

# KAFKA

# 1. 개요

Apache Kafka는 고성능 데이터 파이프 라인, 스트리밍 분석, 데이터 통합 및 미션 크리티컬 애플리케이션을 위해 사용하는 오픈 소스 분산 이벤트 스트리밍 플랫폼

## 01. Kafka는 대량의 데이터를 높은 처리량과 실시간 처리를 위한 오픈 소스로 다음과 특징이 있음

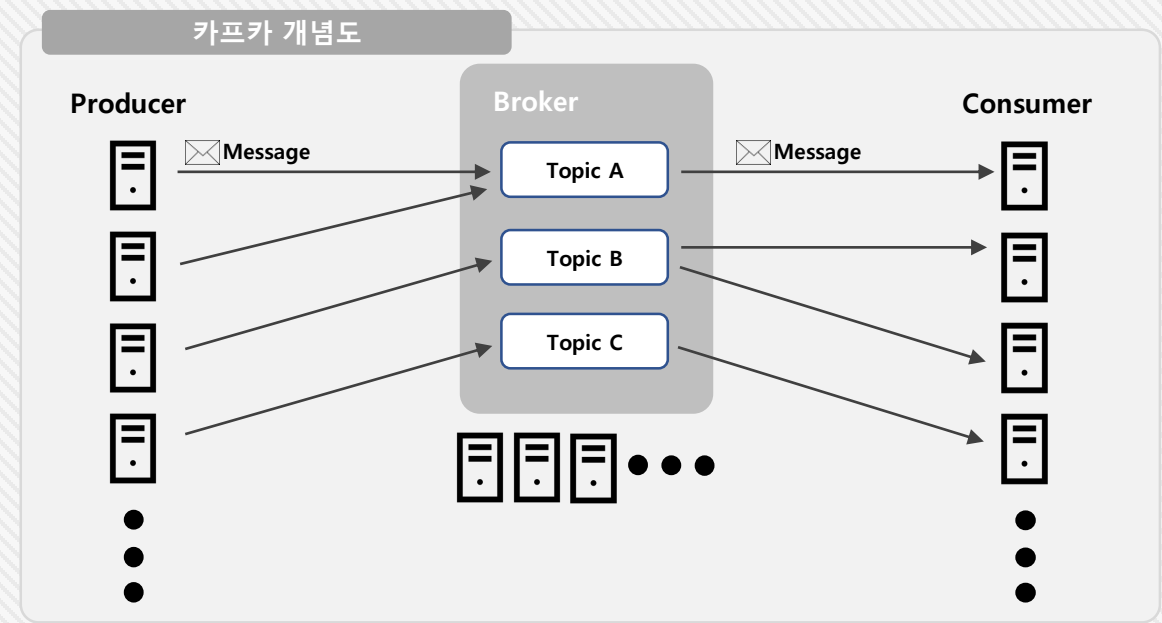
- 확장성 : 데이터 양에 따라서 시스템 확장 가능
- 연속성 : 수신 데이터를 디스크에 저장 하기 때문에 언제라도 데이터를 읽을 수 있음
- 유연성 : 다른 제품이나 시스템을 연결 하는 허브 역할  
Connect API -> Kafka Connect 제공  
Streams API -> Kafka Streams 제공
- 신뢰성 : 메시지 전달 보증으로 데이터 상실은 허용 하지 않음

종류	개요	재전송	중복삭제	비고
At Most Once	1회는 전달 시도	X	X	메시지 중복 없음, 상실 있음
At Least Once	적어도 1회는 전달	O	X	메시지 중복 가능, 상실 없음 Ack, Offset Commit
Exactly Once	1회만 전달	O	O	메시지 중복 없음, 상실 없음, 성능 저하 Ack, Offset Commit 트랜잭션 Abort/Timeout

## 02. Kafka 메시지 모델

- 여러 Consumer가 분산 처리로 메시지를 소비하는 Queuing Model
- 여러 Subscriber에 동일한 메시지를 전달 하고, Topic 기반으로 전달 내용을 변경하는 Pub/Sub Model -> Consumer Group 개념 도입

## 03. Kafka 구성 요소

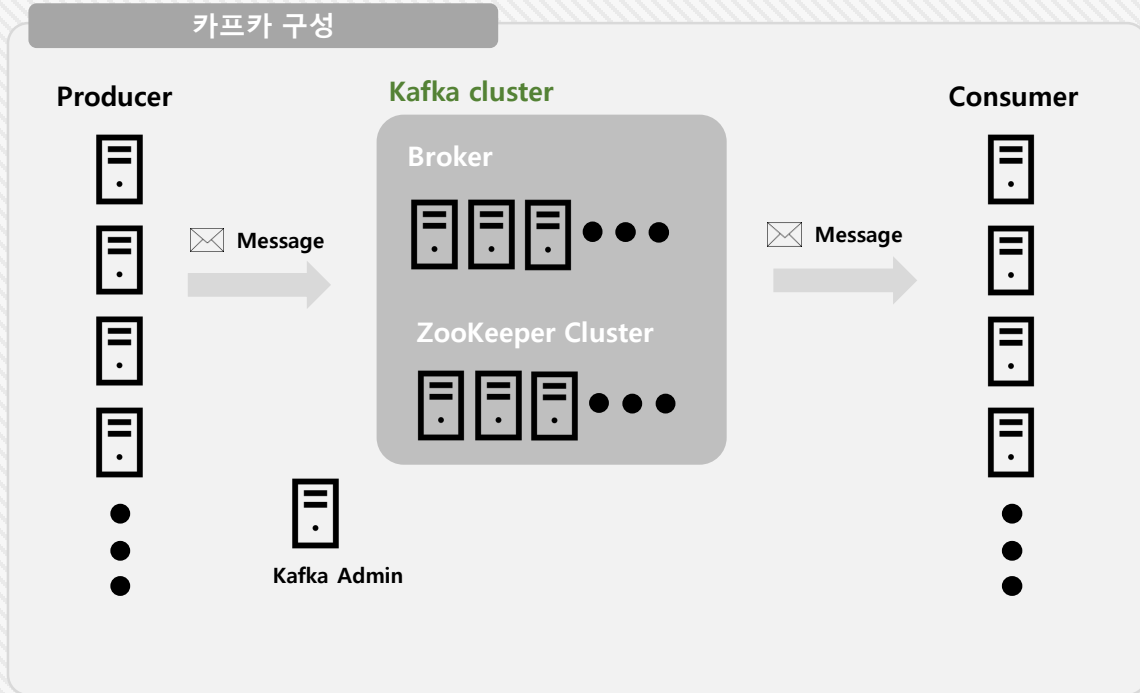


- **Broker** : 데이터를 수신, 전달(Consumer의 요구에 따라 응답)
- **Message** : 데이터의 최소 단위, key/value 구조, 전송 시 Partition 이용
- **Producer**: 데이터 생산자, broker에 Message 전달
  - 레코드를 프로듀스할 때 어느 토픽의 어느 파티션에 할당할 지를 결정한다
- **Consumer** : 메시지 읽음
- **Topic** : 메시지 종류별로 Broker에서 관리
  - 카프카 안에는 여러 레코드 스트림이 있을 수 있고 각 스트림
  - 하나의 토픽에 대해 여러 Subscriber가 붙을 수 있음

# 1. 개요

## 1. KAFKA 1. 기본설명

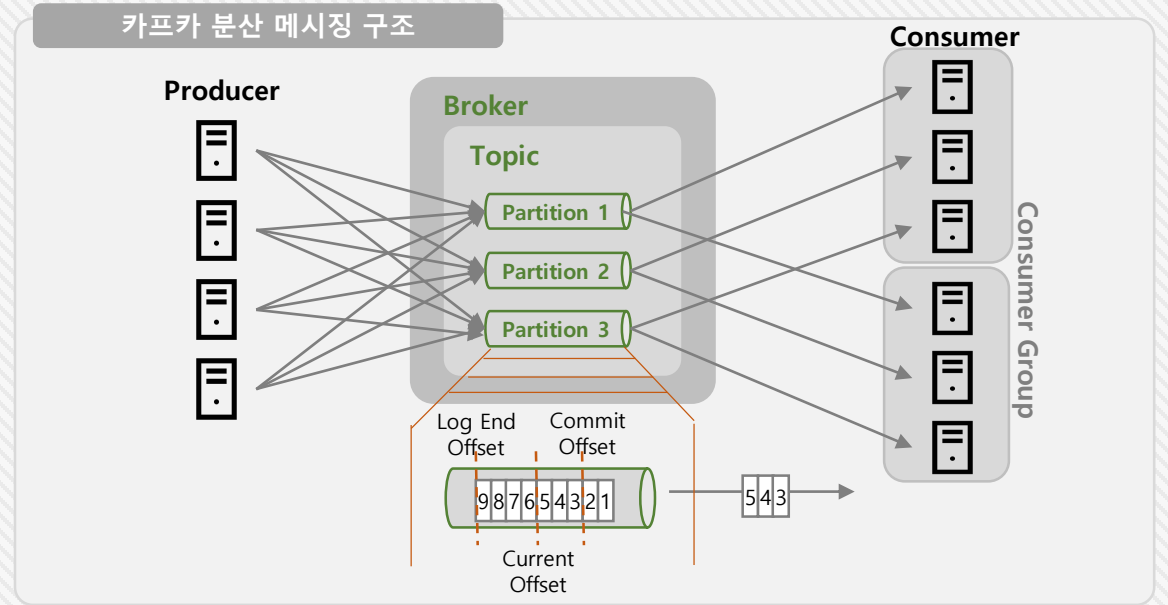
### 04. Kafka 시스템 구성



- **API** : Producer, Consumer 개발을 위한 API
- **ZooKeeper** : 분산 처리를 위한 관리 도구
  - > 분산 메시징의 메타 데이터 ( Topic, Partition .. )를 관리
  - > 카프카 클러스터의 리더(Leader)를 발탁하는 방식도 주키퍼가 제공하는 기능
- **Kafka Admin** : Kafka 관리
- **Kafka Cluster**

- **Consumer Group** : 단일 애플리케이션 안에서 여러 컨슈머가 단일 토픽이나 여러 파티션에서 메시지를 취득 하는 방법
  - 컨슈머 그룹 마다 독립적인 컨슈머 오프셋을 가진다.
  - 컨슈머 그룹 내에서 처리해야 할 파티션이 분배된다. 즉 하나의 파티션은 하나의 서버가 처리한다. 그룹에 서버가 추가되면 카프카 프로토콜에 의해 동적으로 파티션이 재분배 된다.
  - 하나의 토픽 레코드를 분산 처리하는 구조라면 동일 컨슈머 그룹을 가지게 해야 한다.
  - 하나의 토픽 레코드에 각각 별도의 처리를 하는 다른 파이프라인이라면 서로 다른 컨슈머 그룹을 가지게 해야 한다.

