Redis Streams를 이용한 Event-Driven 아키텍처

1 MSA와 Event-Driven 아키텍처

MSA(Microservice Architecture)는 무엇인가

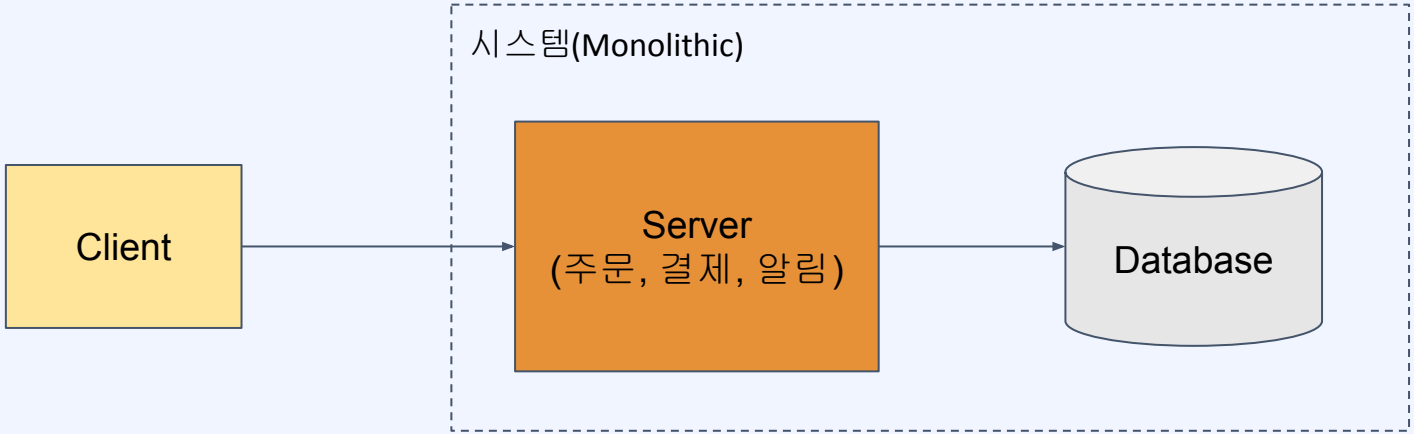
- 시스템을 독립적인 단위의 작은 서비스들로 분리(크기보다는 독립성이 중요)

  => 독립적인 단위: 다른 서비스와 다른 이유로 변경되고, 다른 속도로 변경되는 단위

- 각 서비스들이 사용하는 DB도 분리
- 각 서비스들은 API(인터페이스)를 통해서만 통신(다른 서비스의 DB 접근 불가)

