MSA School

전자정부 표준 프레임워크 기반의

클라우드 네이티브 애플리케이션 구현

Part2. 마이크로서비스 패턴

UEngine

Tutors

장진영 유엔진솔루션즈 대표이사

MSA School 대표 강사
SAFe 공인 컨설턴트
(전) OMG / SEMAT 국제 표준 도구 분과 위원
現 디지털플랫폼정부 기술자문위원
유엔진BPMS / 오픈클라우드 엔진 파운더
클라우드 네이티브 (MSA, DDD) 강의
MSA와 SW모델링 Facebook 그룹 운영
(www.facebook.com/groups/cloudswmoding)

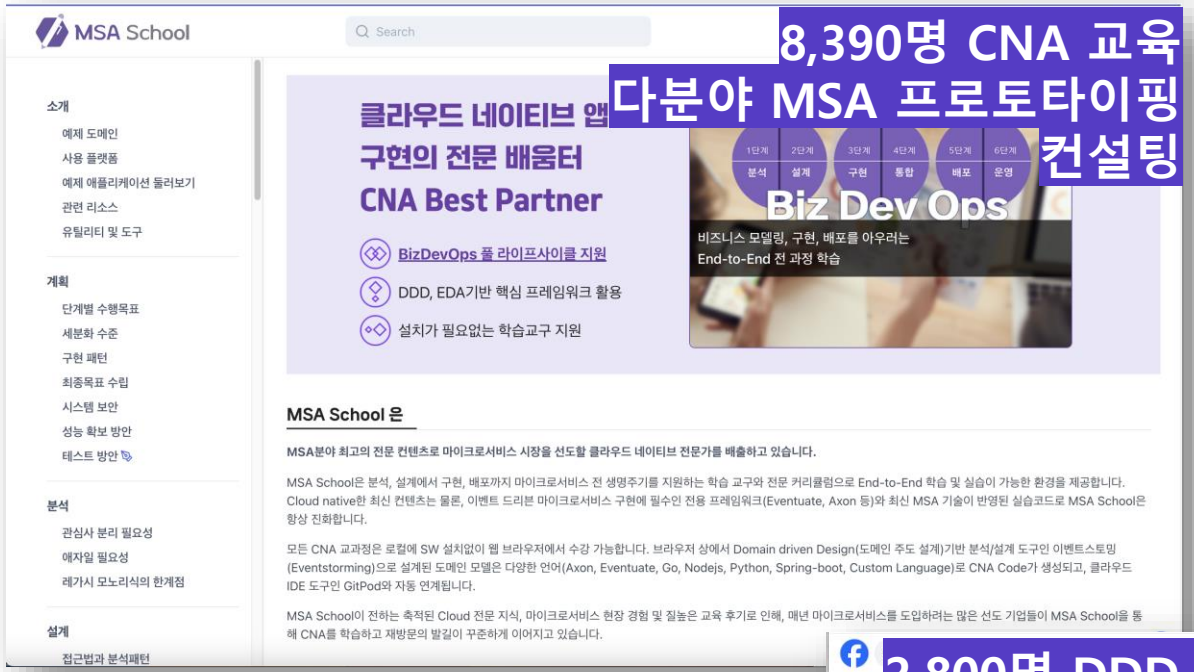
박용주 유엔진솔루션즈 이사

MSA School 대표 강사
• 現 MSA App. Engineering 기업과정 대표강사
• 現 세종사이버대학교 컴퓨터/AI 공학과 겸임교수
• 한국소프트웨어기술진흥협회 전문강사
• '21 : SK MSA App. Engineering 과정
• '21. 06 : KT Microservice 직무전환과정
• '20. 09 : Doosan Microservices 교육
• '19. 09 : KOSTA Microservices 교육
• '19. 02 : LG CNS 이벤트스토밍 교육

윤성열 드림플로우 대표이사

MSA School 대표 강사
• 現 드림플로우 대표이사
• 現 한국소프트웨어기술진흥협회 BAPF 포럼, 교육과정위원회 및 전문강사
• '18. 10 : 원오원글로벌 디지털팀 팀장
• '18. 04 : TTA 사물인터넷 특별기술위원회 사물인터넷 융합서비스 프로젝트그룹 (SPG11) 부의장
• '17. 03 : 가천대학교 겸임교수

About Us



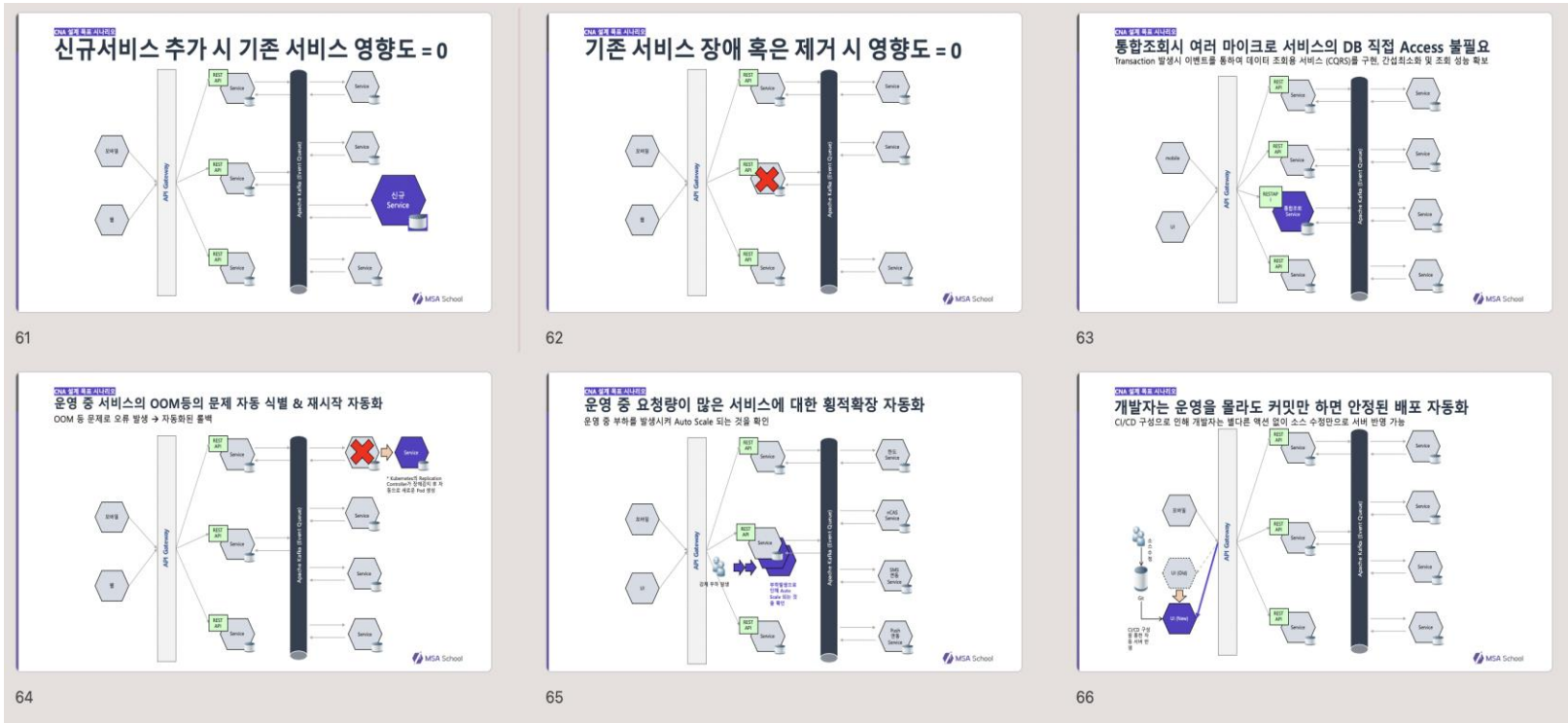
8,390명 CNA 교육
다분야 MSA 프로토타이핑
컨설팅

MSA 분석/구현 전문 도구

2,800명 DDD 페북그룹 운영

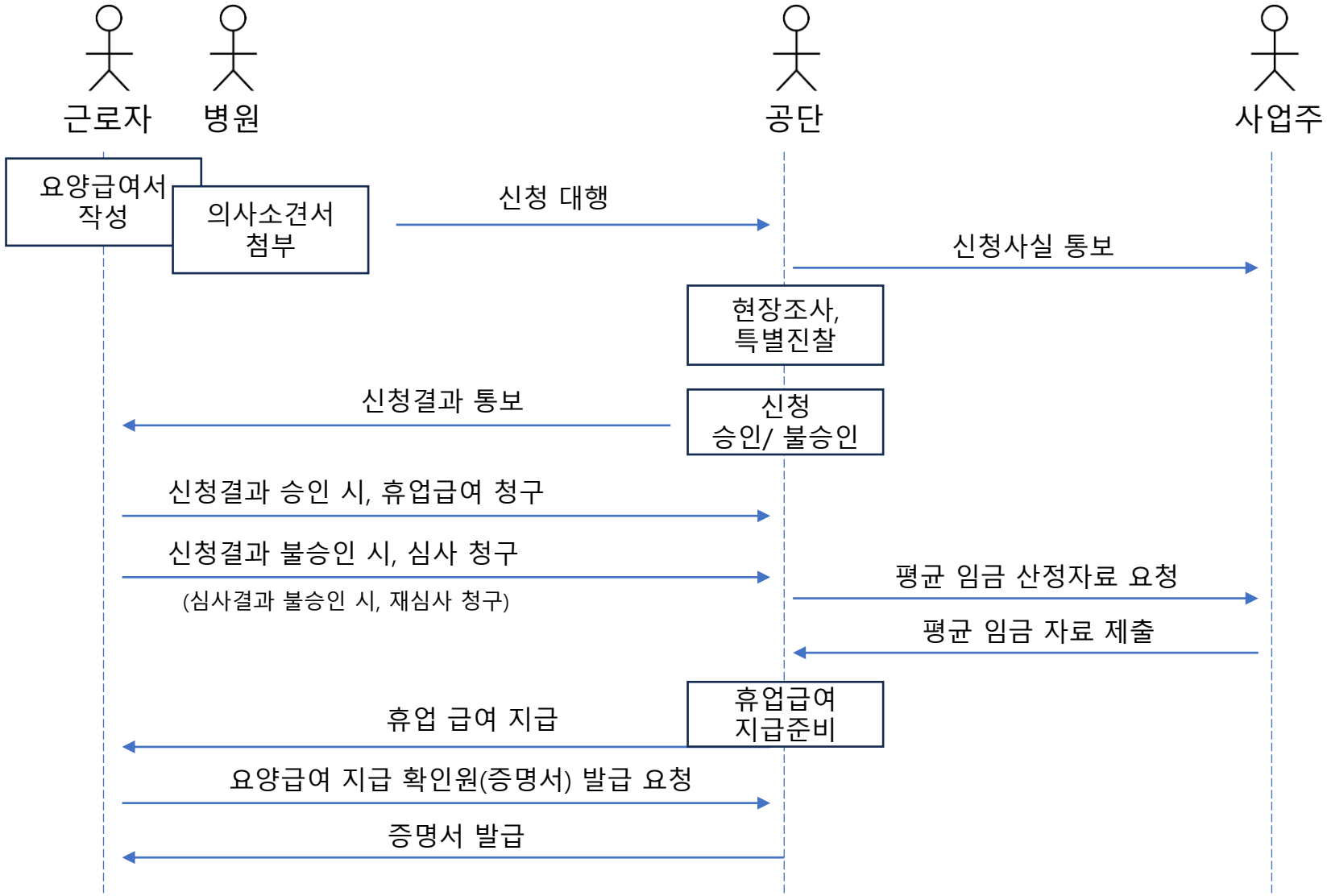
오늘의 주제 – MSA 의 구현

- CNA 설계 목표 시나리오
- 마이크로 서비스 구현 패턴
- RPC와 서킷 브레이커 패턴
- EDA (오케스트레이션과 코레오그라피)와 카프카
- 데이터 프로젝션: GraphQL / CQRS
- 마이크로 프론트엔드
- MSA 테스팅과 목업 생성
  (컨트랙트 테스트 / 단위테스트)
- 보안처리
- 컨테이너 패키징



등을 MSA Easy 를 통하여 세부적으로 설계하고 구현하는 과정을 살펴봅니다

대상 도메인 : 근로복지공단 '산재신청'



‘산재신청’ 요구사항 도출

기능적 요구사항

- 근로자가 요양급여 신청서를 작성하여 제출한다.
- 병원은 의사의 소견서와 요양 병원 직인을 신청서에 첨부하고, 공단에 대행 신청한다.
- 사업주는 근로복지공단을 통해 산재 신청 사실을 통보 받는다.
- 근로복지공단은 질병의 인과관계 확인을 위해 현장조사와 특별 진찰을 수행한다.
- 인과관계가 확인되면, 근로복지공단은 접수된 신청을 승인한다.
- 승인을 통보 받은 근로자는 휴업 급여를 청구한다.
- 불승인된 경우, 근로자는 심사청구를 요청할 수 있다.
- 근로복지공단은 해당 사업장에 근로자의 평균 임금을 요청한다.
- 근로복지공단은 심사결과와 보상 금액을 근로자에게 통보한다.
- 심사결과 금액에 불복하는 근로자는 재심사를 청구할 수 있다.
- 근로자가 요양급여 지급 확인원(증명서) 발급을 요청한다.

비기능적 요구사항

장애격리

- 요양급여 처리 서비스가 장애 시 라도 요양급여 신청 접수는 문제가 없어야 한다 → Event-driven, Eventual Consistency
- 일부 시스템이 과중 되면 적절히 확장이 이루어져야 한다 → Horizontal Auto Scale

프로세스 준수

- 각 마이크로 서비스를 넘나드는 프로세스는 일관성이 있게 준수되어야 한다. (예: 업무상 재해가 명확한 경우 7일 이내 요양 승인 여부가 결정된다.)

