12 장 테스팅, 모니터링 및 배치

이전 장들에서 카프카 스트림즈와 ksqlDB를 사용해 다양한 스트림 처리 애플리케이션을 구축하는 방법을 배웠다. 마지막으로 애플리케이션을 운영 환경에 놓기 위해 취해야 할 조치들을 배울 것이다. 여러분은 카프카 스트림즈와 ksqlDB 애플리케이션 제품화에 대한 장을 왜 통합하기로 결정했는지 의아하게 생각할 수도 있다. 일부 차이에도 불구하고 구체적으로 테스팅 관련하여 프로세스는 대부분 동일하며 소프트웨어 제품화 방법에 대한 정신 모델을 단순화하는 것은 장기적으로 유지보수성을 향상시킬 것이다 (특히 카프카 스트림즈와 ksqlDB 모두를 사용하는 하이브리드 환경에서).

이 장에서 답변할 질문은 다음을 포함한다:

- 카프카 스트림즈 애플리케이션과 ksqlDB 쿼리를 테스트할 수 있는 방법
- 카프카 스트림즈 토폴로지에 대해 벤치마크를 실행시킬 수 있는 방법
- 준비해야할 모니터링 유형
- 카프카 스트림즈와 ksqlDB에서 내장 JMX 지표에 액세스하기 위한 프로세스
- 카프카 스트림즈와 ksqlDB 애플리케이션 컨테이너화 및 배치 방법
- 직면할 가능성이 있는 운영상 태스크

스트림 처리 애플리케이션 테스트 방법을 배우면서 시작해보자.

테스팅

ksqlDB 쿼리 또는 카프카 스트림즈 애플리케이션의 초기 배치가 완료된다면 시간이 지남에 따라 코드에 대한 업데이트를 계속 전달할 것이다. 예를 들어 비즈니스 요건 변경으로 코드에 대한 업데이트가 필요할 수도 있고 수정되어야 할 버그 또는 성능 문제를 식별할 수도 있으며 단지 소프트웨어 버전을 업데이트할 수도 있다.

그러나 변경할 때마다 애플리케이션의 정확성 또는 성능에 영향을 미칠 수 있는 퇴화를 우연히 도입하지 않음을 보장하기 위해 조치를 취해야 한다. 이를 수행하는 가장 최상의 방법은 훌륭한 테스팅 사례를 확립하는 것이다. 이 절에서는 스트림 처리 애플리케이션에 대해 부드러운 진화 경로를 보장하는 데 도움을 줄 몇몇 테스팅 전략을 논의할 것이다.

ksqlDB 쿼리 테스팅

ksqlDB 테스팅은 매우 간단하다. ksqlDB 관리자(maintainer)가 3 개의 인수를 갖는 ksql-test-runner이라는 툴을 구축하였다:

- 하나 이상의 SQL 문을 포함하는 테스트할 파일
- 하나 이상의 소스 토픽에 대해 데이터 입력을 지정한 파일

- 하나 이상의 싱크 토픽에 대해 예상 출력을 지정한 파일

이의 동작원리를 확인하기 위해 예를 진행해보자. 우선 users 토픽에서 데이터를 읽고 각 로우를 인사로 변환하는 일련의 SQL 문을 생성할 것이다. 다음 코드 블록은 작업할 쿼리를 보여주는데, 이를 statements.sql 파일에 저장할 것이다:

```
CREATE STREAM users (
 ROWKEY INT KEY,
 USERNAME VARCHAR
) WITH (kafka_topic='users', value_format='JSON');
CREATE STREAM greetings
WITH (KAFKA_TOPIC = 'greetings') AS
SELECT ROWKEY, 'Hello, ' + USERNAME AS "greeting"
FROM users
EMIT CHANGES;
```

일단 ksqlDB 쿼리가 생성되었다면 테스트에 사용할 입력을 지정해야 한다. 이 예에서는 users라는 단일 소스 토픽을 갖고 있다. 따라서 테스트 진행 시 users 토픽에 2 개의 레코드를 삽입하라고 ksql-test-runner에 지시할 input.json 파일에 다음 컨텐츠를 저장할 것이다:

```
{
  "inputs": [
    {
      "topic": "users",
      "timestamp": 0,
      "value": {"USERNAME": "Isabelle"
      },
      "key": 0
    },
    {
      "topic": "users",
      "timestamp": 0,
      "value": {"USERNAME": "Elyse"
      },
      "key": 0
    }
  ]
}
```

마지막으로 ksqlDB 쿼리의 예상 출력을 지정해야 한다. 쿼리가 greetings 토픽에 쓰기 때문에 다음 라인을 output.json 파일에 저장하여 출력 토픽에 무언가 나타나야 한다고 주장할 것이다 (쿼리가 테스트 데이터를 처리한 후).

```
{
  "outputs": [
    {
      "topic": "greetings",
      "timestamp": 0,
      "value": {
        "greeting": "Hello, Isabelle"
      },
      "key": 0
    },
    {
      "topic": "greetings",
      "timestamp": 0,
      "value": {
        "greeting": "Hello, Elyse"
      },
      "key": 0
    }
  ]
}
```


3 개의 파일이 생성되었으며 이제 다음 명령을 사용하여 테스트를 진행할 수 있다.

```
docker run ₩
 -v "$(pwd)":/ksqldb/ ₩
 -w /ksqldb ₩
 -ti confluentinc/ksqldb-server:0.14.0 ₩
 ksql-test-runner -s statements.sql -i input.json -o output.json
```

이 책 작성 시점에 테스트 툴의 출력은 매우 장황했지만 출력 어딘가에서 다음 텍스트를 봐야 한다:

```
>>> Test passed!
```

나중에 실수로 파괴적인 변경을 도입한다면 테스트는 실패할 것이다. 예를 들어 greeting 칼럼이 생성되는 방식을 변경해보자. Hello로 사용자에게 인사하는 대신 Good morning로 인사할 것이다:

```
CREATE STREAM greetings
WITH (KAFKA_TOPIC = 'greetings') AS
SELECT ROWKEY, 'Good morning, ' + USERNAME AS "greeting"
FROM users
EMIT CHANGES;
```

테스트를 다시 진행한다면 실패할 것이다. 물론 output.json에 어서션(assertion) 변경없이 쿼리를 변경했기 때문에 이는 당연한 결과이다.

```
>>>>> Test failed: Topic 'greetings', message 0:
 Expected <0, {"greeting":"Hello, Isabelle"}> with timestamp=0
 but was <0, {greeting=Good morning, Isabelle}> with timestamp=0
```

이 예는 사소하지만 전체적인 아이디어는 중요하다. 이 방법으로 ksqlDB 쿼리를 테스트하는 것은 우

발적인 퇴화를 방지하고 제품에 코드 변경을 전달하기 전에 매우 권장된다.

> 맞춤형 ksqlDB 함수 (예, UDF, UDAF 및/또는 UDTF)를 활용하고 있다면 기반 Java 코드에 대해 단위 테스트 환경 설정을 하고 싶을 것이다. 프로세스는 예제 12-1에서 다룬 단위 테스팅 전략과 매우 비슷하지만 더욱 자세한 정보는 기사를 참조하기 바란다. "ksqlDB UDFs and UDAFs Made Easy" on the Confluent blog"

이제 약간 더 복잡하지만 (그러나 비교적 단순한) 카프카 스트림즈 애플리케이션 테스트 방법을 살펴보자.

카프카 스트림즈 테스팅

카프카 스트림즈 애플리케이션을 테스트할 때는 테스트 실행을 위해 자동화된 테스팅 프레임워크가 필요할 것이다. 작업할 프레임워크는 결국 여러분에 달려 있지만 이 절의 예에서는 테스트 수행에 Junit을 그리고 주장의 가독성 개선을 위해 AssertJ를 사용할 것이다.

테스팅 프레임워크 외에 공식 카프카 프로젝트의 일부로 유지보수 및 출시되는 kafka-streams-test-utils 라이브러리를 포함하길 원할 것이다. 이 라이브러리는 다음을 포함한다:

- 카프카 스트림즈 토폴로지 실행을 위해 시뮬레이션된 런타임 환경
- 테스트 카프카 토픽에서 데이터를 읽고 쓰기 위한 헬퍼 메소드
- 단위 테스팅 프로세서와 변환에 사용될 수 있는 모의 객체

카프카 스트림즈 프로젝트와 같이 제3자 패키지를 사용하는 것은 단순히 적절한 의존성을 통해 프로젝트의 빌드 파일 (build.gradle)을 업데이트하는 문제이다. 따라서 이전에 언급한 항목을 카프카 스트림즈 프로젝트로 가져가기 위해 다음과 같이 빌드 파일을 업데이트할 수 있다:

```
dependencies {
  testImplementation "org.apache.kafka:kafka-streams-test-utils:${kafkaVersion}"
  testImplementation 'org.assertj:assertj-core:3.15.0'
  testImplementation 'org.junit.jupiter:junit-jupiter:5.6.2'
}

test {
  useJUnitPlatform()
}
```

테스트 의존성을 프로젝트로 가져왔으며 이제 테스트를 작성할 준비를 마쳤다. 카프카 스트림즈 애플리케이션을 테스트할 여러 방법이 존재하기 때문에 이 절을 각각 다른 테스팅 전략에 중점을 두는 작은 하부 절로 나눌 것이다. 다르게 명시하지 않은 경우 프로젝트의 src/test/java[1] 디렉토리에 생성할 테

---

[1] 예를 들어 src/test/java/com/magicalpipelines/GreeterTopologyTest.java라는 파일에 토폴로지 테스트가 정의될 수 있다.

