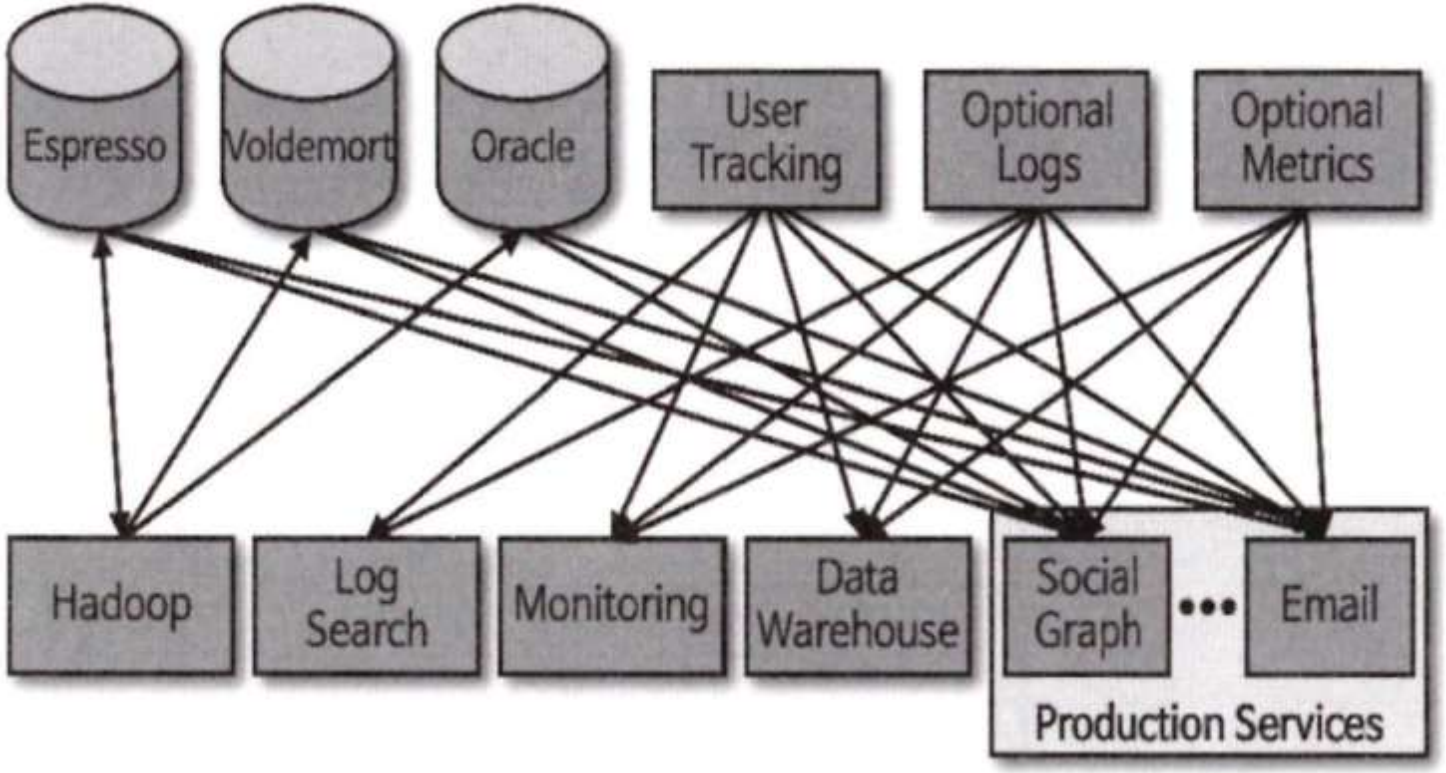
Kafka

Kafka

- Kafka
  - ✓ 2011년 구인/구직 및 동종업계의 동향을 살펴볼 수 있는 소셜 네트워크 사이트인 LinkedIn 에서는 파편화된 데이터 수집 및 분배 아키텍처를 운영하는 데에 큰 어려움을 겪었는데 데이터를 생성하고 적재하기 위해서는 데이터를 생성하는 소스 애플리케이션과 데이터가 최종 적재되는 타깃 애플리케이션을 연결해야 하는데 초기 운영 시에는 단방향 통신을 통해 소스 애플리케이션에서 타깃 애플리케이션으로 연동하는 소스 코드를 작성했고 아키텍처가 복잡하지 않았으므로 운영이 힘들지 않았지만 시간이 지날수록 아키텍처는 거대해졌고 소스 애플리케이션과 타깃 애플리케이션의 개수가 점점 많아지면서 문제가 발생
  - ✓ 데이터를 전송하는 라인이 기하 급수적으로 복잡해지기 시작
  - ✓ 소스 애플리케이션 과 타깃 애플리케이션을 연결하는 파이프라인 개수가 많아지면서 소스 코드 및 버전 관리에서 이슈가 생겼고 타깃 애플리케이션에 장애가 생길 경우 그 영향이 소스 애플리케이션에 그대로 전달됨
  - ✓ 시간이 갈수록 복잡해지는 LinkedIn Back End 아키텍처에서 파편화된 데이터 파이프라인은 서비스를 안정적으로 운영하는 데에 치명적일 것이 불 보듯 뻔했는데 이를 해결하기 위해 LinkedIn 데이터팀에서는 기존에 나와 있던 각종 상용 데이터 프레임워크와 오픈 소스를 아키텍처에 녹여내어 데이터 파이프라인의 파편화를 개선하려고 함
  - ✓ 다양한 메시징 플랫폼 과 ETL(Extract Transform Load) 툴을 적용하여 아키텍처를 변경하려고 노력했지만 파편화된 데이터 파이프라인의 복잡도를 낮춰주는 아키텍처가 되지는 못함

Kafka

- Kafka
  - ✓ 카프카 도입 이전 아키텍처

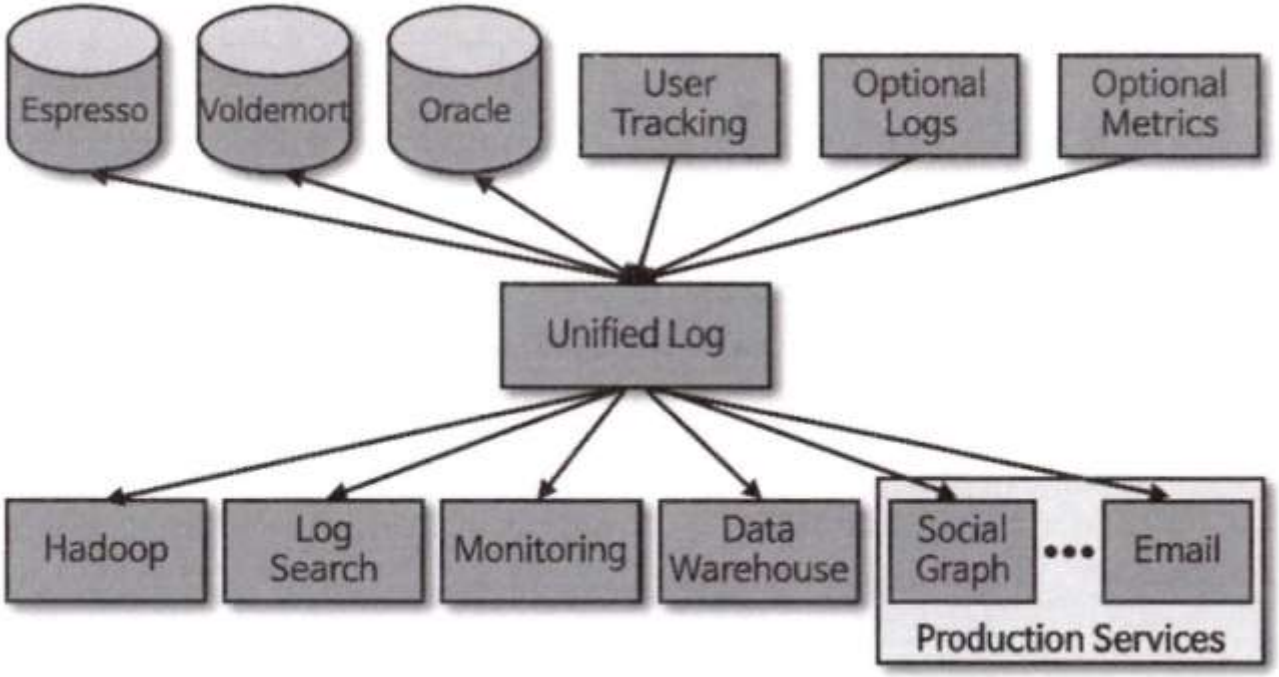


Kafka

- Kafka
  - ✓ LinkedIn의 데이터 팀은 신규 시스템을 만들기로 결정했고 그 결과물이 아파치 카프카 (Apache Kafka)
  - ✓ LinkedIn의 내부 데이터 흐름을 개선하기 위해 개발
  - ✓ 카프카는 각각의 애플리케이션끼리 연결하여 데이터를 처리하는 것이 아니라 한 곳에 모아 처리할 수 있는 중앙 집중화 방식을 사용하는 것이 가능
  - ✓ 카프카를 통해 웹사이트, 애플리케이션, 센서 등에서 취합한 데이터 스트림을 한 곳에서 실시간으로 관리할 수 있게 된 것
  - ✓ 카프카는 기업의 대용량 데이터를 수집하고 이를 사용자들이 실시간 스트림으로 소비할 수 있게 만들어주는 일종의 중추 신경으로 동작

Kafka

- Kafka
  - ✓ 카프카가 적용된 이후의 파이프라인 아키텍처

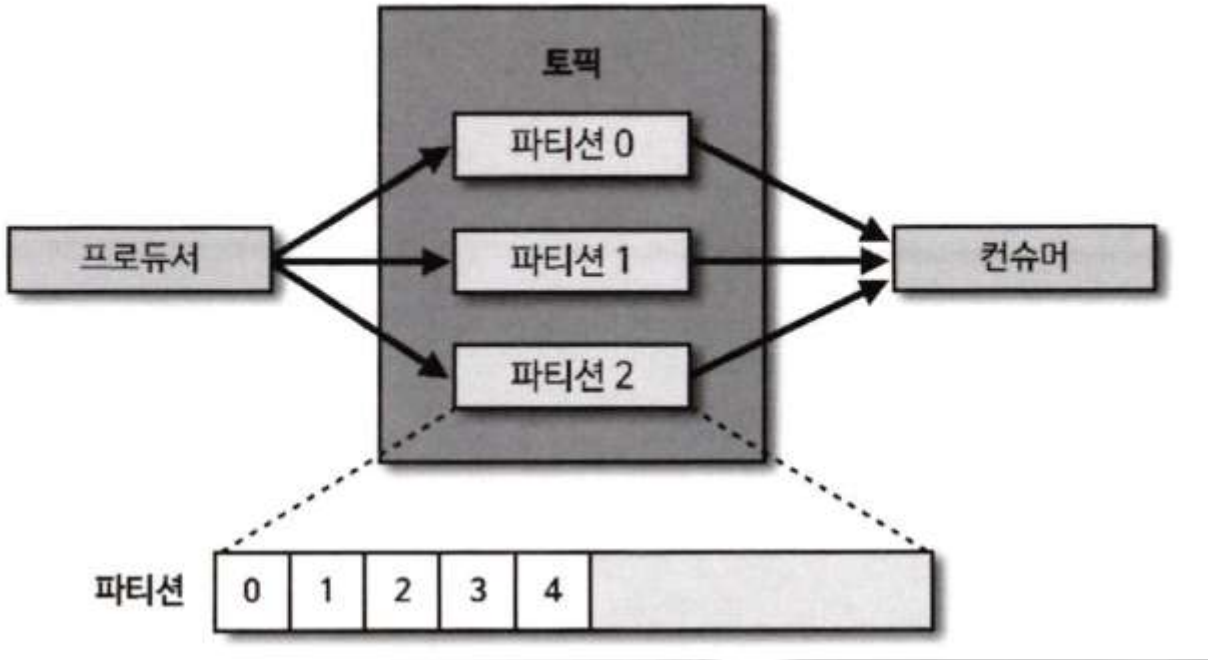


Kafka

- Kafka
  - ✓ 카프카를 중앙에 배치함으로써 소스 애플리케이션과 타깃 애플리케이션 사이의 의존도를 최소화하여 커플링을 완화
  - ✓ 기존에 1:1 매칭으로 개발하고 운영하던 데이터 파이프라인은 커플링으로 인해 한쪽의 이슈가 다른 한쪽의 애플리케이션에 영향을 미치곤 했지만 카프카는 이러한 의존도를 타파
  - ✓ 소스 애플리케이션에서 생성되는 데이터는 어느 타깃 애플리케이션으로 보낼 것인지 고민하지 않고 일단 카프카로 넣으면 되고 카프카 내부에 데이터가 저장되는 파티션의 동작은 FIFO(First In First Out) 방식의 큐 자료 구조 와 유사
  - ✓ 큐에 데이터를 보내는 것이 프로듀서이고 큐에서 데이터를 가져가는 것이 컨슈머

Kafka

- Kafka
  - ✓ 카프카 프로듀서, 컨슈머, 토픽 모식도



Kafka

- Kafka
  - ✓ 카프카를 통해 전달할 수 있는 데이터 포맷은 제한이 없음
  - ✓ 직렬화, 역직렬화를 통해 ByteArray로 통신하기 때문에 자바에서 선언 가능한 모든 객체를 지원
  - ✓ 카프카 클라이언트에서는 기본적으로 ByteArray, ByteBuffer, Double, Long, String 타입에 대응한 직렬화, 역직렬화 클래스가 제공되고 필요할 경우 카프카에서 제공하는 커스텀 직렬화/역직렬화 클래스(Serializer<T>, Deserializer<T>)를 상속받아 개발하여 사용할 수도 있음
  - ✓ 상용 환경에서 카프카는 최소 3대 이상의 서버(브로커)에서 분산 운영하여 프로듀서를 통해 전송받은 데이터를 파일 시스템에 안전하게 기록
  - ✓ 서버 3대 이상으로 이루어진 카프카 클러스터 중 일부 서버에 장애가 발생하더라도 데이터를 지속적으로 복제하기 때문에 안전하게 운영할 수 있으며 데이터를 묶음 단위로 처리하는 배치 전송을 통해 낮은 지연과 높은 데이터 처리량도 가지게 됨
  - ✓ 카프카는 엄청난 양의 데이터를 안전하고 빠르게 처리하는 강점을 가지기 때문에 SK, 삼성, 카카오, 네이버 등과 같이 많은 IT 기업이 사용하고 있으며 이제는 카프카를 사용하지 않는 곳을 찾기 어려울 정도가 되었고 국제적으로도 넷플릭스, 우버, 월마트, 에어비엔비와 같은 기업들의 사용 사례도 각 회사의 엔지니어링 블로그에서 쉽게 찾아볼 수 있음
  - ✓ 미국의 주문형 콘텐츠 서비스 제작 기업이자 엔터테인먼트 OTT 기업인 넷플릭스는 카프카를 아주 잘 활용하는 회사 중 하나로 넷플릭스에서는 36개 이상의 카프카 클러스터를 운영하고 있으며 클러스터를 구성하는 브로커는 4,000개가 넘는데 하나의 클러스터에 100개 이상의 브로커를 운영

Kafka

- Kafka
  - ✓ 카프카는 2011년 초에 오픈 소스 화되면서 관련 생태계를 넓히고 더욱 잘 활용할 수 있음
  - ✓ 카프카 소스 코드는 카프카 깃허브 저장소(https://github.com/apache/kafka)에서 누구든 다운로드하여 내부 동작을 확인할 수 있음
  - ✓ 기능을 추가하거나 수정하고 싶다면 누구든 깃허브 이슈 또는 KIP(Kafka Improvement Proposal)를 통해 카프카를 발전하는 데에 기여할 수 있음
  - ✓ KIP
    - KIP는 카프카의 주요 변경사항을 제안하는 방법 중 하나
    - KIP는 누구든 생성할 수 있음
    - 카프카의 주요 아키텍처 개선을 카프카를 사용하는 전세계 개발자의 손에 위임하는 것
    - KIP에는 제안하게 된 사유, 변경사항, 신규/변경된 인터페이스에 대한 설명, 마이그레이션 계획 및 호환성에 대한 상세한 설명이 포함되어야 함
    - 카프카 아키텍처 그리고 생태계의 대부분은 KIP를 통해 제안되었고 실현
    - KIP를 통해 왜 그런 기능이 나왔고 왜 그렇게 변경되었는지 확인할 수 있음
    - KIP를 통해 제안하는 상세한 방법 - https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Improvement+Proposals

