카프카 스키마 레지스트리

서론

- 스키마란 정보를 구성하고 해석하는 것을 도와주는 프레임워크나 개념을 의미합니다.
- 스키마는 정보를 손쉽게 이해하고 해석하는 데 쓰이며, 특히 데이터베이스의 구조를 정의하고 표현 방법이나 전반적인 명세와 제약 조건을 기술하는 표준 언어로 활용
- 스키마의 정의를 통해 데이터베이스의 관리 효율성이 높아지며, 데이터 충돌을 방지할 수도 있음
- 카프카에서도 토픽으로 전송되는 메시지에 대해 미리 스키마를 정의한 후 전송함으로써 데이터베이스에서 얻을 수 있는 동일한 효과를 얻을 수 있음.
- `스키마 레지스트리` 라는 애플리케이션을 이용해 카프카에서 어떻게 스키마를 정의하고 활용할 수 있는지 살펴보자.

스키마의 개념과 유용성

- 관계형 데이터베이스의 경우
  - 스키마가 미리 정의되어 있고, 데이터를 추가하기 위해서는 반드시 사전에 정의된 스키마의 형태로 데이터를 입력해야 함.
  - 만약 스키마가 없다면 데이터 입력에 실수를 저질렀을 때, 문자열 오류나 파싱 에러가 나며 장애로 이어질 수 있음.
- 카프카에 스키마가 없다면? 첫번째
  - 단 한 명의 사용자만 카프카의 토픽에 읽고 쓰기를 실행한다면 스키마가 정의되어 있지 않더라도 별다른 이슈가 없을 것
  - 중앙 데이터 파이프라인 역할을 하는 카프카에서는 수많은 사용자(또는 수많은 애플리케이션) 가 수많은 토픽을 이용할 것
  - 누군가의 실수로 사전에 정의하지 않은 형태의 데이터를 해당 토픽으로 보낸다면, 연결된 모든 시스템이 영향을 받고 심각한 문제로 이어질 수 있음
- 카프카에 스키마가 없다면? 두번째
  - 카프카의 데이터 흐름은 대부분 broadcast 방식임 ⇒ 카프카는 데이터를 전송하는 프로듀서를 일방적으로 신뢰할 수 밖에 없는 방식
  - 프로듀서 관리자는 카프카 토픽의 데이터를 컨슘하는 관리자에게 반드시 데이터 구조를 설명해야 함.
  - 컨슘하는 관리자가 많다면 그때마다 설명하는 것은 어려운 일
  - **데이터를 컨슘하는 여러 부서에게 그 데이터에 대한 정확한 정의와 의미를 알려주는 역할을 하는 것이 바로 스키마**
- 데이터 흐름에서 가장 어려운 애로사항을 해결하는 역할도 스키마가 함.
  - 기업에서 비즈니스 변화에 따라 모델링이나 데이터 포맷이 변경되면 새로운 필드를 추가하거나 필드 이름을 변경 또는 삭제 해야함.

