



개발자 가이드

# Amazon Kinesis Data Streams



# Amazon Kinesis Data Streams: 개발자 가이드

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon의 상표 및 브랜드 디자인은 Amazon 외 제품 또는 서비스와 함께, 브랜드 이미지를 떨어뜨리거나 고객에게 혼동을 일으킬 수 있는 방식으로 사용할 수 없습니다. Amazon이 소유하지 않은 기타 모든 상표는 과 제휴 관계이거나 관련이 있거나 후원 관계와 관계없이 해당 소유자의 자산입니다.

# Table of Contents

Amazon Kinesis Data Streams란? .....	1
Kinesis Data Streams로 무엇을 할 수 있나요? .....	1
Kinesis Data Streams 사용의 이점 .....	2
관련 서비스 .....	3
용어 및 개념 .....	4
상위 수준 아키텍처 .....	4
Kinesis Data Streams 용어 .....	4
Kinesis 데이터 스트림 .....	4
데이터 레코드 .....	5
용량 모드 .....	5
보존 기간 .....	5
생산자 .....	5
소비자 .....	5
Amazon Kinesis Data Streams 애플리케이션 .....	6
샤드 .....	6
파티션 키 .....	6
시퀀스 번호 .....	6
Kinesis Client Library .....	7
애플리케이션 이름 .....	7
서버 측 암호화 .....	7
할당량과 제한 .....	9
API 제한 .....	10
KDS 제어 플레인 API 제한 .....	10
KDS 데이터 플레인 API 제한 .....	14
할당량 증가 .....	16
설정 .....	17
가입하기 AWS .....	17
라이브러리 및 도구 다운로드 .....	17
개발 환경 구성 .....	18
시작하기 .....	19
AWS CLI 설치 및 구성 .....	19
AWS CLI 설치 .....	19
AWS CLI 구성 .....	21
AWS CLI를 사용하여 기본 Kinesis 데이터 스트림 작업 수행 .....	21

단계 1: VPC 생성 .....	21
2단계: 레코드 넣기 .....	23
3단계: 레코드 가져오기 .....	23
4단계: 정리 .....	27
예시 .....	28
자습서: KPL 및 KCL 2.x를 사용하여 실시간 주식 데이터 처리 .....	28
필수 조건 .....	29
1단계: 데이터 스트림 생성 .....	30
2단계: IAM 정책 및 사용자 생성 .....	31
3단계: 코드 다운로드 및 빌드 .....	36
단계 4: 생산자 구현 .....	37
단계 5: 소비자 구현 .....	41
단계 6: (선택 사항) 소비자 확대 .....	45
단계 7: 마무리 .....	47
자습서: KPL 및 KCL 1.x를 사용하여 실시간 주식 데이터 처리 .....	48
필수 조건 .....	49
1단계: 데이터 스트림 생성 .....	50
2단계: IAM 정책 및 사용자 생성 .....	52
3단계: 구현 코드 다운로드 및 빌드 .....	57
단계 4: 생산자 구현 .....	58
단계 5: 소비자 구현 .....	62
단계 6: (선택 사항) 소비자 확대 .....	65
단계 7: 마무리 .....	67
자습서: Flink 애플리케이션용 Managed Service for Apache Flink를 사용하여 실시간 주식 데이터 분석 .....	68
필수 조건 .....	69
1단계: 계정 설정 .....	70
2단계: 설정 AWS CLI .....	73
3단계: 애플리케이션 생성 .....	74
자습서: Amazon Kinesis Data Streams에서 AWS Lambda 사용 .....	91
AWS Streaming Data Solution .....	91
스트림 생성 및 관리 .....	93
데이터 스트림 용량 모드 선택 .....	93
데이터 스트림 용량 모드란? .....	94
온디맨드 모드 .....	94
프로비저닝된 모드 .....	96

용량 모드 간 전환 .....	97
관리 콘솔을 통한 스트림 생성 AWS .....	97
API를 통해 스트림 생성 .....	98
Kinesis Data Streams 클라이언트 구축 .....	98
스트림 생성 .....	99
스트림 업데이트 .....	100
.....	100
API를 사용하여 스트림 업데이트 .....	101
를 사용하여 스트림 업데이트 AWS CLI .....	102
스트림 나열 .....	102
샤드 나열 .....	103
ListShards API - 권장 .....	103
DescribeStream API - 더 이상 사용되지 않음 .....	106
스트림 삭제 .....	107
스트림 리샤딩 .....	108
리샤딩 전략 .....	109
샤드 분할 .....	109
두 개의 샤드 병합 .....	111
리샤딩 이후 .....	112
데이터 보존 기간 변경 .....	114
스트림 태그 지정 .....	115
태그 기본 사항 .....	116
태그 지정을 사용하여 비용 추적 .....	116
태그 제한 .....	116
Kinesis Data Streams 콘솔을 사용하여 스트림 태그 지정 .....	117
AWS CLI를 사용하여 스트림에 태깅 .....	118
Kinesis Data Streams API를 사용하여 스트림 태그 지정 .....	118
데이터 스트림에 쓰기 .....	120
KPL 사용 .....	120
KPL의 역할 .....	121
KPL 사용의 장점 .....	122
KPL을 사용하지 말아야 할 경우 .....	123
KPL 설치 .....	123
KPL(Kinesis Producer Library)의 Amazon Trust Services(ATS) 인증서로 전환 .....	124
KPL 지원 플랫폼 .....	124
KPL 주요 개념 .....	125

KPL과 생산자 코드의 통합 .....	127
Kinesis 데이터 스트림에 쓰기 .....	129
KPL 구성 .....	130
소비자 분해 .....	131
Firehose와 함께 KPL 사용하기 .....	134
AWS Glue 스키마 레지스트리와 함께 KPL 사용 .....	135
KPL 프록시 구성 .....	135
API 사용 .....	136
스트림에 데이터 추가 .....	137
AWS Glue 스키마 레지스트리를 사용하여 데이터와 상호 작용 .....	143
에이전트 사용 .....	143
필수 조건 .....	144
에이전트 다운로드 및 설치 .....	144
에이전트 구성 및 시작 .....	145
에이전트 구성 설정 .....	146
여러 파일 디렉터리 모니터링 및 여러 스트림에 쓰기 .....	149
에이전트를 사용하여 데이터 사전 처리 .....	150
에이전트 CLI 명령 .....	154
FAQ .....	155
다른 AWS 서비스 사용 .....	156
AWS Amplify .....	157
Amazon Aurora .....	157
Amazon CloudFront .....	157
Amazon CloudWatch Logs .....	157
Amazon Connect .....	158
AWS Database Migration Service .....	158
Amazon DynamoDB .....	158
Amazon EventBridge .....	159
AWS IoT Core .....	159
Amazon Relational Database Service .....	159
Amazon Pinpoint .....	159
Amazon Quantum Ledger Database .....	159
타사 통합 사용 .....	160
Apache Flink .....	160
Fluentd .....	160
Debezium .....	160

Oracle GoldenGate .....	161
Kafka 연결 .....	161
Adobe Experience .....	161
Striim .....	161
문제 해결 .....	161
생산자 애플리케이션이 예상보다 느린 속도로 쓰고 있습니다. ....	161
무단 KMS 마스터 키 권한 오류 .....	163
생산자를 위한 일반적인 문제, 질문 및 문제 해결 아이디어 .....	163
고급 주제 .....	164
재시도 및 속도 제한 .....	164
KPL 집계 사용 시 고려 사항 .....	165
데이터 스트림에서 읽기 .....	166
Kinesis 콘솔에서 데이터 뷰어 사용 .....	167
Kinesis 콘솔에서 데이터 스트림을 쿼리하기 .....	168
사용 AWS Lambda .....	169
Managed Service for Apache Flink 사용 .....	169
Firehose 사용 .....	169
Kinesis Client Library 사용 .....	170
Kinesis Client Library란? .....	170
KCL 사용 가능 버전 .....	171
KCL 개념 .....	172
리스 테이블을 사용하여 KCL 소비자 애플리케이션에서 처리한 샤드 추적 .....	174
동일한 'KCL 2.x for Java' 소비자 애플리케이션으로 여러 데이터 스트림 처리 .....	183
Kinesis 클라이언트 라이브러리를 AWS Glue 스키마 레지스트리와 함께 사용 .....	186
공유 처리량으로 사용자 지정 소비자 개발 .....	187
KCL을 사용하여 공유 처리량으로 사용자 지정 소비자 개발 .....	187
AWS SDK for Java를 사용하여 공유 처리량으로 사용자 지정 소비자 개발 .....	223
전용 처리량으로 사용자 지정 소비자 개발(향상된 팬아웃) .....	229
KCL 2.x를 사용하여 향상된 팬아웃 소비자 개발 .....	231
Kinesis Data Streams API를 사용하여 향상된 팬아웃 소비자 개발 .....	237
AWS Management Console을 사용하여 향상된 팬아웃 소비자 관리 .....	240
KCL 1.x에서 KCL 2.x로 소비자 마이그레이션 .....	241
레코드 프로세서 마이그레이션 .....	242
레코드 프로세서 팩토리 마이그레이션 .....	246
작업자 마이그레이션 .....	248
Amazon Kinesis Client 구성 .....	250

유휴 시간 제거 .....	254
클라이언트 구성 제거 .....	255
다른 AWS 서비스 사용 .....	256
Amazon EMR 사용 .....	256
아마존 EventBridge 파이프 사용 .....	256
AWS Glue 사용 .....	257
Amazon Redshift 사용 .....	257
타사 통합 사용 .....	257
Apache Flink .....	258
Adobe Experience Platform .....	258
Apache Druid .....	258
Apache Spark .....	258
Databricks .....	258
Kafka Confluent 플랫폼 .....	259
Kinesumer .....	259
Talend .....	259
Kinesis 데이터 스트림 소비자 문제 해결 .....	259
Kinesis Client Library를 사용할 때 일부 Kinesis Data Streams 레코드를 건너뛰는 경우 .....	259
같은 샤드에 속한 레코드가 동시에 여러 레코드 프로세서에 의해 처리되는 경우 .....	260
소비자 애플리케이션이 예상보다 느린 속도로 읽는 경우 .....	260
GetRecords 스트림에 데이터가 있는 경우에도 빈 레코드 배열을 반환합니다. ....	261
샤드 반복자가 예기치 않게 만료되는 경우 .....	262
소비자 레코드 처리 속도가 느려지는 경우 .....	262
무단 KMS 마스터 키 권한 오류 .....	263
소비자를 위한 일반적인 문제, 질문 및 문제 해결 아이디어 .....	263
고급 주제 .....	263
짧은 처리 지연 시간 .....	264
Kinesis 프로듀서 AWS Lambda 라이브러리와 함께 사용 .....	265
리샤딩, 크기 조정 및 병렬 처리 .....	265
중복 레코드 처리 .....	266
시작, 종료 및 조절 처리 .....	269
데이터 스트림 모니터링 .....	271
를 사용하여 서비스를 모니터링합니다. CloudWatch .....	271
Amazon Kinesis Data Streams 측정기준 및 지표 .....	272
Kinesis Data Streams용 Amazon CloudWatch 메트릭스에 액세스 .....	285
다음을 사용하여 에이전트를 모니터링합니다. CloudWatch .....	286



를 통한 모니터링 CloudWatch .....	287
AWS CloudTrail을 사용하여 Amazon Kinesis Data Streams API 호출 로깅 .....	287
Kinesis Data Streams 정보 입력 CloudTrail .....	288
예: Kinesis Data Streams 로그 파일 항목 .....	289
를 사용하여 KCL 모니터링하기 CloudWatch .....	293
측정치 및 네임스페이스 .....	294
측정치 수준 및 차원 .....	294
측정치 구성 .....	295
측정치 목록 .....	295
다음을 통한 KPL 모니터링 CloudWatch .....	307
측정치, 차원 및 네임스페이스 .....	307
측정치 수준 및 세부 수준 .....	307
로컬 액세스 및 아마존 CloudWatch 업로드 .....	308
측정치 목록 .....	309
보안 .....	313
데이터 보호 .....	313
Kinesis Data Streams용 서버 측 암호화란? .....	314
비용, 리전 및 성능 고려 사항 .....	315
서버 측 암호화를 시작하는 방법 .....	317
사용자 생성 KMS 마스터 키 생성 및 사용 .....	317
사용자 생성 KMS 마스터 키 사용 권한 .....	318
KMS 키 권한 확인 및 문제 해결 .....	320
인터페이스 VPC 종단점 사용 .....	320
액세스 제어 .....	323
정책 구문 .....	325
Kinesis Data Streams에 대한 작업 .....	325
Kinesis Data Streams용 Amazon 리소스 이름(ARN) .....	326
Kinesis Data Streams에 대한 예제 정책 .....	326
다른 계정과 데이터 스트림 공유 .....	329
AWS Lambda 함수를 다른 계정의 Kinesis Data Streams에서 읽도록 구성 .....	334
리소스 기반 정책을 사용하여 액세스 공유 .....	334
규정 준수 검증 .....	336
복원력 .....	337
재해 복구 .....	337
인프라 보안 .....	338
보안 모범 사례 .....	339

---

최소 권한 액세스 구현 .....	339
IAM 역할 사용 .....	339
종속 리소스에서 서버 측 암호화 구현 .....	339
API 호출을 모니터링하는 CloudTrail 데 사용합니다. ....	340
문서 기록 .....	341
AWS 용어집 .....	344
.....	cccxiv

# Amazon Kinesis Data Streams란?

Amazon Kinesis Data Streams를 사용하여 대규모 데이터 레코드 [스트림](#)을 실시간으로 수집하고 처리할 수 있습니다. Kinesis Data Streams 애플리케이션이라는 데이터 처리 애플리케이션을 생성할 수 있습니다. 일반적인 Kinesis Data Streams 애플리케이션은 데이터가 기록될 때 데이터 스트림에서 데이터를 읽습니다. 이러한 애플리케이션은 Kinesis Client Library를 사용하며 Amazon EC2 인스턴스에서 실행될 수 있습니다. 처리된 레코드를 대시보드로 보내거나, 알림을 생성하는 데 사용하거나, 요금 및 광고 전략을 동적으로 변경하거나, 다른 여러 AWS 서비스에 데이터를 보낼 수 있습니다. Kinesis Data Streams 기능 및 요금에 대한 자세한 내용은 [Amazon Kinesis Data Streams](#)를 참조하세요.

[Kinesis Data Streams는 Firehose, Kinesis Video Streams, 아파치 플링크용 매니지드 서비스와 함께 Kinesis 스트리밍 데이터 플랫폼의 일부입니다.](#)

[빅데이터 솔루션에 대한 자세한 내용은 AWS 빅 데이터 온을 참조하십시오.](#) [AWS](#) AWS 스트리밍 데이터 솔루션에 대한 자세한 내용은 [스트리밍 데이터란 무엇입니까?](#)를 참조하세요.

## 주제

- [Kinesis Data Streams로 무엇을 할 수 있나요?](#)
- [Kinesis Data Streams 사용의 이점](#)
- [관련 서비스](#)

## Kinesis Data Streams로 무엇을 할 수 있나요?

신속하고 지속적인 데이터 인테이크 및 집계를 위해 Amazon Kinesis Data Streams를 사용할 수 있습니다. 사용되는 데이터 유형으로는 IT 인프라 로그 데이터, 애플리케이션 로그, 소셜 미디어, 시장 데이터 피드, 웹 클릭스트림 데이터 등이 있습니다. 데이터 인테이크 및 처리에 대한 응답이 실시간으로 이루어지기 때문에 간단하게 처리되는 것이 일반적입니다.

다음은 일반적인 Kinesis Data Streams 사용 시나리오입니다.

### 가속화된 로그 및 데이터 피드 인테이크 및 처리

생산자를 통해 스트림으로 직접 데이터를 푸시할 수 있습니다. 예를 들어, 시스템 및 애플리케이션 로그를 푸시하고 몇 초 만에 처리할 수 있습니다. 이렇게 하면 프론트엔드 또는 애플리케이션 서버에 장애가 발생할 경우 로그 데이터가 손실되는 것을 방지할 수 있습니다. Kinesis Data Streams는 인테이크를 위해 데이터를 제출하기 전에 서버에서 데이터를 일괄 처리하지 않기 때문에 가속화된 데이터 피드 인테이크를 제공합니다.

## 실시간 측정치 및 보고

Kinesis Data Streams에 수집된 데이터를 간단한 데이터 분석 및 실시간 보고에 사용할 수 있습니다. 예를 들어, 데이터가 스트레밍되는 동안 데이터 처리 애플리케이션이 데이터 배치를 수신할 때까지 기다리는 대신 측정치를 내고 시스템 및 애플리케이션 로그를 보고할 수 있습니다.

## 실시간 데이터 분석

실시간 데이터의 가치와 병렬 처리 능력이 함께 발휘됩니다. 예를 들어, 웹 사이트 클릭 스트림을 실시간으로 처리한 다음 병렬로 실행되는 여러 개의 다른 Kinesis Data Streams 애플리케이션을 사용하여 사이트 가용성 참여를 분석합니다.

## 복잡한 스트림 처리

Kinesis Data Streams 애플리케이션 및 데이터 스트림의 DAG(방향성 비순환 그래프)를 생성할 수 있습니다. 이렇게 하려면 일반적으로 여러 Kinesis Data Streams 애플리케이션의 데이터를 또 다른 스트림에 넣어 다른 Kinesis Data Streams 애플리케이션에서 다운스트림을 처리합니다.

# Kinesis Data Streams 사용의 이점

Kinesis Data Streams를 사용하여 여러 가지 스트리밍 데이터 문제를 해결할 수 있지만 데이터를 실시간으로 집계한 후 집계 데이터를 데이터 웨어하우스나 map-reduce 클러스터에 로드하는 것이 일반적입니다.

데이터가 Kinesis 데이터 스트림으로 들어가므로 지속성과 탄력성이 보장됩니다. 레코드가 스트림에 추가되는 시간과 레코드가 검색될 수 있는 시간 (put-to-get 지연) 사이의 지연 시간은 일반적으로 1초 미만입니다. 즉, 데이터가 추가되고 거의 직후에 Kinesis Data Streams 애플리케이션이 스트림의 데이터를 소비할 수 있습니다. Kinesis Data Streams는 관리형 서비스이므로 데이터 인테이크 파이프라인을 생성하고 실행하는 작업 부담이 줄어듭니다. 스트리밍 map-reduce 유형 애플리케이션을 생성할 수 있습니다. Kinesis Data Streams의 탄력성으로 인해 스트림을 확장하거나 축소할 수 있어 만료 전에 데이터 레코드가 손실되지 않습니다.

여러 Kinesis Data Streams 애플리케이션이 스트림의 데이터를 소비할 수 있어 보관, 처리와 같은 여러 가지 작업이 동시에 개별적으로 이루어질 수 있습니다. 예를 들어, 두 애플리케이션이 같은 스트림에서 데이터를 읽을 수 있습니다. 첫 번째 애플리케이션은 실행 중인 집계를 계산하고 Amazon DynamoDB 테이블을 업데이트하며, 두 번째 애플리케이션은 데이터를 압축하여 Amazon Simple Storage Service(S3)와 같은 데이터 스토어에 보관합니다. 그러면 대시보드에서 실행 집계가 있는 DynamoDB 테이블을 읽고 보고서를 작성합니다. up-to-the-minute

Kinesis Data Streams를 사용하면 결함이 있더라도 정상적으로 스트림의 데이터를 소비할 수 있으며 Kinesis Data Streams 애플리케이션의 규모를 조정할 수 있습니다.

## 관련 서비스

Amazon EMR 클러스터를 사용하여 Kinesis 데이터 스트림을 직접 읽고 처리하는 방법에 대한 자세한 내용은 [Kinesis 커넥터](#)를 참조하세요.

# Amazon Kinesis Data Streams 용어 및 개념

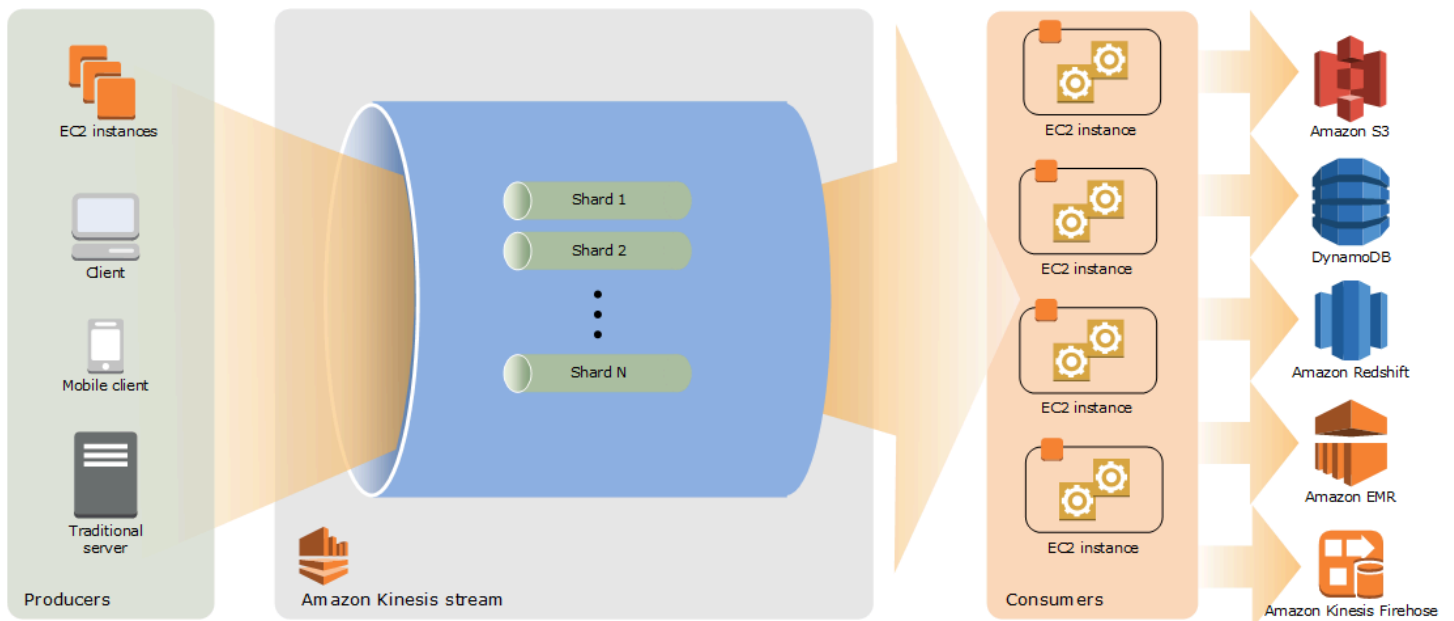
Amazon Kinesis Data Streams를 시작할 때 아키텍처와 용어를 이해하면 도움이 될 수 있습니다.

주제

- [Kinesis Data Streams 상위 수준 아키텍처](#)
- [Kinesis Data Streams 용어](#)

## Kinesis Data Streams 상위 수준 아키텍처

다음 다이어그램은 Kinesis Data Streams의 상위 수준 아키텍처를 보여줍니다. 생산자가 계속해서 Kinesis Data Streams에 데이터를 푸시하고 소비자가 실시간으로 데이터를 처리합니다. 소비자 (예: Amazon EC2에서 실행되는 사용자 지정 애플리케이션 또는 Amazon Data Firehose 전송 스트림) 는 Amazon DynamoDB, Amazon Redshift 또는 Amazon S3와 같은 AWS 서비스를 사용하여 결과를 저장할 수 있습니다.



## Kinesis Data Streams 용어

### Kinesis 데이터 스트림

Kinesis 데이터 스트림은 [샤드](#)의 집합입니다. 샤드마다 데이터 레코드 시퀀스가 있습니다. 각 데이터 레코드에는 Kinesis Data Streams에서 할당한 [시퀀스 번호](#)가 있습니다.

## 데이터 레코드

데이터 레코드는 [Kinesis 데이터 스트림](#)에 저장되는 데이터의 단위입니다. 데이터 레코드는 [시퀀스 번호](#), [파티션 키](#) 및 변경할 수 없는 바이트 시퀀스인 데이터 blob으로 구성됩니다. Kinesis Data Streams는 어떤 방식으로든 blob의 데이터를 검사, 해석 또는 변경하지 않습니다. 데이터 blob은 최대 1MB일 수 있습니다.

## 용량 모드

데이터 스트림 용량 모드는 용량이 관리되는 방식과 데이터 스트림 사용에 대한 요금이 청구되는 방식을 결정합니다. 현재 Kinesis Data Streams에서는 데이터 스트림에 대해 온디맨드 모드와 프로비저닝 모드 중에서 선택할 수 있습니다. 자세한 설명은 [데이터 스트림 용량 모드 선택](#) 섹션을 참조하세요.

온디맨드 모드에서 Kinesis Data Streams는 필요한 처리량을 제공하기 위해 샤드를 자동으로 관리합니다. 사용한 실제 처리량에 대해서만 요금이 부과되며 Kinesis Data Streams는 증가하거나 감소할 때 워크로드의 처리량 요구 사항을 자동으로 수용합니다. 자세한 설명은 [온디맨드 모드](#) 섹션을 참조하세요.

프로비저닝된 모드에서는 데이터 스트림의 샤드 수를 지정해야 합니다. 데이터 스트림의 총 용량은 해당 샤드 용량의 합계입니다. 필요에 따라 데이터 스트림의 샤드 수를 늘리거나 줄일 수 있으며 샤드 수에 대해 시간당 요금이 부과됩니다. 자세한 설명은 [프로비저닝된 모드](#) 섹션을 참조하세요.

## 보존 기간

보존 기간은 데이터 레코드를 스트림에 추가한 후 데이터 레코드에 액세스할 수 있는 시간의 길이입니다. 스트림 보존 기간은 기본적으로 생성 후 24시간으로 설정됩니다. 작업을 사용하면 보존 기간을 최대 8760시간(365일)까지 늘리고, [IncreaseStreamRetentionPeriod](#) 작업을 사용하면 보존 기간을 최소 24시간으로 줄일 수 있습니다. [DecreaseStreamRetentionPeriod](#) 24시간을 초과하여 보존 기간을 설정하면 스트림에 추가 요금이 적용됩니다. 자세한 내용은 [Amazon Kinesis Data Streams 요금](#)을 참조하십시오.

## 생산자

생산자는 레코드를 Amazon Kinesis Data Streams에 저장합니다. 예를 들어, 스트림에 로그 데이터를 보내는 웹 서버가 생산자입니다.

## 소비자

소비자는 Amazon Kinesis Data Streams의 레코드를 가져와서 처리합니다. 이 소비자를 [Amazon Kinesis Data Streams 애플리케이션](#)이라고 합니다.

## Amazon Kinesis Data Streams 애플리케이션

Amazon Kinesis Data Streams 애플리케이션은 EC2 인스턴스의 플릿에서 공통적으로 실행되는 스트림의 소비자입니다.

공유 팬아웃 소비자와 향상된 팬아웃 소비자의 두 가지 유형 소비자를 개발할 수 있습니다. 두 유형 간의 차이와 각 유형의 소비자를 만드는 방법을 살펴보려면 [Amazon Kinesis Data Streams에서 데이터 읽기](#) 단원을 참조하십시오.

Kinesis Data Streams 애플리케이션의 출력은 다른 스트림의 입력이 될 수 있으며 실시간으로 데이터를 처리하는 복잡한 토폴로지를 생성할 수 있도록 해줍니다. 또한 애플리케이션은 다양한 다른 AWS 서비스에 데이터를 보낼 수 있습니다. 스트림 하나에 여러 애플리케이션이 있을 수 있으며 각 애플리케이션이 동시에 독립적으로 스트림의 데이터를 소비할 수 있습니다.

### 샤드

샤드는 스트림에서 고유하게 식별되는 데이터 레코드 시퀀스입니다. 스트림은 하나 이상의 샤드로 구성되며 각 샤드는 고정된 용량 단위를 제공합니다. 각 샤드는 읽기의 경우 초당 최대 5개의 트랜잭션, 초당 최대 2MB의 총 데이터 읽기 속도, 쓰기의 경우 초당 최대 1,000개의 레코드, 초당 최대 1MB의 총 데이터 쓰기 속도를 지원할 수 있습니다(파티션 키 포함). 스트림의 데이터 용량은 스트림에 지정하는 샤드 수의 함수입니다. 스트림의 총 용량은 해당 샤드의 용량의 합계입니다.

데이터 속도가 증가하면 스트림에 할당된 샤드 수를 늘리거나 줄일 수 있습니다. 자세한 설명은 [스트림 리샤딩](#) 섹션을 참조하세요.

### 파티션 키

파티션 키는 스트림 내의 샤드에서 데이터를 그룹화하는 데 사용됩니다. Kinesis Data Streams는 스트림에 속하는 데이터 레코드를 여러 샤드로 분리합니다. 각 데이터 레코드와 연결된 파티션 키를 사용하여 해당 데이터 레코드가 속한 샤드를 확인합니다. 파티션 키는 각 키에 대한 최대 길이 제한이 256자인 유니코드 문자열입니다. 파티션 키를 128비트 정수 값에 매핑하고 샤드의 해시 키 범위를 사용하여 연결된 데이터 레코드를 샤드에 매핑하기 위해 MD5 해시 함수가 사용됩니다. 애플리케이션이 데이터를 스트림에 넣을 때는 파티션 키를 지정해야 합니다.

### 시퀀스 번호

각 데이터 레코드에는 샤드 내 파티션 키별로 고유한 시퀀스 번호가 있습니다. Kinesis Data Streams는 `client.putRecords` 또는 `client.putRecord`를 사용하여 스트림에 쓴 후 시퀀스 번호를 할당합



니다. 같은 파티션 키의 시퀀스 번호는 일반적으로 시간이 지나면서 증가합니다. 쓰기 요청 간의 기간이 길수록 시퀀스 번호가 커집니다.

### Note

같은 스트림에 있는 데이터 세트의 인덱스로 시퀀스 번호를 사용할 수 없습니다. 데이터 세트를 논리적으로 분리하려면 파티션 키를 사용하거나 데이터 세트마다 별도의 스트림을 만드십시오.

## Kinesis Client Library

Kinesis Client Library는 결함이 있어도 정상적으로 스트림의 데이터를 소비할 수 있도록 애플리케이션에 컴파일됩니다. Kinesis Client Library는 각 샤드에 대해 해당 샤드를 실행하고 처리하는 레코드 프로세서가 있도록 보장합니다. 또한 라이브러리는 스트림에서 데이터를 읽는 과정을 간소화합니다. Kinesis Client Library는 Amazon DynamoDB 테이블을 사용하여 제어 데이터를 저장합니다. 데이터를 처리하는 애플리케이션마다 테이블 하나를 만듭니다.

Kinesis Client Library에는 두 가지 주요 버전이 있습니다. 사용하는 버전은 만들려는 소비자 유형에 따라 결정됩니다. 자세한 설명은 [Amazon Kinesis Data Streams에서 데이터 읽기](#) 섹션을 참조하세요.

## 애플리케이션 이름

Amazon Kinesis Data Streams 애플리케이션의 이름은 애플리케이션을 식별합니다. 각 애플리케이션에는 애플리케이션에서 사용하는 AWS 계정 및 지역으로 범위가 지정된 고유한 이름이 있어야 합니다. 이 이름은 Amazon DynamoDB의 제어 테이블 이름과 Amazon 지표의 네임스페이스 이름으로 사용됩니다. CloudWatch

## 서버 측 암호화

Amazon Kinesis Data Streams는 생산자가 민감한 데이터를 스트림에 입력할 때 자동으로 암호화할 수 있습니다. Kinesis Data Streams는 암호화에 [AWS KMS](#) 마스터 키를 사용합니다. 자세한 설명은 [Amazon Kinesis Data Streams의 데이터 보호](#) 섹션을 참조하세요.

### Note

암호화된 스트림에서 읽거나 쓰려면 마스터 키에 액세스할 권한이 생산자 및 소비자 애플리케이션에 있어야 합니다. 생산자 및 소비자 애플리케이션에 권한을 부여하는 데 대한 자세한 내용은 [the section called “사용자 생성 KMS 마스터 키 사용 권한”](#)를 참조하십시오.

**Note**

서버 측 암호화를 사용하면 () 비용이 발생합니다. AWS Key Management Service AWS KMS 자세한 내용은 [AWS Key Management Service 요금](#)을 참조하세요.

## 할당량과 제한

Amazon Kinesis Data Streams에는 다음과 같은 스트림 및 샤드 할당량 및 제한이 적용됩니다.

할당량	온디맨드 모드	프로비저닝된 모드
데이터 스트림 수	AWS 계정 내 스트림 수에는 상한 할당량이 없습니다. 기본적으로 온디맨드 용량 모드를 사용하면 최대 50개의 데이터 스트림을 생성할 수 있습니다. 이 할당량을 늘려야 하는 경우 <a href="#">지원 티켓</a> 을 제출하세요.	계정 내 프로비저닝된 모드의 스트림 수에는 상한 할당량이 없습니다.
샤드 수	상한이 없습니다. 샤드 수는 모은 데이터의 양과 필요한 처리량 수준에 따라 다릅니다. Kinesis Data Streams는 데이터 볼륨 및 트래픽의 변화에 따라 샤드 수를 자동으로 조정합니다.	상한이 없습니다. 기본 샤드 할당량은 미국 동부 (버지니아 북부), 미국 서부 (오레곤), 유럽 (아일랜드) AWS 지역의 AWS 계정당 샤드 500개입니다. 다른 모든 리전의 AWS 계정당 기본 샤드 할당량은 200개입니다. <a href="#">shards-per-data 스트림 할당량 증가를 요청하려면 할당량 증가 요청을 참조하십시오.</a>
데이터 스트림 처리량	기본적으로 온디맨드 용량 모드로 생성된 새 데이터 스트림의 쓰기 처리량은 4MB/s이고 읽기 처리량은 8MB/s입니다. 트래픽이 증가함에 따라 온디맨드 용량 모드의 데이터 스트림은 최대 200MB/s의 쓰기 및 400MB/s의 읽기 처리량으로 스케일 업됩니다. 쓰기 용량 2GB/s 및 읽기 용량 4GB/s로 늘려야 하는 경우 <a href="#">지원 티켓</a> 을 제출하세요.	상한이 없습니다. 최대 처리량은 스트림에 대해 프로비저닝된 샤드 수에 따라 달라집니다. 각 샤드는 최대 1MB/s 또는 1,000레코드/초의 쓰기 처리량 또는 최대 2MB/s 또는 2,000레코드/초 읽기의 처리량을 지원할 수 있습니다. 수집 용량이 더 필요한 경우 AWS Management Console 또는 API를 사용하여 스트림의 샤드 수를 쉽게 확장

할당량	온디맨드 모드	프로비저닝된 모드
		장할 수 있습니다. <a href="#">UpdateShardCount</a>
데이터 페이로드 크기	base64-encoding 전 레코드의 데이터 페이로드 최대 크기는 1MB입니다.	
GetRecords 트랜잭션 크기	<a href="#">GetRecords</a> 단일 샤드에서 호출당 최대 10MB의 데이터를 검색하고 호출당 최대 10,000개의 레코드를 검색할 수 있습니다. 모든 GetRecords 호출은 1개의 읽기 트랜잭션으로 간주됩니다. 각 샤드는 초당 최대 5개의 읽기 트랜잭션을 지원합니다. 각 읽기 트랜잭션은 최대 10,000개의 레코드를 제공하며, 트랜잭션당 상한 할당량은 10MB입니다.	
샤드당 데이터 읽기 속도	각 샤드는 를 통해 초당 최대 2MB의 총 데이터 읽기 속도를 지원할 수 있습니다. <a href="#">GetRecords</a> GetRecords 호출이 10MB를 반환하면, 다음 5초 안에 이루어지는 호출에서 예외가 발생합니다.	
데이터 스트림당 등록된 소비자 수	각 데이터 스트림에 대해 최대 20개의 등록된 소비자(향상된 팬아웃 제한)를 생성할 수 있습니다.	
프로비저닝된 모드와 온디맨드 모드 간 전환	AWS 계정의 각 데이터 스트림에 대해 24시간 내에 온디맨드 용량 모드와 프로비저닝 용량 모드 사이를 두 번 전환할 수 있습니다.	

## API 제한

대부분의 AWS API와 마찬가지로 Kinesis Data Streams API 작업도 속도 제한이 있습니다. 리전별 계정당 AWS 다음과 같은 제한이 적용됩니다. Kinesis Data Streams API에 대한 자세한 내용은 [Amazon Kinesis API 참조](#)를 확인하세요.

## KDS 제어 플레인 API 제한

다음 단원에서는 KDS 제어 플레인 API에 대한 제한을 설명합니다. KDS 제어 플레인 API를 사용하면 데이터 스트림을 생성하고 관리할 수 있습니다. 이러한 제한은 리전별로 AWS 계정당 적용됩니다.

## 제어 플레인 API 제한

API	API 호출 제한	계정/스트림당	설명
AddTagsToStream	5건의 초당 트랜잭션 (TPS)	계정별	데이터 스트림당 태그 50개
CreateStream	5TPS	계정별	<p>한 계정에서 사용할 수 있는 스트림 수는 상한 할당량이 없습니다. 다음 중 하나를 수행하려고 할 때 CreateStream 을 요청하면 LimitExceededException 이 발생합니다.</p> <ul style="list-style-type: none"> <li>언제든지 CREATING 상태에 5 개 이상의 스트림이 있어야 합니다.</li> <li>계정에 대해 승인된 것보다 더 많은 샤드를 생성합니다.</li> </ul>
DecreaseStreamRetentionPeriod	5TPS	스트림당	데이터 스트림 보존 기간의 최소값은 24시간입니다.
DeleteResourcePolicy	5TPS	계정별	이 한도를 늘려야 하는 경우 <a href="#">지원 티켓</a> 을 제출하세요.
DeleteStream	5TPS	계정별	
DeregisterStreamConsumer	5TPS	스트림당	

API	API 호출 제한	계정/스트림당	설명
DescribeLimits	1TPS	계정별	
DescribeStream	10TPS	계정별	
DescribeStreamConsumer	20TPS	스트림당	
DescribeStreamSummary	20TPS	계정별	
DisableEnhancedMonitoring	5TPS	스트림당	
EnableEnhancedMonitoring	5TPS	스트림당	
GetResourcePolicy	5TPS	계정별	이 한도를 늘려야 하는 경우 <a href="#">지원 티켓</a> 을 제출하세요.
IncreaseStreamRetentionPeriod	5TPS	스트림당	스트림 보존 기간의 최대값은 8,760시간(365일)입니다.
ListShards	1000TPS	스트림당	
ListStreamConsumers	5TPS	스트림당	
ListStreams	5TPS	계정별	
ListTagsForStream	5TPS	스트림당	
MergeShards	5TPS	스트림당	프로비저닝됨에만 적용됩니다.
PutResourcePolicy	5TPS	계정별	이 한도를 늘려야 하는 경우 <a href="#">지원 티켓</a> 을 제출하세요.

API	API 호출 제한	계정/스트림당	설명
RegisterStreamConsumer	5TPS	스트림당	데이터 스트림당 최대 20명의 소비자를 등록할 수 있습니다. 해당 소비자는 한 번에 하나의 데이터 스트림에 등록될 수 있습니다. 동시에 5명의 소비자를 생성할 수 있습니다. 즉, 동시에 CREATING 상태의 소비자가 5명 이상 있을 수 없습니다. CREATING 상태의 소비자가 5개인데 6번째 소비자를 등록합니다.
RemoveTagsFromStream	5TPS	스트림당	
SplitShard	5TPS	스트림당	프로비저닝됨에만 적용됩니다.
StartStreamEncryption		스트림당	연속 24시간 동안 서버 측 암호화에 새 AWS KMS 키를 25회 성공적으로 적용할 수 있습니다.
StopStreamEncryption		스트림당	24시간 동안 서버 쪽 암호화를 25회 성공적으로 비활성화할 수 있습니다.

API	API 호출 제한	계정/스트림당	설명
UpdateShardCount		스트림당	프로비저닝됨에만 적용됩니다. 샤드 수의 기본 제한은 1만 개입니다. 이 API에는 추가 제한이 있습니다. 자세한 내용은 <a href="#">UpdateShardCount</a> 을 참조하십시오.
UpdateStreamMode		스트림당	AWS 계정의 각 데이터 스트림에 대해 24 시간 내에 온디맨드 및 프로비저닝 용량 모드 사이를 두 번 전환할 수 있습니다.

## KDS 데이터 플레인 API 제한

다음 단원에서는 KDS 데이터 플레인 API에 대한 제한에 대해 설명합니다. KDS 데이터 플레인 API를 사용하면 데이터 스트림을 사용하여 데이터 레코드를 실시간으로 수집하고 처리할 수 있습니다. 이러한 제한은 데이터 스트림 내의 샤드당 적용됩니다.

### 데이터 영역 API 한도

API	API 호출 제한	페이로드 제한	추가 세부 정보
GetRecords	5TPS	호출당 반환할 수 있는 최대 레코드 수는 10,000개입니다. GetRecords 가 반환할 수 있는 최대 데이터 크기는 10MB입니다.	호출이 이 양의 데이터가 반환하면, 다음 5초 안에 이루어지는 호출에서 ProvisionedThroughputExceededException 이 발생합니다. 스트림



API	API 호출 제한	페이로드 제한	추가 세부 정보
			에 프로비저닝된 처리량이 충분하지 않으면, 다음 1초 안에 이루어지는 호출에서 ProvisionedThroughputExceededException 이 발생합니다.
GetShardIterator	5TPS		샤드 반복자는 요청자에게 반환되고 5분 후에 만료됩니다. GetShardIterator 요청을 너무 자주 하면 요청을 받게 됩니다. ProvisionedThroughputExceededException
PutRecord	1000TPS	각 샤드는 초당 최대 1,000개의 레코드 쓰기를 지원할 수 있으며 최대 데이터 쓰기 총계는 초당 1MB입니다.	

API	API 호출 제한	페이로드 제한	추가 세부 정보
PutRecords		각 PutRecords 요청은 최대 500개의 레코드를 지원할 수 있습니다. 요청에 포함되는 각 레코드 크기의 상한은 1MB이며, 파티션 키를 포함해 전체 요청당 최대 5MB로 제한됩니다. 각 샤드는 초당 최대 1,000개의 레코드 쓰기를 지원할 수 있으며 최대 데이터 쓰기 총계는 초당 1MB입니다.	
SubscribeToShard	샤드당 등록된 SubscribeToShard 소비자당 초당 한 번의 호출을 할 수 있습니다.		동일한 consumerARN으로 SubscribeToShard 다시 전화를 ShardId 걸고 통화가 성공한 후 5초 이내에 전화를 걸면 전화를 받게 됩니다. ResourceNotFoundException

## 할당량 증가

할당량을 조정할 수 있는 경우 Service Quotas를 사용하여 할당량 증가를 요청할 수 있습니다. 일부 요청은 자동으로 해결되고, 일부는 AWS Support에 제출됩니다. AWS Support에 제출된 할당량 증가 요청의 상태를 추적할 수 있습니다. Service Quotas 증가 요청은 우선 순위 지원을 받지 못합니다. 긴급한 요청이 있는 경우 AWS Support에 문의하십시오. 자세한 내용은 [What Is Service Quotas?](#)를 참조하십시오.

서비스 할당량 증가를 요청하려면 [할당량 증가 요청](#)에 설명된 절차를 따르십시오.

# Amazon Kinesis Data Streams 설정

Amazon Kinesis Data Streams를 처음 사용한다면 먼저 다음 작업을 완료해야 합니다.

## Tasks

- [가입하기 AWS](#)
- [라이브러리 및 도구 다운로드](#)
- [개발 환경 구성](#)

## 가입하기 AWS

Amazon Web Services (AWS) 에 가입하면 Kinesis Data Streams를 포함하여 에 있는 AWS 모든 서비스에 AWS 계정이 자동으로 가입됩니다. 사용자에게는 사용한 서비스에 대해서만 요금이 청구됩니다.

이미 AWS 계정이 있다면 다음 작업으로 건너뛰십시오. AWS 계정이 없는 경우 다음 절차에 따라 계정을 생성합니다.

계정을 AWS 등록하려면

1. <https://portal.aws.amazon.com/billing/signup>을 여세요.
2. 온라인 지시 사항을 따르세요.

등록 절차 중에는 전화를 받고 키패드로 인증 코드를 입력하는 과정이 있습니다.

에 AWS 계정가입하면 AWS 계정 루트 사용자a가 생성됩니다. 루트 사용자에게는 계정의 모든 AWS 서비스 및 리소스 액세스 권한이 있습니다. 보안 모범 사례로, 사용자에게 관리자 액세스 권한을 할당하고 루트 사용자 [액세스가 필요한 작업을 수행할 때는 루트 사용자만](#) 사용하십시오.

## 라이브러리 및 도구 다운로드

Kinesis Data Streams로 작업하는 데 도움이 되는 라이브러리와 도구는 다음과 같습니다.

- [Amazon Kinesis API 참조](#)는 Kinesis Data Streams에서 지원하는 기본 작업 세트입니다. Java 코드를 사용하여 기본 작업을 수행하는 것에 대한 자세한 내용은 다음을 참조하십시오.
- [AWS SDK for Java와 Amazon Kinesis Data Streams API를 사용하여 생산자 개발](#)
- [AWS SDK for Java를 사용하여 공유 처리량으로 사용자 지정 소비자 개발](#)

- [스트림 생성 및 관리](#)
- [Go](#), [Java](#), [.NET](#), [Node.js](#), [PHP](#) [JavaScript](#), [Python](#), [Ruby](#)용 [AWS SDK](#)에는 Kinesis Data Streams 지원 및 샘플이 포함되어 있습니다. 사용 중인 버전에 Kinesis Data Streams용 샘플이 포함되어 [AWS SDK for Java](#) 있지 않은 경우 에서 다운로드할 수도 있습니다. [GitHub](#)
- Kinesis 클라이언트 라이브러리 (KCL) 는 데이터 처리를 위한 easy-to-use 프로그래밍 모델을 제공합니다. KCL을 사용하면 Java, Node.js, .NET, Python 및 Ruby에서 Kinesis Data Streams를 빠르게 시작할 수 있습니다. 자세한 내용은 [스트림에서 데이터 읽기](#)를 참조하십시오.
- [AWS Command Line Interface](#)에서 Kinesis Data Streams를 지원합니다. 를 AWS CLI 사용하면 명령줄에서 여러 AWS 서비스를 제어하고 스크립트를 통해 이를 자동화할 수 있습니다.

## 개발 환경 구성

KCL을 사용하려면 Java 개발 환경이 다음 요구 사항을 충족하는지 확인하세요.

- Java 1.7(Java SE 7 JDK) 이상. Oracle 웹 사이트의 [Java SE 다운로드](#)에서 최신 Java 소프트웨어를 다운로드할 수 있습니다.
- Apache Commons 패키지(Code, HTTP Client 및 Logging)
- Jackson JSON 프로세서

[AWS SDK for Java](#)의 경우 Apache Commons 및 Jackson이 타사 폴더에 포함되어 있습니다. 그러나 SDK for Java는 Java 1.6에서 작동하는 반면, Kinesis Client Library에는 Java 1.7이 필요합니다.

# Amazon Kinesis Data Streams 시작하기

이 섹션의 정보는 Amazon Kinesis Data Streams 사용을 시작하는 데 도움이 됩니다. Kinesis Data Streams를 처음 사용하는 경우, 먼저 [Amazon Kinesis Data Streams 용어 및 개념](#)에 있는 개념과 용어를 알아 두세요.

이 섹션에서는 AWS Command Line Interface를 사용하여 기본 Amazon Kinesis Data Streams 작업을 수행하는 방법을 보여줍니다. 기본적인 Kinesis Data Streams 데이터 흐름 원리 및 Kinesis 데이터 스트림에서 데이터를 넣고 가져오는 데 필요한 단계를 학습합니다.

## 주제

- [AWS CLI 설치 및 구성](#)
- [AWS CLI를 사용하여 기본 Kinesis 데이터 스트림 작업 수행](#)

CLI 액세스를 위해서는 액세스 키 ID 및 비밀 액세스 키가 필요합니다. 가능하다면 장기 액세스 키 대신 임시 보안 인증 정보를 사용하십시오. 임시 보안 인증 정보도 액세스 키 ID와 비밀 액세스 키로 구성되지만, 보안 인증이 만료되는 시간을 나타내는 보안 토큰이 포함되어 있습니다. 자세한 내용은 IAM 사용 설명서의 [AWS 리소스에서 임시 자격 증명 사용](#)을 참조하세요.

[IAM 사용자 생성](#)에서 세부 단계별 IAM 및 보안 키 설정 지침을 찾을 수 있습니다.

이 단원에서 설명된 특정 명령은 있는 그대로 제공됩니다. 단, 각 실행에 대해 특정 값이 달라야 하는 경우는 제외됩니다. 또한 예제에서는 미국 서부(오레곤) 리전을 사용하지만 이 섹션의 단계는 [Kinesis Data Streams가 지원되는 모든 리전](#)에서 작동합니다.

## AWS CLI 설치 및 구성

### AWS CLI 설치

Windows용 및 Linux, OS X, Unix 운영 체제용 AWS CLI를 설치하는 방법에 대한 자세한 단계는 [AWS CLI 설치](#)를 참조하세요.

다음 명령을 사용하여 사용 가능한 옵션과 서비스를 나열합니다.

```
aws help
```

Kinesis Data Streams 서비스를 사용하고 있으므로 다음 명령을 사용하여 Kinesis Data Streams와 관련된 AWS CLI 하위 명령을 검토할 수 있습니다.

```
aws kinesis help
```

이 명령을 실행하면 사용 가능한 Kinesis Data Streams 명령이 포함된 결과가 출력됩니다.

#### AVAILABLE COMMANDS

- o add-tags-to-stream
- o create-stream
- o delete-stream
- o describe-stream
- o get-records
- o get-shard-iterator
- o help
- o list-streams
- o list-tags-for-stream
- o merge-shards
- o put-record
- o put-records
- o remove-tags-from-stream
- o split-shard
- o wait

이 명령 목록은 [Amazon Kinesis Service API 참조](#)에 설명된 Kinesis Data Streams API에 해당합니다. 예를 들어, create-stream 명령은 CreateStream API 작업에 해당합니다.

이제 AWS CLI가 성공적으로 설치되었지만 구성되지 않았습니다. 구성은 다음 단원에 표시됩니다.

## AWS CLI 구성

일반적인 용도에서 `aws configure` 명령은 AWS CLI 설치를 설정할 수 있는 가장 빠른 방법입니다. 자세한 내용은 [AWS CLI 구성](#)을 참조하세요.

## AWS CLI를 사용하여 기본 Kinesis 데이터 스트림 작업 수행

이 섹션에서는 AWS CLI를 사용하여 명령줄에서 Kinesis 데이터 스트림을 사용하는 기본 방법을 설명합니다. [Amazon Kinesis Data Streams 용어 및 개념](#)에 설명된 개념을 숙지하십시오.

### Note

Kinesis Data Streams는 AWS 프리 티어에서 제공되지 않으므로 스트림을 생성한 후에는 Kinesis Data Streams 사용에 대한 일반 요금이 계정에 부과됩니다. 이 자습서를 완료하면 AWS 리소스를 삭제하여 요금 발생을 중지하세요. 자세한 내용은 [4단계: 정리](#) 섹션을 참조하세요.

### 주제

- [단계 1: VPC 생성](#)
- [2단계: 레코드 넣기](#)
- [3단계: 레코드 가져오기](#)
- [4단계: 정리](#)

## 단계 1: VPC 생성

첫 번째 단계는 스트림을 생성하고 성공적으로 생성되었는지 확인하는 것입니다. 다음 명령을 사용하여 이름이 "Foo"인 스트림을 생성합니다.

```
aws kinesis create-stream --stream-name Foo
```

그런 다음, 다음 명령을 사용하여 스트림의 생성 진행 상황을 확인합니다.

```
aws kinesis describe-stream-summary --stream-name Foo
```

다음 예와 비슷한 출력 결과를 얻어야 합니다.

```
{
  "StreamDescriptionSummary": {
    "StreamName": "Foo",
    "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/Foo",
    "StreamStatus": "CREATING",
    "RetentionPeriodHours": 48,
    "StreamCreationTimestamp": 1572297168.0,
    "EnhancedMonitoring": [
      {
        "ShardLevelMetrics": []
      }
    ],
    "EncryptionType": "NONE",
    "OpenShardCount": 3,
    "ConsumerCount": 0
  }
}
```

이 예에서 스트림은 아직 사용할 준비가 되지 않았다는 CREATING 상태입니다. 잠시 후 다시 확인하면 다음 예와 비슷한 출력 결과가 표시되어야 합니다.

```
{
  "StreamDescriptionSummary": {
    "StreamName": "Foo",
    "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/Foo",
    "StreamStatus": "ACTIVE",
    "RetentionPeriodHours": 48,
    "StreamCreationTimestamp": 1572297168.0,
    "EnhancedMonitoring": [
      {
        "ShardLevelMetrics": []
      }
    ],
    "EncryptionType": "NONE",
    "OpenShardCount": 3,
    "ConsumerCount": 0
  }
}
```



이 출력에는 이 자습서에 대해 걱정할 필요가 없는 정보가 있습니다. 가장 중요한 것은 스트림이 사용할 준비가 되었음을 알리는 "StreamStatus": "ACTIVE"와 요청한 단일 샤드에 대한 정보입니다. 다음과 같이 `list-streams` 명령을 사용하여 새 스트림의 존재 여부를 확인할 수도 있습니다.

```
aws kinesis list-streams
```

출력:

```
{
  "StreamNames": [
    "Foo"
  ]
}
```

## 2단계: 레코드 넣기

이제 스트림이 활성 상태이므로 일부 데이터를 입력할 준비가 되었습니다. 이 자습서의 경우 가능한 가장 간단한 명령인 `put-record`를 사용합니다. 이 명령은 "testdata" 텍스트를 포함하는 단일 데이터 레코드를 스트림에 넣습니다.

```
aws kinesis put-record --stream-name Foo --partition-key 123 --data testdata
```

이 명령이 제대로 실행되면 다음 예와 비슷한 출력 결과가 발생합니다.

```
{
  "ShardId": "shardId-000000000000",
  "SequenceNumber": "49546986683135544286507457936321625675700192471156785154"
}
```

축하합니다. 스트림에 데이터를 추가했습니다. 그런 다음 스트림에서 데이터를 가져오는 방법이 표시됩니다.

## 3단계: 레코드 가져오기

GetShardIterator

스트림에서 데이터를 가져오기 전에 관심이 있는 샤드에 대한 샤드 반복자를 가져와야 합니다. 샤드 반복자는 소비자(이 경우 `get-record` 명령)가 읽을 샤드와 스트림의 위치를 나타냅니다. 다음과 같이 `get-shard-iterator` 명령을 사용합니다.

```
aws kinesis get-shard-iterator --shard-id shardId-000000000000 --shard-iterator-type
TRIM_HORIZON --stream-name Foo
```

`aws kinesis` 명령의 배경에는 Kinesis Data Streams API가 있으므로 표시된 파라미터에 대해 궁금한 경우 [GetShardIterator](#) API 참조 주제에서 해당 파라미터에 대해 읽을 수 있습니다. 성공적으로 실행되면 다음 예와 비슷한 출력 결과가 발생합니다(전체 출력을 확인하려면 수평으로 스크롤).

```
{
  "ShardIterator": "AAAAAAAAAAHSyw1jv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp
+KEd9I6AJ9ZG41NR1EMi+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWR060TZRNw9gd
+efGN2aHFdkH1rJl4BL9Wyrk+ghYG22D2T1Da2EyNSH1+LAbK33gQweTJADBdyMwlo5r6PqcP2dzhg="
}
```

외견상으로 무작위처럼 보이는 문자의 긴 문자열이 샤드 반복자입니다(사용자의 반복자는 다름). 다음에 표시된 대로 샤드 반복자를 복사하여 가져오기 명령에 붙여 넣어야 합니다. 샤드 반복자에는 300초의 유효한 수명 주기가 있습니다. 이 수명 주기는 샤드 반복자를 복사하여 다음 명령에 붙여 넣는데 충분한 시간이어야 합니다. 다음 명령에 붙여 넣기 전에 샤드 반복자에서 모든 줄 바꿈을 제거해야 합니다. 샤드 반복자가 더 이상 유효하지 않다는 오류 메시지가 발생하는 경우 `get-shard-iterator` 명령을 다시 실행하십시오.

## GetRecords

`get-records` 명령은 스트림에서 데이터를 가져오고 Kinesis Data Streams API의 [GetRecords](#) 호출로 확인됩니다. 샤드 반복기는 샤드에서 순차적으로 데이터 레코드 읽기를 시작할 위치를 지정합니다. 반복기가 가리키는 샤드 부분에 사용 가능한 레코드가 없는 경우 `GetRecords`는 빈 목록을 반환합니다. 레코드를 포함하는 샤드 부분을 가져오기 위해 여러 번의 호출이 수행될 수 있습니다.

다음 `get-records` 명령의 예제에서(전체 명령을 확인하려면 수평으로 스크롤) 다음을 수행합니다.

```
aws kinesis get-records --shard-iterator
AAAAAAAAAAHSyw1jv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp+KEd9I6AJ9ZG41NR1EMi
+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWR060TZRNw9gd+efGN2aHFdkH1rJl4BL9Wyrk
+ghYG22D2T1Da2EyNSH1+LAbK33gQweTJADBdyMwlo5r6PqcP2dzhg=
```

`bash` 등 Unix 유형 명령 프로세서에서 이 자습서를 실행하는 경우 다음과 같이 중첩 명령을 사용하여 샤드 반복기의 획득을 자동화할 수 있습니다(전체 명령을 보려면 수평으로 스크롤).

```
SHARD_ITERATOR=$(aws kinesis get-shard-iterator --shard-id shardId-000000000000 --
shard-iterator-type TRIM_HORIZON --stream-name Foo --query 'ShardIterator')
```

```
aws kinesis get-records --shard-iterator $SHARD_ITERATOR
```

PowerShell을 지원하는 시스템에서 이 자습서를 실행하는 경우 다음과 같이 명령을 사용하여 샤드 반복기의 획득을 자동화할 수 있습니다(전체 명령을 보려면 수평으로 스크롤).

```
aws kinesis get-records --shard-iterator ((aws kinesis get-shard-iterator --shard-id
shardId-000000000000 --shard-iterator-type TRIM_HORIZON --stream-name Foo).split(' '))
[4])
```

get-records 명령을 성공적으로 실행하면 다음 예제와 같이 스트림에서 샤드 반복기를 가져올 때 지정한 샤드에 대한 레코드를 요청합니다(전체 출력을 보려면 수평으로 스크롤).

```
{
  "Records": [ {
    "Data": "dGVzdGRhdGE=",
    "PartitionKey": "123",
    "ApproximateArrivalTimestamp": 1.441215410867E9,
    "SequenceNumber": "49544985256907370027570885864065577703022652638596431874"
  } ],
  "MillisBehindLatest": 24000,

  "NextShardIterator": "AAAAAAAAAAED0W3ugseWPE4503kqN1yN1UaodY8unE0sYs1MUmC6lX9hlig5+t4RtZM0/
tALfiI4QGjunVgJvQsjxjh2aLyxaAaPr
+LaoENQ7eVs4EdYXgKyThTZGPcca2fVXYJWL3yafv9dsDwsYVedI66dbMZFC8rPMWc797zxQkv4pSKvP0ZvrUIudb8UkH3V
}"
```

get-records는 위에 요청으로 설명되어 있습니다. 즉, 스트림에 레코드가 있는 경우에도 0개 이상의 레코드를 받을 수 있으며, 반환된 레코드는 현재 스트림의 모든 레코드를 나타내지 않을 수 있습니다. 이러한 상황은 완벽하게 일반적이며, 프로덕션 코드는 적절한 간격으로 레코드에 대한 스트림을 폴링합니다(이 폴링 속도는 특정 애플리케이션 디자인 요구 사항에 따라 다름).

자습서의 이 부분에서 레코드에 대해 알 수 있는 첫 번째 사항은 데이터가 가비지로 보인다는 것입니다. 이 데이터는 당사가 전송한 일반 텍스트 testdata가 아닙니다. 이는 바이너리 데이터를 전송할 수 있도록 put-record가 Base64 방식의 인코딩을 사용하기 때문입니다. 그러나 AWS CLI의 Kinesis Data Streams 지원은 Base64 방식의 디코딩을 제공하지 않습니다. 왜냐하면 stdout에 인쇄된 원시 바이너리 콘텐츠에 대한 Base64 방식의 디코딩은 특정 플랫폼 및 터미널에 대해 불필

요한 동작과 잠재적인 보안 문제가 발생할 수 있기 때문입니다. Base64 방식의 디코더(예: <https://www.base64decode.org/>)를 사용하여 수동으로 dGVzdGRhdGE=를 디코딩하면 실제로 이 데이터가 testdata임을 알 수 있습니다. 실제로 AWS CLI는 데이터를 소비하는 데 거의 사용되지 않으며, 이전에(describe-stream 및 list-streams)에 살펴본 것처럼 스트림의 상태를 모니터링하고 정보를 얻는 데 더 많이 사용되므로 이 자습서에서는 이 데이터로 충분합니다. 향후 자습서에서는 Base64 방식이 자동으로 처리되는 Kinesis Client Library(KCL)를 사용하여 프로덕션 품질의 소비자 애플리케이션을 빌드하는 방법을 보여줍니다. KCL에 대한 자세한 내용은 [KCL을 사용하여 공유 처리량으로 소비자 지정 소비자 개발](#)을 참조하세요.

경우에 따라 get-records가 지정된 스트림/샤드에서 모든 레코드를 반환하지 않는 경우도 있습니다. 이러한 경우, 마지막 결과에서 NextShardIterator를 사용하여 다음 레코드 세트를 가져옵니다. 따라서 더 많은 데이터를 스트림에 넣은 경우(프로덕션 애플리케이션의 일반적인 상황) 매번 get-records를 사용하여 데이터에 대한 폴링을 유지할 수 있습니다. 그러나 300초 샤드 반복기 수명 주기 이내에 다음 샤드 반복기를 사용하여 get-records를 호출하지 않으면 오류 메시지가 발생하며, get-shard-iterator 명령을 사용하여 새로운 샤드 반복기를 가져와야 합니다.

이 출력에는 MillisBehindLatest도 제공됩니다. 이 값은 스트림의 끝에서 [GetRecords](#) 작업의 응답이 나오는 시간(밀리초)이며, 소비자가 있는 현재 시간에서 경과된 시간을 나타냅니다. 값이 0이면 레코드 처리를 따라잡았으며 이 시점에서 처리할 새 레코드가 없음을 나타냅니다. 이 자습서의 경우 시간을 들여 꾸준히 읽을 경우 상당히 큰 숫자가 표시될 수 있습니다. 기본적으로 데이터 레코드가 검색을 위해 대기하도록 24시간 동안 스트림에 유지되므로 문제가 되지 않습니다. 이 시간을 보존 기간이라고 하며, 최대 365일로 구성할 수 있습니다.

get-records를 성공적으로 실행하면 현재 스트림에 더 이상 레코드가 없는 경우에도 항상 NextShardIterator가 발생합니다. 이는 생산자가 특정 시점에서 스트림에 더 많은 레코드를 잠재적으로 넣을 수 있다고 가정하는 폴링 모델입니다. 자체의 폴링 루틴을 작성할 수 있는 경우에도 소비자 애플리케이션을 개발하기 위해 이전에 언급한 KCL을 사용할 경우 이 폴링이 무리없이 수행됩니다.

가져올 스트림과 샤드에 더 이상 레코드가 없을 때까지 get-records를 호출할 경우 다음 예와 비슷하게 빈 레코드가 포함된 출력이 표시됩니다(전체 출력을 보려면 수평으로 스크롤).

```
{
  "Records": [],
  "NextShardIterator": "AAAAAAAAAAGCJ5jzQNjmdh06B/YDIDE56jmZmrmmA/r1WjoHXC/kPJXc1rckt3TFL55dENfe5meNgdkyCRpUPGzJpMgYHaJ53C3nCAjQ6s7ZupjXeJGoUFs5oCuFwhP+Wu1/EhyNeSs5DYXLSSC5XCcapmCAYGFjYER69QSDqjxMmBPE/hiybFDi5qtkT6/PsZNz6kFoqtDk="
}
```

## 4단계: 정리

마지막으로 이전에 명시된 대로 스트림을 삭제하여 리소스를 비우고 계정에 의도하지 않은 요금 청구를 방지합니다. 스트림을 사용하여 데이터를 넣고 가져오든 사용하지 않든 관계없이 스트림당 요금이 누적되므로 스트림을 생성하고 사용하지 않는 경우 언제든지 이를 실제로 수행합니다. 정리 명령은 다음과 같이 간단합니다.

```
aws kinesis delete-stream --stream-name Foo
```

성공하면 출력이 발생하지 않으므로 `describe-stream`을 사용하여 삭제 진행 상황을 확인할 수 있습니다.

```
aws kinesis describe-stream-summary --stream-name Foo
```

삭제 명령 직후 이 명령을 실행하면 다음 예와 비슷한 출력 부분이 표시될 수 있습니다.

```
{
  "StreamDescriptionSummary": {
    "StreamName": "samplestream",
    "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/samplestream",
    "StreamStatus": "ACTIVE",
```

스트림이 완전히 삭제된 후 `describe-stream`을 실행하면 "찾을 수 없음" 오류가 발생합니다.

```
A client error (ResourceNotFoundException) occurred when calling the
DescribeStreamSummary operation:
Stream Foo under account 123456789012 not found.
```

# Amazon Kinesis Data Streams에 대한 예제 자습서

이 섹션의 예제 자습서는 Amazon Kinesis Data Streams 개념과 기능을 이해하는 데 도움을 주기 위해 작성되었습니다.

## 주제

- [자습서: KPL 및 KCL 2.x를 사용하여 실시간 주식 데이터 처리](#)
- [자습서: KPL 및 KCL 1.x를 사용하여 실시간 주식 데이터 처리](#)
- [자습서: Flink 애플리케이션용 Managed Service for Apache Flink를 사용하여 실시간 주식 데이터 분석](#)
- [자습서: Amazon Kinesis Data Streams에서 AWS Lambda 사용](#)
- [AWS Streaming Data Solution for Amazon Kinesis](#)

## 자습서: KPL 및 KCL 2.x를 사용하여 실시간 주식 데이터 처리

이 자습서의 시나리오에서는 스트림에 주식 거래를 가져와 데이터 스트림에 대한 계산을 수행하는 간단한 Amazon Kinesis Data Streams 애플리케이션을 작성합니다. 레코드의 스트림을 Kinesis Data Streams에 전송하고, 거의 실시간으로 레코드를 사용하고 처리하는 애플리케이션을 구현하는 방법에 대해 알아봅니다.

### Important

Kinesis Data Streams는 AWS 프리 티어에서 제공되지 않으므로 스트림을 생성한 후에는 Kinesis Data Streams 사용에 대한 일반 요금이 계정에 부과됩니다. 소비자 애플리케이션이 시작된 후 Amazon DynamoDB 사용량에 대해 일반 요금도 발생합니다. 소비자 애플리케이션은 DynamoDB를 사용하여 처리 상태를 추적합니다. 이 애플리케이션을 완료하면 AWS 리소스를 삭제하여 요금 발생을 중지하세요. 자세한 내용은 [단계 7: 마무리](#) 섹션을 참조하세요.

이 코드는 실제 주식 시장 데이터에는 액세스하지 않지만, 대신 주식 거래의 스트림을 시뮬레이션합니다. 2015년 2월 현재 시가 총액 상위 25개 주식에 대한 실제 시장 데이터의 시작점이 있는 임의의 주식 거래 생성기를 사용하여 이를 수행합니다. 주식 거래의 실시간 스트림에 액세스할 수 있는 경우 스트림에서 유용하고 시기 적절한 통계를 추출하고 싶을 때도 있습니다. 예를 들어, 마지막 5분 이내에 구매한 가장 인기 있는 주식을 결정하는 슬라이딩 윈도우 분석을 수행하려고 할 수 있습니다. 또는 너무 많은

판매 주문(즉, 너무 많은 공유)이 있을 때마다 알림을 원할 수도 있습니다. 이 시리즈의 코드를 확장하여 이러한 기능을 제공할 수 있습니다.

데스크톱 또는 랩톱 컴퓨터에서 이 자습서의 단계를 완료하고, 동일한 머신에서 또는 정의된 요구 사항을 지원하는 플랫폼에서 생산자 코드와 소비자 코드를 모두 실행할 수 있습니다.

표시된 예제는 미국 서부(오레곤) 리전을 사용하지만 이 예제는 Kinesis Data Streams를 지원하는 모든 [AWS 리전](#)에 적용됩니다.

## 작업

- [필수 조건](#)
- [1단계: 데이터 스트림 생성](#)
- [2단계: IAM 정책 및 사용자 생성](#)
- [3단계: 코드 다운로드 및 빌드](#)
- [단계 4: 생산자 구현](#)
- [단계 5: 소비자 구현](#)
- [단계 6: \(선택 사항\) 소비자 확대](#)
- [단계 7: 마무리](#)

## 필수 조건

이 자습서를 완료하려면 다음 요구 사항을 충족해야 합니다.

### Amazon Web Services 계정

시작하기 전에 [Amazon Kinesis Data Streams 용어 및 개념](#)에서 설명하는 개념을 잘 알고 있어야 합니다. 특히 스트림, 샤드, 생산자 및 소비자 개념을 잘 알아 두십시오. 또한 [AWS CLI 설치 및 구성](#) 가이드의 단계를 완료하면 도움이 됩니다.

AWS Management Console에 액세스하려면 AWS 계정과 웹 브라우저가 있어야 합니다.

콘솔 액세스의 경우 IAM 사용자 이름과 암호를 사용하여 IAM 로그인 페이지에서 [AWS Management Console](#)에 로그인합니다. 프로그래밍 방식 액세스 및 장기 보안 인증에 대한 대안을 포함한 AWS 보안 인증에 대한 자세한 내용은 IAM 사용 설명서의 [AWS 보안 인증](#)을 참조하세요. AWS 계정에 로그인하는 방법에 대한 자세한 내용은 AWS 로그인 사용 설명서의 [AWS에 로그인하는 방법](#)을 참조하세요.

IAM 및 보안 키 설정 지침에 대한 자세한 내용은 [IAM 사용자 생성](#)을 참조하세요.

## 시스템 소프트웨어 요구 사항

애플리케이션을 실행하는 데 사용하는 시스템에는 Java 7 이상이 설치되어 있어야 합니다. 최신 JDK(Java Development Kit)를 다운로드하고 설치하려면 [Oracle의 Java SE 설치 사이트](#)로 이동하십시오.

[Eclipse](#)와 같은 Java IDE가 있는 경우 이 IDE를 사용하여 소스 코드를 열고 편집, 빌드 및 실행할 수 있습니다.

최신 [AWS SDK for Java](#) 버전이 필요합니다. IDE로 Eclipse를 사용하는 경우 [AWS Toolkit for Eclipse](#)를 대신 설치할 수 있습니다.

소비자 애플리케이션에는 Kinesis Client Library(KCL) 버전 2.2.9 이상이 필요하며, GitHub 사이트 <https://github.com/aws-labs/amazon-kinesis-client/tree/master>에서 얻을 수 있습니다.

## 다음 단계

### [1단계: 데이터 스트림 생성](#)

## 1단계: 데이터 스트림 생성

먼저, 이 자습서의 후속 단계에서 사용할 데이터 스트림을 만들어야 합니다.

스트림을 만들려면

1. AWS Management Console에 로그인하여 <https://console.aws.amazon.com/kinesis>에서 Kinesis 콘솔을 엽니다.
2. 탐색 창에서 Data Streams(데이터 스트림)를 선택합니다.
3. 탐색 모음에서 리전 선택기를 확장하고 리전을 선택합니다.
4. Create Kinesis stream(Kinesis 스트림 생성)을 선택합니다.
5. 데이터 스트림의 이름을 입력합니다(예: **StockTradeStream**).
6. 샤드 수에는 **1**을 입력하고, Estimate the number of shards you'll need(필요한 샤드 수 추정)은 축소된 상태로 둡니다.
7. Create Kinesis stream(Kinesis 스트림 생성)을 선택합니다.

(Kinesis 스트림 목록 페이지에서 스트림 상태는 스트림이 생성되는 동안 CREATING으로 표시됩니다. 스트림을 사용할 준비가 되면 상태가 ACTIVE(활성)로 변경됩니다.)



스트림 이름을 선택하면 다음에 나타나는 페이지의 세부 정보 탭에 데이터 스트림 구성의 요약이 표시됩니다. Monitoring(모니터링) 섹션에는 스트림에 대한 모니터링 정보가 표시됩니다.

## 다음 단계

### [2단계: IAM 정책 및 사용자 생성](#)

## 2단계: IAM 정책 및 사용자 생성

AWS의 보안 모범 사례에 따르면 다양한 리소스에 대한 액세스를 제어하기 위해 세부적인 권한을 사용해야 합니다. AWS Identity and Access Management (IAM)를 사용하면 AWS에서 사용자와 사용자 권한을 관리할 수 있습니다. [IAM 정책](#)에는 허용된 작업과 작업이 적용되는 리소스가 명시적으로 나열됩니다.

다음은 Kinesis Data Streams 생산자 및 소비자에 대해 일반적으로 필요한 최소 권한입니다.

### 생산자

작업	리소스	용도
DescribeStream , DescribeStreamSummary , DescribeStreamConsumer	Kinesis 데이터 스트림	레코드를 읽으려고 하기 전에 소비자는 데이터 스트림이 존재하는지 확인하고, 샤드가 데이터 스트림에 포함되어 있는지 확인합니다.
SubscribeToShard , RegisterStreamConsumer	Kinesis 데이터 스트림	소비자를 구독하고 샤드에 등록합니다.
PutRecord , PutRecords	Kinesis 데이터 스트림	Kinesis Data Streams에 레코드를 씁니다.

### 소비자

작업	리소스	용도
DescribeStream	Kinesis 데이터 스트림	레코드를 읽으려고 하기 전에 소비자는 데이터 스트림이 존재하는지 확인하고, 샤드가 데이터 스트림에 포함되어 있는지 확인합니다.

작업	리소스	용도
GetRecords , GetShardIterator	Kinesis 데이터 스트림	샤드에서 레코드를 읽습니다.
CreateTable , DescribeTable , GetItem, PutItem, Scan, UpdateItem	Amazon DynamoDB 테이블	Kinesis Client Library(KCL)(버전 1.x 또는 2.x)를 사용하여 스토리지 애플리케이션의 처리 상태를 추적하려면 DynamoDB 테이블을 사용합니다.
DeleteItem	Amazon DynamoDB 테이블	소비자가 Kinesis Data Streams 샤드에서 분할/병합 작업을 수행합니다.
PutMetricData	Amazon CloudWatch 로그 그룹	또한 KCL은 애플리케이션을 모니터링하는 데 유용한 지표를 기록합니다.

이 자습서에서는 위의 모든 권한을 부여하는 단일 IAM 정책을 생성합니다. 프로덕션 환경에서는 생산자와 소비자에 대해 각각 하나씩 정책을 두 개 만들 수 있습니다.

### IAM 정책을 만들려면

1. 위 단계에서 생성한 새 데이터 스트림의 Amazon 리소스 이름(ARN)을 찾습니다. 세부 정보 탭 상단에 스트림 ARN으로 나열된 이 ARN을 찾을 수 있습니다. ARN 형식은 다음과 같습니다.

```
arn:aws:kinesis:region:account:stream/name
```

#### 리전

AWS 리전 코드입니다(예: us-west-2). 자세한 내용은 [리전 및 가용 영역 개념](#)을 참조하십시오.

#### account

[계정 설정](#)에 표시된 대로 AWS 계정 ID입니다.

#### name

위 단계에서 생성한 데이터 스트림의 이름입니다(StockTradeStream).

2. 소비자가 사용할 DynamoDB 테이블의 ARN을 결정합니다(첫 번째 소비자 인스턴스에서 생성됨). 형식은 다음과 같아야 합니다.

```
arn:aws:dynamodb:region:account:table/name
```

리전 및 계정 ID는 이 자습서에서 사용하는 데이터 스트림의 ARN에 있는 값과 동일하지만, 이름은 소비자 애플리케이션에서 생성하고 사용하는 DynamoDB 테이블의 이름입니다. KCL은 애플리케이션 이름을 테이블 이름으로 사용합니다. 이 단계에서는 DynamoDB 테이블 이름에 StockTradesProcessor를 사용합니다. 이 이름은 이 자습서의 이후 단계에서 사용되는 애플리케이션 이름이기 때문입니다.

3. IAM 콘솔의 정책(<https://console.aws.amazon.com/iam/home#policies>)에서 정책 생성을 선택합니다. IAM 정책을 사용한 첫 번째 작업인 경우 시작하기, 정책 생성을 선택합니다.
4. 정책 생성기 옆의 선택을 선택합니다.
5. Amazon Kinesis를 AWS 서비스로 선택합니다.
6. DescribeStream, GetShardIterator, GetRecords, PutRecord 및 PutRecords를 허용된 작업으로 선택합니다.
7. 이 자습서에서 사용하는 데이터 스트림의 ARN을 입력합니다.
8. 다음의 각각에 대해 Add Statement(문 추가)를 사용합니다.

AWS 서비스	작업	ARN
Amazon DynamoDB	CreateTable , DeleteItem , DescribeTable , GetItem, PutItem, Scan, UpdateItem	이 절차의 2단계에서 생성한 DynamoDB 테이블의 ARN입니다.
Amazon CloudWatch	PutMetricData	*

별표(\*)는 ARN이 필요하지 않다고 지정할 때 사용됩니다. 이 경우에는 PutMetricData 작업이 간접적으로 호출된 CloudWatch에서 특정 리소스가 없기 때문입니다.

9. 다음 단계(Next Step)를 선택합니다.
10. Policy Name(정책 이름)을 StockTradeStreamPolicy로 변경하고, 코드를 검토한 다음 Create Policy(정책 생성)를 선택합니다.

결과 정책 문서는 다음과 같아야 합니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt123",
      "Effect": "Allow",
      "Action": [
        "kinesis:DescribeStream",
        "kinesis:PutRecord",
        "kinesis:PutRecords",
        "kinesis:GetShardIterator",
        "kinesis:GetRecords",
        "kinesis:ListShards",
        "kinesis:DescribeStreamSummary",
        "kinesis:RegisterStreamConsumer"
      ],
      "Resource": [
        "arn:aws:kinesis:us-west-2:123:stream/StockTradeStream"
      ]
    },
    {
      "Sid": "Stmt234",
      "Effect": "Allow",
      "Action": [
        "kinesis:SubscribeToShard",
        "kinesis:DescribeStreamConsumer"
      ],
      "Resource": [
        "arn:aws:kinesis:us-west-2:123:stream/StockTradeStream/*"
      ]
    },
    {
      "Sid": "Stmt456",
      "Effect": "Allow",
      "Action": [
        "dynamodb:*"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-west-2:123:table/StockTradesProcessor"
      ]
    }
  ]
}
```

```

    "Sid": "Stmt789",
    "Effect": "Allow",
    "Action": [
        "cloudwatch:PutMetricData"
    ],
    "Resource": [
        "*"
    ]
}
]
}

```

## IAM 사용자 생성

1. <https://console.aws.amazon.com/iam/>에서 IAM 콘솔을 엽니다.
2. Users(사용자) 페이지에서 Add user(사용자 추가)를 선택합니다.
3. 사용자 이름(User name)에 StockTradeStreamUser를 입력합니다.
4. Access type(액세스 유형)에서 Programmatic access(프로그래밍 방식 액세스)를 선택한 다음 Next: Permissions(다음: 권한)를 선택합니다.
5. [Attach existing policies directly]를 선택합니다.
6. 위의 절차에서 만든 정책을 이름으로 검색합니다(StockTradeStreamPolicy). 정책 이름 왼쪽에 있는 확인란을 선택하고 Next: Review(다음: 검토)를 선택합니다.
7. 세부 정보와 요약을 검토하고 Create user(사용자 생성)를 선택합니다.
8. Access key ID(액세스 키 ID)를 복사하고 비공개로 저장합니다. Secret access key(보안 액세스 키)에서 Show(표시)를 선택하고 키도 비공개로 저장합니다.
9. 액세스 및 보안 키를 사용자만 액세스할 수 있는 안전한 위치에 있는 로컬 파일에 붙여넣습니다. 이 애플리케이션의 경우 ~/.aws/credentials라는 파일 이름을 생성합니다(엄격한 권한 포함). 파일은 다음 형식이어야 합니다.

```

[default]
aws_access_key_id=access key
aws_secret_access_key=secret access key

```

## 사용자에게 IAM 정책 연결

1. IAM 콘솔에서 [정책](#)을 열고 정책 작업을 선택합니다.
2. StockTradeStreamPolicy 및 Attach(연결)를 선택합니다.

3. StockTradeStreamUser 및 Attach Policy(정책 연결)를 선택합니다.

다음 단계

### [3단계: 코드 다운로드 및 빌드](#)

## 3단계: 코드 다운로드 및 빌드

이 주제에서는 데이터 스트림으로 샘플 주식 거래 수집(생산자) 및 이 데이터 처리(소비자)에 대한 샘플 구현 코드를 제공합니다.

코드를 다운로드하고 빌드하려면

1. GitHub 저장소 <https://github.com/aws-samples/amazon-kinesis-learning>에서 컴퓨터로 소스 코드를 다운로드합니다.
2. 제공된 디렉터리 구조를 따라 소스 코드를 사용하여 IDE에서 프로젝트를 생성합니다.
3. 프로젝트에 다음 라이브러리를 추가합니다.
  - Amazon Kinesis Client Library(KCL)
  - AWS SDK
  - Apache HttpCore
  - Apache HttpClient
  - Apache Commons Lang
  - Apache Commons Logging
  - Guava(Google Core Libraries For Java)
  - Jackson Annotations
  - Jackson Core
  - Jackson Databind
  - Jackson Dataformat: CBOR
  - Joda Time
4. IDE에 따라 프로젝트가 자동으로 빌드될 수 있습니다. 그렇지 않으면 IDE에 적합한 단계를 사용하여 프로젝트를 빌드하십시오.

이러한 단계를 성공적으로 완료한 경우 이제 다음 단원([the section called “단계 4: 생산자 구현”](#))으로 이동할 준비가 되었습니다.

## 다음 단계

### 단계 4: 생산자 구현

이 자습서에서는 주식 시장 거래 모니터링의 실제 시나리오를 사용합니다. 다음 원칙은 이 시나리오가 생산자와 생산자의 지원 코드 구조에 매핑되는 방법을 간략하게 설명합니다.

[소스 코드](#)를 참조하여 다음 정보를 검토하십시오.

#### StockTrade 클래스

개별 주식 거래는 StockTrade 클래스의 인스턴스로 표시됩니다. 이 인스턴스에는 티커 기호, 가격, 공유 수, 거래 유형(구매 또는 판매), 거래를 고유하게 식별하는 ID 등의 속성이 포함됩니다. 이 클래스가 사용자를 위해 구현됩니다.

#### 스트림 레코드

스트림은 레코드의 시퀀스입니다. 레코드는 JSON 형식으로 된 StockTrade 인스턴스의 직렬화입니다. 예:

```
{
  "tickerSymbol": "AMZN",
  "tradeType": "BUY",
  "price": 395.87,
  "quantity": 16,
  "id": 3567129045
}
```

#### StockTradeGenerator 클래스

StockTradeGenerator에는 호출될 때마다 임의로 생성된 새 주식 거래를 반환하는 getRandomTrade()라는 메서드가 있습니다. 이 클래스가 사용자를 위해 구현됩니다.

#### StockTradesWriter 클래스

생산자의 main 메서드인 StockTradesWriter는 계속적으로 임의의 거래를 검색하고 다음 작업을 수행하여 Kinesis Data Streams에 전송합니다.

1. 데이터 스트림 이름과 리전 이름을 입력으로 읽습니다.

2. `KinesisAsyncClientBuilder`를 사용하여 리전, 자격 증명 및 클라이언트 구성을 설정합니다.
3. 스트림의 존재 여부와 활성 상태 여부를 확인합니다. 그렇지 않은 경우 오류로 종료됩니다.
4. 연속 루프에서 `StockTradeGenerator.getRandomTrade()` 메서드를 호출하고 `sendStockTrade` 메서드를 호출하여 100밀리초마다 거래를 스트림으로 전송합니다.

`sendStockTrade` 클래스의 `StockTradesWriter` 메서드에는 다음 코드가 있습니다.

```
private static void sendStockTrade(StockTrade trade, KinesisAsyncClient
    kinesisClient,
        String streamName) {
    byte[] bytes = trade.toJsonAsBytes();
    // The bytes could be null if there is an issue with the JSON serialization
    by the Jackson JSON library.
    if (bytes == null) {
        LOG.warn("Could not get JSON bytes for stock trade");
        return;
    }

    LOG.info("Putting trade: " + trade.toString());
    PutRecordRequest request = PutRecordRequest.builder()
        .partitionKey(trade.getTickerSymbol()) // We use the ticker symbol
        as the partition key, explained in the Supplemental Information section below.
        .streamName(streamName)
        .data(SdkBytes.fromByteArray(bytes))
        .build();

    try {
        kinesisClient.putRecord(request).get();
    } catch (InterruptedException e) {
        LOG.info("Interrupted, assuming shutdown.");
    } catch (ExecutionException e) {
        LOG.error("Exception while sending data to Kinesis. Will try again next
        cycle.", e);
    }
}
```

다음 코드 세부 분석을 참조하십시오.

- `PutRecord` API에는 바이트 배열이 필요하며, 거래를 JSON 형식으로 변환해야 합니다. 이 한 줄의 코드는 다음 작업을 수행합니다.



```
byte[] bytes = trade.toJsonAsBytes();
```

- 거래를 전송하기 전에 새 PutRecordRequest 인스턴스(이 경우 요청이라고 함)를 생성합니다. 각 request에는 스트림 이름, 파티션 키 및 데이터 BLOB가 필요합니다.

```
PutRecordRequest request = PutRecordRequest.builder()
    .partitionKey(trade.getTickerSymbol()) // We use the ticker symbol as the
    partition key, explained in the Supplemental Information section below.
    .streamName(streamName)
    .data(SdkBytes.fromByteArray(bytes))
    .build();
```

이 예제는 특정 샤드에 레코드를 매핑하는 주식 티켓을 파티션 키로 사용합니다. 실제로 레코드가 스트림에 대해 균등하게 분산되도록 샤드당 수백 개 또는 수천 개의 파티션 키가 있어야 합니다. 스트림에 데이터를 추가하는 방법에 대한 자세한 내용은 [Amazon Kinesis Data Streams에 데이터 쓰기](#) 단원을 참조하십시오.

이제 request를 클라이언트에 전송할 준비가 되었습니다(put 작업).