Kafka

- 빅데이터 파이프라인에서 카프카의 역할
  - ✓ Big Data
    - 현대의 IT 서비스는 디지털 정보로 기록되는 모든 것을 저장
    - 쇼핑몰의 결제 내역, 방문한 위치 정보, 소셜 사이트에 남긴 댓글 등 사용자가 남긴 모든 발자취가 데이터로 쌓임
    - 기업에서 실시간으로 저장하는 데이터의 양은 최소 테라 바이트(Terabyte, TB) 단위를 넘어서 엑사 바이트(Exabyte, EB)를 웃도는데 구글 데이터 센터에 보관된 데이터 양은 최소 15 엑사 바이트에 이른다고 하는데 과거에 비즈니스를 수행하기 위한 최소한의 정보만 저장하던 것 과는 다른 양상이라고 볼 수 있는데 이를 빅데이터 라고 부름
    - 빅데이터로 적재되는 데이터의 종류는 다양한데 기존 데이터베이스에서도 사용하기 편리한 스키마(schema) 기반의 정형 데이터부터 일정한 규격이나 형태를 지니지 않은 비정형 데이터까지 있는데 그림, 영상, 음성과 같은 데이터는 비정형 데이터 중 일부라고 할 수 있음
    - 수십 테라 바이트가 넘어가는 방대한 양의 데이터를 기존의 데이터베이스로 관리하는 것은 불가능에 가까움

Kafka

- 빅데이터 파이프라인에서 카프카의 역할
  - ✓ Data Pipeline
    - 빅데이터를 저장하고 활용하기 위해서는 일단 생성되는 데이터를 모두 모으는 것이 중요한데 이때 사용되는 개념이 데이터 레이크(data lake)
    - 데이터 레이크는 빅데이터를 관리하고 사용하는 측면에서 중요한 용어로 레이크(Lake: 호수)라는 이름에서 유추할 수 있는 것처럼 데이터가 모이는 저장 공간을 의미
    - 데이터 웨어하우스(data warehouse) 와 다르게 필터링되거나 패키지화되지 않은 데이터가 저장되고 운영되는 서비스로부터 수집 가능한 모든 데이터를 모으는 것이고 데이터 과학자나 데이터 분석가들은 모은 데이터를 토대로 서비스에 활용할 수 있는 인사이트를 찾음
    - 서비스에서 발생하는 데이터를 데이터 레이크로 모으려면 웹, 앱, 백엔드 서버, DB에서 발생하는 데이터를 데이터 레이크에 직접 end-to-end 방식으로 넣을 수 도 있는데 서비스하는 애플리케이션 개수가 적고 트래픽이 많지 않을 때는 큰 문제가 되지 않지만 서비스가 비대해지고 복잡해지면서 파편화되고 복잡도가 올라가는 문제점이 발생했는데 이를 해결하기 위해서 데이터를 추출(extracting)하고 변경(transforming), 적재(loading)하는 과정을 묶은 데이터 파이프라인을 구축해야 함
    - end-to-end 방식의 데이터 수집 및 적재를 개선하고 안정성을 추구하며 유연하면서도 확장 가능하게 자동화한 것을 데이터 파이프라인 이라고 부름

Kafka

- 빅데이터 파이프라인에서 카프카의 역할
  - ✓ Data Pipeline
    - 안정적이고 확장성이 높은 데이터 파이프라인을 구축하는 것은 빅데이터를 활용하는 기업에게 필수적인데 데이터 파이프라인을 구축하지 않고 일회성으로 구축한 데이터 수집은 결국 데이터 흐름의 파편화로 이어지며 이는 반복적인 유지보수를 필요로 하기 때문에 기술 부채(technical debt)로 남아 지속적으로 개발자들을 괴롭힘
    - 데이터 파이프라인을 안정적이고 확장성 높게 운영하기 위한 좋은 방법 중 하나가 아파치 카프카를 활용하는 것

Kafka

- ❖ 빅데이터 파이프라인에서 카프카의 역할
  - ✓ 카프카 사용 이유
    - ● 높은 처리량
      - ◆ 카프카는 프로듀서가 브로커로 데이터를 보낼 때와 컨슈머가 브로커로부터 데이터를 받을 때 모두 묶어서 전송
      - ◆ 많은 양의 데이터를 송수신할 때 맺어지는 네트워크 비용은 무시할 수 없는 규모
      - ◆ 동일한 양의 데이터를 보낼 때 네트워크 통신 횟수를 최소한으로 줄인다면 동일 시간 내에 더 많은 데이터를 전송할 수 있음
      - ◆ 많은 양의 데이터를 묶음 단위로 처리하는 배치로 처리할 수 있기 때문에 대용량의 실시간 로그 데이터를 처리하는 데에 적합
      - ◆ 파티션 단위를 통해 동일 목적의 데이터를 여러 파티션에 분배하고 데이터를 병렬 처리할 수 있는데 파티션 개수만큼 컨슈머 개수를 늘려서 동일 시간당 데이터 처리량을 늘릴 수 있음

Kafka

- ❖ 빅데이터 파이프라인에서 카프카의 역할
  - ✓ 카프카 사용 이유
    - 확장성
      - ◆ 데이터 파이프라인에서 데이터를 모을 때 데이터가 얼마나 들어올지는 예측하기 어려운데 하루에 1,000건 가량 들어오는 로그 데이터라도 예상치 못한 특정 이벤트로 인해 100만 건 이상의 데이터가 들어오는 경우가 있음
      - ◆ 카프카는 이러한 가변적인 환경에서 안정적으로 확장 가능하도록 설계
      - ◆ 데이터가 적을 때는 카프카 클러스터의 브로커를 최소한의 개수로 운영하다가 데이터가 많아지면 클러스터의 브로커 개수를 자연스럽게 늘려 스케일 아웃(scale-out)할 수 있으며 반대로 데이터 개수가 적어지고 추가 서버들이 필요 없어지면 브로커 개수를 줄여 스케일 인(scale-in)할 수 있음
      - ◆ 카프카의 스케일 아웃, 스케일 인 과정은 클러스터의 무중단 운영을 지원하므로 365일 24시간 데이터를 처리해야 하는 커머스나 은행 같은 비즈니스 모델에서도 안정적인 운영이 가능

Kafka

- ❖ 빅데이터 파이프라인에서 카프카의 역할
  - ✓ 카프카 사용 이유
    - ● 영속성
      - ◆ 영속성이란 데이터를 생성한 프로그램이 종료되더라도 사라지지 않은 데이터의 특성을 의미
      - ◆ 카프카는 다른 메시징 플랫폼과 다르게 전송받은 데이터를 메모리에 저장하지 않고 파일 시스템에 저장
      - ◆ 파일 시스템에 데이터를 적재하고 사용하는 것은 보편적으로 느리다고 생각하겠지만 카프카는 운영체제 레벨에서 파일 시스템을 최대한 활용하는 방법을 적용
      - ◆ 운영체제에서는 파일 I/O 성능 향상을 위해 페이지 캐시(page cache) 영역을 메모리에 따로 생성하여 사용하는데 페이지 캐시 메모리 영역을 사용하여 한번 읽은 파일 내용은 메모리에 저장시켰다가 다시 사용하는 방식이기 때문에 카프카가 파일 시스템에 저장하고 데이터를 저장, 전송하더라도 처리량이 높음
      - ◆ 디스크 기반의 파일 시스템을 활용한 덕분에 브로커 애플리케이션이 장애 발생으로 인해 급작스럽게 종료되더라도 프로세스를 재시작하여 안전하게 데이터를 다시 처리할 수 있음

Kafka

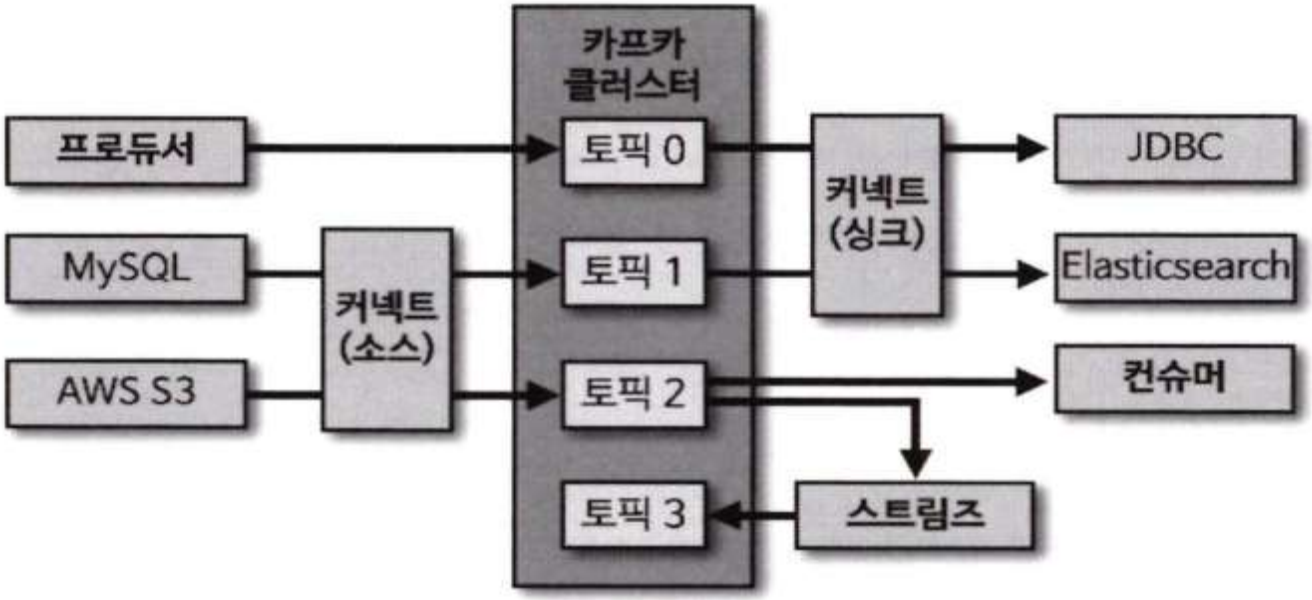
- ❖ 빅데이터 파이프라인에서 카프카의 역할
  - ✓ 카프카 사용 이유
    - 고가용성
      - ◆ 3개 이상의 서버들로 운영되는 카프카 클러스터는 일부 서버에 장애가 발생하더라도 무중단으로 안전하고 지속적으로 데이터를 처리할 수 있음
      - ◆ 클러스터로 이루어진 카프카는 데이터의 복제(replication)를 통해 고가용성의 특징을 가지게 됨
      - ◆ 프로듀서로 전송받은 데이터를 여러 브로커 중 1대의 브로커에만 저장하는 것이 아니라 또 다른 브로커에도 저장
      - ◆ 한 브로커에 장애가 발생하더라도 복제된 데이터가 나머지 브로커에 저장되어 있으므로 저장된 데이터를 기준으로 지속적으로 데이터 처리가 가능한 것
      - ◆ 서버를 직접 운영하는 온프레미스(on-premise) 환경의 서버 랙 또는 퍼블릭 클라우드(public cloud)의 리전 단위 장애에도 데이터를 안전하게 복제할 수 있는 브로커 옵션들이 준비되어 있음

Kafka

- ❖ 빅데이터 파이프라인에서 카프카의 역할
  - ✓ 카프카 사용 이유
    - ● 고가용성
      - ◆ 카프카 클러스터를 1 대, 2대가 아닌 3대 이상의 브로커들로 구성해야 하는 이유
        - ▪ 카프카를 안전하게 운영하기 위해 최소 3대 이상의 브로커로 클러스터를 구성할 것을 추천
        - ▪ 카프카 클러스터를 구축할 때 브로커 개수의 제한은 없음
        - ▪ 1대나 2대의 브로커로도 운영할 수 있는데 1대로 운영할 경우 브로커의 장애는 서비스의 장애로 이어지므로 테스트 목적으로만 사용
        - ▪ 2대로 운영할 경우 한 대의 브로커에 장애가 발생하더라도 나머지 한 대 브로커가 살아 있으므로 안정적으로 데이터를 처리할 수 있지만 브로커 간에 데이터가 복제되는 시간 차이로 인해 일부 데이터가 유실될 가능성이 있으므로 유실을 막기 위해서 min.insync.replicas 옵션을 사용할 수 있음
        - ▪ min.insync.replicas 옵션을 2로 설정하면 최소 2개 이상의 브로커에 데이터가 완전히 복제됨을 보장하고 이 옵션을 2로 사용할 때는 브로커를 3대 이상으로 운영해야 하는데 이유는 3개 중 1개의 브로커에 장애가 나더라도 지속적으로 데이터를 처리할 수 있기 때문
        - ▪ min.insync.replicas 옵션 값보다 작은 수의 브로커가 존재할 때는 토픽에 더는 데이터를 넣을 수 없기 때문에 상용에서 카프카를 운영할 때는 3대 이상의 브로커로 클러스터를 구축해야 함

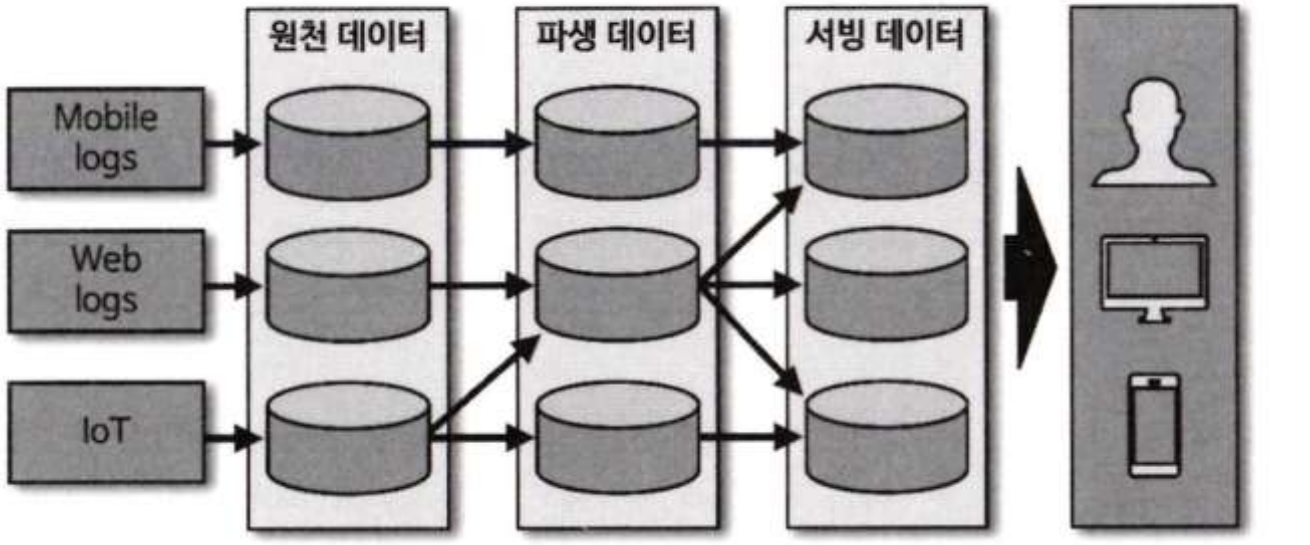
Kafka

- 빅데이터 파이프라인에서 카프카의 역할
  - ✓ 높은 처리량, 확장성, 영속성, 고가용성 특징을 가진 카프카는 데이터 파이프라인을 안전하고 확장성 높게 운영할 수 있도록 설계되었고 여기에 추가적으로 카프카 주변 생태계를 지탱하는 애플리케이션들은 카프카를 데이터 파이프라인으로 더욱 빠르게 적용시킬 수 있도록 도와줌
  - ✓ 카프카 생태계 모식도



Kafka

- 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 카프카의 미래에 대해 설명하려면 데이터 레이크를 구성하는 아키텍처의 역사에 대해 알아야 함
  - ✓ 데이터 레이크 아키텍처의 종류
    - 레거시 데이터 수집 플랫폼
      - 초기 빅데이터 플랫폼은 End To End로 각 서비스 애플리케이션으로부터 데이터를 배치로 모음

