Event-Storming based MSA training

3rd Jan 2020, ver2.0

uEngine

Table of content

Microservice and
Event-storming-Based
DevOps Project

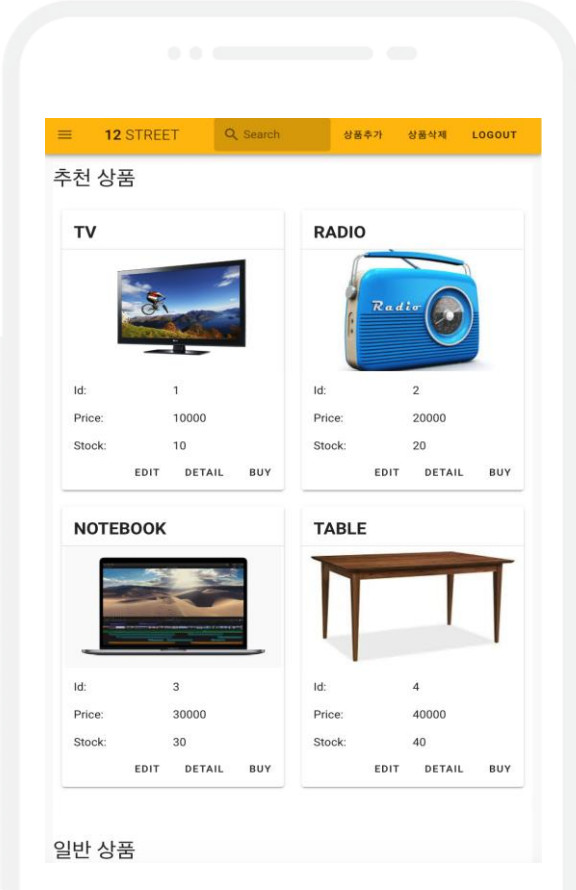
UEngine

Target Domain

: Online Shopping Mall (12 STREET)

- Vision & Mission

Service resiliency	24시간 365일 접속과 주문이 가능 : 자동화된 회복, 장애전파최소, 무정지 재배포
Customer responsiveness	다양한 고객 기능 요구사항의 탐색과 반영
Scalability	조직, 기능 및 데이터의 확장에 열려 있는 아키텍처 : Feature-driven-development



Organization & KPI Definition - 창업시기

- Multiple Concerns Single Organization
- Horizontally aligned

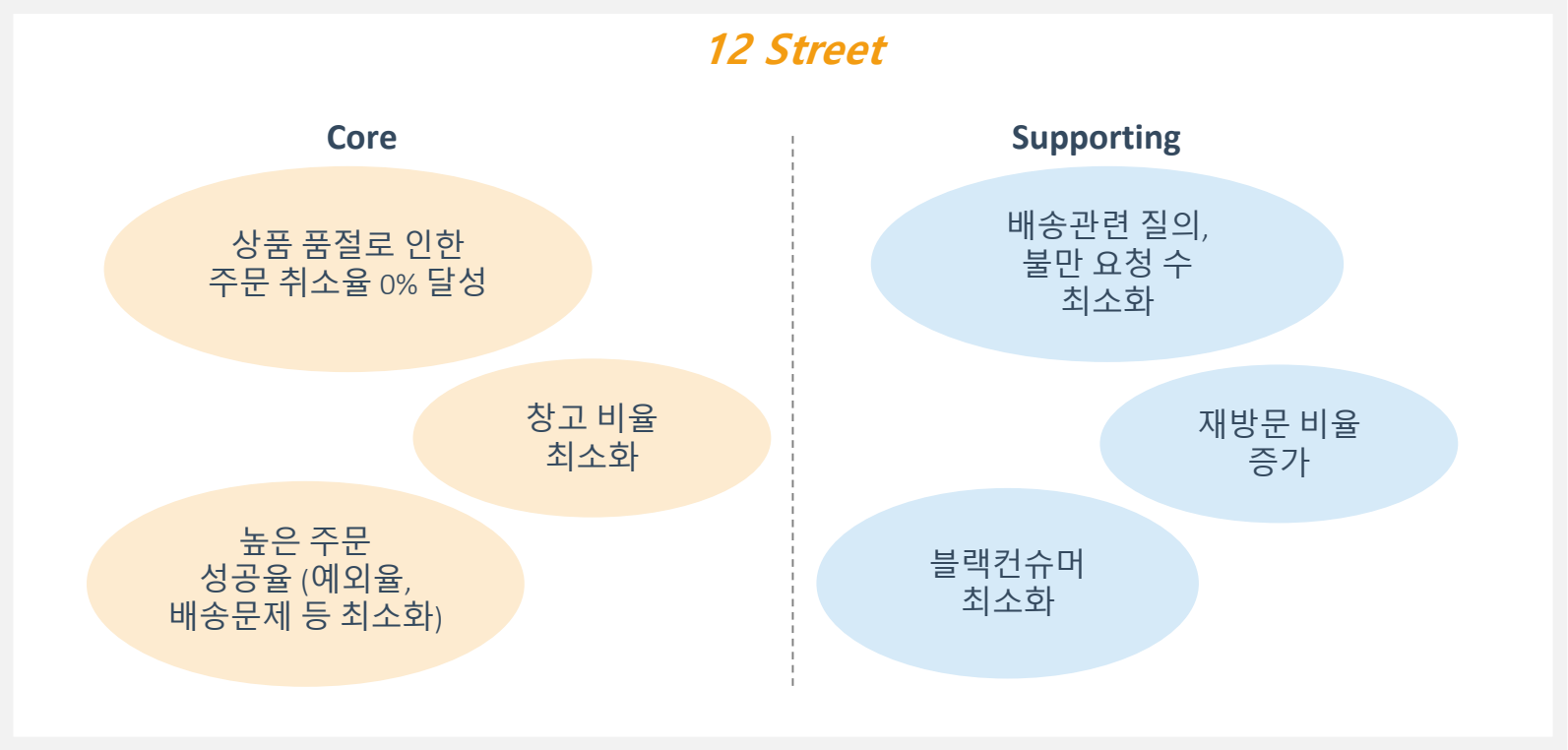
User Stories

1. 고객이 주문할 상품과 개수를 선택하여 주문버튼을 클릭한다.
2. 주문이 벌어지면 배송팀은 해당 상품에 대한 배송을 준비한다.
3. 주문과 취소가 완료됨에 따라서 상품 관리팀이 관리하는 재고량이 변경(+/-)된다.
4. 품절 상품에 대해서도 고객이 구매 가능하며, 상품이 입고된 후, 이를 구매 대기 고객에게 인폼하고 처리(재 입고시 지연 배송, 주문 취소) 한다.

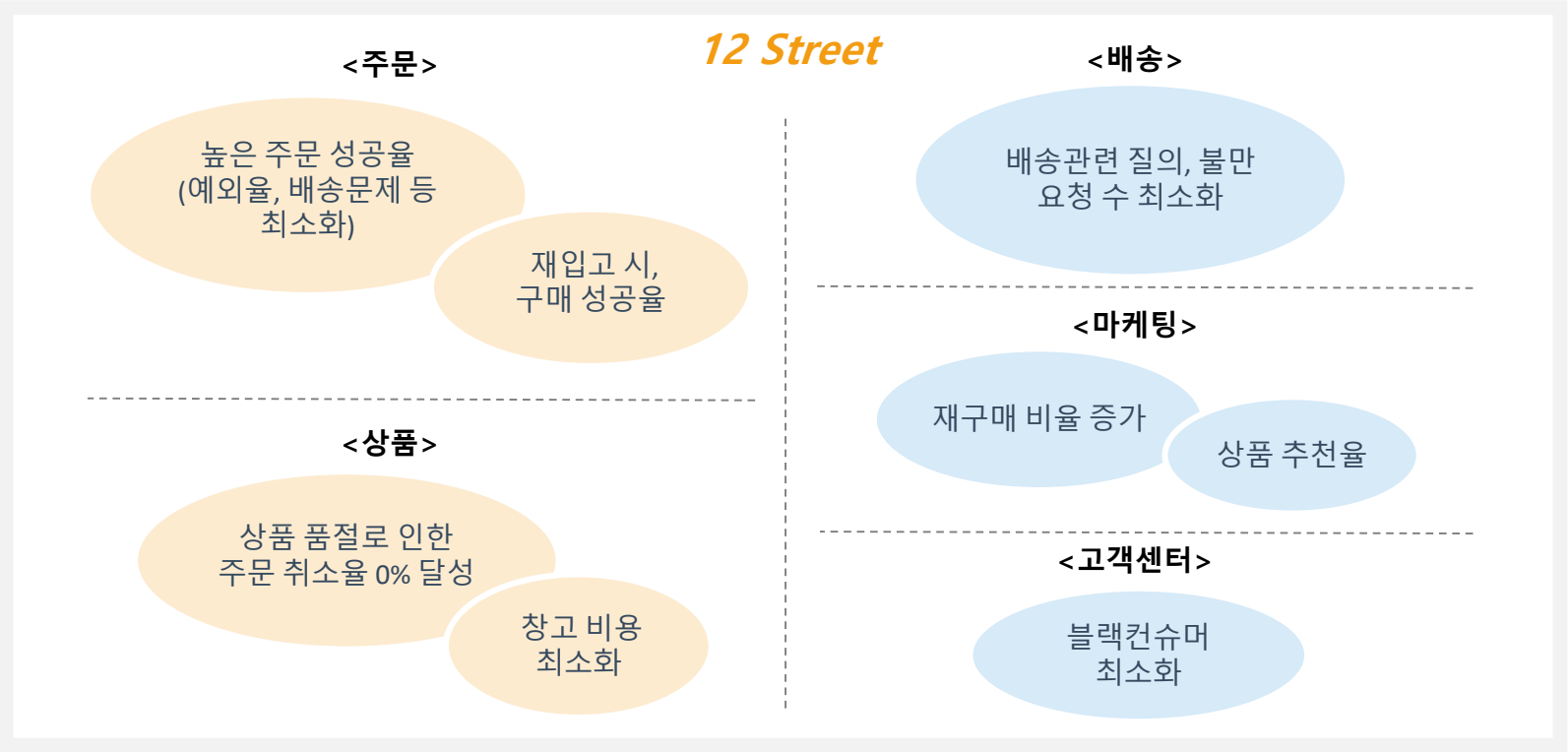
1. The customer enters the number of products to order and clicks the order button.
2. When an order is placed, the delivery team arranges for delivery of the product.
3. As the order or cancellation is completed, the product stock is changed(+/-).
4. Customers can also purchase products that are out of stock and This can be resolved after registering products later.

"책임소재가 불분명한 요구사항"

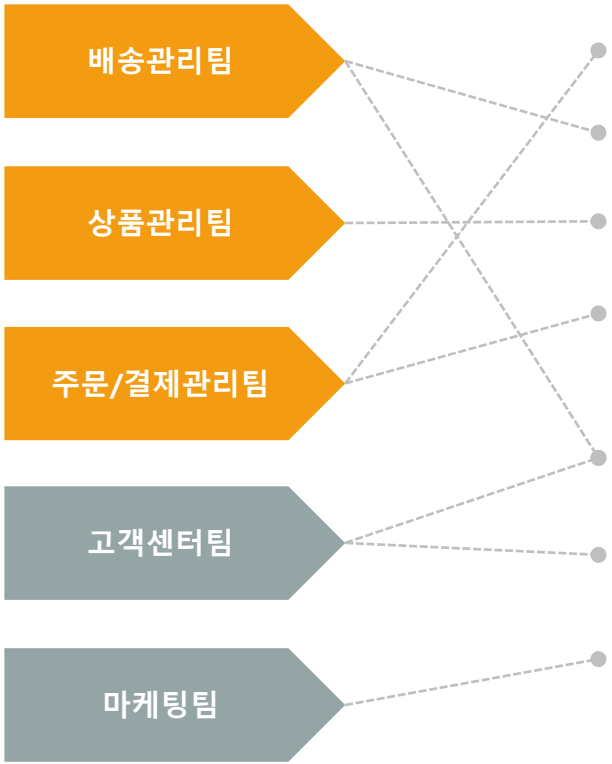
Separate Core Domain from Supporting Domain



Separation of Concerns - 회사의 성장



신설된 User Stories 및 Concerns Mapping



1. The customer enters the number of products to order and clicks the order button.
2. When an order is placed, the delivery team arranges for delivery of the product.
3. As the order or cancellation is completed, the product stock is changed(+/-).
4. Customers can also purchase products that are out of stock and This can be resolved after registering products later

5. A notification is sent to the customer whenever the delivery status changes.
6. Customers who repeat orders and cancellations frequently are blacklisted, and customers with high order purchase rates are whitelisted.
7. The marketing team mails recommendation products tailored to the customers at 9 am daily.

Table of content

Microservice and
Event-storming-Based
DevOps Project

UEngine

AS-IS: Pain-points

A 사 의료분야 SaaS 운영

- 서비스 업그래이드가 수시로 요청이 들어와 거의 매일 야근중. 개발자 행복지수가 매우 낮음.
- 한팀의 반영이 전체팀의 반영에 영향을 주어 거의 매일 야근해야 함. 행복지수 낮음을 토로함.
- 테넌트별 다형성 지원을 제대로 하지 못하여 가입고객이 늘 때마다 전체 관리 비용이 급수로 올라가는 한계에 봉착함
- 자체 IDC를 구성하여 하드웨어, 미들웨어 구성을 직접해야 하는 비용문제.

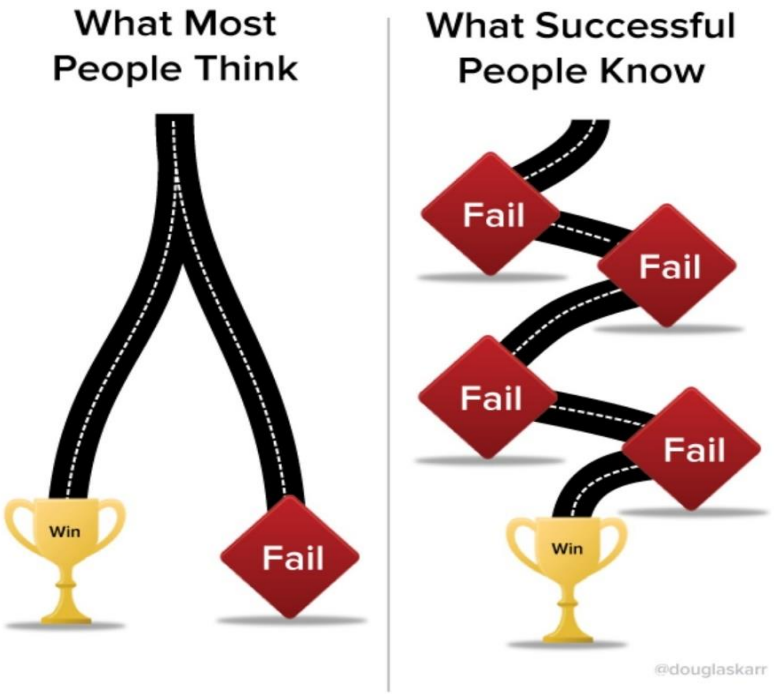
B 사 제조분야 SaaS 운영

- 운영팀과 개발팀이 분리되어 개발팀의 반영을 운영팀이 거부하는 사례 발생
- 개발팀은 새로운 요건을 개발했으나, 이로인해 발생하는 오류가 두려워 배포를 꺼려함
- 현재 미국, 일본, 유럽 등 수요가 늘어나는 상황이나, 상기한 문제로 신규 고객의 요구사항을 받아들이지 못하는 상황
- 수동 운영의 문제로, SLA 준수가 되지 못하여 고객 클레임이 높은편
- 기존 모놀로씩 아키텍처의 한계로 장기적인 발전의 한계에 봉착

What is Agile?

Planning is important!

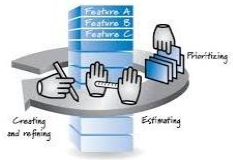
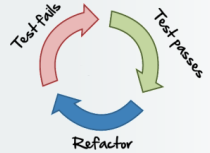
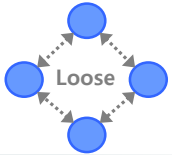
- Fail is bad
- Cost-driven



Agility is important

- Fail Cheap, Fail Fast, Fail Often
- Customer-driven

To be Agile

DELIVERY & REQUIREMENT MANAGEMENT	**MVP** (Minimum Viable Product) & **Incremental delivery**	
S/W ENGINEERING	**Test driven Development** & **Continuous Refactoring**	
TECHNICAL DESIGN	**Loosely Coupled Architecture**	
PROCESS & COLLABORATION	**Continuously Improving**	

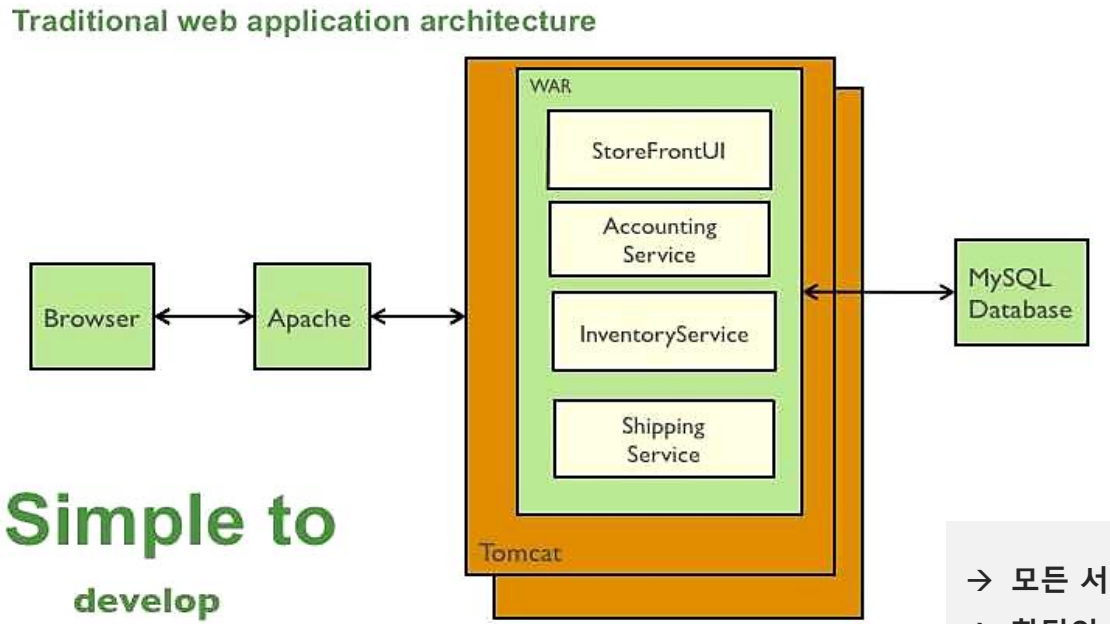
Continuous Delivery

Amazon, Google, Netflix, Facebook, Twitter는 얼마나 자주 배포할까요?

Company	Deploy Frequency	Deploy Lead Time	Reliability	Customer Responsiveness
Amazon	23,000 / day	Minutes	High	High
Google	5,500 / day	Minutes	High	High
Netflix	500 / day	Minutes	High	High
Facebook	1 / day	Hours	High	High
Twitter	3 / week	Hours	High	High
Typical enterprise	Once every 9 months	Months or quarters	Low / Medium	Low / Medium

출처: 도서 The Phoenix Project

Monolithic Architecture



→ 모든 서비스가 한번에 재배포
→ 한팀의 반영을 위하여 모든 팀이 대기
→ 지속적 딜리버리가 어려워

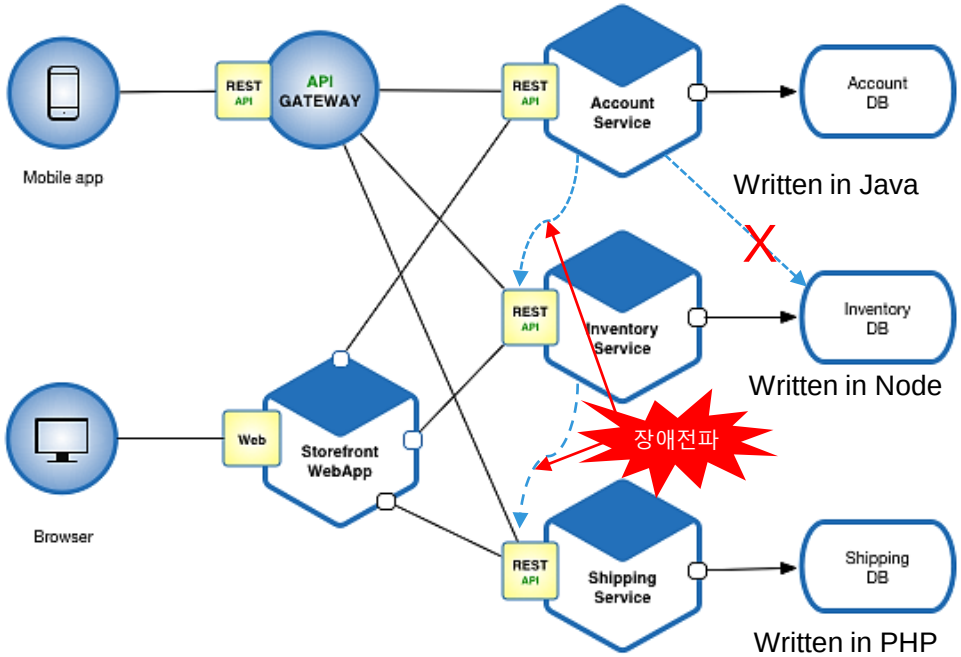
Jeff Bezos Mandate”

1. All teams will henceforth expose their data andfunctionality through service interfaces.
2. Teams must communicate with each other through these interfaces.
3. **There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.**

   …
4. The team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
5. Anyone who doesn't do this will be fired.

Approach #1 : Micro Service Architecture

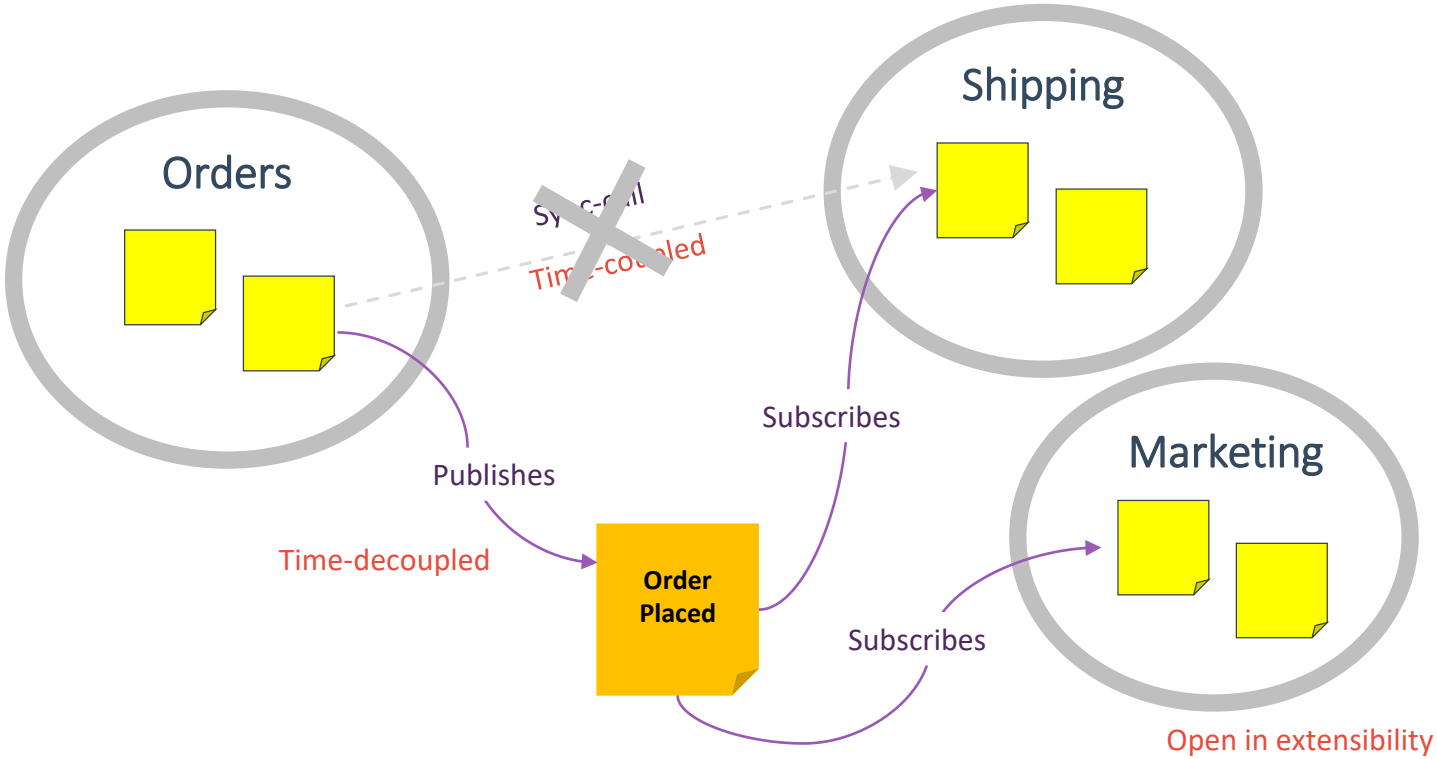
Contract based, Polyglot Programming
→ Separation of Concerns, Parallel Development, Easy Outsourcing



[Limitation]

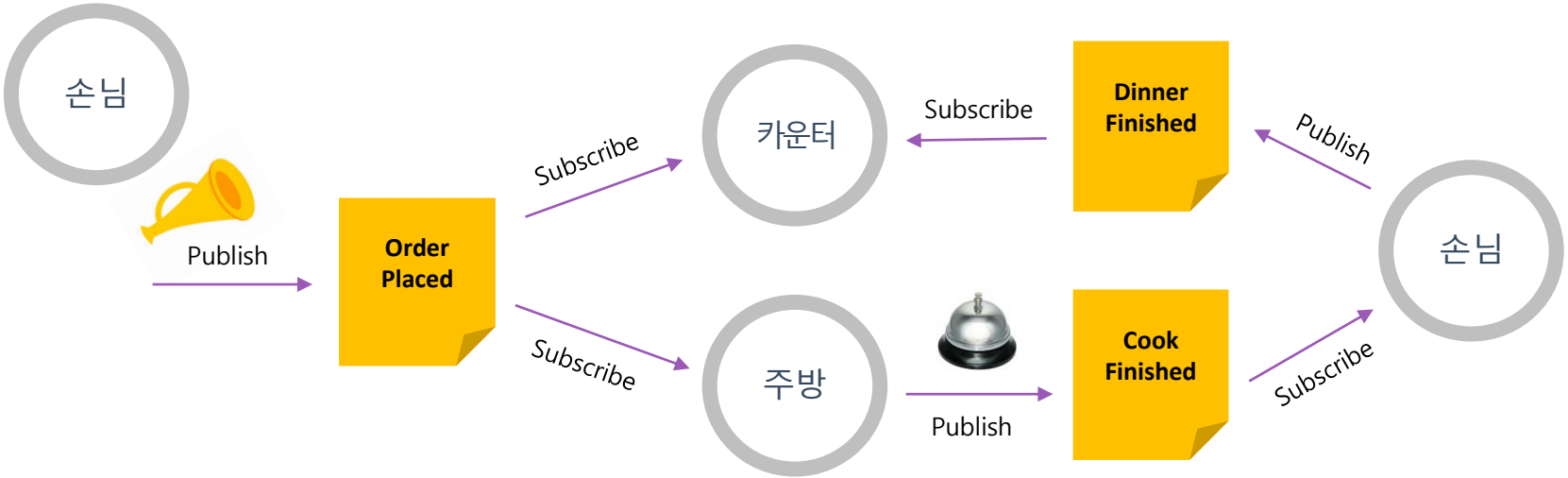
1. Code Coupling 해소
   But, Time Coupling 잔존
2. 서비스 Blocking 가능성 내재
3. 장애 전파 우려
4. Point to Point 연결에 따른 복잡한 스파게티 네트워크

Approach #2 : Event Driven Architecture

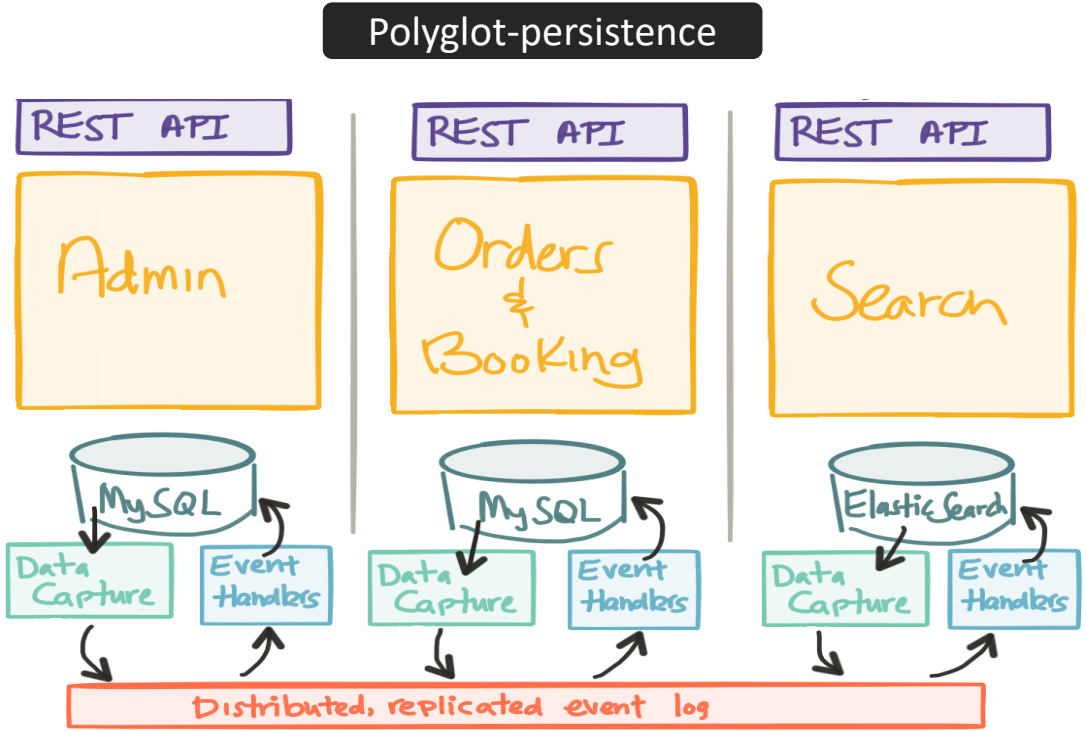


Event Driven in Real World: Restaurant





Approach #2 : Event Driven Architecture



Source: https://blog.redelastic.com/corporate-arts-crafts-modelling-reactive-systems-with-event-storming-73c6236f5dd7, Kevin webber

주문팀

주문 상태 변경

주문 상태 변경

주문 상태 변경

주문들어옴 (PO. No: 1, TV, 5개, 홍길동)

주문들어옴 (PO. No: 2, TV, 5개, 아무개)

주문취소됨 (PO. No:1)

주문들어옴 (PO. No.: 3, Radio, 2개, 아무개)

MySQL

ID	User	Item	Qty	status
1	홍길동	TV	5	Returned
2	아무개	TV	5	Delivery Started
3	아무개	Radio	2	ordered

상품팀

재고변경 (No:1, TV, 5)

재고변경 (No:1, TV, 0)

재입고됨 (ID:1, TV, 10개)

재고변경 (No:1, TV, 15개)

재고수정 (No:2, Radio, 8)

MongoDB

ID	Item	Stock
1	TV	15
2	Radio	8

배송팀

배송준비 (No: 1)

상품발송함 (주번:1)

배송준비 (No: 2)

상품발송함 (주번:2)

배송수거 (No: 3)

상품수거됨 (주번:1)

배송준비 (No: 3)

Redis

ID	Addr	PO-id	Qty	Status
1	강동구	1	5	Returned
2	강서구	2	5	Started
3	강서구	3	2	prepare

마케팅팀

선호도수집

새벽

추천상품 저장

File-System

Id	user	category	cnt
1	홍길동	Electronic	1
2	아무개	Electronic	2

오전

추천상품 이메일 발송

마이페이지

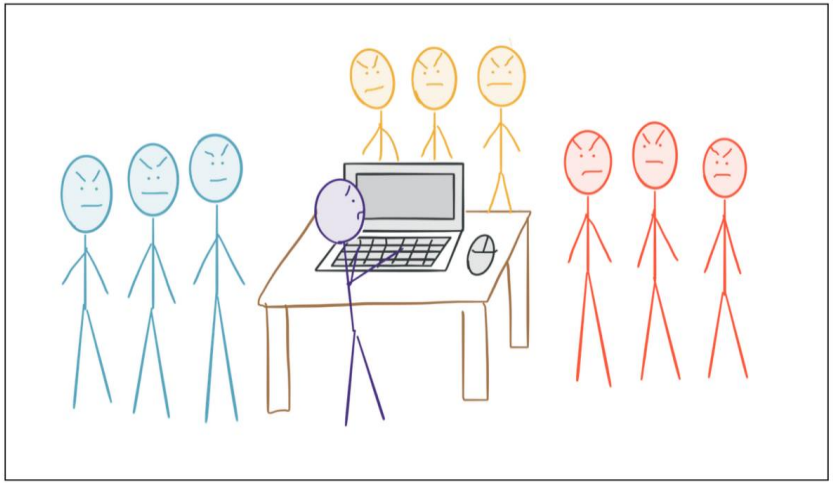
주문정보 수집 (PO. No:3)

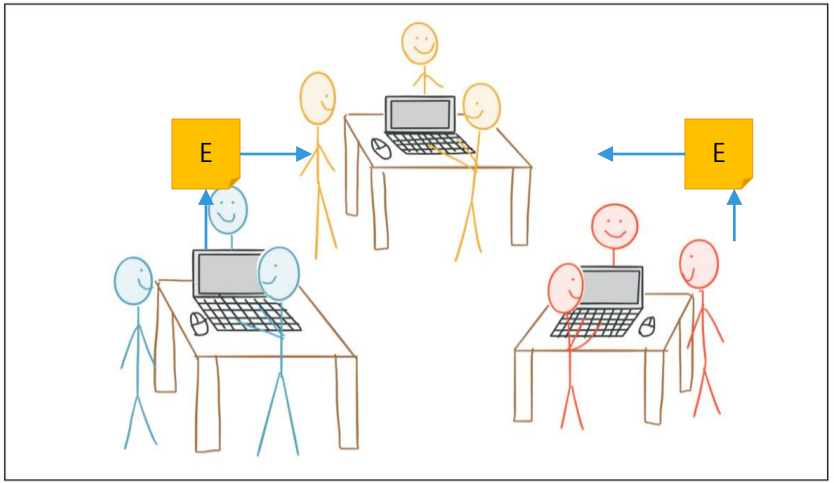
배송수거 정보수집 (PO. No:1)

HTML

id	User	Item	Qty	prc	o-stat	d-stat
001	홍길동	TV	5	50,000	cancel	Returned
002	아무개	TV	5	50,000	order	Delivery Started
003	아무개	RADIO	2	20,000	order	

Event Driven MSA Architecture





Monolith:

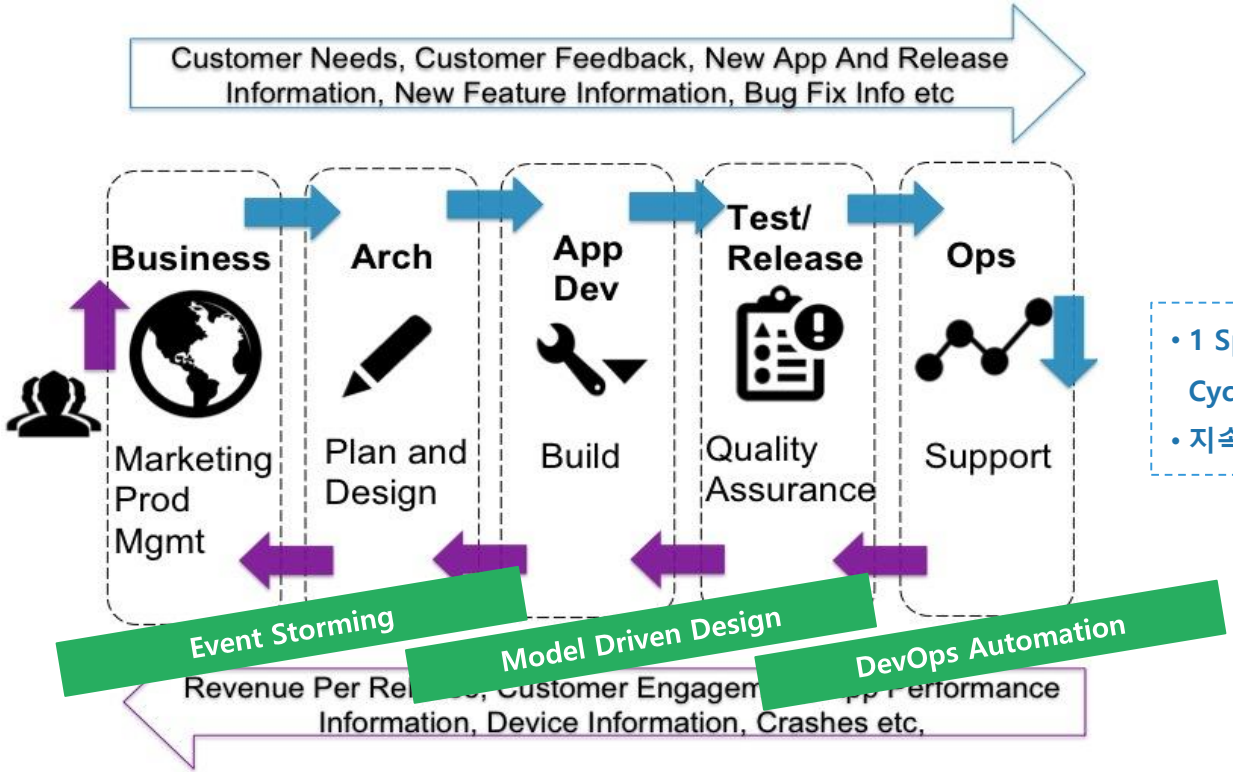
- With Canonical Model
- (“Rule Them All” model)

Reactive Microservices:

- Autonomously designed Model(Document)
- Polyglot Persistence

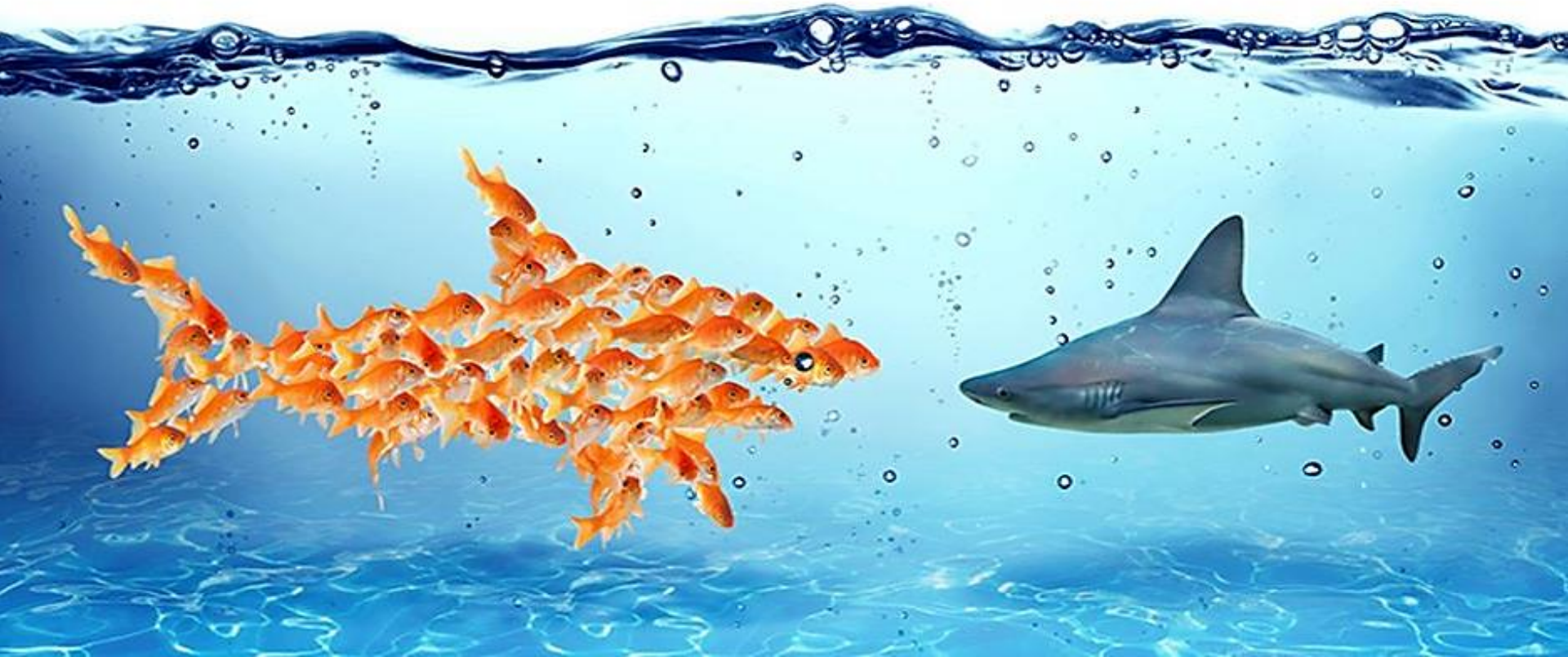
Source: https://blog.redelastic.com/corporate-arts-crafts-modelling-reactive-systems-with-event-storming-73c6236f5dd7, Kevin webber

Approach #3: BizDevOps Process



- 1 Sprint = 1~2 Weeks Cycle 기반
- 지속적인 Needs 탐색

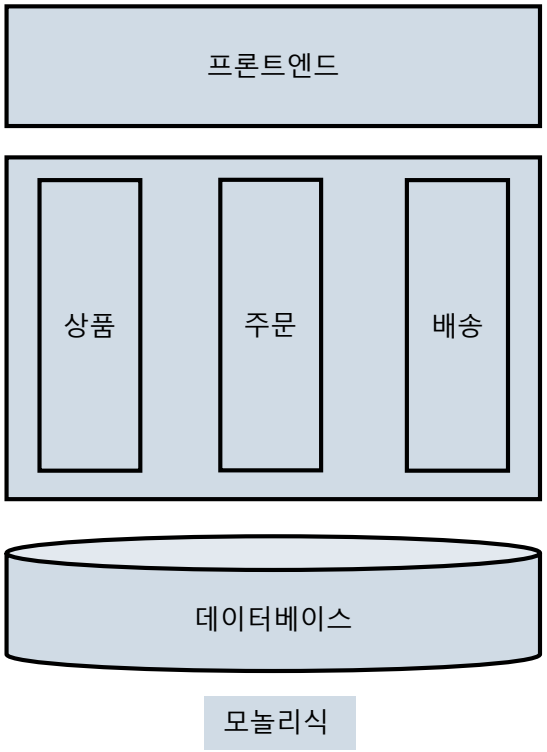
“서비스가 죽지 않으면 어떨까?”

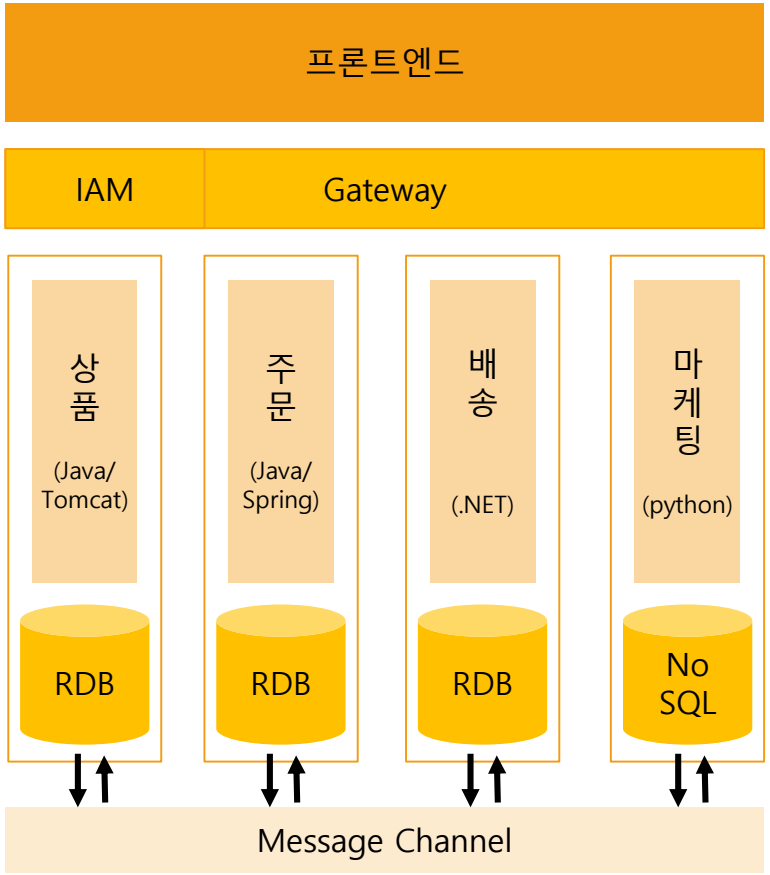


항상성이 (Self-Healing) 자동으로 유지되면 어떨까?

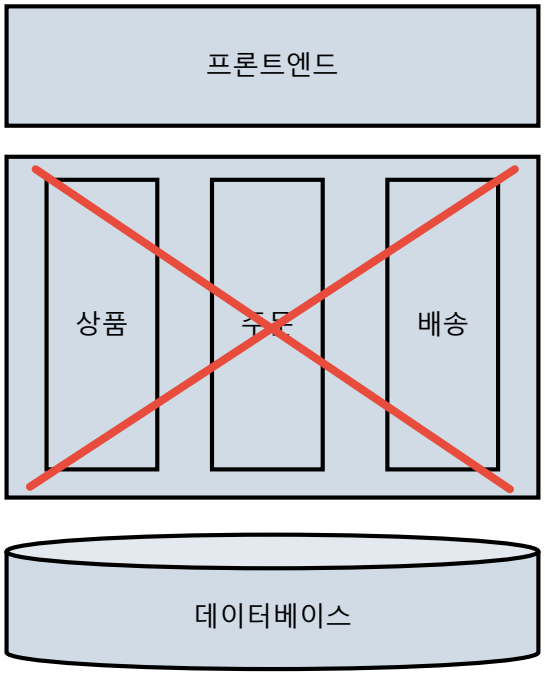
Kubernetes

적용 아키텍처

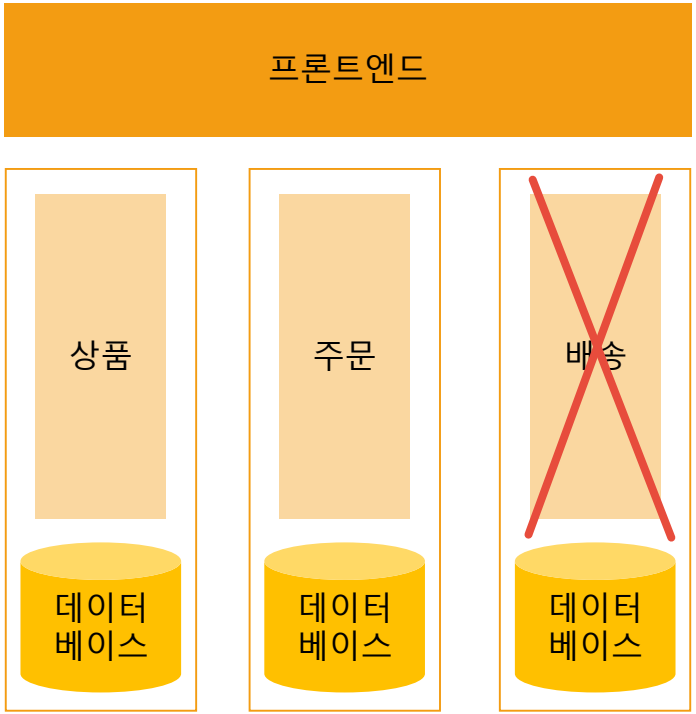

프론트엔드
상품
주문
배송
데이터베이스
모놀리식


프론트엔드
IAM
Gateway
상품
(Java/
Tomcat)
주문
(Java/
Spring)
배송
(.NET)
마케팅
(python)
RDB
RDB
RDB
No SQL
Message Channel

효과 – 장애 최소화

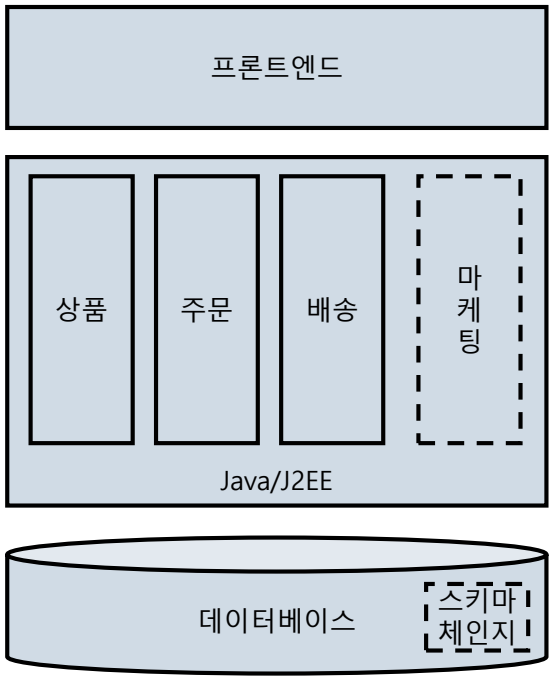


장애가 전파되며, 수동 복구로 장애 지연

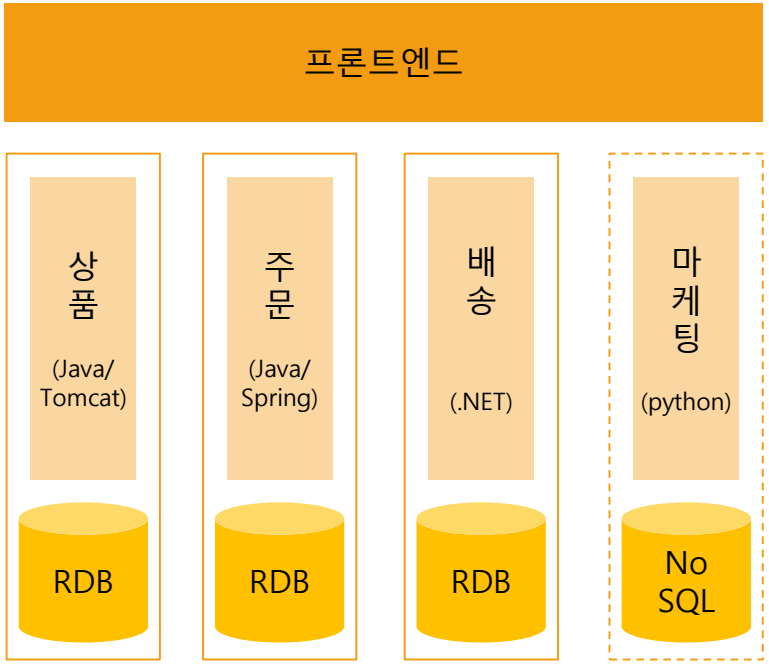


장애가 격리되며, 자동으로 복구됨(이전버전)

효과 – 기능 확장

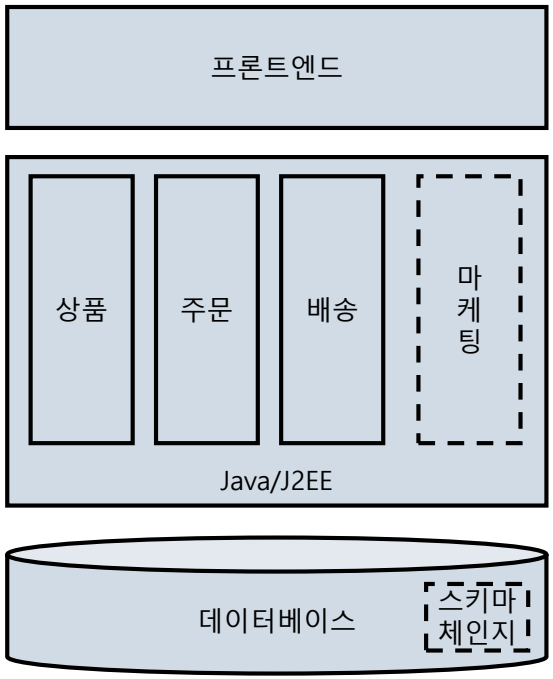


기존 코드의 수정 → Big Impact

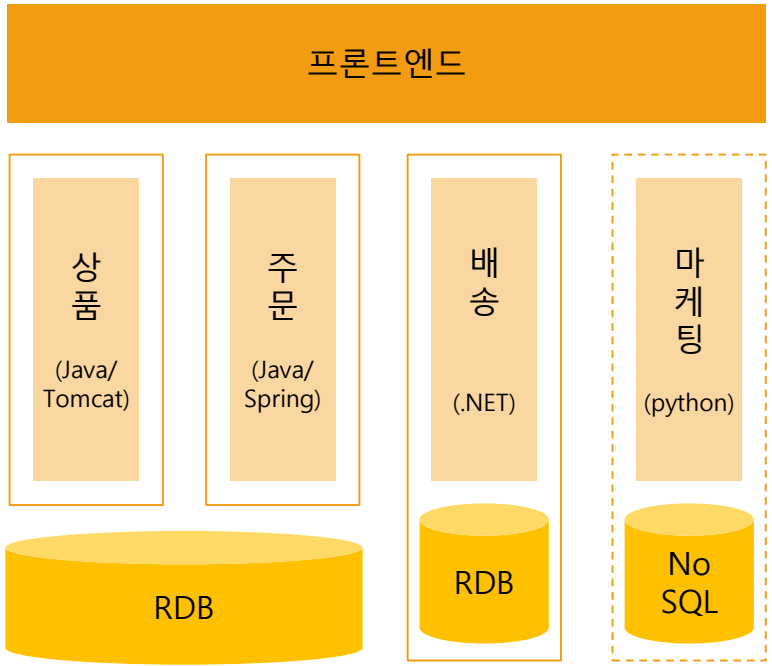


신규 마이크로서비스의 추가 → Minimal Impact
자율적 기술선택(플랫폼 & 데이터레이아웃)

효과 – 기능 확장

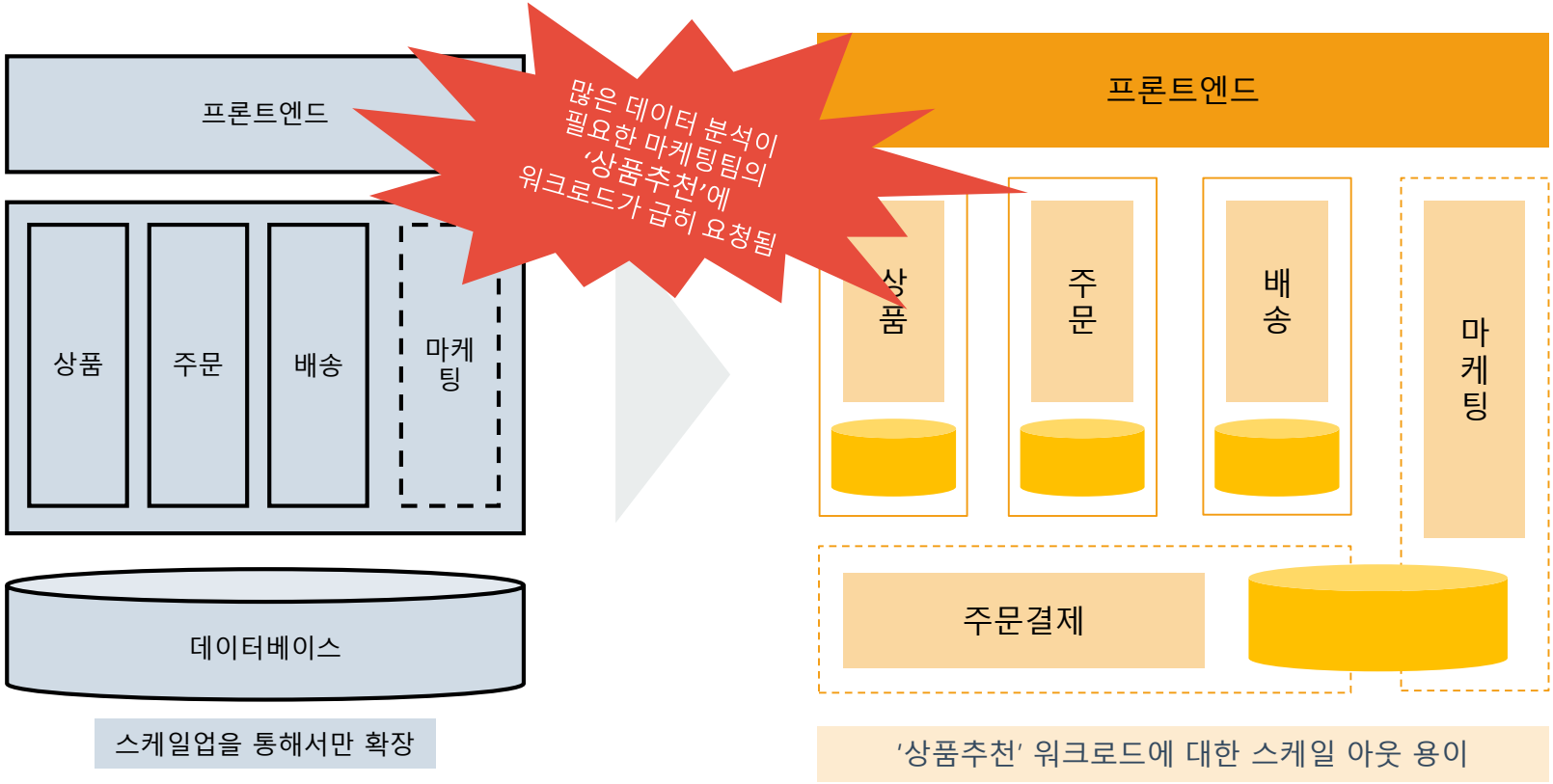


기존 코드의 수정 → Big Impact



신규 마이크로서비스의 추가 → Minimal Impact
자율적 기술선택(플랫폼 & 데이터레이아웃)

효과 – 성능 배분



스케일업을 통해서만 확장

'상품추천' 워크로드에 대한 스케일 아웃 용이

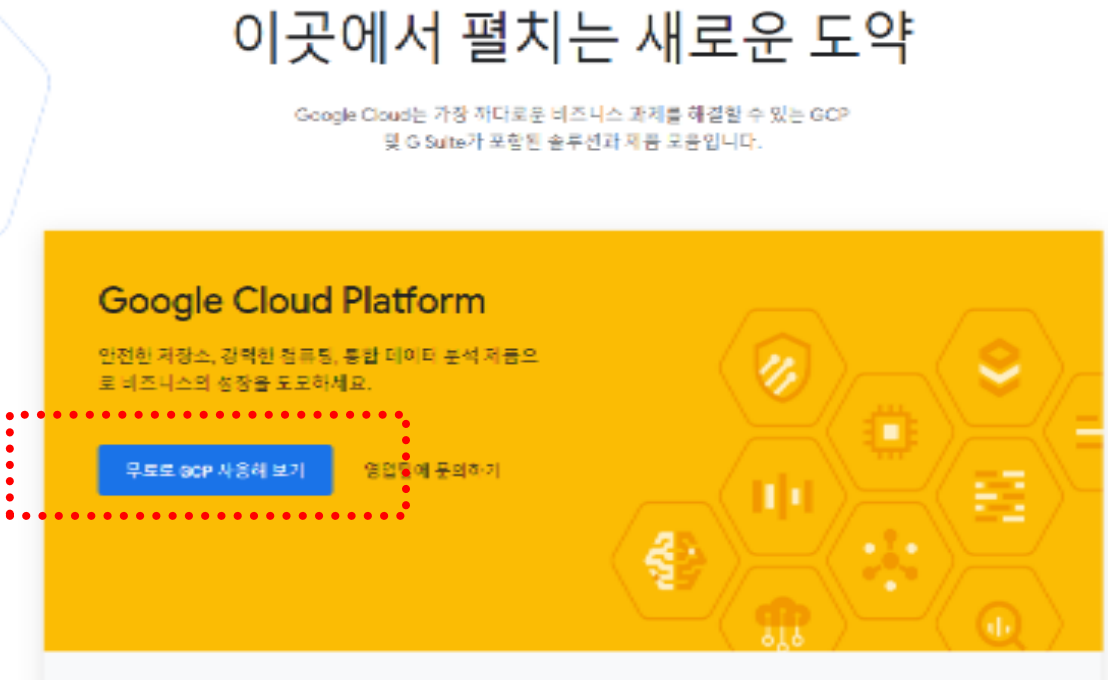
QUICK TOUR


환경설정
소스 저장소 연동
자동배포
무정지 버전업
하나 이상의 서비스 배포
간섭없는 서비스 추가
장애 격리
테스트
모니터링
종료

구글 클라우드 플랫폼(GCP) 시작

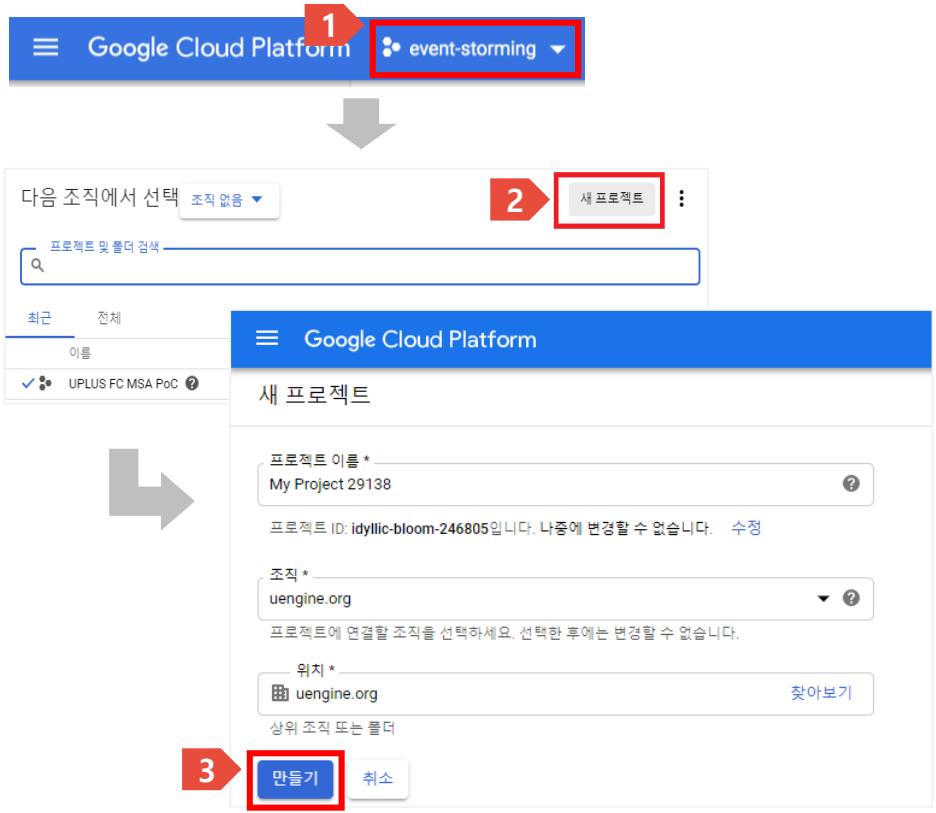
GCP 가입

http://cloud.google.com



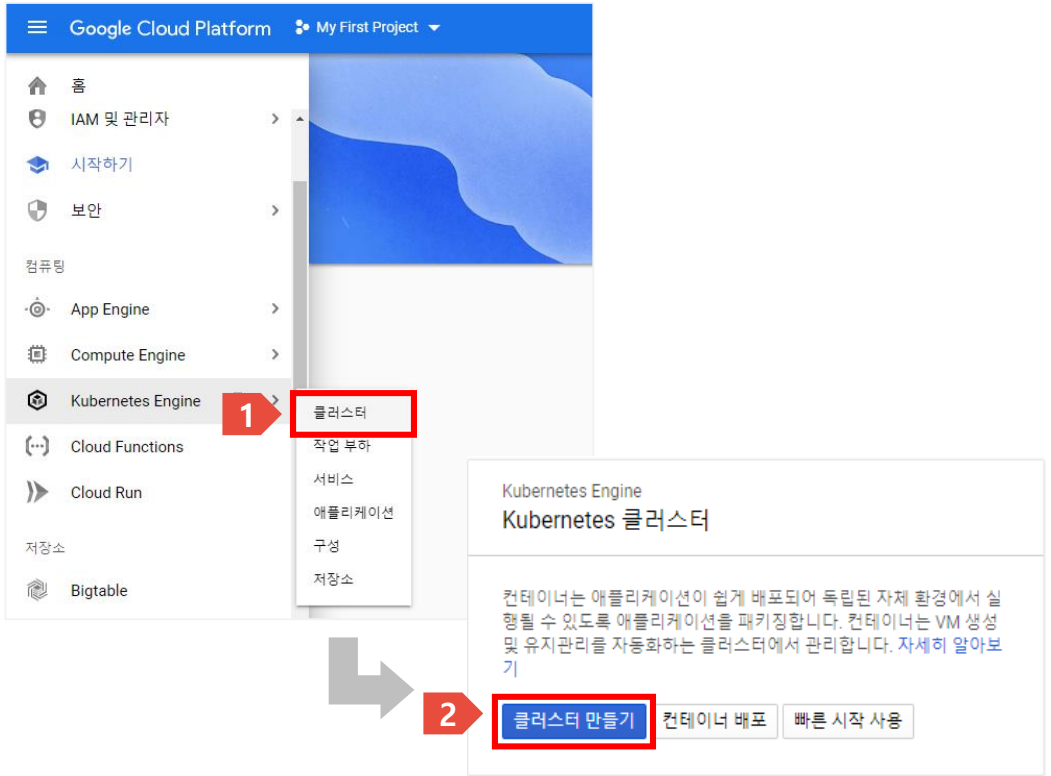
GCP 프로젝트 생성

1. 상단의 프로젝트 리스트를 누른다.
2. 팝업의 오른 상단의 새 프로젝트 버튼을 누른다.
3. 프로젝트 이름(event-storming), 조직, 위치 등을 설정하고 만들기 버튼을 클릭한다.



