7장 프로세서 API

단지 몇 장 전에 카프카 스트림즈에 대해 배우는 여행을 시작했다. 함수적이고 능숙한 인터페이스를 사용하여 스트림 처리 애플리케이션을 구축할 수 있도록 하는 카프카 스트림즈의 고수준 DSL로 시작하였다. 이는 라이브러리의 내장 연산자 (예, filter, flatMap, groupBy 등)와 추상화 (KStream, KTable, GlobalKTable)를 사용한 스트림 처리 함수 구성 및 체이닝을 포함한다.

이 장에서는 카프카 스트림즈에서 사용가능한 저수준 API를 논의할 것이다: 프로세서 API (종종 PAPI로 부름). 프로세서 API는 고수준 DSL보다 적은 추상화를 가지며 명령형 프로그래밍 스타일을 사용한다. 일반적으로 코드가 더 장황하지만 다음에 대해 보다 미세한 제어가 가능하도록 함으로써 더욱 강력하다: 데이터가 토폴로지를 통해 어떻게 흐르는지, 스트림 프로세서가 서로 어떻게 연관되는지, 상태가 어떻게 생성 및 유지되는지와 특정 연산 타이밍.

이 장에서 우리가 답할 일부 질문은 다음을 포함한다:

- 언제 프로세서 API를 사용해야 하는지
- 프로세서 API를 사용하여 소스, 싱크 및 스트림 프로세서를 추가하는 법
- 주기 함수(periodic function)를 어떻게 스케줄링할 수 있는지
- 프로세서 API와 고수준 DSL을 혼합하는 것이 가능한지
- 프로세서와 트랜스포머 간 차이는 무엇인지

평소와 같이 진행 도중에 이전 질문에 답하면서 튜토리얼을 통해 API 기본을 보여줄 것이다. 그러나 프로세서 API를 어떻게 사용하는지를 보여주기 전에 이를 언제 사용할지를 논의해보자.

프로세서 API를 언제 사용하는지

스트림 처리 애플리케이션에 대해 어떤 추상화 수준을 사용할지를 결정하는 것은 중요하다. 일반적으로 프로젝트에 복잡성을 도입할 때마다 그렇게 하는 좋은 이유를 가져야 한다. 프로세서 API가 불필요하게 복잡하지는 않지만 DSL과 ksqlDB에 비해 저수준 특성과 더욱 적은 추상화는 더 많은 코드를 야기할 수 있고 주의하지 않는 경우 더 많은 실수를 야기할 수 있다.

일반적으로 다음 중 어떤 것이라도 이용할 필요가 있다면 프로세서 API를 활용할 수 있다:

- 레코드 메타 데이터에 대한 액세스 (토픽, 파티션, 오프셋 정보, 레코드 헤더 등)
- 주기 함수를 스케줄링할 수 있는 능력
- 레코드가 언제 다운스트림 프로세서에 전달되는지에 대한 더욱 미세한 제어
- 상태 저장소에 대한 보다 세분화된 액세스
- DSL에서 만날 수 있는 한계를 우회할 수 있는 능력 (후반부에 예를 볼 것이다).

반면 프로세스 API 사용 시 다음을 포함하여 몇 가지 단점이 존재한다:

- 보다 높은 유지보수 비용과 가독성 저해를 야기할 수 있는 보다 장황한 코드
- 다른 프로젝트 유지보수자에게 보다 높은 진입 장벽
- DSL 특성 또는 추상화의 우연한 재창조, 이상한 문제 프레이밍[1]과 성능 함정[2]을 포함하여 무슨 일이 일어날 것인지 더 모름

운좋게도 카프카 스트림즈는 애플리케이션에서 DSL과 프로세서 API 모두를 혼합하여 사용할 수 있도록 하며 따라서 한 가지를 선택해 올인할 필요가 없다. 여러분은 보다 간단하고 보다 표준적인 연산에 DSL을 사용할 수 있고 처리 컨텍스트, 상태 또는 레코드 메타 데이터에 대한 저수준 액세스를 필요로 하는 보다 복잡한 또는 고유한 기능에 대해 프로세서 API를 사용할 수 있다. 이 장 후반부에 DSL과 프로세서 API를 결합하는 방법을 논의할 것이지만 처음에는 프로세서 API만을 사용하여 애플리케이션을 구현하는 방법을 볼 것이다. 따라서 더 이상 고민하지 말고 이 장에서 구축할 애플리케이션을 살펴보자.

튜토리얼 소개: IoT 디지털 트윈 서비스

이 튜토리얼에서 해상 풍력 단지를 위한 디지털 트윈 서비스 구축을 위해 프로세서 API를 사용할 것이다. 디지털 트윈 (종종 디바이스 새도우(device shadow)라고 부름)은 물리적 객체의 상태가 디지털 복제본에 미러링되는 IoT (Internet of Things, 사물인터넷)와 IIoT (산업용 IoT) 유스케이스에서 인기가 높다. 이는 카프카 스트림즈의 훌륭한 유스케이스로 대용량의 센서 데이터를 쉽게 수집 및 처리하고 상태 저장소를 사용하여 물리적 객체의 상태를 수집하며 추후 상호대화형 쿼리를 사용하여 이 상태를 나타낼 수 있다.

디지털 트윈이 무엇인지에 대한 예를 보여주기 위해 다음을 고려해보자. 40 개의 풍력 터빈을 갖는 풍력 단지가 있다고 해보자. 터빈 중 하나가 현재 상태 (풍속, 온도, 전력 상태 등)를 보고할 때마다 키-값 저장소에 이 정보를 저장한다. 보고되는 상태 레코드 값의 예는 다음과 같다:

```
{
    "timestamp": "2020-11-23T09:02:00.000Z",
    "wind_speed_mph": 40,
    "temperature_fahrenheit": 60,
    "power": "ON"
}
```

---

[1] DSL은 표현력이 뛰어나고 내장 연산자를 이용할 때 문제 프레이밍이 쉽다. 이러한 풍부한 표현력과 연산자 자체를 쓰지 못함으로써 문제 프레이밍이 덜 표준화되어 솔루션 의사소통이 보다 어려워질 수 있다.

[2] 예를 들어 너무 공격적으로 커밋하거나 성능에 영향을 미칠 수 있는 방식으로 상태 저장소 액세스

디바이스 ID가 레코드 키를 통해 전달됨을 주목하기 바란다 (예, 이전 값은 abc123 ID를 갖는 장치에 해당할 수 있음). 이는 한 디바이스와 다른 디바이스에 대해 보고된/원하는 상태 이벤트를 구분할 수 있도록 한다.

이제 특정 풍력 터빈과 상호작용하고자 한다면[3] 우리는 직접적으로 그렇게 하지 않는다. IoT 디바이스는 자주 오프라인이 될 수 있고 그렇게 된다. 따라서 물리적 장치의 디지털 복제본 (트윈)과만 상호작용한다면 보다 높은 가용성을 얻을 수 있고 에러를 줄일 수 있다.

예를 들어 전력 상태를 ON에서 OFF로 설정하길 원한다면 터빈에 신호를 직접 보내는 대신 디지털 복제본의 소위 원하는 상태를 설정할 것이다. 물리적 터빈은 온라인이 될 때마다 또는 보통 이후 설정 간격으로 추후 상태를 동기화할 것이다 (블레이드로의 전력 차단). 따라서 디지털 트윈 레코드는 보고된 및 원하는 상태 모두를 포함할 것이며 카프카 스트림즈의 프로세서 API를 사용하여 다음과 같이 디지털 트윈 레코드를 생성 및 나타낼 것이다:

```
{
    "desired": {
        "timestamp": "2020-11-23T09:02:01.000Z", "power": "OFF" },
    "reported": {
        "timestamp": "2020-11-23T09:00:01.000Z",
        "windSpeedMph": 68,
        "power": "ON"
    }
}
```

이를 염두에 두고 애플리케이션은 일련의 풍력 터빈으로부터 센서 데이터 스트림을 수집[4]하고 데이터에 대해 일부 사소한 처리를 수행하며 영구 키-값 상태 저장소에 각 풍력 터빈의 최신 상태를 유지할 필요가 있다. 이후 카프카 스트림즈의 상호대화형 쿼리 기능을 통해 데이터를 나타낼 것이다.

상호대화형 쿼리는 이미 이전 장에서 다루었기 때문에 관련 기술적 세부사항을 피할 것이지만 IoT 유스케이스가 제공할 수 있는 상호대화형 쿼리에 대한 추가적인 가치를 설명할 것이다.

그림 7-1은 이 장에서 구축할 토폴로지를 보여준다. 각 단계는 그림 이후에 세부적으로 설명된다.

---

[3] 보통 상호작용은 디바이스의 최신 상태를 얻거나 전력 상태와 같이 가변 상태 특성 중 하나를 업데이트함으로써 디바이스 상태를 변경하는 것을 의미한다.

[4] IoT 유스케이스에서 센서 데이터는 MQTT 프로토콜을 통해 종종 전송된다. 이 데이터를 카프카 내로 전송하는 방법은 컨플루언트의 MQTT 카프카 커넥터를 사용할 것일 것이다.