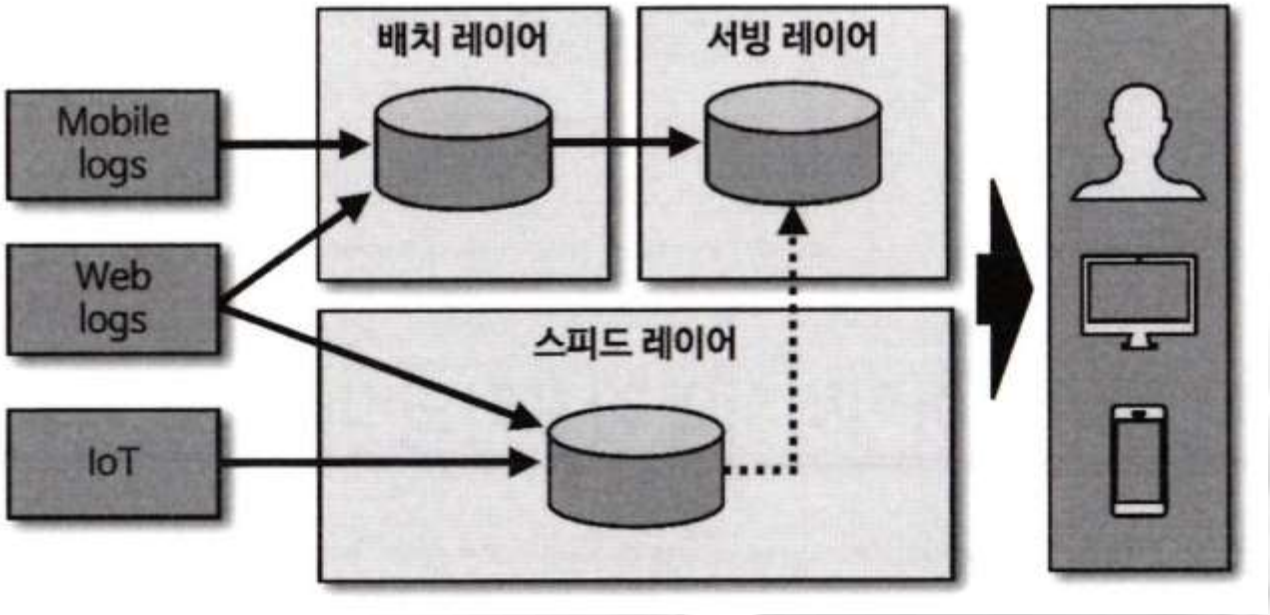


Kafka

- ❖ 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 데이터 레이크 아키텍처의 종류
    - ● 람다 아키텍처(lambda architecture)
      - ◆ 레거시 데이터 수집 플랫폼을 개선하기 위해 구성한 아키텍처
      - ◆ 데이터를 배치로 모으는 구조는 유연하지 못했으며 실시간으로 생성되는 데이터들에 대한 인사이트를 서비스 애플리케이션에 빠르게 전달하지 못하는 단점이 있었고 원천 데이터로부터 파생된 데이터의 히스토리를 파악하기가 어려웠고 계속되는 데이터의 가공으로 인해 데이터가 파편화되면서 데이터 거버넌스(data governance: 데이터 표준 및 정책)를 지키기 어려움
      - ◆ 기존의 배치 데이터를 처리하는 부분 외에 스피드 레이어(speed layer)라고 불리는 실시간 데이터 ETL 작업 영역을 정의한 아키텍처를 만들었는데 이것이 람다 아키텍쳐

Kafka

- ❖ 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 데이터 레이크 아키텍처의 종류
    - ● 람다 아키텍처(lambda architecture)



Kafka

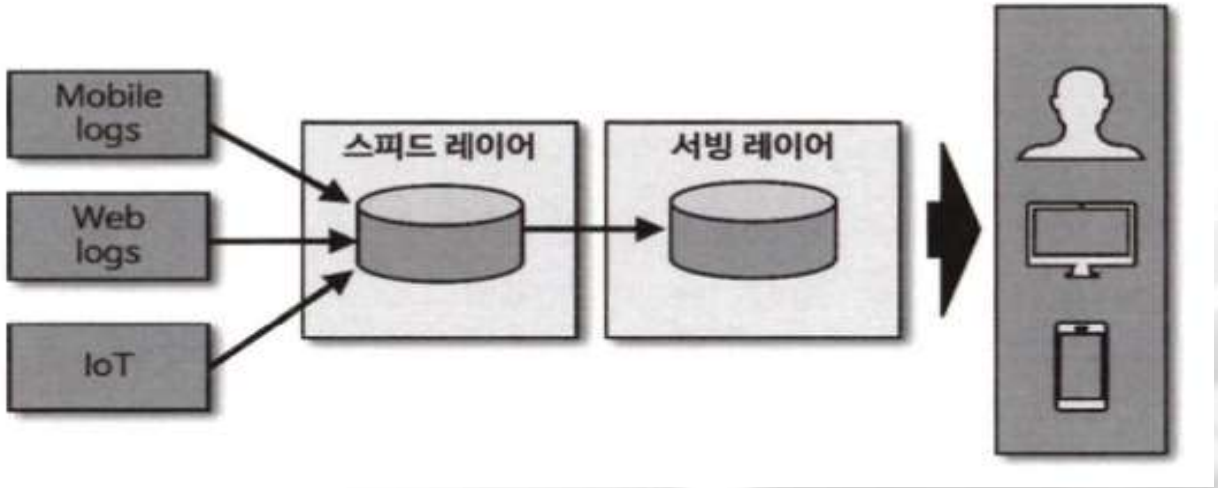
- ❖ 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 데이터 레이크 아키텍처의 종류
    - ● 람다 아키텍처(lambda architecture)
      - ◆ 람다 아키텍처는 3가지 레이어로 나누는데 배치 레이어는 배치 데이터를 모아서 특정 시간, 타이밍마다 일괄 처리하고 서빙 레이어(serving layer)는 가공된 데이터를 데이터 사용자, 서비스 애플리케이션이 사용할 수 있도록 데이터가 저장된 공간이며 스피드 레이어는 서비스에서 생성되는 원천 데이터를 실시간으로 분석하는 용도로 사용
      - ◆ 배치 데이터에 비해 낮은 지연으로 분석이 필요한 경우에 스피드 레이어를 통해 데이터를 분석
      - ◆ 스피드 레이어에서 가공, 분석된 실시간 데이터는 사용자 또는 서비스에서 직접 사용할 수 있지만 필요한 경우에는 서빙 레이어로 데이터를 보내서 저장하고 사용할 수 있음
      - ◆ 람다 아키텍처에서 카프카는 스피드 레이어에 위치하는데 서비스 애플리케이션들의 실시간 데이터를 짧은 지연 시간으로 처리(process), 분석(analysis)할 수 있기 때문
      - ◆ 카프카 스트림즈 와 같은 스트림 프로세싱 도구는 윈도우 함수(windowing functions), 상태 기반 프로세싱(stateful processing), 무상태 기반 프로세싱(stateless processing) 등 분석을 위한 다양한 기능을 제공하여 람다 아키텍처의 중요 플랫폼으로 자리잡음

Kafka

- ❖ 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 데이터 레이크 아키텍처의 종류
    - ● 람다 아키텍처(lambda architecture)
      - ◆ 데이터를 배치 처리하는 레이어와 실시간 처리하는 레이어를 분리한 람다 아키텍처는 데이터 처리 방식을 명확히 나눌 수 있었지만 레이어가 2개로 나뉘기 때문에 생기는 단점이 있는데 데이터를 분석, 처리하는 데에 필요한 로직이 2벌로 각각의 레이어에 따로 존재해야 한다는 점과 배치 데이터와 실시간 데이터를 융합하여 처리할 때는 다소 유연하지 못한 파이프라인을 생성해야 한다는 점으로 로직 구현체의 파편화를 해소하기 위해 1개의 로직을 추상화하여 배치 레이어와 스피드 레이어에 적용하는 형태를 고안한 서밍버드가 있었지만 결국 컴파일 이후에는 배치 레이어 와 스피드 레이어에 각각 디버깅, 배포해야 했기에 문제가 완벽히 해결되는 것은 아님

Kafka

- ❖ 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 데이터 레이크 아키텍처의 종류
    - ● 카파 아키텍처(kappa architecture)
      - ◆ 람다 아키텍처의 단점을 해소하기 위해 제이 크렙스(Jay Kreps: 카프카를 최초로 고안한 개발자로 전 LinkedIn 팀장이며 현 컨플루언트 CEO)는 카파 아키텍처를 제안했는데 카파 아키텍처는 람다 아키텍처와 유사하지만 배치 레이어를 제거하고 모든 데이터를 스피드 레이어에 넣어서 처리한다는 점이 다름

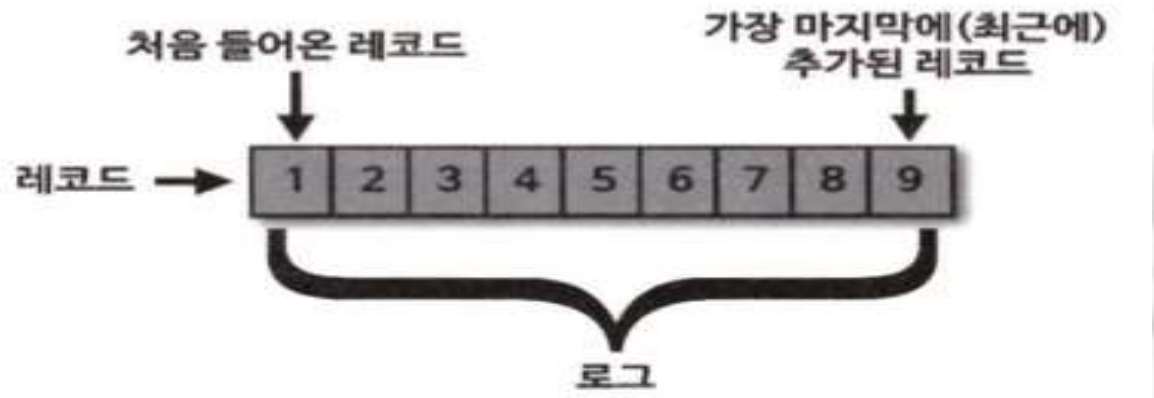


Kafka

- ❖ 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 데이터 레이크 아키텍처의 종류
    - ● 카파 아키텍처(kappa architecture)
      - ◆ 람다 아키텍처에서 단점으로 부각되었던 로직의 파편화, 디버깅, 배포, 운영 분리에 대한 이슈를 제거하기 위해 배치 레이어를 제거한 카파 아키텍처는 스피드 레이어에서 데이터를 모두 처리할 수 있었으므로 엔지니어들은 더욱 효율적으로 개발과 운영에 임할 수 있게 되었는데 카파 아키텍처는 스피드 레이어에서 모든 데이터를 처리하므로 서비스에서 생성되는 모든 종류의 데이터를 스트림 처리해야 함
        - ▪ 배치 데이터와 스트림 데이터
          - 배치 데이터는 초, 분, 시간, 일 등으로 한정된(bounded) 기간 단위 데이터를 의미하는데 배치 데이터는 일괄 처리(batch processing)하는 것이 특징으로 인터넷 쇼핑몰에서 지난 1분 간 주문한 제품 목록, 2021년 신입생 목록이 배치 데이터
          - 스트림 데이터는 한정되지 않은(unbounded) 데이터로 시작 데이터와 끝 데이터가 명확히 정해지지 않은 데이터를 의미하는데 각 지점의 데이터는 보통 작은 단위(KB 단위)로 쪼개져 있으며 웹 사용자의 클릭 로그, 주식 정보, 사물 인터넷의 센서 데이터를 스트림 데이터라고 볼 수 있음

Kafka

- 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 데이터 레이크 아키텍처의 종류
    - 카파 아키텍처(kappa architecture)
      - 배치 데이터를 어떻게 스트림 프로세스로 처리할 수 있게 된 배경에는 제이 크렙스가 모든 데이터를 로그(log)로 바라보는 것에서 시작
      - 로그는 일반적으로 우리가 생각하는 애플리케이션을 로깅(logging)하는 텍스트 로그가 아닌 데이터의 집합을 의미
      - 데이터는 지속적으로 추가가 가능하며 각 데이터에는 일정한 번호(또는 타임스탬프)가 붙음



Kafka

- ❖ 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 데이터 레이크 아키텍처의 종류
    - ● 카파 아키텍처(kappa architecture)
      - ◆ 로그는 배치 데이터를 스트림으로 표현하기에 적합했는데 일반적으로 데이터 플랫폼에서 배치 데이터를 표현할 때는 각 시점(시간 별, 일자 별 등)의 전체 데이터를 백업한 스냅샷 데이터를 의미했지만 배치 데이터를 로그로 표현할 때는 각 시점의 배치 데이터의 변환 기록(change log)을 시간 순서대로 기록함으로써 각 시점의 모든 스냅샷 데이터를 저장하지 않고도 배치 데이터를 표현할 수 있게 됨
      - ◆ 기존 스냅샷으로 배치 데이터를 표현한 모습과 로그로 배치 데이터를 표현한 모습

Kafka

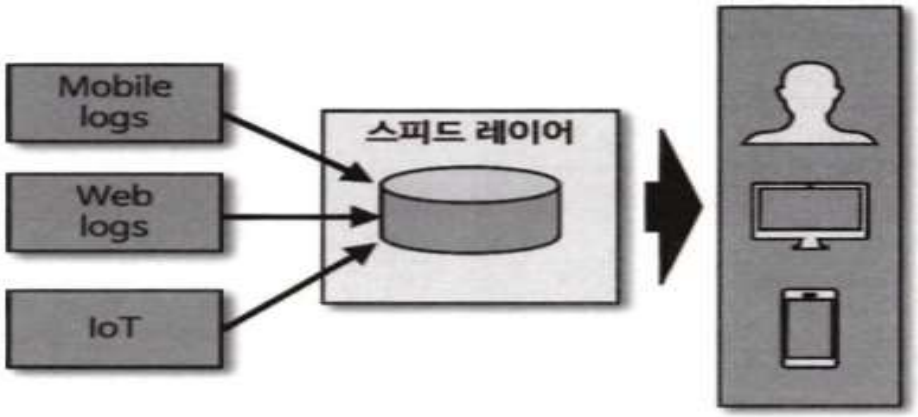
- 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 데이터 레이크 아키텍처의 종류
    - 카파 아키텍처(kappa architecture)
      - 로그로 배치 데이터와 스트림 데이터를 저장하고 사용하기 위해서는 변환 기록이 일정 기간 동안 삭제되어서는 안되고 지속적으로 추가되어야 하고 서비스에서 생성된 모든 데이터가 스피드 레이어에 들어오는 것을 감안하면 스피드 레이어를 구성하는 데이터 플랫폼은 SPOF(Single Point Of Failure)가 될 수 있으므로 반드시 고 결함성(High Availability)과 장애 허용(fault tolerant) 특징을 지녀야 함
      - 아파치 카프카는 이러한 특징에 정확히 부합하는 플랫폼으로 카프카 내부에서 사용되는 파티션, 레코드, 오프셋은 제이 크렙스가 정의한 로그의 데이터 플랫폼 구현체로 볼 수 있음

Kafka

- ❖ 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 데이터 레이크 아키텍처의 종류
    - 람다 아키텍처를 거쳐 카파 아키텍처에서 중요한 역할을 맡은 카프카는 2020년 카프카 서밋에서 제이 크랩스는 카파 아키텍처에서 서빙 레이어를 제거한 아키텍처인 스트리밍 데이터 레이크(streaming data lake)를 제안
    - 카파 아키텍처를 살펴보면 데이터를 사용하는 고객(데이터 사이언티스트, 데이터 엔지니어 등)을 위해 스트림 데이터를 서빙 레이어에 저장하는 것을 알 수 있음
    - 서빙 레이어는 하둡 파일 시스템(HDFS), 오브젝트 스토리지(S3, minio 등) 와 같이 데이터 플랫폼에서 흔히 사용되는 저장소
    - 카프카를 통해 분석하고 프로세싱한 데이터를 다시 서빙 레이어의 저장소에 저장하는데 스피드 레이어로 사용되는 카프카에 분석과 프로세싱을 완료한 거대한 용량의 데이터를 오랜 기간 저장하고 사용할 수 있다면 서빙 레이어는 제거되어도 되며 오히려 서빙 레이어와 스피드 레이어가 이중으로 관리되는 운영 리소스를 줄일 수 있음