Kubernetes 클러스터 생성 (1/2)

1. 메뉴중 Kubenetes Engine의 클러스터를 선택한다.
2. 클러스터 만들기 버튼을 클릭한다.



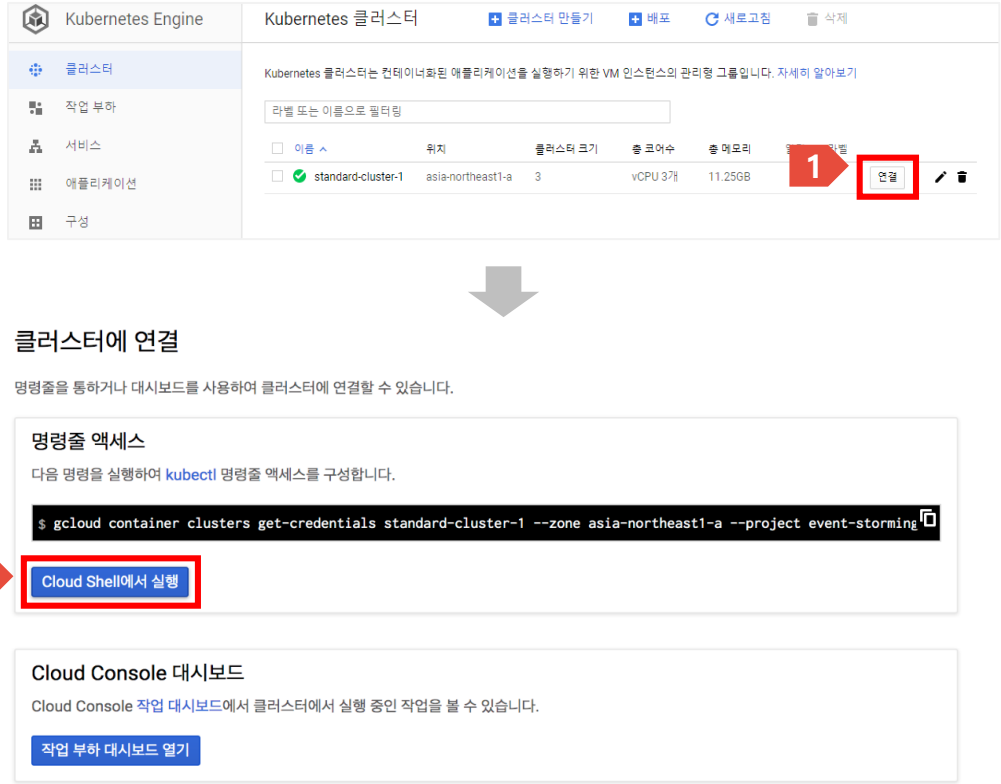
Kubernetes 클러스터 생성 (2/2)

3. 클러스터 세부 설정을 입력한다.
   - 리전(asia-northeast-a) 등
4. 하단의 만들기 버튼을 클릭한다.
5. 클러스터 리스트 화면이 표시된다.

클러스터	
이름	standard-cluster-1 (default)
위치유형	영역
영역	asia-northeast1-a
노드수	3

클러스터 접속 및 확인

1. 연결 버튼을 클릭해서 클러스터 접속
2. 팝업 창에서 Cloud Shell에서 실행 버튼 클릭



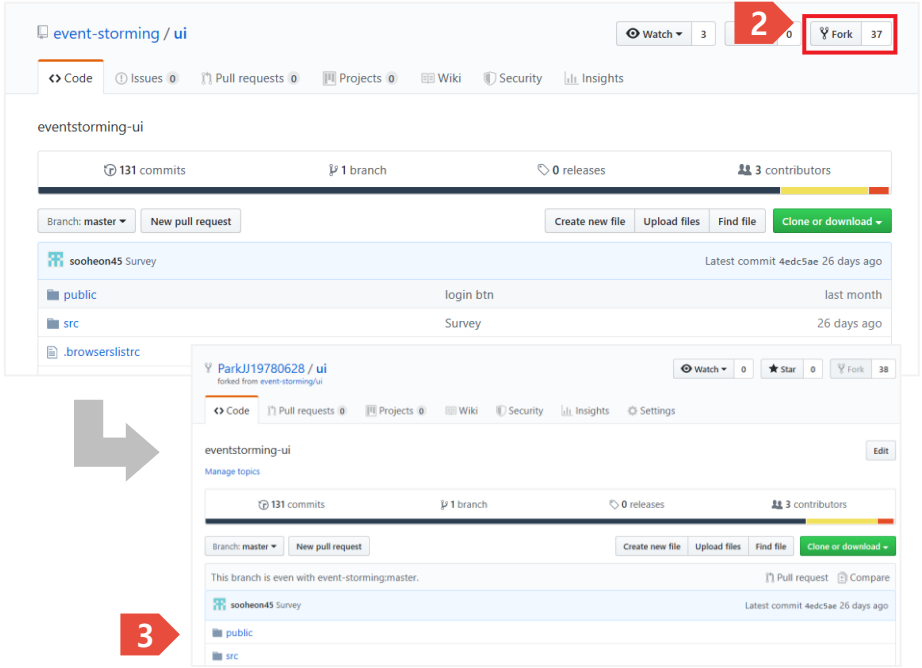
QUICK TOUR


환경설정
소스 저장소 연동
자동배포
무정지 버전업
하나 이상의 서비스 배포
간섭없는 서비스 추가
장애 격리
테스트
모니터링
종료

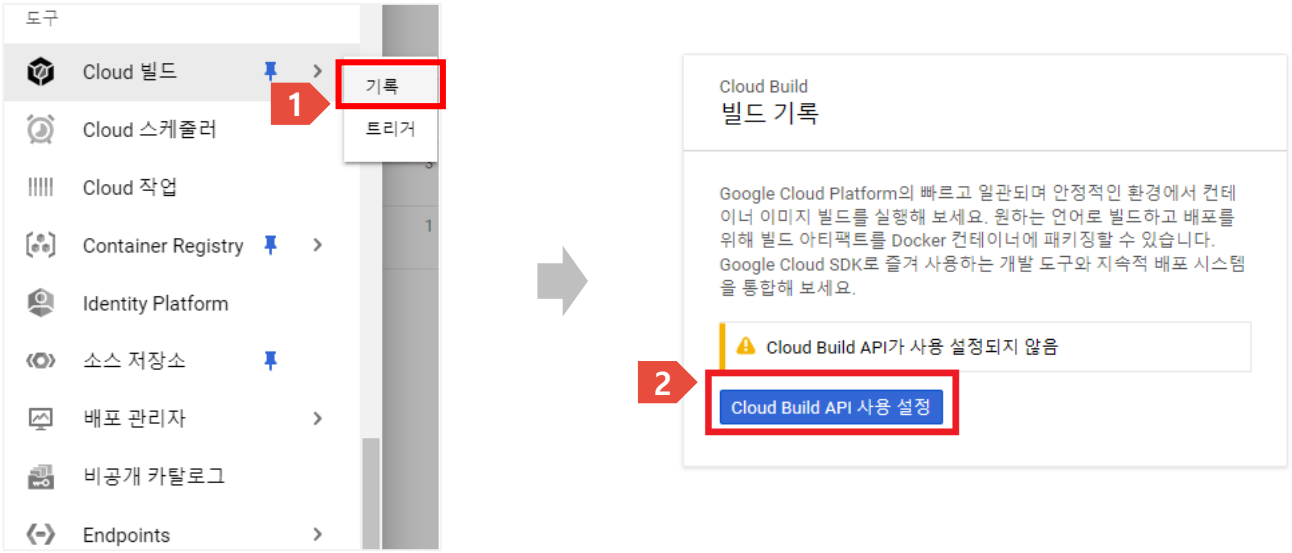
Github 로그인 및 실습 프로젝트 복제

1. https://github.com/event-storming/ui 에 접속
2. 화면 우측의 Fork 버튼 클릭
3. 소스 내용이 자신의 계정으로 clone
   - https://github.com/<my account>/ui.git

1 https://github.com/event-storming/ui

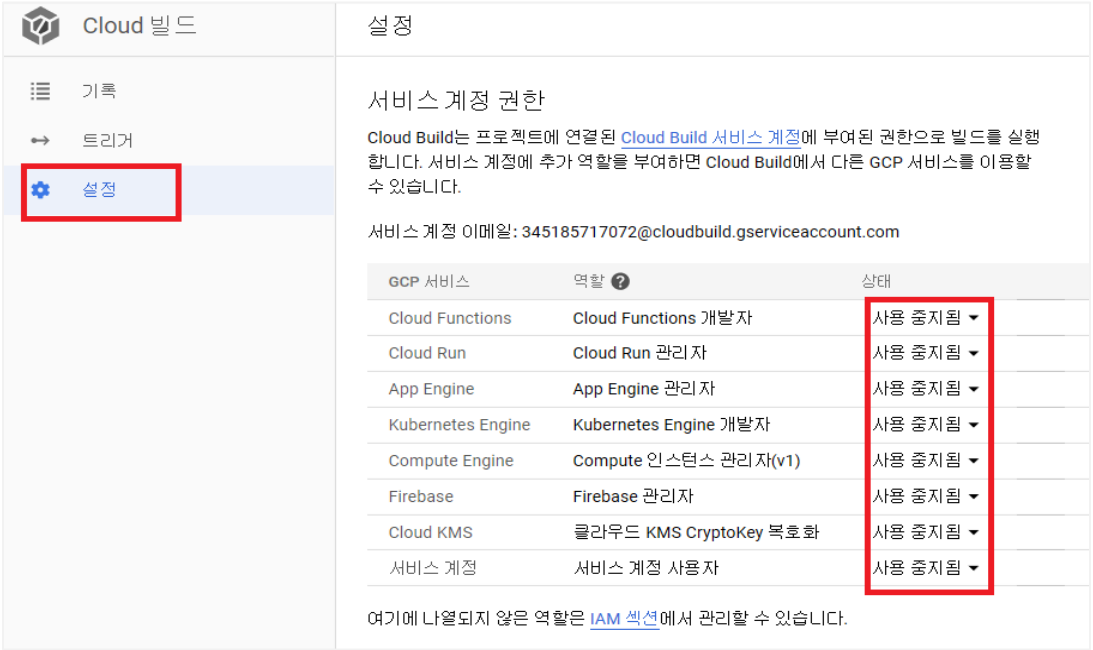


GCB(Google Cloud Build) 사용하기

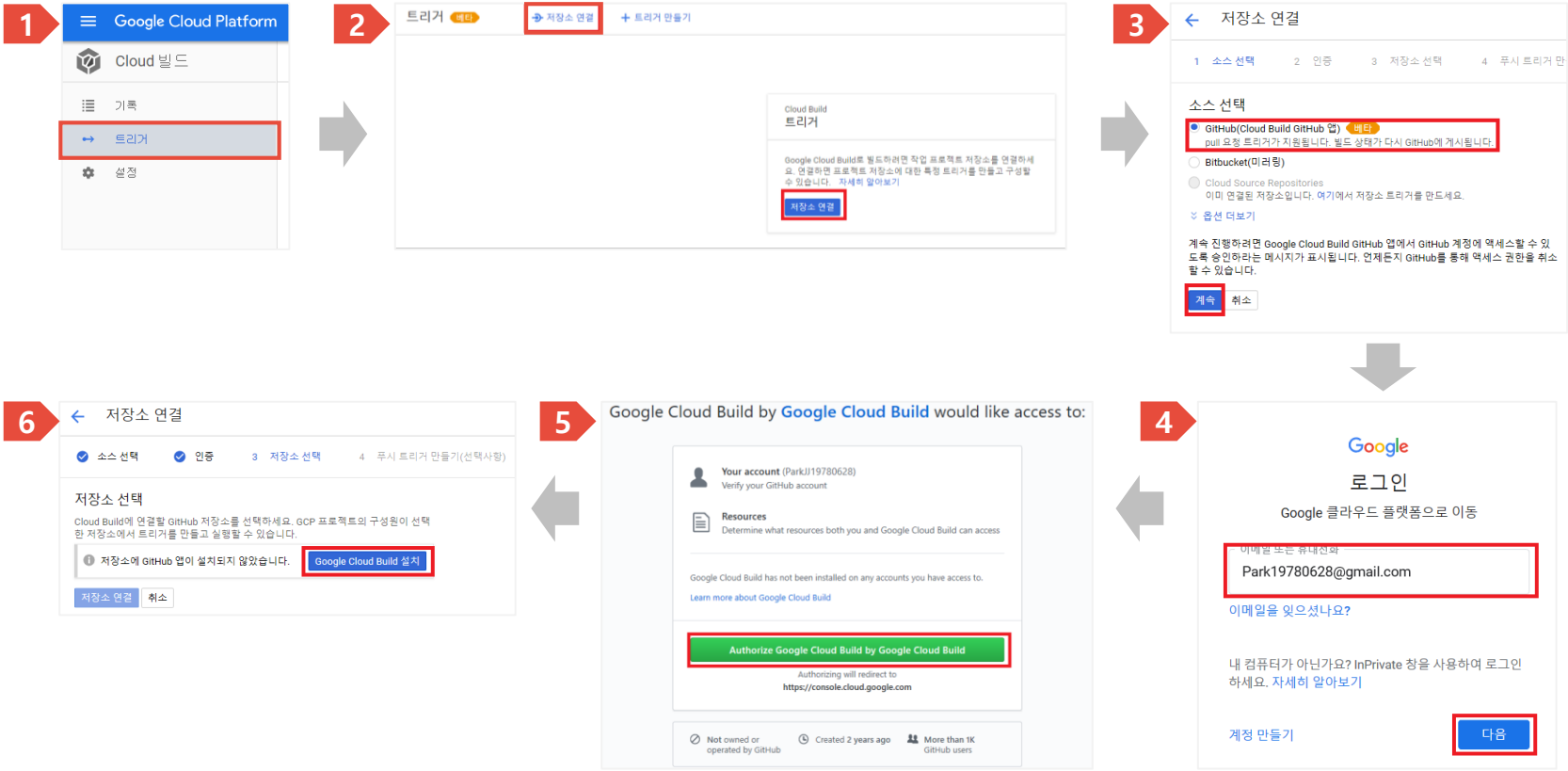


GCB가 쿠버네티스 클러스터를 접근하기 위한 권한 설정

Cloud 빌드 설정메뉴에서 모든 권한 부여 (사용설정으로 Set)

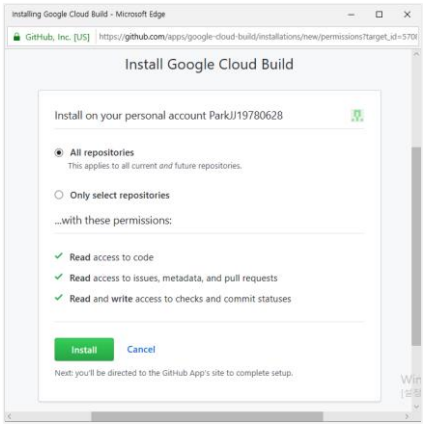


GitHub 연결 - SSO



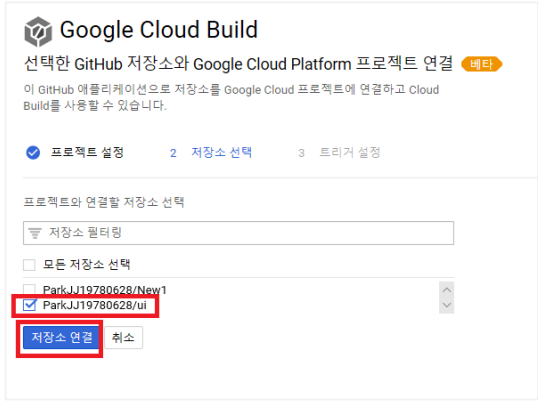
GitHub 연결 - 저장소 연결

1



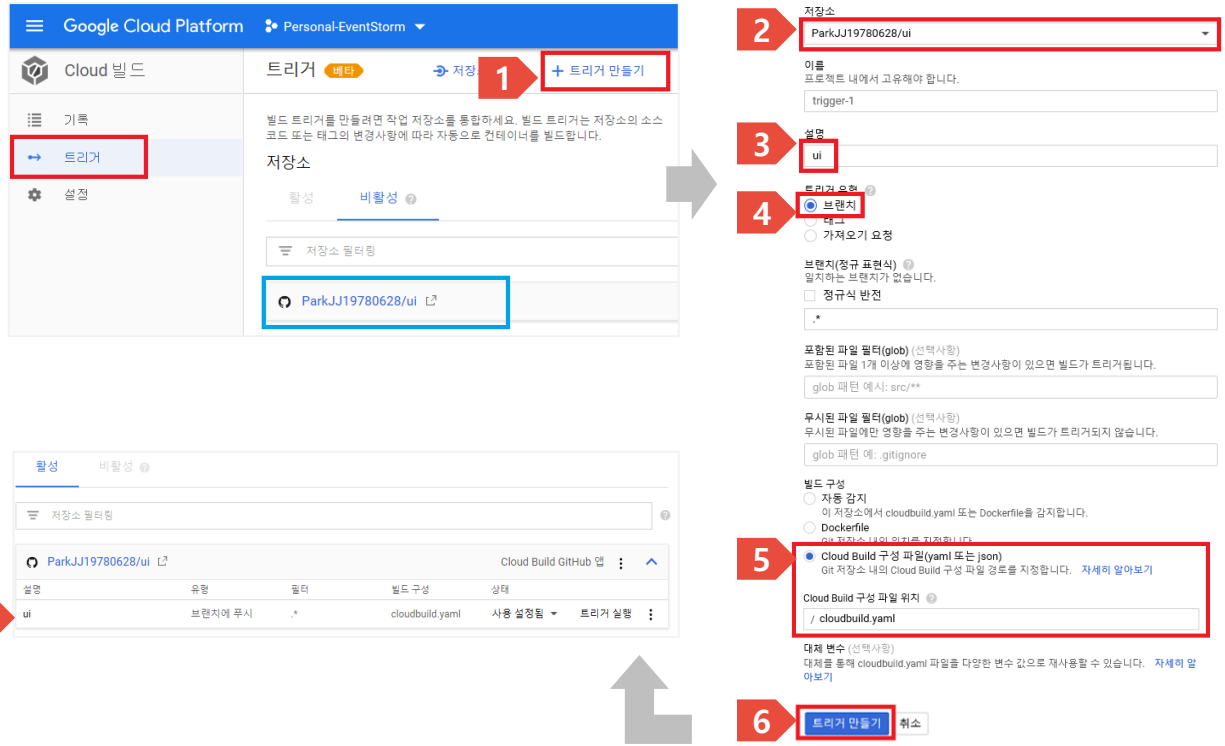
2

3



파이프라인(트리거) 만들기

1. 메뉴의 트리거 만들기 클릭
2. 저장소는 앞서 연결했던 ui repository 선택
3. 설명에 ui 입력
4. 트리거 유형에 브랜치 선택
5. 빌드 구성에 Cloud Build.yaml 구성파일 선택
6. 트리거 만들기 클릭
7. 생성된 트리거 확인



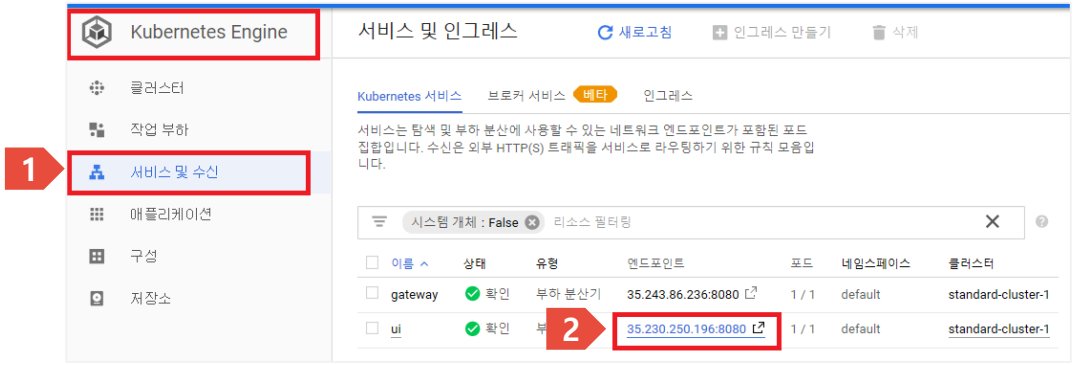
QUICK TOUR

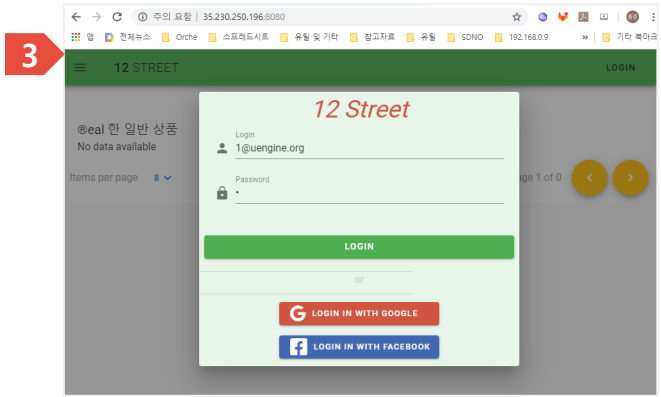

환경설정
소스 저장소 연동
자동배포
무정지 버전업
하나 이상의 서비스 배포
간섭없는 서비스 추가
장애 격리
테스트
모니터링
종료

생성된 트리거 실행 및 확인

UI 서비스 확인

1. 메뉴에서 kubernetes Engine의 서비스 및 수신 선택
2. 리스트 중 ui의 엔드포인트 클릭
3. 브라우저에서 쇼핑몰 화면 확인





Tips: 빌드 성공 및 실패 시 확인 방법

1

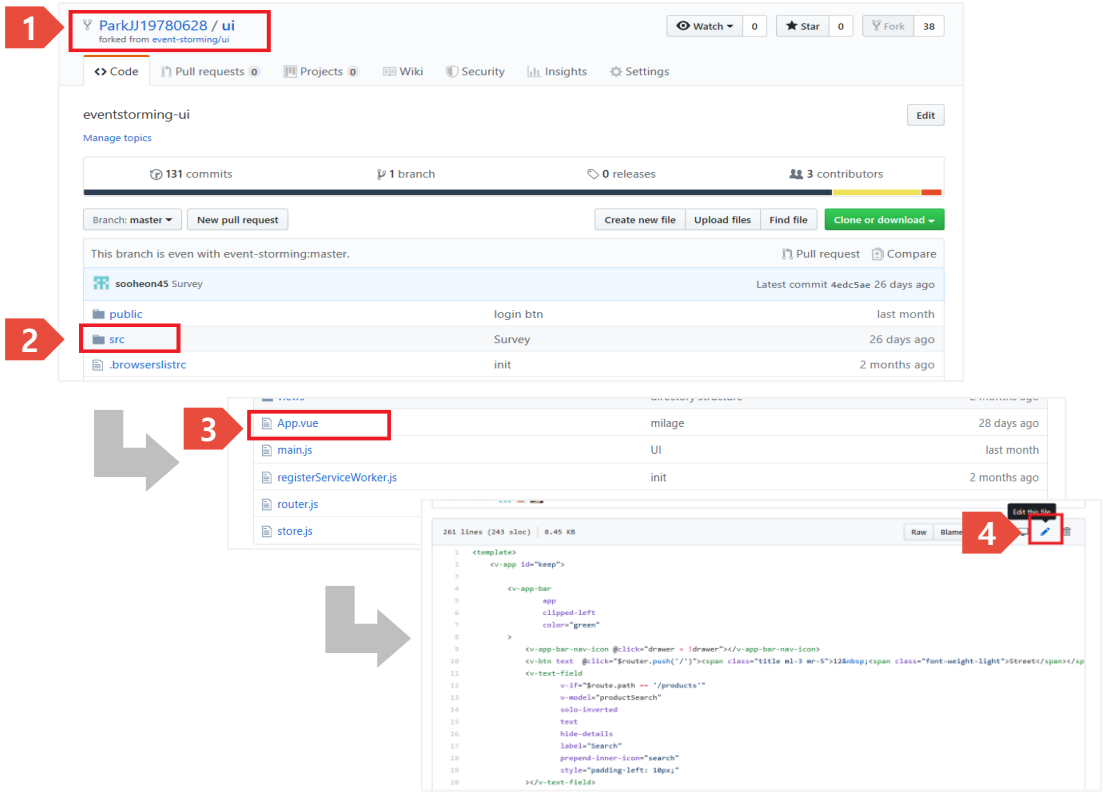
2

QUICK TOUR


환경설정
소스 저장소 연동
자동배포
무정지 버전업
하나 이상의 서비스 배포
간섭없는 서비스 추가
장애 격리
테스트
모니터링
종료

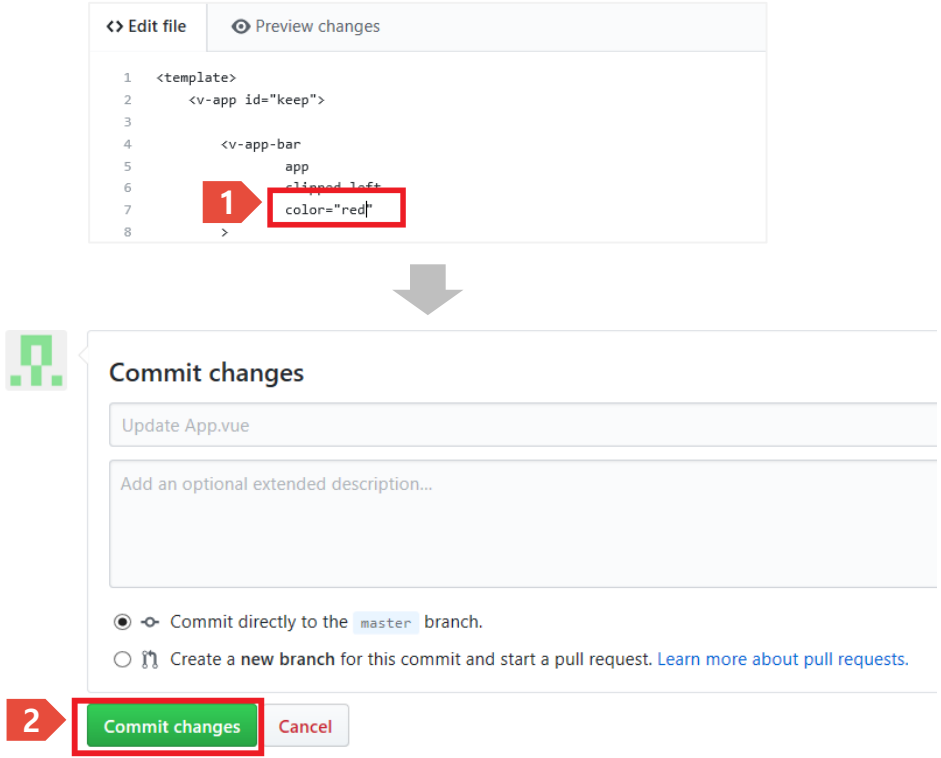
무정지 배포 실습 - 코드 변경 (1/2)

1. 자신의 github ui repository로 이동
2. src 클릭
3. App.vue 클릭
4. 우측의 연필 모양 아이콘 클릭

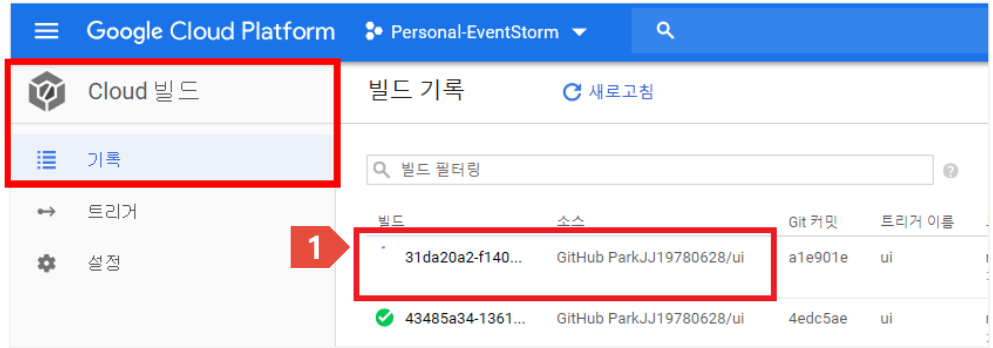


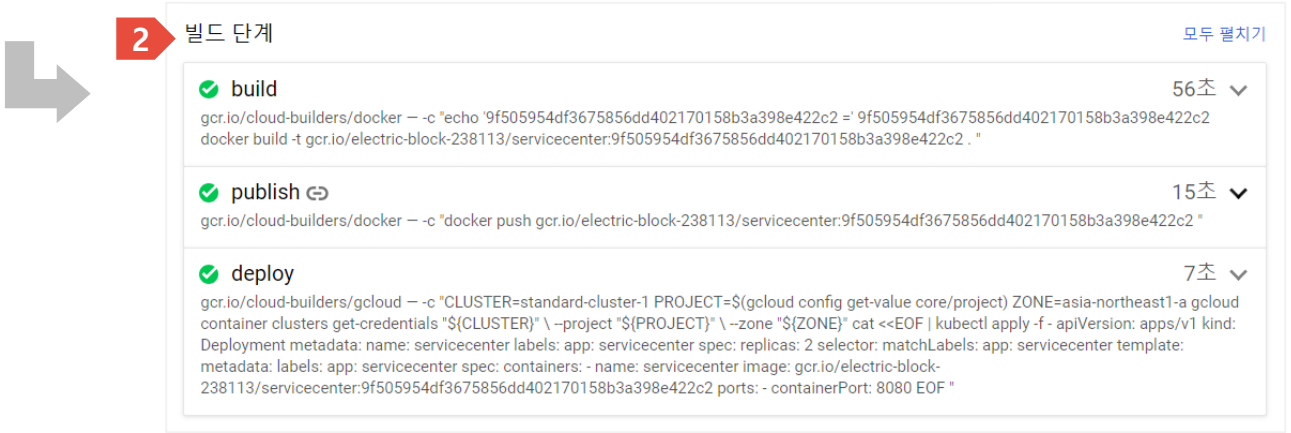
무정지 배포 실습 - 코드 변경 (2/2)

1. 소스 내용중 color="green"을 color="red"로 수정
2. 페이지 하단의 Commit changes 버튼 클릭

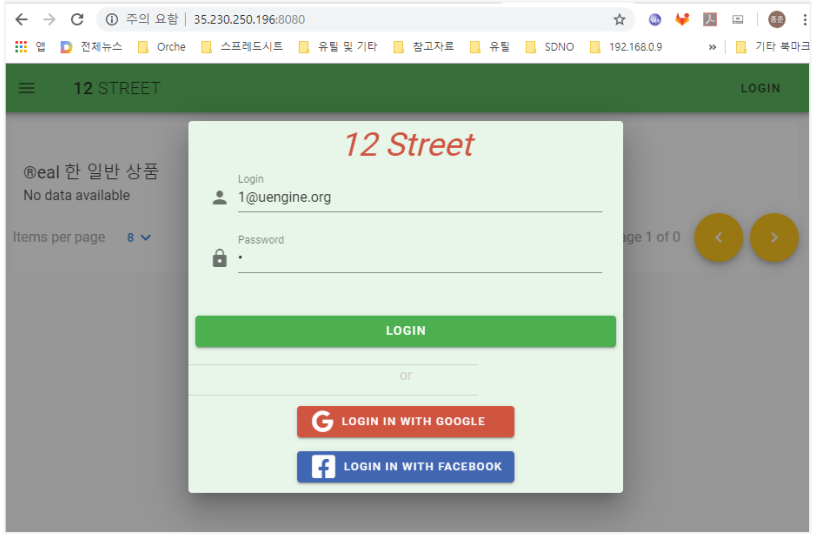


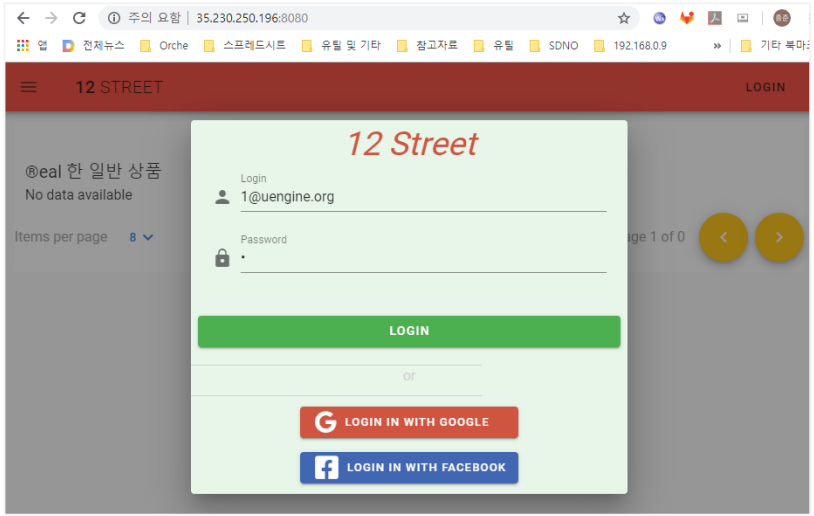
무정지 배포 실습 – 빌드 자동화 및 배포 확인





무정지 배포 실습 – 변경내용 확인





QUICK TOUR


환경설정
소스 저장소 연동
자동배포
무정지 버전업
하나 이상의 서비스 배포
간섭없는 서비스 추가
장애 격리
테스트
모니터링
종료

Micro-Service 간 통신을 위한 메세지 큐 설치

1. Cloud Shell에 연결
2. Shell에서 아래 스크립트 실행

   참고 : 실습스크립트 룩업 (https://workflowy.com/s/msa/27a0ioMCzlpV04Ib)중, '카프카서버 설치' 로 검색

```
curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get | bash
kubectl --namespace kube-system create sa tiller
kubectl create clusterrolebinding tiller --clusterrole cluster-admin --serviceaccount=kube-system:tiller
helm init --service-account tiller

kubectl patch deploy --namespace kube-system tiller-deploy -p '{"spec":{"template":{"spec":{"serviceAccount":"tiller"}}}}'

helm repo add incubator http://storage.googleapis.com/kubernetes-charts-incubator
helm repo update

helm install --name my-kafka --namespace kafka incubator/kafka
```

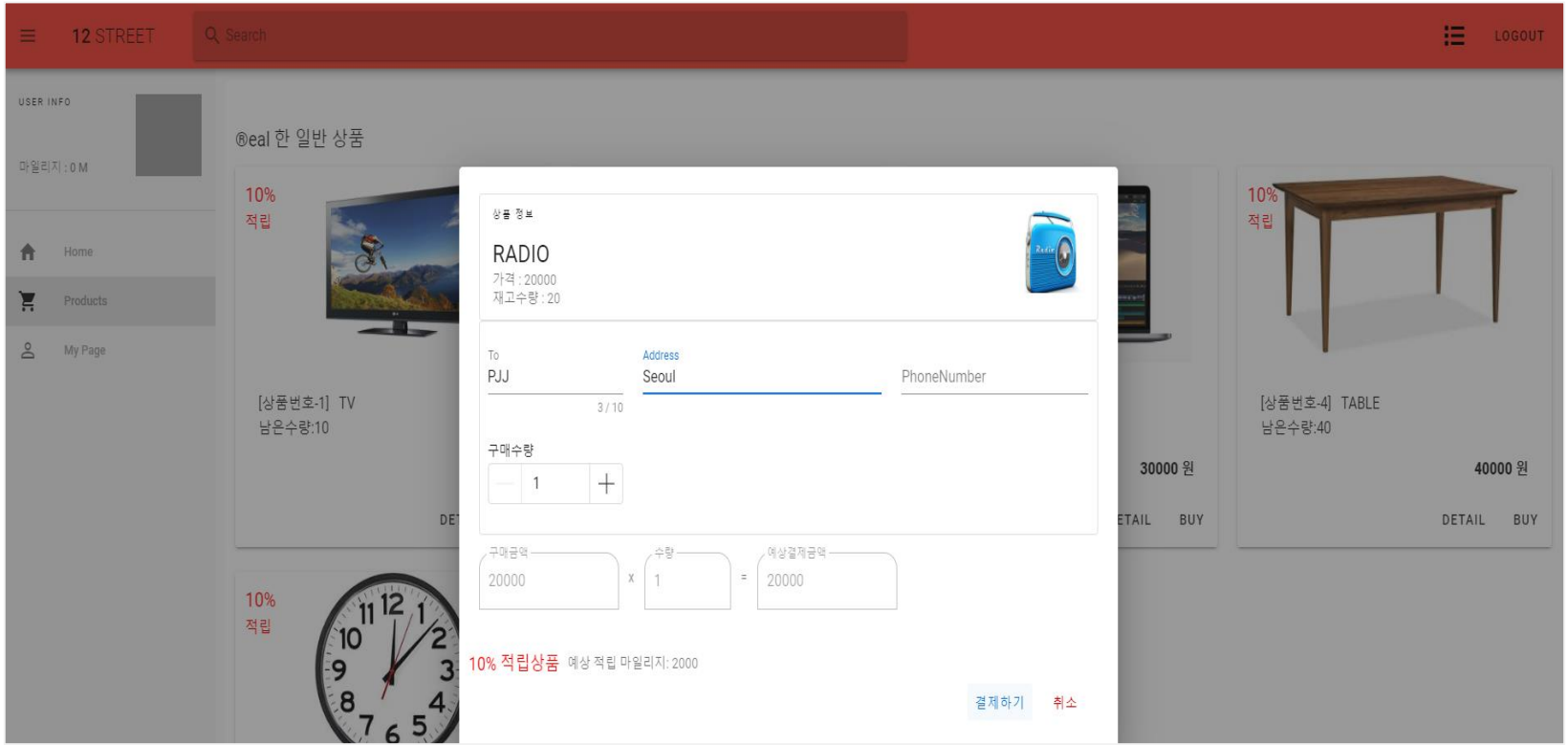
아래 서비스 들에 대하여 저장소연결과 트리거 생성

- orders
- products
- delivery
- oauth
- Gateway

UI 서비스에서 설치된 Gateway 인식하기

- UI 서비스 트리거 재실행
- Browser Cache 삭제

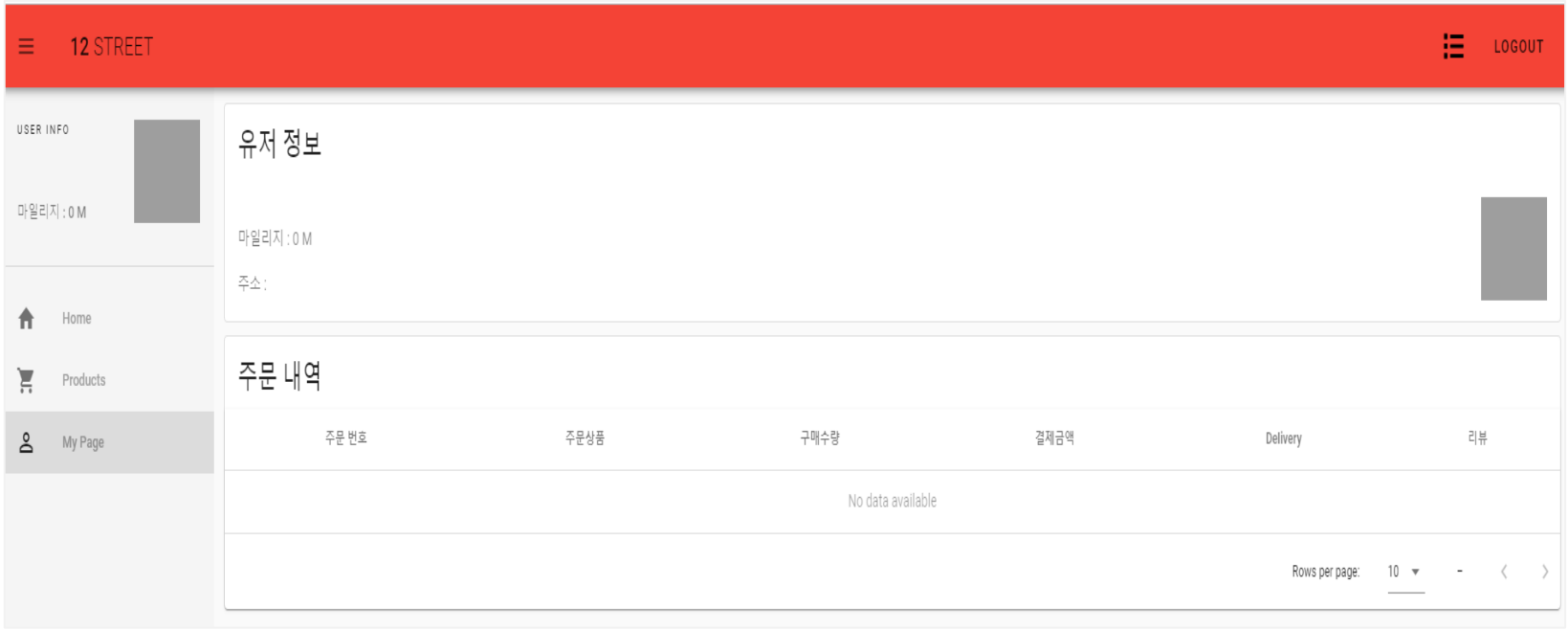
쇼핑몰 재접속 후 기능사용하기(주문, 상품추가, mypage 확인)



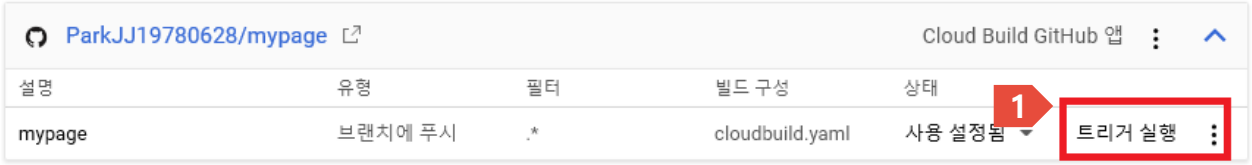
QUICK TOUR

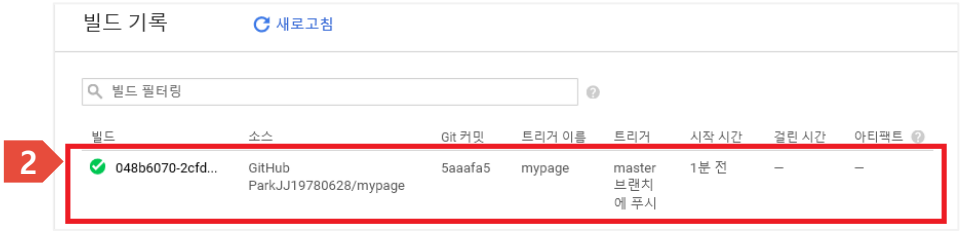

환경설정
소스 저장소 연동
자동배포
무정지 버전업
하나 이상의 서비스 배포
간섭없는 서비스 추가
장애 격리
테스트
모니터링
종료

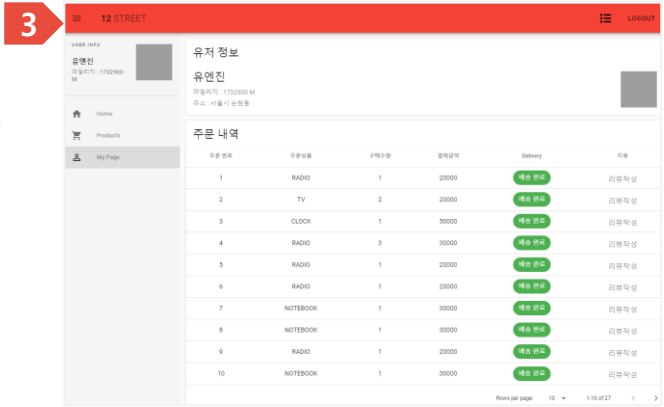
새로운 서비스추가 실습 - My Page 확인



새로운 서비스 추가 실습 - My Page 소스 Fork, 트리거 생성 및 UI 확인



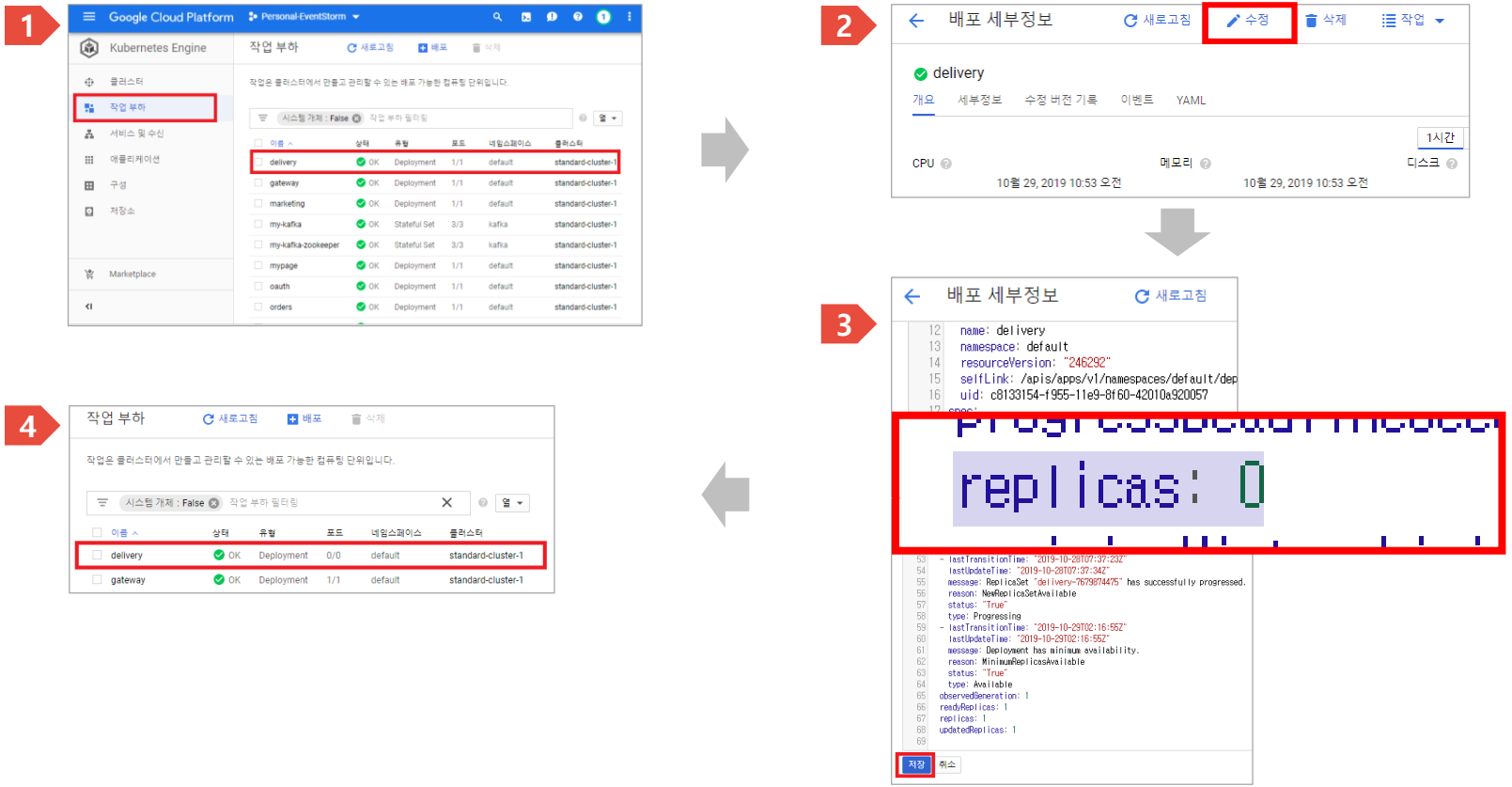




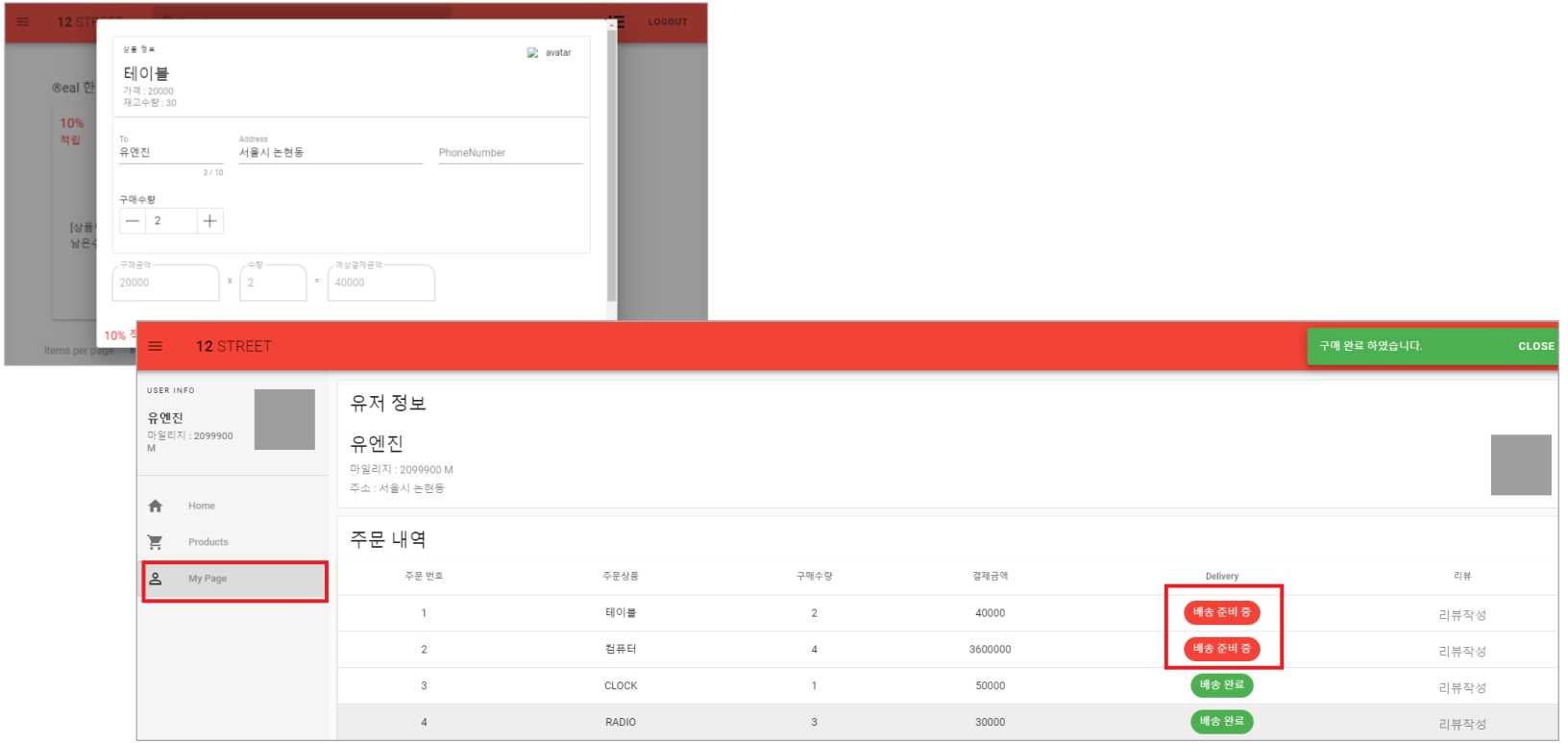
QUICK TOUR


환경설정
소스 저장소 연동
자동배포
무정지 버전업
하나 이상의 서비스 배포
간섭없는 서비스 추가
장애 격리
테스트
모니터링
종료

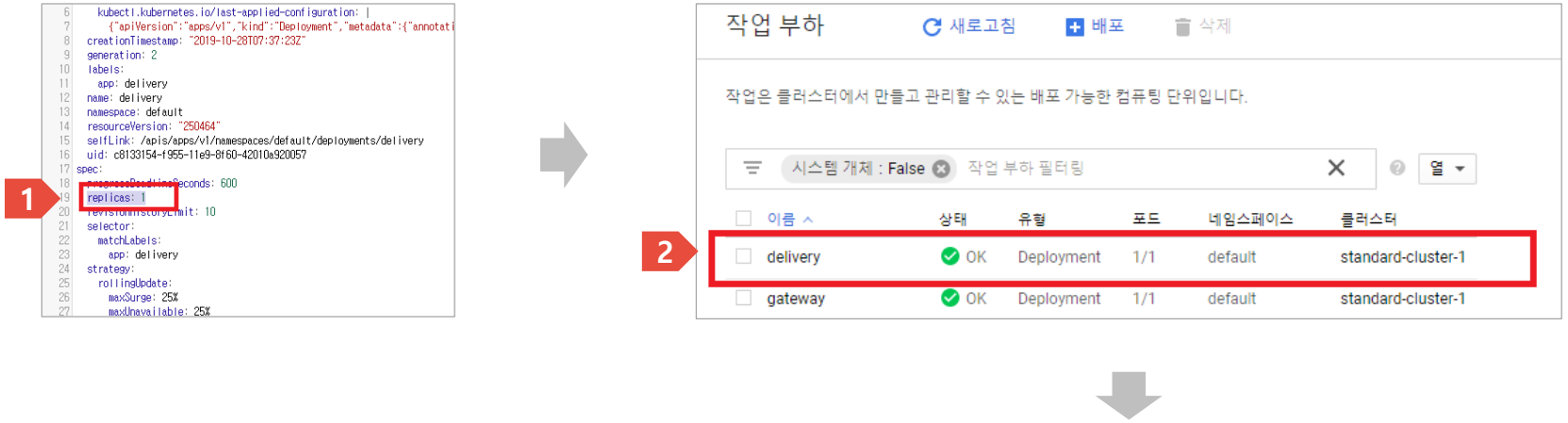
Delivery 서비스 정지



쇼핑 후 My Page 확인 – 배송처리되지 않음



Delivery 서비스 정상화 후 일괄 처리됨



주문 내역

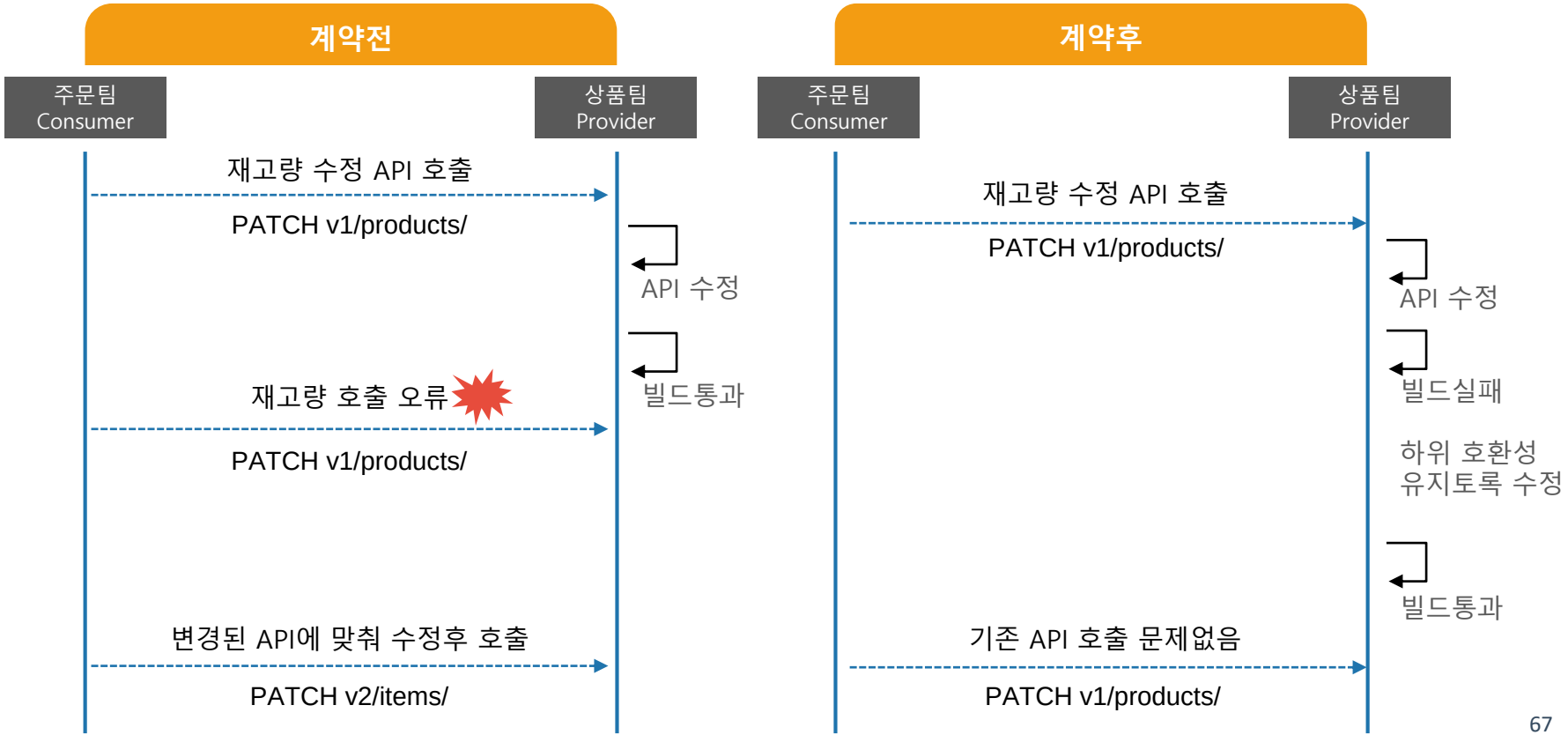
주문 번호	주문상품	구매수량	결제금액	Delivery	리뷰
1	테이블	2	40000	배송 완료	리뷰작성
2	컴퓨터	4	3600000	배송 완료	리뷰작성

3

QUICK TOUR


환경설정
소스 저장소 연동
자동배포
무정지 버전업
하나 이상의 서비스 배포
간섭없는 서비스 추가
장애 격리
테스트
모니터링
종료

Contract Test



상품팀에서 API를 일방적으로 변경

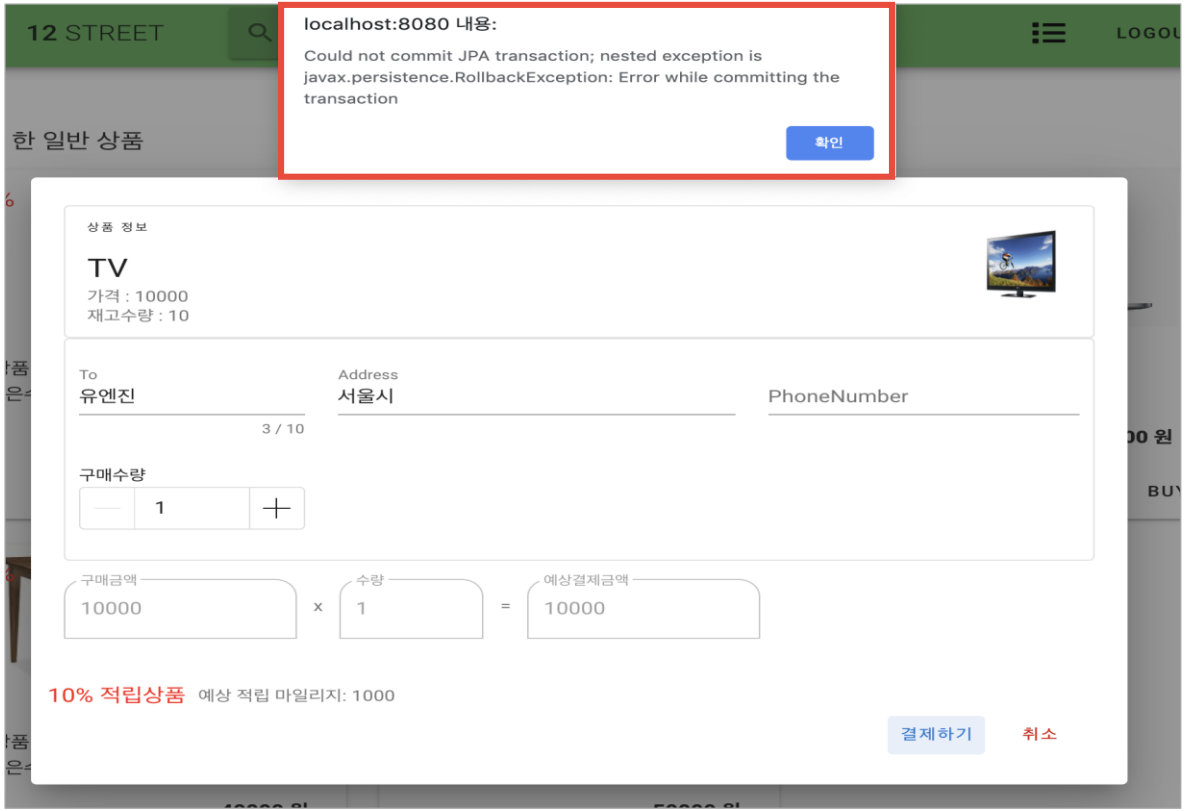
소스 위치

products / src / main / java / com / example / template / ProductController.java

```
13
14      @GetMapping("/product/{productId}")
15      Product productStockCheck(@PathVariable(value = "productId") Long productId) {
16          return  this.productService.getProductById(productId);
17      }
18  }
19
```

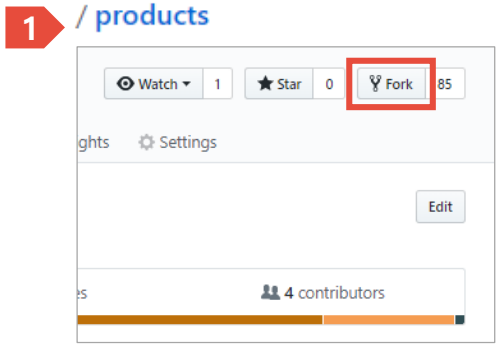
/item/{productId} 로 변경

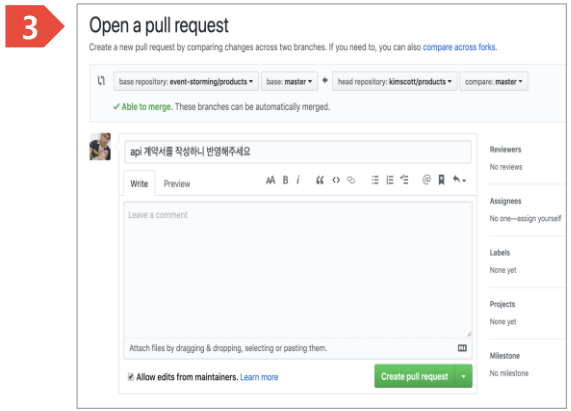
주문팀의 서비스 장애 발생



계약체결(1/2) - 주문팀에서 계약서 작성 후, 상품팀에 체결 요청

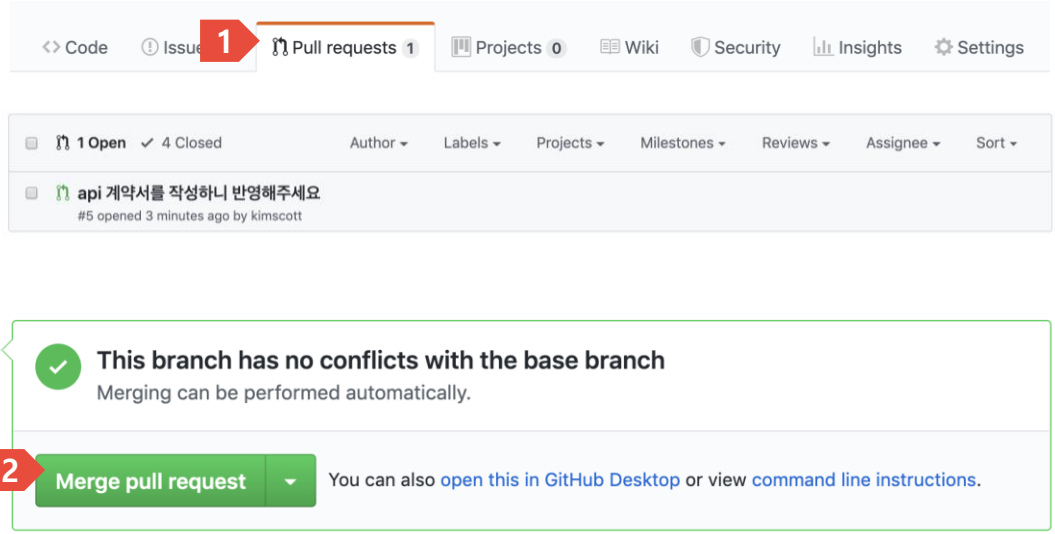
1. 상품팀 소스 복사 (포크 생성)
2. 주문팀이 계약서 생성
3. 주문팀에서 생성한 계약서 (productGet.groovy) 파일을 상품팀에 Pull request 요청함





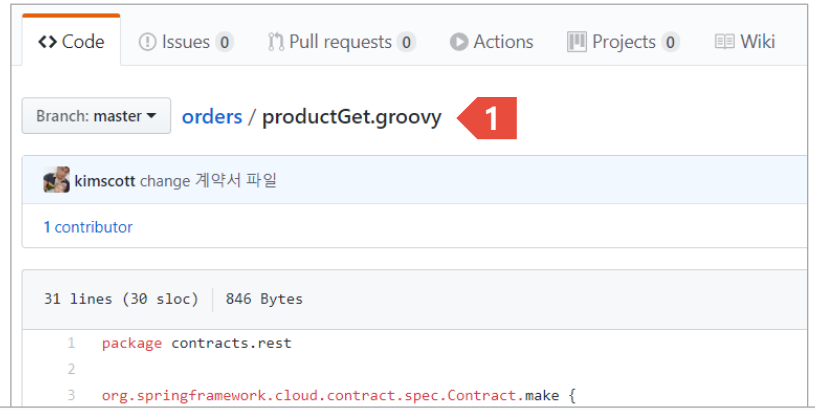
계약체결(2/2) - 상품팀에서 계약서 수락

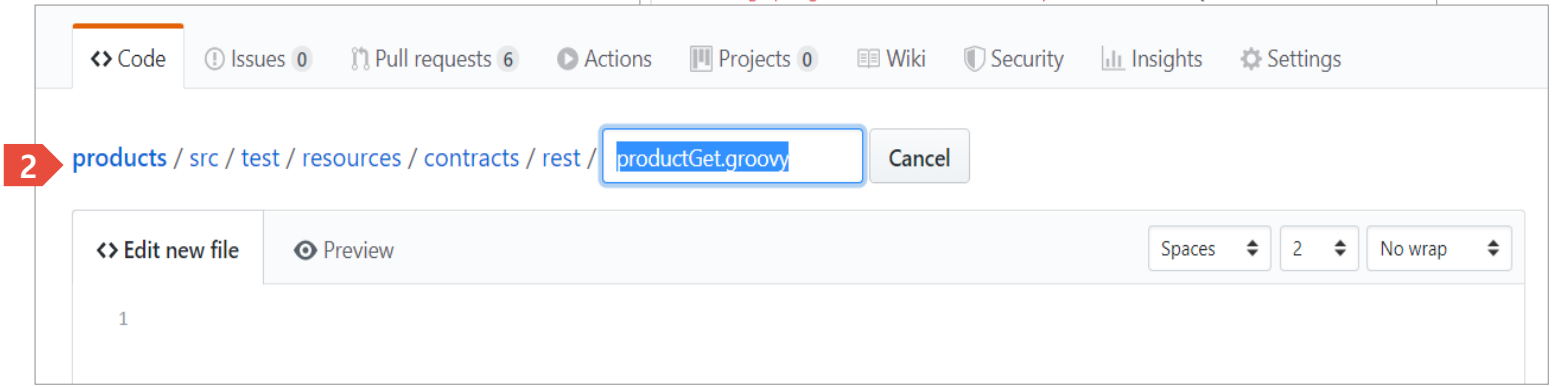
- 상품팀 : Pull Request 메뉴 선택
- 상품팀은 해당 계약서를 accept 하여 반영함
- 상품 서비스는 이제부터는 계약서를 안지켰을때 아예 배포가 안됨