성능

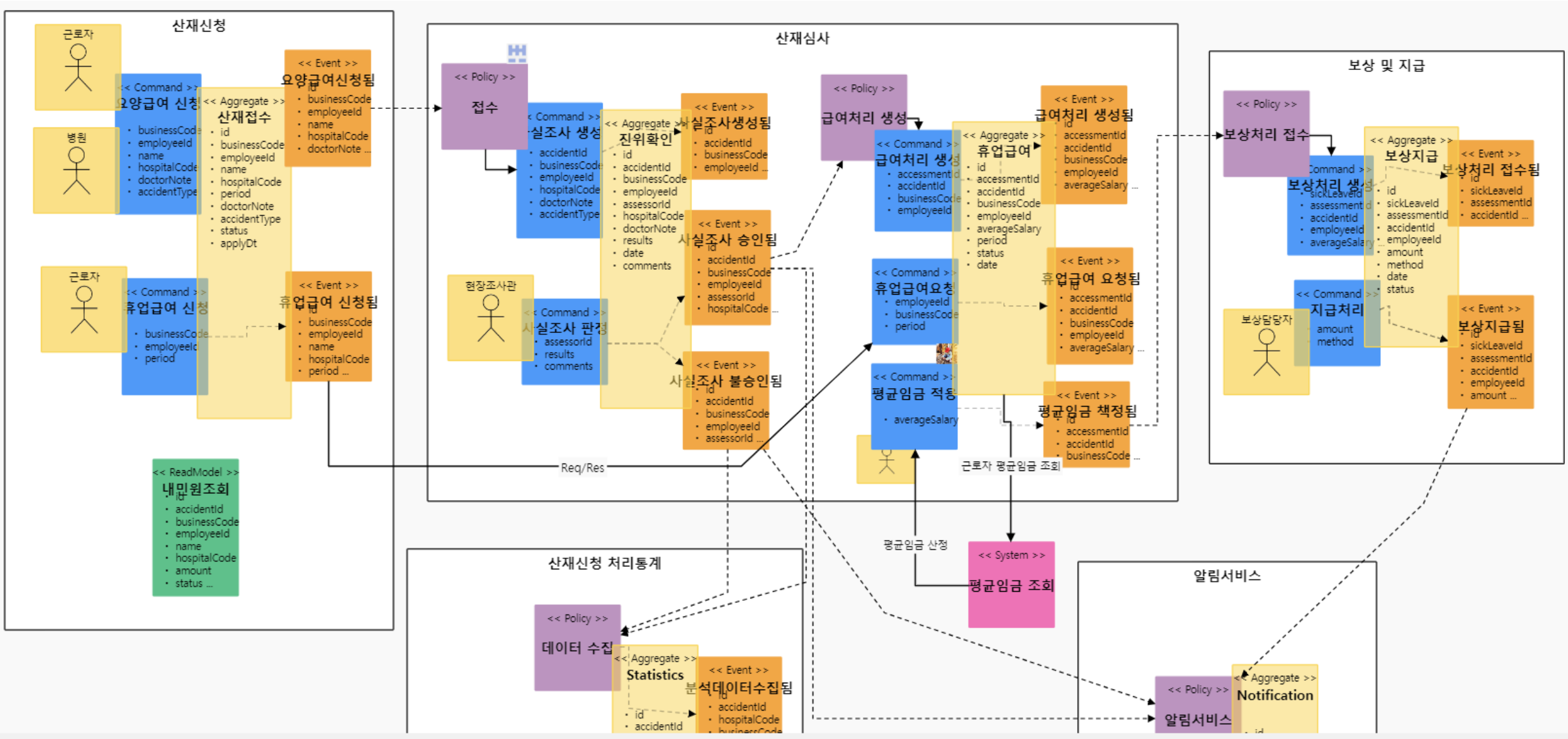
- 근로자 보수총액 신고 기간(매년 3월 말까지) 및 근로자가 자주 민원 진행 상태를 조회하더라도 성능저하가 없어야 한다. → Auto Scale Out, CQRS

도메인 제약사항

- 산재 신청은 4일 이상 요양이 필요한 질병에 대해 산재신청이 가능하다.
- 휴업 급여는 평균 임금의 70%로 산정한다.

‘산재신청’ 앱 설계 – 멀티도메인 경계 식별

‘산재신청’ 이벤트스토밍 모델



마이크로서비스 구현 패턴

구현 패턴	특 징	비고
서비스 단일 진입을 위한 API 게이트웨이 패턴	• 단일 진입점을 만들어 다른 유형의 클라이언트에게 서로 다른 API조합을 제공할 수도 있고 각 서비스 접근 시 필요한 인증/인가 기능을 한 번에 처리 가능	• Spring Cloud GW • Kubernetes Ingress
장애전파 차단을 위한 서킷 브레이커 패턴	• 하나의 서비스 장애로 다른 서비스가 영향을 받을 수 있는데, 장애가 발생된 서비스를 격리하여 유연하게 처리할 수 있는 방법으로 다양한 구현 단계에서 적용 가능	• Netflix OSS • Service Mesh
SAGA 패턴 (Eventual consistency, Compensation Trx)	• 각 서비스의 로컬 트랜잭션을 순차적으로 처리하는 패턴. 여러 개의 분산된 서비스들을 하나의 트랜잭션을 묶지 않고, 각 로컬 트랜잭션과 보상 트랜잭션을 이용해 비지니스 데이터 정합성 일치	• 오케스트레이션 Saga • 코레오그래피 Saga
CQRS 패턴 (명령 조회 책임 분리, 읽기모델과 쓰기모델 분리)	• 하나의 저장소에 쓰기 모델과 읽기 모델을 분리하는 방식으로 변화시켜 쓰기 서비스와 조회서비스를 분리해 저장소 처리 성능을 향상	• Materialized View
쓰기 최적화 이벤트 소싱(Event Sourcing) 패턴	• 메시지 브로커와 데이터 저장소를 분리하지 않고 하나로 사용해 상태가 아닌 트랜잭션 차체를 저장 하는 전략으로 상태 변경 이벤트를 바로 이벤트 저장소에 저장	• Replay, Snapshot
이벤트 드리븐 마이크로서비스(Event Driven Microservice)	• 발신자가 이벤트를 생성, 발행 (publish)하고 필요로 하는 수신자가 이를 구독((Subscribe)하고, 이벤트를 처리하는 형태의 커뮤니케이션 아키텍처	• EDA, Pub/Sub
서비스 매시(Service Mesh) 패턴	• 통신 네트워크 기능을 비즈니스 로직 기능과 분리하여 네트워크 인프라 계층에서 수행하게 해, 마이크로서비스 개발자가 비즈니스 로직 구현에 포커싱 가능	• Istio, Consul, Linkerd
마이크로 프론트엔드 UI 패턴	• 프론트 엔드가 여러 개의 조각들로 구성되고 각 조각은 여러 개의 마이크로 프론트 엔드의 조합으로 서비스를 제공	• Monolith Frontend • Micro Fronted
마이크로서비스 테스트 패턴 (Unit Test, API Mocking, CDC Test)	• Given-When-Then 테스트 패턴을 활용한 단위 마이크로서비스 테스트, 가상의 실체화된 서비스를 활용한 병렬 개발, 고객 주도 컨트랙 기반 API 정합성 테스트 전략	• Microcks (CNCF) • Spring Cloud Contract
마이크로서비스 인증/인가 패턴	• 인증/인가 처리를 위한 별도의 전담 서비스(또는 SSO 서버)를 두고, API 게이트웨이와 연동해 인증 인가를 처리하고 발급된 토큰을 다운 스트림으로 포워딩	• JWT Token, • Keycloak (CNCF)
중앙화된 로그 집계, 모니터링 및 추적 패턴	• 마이크로서비스 장애 실시간 감지와 서비스 간의 호출 인지 및 실시간 로그 집계를 위한 스택과 각가 서비스 트랜잭션의 추적을 통한 분산 트레이싱 구현	• 모니터링, 로그리게이션 • Zipkin 기반 트레이싱

트랜잭션 관점에서 모델 재조명

- 마이크로서비스 환경에서 트랜잭션이 여러 서비스에 걸쳐 있기 때문에 일관성을 지키는 것은 쉽지 않다.
  그래서, 여러 서비스에서 트랜잭션을 구현할 수 있는 패턴 필요..
- 이전에는 분산 트랜잭션을 관리하기 위해 2PC(two-phase commit) 프로토콜을 이용
- 모든 참가자가 커밋하거나 롤백 할지에 대해 관여했기 때문에 트랜잭션 관리가 가능

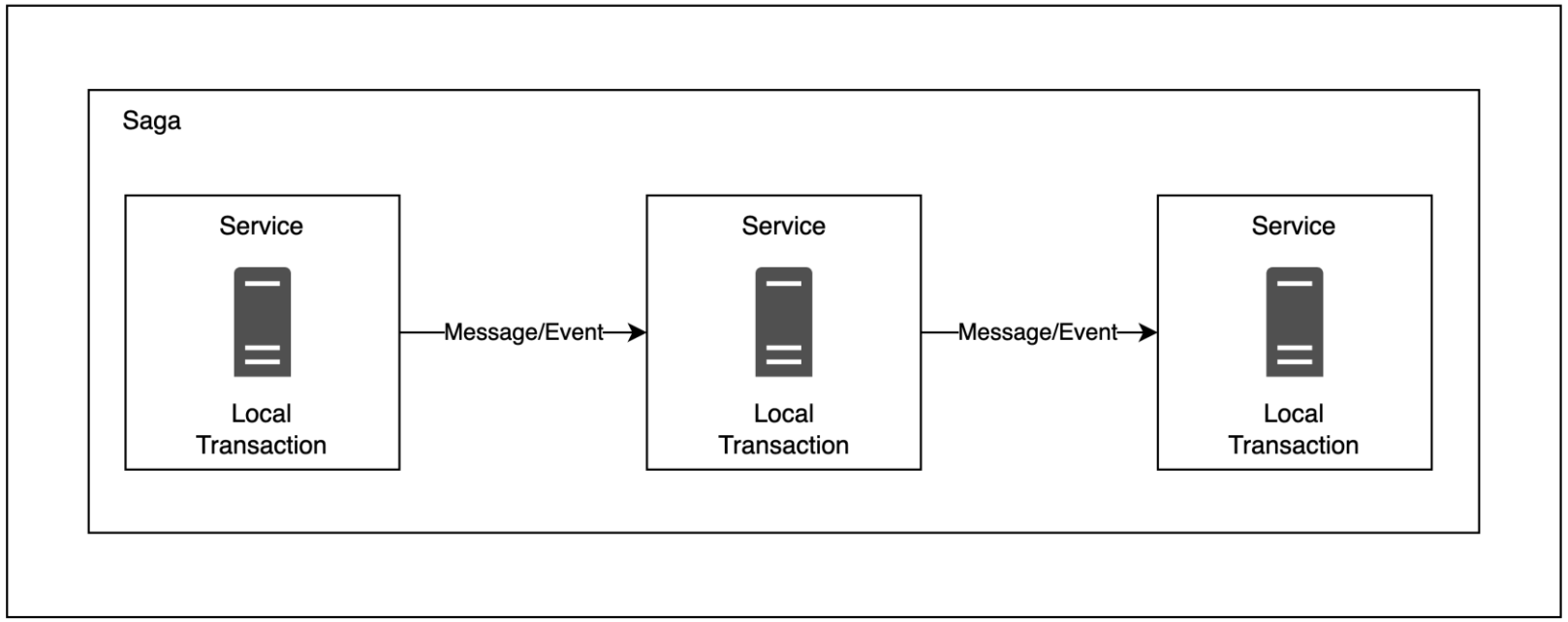
하지만,

- 2PC의 전제는 동일 DB여야 하고, DBMS가 분산 트랜잭션을 지원해야 한다. (ex. NoSQL은 2PC 지원하지 않음)

트랜잭션 관점에서 모델 재조명

그래서, 등장한 패턴이 Saga 패턴

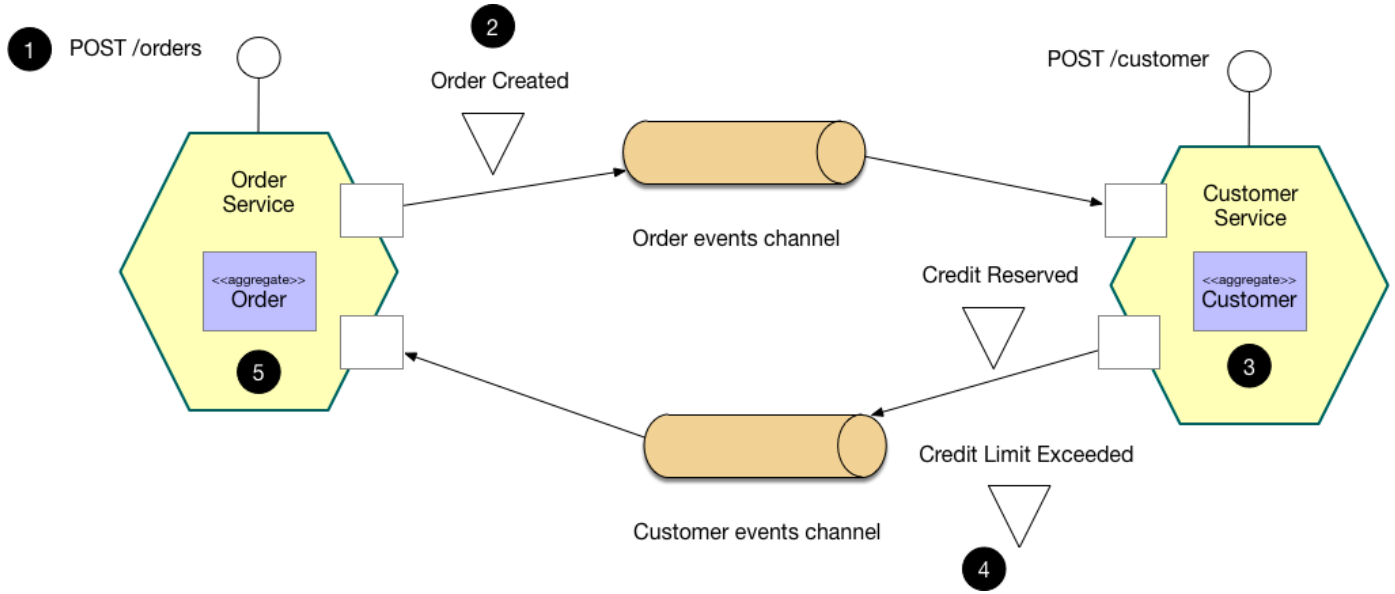
- Saga Pattern 은 분산 트랜잭션 환경에서 메시지 또는 이벤트를 주고 받으며 서비스 간의 데이터 일관성을 지키는 패턴
- 로컬 트랜잭션을 사용하며, 트랜잭션이 실패되면 변경된 내용을 취소하는 보상 트랜잭션을 발행해 일관성 유지



출처: https://learn.microsoft.com/ko-kr/azure/architecture/reference-architectures/saga/saga