Kafka

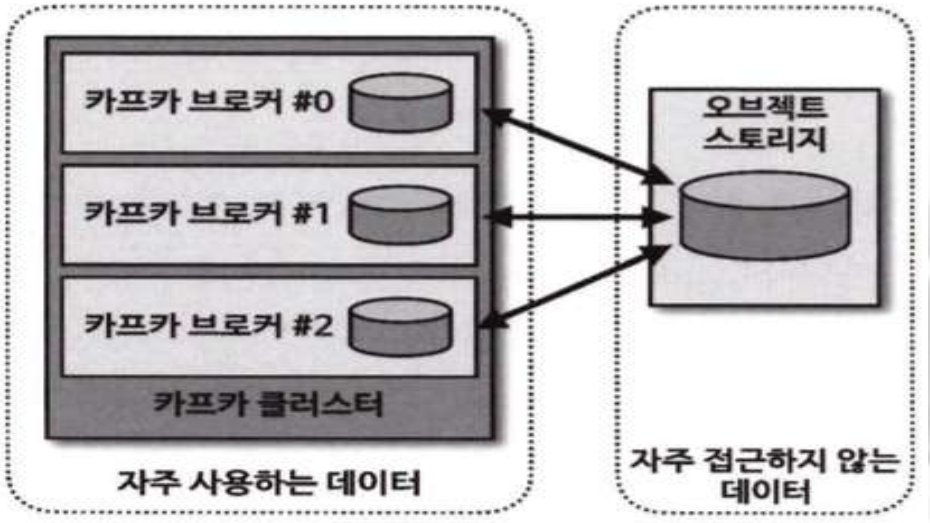
- ❖ 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 데이터 레이크 아키텍처의 종류
    - ● 스트리밍 데이터 레이크 아키텍쳐



- ◆ 스피드 레이어에서 데이터를 분석, 프로세싱, 저장함으로써 단일 진실 공급원 (SSOT, Single Source Of Truth)이 되는 것
- ◆ 데이터가 필요한 모든 고객과 서비스 애플리케이션은 스트리밍 데이터 레이크의 스피드 레이어만 참조함으로써 데이터의 중복, 비정합성 과 같은 문제에서 벗어 날 수 있음

Kafka

- ❖ 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 데이터 레이크 아키텍처의 종류
    - 카프카를 스트리밍 데이터 레이크로 사용하기 위해 개선해야 하는 부분이 있는데 우선 자주 접근하지 않는 데이터를 굳이 비싼 자원(브로커의 메모리, 디스크)에 유지할 필요가 없다라는 것인데 카프카 클러스터에서 자주 접근하지 않는 데이터는 오브젝트 스토리지 와 같이 저렴하면서도 안전한 저장소에 옮겨 저장하고 자주 사용하는 데이터만 브로커에서 사용하는 구분 작업이 필요
    - 카프카 클러스터가 단계별 저장소를 가질 수 있도록 추가 기능을 개발하고 있으며 자세한 진행 사항은 KIP-405 티켓에서 찾아볼 수 있음
    - 단계별 저장소를 가진 카프카 클러스터 모식도



Kafka

- ❖ 데이터 레이크 아키텍처 와 카프카의 미래
  - ✓ 데이터 레이크 아키텍처의 종류
    - 데이터를 사용하는 고객이나 서비스 애플리케이션에서 카프카의 데이터를 쿼리(query)할 수 있는 주변 데이터 플랫폼이 필요한데 컨플루언트(Confluent)에서는 카프카의 데이터를 SQL 기반으로 조회할 수 있도록 카프카 스트림즈를 추상화한 ksqlDB를 오픈소스로 제공하고 있음
    - 카프카와 ksqlDB를 사용한다면 제이 크렙스가 제안한 스트리밍 데이터 레이크에 가장 근접한 모습이지만 ksqlDB는 아직 타임스탬프, 오프셋, 파티션 기반 쿼리를 제공하지 않기 때문에 배치 데이터를 완벽히 처리하기에는 부족하고 이외에도 아파치 스파크 스트리밍, 프레스토, 아파치 드릴, 하이브/스파크SQL 등을 조합하는 방식도 스트리밍 데이터 레이크로 사용하는 방법이라 볼 수 있지만 모든 데이터 형태와 포맷을 지원하는 것은 아니므로 앞으로 추가적인 기능 개발이 필요
    - 카프카가 스트리밍 데이터 레이크로 완성될 수 있도록 주변 데이터 플랫폼 애플리케이션들이 개발 완료된다면 가까운 미래에는 카프카를 사용하는 것이 데이터 레이크를 사용하는 것과 동일한 의미를 가질 것

Kafka

- 설치
  - ✓ Docker를 이용한 설치
    - docker-compose.yml 파일 생성

```
version: '3.8'
services:
  zookeeper:
    image: wurstmeister/zookeeper:latest
    container_name: zookeeper
    ports:
      - "2181:2181"
  kafka:
    image: wurstmeister/kafka:latest
    container_name: kafka
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: 127.0.0.1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

Kafka

- ❖ 설치
  - ✓ Docker를 이용한 설치
    - ● docker-compose.yml 파일 생성
      - ◆ KAFKA_ADVERTISED_HOST_NAME은 카프카 클라이언트와 브로커가 통신을 하기 위해 처음으로 client가 부여받는 ip로 kafka client가 broker에게 통신 요청을 하고 이를 부여받으면 해당 ip와 연결하여 사용 여부를 검사할 수 있음
      - ◆ 하나의 서버에 kafka client와 broker가 함께 설치되어 있다면 굳이 adv_host를 private ip로 부여할 필요없이 localhost로 지정

Kafka

- Docker-Compose 설치
  - ✓ Mac 이나 Windows에서는 Docker Desktop을 설치한 경우 별도의 설치 과정 필요 없음
  - ✓ 리눅스
    - sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
    - sudo chmod +x /usr/local/bin/docker-compose
    - sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
    - sudo chown 계정 /usr/local/bin/docker-compose
    - docker-compose --version

Kafka

- 설치
  - ✓ Docker를 이용한 설치
    - Container 실행
      - docker-compose up -d
    - Container 확인
      - docker ps

```
(base) adam@choongangui-MacBookPro kafka % docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED         STATUS         PORTS                                        NAMES
8b808de52da5   wurstmeister/zookeeper         "/bin/sh -c '/usr/sb…"   34 seconds ago  Up 32 seconds  22/tcp, 2888/tcp, 3888/tcp, 0.0.0.0:2181->2181/tcp   zookeeper
15b9a80fc53d   wurstmeister/kafka:2.12-2.5.0  "start-kafka.sh"         34 seconds ago  Up 32 seconds  0.0.0.0:9092->9092/tcp                       kafka
```

Kafka

- 설치
  - ✓ Docker를 이용한 설치
    - 설정 변경
      - $ docker exec -it kafka /bin/bash
      - $ bash# vi /opt/kafka/config/server.properties

```
############################## Socket Server Settings
##############################
# The address the socket server listens on. It will get the value returned
from
# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#     listeners = listener_name://host_name:port
#   EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://:9092
# Hostname and port the broker will advertise to producers and consumers.
If not set,
# it uses the value for "listeners" if configured.  Otherwise, it will use the
value
# returned from java.net.InetAddress.getCanonicalHostName().
delete.topic.enable=true
auto.create.topics.enable=true
```

Kafka

- ❖ 설치
  - ✓ Docker를 이용한 설치
    - ● Kafka 테스트
      - ◆ 토픽 생성
        - % docker exec -it kafka /bin/bash
        - bash-5.1# cd /opt/kafka/bin
        - bash-5.1# kafka-topics.sh --create --zookeeper zookeeper:2181 --replication-factor 1 --partitions 1 --topic exam-topic

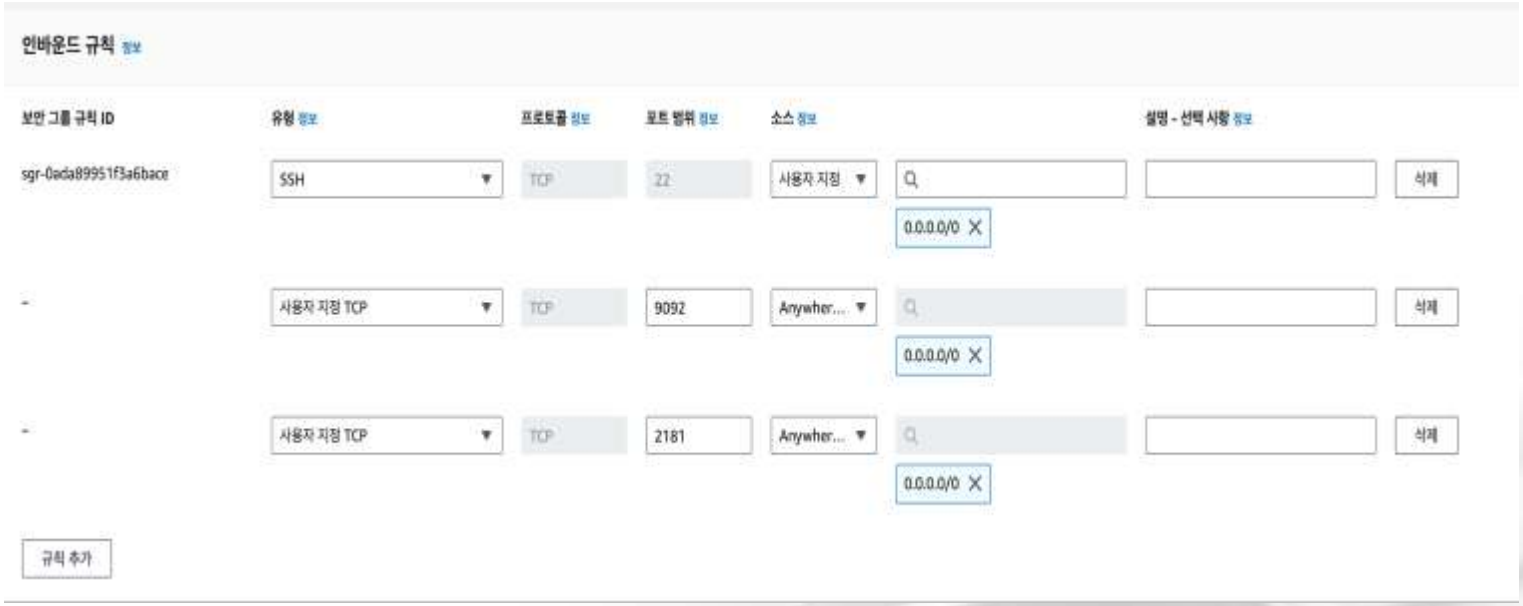
        Created topic exam-topic.
      - ◆ 토픽 리스트 조회: kafka-topics.sh --bootstrap-server localhost:9092 --list
      - ◆ 토픽 삭제
        - kafka-topics.sh --delete --zookeeper zookeeper:2181 --topic exam-topic

Kafka

- ❖ 설치
  - ✓ Docker를 이용한 설치
    - ● Kafka 테스트
      - ◆ 메시지 전송
        - bash-5.1# kafka-console-producer.sh --topic exam-topic --broker-list localhost:9092
      - ◆ 메시지 확인 – 새로운 터미널 실행
        - bash-5.1# kafka-console-consumer.sh --topic exam-topic --bootstrap-server localhost:9092 --from-beginning

Kafka

- ❖ 설치
  - ✓ EC2에 Kafka 설치
    - ● EC2 인스턴스 생성
      - ◆ 9092, 2181 번 포트의 InBound 설정에 모든 IP를 허용



      - ◆ EC2 인스턴스에 접속

Kafka

- ❖ 설치
  - ✓ EC2에 Kafka 설치
    - ● JDK 설치
      - ◆ $ sudo apt-get update
      - ◆ $ sudo apt install openjdk-17-jdk
      - ◆ $ java -version

        openjdk version "17.0.11" 2024-04-16

        OpenJDK Runtime Environment (build 17.0.11+9-Ubuntu-1)

        OpenJDK 64-Bit Server VM (build 17.0.11+9-Ubuntu-1, mixed mode, sharing)

Kafka

- ❖ 설치
  - ✓ EC2에 Kafka 설치
    - ● 주키퍼 와 카프카 브로커 실행
      - ◆ 카프카 바이너리 패키지 다운로드 - https://kafka.apache.org/downloads
        - ▪ $ wget https://archive.apache.org/dist/kafka/3.6.0/kafka_2.13-3.6.0.tgz
      - ◆ 압축 해제
        - ▪ $ tar xvf kafka_2.13-3.6.0.tgz
      - ◆ 압축 해제 확인
        - ▪ $ ll

```
total 110640
drwxr-x--- 5 ubuntu ubuntu      4096 Nov  2 05:32 ./
drwxr-xr-x 3 root   root        4096 Nov  2 05:09 ../
-rw-r--r-- 1 ubuntu ubuntu       220 Jan  6  2022 .bash_logout
-rw-r--r-- 1 ubuntu ubuntu      3771 Jan  6  2022 .bashrc
drwx------ 2 ubuntu ubuntu      4096 Nov  2 05:12 .cache/
-rw-r--r-- 1 ubuntu ubuntu       807 Jan  6  2022 .profile
drwx------ 2 ubuntu ubuntu      4096 Nov  2 05:09 .ssh/
-rw-r--r-- 1 ubuntu ubuntu         0 Nov  2
    05:13 .sudo_as_admin_successful
drwxr-xr-x 7 ubuntu ubuntu      4096 Sep 29 05:03 kafka_2.13-3.6.0/
-rw-rw-r-- 1 ubuntu ubuntu 113257079 Oct  4 03:54 kafka_2.13-3.6.0.tgz
```

Kafka

- ❖ 설치
  - ✓ EC2에 Kafka 설치
    - ● 주키퍼 와 카프카 브로커 실행
      - ◆ 디렉토리 수정: sudo mv kafka_2.13-3.6.0 /opt/kafka
      - ◆ 환경 변수 추가: nano ~/.bashrc

        export KAFKA_HOME=/opt/kafka

        export PATH=$PATH:$KAFKA_HOME/bin
      - ◆ 변경한 내용 반영

        source ~/.bashrc

Kafka

- ❖ 설치
  - ✓ EC2에 Kafka 설치
    - ● 주키퍼 와 카프카 브로커 실행
      - ◆ 카프카 브로커 힙 메모리 설정
        - ▪ 카프카 브로커를 실행하기 위해서는 힙 메모리 설정이 필요
        - ▪ 카프카 브로커는 레코드의 내용은 페이지 캐시로 시스템 메모리를 사용하고 나머지 객체들을 힙 메모리에 저장하여 사용하기 때문에 카프카 브로커를 운영할 때 힙 메모리를 5GB 이상으로 설정하지 않는 것이 일반적
        - ▪ 실습이나 테스트 목적이라면 힙 메모리를 작게 설정해서 실행해도 무관
        - ▪ 카프카 패키지의 힙 메모리는 카프카 브로커는 1G, 주키퍼는 512MB로 기본 설정
        - ▪ 인스턴스(t2.micro)는 1G 메모리를 가지고 있으므로 카프카 브로커와 주키퍼를 기본 설정과 함께 동시에 실행하면 1.5G 메모리가 필요하기 때문에 Cannot allocate memory 에러가 출력되면서 실행되지 않음
        - ▪ export 명령어를 사용하여 힙 메모리 사이즈를 미리 환경변수로 지정해서 실행해야 주키퍼와 카프카 브로커를 실행할 때 힙 메모리 에러가 발생하지 않음

Kafka

- ❖ 설치
  - ✓ EC2에 Kafka 설치
    - ● 주키퍼 와 카프카 브로커 실행
      - ◆ 카프카 브로커 힙 메모리 설정
        - ▪ 전역 환경 변수 설정
          - $ nano ~/.bashrc

```
# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi

export KAFKA_HEAP_OPTS="-Xmx400m -Xms400m"
~
-- INSERT --                                              119,43        Bot
```

          - $ source ~/.bashrc
          - $ echo $KAFKA_HEAP_OPTS
            -Xmx400m -Xms400m

Kafka

- ❖ 설치
  - ✓ EC2에 Kafka 설치
    - ● 주키퍼 와 카프카 브로커 실행
      - ◆ 카프카 브로커 실행 옵션 설정: config 디렉토리에 있는 server.properties 파일에서 작업