### 05. 분산 메시징 구조



- **Topic** : Topic : 카프카 클러스터에서 여러개 만들 수 있으며 하나의 토픽은 1개 이상의 파티션(Partition)으로 구성되어 있음.
- **Partition** : 각 토픽 당 데이터를 분산 처리하는 단위
- **Consumer Group** : 단일 애플리케이션 안에서 여러 컨슈머가 단일 토픽이나 여러 파티션에서 메시지를 취득 하는 방법
- **Offset** : 파티션 단위로 메시지 위치를 나타냄
  - > **Log-End-Offset(LEO)** : 파티션 데이터의 끝
  - > **Current Offset** : 컨슈머가 어디까지 메시지를 읽은 위치 ( Consumer Group 별 )
  - > **Commit Offset** : 컨슈머가 어디까지 커밋 했는지를 나타냄 ( Consumer Group 별 )

# 2. 메시지 송수신

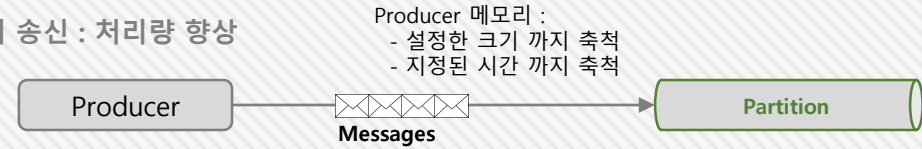
Apache Kafka는 반드시 한 메시지 단위로 송수신 하는 기능과 처리량을 높이기 위해서 메시지를 축적하여 처리 하는 기능을 제공한다.

## 01. 프로듀서 메시지 송신

- 디폴트 설정 : 한 메시지만 송신



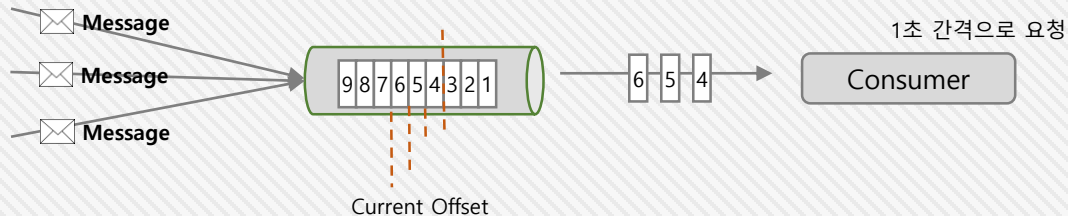
- 배치 송신 : 처리량 향상



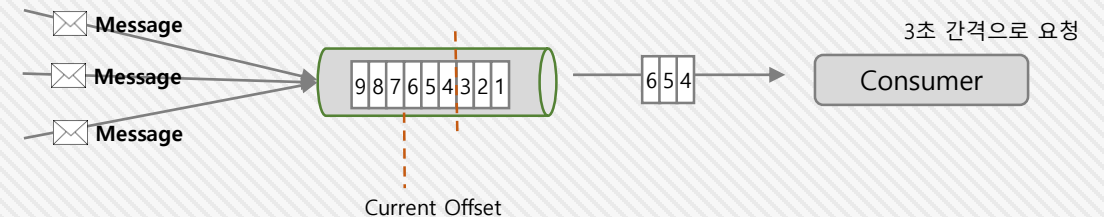
## 02. 컨슈머 메시지 수신

토픽과 파티션에 대해서 Current Offset에서 마지막으로 취득한 메시지 부터 브로커에 요청 하여 브로커에 보관 되어 있는 최신 메시지까지 수신 한다.  
메시지의 유입 빈도가 동일한 경우 컨슈머의 브로커 요청 간격이 길수록 모인 메시지가 많아 진다.

- 작은 범위로 요청 : 하나의 메시지 마다 Current Offset Update



- 일정 간격으로 요청 : 여러 메시지 수신 후 Current Offset Update



모아서 받는 경우 프로듀서 송신과 컨슈머 수신 of 지연 시간이 발생 할 수 있으므로 처리량과 대기 시간의 트레이드 오프를 고려한 설계를 해야 함

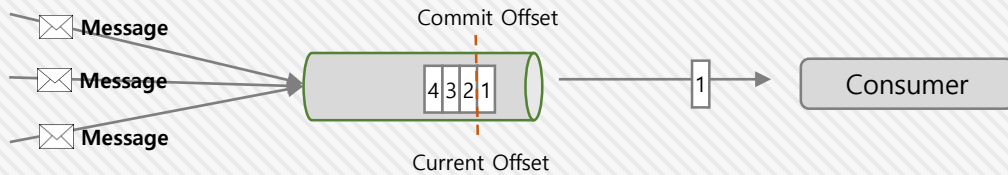
### Kafka 는 4가지 코어 API

- Producer API : 하나 또는 그 이상의 Kafka Topics 스트림을 Publish
- Consumer API : 하나 이상의 Topic을 구독하고 생성된 레코드 스트림을 처리하는 API
- Streams API : 응용 프로그램이 하나 이상의 주제에서 입력 스트림을 소비하고 하나 이상의 출력 항목으로 출력 스트림을 생성하여 효과적으로 입력 스트림을 출력 스트림으로 변환하는 스트림 프레임워크로 작동 하도록 한다.
- Connector API : Connector API를 사용하면 Kafka 주제를 기존 응용 프로그램이나 데이터 시스템에 연결하는 재사용 가능한 제작자 또는 소비자를 만들고 실행할 수 있다.  
예를 들어, 관계형 데이터베이스에 대한 커넥터는 테이블에 대한 모든 변경 사항을 캡처 할 수 있다.

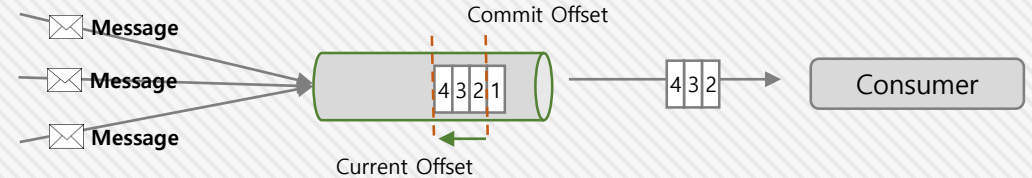
# 3. 컨슈머 장애 대응

Offset Commit의 구조를 이용해 컨슈머 처리 실패, 고장 시 롤백 메시지 재 취득을 하여야 함.  
- Commit Offset update 직전 고장의 경우 : 중복 메시지가 수신 될 수 있으므로 고려 해야 함 ( At Least Once )

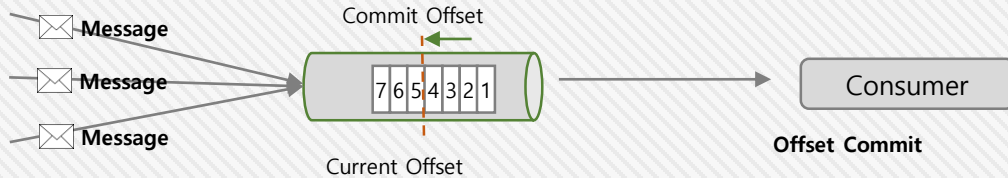
1. Offset 1 까지 수신. Offset Commit이 끝난 상태



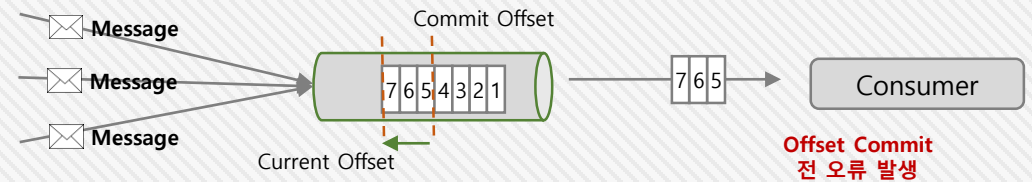
2. Offset 2,3,4 수신



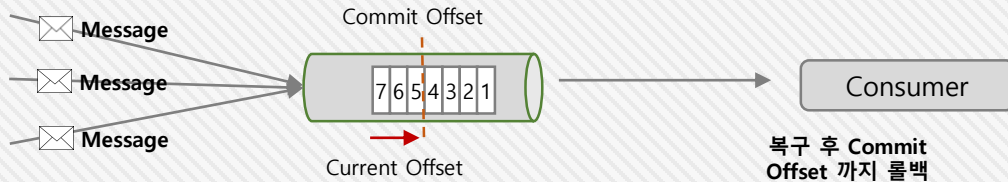
3. 컨슈머 Offset Commit 실행



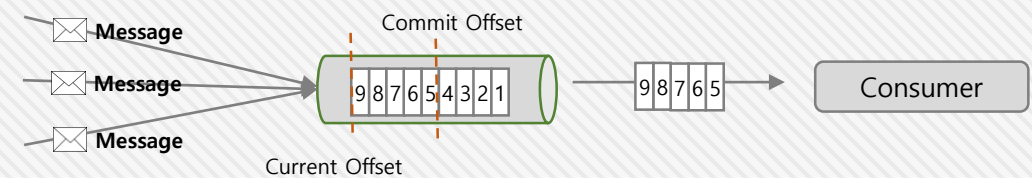
4. 컨슈머 Offset 5,6,7 수신 후 처리 중 장애 발생



5. 컨슈머 장애 복구 되면 Commit Offset 위치 부터 재개



6. 컨슈머 메시지 재 취득



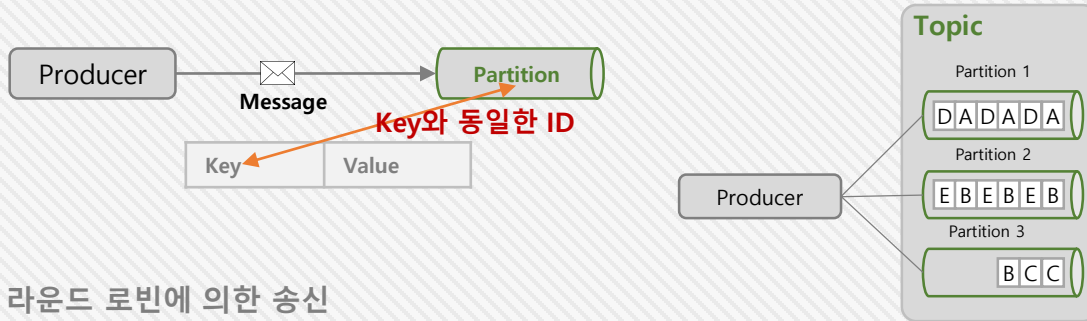
# 4. 파티셔닝 및 보관 주기

토픽 당 데이터를 분산 처리하는 단위로 토픽 안에 파티션을 나누어 그 수대로 데이터를 분산 처리.  
카프카 옵션에서 지정한 replica의 수만큼 파티션이 각 서버들에게 복제됨.