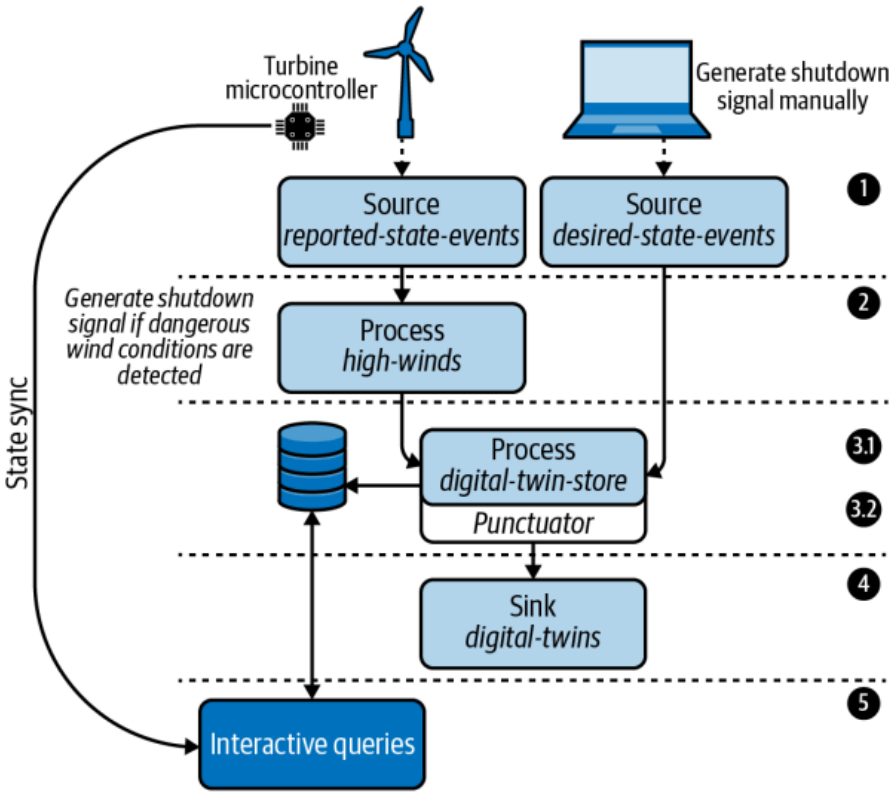

그림 7-1. IoT 디지털 트윈 서비스를 위해 구현할 토폴로지

① 카프카 클러스터는 2 개의 토픽을 포함하며 따라서 프로세서 API를 사용하여 소스 프로세서를 추가하는 법을 배울 필요가 있다. 다음은 토픽에 대한 설명이다:

- 각 풍력 터빈 (엣지 노드)에는 일련의 환경 센서가 장착되어 있으며 터빈 자체의 일부 메타데이터와 함께 이 데이터 (예, 풍속)가 주기적으로 reported-state-events 토픽에 전송된다.
- Desired-state-events 토픽은 사용자 또는 프로세스가 터빈의 전력 상태를 변경하고자 할 때 쓰여진다 (예, 전원 끄기 또는 켜기).

② 환경 센서 데이터가 reported-state-events 토픽에 보고되기 때문에 해당 터빈에 대해 보고된 풍속의 안전 운영 수준[5] 초과 여부를 결정할 스트림 프로세서를 추가할 것이고 그런 경우 자동적으로 셧다운 신호를 생성할 것이다. 이를 통해 프로세서 API를 사용하여 스테이트리스 스트림 프로세서를 추가하는 법을 배울 것이다.

③ 세번째 단계는 두 부분으로 나뉜다:

- 우선 (보고된 및 원하는) 두 유형의 이벤트가 소위 디지털 트윈 레코드로 결합될 것이다. 이 레코드는 처리되어 digital-twin-store라는 영구 키-값 저장소에 쓰여 질 것이다. 이 단계에

[5] 실제로 풍속 임계치는 양면성을 가질 수 있다. 풍속이 높을수록 위험한 운영 상태를 야기하는 반면 낮은 풍속은 터빈 가동 비용을 정당화할 수 없다.

서 프로세서 API를 사용하여 상태 저장소에 연결하고 이와 상호작용하는 법을 배울 것이며 또한 DSL을 통해 액세스할 수 없는 어떤 레코드 메타데이터에 액세스하는 법을 배울 것이다.

- 두번째 단계는 7일 이상 업데이트되지 않은 이전 디지털 트윈 레코드를 정리하는 punctuator라는 주기 함수의 스케줄링을 포함한다. 이는 프로세서 API의 punctuation 인터페이스를 소개할 것이며 또한 상태 저장소로부터 키를 제거하는 대안 방법을 보여줄 것이다[6].

④ 각각의 디지털 트윈 레코드는 분석 목적으로 digital-twins라는 출력 토픽에 써질 것이다. 이 단계에서 프로세서 API를 사용하여 싱크 프로세서를 추가하는 법을 배울 것이다.

⑤ 카프카 스트림즈의 상호대화형 쿼리 기능을 통해 디지털 트윈 레코드를 나타낼 것이다. 매 수초마다 풍력 터빈의 마이크로 컨트롤러는 자신의 상태와 카프카 스트림즈에 의해 나타낸 원하는 상태를 동기화하려고 할 것이다. 예를 들어 2 단계에서 셧다운 신호를 생성 (원하는 전력 상태를 OFF로 설정)했다면 터빈은 카프카 스트림즈 앱에 쿼리할 때 이 원하는 상태를 보고 블레이드의 전원을 차단할 것이다.

이제 우리가 무엇을 구축할지를 (그리고 각 단계에서 무엇을 배울지를) 이해했고 프로젝트 환경을 설정해보자.

프로젝트 환경설정

이 장의 코드는 https://github.com/mitch-seymour/mastering-kafkaa-streams-and-ksqldb.git에 위치해 있다.

각 토폴로지 단계를 통해 작업할 때 코드를 참조하고 싶다면 저장소를 복제하고 이 장 튜토리얼을 포함한 디렉토리로 이동한다. 다음 명령을 사용할 수 있다:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-07/digital-twin
```

다음 명령을 실행함으로써 프로젝트를 언제라도 빌드할 수 있다.

```
$ ./gradlew build –info
```

프로젝트 환경 설정이 끝났고 디지털 트윈 애플리케이션 구현을 시작해보자.

데이터 모델

평소와 같이 토폴로지를 생성하기 전에 우선 데이터 모델을 정의할 것이다. 표 7-1의 예제 레코드와 클래스 정의는 입력 토픽의 데이터에 해당한다 (그림 7-1 프로세서 토폴로지의 1단계).

---

[6] 다른 방법에 대해서는 툼스톤에 대한 논의를 보기 바란다.

두 입력 토픽을 통해 들어오는 데이터는 편의상 JSON 포맷임을 주목하기 바라며 두 레코드 타입은 공통 클래스인 TurbineState를 사용하여 표현된다:

편의상 TurbineState 클래스 내 액세서 함수는 생략하였다.

표 7-1. 각 소스 토픽에 대한 예제 레코드와 데이터 클래스

카프카 토픽	예제 레코드	데이터 클래스

Reported-state-events	{
	"timestamp": "...",
	"wind_speed_mph": 40,
	"power": "ON"
	}
		public class TurbineState {
		private String timestamp;
		private Double windSpeedMph;
		public enum Power { ON, OFF }
		public enum Type { DESIRED, REPORTED }
		private Power power;
		private Type type;
		}

Desired-state-events	{
	"timestamp": "...",
	"power": "OFF"
	}
		동일

튜토리얼 개요에서 언급했듯이 디지털 트윈 레코드를 생성하기 위해 보고된 및 원하는 상태 레코드를 결합해야 한다. 따라서 결합 레코드에 대한 데이터 클래스도 필요하다. 다음 테이블은 결합된 디지털 트윈 레코드와 해당 데이터 클래스의 JSON 구조를 보여준다.

예제 레코드	데이터 클래스

{
"desired": {
"timestamp": "2020-11-23T09:02:01.000Z",
"power": "OFF"
},
"reported": {
"timestamp": "2020-11-23T09:00:01.000Z",
"windSpeedMph": 68,
"power": "ON"
}
}
	public class DigitalTwin {
	private TurbineState desired;
	private TurbineState reported;
	// getters and setters omitted for
	// brevity
	}

예제 레코드에서 볼 수 있듯이 원하는 상태는 전원이 꺼진 터빈을 보여주지만 최종 보고된 상태는 전원이 켜져 있음을 보여준다. 결국 터빈은 디지털 트윈과 상태를 동기화하고 전원을 끌 것이다.

이제 이 시점에서 레코드 직렬화 및 역직렬화가 고수준 DSL과 비교해 프로세서 API에서 어떻게 동작하는지 궁금해할 수 있다. 다른 말로 카프카 토픽의 원시 레코드 바이트를 표 7-1의 데이터 클래스로 실제 어떻게 변환하는가? "직렬화/역직렬화"에서 직렬화기와 역직렬화기 모두를 포함하는 랩퍼 클래스인 DLS의 SerDes 클래스 사용에 대해 논의했다. Stream, table, join 등 많은 DSL 연산자는 SerDes

인스턴스를 지정할 수 있도록 하며 따라서 DSL을 사용하는 애플리케이션에서 이들은 평범한 클래스이다.

프로세서 API에서 다양한 API 메소드는 SerDes 인스턴스가 일반적으로 포함할 기본 직렬화기 또는 역직렬화기만을 필요로 한다. 그러나 데이터 클래스에 대해 SerDes를 정의하는 것이 종종 편리하다. 왜냐하면 프로세서 API 메소드 시그니처를 충족시키는 직렬화기/역직렬화기를 항상 추출할 수 있고 SerDes가 테스팅 목적에 종종 유용하기 때문이다.

이로 인해 이 튜토리얼에서는 에제 7-1의 SerDes 클래스를 활용할 것이다.

예제 7-1. 디지털 트윈 및 터빈 상태 레코드에 대한 SerDes

```
public class JsonSerdes {
  public static Serde DigitalTwin() { ①
    JsonSerializer<DigitalTwin> serializer = new JsonSerializer<>();
    JsonDeserializer<DigitalTwin> deserializer =
      new JsonDeserializer<>(DigitalTwin.class);
    return Serdes.serdeFrom(serializer, deserializer);
}
public static Serde TurbineState() { ②
  JsonSerializer<TurbineState> serializer = new JsonSerializer<>();
  JsonDeserializer<TurbineState> deserializer =
    new JsonDeserializer<>(TurbineState.class);
  return Serdes.serdeFrom(serializer, deserializer);③
  }
}
```

① DIgitalTwin SerDes 검색 메소드

② TurbineState SerDes 검색 메소드

③ 이전 튜토리얼에서 SerDes 인터페이스를 직접 구현했다 ("트윗 SerDes 구축" 참조). 이는 대안 방법으로 직렬화기 및 역직렬화기 인스턴스로부터 SerDes를 구축하기 위해 카프카 스트림즈에서 SerDes.serdeFrom 메소드를 사용하는 것이다.

다음 절에서 프로세서 API를 사용하여 소스 프로세서를 추가하고 입력 레코드를 역직렬화하는 법을 배울 것이다.

소스 프로세서 추가하기

이제 데이터 클래스를 정의하였고 (그림 7-1의) 프로세서 토폴로지의 1 단계를 다룰 준비가 되어 있다. 이는 입력 토픽으로부터 데이터를 카프카 스트림즈 애플리케이션 내로 스트리밍할 수 있도록 하는 2개의 소스 프로세서 추가를 포함한다. 예제 7-2는 프로세서 API를 사용하여 이를 달성하는 법을 보여준다.

에제 7-2. 2 개의 소스 프로세서가 추가된 초기 토폴로지