계약체결 WORKAROUND

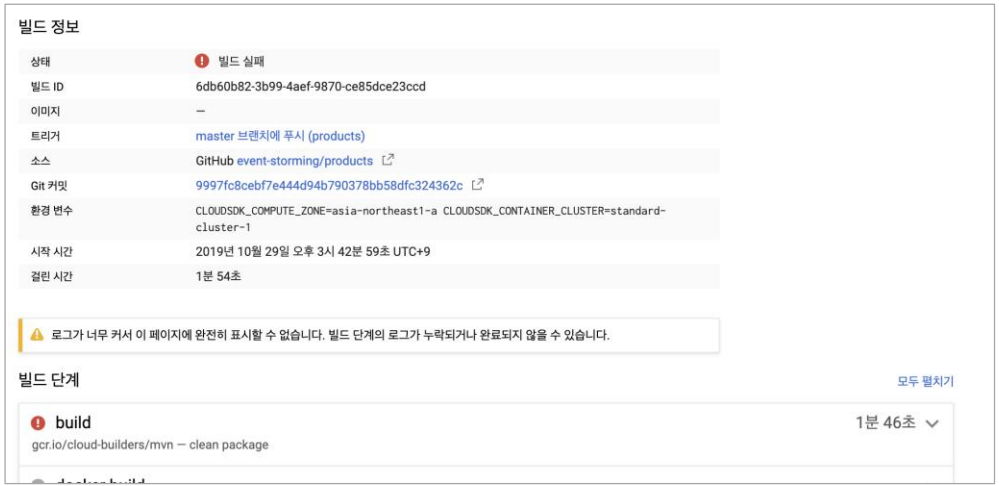
1. 주문팀에서 계약서 내용 복사
2. 상품팀에 계약서 붙여 넣기
3. Commit





계약체결 후, 상품팀은 계약 위반으로 배포 실패함

CloudBuild 에서 mvn package 단계 실패



실패 LOG

```
Step #0 - "build": 2019-10-29 06:44:51.547 DEBUG 69 --- [           main] o.s.c.c.v.m.stream.StreamStubMessages    : Picked channel name is [event-out]
Step #0 - "build": [ERROR] Tests run: 1, Failures: 0, Errors: 1, Skipped: 0, Time elapsed: 4.976 s <<< FAILURE! - in com.example.template.MessagingTest
Step #0 - "build": [ERROR] validate_productChanged(com.example.template.MessagingTest)  Time elapsed: 0.221 s  <<< ERROR!
Step #0 - "build": com.jayway.jsonpath.PathNotFoundException: No results for path: $['productName']
Step #0 - "build":      at com.example.template.MessagingTest.validate_productChanged(MessagingTest.java:43)
Step #0 - "build":
```

상품팀에서는 하위 호환성을 유지하며, 추가 API 제공

products / src / main / java / com / example / template / ProductController.java

```
@GetMapping("/item/{productId}")
Product productStockCheck(@PathVariable(value = "productId") Long productId) {
    return  this.productService.getProductById(productId);
}

 @GetMapping("/product/{productId}")
 Product productStockCheck1(@PathVariable(value = "productId") Long productId) {
     return  this.productService.getProductById(productId);
 }
```

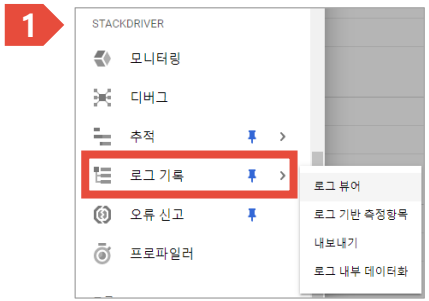
기존의 하위 호환성 API 유지

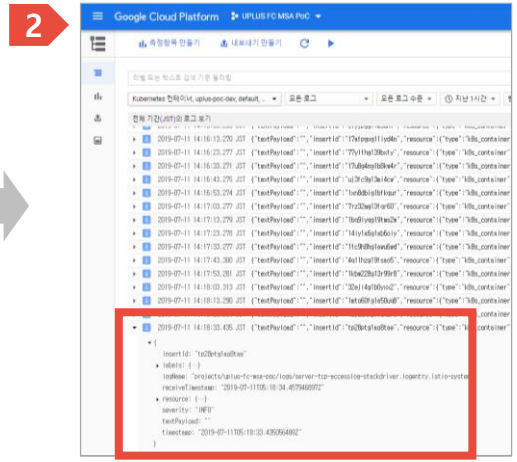
QUICK TOUR

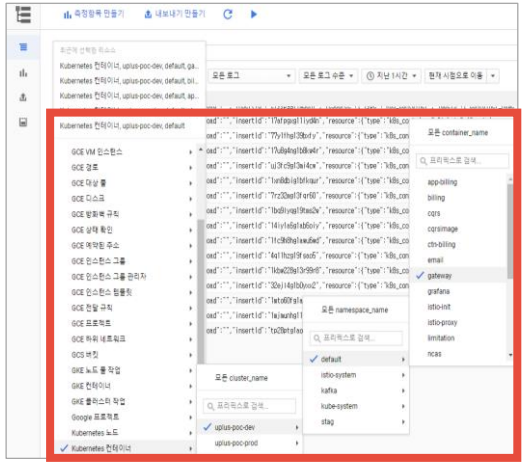

환경설정
소스 저장소 연동
자동배포
무정지 버전업
하나 이상의 서비스 배포
간섭없는 서비스 추가
장애 격리
테스트
모니터링
종료

Stackdriver Logging 확인

1. 메뉴 중 로그기록을 클릭한다.
2. 로그기록 리스트중 하나를 선택한다.
3. 로그에 대한 필터링 설정 예시
   - kubernetes 컨테이너 > uplus-poc-dev > default > gateway

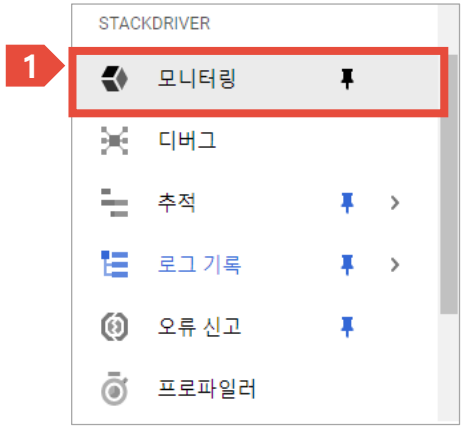


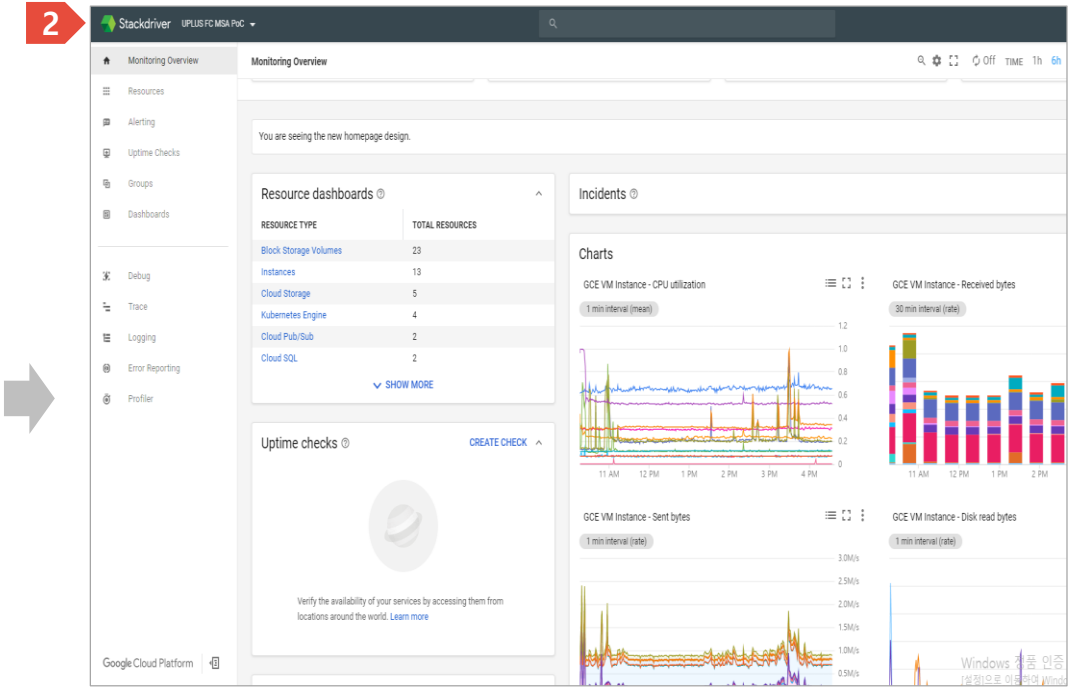




Stackdriver Monitoring 확인 (1/2)

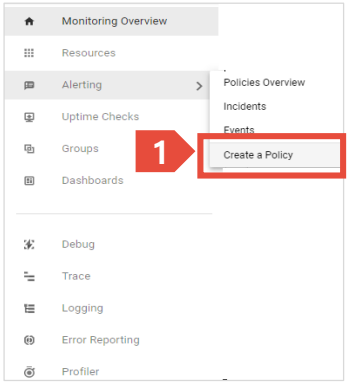
1. 메뉴 중 모니터링을 클릭한다.
2. 모니터링 화면이 표시된다.

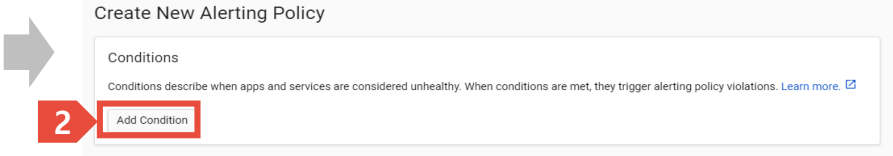


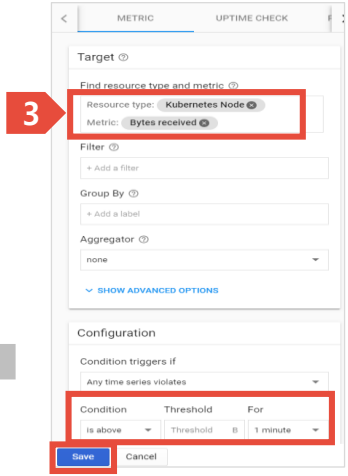


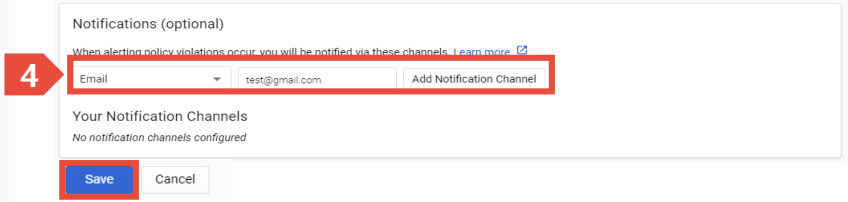
Stackdriver Monitoring 확인 (2/2)

1. 메뉴 중 Alerting > Create Policy 클릭한다.
2. 알람 받을 조건을 추가하기 위해 Add Condition 버튼을 클릭한다.
3. 조건에 대한 내용을 입력하고 Save 버튼을 클릭한다.
4. 알람 받을 메일 계정을 입력후 하단의 Save 버튼을 클릭한다.
   - 현재는 Mail발송 기능만 제공됨









Stackdriver Tracing 사용 설정 - pom.xml, Dockerfile

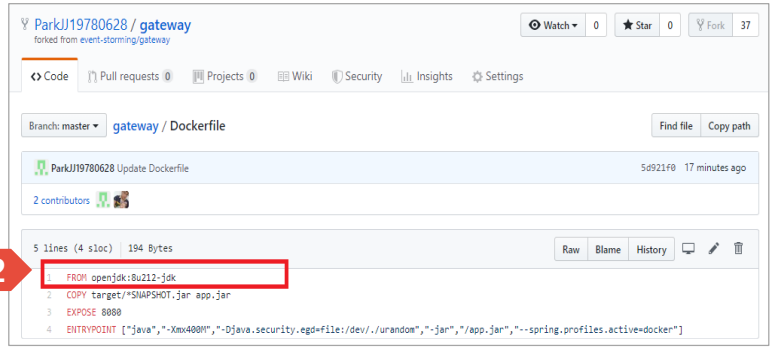
1. gateway repository 소스 수정

2. Pom.xml 내용 추가
   <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-gcp-starter-trace</artifactId>
   </dependency>

3. Dockerfile 내용 수정
   FROM openjdk:8u212-jdk-alpine
   → FROM openjdk:8u212-jdk

```
<dependencyManagement>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.cloud</groupId>
                        <artifactId>spring-cloud-dependencies</artifactId>
                        <version>${spring-cloud.version}</version>
                        <type>pom</type>
                        <scope>import</scope>
                </dependency>
                <dependency>
                        <groupId>org.springframework.cloud</groupId>
                        <artifactId>spring-cloud-gcp-starter-trace</artifactId>
                </dependency>
        </dependencies>
</dependencyManagement>
```

1



2

Stackdriver Tracing 확인

1

2

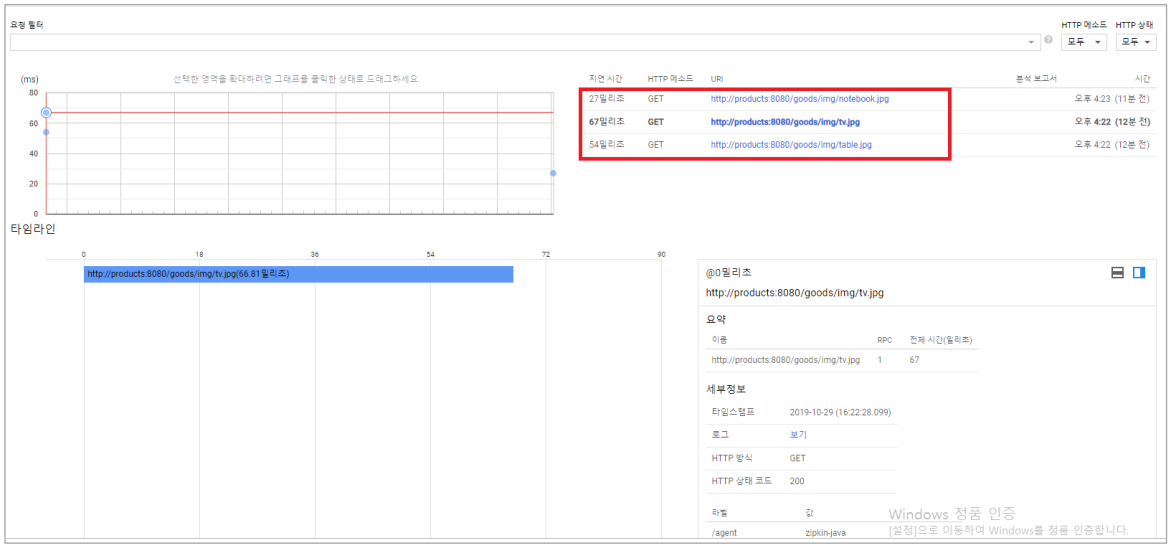
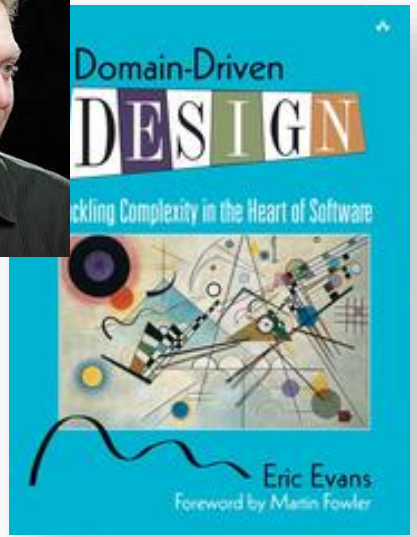


Table of content

DDD Patterns

Domain-Driven Design & MSA

DDD for MSA



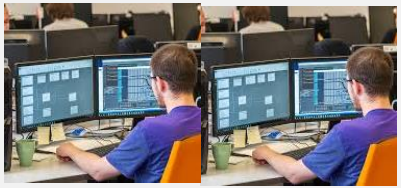
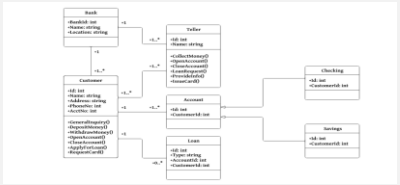
- **Bounded Context 와 Ubiquitous Language**
  → 어떤 단위로 마이크로 서비스를 쪼개면 좋은가?
- **Context Mapping**
  → 서비스를 어떻게 결합할 것인가?
- **Domain Events**
  → 어떤 비즈니스 이벤트에 의하여 마이크로 서비스들이 상호 반응하는가?

The key to controlling complexity is a good domain model
– Martin Fowler

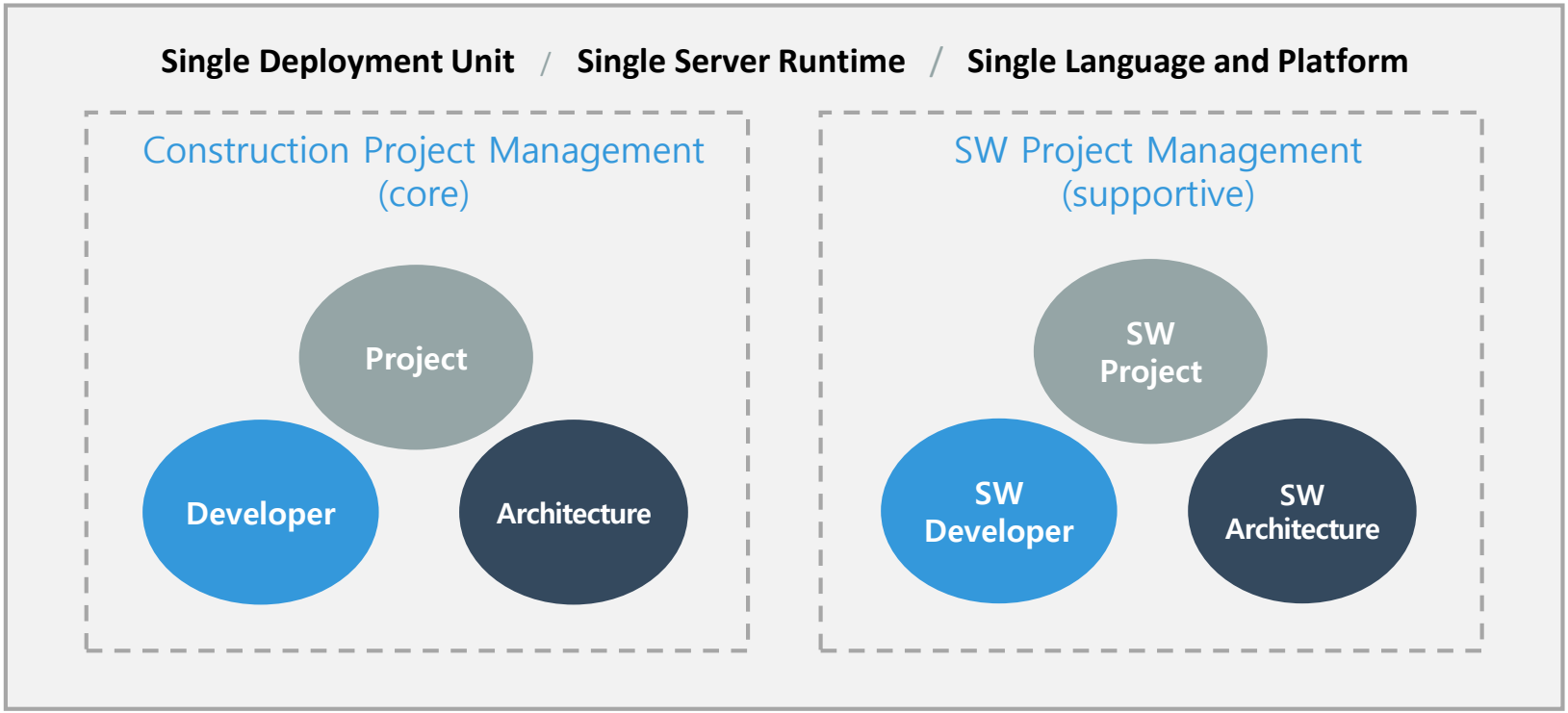
Bounded Context
(한정된 맥락)

Ubiquitous Language
(도메인 언어)

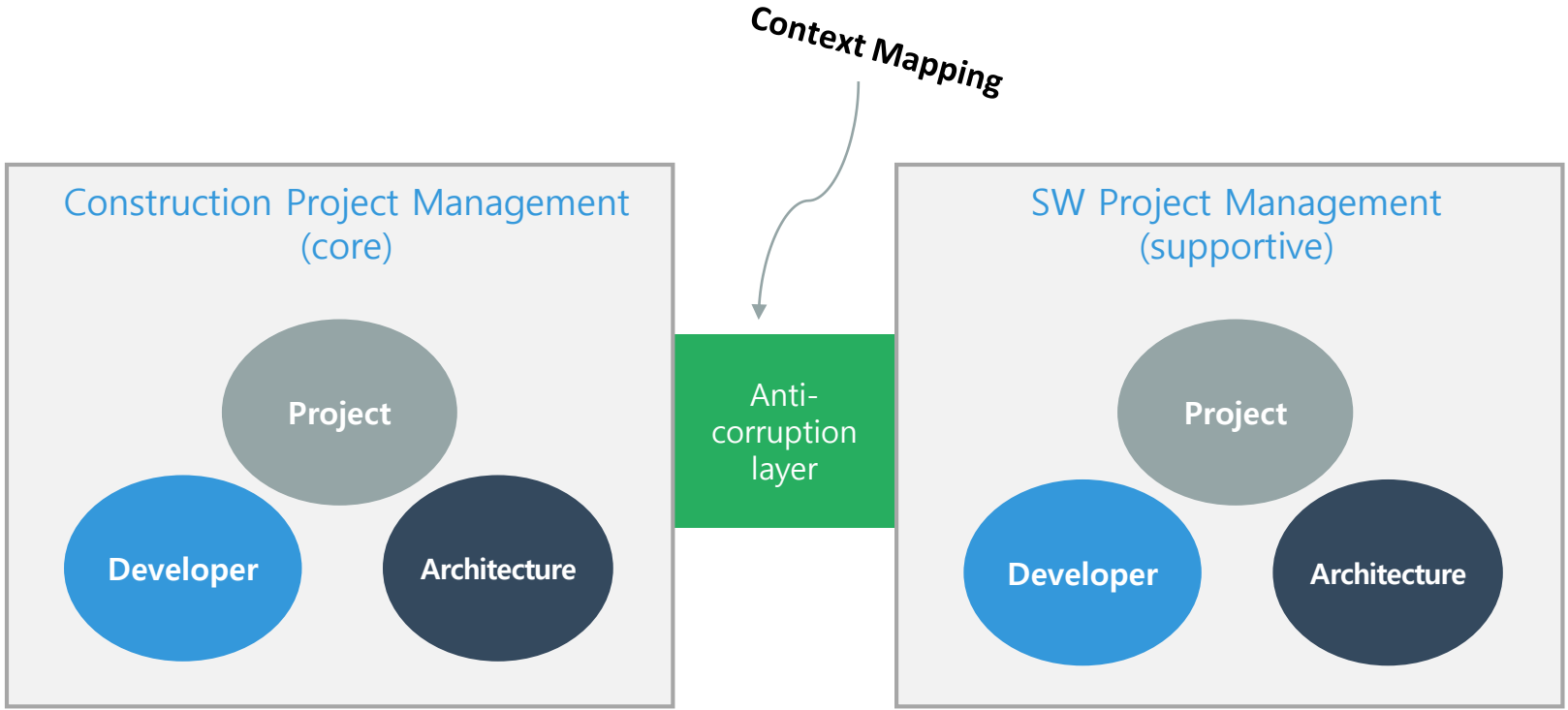
Bounded Context and Ubiquitous Language

	SW	CONSTRUCTION
A Project		
An Architecture		
A Developer		

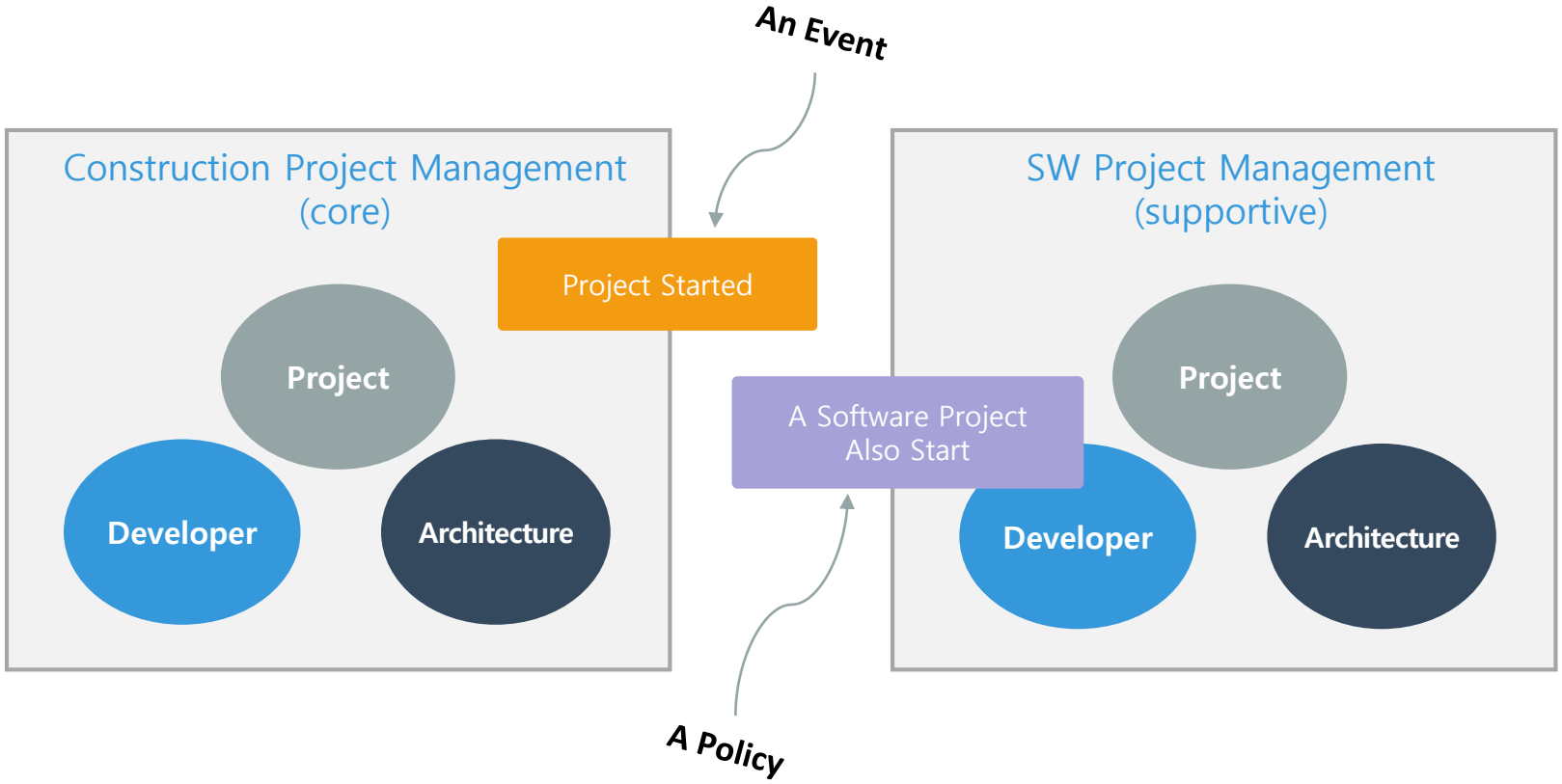
In Monolith :


Single Deployment Unit / Single Server Runtime / Single Language and Platform
Construction Project Management (core)
Project
Developer
Architecture
SW Project Management (supportive)
SW Project
SW Developer
SW Architecture

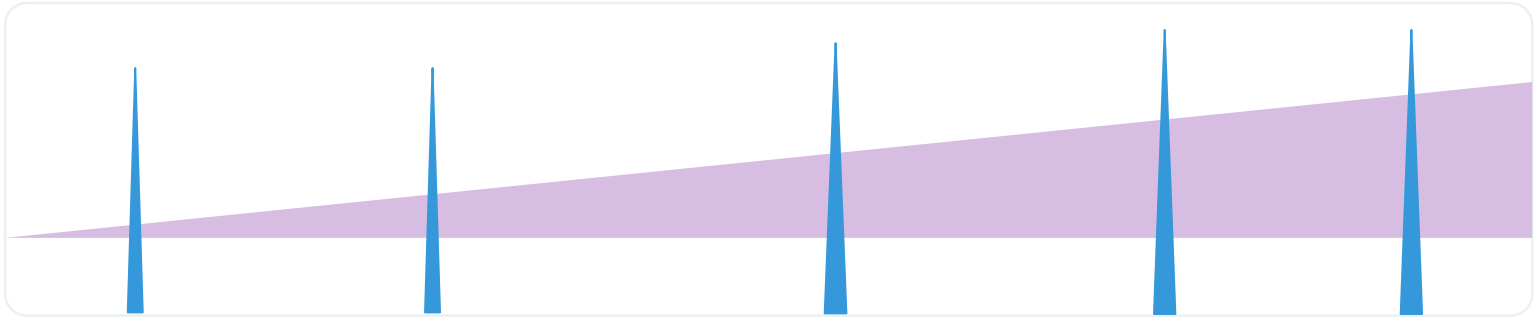
In Microservices :



In Microservices :



5개의 눈금: 어느 굵기로 쪼갤 것인가? 그것이 문제로다...



Aggregate

최소 사이즈 – ACID 가 준수되어야 하는 원자적 단위
- 유연성이 높은

Bounded context

커뮤니케이션이 원활한 단위 – Separation of Concerns 이 좋은

Business capability

비즈니스적 성과 측정과 피버팅이 용이한 단위 – 팀단위와 연결하기 좋은

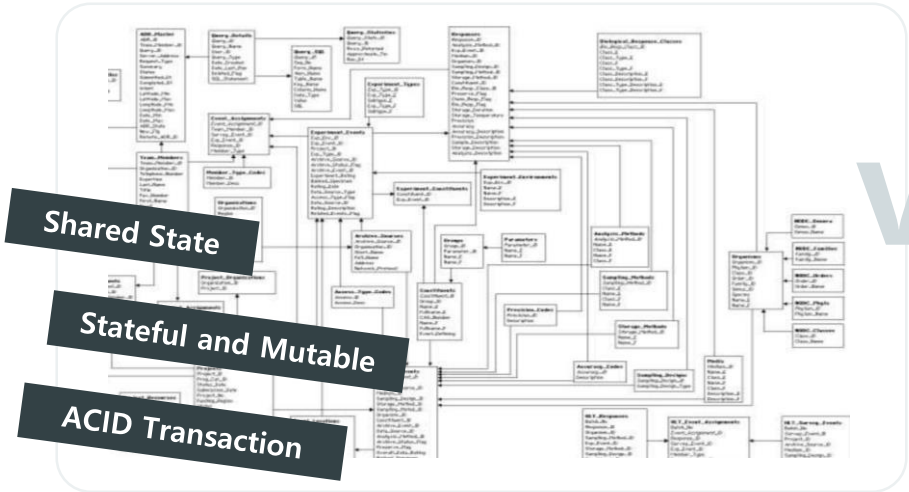
Mini

구현이 용이한 단위
- 점진적 MSA 전환이 용이한 단위

Monolith

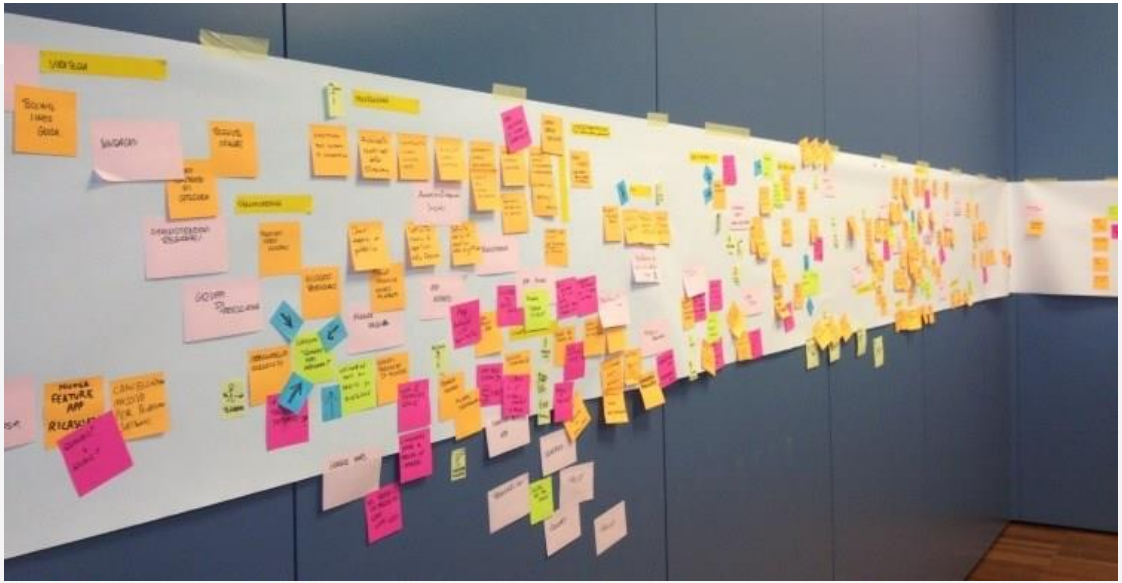
Event Storming : DDD 를 쉽게 하는 방법

- **이벤트스토밍은** 시스템에서 발생하는 **이벤트를 중심 (Event-First) 으로** 분석하는 기법으로 특히 Non-blocking, Event-driven 한 **MSA 기반 시스템을 분석에서 개발까지 필요한 도메인에 대한 탁월하게 빠른 이해를 도모**하는데 유리하다.
- 기존의 유즈케이스나 클래스 다이어그래밍 방식은 고객 인터뷰나 엔티티 구조를 인지하는 방식과 다르게 별다른 **사전 훈련된 지식과 도구 없이 진행할 수 있다**.
- 진행과정은 참여자 워크숍 방식의 방법론으로 결과는 **스티키 노트를 벽에 붙힌 것으로 결과가 남으며, 오랜지색 스티키 노트들의 연결로 비즈니스 프로세스가 도출**되며 이들을 이후 BPMN과 UML 등으로 정재하여 전환할 수 있다.



Event Storming : Prepare

- 3팀 이상으로의 구성 및 팀당 최소 2명 이상 구성
- 큰 종이 시트 및 종이 시트를 여러 장 붙일 수 있는 충분한 벽이 있는 넓은 공간
- 여러 종류의 색깔 스티커, 검은색 펜, 검은색 or 파란색 테이프
- 서서 하는 방식으로 의자 필요 없음.



Type of stickers

Domain Event (Orange)

P.P 형태의 동사

도메인 전문가가 정의
이벤트 퍼블리싱

사용자, 페르소나, 스테이크 홀더

유저 인터페이스를 통해 데이터를
소비하고 명령을 실행하여
시스템과 상호 작용

Definition

정의

도메인에 대한 용어 등의
설명,기술

Command (Sky Blue)

명령

현재형으로 작성,
행동, 결정 등의 값들에 대한
정의 UI 혹은 API

Aggregate (Yellow)

집합체

비즈니스 로직 처리의 도메인 객체
덩어리. 서로 연결된 하나 이상의
엔터티 및 value objects의 집합체

Read Model (Green)

리드모델

행위와 결정을 하기 위하여
유저가 참고하는 데이터, 데이터
프로젝션이 필요 : CQRS 등으로 수집

External System (Pink)

외부 시스템

시스템 호출을 암시 (REST)

Policy (Lilac)

업무정책

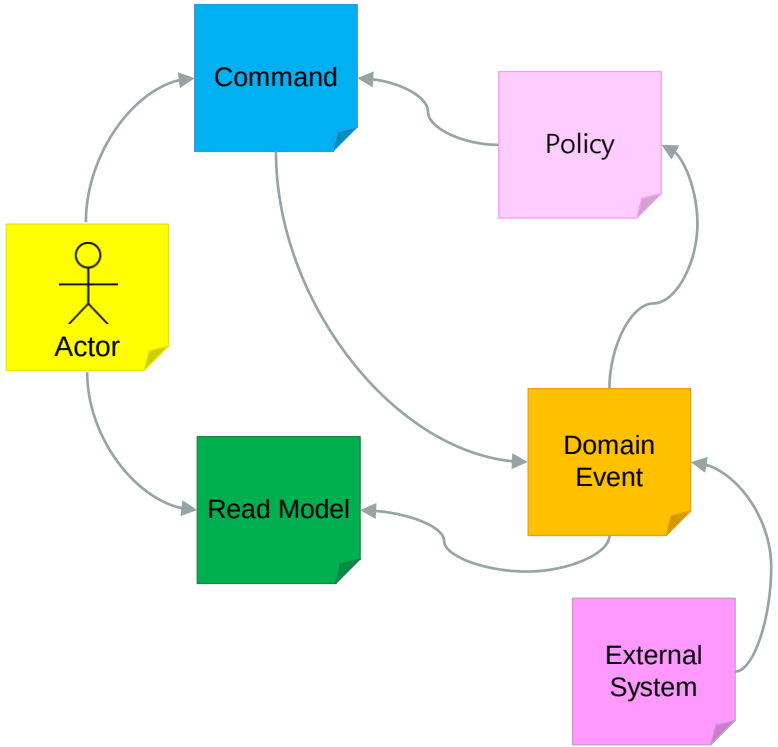
이벤트에 대한 반응
(서브스크라이브)
비즈니스 룰 엔진 등

Comment Or Question (Purple)

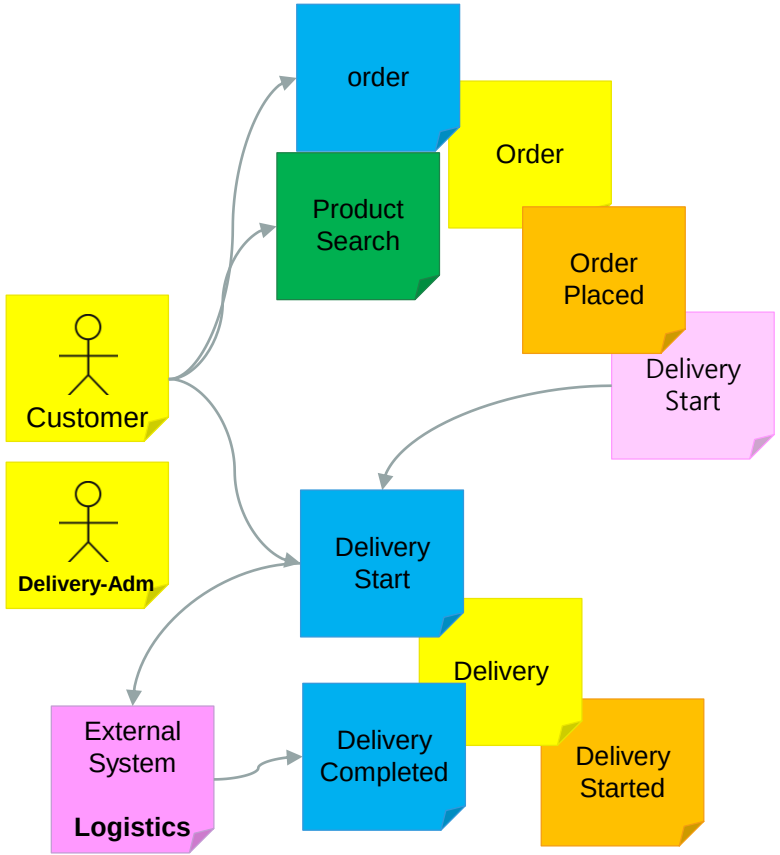
의견 또는 질문

추기적인 네용 입력,
예측되는 Risk

Examples



[Meta Model]



[12st Sample]

Firstly, Event Discovery

- 오렌지(주황색) 스티커 사용
- 각 도메인 전문가들이 개별 도메인 이벤트 목록 작성
- 이벤트는 도메인 전문가와 비즈니스 관계자 서로 이해할 수 있는 의미 있는 방식으로 표현하고 동사의 과거형 (p.p.)으로 표현한다.
- 시작 및 종료 이벤트를 식별하고 스티커를 붙일 벽의 시작과 끝의 타임라인에 배치
- 이벤트를 페르소나와 관련시키는 방법에 대해 논의
- 중복된 이벤트를 발견하면 중복된 이벤트를 벽에서 제거
- 불분명한 경우 다른 색상의 스티커 메모를 사용하여 질문이나 의견을 추가(빨간색 스티커)
- 이벤트에 대해서 동사를 과거 시제로 입력하고 다른 이벤트와 명확하게 구분되는 용어를 사용

잘 도출된 이벤트와 아닌 경우

O

다른 팀에서 관심 가질만함

다른 팀에서 관심 가질만함

적당한 사이즈임

X

It's a command

It doesn't make any state change

Too technical and too fine-grained

Be business level

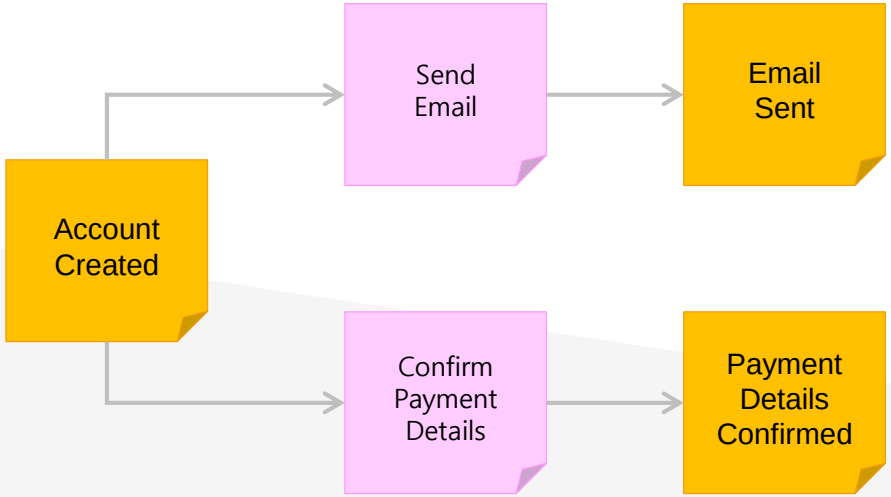
Nobody will be interested in this event

Command 와 Actor, UI 도출

- Command는 파란색 스티커 메모를 사용
- 도메인 분석에서 시스템 설계의 첫 단계
- 관심있는 비즈니스 프로세스를 구현하는 시스템을 구축하려면 이러한 이벤트가 발생하는 방식에 대한 질문 등으로 정의한다.
- 목표는 이벤트가 효과를 기록하는 원인을 찾는 것
- 예상되는 이벤트 트리거 유형의 예
  - Operator 결정을 내리고 명령을 내릴 시
  - 외부 시스템 또는 센서가 자극될 시
  - 정해진 시간 경과 시
- 마이크로 서비스 구현에서 API가 될 수 있다.
- Command를 사용하는 인간 혹은 주체는 노란색 스티커 메모를 사용한다.

Policy 도출

- 라일락 색의 스티커 사용
- Policy는 이벤트가 발생한 후 발생하는 반응형 논리
- Policy는 다른 Command를 트리거
- Policy는 process 같이 사람이 행하는 수동 동작 및 자동화 될 수 있다.



*"Whenever a **new user account is created** we will **send her an acknowledgement by email.**"*

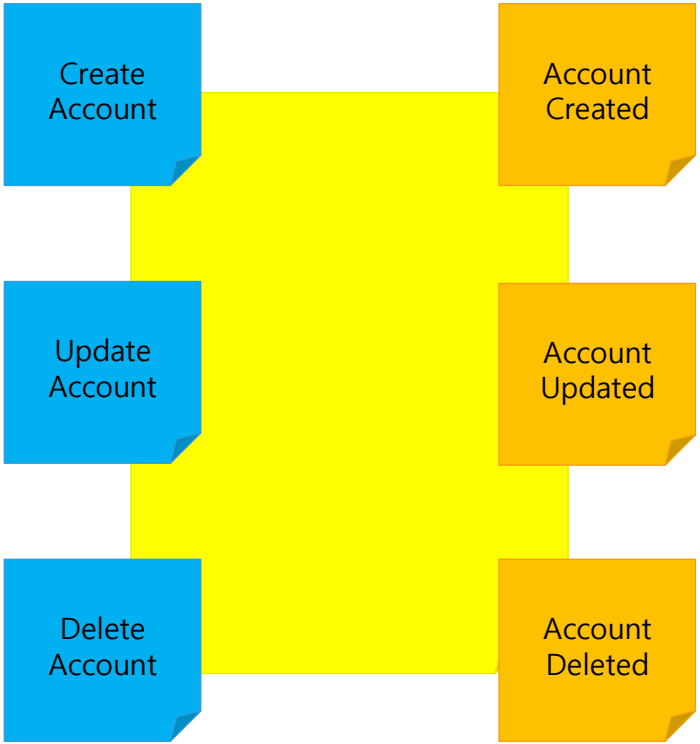
Read Model 도출

- 녹색 스티커 사용
- 이벤트를 생성하기 위해 Command을 실행하는데 필요한 데이터의 이해
- The Data needed in order to make that decision
  사용자의 의사 결정 과정을 지원하는 도구로 사용됨
- 각 Command 및 Event에 대해 필요한 속성 및 데이터 요소에 대한 설명

" Read Model Derivation "

Aggregate 도출

- 노란색 스티커 사용
- 같은 Entity를 사용하는 연관 있는 도메인 이벤트들의 집합
- 관련 데이터 (Entity 및 value objects)뿐만 아니라 해당 Aggregates의 Life Cycle에 의해 연결된 작업(Command)으로 구성

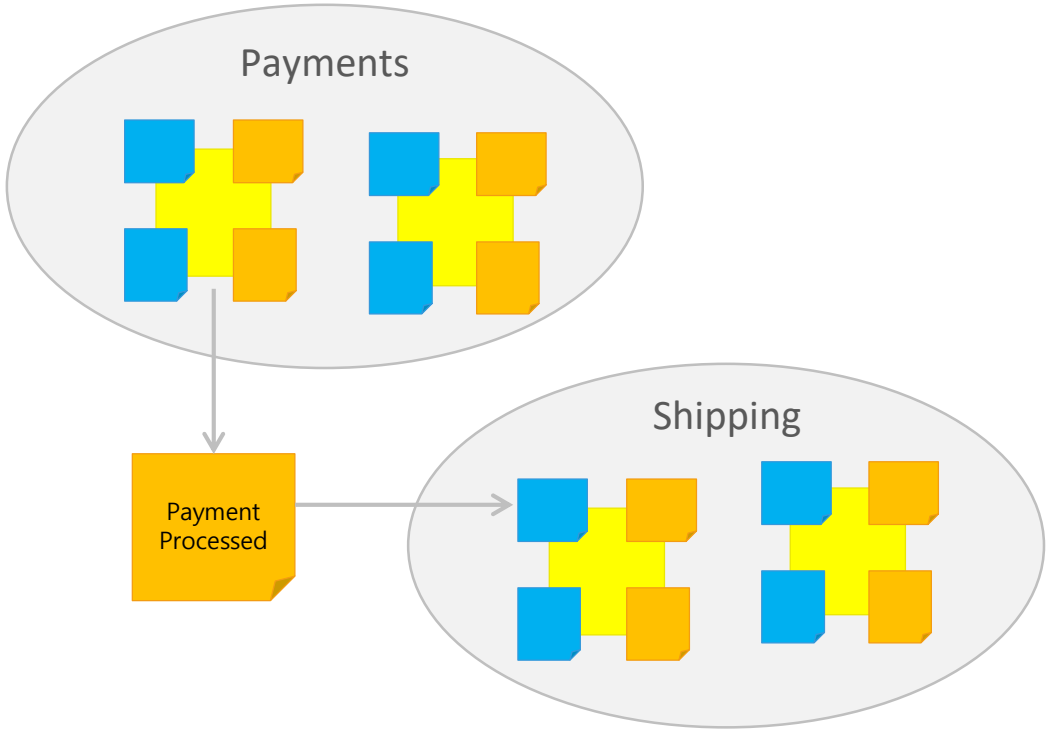


Bounded Contexts 도출

- 이벤트의 내용을 정의하고 시스템의 경계를 구분

- 찾는 방식은 두가지 유형으로 구분되어진다

  - Time Boundary
  - Subject Boundary



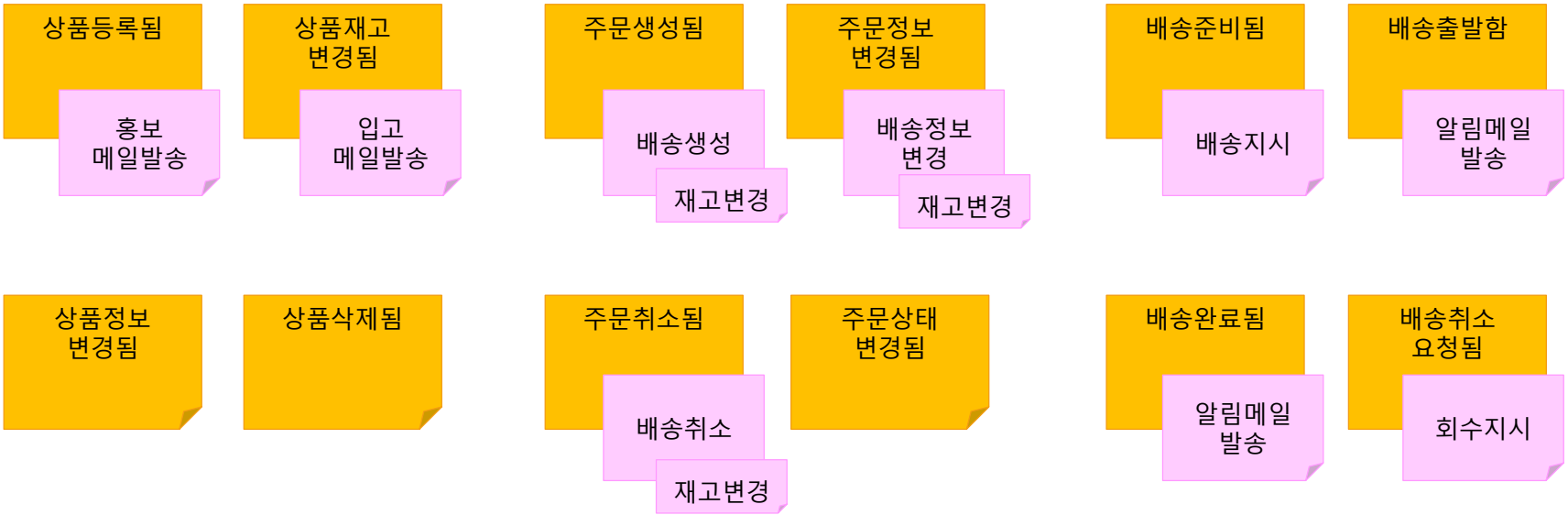
Source: https://blog.redelastic.com/corporate-arts-crafts-modelling-reactive-systems-with-event-storming-73c6236f5dd7, Kevin webber

Lab Time – Event 들을 먼저 도출

우리 서비스에는 어떤 비즈니스 이벤트들이 발생하는가?
현업이 사용하는 용어를 그대로 사용 (Ubiquitous Language)
용어의 namespace 를 굳이 나누려는 노력을 하지 않음

이벤트 : 오렌지 스티커!

Lab Time – Policy 도출



어떤 이벤트에 이어서 곧바로 항상 발생해야 하는 업무규칙
구현상에서는 이벤트의 Publish에 따라 벌어지는 이후의 프로세스가 자동으로 트리거 되게 함

규칙 : 라일락 스티커!

Lab Time – Command 도출 (쓰기행위)

Event를 발생시키는 명령은 무엇인가? UI를 통해? or 시간도래? or 다른 이벤트에 의해?

Command 는 어떠한 상태의 변화를 일으키는 서비스를 말함.
** Command 라는 용어는 CQRS 에서 유래했으며, 쓰기서비스인 Command 와 읽기행위인 Query는 구분됨.*

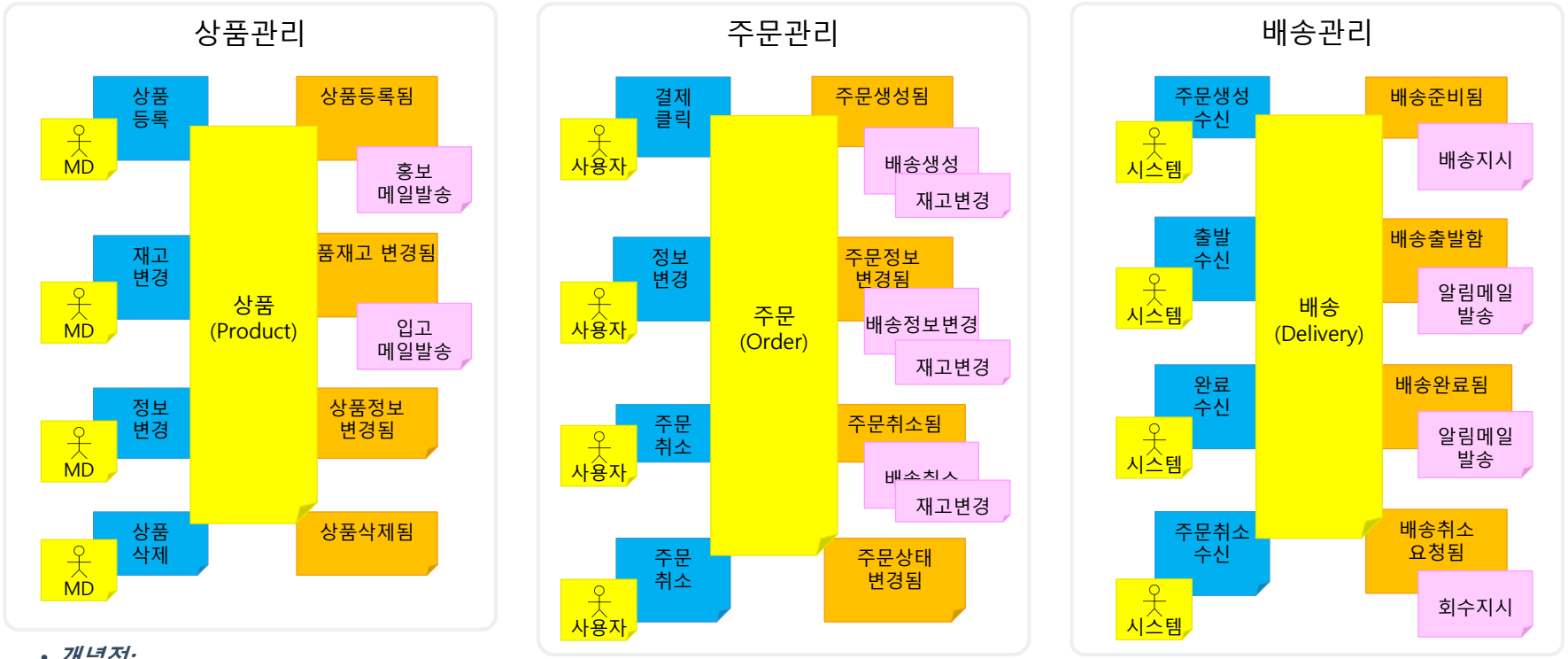
Lab Time – Command Actor 식별

Actor 는 Command를 발생시키는 주체 , 담당자 또는 시스템 (외부 (External) 또는 내부)이 될 수 있음

Lab Time – Aggregate 도출

도메인 이벤트가 발생하는 데는, 어떠한 도메인 객체의 변화가 발생했기 때문이다.
하나의 ACID 한 트랜잭션에 묶여 변화되어야 할 객체의 묶음을 도출하고, 그것들을 커맨드, 이벤트와 함께 묶는다.

Lab Time – Bounded Context 도출

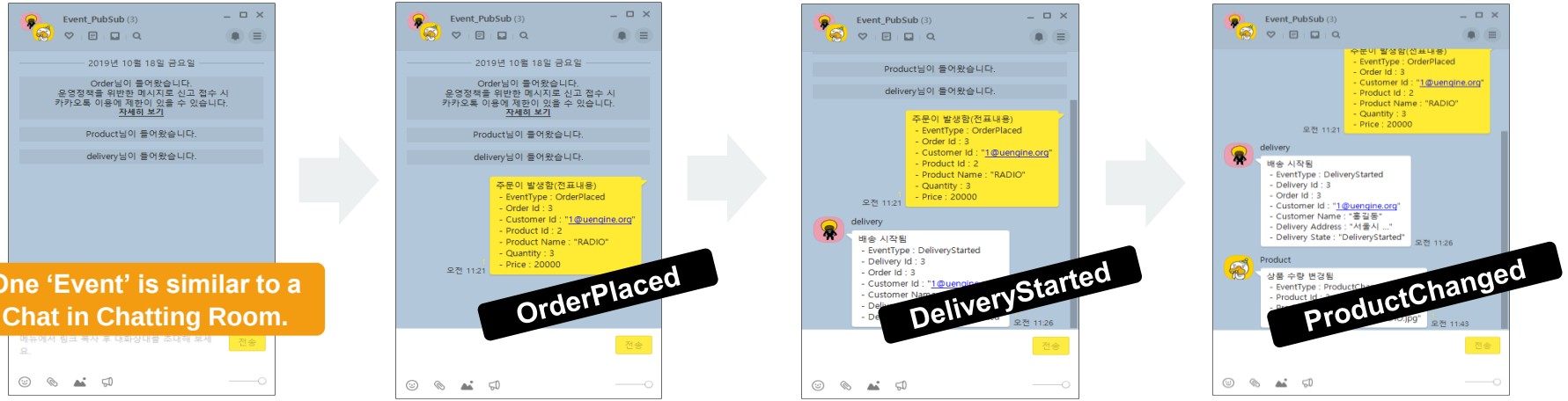


- ***개념적:***
  Bounded Context 는 동일한 문맥으로 효율적으로 업무 용어 (도메인 클래스)를 사용할 수 있는 객체 범위를 뜻한다.
  하나의 BC 는 하나이상의 어그리게잇을 원소로 구성될 수 있다. BC를 Microservice 구성단위로 정하게 되면 이를 담당한 팀 내의 커뮤니케이션이 효율화된다.
- ***구현관점:***
  어그리게잇은 ACID 트랜잭션 범위이기도 하기때문에 이를 더 쪼개서 BC 를 구성할 수는 없다.
  BC 내의 어그리게잇들에 대한 보존 (Persistence)은 아마도 그 목적에 맞는 최적화된 데이터모델을 독립적으로 설계하고 구현할 수 있다.

12번가 Shopping Mall Bounded Context

REAL CASE

12st Microservices Event Sample :



One 'Event' is similar to a Chat in Chatting Room.

Chatting Room

```
>
>{"eventType":"OrderPlaced","timestamp":"20190916151922","stateMessage":"Order Placed","productId":2,"orderId":3,"productName":"RADIO","quantity":3,"price":20000,"customerId":"1@uengine.org","customerName":"Hong Gil Dong","customerAddr":"Seoul"}
>
```

```
r.bat --broker-list http://localhost:9092 --topic eventTopic
>{"eventType":"DeliveryStarted","timestamp":"20190916151922","stateMessage":"Delivery Started","deliveryId":3,"orderId":3,"customerId":"1@uengine.org","customerName":"Hong Gil Dong","deliveryAddress":"Seoul ...","deliveryState":"DeliveryStarted"}
>
```

```
12-2.1.0₩bin₩windows>kafka-console-producer.bat --broker-list http://localhost:9092 --topic eventTopic
>{"eventType":"ProductChanged","timestamp":"20190916151922","stateMessage":"Product Changed","productId":2,"productName":"RADIO","productPrice":20000,"productStock":14,"imageUrl":"/goods/img/RADIO.jpg"}
>
```

```
{"eventType":"OrderPlaced","timestamp":"20190916151922","stateMessage":"Order Placed","productId":2,"orderId":3,"productName":"RADIO","quantity":3,"price":20000,"customerId":"1@uengine.org","customerName":"Hong Gil Dong","customerAddr":"Seoul"}
{"eventType":"DeliveryStarted","timestamp":"20190916151922","stateMessage":"Delivery Started","deliveryId":3,"orderId":3,"customerId":"1@uengine.org","customerName":"Hong Gil Dong","deliveryAddress":"Seoul ...","deliveryState":"DeliveryStarted"}
{"eventType":"ProductChanged","timestamp":"20190916151922","stateMessage":"Product Changed","productId":2,"productName":"RADIO","productPrice":20000,"productStock":14,"imageUrl":"/goods/img/RADIO.jpg"}
```

Table of content

Microservice and
Event-storming-Based
DevOps Project

UEngine

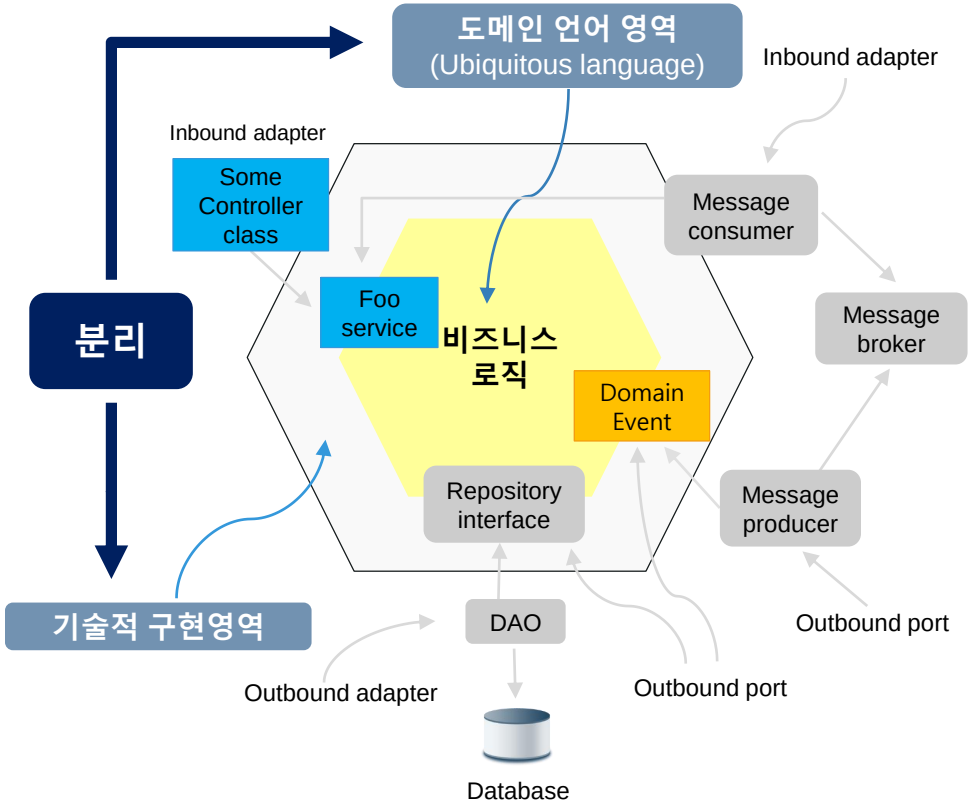
Microservice Implementation Pattern (1)

- **Hexagonal Architecture**

Create your service to be independent of either UI or database and to provide adapters for different input/output sources such as GUI, DB, test harness, RESTful resource, etc.

Implement the publish-and-subscribe messaging pattern: As events arrive at a port, an adapter (a.k.a. service agent) converts it into a procedure call or message and passes it to the application. When the application has something to publish, it does it through a port to an adapter, which creates the appropriate signals needed by the receiver.