스트가 존재할 것이며 다음 명령을 사용하여 실행될 수 있다.

```
./gradlew test --info
```

단위 테스트

단위 테스팅은 코드에 대해 개별 조각을 테스트하는 것이다. 카프카 스트림즈 토폴로지를 구축할 때 테스트하길 가장 원하는 단위는 토폴로지를 구성하는 개별 프로세서이다. DSL 또는 프로세서 API 사용 여부에 따라 프로세서가 다른 방식으로 정의될 수 있기 때문에 어떤 API를 사용하는지에 따라 테스트가 다를 것이다. 우선 DSL의 스트림 프로세서 단위 테스팅을 살펴보자.

DSL. DSL을 사용할 때 일반적인 사례는 내장 카프카 스트림즈 연산자 중 하나에 람다식을 전달하는 것이다. 예를 들어 다음 토폴로지 정의에서 selectKey 연산자에 대한 로직은 람다 함수를 사용하여 인라인으로 정의된다:

```
public class MyTopology {
 public Topology build() {
Testing | 353
 StreamsBuilder builder = new StreamsBuilder();
 builder
 .stream("events", Consumed.with(Serdes.String(), Serdes.ByteArray()))
 .selectKey(
 (key, value) -> { ①
 // ... ②
 return newKey;
 })
 .to("events-repartitioned");
 return builder.build();
 }
}
```

① selectKey 로직은 람다식 내부에 정의된다.

② 간결함을 위해 로직은 생략했다.

람다식 내부의 코드가 간결하고 간단할 때 이는 잘 동작한다. 그러나 보다 크고 더욱 복잡한 코드 조각의 경우 로직을 전용 메소드로 분리하여 애플리케이션의 테스트 가능성을 개선할 수 있다. 예를 들어 selectKey 연산에 대해 정의하려고 하는 로직이 여러 라인에 걸쳐 있고 이 로직을 보다 큰 토폴로지로부터 분리하여 테스트한다고 가정해보자. 이 경우 다음 코드에서와 같이 람다식을 메소드 참조로 대체할 수 있다:

```
public class MyTopology {
 public Topology build() {
 StreamsBuilder builder = new StreamsBuilder();
```

```
 builder
 .stream("events", Consumed.with(Serdes.String(), Serdes.ByteArray()))
 .selectKey(MyTopology::decodeKey) ①
 .to("events-repartitioned");
 return builder.build();
 }
 public static String decodeKey(String key, byte[] payload) {
 // ... ②
 return newKey;
 }
}
```

① 람다가 메소드 참조로 대체되었다.

② 람다로 최초 정의되었던 로직이 전용 메소드로 이전되었다. 이는 코드의 테스트 가능성을 크게 개선한다. 다시 한번 간결함을 위해 실제 로직은 생략했다.

로직이 전용 메소드로 이전된 후 테스트하는 것은 더욱 쉽다. 사실 이 코드를 테스트하기 위해 kafka-streams-test-utils 패키지는 필요하지 않다. 왜냐하면 예제 12-1에서 보듯이 이 메소드를 단위 테스트하기 위해 테스팅 프레임워크를 사용할 수 있기 때문이다.

예제 12-1. selectKey 토폴로지 단계에 대한 간단한 단위 테스트. MyTopology.decodeKey라는 전용 메소드에 의존한다.

```
class MyTopologyTest {
 @Test
 public void testDecodeId() {
 String key = "1XRZTUW3";
 byte[] value = new byte[] {};
 String actualValue = MyTopology.decodeKey(key, value); ①
 String expectedValue = "decoded-1XRZTUW3"; ②
 assertThat(actualValue).isEqualTo(expectedValue); ③
 }
}
```

① 테스트는 selectKey 프로세서에서 사용한 동일한 메소드를 호출할 것이다. 여기서는 하드코딩된 키와 값을 전달한다. 다른 키-값 쌍을 사용하여 이 메소드를 여러 번 반복 실행하길 원한다면 파라미터화된 테스트라 불리는 Junit의 기능을 사용할 수도 있다.

② 메소드의 출력이 무엇인지에 대한 예상을 정의한다.

③ MyTopology.decodeKey 메소드가 반환하는 실제 값이 예상 값과 일치하는지를 주장하기 위해 AssertJ를 사용한다.

이런 식으로 처리 로직을 테스트함으로써 코드에서 퇴화를 식별하고 방지하는데 도움이 되는 매우 세부적이고 좁은 테스트 케이스를 가질 수 있다. 또한 카프카 스트림즈 애플리케이션은 종종 프로세서 모음 이상으로 구성된다. 다른 방식으로 토폴로지를 지원하는 다양한 헬퍼 메소드, 유틸리티 클래스 또는 맞춤형 Serdes 구현을 가질 수도 있다. 이 경우 이들 코드 단위도 여기서 기술한 동일 방법을 사용하여 테스트하는 것이 중요하다.

이런 형태의 테스팅은 DSL을 사용하는 애플리케이션에 대해 실제 잘 동작한다. 그러나 프로세서 API를 사용하고 있다면 테스트 구조와 실행 방법에 대해 추가적으로 고려해야 한다. 이를 다음 절에서 다룰 것이다.

프로세서 API. 프로세서 API의 경우 스트림 프로세서 로직을 정의할 때 일반적으로 람다 또는 메소드 참조로 작업하지 않는다. 대신 Processor 또는 Transformer 인터페이스를 구현하는 클래스로 작업한다. 카프카 스트림즈가 저수준 스트림 프로세서 초기화 시 이들에 전달하는 기반 ProcessorContext를 흉내내야 하기 때문에 테스팅 전략은 약간 다르다.

> 스트림 프로세서를 언급할 때 Processor 인터페이스를 언급하는 것은 아니다. 데이터 처리/변환 로직을 입력 스트림에 적용하는 역할을 하는 어떤 코드를 언급하기 위해 보다 넓은 의미에서 이 용어를 사용한다. 프로세서 API를 사용할 때 스트림 프로세서는 Processor 또는 Transformer 인터페이스 구현 중 하나일 수 있다. 이들 인터페이스 간 차이에 대해 복습이 필요하다면 "Processor과 Transformer"를 참조하기 바란다.

보다 저수준 프로세서 테스트 방법을 보여주기 위해 각각 고유 키에 대해 보았던 레코드 수를 추적하는 스테이트풀 변환자를 구현해보자. 다음 코드 블록은 테스팅할 트랜스포머 구현을 보여준다:

```
public class CountTransformer
 implements ValueTransformerWithKey<String, String, Long> {
 private KeyValueStore<String, Long> store;
 @Override
 public void init(ProcessorContext context) { ①
 this.store =
 (KeyValueStore<String, Long>) context.getStateStore("my-store"); ②
 }
 @Override
 public Long transform(String key, String value) { ③
 // process tombstones
 if (value == null) {
 store.delete(key);
 return null;
 }
 // get the previous count for this key,
 // or set to 0 if this is the first time
```

```
// we've seen this key
Long previousCount = store.get(key);
if (previousCount == null) {
previousCount = 0L;
}
// calculate the new count
Long newCount = previousCount + 1;
store.put(key, newCount);
return newCount;
}
@Override
public void close() {}
}
```

① 프로세서와 트랜스포머 구현은 ProcessorContext를 받아들이는 초기화 함수를 갖는다. 이 컨텍스트는 상태 저장소 검색과 주기적 함수 스케줄링과 같이 다양한 태스크에 사용된다.

② 이 트랜스포머는 보는 각각 고유 키에 대해 카운트를 기억해야 하기 때문에 스테이트풀이며 따라서 상태 저장소에 대한 참조를 저장할 것이다.

③ transform 메소드는 스트림 프로세서에 대한 로직을 갖고 있다. 이 코드 블록의 코멘트가 구현 세부사항을 논의한다.

트랜스포머가 준비되었고 단위 테스트를 작성할 수 있다. 이전 절에서 구현했던 간단한 메소드 수준 테스트와 달리 이 테스트는 kafka-streams-test-utils 패키지 내 헬퍼를 사용해야 한다. 다른 무엇보다 이 패키지를 통해 실제 토폴로지 실행없이 상태 액세스와 punctuation 스케줄링을 가능케 하는 MockProcessorContext 객체를 생성할 수 있다.