## 01. 프로듀서 파티셔닝 : 메시지에 포함된 key의 명시적인 지정 여부에 따라 두가지 패턴

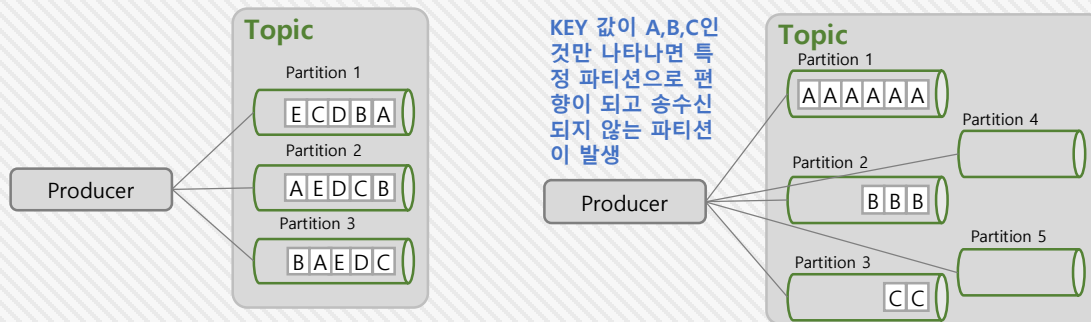
### ▪ Key 해시 값을 사용한 송신

메시지 key를 명시적으로 지정 함으로 Key에 따라서 파티션을 결정 하는 로직  
동일한 Key를 가진 메시지는 동일한 ID를 가진 파티션에 송신 됨.



### ▪ 라운드 로빈에 의한 송신

메시지 Key를 지정 하지 않고 null로 하면 여러 파티션에 라운드 로빈 방식으로 송신



- DefaultPartitioner Class이용해서 구현
- Producer API에서 제공 하는 Partitioner 인터페이스를 구현함으로써 Key,Value 값에 따라서 송신 로직을 커스텀으로 구현

## 02. 브로커 데이터 보관 기간

카프카는 수신된 메시지를 디스크에 보관(영속화)하고 컨슈머는 보관되어 있는 메시지를 읽어 들이는 구조로 보관된 데이터는 메시지 취득 후 경과 시간, 데이터 크기 두가지 정책으로 설정함.

### ▪ 데이터 취득 후 경과 시간을 트리거 하는 경우

- 시간, 분, 밀리초 등으로 지정, 시정한 시간 보다 오래된 데이터 삭제
- 기본 168시간 (1주)

### ▪ 데이터 크기를 트리거 하는 경우

- 지정한 데이터 크기보다 커진 경우 데이터 삭제
- 기본 -1 ( 크기 제한 없음 )

- \* 압축 : 최신 Key의 데이터를 남겨두고 중복하는 Key의 오래된 메시지 삭제
- \* **cleanup.policy : delete 또는 compact 값을 이용해서 설정 함**

## 03. 파티션

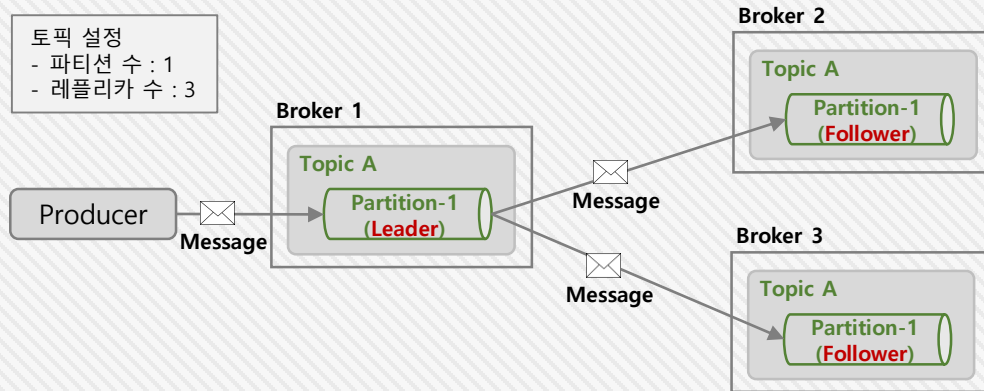
- 각 토픽마다 데이터를 여러개의 파티션에 나누어서 저장/처리
- 토픽 사이즈가 커질 경우 파티션을 늘려서 스케일아웃을 할 수 있다
- 일반적으로 브로커 하나당 파티션 하나
- 컨슈머 인스턴스 수는 파티션 갯수를 넘을 수 없다. 그러므로 병렬처리의 수준은 파티션 수에 의해 결정된다. 즉 파티션이 많을 수록 병렬처리 정도가 높아진다.
- 각 파티션마다 Publish되는 레코드에 고유 오프셋을 부여한다. 때문에 레코드는 파티션 내에서는 유니크하게 식별된다. 하지만 파티션간에는 순서를 보장하지 않는다.
- 전체 순서를 보장하고 싶으면 파티션을 하나만 두는 수 밖에 없는데, 이러면 데이터량이 많아져도 스케일아웃이 안되고, 컨슈머 인스턴스도 하나만 둘 수 있으므로 병렬 컨슈밍도 안된다.

# 5. 복제 구조 ( 데이터 견고성 )

카프카는 수신된 메시지를 잃지 않기 위해서 복제 구조로 갖추고 있음.

## 01. 복제

파티션은 단일 또는 여러 개의 레플리카로 구성 되어 토픽 단위로 레플리카 수를 지정 Leader와 Follower라 나뉘며, **Leader**는 **Producer**와 **Consumer**와의 데이터 교환 역할을 담당 하고 **Follower**는 **Leader**로 부터 메시지를 받아서 복제를 유지 하는 기능을 담당함



## 02. 메시지 순서 보장

카프카는 기본적으로 파티션이 여러 개로 구성되어 있어 컨슈머가 메시지를 받는 시점에 따라서 메시지 생성 시기와 프로듀서 송신 순서가 바뀌는 경우가 존재 함  
-> 처리기를 1개만 두면 병렬성이 떨어진다

카프카는 동일 partition 내에서만 순서를 보장 (파티션당 컨슈머를 하나만 가져야하는 이유는 파티션 내에서 처리순서를 보장하기 위함) -> 전체가 아닌 부분 -> 병렬성 보장

순서 보증을 위한 정렬 기능을 브로커에서 구현 할지, 컨슈머에서 구현 할 지 시스템 전체를 고려 하여 판단 해야 함

## 03. 브로커 데이터 보관 기간

카프카는 수신된 메시지를 디스크에 보관(영속화)하고 컨슈머는 보관되어 있는 메시지를 읽어 들이는 구조로 보관된 데이터는 메시지 취득 후 경과 시간, 데이터 크기 두가지 정책으로 설정함.

### ■ 데이터 취득 후 경과 시간을 트리거 하는 경우

- 시간, 분, 밀리초 등으로 지정, 시정한 시간 보다 오래된 데이터 삭제
- 기본 168시간 (1주)

### ■ 데이터 크기를 트리거 하는 경우

- 지정한 데이터 크기보다 커진 경우 데이터 삭제
- 기본 -1 ( 크기 제한 없음 )

- \* 압축 : 최신 Key의 데이터를 남겨두고 중복하는 Key의 오래된 메시지 삭제
- \* **cleanup.policy : delete 또는 compact 값을 이용해서 설정 함**

## 04. 복제 상태

- **In-Sync Replica** : Leader Replica의 복제 상태를 유지하고 있는 레플리카
- **Under Replicated Partitions** : In-Sync Replica로 되어 있지 않은 파티션

- **replica.lag.time.max.ms** 에서 정한 시간 보다도 오랫동안 복제의 요청 및 복제가 이루어 지지 않을 경우 복제 상태를 유지하지 않는 것으로 간주
- **Min.insync.replica** : 복제 수와 독립적으로 최소 ISR 수 설정

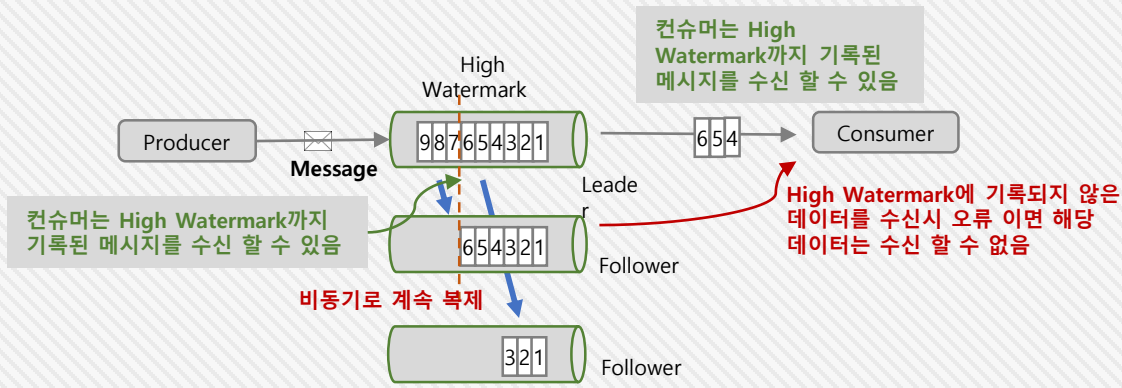


# 5. 복제 구조 ( 데이터 견고성 )

카프카는 수신된 메시지를 잃지 않기 위해서 복제 구조로 갖추고 있음.