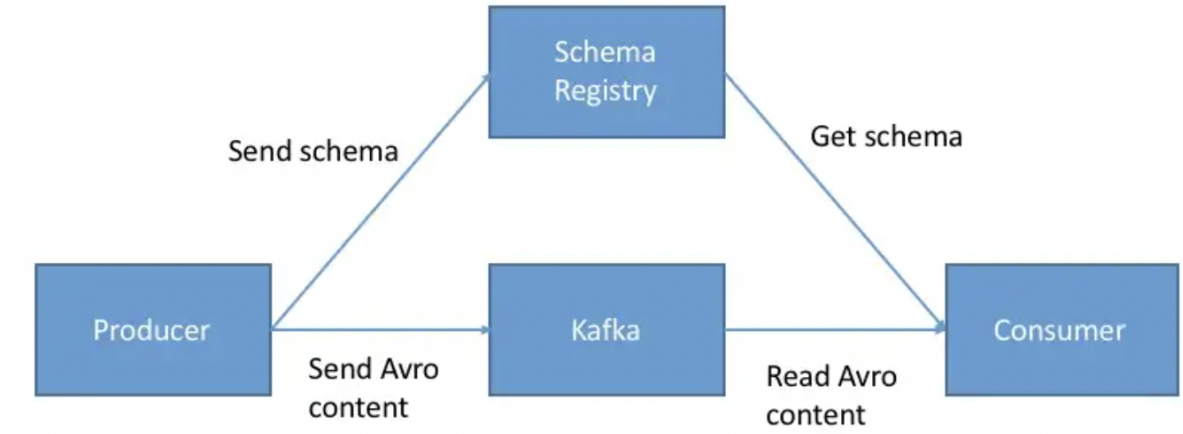
    - 스키마의 변경이 일어날 때마다 변경된 내용을 유관 부서 담당자들에게 알려줘야 함.
    - 중앙 데이터 파이프라인 역할을 하는 카프카에서는 이러한 스키마의 진화를 지원
- 수많은 데이터 엔지니어나 데이터 분석가들이 분석, 적재 업무보다 데이터 파싱 작업에 많은 시간을 보내고 있음
    - 매번 형식을 벗어난 데이터 필드를 찾고 파싱하고 에러 처리하는 데 시간을 허비하는 것은 안타까운 일
- 스키마 정의했을 때, 단점은?
    - 데이터를 처리하기에 앞서 스키마를 정의하는 작업 자체가 시간과 노력이 듬
    - json처럼 간단하게 정의하고 사용할 수 있는 것이 아니라, 데이터 타입은 무엇이며 데이터가 의미하는 바는 무엇인지 주석을 달아 상세하게 표시해야함 ⇒ 스키마를 미리 정의하는 작업이 개발자에게는 매우 불편하고 번거로운 일
- 스키마 정의했을 때, 장점은?
    - 데이터 트러블 슈팅 감소
    - 용이한 데이터 포맷 확인
    - 데이터 스키마 관련 커뮤니케이션 감소

> 💡 따라서 카프카에서 스키마 사용은 **권장사항**에 속합니다.

카프카와 스키마 레지스트리

스키마 레지스트리 개요

- 카프카에서 스키마를 활용하는 방법은 스키마 레지스트리라는 별도의 애플리케이션을 이용하는 것
- 스키마 레지스트리란?
    - 글자 그대로 스키마를 등록하고 관리하는 애플리케이션
    - 2015년 2월 아파치 카프카 0.8.2 버전 출시와 함께 처음 공개 되었음.
    - 아파치 오픈소스 라이선스가 아닌 컨플루언트 커뮤니티 라이선스 ⇒ 비상업적 용도에서만 무료



- 그림에서 알 수 있듯이 스키마 레지스트리는 카프카와 별도로 구성된 독립적 애플리케이션
- 프로듀서, 컨슈머와 직접 통신함.
- 클라이언트들이 스키마 정보를 사용하기 위해서는 **프로듀서와 컨슈머, 스키마 레지스트리간에 직접 통신이 이뤄져야 함.**
  1. 프로듀서는 스키마 레지스트리에 스키마를 등록
  2. 스키마 레지스트리는 프로듀서에 의해 등록된 스키마의 ID와 메시지를 카프카로 전송
  3. 컨슈머는 스키마 ID를 스키마 레지스트리로부터 읽어 온 후 프로듀서가 전송한 스키마 ID와 메시지를 조합해 읽을 수 있음.
- 스키마 레지스트리를 이용하기 위해서는 스키마 레지스트리가 지원하는 데이터 포맷을 사용해야 함.
  - 가장 대표적인 포맷은 **Avro 임.**

스키마 레지스트리의 에이브로 지원

- Avro
  - 시스템, 프로그래밍 언어, 프로세싱 프레임워크 사이에서 데이터 교환을 도와주는 오픈소스 직렬화 시스템
  - 빠른 바이너리 포맷을 지원하며 JSON 형태의 스키마를 정의할 수 있는 매우 간결한 포맷
- 스키마 레지스트리는 Avro 포맷을 가장 먼저 지원했으며, 최근에는 JSON, Protocol Buffer 포맷도 지원함.
- 가장 대중적으로 많이 사용하는 포맷은 JSON이겠지만, 컨플루언트는 다음과 같은 이유로 스키마 레지스트리에서 에이브로 포맷 사용을 권장함.
  - Avro는 JSON과 매핑 된다.
  - Avro는 매우 간결한 데이터 포맷이다.
  - JSON은 메시지마다 필드 네임들이 포함되어 전송되므로 효율이 떨어진다.