```
kinesisClient.putRecord(request).get();
```

- 오류 확인과 로깅 기능은 항상 유용한 추가 기능입니다. 이 코드는 오류 조건을 기록합니다.

```
if (bytes == null) {
    LOG.warn("Could not get JSON bytes for stock trade");
    return;
}
```

put넣기 작업에 try/catch 블록을 추가합니다.

```
try {
    kinesisClient.putRecord(request).get();
}
```

```

} catch (InterruptedException e) {
    LOG.info("Interrupted, assuming shutdown.");
} catch (ExecutionException e) {
    LOG.error("Exception while sending data to Kinesis. Will try again
next cycle.", e);
}

```

이렇게 하는 이유는 네트워크 오류로 인해 또는 처리량 제한에 도달하여 병목 현상이 발생한 데이터 스트림으로 인해 Kinesis Data Streams put 작업이 실패할 수 있기 때문입니다. 데이터 손실을 방지하기 위해 단순 재시도 사용과 같은 put 작업에 대한 재시도 정책을 신중히 고려하는 것이 좋습니다.

- 상태 로깅은 유용하지만 선택 사항입니다.

```
LOG.info("Putting trade: " + trade.toString());
```

여기에 표시된 생산자는 Kinesis Data Streams API 단일 레코드 기능인 PutRecord를 사용합니다. 실제로 개별 생산자가 많은 레코드를 생성하는 경우 PutRecords의 여러 레코드 기능을 사용하고 레코드의 배치를 한 번에 전송하는 것이 더 효율적인 경우가 많습니다. 자세한 내용은 [Amazon Kinesis Data Streams에 데이터 쓰기](#) 섹션을 참조하세요.

생산자를 실행하려면

1. [2단계: IAM 정책 및 사용자 생성](#)에서 검색한 액세스 키 및 보안 키 페어가 파일 `~/.aws/credentials`에 저장되었는지 확인합니다.
2. 다음과 같은 인수를 사용하여 StockTradeWriter 클래스를 실행합니다.

```
StockTradeStream us-west-2
```

us-west-2 이외의 리전에서 스트림을 생성한 경우 해당 리전을 여기에 대신 지정해야 합니다.

다음과 유사한 출력 화면이 표시되어야 합니다.

```

Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
  sendStockTrade
INFO: Putting trade: ID 8: SELL 996 shares of BUD for $124.18
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
  sendStockTrade
INFO: Putting trade: ID 9: BUY 159 shares of GE for $20.85
Feb 16, 2015 3:53:01 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
  sendStockTrade
INFO: Putting trade: ID 10: BUY 322 shares of WMT for $90.08

```

이제 주식 거래가 Kinesis Data Streams에서 수집됩니다.

## 다음 단계

### [단계 5: 소비자 구현](#)

## 단계 5: 소비자 구현

이 자습서의 소비자 애플리케이션은 데이터 스트림에서 주식 거래를 지속적으로 처리합니다. 그런 다음 1분마다 매매된 가장 인기 있는 주식들을 출력합니다. 애플리케이션은 소비자 앱에 공통적인 과중한 업무를 많이 수행하는 Kinesis Client Library(KCL)를 기반으로 하여 빌드됩니다. 자세한 내용은 [Kinesis Client Library 사용](#) 섹션을 참조하세요.

소스 코드를 참조하여 다음 정보를 검토하십시오.

### StockTradesProcessor 클래스

다음 작업을 수행하는 소비자의 기본 클래스가 제공됩니다.

- 인수로 전달된 애플리케이션, 데이터 스트림 및 리전 이름을 읽습니다.
- 리전 이름을 사용하여 KinesisAsyncClient 인스턴스를 생성합니다.
- StockTradeRecordProcessorFactory 인스턴스에 의해 구현된 ShardRecordProcessor의 서버 인스턴스를 제공하는 StockTradeRecordProcessor 인스턴스를 생성합니다.
- KinesisAsyncClient, StreamName, ApplicationName 및 StockTradeRecordProcessorFactory 인스턴스를 사용하여 ConfigsBuilder 인스턴스를 생성합니다. 이 기능은 기본값을 사용하여 모든 구성을 생성하는 경우에 유용합니다.

- `ConfigsBuilder` 인스턴스를 사용하여 KCL 스케줄러(이전에 KCL 버전 1.x에서는 KCL 작업자라고 함)를 생성합니다.
- 스케줄러는 각 샤드(이 소비자 인스턴스에 할당된 샤드)에 대해 새 스레드를 생성합니다. 이 스레드는 데이터 스트림에서 계속 반복적으로 레코드를 읽습니다. 그런 다음 `StockTradeRecordProcessor` 인스턴스를 호출하여 수신한 각 일괄 레코드를 처리합니다.

## StockTradeRecordProcessor 클래스

`StockTradeRecordProcessor` 인스턴스의 구현입니다. 이 클래스는 다시 `initialize`, `processRecords`, `leaseLost`, `shardEnded` 및 `shutdownRequested`라는 다섯 가지 필수 메서드를 구현합니다.

`initialize` 및 `shutdownRequested` 메서드는 KCL에서 레코드 수신을 시작할 준비가 될 때 및 레코드 수신을 중지해야 할 때 애플리케이션별 설정 및 종료 작업을 수행할 수 있도록 레코드 프로세서에 알리기 위해 사용됩니다. `leaseLost` 및 `shardEnded`는 리스가 손실되거나 처리가 샤드의 끝에 도달할 때 수행할 작업에 대한 로직을 구현하는 데 사용됩니다. 이 예에서는 이러한 이벤트를 나타내는 메시지만 기록합니다.

이러한 메서드에 대한 코드가 제공됩니다. `processRecords` 메서드에서 기본 처리가 발생하며, 각 레코드에 대해 `processRecord`를 사용합니다. 이 후자의 메서드는 다음 단계에서 구현할 수 있도록 대부분 비어 있는 스켈레톤 코드로 사용자에게 제공됩니다. 이 코드에 대해서는 다음 단계에서 더 자세히 설명합니다.

또한 `processRecord: reportStats` 및 `resetStats`에 대한 지원 메서드의 구현도 중요합니다. 이러한 메서드는 원래 소스 코드에서 비어 있습니다.

`processRecords` 메서드가 구현되며 다음 단계를 수행합니다.

- 전달된 각 레코드에 대해 `processRecord`를 호출합니다.
- 마지막 보고 이후 1분 이상이 경과된 경우 최신 통계를 인쇄하는 `reportStats()`를 호출한 후 통계를 지우는 `resetStats()`를 호출하여 다음 간격에 새 레코드만 포함되도록 합니다.
- 다음 보고 시간을 설정합니다.
- 마지막 체크포인트 이후 1분 이상이 경과된 경우 `checkpoint()`를 호출합니다.
- 다음 검사 시간을 설정합니다.

이 메서드는 보고 및 검사 속도에 대해 60초 간격을 사용합니다. 체크포인트 수행에 대한 자세한 내용은 [Using the Kinesis Client Library](#)를 참조하세요.

## StockStats 클래스

이 클래스는 시간에 따른 가장 인기 있는 주식에 대한 통계 추적 및 데이터 보존을 제공합니다. 다음 메서드가 포함된 이 코드가 제공됩니다.

- `addStockTrade(StockTrade)`: 지정된 `StockTrade`를 실행 중인 통계에 주입합니다.
- `toString()`: 형식이 지정된 문자열로 통계를 반환합니다.

이 클래스는 각 주식에 대한 총 거래 수의 실행 개수와 최대 개수를 유지하여 가장 인기 있는 주식을 추적합니다. 그리고 주식 거래가 발생할 때마다 이러한 계수가 업데이트됩니다.

다음 단계에 표시된 대로 `StockTradeRecordProcessor` 클래스의 메서드에 코드를 추가합니다.

### 소비자를 구현하려면

1. 정확한 크기의 `processRecord` 객체를 인스턴스화하고, 해당 객체에 레코드 데이터를 추가하고, 문제가 있는 경우 경고를 기록하여 `StockTrade` 메서드를 구현합니다.

```
byte[] arr = new byte[record.data().remaining()];
record.data().get(arr);
StockTrade trade = StockTrade.fromJsonAsBytes(arr);
    if (trade == null) {
        log.warn("Skipping record. Unable to parse record into StockTrade.
Partition Key: " + record.partitionKey());
        return;
    }
stockStats.addStockTrade(trade);
```

2. 간단한 `reportStats` 메서드를 구현합니다. 기본 설정에 적합하게 출력 형식을 자유롭게 수정합니다.

```
System.out.println("***** Shard " + kinesisShardId + " stats for last 1 minute
*****\n" +
stockStats + "\n" +
"*****\n");
```

3. `resetStats` 메서드를 구현합니다. 이 메서드는 새 `stockStats` 인스턴스를 생성합니다.

```
stockStats = new StockStats();
```

#### 4. ShardRecordProcessor 인터페이스에 필요한 다음과 같은 메서드를 구현합니다.

```
@Override
public void leaseLost(LeaseLostInput leaseLostInput) {
    log.info("Lost lease, so terminating.");
}

@Override
public void shardEnded(ShardEndedInput shardEndedInput) {
    try {
        log.info("Reached shard end checkpointing.");
        shardEndedInput.checkpointer().checkpoint();
    } catch (ShutdownException | InvalidStateException e) {
        log.error("Exception while checkpointing at shard end. Giving up.", e);
    }
}

@Override
public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
    log.info("Scheduler is shutting down, checkpointing.");
    checkpoint(shutdownRequestedInput.checkpointer());
}

private void checkpoint(RecordProcessorCheckpointer checkpointer) {
    log.info("Checkpointing shard " + kinesisShardId);
    try {
        checkpointer.checkpoint();
    } catch (ShutdownException se) {
        // Ignore checkpoint if the processor instance has been shutdown (fail
        over).
        log.info("Caught shutdown exception, skipping checkpoint.", se);
    } catch (ThrottlingException e) {
        // Skip checkpoint when throttled. In practice, consider a backoff and
        retry policy.
        log.error("Caught throttling exception, skipping checkpoint.", e);
    } catch (InvalidStateException e) {
```

```
// This indicates an issue with the DynamoDB table (check for table,
// provisioned IOPS).
log.error("Cannot save checkpoint to the DynamoDB table used by the Amazon
Kinesis Client Library.", e);
}
}
```

## 소비자를 실행하려면

1. 에서 작성한 생산자를 실행하여 시뮬레이션된 주식 거래 레코드를 스트림에 첨가합니다.
2. 앞에서(IAM 사용자를 생성할 때) 검색한 액세스 키 및 보안 키 페어가 ~/.aws/credentials 파일에 저장되었는지 확인합니다.
3. 다음과 같은 인수를 사용하여 StockTradesProcessor 클래스를 실행합니다.

```
StockTradesProcessor StockTradeStream us-west-2
```

us-west-2 이외의 리전에서 스트림을 생성한 경우 해당 리전을 여기에 대신 지정해야 합니다.

1분 후 다음과 같은 출력이 표시되어야 하며, 그 이후로 매분마다 새로 고침됩니다.

```
***** Shard shardId-000000000001 stats for last 1 minute *****
Most popular stock being bought: WMT, 27 buys.
Most popular stock being sold: PTR, 14 sells.
*****
```

## 다음 단계

### [단계 6: \(선택 사항\) 소비자 확대](#)

## 단계 6: (선택 사항) 소비자 확대

이 단원은 선택 사항이며 더 자세한 시나리오를 위해 소비자 코드를 확장할 수 있는 방법을 보여줍니다.

1분마다 가장 큰 판매 주문을 파악하려는 경우, 세 위치에서 StockStats 클래스를 수정하여 이 새로운 우선순위를 수용할 수 있습니다.

## 소비자를 확장하려면

1. 새 인스턴스 변수를 추가합니다.

```
// Ticker symbol of the stock that had the largest quantity of shares sold
private String largestSellOrderStock;
// Quantity of shares for the largest sell order trade
private long largestSellOrderQuantity;
```

2. 다음 코드를 addStockTrade에 추가합니다.

```
if (type == TradeType.SELL) {
    if (largestSellOrderStock == null || trade.getQuantity() >
        largestSellOrderQuantity) {
        largestSellOrderStock = trade.getTickerSymbol();
        largestSellOrderQuantity = trade.getQuantity();
    }
}
```

3. toString 메서드를 수정하여 추가 정보를 인쇄합니다.

```
public String toString() {
    return String.format(
        "Most popular stock being bought: %s, %d buys.%n" +
        "Most popular stock being sold: %s, %d sells.%n" +
        "Largest sell order: %d shares of %s.",
        getMostPopularStock(TradeType.BUY),
        getMostPopularStockCount(TradeType.BUY),
        getMostPopularStock(TradeType.SELL),
        getMostPopularStockCount(TradeType.SELL),
        largestSellOrderQuantity, largestSellOrderStock);
}
```

이제 소비자를 실행하면(생산자도 실행함을 주의) 다음과 비슷한 출력 화면이 표시되어야 합니다.



```

***** Shard shardId-000000000001 stats for last 1 minute *****
Most popular stock being bought: WMT, 27 buys.
Most popular stock being sold: PTR, 14 sells.
Largest sell order: 996 shares of BUD.
*****

```

## 다음 단계

### [단계 7: 마무리](#)

## 단계 7: 마무리

Kinesis 데이터 스트림 사용 요금을 결제하고 있으므로 작업이 완료되면 이 스트림과 해당 Amazon DynamoDB 테이블을 삭제해야 합니다. 레코드를 전송하거나 가져오지 않는 경우에도 활성 스트림에 대해 일반 요금이 부과됩니다. 이는 활성 스트림이 레코드를 가져오라는 요청과 들어오는 레코드를 지속적으로 "수신"하여 리소스를 사용하고 있기 때문입니다.

스트림과 테이블을 삭제하려면

1. 계속 실행 중인 생산자와 소비자를 종료합니다.
2. <https://console.aws.amazon.com/kinesis>에서 Kinesis 콘솔을 엽니다.
3. 이 애플리케이션에 대해 생성한 스트림을 선택합니다(`StockTradeStream`).
4. Delete Stream(스트림 삭제)을 선택합니다.
5. <https://console.aws.amazon.com/dynamodb/>에서 DynamoDB 콘솔을 엽니다.
6. `StockTradesProcessor` 테이블을 삭제합니다.

## 요약

대량의 데이터를 거의 실시간으로 처리하는 데에는 매직 코드를 작성하거나 거대한 인프라를 개발할 필요가 없습니다. 간단히 로직을 작성하여 소량의 데이터를 처리할 수 있지만(예: `processRecord(Record)` 작성), Kinesis Data Streams를 사용하여 확장할 수 있으므로 대량의 스트리밍 데이터에도 효과적입니다. Kinesis Data Streams에서 자동으로 처리되므로 처리를 확장하는 방법에 대해 걱정할 필요가 없습니다. 사용자는 스트리밍 레코드를 Kinesis Data Streams에 전송하고 수신된 각 새 레코드를 처리하는 로직을 작성하면 됩니다.

이 애플리케이션에 대한 몇 가지 잠재적 개선 사항이 있습니다.

## 모든 샤드에 대한 집계

현재는 단일 샤드에서 단일 작업자가 수신한 데이터 레코드의 집계로 인해 생성된 통계를 가져옵니다. (단일 애플리케이션에서 동시에 둘 이상의 작업자가 하나의 샤드를 처리할 수 없음) 물론, 샤드를 확장하여 샤드가 두 개 이상인 경우 모든 샤드에 대해 집계할 수 있습니다. 각 작업자의 출력이 단일 샤드가 있는 다른 스트림으로 공급되어 첫 단계의 출력을 집계하는 작업자가 처리하는 파이프라인 아키텍처를 구축하여 이를 수행할 수 있습니다. 첫 단계의 데이터가 제한(샤드마다 분당 샘플 하나)되므로 샤드 하나로 쉽게 처리할 수 있습니다.

## 처리 확장

스트림이 여러 샤드가 있도록 확장되면(여러 생산자가 데이터를 전송하기 때문) 더 많은 작업자를 추가하는 방식으로 처리가 확장됩니다. Amazon EC2 인스턴스에서 워커를 실행하고 오토 스케일링을 사용할 수 있습니다.

## Amazon S3/DynamoDB/Amazon Redshift/Storm에 대한 커넥터 사용

스트림이 지속적으로 처리됨에 따라 해당 출력이 다른 대상으로 전송될 수 있습니다. AWS는 Kinesis Data Streams를 다른 AWS 서비스 및 타사 도구와 통합하기 위한 [커넥터](#)를 제공합니다.

## 자습서: KPL 및 KCL 1.x를 사용하여 실시간 주식 데이터 처리

이 자습서의 시나리오에서는 스트림에 주식 거래를 가져와 데이터 스트림에 대한 계산을 수행하는 간단한 Amazon Kinesis Data Streams 애플리케이션을 작성합니다. 레코드의 스트림을 Kinesis Data Streams에 전송하고, 거의 실시간으로 레코드를 사용하고 처리하는 애플리케이션을 구현하는 방법에 대해 알아봅니다.

### Important

Kinesis Data Streams는 AWS 프리 티어에서 제공되지 않으므로 스트림을 생성한 후에는 Kinesis Data Streams 사용에 대한 일반 요금이 계정에 부과됩니다. 소비자 애플리케이션이 시작된 후 Amazon DynamoDB 사용량에 대해 일반 요금도 발생합니다. 소비자 애플리케이션은 DynamoDB를 사용하여 처리 상태를 추적합니다. 이 애플리케이션을 완료하면 AWS 리소스를 삭제하여 요금 발생을 중지하세요. 자세한 내용은 [단계 7: 마무리](#) 섹션을 참조하세요.

이 코드는 실제 주식 시장 데이터에는 액세스하지 않지만, 대신 주식 거래의 스트림을 시뮬레이션합니다. 2015년 2월 현재 시가 총액 상위 25개 주식에 대한 실제 시장 데이터의 시작점이 있는 임의의 주식 거래 생성기를 사용하여 이를 수행합니다. 주식 거래의 실시간 스트림에 액세스할 수 있는 경우 스트림에서 유용하고 시기 적절한 통계를 추출하고 싶을 때도 있습니다. 예를 들어, 마지막 5분 이내에 구매한

가장 인기 있는 주식을 결정하는 슬라이딩 윈도우 분석을 수행하려고 할 수 있습니다. 또는 너무 많은 판매 주문(즉, 너무 많은 공유)이 있을 때마다 알림을 원할 수도 있습니다. 이 시리즈의 코드를 확장하여 이러한 기능을 제공할 수 있습니다.

데스크톱 또는 랩톱 컴퓨터에서 이 자습서의 단계를 수행하고, 동일한 시스템에서 생산자 코드와 소비자 코드를 둘 다 실행하거나 Amazon Elastic Compute Cloud(Amazon EC2)와 같은 정의된 요건을 지원하는 모든 플랫폼을 실행할 수 있습니다.

표시된 예제는 미국 서부(오레곤) 리전을 사용하지만 이 예제는 [Kinesis Data Streams를 지원하는 모든 AWS 리전에](#) 적용됩니다.

## 작업

- [필수 조건](#)
- [1단계: 데이터 스트림 생성](#)
- [2단계: IAM 정책 및 사용자 생성](#)
- [3단계: 구현 코드 다운로드 및 빌드](#)
- [단계 4: 생산자 구현](#)
- [단계 5: 소비자 구현](#)
- [단계 6: \(선택 사항\) 소비자 확대](#)
- [단계 7: 마무리](#)

## 필수 조건

다음은 [자습서: KPL 및 KCL 1.x를 사용하여 실시간 주식 데이터 처리](#) 단원을 완료하는 데 필요한 요구 사항입니다.

### Amazon Web Services 계정

시작에 앞서 [Amazon Kinesis Data Streams 용어 및 개념](#)에 기술된 개념을 잘 알아야 합니다. 특히 스트림, 샤드, 생산자 및 소비자에 유의하십시오. [AWS CLI 설치 및 구성](#) 단원의 내용을 숙지하면 도움이 됩니다.

AWS Management Console에 액세스하려면 AWS 계정과 웹 브라우저가 필요합니다.

콘솔 액세스의 경우 IAM 사용자 이름과 암호를 사용하여 IAM 로그인 페이지에서 [AWS Management Console](#)에 로그인합니다. 프로그래밍 방식 액세스 및 장기 보안 인증에 대한 대안을 포함한 AWS 보안 인증에 대한 자세한 내용은 IAM 사용 설명서의 [AWS 보안 인증](#)을 참조하세요. AWS 계정에 로그인하는 방법에 대한 자세한 내용은 AWS 로그인 사용 설명서의 [AWS에 로그인하는 방법](#)을 참조하세요.

IAM 및 보안 키 설정 지침에 대한 자세한 내용은 [IAM 사용자 생성](#)을 참조하세요.

## 시스템 소프트웨어 요구 사항

애플리케이션을 실행하는 데 사용된 시스템에는 Java 7 이상이 설치되어 있어야 합니다. 최신 JDK(Java Development Kit)를 다운로드하고 설치하려면 [Oracle의 Java SE 설치 사이트](#)로 이동하십시오.

[Eclipse](#)와 같이 Java IDE가 있는 경우 소스 코드를 열고 편집, 빌드 및 실행할 수 있습니다.

최신 [AWS SDK for Java](#) 버전이 필요합니다. IDE로 Eclipse를 사용하는 경우 [AWS Toolkit for Eclipse](#)를 대신 설치할 수 있습니다.

소비자 애플리케이션에는 [Kinesis Client Library\(Java\)](#)의 GitHub에서 얻을 수 있는 Kinesis Client Library(KCL) 버전 1.2.1 이상이 필요합니다.

## 다음 단계

### [1단계: 데이터 스트림 생성](#)

## 1단계: 데이터 스트림 생성

[자습서: KPL 및 KCL 1.x를 사용하여 실시간 주식 데이터 처리](#)의 첫 단계에서는 후속 단계에서 사용할 스트림을 생성합니다.

스트림을 만들려면

1. AWS Management Console에 로그인하여 <https://console.aws.amazon.com/kinesis>에서 Kinesis 콘솔을 엽니다.
2. 탐색 창에서 Data Streams(데이터 스트림)를 선택합니다.
3. 탐색 모음에서 리전 선택기를 확장하고 리전을 선택합니다.
4. Create Kinesis stream(Kinesis 스트림 생성)을 선택합니다.
5. 스트림 이름을 입력합니다(예: **StockTradeStream**).
6. 샤드 수에는 **1**을 입력하고, Estimate the number of shards you'll need(필요한 샤드 수 추정)은 축소된 상태로 둡니다.
7. Create Kinesis stream(Kinesis 스트림 생성)을 선택합니다.

스트림이 생성되는 동안 Kinesis 스트림 목록 페이지에서 스트림 상태는 CREATING입니다. 스트림을 사용할 준비가 되면 상태가 ACTIVE(활성)로 변경됩니다. 스트림 이름을 선택합니다. 다음에 나타나는

페이지의 Details(세부 정보) 탭에는 스트림 구성 요약이 표시됩니다. Monitoring(모니터링) 섹션에는 스트림에 대한 모니터링 정보가 표시됩니다.

## 샤드에 대한 추가 정보

이 자습서가 아니라 실제로 Kinesis Data Streams를 사용하기 시작할 경우 스트림 생성 프로세스를 더 신중하게 계획해야 할 수 있습니다. 샤드를 프로비저닝할 때 예상되는 최대 수요를 계획해야 합니다. 이 시나리오를 예제로 사용하면, 미국 주식 시장 거래 트래픽이 낮(동부 시간) 동안 최대가 되며 수요 예상은 해당 시간대에서 샘플링되어야 합니다. 그런 다음 최대 예상 수요에 대한 프로비저닝을 선택하거나 수요 변동에 따라 스트림을 확장 및 축소합니다.

샤드는 처리 용량의 단위입니다. Kinesis 스트림 생성 페이지에서 필요한 샤드 수 추정을 확장합니다. 다음 지침에 따라 평균 레코드 크기, 초당 작성된 최대 레코드 및 사용하는 애플리케이션의 수를 입력합니다.

### 평균 레코드 크기

계산된 평균 레코드 크기의 추정입니다. 이 값을 모르는 경우 이 값에 대해 예상 최대 레코드 크기를 사용하십시오.

### 작성된 최대 레코드

데이터를 제공하는 개체의 수와 각 개체가 생성한 대략적인 초당 레코드 수를 고려하십시오. 예를 들어, 20개의 거래 서버에서 주식 거래 데이터를 가져오고 각각 초당 250개의 거래를 생성하는 경우 초당 총 거래(레코드) 수는 초당 5000개입니다.

### 사용하는 애플리케이션 수

다른 방식으로 스트림을 처리하고 다른 출력을 생성하기 위해 스트림에서 독립적으로 읽는 애플리케이션의 수입니다. 각 애플리케이션에는 다른 시스템에서 실행(즉, 클러스터에서 실행)되는 여러 인스턴스가 있을 수 있으므로 대량의 스트림을 유지할 수 있습니다.

표시된 예상 샤드 수가 현재 샤드 제한을 초과하는 경우 해당 샤드 수가 있는 스트림을 생성하기 전에 해당 제한의 증가 요청을 제출해야 할 수도 있습니다. 샤드 제한 증가를 요청하려면 [Kinesis Data Streams 제한 양식](#)을 사용하세요. 스트림 및 샤드에 대한 자세한 내용은 [스트림 생성 및 관리](#) 섹션을 참조하세요.

## 다음 단계

### [2단계: IAM 정책 및 사용자 생성](#)

## 2단계: IAM 정책 및 사용자 생성

AWS의 보안 모범 사례에 따르면 다양한 리소스에 대한 액세스를 제어하기 위해 세부적인 권한을 사용해야 합니다. AWS Identity and Access Management (IAM)를 사용하면 AWS에서 사용자와 사용자 권한을 관리할 수 있습니다. [IAM 정책](#)에는 허용된 작업과 작업이 적용되는 리소스가 명시적으로 나열됩니다.

다음은 Kinesis Data Streams 생산자 및 소비자에 대해 일반적으로 필요한 최소 권한입니다.

### 생산자

작업	리소스	용도
DescribeStream , DescribeStreamSummary , DescribeStreamConsumer	Kinesis 데이터 스트림	레코드를 쓰려고 시도하기 전에 생산자는 스트림이 존재하고 샤드가 스트림에 포함되어 있는지 여부, 그리고 스트림에 소 확인합니다.
SubscribeToShard , RegisterStreamConsumer	Kinesis 데이터 스트림	Kinesis 데이터 스트림 샤드에 소비자를 구독 및 등록합니다.
PutRecord , PutRecords	Kinesis 데이터 스트림	레코드를 Kinesis Data Streams에 씁니다.

### 소비자

작업	리소스	용도
DescribeStream	Kinesis 데이터 스트림	레코드를 읽으려고 시도하기 전에 소비자는 스트림이 존재하는지 여부와 샤드가 스트림에 포함되어 있는지 여부를 확인합니다.
GetRecords , GetShardIterator	Kinesis 데이터 스트림	Kinesis Data Streams 샤드에서 레코드를 읽습니다.
CreateTable , DescribeTable , GetItem, PutItem, Scan, UpdateItem	Amazon DynamoDB 테이블	Kinesis Client Library(KCL)를 사용하여 소비자를 개발하는 처리 상태를 추적하려면 DynamoDB 테이블에 대한 권한이 번째 소비자가 테이블을 생성합니다.

작업	리소스	용도
DeleteItem	Amazon DynamoDB 테이블	소비자가 Kinesis Data Streams 샷드에서 분할/병합 작업을 합니다.
PutMetricData	Amazon CloudWatch 로그	또한 KCL은 애플리케이션을 모니터링하는 데 유용한 지표를 합칩니다.

이 애플리케이션의 경우 위의 모든 권한을 부여하는 단일 IAM 정책을 생성합니다. 실제로 생산자에 대한 정책 하나와 소비자에 대한 정책 하나 등 정책 두 개의 생성을 고려해야 할 수 있습니다.

### IAM 정책을 만들려면

1. 새 스트림에 대한 ARN(Amazon 리소스 이름)을 찾습니다. 세부 정보 탭 상단에 스트림 ARN으로 나열된 이 ARN을 찾을 수 있습니다. ARN 형식은 다음과 같습니다.

```
arn:aws:kinesis:region:account:stream/name
```

#### 리전

리전 코드입니다(예: us-west-2). 자세한 내용은 [리전 및 가용 영역 개념](#)을 참조하십시오.

#### account

[계정 설정](#)에 표시된 대로 AWS 계정 ID입니다.

#### name

[1단계: 데이터 스트림 생성](#)의 스트림 이름입니다. 이 경우 StockTradeStream입니다.

2. 소비자가 사용할 DynamoDB 테이블에 대한 ARN을 결정합니다(첫 번째 소비자 인스턴스에서 생성됨). 형식은 다음과 같아야 합니다.

```
arn:aws:dynamodb:region:account:table/name
```

리전과 계정은 이전 단계와 동일한 위치에서 가져오지만 이때 이름은 소비자 애플리케이션에서 생성되고 사용되는 테이블의 이름입니다. 소비자가 사용하는 KCL은 애플리케이션 이름을 테이블

이름으로 사용합니다. 나중에 사용되는 애플리케이션 이름인 StockTradesProcessor를 사용합니다.

3. IAM 콘솔의 정책(<https://console.aws.amazon.com/iam/home#policies>)에서 정책 생성을 선택합니다. IAM 정책을 사용한 첫 번째 작업인 경우 시작하기, 정책 생성을 선택합니다.
4. 정책 생성기 옆의 선택을 선택합니다.
5. Amazon Kinesis를 AWS 서비스로 선택합니다.
6. DescribeStream, GetShardIterator, GetRecords, PutRecord 및 PutRecords를 허용된 작업으로 선택합니다.
7. 1단계에서 생성한 ARN을 입력합니다.
8. 다음의 각각에 대해 Add Statement(문 추가)를 사용합니다.

AWS 서비스	작업	ARN
Amazon DynamoDB	CreateTable , DeleteItem , DescribeTable , GetItem, PutItem, Scan, UpdateItem	2단계에서 생성한 ARN
Amazon CloudWatch	PutMetricData	*

별표(\*)는 ARN이 필요하지 않다고 지정할 때 사용됩니다. 이 경우에는 PutMetricData 작업이 간접적으로 호출된 CloudWatch에서 특정 리소스가 없기 때문입니다.

9. 다음 단계(Next Step)를 선택합니다.
10. Policy Name(정책 이름)을 StockTradeStreamPolicy로 변경하고, 코드를 검토한 다음 Create Policy(정책 생성)를 선택합니다.

이를 통해 생성된 정책 문서는 다음과 같아야 합니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt123",
      "Effect": "Allow",
```



```

    "Action": [
      "kinesis:DescribeStream",
      "kinesis:PutRecord",
      "kinesis:PutRecords",
      "kinesis:GetShardIterator",
      "kinesis:GetRecords",
      "kinesis:ListShards",
      "kinesis:DescribeStreamSummary",
      "kinesis:RegisterStreamConsumer"
    ],
    "Resource": [
      "arn:aws:kinesis:us-west-2:123:stream/StockTradeStream"
    ]
  },
  {
    "Sid": "Stmt234",
    "Effect": "Allow",
    "Action": [
      "kinesis:SubscribeToShard",
      "kinesis:DescribeStreamConsumer"
    ],
    "Resource": [
      "arn:aws:kinesis:us-west-2:123:stream/StockTradeStream/*"
    ]
  },
  {
    "Sid": "Stmt456",
    "Effect": "Allow",
    "Action": [
      "dynamodb:*"
    ],
    "Resource": [
      "arn:aws:dynamodb:us-west-2:123:table/StockTradesProcessor"
    ]
  },
  {
    "Sid": "Stmt789",
    "Effect": "Allow",
    "Action": [
      "cloudwatch:PutMetricData"
    ],
    "Resource": [
      "*"
    ]
  }
]

```

```

    }
  ]
}

```

## IAM 사용자 생성

1. <https://console.aws.amazon.com/iam/>에서 IAM 콘솔을 엽니다.
2. Users(사용자) 페이지에서 Add user(사용자 추가)를 선택합니다.
3. 사용자 이름(User name)에 StockTradeStreamUser를 입력합니다.
4. Access type(액세스 유형)에서 Programmatic access(프로그래밍 방식 액세스)를 선택한 다음 Next: Permissions(다음: 권한)를 선택합니다.
5. [Attach existing policies directly]를 선택합니다.
6. 생성한 정책의 이름으로 검색합니다. 정책 이름 왼쪽에 있는 확인란을 선택하고 Next: Review(다음: 검토)를 선택합니다.
7. 세부 정보와 요약 검토하고 Create user(사용자 생성)를 선택합니다.
8. Access key ID(액세스 키 ID)를 복사하고 비공개로 저장합니다. Secret access key(보안 액세스 키)에서 Show(표시)를 선택하고 키도 비공개로 저장합니다.
9. 액세스 및 보안 키를 사용자만 액세스할 수 있는 안전한 위치에 있는 로컬 파일에 붙여넣습니다. 이 애플리케이션의 경우 ~/.aws/credentials라는 파일 이름을 생성합니다(엄격한 권한 포함). 파일은 다음 형식이어야 합니다.

```

[default]
aws_access_key_id=access key
aws_secret_access_key=secret access key

```

## 사용자에게 IAM 정책 연결

1. IAM 콘솔에서 [정책](#)을 열고 정책 작업을 선택합니다.
2. StockTradeStreamPolicy 및 Attach(연결)를 선택합니다.
3. StockTradeStreamUser 및 Attach Policy(정책 연결)를 선택합니다.

## 다음 단계

### [3단계: 구현 코드 다운로드 및 빌드](#)

## 3단계: 구현 코드 다운로드 및 빌드

[the section called “자습서: KPL 및 KCL 1.x를 사용하여 실시간 주식 데이터 처리”](#)에 대한 스켈레톤 코드가 제공됩니다. 여기에는 주식 거래 스트림 수집(생산자)과 데이터 처리(소비자)를 위한 스텝 구현이 포함되어 있습니다. 다음 절차는 구현을 완료하는 방법을 보여줍니다.

구현 코드를 다운로드하고 빌드하려면

1. [소스 코드](#)를 컴퓨터에 다운로드합니다.
2. 소스 코드를 사용하여 선호하는 IDE에 프로젝트를 생성하면 제공된 디렉터리 구조를 따르게 됩니다.
3. 프로젝트에 다음 라이브러리를 추가합니다.
  - Amazon Kinesis Client Library(KCL)
  - AWS SDK
  - Apache HttpCore
  - Apache HttpClient
  - Apache Commons Lang
  - Apache Commons Logging
  - Guava(Google Core Libraries For Java)
  - Jackson Annotations
  - Jackson Core
  - Jackson Databind
  - Jackson Dataformat: CBOR
  - Joda Time
4. IDE에 따라 프로젝트가 자동으로 빌드될 수 있습니다. 그렇지 않으면 IDE에 적합한 단계를 사용하여 프로젝트를 빌드하십시오.

이러한 단계를 성공적으로 완료한 경우 이제 다음 단원([the section called “단계 4: 생산자 구현”](#))으로 이동할 준비가 되었습니다. 임의의 단계에서 빌드에 오류가 발생하는 경우 오류를 조사하고 수정한 다음 진행하십시오.

다음 단계

## 단계 4: 생산자 구현

[자습서: KPL 및 KCL 1.x를 사용하여 실시간 주식 데이터 처리](#)의 애플리케이션은 실제 주식 시장 거래 모니터링의 시나리오를 사용합니다. 다음 원칙은 이 시나리오를 생산자와 지원 코드 구조에 매핑하는 방법을 간략하게 설명합니다.

소스 코드를 참조하여 다음 정보를 검토하십시오.

### StockTrade 클래스

개별 주식 거래는 StockTrade 클래스의 인스턴스로 표시됩니다. 이 인스턴스에는 티커 기호, 가격, 공유 수, 거래 유형(구매 또는 판매), 거래를 고유하게 식별하는 ID 등의 속성이 포함됩니다. 이 클래스가 사용자를 위해 구현됩니다.

### 스트림 레코드

스트림은 레코드의 시퀀스입니다. 레코드는 JSON 형식으로 된 StockTrade 인스턴스의 직렬화입니다. 예:

```
{
  "tickerSymbol": "AMZN",
  "tradeType": "BUY",
  "price": 395.87,
  "quantity": 16,
  "id": 3567129045
}
```

### StockTradeGenerator 클래스

StockTradeGenerator에는 호출될 때마다 새로 생성된 임의의 주식 거래를 반환하는 getRandomTrade()라는 메서드가 있습니다. 이 클래스가 사용자를 위해 구현됩니다.

### StockTradesWriter 클래스

생산자의 main 메서드인 StockTradesWriter는 계속적으로 임의의 거래를 검색하고 다음 작업을 수행하여 Kinesis Data Streams에 전송합니다.

1. 스트림 이름과 리전 이름을 입력으로 읽습니다.
2. AmazonKinesisClientBuilder를 생성합니다.
3. 클라이언트 빌더를 사용하여 리전, 자격 증명 및 클라이언트 구성을 설정합니다.
4. 클라이언트 빌더를 사용하여 AmazonKinesis 클라이언트를 빌드합니다.

5. 스트림의 존재 여부와 활성 상태 여부를 확인합니다. 그렇지 않은 경우 오류로 종료됩니다.
6. 연속 루프에서 `StockTradeGenerator.getRandomTrade()` 메서드를 호출하고 `sendStockTrade` 메서드를 호출하여 100밀리초마다 거래를 스트림으로 전송합니다.

`sendStockTrade` 클래스의 `StockTradesWriter` 메서드에는 다음 코드가 있습니다.

```
private static void sendStockTrade(StockTrade trade, AmazonKinesis kinesisClient,
String streamName) {
    byte[] bytes = trade.toJsonAsBytes();
    // The bytes could be null if there is an issue with the JSON serialization by
the Jackson JSON library.
    if (bytes == null) {
        LOG.warn("Could not get JSON bytes for stock trade");
        return;
    }

    LOG.info("Putting trade: " + trade.toString());
    PutRecordRequest putRecord = new PutRecordRequest();
    putRecord.setStreamName(streamName);
    // We use the ticker symbol as the partition key, explained in the Supplemental
Information section below.
    putRecord.setPartitionKey(trade.getTickerSymbol());
    putRecord.setData(ByteBuffer.wrap(bytes));

    try {
        kinesisClient.putRecord(putRecord);
    } catch (AmazonClientException ex) {
        LOG.warn("Error sending record to Amazon Kinesis.", ex);
    }
}
```

다음 코드 세부 분석을 참조하십시오.

- `PutRecord` API에는 바이트 어레이가 필요하며, `trade`를 JSON 형식으로 변환해야 합니다. 이 한 줄의 코드는 다음 작업을 수행합니다.

```
byte[] bytes = trade.toJsonAsBytes();
```

- 거래를 전송하기 전에 새 `PutRecordRequest` 인스턴스(이 경우 `putRecord`라고 함)를 생성합니다.

```
PutRecordRequest putRecord = new PutRecordRequest();
```

각 PutRecord 호출에는 스트림 이름, 파티션 키 및 데이터 BLOB가 필요합니다. 다음 코드는 putRecord 메서드를 사용하여 setXxxx() 객체의 이러한 필드를 채웁니다.

```
putRecord.setStreamName(streamName);
putRecord.setPartitionKey(trade.getTickerSymbol());
putRecord.setData(ByteBuffer.wrap(bytes));
```

이 예제는 특정 샤드에 레코드를 매핑하는 주식 티켓을 파티션 키로 사용합니다. 실제로 레코드가 스트림에 대해 균등하게 분산되도록 샤드당 수백 개 또는 수천 개의 파티션 키가 있어야 합니다. 스트림에 데이터를 추가하는 방법에 대한 자세한 내용은 [스트림에 데이터 추가](#) 단원을 참조하십시오.

이제 putRecord를 클라이언트에 전송할 준비가 되었습니다(put 작업).

```
kinesisClient.putRecord(putRecord);
```

- 오류 확인과 로깅 기능은 항상 유용한 추가 기능입니다. 이 코드는 오류 조건을 기록합니다.

```
if (bytes == null) {
    LOG.warn("Could not get JSON bytes for stock trade");
    return;
}
```

put넣기 작업에 try/catch 블록을 추가합니다.

```
try {
    kinesisClient.putRecord(putRecord);
} catch (AmazonClientException ex) {
    LOG.warn("Error sending record to Amazon Kinesis.", ex);
}
```

이렇게 하는 이유는 네트워크 오류로 인해 또는 처리량 제한에 도달하여 병목 현상이 발생한 스트림으로 인해 Kinesis Data Streams put 작업이 실패할 수 있기 때문입니다. 데이터 손실을 방지하기 위해 단순 재시도를 사용하는 것과 같이 put넣기 작업에 대한 재시도 정책을 신중히 고려하는 것이 좋습니다.

- 상태 로깅은 유용하지만 선택 사항입니다.

```
LOG.info("Putting trade: " + trade.toString());
```

여기에 표시된 생산자는 Kinesis Data Streams API 단일 레코드 기능인 PutRecord를 사용합니다. 실제로 개별 생산자가 많은 레코드를 생성하는 경우 PutRecords의 여러 레코드 기능을 사용하고 레코드의 배치를 한 번에 전송하는 것이 더 효율적인 경우가 많습니다. 자세한 내용은 [스트림에 데이터 추가](#) 섹션을 참조하세요.

생산자를 실행하려면

1. 앞에서(IAM 사용자를 생성할 때) 검색한 액세스 키 및 보안 키 페어가 ~/.aws/credentials 파일에 저장되었는지 확인합니다.
2. 다음과 같은 인수를 사용하여 StockTradeWriter 클래스를 실행합니다.

```
StockTradeStream us-west-2
```

us-west-2 이외의 리전에 스트림을 생성한 경우 여기에 해당 리전을 대신 지정해야 합니다.

다음과 유사한 출력 화면이 표시되어야 합니다.

```
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
sendStockTrade
INFO: Putting trade: ID 8: SELL 996 shares of BUD for $124.18
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
sendStockTrade
INFO: Putting trade: ID 9: BUY 159 shares of GE for $20.85
Feb 16, 2015 3:53:01 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
sendStockTrade
INFO: Putting trade: ID 10: BUY 322 shares of WMT for $90.08
```

이제 주식 거래 스트림이 Kinesis Data Streams에서 수집됩니다.

다음 단계

[단계 5: 소비자 구현](#)

## 단계 5: 소비자 구현

[자습서: KPL 및 KCL 1.x를 사용하여 실시간 주식 데이터 처리](#)의 소비자 애플리케이션은 에서 만든 주식 거래 스트림을 계속 처리합니다. 그런 다음 1분마다 매매된 가장 인기 있는 주식들을 출력합니다. 애플리케이션은 소비자 앱에 공통적인 과중한 업무를 많이 수행하는 Kinesis Client Library(KCL)를 기반으로 하여 빌드됩니다. 자세한 내용은 [KCL 1.x 소비자 개발](#) 섹션을 참조하세요.

소스 코드를 참조하여 다음 정보를 검토하십시오.

### StockTradesProcessor 클래스

다음 작업을 수행하는 소비자의 기본 클래스가 제공됩니다.

- 인수로 전달된 애플리케이션, 스트림 및 리전 이름을 읽습니다.
- `~/.aws/credentials`에서 자격 증명을 읽습니다.
- `RecordProcessorFactory` 인스턴스에 의해 구현된 `RecordProcessor`의 서버 인스턴스를 제공하는 `StockTradeRecordProcessor` 인스턴스를 생성합니다.
- `RecordProcessorFactory` 인스턴스가 있는 KCL 워커와 스트림 이름, 보안 인증 및 애플리케이션 이름이 포함된 표준 구성을 생성합니다.
- 워커는 각 샤드(이 소비자 인스턴스에 할당된 샤드)에 대해 새 스레드를 생성하며, 이 스레드는 Kinesis Data Streams에서 계속 반복적으로 레코드를 읽습니다. 그런 다음 `RecordProcessor` 인스턴스를 호출하여 수신한 각 일괄 레코드를 처리합니다.

### StockTradeRecordProcessor 클래스

`RecordProcessor` 인스턴스를 구현하고 `initialize`, `processRecords` 및 `shutdown`의 세 가지 필수 메서드를 구현합니다.

Kinesis Client Library는 `initialize` 및 `shutdown`을 사용하여 레코드 프로세서에 레코드 수신을 시작할 준비가 될 시점과 레코드 수신을 중지해야 할 시점을 각각 알리므로, 레코드 프로세스가 모든 애플리케이션별 설정 및 종료 작업을 수행할 수 있습니다. 이에 대한 코드가 제공됩니다. `processRecords` 메서드에서 기본 처리가 발생하며, 각 레코드에 대해 `processRecord`를 사용합니다. 이 후자의 메서드는 사용자에게 대해 대부분의 빈 스켈레톤 코드로 제공되어 향후 설명할 다음 단계에서 구현됩니다.

원래 소스 코드에서 비어 있는 `processRecord: reportStats`, and `resetStats`에 대한 지원 메서드의 구현에 유의하십시오.

`processRecords` 메서드가 구현되며 다음 단계를 수행합니다.



- 전달된 각 레코드의 경우 processRecord를 호출합니다.
- 마지막 보고 이후 1분 이상이 경과된 경우 최신 통계를 인쇄하는 reportStats()를 호출한 후 통계를 지우는 resetStats()를 호출하여 다음 간격에 새 레코드만 포함되도록 합니다.
- 다음 보고 시간을 설정합니다.
- 마지막 체크포인트 이후 1분 이상이 경과된 경우 checkpoint()를 호출합니다.
- 다음 검사 시간을 설정합니다.

이 메서드는 보고 및 검사 속도에 대해 60초 간격을 사용합니다. 검사에 대한 자세한 내용은 [소비자에 대한 추가 정보](#) 단원을 참조하십시오.

### StockStats 클래스

이 클래스는 시간에 따른 가장 인기 있는 주식에 대한 통계 추적 및 데이터 보존을 제공합니다. 다음 메서드가 포함된 이 코드가 제공됩니다.

- addStockTrade(StockTrade): 지정된 StockTrade를 실행 중인 통계에 삽입합니다.
- toString(): 형식이 지정된 문자열로 통계를 반환합니다.

이 클래스는 각 주식에 대한 총 거래 수의 실행 개수와 최대 개수를 유지하여 가장 인기 있는 주식을 추적합니다. 그리고 주식 거래가 발생할 때마다 이러한 계수가 업데이트됩니다.

다음 단계에 표시된 대로 StockTradeRecordProcessor 클래스의 메서드에 코드를 추가합니다.

### 소비자를 구현하려면

1. 정확한 크기의 processRecord 객체를 인스턴스화하고, 해당 객체에 레코드 데이터를 추가하고, 문제가 있는 경우 경고를 기록하여 StockTrade 메서드를 구현합니다.

```
StockTrade trade = StockTrade.fromJsonAsBytes(record.getData().array());
if (trade == null) {
    LOG.warn("Skipping record. Unable to parse record into StockTrade. Partition
    Key: " + record.getPartitionKey());
    return;
}
stockStats.addStockTrade(trade);
```

2. 간단한 reportStats 메서드를 구현합니다. 기본 설정에 대한 출력 형식을 자유롭게 수정합니다.

```
System.out.println("***** Shard " + kinesisShardId + " stats for last 1 minute
*****\n" +
```

```
stockStats + "\n" +
"*****\n");
```

3. 마지막으로 `resetStats` 메서드를 구현합니다. 그러면 새 `stockStats` 인스턴스가 생성됩니다.

```
stockStats = new StockStats();
```

## 소비자를 실행하려면

1. 에서 작성한 생산자를 실행하여 시뮬레이션된 주식 거래 레코드를 스트림에 첨가합니다.
2. 앞에서(IAM 사용자를 생성할 때) 검색한 액세스 키 및 보안 키 페어가 `~/.aws/credentials` 파일에 저장되었는지 확인합니다.
3. 다음과 같은 인수를 사용하여 `StockTradesProcessor` 클래스를 실행합니다.

```
StockTradesProcessor StockTradeStream us-west-2
```

`us-west-2` 이외의 리전에 스트림을 생성한 경우 여기에 해당 리전을 대신 지정해야 합니다.

1분 후 다음과 같은 출력이 표시되어야 하며, 그 이후로 매분마다 새로 고칩니다.

```
***** Shard shardId-000000000001 stats for last 1 minute *****
Most popular stock being bought: WMT, 27 buys.
Most popular stock being sold: PTR, 14 sells.
*****
```

## 소비자에 대한 추가 정보

[KCL 1.x 소비자 개발](#) 및 다른 부분에서 설명하는 Kinesis Client Library의 이점을 잘 알고 있는 경우 여기에서 Kinesis Client Library를 사용해야 하는 이유에 대해 궁금할 수 있습니다. 단일 샤드 스트림과 단일 소비자 인스턴스만 사용하여 처리하는 경우에도 KCL을 사용하여 소비자를 구현하는 것이 훨씬 더 쉽습니다. 생산자 단원의 코드 구현 단계를 소비자와 비교하면 소비자 구현이 비교적 쉽다는 것을 알 수 있습니다. 이는 대부분 KCL이 제공하는 서비스로 인한 것입니다.

이 애플리케이션에서는 개별 레코드를 처리할 수 있는 레코드 프로세서 클래스를 구현하는 것에 중점을 둡니다. Kinesis Data Streams에서 레코드를 가져오는 방식에 대해서는 걱정할 필요가 없습니다. KCL은 새 레코드가 사용 가능할 때마다 레코드를 가져오고 레코드 프로세서를 간접적으로 호출합니다. 또한 얼마나 많은 샤드와 소비자 인스턴스가 있는지에 대해서도 걱정할 필요가 없습니다. 스트림이

확장되면 둘 이상의 샤드 또는 소비자 인스턴스를 처리하기 위해 애플리케이션을 다시 작성할 필요가 없습니다.

검사라는 용어는 지금까지 사용 및 처리된 데이터 레코드까지 스트림의 지점을 기록하는 것을 의미하므로, 애플리케이션이 충돌할 경우 스트림의 시작이 아닌 해당 지점에서 스트림을 읽습니다. 검사 주체와 다양한 디자인 패턴 및 이에 대한 모범 사례는 이 장의 범위를 벗어나지만, 프로덕션 환경에서 직면할 수 있는 사항입니다.

에서 배운 바와 같이, Kinesis Data Streams API의 put 작업은 파티션 키를 입력으로 사용합니다. Kinesis Data Streams는 파티션 키를 메커니즘으로 사용하여 레코드를 여러 샤드로 분할합니다(스트림에 샤드가 여러 개 있는 경우). 동일한 파티션 키는 항상 동일한 샤드에 라우팅됩니다. 이를 통해 특정 샤드를 처리하는 소비자는 동일한 파티션 키가 있는 레코드는 해당 소비자에게만 전송되며, 다른 소비자에 전송될 수 없다는 가정에 기반하여 설계할 수 있습니다. 따라서 소비자의 작업자는 필요한 데이터가 누락될 수 있다는 걱정 없이 동일한 파티션 키가 있는 모든 레코드를 집계할 수 있습니다.

이 애플리케이션에서 소비자의 레코드 처리는 집약적이지 않으므로 샤드 하나를 사용하고 KCL 스레드와 동일한 스레드에서 처리할 수 있습니다. 하지만 실제로 먼저 샤드 수를 확장하는 것을 고려해 보십시오. 일부 경우에는 처리를 다른 스레드로 전환하거나, 레코드 처리가 집약적으로 예상될 경우 스레드 풀을 사용할 수 있습니다. 이러한 방식으로 KCL은 다른 스레드가 레코드를 병렬로 처리하는 동안 새 레코드를 더욱 신속하게 가져올 수 있습니다. 멀티스레드 디자인은 중요한 요소이며 고급 기술을 사용하여 접근해야 합니다. 따라서 샤드 수를 늘리는 것은 일반적으로 확장하기 위한 가장 효과적이고 쉬운 방식입니다.

다음 단계

## [단계 6: \(선택 사항\) 소비자 확대](#)

### 단계 6: (선택 사항) 소비자 확대

[자습서: KPL 및 KCL 1.x를 사용하여 실시간 주식 데이터 처리](#)의 애플리케이션은 이미 해당 용도에 충분할 수 있습니다. 이 단원은 선택 사항이며 더 자세한 시나리오를 위해 소비자 코드를 확장할 수 있는 방법을 보여줍니다.

1분마다 가장 큰 판매 주문을 파악하려는 경우, 세 위치에서 StockStats 클래스를 수정하여 이 새로운 우선순위를 수용할 수 있습니다.

소비자를 확장하려면

1. 새 인스턴스 변수를 추가합니다.

```
// Ticker symbol of the stock that had the largest quantity of shares sold
private String largestSellOrderStock;
// Quantity of shares for the largest sell order trade
private long largestSellOrderQuantity;
```

2. 다음 코드를 addStockTrade에 추가합니다.

```
if (type == TradeType.SELL) {
    if (largestSellOrderStock == null || trade.getQuantity() >
largestSellOrderQuantity) {
        largestSellOrderStock = trade.getTickerSymbol();
        largestSellOrderQuantity = trade.getQuantity();
    }
}
```

3. toString 메서드를 수정하여 추가 정보를 인쇄합니다.

```
public String toString() {
    return String.format(
        "Most popular stock being bought: %s, %d buys.%n" +
        "Most popular stock being sold: %s, %d sells.%n" +
        "Largest sell order: %d shares of %s.",
        getMostPopularStock(TradeType.BUY),
        getMostPopularStockCount(TradeType.BUY),
        getMostPopularStock(TradeType.SELL),
        getMostPopularStockCount(TradeType.SELL),
        largestSellOrderQuantity, largestSellOrderStock);
}
```

이제 소비자를 실행하면(생산자도 실행함을 주의) 다음과 비슷한 출력 화면이 표시되어야 합니다.

```
***** Shard shardId-000000000001 stats for last 1 minute *****
Most popular stock being bought: WMT, 27 buys.
Most popular stock being sold: PTR, 14 sells.
Largest sell order: 996 shares of BUD.
*****
```

다음 단계

[단계 7: 마무리](#)

## 단계 7: 마무리

Kinesis 데이터 스트림 사용 요금을 결제하고 있으므로 작업이 완료되면 이 스트림과 해당 Amazon DynamoDB 테이블을 삭제해야 합니다. 레코드를 전송하거나 가져오지 않는 경우에도 활성 스트림에 대해 일반 요금이 부과됩니다. 이는 활성 스트림이 레코드를 가져오라는 요청과 들어오는 레코드를 지속적으로 "수신"하여 리소스를 사용하고 있기 때문입니다.

스트림과 테이블을 삭제하려면

1. 계속 실행 중인 생산자와 소비자를 종료합니다.
2. <https://console.aws.amazon.com/kinesis>에서 Kinesis 콘솔을 엽니다.
3. 이 애플리케이션에 대해 생성한 스트림을 선택합니다(`StockTradeStream`).
4. `Delete Stream`(스트림 삭제)을 선택합니다.
5. <https://console.aws.amazon.com/dynamodb/>에서 DynamoDB 콘솔을 엽니다.
6. `StockTradesProcessor` 테이블을 삭제합니다.

### 요약

대량의 데이터를 거의 실시간으로 처리하는 데에는 매직 코드를 작성하거나 거대한 인프라를 개발할 필요가 없습니다. 간단히 로직을 작성하여 소량의 데이터를 처리할 수 있지만(예: `processRecord(Record)` 작성), Kinesis Data Streams를 사용하여 확장할 수 있으므로 대량의 스트리밍 데이터에도 효과적입니다. Kinesis Data Streams에서 자동으로 처리되므로 처리를 확장하는 방법에 대해 걱정할 필요가 없습니다. 사용자는 스트리밍 레코드를 Kinesis Data Streams에 전송하고 수신된 각 새 레코드를 처리하는 로직을 작성하면 됩니다.

이 애플리케이션에 대한 몇 가지 잠재적 개선 사항이 있습니다.

#### 모든 샤드에 대한 집계

현재는 단일 샤드에서 단일 작업자가 수신한 데이터 레코드의 집계로 인해 생성된 통계를 가져옵니다. (단일 애플리케이션에서 동시에 둘 이상의 작업자가 하나의 샤드를 처리할 수 없음) 물론, 샤드를 확장하여 샤드가 두 개 이상인 경우 모든 샤드에 대해 집계할 수 있습니다. 각 작업자의 출력이 단일 샤드가 있는 다른 스트림으로 공급되어 첫 단계의 출력을 집계하는 작업자가 처리하는 파이프라인 아키텍처를 구축하여 이를 수행할 수 있습니다. 첫 단계의 데이터가 제한(샤드마다 분당 샘플 하나)되므로 샤드 하나로 쉽게 처리할 수 있습니다.

## 처리 확장

스트림이 여러 샤드가 있도록 확장되면(여러 생산자가 데이터를 전송하기 때문) 더 많은 작업자를 추가하는 방식으로 처리가 확장됩니다. Amazon EC2 인스턴스에서 워커를 실행하고 오토 스케일링을 사용할 수 있습니다.

### Amazon S3/DynamoDB/Amazon Redshift/Storm에 대한 커넥터 사용

스트림이 지속적으로 처리됨에 따라 해당 출력이 다른 대상으로 전송될 수 있습니다. AWS는 Kinesis Data Streams를 다른 AWS 서비스 및 타사 도구와 통합하기 위한 [커넥터](#)를 제공합니다.

## 다음 단계

- Kinesis Data Streams API 작업 사용에 대한 자세한 내용은 [AWS SDK for Java와 Amazon Kinesis Data Streams API를 사용하여 생산자 개발](#), [AWS SDK for Java를 사용하여 공유 처리량으로 사용자 지정 소비자 개발](#) 및 [스트림 생성 및 관리](#) 섹션을 참조하세요.
- Kinesis Client Library에 대한 자세한 내용은 [KCL 1.x 소비자 개발](#) 섹션을 참조하세요.
- 애플리케이션을 최적화하는 방법에 대한 자세한 내용은 [고급 주제](#) 단원을 참조하십시오.

## 자습서: Flink 애플리케이션용 Managed Service for Apache Flink를 사용하여 실시간 주식 데이터 분석

이 자습서의 시나리오에서는 스트림에 주식 거래를 가져와 데이터 스트림에 대한 계산을 수행하는 간단한 [Amazon Managed Service for Apache Flink](#) 애플리케이션을 작성합니다. 레코드의 스트림을 Kinesis Data Streams에 전송하고, 거의 실시간으로 레코드를 사용하고 처리하는 애플리케이션을 구현하는 방법에 대해 알아봅니다.

Flink 애플리케이션용 Managed Service for Apache Flink를 사용하면 Java 또는 Scala로 스트리밍 데이터를 처리하고 분석할 수 있습니다. 이 서비스를 사용하면 스트리밍 소스에 대해 Java 또는 Scala 코드를 작성하고 실행하여 시계열 분석을 수행하고, 실시간 대시보드를 공급하고, 실시간 지표를 생성할 수 있습니다.

[Apache Flink](#) 기반 오픈 소스 라이브러리를 사용하여 Managed Service for Apache Flink에서 Flink 애플리케이션을 구축할 수 있습니다. Apache Flink는 데이터 스트림을 처리하기 위한 인기 있는 프레임워크 및 엔진입니다.

**⚠ Important**

데이터 스트림 2개와 애플리케이션 1개를 생성한 후에는 프리 티어를 사용할 수 없으므로 Kinesis Data Streams 및 Apache Flink Managed Service 사용에 대해서는 소액의 요금이 계정에 부과됩니다. AWS 이 애플리케이션 사용을 마치면 리소스를 삭제하여 요금 발생을 중지하십시오. AWS

이 코드는 실제 주식 시장 데이터에는 액세스하지 않지만, 대신 주식 거래의 스트림을 시뮬레이션합니다. 이 작업은 임의의 주식 거래 생성기를 통해 수행됩니다. 주식 거래의 실시간 스트림에 액세스할 수 있는 경우 스트림에서 유용하고 시기 적절한 통계를 추출하고 싶을 때도 있습니다. 예를 들어, 마지막 5분 이내에 구매한 가장 인기 있는 주식을 결정하는 슬라이딩 윈도우 분석을 수행하려고 할 수 있습니다. 또는 너무 많은 판매 주문(즉, 너무 많은 공유)이 있을 때마다 알림을 원할 수도 있습니다. 이 시리즈의 코드를 확장하여 이러한 기능을 제공할 수 있습니다.

표시된 예제는 미국 서부(오레곤) 리전을 사용하지만 이 예제는 [Managed Service for Apache Flink를 지원하는 모든 AWS 리전](#)에 적용됩니다.

**Tasks**

- [연습 완료를 위한 필수 조건](#)
- [1단계: AWS 계정 설정 및 관리자 사용자 생성](#)
- [2단계: AWS Command Line Interface \(AWS CLI\) 설정](#)
- [3단계: Flink 애플리케이션용 Managed Service for Apache Flink 생성 및 실행](#)

**연습 완료를 위한 필수 조건**

이 가이드의 단계를 완료하려면 다음이 필요합니다.

- [Java Development Kit\(JDK\)](#) 버전 8. JAVA\_HOME 환경 변수가 JDK 설치 위치를 가리키도록 설정합니다.
- 애플리케이션을 개발하고 컴파일하려면 개발 환경(예: [Eclipse Java Neon](#) 또는 [IntelliJ Idea](#))을 사용하는 것이 좋습니다.
- [Git 클라이언트](#). 아직 설치하지 않았다면 Git 클라이언트를 설치합니다.
- [Apache Maven 컴파일러 플러그인](#). Maven이 해당 작업 경로에 있어야 합니다. Apache Maven 설치를 테스트하려면 다음을 입력하십시오.

```
$ mvn -version
```

시작하려면 [1단계: AWS 계정 설정 및 관리자 사용자 생성](#) 섹션으로 이동하십시오.

## 1단계: AWS 계정 설정 및 관리자 사용자 생성

Flink 애플리케이션용 Amazon Managed Service for Apache Flink를 처음 사용하는 경우 먼저 다음 작업을 완료해야 합니다.

1. [가입: AWS](#)
2. [IAM 사용자 생성](#)

### 가입: AWS

Amazon Web Services (AWS) 에 가입하면 Apache Flink용 Amazon Managed Service를 포함하여 내 AWS모든 서비스에 AWS 계정이 자동으로 가입됩니다. 사용자에게는 사용한 서비스에 대해서만 요금이 청구됩니다.

Managed Service for Apache Flink에서는 사용한 리소스에 대해서만 비용을 지불합니다. AWS 를 처음 사용하는 고객인 경우 Managed Service for Apache Flink를 무료로 시작할 수 있습니다. 자세한 내용은 [AWS 프리 티어](#)를 참조하세요.

이미 AWS 계정이 있다면 다음 작업으로 건너뛰십시오. AWS 계정이 없다면 다음 단계에 따라 계정을 생성합니다.

계정을 AWS 만들려면

1. <https://portal.aws.amazon.com/billing/signup>을 여세요.
2. 온라인 지시 사항을 따르세요.

등록 절차 중에는 전화를 받고 키패드로 인증 코드를 입력하는 과정이 있습니다.

에 AWS 계정가입하면 AWS 계정 루트 사용자a가 생성됩니다. 루트 사용자에게는 계정의 모든 AWS 서비스 및 리소스 액세스 권한이 있습니다. 보안 모범 사례로, 사용자에게 관리자 액세스 권한을 할당하고 루트 사용자 [액세스가 필요한 작업을 수행할 때는 루트 사용자만](#) 사용하십시오.

다음 작업에 필요하므로 AWS 계정 ID를 기록해 두십시오.



## IAM 사용자 생성

Apache Flink용 Amazon Managed Service와 같은 서비스의 경우 액세스 시 자격 증명을 제공해야 합니다. AWS를 통해 서비스가 소유한 리소스에 액세스할 수 있는 권한이 있는지를 확인합니다. 비밀번호를 AWS Management Console 입력해야 합니다.

AWS 계정의 액세스 키를 생성하여 AWS Command Line Interface (AWS CLI) 또는 API에 액세스할 수 있습니다. 하지만 AWS 계정의 자격 증명을 AWS 사용하여 액세스하는 것은 권장하지 않습니다. 대신 AWS Identity and Access Management (IAM) 을 사용하는 것이 좋습니다. IAM 사용자를 생성하여 관리자 권한과 함께 IAM 그룹에 추가한 다음, 생성한 IAM 사용자에게 관리자 권한을 부여하십시오. 그러면 특정 URL이나 그 IAM 사용자의 자격 증명을 사용하여 AWS 에 액세스할 수 있습니다.

AWS가입했지만 아직 IAM 사용자를 만들지 않았다면 IAM 콘솔을 사용하여 새로 만들 수 있습니다.

이 안내서의 시작하기 연습에서는 관리자 권한이 있는 사용자(adminuser)가 있다고 가정합니다. 절차에 따라 계정에서 adminuser를 만듭니다.

관리자 그룹을 생성하려면

1. [에 AWS Management Console 로그인하고 https://console.aws.amazon.com/iam/ 에서 IAM 콘솔을 엽니다.](https://console.aws.amazon.com/iam/)
2. 탐색 창에서 그룹을 선택한 다음, 새 그룹 생성을 선택합니다.
3. 그룹 이름에 **Administrators**와 같은 그룹 이름을 입력한 후 다음 단계를 선택합니다.
4. 정책 목록에서 해당 정책 옆의 AdministratorAccess확인란을 선택합니다. 필터 메뉴와 검색 상자를 사용하여 정책 목록을 필터링합니다.
5. 다음 단계를 선택한 다음 그룹 생성을 선택합니다.

그룹 이름 아래에 새 그룹이 나열됩니다.

자신을 위한 IAM 사용자를 생성하려면 관리자 그룹에 사용자를 추가하고 암호를 생성합니다.

1. 탐색 창에서 Users와 Add user를 차례대로 선택합니다.
2. 사용자 이름 상자에 사용자 이름을 입력합니다.
3. 프로그래밍 방식 액세스와 AWS Management Console 액세스를 모두 선택합니다.
4. 다음: 권한을 선택합니다.

- Administrators(관리자) 그룹 옆의 확인란을 선택합니다. 그런 다음 다음: 검토(Next: Review)를 선택합니다.
- 사용자 생성을 선택합니다.

새 IAM 사용자로 로그인하려면

- 에서 로그아웃합니다 AWS Management Console.
- 다음 URL 형식을 사용하여 콘솔에 로그인합니다.

`https://aws_account_number.signin.aws.amazon.com/console/`

`aws_account_number#` 하이픈이 없는 계정 ID입니다. AWS ## ## AWS ## ID#  
`1234-5678-9012# ## aws_account_number# # #####. 123456789012` [계정 번호를 찾는 방법에 대한 자세한 내용은 IAM 사용 설명서의 계정 ID 및 별칭을 참조하십시오.](#) AWS

- 방금 생성한 IAM 사용자 이름과 암호를 입력합니다. 로그인하면 탐색 모음에 `your_user_name @ your_aws_account_id`가 표시됩니다.

#### Note

로그인 페이지의 URL에 계정 ID가 포함되지 않도록 하려면 AWS 계정 별칭을 생성하면 됩니다.

계정 별칭을 만들거나 제거하려면

- <https://console.aws.amazon.com/iam/>에서 IAM 콘솔을 엽니다.
- 탐색 창에서 대시보드를 선택합니다.
- IAM 사용자 로그인 링크를 찾습니다.
- 별칭을 생성하려면 사용자 지정을 선택합니다. 별칭에 사용할 이름을 입력한 후 예, 생성을 선택합니다.
- 별칭을 제거하려면 사용자 지정을 선택한 다음 예, 삭제를 선택합니다. 로그인 URL은 계정 ID를 사용하는 것으로 되돌아갑니다. AWS

계정 별칭 생성 후에는 다음 URL에서 로그인하세요.

`https://your_account_alias.signin.aws.amazon.com/console/`

사용자 계정의 IAM 사용자 로그인 링크를 확인하려면 IAM 콘솔을 열고 대시보드의 IAM 사용자 로그인 링크에서 확인합니다.

IAM에 대한 자세한 내용은 다음을 참조하십시오.

- [AWS Identity and Access Management \(IAM\)](#)
- [시작하기](#)
- [IAM 사용자 가이드](#)

다음 단계

[2단계: AWS Command Line Interface \(\)AWS CLI설정](#)

## 2단계: AWS Command Line Interface ()AWS CLI설정

이 단계에서는 Flink 애플리케이션용 Apache Flink용 Amazon 매니지드 서비스와 함께 사용하도록 다운로드하고 구성합니다. AWS CLI

### Note

이 가이드의 시작하기 연습에서는 해당 계정에서 관리자 자격 증명(adminuser)을 사용하여 작업을 수행한다고 가정합니다.

### Note

이미 AWS CLI 설치한 경우 최신 기능을 사용하려면 업그레이드해야 할 수 있습니다. 자세한 내용은 AWS Command Line Interface 사용 [설명서의 AWS 명령줄 인터페이스 설치를 참조](#)하십시오. 의 버전을 확인하려면 다음 명령을 실행합니다. AWS CLI

```
aws --version
```

이 자습서의 연습에는 다음 AWS CLI 버전 이상이 필요합니다.

```
aws-cli/1.16.63
```

## 설정하려면 AWS CLI

1. AWS CLI를 다운로드하고 구성합니다. 관련 지침은 [AWS Command Line Interface 사용 설명서](#)에서 다음 토픽을 참조하세요.
  - [AWS Command Line Interface설치](#)
  - [AWS CLI구성](#)
2. AWS CLI 구성 파일에 관리자 사용자의 이름이 지정된 프로필을 추가합니다. 명령을 실행할 때 이 프로필을 사용합니다. AWS CLI 프로파일 명명에 대한 자세한 설명은 [AWS Command Line Interface 사용자 가이드의 프로파일 명명](#)을 참조하십시오.

```
[profile adminuser]
aws_access_key_id = adminuser access key ID
aws_secret_access_key = adminuser secret access key
region = aws-region
```

사용 가능한 AWS 지역 목록은 의 [AWS 지역 및 엔드포인트](#)를 참조하십시오. Amazon Web Services 일반 참조

3. 명령 프롬프트에서 다음 help 명령을 입력하여 설정을 확인하십시오:

```
aws help
```

AWS 계정을 설정하고 나면 샘플 애플리케이션을 구성하고 end-to-end 설정을 테스트하는 다음 연습을 시도할 수 있습니다. AWS CLI

## 다음 단계

### [3단계: Flink 애플리케이션용 Managed Service for Apache Flink 생성 및 실행](#)

## 3단계: Flink 애플리케이션용 Managed Service for Apache Flink 생성 및 실행

이 연습에서는 데이터 스트림을 소스 및 싱크로 사용하여 Flink 애플리케이션용 Managed Service for Apache Flink를 생성합니다.

이 섹션은 다음 주제를 포함합니다:

- [2개의 Amazon Kinesis Data Streams 생성](#)

- [샘플 레코드를 입력 스트림에 쓰기](#)
- [Apache Flink 스트리밍 Java 코드 다운로드 및 검사](#)
- [애플리케이션 코드 컴파일](#)
- [Apache Flink 스트리밍 Java 코드 업로드](#)
- [Managed Service for Apache Flink 애플리케이션 생성 및 실행](#)

## 2개의 Amazon Kinesis Data Streams 생성

이 연습을 위해 Flink 애플리케이션용 Managed Service for Apache Flink를 생성하기 전에 2개의 Kinesis 데이터 스트림(ExampleInputStream 및 ExampleOutputStream)을 생성합니다. 이 애플리케이션은 애플리케이션 소스 및 대상 스트림에 대해 이러한 스트림을 사용합니다.

Amazon Kinesis 콘솔 또는 다음 AWS CLI 명령을 사용하여 이러한 스트림을 생성할 수 있습니다. 콘솔 지침은 [데이터 스트림 만들기 및 업데이트](#)를 참조하십시오.

데이터 스트림 (AWS CLI)을 생성하려면

1. 첫 번째 스트림 (ExampleInputStream) 을 생성하려면 다음 Amazon Kinesis 명령을 create-stream AWS CLI 사용하십시오.

```
$ aws kinesis create-stream \  
--stream-name ExampleInputStream \  
--shard-count 1 \  
--region us-west-2 \  
--profile adminuser
```

2. 애플리케이션에서 출력을 쓰는 데 사용하는 두 번째 스트림을 생성하려면 동일한 명령을 실행하여 스트림 명칭을 ExampleOutputStream으로 변경합니다.

```
$ aws kinesis create-stream \  
--stream-name ExampleOutputStream \  
--shard-count 1 \  
--region us-west-2 \  
--profile adminuser
```

## 샘플 레코드를 입력 스트림에 쓰기

이 섹션에서는 Python 스크립트를 사용하여 애플리케이션에서 처리할 샘플 레코드를 스트림에 쓰기 합니다.

### Note

이 섹션에서는 [AWS SDK for Python \(Boto\)](#)이 필요합니다.

1. 다음 콘텐츠를 가진 `stock.py`이라는 파일을 생성합니다:

```
import datetime
import json
import random
import boto3

STREAM_NAME = "ExampleInputStream"

def get_data():
    return {
        "EVENT_TIME": datetime.datetime.now().isoformat(),
        "TICKER": random.choice(["AAPL", "AMZN", "MSFT", "INTC", "TBV"]),
        "PRICE": round(random.random() * 100, 2),
    }

def generate(stream_name, kinesis_client):
    while True:
        data = get_data()
        print(data)
        kinesis_client.put_record(
            StreamName=stream_name, Data=json.dumps(data),
            PartitionKey="partitionkey"
        )

if __name__ == "__main__":
    generate(STREAM_NAME, boto3.client("kinesis"))
```

- 이 자습서의 뒷부분에서 `stock.py` 스크립트를 실행하여 애플리케이션으로 데이터를 전송합니다.

```
$ python stock.py
```

## Apache Flink 스트리밍 Java 코드 다운로드 및 검사

이 예제의 Java 애플리케이션 코드는 에서 GitHub 제공됩니다. 애플리케이션 코드를 다운로드하려면 다음을 수행하세요.

- 다음 명령을 사용하여 원격 리포지토리를 복제합니다:

```
git clone https://github.com/aws-samples/amazon-kinesis-data-analytics-java-examples.git
```

- `GettingStarted` 디렉터리로 이동합니다.

애플리케이션 코드는 `CustomSinkStreamingJob.java` 및 `CloudWatchLogSink.java` 파일에 있습니다. 애플리케이션 코드에 대해 다음을 유의하십시오:

- 애플리케이션은 Kinesis 소스를 사용하여 소스 스트림에서 읽습니다. 다음 조각은 Kinesis 싱크를 생성합니다.

```
return env.addSource(new FlinkKinesisConsumer<>(inputStreamName,
    new SimpleStringSchema(), inputProperties));
```

## 애플리케이션 코드 컴파일

이 섹션에서는 Apache Maven 컴파일러를 사용하여 애플리케이션용 Java 코드를 생성합니다. Apache Maven 및 Java Development Kit(JDK) 설치에 대한 자세한 내용을 알아보려면 [연습 완료를 위한 필수 조건](#) 섹션을 참조하십시오.

Java 애플리케이션을 사용하려면 다음 구성 요소가 필요합니다.

- [Project Object Model\(pom.xml\)](#) 파일. 이 파일에는 Flink 애플리케이션용 Managed Service for Apache Flink 라이브러리를 포함하여 애플리케이션의 구성 및 종속성에 대한 정보가 들어 있습니다.
- 애플리케이션의 로직을 포함하는 main 메서드

**Note**

다음 애플리케이션에 대해 Kinesis 커넥터를 사용하려면 커넥터의 소스 코드를 다운로드하고 [Apache Flink 설명서](#)에 설명된 대로 빌드해야 합니다.

애플리케이션 코드를 생성 및 컴파일하려면

1. 개발 환경에서 Java/Maven 애플리케이션을 생성합니다. 애플리케이션 생성에 대한 자세한 내용은 해당 개발 환경 설명서를 참조하십시오.
  - [첫 번째 Java 프로젝트 생성\(Eclipse Java Neon\)](#)
  - [첫 번째 Java 애플리케이션 생성, 실행 및 패키징\(IntelliJ Idea\)](#)
2. StreamingJob.java라는 파일에 다음 코드를 사용하십시오.

```
package com.amazonaws.services.kinesisanalytics;

import com.amazonaws.services.kinesisanalytics.runtime.KinesisAnalyticsRuntime;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kinesis.FlinkKinesisConsumer;
import org.apache.flink.streaming.connectors.kinesis.FlinkKinesisProducer;
import
    org.apache.flink.streaming.connectors.kinesis.config.ConsumerConfigConstants;

import java.io.IOException;
import java.util.Map;
import java.util.Properties;

public class StreamingJob {

    private static final String region = "us-east-1";
    private static final String inputStreamName = "ExampleInputStream";
    private static final String outputStreamName = "ExampleOutputStream";

    private static DataStream<String>
    createSourceFromStaticConfig(StreamExecutionEnvironment env) {
        Properties inputProperties = new Properties();
        inputProperties.setProperty(ConsumerConfigConstants.AWS_REGION, region);
```



```
inputProperties.setProperty(ConsumerConfigConstants.STREAM_INITIAL_POSITION,
"LATEST");

    return env.addSource(new FlinkKinesisConsumer<>(inputStreamName, new
SimpleStringSchema(), inputProperties));
}

private static DataStream<String>
createSourceFromApplicationProperties(StreamExecutionEnvironment env)
throws IOException {
    Map<String, Properties> applicationProperties =
KinesisAnalyticsRuntime.getApplicationProperties();
    return env.addSource(new FlinkKinesisConsumer<>(inputStreamName, new
SimpleStringSchema(),
        applicationProperties.get("ConsumerConfigProperties")));
}

private static FlinkKinesisProducer<String> createSinkFromStaticConfig() {
    Properties outputProperties = new Properties();
    outputProperties.setProperty(ConsumerConfigConstants.AWS_REGION, region);
    outputProperties.setProperty("AggregationEnabled", "false");

    FlinkKinesisProducer<String> sink = new FlinkKinesisProducer<>(new
SimpleStringSchema(), outputProperties);
    sink.setDefaultStream(outputStreamName);
    sink.setDefaultPartition("0");
    return sink;
}

private static FlinkKinesisProducer<String>
createSinkFromApplicationProperties() throws IOException {
    Map<String, Properties> applicationProperties =
KinesisAnalyticsRuntime.getApplicationProperties();
    FlinkKinesisProducer<String> sink = new FlinkKinesisProducer<>(new
SimpleStringSchema(),
        applicationProperties.get("ProducerConfigProperties"));

    sink.setDefaultStream(outputStreamName);
    sink.setDefaultPartition("0");
    return sink;
}

public static void main(String[] args) throws Exception {
```

```

// set up the streaming execution environment
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

/*
 * if you would like to use runtime configuration properties, uncomment the
 * lines below
 * DataStream<String> input = createSourceFromApplicationProperties(env);
 */

DataStream<String> input = createSourceFromStaticConfig(env);

/*
 * if you would like to use runtime configuration properties, uncomment the
 * lines below
 * input.addSink(createSinkFromApplicationProperties())
 */

input.addSink(createSinkFromStaticConfig());

env.execute("Flink Streaming Java API Skeleton");
}
}

```

앞의 코드 예제에 대해서는 다음 사항에 유의하십시오.

- 이 파일에는 애플리케이션의 기능을 정의하는 main 메서드가 들어 있습니다.
  - 애플리케이션은 `StreamExecutionEnvironment` 객체를 사용하여 외부 리소스에 액세스하기 위한 소스 및 싱크 커넥터를 생성합니다.
  - 애플리케이션은 정적 속성을 사용하여 소스 및 싱크 커넥터를 만듭니다. 동적 애플리케이션 속성을 사용하려면 `createSourceFromApplicationProperties` 및 `createSinkFromApplicationProperties` 메서드를 사용하여 커넥터를 생성합니다. 이 메서드는 애플리케이션의 속성을 읽어 커넥터를 구성합니다.
3. 애플리케이션 코드를 사용하려면 이를 컴파일하고 JAR 파일로 패키징합니다. 다음 두 가지 방법 중 하나로 코드를 컴파일하고 패키징할 수 있습니다:
- 명령줄 Maven 도구 사용. `pom.xml` 파일이 있는 디렉터리에서 다음 명령을 실행하여 JAR 파일을 생성합니다:

```
mvn package
```

- 귀하의 개발 환경 사용. 자세한 내용을 알아보려면 해당 개발 환경 설명서를 참조하십시오.

패키지를 JAR 파일로 업로드하거나 패키지를 압축하여 ZIP 파일로 업로드할 수 있습니다. 를 사용하여 애플리케이션을 생성하는 경우 코드 콘텐츠 유형 (JAR 또는 ZIP) 을 지정합니다. AWS CLI

4. 컴파일하는 동안 오류가 발생하면 JAVA\_HOME 환경 변수가 올바르게 설정되어 있는지 확인하십시오.

애플리케이션이 성공적으로 컴파일되면 다음 파일이 생성됩니다:

```
target/java-getting-started-1.0.jar
```

## Apache Flink 스트리밍 Java 코드 업로드

이 섹션에서는 Amazon Simple Storage Service(Amazon S3) 버킷을 만들고 애플리케이션 코드를 업로드합니다.

애플리케이션 코드 업로드하기

1. <https://console.aws.amazon.com/s3/>에서 Amazon S3 콘솔을 엽니다.
2. 버킷 만들기를 선택합니다.
3. 버킷 명칭 필드에 **ka-app-code-*<username>***을 입력합니다. 버킷 명칭에 사용자 이름 등의 접미사를 추가하여 전역적으로 고유하게 만듭니다. 다음을 선택합니다.
4. 옵션 구성 단계에서 설정을 기본값 그대로 두고 다음을 선택합니다.
5. 권한 설정 단계에서 설정을 기본값 그대로 두고 다음을 선택합니다.
6. 버킷 생성을 선택합니다.
7. Amazon S3 콘솔에서 ka-app-code- *<username>*버킷을 선택하고 업로드를 선택합니다.
8. 파일 선택 단계에서 파일 추가를 선택합니다. 이전 단계에서 생성한 java-getting-started-1.0.jar 파일로 이동합니다. 다음을 선택합니다.
9. 권한 설정 단계에서 설정을 기본값 그대로 유지합니다. 다음을 선택합니다.
10. 속성 설정 단계에서 설정을 기본값 그대로 유지합니다. 업로드를 선택합니다.

이제 애플리케이션 코드가 애플리케이션에서 액세스할 수 있는 Amazon S3 버킷에 저장됩니다.

## Managed Service for Apache Flink 애플리케이션 생성 및 실행

콘솔이나 AWS CLI를 사용하여 Flink 애플리케이션용 Managed Service for Apache Flink를 생성하고 실행할 수 있습니다.

### Note

콘솔을 사용하여 애플리케이션을 생성하면 사용자 AWS Identity and Access Management (IAM) 및 Amazon CloudWatch Logs 리소스가 자동으로 생성됩니다. 를 사용하여 애플리케이션을 생성할 때는 이러한 리소스를 별도로 생성합니다. AWS CLI

### 주제

- [애플리케이션 생성 및 실행\(콘솔\)](#)
- [애플리케이션 생성 및 실행 \(AWS CLI\)](#)

### 애플리케이션 생성 및 실행(콘솔)

콘솔을 사용하여 애플리케이션을 생성, 구성, 업데이트 및 실행하려면 다음 단계를 수행하세요.

#### 애플리케이션 생성

1. <https://console.aws.amazon.com/kinesis>에서 Kinesis 콘솔을 엽니다.
2. Amazon Kinesis 대시보드에서 분석 애플리케이션 생성을 선택합니다.
3. Kinesis Analytics - 애플리케이션 생성 페이지에서 다음과 같이 애플리케이션 세부 정보를 제공합니다.
  - 애플리케이션 명칭에 **MyApplication**을 입력합니다.
  - 설명에 **My java test app**를 입력합니다.
  - 런타임에서 Apache Flink 1.6을 선택합니다.
4. 액세스 권한에서 IAM 역할 **kinesis-analytics-MyApplication-us-west-2** 생성/업데이트를 선택합니다.
5. 애플리케이션 생성을 선택합니다.

**Note**

콘솔을 사용하여 Flink 애플리케이션용 Managed Service for Apache Flink를 생성할 때 애플리케이션에 대한 IAM 역할 및 정책을 생성할 수 있습니다. 귀하의 애플리케이션은 이 역할 및 정책을 사용하여 종속 리소스에 액세스합니다. 이러한 IAM 리소스의 이름은 애플리케이션 명칭과 리전을 사용하여 다음과 같이 지정됩니다.

- 정책: `kinesis-analytics-service-MyApplication-us-west-2`
- 역할: `kinesis-analytics-MyApplication-us-west-2`

**IAM 정책 편집**

IAM 정책을 편집하여 Kinesis Data Streams에 액세스할 수 있는 권한을 추가합니다.