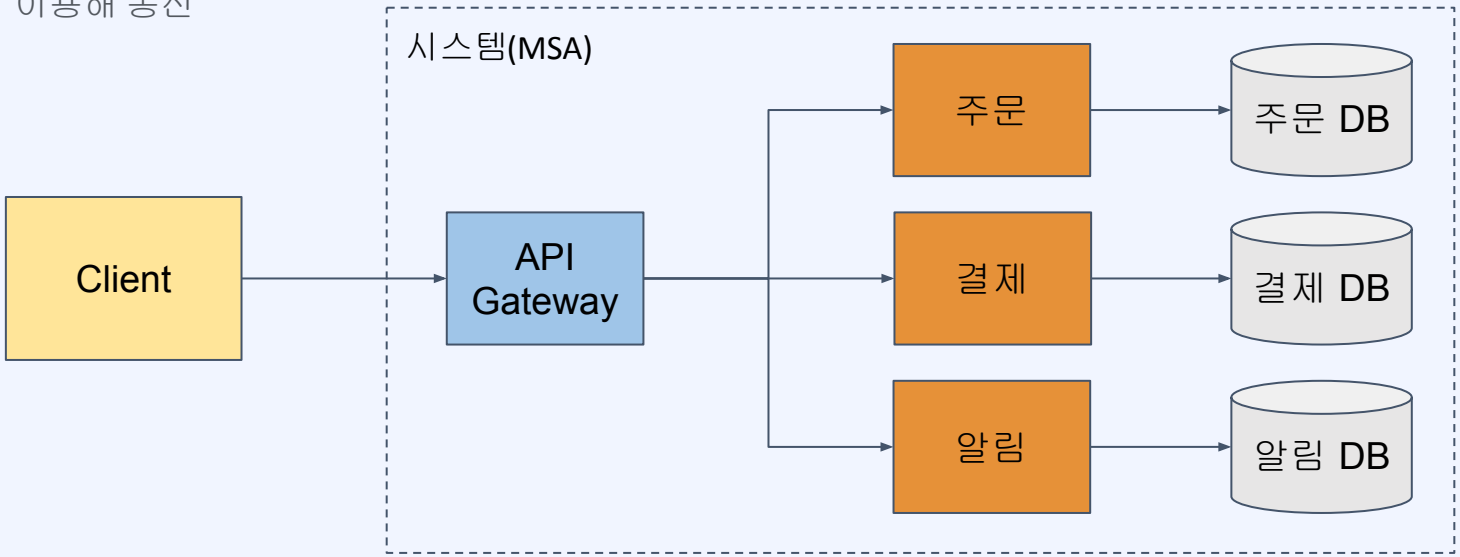
기존의 Monolithic 아키텍처

- 모든 기능들이 한 서버 안에 들어가 있고, 공유 데이터베이스를 사용



MSA 아키텍처

- 기능 별로(도메인 별로) 서버가 나뉘어 있고, 각자의 데이터베이스를 사용하며, API를 이용해 통신



MSA로 얻으려는 것

- 모듈성(높은 응집도, 낮은 결합도)
- 서비스 별로 독립적인 개발과 배포가 가능
- 서비스(코드) 크기가 작아져 이해가 쉽고 유지보수가 용이함
- 더 빠른 개발, 테스트, 배포
- 확장성(서비스 별로 개별 확장이 가능)
- 결함 격리(일부 서비스 실패가 전체 시스템 실패로 이어지지 않음)

MSA의 단점

- 분산 시스템의 단점을 그대로 가짐
- 통합 테스트의 어려움
- 모니터링과 디버깅의 복잡도 증가
- 트랜잭션 관리의 어려움
- 서비스간 통신 구조에 대한 고민이 필요

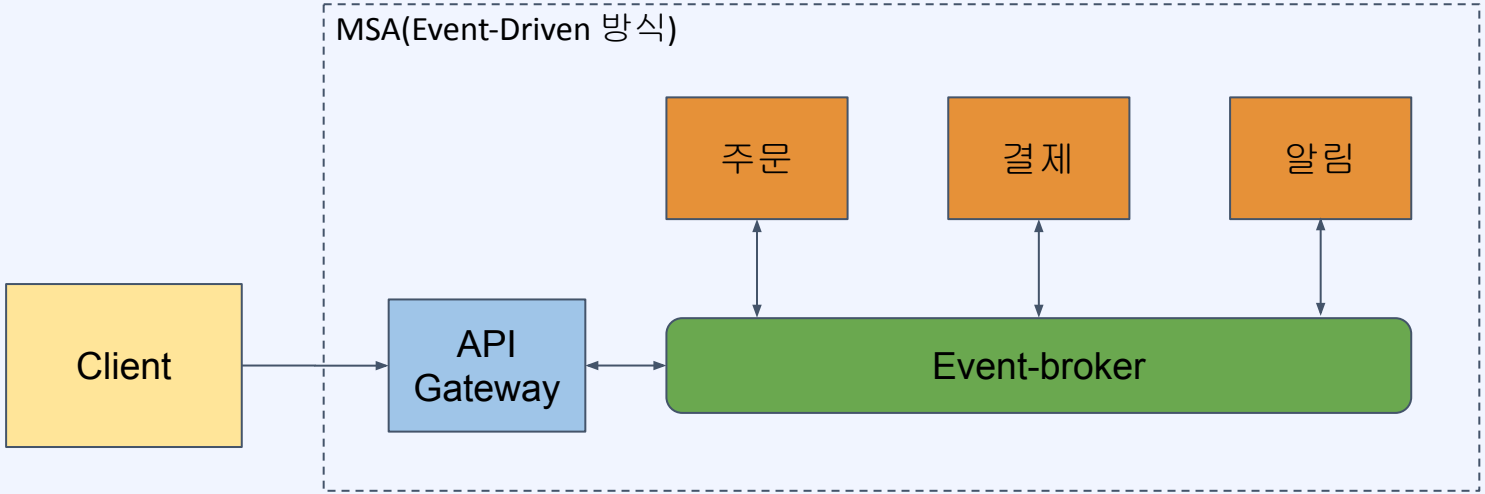
  => 동비 vs 비동기, 프로토콜, 통신 브로커 사용 등

Event-Driven 아키텍처란?