일반적인 방법은 단위 테스트 실행 전에 실행하는 셋업 함수 내에 MockProcessorContext를 생성하는 것이다. 프로세서가 스테이트풀이라면 상태 저장소도 초기화 및 등록해야 한다. 또한 토폴로지가 영구 저장소를 사용할 수 있지만 단위 테스트를 위해 인메모리 저장소 사용이 추천된다. 다음 코드 블록은 MockProcessorContext 설정 및 상태 저장소 등록 방법 예를 보여준다:

```
public class CountTransformerTest {
MockProcessorContext processorContext; ①
@BeforeEach ②
public void setup() {
Properties props = new Properties(); ③
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "test");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "dummy:1234");
processorContext = new MockProcessorContext(props); ④
KeyValueStore<String, Long> store = ⑤
```

```
Stores.keyValueStoreBuilder(
Stores.inMemoryKeyValueStore("my-store"),
Serdes.String(), Serdes.Long())
.withLoggingDisabled()
.build();
store.init(processorContext, store); ⑥
processorContext.register(store, null);
}
}
```

① 테스트에서 MockProcessorContext 객체를 추후 참조할 수 있도록 인스턴스 변수로 저장할 것이다.

② setup 메소드는 Junit의 BeforeEach 주석으로 주석처리되며, 이는 이 코드가 각 테스트 전에 실행되도록 한다.

③ MockProcessrContext 생성을 위해 카프카 스트림즈 특성을 제공해야 한다. 여기서는 2 개의 필수 특성만을 제공한다.

④ MockProcessorContext 인스턴스를 인스턴스화한다.

⑤ 트랜스포머가 스테이트풀로 이를 사용하기 위해 인메모리 상태 저장소를 생성한다.

⑥ 테스트에 사용하기 위해 준비한 상태 저장소를 초기화 및 등록한다.

MockProcessorContext가 생성되었다면 이 컨텍스트 객체를 사용하여 Processor 또는 Transfromer 클래스를 인스턴스화 및 초기화할 수 있고 그 후 단위 테스트를 수행할 수 있다. 다음 코드 블록은 이방법을 사용하여 CountTransformer를 테스트하는 방법을 보여준다. 테스트는 기본 카운팅 동작과 툼스톤-처리 동작을 다룬다:

```
public class CountTransformerTest {
MockProcessorContext processorContext;
@BeforeEach
public void setup() {
// see the previous section
}
@Test ①
public void testTransformer() {
String key = "123";
String value = "some value";
CountTransformer transformer = new CountTransformer(); ②
transformer.init(processorContext); ③
assertThat(transformer.transform(key, value)).isEqualTo(1L); ④
assertThat(transformer.transform(key, value)).isEqualTo(2L);
```

```
 assertThat(transformer.transform(key, value)).isEqualTo(3L);
 assertThat(transformer.transform(key, null)).isNull(); ⑤
 assertThat(transformer.transform(key, value)).isEqualTo(1L);
 }
}
```

① Junit Test 주석은 이 메소드가 테스트를 실행함을 알려준다.

② CountTransformer를 인스턴스화한다.

③ setup 메소드에서 초기화했던 MockProcessorContext를 통해 CountTransformer 인스턴스를 인스턴스화한다.

④ 트랜스포머 동작을 테스트한다. CountTransformer가 각 키를 본 횟수를 세기 때문에 카운트가 예상한데로 증분되는지 보장하기 위해 일련의 테스트를 수행한다.

⑤ 트랜스포머는 툼스톤 (키가 있지만 값이 null인 레코드) 처리 로직을 갖고 있다. 카프카 스트림즈에서 툼스톤은 레코드가 상태 저장소에서 삭제되어야 함을 알려준다. 이 라인과 다음 라인은 툼스톤이 정확하게 처리되었음을 보장한다.

MockPorcessorContext가 가능케하는 다른 테스팅 능력도 있다. 예를 들어 레코드를 다운스트림 프로세서로 보내기 위해 punctuator 함수에 의존하는 프로세서도 테스트할 수 있다. 저수준 프로세서의 단위 테스트 방법에 대해 보다 많은 예는 공식 문서를 참조하기 바란다.

동작 테스트

단위 테스팅이 도움이 되지만 카프카 스트림즈 토폴로지는 보통 함께 동작하는 프로세서 모음으로 구성된다. 그렇다면 전체 토폴로지의 동작을 테스트할 수 있는 방법은? 다시한번 답은 kafka-streams-test-utils 패키지에 의해 제공된다. 이 시점에서 이 라이브러리에 포함된 시뮬레이션된 런타임 환경을 활용할 것이다.

데모 목적으로 테스트에 사용할 수 있는 매우 간단한 토폴로지를 생성해보자.

다음 코드 블록은 users 토픽에서 사용자 이름을 읽고 greetings 토픽에 각 사용자에 대한 인사를 생산하는 간단한 카프카 스트림즈 토폴로지를 보여준다. 그러나 한 가지 예외가 있다: 사용자 이름이 Randy인 경우 인사를 생성하지 않는다. 이 필터링 조건은 보다 흥미로운 테스트 시나리오를 제공할 것이다 (Randy라는 누군가를 공격할 의도는 없다):

```
class GreeterTopology {
 public static String generateGreeting(String user) {
 return String.format("Hello %s", user);
 }
 public static Topology build() {
 StreamsBuilder builder = new StreamsBuilder();
```

```
 builder
 .stream("users", Consumed.with(Serdes.Void(), Serdes.String()))
 .filterNot((key, value) -> value.toLowerCase().equals("randy"))
 .mapValues(GreeterTopology::generateGreeting)
 .to("greetings", Produced.with(Serdes.Void(), Serdes.String()));
 return builder.build();
 }
}
```

다음에 topology test driver라는 카프카 스트림즈 라이브러리에 포함된 테스트 드라이버를 사용하는 단위 테스트를 작성해보자. 이 테스트 드리어버를 통해 카프카 스트림즈 토폴로지에 데이터를 넣고 (이 경우 GreeterTopology) 출력 토픽에 추가되는 해당 데이터를 분석할 수 있다. 다음과 같이 테스트 드라이버 인스턴스 생성 및 파괴를 위한 setup과 teardown 메소드 정의로 시작할 것이다.

```
class GreeterTopologyTest {
 private TopologyTestDriver testDriver;
 private TestInputTopic<Void, String> inputTopic;
 private TestOutputTopic<Void, String> outputTopic;
 @BeforeEach
 void setup() {
 Topology topology = GreeterTopology.build(); ①
 Properties props = new Properties(); ②
 props.put(StreamsConfig.APPLICATION_ID_CONFIG, "test");
 props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "dummy:1234"); ③
 testDriver = new TopologyTestDriver(topology, props); ④
 inputTopic =
 testDriver.createInputTopic( ⑤
 "users",
 Serdes.Void().serializer(),
 Serdes.String().serializer());
 outputTopic =
 testDriver.createOutputTopic( ⑥
 "greetings",
 Serdes.Void().deserializer(),
 Serdes.String().deserializer());
 }
 @AfterEach
 void teardown() {
 testDriver.close(); ⑦
 }
}
```

① 카프카 스트림즈 토폴로지를 구축한다.

② 2 개의 필수 구성 파라미터를 제공하여 카프카 스트림즈를 구성한다.

③ 토롤로지가 시뮬레이션된 런타임 환경에서 실행될 것이기 때문에 부트스트랩 서버를 확인할 필요는 없다.

④ 토폴로지 테스트 드라이버를 생성한다. 테스트 드라이버는 토폴로지 테스팅을 매우 쉽게 하는 몇몇 헬퍼 메소드를 갖고 있다.

⑤ 테스트 드라이버는 입력 토픽 생성을 위한 헬퍼 메소드를 갖고 있다. 여기서 users 토픽을 생성한다.

⑥ 테스트 드라이버는 또한 출력 토픽 생성을 위한 헬퍼 메소드를 갖고 있다. 여기서 greetings 토픽을 생성한다.

⑦ TopologyTestDriver.close() 메소드를 호출하여 각 테스트 후 리소스가 적절히 정리되었는지 확인한다.

```
class GreeterTopologyTest {
 // ...
 @Test
 void testUsersGreeted() {
 String value = "Izzy"; ①
 inputTopic.pipeInput(value); ②
 assertThat(outputTopic.isEmpty()).isFalse(); ③
 List<TestRecord<Void, String>> outRecords =
 outputTopic.readRecordsToList(); ④
 assertThat(outRecords).hasSize(1); ⑤
 String greeting = outRecords.get(0).getValue(); ⑥
 assertThat(greeting).isEqualTo("Hello Izzy");
 }
}
```

① 테스트 케이스가 입력 토픽으로 넣기 위한 레코드를 생성할 것이다.

② 테스트 레코드를 입력 토픽으로 넣는다. 키와 값을 받아들이는 pipeInput 메소드의 오버로드된 버전도 존재한다.

③ 출력 토픽이 최소한 하나의 레코드를 포함함을 주장하기 위해 isEmpty() 메소드를 사용한다.

④ 출력 토픽의 레코드를 읽는 다양한 방법이 존재한다. 여기서는 readRecoresToList() 메소드를 사용하여 출력 레코드 모두를 목록으로 읽는다. 확인할 수 있는 다른 메소드로는 readValue(), readKeyValue(), readRecord()와 readKeyValuesToMap()이 있다.