```
Topology builder = new Topology(); ①
builder.addSource( ②
    "Desired State Events", ③
    Serdes.String().deserializer(), ④
    JsonSerdes.TurbineState().deserializer(), ⑤
    "desired-state-events"); ⑥
builder.addSource( ⑦
    "Reported State Events",
    Serdes.String().deserializer(),
    JsonSerdes.TurbineState().deserializer(),
    "reported-state-events");
```

① Topology 인스턴스를 직접 인스턴스화한다. 이는 소스, 싱크 및 스트림 프로세서를 연결하기 위해 사용할 인스턴스이다. 비고: Topology를 직접 인스턴스화하는 것은 StreamsBuilder 객체를 인스턴스화하고 StreamsBuilder 인스턴스에 DSL 연산자 (예, map, flatMap, merge, branch 등)를 추가하여 StreamsBuilder#Builder 메소드를 사용하여 Topology를 최종적으로 구축하는 것을 필요로 하는 DSL을 사용하여 작업하는 것과는 다르다.

② 소스 프로세서를 생성하기 위해 addSource 메소드를 사용한다. 이 메소드에 대해 오프셋 리셋 전략, 토픽 패턴 등을 지원하는 변형을 포함하여 많은 오버로드 버전이 존재한다. 따라서 카프카 스트림즈의 Javadocs를 검사하거나 IDE를 사용하는 경우 Topology 클래스로 이동하여 니즈에 가장 잘 맞는 addSource 변형을 선택하기 바란다.

③ 소스 프로세서명. 각 프로세서는 내부에서 카프카 스트림즈가 프로세서명을 토폴로지적으로 정렬된 맵으로 저장하기 때문에 고유한 이름을 가져야 한다 (따라서 각 키는 고유해야 한다). 잠시 후 볼 것이지만 자식 프로세서를 연결하기 위해 사용되기 때문에 프로세서 API에서 이름은 중요하다. 다시 이는 프로세서 간 관계를 정의하기 위해 명시적 이름을 필요로 하지 않는 DSL이 연결하는 방법과 다르다 (기본적으로 DSL은 내부 이름을 생성한다). 코드 가독성 개선을 위해 여기서는 설명적인 이름을 사용하는 것이 좋다.

④ 키 역직렬화기. 여기서는 키가 스트링 포맷이기 때문에 내장 String 역직렬화기를 사용한다. 이는 프로세서 API와 DSL 간 다른 차이이다. 후자는 (레코드 직렬화기와 역직렬화기 모두를 포함하는) SerDes에 전달을 필요로 하지만 프로세서 API는 (여기서 하듯이 SerDes로부터 직접 추출될 수 있는) 역직렬화기만을 필요로 한다.

⑤ 값 역직렬화기. 여기서는 레코드 값은 TurbineState 객체로 변환하기 위해 (이 튜토리얼의 소스 코드에 있는) 맞춤형 SerDes를 사용한다. 키 역직렬화기에 대한 추가적인 참고사항 또한 여기에 적용된다.

⑥ 소스 프로세서가 소비하는 토픽의 이름

⑦ reported-state-events 토픽을 위해 두번째 소스 프로세서를 추가한다. 파라미터 타입이 이전 소스 프로세서와 일치하기 때문에 다시 각 파라미터를 언급하지 않을 것이다.

이전 예제에서 주목해야 할 한 가지 사실은 스트림 또는 테이블에 대한 언급을 볼 수 없다는 것이다. 이 추상화는 프로세서 API에는 존재하지 않는다. 그러나 개념적으로 말해서 이젠 코드에서 추가한 두 소스 프로세서는 스트림을 표현한다. 이는 프로세서가 스테이트리스고 (상태 저장소에 연결되지 않음) 따라서 해당 키의 최신 상태/표현을 기억할 방법이 없기 때문이다.

프로세서 토폴로지의 3 단계에서 스트림의 테이블과 같은 표현을 볼 것이지만 이 절은 토폴로지 1 단계를 다룬다. 이제 풍력 터빈이 위험한 풍속을 보고할 때 셧다운 신호를 생성하기 위한 스트림 프로세서를 추가하는 법을 살펴보자.

스테이트리스 스트림 프로세서 추가하기

프로세서 토폴로지의 다음 단계는 터빈에 의해 기록된 풍속이 안전 운영 수준 (65 mph)를 초과할 때마다 자동적으로 셧다운 신호를 생성하는 것을 필요로 한다. 이를 위해 프로세서 API를 사용하여 스트림 프로세서를 추가하는 법을 배워야 한다. 이 목적으로 사용할 수 있는 API 메소드는 addProcessor로 이 메소드 사용법 예는 다음과 같다:

```
builder.addProcessor(
    "High Winds Flatmap Processor", ①
    HighWindsFlatmapProcessor::new, ②
    "Reported State Events"); ③
```

① 스트림 프로세서 이름

② 두번째 인수는 Processor 인터페이스를 반환하는 함수 인터페이스인 ProcessSupplier를 제공하길 예상한다. Processor 인스턴스는 스트림 프로세서에 대한 모든 데이터 처리/변환 로직을 포함하고 있다. 다음 절에서 Processor 인터페이스를 구현할 HighWIndsFlatmapProcessor 클래스를 정의할 것이다. 따라서 그 클래스 생성자에 대한 메소드 참조를 사용할 수 있다.

③ 부모 프로세서의 이름. 이 경우 예제 7-2에서 생성했던 Reported State Events 프로세서인 하나의 부모 프로세스만을 갖는다. 스트림 프로세서는 하나 이상의 부모 노드에 연결될 수 있다.

스트림 프로세서를 추가할 때마다 Processor 인터페이스를 구현해야 한다. 이는 (논의했던 ProcessSupplier과는 달리) 함수 인터페이스가 아니며 따라서 DSL 연산자로 종종 했던 것과 같이 addProcessor 메소드에 람다 표현식을 전달할 수 없다. 이는 조금 더 복잡하고 여러분이 익숙한 보다 많은 코드를 필요로 하며 다음 절에서 이를 수행하는 방법을 살펴볼 것이다.

스테이트리스 프로세서 생성하기

프로세서 API에서 addProcess 메소드를 추가할 때마다 스트림 내 레코드를 처리 및 변환하기 위한 로직을 포함할 Processor 인터페이스를 구현해야 한다. 인터페이스는 다음과 같이 2 개의 메소드를 갖고

있다:

```
public interface Processor { ①
    void init(ProcessorContext context); ②
    void process(K key, V value); ③
    void close(); ④
}
```

① Processor 인터페이스가 2 개의 제네릭을 지정함을 주의하기 바란다: 키 타입 (K)과 값 타입 (V)에 대해 각각 하나. 이 절 후반에 프로세서를 구현할 때 이러한 제네릭을 활용하는 법을 살펴볼 것이다.

② Processor가 처음 인스턴스화될 때 init 메소드가 호출된다. 프로세서가 어떤 초기화 태스크를 수행해야 한다면 이 메소드에 초기화 로직을 지정할 수 있다. Init 메소드에 전달되는 ProcessorContext는 매우 유용하며 이 장에서 논의할 많은 메소드를 포함하고 있다.

③ 이 프로세서가 새로운 레코드를 받을 때마다 process 메소드가 호출된다. 이는 레코드 당 데이터 변환/처리 로직을 포함하고 있다. 우리 예에서 이는 풍속이 터빈에 대한 안전 운영 수준 초과 여부를 탐지하기 위한 로직을 추가할 곳이다.

④ close 메소드는 카프카 스트림즈가 이 연산자로 종료될 때마다 카프카 스트림즈에 의해 호출된다 (예, 셧다운 동안). 이 메소드는 프로세서와 로컬 리소스에 대해 필요한 정리 로직을 일반적으로 캡슐화한다. 그러나 이 메소드에서 상태 저장소와 같은 카프카 스트림즈 관리 리소스를 정리하려고 하지 않아야 한다. 왜냐하면 그것들은 라이브러리 자체에 의해 다루어지기 때문이다.

이 인터페이스를 염두에 두고 풍속이 위험한 수준에 도달할 때 셧다운 신호를 생성할 Processor를 구현해보자. 에제 7-3의 코드는 high wind processor가 어떻게 생겼는지를 보여준다.

예제 7-3. 위험한 풍속을 탐지하는 Processor 구현

```
public class HighWindsFlatmapProcessor
   implements Processor<String, TurbineState, String, TurbineState> { ①
  private ProcessorContext<String, TurbineState> context;
  @Override
  public void init(ProcessorContext<String, TurbineState> context) { ②
    this.context = context; ③
  }
  @Override
  public void process(Record<String, TurbineState> record) {
    TurbineState reported = record.value();
    context.forward(record); ④
    if (reported.getWindSpeedMph() > 65 && reported.getPower() == Power.ON) { ⑤
      TurbineState desired = TurbineState.clone(reported); ⑥
```