```
# The address the socket server listens on. It will get the value returned
    from
# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#     listeners = listener_name://host_name:port
#   EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://:9092
# Hostname and port the broker will advertise to producers and consumers.
    If not set,
# it uses the value for "listeners" if configured.  Otherwise, it will use the
    value
# returned from java.net.InetAddress.getCanonicalHostName().
advertised.listeners=PLAINTEXT://공인ip:9092

delete.topic.enable=true
auto.create.topics.enable=true
```

Kafka

- ❖ 설치
  - ✓ EC2에 Kafka 설치
    - ● 주키퍼 와 카프카 브로커 실행
      - ◆ 카프카 브로커 실행 옵션 설정: config 디렉토리에 있는 server.properties 파일에서 작업
        - ▪ listeners=PLAINTEXT://:9092
          - • 카프카 브로커가 통신을 위해 열어둘 인터페이스 IP, port, 프로토콜을 설정할 수 있는데 설정하지 않으면 모든 IP와 port에서 접속 가능
        - ▪ advertised.listeners의 주석 처리를 해제하고 후자에는 공인 ip를 작성
          - • 카프카 클라이언트 또는 카프카 커맨드 라인 툴에서 접속할 때 사용하는 IP와 port 정보
          - • 인스턴스를 생성할 때 발급받은 퍼블릭 IPv4값을 IP에 넣고 port에는 카프카 기본 포트인 9092를 설정
        - ▪ delete.topic.enable 설정은 토픽을 삭제 할 수 있게 하기 위함이고 auto.create.topics.enable 설정은 토픽이 존재하지 않은 상태에서 토픽을 찾으려 할 때 자동으로 이를 생성해주는 옵션
        - ▪ exit로 컨테이너를 나와 다시 재기동 시켜주면 되는데 restart를 쓰면 kafka 버그 때문에 바로 안 올라가는 경우가 있으니 당황하지말고 stop 후 좀 대기하여 start로 재기동

Kafka

- ❖ 설치
  - ✓ EC2에 Kafka 설치
    - ● 주키퍼 와 카프카 브로커 실행
      - ◆ 주키퍼 실행
        - ▪ $ zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
        - ▪ $ jps -vm

          13639 Jps -vm -Dapplication.home=/usr/lib/jvm/java-8-openjdk-amd64 -Xms8m

          6142 QuorumPeerMain ./config/zookeeper.properties -Xmx400m -Xms400m -XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -XX:+ExplicitGCInvokesConcurrent -XX:MaxInlineLevel=15 -Djava.awt.headless=true -Xloggc:/home/ubuntu/kafka_2.13-3.6.0/bin/../logs/zookeeper-gc.log -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=100M -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false -Dkafka.logs.dir=/home/ubuntu/kafka_2.13-3.6.0/bin/../logs -Dlog4j.configuration=file:./bin/../config/log4j.properties

Kafka

- ❖ 설치
  - ✓ EC2에 Kafka 설치
    - ● 주키퍼 와 카프카 브로커 실행
      - ◆ 카프카 브로커 실행
        - ▪ $ kafka-server-start.sh -daemon /opt/kafka/config/server.properties
        - ▪ $ jps -m

          14516 Jps -m

          14444 Kafka ./config/server.properties

          6142 QuorumPeerMain ./config/zookeeper.properties

Kafka

- ❖ 설치
  - ✓ EC2에 Kafka 설치
    - ● 주키퍼 와 카프카 브로커 실행
      - ◆ 카프카 로그 확인: kafka 루트 디렉토리에서 수행
        - ▪ $ tail -f logs/server.log

[2023-11-02 06:19:38,774] INFO [Controller id=0, targetBrokerId=0] Client requested connection close from node 0 (org.apache.kafka.clients.NetworkClient)

[2023-11-02 06:19:38,820] INFO [/config/changes-event-process-thread]: Starting (kafka.common.ZkNodeChangeNotificationListener$ChangeEventProcessThread)

Kafka

- ❖ 설치
  - ✓ EC2에 Kafka 설치
    - ● 주키퍼 와 카프카 브로커 실행
      - ◆ 카프카 로그 확인: kafka 루트 디렉토리에서 수행
        - ▪ 토픽 생성: ./bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --topic ec2topic

          Created topic exam-topic.
        - ▪ 토픽 리스트 조회: ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --list

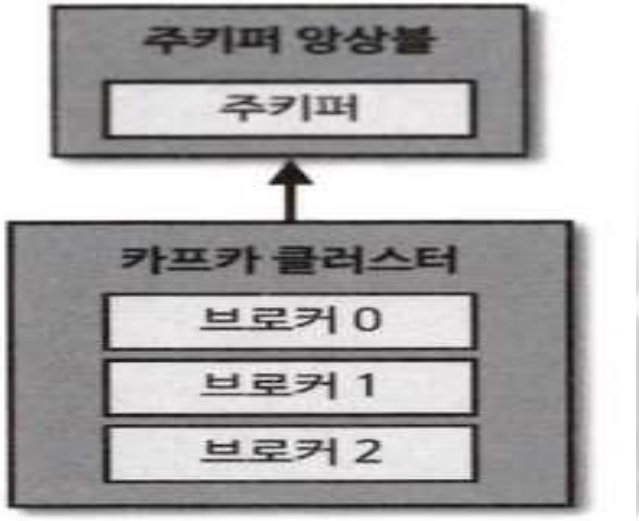
          exam-topic

Kafka

- ❖ 설치
  - ✓ EC2에 Kafka 설치
    - ● 주키퍼 와 카프카 브로커 실행
      - ◆ 카프카 로그 확인: kafka 루트 디렉토리에서 수행
        - ◆ 토픽 전송
          - ▪ ./bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic exam-topic
          - ▪ ./bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic ec2topic --property parse.key=true --property key.separator=:
        - ◆ 토픽 수신
          - ▪ ./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic exam-topic --from-beginning
          - ▪ ./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic ec2topic --property print.key=true --property key.separator=":" --from-beginning

Kafka 개념

- 카프카 브로커, 클러스터, 주키퍼
  - ✓ 카프카 브로커는 카프카 클라이언트와 데이터를 주고받기 위해 사용하는 주체이자 데이터를 분산 저장하여 장애가 발생 하더라도 안전하게 사용할 수 있도록 도와주는 애플리케이션
  - ✓ 하나의 서버에는 한 개의 카프카 브로커 프로세스가 실행되는데 카프카 브로커 서버 1 대로도 기본 기능이 실행되지만 데이터를 안전하게 보관하고 처리하기 위해 3대 이상의 브로커 서버를 1개의 클러스터로 묶어서 운영
  - ✓ 카프카 클러스터로 묶인 브로커들은 프로듀서가 보낸 데이터를 안전하게 분산 저장하고 복제하는 역할을 수행
  - ✓ 주키퍼와 연동하여 동작하는 카프카 클러스터



Kafka 개념

- ❖ 카프카 브로커, 클러스터, 주키퍼
  - ✓ 데이터 저장, 전송
    - 프로듀서로부터 데이터를 전달받으면 카프카 브로커는 프로듀서가 요청한 토픽의 파티션에 데이터를 저장하고 컨슈머가 데이터를 요청하면 파티션에 저장된 데이터를 전달
    - 프로듀서로부터 전달된 데이터는 파일 시스템에 저장
    - 카프카는 메모리나 데이터베이스에 저장하지 않으며 캐시 메모리를 구현하여 사용하지도 않으며 파일 시스템에 저장하기 때문에 파일 입출력으로 인해 속도 이슈가 발생하지 않을까 의문을 가질 수 있는데 일반적으로 파일 시스템은 다루기 편하지만 지속적으로 입출력 할 경우 메모리에 올려서 사용하는 것보다 처리 속도가 현저히 느리기 때문이지만 카프카는 페이지 캐시(page cache)를 사용하여 디스크 입출력 속도를 높여서 이 문제를 해결
    - 페이지 캐시란 OS에서 파일 입출력의 성능 향상을 위해 만들어 놓은 메모리 영역을 의미하는데 한번 읽은 파일의 내용은 메모리의 페이지 캐시 영역에 저장시키고 추후 동일한 파일의 접근이 일어나면 디스크에서 읽지 않고 메모리에서 직접 읽는 방식
    - JVM 위에서 동작하는 카프카 브로커가 페이지 캐시를 사용하지 않는다면 빠른 동작을 기대할 수 없으며 페이지 캐시를 사용하지 않으면 카프카에서 캐시를 직접 구현해야 했을 것이고 지속적으로 변경되는 데이터 때문에 가비지 컬렉션이 자주 일어나 속도가 현저히 느려질 것이기 때문
    - 이러한 특징 때문에 카프카 브로커를 실행하는데 힙 메모리 사이즈를 크게 설정할 필요가 없음

Kafka 개념

- 카프카 브로커, 클러스터, 주키퍼
  - ✓ 데이터 복제, 싱크
    - 데이터 복제(replication)는 카프카를 장애 허용 시스템(fault tolerant system)으로 동작하도록 하는 원동력
    - 복제의 이유는 클러스터로 묶인 브로커 중 일부에 장애가 발생하더라도 데이터를 유실하지 않고 안전하게 사용하기 위함
    - 카프카의 데이터 복제는 파티션 단위로 이루어지는데 토픽을 생성할 때 파티션의 복제 개수(replication factor)도 같이 설정되는데 직접 옵션을 선택하지 않으면 브로커에 설정된 옵션 값을 따라가며 복제 개수의 최솟값은 1(복제 없음)이고 최댓값은 브로커 개수만큼 설정하여 사용할 수 있음

Kafka 개념

- ❖ 카프카 브로커, 클러스터, 주키퍼
  - ✓ 데이터 복제, 싱크
    - 복제된 파티션은 리더(leader)와 팔로워 (follower)로 구성
    - 프로듀서 또는 컨슈머 와 직접 통신하는 파티션을 리더라고 하고 나머지 복제 데이터를 가지고 있는 파티션을 팔로워라고 부름
    - 팔로워 파티션들은 리더 파티션의 오프셋을 확인하여 현재 자신이 가지고 있는 오프셋과 차이가 나는 경우 리더 파티션으로부터 데이터를 가져와서 자신의 파티션에 저장하는데 이 과정을 복제(replication)라고 부름
    - 파티션 복제로 인해 나머지 브로커에도 파티션의 데이터가 복제되므로 복제 개수만큼의 저장 용량이 증가한다는 단점이 있지만 복제를 통해 데이터를 안전하게 사용할 수 있다는 강력한 장점 때문에 카프카를 운영할 때 2 이상의 복제 개수를 정하는 것이 중요
    - 카프카 브로커가 설치된 기업용 서버는 개인용 컴퓨터와 비교가 안 될 정도로 안정성이 좋지만 서버는 해커로 인한 침입, 디스크 오류, 네트워크 연결 장애 등의 이유로 언제든 장애가 발생할 수 있음
    - 브로커가 다운되면 해당 브로커에 있는 리더 파티션은 사용할 수 없기 때문에 팔로워 파티션 중 하나가 리더 파티션 지위를 넘겨받고 이를 통해 데이터가 유실되지 않고 컨슈머 와 프로듀서 간에 데이터를 주고받도록 동작할 수 있는데 운영 시에는 데이터 종류마다 다른 복제 개수를 설정하고 상황에 따라서는 토픽마다 복제 개수를 다르게 설정하여 운영하기도 함

Kafka 개념

- ❖ 카프카 브로커, 클러스터, 주키퍼
  - ✓ 데이터 복제, 싱크
    - 데이터가 일부 유실되어도 무관하고 데이터 처리 속도가 중요하다면 1 또는 2 로 설정
    - 금융 정보와 같이 유실이 일어나면 안 되는 데이터의 경우 복제 개수를 3으로 설정하여 최대 2개의 브로커 에서 동시에 장애가 발생하더라도 데이터를 안정적으로 유지할 수 있도록 함
    - 카프카 브로커의 장애
      - ◆ 기업용 서버는 안정적으로 365일 24시간 동작하도록 설계되었기 때문에 이슈가 발생하는 것은 극히 드물지만 만약 서버의 장애 발생이 365일 중 하루 발생한다고 가정해 보면 1 대일 때는 1년 중 하루만 장애 처리를 하면 될 것이지만 카프카 브로커 300대를 한 개 클러스터로 운영한다고 가정하면 거의 매일 한 번씩 서버 장애가 발생할 수 있음
      - ◆ 카프카가 복제를 이용한 장애 허용 시스템이 아니라면 매일 한 번씩 카프카를 사용한 서비스가 중단되어야 할 것
      - ◆ 장애가 발생하더라도 안정적으로 데이터를 유지하고 사용할 수 있다는 특징이 카프카를 사용해야 하는 이유가 될 수 있음

Kafka 개념

- 카프카 브로커, 클러스터, 주키퍼
  - ✓ 컨트롤러(controller)
    - 클러스터의 다수 브로커 중 한 대가 컨트롤러의 역할을 수행
    - 컨트롤러는 다른 브로커들의 상태를 체크하고 브로커가 클러스터에서 빠지는 경우 해당 브로커에 존재하는 리더 파티션을 재 분배
    - 카프카는 지속적으로 데이터를 처리해야 하므로 브로커의 상태가 비정상이라면 빠르게 클러스터에서 빼내는 것이 중요
    - 컨트롤러 역할을 하는 브로커에 장애가 생기면 다른 브로커가 컨트롤러 역할을 수행

Kafka 개념

- 카프카 브로커, 클러스터, 주키퍼
  - ✓ 데이터 삭제
    - 카프카는 다른 메시징 플랫폼과 다르게 컨슈머가 데이터를 가져가더라도 토픽의 데이터는 삭제되지 않으며 컨슈머나 프로듀서가 데이터 삭제를 요청할 수도 없음
    - 오직 브로커만이 데이터를 삭제할 수 있으며 데이터 삭제는 파일 단위로 이루어지는데 이 단위를 로그 세그먼트(log segment) 라고 부름
    - 이 세그먼트에는 다수의 데이터가 들어있기 때문에 일반적인 데이터베이스처럼 특정 데이터를 선별해서 삭제할 수 없음
    - 세그먼트는 데이터가 쌓이는 동안 파일 시스템으로 열려있으며 카프카 브로커에 log.segement.bytes 또는 log.segment.ms 옵션에 값이 설정되면 세그먼트 파일이 닫힘
    - 세그먼트 파일이 닫히게 되는 기본값은 1GB 용량에 도달했을 때인데 간격을 더 줄이고 싶다면 작은 용량으로 설정하면 되지만 너무 작은 용량으로 설정하면 데이터들을 저장하는 동안 세그먼트 파일을 자주 여닫음으로써 부하가 발생할 수 있으므로 주의
    - 닫힌 세그먼트 파일은 log.retention.bytes 또는 log. retention.ms 옵션에 설정 값이 넘으면 삭제되며 닫힌 세그먼트 파일을 체크하는 간격은 카프카 브로커의 옵션에 설정된 log.retention.check.interval.ms에 따름
    - 카프카는 데이터를 삭제하지 않고 메시지 키를 기준으로 오래된 데이터를 압축하는 정책을 가져갈 수 도 있음

Kafka 개념

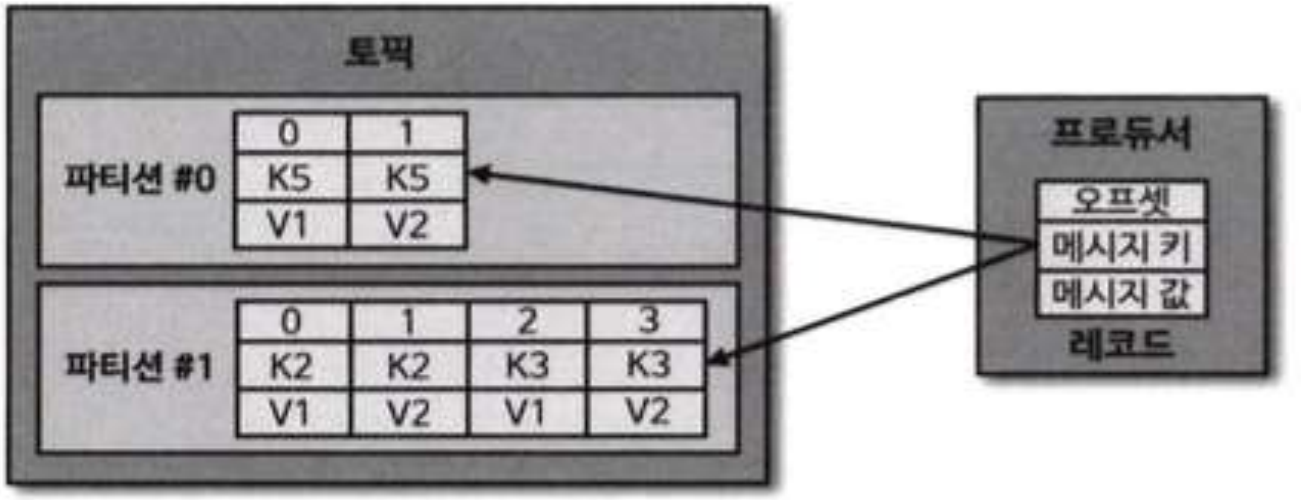
- 카프카 브로커, 클러스터, 주키퍼
  - ✓ 컨슈머 오프셋 저장
    - 컨슈머 그룹은 토픽이 특정 파티션으로부터 데이터를 가져가서 처리하고 이 파티션의 어느 레코드까지 가져갔는지 확인하기 위해 오프셋을 커밋
    - 커밋한 오프셋은 _consiimer_ offsets 토픽에 저장하는데 여기에 저장된 오프셋을 토대로 컨슈머 그룹은 다음 레코드를 가져가서 처리
  - ✓ 코디네이터 (coordinator)
    - 클러스터의 다수 브로커 중 한 대는 코디네이터의 역할을 수행
    - 코디네이터는 컨슈머 그룹의 상태를 체크하고 파티션을 컨슈머 와 매칭되도록 분배하는 역할을 수행
    - 컨슈머가 컨슈머 그룹에서 빠지면 매칭되지 않은 파티션을 정상 동작하는 컨슈머로 할당하여 끊임없이 데이터가 처리되도록 도와주는데 이렇게 파티션을 컨슈머로 재할당하는 과정을 리밸런스(rebalance)라고 부름

Kafka 개념

- 카프카 브로커, 클러스터, 주키퍼
  - ✓ 주키퍼
    - 분산 코디네이션 서비스를 제공하는 주키퍼는 카프카의 클러스터 설정 리더 정보 와 컨트롤러 정보를 담고 있어 카프카를 실행하는 데에 필요한 필수 애플리케이션으로 카프카의 메타 데이터를 관리하는 데에 사용

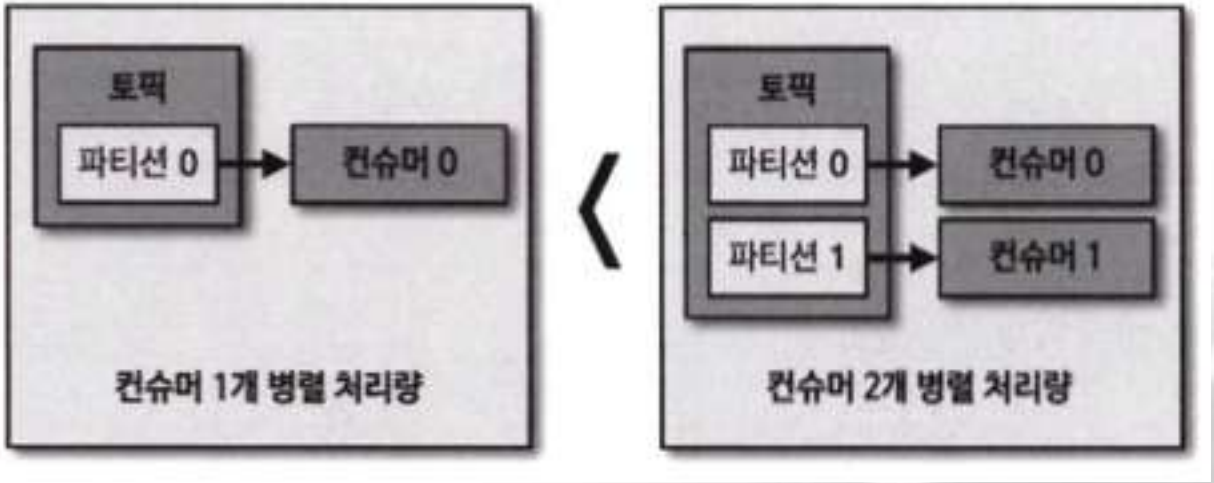
Kafka 개념

- 토픽 과 파티션
  - ✓ 토픽은 카프카에서 데이터를 구분하기 위해 사용하는 단위
  - ✓ 토픽은 1개 이상의 파티션을 소유
  - ✓ 파티션에는 프로듀서가 보낸 데이터들이 들어가 저장되는데 이 데이터를 레코드(record) 라고 부름



Kafka 개념

- 토픽 과 파티션
  - ✓ 파티션은 카프카의 병렬 처리의 핵심으로써 그룹으로 묶인 컨슈머들이 레코드를 병렬로 처리 할 수 있도록 매칭
  - ✓ 컨슈머의 처리량이 한정된 상황에서 많은 레코드를 병렬로 처리하는 가장 좋은 방법은 컨슈머의 개수를 늘려 스케일 아웃하는 것
  - ✓ 컨슈머 개수를 늘림과 동시 에 파티션 개수도 늘리면 처리량이 증가하는 효과를 볼 수 있음



Kafka 개념

- 토픽 과 파티션
  - ✓ 파티션은 자료구조에서 접하는 큐(queue)와 비슷한 구조