    - Avro는 바이너리 형태이므로 매우 빠르다.

대략적인 개념만 살펴봤지만, 실습을 직접 진행해보면 위 장점들이 어떤 의미인지 이해할 수 있을 것

- avro를 활용해 스키마 예제 파일을 만들어보자.
    - 한 반의 학생 명단을 작성할 것

```
// student.avro
{"namespace": "student.avro",
 "type": "record",
 "doc": "This is an example of Avro.",
 "name": "Student",
 "fields": [
   {"name": "name", "type": "string", "doc": "Name of the student"},
   {"name": "class", "type": "int", "doc": "Class of the student"}
 ]
}
```

- namespace: 이름을 식별하는 문자열
- type: Avro는 record, enums, arrays, maps 등을 지원하며, 여기서는 record 타입으로 정의
- doc: 사용자들에게 이 스키마 정의 대한 설명 제공 (주석)
- name: 레코드의 이름을 나타내는 문자열(required)
- fields: JSON 배열로서, 필드의 리스트를 뜻함.
- name: 필드의 이름
    - type: boolean, int, long, string 등의 데이터 타입 정의
    - doc: 사용자들에게 해당 필드의 의미 설명 (주석)

- JSON과 달리 avro는 데이터 필드마다 데이터 타입을 정의할 수 있고, doc을 이용해 각 필드의 의미를 데이터를 사용하고자 하는 사용자들에게 정확하게 전달할 수 있음.
- doc(주석) 기능을 통해 별도의 문서를 작성할 필요 없이 스키마로 유지할 수 있음.

스키마 레지스트리 설치

- ansible 방식이 아닌 container로 설치했습니다.
    - 브로커 1대: schema-registry-kafka-0
    - 주키퍼 1대: schema-registry-zookeeper-0
    - 스키마 레지스트리 1대: schema-registry-0

```
% k get po -n kafka | grep schema-registry
NAME                           READY   STATUS    RESTARTS   AGE
schema-registry-0              1/1     Running   0          61m
schema-registry-kafka-0        1/1     Running   0          96m
schema-registry-zookeeper-0    1/1     Running   0          96m
```

- 살펴봐야 할만한 설정
  - 스키마 레지스트리에서 스키마 저장과 관리 목적으로 카프카의 토픽을 사용한다는 점
  - 스키마 호환성 레벨은 BACKWORD, FORWARD, FULL 세 종류가 있는데, 추후 자세히 알아볼 것

```
listeners=http://0.0.0.0:8081
kafkastore.bootstrap.servers=${설치한 broker 주소)
kafkastore.topic=_schemas
schema.compatibility.level=full
```

> 이 책에서는 스키마 레지스트리를 단일 모드 형태로 구성하지만, 운영 환경에서는 이중화 구성을 추천. 2 개의 스키마 레지스트리를 로드밸런서 등을 이용해 바인딩하고, leader.eligibility=true 옵션을 적용하면 된다. 2개의 스키마 레지스트리는 마스터/슬레이브 구조로서, 스키마 레지스트리의 로그를 통해 누가 현재 리더 역할을 하는지 알 수 있음.

> 기본적으로 스키마 레지스트리의 읽기는 마스터와 슬레이브 양쪽 모두 허용되고 스키마 레지스트리에 쓰기는 마스터 노드만 가능함.

- 브로커의 _schemas 토픽이 스키마 레지스트리의 저장소로 활용되며, 모든 스키마의 제목, 버전, ID 등이 저장 됨.
- **스키마 관리 목적으로 사용되는 메시지들은 순서가 중요하기 때문에, _schemas 토픽의 파티션 수는 항상 1이다.**
- 그리고 주의 깊게 살펴봐야할 토픽 설정은 cleanup.policy가 `compact` 로 설정됨.

  ⇒ cleanup.policy: 토픽 레벨에서 로그 컴팩션을 설정할 때 적용하는 옵션 (토픽 단위)

  ⇒ 키 값을 기준으로 여러 값이 존재할 때 가장 마지막 값만 남겨두고 날리는 설정인듯 아마도.

