Guide Kafka

Table des Matières

---

1. Installation et Configuration

1.1 Installation avec Docker Compose

Pour démarrer un environnement Kafka complet :

```
git clone https://github.com/confluentinc/cp-all-in-one.git
cd cp-all-in-one/cp-all-in-one
docker compose up -d
```

1.2 Vérification de l'installation

```
docker compose ps
```

1.3 Accès aux services

Service	Port	Description
Kafka Broker	9092	Point d'entrée principal
JMX	9101	Monitoring

Service	Port	Description
Schema Registry	8081	Gestion des schémas
Kafka Connect	8083	Connecteurs
Control Center	9021	Interface web
ksqlDB	8088	SQL streaming
REST Proxy	8082	API REST
Flink Job Manager	9081	Stream processing

Accès à l'interface web : `http://localhost:9021`

---

2. Concepts Fondamentaux

2.1 Architecture Kafka

Kafka est un système de streaming distribué basé sur :

- **Brokers** : Serveurs Kafka qui stockent et servent les données
- **Topics** : Canaux de communication logiques
- **Partitions** : Subdivisions physiques des topics
- **Producers** : Clients qui publient des messages
- **Consumers** : Clients qui consomment des messages

2.2 Topics et Partitions

Un topic est divisé en partitions pour :

- Parallélisme
- Tolérance aux pannes
- Scalabilité horizontale

2.3 Producers et Consumers

- **Producers** : Envoient des messages vers les topics
- **Consumers** : Lisent les messages depuis les topics
- **Consumer Groups** : Ensemble de consumers qui partagent la charge

---

3. Opérations de Base

3.1 Gestion des topics

Se connecter au broker :

```
docker exec -it broker /bin/bash
```

Créer un topic :

```
kafka-topics --create --topic mon_topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
```

Lister les topics :

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

Modifier le nombre de partitions :

```
kafka-topics --alter --topic mon-topic --partitions 5 --bootstrap-server localhost:9092
```

3.2 Production de messages

Producer simple :

```
kafka-console-producer --topic mon_topic --bootstrap-server localhost:9092
```

Producer avec clés :

```
kafka-console-producer --broker-list localhost:9092 --topic topic-name --property "parse.key=true" --property "key.separator=:"
```

3.3 Consommation de messages

Consumer depuis le début :

```
kafka-console-consumer --topic mon_topic --bootstrap-server localhost:9092 --from-beginning
```

Consumer avec clés :

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

Consumer avec groupe :

```
kafka-console-consumer --topic topic15 --bootstrap-server localhost:9092 --from-beginning --property print.key=true --group myapp
```

---

4. Partitioning et Distribution

4.1 Stratégies de partitioning

Kafka propose plusieurs stratégies :

1. **DefaultPartitioner** : Hash de clé + round-robin pour les messages sans clé
2. **RoundRobinPartitioner** : Distribution cyclique
3. **UniformStickyPartitioner** : Sticky + batching optimisé

4.2 Configuration des partitioners

En ligne de commande :

```
kafka-console-producer --topic mon_topic --bootstrap-server localhost:9092 --producer-property partitioner.class=org.apache.kafka.clients.producer.RoundRobinPartitioner
```

En Python :

```
from kafka import KafkaProducer
import hashlib

def default_partitioner(key_bytes, all_partitions, available_partitions):
    if key_bytes:
        # Hash-based partitioning
        partition = int(hashlib.md5(key_bytes).hexdigest(), 16) % len(all_partitions)
        return partition
    else:
        # Round-robin fallback
        return available_partitions[0]

producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
    key_serializer=str.encode,
    value_serializer=str.encode,
    partitioner=default_partitioner
)
```

En Java :

```
import org.apache.kafka.clients.producer.ProducerConfig;
import java.util.Properties;

Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.seriali
zation.StringSerializer");
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.seria
lization.StringSerializer");

// Set the partitioner
props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, "org.apache.kafka.clients.producer.
internals.DefaultPartitioner");
// Alternatives:
// props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, "org.apache.kafka.clients.produc
```

```
er.RoundRobinPartitioner");
// props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, "org.apache.kafka.clients.produc
er.UniformStickyPartitioner");

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

---

5. Consumer Groups

5.1 Gestion des groupes

Lister les groupes :