⑤ 단지 하나의 출력 레코드만 있음을 어서션한다 (flatMap 연산자를 사용하는 토폴로지는 입력 대 출력 레코드에 대해 1:N 비율을 가질 수 있다).

⑥ 출력 레코드 값을 읽는다. getKey(), getRecordTime()와 getHeaders()를 포함하여 더 많은 레코드 데이터에 액세스하기 위한 추가적인 메소드가 존재한다.

스트림으로 작업할 때는 TestOutputTopic.readRecordsToList() 메소드가 유용하다. 왜냐하면 이 메소드가 전체 출력 이벤트 시퀀스를 포함하기 때문이다. 반면 TestOutputTopic.readKeyValuesToMap()은 스트림과 달리 테이블로 작업할 때 유용하다. 왜냐하면 단지 각 키에 대해 최신 표현만을 포함하기 때문이다. 사실 스트림과 테이블 간 (둘 모두 인서트 시맨틱을 사용) 그리고 테이블과 맵 (둘 모두 업데이트 시맨틱을 사용)간 밀접한 관계를 이미 논의했는데 이들 관계는 이들 메소드로 모델링된다.

벤치마킹

ksqlDB가 기반 카프카 스트림즈 토폴로지를 처리하는 반면 카프카 스트림즈 DSL 또는 프로세서 API 사용 시 오는 자유는 잠재적인 성능 퇴화에 대해 더욱 많은 요인을 도입한다. 예를 들어 실수로 느린 계산 단계를 도입했거나 덜 효율적으로 만드는 방식으로 토폴로지를 재작업한 경우 변경을 제품에 전달하기 전에 퇴화를 잡아 낸다면 문제해결 하는 것이 덜 고통스러울 것이다.

성능 퇴화에 대해 보호하기 위해 카프카 스트림즈 애플리케이션을 변경할 때마다 코드에 대해 벤치마크를 실행해야 한다. 운좋게도 이를 위해 kafka-streams-test-utils 패키지가 제공하는 시뮬레이션된 런타임 환경과 JMH와 같은 벤치마킹 프레임워크를 결합할 수 있다.

해야 할 첫번째 일은 build.gradle 파일에 me.champeau.gradle.jmh 플러그인을 추가하고 이 플러그인이 생성하는 jmh 태스크를 구성하는 것이다. 다음 코드 블록은 이 둘을 수행하는 방법 예를 보여준다.

```
plugins {
 id 'me.champeau.gradle.jmh' version '0.5.2'
}
jmh { ①
 iterations = 4
 benchmarkMode = ['thrpt']
 threads = 1
 fork = 1
 timeOnIteration = '3s'
 resultFormat = 'TEXT'
 profilers = []
 warmupIterations = 3
 warmup = '1s'
}
```

① jmh 플러그인 여러 구성 파라미터를 지원한다. 이들 파라미터에 대한 전체 목록과 설명은 플러그인 문서를 참조하기 바란다.

이제 벤치마크 실행을 위한 클래스를 생성해보자. 지금까지 개발했던 다른 카프카 스트림즈 테스트 코드와 달리 벤치마킹 코드는 src/jmh/java 디렉토리에 있을 것으로 예상된다. 다음은 벤치마킹 클래스이다:

```
public class TopologyBench {
 @State(org.openjdk.jmh.annotations.Scope.Thread)
 public static class MyState {
 public TestInputTopic<Void, String> inputTopic;
 @Setup(Level.Trial) ①
 public void setupState() {
 Properties props = new Properties();
 props.put(StreamsConfig.APPLICATION_ID_CONFIG, "test");
 props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "dummy:1234");
 props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 0);
 // build the topology
 Topology topology = GreeterTopology.build();
 // create a test driver. we will use this to pipe data to our topology
 TopologyTestDriver testDriver = new TopologyTestDriver(topology, props);
 testDriver = new TopologyTestDriver(topology, props);
 // create the test input topic
 inputTopic =
 testDriver.createInputTopic(
 "users", Serdes.Void().serializer(), Serdes.String().serializer());
 }
 }
 @Benchmark ②
 @BenchmarkMode(Mode.Throughput) ③
 @OutputTimeUnit(TimeUnit.SECONDS)
 public void benchmarkTopology(MyState state) {
 state.inputTopic.pipeInput("Izzy"); ④
 }
}
```

① Setup 주석은 벤치마크 전에 이 메소드가 실행되어야 함을 알리기 위해 사용된다. 이 메소드를 사용해서 TopologyTestDriver 인스턴스와 입력 토픽을 설정한다. Level.Trial 인수는 벤치마크 각 실행 전에 이 메소드가 실행되어야 함을 JMH에 알려준다 (비고: 각 반복 전에 setup이 실행될 것임을 의미하지 않는다).

② Benchmark 주석은 벤치마크 메소드임을 JMH에 알려준다.

③ 여기서 벤치마크 모드는 Mode.Throught로 설정된다. 이는 벤치마킹 메소드가 실행될 수 있는 토당 횟수를 측정할 것이다.

④ 테스트 레코드를 입력 토픽에 넣는다.

벤치마킹 코드가 생성되었기 때문에 다음 명령을 사용하여 Gradle을 통해 벤치마크를 실행할 수 있다:

```
$ ./gradlew jmh
```

다음과 비슷한 출력을 봐야 한다:

```
Benchmark                          Mode  Cnt       Score       Error  Units
TopologyBench.benchmarkTopology   thrpt    4  264794.572 ± 39462.097  ops/s
```

애플리케이션이 초당 264K 보다 많은 메시지를 처리할 수 있는 것이 의미가 있는가? 그렇지는 않다. 이전 절의 토폴로지 테스트와 같이 벤치마크가 시뮬레이션된 런타임 환경에서 실행됨을 기억하기 바란다. 또한 운영 단계 애플리케이션이 할 것 같은 외부 카프카 클러스터에 대한 네크워크 호출을 하지 않고 있으며 벤치마크는 종종 운영 워크로드가 실행되는 것과는 다른 하드웨어 (예, 연속적인 통합 서버)에서 실행된다. 따라서 이 숫자는 토폴로지에 대한 성능 기준을 확립하고 향후 벤치마킹 결과를 비교하기 위해서만 사용되어야 한다.

카프카 클러스터 벤치마킹

카프라 스트림 또는 ksqlDB로 작업하든 상관없이 카프카 클러스터 수준에서 성능 테스트를 실행하고 싶을 수 있다. 카프카는 이를 도울 수 있는 2 개의 컨솔 스크립트를 포함하는데, 이를 통해 읽기/쓰기 처리량을 측정하고, 클러스터에 대한 부하 및 스트레스 테스트를 실행하며, 특정 클라이언트 설정 (배치 크기, 버퍼 메모리, 프로듀서 acks, 컨슈머 쓰레드 카운트)과 입력 특성 (레코드 크기, 메시지 양)이 클러스터 성능에 미치는 영향을 결정할 수 있다.

토픽에 데이터를 생산하는 성능을 분석하고 싶다면 kafka-producer-perf-test 명령을 사용할 수 있다. 많은 사용가능한 옵션이 있는데 다음 예는 이들 중 일부를 보여준다:

```
kafka-producer-perf-test ₩
 --topic users ₩
 --num-records 1000000 ₩
 --record-size 100 ₩
 --throughput -1 ₩
 --producer-props acks=1 ₩
 bootstrap.servers=kafka:9092 ₩
 buffer.memory=67108864 ₩
 batch.size=8196
```

이 명령에 대해 다른 유용한 버전은 페이로드 파일을 지정하는 것이다. 이 파일의 레코드는 지정된 카프카 토픽에 생산될 수 있다. 다음 코드 블록은 input.json 파일에 3 개의 레코드를 쓸 것이다.

```
cat <<EOF >./input.json
{"username": "Mitch", "user_id": 1}
{"username": "Isabelle", "user_id": 2}
```

```
{"username": "Elyse", "user_id": 3}
EOF
```

이제 하드코딩된 입력 레코드를 사용하여 성능 테스트를 수행하기 위해 –record-size 플래그를 –payload-file input.json 인수로 대체할 수 있다:

```
kafka-producer-perf-test ₩
 --topic users ₩
 --num-records 1000000 ₩
 --payload-file input.json ₩
 --throughput -1 ₩
 --producer-props acks=1 ₩
 bootstrap.servers=kafka:9092 ₩
 buffer.memory=67108864 ₩
 batch.size=8196
```

성능 리포트 예는 다음과 같다:

```
1000000 records sent, 22166.559528 records/sec (0.76 MB/sec),
58.45 ms avg latency, 465.00 ms max latency,
65 ms 50th, 165 ms 95th, 285 ms 99th, 380 ms 99.9th.
```

컨슈머 성능 테스트 컨솔 스크립트도 존재한다. 다음 코드는 호출 예를 보여준다:

```
kafka-consumer-perf-test ₩
 --bootstrap-server kafka:9092 ₩
 --messages 100000 ₩
 --topic users ₩
 --threads 1
```

```
# example output (modified to fit)
start.time                end.time                  data.consumed.in.MB
2020-09-17 01:23:41:932   2020-09-17 01:23:42:817   9.5747

MB.sec    data.consumed.in.nMsg   nMsg.sec
10.8189   100398
```

테스팅에 대한 최종 생각

코드를 변경할 때마다 테스트를 실행하기 위한 자동화된 워크플로우 설정을 고려해보자. 워크플로우 예는 다음과 같다"

- 모든 코드 변경이 별도 버전 제어 시스템 (예, GitHub, Butbucket)에 푸시된다.
- 개발자가 코드 변경을 병합하고자 할 때 풀 요청을 열어야 한다.
- 풀 요청이 열릴 때 Jenkins, Travis, Github Actions 등과 같은 시스템을 사용하여 테스트가 자동으로 실행된다.