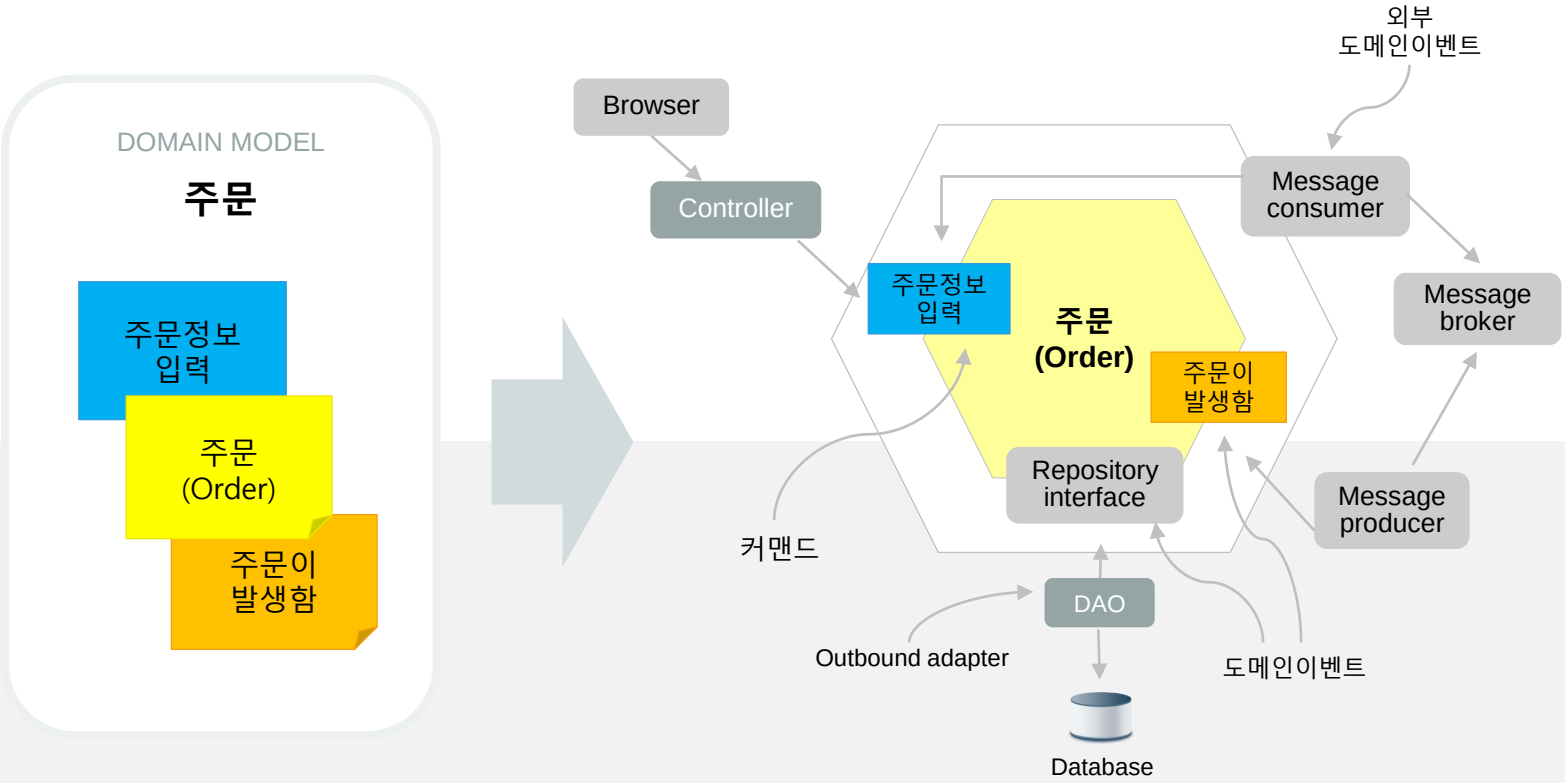
https://alistair.cockburn.us/Hexagonal+architecture



Service Implementation

- You have Powerful Tool:
  Domain-Driven Design and Spring Boot / Spring Data REST
- Domain Classes : Entity or Value Object
- Resources can be bound to Repositories → Full HATEOAS service can be generated!
- Services can be implemented with Resource model firstly
- Low-level JAX-RS can be used if not applicable above

Step 1 : Applying Hexagonal Architecture

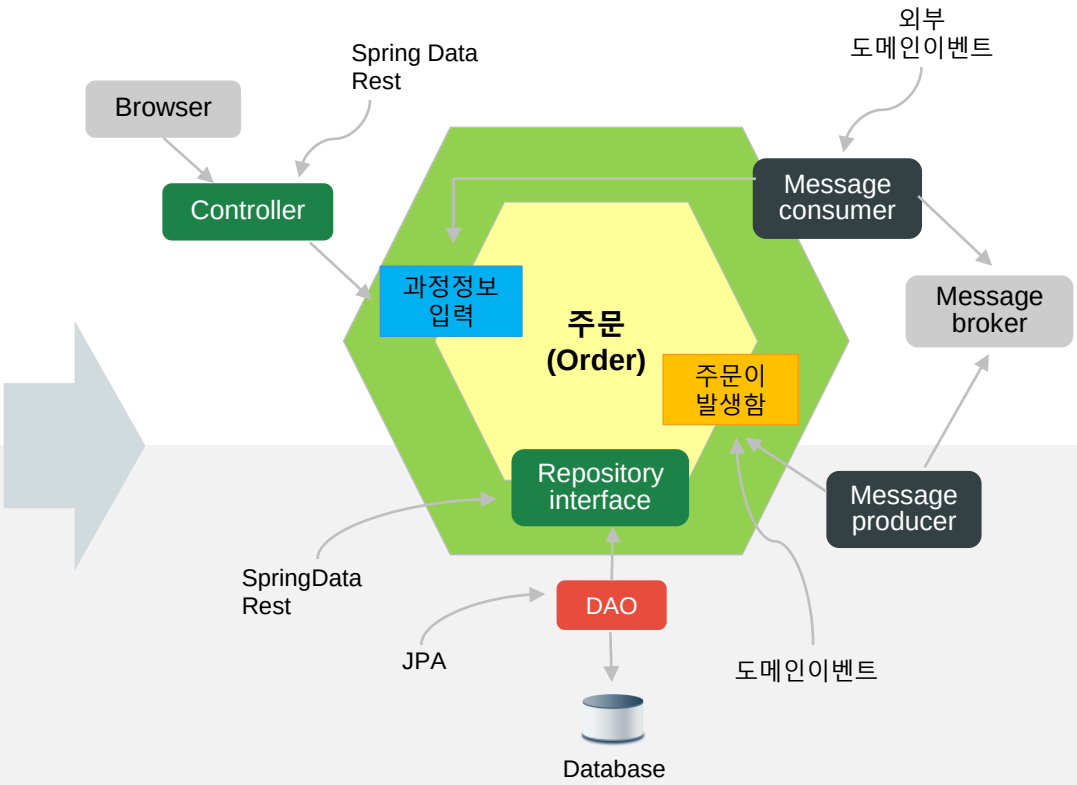


Step 2 : Applying MSA Chassis

DOMAIN MODEL

Spring Boot / Cloud

- Spring Boot
- Spring Data Rest
- JPA
- Spring Cloud Stream



이벤트 스토밍 결과에서 구현 기술 연동

요소	구현체	미들웨어/프레임워크
이벤트 (Domain Event)	• 도메인 이벤트 클래스 (POJO)	• 카프카 퍼블리시 • Rabbit MQ
커맨드 (Command)	• 서비스 객체 • 레포지토리 객체 (PagingAndSortingRepository)	• Spring Data REST • RESTeasy
결합물 (Aggregate)	• 엔티티 클래스 (ORM) • 도메인 모델	• JPA Entity • Value Objects
정책 (Policy)	• 엔티티 클래스의 Hook에 정책 호출 구문 주입 • CDC 를 통한 이벤트 후크	• 카프카 Event Listening Code • JPA Lifecycle Hook • CDC (Change Data Capturing)
바운디드 컨텍스트	• 마이크로 서비스 (후보)	• Spring Boot

분석/설계 (Sticker)

Order

- orderId
- name
- userId
- price
- quantity

Aggregate → Domain Model

```
@Entity
public class Order{

  Long id;
  String name;
  String customerId;
  double price;
  int quantity;

… setter/getters …

}
```

```
@Entity
public class OrderDetail {

….

… setter/getters …

}
```

주문
(POST)

주문수정
(PATCH)

주문삭제
(DELETE)

Command → CRUD 에 해당하면? Repository Pattern 으로 자동생성

```
public interface OrderRepository extends
            PagingAndSortingRepository<Order, Long>{


}
```

Command → Repository Pattern 이 안될시에 MVC 패턴으로 구현

```
@Service
public interface ProductService {
   @RequestMapping(method = RequestMethod.GET,
path=/myservice/{productId}/")
   Resources getProduct(@PathVariable("productId") Long productId);
}
```

분석/설계 (Sticker)

OrderPlaced

- orderId
- name
- userId
- price
- quantity

개발 (POJO)

```
public class OrderPlaced{

  Long orderId;
  String name;
  String userId;
  double price;
  int quantity;

… setter/getters …

}
```

실행 (JSON)

```
{
  type: “OrderPlaced”,
  name: “캠핑의자”,
 userId : “1@uengine.org”,
  orderId: 12345,
  price:  100
 quantity :   10
}
```

OrderPlaced

- orderId
- name
- userId
- price
- quantity

Event → POJO Class 와 이벤트 발사로직

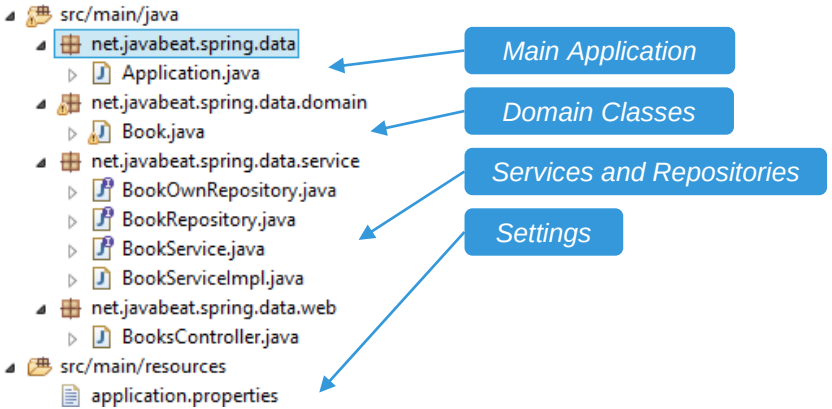
```
@Entity
public class Order {
 …

   @PostPersist   // 주문이 저장된 후에
   private void publishOrderPlaced() {
      OrderPlaced orderPlaced = new OrderPlaced();  // 주문이 들어온 사실을
이벤트로 작성

      orderPlaced.setOrderId(id);
          …

      kafka.send(orderPlaced);   // 메시지 큐에 주문이 들어왔음을 신고
   }
}
```

재고변경

Policy → Kafka Listener Logic

```
@StreamListener(Processor.INPUT)  // 쇼핑 토픽을 수신
   public void onOrderPlaced(…) {  // 주문이 들어오는 이벤트가 오면

      // 해당 상품의 재고량을 주문량 만큼 줄여서 저장
      Product product =
productRepository.findById(orderPlaced.getProductId()).get();
      product.setStock(product.getStock() - orderPlaced.getQuantity());

      productRepository.save(product);

   }
```

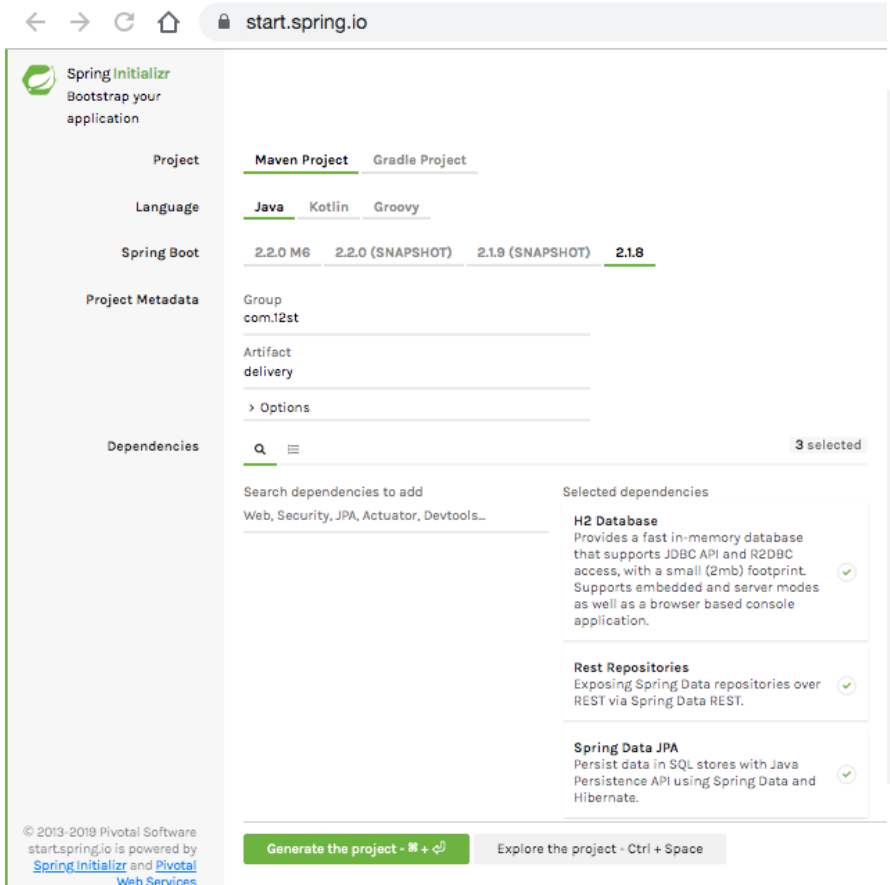
Spring Boot : A MSA Chassis



- Simple Java (POJO)
- Minimal Understanding Of Frameworks and Platforms (Using Annotations)

- No Server-side GUI rendering (Only Exposes REST Service)
- No WAS deployment required (Code is server; Tomcat embedded)
- No XML-based Configuration

Lab : Create a Spring boot application



Go to start.spring.io
Set metadata:

- Group: com.12st
- Artifact: order
- Dependencies:
  - H2
  - Rest Repositories
  - JPA

Press “Generate Project” Extract the downloaded zip file Build the project:
./mvnw spring-boot:run

Port 충돌시:
./mvnw spring-boot:run -Dserver.port=8081

Running Spring Boot Application

$ mvn spring-boot:run # 혹은 ./mvnw spring-boot:run

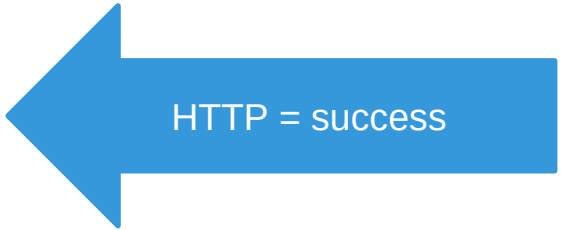
```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building ….
[INFO] ------------------------------------------------------------------------
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/springframework/security/spring-security-core/maven-metadata.xml
[INFO] Downloading: https://oss.sonatype.org/content/repositories/snapshots/org/springframework/security/spring-security-core/maven-metadata.xml
[INFO] Downloading: https://repo.spring.io/libs-release
:
```

Started Application in 11.691 seconds (JVM running for 14.505)

Test Generated Services

```
$ http localhost:8080
```

```
HTTP/1.1 200
Content-Type: application/hal+json;charset=UTF-8
Date: Wed, 05 Dec 2018 04:20:52 GMT
Transfer-Encoding: chunked

{
    "_links": {
        "profile": {
            "href": "http://localhost:8080/profile"
        }
    }
}
```



Aggregate → Entity Class 작성

상품

```
@Entity
public class Product {

    @Id
    @GeneratedValue
    private Long id;

    String name;
    int price;
    int stock;


    @OneToMany( cascade =
CascadeType.ALL, fetch = FetchType.EAGER,
mappedBy = "order")
    List<Orde> orders;
}
```

주문

```
@Entity
public class Order {

    @Id
    @GeneratedValue
    private Long id;
    private Long productId;
    private String productName;
    private int quantity;
    private int price;
    private String customerName;
    private String customerAddr;

     @ManyToOne
@JoinColumn(name="productId")
    Product product;
}
```

배송

```
@Entity
public class Delivery {

    @Id @GeneratedValue
    private Long deliveryId;
    private Long orderId;
    private String customerName;
    private String deliveryAddress;
    private String deliveryState;

    @OneToOne
    Order order;
}
```

Commands → Service Object* 와 API의 생성

Repository Pattern 으로 생성된 것
사용할 수 있는 경우

주문과 배송 API 는 Order Repository 에서 생성된
CRUD actions 들 중 POST Service 를
그대로 사용할 수 있으므로 별도 구현할 필요 없다.
마찬가지 주문서 수정, 삭제도
각각 PATCH, DELETE 로 API 생성

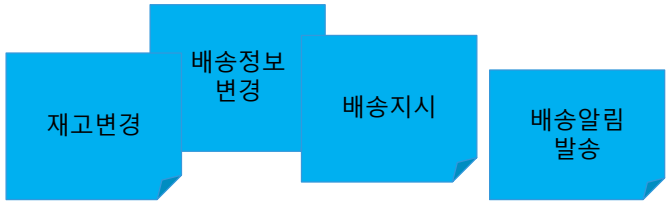
전체의 약 **7~80%** 커버


```
public interface OrderRepository extends PagingAndSortingRepository<Course, Long> {
    List<Course> findByOrderId(@Param("orderId") Long OrderId);
}
```

Repository 에서 생성된 API를 사용 못하는 경우,
별도 Spring MVC Service 로 생성

배송일정 등을 위한 백엔드 API 는
Order Repository 에 생성된
PUT-POST-PATCH-DELETE 중 적합한 것이 없으므로
별도 추가 action URI를 만드는 것이 적합

나머지 **2~30%** 수준




```
@Service
public interface ProductService {
    @RequestMapping(method = RequestMethod.GET, path="/myservice/{productId}/")
    Resources getProduct(@PathVariable("productId") Long productId);
}
```

Main Class

```
@SpringBootApplication
public class Application
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

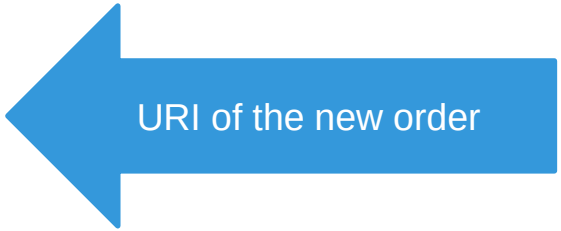
Test Generated Services

```
$ http localhost:8088
{
  "_links": {
    "deliveries": {
      "href": "http://localhost:8088/deliveries{?page,size,sort}",
      "templated": true
    },
    "orders": {
      "href": "http://localhost:8088/orders{?page,size,sort}",
      "templated": true
    },
    "products": {
      "href": "http://localhost:8088/products{?page,size,sort}",
      "templated": true
    },
    "profile": {
      "href": "http://localhost:8088/profile"
    }
  }
}
```

Create new Order

```
$ http localhost:888/orders poductId=1 quantity=3 customerId="1@uengine.org"
 customerName="홍길동" customerAddr="서울시"
```

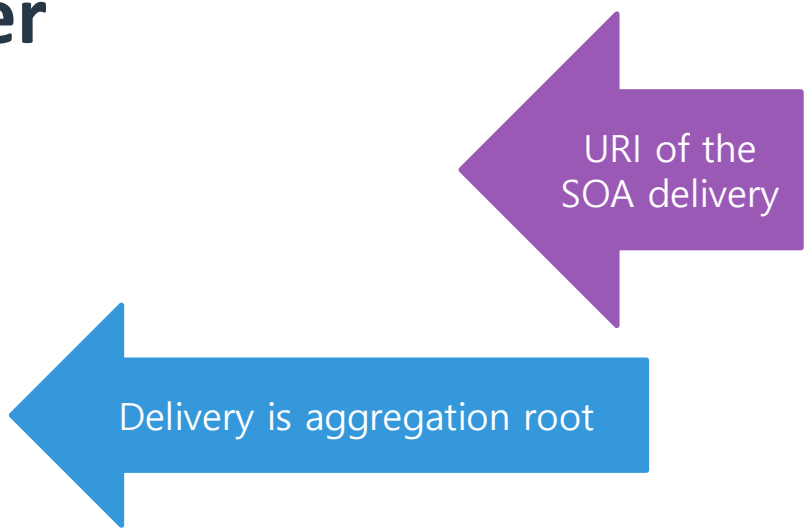

```
{
  "_links": {
    "delivery": {
      "href": "http://localhost:8088/orders/1/delivery"
    },
    "order": {
      "href": "http://localhost:8088/orders/1"
    },
    "product": {
      "href": "http://localhost:8088/orders/1/product"
    },
    "self": {
      "href": "http://localhost:8088/orders/1"
    }
  },
  "customerAddr": "서울시",
  "customerId": "1@uengine.org",
  "customerName": "홍길동",
  "price": 10000,
  "productId": 1,
  "productName": "TV",
  "quantity": 3,
  "state": "OrderPlaced"
}
```

Create a delivery form order

$ http **http://localhost:8088/orders/1/delivery**

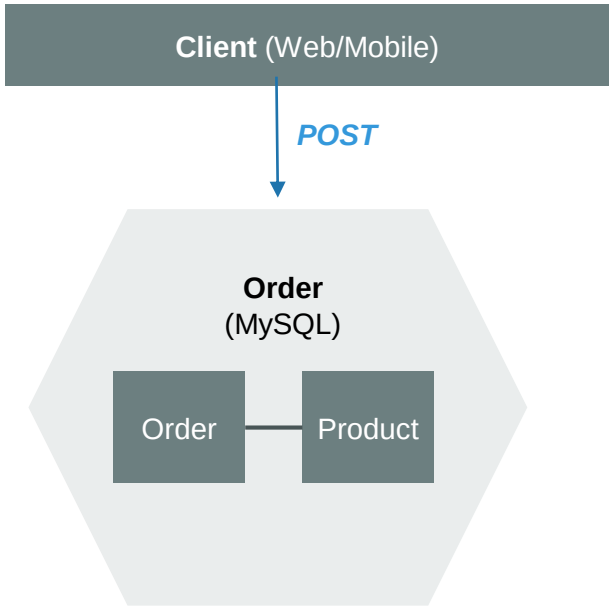

```
{
"_links": {
  "delivery": {
    "href": "http://localhost:8088/deliveries/1"
  },
  "order": {
    "href": "http://localhost:8088/deliveries/1/order"
  },
  "self": {
    "href": "http://localhost:8088/deliveries/1"
  }
},
"customerId": "1@uengine.org",
"customerName": "홍길동",
"deliveryAddress": "서울시",
"deliveryState": "DeliveryStarted",
"productName": "TV",
"quantity": 3
}
```

Monolithic 에서의 분산 트랜잭션 – 이슈없음

In-consistent Consistent

- 서비스구성 (모놀로씩)



- 조회화면

주문량 : 0 , 재고량 : 10

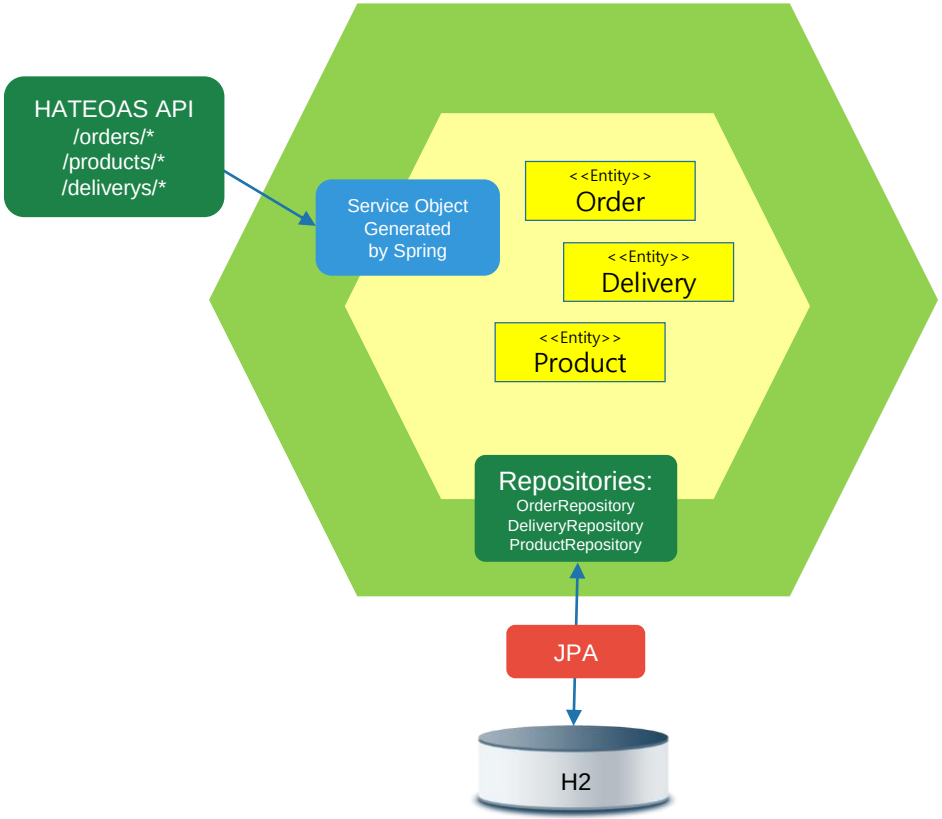
주문량 : 1 , 재고량 : 9

주문량 : 2 , 재고량 : 8

항상 일치함

But,
Tight-Coupling Shared Database Single DB Architecture….

Nice but Too big (monolith)



Plan to add…

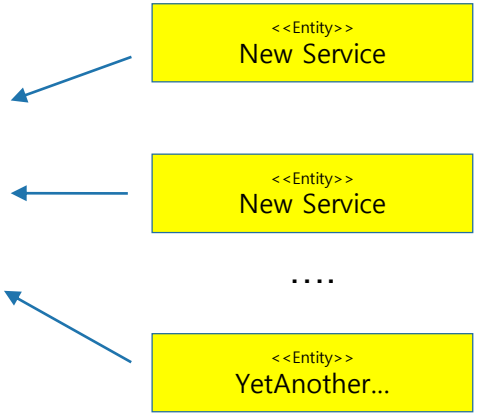


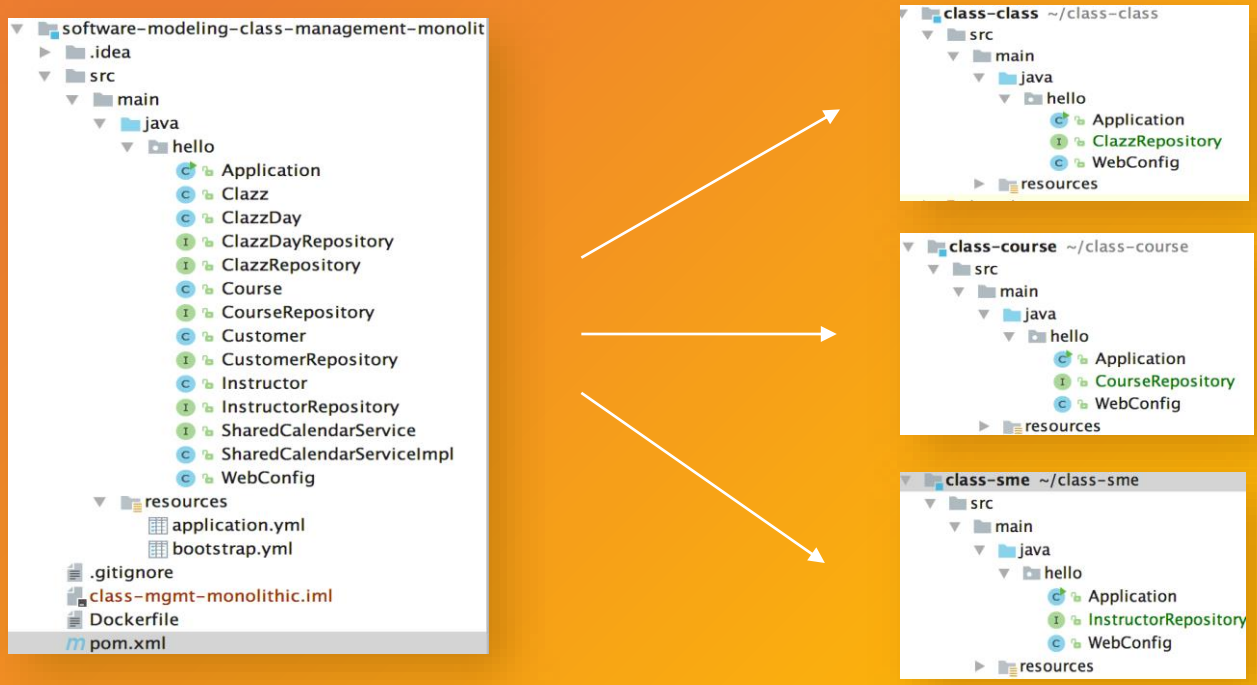
Table of content

Microservice and
Event-storming-Based
DevOps Project

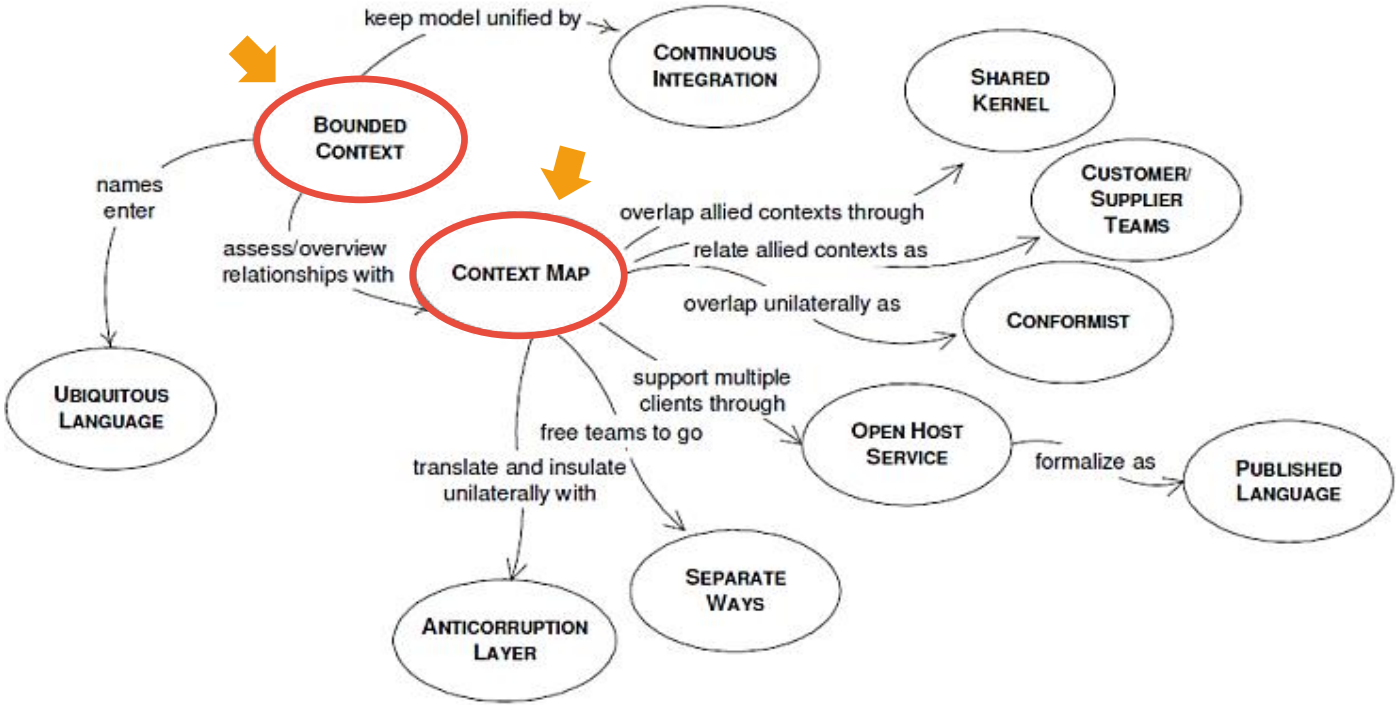
UEngine

From Monolith to Microservices

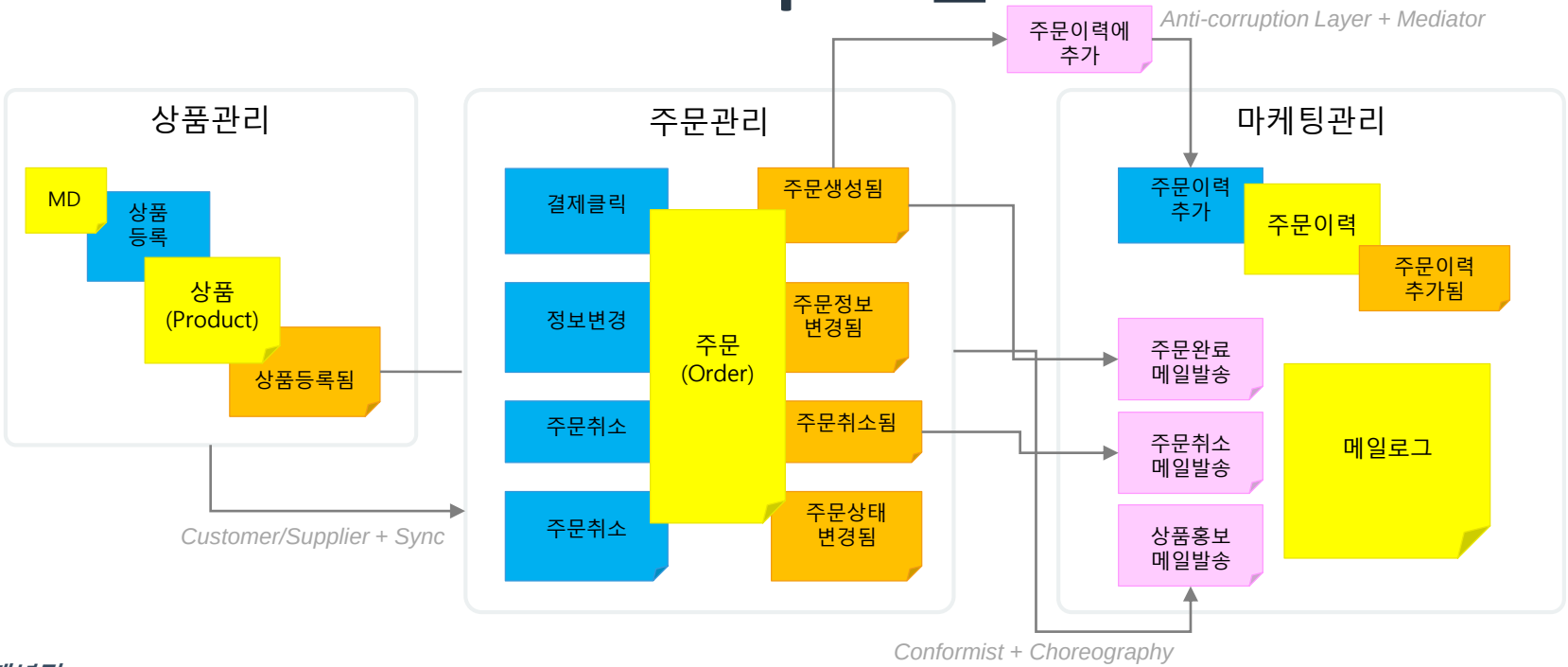
https://github.com/event-storming



DDD Pattern Applied



Lab Time – Context Map 도출

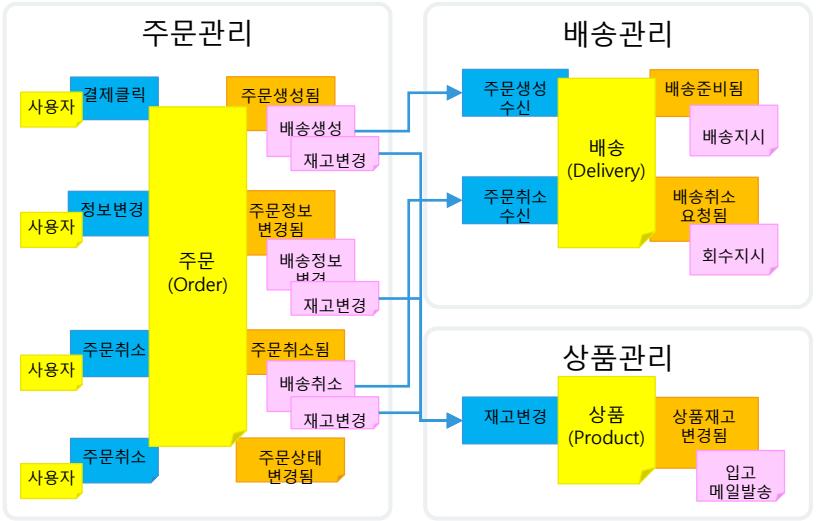


- ***개념적:***
  BC 간의 정보참조의 릴레이션, 혹은 이벤트가 발생한 이후의 동반된 행위의 호출의 관계를 선으로 표시함. Upstream → Downstream 으로 정보가 내려가게 됨.
- ***구현관점:***
  자치적으로 구현설계가 이루어질 수 있는 BC 간에 커뮤니케이션을 위해서는 데이터의 적절한 변환이나 표준화, Pub/Sub 등의 커뮤니케이션 방법 및 데이터 전환에 대한 이슈가 발생하게 됨. 해당 이슈를 적절한 패턴을 선택하여 실제 연동을 위한 Adapter 를 개발해야 함.

Lab Time – Topology Consideration

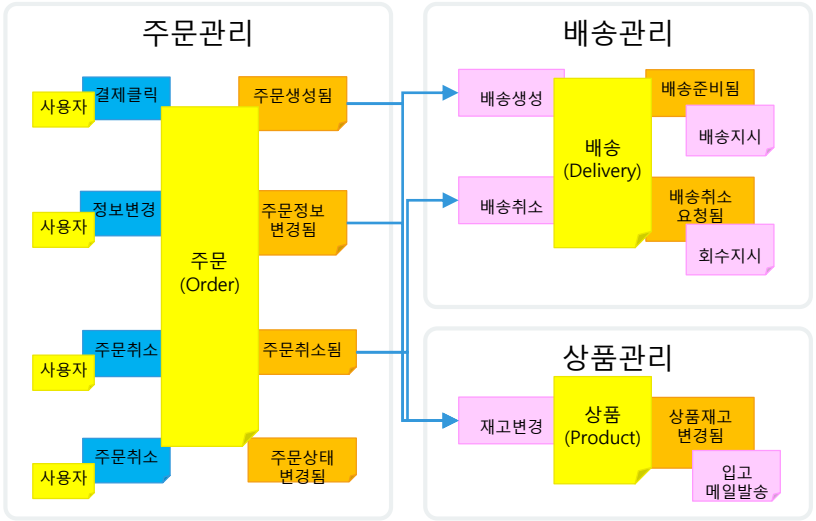
라이락 스티커 (Policy) 를 어디에 둘 것인가? - Orchestration or Choreography? Or Mediation?

Orchestration



Originator should know how to handle the policy
→ Coupling is High

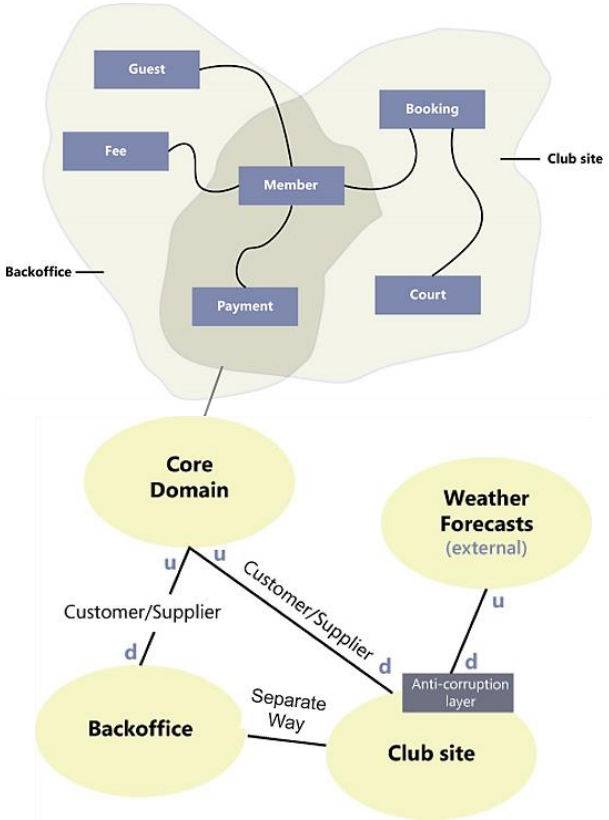
Choreography



More autonomy to handle the policy
→ Low-Coupling
→ Easy to add new policies

Ref : Relation types between services

- Shared Kernel
  - Designate a subset of the domain model that the two teams agree to share.
- Customer/Supplier
  - Make the downstream team play the customer role to the upstream team.
- Conformist
  - Eliminate the complexity of translation between bounded contexts by slavishly adhering to the model of the upstream team.
- Separate Ways
  - Declare a bounded context to have no connection to the others at all. The feat ures can still be organized in middleware or the UI layer.
- Open Host Service
  - Open the protocol so that all who need to integrate with you can use it. Other t eams are forced to learn the particular dialect used by the host team. In some situations, using a well-known published language as the interchange model c an reduce coupling and ease understanding.



Eric Evans, Domain-Driven Design, 2003.

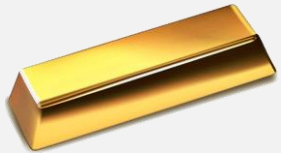
마이크로 서비스간의 서열과 역학관계

" 무엇을 우선적으로 챙길 것인가? "

1순위 : Core Domain

- 버릴 수 없는. 이 기능이 제공되지 않으면 회사가 망하는

예) 쇼핑몰 시스템에서 주문, 카탈로그 서비스 등



2순위 : Supportive Domain

- 기업의 핵심 경쟁력이 아닌, 직접 운영해도 좋지만 상황에 따라 아웃소싱 가능한
- 시스템 리소스가 모자라면 외부서비스를 빌려쓰는 것을 고려할만한

예) 재고관리, 배송, 회원관리 서비스 등

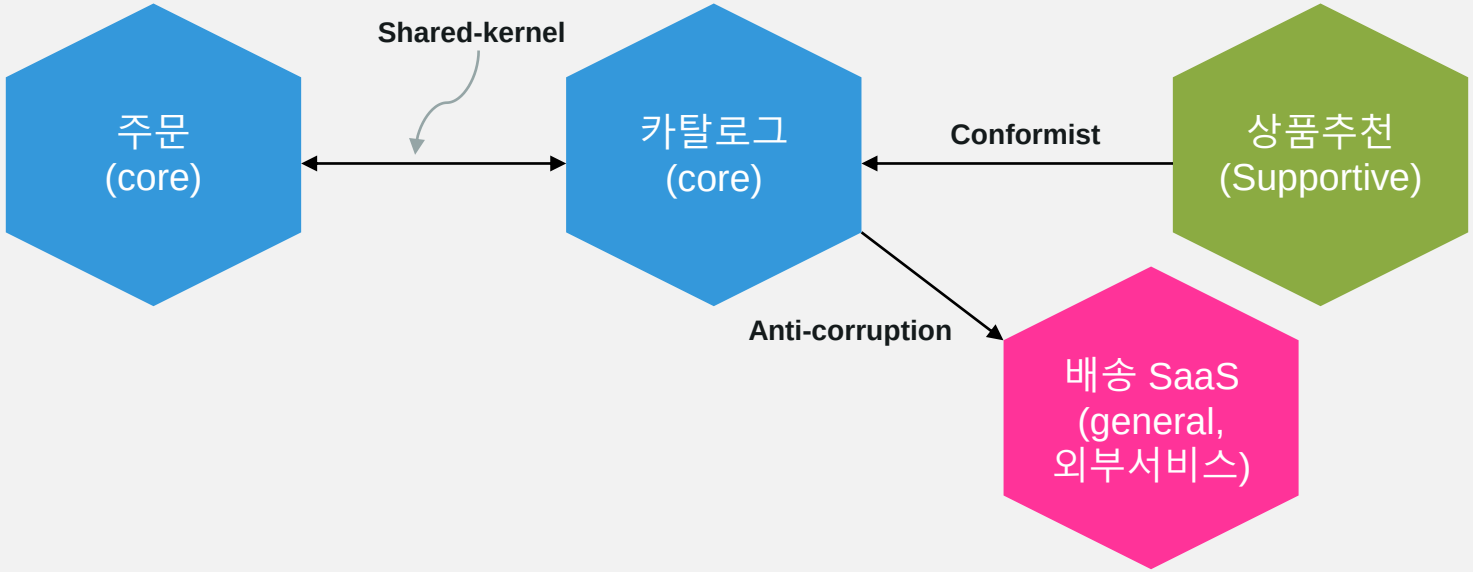
3순위 : General Domain

- SaaS 등을 사용하는게 더 저렴한, 기업 경쟁력과는 완전 무관한

예) 결재, 빌링 서비스 등

마이크로 서비스간의 서열과 역학관계

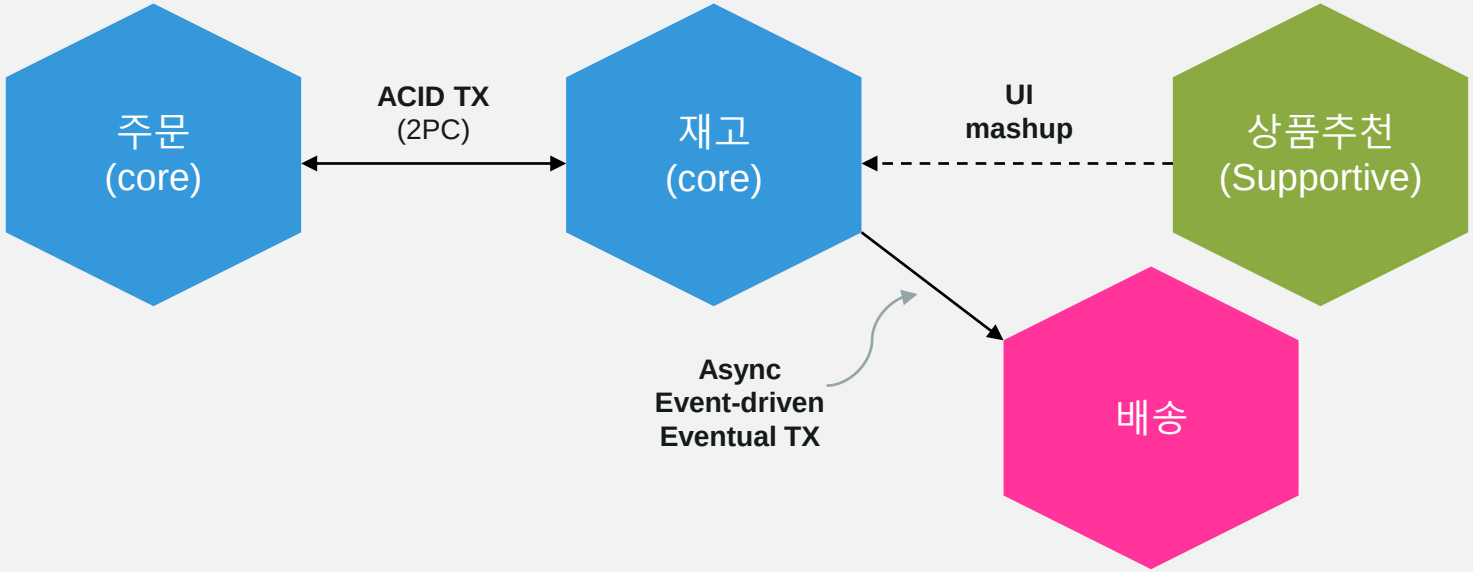
" 어느 마이크로 서비스의 인터페이스를 더 중요하게 관리할 것인가? "



Core Domain 간 (높은 서열끼리) 에는 Shared-kernel 도 허용가능하다.
하지만, 중요도가 낮은 서비스를 위해 높은 서열의 서비스가 인터페이스를 맞추는 경우는 없을 것이다.

마이크로 서비스간의 서열과 역학관계

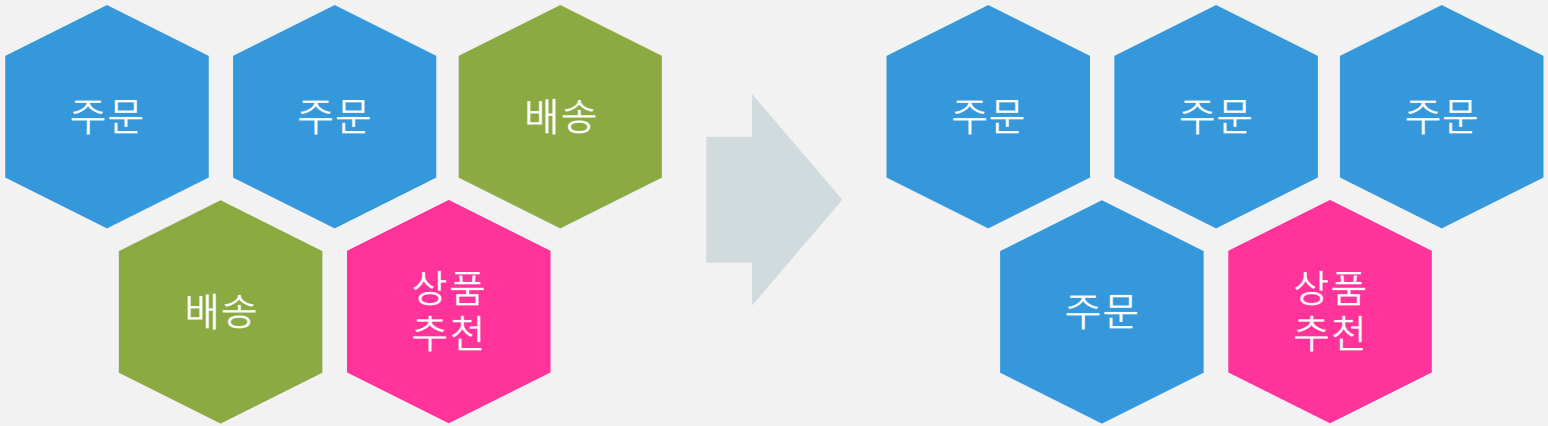
" 마이크로서비스간 트랜잭션의 묶음을 어떻게 할 것인가? "



배송기사가 없다고 주문을 안받을 것인가? 상품 추천이 안된다고 주문버튼을 안보여줄 것인가?
Core Domain 간에는 강결합이 요구되는 경우가 생길 수 있다.
하지만 우선순위가 떨어지는 비즈니스 기능을 위해 강한 트랜잭션을 연결할 이유는 하등에 없다.

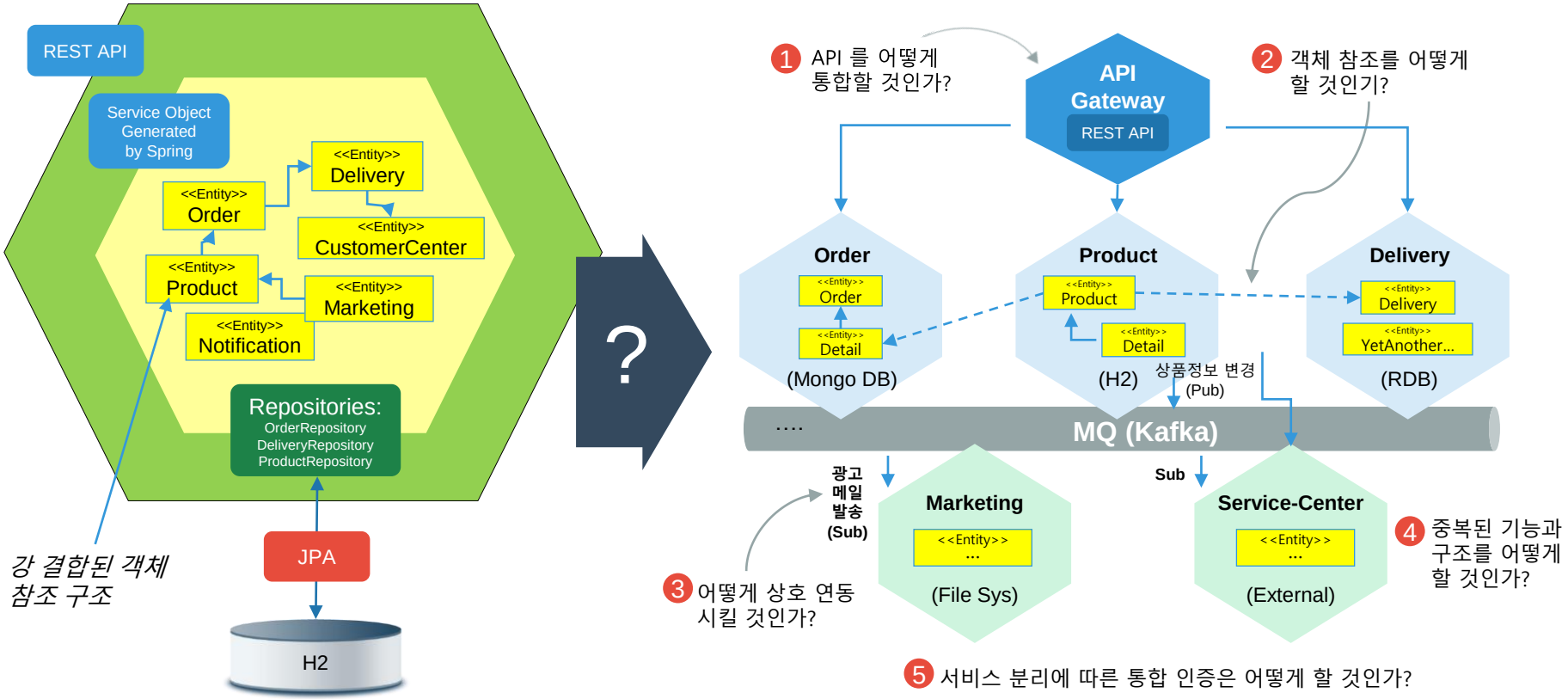
마이크로 서비스간의 서열과 역학관계

"손님이 많이 와서 집이 좁아졌다.. 무엇을 우선적으로 줄일 것인가?"



배송서비스는 야간에 올라와서 후에 처리되어도 된다.
주문이 몰려드는 시간에 주문을 못 받으면 배송은 의미가 없다

Issues in transforming Monolith to Microservices



1. API 를 어떻게 통합할 것인가?

- API Gateway

  → 진입점의 통일
  → URI Path-based Routing (기존에 REST 로 된경우 가능)

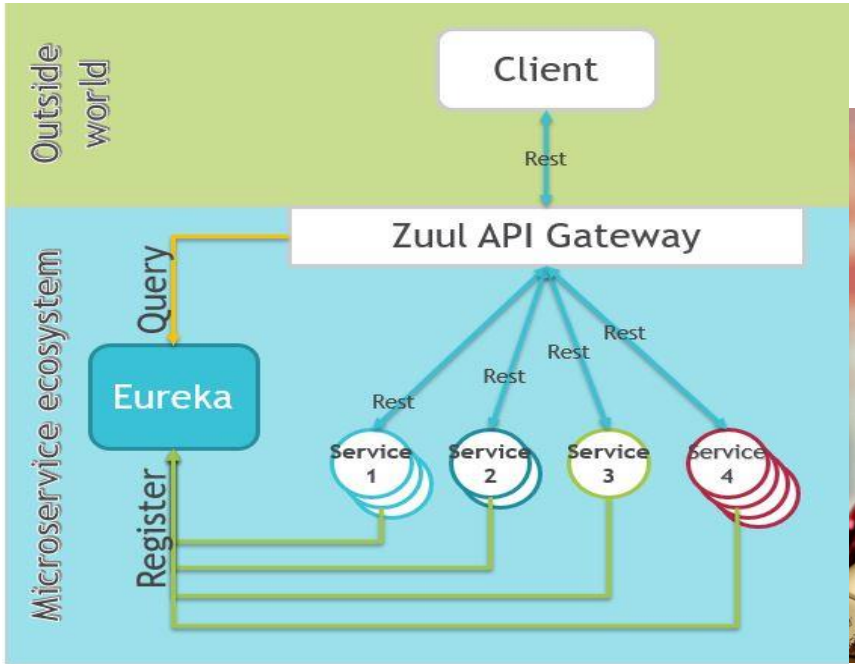
- Service Registry

  → API Gateway 가 클러스터 내의 인스턴스를 찾아가는 맵

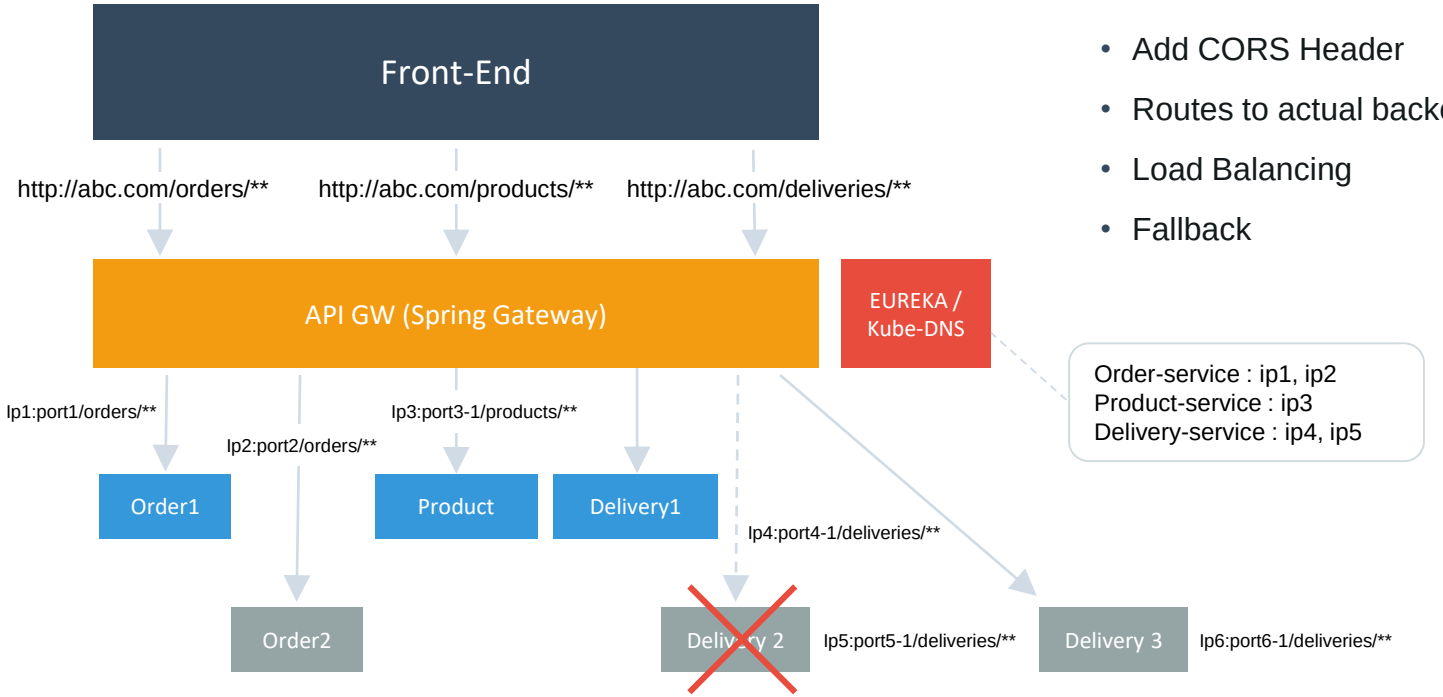
API Gateway : Edge Service

Acting like “Skin” to access our services :

- Re-Routes to multiple services
- Allows CORS
- Checks ACLs
- Prevent DDOS etc.



API Gateway : Routing, Securing and Load-balancing



- Add CORS Header
- Routes to actual backend service
- Load Balancing
- Fallback

By container orchestrator something

Configuring Spring Gateway

#application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: product
          uri: http://legacy:8080
          predicates:
            - Path=/product/**
        - id: order
          uri: http://legacy:8080
          predicates:
            - Path=/order/**
        - id: delivery
          uri: http://delivery:8080
          predicates:
            - Path=/deliveries/**
```


#application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: product
          uri: http://legacy:8080
          predicates:
            - Path=/product/**
        - id: order
          uri: http://order:8080
          predicates:
            - Path=/order/**
        - id: delivery
          uri: http://delivery:8080
          predicates:
            - Path=/deliveries/**
```

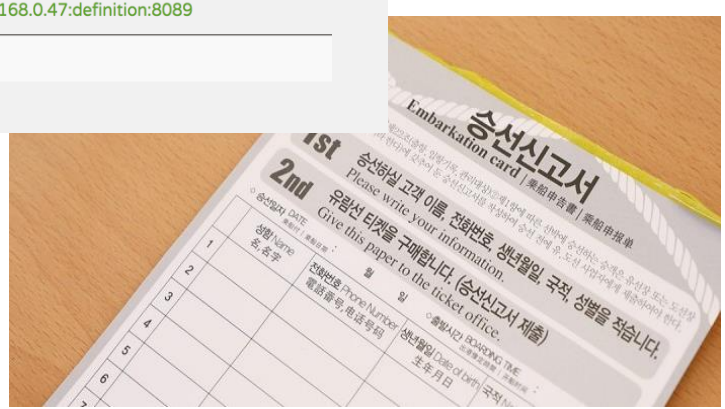
Service Registration : EUREKA or Kube-DNS

spring Eureka

HOME LAST 1000 SINCE STARTUP

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
BPM	n/a (1)	(1)	UP (1) - 192.168.0.47:bpm:8090
DEFINITION	n/a (2)	(2)	UP (2) - 192.168.0.47:definition:8091 , 192.168.0.47:definition:8089
ZUUL-ROUTER	n/a (1)	(1)	UP (1) - 192.168.0.47:zuul-router:8080

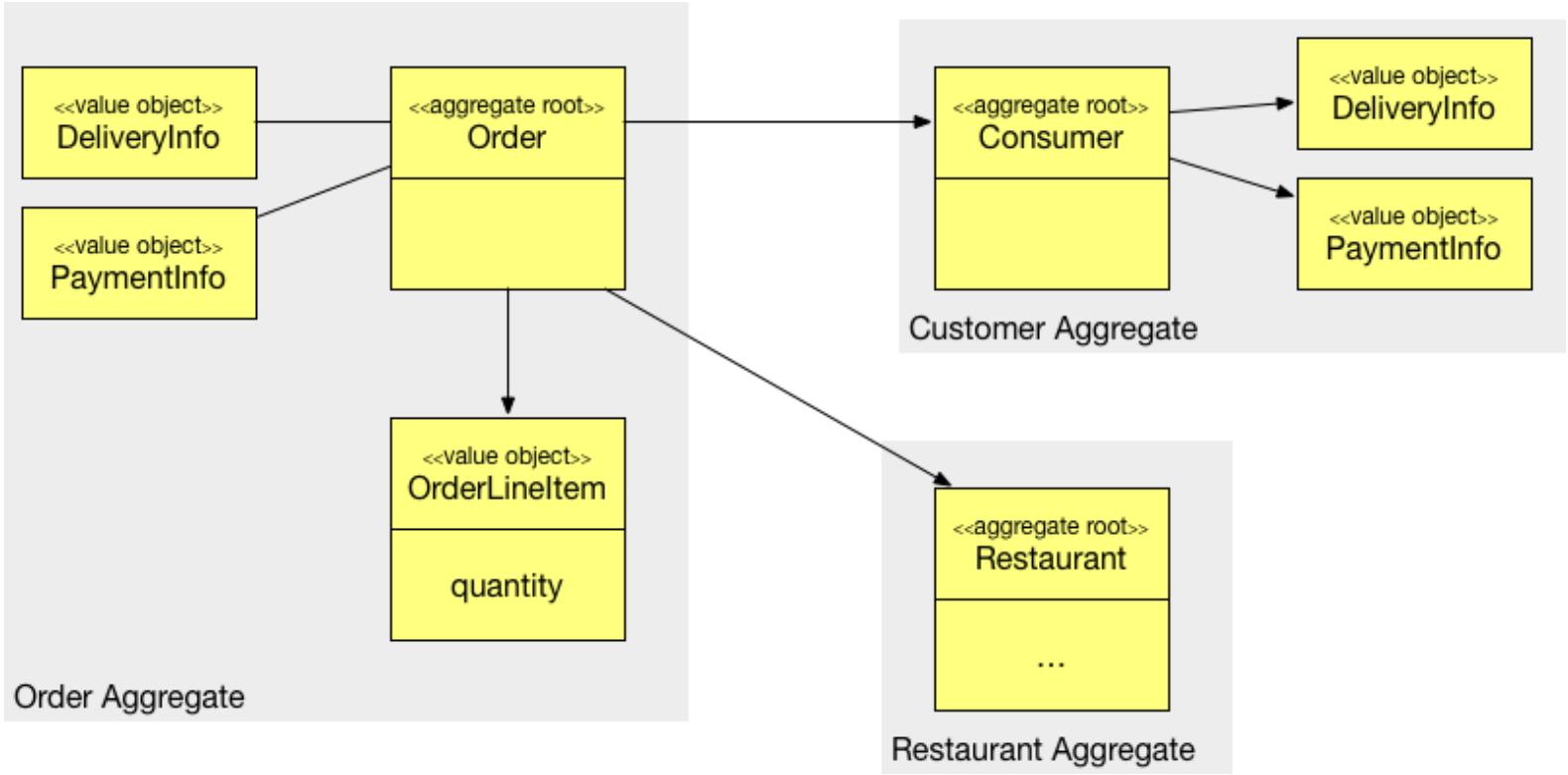


2. 객체 참조를 어떻게 할 것인가?

- 직접적 메모리 기반 객체 참조는 불가능

- 분리된 Aggregate 내부의 Entity 간에는 Key 값으로만 연결
  → Primary Key 를 통해서만 참조
  → HATEOAS link 를 이용

Aggregate Root를 통한 객체 참조



URI 를 통한 객체 참조

- **HATEOAS** (Hypertext As The Engine Of Application State)
  - HATEOAS is deemed the highest maturity level of REST. (https://martinfowler.com/articles/richardsonMaturityModel.html)
  - In the HATEOAS architecture, a client enters a REST application through a specific URL, and all future actions the client may take are discovered within resource representations returned from the server.
  - This self-contained discoverability can be a drawback for most service consumers who prefer API documentation.

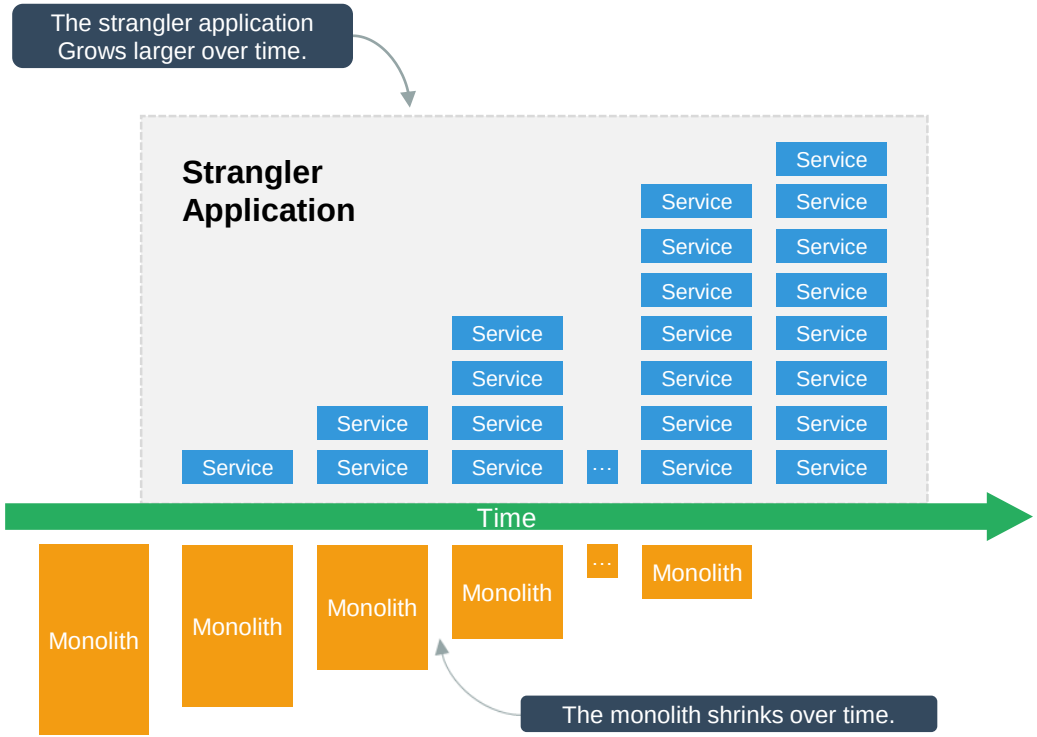
```
GET .../followers/ids.json?cursor=-1&screen_name=josdirksen