```
kafka-consumer-groups --bootstrap-server localhost:9092 --list
```

Décrire un groupe :

```
kafka-consumer-groups --bootstrap-server localhost:9092 --describe --group myapp
```

5.2 Gestion des offsets

Reset au début :

```
kafka-consumer-groups --reset-offsets --group myapp --topic topic15 --to-earliest --ex
ecute --bootstrap-server localhost:9092
```

Reset à un offset spécifique :

```
kafka-consumer-groups --reset-offsets --group myapp --topic topic15 --to-offset 10 --e
xecute --bootstrap-server localhost:9092
```

Reset à une date :

```
kafka-consumer-groups --reset-offsets --group myapp --topic topic15 --to-datetime 2025
-09-24T00:00:00.000 --execute --bootstrap-server localhost:9092
```

6. Kafka Connect

6.1 FileStream Connectors

Configuration Source :

```
{
    "name": "file-source-connector",
    "config": {
      "connector.class": "FileStreamSource",
      "file": "/tmp/input.txt",
      "topic": "file-topic"
    }
}
```

Configuration Sink :

```
{
    "name": "local-file-sink",
    "config": {
      "connector.class": "FileStreamSink",
      "tasks.max": "1",
      "file": "/tmp/test.sink.txt",
      "topics": "file-topic"
    }
}
```

Test du connector :

```
docker exec -it connect /bin/bash
echo "toto" >> /tmp/input.txt
```

6.2 JDBC Connectors

Configuration JDBC Source :

```
{
  "name": "JDBC_Source",
  "config": {
    "topic.creation.default.partitions": "1",
    "topic.creation.default.replication.factor": "1",
    "name": "JDBC_Source",
    "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
    "topic.creation.groups": "jdbc_topic_group",
    "connection.url": "jdbc:postgresql://postgres/kafka",
    "connection.user": "kafka",
    "connection.password": "connect",
    "mode": "incrementing",
    "incrementing.column.name": "id",
    "timestamp.column.name": "creation_date",
    "validate.non.null": "false",
    "query": "SELECT * FROM public.users",
    "topic.prefix": "topic_jdbc"
  }
}
```

Configuration JDBC Sink :

```
{
  "name": "JDBC_Sink",
  "config": {
    "name": "JDBC_Sink",
    "connector.class": "io.confluent.connect.jdbc.JdbcSinkConnector",
    "tasks.max": "1",
    "errors.tolerance": "all",
    "errors.log.enable": "true",
    "errors.log.include.messages": "true",
    "topics": "topic_jdbc",
    "connection.url": "jdbc:postgresql://postgres/kafka",
    "connection.user": "kafka",
    "connection.password": "connect",
    "insert.mode": "insert",
```

```
    "table.name.format": "users_dest"
  }
}
```

6.3 Configuration PostgreSQL

Ajout de PostgreSQL au docker-compose :

```
  postgres:
    image: postgres
    container_name: local_pgdb
    restart: always
    ports:
      - "5432:5432"
    environment:
      POSTGRES_USER: kafka
      POSTGRES_PASSWORD: connect
    volumes:
      - postgres-data:/var/lib/postgresql/data

  pgadmin:
    image: dpage/pgadmin4
    container_name: pgadmin4_container
    restart: always
    ports:
      - "8888:80"
    environment:
      PGADMIN_DEFAULT_EMAIL: [email protected]
      PGADMIN_DEFAULT_PASSWORD: connect
    volumes:
      - pgadmin-data:/var/lib/pgadmin

volumes:
  postgres-data:
  pgadmin-data:
```

Setup de la base de données :

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    creation_date TIMESTAMP,
    first_name varchar(100),
    last_name varchar(100)
);

INSERT INTO users (creation_date, first_name, last_name) VALUES
    (CURRENT_TIMESTAMP, 'John', 'Doe'),
    (CURRENT_TIMESTAMP, 'Nobody', 'Nobody');
```

Configuration du CLASSPATH dans docker-compose :

```
      CLASSPATH: /usr/share/jdbc/postgresql-42.4.4.jar:/usr/share/java/monitoring-inte
rceptors/monitoring-interceptors-8.0.0.jar
      CONNECT_PLUGIN_PATH: "/usr/share/java,/usr/share/confluent-hub-components,/usr/l
ocal/share/kafka/plugins,/usr/share/filestream-connectors,/usr/share/jdbc"
```