SAGA Pattern - 코레오그래피

- 코레오그래피(Choreography) Saga : 중앙 제어없이 서비스끼리 이벤트로 통신하는 패턴



장점

- 참여자가 적고 중앙제어가 필요 없는 경우 적합
- 추가 서비스 생성/소멸에 간섭이 없어 구성이 간편
- 역할 분산으로 단일 실패 시점이 존재하지 않음

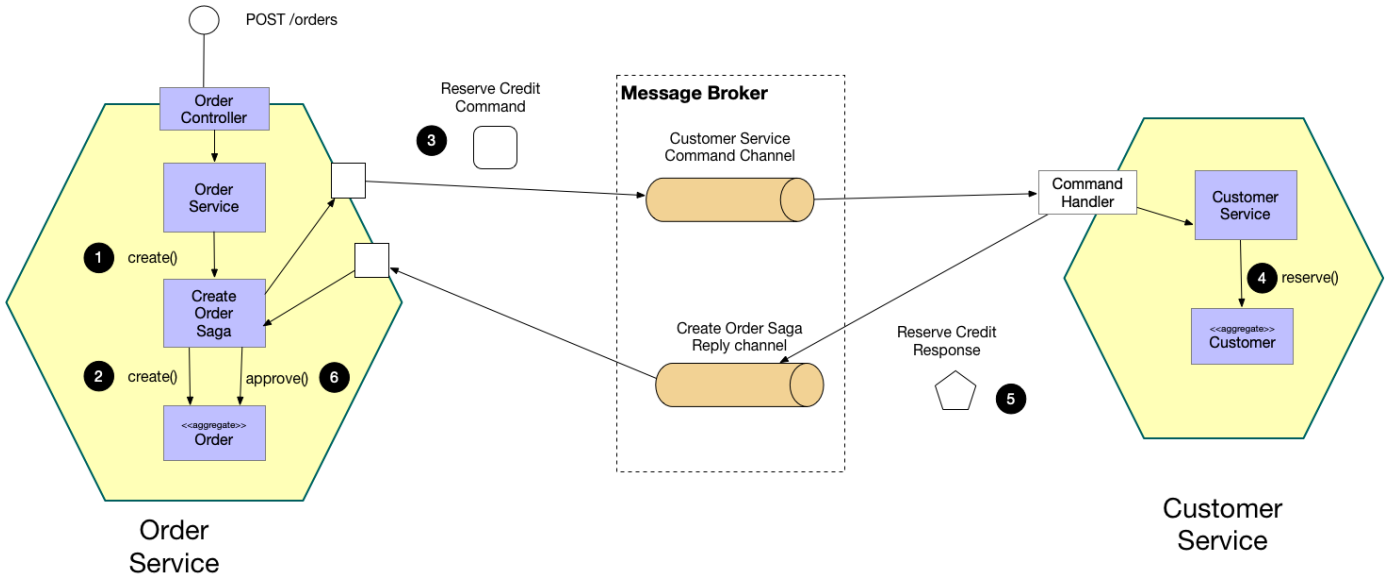
단점

- 명령 추적이 어렵기 때문에 워크플로우 파악이 힘듦
- Saga 참가자간 순환 종속성 발생 가능성 내재
- 통합 테스트가 어려움

출처: https://microservices.io/patterns/data/saga.html

SAGA Pattern - 오케스트레이션

- 오케스트레이션(Orchestration) Saga : 중앙의 컨트롤러가 전체 흐름과 보상 작업을 제어하는 패턴



장점

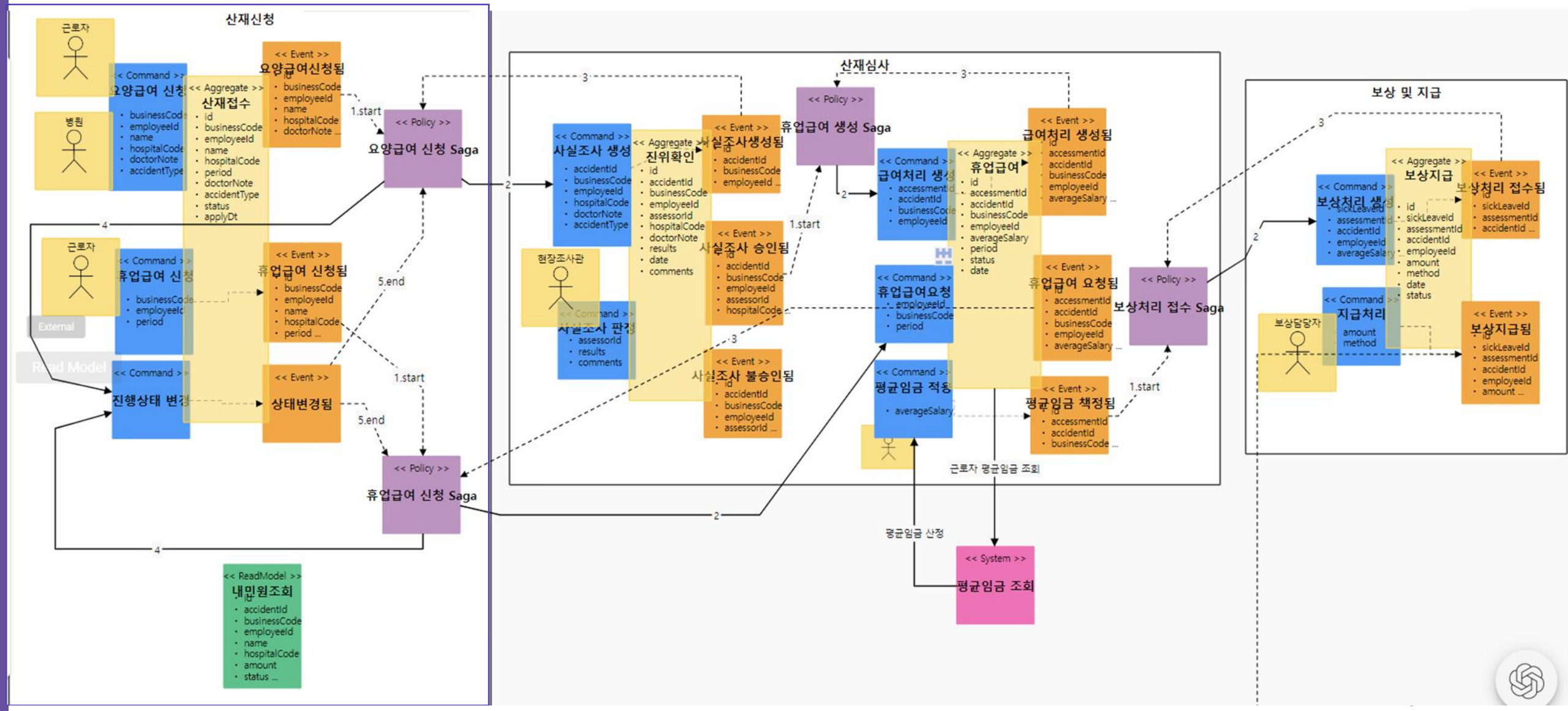
- 참여자가 많고 복잡한 워크 플로우에 적합
- 프로세스 흐름의 제어 가능
- 오케스트레이터 존재로 순환 종속성 발생 우려가 없음
- 참여자는 다른 참여자의 명령을 몰라도 됨

단점

- 중앙 통제를 위한 복잡한 로직 구현 필요
- 모든 워크플로우를 관리하므로 단일 실패지점이 될 우려

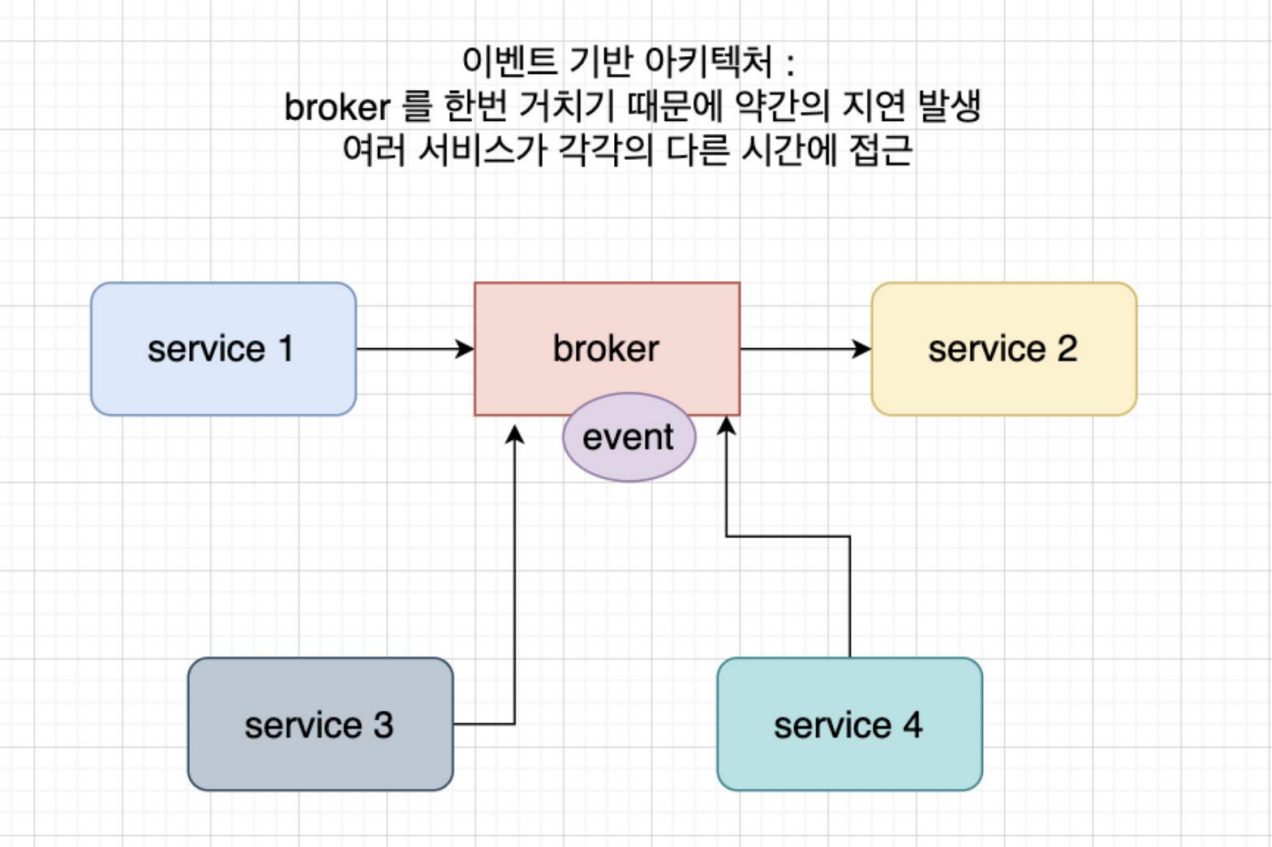
출처: https://microservices.io/patterns/data/saga.html

‘산재신청’ 이벤트스토밍 모델 (오케스트레이션 Saga)



EDA & Event Store(Kafka) 패턴

- 이벤트 드리븐 아키텍처는 독립적으로 실행되는 마이크로서비스들의 Loosely Coupling과 확장성을 위한 최고의 전략



구성

- Producer, Broker, Consumer
- Producer는 이벤트를 발행 - 브로드캐스팅
- Broker는 이벤트 저장소 (SSOT)
- (관심있는) Consumer는 이벤트를 구독

