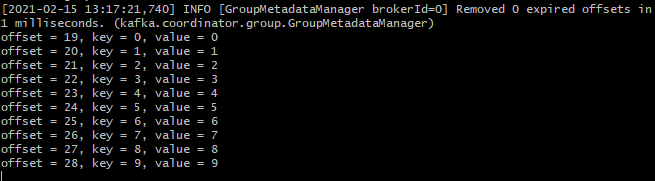
Pour compiler le consommateur on va lancer les deux commandes suivant:



On va relancer le producer:

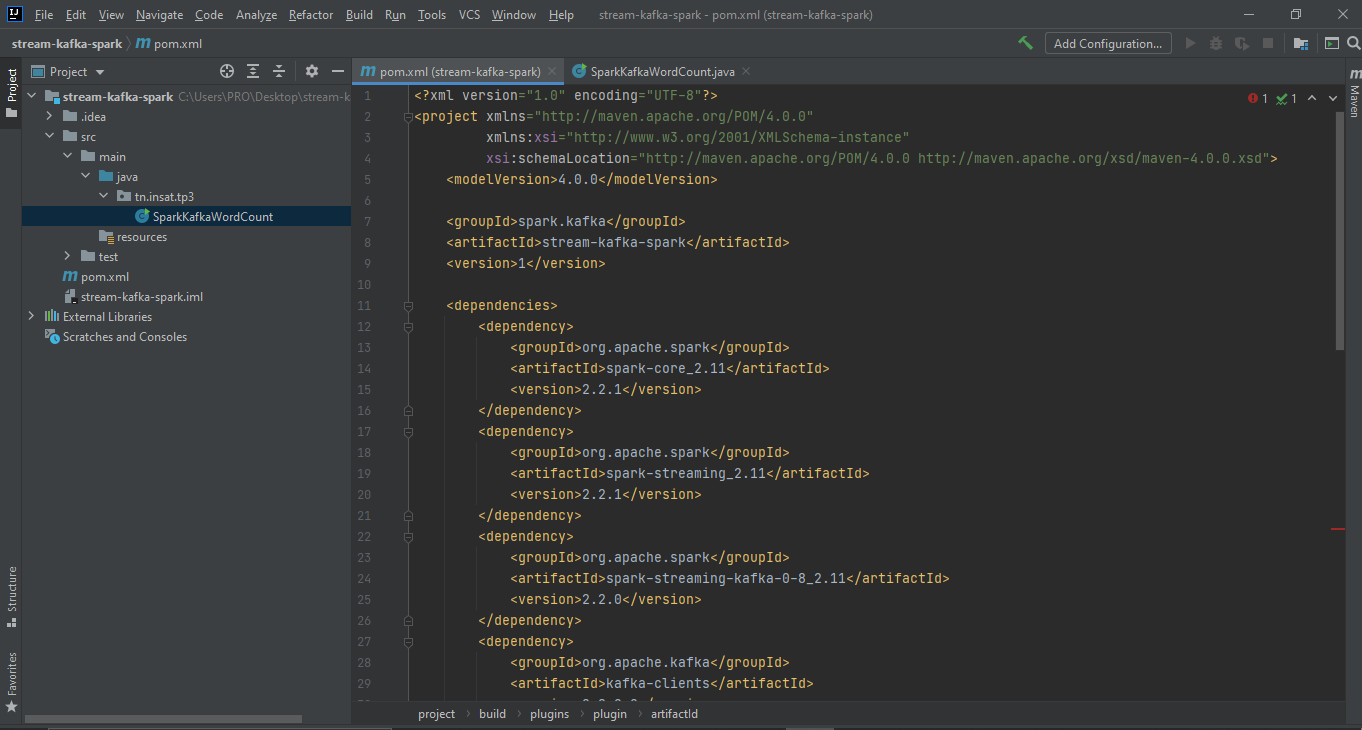


Et comme résultat on a :