## 05. High Watermark

- 복제가 완료된 오프셋
- Log End Offset과 동일하거나 오래된 Offset를 나타냄
- 컨슈머는 High Watermark까지 기록된 메시지를 수신 할 수 있음



## 06. Producer 메시지 송신 시 Ack 설정

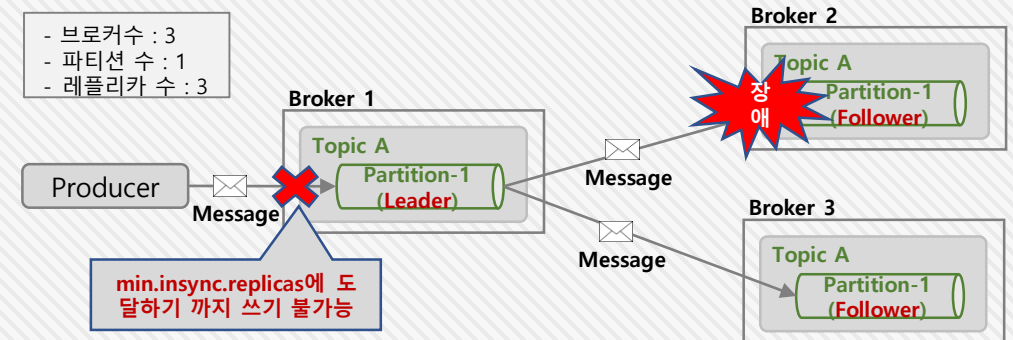
- 브로커에서 프로듀서로 메시지 송신된 것을 나타내는 Ack 설정 값에 따라서 성능과 내장애성(브로커 고장 시 데이터 분실 방지)에 큰 영향을 줌

종류	설명
0	Producer는 메시지 송신시 Ack를 기다리지 않고 다음 메시지를 송신
1	Leader Replica에 메시지가 전달 되면 Ack 반환
all	모든 ISR의 수만큼 복제 되면 Ack를 반환

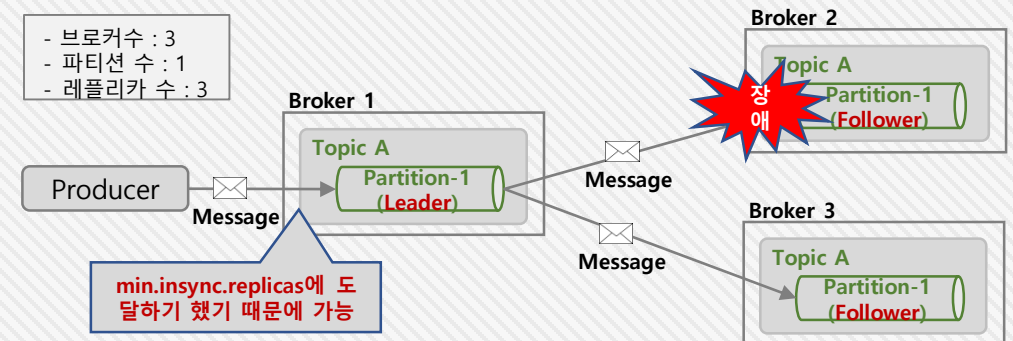
- 반환 타이밍은 메모리에 적재 이며 디스크에 Flush는 아님 ( 별도의 속성으로 설정 )

## 07. In-Sync Replica와 Ack 관계

- `min.insync.replicas = 3, Ack = all , Replica = 3`



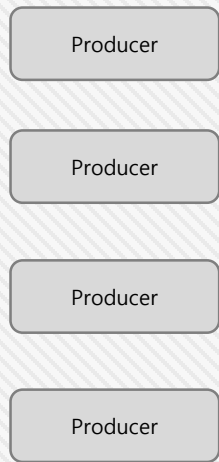
- `min.insync.replicas = 2, Ack = all , Replica = 3`





# 6. Kafka 내부 구조

## Producer

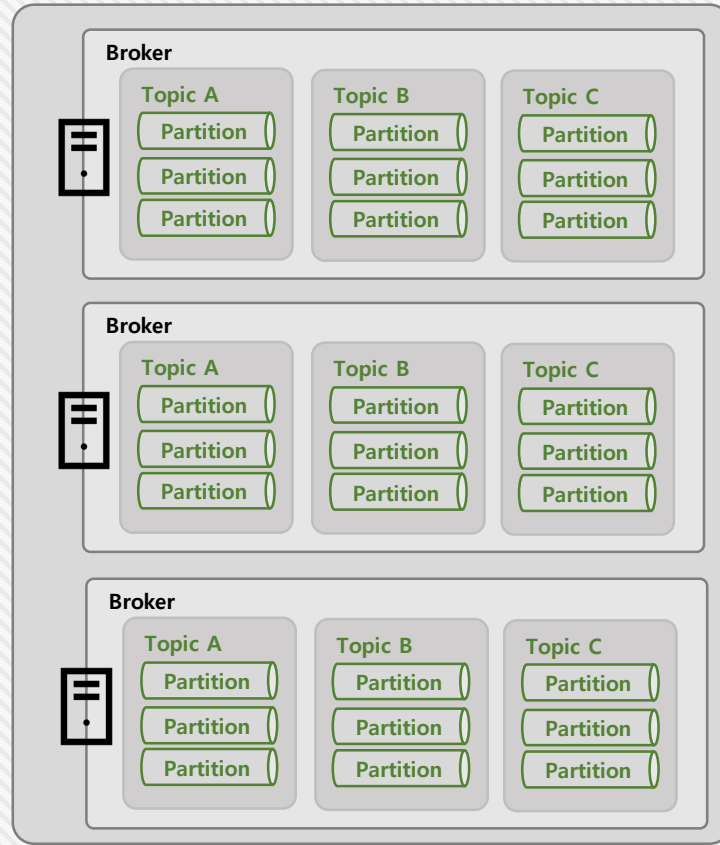


### 연계 제품

- Apache Log4j
- Apache Flume
- Fluentd
- Logstash
- .....

메시지 도달 보증  
- Ack, Offset Commit

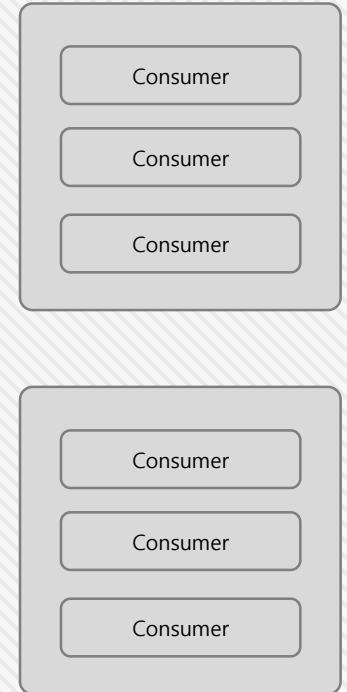
## Kafka Cluster



데이터 영속화  
- 과거 데이터 저장, 재 수신 기능

Scale Out  
- 전체 처리량 증가

## Consumer ( Group )



### 연계 제품

- Apache Spark
- Apache Flink
- Apache Flume
- Fluentd
- Logstash
- .....

# 7. Kafka 구성

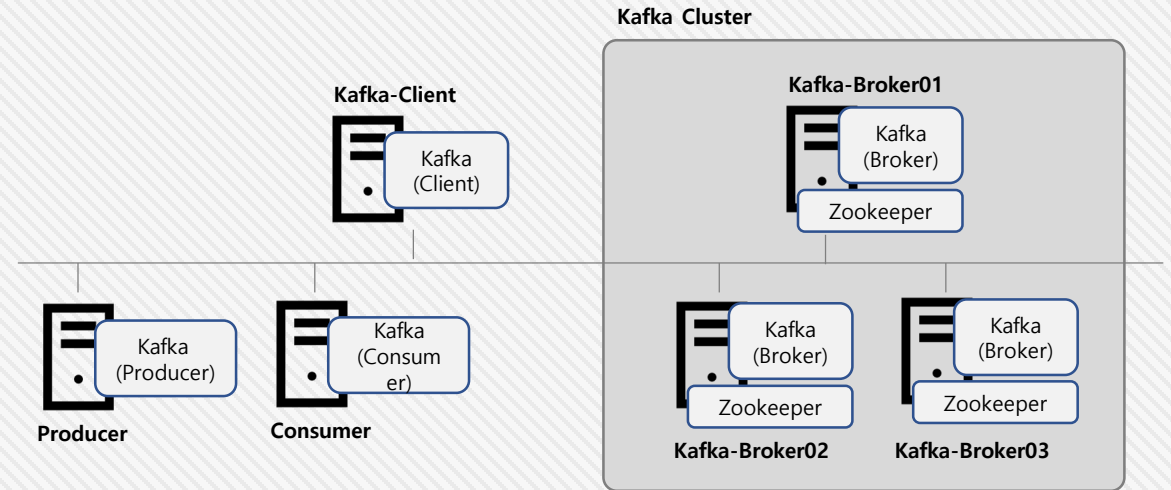
## 01. Single

- 브로커, 프로듀서, 컨슈머, 카프카 클라이언트의 모든 역할 담당



## 02. Cluster

- 브로커, 프로듀서, 컨슈머, 카프카 클라이언트는 각각의 서버에서 담당



- Zookeeper는 데이터 쓰기가 과반수 서버에 성공했을 때 성공으로 간주하므로 홀수 노드 수가 바람직하며, Kafka와 동일 서버에 설치 여부는 시스템 요구 사항에 따라서 달라진다.

# 8. Kafka 설치

## 01. 설치

- ① Kafka 다운로드 : [http://mirror.navercorp.com/apache/kafka/2.6.0/kafka\\_2.13-2.6.0.tgz](http://mirror.navercorp.com/apache/kafka/2.6.0/kafka_2.13-2.6.0.tgz)
- ② 압축 해제  
\$ `tar -xzf kafka_2.13-2.6.0.tgz`  
\$ `cd kafka_2.13-2.6.0`

## 02. 서버 실행

- ① Zookeeper 실행 : `> bin/zookeeper-server-start.sh config/zookeeper.properties`
- ② Kafka 실행 : `> bin/kafka-server-start.sh config/server.properties`

## 03. 토픽 생성 및 확인

- ① 토픽 생성 :  
`> bin/kafka-topics.sh --create --topic firstTopic --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1`  
  
--topic : 토픽 이름  
--partitions : 파티션 수  
--replication-factor : 복제 수  
--create : 토픽 생성  
--list : 토픽 목록 확인  
--describe : 토픽 상세 확인
- ① 토픽 확인 :  
`> bin/kafka-topics.sh --list --bootstrap-server localhost:9092`  
`> bin/kafka-topics.sh --describe --topic firstTopic --bootstrap-server localhost:9092`  
  
Topic: firstTopic    PartitionCount: 1    ReplicationFactor: 1    Configs: segment.bytes=1073741824  
Topic: firstTopic    Partition: 0    Leader: 0    Replicas: 0    Isr: 0

## 04. 이벤트 발행

- ① 이벤트 발행 :  
`> bin/kafka-console-producer.sh --topic firstTopic --bootstrap-server localhost:9092`

first Message  
second Message

Ctrl-C로 중지

--broker-list : 브로커의 호스트명:포트번호, 호스트명:포트번호 형식으로 카프카 클러스터 내 모든 브로커 리스트를 입력  
--topic : 메시지를 보내고자 하는 토픽 이름

## 05. 이벤트 읽기

- ① 이벤트 읽기 :  
`> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic StudyTopic --from-beginning`

Ctrl-C로 중지

--from-beginning : 처음부터 메시지를 가지고오도록 설정

# 9. 멀티 클러스터 구성

## 01. 설정 파일 구성

- ① **server.properties**를 복사 한다.
- > cp config/server.properties config/server-1.properties
  - > cp config/server.properties config/server-2.properties