- 하나 이상의 테스트가 실패하는 경우 병합이 차단되어야 한다. 그렇지 않은 경우 검사가 통과한다면 코드를 검토할 준비를 마친 것이다.

이와 같은 워크플로우가 많은 독자에 대해 두번째 본질일 수 있다. 그러나 자동화된 테스트는 코드 퇴화를 예방하는데 있어 중요한 역할을 하며 따라서 정기적으로 테스팅 사례와 워크플로우를 검토하는 것은 가치가 있다.

ksqlDB와 카프카 스트림즈 애플리케이션 테스트 방법을 배웠기 때문에 제품 단계로 소프트웨어를 전달하기 위한 다른 중요한 선결조건을 살펴보자: 모니터링.

모니터링

모니터링은 거대한 주제로 종종 많은 다른 기술을 포함한다. 따라서 이 절에서는 모든 방법을 세부적으로 다루지 않을 것이며 여러분이 기억하길 원할 수 있는 다양한 모니터링 유형에 대한 체크리스트와 사용하기 원할 수 있는 일부 기술 예를 제공할 것이다.

그러나 기술적인 세부사항을 다룰 카프카 스트림즈와 ksqlDB 모니터링 퍼즐에 대한 한 부분이 있다. 두 기술 모두 일련의 내장 JMX 지표[2]를 포함하며 이들 지표를 추출할 수 있다는 것은 애플리케이션과 쿼리의 관찰가능성을 크게 향상시킬 것이다. 뒤의 기술적인 세부사항에서 지표 추출을 다룰 것이다. 우선 모니터링 체크리스트를 살펴보자.

모니터링 체크리스트

다음 표는 제품 단계 카프카 스트림즈와 ksqlDB 애플리케이션에 대해 기억하길 원할 수 있는 모니터링 전략을 보여준다.

모니터링 전략	모니터링 대상	기술 예
클러스터 모니터링	• 복제 부족 파티션 • 컨슈머 지연 • 오프셋 전진 • 토픽 처리량	Kafka_exporter[a]
로그 모니터링	• 총 로그 비율 • 에러 로그 비율	ELK, ElastAlert, Cloud Loggging
지표 모니터링	• 소비 속도 • 생산 속도 • 처리 지연 • 폴 타임	Prometheus

[2] JMX는 Java Management Extension을 의미하며 리소스 모니터링 및 관리에 사용된다. 카프카 스트림즈 라이브러리는 애플리케이션이 어떻게 수행되고 있는지에 대한 통찰력을 제공하는 일련의 JMX 지표를 등록 및 업데이트한다. ksqlDB는 카프카 스트림즈 기반으로 구축되었기 때문에 이들 JMX 지표를 무료로 얻는다.

맞춤형 계측[b]	• 비즈니스 지표	OpenCensus
프로파일링	• 데드락 • 핫스팟	YourKit
시각화	• 위 모두	Grafana
경고	• SLO (service-level objectives)	Alertmanager

[a] https://github.com/danielqsj/kafka_exporter 참조

[b]주로 카프카 스트림즈에 대해 사용

JMX 지표 추출

카프카 스트림즈와 ksqlDB 모두 JMX를 통해 지표를 노출한다. 카프카 스트림즈 애플리케이션 또는 ksqlDB 서버를 실행시킬 때 JConsole을 사용하여 이들 지표에 액세스할 수 있으며 이는 대부분의 튜토리얼에서 언급하는 지침이다. 카프카 스트림즈 및/또는 ksqlDB와 동일 머신에서 JConsole를 실행시킨다면 JMX 지표를 검사하기 위해 어떤 인수도 없이 단순히 jconsole 명령을 실행할 수 있다.

예를 들어 다음 코드는 원격 JMX 모니터링 활성화를 위해 다양한 시스템 특성을 설정하고 JMX가 활성화된 ksqlDB를 실행시키는 방법을 보여준다.

```
docker-compose up ①

MY_IP=$(ipconfig getifaddr en0); ②

docker run ₩ ③
 --net=chapter-12_default ₩ ④
 -p 1099:1099 ₩
 -v "$(pwd)/ksqldb":/ksqldb ₩
 -e KSQL_JMX_OPTS="₩ ⑤
-Dcom.sun.management.jmxremote ₩
 -Djava.rmi.server.hostname=$MY_IP ₩
 -Dcom.sun.management.jmxremote.port=1099 ₩
 -Dcom.sun.management.jmxremote.rmi.port=1099 ₩
 -Dcom.sun.management.jmxremote.authenticate=false ₩
 -Dcom.sun.management.jmxremote.ssl=false" ₩
 -ti confluentinc/ksqldb-server:0.14.0 ₩
 ksql-server-start /ksqldb/config/server.properties ⑥
```

① 소스 코드는 카프카 클러스터 실행을 위한 도커 컴포즈 설정을 포함하고 있다. 맨 밑의 2 줄로 실행시킬 ksqlDB 서버 인스턴스는 이 카프카 클러스터와 통신할 것이다.

② IP를 환경 변수로 저장한다. 이 IP를 통해 JMX 지표를 노출할 수 있도록 뒷 부분의 JMX 특성에서 이를 참조할 것이다.

③ JMX가 활성화된 ksqlDB 서버 인스턴스를 실행한다.

④ 도커 컴포즈 외부에서 수동으로 ksqlDB 서버를 실행시키기 때문에 --et 플래그를 사용하여 도커 네트워크에 연결할 수 있다.

⑤JMX 지표 액세스에 필요한 시스템 특성을 설정한다.

⑥ ksqlDB 서버 인스턴스를 시작한다.

다음 명령을 사용하여 JConsole을 열 수 있다:

```
jconsole $MY_IP:1099
```

MBeans 탭을 클릭하면 카프카 스트림즈 또는 ksqlDB 애플리케이션이 노출하는 모든 사용가능한 지표를 볼 수 있다. 예를 들어 그림 12-1은 ksqlDB 서버에 대한 지표 목록을 보여준다. ksqlDB가 카프카 스트림즈에 기반하여 구축되고 카프카 스트림즈가 저수준 카프카 컨슈머와 카프카 프로듀서 클라이언트에 기반하여 구축되기 때문에 기반 라이브러리 각각이 보고하는 지표를 볼 것이다.

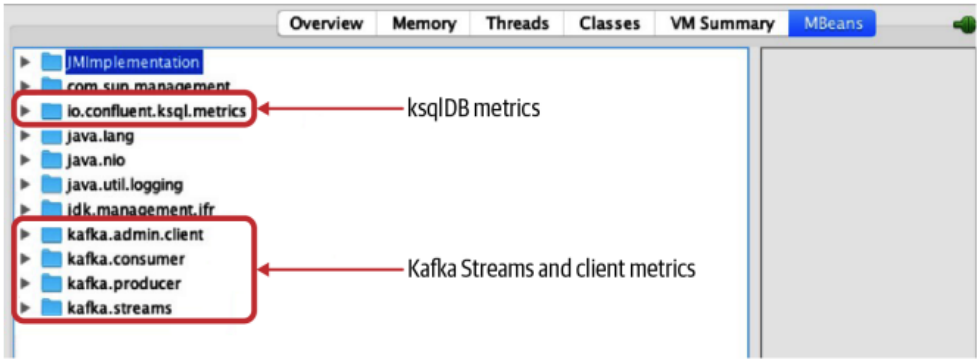

그림 12-1. JConsole에서 카프카 스트림즈 관련 지표 그룹

그림 12-2와 같이 각 지표에 대해 메타 데이터와 값을 보기 위해 각 그룹으로 드릴다운할 수 있다.

