Kafka

CQRS

- CQRS
  - ✓ CQRS란 Command and Query Responsibility Segregation 의 약자로 데이터 저장소로부터 읽기와 업데이트 작업을 분리하는 패턴
  - ✓ CQRS를 사용하면 애플리케이션의 퍼포먼스, 확장성, 보안성을 극대화할 수 있고 CQRS 패턴을 통해 만들어진 시스템의 유연성을 바탕으로 시간이 지나면서 지속적으로 시스템을 발전시켜나갈 수 있으며 여러 요청으로 부터 들어온 복수의 업데이트 명령들에 대한 충돌도 방지할 수 있음

CQRS

- 전통적인 방식의 문제점
  - ✓ 전통적인 아키텍처에서는 데이터베이스에서 데이터를 조회하고 업데이트하는데 같은 데이터 모델을 사용
  - ✓ 간단한 CRUD 작업에 대해서라면 이것은 문제없이 동작하지만 조금 더 복잡한 애플리케이션에서 이러한 접근 방식은 유지 보수를 어렵게 만들 수 있는데 애플리케이션은 데이터 조회 시 각기 다른 형태의 DTO를 반환하는 매우 다양한 쿼리들을 수행할 수 있는데 각각 다른 형태의 DTO들에 대해 객체 매핑을 하는 것은 복잡해질 수 있으며 데이터를 쓰거나 업데이트를 할 때는 복잡한 유효성 검사와 비즈니스 로직이 수행되어야 하는데 이 모든 걸 하나의 데이터 모델이 한다면 너무 많은 것을 수행하는 복잡한 모델이 되는 것
  - ✓ 읽기와 쓰기의 부하는 보통 같지 않기 때문에 각각에 대해 다른 성능이 요구됨
  - ✓ 읽기와 쓰기 작업에서 사용되는 데이터 표현들이 서로 일치하지 않는 경우가 많은데 그로 인해 일부 작업에서는 필요하지 않은 추가적인 컬럼이나 속성의 업데이트가 이루어 져야 함
  - ✓ 동일한 데이터 세트에 대해 병렬로 작업이 수행될 때 데이터 경합이 발생할 수 있음
  - ✓ 정보 조회를 위해 요구되는 복잡한 쿼리로 인해 성능에 부정적인 영향을 줄 수 있음
  - ✓ 하나의 데이터 모델이 읽기와 쓰기를 모두 수행하기 때문에 보안 관리가 복잡해질 수 있음

CQRS

- 해결책
  - ✓ CQRS는 읽기와 쓰기를 각각 다른 모델로 분리하는데 명령(Command)을 통해 데이터를 쓰고 쿼리(Query)를 통해 데이터를 읽음
  - ✓ 명령(Command)은 데이터 중심적이 아니라 수행할 작업 중심이 되어야 하는데 호텔룸의 상태를 예약됨으로 변경한다 가 아니라 호텔 룸 예약 과 같이 생성
  - ✓ 명령(Command)은 보통 동기적으로 처리되기보단 비동기적으로 큐에 쌓인 후 수행
  - ✓ 쿼리(Query)는 데이터베이스를 결코 수정하지 않는데 쿼리(Query)는 어떠한 도메인 로직도 캡슐화하지 않은 DTO 만을 반환
  - ✓ 읽기/쓰기 모델들은 서로 격리될 수 있는데 이렇게 읽기/쓰기 모델을 분리하는 것은 애플리케이션 디자인과 구현을 더욱 간단하게 만들어주지만 CQRS 코드는 ORM 툴을 통해 DB 스키마로부터 자동으로 생성되도록 할 수는 없다는 단점이 있음
  - ✓ 확실한 격리를 위해 물리적으로 읽기와 쓰기를 분리할 수 있는데 읽기 DB의 경우 복잡한 조인문이나 ORM 매핑을 방지하기 위해 materialized view를 가지는 조회에 최적화된 별도의 DB 스키마를 가질 수 있도록 만드는 것으로 단지 다른 DB 스키마가 아니라 아예 다른 타입의 데이터 저장소를 사용할 수도 있는데 쓰기는 RDBMS를 사용하고 읽기의 경우 몽고디비와 같은 NoSQL을 사용하는 것
  - ✓ 이와 같이 별도의 읽기/쓰기 데이터 저장소가 사용된다면 반드시 동기화가 이뤄져야 하는데 보통 이는 쓰기 모델이 DB에 수정사항이 발생할 때마다 이벤트를 발행함으로써 이뤄지며 이때 DB 업데이트와 이벤트 발행은 반드시 하나의 트랜잭션 안에서 이뤄져야 함