```
        desired.setPower(Power.OFF);
        desired.setType(Type.DESIRED);
        Record<String, TurbineState> newRecord = ⑦
            new Record<>(record.key(), desired, record.timestamp());
        context.forward(newRecord); ⑧
      }
    }
    @Override
    public void close() {
  // nothing to do ⑨
    }
}
```

① Processor 인터페이스가 4 개의 제네릭을 파라미터로 가짐을 상기하기 바란다. 첫번째 두 제네릭 (Processor<String, TurbineState, ..., ...>)은 입력 키와 값 타입과 관련이 있다. 마지막 두 제네릭 (Processor<..., ..., String, TurbineState>)은 출력 키와 값 타입과 관련이 있다.

② ProcessorContext 인터페이스의 제네릭은 출력 키와 값 타입과 관련이 있다 (ProcessorContext<String, TurbineState>)

③ 여기서와 같이 인스턴스 특성을 프로세서 컨텍스트로 저장하는 것이 일반적이며 따라서 추후 이를 액세스할 수 있다 (process 및/또는 close 메소드로부터).

④ 레코드를 다운스트림 프로세서에 전달하고자 할 때마다 ProcessorContext 인스턴스 (context 특성에 이를 저장하였다)의 forward 메소드를 호출할 수 있다. 이 메소드는 전달하고자 하는 레코드를 받아들인다. 프로세서 구현에서 우리는 항상 보고된 상태 레코드를 전달하길 원하며 이는 이 라인에서 수정되지 않은 레코드를 사용하여 context.forward를 호출하는 이유이다.

⑤ 터빈이 셧다운 신호 전송을 위한 조건을 충족하는지 여부를 검사한다. 이 경우 풍속이 안전 임계치 (65 mph)를 초과하고 현재 터빈의 전원이 켜져 있는 지를 검사한다.

⑥ 이전 조건이 충족되는 경우 원하는 전원 상태인 OFF를 포함하는 새로운 레코드를 생성한다. 이미 원래 보고된 상태 레코드를 다운스트림으로 전송했고 원하는 상태 레코드를 이제 생성하고 있기 때문에 이는 flatMap 연산 유형이다 (프로세서가 하나의 입력 레코드로부터 2 개의 출력 레코드를 생성한다).

⑦ 상태 저장소에 저장할 원하는 상태를 포함하는 출력 레코드를 생성한다. 레코드 키와 타임스탬프는 입력 레코드로부터 상속된다.

⑧ 새로운 레코드 (셧다운 신호)를 다운스트림 프로세서로 보내기 위해 context.forward 메소드를 호출한다.

⑨ 이 프로세서에서 프로세서가 종료될 때 실행해야 할 특수한 로직은 없다.

위에서 볼 수 있듯이 Processor를 구현하는 것은 매우 쉽다. 한 가지 흥미로운 사실은 이와 같은 프로세서를 구축할 때 여러분은 항상 출력 레코드가 Process 구현 자체에서 어디로 전달될지 걱정할 필요가 없다는 것이다 (다운스트림 프로세서의 부모 이름을 설정함으로써 데이터 흐름을 정의할 수 있다). 이에 대한 예외는 다운스트림 프로세서 이름 목록을 받아들이는 ProcessorContext#forward 변형을 사용하는 경우로 이는 출력을 어떤 자식 프로세서로 전달할 지를 카프카 스트림즈에 전달한다. 이의 예는 다음과 같다:

```
context.forward(newRecord, "some-child-node");
```

여러분이 forward 메소드의 이 변형을 사용할지 여부는 출력을 모든 다운스트림 프로세서 또는 특정 다운스트림 프로세서에 브로드캐스팅하길 원하는지에 달려있다. 예를 들어 DSL의 branch 메소드는 출력을 사용가능한 다운스트림 프로세서의 일부에 브로캐스팅해야 하기 때문에 이 변형을 사용한다.

이제 (그림 7-1의) 프로세서 토폴로지의 2 단계를 완료했다. 다음에는 키-값 저장소에 디지털 트윈을 생성 및 저장하는 스테이트풀 스트림 프로세서를 구현해야 한다.

스테이트풀 프로세서 생성하기

"상태 저장소"에서 카프카 스트림즈의 스테이트풀 연산이 이전에 보았던 데이터의 메모리 유지를 위해 상태 저장소를 필요로 함을 배웠다. 디지털 트윈 레코드를 생성하기 위해서는 원하는 상태와 기록된 상태 이벤트를 단일 레코드로 결합해야 한다. 이러한 레코드는 풍력 터빈에 대해 다른 시간에 도착할 것이기 때문에 각 터빈에 대해 가장 최근에 기록된 및 원하는 상태 레코드를 기억해야 한다는 스테이트풀 요건을 갖는다.

지금까지 이 책에서 우리는 DSL에서 상태 저장소를 사용하는 것에 대부분 중점을 두었다. 더구나 DSL은 상태 저장소 사용에 대해 몇몇 다른 옵션을 제공한다. 여기서는 다음과 같이 상태 저장소 지정없이 단순히 스테이트풀 연산자를 사용함으로써 기본 내부 상태 저장소를 사용할 수 있다:

```
Grouped.aggregate(initializer, adder);
```

또는 상태 공급자를 생성하기 위해 Stores 팩토리 클래스를 사용하고 다음 코드와 같이 스테이트풀 연산자와 함께 Materialized 클래스를 사용하여 상태 저장소를 구체화할 수 있다:

```
KeyValueBytesStoreSupplier storeSupplier =
    Stores.persistentTimestampedKeyValueStore("my-store");
grouped.aggregate(
    initializer,
    adder,
    Materialized.as(storeSupplier));
```

프로세서 API에서 상태 저장소를 사용하는 것은 약간 다르다. DSL과 달리 프로세서 API는 내부 상태 저장소를 생성하지 않는다. 따라서 스테이트풀 연산을 수행해야 할 때 항상 상태 저장소를 생성하고 이를 적절한 스트림 프로세서에 연결해야 한다. 또한 Stores 팩토리 클래스를 사용할 수도 있지만 상

태 저장소 생성을 위해 이 클래스에서 사용가능한 일련의 다른 메소드를 사용할 것이다. Store supplier를 반환하는 메소드 중 하나를 사용하는 대신 store builders를 생성하는 메소드를 사용할 것이다.

예를 들어 디지털 트윈 레코드를 저장하기 위해서는 간단한 키-값 저장소가 필요하다. 키-값 저장소 빌더를 검색하는 팩토리 메소드는 KeyValueStoreBuilder이라고 하는데 다음 코드는 디지털 트윈 저장소를 생성하기 위해 이 메소드를 사용하는 방법을 보여준다:

```
StoreBuilder<KeyValueStore> storeBuilder =
    Stores.keyValueStoreBuilder(
        Stores.persistentKeyValueStore("digital-twin-store"),
        Serdes.String(), ①
        JsonSerdes.DigitalTwin()); ②
```

① 키 직렬화/역직렬화를 위해 내장 String SerDes를 사용할 것이다.

② 값 직렬화/역직렬화를 위해 예제 7-1에 정의했던 SerDes를 사용한다.

스테이트풀 프로세서에 대한 상태 빌더를 생성하였다면 Processor 인터페이스를 구현할 차례이다. 이 프로세스는 "스테이트리스 프로세서 생성하기"에서 스테이트리스 프로세서를 추가했을 때 했던 것과 비슷하다. 다음 코드와 같이 프로세서 API의 addProcess 메소드를 사용하기만 하면 된다:

```
builder.addProcessor(
  "Digital Twin Processor", ①
  DigitalTwinProcessor::new, ②
  "High Winds Flatmap Processor", "Desired State Events"); ③
```

① 스트림 프로세서의 이름

②Processor 인스턴스를 검색하는데 사용될 수 있는 메소드인 ProcessSupplier. 이 라인에서 참조되는 DigitalTwinProcessor을 구현할 것이다.

③ 부모 프로세서의 이름. 복수의 부모를 지정함으로써 DSL의 merge 연산일 수 있는 것을 효과적으로 수행한다.

DigitalTwinProcessor을 구현하기 전에 계속해서 토폴로지에 새로운 상태 저장소를 추가해보자. Topology#addStateStore 메소드를 사용하여 이를 수행할 수 있으며 예제 7-4에 이를 나타내었다.

예제 7-4. addStateStore 메소드 사용 예

```
builder.addStateStore(
  storeBuilder, ①
  "Digital Twin Processor" ②
);
```

① 상태 저장소를 얻기 위해 사용될 수 있는 저장소 빌더

② 선택적으로 이 저장소에 액세스해야 하는 프로세서의 이름을 전달할 수 있다. 이 경우 이전 코드

블록에서 생성했던 DigitalTwinProcessor가 액세스해야 한다. 상태를 공유하는 복수의 프로세서를 갖고 있는 경우 더 많은 프로세서 이름을 전달할 수도 있다. 마지막으로 이 선택적인 인수를 생략하는 경우 저장소가 토폴로지에 추가된 이후 (위에서 수행했던 동일 시간 대신) 상태 저장소를 프로세서에 연결하기 위해 Topology#connectProcessorAndState를 대신 사용할 수 있다.

마지막 단계는 새로운 스테이트풀 프로세서인 DigitalTwinProcessor을 구현하는 것이다. 스테이트리스 스트림 프로세서와 같이 Processor 인터페이스를 구현해야 할 것이다. 그러나 이 프로세서가 상태 저장소와 상호작용해야 하기 때문에 구현이 좀더 복잡해질 것이다. 예제 7-5의 코드와 다음의 주석은 스테이트풀 프로세서 구현 방법을 설명할 것이다.

예제 7-5. 디지털 트윈 레코드 생성을 위한 스테이트풀 프로세서

```
public class DigitalTwinProcessor
    implements Processor<String, TurbineState, String, DigitalTwin> { ①
  private ProcessorContext<String, DigitalTwin> context;
  private KeyValueStore<String, DigitalTwin> kvStore;
  @Override
  public void init(ProcessorContext context) { ②
    this.context = context; ③
    this.kvStore = (KeyValueStore) context.getStateStore("digital-twin-store"); ④
  }
  @Override
  public void process(Record<String, TurbineState> record) {
    String key = record.key(); ⑤
    TurbineState value = record.value();
    DigitalTwin digitalTwin = kvStore.get(key); ⑥
    if (digitalTwin == null) { ⑦
      digitalTwin = new DigitalTwin();
    }
    if (value.getType() == Type.DESIRED) { ⑧
      digitalTwin.setDesired(value);
    } else if (value.getType() == Type.REPORTED) {
      digitalTwin.setReported(value);
    }
    kvStore.put(key, digitalTwin); ⑨
    Record<String, DigitalTwin> newRecord =
      new Record<>(record.key(), digitalTwin, record.timestamp()); ⑩
      context.forward(newRecord); ⑪
  }
  @Override
  public void close() {
```