1. <https://console.aws.amazon.com/iam/>에서 IAM 콘솔을 여세요.
2. 정책을 선택하세요. 이전 섹션에서 콘솔이 생성한 **kinesis-analytics-service-MyApplication-us-west-2** 정책을 선택합니다.
3. 요약 페이지에서 정책 편집을 선택합니다. JSON 탭을 선택합니다.
4. 다음 정책 예제의 강조 표시된 부분을 정책에 추가하세요. 샘플 계정 ID(`012345678901`)를 내 계정 ID로 바꿉니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ReadCode",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:GetObjectVersion"
      ],
      "Resource": [
        "arn:aws:s3:::ka-app-code-username/java-getting-started-1.0.jar"
      ]
    },
    {
      "Sid": "ListCloudwatchLogGroups",
      "Effect": "Allow",
      "Action": [
```

```

        "logs:DescribeLogGroups"
    ],
    "Resource": [
        "arn:aws:logs:us-west-2:012345678901:log-group:*"
    ]
},
{
    "Sid": "ListCloudwatchLogStreams",
    "Effect": "Allow",
    "Action": [
        "logs:DescribeLogStreams"
    ],
    "Resource": [
        "arn:aws:logs:us-west-2:012345678901:log-group:/aws/kinesis-
analytics/MyApplication:log-stream:*"
    ]
},
{
    "Sid": "PutCloudwatchLogs",
    "Effect": "Allow",
    "Action": [
        "logs:PutLogEvents"
    ],
    "Resource": [
        "arn:aws:logs:us-west-2:012345678901:log-group:/aws/kinesis-
analytics/MyApplication:log-stream:kinesis-analytics-log-stream"
    ]
},
{
    "Sid": "ReadInputStream",
    "Effect": "Allow",
    "Action": "kinesis:*",
    "Resource": "arn:aws:kinesis:us-west-2:012345678901:stream/
ExampleInputStream"
},
{
    "Sid": "WriteOutputStream",
    "Effect": "Allow",
    "Action": "kinesis:*",
    "Resource": "arn:aws:kinesis:us-west-2:012345678901:stream/
ExampleOutputStream"
}
]

```

```
}

```

## 애플리케이션 구성

1. MyApplication페이지에서 구성을 선택합니다.
2. 애플리케이션 구성 페이지에서 코드 위치를 입력합니다.
  - Amazon S3 버킷의 경우 **ka-app-code-*<username>***를 입력합니다.
  - Amazon S3 객체 경로에는 **java-getting-started-1.0.jar**를 입력합니다.
3. 애플리케이션 리소스에 대한 액세스 아래에서 액세스 권한의 경우 IAM 역할 **kinesis-analytics-MyApplication-us-west-2** 생성/업데이트를 선택합니다.
4. 속성에서 그룹 ID에 **ProducerConfigProperties**를 입력합니다.
5. 다음 애플리케이션 속성 및 값을 입력합니다:

키	값
<b>flink.inputstream.initpos</b>	<b>LATEST</b>
<b>aws:region</b>	<b>us-west-2</b>
<b>AggregationEnabled</b>	<b>false</b>

6. 모니터링에서 지표 수준 모니터링이 애플리케이션으로 설정되어 있는지 확인합니다.
7. CloudWatch 로깅의 경우 활성화 확인란을 선택합니다.
8. 업데이트를 선택합니다.

### Note

CloudWatch 로깅을 활성화하도록 선택하면 Apache Flink용 관리형 서비스에서 로그 그룹과 로그 스트림을 자동으로 생성합니다. 이러한 리소스의 이름은 다음과 같습니다.

- 로그 그룹: /aws/kinesis-analytics/MyApplication
- 로그 스트림: kinesis-analytics-log-stream

## 애플리케이션 실행

1. MyApplication페이지에서 [Run] 을 선택합니다. 작업을 확인합니다.
2. 애플리케이션이 실행 중이면 페이지를 새로 고칩니다. 콘솔에 애플리케이션 그래프가 표시됩니다.

## 애플리케이션 중지

MyApplication페이지에서 중지를 선택합니다. 작업을 확인합니다.

## 애플리케이션 업데이트

콘솔을 사용하여 애플리케이션 속성, 모니터링 설정, 애플리케이션 JAR의 위치 또는 파일 명칭과 같은 애플리케이션 설정을 업데이트할 수 있습니다. 애플리케이션 코드를 업데이트해야 하는 경우 Amazon S3 버킷에서 애플리케이션 JAR을 다시 로드할 수도 있습니다.

MyApplication페이지에서 구성을 선택합니다. 애플리케이션 설정을 업데이트하고 업데이트를 선택합니다.

## 애플리케이션 생성 및 실행 (AWS CLI)

이 섹션에서는 `aws` 를 사용하여 Apache Flink용 관리 서비스 애플리케이션을 만들고 실행합니다. AWS CLI Flink 응용 프로그램용 Apache Flink 관리 서비스는 `kinesisanalyticsv2` AWS CLI 명령을 사용하여 Apache Flink 응용 프로그램용 관리 서비스 응용 프로그램을 만들고 상호 작용합니다.

## 권한 정책 생성

먼저 소스 스트림에서 두 개의 명령문, 즉 `read` 작업에 대한 권한을 부여하는 명령문과 싱크 스트림에서 `write` 작업에 대한 권한을 부여하는 명령문이 있는 권한 정책을 만듭니다. 그런 다음 정책을 IAM 역할(다음 섹션에서 생성)에 연결합니다. 따라서 Managed Service for Apache Flink가 역할을 맡을 때 서비스는 소스 스트림에서 읽고 싱크 스트림에 쓸 수 있는 권한이 있습니다.

다음 코드를 사용하여 `KAReadSourceStreamWriteSinkStream` 권한 정책을 생성합니다.

`username`을 애플리케이션 코드를 저장하기 위해 Amazon S3 버킷을 만들 때 사용한 사용자 이름으로 바꿉니다. Amazon 리소스 이름(ARN)의 계정 ID(`012345678901`)를 사용자의 계정 ID로 바꿉니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "S3",
      "Effect": "Allow",
```



```

    "Action": [
      "s3:GetObject",
      "s3:GetObjectVersion"
    ],
    "Resource": ["arn:aws:s3:::ka-app-code-username",
      "arn:aws:s3:::ka-app-code-username/*"
    ]
  },
  {
    "Sid": "ReadInputStream",
    "Effect": "Allow",
    "Action": "kinesis:*",
    "Resource": "arn:aws:kinesis:us-west-2:012345678901:stream/
ExampleInputStream"
  },
  {
    "Sid": "WriteOutputStream",
    "Effect": "Allow",
    "Action": "kinesis:*",
    "Resource": "arn:aws:kinesis:us-west-2:012345678901:stream/
ExampleOutputStream"
  }
]
}

```

권한 정책을 생성하는 step-by-step 방법에 대한 지침은 IAM 사용 [설명서의 자습서: 첫 번째 고객 관리 형 정책 생성 및 연결](#)을 참조하십시오.

#### Note

를 사용하여 다른 AWS 서비스에 액세스할 수 있습니다. AWS SDK for Java Managed Service for Apache Flink는 SDK에 필요한 자격 증명을 애플리케이션과 연결된 서비스 실행 IAM 역할의 자격 증명으로 자동 설정합니다. 추가 단계는 필요 없습니다.

## IAM 역할 생성

이 섹션에서는 Flink 애플리케이션용 Managed Service for Apache Flink에서 소스 스트림을 읽고 싱크 스트림에 쓸 수 있는 IAM 역할을 생성합니다.

Managed Service for Apache Flink는 권한 없이 스트림에 액세스할 수 없습니다. IAM 역할을 통해 이러한 권한을 부여합니다. 각 IAM 역할에는 두 가지 정책이 연결됩니다. 신뢰 정책은 Managed Service

for Apache Flink가 역할을 취할 수 있는 권한을 부여하고, 권한 정책은 역할을 취한 후 Managed Service for Apache Flink에서 수행할 수 있는 작업을 결정합니다.

이전 섹션에서 생성한 권한 정책을 이 역할에 연결합니다.

IAM 역할을 생성하려면

1. <https://console.aws.amazon.com/iam/>에서 IAM 콘솔을 엽니다.
2. 탐색 창에서 역할, 역할 생성을 선택합니다.
3. 신뢰할 수 있는 유형의 자격 증명 선택에서 AWS 서비스를 선택합니다. 이 역할을 사용할 서비스 선택에서 Kinesis를 선택합니다. 사용 사례 선택에서 Kinesis Analytics를 선택합니다.

다음: 권한을 선택합니다.

4. 권한 정책 연결 페이지에서 다음: 검토를 선택합니다. 역할을 생성한 후에 권한 정책을 연결합니다.
5. 역할 생성 페이지에서 역할 이름으로 **KA-stream-rw-role**을 입력합니다. 역할 생성을 선택합니다.

KA-stream-rw-role이라는 새 IAM 역할이 생성되었습니다. 그런 다음 역할의 신뢰 정책 및 권한 정책을 업데이트합니다.

6. 역할에 권한 정책을 연결합니다.

#### Note

이 연습에서는 Managed Service for Apache Flink가 이 역할을 취하여 Kinesis 데이터 스트림(소스)에서 데이터를 읽고 출력을 다른 Kinesis 데이터 스트림에 씁니다. 따라서 이전 단계 [the section called “권한 정책 생성”](#)에서 생성한 정책을 연결합니다.

- a. 요약 페이지에서 권한 탭을 선택합니다.
- b. 정책 연결을 선택합니다.
- c. 검색 상자에 **KAReadSourceStreamWriteSinkStream**(이전 섹션에서 생성한 정책)을 입력합니다.
- d. KA ReadInputStreamWriteOutputStream 정책을 선택하고 [Attach policy] 를 선택합니다.

이제 애플리케이션이 리소스에 액세스하는 데 사용하는 서비스 실행 역할이 생성되었습니다. 새 역할의 ARN을 기록합니다.

역할 생성에 대한 step-by-step 지침은 IAM 사용 설명서의 [IAM 역할 생성 \(콘솔\)](#) 을 참조하십시오.

## Managed Service for Apache Flink 애플리케이션 생성

1. 다음 JSON 코드를 `create_request.json`이라는 파일에 저장합니다. 샘플 역할 ARN을 이전에 생성한 역할을 위한 ARN으로 바꿉니다. 버킷 ARN 접미사(`username`)를 이전 섹션에서 선택한 접미사로 바꿉니다. 서비스 실행 역할의 샘플 계정 ID(`012345678901`)를 해당 계정 ID로 바꿉니다.

```
{
  "ApplicationName": "test",
  "ApplicationDescription": "my java test app",
  "RuntimeEnvironment": "FLINK-1_6",
  "ServiceExecutionRole": "arn:aws:iam::012345678901:role/KA-stream-rw-role",
  "ApplicationConfiguration": {
    "ApplicationCodeConfiguration": {
      "CodeContent": {
        "S3ContentLocation": {
          "BucketARN": "arn:aws:s3:::ka-app-code-username",
          "FileKey": "java-getting-started-1.0.jar"
        }
      },
      "CodeContentType": "ZIPFILE"
    },
    "EnvironmentProperties": {
      "PropertyGroups": [
        {
          "PropertyGroupId": "ProducerConfigProperties",
          "PropertyMap": {
            "flink.stream.initpos" : "LATEST",
            "aws.region" : "us-west-2",
            "AggregationEnabled" : "false"
          }
        },
        {
          "PropertyGroupId": "ConsumerConfigProperties",
          "PropertyMap": {
            "aws.region" : "us-west-2"
          }
        }
      ]
    }
  }
}
```

```
}
```

- 위의 요청과 함께 [CreateApplication](#) 작업을 실행하여 애플리케이션을 생성합니다.

```
aws kinesisanalyticsv2 create-application --cli-input-json file://  
create_request.json
```

애플리케이션이 생성되었습니다. 다음 단계에서는 애플리케이션을 시작합니다.

### 애플리케이션 시작

이 섹션에서는 [StartApplication](#) 작업을 사용하여 애플리케이션을 시작합니다.

### 애플리케이션을 시작하려면

- 다음 JSON 코드를 `start_request.json`이라는 파일에 저장합니다.

```
{  
  "ApplicationName": "test",  
  "RunConfiguration": {  
    "ApplicationRestoreConfiguration": {  
      "ApplicationRestoreType": "RESTORE_FROM_LATEST_SNAPSHOT"  
    }  
  }  
}
```

- 위의 요청과 함께 [StartApplication](#) 작업을 실행하여 애플리케이션을 시작합니다.

```
aws kinesisanalyticsv2 start-application --cli-input-json file://start_request.json
```

애플리케이션이 실행됩니다. Amazon CloudWatch 콘솔에서 Apache Flink용 관리형 서비스 지표를 확인하여 애플리케이션이 작동하는지 확인할 수 있습니다.

### 애플리케이션 중지

이 섹션에서는 [StopApplication](#) 작업을 사용하여 애플리케이션을 중지합니다.

### 애플리케이션을 중지하려면

- 다음 JSON 코드를 `stop_request.json`이라는 파일에 저장합니다.

```
{"ApplicationName": "test"
}
```

2. 다음 요청과 함께 [StopApplication](#) 작업을 실행하여 애플리케이션을 중지합니다.

```
aws kinesisanalyticstv2 stop-application --cli-input-json file://stop_request.json
```

애플리케이션이 중지됩니다.

## 자습서: Amazon Kinesis Data Streams에서 AWS Lambda 사용

이 자습서에서는 Kinesis 데이터 스트림의 이벤트를 소비하기 위한 Lambda 함수를 생성합니다. 이 예제 시나리오에서 사용자 지정 애플리케이션은 Kinesis 데이터 스트림에 레코드를 씁니다. AWS 그런 다음 Lambda가 이 데이터 스트림을 폴링하고, 새 데이터 레코드를 감지하면 Lambda 함수를 간접적으로 호출합니다. AWS 그런 다음 Lambda는 Lambda 함수를 생성할 때 지정한 실행 역할을 수입하여 Lambda 함수를 실행합니다.

세부적인 단계별 지침은 [자습서: Amazon Kinesis에서 AWS Lambda 사용](#)을 참조하세요.

### Note

이 자습서에서는 사용자가 기본 Lambda 작업과 Lambda 콘솔에 대해 어느 정도 알고 있다고 가정합니다. 아직 없는 경우 [AWS Lambda 시작하기](#)의 지침에 따라 첫 번째 Lambda 함수를 생성합니다.

## AWS Streaming Data Solution for Amazon Kinesis

AWS Streaming Data Solution for Amazon Kinesis는 스트리밍 데이터를 쉽게 캡처, 저장, 처리 및 전송하는 데 필요한 AWS 서비스를 자동으로 구성합니다. 이 솔루션은 Kinesis Data Streams, AWS Lambda, Amazon API Gateway, Amazon Managed Service for Apache Flink 등의 여러 AWS 서비스를 사용하는 스트리밍 데이터 사용 사례를 해결하기 위한 다양한 옵션을 제공합니다.

각 솔루션은 다음 구성 요소를 포함합니다.

- 전체 예제를 배포하기 위한 AWS CloudFormation 패키지
- 애플리케이션 지표를 표시하는 CloudWatch 대시보드

- 가장 관련성이 높은 애플리케이션 지표에 대한 CloudWatch 경보
- 필요한 모든 IAM 역할 및 정책

솔루션은 [Streaming Data Solution for Amazon Kinesis](#)에서 찾을 수 있습니다.

# 스트림 생성 및 관리

Amazon Kinesis Data Streams는 대량의 데이터를 실시간으로 수집하고 내구성을 고려하여 데이터를 저장하며 데이터를 소비할 수 있는 상태로 만듭니다. Kinesis Data Streams에서 저장하는 데이터의 단위는 데이터 레코드입니다. 데이터 스트림은 데이터 레코드 그룹을 나타냅니다. 데이터 스트림의 데이터 레코드는 샤드에 배포됩니다.

샤드에는 스트림의 데이터 레코드 시퀀스가 있습니다. Kinesis 데이터 스트림의 기본 처리량 단위 역할을 합니다. 샤드는 온디맨드 및 프로비저닝된 용량 모드 모두에서 쓰기의 경우 초당 1000개의 레코드 및 1MB/s를 지원하고 읽기의 경우 2MB/s를 지원합니다. 샤드 한도는 예측 가능한 성능을 보장하므로 매우 안정적인 데이터 스트리밍 워크플로를 쉽게 설계하고 운영할 수 있습니다.

## 주제

- [데이터 스트림 용량 모드 선택](#)
- [관리 콘솔을 통한 스트림 생성 AWS](#)
- [API를 통해 스트림 생성](#)
- [스트림 업데이트](#)
- [스트림 나열](#)
- [샤드 나열](#)
- [스트림 삭제](#)
- [스트림 리샤딩](#)
- [데이터 보존 기간 변경](#)
- [Amazon Kinesis Data Streams의 스트림 태그 지정](#)

## 데이터 스트림 용량 모드 선택

### 주제

- [데이터 스트림 용량 모드란?](#)
- [온디맨드 모드](#)
- [프로비저닝된 모드](#)
- [용량 모드 간 전환](#)

## 데이터 스트림 용량 모드란?

용량 모드는 데이터 스트림의 용량이 관리되는 방식과 데이터 스트림 사용에 대한 요금이 청구되는 방식을 결정합니다. Amazon Kinesis Data Streams에서 데이터 스트림에 대해 온디맨드 모드와 프로비저닝된 모드 중에서 선택할 수 있습니다.

- 온디맨드 - 온디맨드 모드의 데이터 스트림은 용량 계획이 필요 없으며 분당 수 기가바이트의 쓰기 및 읽기 처리량을 처리하도록 자동으로 확장됩니다. 온디맨드 모드에서 Kinesis Data Streams는 필요한 처리량을 제공하기 위해 샤드를 자동으로 관리합니다.
- 프로비저닝된 - 프로비저닝된 모드의 데이터 스트림의 경우 데이터 스트림의 샤드 수를 지정해야 합니다. 데이터 스트림의 총 용량은 해당 샤드 용량의 합계입니다. 필요한 만큼 데이터 스트림의 샤드 수를 늘리거나 줄일 수 있습니다.

Kinesis Data Streams PutRecord 및 PutRecords API를 사용하여 온디맨드 및 프로비저닝된 용량 모드 모두에서 데이터 스트림에 데이터를 쓸 수 있습니다. 데이터를 검색하기 위해 두 용량 모드 모두 GetRecords API를 사용하는 기본 소비자와 SubscribeToShard API를 사용하는 향상된 팬아웃 (EFO) 소비자를 지원합니다.

보존 모드, 암호화, 모니터링 지표 등을 포함한 모든 Kinesis Data Streams 기능은 온디맨드 모드와 프로비저닝된 모드 모두에 대해 지원됩니다. Kinesis Data Streams는 온디맨드 및 프로비저닝된 용량 모드 모두에서 높은 내구성과 가용성을 제공합니다.

## 온디맨드 모드

온디맨드 모드의 데이터 스트림은 용량 계획이 필요 없으며 분당 수 기가바이트의 쓰기 및 읽기 처리량을 처리하도록 자동으로 확장됩니다. 온디맨드 모드를 사용하면 서버, 스토리지 또는 처리량을 프로비저닝하고 관리할 필요가 없으므로 짧은 지연 시간으로 대용량 데이터를 간편하게 수집하고 저장할 수 있습니다. 운영 오버헤드 없이 하루에 수십억 개의 레코드를 수집할 수 있습니다.

온디맨드 모드는 매우 가변적이고 예측할 수 없는 애플리케이션 트래픽의 요구 사항을 해결하는 데 이상적입니다. 더 이상 최대 용량을 위해 이러한 워크로드를 프로비저닝하지 않아도 되므로 사용률이 낮아 비용이 증가할 수 있습니다. 온디맨드 모드는 예측할 수 없고 매우 가변적인 트래픽 패턴을 가진 워크로드에 적합합니다.

온디맨드 용량 모드를 사용하면 데이터 스트림에서 쓰고 읽은 데이터에 대해 GB당 요금을 지불합니다. 애플리케이션에서 수행할 것으로 예상되는 읽기 및 쓰기 처리량을 지정할 필요가 없습니다. Kinesis Data Streams는 워크로드가 증가하거나 감소할 때 즉시 워크로드를 수용합니다. 자세한 내용은 [Amazon Kinesis Data Streams 요금](#)을 참조하세요.



Kinesis Data Streams 콘솔, API 또는 CLI 명령을 사용하여 온디맨드 모드로 새 데이터 스트림을 생성할 수 있습니다.

온디맨드 모드의 데이터 스트림은 지난 30일 동안 관찰된 최대 쓰기 처리량의 최대 2배를 수용합니다. 데이터 스트림의 쓰기 처리량이 최고치에 도달하면 Kinesis Data Streams는 데이터 스트림의 용량을 자동으로 조정합니다. 예를 들어, 데이터 스트림의 쓰기 처리량이 10MB/s에서 40MB/s 사이인 경우 Kinesis Data Streams를 사용하면 최대 피크 처리량의 2배인 80MB/s까지 쉽게 버스트할 수 있습니다. 동일한 데이터 스트림이 새로운 최대 처리량인 50MB/s를 유지하는 경우 Kinesis Data Streams는 100MB/s의 쓰기 처리량을 수집하기에 충분한 용량을 확보합니다. 하지만 15분 내에 트래픽이 이전 최고치의 2배 이상으로 증가하면 쓰기 제한이 발생할 수 있습니다. 제한된 이러한 요청을 다시 시도해야 합니다.

온디맨드 모드를 사용하는 데이터 스트림의 총 읽기 용량은 쓰기 처리량에 비례하여 증가합니다. 이를 통해 소비자 애플리케이션은 수신 데이터를 실시간으로 처리하는 데 필요한 적절한 읽기 처리량을 항상 확보할 수 있습니다. GetRecords API를 사용한 데이터 읽기에 비해 쓰기 처리량이 2배 이상 높습니다. GetRecord API와 함께 하나의 소비자 애플리케이션을 사용하여 애플리케이션이 다운타임에서 복구해야 할 때 이를 따라잡을 수 있는 충분한 공간을 확보하는 것이 좋습니다. 2개 이상의 소비자 애플리케이션을 추가해야 하는 시나리오에는 Kinesis Data Streams의 향상된 팬아웃 기능을 사용하는 것이 좋습니다. 향상된 팬아웃은 SubscribeToShard API 사용하여 데이터 스트림에 최대 20개의 소비자 애플리케이션을 추가할 수 있도록 지원하며, 각 소비자 애플리케이션에는 전용 처리량이 있습니다.

## 읽기 및 쓰기 처리량 예외 처리

프로비저닝된 용량 모드와 마찬가지로 온디맨드 용량 모드에서는 데이터 스트림에 데이터를 쓰려면 각 레코드에 파티션 키를 지정해야 합니다. Kinesis Data Streams는 파티션 키를 사용하여 샤드 전체에 데이터를 배포합니다. Kinesis Data Streams는 각 샤드의 트래픽을 모니터링합니다. 수신 트래픽이 샤드당 500KB/s를 초과하면 15분 내에 샤드를 분할합니다. 상위 샤드의 해시 키 값은 하위 샤드에 균등하게 재배포됩니다.

수신 트래픽이 이전 최고치의 2배를 초과하는 경우 데이터가 샤드에 균등하게 배포되어 있더라도 약 15분 동안 읽기 또는 쓰기 예외가 발생할 수 있습니다. 모든 레코드가 Kinesis Data Streams에 제대로 저장되도록 이러한 요청을 모두 재시도하는 것이 좋습니다.

파티션 키를 사용하여 데이터가 고르지 않게 배포되고 특정 샤드에 할당된 레코드가 제한을 초과하는 경우 읽기 및 쓰기 예외가 발생할 수 있습니다. 온디맨드 모드에서는 파티션 키 하나가 샤드의 1MB/s 처리량과 초당 레코드 1,000개 제한을 초과하지 않는 한 데이터 스트림이 고르지 않은 데이터 배포 패턴을 처리하도록 자동으로 조정됩니다.

온디맨드 모드에서 Kinesis Data Streams는 트래픽 증가를 감지하면 샤드를 균등하게 분할합니다. 하지만 수신 트래픽의 더 많은 부분을 특정 샤드로 유도하는 해시 키를 탐지하고 격리하지는 않습니다.

매우 고르지 않은 파티션 키를 사용하는 경우 쓰기 예외가 계속 발생할 수 있습니다. 이러한 사용 사례에서는 세분화된 샤드 분할을 지원하는 프로비저닝된 용량 모드를 사용하는 것이 좋습니다.

## 프로비저닝된 모드

프로비저닝 모드를 사용하면 데이터 스트림을 생성한 후 또는 API를 사용하여 샤드 용량을 동적으로 늘리거나 줄일 수 있습니다. AWS Management Console [UpdateShardCount](#) Kinesis Data Streams 생산자 또는 소비자 애플리케이션이 스트림에 데이터를 쓰거나 스트림에서 데이터를 읽는 동안 업데이트할 수 있습니다.

프로비저닝된 모드는 용량 요구 사항을 예측하기 쉬운 예측 가능한 트래픽에 적합합니다. 샤드에 데이터가 배포되는 방식을 세밀하게 제어하려는 경우 프로비저닝된 모드를 사용할 수 있습니다.

프로비저닝된 모드에서는 데이터 스트림의 샤드 수를 지정해야 합니다. 프로비저닝된 모드 사용 시 데이터 스트림의 크기를 결정하려면 다음 입력 값이 필요합니다.

- 스트림에 쓰여지는 평균 데이터 레코드 크기(KB 단위), 가장 가까운 1KB로 반올림됨 (average\_data\_size\_in\_KB)
- 스트림에서 초당 쓰고 읽는 데이터 레코드 수(records\_per\_second)
- 스트림과 동시에 독립적으로 데이터를 소비하는 Kinesis Data Streams 애플리케이션인 소비자 수 (number\_of\_consumers)
- KB 단위의 수신 쓰기 대역폭(incoming\_write\_bandwidth\_in\_KB)은 average\_data\_size\_in\_KB에 records\_per\_second를 곱한 값과 같습니다.
- KB 단위의 발신 읽기 대역폭(outgoing\_read\_bandwidth\_in\_KB)은 incoming\_write\_bandwidth\_in\_KB에 number\_of\_consumers를 곱한 값과 같습니다.

다음 형식의 입력 값을 사용하여 스트림에 필요한 샤드 수(number\_of\_shards)를 계산할 수 있습니다.

```
number_of_shards = max(incoming_write_bandwidth_in_KiB/1024,
    outgoing_read_bandwidth_in_KiB/2048)
```

최대 처리량을 처리하도록 데이터 스트림을 구성하지 않으면 프로비저닝된 모드에서 읽기 및 쓰기 처리량 예외가 계속 발생할 수 있습니다. 이 경우 데이터 트래픽을 수용할 수 있도록 데이터 스트림 규모를 수동으로 조정해야 합니다.

또한 파티션 키를 사용하여 데이터가 고르지 않게 배포되고 샤드에 할당된 레코드가 제한을 초과하는 경우 읽기 및 쓰기 예외가 발생할 수 있습니다. 프로비저닝된 모드에서 이 문제를 해결하려면 해당 샤

드를 식별하고 트래픽을 더 잘 수용할 수 있도록 수동으로 분할합니다. 자세한 내용은 [스트림 리샤딩](#)을 참조하세요.

## 용량 모드 간 전환

온디맨드 용량 모드와 프로비저닝된 용량 모드 사이를 전환할 수 있습니다. AWS 계정의 각 데이터 스트림에 대해 24시간 내에 두 번 온디맨드 용량 모드와 프로비저닝된 용량 모드 사이를 전환할 수 있습니다.

데이터 스트림의 용량 모드를 전환해도 이 데이터 스트림을 사용하는 애플리케이션이 중단되지는 않습니다. 이 데이터 스트림에서 계속 쓰고 읽을 수 있습니다. 온디맨드 용량 모드와 프로비저닝된 용량 모드 사이를 전환할 때 스트림 상태는 업데이트 중으로 설정됩니다. 속성을 다시 수정하려면 데이터 스트림이 활성 상태가 될 때까지 기다려야 합니다.

프로비저닝된 용량 모드에서 온디맨드 용량 모드로 전환하면 처음에는 데이터 스트림이 전환 전의 샤드 수를 유지하며, 이 시점부터 Kinesis Data Streams는 데이터 트래픽을 모니터링하고 쓰기 처리량에 따라 이 온디맨드 데이터 스트림의 샤드 수를 조정합니다.

온디맨드 모드에서 프로비저닝된 모드로 전환하면 처음에는 데이터 스트림에 전환 전의 샤드 수가 그대로 유지되지만, 이 때부터는 쓰기 처리량을 적절하게 수용할 수 있도록 이 데이터 스트림의 샤드 수를 모니터링하고 조정해야 합니다.

## 관리 콘솔을 통한 스트림 생성 AWS

Kinesis Data Streams 콘솔, Kinesis Data Streams API 또는 AWS Command Line Interface (AWS CLI)를 사용하여 스트림을 생성할 수 있습니다.

콘솔을 사용하여 데이터 스트림을 만들려면

1. [에 AWS Management Console 로그인](#)하고 <https://console.aws.amazon.com/kinesis> 에서 Kinesis 콘솔을 엽니다.
2. 탐색 모음에서 리전 선택기를 확장하고 리전을 선택합니다.
3. 데이터 스트림 생성을 선택합니다.
4. Kinesis 스트림 생성 페이지에서 데이터 스트림의 이름을 입력한 다음 온디맨드 또는 프로비저닝된 용량 모드를 선택합니다. 온디맨드 모드가 기본적으로 선택됩니다. 자세한 정보는 [데이터 스트림 용량 모드 선택](#)을 참조하세요.

온디맨드 모드를 사용하면 Kinesis 스트림 생성을 선택하여 데이터 스트림을 생성할 수 있습니다. 프로비저닝된 모드에서는 필요한 샤드 수를 지정한 다음 Kinesis 스트림 생성을 선택해야 합니다.

스트림이 생성 중인 동안 Kinesis streams(Kinesis 스트림) 페이지에서 스트림의 Status(상태)는 Creating(생성 중)입니다. 스트림을 사용할 준비가 되면 Status(상태)가 Active(활성)로 변경됩니다.

5. 스트림 명칭을 선택합니다. 스트림 세부 정보 페이지에 모니터링 정보와 함께 스트림 구성 요약이 표시됩니다.

## Kinesis Data Streams API를 사용하여 스트림 생성

- Kinesis Data Streams API를 사용하여 스트림을 생성하는 데 대한 자세한 내용은 [API를 통해 스트림 생성](#) 섹션을 참조하세요.

## 를 사용하여 스트림을 생성하려면 AWS CLI

- 를 사용하여 스트림을 만드는 방법에 대한 자세한 내용은 [create-stream](#) 명령을 참조하십시오.  
AWS CLI

## API를 통해 스트림 생성

다음 단계에 따라 Kinesis 데이터 스트림을 생성합니다.

## Kinesis Data Streams 클라이언트 구축

Kinesis 데이터 스트림 작업을 수행하기 전에 클라이언트 객체를 구축해야 합니다. 다음 Java 코드는 클라이언트 빌더를 인스턴스화하고 이를 사용하여 리전, 자격 증명 및 클라이언트 구성을 설정합니다. 그리고 클라이언트 객체를 구축합니다.

```
AmazonKinesisClientBuilder clientBuilder = AmazonKinesisClientBuilder.standard();

clientBuilder.setRegion(regionName);
clientBuilder.setCredentials(credentialsProvider);
clientBuilder.setClientConfiguration(config);

AmazonKinesis client = clientBuilder.build();
```

자세한 내용은 AWS 일반 참조의 [Kinesis Data Streams Regions and Endpoints](#)를 참조하세요.

## 스트림 생성

Kinesis Data Streams 클라이언트를 생성했으므로 작업할 스트림을 생성할 수 있습니다. 이 스트림은 Kinesis Data Streams 콘솔을 통해 또는 프로그래밍 방식으로 완료할 수 있습니다. 프로그래밍 방식으로 스트림을 생성하려면 `CreateStreamRequest` 객체를 인스턴스화하고 스트림의 이름과 사용할 스트림에 대한 샤드 수(프로비저닝된 모드를 사용하려는 경우)를 지정합니다.

- 온디맨드:

```
CreateStreamRequest createStreamRequest = new CreateStreamRequest();
createStreamRequest.setStreamName( myStreamName );
```

- 프로비저닝됨:

```
CreateStreamRequest createStreamRequest = new CreateStreamRequest();
createStreamRequest.setStreamName( myStreamName );
createStreamRequest.setShardCount( myStreamSize );
```

스트림 이름은 스트림을 식별합니다. 이름은 애플리케이션에서 사용하는 AWS 계정으로 범위가 지정됩니다. 또한 리전에 의해서도 범위가 지정됩니다. 즉, 서로 다른 두 계정의 스트림 두 개가 같은 이름을 가질 수 있고, 같은 AWS 계정이지만 서로 다른 두 지역에 있는 두 스트림의 이름은 같을 수 있지만 같은 계정과 같은 지역에 있는 두 스트림은 이름이 같을 수 없습니다.

스트림의 처리량은 샤드 수를 결정합니다. 프로비저닝된 처리량이 많을수록 더 많은 샤드가 필요합니다. 샤드가 많을수록 스트림에 AWS 부과되는 비용도 늘어납니다. 애플리케이션에 적절한 샤드 수 계산에 대한 자세한 내용은 [데이터 스트림 용량 모드 선택](#) 단원을 참조하십시오.

`createStreamRequest` 객체가 구성된 후 클라이언트에 대해 `createStream` 메서드를 호출하여 스트림을 생성합니다. `createStream`을 호출한 후, 스트림이 ACTIVE 상태에 도달할 때까지 기다린 다음 스트림에 대한 작업을 수행합니다. 스트림의 상태를 확인하려면 `describeStream` 메서드를 호출하십시오. 그러나 스트림이 존재하지 않는 경우 `describeStream`을 호출하면 예외가 발생합니다. 따라서 `describeStream` 호출을 try/catch 블록으로 묶으십시오.

```
client.createStream( createStreamRequest );
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );

long startTime = System.currentTimeMillis();
long endTime = startTime + ( 10 * 60 * 1000 );
```

```
while ( System.currentTimeMillis() < endTime ) {
    try {
        Thread.sleep(20 * 1000);
    }
    catch ( Exception e ) {}

    try {
        DescribeStreamResult describeStreamResponse =
client.describeStream( describeStreamRequest );
        String streamStatus =
describeStreamResponse.getStreamDescription().getStreamStatus();
        if ( streamStatus.equals( "ACTIVE" ) ) {
            break;
        }
        //
        // sleep for one second
        //
        try {
            Thread.sleep( 1000 );
        }
        catch ( Exception e ) {}
    }
    catch ( ResourceNotFoundException e ) {}
}
if ( System.currentTimeMillis() >= endTime ) {
    throw new RuntimeException( "Stream " + myStreamName + " never went active" );
}
```

## 스트림 업데이트

Kinesis Data Streams 콘솔, Kinesis Data Streams API 또는 AWS CLI를 사용하여 스트림의 세부 정보를 업데이트할 수 있습니다.

### Note

기존의 스트림이나 최근에 만든 스트림에 서버 측 암호화를 사용하도록 설정할 수 있습니다.

콘솔을 사용하여 데이터 스트림을 업데이트하려면

1. <https://console.aws.amazon.com/kinesis/>에서 Amazon Kinesis 콘솔을 엽니다.

2. 탐색 모음에서 리전 선택기를 확장하고 리전을 선택합니다.
3. 목록에서 스트림 이름을 선택합니다. Stream Details(스트림 세부 정보) 페이지에 스트림 구성 요약과 모니터링 정보가 표시됩니다.
4. 데이터 스트림에 대한 온디맨드 용량 모드와 프로비저닝된 용량 모드 사이를 전환하려면 구성 탭에서 용량 모드 편집을 선택합니다. 자세한 정보는 [데이터 스트림 용량 모드 선택](#)을 참조하세요.

### Important

AWS 계정의 각 데이터 스트림에 대해 24시간 내에 온디맨드 모드와 프로비저닝 모드 사이를 두 번 전환할 수 있습니다.

5. 프로비저닝된 모드를 사용하는 데이터 스트림의 경우 샤드 수를 편집하려면 구성 탭에서 프로비저닝된 샤드 편집을 선택한 다음 새 샤드 수를 입력합니다.
6. 데이터 레코드의 서버 측 암호화를 활성화하려면 서버 측 암호화 섹션에서 편집을 선택합니다. 암호화를 위한 마스터 키로 사용할 KMS 키를 선택하거나 Kinesis에서 관리하는 기본 마스터 키인 `aws/kinesis`를 사용합니다. 스트림에 대해 암호화를 활성화하고 자체 AWS KMS 마스터 키를 사용하는 경우 생산자 및 소비자 애플리케이션이 사용한 AWS KMS 마스터 키에 액세스할 수 있는지 확인하십시오. 사용자가 생성한 AWS KMS 키에 액세스할 권한을 애플리케이션에 할당하려면 [the section called “사용자 생성 KMS 마스터 키 사용 권한”](#)를 참조하십시오.
7. 데이터 보존 기간을 편집하려면 데이터 보존 기간 섹션에서 편집을 선택한 다음 새 데이터 보존 기간을 입력합니다.
8. 계정에서 사용자 지정 지표를 활성화한 경우 샤드 수준 지표 섹션에서 편집을 선택한 다음 스트림의 지표를 지정합니다. 자세한 정보는 [the section called “를 사용하여 서비스를 모니터링합니다. CloudWatch”](#)을 참조하세요.

## API를 사용하여 스트림 업데이트

API를 사용하여 스트림 세부 정보를 업데이트하려면 다음 방법을 참조하십시오.

- [AddTagsToStream](#)
- [DecreaseStreamRetentionPeriod](#)
- [DisableEnhancedMonitoring](#)
- [EnableEnhancedMonitoring](#)
- [IncreaseStreamRetentionPeriod](#)
- [RemoveTagsFromStream](#)

- [StartStreamEncryption](#)
- [StopStreamEncryption](#)
- [UpdateShardCount](#)

## 를 사용하여 스트림 업데이트 AWS CLI

를 사용하여 스트림을 업데이트하는 방법에 대한 자세한 내용은 [Kinesis CLI](#) 참조를 참조하십시오.  
AWS CLI

## 스트림 나열

이전 섹션에서 설명한 대로 스트림의 범위는 Kinesis Data Streams 클라이언트를 인스턴스화하는 데 사용된 AWS 자격 증명과 연결된 AWS 계정과 클라이언트에 지정된 지역으로 제한됩니다. AWS 계정에는 한 번에 여러 스트림이 활성화될 수 있습니다. Kinesis Data Streams 콘솔에서 또는 프로그래밍 방식으로 스트림을 나열할 수 있습니다. 이 섹션의 코드는 AWS 계정의 모든 스트림을 나열하는 방법을 보여줍니다.

```
ListStreamsRequest listStreamsRequest = new ListStreamsRequest();
listStreamsRequest.setLimit(20);
ListStreamsResult listStreamsResult = client.listStreams(listStreamsRequest);
List<String> streamNames = listStreamsResult.getStreamNames();
```

이 코드 예제는 먼저 `ListStreamsRequest`의 새 인스턴스를 생성하고 해당 `setLimit` 메서드를 호출하여 각 호출에 대해 최대 20개의 스트림이 `listStreams`에 반환되도록 지정합니다. `setLimit`에 대한 값을 지정하지 않으면 Kinesis Data Streams에서는 계정의 수보다 작거나 같은 스트림 수를 반환합니다. 그런 다음 이 코드는 `listStreamsRequest`를 클라이언트의 `listStreams` 메서드로 전달합니다. 반환 값 `listStreams`는 `ListStreamsResult` 객체에 저장됩니다. 이 코드는 이 객체에 대해 `getStreamNames` 메서드를 호출하고 반환된 스트림 이름을 `streamNames` 목록에 저장합니다. Kinesis Data Streams는 계정 및 리전에서 더 많은 스트림이 있는 경우에도 지정된 한도에서 지정된 수보다 더 작은 스트림을 반환할 수 있습니다. 모든 스트림을 검색하려면 다음 코드 예제에 설명된 대로 `getHasMoreStreams` 메서드를 사용하십시오.

```
while (listStreamsResult.getHasMoreStreams())
{
    if (streamNames.size() > 0) {
        listStreamsRequest.setExclusiveStartStreamName(streamNames.get(streamNames.size()
- 1));
```



```
    }  
    listStreamsResult = client.listStreams(listStreamsRequest);  
    streamNames.addAll(listStreamsResult.getStreamNames());  
}
```

이 코드는 `getHasMoreStreams`에 대해 `listStreamsRequest` 메서드를 호출하여 `listStreams`에 대해 초기 호출에서 반환된 스트림 이외에 사용 가능한 추가 스트림이 있는지 여부를 확인합니다. 있는 경우 이 코드는 `setExclusiveStartStreamName`에 대해 이전 호출에서 반환된 마지막 스트림의 이름으로 `listStreams` 메서드를 호출합니다. `setExclusiveStartStreamName` 메서드는 `listStreams`에 대한 다음 호출이 해당 스트림 이후에 시작하도록 합니다. 이 호출에 의해 반환된 스트림 이름의 그룹은 `streamNames` 목록에 추가됩니다. 이 프로세스는 모든 스트림 이름이 목록에 수집될 때까지 계속 진행됩니다.

`listStreams`에 의해 호출된 스트림은 다음 상태 중 하나일 수 있습니다.

- CREATING
- ACTIVE
- UPDATING
- DELETING

이전 단원인 `describeStream`에 표시된 대로 [API를 통해 스트림 생성](#) 메서드를 사용하여 스트림의 상태를 확인할 수 있습니다.

## 샤드 나열

한 데이터 스트림에는 하나 이상의 샤드가 있을 수 있습니다. 데이터 스트림에서 샤드를 나열 또는 검색하는 방법에는 두 가지가 있습니다.

주제

- [ListShards API - 권장](#)
- [DescribeStream API - 더 이상 사용되지 않음](#)

### ListShards API - 권장

데이터 스트림에서 샤드를 나열하거나 검색하는 데 권장되는 방법은 API를 사용하는 것입니다.

[ListShards](#) 다음 예제에서는 데이터 스트림에서 샤드 목록을 가져오는 방법을 보여줍니다. 이 예제

에 사용된 기본 작업 및 작업에 설정할 수 있는 모든 파라미터에 대한 전체 설명은 을 참조하십시오.

## [ListShards](#)

```
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.ListShardsRequest;
import software.amazon.awssdk.services.kinesis.model.ListShardsResponse;

import java.util.concurrent.TimeUnit;

public class ShardSample {

    public static void main(String[] args) {

        KinesisAsyncClient client = KinesisAsyncClient.builder().build();

        ListShardsRequest request = ListShardsRequest
            .builder().streamName("myFirstStream")
            .build();

        try {
            ListShardsResponse response = client.listShards(request).get(5000,
                TimeUnit.MILLISECONDS);
            System.out.println(response.toString());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

이전 코드 예제를 실행하려면 다음과 같은 POM 파일을 사용할 수 있습니다.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>kinesis.data.streams.samples</groupId>
    <artifactId>shards</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
```

```

    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>8</source>
          <target>8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>software.amazon.awssdk</groupId>
      <artifactId>kinesis</artifactId>
      <version>2.0.0</version>
    </dependency>
  </dependencies>
</project>

```

ListShardsAPI를 사용하면 [ShardFilter](#) 파라미터를 사용하여 API의 응답을 필터링할 수 있습니다. 한 번에 하나의 필터만 지정할 수 있습니다.

ListShardsAPI를 호출할 때 ShardFilter 파라미터를 사용하는 경우 Type 는 필수 속성이므로 반드시 지정해야 합니다. AT\_TRIM\_HORIZON, FROM\_TRIM\_HORIZON 또는 AT\_LATEST 유형을 지정하는 경우 ShardId 또는 Timestamp 선택적 속성을 지정할 필요가 없습니다.

AFTER\_SHARD\_ID 유형을 지정하는 경우 선택적 ShardId 속성의 값도 제공해야 합니다. ShardId 속성은 ListShards API의 ExclusiveStartShardId 매개변수와 기능면에서 동일합니다. ShardId 속성이 지정되면 제공된 ShardId 바로 뒤에 ID가 있는 샤드로 시작하는 샤드가 응답에 포함됩니다.

AT\_TIMESTAMP 또는 FROM\_TIMESTAMP\_ID 유형을 지정하면 선택적 Timestamp 속성의 값도 제공해야 합니다. AT\_TIMESTAMP 유형을 지정하면 제공된 타임스탬프에 열려 있던 모든 샤드가 반환됩니다. FROM\_TIMESTAMP 유형을 지정하면 제공된 타임스탬프부터 TIP까지의 모든 샤드가 반환됩니다.

#### Important

DescribeStreamSummary 및 ListShard API는 데이터 스트림에 대한 정보를 검색할 수 있는 보다 확장 가능한 방법을 제공합니다. 좀 더 구체적으로 설명하자면, DescribeStream API의 할당량으로 인해 병목 현상이 발생할 수 있습니다. 자세한 정보는 [할당량과 제한](#)을 참조하세요. 또한 DescribeStream 할당량은 계정의 모든 데이터 스트림과 상호 작용하는 모든 애

폴리케이션에서 공유된다는 점에 유의하세요. AWS 반면 ListShards API의 할당량은 단일 데이터 스트림에만 적용됩니다. 따라서 ListShards API를 사용하면 더 높은 TPS를 얻을 수 있을 뿐만 아니라 더 많은 데이터 스트림을 생성할수록 작업 규모가 더 잘 확장됩니다. API를 호출하는 모든 생산자와 소비자를 마이그레이션하여 대신 DescribeStream API를 호출하는 것이 좋습니다. DescribeStreamSummary ListShard 이러한 생산자와 소비자를 식별하려면 API 호출에서 KPL 및 KCL용 사용자 에이전트가 캡처되므로 Athena를 사용하여 CloudTrail 로그를 분석하는 것이 좋습니다.

```
SELECT useridentity.sessioncontext.sessionissuer.username,
useridentity.arn,eventname,useragent, count(*) FROM
cloudtrail_logs WHERE Eventname IN ('DescribeStream') AND
eventtime
    BETWEEN ''
    AND ''
GROUP BY
    useridentity.sessioncontext.sessionissuer.username,useridentity.arn,eventname,useragent
ORDER BY count(*) DESC LIMIT 100
```

또한 API를 호출하는 Kinesis Data Streams와의 AWS Lambda 및 Amazon Firehose의 통합은 통합에서 대신 DescribeStream 호출하도록 재구성하는 것이 좋습니다. DescribeStreamSummary ListShards 특히 AWS Lambda의 경우 이벤트 소스 매핑을 업데이트해야 합니다. Amazon Firehose의 경우 ListShards IAM 권한이 포함되도록 해당 IAM 권한을 업데이트해야 합니다.

## DescribeStream API - 더 이상 사용되지 않음

### Important

아래 정보는 API를 통해 데이터 스트림에서 샤드를 검색하는 현재 더 이상 사용되지 않는 방법을 설명합니다. DescribeStream 현재로서는 ListShards API를 사용하여 데이터 스트림을 구성하는 샤드를 검색하는 것이 좋습니다.

describeStream 메서드에 의해 반환되는 응답 객체를 통해 스트림을 구성하는 샤드에 대한 정보를 검색할 수 있습니다. 샤드를 검색하려면 이 객체에 대해 getShards 메서드를 호출합니다. 이 메서드는 단일 호출로 스트림에서 모든 샤드를 반환하지 않을 수 있습니다. 다음 코드에서

getHasMoreShards에 대해 getStreamDescription 메서드를 확인하여 반환되지 않은 추가 샤드가 있는지 확인합니다. 있는 경우, 즉 이 메서드가 true를 반환하면 계속해서 getShards를 반복적으로 호출하여 반환된 샤드의 새로운 배치를 각각 샤드 목록에 추가합니다. getHasMoreShards가 false를 반환하면, 즉 모든 샤드가 반환되면 반복이 종료됩니다. getShards는 EXPIRED 상태인 샤드를 반환하지 않습니다. EXPIRED 상태를 비롯한 샤드 상태에 대한 자세한 내용은 [리샤딩 후 데이터 라우팅, 데이터 지속성 및 샤드 상태](#) 단원을 참조하십시오.

```
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );
List<Shard> shards = new ArrayList<>();
String exclusiveStartShardId = null;
do {
    describeStreamRequest.setExclusiveStartShardId( exclusiveStartShardId );
    DescribeStreamResult describeStreamResult =
client.describeStream( describeStreamRequest );
    shards.addAll( describeStreamResult.getStreamDescription().getShards() );
    if ( describeStreamResult.getStreamDescription().getHasMoreShards() && shards.size()
> 0 ) {
        exclusiveStartShardId = shards.get(shards.size() - 1).getShardId();
    } else {
        exclusiveStartShardId = null;
    }
} while ( exclusiveStartShardId != null );
```

## 스트림 삭제

Kinesis Data Streams 콘솔 또는 프로그래밍 방식으로 스트림을 삭제할 수 있습니다. 프로그래밍 방식으로 스트림을 삭제하려면 다음 코드에 표시된 대로 DeleteStreamRequest를 사용하십시오.

```
DeleteStreamRequest deleteStreamRequest = new DeleteStreamRequest();
deleteStreamRequest.setStreamName(myStreamName);
client.deleteStream(deleteStreamRequest);
```

삭제하기 전에 스트림에서 작동 중인 모든 애플리케이션을 종료하십시오. 애플리케이션이 삭제된 스트림에서 작동을 시도하면 ResourceNotFound 예외가 발생합니다. 또한 이후에 이전 스트림과 이름이 동일한 새 스트림을 생성하고 이전 스트림에서 작동 중인 애플리케이션이 계속 실행되고 있는 경우 이러한 애플리케이션은 이전 스트림인 것처럼 새 스트림과 상호 작용하려고 시도할 수 있으며 이로 인해 예기치 않은 결과가 발생할 수 있습니다.

## 스트림 리샤딩

### Important

API를 사용하여 스트림을 리샤딩할 수 있습니다. [UpdateShardCount](#) 그렇지 않은 경우 여기에 설명된 대로 분할 및 병합을 계속 수행할 수 있습니다.

Amazon Kinesis Data Streams는 스트림을 통과하는 데이터 흐름 속도의 변화에 맞게 스트림의 샤드 수를 조정할 수 있는 리샤딩을 지원합니다. 리샤딩은 고급 작업으로 간주됩니다. Kinesis Data Streams를 처음 사용하는 경우 Kinesis Data Streams의 다른 모든 측면을 숙지한 후 이 주제로 돌아오세요.

샤드 분할과 샤드 병합이라는 두 가지 유형의 리샤딩 작업이 있습니다. 샤드 분할에서는 단일 샤드를 샤드 두 개로 나눕니다. 샤드 병합에서는 샤드 두 개를 단일 샤드로 결합합니다. 리샤딩은 단일 작업으로 두 개를 초과하는 샤드로 분할할 수 없으며, 단일 작업으로 두 개를 초과하는 샤드를 병합할 수 없다는 의미에서 항상 쌍으로 이루어집니다. 리샤딩 작업이 실행되는 샤드 또는 샤드 쌍을 상위 샤드라고 합니다. 리샤딩 작업으로 인해 발생하는 샤드 또는 샤드 쌍을 하위 샤드라고 합니다.

분할로 인해 스트림에서 샤드 수가 증가하므로 스트림의 데이터 용량이 증가합니다. 샤드 수를 기준으로 요금이 부과되므로 분할하면 스트림의 비용이 증가합니다. 마찬가지로 병합하면 스트림의 샤드 수가 줄어들어 스트림의 데이터 용량과 비용이 감소합니다.

일반적으로 리샤딩은 생산자(넣기) 애플리케이션 및 소비자(가져오기) 애플리케이션과 구별되는 관리자 애플리케이션에 의해 수행됩니다. 이러한 관리 애플리케이션은 Amazon에서 제공하는 지표 CloudWatch 또는 생산자와 소비자로부터 수집한 지표를 기반으로 스트림의 전체 성능을 모니터링합니다. 또한 소비자 및 생산자는 일반적으로 리샤딩에 사용되는 API에 대한 액세스 권한이 필요하지 않으므로, 관리자 애플리케이션은 소비자 또는 생산자보다 광범위한 IAM 권한이 있어야 합니다. Kinesis Data Streams의 IAM 권한에 대한 자세한 내용은 [IAM을 사용하여 Amazon Kinesis Data Streams 리소스에 대한 액세스 제어](#) 섹션을 참조하세요.

리샤딩에 대한 자세한 내용은 [Kinesis Data Streams에서 열린 샤드 수를 변경하려면 어떻게 해야 할까요?](#)를 참조하세요.

### 주제

- [리샤딩 전략](#)
- [샤드 분할](#)
- [두 개의 샤드 병합](#)
- [리샤딩 이후](#)

## 리샤딩 전략

Amazon Kinesis Data Streams의 리샤딩 목적은 데이터 흐름 속도의 변화에 따라 스트림을 조정할 수 있도록 하는 것입니다. 스트림의 용량(및 비용)을 늘리려면 샤드를 분할합니다. 스트림의 비용(및 용량)을 줄이려면 샤드를 병합합니다.

리샤딩에 대한 한 가지 접근 방식은 스트림의 모든 샤드를 분할하는 것입니다. 이렇게 하면 스트림 용량이 2배가 됩니다. 그러나 이렇게 하면 실제로 필요한 용량보다 더 많은 추가 용량을 제공하므로 불필요한 비용이 발생합니다.

또한 지표를 사용하여 핫 또는 콜드 샤드, 즉 예상보다 더 많은 데이터 또는 더 적은 데이터를 받는 샤드를 결정할 수 있습니다. 그런 다음 선택적으로 핫 샤드를 분할하여 해당 샤드를 대상으로 지정하는 해시 키에 대한 용량을 늘릴 수 있습니다. 마찬가지로 콜드 샤드를 병합하여 사용하지 않은 용량을 유용하게 사용할 수 있습니다.

Kinesis Data Streams가 게시하는 Amazon CloudWatch 지표에서 스트림에 대한 일부 성능 데이터를 얻을 수 있습니다. 그러나 스트림에 대한 고유한 측정치 중 일부를 수집할 수도 있습니다. 한 가지 접근 방법은 데이터 레코드에 대한 파티션 키에 의해 생성된 해시 키 값을 기록하는 것입니다. 스트림에 레코드를 추가할 때 파티션 키를 지정해야 함을 유의하십시오.

```
putRecordRequest.setPartitionKey( String.format( "myPartitionKey" ) );
```

Kinesis Data Streams는 [MD5](#)를 사용하여 파티션 키에서 해시 키를 계산합니다. 레코드에 대한 파티션 키를 지정하므로 MD5를 사용하여 해당 레코드에 대한 해시 키 값을 계산하고 기록할 수 있습니다.

또한 데이터 레코드가 할당된 샤드 ID도 기록할 수 있습니다. `getShardId` 메서드에 의해 반환된 `putRecordResults` 객체와 `putRecords` 메서드에 의해 반환된 `putRecordResult` 객체의 `putRecord` 메서드를 통해 샤드 ID를 사용할 수 있습니다.

```
String shardId = putRecordResult.getShardId();
```

샤드 ID와 해시 키 값을 사용하면 가장 많은 또는 가장 적은 트래픽을 받는 샤드 및 해시 키를 결정할 수 있습니다. 그런 다음 리샤딩을 사용하여 이러한 키에 적합하게 더 많은 또는 더 적은 용량을 제공할 수 있습니다.

## 샤드 분할

Amazon Kinesis Data Streams에서 샤드를 분할하려면 해시 키 값을 상위 샤드에서 하위 샤드로 재배포하는 방법을 지정해야 합니다. 스트림에 데이터 레코드를 추가하면 해시 키 값에 따라 샤드에 할당됨

니다. 해시 키 값은 스트림에 데이터 레코드를 추가할 때 데이터 레코드에 대해 지정하는 파티션 키의 [MD5](#) 해시입니다. 동일한 파티션 키가 있는 데이터 레코드에는 동일한 해시 키 값도 있습니다.

지정된 샤드에 대해 가능한 해시 키 값은 정렬된 연속적인 음수가 아닌 정수의 집합을 구성합니다. 가능한 해시 키 값의 범위는 다음에 의해 지정됩니다.

```
shard.getHashKeyRange().getStartingHashKey();
shard.getHashKeyRange().getEndingHashKey();
```

샤드를 분할할 때 이 범위의 값을 지정합니다. 해당 해시 키 값과 더 높은 모든 해시 키 값은 하위 샤드 중 하나로 배포됩니다. 더 낮은 모든 해시 키 값은 다른 하위 샤드로 배포됩니다.

다음 코드는 각 하위 샤드에서 해시 키를 고르게 재배포하고 기본적으로 상위 샤드를 절반으로 분할하는 샤드 분할 작업을 보여줍니다. 이는 상위 샤드를 분할할 수 있는 방법 중 하나일 뿐입니다. 예를 들어, 상위에서 키의 하위 3분의 1이 하위 샤드 하나로 이동하고, 키의 상위 3분의 2가 다른 하위 샤드로 이동하도록 샤드를 분할할 수 있습니다. 그러나 여러 애플리케이션의 경우 샤드를 절반으로 분할하는 것이 효과적인 방식입니다.

이 코드에서는 `myStreamName`이 스트림의 이름을 보유하고 객체 변수 `shard`가 분할할 샤드를 보유하고 있다고 가정합니다. 먼저 새 `splitShardRequest` 객체를 인스턴스화하고 스트림 이름과 샤드 ID를 설정합니다.

```
SplitShardRequest splitShardRequest = new SplitShardRequest();
splitShardRequest.setStreamName(myStreamName);
splitShardRequest.setShardToSplit(shard.getShardId());
```

샤드에서 가장 낮은 값과 가장 높은 값의 절반으로 해시 키 값을 결정합니다. 이는 상위 샤드에서 해시 키의 상위 절반을 포함하는 하위 샤드에 대한 시작 해시 키 값입니다. `setNewStartingHashKey` 메서드에서 이 값을 지정합니다. 이 값만 지정하면 됩니다. Kinesis Data Streams가 분할에 의해 생성된 다른 하위 샤드에 이 값 미만의 해시 키를 자동으로 배포합니다. 마지막 단계는 Kinesis Data Streams 클라이언트에서 `splitShard` 메서드를 직접적으로 호출하는 것입니다.

```
BigInteger startingHashKey = new
    BigInteger(shard.getHashKeyRange().getStartingHashKey());
BigInteger endingHashKey = new
    BigInteger(shard.getHashKeyRange().getEndingHashKey());
String newStartingHashKey = startingHashKey.add(endingHashKey).divide(new
    BigInteger("2")).toString();

splitShardRequest.setNewStartingHashKey(newStartingHashKey);
```



```
client.splitShard(splitShardRequest);
```

이 절차 이후 첫 번째 단계는 [스트림이 다시 활성 상태가 될 때까지 대기](#)에 표시됩니다.

## 두 개의 샤드 병합

샤드 병합 작업은 지정된 샤드 두 개를 가져와 단일 샤드로 결합합니다. 병합 이후 단일 하위 샤드는 두 개의 상위 샤드가 포함하는 모든 해시 키 값에 대한 데이터를 받습니다.

### 샤드 인접

두 개의 샤드를 병합하려면 샤드가 인접해야 합니다. 두 개의 샤드에 대한 해시 키 범위의 조합이 간격이 없는 연속적인 집합을 이룰 경우 두 개의 샤드는 인접했다고 간주됩니다. 예를 들어, 두 개의 샤드가 있으며 샤드 하나의 해시 키 범위가 276...381이고 다른 샤드 하나의 해시 키 범위가 382...454라고 가정하면, 이 두 개의 샤드를 해시 키 범위가 276...454인 단일 샤드로 병합할 수 있습니다.

또 다른 예를 들어, 두 개의 샤드가 있으며 샤드 하나의 해시 키 범위가 276...381이고 다른 샤드 하나의 해시 키 범위가 455...560이라고 가정하면, 이 두 개의 샤드 간에는 범위가 382..454인 하나 이상의 샤드가 있으므로 이 두 개의 샤드를 병합할 수 없습니다.

스트림의 모든 OPEN 샤드 세트(그룹)는 항상 MD5 해시 키 값의 전체 범위에 걸쳐 있습니다. CLOSED와 같은 샤드 상태에 대한 자세한 내용은 [리샤딩 후 데이터 라우팅, 데이터 지속성 및 샤드 상태](#) 섹션을 참조하세요.

병합을 위한 후보인 샤드를 식별하려면 CLOSED 상태의 모든 샤드를 필터링해야 합니다. OPEN 상태, 즉 CLOSED 상태가 아닌 샤드의 종료 시퀀스 번호는 null입니다. 다음을 사용하여 샤드에 대한 종료 시퀀스 번호를 테스트할 수 있습니다.

```
if( null == shard.getSequenceNumberRange().getEndingSequenceNumber() )
{
    // Shard is OPEN, so it is a possible candidate to be merged.
}
```

닫힌 샤드를 필터링한 후 각 샤드에 의해 지원되는 가장 높은 해시 키 값을 기준으로 남은 샤드를 정렬합니다. 다음을 사용하여 이 값을 검색할 수 있습니다.

```
shard.getHashKeyRange().getEndingHashKey();
```

필터링되고 정렬된 목록에서 두 개의 샤드가 인접한 경우 해당 샤드를 병합할 수 있습니다.

## 병합 작업에 대한 코드

다음 코드는 두 개의 샤드를 병합합니다. 이 코드에서는 myStreamName이 스트림의 이름을 보유하고 객체 변수 shard1 및 shard2가 병합할 두 인접 샤드를 보유한다고 가정합니다.

병합 작업의 경우 먼저 새 mergeShardsRequest 객체를 인스턴스화합니다. setStreamName 메서드로 스트림 이름을 지정합니다. 그런 다음 setShardToMerge 및 setAdjacentShardToMerge 메서드를 사용하여 병합할 두 개의 샤드를 지정합니다. 마지막으로 Kinesis Data Streams 클라이언트에서 mergeShards 메서드를 직접적으로 호출하여 작업을 수행합니다.

```
MergeShardsRequest mergeShardsRequest = new MergeShardsRequest();
mergeShardsRequest.setStreamName(myStreamName);
mergeShardsRequest.setShardToMerge(shard1.getShardId());
mergeShardsRequest.setAdjacentShardToMerge(shard2.getShardId());
client.mergeShards(mergeShardsRequest);
```

이 절차 이후 첫 번째 단계는 [스트림이 다시 활성화 상태가 될 때까지 대기](#)에 표시됩니다.

## 리샤딩 이후

Amazon Kinesis Data Streams의 리샤딩 절차 이후 일반 레코드 처리를 재개하기 전에 다른 절차 및 고려 사항이 필요합니다. 다음 단원에서는 이에 대해 설명합니다.

### 주제

- [스트림이 다시 활성화 상태가 될 때까지 대기](#)
- [리샤딩 후 데이터 라우팅, 데이터 지속성 및 샤드 상태](#)

## 스트림이 다시 활성화 상태가 될 때까지 대기

splitShard 또는 mergeShards의 리샤딩 작업을 호출한 후 스트림이 다시 활성화 상태가 될 때까지 기다려야 합니다. 사용할 코드는 [스트림 생성](#) 후 스트림이 활성화될 때까지 대기할 경우와 동일합니다. 코드는 다음과 같습니다.

```
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );

long startTime = System.currentTimeMillis();
long endTime = startTime + ( 10 * 60 * 1000 );
while ( System.currentTimeMillis() < endTime )
{
```

```

try {
    Thread.sleep(20 * 1000);
}
catch ( Exception e ) {}

try {
    DescribeStreamResult describeStreamResponse =
client.describeStream( describeStreamRequest );
    String streamStatus =
describeStreamResponse.getStreamDescription().getStreamStatus();
    if ( streamStatus.equals( "ACTIVE" ) ) {
        break;
    }
    //
    // sleep for one second
    //
    try {
        Thread.sleep( 1000 );
    }
    catch ( Exception e ) {}
}
catch ( ResourceNotFoundException e ) {}
}
if ( System.currentTimeMillis() >= endTime )
{
    throw new RuntimeException( "Stream " + myStreamName + " never went active" );
}

```

## 리샤딩 후 데이터 라우팅, 데이터 지속성 및 샤드 상태

Kinesis Data Streams는 실시간 데이터 스트리밍 서비스입니다. 즉, 애플리케이션에서는 데이터가 스트림의 샤드를 통해 지속적으로 흐르고 있다고 가정해야 합니다. 리샤딩하면 상위 샤드로 이동하는 데이터 레코드가 데이터 레코드 파티션 키가 매핑되는 해시 키 값에 따라 하위 샤드로 이동하도록 다시 라우팅됩니다. 그러나 리샤딩 전에 상위 샤드에 있는 데이터 레코드는 해당 샤드에 유지됩니다. 즉, 리샤딩이 발생할 때 상위 샤드가 사라지지 않습니다. 리샤딩 전에 포함하는 데이터와 함께 유지됩니다. 상위 샤드의 데이터 레코드는 Kinesis Data Streams API의 [getShardIterator](#) 및 [getRecords](#) 작업을 사용하거나 Kinesis Client Library를 통해 액세스할 수 있습니다.

### Note

데이터 레코드가 스트림에 추가된 시점으로부터 현재 보존 기간까지 데이터 레코드에 액세스할 수 있습니다. 이는 해당 기간 동안 스트림의 샤드가 변경되는지 여부와 관계없이 마찬가지

입니다. 스트림의 보존 기간에 대한 자세한 내용은 [데이터 보존 기간 변경](#) 단원을 참조하십시오.

리샤딩 프로세스에서 상위 샤드가 OPEN 상태에서 CLOSED 상태로, EXPIRED 상태로 전환됩니다.

- OPEN: 리샤딩 작업 전 상위 샤드는 OPEN 상태입니다. 즉, 데이터 레코드를 샤드에 추가하고 샤드에서 검색할 수 있습니다.
- CLOSED: 리샤딩 작업 후 상위 샤드가 CLOSED 상태로 전환됩니다. 즉, 데이터 레코드가 더 이상 샤드에 추가되지 않습니다. 이 샤드에 추가된 데이터 레코드는 이제 하위 샤드에 대신 추가됩니다. 그러나 제한된 기간 동안 계속 샤드에서 데이터 레코드를 검색할 수 있습니다.
- EXPIRED: 스트림의 보존 기간이 만료된 후 상위 샤드에 있는 모든 데이터 레코드가 만료되며 더 이상 액세스할 수 없습니다. 이때 샤드 자체는 EXPIRED 상태로 전환됩니다. 스트림에 샤드를 열거하기 위한 `getStreamDescription().getShards`를 호출하면 반환된 목록 샤드에 EXPIRED 샤드가 포함되지 않습니다. 스트림의 보존 기간에 대한 자세한 내용은 [데이터 보존 기간 변경](#) 단원을 참조하십시오.

리샤딩이 발생하고 스트림이 다시 ACTIVE 상태가 된 후 즉시 하위 샤드에서 데이터를 읽기 시작할 수 있습니다. 하지만 리샤드 이후에 남아 있는 상위 샤드에는 리샤드 이전에 스트림에 추가된 아직 읽지 않은 데이터가 여전히 포함되어 있을 수 있습니다. 상위 샤드에서 모든 데이터를 읽기 전에 하위 샤드에서 데이터를 읽는 경우 데이터 레코드의 시퀀스 번호에 의해 지정된 순서가 아닌 다른 순서로 특정 해시 키에 대한 데이터를 읽을 수 있습니다. 따라서 데이터 순서가 중요하다고 간주하는 경우, 리샤딩 후 다 읽을 때까지 항상 상위 샤드에서 데이터를 계속 읽어야 합니다. 그런 다음 하위 샤드에서 데이터를 읽기 시작해야 합니다. `getRecordsResult.getNextShardIterator`가 null을 반환하면 상위 샤드에서 모든 데이터를 읽었음을 나타냅니다. Kinesis Client Library를 사용하여 데이터를 읽는 경우 리샤드가 발생한 후 데이터를 순서대로 받지 못할 수 있습니다.

## 데이터 보존 기간 변경

Amazon Kinesis Data Streams는 데이터 스트림의 데이터 레코드 보존 기간에 대한 변경을 지원합니다. Kinesis 데이터 스트림은 실시간으로 읽고 쓸 수 있는 정렬된 순서의 데이터 레코드입니다. 따라서 데이터 레코드는 스트림의 샤드에 일시적으로 저장됩니다. 데이터가 추가된 시점부터 더 이상 액세스할 수 없는 시점까지의 기간을 보관 기간이라고 합니다. Kinesis 데이터 스트림은 기본적으로 24시간부터 최대 8,760시간(365일)까지 레코드를 저장합니다.

Kinesis Data Streams 콘솔을 통해 또는 및 [DecreaseStreamRetentionPeriod](#)작업을 사용하여 [IncreaseStreamRetentionPeriod](#)보존 기간을 업데이트할 수 있습니다. Kinesis Data Streams

콘솔을 사용하면 둘 이상의 데이터 스트림의 보존 기간을 동시에 일괄 편집할 수 있습니다.

[IncreaseStreamRetentionPeriod](#)작업 또는 Kinesis Data Streams 콘솔을 사용하여 보존 기간을 최대 8760시간 (365일) 까지 늘릴 수 있습니다. [DecreaseStreamRetentionPeriod](#)작업 또는 Kinesis Data Streams 콘솔을 사용하여 보존 기간을 최소 24시간으로 줄일 수 있습니다. 두 작업에 대한 요청 구문에는 스트림 이름과 보존 기간(시간)이 포함됩니다. 마지막으로 [DescribeStream](#)작업을 호출하여 스트림의 현재 보존 기간을 확인할 수 있습니다.

다음은 AWS CLI를 사용하여 보존 기간을 변경하는 예제입니다.

```
aws kinesis increase-stream-retention-period --stream-name retentionPeriodDemo --
retention-period-hours 72
```

Kinesis Data Streams는 보존 기간이 증가한 몇 분 내에 이전 보존 기간에서 레코드에 액세스할 수 있도록 합니다. 예를 들어, 보존 기간을 24시간에서 48시간으로 변경하면 23시간 55분 전에 스트림에 추가된 레코드는 24시간 후에도 계속 사용할 수 있습니다.

보존 기간이 감소되면 Kinesis Data Streams는 새 보존 기간보다 이전인 레코드를 즉시 액세스할 수 없도록 합니다. 따라서 오퍼레이션을 호출할 때는 각별히 주의해야 [DecreaseStreamRetentionPeriod](#)합니다.

문제가 발생할 경우 데이터가 만료되기 전에 소비자가 데이터를 읽을 수 있도록 데이터 보존 기간을 설정하십시오. 레코드 처리 로직 문제 또는 장기간 동안 다운스트림 종속성이 중단된 문제 등 모든 가능성을 신중하게 고려해야 합니다. 보존 기간은 데이터 소비자가 복구하는 데 많은 시간을 허용하는 안전망으로 생각해야 합니다. 보존 기간 API 작업을 통해 이 기간을 사전에 설정하거나 운영 이벤트에 대해 사후 예방적으로 대응할 수 있습니다.

24시간을 초과하여 보존 기간을 설정하면 스트림에 추가 요금이 적용됩니다. 자세한 내용은 [Amazon Kinesis Data Streams 요금](#)을 참조하십시오.

## Amazon Kinesis Data Streams의 스트림 태그 지정

Amazon Kinesis Data Streams에서 생성한 스트림에 고유한 메타데이터를 태그 형태로 할당할 수 있습니다. 태그는 스트림에 대해 정의된 키-값 페어입니다. 태그를 사용하면 간단하면서도 효과적으로 AWS 리소스를 관리하고 결제 데이터를 포함하여 데이터를 구성할 수 있습니다.

목차

- [태그 기본 사항](#)
- [태그 지정을 사용하여 비용 추적](#)
- [태그 제한](#)

- [Kinesis Data Streams 콘솔을 사용하여 스트림 태그 지정](#)
- [AWS CLI를 사용하여 스트림에 태깅](#)
- [Kinesis Data Streams API를 사용하여 스트림 태그 지정](#)

## 태그 기본 사항

Kinesis Data Streams 콘솔, AWS CLI 또는 Kinesis Data Streams API를 사용하여 다음 작업을 완료합니다.

- 스트림에 태그 추가
- 스트림에 대한 태그 나열
- 스트림에서 태그 제거

태그를 사용하여 스트림을 분류할 수 있습니다. 예를 들어, 용도, 소유자 또는 환경별로 스트림을 분류할 수 있습니다. 각 태그에 대해 키와 값이 정의되기 때문에 특정 요구를 충족하는 사용자 지정 범주 세트를 생성할 수 있습니다. 예를 들어, 태그 세트를 정의하여 소유자 및 연관된 애플리케이션에 따라 스트림을 추적할 수 있습니다. 다음은 태그의 몇 가지 예제입니다.

- 프로젝트: 프로젝트 이름
- 소유자: 이름
- 용도: 로드 테스트
- 애플리케이션: 애플리케이션 이름
- 환경: 프로덕션

## 태그 지정을 사용하여 비용 추적

태그를 사용하여 AWS 비용을 분류하고 추적할 수 있습니다. 스트림을 비롯하여 AWS 리소스에 태그를 적용할 때 AWS 비용 할당 보고서에는 태그별로 집계된 사용 내역 및 비용이 포함됩니다. 비즈니스 범주를 나타내는 태그(예: 비용 센터, 애플리케이션 이름 또는 소유자)를 적용하여 여러 서비스에 대한 비용을 정리할 수 있습니다. 자세한 내용은 AWS Billing 사용 설명서의 [사용자 지정 결제 보고서에 비용 할당 태그 사용](#)을 참조하세요.

## 태그 제한

태그에 적용되는 제한은 다음과 같습니다.

## 기본 제한

- 리소스(스트림)당 최대 태그 수는 50개입니다.
- 태그 키와 값은 대/소문자를 구분합니다.
- 삭제된 스트림에 대해 태그를 변경하거나 편집할 수 없습니다.

## 태그 키 제한

- 각 태그 키는 고유해야 합니다. 이미 사용 중인 키를 가진 태그를 추가하면 기존 키-값 페어에 새 태그가 덮어쓰기 됩니다.
- `aws:`를 사용하여 태그 키를 시작할 수 없습니다. 이 접두사는 AWS용으로 예약되어 있기 때문입니다. AWS는 이 접두사로 시작되는 태그를 생성하지만, 사용자는 이를 편집하거나 삭제할 수 없습니다.
- 태그 키의 길이는 유니코드 1~128자여야 합니다.
- 태그 키의 문자로는 유니코드 문자, 숫자, 공백, 그리고 `_ . / = + - @` 같은 특별 문자가 허용됩니다.

## 태그 값 제한

- 태그 값의 길이는 유니코드 0~255자여야 합니다.
- 태그 값은 공백 상태로 둘 수 있습니다. 아니면 유니코드 문자, 숫자, 공백, 그리고 `_ . / = + - @` 같은 특별 문자를 사용할 수 있습니다.

## Kinesis Data Streams 콘솔을 사용하여 스트림 태그 지정

Kinesis Data Streams 콘솔을 사용하여 태그를 추가, 나열 및 제거할 수 있습니다.

### 스트림에 대한 태그를 보려면

1. Kinesis Data Streams 콘솔을 엽니다. 탐색 모음에서 리전 선택기를 확장하고 리전을 선택합니다.
2. Stream List(스트림 목록) 페이지에서 스트림을 선택합니다.
3. Stream Details(스트림 세부 정보) 페이지에서 Tags(태그) 탭을 클릭합니다.

### 스트림에 태그를 추가하려면

1. Kinesis Data Streams 콘솔을 엽니다. 탐색 모음에서 리전 선택기를 확장하고 리전을 선택합니다.

2. Stream List(스트림 목록) 페이지에서 스트림을 선택합니다.
3. Stream Details(스트림 세부 정보) 페이지에서 Tags(태그) 탭을 클릭합니다.
4. Key(키) 필드에서 태그 키를 지정하고, 선택적으로 Value(값) 필드에서 태그 값을 지정한 다음, Add Tag(태그 추가)를 클릭합니다.

Add Tag(태그 추가) 버튼이 활성화되지 않는 경우 지정한 태그 키 또는 태그 값이 태그 제한 사항을 충족하지 않는 것입니다. 자세한 내용은 [태그 제한](#) 섹션을 참조하세요.

5. Tags(태그) 탭의 목록에서 새 태그를 보려면 새로 고침 아이콘을 클릭합니다.

스트림에서 태그를 제거하려면

1. Kinesis Data Streams 콘솔을 엽니다. 탐색 모음에서 리전 선택기를 확장하고 리전을 선택합니다.
2. [Stream List] 페이지에서 스트림을 선택합니다.
3. Stream Details(스트림 세부 정보) 페이지에서 Tags(태그) 탭을 클릭한 다음, 태그에 대한 Remove(제거) 아이콘을 클릭합니다.
4. Delete Tag(태그 삭제) 대화 상자에서 Yes, Delete(예, 삭제합니다)를 클릭합니다.

## AWS CLI를 사용하여 스트림에 태깅

AWS CLI를 사용하여 태그를 추가, 나열 및 제거할 수 있습니다. 예제는 다음 설명서를 참조하십시오.

### [add-tags-to-stream](#)

지정된 스트림에 대한 태그를 추가 또는 업데이트합니다.

### [list-tags-for-stream](#)

지정된 스트림에 대한 태그를 나열합니다.

### [remove-tags-from-stream](#)

지정된 스트림에 대한 태그를 제거합니다.

## Kinesis Data Streams API를 사용하여 스트림 태그 지정

Kinesis Data Streams API를 사용하여 태그를 추가, 나열 및 제거할 수 있습니다. 예제는 다음 설명서를 참조하세요.



## [AddTagsToStream](#)

지정된 스트림에 대한 태그를 추가 또는 업데이트합니다.

## [ListTagsForStream](#)

지정된 스트림에 대한 태그를 나열합니다.

## [RemoveTagsFromStream](#)

지정된 스트림에 대한 태그를 제거합니다.

# Amazon Kinesis Data Streams에 데이터 쓰기

생산자는 Amazon Kinesis Data Streams에 데이터를 쓰는 애플리케이션입니다. AWS SDK for Java와 Kinesis Producer Library를 사용하여 Kinesis Data Streams를 위한 생산자를 만들 수 있습니다.

Kinesis Data Streams를 처음 사용하는 경우, 먼저 [Amazon Kinesis Data Streams란?](#) 및 [Amazon Kinesis Data Streams 시작하기](#)에 있는 개념과 용어를 알아 두세요.

## Important

Kinesis Data Streams는 데이터 스트림의 데이터 레코드 보존 기간에 대한 변경을 지원합니다. 자세한 내용은 [데이터 보존 기간 변경](#) 섹션을 참조하세요.

스트림에 데이터를 넣으려면 스트림 이름, 파티션 키 및 스트림에 추가할 데이터 BLOB를 지정해야 합니다. 데이터 레코드를 추가할 스트림의 샤드를 결정하기 위해 파티션 키가 사용됩니다.

샤드의 모든 데이터는 샤드를 처리하는 동일한 작업자에게 전송됩니다. 애플리케이션 로직에 따라 사용할 파티션 키가 결정됩니다. 일반적으로 파티션 키의 수는 샤드 수보다 훨씬 커야 합니다. 데이터 레코드를 특정 샤드에 매핑하는 방법을 결정하기 위해 파티션 키가 사용되기 때문입니다. 파티션 키의 수가 충분하면 스트림의 샤드에 데이터를 균등하게 배포할 수 있습니다.

## 목차

- [Amazon Kinesis Producer Library를 사용하여 생산자 개발](#)
- [AWS SDK for Java와 Amazon Kinesis Data Streams API를 사용하여 생산자 개발](#)
- [Kinesis 에이전트를 사용하여 Amazon Kinesis Data Streams에 쓰기](#)
- [다른 AWS 서비스를 사용하여 Kinesis Data Streams에 쓰기](#)
- [타사 통합 사용](#)
- [Amazon Kinesis Data Streams 생산자 문제 해결](#)
- [Kinesis Data Streams 생산자를 위한 고급 주제](#)

## Amazon Kinesis Producer Library를 사용하여 생산자 개발

Amazon Kinesis Data Streams 생산자는 Kinesis 데이터 스트림에 사용자 데이터 레코드를 입력하는 (데이터 수집이라고도 함) 애플리케이션입니다. Kinesis Producer Library(KPL)는 개발자가 Kinesis 데이터 스트림에 대한 높은 쓰기 처리량을 달성할 수 있도록 생산자 애플리케이션 개발을 간소화합니다.

CloudWatchAmazon에서 KPL을 모니터링할 수 있습니다. 자세한 내용은 [Amazon을 통한 Kinesis 프로듀서 라이브러리 모니터링 CloudWatch](#) 단원을 참조하십시오.

## 목차

- [KPL의 역할](#)
- [KPL 사용의 장점](#)
- [KPL을 사용하지 말아야 할 경우](#)
- [KPL 설치](#)
- [KPL\(Kinesis Producer Library\)의 Amazon Trust Services\(ATS\) 인증서로 전환](#)
- [KPL 지원 플랫폼](#)
- [KPL 주요 개념](#)
- [KPL과 생산자 코드의 통합](#)
- [KPL을 사용하여 Kinesis 데이터 스트림에 쓰기](#)
- [Kinesis Producer Library 구성](#)
- [소비자 분해](#)
- [Firehose와 함께 KPL 사용하기](#)
- [AWS Glue 스키마 레지스트리와 함께 KPL 사용](#)
- [KPL 프록시 구성](#)

### Note

최신 KPL 버전으로 업그레이드하는 것이 좋습니다. KPL은 모두 최신 종속성 및 보안 패치, 버그 수정, 이전 버전과 호환되는 새 기능이 포함된 최신 릴리스로 정기적으로 업데이트됩니다. 자세한 내용은 <https://github.com/aws-labs/amazon-kinesis-producer/releases/>를 참조하십시오.

## KPL의 역할

KPL은 Kinesis easy-to-use 데이터 스트림에 쓰는 데 도움이 되는 고도로 구성 가능한 라이브러리입니다. 이 라이브러리는 생산자 애플리케이션 코드와 Kinesis Data Streams API 작업 간에 중간 역할을 합니다. KPL은 다음과 같은 기본 작업을 수행합니다.

- 자동 및 구성 가능한 재시도 메커니즘을 사용하여 하나 이상의 Kinesis 데이터 스트림에 쓰기

- 레코드를 수집하고 PutRecords를 사용하여 요청당 여러 샤드에 여러 레코드 쓰기
- 사용자 레코드를 집계하여 페이로드 크기 증가 및 처리량 향상
- [Kinesis Client Library\(KCL\)](#)와 자연스럽게 통합하여 소비자에서 배치 처리된 레코드 분해
- 생산자 성과에 대한 가시성을 제공하기 위해 사용자를 대신하여 Amazon CloudWatch 지표를 제출합니다.

KPL은 [AWS SDK](#)에서 사용할 수 있는 Kinesis Data Streams API와 다릅니다. Kinesis Data Streams API는 Kinesis Data Streams의 여러 가지 측면(스트림 생성, 리샤딩, 데이터 넣기 및 가져오기 포함)을 관리하는 데 도움이 되는 반면, KPL은 특히 데이터 수집을 위한 추상화 계층을 제공합니다. Kinesis Data Streams API에 대한 자세한 내용은 [Amazon Kinesis API 참조](#)를 확인하세요.

## KPL 사용의 장점

다음 목록은 Kinesis Data Streams 생산자 개발을 위해 KPL을 사용할 때 경험할 수 있는 몇 가지 장점을 보여줍니다.

동기 또는 비동기 사용 사례에서 KPL을 사용할 수 있습니다. 특별히 동기 동작을 사용할 이유가 없다면 고성능 비동기 인터페이스를 사용하는 것이 좋습니다. 두 사용 사례와 예제 코드에 대한 자세한 내용은 [KPL을 사용하여 Kinesis 데이터 스트림에 쓰기](#)를 참조하십시오.

### 성능 이점

KPL은 고성능 생산자를 구축하는 데 도움이 됩니다. Amazon EC2 인스턴스가 수백 또는 수천 개의 저전력 디바이스에서 100바이트 이벤트를 수집하고 Kinesis 데이터 스트림에 레코드를 쓰기 위해 프록시 역할을 한다고 가정해 보세요. 이 EC2 인스턴스는 초당 수천 개의 이벤트를 데이터 스트림에 써야 합니다. 필요한 처리량을 달성하기 위해 생산자는 소비자 측에서 이루어지는 레코드 분해 및 재시도 로직 외에도 일괄 처리나 멀티스레딩과 같은 복잡한 로직을 구현해야 합니다. KPL은 이 모든 작업을 수행합니다.

### 소비자 측 사용 편의성

Java에서 KCL을 사용하는 소비자 측 개발자의 경우 추가 작업 없이 KPL이 통합됩니다. KCL이 여러 KPL 사용자 레코드로 이루어진 집계된 Kinesis Data Streams 레코드를 검색할 때 KPL을 자동으로 간접적으로 호출하여 개별 사용자 레코드를 추출한 후 사용자에게 반환합니다.

KCL 대신 API 작업 GetRecords를 직접 사용하는 소비자 측 개발자의 경우 KPL Java 라이브러리를 통해 개별 사용자 레코드를 추출한 후 사용자에게 반환할 수 있습니다.

## 생산자 모니터링

CloudWatch Amazon 및 KPL을 사용하여 Kinesis Data Streams 생산자를 수집, 모니터링 및 분석할 수 있습니다. KPL은 사용자를 CloudWatch 대신하여 처리량, 오류 및 기타 지표를 내보내며 스트림, 샤드 또는 프로듀서 수준에서 모니터링하도록 구성할 수 있습니다.

## 비동기 아키텍처

KPL은 레코드를 Kinesis Data Streams로 보내기 전에 버퍼링할 수 있으므로 실행을 계속하기 전에 레코드가 서버에 도착했는지 확인하기 위해 차단하고 대기하도록 호출자 애플리케이션을 강제하지 않습니다. KPL에 레코드를 넣는 호출은 언제나 즉시 반환되며 전송할 레코드 또는 서버에서 수신할 응답을 기다리지 않습니다. 대신 나중에 Kinesis Data Streams로 레코드를 보낸 결과를 수신하는 Future 객체가 생성됩니다. 이는 SDK의 비동기 클라이언트와 동일한 동작입니다. AWS

## KPL을 사용하지 말아야 할 경우

KPL은 라이브러리 내에 RecordMaxBufferedTime까지 추가 처리 지연을 일으킬 수 있습니다(사용자 구성 가능). RecordMaxBufferedTime 값이 클수록 패킹 효율이 높아지고 성능이 향상됩니다. 이 추가 지연을 허용할 수 없는 애플리케이션은 AWS SDK를 직접 사용해야 할 수도 있습니다. Kinesis Data Streams와 함께 AWS SDK를 사용하는 방법에 대한 자세한 내용은 [AWS SDK for Java와 Amazon Kinesis Data Streams API를 사용하여 생산자 개발](#) RecordMaxBufferedTime 및 그 밖에 사용자가 구성할 수 있는 KPL의 속성에 대한 자세한 내용은 [Kinesis Producer Library 구성](#) 섹션을 참조하세요.

## KPL 설치

Amazon에서는 macOS, Windows 및 최신 Linux 배포를 위해 사전 빌드된 C++ Kinesis Producer Library(KPL)의 바이너리를 제공합니다. 지원되는 플랫폼 세부 정보는 다음 섹션을 참조하세요. 이 바이너리는 Java .jar 파일의 일부로 패키징되며 Maven을 사용하여 패키지를 설치하면 자동으로 호출 및 사용됩니다. KPL 및 KCL의 최신 버전을 찾으려면 다음 Maven 검색 링크를 사용하세요.

- [KPL](#)
- [KCL](#)

GCC(GNU Compiler Collection)로 컴파일된 Linux 바이너리는 Linux의 libstdc++에 정적으로 연결됩니다. glibc 버전 2.5 이상이 포함된 64비트 Linux 배포에서 작동합니다.

이전 Linux 배포판 사용자는 소스와 함께 제공된 빌드 지침을 사용하여 KPL을 빌드할 수 있습니다. GitHub 에서 GitHub KPL을 다운로드하려면 [Kinesis 프로듀서 라이브러리를](#) 참조하십시오.

## KPL(Kinesis Producer Library)의 Amazon Trust Services(ATS) 인증서로 전환

2018년 2월 9일 오전 9시(PST)에 Amazon Kinesis Data Streams가 ATS 인증서를 설치했습니다. Kinesis Producer Library(KPL)를 사용하여 레코드를 계속 Kinesis Data Streams에 쓰려면 KPL 설치를 [버전 0.12.6](#) 이상으로 업그레이드해야 합니다. 이 변경 사항은 모든 AWS 지역에 적용됩니다.