중간 실습

1. schema registry가 잘 실행중인지 살펴보기

```
k exec kafka-consumer -n kafka -it -- /bin/bash
curl schema-registry:8081/config
```

- API 응답을 통해 schema registry의 호환성 레벨이 FULL로 출력되는 것을 확인할 수 있음.

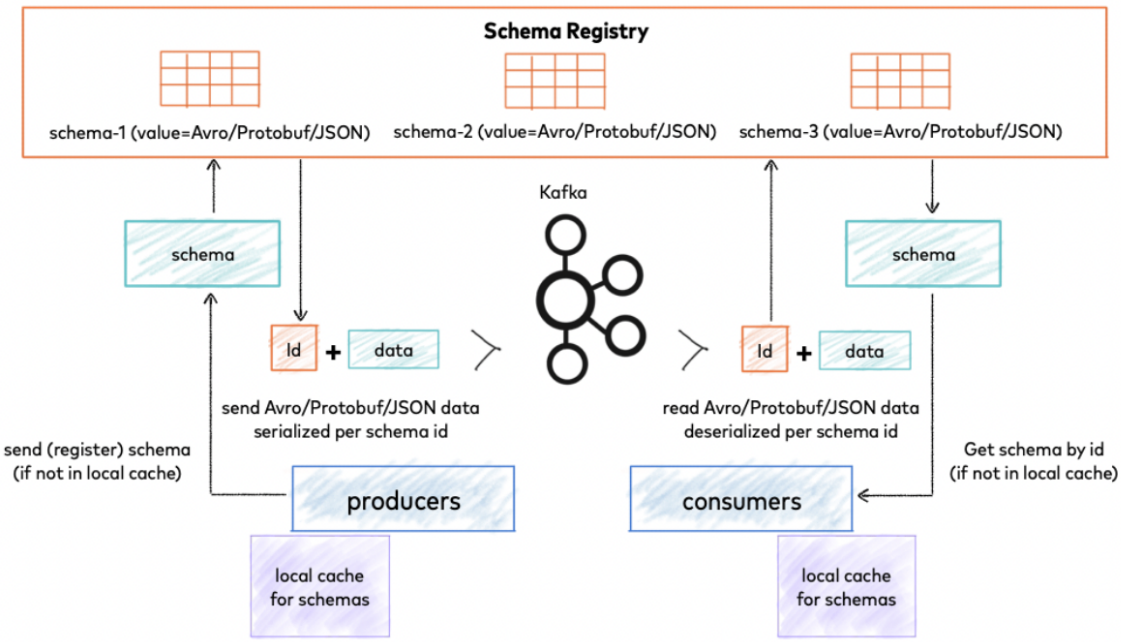
2. _schemas 토픽의 설정 살펴보기
    - http://10.128.168.67/ui/clusters/local/all-topics/_schemas/settings

- 스키마 레지스트리는 HTTP 기반으로 통신이 이뤄지며, 사용자들의 유연성과 편의성을 위해 스키마 레지스트리의 주요 기능들에 대한 API를 제공
- 이 API를 통해 애플리케이션 개발을 간소화할 수 있으며, 시간도 절약함.

옵션	설명
GET /schemas	현재 스키마 레지스트리에 등록된 전체 스키마 리스트 조회
GET /schemas/ids/id	스키마 ID로 조회
GET /schemas/ids/id/versions	스키마 ID의 버전
GET /subjects	스키마 레지스트리에 등록된 subject 리스트, subject는 토픽이름-key, 토픽이름-value 형태로 쓰임
GET /subjects/서브젝트 이름/versions	특정 서브젝트의 버전 리스트 조회
GET /config	전역으로 설정된 호환성 레벨 조회
GET /config/서브젝트 이름	서브젝트에 설정된 호환성 조회
DELETE /subjects/서브젝트 이름	특정 서브젝트 전체 삭제
DELETE /subjects/서브젝트 이름/versions/버전	특정 서브젝트에서 특정 버전만 삭제

스키마 레지스트리와 클라이언트 동작

- 카프카 모델은 pub/sub 모델로서, 프로듀서와 컨슈머는 직접 통신을 주고받지 않음.
- 스키마가 정의되어있는 메시지를 컨슈머가 읽기 위해서는 프로듀서가 정의한 스키마 정보를 알아야함
    - 직접 통신하지 않는데 컨슈머는 어떻게 프로듀서가 정의한 스키마 정보를 알 수 있을까?