활용

- 성능보다 확장성이 우선시 될 때
- 데이터 복제나, 병렬로 여러 서비스 실행이 필요 할 때

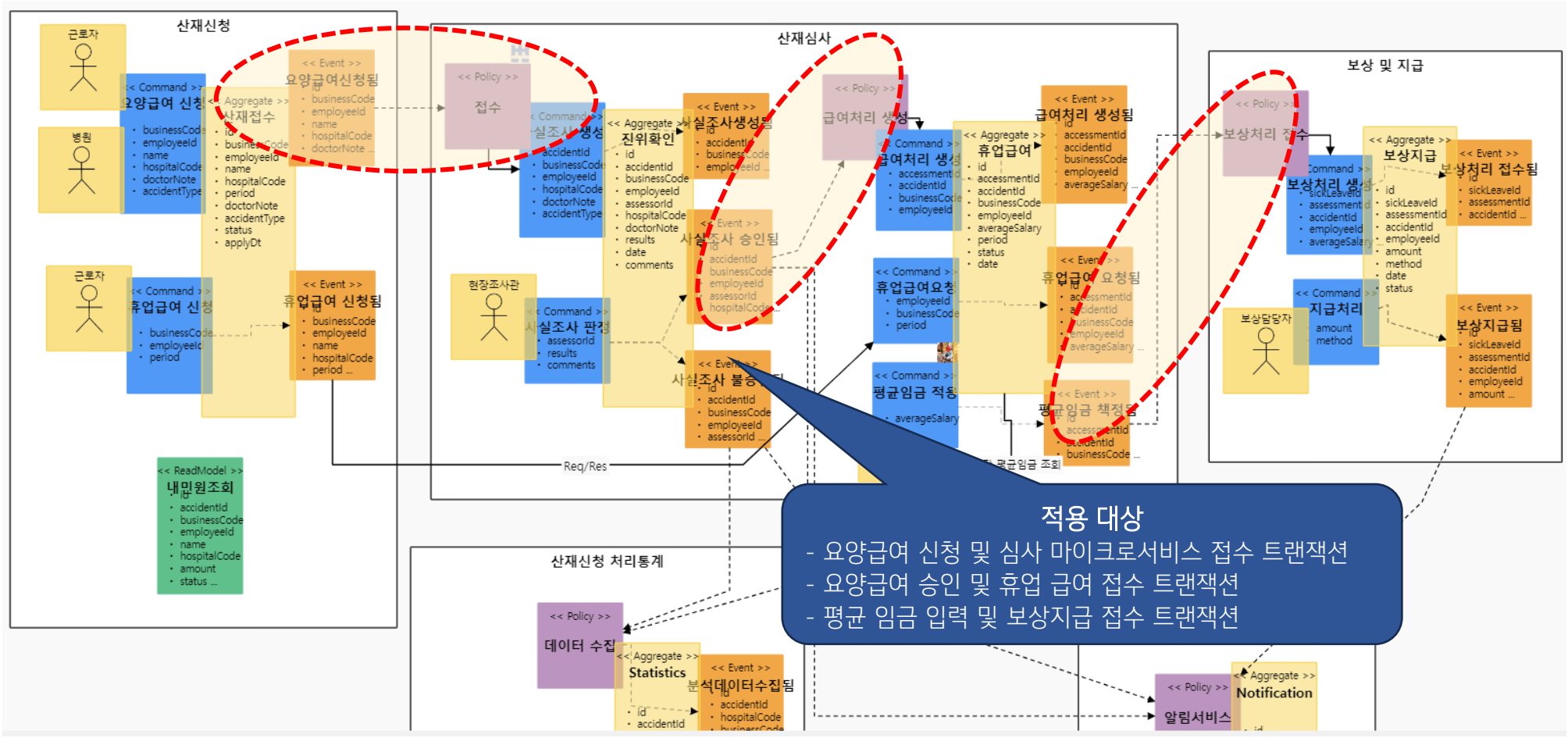
단점

- Pub/Sub 시, 약간의 지연 발생
- 이벤트 발생부터 리액티브 서비스들의 추적이 어려움
- 싱글 메시지 토픽으로 해소

출처: https://velog.io/@borab/분산-환경과-Event-Driven-Architecture

Lab Time

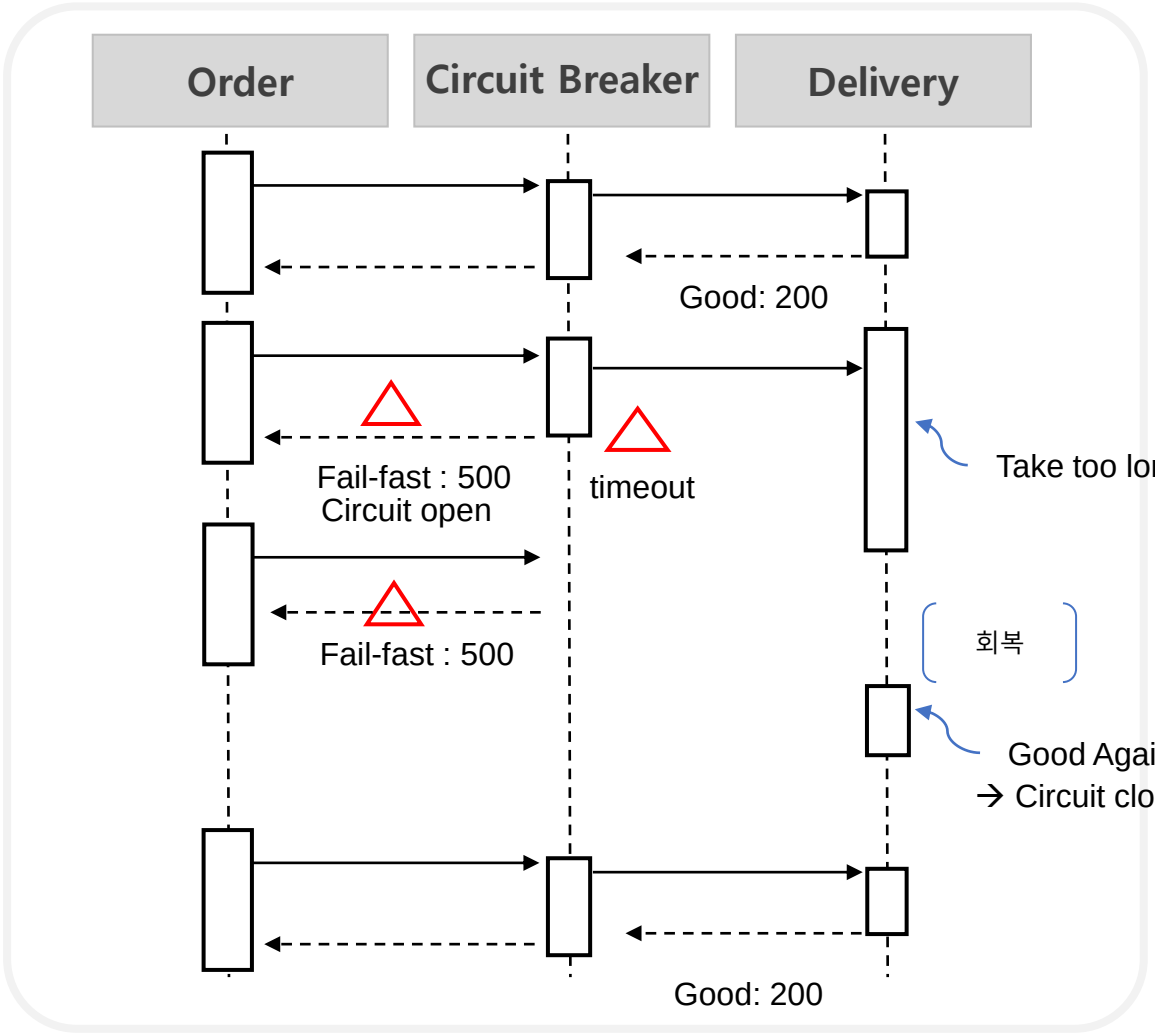
EDA & Event Store(Kafka) 패턴



MSA School

RPC와 서킷 브레이커 패턴

- 호출된 마이크로서비스 트랜잭션이 가능한 블로킹이 발생하지 않도록 미리 차단 (Fail-Fast) 시키는 전략



장애 감지 시, 차단기 작동(Open)

↓

일시 동안 Fallback으로 서비스 대체

↓

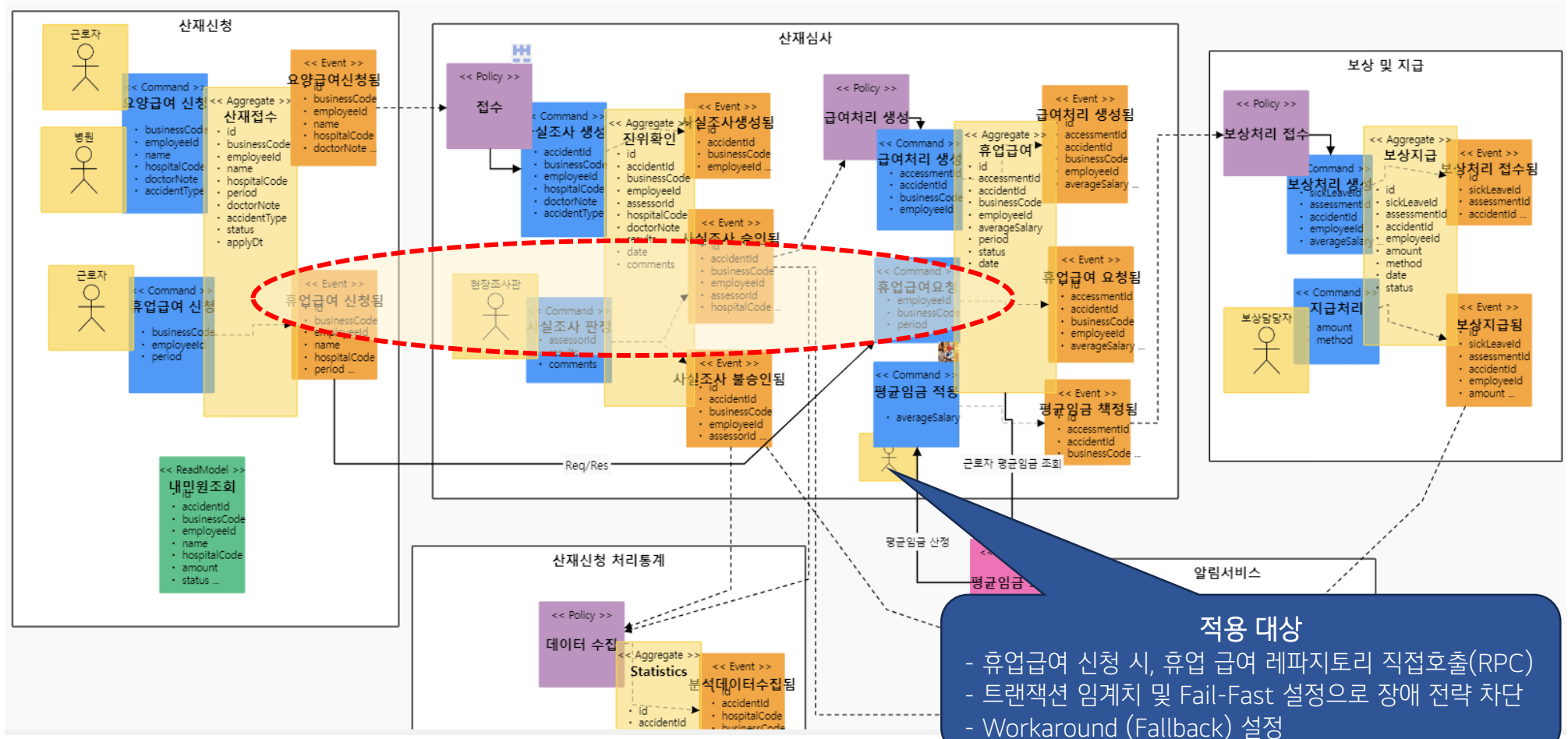
일부 트래픽으로 서비스 정상여부 확인

↓

정상 확인 시, 차단기 해제(Close)