{
 "previous_cursor": 0,
 "id": {
   "name": "John Smit",
   "id": "12345678"
   "links" : [
   { "type: "application/vnd.twitter.com.user",
     "rel": "User info",
     "href": "https://.../user/12345678"},
   { "type": "application/vnd.twitter.com.user.follow",
     "rel": "Follow user",
     "href": "https://.../friendship/12345678"}
   ] // and add other options: tweet to, send direct message,
     // block, report for spam, add or remove from list
  }  // This is how you create a self-describing API.
}
```

3. 어떻게 (다시) 상호 연동시킬 것인가?

- Find the seams and replace with proxy
  → 기존 연계되던 이음매 객체 (interface) 를 찾아, 그에 상응하는 Façade, Proxy 로 대체

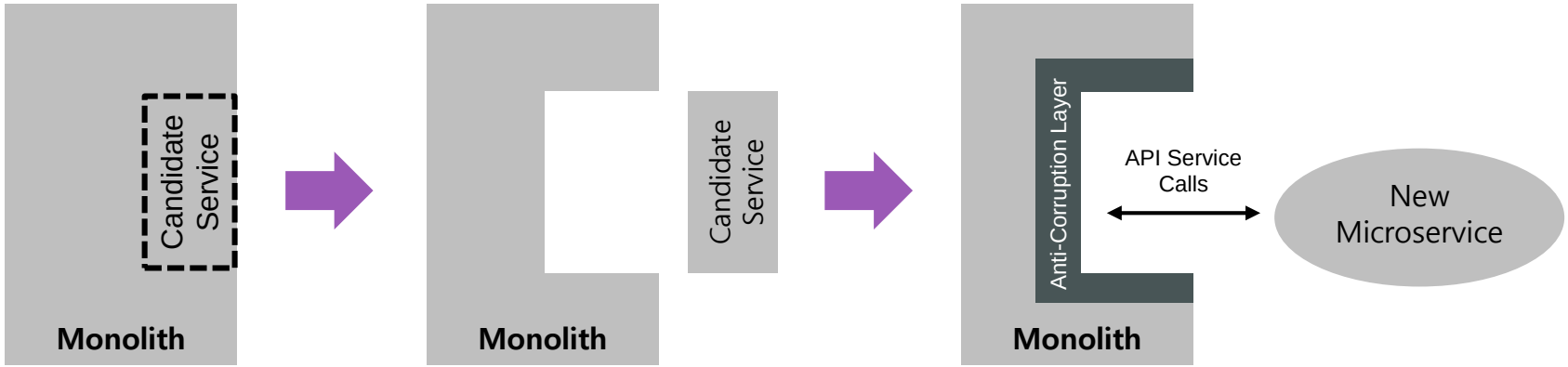
- Event Shunting
  → 레가시에서 이벤트를 퍼블리시하도록 전환, 신규 서비스가 주요 비즈니스 이벤트에 대하여 반응하도록 처리

Legacy Transformation – Strangler Pattern



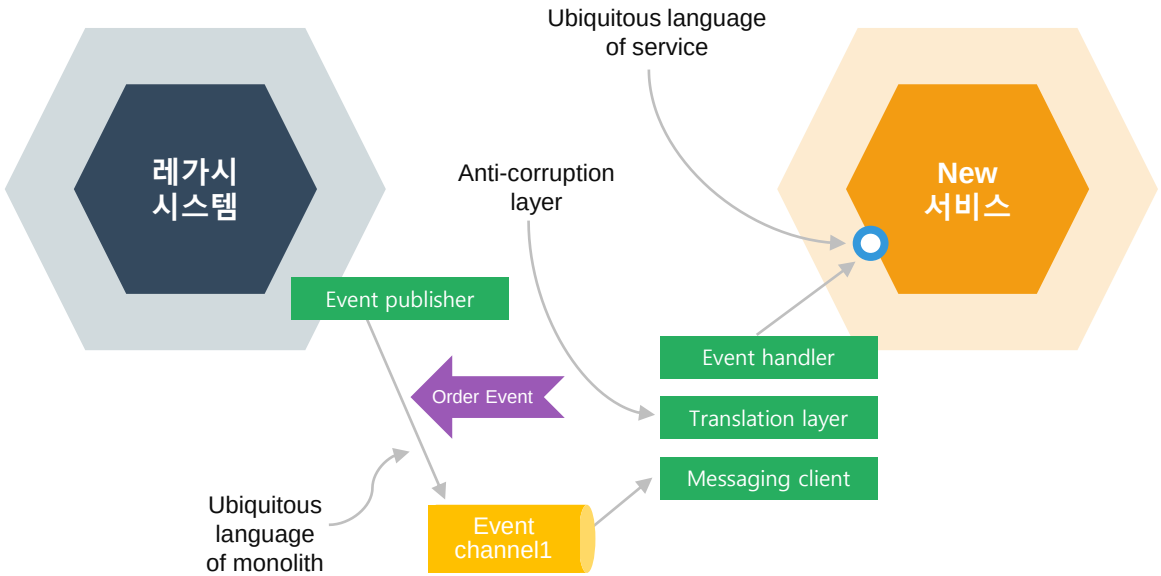
- Strangler 패턴으로 레가시의 모노리스 서비스가 마이크로서비스로 점진적 대체를 통한 Biz 임팩트 최소화를 통한 구조적 변화
- 기존에서 분리된 서비스 영역이 기존 모노리스와 연동 될 수 있도록 해주는 것이 필요
- 그 방법은 앞서 동기/ 비동기 방식이 채용될 수 있음
- 동기 – 레거시로 하여금 기존 소스코드 수정 요인이 높음, 따라서 이벤트 기반 비동기 연동 (CQRS) 추천

Find the seams and replace with proxy



- Determine candidate service from Monolith and strangle it out
- Convert candidate service to Microservice and Add Anti-Corruption Layer to enable communication with Monolith

Event Shunting



- **Aggregate 내 코드 주입**

  → 호출 코드 직접 주입, Hexagonal Architecture 의 손상
  → JPA 의 Lifecycle Annotation 사용

- **CDC (Change Data Capturing) 기능 사용**

  → DB 의 Change Log 를 Listening, Event 자동퍼블리시 하는 툴
  → Debezium, Eventuate Tram 등이 존재

Modified from Microservices Patterns, Chris Richardson, Manning, 2018

4. 중복된 기능과 데이터를 어떻게 할 것인가?

- 재사용하지 않고 중복 구현하는 것이 MSA 스러운 것
  → 재사용 통한 경제성(SOA사상, 디펜던시발생)과 자율적 창발 (낮은 간섭과 빠른 출시)의 트레이드 오프에서 후자의 전략을 선택
  → Utility service X, DRY(Don’t Repeat Yourself) 룰 X
  → Polyglot Persistency

- 예외상황 : 데이터 참조에 intensive 한 서비스 (인증정보 등)는 Utility 서비스 성격으로 구현 가능함

5. 서비스 분리에 따른 통합인증은 어떻게 할 것인가?

- **OAuth2.0**

  → 웹, 모바일 어플리케이션에서 타사의 API 권한 획득을 위한 프로토콜
  → Google, facebook 등을 통한 인증 위임

- **JWT(Json Web Token) 토큰**

  → Header, Claim Set, Signature로 구성
  → 요청 헤더에 Authorization 값을 담아서 서버로 송신

OAuth Authorization

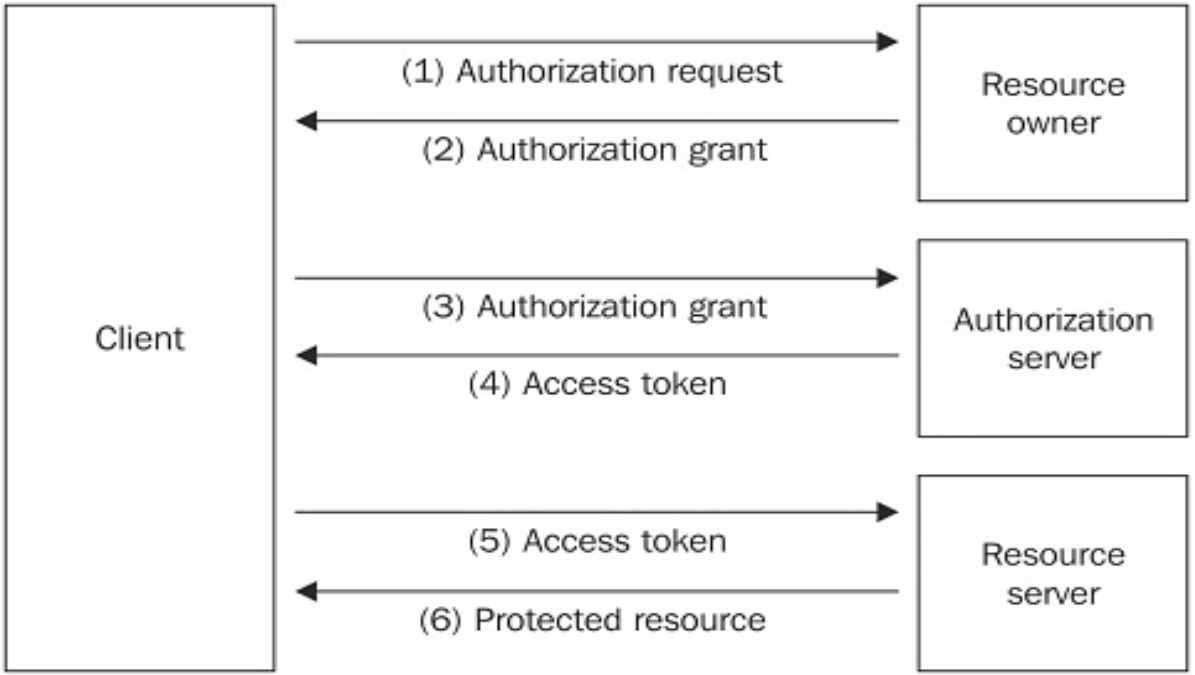
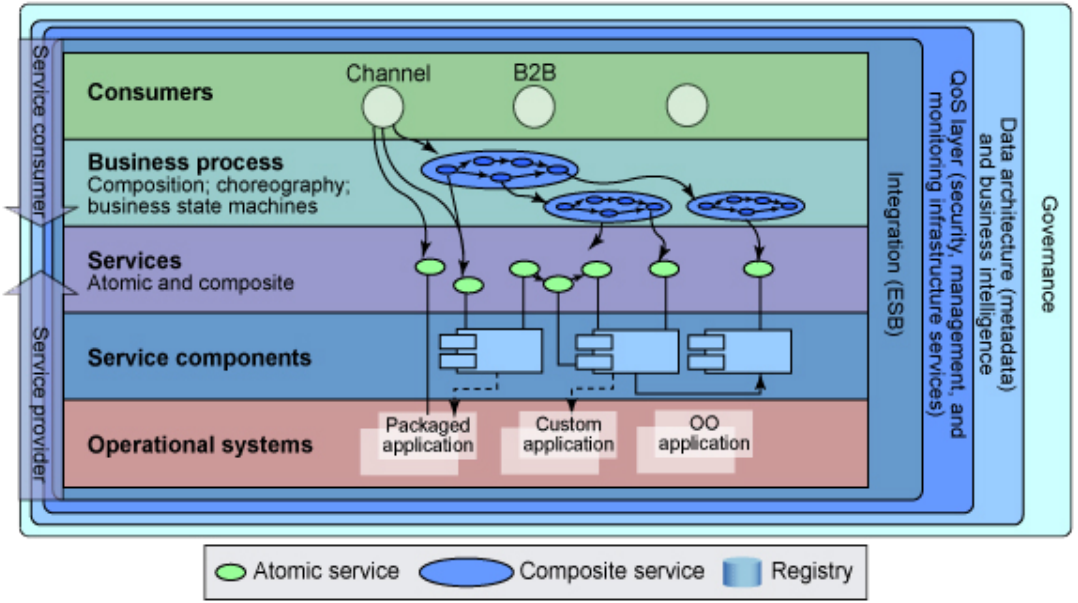


Table of content

Microservice and
Event-storming-Based
DevOps Project

UEngine

Integration Patterns



By UI

By Request-Response

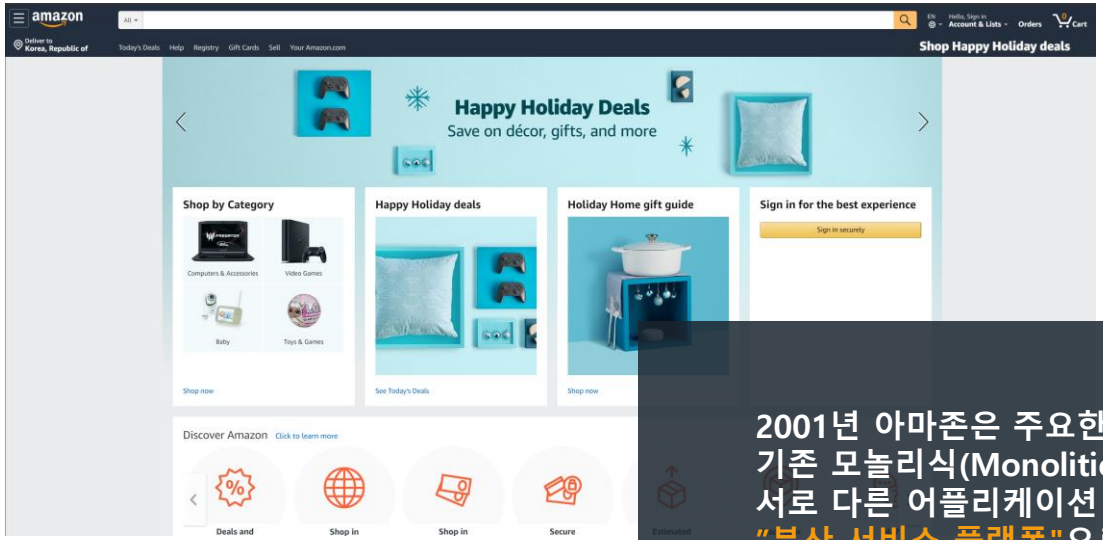
By Event-driven Architecture

Service Composition with User Interface

- Extended Role of Front-end in MSA Architecture: Service Aggregation
- Why MVVM?
- W3C Web Components Standard – Domain HTML Tags
- Implementation: Polymer and VueJS
- Another: ReactJS and Angular2
- Micro-service Mashups with Domain Tags: i.e. IBM bluetags
- Cross-Origin Resource Sharing
- API Gateway (Netflix Zuul)

https://www.youtube.com/watch?v=djQh8XKRzRg
https://github.com/IBM-Cloud/bluetag

- 아마존 닷컴은 다양한 서비스 제공과 효율적인 운영 환경 전환을 위해 분산 서비스 플랫폼으로 전환 하였습니다.



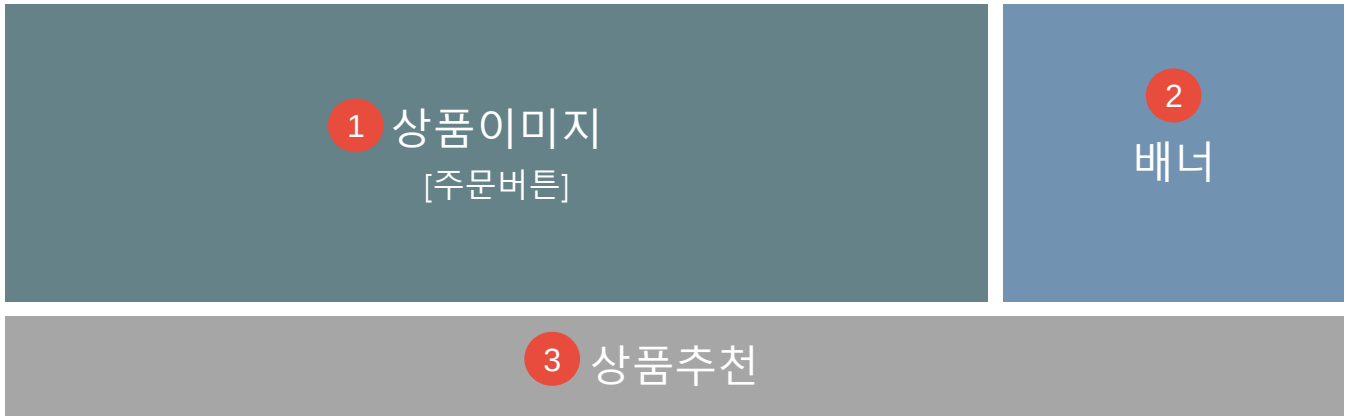
amazon

2001년 아마존은 주요한 아키텍쳐 변화가 있었는데
기존 모놀리식(Monolitic) 기반에서
서로 다른 어플리케이션 기능을 제공하는
"분산 서비스 플랫폼"으로 변화 하였습니다.

현재의 Amazon.com의 첫 화면에 들어 온다면,
그 페이지를 생성하기 위해
100여개가 넘는 서비스를 호출하여 만들고 있습니다.

Client-side Rendering : 장애 전파 회피

“ 다음중 가능한 빨리 로딩되어야 하고, 문제가 없어야 할 화면 영역은? ”



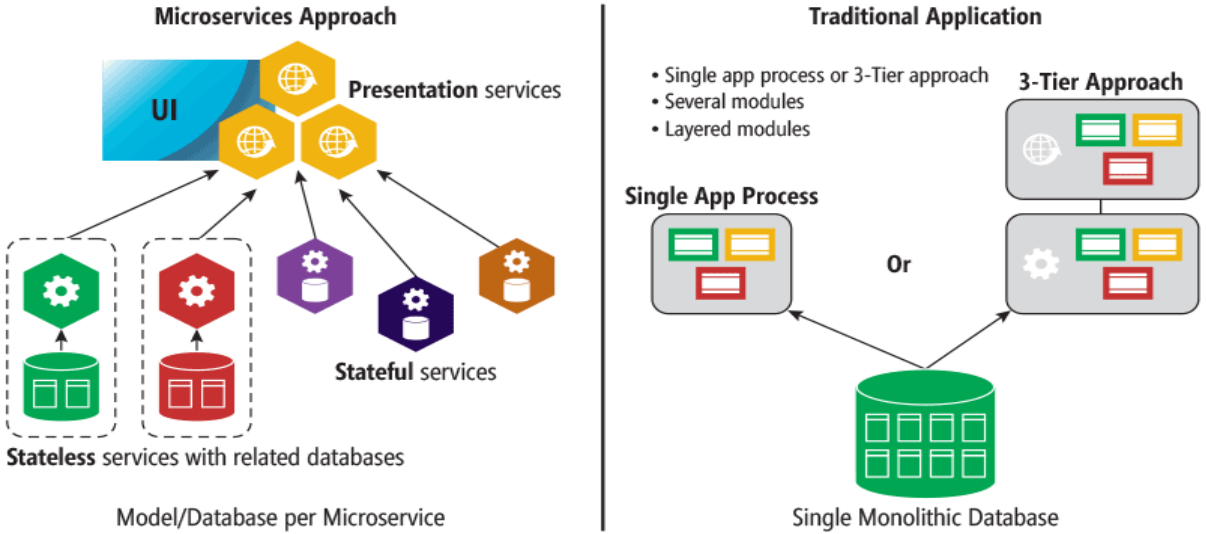
Server-side Rendering 은 모든 화면의 콘텐츠가 도달해야만 화면을 보여줄 수 있지만,
Client-side Rendering 은 먼저 데이터가 도달한 화면부터 우선적으로 표출할 수 있다.

광고배너가 나가지 못한다고 주문을 안받을 것인가?

그걸 원하지 않는다면 **AJAX / MVVM 같은 Client-side Rendering 기술에 주목하자.**

Microservice Integration with by UI

- 서비스의 통합을 위하여 기존에 Join SQL 등을 사용하지 않고 프론트-엔드 기술이나 API Gateway 를 통하여 서비스간 데이터를 통합함
- 프론트엔드에서 데이터를 통합하기 위한 접근 방법으로는 W3C 의 Web Components 기법과 MVVM 그리고 REST API 전용 스크립트가 유용함



W3C Web Components

- Custom elements
- HTML imports
- Template
- Shadow DOM

```
<style>
   style for A
   style for B
   style for C
</style>

<html>
   element for A
   element for B
   element for C
</html>

<script>
    script for A
    script for B
    script for C
</script>
```



```
#index.html
<A> <A>
<B> <B> <B>
<C>
```

```
#A.html
<style>
   style for A
</style>

<html>
  element for A
</html>

<script>
   script for A
</script>
```

```
#B.html
<style>
   style for B
</style>

<html>
  element for B
</html>

<script>
   script for B
</script>
```

```
#C.html
<style>
   style for C
</style>

<html>
  element for C
</html>

<script>
   script for C
</script>
```

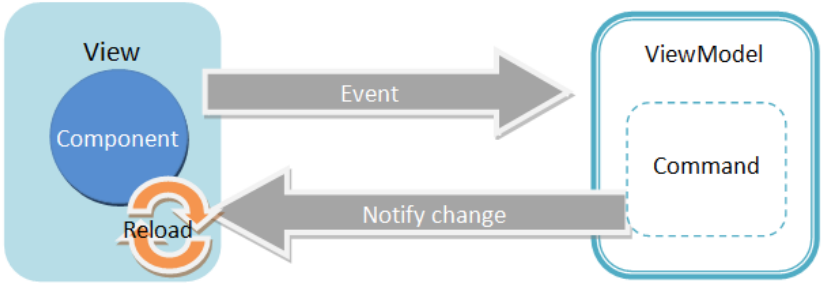
Custom tag for Domain Class: <product> tag

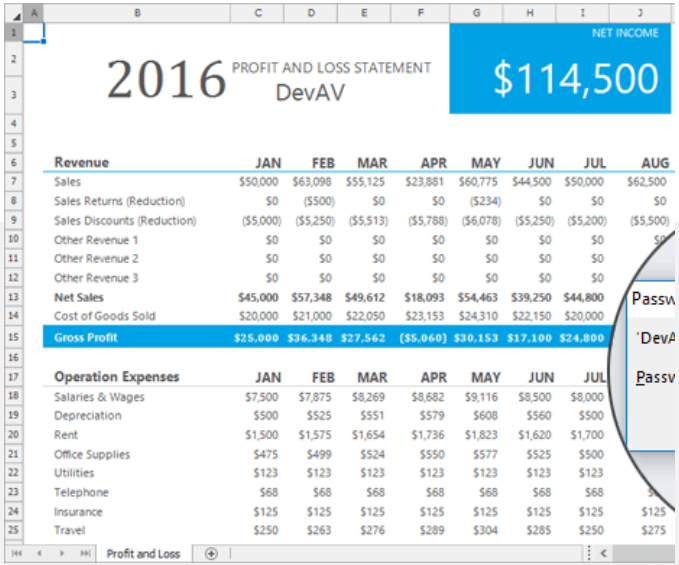
```
<table>
   <tr
         v-for="(item, index) in props.items"
         :key="item.id"
   >
      <product
            v-model="props.items[index]"
            @inputBuy="showBuy"
            @inputEdit="showEdit"
      ></product>

   </tr>
</table>
```

Data binding to Domain Class Tags

```
<product
        v-model="props.items[index]"
        @inputBuy="showBuy"
        @inputEdit="showEdit"
></product>

<script>
  methods: {
      getProdList() {
        var me = this
        me.$http.get(`${API_HOST}/products`).then(function (e) {
          me.items = e.data._embedded.products;
        })
      }
  }
</script>
```

MVVM





2016 PROFIT AND LOSS STATEMENT DevAV

NET INCOME $114,500

Revenue	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG
Sales	$50,000	$63,098	$55,125	$23,881	$60,775	$44,500	$50,000	$62,500
Sales Returns (Reduction)	$0	($500)	$0	$0	($234)	$0	$0	$0
Sales Discounts (Reduction)	($5,000)	($5,250)	($5,513)	($5,788)	($6,078)	($5,250)	($5,200)	($5,500)
Other Revenue 1	$0	$0	$0	$0	$0	$0	$0	$0
Other Revenue 2	$0	$0	$0	$0	$0	$0	$0	
Other Revenue 3	$0	$0	$0	$0	$0	$0	$0	
Net Sales	**$45,000**	**$57,348**	**$49,612**	**$18,093**	**$54,463**	**$39,250**	**$44,800**	
Cost of Goods Sold	$20,000	$21,000	$22,050	$23,153	$24,310	$22,150	$20,000	
Gross Profit	**$25.000**	**$36,348**	**$27,562**	**($5,060)**	**$30,153**	**$17,100**	**$24,800**	

Operation Expenses	JAN	FEB	MAR	APR	MAY	JUN	JUL
Salaries & Wages	$7,500	$7,875	$8,269	$8,682	$9,116	$8,500	$8,000
Depreciation	$500	$525	$551	$579	$608	$560	$500
Rent	$1,500	$1,575	$1,654	$1,736	$1,823	$1,620	$1,700
Office Supplies	$475	$499	$524	$550	$577	$525	$500
Utilities	$123	$123	$123	$123	$123	$123	$123
Telephone	$68	$68	$68	$68	$68	$68	$68
Insurance	$125	$125	$125	$125	$125	$125	$125
Travel	$250	$263	$276	$289	$304	$285	$250

Profit and Loss

Frameworks supporting Web Components

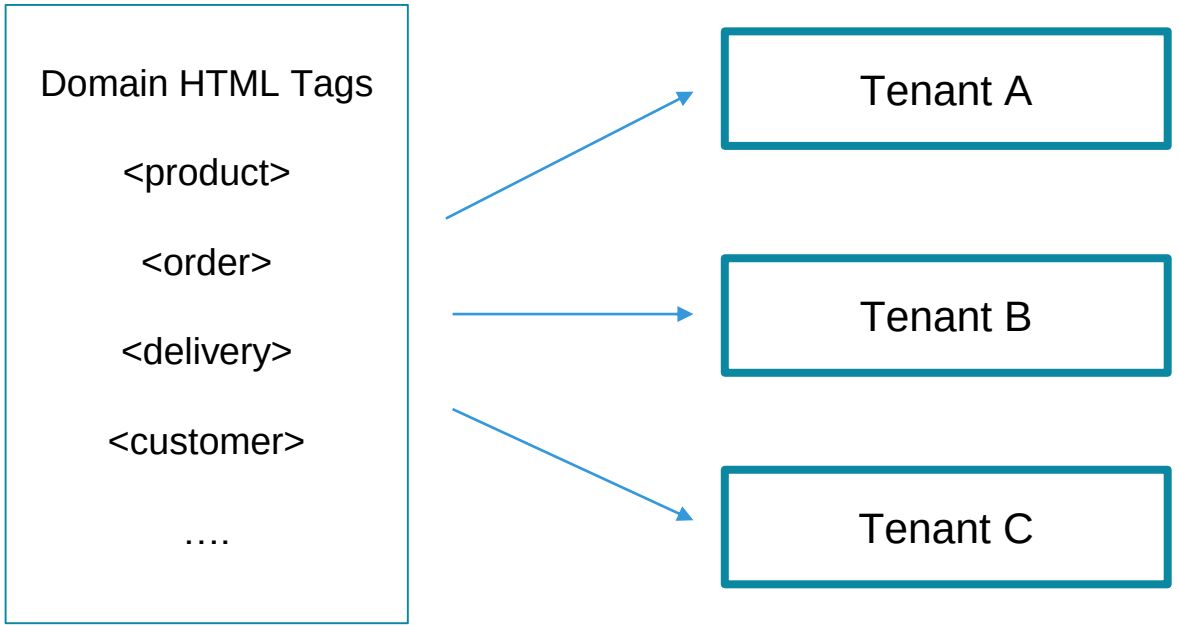
Polymer

VueJS

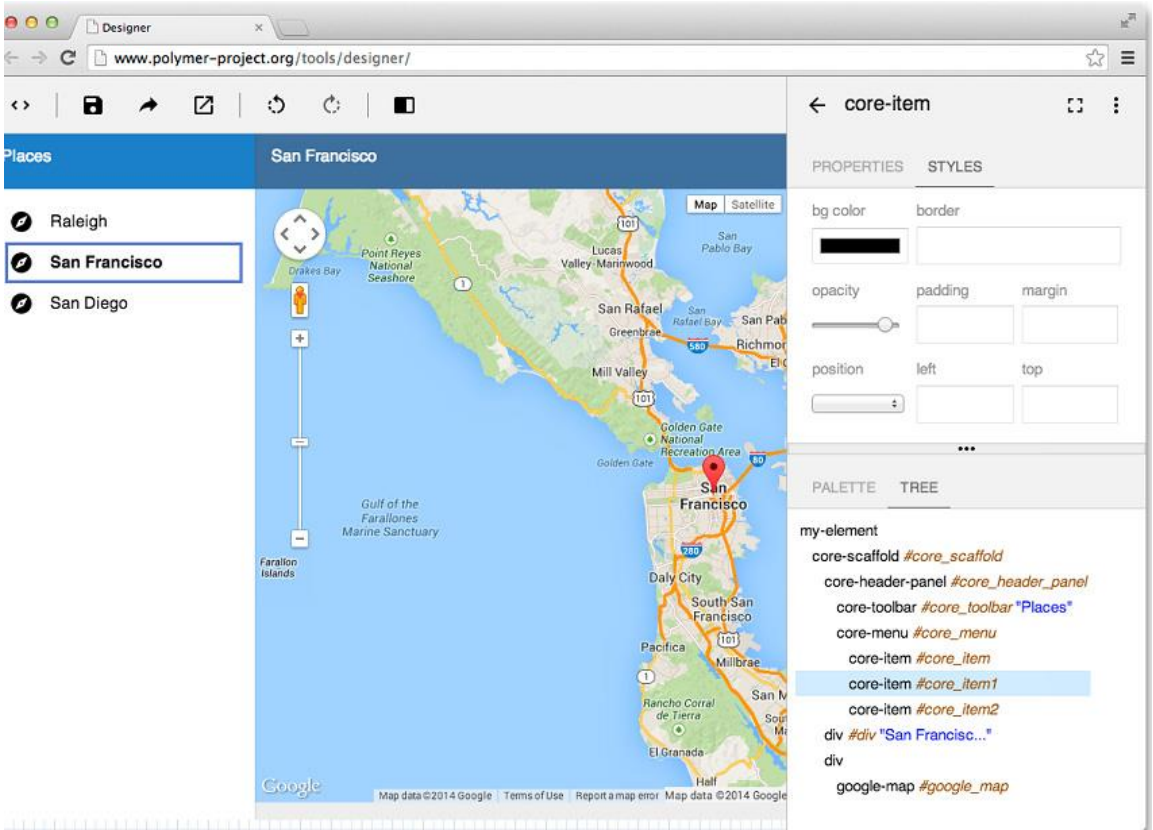
- Provides Cohesive Component Model
- Component Composition by HTML markup
- Dynamic Data Binding
- Responsive Web by Material Design
- Standard

** 비표준 : Angular JS, React

Tenant UI Customization (Composition) by Tag composition



UI Mashups by domain tags

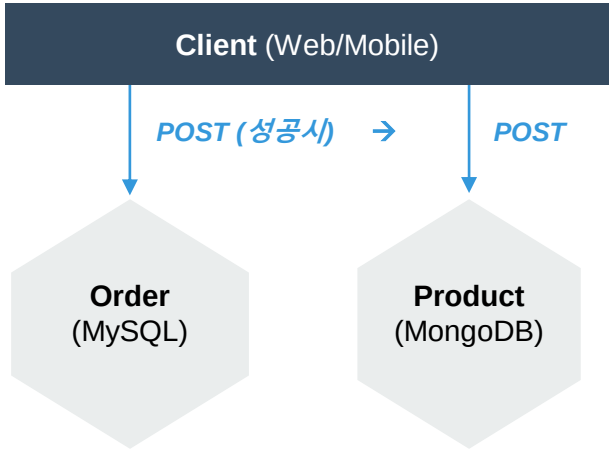


- https://www.ibm.com/developerworks/library/wa-polymer/

Transaction Problem in UI composition

In-consistent Consistent

- 서비스구성 (클라이언트가 순차 호출)



- 조회화면

주문량 : 0 , 재고량 : 10

주문량 : 1 , 재고량 : 10

주문량 : 2 , 재고량 : 9

주문량 : 3 , 재고량 : 8

둘다 순조롭게 호출 성공시 일치

불일치 영원히 생길 수 있음!!

Table of content

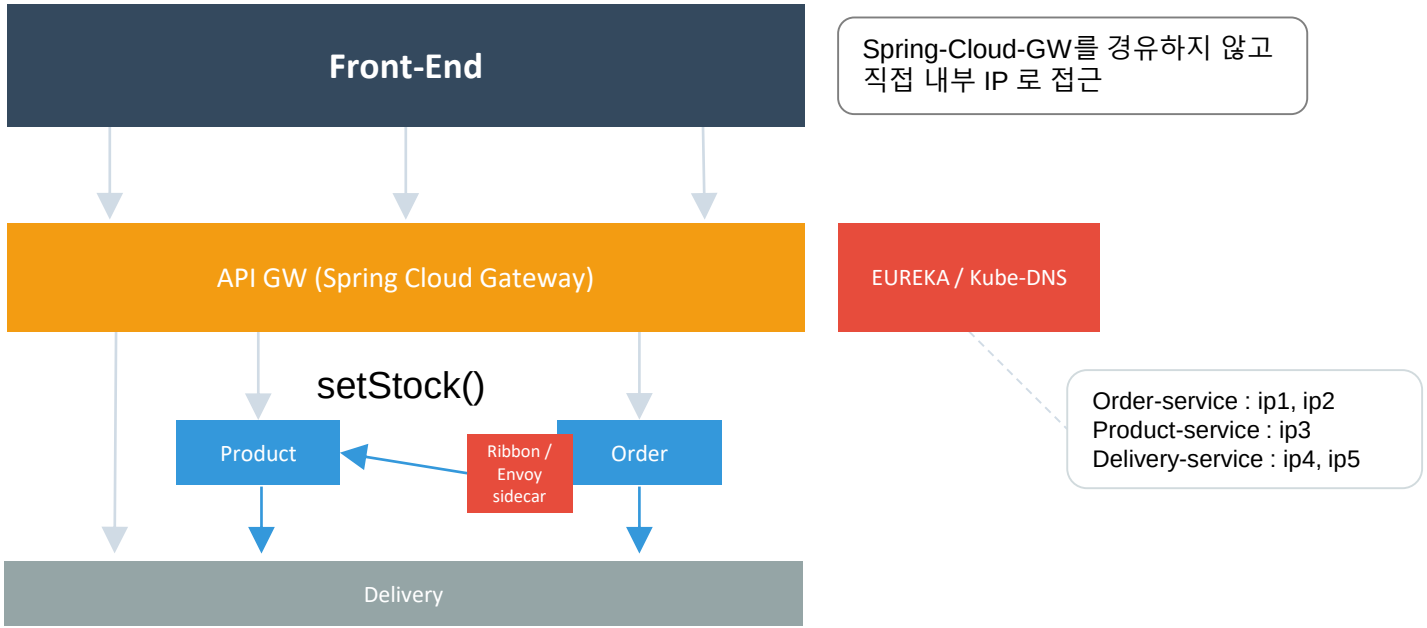
Microservice and
Event-storming-Based
DevOps Project

UEngine

Service Integration by Request-Response

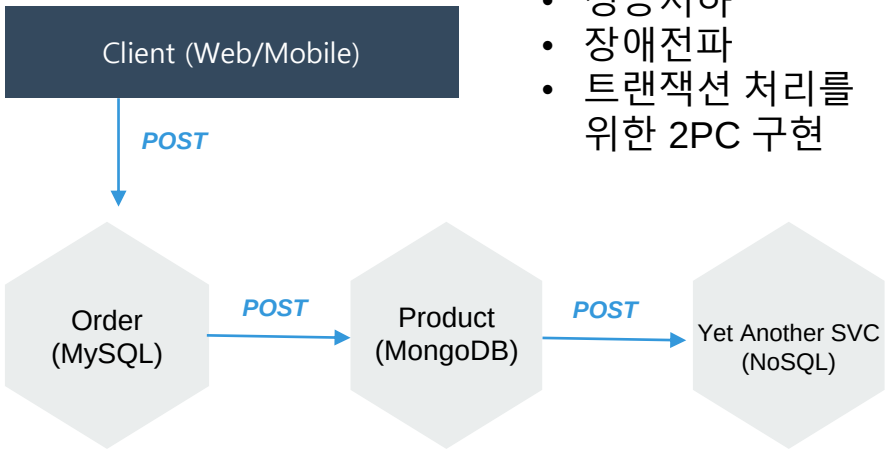
- Inter-microservices call requires client-side discovery and load-balancing
- Netflix Ribbon and Eureka
- Hiding the transportation layer:
  Spring Feign library and JAX-RS
- Circuit Breaker Patterns

Inter-microservices Call – Request/Response



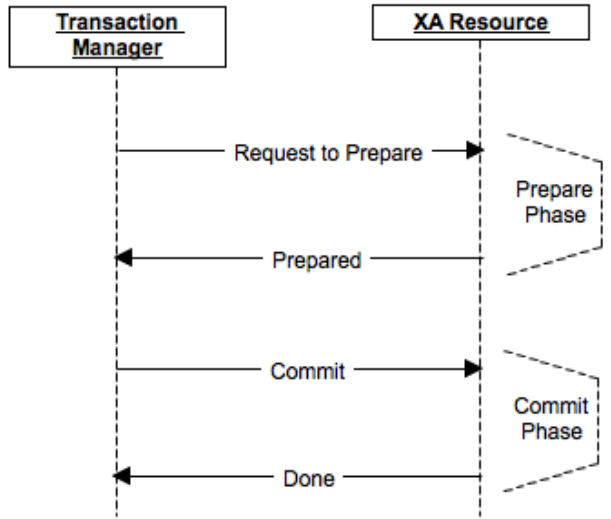
Issues in Request-Response model

서비스구성 (Request-Response, Sync 호출)

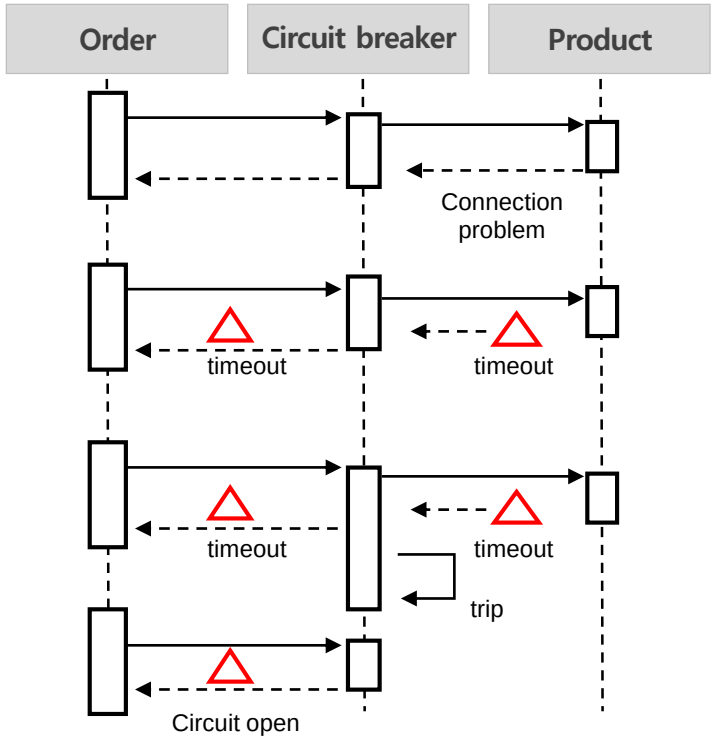


- 성능저하
- 장애전파
- 트랜잭션 처리를 위한 2PC 구현

Atomic Transaction / 2PC



장애전파 차단: 서킷브레이커 패턴



하나의 트랜잭션이 가능한 블로킹이
발생하지 않도록 미리 차단, Fail-Fast 전략

➔ 메모리 사용 폭주 막음

장애 감지 시, 차단기 작동(Open)

↓

일시 동안 Fallback으로 서비스 대체

↓

일부 트래픽으로 서비스 정상여부 확인

↓

정상 확인 시, 차단기 해제(Close)

Consumer Code: Order

```
@Entity
public class Order{

   @PrePersist  // 주문이 저장되기 전에
    public void beforeSave() {

            Product product = ProductService.getProduct(getProductId()); //제품정보를 얻어서

            if(product.getStock() < getQty()){   // 재고량이 주문량 보다 적다면
               throw new RuntimeException("No Available stock!" );    // 재고 부족이라고 오류를 던져라
            }
    }

    @PostPersist   // 주문이 저장된 후에는
    public void afterSave() {
        ProductService.setStock(getProductId(), getQty()*-1);     //재고량을 수정하라
    }
}
```

marketingService.collectOrderData(this);

FeignClient

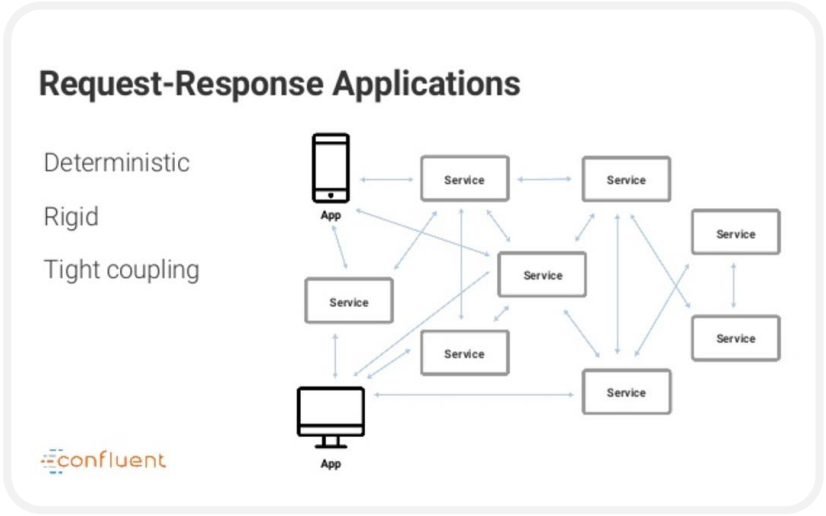
---- url 을 기반으로 Kube-DNS 를 찾아감

```
@FeignClient(name = ”product”, url=“http://product:8080”)
public interface ProductService {

 @RequestMapping(method = RequestMethod.GET, path="/products/{productId}")
   Product setStock(@PathVariable(”productId") Long productId, int stockChange);

}
```

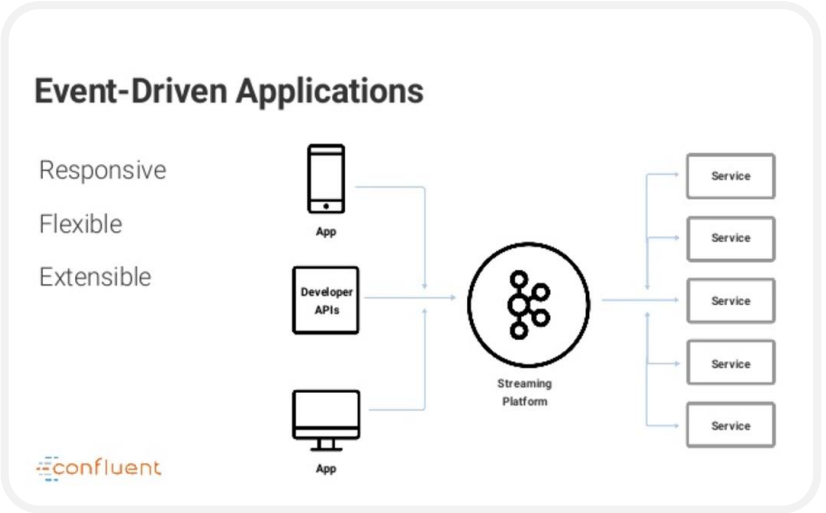
Data Mutation by Event-driven Model

- Event-driven Approach
- ACID Tx. vs Eventual Tx.
- Business Transaction / Compensation (Saga) Pattern

Microservice Integration with Event-driven Architecture



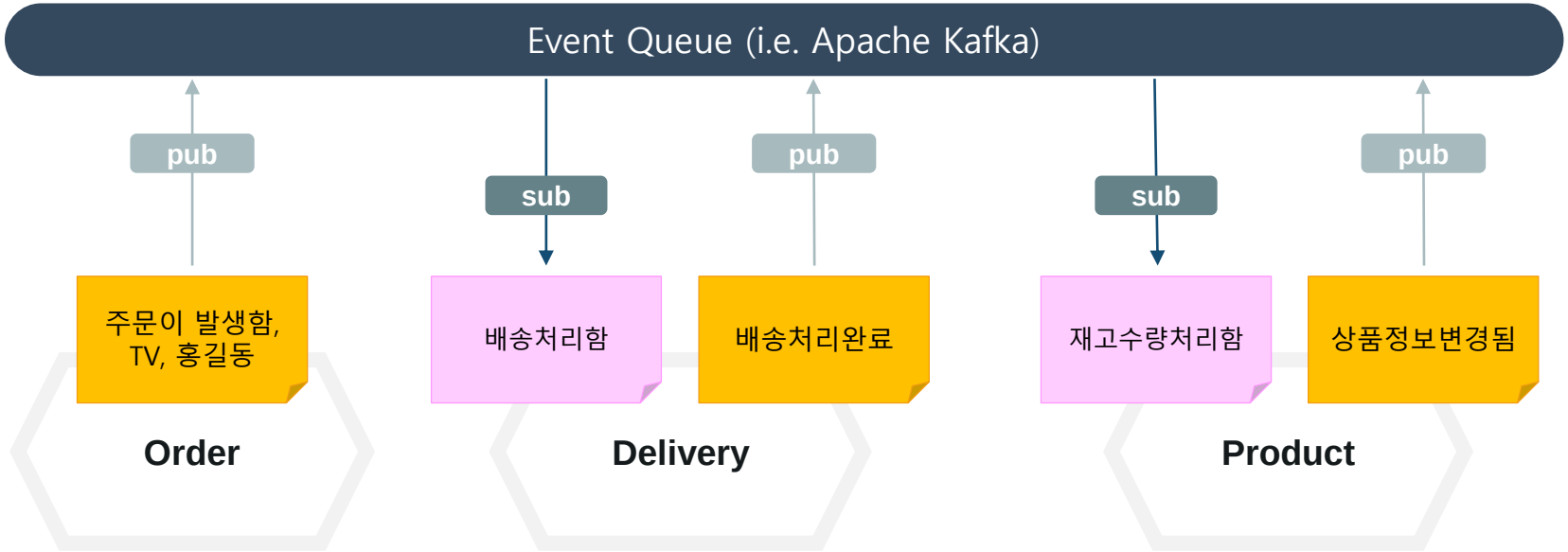
VS



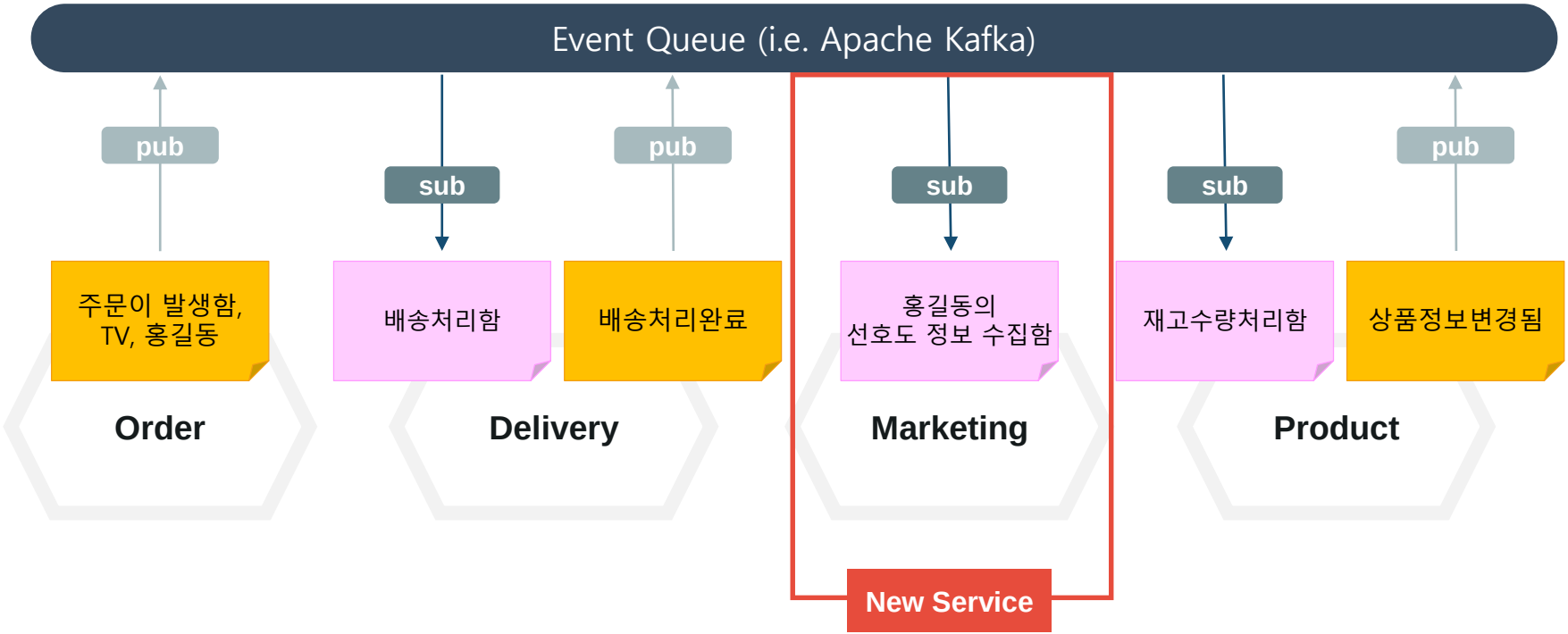
- Point to Point → Spagetti network
- Blocking Model
- System fault spread

- Broadcasting
- Non-Blocking Model
- System fault isolation

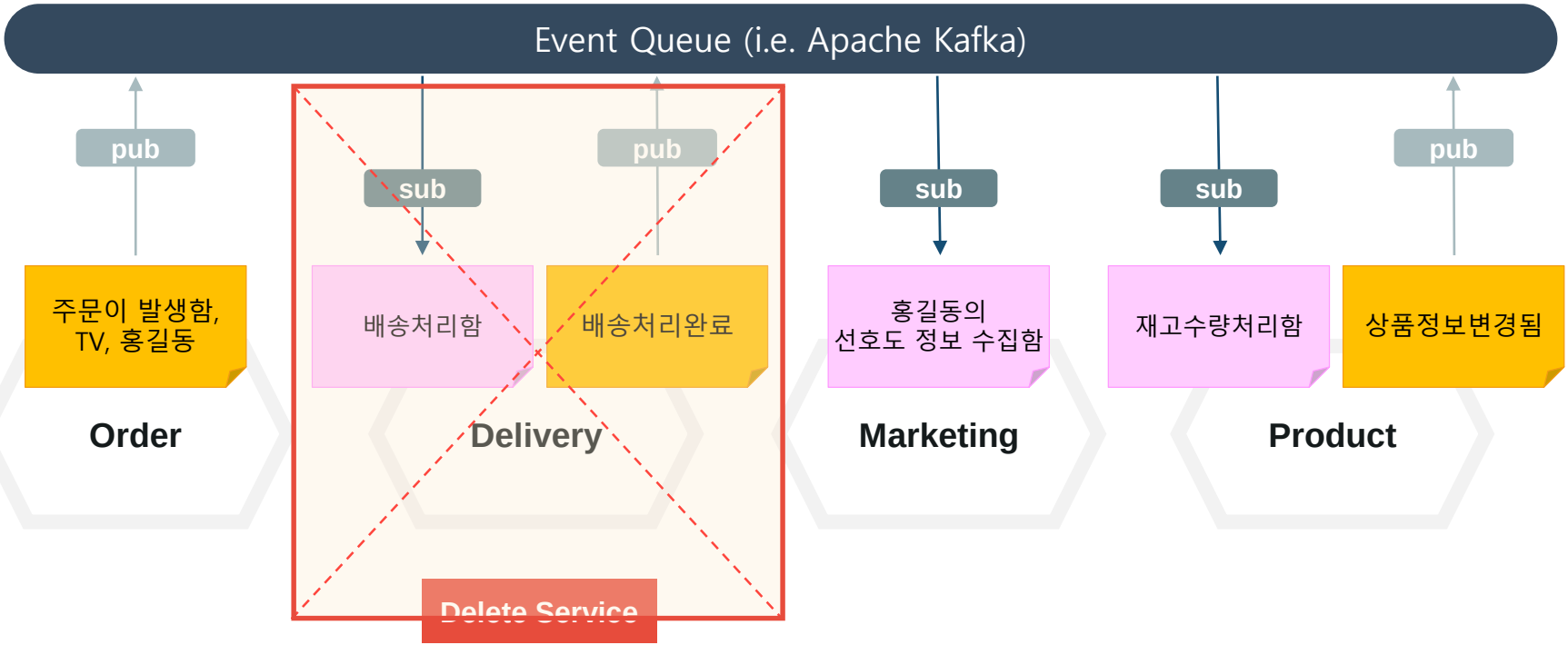
Inter-microservice Call - Event Publish / Subscribe (PubSub)



Adding a new service



Removing a service



Event Publisher : Order

```
@Entity
public class Order {
 …

    @PostPersist   // 주문이 저장된 후에
    private void publishOrderPlaced() {
        OrderPlaced orderPlaced = new OrderPlaced();  // 주문이 들어온 사실을 이벤트로 작성

        orderPlaced.setOrderId(id);
            …

        kafka.send(orderPlaced);   // 메시지 큐에 주문이 들어왔음을 신고
    }
}
```

Event Consumer : Delivery

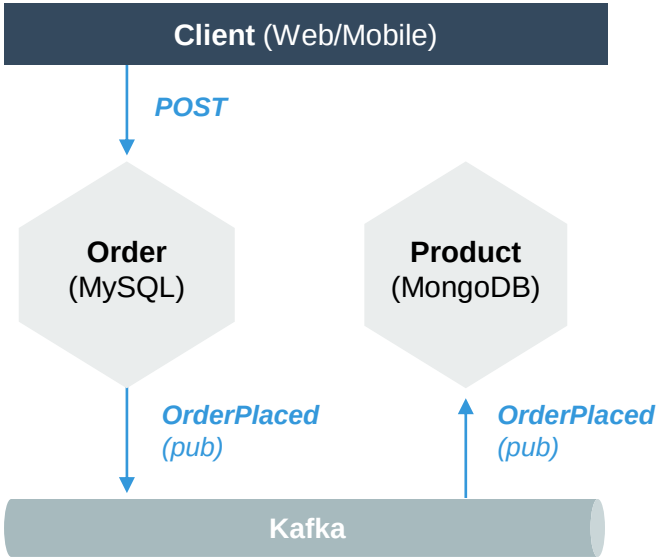
```
@KafkaListener(topics = ”shopping")   // shopping 토픽의 이벤트를 수신
public void onOrderPlaced(…) {   // OrderPlaced 이벤트가 들어오면..

    deliveryService.start(order);  // 해당 주문건의 배송 준비를 시작

    Kafka.send(new OrderShipped(delivery));  // 배송 준비 한 후에는 주문배송처리 됨을 메시지 큐에 신고
}
```

Event Consumer : Product

```
@KafkaListener(topics = ”shopping")  // 쇼핑 토픽을 수신
public void onOrderPlaced(…) {  // 주문이 들어오는 이벤트가 오면

    // 해당 상품의 재고량을 주문량 만큼 줄여서 저장
    Product product =
productRepository.findById(orderPlaced.getProductId()).get();
    product.setStock(product.getStock() - orderPlaced.getQuantity());

    productRepository.save(product);

}
```

Eventual TX를 통한 분산 트랜잭션 처리

- 서비스구성 (Eventual TX)



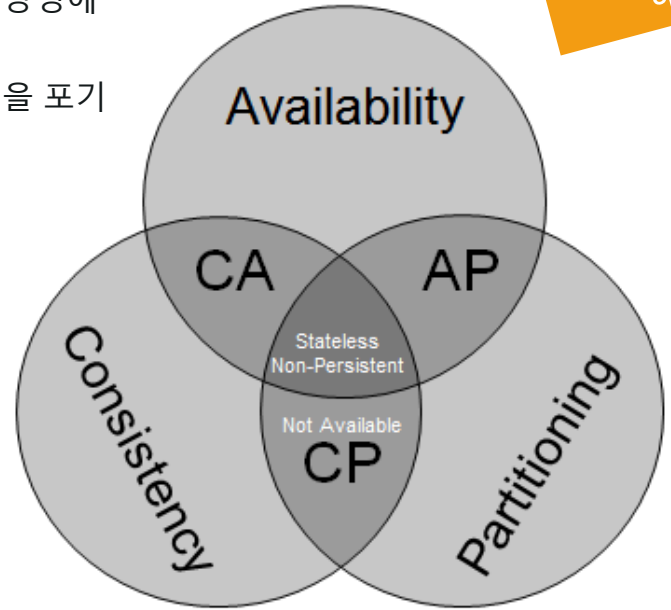
- 조회화면

여기서 선택의 길을 만남...

- 내 서비스에서 데이터 불일치가 얼마나 미션 크리티컬 한가?
- 크리티컬 하다고 응답한다면, 내 서비스의 성능과 확장성에 비하여 크리티컬 한가?
- 성능과 확장성 보다 크리티컬 하다면, 성능과 확장성을 포기 할 수 있는가?

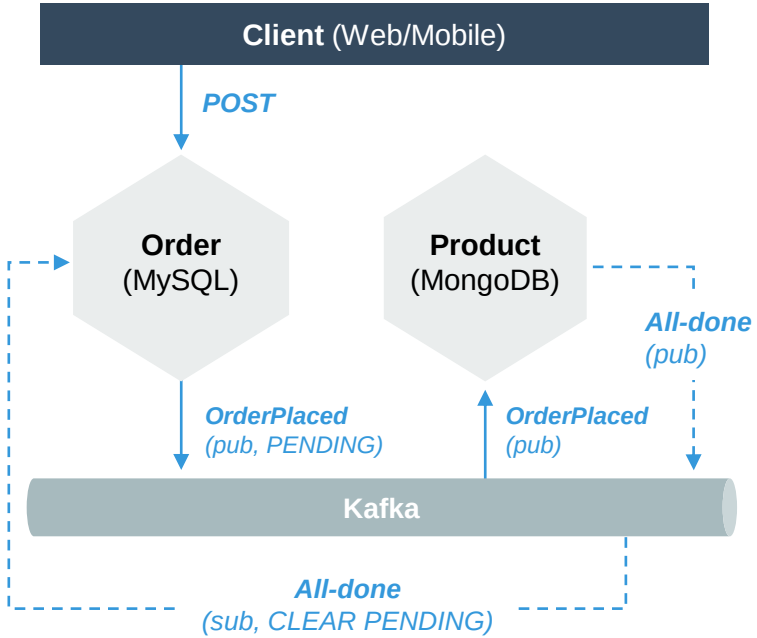
"성공한 내 서비스의 경쟁자들은 무엇을 포기했는가?"

➔ 강력한 의사결정 필요
(무엇으로 태어날 것인가...)



Eventual TX – 불일치가 Mission Critical 한 경우

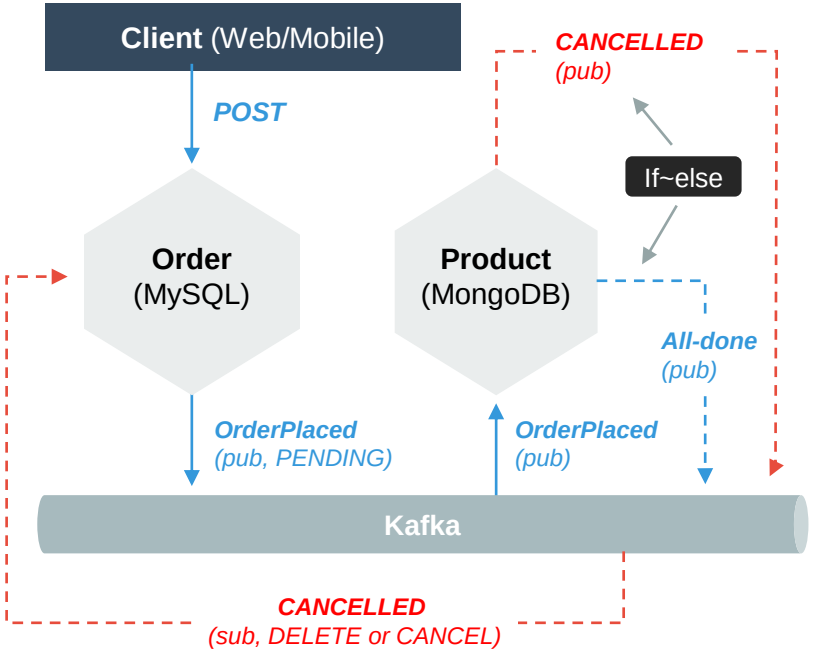
- 서비스구성 (Eventual TX)



- 조회화면

Eventual TX – Rollback (Saga Pattern)

- 서비스구성 (Eventual TX)



- 조회화면

Consistent In-consistent

주문량 : 0 , 재고량 : 10

주문량 : 1 , 재고량 : 10 (PENDING)

주문량 : 1 , 재고량 : 9

주문량 : 2 , 재고량 : 9 (PENDING)

주문량 : 2 , 재고량 : 8 → **취소됨**

주문량 : 2 , 재고량 : 9

복구 & 결국 일치

Resources on Sagas

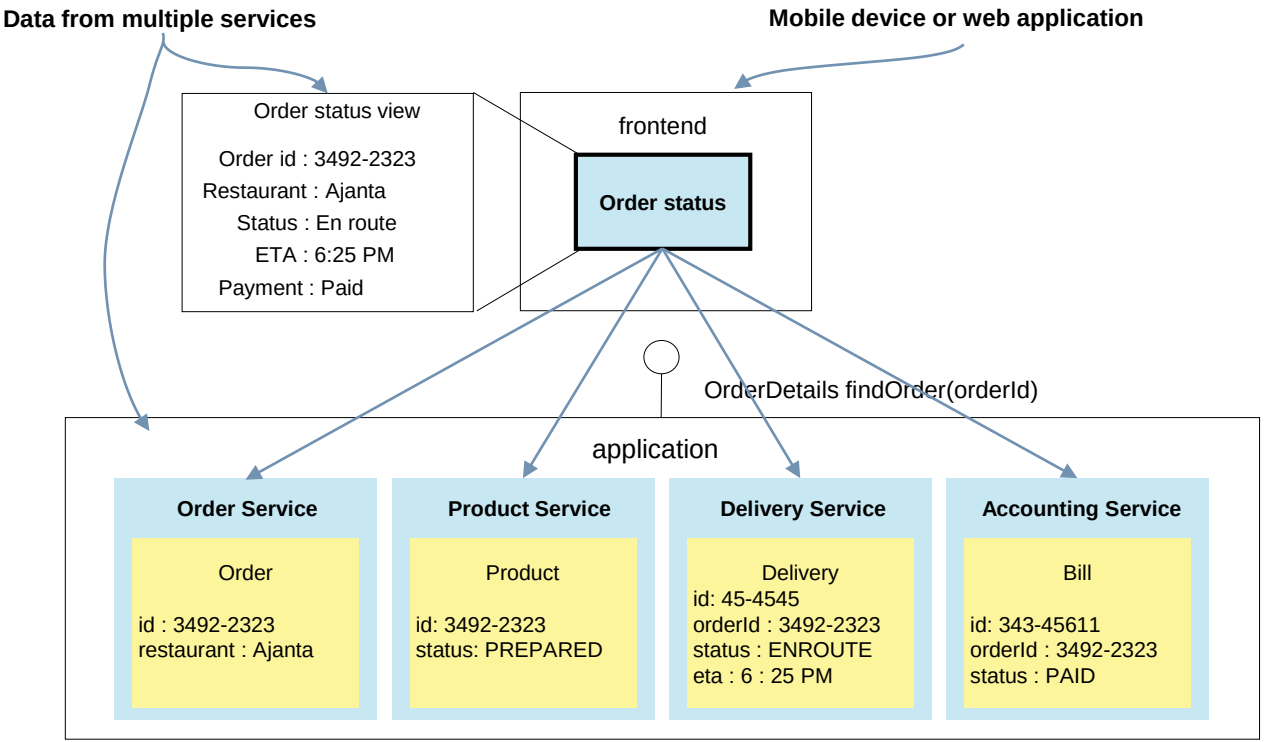
- http://eventuate.io/

- https://vladmihalcea.com/how-to-extract-change-data-events-from-mysql-to-kafka-using-debezium/?fbclid=IwAR33Spb4jPBNI6VNHuCxdu_BxpWdzOLzMvbCtHHvJrRmJPfiEoXwM1qWYBs

Data Query (Projection) in MSA

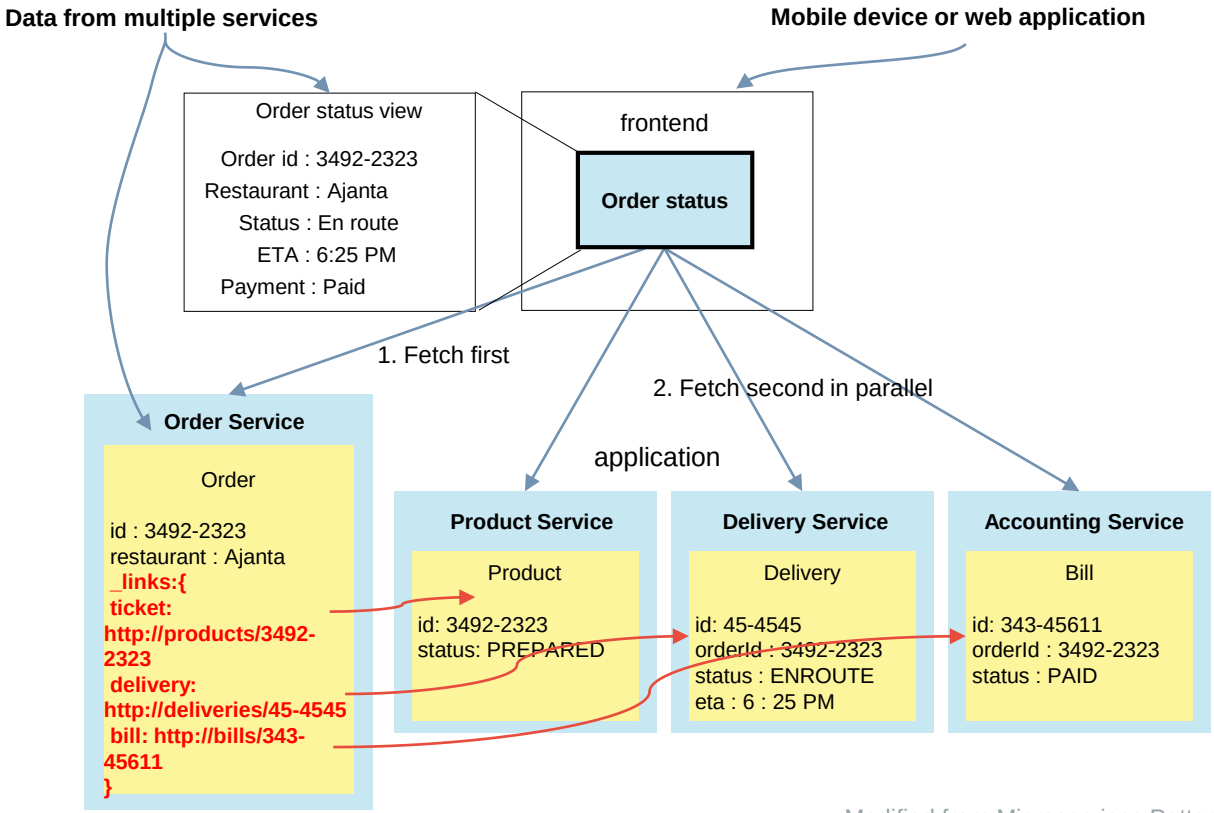
- Issues: Existing Join Queries and Performance Issues
- Composite Services or GraphQL
- Event Sourcing and CQRS

분산서비스에서의 Data Projection Issue



Issues :
1. 성능/속도
2. 데이터량

Data Projection by UI and HATEOAS

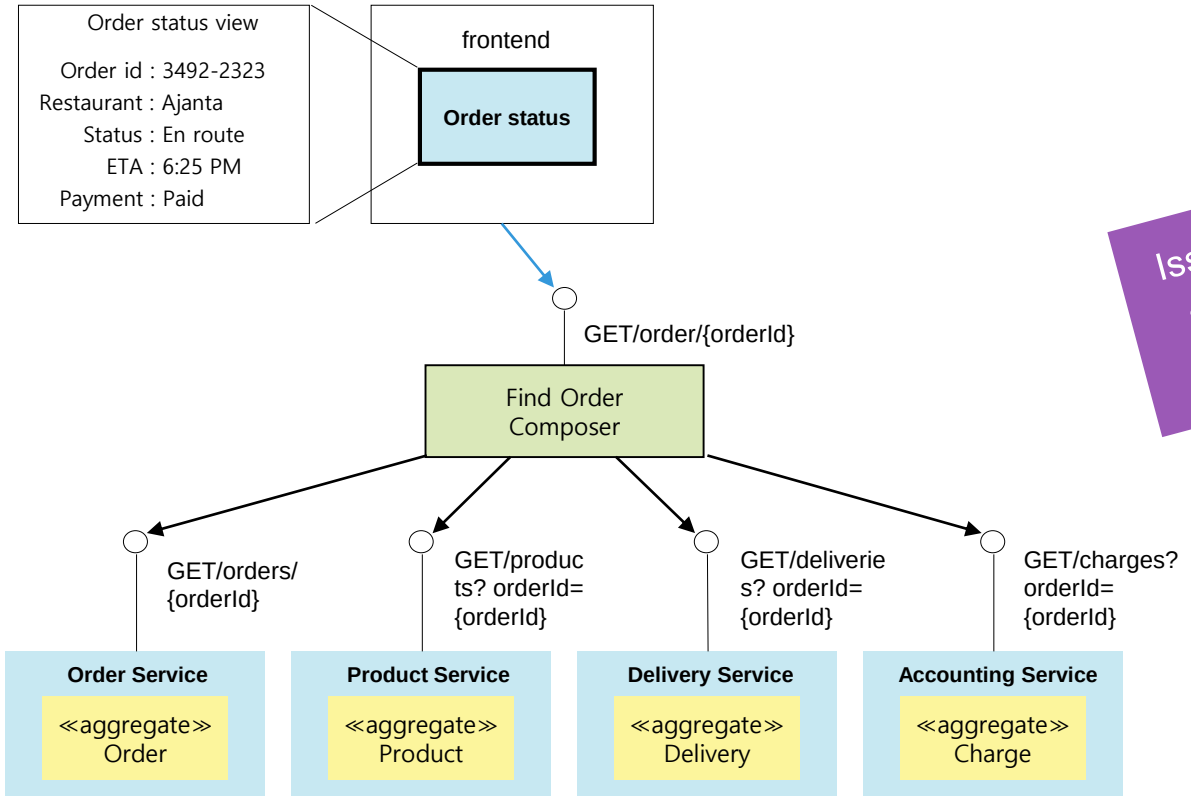


Issues :
- 단점 : UI 개발 복잡성과 성능

Data Projection by UI and HATEOAS