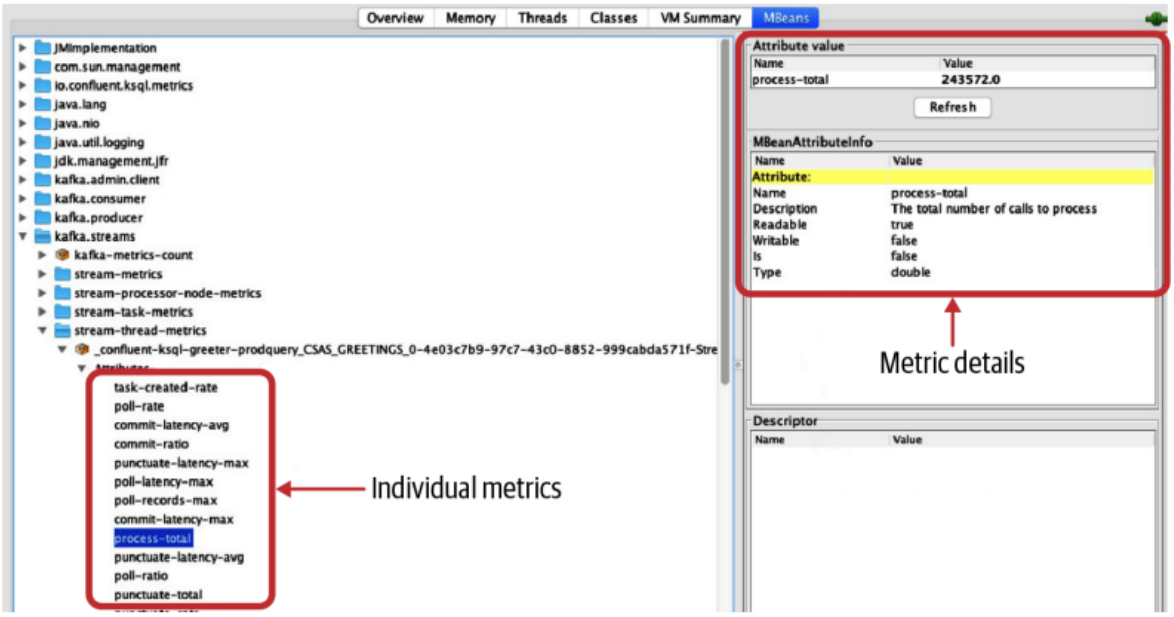

그림 12-2. 카프카 스트림즈 지표 일부의 보다 세부적인 뷰

JConsole는 테스팅 또는 개발 환경에서 즉석에서 지표를 검사하는데 유용하지만 소프트웨어를 운영 단계로 배치할 시간이 될 때는 이력 데이터셋을 저장하고 데이터에 대해 쿼리를 수행하며 경고 시스템과 통합할 수 있는 보다 강건한 솔루션을 원할 것이다.

이 책 작성 시점에 카프카 스트림즈와 ksqlDB JMX 지표로 작업하기 위해 추천하는 기술은 Prometheus이다. Prometheus 내로 지표를 얻기 위해서는 HTTP 엔드포인트를 통해 지표를 노출할 exporter이 필요하다. 카프카 스트림즈와 ksqlDB 인스턴스 모두에 대해 exporter을 설정하는 것은 비슷하다. 다음이 필요할 것이다:

- Prometheus JMX exporter JAR 다운로드
- 다음 플래그를 사용하여 카프카 스트림즈 또는 ksqlDB 시작
- -javaagent:./jmx_prometheus_javaagent-0.14.0.jar=8080:config.yaml
- 카프카 스트림즈 또는 ksqlDB 애플리케이션이 실행 중인 IP 주소와 호스트명과 JMX 지표를 노출하도록 구성된 포트 번호로 구성된 HTTP 엔드포인트를 스크랩하도록 Prometheus 구성

이 장의 소스 코드에는 카프카 스트림즈와 ksqlDB와 함께 Prometheus를 사용하는 예가 포함되어 있다. 이는 구현 세부사항이기 때문에 책에서는 Prometheus의 기술적 세부사항을 생략했지만 요점은 Prometheus 또는 다른 모니터링 시스템 사용 여부와 상관없이 애플리케이션의 관찰가능성을 향상시키기 위해 카프카 스트림즈와 ksqlDB가 제공하는 내장 JMX 지표를 외부 시스템에 내보내야 한다는 것이다.

배치

카프카 스트림즈와 ksqlDB는 베어 메탈 머신 상에서, VM 내부에서 또는 컨테이너를 사용하여 실행할 수 있다. 그러나 도커를 사용하여 애플리케이션을 컨테이너화하는 것이 추천된다. 이 방법의 장점 중 일부는 다음과 같다:

컨테이너는 코드가 실행될 경량의 격리된 런타임 환경을 제공한다.

사설 및 공공 컨테이너 레지스트리 모두를 사용하여 복수의 컨테이너 버전을 저장, 관리, 공유 및 유지하는 것은 매우 쉽다[3].

컨테이너를 통해 기반 인프라와 코드를 분리할 수 있다. 이는 동일 컨테이너를 온프레미스, 함께 배치한 서버, 다수의 클라우드 플랫폼 또는 로컬 머신에서도 실행시킬 수 있음을 의미한다.

컨테이너를 사용하여 두 기술 모두로 작업하는 방법을 살펴보자.

---

[3] Docker Hub, artifactory 및 Goolge Container Registry가 일부 유명한 컨테이너 레지스트리이다.

ksqlDB 컨테이너

컨플루언트는 ksqlDB 서버와 CLI 모두에 대해 공식 컨테이너 이미지를 이미 공개했다. 여러분이 쿼리를 운영 단계에 배치시킬 준비가 되었을 때 공식 ksqlDB 서버 컨테이너 이미지 (confluentinc/ksqldb-server) 또는 런타임 환경에 조정이 필요한 경우라면 파생 이미지를 사용할 수 있다[4].

공식 ksqlDB 서버를 통해 환경 변수를 사용하여 ksqlDB 구성 특성 중 어떤 것이라도 사실상 지정할 수 있다. 그러나 이는 몇 가지 단점을 갖고 있다:

- 구성이 버전 제어되어야 한다. 환경 변수를 사용하여 모든 ksqlDB 특성을 설정하는 것은 이를 어렵게 한다.
- 구성이 많은 애플리케이션의 경우 컨테이너 이미지를 실행시키는 명령이 크기가 커지고 다루기 어려울 수 있다.
- 환경 변수 설정이 컨테이너 내부에 실제 구성 파일을 마운트하기 위한 쉬운 방법이지만 ksqlDB를 헤드리스 모드 또는 특정 데이터 통합 기능을 사용하여 실행시킨다면 컨테이너 내부에 파일을 마운트해야 한다.

따라서 운영 단계 ksqlDB 서버 인스턴스를 구성하기 위해 환경 변수에 전적으로 의존하는 대신 컨테이너 내부에 버전 제어된 구성 파일을 마운팅할 것을 추천한다. 예를 들어 ksqlDB 서버 구성을 로컬 머신의 config/server.properties 파일에 저장해보자. 데모 목적으로 이 파일의 내용을 간단하게 유지할 것이지만 구성 파일에 사용가능한 ksqlDB 특성을 포함할 수 있다 (부록 B 참조).

```
bootstrap.servers=kafka:9092
ksql.service.id=greeter-prod
```

우리는 공식 컨데이너 이미지 내부에 이 구성을 마운트하여 ksql-server-start 명령을 호출할 수 있다.

```
docker run ₩
 --net=chapter-12_default ₩
 -v "$(pwd)/config":/config ₩
 -ti confluentinc/ksqldb-server:0.14.0 ₩
 ksql-server-start /config/server.properties
```

① 이 플래그는 튜토리얼을 위한 것으로 이를 통해 도커 컴포즈 환경에서 실행 중인 카프카 클러스터에 연결할 수 있다.

② 구성을 호스트 시스템에서 실행 중인 컨테이너로 마운트한다.

③ 마운트된 구성 파일을 사용하여 ksqlDB 서버 인스턴스를 시작한다.

---

[4] 파생 이미지는 도커 파일 앞 부분에 FROM 지침을 사용하여 공식 ksqlDB 이미지를 기본 이미지로 지정한 것이다. 환경에 대한 맞춤화가 도커 파일 내에 지정될 것이다.

다른 파일 경로를 참조하는 2 개의 ksqlDB 구성을 있음을 상기하기 바란다:

Ksql.connect.worker.config

카프카 커넥트가 내장 모드로 실행 중인 경우 카프카 커넥트 워커 구성 파일의 경로를 지정하는 선택적인 구성 (그림 9-3 참조)

Queries.file

ksqlDB가 헤드리스 모드로 실행 중인 경우 실행할 쿼리 파일의 경로를 지정한다.

이들 특성 중 하나가 구성에 포함된 경우 컨테이너 내부에 이들 구성이 참조하는 파일을 마운트해야 한다. 이제 컨테이너 내부에서 카프카 스트림즈 애플리케이션을 실행시키는 방법을 살펴보자.

카프카 스트림즈 컨테이너

컨테이너 내부에서 카프카 스트림즈를 실행시키는 것은 약간 더 복잡하지만 아직까지는 매우 간단하다. 저수준 기본 이미지를 사용하여 도커 이미지를 쉽게 생성할 수 있다[5]. 그러나 보다 간단한 옵션은 구글이 개발하고 Java 애플리케이션 컨테이너화와 배포 방법을 단순화한 인기있는 이미지 빌더인 Jib를 사용하는 것이다.