  - ✓ First-In-First-Out(FIFO) 구조와 같이 먼저 들어간 레코드는 컨슈머가 먼저 가져가게 되는데 다만 일반적인 자료 구조로 사용되는 큐는 데이터를 가져가면(pop) 레코드를 삭제하지만 카프카에서는 삭제하지 않음
  - ✓ 파티션의 레코드는 컨슈머가 가져가는 것과 별개로 관리
  - ✓ 이러한 특징 때문에 토픽의 레코드는 다양한 목적을 가진 여러 컨슈머 그룹들이 토픽의 데이터를 여러 번 가져 갈 수 있음

Kafka 개념

- ❖ 토픽 과 파티션
  - ✓ 토픽 이름 제약 조건
    - 빈 문자열 토픽 이름은 지원하지 않음
    - 토픽 이름은 마침표 하나(.) 또는 마침표 둘(..)로 생성될 수 없음
    - 토픽 이름의 길이는 249자 미만으로 생성
    - 토픽 이름은 영어 대소문자 와 숫자 0 부터 9 그리고 마침표(.), 언더바(_), 하이픈(-) 조합으로 생성할 수 있으며 이외의 문자열이 포함된 토픽 이름은 생성 불가
    - 카프카 내부 로직 관리 목적으로 사용되는 2개 토픽(_consumer_offsets, _transaction_state)과 동일한 이름으로 생성 불가능
    - 카프카 내부적으로 사용하는 로직 때문에 토픽 이름에 마침표(.)와 언더바(_)가 동시에 들어가면 안되는데 생성은 할 수 있지만 사용 시 이슈가 발생할 수 있기 때문에 마침표(.)와 언더바(_)가 들어간 토픽 이름을 사용하면 WARNING 메시지가 발생
    - 이미 생성된 토픽 이름의 마침표(.)를 언더바(_)로 바꾸거나 언더버(_)를 마침표(.)로 바꾼 경우 신규 토픽 이름과 동일하다면 생성할 수 없는데 topic 이름의 토픽이 생성되어 있다면 topic 이름의 토픽을 생성할 수 없음

Kafka 개념

- ❖ 토픽 과 파티션
  - ✓ 의미있는 토픽 이름 작명 방법
    - 토픽 이름을 모호하게 작성하면 유지 보수 시 큰 어려움을 겪을 수 있는데 예를 들어 test-20210204, abcd, bigdata, test와 같은 토픽 이름은 어떤 용도로 누가 사용하고 있는지 어떻게 만들어졌는지 알 수 없으므로 지양해야 함
    - 최소한 토픽 이름을 통해 어떤 개발환경에서 사용되는 것인지 판단 가능해야 하고 어떤 애플리케이션에서 어떤 데이터 타입으로 사용되는지 유추할 수 있어야 함
    - 만약 전사에서 공용으로 사용하고 있는 카프카라면 추가적으로 토픽의 오너십(ownership)을 가진 팀의 이름을 토픽 이름에 추가하는 것도 고려
    - 히스토리를 명확히 파악하고 싶다면 회사 내부에서 사용 중인 JIRA(아틀라시안에서 만든 개발 프로젝트 기획, 트래킹 도구) 티켓 번호를 토픽 이름에 넣는 것도 좋은 방법
    - 카프카 클러스터를 2대 이상 운영할 경우에는 클러스터를 구분하기 위해 카프카 클러스터 이름을 넣을 수도 있음
    - 토픽 이름에는 영어 대소문자 외에 마침표(.), 언더바(_), 하이픈(-)을 넣을 수 있는데 토픽 이름 작성 시 구분자로 이 문자들을 사용하면 편리하게 읽을 수 있음
    - 토픽 이름은 영어 대소문자 모두 지원하며 프로듀서나 컨슈머에서 사용 시 대소문자를 구분하여 처리하기 때문에 토픽 이름에 대소문자가 섞여 있어도 생성하는 데에는 이슈가 없지만 휴먼 에러(human error)로 인한 실수를 방지하기 위해 대문자와 소문자를 섞어서 쓰는 카멜케이스(CamelCase)를 사용하기 보다는 케밥케이스(kebab-case) 또는 스네이크 표기법(snake_ case)과 같이 소문자를 쓰되 구분자로 특수문자를 조합하여 사용하면 좋음

Kafka 개념

- ❖ 토픽 과 파티션
  - ✓ 의미있는 토픽 이름 작명 방법
    - 예시

      <환경>.<팀-명>.<애플리케이션-명>.<메시지-타입>

      prd.marketing-team.sms-platformjson

      <프로젝트-명>.<서비스-명>.<환경>.<이벤트-명>

      commerce.payment.prd.notification

      <환경>.<서비스-명>.<JIRA-번호>.<메시지-타입>

      dev email-senderjira-1234 email-vo-custom

      <카프카-클러스터-명>.<환경>.<서비스-명>.<메시지-타입>

      aws-kafka.live.marketing-platformJson

Kafka 개념

- ❖ 토픽 과 파티션
  - ✓ 의미있는 토픽 이름 작명 방법
    - 중요한 것은 토픽 이름에 대한 규칙을 사전에 정의하고 구성원들이 그 규칙을 잘 따르는 것
    - 아무리 규칙을 정해도 따르지 않으면 예측하지 못한 방향으로 토픽 이름이 생성될 것이고 이것들은 기술 부채(technical debt)로 남아 카프카 운영자의 머리를 아프게 할 것
    - 카프카는 토픽 이름 변경을 지원하지 않으므로 이름을 변경하기 위해서는 삭제 후 다시 생성하는 것 외에는 방법이 없기 때문에 다소 까다롭더라도 카프카 클러스터 사용자에게 토픽 생성에 대한 규칙을 인지시키고 규칙에 따르도록 하는 것이 중요
    - 규칙을 따르지 않은 토픽 이름의 경우 주기적으로 확인하고 검토하여 사용 여부를 판단
    - 만일 단발성으로 생성된 토픽이라면 삭제 처리하고 실제로 사용을 위한 토픽이라면 삭제 후 신규로 토픽을 만드는 것을 권장

Kafka 개념

- 레코드
  - ✓ 레코드는 타임스탬프, 메시지 키, 메시지 값, 오프셋, 헤더로 구성
  - ✓ 프로듀서가 생성한 레코드가 브로커로 전송되면 오프셋과 타임스탬프가 지정되어 저장
  - ✓ 브로커에 한번 적재된 레코드는 수정할 수 없고 로그 리텐션 기간 또는 용량에 따라서만 삭제
  - ✓ 타임스탬프는 프로듀서에서 해당 레코드가 생성된 시점(Create Time)의 유닉스 타임이 설정
  - ✓ 컨슈머는 레코드의 타임스탬프를 토대로 레코드가 언제 생성되었는지 알 수 있는데 다만 프로듀서가 레코드를 생성할 때 임의의 타임스탬프 값을 설정할 수 있고 토픽 설정에 따라 브로커에 적재된 시간(Log Append Time)으로 설정될 수 있다는 점을 유의
  - ✓ 메시지 키는 메시지 값을 순서대로 처리하거나 메시지 값의 종류를 나타내기 위해 사용
  - ✓ 메시지 키를 사용하면 프로듀서가 토픽에 레코드를 전송할 때 메시지 키의 해시값을 토대로 파티션을 지정하게 되는데 동일한 메시지 키라면 동일 파티션에 들어가는 것
  - ✓ 다만 어느 파티션에 지정될지 알 수 없고 파티션 개수가 변경되면 메시지 키와 파티션 매칭이 달라지게 되므로 주의해야 하는데 만약 메시지 키를 사용하지 않는다면 프로듀서에서 레코드를 전송 할 때 메시지 키를 선언하지 않으면 됨
  - ✓ 메시지 키를 선언하지 않으면 null로 설정되는데 메시지 키가 null로 설정된 레코드는 프로듀서 기본 설정 파티셔너에 따라서 파티션에 분배되어 적재

Kafka 개념

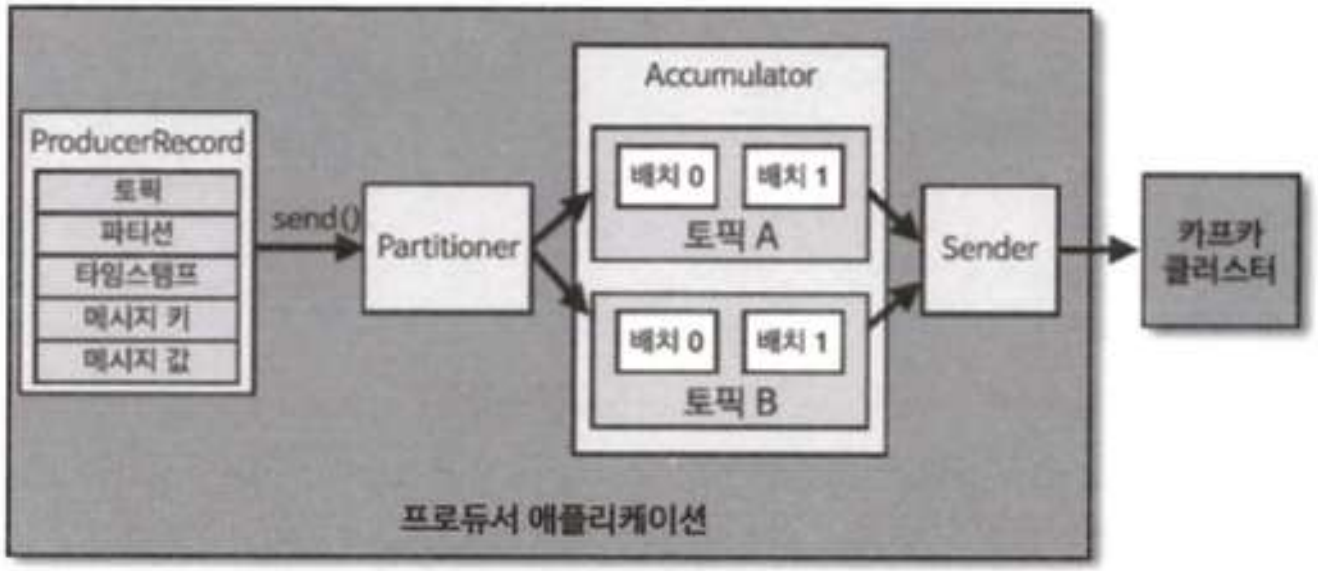
- 레코드
  - ✓ 메시지 값에는 실질적으로 처리할 데이터가 들어있는데 메시지 키와 메시지 값은 직렬화되어 브로커로 전송되기 때문에 컨슈머가 이용할 때는 직렬화 한 형태와 동일한 형태로 역직렬화를 수행해야 함
  - ✓ 직렬화, 역직렬 화할 때는 반드시 동일한 형태로 처리해야 함
  - ✓ 만약 프로듀서가 StringSerializer로 직렬화한 메시지 값을 컨슈머가 IntegerDeserializer로 역직렬화하면 정상적인 데이터를 얻을 수 없음
  - ✓ 레코드의 오프셋은 0 이상의 숫자로 이루어져 있는데 레코드의 오프셋은 직접 지정할 수 없고 브로커에 저장될 때 이전에 전송된 레코드의 오프셋+1 값으로 생성
  - ✓ 오프셋은 카프카 컨슈머가 데이터를 가져갈 때 사용되는데 오프셋을 사용하면 컨슈머 그룹으로 이루어진 카프카 컨슈머들이 파티션의 데이터를 어디까지 가져갔는지 명확히 지정할 수 있음
  - ✓ 헤더는 레코드의 추가적인 정보를 담는 메타데이터 저장소 용도로 사용
  - ✓ 헤더는 키/값 형태 로 데이터를 추가하여 레코드의 속성(스키마 버전 등)을 저장하여 컨슈머에서 참조할 수 있음

Kafka 개념

- ❖ 카프카 클라이언트
  - ✓ Producer API
    - 카프카에서 데이터의 시작점은 프로듀서
    - 프로듀서 애플리케이션은 카프카에 필요한 데이터를 선언하고 브로커의 특정 토픽의 파티션에 전송
    - 프로듀서는 데이터를 전송할 때 리더 파티션을 가지고 있는 카프카 브로커와 직접 통신
    - 프로듀서를 구현하는 가장 기초적인 방법은 카프카 클라이언트를 라이브러리로 추가하여 애플리케이션을 만드는 것
    - 프로듀서는 데이터를 직렬화하여 카프카 브로커로 보내기 때문에 선언 가능한 모든 형태를 브로커로 전송할 수 있음
    - 직렬화란 자바 또는 외부 시스템에서 사용 가능하도록 바이트 형태로 데이터를 변환하는 기술
    - 직렬화를 사용하면 프로듀서는 자바 기본형과 참조형 뿐만 아니라 동영상, 이미지 같은 바이너리 데이터도 프로듀서를 통해 전송할 수 있음

Kafka 개념

- 카프카 클라이언트
  - ✓ Producer API



Kafka 개념

- ❖ 카프카 클라이언트
  - ✓ Producer API
    - ● 주요 옵션
      - ◆ 필수 옵션
        - ▪ bootstrap.servers: 프로듀서가 데이터를 전송할 대상 카프카 클러스터에 속한 브로커의 호스트 이름:포트를 1개 이상 작성하는데 2개 이상 브로커 정보를 입력하여 일부 브로커에 이슈가 발생하더라도 접속하는 데에 이슈가 없도록 설정 가능
        - ▪ key.serializer: 레코드의 메시지 키를 직렬화하는 클래스를 지정
        - ▪ value.serializer: 레코드의 메시지 값을 직렬화하는 클래스를 지정
      - ◆ 선택 옵션
        - ▪ acks
          - • 프로듀서가 전송한 데이터가 브로커들에 정상적으로 저장되었는지 전송 성공 여부를 확인하는 데에 사용하는 옵션으로 0,1, -1(all) 중 하나로 설정
          - • 설정 값에 따라 데이터의 유실 가능성이 달라지는데 기본값은 1로써 리더 파티션에 데이터가 저장되면 전송 성공으로 판단하고 0은 프로듀서가 전송한 즉시 브로커에 데이터 저장 여부와 상관없이 성공으로 판단하며 -1 또는 all은 토픽의 min.insync.replicas 개수에 해당하는 리더 파티션과 팔로워 파티션에 데이터가 저장되면 성공하는 것으로 판단

Kafka 개념

- ❖ 카프카 클라이언트
  - ✓ Producer API
    - ● 주요 옵션
      - ◆ 선택 옵션
        - buffer.memory: 브로커로 전송할 데이터를 배치로 모으기 위해 설정할 버퍼 메모리 양을 지정하는데 기본값은 33554432(32MB)이다.
        - retries: 프로듀서가 브로커로부터 에러를 받고 난 뒤 재전송을 시도하는 횟수를 지정하는데 기본값은 2147483647
        - batch.size: 배치로 전송할 레코드 최대 용량을 지정하는데 너무 작게 설정하면 프로듀서가 브로커로 더 자주 보내기 때문에 네트워크 부담이 있고 너무 크게 설정하면 메모리를 더 많이 사용하게 되는 점을 주의해야 하는데 기본값은 16384
        - linger.ms: 배치를 전송하기 전까지 기다리는 최소 시간으로 기본값은 0
        - partitioner.class: 레코드를 파티션에 전송할 때 적용하는 파티셔너 클래스를 지정하는데 기본값은 org.apache.kafka.clients.producer.intemals.DefaultPartitioner
        - enable.idempotence: 멱등성 프로듀서로 동작할지 여부를 설정하는데 기본값은 false
        - transactional.id: 프로듀서가 레코드를 전송할 때 레코드를 트랜잭션 단위로 묶을지 여부를 설정

Kafka 개념

- 카프카 클라이언트
  - ✓ Consumer API
    - 프로듀서가 전송한 데이터는 카프카 브로커에 적재
    - 컨슈머는 적재된 데이터를 사용하기 위해 브로커로부터 데이터를 가져와서 필요한 처리를 수행
    - 마케팅 문자를 고객에게 보내는 기능이 있다면 컨슈머는 토픽으로부터 고객 데이터를 가져와서 문자 발송 처리
    - 토픽의 파티션으로부터 데이터를 가져가기 위해 컨슈머를 운영하는 방법은 크게 2가지인데 하나는 1개 이상의 컨슈머로 이루어진 컨슈머 그룹을 운영하는 것이고 다른 하나는 토픽의 특정 파티션만 구독하는 컨슈머를 운영하는 것

Kafka 개념

- ❖ 카프카 클라이언트
  - ✓ Consumer API
    - ● 주요 옵션
      - ◆ 필수 옵션
        - bootstrap.servers: 프로듀서가 데이터를 전송할 대상 카프카 클러스터에 속한 브로커의 호스트 이름:포트를 1 개 이상 작성하는데 2개 이상 브로커 정보를 입력하여 일부 브로커에 이슈가 발생하더라도 접속하는 데에 이슈가 없도록 설정 가능
        - key.serializer: 레코드의 메시지 키를 직렬화하는 클래스를 지정
        - value.serializer: 레코드의 메시지 값을 직렬화하는 클래스를 지정
      - ◆ 선택 옵션
        - group.id: 컨슈머 그룹 아이디를 지정하는데 subscribe() 메서드로 토픽을 구독하여 사용할 때는 이 옵션을 필수로 넣어야 하는데 기본값은 null
        - auto.offset.reset: 컨슈머 그룹이 특정 파티션을 읽을 때 저장된 컨슈머 오프셋이 없는 경우 어느 오프셋부터 읽을지 선택하는 옵션으로 이미 컨슈머 오프셋이 있다면 이 옵션값은 무시되는데 옵션은 latest, earliest, none 중 1 개를 설정할 수 있고 latest로 설정하면 가장 높은(가장 최근에 넣은) 오프셋부터 읽기 시작하고 earliest로 설정하면 가장 낮은(가장 오래전에 넣은) 오프셋부터 읽기 시작하며 none으로 설정하면 컨슈머 그룹이 커밋한 기록이 있는지 찾아보고 만약 커밋 기록이 없으면 오류를 반환하고 커밋 기록이 있다면 기존 커밋 기록 이후 오프셋부터 읽기 시작하며 기본값은 latest

Kafka 개념

- ❖ 카프카 클라이언트
  - ✓ Consumer API
    - ● 주요 옵션
      - ◆ 선택 옵션
        - enable.auto.commit: 자동 커밋으로 할 지 수동 커밋으로 할지 선택하는데 기본값은 true
        - auto.commit.interval.ms: 자동 커밋(enable.auto.commit=true)일 경우 오프셋 커밋 간격을 지정하는데 기본값은 5000(5초)
        - max.poll.records: poll() 메서드를 통해 반환되는 레코드 개수를 지정하는데 기본값은 500
        - session.timeout.ms: 컨슈머가 브로커와 연결이 끊기는 최대 시간으로 이 시간 내에 하트 비트(hearbeat)를 전송하지 않으면 브로커는 컨슈머에 이슈가 발생했다고 가정하고 리밸런싱을 시작하는데 보통 하트비트 시간 간격의 3배로 설정하며 기본값은 10000(10초)
        - hearbeat.interval.ms: 하트비트를 전송하는 시간 간격으로 기본값은 3000(3초)
        - max.poll.interval.ms: poll() 메서드를 호출하는 간격의 최대 시간을 지정하는데 poll() 메서드를 호출한 이후에 데이터를 처리하는 데에 시간이 너무 많이 걸리는 경우 비정상으로 판단하고 리밸런싱을 시작하는데 기본값은 300000(5분)

Kafka 개념

- ❖ 카프카 클라이언트
  - ✓ Consumer API
    - ● 주요 옵션
      - ◆ 선택 옵션
        - ▪ Isolation.level: 트랜잭션 프로듀서가 레코드를 트랜잭션 단위로 보낼 경우 사용하는데 이 옵션은 read_committed, read_uncommitted로 설정할 수 있으며 read_committed로 설정하면 커밋이 완료된 레코드만 읽는데 read_uncommitted로 설정하면 커밋 여부와 관계없이 파티션에 있는 모든 레코드를 읽으며 기본값은 read_uncommitted

Kafka 개념

- ❖ 카프카 클라이언트
  - ✓ Admin API
    - 실제 운영환경에서는 프로듀서와 컨슈머를 통해 데이터를 주고받는 것만큼 카프카에 설정된 내부 옵션을 설정하고 확인하는 것이 중요
    - 내부 옵션을 확인하는 가장 확실한 방법은 브로커 중 한 대에 접속하여 카프카 브로커 옵션을 확인하는 것이지만 매우 번거로운 작업
    - 카프카 커맨드 라인 인터페이스로 명령을 내려 확인하는 방법도 있지만 일회성 작업에 그침
    - 카프카 클라이언트에서는 내부 옵션들을 설정하거나 조회하기 위해 AdminClient 클래스를 제공
    - AdminClient 클래스를 활용하면 클러스터의 옵션과 관련된 부분을 자동화할 수 있음
    - AdminClient 활용하는 예시
      - ◆ 카프카 컨슈머를 멀티 스레드로 생성할 때, 구독하는 토픽의 파티션 개수만큼 스레드를 생 성하고 싶을 때, 스레드 생성 전에 해당 토픽의 파티션 개수를 어드민 API를 통해 가져올 수 있다.
      - ◆ AdminClient 클래스로 구현한 웹 대시보드를 통해 ACL(Access Control List)이 적용된 클 러스터의 리소스 접근 권한 규칙 추가를 할 수 있다.
      - ◆ • 특정 토픽의 데이터양이 늘어남을 감지하고 AdminClient 클래스로 해당 토픽의 파티션을 늘 릴 수 있다.

Kafka 활용

- 파이썬에서 Kafka 사용
  - ✓ kafka-python library를 활용: pip install kafka-python

Kafka 활용

- 파이썬에서 Kafka 사용
  - ✓ 토픽을 생성해서 전송: pythonproducer.py