CQRS

- 해결책
  - ✓ 읽기 저장소는 단순히 쓰기 저장소의 레플리카 일 수도 있고 완전히 다른 구조를 가질 수도 있는데 어찌되었건 읽기와 쓰기를 분리하는 것은 각각의 부하에 알맞게 스케일링을 하는 것을 가능하게 해 주는데 보통 읽기 저장소가 쓰기보다 훨씬 더 많은 부하를 받게 됨
  - ✓ CQRS의 장점
    - 독립적인 스케일링: CQRS는 읽기와 쓰기 각각에 대해 독립적으로 스케일링을 하는 것을 가능하게 해주는데 이는 훨씬 더 적은 Lock 경합이 발생하는 것을 가능하게 함
    - 최적화된 데이터 스키마: 읽기 저장소는 쿼리에 최적화된 스키마를 사용할 수 있고 쓰기 저장소는 쓰기에 최적화된 스키마를 사용할 수 있음
    - 보안: 읽기와 쓰기를 분리함으로써 보안 관리가 용이
    - 관심사 분리: 읽기와 쓰기에 대한 관심사 분리는 시스템의 유지 보수를 더 쉽게 해 주고 유연하게 해주는데 대부분의 복잡한 비즈니스 로직은 쓰기 모델에 들어가고 읽기 모델은 상대적으로 간단한 조회 로직만 포함
    - 간단한 쿼리: 읽기 저장소의 materialized view를 통해 복잡한 조인문을 사용하지 않을 수 있음

CQRS

- 구현 이슈
  - ✓ 복잡성: CQRS의 기본 아이디어는 간단하지만 이 패턴은 이벤트 소싱 패턴을 포함할 경우 복잡해질 수 있음
  - ✓ 메세징: 메세징이 CQRS의 필수 요소는 아니지만 명령(Command)을 수행하고 업데이트 이벤트를 발행하는 것이 보편적인 사용법인데 이 경우 애플리케이션은 반드시 메세지 실패나 중복 메세지 와 같은 것들에 대한 처리를 해야 함
  - ✓ 데이터 일관성: 만약 읽기/쓰기 저장소가 분리된다면 읽기 데이터가 최신의 데이터가 아닐 수도 있는데 읽기 저장소에는 쓰기 저장소의 변경 사항들이 반영되어야 하는데 이에는 딜레이가 생기기 때문