```
me.$http.get(`${API_HOST}/orders/search/findByCustomerId?customerId=${localStorage.getItem('userId')}`)
.then(function (orderResult) {

    orderResult.data._embedded.orders.forEach(function (orderItem, orderIndex) {
        me.$http.get(`${API_HOST}` + orderItem._links.delivery.href)
        .then(function (deliveryResult) {

            orderItem.orderId = deliveryResult.data._embedded.deliveries[0].orderId
            orderItem.deli = deliveryResult.data._embedded.deliveries[0].deliveryState

            me.orderList.push(orderItem)

        })
        .catch(function (deliveryError) {
        })
    })
})
```

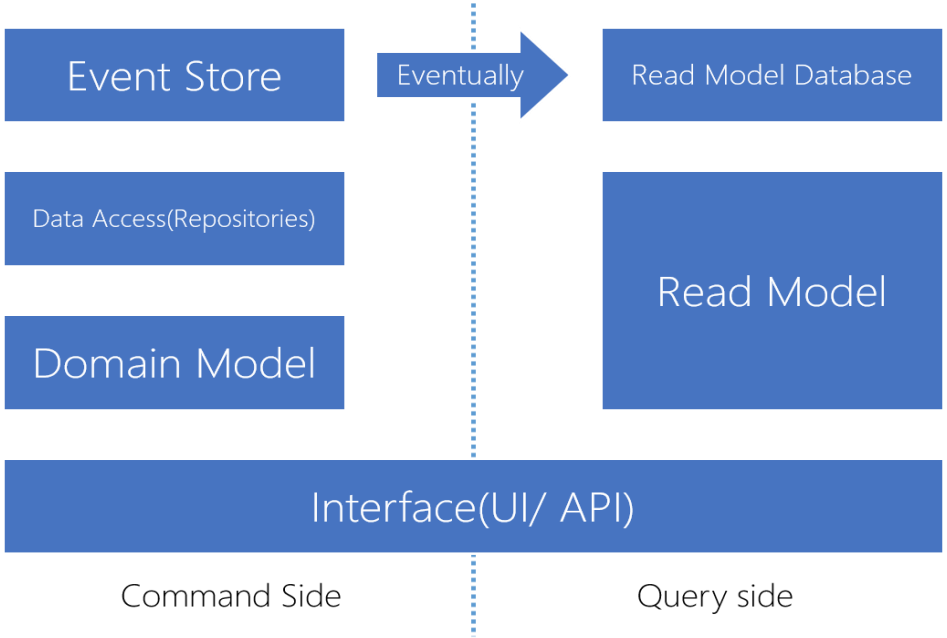
Data Projection by Composite Service or GraphQL



Issues :
- 단점 : 성능(속도), 장애전파

생각을 다르게 하자 : CQRS and Event-Sourcing

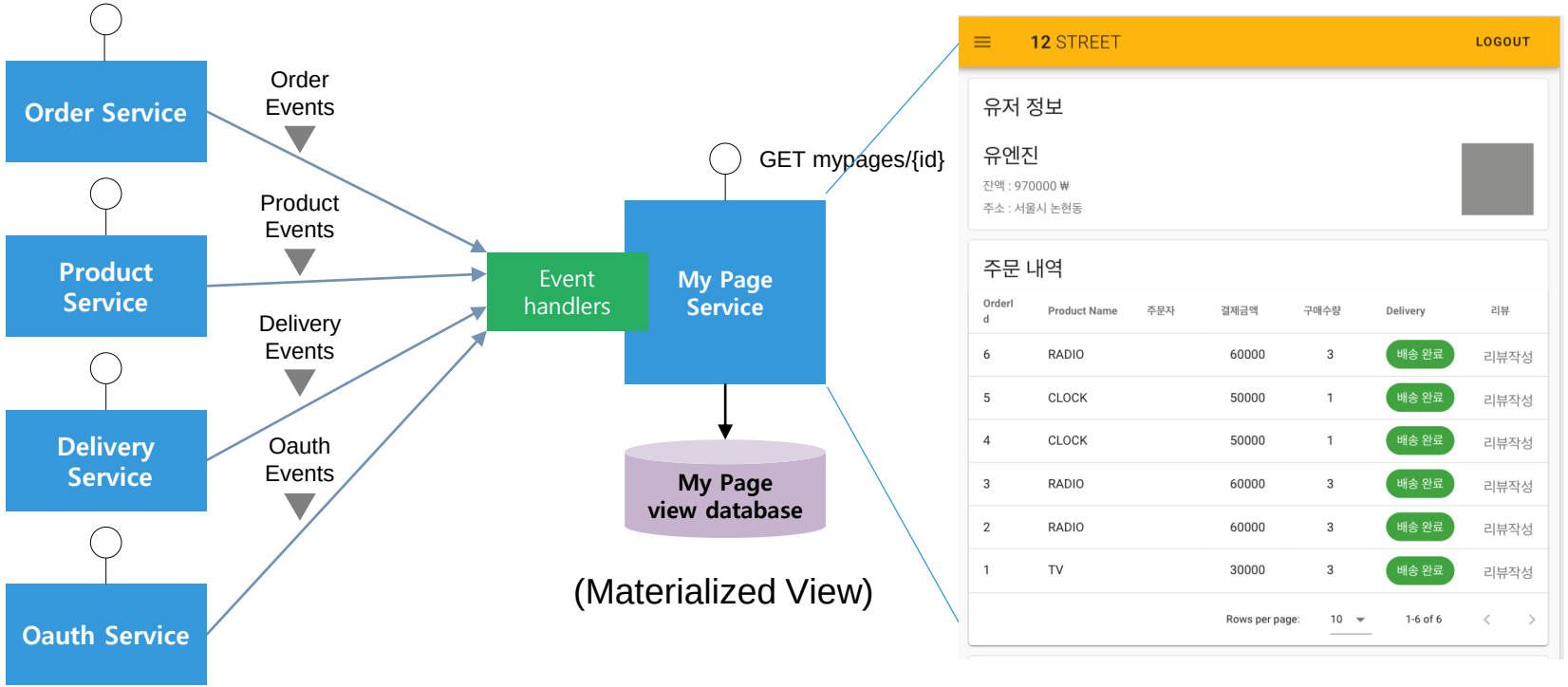
- CQRS 패턴
- 읽기전용 DB와 쓰기 DB를 분리함으로써 Optimistic Locking 구현
- Query 뷰를 다양하게 구성하여 여러 MSA 서비스 목적에 맞추어 각 서비스의 리드모델에 부합
- Polyglot Persistence



https://justhackem.wordpress.com/2016/09/17/what-is-cqrs/

https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson

CQRS 의 확장 : Multiple Event Sources



CQRS 의 확장 : Multiple Event Sources

```
getOrderList() {
    var me = this
    return new Promise(function (resolve, reject) {

                me.$http.get(`${API_HOST}/mypage/order/${localStorage.getItem
                ('userId')}`).then(function (e) {
                            resolve(e.data)
                 });
    });

}
```

Table of content

Microservice and
Event-storming-Based
DevOps Project

DevOps

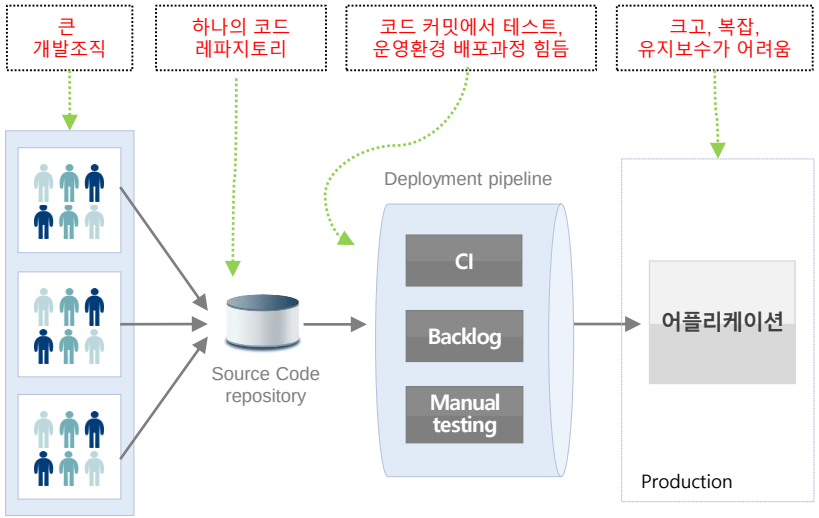
- DevOps Process and Tools
- Deploy Strategies
- Containers and Orchestrators
- Cloud Foundry, Bluemix, and Kubernetes

https://www.youtube.com/watch?v=_I94-tJlovg

Process Change : 열차말고 택시를 타라!

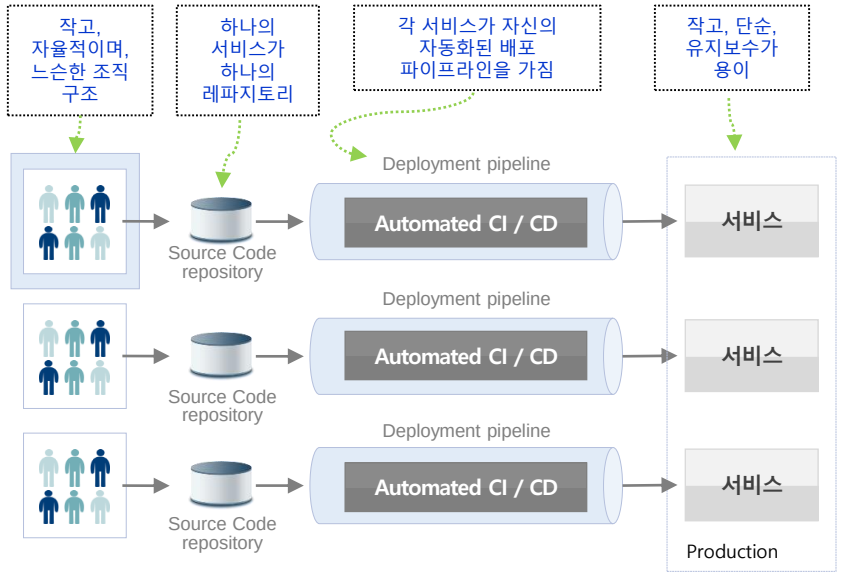
모노리식 개발 및 운영환경

- 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 파이프라인을 적용하기가 어렵고 Delivery 시간이 과다 소요



마이크로서비스 개발 및 운영환경

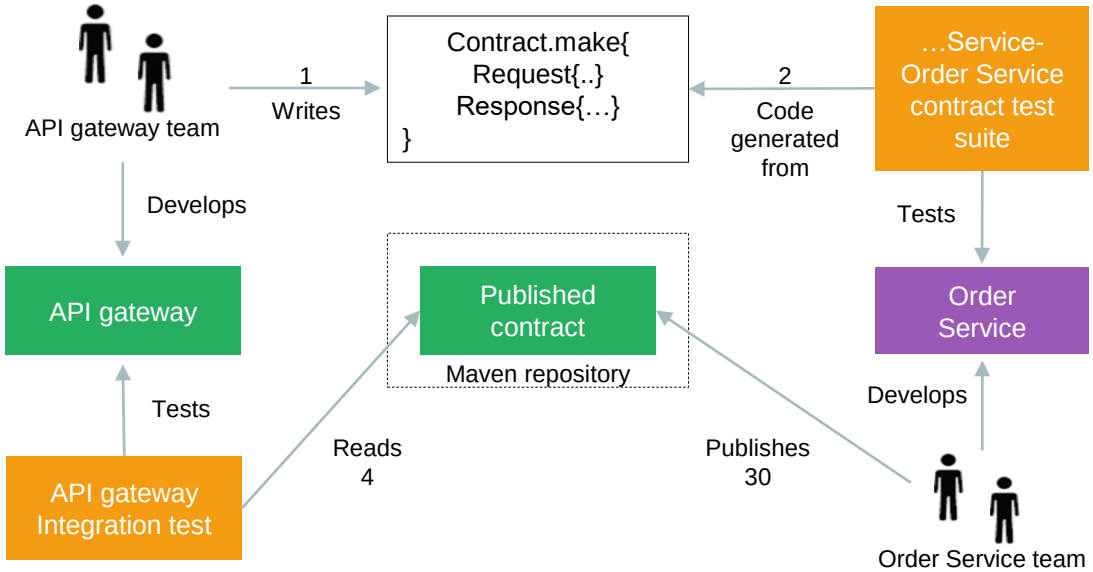
- 작고 분화된 조직에서 서비스를 작은 크기로 나누어 개발하므로 해당 비즈니스 로직에만 집중하게 되어 개발 생산성 향상
- 연관된 마이크로서비스만 테스트를 수행하므로 개발/테스트/배포 일정이 대폭 축소



Process Change : Consumer-driven Test

MSA 서비스의 테스트

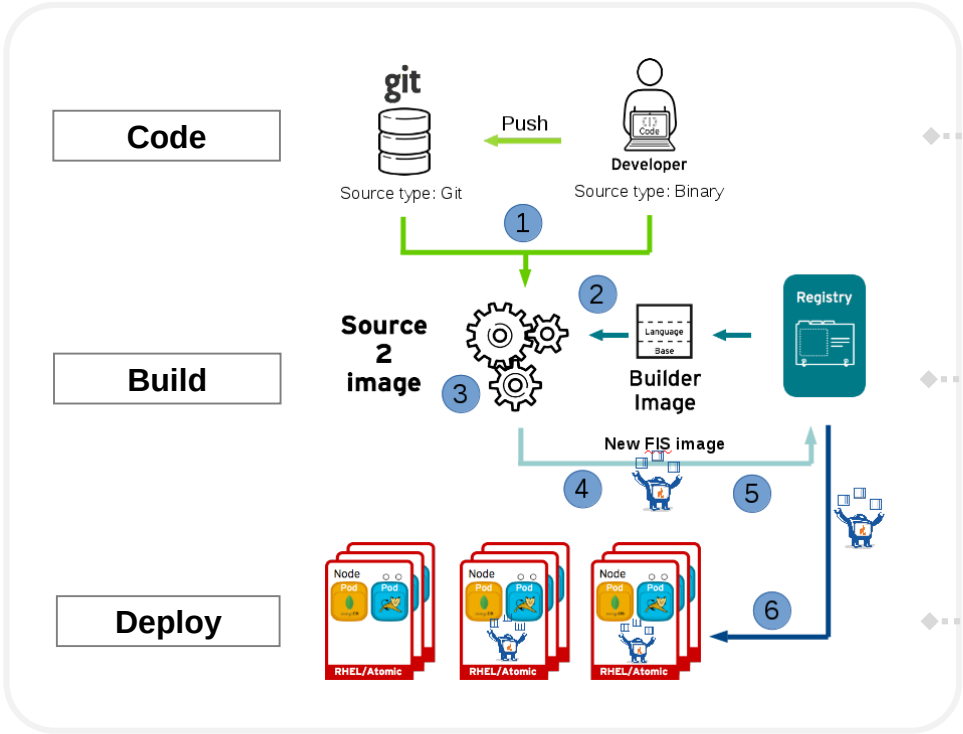
컨트랙트 테스트는 서비스 수요자가 주도하여 테스트를 작성한다.
수요자가 테스트를 작성하여 제공자의 레포지토리에 Pull-Request 하면, 제공자가 이를 Accept 한후에 제공자측에서 테스트가 생성되어 테스트가 벌어진다.



특징

- MSA 테스트에는 Contract-based Test 가 특징적
- API Consumer가 먼저 테스트 작성
- API Provider가 Pull Request 수신 후 테스트는 자동으로 생성됨
- Consumer 측에 Mock 객체가 자동생성 병렬 개발이 가능해짐
- Spring Cloud Contract가 이를 제공
- Provider의 일방적인 버전업에 따른 하위 호환성 위배의 원천적 방지

DevOps toolchain



GitLab

Travis CI

docker

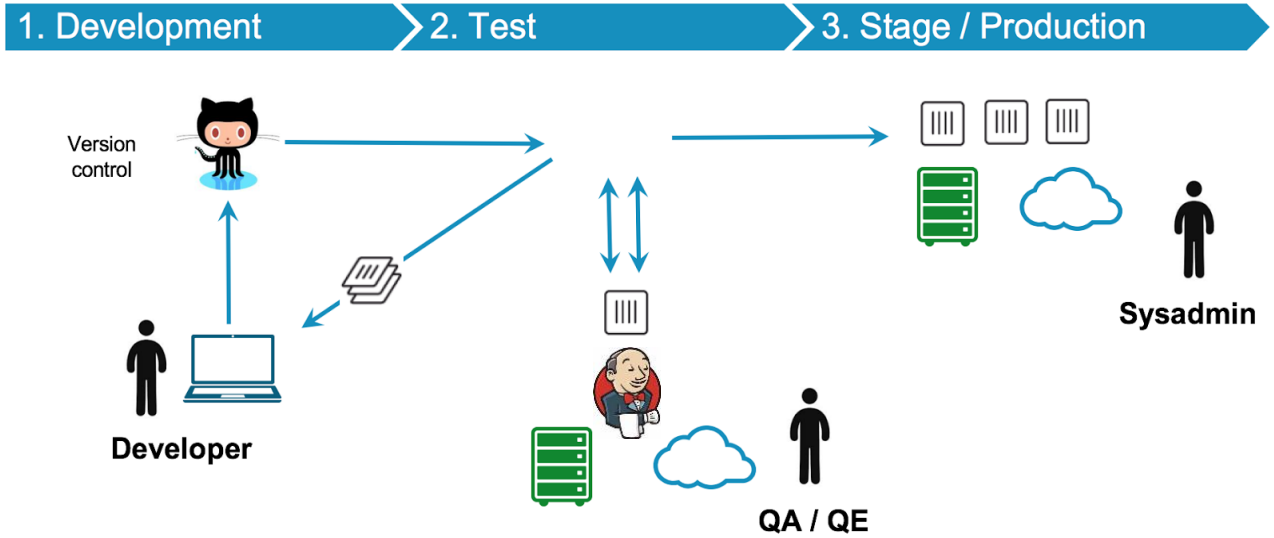
Maven

kubernetes

DC/OS

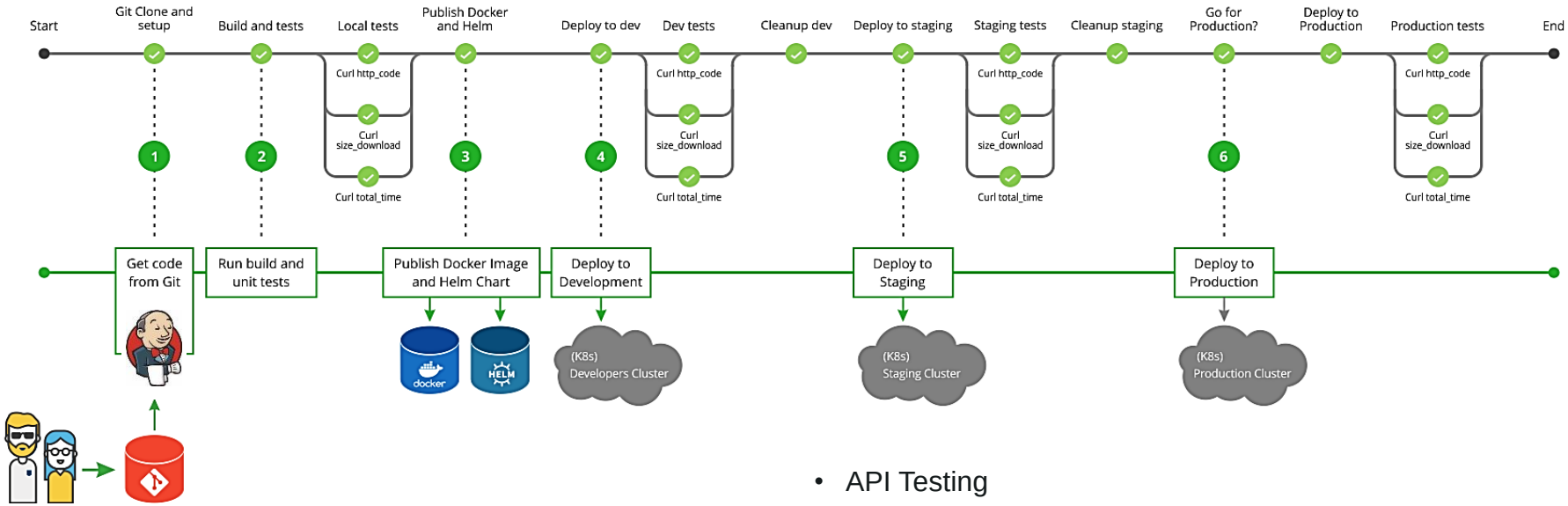
CI/CD Tools – conventional CI/CD

CI /CD Workflow



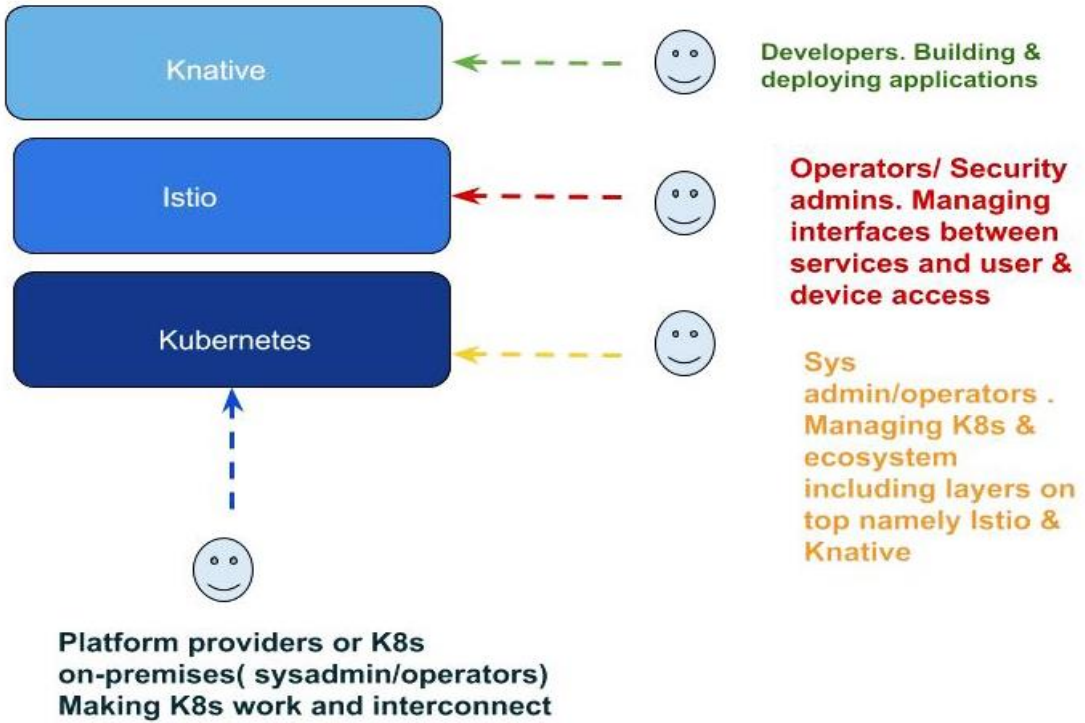
- 잦은 배포
- 일반 서버에 배포
- 테스트 자동화/커버리지
- 도구: Jenkins, Travis, Gitlab 등

CI/CD Tools – Container-based



- API Testing
- Blue/Green, Canary
- 도구 : Jenkins + Docker + Spinnaker + Helm + Kubernetes
  → 매우복잡

CI/CD Tools - Serverless



- Infrastructure As A Code
- Application Configuration 과 Infra Configuration을 하나의 설정에 통합
- 단일 도구

배포 전략별 비교

When it comes to production, a ramped or blue/green deployment is usually a good fit, but proper testing of the new platform is necessary.

Blue/green and shadow strategies have more impact on the budget as it requires double resource capacity. If the application lacks in tests or if there is little confidence about the impact/stability of the software, then a canary, a/b testing or shadow release can be used.

If your business requires testing of a new feature amongst a specific pool of users that can be filtered depending on some parameters like geolocation, language, operating system or browser features, then you may want to use the a/b testing technique.

Container Solutions

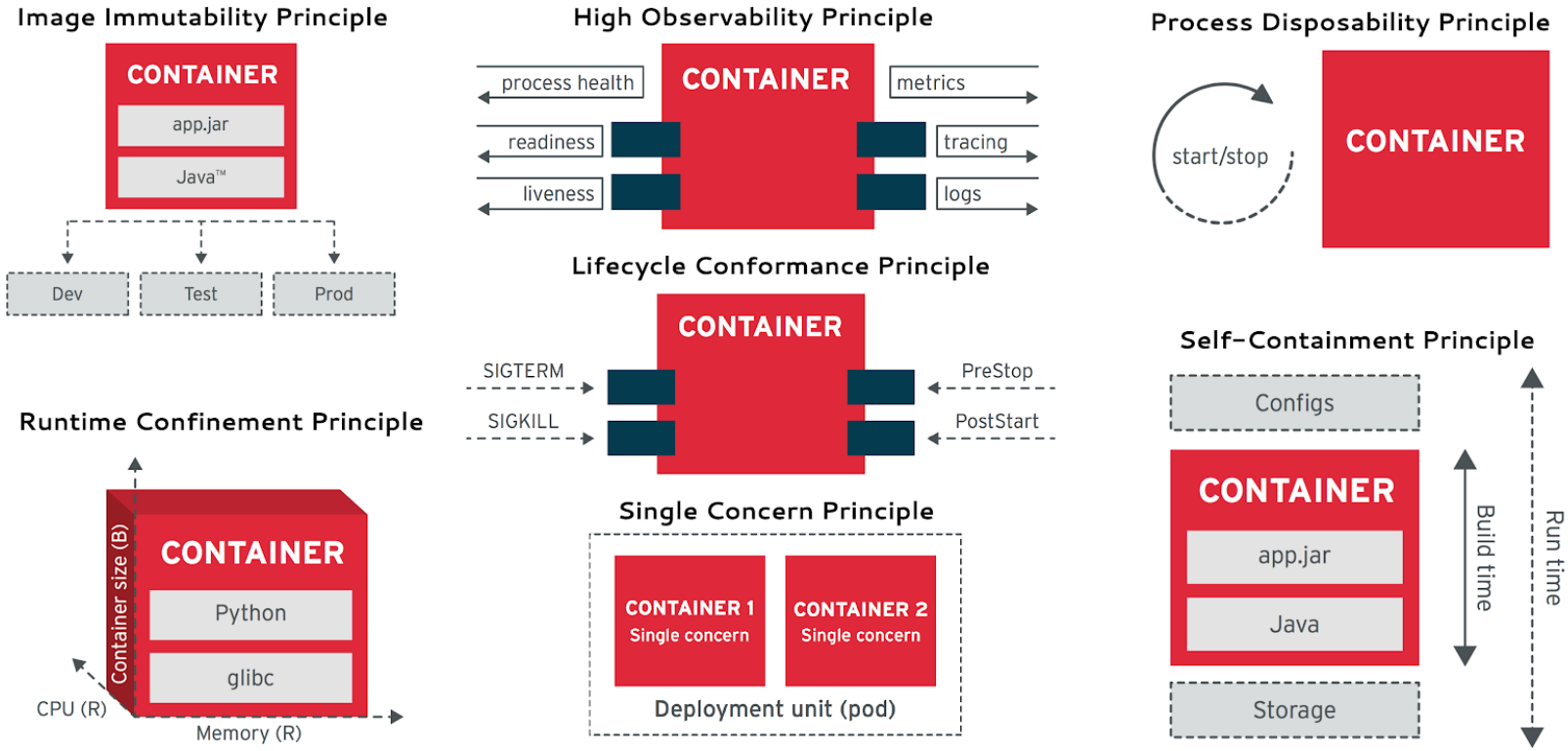
Strategy	ZERO DOWNTIME	REAL TRAFFIC TESTING	TARGETED USERS	CLOUD COST	ROLLBACK DURATION	NEGATIVE IMPACT ON USER	COMPLEXITY OF SETUP
RECREATE version A is terminated then version B is rolled out	✗	✗	✗	■□□	■■■	■■■	□□□
RAMPED version B is slowly rolled out and replacing version A	✓	✗	✗	■□□	■■■	■□□	■□□
BLUE/GREEN version B is released alongside version A, then the traffic is switched to version B	✓	✗	✗	■■■	□□□	■■□	■■□
CANARY version B is released to a subset of users, then proceed to a full rollout	✓	✓	✗	■□□	■□□	■□□	■■□
A/B TESTING version B is released to a subset of users under specific condition	✓	✓	✓	■□□	■□□	■□□	■■■
SHADOW version B receives real world traffic alongside version A and doesn't impact the response	✓	✓	✗	■■■	□□□	□□□	■■■

DevOps Platforms

Container	Workload Distribution Engine (Container Orchestrator)	PaaS
• Docker	• Kubernetes	• Google Cloud Platform
	• Docker SWARM(toy)	• Redhat Open Shift
	• Mesos Marathon(DCOS)	• Amazon EKS
	• Cloud Foundry	• IBM Bluemix
• Warden(Garden)		• Heroku
		• GE's Predix
		• Pivotal Web Services
• Hypervisor	• CF version 1	• Amazon Beanstalk
	• Engine yard….	

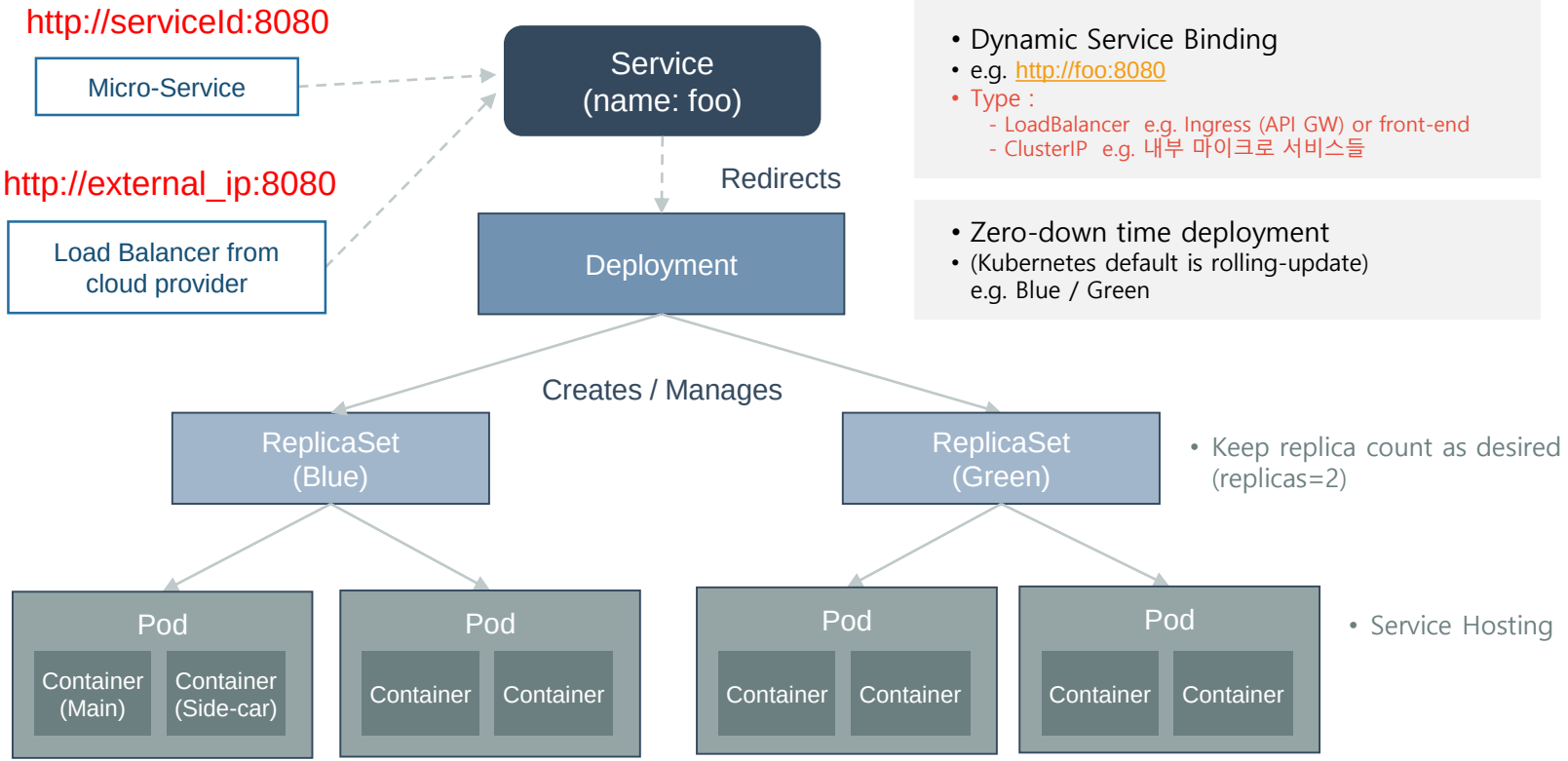
Kubernetes

Container-based Application Design



https://kubernetes.io/blog/2018/03/principles-of-container-app-design/?fbclid=IwAR2oMrdP0d1Q6LXebtxNPnt-RS5DIlkCIwpaMSL5mmW7VMaQb6hRV8hkd38

Kubernetes Object Model



Responsible for :

- Dynamic Service Binding
- e.g. http://foo:8080
- Type :
  - LoadBalancer e.g. Ingress (API GW) or front-end
  - ClusterIP e.g. 내부 마이크로 서비스들

- Zero-down time deployment
- (Kubernetes default is rolling-update)
  e.g. Blue / Green

- Keep replica count as desired (replicas=2)

- Service Hosting

Declaration based configuration

> Desired state

> my-app.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 2
…
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
---
apiVersion: apps/v1
kind: Service
metadata:
  name: nginx-service
  labels:
    app: nginx
```

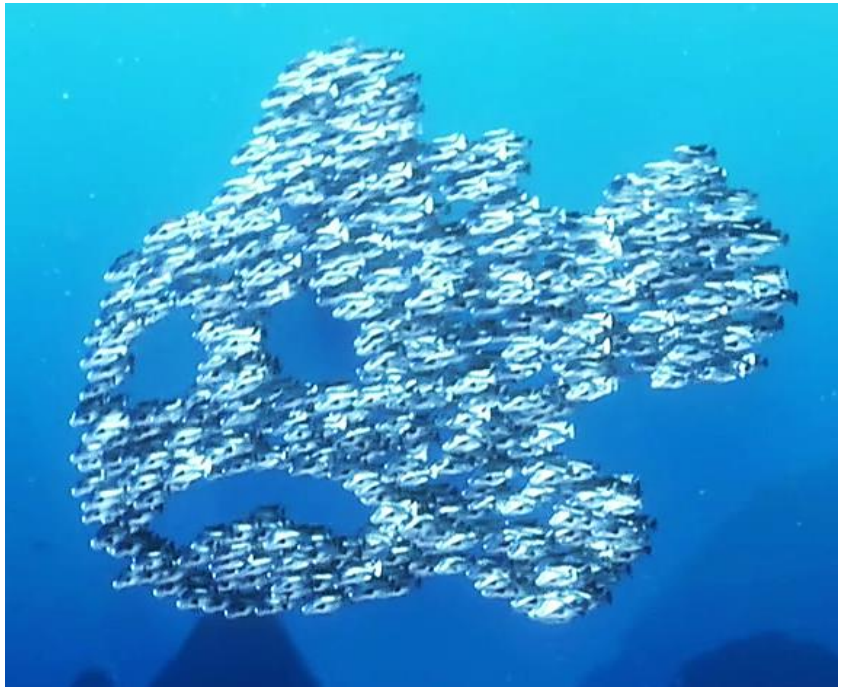
```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
  labels:
    app: backend
spec:
  replicas: 3
…
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: backend:latest
        ports:
        - containerPort: 8080
---
apiVersion: apps/v1
kind: Service
metadata:
  name: backend-service
  labels:
    app: backend
```

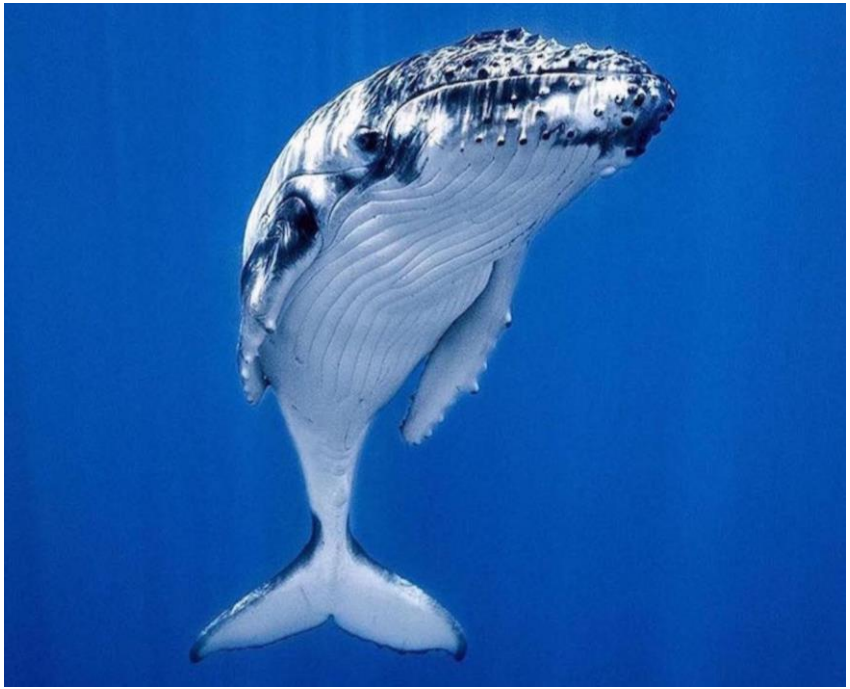
```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: db-deployment
  labels:
    app: db
spec:
  replicas: 2
…
  template:
    metadata:
      labels:
        app: db
    spec:
      containers:
      - name: db
        image: mongo
        ports:
        - containerPort: 27017
---
apiVersion: apps/v1
kind: Service
metadata:
  name: mongo-service
  labels:
    app: mongo
```

Appendix :
Outer-Architect Course

- What is outer-architecture and its components
- Example Configurations

“서비스가 죽지 않으면 어떨까?”





항상성이 (Self-Healing) 자동으로 유지되면 어떨까?

자원을 최대한 공유할 수 있을까?

Continuous Delivery &
Zero Downtime Deployment 가 되면 어떨까?

Facebook

2. **배포 주기**
   - 매일 마이너 업데이트
   - 메이저 업데이트 (매주 화요일 오후)

3. **DeploymentPipeline**

출처: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6449236

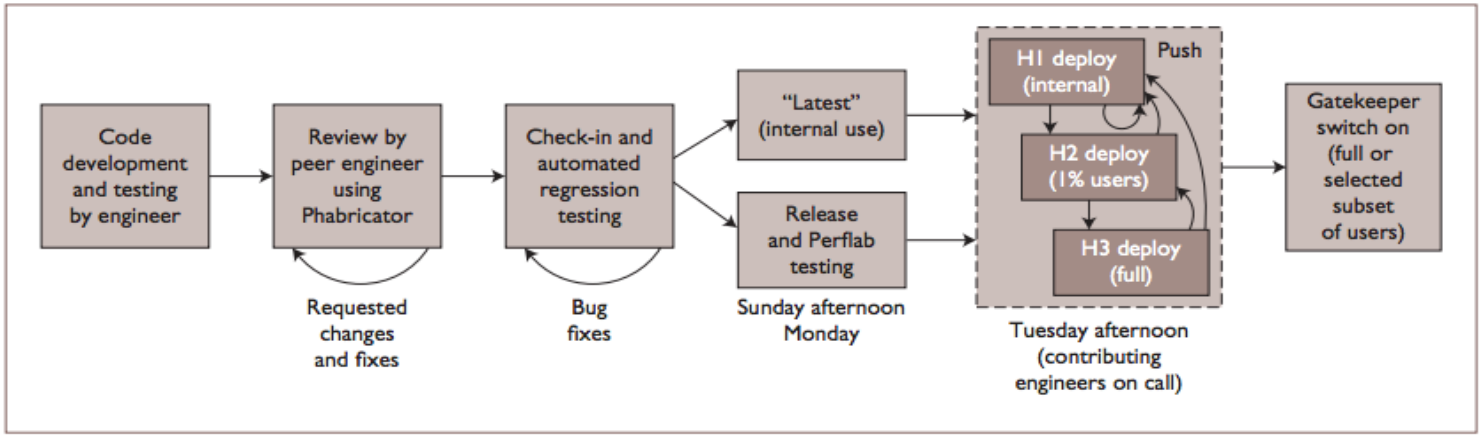


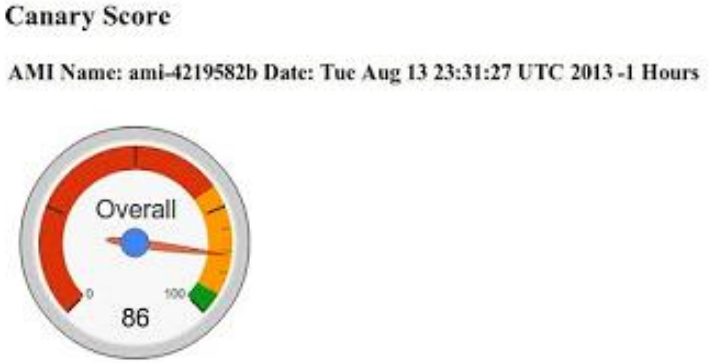
Figure 8. The Facebook deployment pipeline. Multiple controls exist over new code.

Netflix

4. Canary를 통해서 확신 갖기

- Canary란? 실제 production 환경에서 small subset에서 새로운 코드를 돌려보고 옛날 코드와 비교해서 새로운 코드가 어떻게 돌아가는 지 보는 것

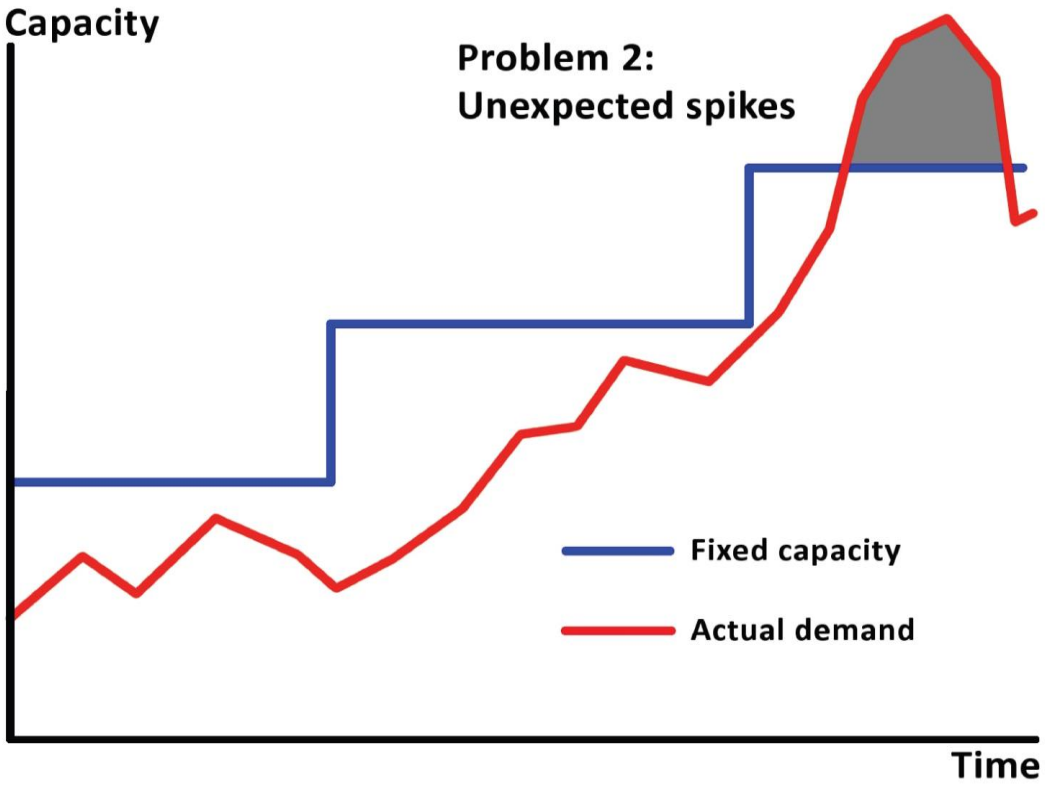
- Canary 분석 작업(HTTP status code, response time, 실행수, load avg 등이 옛날 코드랑 새로운 코드랑 비교해서 어떻게 다른 지 확인하는 것)은 1000개 이상의 metric을 비교해서 100점 만점에 점수를 주고 일정 점수일 경우만 배포할 수 있음.



출처: http://techblog.netflix.com/2013/08/deploying-netflix-api.html

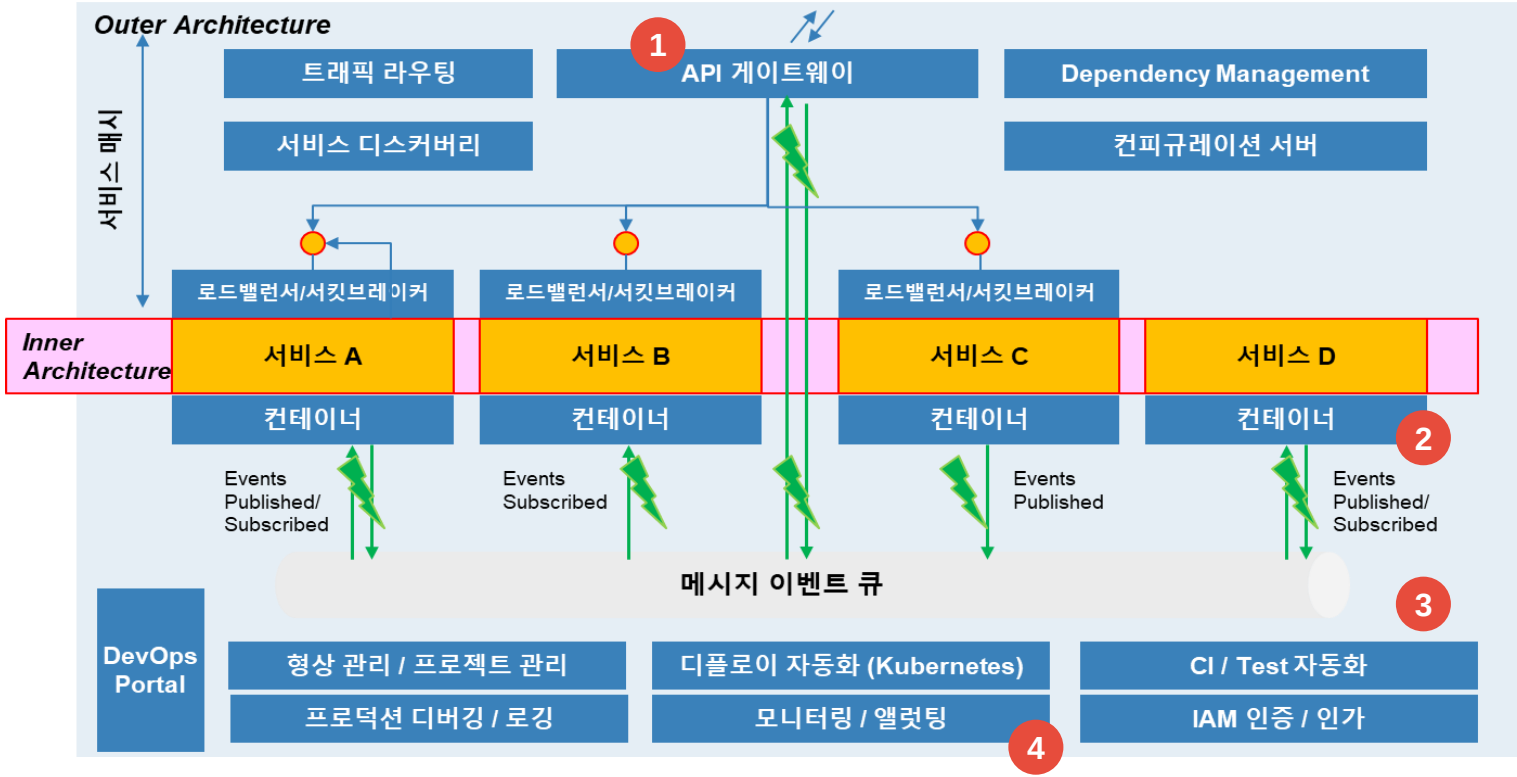
DevOps : Issues

Managing Scalability



출처 : *2010 architecting for the cloud (*http://www.slideshare.net/simone.brunozzi/2010-architecting-for-the-cloud-4719195*)*

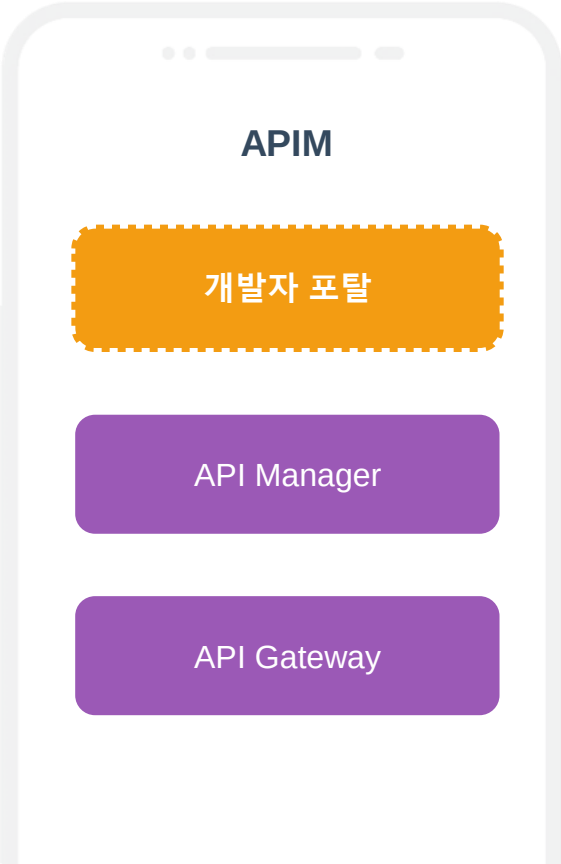
What is Outer-Architecture



1. API Gateway : API를 사용하여 통신하는 모든 서비스들의 인증 및 제어, 트래픽 모니터링 등 수행
2. Managed Container System : 서비스들의 중앙 관리 및 인증, 라우팅등의 기능 수행
3. Backing Services : 스토리지(Block, Object), Cache, Message Queue 등의 기능 영역
4. Observability Services : 서비스들의 이벤트 모니터링 및 알림, 로그 취합, 진단 등 수행

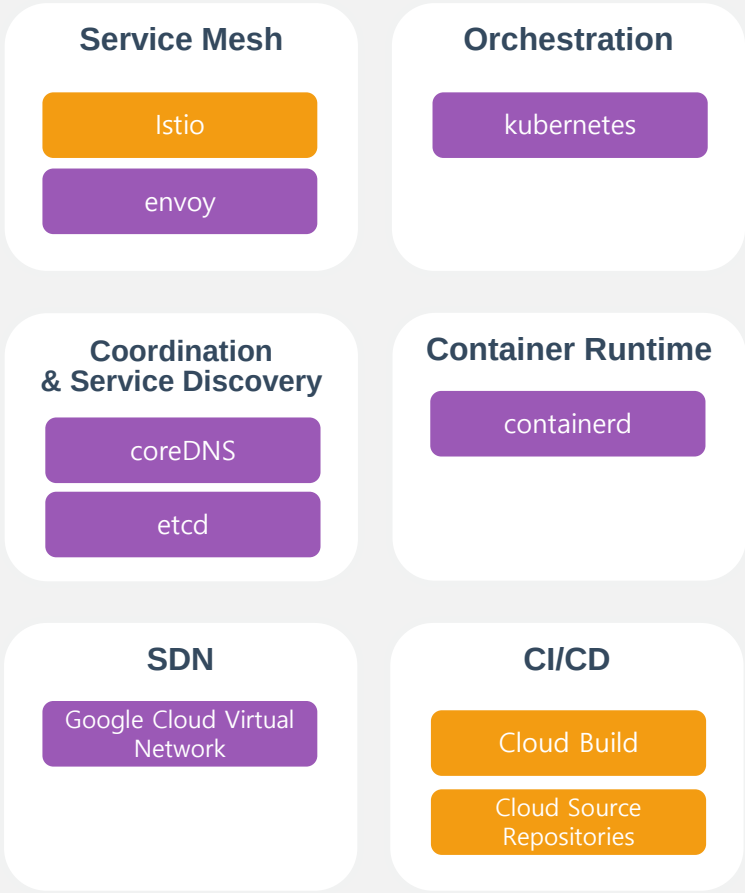
APIM

- **개발자 포탈** : 개발자가 API등록하고 조회할 수 있는 포탈. 개발자포탈에 등록된 API를 기준으로 OAuth서버를 통해 인가 처리가 이루어진다.
- **API Manager** : 개발자포털에 등록된 API를 기준으로 실제 API를 등록하고 관리하는 시스템
- **API Gateway** : 모든 API서비스를 해당 gateway를 진입점으로 통신하게 되어 인증, 통제, 로깅 등의 통제 및 처리를 가능케 해준다.



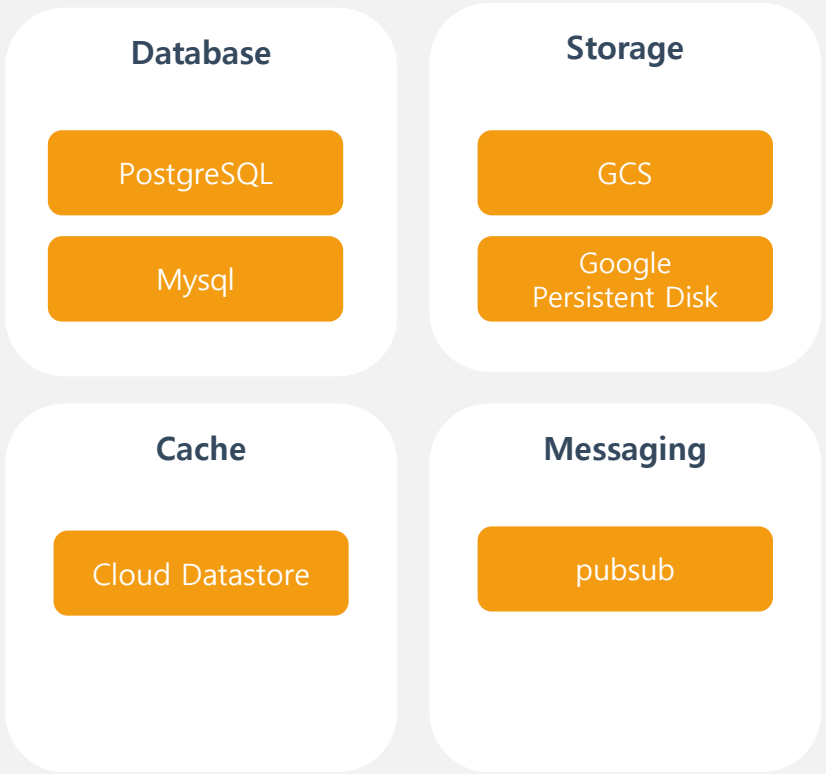
PaaS

- Service Mesh : 서비스 레지스트리, 중앙 로그 수집, 분산 트렌젝션 추적, 메세지 라우팅등 다양한 기능들을 중앙에서 통제하는 컨셉. 대표적인 솔루션으로는 Istio, Linkerd 등이 있다.
- Orchestration : 컨테이너 오케스트레이션은 컨테이너와 서비스들이 대규모로 운영 될 때, 컨테이너의 라이프사이클을 관리하는 것. 대표적으로 kubernetes가 있다.
- Service Discovery : 클라이언트가 서비스를 호출할때 서비스의 위치 (IP주소와 포트)를 알아낼 수 있는 기능. 대표적으로 etcd가 있다.
- Coordination : 분산 환경에서 정보를 공유하고 서버달의 상태를 체크하고 동기화 하는 기능. 대표적으로 zookeeper, consul등이 있다.
- Container Runtime : 컨테이너를 실행시키는 역할. 대표적으로 docker, cri-o, rkt 등이 있다.
- SDN : software defined network. network를 software로 가상화하여 중앙 관리하는 기술. 대표적으로 ovs, flannel calico 등이 있다.
- CI/CD : 지속적인 통합, 지속적인 서비스 제공, 지속적인 배포. 대표적으로 Jenkins, GitLab 등이 있다.

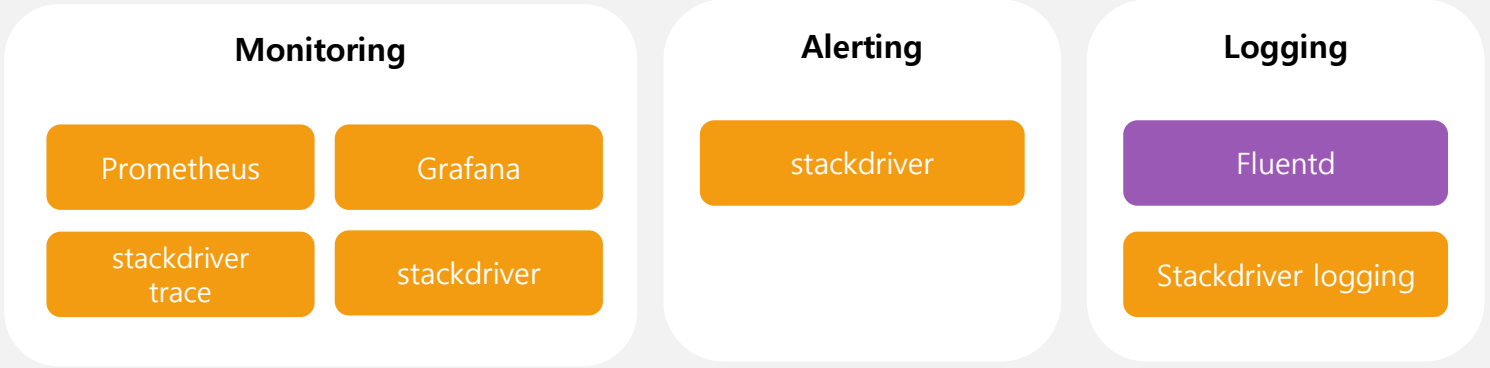


Backing Services

- DBMS : 데이터를 일정한 룰에 의해 관리하는 기술. 일반적으로 관계형 데이터베이스(RDB)를 이용. 대표적으로 Oracle, Mysql 등이 있다.
- Starage : File Starage, Block Storage, Object Storage 등 데이터를 저장하기 위한 기술.
- Cache : 데이터를 메모리에 저장하여 활용하는 기술. 대표적으로 Redis, HBase, CouchBase 등이 있다.
- Messaging : 비동기 메시지를 처리하기 위해 사용하는 기술. 대표적으로 kafka, Rabbit MQ 등이 있다.