---

7. Configuration Multi-Brokers

7.1 Cluster 3 brokers

Pour un environnement de production, configurez plusieurs brokers avec réplication :

Broker 1 :

```
  broker1:
    image: confluentinc/cp-server:8.0.0
    hostname: broker1
    container_name: broker1
    ports:
      - "9092:9092"
      - "9101:9101"
```

```
    environment:
      KAFKA_NODE_ID: 1
      KAFKA_CONTROLLER_QUORUM_VOTERS: '1@broker1:29093,2@broker2:29093,3@broker3:2909
3'
      KAFKA_ADVERTISED_LISTENERS: 'PLAINTEXT://broker1:29092,PLAINTEXT_HOST://localhos
t:9092'
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 3
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 3
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 2
```

7.2 Réplication et tolérance aux pannes

Créer un topic répliqué :

```
kafka-topics --bootstrap-server broker1:29092,broker2:29092,broker3:29092 --create --t
opic test-replication --partitions 3 --replication-factor 3 --config min.insync.replic
as=2 --config acks=all
```

Test de tolérance aux pannes :

```
docker stop broker2
```

Calcul du quorum :

- Taille du quorum = (nombre total de voters / 2) + 1
- Exemple : 3 voters → quorum = (3/2) + 1 = 2
- Le cluster peut tolérer la panne d'1 broker

Élection de leader :

```
kafka-leader-election --bootstrap-server broker1:29092 --election-type preferred --all
-topic-partitions
```

---

8. Kafka Streams vs Consumer API

8.1 Comparaison des approches

Aspect	Kafka Consumer API	Kafka Streams
Niveau d'abstraction	Bas niveau (low-level)	Haut niveau (high-level)
Usage principal	Consommation manuelle de messages, contrôle fin	Traitement de streams, transformations, analyses
Gestion d'état	Pas de gestion d'état intégrée	Gestion automatique d'état (KTable, fenêtres)
Tolérance aux pannes	Manuel (offsets, rebalancing)	Automatique via Kafka (état répliqué)
Programmation	Code impératif, polling des messages	DSL déclaratif (map, filter, join, etc.)
Scalabilité	Haute, via partitions et groupes de consommateurs	Haute, avec parallélisation automatique
Latence	Faible, mais dépend de l'implémentation	Optimisée pour le temps réel
Complexité	Moyenne à élevée (gestion offsets, commits)	Faible pour les transformations courantes
Cas d'usage	Intégration personnalisée, micro-batching	ETL temps réel, agrégations, joins de streams
Dépendances	kafka-clients.jar	kafka-streams.jar (inclut consumer/producer)

8.2 Cas d'usage

Utiliser Consumer API quand :

- Besoin de contrôle fin sur la consommation
- Intégration avec des systèmes externes spécifiques
- Logic métier complexe nécessitant un contrôle manuel

Utiliser Kafka Streams quand :

- Transformations de données en temps réel
- Agrégations et analyses de streams
- Joins entre différents topics
- Besoins de fenêtres temporelles

Exemple Kafka Streams simple :

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> source = builder.stream("input-topic");

source.filter((key, value) -> value.length() > 5)
      .mapValues(value -> value.toUpperCase())
      .to("output-topic");

KafkaStreams streams = new KafkaStreams(builder.build(), props);
streams.start();
```

---

9. ksqlDB

9.1 Introduction à ksqlDB

ksqlDB est un moteur de traitement de streams qui permet d'écrire des applications de streaming en utilisant SQL. Il combine la puissance de Kafka Streams avec la simplicité du SQL.

Accès à ksqlDB :

- Interface web : `http://localhost:8088`
- CLI : `docker exec -it ksqldb-cli ksql http://ksqldb-server:8088`

Avantages de ksqlDB :

- Syntaxe SQL familière
- Traitement en temps réel
- Intégration native avec Kafka
- Gestion automatique de l'état
- Scalabilité horizontale

9.2 Requêtes de base

Créer un stream :

```
CREATE STREAM user_stream (
  id INT,
  name VARCHAR,
  email VARCHAR
) WITH (
  KAFKA_TOPIC='users',
  VALUE_FORMAT='JSON'
);
```

Créer une table :