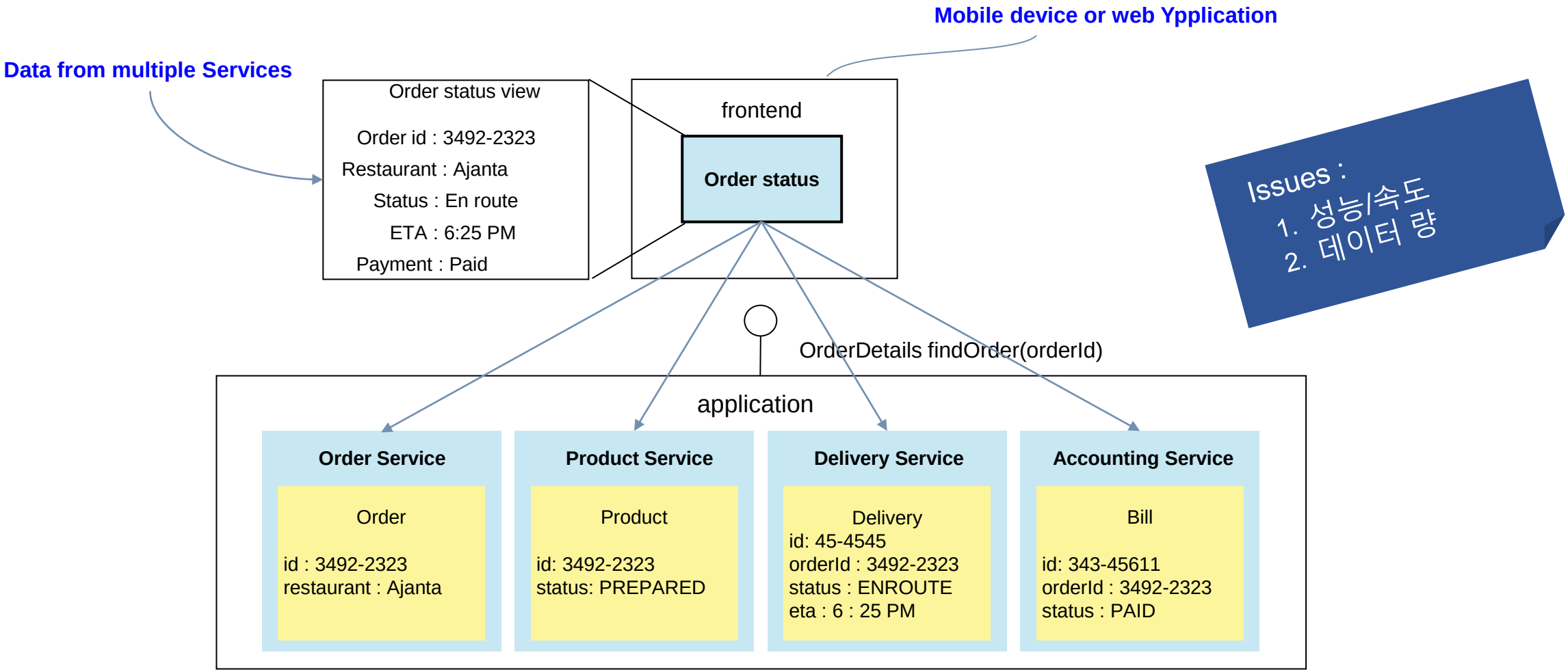
Lab Time

RPC와 서킷 브레이커 패턴



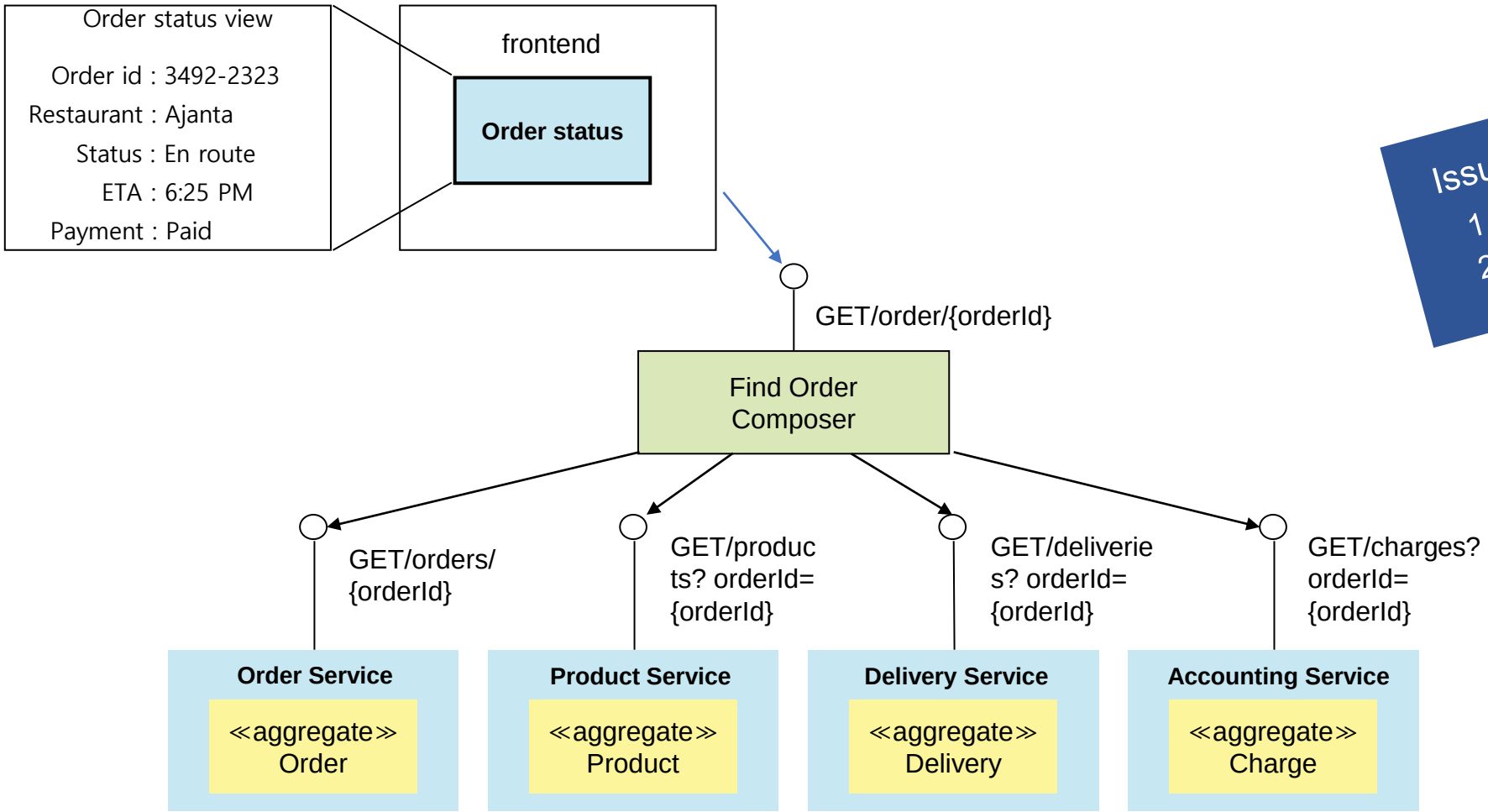
데이터 프로젝션 패턴

- 분산 환경에서의 Data Projection Issue



데이터 프로젝션 패턴

- 분산 환경에서의 Data Project Issue – Composite Service, 또는 Graph QL 패턴으로 해결



Issues :
1. 성능/속도
2. 장애 전파

데이터 프로젝션 패턴

- 분산 환경에서의 Data Project Issue – Composite Service, 또는 Graph QL 패턴으로 해결

GraphQL

- ✓ Facebook(Meta)에서 만든 데이터 질의어, "또 하나의 API 스펙 (https://graphql.org/)"
- ✓ 클라이언트가 데이터를 서버로부터 효율적으로 가져오는 것이 목적으로 클라이언트가 GQL 작성해 호출
- ✓ Rest API의 over fetching과 under fetching의 한계 극복
  - ✓ 열람하고자 하는 리소스의 일부 정보만 fetch (주문자 이름만 보고자 할 경우)
  - ✓ 주문에 따른 상품/배송 정보 등 교차정보를 한꺼번에 Fetch 가능

REST API	상황에 맞는 HTTP Method 사용 (GET/POST/PUT/DELETE)	Api별로 각각의 end point POST /api/orders DELETE /api/orders/1
GraphQL	POST 만 사용	단일 end point 만 사용 POST /graphql

- ✓ 스키마를 정의하고, 스키마 쿼리를 실행하는 리졸버(Resolver)를 구현해야 하며, API호출에 따라 백엔드에서 이 리졸버가 실행
- ✓ Rest API의 weakness를 보완하며, 구현 시 Rest API와 Hybrid하게 사용

데이터 프로젝션 패턴

- 분산 환경에서의 Data Project Issue –GraphQL, BFF 패턴 (Backend For Frontend)

- GQL Sample
  - 주문 조회

```
query getOrders {
  orders {
    productId
    productName
    quantity
    price
  }
}
```

주문 중 상품이름만 조회

```
query getOrders {
  orders {
    productName
  }
}
```

1번 주문 조회

```
query getOrderById {
  order(orderId: 1) {
    productId
    productName
    quantity
    price
  }
}
```

복합 서비스 조회

```
query Com
  orders {
    quantity
    customer
    state
    product {
      price
      name
    }
    delivery {
      deliveryAddress
    }
  }
}
```

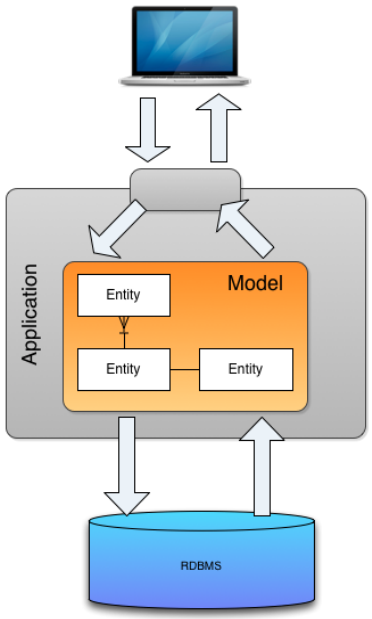
Issues :
1. 성능/속도
2. 장애 전파

단점
- 주문, 배송, 상품 서비스가 모두 가동중 이어야 데이터 조회가 됨
- RESTful API에 비해 캐싱이나, 파일 업로드를 위한 명세가 없어 별도 구현해야 함

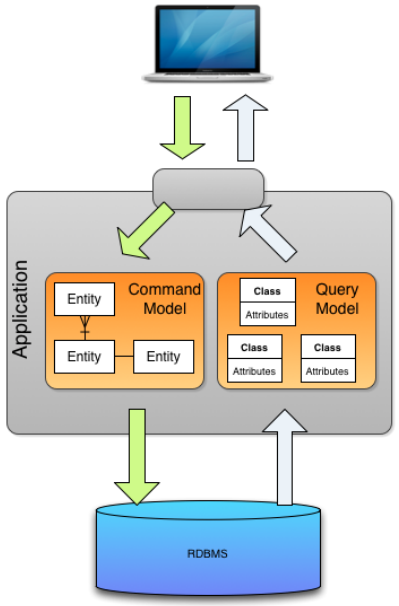
데이터 프로젝션 패턴

- 비즈니스 로직을 처리하는 데이터 변경영역(C,U,D)과 데이터 조회(R) 영역을 분리해 복잡도 해소, 성능이슈 해결
- 파편화 되어 있는 MSA 분산 환경에서 대쉬보드(ex. 마이페이지) 등 통합 데이터를 구축하는 전략

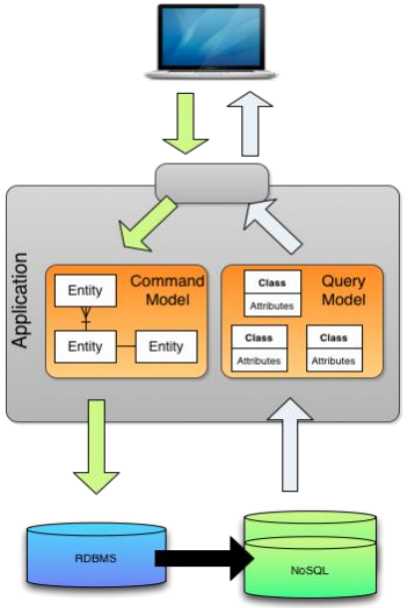
CQRS (명령을 처리하는 책임과 조회를 처리하는 책임을 분리)



▪ 전통적인 CRUD



▪ Simple CQRS architecture

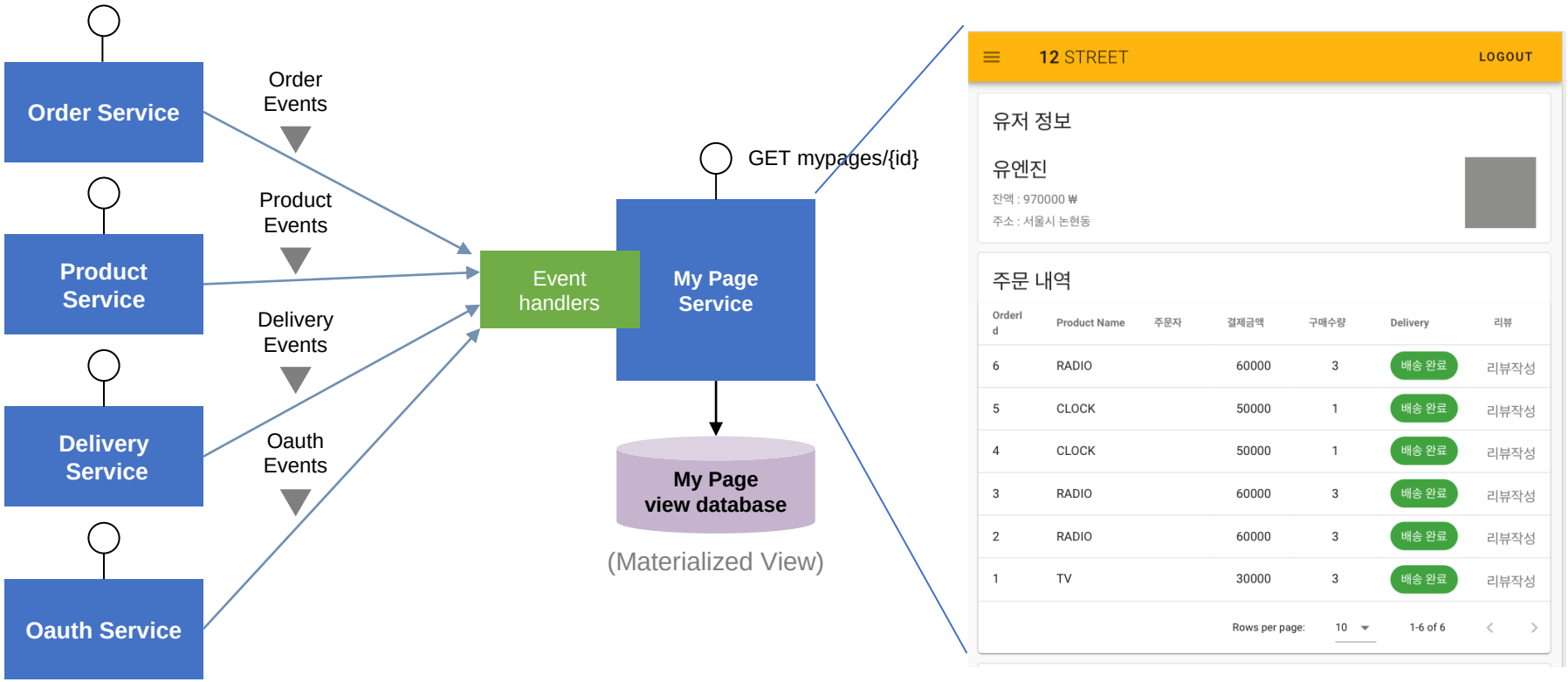


▪ CQRS with separated persistence

출처: https://azderica.github.io/02-architecture-msa/

데이터 프로젝션 패턴

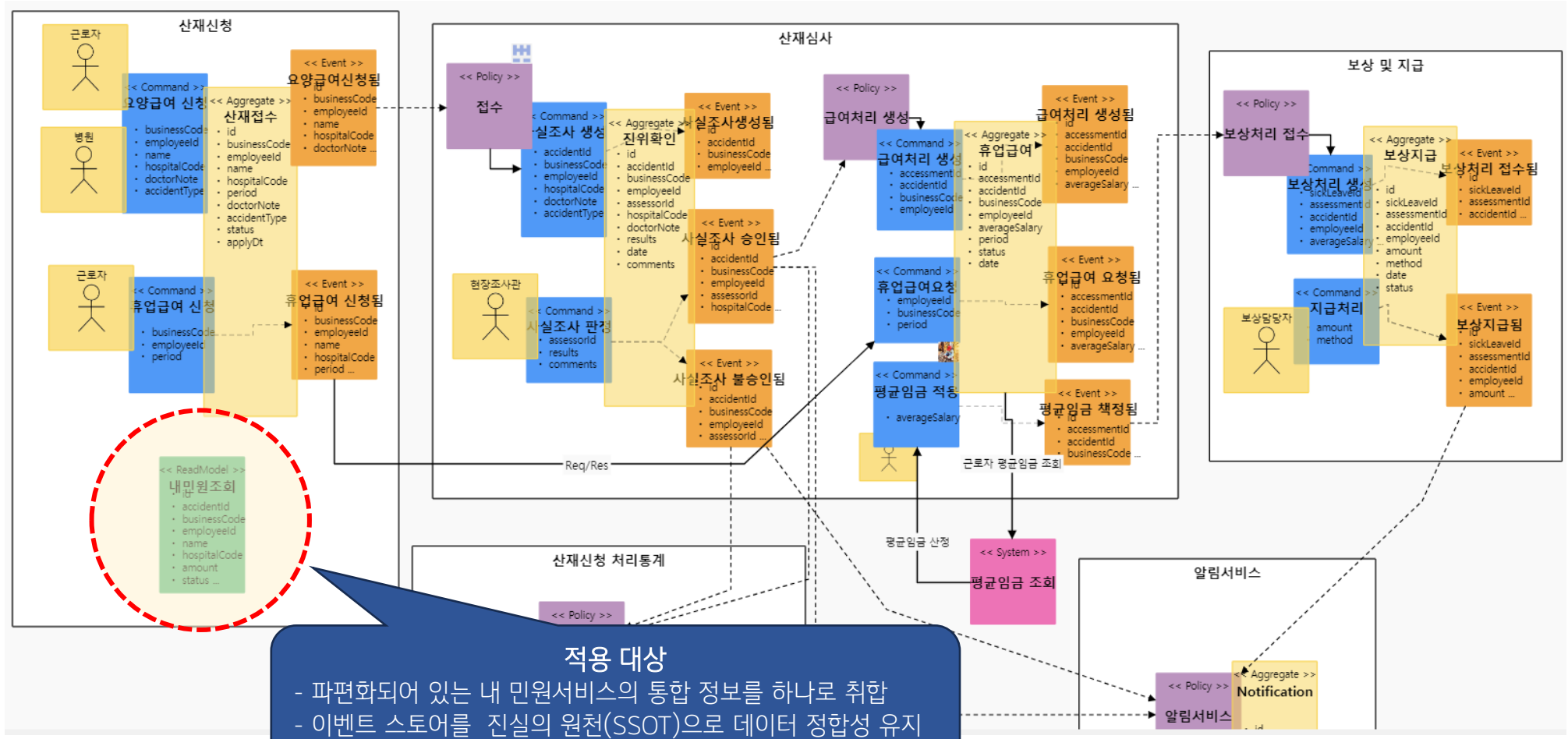
- 비즈니스 로직을 처리하는 데이터 변경영역(C,U,D)과 데이터 조회(R) 영역을 분리해 복잡도 해소, 성능이슈 해결
- 파편화 되어 있는 MSA 분산 환경에서 대쉬보드(ex. 마이페이지) 등 통합 데이터를 구축하는 전략