ATS로의 전환에 대한 자세한 내용은 [자체 인증 기관으로의 이전 준비 방법을](#) 참조하십시오. AWS

문제를 발견하고 기술 지원이 필요한 경우 AWS Support Center에서 <https://console.aws.amazon.com/support/v1#/case/create> 사례를 작성합니다.

### KPL 지원 플랫폼

Kinesis Producer Library(KPL)는 C++로 작성되고 주 사용자 프로세스의 하위 프로세스로 실행됩니다. 미리 컴파일된 64비트 기본 바이너리가 Java 릴리스에 번들로 제공되며 Java 래퍼로 관리됩니다.

다음과 같은 운영 체제에서 추가 라이브러리를 설치하지 않고도 Java 패키지가 실행됩니다.

- 커널 2.6.18(2006년 9월) 이상의 Linux 배포
- Apple OS X 10.9 이상
- Windows Server 2008 이상

#### Important

Windows Server 2008 이상은 버전 0.14.0까지의 모든 KPL 버전에 대해 지원됩니다. Windows 플랫폼은 KPL 버전 0.14.0 이상부터 지원되지 않습니다.

KPL은 64비트 전용입니다.

### 소스 코드

KPL 설치에 제공된 바이너리가 환경에 맞지 않으면 KPL의 핵심 내용이 C++ 모듈로 작성됩니다. C++ 모듈과 Java 인터페이스의 소스 코드는 Amazon 퍼블릭 라이선스에 따라 출시되었으며 [Kinesis](#) 프로듀서 GitHub 라이브러리에서 사용할 수 있습니다. 최근 표준 호환 C++ 컴파일러 및 JRE를 사용할 수 있는 모든 플랫폼에서 KPL을 사용할 수 있지만 Amazon에서는 지원되는 플랫폼 목록에 없는 다른 플랫폼을 공식적으로 지원하지 않습니다.

## KPL 주요 개념

다음 섹션에는 Kinesis Producer Library(KPL)를 이해하고 활용하는 데 필요한 개념과 용어가 나와 있습니다.

주제

- [레코드](#)
- [배칭](#)
- [집계](#)
- [수집](#)

### 레코드

이 안내서에서는 KPL 사용자 레코드와 Kinesis Data Streams 레코드를 구별합니다. 한정자 없이 레코드라는 용어를 사용할 때는 KPL 사용자 레코드를 의미하고 Kinesis Data Streams 레코드를 언급할 때 Kinesis Data Streams 레코드라고 명시적으로 말합니다.

KPL 사용자 레코드는 사용자에게 특정 의미를 갖는 데이터 blob입니다. 예를 들어, 웹 사이트에서 UI 이벤트를 나타내는 JSON BLOB이나 웹 서버의 로그 항목이 있습니다.

Kinesis Data Streams 레코드는 Kinesis Data Streams 서비스 API에서 정의한 Record 데이터 구조의 인스턴스입니다. 파티션 키, 시퀀스 번호 및 데이터 BLOB이 여기에 포함됩니다.

### 배칭

일괄 처리는 개별 항목으로 반복 작업을 하는 대신 여러 항목으로 단일 작업을 수행하는 것을 의미합니다.

이 컨텍스트에서 '항목'은 레코드이며 작업은 항목을 Kinesis Data Streams로 전송하는 것입니다. 일괄 처리가 아닌 상황에서는 각 Kinesis Data Streams 레코드를 개별 Kinesis Data Streams 레코드에 배치하고 HTTP 요청 하나를 만들어 Kinesis Data Streams로 전송합니다. 일괄 처리를 통해 각 HTTP 요청이 레코드 하나가 아니라 여러 개를 전달할 수 있습니다.

KPL에서 지원하는 배치 처리 유형은 다음 두 가지입니다.

- 집계 – 단일 Kinesis Data Streams 레코드에 여러 레코드를 저장합니다.
- 수집 - API 작업 PutRecords를 사용하여 Kinesis 데이터 스트림에 있는 하나 이상의 샤드로 여러 Kinesis Data Streams 레코드를 전송합니다.

두 유형의 KPL 배치 처리는 동시에 존재하도록 설계되었으며 서로 독립적으로 활성화하거나 비활성화할 수 있습니다. 기본적으로 둘 다 활성화됩니다.

## 집계

집계는 Kinesis Data Streams 레코드에 있는 여러 레코드의 스토리지를 의미합니다. 집계를 통해 고객이 API 호출당 전송되는 레코드 수를 늘릴 수 있으므로 생산자 처리량이 효과적으로 증가합니다.

Kinesis Data Streams 샤드는 최대 1,000개의 초당 Kinesis Data Streams 레코드 또는 1MB의 처리량을 지원합니다. 초당 Kinesis Data Streams 레코드 제한은 1KB 미만의 레코드와 고객을 바인딩합니다. 레코드 집계를 통해 고객이 여러 레코드를 Kinesis Data Streams 레코드 하나로 결합할 수 있어 샤드당 처리량을 향상시킬 수 있습니다.

리전 us-east-1의 샤드 하나가 현재 초당 레코드 1,000개의 일정한 속도로 실행되며 각 레코드가 512 바이트라고 가정해 보겠습니다. KPL 집계를 사용하면 레코드 1,000개를 Kinesis Data Streams 레코드 10개로 압축하여 RPS를 10(각각 50KB로)으로 줄일 수 있습니다.

## 수집

수집은 자체 HTTP 요청에 각각의 Kinesis Data Streams 레코드를 전송하는 대신 여러 Kinesis Data Streams 레코드를 배치 처리하고 API 작업 PutRecords를 직접적으로 호출하여 단일 HTTP 요청으로 전송하는 것을 의미합니다.

그러면 여러 HTTP 요청을 따로 만드는 오버헤드가 줄어들기 때문에 수집을 사용하지 않는 것보다 처리량이 늘어납니다. 사실 PutRecords 자체는 이 목적을 위해 특별히 고안되었습니다.

수집은 Kinesis Data Streams 레코드 그룹과 함께 사용된다는 점에서 집계와 다릅니다. 수집 중인 Kinesis Data Streams 레코드에 사용자로부터 받은 여러 레코드가 들어 있을 수 있습니다. 다음과 같이 관계를 시각화할 수 있습니다.

```
record 0 --|
record 1   |           [ Aggregation ]
...      |---> Amazon Kinesis record 0 --|
...      |
record A --|
...      |
...      |
record K --|
record L   |           [ Collection ]
...      |---> Amazon Kinesis record C --|---> PutRecords Request
...      |
```





생산자 측 언어	소비자 측 언어	KCL 버전	체크포인트 로직	KPL 사용 가능성 여부	경고
Java	Java	Java SDK 직접 사용	N/A	예	집계가 사용되는 경우 GetRecords 호출 후에 제공된 분해 라이브러리를 사용해야 합니다.
Java	자바 외에 모든 언어	SDK 직접 사용	N/A	예	집계를 비활성화 해야 함
Java	Java	1.3.x	N/A	예	집계를 비활성화 해야 함
Java	Java	1.4.x	인수 없이 체크포인트 호출	예	None
Java	Java	1.4.x	명시적 시퀀스 번호로 체크포인트 호출	예	집계를 비활성화하거나 확장된 시퀀스 번호를 검사에 사용하도록 코드 변경.
Java	자바 외에 모든 언어	1.3.x + 다국어 데몬 + 언어별 래퍼	N/A	예	집계를 비활성화 해야 함

## KPL을 사용하여 Kinesis 데이터 스트림에 쓰기

다음 단원에서는 가장 단순하고 가능한 "베어본" 생산자부터 완전 비동기식 코드까지 진행되는 샘플 코드를 보여줍니다.

### 베어본 생산자 코드

다음 코드만 있으면 작동되는 최소한의 생산자를 쓸 수 있습니다. Kinesis Producer Library(KPL) 사용자 레코드는 백그라운드에서 처리됩니다.

```
// KinesisProducer gets credentials automatically like
// DefaultAWSCredentialsProviderChain.
// It also gets region automatically from the EC2 metadata service.
KinesisProducer kinesis = new KinesisProducer();
// Put some records
for (int i = 0; i < 100; ++i) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));
    // doesn't block
    kinesis.addUserRecord("myStream", "myPartitionKey", data);
}
// Do other stuff ...
```

### 결과에 동기적으로 응답

이전 예제에서 코드는 KPL 사용자 레코드가 성공했는지 여부를 확인하지 않습니다. KPL은 실패를 설명하는 데 필요한 재시도를 수행합니다. 하지만 결과를 확인하려면 다음 예제(컨텍스트에 맞게 표시된 이전 예제)와 같이 addUserRecord에서 반환되는Future 객체를 사용하여 검사할 수 있습니다.

```
KinesisProducer kinesis = new KinesisProducer();

// Put some records and save the Futures
List<Future<UserRecordResult>> putFutures = new
    LinkedList<Future<UserRecordResult>>();
for (int i = 0; i < 100; i++) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));
    // doesn't block
    putFutures.add(
        kinesis.addUserRecord("myStream", "myPartitionKey", data));
}

// Wait for puts to finish and check the results
```

```

for (Future<UserRecordResult> f : putFutures) {
    UserRecordResult result = f.get(); // this does block
    if (result.isSuccessful()) {
        System.out.println("Put record into shard " +
            result.getShardId());
    } else {
        for (Attempt attempt : result.getAttempts()) {
            // Analyze and respond to the failure
        }
    }
}
}

```

## 결과에 비동기적으로 응답

이전 예제에서는 `get()` 객체에서 `Future`를 호출하므로 실행이 차단됩니다. 실행을 차단하지 않으려면 다음 예제에 나온 대로 비동기 콜백을 사용할 수 있습니다.

```

KinesisProducer kinesis = new KinesisProducer();

FutureCallback<UserRecordResult> myCallback = new FutureCallback<UserRecordResult>() {

    @Override public void onFailure(Throwable t) {
        /* Analyze and respond to the failure */
    };

    @Override public void onSuccess(UserRecordResult result) {
        /* Respond to the success */
    };
};

for (int i = 0; i < 100; ++i) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));
    ListenableFuture<UserRecordResult> f = kinesis.addUserRecord("myStream",
        "myPartitionKey", data);
    // If the Future is complete by the time we call addCallback, the callback will be
    invoked immediately.
    Futures.addCallback(f, myCallback);
}

```

## Kinesis Producer Library 구성

대부분의 사용 사례에서는 기본 설정으로 충분하지만 `KinesisProducer`의 동작을 필요에 맞게 조정할 수 있도록 기본 설정을 변경할 수 있습니다. 그러려면 다음과 같이

KinesisProducerConfiguration 클래스의 인스턴스를 KinesisProducer 생성자로 전달할 수 있습니다.

```
KinesisProducerConfiguration config = new KinesisProducerConfiguration()
    .setRecordMaxBufferedTime(3000)
    .setMaxConnections(1)
    .setRequestTimeout(60000)
    .setRegion("us-west-1");

final KinesisProducer kinesisProducer = new KinesisProducer(config);
```

속성 파일에서 구성을 로드할 수도 있습니다.

```
KinesisProducerConfiguration config =
    KinesisProducerConfiguration.fromPropertiesFile("default_config.properties");
```

사용자 프로세스에서 액세스할 수 있는 경로와 파일 이름을 대체할 수 있습니다. 이렇게 생성된 KinesisProducerConfiguration 인스턴스에서 설정 메서드를 추가로 호출하여 구성을 사용자 지정할 수 있습니다.

속성 파일은 에서 해당 이름을 사용하여 파라미터를 지정해야 합니다. PascalCase 이름은 KinesisProducerConfiguration 클래스의 설정 메서드에 사용된 이름과 일치해야 합니다. 예:

```
RecordMaxBufferedTime = 100
MaxConnections = 4
RequestTimeout = 6000
Region = us-west-1
```

구성 매개변수 사용 규칙 및 값 제한에 대한 자세한 내용은 의 [샘플 구성 속성 파일을](#) 참조하십시오 GitHub.

KinesisProducer가 초기화된 후에는 사용한 KinesisProducerConfiguration 인스턴스를 변경해도 더 이상 아무 효과가 없습니다. KinesisProducer는 현재 동적 재구성을 지원하지 않습니다.

## 소비자 분해

KCL은 릴리스 1.4.0부터 KPL 사용자 레코드 자동 분해를 지원합니다. 이전 버전의 KCL로 쓴 소비자 애플리케이션 코드는 KCL 업데이트 후에 수정하지 않고 컴파일됩니다. 그러나 생산자 측에서 KPL 집계를 사용하는 경우에는 체크포인트 수행을 포함한 세부적인 사항이 있습니다. 집계된 레코드 내의 모

든 하위 레코드는 시퀀스 번호가 동일하므로 하위 레코드를 구별해야 하는 경우 추가 데이터를 체크포인트와 함께 저장해야 합니다. 이 추가 데이터를 하위 시퀀스 번호라고 합니다.

## 이전 버전의 KPL에서 마이그레이션

기존 호출을 변경하지 않고도 집계와 함께 검사를 수행할 수 있습니다. Kinesis Data Streams에 성공적으로 저장된 모든 레코드를 여전히 검색할 수 있습니다. 이제 KCL은 두 가지 체크포인트 작업을 새로 제공하여 아래에 설명된 특정 사용 사례를 지원합니다.

KPL 지원 전에 KCL용으로 기존 코드가 쓰여지고 인수를 사용하지 않고 체크포인트 작업이 직접적으로 호출되면 배치에 있는 마지막 KPL 사용자 레코드의 시퀀스 번호 체크포인트를 수행하는 것과 같습니다. 체크포인트 작업이 시퀀스 번호 문자열로 호출되면 암시적 시퀀스 번호 0(영)과 함께 배치의 제공된 시퀀스 번호를 검사하는 것과 같습니다.

인수를 사용하지 않고 새로운 KCL 체크포인트 작업 `checkpoint()`를 직접적으로 호출하는 것은 암시적 시퀀스 번호 0(영)과 함께 배치에서 마지막 Record 호출의 시퀀스 번호 체크포인트를 수행하는 것과 같은 의미입니다.

새로운 KCL 체크포인트 작업 `checkpoint(Record record)`를 직접적으로 호출하는 것은 암시적 시퀀스 번호 0(영)과 함께 제공된 Record의 시퀀스 번호 체크포인트를 수행하는 것과 같은 의미입니다. Record 호출이 실제로 UserRecord인 경우 UserRecord 시퀀스 번호 및 하위 시퀀스 번호가 검사됩니다.

새로운 KCL 체크포인트 작업 `checkpoint(String sequenceNumber, long subSequenceNumber)`를 직접적으로 호출하면 주어진 하위 시퀀스 번호와 함께 제공된 시퀀스 번호 체크포인트가 명시적으로 수행됩니다.

이 경우 체크포인트가 Amazon DynamoDB 체크포인트 테이블에 저장되면 애플리케이션이 충돌하고 다시 시작될 때 KCL이 레코드 검색을 올바르게 다시 시작할 수 있습니다. 시퀀스에 레코드가 더 있으면 가장 최근에 검사된 시퀀스 번호가 있는 레코드에서 다음 하위 시퀀스 번호 레코드부터 검색이 시작됩니다. 가장 최근 체크포인트에 이전 시퀀스 번호 레코드의 마지막 하위 시퀀스 번호가 포함된 경우 다음 시퀀스 번호를 가진 레코드부터 검색이 시작됩니다.

다음 단원에서는 레코드 건너뛰기와 중복을 방지해야 하는 소비자의 시퀀스 및 하위 시퀀스 검사를 자세히 설명합니다. 소비자의 레코드 처리를 중단하고 다시 시작할 때 레코드 건너뛰기나 중복이 문제가 되지 않는다면 수정하지 않고 기존의 코드를 실행해도 좋습니다.

## KPL 분해를 위한 KCL 확장

앞에서 설명한 대로 KPL 분해에 하위 시퀀스 체크포인트 수행을 포함할 수 있습니다. 하위 시퀀스 체크포인트 수행을 쉽게 사용할 수 있도록 UserRecord 클래스가 KCL에 추가되었습니다.

```
public class UserRecord extends Record {
    public long getSubSequenceNumber() {
        /* ... */
    }
    @Override
    public int hashCode() {
        /* contract-satisfying implementation */
    }
    @Override
    public boolean equals(Object obj) {
        /* contract-satisfying implementation */
    }
}
```

이제 Record 대신 이 클래스가 사용됩니다. 이 클래스는 Record의 하위 클래스이므로 기존 코드가 손상되지 않습니다. UserRecord 클래스는 실제 하위 레코드와 집계되지 않은 표준 레코드를 나타냅니다. 집계되지 않은 레코드는 하위 레코드가 하나뿐인 집계된 레코드라고 간주할 수 있습니다.

또한 IRecordProcessorCheckpoint에 두 가지 새로운 작업이 추가되었습니다.

```
public void checkpoint(Record record);
public void checkpoint(String sequenceNumber, long subSequenceNumber);
```

하위 시퀀스 번호 검사를 사용하기 위해 다음 변환을 수행할 수 있습니다. 다음 양식 코드를 변경합니다.

```
checkpointer.checkpoint(record.getSequenceNumber());
```

새 양식 코드:

```
checkpointer.checkpoint(record);
```

하위 시퀀스 검사에 checkpoint(Record record) 양식을 사용하는 것이 좋습니다. 그러나 검사에 사용할 sequenceNumbers를 문자열에 이미 저장한 경우 다음 예제와 같이 subSequenceNumber도 저장해야 합니다.

```
String sequenceNumber = record.getSequenceNumber();
long subSequenceNumber = ((UserRecord) record).getSubSequenceNumber(); // ... do other processing
```

```
checkpointer.checkpoint(sequenceNumber, subSequenceNumber);
```

항상 Record가 기본적으로 구현에 사용되므로 UserRecord에서 UserRecord로 향하는 캐스트가 언제나 성공합니다. 시퀀스 번호에 산술 연산을 수행할 필요가 없으면 이 방법을 사용하지 않는 것이 좋습니다.

KPL 사용자 레코드를 처리하는 동안 KCL은 하위 시퀀스 번호를 각 행의 추가 필드로 Amazon DynamoDB에 씁니다. 이전 버전의 KCL에서는 체크포인트를 다시 시작할 때 레코드를 가져오기 위해 AFTER\_SEQUENCE\_NUMBER를 사용하지만 KPL이 지원되는 현재 KCL에서는 AT\_SEQUENCE\_NUMBER를 대신 사용합니다. 검사된 시퀀스 번호의 레코드가 검색되면 검사된 하위 시퀀스 번호가 확인되고 하위 레코드가 적절히 배치됩니다(마지막 하위 레코드가 검사된 레코드일 경우 모두 해당). 다시 말해 집계되지 않은 레코드는 단일 하위 레코드가 있는 집계 레코드로 간주할 수 있으므로 집계된 레코드와 집계되지 않은 레코드 모두 동일한 알고리즘이 적용됩니다.

## GetRecords 직접 사용

KCL을 사용하지 않고 API 작업 GetRecords를 간접적으로 호출하여 Kinesis Data Streams 레코드를 검색할 수도 있습니다. 원래의 KPL 사용자 레코드에 검색된 이 레코드의 압축을 풀려면 UserRecord.java에서 다음 정적 작업 중 하나를 직접적으로 호출합니다.

```
public static List<Record> deaggregate(List<Record> records)

public static List<UserRecord> deaggregate(List<UserRecord> records, BigInteger
    startingHashKey, BigInteger endingHashKey)
```

첫 번째 작업에서는 0에 기본값 startingHashKey(영)을 사용하고  $2^{128} - 1$ 에 기본값 endingHashKey을 사용합니다.

이 작업은 각각 지정된 Kinesis Data Streams 레코드 목록을 KPL 사용자 레코드 목록으로 분해합니다. 명시적 해시 키 또는 파티션 키가 startingHashKey(포함) 및 endingHashKey(포함) 범위를 벗어나는 모든 KPL 사용자 레코드는 반환된 레코드 목록에서 삭제됩니다.

## Firehose와 함께 KPL 사용하기

Kinesis Producer Library(KPL)를 사용하여 Kinesis 데이터 스트림에 데이터를 쓰는 경우, 집계를 사용하여 해당 Kinesis 데이터 스트림에 쓰는 레코드를 결합할 수 있습니다. 그런 다음 해당 데이터 스트림을 Firehose 전송 스트림의 소스로 사용하는 경우 Firehose는 대상으로 전송하기 전에 레코드를 집계해제합니다. 데이터를 변환하도록 전송 스트림을 구성하는 경우 Firehose는 레코드를 전송하기



전에 해당 레코드를 집계해제합니다. AWS Lambda 자세한 내용은 [Kinesis Data Streams를 사용하여 Amazon Firehose에 쓰기](#)를 참조하세요.

## AWS Glue 스키마 레지스트리와 함께 KPL 사용

Kinesis 데이터 스트림을 AWS Glue 스키마 레지스트리와 통합할 수 있습니다. AWS Glue 스키마 레지스트리를 사용하면 스키마를 중앙에서 검색, 제어 및 발전시키는 동시에 생성된 데이터가 등록된 스키마에 의해 지속적으로 검증되도록 할 수 있습니다. 스키마는 데이터 레코드의 구조와 포맷을 정의합니다. 스키마는 신뢰할 수 있는 데이터 게시, 소비 또는 저장을 위한 버전 지정 사양입니다. AWS Glue Schema Registry를 사용하면 스트리밍 애플리케이션 내에서 end-to-end 데이터 품질 및 데이터 거버넌스를 개선할 수 있습니다. 자세한 내용은 [AWS Glue Schema Registry](#)를 참조하세요. 이 통합을 설정하는 방법 중 하나는 Java에서 KPL 및 Kinesis Client Library(KCL) 라이브러리를 사용하는 것입니다.

### Important

현재 Kinesis Data Streams 및 AWS Glue 스키마 레지스트리 통합은 Java로 구현된 KPL 프로듀서를 사용하는 Kinesis 데이터 스트림에서만 지원됩니다. 다국어 지원은 제공되지 않습니다.

KPL을 사용하여 Kinesis Data Streams와 스키마 레지스트리의 통합을 설정하는 방법에 대한 자세한 지침은 사용 사례: [Amazon Kinesis 데이터 스트림과 Glue 스키마 레지스트리 통합의 “KPL/KCL 라이브러리를 사용한 데이터 상호 작용”](#) 섹션을 참조하십시오. AWS

## KPL 프록시 구성

인터넷에 직접 연결할 수 없는 애플리케이션의 경우 모든 AWS SDK 클라이언트가 HTTP 또는 HTTPS 프록시 사용을 지원합니다. 일반적인 엔터프라이즈 환경에서는 모든 아웃바운드 네트워크 트래픽이 프록시 서버를 거쳐야 합니다. 애플리케이션이 Kinesis Producer Library (KPL) 를 사용하여 프록시 서버를 사용하는 환경에서 데이터를 수집하고 전송하는 경우 애플리케이션에는 KPL 프록시 구성이 필요합니다. AWS KPL은 AWS Kinesis SDK를 기반으로 구축된 고급 라이브러리입니다. 네이티브 프로세스와 래퍼로 나뉩니다. 네이티브 프로세스는 레코드를 처리하고 전송하는 모든 작업을 수행하는 반면 래퍼는 네이티브 프로세스를 관리하고 래퍼와 통신합니다. 자세한 내용은 [Implementing Efficient and Reliable Producers with the Amazon Kinesis Producer Library](#)를 참조하세요.

래퍼는 Java로 작성되고 네이티브 프로세스는 Kinesis SDK를 사용하여 C++로 작성됩니다. KPL 버전 0.14.7 이상은 이제 모든 프록시 구성을 네이티브 프로세스로 전달할 수 있는 Java 래퍼의 프록시 구성을 지원합니다. [자세한 내용은 https://github.com/aws-labs/ /releases/tag/v0.14.7](https://github.com/aws-labs/ /releases/tag/v0.14.7)을 참조하십시오. [amazon-kinesis-producer](#)

다음 코드를 사용하여 KPL 애플리케이션에 프록시 구성을 추가할 수 있습니다.

```
KinesisProducerConfiguration configuration = new KinesisProducerConfiguration();
// Next 4 lines used to configure proxy
configuration.setProxyHost("10.0.0.0"); // required
configuration.setProxyPort(3128); // default port is set to 443
configuration.setProxyUserName("username"); // no default
configuration.setProxyPassword("password"); // no default

KinesisProducer kinesisProducer = new KinesisProducer(configuration);
```

## AWS SDK for Java와 Amazon Kinesis Data Streams API를 사용하여 생산자 개발

AWS SDK for Java와 함께 Amazon Kinesis Data Streams API를 사용하여 생산자를 개발할 수 있습니다. Kinesis Data Streams를 처음 사용하는 경우, 먼저 [Amazon Kinesis Data Streams란?](#) 및 [Amazon Kinesis Data Streams 시작하기](#)에 있는 개념과 용어를 알아 두세요.

이 예제에서는 [Kinesis Data Streams API](#)를 설명하고 [AWS SDK for Java](#)를 사용하여 스트림에 데이터를 추가(넣기)합니다. 그러나 대부분의 사용 사례에서는 Kinesis Data Streams KPL 라이브러리를 우선적으로 사용해야 합니다. 자세한 내용은 [Amazon Kinesis Producer Library를 사용하여 생산자 개발](#) 섹션을 참조하세요.

이 장의 Java 예제 코드는 기본 Kinesis Data Streams API 작업을 수행하는 방법을 설명하며, 작업 유형에 따라 논리적으로 나뉘어집니다. 이 예제는 가능한 모든 예외를 확인하지 않거나 가능한 모든 보안 및 성능 고려 사항을 감안하지 않는다는 점에서 프로덕션 지원 코드가 아닙니다. 또한 다른 프로그래밍 언어를 사용하여 [Kinesis Data Streams API](#)를 직접적으로 호출할 수 있습니다. 사용 가능한 모든 AWS SDK에 대한 자세한 내용은 [Amazon Web Services로 개발 시작](#)을 참조하세요.

각 작업에는 사전 요구 사항이 있습니다. 예를 들어, 스트림을 생성하기 전에는 데이터를 스트림에 추가할 수 없으므로 클라이언트를 만들어야 합니다. 자세한 내용은 [스트림 생성 및 관리](#) 섹션을 참조하세요.

### 주제

- [스트림에 데이터 추가](#)

- [AWS Glue 스키마 레지스트리를 사용하여 데이터와 상호 작용](#)

## 스트림에 데이터 추가

스트림이 생성되면 레코드의 형태로 데이터를 스트림에 추가할 수 있습니다. 레코드는 데이터 BLOB 형식으로 처리할 데이터가 포함된 데이터 구조입니다. 데이터를 레코드에 저장한 후에는 Kinesis Data Streams가 어떤 식으로든 데이터를 검사하거나 해석하거나 변경하지 않습니다. 또한 레코드마다 연결된 시퀀스 번호와 파티션 키가 있습니다.

Kinesis Data Streams API에는 스트림에 데이터를 추가하는 두 가지 작업, [PutRecords](#)와 [PutRecord](#)가 있습니다. PutRecords 작업은 HTTP 요청마다 스트림에 여러 레코드를 보내고, 단일 PutRecord 작업은 한 번에 하나씩 스트림에 레코드를 보냅니다(레코드마다 별도의 HTTP 요청 필요). 데이터 생산자마다 더 높은 처리량을 보관하므로 대부분의 애플리케이션에 PutRecords를 우선 사용해야 합니다. 각 작업에 대한 자세한 내용은 아래에 나와 있는 별도의 하위 단원을 참조하십시오.

### 주제

- [PutRecords를 사용하여 여러 레코드 추가](#)
- [PutRecord를 사용하여 단일 레코드 추가](#)

소스 애플리케이션이 Kinesis Data Streams API를 사용하여 스트림에 데이터를 추가할 때는 스트림의 데이터를 동시에 처리하는 소비자 애플리케이션이 하나 이상일 가능성이 높습니다. Kinesis Data Streams API를 사용하여 소비자가 데이터를 가져오는 방법에 대한 자세한 내용은 [스트림에서 데이터 가져오기](#) 섹션을 참조하세요.

### Important

[데이터 보존 기간 변경](#)

## PutRecords를 사용하여 여러 레코드 추가

[PutRecords](#) 작업은 단일 요청에서 여러 레코드를 Kinesis Data Streams에 보냅니다. 데이터를 Kinesis 데이터 스트림에 보낼 때 PutRecords를 사용하여 생산자가 더 높은 처리량을 보관할 수 있습니다. 각 PutRecords 요청은 최대 500개의 레코드를 지원할 수 있습니다. 요청에 포함되는 각 레코드 크기의 상한은 1MB이며, 파티션 키를 포함해 전체 요청당 최대 5MB로 제한됩니다. 아래에서 설명하는 단일 PutRecord 작업과 마찬가지로 PutRecords도 시퀀스 번호와 파티션 키를 사용합니다. 그러나

PutRecord 파라미터 SequenceNumberForOrdering이 PutRecords 호출에 포함되지 않습니다. PutRecords 작업은 요청의 일반 순서에 따라 모든 레코드를 처리하려고 합니다.

데이터 레코드마다 고유한 시퀀스 번호가 있습니다. 스트림에 데이터 레코드를 추가하기 위해 client.putRecords를 직접적으로 호출한 후 Kinesis Data Streams에서 시퀀스 번호를 할당합니다. 동일한 파티션 키에 대한 시퀀스 번호는 일반적으로 시간이 지남에 따라 증가합니다. PutRecords 요청 기간이 길어질수록 시퀀스 번호도 커집니다.

### Note

같은 스트림에 있는 데이터 세트의 인덱스로 시퀀스 번호를 사용할 수 없습니다. 데이터 세트를 논리적으로 분리하려면 파티션 키를 사용하거나 데이터 세트마다 별도의 스트림을 만드십시오.

PutRecords 요청은 다른 파티션 키를 가진 레코드를 포함할 수 있습니다. 요청의 범위는 스트림이며, 요청은 요청 제한 이내에서 파티션 키와 레코드 조합을 포함할 수 있습니다. 일반적으로 여러 다른 파티션 키로 만들어져 여러 다른 샤드를 가진 스트림으로 전송되는 요청은 소수의 파티션 키를 가지고 있고 소수의 샤드로 전송되는 요청보다 빠릅니다. 파티션 키의 수가 샤드 수보다 훨씬 커야 지연 시간을 줄이고 처리량을 최대화할 수 있습니다.

### PutRecords 예제

다음 코드는 파티션 키가 순차적인 데이터 레코드 100개를 만들어 DataStream이라는 스트림에 넣습니다.

```
AmazonKinesisClientBuilder clientBuilder =
AmazonKinesisClientBuilder.standard();

clientBuilder.setRegion(regionName);
clientBuilder.setCredentials(credentialsProvider);
clientBuilder.setClientConfiguration(config);

AmazonKinesis kinesisClient = clientBuilder.build();

PutRecordsRequest putRecordsRequest = new PutRecordsRequest();
putRecordsRequest.setStreamName(streamName);
List <PutRecordsRequestEntry> putRecordsRequestEntryList = new ArrayList<>();
for (int i = 0; i < 100; i++) {
    PutRecordsRequestEntry putRecordsRequestEntry = new
PutRecordsRequestEntry();
```

```

putRecordsRequestEntry.setData(ByteBuffer.wrap(String.valueOf(i).getBytes()));
    putRecordsRequestEntry.setPartitionKey(String.format("partitionKey-%d",
i));
    putRecordsRequestEntryList.add(putRecordsRequestEntry);
}

putRecordsRequest.setRecords(putRecordsRequestEntryList);
PutRecordsResult putRecordsResult =
kinesisClient.putRecords(putRecordsRequest);
System.out.println("Put Result" + putRecordsResult);

```

PutRecords 응답에는 응답 Records의 어레이가 포함됩니다. 응답 어레이의 각 레코드는 요청 및 응답의 일반 순서(위에서 아래로)를 이용해 요청 어레이의 레코드와 직접 연관됩니다. 응답 Records 어레이에는 항상 요청 어레이와 같은 수의 레코드가 포함됩니다.

### PutRecords 사용 시 실패 처리

기본적으로 요청에 있는 개별 레코드가 실패해도 PutRecords 요청에 있는 후속 레코드의 처리가 중단되지 않습니다. 즉, 응답 Records 어레이에 성공적으로 처리된 레코드와 그렇지 않은 레코드가 모두 포함됩니다. 따라서 성공적으로 처리되지 않은 레코드를 찾아 후속 호출에 포함해야 합니다.

성공한 레코드에는 SequenceNumber 및 ShardID 값이 포함되며 성공하지 못한 레코드에는 ErrorCode 및 ErrorMessage 값이 포함됩니다. ErrorCode 파라미터는 오류 유형을 반영하며 ProvisionedThroughputExceededException 값 또는 InternalFailure 값 중 하나 일 수 있습니다. ErrorMessage는 병목 현상이 발생한 레코드의 계정 ID, 스트림 이름 및 샤드 ID를 포함하여 ProvisionedThroughputExceededException 예외에 대한 세부 정보를 제공합니다. 아래 예제에서는 PutRecords 요청에 레코드 3개가 있습니다. 두 번째 레코드가 실패하고 응답에 반영됩니다.

### Example PutRecords 요청 구문

```

{
  "Records": [
    {
      "Data": "XzxcYXRhPl8w",
      "PartitionKey": "partitionKey1"
    },
    {
      "Data": "AbceddeRFfg12asd",
      "PartitionKey": "partitionKey1"
    },
  ],

```

```

    {
      "Data": "KFpcd98*7nd1",
      "PartitionKey": "partitionKey3"
    }
  ],
  "StreamName": "myStream"
}

```

## Example PutRecords 응답 구문

```

{
  "FailedRecordCount": 1,
  "Records": [
    {
      "SequenceNumber": "21269319989900637946712965403778482371",
      "ShardId": "shardId-000000000001"

    },
    {
      "ErrorCode": "ProvisionedThroughputExceededException",
      "ErrorMessage": "Rate exceeded for shard shardId-000000000001 in stream
exampleStreamName under account 111111111111."

    },
    {
      "SequenceNumber": "21269319989999637946712965403778482985",
      "ShardId": "shardId-000000000002"
    }
  ]
}

```

성공적으로 처리되지 않은 레코드를 후속 PutRecords 요청에 포함할 수 있습니다. 먼저, putRecordsResult에서 FailedRecordCount 파라미터를 확인해서 요청에 처리에 실패한 레코드가 있는지 확인합니다. 그런 레코드가 있으면 putRecordsEntry이 아닌 ErrorCode가 있는 각 null를 후속 요청에 추가해야 합니다. 이러한 유형의 핸들러 예제는 다음 코드를 참조하세요.

## Example PutRecords 실패 핸들러

```

PutRecordsRequest putRecordsRequest = new PutRecordsRequest();
putRecordsRequest.setStreamName(myStreamName);
List<PutRecordsRequestEntry> putRecordsRequestEntryList = new ArrayList<>();
for (int j = 0; j < 100; j++) {

```

```

PutRecordsRequestEntry putRecordsRequestEntry = new PutRecordsRequestEntry();
putRecordsRequestEntry.setData(ByteBuffer.wrap(String.valueOf(j).getBytes()));
putRecordsRequestEntry.setPartitionKey(String.format("partitionKey-%d", j));
putRecordsRequestEntryList.add(putRecordsRequestEntry);
}

putRecordsRequest.setRecords(putRecordsRequestEntryList);
PutRecordsResult putRecordsResult = amazonKinesisClient.putRecords(putRecordsRequest);

while (putRecordsResult.getFailedRecordCount() > 0) {
    final List<PutRecordsRequestEntry> failedRecordsList = new ArrayList<>();
    final List<PutRecordsResultEntry> putRecordsResultEntryList =
putRecordsResult.getRecords();
    for (int i = 0; i < putRecordsResultEntryList.size(); i++) {
        final PutRecordsRequestEntry putRecordRequestEntry =
putRecordsRequestEntryList.get(i);
        final PutRecordsResultEntry putRecordsResultEntry =
putRecordsResultEntryList.get(i);
        if (putRecordsResultEntry.getErrorCode() != null) {
            failedRecordsList.add(putRecordRequestEntry);
        }
    }
    putRecordsRequestEntryList = failedRecordsList;
    putRecordsRequest.setRecords(putRecordsRequestEntryList);
    putRecordsResult = amazonKinesisClient.putRecords(putRecordsRequest);
}

```

## PutRecord를 사용하여 단일 레코드 추가

각 [PutRecord](#) 호출은 레코드 1개에 적용됩니다. 특별히 애플리케이션이 항상 요청당 레코드를 1개만 보내야 하거나 다른 어떤 이유로 PutRecords를 사용할 수 없는 경우가 아니라면 [PutRecords를 사용하여 여러 레코드 추가](#)에서 설명하는 PutRecords 작업을 우선 사용하십시오.

데이터 레코드마다 고유한 시퀀스 번호가 있습니다. 스트림에 데이터 레코드를 추가하기 위해 `client.putRecord`를 직접적으로 호출한 후 Kinesis Data Streams에서 시퀀스 번호를 할당합니다. 동일한 파티션 키에 대한 시퀀스 번호는 일반적으로 시간이 지남에 따라 증가합니다. PutRecord 요청 기간이 길어질수록 시퀀스 번호도 커집니다.

널기 작업은 본질적으로 Kinesis Data Streams와 동시에 나타나기 때문에 널기 작업이 연달아 빠르게 발생할 때는 반환된 시퀀스 번호가 증가한다는 보장은 없습니다. 같은 파티션 키의 시퀀스 번호가 증가하도록 확실하게 보장하려면 `SequenceNumberForOrdering` 코드 샘플에 나온 대로 [PutRecord 예제](#) 파라미터를 사용하십시오.

SequenceNumberForOrdering 사용 여부와 관계없이 Kinesis Data Streams가 GetRecords 호출을 통해 검색하는 레코드는 시퀀스 번호대로 엄격하게 지정됩니다.

### Note

같은 스트림에 있는 데이터 세트의 인덱스로 시퀀스 번호를 사용할 수 없습니다. 데이터 세트를 논리적으로 분리하려면 파티션 키를 사용하거나 데이터 세트마다 별도의 스트림을 만드십시오.

파티션 키는 스트림에서 데이터를 그룹화하는 데 사용됩니다. 해당 파티션 키에 따라 스트림 내의 샤드에 데이터 레코드가 할당됩니다. 특히 Kinesis Data Streams는 파티션 키 및 연결된 데이터를 특정 샤드에 매핑하는 해시 함수에 대한 입력으로 파티션 키를 사용합니다.

이 해시 메커니즘의 결과로 같은 파티션 키를 가진 모든 데이터 레코드가 스트림에 있는 동일한 샤드에 매핑됩니다. 그러나 파티션 키의 수가 샤드 수를 초과하면 일부 샤드에 서로 다른 파티션 키를 가진 레코드가 반드시 포함됩니다. 설계 면에서 모든 샤드를 잘 활용하려면 setShardCount의 CreateStreamRequest 메서드로 지정된 샤드 수가 고유한 파티션 수보다 상당히 적고, 단일 파티션 키로 전송되는 데이터의 양이 샤드의 용량보다 상당히 적어야 합니다.

### PutRecord 예제

다음 코드는 파티션 키 2개에 배포되는 데이터 레코드 10개를 만들어 myStreamName이라는 스트림에 넣습니다.

```
for (int j = 0; j < 10; j++)
{
    PutRecordRequest putRecordRequest = new PutRecordRequest();
    putRecordRequest.setStreamName( myStreamName );
    putRecordRequest.setData(ByteBuffer.wrap( String.format( "testData-%d",
j ).getBytes() ));
    putRecordRequest.setPartitionKey( String.format( "partitionKey-%d", j/5 ));
    putRecordRequest.setSequenceNumberForOrdering( sequenceNumberOfPreviousRecord );
    PutRecordResult putRecordResult = client.putRecord( putRecordRequest );
    sequenceNumberOfPreviousRecord = putRecordResult.getSequenceNumber();
}
```

앞에 나온 코드 샘플에서는 setSequenceNumberForOrdering을 사용하여 각 파티션 키 내에서 순서가 증가하도록 확실하게 보장합니다. 이 파라미터를 효과적으로 사용하려면 현재 레코드(레코드 n)의 SequenceNumberForOrdering을 이전 레코드(레코드 n-1)의 시퀀스 번호로 설정합니다. 스트림



에 추가된 레코드의 시퀀스 번호를 가져오려면 `putRecord`의 결과에 `getSequenceNumber`를 호출하십시오.

`SequenceNumberForOrdering` 파라미터는 동일한 파티션 키에 대한 시퀀스 번호가 확실하게 증가하도록 보장합니다. `SequenceNumberForOrdering`은 여러 파티션 키에 대한 레코드의 순서를 지정하지 않습니다.

## AWS Glue 스키마 레지스트리를 사용하여 데이터와 상호 작용

Kinesis 데이터 스트림을 AWS Glue 스키마 레지스트리와 통합할 수 있습니다. AWS Glue 스키마 레지스트리를 사용하면 스키마를 중앙에서 검색, 제어 및 발전시키는 동시에 생성된 데이터가 등록된 스키마에 의해 지속적으로 검증되도록 할 수 있습니다. 스키마는 데이터 레코드의 구조와 포맷을 정의합니다. 스키마는 신뢰할 수 있는 데이터 게시, 소비 또는 저장을 위한 버전 지정 사양입니다. AWS Glue 스키마 레지스트리를 사용하면 스트리밍 애플리케이션 내에서 엔드 투 엔드 데이터 품질 및 데이터 거버넌스를 개선할 수 있습니다. 자세한 내용은 [AWS Glue Schema Registry](#)를 참조하세요. 이 통합을 설정하는 방법 중 하나는 AWS Java SDK에서 사용 가능한 `PutRecords` 및 `PutRecord` Kinesis Data Streams API를 사용하는 것입니다.

Kinesis Data Streams API를 사용하여 Kinesis Data Streams와 스키마 레지스트리의 통합을 설정하는 방법에 대한 자세한 지침은 [사용 사례: AWS Glue Schema Registry와 Amazon Kinesis Data Streams 통합](#)의 'Kinesis Data Streams API를 사용하여 데이터와 상호 작용'을 참조하세요.

## Kinesis 에이전트를 사용하여 Amazon Kinesis Data Streams에 쓰기

Kinesis 에이전트는 간편하게 데이터를 수집하여 Kinesis Data Streams로 전송할 수 있는 독립형 Java 소프트웨어 애플리케이션입니다. 에이전트가 파일 집합을 지속적으로 모니터링하고 새로운 데이터를 스트림에 보냅니다. 에이전트는 파일 로테이션, 검사를 처리하고 실패할 경우 다시 시도하며 적시에 안정적이고 단순한 방식으로 모든 데이터를 전달할 뿐 아니라 또한 스트리밍 프로세스를 더 잘 모니터링하고 문제를 해결하는 데 도움이 되는 Amazon CloudWatch 메트릭을 생성합니다.

기본적으로 줄 바꿈 문자('\n')를 기반으로 각 파일에서 레코드가 구문 분석됩니다. 그러나 여러 줄 레코드를 구문 분석하도록 에이전트를 구성할 수도 있습니다([에이전트 구성 설정](#) 참조).

웹 서버, 로그 서버, 데이터베이스 서버 등 Linux 기반 서버 환경에 에이전트를 설치할 수 있습니다. 에이전트를 설치한 후 모니터링할 파일과 데이터의 스트림을 지정하여 에이전트를 구성하십시오. 에이전트가 구성되면 파일에서 일관되게 데이터를 수집하고, 안정적으로 스트림에 전송합니다.

주제

- [필수 조건](#)

- [에이전트 다운로드 및 설치](#)
- [에이전트 구성 및 시작](#)
- [에이전트 구성 설정](#)
- [여러 파일 디렉터리 모니터링 및 여러 스트림에 쓰기](#)
- [에이전트를 사용하여 데이터 사전 처리](#)
- [에이전트 CLI 명령](#)
- [FAQ](#)

## 필수 조건

- 운영 체제는 Amazon Linux AMI 버전 2015.09 이상 또는 Red Hat Enterprise Linux 버전 7 이상이어야 합니다.
- Amazon EC2를 사용하여 에이전트를 실행하는 경우 EC2 인스턴스를 시작합니다.
- 다음 방법 중 하나를 사용하여 AWS 자격 증명을 관리하십시오.
  - EC2 인스턴스를 시작할 때 IAM 역할을 지정합니다.
  - 에이전트를 구성할 때 AWS 자격 증명을 지정하십시오 ([awsAccessKeyId](#) 및 [awsSecretAccess키](#) 참조).
  - 지역 및 AWS 액세스 키를 `/etc/sysconfig/aws-kinesis-agent` 지정하려면 편집하십시오.
  - [EC2 인스턴스가 다른 AWS 계정에 있는 경우 Kinesis Data Streams 서비스에 대한 액세스를 제공하는 IAM 역할을 생성하고 에이전트를 구성할 때 해당 역할을 지정하십시오 \(Assumerolearn 및 Id 참조\). `assumeRoleExternal` 이전 방법 중 하나를 사용하여 이 역할을 수입할 권한이 있는 다른 계정의 사용자 AWS 자격 증명을 지정하십시오.](#)
- 에이전트가 스트림으로 데이터를 전송하려면 지정한 IAM 역할 또는 AWS 자격 증명에 Kinesis Data [PutRecordsStreams](#) 작업을 수행할 권한이 있어야 합니다. 에이전트에 대한 CloudWatch 모니터링을 활성화하는 경우 작업을 수행할 권한도 필요합니다. CloudWatch [PutMetricData](#) 자세한 내용은 [IAM을 사용하여 Amazon Kinesis Data Streams 리소스에 대한 액세스 제어아마존을 통한 Kinesis Data Streams 에이전트 헬스 모니터링 CloudWatch](#), 및 [CloudWatch 액세스 제어를](#) 참조하십시오.

## 에이전트 다운로드 및 설치

우선 인스턴스에 연결합니다. 자세한 내용은 Linux용 Amazon EC2 사용 설명서에서 [인스턴스에 연결](#)을 참조하세요. 연결 문제가 있는 경우 Amazon EC2 - Linux 인스턴스용 사용 설명서의 [인스턴스 연결 문제 해결](#)을 참조하세요.

## Amazon Linux AMI를 이용해 에이전트를 설치하려면

다음 명령을 사용하여 에이전트를 다운로드하고 설치합니다.

```
sudo yum install -y aws-kinesis-agent
```

## Red Hat Enterprise Linux를 사용하여 에이전트를 설치하려면

다음 명령을 사용하여 에이전트를 다운로드하고 설치합니다.

```
sudo yum install -y https://s3.amazonaws.com/streaming-data-agent/aws-kinesis-agent-latest.amzn2.noarch.rpm
```

를 사용하여 에이전트를 설정하려면 GitHub

1. [amazon-kinesis-agentawslabs/에서](#) 에이전트를 다운로드하십시오.
2. 다운로드 디렉터리로 이동하고 다음 명령을 실행해 에이전트를 설치합니다.

```
sudo ./setup --install
```

## Docker 컨테이너에서 에이전트 설정

[amazonlinux](#) 컨테이너 베이스를 통해서도 컨테이너에서 Kinesis 에이전트를 실행할 수 있습니다. 다음 Dockerfile을 사용하여 `docker build`를 실행합니다.

```
FROM amazonlinux

RUN yum install -y aws-kinesis-agent which findutils
COPY agent.json /etc/aws-kinesis/agent.json

CMD ["start-aws-kinesis-agent"]
```

## 에이전트 구성 및 시작

에이전트를 구성하고 시작하려면

1. 구성 파일(`/etc/aws-kinesis/agent.json`)을 열고 편집합니다(기본 파일 액세스 권한을 사용하는 경우 슈퍼유저로).

이 구성 파일에서, 에이전트가 데이터를 수집하는 파일("filePattern") 및 에이전트가 데이터를 보내는 스트림의 이름("kinesisStream")을 지정합니다. 파일 이름은 패턴이며, 에이전트가 파일 로테이션을 인식합니다. 파일을 로테이션하거나 초당 1회 이하 새 파일을 생성할 수 있습니다. 에이전트가 파일 생성 타임스탬프를 사용하여 어떤 파일을 스트림까지 추적할지 결정합니다. 초당 1회보다 자주 새 파일을 생성하거나 파일을 로테이션하면 에이전트가 제대로 구별할 수 없습니다.

```
{
  "flows": [
    {
      "filePattern": "/tmp/app.log*",
      "kinesisStream": "yourkinesisstream"
    }
  ]
}
```

2. 에이전트를 수동으로 시작합니다.

```
sudo service aws-kinesis-agent start
```

3. (선택 사항) 시스템 시작 시 에이전트가 시작되도록 구성합니다.

```
sudo chkconfig aws-kinesis-agent on
```

현재 이 에이전트는 배경에서 시스템 서비스로 실행 중입니다. 지정된 파일을 지속적으로 모니터링하고 지정된 스트림으로 데이터를 보냅니다. 에이전트 활동은 /var/log/aws-kinesis-agent/aws-kinesis-agent.log에 기록됩니다.

## 에이전트 구성 설정

이 에이전트는 두 가지 필수 구성 설정인 filePattern 및 kinesisStream과 추가 기능을 제공하는 선택적 구성 설정을 지원합니다. /etc/aws-kinesis/agent.json에서 필수 및 선택적 구성을 지정할 수 있습니다.

구성 파일을 변경할 때마다 다음 명령을 이용해 에이전트를 중지했다 시작해야 합니다.

```
sudo service aws-kinesis-agent stop
sudo service aws-kinesis-agent start
```

또는 다음 명령을 사용할 수 있습니다.

```
sudo service aws-kinesis-agent restart
```

다음은 일반적인 구성 설정입니다.

구성 설정	설명
assumeRoleARN	사용자가 맡을 역할의 ARN입니다. 자세한 내용은 IAM 사용 설명서의 <a href="#">IAM 역할을 사용한 AWS 계정 간 액세스 위임을</a> 참조하십시오.
assumeRoleExternalId	역할을 맡을 사람을 결정하는 선택적 식별자입니다. 자세한 내용은 IAM 사용 설명서의 <a href="#">외부 ID 사용 방법</a> 을 참조하세요.
awsAccessKeyId	AWS 기본 자격 증명을 재정의하는 액세스 키 ID. 이 설정은 다른 모든 자격 증명 공급자보다 우선 적용됩니다.
awsSecretAccessKey	AWS 기본 자격 증명을 재정의하는 비밀 키. 이 설정은 다른 모든 자격 증명 공급자보다 우선 적용됩니다.
cloudwatch.emitMetrics	설정된 CloudWatch 경우 (true) 에이전트가 메트릭을 내보낼 수 있도록 합니다.  기본값: true
cloudwatch.endpoint	의 지역별 엔드포인트. CloudWatch  기본값: monitoring.us-east-1.amazonaws.com
kinesis.endpoint	Kinesis Data Streams에 대한 리전 엔드포인트.  기본값: kinesis.us-east-1.amazonaws.com

다음은 흐름 구성 설정입니다.

구성 설정	설명
dataProcessingOptions	스트림으로 전송되기 전에 구문 분석된 각 레코드에 적용되는 처리 옵션 목록입니다. 처리 옵션은 지정된 순서로 진행됩니다. 자세한 설명은 <a href="#">에이전트를 사용하여 데이터 사전 처리</a> 섹션을 참조하세요.
kinesisStream	[필수] 스트림 이름입니다.
filePattern	[필수] 에이전트가 픽업하기 위해 일치해야 하는 디렉터리 및 파일 패턴. 이 패턴과 일치하는 모든 파일에 대한 읽기 권한을 aws-kinesis-agent-user 에게 부여해야 합니다. 파일이 포함된 디렉터리에 대한 읽기 및 실행 권한을 aws-kinesis-agent-user 에게 부여해야 합니다.
initialPosition	파일 구문 분석이 처음 시작된 위치입니다. 유효 값은 START_OF_FILE 및 END_OF_FILE 입니다.  기본값: END_OF_FILE
maxBufferAgeMillis	스트림으로 보내기 전 에이전트가 데이터를 버퍼링하는 최대 시간(밀리초)입니다.  값 범위: 1,000 - 900,000(1초 - 15분)  기본값: 60,000(1분)
maxBufferSizeBytes	스트림으로 보내기 전 에이전트가 데이터를 버퍼링하는 최대 크기(바이트)입니다.  값 범위: 1 - 4,194,304(4MB)  기본값: 4,194,304(4MB)
maxBufferSizeRecords	스트림으로 보내기 전 에이전트가 데이터를 버퍼링하는 최대 레코드 수입니다.  값 범위: 1 - 500  기본값: 500

구성 설정	설명
minTimeBetweenFilePollsMillis	에이전트가 새로운 데이터에 대해 모니터링한 파일을 폴링하고 구문 분석하는 시간 간격(밀리초)입니다.  값 범위: 1 이상  기본값: 100
multiLineStartPattern	레코드의 시작을 식별하기 위한 패턴입니다. 레코드는 패턴과 일치하는 줄 1개 및 패턴과 일치하지 않는 나머지 줄로 이루어져 있습니다. 유효한 값은 정규식입니다. 기본적으로 로그 파일에서 각각의 줄 바꿈은 하나의 레코드로 구문 분석됩니다.
partitionKeyOption	파티션 키를 생성하기 위한 방법입니다. 유효한 값은 RANDOM(임의로 생성된 정수) 및 DETERMINISTIC (데이터에서 계산된 해시 값)입니다.  기본값: RANDOM
skipHeaderLines	모니터링한 파일을 시작할 때 에이전트가 구문 분석을 건너뛰는 줄의 개수입니다.  값 범위: 0 이상  기본값: 0(영)
truncatedRecord Terminator	레코드 크기가 Kinesis Data Streams 레코드 크기 제한을 초과할 때 에이전트가 구문 분석된 레코드를 자르는 데 사용하는 문자열입니다. (1,000KB)  기본값: '\n'(줄 바꿈)

## 여러 파일 디렉터리 모니터링 및 여러 스트림에 쓰기

여러 개의 흐름 구성 설정을 지정하여, 에이전트가 여러 파일 디렉터리를 모니터링하고 여러 스트림으로 데이터를 보내도록 구성할 수 있습니다. 다음 구성 예제에서 에이전트는 두 개의 파일 디렉터리를 모니터링하고 데이터를 각각 Kinesis 스트림과 Firehose 전송 스트림으로 보냅니다. Kinesis 스트림과 Firehose에 서로 다른 엔드포인트를 지정할 수 있으므로 Kinesis 스트림과 Firehose 전송 스트림이 같은 지역에 있지 않아도 됩니다.

```
{
  "cloudwatch.emitMetrics": true,
  "kinesis.endpoint": "https://your/kinesis/endpoint",
  "firehose.endpoint": "https://your/firehose/endpoint",
  "flows": [
    {
      "filePattern": "/tmp/app1.log*",
      "kinesisStream": "yourkinesisstream"
    },
    {
      "filePattern": "/tmp/app2.log*",
      "deliveryStream": "yourfirehosedeliverystream"
    }
  ]
}
```

Firehose에서 에이전트를 사용하는 방법에 대한 자세한 내용은 [Kinesis 에이전트를 사용하여 Amazon Data Firehose에 쓰기를](#) 참조하십시오.

## 에이전트를 사용하여 데이터 사전 처리

모니터링한 파일에서 구문 분석한 레코드를 스트림으로 보내기 전에 에이전트가 사전 처리할 수 있습니다. 파일 흐름에 `dataProcessingOptions` 구성 설정을 추가하여 이 기능을 활성화할 수 있습니다. 하나 이상의 처리 옵션을 추가할 수 있으며, 추가된 옵션은 지정된 순서로 수행됩니다.

에이전트는 다음에 나열된 처리 옵션을 지원합니다. 에이전트는 오픈 소스이므로, 처리 옵션을 더 개발하고 확장할 수 있습니다. 에이전트는 [Kinesis 에이전트](#)에서 다운로드할 수 있습니다.

### 처리 옵션

#### SINGLELINE

줄 바꿈 문자, 선행 공백과 후행 공백을 삭제해 여러 줄 레코드를 한 줄 레코드로 변환합니다.

```
{
  "optionName": "SINGLELINE"
}
```

#### CSVTOJSON

구분 기호로 구분된 형식에서 JSON 형식으로 레코드를 변환합니다.



```
{
  "optionName": "CSVTOJSON",
  "customFieldNames": [ "field1", "field2", ... ],
  "delimiter": "yourdelimiter"
}
```

### customFieldNames

[필수] 각각의 JSON 키 값 쌍에서 키로 사용되는 필드 이름입니다. 예를 들어, ["f1", "f2"]를 지정하면 레코드 "v1,v2"가 {"f1":"v1","f2":"v2"}로 변환됩니다.

### delimiter

레코드에서 구분 기호로 사용되는 문자열입니다. 기본값은 쉼표(,)입니다.

## LOGTOJSON

로그 형식에서 JSON 형식으로 레코드를 변환합니다. 지원되는 로그 형식은 Apache Common Log, Apache Combined Log, Apache Error Log 및 RFC3164 Syslog입니다.

```
{
  "optionName": "LOGTOJSON",
  "logFormat": "logformat",
  "matchPattern": "yourregexpattern",
  "customFieldNames": [ "field1", "field2", ... ]
}
```

### logFormat

[필수] 로그 항목 형식입니다. 유효한 값은 다음과 같습니다.

- COMMONAPACHELOG - Apache Common Log 형식입니다. 각 로그 항목에는 기본적으로 "%{host} %{ident} %{authuser} [%{datetime}] \" %{request}\" %{response} %{bytes}" 패턴이 있습니다.
- COMBINEDAPACHELOG - Apache Combined Log 형식입니다. 각 로그 항목에는 기본적으로 "%{host} %{ident} %{authuser} [%{datetime}] \" %{request}\" %{response} %{bytes} %{referrer} %{agent}" 패턴이 있습니다.
- APACHEERRORLOG - Apache Error Log 형식입니다. 각 로그 항목에는 기본적으로 "[%{timestamp}] [%{module}:%{severity}] [pid %{processid}:tid %{threadid}] [client: %{client}] %{message}" 패턴이 있습니다.
- SYSLOG - RFC3164 Syslog 형식입니다. 각 로그 항목에는 기본적으로 "%{timestamp} %{hostname} %{program}[%{processid}]: %{message}" 패턴이 있습니다.

## matchPattern

로그 항목에서 값을 추출하는 데 사용되는 정규식 패턴입니다. 로그 항목이 사전 정의된 로그 형식 중 하나가 아니면 이 설정이 사용됩니다. 이 설정이 사용되면 `customFieldNames`도 지정해야 합니다.

## customFieldNames

각각의 JSON 키 값 쌍에서 키로 사용되는 사용자 지정 필드 이름입니다. 이 설정을 사용하여 `matchPattern`에서 추출한 값에 필드 이름을 정의하거나 사전 정의된 로그 형식의 기본 필드 이름을 재정의합니다.

### Example : LOGTOJSON 구성

다음은 Apache Common Log 항목을 JSON 형식으로 변환하는 LOGTOJSON 구성의 예제입니다.

```
{
  "optionName": "LOGTOJSON",
  "logFormat": "COMMONAPACHELOG"
}
```

변환 전:

```
64.242.88.10 - - [07/Mar/2004:16:10:02 -0800] "GET /mailman/listinfo/hsdivision
HTTP/1.1" 200 6291
```

변환 후:

```
{"host":"64.242.88.10","ident":null,"authuser":null,"datetime":"07/
Mar/2004:16:10:02 -0800","request":"GET /mailman/listinfo/hsdivision
HTTP/1.1","response":"200","bytes":"6291"}
```

### Example : 사용자 지정 필드가 있는 LOGTOJSON 구성

다음은 LOGTOJSON 구성의 또 다른 예제입니다.

```
{
  "optionName": "LOGTOJSON",
  "logFormat": "COMMONAPACHELOG",
  "customFieldNames": ["f1", "f2", "f3", "f4", "f5", "f6", "f7"]
}
```

이 구성 설정을 사용하면 이전 예제의 동일한 Apache Common Log 항목이 다음과 같이 JSON 형식으로 변환됩니다.

```
{"f1":"64.242.88.10","f2":null,"f3":null,"f4":"07/Mar/2004:16:10:02 -0800","f5":"GET / mailman/listinfo/hsdivision HTTP/1.1","f6":"200","f7":"6291"}
```

Example : Apache Common Log 항목 변환

다음 흐름 구성은 Apache Common Log 항목을 JSON 형식의 한 줄 레코드로 변환합니다.

```
{
  "flows": [
    {
      "filePattern": "/tmp/app.log*",
      "kinesisStream": "my-stream",
      "dataProcessingOptions": [
        {
          "optionName": "LOGTOJSON",
          "logFormat": "COMMONAPACHELOG"
        }
      ]
    }
  ]
}
```

Example : 여러 줄 레코드 변환

다음 흐름 구성은 첫 줄이 "[SEQUENCE="로 시작하는 여러 줄 레코드를 구문 분석합니다. 먼저 각각의 레코드가 한 줄 레코드로 변환됩니다. 그런 다음 탭 구분 기호를 기반으로 레코드에서 값이 추출됩니다. 추출된 값은 지정된 customFieldNames 값에 매핑되어 JSON 형식의 한 줄 레코드를 형성합니다.

```
{
  "flows": [
    {
      "filePattern": "/tmp/app.log*",
      "kinesisStream": "my-stream",
      "multiLineStartPattern": "\\[SEQUENCE=",
      "dataProcessingOptions": [
        {
          "optionName": "SINGLELINE"
        }
      ]
    }
  ]
}
```

```

    },
    {
      "optionName": "CSVTOJSON",
      "customFieldNames": [ "field1", "field2", "field3" ],
      "delimiter": "\\t"
    }
  ]
}

```

### Example : 일치 패턴이 있는 LOGTOJSON 구성

다음은 마지막 필드(바이트)가 생략되어 있으며 Apache Common Log 항목을 JSON 형식으로 변환하는 LOGTOJSON 구성의 예제입니다.

```

{
  "optionName": "LOGTOJSON",
  "logFormat": "COMMONAPACHELOG",
  "matchPattern": "^(\\d\\.\\d+\\.\\d+\\.\\d+) (\\S+) (\\S+) \\[[([\\w:/]+\\s[+\\-]\\d{4})\\]\\] \\\"(.+?)\\\" (\\d{3})",
  "customFieldNames": ["host", "ident", "authuser", "datetime", "request", "response"]
}

```

변환 전:

```

123.45.67.89 - - [27/Oct/2000:09:27:09 -0400] "GET /java/javaResources.html HTTP/1.0"
200

```

변환 후:

```

{"host":"123.45.67.89","ident":null,"authuser":null,"datetime":"27/Oct/2000:09:27:09-0400","request":"GET /java/javaResources.html HTTP/1.0","response":"200"}

```

## 에이전트 CLI 명령

시스템 시작 시 에이전트가 자동으로 시작됩니다.

```

sudo chkconfig aws-kinesis-agent on

```

에이전트의 상태를 확인합니다:

```
sudo service aws-kinesis-agent status
```

에이전트를 중지합니다.

```
sudo service aws-kinesis-agent stop
```

이 위치에서 에이전트의 로그 파일을 읽습니다.

```
/var/log/aws-kinesis-agent/aws-kinesis-agent.log
```

에이전트를 제거합니다.

```
sudo yum remove aws-kinesis-agent
```

## FAQ

Windows용 Kinesis 에이전트가 있나요?

[Windows용 Kinesis 에이전트](#)는 Linux 플랫폼용 Kinesis 에이전트와 다른 소프트웨어입니다.

왜 Kinesis 에이전트가 느려지거나 **RecordSendErrors**가 증가하나요?

이는 대개 Kinesis에서 제한하기 때문입니다. Kinesis Data Streams의 `WriteProvisionedThroughputExceeded` 측정치 또는 Firehose 전송 `ThrottledRecords` 스트림의 측정치를 확인하십시오. 이 지표 중 0에서 증가한 수치가 있으면 스트림 제한을 늘려야 한다는 의미입니다. 자세한 내용은 [Kinesis Data Stream limits](#) 및 [Amazon Firehose Delivery Streams](#)를 참조하십시오.

제한을 확인한 후에는 Kinesis 에이전트가 대량의 작은 파일을 테일링하도록 구성되어 있는지 확인하십시오. Kinesis 에이전트가 새 파일을 테일링할 때 지연이 발생하므로 Kinesis 에이전트는 소량의 대용량 파일을 추적합니다. 로그 파일을 더 큰 파일로 통합해 보세요.

왜 **java.lang.OutOfMemoryError** 예외가 발생하나요?

Kinesis 에이전트에는 현재 워크로드를 처리할 메모리가 충분하지 않습니다. `/usr/bin/start-aws-kinesis-agent`의 `JAVA_START_HEAP` 및 `JAVA_MAX_HEAP`을 늘리고 에이전트를 다시 시작해 보세요.

## 왜 `IllegalStateException : connection pool shut down` 예외가 발생하나요?

Kinesis 에이전트에는 현재 워크로드를 처리할 연결이 충분하지 않습니다. `/etc/aws-kinesis/agent.json`에서 일반 에이전트 구성 설정의 `maxConnections` 및 `maxSendingThreads`를 늘려 보세요. 이 필드의 기본값은 제공되는 런타임 프로세서의 12배입니다. 고급 에이전트 구성 설정에 대한 자세한 [AgentConfiguration내용은.java](#)를 참조하십시오.

### Kinesis 에이전트의 다른 문제는 어떻게 디버그할 수 있나요?

`/etc/aws-kinesis/log4j.xml`에서 DEBUG 레벨 로그를 활성화할 수 있습니다.

### Kinesis 에이전트는 어떻게 구성하나요?

`maxBufferSizeBytes`의 크기가 작을수록 Kinesis 에이전트는 더 자주 데이터를 전송합니다. 이렇게 하면 레코드의 전송 시간이 줄어 유용하지만, Kinesis에 대한 초당 요청 수가 증가합니다.

### 왜 Kinesis 에이전트가 중복 레코드를 보내나요?

이 문제는 파일 테일링이 잘못 구성되어 발생합니다. `fileFlow's filePattern`마다 매칭되는 파일은 하나뿐이어야 합니다. 사용 중인 `logrotate` 모드가 `copytruncate` 모드인 경우에도 이 문제가 발생할 수 있습니다. 모드를 기본 모드 또는 생성 모드로 변경하여 중복을 피하세요. 중복 레코드 처리에 대한 자세한 내용은 [중복 레코드 처리](#)를 참조하세요.

## 다른 AWS 서비스를 사용하여 Kinesis Data Streams에 쓰기

다음은 Kinesis Data Streams에 데이터를 쓰기 위해 Kinesis Data Streams와 직접 통합할 수 있는 다른 AWS 서비스의 목록입니다.

### 주제

- [AWS Amplify](#)
- [Amazon Aurora](#)
- [Amazon CloudFront](#)
- [Amazon CloudWatch Logs](#)
- [Amazon Connect](#)
- [AWS Database Migration Service](#)
- [Amazon DynamoDB](#)
- [Amazon EventBridge](#)

- [AWS IoT Core](#)
- [Amazon Relational Database Service](#)
- [Amazon Pinpoint](#)
- [Amazon Quantum Ledger Database](#)

## AWS Amplify

Amazon Kinesis Data Streams를 사용하면 실시간 처리를 위해 AWS Amplify로 구축된 모바일 애플리케이션에서 데이터를 쉽게 스트리밍할 수 있습니다. 그런 다음 실시간 대시보드를 구축하고, 예외를 캡처하고, 알림을 생성하고, 권장 사항을 제시하고, 기타 실시간 비즈니스 또는 운영 결정을 내릴 수 있습니다. Amazon Simple Storage Service, Amazon DynamoDB, Amazon Redshift 등의 다른 서비스로 데이터를 쉽게 전송할 수도 있습니다.

자세한 내용은 AWS Amplify Developer Center의 [Using Amazon Kinesis](#)를 참조하세요.

## Amazon Aurora

Amazon Kinesis Data Streams를 사용하여 Amazon Aurora DB 클러스터의 활동을 모니터링할 수 있습니다. Aurora DB 클러스터는 데이터베이스 활동 스트림을 사용하여 Amazon Kinesis 데이터 스트림에 활동을 실시간으로 푸시합니다. 그런 다음 이러한 활동을 소비 및 감사하고 알림을 생성하는 규정 준수 관리용 애플리케이션을 구축할 수 있습니다. Amazon Firehose를 사용하여 데이터를 저장할 수도 있습니다.

자세한 내용은 Amazon Aurora 사용 설명서의 [데이터베이스 활동 스트림](#)을 참조하세요.

## Amazon CloudFront

CloudFront 실시간 로그와 함께 Amazon Kinesis Data Streams를 사용하고 배포에 대한 요청 관련 정보를 실시간으로 얻을 수 있습니다. 자체 [Kinesis 데이터 스트림 소비자](#)를 생성하거나 Amazon Firehose를 사용하여 로그 데이터를 Amazon Simple Storage Service(S3), Amazon Redshift, Amazon OpenSearch Service(OpenSearch Service) 또는 서드 파티 로그 처리 서비스로 전송할 수 있습니다.

자세한 내용을 알아보려면 Amazon CloudFront 개발자 안내서의 [실시간 로그](#)를 참조하세요.

## Amazon CloudWatch Logs

CloudWatch 구독을 사용하면 Amazon CloudWatch Logs에서 로그 이벤트의 실시간 피드에 액세스하고 처리, 분석 및 다른 시스템에 로드를 위해 Amazon Kinesis Data Stream으로 전송할 수 있습니다.

자세한 내용은 Amazon CloudWatch Logs 사용 설명서의 [구독을 통한 로그 데이터 실시간 처리](#)를 참조하세요.

## Amazon Connect

Kinesis Data Streams를 사용하여 Amazon Connect 인스턴스에서 실시간으로 연락처 레코드와 에이전트 이벤트를 내보낼 수 있습니다. 또한 Amazon Connect Customer Profiles에서 데이터 스트리밍을 활성화하여 새 프로파일 생성 또는 기존 프로파일 변경에 대한 Kinesis Data Stream의 업데이트를 자동으로 수신할 수 있습니다.

그런 다음 소비자 애플리케이션을 구축하여 실시간으로 데이터를 처리하고 분석할 수 있습니다. 예를 들어 연락처 레코드와 고객 프로파일 데이터를 사용하여 CRM 및 마케팅 자동화 도구와 같은 소스 시스템 데이터를 최신 정보로 유지할 수 있습니다. 에이전트 이벤트 데이터를 사용하여 에이전트 정보와 이벤트를 표시하는 대시보드를 생성하고 특정 에이전트 활동의 사용자 지정 알림을 트리거할 수 있습니다.

자세한 내용은 Amazon Connect 관리자 안내서의 [data streaming for your instance](#), [set up real-time export](#) 및 [agent event streams](#)를 참조하세요.

## AWS Database Migration Service

AWS Database Migration Service를 사용하여 데이터를 Amazon Kinesis Data Stream으로 마이그레이션할 수 있습니다. 그런 다음 실시간으로 데이터 레코드를 처리하는 소비자 애플리케이션을 구축할 수 있습니다. Amazon Simple Storage Service, Amazon DynamoDB, Amazon Redshift 등의 다른 서비스로 데이터를 다운스트림으로 쉽게 전송할 수도 있습니다.

자세한 내용을 알아보려면 AWS Database Migration Service 사용 설명서의 [Kinesis Data Streams 사용](#)을 참조하세요.

## Amazon DynamoDB

Amazon Kinesis Data Streams를 사용하여 Amazon DynamoDB 변경 사항을 캡처할 수 있습니다. Kinesis 데이터 스트림은 모든 DynamoDB 테이블에서 항목 수준 수정 사항을 캡처하여 Kinesis 데이터 스트림에 복제합니다. 소비자 애플리케이션은 이 스트림에 액세스하여 항목 수준 변경 사항을 실시간으로 확인하고 해당 변경 사항을 다운스트림으로 전송하거나 콘텐츠에 따라 조치를 취할 수 있습니다.

자세한 내용은 Amazon DynamoDB 개발자 안내서의 [Kinesis Data Streams가 DynamoDB에서 작동하는 방식](#)을 참조하세요.



## Amazon EventBridge

Kinesis Data Streams를 사용하면 EventBridge의 AWS API 호출 [이벤트](#)를 스트림으로 전송하고, 소비자 애플리케이션을 구축하고, 대량의 데이터를 처리할 수 있습니다. 또한 EventBridge 파이프에서 Kinesis Data Streams를 대상으로 사용하고 선택적 필터링 및 보강 후 사용 가능한 소스 중 하나에서 스트림으로 레코드를 전송할 수 있습니다.

자세한 내용은 Amazon EventBridge 사용 설명서의 [Send events to an Amazon Kinesis stream](#)과 [EventBridge Pipes](#)를 참조하세요.

## AWS IoT Core

AWS IoT 규칙 작업을 사용하면 AWS IoT Core의 MQTT 메시지에서 실시간으로 데이터를 쓸 수 있습니다. 그런 다음 데이터를 처리하고, 해당 내용을 분석하고, 알림을 생성하고, 이를 분석 애플리케이션이나 기타 AWS 서비스에 전송하는 애플리케이션을 구축할 수 있습니다.

자세한 내용은 AWS IoT Core 개발자 안내서의 [Kinesis Data Streams](#)를 참조하세요.

## Amazon Relational Database Service

Amazon Kinesis Data Streams를 사용하여 Amazon RDS 인스턴스의 활동을 모니터링할 수 있습니다. Amazon RDS는 데이터베이스 활동 스트림을 사용하여 Amazon Kinesis 데이터 스트림에 활동을 실시간으로 푸시합니다. 그런 다음 이러한 활동을 소비 및 감사하고 알림을 생성하는 규정 준수 관리용 애플리케이션을 구축할 수 있습니다. Amazon Firehose를 사용하여 데이터를 저장할 수도 있습니다.

자세한 내용은 Amazon RDS 사용 설명서의 [데이터베이스 활동 스트림](#)을 참조하세요.

## Amazon Pinpoint

이벤트 데이터를 Amazon Kinesis Data Streams로 전송하도록 Amazon Pinpoint를 설정할 수 있습니다. Amazon Pinpoint는 캠페인, 여정, 트랜잭션 이메일 및 SMS 메시지에 대한 이벤트 데이터를 전송할 수 있습니다. 그런 다음 데이터를 분석 애플리케이션으로 수집하거나 이벤트 내용에 따라 조치를 취하는 자체 소비자 애플리케이션을 구축할 수 있습니다.

자세한 내용은 Amazon Pinpoint 개발자 안내서의 [Streaming Events](#)를 참조하세요.

## Amazon Quantum Ledger Database

저널에 커밋된 모든 문서 개정을 캡처하고 이 데이터를 Amazon Kinesis Data Streams로 실시간 전송하는 스트림을 QLDB에서 생성할 수 있습니다. QLDB 스트림은 원장의 저널에서 Kinesis 데이터 스트

림 리소스로의 지속적인 데이터 흐름입니다. 그런 다음 Kinesis 스트리밍 플랫폼 또는 Kinesis Client Library를 사용하여 스트림을 소비하고, 데이터 레코드를 처리하고, 데이터 콘텐츠를 분석할 수 있습니다. QLDB 스트림은 control, block summary, revision details의 세 가지 레코드 유형으로 Kinesis Data Streams에 데이터를 씁니다.

자세한 내용을 알아보려면 Amazon QLDB Developer Guide의 [Streams](#)를 참조하세요.

## 타사 통합 사용

Kinesis Data Streams와 통합되는 다음 타사 옵션 중 하나를 사용하여 Kinesis Data Streams에 데이터를 쓸 수 있습니다.

주제

- [Apache Flink](#)
- [Fluentd](#)
- [Debezium](#)
- [Oracle GoldenGate](#)
- [Kafka 연결](#)
- [Adobe Experience](#)
- [Striim](#)

### Apache Flink

Apache Flink는 무제한 및 제한 데이터 스트림에 대한 상태 저장 계산을 위한 프레임워크 및 분산 처리 엔진입니다. Apache Flink에서 Kinesis Data Streams에 쓰는 방법에 대한 자세한 내용은 [Amazon Kinesis Data Streams Connector](#)를 참조하세요.

### Fluentd

Fluentd는 통합 로깅 계층을 위한 오픈 소스 데이터 수집기입니다. Fluentd에서 Kinesis Data Streams에 쓰는 방법에 대한 자세한 내용은 [Stream processing with Kinesis](#)를 참조하세요.

### Debezium

Debezium은 변경 데이터 캡처를 위한 오픈 소스 분산 플랫폼입니다. Debezium에서 Kinesis Data Streams에 쓰는 방법에 대한 자세한 내용은 [Streaming MySQL Data Changes to Amazon Kinesis](#)를 참조하세요.

## Oracle GoldenGate

Oracle GoldenGate는 한 데이터베이스의 데이터를 다른 데이터베이스로 복제, 필터링 및 변환할 수 있는 소프트웨어 제품입니다. Oracle GoldenGate에서 Kinesis Data Streams에 쓰는 방법에 대한 자세한 내용은 [Data replication to Kinesis Data Stream using Oracle GoldenGate](#)를 참조하세요.

## Kafka 연결

Kafka Connect는 Apache Kafka와 다른 시스템 간에 데이터를 확장 가능하고 안정적으로 스트리밍하기 위한 도구입니다. Apache Kafka에서 Kinesis Data Streams에 데이터를 쓰는 방법에 대한 자세한 내용은 [Kinesis kafka connector](#)를 참조하세요.

## Adobe Experience

Adobe Experience Platform을 통해 조직은 모든 시스템의 고객 데이터를 중앙 집중화하고 표준화할 수 있습니다. 그런 다음 데이터 과학과 기계 학습을 적용하여 풍부하고 개인화된 경험의 설계와 전송을 획기적으로 개선합니다. Adobe Experience Platform에서 Kinesis Data Streams에 데이터를 쓰는 방법에 대한 자세한 내용은 [Amazon Kinesis connection](#)을 생성하는 방법을 참조하세요.

## Striim

Striim은 데이터를 실시간으로 수집, 필터링, 변환, 강화, 집계, 분석 및 전송하는 완전한 엔드 투 엔드 인 메모리 플랫폼입니다. Striim에서 Kinesis Data Streams에 데이터를 쓰는 방법에 대한 자세한 내용은 [Kinesis Writer](#)를 참조하세요.

## Amazon Kinesis Data Streams 생산자 문제 해결

다음 섹션에서는 Amazon Kinesis Data Streams 생산자 작업 시 발견할 수 있는 몇 가지 일반적인 문제의 해결 방법을 제공합니다.

- [생산자 애플리케이션이 예상보다 느린 속도로 쓰고 있습니다.](#)
- [무단 KMS 마스터 키 권한 오류](#)
- [생산자를 위한 일반적인 문제, 질문 및 문제 해결 아이디어](#)

### 생산자 애플리케이션이 예상보다 느린 속도로 쓰고 있습니다.

예상보다 느린 속도의 쓰기 처리량에 대한 가장 공통적인 이유는 다음과 같습니다.

- [서비스 제한 초과](#)
- [생산자 최적화](#)

## 서비스 제한 초과

서비스 제한이 초과되는지 여부를 확인하려면, 생산자가 서비스에 대해 처리량 예외를 발생하는지 여부를 확인하고 병목 현상이 발생한 API 작업을 검사합니다. 호출에 따라 여러 제한이 있음을 유의하십시오([할당량과 제한](#) 참조). 예를 들어, 가장 잘 알려진 쓰기 및 읽기에 대한 샤드 수준 제한 외에도 다음과 같은 스트림 수준 제한이 있습니다.

- [CreateStream](#)
- [DeleteStream](#)
- [ListStreams](#)
- [GetShardIterator](#)
- [MergeShards](#)
- [DescribeStream](#)
- [DescribeStreamSummary](#)

CreateStream, DeleteStream, ListStreams, GetShardIterator 및 MergeShards 작업은 초당 호출 5번으로 제한됩니다. DescribeStream 작업은 초당 호출 10번으로 제한됩니다. DescribeStreamSummary 작업은 초당 호출 20번으로 제한됩니다.

이러한 호출이 문제가 되지 않는 경우, 모든 샤드에 균등하게 넣기 작업을 배포할 수 있는 파티션 키를 선택했는지 및 나머지 파티션이 서비스 제한에 도달하지 않았을 때 특정 파티션 키가 서비스 제한에 도달했는지를 확인하십시오. 이렇게 하려면 최대 처리량을 측정하고 스트림의 샤드 수를 고려해야 합니다. 스트림 관리에 대한 자세한 내용은 [스트림 생성 및 관리](#) 단원을 참조하십시오.

### Tip

단일 레코드 작업 [PutRecord](#)를 사용할 때는 처리량 제한 계산을 가장 가까운 킬로바이트로 올림해야 하는 반면, 멀티 레코드 작업 [PutRecords](#)에서는 각 호출에서 레코드의 누적 합계를 반올림해야 합니다. 예를 들어, 크기가 1.1KB인 레코드 600개의 PutRecords 요청은 병목 현상이 발생하지 않습니다.

## 생산자 최적화

생산자 최적화를 시작하기 전에 완료해야 할 몇 가지 핵심 작업이 있습니다. 먼저 레코드 크기와 초당 레코드를 기준으로 원하는 최대 처리량을 식별합니다. 그런 다음 스트림 용량을 제한 요인([서비스 제한 초과](#))으로 배제합니다. 스트림 용량을 배제한 경우, 두 가지 일반적인 생산자 유형에 대해 다음의 문제 해결 팁과 최적화 지침을 사용하십시오.

### 대규모 생산자

대규모 생산자는 일반적으로 온프레미스 서버 또는 Amazon EC2 인스턴스에서 실행됩니다. 대규모 생산자로부터 더 높은 처리량을 필요로 하는 고객은 일반적으로 초당 지연 시간을 고려합니다. 지연 시간을 처리하는 전략에는 다음 사항이 포함됩니다. 고객이 레코드를 마이크로 배치/버퍼 처리할 수 있는 경우 [Kinesis Producer Library](#)(고급 집계 로직이 있음) 또는 멀티 레코드 작업 [PutRecords](#)를 사용하거나, 레코드를 더 큰 파일로 집계한 후 단일 레코드 작업 [PutRecord](#)를 사용합니다. 배치/버퍼를 사용할 수 없는 경우 여러 스레드를 사용하여 동시에 Kinesis Data Streams 서비스에 쏩니다. AWS SDK for Java 및 다른 SDK에는 매우 적은 수의 코드로 이 작업을 수행할 수 있는 비동기 클라이언트가 포함됩니다.

### 소규모 생산자

소규모 생산자는 일반적으로 모바일 앱, IoT 디바이스 또는 웹 클라이언트입니다. 모바일 앱인 경우 AWS Mobile SDK에서 PutRecords 작업 또는 Kinesis 레코더를 사용하는 것이 좋습니다. 자세한 내용은 AWS Mobile SDK for Android 시작 안내서와 AWS Mobile SDK for iOS 시작 안내서를 참조하십시오. 모바일 앱은 간헐적 연결을 본질적으로 처리해야 하며, PutRecords와 같은 일종의 배치 넣기를 필요로 합니다. 몇 가지 사유로 인해 배치를 사용할 수 없는 경우 위의 대규모 생산자 정보를 참조하십시오. 생산자가 브라우저인 경우 생성되는 데이터의 양은 일반적으로 매우 작습니다. 그러나 넣기 작업을 애플리케이션의 중요한 경로에 넣는 것이므로 이 방법은 권장되지 않습니다.

## 무단 KMS 마스터 키 권한 오류

생산자 애플리케이션이 KMS 마스터 키에 대한 권한 없이 암호화된 스트림에 작성하는 경우 이 오류가 발생합니다. KMS 키에 대한 액세스 권한을 애플리케이션에 할당하려면 [AWS KMS에서 키 정책 사용](#) 및 [AWS KMS에서 IAM 정책 사용](#)을 참조하십시오.

## 생산자를 위한 일반적인 문제, 질문 및 문제 해결 아이디어

- [Kinesis 데이터 스트림이 500 내부 서버 오류를 반환하는 이유는 무엇인가요?](#)
- [FFlink에서 Kinesis Data Streams로 작성 시 발생하는 시간 초과 오류를 해결하려면 어떻게 해야 하나요?](#)

- [Kinesis Data Streams에서 제한 오류를 해결하려면 어떻게 해야 하나요?](#)
- [Kinesis 데이터 스트림이 제한되는 이유는 무엇인가요?](#)
- [KPL을 사용하여 Kinesis 데이터 스트림에 데이터 레코드를 넣으려면 어떻게 해야 하나요?](#)

## Kinesis Data Streams 생산자를 위한 고급 주제

이 섹션에서는 Amazon Kinesis Data Streams 생산자를 최적화하는 방법을 설명합니다.

### 주제

- [KPL 재시도 및 속도 제한](#)
- [KPL 집계 사용 시 고려 사항](#)

## KPL 재시도 및 속도 제한

KPL `addUserRecord()` 작업을 사용하여 Kinesis Producer Library(KPL) 사용자 레코드를 추가하면 레코드에 타임스탬프가 제공되고 `RecordMaxBufferedTime` 구성 파라미터를 통해 설정한 기한과 함께 레코드가 버퍼에 추가됩니다. 이 타임스탬프/기한 조합은 버퍼 우선순위를 설정합니다. 다음 기준에 따라 레코드가 버퍼에서 풀러시됩니다.

- 버퍼 우선 순위
- 집계 구성
- 수집 구성

버퍼 동작에 영향을 주는 집계 및 수집 구성 파라미터는 다음과 같습니다.

- `AggregationMaxCount`
- `AggregationMaxSize`
- `CollectionMaxCount`
- `CollectionMaxSize`

풀러시된 레코드는 Kinesis Data Streams API 작업 `PutRecords`에 대한 호출을 사용하여 Amazon Kinesis Data Streams 레코드로 Kinesis 데이터 스트림으로 전송됩니다. `PutRecords` 작업은 가끔 전체 또는 부분적 오류가 나타나는 스트림에 요청을 전송합니다. 실패한 레코드는 KPL 버퍼에 자동으로 다시 추가됩니다. 다음 두 가지 값 중 최소값을 기반으로 새로운 기한이 설정됩니다.

- 현재 RecordMaxBufferedTime 구성의 절반
- 레코드의 time-to-live 값

이 전략으로 Kinesis Data Streams 레코드의 time-to-live 값을 적용하면서 처리량을 높이고 복잡성을 줄이기 위해 재시도된 KPL 사용자 레코드를 후속 Kinesis Data Streams API 호출에 포함할 수 있습니다. 백오프 알고리즘이 없어 비교적 적극적인 재시도 전략입니다. 다음 단원에서 설명하는 속도 제한을 통해 과도한 재시도로 인한 스팸 발송을 방지합니다.

## 속도 제한

KPL에는 속도 제한 기능이 있어 단일 생산자가 전송하는 샤드당 처리량을 제한합니다. 속도 제한은 Kinesis Data Streams 레코드와 바이트 모두에 대해 별도의 버킷이 있는 토큰 버킷 알고리즘을 사용하여 구현됩니다. Kinesis 데이터 스트림에 성공적으로 쓸 때마다 특정 임계값까지 각 버킷에 토큰이 한 개 또는 여러 개 추가됩니다. 이 임계값은 구성할 수 있지만 기본적으로 실제 샤드 제한보다 50퍼센트 높게 설정되어 단일 생산자의 샤드 포화를 허용합니다.