Confluent Schema Registry for storing and retrieving schemas

1. Avro 프로듀서는 컨플루언트에서 제공하는 `io.confluent.kafka.serializers.KafkaAvroSerializer`라는 새로운 직렬화를 사용해 스키마 레지스트리의 스키마가 유효한지 여부를 확인. 만약 스키마가 확인되지 않으면, Avro 프로듀서는 스키마를 등록하고 캐시함.
2. 스키마 레지스트리는 현 스키마가 저장소에 저장된 스키마와 동일한 것인지, 진화된 스키마인지 확인함. 스키마 레지스트리 자체적으로 각 스키마에 대해 고유 ID를 할당. 이 ID는 순차적으로 1씩 증가하지만, 반드시 연속적이지 않을 수 있음. 스키마에 문제가 없다면 스키마 레지스트리는 프로듀서에게 고유 ID를 응답
3. 이제 프로듀서는 스키마 레지스트리로부터 받은 스키마ID를 참고해 메시지를 카프카로 전송. 이 때 프로듀서는 스키마 전체 내용이 아닌 오로지 메시지와 스키마 ID만 보냄. JSON은 key:value 형태로 전체 메시지를 보내야 하지만, Avro를 사용하면 스키마 ID와 value만 보내게 되어 전체 메세지 크기를 줄일 수 있음.
4. Avro 컨슈머는 스키마 ID로 컨플루언트에서 제공하는 `io.confluent.kafka.serializers.KafkaAvroDeSerializer` 라는 새로운 역직렬화를 사용해 카프카의 토픽에 저장된 메시지를 읽음. 이 때 컨슈머가 스키마 ID를 갖고 있지 않다면 스키마 레지스트리로부터 가져옴.

- 이와 같이 프로듀서와 컨슈머는 직접 스키마를 주고받지 않지만, 스키마 레지스트리와 통신하면서 스키마의 정보를 주고 받음.
- 스키마 정보를 레지스트리에 등록함으로써 프로듀서가 전송하는 메시지의 크기도 줄일 수 있고, 컨슈머가 읽는 메시지의 크기도 줄일 수 있음.
- 또한 사전에 정의하지 않은 형식의 메시지는 전송할 수 없으므로 데이터를 처리하는 쪽에서 협의되지 않은 메시지가 들어오는 경우에 대해 고민하지 않아도 됨.

본 실습

- 파이썬을 이용해 스키마 레지스트리를 실습

```
clone 해서 진행하자. => https://github.com/onlybooks/kafka2.git

vim ~/kafka2/chapter10/python-avro-producer.py // 수정
python ~/kafka2/chapter10/python-avro-producer.py

// 메시지 작성 확인

vim ~/kafka2/chapter10/python-avro-consumer.py // 수정
python ~/kafka2/chapter10/python-avro-consumer.py

// 스키마 레지스트리 확인
curl http://schema-registry:8081/schemas | python -m json.tool

deactivate // 가상환경 해제
```

- default값으로 null, 1을 지정할 수 있음을 언급
- 기본적인 스키마 레지스트리의 동작 확인 완료

스키마 변경 실습

- 이전 스키마에서는 이름과 반만 정의했지만, 한 반에 이름이 중복되는 현상이 발생할 수 있음
- 이름 중복 문제를 해결하기 위해, 학생의 성과 이름 2개 항목을 모두 저장해보자.
  - name 필드를 삭제
  - first_name, last_name 이라는 2개의 필드 추가
- 만약 스키마 레지스트리를 도입하지 않은 상황이라면, 데이터 처리 업무를 맡은 담당자들에게 변경 사항을 공유해야할 뿐 아니라 프로듀서와 컨슈머의 코드 변경 및 다른 클라이언트를 중지 후 배포해야하는 곤란한 상황
- 하지만 스키마 레지스트리를 사용하고 있으므로 비교적 간단하게 변경된 내용을 적용할 수 있음.

```
// python-avro-producer2.py
// field 및 메시지 변경 확인

전송했을 때 에러가 나지 않고 계속 메시지를 받았다.

// python-avro-consumer2.py
// 진화된 스키마르 컨슈머에 적용해보자.
// 처음부터 받기 위해 group id도 변경
// 출력결과 컨슈머가 2개의 메시지를 읽었을 것. 둘 다 잘 가져왔는지 확인
```