- CQRS with EventSourcing architecture

Lab Time

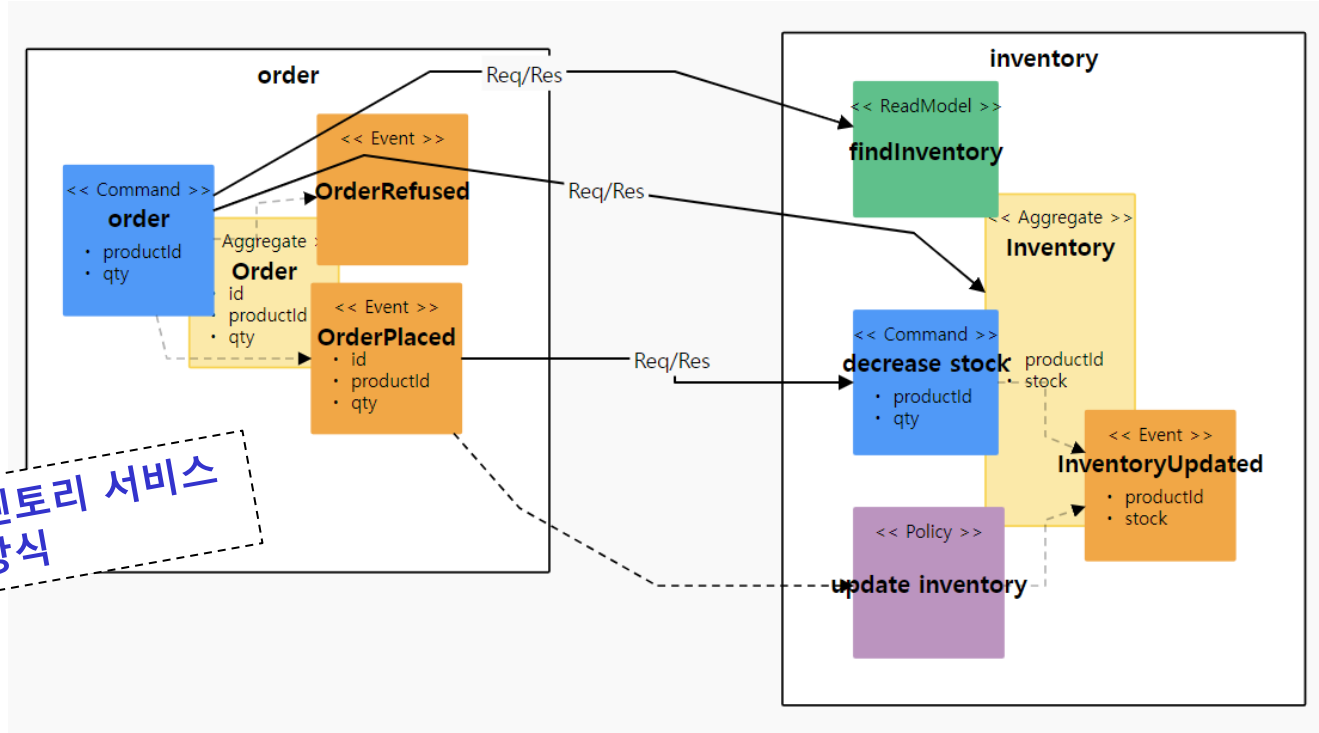
데이터 프로젝션 패턴



MSA School

MSA 테스트 패턴

- 다양한 방식과 API로 상호 참조되는 분산 환경에서 서비스들은 실행 중이 아니어도 병렬개발과 테스트가 가능해야 함



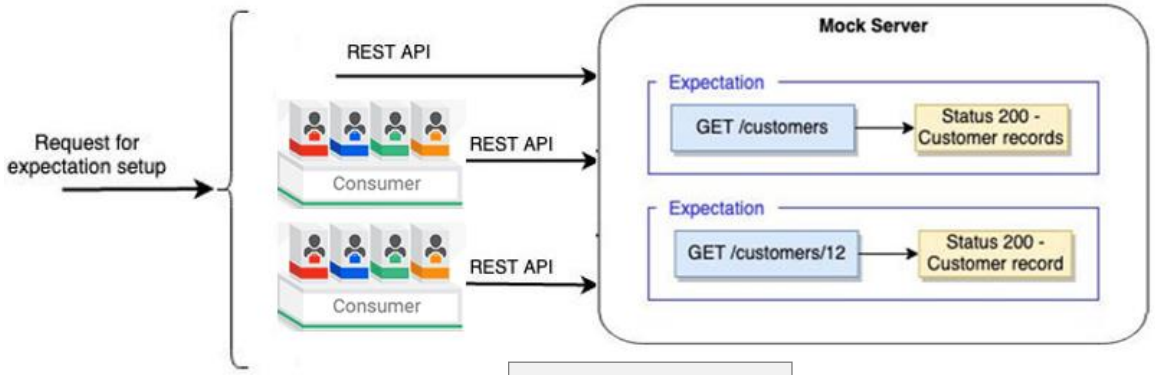
➔ 주문팀은 개발에 있어, 언제든지 상품팀의 인벤토리 서비스를 참조해 병렬 개발이 가능한 환경을 구축해야..

➔ 다른 팀이 참조하는 상품팀의 API는 일방적으로 임의 변경하지 않고, 항상 합의된 계약을 준수해야..

MSA 테스트 패턴

- 마이크로서비스 병렬 개발과 합의된 계약 준수를 위한 디펜던시 API(서비스) Mocking 기반 테스트 방법론

✓ 외부 API를 Mocking해 내부에 디펜던시 API를 구축하는 전략

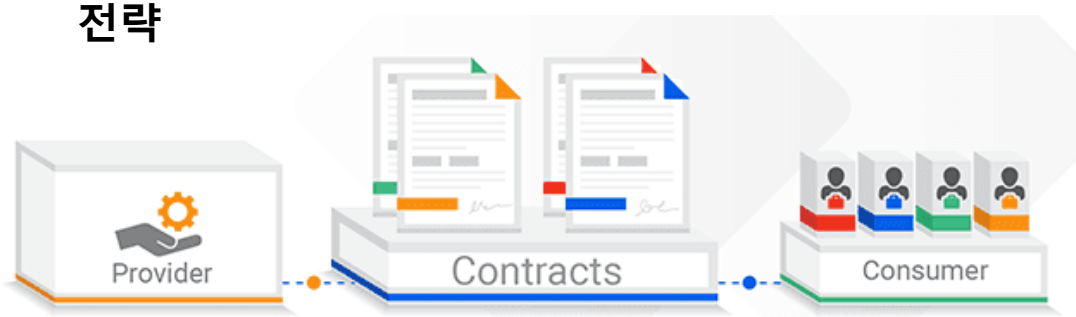


➔ Tools : Microcks, Prism

➔ API 기반 Mock Server 생성 & Kafka Mocking 지원

➔ Swagger UI, Example Spec.(given/when/then) 지원

✓ 계약(Contract)을 체결해 임의 변경을 원천 차단하는 전략



➔ Tools : Pact, Spring Cloud Contract

➔ Contract 기반 Test 결과를 Mock Server로 생성해 제공

➔ Stub Runner 통해 WireMock 테스트 및 상호 검증

MSA 테스트 패턴

- 병렬 개발과 합의된 계약 준수를 위한 디펜던시 API(서비스) Mocking 기반 구현 패턴

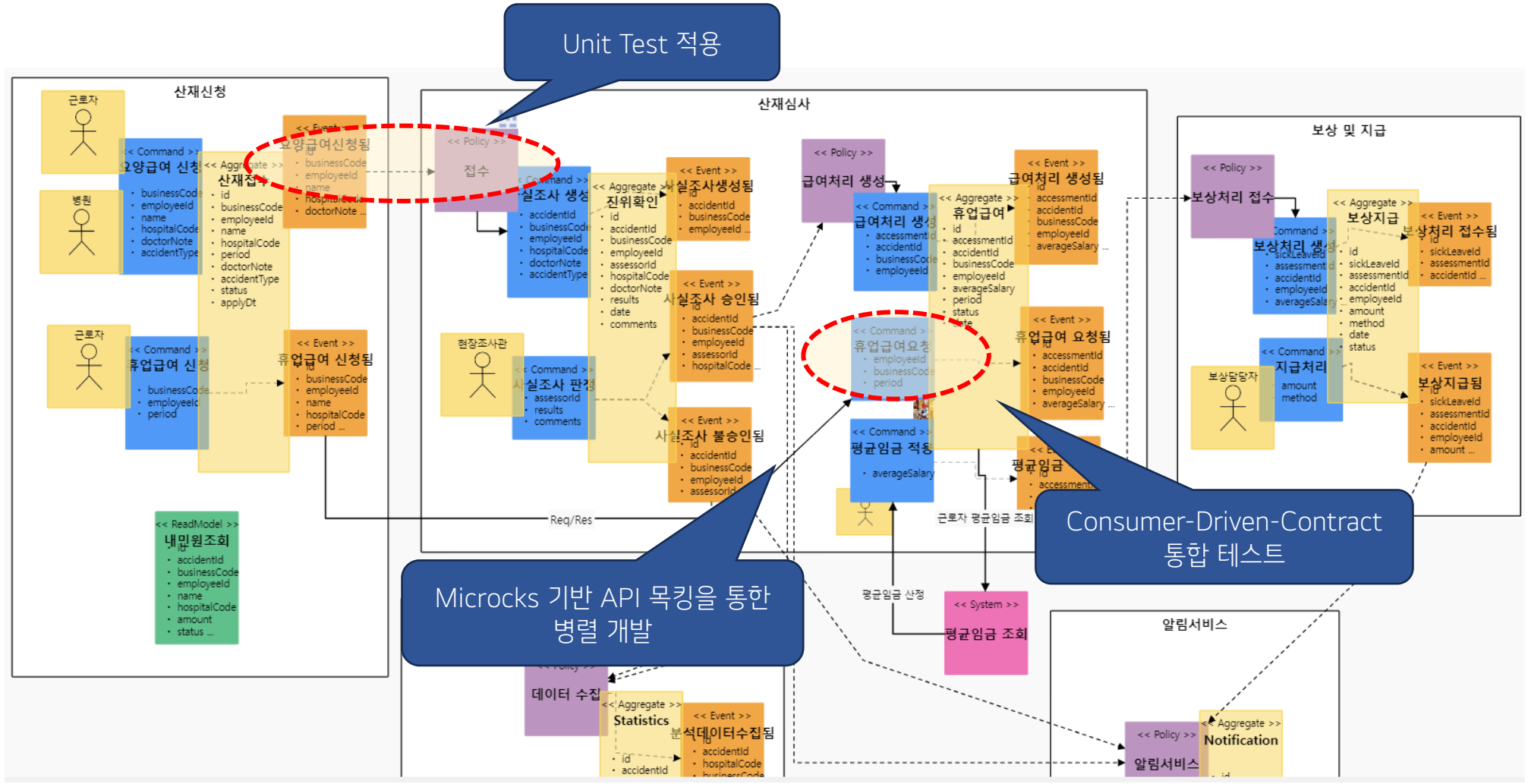
구현 패턴	목 적	특 징
API 목킹 (API Mocking)	실제 백엔드 서비스 구현없이 API 응답을 모의(Mock)해 이의 API에 맞도록 병렬 개발	• 백엔드 서비스가 아직 개발되지 않았을 때, • API 목킹을 통해 통합 테스트를 수행하고자 할 때,
CDC (Consumer-Driven Contract)	스펙 정의, 문서화, 스펙과 일치하는 테스트코드로 하위 호환성 유지여부 점검	• 소비자 중심의 계약을 정의하고, 이 계약을 기반으로 프로바이더와 컨슈머간의 통합 테스트

✓ 외부 API가 변경되더라도 Mocking Code는 변경되지 않기 때문에, Contract 정합성 유지, 스펙과 일치하는 테스트 코드, 런타임에서도 계약 준수를 위해서는 → CDC 테스트