이 제한을 낮추어 과도한 재시도로 인한 스팸 발송을 줄일 수 있습니다. 그러나 모범 사례는 각 생산자가 최대 처리량을 적극적으로 재시도하고, 스트림의 용량을 확장하고 적절한 파티션 키 전략을 구현하여 과하다고 판단되는 결과를 조절하는 것입니다.

## KPL 집계 사용 시 고려 사항

생성된 Amazon Kinesis Data Streams 레코드의 시퀀스 번호 체계가 동일하게 유지되는 동안 집계는 집계된 Kinesis Data Streams 레코드에 포함된 Kinesis Producer Library(KPL) 사용자 레코드의 인덱싱을 0에서 시작합니다. 그러나 시퀀스 번호를 사용하여 KPL 사용자 레코드를 고유하게 식별하지 않는 한 집계(KPL 사용자 레코드를 Kinesis Data Streams 레코드로) 및 분해(Kinesis Data Streams 레코드를 KPL 사용자 레코드로)로 이 작업이 자동 처리되므로 코드는 이를 무시할 수 있습니다. 소비자가 KCL을 사용하는지 AWS SDK를 사용하는지 관계없이 적용되는 사항입니다. 소비자가 AWS SDK에서 제공하는 API를 사용해 작성된 경우 이 집계 기능을 사용하려면 KPL의 Java 부분을 빌드로 가져와야 합니다.

시퀀스 번호를 KPL 사용자 레코드의 고유 식별자로 사용하려면 KPL 사용자 레코드를 비교할 수 있도록 Record 및 UserRecord에서 제공하는 계약 준수 `public int hashCode()` 및 `public boolean equals(Object obj)` 작업을 사용하는 것이 좋습니다. 또한 KPL 사용자 레코드의 하위 시퀀스 번호를 검사하려면 UserRecord 인스턴스로 해당 레코드를 캐스팅하고 하위 시퀀스 번호를 검색할 수 있습니다.

자세한 내용은 [소비자 분해](#) 섹션을 참조하세요.

## Amazon Kinesis Data Streams에서 데이터 읽기

소비자는 Kinesis 데이터 스트림의 모든 데이터를 처리하는 애플리케이션입니다. 소비자가 향상된 팬아웃을 사용하는 경우, 고유의 2MB/sec의 읽기 처리량 할당을 받으며 여러 소비자가 다른 소비자와 읽기 처리량을 경쟁할 필요 없이 동일한 스트림으로부터 동시에 데이터를 읽을 수 있습니다. 샤드의 향상된 팬아웃 기능을 사용하려면 [전용 처리량으로 사용자 지정 소비자 개발\(향상된 팬아웃\)](#) 단원을 참조하십시오.

기본적으로 스트림의 샤드는 샤드당 2MB/sec의 읽기 처리량을 제공합니다. 이 처리량은 해당 샤드로부터 읽는 모든 소비자에 대해 공유됩니다. 즉, 샤드로부터 읽는 소비자가 여러 개인 경우에도 샤드당 기본 2MB/sec의 처리량이 적용됩니다. 샤드에 대한 이 기본 처리량을 사용하려면 [공유 처리량으로 사용자 지정 소비자 개발](#) 단원을 참조하십시오.

다음 표에서는 기본 처리량을 향상된 팬아웃과 비교합니다. 메시지 전파 지연은 페이로드 디스패치 API (예: `Put`) 를 사용하여 전송된 페이로드가 페이로드를 소비하는 API (예: `PutRecord` 및 `PutRecords`) 를 통해 소비자 애플리케이션에 도달하는 데 걸리는 시간 (밀리초) 으로 정의됩니다. `GetRecords` `SubscribeToShard`

특성	향상된 팬아웃이 없는 미등록 소비자	향상된 팬아웃이 있는 등록 소비자
샤드 읽기 처리량	샤드당 총 2MB/sec로 고정됩니다. 동일한 샤드로부터 읽는 소비자가 여러 개인 경우 모두 이 처리량을 공유합니다. 샤드로부터 수신하는 처리량의 합은 2MB/sec를 넘지 않습니다.	소비자가 향상된 팬아웃을 사용하도록 등록될 때 확장됩니다. 향상된 팬아웃을 사용하도록 등록된 각 소비자는 다른 소비자와 독립적으로 샤드당 2MB/sec의 읽기 처리량을 받습니다.
메시지 전파 지연	한 소비자가 스트림에서 읽고 있을 경우 평균 약 200ms입니다. 소비자가 5개면 이 평균이 약 1,000ms로 증가합니다.	소비자가 1개든 또는 5개든 일반적으로 평균 70ms입니다.
비용	N/A	데이터 검색 비용과 소비자-샤드 시간 비용이 있습니다. 자세한 내용은 <a href="#">Amazon Kinesis Data Streams 요금</a> 을 참조하십시오.



특성	향상된 팬아웃이 없는 미등록 소비자	향상된 팬아웃이 있는 등록 소비자
레코드 전송 모델	를 사용하여 HTTP를 통해 모델을 가져옵니다. <a href="#">GetRecords</a>	Kinesis Data Streams는 HTTP/2를 사용하여 레코드를 사용자에게 푸시합니다. <a href="#">SubscribeToShard</a>

## 주제

- [Kinesis 콘솔에서 데이터 뷰어 사용](#)
- [Kinesis 콘솔에서 데이터 스트림을 쿼리하기](#)
- [를 사용한 소비자 개발 AWS Lambda](#)
- [Amazon Managed Service for Apache Flink를 사용하여 소비자 개발](#)
- [Amazon Data Firehose를 사용한 소비자 개발](#)
- [Kinesis Client Library 사용](#)
- [공유 처리량으로 사용자 지정 소비자 개발](#)
- [전용 처리량으로 사용자 지정 소비자 개발\(향상된 팬아웃\)](#)
- [KCL 1.x에서 KCL 2.x로 소비자 마이그레이션](#)
- [다른 AWS 서비스를 사용하여 Kinesis Data Streams에서 데이터 읽기](#)
- [타사 통합 사용](#)
- [Kinesis 데이터 스트림 소비자 문제 해결](#)
- [Amazon Kinesis Data Streams 소비자를 위한 고급 주제](#)

## Kinesis 콘솔에서 데이터 뷰어 사용

Kinesis Management Console의 데이터 뷰어를 사용하면 소비자 애플리케이션을 개발할 필요 없이 데이터 스트림의 지정된 샤드 내에 있는 데이터 레코드를 볼 수 있습니다. 데이터 뷰어를 사용하려면 다음 단계를 따릅니다.

1. [에 AWS Management Console 로그인하고 <https://console.aws.amazon.com/kinesis> 에서 Kinesis 콘솔을 엽니다.](#)
2. 데이터 뷰어로 레코드를 보려는 활성 데이터 스트림을 선택한 다음 데이터 뷰어 탭을 선택합니다.
3. 선택한 활성 데이터 스트림의 데이터 뷰어 탭에서 레코드를 보려는 샤드를 선택하고 시작 위치를 선택한 다음 레코드 가져오기를 클릭합니다. 시작 위치를 다음 값 중 하나로 설정할 수 있습니다.

- 시퀀스 번호에서: 시퀀스 번호 필드에 지정된 시퀀스 번호로 표시된 위치의 레코드를 표시합니다.
- 시퀀스 번호 이후: 시퀀스 번호 필드에 지정된 시퀀스 번호로 표시된 위치 바로 뒤의 레코드를 표시합니다.
- 타임 스탬프에서: 타임스탬프 필드에 지정된 타임스탬프로 표시된 위치의 레코드를 표시합니다.
- 트림 호라이즌: 샤드에서 가장 오래된 데이터 레코드인 샤드에서 트리밍되지 않은 마지막 레코드의 레코드를 표시합니다.
- 최신: 항상 샤드에서 가장 최근 데이터를 읽을 수 있도록 샤드에서 가장 최근 레코드 바로 뒤에 레코드를 표시합니다.

그러면 지정된 샤드 ID 및 시작 위치와 일치하는 생성된 데이터 레코드가 콘솔의 레코드 테이블에 표시됩니다. 한 번에 최대 50개의 레코드가 표시됩니다. 다음 레코드 세트를 보려면 다음 버튼을 클릭합니다.

4. 개별 레코드를 클릭하면 별도의 창에서 해당 레코드 페이로드를 원시 데이터 또는 JSON 형식으로 볼 수 있습니다.

참고로 데이터 뷰어에서 레코드 가져오기 또는 다음 버튼을 클릭하면 API가 호출되며, 이는 GetRecordsGetRecordsAPI 한도인 초당 5개 트랜잭션에 적용됩니다.

## Kinesis 콘솔에서 데이터 스트림을 쿼리하기

Kinesis Data Streams 콘솔의 데이터 분석 탭을 사용하면 SQL을 사용하여 데이터 스트림을 쿼리할 수 있습니다. 이 기능을 사용하려면 다음 단계를 따르십시오.

1. [에 AWS Management Console 로그인하고 https://console.aws.amazon.com/kinesis](https://console.aws.amazon.com/kinesis) 에서 Kinesis 콘솔을 엽니다.
2. SQL로 쿼리하려는 활성 데이터 스트림을 선택한 다음 데이터 분석 탭을 선택합니다.
3. 데이터 분석 탭에서 관리형 Apache Flink Studio 노트북을 사용하여 스트림 검사 및 시각화를 수행할 수 있습니다. Apache Zeppelin을 사용하면 임시 SQL 쿼리를 수행하여 데이터 스트림을 검사하고 몇 초 만에 결과를 확인할 수 있습니다. 데이터 분석 탭에서 동의함을 선택한 다음 노트북 만들기를 선택하여 노트북을 생성합니다.
4. 노트북을 만든 후 Apache Zeppelin에서 열기를 선택합니다. 그러면 노트북이 새 탭에서 열립니다. 노트북은 SQL 쿼리를 제출할 수 있는 대화형 인터페이스입니다. 스트림 이름이 포함된 메모를 선택합니다.

5. 이미 실행 중인 스트림의 데이터를 출력하기 위한 샘플 SELECT 쿼리가 포함된 메모가 표시됩니다. 이렇게 하면 데이터 스트림의 스키마를 볼 수 있습니다.
6. 텀블링 또는 슬라이딩 윈도우와 같은 다른 쿼리를 사용해 보려면 데이터 분석 탭에서 샘플 쿼리 보기를 선택합니다. 쿼리를 복사하고 데이터 스트림 스키마에 맞게 수정한 다음 Zeppelin 노트의 새 단락에서 실행하세요.

## 를 사용한 소비자 개발 AWS Lambda

AWS Lambda 함수를 사용하여 데이터 스트림의 레코드를 처리할 수 있습니다. AWS Lambda 서버를 프로비저닝하거나 관리하지 않고도 코드를 실행할 수 있는 컴퓨팅 서비스입니다. 이는 필요 시에만 코드를 실행하며, 하루에 몇 개의 요청에서 초당 수천 개의 요청까지 자동으로 확장 등의 조정이 가능합니다. 사용한 컴퓨팅 시간에 대해서만 비용을 지불하면 됩니다. 코드가 실행되지 않을 때는 비용이 부과되지 않습니다. 를 사용하면 관리 없이도 거의 모든 유형의 애플리케이션 또는 백엔드 서비스에 대한 코드를 실행할 수 있습니다. AWS Lambda는 고가용성 컴퓨팅 인프라에서 코드를 실행하고 서버와 운영 체제의 유지 관리, 용량 프로비저닝 및 자동 조정, 코드 모니터링 및 로깅 등 모든 컴퓨팅 리소스 관리를 수행합니다. 자세한 내용은 [Amazon AWS Lambda Kinesis와 함께 사용](#)을 참조하십시오.

문제 해결 정보는 [Kinesis Data Streams 트리거로 Lambda 함수를 호출할 수 없는 이유는 무엇인가요?](#)를 참조하세요.

## Amazon Managed Service for Apache Flink를 사용하여 소비자 개발

Amazon Managed Service for Apache Flink 애플리케이션을 사용하면 SQL, Java 또는 Scala를 사용하여 Kinesis 스트림의 데이터를 처리하고 분석할 수 있습니다. Managed Service for Apache Flink 애플리케이션은 참조 소스를 사용하여 데이터를 보강하고, 시간 경과에 따라 데이터를 집계하거나, 기계 학습을 사용하여 데이터 이상치를 찾을 수 있습니다. 그런 다음 다른 Kinesis 스트림, Firehose 전송 스트림 또는 Lambda 함수에 분석 결과를 기록할 수 있습니다. 자세한 내용은 [SQL 애플리케이션용 Managed Service for Apache Flink 개발자 안내서](#) 또는 [Flink 애플리케이션용 Managed Service for Apache Flink 개발자 안내서](#)를 참조하세요.

## Amazon Data Firehose를 사용한 소비자 개발

Firehose를 사용하여 Kinesis 스트림에서 레코드를 읽고 처리할 수 있습니다. Firehose는 Amazon S3, Amazon Redshift, Amazon 서비스 및 Splunk와 같은 대상에 실시간 스트리밍 데이터를 전송하는 완전 관리형 서비스입니다. OpenSearch 또한 Firehose는 Datadog, MongoDB, New Relic 등 지원되는 타사

서비스 공급자가 소유한 모든 사용자 지정 HTTP 엔드포인트 또는 HTTP 엔드포인트를 지원합니다. 데이터를 목적지로 전달하기 전에 데이터 레코드를 변환하고 레코드 형식을 변환하도록 Firehose를 구성할 수도 있습니다. 자세한 내용은 [Kinesis 데이터 스트림을 사용하여 Firehose에 쓰기](#)를 참조하십시오.

## Kinesis Client Library 사용

KDS 데이터 스트림의 데이터를 처리할 수 있는 사용자 지정 소비자 애플리케이션을 개발하는 방법 중 하나는 Kinesis Client Library(KCL)를 사용하는 것입니다.

주제

- [Kinesis Client Library란?](#)
- [KCL 사용 가능 버전](#)
- [KCL 개념](#)
- [리스 테이블을 사용하여 KCL 소비자 애플리케이션에서 처리한 샤드 추적](#)
- [동일한 'KCL 2.x for Java' 소비자 애플리케이션으로 여러 데이터 스트림 처리](#)
- [Kinesis 클라이언트 라이브러리를 AWS Glue 스키마 레지스트리와 함께 사용](#)

### Note

KCL 1.x와 KCL 2.x 모두 사용 시나리오에 따라 최신 KCL 1.x 버전 또는 KCL 2.x 버전으로 업그레이드하는 것이 좋습니다. KCL 1.x와 KCL 2.x 모두 최신 종속성 및 보안 패치, 버그 수정, 이전 버전과 호환되는 새 기능이 포함된 최신 릴리스로 정기적으로 업데이트됩니다. 자세한 내용은 <https://github.com/aws-labs/amazon-kinesis-client/releases>를 참조하십시오.

## Kinesis Client Library란?

KCL을 사용하면 분산 컴퓨팅과 관련된 많은 복잡한 작업을 처리하여 Kinesis 데이터 스트림의 데이터를 사용하고 처리할 수 있습니다. 여기에는 여러 소비자 애플리케이션 인스턴스 간의 로드 밸런싱, 소비자 애플리케이션 인스턴스 장애 대응, 처리된 레코드 체크포인트 및 리샤딩 대응이 포함됩니다. 사용자가 사용자 지정 레코드 처리 로직을 작성하는 데 집중할 수 있도록 KCL이 이러한 하위 작업을 모두 처리합니다.

KCL은 AWS SDK에서 사용할 수 있는 Kinesis Data Streams API와 다릅니다. Kinesis Data Streams API는 스트림 생성, 리샤딩, 레코드 넣기 및 가져오기를 비롯한 Kinesis Data Streams의 여러 측면을 관리하는 데 도움이 됩니다. 특히 사용자가 소비자 애플리케이션의 사용자 지정 데이터 처리 로직

에 집중할 수 있도록 KCL은 이러한 모든 하위 작업에 대한 추상화 계층을 제공합니다. Kinesis Data Streams API에 대한 자세한 내용은 [Amazon Kinesis API 참조](#)를 확인하세요.

### ⚠ Important

KCL은 Java 라이브러리입니다. Java 이외의 언어에 대한 지원은 라는 다국어 인터페이스를 사용하여 제공됩니다. MultiLangDaemon 이 데몬은 Java 기반이며, Java 이외의 KCL 언어를 사용하는 경우 배경에서 실행됩니다. 예를 들어, KCL for Python을 설치하고 소비자 응용 프로그램 전체를 Python으로 작성하는 경우 다음과 같은 이유로 시스템에 Java를 설치해야 합니다. MultiLangDaemon 또한 MultiLangDaemon 에는 사용 사례에 맞게 사용자 지정해야 할 수 있는 몇 가지 기본 설정 (예: 연결 AWS 지역) 이 있습니다. MultiLangDaemon GitHubon에 대한 자세한 내용은 [KCL MultiLangDaemon 프로젝트](#)를 참조하십시오.

KCL은 레코드 처리 로직과 Kinesis Data Streams 간에 중간 역할을 합니다. KCL은 다음과 같은 작업을 수행합니다.

- 데이터 스트림에 연결합니다
- 데이터 스트림 내 샤드를 열거합니다.
- 리스를 사용하여 워커와의 샤드 연결을 조정합니다.
- 관리하는 모든 샤드의 레코드 프로세서를 인스턴스화합니다
- 데이터 스트림에서 데이터 레코드를 가져옵니다.
- 해당하는 레코드 프로세서로 레코드를 푸시합니다
- 처리된 레코드에 대해 체크포인트를 수행합니다
- 워커 인스턴스 수가 변경되거나 데이터 스트림이 리샤딩(샤드 분할 또는 병합)될 때 샤드-워커 연결 (리스)의 균형을 유지합니다.

## KCL 사용 가능 버전

현재 지원되는 다음 KCL 버전 중 하나를 사용하여 사용자 지정 소비자 애플리케이션을 구축할 수 있습니다.

- KCL 1.x

자세한 내용은 [KCL 1.x 소비자 개발단원](#)을 참조하세요.

- KCL 2.x

자세한 내용은 [KCL 2.x 소비자 개발](#) 단원을 참조하세요.

KCL 1.x 또는 KCL 2.x를 사용하여 공유 처리량을 사용하는 소비자 애플리케이션을 구축할 수 있습니다. 자세한 정보는 [KCL을 사용하여 공유 처리량으로 사용자 지정 소비자 개발](#)을 참조하세요.

전용 처리량을 사용하는 소비자 애플리케이션(향상된 팬아웃 소비자)을 구축하려면 KCL 2.x만 사용할 수 있습니다. 자세한 정보는 [전용 처리량으로 사용자 지정 소비자 개발\(향상된 팬아웃\)](#)을 참조하세요.

KCL 1.x와 KCL 2.x의 차이점에 대한 자세한 내용과 KCL 1.x에서 KCL 2.x로 마이그레이션하는 방법에 대한 지침은 [KCL 1.x에서 KCL 2.x로 소비자 마이그레이션](#) 섹션을 참조하세요.

## KCL 개념

- KCL 소비자 애플리케이션 - KCL을 사용하여 맞춤 구축되고 데이터 스트림에서 레코드를 읽고 처리하도록 설계된 애플리케이션입니다.
- 소비자 애플리케이션 인스턴스 - KCL 소비자 애플리케이션은 일반적으로 장애를 조정하고 데이터 레코드 처리를 동적으로 로드 밸런싱하기 위해 하나 이상의 애플리케이션 인스턴스가 동시에 실행되는 분산형입니다.
- 워커 - KCL 소비자 애플리케이션 인스턴스가 데이터 처리를 시작하는 데 사용하는 상위 수준 클래스입니다.

### Important

KCL 소비자 애플리케이션 인스턴스마다 워커가 하나씩 있습니다.

워커는 샤드 및 리스 정보 동기화, 샤드 할당 추적, 샤드의 데이터 처리 등 다양한 작업을 초기화하고 감독합니다. 작업자는 KCL 소비자 응용 프로그램이 처리할 데이터 기록의 데이터 스트림 이름, 이 데이터 스트림에 액세스하는 데 필요한 AWS 자격 증명 등 소비자 응용 프로그램에 대한 구성 정보를 KCL에 제공합니다. 또한 워커는 데이터 스트림에서 레코드 프로세서로 데이터 레코드를 전송하기 위해 특정 KCL 소비자 애플리케이션 인스턴스를 시작합니다.

### Important

KCL 1.x에서는 이 클래스를 워커라고 합니다. Java KCL 리포지토리에 대한 자세한 내용은 <https://github.com/aws-labs/amazon-kinesis-client/blob/v1.x/src/main/java/com/Amazonaws/Services/Kinesis/ClientLibrary/lib/Worker/Worker.java>를 참조하십시오. KCL

2.x에서는 이 클래스를 스케줄러라고 합니다. KCL 2.x에서 스케줄러의 용도는 KCL 1.x에서 워커의 용도와 동일합니다. KCL 2.x의 스케줄러 클래스에 대한 자세한 내용은 [amazon-kinesis-client](https://github.com/aws-labs/amazon-kinesis-client/blob/master/src/main/java/software.amazon.kinesis.coordinator/Scheduler.java)를 참조하십시오.

- 리스 - 워커와 샤드 간의 바인딩을 정의하는 데이터입니다. 분산된 KCL 소비자 애플리케이션은 리스를 사용하여 워커 플릿의 데이터 레코드 처리를 분할합니다. 언제든지 각 데이터 레코드 샤드는 leaseKey 변수로 식별되는 리스를 통해 특정 워커에게 바인딩됩니다.

기본적으로 작업자는 하나 이상의 임대 계약 (작업자 변수 값에 따름) 을 동시에 보유할 수 있습니다. maxLeasesFor

#### Important

모든 워커는 데이터 스트림에서 사용 가능한 모든 샤드에 대해 사용 가능한 모든 리스를 보유하고자 경쟁합니다. 하지만 한 번에 하나의 워커만 각 리스를 성공적으로 보유할 수 있습니다.

예를 들어, 소비자 애플리케이션 인스턴스 A에 워커 A가 있고 이 인스턴스가 4개의 샤드로 구성된 데이터 스트림을 처리하는 경우 워커 A는 샤드 1, 2, 3, 4에 대한 리스를 동시에 보유할 수 있습니다. 그러나 A와 B라는 2개의 소비자 애플리케이션 인스턴스에 각각 워커 A와 워커 B가 있고 이들 인스턴스가 4개의 샤드로 구성된 데이터 스트림을 처리하는 경우 워커 A와 워커 B는 샤드 1에 대한 리스를 동시에 보유할 수 없습니다. 한 워커가 특정 샤드의 데이터 레코드 처리를 중지할 준비가 되거나 실패할 때까지 이 샤드에 대한 리스를 보유합니다. 한 워커가 리스 보유를 중지하면 다른 워커가 리스를 인수하여 보유합니다.

자세한 내용 (Java KCL 리포지토리) 은 <https://github.com/aws-labs/amazon-kinesis-client/blob/v1.x/SRC/main/java/com/Amazonaws/Services/Kinesis/Leases/impl/Lease>. KCL 1.x의 경우 <https://github.com/aws-labs/amazon-kinesis-client/blob/master/src/main/java/software.amazon.kinesis.leases/Lease.java> 를 참조하십시오. [amazon-kinesis-client](#) [amazon-kinesis-client](#)

- 리스 테이블 - KCL 소비자 애플리케이션의 워커가 리스하고 처리하고 있는 KDS 데이터 스트림의 샤드를 추적하는 데 사용되는 고유한 Amazon DynamoDB 테이블입니다. KCL 소비자 애플리케이션이 실행되는 동안 리스 테이블은 데이터 스트림의 최신 샤드 정보와 동기화된 상태를 유지해야 합니다 (워커 내 및 모든 워커 간). 자세한 정보는 [리스 테이블을 사용하여 KCL 소비자 애플리케이션에서 처리한 샤드 추적을 참조하십시오.](#)

- 레코드 프로세서 - KCL 소비자 애플리케이션이 데이터 스트림에서 가져온 데이터를 처리하는 방법을 정의하는 로직입니다. 런타임 시 KCL 소비자 애플리케이션 인스턴스는 워커를 인스턴스화하고 이 워커는 리스를 보유한 모든 샤드에 대해 하나의 레코드 프로세서를 인스턴스화합니다.

## 리스 테이블을 사용하여 KCL 소비자 애플리케이션에서 처리한 샤드 추적

### 주제

- [리스 테이블이란?](#)
- [처리량](#)
- [리스 테이블이 KDS 데이터 스트림의 샤드와 동기화되는 방법](#)

### 리스 테이블이란?

각 Amazon Kinesis Data Streams 애플리케이션에 대해 KCL은 Amazon DynamoDB 테이블에 저장되는 고유한 리스 테이블을 사용하여 KCL 소비자 애플리케이션의 워커가 리스하고 처리하고 있는 KDS 데이터 스트림의 샤드를 추적합니다.

#### Important

KCL은 소비자 애플리케이션의 이름을 사용하여 이 소비자 애플리케이션이 사용하는 리스 테이블의 이름을 생성하므로 각 소비자 애플리케이션 이름은 고유해야 합니다.

소비자 애플리케이션이 실행되는 동안 [Amazon DynamoDB 콘솔](#)을 사용하여 리스 테이블을 볼 수 있습니다.

애플리케이션이 시작될 때 KCL 소비자 애플리케이션에 대한 리스 테이블이 없는 경우 워커 중 하나가 이 애플리케이션에 대한 리스 테이블을 생성합니다.

#### Important

Kinesis Data Streams 자체와 관련된 비용 외에도 DynamoDB 테이블 관련 비용이 계정에 청구됩니다.

리스 테이블의 각 행은 소비자 애플리케이션의 워커가 처리 중인 샤드를 나타냅니다. KCL 소비자 애플리케이션이 하나의 데이터 스트림만 처리하는 경우 리스 테이블의 해시 키인 leaseKey가 샤드 ID



입니다. [동일한 'KCL 2.x for Java' 소비자 애플리케이션으로 여러 데이터 스트림 처리](#)를 수행하는 경우 leaseKey의 구조는 account-id:StreamName:streamCreationTimestamp:ShardId와 같습니다. 예를 들어 111111111:multiStreamTest-1:12345:shardId-000000000336입니다.

각 행에는 샤드 ID 외에도 다음 데이터가 포함됩니다.

- **checkpoint**: 샤드의 가장 최근 체크포인트 시퀀스 번호입니다. 이 값은 데이터 스트림의 모든 샤드에서 고유합니다.
- **checkpointSubSequence번호**: Kinesis 프로듀서 라이브러리의 집계 기능을 사용할 때 이는 Kinesis 레코드 내의 개별 사용자 레코드를 추적하는 체크포인트의 확장입니다.
- **leaseCounter**: 다른 작업자가 리스를 받았다는 것을 작업자가 감지할 수 있도록 리스 버전 관리에 사용됩니다.
- **leaseKey**: 임대인의 고유 식별자입니다. 각 리스는 데이터 스트림의 샤드에 특정적이며 한 번에 한 워커가 리스를 보유하고 있습니다.
- **leaseOwner**: 이 임대를 보유하고 있는 작업자입니다.
- **ownerSwitchesSince체크포인트**: 마지막으로 체크포인트를 작성한 이후 이번 임대로 인해 근로자가 몇 번이나 변경되었습니까?
- **parentShardId**: 하위 샤드에서 처리가 시작되기 전에 상위 샤드가 완전히 처리되었는지 확인하는 데 사용됩니다. 그러면 스트림에 입력된 순서대로 레코드가 처리됩니다.
- **hashrange**: PeriodicShardSyncManager에서 주기적 동기화를 실행하여 리스 테이블에서 누락된 샤드를 찾고 필요한 경우 리스를 생성하는 데 사용됩니다.

#### Note

이 데이터는 KCL 1.14 및 KCL 2.3부터 시작하는 모든 샤드의 리스 테이블에 있습니다. PeriodicShardSyncManager 및 리스와 샤드 간의 주기적 동기화에 대한 자세한 내용은 [리스 테이블이 KDS 데이터 스트림의 샤드와 동기화되는 방법](#) 섹션을 참조하세요.

- **childshards**: LeaseCleanupManager에서 하위 샤드의 처리 상태를 검토하고 리스 테이블에서 상위 샤드를 삭제할 수 있는지 여부를 결정하는 데 사용됩니다.

#### Note

이 데이터는 KCL 1.14 및 KCL 2.3부터 시작하는 모든 샤드의 리스 테이블에 있습니다.

- **shardID**: 샤드의 ID입니다.

**Note**

이 데이터는 사용자가 [동일한 'KCL 2.x for Java' 소비자 애플리케이션으로 여러 데이터 스트림 처리](#)를 수행하는 경우에만 리스 테이블에 있습니다. 이는 KCL 2.3 for Java 이상부터 시작하여 KCL 2.x for Java에서만 지원됩니다.

- stream name account-id:StreamName:streamCreationTimestamp 형식의 데이터 스트림 식별자입니다.

**Note**

이 데이터는 사용자가 [동일한 'KCL 2.x for Java' 소비자 애플리케이션으로 여러 데이터 스트림 처리](#)를 수행하는 경우에만 리스 테이블에 있습니다. 이는 KCL 2.3 for Java 이상부터 시작하여 KCL 2.x for Java에서만 지원됩니다.

## 처리량

Amazon Kinesis Data Streams 애플리케이션이 프로비저닝된 처리량을 예외를 수신하는 경우 DynamoDB 테이블의 프로비저닝된 처리량을 늘려야 합니다. KCL은 프로비저닝된 처리량이 초당 읽기 10개, 초당 쓰기 10개인 테이블을 생성하지만 애플리케이션에 충분하지 않을 수도 있습니다. 예를 들어, Amazon Kinesis Data Streams 애플리케이션이 체크포인트를 자주 수행하거나 여러 샤드로 구성된 스트림에서 작동하는 경우 처리량이 더 필요할 수 있습니다.

DynamoDB의 프로비저닝된 처리량에 대한 자세한 내용은 Amazon DynamoDB 개발자 안내서의 [읽기/쓰기 용량 모드](#) 및 [DynamoDB의 테이블 및 데이터 작업을](#) 참조하세요.

## 리스 테이블이 KDS 데이터 스트림의 샤드와 동기화되는 방법

KCL 소비자 애플리케이션의 워커는 리스를 사용하여 지정된 데이터 스트림의 샤드를 처리합니다. 특정 시간에 어떤 워커가 어떤 샤드를 리스하는지에 대한 정보는 리스 테이블에 저장됩니다. KCL 소비자 애플리케이션이 실행되는 동안 리스 테이블은 데이터 스트림의 최신 샤드 정보와 동기화된 상태를 유지해야 합니다. KCL은 소비자 애플리케이션 부트스트래핑(소비자 애플리케이션 초기화 또는 재시작 시) 동안 그리고 처리 중인 샤드가 종료(리샤딩)에 도달할 때마다 Kinesis Data Streams 서비스에서 획득한 샤드 정보와 리스 테이블을 동기화합니다. 즉, 워커 또는 KCL 소비자 애플리케이션은 초기 소비자 애플리케이션 부트스트랩 동안 그리고 소비자 애플리케이션에서 데이터 스트림 리샤딩 이벤트가 발생할 때마다 처리 중인 데이터 스트림과 동기화됩니다.

## 주제

- [KCL 1.0~1.13 및 KCL 2.0~2.2에서의 동기화](#)
- [KCL 2.x에서의 동기화\(KCL 2.3 이상부터\)](#)
- [KCL 1.x에서의 동기화\(KCL 1.14 이상부터\)](#)

### KCL 1.0~1.13 및 KCL 2.0~2.2에서의 동기화

KCL 1.0~1.13 및 KCL 2.0~2.2에서 KCL은 소비자 애플리케이션의 부트스트래핑 및 각 데이터 스트림 리샤딩 이벤트 중에 ListShards 또는 DescribeStream 검색 API를 간접적으로 호출하여 Kinesis Data Streams 서비스에서 획득한 샤드 정보와 리스 테이블을 동기화합니다. 위에 나열된 모든 KCL 버전에서 KCL 소비자 애플리케이션의 각 워커는 소비자 애플리케이션의 부트스트래핑 중 및 각 스트림 리샤드 이벤트에서 다음 단계를 완료하여 리스/샤드 동기화 프로세스를 수행합니다.

- 처리 중인 데이터 스트림에 대한 모든 샤드를 가져옵니다.
- 리스 테이블에서 모든 샤드 리스를 가져옵니다.
- 리스 테이블에 리스가 없는 각 열린 샤드를 필터링합니다.
- 발견된 모든 열린 샤드와 열린 상위 항목이 없는 각 열린 샤드를 반복합니다.
  - 상위 항목 경로를 통해 계층 트리를 탐색하여 샤드가 하위 항목인지 확인합니다. 상위 샤드가 처리 중이거나(상위 샤드에 대한 리스 항목이 리스 테이블에 있음) 상위 샤드를 처리해야 하는 경우(예: 초기 위치가 TRIM\_HORIZON 또는 AT\_TIMESTAMP인 경우) 샤드는 하위 항목으로 간주됩니다.
  - 컨텍스트의 열린 샤드가 하위 항목인 경우 KCL은 초기 위치를 기준으로 샤드 체크포인트를 수행하고 필요한 경우 상위 항목에 대한 리스를 생성합니다.

### KCL 2.x에서의 동기화(KCL 2.3 이상부터)

지원되는 최신 버전의 KCL 2.x(KCL 2.3) 이상부터 라이브러리는 이제 다음과 같은 동기화 프로세스 변경 사항을 지원합니다. 이러한 리스/샤드 동기화 변경은 KCL 소비자 애플리케이션에서 Kinesis Data Streams 서비스에 대한 API 호출 수를 크게 줄이고 KCL 소비자 애플리케이션의 리스 관리를 최적화합니다.

- 애플리케이션 부트스트래핑 중 리스 테이블이 비어 있으면 KCL은 ListShard API의 필터링 옵션 (ShardFilter 선택적 요청 파라미터)을 활용하여 ShardFilter 파라미터로 지정된 시간에 열려 있는 샤드의 스냅샷에 대해서만 리스를 검색하고 생성합니다. ShardFilter 파라미터를 사용하면 ListShards API의 응답을 필터링할 수 있습니다. ShardFilter 파라미터의 유일한 필수 속성은 Type입니다. KCL은 Type 필터 속성과 다음과 같은 유효한 값을 사용하여 새 리스가 필요할 수 있는 열린 샤드의 스냅샷을 식별하고 반환합니다.

- AT\_TRIM\_HORIZON - TRIM\_HORIZON에서 열려 있던 모든 샤드가 응답에 포함됩니다.
- AT\_LATEST - 데이터 스트림의 현재 열려 있는 샤드만 응답에 포함됩니다.
- AT\_TIMESTAMP - 시작 타임스탬프가 지정된 타임스탬프보다 작거나 같고 종료 타임스탬프가 지정된 타임스탬프보다 크거나 같거나 여전히 열려 있는 모든 샤드가 응답에 포함됩니다.

ShardFilter는 RetrievalConfig#initialPositionInStreamExtended에 지정된 샤드의 스냅샷에 대한 리스를 초기화하기 위해 빈 리스 테이블에 대한 리스를 생성할 때 사용됩니다.

ShardFilter에 대한 자세한 정보는 [https://docs.aws.amazon.com/kinesis/latest/APIReference/API\\_ShardFilter.html](https://docs.aws.amazon.com/kinesis/latest/APIReference/API_ShardFilter.html) 섹션을 참조하십시오.

- 모든 워커가 리스/하드 동기화를 수행하여 데이터 스트림의 최신 샤드로 리스 테이블을 최신 상태로 유지하는 대신 선출된 단일 워커 리더가 리스/샤드 동기화를 수행합니다.
- KCL 2.3은 GetRecords 및 SubscribeToShard API의 ChildShards 반환 파라미터를 사용하여 닫힌 샤드에 대해 SHARD\_END에서 발생하는 리스/샤드 동기화를 수행하므로 KCL 워커는 처리가 완료된 샤드의 하위 샤드에 대해서만 리스를 생성할 수 있습니다. 공유 처리량 소비자 애플리케이션의 경우 리스/하드 동기화의 이 최적화는 GetRecords API의 ChildShards 파라미터를 사용합니다. 전용 처리량(향상된 팬아웃) 소비자 애플리케이션의 경우 리스/하드 동기화의 이 최적화는 SubscribeToShard API의 ChildShards 파라미터를 사용합니다. 자세한 내용은, 및 을 참조하십시오 [GetRecords](#). [SubscribeToShardsChildShard](#)
- 위의 변경으로 인해 KCL의 동작은 모든 기존 샤드에 대해 학습하는 모든 워커 모델에서 각 워커가 소유한 샤드의 하위 샤드에 대해서만 학습하는 워커 모델로 바뀌고 있습니다. 따라서 소비자 애플리케이션 부트스트래핑 및 리샤드 이벤트 중 발생하는 동기화 외에도 KCL은 이제 리스 테이블의 잠재적 구멍을 식별하기 위해 즉, 모든 새로운 샤드에 대해 알아보기 위해 추가로 주기적 샤드/리스 스캔도 수행하여 데이터 스트림의 전체 해시 범위가 처리되고 있는지 확인하고 필요한 경우 리스를 생성합니다. PeriodicShardSyncManager는 주기적 리스/샤드 스캔 실행을 담당하는 구성 요소입니다.

KCL 2.3에 대한 PeriodicShardSyncManager 자세한 내용은 <https://github.com/aws-labs/amazon-kinesis-client/blob/master/src/main/java/software.amazon.kinesis/leases/java#L201> [amazon-kinesis-client-L213](#)을 참조하십시오. LeaseManagementConfig

KCL 2.3에서는 새로운 구성 옵션을 사용하여 LeaseManagementConfig에서 PeriodicShardSyncManager를 구성할 수 있습니다.

명칭	기본값	설명
leasesRecoveryAuditorExecutionFrequencyMillis	12만(2분)	리스 테이블에서 부분 리스를 검색하는 감사자 작업의 빈도(밀리초)입니다. 감사자가 스트림의 리스에서 구멍을 발견하면 leasesRecoveryAuditorInconsistencyConfidenceThreshold 를 기반으로 샤드 동기화를 트리거합니다.
leasesRecoveryAuditorInconsistencyConfidenceThreshold	3	리스 테이블의 데이터 스트림에 대한 리스가 불일치하는지 확인하기 위한 정기 감사 작업의 신뢰도 임계값입니다. 감사자가 데이터 스트림에 대해 동일한 불일치 세트를 여러 번 연속해서 발견하면 샤드 동기화가 트리거됩니다.

또한 이제 의 상태를 모니터링하기 위한 CloudWatch 새 메트릭이 생성되었습니다.

PeriodicShardSyncManager 자세한 정보는 [PeriodicShardSyncManager](#)을 참조하세요.

- 하나의 샤드 계층에 대해서만 리스를 생성하도록 HierarchicalShardSyncer에 대한 최적화를 포함합니다.

## KCL 1.x에서의 동기화(KCL 1.14 이상부터)

지원되는 최신 버전의 KCL 1.x(KCL 1.14) 이상부터 라이브러리는 이제 다음과 같은 동기화 프로세스 변경 사항을 지원합니다. 이러한 리스/샤드 동기화 변경은 KCL 소비자 애플리케이션에서 Kinesis Data Streams 서비스에 대한 API 호출 수를 크게 줄이고 KCL 소비자 애플리케이션의 리스 관리를 최적화합니다.

- 애플리케이션 부트스트래핑 중 리스 테이블이 비어 있으면 KCL은 ListShard API의 필터링 옵션 (ShardFilter 선택적 요청 파라미터)을 활용하여 ShardFilter 파라미터로 지정된 시간에 열려 있는 샤드의 스냅샷에 대해서만 리스를 검색하고 생성합니다. ShardFilter 파라미터를 사용하면 ListShards API의 응답을 필터링할 수 있습니다. ShardFilter 파라미터의 유일한 필수 속성은 Type입니다. KCL은 Type 필터 속성과 다음과 같은 유효한 값을 사용하여 새 리스가 필요할 수 있는 열린 샤드의 스냅샷을 식별하고 반환합니다.
  - AT\_TRIM\_HORIZON - TRIM\_HORIZON에서 열려 있던 모든 샤드가 응답에 포함됩니다.
  - AT\_LATEST - 데이터 스트림의 현재 열려 있는 샤드만 응답에 포함됩니다.
  - AT\_TIMESTAMP - 시작 타임스탬프가 지정된 타임스탬프보다 작거나 같고 종료 타임스탬프가 지정된 타임스탬프보다 크거나 같거나 여전히 열려 있는 모든 샤드가 응답에 포함됩니다.

## ShardFilter는

KinesisClientLibConfiguration#initialPositionInStreamExtended에 지정된 샤드의 스냅샷에 대한 리스를 초기화하기 위해 빈 리스 테이블에 대한 리스를 생성할 때 사용됩니다.

ShardFilter에 대한 자세한 정보는 [https://docs.aws.amazon.com/kinesis/latest/APIReference/API\\_ShardFilter.html](https://docs.aws.amazon.com/kinesis/latest/APIReference/API_ShardFilter.html) 섹션을 참조하십시오.

- 모든 워커가 리스/하드 동기화를 수행하여 데이터 스트림의 최신 샤드로 리스 테이블을 최신 상태로 유지하는 대신 선출된 단일 워커 리더가 리스/샤드 동기화를 수행합니다.
- KCL 1.14는 GetRecords 및 SubscribeToShard API의 ChildShards 반환 파라미터를 사용하여 닫힌 샤드에 대해 SHARD\_END에서 발생하는 리스/샤드 동기화를 수행하므로 KCL 워커는 처리가 완료된 샤드의 하위 샤드에 대해서만 리스를 생성할 수 있습니다. 자세한 내용은 [GetRecords](#) 및 [ChildShard](#)을 참조하십시오.

- 위의 변경으로 인해 KCL의 동작은 모든 기존 샤드에 대해 학습하는 모든 워커 모델에서 각 워커가 소유한 샤드의 하위 샤드에 대해서만 학습하는 워커 모델로 바뀌고 있습니다. 따라서 소비자 애플리케이션 부트스트래핑 및 리샤드 이벤트 중 발생하는 동기화 외에도 KCL은 이제 리스 테이블의 잠재적 구멍을 식별하기 위해 즉, 모든 새로운 샤드에 대해 알아보기 위해 추가로 주기적 샤드/리스 스캔도 수행하여 데이터 스트림의 전체 해시 범위가 처리되고 있는지 확인하고 필요한 경우 리스를 생성합니다. `PeriodicShardSyncManager`는 주기적 리스/샤드 스캔 실행을 담당하는 구성 요소입니다.

`KinesisClientLibConfiguration#shardSyncStrategyType`이 `ShardSyncStrategyType.SHARD_END`로 설정된 경우 `PeriodicShardSyncLeasesRecoveryAuditorInconsistencyConfidenceThreshold`는 샤드 동기화 시행을 위한 리스 테이블에 구멍이 포함된 연속 스캔 수에 대한 임계값을 결정하는데 사용됩니다. `KinesisClientLibConfiguration#shardSyncStrategyType`이 `ShardSyncStrategyType.PERIODIC`으로 설정된 경우 `LeasesRecoveryAuditorInconsistencyConfidenceThreshold`은 무시됩니다.

KCL 1.14에 대한 자세한 내용은 <https://github.com/aws-labs/PeriodicShardSyncManager-aws-kinesis-client/blob/v1.x/src/main/java/com/amazonaws/services/kinesis/clientlibrary/lib/worker/PeriodicShardSyncManager.java#L987> `KinesisClientLibConfiguration` -L999를 참조하십시오.

KCL 1.14에서는 새로운 구성 옵션을 사용하여 `LeaseManagementConfig`에서 `PeriodicShardSyncManager`를 구성할 수 있습니다.

명칭	기본값	설명
leasesRecoveryAuditorInconsistencyConfidenceThreshold	3	리스 테이블의 데이터 스트림에 대한 리스가 불일치하는지 확인하기 위한 정기 감사 작업의 신뢰도 임계값입니다. 감사자가 데이터 스트림에 대해 동일한 불일치 세트를 여러 번 연속해서 발견하면 샤드 동기화가 트리거됩니다.

또한 이제 의 상태를 모니터링하기 위한 CloudWatch 새로운 메트릭이 생성되었습니다.

PeriodicShardSyncManager 자세한 정보는 [PeriodicShardSyncManager](#)을 참조하세요.

- KCL 1.14는 이제 지연된 리스 정리도 지원합니다. 샤드가 데이터 스트림의 보존 기간을 지나 만료되었거나 리샤딩 작업의 결과로 닫힌 경우 SHARD\_END에 도달하면 LeaseCleanupManager가 리스를 비동기식으로 삭제합니다.

새로운 구성 옵션을 사용하여 LeaseCleanupManager를 구성할 수 있습니다.

명칭	기본값	설명
leaseCleanupIntervalMillis	1분	리스 정리 스레드를 실행하는 간격입니다.
completedLeaseCleanupIntervalMillis	5분	리스가 완료되었는지 여부를 확



명칭	기본값	설명
		인하는 간격입니다.
garbageLe aseCleanu pIntervalMillis	30 분	리스가 가비지 (즉, 데이터 스트림의 보존 기간을 지나 트리밍됨)인지 여부를 확인하는 간격입니다.

- 하나의 샤드 계층에 대해서만 리스를 생성하도록 `KinesisShardSyncer`에 대한 최적화를 포함합니다.

## 동일한 'KCL 2.x for Java' 소비자 애플리케이션으로 여러 데이터 스트림 처리

이 섹션에서는 두 개 이상의 데이터 스트림을 동시에 처리할 수 있는 KCL 소비자 응용 프로그램을 만들 수 있도록 하는 Java용 KCL 2.x의 다음 변경 사항에 대해 설명합니다.

### Important

멀티스트림 처리는 KCL 2.3 for Java 이상부터 시작하여 KCL 2.x for Java에서만 지원됩니다. KCL 2.x를 구현할 수 있는 다른 언어에 대해 멀티스트림 처리가 지원되지 않습니다. KCL 1.x의 모든 버전에서 멀티스트림 처리가 지원되지 않습니다.

- `MultistreamTracker` 인터페이스

여러 스트림을 동시에 처리할 수 있는 소비자 애플리케이션을 구축하려면 라는 새 인터페이스를 구현해야 합니다 [MultistreamTracker](#). 이 인터페이스에는 KCL 소비자 애플리케이션에서 처리할 데이터 스트림 및 해당 구성 목록을 반환하는 `streamConfigList` 메서드가 포함되어 있습니다. 처리 중인 데이터 스트림은 소비자 애플리케이션 런타임 중에 변경될 수 있습니다. `streamConfigList`는 처리할 데이터 스트림의 변경 사항을 알아보기 위해 KCL에서 주기적으로 직접적으로 호출됩니다.

`streamConfigList`메서드가 [StreamConfig](#)목록을 채웁니다.

```

package software.amazon.kinesis.common;

import lombok.Data;
import lombok.experimental.Accessors;

@Data
@Accessors(fluent = true)
public class StreamConfig {
    private final StreamIdentifier streamIdentifier;
    private final InitialPositionInStreamExtended initialPositionInStreamExtended;
    private String consumerArn;
}

```

StreamIdentifier 및 InitialPositionInStreamExtended는 필수 필드이고 consumerArn은 선택 사항입니다. KCL 2.x를 사용하여 향상된 팬아웃 소비자 애플리케이션을 구현하는 경우에만 consumerArn을 제공해야 합니다.

에 대한 자세한 내용은 <https://github.com/aws-labs/amazon-kinesis-client/blob/v2.5.8/StreamIdentifier/src/main/java/software/amazon/kinesis/common/amazon-kinesis-client.java#L129> 를 참조하십시오. StreamIdentifier 를 StreamIdentifier 생성하려면 v2.5.0 이상에서 사용할 수 있는 에서 멀티스트림 인스턴스를 만드는 것이 좋습니다. streamArn streamCreationEpoch 지원하지 않는 KCL v2.3 및 v2.4에서는 다음 형식을 사용하여 streamArn 멀티스트림 인스턴스를 생성합니다. account-id:StreamName:streamCreationTimestamp 이 형식은 더 이상 사용되지 않으며 다음 메이저 릴리스부터 더 이상 지원되지 않습니다.

MultistreamTracker에는 리스 테이블(formerStreamsLeasesDeletionStrategy)에서 오래된 스트림의 리스를 삭제하기 위한 전략도 포함되어 있습니다. 소비자 애플리케이션 런타임 중에는 전략을 변경할 수 없다는 점에 유의하세요. 자세한 내용은 <https://github.com/aws-labs/amazon-kinesis-client/blob/0c5042dadf794fe988438436252a5a8fe70b6b0b/src/main/java/software/amazon/kinesis/processor/amazon-kinesis-client.java> 를 참조하십시오. FormerStreamsLeasesDeletionStrategy

- [ConfigsBuilder](#) KCL 소비자 애플리케이션을 구축할 때 사용할 모든 KCL 2.x 구성 설정을 지정하는 데 사용할 수 있는 애플리케이션 전반의 클래스입니다. ConfigsBuilder 클래스는 이제 인터페이스를 지원합니다. MultistreamTracker ConfigsBuilder 둘 중 하나를 데이터 스트림의 이름으로 초기화하여 다음 데이터 스트림의 레코드를 소비할 수 있습니다.

```

/**
 * Constructor to initialize ConfigsBuilder with StreamName
 * @param streamName
 * @param applicationName
 * @param kinesisClient
 * @param dynamoDBClient
 * @param cloudWatchClient
 * @param workerIdentifier
 * @param shardRecordProcessorFactory
 */
public ConfigsBuilder(@NonNull String streamName, @NonNull String
applicationName,
                    @NonNull KinesisAsyncClient kinesisClient, @NonNull DynamoDbAsyncClient
dynamoDBClient,
                    @NonNull CloudWatchAsyncClient cloudWatchClient, @NonNull String
workerIdentifier,
                    @NonNull ShardRecordProcessorFactory shardRecordProcessorFactory) {
    this.appStreamTracker = Either.right(streamName);
    this.applicationName = applicationName;
    this.kinesisClient = kinesisClient;
    this.dynamoDBClient = dynamoDBClient;
    this.cloudWatchClient = cloudWatchClient;
    this.workerIdentifier = workerIdentifier;
    this.shardRecordProcessorFactory = shardRecordProcessorFactory;
}

```

또는 여러 스트림을 동시에 처리하는 KCL 소비자 애플리케이션을 구현하려는 MultiStreamTracker 경우 를 사용하여 초기화할 ConfigsBuilder 수 있습니다.

```

* Constructor to initialize ConfigsBuilder with MultiStreamTracker
 * @param multiStreamTracker
 * @param applicationName
 * @param kinesisClient
 * @param dynamoDBClient
 * @param cloudWatchClient
 * @param workerIdentifier
 * @param shardRecordProcessorFactory
 */
public ConfigsBuilder(@NonNull MultiStreamTracker multiStreamTracker, @NonNull
String applicationName,

```

```

        @NonNull KinesisAsyncClient kinesisClient, @NonNull DynamoDbAsyncClient
dynamoDBClient,
        @NonNull CloudWatchAsyncClient cloudWatchClient, @NonNull String
workerIdentifier,
        @NonNull ShardRecordProcessorFactory shardRecordProcessorFactory) {
    this.appStreamTracker = Either.left(multiStreamTracker);
    this.applicationName = applicationName;
    this.kinesisClient = kinesisClient;
    this.dynamoDBClient = dynamoDBClient;
    this.cloudWatchClient = cloudWatchClient;
    this.workerIdentifier = workerIdentifier;
    this.shardRecordProcessorFactory = shardRecordProcessorFactory;
}

```

- KCL 소비자 애플리케이션에 대해 멀티스트림 지원이 구현됨에 따라 이제 애플리케이션 리스 테이블의 각 행에는 이 애플리케이션이 처리하는 여러 데이터 스트림의 샤드 ID와 스트림 이름이 포함됩니다.
- KCL 소비자 애플리케이션에 대한 멀티스트림 지원이 구현되면 leaseKey는 account-id:StreamName:streamCreationTimestamp:ShardId 구조를 취합니다. 예를 들어 111111111:multiStreamTest-1:12345:shardId-000000000336입니다.

### Important

기존 KCL 소비자 애플리케이션이 하나의 데이터 스트림만 처리하도록 구성된 경우 leaseKey(리스 테이블의 해시 키)는 샤드 ID입니다. 여러 데이터 스트림을 처리하도록 이 기존 KCL 소비자 애플리케이션을 재구성하면 리스 테이블이 손상됩니다. 멀티스트림 지원을 사용할 경우 leaseKey 구조는 account-id:StreamName:StreamCreationTimestamp:ShardId와 같아야 하기 때문입니다.

## Kinesis 클라이언트 라이브러리를 AWS Glue 스키마 레지스트리와 함께 사용

Kinesis 데이터 스트림을 AWS Glue 스키마 레지스트리와 통합할 수 있습니다. AWS Glue 스키마 레지스트리를 사용하면 스키마를 중앙에서 검색, 제어 및 발전시키는 동시에 생성된 데이터가 등록된 스키마에 의해 지속적으로 검증되도록 할 수 있습니다. 스키마는 데이터 레코드의 구조와 포맷을 정의합니다. 스키마는 신뢰할 수 있는 데이터 게시, 소비 또는 저장을 위한 버전 지정 사양입니다. AWS Glue Schema Registry를 사용하면 스트리밍 애플리케이션 내에서 end-to-end 데이터 품질 및 데이터 거버

년스를 개선할 수 있습니다. 자세한 내용은 [AWS Glue Schema Registry](#)를 참조하세요. 이 통합을 설정하는 방법 중 하나는 Java에서 KCL을 사용하는 것입니다.

### ⚠ Important

현재 Kinesis Data Streams 및 AWS Glue 스키마 레지스트리 통합은 Java로 구현된 KCL 2.3 컨슈머를 사용하는 Kinesis 데이터 스트림에서만 지원됩니다. 다국어 지원은 제공되지 않습니다. KCL 1.0 소비자는 지원되지 않습니다. KCL 2.3 이전의 KCL 2.x 소비자는 지원되지 않습니다.

KCL을 사용하여 Kinesis Data Streams와 스키마 레지스트리의 통합을 설정하는 방법에 대한 자세한 지침은 사용 사례: [Amazon Kinesis 데이터 스트림과 Glue 스키마 레지스트리 통합의 “KPL/KCL 라이브러리를 사용한 데이터 상호 작용”](#) 섹션을 참조하십시오. AWS

## 공유 처리량으로 사용자 지정 소비자 개발

Kinesis Data Streams로부터 데이터를 수신할 때 전용 처리량이 필요하지 않은 경우 그리고 200ms 미만의 읽기 전파 지연이 필요하지 않은 경우에는 다음 섹션에서 설명한 대로 소비자 애플리케이션을 구축하면 됩니다.

### 주제

- [KCL을 사용하여 공유 처리량으로 사용자 지정 소비자 개발](#)
- [AWS SDK for Java를 사용하여 공유 처리량으로 사용자 지정 소비자 개발](#)

전용 처리량으로 Kinesis 데이터 스트림에서 레코드를 수신할 수 있는 소비자를 구축하는 방법에 대한 자세한 내용은 [전용 처리량으로 사용자 지정 소비자 개발\(향상된 팬아웃\)](#) 섹션을 참조하세요.

## KCL을 사용하여 공유 처리량으로 사용자 지정 소비자 개발

공유 처리량으로 사용자 지정 소비자 애플리케이션을 개발하는 방법 중 하나는 Kinesis Client Library(KCL)를 사용하는 것입니다.

### 주제

- [KCL 1.x 소비자 개발](#)
- [KCL 2.x 소비자 개발](#)

## KCL 1.x 소비자 개발

Kinesis Client Library(KCL)를 사용하여 Amazon Kinesis Data Streams의 소비자 애플리케이션을 개발할 수 있습니다. KCL에 대한 자세한 내용은 [Kinesis Client Library란?](#) 단원을 참조하십시오.

### 내용

- [Java로 Kinesis Client Library 소비자 개발](#)
- [Node.js로 Kinesis Client Library 소비자 개발](#)
- [.NET으로 Kinesis Client Library 소비자 개발](#)
- [Python으로 Kinesis Client Library 소비자 개발](#)
- [Ruby로 Kinesis Client Library 소비자 개발](#)

### Java로 Kinesis Client Library 소비자 개발

Kinesis Client Library(KCL)를 사용하여 Kinesis 데이터 스트림의 데이터를 처리하는 애플리케이션을 빌드합니다. Kinesis Client Library는 여러 언어로 제공됩니다. 이 주제에서는 Java에 대해 설명합니다. Javadoc 레퍼런스를 보려면 클래스에 대한 [AWSJavadoc](#) 항목을 참조하십시오. AmazonKinesisClient

에서 GitHub Java KCL을 다운로드하려면 [Kinesis 클라이언트 라이브러리 \(Java\)](#) 로 이동하십시오. Apache Maven에서 Java KCL을 찾으려면 [KCL 검색 결과](#) 페이지로 이동하세요. 에서 Java KCL 소비자 애플리케이션용 샘플 코드를 다운로드하려면 에서 GitHub [Java용 KCL 샘플](#) 프로젝트 페이지로 이동하십시오. GitHub

샘플 애플리케이션에 [Apache Commons Logging](#)이 사용됩니다. configure 파일에 정의된 정적 AmazonKinesisApplicationSample.java 메서드에서 로깅 구성을 변경할 수 있습니다. Log4j 및 AWS Java 애플리케이션과 함께 Apache Commons Logging을 사용하는 방법에 대한 자세한 내용은 AWS SDK for Java Developer Guide의 <https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/java-dg-logging.html> Logging with Log4j를 참조하세요.

Java로 KCL 소비자 애플리케이션을 구현할 때 다음 작업을 완료해야 합니다.

### Tasks

- [I 메서드 구현 RecordProcessor](#)
- [I RecordProcessor 인터페이스용 클래스 팩토리 구현](#)
- [작업자 생성](#)
- [구성 속성 수정](#)
- [레코드 프로세서 인터페이스 버전 2로 마이그레이션](#)

## I 메서드 구현 RecordProcessor

KCL은 현재 두 버전의 IRecordProcessor 인터페이스를 지원합니다. KCL의 첫 번째 버전에 제공되는 원래 인터페이스와 KCL 버전 1.5.0부터 사용할 수 있는 버전 2입니다. 두 인터페이스 모두 완벽하게 지원되며, 특정 시나리오 요구 사항에 따라 선택할 수 있습니다. 모든 차이점을 보려면 로컬로 빌드된 Javadoc 또는 소스 코드를 참조하십시오. 다음 단원에서는 시작에 필요한 최소한의 구현을 설명합니다.

### I RecordProcessor 버전

- [원래의 인터페이스\(버전 1\)](#)
- [업데이트된 인터페이스\(버전 2\)](#)

#### 원래의 인터페이스(버전 1)

원래의 IRecordProcessor 인터페이스(package com.amazonaws.services.kinesis.clientlibrary.interfaces)는 소비자가 구현할 다음과 같은 레코드 프로세서 메서드를 노출합니다. 이 샘플에서는 시작점으로 사용할 수 있는 구현을 제공합니다(AmazonKinesisApplicationSampleRecordProcessor.java 참조).

```
public void initialize(String shardId)
public void processRecords(List<Record> records, IRecordProcessorCheckpoint
    checkpoint)
public void shutdown(IRecordProcessorCheckpoint checkpoint, ShutdownReason reason)
```

#### 초기화

KCL은 레코드 프로세서가 인스턴스화될 때 특정 샤드 ID를 파라미터로 전달하여 initialize 메서드를 직접적으로 호출합니다. 이 레코드 프로세서는 해당 샤드만 처리하고 일반적으로 반대의 경우도 마찬가지입니다. 이 샤드는 해당 레코드 프로세서로만 처리됩니다. 하지만 소비자는 데이터 레코드가 두 번 이상 처리될 가능성을 고려해야 합니다. Kinesis Data Streams에서는 소비자의 워커가 샤드의 모든 데이터 레코드를 적어도 한 번은 처리한다는 적어도 한 번 의미론이 통용됩니다. 둘 이상의 작업자가 특정 샤드를 처리할 수 있는 경우에 대한 자세한 내용은 [리샤딩, 크기 조정 및 병렬 처리](#)를 참조하십시오.

```
public void initialize(String shardId)
```

#### processRecords

KCL은 `processRecords` 메서드에 지정된 샤드의 데이터 레코드 목록을 전달하여 `initialize(shardId)` 메서드를 직접적으로 호출합니다. 레코드 프로세서는 소비자의 의미론에 따라 이 레코드의 데이터를 처리합니다. 예를 들어, 워커가 데이터를 전환한 후 그 결과를 Amazon Simple Storage Service(S3) 버킷에 저장할 수 있습니다.

```
public void processRecords(List<Record> records, IRecordProcessorCheckpoint
    checkpointer)
```

데이터 자체뿐 아니라 시퀀스 번호와 파티션 키도 데이터 레코드에 포함됩니다. 작업자가 데이터를 처리할 때 이 값을 사용할 수 있습니다. 예를 들어, 작업자는 파티션 키의 값을 기반으로 데이터를 저장할 S3 버킷을 선택할 수 있습니다. `Record` 클래스는 레코드의 데이터, 시퀀스 번호 및 파티션 키에 대한 액세스를 제공하는 다음 메서드를 노출합니다.

```
record.getData()
record.getSequenceNumber()
record.getPartitionKey()
```

이 샘플의 프라이빗 메서드 `processRecordsWithRetries`에는 작업자가 레코드의 데이터, 시퀀스 번호 및 파티션 키에 액세스하는 방법을 보여주는 코드가 있습니다.

Kinesis Data Streams는 샤드에서 이미 처리된 레코드를 추적하도록 레코드 프로세서에 요구합니다. KCL은 체크포인트(`IRecordProcessorCheckpoint`)를 `processRecords`에 전달하여 이 추적을 처리합니다. 레코드 프로세서는 해당 인터페이스에서 `checkpoint` 메서드를 직접적으로 호출하여 샤드의 레코드 처리가 얼마나 진행되었는지 KCL에 알려줍니다. 워커가 실패할 경우 KCL은 이 정보를 사용하여 마지막으로 처리된 레코드에서 샤드 처리를 다시 시작합니다.

분할 또는 병합 작업의 경우 소스 샤드의 프로세서가 `checkpoint`를 직접적으로 호출하여 소스 샤드의 모든 처리가 완료되었다고 표시할 때까지 KCL은 새 샤드의 처리를 시작하지 않습니다.

파라미터를 전달하지 않으면 KCL은 `checkpoint`에 대한 호출이 레코드 프로세서에 전달된 마지막 레코드까지 모두 처리되었다는 의미로 간주합니다. 따라서 레코드 프로세서는 전달된 목록에 있는 모든 레코드를 반드시 처리한 후에 `checkpoint`를 호출해야 합니다. 레코드 프로세서는 `checkpoint`를 호출할 때마다 `processRecords`를 호출할 필요가 없습니다. 예를 들어, 프로세서는 `checkpoint`를 세 번째 호출할 때마다 `processRecords`를 호출할 수 있습니다. 선택적으로 레코드의 정확한 시퀀스 번호를 `checkpoint`의 파라미터로 지정할 수도 있습니다. 이 경우 KCL은 모든 레코드가 해당 레코드까지만 처리되었다고 간주합니다.

이 샘플에서는 프라이빗 메서드 `checkpoint`가 적절한 예외 처리 및 재시도 로직을 사용하여 `IRecordProcessorCheckpoint.checkpoint`를 호출하는 방법을 보여줍니다.



KCL은 `processRecords`를 사용하여 데이터 레코드를 처리할 때 발생하는 모든 예외를 처리합니다. `processRecords`에서 예외가 발생하면 KCL은 예외 이전에 전달된 데이터 레코드를 건너뛵니다. 이러한 레코드는 예외가 발생한 프로세서 또는 소비자의 다른 레코드 프로세서로 다시 전송되지 않습니다.

## shutdown

처리가 종료될 때(종료 이유가 `TERMINATE`) 또는 워커가 더 이상 응답하지 않을 때(종료 이유가 `ZOMBIE`) KCL은 `shutdown` 메서드를 직접적으로 호출합니다.

```
public void shutdown(IRecordProcessorCheckpointter checkpointer, ShutdownReason reason)
```

샤드 분할이나 병합 또는 스트림 삭제로 인해 레코드 프로세서가 샤드에서 추가 레코드를 수신하지 않으면 처리가 종료됩니다.

또한 KCL은 `IRecordProcessorCheckpointter` 인터페이스를 `shutdown`에 전달합니다. 종료 이유가 `TERMINATE`이면 레코드 프로세서가 데이터 레코드 처리를 완료하고 이 인터페이스의 `checkpoint` 메서드를 호출해야 합니다.

## 업데이트된 인터페이스(버전 2)

업데이트된 `IRecordProcessor` 인터페이스(`package com.amazonaws.services.kinesis.clientlibrary.interfaces.v2`)는 소비자가 구현할 다음과 같은 레코드 프로세서 메서드를 노출합니다.

```
void initialize(InitializationInput initializationInput)
void processRecords(ProcessRecordsInput processRecordsInput)
void shutdown(ShutdownInput shutdownInput)
```

원래 인터페이스 버전의 모든 인수는 컨테이너 객체에서 `get` 메서드를 통해 액세스할 수 있습니다. 예를 들어, `processRecords()`를 사용하여 `processRecordsInput.getRecords()`의 레코드 목록을 검색할 수 있습니다.

이 인터페이스 버전 2에서(KCL 1.5.0 이상) 원래의 인터페이스가 제공하는 입력 외에도 다음과 같은 새 입력을 사용할 수 있습니다.

## 시작 시퀀스 번호

`InitializationInput` 작업에 전달된 `initialize()` 객체에서 레코드가 레코드 프로세서 인스턴스에 제공될 시작 시퀀스 번호이며, 전에 같은 샤드를 처리하는 레코드 프로세서가 마지막으로 검사한 시퀀스 번호입니다. 애플리케이션에 이 정보가 필요할 경우 제공됩니다.

## 보류 중인 체크포인트 시퀀스 번호

`initialize()` 작업에 전달된 `InitializationInput` 객체에서 이전 레코드 프로세서 인스턴스가 중단되기 전에 커밋되지 못한 보류 중인 체크포인트 시퀀스 번호(있는 경우)입니다.

## IRecordProcessor 인터페이스용 클래스 팩토리 구현

레코드 프로세서 메서드를 구현하는 클래스 팩토리도 구현해야 합니다. 소비자가 작업자를 인스턴스화할 때 이 팩토리에 참조를 전달합니다.

이 샘플은 원래의 레코드 프로세서 인터페이스를 사용하여

`AmazonKinesisApplicationSampleRecordProcessorFactory.java` 파일에서 팩토리 클래스를 구현합니다. 클래스 팩토리에서 레코드 프로세서 버전 2를 만들려면 패키지 이름 `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2`를 사용하십시오.

```
public class SampleRecordProcessorFactory implements IRecordProcessorFactory {
    /**
     * Constructor.
     */
    public SampleRecordProcessorFactory() {
        super();
    }
    /**
     * {@inheritDoc}
     */
    @Override
    public IRecordProcessor createProcessor() {
        return new SampleRecordProcessor();
    }
}
```

## 작업자 생성

[I 메서드 구현 RecordProcessor](#) 에서 설명한 대로 두 가지 KCL 레코드 프로세서 인터페이스 버전 중에서 선택할 수 있으며 이 선택은 워커 생성 방법에 영향을 줍니다. 원래의 레코드 프로세서 인터페이스는 다음 코드 구조를 사용하여 작업자를 생성합니다.

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = new Worker(recordProcessorFactory, config);
```

레코드 프로세서 인터페이스 버전 2를 통해 인수의 순서와 사용할 생성자를 고민할 필요 없이 `Worker.Builder`를 사용하여 작업자를 만들 수 있습니다. 업데이트된 레코드 프로세서 인터페이스는 다음 코드 구조를 사용하여 작업자를 생성합니다.

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = new Worker.Builder()
    .recordProcessorFactory(recordProcessorFactory)
    .config(config)
    .build();
```

## 구성 속성 수정

이 샘플은 구성 속성의 기본값을 제공합니다. 그러면 작업자의 이 구성 데이터가 `KinesisClientLibConfiguration` 객체에 통합됩니다. `IRecordProcessor`의 클래스 팩토리에 대한 참조와 이 객체는 작업자를 인스턴스화하는 호출에서 전달됩니다. Java 속성 파일을 사용하여 이 속성을 사용자의 값으로 재정의할 수 있습니다(`AmazonKinesisApplicationSample.java` 참조).

## 애플리케이션 이름

KCL에는 애플리케이션 및 같은 리전의 Amazon DynamoDB 테이블에서 고유한 애플리케이션 이름이 필요합니다. 다음과 같이 애플리케이션 이름 구성 값이 사용됩니다.

- 이 애플리케이션 이름과 관련된 모든 작업자는 동일한 스트림에서 함께 작업한다고 간주됩니다. 이 작업자는 여러 인스턴스에 분산되어 있을 수 있습니다. 동일한 애플리케이션 코드의 추가 인스턴스를 다른 애플리케이션 이름으로 실행하는 경우 KCL은 두 번째 인스턴스를 동일한 스트림에서 작동하는 완전히 별개의 애플리케이션으로 취급합니다.
- KCL은 애플리케이션 이름이 있는 DynamoDB 테이블을 생성하고 테이블을 사용하여 애플리케이션의 상태 정보(예: 체크포인트 및 워커와 샤드의 매핑)를 보관합니다. 각각의 애플리케이션에는 자체 DynamoDB 테이블이 있습니다. 자세한 설명은 [리스 테이블을 사용하여 KCL 소비자 애플리케이션에서 처리한 샤드 추적](#) 섹션을 참조하세요.

## 자격 증명 설정

기본 보안 인증 공급자 체인의 보안 인증 공급자 중 하나가 AWS 보안 인증을 사용할 수 있도록 해야 합니다. 예를 들어, EC2 인스턴스에서 소비자를 실행하는 경우 IAM 역할로 인스턴스를 시작하는 것이 좋습니다. 이 IAM 역할과 연결된 권한을 반영하는 AWS 보안 인증은 인스턴스 메타데이터를 통해 인스턴스의 애플리케이션에 제공됩니다. 이것이 EC2 인스턴스에서 실행되는 소비자의 자격 증명을 관리하는 가장 안전한 방법입니다.

먼저 샘플 애플리케이션이 인스턴스 메타데이터에서 IAM 보안 인증 검색을 시도합니다.

```
credentialsProvider = new InstanceProfileCredentialsProvider();
```

샘플 애플리케이션이 인스턴스 메타데이터에서 자격 증명을 가져오지 못하면 속성 파일에서 자격 증명 검색을 시도합니다.

```
credentialsProvider = new ClasspathPropertiesFileCredentialsProvider();
```

인스턴스 메타데이터에 대한 자세한 내용은 Amazon EC2 - Linux 인스턴스용 사용 설명서의 [인스턴스 메타데이터](#)를 참조하세요.

### 여러 인스턴스에 작업자 ID 사용

샘플 초기화 코드는 로컬 컴퓨터 이름을 사용하고 다음 코드 조각과 같이 전역적으로 고유한 식별자를 추가하여 작업자의 ID인 workerId를 만듭니다. 이 방법은 단일 컴퓨터에서 소비자 애플리케이션의 여러 인스턴스가 실행되는 시나리오를 지원합니다.

```
String workerId = InetAddress.getLocalHost().getCanonicalHostName() + ":" +
    UUID.randomUUID();
```

### 레코드 프로세서 인터페이스 버전 2로 마이그레이션

위에서 설명한 단계 외에도 원래의 인터페이스를 사용하는 코드를 마이그레이션하려면 다음 단계가 필요합니다.

1. 버전 2 레코드 프로세서 인터페이스를 가져오도록 레코드 프로세서 클래스를 변경합니다.

```
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
```

2. 컨테이너 객체에서 get 메서드를 사용하도록 참조를 변경합니다. 예를 들어, shutdown() 작업에서 "checkpointer"를 "shutdownInput.getCheckpointer()"로 변경합니다.
3. 버전 2 레코드 프로세서 팩토리 인터페이스를 가져오도록 레코드 프로세서 팩토리 클래스를 변경합니다.

```
import
    com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
```

4. Worker.Builder를 사용하도록 작업자의 구성을 변경합니다. 예:

```
final Worker worker = new Worker.Builder()
    .recordProcessorFactory(recordProcessorFactory)
    .config(config)
    .build();
```

## Node.js로 Kinesis Client Library 소비자 개발

Kinesis Client Library(KCL)를 사용하여 Kinesis 데이터 스트림의 데이터를 처리하는 애플리케이션을 빌드합니다. Kinesis Client Library는 여러 언어로 제공됩니다. 이 주제에서는 Node.js에 대해 설명합니다.

KCL은 Java 라이브러리입니다. Java 이외의 언어에 대한 지원은 라는 다국어 인터페이스를 사용하여 제공됩니다. MultiLangDaemon 이 데몬은 Java 기반이며, Java 이외의 KCL 언어를 사용하는 경우 배경에서 실행됩니다. 따라서 Node.js 용 KCL을 설치하고 소비자 앱 전체를 Node.js 형식으로 작성하더라도 시스템에 Java를 설치해야 하는 이유는 다음과 같습니다. MultiLangDaemon 또한 사용 사례에 맞게 사용자 지정해야 할 몇 가지 기본 설정 (예: 연결 AWS 지역) 도 있습니다. MultiLangDaemon MultiLangDaemon GitHubon에 대한 자세한 내용은 [KCL MultiLangDaemon 프로젝트 페이지를 참조하세요](#).

에서 GitHub Node.js KCL을 다운로드하려면 [Kinesis 클라이언트 라이브러리 \(Node.js\)](#) 로 이동하십시오.

### 샘플 코드 다운로드

Node.js KCL에 두 가지 코드 샘플을 사용할 수 있습니다.

- [basic-sample](#)

Node.js로 KCL 소비자 애플리케이션을 빌드하기 위한 기초를 설명하기 위해 다음 섹션에서 사용됩니다.

- [click-stream-sample](#)

기본 샘플 코드에 익숙해진 후에 실제 시나리오를 사용할 수 있는 약간 높은 수준의 코드입니다. 여기서는 이 샘플을 설명하지 않으며 자세한 내용은 README 파일에 있습니다.

Node.js로 KCL 소비자 애플리케이션을 구현할 때 다음 작업을 완료해야 합니다.

### Tasks

- [레코드 프로세서 구현](#)
- [구성 속성 수정](#)

## 레코드 프로세서 구현

Node.js용 KCL을 사용하는 가능한 한 가장 단순한 소비자는 `initialize`, `processRecords` 및 `shutdown` 함수를 차례로 포함하는 `recordProcessor` 함수를 구현해야 합니다. 이 샘플에서는 시작점으로 사용할 수 있는 구현을 제공합니다(`sample_kcl_app.js` 참조).

```
function recordProcessor() {
  // return an object that implements initialize, processRecords and shutdown
  functions.}
```

### 초기화

레코드 프로세서가 시작되면 KCL은 `initialize` 함수를 직접적으로 호출합니다. 이 레코드 프로세서는 `initializeInput.shardId`로 전달된 샤드 ID만 처리하고 일반적으로 반대의 경우도 마찬가지입니다. 이 샤드는 해당 레코드 프로세서로만 처리됩니다. 하지만 소비자는 데이터 레코드가 두 번 이상 처리될 가능성을 고려해야 합니다. Kinesis Data Streams에서는 소비자의 워커가 샤드의 모든 데이터 레코드를 적어도 한 번은 처리한다는 적어도 한 번 의미론이 통용되기 때문입니다. 둘 이상의 작업자가 특정 샤드를 처리할 수 있는 경우에 대한 자세한 내용은 [리샤딩, 크기 조정 및 병렬 처리](#)를 참조하십시오.

```
initialize: function(initializeInput, completeCallback)
```

### processRecords

KCL은 지정된 샤드에서 `initialize` 함수까지 데이터 레코드 목록을 포함하는 입력으로 이 함수를 직접적으로 호출합니다. 구현하는 레코드 프로세서가 소비자의 의미론에 따라 이 레코드의 데이터를 처리합니다. 예를 들어, 워커가 데이터를 전환한 후 그 결과를 Amazon Simple Storage Service(S3) 버킷에 저장할 수 있습니다.

```
processRecords: function(processRecordsInput, completeCallback)
```

데이터 자체뿐 아니라 작업자가 데이터를 처리할 때 사용할 수 있는 시퀀스 번호와 파티션 키도 데이터 레코드에 포함됩니다. 예를 들어, 작업자는 파티션 키의 값을 기반으로 데이터를 저장할 S3 버킷을 선택할 수 있습니다. `record` 딕셔너리가 레코드의 데이터, 시퀀스 번호 및 파티션 키에 액세스하기 위해 다음 키-값 페어를 노출합니다.

```
record.data
record.sequenceNumber
record.partitionKey
```

데이터가 Base64로 인코딩됩니다.

기본 샘플에서 `processRecords` 함수에는 작업자가 레코드의 데이터, 시퀀스 번호 및 파티션 키에 액세스하는 방법을 보여주는 코드가 있습니다.

Kinesis Data Streams는 샤드에서 이미 처리된 레코드를 추적하도록 레코드 프로세서에 요구합니다. KCL은 `processRecordsInput.checkpointer`로 전달된 `checkpointer` 객체를 통해 이 추적을 처리합니다. 레코드 프로세서는 `checkpointer.checkpoint` 함수를 직접적으로 호출하여 샤드의 레코드 처리가 얼마나 진행되었는지 KCL에 알려줍니다. 워커가 실패할 경우 KCL은 샤드 처리를 다시 시작할 때 마지막으로 처리된 레코드에서 계속되도록 이 정보를 사용합니다.

분할 또는 병합 작업의 경우 소스 샤드의 프로세서가 `checkpoint`를 직접적으로 호출하여 소스 샤드의 모든 처리가 완료되었다고 표시할 때까지 KCL은 새 샤드의 처리를 시작하지 않습니다.

`checkpoint` 함수에 시퀀스 번호를 전달하지 않으면 KCL은 `checkpoint`에 대한 호출이 레코드 프로세서에 전달된 마지막 레코드까지 모두 처리되었다는 의미로 간주합니다. 따라서 레코드 프로세서는 전달된 목록의 모든 레코드를 처리한 후에만 `checkpoint`를 직접적으로 호출해야 합니다. 레코드 프로세서는 `checkpoint`를 호출할 때마다 `processRecords`를 호출할 필요가 없습니다. 예를 들어, 프로세서는 세 번째 호출마다 `checkpoint` 또는 구현된 사용자 지정 검증/확인 서비스와 같은 레코드 프로세서 외부의 이벤트를 호출할 수 있습니다.

선택적으로 레코드의 정확한 시퀀스 번호를 `checkpoint`의 파라미터로 지정할 수도 있습니다. 이 경우 KCL은 모든 레코드가 해당 레코드까지만 처리되었다고 간주합니다.

기본 샘플 애플리케이션은 `checkpointer.checkpoint` 함수의 가장 단순하고 가능한 호출을 보여줍니다. 소비자에 필요한 다른 검사 로직을 함수의 이 지점에 추가할 수 있습니다.

## shutdown

처리가 종료될 때(`shutdownInput.reason`이 `TERMINATE`) 또는 워커가 더 이상 응답하지 않을 때(`shutdownInput.reason`이 `ZOMBIE`) KCL은 `shutdown` 함수를 직접적으로 호출합니다.

```
shutdown: function(shutdownInput, completeCallback)
```

샤드 분할이나 병합 또는 스트림 삭제로 인해 레코드 프로세서가 샤드에서 추가 레코드를 수신하지 않으면 처리가 종료됩니다.

또한 KCL은 `shutdownInput.checkpointer` 객체를 `shutdown`에 전달합니다. 종료 이유가 `TERMINATE`이면 레코드 프로세서가 모든 데이터 레코드 처리를 완료했는지 확인하고 이 인터페이스의 `checkpoint` 함수를 호출해야 합니다.

## 구성 속성 수정

이 샘플은 구성 속성의 기본값을 제공합니다. 이 속성을 사용자의 값으로 재정의할 수 있습니다(기본 샘플의 `sample.properties` 참조).

## 애플리케이션 이름

KCL에는 애플리케이션 및 같은 리전의 Amazon DynamoDB 테이블에서 고유한 애플리케이션이 필요합니다. 다음과 같이 애플리케이션 이름 구성 값이 사용됩니다.

- 이 애플리케이션 이름과 관련된 모든 작업자는 동일한 스트림에서 함께 작업한다고 간주됩니다. 이 작업자는 여러 인스턴스에 분산되어 있을 수 있습니다. 동일한 애플리케이션 코드의 추가 인스턴스를 다른 애플리케이션 이름으로 실행하는 경우 KCL은 두 번째 인스턴스를 동일한 스트림에서 작동하는 완전히 별개의 애플리케이션으로 취급합니다.
- KCL은 애플리케이션 이름이 있는 DynamoDB 테이블을 생성하고 테이블을 사용하여 애플리케이션의 상태 정보(예: 체크포인트 및 워커와 샤드의 매핑)를 보관합니다. 각각의 애플리케이션에는 자체 DynamoDB 테이블이 있습니다. 자세한 설명은 [리스 테이블을 사용하여 KCL 소비자 애플리케이션에서 처리한 샤드 추적](#) 섹션을 참조하세요.

## 자격 증명 설정

기본 보안 인증 공급자 체인의 보안 인증 공급자 중 하나가 AWS 보안 인증을 사용할 수 있도록 해야 합니다. `AWSCredentialsProvider` 속성을 사용하여 자격 증명 공급자를 설정할 수 있습니다. `sample.properties` 파일에서 [기본 자격 증명 공급자 체인](#)의 자격 증명 공급자 중 하나에 자격 증명을 사용할 수 있도록 해야 합니다. Amazon EC2 인스턴스에서 소비자를 실행하는 경우 IAM 역할로 인스턴스를 구성하는 것이 좋습니다. 이 IAM 역할과 연결된 권한을 반영하는 AWS 보안 인증은 인스턴스 메타데이터를 통해 인스턴스의 애플리케이션에 제공됩니다. 이것이 EC2 인스턴스에서 실행되는 소비자 애플리케이션의 자격 증명을 관리하는 가장 안전한 방법입니다.

다음 예제는 `sample_kcl_app.js`에 제공된 레코드 프로세서를 사용하여 `kclnodejssample`이라는 Kinesis 데이터 스트림을 처리하도록 KCL을 구성합니다.

```
# The Node.js executable script
executableName = node sample_kcl_app.js
# The name of an Amazon Kinesis stream to process
```



```
streamName = kclnodejssample
# Unique KCL application name
applicationName = kclnodejssample
# Use default AWS credentials provider chain
AWSCredentialsProvider = DefaultAWSCredentialsProviderChain
# Read from the beginning of the stream
initialPositionInStream = TRIM_HORIZON
```

## .NET으로 Kinesis Client Library 소비자 개발

Kinesis Client Library(KCL)를 사용하여 Kinesis 데이터 스트림의 데이터를 처리하는 애플리케이션을 빌드합니다. Kinesis Client Library는 여러 언어로 제공됩니다. 이 주제에서는 .NET에 대해 설명합니다.

KCL은 Java 라이브러리입니다. Java 이외의 언어에 대한 지원은 라는 다국어 인터페이스를 사용하여 제공됩니다. MultiLangDaemon 이 데몬은 Java 기반이며, Java 이외의 KCL 언어를 사용하는 경우 배경에서 실행됩니다. 따라서 .NET용 KCL을 설치하고 소비자 앱을 전체.NET으로 작성하는 경우에도 다음과 같은 이유로 시스템에 Java를 설치해야 합니다. MultiLangDaemon 또한 사용 사례에 맞게 사용자 지정해야 할 몇 가지 기본 설정 (예: 연결 AWS 지역) 도 있습니다. MultiLangDaemon MultiLangDaemon GitHub에 대한 자세한 내용은 [KCL MultiLangDaemon 프로젝트 페이지를 참조하세요](#).

.NET KCL을 다운로드하려면 [Kinesis 클라이언트 라이브러리 \(.NET\)](#) 로 이동하십시오. GitHub .NET KCL 소비자 애플리케이션용 샘플 코드를 다운로드하려면 의 [.NET용 KCL](#) 샘플 소비자 프로젝트 페이지로 이동하십시오. GitHub

.NET으로 KCL 소비자 애플리케이션을 구현할 때 다음 작업을 완료해야 합니다.

### Tasks

- [I 클래스 메서드 구현 RecordProcessor](#)
- [구성 속성 수정](#)

### I 클래스 메서드 구현 RecordProcessor

소비자는 IRecordProcessor를 위해 다음의 메서드를 구현해야 합니다. 이 샘플 소비자는 시작점으로 사용할 수 있는 구현을 제공합니다(SampleRecordProcessor의 SampleConsumer/AmazonKinesisSampleConsumer.cs 클래스 참조).

```
public void Initialize(InitializationInput input)
public void ProcessRecords(ProcessRecordsInput input)
```

```
public void Shutdown(ShutdownInput input)
```

## 초기화

KCL은 레코드 프로세서가 인스턴스화될 때 `input` 파라미터(`input.ShardId`)에서 특정 샤드 ID를 전달하여 이 메서드를 직접적으로 호출합니다. 이 레코드 프로세서는 해당 샤드만 처리하고 일반적으로 반대의 경우도 마찬가지입니다. 이 샤드는 해당 레코드 프로세서로만 처리됩니다. 하지만 소비자는 데이터 레코드가 두 번 이상 처리될 가능성을 고려해야 합니다. Kinesis Data Streams에서는 소비자의 워커가 샤드의 모든 데이터 레코드를 적어도 한 번은 처리한다는 적어도 한 번 의미론이 통용되기 때문입니다. 둘 이상의 작업자가 특정 샤드를 처리할 수 있는 경우에 대한 자세한 내용은 [리샤딩, 크기 조정 및 병렬 처리](#)를 참조하십시오.

```
public void Initialize(InitializationInput input)
```

## ProcessRecords

KCL은 `Initialize` 메서드를 통해 지정된 샤드에서 `input` 파라미터(`input.Records`)의 데이터 레코드 목록을 전달하여 이 메서드를 직접적으로 호출합니다. 구현하는 레코드 프로세서가 소비자의 의미론에 따라 이 레코드의 데이터를 처리합니다. 예를 들어, 워커가 데이터를 전환한 후 그 결과를 Amazon Simple Storage Service(S3) 버킷에 저장할 수 있습니다.

```
public void ProcessRecords(ProcessRecordsInput input)
```

데이터 자체뿐 아니라 시퀀스 번호와 파티션 키도 데이터 레코드에 포함됩니다. 작업자가 데이터를 처리할 때 이 값을 사용할 수 있습니다. 예를 들어, 작업자는 파티션 키의 값을 기반으로 데이터를 저장할 S3 버킷을 선택할 수 있습니다. `Record` 클래스는 레코드의 데이터, 시퀀스 번호 및 파티션 키에 액세스하기 위해 다음 항목을 노출합니다.

```
byte[] Record.Data
string Record.SequenceNumber
string Record.PartitionKey
```

이 샘플의 메서드 `ProcessRecordsWithRetries`에는 작업자가 레코드의 데이터, 시퀀스 번호 및 파티션 키에 액세스하는 방법을 보여주는 코드가 있습니다.