CQRS

- CQRS를 사용해야 하는 경우
  - ✓ 많은 사용자가 동일한 데이터에 병렬로 액세스하는 도메인 - CQRS는 도메인 레벨에서의 병합 충돌을 최소화할 수 있도록 충분히 세분화된 명령(Command)를 정의하는 것을 가능하게 함
  - ✓ 복잡한 프로세스나 도메인 모델을 통해 가이드 되는 작업 기반 사용자 인터페이스 - 쓰기 모델은 비즈니스 로직, 유효성 검사 등을 모두 가진 완전한 명령(Command) 처리 기능을 가지며 쓰기 모델은 관련된 객체들의 집합을 하나의 단위(DDD에서의 aggregate)로 다룰 수 있고 이 객체들이 항상 일관된 상태를 가지도록 보장할 수 있지만 읽기 모델의 경우 비즈니스 로직이나 유효성 검사 같은 것 없이 오직 DTO만 반환하며 읽기 모델은 최종적으로 쓰기 모델과 일치(딜레이는 있을 수 있음)
  - ✓ 데이터 읽기의 성능이 데이터 쓰기의 성능과 별도로 조정이 가능해야 할 때 - 특히 읽기의 수가 쓰기의 수보다 훨씬 많을 때로 이 경우 읽기 모델은 스케일 아웃을 하고 쓰기 모델은 적은 수로 유지할 수 있으며 적은 수의 쓰기 모델은 충돌의 가능성을 최소화
  - ✓ 한 팀은 쓰기 모델에 대한 복잡한 도메인 모델에만 집중해야 하고 다른 한 팀은 사용자 인터페이스에 대한 읽기 모델에만 집중해야 할 때
  - ✓ 시스템이 시간이 지남에 따라 계속해서 진화하고 여러 버전을 가질 수 있으며 정기적으로 바뀔 수 있는 경우
  - ✓ 다른 시스템과의 통합 - 특히 이벤트 소싱 과 결합할 때 서브 시스템의 일시적 장애가 다른 시스템에 영향을 줘선 안됨

CQRS

- CQRS 사용이 권장되지 않는 경우
  - ✓ 도메인과 비즈니스 로직이 간단할 때
  - ✓ 단순한 CRUD일 때

CQRS

- 이벤트 소싱 과 CQRS 패턴
  - ✓ CQRS 패턴은 이벤트 소싱 패턴과 자주 같이 사용되는데 CQRS 기반 시스템은 분리된 읽기/쓰기 모델을 가지고 이들은 각자의 작업에 최적화되어 있으며 대부분 물리적으로도 다른 저장소에 위치
  - ✓ 이벤트 소싱 패턴과 CQRS를 같이 사용할 때는 이벤트 저장소가 쓰기 모델이 되며 이것이 메인 저장소가 되고 읽기 모델은 일반적으로 매우 역정규화된 materialized view를 제공하며 이 뷰들은 애플리케이션의 요구사항에 최적화되어 있으며 이로 인해 조회 성능을 극대화
  - ✓ 특정 시점의 실제 데이터를 사용하는 것 대신 쓰기 저장소로서 이벤트 스트림을 사용하는 것은 하나의 aggregate에 대한 병합 충돌을 방지해주고 성능과 확장성을 극대화 시켜줌
  - ✓ 이벤트들은 비동기적으로 읽기 저장소의 materialized view들을 생성하고 변경하는 데 사용
  - ✓ 이벤트 저장소가 공식 메인 저장소이기 때문에 시스템이 변경되거나 단순히 읽기 모델이 변경되었을 때 materialized view를 삭제하고 과거의 모든 이벤트를 다시 실행하여 새로운 형태로 생성하는 것도 가능한데 materialized view는 사실상 읽기 전용 캐시라고 보면 됨

CQRS

- 이벤트 소싱 과 CQRS 패턴
  - ✓ CQRS와 이벤트 소싱을 같이 사용할 때는 다음 사항들을 고려
    - 읽기/쓰기 저장소가 분리되어 있는 시스템에서는 이벤트가 실행되어 저장소가 수정되기 전 약간의 딜레이가 있을 수 있음
    - 이 패턴은 시스템의 복잡성을 증가시키는데 CQRS의 복잡성은 성공적인 구현을 더 어렵게 만들 수도 있지만 이벤트 소싱 방식을 사용하면 도메인 모델링을 더 쉽게 할 수도 있고 데이터 변경 이벤트들이 보존되기 때문에 뷰들을 리빌딩하는 것도 쉽게 할 수 있음
    - 읽기 모델에서 materialized view를 만들어내는 데는 오랜 실행 시간이나 리소스가 필요할 수 있는데 특히 오랜 기간 동안의 데이터 합계나 분석 자료와 같은 것이 필요할 땐 더더욱 그러하며 이럴 경우 일정 간격으로 데이터의 스냅샷을 만들어냄으로써 (예: 특정 액션의 총합, 한 엔티티의 현재 상태) 해결할 수 있음