Jib를 사용하기 위해서는 빌드 파일에 Jib 의존성을 추가해야 한다. 이 책에서는 빌드 시스템으로 Gradle을 사용했기 때문에 다음을 build.gradle 파일에 추가할 것이다:

```
plugins {
 id 'com.google.cloud.tools.jib' version '2.1.0'
}
jib {
 to {
 image = 'magicalpipelines/myapp:0.1.0'
 }
 container {
 jvmFlags = []
 mainClass = application.mainClassName
 format = 'OCI'
 }
}
```

Jib가 자동으로 이미지를 컨테이너 레지스트리에 푸시하길 원하는지 여부와 도커 이미지가 푸시되는 컨테이너 레지스트리에 따라 이미지에 대해 다양한 다른 네이밍 스킴을 사용할 수 있다. 공식 Jib 문서가 사용가능한 컨테이너 레지스트리와 네이밍 스킴에 대한 최상의 리소스지만 이 책 작성 시점에 다음의 컨테이너 레지스트리도 지원되었다:

---

[5] 적절한 기본 이미지 예는 openjdk:8-jdk-slim이다.

- Docker Hub
- Google Container Registry (GCR)
- Amazon Elastic Container Registry (ECR)
- Azure Container Registry (ACR)

빌드 파일이 업데이트되었다면 다음 태스크 중 하나를 실행할 수 있다.

./gradlew jib

이 태스크는 컨테이너 이미지를 빌드하여 해당 컨테이너 이미지를 컨테이너 레지스트리에 푸시할 것이다. 이를 위해 원하는 컨테이너 레지스트리에 대해 인증을 받아야 한다.

./gradlew jibDockerBuild

이 태스크는 로컬 도커 데몬을 사용하여 컨테이너 이미지를 빌드하지만 이미지를 컨테이너 레지스트리에 푸시하지는 않을 것이다. 이는 이미지를 빌드할 수 있는지 테스트하는 가장 쉬운 방법이다.

데모 목적으로 다음 명령을 사용하여 로컬에서 도커 이미지를 빌드할 것이다.

```
./gradlew jibDockerBuild
```

다음과 비슷한 출력을 봐야 한다:

```
Built image to Docker daemon as magicalpipelines/myapp:0.1.0
```

출력에서 참조된 도커 이미지 태그는 카프카 스트림즈 애플리케이션을 포함하며 컨테이너를 실행시킬 수 있는 모든 머신에 실행될 수 있다. 이제 컨테이너를 사용하여 ksqlDB 또는 카프카 스트림즈 워크로드를 실행시킬 수 있기 때문에 이 시점에서는 ksqlDB와 동등하다. 그렇다면 이들 컨테이너를 어떻게 실행하는가? 다음 절에서 하나를 추천할 것이다.

컨테이너 오케스트레이션

카프카 스트림즈 또는 ksqlDB 애플리케이션을 실행시키는 가장 좋은 방법은 쿠버네티스와 같은 컨테이너 오케스트레이션 시스템을 사용하는 것이라고 믿고 있다. 이는 다음의 장점을 갖고 있다:

- 컨테이너가 실행 중인 노드가 실패한다면 오케스트레이터가 자동으로 컨테이너를 정상 노드로 이동시킬 것이다.
- 컨테이너의 복제 수를 증가시킴으로써 워크로드를 확장하는 것이 쉽다.
- 기반 인프라가 추상화된다. 그러나 컨테이너 실행 장소에 대해 제어를 원하는 경우 보다 많은 제어를 얻기 위해 node selector 또는 affinity 규칙을 사용할 수 있다.
- 다음과 같은 서비스를 제공하기 위해 카프카 스트림즈 또는 ksqlDB 컨테이와 함께 실행되는

소위 사이드카 컨테이너를 쉽게 배치할 수 있다:

-지표 수집 및 모니터링 시스템으로 이를 내보내기

-로그 전달

-통신 프록시 (예를 들어 상호대화식 쿼리가 활성화되어 있는 카프카 스트림즈 앱 전면에 인증 계층을 놓길 원한다면 이는 카프카 스트림즈 애플리케이션 내부에 인증 및 인가 계층을 구축하는 것보다 쉽다.

- 코드 업데이트 푸시 시 애플리케이션의 롤링 재시작을 조정하는 것이 쉽다.
- StatefulSet 리소스 지원은 스테이트풀 토폴로지에 대해 영구적이고 안정적인 스토리지를 제공한다.

쿠버네티스를 자세히 논의하는 것은 이 책의 범위를 벗어나며 Kubernetes: Up and Running by Brendan Burns et al. (O'Reilly)를 포함하여 이 주제에 대해 많은 훌륭한 자료들이 존재한다.

운영

이 마지막 절에서는 카프카 스트림즈 또는 ksqlDB 애플리케이션을 유지하면서 직면할 가능성이 높은 몇 가지 운영 태스크를 다룰 것이다.

카프카 스트림즈 애플리케이션 초기화

어떤 경우 카프카 스트림즈 애플리케이션을 초기화해야 할 수도 있다. 일반적인 유스케이스는 시스템에서 버그를 발견하고 카프카 토픽의 데이터 전부 또는 일부를 재처리해야 하는 경우이다. 이를 용이하게 하기 위해 카프카 관리자가 카프카 소스와 함께 배포되는 애플리케이션 초기화 툴을 구축하였다.

이 툴은 이론적으로 ksqlDB에도 사용할 수 있다. 그러나 이 툴을 사용하기 위해서는 컨슈머 그룹을 알아야 한다. 카프카 스트림즈에는 application.id 구성과 컨슈머 그룹 ID 간 직접적인 매핑이 존재한다. 그러나 ksqlDB의 경우 service.id와 컨슈머 그룹 간 직접적인 매핑이 존재하지 않는다.

따라서 ksqlDB 애플리케이션을 초기화하기 위해 이들 지침을 채택한다면 각별히 주의하고 애플리케이션에 대해 정확한 컨슈머 그룹을 식별했는지를 보장하기 바란다 (kafka-consumer-groups 컨소 스크립트가 도움이 될 수 있다).

이 툴은 몇 가지를 수행한다:

- 소스 토픽에서 지정된 위치로 컨슈머 오프셋을 업데이트한다.
- 중간 토픽의 끝까지 건너뛴다.
- 내부 체인지로그 및 리파티션 토픽을 삭제한다.

이 툴은 애플리케이션 상태를 초기화하지 않기 때문에 스테이트풀 애플리케이션에 대해 이 툴을

사용할 때 각별히 주의하기 바란다. 결정적인 단계를 놓치지 않도록 다음 지침을 완전히 읽기 바란다.

초기화 툴을 사용하기 위해 다음 단계를 따른다:

1. 애플리케이션의 모든 인스턴스를 중지한다. 컨슈머 그룹이 비활성 상태가 될 때까지 다음 단계로 넘어가지 않는다. 다음 명령을 실행시켜 컨슈머 그룹이 비활성 상태인지 여부를 결정할 수 있다. 비활성 컨슈머 그룹은 출력에서 상태가 EMPTY일 것이다:

```
kafka-consumer-groups ₩
 --bootstrap-server kafka:9092 ₩ ①
 --describe ₩
 --group dev ₩ ②
 --state
```

① 브로커의 호스트/포트 쌍을 부트스트랩 서버로 업데이트한다.

② 카프카 스트림즈에서 그룹은 application.id로 지정한다. ksqlDB 애플리케이션을 초기화하는 경우 불행히도 ksqlDB의 service.id 파라미터와 컨슈머 그룹 이름 간 직접적인 연관이 없다.

2. 적절한 파라미터로 애플리케이션 초기화 툴을 실행한다. 전체 파라미터 목록과 세부 사용법 정보는 다음을 명령을 실행하여 볼 수 있다:

```
kafka-streams-application-reset –help
```

예를 들어 초기에 생성했던 인사말 애플리케이션을 초기화하고 싶다면 다음을 실행시킬 수 있다:

```
kafka-streams-application-reset ₩
 --application-id dev ₩
 --bootstrap-servers kafka:9092 ₩
 --input-topics users ₩
 --to-earliest
```

출력은 다음과 비슷하게 보일 것이다:

```
Reset-offsets for input topics [users]
Following input topics offsets will be reset to (for consumer group dev)
Topic: users Partition: 3 Offset: 0
Topic: users Partition: 2 Offset: 0
Topic: users Partition: 1 Offset: 0
Topic: users Partition: 0 Offset: 0
Done.
Deleting all internal/auto-created topics for application dev
Done.
```

3. 애플리케이션이 스테이트리스라면 kafka-streams-application-reset 명령 실행 후 애플리케이션 인

스턴스를 재시작할 수 있다. 그렇지 않고 애플리케이션이 스테이트풀이라면 재시작 전에 애플리케이션 상태를 초기화해야 할 것이다. 2 가지 애플리케이션 상태 초기화 방법이 존재한다:

각 상태 디렉토리를 직접 삭제한다.