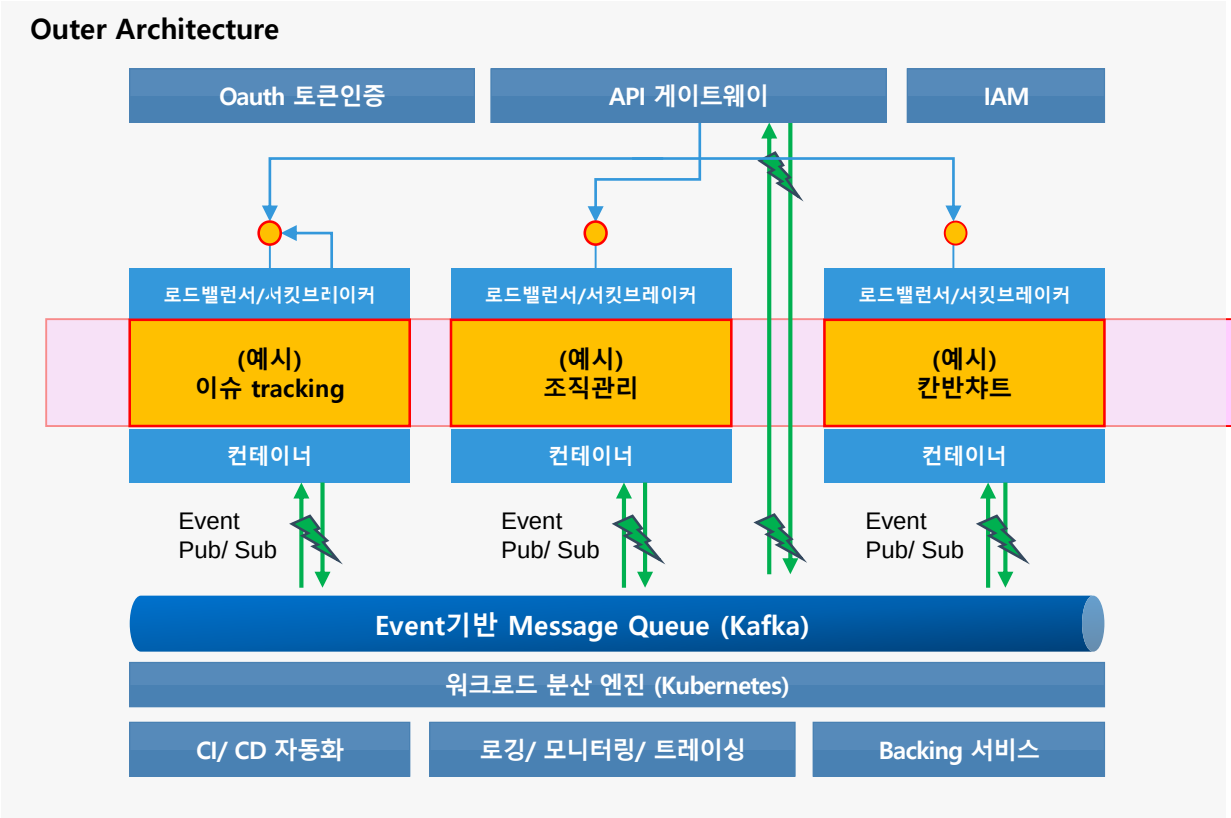


Observability

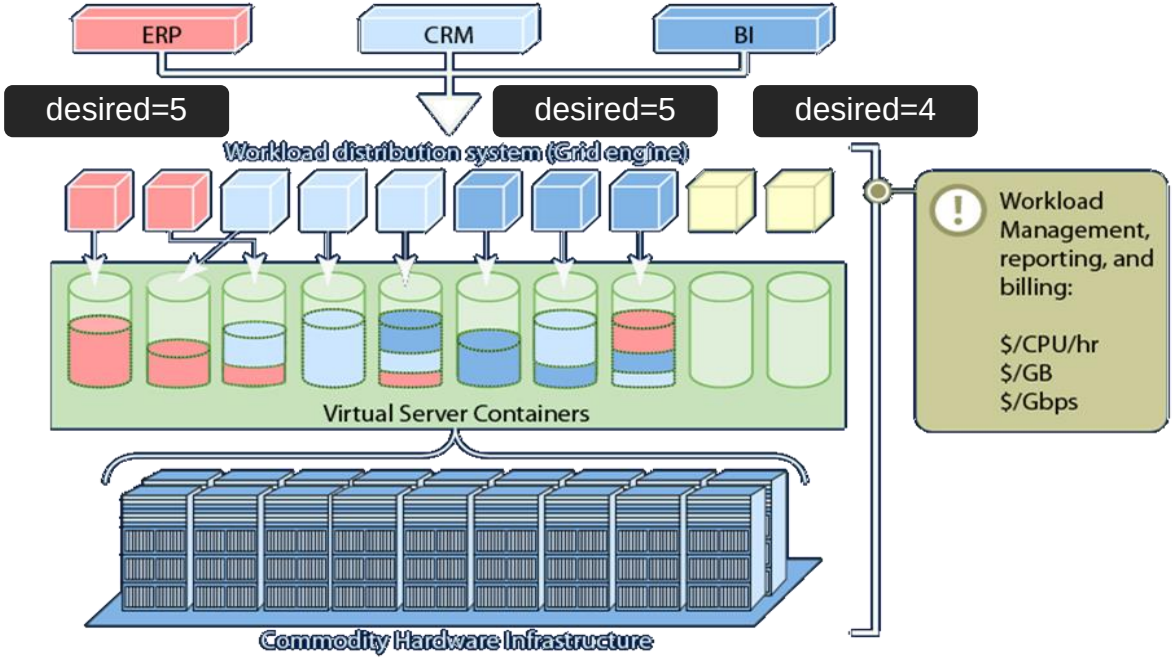


- Monitoring : 시스템 자원의 사용량이나 에러등에 대해 모니터링 하는 기술.
- Alerting : 로깅 및 모니터링의 정보를 활용하여 특정 규칙을 걸어 위반할 경우 알림을 보내는 기술.
- Logging : 시스템에서 발생하는 로그를 수집 및 분석을 하는 기술.

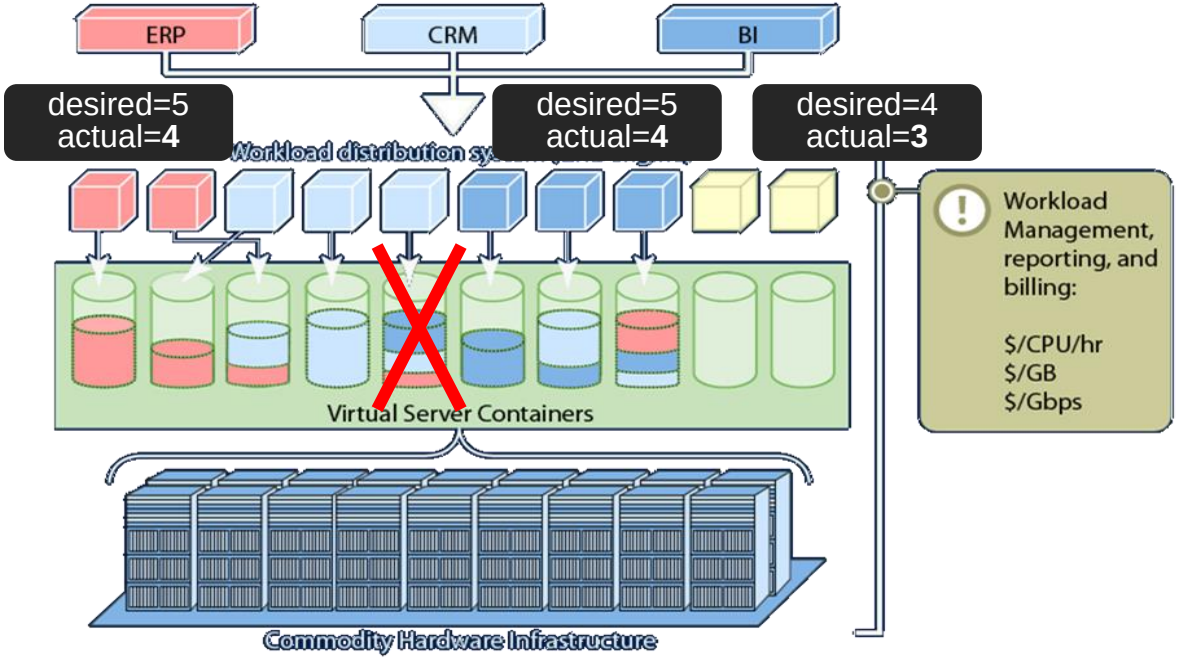
Applied Configuration



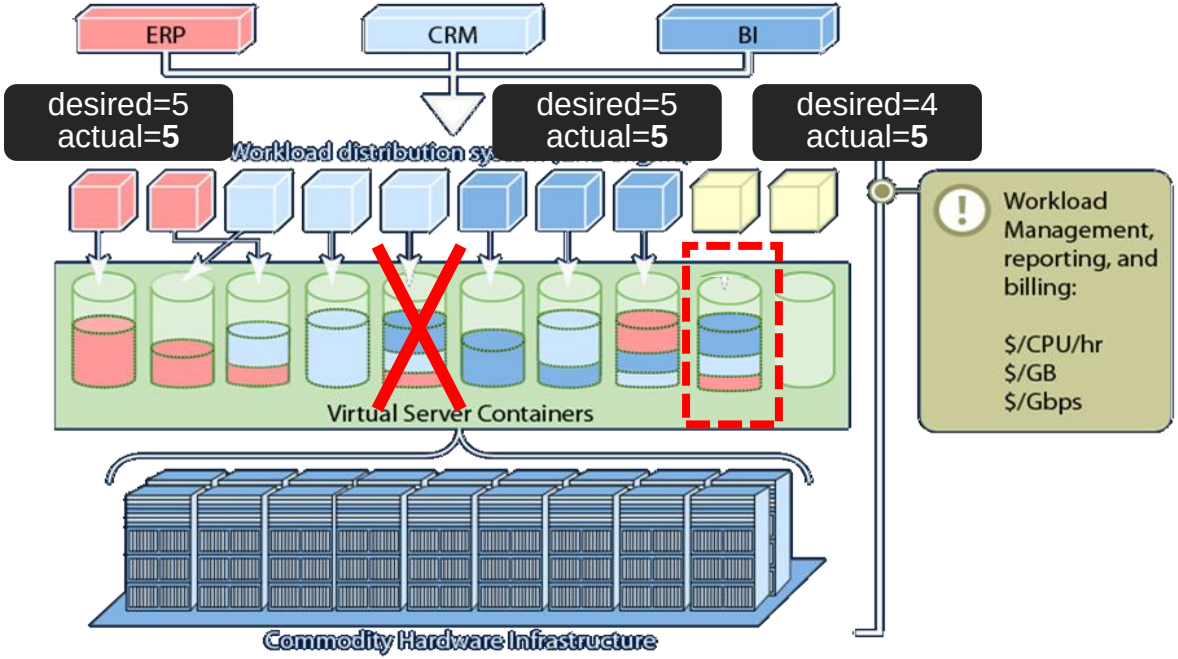
Workload Distribution Engine



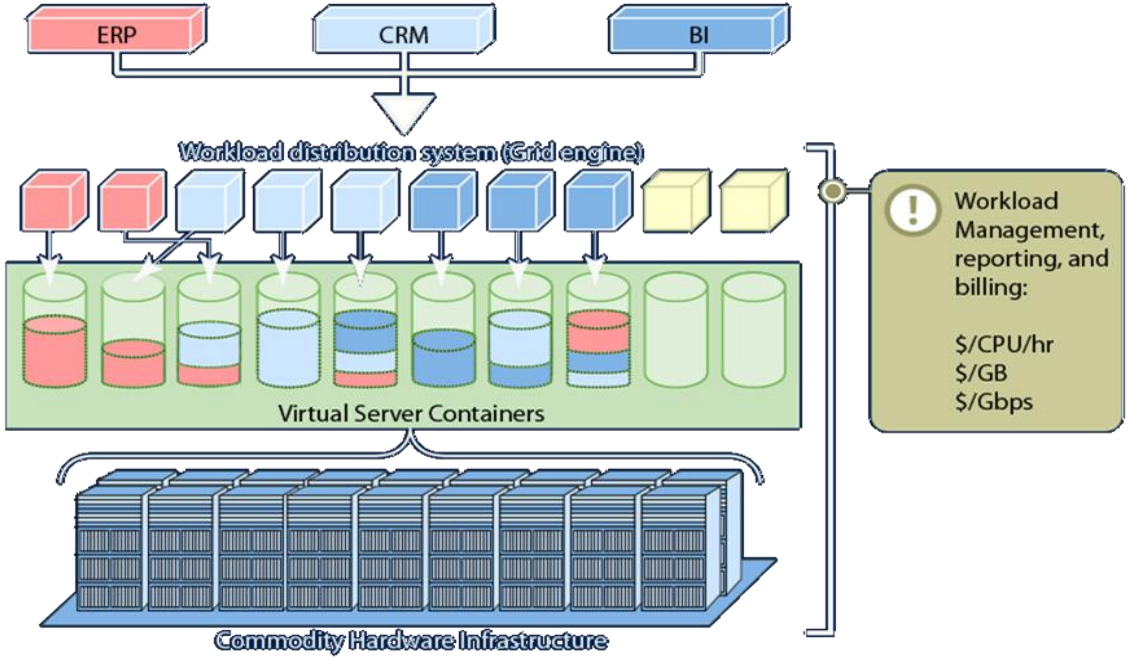
Workload Distribution Engine



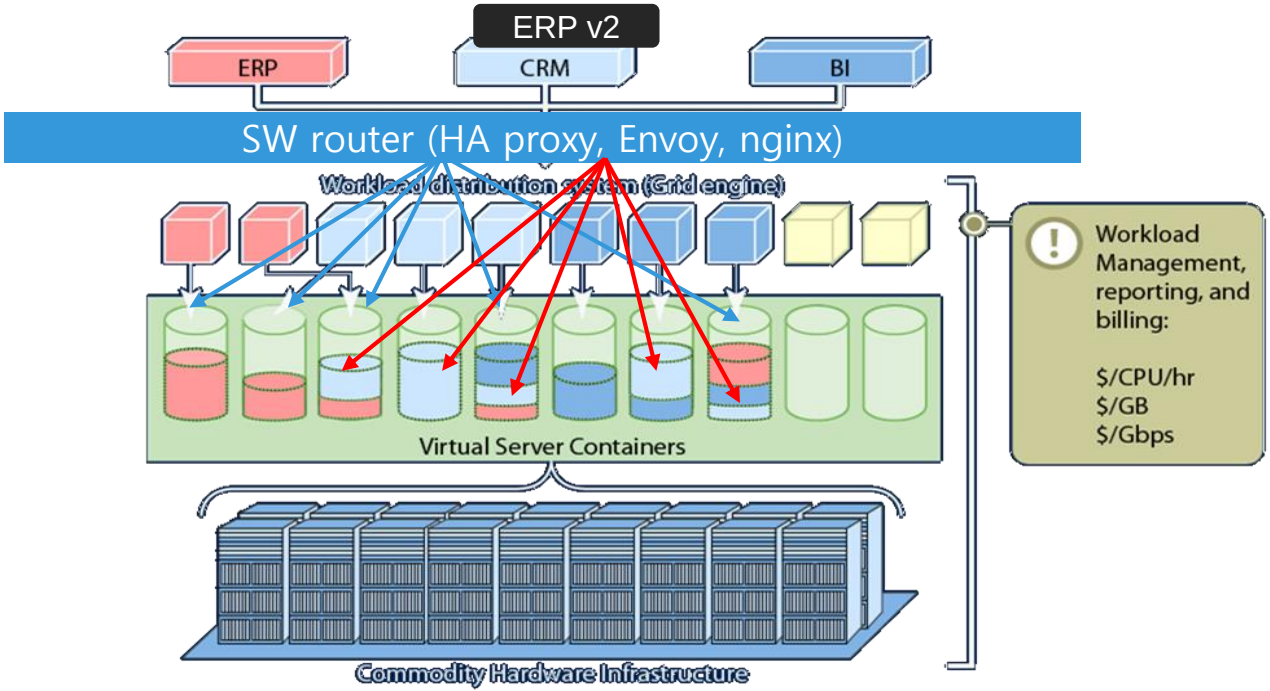
Workload Distribution Engine



Workload Distribution Engine



Workload Distribution Engine

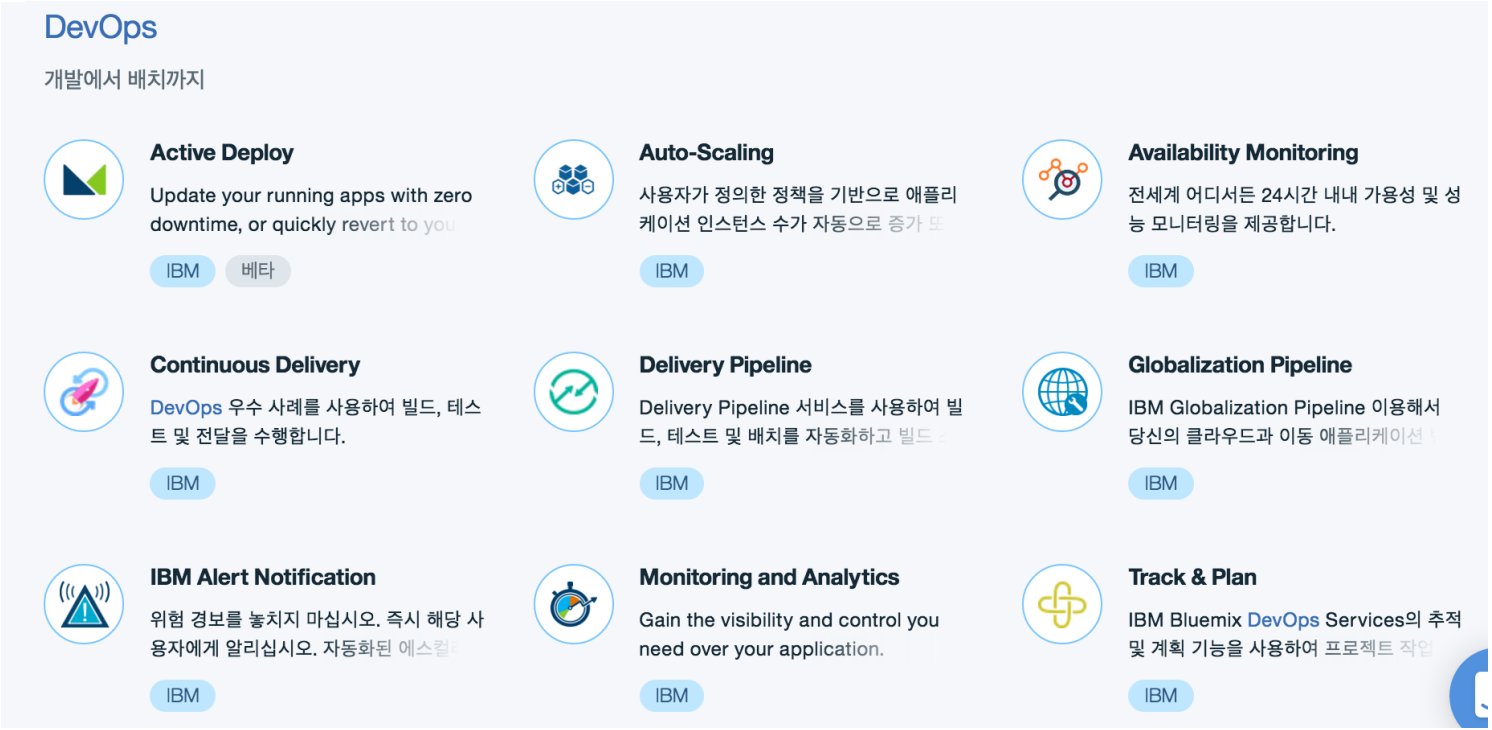


Platforms around the outer Architecture

Container	Workload Distribution Engine (Container Orchestrator)	PaaS
• Docker	• Kubernetes	• Google Cloud Platform
	• Docker SWARM(toy)	• Redhat Open Shift
	• Mesos Marathon(DCOS)	• Amazon EKS
	• Cloud Foundry	• IBM Bluemix
• Warden(Garden)		• Heroku
		• GE's Predix
		• Pivotal Web Services
• Hypervisor	• CF version 1	• Amazon Beanstalk
	• Engine yard....	

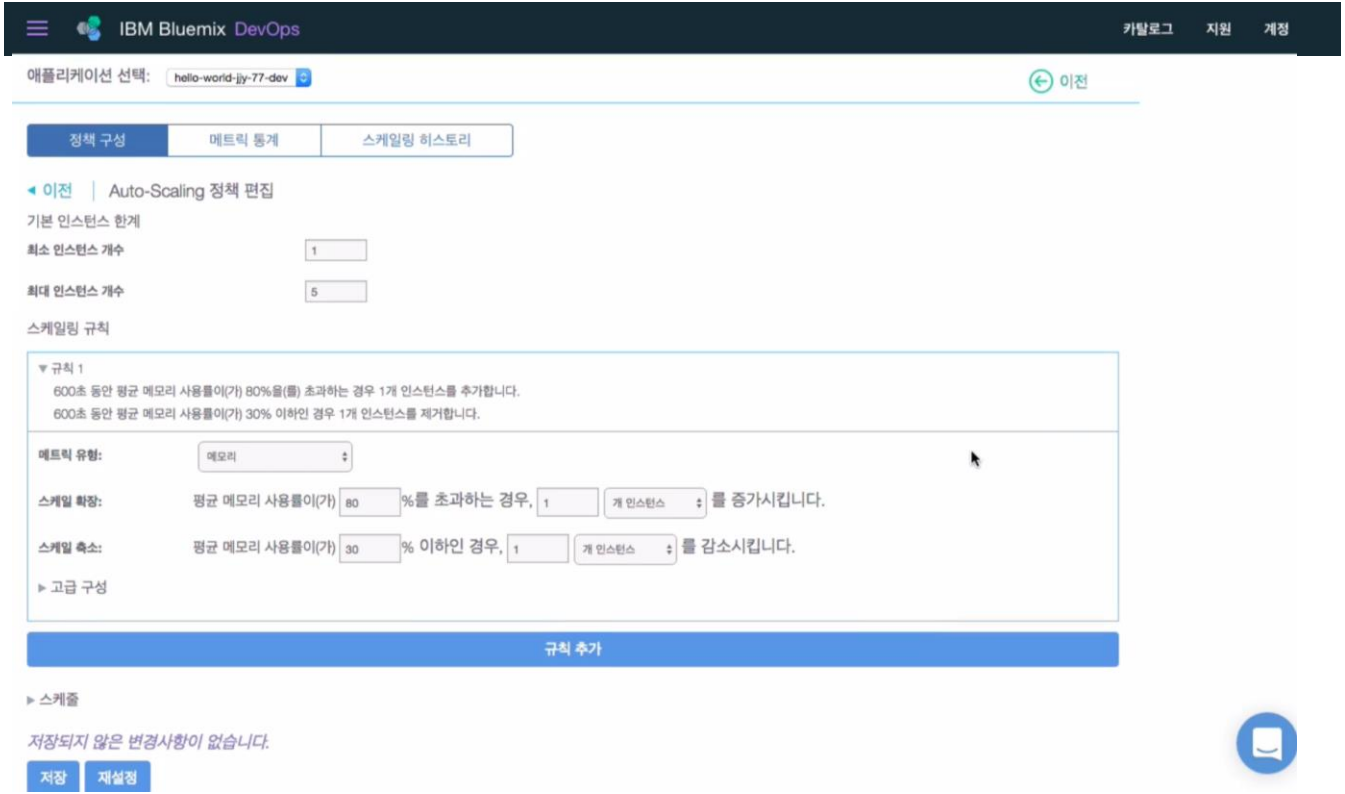
DevOps Platform

IBM Bluemix



DevOps Platform

IBM Bluemix – Auto Scaling



DevOps Platform

IBM Bluemix – Zero Downtime Deploy

Lab: Outer-Architect Course

- Deep dive in Kubernetes operation
- Istio Service Mesh

Kubernetes Object Model

- **apiVersion** : 해당 Object description 을 해석할 수 있는 API server 의 버전
- **kind** : 오브젝트의 타입 – 예제는 **Deployment**.
- **metadata** : 객체의 기본 정보. 예) 이름
- **spec (spec and spec.template.spec)** : 원하는 ”Desired State” 의 세부 내역. 예제에서는 3개의 replica를 template 내의 pod 정의대로 찍어내어 유지하라는 desired state 설정임
- **spec.template.spec** : defines the desired state of the Pod. The example Pod would be created using **nginx:1.7.9**.

Once the object is created, the Kubernetes system attaches the **status** field to the object

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Deployment object Example

Lab Time: 기본적인 kubectl명령어

- 객체 목록 불러오기
  - “kubectl get [객체 타입]“ Ex) kubectl get pods : pod 목록 불러온다.
- 객체 삭제하기
  - “kubectl delete [객체 타입] [객체 이름]”
    Ex) “kubectl delete pods wordpress-5bb5dddcff-kwgf8”
  - “kubectl delete [객체타입,객체타입,…] --all“
    Ex) “kubectl delete services,depeloyments,pods –-all”
    - 실습 중에 잘못 생성되었거나 초기화가 필요할 경우 사용
- 객체 상세 설명 확인하기
  - “kubectl describe [객체 타입] [객체 이름]”
    Ex) “kubectl describe service wordpress”

이미지를 통한 어플리케이션 배포

- 현재 작동 중인 pod들이 없는지 확인
  - kubectl get pods

- Nginx이미지 예제로 배포하기
  - kubectl run first-deployment --image=nginx

- 실행된 Pods 확인
  - kubectl get pods
    (검색된 Pod의 이름을 복사한다. (각 pod들은 고유한 이름을 갖는다.)

Pod에 접속하기

- 로그 보기
  - kubectl logs [복사된 pod 이름] –f
- 복사된 pod이름으로 접속하기
  - kubectl exec -it [복사된 pod 이름] -- /bin/bash
- Pod 내에서 접속하기 (위 명령어로 진입후에는 리눅스 명령어를 받는다)
  - 해당 경로에 있는 html 파일을 호출하는 어플리케이션이다.
    echo Hello nginx
  - Curl 명령어를 사용하기 위한 업데이트를 한다.
    apt-get update
    apt-get curl
  - Curl 명령어로 호출하기
    curl localhost

서비스 접속 문제해결하기

- **첫번째** **해당 pod 의 log 를 확인한다.**
  - kubectl logs [pod 이름] –f

- **두번째** **해당 서비스가 localhost 로 접근이 되는가를 해당 컨테이너 내부에서 확인**
  - e.g.
    kubectl exec -it [복사된 pod 이름] -- /bin/bash
    curl localhost
    위와 같이 확인했을때 서비스가 연결된다면, 다음단계 확인

- **세번째** **해당 서비스의 도커이미지에서 Port 설정이 제대로 되었는가?**
  - e.g.
    Nginx 서비스는 80으로 노출되었는데, Dockerfile 에서 EXPOSE 8080 으로 설정되었다면,
    해당 서비스는 도커 컨테이너 밖으로 포트를 노출하지 못함
    EXPOSE 80 으로 수정해야 함

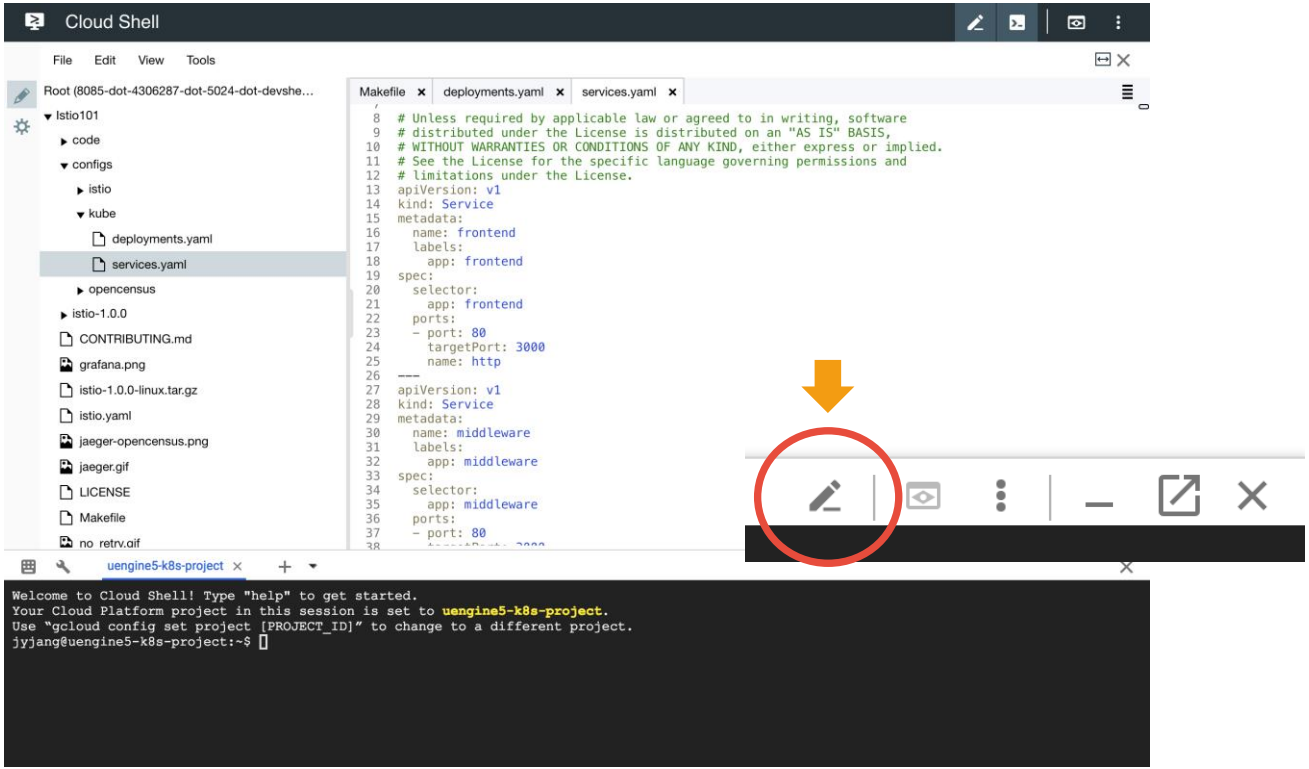
- **네번째** **해당 쿠버네티스 Service yaml 설정에서 port 넘버가 잘못된 경우**
  - 도커이미지의 포트넘버를 쿠버네티스의 Service yaml 이 자동으로 가져오지 못하므로
    spec: port: 80 ← 이 부분을 확인 도커파일에서 EXPOSE 한 포트와 동일한지 확인
    protocol: TCP
    targetPort: 80

설정 파일을 통한 pod 배포

- 아래 내용으로 nano editor 를 이용하여 declarative-pod.yaml 파일을 만든다.
  - Nginx 이미지를 기반으로 pod를 배포하는 설정파일이다

```
apiVersion: v1
kind: Pod
metadata:
  name: declarative-pod
spec:
  containers:
  - name: memory-demo-ctr
    image: nginx
```

웹 편집기 사용


Cloud Shell
File Edit View Tools
Root (8085-dot-4306287-dot-5024-dot-devshe...
Istio101
code
configs
istio
kube
deployments.yaml
services.yaml
opencensus
istio-1.0.0
CONTRIBUTING.md
grafana.png
istio-1.0.0-linux.tar.gz
istio.yaml
jaeger-opencensus.png
jaeger.gif
LICENSE
Makefile
no_retry.gif
Makefile
deployments.yaml
services.yaml
Unless required by applicable law or agreed to in writing, software
distributed under the License 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.
apiVersion: v1
kind: Service
metadata:
name: frontend
labels:
app: frontend
spec:
selector:
app: frontend
ports:
- port: 80
targetPort: 3000
name: http
---
apiVersion: v1
kind: Service
metadata:
name: middleware
labels:
app: middleware
spec:
selector:
app: middleware
ports:
- port: 80
uengine5-k8s-project
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to uengine5-k8s-project.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
jyjang@uengine5-k8s-project:~$

설정 파일을 통한 pod 배포

- nano declarative-pod.yaml 파일로 배포를 한다.
  - kubectl create -f declarative-pod.yaml
    (-f 는 파일 경로를 설정해서 배포할 수 있는 옵션이다.)

- 배포된 pods들을 검색하여 접속을 해보자
  - 이름이 설정대로 declarative-pod 로 생성되었다.
    kubectl get pods
  - 접속해보자
    kubectl exec -it declarative-pod -- /bin/bash
    apt-get update
    apt-get install curl
    curl localhost

원하는 노드 타입에 pod 몰기

- Node를 확인하여 label 붙이기
  - kubectl get nodes
    kubectl label nodes [노드이름] disktype=ssd
- Label과 함께 노드 목록을 불러오기
  - kubectl get nodes --show-labels
- 설정 파일생성(오른쪽 내용)
  - nano dev-pod.yaml
- 설정파일을 기반으로 pod 배포
  - kubectl create -f dev-pod.yaml
- 생성된 pod의 상세 내용 확인
  - kubectl get pods -o wide

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

#여기가 중요

Pod initialization

- Initl.yaml 파일을 만든다.
  - Nano init.yaml (오른쪽 내용을 복붙)
- Pod 생성
  - kubectl create -f init.yaml
- Pod 접속하기
  - kubectl exec -it init-demo -- /bin/bash
    apt-get update
    apt-get install curl
    curl localhost

```
apiVersion: v1
kind: Pod
metadata:
  name: init-demo
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: workdir
      mountPath: /usr/share/nginx/html
initContainers:
  - name: install
    image: busybox
    command:
    - wget
    - "-O"
    - "/work-dir/index.html"
    - http://kubernetes.io
    volumeMounts:
    - name: workdir
      mountPath: "/work-dir"
  dnsPolicy: Default
  volumes:
  - name: workdir
    emptyDir: {}
```

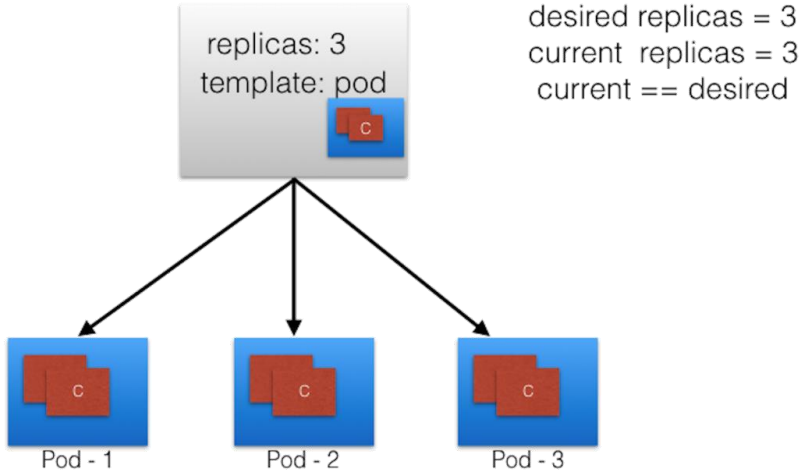
<<해당 컨테이너들은 pod 초기화단계에 실행이 된다

ReplicationControllers

- It is a part of the master node's controller manager.
- It makes sure the specified number of replicas for a Pod is running at any given point in time.
  - If there are more Pods than the desired count, the ReplicationController would kill the extra Pods, and, if there are less Pods, then the ReplicationController would create more Pods to match the desired count.
- ReplicationController creates and manage Pods.
  - Generally, Pods don’t get deployed independently, as it would not be able to re-start itself, if something goes wrong.

ReplicaSets

- A ReplicaSet (rs) is the next-generation ReplicationController. ReplicaSets support both equality- and set-based selectors, whereas ReplicationControllers only support equality-based Selectors.

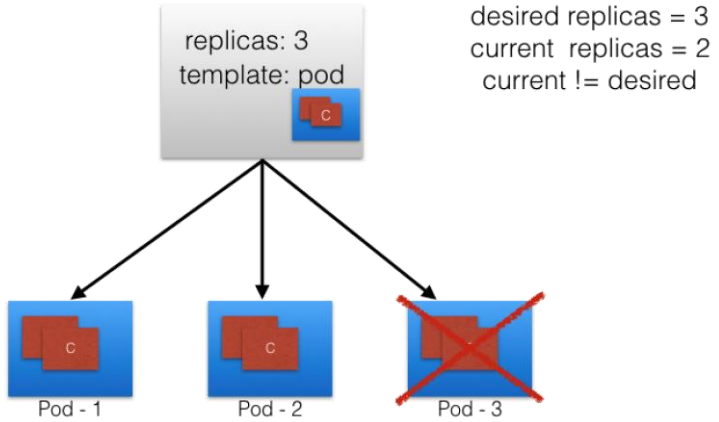


ReplicaSets

- The ReplicaSet will detect that the current state is no longer matching the desired state.
  - (In the given scenario, the ReplicaSet will create one more Pod)

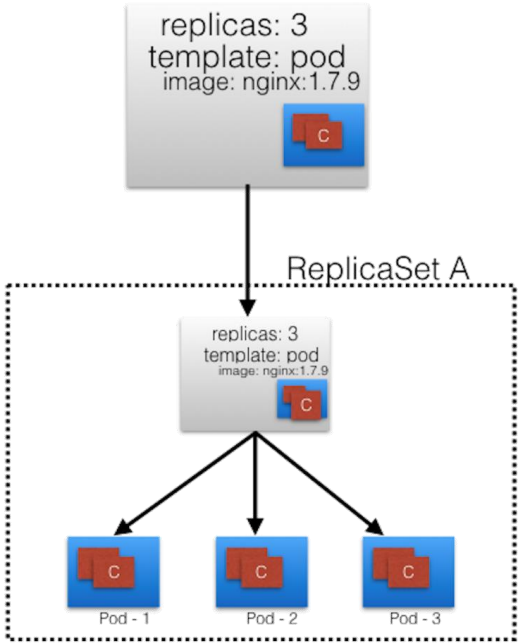
- ReplicaSets can be used independently, but they are mostly used by Deployments to orchestrate the Pod creation, deletion, and updates.

- A Deployment automatically creates the ReplicaSets, and we do not have to worry about managing them.



Deployments

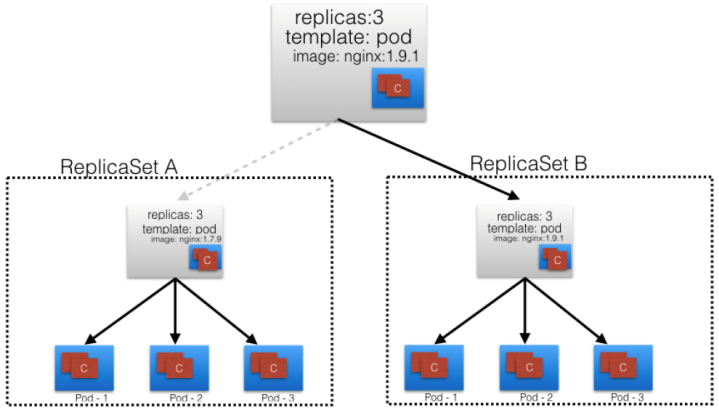
- Deployment objects provide declarative updates to Pods and ReplicaSets.
- The DeploymentController is part of the master node's controller manager, and it makes sure that the current state always matches the desired state.
- In the following example, we have a **Deployment** which creates a **ReplicaSet A. ReplicaSet A** then creates **3 Pods.** In each Pod, one of the containers uses the **nginx:1.7.9** image.



Deployments

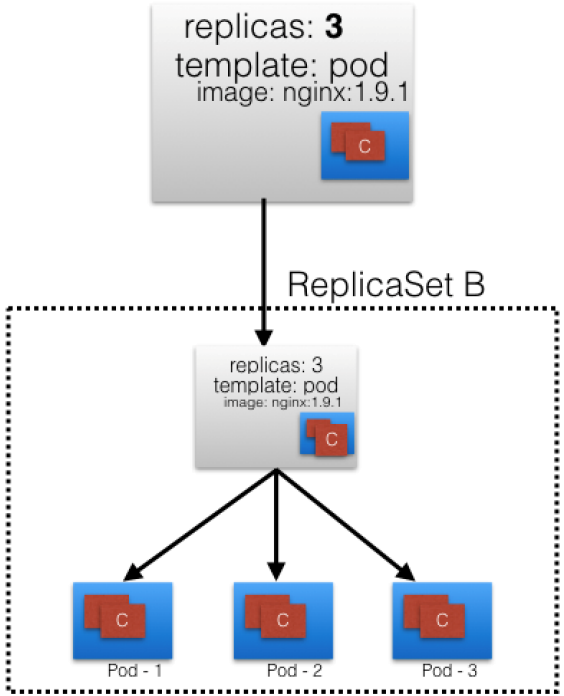
- Now, in the Deployment, we change the Pods Template and we update the image for the **nginx** container from **nginx:1.7.9** to **nginx:1.9.1.** As have modified the Pods Template, a new **ReplicaSet B** gets created. This process is referred to as a **Deployment rollout.**

- A rollout is only triggered when we update the Pods Template for a deployment. Operations like scaling the deployment do not trigger the deployment.



Deployments

- Once **ReplicaSet B** is ready, the Deployment starts pointing to it.
- On top of ReplicaSets, Deployments provide features like Deployment recording, with which, if something goes wrong, we can rollback to a previously known state.



Lab time: ReplicaSet

- ReplicaSet 파일 생성
  - nano frontend.yaml
- 파일을 기반으로 ReplicaSet 배포
  - kubectl create -f frontend.yaml
- ReplicaSet을 확인
  - kubectl get pods
    kubectl describe rs/frontend
- RsplicaSet을 삭제
  - kubectl delete rs/frontend
    (관련된 pod 전부 제거한다.)
  - kubectl delete rs/frontend --cascade=false
    (ReplicaSet 은 제거하지만 Pod 는 유지)

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        ports:
        - containerPort: 80
```

Deployment 생성

- 설정 파일을 생성
  - nano nginx-deployment.yaml
- 파일을 기반을 배포
  - kubectl create -f nginx-deployment.yaml
- 생성된 deployment 확인
  - kubectl describe deployment nginx-deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Scaling

- Deployment의 replica의 개수를 확인한다.
  - kubectl get pods
- Deplyment의 이름을 복사한다.
  - kubectl get deployments
- 해당 Deployment의 scale 조정
  - kubectl scale deployments [deployment 이름] --replicas=3
- 변경을 확인
  - kubectl get pods

Deployments 의 변경

- Deployment 파일을 변경
  - kubectl get deployments nano nginx-deployment.yaml
    (spec 아래 항목에 replica: 3 속성을 추가)
- 변경한 파일을 적용
  - kubectl apply -f nginx-deployment.yaml
- 변경 내용을 확인
  - kubectl get pods
- 설정파일에 추가된 replica 속성을 삭제 후 다시 적용
  - nano nginx-deployment.yaml (replica)
  - kubectl apply -f nginx-deployment.yaml
  - kubectl delete pods --all
  - kubectl get pods

Rolling update

- 작동중인 pod와 deployments 확인
  - kubectl get pods
    kubectl get deployments
- 새로운 버전의 deployments 배포 및 배포 상태 확인
  - kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
    kubectl rollout status deployment/nginx-deployment
- 변경을 확인
  - kubectl get deployments
    kubectl get pods
    kubectl describe pods [pod 이름]
- Pod에 접속하여 확인
  - kubectl exec -it podname -- /bin/bash
    apt-get update
    apt-get install curl
    curl localhost

Rollback

- 현재 실행중인 객체들을 확인
  - kubectl get pods
    kubectl get deployments -o wide
- 객체를 롤백 처리
  - kubectl rollout undo deployment/nginx-deployment
- 진행을 확인
  - kubectl get deployments -o wide

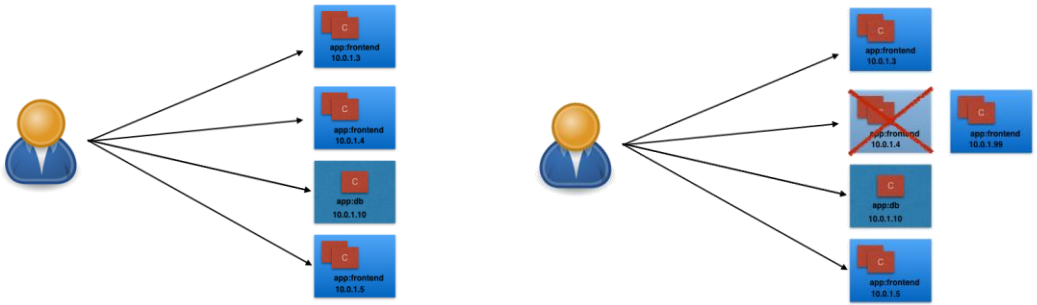
Scaling Deployments

- 현재 실행중인 객체들을 확인
  - kubectl get pods
    kubectl get deployments
- Deployemts의 속성을 kubectl 명령어로 변경
  - kubectl scale deployments example --replicas=3
- 변경된 객체를 확인
  - kubectl get pods

Connecting Users to Pods

To access the application, a user/client needs to connect to the Pods. As Pods are ephemeral in nature, resources like IP addresses allocated to it cannot be static. Pods could die abruptly or be rescheduled based on existing requirements.

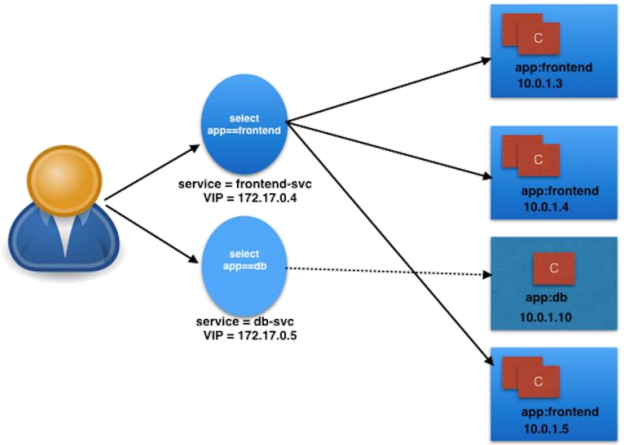
Connecting Users to Pods



- When user/client is connected to a Pod using its IP address. Unexpectedly, the Pod to which the user/client is connected can die, and a new Pod is created by the controller.
- The new Pod will have a new IP address, which will not be known automatically to the user/client of the earlier Pod.
- To overcome this situation, Kubernetes provides a higher-level abstraction called Service, which logically groups Pods and a policy to access them. (This grouping is achieved via Labels and Selectors)

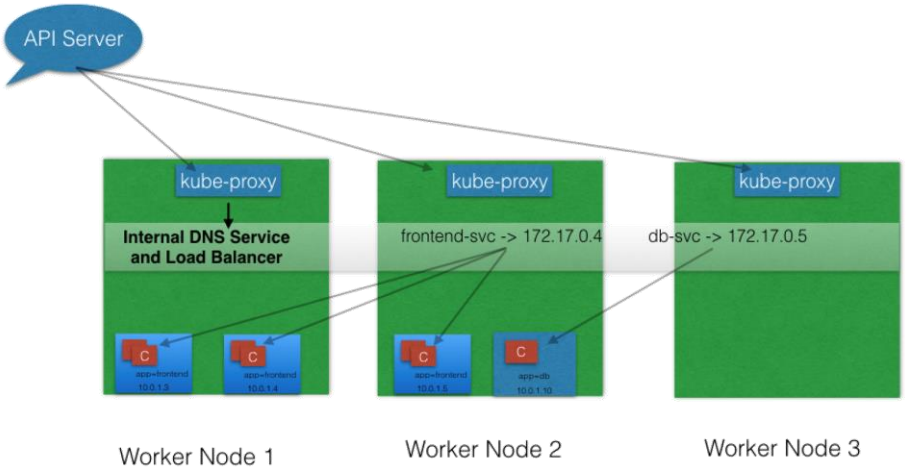
Services

- Using selectors, we can group them into two logical groups: one with 3 Pods, and one with just one Pod.
- We can assign a name to the logical grouping, referred to as a **Service name**.
- The user/client now connects to a service via the IP address, which forwards the traffic to one of the Pods attached to it.
  - The IP address attached to each Service is also known as the ClusterIP for that Service.
- A service does the load balancing while selecting the Pods for forwarding the data/traffic.
  - If the target port is not defined explicitly, then traffic will be forwarded to Pods on the port on which the Service receives traffic.
  - A tuple of Pods, IP addresses, along with the targetPort is referred to as a **Service endpoint**



kube-proxy

- All of the worker nodes run a daemon called kube-proxy, which watches the API server on the master node for the addition and removal of Services and endpoints.
- For each new Service, on each node, **kube-proxy** configures the iptables rules to capture the traffic for its ClusterIP and forwards it to one of the endpoints.
- When the service is removed, **kube-proxy** removes the IPtables rules on all nodes as well.



Service Discovery

As Services are the primary mode of communication in Kubernetes, we need a way to discover them at runtime.
Kubernetes supports two methods of discovering a Service:

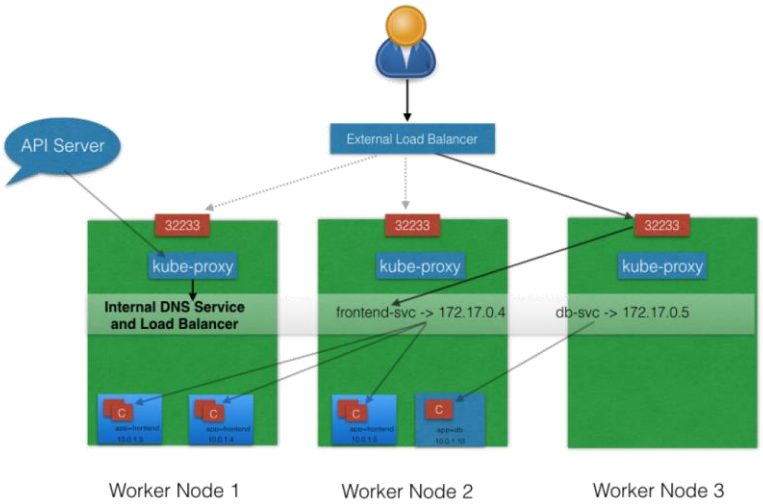
- **Environment Variables**
  As soon as the Pod starts on any worker node, the **kubelet** daemon running on that node adds a set of environment variables in the Pod for all active Services.
- **DNS**
  Kubernetes has an add-on for DNS, which creates a DNS record for each Service and its format is like **my-svc.my-namespace.svc.cluster.local**. Services within the same Namespace can reach to other Services with just their name. Pods from other Namespaces can reach the Service by adding the respective Namespace as a suffix. EX) **namespace2.pod1**.

This is the most common and highly recommended solution.

ServiceType : LoadBalancer

- With the **LoadBalancer** *ServiceType*:
  - NodePort and ClusterIP Services are automatically created, and the external load balancer will route to them
  - The Services are exposed at a static port on each worker node
  - The Service is exposed externally using the underlying cloud provider's load balancer feature.

The LoadBalancer *ServiceType* will only work if the underlying infrastructure supports the automatic creation of Load Balancers and have the respective support in Kubernetes, as is the case with the Google Cloud Platform and AWS.



Custom Resources

- In Kubernetes, a **resource** is an API endpoint which stores a collection of API objects.
- Although in most cases existing Kubernetes resources are sufficient to fulfill our requirements, we can also create new resources using **custom resources**.
  - Modifing Kubernetes source. Is not needed for Custom resources
  - Custom resources are dynamic in nature, and they can appear and disappear in an already running cluster at any time.
- To make a resource declarative, we must create and install a **custom controller**, which can interpret the resource structure and perform the required actions.
  - Custom controllers can be deployed and managed in an already running cluster.
- There are two ways to add custom resources:
  - **Custom Resource Definitions (CRDs)**
    This is the easiest way to add custom resources and it does not require any programming knowledge. However, building the custom controller would require some programming.
  - **API Aggregation**
    For more fine-grained control, we can write API Aggregators. They are subordinate API servers which sit behind the primary API server and act as proxy.

Helm

- To deploy an application, we use different Kubernetes manifests.
  - Such as Deployments, Services, Volume Claims, Ingress, etc. Sometimes,
  - It can be tiresome to deploy them one by one.
- We can bundle all those manifests after templatizing them into a well-defined format, along with other metadata. Such a bundle is referred to as *Chart*.
  - These Charts can then be served via repositories, such as those that we have for rpm and deb packages.
- **Helm** is a package manager (analogous to **yum** and **apt**) for Kubernetes, which can install/update/delete those Charts in the Kubernetes cluster.
- Helm has two components:
  - A client called *helm*, which runs on your user's workstation
  - A server called *tiller*, which runs inside your Kubernetes cluster.
- The client *helm* connects to the server *tiller* to manage Charts

Install Helm

curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get | bash
add a service account within a namespace to segregate tiller
kubectl --namespace kube-system create sa tiller
create a cluster role binding for tiller
**kubectl create clusterrolebinding tiller **
**--clusterrole cluster-admin **
--serviceaccount=kube-system:tiller

echo "initialize helm"
initialized helm within the tiller service account
helm init --service-account tiller
updates the repos for Helm repo integration
helm repo update

echo "verify helm"
verify that helm is installed in the cluster
kubectl get deploy,svc tiller-deploy -n kube-system

https://medium.com/google-cloud/installing-helm-in-google-kubernetes-engine-7f07f43c536e

Monitoring and Logging

- In Kubernetes, we have to collect resource usage data by Pods, Services, nodes, etc., to understand the overall resource consumption and to make decisions for scaling a given application.
- Two popular Kubernetes monitoring solutions are **Heapster** and **Prometheus**.
  - **Heapster**
    Heapster is a cluster-wide aggregator of monitoring and event data, which is natively supported on Kubernetes.
  - **Prometheus**
    Prometheus, now part of CNCF (Cloud Native Computing Foundation), can also be used to scrape the resource usage from different Kubernetes components and objects. Using its client libraries, we can also instrument the code of our application.
- Another important aspect for troubleshooting and debugging is Logging, in which we collect the logs from different components of a given system.
  - In Kubernetes, we can collect logs from different cluster components, objects, nodes, etc.
  - The most common way to collect the logs is using Elasticsearch, which uses fluentd with custom configuration as an agent on the nodes.
    (**fluentd** is an open source data collector, which is also part of CNCF.)

Liveness and Readiness Probes

- While we are discussing application deployment, let's talk about **Liveness** and **Readiness Probes**. These probes are very important, because they allow the kubelet to control the health of the application running inside a Pod's container.

Liveness

- If a container in the Pod is running, but the application running inside this container is not responding to our requests, then that container is of no use to us.
  - This kind of situation can occur, for example, due to application deadlock or memory pressure.
  - In such a case, it is recommended to restart the container to make the application available.
- Rather than doing it manually, we can use **Liveness Probe**. Liveness probe checks on an application's health, and, if for some reason, the health check fails, it restarts the affected container automatically.
- Liveness Probes can be set by defining:
  - Liveness command
  - Liveness HTTP request
  - TCP Liveness Probe.

Liveness Command

- In the following example, we are checking the existence of a file **/tmp/healthy**:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 3
      periodSeconds: 5
```

Liveness Command

- The existence of the **/tmp/healthy** file is configured to be checked every 5 seconds using the **periodSeconds** parameter.
- The **initialDelaySeconds** parameter requests the kubelet to wait for 3 seconds before doing the first probe.
- When running the command line argument to the container, we will first create the **/tmp/healthy** file, and then we will remove it after 30 seconds.
- The deletion of the file would trigger a health failure, and our Pod would get restarted.

Liveness HTTP Request

- In the following example, the kubelet sends the **HTTP GET** request to the **/healthz** endpoint of the application, on port **8080**.
- If that returns a failure, then the kubelet will restart the affected container; otherwise, it would consider the application to be alive.

```
livenessProbe:
    httpGet:
      path: /healthz
      port: 8080
      httpHeaders:
      - name: X-Custom-Header
        value: Awesome
    initialDelaySeconds: 3
    periodSeconds: 3
```

TCP Liveness Probe

- With TCP Liveness Probe, the kubelet attempts to open the TCP Socket to the container which is running the application.
- If it succeeds, the application is considered healthy, otherwise the kubelet would mark it as unhealthy and restart the affected container.

```
livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

Readiness Probes

- Sometimes, applications have to meet certain conditions before they can serve traffic. These conditions include ensuring that the depending service is ready, or acknowledging that a large dataset needs to be loaded, etc.
  - In such cases, we use **Readiness Probes** and wait for a certain condition to occur. Only then, the application can serve traffic.
- A Pod with containers that do not report ready status will not receive traffic from Kubernetes Services.
- Readiness Probes are configured similarly to Liveness Probes. Their configuration also remains the same.

```
readinessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

Lab time: Liveness 와 readiness probe

- exec-liveness 설정파일을 생성
  - nano exec-liveness.yaml
- 파일 설정으로 배포
  - kubectl create –f exec-liveness.yaml
- 내용 확인
  - kubectl describe pod liveness-exec

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf
/tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

Liveness 와 readiness probe

- http-livenes 설정파일을 생성
  - nano http-liveness.yaml
- 파일 설정으로 배포
  - kubectl create -f http-liveness.yaml
- 내용 확인
  - kubectl describe pod liveness-http

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

Liveness 와 readiness probe

- tcp-liveness-readiness 파일을 생성
  - nano tcp-liveness-readiness.yaml
- 파일 설정으로 배포
  - kubectl create -f tcp-liveness-readiness.yaml
- 내용 확인
  - kubectl describe pod goproxy

```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

Container lifecycle 후킹

- lifecycle.yaml 파일 생성
  - nano lifecycle.yaml
- 파일 설정으로 배포
  - kubectl create -f lifecycle.yaml
- Pod에 적용된 내용 확인
  - kubectl get pod lifecycle-demo

    kubectl exec -it lifecycle-demo -- /bin/bash cat /usr/share/message