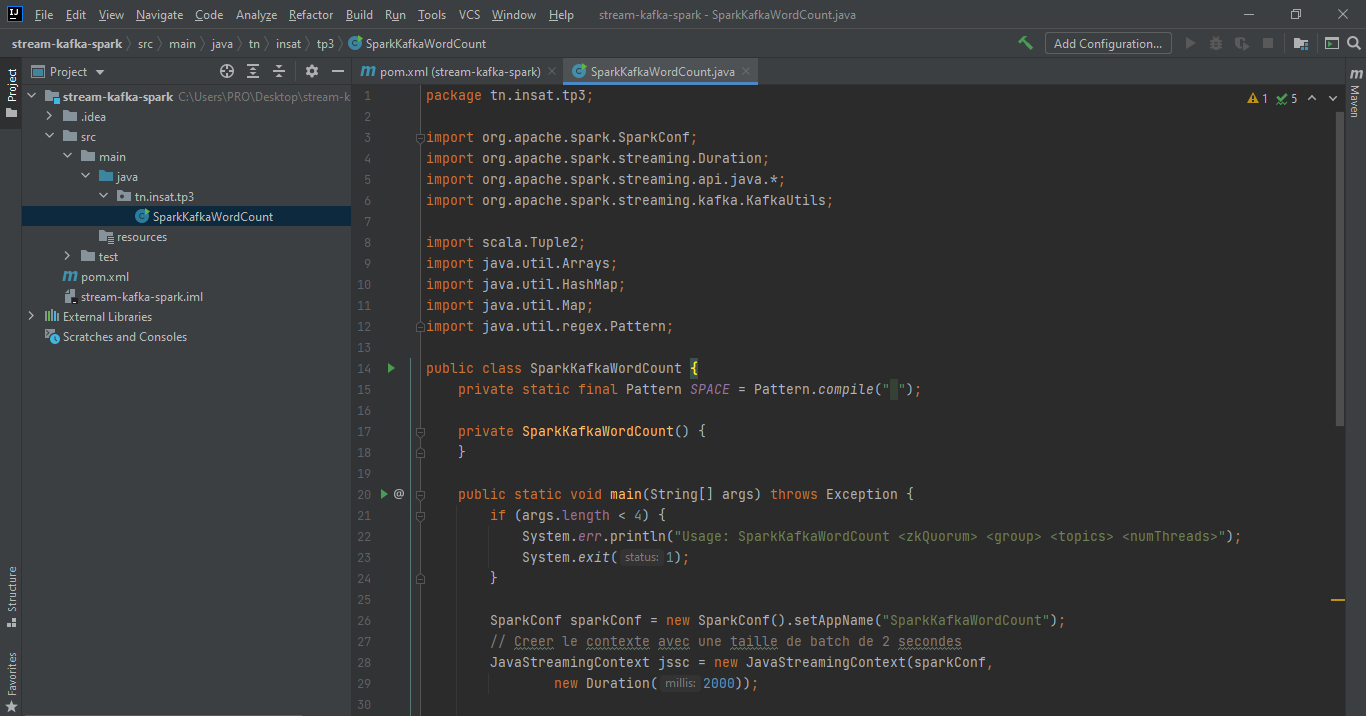


**Intégration de Kafka avec Spark**

On va créer un nouveau projet, modifier le fichier pom.xml pour qu'il ressemble à ce qui suit:

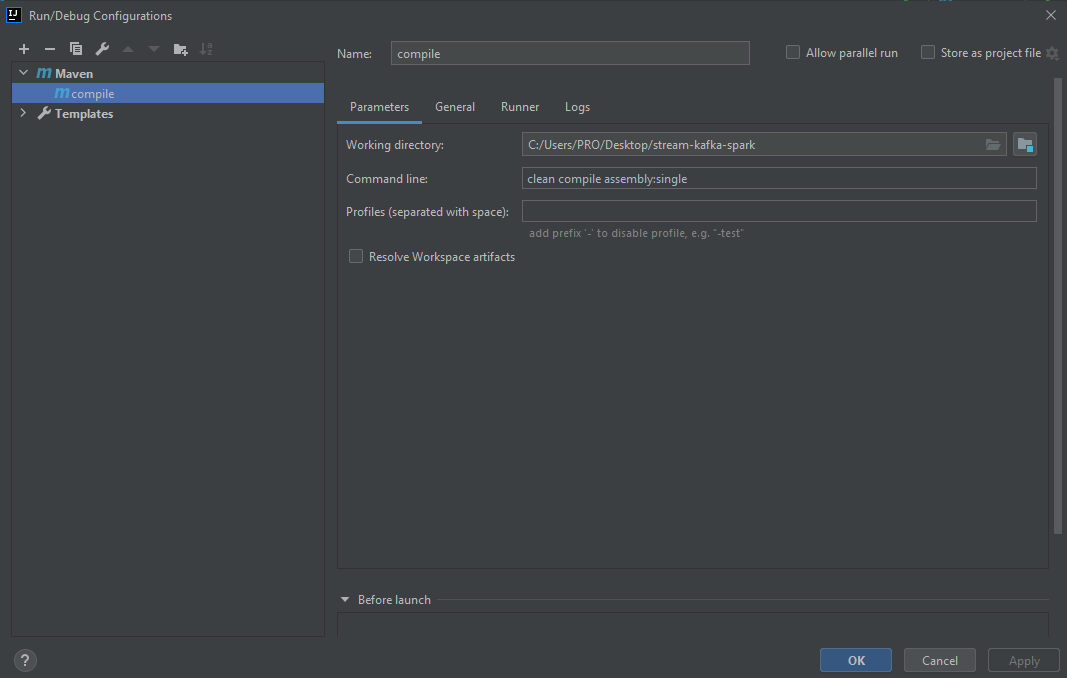


On va un créer un package et un class java :