Kinesis Data Streams는 샤드에서 이미 처리된 레코드를 추적하도록 레코드 프로세서에 요구합니다. KCL은 `Checkpointter` 객체를 `ProcessRecords(input.Checkpointer)`에 전달하여 이 추적을 처리합니다. 레코드 프로세서는 `Checkpointter.Checkpoint` 메서드를 직접적으로 호출하여 샤드의

레코드 처리가 얼마나 진행되었는지를 KCL에 알려줍니다. 워커가 실패할 경우 KCL은 이 정보를 사용하여 마지막으로 처리된 레코드에서 샤드 처리를 다시 시작합니다.

분할 또는 병합 작업의 경우 소스 샤드의 프로세서가 `Checkpoint`를 직접적으로 호출하여 소스 샤드의 모든 처리가 완료되었다고 표시할 때까지 KCL은 새 샤드의 처리를 시작하지 않습니다.

파라미터를 전달하지 않으면 KCL은 `Checkpoint`에 대한 호출이 레코드 프로세서에 전달된 마지막 레코드까지 모두 처리되었다는 의미로 간주합니다. 따라서 레코드 프로세서는 전달된 목록에 있는 모든 레코드를 반드시 처리한 후에 `Checkpoint`를 호출해야 합니다. 레코드 프로세서는 `Checkpoint`를 호출할 때마다 `ProcessRecords`를 호출할 필요가 없습니다. 예를 들어, 프로세서는 세 번째 또는 네 번째 호출마다 `Checkpoint`를 호출할 수 있습니다. 선택적으로 레코드의 정확한 시퀀스 번호를 `Checkpoint`의 파라미터로 지정할 수도 있습니다. 이 경우 KCL은 레코드가 해당 레코드까지만 처리되었다고 간주합니다.

이 샘플에서는 프라이빗 메서드 `Checkpoint`(`Checkpoint`)가 적절한 예외 처리 및 재시도 로직을 사용하여 `Checkpoint` 메서드를 호출하는 방법을 보여줍니다.

.NET용 KCL은 데이터 레코드를 처리할 때 발생하는 예외를 처리하지 않는다는 점에서 다른 KCL 언어 라이브러리와는 다르게 예외를 처리합니다. 사용자 코드에서 확인할 수 없는 예외가 발생하면 프로그램이 중단됩니다.

## Shutdown

처리가 종료될 때(종료 이유가 `TERMINATE`) 또는 워커가 더 이상 응답하지 않을 때(종료 `input.Reason` 값이 `ZOMBIE`) KCL은 `Shutdown` 메서드를 직접적으로 호출합니다.

```
public void Shutdown(ShutdownInput input)
```

샤드 분할이나 병합 또는 스트림 삭제로 인해 레코드 프로세서가 샤드에서 추가 레코드를 수신하지 않으면 처리가 종료됩니다.

또한 KCL은 `Checkpoint` 객체를 `shutdown`에 전달합니다. 종료 이유가 `TERMINATE`이면 레코드 프로세서가 데이터 레코드 처리를 완료하고 이 인터페이스의 `checkpoint` 메서드를 호출해야 합니다.

## 구성 속성 수정

이 샘플 소비자는 구성 속성의 기본값을 제공합니다. 속성을 사용자의 값으로 재정의할 수 있습니다 (`SampleConsumer/kcl.properties` 참조).

## 애플리케이션 이름

KCL에는 애플리케이션 및 같은 리전의 Amazon DynamoDB 테이블에서 고유한 애플리케이션이 필요합니다. 다음과 같이 애플리케이션 이름 구성 값이 사용됩니다.

- 이 애플리케이션 이름과 관련된 모든 작업자는 동일한 스트림에서 함께 작업한다고 간주됩니다. 이 작업자는 여러 인스턴스에 분산되어 있을 수 있습니다. 동일한 애플리케이션 코드의 추가 인스턴스를 다른 애플리케이션 이름으로 실행하는 경우 KCL은 두 번째 인스턴스를 동일한 스트림에서 작동하는 완전히 별개의 애플리케이션으로 취급합니다.
- KCL은 애플리케이션 이름이 있는 DynamoDB 테이블을 생성하고 테이블을 사용하여 애플리케이션의 상태 정보(예: 체크포인트 및 워커와 샤드의 매핑)를 보관합니다. 각각의 애플리케이션에는 자체 DynamoDB 테이블이 있습니다. 자세한 설명은 [리스 테이블을 사용하여 KCL 소비자 애플리케이션에서 처리한 샤드 추적](#) 섹션을 참조하세요.

## 자격 증명 설정

기본 보안 인증 공급자 체인의 보안 인증 공급자 중 하나가 AWS 보안 인증을 사용할 수 있도록 해야 합니다. `AWSCredentialsProvider` 속성을 사용하여 자격 증명 공급자를 설정할 수 있습니다. [sample.properties](#)에서 [기본 자격 증명 공급자 체인](#)의 자격 증명 공급자 중 하나에 대해 자격 증명을 사용할 수 있도록 해야 합니다. EC2 인스턴스에서 소비자 애플리케이션을 실행하는 경우 IAM 역할로 인스턴스를 구성하는 것이 좋습니다. 이 IAM 역할과 연결된 권한을 반영하는 AWS 보안 인증은 인스턴스 메타데이터를 통해 인스턴스의 애플리케이션에 제공됩니다. 이것이 EC2 인스턴스에서 실행되는 소비자의 자격 증명을 관리하는 가장 안전한 방법입니다.

샘플의 속성 파일에서는 `AmazonKinesisSampleConsumer.cs`에 제공된 레코드 프로세서를 사용하여 'words'라는 Kinesis 데이터 스트림을 처리하도록 KCL을 구성합니다.

## Python으로 Kinesis Client Library 소비자 개발

Kinesis Client Library(KCL)를 사용하여 Kinesis 데이터 스트림의 데이터를 처리하는 애플리케이션을 빌드합니다. Kinesis Client Library는 여러 언어로 제공됩니다. 이 주제에서는 Python에 대해 설명합니다.

KCL은 Java 라이브러리입니다. Java 이외의 언어에 대한 지원은 라는 다국어 인터페이스를 사용하여 제공됩니다. `MultiLangDaemon` 이 데몬은 Java 기반이며, Java 이외의 KCL 언어를 사용하는 경우 배경에서 실행됩니다. 따라서 KCL for Python을 설치하고 소비자 앱을 완전히 Python으로 작성하는 경우에도 다음과 같은 이유로 시스템에 Java를 설치해야 합니다. `MultiLangDaemon` 또한 사용 사례에 맞게 사용자 지정해야 할 몇 가지 기본 설정 (예: 연결 AWS 지역) 도 있습니다. `MultiLangDaemon`

MultiLangDaemon GitHub에 대한 자세한 내용은 [KCL MultiLangDaemon 프로젝트 페이지를 참조하십시오](#).

에서 GitHub Python KCL을 다운로드하려면 [Kinesis 클라이언트 라이브러리 \(Python\)](#) 로 이동하십시오. Python KCL 소비자 애플리케이션용 샘플 코드를 다운로드하려면 의 [KCL for Python 샘플 프로젝트](#) 페이지를 참조하십시오. GitHub

Python으로 KCL 소비자 애플리케이션을 구현할 때 다음 작업을 완료해야 합니다.

## Tasks

- [클래스 메서드 구현 RecordProcessor](#)
- [구성 속성 수정](#)

### 클래스 메서드 구현 RecordProcessor

RecordProcess 클래스가 RecordProcessorBase를 확장하여 다음 메서드를 구현해야 합니다. 이 샘플에서는 시작점으로 사용할 수 있는 구현을 제공합니다(sample\_kclpy\_app.py 참조).

```
def initialize(self, shard_id)
def process_records(self, records, checkpoint)
def shutdown(self, checkpoint, reason)
```

### 초기화

KCL은 레코드 프로세서가 인스턴스화될 때 특정 샤드 ID를 파라미터로 전달하여 initialize 메서드를 직접적으로 호출합니다. 이 레코드 프로세서는 해당 샤드만 처리하고 일반적으로 반대의 경우도 마찬가지입니다. 이 샤드는 해당 레코드 프로세서로만 처리됩니다. 하지만 소비자는 데이터 레코드가 두 번 이상 처리될 가능성을 고려해야 합니다. Kinesis Data Streams에서는 소비자의 워커가 샤드의 모든 데이터 레코드를 적어도 한 번은 처리한다는 적어도 한 번 의미론이 통용되기 때문입니다. 둘 이상의 작업자가 특정 샤드를 처리할 수 있는 경우에 대한 자세한 내용은 [리샤딩, 크기 조정 및 병렬 처리](#)를 참조하십시오.

```
def initialize(self, shard_id)
```

### process\_records

KCL은 initialize 메서드에 지정된 샤드의 데이터 레코드 목록을 전달하여 이 메서드를 직접적으로 호출합니다. 구현하는 레코드 프로세서가 소비자의 의미론에 따라 이 레코드의 데이터를 처리합니다.

예를 들어, 워커가 데이터를 전환한 후 그 결과를 Amazon Simple Storage Service(S3) 버킷에 저장할 수 있습니다.

```
def process_records(self, records, checkpointer)
```

데이터 자체뿐 아니라 시퀀스 번호와 파티션 키도 데이터 레코드에 포함됩니다. 작업자가 데이터를 처리할 때 이 값을 사용할 수 있습니다. 예를 들어, 작업자는 파티션 키의 값을 기반으로 데이터를 저장할 S3 버킷을 선택할 수 있습니다. record 딕셔너리가 레코드의 데이터, 시퀀스 번호 및 파티션 키에 액세스하기 위해 다음 키-값 페어를 노출합니다.

```
record.get('data')
record.get('sequenceNumber')
record.get('partitionKey')
```

데이터가 Base64로 인코딩됩니다.

이 샘플의 메서드 process\_records에는 작업자가 레코드의 데이터, 시퀀스 번호 및 파티션 키에 액세스하는 방법을 보여주는 코드가 있습니다.

Kinesis Data Streams는 샤드에서 이미 처리된 레코드를 추적하도록 레코드 프로세서에 요구합니다. KCL은 Checkpointer 객체를 process\_records에 전달하여 이 추적을 처리합니다. 레코드 프로세서는 해당 객체에서 checkpoint 메서드를 직접적으로 호출하여 샤드의 레코드 처리가 얼마나 진행되었는지를 KCL에 알려줍니다. 워커가 실패할 경우 KCL은 이 정보를 사용하여 마지막으로 처리된 레코드에서 샤드 처리를 다시 시작합니다.

분할 또는 병합 작업의 경우 소스 샤드의 프로세서가 checkpoint를 직접적으로 호출하여 소스 샤드의 모든 처리가 완료되었다고 표시할 때까지 KCL은 새 샤드의 처리를 시작하지 않습니다.

파라미터를 전달하지 않으면 KCL은 checkpoint에 대한 호출이 레코드 프로세서에 전달된 마지막 레코드까지 모두 처리되었다는 의미로 간주합니다. 따라서 레코드 프로세서는 전달된 목록에 있는 모든 레코드를 반드시 처리한 후에 checkpoint를 호출해야 합니다. 레코드 프로세서는 checkpoint를 호출할 때마다 process\_records를 호출할 필요가 없습니다. 예를 들어, 프로세서는 세 번째 호출할 때마다 checkpoint를 호출할 수 있습니다. 선택적으로 레코드의 정확한 시퀀스 번호를 checkpoint의 파라미터로 지정할 수도 있습니다. 이 경우 KCL은 모든 레코드가 해당 레코드까지만 처리되었다고 간주합니다.

이 샘플에서는 프라이빗 메서드 checkpoint가 적절한 예외 처리 및 재시도 로직을 사영하여 Checkpointer.checkpoint 메서드를 호출하는 방법을 보여줍니다.

KCL은 `process_records`를 사용하여 데이터 레코드를 처리할 때 발생하는 모든 예외를 처리합니다. `process_records`에서 예외가 발생하면 KCL은 예외 이전에 `process_records`에 전달된 데이터 레코드를 건너뜁니다. 이러한 레코드는 예외가 발생한 프로세서 또는 소비자의 다른 레코드 프로세서로 다시 전송되지 않습니다.

## shutdown

처리가 종료될 때(종료 이유가 `TERMINATE`) 또는 워커가 더 이상 응답하지 않을 때(종료 `reason`이 `ZOMBIE`) KCL은 `shutdown` 메서드를 직접적으로 호출합니다.

```
def shutdown(self, checkpointer, reason)
```

샤드 분할이나 병합 또는 스트림 삭제로 인해 레코드 프로세서가 샤드에서 추가 레코드를 수신하지 않으면 처리가 종료됩니다.

또한 KCL은 `Checkpointer` 객체를 `shutdown`에 전달합니다. 종료 `reason`이 `TERMINATE`이면 레코드 프로세서가 데이터 레코드 처리를 완료하고 이 인터페이스의 `checkpoint` 메서드를 호출해야 합니다.

## 구성 속성 수정

이 샘플은 구성 속성의 기본값을 제공합니다. 속성을 사용자의 값으로 재정의할 수 있습니다 (`sample.properties` 참조).

## 애플리케이션 이름

에는 다른 애플리케이션과 다르게 동일한 리전의 Amazon DynamoDB 테이블에서도 고유한 애플리케이션 이름이 필요합니다. 다음과 같이 애플리케이션 이름 구성 값이 사용됩니다.

- 이 애플리케이션 이름과 관련된 모든 작업자는 동일한 스트림에서 함께 작업한다고 간주됩니다. 이 작업자는 여러 인스턴스에 분산되어 있을 수 있습니다. 동일한 애플리케이션 코드의 추가 인스턴스를 다른 애플리케이션 이름으로 실행하는 경우 KCL은 두 번째 인스턴스를 동일한 스트림에서 작동하는 완전히 별개의 애플리케이션으로 취급합니다.
- KCL은 애플리케이션 이름이 있는 DynamoDB 테이블을 생성하고 테이블을 사용하여 애플리케이션의 상태 정보(예: 체크포인트 및 워커와 샤드의 매핑)를 보관합니다. 각각의 애플리케이션에는 자체 DynamoDB 테이블이 있습니다. 자세한 설명은 [리스 테이블을 사용하여 KCL 소비자 애플리케이션에서 처리한 샤드 추적](#) 섹션을 참조하세요.

## 자격 증명 설정

기본 보안 인증 공급자 체인의 보안 인증 공급자 중 하나가 AWS 보안 인증을 사용할 수 있도록 해야 합니다. `AWSCredentialsProvider` 속성을 사용하여 자격 증명 공급자를 설정할 수 있습니다. [sample.properties](#)에서 [기본 자격 증명 공급자 체인](#)의 자격 증명 공급자 중 하나에 대해 자격 증명을 사용할 수 있도록 해야 합니다. Amazon EC2 인스턴스에서 소비자 애플리케이션을 실행하는 경우 IAM 역할로 인스턴스를 구성하는 것이 좋습니다. 이 IAM 역할과 연결된 권한을 반영하는 AWS 보안 인증은 인스턴스 메타데이터를 통해 인스턴스의 애플리케이션에 제공됩니다. 이것이 EC2 인스턴스에서 실행되는 소비자 애플리케이션의 자격 증명을 관리하는 가장 안전한 방법입니다.

샘플의 속성 파일에서는 `sample_kc1py_app.py`에 제공된 레코드 프로세서를 사용하여 'words'라는 Kinesis 데이터 스트림을 처리하도록 KCL을 구성합니다.

### Ruby로 Kinesis Client Library 소비자 개발

Kinesis Client Library(KCL)를 사용하여 Kinesis 데이터 스트림의 데이터를 처리하는 애플리케이션을 빌드합니다. Kinesis Client Library는 여러 언어로 제공됩니다. 이 주제에서는 Ruby에 대해 설명합니다.

KCL은 Java 라이브러리입니다. Java 이외의 언어에 대한 지원은 라는 다국어 인터페이스를 사용하여 제공됩니다. `MultiLangDaemon` 이 데몬은 Java 기반이며, Java 이외의 KCL 언어를 사용하는 경우 배경에서 실행됩니다. 따라서 Ruby용 KCL을 설치하고 소비자 앱을 완전히 Ruby로 작성하는 경우에도 다음과 같은 이유로 시스템에 Java를 설치해야 합니다. `MultiLangDaemon` 또한 사용 사례에 맞게 사용자 지정해야 할 몇 가지 기본 설정 (예: 연결 AWS 지역) 도 있습니다. `MultiLangDaemon` `MultiLangDaemon` GitHub에 대한 자세한 내용은 [KCL MultiLangDaemon 프로젝트 페이지를 참조하십시오](#).

에서 GitHub Ruby KCL을 다운로드하려면 [Kinesis 클라이언트 라이브러리 \(Ruby\)](#) 로 이동하십시오. Ruby KCL 소비자 애플리케이션용 샘플 코드를 다운로드하려면 의 Ruby용 [KCL](#) 샘플 프로젝트 페이지로 이동하십시오. GitHub

KCL Ruby 지원 라이브러리에 대한 자세한 내용은 [KCL Ruby Gems 설명서](#)를 참조하세요.

### KCL 2.x 소비자 개발

이 주제에서는 Kinesis Client Library(KCL) 버전 2.0을 사용하는 방법을 설명합니다. KCL에 대한 자세한 내용은 [Kinesis Client Library 1.x를 사용하여 소비자 개발](#)에 나와 있는 개요를 참조하세요.

#### 목차

- [Java로 Kinesis Client Library 소비자 개발](#)



- [Python으로 Kinesis Client Library 소비자 개발](#)

## Java로 Kinesis Client Library 소비자 개발

다음 코드는 Java에서 ProcessorFactory 및 RecordProcessor의 구현 예를 보여 줍니다. 향상된 팬아웃 기능을 활용하고 싶다면 [향상된 팬아웃을 사용하는 소비자 만들기](#)를 참조하십시오.

```
/*
 * Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Amazon Software License (the "License").
 * You may not use this file except in compliance with the License.
 * A copy of the License is located at
 *
 * http://aws.amazon.com/asl/
 *
 * or in the "license" file accompanying this file. This file is distributed
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 * express or implied. See the License for the specific language governing
 * permissions and limitations under the License.
 */

/*
 * Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License.
 * A copy of the License is located at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * or in the "license" file accompanying this file. This file is distributed
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 * express or implied. See the License for the specific language governing
 * permissions and limitations under the License.
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.UUID;
```

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

import org.apache.commons.lang3.ObjectUtils;
import org.apache.commons.lang3.RandomStringUtils;
import org.apache.commons.lang3.RandomUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;

import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.PutRecordRequest;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;
import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;

import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;
import software.amazon.kinesis.retrieval.polling.PollingConfig;

/**
 * This class will run a simple app that uses the KCL to read data and uses the AWS SDK
 * to publish data.
 * Before running this program you must first create a Kinesis stream through the AWS
 * console or AWS SDK.
 */
public class SampleSingle {
```

```
private static final Logger log = LoggerFactory.getLogger(SampleSingle.class);

/**
 * Invoke the main method with 2 args: the stream name and (optionally) the region.
 * Verifies valid inputs and then starts running the app.
 */
public static void main(String... args) {
    if (args.length < 1) {
        log.error("At a minimum, the stream name is required as the first argument.
The Region may be specified as the second argument.");
        System.exit(1);
    }

    String streamName = args[0];
    String region = null;
    if (args.length > 1) {
        region = args[1];
    }

    new SampleSingle(streamName, region).run();
}

private final String streamName;
private final Region region;
private final KinesisAsyncClient kinesisClient;

/**
 * Constructor sets streamName and region. It also creates a KinesisClient object
to send data to Kinesis.
 * This KinesisClient is used to send dummy data so that the consumer has something
to read; it is also used
 * indirectly by the KCL to handle the consumption of the data.
 */
private SampleSingle(String streamName, String region) {
    this.streamName = streamName;
    this.region = Region.of(ObjectUtils.firstNonNull(region, "us-east-2"));
    this.kinesisClient =
KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(this.region));
}

private void run() {

    /**
     * Sends dummy data to Kinesis. Not relevant to consuming the data with the KCL
```

```
    */
    ScheduledExecutorService producerExecutor =
Executors.newSingleThreadScheduledExecutor();
    ScheduledFuture<?> producerFuture =
producerExecutor.scheduleAtFixedRate(this::publishRecord, 10, 1, TimeUnit.SECONDS);

    /**
     * Sets up configuration for the KCL, including DynamoDB and CloudWatch
dependencies. The final argument, a
     * ShardRecordProcessorFactory, is where the logic for record processing lives,
and is located in a private
     * class below.
     */
    DynamoDbAsyncClient dynamoClient =
DynamoDbAsyncClient.builder().region(region).build();
    CloudWatchAsyncClient cloudWatchClient =
CloudWatchAsyncClient.builder().region(region).build();
    ConfigsBuilder configsBuilder = new ConfigsBuilder(streamName, streamName,
kinesisClient, dynamoClient, cloudWatchClient, UUID.randomUUID().toString(), new
SampleRecordProcessorFactory());

    /**
     * The Scheduler (also called Worker in earlier versions of the KCL) is the
entry point to the KCL. This
     * instance is configured with defaults provided by the ConfigsBuilder.
     */
    Scheduler scheduler = new Scheduler(
        configsBuilder.checkpointConfig(),
        configsBuilder.coordinatorConfig(),
        configsBuilder.leaseManagementConfig(),
        configsBuilder.lifecycleConfig(),
        configsBuilder.metricsConfig(),
        configsBuilder.processorConfig(),
        configsBuilder.retrievalConfig().retrievalSpecificConfig(new
PollingConfig(streamName, kinesisClient))
    );

    /**
     * Kickoff the Scheduler. Record processing of the stream of dummy data will
continue indefinitely
     * until an exit is triggered.
     */
    Thread schedulerThread = new Thread(scheduler);
    schedulerThread.setDaemon(true);
```

```
    schedulerThread.start();

    /**
     * Allows termination of app by pressing Enter.
     */
    System.out.println("Press enter to shutdown");
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    try {
        reader.readLine();
    } catch (IOException ioex) {
        log.error("Caught exception while waiting for confirm. Shutting down.",
ioex);
    }

    /**
     * Stops sending dummy data.
     */
    log.info("Cancelling producer and shutting down executor.");
    producerFuture.cancel(true);
    producerExecutor.shutdownNow();

    /**
     * Stops consuming data. Finishes processing the current batch of data already
received from Kinesis
     * before shutting down.
     */
    Future<Boolean> gracefulShutdownFuture = scheduler.startGracefulShutdown();
    log.info("Waiting up to 20 seconds for shutdown to complete.");
    try {
        gracefulShutdownFuture.get(20, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        log.info("Interrupted while waiting for graceful shutdown. Continuing.");
    } catch (ExecutionException e) {
        log.error("Exception while executing graceful shutdown.", e);
    } catch (TimeoutException e) {
        log.error("Timeout while waiting for shutdown. Scheduler may not have
exited.");
    }
    log.info("Completed, shutting down now.");
}

    /**
     * Sends a single record of dummy data to Kinesis.
     */
```

```

private void publishRecord() {
    PutRecordRequest request = PutRecordRequest.builder()
        .partitionKey(RandomStringUtils.randomAlphabetic(5, 20))
        .streamName(streamName)
        .data(SdkBytes.fromByteArray(RandomUtils.nextBytes(10)))
        .build();

    try {
        kinesisClient.putRecord(request).get();
    } catch (InterruptedException e) {
        log.info("Interrupted, assuming shutdown.");
    } catch (ExecutionException e) {
        log.error("Exception while sending data to Kinesis. Will try again next
cycle.", e);
    }
}

private static class SampleRecordProcessorFactory implements
ShardRecordProcessorFactory {
    public ShardRecordProcessor shardRecordProcessor() {
        return new SampleRecordProcessor();
    }
}

/**
 * The implementation of the ShardRecordProcessor interface is where the heart of
the record processing logic lives.
 * In this example all we do to 'process' is log info about the records.
 */
private static class SampleRecordProcessor implements ShardRecordProcessor {

    private static final String SHARD_ID_MDC_KEY = "ShardId";

    private static final Logger log =
LoggerFactory.getLogger(SampleRecordProcessor.class);

    private String shardId;

    /**
     * Invoked by the KCL before data records are delivered to the
ShardRecordProcessor instance (via
     * processRecords). In this example we do nothing except some logging.
     *
     * @param initializationInput Provides information related to initialization.
     */
}

```

```

public void initialize(InitializationInput initializationInput) {
    shardId = initializationInput.shardId();
    MDC.put(SHARD_ID_MDC_KEY, shardId);
    try {
        log.info("Initializing @ Sequence: {}",
initializationInput.extendedSequenceNumber());
    } finally {
        MDC.remove(SHARD_ID_MDC_KEY);
    }
}

/**
 * Handles record processing logic. The Amazon Kinesis Client Library will
invoke this method to deliver
 * data records to the application. In this example we simply log our records.
 *
 * @param processRecordsInput Provides the records to be processed as well as
information and capabilities
 *
 * related to them (e.g. checkpointing).
 */
public void processRecords(ProcessRecordsInput processRecordsInput) {
    MDC.put(SHARD_ID_MDC_KEY, shardId);
    try {
        log.info("Processing {} record(s)",
processRecordsInput.records().size());
        processRecordsInput.records().forEach(r -> log.info("Processing record
pk: {} -- Seq: {}", r.partitionKey(), r.sequenceNumber()));
    } catch (Throwable t) {
        log.error("Caught throwable while processing records. Aborting.");
        Runtime.getRuntime().halt(1);
    } finally {
        MDC.remove(SHARD_ID_MDC_KEY);
    }
}

/** Called when the lease tied to this record processor has been lost. Once the
lease has been lost,
 * the record processor can no longer checkpoint.
 *
 * @param leaseLostInput Provides access to functions and data related to the
loss of the lease.
 */
public void leaseLost(LeaseLostInput leaseLostInput) {
    MDC.put(SHARD_ID_MDC_KEY, shardId);
}

```

```
        try {
            log.info("Lost lease, so terminating.");
        } finally {
            MDC.remove(SHARD_ID_MDC_KEY);
        }
    }

    /**
     * Called when all data on this shard has been processed. Checkpointing must
     occur in the method for record
     * processing to be considered complete; an exception will be thrown otherwise.
     *
     * @param shardEndedInput Provides access to a checkpointer method for
     completing processing of the shard.
     */
    public void shardEnded(ShardEndedInput shardEndedInput) {
        MDC.put(SHARD_ID_MDC_KEY, shardId);
        try {
            log.info("Reached shard end checkpointing.");
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            log.error("Exception while checkpointing at shard end. Giving up.", e);
        } finally {
            MDC.remove(SHARD_ID_MDC_KEY);
        }
    }

    /**
     * Invoked when Scheduler has been requested to shut down (i.e. we decide to
     stop running the app by pressing
     * Enter). Checkpoints and logs the data a final time.
     *
     * @param shutdownRequestedInput Provides access to a checkpointer, allowing a
     record processor to checkpoint
     *
     *                                     before the shutdown is completed.
     */
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        MDC.put(SHARD_ID_MDC_KEY, shardId);
        try {
            log.info("Scheduler is shutting down, checkpointing.");
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            log.error("Exception while checkpointing at requested shutdown. Giving
up.", e);
        }
    }
}
```



```

        } finally {
            MDC.remove(SHARD_ID_MDC_KEY);
        }
    }
}
}
}

```

## Python으로 Kinesis Client Library 소비자 개발

Kinesis Client Library(KCL)를 사용하여 Kinesis 데이터 스트림의 데이터를 처리하는 애플리케이션을 빌드합니다. Kinesis Client Library는 여러 언어로 제공됩니다. 이 주제에서는 Python에 대해 설명합니다.

KCL은 Java 라이브러리입니다. Java 이외의 언어에 대한 지원은 라는 다국어 인터페이스를 사용하여 제공됩니다. MultiLangDaemon 이 데몬은 Java 기반이며, Java 이외의 KCL 언어를 사용하는 경우 배경에서 실행됩니다. 따라서 KCL for Python을 설치하고 소비자 앱을 완전히 Python으로 작성하는 경우에도 다음과 같은 이유로 시스템에 Java를 설치해야 합니다. MultiLangDaemon 또한 사용 사례에 맞게 사용자 지정해야 할 몇 가지 기본 설정 (예: 연결 AWS 지역) 도 있습니다. MultiLangDaemon MultiLangDaemon GitHub에 대한 자세한 내용은 [KCL MultiLangDaemon 프로젝트 페이지를 참조하세요](#).

에서 GitHub Python KCL을 다운로드하려면 [Kinesis 클라이언트 라이브러리 \(Python\)](#) 로 이동하십시오. Python KCL 소비자 애플리케이션용 샘플 코드를 다운로드하려면 의 [KCL for Python 샘플 프로젝트](#) 페이지를 참조하십시오. GitHub

Python으로 KCL 소비자 애플리케이션을 구현할 때 다음 작업을 완료해야 합니다.

### 태스크

- [클래스 메서드 구현 RecordProcessor](#)
- [구성 속성 수정](#)

### 클래스 메서드 구현 RecordProcessor

RecordProcess 클래스가 RecordProcessorBase 클래스를 확장하여 다음 메서드를 구현해야 합니다.

```

initialize
process_records

```

```
shutdown_requested
```

이 샘플의 구현을 시작점으로 이용할 수 있습니다.

```
#!/usr/bin/env python

# Copyright 2014-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# Licensed under the Amazon Software License (the "License").
# You may not use this file except in compliance with the License.
# A copy of the License is located at
#
# http://aws.amazon.com/asl/
#
# or in the "license" file accompanying this file. This file is distributed
# on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language governing
# permissions and limitations under the License.

from __future__ import print_function

import sys
import time

from amazon_kclpy import kcl
from amazon_kclpy.v3 import processor

class RecordProcessor(processor.RecordProcessorBase):
    """
    A RecordProcessor processes data from a shard in a stream. Its methods will be
    called with this pattern:

    * initialize will be called once
    * process_records will be called zero or more times
    * shutdown will be called if this MultiLangDaemon instance loses the lease to this
    shard, or the shard ends due
      a scaling change.
    """
    def __init__(self):
        self._SLEEP_SECONDS = 5
        self._CHECKPOINT_RETRIES = 5
        self._CHECKPOINT_FREQ_SECONDS = 60
```

```

self._largest_seq = (None, None)
self._largest_sub_seq = None
self._last_checkpoint_time = None

def log(self, message):
    sys.stderr.write(message)

def initialize(self, initialize_input):
    """
    Called once by a KCLProcess before any calls to process_records

    :param amazon_kclpy.messages.InitializeInput initialize_input: Information
about the lease that this record
    processor has been assigned.
    """
    self._largest_seq = (None, None)
    self._last_checkpoint_time = time.time()

def checkpoint(self, checkpointer, sequence_number=None, sub_sequence_number=None):
    """
    Checkpoints with retries on retryable exceptions.

    :param amazon_kclpy.kcl.Checkpointer checkpointer: the checkpointer provided to
either process_records
    or shutdown
    :param str or None sequence_number: the sequence number to checkpoint at.
    :param int or None sub_sequence_number: the sub sequence number to checkpoint
at.
    """
    for n in range(0, self._CHECKPOINT_RETRIES):
        try:
            checkpointer.checkpoint(sequence_number, sub_sequence_number)
            return
        except kcl.CheckpointError as e:
            if 'ShutdownException' == e.value:
                #
                # A ShutdownException indicates that this record processor should
be shutdown. This is due to
                # some failover event, e.g. another MultiLangDaemon has taken the
lease for this shard.
                #
                print('Encountered shutdown exception, skipping checkpoint')
                return
            elif 'ThrottlingException' == e.value:

```

```

        #
        # A ThrottlingException indicates that one of our dependencies is
is over burdened, e.g. too many
        # dynamo writes. We will sleep temporarily to let it recover.
        #
        if self._CHECKPOINT_RETRIES - 1 == n:
            sys.stderr.write('Failed to checkpoint after {n} attempts,
giving up.\n'.format(n=n))
            return
        else:
            print('Was throttled while checkpointing, will attempt again in
{s} seconds'
                  .format(s=self._SLEEP_SECONDS))
        elif 'InvalidStateException' == e.value:
            sys.stderr.write('MultiLangDaemon reported an invalid state while
checkpointing.\n')
        else: # Some other error
            sys.stderr.write('Encountered an error while checkpointing, error
was {e}.\n'.format(e=e))
            time.sleep(self._SLEEP_SECONDS)

    def process_record(self, data, partition_key, sequence_number,
sub_sequence_number):
        """
        Called for each record that is passed to process_records.

        :param str data: The blob of data that was contained in the record.
        :param str partition_key: The key associated with this record.
        :param int sequence_number: The sequence number associated with this record.
        :param int sub_sequence_number: the sub sequence number associated with this
record.
        """
        #####
        # Insert your processing logic here
        #####
        self.log("Record (Partition Key: {pk}, Sequence Number: {seq}, Subsequence
Number: {sseq}, Data Size: {ds}"
                .format(pk=partition_key, seq=sequence_number,
sseq=sub_sequence_number, ds=len(data)))

    def should_update_sequence(self, sequence_number, sub_sequence_number):
        """
        Determines whether a new larger sequence number is available

```

```

        :param int sequence_number: the sequence number from the current record
        :param int sub_sequence_number: the sub sequence number from the current record
        :return boolean: true if the largest sequence should be updated, false
otherwise
    """
    return self._largest_seq == (None, None) or sequence_number >
self._largest_seq[0] or \
        (sequence_number == self._largest_seq[0] and sub_sequence_number >
self._largest_seq[1])

    def process_records(self, process_records_input):
        """
        Called by a KCLProcess with a list of records to be processed and a
        checkpoint which accepts sequence numbers
        from the records to indicate where in the stream to checkpoint.

        :param amazon_kclpy.messages.ProcessRecordsInput process_records_input: the
        records, and metadata about the
            records.
        """
        try:
            for record in process_records_input.records:
                data = record.binary_data
                seq = int(record.sequence_number)
                sub_seq = record.sub_sequence_number
                key = record.partition_key
                self.process_record(data, key, seq, sub_seq)
                if self.should_update_sequence(seq, sub_seq):
                    self._largest_seq = (seq, sub_seq)

            #
            # Checkpoints every self._CHECKPOINT_FREQ_SECONDS seconds
            #
            if time.time() - self._last_checkpoint_time >
self._CHECKPOINT_FREQ_SECONDS:
                self.checkpoint(process_records_input.checkpointer,
str(self._largest_seq[0]), self._largest_seq[1])
                self._last_checkpoint_time = time.time()

        except Exception as e:
            self.log("Encountered an exception while processing records. Exception was
{e}\n".format(e=e))

    def lease_lost(self, lease_lost_input):

```

```

        self.log("Lease has been lost")

    def shard_ended(self, shard_ended_input):
        self.log("Shard has ended checkpointing")
        shard_ended_input.checkpointer.checkpoint()

    def shutdown_requested(self, shutdown_requested_input):
        self.log("Shutdown has been requested, checkpointing.")
        shutdown_requested_input.checkpointer.checkpoint()

if __name__ == "__main__":
    kcl_process = kcl.KCLProcess(RecordProcessor())
    kcl_process.run()

```

## 구성 속성 수정

이 샘플은 아래 스크립트에 나와 있는 구성 속성의 기본값을 제공합니다. 속성을 원하는 값으로 재정의할 수 있습니다.

```

# The script that abides by the multi-language protocol. This script will
# be executed by the MultiLangDaemon, which will communicate with this script
# over STDIN and STDOUT according to the multi-language protocol.
executableName = sample_kclpy_app.py

# The name of an Amazon Kinesis stream to process.
streamName = words

# Used by the KCL as the name of this application. Will be used as the name
# of an Amazon DynamoDB table which will store the lease and checkpoint
# information for workers with this application name
applicationName = PythonKCLSample

# Users can change the credentials provider the KCL will use to retrieve credentials.
# The DefaultAWSCredentialsProviderChain checks several other providers, which is
# described here:
# http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/auth/
# DefaultAWSCredentialsProviderChain.html
AWSCredentialsProvider = DefaultAWSCredentialsProviderChain

# Appended to the user agent of the KCL. Does not impact the functionality of the
# KCL in any other way.
processingLanguage = python/2.7

```

```
# Valid options at TRIM_HORIZON or LATEST.
# See http://docs.aws.amazon.com/kinesis/latest/APIReference/API\_GetShardIterator.html#API\_GetShardIterator\_RequestSyntax
initialPositionInStream = TRIM_HORIZON

# The following properties are also available for configuring the KCL Worker that is
  created
# by the MultiLangDaemon.

# The KCL defaults to us-east-1
#regionName = us-east-1

# Fail over time in milliseconds. A worker which does not renew it's lease within this
  time interval
# will be regarded as having problems and it's shards will be assigned to other
  workers.
# For applications that have a large number of shards, this msy be set to a higher
  number to reduce
# the number of DynamoDB IOPS required for tracking leases
#failoverTimeMillis = 10000

# A worker id that uniquely identifies this worker among all workers using the same
  applicationName
# If this isn't provided a MultiLangDaemon instance will assign a unique workerId to
  itself.
#workerId =

# Shard sync interval in milliseconds - e.g. wait for this long between shard sync
  tasks.
#shardSyncIntervalMillis = 60000

# Max records to fetch from Kinesis in a single GetRecords call.
#maxRecords = 10000

# Idle time between record reads in milliseconds.
#idleTimeBetweenReadsInMillis = 1000

# Enables applications flush/checkpoint (if they have some data "in progress", but
  don't get new data for while)
#callProcessRecordsEvenForEmptyRecordList = false

# Interval in milliseconds between polling to check for parent shard completion.
```

```

# Polling frequently will take up more DynamoDB IOPS (when there are leases for shards
  waiting on
# completion of parent shards).
#parentShardPollIntervalMillis = 10000

# Cleanup leases upon shards completion (don't wait until they expire in Kinesis).
# Keeping leases takes some tracking/resources (e.g. they need to be renewed,
  assigned), so by default we try
# to delete the ones we don't need any longer.
#cleanupLeasesUponShardCompletion = true

# Backoff time in milliseconds for Amazon Kinesis Client Library tasks (in the event of
  failures).
#taskBackoffTimeMillis = 500

# Buffer metrics for at most this long before publishing to CloudWatch.
#metricsBufferTimeMillis = 10000

# Buffer at most this many metrics before publishing to CloudWatch.
#metricsMaxQueueSize = 10000

# KCL will validate client provided sequence numbers with a call to Amazon Kinesis
  before checkpointing for calls
# to RecordProcessorCheckpoint#checkpoint(String) by default.
#validateSequenceNumberBeforeCheckpointing = true

# The maximum number of active threads for the MultiLangDaemon to permit.
# If a value is provided then a FixedThreadPool is used with the maximum
# active threads set to the provided value. If a non-positive integer or no
# value is provided a CachedThreadPool is used.
#maxActiveThreads = 0

```

## 애플리케이션 이름

KCL에는 다른 애플리케이션과 다르게 동일한 리전의 Amazon DynamoDB 테이블에서도 고유한 애플리케이션 이름이 필요합니다. 다음과 같이 애플리케이션 이름 구성 값이 사용됩니다.

- 이 애플리케이션 이름과 관련된 모든 작업자는 동일한 스트림에서 함께 작업한다고 간주됩니다. 작업자는 여러 인스턴스에 분산되어 있을 수 있습니다. 동일한 애플리케이션 코드의 추가 인스턴스를 다른 애플리케이션 이름으로 실행하는 경우 KCL은 두 번째 인스턴스를 동일한 스트림에서 작동하는 완전히 별개의 애플리케이션으로 취급합니다.



- KCL은 애플리케이션 이름이 있는 DynamoDB 테이블을 생성하고 테이블을 사용하여 애플리케이션의 상태 정보(예: 체크포인트 및 워커와 샤드의 매핑)를 보관합니다. 각각의 애플리케이션에는 자체 DynamoDB 테이블이 있습니다. 자세히 알아보려면 [리스 테이블을 사용하여 KCL 소비자 애플리케이션에서 처리한 샤드 추적](#)의 내용을 참조하세요.

## 자격 증명

[기본 보안 인증 공급자 체인](#)의 보안 인증 공급자 중 하나가 AWS 보안 인증을 사용할 수 있도록 해야 합니다. AWSCredentialsProvider 속성을 사용하여 자격 증명 공급자를 설정할 수 있습니다. Amazon EC2 인스턴스에서 소비자 애플리케이션을 실행하는 경우 IAM 역할로 인스턴스를 구성하는 것이 좋습니다. 이 IAM 역할과 연결된 권한을 반영하는 AWS 보안 인증은 인스턴스 메타데이터를 통해 인스턴스의 애플리케이션에 제공됩니다. 이것이 EC2 인스턴스에서 실행되는 소비자 애플리케이션의 자격 증명을 관리하는 가장 안전한 방법입니다.

## AWS SDK for Java를 사용하여 공유 처리량으로 사용자 지정 소비자 개발

공유 처리량으로 사용자 지정 Kinesis Data Streams 소비자를 개발하는 방법 중 하나는 Amazon Kinesis Data Streams API를 사용하는 것입니다. 이 섹션에서는 AWS SDK for Java와 함께 Kinesis Data Streams API를 사용하는 방법을 설명합니다. 이 섹션의 Java 샘플 코드는 기본 KDS API 작업을 수행하는 방법을 설명하며, 작업 유형에 따라 논리적으로 나뉘어집니다.

이러한 예제는 프로덕션에 사용할 수 있는 코드를 제공하지 않습니다. 이 예제는 가능한 모든 예외를 확인하지 않거나 가능한 모든 보안 및 성능 고려 사항을 감안하지 않습니다.

다른 프로그래밍 언어를 사용하여 Kinesis Data Streams API를 직접적으로 호출할 수 있습니다. 가능한 모든 AWS SDK에 대한 자세한 내용은 [Amazon Web Services로 개발 시작](#)을 참조하세요.

### Important

공유 처리량으로 사용자 지정 Kinesis Data Streams 소비자를 개발하는 데 권장되는 방법은 Kinesis Client Library(KCL)를 사용하는 것입니다. KCL을 사용하면 분산 컴퓨팅과 관련된 많은 복잡한 작업을 처리하여 Kinesis 데이터 스트림의 데이터를 사용하고 처리할 수 있습니다. 자세한 내용은 [KCL을 사용하여 공유 처리량으로 사용자 지정 소비자 개발](#)을 참조하세요.

## 주제

- [스트림에서 데이터 가져오기](#)
- [샤드 반복자 사용](#)

- [GetRecords 사용](#)
- [리샤딩에 적용](#)
- [AWS Glue 스키마 레지스트리를 사용하여 데이터와 상호 작용](#)

## 스트림에서 데이터 가져오기

Kinesis Data Streams API에는 데이터 스트림에서 레코드를 검색하기 위해 간접적으로 호출할 수 있는 `getShardIterator` 및 `getRecords` 메서드가 포함되어 있습니다. 이 풀 모델에서는 코드가 데이터 스트림의 샤드에서 직접 데이터 레코드를 가져옵니다.

### Important

KCL에서 제공하는 레코드 프로세서 지원을 사용하여 데이터 스트림에서 레코드를 검색하는 것이 좋습니다. 이 푸시 모델에서는 데이터를 처리하는 코드를 구현합니다. KCL은 데이터 스트림에서 데이터 레코드를 검색하고 애플리케이션 코드로 전송합니다. KCL에서는 장애 조치, 복구 및 로드 밸런싱 기능도 제공합니다. 자세한 내용은 [KCL을 사용하여 공유 처리량으로 사용자 지정 소비자 개발](#)을 참조하세요.

그러나 Kinesis Data Streams API 사용을 선호하는 경우도 있을 수 있습니다. 예를 들어, 데이터 스트림을 모니터링하거나 디버깅하기 위해 사용자 지정 도구를 구현할 수 있습니다.

### Important

Kinesis Data Streams는 데이터 스트림의 데이터 레코드 보존 기간에 대한 변경을 지원합니다. 자세한 내용은 [데이터 보존 기간 변경](#) 섹션을 참조하세요.

## 샤드 반복자 사용

샤드 수를 기준으로 스트림에서 레코드를 검색합니다. 각 샤드와 샤드에서 검색한 레코드의 각 배치에 대한 샤드 반복자를 가져와야 합니다. 레코드를 검색할 샤드를 지정하기 위해 `getRecordsRequest` 객체에 샤드 반복자가 사용됩니다. 샤드 반복자와 연결된 유형은 샤드에서 레코드를 검색할 지점을 결정합니다(자세한 내용은 이 단원의 뒷부분 참조). 샤드 반복자로 작업하기 전에 [DescribeStream API - 더 이상 사용되지 않음](#)에서 설명하는 대로 샤드를 검색해야 합니다.

`getShardIterator` 메서드를 사용하여 초기 샤드 반복자를 가져옵니다. `getNextShardIterator` 메서드에서 반환된 `getRecordsResult` 객체의 `getRecords` 메서드를 사용하여 추가 레코드 배치

의 샤드 반복자를 가져옵니다. 샤드 반복자는 5분간 유효합니다. 유효한 시간 동안 샤드 반복자를 사용하면 새로운 샤드 반복자를 얻을 수 있습니다. 각 샤드 반복자는 사용 후에도 5분간 유효합니다.

초기 샤드 반복자를 가져오려면 `GetShardIteratorRequest`를 인스턴스화하고 `getShardIterator` 메서드에 전달합니다. 요청을 구성하려면 스트림과 샤드 ID를 지정하십시오. AWS 계정에서 스트림을 가져오는 방법에 대한 자세한 내용은 [스트림 나열](#) 섹션을 참조하세요. 스트림에서 샤드를 가져오는 방법에 대한 자세한 내용은 [DescribeStream API - 더 이상 사용되지 않음](#)을 참조하십시오.

```
String shardIterator;
GetShardIteratorRequest getShardIteratorRequest = new GetShardIteratorRequest();
getShardIteratorRequest.setStreamName(myStreamName);
getShardIteratorRequest.setShardId(shard.getShardId());
getShardIteratorRequest.setShardIteratorType("TRIM_HORIZON");

GetShardIteratorResult getShardIteratorResult =
    client.getShardIterator(getShardIteratorRequest);
shardIterator = getShardIteratorResult.getShardIterator();
```

이 샘플 코드는 초기 샤드 반복자를 가져올 때 반복자 유형으로 `TRIM_HORIZON`을 지정합니다. 이 반복자 유형은 가장 최근에 추가한 레코드(팁이라고도 함)가 아니라 샤드에 추가된 첫 번째 레코드부터 반환해야 함을 의미합니다. 다음은 가능한 반복자 유형입니다.

- `AT_SEQUENCE_NUMBER`
- `AFTER_SEQUENCE_NUMBER`
- `AT_TIMESTAMP`
- `TRIM_HORIZON`
- `LATEST`

자세한 내용은 [ShardIteratorType](#)을 참조하십시오.

일부 반복자 유형에는 유형 외에 시퀀스 번호도 지정해야 합니다. 예를 들면 다음과 같습니다.

```
getShardIteratorRequest.setShardIteratorType("AT_SEQUENCE_NUMBER");
getShardIteratorRequest.setStartingSequenceNumber(specialSequenceNumber);
```

`getRecords`를 사용하여 레코드를 가져온 후에 레코드의 `getSequenceNumber` 메서드를 호출하여 레코드의 시퀀스 번호를 가져올 수 있습니다.

```
record.getSequenceNumber()
```

또한 데이터 스트림에 레코드를 추가하는 코드는 `getSequenceNumber` 결과에 `putRecord`를 호출하여 추가된 레코드의 시퀀스 번호를 가져올 수 있습니다.

```
lastSequenceNumber = putRecordResult.getSequenceNumber();
```

시퀀스 번호를 사용하여 레코드 순서가 확실하게 증가하도록 보장할 수 있습니다. 자세한 내용은 [PutRecord 예제](#)의 코드 예제를 참조하십시오.

## GetRecords 사용

샤드 반복자를 가져온 후 `GetRecordsRequest` 객체를 인스턴스화하십시오. `setShardIterator` 메서드를 사용하여 요청에 반복자를 지정하십시오.

선택적으로 `setLimit` 메서드를 사용하여 검색할 레코드 수를 설정할 수 있습니다. `getRecords`에 의해 반환된 레코드 수는 항상 이 제한보다 적거나 같습니다. 이 제한을 지정하지 않으면 `getRecords`가 검색된 레코드의 10MB를 반환합니다. 아래 샘플 코드에서는 제한을 레코드 25개로 설정합니다.

반환된 레코드가 없으면 샤드 반복자가 참조하는 시퀀스 번호로 이 샤드에서 현재 사용할 수 있는 데이터 레코드가 없음을 의미합니다. 이 상황에서, 애플리케이션은 스트림의 데이터 소스에 적절한 시간 동안 대기해야 합니다. 그런 다음 이전의 `getRecords` 호출에서 반환된 샤드 반복자를 사용하여 샤드에서 데이터를 다시 가져오십시오.

`getRecordsRequest`를 `getRecords` 메서드에 전달하고 반환된 값을 `getRecordsResult` 객체로 캡처하십시오. 데이터 레코드를 가져오려면 `getRecords` 객체에서 `getRecordsResult` 메서드를 호출하십시오.

```
GetRecordsRequest getRecordsRequest = new GetRecordsRequest();
getRecordsRequest.setShardIterator(shardIterator);
getRecordsRequest.setLimit(25);

GetRecordsResult getRecordsResult = client.getRecords(getRecordsRequest);
List<Record> records = getRecordsResult.getRecords();
```

`getRecords`에 대한 또 다른 호출을 준비하려면 `getRecordsResult`에서 다음 샤드 반복자를 가져 오십시오.

```
shardIterator = getRecordsResult.getNextShardIterator();
```

최상의 결과를 얻으려면 `getRecords` 빈도 제한을 초과하지 않도록 `getRecords` 호출 사이에 적어도 1초(1,000밀리초)의 대기 시간을 유지하십시오.

```
try {
    Thread.sleep(1000);
}
catch (InterruptedException e) {}
```

일반적으로 테스트 시나리오에서 단일 레코드를 검색하는 경우에도 루프에서 `getRecords`를 호출해야 합니다. `getRecords`를 한 번 호출하면 샤드가 이후의 시퀀스 번호에 더 많은 레코드를 포함하더라도 빈 레코드 목록이 반환될 수 있습니다. 이 경우 빈 레코드 목록과 함께 반환된 `NextShardIterator`가 샤드의 이후 시퀀스 번호를 참조하며 연속적으로 `getRecords` 호출하면 최종적으로 레코드가 반환됩니다. 다음 샘플은 루프의 사용을 보여줍니다.

예제: `getRecords`

다음 코드 예제는 루프에서 이루어지는 호출을 포함하여 이 단원에 나온 `getRecords` 팁을 반영합니다.

```
// Continuously read data records from a shard
List<Record> records;

while (true) {

    // Create a new getRecordsRequest with an existing shardIterator
    // Set the maximum records to return to 25

    GetRecordsRequest getRecordsRequest = new GetRecordsRequest();
    getRecordsRequest.setShardIterator(shardIterator);
    getRecordsRequest.setLimit(25);

    GetRecordsResult result = client.getRecords(getRecordsRequest);

    // Put the result into record list. The result can be empty.
```

```

records = result.getRecords();

try {
    Thread.sleep(1000);
}
catch (InterruptedException exception) {
    throw new RuntimeException(exception);
}

shardIterator = result.getNextShardIterator();
}

```

Kinesis Client Library를 사용하는 경우 데이터를 반환하기 전에 여러 번 직접적으로 호출할 수 있습니다. 이 동작은 설계에 따른 것이며 KCL이나 데이터에 문제가 있는 것은 아닙니다.

## 리샤딩에 적용

`getRecordsResult.getNextShardIterator`가 `null`을 반환하는 경우 이는 이 샤드와 관련된 샤드 분할 또는 병합이 발생했음을 나타냅니다. 이 샤드는 현재 CLOSED 상태이며 이 샤드에서 사용할 수 있는 모든 데이터 레코드를 읽었습니다.

이 시나리오에서는 `getRecordsResult.childShards`를 사용하여 분할 또는 병합으로 생성된 처리 중인 샤드의 새 하위 샤드에 대해 알아볼 수 있습니다. 자세한 내용은 [ChildShard](#)를 참조하세요.

이 분할의 경우 새로운 샤드 2개 모두 이전에 처리한 샤드의 샤드 ID와 동일한 `parentShardId`가 있습니다. 이 샤드 2개의 `adjacentParentShardId` 값은 `null`입니다.

병합으로 생성된 새 단일 샤드에는 상위 샤드 중 하나의 샤드 ID 하나와 동일한 `parentShardId` 및 다른 상위 샤드의 샤드 ID와 동일한 `adjacentParentShardId`가 있습니다. 애플리케이션은 이러한 샤드 중 하나에서 모든 데이터를 이미 읽었습니다. 이것은 `getRecordsResult.getNextShardIterator`가 `null`을 반환한 샤드입니다. 애플리케이션에서 데이터의 순서가 중요한 경우 병합으로 생성된 하위 샤드에서 새로운 데이터를 읽기 전에 다른 상위 샤드의 모든 데이터를 읽으십시오.

여러 프로세서를 사용하여 스트림에서 데이터를 검색하는 경우 만약 샤드당 프로세서가 1개이고 샤드 분할 또는 병합이 발생하면 샤드 수 변경에 따라 프로세서 수를 늘리거나 줄여 조정하십시오.

CLOSED와 같은 샤드 상태 설명을 포함하여 리샤딩에 대한 자세한 내용은 [스트림 리샤딩](#) 섹션을 참조하세요.

## AWS Glue 스키마 레지스트리를 사용하여 데이터와 상호 작용

Kinesis 데이터 스트림을 AWS Glue 스키마 레지스트리와 통합할 수 있습니다. AWS Glue 스키마 레지스트리를 사용하면 스키마를 중앙에서 검색, 제어 및 발전시키는 동시에 생성된 데이터가 등록된 스키마에 의해 지속적으로 검증되도록 할 수 있습니다. 스키마는 데이터 레코드의 구조와 포맷을 정의합니다. 스키마는 신뢰할 수 있는 데이터 게시, 소비 또는 저장을 위한 버전 지정 사양입니다. AWS Glue 스키마 레지스트리를 사용하면 스트리밍 애플리케이션 내에서 엔드 투 엔드 데이터 품질 및 데이터 거버넌스를 개선할 수 있습니다. 자세한 내용은 [AWS Glue Schema Registry](#)를 참조하세요. 이 통합을 설정하는 방법 중 하나는 AWS Java SDK에서 사용 가능한 GetRecords Kinesis Data Streams API를 사용하는 것입니다.

GetRecords Kinesis Data Streams API를 사용하여 Kinesis Data Streams와 스키마 레지스트리의 통합을 설정하는 방법에 대한 자세한 지침은 [사용 사례: AWS Glue Schema Registry와 Amazon Kinesis Data Streams 통합](#)의 'Kinesis Data Streams API를 사용하여 데이터와 상호 작용'을 참조하세요.

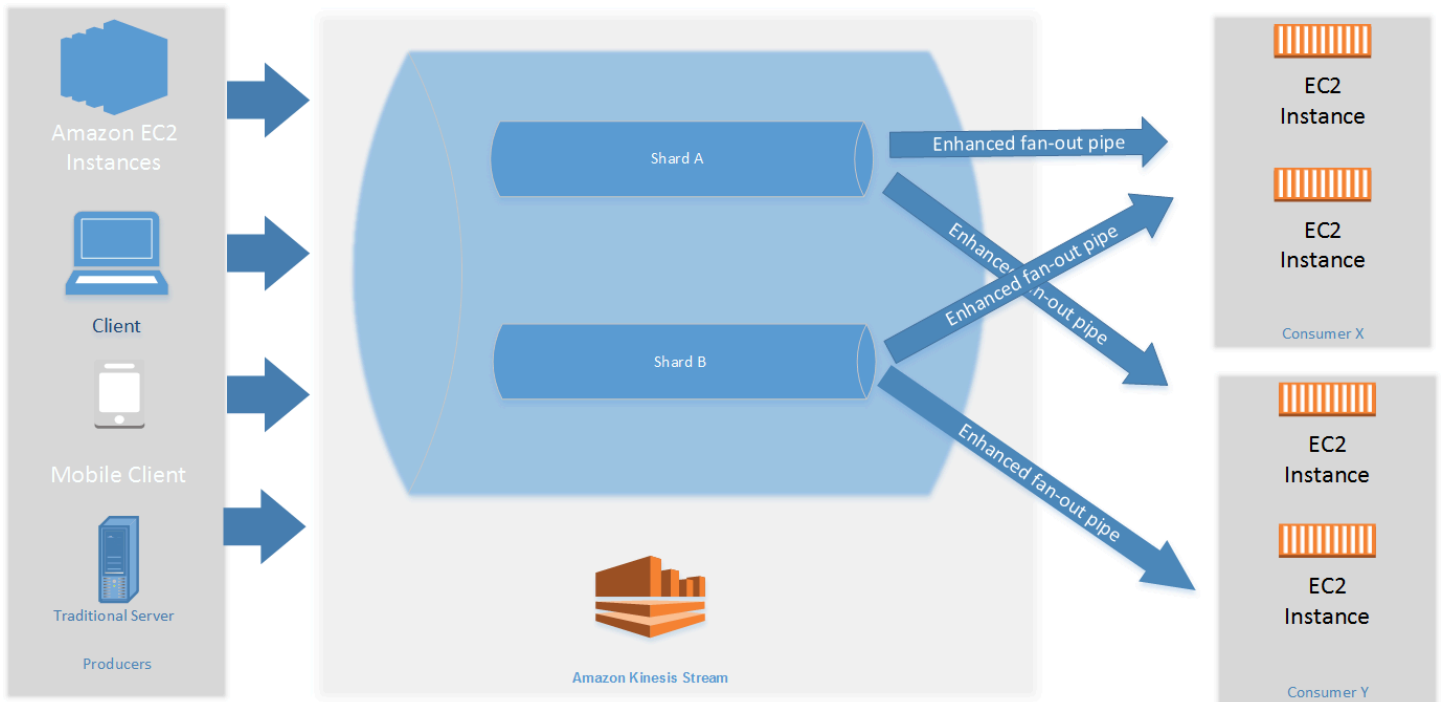
## 전용 처리량으로 사용자 지정 소비자 개발(항상된 팬아웃)

Amazon Kinesis Data Streams에서는 항상된 팬아웃이라는 기능을 사용하는 소비자를 만들 수 있습니다. 이 기능을 사용하면 소비자가 샤드당 1초에 최대 2MB 데이터의 처리량으로 스트림으로부터 레코드를 수신할 수 있습니다. 이 처리량은 전용이므로, 항상된 팬아웃을 사용하는 소비자는 스트림으로부터 데이터를 수신하는 다른 소비자와 경쟁할 필요가 없습니다. Kinesis Data Streams는 스트림의 데이터 레코드를 항상된 팬아웃을 사용하는 소비자에게 푸시합니다. 따라서 이러한 소비자는 데이터를 폴링할 필요가 없습니다.

### Important

스트림당 최대 20명의 소비자를 등록하여 항상된 팬아웃을 사용할 수 있습니다.

다음 다이어그램은 항상된 팬아웃 아키텍처를 보여 줍니다. Amazon Kinesis Client Library(KCL) 버전 2.0 이상을 사용하여 소비자를 빌드하는 경우 KCL은 항상된 팬아웃을 사용하여 스트림의 모든 샤드로부터 데이터를 수신하도록 소비자를 설정합니다. API를 사용하여 항상된 팬아웃을 사용하는 소비자를 빌드하는 경우에는 개별 샤드를 구독할 수 있습니다.



이 다이어그램은 다음을 보여 줍니다.

- 두 개의 샤드를 포함하는 스트림.
- 향상된 팬아웃을 사용하여 소비자 X 스트림과 소비자 Y 스트림으로부터 데이터를 수신하는 소비자 2개. 각 소비자는 스트림의 모든 샤드와 모든 레코드를 구독합니다. KCL 버전 2.0 이상을 사용하여 소비자를 빌드하는 경우, KCL은 이러한 소비자를 스트림의 모든 샤드에 자동으로 등록합니다. 반면에 API를 사용하여 소비자를 빌드하는 경우에는 개별 샤드를 구독할 수 있습니다.
- 소비자가 스트림으로부터 데이터를 수신하기 위해 사용하는 향상된 팬아웃 파이프를 나타내는 화살표. 향상된 팬아웃 파이프는 다른 파이프 또는 총 소비자 수와 상관없이 샤드당 최대 2MB/sec의 데이터를 제공합니다.

## 주제

- [KCL 2.x를 사용하여 향상된 팬아웃 소비자 개발](#)
- [Kinesis Data Streams API를 사용하여 향상된 팬아웃 소비자 개발](#)
- [AWS Management Console을 사용하여 향상된 팬아웃 소비자 관리](#)



## KCL 2.x를 사용하여 향상된 팬아웃 소비자 개발

Amazon Kinesis Data Streams에서 향상된 팬아웃을 사용하는 소비자는 샤드당 초당 최대 데이터 2MB의 전용 처리량으로 데이터 스트림에서 레코드를 수신할 수 있습니다. 이 유형의 소비자는 스트림으로부터 데이터를 수신하는 다른 소비자와 경쟁할 필요가 없습니다. 자세한 내용은 [전용 처리량으로 사용자 지정 소비자 개발\(향상된 팬아웃\)](#) 섹션을 참조하세요.

Kinesis Client Library(KCL) 버전 2.0 이상을 사용하면 향상된 팬아웃을 사용하여 스트림으로부터 데이터를 수신하는 애플리케이션을 개발할 수 있습니다. KCL은 애플리케이션을 스트림의 모든 샤드에 자동으로 등록하므로 소비자 애플리케이션이 샤드당 2MB/sec의 처리 속도로 읽을 수 있습니다. 향상된 팬아웃을 켜지 않고 KCL을 사용하고 싶다면 [Developing Consumers Using the Kinesis Client Library 2.0](#)을 참조하세요.

### 주제

- [Java에서 KCL 2.x를 사용하여 향상된 팬아웃 소비자 개발](#)

## Java에서 KCL 2.x를 사용하여 향상된 팬아웃 소비자 개발

Kinesis Client Library(KCL) 버전 2.0 이상을 사용하면 Amazon Kinesis Data Streams에서 향상된 팬아웃을 통해 스트림에서 데이터를 수신하는 애플리케이션을 개발할 수 있습니다. 다음 코드는 Java에서 ProcessorFactory 및 RecordProcessor의 구현 예를 보여 줍니다.

KinesisClientUtil을 사용하여 KinesisAsyncClient를 만들고 KinesisAsyncClient에 maxConcurrency를 구성하는 것이 좋습니다.

### Important

Amazon Kinesis Client는 KinesisAsyncClient를 구성하여 KinesisAsyncClient의 전체 임대 수에 더해 추가 사용량까지 허용할 수 있을 만큼 maxConcurrency를 충분히 높이지 않을 경우 지연 시간이 크게 증가할 수 있습니다.

```
/*
 * Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Amazon Software License (the "License").
 * You may not use this file except in compliance with the License.
 * A copy of the License is located at
 *
 */
```

```
* http://aws.amazon.com/asl/
*
* or in the "license" file accompanying this file. This file is distributed
* on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
* express or implied. See the License for the specific language governing
* permissions and limitations under the License.
*/

/*
* Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
*
* Licensed under the Apache License, Version 2.0 (the "License").
* You may not use this file except in compliance with the License.
* A copy of the License is located at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* or in the "license" file accompanying this file. This file is distributed
* on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
* express or implied. See the License for the specific language governing
* permissions and limitations under the License.
*/

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.UUID;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

import org.apache.commons.lang3.ObjectUtils;
import org.apache.commons.lang3.RandomStringUtils;
import org.apache.commons.lang3.RandomUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;

import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
```

```
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.PutRecordRequest;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;
import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;
import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class SampleSingle {

    private static final Logger log = LoggerFactory.getLogger(SampleSingle.class);

    public static void main(String... args) {
        if (args.length < 1) {
            log.error("At a minimum, the stream name is required as the first argument.
The Region may be specified as the second argument.");
            System.exit(1);
        }

        String streamName = args[0];
        String region = null;
        if (args.length > 1) {
            region = args[1];
        }

        new SampleSingle(streamName, region).run();
    }

    private final String streamName;
    private final Region region;
    private final KinesisAsyncClient kinesisClient;

    private SampleSingle(String streamName, String region) {
        this.streamName = streamName;
        this.region = Region.of(ObjectUtils.firstNonNull(region, "us-east-2"));
    }
}
```

```
        this.kinesisClient =
KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(this.region));
    }

    private void run() {
        ScheduledExecutorService producerExecutor =
Executors.newSingleThreadScheduledExecutor();
        ScheduledFuture<?> producerFuture =
producerExecutor.scheduleAtFixedRate(this::publishRecord, 10, 1, TimeUnit.SECONDS);

        DynamoDbAsyncClient dynamoClient =
DynamoDbAsyncClient.builder().region(region).build();
        CloudWatchAsyncClient cloudWatchClient =
CloudWatchAsyncClient.builder().region(region).build();
        ConfigsBuilder configsBuilder = new ConfigsBuilder(streamName, streamName,
kinesisClient, dynamoClient, cloudWatchClient, UUID.randomUUID().toString(), new
SampleRecordProcessorFactory());

        Scheduler scheduler = new Scheduler(
            configsBuilder.checkpointConfig(),
            configsBuilder.coordinatorConfig(),
            configsBuilder.leaseManagementConfig(),
            configsBuilder.lifecycleConfig(),
            configsBuilder.metricsConfig(),
            configsBuilder.processorConfig(),
            configsBuilder.retrievalConfig()
        );

        Thread schedulerThread = new Thread(scheduler);
        schedulerThread.setDaemon(true);
        schedulerThread.start();

        System.out.println("Press enter to shutdown");
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        try {
            reader.readLine();
        } catch (IOException ioex) {
            log.error("Caught exception while waiting for confirm. Shutting down.",
ioex);
        }

        log.info("Cancelling producer, and shutting down executor.");
        producerFuture.cancel(true);
        producerExecutor.shutdownNow();
    }
}
```

```
Future<Boolean> gracefulShutdownFuture = scheduler.startGracefulShutdown();
log.info("Waiting up to 20 seconds for shutdown to complete.");
try {
    gracefulShutdownFuture.get(20, TimeUnit.SECONDS);
} catch (InterruptedException e) {
    log.info("Interrupted while waiting for graceful shutdown. Continuing.");
} catch (ExecutionException e) {
    log.error("Exception while executing graceful shutdown.", e);
} catch (TimeoutException e) {
    log.error("Timeout while waiting for shutdown. Scheduler may not have
exited.");
}
log.info("Completed, shutting down now.");
}

private void publishRecord() {
    PutRecordRequest request = PutRecordRequest.builder()
        .partitionKey(RandomStringUtils.randomAlphabetic(5, 20))
        .streamName(streamName)
        .data(SdkBytes.fromByteArray(RandomUtils.nextBytes(10)))
        .build();

    try {
        kinesisClient.putRecord(request).get();
    } catch (InterruptedException e) {
        log.info("Interrupted, assuming shutdown.");
    } catch (ExecutionException e) {
        log.error("Exception while sending data to Kinesis. Will try again next
cycle.", e);
    }
}

private static class SampleRecordProcessorFactory implements
ShardRecordProcessorFactory {
    public ShardRecordProcessor shardRecordProcessor() {
        return new SampleRecordProcessor();
    }
}

private static class SampleRecordProcessor implements ShardRecordProcessor {

    private static final String SHARD_ID_MDC_KEY = "ShardId";
```

```
private static final Logger log =
LoggerFactory.getLogger(SampleRecordProcessor.class);

private String shardId;

public void initialize(InitializationInput initializationInput) {
    shardId = initializationInput.shardId();
    MDC.put(SHARD_ID_MDC_KEY, shardId);
    try {
        log.info("Initializing @ Sequence: {}",
initializationInput.extendedSequenceNumber());
    } finally {
        MDC.remove(SHARD_ID_MDC_KEY);
    }
}

public void processRecords(ProcessRecordsInput processRecordsInput) {
    MDC.put(SHARD_ID_MDC_KEY, shardId);
    try {
        log.info("Processing {} record(s)",
processRecordsInput.records().size());
        processRecordsInput.records().forEach(r -> log.info("Processing record
pk: {} -- Seq: {}", r.partitionKey(), r.sequenceNumber()));
    } catch (Throwable t) {
        log.error("Caught throwable while processing records. Aborting.");
        Runtime.getRuntime().halt(1);
    } finally {
        MDC.remove(SHARD_ID_MDC_KEY);
    }
}

public void leaseLost(LeaseLostInput leaseLostInput) {
    MDC.put(SHARD_ID_MDC_KEY, shardId);
    try {
        log.info("Lost lease, so terminating.");
    } finally {
        MDC.remove(SHARD_ID_MDC_KEY);
    }
}

public void shardEnded(ShardEndedInput shardEndedInput) {
    MDC.put(SHARD_ID_MDC_KEY, shardId);
    try {
        log.info("Reached shard end checkpointing.");
    }
}
```



해당 샤드의 레코드를 사용자에게 푸시하기 시작합니다. 이 연결은 최대 5분 동안 활성화됩니다. [SubscribeToShard](#) 호출에 의해 반환된 future가 정상적으로 또는 예외적으로 완료된 후에도 샤드에서 레코드를 계속 수신하려면 [SubscribeToShard](#)를 다시 직접적으로 호출합니다.

### Note

또한 [SubscribeToShard](#) API는 현재 샤드의 끝에 도달하면 현재 샤드의 하위 샤드 목록을 반환합니다.

3. 향상된 팬아웃을 사용하는 소비자의 등록을 해제하려면 [DeregisterStreamConsumer](#)를 호출합니다.

다음 코드는 소비자를 샤드에 등록하고, 등록을 정기적으로 갱신하고, 이벤트를 처리하는 방법을 보여주는 예제입니다.

```
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.ShardIteratorType;
import software.amazon.awssdk.services.kinesis.model.SubscribeToShardEvent;
import software.amazon.awssdk.services.kinesis.model.SubscribeToShardRequest;
import
software.amazon.awssdk.services.kinesis.model.SubscribeToShardResponseHandler;

import java.util.concurrent.CompletableFuture;

/**
 * See https://github.com/awsdocs/aws-doc-sdk-examples/blob/master/javav2/
example_code/kinesis/src/main/java/com/example/kinesis/KinesisStreamEx.java
 * for complete code and more examples.
 */
public class SubscribeToShardSimpleImpl {

    private static final String CONSUMER_ARN = "arn:aws:kinesis:us-
east-1:123456789123:stream/foobar/consumer/test-consumer:1525898737";
    private static final String SHARD_ID = "shardId-000000000000";

    public static void main(String[] args) {

        KinesisAsyncClient client = KinesisAsyncClient.create();

        SubscribeToShardRequest request = SubscribeToShardRequest.builder()
            .consumerARN(CONSUMER_ARN)
```



```

        .shardId(SHARD_ID)
        .startingPosition(s -> s.type(ShardIteratorType.LATEST)).build();

    // Call SubscribeToShard iteratively to renew the subscription
    periodically.
    while(true) {
        // Wait for the CompletableFuture to complete normally or
    exceptionally.
        callSubscribeToShardWithVisitor(client, request).join();
    }

    // Close the connection before exiting.
    // client.close();
}

/**
 * Subscribes to the stream of events by implementing the
    SubscribeToShardResponseHandler.Visitor interface.
 */
private static CompletableFuture<Void>
callSubscribeToShardWithVisitor(KinesisAsyncClient client, SubscribeToShardRequest
    request) {
    SubscribeToShardResponseHandler.Visitor visitor = new
    SubscribeToShardResponseHandler.Visitor() {
        @Override
        public void visit(SubscribeToShardEvent event) {
            System.out.println("Received subscribe to shard event " + event);
        }
    };
    SubscribeToShardResponseHandler responseHandler =
    SubscribeToShardResponseHandler
        .builder()
        .onError(t -> System.err.println("Error during stream - " +
    t.getMessage()))
        .subscriber(visitor)
        .build();
    return client.subscribeToShard(request, responseHandler);
}
}

```

`event.ContinuationSequenceNumber`가 `null`을 반환하는 경우 이는 이 샤드와 관련된 샤드 분할 또는 병합이 발생했음을 나타냅니다. 이 샤드는 현재 CLOSED 상태이며 이 샤드에서 사용 가능한 모

든 데이터 레코드를 읽었습니다. 이 시나리오에서는 위의 예에 따라 `event.childShards`를 사용하여 분할 또는 병합으로 생성된 처리 중인 샤드의 새 하위 샤드에 대해 알아볼 수 있습니다. 자세한 내용은 [ChildShard](#)를 참조하세요.

## AWS Glue 스키마 레지스트리를 사용하여 데이터와 상호 작용

Kinesis 데이터 스트림을 AWS Glue 스키마 레지스트리와 통합할 수 있습니다. AWS Glue 스키마 레지스트리를 사용하면 스키마를 중앙에서 검색, 제어 및 발전시키는 동시에 생성된 데이터가 등록된 스키마에 의해 지속적으로 검증되도록 할 수 있습니다. 스키마는 데이터 레코드의 구조와 포맷을 정의합니다. 스키마는 신뢰할 수 있는 데이터 게시, 소비 또는 저장을 위한 버전 지정 사양입니다. AWS Glue 스키마 레지스트리를 사용하면 스트리밍 애플리케이션 내에서 엔드 투 엔드 데이터 품질 및 데이터 거버넌스를 개선할 수 있습니다. 자세한 내용은 [AWS Glue Schema Registry](#)를 참조하세요. 이 통합을 설정하는 방법 중 하나는 AWS Java SDK에서 사용 가능한 `GetRecords` Kinesis Data Streams API를 사용하는 것입니다.

`GetRecords` Kinesis Data Streams API를 사용하여 Kinesis Data Streams와 스키마 레지스트리의 통합을 설정하는 방법에 대한 자세한 지침은 [사용 사례: AWS Glue Schema Registry와 Amazon Kinesis Data Streams 통합](#)의 'Kinesis Data Streams API를 사용하여 데이터와 상호 작용'을 참조하세요.

## AWS Management Console을 사용하여 향상된 팬아웃 소비자 관리

Amazon Kinesis Data Streams에서 향상된 팬아웃을 사용하는 소비자는 샤드당 초당 최대 데이터 2MB의 전용 처리량으로 데이터 스트림에서 레코드를 수신할 수 있습니다. 자세한 내용은 [전용 처리량으로 사용자 지정 소비자 개발\(향상된 팬아웃\)](#) 섹션을 참조하세요.

AWS Management Console을 사용하여 특정 스트림에 팬아웃을 사용하여 등록된 모든 소비자 목록을 볼 수 있습니다. 이러한 각 소비자에 대해 ARN, 상태, 생성일, 모니터링 지표 등의 세부 정보를 볼 수 있습니다.

팬아웃을 사용하도록 등록된 소비자와 해당 상태, 생성일, 지표 등을 콘솔에서 보려면

1. AWS Management Console에 로그인하여 <https://console.aws.amazon.com/kinesis>에서 Kinesis 콘솔을 엽니다.
2. 탐색 창에서 Data Streams(데이터 스트림)를 선택합니다.
3. 세부 정보를 보려는 Kinesis 데이터 스트림을 선택합니다.
4. 스트림의 세부 정보 페이지에서 향상된 팬아웃 탭을 선택합니다.
5. 이름, 상태, 등록일을 보려는 소비자를 선택합니다.

## 소비자 등록을 취소하려면

1. <https://console.aws.amazon.com/kinesis>에서 Kinesis 콘솔을 엽니다.
2. 탐색 창에서 Data Streams(데이터 스트림)를 선택합니다.
3. 세부 정보를 보려는 Kinesis 데이터 스트림을 선택합니다.
4. 스트림의 세부 정보 페이지에서 향상된 팬아웃 탭을 선택합니다.
5. 등록을 해제하려는 각 소비자 이름 왼쪽의 확인란을 선택합니다.
6. 소비자 등록 취소를 선택합니다.

## KCL 1.x에서 KCL 2.x로 소비자 마이그레이션

이 주제에서는 Kinesis Client Library(KCL) 버전 1.x와 버전 2.x 간의 차이점을 설명합니다. 또한, 소비자를 KCL 버전 1.x에서 버전 2.x로 마이그레이션하는 방법을 알려 드립니다. 클라이언트 마이그레이션 후 마지막 체크포인트 위치에서 처리 기록을 시작합니다.

KCL 버전 2.0에서는 다음과 같은 인터페이스 변경이 이루어졌습니다.

### KCL 인터페이스 변경 사항

KCL 1.x 인터페이스	KCL 2.0 인터페이스
<code>com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor</code>	<code>software.amazon.kinesis.processor.ShardRecordProcessor</code>
<code>com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory</code>	<code>software.amazon.kinesis.processor.ShardRecordProcessorFactory</code>
<code>com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware</code>	<code>software.amazon.kinesis.processor.ShardRecordProcessor</code> 에 포함됨

### 주제

- [레코드 프로세서 마이그레이션](#)
- [레코드 프로세서 팩토리 마이그레이션](#)

- [작업자 마이그레이션](#)
- [Amazon Kinesis Client 구성](#)
- [유휴 시간 제거](#)
- [클라이언트 구성 제거](#)

## 레코드 프로세서 마이그레이션

다음은 KCL 1.x에 구현된 레코드 프로세서를 보여주는 예제입니다.

```
package com.amazonaws.kcl;

import com.amazonaws.services.kinesis.clientlibrary.exceptions.InvalidStateException;
import com.amazonaws.services.kinesis.clientlibrary.exceptions.ShutdownException;
import
    com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessorCheckpoint;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import
    com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.ShutdownReason;
import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ShutdownInput;

public class TestRecordProcessor implements IRecordProcessor,
    IShutdownNotificationAware {
    @Override
    public void initialize(InitializationInput initializationInput) {
        //
        // Setup record processor
        //
    }

    @Override
    public void processRecords(ProcessRecordsInput processRecordsInput) {
        //
        // Process records, and possibly checkpoint
        //
    }

    @Override
    public void shutdown(ShutdownInput shutdownInput) {
```

```

        if (shutdownInput.getShutdownReason() == ShutdownReason.TERMINATE) {
            try {
                shutdownInput.getCheckpoint().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                throw new RuntimeException(e);
            }
        }
    }

    @Override
    public void shutdownRequested(IRecordProcessorCheckpoint checkpoint) {
        try {
            checkpoint.checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow exception
            //
            e.printStackTrace();
        }
    }
}

```

## 레코드 프로세서 클래스를 마이그레이션하려면

### 1. 다음과 같이

`com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor` 및 `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware` 인터페이스를 `software.amazon.kinesis.processor.ShardRecordProcessor`로 변경합니다.

```

// import
com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
// import
com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware;
import software.amazon.kinesis.processor.ShardRecordProcessor;

// public class TestRecordProcessor implements IRecordProcessor,
// IShutdownNotificationAware {
public class TestRecordProcessor implements ShardRecordProcessor {

```

### 2. import 및 initialize 메서드의 processRecords 문을 업데이트합니다.

```
// import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;
import software.amazon.kinesis.lifecycle.events.InitializationInput;

//import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
```

3. shutdown 메서드를 새 메서드인 leaseLost, shardEnded, shutdownRequested으로 바꿉니다.

```
// @Override
// public void shutdownRequested(IRecordProcessorCheckpointter checkpointter) {
//     //
//     // This is moved to shardEnded(...)
//     //
//     try {
//         checkpointter.checkpoint();
//     } catch (ShutdownException | InvalidStateException e) {
//         //
//         // Swallow exception
//         //
//         e.printStackTrace();
//     }
// }

@Override
public void leaseLost(LeaseLostInput leaseLostInput) {

}

@Override
public void shardEnded(ShardEndedInput shardEndedInput) {
    try {
        shardEndedInput.checkpointer().checkpoint();
    } catch (ShutdownException | InvalidStateException e) {
        //
        // Swallow the exception
        //
        e.printStackTrace();
    }
}

// @Override
```

```

//     public void shutdownRequested(IRecordProcessorCheckpoint checkpoint) {
//         //
//         // This is moved to shutdownRequested(ShutdownRequestedInput)
//         //
//         try {
//             checkpoint.checkpoint();
//         } catch (ShutdownException | InvalidStateException e) {
//             //
//             // Swallow exception
//             //
//             e.printStackTrace();
//         }
//     }

@Override
public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
    try {
        shutdownRequestedInput.checkpointer().checkpoint();
    } catch (ShutdownException | InvalidStateException e) {
        //
        // Swallow the exception
        //
        e.printStackTrace();
    }
}
}

```

다음은 업데이트된 버전의 레코드 프로세서 클래스입니다.

```

package com.amazonaws.kcl;

import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;
import software.amazon.kinesis.processor.ShardRecordProcessor;

public class TestRecordProcessor implements ShardRecordProcessor {
    @Override
    public void initialize(InitializationInput initializationInput) {

```

```
    }

    @Override
    public void processRecords(ProcessRecordsInput processRecordsInput) {

    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {

    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
        try {
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        try {
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }
}
```

## 레코드 프로세서 팩토리 마이그레이션

레코드 프로세스 팩토리는 리스가 필요할 경우 레코드 프로세서 생성을 담당합니다. 다음은 KCL 1.x 팩토리의 예입니다.



```

package com.amazonaws.kcl;

import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import
  com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;

public class TestRecordProcessorFactory implements IRecordProcessorFactory {
    @Override
    public IRecordProcessor createProcessor() {
        return new TestRecordProcessor();
    }
}

```

## 레코드 프로세서 팩토리를 마이그레이션하려면

### 1. 구현된 인터페이스를 다음과 같이

`com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory`에서 `software.amazon.kinesis.processor.ShardRecordProcessorFactory`로 변경합니다.

```

// import
  com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessor;

// import
  com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

// public class TestRecordProcessorFactory implements IRecordProcessorFactory {
public class TestRecordProcessorFactory implements ShardRecordProcessorFactory {

```

### 2. `createProcessor`에 대한 반환 서명을 변경합니다.

```

// public IRecordProcessor createProcessor() {
public ShardRecordProcessor shardRecordProcessor() {

```

다음은 2.0의 레코드 프로세서 팩토리의 예입니다.

```

package com.amazonaws.kcl;

```

```
import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class TestRecordProcessorFactory implements ShardRecordProcessorFactory {
    @Override
    public ShardRecordProcessor shardRecordProcessor() {
        return new TestRecordProcessor();
    }
}
```

## 작업자 마이그레이션

KCL 버전 2.0에서는 Scheduler라는 새로운 클래스가 Worker 클래스를 대체합니다. 다음은 KCL 1.x 워커의 예입니다.

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = new Worker.Builder()
    .recordProcessorFactory(recordProcessorFactory)
    .config(config)
    .build();
```

작업자를 마이그레이션하려면

1. Worker 클래스에 대한 import 문을 Scheduler 및 ConfigsBuilder 클래스에 대한 가져오기 문으로 변경합니다.

```
// import com.amazonaws.services.kinesis.clientlibrary.lib.worker.Worker;
import software.amazon.kinesis.coordinator.Scheduler;
import software.amazon.kinesis.common.ConfigsBuilder;
```

2. 다음 예와 같이 ConfigsBuilder 및 Scheduler를 생성합니다.

KinesisClientUtil을 사용하여 KinesisAsyncClient를 만들고 KinesisAsyncClient에 maxConcurrency를 구성하는 것이 좋습니다.

**⚠ Important**

Amazon Kinesis Client는 KinesisAsyncClient를 구성하여 KinesisAsyncClient의 전체 임대 수에 더해 추가 사용량까지 허용할 수 있을 만큼 maxConcurrency를 충분히 높이지 않을 경우 지연 시간이 크게 증가할 수 있습니다.

```
import java.util.UUID;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;

...

Region region = Region.AP_NORTHEAST_2;
KinesisAsyncClient kinesisClient =
    KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(region));
DynamoDbAsyncClient dynamoClient =
    DynamoDbAsyncClient.builder().region(region).build();
CloudWatchAsyncClient cloudWatchClient =
    CloudWatchAsyncClient.builder().region(region).build();

ConfigsBuilder configsBuilder = new ConfigsBuilder(streamName, applicationName,
    kinesisClient, dynamoClient, cloudWatchClient, UUID.randomUUID().toString(), new
    SampleRecordProcessorFactory());

Scheduler scheduler = new Scheduler(
    configsBuilder.checkpointConfig(),
    configsBuilder.coordinatorConfig(),
    configsBuilder.leaseManagementConfig(),
    configsBuilder.lifecycleConfig(),
    configsBuilder.metricsConfig(),
    configsBuilder.processorConfig(),
    configsBuilder.retrievalConfig()
);
```

## Amazon Kinesis Client 구성

Kinesis Client Library 2.0 릴리스에서는 클라이언트 구성이 단일 구성 클래스 (KinesisClientLibConfiguration)에서 6개 구성 클래스로 이동했습니다. 다음 표에 마이그레이션에 대한 설명이 나와 있습니다.

### 구성 필드와 새 클래스

원래 필드	새 구성 클래스	설명
applicationName	ConfigsBuilder	이 KCL 애플리케이션의 이름입니다. tableName 및 consumerName 의 기본값으로 사용됩니다.
tableName	ConfigsBuilder	Amazon DynamoDB 리스 테이블에 사용된 테이블 이름 재정의 허용합니다.
streamName	ConfigsBuilder	이 애플리케이션이 레코드를 처리하는 스트림의 이름입니다.
kinesisEndpoint	ConfigsBuilder	이 옵션은 제거되었습니다. 클라이언트 구성 제거를 참조하십시오.
dynamoDBEndpoint	ConfigsBuilder	이 옵션은 제거되었습니다. 클라이언트 구성 제거를 참조하십시오.
initialPositionInStreamExtended	RetrievalConfig	KCL의 레코드 가져오기가 시작되는 샤드의 위치입니다 (애플리케이션의 초기 실행으로 시작).
kinesisCredentialsProvider	ConfigsBuilder	이 옵션은 제거되었습니다. 클라이언트 구성 제거를 참조하십시오.
dynamoDBCredentialsProvider	ConfigsBuilder	이 옵션은 제거되었습니다. 클라이언트 구성 제거를 참조하십시오.