✓ 각각의 구현 전략에는 Given–When–Then 테스트 패턴 적용

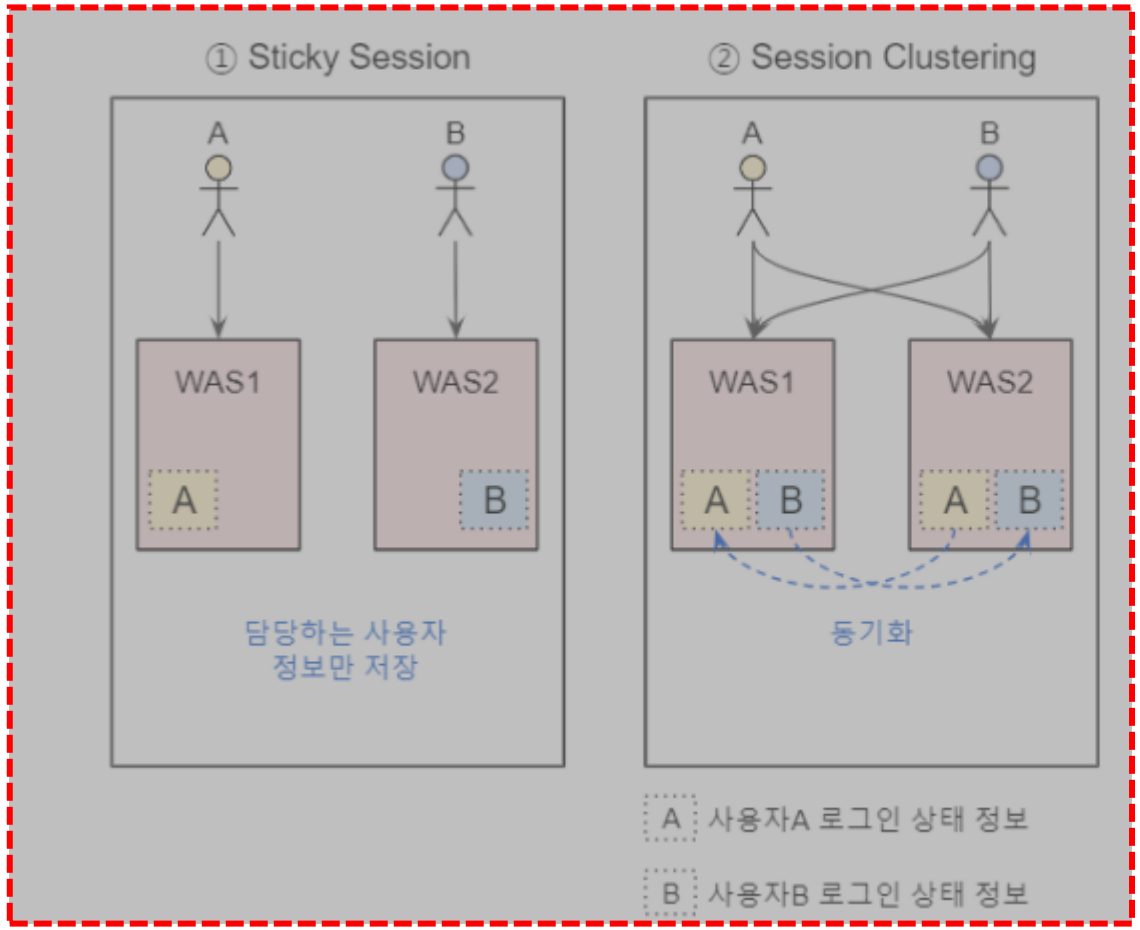
Lab Time

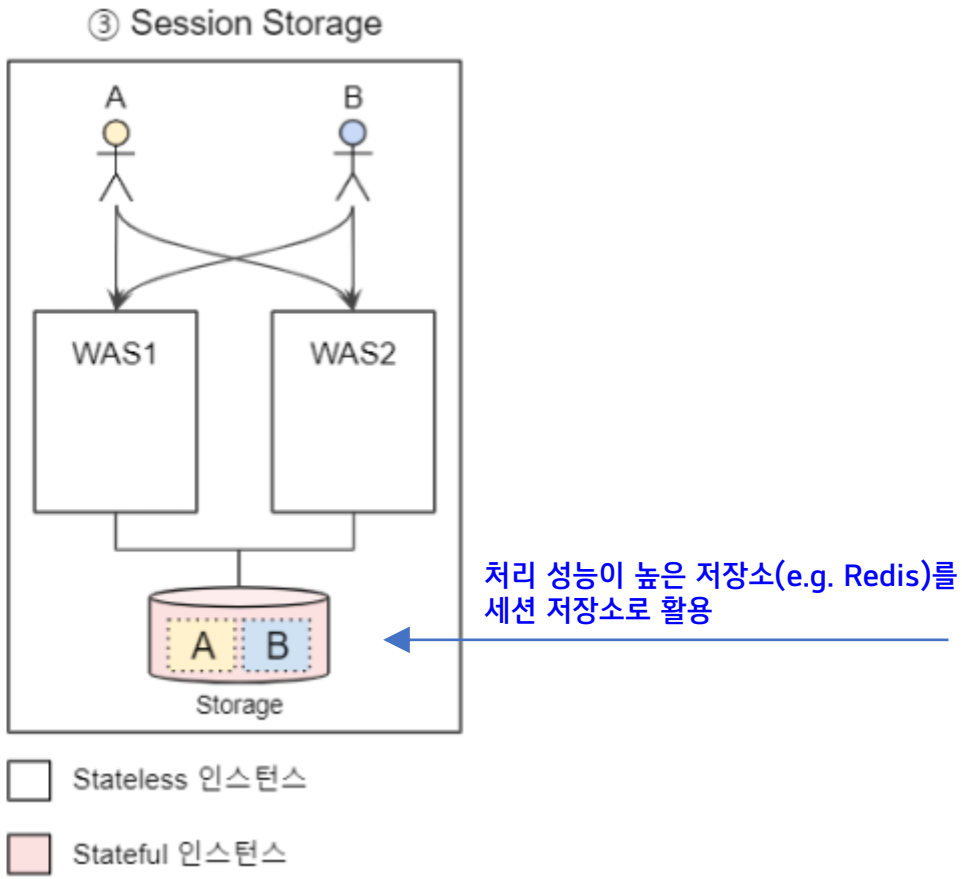
MSA 테스트 패턴



보안처리 패턴

- 인프라 확장/축소가 자유롭고 상태가 없는(Stateless한) 무중단 배포가 가능하기 위한 보안 패턴

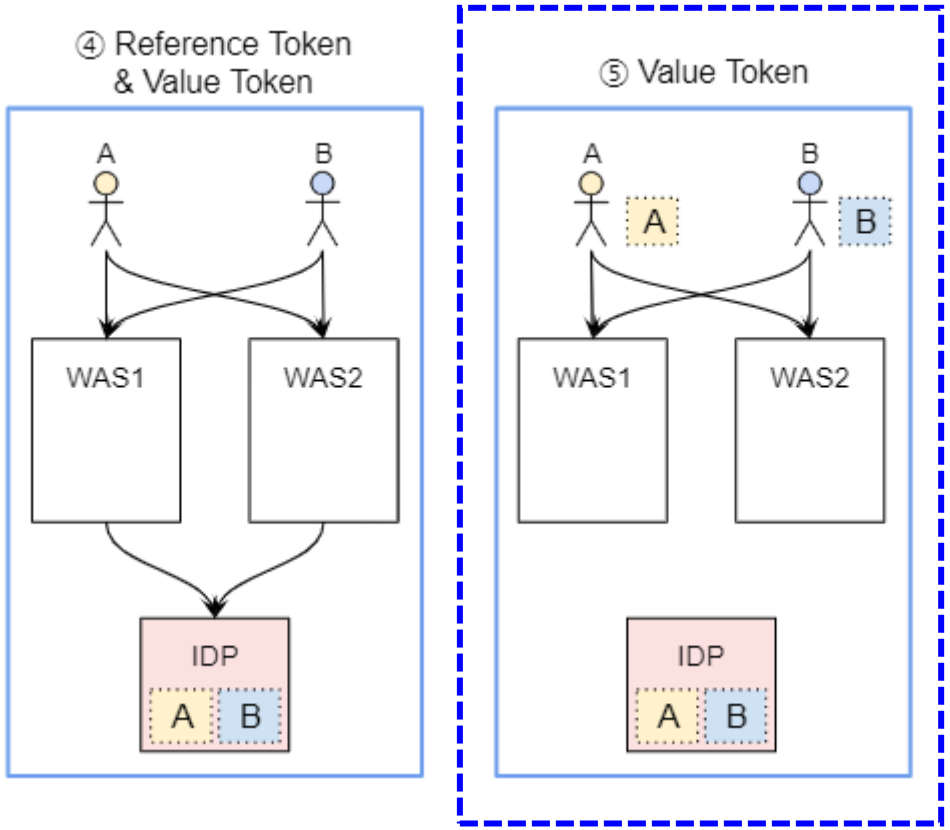




출처: https://blog.naver.com/wharup/223281848620

보안처리 패턴

- 인프라 확장/축소가 자유롭고 상태가 없는(Stateless한) 무중단 배포가 가능하기 위한 보안 패턴



* IDP : ID Provider

토큰기반 보안 구성 전략

- 모든 서비스들의 진입 관문인 게이트웨이 설치
- 게이트웨이와 밸류 토큰 인증서버 구성
- 게이트웨이는 인증된 토큰을 다운 스트림으로 전송
- 다운 스트림(백엔드/ 프론트엔드 서비스) 토큰 활용

출처: https://m.blog.naver.com/wharup/223329488539

보안처리 패턴

- 인프라 확장/축소가 자유롭고 상태가 없는(Stateless한) 무중단 배포가 가능하기 위한 보안 패턴

	단일 장애 지점	보안	인프라 비용
세션 저장소 방식	세션 저장소	◎	높음
레퍼런스 & 밸류 토큰	IDP, or 토큰 저장소	◎	높음
밸류 토큰	없음	○	낮음

밸류 토큰(e.g. JWT Token) 기반 SSO 솔루션

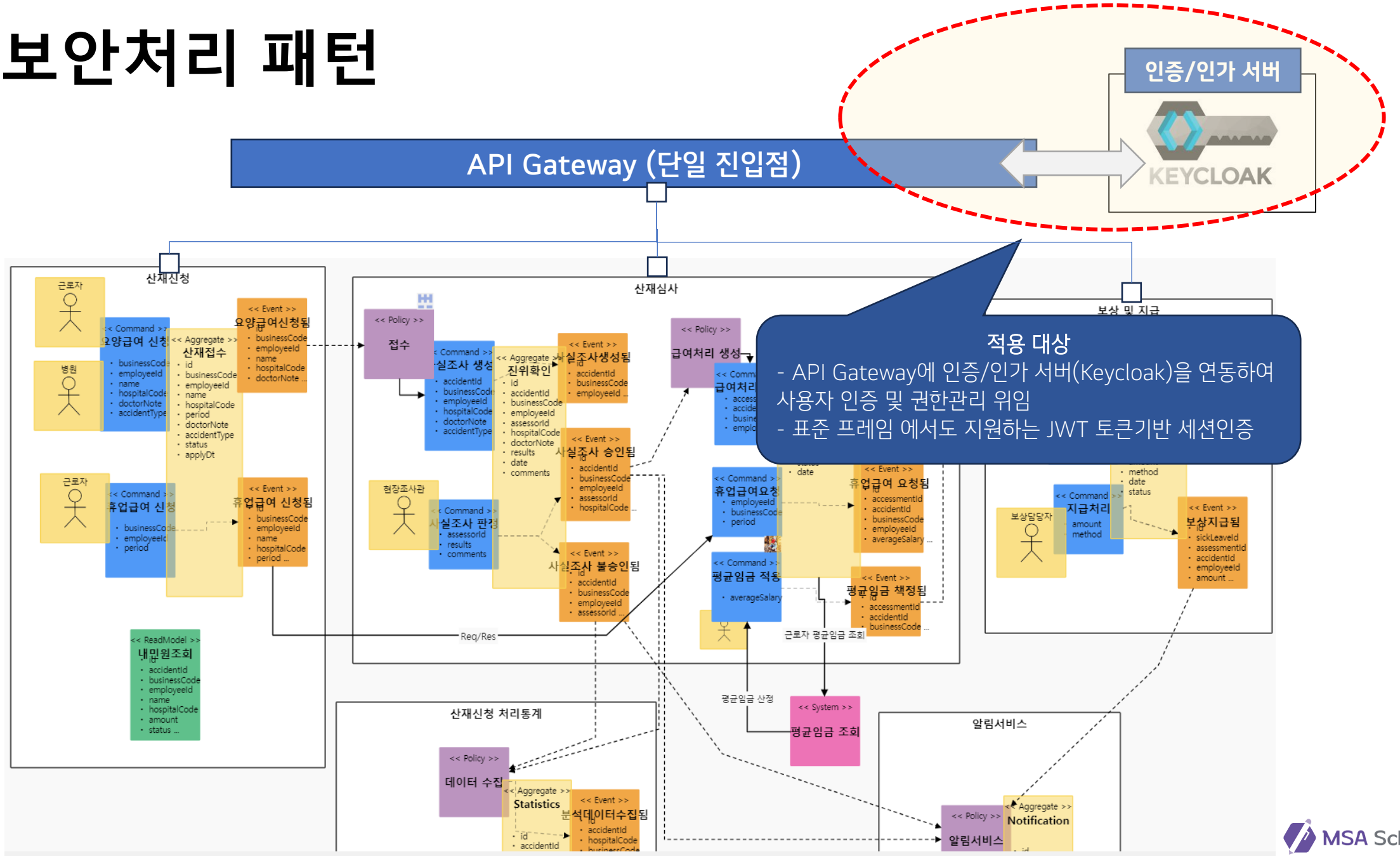
✓ Keycloak (https://Keycloak.org, CNCF)

✓ CNCF(cncf.io)에 등재된 Enterprise Cloud native SSO 실행 환경을 위한 솔루션

출처: https://m.blog.naver.com/wharup/223329488539

Lab Time

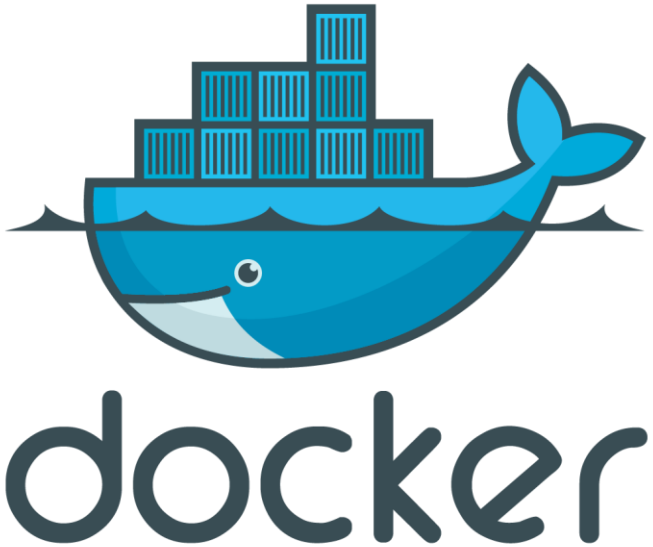
보안처리 패턴



MSA School

컨테이너 패키징

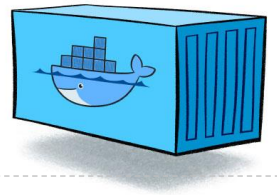
- 구현된 각 마이크로서비스는 각자의 에코시스템을 만들어 독립적으로 자생
- 이를 위한 클라우드 기술로 도커(Docker)와 컨테이너(Container)를 활용해 격리된 환경으로 패키징



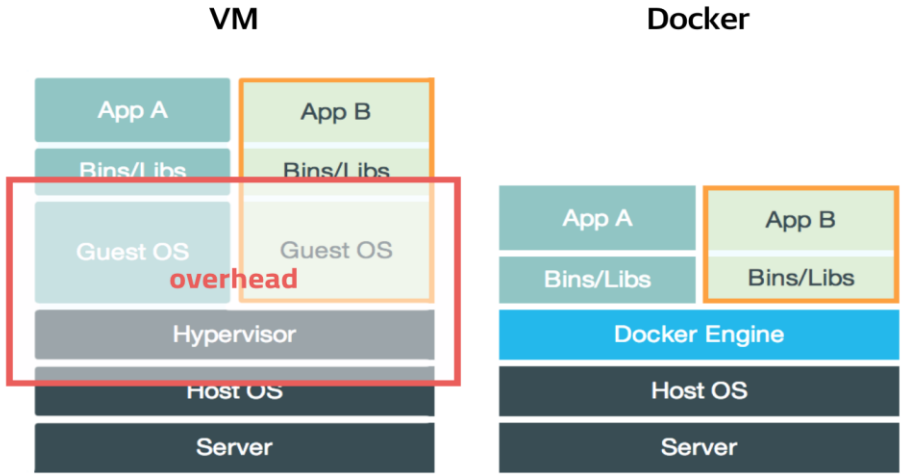
✓ 2013년 3월,
dotCloud 창업자 Solomon Hykes가 Pycon Conference에서 발표