- 분산 시스템에서의 통신 방식을 정의한 아키텍처로, 이벤트의 생성/소비 구조로 통신이 이루어짐
- 각 서비스들은 이벤트 저장소인 Even-broker와의 의존성만 가짐

Event-Driven 아키텍처의 모습

- 각 서버들은 Event Broker에 이벤트를 생산/소비함으로써 통신



Event-Driven 아키텍처의 장점

- 이벤트 생산자/소비자 간의 결합도가 낮아짐(공통적인 Event-broker에 대한 결합만 있음)
- 생산자/소비자의 유연한 변경(서버 추가, 삭제 시에 다른 서버를 변경할 필요가 적어짐)
- 장애 탄력성(이벤트를 소비할 일부 서비스에 장애 발생해도 이벤트는 저장되고 이후에 처리됨)

Event-Driven 아키텍처의 단점

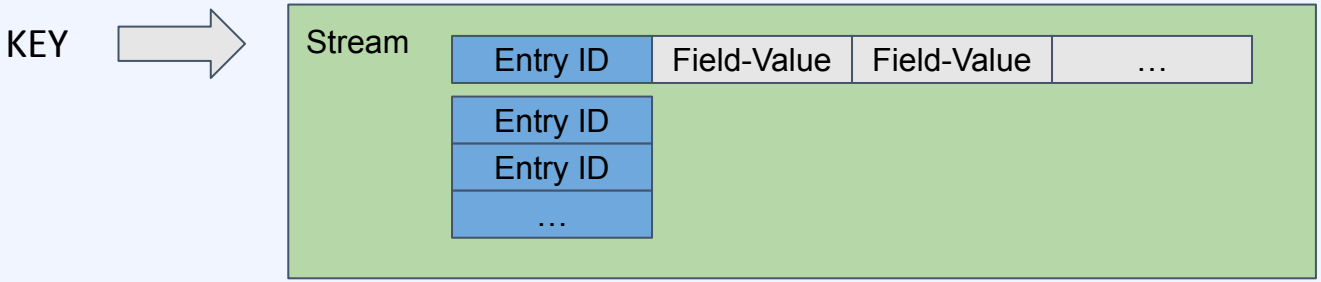
- 시스템의 예측가능성이 떨어짐(느슨하게 연결된 상호작용에서 기인함)
- 테스트의 어려움
- 장애 추적의 어려움

Redis Streams를 이용한 Event-Driven 아키텍처

2 Redis Streams의 이해

Redis Streams

- append-only log를 구현한 자료 구조
- 하나의 key로 식별되는 하나의 stream에 엔트리가 계속 추가되는 구조
- 하나의 엔트리는 entry ID + (key-value 리스트)로 구성
- 추가된 데이터는 사용자가 삭제하지 않는 한 지워지지 않음



Redis Streams의 활용

- 센서 모니터링(지속적으로 변하는 데이터인 시간 별 날씨 수집 등)
- 유저별 알림 데이터 저장
- 이벤트 저장소

Redis Streams의 명령어: 엔트리 추가

XADD: 특정 key의 stream에 엔트리를 추가한다.(해당 key에 stream이 없으면 생성)

```
XADD [key] [id] [field-value]
```

예제) user-notifications라는 stream에 1개의 엔트리를 추가하며 2개의 field-value 쌍을

```
127.0.0.1:6379> XADD user-notifications * user-a hi user-b hello
"1672002019152-0"
```

Redis Streams의 명령어: 엔트리 읽기(범위 기반)

XRANGE: 특정 ID 범위의 엔트리를 반환한다.

```
XRANGE [key] [start] [end]
```

예제) user-notifications의 모든 범위를 조회

```
127.0.0.1:6379> XRANGE user-notifications - +
1) 1) "1672002019152-0"
   2) 1) "user-a"
      2) "hi"
      3) "user-b"
      4) "hello"
2) 1) "1672002032075-0"
   2) 1) "user-c"
      2) "nice"
3) 1) "1672003596890-0"
   2) 1) "user-d"
      2) "good"
```

Redis Streams의 명령어: 엔트리 읽기(Offset 기반) - 1

XREAD: 한 개 이상의 key에 대해 특정 ID 이후의 엔트리를 반환한다. (동기 수행 가능)

```
XREAD BLOCK [milliseconds] STREAMS [key] [id]
```

예제) user-notifications의 0보다 큰 ID조회

```
127.0.0.1:6379> XREAD BLOCK 0 STREAMS user-notifications 0
1) 1) "user-notifications"
   2) 1) 1) "1672002019152-0"
         2) 1) "user-a"
            2) "hi"
            3) "user-b"
            4) "hello"
      2) 1) "1672002032075-0"
         2) 1) "user-c"
            2) "nice"
      3) 1) "1672003596890-0"
         2) 1) "user-d"
            2) "good"
```