원래 필드	새 구성 클래스	설명
cloudWatchCredentialsProvider	ConfigsBuilder	이 옵션은 제거되었습니다. 클라이언트 구성 제거를 참조하십시오.
failoverTimeMillis	LeaseManagementConfig	리스 소유자가 실패했다고 간주하기 전에 전달해야 하는 시간(밀리초)입니다.
workerIdentifier	ConfigsBuilder	이 애플리케이션 프로세서 인스턴스화를 나타내는 고유 식별자입니다. 고유해야 합니다.
shardSyncIntervalMillis	LeaseManagementConfig	샤드 sync 호출 사이의 시간.
maxRecords	PollingConfig	Kinesis가 반환하는 최대 레코드 수를 설정할 수 있습니다.
idleTimeBetweenReadsInMillis	CoordinatorConfig	이 옵션은 제거되었습니다. 유휴 시간 제거를 참조하십시오.
callProcessRecordsEvenForEmptyRecordList	ProcessorConfig	설정하면, Kinesis에서 제공된 레코드가 없는 경우에도 레코드 프로세서가 직접적으로 호출됩니다.
parentShardPollIntervalMillis	CoordinatorConfig	상위 샤드가 완료되었는지를 확인하기 위해 레코드 프로세서가 폴링을 수행해야 하는 간격입니다.
cleanupLeasesUponShardCompletion	LeaseManagementConfig	설정하면, 하위 리스에서 처리를 시작하자마자 리스가 제거됩니다.

원래 필드	새 구성 클래스	설명
ignoreUnexpectedChildShards	LeaseManagementConfig	설정하면, 진행 중인 샤드가 있는 하위 샤드가 무시됩니다. 이는 주로 DynamoDB Streams용입니다.
kinesisClientConfig	ConfigsBuilder	이 옵션은 제거되었습니다. 클라이언트 구성 제거를 참조하십시오.
dynamoDBClientConfig	ConfigsBuilder	이 옵션은 제거되었습니다. 클라이언트 구성 제거를 참조하십시오.
cloudWatchClientConfig	ConfigsBuilder	이 옵션은 제거되었습니다. 클라이언트 구성 제거를 참조하십시오.
taskBackoffTimeMillis	LifecycleConfig	실패한 작업을 재시도하기 위한 대기 시간.
metricsBufferTimeMillis	MetricsConfig	CloudWatch 측정치 게시를 제어합니다.
metricsMaxQueueSize	MetricsConfig	CloudWatch 측정치 게시를 제어합니다.
metricsLevel	MetricsConfig	CloudWatch 측정치 게시를 제어합니다.
metricsEnabledDimensions	MetricsConfig	CloudWatch 측정치 게시를 제어합니다.
validateSequenceNumberBeforeCheckpointing	CheckpointConfig	이 옵션은 제거되었습니다. 체크포인트 시퀀스 번호 검증을 참조하십시오.

원래 필드	새 구성 클래스	설명
regionName	ConfigsBuilder	이 옵션은 제거되었습니다. 클라이언트 구성 제거를 참조하십시오.
maxLeasesForWorker	LeaseManagementConfig	애플리케이션의 한 인스턴스가 허용해야 하는 최대 리스 수입니다.
maxLeasesToStealAtOneTime	LeaseManagementConfig	애플리케이션이 한 번에 스틸을 시도해야 하는 최대 리스 수입니다.
initialLeaseTableReadCapacity	LeaseManagementConfig	Kinesis Client Library에서 DynamoDB 리스 테이블을 새로 생성해야 하는 경우 사용되는 DynamoDB 읽기 IOP입니다.
initialLeaseTableWriteCapacity	LeaseManagementConfig	Kinesis Client Library에서 DynamoDB 리스 테이블을 새로 생성해야 하는 경우 사용되는 DynamoDB 쓰기 IOP입니다.
initialPositionInStreamExtended	LeaseManagementConfig	애플리케이션이 읽기를 시작해야 하는 스트림 내 초기 위치입니다. 최초 리스 생성 시에만 사용됩니다.
skipShardSyncAtWorkerInitializationIfLeasesExist	CoordinatorConfig	리스 테이블에 기존 리스가 있는 경우 샤드 데이터 동기화를 비활성화합니다. TODO: KinesisEco-438
shardPrioritization	CoordinatorConfig	사용할 샤드 우선 순위.
shutdownGraceMillis	해당 사항 없음	이 옵션은 제거되었습니다. MultiLang 제거를 참조하십시오.

원래 필드	새 구성 클래스	설명
timeoutInSeconds	해당 사항 없음	이 옵션은 제거되었습니다. MultiLang 제거를 참조하십시오.
retryGetRecordsInSeconds	PollingConfig	실패에 대한 GetRecords 시도 사이의 지연을 구성합니다.
maxGetRecordsThreadPool	PollingConfig	GetRecords에 사용되는 스레드 풀 크기.
maxLeaseRenewalThreads	LeaseManagementConfig	리스 갱신 스레드 풀의 크기를 제어합니다. 애플리케이션에서 사용할 수 있는 리스가 많을수록 이 풀의 크기가 커야 합니다.
recordsFetcherFactory	PollingConfig	스트림에서 검색하는 페처를 생성하는 데 사용되는 팩토리를 교체할 수 있습니다.
logWarningForTaskAfterMillis	LifecycleConfig	작업이 완료되지 않은 경우 얼마나 대기한 후 경고가 기록될지를 지정합니다.
listShardsBackoffTimeInMillis	RetrievalConfig	실패 발생 시 ListShards 호출 간에 대기하는 시간(밀리초)입니다.
maxListShardsRetryAttempts	RetrievalConfig	포기하기 전에 ListShards 가 재시도하는 최대 횟수입니다.

## 유휴 시간 제거

KCL 1.x 버전에서 `idleTimeBetweenReadsInMillis`는 다음 두 가지 수량에 해당합니다.



- 작업 디스패칭 확인 간의 시간. 작업 간의 이 시간은 이제 `CoordinatorConfig#shardConsumerDispatchPollIntervalMillis`를 설정하여 구성할 수 있습니다.
- Kinesis Data Streams에서 반환하는 레코드가 없는 경우 절전 모드로 들어가는 시간. 버전 2.0에서는 향상된 팬아웃 레코드가 해당 검색자로부터 푸시됩니다. 샤드 소비자에 대한 작업은 푸시된 요청이 도착한 경우에만 발생합니다.

## 클라이언트 구성 제거

버전 2.0에서 KCL은 더 이상 클라이언트를 생성하지 않습니다. 이제 사용자가 유효한 클라이언트를 제공해야 합니다. 따라서 클라이언트 생성을 제어하던 모든 구성 파라미터는 삭제되었습니다. 이러한 파라미터가 필요한 경우 `ConfigsBuilder`에 클라이언트를 제공하기 전에 클라이언트에서 설정할 수 있습니다.

제거된 필드	동일 구성
<code>kinesisEndpoint</code>	SDK <code>KinesisAsyncClient</code> 를 원하는 엔드포인트 <code>KinesisAsyncClient.builder().endpointOverride(URI.create("https://&lt;kinesis endpoint&gt;")).build()</code> 로 구성합니다.
<code>dynamoDBEndpoint</code>	SDK <code>DynamoDbAsyncClient</code> 를 원하는 엔드포인트 <code>DynamoDbAsyncClient.builder().endpointOverride(URI.create("https://&lt;dynamodb endpoint&gt;")).build()</code> 로 구성합니다.
<code>kinesisClientConfiguration</code>	SDK <code>KinesisAsyncClient</code> 를 필요한 구성 <code>KinesisAsyncClient.builder().overrideConfiguration(&lt;your configuration&gt;).build()</code> 로 구성합니다.
<code>dynamoDBClientConfiguration</code>	SDK <code>DynamoDbAsyncClient</code> 를 필요한 구성 <code>DynamoDbAsyncClient.builder().overrideConfiguration(&lt;your configuration&gt;).build()</code> 로 구성합니다.
<code>cloudWatchClientConfiguration</code>	SDK <code>CloudWatchAsyncClient</code> 를 필요한 구성 <code>CloudWatchAsyncClient.builder().overrideConfiguration(&lt;your configuration&gt;).build()</code> 로 구성합니다.

제거된 필드	동일 구성
regionName	SDK를 원하는 리전으로 구성합니다. 모든 SDK 클라이언트에 대해 동일합니다. 예: <code>KinesisAsyncClient.builder().region(Region.US_WEST_2).build()</code> .

## 다른 AWS 서비스를 사용하여 Kinesis Data Streams에서 데이터 읽기

다음은 Kinesis Data Streams 데이터를 읽기 위해 Kinesis Data Streams와 직접 통합할 수 있는 다른 AWS 서비스의 목록입니다.

주제

- [Amazon EMR 사용](#)
- [아마존 EventBridge 파이프 사용](#)
- [AWS Glue 사용](#)
- [Amazon Redshift 사용](#)

### Amazon EMR 사용

Amazon EMR 클러스터는 하이브, 피그 MapReduce, 하둡 스트리밍 API, 캐스케이딩과 같은 하둡 에코시스템의 친숙한 도구를 사용하여 Amazon Kinesis 스트림을 직접 읽고 처리할 수 있습니다. 실행 중인 클러스터에서 Amazon S3, Amazon DynamoDB 및 HDFS의 기존 데이터와 Amazon Kinesis의 실시간 데이터를 조인할 수도 있습니다. 사후 처리 활동을 위해 Amazon EMR에서 Amazon S3 또는 DynamoDB로 데이터를 직접 로드할 수 있습니다.

자세한 내용은 Amazon EMR 릴리스 안내서의 [Amazon Kinesis](#)를 참조하세요.

### 아마존 EventBridge 파이프 사용

Amazon EventBridge Pipes는 아마존 Kinesis Data Streams를 소스로 지원합니다. Amazon EventBridge Pipes를 사용하면 선택적 변환, 필터링 및 강화 단계를 통해 이벤트 생산자와 소비자 간의 point-to-point 통합을 생성할 수 있습니다. EventBridge 파이프를 사용하여 Kinesis Data Stream에서 레코드를 수신하고 Kinesis Data Streams를 포함하여 처리할 수 있는 대상 중 하나로 보내기 전에 선택적으로 이러한 레코드를 필터링하거나 개선할 수 있습니다.

자세한 내용은 Amazon EventBridge 릴리스 가이드의 [Amazon Kinesis 스트림을 소스로 참조하십시오](#)입니다.

## AWS Glue 사용

AWS Glue 스트리밍 ETL을 사용하여 지속적으로 실행되고 Amazon Kinesis Data Streams의 데이터를 사용하는 스트리밍 추출, 전환, 적재(ETL) 작업을 생성할 수 있습니다. 작업은 데이터를 정리 및 변환한 다음 결과를 Amazon S3 데이터 레이크 또는 JDBC 데이터 스토어에 로드합니다.

자세한 내용은 AWS Glue 사용 설명서의 [AWS Glue에서 스트리밍 ETL 작업을 참조하십시오](#)입니다.

## Amazon Redshift 사용

Amazon Redshift는 Amazon Kinesis Data Streams의 스트리밍 수집을 지원합니다. Amazon Redshift 스트리밍 수집 기능은 지연 시간이 짧고 빠른 속도로 Amazon Kinesis Data Streams에서 Amazon Redshift 구체화된 뷰로 스트리밍 데이터를 수집할 수 있습니다. Amazon Redshift 스트리밍 수집을 사용하면 Amazon Redshift로 수집하기 전에 Amazon S3에서 데이터를 준비할 필요가 없습니다.

자세한 내용은 Amazon Redshift 데이터베이스 개발자 안내서의 [스트리밍 수집](#)을 참조하십시오.

## 타사 통합 사용

Kinesis Data Streams와 통합되는 다음 타사 옵션 중 하나를 사용하여 Amazon Kinesis 데이터 스트림에서 데이터를 읽을 수 있습니다.

### 주제

- [Apache Flink](#)
- [Adobe Experience Platform](#)
- [Apache Druid](#)
- [Apache Spark](#)
- [Databricks](#)
- [Kafka Confluent 플랫폼](#)
- [Kinesumer](#)
- [Talend](#)

## Apache Flink

Apache Flink는 무제한 및 제한 데이터 스트림에 대한 상태 저장 계산을 위한 프레임워크 및 분산 처리 엔진입니다. Apache Flink를 사용하여 Kinesis Data Streams를 소비하는 방법에 대한 자세한 내용은 [Amazon Kinesis Data Streams Connector](#)를 참조하세요.

## Adobe Experience Platform

Adobe Experience Platform을 통해 조직은 모든 시스템의 고객 데이터를 중앙 집중화하고 표준화할 수 있습니다. 그런 다음 데이터 과학과 기계 학습을 적용하여 풍부하고 개인화된 경험의 설계와 전송을 획기적으로 개선합니다. Adobe Experience 플랫폼을 사용하여 Kinesis 데이터 스트림을 사용하는 방법에 대한 자세한 내용은 [Amazon Kinesis](#) 커넥터를 참조하십시오.

## Apache Druid

Druid는 스트리밍 및 배치 데이터에 대한 1초 미만의 쿼리를 규모와 부하에 맞게 전송하는 고성능 실시간 분석 데이터베이스입니다. [Apache Druid를 사용하여 Kinesis 데이터 스트림을 수집하는 방법에 대한 자세한 내용은 Amazon Kinesis 통합을 참조하십시오.](#)

## Apache Spark

Apache Spark는 대규모 데이터 처리를 위한 통합 분석 엔진으로서, Java, Scala, Python 및 R의 고급 API와 일반 실행 그래프를 지원하는 최적화된 엔진을 제공합니다. Apache Spark를 사용하여 Kinesis 데이터 스트림의 데이터를 사용하는 스트림 처리 애플리케이션을 구축할 수 있습니다.

[Apache Spark 구조적 스트리밍을 사용하여 Kinesis 데이터 스트림을 사용하려면 Amazon Kinesis 데이터 스트림 커넥터를 사용하십시오.](#) 이 커넥터는 샤드당 초당 최대 2MB의 데이터 전용 읽기 처리량을 애플리케이션에 제공하는 향상된 팬아웃을 통한 사용을 지원합니다. 자세한 내용은 [전용 처리량을 갖춘 사용자 지정 소비자 개발 \(향상된 팬아웃\)](#) 을 참조하십시오.

스파크 스트리밍을 사용하여 Kinesis 데이터 스트림을 사용하려면 [Spark 스트리밍 + Kinesis 통합](#)을 참조하십시오.

## Databricks

Databricks는 데이터 엔지니어링, 데이터 과학 및 기계 학습을 위한 협업 환경을 제공하는 클라우드 기반 플랫폼입니다. Databricks를 사용하여 Kinesis 데이터 스트림을 사용하는 방법에 대한 자세한 내용은 [Amazon Kinesis](#)에 연결을 참조하십시오.

## Kafka Confluent 플랫폼

Confluent Platform은 Kafka를 기반으로 구축되었으며 기업이 실시간 데이터 파이프라인과 스트리밍 애플리케이션을 구축하고 관리하는 데 도움이 되는 추가 기능을 제공합니다. Confluent 플랫폼을 사용하여 Kinesis 데이터 스트림을 사용하는 방법에 대한 자세한 내용은 Confluent 플랫폼용 [Amazon Kinesis 소스 커넥터](#)를 참조하십시오.

## Kinesumer

Kinesumer는 Kinesis 데이터 스트림을 위한 클라이언트 측 분산 소비자 그룹 클라이언트를 구현하는 Go 클라이언트입니다. 자세한 내용은 [Kinesumer GitHub 리포지토리](#)를 참조하세요.

## Talend

Talend는 사용자가 확장 가능하고 효율적인 방식으로 다양한 소스의 데이터를 수집, 변환 및 연결할 수 있는 데이터 통합 및 관리 소프트웨어입니다. Talend를 사용하여 Kinesis 데이터 스트림을 사용하는 방법에 대한 자세한 내용은 [Amazon Kinesis 스트림에 텔런드 연결](#)을 참조하십시오.

## Kinesis 데이터 스트림 소비자 문제 해결

다음 섹션에서는 Amazon Kinesis Data Streams 소비자 작업 시 발견할 수 있는 몇 가지 일반적인 문제의 해결 방법을 제공합니다.

- [Kinesis Client Library를 사용할 때 일부 Kinesis Data Streams 레코드를 건너뛰는 경우](#)
- [같은 샤드에 속한 레코드가 동시에 여러 레코드 프로세서에 의해 처리되는 경우](#)
- [소비자 애플리케이션이 예상보다 느린 속도로 읽는 경우](#)
- [GetRecords 스트림에 데이터가 있는 경우에도 빈 레코드 배열을 반환합니다.](#)
- [샤드 반복자가 예기치 않게 완료되는 경우](#)
- [소비자 레코드 처리 속도가 느려지는 경우](#)
- [무단 KMS 마스터 키 권한 오류](#)
- [소비자를 위한 일반적인 문제, 질문 및 문제 해결 아이디어](#)

## Kinesis Client Library를 사용할 때 일부 Kinesis Data Streams 레코드를 건너뛰는 경우

레코드를 건너뛰는 가장 일반적인 원인은 processRecords에서 발생한 예외가 처리되지 않았기 때문입니다. Kinesis Client Library(KCL)는 processRecords 코드를 사용하여 데이터 레코드를 처리할

때 발생하는 예외를 처리합니다. `processRecords`에서 발생한 모든 예외는 KCL에 흡수됩니다. 실패가 반복될 때 무제한 재시도를 방지하기 위해 KCL은 예외가 발생할 때 처리되는 레코드 배치를 재전송하지 않습니다. 그런 다음 KCL은 레코드 프로세서를 다시 시작하지 않고 다음 번 데이터 레코드 배치에 대해 `processRecords`를 직접적으로 호출합니다. 그래서 소비자 애플리케이션이 건너뛴 레코드를 효과적으로 관찰합니다. 레코드를 건너뛰지 않도록 하려면 `processRecords` 내의 모든 예외를 적절하게 처리하십시오.

## 같은 샤드에 속한 레코드가 동시에 여러 레코드 프로세서에 의해 처리되는 경우

실행 중인 Kinesis Client Library(KCL) 애플리케이션의 경우 샤드에 하나의 소유자만 있습니다. 그러나 여러 레코드 프로세서가 같은 샤드를 임시로 처리할 수 있습니다. 네트워크 연결이 끊어지는 워커 인스턴스의 경우 KCL은 장애 조치 시간이 만료된 후 연결할 수 없는 워커가 더 이상 레코드를 처리하지 않는다고 가정하고 다른 워커 인스턴스가 대체하도록 지시합니다. 잠시동안 새로운 레코드 프로세서와 연결할 수 없는 작업자의 레코드 프로세서가 같은 샤드의 데이터를 처리할 수 있습니다.

애플리케이션에 적합한 장애 조치 시간을 설정해야 합니다. 지연 시간이 짧은 애플리케이션이라면 최대 대기할 시간이 기본값 10초로 충분하지만 연결이 자주 끊어지는 지역에서 호출하는 경우처럼 연결 문제가 예상된다면 이 시간이 너무 짧습니다.

특히 전에 연결할 수 없었던 작업자로 네트워크 연결이 주로 복원되기 때문에 애플리케이션이 이 시나리오를 예측하고 처리해야 합니다. 레코드 프로세서가 다른 레코드 프로세서로 샤드를 인계하면 다음 두 가지 경우를 처리해야 정상적으로 종료됩니다.

1. 현재 `processRecords` 호출이 완료된 후 KCL은 종료 이유 'ZOMBIE'로 레코드 프로세서에서 종료 메시지를 간접적으로 호출합니다. 레코드 프로세서는 모든 리소스를 적절하게 정리한 후 종료되어야 합니다.
2. 'zombie' 워커에서 체크포인트를 수행하려고 시도하면 KCL에서 `ShutdownException`이 발생합니다. 이 예외를 받은 후 코드가 현재 메시지를 확실하게 종료해야 합니다.

자세한 정보는 [중복 레코드 처리](#)를 참조하세요.

## 소비자 애플리케이션이 예상보다 느린 속도로 읽는 경우

읽기 처리량의 속도가 예상보다 느린 가장 일반적인 이유는 다음과 같습니다.

1. 여러 소비자 애플리케이션의 읽기 합계가 샤드당 제한을 초과합니다. 자세한 정보는 [할당량과 제한](#)을 참조하세요. 이 경우 Kinesis 데이터 스트림의 샤드 수를 늘립니다.

- 호출당 최대 GetRecords 수를 지정하는 [제한](#)이 낮은 값으로 구성되었을 수 있습니다. KCL을 사용하고 있다면 작업자의 maxRecords 속성을 낮은 값으로 구성한 것일 수도 있습니다. 일반적으로 이 속성에 시스템 기본값을 사용하는 것이 좋습니다.
- 여러 가지 가능한 이유로 processRecords 호출에 포함된 로직이 예상보다 오래 걸릴 수 있습니다. 로직이 CPU 집약적, I/O 차단 또는 동기화에서 병목 현상이 발생한 것일 수 있습니다. 이 경우에 해당하는지 테스트하려면 빈 레코드 프로세서를 테스트 실행하고 읽기 처리량을 비교하십시오. 수신 데이터를 따라 잡는 방법에 대한 자세한 내용은 [리샤딩, 크기 조정 및 병렬 처리](#)를 참조하십시오.

소비자 애플리케이션이 하나뿐이면 쓰기 속도보다 항상 최소한 2배 더 빠르게 읽을 수 있습니다. 쓰기를 위해 초당 최대 1,000개의 레코드를 쓸 수 있고 최대 총 데이터 쓰기 속도는 초당 1MB(파티션 키 포함)이기 때문입니다. 열린 각 샤드는 읽기에 대해 초당 최대 5개의 트랜잭션을 지원할 수 있으며, 최대 총 데이터 읽기 속도는 초당 2MB입니다. 각 읽기(GetRecords 호출)는 레코드 배치를 가져옵니다. GetRecords가 반환하는 데이터 크기는 샤드 사용률에 따라 다릅니다. GetRecords가 반환할 수 있는 최대 데이터 크기는 10MB입니다. 호출이 그 한도를 반환하면 다음 5초 안에 이루어지는 호출에서 ProvisionedThroughputExceededException이 발생합니다.

## GetRecords 스트림에 데이터가 있는 경우에도 빈 레코드 배열을 반환합니다.

레코드를 소비하거나 가져오는 것은 가져오기(pull) 모델입니다. 개발자는 백오프 없이 연속 루프로 [GetRecords](#) 호출해야 합니다. 또한 모든 GetRecords 호출은 다음 루프 반복에서 사용해야 하는 ShardIterator 값을 반환합니다.

GetRecords 작업은 차단하지 않습니다. 관련 데이터 레코드나 빈 Records 요소를 즉시 반환합니다. 다음 두 가지 조건에서 빈 Records 요소가 반환됩니다.

- 현재 샤드에 데이터가 더 이상 없습니다.
- ShardIterator가 가리키는 샤드 부분 근처에 데이터가 없습니다.

두 번째 조건은 미미한 편이지만 레코드를 검색할 때 무한 탐색 시간(지연 시간)을 피하기 위해 설계상 필요한 사항입니다. 따라서 스트림을 소비하는 애플리케이션은 GetRecords를 반복하고 호출하여 당연히 비어 있는 레코드를 처리해야 합니다.

프로덕션 시나리오에서는 NextShardIterator 값이 NULL이어야 연속 루프가 종료됩니다. NextShardIterator가 NULL이면 현재 샤드가 닫혀 있고 ShardIterator 값이 마지막 레코드를 지나 다른 항목을 가리킵니다. 소비하는 애플리케이션이 SplitShard 또는 MergeShards를 호출하지 않

으면 샤드는 계속 열려 있으며 GetRecords 호출은 NULL인 NextShardIterator 값을 반환하지 않습니다.

Kinesis Client Library(KCL)를 사용하는 경우 위의 소비 패턴이 추상화됩니다. 동적으로 변화하는 샤드 세트의 자동 처리가 여기에 포함됩니다. KCL에서 개발자는 수신 레코드를 처리하기 위한 로직만 제공합니다. 이러한 상태는 라이브러리가 GetRecords를 연속으로 호출하기 때문에 가능합니다.

## 샤드 반복자가 예기치 않게 만료되는 경우

모든 GetRecords 요청에서 새로운 반복자가 NextShardIterator로 반환되며, 다음 GetRecords 요청에서 이 샤드 반복자를 ShardIterator로 사용합니다. 대개 이 샤드 반복자는 사용하기 전에 만료되지 않지만 하지만 5분 이상 GetRecords를 호출하지 않거나 소비자 애플리케이션의 다시 시작을 수행하면 샤드 반복자가 만료될 수 있습니다.

사용하기 전에 샤드 반복자가 즉시 만료되는 경우 Kinesis에 사용된 DynamoDB 테이블에 리스 데이터를 저장할 용량이 충분하지 않을 수 있습니다. 샤드 수가 많으면 이런 문제가 생기기 쉽습니다. 이 문제를 해결하려면 샤드 테이블에 할당된 쓰기 용량을 늘리십시오. 자세한 정보는 [리스 테이블을 사용하여 KCL 소비자 애플리케이션에서 처리한 샤드 추적](#)을 참조하세요.

## 소비자 레코드 처리 속도가 느려지는 경우

대다수 사용 사례에서 소비자 애플리케이션은 스트림에서 최신 데이터를 읽습니다. 바람직하지는 않지만 상황에 따라 소비자가 읽는 속도가 느려지기도 합니다. 소비자가 얼마나 느리게 읽는지 파악한 후에 속도가 느려지는 가장 일반적인 이유를 알아보십시오.

스트림의 모든 샤드와 소비자에서 읽기 위치를 추적하는 GetRecords.IteratorAgeMilliseconds 측정치를 시작합니다. 반복자 수명이 보존 기간(기본적으로 24시간, 최대 365일까지 구성 가능)의 50%를 경과하면 레코드 만료로 인해 데이터가 손실될 위험이 있음을 알아 두세요. 보존 기간을 늘리는 것이 신속한 임시 조치입니다. 그러면 문제를 해결하는 동안 중요한 데이터가 손실되지 않습니다. 자세한 정보는 [아마존을 통한 아마존 Kinesis 데이터 스트림 서비스 모니터링 CloudWatch](#)을 참조하세요. 다음으로 Kinesis Client Library (KCL)에서 내보낸 사용자 지정 CloudWatch 메트릭을 사용하여 소비자 애플리케이션이 각 샤드를 읽는 데 얼마나 뒤쳐져 있는지 확인합니다. MillisBehindLatest 자세한 정보는 [Amazon을 통한 Kinesis 클라이언트 라이브러리 모니터링 CloudWatch](#)을 참조하세요.

다음은 소비자 속도가 느려지는 가장 일반적인 이유입니다.

- GetRecords.IteratorAgeMilliseconds 또는 MillisBehindLatest가 갑자기 크게 증가하면 대개 다운스트림 애플리케이션에 API 작업 실패와 같은 일시적인 문제가 있는 것입니다. 두 측정치 중 하나가 지속적으로 이 동작을 나타내면 갑작스러운 증가를 조사해야 합니다.



- 이 측정치가 점차 증가하면 소비자가 충분히 빠른 속도로 레코드를 처리하지 않아 스트림을 따라 잡지 못하는 것입니다. 이 동작이 나타나는 가장 일반적인 근본 원인은 물리적 리소스가 부족하거나 레코드 처리 로직이 스트림 처리량의 증가에 따라 조정되지 않기 때문입니다. KCL이 `processTask` 작업과 관련하여 내보내는 다른 사용자 지정 CloudWatch 지표 (`,,` 등) 를 살펴보면 이 동작을 확인할 수 있습니다. `RecordProcessor.processRecords.Time Success RecordsProcessed`
- 증가한 처리량과 상관 관계가 있는 `processRecords.Time` 측정치의 증가가 발견되면 레코드 처리 로직을 분석하여 이 로직이 증가한 처리량에 따라 조정되지 않는 이유를 파악해야 합니다.
- 증가한 처리량과 상관 관계가 없는 `processRecords.Time` 값의 증가가 발견되면 중요한 경로에서 차단 호출이 이루어지고 있는지 확인하십시오. 차단 호출은 종종 레코드 처리가 종료되는 원인입니다. 샤드 수를 늘려 병렬 처리를 늘리는 것도 또 다른 방법입니다. 마지막으로, 피크 수요 중 기본 처리 노드에 충분한 양의 물리적 리소스(메모리, CPU 사용률 등)가 있는지 확인합니다.

## 무단 KMS 마스터 키 권한 오류

소비자 애플리케이션이 KMS 마스터 키에 대한 권한 없이 암호화된 스트림에서 읽으면 이 오류가 발생합니다. KMS 키에 대한 액세스 권한을 애플리케이션에 할당하려면 [AWS KMS에서 키 정책 사용 및 AWS KMS에서 IAM 정책 사용](#)을 참조하세요.

## 소비자를 위한 일반적인 문제, 질문 및 문제 해결 아이디어

- [Kinesis Data Streams 트리거로 Lambda 함수를 간접적으로 호출할 수 없는 이유는 무엇인가요?](#)
- [Kinesis Data Streams에서 ReadProvisionedThroughputExceeded 예외를 감지하고 문제를 해결하려면 어떻게 해야 하나요?](#)
- [Kinesis Data Streams에서 긴 지연 시간 문제가 발생하는 이유는 무엇입니까?](#)
- [Kinesis 데이터 스트림이 500 내부 서버 오류를 반환하는 이유는 무엇인가요?](#)
- [Kinesis Data Streams용 KCL 애플리케이션의 차단 또는 중단 문제를 해결하려면 어떻게 해야 하나요?](#)
- [동일한 Amazon DynamoDB 테이블로 다른 Amazon Kinesis Client Library 애플리케이션을 사용하면 어떻게 해야 하나요?](#)

## Amazon Kinesis Data Streams 소비자를 위한 고급 주제

Amazon Kinesis Data Streams 소비자를 최적화하는 방법을 알아보세요.

### 내용

- [짧은 처리 지연 시간](#)
- [Kinesis 프로듀서 AWS Lambda 라이브러리와 함께 사용](#)
- [리샤딩, 크기 조정 및 병렬 처리](#)
- [중복 레코드 처리](#)
- [시작, 종료 및 조절 처리](#)

## 짧은 처리 지연 시간

전파 지연은 레코드가 스트림에 기록된 순간부터 소비자 애플리케이션에서 읽을 때까지의 end-to-end 지연 시간으로 정의됩니다. 이 지연은 여러 요인에 따라 달라지만 주로 소비자 애플리케이션 폴링 간격의 영향을 받습니다.

대다수 애플리케이션에서 애플리케이션마다 각 샤드를 초당 1회씩 폴링하는 것이 좋습니다. 그러면 Amazon Kinesis Data Streams의 초당 5회 GetRecords 호출 제한을 초과하지 않고 여러 소비자 애플리케이션에서 동시에 스트림을 처리할 수 있습니다. 또한 더 큰 데이터 배치를 처리하는 것이 네트워크와 애플리케이션의 다른 다운스트림 지연 시간을 줄이는 데 보다 효율적입니다.

1초 마다 폴링하는 모범 사례에 따라 KCL 기본값이 설정됩니다. 이 기본값을 사용하면 일반적으로 평균 전파 지연이 1초 미만입니다.

Kinesis Data Streams 레코드를 쓴 후 즉시 읽을 수 있습니다. 이 점을 이용해 데이터를 사용할 수 있게 되면 즉시 스트림에서 데이터를 소비해야 하는 몇몇 사용 사례가 있습니다. 다음 예제와 같이 더 자주 폴링하도록 KCL 기본 설정을 재정의하여 전파 지연을 크게 줄일 수 있습니다.

Java KCL 구성 코드는 다음과 같습니다.

```
kinesisClientLibConfiguration = new
    KinesisClientLibConfiguration(applicationName,
        streamName,
        credentialsProvider,

workerId).withInitialPositionInStream(initialPositionInStream).withIdleTimeBetweenReadsInMilli
```

Python 및 Ruby KCL의 속성 파일 설정은 다음과 같습니다.

```
idleTimeBetweenReadsInMillis = 250
```

**Note**

Kinesis Data Streams에는 샤드별로 초당 5회의 GetRecords 호출 제한이 있으므로 `idleTimeBetweenReadsInMillis` 속성을 200ms보다 낮게 설정하면 애플리케이션에 `ProvisionedThroughputExceededException` 예외가 관찰될 수 있습니다. 이 예외가 너무 많으면 지수 백오프가 발생하여 예기치 않게 처리가 많이 지연됩니다. 이 속성을 200ms 또는 더 높게 설정하고 처리 애플리케이션이 2개 이상인 경우 유사한 조절이 발생합니다.

## Kinesis 프로듀서 AWS Lambda 라이브러리와 함께 사용

[Kinesis Producer Library\(KPL\)](#)는 사용자 형식의 작은 레코드를 최대 1MB의 큰 레코드로 집계하여 Amazon Kinesis Data Streams 처리량을 효율적으로 활용할 수 있게 합니다. Java용 KCL은 이러한 레코드의 디애그리게이션을 지원하지만 스트림의 소비자로서 사용하는 AWS Lambda 경우 레코드 집계를 해제하려면 특수 모듈을 사용해야 합니다. Lambda용 [Kinesis 프로듀서 라이브러리 집계 해제 GitHub 모듈에서](#) 필요한 프로젝트 코드와 지침을 얻을 수 있습니다. AWS 이 프로젝트의 구성 요소를 사용하면 Java AWS Lambda, Node.js 및 Python 내에서 KPL 직렬화된 데이터를 처리할 수 있습니다. [다중 언어 KCL 애플리케이션](#)의 일부로 이러한 구성 요소를 사용할 수도 있습니다.

## 리샤딩, 크기 조정 및 병렬 처리

리샤딩을 사용하면 스트림을 통과하는 데이터의 속도 변화에 맞게 스트림의 샤드 수를 늘리거나 줄일 수 있습니다. 일반적으로 샤드 데이터 처리 측정치를 모니터링하는 관리 애플리케이션에서 리샤딩을 수행합니다. KCL 자체는 리샤딩 작업을 시작하지 않지만 리샤딩으로 인한 샤드 수 변화에 맞게 조정됩니다.

[리스 테이블을 사용하여 KCL 소비자 애플리케이션에서 처리한 샤드 추적](#)에서 언급한 대로 KCL은 Amazon DynamoDB 테이블을 사용하여 스트림의 샤드를 추적합니다. 리샤딩으로 인해 새 샤드가 생성되면 KCL이 새로운 샤드를 찾아 테이블의 새 행을 채웁니다. 작업자가 자동으로 새로운 샤드를 찾고 프로세서를 생성하여 데이터를 처리합니다. KCL은 사용 가능한 모든 워커 및 레코드 프로세서에 스트림의 샤드를 배포합니다.

KCL에서는 리샤딩 전에 샤드에 있던 모든 데이터가 먼저 처리됩니다. 해당 데이터가 처리된 후 새로운 샤드의 데이터가 레코드 프로세서로 전송됩니다. KCL에서는 이 방식으로 특정 파티션 키의 스트림에 데이터 레코드가 추가되는 순서를 유지합니다.

### 예: 리샤딩, 크기 조정 및 병렬 처리

다음 예에서는 KCL을 사용하여 크기 조정 및 리샤딩을 처리하는 방법을 보여줍니다.

- 예를 들어, 애플리케이션이 하나의 EC2 인스턴스에서 실행 중이고 4개의 샤드가 있는 Kinesis 데이터 스트림 하나를 처리하는 경우를 생각해 봅시다. 이 인스턴스 하나는 KCL 워커 하나와 레코드 프로세서 4개(샤드마다 레코드 프로세서 1개)가 있습니다. 이 레코드 프로세서 4개가 같은 프로세스에서 병렬로 실행됩니다.
- 그 다음 다른 인스턴스를 사용하기 위해 애플리케이션 크기를 조정하면 인스턴스 2개에서 샤드가 4개인 스트림 하나를 처리합니다. KCL 워커가 두 번째 인스턴스에서 시작될 때 첫 번째 인스턴스와 로드 밸런싱하여 각 인스턴스가 샤드 2개를 처리합니다.
- 샤드 4개를 5개로 분할합니다. KCL은 인스턴스 하나가 샤드 3개를 처리하고 나머지 인스턴스가 2개를 처리하도록 인스턴스 간 처리를 다시 조정합니다. 샤드를 병합할 때도 비슷하게 조정됩니다.

일반적으로 KCL을 사용할 때 인스턴스 수가 샤드 수를 초과하지 않도록 해야 합니다(장애 대기 용도 제외). 각 샤드는 정확히 KCL 워커 하나에 의해 처리되며 해당하는 레코드 프로세서가 하나만 있으므로 샤드 하나를 처리하기 위해 여러 인스턴스가 필요하지 않습니다. 그러나 한 작업자가 처리할 수 있는 샤드 수에는 제한이 없으므로 샤드 수가 인스턴스 수를 초과해도 괜찮습니다.

애플리케이션에서 처리를 확장하려면 다음 방법을 조합해 테스트해야 합니다.

- 인스턴스 크기 늘리기(모든 레코드 프로세서가 프로세스에서 병렬로 실행되기 때문)
- 최대 열린 샤드 수까지 인스턴스 수 늘리기(샤드를 독립적으로 처리할 수 있기 때문)
- 샤드 수 늘리기(병렬 처리 수준 향상)

Auto Scaling을 사용하여 적절한 지표에 따라 자동으로 인스턴스 크기를 조정할 수 있습니다. 자세한 내용은 [Amazon EC2 Auto Scaling 사용 설명서](#)를 참조하세요.

리샤딩으로 스트림의 샤드 수를 늘리면 해당 레코드 프로세서의 수가 증가하여 이를 호스팅하는 EC2 인스턴스의 로드가 커집니다. 인스턴스가 Auto Scaling 그룹의 일부이고 로드가 충분히 커지면 Auto Scaling 그룹이 인스턴스를 추가하여 증가한 로드를 처리합니다. 시작할 때 Amazon Kinesis Data Streams 애플리케이션을 실행하도록 인스턴스를 구성하여 추가 워커 및 레코드 프로세서를 새로운 인스턴스에서 즉시 활성화해야 합니다.

리샤딩에 대한 자세한 내용은 [스트림 리샤딩](#)을 참조하십시오.

## 중복 레코드 처리

Amazon Kinesis Data Streams 애플리케이션에 레코드가 두 번 이상 전송되는 두 가지 주된 이유는 생산자 재시도 및 소비자 재시도입니다. 애플리케이션은 개별 레코드가 여러 번 처리될 것을 예상하고 이 문제를 적절하게 처리해야 합니다.

## 생산자 재시도

PutRecord를 직접적으로 호출한 후 Amazon Kinesis Data Streams에서 승인을 받기 전에 생산자에서 네트워크 관련 시간 초과가 발생한다고 생각해 보세요. 생산자는 레코드가 Kinesis Data Streams에 전송되었는지 확실히 알 수 없습니다. 모든 레코드가 애플리케이션에 중요하다는 가정하에, 동일한 데이터로 호출을 재시도하도록 생산자가 작성되었습니다. 동일한 데이터에 대한 2개의 PutRecord 호출이 모두 성공적으로 Kinesis Data Streams에 커밋되면 Kinesis Data Streams 레코드 2개가 생깁니다. 두 레코드의 데이터가 같아도 시퀀스 번호는 고유합니다. 중복을 철저히 방지해야 하는 애플리케이션은 처리할 때 중복 항목을 제거하기 위해 레코드에 기본 키를 포함해야 합니다. 생산자 재시도로 인한 중복 수는 대개 소비자 재시도로 인한 중복 수보다 적습니다.

### Note

SDK를 사용하는 경우 AWS SDK *PutRecord* 및 도구 사용 설명서에서 SDK [재시도 동작](#)에 AWS 대해 알아보세요.

## 소비자 재시도

레코드 프로세서가 다시 시작될 때 소비자(데이터 처리 애플리케이션) 재시도가 발생합니다. 다음과 같은 경우 동일한 샤드의 레코드 프로세서가 다시 시작됩니다.

1. 작업자가 예기치 않게 종료된 경우
2. 작업자 인스턴스가 추가 또는 제거된 경우
3. 샤드가 병합 또는 분할된 경우
4. 애플리케이션이 배포된 경우

이 모든 경우에 shards-to-worker-to-record-processor 매핑은 로드 밸런싱 처리를 위해 지속적으로 업데이트됩니다. 다른 인스턴스로 마이그레이션된 샤드 프로세서가 마지막 체크포인트에서 레코드 처리를 다시 시작합니다. 그 결과 아래의 예와 같이 중복된 레코드가 처리됩니다. 로드 밸런싱에 대한 자세한 내용은 [리샤딩, 크기 조정 및 병렬 처리](#)를 참조하십시오.

예: 다시 전달된 레코드로 인한 소비자 재시도

이 예에서 애플리케이션이 스트림에서 지속적으로 레코드를 읽고 로컬 파일로 레코드를 집계하며 파일을 Amazon S3에 업로드합니다. 간단한 설명을 위해 샤드 1개와 샤드를 처리하는 작업자 1개가 있다고 가정해 보겠습니다. 마지막 체크포인트가 레코드 번호 10000에 있다는 가정하에 다음의 이벤트 시퀀스 예를 살펴보세요.

1. 작업자가 샤드에서 다음 레코드 배치(레코드 10001부터 20000까지)를 읽습니다.
2. 그런 다음 작업자가 레코드 배치를 연결된 레코드 프로세서로 전달합니다.
3. 레코드 프로세서가 데이터를 집계하고 Amazon S3 파일을 생성하며 파일을 Amazon S3에 성공적으로 업로드합니다.
4. 새로운 체크포인트가 발생하기 전에 작업자가 예기치 않게 종료됩니다.
5. 애플리케이션, 작업자 및 레코드 프로세서가 다시 시작됩니다.
6. 이제 작업자가 마지막으로 성공한 체크포인트(여기서는 10001)에서 읽기 시작합니다.

따라서 레코드 10001 ~ 20000이 두 번 이상 소비됩니다.

### 소비자 재시도에 대한 복원

레코드가 두 번 이상 처리되더라도 한 번만 처리된 것처럼 보이는 부작용이 애플리케이션에 발생할 수 있습니다(idempotent 처리). 이 문제의 해결 방법은 복잡성과 정확도에 따라 다릅니다. 최종 데이터의 대상이 중복 항목을 잘 처리할 수 있으면 최종 대상을 통해 idempotent 처리를 수행하는 것이 좋습니다. 예를 들어, [Opensearch](#)를 통해 버전 관리와 고유 ID를 함께 사용하여 중복 처리를 방지할 수 있습니다.

이전 섹션의 예제 애플리케이션은 스트림에서 계속 레코드를 읽고 로컬 파일로 레코드를 집계하며 파일을 Amazon S3에 업로드합니다. 설명한 대로 레코드 10001~20000이 두 번 이상 소비되어 데이터가 동일한 Amazon S3 파일이 여러 개 생깁니다. 이 예에서 중복을 완화하는 한 가지 방법은 3단계에서 다음 방법을 사용하는 것입니다.

1. 레코드 프로세서가 Amazon S3 파일마다 일정한 수(예: 5000)의 레코드를 사용합니다.
2. 파일 이름에는 Amazon S3 접두사, 샤드 ID 및 First-Sequence-Num이라는 스키마가 사용됩니다. 이 경우에는 sample-shard000001-10001과 같은 이름일 수 있습니다.
3. Amazon S3 파일을 업로드한 후 Last-Sequence-Num을 지정하여 체크포인트를 수행합니다. 이 경우 레코드 번호 15000에서 검사합니다.

이 방법을 사용하면 레코드가 두 번 이상 처리되더라도 Amazon S3 파일의 이름과 데이터가 동일합니다. 재시도가 발생하면 반드시 동일한 데이터가 두 번 이상 같은 파일에 쓰여집니다.

리샤딩 작업에서는 샤드에 남은 레코드 수가 필요한 일정한 수보다 적을 수 있습니다. 이 경우 shutdown() 메서드가 Amazon S3 및 마지막 시퀀스 번호의 체크포인트로 파일을 플래시해야 합니다. 위의 방법은 리샤딩 작업에도 적용됩니다.

## 시작, 종료 및 조절 처리

다음은 Amazon Kinesis Data Streams 애플리케이션 설계에 통합할 몇 가지 추가 고려 사항입니다.

### 내용

- [데이터 생산자 및 데이터 소비자 시작](#)
- [Amazon Kinesis Data Streams 애플리케이션 종료](#)
- [읽기 조절](#)

### 데이터 생산자 및 데이터 소비자 시작

기본적으로 KCL은 스트림 끝(가장 최근에 추가한 레코드)에서 레코드를 읽기 시작합니다. 이 구성에서 수신 레코드 프로세서가 실행되기 전에 데이터 생성 애플리케이션이 레코드를 스트림에 추가하면 레코드 프로세서가 시작된 후 레코드 프로세서에서 레코드를 읽지 않습니다.

항상 스트림 시작부터 데이터를 읽도록 레코드 프로세서 동작을 변경하려면 Amazon Kinesis Data Streams 애플리케이션의 속성 파일에서 다음 값을 설정하세요.

```
initialPositionInStream = TRIM_HORIZON
```

기본적으로 Amazon Kinesis Data Streams는 모든 데이터를 24시간 동안 저장합니다. 또한 최대 7일의 연장 보존과 최대 365일의 장기 보존을 지원합니다. 이 기간을 보존 기간이라고 합니다. 시작 위치를 TRIM\_HORIZON으로 설정하면 보존 기간에 정의된 대로 스트림의 가장 오래된 데이터부터 레코드 프로세서가 시작됩니다. TRIM\_HORIZON 설정을 사용하더라도 보존 기간보다 오랜 시간이 지난 후에 레코드 프로세서가 시작되면 스트림의 일부 레코드를 더 이상 사용할 수 없게 됩니다. 따라서 항상 소비자 애플리케이션이 스트림에서 `GetRecords.IteratorAgeMilliseconds` 읽도록 하고 CloudWatch 메트릭을 사용하여 애플리케이션이 들어오는 데이터를 따라잡고 있는지 모니터링해야 합니다.

일부 시나리오에서는 레코드 프로세서가 스트림의 처음 레코드 몇 개를 놓쳐도 문제가 되지 않습니다. 예를 들어 스트림을 통해 일부 초기 레코드를 실행하여 스트림이 end-to-end 예상대로 작동하는지 테스트할 수 있습니다. 이 초기 확인을 수행한 후 작업자를 시작하고 스트림에 프로덕션 데이터를 입력하기 시작합니다.

TRIM\_HORIZON 설정에 관한 자세한 내용은 [샤드 반복자 사용](#)을 참조하십시오.

## Amazon Kinesis Data Streams 애플리케이션 종료

Amazon Kinesis Data Streams 애플리케이션이 의도한 작업을 완료하면 애플리케이션이 실행 중인 EC2 인스턴스를 종료하여 애플리케이션을 끝내야 합니다. [AWS Management Console](#) 또는 [AWS CLI](#)를 사용하여 인스턴스를 종료할 수 있습니다.

Amazon Kinesis Data Streams 애플리케이션을 종료한 후 KCL이 애플리케이션 상태를 추적하는데 사용한 Amazon DynamoDB 테이블을 삭제해야 합니다.

### 읽기 조절

스트림 처리량은 샤드 수준에서 프로비저닝됩니다. 각 샤드의 읽기 처리량은 초당 최대 5개의 트랜잭션이며, 최대 총 데이터 읽기 속도는 초당 2MB입니다. 애플리케이션(또는 동일한 스트림에서 작동하는 애플리케이션 그룹)이 더 빠른 속도로 샤드에서 데이터를 가져오려고 시도하면 Kinesis Data Streams가 해당 Get 작업을 조절합니다.

Amazon Kinesis Data Streams 애플리케이션에서 레코드 프로세서가 장애 조치의 경우와 같이 제한보다 빨리 데이터를 처리하면 제한이 발생합니다. KCL이 애플리케이션과 Kinesis Data Streams의 상호 작용을 관리하기 때문에 애플리케이션 코드보다는 KCL 코드에서 제한 예외가 발생합니다. 그러나 KCL이 이러한 예외를 기록하기 때문에 로그에서 예외를 볼 수 있습니다.

애플리케이션이 일관되게 조절되는 경우 스트림의 샤드 수를 늘려야 합니다.



# Amazon Kinesis Data Streams 모니터링

다음 기능을 사용하여 Amazon Kinesis Data Streams의 데이터 스트림을 모니터링할 수 있습니다.

- [CloudWatch 지표](#) — Kinesis Data Streams는 각 스트림에 대한 세부 모니터링이 포함된 CloudWatch Amazon 사용자 지정 지표를 전송합니다.
- [Kinesis Agent](#) — Kinesis 에이전트는 에이전트가 예상대로 작동하는지 평가하는 데 도움이 되는 사용자 지정 CloudWatch 메트릭을 게시합니다.
- [API 로깅](#) - Kinesis Data Streams는 AWS CloudTrail 을 사용하여 API 호출을 기록하고 Amazon S3 버킷에 데이터를 저장합니다.
- [Kinesis Client Library](#) – Kinesis Client Library(KCL)는 샤드, 워커 및 KCL 애플리케이션별 지표를 제공합니다.
- [Kinesis Producer Library](#) – Kinesis Producer Library(KPL)는 샤드, 워커 및 KPL 애플리케이션별 지표를 제공합니다.

일반적인 모니터링 문제, 질문 및 문제 해결에 대한 자세한 내용은 다음을 참조하세요.

- [Kinesis Data Streams 문제를 모니터링하고 해결하려면 어떤 지표를 사용해야 합니까?](#)
- [Kinesis Data Streams의 IteratorAgeMilliseconds 가치가 계속 증가하는 이유는 무엇입니까?](#)

## 아마존을 통한 아마존 Kinesis 데이터 스트림 서비스 모니터링 CloudWatch

Amazon Kinesis 데이터 스트림과 CloudWatch Amazon은 통합되어 있으므로 Kinesis 데이터 스트림에 대한 지표를 수집, 확인 및 CloudWatch 분석할 수 있습니다. 예를 들어 샤드 사용을 추적하기 위해 IncomingBytes 지표와 OutgoingBytes 지표를 모니터링하여 이 값을 스트림의 샤드 수와 비교할 수 있습니다.

스트림에 구성된 지표는 자동으로 수집되어 CloudWatch 1분마다 푸시됩니다. 측정치는 2주 간 보관되고 그 후에는 삭제됩니다.

다음 표에서는 Kinesis 데이터 스트림의 기본 스트림 수준 및 향상된 샤드 수준 모니터링을 설명합니다.

유형	설명
기본(스트림 수준)	자동으로 스트림 수준 데이터가 1분마다 무료로 전송됩니다.
향상(샤드 수준)	<p>추가 비용을 부담하면 샤드 수준 데이터가 1분마다 전송됩니다. 이 수준의 데이터를 가져오려면 <a href="#">EnableEnhancedMonitoring</a> 작업을 사용하여 스트림에 대해 특별히 활성화해야 합니다.</p> <p>요금에 대한 자세한 내용은 <a href="#">Amazon CloudWatch 제품 페이지</a>를 참조하십시오.</p>

## Amazon Kinesis Data Streams 측정기준 및 지표

Kinesis Data Streams는 두 가지 수준, 즉 스트림 수준과 샤드 수준 (선택 사항) 으로 지표를 CloudWatch 전송합니다. 스트림 수준 지표는 정상적인 조건에서 가장 일반적인 모니터링 사용 사례를 위한 것입니다. 샤드 수준 지표는 일반적으로 문제 해결과 관련된 특정 모니터링 작업용이며 작업을 통해 활성화됩니다. [EnableEnhancedMonitoring](#)

CloudWatch 지표에서 수집한 통계에 대한 설명은 Amazon CloudWatch 사용 설명서의 [CloudWatch 통계를 참조하십시오](#).

### 주제

- [기본적인 스트림 수준 지표](#)
- [향상된 샤드 수준 지표](#)
- [Amazon Kinesis Data Streams 지표의 측정기준](#)
- [권장되는 Amazon Kinesis Data Streams 지표](#)

### 기본적인 스트림 수준 지표

AWS/Kinesis 네임스페이스에는 다음과 같은 스트림 수준 지표가 포함되어 있습니다.

Kinesis Data Streams는 이러한 스트림 수준 CloudWatch 메트릭을 1분마다 전송합니다. 이 지표는 언제든지 사용할 수 있습니다.

지표	설명
GetRecords.Bytes	<p>지정한 기간 동안 측정된, Kinesis 스트림에서 가져온 바이트 수입입니다. Minimum, Maximum 및 Average 통계는 지정한 시간에 스트림에 사용된 단일 GetRecords 작업의 바이트 수를 의미합니다.</p> <p>샤드 수준 지표 이름: OutgoingBytes</p> <p>측정기준: StreamName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 바이트</p>
GetRecords.IteratorAge	<p>이 지표는 더 이상 사용되지 않습니다. GetRecords.IteratorAgeMilliseconds 를 사용합니다.</p>
GetRecords.IteratorAgeMilliseconds	<p>Kinesis 스트림에 대한 모든 GetRecords 호출에서 지정한 기간 동안 측정된 마지막 레코드의 경과 시간입니다. 여기에서 경과 시간이란 현재 시간과 마지막 GetRecords 호출 레코드가 스트림에 작성된 시간의 차이를 말합니다. Minimum 통계와 Maximum 통계는 Kinesis 소비자 애플리케이션의 진행 상황을 추적하는 데 사용할 수 있습니다. 값이 0이면 읽어오는 레코드가 스트림을 완전히 따라잡았다는 것을 의미합니다.</p> <p>샤드 수준 지표 이름: IteratorAgeMilliseconds</p> <p>치수: StreamName</p> <p>통계: Minimum, Maximum, Average, Samples</p> <p>단위: 밀리초</p>
GetRecords.Latency	<p>GetRecords 작업 1건당 지정한 시간 동안 측정된 소요 시간</p> <p>치수: StreamName</p>

지표	설명
	<p>Statistics: Minimum, Maximum, Average</p> <p>단위: 밀리초</p>
GetRecords.Records	<p>지정한 시간 동안 측정하며, 샤드에서 가져온 레코드 수. Minimum, Maximum 및 Average 통계는 지정한 시간에 스트림에 사용된 단일 GetRecords 작업의 레코드 수를 의미합니다.</p> <p>샤드 수준 지표 이름: OutgoingRecords</p> <p>치수: StreamName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>
GetRecords.Success	<p>지정한 시간 동안 측정하며, 스트림 1회마다 성공한 GetRecords 작업 수</p> <p>치수: StreamName</p> <p>통계: Average, Sum, Samples</p> <p>단위: 개</p>
IncomingBytes	<p>지정한 기간 동안 Kinesis 스트림에 성공적으로 입력된 바이트 수입니다. 이 지표에는 PutRecord 작업과 PutRecords 작업의 바이트 수도 포함됩니다. Minimum, Maximum 및 Average 통계는 지정한 시간에 스트림에 사용된 단일 입력 작업의 바이트 수를 의미합니다.</p> <p>샤드 수준 지표 이름: IncomingBytes</p> <p>치수: StreamName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 바이트</p>

지표	설명
IncomingRecords	<p>지정한 기간 동안 Kinesis 스트림에 성공적으로 입력된 레코드 수입니다. 이 지표에는 PutRecord 작업과 PutRecords 작업의 레코드 수도 포함됩니다. Minimum, Maximum 및 Average 통계는 지정한 시간에 스트림에 사용된 단일 입력 작업의 레코드 수를 의미합니다.</p> <p>샤드 수준 지표 이름: IncomingRecords</p> <p>치수: StreamName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>
PutRecord.Bytes	<p>지정한 기간 동안 PutRecord 작업을 사용하여 Kinesis 스트림에 입력된 바이트 수입니다.</p> <p>치수: StreamName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 바이트</p>
PutRecord.Latency	<p>PutRecord 작업 1건당 지정한 시간 동안 측정된 소요 시간</p> <p>치수: StreamName</p> <p>Statistics: Minimum, Maximum, Average</p> <p>단위: 밀리초</p>

지표	설명
PutRecord.Success	<p>지정한 시간 동안 측정된, Kinesis 스트림당 성공한 PutRecord 작업 수입입니다. Average는 스트림에 대한 성공적인 쓰기 작업 비율을 반영합니다.</p> <p>치수: StreamName</p> <p>통계: Average, Sum, Samples</p> <p>단위: 개</p>
PutRecords.Bytes	<p>지정한 기간 동안 PutRecords 작업을 사용하여 Kinesis 스트림에 입력된 바이트 수입입니다.</p> <p>치수: StreamName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 바이트</p>
PutRecords.Latency	<p>PutRecords 작업 1건당 지정한 시간 동안 측정된 소요 시간</p> <p>치수: StreamName</p> <p>Statistics: Minimum, Maximum, Average</p> <p>단위: 밀리초</p>
PutRecords.Records	<p>이 지표는 더 이상 사용되지 않습니다. PutRecords.SuccessfulRecords 를 사용합니다.</p> <p>치수: StreamName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>

지표	설명
PutRecords.Success	<p>지정한 기간 동안 측정된, Kinesis 스트림당 최소 1개 이상의 레코드가 성공한 PutRecords 작업 수입니다.</p> <p>치수: StreamName</p> <p>통계: Average, Sum, Samples</p> <p>단위: 개</p>
PutRecords.TotalRecords	<p>지정된 기간 동안 측정된, Kinesis 데이터 스트림당 PutRecords 작업에서 전송된 총 레코드 수입니다.</p> <p>치수: StreamName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>
PutRecords.SuccessfulRecords	<p>지정한 기간 동안 측정된, Kinesis 데이터 스트림당 PutRecords 작업에서 성공한 레코드 수입니다.</p> <p>치수: StreamName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>
PutRecords.FailedRecords	<p>지정된 기간 동안 측정된, Kinesis 데이터 스트림당 PutRecords 작업에서 내부 오류로 인해 거부된 레코드 수입니다. 간헐적인 내부 오류가 예상되므로 재시도가 필요합니다.</p> <p>치수: StreamName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>

지표	설명
PutRecords.ThrottledRecords	<p>지정된 기간 동안 측정된, Kinesis 데이터 스트림당 PutRecords 작업에서 제한으로 인해 거부된 레코드 수입니다.</p> <p>치수: StreamName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>
ReadProvisionedThroughputExceeded	<p>지정한 시간 동안 측정하며, 스트림 병목 현상을 초래한 GetRecords 호출 수. 이 지표에서 가장 흔하게 사용되는 통계는 Average입니다.</p> <p>Minimum 통계 값이 1일 때는 지정한 시간 동안 모든 레코드가 스트림 병목 현상을 초래한 것을 의미합니다.</p> <p>Maximum 통계 값이 0(영)일 때는 지정한 시간 동안 스트림 병목 현상을 초래한 레코드가 없다는 것을 의미합니다.</p> <p>샤드 수준 지표 이름: ReadProvisionedThroughputExceeded</p> <p>치수: StreamName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>
SubscribeToShard.RateExceeded	<p>이 측정치는 동일한 소비자에 의한 활성 구독이 이미 있기 때문에 새로운 구독 시도가 실패할 경우 또는 이 작업에 허용되는 초당 호출 수를 초과할 경우 생성됩니다.</p> <p>치수: StreamName, ConsumerName</p>



지표	설명
SubscribeToShard.Success	<p>이 지표는 SubscribeToShard 구독이 성공적으로 설정되었는지 여부를 기록합니다. 구독은 최대 5분간만 지속됩니다. 따라서 이 측정치는 적어도 5분마다 한 번 생성됩니다.</p> <p>치수: StreamName, ConsumerName</p>
SubscribeToShardEvent.Bytes	<p>지정한 시간 동안 측정하며, 샤드로부터 수신한 바이트 수. Minimum, Maximum 및 Average 통계는 지정한 시간 동안 단일 이벤트에 게시된 바이트 수를 의미합니다.</p> <p>샤드 수준 지표 이름: OutgoingBytes</p> <p>치수: StreamName, ConsumerName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 바이트</p>
SubscribeToShardEvent.MillisBehindLatest	<p>현재 시간과 SubscribeToShard 이벤트의 마지막 레코드가 스트림에 기록된 시점의 차이.</p> <p>크기: StreamName, ConsumerName</p> <p>통계: Minimum, Maximum, Average, Samples</p> <p>단위: 밀리초</p>
SubscribeToShardEvent.Records	<p>지정한 시간 동안 측정하며, 샤드로부터 수신한 레코드 수. Minimum, Maximum 및 Average 통계는 지정한 시간 동안 단일 이벤트의 레코드 수를 의미합니다.</p> <p>샤드 수준 지표 이름: OutgoingRecords</p> <p>치수: StreamName, ConsumerName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>

지표	설명
SubscribeToShardEvent.Success	<p>이 측정치는 이벤트가 성공적으로 게시될 때마다 생성됩니다. 활성 구독이 있는 경우에만 생성됩니다.</p> <p>치수: StreamName, ConsumerName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>
WriteProvisionedThroughputExceeded	<p>지정한 시간 동안 스트림 병목 현상으로 인해 거부된 레코드 수. 이 지표에는 PutRecord 작업과 PutRecords 작업에서 발생하는 병목 현상도 포함됩니다. 이 지표에서 가장 흔하게 사용되는 통계는 Average입니다.</p> <p>Minimum 통계가 0이 아닌 값일 때는 지정한 시간 동안 레코드에 스트림 병목 현상이 발생하였다는 것을 의미합니다.</p> <p>Maximum 통계 값이 0(영)일 때는 지정한 시간 동안 스트림 병목 현상이 발생한 레코드가 없다는 것을 의미합니다.</p> <p>샤드 수준 지표 이름: WriteProvisionedThroughputExceeded</p> <p>치수: StreamName</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>

## 향상된 샤드 수준 지표

AWS/Kinesis 네임스페이스에는 다음과 같은 샤드 수준 지표가 포함되어 있습니다.

Kinesis는 1분마다 다음과 같은 샤드 레벨 메트릭을 전송합니다. CloudWatch 각 지표 차원은 CloudWatch 지표 1개를 생성하고 매월 약 43,200건의 PutMetricData API 호출을 수행합니다. 이 지표는 기본적으로 활성화되어 있지 않습니다. Kinesis에서 내보낸, 향상된 지표에는 요금이 부과됩니다.

다. 자세한 내용은 [Amazon CloudWatch 커스텀 메트릭](#)이라는 제목 아래의 Amazon CloudWatch 요금을 참조하십시오. 요금은 매월 지표당 공유마다 부과됩니다.

지표	설명
IncomingBytes	<p>지정한 시간 동안 샤드에 성공적으로 입력된 바이트 수. 이 지표에는 PutRecord 작업과 PutRecords 작업의 바이트 수도 포함됩니다. Minimum, Maximum 및 Average 통계는 지정한 시간에 샤드에 사용된 단일 입력 작업의 바이트 수를 의미합니다.</p> <p>스트림 수준 지표 이름: IncomingBytes</p> <p>치수: StreamName, ShardId</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 바이트</p>
IncomingRecords	<p>지정한 시간 동안 샤드에 성공적으로 입력된 레코드 수. 이 지표에는 PutRecord 작업과 PutRecords 작업의 레코드 수도 포함됩니다. Minimum, Maximum 및 Average 통계는 지정한 시간에 샤드에 사용된 단일 입력 작업의 레코드 수를 의미합니다.</p> <p>스트림 수준 지표 이름: IncomingRecords</p> <p>치수: StreamName, ShardId</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>
IteratorAgeMilliseconds	<p>샤드에 대한 모든 GetRecords 호출에서 지정한 기간 동안 측정된 마지막 레코드의 경과 시간. 여기에서 경과 시간이란 현재 시간과 마지막 GetRecords 호출 레코드가 스트림에 작성된 시간의 차이를 말합니다. Minimum 통계와 Maximum 통계는 Kinesis 소비자 애플리케이션의 진행 상황을 추적하는 데 사용할 수 있습니다. 값이 0(영)이면 읽</p>

지표	설명
	<p>어오는 레코드가 스트림을 완전히 따라잡았다는 것을 의미합니다.</p> <p>스트림 수준 지표 이름: <code>GetRecords.IteratorAgeMilliseconds</code></p> <p>치수: <code>StreamName, ShardId</code></p> <p>통계: <code>Minimum, Maximum, Average, Samples</code></p> <p>단위: 밀리초</p>
<code>OutgoingBytes</code>	<p>지정한 시간 동안 측정하며, 샤드에서 가져온 바이트 수. <code>Minimum, Maximum</code> 및 <code>Average</code> 통계는 지정한 시간 동안 단일 <code>GetRecords</code> 작업에서 반환된 또는 샤드의 단일 <code>SubscribeToShard</code> 이벤트에서 게시된 바이트 수를 의미합니다.</p> <p>스트림 수준 지표 이름: <code>GetRecords.Bytes</code></p> <p>치수: <code>StreamName, ShardId</code></p> <p>통계: <code>Minimum, Maximum, Average, Sum, Samples</code></p> <p>단위: 바이트</p>
<code>OutgoingRecords</code>	<p>지정한 시간 동안 측정하며, 샤드에서 가져온 레코드 수. <code>Minimum, Maximum</code> 및 <code>Average</code> 통계는 지정한 시간 동안 단일 <code>GetRecords</code> 작업에서 반환된 또는 샤드의 단일 <code>SubscribeToShard</code> 이벤트에서 게시된 레코드 수를 의미합니다.</p> <p>스트림 수준 지표 이름: <code>GetRecords.Records</code></p> <p>치수: <code>StreamName, ShardId</code></p> <p>통계: <code>Minimum, Maximum, Average, Sum, Samples</code></p> <p>단위: 개</p>

지표	설명
ReadProvisionedThroughputExceeded	<p>지정한 시간 동안 측정하며, 샤드 병목 현상을 초래한 GetRecords 호출 수. 이 예외 수는 1초마다 샤드 1개당 읽기 수가 5개이거나, 샤드 1개마다 초당 읽기 크기가 2MB로 제한되는 모든 차원에 적용됩니다. 이 지표에서 가장 흔하게 사용되는 통계는 Average입니다.</p> <p>Minimum 통계 값이 1일 때는 지정한 시간 동안 모든 레코드가 샤드 병목 현상을 초래한 것을 의미합니다.</p> <p>Maximum 통계 값이 0(영)일 때는 지정한 시간 동안 샤드 병목 현상을 초래한 레코드가 없다는 것을 의미합니다.</p> <p>스트림 수준 지표 이름: ReadProvisionedThroughputExceeded</p> <p>치수: StreamName, ShardId</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>

지표	설명
WriteProvisionedThroughputExceeded	<p>지정한 시간 동안 샤드 병목 현상으로 인해 거부된 레코드 수. 이 지표는 PutRecord 작업과 PutRecords 작업의 병목 현상도 포함되며, 샤드 1개마다 초당 레코드 수가 1,000개이거나, 혹은 샤드 1개마다 초당 레코드 크기가 1MB로 제한되는 모든 차원에 적용됩니다. 이 지표에서 가장 흔하게 사용되는 통계는 Average입니다.</p> <p>Minimum 통계가 0이 아닌 값일 때는 지정한 시간 동안 레코드에 샤드 병목 현상이 발생하였다는 것을 의미합니다.</p> <p>Maximum 통계 값이 0(영)일 때는 지정한 시간 동안 샤드 병목 현상이 발생한 레코드가 없다는 것을 의미합니다.</p> <p>스트림 수준 지표 이름: WriteProvisionedThroughputExceeded</p> <p>치수: StreamName, ShardId</p> <p>통계: Minimum, Maximum, Average, Sum, Samples</p> <p>단위: 개</p>

## Amazon Kinesis Data Streams 지표의 측정기준

측정기준	설명
StreamName	Kinesis 비디오 스트림의 이름입니다. 사용 가능한 모든 통계는 StreamName 로 필터링됩니다.

## 권장되는 Amazon Kinesis Data Streams 지표

Kinesis Data Streams 고객은 몇 가지 Amazon Kinesis Data Streams 지표에 특히 관심을 가질 수 있습니다. 다음은 권장되는 측정치와 그 용도 목록입니다.

지표	사용 관련 참고 사항
GetRecords.IteratorAgeMilliseconds	스트림의 모든 샤드와 소비자에서 읽기 위치를 추적합니다. 반복자 수명이 보존 기간(기본적으로 24시간, 최대 7일까지 구성 가능)의 50%를 경과하면 레코드 만료로 인해 데이터가 손실될 위험이 있습니다. 이러한 손실이 위험하기 전에 최대 통계에 대한 CloudWatch 경보를 사용하여 경고하는 것이 좋습니다. 이 측정치가 사용되는 시나리오의 예는 <a href="#">소비자 레코드 처리 속도가 느려지는 경우</a> 를 참조하십시오.
ReadProvisionedThroughputExceeded	소비자 측 레코드 처리 속도가 느려질 때 병목 현상이 일어난 위치를 알기 어려울 때가 있습니다. 이 측정치를 사용하여 읽기 처리량 제한 초과로 인해 읽기가 제한되고 있는지 확인하십시오. 이 지표에서 가장 흔하게 사용되는 통계는 Average입니다.
WriteProvisionedThroughputExceeded	ReadProvisionedThroughputExceeded 측정치와 같은 용도지만 스트림의 생산자(널기) 측에 사용됩니다. 이 지표에서 가장 흔하게 사용되는 통계는 Average입니다.
PutRecords.Success, PutRecords.Success	레코드가 스트림에 실패하는 시점을 CloudWatch 알려려면 평균 통계에서 경보를 사용하는 것이 좋습니다. 생산자가 무엇을 사용하는지에 따라 put 유형을 한 개 또는 둘 다 선택합니다. Kinesis Producer Library(KPL)를 사용하는 경우 PutRecords.Success 를 사용합니다.
GetRecords.Success	스트림에서 레코드가 실패하는 경우를 나타내려면 평균 통계에서 CloudWatch 경보를 사용하는 것이 좋습니다.

## Kinesis Data Streams용 Amazon CloudWatch 메트릭스에 액세스

콘솔, 명령줄 또는 CloudWatch API를 사용하여 CloudWatch Kinesis Data Streams의 메트릭을 모니터링할 수 있습니다. 다음의 절차는 이처럼 다양한 방법을 사용하여 측정치에 액세스하는 방법을 설명합니다.

콘솔을 사용하여 메트릭에 CloudWatch 액세스하려면

1. <https://console.aws.amazon.com/cloudwatch/> 에서 CloudWatch 콘솔을 엽니다.
2. 탐색 모음에서 리전을 선택합니다.

3. 탐색 창에서 지표(Metrics)를 선택합니다.
4. 범주별 CloudWatch 지표 창에서 Kinesis 지표를 선택합니다.
5. 지정된 MetricName 및 StreamName에 대한 통계를 보려면 관련 행을 클릭합니다.

참고: 읽기 처리량 및 쓰기 처리량을 제외하고 대부분의 콘솔 통계 이름은 위에 나열된 해당 CloudWatch 지표 이름과 일치합니다. 이러한 통계는 5분 간격으로 계산됩니다. 쓰기 처리량은 IncomingBytes CloudWatch 지표를 모니터링하고 읽기 처리량은 모니터링합니다. GetRecords.Bytes.

6. (선택 사항) 그래프 창에서 통계와 기간을 선택한 다음 이러한 설정을 사용하여 CloudWatch 경보를 생성합니다.

를 사용하여 지표에 액세스하려면 AWS CLI

[목록 지표](#) 및 [get-metric-statistics](#) 명령을 사용하십시오.

CloudWatch CLI를 사용하여 지표에 액세스하려면

[mon-list-metrics](#) 및 [mon-get-stats](#) 명령을 사용합니다.

CloudWatch API를 사용하여 지표에 액세스하려면

[ListMetrics](#) 및 [GetMetricStatistics](#) 작업을 사용합니다.

## 아마존을 통한 Kinesis Data Streams 에이전트 헬스 모니터링 CloudWatch

에이전트는 네임스페이스가 인 사용자 지정 CloudWatch 지표를 게시합니다. AWS KinesisAgent 이러한 지표는 에이전트가 지정된 대로 Kinesis Data Streams에 데이터를 제출하고 있는지, 에이전트가 정상이며 데이터 생산자에서 적절한 양의 CPU와 메모리 리소스를 소비하고 있는지 평가하는 데 도움이 됩니다. 전송된 레코드 수와 바이트 등의 측정치는 에이전트가 스트림에 데이터를 제출하는 속도를 이해하는 데 유용합니다. 이러한 측정치가 예상 임계값에 다소 못 미치거나 0으로 떨어지는 경우, 구성 문제, 네트워크 오류 또는 에이전트 상태 문제를 나타낼 수 있습니다. 호스트상의 CPU 및 메모리 소비 등의 측정치와 에이전트 오류 카운터는 데이터 생산자의 리소스 사용량을 나타내며, 잠재적인 구성 또는 호스트 오류에 대한 통찰을 제공합니다. 마지막으로 에이전트는 서비스 예외도 로깅하여 에이전트 문제를 조사하는 데 도움을 줍니다. 이러한 측정치는 에이전트 구성 설정 `cloudwatch.endpoint`에 지정된 리전에서 보고됩니다. 여러 Kinesis 에이전트에서 게시된 Cloudwatch 지표는 집계되거나 결합됩니다. 에이전트 구성에 대한 자세한 내용은 [에이전트 구성 설정](#) 단원을 참조하세요.



## 를 통한 모니터링 CloudWatch

Kinesis Data Streams 에이전트는 에 다음 CloudWatch 메트릭을 전송합니다.

지표	설명
BytesSent	지정된 기간 동안 Kinesis Data Streams에 전송된 바이트 수입니다.  단위: 바이트
RecordSendAttempts	지정된 기간 동안 PutRecords 에 대한 호출에서 시도한 레코드 수입니다(처음 또는 다시 시도).  단위: 개
RecordSendErrors	지정한 기간 동안 재시도를 포함해 PutRecords 에 대한 호출에서 실패 상태를 반환한 레코드 수입니다.  단위: 개
ServiceErrors	지정된 기간 동안 서비스 오류(조절 오류 제외)를 일으킨 PutRecords 에 대한 호출 수입니다.  단위: 개

## AWS CloudTrail을 사용하여 Amazon Kinesis Data Streams API 호출 로깅

Amazon Kinesis Data Streams는 Kinesis Data Streams에서 사용자, 역할 또는 AWS 서비스가 수행한 작업의 기록을 제공하는 서비스와 AWS CloudTrail통합됩니다. CloudTrail Kinesis Data Streams에 대한 모든 API 호출을 이벤트로 캡처합니다. 캡처되는 호출에는 Kinesis Data Streams 콘솔로부터의 호출과 Kinesis Data Streams API 작업에 대한 코드 호출이 포함됩니다. 트레일을 생성하면 Kinesis Data Streams에 대한 CloudTrail 이벤트를 포함하여 Amazon S3 버킷에 이벤트를 지속적으로 전송할 수 있습니다. 트레일을 구성하지 않아도 CloudTrail 콘솔의 이벤트 기록에서 최신 이벤트를 계속 볼 수 있습니다. 에서 수집한 CloudTrail 정보를 사용하여 Kinesis Data Streams에 대한 요청, 요청이 이루어진 IP 주소, 요청한 사람, 요청 시기 및 추가 세부 정보를 확인할 수 있습니다.

구성 및 활성화 방법을 CloudTrail 포함하여 자세한 내용은 [사용 AWS CloudTrail 설명서를 참조하십시오](#).

## Kinesis Data Streams 정보 입력 CloudTrail

CloudTrail 계정을 생성할 때 AWS 계정에서 활성화됩니다. 지원되는 이벤트 활동이 Kinesis Data Streams에서 발생하면 해당 활동이 이벤트 기록의 CloudTrail AWS 다른 서비스 이벤트와 함께 이벤트에 기록됩니다. AWS 계정에서 최근 이벤트를 보고, 검색하고, 다운로드할 수 있습니다. 자세한 내용은 이벤트 [기록으로 CloudTrail 이벤트를 보기를](#) 참조하십시오.

Kinesis Data Streams의 이벤트를 포함하여 AWS 계정의 진행 중인 이벤트 기록을 보려면 트레일을 생성하십시오. 트레일을 사용하면 CloudTrail Amazon S3 버킷으로 로그 파일을 전송할 수 있습니다. 기본적으로 콘솔에서 트레일을 생성하면 트레일이 모든 AWS 지역에 적용됩니다. 트레일은 AWS 파티션에 있는 모든 지역의 이벤트를 기록하고 지정한 Amazon S3 버킷으로 로그 파일을 전송합니다. 또한 CloudTrail 로그에서 수집된 이벤트 데이터를 추가로 분석하고 이에 따라 조치를 취하도록 다른 AWS 서비스를 구성할 수 있습니다. 자세한 내용은 다음을 참조하십시오.

- [추적 생성 개요](#)
- [CloudTrail 지원되는 서비스 및 통합](#)
- [예 대한 Amazon SNS 알림 구성 CloudTrail](#)
- [여러 지역에서 CloudTrail 로그 파일 수신 및 여러 계정으로부터 CloudTrail 로그 파일 수신](#)

Kinesis Data Streams는 다음과 같은 작업을 로그 파일에 이벤트로 기록할 CloudTrail 수 있도록 지원합니다.

- [AddTagsToStream](#)
- [CreateStream](#)
- [DecreaseStreamRetentionPeriod](#)
- [DeleteStream](#)
- [DeregisterStreamConsumer](#)
- [DescribeStream](#)
- [DescribeStreamConsumer](#)
- [DisableEnhancedMonitoring](#)
- [EnableEnhancedMonitoring](#)
- [GetRecords](#)

- [GetShardIterator](#)
- [IncreaseStreamRetentionPeriod](#)
- [ListStreamConsumers](#)
- [ListStreams](#)
- [ListTagsForStream](#)
- [MergeShards](#)
- [PutRecord](#)
- [PutRecords](#)
- [RegisterStreamConsumer](#)
- [RemoveTagsFromStream](#)
- [SplitShard](#)
- [StartStreamEncryption](#)
- [StopStreamEncryption](#)
- [SubscribeToShard](#)
- [UpdateShardCount](#)
- [UpdateStreamMode](#)

모든 이벤트 및 로그 항목에는 요청을 생성한 사용자에게 대한 정보가 들어 있습니다. 신원 정보를 이용하면 다음을 쉽게 알아볼 수 있습니다.

- 요청이 루트 또는 AWS Identity and Access Management (IAM) 사용자 자격 증명으로 이루어졌는지 여부.
- 역할 또는 연동 사용자를 위한 임시 보안 인증으로 요청을 생성하였는지.
- 다른 AWS 서비스에서 요청했는지 여부.

자세한 내용은 [CloudTrail사용자 ID 요소를 참조하십시오](#).

## 예: Kinesis Data Streams 로그 파일 항목

트레일은 지정한 Amazon S3 버킷에 이벤트를 로그 파일로 전송할 수 있는 구성입니다. CloudTrail 로그 파일에는 하나 이상의 로그 항목이 포함되어 있습니다. 이벤트는 모든 소스로부터의 단일 요청을 나타내며 요청 작업, 작업 날짜와 시간, 요청 파라미터 등에 대한 정보가 들어 있습니다. CloudTrail 로그 파일은 공개 API 호출의 정렬된 스택 추적이 아니므로 특정 순서로 표시되지 않습니다.

다음 예제는 CreateStream, DescribeStream ListStreams DeleteStream SplitShard, 및 MergeShards 작업을 보여주는 CloudTrail 로그 항목을 보여줍니다.

```
{
  "Records": [
    {
      "eventVersion": "1.01",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "EX_PRINCIPAL_ID",
        "arn": "arn:aws:iam::012345678910:user/Alice",
        "accountId": "012345678910",
        "accessKeyId": "EXAMPLE_KEY_ID",
        "userName": "Alice"
      },
      "eventTime": "2014-04-19T00:16:31Z",
      "eventSource": "kinesis.amazonaws.com",
      "eventName": "CreateStream",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
      "requestParameters": {
        "shardCount": 1,
        "streamName": "GoodStream"
      },
      "responseElements": null,
      "requestID": "db6c59f8-c757-11e3-bc3b-57923b443c1c",
      "eventID": "b7acfc0d-6ca9-4ee1-a3d7-c4e8d420d99b"
    },
    {
      "eventVersion": "1.01",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "EX_PRINCIPAL_ID",
        "arn": "arn:aws:iam::012345678910:user/Alice",
        "accountId": "012345678910",
        "accessKeyId": "EXAMPLE_KEY_ID",
        "userName": "Alice"
      },
      "eventTime": "2014-04-19T00:17:06Z",
      "eventSource": "kinesis.amazonaws.com",
      "eventName": "DescribeStream",
      "awsRegion": "us-east-1",
```

```
"sourceIPAddress": "127.0.0.1",
"userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
"requestParameters": {
  "streamName": "GoodStream"
},
"responseElements": null,
"requestID": "f0944d86-c757-11e3-b4ae-25654b1d3136",
"eventID": "0b2f1396-88af-4561-b16f-398f8eaea596"
},
{
  "eventVersion": "1.01",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "EX_PRINCIPAL_ID",
    "arn": "arn:aws:iam::012345678910:user/Alice",
    "accountId": "012345678910",
    "accessKeyId": "EXAMPLE_KEY_ID",
    "userName": "Alice"
  },
  "eventTime": "2014-04-19T00:15:02Z",
  "eventSource": "kinesis.amazonaws.com",
  "eventName": "ListStreams",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "127.0.0.1",
  "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
  "requestParameters": {
    "limit": 10
  },
  "responseElements": null,
  "requestID": "a68541ca-c757-11e3-901b-cbcfe5b3677a",
  "eventID": "22a5fb8f-4e61-4bee-a8ad-3b72046b4c4d"
},
{
  "eventVersion": "1.01",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "EX_PRINCIPAL_ID",
    "arn": "arn:aws:iam::012345678910:user/Alice",
    "accountId": "012345678910",
    "accessKeyId": "EXAMPLE_KEY_ID",
    "userName": "Alice"
  },
  "eventTime": "2014-04-19T00:17:07Z",
  "eventSource": "kinesis.amazonaws.com",
```

```

    "eventName": "DeleteStream",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "127.0.0.1",
    "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
    "requestParameters": {
      "streamName": "GoodStream"
    },
    "responseElements": null,
    "requestID": "f10cd97c-c757-11e3-901b-cbcfe5b3677a",
    "eventID": "607e7217-311a-4a08-a904-ec02944596dd"
  },
  {
    "eventVersion": "1.01",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "EX_PRINCIPAL_ID",
      "arn": "arn:aws:iam::012345678910:user/Alice",
      "accountId": "012345678910",
      "accessKeyId": "EXAMPLE_KEY_ID",
      "userName": "Alice"
    },
    "eventTime": "2014-04-19T00:15:03Z",
    "eventSource": "kinesis.amazonaws.com",
    "eventName": "SplitShard",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "127.0.0.1",
    "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
    "requestParameters": {
      "shardToSplit": "shardId-000000000000",
      "streamName": "GoodStream",
      "newStartingHashKey": "11111111"
    },
    "responseElements": null,
    "requestID": "a6e6e9cd-c757-11e3-901b-cbcfe5b3677a",
    "eventID": "dcd2126f-c8d2-4186-b32a-192dd48d7e33"
  },
  {
    "eventVersion": "1.01",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "EX_PRINCIPAL_ID",
      "arn": "arn:aws:iam::012345678910:user/Alice",
      "accountId": "012345678910",
      "accessKeyId": "EXAMPLE_KEY_ID",

```

```

        "userName": "Alice"
    },
    "eventTime": "2014-04-19T00:16:56Z",
    "eventSource": "kinesis.amazonaws.com",
    "eventName": "MergeShards",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "127.0.0.1",
    "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
    "requestParameters": {
        "streamName": "GoodStream",
        "adjacentShardToMerge": "shardId-000000000002",
        "shardToMerge": "shardId-000000000001"
    },
    "responseElements": null,
    "requestID": "e9f9c8eb-c757-11e3-bf1d-6948db3cd570",
    "eventID": "77cf0d06-ce90-42da-9576-71986fec411f"
}
]
}

```

## Amazon을 통한 Kinesis 클라이언트 라이브러리 모니터링 CloudWatch

Amazon [Kinesis 데이터 스트림용 Kinesis 클라이언트 라이브러리](#) (KCL) 는 KCL 애플리케이션의 이름을 네임스페이스로 사용하여 사용자를 대신하여 사용자 지정 CloudWatch Amazon 지표를 게시합니다. [콘솔로 이동한 다음 사용자 지정 지표를 선택하면 이러한 지표를 볼 수 있습니다.](#) [CloudWatch](#) 사용자 지정 지표에 대한 자세한 내용은 Amazon CloudWatch 사용 설명서의 [사용자 지정 지표 게시](#)를 참조하십시오.

KCL에서 업로드한 지표에는 소액의 요금이 CloudWatch 부과되며, 특히 Amazon CloudWatch 커스텀 메트릭과 Amazon CloudWatch API 요청 요금이 적용됩니다. 자세한 내용은 [Amazon CloudWatch 요금](#)을 참조하십시오.

### 주제

- [측정치 및 네임스페이스](#)
- [측정치 수준 및 차원](#)
- [측정치 구성](#)
- [측정치 목록](#)

## 측정치 및 네임스페이스

지표를 업로드하는 데 사용되는 네임스페이스는 KCL을 시작할 때 지정된 애플리케이션 이름입니다.

## 측정치 수준 및 차원

업로드할 지표를 제어하는 두 가지 옵션이 있습니다 CloudWatch.

### 측정치 수준

모든 측정치에는 개별 수준이 할당됩니다. 지표 보고 수준을 설정하면 개별 수준이 보고 수준보다 낮은 지표는 전송되지 않습니다 CloudWatch. 수준은 NONE, SUMMARY 및 DETAILED입니다. 기본 설정은 다음과 같습니다 DETAILED. 즉, 모든 지표가 전송되는 CloudWatch 위치입니다. 보고 수준이 NONE이면 측정치가 전혀 전송되지 않습니다. 어떤 지표에 어떤 수준이 할당되는지에 대한 자세한 내용은 [측정치 목록](#) 단원을 참조하십시오.

### 활성화된 차원

모든 KCL 지표에는 전송되는 관련 측정기준이 있습니다. CloudWatch KCL 2.x에서 KCL이 단일 데이터 스트림을 처리하도록 구성된 경우 모든 지표 측정기준(Operation, ShardId 및 WorkerIdentifier)이 기본적으로 활성화됩니다. 또한 KCL 2.x에서 KCL이 단일 데이터 스트림을 처리하도록 구성된 경우 Operation 측정기준을 비활성화할 수 없습니다. KCL 2.x에서 KCL이 여러 데이터 스트림을 처리하도록 구성된 경우 모든 지표 차원(Operation, ShardId, StreamId 및 WorkerIdentifier)이 기본적으로 활성화됩니다. 또한 KCL 2.x에서 KCL이 여러 데이터 스트림을 처리하도록 구성된 경우 Operation 및 StreamId 측정기준을 비활성화할 수 없습니다. StreamId 측정기준은 샤드별 지표에만 사용할 수 있습니다.