✓ Go 언어로 작성된 “The future of linux Containers”

✓ Container based
- ✓ 프로세스 Isolation(격리) 기술
- ✓ 오픈소스 가상화 플랫폼



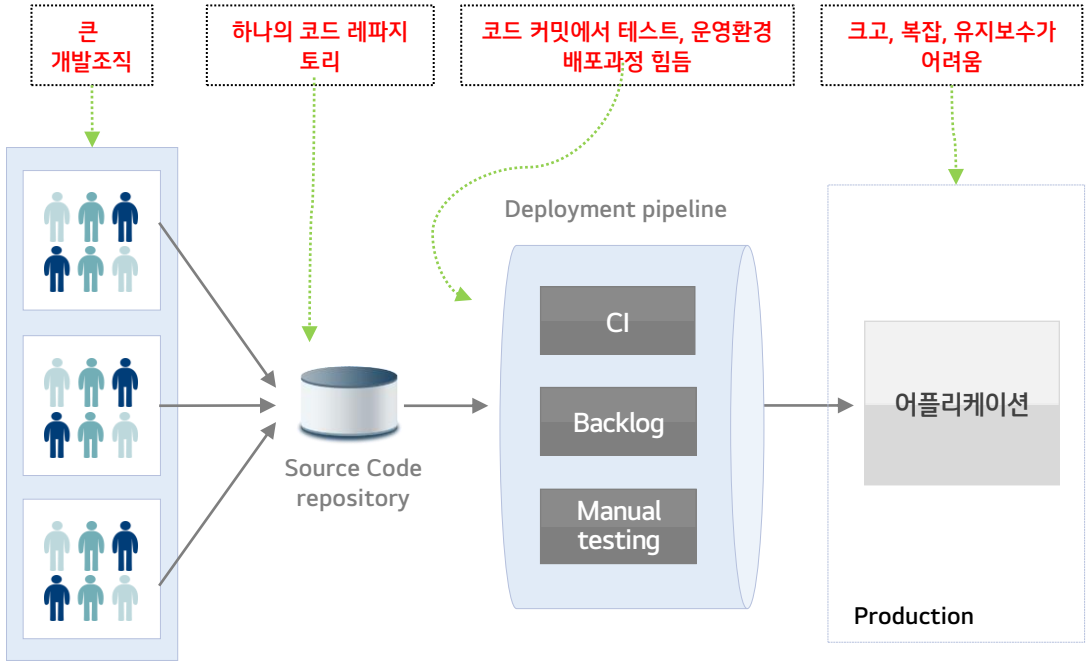
✓ 가상머신 .vs. Docker



컨테이너 패키징

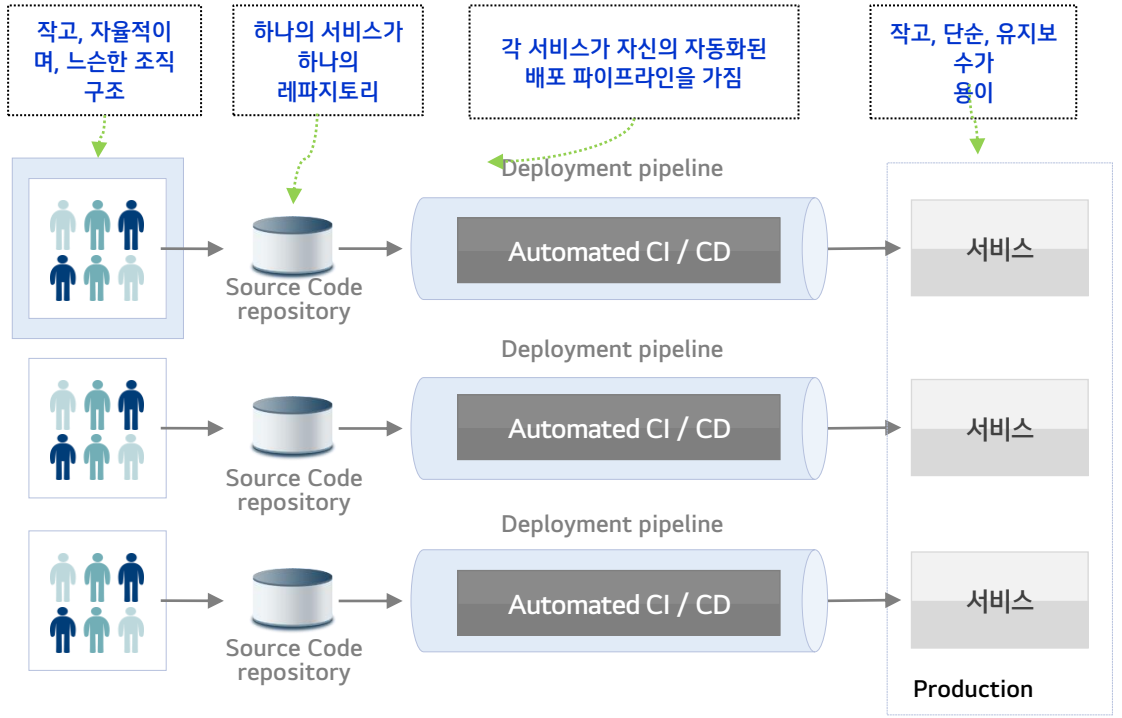
모노리식 개발 및 운영 환경

- 모노리식 환경하에서는 큰 개발팀이 하나의 소스코드 레파지토리에 변경 사항을 커밋하므로 코드간 상호 의존도가 높아 신규 개발 및 변경이 어려움
- 작은 변경에도 전제를 다시 테스트/ 배포하는 구조이므로 통합 스케줄에 맞춘 파이프 라인을 적용하기가 어렵고 Delivery 시간이 과다 소요



마이크로서비스 개발 및 운영 환경

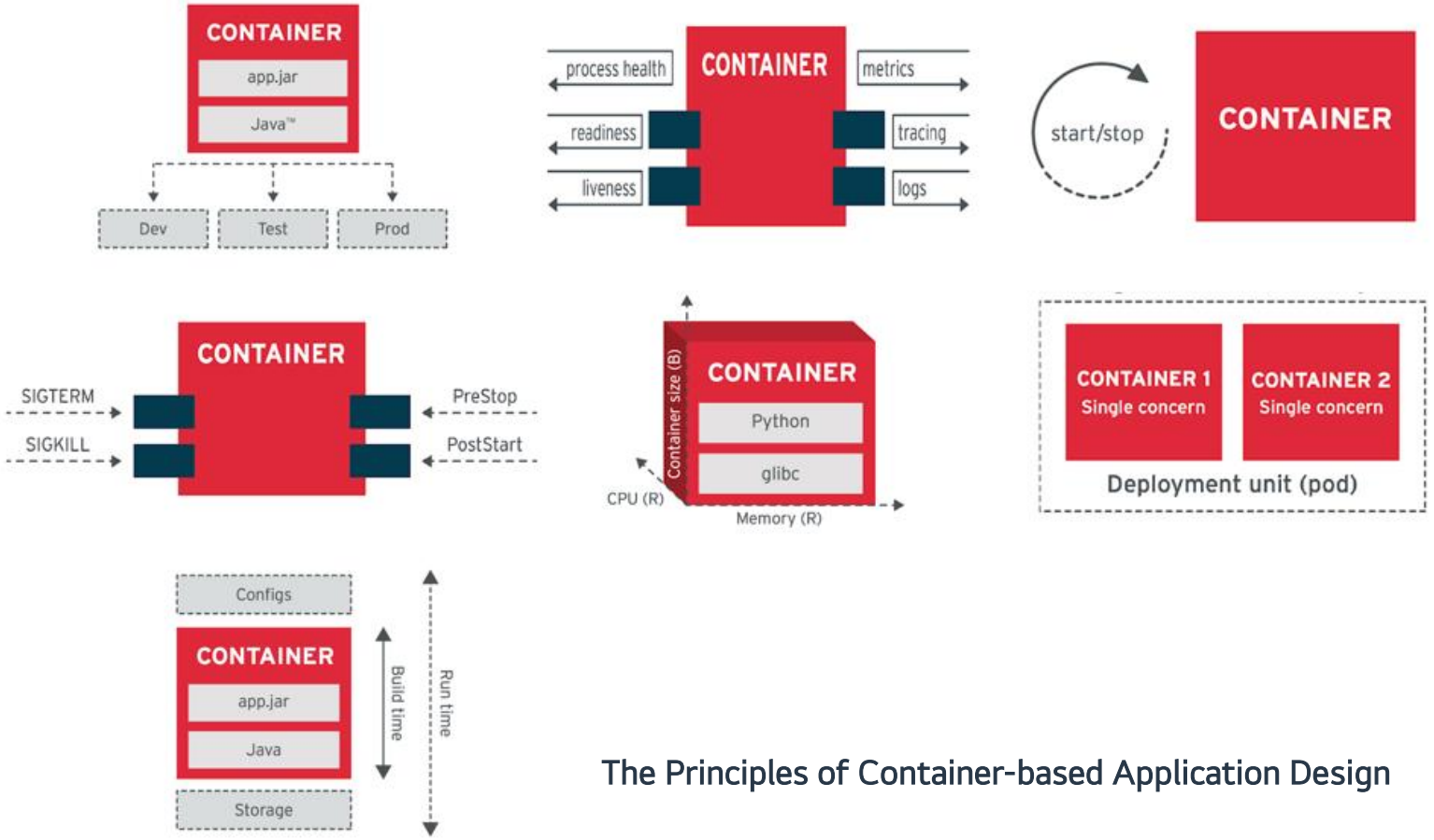
- 작고 분화된 조직에서 서비스를 작은 크기로 나누어 개발하므로 해당 비즈니스 로직에만 집중하게 되어 개발 생산성 향상
- 각 팀만 다른 팀 간섭없이 언제든 새로운 기능의 테스트와 롤 아웃이 가능하고 연관된 마이크로서비스만 테스트를 수행하므로 개발/테스트/배포 일정이 대폭 축소



다음 편 예고 (6/4) – 쿠버네티스 Patterns

- 쿠버네티스 배포 자동화
- 서비스 운영 및 트러블 슈팅
- 컨테이너 오토 스케일
- 서비스 메시의 활용
- 옵저빌리티와 로그리게이션
- 파이프라인 툴 체인

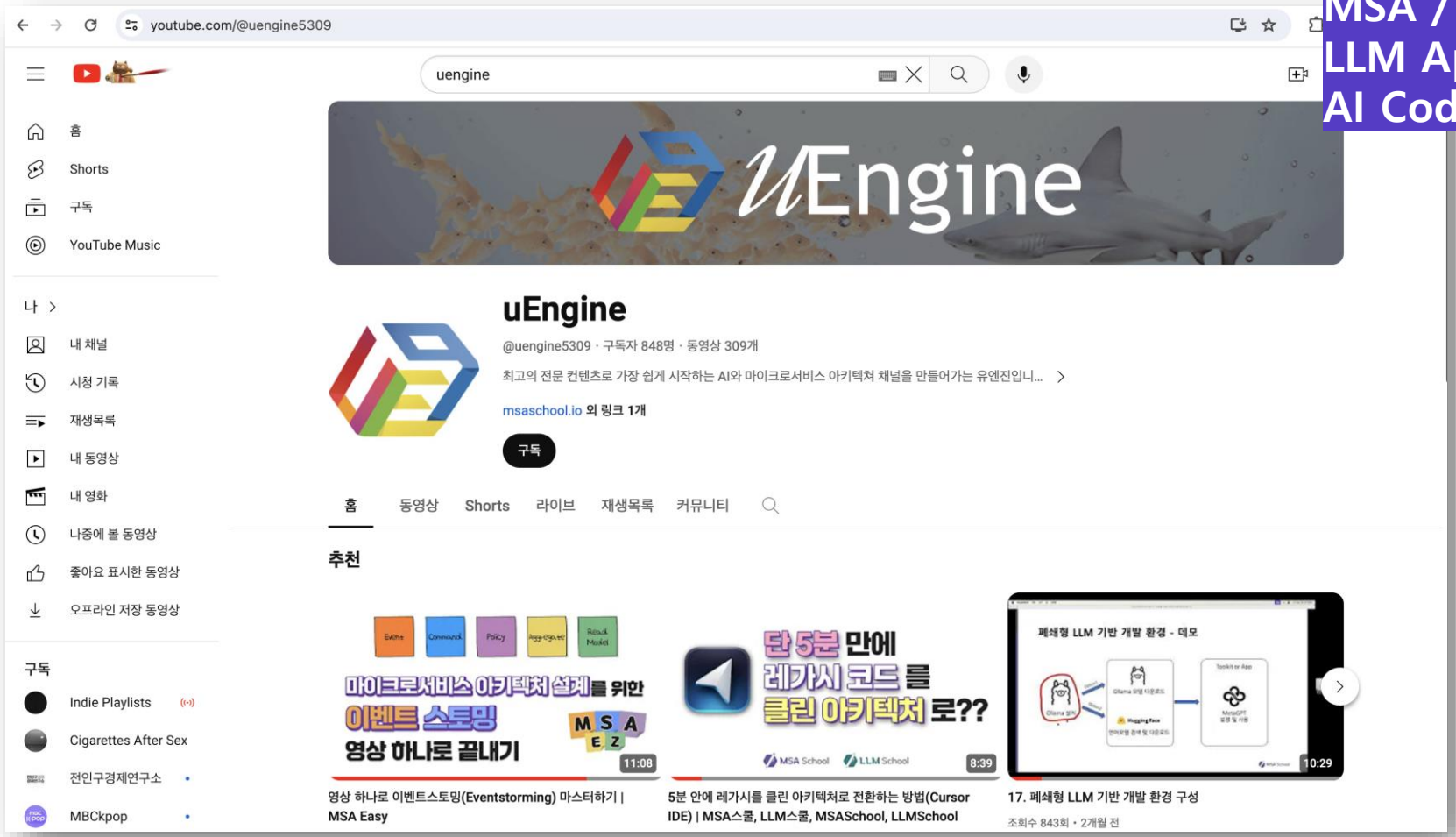
등을 MSA Easy (CLI 기반)를 통하여 클러스터에 배포하고 모니터링하는 전체 과정을 살펴봅니다



The Principles of Container-based Application Design

유엔진 콘텐츠를 계속 시청하시려면 유튜브를 구독하세요

주제:
MSA / CNA / DDD
LLM Application 개발방법
AI Coding / Legacy Modernization



MSA School