```
from kafka import KafkaProducer
import json


class MessageProducer:
    def __init__(self, broker, topic):
        self.broker = broker
        self.topic = topic
        #key_serializer=str.encode 를 추가하면 key 와 함께 전송
        #그렇지 않으면 value 만 전송
        self.producer = KafkaProducer(
            bootstrap_servers=self.broker,
            value_serializer=lambda x: json.dumps(x).encode("utf-8"),
            acks=0,
            api_version=(2, 5, 0),
            key_serializer=str.encode,
            retries=3,
        )
```

Kafka 활용

- 파이썬에서 Kafka 사용
  - ✓ 토픽을 생성해서 전송: pythonproducer.py

```
def send_message(self, msg, auto_close=True):
    try:
        print(self.producer)
        future = self.producer.send(self.topic, value=msg, key="key")
        self.producer.flush()  # 비우는 작업
        if auto_close:
            self.producer.close()
        future.get(timeout=2)
        return {"status_code": 200, "error": None}
    except Exception as exc:
        raise exc
```

Kafka 활용

- 파이썬에서 Kafka 사용
  - ✓ 토픽을 생성해서 전송: pythonproducer.py

```
# 브로커와 토픽명을 지정
broker = ["localhost:9092"]
topic = "ec2topic"
pd = MessageProducer(broker, topic)
#전송할 메시지 생성
msg = {"name": "John", "age": 30}
res = pd.send_message(msg)
print(res)
```

Kafka 활용

- 파이썬에서 Kafka 사용
  - ✓ 토픽 수신: pythonconsumer.py

```
from kafka import KafkaConsumer
import json
class MessageConsumer:
    def __init__(self, broker, topic):
        self.broker = broker
        self.consumer = KafkaConsumer(
            topic,  # Topic to consume
            bootstrap_servers=self.broker,
            value_deserializer=lambda x: x.decode(
                "utf-8"
            ),  # Decode message value as utf-8
            group_id="my-group",  # Consumer group ID
            auto_offset_reset="earliest",  # Start consuming from earliest available
message
            enable_auto_commit=True,  # Commit offsets automatically
        )
```

Kafka 활용

- 파이썬에서 Kafka 사용
  - ✓ 토픽 수신: pythonconsumer.py

```
def receive_message(self):
    try:
        for message in self.consumer:
            #print(message.value)
            result = json.loads(message.value)
            for k, v in result.items():
                print(k, ":", result[k])
            print(result["name"])
            print(result["age"])
    except Exception as exc:
        raise exc
```

Kafka 활용

- 파이썬에서 Kafka 사용
  - ✓ 토픽 수신: pythonconsumer.py

```
# 브로커와 토픽명을 지정한다.
broker = ["localhost:9092"]
topic = "exam-topic"
cs = MessageConsumer(broker, topic)
cs.receive_message()
```

Kafka 활용

❖ 파이썬에서 Kafka 사용

✓ python 3.12 버전 사용 시 아래와 같은 에러가 발생하는 경우

ModuleNotFoundError: No module named 'kafka.vendor.six.moves'

◆ 모듈 설치

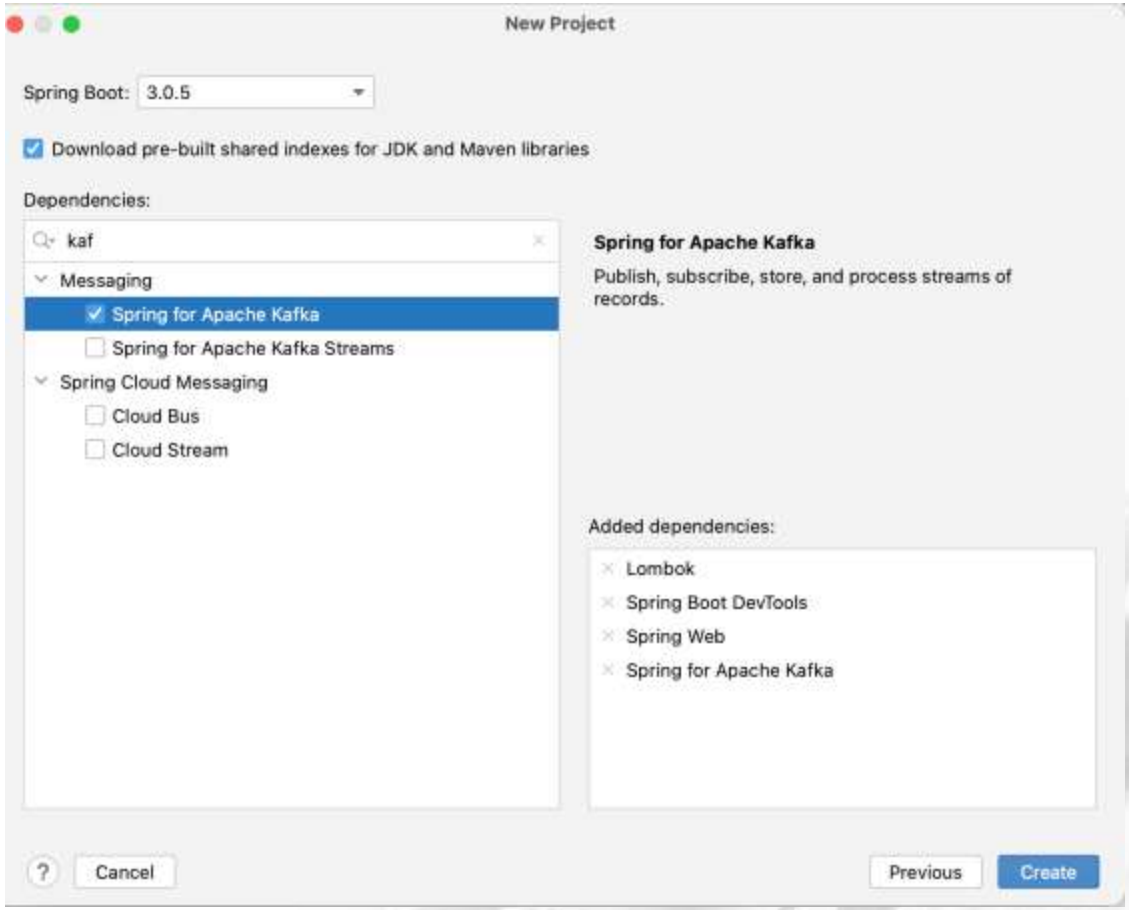
```
pip install six==1.6.0
```

◆ kafka를 import 하기 전에 아래 코드 추가