KCL 1.x에서 기본적으로 Operation 및 ShardId 측정기준만 활성화되어 있으며 WorkerIdentifier 측정기준은 비활성화되어 있습니다. KCL 1.x에서는 Operation 측정기준을 비활성화할 수 없습니다.

CloudWatch 지표 차원에 대한 자세한 내용은 Amazon CloudWatch 사용 설명서의 Amazon CloudWatch Concepts 주제에 있는 [치수](#) 섹션을 참조하십시오.

WorkerIdentifier 차원이 활성화된 경우 특정 KCL 작업자가 다시 시작될 때마다 작업자 ID 속성에 다른 값을 사용하면 새 WorkerIdentifier 측정기준 값이 포함된 새 지표 세트가 로 전송됩니다. CloudWatch 특정 KCL 워커가 다시 시작될 때 WorkerIdentifier 측정기준 값이 동일해야 하는 경우 각 워커의 초기화 중에 동일한 워커 ID 값을 명시적으로 지정해야 합니다. 각 활성 KCL 워커의 워커 ID 값은 모든 KCL 워커에서 고유해야 합니다.



## 측정치 구성

인스턴스를 사용하여 지표 수준 및 활성화된 차원을 구성할 수 있으며, 이 KinesisClientLibConfiguration 인스턴스는 KCL 애플리케이션을 시작할 때 Worker에 전달됩니다. 이 MultiLangDaemon 경우 MultiLangDaemon KCL 애플리케이션을 시작하는 데 사용된.properties 파일에 metricsLevel 및 metricsEnabledDimensions 속성을 지정할 수 있습니다.

세 가지 값(NONE, SUMMARY 또는 DETAILED) 중 하나를 측정치 수준에 할당할 수 있습니다. 활성화된 측정기준 값은 지표에 허용되는 차원 목록이 포함된 심포로 구분된 문자열이어야 합니다. CloudWatch KCL 애플리케이션에서 사용하는 측정기준은 Operation, ShardId 및 WorkerIdentifier입니다.

## 측정치 목록

다음 표에는 범위와 작업을 기준으로 그룹화된 KCL 지표가 나와 있습니다.

### 주제

- [KCL 애플리케이션별 지표](#)
- [작업자별 측정치](#)
- [샤드별 측정치](#)

## KCL 애플리케이션별 지표

이러한 지표는 Amazon CloudWatch 네임스페이스에서 정의한 대로 애플리케이션 범위 내의 모든 KCL 작업자에 대해 집계됩니다.

### 주제

- [InitializeTask](#)
- [ShutdownTask](#)
- [ShardSyncTask](#)
- [BlockOnParentTask](#)
- [PeriodicShardSyncManager](#)
- [MultistreamTracker](#)

## InitializeTask

InitializeTask 작업은 KCL 애플리케이션의 레코드 프로세서를 초기화합니다. 이 작업의 로직에는 Kinesis Data Streams에서 샤드 반복자 가져오기 및 레코드 프로세서 초기화가 포함됩니다.

지표	설명
KinesisDataFetcher .getIterator.Success	KCL 애플리케이션별로 성공한 GetShardIterator 작업 수입니다.  측정치 수준: Detailed  단위: 개
KinesisDataFetcher. 이터레이터 시간 가져 오기	지정된 KCL 애플리케이션의 GetShardIterator 작업당 소요된 시간 입니다.  측정치 수준: Detailed  단위: 밀리초
RecordProcessor. 시 간 초기화	레코드 프로세서의 초기화 메서드에 소요된 시간입니다.  측정치 수준: Summary  단위: 밀리초
Success	성공한 레코드 프로세서 초기화 수입니다.  측정치 수준: Summary  단위: 개
Time	KCL 워커가 레코드 프로세서를 초기화하는 데 소요된 시간입니다.  측정치 수준: Summary  단위: 밀리초

## ShutdownTask

ShutdownTask 작업은 샤드 처리를 위해 종료 시퀀스를 시작합니다. 샤드가 분할되거나 병합되어 또는 작업자에서 샤드 리스가 손실된 경우 발생할 수 있습니다. 두 경우 모두 레코드 프로세서 shutdown() 함수가 호출됩니다. 샤드가 분할 또는 합병된 경우 새 샤드가 발견되어 새 샤드가 1개나 2개 생성됩니다.

지표	설명
CreateLease. 성공	<p>상위 샤드 종료 후 새로운 하위 샤드가 KCL 애플리케이션 DynamoDB 테이블에 성공적으로 추가된 횟수입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 개</p>
CreateLease. 시간	<p>KCL 애플리케이션 DynamoDB 테이블에 새 하위 샤드 정보를 추가하는데 소요된 시간입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 밀리초</p>
UpdateLease. 성공	<p>레코드 프로세서 종료 중 성공한 최종 체크포인트 수입입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 개</p>
UpdateLease. 시간	<p>레코드 프로세서 종료 중 체크포인트 작업에 소요된 시간입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 밀리초</p>
RecordProcessor. 종료. 시간	<p>레코드 프로세서의 종료 메서드에 소요된 시간입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 밀리초</p>
Success	성공한 종료 작업 수입입니다.

지표	설명
	측정치 수준: Summary 단위: 개
Time	KCL 워커가 종료 작업에 소요한 시간입니다. 측정치 수준: Summary 단위: 밀리초

## ShardSyncTask

ShardSyncTask 작업은 KCL 애플리케이션에서 새로운 샤드를 처리할 수 있도록 Kinesis 데이터 스트림의 샤드 정보 변경 사항을 검색합니다.

지표	설명
CreateLease. 성공	KCL 애플리케이션 DynamoDB 테이블에 새 샤드 정보를 추가하려는 시도가 성공한 횟수입니다. 측정치 수준: Detailed 단위: 개
CreateLease. 시간	KCL 애플리케이션 DynamoDB 테이블에 새 샤드 정보를 추가하는 데 소요된 시간입니다. 측정치 수준: Detailed 단위: 밀리초
Success	성공적인 샤드 동기화 작업 수입니다. 측정치 수준: Summary 단위: 개
Time	샤드 동기화 작업에 소요된 시간입니다.

지표	설명
	측정치 수준: Summary 단위: 밀리초

### BlockOnParentTask

샤드가 분할되거나 다른 샤드와 병합되면 새로운 하위 샤드가 생성됩니다. BlockOnParentTask 작업은 KCL이 상위 샤드를 완전히 처리할 때까지 새로운 샤드의 레코드 처리가 시작되지 않도록 보장합니다.

지표	설명
Success	성공적인 상위 샤드 완료 확인 수입입니다. 측정치 수준: Summary 단위: 개
Time	상위 샤드 완료에 소요된 시간입니다. 측정치 수준: Summary 단위: 밀리초

### PeriodicShardSyncManager

PeriodicShardSyncManager는 KCL 소비자 애플리케이션에서 처리 중인 데이터 스트림을 검사하고, 부분 리스가 있는 데이터 스트림을 식별하여 동기화를 위해 전달하는 역할을 합니다.

다음 지표는 KCL이 단일 데이터 스트림을 처리하도록 구성된 경우 ( NumStreamsToSync 및 의 값이 1로 설정됨) 와 NumStreamsWithPartialLeases KCL이 여러 데이터 스트림을 처리하도록 구성된 경우에도 사용할 수 있습니다.

지표	설명
NumStreamsToSync	부분 임대 포함되어 있고 동기화를 위해 전달해야 하는 소비자 애플리케이션에서 처리 중인 데이터 스트림 ( AWS 계정당) 의 수입입니다.

지표	설명
	<p>측정치 수준: Summary</p> <p>단위: 개</p>
NumStreamsWithPartialLeases	<p>소비자 애플리케이션에서 처리 중인 데이터 스트림 중 부분 임대 포함 된 데이터 스트림 수 (AWS 계정당).</p> <p>측정치 수준: Summary</p> <p>단위: 개</p>
Success	<p>PeriodicShardSyncManager 가 소비자 애플리케이션에서 처리 중인 데이터 스트림에서 부분 리스를 성공적으로 식별할 수 있었던 횟수입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 개</p>
Time	<p>샤드 동기화가 필요한 데이터 스트림을 결정하기 위해 PeriodicShardSyncManager 가 소비자 애플리케이션에서 처리 중인 데이터 스트림을 검사하는 데 걸리는 시간(밀리초)입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 밀리초</p>

## MultistreamTracker

MultistreamTracker 인터페이스를 사용하면 여러 데이터 스트림을 동시에 처리할 수 있는 KCL 소비자 애플리케이션을 구축할 수 있습니다.

지표	설명
DeletedStreams.Count	<p>이 기간에 삭제된 데이터 스트림 수입니다.</p> <p>측정치 수준: Summary</p>

지표	설명
	단위: 개
ActiveStreams.카운트	처리 중인 활성 데이터 스트림 수입니다.  측정치 수준: Summary  단위: 개
StreamsPendingDeletion.카운트	FormerStreamsLeasesDeletionStrategy 에 따라 삭제 보류 중인 데이터 스트림 수입니다.  측정치 수준: Summary  단위: 개

## 작업자별 측정치

이러한 지표는 Amazon EC2 인스턴스와 같이 Kinesis 데이터 스트림의 데이터를 사용하는 모든 레코드 프로세서에서 집계됩니다.

주제

- [RenewAllLeases](#)
- [TakeLeases](#)

### RenewAllLeases

RenewAllLeases 작업은 특정 작업자 인스턴스가 소유한 샤드 리스를 주기적으로 갱신합니다.

지표	설명
RenewLease. 성공	작업자의 성공한 리스 갱신 수입니다.  측정치 수준: Detailed  단위: 개
RenewLease.시간	리스 갱신 작업에 소요된 시간입니다.

지표	설명
	<p>측정치 수준: Detailed</p> <p>단위: 밀리초</p>
CurrentLeases	<p>모든 리스가 갱신된 후 작업자가 소유한 샤드 리스 수입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 개</p>
LostLeases	<p>작업자가 소유한 모든 리스를 갱신하려는 시도 후에 손실된 샤드 리스 수입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 개</p>
Success	<p>작업자의 리스 갱신 작업 성공 횟수입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 개</p>
Time	<p>작업자의 모든 리스 갱신에 소요된 시간입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 밀리초</p>

## TakeLeases

TakeLeases 작업은 모든 KCL 워커 간에 레코드 처리의 균형을 유지합니다. 현재 KCL 워커가 필요한 것보다 적은 샤드 리스를 보유하면 오버로드된 다른 KCL 워커에서 샤드 리스를 가져옵니다.

지표	설명
ListLeases. 성공	KCL 애플리케이션 DynamoDB 테이블에서 모든 샤드 리스가 성공적으로 검색된 횟수입니다.



지표	설명
	<p>측정치 수준: Detailed</p> <p>단위: 개</p>
ListLeases.시간	<p>KCL 애플리케이션 DynamoDB 테이블에서 모든 샤드 리스를 검색하는 데 소요된 시간입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 밀리초</p>
TakeLease. 성공	<p>워커가 다른 KCL 워커에서 샤드 리스를 성공적으로 가져온 횟수입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 개</p>
TakeLease.시간	<p>작업자가 가져온 리스로 리스 테이블을 업데이트하는데 소요된 시간입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 밀리초</p>
NumWorkers	<p>특정 작업자가 식별한 총 작업자 수입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 개</p>
NeededLeases	<p>밸런싱된 샤드 처리 로드를 위해 현재 작업자에 필요한 샤드 리스 수입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 개</p>

지표	설명
LeasesToTake	<p>작업자가 가져오려고 시도할 리스 수입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 개</p>
TakenLeases	<p>작업자가 성공적으로 가져온 리스 수입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 개</p>
TotalLeases	<p>KCL 애플리케이션에서 처리 중인 총 샤드 수입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 개</p>
ExpiredLeases	<p>특정 작업자가 식별한 대로 임의의 작업자가 처리하지 않는 총 샤드 수입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 개</p>
Success	<p>TakeLeases 작업이 성공적으로 완료된 횟수입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 개</p>
Time	<p>작업자의 TakeLeases 작업에 소요된 시간입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 밀리초</p>

## 샤드별 측정치

이 측정치는 단일 레코드 프로세서에서 집계됩니다.

## ProcessTask

ProcessTask작업은 현재 반복자 위치를 [GetRecords](#)호출하여 스트림에서 레코드를 검색하고 레코드 프로세서 함수를 호출합니다. processRecords

지표	설명
KinesisDataFetcher .getRecords.Success	Kinesis 데이터 스트림 샤드별로 성공한 GetRecords 작업 수입니다.  측정치 수준: Detailed  단위: 개
KinesisDataFetcher. 기록 가져오기. 시간	Kinesis 데이터 스트림 샤드의 GetRecords 작업당 소요된 시간입니다.  측정치 수준: Detailed  단위: 밀리초
UpdateLease. 성공	지정된 샤드의 레코드 프로세서가 성공적으로 수행한 체크포인트 수입니다.  측정치 수준: Detailed  단위: 개
UpdateLease.시간	지정된 샤드의 각 체크포인트 작업에 소요된 시간입니다.  측정치 수준: Detailed  단위: 밀리초
DataBytesProcessed	각 ProcessTask 호출에서 바이트 단위로 처리된 레코드의 총 크기입니다.  측정치 수준: Summary  단위: 바이트
RecordsProcessed	각 ProcessTask 호출에서 처리된 레코드 수입니다.  측정치 수준: Summary

지표	설명
	단위: 개
ExpiredIterator	<p>통화 GetRecords 시 ExpiredIteratorException 수신된 횟수</p> <p>측정치 수준: Summary</p> <p>단위: 개</p>
MillisBehindLatest	<p>현재 반복자가 샤드의 최신 레코드(팁)에서 뒤쳐진 시간입니다. 이 값은 응답의 최신 레코드와 현재 시간의 시간 차이보다 작거나 같습니다. 마지막 응답 레코드의 타임스탬프와 비교하는 것보다 팁에서 샤드가 얼마나 멀리 있는지 반영하는 것이 더 정확합니다. 각 레코드에 있는 모든 타임스탬프의 평균이 아니라 최신 레코드 배치에 이 값이 적용됩니다.</p> <p>측정치 수준: Summary</p> <p>단위: 밀리초</p>
RecordProcessor. 프로세스 레코드. 시간	<p>레코드 프로세서의 processRecords 메서드에 소요된 시간입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 밀리초</p>
Success	<p>성공적인 process task 작업 수입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 개</p>
Time	<p>process task 작업에 소요된 시간입니다.</p> <p>측정치 수준: Summary</p> <p>단위: 밀리초</p>

# Amazon을 통한 Kinesis 프로듀서 라이브러리 모니터링 CloudWatch

Amazon [Kinesis 데이터 스트림용 Kinesis 프로듀서 라이브러리](#) (KPL) 는 사용자를 대신하여 사용자 지정 Amazon CloudWatch 지표를 게시합니다. [CloudWatch 콘솔로](#) 이동하여 사용자 지정 지표를 선택하면 이러한 지표를 볼 수 있습니다. 사용자 지정 지표에 대한 자세한 내용은 Amazon CloudWatch 사용 설명서의 [사용자 지정 지표 게시](#)를 참조하십시오.

KPL이 업로드한 지표에는 소액의 요금이 CloudWatch 부과되며, 특히 Amazon CloudWatch 커스텀 메트릭과 Amazon CloudWatch API 요청 요금이 적용됩니다. 자세한 내용은 [Amazon CloudWatch 요금](#)을 참조하십시오. 로컬 지표 수집에는 CloudWatch 요금이 부과되지 않습니다.

## 주제

- [측정치, 차원 및 네임스페이스](#)
- [측정치 수준 및 세부 수준](#)
- [로컬 액세스 및 아마존 CloudWatch 업로드](#)
- [측정치 목록](#)

## 측정치, 차원 및 네임스페이스

KPL을 시작할 때 애플리케이션 이름을 지정할 수 있으며, 이 이름은 지표를 업로드할 때 네임스페이스의 일부로 사용됩니다. 이 이름은 선택 사항이며 애플리케이션 이름을 설정하지 않으면 KPL에서 기본 값을 제공합니다.

지표에 임의의 다른 측정기준을 추가하도록 KPL을 구성할 수도 있습니다. 이는 지표에서 더 세밀한 데이터를 원하는 경우에 유용합니다. CloudWatch 예를 들어, 호스트 이름을 차원으로 추가하여 플릿에서 고르지 못한 로드 분산을 식별할 수 있습니다. 모든 KPL 구성 설정은 변경할 수 없으므로 KPL 인스턴스가 초기화된 후에는 이 추가 측정기준을 변경할 수 없습니다.

## 측정치 수준 및 세부 수준

업로드되는 지표 수를 제어하는 두 가지 옵션이 있습니다. CloudWatch

### 지표 수준

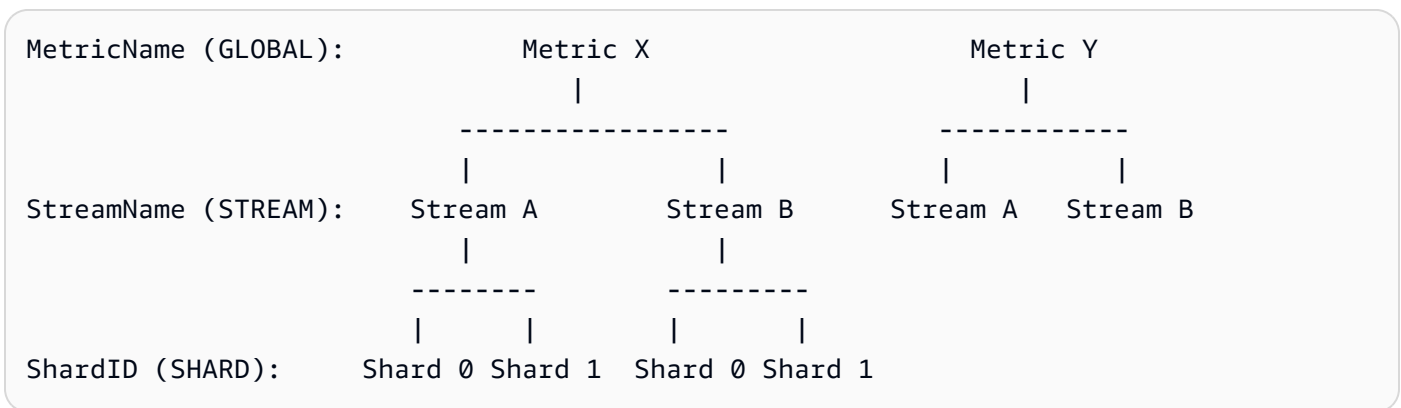
측정치의 중요성을 나타내는 대략적인 게이지입니다. 모든 측정치에는 수준이 할당됩니다. 수준을 설정하면 수준 이하의 지표는 전송되지 않습니다 CloudWatch. 수준은 NONE, SUMMARY 및 DETAILED입니다. 기본 설정은 DETAILED, 즉 모든 지표입니다. NONE은 지표가 전혀 없음을 의미하므로 실제로 해당 수준에 지표가 할당되지 않습니다.

## 세부 수준

추가 세부 수준에서 동일한 측정치를 내보내는지 여부를 제어합니다. 수준은 GLOBAL, STREAM 및 SHARD입니다. 기본 설정은 가장 세분화된 측정치를 포함하는 SHARD입니다.

SHARD를 선택하면 스트림 이름과 샤드 ID를 차원으로 하여 측정치를 내보냅니다. 또한 스트림 이름이 없는 측정치와 스트림 이름 차원만으로도 동일한 측정치를 내보냅니다. 즉, 특정 지표에 대해 각각 샤드가 두 개씩 있는 두 개의 스트림은 샤드당 하나, 스트림당 하나, 전체적으로 하나씩 총 7개의 CloudWatch 지표를 생성합니다. 모두 동일한 통계를 설명하지만 세부 수준은 다릅니다. 다음 다이어그램을 참조하십시오.

여러 세부 수준이 계층 구조를 형성하고 시스템의 모든 지표는 지표 이름을 토대로 트리를 형성합니다.



모든 측정치를 샤드 수준에서 사용할 수 있는 것은 아니며 본질적으로 스트림 수준이나 전역 수준에서 사용할 수 있는 측정치도 있습니다. 샤드 수준 지표를 활성화한 경우에도 이 측정치가 샤드 수준에서 생성되지 않습니다(위 다이어그램의 Metric Y).

추가 차원을 지정할 때 tuple:<DimensionName, DimensionValue, Granularity>에 값을 제공해야 합니다. 사용자 지정 차원이 계층 구조에 삽입되는 위치를 결정하기 위해 세부 수준을 사용합니다. GLOBAL은 지표 이름 뒤, STREAM은 스트림 이름 뒤, SHARD는 샤드 ID 뒤에 추가 차원을 삽입한다는 의미입니다. 여러 추가 차원이 세부 수준별로 제공되면 지정된 순서대로 삽입됩니다.

## 로컬 액세스 및 아마존 CloudWatch 업로드

현재 KPL 인스턴스의 지표는 로컬에서 실시간으로 사용할 수 있으며 언제든지 KPL을 쿼리하여 가져올 수 있습니다. KPL은 에서와 같이 모든 지표의 합계, 평균, 최소값, 최대값, 개수를 로컬에서 계산합니다. CloudWatch

프로그램 시작부터 현재 시점까지 누적된 통계를 가져오거나 지난 N초(N은 1~60 사이의 정수) 동안 롤링 윈도우를 사용하여 통계를 가져올 수 있습니다.

모든 지표를 업로드할 수 있습니다. CloudWatch 특히 여러 호스트의 데이터 집계, 모니터링 및 경보에 유용합니다. 로컬에서는 이 기능을 사용할 수 없습니다.

전에 설명한 대로 측정치 수준 및 세부 수준 설정으로 업로드할 측정치를 선택할 수 있습니다. 업로드되지 않은 측정치를 로컬에서 사용할 수 있습니다.

트래픽이 많을 때 초당 수백만 개의 업로드가 이루어지기 때문에 데이터 요소를 개별적으로 업로드하는 것은 적절하지 않습니다. 이러한 이유로 KPL은 로컬에서 지표를 1분 버킷으로 집계하고 통계 개체를 활성화된 지표당 분당 1회로 업로드합니다. CloudWatch

## 측정치 목록

지표	설명
UserRecordsReceived	<p>널기 작업을 위해 KPL 코어에 수신된 논리적 사용자 레코드 수입입니다. 샤드 수준에서 사용할 수 없습니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 수</p>
UserRecordsPending	<p>현재 대기 중인 사용자 레코드 수의 주기적 샘플입니다. 레코드가 현재 버퍼되고 전송을 기다리고 있거나 백엔드 서비스로 전송 및 이동 중일 경우 레코드는 대기 중 상태입니다. 샤드 수준에서 사용할 수 없습니다.</p> <p>KPL은 전용 메서드를 제공하여 고객이 널기 속도를 관리할 수 있도록 전역 수준에서 이 지표를 검색합니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 수</p>
UserRecordsPut	<p>성공적으로 입력된 논리적 사용자 레코드 수입입니다.</p> <p>KPL은 이 지표의 실패한 레코드를 계산하지 않습니다. 평균을 통해 성공률을, 개수를 통해 총 시도 수를, 개수와 합계의 차이를 통해 실패 수를 알 수 있습니다.</p>

지표	설명
	<p>측정치 수준: Summary</p> <p>단위: 수</p>
UserRecordsDataPut	<p>성공적으로 입력된 논리적 사용자 레코드의 바이트입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 바이트</p>
KinesisRecordsPut	<p>성공적으로 입력된 Kinesis Data Streams 레코드 수입입니다. 각 Kinesis Data Streams 레코드에는 여러 사용자 레코드가 포함될 수 있습니다.</p> <p>KPL은 실패한 레코드에 대해 0을 출력합니다. 평균을 통해 성공률을, 개수를 통해 총 시도 수를, 개수와 합계의 차이를 통해 실패 수를 알 수 있습니다.</p> <p>측정치 수준: Summary</p> <p>단위: 수</p>
KinesisRecordsDataPut	<p>Kinesis Data Streams 레코드의 바이트 수입입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 바이트</p>



지표	설명
ErrorsByCode	<p>각 오류 코드 유형의 개수입니다. ErrorCode 및 StreamName 와 같은 일반 차원 이외에 ShardId의 추가 차원이 있습니다. 모든 오류를 샤드로 추적할 수 있는 것은 아닙니다. 스트림 또는 전역 수준에서만 추적할 수 없는 오류를 내보냅니다. 조절, 샤드 맵 변경, 내부 오류, 서비스 사용 불가, 시간 초과 등에 대한 정보가 이 측정치에 캡처됩니다.</p> <p>Kinesis Data Streams API 오류는 Kinesis Data Streams 레코드당 한 번 계산됩니다. Kinesis Data Streams 레코드에 있는 여러 사용자 레코드는 여러 번 계산되지 않습니다.</p> <p>측정치 수준: Summary</p> <p>단위: 수</p>
AllErrors	<p>Errors by Code(코드별 오류)와 동일한 오류로 트리거되지만 유형을 구별하지 않습니다. 다른 모든 유형의 오류 개수 합계를 수동으로 계산하지 않고도 오류 비율을 전반적으로 모니터링할 수 있어 유용합니다.</p> <p>측정치 수준: Summary</p> <p>단위: 수</p>
RetriesPerRecord	<p>사용자 레코드마다 수행한 재시도 수입니다. 레코드가 한 번에 성공하면 0을 내보냅니다.</p> <p>사용자 레코드가 완료(성공하거나 더 이상 재시도할 수 없는 경우)되면 데이터를 내보냅니다. 레코드 값이 크면 이 time-to-live 지표가 상당히 지연될 수 있습니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 수</p>

지표	설명
BufferingTime	<p>사용자 레코드가 KPL에 도착했다가 백엔드로 나가는 사이의 시간입니다. 레코드별로 이 정보를 사용자에게 다시 전송하지만 집계된 통계로 사용할 수도 있습니다.</p> <p>측정치 수준: Summary</p> <p>단위: 밀리초</p>
Request Time	<p>PutRecordsRequests 를 수행하는 데 소요되는 시간입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 밀리초</p>
User Records per Kinesis Record	<p>단일 Kinesis Data Streams 레코드에 집계된 논리적 사용자 레코드 수입니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 수</p>
Amazon Kinesis Records per PutRecord sRequest	<p>단일 PutRecordsRequest 에 집계된 Kinesis Data Streams 레코드 수입니다. 샤드 수준에서 사용할 수 없습니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 수</p>
User Records per PutRecord sRequest	<p>PutRecordsRequest 에 포함된 총 사용자 레코드 수입니다. 앞의 두 측정치의 결과와 거의 비슷합니다. 샤드 수준에서 사용할 수 없습니다.</p> <p>측정치 수준: Detailed</p> <p>단위: 수</p>

# Amazon Kinesis Data Streams의 보안

클라우드 AWS 보안이 최우선 과제입니다. AWS 고객은 가장 보안에 민감한 조직의 요구 사항을 충족하도록 구축된 데이터 센터 및 네트워크 아키텍처를 활용할 수 있습니다.

보안은 두 사람 사이의 AWS 공동 책임입니다. [공동 책임 모델](#)은 이 사항을 클라우드 내 보안 및 클라우드의 보안으로 설명합니다.

- 클라우드 보안 - AWS 클라우드에서 AWS 서비스를 실행하는 인프라를 보호하는 역할을 합니다. AWS 또한 안전하게 사용할 수 있는 서비스를 제공합니다. 서드 파티 감사자는 정기적으로 [AWS 규정 준수 프로그램](#)의 일환으로 보안 효과를 테스트하고 검증합니다. Kinesis Data Streams에 적용되는 규정 준수 프로그램에 대한 자세한 내용은 [규정 준수 프로그램 제공 범위 내의 AWS 서비스](#)를 참조하세요.
- 클라우드에서의 보안 — 귀하의 책임은 사용하는 AWS 서비스에 따라 결정됩니다. 또한 데이터의 민감도, 조직의 요건 및 관련 법률 및 규정을 비롯한 기타 요소에 대해서도 책임이 있습니다.

이 설명서는 Kinesis Data Streams 사용 시 책임 분담 모델을 적용하는 방법을 이해하는 데 도움이 됩니다. 다음 항목에서는 보안 및 규정 준수 목표를 충족하도록 Kinesis Data Streams를 구성하는 방법을 보여줍니다. 또한 Kinesis Data Streams 리소스를 모니터링하고 보호하는 데 도움이 되는 다른 AWS 서비스를 사용하는 방법도 알아봅니다.

## 주제

- [Amazon Kinesis Data Streams의 데이터 보호](#)
- [IAM을 사용하여 Amazon Kinesis Data Streams 리소스에 대한 액세스 제어](#)
- [Amazon Kinesis Data Streams의 규정 준수 검증](#)
- [Amazon Kinesis Data Streams의 복원력](#)
- [Kinesis Data Streams의 인프라 보안](#)
- [Kinesis Data Streams의 보안 모범 사례](#)

## Amazon Kinesis Data Streams의 데이터 보호

AWS Key Management Service (AWS KMS) 키를 사용한 서버 측 암호화를 사용하면 Amazon Kinesis Data Streams 내에 저장된 데이터를 암호화하여 엄격한 데이터 관리 요구 사항을 쉽게 충족할 수 있습니다.

**Note**

명령줄 인터페이스 또는 API를 AWS 통해 액세스할 때 FIPS 140-2로 검증된 암호화 모듈이 필요한 경우 FIPS 엔드포인트를 사용하십시오. 사용 가능한 FIPS 엔드포인트에 대한 자세한 내용은 [FIPS\(Federal Information Processing Standard\) 140-2](#)를 참조하세요.

## 주제

- [Kinesis Data Streams용 서버 측 암호화란?](#)
- [비용, 리전 및 성능 고려 사항](#)
- [서버 측 암호화를 시작하는 방법](#)
- [사용자 생성 KMS 마스터 키 생성 및 사용](#)
- [사용자 생성 KMS 마스터 키 사용 권한](#)
- [KMS 키 권한 확인 및 문제 해결](#)
- [인터페이스 VPC 엔드포인트와 함께 Amazon Kinesis Data Streams 사용](#)

## Kinesis Data Streams용 서버 측 암호화란?

서버 측 암호화는 사용자가 지정한 AWS KMS 고객 마스터 키 (CMK) 를 사용하여 데이터가 저장되기 전에 자동으로 암호화하는 Amazon Kinesis Data Streams의 기능입니다. 데이터는 Kinesis 스트림 스토리지 계층에 쓰여지기 전에 암호화되고 스토리지에서 검색된 후 해독됩니다. 결과적으로 데이터는 Kinesis Data Streams 서비스에서 유희 시 암호화되어 엄격한 규제 요구 사항을 충족시키고 데이터 보안을 강화할 수 있습니다.

서버 측 암호화를 사용하면 Kinesis 스트림 생산자 및 소비자가 마스터 키나 암호화 작업을 관리할 필요가 없습니다. Kinesis Data Streams 서비스에 들어오고 나가는 데이터는 자동으로 암호화되므로 저장된 데이터는 암호화됩니다. AWS KMS 서버 측 암호화 기능에 사용되는 모든 마스터 키를 제공합니다. AWS KMS 에서 관리하는 AWS Kinesis용 CMK, AWS KMS 사용자 지정 CMK 또는 서비스로 가져온 마스터 키를 쉽게 사용할 수 있습니다. AWS KMS

**Note**

서버 측 암호화는 암호화가 활성화된 직후에 수신되는 데이터만 암호화합니다. 암호화되지 않은 스트림의 기존 데이터는 서버 측 암호화가 활성화 된 후에도 암호화되지 않습니다.

데이터 스트림을 암호화하고 다른 보안 주체와 액세스를 공유하는 경우 키에 대한 키 정책과 외부 계정의 IAM 정책 모두에서 권한을 부여해야 합니다. AWS KMS 자세한 내용은 [다른 계정의 사용자가 KMS 키를 사용하도록 허용](#)을 참조하세요.

AWS 관리형 KMS 키로 데이터 스트림에 대해 서버 측 암호화를 활성화한 후 리소스 정책을 통해 액세스를 공유하려면 다음과 같이 고객 관리 키 (CMK) 를 사용하도록 전환해야 합니다.

**Edit encryption for test\_encryption**

**Encryption** [Info](#)

- Enable server-side encryption**  
Kinesis Data Stream uses AWS Key Management Service (KMS) to encrypt your data. You can choose the AWS managed customer master key (CMK) to encrypt your data or specify a customer-managed CMK.
- Use AWS managed CMK**  
The AWS managed CMK (aws/kinesis) in your account is created, managed, and used on your behalf by Kinesis Data Streams.
- Use customer-managed CMK**  
Customer-managed CMKs in your AWS account are created, owned, and managed by you.

Customer-managed CMK in KMS

또한 KMS 크로스 계정 공유 기능을 사용하여 공유 보안 주체 엔터티가 CMK에 액세스할 수 있도록 허용해야 합니다. 공유 보안 주체 엔터티에 대한 IAM 정책도 변경해야 합니다. 자세한 내용은 [다른 계정의 사용자가 KMS 키를 사용하도록 허용](#)을 참조하세요.

## 비용, 리전 및 성능 고려 사항

서버 측 암호화를 적용할 경우 API 사용 및 키 비용이 부과됩니다. AWS KMS 사용자 지정 KMS 마스터 키와는 달리 (Default) aws/kinesis 사용자 지정 마스터 키(CMK)는 무료로 제공됩니다. 하지만 Amazon Kinesis Data Streams가 사용자를 대신하여 초래하는 API 사용 비용을 지불해야 합니다.

API 사용 비용은 사용자 지정을 포함한 모든 CMK에 적용됩니다. Kinesis Data Streams는 데이터 키를 교체할 때 약 5분마다 한 번씩 AWS KMS 를 직접적으로 호출합니다. 한 달에 30일 기준으로 Kinesis 스트림으로 시작된 AWS KMS API 호출의 총 비용은 몇 달러 미만이어야 합니다. 각 사용자 자격 증명에는 고유한 API 호출이 필요하기 때문에 이 비용은 데이터 생산자와 소비자가 사용하는 사용자 자격

증명의 수에 따라 조정됩니다. AWS KMS IAM 역할을 권한 부여에 사용하면 각 역할 수입 호출은 결국 고유의 사용자 보안 인증이 됩니다. KMS 비용 절감을 위해 역할 수입 호출에 의해 반환된 사용자 자격 증명을 캐시하고자 할 수 있습니다.

다음은 리소스별 비용에 대한 설명입니다.

## 키

- (별칭 `aws/kinesis =`) 에서 관리하는 AWS Kinesis용 CMK는 무료입니다.
- 사용자 생성 KMS 키에는 KMS 키 비용이 적용됩니다. 자세한 내용은 [AWS Key Management Service 요금](#)을 참조하세요.

API 사용 비용은 사용자 지정을 포함한 모든 CMK에 적용됩니다. Kinesis Data Streams는 데이터 키를 교체할 때 약 5분마다 한 번씩 KMS를 직접적으로 호출합니다. 월 30일 동안 Kinesis 데이터 스트림을 통해 시작된 KMS API 호출의 총 비용은 몇 달러 미만이어야 합니다. 각 사용자 자격 증명에는 KMS에 대한 고유한 API 호출이 필요하므로 이 비용은 데이터 생산자와 소비자에게 사용하는 사용자 자격 증명의 수에 따라 조정된다는 점에 유의하십시오. AWS IAM 역할을 인증에 사용하면 각각 고유한 사용자 자격 증명에 `assume-role-call` 생성되므로 KMS 비용을 `assume-role-call` 절약하기 위해 에서 반환한 사용자 자격 증명을 캐시에 저장하는 것이 좋습니다.

## KMS API 사용

모든 암호화된 스트림에 대해 TIP에서 읽고 리더와 라이터 전체에서 단일 IAM 계정/사용자 액세스 키를 사용할 때 Kinesis 서비스는 5분마다 약 12번 AWS KMS 서비스를 직접적으로 호출합니다. TIP에서 읽지 않으면 서비스 요청 건수가 증가할 수 있습니다. AWS KMS 새 데이터 암호화 키를 생성하기 위한 API 요청에는 AWS KMS 사용 비용이 부과됩니다. 자세한 내용은 [AWS Key Management Service 요금: 사용량](#)을 참조하세요.

## 리전별 서버 측 암호화 가용성

현재 Kinesis 스트림의 서버 측 암호화는 Kinesis Data Streams가 지원되는 모든 지역 (미국 서부) 과 중국 지역에서 사용할 수 있습니다. AWS GovCloud Kinesis Data Streams가 지원되는 리전에 대한 자세한 내용은 <https://docs.aws.amazon.com/general/latest/gr/ak.html>을 참조하세요.

## 성능 고려 사항

암호화를 적용할 경우 생기는 서비스 오버헤드로 인해 서버 측 암호화를 적용하면 `PutRecord`, `PutRecords` 및 `GetRecords`의 일반적인 지연 시간이 100µs 미만 증가합니다.

## 서버 측 암호화를 시작하는 방법

서버 측 암호화를 시작하는 가장 쉬운 방법은 AWS Management Console 및 Amazon Kinesis KMS 서비스 키를 사용하는 것입니다. `aws/kinesis`

다음 절차는 Kinesis 스트림에 서버 측 암호화를 사용하도록 설정하는 방법을 보여줍니다.

Kinesis 스트림에 서버 측 암호화를 사용하도록 설정

1. `aws`에 AWS Management Console 로그인하여 [Amazon Kinesis 데이터 스트림](#) 콘솔을 엽니다.
2. AWS Management Console에서 Kinesis 스트림을 생성하거나 선택합니다.
3. `details`(세부 정보) 탭을 선택합니다.
4. `Server-side encryption`(서버 측 암호화)에서 `edit`(편집)를 선택합니다.
5. 사용자 생성 KMS 마스터 키를 사용하려는 경우가 아니라면 (Default) `aws/kinesis` KMS 마스터 키를 선택해야 합니다. 이 키는 Kinesis 서비스에서 생성한 KMS 마스터 키입니다. `Enabled`(활성)를 선택한 다음 `Save`(저장)를 선택합니다.

### Note

기본 Kinesis 서비스 마스터 키는 무료이지만 Kinesis에서 AWS KMS 서비스에 대해 수행하는 API 호출에는 KMS 사용 요금이 적용됩니다.

6. 스트림이 보류 중 상태로 전환됩니다. 암호화가 활성화된 상태에서 스트림이 활성 상태로 전환되면 스트림에 쓰여진 모든 수신 데이터는 선택한 KMS 마스터 키를 사용하여 암호화됩니다.
7. 서버 측 암호화를 비활성화하려면 의 서버 측 암호화에서 비활성화됨을 선택한 다음 저장을 선택합니다. AWS Management Console

## 사용자 생성 KMS 마스터 키 생성 및 사용

이 섹션에서는 Amazon Kinesis에서 관리하는 마스터 키 대신 자체 KMS 마스터 키를 생성하고 사용하는 방법을 설명합니다.

### 사용자 생성 KMS 마스터 키 만들기

자체 마스터 키 생성에 대한 지침은 AWS Key Management Service 개발자 안내서의 [Creating Keys](#)를 참조하세요. 계정 키를 생성하면 Kinesis Data Streams 서비스가 KMS 마스터 키 목록에 이 키를 반환합니다.

## 사용자 생성 KMS 마스터 키 사용

소비자, 생산자, 관리자에게 올바른 권한을 적용한 후에는 자신의 AWS 계정이나 다른 계정에서 사용자 지정 KMS 마스터 키를 사용할 수 있습니다. AWS 계정의 모든 KMS 마스터 키는 AWS Management Console의 KMS 마스터 키 목록에 표시됩니다.

다른 계정에 있는 사용자 지정 KMS 마스터 키를 사용하려면 해당 키를 사용할 수 있는 권한이 필요합니다. 또한 AWS Management Console의 ARN 입력란에서 KMS 마스터 키의 ARN을 지정해야 합니다.

## 사용자 생성 KMS 마스터 키 사용 권한

사용자가 생성한 KMS 마스터 키로 서버 측 암호화를 사용하려면 먼저 스트림의 암호화와 스트림 레코드의 암호화와 암호 해독을 허용하도록 AWS KMS 키 정책을 구성해야 합니다. 권한에 대한 예와 자세한 내용은 [AWS KMS API AWS KMS](#) 권한: 작업 및 리소스 참조를 참조하십시오.

### Note

암호화에 기본 서비스 키를 사용할 때는 사용자 지정 IAM 권한을 적용할 필요가 없습니다.

사용자 생성 KMS 마스터 키를 사용하려면 먼저 Kinesis 스트림 생산자 및 소비자(IAM 보안 주체)가 KMS 마스터 키 정책에 속한 사용자여야 합니다. 그렇지 않으면 스트림에서 읽기 및 쓰기가 실패하여 궁극적으로 데이터 손실, 처리 지연 또는 애플리케이션 중단이 발생할 수 있습니다. IAM 정책을 사용하여 KMS 키 권한을 관리할 수 있습니다. 자세한 내용은 KMS에서 [IAM 정책 사용](#)을 참조하십시오. AWS

## 예제 생산자 권한

Kinesis 스트림 생산자에 kms:GenerateDataKey 권한이 있어야 합니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "kms:GenerateDataKey"
      ],
      "Resource": "arn:aws:kms:us-west-2:123456789012:key/1234abcd-12ab-34cd-56ef-1234567890ab"
    }
  ]
}
```



```

    },
    {
      "Effect": "Allow",
      "Action": [
        "kinesis:PutRecord",
        "kinesis:PutRecords"
      ],
      "Resource": "arn:aws:kinesis:*:123456789012:MyStream"
    }
  ]
}

```

## 예제 소비자 권한

Kinesis 스트림 소비자에 kms:Decrypt 권한이 있어야 합니다.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "kms:Decrypt"
      ],
      "Resource": "arn:aws:kms:us-west-2:123456789012:key/1234abcd-12ab-34cd-56ef-1234567890ab"
    },
    {
      "Effect": "Allow",
      "Action": [
        "kinesis:GetRecords",
        "kinesis:DescribeStream"
      ],
      "Resource": "arn:aws:kinesis:*:123456789012:MyStream"
    }
  ]
}

```

아파치 플링크용 아마존 매니지드 서비스 및 역할을 AWS Lambda 사용하여 Kinesis 스트림을 사용합니다. 이 소비자가 사용하는 역할에 kms:Decrypt 권한을 추가해야 합니다.

## 스트림 관리자 권한

Kinesis 스트림 관리자는 `kms:List*` 및 `kms:DescribeKey*`를 직접적으로 호출할 수 있는 권한 부여가 필요합니다.

## KMS 키 권한 확인 및 문제 해결

Kinesis 스트림에서 암호화를 활성화한 후에는 다음 Amazon CloudWatch 지표를 사용하여 `PutRecord`, `PutRecords`, `GetRecords` 호출의 성공 여부를 모니터링하는 것이 좋습니다.

- `PutRecord.Success`
- `PutRecords.Success`
- `GetRecords.Success`

자세한 내용은 [Amazon Kinesis Data Streams 모니터링](#) 단원을 참조하세요.

## 인터페이스 VPC 엔드포인트와 함께 Amazon Kinesis Data Streams 사용

인터페이스 VPC 엔드포인트를 사용하여 Amazon VPC와 Kinesis Data Streams 간 트래픽이 Amazon 네트워크를 벗어나지 않도록 할 수 있습니다. 인터페이스 VPC 엔드포인트에는 인터넷 게이트웨이, NAT 디바이스, VPN 연결 또는 연결이 필요하지 않습니다. AWS Direct Connect 인터페이스 VPC 엔드포인트는 Amazon VPC의 사설 IP가 있는 Elastic Network 인터페이스를 사용하여 AWS 서비스 간 사설 통신을 가능하게 하는 AWS 기술인 에 의해 AWS PrivateLink 구동됩니다. 자세한 내용은 [Amazon Virtual Private 클라우드 및 인터페이스 VPC 엔드포인트](#) () 를 참조하십시오. AWS PrivateLink

### 주제

- [Kinesis Data Streams에 인터페이스 VPC 엔드포인트 사용](#)
- [Kinesis Data Streams의 VPCE 엔드포인트에 대한 액세스 제어](#)
- [Kinesis Data Streams에 대한 VPC 엔드포인트 정책 가용성](#)

## Kinesis Data Streams에 인터페이스 VPC 엔드포인트 사용

시작할 때 스트림, 생산자 또는 소비자에 대한 설정을 변경할 필요가 없습니다. Amazon VPC 리소스에서 나오고 들어가는 Kinesis Data Streams 트래픽이 인터페이스 VPC 엔드포인트를 통해 흐름을 시작하려면 간단히 인터페이스 VPC 엔드포인트를 생성하면 됩니다. 자세한 내용은 [인터페이스 엔드포인트 생성](#)을 참조하세요.

Kinesis 프로듀서 라이브러리 (KPL) 와 Kinesis 소비자 라이브러리 (KCL) 는 퍼블릭 엔드포인트 또는 프라이빗 인터페이스 VPC 엔드포인트 중 사용 중인 엔드포인트를 사용하여 AWS Amazon CloudWatch 및 Amazon DynamoDB와 같은 서비스를 호출합니다. 예를 들어 DynamoDB 인터페이스 VPC 엔드포인트가 활성화된 VPC에서 KCL 애플리케이션이 실행 중인 경우 DynamoDB와 KCL 애플리케이션 간의 호출은 인터페이스 VPC 엔드포인트를 통해 흐릅니다.

## Kinesis Data Streams의 VPCE 엔드포인트에 대한 액세스 제어

VPC 엔드포인트 정책을 사용하면 정책을 VPC 엔드포인트에 연결하거나 IAM 사용자, 그룹 또는 역할에 연결된 정책의 추가 필드를 사용하여 액세스를 제어할 수 있습니다. 이를 통해 지정된 VPC 엔드포인트를 통해서만 발생하도록 액세스를 제한할 수 있습니다. 이러한 정책을 사용하면 IAM 정책과 함께 사용하여 지정된 VPC 엔드포인트를 통해 Kinesis 데이터 스트림 작업에 대한 액세스 권한만 부여할 때 특정 스트림에 대한 액세스를 지정된 VPC 엔드포인트로 제한할 수 있습니다.

다음은 Kinesis 데이터 스트림에 액세스하기 위한 엔드포인트 정책의 예입니다.

- VPC 정책 예제: 읽기 전용 액세스 - 이 샘플 정책은 VPC 엔드포인트에 연결할 수 있습니다. 자세한 내용은 [Amazon VPC 리소스에 대한 액세스 제어](#)를 참조하십시오. 연결된 VPC 엔드포인트를 통해서만 Kinesis 데이터 스트림을 나열하고 설명할 수 있도록 작업을 제한합니다.

```
{
  "Statement": [
    {
      "Sid": "ReadOnly",
      "Principal": "*",
      "Action": [
        "kinesis:List*",
        "kinesis:Describe*"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

- VPC 정책 예: 특정 Kinesis 데이터 스트림에 대한 액세스 제한 - 이 샘플 정책은 VPC 엔드포인트에 연결할 수 있습니다. 연결된 VPC 엔드포인트를 통한 특정 데이터 스트림에 대한 액세스를 제한합니다.

```
{
  "Statement": [
    {
      "Sid": "AccessToSpecificDataStream",
      "Principal": "*",
      "Action": "kinesis:*",
      "Effect": "Allow",
      "Resource": "arn:aws:kinesis:us-east-1:123456789012:stream/MyStream"
    }
  ]
}
```

- IAM 정책 예: 특정 VPC 엔드포인트의 특정 스트림에 대한 액세스만 제한 - 이 샘플 정책은 IAM 사용자, 역할 또는 그룹에 연결할 수 있습니다. 지정된 VPC 엔드포인트에서만 발생하도록 지정된 Kinesis 데이터 스트림에 대한 액세스를 제한합니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AccessFromSpecificEndpoint",
      "Action": "kinesis:*",
      "Effect": "Deny",
      "Resource": "arn:aws:kinesis:us-east-1:123456789012:stream/MyStream",
      "Condition": { "StringNotEquals" : { "aws:sourceVpce": "vpce-11aa22bb" } }
    }
  ]
}
```

## Kinesis Data Streams에 대한 VPC 엔드포인트 정책 가용성

정책이 포함된 Kinesis Data Streams 인터페이스 VPC 엔드포인트는 다음 리전에서 지원됩니다.

- 유럽(파리)
- 유럽(아일랜드)
- 미국 동부(버지니아 북부)
- 유럽(스톡홀름)

- 미국 동부(오하이오)
- 유럽(프랑크푸르트)
- 남아메리카(상파울루)
- 유럽(런던)
- 아시아 태평양(도쿄)
- 미국 서부(캘리포니아 북부)
- 아시아 태평양(싱가포르)
- 아시아 태평양(시드니)
- 중국(베이징)
- 중국(닝샤)
- 아시아 태평양(홍콩)
- 중동(바레인)
- 중동(UAE)
- 유럽(밀라노)
- 아프리카(케이프타운)
- 아시아 태평양(뭄바이)
- 아시아 태평양(서울)
- 캐나다(중부)
- usw2-az4를 제외한 미국 서부(오레곤)
- AWS GovCloud (미국 동부)
- AWS GovCloud (미국 서부)
- 아시아 태평양(오사카)
- 유럽(취리히)
- 아시아 태평양(하이데라바드)

## IAM을 사용하여 Amazon Kinesis Data Streams 리소스에 대한 액세스 제어

AWS Identity and Access Management (IAM) 을 사용하면 다음 작업을 수행할 수 있습니다.

- 계정에서 사용자 및 그룹 생성 AWS

- AWS 계정 내 각 사용자에게 고유한 보안 자격 증명을 할당하세요.
- AWS 리소스를 사용하여 작업을 수행할 수 있는 각 사용자의 권한을 제어하세요.
- 다른 AWS 계정의 사용자가 AWS 리소스를 공유하도록 허용
- AWS 계정의 역할을 생성하고 역할을 맡을 수 있는 사용자 또는 서비스를 정의하세요.
- 기업의 기존 ID를 사용하여 리소스를 사용하여 AWS 작업을 수행할 수 있는 권한을 부여하세요.

IAM과 Kinesis Data Streams를 함께 사용하면 조직 내 사용자별로 특정 Kinesis Data Streams API 작업을 사용하여 작업을 수행할 수 있는지 여부와 특정 AWS 리소스를 사용할 수 있는지 여부를 제어할 수 있습니다.

Kinesis 클라이언트 라이브러리 (KCL) 를 사용하여 애플리케이션을 개발하는 경우 정책에 Amazon DynamoDB 및 Amazon에 대한 권한이 포함되어야 합니다. CloudWatch KCL은 DynamoDB를 사용하여 애플리케이션의 상태 정보를 추적하고 KCL 지표를 사용자 대신 전송합니다. CloudWatch CloudWatch KCL에 대한 자세한 내용은 [KCL 1.x 소비자 개발](#) 단원을 참조하십시오.

IAM에 대한 자세한 내용은 다음을 참조하십시오.

- [AWS Identity and Access Management \(IAM\)](#)
- [시작하기](#)
- [IAM 사용 설명서](#)

IAM 및 Amazon DynamoDB에 대한 자세한 내용은 Amazon DynamoDB 개발자 안내서의 [Using IAM to Control Access to Amazon DynamoDB Resources](#)를 참조하세요.

IAM 및 Amazon에 대한 자세한 내용은 Amazon CloudWatch 사용 설명서의 AWS 계정에 대한 CloudWatch 사용자 [액세스 제어](#)를 참조하십시오.

## 내용

- [정책 구문](#)
- [Kinesis Data Streams에 대한 작업](#)
- [Kinesis Data Streams용 Amazon 리소스 이름\(ARN\)](#)
- [Kinesis Data Streams에 대한 예제 정책](#)
- [다른 계정과 데이터 스트림 공유](#)
- [AWS Lambda 함수를 다른 계정의 Kinesis Data Streams에서 읽도록 구성](#)
- [리소스 기반 정책을 사용하여 액세스 공유](#)

## 정책 구문

IAM 정책은 하나 이상의 문으로 구성된 JSON 문서입니다. 각 명령문의 구조는 다음과 같습니다.

```
{
  "Statement": [{
    "Effect": "effect",
    "Action": "action",
    "Resource": "arn",
    "Condition": {
      "condition": {
        "key": "value"
      }
    }
  }
]
```

명령문을 이루는 요소는 다양합니다.

- 효과: effect는 Allow 또는 Deny일 수 있습니다. 기본적으로 IAM 사용자에게는 리소스 및 API 작업을 사용할 권한이 없으므로 모든 요청이 거부됩니다. 명시적 허용은 기본 설정을 무시합니다. 명시적 거부는 모든 허용을 무시합니다.
- 작업: 작업은 권한을 부여하거나 거부할 특정 API 작업입니다.
- 리소스: 작업의 영향을 받는 리소스입니다. 문에서 리소스를 지정하려면 Amazon 리소스 이름(ARN)을 사용해야 합니다.
- 조건(Condition): 조건(Condition)은 선택 사항입니다. 정책이 적용되는 시점을 제어하는 데 사용할 수 있습니다.

IAM 정책을 생성하고 관리할 때 [IAM 정책 생성기](#) 및 [IAM 정책 시뮬레이터](#)를 사용하려고 할 수 있습니다.

## Kinesis Data Streams에 대한 작업

IAM 정책 설명에는 IAM을 지원하는 모든 서비스의 모든 API 작업을 지정할 수 있습니다. Kinesis Data Streams의 경우 접두사 `kinesis:`와 함께 API 작업 이름을 사용합니다. 예를 들어, `kinesis:CreateStream`, `kinesis:ListStreams` 및 `kinesis:DescribeStreamSummary`입니다.

문 하나에 여러 작업을 지정하려면 다음과 같이 쉼표로 구분합니다.

```
"Action": ["kinesis:action1", "kinesis:action2"]
```

와일드카드를 사용하여 여러 작업을 지정할 수도 있습니다. 예를 들어 다음과 같이 이름이 "Get"으로 시작되는 모든 작업을 지정할 수 있습니다.

```
"Action": "kinesis:Get*"
```

모든 Kinesis Data Streams 작업을 지정하려면 다음과 같이 \* 와일드카드를 사용하세요.

```
"Action": "kinesis:*"
```

Kinesis Data Streams API 작업의 전체 목록은 [Amazon Kinesis API 참조](#)를 확인하세요.

## Kinesis Data Streams용 Amazon 리소스 이름(ARN)

각 IAM 정책 명령문은 ARN을 사용하여 지정한 리소스에 적용됩니다.

Kinesis 데이터 스트림에는 다음 ARN 리소스 형식을 사용합니다.

```
arn:aws:kinesis:region:account-id:stream/stream-name
```

예:

```
"Resource": arn:aws:kinesis:*:111122223333:stream/my-stream
```

## Kinesis Data Streams에 대한 예제 정책

다음은 Kinesis 데이터 스트림에 대한 사용자 액세스를 제어하는 방법을 설명하는 예제 정책입니다.

Example 1: Allow users to get data from a stream

### Example

이 정책을 사용하면 사용자 또는 그룹이 지정된 스트림에 대해 DescribeStreamSummary, GetShardIterator 및 GetRecords를 수행할 수 있으며 모든 스트림에 대해 ListStreams를 수행할 수 있습니다. 이 정책은 특정 스트림에서 데이터를 가져올 수 있는 사용자에게 적용할 수 있습니다.



```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "kinesis:Get*",
        "kinesis:DescribeStreamSummary"
      ],
      "Resource": [
        "arn:aws:kinesis:us-east-1:111122223333:stream/stream1"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "kinesis:ListStreams"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

## Example 2: Allow users to add data to any stream in the account

### Example

이 정책을 사용하면 사용자 또는 그룹이 계정의 스트림에 PutRecord 작업을 사용할 수 있습니다. 이 정책은 계정의 모든 스트림에 데이터 레코드를 추가할 수 있는 사용자에 적용할 수 있습니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "kinesis:PutRecord"
      ],
      "Resource": [
        "arn:aws:kinesis:us-east-1:111122223333:stream/*"
      ]
    }
  ]
}
```

```

    }
  ]
}

```

### Example 3: Allow any Kinesis Data Streams action on a specific stream

#### Example

이 정책을 사용하면 사용자 또는 그룹이 지정된 스트림에 대해 Kinesis Data Streams 작업을 사용할 수 있습니다. 이 정책은 특정 스트림에 대해 관리 제어 권한이 있는 사용자에게 적용할 수 있습니다.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "kinesis:*",
      "Resource": [
        "arn:aws:kinesis:us-east-1:111122223333:stream/stream1"
      ]
    }
  ]
}

```

### Example 4: Allow any Kinesis Data Streams action on any stream

#### Example

이 정책을 사용하면 사용자 또는 그룹이 계정의 모든 스트림에 대해 Kinesis Data Streams 작업을 사용할 수 있습니다. 이 정책은 모든 스트림에 대한 모든 액세스 권한을 부여하므로 관리자에게만 해당하도록 제한해야 합니다.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "kinesis:*",
      "Resource": [
        "arn:aws:kinesis*:111122223333:stream/*"
      ]
    }
  ]
}

```

```
]
}
```

## 다른 계정과 데이터 스트림 공유

### Note

Kinesis 프로듀서 라이브러리는 현재 데이터 스트림에 쓸 때 스트림 ARN을 지정하는 것을 지원하지 않습니다. 크로스 계정 데이터 스트림에 쓰려면 AWS SDK를 사용하십시오.

[리소스 기반 정책](#)을 데이터 스트림에 연결하여 다른 계정, IAM 사용자 또는 IAM 역할에 액세스 권한을 부여합니다. 리소스 기반 정책은 데이터 스트림과 같은 리소스에 연결하는 JSON 정책 문서입니다. 이 정책은 [지정된 보안 주체](#)에 해당 리소스에 대한 특정 작업을 수행할 수 있는 권한을 부여하고 이 권한이 적용되는 조건을 정의합니다. 정책에는 여러 명령문이 있을 수 있습니다. 리소스 기반 정책에서 보안 주체를 지정해야 합니다. 보안 주체에는 계정, 사용자, 역할, 연동 사용자 또는 서비스가 포함될 수 있습니다. AWS Kinesis Data Streams 콘솔, API 또는 SDK에서 정책을 구성할 수 있습니다.

[향상된 팬아웃](#)과 같은 등록된 소비자와의 액세스를 공유하려면 데이터 스트림 ARN 및 소비자 ARN 모두에 정책이 필요합니다.

### 크로스 계정 액세스 활성화

계정 간 액세스를 활성화하려는 경우 전체 계정이나 다른 계정의 IAM 엔티티를 리소스 기반 정책의 보안 주체로 지정할 수 있습니다. 리소스 기반 정책에 크로스 계정 보안 주체를 추가하는 것은 트러스트 관계 설정의 절반밖에 되지 않는다는 것을 유념하세요. 주체와 리소스가 별도의 AWS 계정에 있는 경우 ID 기반 정책을 사용하여 주체에게 리소스에 대한 액세스 권한을 부여해야 합니다. 하지만 리소스 기반 정책이 동일 계정의 보안 주체에 액세스를 부여하는 경우 추가 자격 증명 기반 정책이 필요하지 않습니다.

크로스 계정 액세스에 대해 리소스 기반 정책을 사용하는 방법에 대한 자세한 내용은 [IAM에서 크로스 계정 리소스 액세스](#)를 참조하세요.

데이터 스트림 관리자는 AWS Identity and Access Management 정책을 사용하여 누가 무엇에 액세스할 수 있는지 지정할 수 있습니다. 즉, 어떤 보안 주체가 어떤 리소스와 어떤 조건에서 작업을 수행할 수 있는지를 지정할 수 있습니다. JSON 정책의 Action 요소는 정책에서 액세스를 허용하거나 거부하는데 사용할 수 있는 태스크를 설명합니다. 정책 작업은 일반적으로 관련 AWS API 작업과 이름이 같습니다.

## 공유할 수 있는 Kinesis Data Streams 작업:


작업	액세스 수준
<a href="#">DescribeStreamConsumers</a>	소비자
<a href="#">DescribeStreamSummary</a>	데이터 스트림
<a href="#">GetRecords</a>	데이터 스트림
<a href="#">GetShardIterator</a>	데이터 스트림
<a href="#">ListShards</a>	데이터 스트림
<a href="#">PutRecord</a>	데이터 스트림
<a href="#">PutRecords</a>	데이터 스트림
<a href="#">SubscribeToShard</a>	소비자

다음은 리소스 기반 정책을 사용하여 데이터 스트림 또는 등록된 소비자에게 크로스 계정 액세스 권한을 부여하는 예제입니다.

크로스 계정 작업을 수행하려면 데이터 스트림 액세스를 위한 스트림 ARN과 등록된 소비자 액세스를 위한 소비자 ARN을 지정해야 합니다.

## Kinesis 데이터 스트림에 대한 예제 리소스 기반 정책

등록된 소비자를 공유하려면 필요한 조치 때문에 데이터 스트림 정책과 소비자 정책이 모두 포함됩니다.

 Note

다음은 Principal에 대한 유효한 값의 예제입니다.

- {"AWS": "123456789012"}
- IAM 사용자 – {"AWS": "arn:aws:iam::123456789012:user/user-name"}
- IAM 역할 – {"AWS": ["arn:aws:iam::123456789012:role/role-name"]}

- 여러 보안 주체(계정, 사용자, 역할의 조합일 수 있음) - {"AWS":["123456789012", "123456789013", "arn:aws:iam::123456789012:user/user-name"]}

## Example 1: Write access to the data stream

### Example

```
{
  "Version": "2012-10-17",
  "Id": "__default_write_policy_ID",
  "Statement": [
    {
      "Sid": "writestatement",
      "Effect": "Allow",
      "Principal": {
        "AWS": "Account12345"
      },
      "Action": [
        "kinesis:DescribeStreamSummary",
        "kinesis:ListShards",
        "kinesis:PutRecord",
        "kinesis:PutRecords"
      ],
      "Resource": "arn:aws:kinesis:us-east-2:123456789012:stream/
datastreamABC"
    }
  ]
}
```

## Example 2: Read access to the data stream

### Example

```
{
  "Version": "2012-10-17",
  "Id": "__default_sharedthroughput_read_policy_ID",
  "Statement": [
    {
      "Sid": "sharedthroughputreadstatement",
      "Effect": "Allow",
      "Principal": {
```

```

        "AWS": "Account12345"
    },
    "Action": [
        "kinesis:DescribeStreamSummary",
        "kinesis:ListShards",
        "kinesis:GetRecords",
        "kinesis:GetShardIterator"
    ],
    "Resource": "arn:aws:kinesis:us-east-2:123456789012:stream/
datastreamABC"
    }
]
}

```

### Example 3: Share enhanced fan-out read access to a registered consumer

#### Example

데이터 스트림 정책 명령문:

```

{
  "Version": "2012-10-17",
  "Id": "__default_sharedthroughput_read_policy_ID",
  "Statement": [
    {
      "Sid": "consumerreadstatement",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::Account12345:role/role-name"
      },
      "Action": [
        "kinesis:DescribeStreamSummary",
        "kinesis:ListShards"
      ],
      "Resource": "arn:aws:kinesis:us-east-2:123456789012:stream/
datastreamABC"
    }
  ]
}

```

소비자 정책 명령문:

```

{

```

```

"Version": "2012-10-17",
"Id": "__default_efs_read_policy_ID",
"Statement": [
  {
    "Sid": "eforeadstatement",
    "Effect": "Allow",
    "Principal": {
      "AWS": "arn:aws:iam::Account12345:role/role-name"
    },
    "Action": [
      "kinesis:DescribeStreamConsumer",
      "kinesis:SubscribeToShard"
    ],
    "Resource": "arn:aws:kinesis:us-east-2:123456789012:stream/
datastreamABC/consumer/consumerDEF:1674696300"
  }
]
}

```

최소 권한 원칙을 유지하기 위해 작업 또는 보안 주체 필드에서는 와일드카드(\*)가 지원되지 않습니다.

## 프로그래밍 방식으로 데이터 스트림에 대한 정책 관리

이외에도 Kinesis Data Streams에는 데이터 스트림 정책을 관리하기 위한 세 가지 API가 있습니다.

AWS Management Console

- [PutResourcePolicy](#)
- [GetResourcePolicy](#)
- [DeleteResourcePolicy](#)

데이터 스트림 또는 소비자에 대한 정책을 연결하거나 덮어쓰려면 `PutResourcePolicy`를 사용합니다. 지정된 데이터 스트림 또는 소비자에 대한 정책을 확인하고 조회하려면 `GetResourcePolicy`를 사용합니다. 지정된 데이터 스트림 또는 소비자에 대한 정책을 삭제하려면 `DeleteResourcePolicy`를 사용합니다.

## 정책 제한

Kinesis Data Streams 리소스 정책에는 다음과 같은 제한이 있습니다.

- 와일드카드 (\*) 는 데이터 스트림 또는 등록된 소비자에게 직접 연결된 리소스 정책을 통해 광범위한 액세스가 부여되는 것을 방지하기 위해 지원되지 않습니다. 또한 다음 정책을 면밀히 검토하여 광범위한 액세스를 허용하지 않는지 확인하십시오.
  - 관련 AWS 주체에 연결된 ID 기반 정책 (예: IAM 역할)
  - 관련 리소스에 연결된 AWS 리소스 기반 정책 (예: KMS 키) AWS Key Management Service
- [AWS 대리인이 혼동할 가능성을 방지하기 위해 보안 주체에 대해서는 서비스 주체가 지원되지 않습니다.](#)
- 페더레이션 보안 주체는 지원되지 않습니다.
- 정식 사용자 ID는 지원되지 않습니다.
- 정책 크기는 20KB를 초과할 수 없습니다.

## 암호화된 데이터에 대한 액세스 공유

AWS 관리형 KMS 키로 데이터 스트림에 대해 서버 측 암호화를 활성화한 상태에서 리소스 정책을 통해 액세스를 공유하려는 경우 고객 관리 키 (CMK) 를 사용하도록 전환해야 합니다. 자세한 정보는 [Kinesis Data Streams용 서버 측 암호화란?](#)을 참조하세요. 또한 KMS 크로스 계정 공유 기능을 사용하여 공유 보안 주체 엔터티가 CMK에 액세스할 수 있도록 허용해야 합니다. 공유 보안 주체 엔터티에 대한 IAM 정책도 변경해야 합니다. 자세한 내용은 [다른 계정의 사용자가 KMS 키를 사용하도록 허용](#)을 참조하세요.

## AWS Lambda 함수를 다른 계정의 Kinesis Data Streams에서 읽도록 구성

다른 계정의 Kinesis Data Streams에서 읽도록 Lambda 함수를 구성하는 방법에 대한 예는 [교차 AWS Lambda 계정 함수를 통한 액세스 공유](#) 섹션을 참조하세요.

## 리소스 기반 정책을 사용하여 액세스 공유

### Note

기존 리소스 기반 정책을 업데이트하면 기존 정책을 교체하므로, 새 정책에 필요한 모든 정보를 포함해야 합니다.



## 교차 AWS Lambda 계정 함수를 통한 액세스 공유

### Lambda 연산자

1. [IAM 콘솔로](#) 이동하여 함수의 [Lambda](#) 실행 역할로 사용할 IAM 역할을 생성합니다. AWS Lambda 필수 Kinesis Data Streams 및 Lambda 호출 권한이 AWSLambdaKinesisExecutionRole 있는 관리형 IAM 정책을 추가합니다. 또한 이 정책은 액세스 권한이 있는 모든 잠재적 Kinesis Data Streams 리소스에 대한 액세스 권한을 부여합니다.
2. [AWS Lambda 콘솔에서 Kinesis Data Streams 데이터 스트림의 레코드를 처리하는 AWS Lambda 함수를 생성하고 실행 역할을 설정하는 동안 이전 단계에서 생성한 역할을 선택합니다.](#)
3. 리소스 정책 구성을 위해 Kinesis Data Streams 리소스 소유자에게 실행 역할을 제공합니다.
4. Lambda 함수 설정을 마칩니다.

### Kinesis Data Streams 리소스 소유자

1. Lambda 함수를 간접 호출할 크로스 계정 Lambda 실행 역할을 가져옵니다.
2. =Amazon Kinesis Data Streams 콘솔에서 데이터 스트림을 선택합니다. 데이터 스트림 공유 탭을 선택하고 공유 정책 생성 버튼을 선택하여 시각적 정책 편집기를 시작합니다. 데이터 스트림 내에서 등록된 소비자를 공유하려면 소비자를 선택하고 공유 정책 생성을 선택합니다. JSON 정책을 직접 작성할 수도 있습니다.
3. 크로스 계정 Lambda 실행 역할을 보안 주체로 지정하고 액세스를 공유하는 정확한 Kinesis Data Streams 작업을 지정합니다. `kinesis:DescribeStream` 작업도 포함해야 합니다. Kinesis Data Streams의 리소스 정책 예제에 대한 자세한 내용은 [Kinesis 데이터 스트림에 대한 예제 리소스 기반 정책](#) 섹션을 참조하세요.
4. 정책 생성을 선택하거나 [PutResourcePolicy](#)를 사용하여 정책을 리소스에 연결합니다.

### 크로스 계정 KCL 소비자와 액세스 공유

- KCL 1.x를 사용하는 경우 KCL 1.15.0 이상을 사용하는지 확인합니다.
- KCL 2.x를 사용하는 경우 KCL 2.5.3 이상을 사용하는지 확인합니다.

### KCL 연산자

1. KCL 애플리케이션을 실행할 IAM 사용자 또는 IAM 역할을 리소스 소유자에게 제공합니다.
2. 리소스 소유자에게 데이터 스트림 또는 소비자 ARN을 요청합니다.

3. 제공된 스트림 ARN을 KCL 구성의 일부로 지정해야 합니다.
  - KCL 1.x의 경우: [KinesisClientLibConfiguration](#) 생성자를 사용하고 스트림 ARN을 제공하십시오.
  - KCL 2.x의 경우: 스트림 ARN 또는 Kinesis 클라이언트 라이브러리에만 제공할 수 있습니다. [StreamTrackerConfigsBuilder](#)의 경우 StreamTracker, 라이브러리에서 생성한 DynamoDB 임대 테이블의 스트림 ARN 및 생성 에포크를 제공하십시오. Enhanced StreamTracker Fan-Out과 같은 등록된 공유 소비자의 글을 읽으려면 소비자 ARN을 사용하고 함께 제공하십시오.

## Kinesis Data Streams 리소스 소유자

1. KCL 애플리케이션을 실행할 크로스 계정 IAM 사용자 또는 IAM 역할을 가져옵니다.
2. =Amazon Kinesis Data Streams 콘솔에서 데이터 스트림을 선택합니다. 데이터 스트림 공유 탭을 선택하고 공유 정책 생성 버튼을 선택하여 시각적 정책 편집기를 시작합니다. 데이터 스트림 내에서 등록된 소비자를 공유하려면 소비자를 선택하고 공유 정책 생성을 선택합니다. JSON 정책을 직접 작성할 수도 있습니다.
3. 크로스 계정 KCL 애플리케이션의 IAM 사용자 또는 IAM 역할을 보안 주체로 지정하고 액세스를 공유하는 정확한 Kinesis Data Streams 작업을 지정합니다. Kinesis Data Streams의 리소스 정책 예제에 대한 자세한 내용은 [Kinesis 데이터 스트림에 대한 예제 리소스 기반 정책](#) 섹션을 참조하세요.
4. 정책 생성을 선택하거나 [PutResourcePolicy](#)를 사용하여 정책을 리소스에 연결하십시오.

## 암호화된 데이터에 대한 액세스 공유

AWS 관리형 KMS 키로 데이터 스트림에 대해 서버 측 암호화를 활성화하고 리소스 정책을 통해 액세스를 공유하려는 경우 고객 관리 키 (CMK) 사용으로 전환해야 합니다. 자세한 정보는 [Kinesis Data Streams용 서버 측 암호화란?](#)을 참조하세요. 또한 KMS 크로스 계정 공유 기능을 사용하여 공유 보안 주체 엔터티가 CMK에 액세스할 수 있도록 허용해야 합니다. 공유 보안 주체 엔터티에 대한 IAM 정책도 변경해야 합니다. 자세한 내용은 [다른 계정의 사용자가 KMS 키를 사용하도록 허용](#)을 참조하세요.

## Amazon Kinesis Data Streams의 규정 준수 검증

타사 감사자는 AWS 여러 규정 준수 프로그램의 일환으로 Amazon Kinesis Data Streams의 보안 및 규정 준수를 평가합니다. 여기에는 SOC, PCI, FedRAMP, HIPAA 등이 포함됩니다.

특정 규정 준수 프로그램 범위 내 AWS 서비스 목록은 규정 준수 [프로그램별 범위 내 AWS 서비스](#)를 참조하십시오. 일반 정보는 [AWS 규정 준수 프로그램](#)을 참조하십시오.

를 사용하여 타사 감사 보고서를 다운로드할 수 AWS Artifact 있습니다. 자세한 내용은 [AWS Artifact의 보고서 다운로드](#)를 참조하십시오.

Kinesis Data Streams 사용 시 규정 준수 책임은 데이터의 민감도, 회사의 규정 준수 목표 및 관련 법률 및 규정에 따라 결정됩니다. Kinesis Data Streams 사용이 HIPAA, PCI 또는 AWS FedRAMP와 같은 표준을 준수해야 하는 경우 다음과 같은 도움이 되는 리소스를 제공합니다.

- [보안 및 규정 준수 킷 스타트 가이드](#) — 이 배포 가이드에서는 아키텍처 고려 사항을 설명하고 보안 및 규정 준수에 중점을 둔 기본 환경을 배포하기 위한 단계를 제공합니다. AWS
- [HIPAA 보안 및 규정 준수를 위한 설계 백서](#) — 이 백서는 기업이 HIPAA 준수 애플리케이션을 개발하는 데 사용할 수 있는 방법을 설명합니다. AWS
- [AWS 규정 준수 리소스](#) — 업종과 지역에 적용할 수 있는 워크북 및 가이드 모음입니다.
- [AWS Config](#) — 리소스 구성이 내부 관행, 업계 지침 및 규정을 얼마나 잘 준수하는지 평가하는 AWS 서비스입니다.
- [AWS Security Hub](#) — 이 AWS 서비스는 보안 업계 표준 및 모범 사례를 준수하는지 확인하는 데 도움이 되는 보안 상태를 종합적으로 보여줍니다.

## Amazon Kinesis Data Streams의 복원력

AWS 글로벌 인프라는 지역 및 가용 AWS 영역을 중심으로 구축됩니다. AWS 지역은 물리적으로 분리되고 격리된 여러 가용 영역을 제공하며, 이러한 가용 영역은 지연 시간이 짧고 처리량이 높으며 중복성이 높은 네트워킹으로 연결됩니다. 가용 영역을 사용하면 중단 없이 가용 영역 간에 자동으로 장애 조치가 이루어지는 애플리케이션 및 데이터베이스를 설계하고 운영할 수 있습니다. 가용 영역은 기존의 단일 또는 복수 데이터 센터 인프라보다 가용성, 내결함성, 확장성이 뛰어납니다.

AWS [지역 및 가용 영역에 대한 자세한 내용은 글로벌 인프라를 참조하십시오](#) AWS .

Kinesis Data Streams는 AWS 글로벌 인프라 외에도 데이터 복원력 및 백업 요구 사항을 지원하는 데 도움이 되는 여러 기능을 제공합니다.

## Amazon Kinesis Data Streams의 재해 복구

Amazon Kinesis Data Streams 애플리케이션을 사용하여 스트림의 데이터를 처리할 때 다음 수준의 장애가 발생할 수 있습니다.

- 레코드 프로세서가 실패할 수 있습니다.
- 작업자 또는 작업자를 인스턴스화한 애플리케이션 인스턴스가 실패할 수 있습니다.
- 애플리케이션 인스턴스를 1개 이상 호스팅하는 EC2 인스턴스가 실패할 수 있습니다.

## 레코드 프로세서 장애

작업자는 Java 작업을 사용하여 레코드 프로세서 메서드를 호출합니다. [ExecutorService](#) 작업이 실패하면 작업자가 레코드 프로세서에서 처리하던 샤드를 제어하게 됩니다. 작업자는 새로운 레코드 프로세서 작업을 시작하여 해당 샤드를 처리합니다. 자세한 정보는 [읽기 조절](#)을 참조하세요.

## 작업자 또는 애플리케이션 장애

워커 또는 Amazon Kinesis Data Streams 애플리케이션의 인스턴스가 실패하면 상황을 감지하고 처리해야 합니다. 예를 들어, `Worker.run` 메서드에서 예외가 발생하면 이를 파악하고 처리해야 합니다.

애플리케이션 자체가 실패하면 이를 감지하고 애플리케이션을 다시 시작해야 합니다. 애플리케이션이 시작되면 새 작업자를 인스턴스화하고, 처리할 샤드가 자동으로 할당된 새 레코드 프로세서를 인스턴스화합니다. 장애가 발생하기 전에 이 레코드 프로세서가 처리한 것과 동일한 샤드이거나 이 프로세서에 처음 할당된 샤드일 수 있습니다.

작업자 또는 애플리케이션에 장애가 발생했지만 장애가 감지되지 않고, 다른 EC2 인스턴스에서 실행 중인 애플리케이션의 다른 인스턴스가 있는 경우 이러한 다른 인스턴스의 작업자가 장애를 처리합니다. 이러한 작업자는 실패한 작업자가 더 이상 처리하지 않는 샤드를 처리하기 위해 추가 레코드 프로세서를 만듭니다. 따라서 이 다른 EC2 인스턴스의 로드가 증가합니다.

여기서 설명하는 시나리오에서는 워커 또는 애플리케이션이 실패해도 호스팅 EC2 인스턴스가 계속 실행되어 오토 스케일링이 이 인스턴스를 다시 시작하지 않는다고 가정합니다.

## Amazon EC2 인스턴스 장애

오토 스케일링의 애플리케이션을 위해 EC2 인스턴스를 실행하는 것이 좋습니다. 그러면 EC2 인스턴스 중 하나가 실패할 경우 오토 스케일링이 자동으로 새 인스턴스를 시작하여 이를 대체합니다. 시작 시 Amazon Kinesis Data Streams 애플리케이션을 시작하도록 인스턴스를 구성해야 합니다.

## Kinesis Data Streams의 인프라 보안

관리형 서비스인 Amazon Kinesis Data Streams는 [Amazon Web Services: 보안 프로세스 개요 백서에 설명된 글로벌 네트워크 보안 절차에 따라](#) 보호됩니다. AWS

AWS 게시된 API 호출을 사용하여 네트워크를 통해 Kinesis Data Streams에 액세스할 수 있습니다. 클라이언트가 전송 계층 보안(TLS) 1.2 이상을 지원해야 합니다. 클라이언트는 Ephemeral Diffie-Hellman(DHE) 또는 Elliptic Curve Ephemeral Diffie-Hellman(ECDHE)과 같은 PFS(전달 완전 보안, Perfect Forward Secrecy)가 포함된 암호 제품군도 지원해야 합니다. Java 7 이상의 최신 시스템은 대부분 이러한 모드를 지원합니다.

또한 요청은 액세스 키 ID 및 IAM 주체와 관련된 비밀 액세스 키를 사용하여 서명해야 합니다. 또는 [AWS Security Token Service\(AWS STS\)](#)를 사용하여 임시 보안 인증을 생성하여 요청에 서명할 수 있습니다.

## Kinesis Data Streams의 보안 모범 사례

Amazon Kinesis Data Streams는 자체 보안 정책을 개발하고 구현할 때 고려해야 할 여러 보안 기능을 제공합니다. 다음 모범 사례는 일반적인 지침이며 완벽한 보안 솔루션을 나타내지는 않습니다. 이러한 모범 사례는 환경에 적절하지 않거나 충분하지 않을 수 있으므로 참고용으로만 사용해 주세요.

### 최소 권한 액세스 구현

권한을 부여할 때 누가 어떤 Kinesis Data Streams 리소스에 대해 어떤 권한을 갖는지 결정합니다. 해당 리소스에서 허용할 작업을 사용 설정합니다. 따라서 작업을 수행하는 데 필요한 권한만 부여해야 합니다. 최소 권한 액세스를 구현하는 것이 오류 또는 악의적인 의도로 인해 발생할 수 있는 보안 위협과 영향을 최소화할 수 있는 근본적인 방법입니다.

### IAM 역할 사용

Kinesis 데이터 스트림에 액세스하려면 생산자 및 클라이언트 애플리케이션에 유효한 보안 인증이 있어야 합니다. AWS 자격 증명을 클라이언트 애플리케이션이나 Amazon S3 버킷에 직접 저장해서는 안 됩니다. 이러한 보안 인증은 자동으로 교체되지 않으며 손상된 경우 비즈니스에 큰 영향을 줄 수 있는 장기 보안 인증입니다.

대신 IAM 역할을 사용하여 생산자 및 클라이언트 애플리케이션이 Kinesis 데이터 스트림에 액세스하기 위한 임시 보안 인증을 관리해야 합니다. 역할을 사용하면 장기 자격 증명(예: 사용자 이름과 암호 또는 액세스 키)을 사용하여 다른 리소스에 액세스할 필요가 없습니다.

자세한 설명은 IAM 사용자 가이드에서 다음 주제를 참조하십시오:

- [IAM 역할](#)
- [역할에 대한 일반적인 시나리오: 사용자, 애플리케이션 및 서비스](#)

### 종속 리소스에서 서버 측 암호화 구현

Kinesis Data Streams에서 저장 데이터와 전송 중 데이터를 암호화할 수 있습니다. 자세한 정보는 [Amazon Kinesis Data Streams의 데이터 보호](#)을 참조하세요.

## API 호출을 모니터링하는 CloudTrail 데 사용합니다.

Kinesis Data Streams는 Kinesis Data Streams에서 사용자, 역할 또는 AWS 서비스가 수행한 작업의 기록을 제공하는 서비스와 AWS CloudTrail 통합됩니다.

에서 수집한 CloudTrail 정보를 사용하여 Kinesis Data Streams에 대한 요청, 요청이 이루어진 IP 주소, 요청한 사람, 요청 시기 및 추가 세부 정보를 확인할 수 있습니다.

자세한 내용은 [the section called “AWS CloudTrail을 사용하여 Amazon Kinesis Data Streams API 호출 로깅”](#)을(를) 참조하세요.

## 문서 기록

다음 표에서는 Amazon Kinesis Data Streams 설명서의 주요 변경 사항에 대해 설명합니다.

변경 사항	설명	변경 날짜
계정 간 데이터 스트림 공유에 대한 지원이 추가되었습니다.	<a href="#">다른 계정과 데이터 스트림 공유</a> 을 추가했습니다.	2023년 11월 22일
온디맨드 및 프로비저닝된 데이터 스트림 용량 모드에 대한 지원이 추가되었습니다.	<a href="#">데이터 스트림 용량 모드 선택</a> 을 추가했습니다.	2021년 11월 29일
서버 측 암호화의 새로운 콘텐츠	<a href="#">Amazon Kinesis Data Streams의 데이터 보호</a> 을 추가했습니다.	2017년 7월 7일
향상된 CloudWatch 지표를 위한 새로운 콘텐츠.	<a href="#">Amazon Kinesis Data Streams 모니터링</a> 단원을 업데이트했습니다.	2016년 4월 19일
Kinesis 에이전트 향상을 위한 새로운 콘텐츠	<a href="#">Kinesis 에이전트를 사용하여 Amazon Kinesis Data Streams에 쓰기</a> 단원을 업데이트했습니다.	2016년 4월 11일
Kinesis 에이전트 사용을 위한 새로운 콘텐츠	<a href="#">Kinesis 에이전트를 사용하여 Amazon Kinesis Data Streams에 쓰기</a> 을 추가했습니다.	2015년 10월 2일
릴리스 0.10.0의 KPL 콘텐츠 업데이트	<a href="#">Amazon Kinesis Producer Library를 사용하여 생산자 개발</a> 을 추가했습니다.	2015년 7월 15일

변경 사항	설명	변경 날짜
구성 가능한 측정치의 KCL 측정치 주제 업데이트	<a href="#">Amazon을 통한 Kinesis 클라이언트 라이브러리 모니터링 CloudWatch</a> 를 추가했습니다.	2015년 7월 9일
내용 재구성	더욱 간결한 트리 보기와 보다 논리적인 그룹화를 위해 콘텐츠 주제를 대폭 재구성했습니다.	2015년 7월 01일
새로운 KPL 개발자 안내서 주제	<a href="#">Amazon Kinesis Producer Library를 사용하여 생산자 개발</a> 을 추가했습니다.	2015년 6월 02일
새로운 KCL 측정치 주제	<a href="#">Amazon을 통한 Kinesis 클라이언트 라이브러리 모니터링 CloudWatch</a> 를 추가했습니다.	2015년 5월 19일
KCL .NET 지원	<a href="#">.NET으로 Kinesis Client Library 소비자 개발</a> 을 추가했습니다.	2015년 5월 1일
KCL Node.js 지원	<a href="#">Node.js로 Kinesis Client Library 소비자 개발</a> 을 추가했습니다.	2015년 3월 26일
KCL Ruby 지원	KCL Ruby 라이브러리 링크를 추가했습니다.	2015년 1월 12일
새 API PutRecords	새 PutRecords API에 대한 정보가 <a href="#">여기</a> 에 추가되었습니다. <a href="#">the section called "PutRecords를 사용하여 여러 레코드 추가"</a> .	2014년 12월 15일
태그 지정 지원	<a href="#">Amazon Kinesis Data Streams의 스트림 태그 지정</a> 을 추가했습니다.	2014년 9월 11일
새 CloudWatch 지표	측정치 <code>GetRecords.IteratorAgeMilliseconds</code> 를 <a href="#">Amazon Kinesis Data Streams 측정기준 및 지표</a> 에 추가했습니다.	2014년 9월 3일
새로운 모니터링 장	<a href="#">Amazon Kinesis Data Streams 모니터링 및 아마존을 통한 아마존 Kinesis 데이터 스트림 서비스 모니터링 CloudWatch</a> 가 추가되었습니다.	2014년 7월 30일



변경 사항	설명	변경 날짜
기본 샤드 제한	<a href="#">할당량과 제한</a> 을 업데이트했습니다. 기본 샤드 제한이 5에서 10으로 높아졌습니다.	2014년 2월 25일
기본 샤드 제한	<a href="#">할당량과 제한</a> 을 업데이트했습니다. 기본 샤드 제한이 2에서 5로 높아졌습니다.	2014년 1월 28일
API 버전 업데이트	Kinesis Data Streams API의 버전 2013-12-02용 업데이트	2013년 12월 12일
최초 릴리스	Amazon Kinesis 개발자 안내서의 최초 릴리스.	2013년 11월 14일

# AWS 용어집

최신 AWS 용어는 AWS 용어집 참조서의 [AWS 용어집](#)을 참조하세요.

기계 번역으로 제공되는 번역입니다. 제공된 번역과 원본 영어의 내용이 상충하는 경우에는 영어 버전이 우선합니다.