```
    // nothing to do
  }
}
```

① Processor 인터페이스의 첫번째 두 제네릭 (Processor<String, TurbineState, ..., ...>)은 입력 키와 값 타입과 관련이 있다. 마지막 두 제네릭 (Processor<..., ..., String, TurbineState>)은 출력 키와 값 타입과 관련이 있다.

② ProcessorContext 인터페이스의 제네릭은 출력 키와 값 타입과 관련이 있다 (ProcessorContext<String, TurbineState>)

③ ProcessorContext (context 특성에 의해 참조)를 저장하며 추후 이를 액세스할 수 있다.

④ ProcessorContext의 getStateStore 메소드를 통해 이전에 스트림 프로세서에 연결했던 상태 저장소를 검색할 수 있도록 한다. 레코드가 처리될 때마다 이 상태 저장소와 직접 상호작용할 것이며 따라서 kvStore라는 인스턴스 특성에 이를 저장할 것이다.

⑤ 이 라인과 다음 라인은 입력 레코드의 키와 값을 추출하는 방법을 보여준다.

⑥ 현재 레코드 키에 대한 포인트 조회를 수행하기 위해 키-값 저장소를 사용한다. 이전에 이 키를 본 경우 이전에 저장된 디지털 트윈 레코드를 반환할 것이다.

⑦ 포인트 조회가 아무런 결과도 반환하지 않는다면 새로운 디지털 트윈 레코드를 생성할 것이다.

⑧ 이 코드 블록에서 현재 레코드 타입 (보고된 상태 또는 원하는 상태)에 따라 디지털 트윈 레코드에 적절한 값을 설정한다.

⑨ 키-값 저장소의 put 메소드를 사용하여 상태 저장소에 직접 디지털 트윈 레코드를 저장한다.

⑩ 상태 저장소에 저장했던 디지털 트윈 인스턴스를 포함하는 출력 레코드를 생성한다. 레코드 키와 타임스탬프는 입력 레코드로부터 상속된다.

⑪ 출력 레코드를 다운스트림 프로세서에 전달한다.

이제 프로세서 토폴로지의 3단계 중 첫번째 부분을 구현하였다 (그림 7-1의 3.2 단계) 다음 단계는 DSL에 대응되는 것이 없는 프로세서 API의 매우 중요한 기능을 소개할 것이다. DSL에서 주기 함수를 스케줄링하는 방법을 살펴보자.

Punctuate을 사용하는 주기 함수

유스케이스에 따라 카프카 스트림즈 애플리케이션에서 주기적 태스크를 수행할 필요가 있다. 이는 프로세서 API가 실제로 빛을 발하는 분야로 ProcessorContext#schedule 메소드를 사용하여 쉽게 태스크를 스케줄링할 수 있기 때문이다. 불필요한 레코드를 삭제함으로써 상태 저장소 크기를 최소로 유지하는 방법을 논의했던 "툼스톤"에서 이를 상기하기 바란다. 이 튜토리얼에서는 상태 저장소 정리를 위해 이러한 태스크 스케줄링 능력을 활용하는 다른 방법을 나타낼 것이다. 여기서는 지난 7일 동안 어떠한

상태 업데이트도 없는 모든 디지털 트윈 레코드를 삭제할 것이다. 이들 터빈이 더 이상 활성이 아니거나 장기 유지보수에 놓여있다고 가정할 것이며 따라서 키-값 저장소에서 이들 레코드를 삭제할 것이다.

5 장에서 스트림 처리와 관련하여 시간이 복잡한 주제임을 보였다. 카프카 스트림즈에서 주기 함수가 실행될 때에 대해 생각할 때 우리는 이러한 복잡성을 기억한다. 표 7-2와 같이 선택할 수 있는 2 가지 punctuation 타입 (즉, 타이밍 전략)이 존재한다.

표 7-2. 카프카 스트림즈에서 사용가능한 punctuation 타입

Punctuation 타입	Enum	설명
Stream time	PunctuationType.STREAM_TIME	특정 토픽-파티션에서 관측되는 가장 큰 타임스탬프. 초기에는 모르며 단지 증가하거나 동일하게 유지될 수만 있다. 이는 새로운 데이터가 보이는 경우에만 전진하며 따라서 이 타입을 사용하는 경우 데이터가 연속적으로 들어오지 않는다면 함수는 실행되지 않을 것이다.
Wall clock time	PunctuationType.WALL_CLOCK_TIME	컨슈머 폴 메소드의 각 반복 동안 전진하는 로컬 시스템 타임. 얼마나 자주 업데이트되는지에 대한 상한은 StreamsConfig#POLL_MS_CONFIG 설정에 의해 정의된다. 이는 새로운 데이터를 기다릴 때 폴 메소드가 차단할 밀리초 단위의 시간의 최대량이다. 이는 새로운 메시지 도착 여부에 상관없이 주기적 함수가 계속 실행될 것임을 의미한다.

우리는 주기적 함수가 도착하는 새로운 데이터와 연계되는 것을 원하지 않기 때문에 (사실 이 TTL ("time to live") 함수의 존재는 데이터 도착이 중지될 수 있다는 가정에 기반한다) punctuation 타입으로 wall clock time을 사용할 것이다. 이제 사용할 추상화를 결정했기 때문에 나머지 작업은 단순히 TTL 함수를 스케줄링하고 구현하는 것이다.

다음 코드는 구현을 보여준다:

```
public class DigitalTwinProcessor
    implements Processor<String, TurbineState, String, DigitalTwin> {
  private Cancellable punctuator; ①
  // other omitted for brevity
  @Override
  public void init(ProcessorContext<String, DigitalTwin> context) {
    punctuator = this.context.schedule(
      Duration.ofMinutes(5),
      PunctuationType.WALL_CLOCK_TIME, this::enforceTtl); ②
  // ...
  }
```

```
  @Override
  public void close() {
   punctuator.cancel(); ③
 }
 public void enforceTtl(Long timestamp) {
   try (KeyValueIterator<String, DigitalTwin> iter = kvStore.all()) { ④
     while (iter.hasNext()) {
       KeyValue<String, DigitalTwin> entry = iter.next();
       TurbineState lastReportedState = entry.value.getReported(); ⑤
       if (lastReportedState == null) {
         continue;
       }
       Instant lastUpdated = Instant.parse(lastReportedState.getTimestamp());
       long daysSinceLastUpdate =
         Duration.between(lastUpdated, Instant.now()).toDays(); ⑥
      if (daysSinceLastUpdate >= 7) {
        kvStore.delete(entry.key); ⑦
       }
     }
   }
 }
 // ...
}
```

① punctuator 함수를 스케줄링할 때 추후 예약된 함수를 중지시키기 위해 사용할 수 있는 Cancellable 객체를 반환할 것이다. 이 객체를 추적하기 위해 punctuator라는 객체 변수를 사용할 것이다.

② wall clock time에 기반하여 매 5분 마다 실행할 주기 함수를 스케줄링하고 punctuator 특성 하에 반환되는 Cancellable을 저장한다 (이전 callout 참조)

③ 프로세서 종료 시 punctuator을 취소한다 (예, 카프카 스트림즈 애플리케이션의 클린 셧다운 동안).

④ 각 함수 호출 동안 상태 저장소에서 각 값을 검색한다. 반복자가 적절히 종료됨을 보장하기 위해 try with resources 문을 사용함을 주목하기 바란다. 이는 리소스 누수를 방지할 것이다.

⑤ 현재 레코드의 가장 최신 보고된 상태를 추출한다 (물리적인 풍력 터빈에 해당).

⑥ 이 터빈이 상태를 마지막으로 보고한 이후 얼마나 지났는지 (일 단위) 결정한다.

⑦ stale로 최소 7일 동안 업데이트 되지 않았다면 상태 저장소로부터 레코드를 삭제한다.

스케줄링하는 process 함수와 punctuation은 동일 쓰레드로 실행될 것이며 (즉 punctuation을

위한 백그라운드 쓰레드가 없다) 따라서 동시성 문제를 걱정할 필요가 없다.

위에서 볼 수 있듯이 주기 함수를 스케줄링하는 것은 매우 쉽다. 이제 프로세서 API가 빛을 발하는 다른 분야인 레코드 메타데이터 액세스하기를 살펴보자.

레코드 메타데이터 액세스하기

DSL 사용 시 일반적으로 레코드의 키와 값만 액세스하면 된다. 그러나 해당 레코드 관련 DSL에 의해 노출되지 않지만 프로세서 API를 사용하여 액세스할 수 있는 많은 추가 정보가 존재한다. 액세스하길 원할 수 있는 레코드 메타데이터의 보다 두드러진 일부 예를 표 7-3에 나타내었다. 다음 표에서 context 변수는 예제 7-3에서 처음 보았듯이 init 메소드에서 사용가능한 ProcessorContext의 인스턴스와 관련이 있음을 주목하기 바란다.

표 7-3. 추가적인 레코드 메타 데이터를 액세스하기 위한 메소드

메타데이터	예
레코드 헤더	context.headers()
오프셋	context.offset()
파티션	context.partition()
타임스탬프	context.timestamp()
토픽	context.topic()

표 7-3의 메소드는 현재 레코드에 대한 메타데이터를 가져오며 process() 함수 내에서 사용될 수 있다. 그러나 init() 또는 close() 함수가 호출되거나 punctuation이 실행 중일 때에는 현재 레코드가 없으며 따라서 추출할 메타데이터가 존재하지 않는다.

그렇다면 이 메타데이터로 무엇을 할 수 있을까? 하나의 유스케이스는 레코드가 어떤 다운스트림 시스템에 써지기 전에 추가적인 컨텍스트를 사용하여 레코드 값을 장식하는 것이다. 또한 디버깅에 도움이 되기 위해 이 정보를 사용하여 애플리케이션 로그를 장식할 수도 있다. 예를 들어 잘못된 형식의 레코드를 만나는 경우 해당 레코드의 파티션 및 오프셋을 포함하는 에러를 기록할 수 있고 추후 문제 해결의 기초로 이를 사용할 수 있다.

레코드 헤더는 추가적인 메타데이터를 삽입하는데 사용될 수 있기 때문에 흥미롭다 (예를 들어 분산 추적에 사용될 수 있는 컨텍스트 추적). 레코드 헤더와 상호작용하는 방법의 예는 다음과 같다:

```
Headers headers = context.headers();
headers.add("hello", "world".getBytes(StandardCharsets.UTF_8)); ①
headers.remove("goodbye"); ②
headers.toArray(); ③
```

① hello라는 이름의 헤더를 추가한다. 이 헤더는 다운스트림 프로세서로 전파될 것이다.

② goodbye라는 이름의 헤더를 삭제한다.

③ 모든 사용가능 헤더에 대한 배열을 얻는다. 이에 대해 반복할 수 있고 각각에 대해 무언가를 할

수도 있다.

마지막으로 레코드의 기원을 추적하고 싶다면 이 목적으로 topic() 메소드가 사용될 수 있다. 이 튜토리얼에서는 이를 수행하거나 실제 메타데이터를 액세스할 필요는 없지만 향후 이를 필요로 하는 유스케이스를 만나는 경우 추가적인 메타데이터에 액세스하는 방법에 대해 충분히 이해해야 한다.

프로세서 토폴로지의 다음 단계로 넘어가 프로세서 API를 사용하여 싱크 프로세서를 추가하는 법을 배울 준비가 되어 있다.

싱크 프로세서 추가하기

(그림 7-1의) 프로세서 토폴로지의 4 단계를 해결하는 것은 digital-twins 출력 토픽에 모든 디지털 트윈 레코드를 쓸 싱크 프로세서 추가를 포함한다. 이는 프로세서 API 사용 시 매우 간단하며 따라서 이 절은 짧을 것이다. 다음 코드에서와 같이 addSink 메소드를 사용해 몇 개의 추가적인 파라미터를 지정하기만 하면 된다:

```
builder.addSink(
  "Digital Twin Sink", ①
  "digital-twins", ②
  Serdes.String().serializer(), ③
  JsonSerdes.DigitalTwin().serializer(), ④
  "Digital Twin Processor"); ⑤
```

① 싱크 노드의 이름

② 출력 토픽의 이름

③ 키 직렬화기

④ 값 직렬화기

⑤ 싱크 노드에 연결된 하나 이상의 부모 노드 이름

이것이 싱크 프로세서를 추가하기 위한 전부이다. 물론 카프카 스트림즈의 대다수 메소드와 같이 활용할 수 있는 이 메소드의 몇몇 추가 변형이 존재한다. 예를 들어 한 변형은 출력 레코드를 파티션 넘버에 매핑하는 맞춤형 StreamPartitioner을 지정할 수 있도록 한다. 다른 변형은 키 및 값 직렬화기를 배제하고 대신 DEFAULT_KEY_SERDE_CLASS_CONFIG 특성으로부터 유도된 기본 직렬화기를 사용할 수 있도록 한다. 여러분이 어떤 오버로드 메소드를 사용하는지에 상관없이 싱크 프로세서를 추가하는 것은 매우 간단한 동작이다.

디지털 트윈 레코드를 외부 서비스에 노출시키는 마지막 단계로 가보자 (상태를 상태 저장소의 디지털 트윈 레코드로 동기화할 풍력 터빈 자체를 포함).

상호대화형 쿼리

토폴로지의 1-4 단계를 완료하였다. 5 단계는 카프카 스트림즈의 상호대화형 쿼리를 사용하여 디지털 트윈 레코드를 노출하는 것을 포함한다. 이미 "상호대화형 쿼리"에서 이 주제를 세부적으로 논의하였으며 따라서 세부 사항을 논의하거나 전체 구현을 보이지는 않을 것이다. 그러나 예제 7-6은 최신 디지털 트윈 레코드를 가져오기 위해 상호대화형 쿼리를 사용하는 매우 간단한 REST 서비스를 보여준다. 이 예에서 원격 쿼리는 없으며 더욱 완벽한 예제의 경우 이 예제의 소스 코드를 볼 수 있음을 주목하기 바란다.

주목해야 할 중요한 사실은 상호대화형 쿼리 관점에서 프로세서 API를 사용하는 것이 DSL을 사용하는 것과 완전히 동일하다는 것이다.

예제 7-6. 디지털 트윈 레코드를 노출시키기 위한 예제 REST 서비스

```
class RestService {
 private final HostInfo hostInfo;
 private final KafkaStreams streams;
 RestService(HostInfo hostInfo, KafkaStreams streams) {
 this.hostInfo = hostInfo;
 this.streams = streams;
 }
 ReadOnlyKeyValueStore<String, DigitalTwin> getStore() {
 return streams.store(
 StoreQueryParameters.fromNameAndType(
 "digital-twin-store", QueryableStoreTypes.keyValueStore()));
 }
 void start() {
 Javalin app = Javalin.create().start(hostInfo.port());
 app.get("/devices/:id", this::getDevice);
 }
 void getDevice(Context ctx) {
 String deviceId = ctx.pathParam("id");
 DigitalTwin latestState = getStore().get(deviceId);
 ctx.json(latestState);
 }
}
```

이제 프로세서 토폴로지의 5 단계를 완료했고 구축했던 다양한 조각들을 함께 모아보자.

함께 모으기

다음 코드 블록은 이 시점에서 전체 프로세서 토폴로지가 어떻게 생겼는지를 보여준다.

```
  Topology builder = new Topology();
  builder.addSource( ①
```

```
 "Desired State Events",
 Serdes.String().deserializer(),
 JsonSerdes.TurbineState().deserializer(),
 "desired-state-events");
builder.addSource( ②
 "Reported State Events",
 Serdes.String().deserializer(),
 JsonSerdes.TurbineState().deserializer(),
 "reported-state-events");
builder.addProcessor( ③
 "High Winds Flatmap Processor",
 HighWindsFlatmapProcessor::new,
 "Reported State Events");
builder.addProcessor( ④
 "Digital Twin Processor",
 DigitalTwinProcessor::new,
 "High Winds Flatmap Processor",
 "Desired State Events");
StoreBuilder<KeyValueStore<String, DigitalTwin>> storeBuilder =
 Stores.keyValueStoreBuilder( ⑤
 Stores.persistentKeyValueStore("digital-twin-store"),
 Serdes.String(),
 JsonSerdes.DigitalTwin());
builder.addStateStore(storeBuilder, "Digital Twin Processor"); ⑥
builder.addSink( ⑦
 "Digital Twin Sink",
 "digital-twins",
 Serdes.String().serializer(),
 JsonSerdes.DigitalTwin().serializer(),
 "Digital Twin Processor");
```

① desired-state-events 토픽으로부터 데이터를 소비하는 Desired State Events 소스 프로세서를 생성한다. 이는 DSL에서 stream에 해당한다.

② reported-state-events 토픽으로부터 데이터를 소비하는 Reported State Events 소스 프로세서를 생한다. 이도 또한 DSL의 stream에 해당한다.

③ 높은 풍속이 탐지되는 경우 셧다운 신호를 생성하는 High Winds Flatmap Processor 스트림 프로세서를 추가한다. 이 프로세서는 Reported State Events 프로세서로부터 이벤트를 수신한다. 이는 스트림 프로세서에 대해 입력 및 출력 레코드 수 간 1:N 관계가 존재하기 때문에 DSL의 flatMap 연산일 것이다. 예제 7-3은 이 프로세서의 구현을 보여준다.

④ High Winds Flatmap Processor과 Desired State Events 모두로부터 방출된 데이터를 사용하여 디지털 트윈 레코드를 생성하는 Digital Twin Processor 프로세서를 추가한다. 이는 복수의 소스가 포함되기 때문에 DSL의 merge 연산일 것이다. 더구나 스테이트풀 프로세서이기 때문에 이는 DSL의 집계 테이블일 것이다. 예제 7-5는 이 프로세서의 구현을 보여준다.

⑤ Digital Twin Processor 노드로부터 액세스 가능한 영구 키-값 저장소를 구축하기 위해 카프카 스트림즈가 사용할 수 있는 store builder을 생성하기 위해 Stores 팩토리 클래스를 사용한다.

⑥ 상태 저장소를 토폴로지에 추가하고 이를 Digital Twin Process 노드에 연결한다.

⑦ Digital Twin Processor 노드로부터 방출된 모든 디지털 트윈 레코드를 digital-twins 출력 토픽에 쓰는 Digital Twin Sink 싱크 프로세서를 생성한다.

이제 애플리케이션을 실행하고 일부 테스트 데이터를 카프카에 써서 디지털 트윈 서비스에 쿼리할 수 있다 애플리케이션을 실행하는 것은 다음 코드 블록에서 볼 수 있듯이 이전 장에서 수행했던 것과 다르지 않다:

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "dev-consumer"); ①
// ...
KafkaStreams streams = new KafkaStreams(builder, props); ②
streams.start(); ③
Runtime.getRuntime().addShutdownHook(new Thread(streams::close)); ④
RestService service = new RestService(hostInfo, streams); ⑤
service.start();
```

① 카프카 스트림즈 애플리케이션을 구성한다. 이는 DSL을 사용하여 애플리케이션을 구축할 때 본 방식과 동일하게 작동한다. 편의상 대부분의 설정을 생략하였다.

② 토폴로지 실행에 사용될 수 있는 새로운 KafkaStreams 인스턴스를 인스턴스화한다.

③ 카프카 스트림즈 애플리케이션을 시작한다.

④ 글로벌 셧다운 신호 수신 시 카프카 스트림즈 애플리케이션을 우아하게 중지시키기 위해 셧다운 훅을 추가한다.