```
CREATE TABLE user_table (
  id INT PRIMARY KEY,
  name VARCHAR,
  email VARCHAR
) WITH (
  KAFKA_TOPIC='users',
  VALUE_FORMAT='JSON'
);
```

Requêtes de transformation :

```
-- Filtrage
CREATE STREAM filtered_users AS
SELECT * FROM user_stream
WHERE name IS NOT NULL;

-- Agrégation
CREATE TABLE user_counts AS
SELECT COUNT(*) as user_count
FROM user_stream
GROUP BY name;

-- Join
CREATE STREAM enriched_orders AS
SELECT o.order_id, o.amount, u.name
```

```
FROM orders_stream o
JOIN user_table u ON o.user_id = u.id;
```

9.3 Streams et Tables

Stream vs Table dans ksqlDB :

Stream	Table
Séquence d'événements	État actuel
Append-only	Upsert (mise à jour)
Tous les changements	Dernière valeur par clé
Events	State

Exemples pratiques :

```
-- Créer un stream depuis un topic existant
CREATE STREAM pageviews_stream (
  viewtime BIGINT,
  userid VARCHAR,
  pageid VARCHAR
) WITH (
  KAFKA_TOPIC='pageviews',
  VALUE_FORMAT='JSON'
);

-- Compter les vues par utilisateur (fenêtre glissante)
CREATE TABLE pageviews_per_user AS
SELECT userid, COUNT(*) as view_count
FROM pageviews_stream
WINDOW TUMBLING (SIZE 1 HOUR)
GROUP BY userid;

-- Requête continue (push query)
SELECT * FROM pageviews_per_user EMIT CHANGES;
```

```
-- Requête ponctuelle (pull query)
SELECT * FROM pageviews_per_user WHERE userid = 'user123';
```

Configuration avancée :

```
-- Stream avec fenêtre temporelle
CREATE STREAM user_activity_hourly AS
SELECT userid, COUNT(*) as activity_count
FROM user_stream
WINDOW TUMBLING (SIZE 1 HOUR)
GROUP BY userid;

-- Détection d'anomalies
CREATE STREAM high_activity_users AS
SELECT userid, activity_count
FROM user_activity_hourly
WHERE activity_count > 100;
```

---

10. Développement avec Kafka

10.1 Configuration Maven

Repository Confluent :

```
<repositories>
    <repository>
        <id>confluent</id>
        <url>https://packages.confluent.io/maven/</url>
    </repository>
</repositories>
```

Dépendances :

```
<dependency>
    <groupId>org.apache.kafka</groupId>
```

```
    <artifactId>kafka-clients</artifactId>
    <version>3.4.0</version>
</dependency>
```

10.2 Exemples Python

Installation :

```
git clone https://github.com/kyo06/kafka-examples
cd kafka-consumer-example
sudo apt update
sudo apt install python3-full
python3 -m venv .
source bin/activate
python3 -m pip install -r requirements.txt
python3 -m pip install kafka-python-ng
python3 main.py
```

10.3 Exemples Java

Résolution du problème de groupe :

```
Exception in thread "main" org.apache.kafka.common.errors.InvalidGroupIdException: To use the group management or offset commit APIs, you must provide a valid group.id in the consumer configuration.
```

Solution : Ajouter la configuration `group.id` dans les propriétés du consumer.

Ressources Supplémentaires

- Documentation Confluent Cloud Connectors
- Guide des stratégies de partitioning
- Documentation Kafka Connect FileStream
- Debezium - Change Data Capture
- Confluent Hub

- Kafka Message Key Guide

Troubleshooting

Problèmes courants

1. **Répertoire manquant :** `/usr/share/filestream-connectors`
    - Solution : Ajouter le chemin dans `CONNECT_PLUGIN_PATH`
2. **Dépendance Maven introuvable :** `io.confluent.ksql`
    - Solution : Vérifier le repository Confluent dans `pom.xml`
3. **Erreur de groupe consumer :**
    - Solution : Définir `group.id` dans la configuration
4. **Problèmes de connecteurs JDBC :**
    - Vérifier le CLASSPATH
    - S'assurer que les drivers sont dans le bon répertoire
    - Redémarrer le service Connect après ajout de plugins