- 스키마 레지스트리를 도입하지 않는 상황이라면, 메시지 형식이 바뀔 때마다 컨슈머에서는 문자열 파싱에러가 발생해 컨슈머 프로세스가 종료되기 때문에 코드를 수정해야 하는 일이 빈전
- 하지만 스키마 레지스트리를 사용하면, 프로듀서가 보내는 메시지의 스키마가 바뀌어도 오류 없이 메시지를 읽을 수 있음.
- 그리고 컨슈머 작업이 가능한 시간에 맞추어 새로운 스키마를 적용한다면 아무런 이슈없이 처리 가능

스키마 레지스트리 호환성

- 스키마 레지스트리는 버전별 스키마에 대한 관리를 효율적으로 해주며, 각 스키마에 대한 고유한 ID와 버전 정보를 관리합니다.
- 스키마 레지스트리에서 하나의 서브젝트에 대한 버전 정보별로 진화하는 각 스키마를 관리함.
- 스키마가 진화함에 따라 호환성 레벨을 검사해야 하는데, 스키마 레지스트리에서는 대표적으로 BACKWARD, FORWARD, FULL 등의 호환성 레벨을 제공

BACKWARD 호환성

- 스키마 레지스트리를 사용하다 보면 데이터 포맷의 변경이 필요하며, 데이터 포맷을 변경하는 스키마는 진화함.
- 스키마가 진화함에 따라 스키마 레지스트리 내에는 버전별 스키마를 갖게 됨.
- BACKWARD 호환성이란 “진화된 스키마를 적용한 컨슈머가 진화 전의 스키마가 적용된 프로듀서가 보낸 메시지를 읽을 수 있도록 허용하는 호환성”

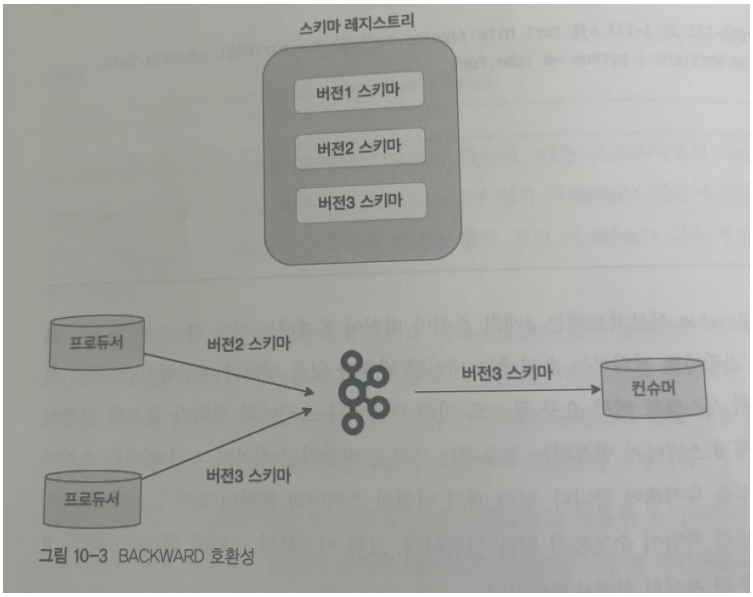
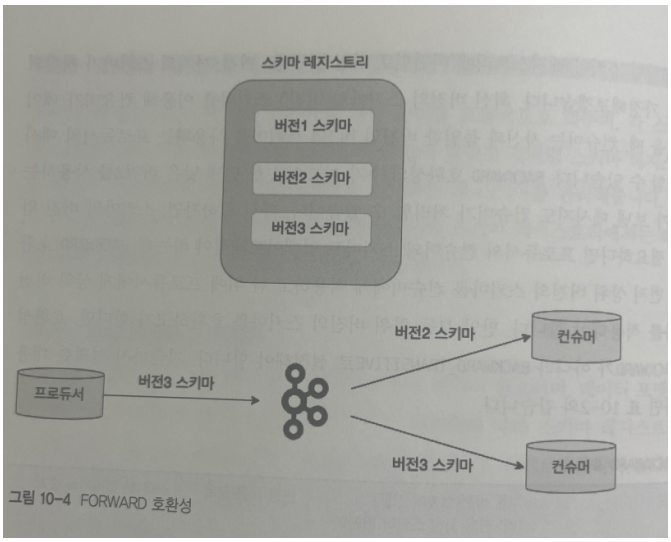