```
import sys
import six
if sys.version_info >= (3, 12, 0):
    sys.modules['kafka.vendor.six.moves'] = six.moves

from kafka import KafkaConsumer
```

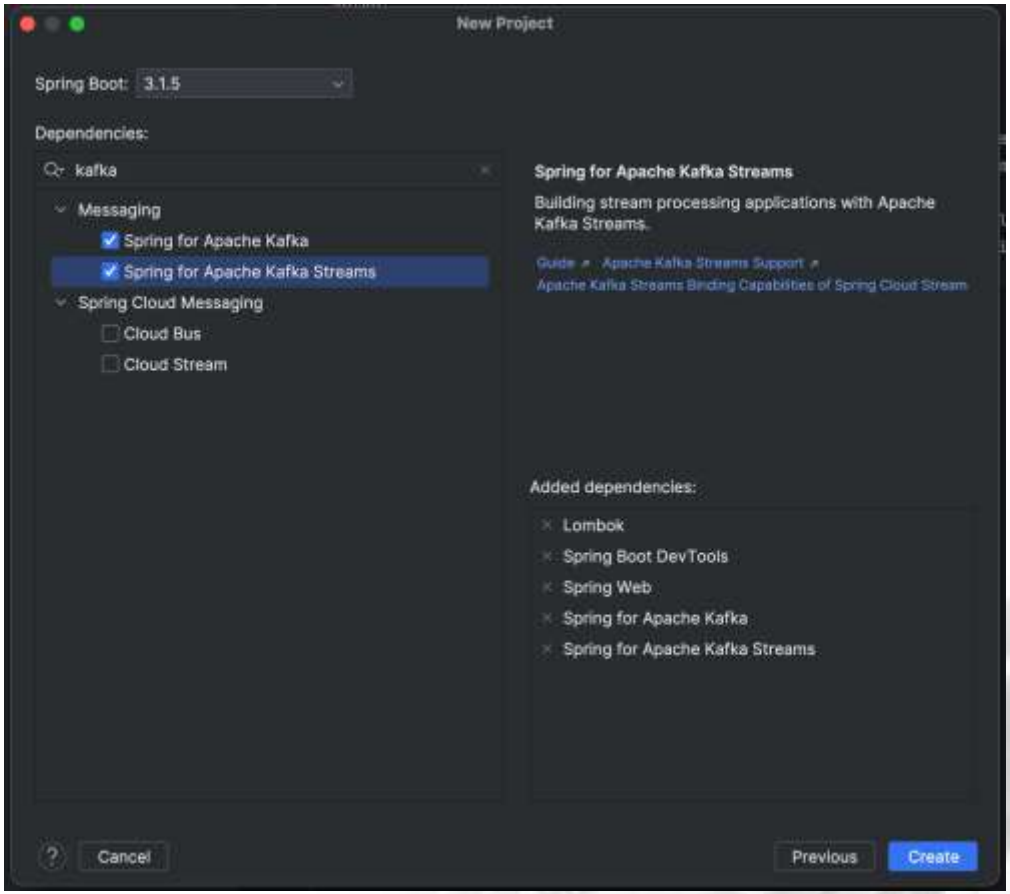
Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ Spring Boot 프로젝트 생성



Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ Spring Boot 프로젝트 생성



Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ Spring Boot 프로젝트에 application.yml 파일을 생성하고 작성

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

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ Spring Boot 프로젝트에 application.yml 파일을 생성하고 작성
    - spring.kafka.bootstrap-servers
      - kafka 서버와 연결할 호스트와 포트 정보
      - producer와 consumer가 다른 서버에 있다면 spring.kafka.consumer(또는 producer).bootstrap-servers으로 설정)
      - 글로벌 설정 정보가 있어도(spring.kafka.bootstrap-servers) consumer 전용으로 오버라이딩
    - consumer.group-id
      - consumer group을 구별하기 위한 유니크한 id 작성
    - auto-offset-reset
      - Kafka 서버에 초기 offset이 없거나 서버에 현재 offset이 더 이상 없는 경우 수행할 작업을 작성
      - earliest: 가장 오래된 메세지로 offset reset
      - latest: 가장 최근에 생산된 메세지로 offset reset
      - none: offset 정보가 없으면 Exception을 발생

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ Spring Boot 프로젝트에 application.yml 파일을 생성하고 작성
    - key-deserializer/value-desrializer
      - Kafka에서 consumer가 데이터를 받아올 때 key/value를 역직렬화
      - 우리가 주고 받는 데이터의 형태는 String이므로 StringDeserializer를 사용

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ Spring Boot 프로젝트에 Kafka 설정 클래스 생성 - KafkaConfiguration

```
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;

import java.util.HashMap;
import java.util.Map;

@Configuration
public class KafkaConfiguration {
```

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ Spring Boot 프로젝트에 Kafka 설정 클래스 생성 - KafkaConfiguration

```
@Value("${spring.kafka.bootstrap-servers}")
private String bootstrapServers;

@Bean
public ProducerFactory<String, String> producerFactory() {

    Map<String,Object> configs = new HashMap<>();
    configs.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
    configs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
 StringSerializer.class);
    return new DefaultKafkaProducerFactory(configs);
}

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

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ Spring Boot 프로젝트에 메시지 게시 서비스 클래스 생성 - KafkaProducer

```
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

@Service
@Slf4j
@RequiredArgsConstructor
public class KafkaProducer {
    private static final String TOPIC = "exam-topic";

    @Autowired
    private   KafkaTemplate<String, String> kafkaTemplate;
     private final Logger log = LoggerFactory.getLogger(getClass());
```

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ Spring Boot 프로젝트에 메시지 게시 서비스 클래스 생성 - KafkaProducer

```
    public void sendMessage(String name, int age) {
        log.info("Produce message : {}{}", name, age);
        String message = "{₩"name₩":" + "₩"" + name + "₩"" + ", ₩"age₩":" + age +
     "}";
        this.kafkaTemplate.send(TOPIC, message);
    }
}
```

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ Spring Boot 프로젝트의 build.gradle 파일에 의존성 추가

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.apache.kafka:kafka-streams'
    implementation 'org.springframework.kafka:spring-kafka'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'org.springframework.kafka:spring-kafka-test'
    implementation 'org.json:json:20190722'
}
```

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ Spring Boot 프로젝트에 메시지 구독 서비스 클래스 생성 – KafkaConsumer

```
import lombok.extern.slf4j.Slf4j;
import org.json.JSONObject;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

import java.io.IOException;

@Service
@Slf4j
public class KafkaConsumer {
    private final Logger log = LoggerFactory.getLogger(getClass());

    @KafkaListener(topics = "exam-topic", groupId = "adamsoft")
    public void consume(String message) throws IOException {
        log.info("Consumed message : {}", message);
        JSONObject messageObj = new JSONObject(message);
        log.info(messageObj.getString("name"));
        log.info(messageObj.getInt("age") + "");
    }
}
```

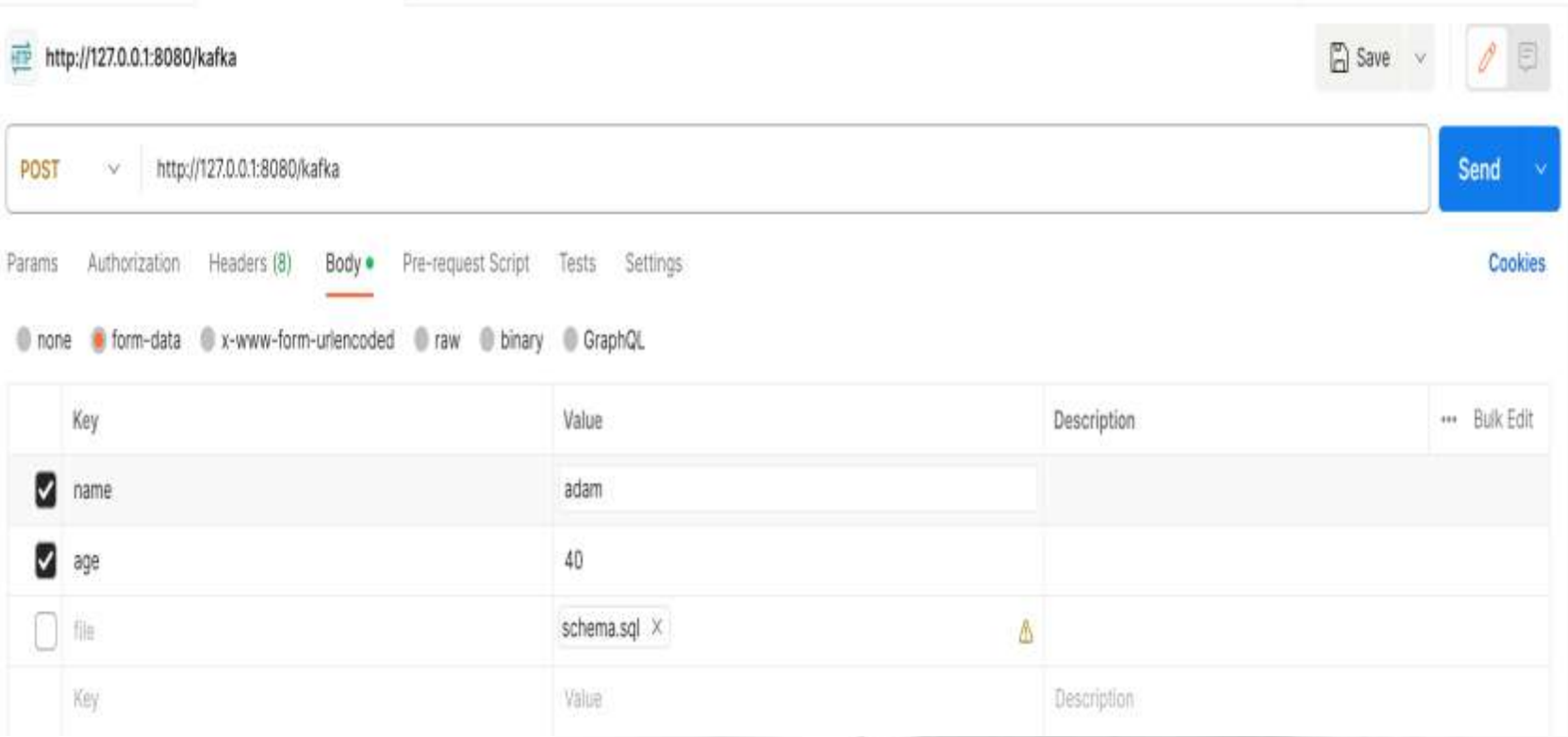
Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ Spring Boot 프로젝트에 메시지 Controller 클래스 생성 – KafkaController

```
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping(value = "/kafka")
@Slf4j
@RequiredArgsConstructor
public class KafkaController {

    @Autowired
    private KafkaProducer producer;

    @PostMapping
    @ResponseBody
    public String sendMessage(@RequestParam("name") String name,
     @RequestParam("age") int age) {
        this.producer.sendMessage(name, age);
        return "success";
    }
}
```

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ 프로젝트 실행 및 테스트



Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독
  - ✓ 프로젝트 실행 및 테스트

```
2023-04-12T17:17:24.543+09:00  INFO 30888 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Completed initialization in 0 ms
2023-04-12T17:17:24.592+09:00  INFO 30888 --- [nio-8080-exec-2] c.example.kafkamessage.KafkaController   : message : Spring 카프카,
2023-04-12T17:17:24.592+09:00  INFO 30888 --- [nio-8080-exec-2] com.example.kafkamessage.KafkaProducer   : Produce message : Spring 카프카,
2023-04-12T17:17:24.605+09:00  INFO 30888 --- [nio-8080-exec-2] o.a.k.clients.producer.ProducerConfig    : ProducerConfig values:
```

```
bash-5.1# kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic ad
am-topic --from-beginning
Spring 카프카,
Spring 카프카,
```

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독 – JSON 전송
  - ✓ KafkaConfiguration, KafkaProducer, KafkaConsumer 클래스를 삭제하거나 주석 처리
  - ✓ application.yml 파일 수정

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
```

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독 – JSON 전송
  - ✓ DTO 클래스 생성 – ChatMessage

```
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;

@Getter
@NoArgsConstructor
@AllArgsConstructor
public class ChatMessage {

    private String name;
    private int age;
}
```

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독 – JSON 전송
  - ✓ 메시지 게시 설정 클래스 – KafkaProducerConfig

```
import org.springframework.kafka.support.serializer.JsonSerializer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;

import java.util.HashMap;
import java.util.Map;
```

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독 – JSON 전송
  - ✓ 메시지 게시 설정 클래스 – KafkaProducerConfig

```
@EnableKafka
@Configuration
public class KafkaProducerConfig {
    @Value("${spring.kafka.bootstrap-servers}")
    private String servers;

    @Bean
    public ProducerFactory<String, ChatMessage> producerFactory() {
        Map<String, Object> config = new HashMap<>();
        config.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, servers);
        config.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        config.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);
        return new DefaultKafkaProducerFactory<>(config);
    }

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

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독 – JSON 전송
  - ✓ 메시지 게시 서비스 클래스 – KafkaProducerService

```
import lombok.RequiredArgsConstructor;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class KafkaProducerService {
    private static final String TOPIC = "exam-topic";
    private final KafkaTemplate<String, ChatMessage> kafkaTemplate;

    public void sendMessage(ChatMessage chatmessage) {
        kafkaTemplate.send(TOPIC, chatmessage);
    }
}
```

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독 – JSON 전송
  - ✓ 메시지 구독 설정 클래스 – KafkaConsumerConfig

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.support.serializer.JsonDeserializer;
import java.util.HashMap;
import java.util.Map;
```

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독 – JSON 전송
  - ✓ 메시지 구독 설정 클래스 – KafkaConsumerConfig

```
@EnableKafka
@Configuration
public class KafkaConsumerConfig {
    @Value("${spring.kafka.bootstrap-servers}")
    private String servers;
    @Bean
    public ConsumerFactory<String, ChatMessage> consumerFactory() {
        Map<String, Object> config = new HashMap<>();
        config.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, servers);
        config.put(ConsumerConfig.GROUP_ID_CONFIG, "testgroup");
        return new DefaultKafkaConsumerFactory<>(config, new StringDeserializer(), new
     JsonDeserializer<>(ChatMessage.class));
    }
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, ChatMessage>
     kafkaListener() {
        ConcurrentKafkaListenerContainerFactory<String, ChatMessage> factory = new
     ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        return factory;
    }
}
```

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독 – JSON 전송
  - ✓ 메시지 구독 서비스 클래스 – KafkaConsumerService

```
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class KafkaConsumerService {

    @KafkaListener(topics = "exam-topic", groupId = "examtopic", containerFactory =
     "kafkaListener")
    public void consume(ChatMessage message){
        System.out.println("name = " + message.getName());
        System.out.println("age = " + message.getAge());
    }
}
```

Kafka 활용

❖ 하나의 자바 애플리케이션에서 메시지 게시 와 구독 – JSON 전송

✓ Controller 클래스 – KafkaController

```
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping(value = "/kafka")
@Slf4j
@RequiredArgsConstructor
public class KafkaController {

    private final KafkaProducerService producerService;


    @PostMapping
    @ResponseBody
    public String sendMessage(@RequestBody ChatMessage chatmessage) {
        System.out.println("chatmessage = " + chatmessage);
        producerService.sendMessage(chatmessage);

        return "success";
    }
}
```

Kafka 활용

- 하나의 자바 애플리케이션에서 메시지 게시 와 구독 – JSON 전송
  - ✓ 테스트 및 확인