Redis Streams의 명령어: 엔트리 읽기(Offset 기반) - 2

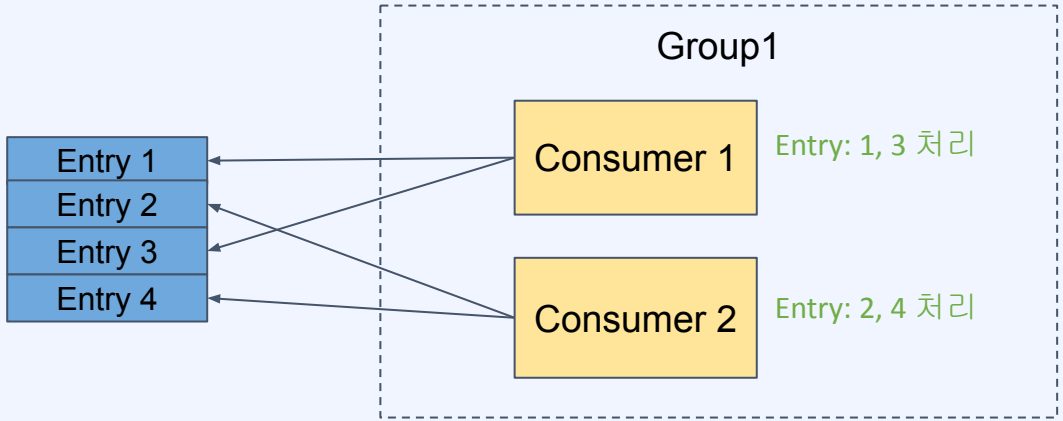
예제) user-notifications에서 새로 들어오는 엔트리를 동기 방식으로 조회

```
127.0.0.1:6379> XREAD BLOCK 0 STREAMS user-notifications $
```

=> 앞으로 들어올 데이터를 동기 방식으로 조회하여 event listener와 같은 방식으로 사용 가능.

Redis Streams의 명령어: Consumer Group

- 한 stream을 여러 consumer가 분산 처리할 수 있는 방식
- 하나의 그룹에 속한 consumer는 서로 다른 엔트리들을 조회하게 됨



Redis Streams의 명령어: Consumer Group - 1

XGROUP CREATE: consumer group을 생성

```
XGROUP CREATE [key] [group name] [id]
```

예제) user-notifications에 group1이라는 consumer group을 생성

```
127.0.0.1:6379> XGROUP CREATE user-notifications group1 $
OK
```

Redis Streams의 명령어: Consumer Group - 2

XREADGROUP: 특정 key의 stream을 조회하되, 특정 consumer group에 속한 consumer로 읽음

```
XREADGROUP GROUP [group name] [consumer name] COUNT [count] STREAMS [key] [id]
```

예제) user-notifications에서 group1 그룹으로 2개의 컨슈머가 각각 1개씩 조회

```
127.0.0.1:6379> XREADGROUP GROUP group1 consumer1 COUNT 1 STREAMS user-notifications >
1) 1) "user-notifications"
   2) 1) 1) "1672006144202-0"
         2) 1) "user-c"
            2) "new-message1"
127.0.0.1:6379> XREADGROUP GROUP group1 consumer2 COUNT 1 STREAMS user-notifications >
1) 1) "user-notifications"
   2) 1) 1) "1672006149548-0"
         2) 1) "user-d"
            2) "new-message1"
```

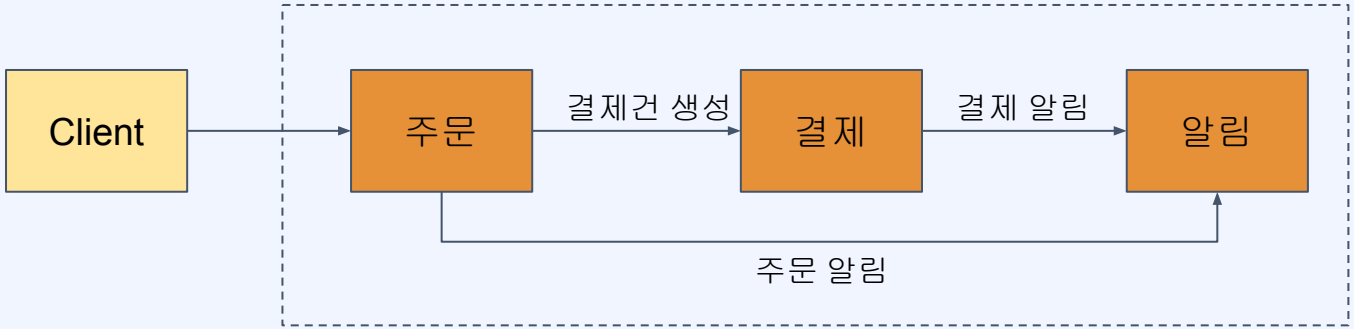
=> id에 “>”를 지정하면 아직 소비되지 않은 메시지를 가져오게 된다.

Redis Streams를 이용한 Event-Driven 아키텍처

3 Redis Streams를 이용한 이벤트 기반 통신 개발

HTTP를 이용한 동기 통신 방식

- 각 서비스는 필요한 서비스를 직접 호출



Event-broker를 이용한 메시지 기반 통신

- 각 서비스는 미리 정의된 이벤트를 소비/생성함으로써 통신

실습 예제 event 리스트

- order-events: 주문이 완료되면 발행
- payment-events: 결제가 완료되면 발행