코드의 KafkaStreams.cleanUp 메소드 (카프카 스트림즈인 경우)를 호출한다. 초기화 툴 실행 후 애플리케이션을 최초 재시작할 때만 이를 하면 된다.

애플리케이션 상태가 이전 방법 중 하나를 사용하여 정리되었다면 애플리케이션을 재시작할 수 있다.

애플리케이션의 출력 속도 제한

카프카 스트림즈와 ksqlDB는 높은 처리량이 가능하지만 출력 토픽을 처리하는 다운스트림 시스템이 양을 따라갈 수 없는 상황에 놓일 수도 있다. 이 경우 레코드 캐시를 사용하여 애플리케이션의 출력 속도를 제한할 수 있다.

카프카 스트림즈 DSL 또는 ksqlDB를 사용할 때 레코드 캐시는 상태 저장소에 써지는 출력 레코드의 수와 다운스트림 프로세서로 전달되는 레코드의 수를 줄이는데 도움을 줄 수 있다. 프로세서 API를 사용할 때는 레코드 캐시가 상태 저장소에 써지는 레코드의 수만 줄일 수 있다.

(num.stream.threads 파라미터에 의해 제어되는) 각 스트림 쓰레드에 총 캐시 크기가 균등하게 할당된다. 따라서 총 캐시 크기가 10485760 (10 MB)이고 애플리케이션이 10 개 쓰레드로 동작하도록 구성된 경우 각 쓰레드에는 레코드 캐시를 위해 대략 1 MB의 메모리가 할당될 것이다.

이 설정은 애플리케이션뿐만 아니라 애플리케이션이 카프카에 생산하는 데이터에 의존하는 다운스트림 시스템의 성능을 향상시키는데 매우 중요하다. 이를 이해하기 위해 키 당 본 메시지 수를 세야 하는 토폴로지를 고려해보자. 다음 이벤트 시퀀스를 갖고 있는 경우:

- <key1, value1>
- <key1, value2>
- <key1, value3>

레코드 캐시를 비활성화하지 않고 (cache.max.bytes.buffering = 0) 카운트 집계를 수행하는 토폴로지는 다음 집계 시퀀스를 산출할 것이다:

- <key1, 1>
- <key1, 2>
- <key1, 3>

그러나 이 업데이트 시퀀스가 충분히 빨리 발생한다면 (예, 몇 밀리초 또는 몇 초) 모든 단일 중간 상태 (<key1, 1>과 <key1, 2>)에 대해 알아야 하는지? 이 질문에 대해 옳고 그른 답은 없으며 단지 유스케

이스와 얻고 싶은 시맨틱에 따라 다르다. 애플리케이션의 속도를 제한하고 중간 상태의 수를 줄이려면 레코드 캐시 사용을 원할 수 있다.

이 구성을 사용하여 레코드 캐시에 메모리를 할당한다면 카프카 스트림즈와 ksqlDB는 잠재적으로 원래의 집계 시퀀스를 단순히 다음으로 줄일 수 있다.

- <key1, 3>

캐시가 플러시될 때와 메시지가 도착할 때를 포함하여 여러 항목이 있기 때문에 레코드 캐시가 출력 레코드를 완전히 줄일 수 있을 것 같지는 않다. 그러나 이는 애플리케이션의 출력을 줄이는데 사용될 수 있는 중요한 레버이다. 애플리케이션이 보다 덜 쓸 때 토폴로지 자체의 다운스트림 프로세서와 카프카 스트림즈와 ksqlDB 외부에서 데이터에 대해 연산하는 다운스트림 시스팀이 처리할 데이터가 보다 작다.

이 중요한 설정에 대해 보다 자세한 정보는 공식 문서를 참조하기 바란다.

카프카 스트림즈 업그레이드

이 책 작성 시점에 아파치 카프카 (카프카 스트림즈 포함)는 시간 기반 출시 계획에 있었다. 따라서 매 4개월마다 새로운 카프카 스트림즈 버전이 출시될 것으로 예상되는데, 버전 전략은 다음과 같이 표시된다:

```
major.minor.bug-fix
```

대부분의 경우 4 달 주기 출시 주기는 마이너 버전에 적용된다. 예를 들어 2020년 4월에 2.5.0이 출시되었고 2.6.0은 4 개월 후인 2020년 8월에 출시되었다. 드문 경우로 더 영향력있는 변경 (예, 카프카 메시지 포맷 변경 또는 공개 API의 주요 변경) 또는 프로젝트에서 주요 마일스톤에 도달할 때 메이저 버전 충돌을 볼 수 있다. 버그 수정은 일반적인 것으로 언제라도 일어날 수 있다.

카프카 스트림즈를 업그레이드할 때마다 각 출시에 대해 공식 카프카 웹사이트에 발표되는 업그레이드 가이드를 따르는 것이 중요하다. 어떤 경우 upgrade.from 파라미터에 이전 버전을 설정하고 업그레이드가 안전하게 수행되었는지 확인하기 위해 몇 번의 재시작이 필요할 수도 있다. 이는 업그레이드 전에 잠시 기다린다면 특히 사실이다 (예, 2.3.0에서 2.6.0으로 업그레이드).

새로운 출시에 대한 정보를 받을 수 있는 몇 가지 옵션이 있다.

- 출시 발표를 받기 위해 하나 이상의 공식 메일링 리스트에 가입한다
- 아파치 카프카 공식 Github 저장소를 방문해 프로젝트를 "watch"한다 (소음을 줄이려면 출시 전용 옵션이 있다). 카프카 스트림즈와 보다 넓게 카프카 생태계에서 작업하는 사람들을 지원하기 위해 진행 중인 프로젝트에 별표를 표시할 수도 있다.
- 트위터에서 @apachekafka를 팔로우한다.

ksqlDB 업그레이드

ksqlDB는 매우 빠르게 진화하고 있으며 때때로 주요 업그레이드 경로를 약간 더 불안정하게 만드는 주요 변경 사항을 가질 가능성이 높다. 공식 ksqlDB 문서에 따르면:

> ksqlDB 버전 1.0까지 각 마이너 출시는 잠재적으로 주요 변경 사항을 가질 것이다. 이는 ksqlDB 바이너리를 단순히 다운로드한 후 서버를 재시작할 수 없다는 것을 의미한다.
>
> ksqlDB 내에서 사용하는 데이터 모델과 바이너리 포맷은 유동적이다. 이는 각 ksqbDB 노드에 로컬이고 내부 카프카 토픽 내부 중앙에 저장된 데이터가 배치하려는 새로운 버전과 호환되지 않을 수도 있음을 의미한다:
>
> -ksqldb.io

사실 문서는 운영 단계로 배치한 후 후방 호환성이 약속될 수 있을 때까지[6] 업그레이드를 좌절시키기까지 한다. 그러나 이것이 업그레이드하지 않아야 함을 의미하지는 않는다; 이는 단지 업그레이드에 수반되는 것, 업그레이드를 어떻게 수행해야 되는지와 업그레이드에 어떤 잠재적인 주요 변경 사항이 있는지에 대한 불확실성을 강조하는 것이다. ksqlDB 클러스터를 업그레이드하기 전에 항상 공식 ksqlDB 문서를 참조해야 한다.

요약

카프카 스트림즈와 ksqlDB 모두 운영 단계로 전달하기 전에 코드에 대해 신뢰를 구축할 수 있도록 하고 시간에 따라 애플리케이션을 계속 변경 및 개선해 나갈 때 의도치 않은 코드 퇴화 방지를 도울 수 있는 테스팅 유틸리티를 갖고 있다. 또한 어떤 기술을 사용하는지 상관없이 내장 JMX 지표를 사용하여 애플리케이션이 어떻게 운영되고 있는지에 대한 가시성을 유지하는 것은 쉽다. 또한 애플리케이션을 컨테이너 환경에서 실행시켜 이것이 카프카 스트림즈 애플리케이션에 대해 애플리케이션 특정적인 도커 이미지 구축 또는 ksqlDB 서버와 CLI 인스턴스 실행을 위해 컨플루언트 이미지 사용을 포함하는지에 상관없이 이들을 이식성이 높게 만들 수도 있다.

애플리케이션 모니터링, 벤치마킹, 속도 제한과 초기화에 대해 제공한 추가 팁은 스트림 처리 애플리케이션의 제품화와 유지를 도울 것이다. 이 책에서 제공한 정보와 함께 이들 지식은 스트림 처리 분야에서 최상 기술 중 2 가지를 사용하여 다양한 비즈니스 문제 해결을 도울 것으로 확신한다.

축하한다. 이 책의 끝이다

---

[6] 문서에서 정확한 인용은 다음과 같다: "제품 단계에서 ksqlDB를 실해시키고 있고 새로운 버전의 기능 또는 수정이 아직 필요치 않은 경우 필요한 기능 또는 수정을 갖는 다른 버전이 출시될 때까지 또는 ksqlDB 버전이 1.0에 도달하여 후방 호환성을 약속할 때까지 어떤 업그레이드도 연기하는 고려하기 바란다".