⑤ 예제 7-6에 구현했던 REST 서비스를 인스턴스화하고 시작한다.

이제 애플리케이션이 복수의 소스 토픽으로부터 읽지만 이 책에서는 reported-state-events 토픽에 테스트 데이터만을 생산할 것이다 (보다 완벽한 예의 겨우 소스 코드를 참조). 애플리케이션이 셧다운 신호를 생성하는지를 테스트하기 위해 안전 운영 임계치 65 mph를 초과하는 풍속을 포함하는 하나의 레코드를 포함할 것이다. 다음 코드는 | 로 구분된 레코드 키와 값을 갖는 우리가 생산할 테스트 데이터를 보여준다. 편의상 타임스탬프는 생략되었다:

```
1|{"timestamp": "...", "wind_speed_mph": 40, "power": "ON", "type": "REPORTED"}
1|{"timestamp": "...", "wind_speed_mph": 42, "power": "ON", "type": "REPORTED"}
1|{"timestamp": "...", "wind_speed_mph": 44, "power": "ON", "type": "REPORTED"}
1|{"timestamp": "...", "wind_speed_mph": 68, "power": "ON", "type": "REPORTED"} ①
```

① 이 센서 데이터의 풍속은 68 mph로 애플리케이션이 이 레코드를 보는 경우 원하는 전력 상태가 OFF인 새로운 TurbineState 레코드를 생성함으로써 셧다운 신호를 생성해야 한다.

이 테스트 데이터를 reported-state-events 토픽에 생산하고 디지털 트윈 서비스에 쿼리하는 경우 카프카 스트림즈 애플리케이션이 풍력 단지의 보고된 상태를 처리하고 전원이 OFF로 설정된 원하는 상태 레코드를 산출함을 볼 것이다. 다음 코드 블록은 REST 서비스에 대한 요청 및 응답 예를 보여준다:

```
$ curl localhost:7000/devices/1 | jq '.'
{
 "desired": {
 "timestamp": "2020-11-23T09:02:01.000Z",
 "windSpeedMph": 68,
 "power": "OFF",
 "type": "DESIRED"
 },
 "reported": {
 "timestamp": "2020-11-23T09:02:01.000Z",
 "windSpeedMph": 68,
 "power": "ON",
 "type": "REPORTED"
 }
```

이제 풍력 터빈은 REST 서비스에 쿼리하여 (desired-sate-events 토픽을 통해) 수집되었거나 또는 카프카 스트림즈에 의해 (셧다운 신호를 보내기 위해 high winds 프로세서를 사용하여) 강제된 원하는 상태로 자신의 상태를 동기화할 수 있다.

프로세서 API와 DSL 결합하기

애플리케이션이 동작하는 것을 확인하였다. 그러나 코드를 자세히 살펴보면 토폴로지 단계 중 하나의 단계만 프로세서 API가 제공하는 저수준 액세스를 필요로 한다. 언급한 단계는 그림 7-1의 3 단계인 Digital Twin Processor 단계로 프로세서 API의 중요한 기능을 활용하고 있다: 주기 함수 스케줄링.

카프카 스트림즈는 프로세서 API와 DSL을 결합할 수 있도록 하기 때문에 Digital Twin Processor 단계에 대해서만 프로세서 API를 사용하고 나머지의 경우 DSL를 사용하도록 애플리케이션을 쉽게 리팩토링할 수 있다. 이러한 유형의 리팩토링 수행 시 가장 큰 장점은 다른 스트림 처리 단계가 간단화될 수 있다는 것이다. 이 튜토리얼에서 High Winds Flatmap Process는 단순화에 대해 가장 큰 기회를 제공하는데 보다 큰 애플리케이션의 경우 이러한 유형의 리팩토링이 보다 큰 규모의 복잡성을 줄인다.

프로세서 토폴로지에서 첫번째 두 단계 (소스 프로세서 등록과 flatMap과 유사한 연산을 사용하여 셧다운 신호 생성)는 이 책에서 이미 논의했던 연산자를 사용하여 리팩토링될 수 있다. 구체적으로 다음과 같이 변경할 수 있다.

프로세서 API	DSL
Topology builder = new Topology(); builder.addSource("Desired State Events", Serdes.String().deserializer(), JsonSerdes.TurbineState().deserializer(), "desired-state-events"); builder.addSource("Reported State Events", Serdes.String().deserializer(), JsonSerdes.TurbineState().deserializer(), "reported-state-events"); builder.addProcessor("High Winds Flatmap Processor", HighWindsFlatmapProcessor::new, "Reported State Events");	StreamsBuilder builder = new StreamsBuilder(); KStream<String, TurbineState> desiredStateEvents = builder.stream("desired-state-events", Consumed.with(Serdes.String(), JsonSerdes.TurbineState())); KStream<String, TurbineState> highWinds = builder.stream("reported-state-events", Consumed.with(Serdes.String(), JsonSerdes.TurbineState())) .flatMapValues((key, reported) -> ...) .merge(desiredStateEvents);

위에서 볼 수 있듯이 이러한 변경은 매우 쉽다. 그러나 토폴로지에서 3 단계는 실제로 프로세서 API를 필요로 하는데 그렇다면 이 단계에서 DSL과 프로세서 API를 어떻게 결합할까? 답은 다음에 논의할 일련의 특수한 DSL 연산자에 달려있다.

프로세서와 트랜스포머

DSL은 상태 저장소, 레코드 메타데이터와 (주기 함수 스케줄링에 사용될 수 있는) 프로세서 컨텍스트에 대한 저수준 액세스가 필요할 때마다 프로세서 API를 사용할 수 있도록 하는 일련의 특수한 연산자를 포함하고 있다. 이 특수 연산자는 2 가지 범주로 나뉜다: 프로세서와 트랜스포머. 다음은 이들 간 차이점을 설명한다.

프로세서는 (void를 반환하고 다운스트림 연산자가 체이닝될 수 없음을 의미하는) 터미널 동작로 계산 로직이 Processor 인터페이스 ("스테이트리스 스트림 프로세서 추가하기"에서 처음 논의했던)를 사용하여 구현되어야 한다. 프로세서는 DSL에서 프로세서 API를 활용할 필요가 있을 때마다 사용되어야 하며 어떠한 다운스트림 연산자와 체이닝 될 필요는 없다. 다음 표에서 보듯이 이러한 유형의 연산자에 대해서는 현재 단지 하나의 변형만 존재한다:

DSL 연산자	구현 인터페이스	설명
process	Processor	Processor을 한 번에 각 레코드에 적용

트랜스포머는 보다 다양한 일련의 연산자로 (사용 변형에 따라) 하나 이상의 레코드를 반환할 수 있으며 따라서 다운스트림 연산자와 체이닝하는 경우 보다 최적이다. 변환 연산자의 변형은 표 7-4와 같다.

표 7-4. 카프카 스트림즈에서 사용가능한 다양한 변환 연산자

DSL 연산자	구현 인터페이스	설명	입력/출력 비율
transform	Transformer	각 레코드에 Transformer를 적용하여 하나 이상의 출력 레코드를 생성한다. Transformer#transform 메소드로부터 단일 레코드가 반환될 수 있으며 ProcessorContext#forward[a]를 사용하여 복수의 값이 방출될 수 있다. 레코드 키, 값, 메타데이터, (주기적 함수 스케줄링에 사용될 수 있는) 프로세서 컨텍스트와 연결된 상태 저장소에 액세스한다.	1:N
transformValues	ValueTransformer	Transform과 비슷하지만 레코드 키에 액세스하지 못하고 ProcessorContext#forward를 사용하여 복수 레코드를 전달할 수 없다 (복수 레코드를 전달하려는 경우 StreamsException 예외를 얻을 것이다). 상태 저장소 작동이 키 기반이기 대문에 상태 저장소에 대한 조회 수행이 필요한 경우 이 연산자는 적합하지 않다. 더구나 출력 레코드는 입력 레코드와 동일한 키를 가질 것이고 키가 수정될 수 없기 때문에 다운스트림 자동 재파티셔닝이 격발되지 않을 것이다. (네트워크 트립 방지에 도움이 될 수 있기 때문에 장점이 있다).	1:1
transformValues	ValueTransformerWithKey	Transform과 비슷하지만 레코드 키는 읽기 전용으로 수정되지 않는다. 또한 ProcessorContext#forward를 사용하여 복수 레코드를 전달할 수 없다 (복수 레코드를 전달하려는 경우 StreamsException 예외를 얻을 것이다).	1:1
flatTransform	Transformer (with an iterable return value)	Transform과 비슷하지만 복수 레코드를 반환하기 위해 ProcessorContext#forward에 의존하는 대신 단순히 값의 모음을 반환할 수 있다. 이러한 이유로 복수의 레코드를 방출할 필요가 있는 경우 transform보다는 flatTransform 사용이 권고된다. 왜냐하면 타입 안정성이 없는 transform (ProcessorContext#forward에 의존하기 때문)과는 달리 타입 안정성이 있기 때문이다.	1:N
flatTransformValues	ValueTransformer (with an iterable return value)	Transformer를 각 레코드에 적용하며 ValueTransformer#transform 메소드로부터 직접 하나 이상의 출력 레코드를 반환한다.	1:N
flatTransformValues	ValueTransformerWithKey (with an iterable return value)	읽기 전용 키가 transform 메소드에 전달되는 flatTransformValues의 스테이트풀 변형으로 상태 조회에 사용될 수 있다. ValueTransformerWithKey#transform 메소드로부터 하나 이상의 출력 레코드가 직접 반환된다.	1:N

[a] 기술적으로 1:N 변환이 지원되지만 transform은 단일 레코드가 직접 반환되는 1:1 또는 1:0 변환에 더욱 좋다. 왜냐하면 ProcessorContext#forward가 타입 안정적이지 않기 때문이다. 따라서

transform으로부터 복수의 레코드를 전달하고자 한다면 타입 안정적인 flatTransform이 권고된다.