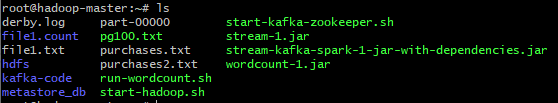
On va créer une configuration Maven pour lancer la commande:

**mvn clean compile assembly:single**



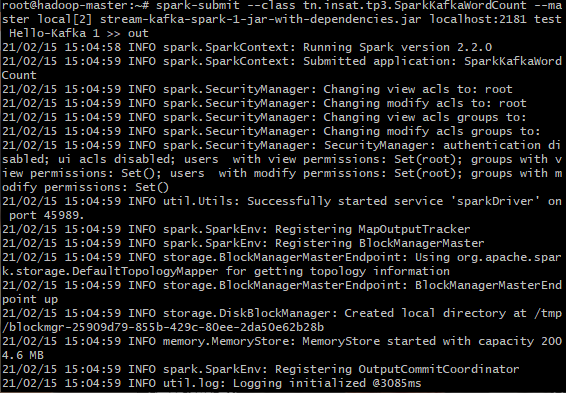
On va copier le fichier de type jar dans le container master à l’aide de la commande :

**docker cp target/stream-kafka-spark-1-jar-with-dependencies.jar hadoop-master:/root**

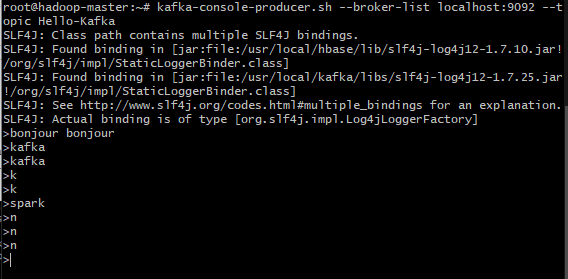


On va lancer la commande spark-submit pour lancer l'écouteur de streaming spark :

**spark-submit --class tn.insat.tp3.SparkKafkaWordCount --master local[2] stream-kafka-spark-1-jar-with-dependencies.jar localhost:2181 test Hello-Kafka 1 >> out**

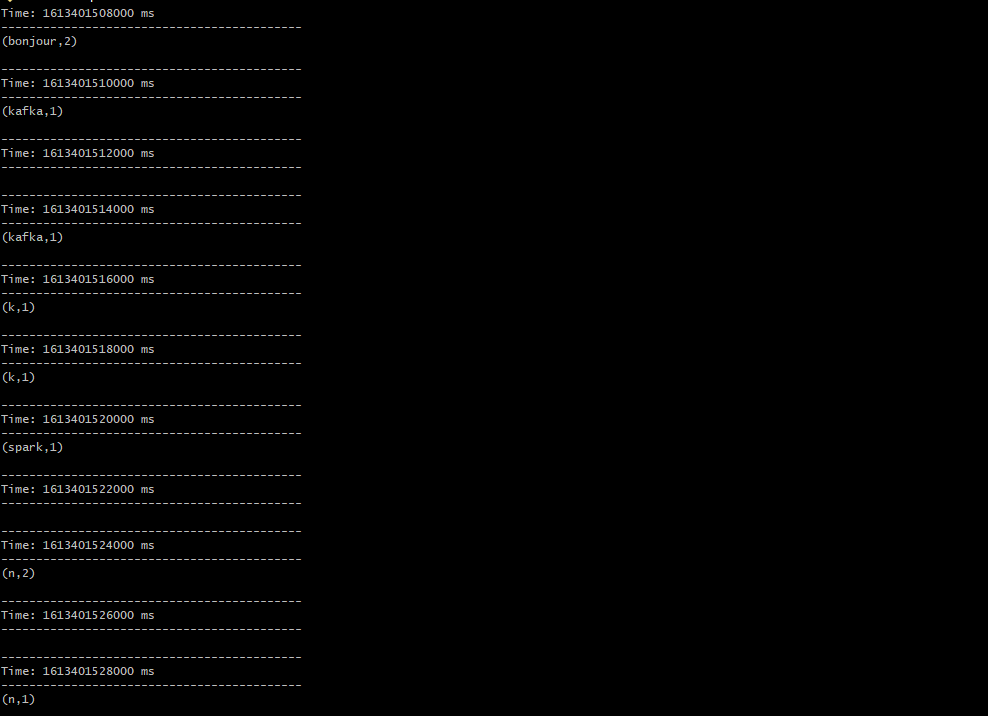


Dans une autre terminale, on va lancer le producteur prédéfini de Kafka pour tester la réaction du consommateur spark streaming:



Ecrire du texte dans la fenêtre du producteur. Ensuite, arrêter le flux de spark-submit, et observer le contenu du fichier out.

Résultat:



* The End .