그림 10-3 BACKWARD 호환성

- 스키마 레지스트리에 버전별로 버전1, 버전2, 버전3까지의 스키마가 저장되어 있다고 가정할 때, 최신 스키마인 버전 3 스키마를 이용해 컨슈머가 데이터를 가져올 때, 컨슈머는 자신과 동일한 버전3 스키마를 사용하는 프로듀서의 메시지를 처리할 수 있음.
- 이 때 BACKWARD 호환성의 추가 기능으로 한 단계 낮은 버전2를 사용하는 프로듀서가 보낸 메시지도 컨슈머가 처리할 수 있음.
- 스키마의 버전 업데이트가 필요하다면 프로듀서와 컨슈머의 스키마도 업데이트해줘야 하는데, BACKWARD 호환성에서는 **먼저 상위 버전의 스키마를 컨슈머에게 적용하고 난 뒤에 프로듀서에게 상위 버전의 스키마를 적용해야 함**.
- 만약 모든 버전의 하위 버전 스키마를 호환하고자 한다면 `BACKWARD_TRANSITIVE` 로 설정해야 함.
  - 변경 허용 항목: 필드 삭제, 기본 값이 지정된 필드 추가
  - 스키마 업데이트 순서: 컨슈머 ⇒ 프로듀서

FORWARD 호환성

- `FORWARD` 호환성이라는 용어대로 `BACKWARD` 대비되며, **진화된 스키마가 적용된 프로듀서가 보낸 메시지를 진화 전의 스키마가 적용된 컨슈머**가 읽을 수 있게 하는 호환성을 말함.



- 그림 10-4처럼 스키마 레지스트리에 버전별로 버전1, 버전2, 버전3의 스키마가 저장되어있다고 가정하자.
- 최신 버전 스키마인 버전3 스키마를 이용해 프로듀서가 메시지를 전송할 때, 프로듀서와 동일한 버전인 버전3를 사용한 컨슈머는 당연히 처리가 가능함.
- 이 때 `FORWARD` 호환성의 추가 기능으로 **한 단계 낮은 버전2 스키마를 이용하는 컨슈머도 버전3 스키마를 이용하는 컨슈머도 버전3 스키마를 이용하는 프로듀서가 보내는 메시지를 처리할 수 있음**.
- 따라서 스키마 진화가 일어나는 경우 상위 버전의 스키마를 프로듀서에 먼저 적용함.
  - 다시 정리하면, FORWARD는 상위 버전 스키마를 먼저 프로듀서에게 적용한 다음, 컨슈머에게 적용

- 마찬가지로 한 단계가 아닌 모든 버전의 스키마를 호환하고 싶다면 `FORWARD_TRANSITIVE` 로 설정해야 함.
  - 변경 허용 항목 : 필드 추가, 기본 값이 지정된 필드 삭제
  - 스키마 업데이트 순서: 프로듀서 ⇒ 컨슈머

FULL 호환성

- FULL 호환성은 BACKWARD, FORWARD 를 모두 지원합니다.
- 스키마가 진화함에 따라 프로듀서 측면과 컨슈머 측면 양쪽에서 호환되므로 앞서 살펴본 BACKWARD, FORWARD 호환성에 비해 제약 없이 더 편리하게 사용할 수 있다는 장점이 있습니다.
- 스키마 지원 버전은 가장 최근 2개의 버전 스키마를 지원하며, 모든 버전의 스키마를 호환하고자 한다면, FULL_TRANSITIVE로 설정해야 합니다.
  - 변경 허용 항목: 기본 값이 지정된 필드 추가, 기본 값이 지정된 필드 삭제
  - 스키마 업데이트 순서: 상관 없음