- ② **설정파일 수정**
- config/server-1.properties:
- broker.id=1
  - listeners=PLAINTEXT://:9093
  - log.dirs=/tmp/kafka-logs-1

config/server-2.properties:

- broker.id=2
- listeners=PLAINTEXT://:9094
- log.dirs=/tmp/kafka-logs-2

\*\* PLAINTEXT : 리스너가 인증없이 암호화되지 않음을 의미

## 02. 카프카 브로커 실행 및 토픽 생성

- ① > bin/kafka-server-start.sh config/server-1.properties &
- ② > bin/kafka-server-start.sh config/server-2.properties &
- ③ > bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 3 --partitions 1 --topic my-replicated-topic
- ④ > bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic my-replicated-topic
- Leader : 토픽의 리더
  - Replicas : 토픽이 리플리케이션 되고 있는 브로커 위치
  - Isr(In Sync Replica) : 현재 리플리케이션되고 있는 리플리케이션 그룹, ISR 그룹에 속해 있는 구성원만이 리더의 자격을 가질수 있음

## 03. 카프카 토픽 발행 및 소비

- ① > bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-replicated-topic
- ② > bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-topic

## 04. 카프카 브로커 중지 (Fault tolerance 테스트)

- ① > ps aux | grep server-1.properties
- ② > kill -9 XXXX
- ③ > bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic my-replicated-topic
- ④ > bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-topic

# 10. Kafka Connect

## 01. 테스트 데이터 생성

① > `echo -e "fooWnbar" > test.txt`

## 02. 커넥트 실행

① > `bin/connect-standalone.sh config/connect-standalone.properties config/connect-file-source.properties config/connect-file-sink.properties`

## 03. 컨슈머 실행

① > `bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic connect-test --from-beginning`

## 04. 데이터 추가

① > `echo Another line>> test.txt`

# 1. Kafka Producer

Kafka 토픽에 메시지를 게시하는 Kafka 생산자와 해당 메시지를 읽는 Kafka 소비자가 있는 JAVA 애플리케이션.

## 01. Maven Project 생성

- STS or IntelliJ를 이용해서 Maven Project를 생성 한다.  
: Spring Boot를 사용 하지만 기본만으로 구성 함
- pom.xml에 의존성을 작성 한다.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>

<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.6.0</version>
</dependency>
```

## 02. 메시지 생성 객체

- Kafka 환경 설정을 위한 Properties 객체를 멤버 변수 를 선언 한다.

```
private Properties conf ;
```

- 선언한 Properties 변수에 Kaka 환경을 설정 한다

▪ .

```
private void setKafkaProperties() {
  // Kafka Producer에 필요한 설정
  this.conf = new Properties();
  // Kafka Producer가 접속 하는 브로커의 호스명과 포트
  // 예) kafka-broker01:9092, kafka-broker02:9092, kafka-broker03:9092
  this.conf.setProperty("bootstrap.servers", "localhost:9092");

  // kafka의 모든 메시지는 직렬화된 상태로 전송하므로 직렬화 처리에 이용되는
  시리얼라이저 클래스를 key와 Value로 각각 설정 함
  this.conf.setProperty("key.serializer", "org.apache.kafka.common.serialization.IntegerSerializer" );
  conf.setProperty("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
}
```

# 1. Kafka Producer

Kafka 토픽에 메시지를 게시하는 Kafka 생산자와 해당 메시지를 읽는 Kafka 소비자가 있는 JAVA 애플리케이션.

## 03. Producer 구현

- ① Kafka 환경 설정
- ② Kafka Cluster에 메시지를 송신하는 KafkaProducer 객체 생성
- ③ Kafka Cluster에 송신할 메시지 ProducerRecord 객체 생성
- ④ Kafka Cluster에 메시지 송신  
: Callback 객체를 인자로 생성 하여 성공 한 경우와 실패한 경우에 대해서 처리를 함.
- ⑤ 객체 Close

```
// 1. kafka 환경 설정
setKafkaProperties();

// 2. Kafka Cluster에 메시지를 송신하는 객체 생성
Producer<Integer, String> producer = new KafkaProducer<>(this.conf);

int key;
String value;
for ( int i = 1; i <= loopcnt; i++) {
    key = i;
    value = "메세지 :: " + String.valueOf(i);

// 3. 송신할 메세지 생성
    ProducerRecord<Integer, String> record = new ProducerRecord<>(topicName, key, value);

// 4. 메세지 송신
    producer.send(record, new Callback() {
        @Override
        public void onCompletion(RecordMetadata recordMetadata, Exception e) {
            if (recordMetadata != null) {
                // 송신에 성공한 경우의 처리
                String infoString = String.format("Success partition:%d, offset:%d",
                    recordMetadata.partition(),
                    recordMetadata.offset());
                System.out.println(infoString);
            } else {
                // 송신에 실패한 경우의 처리
                String infoString = String.format("Failed:%s", e.getMessage());
                System.err.println(infoString);
            }
        }
    });
}

// 5. 객체 Close
producer.close();
```



# 1. Kafka Producer

Kafka 토픽에 메시지를 게시하는 Kafka 생산자와 해당 메시지를 읽는 Kafka 소비자가 있는 JAVA 애플리케이션.

## 04. Spring Boot ApplicationRunner을 이용한 실행

```
@SpringBootApplication
public class KafkaApplication {

    public static void main(String[] args) {
        SpringApplication.run(KafkaApplication.class, args);
    }

    @Bean
    public ApplicationRunner KafkaProducerRun() {
        return args -> {
            KafkaFirstProducer kafkaProducer = new KafkaFirstProducer();
            kafkaProducer.kafkaPublish("StudyTopic", 5);
        };
    }
}
```

```
.....
2020-10-03 20:06:40.493 INFO 6411532 --- [main] o.a.kafka.common.utils.AppInfoParser : Kafka version: 2.6.0
2020-10-03 20:06:40.495 INFO 6411532 --- [main] o.a.kafka.common.utils.AppInfoParser : Kafka commitId: 62abe01bee039651
2020-10-03 20:06:40.496 INFO 6411532 --- [main] o.a.kafka.common.utils.AppInfoParser : Kafka startTimeMs: 1601723200489
Kafka Cluster에 메시지를 송신하는 객체 생성
Kafka Cluster :: 메시지 :: 1
2020-10-03 20:06:41.160 INFO 6411532 --- [ad | producer-1] org.apache.kafka.clients.Metadata : [Producer clientId=producer-1] Cluster ID: poFI6cn4SHKAUHgYjx6yZQ
Kafka Cluster :: 메시지 :: 2
Kafka Cluster :: 메시지 :: 3
Kafka Cluster :: 메시지 :: 4
Kafka Cluster :: 메시지 :: 5
2020-10-03 20:06:41.182 INFO 6411532 --- [main] o.a.k.clients.producer.KafkaProducer : [Producer clientId=producer-1] Closing the Kafka producer with timeoutMillis = 9223372036854775807 ms.
Success partition:0, offset:1241
Success partition:0, offset:1242
Success partition:0, offset:1243
Success partition:0, offset:1244
Success partition:0, offset:1245
```

# 1. Kafka Consumer

Kafka 토픽에 메시지를 게시하는 Kafka 생산자와 해당 메시지를 읽는 Kafka 소비자가 있는 JAVA 애플리케이션.

## 01. Maven Project 생성

- STS or IntelliJ를 이용해서 Maven Project를 생성 한다.  
: Spring Boot를 사용 하지만 기본만으로 구성 함
- pom.xml에 의존성을 작성 한다.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>

<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.6.0</version>
</dependency>
```

## 02. 메시지 생성 객체

- Kafka 환경 설정을 위한 Properties 객체를 멤버 변수 를 선언 한다.

선언한 Properties 변수에 Kaka 환경을 설정 한다.

: **bootstrap.servers** : Kafka Cluster Server 지정  
: **goup.id** : Consumer Group 지정  
: **enable.auto.commit** : Offset Commit 자동 여부 설정  
: **key.deserializer** : 직렬화 처리에 이용되는 시리얼라이저 클래스  
: **value.deserializer** : 직렬화 처리에 이용되는 시리얼라이저 클래스

```
private Properties conf ;
```

```
private void setKafkaProperties() {
  // KafkaConsumer에 필요한 설정
  conf = new Properties();
  conf.setProperty("bootstrap.servers", "localhost:9092");
  conf.setProperty("group.id", "FirstAppConsumerGroup");
  conf.setProperty("enable.auto.commit", "false");
  conf.setProperty("key.deserializer", "org.apache.kafka.common.serialization.IntegerDeserializer");
  conf.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
}
```

# 1. Kafka Consumer

Kafka 토픽에 메시지를 게시하는 Kafka 생산자와 해당 메시지를 읽는 Kafka 소비자가 있는 JAVA 애플리케이션.

## 03. consumer 구현

- ① Kafka 환경 설정
- ② Kafka Cluster에 메시지를 수신하는 KafkaConsumer 객체 생성
- ③ Kafka Cluster에 수신(subscribe)하는 Topic을 등록
- ④ Kafka Cluster에 메시지 수신  
: Callback 객체를 인자로 생성 하여 성공 한 경우와 실패한 경우에 대해서 처리를 함.
- ⑤ 처리가 완료한 Message의 Offset을 Commit
- ⑥ 객체 Close

```
// 1.환경설정
setKafkaProperties();

// 2. Kafka클러스터에서 Message를 수신(Consume)하는 객체를 생성
Consumer<Integer, String> consumer = new KafkaConsumer<>(conf);

// 3. 수신(subscribe)하는 Topic을 등록
consumer.subscribe(Collections.singletonList(topicName));

for(int count = 0; count < cnt; count++) {
    // 4. Message를 수신하여, 콘솔에 표시한다
    ConsumerRecords<Integer, String> records = consumer.poll(1);
    for(ConsumerRecord<Integer, String> record: records) {
        String msgString = String.format("key:%d, value:%s", record.key(), record.value());
        System.out.println(msgString);

        // 5. 처리가 완료한 Message의 Offset을 Commit한다
        TopicPartition tp = new TopicPartition(record.topic(), record.partition());
        OffsetAndMetadata oam = new OffsetAndMetadata(record.offset() + 1);
        Map<TopicPartition, OffsetAndMetadata> commitInfo = Collections.singletonMap(tp, oam);
        consumer.commitSync(commitInfo);
    }
    System.out.println("End .....");
    try {
        Thread.sleep(1000);
    } catch (InterruptedException ex ) {
        ex.printStackTrace();
    }
}

// 6. KafkaConsumer를 클로즈하여 종료
consumer.close();
```

# 1. Kafka Consumer

Kafka 토픽에 메시지를 게시하는 Kafka 생산자와 해당 메시지를 읽는 Kafka 소비자가 있는 JAVA 애플리케이션.