어떤 변형을 선택하든 연산자가 스테이프풀이라면 새로운 연산자를 추가하기 전에 토폴로지 빌더에 상태 저장소에 연결할 필요가 있다. 스테이트풀 Digital Twin Processor 단계를 리팩토링했기 때문에 다음을 수행해보자:

```
StoreBuilder<KeyValueStore<String, DigitalTwin>> storeBuilder =
 Stores.keyValueStoreBuilder(
   Stores.persistentKeyValueStore("digital-twin-store"),
   Serdes.String(),
   JsonSerdes.DigitalTwin());
builder.addStateStore(storeBuilder); ①
```

① 예제 7-4에서 상태 저장소에 연결되어야 하는 프로세서 이름을 지정하는 Topology#addStateStore 메소드의 선택적인 두번째 파라미터를 논의하였다. 여기서는 이를 생략하였으며 따라서 이 상태 저장소는 dangling 되어 있다 (다음 코드 블록에서 이를 연결할 것이지만).

이제 결정을 해야 한다. Digital Twin Processor 단계 리팩토링에 프로세서를 사용할지 트랜스포머를 사용할지? 이전 테이블의 정의를 살펴보면 (예제 7-5의) 앱의 순수한 프로세서 API 버전에서 이미 Processor 인터페이스를 구현했기 대문에 process 연산자를 사용하고 싶은 유혹이 생길 수 있다. 이 방법을 사용하는 경우(문제의 소지가 있는데 이유에 대해서는 짧게 논의할 것이다) 다음과 같이 구현할 수 있다:

```
highWinds.process(
 DigitalTwinProcessor::new, ①
 "digital-twin-store"); ②
```

① DigitalTwinProcessor의 인스턴스를 검색하는데 사용하는 ProcessSupplier

② 프로세서가 상호작용할 상태 저장소의 이름

불행히도 이는 노드에 싱크 프로세서를 연결해야 하고 process 연산자가 터미널 동작이기 때문에 이상적이지 않다. 대신 트랜스포머 연산자 중 하나가 더욱 나을 것이다. 왜냐하면 쉽게 싱크 프로세서에 연결할 수 있기 때문이다. 이제 표 7-4를 살펴보고 우리의 요건을 충족시키는 연산자를 찾아보자:

- 각 입력 레코드는 항상 하나의 출력 레코드를 산출할 것이다 (1:1 매핑)
- 상태 저장소에서 포인트 조회를 수행하고 있고 어떤 식이든 키를 수정할 필요가 없기 때문에 레코드 키에 대해 읽기 전용 액세스가 필요하다.

이러한 요건에 가장 잘 맞는 연산자는 ValueTransformerWithKey를 사용하는 변형인 transformValues이다. 이미 프로세서를 사용하여 이 단계의 계산 로직을 구현하였고 따라서 valueTransformerWithKey 인터페이스를 구현하여 다음과 같이 예제 7-5의 process 메소드에서 transform 메소드로 로직을 복사하기만 하면 된다. 코드 대부분은 프로세서 구현과 동일하기 때문에 생략되었다. 이 예 다음의 주석에

서 변경을 강조 표시하였다:

```
public class DigitalTwinValueTransformerWithKey
 implements ValueTransformerWithKey<String, TurbineState, DigitalTwin> { ①
 @Override
 public void init(ProcessorContext context) {
 // ...
 }
 @Override
 public DigitalTwin transform(String key, TurbineState value) {
 // ...
 return digitalTwin; ②
 }
 @Override
 public void close() {
 // ...
 }
 public void enforceTtl(Long timestamp) {
 // ...
 }
}
```

① ValueTransformerWithKey 인터페이스를 구현한다. String은 키 타입, TurbineState은 입력 레코드의 값 타입 그리고 DigitalTwin은 출력 레코드의 값 타입과 관련이 있다.

② 레코드를 다운스트림 프로세서로 보내기 위해 context.forward 메소드를 사용하는 대신 transform 메소드로부터 직접 레코드를 반환할 수 있다. 볼 수 있듯이 이는 이미 보다 더 DSL과 같은 느낌이다.

트랜스포머 구현 완료 시 다음 라인을 애플리케이션에 추가할 수 있다:

```
highWinds
 .transformValues(DigitalTwinValueTransformerWithKey::new, "digital-twin-store")
 .to("digital-twins", Produced.with(Serdes.String(), JsonSerdes.DigitalTwin()));
```

함께 모으기: 리팩터

이제 DSL 리팩토링의 각 단계를 논의하였고 표 7-5에서 보듯이 애플리케이션의 2 가지 구현을 하나씩 하나씩 살펴보자.

표 7-5. 디지털 트윈 토폴로지의 2 가지 다른 구현

프로세서 API 만 사용	DSL + 프로세서 API
Topology builder = new Topology(); builder.addSource("Desired State Events",	StreamsBuilder builder = new StreamsBuilder(); KStream<String, TurbineState> desiredStateEvents = builder.stream("desired-state-events",

```
 Serdes.String().deserializer(),
 JsonSerdes.TurbineState().deserializer(),
 "desired-state-events");
builder.addSource(
 "Reported State Events",
 Serdes.String().deserializer(),
 JsonSerdes.TurbineState().deserializer(),
 "reported-state-events");
builder.addProcessor(
 "High Winds Flatmap Processor",
 HighWindsFlatmapProcessor::new,
 "Reported State Events");
builder.addProcessor(
 "Digital Twin Processor",
 DigitalTwinProcessor::new,
 "High Winds Flatmap Processor",
 "Desired State Events");
StoreBuilder<KeyValueStore<String, DigitalTwin>>
 storeBuilder =
 Stores.keyValueStoreBuilder(
 Stores.persistentKeyValueStore(
 "digital-twin-store"),
 Serdes.String(),
 JsonSerdes.DigitalTwin());
builder.addStateStore(storeBuilder,
 "Digital Twin Processor");
builder.addSink(
 "Digital Twin Sink",
 "digital-twins",
 Serdes.String().serializer(),
 JsonSerdes.DigitalTwin().serializer(),
 "Digital Twin Processor");
```

```
   Consumed.with(
     Serdes.String(),
     JsonSerdes.TurbineState()));
KStream<String, TurbineState> highWinds =
 builder.stream("reported-state-events",
   Consumed.with(
     Serdes.String(),
     JsonSerdes.TurbineState()))
 .flatMapValues((key, reported) -> ... )
 .merge(desiredStateEvents);



// empty space to align next topology step



StoreBuilder<KeyValueStore<String, DigitalTwin>>
 storeBuilder =
   Stores.keyValueStoreBuilder(
     Stores.persistentKeyValueStore(
       "digital-twin-store"),
     Serdes.String(),
     JsonSerdes.DigitalTwin());
builder.addStateStore(storeBuilder);
highWinds
 .transformValues(
   DigitalTwinValueTransformerWithKey::new,
     "digital-twin-store")
 .to("digital-twins",
   Produced.with(
     Serdes.String(),
     JsonSerdes.DigitalTwin()));
```

각각의 구현이 완벽하게 좋다. 그러나 초반에 언급했던 것으로 돌아가서 이유가 없다면 추가적인 복잡성을 도입하길 원하지 않는다.

하이브리드 DSL + 프로세서 API 구현의 장점은 다음과 같다:

- 노드 이름과 부모 이름을 사용하여 프로세서 간 관계를 정의해야 하는 대신 연산자 체이닝을 통해 데이터 흐름의 정신적 지도를 구축하는 것이 더욱 쉽다.
- DLS의 경우 대부분 연산자에 대해 람다식 표현을 지원하며 명료한 변환에 이로울 수 있다 (프로세서 API는 간단한 연산에 대해서도 Processor 인터페이스 구현을 필요로 하며 지루할 수

있다).

- 이 튜토리얼에서는 어떤 레코드도 키재생성할 필요가 없었지만 프로세서 API로 이를 수행하는 방법은 보다 더 지루한 작업이다. 간단한 키재생성 연산에 대해 Processor 인터페이스를 구현해야 할 뿐만 아니라 중간 리파티션 토픽에 대한 rewrite도 다루어야 한다 (이는 명시적으로 추가적인 싱크 및 소스 프로세서 추가를 포함하며 불필요하게 복잡한 코드를 야기할 수 있다).

- DSL 연산자는 스트림 처리 단계에서 무엇이 발생할지를 정의하기 위한 표준 어휘를 제공한다. 예를 들어 계산 로직에 대해 어떤 것도 모르고서 flatMap 연산자가 입력과 다른 수의 레코드를 산출할 수 있음을 추론할 수 있다. 반면 프로세서 API는 주어진 Processor 구현의 특성 위장을 손쉽게 하며 이는 코드 가독성을 떨어뜨리고 유지보수 측면에서 좋지 않은 영향을 미칠 수 있다.

- DSL은 또한 다른 유형의 스트림에 대해 공통적인 어휘를 제공한다. 이는 순수한 레코드 스트림, 로컬 집계 스트림 (항상 테이블과 관련이 있음)과 글로벌 집계 스트림 (글로벌 테이블과 관련이 있음)을 포함한다.

따라서, 애플리케이션을 프로세서 API만으로 구현하는 대신 저수준 액세스 필요시 프로세서 API를 사용하는 DSL의 일련의 특수 연산자를 활용하는 것을 권고된다.

요약

이 장에서는 카프카 레코드와 카프카 스트림즈의 프로세서 컨텍스트에 대한 저수준 액세스를 얻기 위해 프로세서 API 사용법을 배웠다. 또한 주기 함수를 스케줄링할 수 있도록 하는 프로세서 API의 유용한 기능과 스케줄링된 함수에 대해 punctuator을 정의 시 사용될 수 있는 다양한 시간 개념을 논의하였다. 마지막으로 프로세서 API와 고수준 DSL을 결합하는 것이 두 API 모두의 장점을 활용하는 좋은 방식임을 보였다. 다음 장에서는 단순성 측면에서 스펙트럼의 반대 쪽으로 데려갈 ksqlDB 논의를 시작할 것이다 (이는 이 책에서 논의할 스트림 처리 애플리케이션을 구축하는데 있어 가장 간단한 옵션, 아니 틀림없이 가장 간단한 옵션이다).