스키마 레지스트리 호환성 실습

- 많은 사용자가 기본값을 사용하는 경우가 많으므로 기본값인 `BACKWARD` 로 스키마 레지스트리 변경해서 실습을 진행해보자.
- `BACKWARD` 의 경우 컨슈머를 먼저 상위 버전의 스키마로 업데이트 해야 함.
- 하지만 프로듀서에게 먼저 상위 버전 스키마를 적용했을 때 어떻게 되는지 살펴보자.

```
스키마 레지스트리 호환성 살펴보기
k exec ubuntu -n kafka -it -- /bin/bash
curl http://schema-registry:8081/config

helm upgrade schema-registry bitnami/schema-registry --namespace kafka -f override-schema-registry.yaml

// 그다음 토픽만들기, 브로커 한 개니까 리플리케이션 팩터 1로
kafka-ui: 10.128.168.67
스키마 레지스트리: http://schema-registry:8081
카프카 브로커: schema-registry-kafka


// v1 스키마를 적용한 프로듀서 python-avro-producer_v1.py
source venv10/bin/activate
python python-avro-consumer_v1.py


// 이제 컨슈머가 메시지를 잘 가져오는지 확인
// v1 스키마를 적용한 컨슈머 python-avro-consumer_v1.py

// `BACKWARD` 이므로 클라이언트 업데이트 순서상 컨슈머를 먼저 업데이트해야하지만,
// 호환성 규칙을 무시하고 프로듀서에 진화한 스키마를 적용해보자.
```

```
// v2 스키마를 적용한 프로듀서 python-avro-producer_v2.py

// 스키마와 메시지에 age가 추가된 것을 확인.
// 그리고 전송하고 python-avro-consumer_v1.py가 가져왔는지 확인
// 가져왔으나 age 필드가 없음을 확인! => 진화하는 스키마를 지원하는 호환성 동작이 아님.

// 컨슈머 종료하고, V2 기반 스키마르 컨슈머 실행
// python-avro-consumer_v2.py
// 다시 V2 프로듀서로 메시지 전송해보고 V2 consumer 읽어오자.
// 이번에는 프로듀서가 보낸 메시지를 정확히 가져오고 age 필드가 추가된 것 확인

// 마지막으로 V2 컨슈머 상태에서 v1 프로듀서로 메시지를 전송했을 때 출력 화면 확인
// age 필드를 무시하지 않고, default값 1로 잘 출력되는 것 확인
```

- 이렇게 호환성 레벨에 따라 상위 버전의 스키마를 프로듀서에 먼저 적용할지, 컨슈머에 먼저 적용할지가 달라짐.
- 스키마 레지스트리를 이용하는 회사나 단체의 내부 정책에 따라 스키마 호환성 레벨을 지정하고, 그 레벨에 맞게 알맞게 프로듀서나 컨슈머를 먼저 업데이트 한다면 스키마 호환성은 잘 유지될 것.
- 사내 정책에 따라 FORWARD, BACKWRAD 처럼 약간은 타이트하게 스키마를 관리하는 방법도 있지만, BACKWARD와 FORWARD의 장점을 두루 지원하고 싶은 경우라면 호환성 레벨을 FULL로 사용할 수 있음.

정리

- 스키마 레지스트리를 처음 접하면 필요성에 대해 의구심을 품을 수 있음.
- 프로듀서와 컨슈머, 카프카만 이용하면 모든 데이터 처리를 할 수 있을것 같고, 굳이 스키마 레지스트리라는 별도의 앱을 추가해서 운영 부담을 가중시킬 필요가 있느냐는 생각도 할 수 있음.
- 하지만 도입부에서 설명한 스키마 레지스트리의 장점을 곰곰이 생각해보고, 앞으로 조직이나 회사가 추구할 확장성과 방향성, 메시지 변화에 따른 유연성을 고려한다면 스키마 레지스트리는 카프카에 반드시 필요한 애플리케이션일 것
- 스키마 정의 자체가 매우 번거롭고 귀찮은 일지만, 한 번 스키마를 깔끔하게 정의해놓는다면 이후 스키마 변경에 매우 유연하게 대처할 수 있음.
- 카프카 활용에 있어서 메시지를 잘 전송하고 제대로 처리하는 것도 중요하지만 스키마 레지스트리를 활용함으로써 데이터 관리적인 측면을 강화한다면 지금보다 더욱 효율적인 데이터 처리와 운영이 가능할 것.