## 04. Spring Boot ApplicationRunner을 이용한 실행

```
@SpringBootApplication
public class KafkaApplication {

    public static void main(String[] args) {
        SpringApplication.run(KafkaApplication.class, args);
    }

    @Bean
    public ApplicationRunner KafkaProducerRun() {
        return args -> {
            KafkaFirstConsumer kafkaConsumer = new KafkaFirstConsumer();
            kafkaConsumer.kafkaSubscriber("StudyTopic", 5);
        };
    }

}
```

```
2020-10-03 20:27:04.867 INFO 6737784 --- [main] o.a.k.c.c.internals.ConsumerCoordinator : [Consumer clientId=consumer-FirstAppConsumerGroup-1, groupId=FirstAppConsumerGroup] Setting offset for partition StudyTopic-0 to the
committed offset FetchPosition(offset=1246, offsetEpoch=Optional[0], currentLeader=LeaderAndEpoch{leader=Optional[localhost:9092 (id: 1 rack: null)], epoch=0})
End .....
End .....
End .....
End .....
key:1, value:메세지 :: 1
key:2, value:메세지 :: 2
key:3, value:메세지 :: 3
key:4, value:메세지 :: 4
key:5, value:메세지 :: 5
End .....
2020-10-03 20:27:09.968 INFO 6737784 --- [main] o.a.k.c.c.internals.ConsumerCoordinator : [Consumer clientId=consumer-FirstAppConsumerGroup-1, groupId=FirstAppConsumerGroup] Revoke previously assigned partitions StudyTopic-0
2020-10-03 20:27:09.968 INFO 6737784 --- [main] o.a.k.c.c.internals.AbstractCoordinator : [Consumer clientId=consumer-FirstAppConsumerGroup-1, groupId=FirstAppConsumerGroup] Member consumer-FirstAppConsumerGroup-1-
b0ea2755-797d-4cfe-b7e7-c938c951b6dd sending LeaveGroup request to coordinator localhost:9092 (id: 2147483646 rack: null) due to the consumer is being closed
```

# 1. Spring Boot Kafka

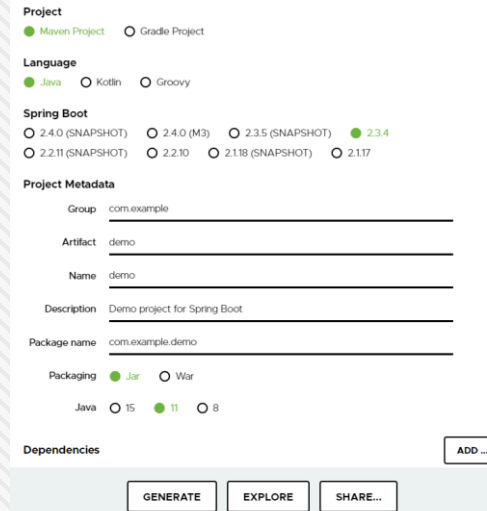
2. KAFKA 예제

2. Spring Boot

Kafka 토픽에 메시지를 게시하는 Kafka 생산자와 해당 메시지를 읽는 Kafka 소비자가 있는 Spring Boot 애플리케이션.

## 01. 프로젝트 생성

- ① Spring Initializr 로 이동하여 프로젝트를 생성  
: <https://start.spring.io/>



The image shows the Spring Initializr web form for creating a new project. The 'Project' section has 'Maven Project' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '2.3.4' selected. The 'Project Metadata' section has 'Group' as 'com.example', 'Artifact' as 'demo', 'Name' as 'demo', and 'Description' as 'Demo project for Spring Boot'. The 'Package name' is 'com.example.demo'. The 'Packaging' section has 'Jar' selected. The 'Java' version is '11'. There is an 'ADD ...' button and 'GENERATE', 'EXPLORE', and 'SHARE...' buttons at the bottom.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
  </dependency>

  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>
</dependencies>
```

## 02. 구성 application.yml파일을 통해 Kafka 구성

- ① Kafka 생산자와 소비자가 토픽에서 메시지를 발행하고 읽을 수 있도록 구성  
: Java 클래스를 생성하고 @Configuration 대신 Spring 환경 설정을 이용해서 구성

```
server: port: 9000
spring:
  kafka:
    consumer:
      bootstrap-servers: localhost:9092
      group-id: group_id
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    producer:
      bootstrap-servers: localhost:9092
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer>
```

# 1. Spring Boot Kafka

2. KAFKA 예제

2. Spring Boot

Kafka 토픽에 메시지를 게시하는 Kafka 생산자와 해당 메시지를 읽는 Kafka 소비자가 있는 JAVA 애플리케이션.

## 03. Producer 구현

- ① kafkaTemplate 인스턴스를 사용하여 주제에 메시지를 게시

```
@Service
public class Producer {

    private static final Logger logger = LoggerFactory.getLogger(Producer.class);
    private static final String TOPIC = "StudyTopic";

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public void sendMessage(String message) {
        logger.info(String.format("####->Producer Message-> %s", message));
        this.kafkaTemplate.send(TOPIC, message);
    }
}
```

## 04. Consumer 구현

- ① KafkaListener를 사용 해서 메시지 읽기

```
@Service
public class Consumer {

    private static final Logger logger = LoggerFactory.getLogger(Consumer.class);

    @KafkaListener(topics = "StudyTopic", groupId = "group_id")
    public void consume (String message) throws IOException {
        logger.info(String.format("#### -> Consumed message -> %s", message));
    }
}
```

```
2020-10-03 20:45:06.545 INFO 4482684 --- [ntainer#0-0-C-1] com.hyomee.kafka.service.Consumer : #### -> Consumed message -> 메세지 :: 1
2020-10-03 20:45:06.546 INFO 4482684 --- [ntainer#0-0-C-1] com.hyomee.kafka.service.Consumer : #### -> Consumed message -> 메세지 :: 2
2020-10-03 20:45:06.547 INFO 4482684 --- [ntainer#0-0-C-1] com.hyomee.kafka.service.Consumer : #### -> Consumed message -> 메세지 :: 3
2020-10-03 20:45:06.547 INFO 4482684 --- [ntainer#0-0-C-1] com.hyomee.kafka.service.Consumer : #### -> Consumed message -> 메세지 :: 4
2020-10-03 20:45:06.594 INFO 4482684 --- [ntainer#0-0-C-1] com.hyomee.kafka.service.Consumer : #### -> Consumed message -> 메세지 :: 5
```

# 1. Spring Boot Kafka

2. KAFKA 예제

2. Spring Boot

참고 : <https://www.baeldung.com/spring-kafka#consuming-messages>

## 05. REST 컨트롤러 생성

```
@RestController
@RequestMapping( value = "/kafka")
public class KafkaController {

    private Producer producer;

    @Autowired
    KafkaController(Producer producer) {
        this.producer = producer;
    }

    @PostMapping(value="/publish")
    public void sendMessageToKafkaTpoic(@RequestParam("message") String message) {
        this.producer.sendMessage(message);
    }
}
```

POST localhost:9000/kafka/publish Send Save

Params Auth Headers (9) **Body** Pre-req. Tests Settings Cookies Code

form-data

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	message	메세지 문제가 있나			
	Key	Value	Description		

Body 200 OK 489 ms 123 B Save Response

Pretty Raw Preview Visualize Text 🔍

```
2020-10-03 20:52:07.456 INFO 4482684 --- [nio-9000-exec-4]
com.hyomee.kafka.service.Producer : ###->Producer Message-> 메세지 문제가 있나
2020-10-03 20:52:07.625 INFO 4482684 --- [ntainer#0-0-C-1]
com.hyomee.kafka.service.Consumer : ##### -> Consumed message -> 메세지 문제가 있나
```



# 1. Spring Boot Kafka – 환경설정

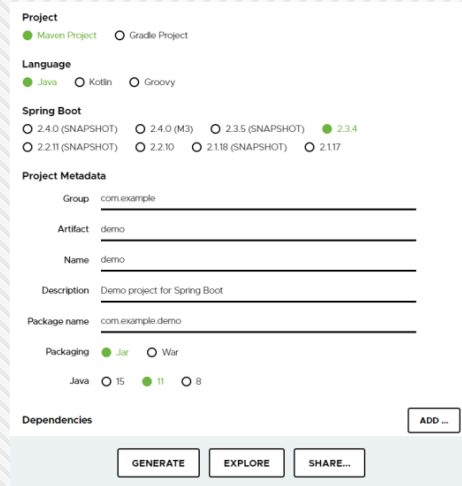
2. KAFKA 예제

3. Spring을 사용한 Apache Kafka

## Intro to Apache Kafka with Spring 에서 작성된 소스 설명

### 01. 프로젝트 생성

- ① Spring Initializr 로 이동하여 프로젝트를 생성  
: <https://start.spring.io/>



The image shows the Spring Initializr web form. It is configured for a Maven Project in Java, using Spring Boot 2.3.4. The project metadata includes Group: com.example, Artifact: demo, Name: demo, Description: Demo project for Spring Boot, and Package name: com.example.demo. The packaging is set to Jar, and the Java version is 11. There are buttons for GENERATE, EXPLORE, and SHARE... at the bottom.

```
<dependencies>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>

<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
```

### 02. application.yml파일을 통해 Kafka 구성

- ① server : Tomcat 포트 설정  
② Kafka : kafka 서버 설정  
③ topic : topic 설정
- message.name : 일반 메시지 토픽
  - modelVo.name : 자바 객체 메시지를 위한 토픽
  - filtered.name : 메시지를 받아서 일부 메시지만 허용 하기 위한 토픽
  - partitioned.name : 일부 파티션에서만 데이터를 받기 위한 토픽
- ④ consumerGroup : 컨슈머에서 받은 그룹
- hyomeeGrp : 일반메시지 그룹
  - headerGrp : 헤더 그룹 그룹
  - partitionsGrp : 파티션 예제 그룹
  - filterGrp : 필터 예제 그룹
  - modelVoGrp : 자바 객체 예제 그룹
- ⑤ Filter : 제외 필터  
⑥ logging : 로그 레벨

```
server:
  port: 9000
kafka:
  bootstrapAddress: localhost:9092

topic:
  message:
    name: hyomee
  modelVo:
    name: modelvo
  filtered:
    name: filtered
  partitioned:
    name: partitioned

consumerGroup:
  hyomeeGrp: hyomee
  headerGrp: headers
  partitionsGrp: partitions
  filterGrp: filter
  modelVoGrp: greeting

filter: hi

logging:
  level:
    root: ERROR ##
    org.springframework: ERROR
    org.springframework.web: INFO
    org.hibernate.SQL: DEBUG
    com.hyomee : DEBUG
```

# 2. Spring Boot Kafka – Topic 생성

2. KAFKA 예제

3. Spring을 사용한 Apache Kafka

## Kafka에 AdminClient를 사용한 Topic 생성

### 03 KafkaAdmin Spring Bean 주입

- Spring Boot를 사용하면 KafkaAdmin빈이 자동으로 등록  
- application.yml 에 서버 설정 필요  
: spring.kafka.bootstrap-servers=localhost:9092

- 수동등록

- ① 새로운 토픽을 등록 하기 위해서 KafkaAdmin 주입  
- application.yml 에 서버 설정 필요  
kafka:  
bootstrapAddress: localhost:9092

- ② 새로운 토픽 생성

```
topic:  
  message:  
    name: hyomee  
  modelVo:  
    name: modelvo  
  filtered:  
    name : filtered  
  partitioned:  
    name: partitioned
```

```
@Value(value="${kafka.bootstrapAddress}")  
private String bootstrapAddress;  
  
@Value(value="${topic.message.name}")  
private String topicMessageName;
```

\* 환경파일에 있는 속성 읽어 오기

```
@Configurationclass  
KafkaTopicConfig {  
    @Bean  
    public NewTopic topic1() {  
        return TopicBuilder.name("reflectoring-1").build();  
    }  
    @Bean  
    public NewTopic topic2() {  
        return TopicBuilder.name("reflectoring-2").build(); }  
    ...  
}
```

// 1. 새로운 토픽을 등록 하기 위해서 KafkaAdmin 주입

```
@Bean  
public KafkaAdmin kafkaAdmin() {  
    log.info(String.format("#### Bootstrap Address :: %s", bootstrapAddress));  
    Map<String, Object> configs = new HashMap();  
    configs.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);  
    return new KafkaAdmin(configs);  
}
```

// 2. 토픽 생성

```
@Bean  
public NewTopic topic1() {  
    return new NewTopic(topicMessageName, 1, (short) 1);  
}  
  
@Bean  
public NewTopic topic2() {  
    return new NewTopic(topicPartitionedName, 6, (short) 1);  
}
```

# 3. Spring Boot Kafka – 메시지 생성

2. KAFKA 예제

3. Spring을 사용한 Apache Kafka

## 메시지를 생성 하기 위해서는

- ① **ProducerFactory** 를 구성
- ② **Producer** 인스턴스 를 래핑
- ③ **Kafka** 토픽에 메시지를 보내기위한 편리한 메서드를 제공 하는 **KafkaTemplate**

**Producer** 인스턴스는 스레드로부터 안전하므로 애플리케이션 컨텍스트 전체에서 단일 인스턴스를 사용하면 성능이 향상됨  
따라서 **KafkaTemplate** 인스턴스도 스레드로부터 안전하며 하나의 인스턴스를 사용하는 것이 좋음

## 01 Producer 구성

- **ProducerFactoryKafka** **Producer** 인스턴스 생성

**BOOTSTRAP\_SERVERS\_CONFIG** -Kafka가 실행되는 호스트 및 포트.

**KEY\_SERIALIZER\_CLASS\_CONFIG** -키에 사용할 **Serializer** 클래스. (**StringSerializer**)

**VALUE\_SERIALIZER\_CLASS\_CONFIG**-값에 사용할 **Serializer** 클래스. . (**StringSerializer**)

```
@Configuration
public class KafkaProducerConfig {

    @Value(value="${kafka.bootstrapAddress}")
    private String bootstrapAddress;

    // Kafka Producer 인스턴스 생성 전략을 설정 하는 ProducerFactory 를 구성
    @Bean
    public ProducerFactory<String, String> producerFactory() {
        Map<String, Object> configProps = new HashMap<>();
        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        return new DefaultKafkaProducerFactory<>(configProps);
    }

    @Bean
    public KafkaTemplate<String, String> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}
```

# 3. Spring Boot Kafka – 메시지 생성

2. KAFKA 예제

3. Spring을 사용한 Apache Kafka

## 메시지를 생성 하기 위해서는

- ① **ProducerFactory** 를 구성
- ② **Producer** 인스턴스 를 래핑
- ③ **Kafka** 토픽에 메시지를 보내기위한 편리한 메서드를 제공 하는 **KafkaTemplate**

**Producer** 인스턴스는 스레드로부터 안전하므로 애플리케이션 컨텍스트 전체에서 단일 인스턴스를 사용하면 성능이 향상됨  
따라서 **KafkaTemplate** 인스턴스도 스레드로부터 안전하며 하나의 인스턴스를 사용하는 것이 좋음

## 02 메시지 게시

- **KafkaTemplate** 클래스를 사용하여 메시지 보내기

```
@Autowiredprivate
KafkaTemplate<String, String> kafkaTemplate;

public void sendMessage(String msg) {
    kafkaTemplate.send(topicName, msg);
}
```

- **ListenableFuture**을 이용한 결과 받기

```
public void sendTopicMessage(String message) {
    log.debug(String.format("### Topic Test :: Topic=%s, Message=%s", topicMessageName, message));
    ListenableFuture<SendResult<String, String>> future = kafkaTemplate.send(topicMessageName, message);
    future.addCallback(new ListenableFutureCallback<SendResult<String, String>>() {
        @Override
        public void onFailure(Throwable throwable) {
            log.error(String.format("### Send message= [ %s ] :: %s ", message, throwable.getMessage()));
        }

        @Override
        public void onSuccess(SendResult<String, String> result) {
            log.debug(String.format("### Send message= [ %s ] , Offset :: %s , ", message,
            result.getRecordMetadata().offset()));
        }
    });
}
```

# 4. Spring Boot Kafka – 메시지 소비

2. KAFKA 예제

3. Spring을 사용한 Apache Kafka

ConsumerFactory 및 KafkaListenerContainerFactory 를 구성.

Spring Bean Factory에서 이러한 Bean을 사용할 수 있게되면 @KafkaListener 주석을 사용하여 POJO 기반 소비자를 구성 할 수 있다.

@EnableKafka 어노테이션은 스프링 관리 Bean에서 @KafkaListener 어노테이션을 감지 할 수 있도록 함

## 01 Consumer 구성

- ConsumerFactory 및 KafkaListenerContainerFactory 를 구성

BOOTSTRAP\_SERVERS\_CONFIG  
: Kafka가 실행되는 호스트 및 포트.  
GROUP\_ID\_CONFIG  
: 컨슈머 그룹  
KEY\_DESERIALIZER\_CLASS\_CONFIG –  
: 키에 사용할 Serializer 클래스. (StringSerializer)  
VALUE\_DESERIALIZER\_CLASS\_CONFIG –  
: 값에 사용할 Serializer 클래스. (StringSerializer)

```
@EnableKafka
@Configuration
public class KafkaConsumerConfig {

    @Value(value="${kafka.bootstrapAddress}")
    private String bootstrapAddress;

    @Value(value="${consumerGroup.hyomeeGrp}")
    private String hyomeeGrp;

    public ConsumerFactory<String, String> consumerFactory(String groupId) {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        return new DefaultKafkaConsumerFactory<>(props);
    }

    public ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory(String groupId) {
        ConcurrentKafkaListenerContainerFactory<String, String> factory = new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory(groupId));
        return factory;
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String> hyomeeGrpKafkaListenerContainerFactory() {
        return kafkaListenerContainerFactory(hyomeeGrp);
    }
}
```

# 4. Spring Boot Kafka – 메시지 소비

2. KAFKA 예제

3. Spring을 사용한 Apache Kafka

## 02 메시지 소비

- 메시지 소비
  - 복수의 topic에서 받는 경우 ,로 기재 한다. (topics = "topic\_name\_1, topic\_name2");)

```
@KafkaListener(topics = "${topic.message.name}",
                groupId = "${consumerGroup.hyomeeGrp}",
                containerFactory = "hyomeeGrpKafkaListenerContainerFactory")
public void listenHyomeeGrp(String message) {
    log.debug(String.format("#### Topic Test : Topic : %s, GroupId : %s, Message : %s", topicMessageName , hyomeeGrp, message));
}
```

22:06:23.114 [http-nio-9000-exec-1] INFO com.hyomee.kafka.producer.Publisher - ### Topic Test :: Topic=hyomee, Message=123 메세지 문제가 있나

- 메시지 발송

```
@RestController
@RequestMapping( value = "/kafka")
public class KafkaController {

    private Publisher publisher;

    @Autowired
    KafkaController(Publisher publisher) {
        this.publisher = publisher;
    }

    @PostMapping(value="/publish")
    public void sendMessageToKafkaTpoic(@RequestParam("message") String
    message) {
        this.publisher.sendTopicMessage(message);
    }
}
```

```
public void sendTopicMessage(String message) {
    log.debug(String.format("### Topic Test :: Topic=%s, Message=%s", topicMessageName, message));
    ListenableFuture<SendResult<String, String>> future = kafkaTemplate.send(topicMessageName, message);
    future.addCallback(new ListenableFutureCallback<SendResult<String, String>>() {
        @Override
        public void onFailure(Throwable throwable) {
            log.error(String.format("### Send message= [ %s ] :: %s ", message, throwable.getMessage()));
        }

        @Override
        public void onSuccess(SendResult<String, String> result) {
            log.debug(String.format("### Send message= [ %s ] , Offset :: %s , ", message,
            result.getRecordMetadata().offset());
        }
    });
}
```

22:06:23.228 [org.springframework.kafka.KafkaListenerEndpointContainer#0-0-C-1] INFO com.hyomee.kafka.consumer.Subscriber - ##### Topic Test : Topic : hyomee, GroupId : hyomee, Message : 123 메세지 문제가 있나

# 4. Spring Boot Kafka – 메시지 소비

2. KAFKA 예제

3. Spring을 사용한 Apache Kafka

## 02 메시지 소비

- Spring은 또한 리스너의 @Header 주석을 사용하여 하나 이상의 메시지 헤더 검색을 지원

```
@KafkaListener(topics = "${topic.message.name}",
               containerFactory = "headersKafkaListenerContainerFactory")
public void listenWithHeaders(@Payload String message,
                              @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {
    log.debug(String.format("#### Headers Topic : %s, partition : %s, Message : %s", topicMessageName, partition, message));
}
```

```
22:06:23.232 [org.springframework.kafka.KafkaListenerEndpointContainer#3-0-C-1] INFO com.hyomee.kafka.consumer.Subscriber - #### Headers Topic : hyomee, partition : 0, Message : 123 메시지 문제가 있나
```



# 4. Spring Boot Kafka – 메시지 소비

2. KAFKA 예제

3. Spring을 사용한 Apache Kafka

## 02 메시지 소비

- 특정 파티션에서 메시지 사용

```
// 메시지 특정 파티션에 보냄
for (int i = 0; i < 5; i++) {
    publisher.sendMessageToPartition(message + "Hello To Partitioned Topic! :: " + i, i);
}
```

```
public void sendMessageToPartition(String message, int partition) {
    kafkaTemplate.send(topicPartitionedName, partition, null, message);
}
```

```
22:06:23.194 [http-nio-9000-exec-1] INFO com.hyomee.kafka.producer.Publisher - ### Partition Test : Topic :: partitioned , Partition :: 0, Message :: 123 메시지 문제가 있나 Hello To Partitioned Topic! :: 0
22:06:23.219 [http-nio-9000-exec-1] INFO com.hyomee.kafka.producer.Publisher - ### Partition Test : Topic :: partitioned , Partition :: 1, Message :: 123 메시지 문제가 있나 Hello To Partitioned Topic! :: 1
22:06:23.220 [http-nio-9000-exec-1] INFO com.hyomee.kafka.producer.Publisher - ### Partition Test : Topic :: partitioned , Partition :: 2, Message :: 123 메시지 문제가 있나 Hello To Partitioned Topic! :: 2
22:06:23.222 [http-nio-9000-exec-1] INFO com.hyomee.kafka.producer.Publisher - ### Partition Test : Topic :: partitioned , Partition :: 3, Message :: 123 메시지 문제가 있나 Hello To Partitioned Topic! :: 3
22:06:23.225 [http-nio-9000-exec-1] INFO com.hyomee.kafka.producer.Publisher - ### Partition Test : Topic :: partitioned , Partition :: 4, Message :: 123 메시지 문제가 있나 Hello To Partitioned Topic! :: 4
```

```
@KafkaListener(topicPartitions = @TopicPartition(topic = "${topic.partitioned.name}",
    partitions = { "0", "3" } ),
    containerFactory = "partitionsGrpKafkaListenerContainerFactory")
public void listenToPartition(@Payload String message,
    @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {
    log.debug(String.format("#### Partition Topic : %s, partition : %s, Message : %s", topicPartitionedName , partition, message));
}
```

```
partitionOffsets = { @PartitionOffset(partition = "0", initialOffset = "0"),
    @PartitionOffset(partition = "3", initialOffset = "0") },
```

```
22:06:23.232 [org.springframework.kafka.KafkaListenerEndpointContainer#2-0-C-1] INFO com.hyomee.kafka.consumer.Subscriber - #### Partition Topic : partitioned, partition : 0, Message : 123 메시지 문제가 있나 Hello To Partitioned Topic! :: 0
```

```
22:06:23.247 [org.springframework.kafka.KafkaListenerEndpointContainer#2-0-C-1] INFO com.hyomee.kafka.consumer.Subscriber - #### Partition Topic : partitioned, partition : 3, Message : 123 메시지 문제가 있나 Hello To Partitioned Topic! :: 3
```

# 5. Spring Boot Kafka – 메시지 필터

2. KAFKA 예제

3. Spring을 사용한 Apache Kafka

RecordFilterStrategy를 사용 하여 특정 메시지 제외

## 01 Consumer 구성

- 메시지 필터 추가로 메시지 제거를 위한 구성

- application.xml  
filter: hi

```
public ConsumerFactory<String, String> consumerFactory(String groupId) {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
    props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
    return new DefaultKafkaConsumerFactory<>(props);
}

public ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory(String groupId) {
    ConcurrentKafkaListenerContainerFactory<String, String> factory = new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory(groupId));
    return factory;
}

@Bean
public ConcurrentKafkaListenerContainerFactory<String, String> filterKafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, String> factory = kafkaListenerContainerFactory(filterGrp);
    factory.setRecordFilterStrategy(record -> record.value()
        .contains(filter));
    return factory;
}
```

# 5. Spring Boot Kafka – 메시지 필터

2. KAFKA 예제

3. Spring을 사용한 Apache Kafka

## RecordFilterStrategy를 사용 하여 특정 메시지 제외

### 02 메시지 송신/수신

- 메시지 보내기

```
publisher.sendMessageToFiltered(message);  
publisher.sendMessageToFiltered("hi " + message);  
publisher.sendMessageToFiltered( message + " hi!");
```

```
public void sendMessageToFiltered(String message) {  
    kafkaTemplate.send(topicFilteredName, message);  
}
```

```
22:06:23.228 [http-nio-9000-exec-1] INFO com.hyomee.kafka.producer.Publisher - ### Filter Test : Topic :: filtered , Message :: 123 메세지 문제가 있나  
22:06:23.238 [http-nio-9000-exec-1] INFO com.hyomee.kafka.producer.Publisher - ### Filter Test : Topic :: filtered , Message :: hi 123 메세지 문제가 있나  
22:06:23.240 [http-nio-9000-exec-1] INFO com.hyomee.kafka.producer.Publisher - ### Filter Test : Topic :: filtered , Message :: 123 메세지 문제가 있나 hi!
```

- 메시지 받기

```
@KafkaListener(topics = "${topic.filtered.name}", containerFactory = "filterKafkaListenerContainerFactory")  
public void listenWithFilter(String message) {  
    log.debug(String.format("#### Filtered Topic : %s, Message : %s", topicFilteredName , message));  
}
```

Filter조건: hi 가 있으면 제외

```
22:06:23.241 [http-nio-9000-exec-1] INFO com.hyomee.kafka.producer.Publisher - ### ModelVO Test : Topic :: modelvo , Message :: 123 메세지 문제가 있나
```

# 6. Spring Boot Kafka – 자바 객체 메시지

2. KAFKA 예제

3. Spring을 사용한 Apache Kafka

## ProducerFactory 에서 적절한 serializer를 구성 하고 ConsumerFactory 에서 deserializer를 구성

### 01 Producer 구성

- **ProducerFactoryKafka Producer 인스턴스 생성**

BOOTSTRAP\_SERVERS\_CONFIG -Kafka가 실행되는 호스트 및 포트.  
KEY\_SERIALIZER\_CLASS\_CONFIG -키에 사용할 Serializer 클래스. (StringSerializer)  
VALUE\_SERIALIZER\_CLASS\_CONFIG-값에 사용할 Serializer 클래스. . (JsonSerializer)

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.7</version>
</dependency>
```

```
// 자바 객체를 보낸다.
// JsonSerializer를 사용함
@Bean
public ProducerFactory<String, ModelVO> modelVOProducerFactory() {
    Map<String, Object> configProps = new HashMap<>();
    configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
    configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);
    return new DefaultKafkaProducerFactory<>(configProps);
}

@Bean
public KafkaTemplate<String, ModelVO> modelVOKafkaTemplate() {
    return new KafkaTemplate<>(modelVOProducerFactory());
}
```

### 02 Consumer 구성

- **ConsumerFactory 및 KafkaListenerContainerFactory 를 구성**

BOOTSTRAP\_SERVERS\_CONFIG  
: Kafka가 실행되는 호스트 및 포트.  
GROUP\_ID\_CONFIG  
: 컨슈머 그룹  
KEY\_DESERIALIZER\_CLASS\_CONFIG –  
: 키에 사용할 Serializer 클래스. (StringDeserializer)  
VALUE\_DESERIALIZER\_CLASS\_CONFIG-  
: 값에 사용할 Serializer 클래스. . (JsonDeserializer)

```
public ConsumerFactory<String, ModelVO> modelVoConsumerFactory() {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
    props.put(ConsumerConfig.GROUP_ID_CONFIG, modelVoGrp);
    return new DefaultKafkaConsumerFactory<>(props,
                                              new StringDeserializer(),
                                              new JsonDeserializer<>(ModelVO.class));
}

@Bean
public ConcurrentKafkaListenerContainerFactory<String, ModelVO>
modelVoKafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, ModelVO> factory = new
ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(modelVoConsumerFactory());
    return factory;
}
```

# 6. Spring Boot Kafka – 자바 객체 메시지

2. KAFKA 예제

3. Spring을 사용한 Apache Kafka

ProducerFactory 에서 적절한 serializer를 구성 하고 ConsumerFactory 에서 deserializer를 구성

## 03 메시지 보내기

```
@Autowired
private KafkaTemplate<String, ModelVO> modelVOKafkaTemplate

public void sendMessageToModelVO(String message) {
    log.debug(String.format("### ModelVO Test : Topic :: %s , Message :: %s", topicModelVoName, message));
    modelVOKafkaTemplate.send(topicModelVoName, new ModelVO(message, "Hong!"));
}
```

```
22:06:23.241 [http-nio-9000-exec-1] INFO com.hyomee.kafka.producer.Publisher - ### ModelVO Test : Topic :: modelvo , Message :: 123 메세지 문제가 있나
```

## 04 메시지 받기

```
@KafkaListener(topics = "${topic.modelVo.name}", containerFactory = "modelVoKafkaListenerContainerFactory")
public void modelVoListener(ModelVO modelVo) {
    log.debug(String.format("#### ModelVo Topic : %s, Message : %s", topicModelVoName , modelVo.toString()));
}
```

```
22:06:23.353 [org.springframework.kafka.KafkaListenerEndpointContainer#4-0-C-1] DEBUG com.hyomee.kafka.consumer.Subscriber - #### ModelVo Topic : modelvo, Message : ModelVO(msg=123 메세지 문제가 있나 , name=Hong!)
```

소스 위치 : <https://github.com/hyomee/springkafka>

- 아파치 카프가 :  
<https://kafka.apache.org/quickstart>
- 아파치 카프카 스트림 :  
<https://kafka.apache.org/22/documentation/streams/quickstart>
- 주키퍼 :  
<https://zookeeper.apache.org/doc/current/zookeeperOver.html>
- 아파치 카프가 오픈 소스 Confluent :  
<https://www.confluent.io/download>
- 스프링 카프카 Reference :  
<https://docs.spring.io/spring-kafka/reference/>
- 스프링 카프카 API :  
<https://docs.spring.io/spring-kafka/api/overview-summary.html>
- Intro to Apache Kafka with Spring :  
<https://www.baeldung.com/spring-kafka>  
<https://github.com/eugenp/tutorials/blob/master/spring-kafka/src/main/java/com/baeldung/spring/kafka/KafkaApplication.java>
- Spring Boot와 함께 Kafka 사용 :  
<https://reflectoring.io/spring-boot-kafka/>  
<https://github.com/thombergs/code-examples/tree/master/spring-boot/spring-boot-kafka>
- Spring for Apache Kafka 심층 분석 :  
<https://www.confluent.io/blog/spring-for-apache-kafka-deep-dive-part-1-error-handling-message-conversion-transaction-support/>

# THANKS



[www.iabacus.co.kr](http://www.iabacus.co.kr)

Tel. 82-2-2109-5400

Fax. 82-2-6442-5409