```
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
  - name: lifecycle-demo-container
    image: nginx
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c", "echo Hello
from the postStart handler >
/usr/share/message"]
      preStop:
        exec:
          command: ["/usr/sbin/nginx","-s","quit"]
```

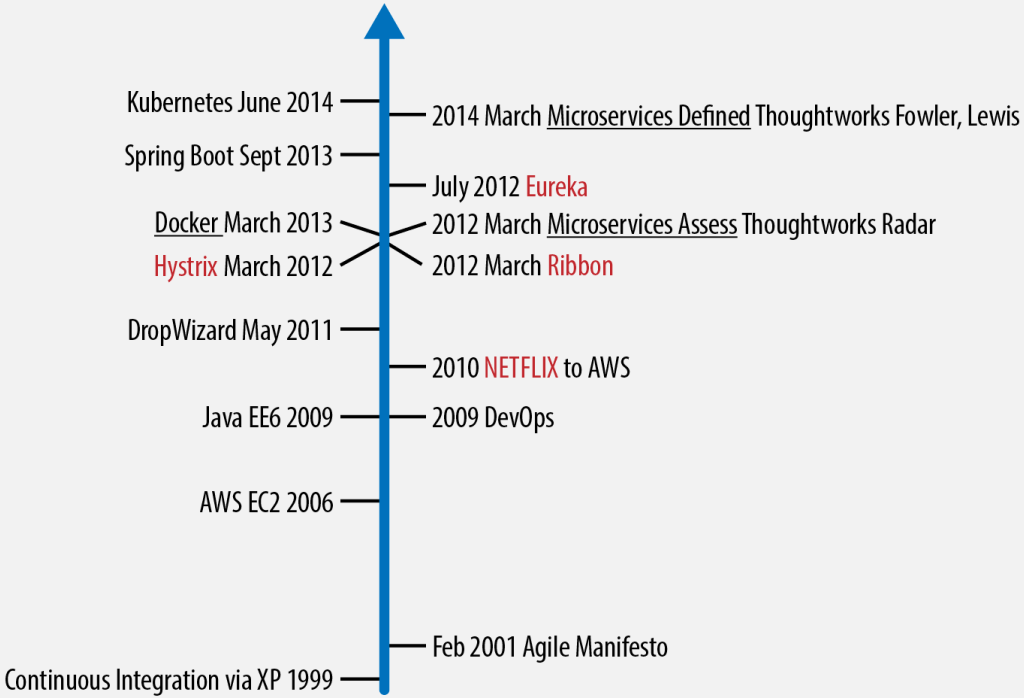
- Deploying a Multi-Tier Application

Workflowy에 있는 제로 베이스의 디플로이 내용으로 대체

Istio Service Mesh

- Improved Service Resilience
- Improved Smart Deployment
- Improved Observability
- Improved Security

Microservices timeline



K8S + Spring Cloud + Istio

Capabilities				Spring Cloud with Kubernetes & Istio on IaaS+
DevOps Experience			Kubernetes	Self service, multi-environment capabilities
Auto Scaling & Self Healing				Pod/Cluster Autoscaler, HealthIndicator, Scheduler, Pool Ejection
Deployment & Scheduling		Istio		Deployment strategy, DarkLaunch, A/B, Canary, Scheduler Strategy
Resilience & Fault Tolerance	Spring Cloud			HealthIndicator, Hystrix, HealthCheck, Process Check, Circuit Breaker/Timeout/Retry
Distributed Tracing				Zipkin, Jaeger
Centralized Metrics				Heapster, Prometheus, Grafana
Centralized Logging				EFK
API Gateway				Zuul, Traffic Control, Egress
Load Balancing				Ribbon, Service, Envoy
Chaos Engineering				Chaos Monkey for Spring Boot, Chaos Toolkit Kubernetes, Envoy
Service Discovery				Service
Configuration Management				Externalized Configuration, ConfigMaps, Secrets
Application Packaging				Spring Boot Maven/Gradle plugin
Job Management				Spring Batch, Scheduled Jobs
Process Isolation				Docker, Pods, Envoy
Environment Management				Namespaces, Authorization
Resource Management				CPU and Memory Limits, Namespace resource quotas
Operating System	IaaS+			Ubuntu, Atomic, ...
Virtualization				VMWare, Openstack, ...
Hardware, Storage, Networking				AWS, GCP, Azure, ...

Istio 로 할 수 있는 것들:

1. Improved Routing & Deployment Strategy

- Ingress / Egress Gateway
- Canary Deploy
  - 특정 유저의 신상, 지역, 권한, 접근 단말에 따른 다른 버전의 노출
- AB Testing / Shadow Deploy
  - 신규 버전의 오류 노출 없는 실질적 테스트

Istio 로 할 수 있는 것들:

2. Improved Resilience

- Retry
  - 서비스간 호출의 실패에 대한 재시도
- Circuit Breaking / Rate Limiting
  - 인스턴스의 보호
  - 전체 서비스 장애 차단
- Pool Ejection
  - 죽은 인스턴스의 제외

- Circuit Breaking + Pool Ejection + Retry
  = High Resilience

Istio 로 할 수 있는 것들:

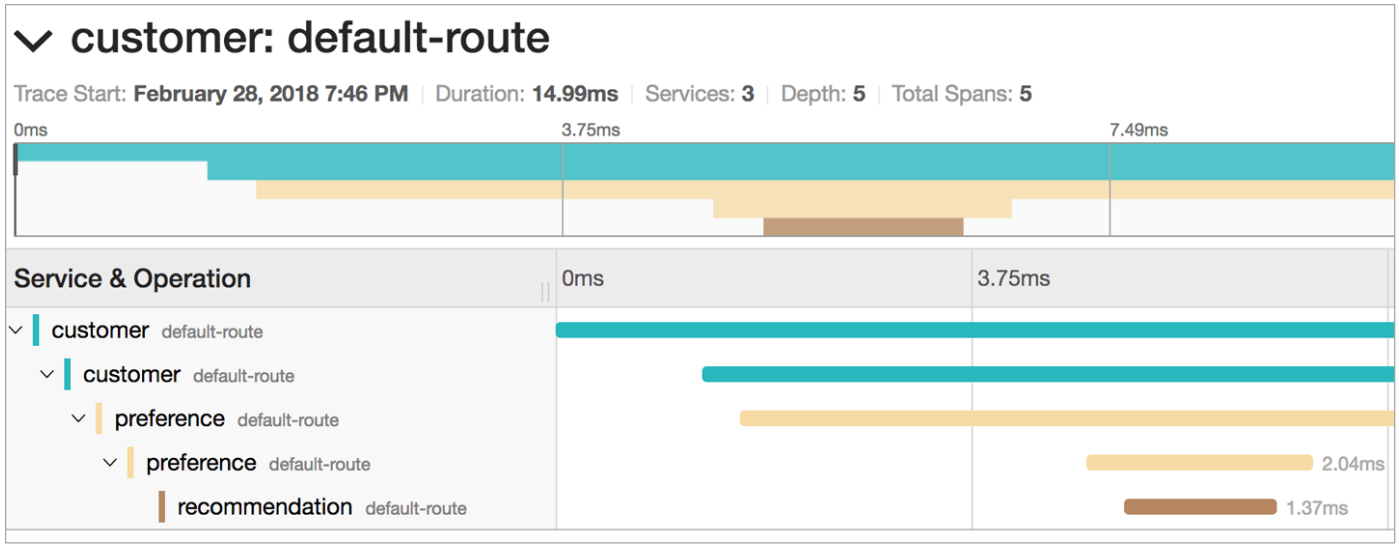
3. Improved Security

- TLS based Inter-Mi-services Communication
  - By Auth (신규모듈)
- Whitelist and Blacklist

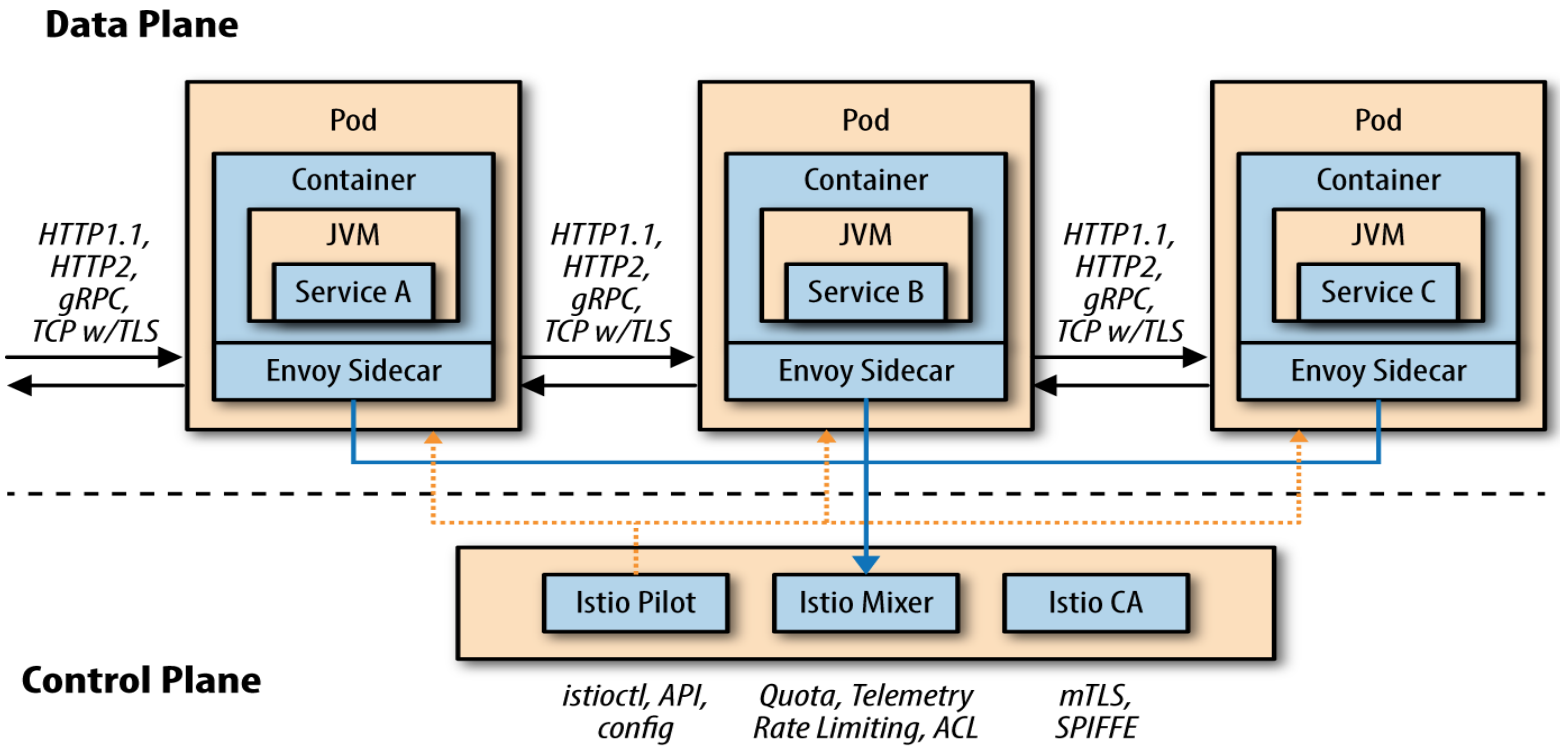
Istio 로 할 수 있는 것들:

4. Improved Observability

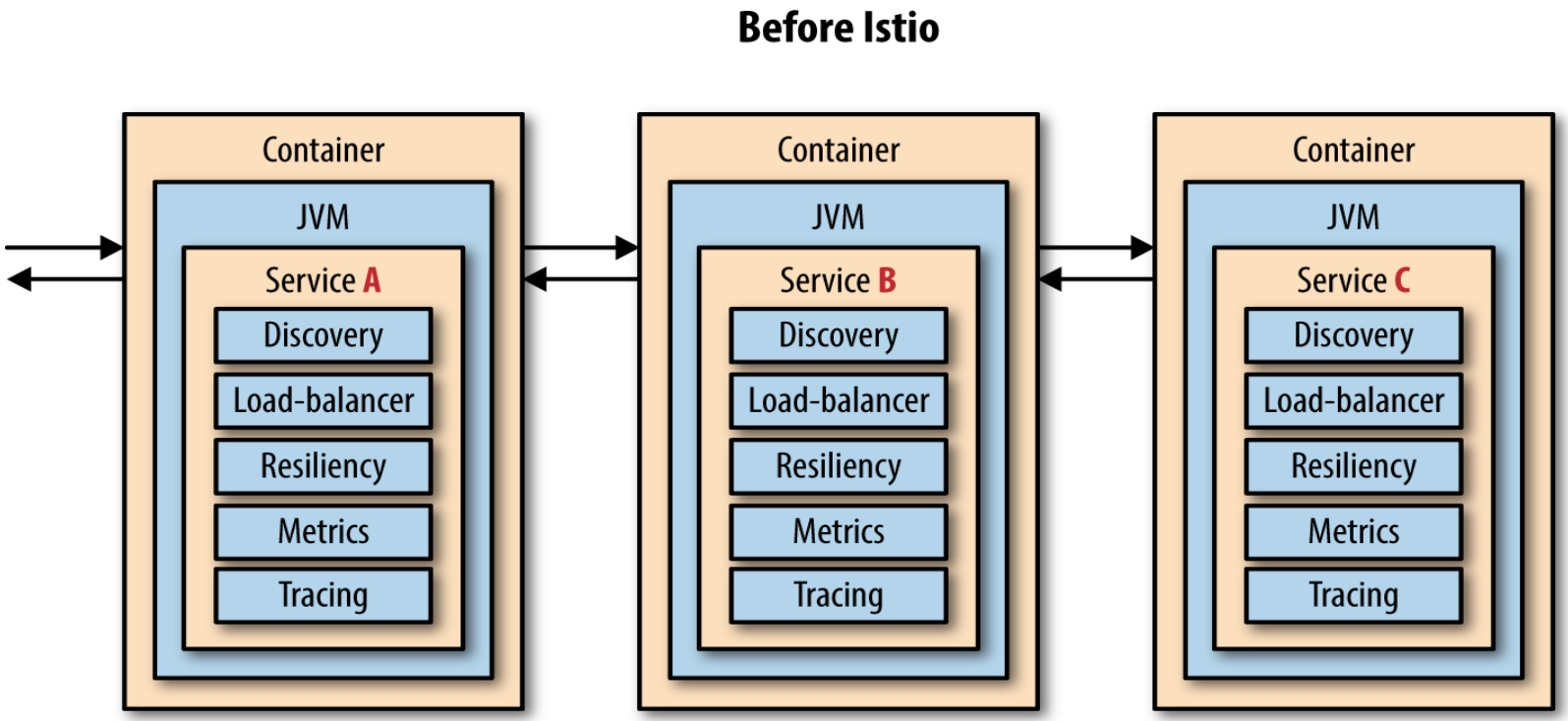
- Distributed Tracing and Measure
  - 서비스간 호출의 내용 기록



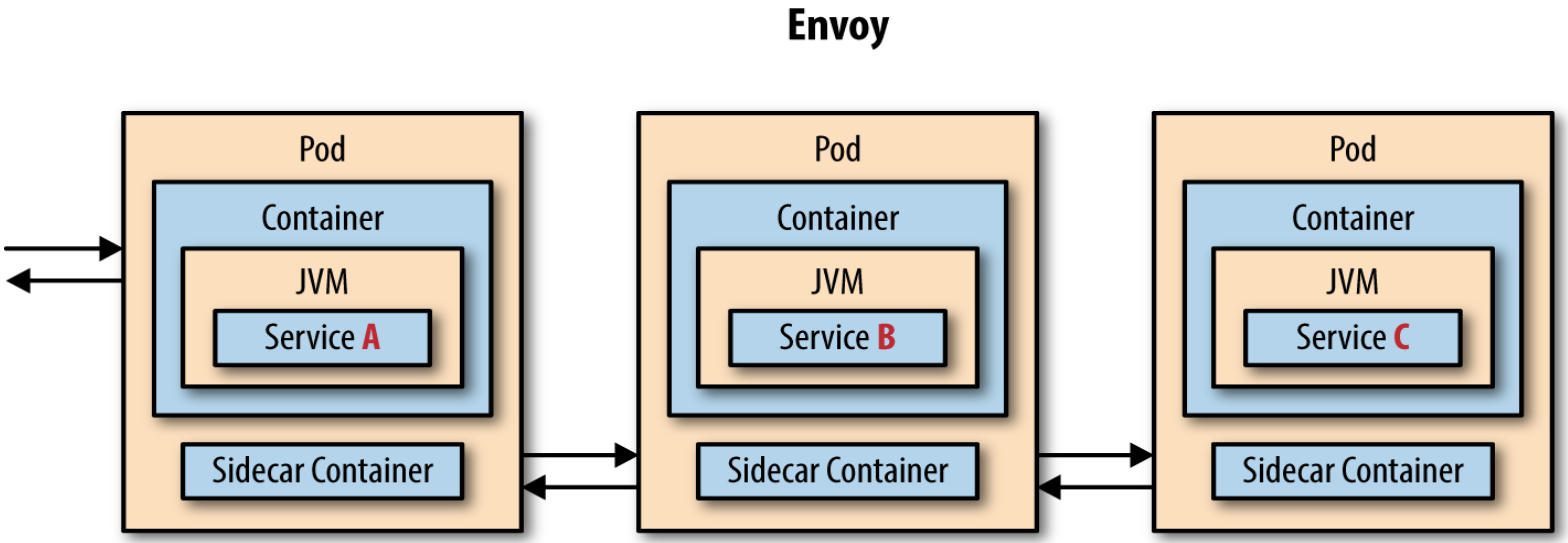
Istio: Components



Spring Cloud + Netflix

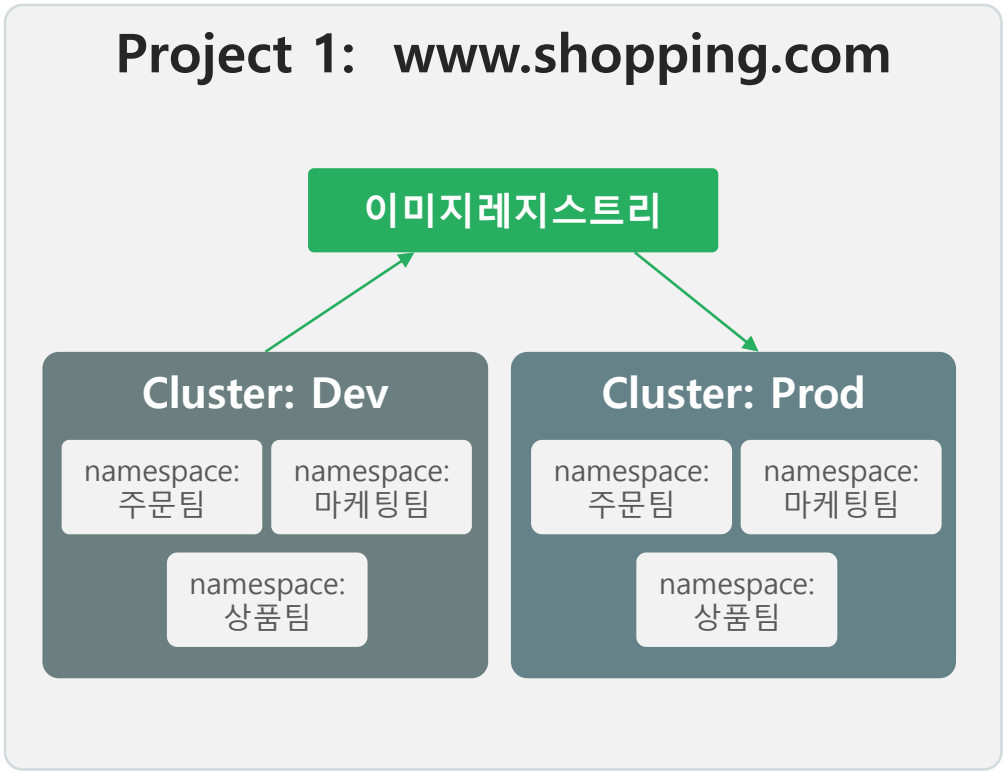


After Istio



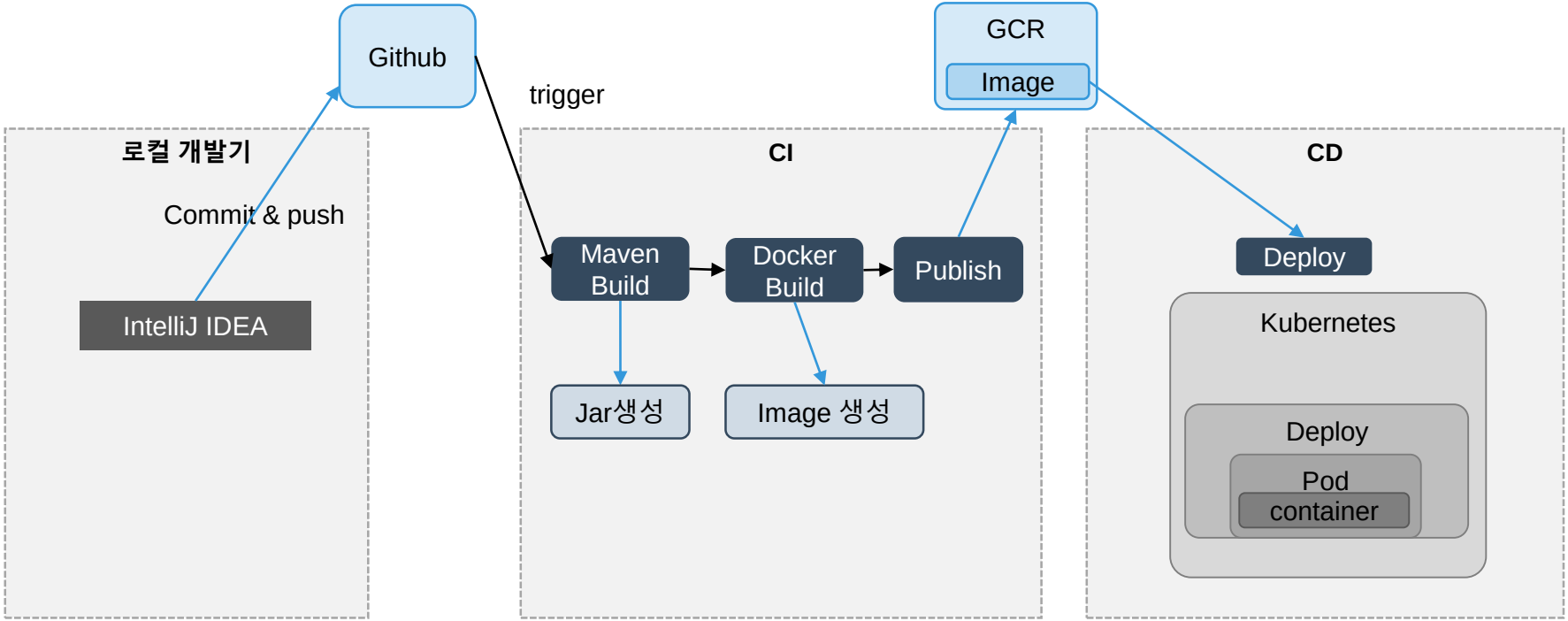
좋은점:
1. L7 레이어를 사용, 성능이 높음
2. Code 변경 없이 Cross-cutting 이슈를 다루어줌
3. Main 서비스의 재배포 없이 Sidecar 를 관리 가능함

Tip: GCP 프로젝트와 클러스터 구성의 좋은예



- **하나의 사이트는 프로젝트로 분리:**
  해당 사이트 운영에 필요한 모든 리소스 목록이 관리되어 프로젝트를 제거하면 관련된 모든 GCP 리소스를 해제시킬 수 있음
- **개발기와 운영기 등은 쿠버네티스 클러스터로 분리:**
  클러스터가 분리되면 네트워크 호출이 불가하여 엄격하게 분리 가능 하며, 공유된 이미지레지트리 통해서만 배포할 정확한 이미지(기능의 버전)를 공유
- **쿠버네티스 네임스페이스를 각 팀에게 배정:**
  논리적인 클러스터 내부의 분리단위, 개발자 권한이 없는경우 네임스페이스를 넘어선 pod 관리 권한이 제공되지 않음. But, 네트워크 호출은 가능(inter-마이크로서비스 통신 허용)

Lab 2 - Automated CI/CD



Google Cloud Build (cloudbuild.yaml 실행)

cloudbuild.yaml - CI Part

```
steps:
  ### Step 1. Test
  - id: 'test'
    name: 'gcr.io/cloud-builders/mvn'
    args: [
      'test',
      '-Dspring.profiles.active=test'
    ]
```

```
  ### Step 2. Build
  - id: 'build'
    name: 'gcr.io/cloud-builders/mvn'
    args: [
      'clean',
      'package',
      '-Dmaven.test.skip=true'
    ]
```

```
  ### Step 3. dockerizing
  - id: 'docker build'
    name: 'gcr.io/cloud-builders/docker'
    args:
      - 'build'
      - '--tag=gcr.io/$PROJECT_ID/$_PROJECT_NAME:$COMMIT_SHA'
      - '.'
```

```
  ### Step 4. Publish image to docker registory (GCR)
  - id: 'publish'
    name: 'gcr.io/cloud-builders/docker'
    entrypoint: 'bash'
    args:
      - '-c'
      - |
        docker push
gcr.io/$PROJECT_ID/$_PROJECT_NAME:$COMMIT_SHA
```

cloudbuild.yaml - CD Part

```
### deploy
 - id: 'deploy'
   name: 'gcr.io/cloud-builders/gcloud'
   entrypoint: 'bash'
   args:
    - '-c'
    - |
      PROJECT=$$(gcloud config get-value core/project)
      gcloud container clusters get-credentials "$${CLOUDSDK_CONTAINER_CLUSTER}" \
        --project "$${PROJECT}" \
        --zone "$${CLOUDSDK_COMPUTE_ZONE}"

      cat <<EOF | kubectl apply -f -
      apiVersion: v1
      kind: Service
      metadata:
        name: $_PROJECT_NAME
        labels:
          app: $_PROJECT_NAME
      spec:
        ports:
          - port: 8080
            targetPort: 8080
        selector:
          app: $_PROJECT_NAME
      EOF
```

1. grc.io/cloud-builders/gcloud 로 name 을 선언하면 gcloud 명령어를 사용하여 클라우드 정보 등을 조회 할 수 있다.

Tracing kiali – istio 설치

*** Cloud shell에서 다음의 스크립트 실행 ***

> curl -L https://git.io/getLatestIstio | ISTIO_VERSION=1.3.3 sh -
> cd istio-1.3.3
> export PATH=$PWD/bin:$PATH
> for i in install/kubernetes/helm/istio-init/files/crd*yaml; do kubectl apply -f $i; done
> kubectl apply -f install/kubernetes/istio-demo.yaml

*** 확인
kubectl get po,svc -n istio-system

```
parkjj780628@cloudshell:~/istio-1.3.3 (omega-byte-243400)$ kubectl get po,svc -n istio-system
NAME                                        READY   STATUS      RESTARTS   AGE
pod/grafana-575c7c4784-8p7cs                1/1     Running     0          32m
pod/istio-citadel-6cb95997f8-2kfml          1/1     Running     0          32m
pod/istio-egressgateway-6d4f69787b-6dzzk    1/1     Running     0          32m
pod/istio-galley-b877d99f4-x5nk7            1/1     Running     0          32m
pod/istio-grafana-post-install-1.3.3-qnhsc  0/1     Completed   0          32m
pod/istio-ingressgateway-67fbf57b85-6lzqv   1/1     Running     0          8m18s
pod/istio-pilot-8687d5696-m4f2t             2/2     Running     0          8m17s
pod/istio-policy-9f5d7d7b4-x5kk7            2/2     Running     0          8m17s
pod/istio-security-post-install-1.3.3-4vw6g 0/1     Completed   0          32m
pod/istio-sidecar-injector-6c65cfff5-5rl55  1/1     Running     0          32m
pod/istio-telemetry-75cbc855b5-5b6sc        0/2     Pending     0          8m17s
pod/istio-telemetry-c8fdf6c46-6fhrr         2/2     Running     2          32m
pod/istio-tracing-8456d6548f-j6mp9          1/1     Running     0          32m
pod/kiali-7dd44f7696-znshs                  1/1     Running     0          32m
pod/prometheus-5679cb4dcd-g2d7k             1/1     Running     0          32m

NAME                             TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)
service/grafana                  ClusterIP      10.4.15.128    <none>          3000/TCP
service/istio-citadel            ClusterIP      10.4.13.187    <none>          8060/TCP,15014/TCP
service/istio-egressgateway      ClusterIP      10.4.12.47     <none>          80/TCP,443/TCP,15443/TCP
service/istio-galley             ClusterIP      10.4.11.227    <none>          443/TCP,15014/TCP,9901/TCP
```

Tracing kiali – kiali 설치

*** vi 편집기로 다음의 내용 저장 ***

> Vi kiali.yaml

```
------파일 시작-------
apiVersion: v1
Kind: Secret
Metadata:
  name: kiali
  namespace: istio-system
  labels:
    app: kiali
type: Opaque
data:
  username: YWRtaW4=
  passphrase: YWRtaW4=
------파일 끝-------
```

> kubectl apply -f kiali.yaml

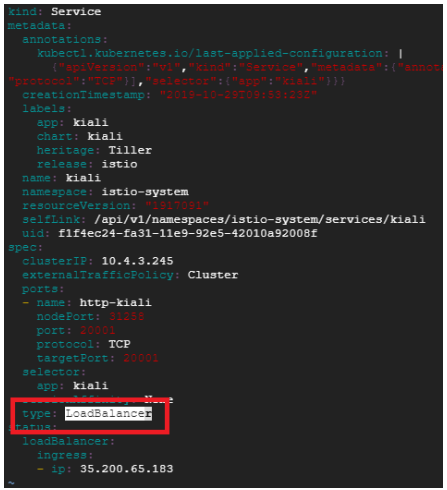
> helm template --set kiali.enabled=true install/kubernetes/helm/istio --name istio -- namespace istio-system > kiali_istio.yaml
> kubectl apply -f kiali_istio.yaml

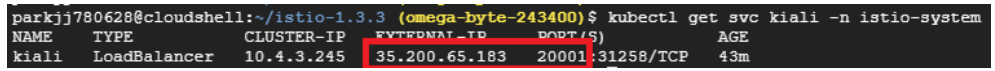
Tracing kiali – app injection 및 kiali loadbalancer

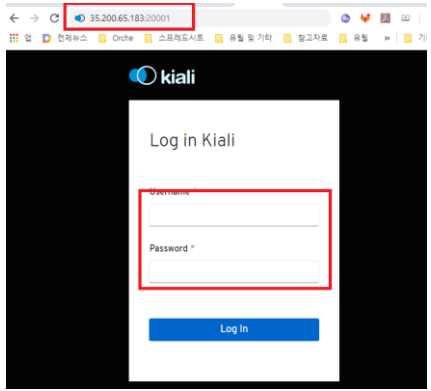
default 네임스페이스에 모든 po 들을 자동 istio 로 inject
> kubectl label namespace default istio-injection=enabled

기존의 default 네임스페이스에 있는 pods 를 모두 삭제하여 다시 띄운다
> kubectl delete po --all

kiali svc loadbalancer
> kubectl edit svc kiali -n istio-system
> 수정 후 external IP 확인 후 브라우저 접속

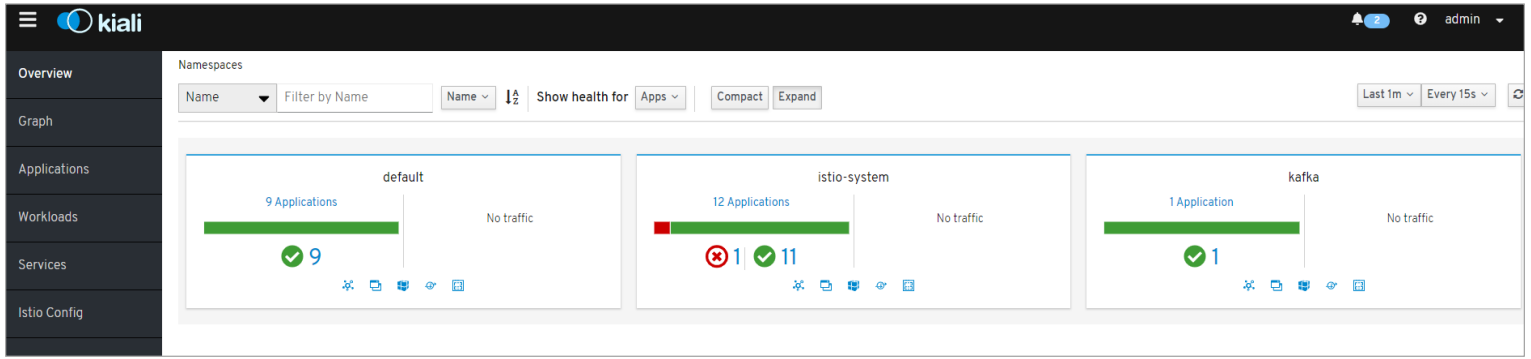


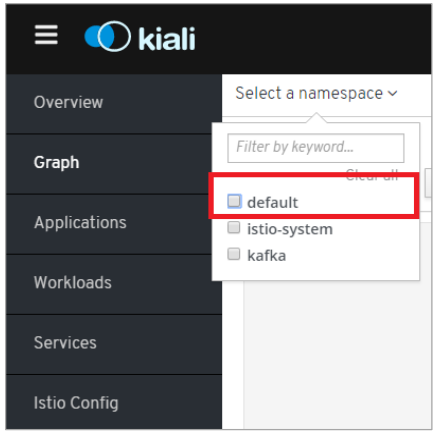


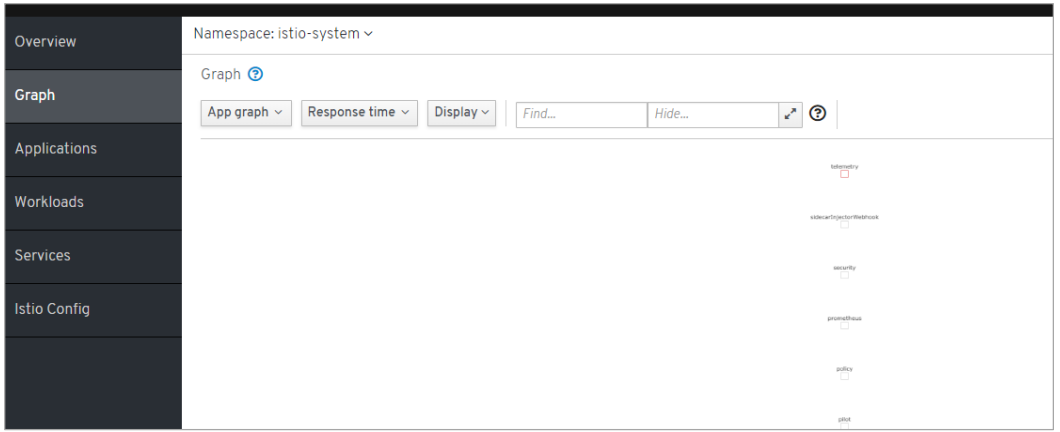


Id : admin
Pwd : admin

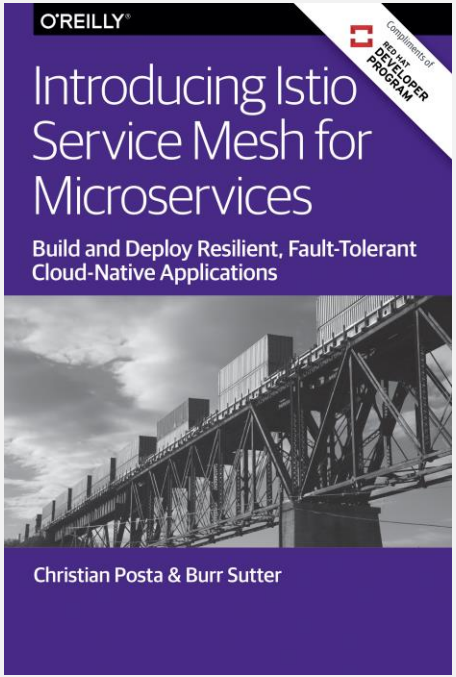
Tracing kiali – 사용







Recommended Book

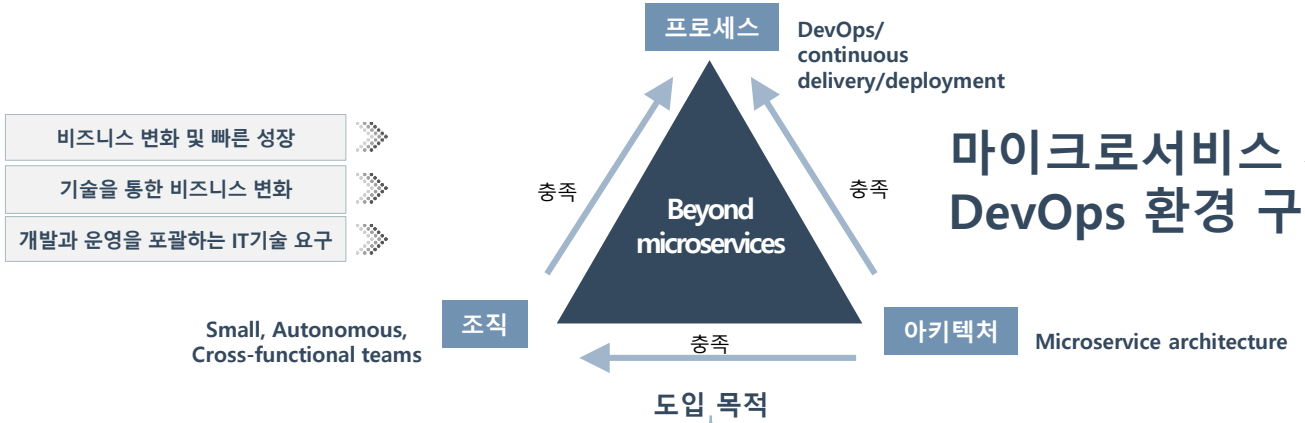


- Java (Spring Boot) based example
- Source code available
- Concise
- Free

Appendix: MSA 프로젝트 관리와 조직 변화 관리

MSA 프로젝트 목표 수립

마이크로 서비스 아키텍처는 하나 또는 소수의 큰 시스템을 작은 크기의 서비스로 나누고, 서비스 별로 개별적인 데이터 저장소를 소유하고 독립된 실행 환경을 구축함으로써 서비스 단위로 독립적인 개발, 빌드, 테스트, 배포 모니터링, 라우팅, 확장이 가능한 아키텍처 입니다.



마이크로서비스 기반 DevOps 환경 구축

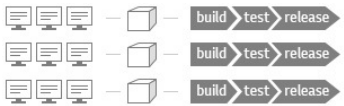
도입 목적

개발 생산성

- 서비스를 충분히 작은 크기로 나누어 독립적으로 개발하고 마이크로서비스간에는 API 규약에 따라 상호 연계
- 개발자는 비즈니스 로직에만 집중 가능
- 테스트 시에도 연관 마이크로서비스만 테스트를 수행하므로 배포 일정 축소

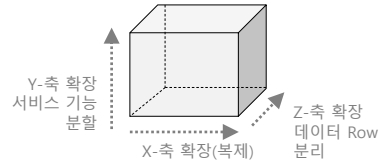
배포 유연성

- 마이크로서비스 아키텍처는 개별 서비스 단위로 개발/패키징/빌드/테스트/배포가 가능하므로 각 서비스를 각자의 로드 맵에 따라 개발 배포



확장을 통한 안정성

- 시스템의 기능을 분할하여, 서비스 단위 별로 트래픽에 대해 자율적 확장을 통한 성능 향상



Specific Goal Setting – MSA Maturity Levels

	Early	Inception	Expanding	Mature
기능분해	비즈니스 역량은 도출한 비즈니스 기능과 유즈케이스를 단위로 하여 분리 가능	뭉뚱그리지 말고 확실히 분해해라. 추출 된 각 유스 케이스와 인터페이스를 통해 액세스 할 데이터에 대해 잘 정의 된 인터페이스를 가짐.	컨텍스트매핑의 결과로 도출된 범위가 한정된 문맥 (Bounded context) 을 기준으로 분해한다. ubiquitous language 가 다른 bounded context간의 커뮤니케이션시 Anti-corruption layer를 통해 연동	이벤트 기반으로 식별된 도메인 영역, 머털 럴라이즈드 뷰, 읽기/쓰기 (커맨드) 를 위한 분리된 별도의 모델 (CQRS)
데이터	서비스간 스키마를 공유한다. 2 PC 를 사용할 수 있다. ACID 기반의 트랜잭션을 유지한다. Canonical Data Model 를 지향한다	각각의 서비스는 자신만의 데이터베이스를 가짐. 서비스들과 다중 엔터프라이즈 데이터 저장소간의 트랜젝션이 적은 조정으로 이루어짐.	-완전히 분산 된 데이터 관리.RDBMS, Search index. Document DB, Graph DB 등등 데이터를 해당 노드에 도달할 때까지는 데이터에 일관성이 없는 상태이나 일정 시간이 지나면, 다시 Consistency를 충족	이벤트 기반 데이터 관리, 이벤트 소싱 및 커맨드 쿼리
테스팅	통합, 회귀, 시스템 통합 테스트(SIT: System Integration Testing), 사용자 인수 검사(AUT: User Acceptance Testing)를 위한 부분 자동화된 유닛테스팅	Junit, Integration, Functional, SIT, UAT, Regression, Performance 의 완전한 자동 테스팅	component 테스트, A/B 테스트, 실패테스트(Chaos Monkey)	컨트랙트 테스트, 컨슈머 주도 계약, 테스트 데이터별 유저 퍼소나(persona)와 여정(journey)을 활용한 E2E 테스트
인프라 스트럭쳐	지속적인 빌드, 지속적인 통합 운용	지속적 딜리버리와 배포, 로그의 중앙 집중화	컨테이너 사용(도커), 컨테이너 지휘자(k8s), 외부 구성(유레카, 주키퍼)	자동 프로비저닝을 갖춘 PaaS기반 솔루션
배포	설치 스크립트 구동, 호스트 당 멀티 서비스 인스턴스	VM 당 하나의 서비스 인스턴스 클라이언트 사이드 로드벨런싱 서버사이드 로드벨런싱	컨테이너 당 하나의 서비스 인스턴스이고 변경 불가능한 서버, blue/green 배포	멀티 클라우드 및 멀티 데이터 센터 지원
모니터링	SPLUNK와 함께 사용되는 APM 툴(App dynamics)	LogStash, Elastic Search, Sensu, Kibana And Graphite 를 사용한 중앙집중 로깅	Docker daemon 사용하여 통계 및 상태정보 수집하고 이를 third party 모니터링 툴인 Circonus, App Dynamics. 에 전달	종합 트랜잭션, 추적 서비스 지원
거버넌스	IT와 운영부분 최고 경영진의 의사 결정을 통해 중앙화 된 거버넌스 제공	아키텍처 및 배포 결정을 담당하는 모노리스 및 마이크로서비스 팀 직원들과 공유 된 관리 모델	팀별 완전히 분산된 거버넌스, 높은 자율성, 높은 책임과 중앙 통제적 프로세스 중심 접근 방식으로로부터 벗어난다. 각 팀은 열차를 기다리지 않고 택시를 탄다! 팀별 배포 파이프라인.	다수의 자율팀간 조율에 필요한 일종의 중앙화된 관리
팀구조	개발,QA,릴리즈, 운영이 분리된 하나의 기능 팀	공유된 서비스 모델로 팀 공동 작업 내부 소스 공개	Product Team(Prod mgr, UX, Dev, QA, dbAdmin) and Platform team(Sys Admin, Net Admin, SAN Admin), 2 Pizza team.	업무 기능별 혹은 도메인별 팀들이 모든 관점에서 책임을 수반. "네가 구축한 것은 네가 운영한다."
구조	ESB(Enterprise Service Bus)에서 실행되는 기본 SOA기반 서비스, 여전히 단일 앱이지만 모듈화	마이크로서비스를 사용하여 개발 된 새로운 기능을 갖춘 모노리스 및 마이크로서비스의 하이브리드	마이크로 서비스를 사용하여 가벼운 미들웨어 통합 레이어를 접하는 API 게이트웨이	AWS 람다와 같은 이벤트 기반의 단일 목적을 위한 전용의 서비스

https://www.linkedin.com/pulse/microservice-maturity-model-enterprise-adoption-rishi-singh/

MSA 서비스 전환 투자 우선순위 식별

* 50% 이하 : MSA 부적합, 50% ~ 70% : 자체 검토, 70% 이상 : MSA 전환 대상

평가항목	측정항목	측정내용	측정방법	작성방법	기준 데이터 (예시)
비즈니스 확장성	업무변경량	이전 변경 요청 건수 확인하여 변경 많은 업무 확인	신규 기능/기능 변경 CSR 건수	1. 업무 변경 빈도가 높다고 판단할 수 있는 CSR건수 기준 정의 2. CSR 건수와 인터뷰 결과를 종합하여 변경빈도 높은 업무 Y로 표기	100건 / 월
		업무변경 현황 및 향후 업무변경 계획 확인	현업 인터뷰, 시스템운영자 인터뷰		
	확장가능성	채널 확장, 신규 업무/기능 도입, 신기술 적용 계획 확인	현업 인터뷰	1. 확장 가능성과 관련된 인터뷰 내용을 자유 기술로 작성 2. 계획이 확인되면 Y로 표기	정성적 판단
장애대응	트랜잭션발생량	시스템, DB 부하분석을 통해 트랜잭션 발생이 집중되는 업무 확인	현행 시스템 트랜잭션 발생량 측정	1. Read와 Write로 나누어 트랜잭션 발생량 측정하여 트랜잭션 발생량이 많다고 판단할 수 있는 기준 정의 2. 1번 기준에 따라 트랜잭션양이 많은 업무에 대해 Y로 표기	월 평균 50 TPS 이상
	데이터 용량	데이터 용량이 많아 부하와 성능에 영향을 줄 수 있는지 업무 확인	현행 시스템 기준 핵심 엔터티 데이터 용량 측정	1. 성능에 영향을 줄 수 있는 데이터량 기준 정의 2. 1번 기준 이상인 업무 Y로 표기	500GB 이상
	데이터 증가량	현행 시스템에 있는 업무 중 데이터 증가량이 많아 부하와 성능에 영향을 줄 수 있는 업무 확인	현행 시스템 기준 핵심 엔터티 데이터 증가량 측정	1. 성능에 영향을 줄 수 있는 데이터 증가량 기준 정의 2. 1번 기준 이상인 업무와 인터뷰 결과를 종합하여 Y로 표기	30GB 이상 / 월
		현해 시스템에 없는 업무 중 데이터 증가량이 많은 업무 확인	현업 인터뷰		
	비즈니스 영향	장애 발생 시 매출, 신뢰도 등 비즈니스적으로 손실이나 타격을 줄 수 있는 업무 확인	현업 인터뷰	1. 인터뷰 결과를 종합하여 비즈니스적으로 손실이나 타격을 주는 업무 Y로 표기	정성적 판단
	트랜잭션 피크 발생시점	트랜잭션 발생이 특정시점에 집중되는 업무 확인	현행 시스템 트랜잭션 분석	1. 부하와 성능에 영향을 줄 수 있는 트랜젝션이 발생하는 업무 확인하고 트랜잭션 발생 시점에 대해 자유 기술 2. 특정 트랜잭션이 발생하는 업무가 있으면 Y로 표기	순간부하 150TPS 이상
배포용이성	소스 라인 수 / 용량	소스 파일 라인 수 와 용량의 크기로 배포에 용이한 크기인지 확인	현행 시스템 기준 측정	1. 배포하기에 적합한 크기 기준을 정의 2. 1번 기준 이상의 크기인 업무인 경우 Y로 표기	소스 : 10,000 라인 이상 테이블 : 50개 이상
	테이블 개수	업무와 관련 있는 테이블 개수로 배포에 용이한 크기인지 확인	현행 시스템 기준 측정		
요구사항 요청 조직 독립성	요구사항 요청 조직 수	요구사항을 요청할 예상 조직 수가 2개 이상인지 확인	현업 인터뷰	2개 이상 조직인 경우 Y로 표기	2개 이상
운영 조직 독립성	운영 조직 수	시스템을 운영할 예상 조직 수가 1개 이상인지 확인	시스템 운영자 인터뷰	2개 이상 조직인 경우 Y로 표기	2개 이상

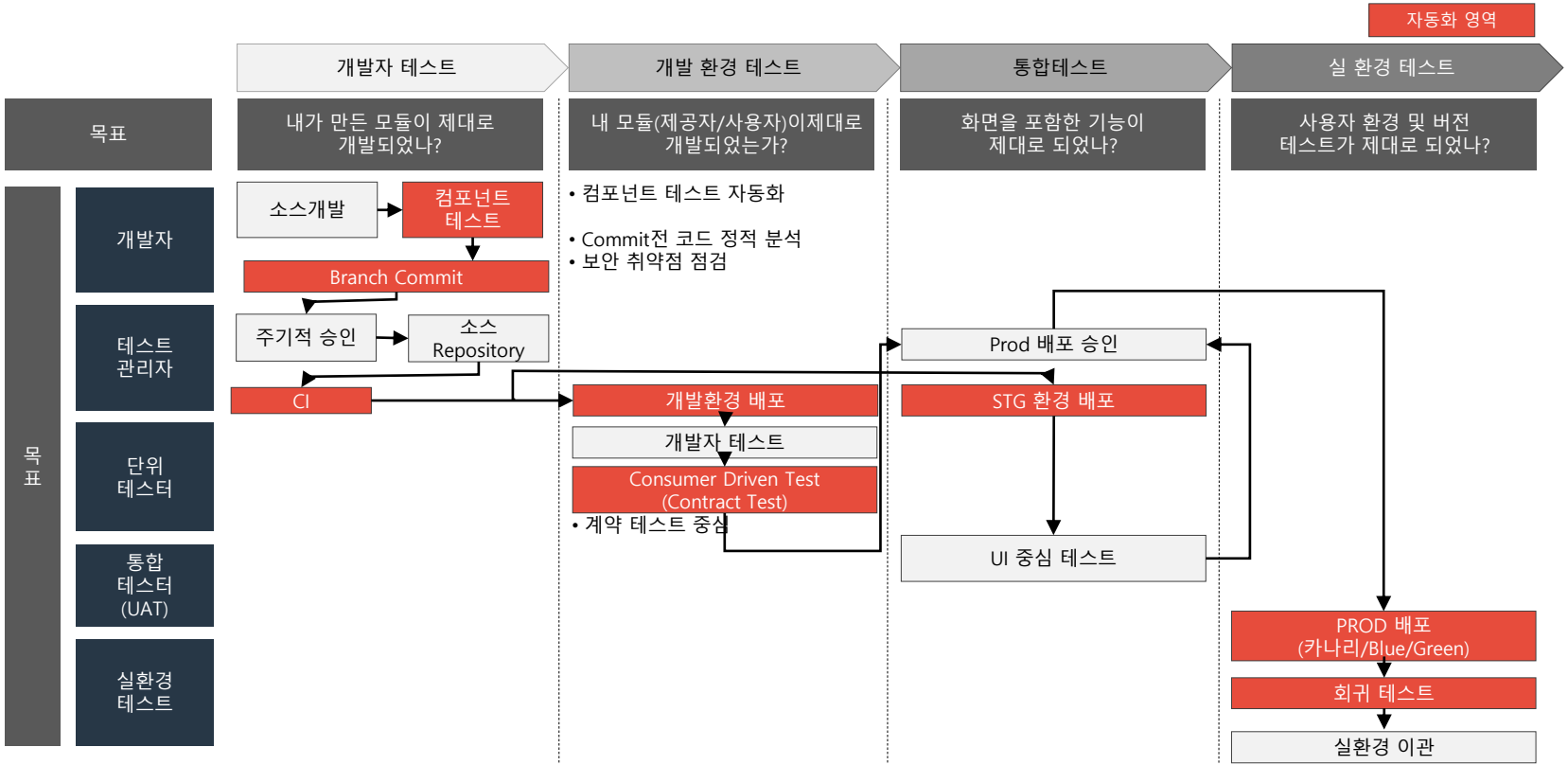
MSA 적용 전술 수립

아키텍처 패턴	체크 포인트	회피 패턴
Circuit Breaker	• 성능저하를 막아주나, Fail-fast 전략으로 사용자 경험이 나빠질 수 있음 • 적용대상이 비동기, 이벤트 기반으로 처리가능하다면 그 기반으로 전환	Saga
Database per service	• ACID 트랜잭션 비용을 포기 • ACID 트랜잭션 비용 포기가 불가하다면, Shared database 로 처리해야 하거나 Semantic Locking 통한 Eventual Transaction 상의 Lock 을 구현해야 함	Schema Per Service (Anti)
Service Registry	• 서비스 레지스트리의 유형 선택: API 기반(Eureka), DNS 기반(Kube-dns)	Saga, CQRS
Saga	• 하나이상의 마이크로서비스를 걸친 트랜잭션이 필요한 경우 & Database per service 패턴을 적용했을때 유효함 • 마이크로 서비스간 프로세스 실행시간이 상대적으로 길거나 예측하기 힘든 경우 (e.g. 결재), 비용이 높은 경우 (2PC 를 사용하기 힘든 상황)	Circuit Breaker
CQRS	• 하나 이상의 마이크로서비스에서 추출한 데이터로 뷰를 구성해야 하는경우 • 잦고 빠른 마이크로서비스 내에서의 Read 가 발생하는 경우에 사용	HATEOAS
Event Sourcing	• 이벤트 소싱은 비용이 높기 때문에 다음의 요구사항이 존재하는지 확인 필요: Undo 기능 등의 요구가 향후 생길 수 있는가?, 마이크로 서비스간의 폴리글랏 퍼시스턴스 요구?, 기능의 추가 잦음 • 이벤트 소싱에서의 이벤트는 Append only 이기 때문에 데이터의 Diff 의 정보를 충실히 포함해야 함	
Backends for frontends	• BFF 는 매번 composite service 를 구현해야 하기 때문에 관련한 frontend 의 유형이 매우 다양한 경우는 가능한 API Gateway 의 기능을 사용하거나 RESTful, HATEOAS 를 사용 권고	API Gateway
API Gateway	• API Gateway 의 유형이 다양하기 때문에 해당 기능과 역할에 따라, Service Mesh 혹은 기존 EAI (Camel library) 등에서 처리해야 하는 경우 발생할 수 있음.	BFF
Client-side UI Composition	• Server-side Rendering 은 Microservice 의 장점을 희석하므로 가능한 MVVM 기반 Client-side Rendering 을 적용해야 함	Server-side rendering (Anti)

MSA 성능 테스트 방안

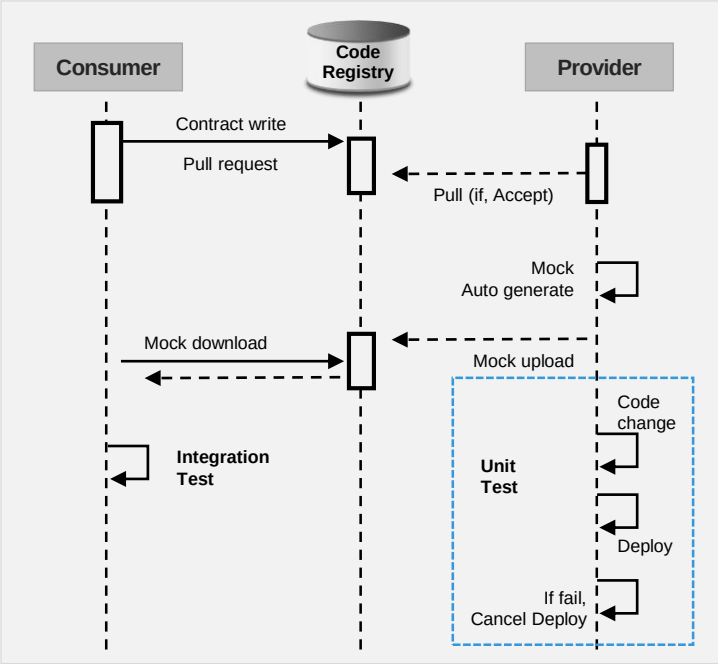
비기능 항목		품질지수 기준	비기능 성능지표 측정 방안	목표
가용성 (Availability)	가용률	가용률 측정	• 컨테이너 가용률(URL 접근성) 측정 • 측정 대상 : 컨데이너 기반의 어플리케이션 • 측정 기간 : 7 일 X 24시간	• 99.999% 이상 (Five Nines)
응답성 (Responsiveness)	응답시간	응답시간 측정	• 컨테이너 배포 시, 평균 응답시간 측정	• 평균 3초 이내
확장성 (Scalability)	확장성	리소스 확장 확인	• 컨테이너 자동/ 수동 확장 및 부하분산 처리 여부 확인	• 정상 동작
		확장요청 처리시간	• 컨테이너 확장요청 식별 시점부터 확장이 완료되기까지 소요된 시간 측정	• 평균 1분 이내
신뢰성 (Reliability)	서비스 회복시간	서비스 회복시간 측정	• 웹서버 장애(삭제) 발생 시, 서비스 회복 시간 측정	• 평균 1분 이내
			• DB 백업 파일 복구를 통한 평균 서비스 회복시간 측정	• 평균 10분 이내
	백업 준수율	백업 및 복구 기능	• 백업 및 복구 기능의 정상 동작 여부 확인	• 정상 동작

MSA 테스트 프로세스


자동화 영역
개발자 테스트
개발 환경 테스트
통합테스트
실 환경 테스트
목표
내가 만든 모듈이 제대로 개발되었나?
내 모듈(제공자/사용자)이제대로 개발되었는가?
화면을 포함한 기능이 제대로 되었나?
사용자 환경 및 버전 테스트가 제대로 되었나?
목표
개발자
테스트 관리자
단위 테스터
통합 테스터 (UAT)
실환경 테스트
소스개발
컴포넌트 테스트
Branch Commit
주기적 승인
소스 Repository
CI
• 컴포넌트 테스트 자동화
• Commit전 코드 정적 분석
• 보안 취약점 점검
개발환경 배포
개발자 테스트
Consumer Driven Test (Contract Test)
• 계약 테스트 중심
Prod 배포 승인
STG 환경 배포
UI 중심 테스트
PROD 배포 (카나리/Blue/Green)
회귀 테스트
실환경 이관

MSA 계약 테스트 방안

Consumer Driven Contracts Test 흐름도



✓ **DSL Contract 예제**

```
Contract.make {
  request {
    url "/check"
    method POST()
    body (
       age: value(regex('[2-9][0-9]')))
    headers {
      contentType(applicationJson()) }}
  response {
    status 200
    body("""
      {"status": "OK" }
"""")
}
```

```
Contract.make {
  description "should return person by id=1":
  request {
    url "/person/1"
    method GET()
  response {
    status OK()
    headers {
      contentType applicationJson()
    body(
      id: 1,
      name: "foo",
    )
  }
}
```

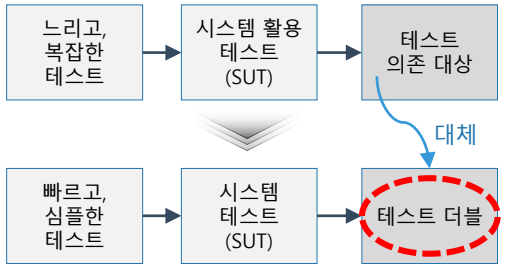
상세 설명

- 마이크로서비스 테스트는 Contract-based Test가 특징
- API Consumer가 먼저 테스트 작성
- API Provider가 Pull Request 수신 후 테스트 수행 및 Consumer Mock 객체가 자동 생성
- API Consumer는 생성된 Mock 객체를 통해 Stub 서버를 자동 생성하여 테스트 케이스 수행
  (Spring의 경우, 내부적으로 Wire mock 서버가 실행되어 Contract API 노출)
- **이를 통해 Provider의 일방적인 버전업에 따른 하위 호환성 위배를 원천적 방지**

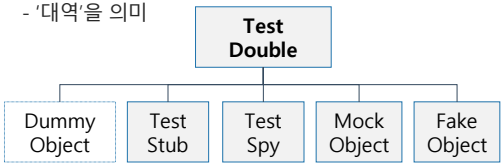
MSA 단위 기능 테스트 방안

단위(Unit) 테스트

- 소스코드의 특정 모듈이 의도된 대로 작동하는지에 대한 검사로 MSA에선 테스트 더블(Mock 객체)을 활용하여 테스트



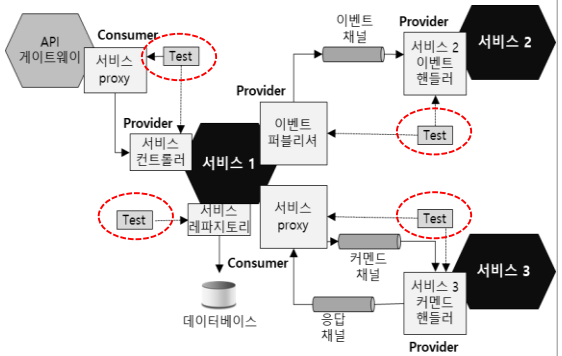
- 테스트 더블이란?
  - '대역'을 의미



출처 : http://xunitpatterns.com/TestDouble.html

서비스(통합) 테스트

- 배포 완료된 마이크로서비스에 대해 수행
- 단위 테스트의 상위 레이어에 위치하며, 구체적인 로직보다는 실제 네트워크 호출이나 데이터베이스 호출을 포함



- 서비스 테스트 전략의 일환으로, 모든 서비스를 테스트 하는 대신, 서비스간 통신을 담당하는 각 어댑터를 테스트 하는 방안도 고려

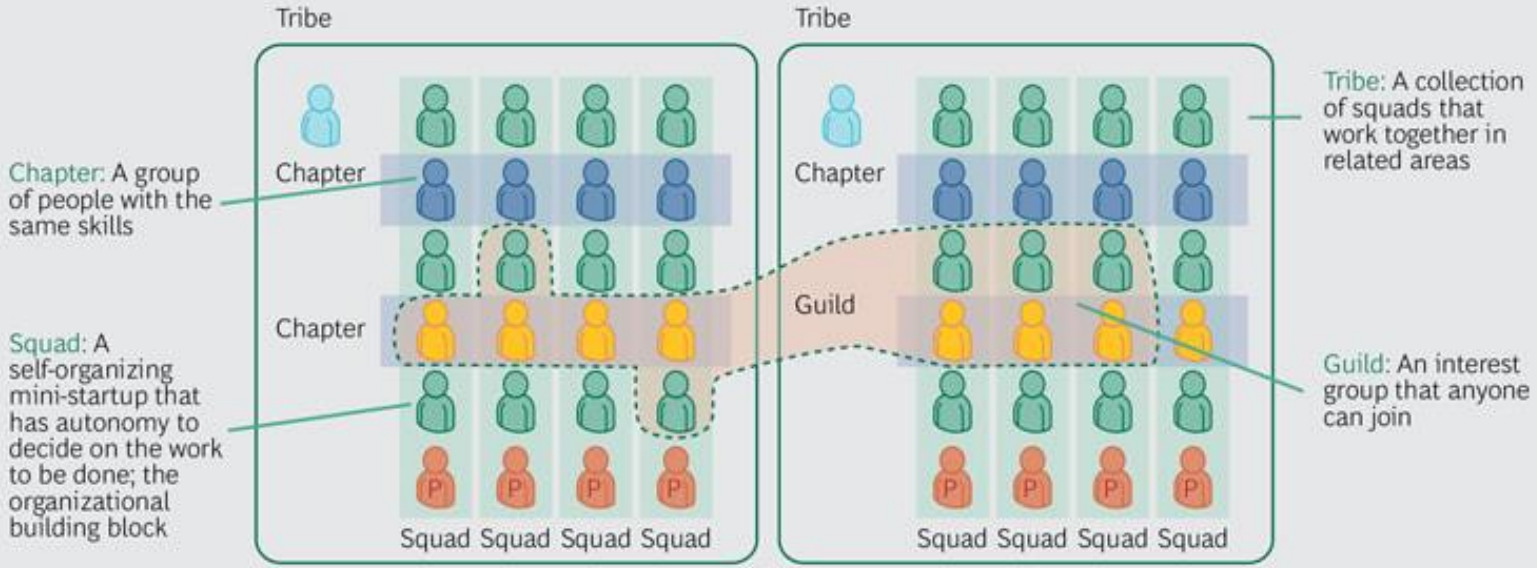
UI(E2E) 테스트

- 시스템의 모든 컴포넌트가 예정된 방식으로 잘 동작하는지를 확인하는 테스트
- 모든 서비스에 존재하는 엔드 포인트에 대한 부하 및 성능 테스트도 고려

- 거의 수행하진 않으나, BDD(Behaviour-Driven Development)를 기반으로 레코딩된 테스트를 수행하는 방안도 고려

- BDD(Behaviour-Driven Development) 란?
  - TDD와 유사하나, TDD는 테스트 자체에 집중하여 개발하는 반면, BDD는 비즈니스 요구사항에 집중하여 테스트 케이스를 개발

조직 전환 방안 – TO-BE



Source: Spotify

사례1 – ING BANK

This way of working is based on 8 important principles…

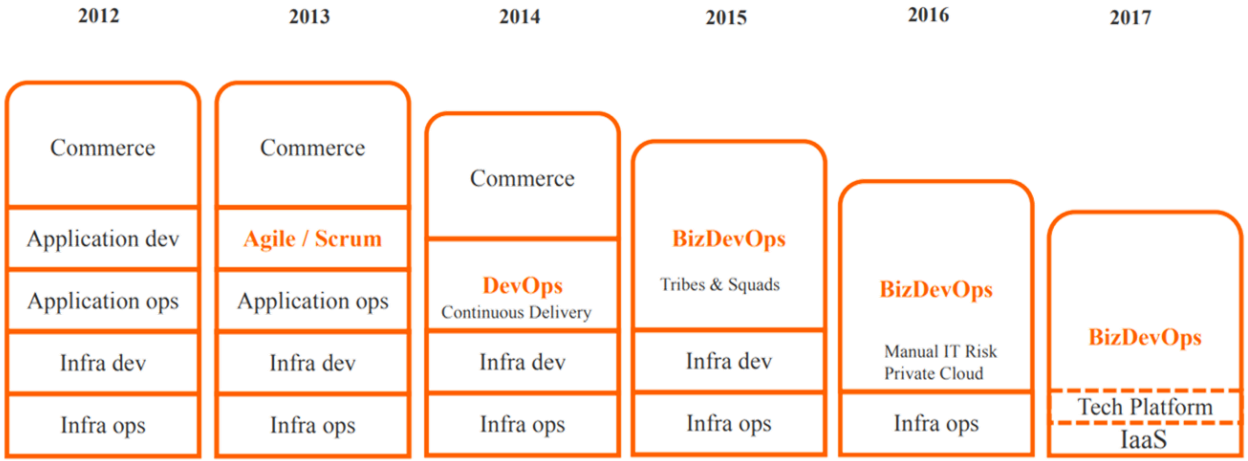
Principles

Inspired by the Agile methodology, eight principles are at the heart of our Way of Working:

1. We work in high **performing** teams
2. We **empower** teams
3. We care about talent and **craftmanship**
4. We continously **learn** from customers and apply learning to improve
5. We set **priorities** with the big picture in mind
6. We are consistent in our **organisational** design and way of working
7. We organise for **simplicity**
8. We **re-use** instead of reinvent

사례1 – ING BANK

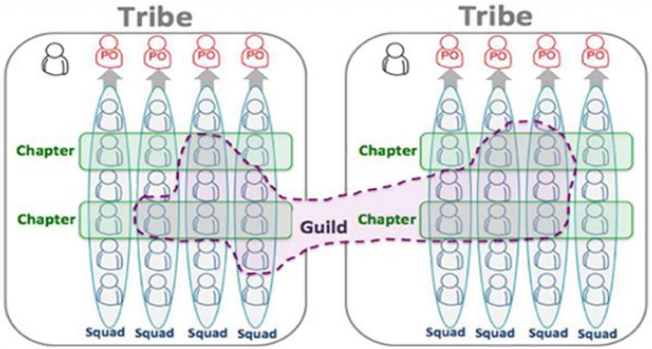
In the past five years, ING has been reorganizing for speed and skill. Roles and responsibilities have shifted radically



Engineer: From single discipline to **full stack engineers**: designing, coding, test engineering, infra engineering, etc
Product Owner: From writing PIDs to product vision and backlog to **end to end bizdevops responsibility**
IT Manager: from delivery manager to perhaps the most differntiating role: **skill and competency coach**.

사례1 – ING BANK

And this is how we organize ourselves

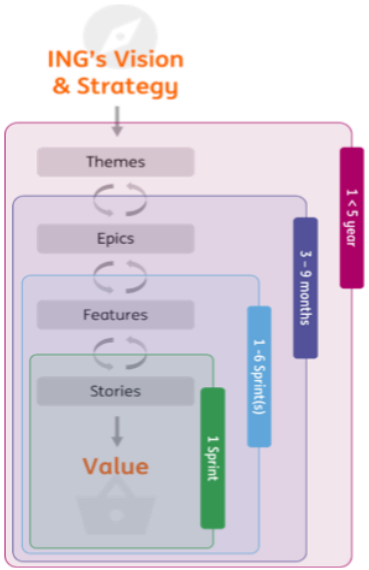


- **Squads**, are high performing "stable" teams who provide the execution power to the One WoW
- A **Tribe** is a collection of Squads organized around the same purpose
- **Chapters** are formal "groups" of members with the same background / expertise deciding on how things need to be done within a Tribe, regarding their area of expertise
- A **Guild** is a group of experts or community, which can be set up across Tribes (and even across countries) typically based on a shared interest in a technology or product / service

Squad over I, Tribe over Squad, Company over Tribe, Customer over Company

사례1 – ING BANK

These are the fundamentals of One Way of Working Agile

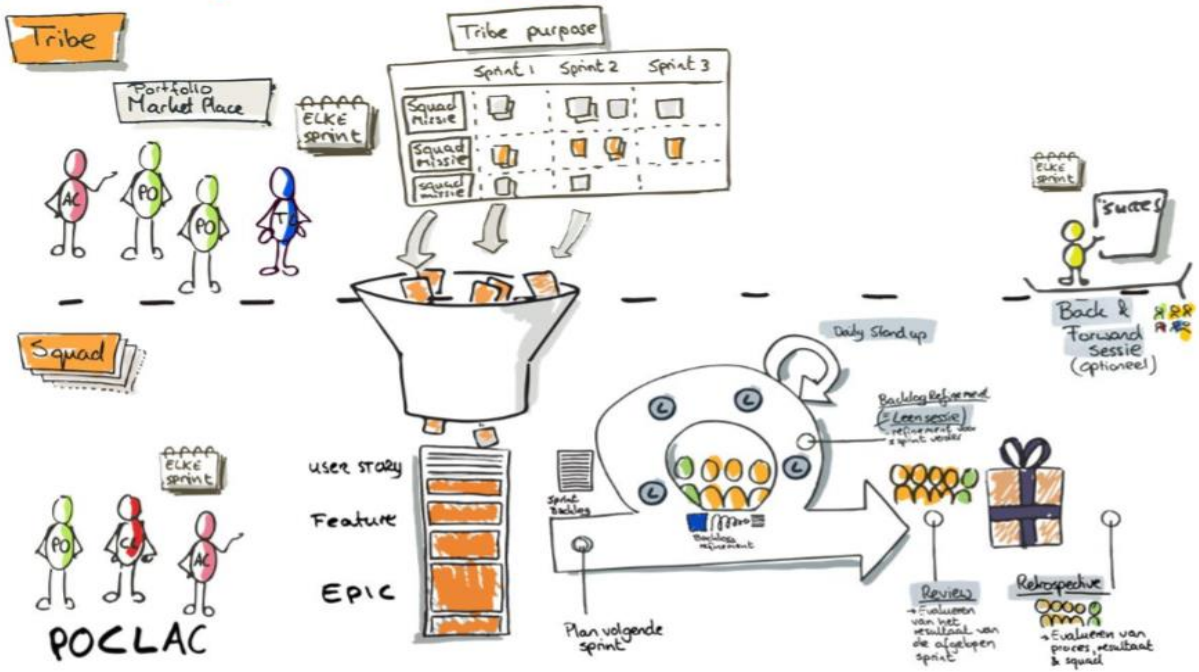


- A **Theme** connects the dots between Strategy and execution. Hence, a Theme represents an (investment) area that supports the strategic vision. Themes are representing one to five year(s).
- A Theme is broken down in multiple **Epics**: large scale bank initiatives that will be delivered in one to three quarters. At the Epic level, a.o. the benefits should be defined.
- An Epic can be broken down in one or more **Features**. A Feature reflects **part of the value** (both functional and non-functional) for a stakeholder that could be delivered within multiple (typically one to six) Sprints.
- Features have to be refined to **Story** level. Stories are explicitly defined by teams themselves. They have to be small enough to be picked up by one team and delivered within one Sprint.

A structured way to manage the demand and focus on the minimum viable products

사례1 – ING BANK

Our way of working in the new organisation



Recommended Readings

1. **Overall MSA Design patterns:**
   https://www.manning.com/books/microservices-patterns

2. **Microservice decomposition strategy:**
   - DDD distilled: https://www.oreilly.com/library/view/domain-driven-design-distilled/9780134434964/
   - Event Storming: https://leanpub.com/introducing_eventstorming

3. **Database Design in MSA:**
   - Lightly:
     https://www.confluent.io/wp-content/uploads/2016/08/Making_Sense_of_Stream_Processing_Confluent_1.pdf
   - Deep dive:
     https://dataintensive.net/?fbclid=IwAR3OSWkhqRjLI9gBoMpbsk-QGxeLpTYVXIJVCSaw_A5eYrBDc0piKSm4pMM

1. **API design and REST:**
   http://pepa.holla.cz/wp-cont…/…/2016/01/REST-in-Practice.pdf

THANKS!

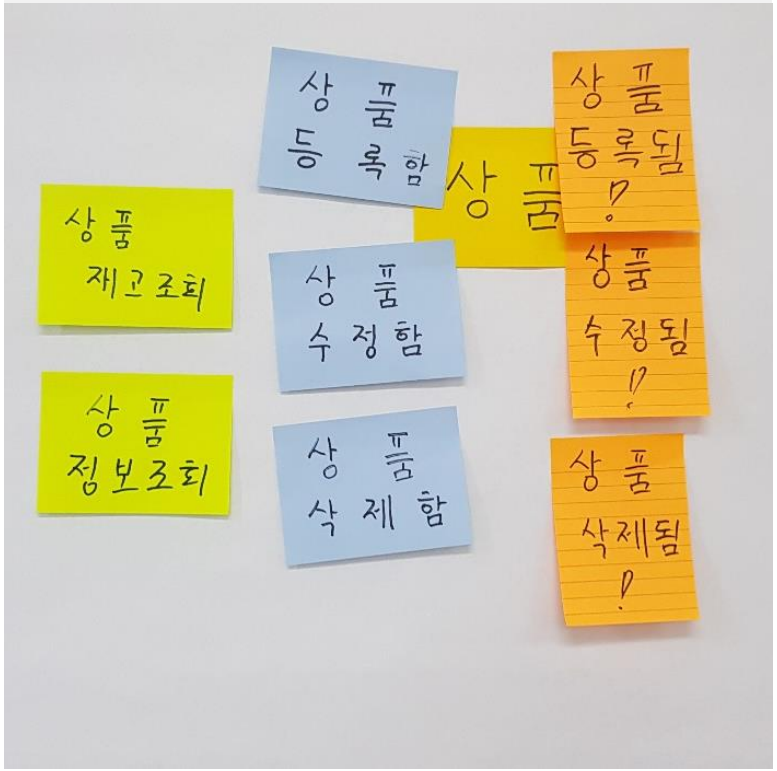
Any Question?

You can find me at:

jyjang@uengine.org
https://github.com/jinyoung
https://github.com/TheOpenCloudEngine

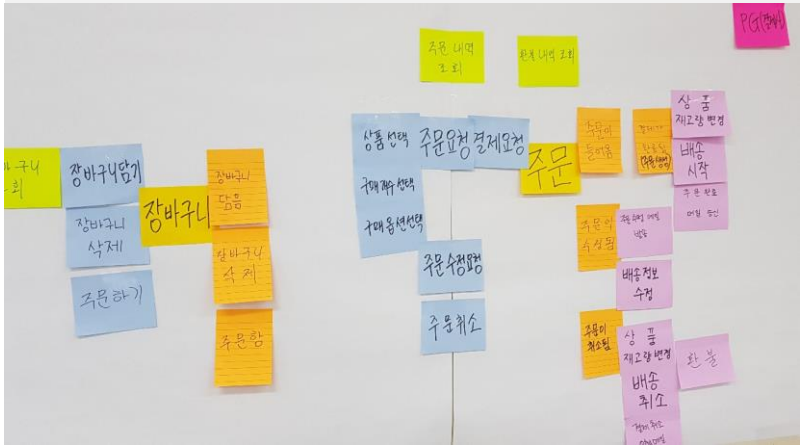
Appendix:
실제 이벤트 스토밍 워크숍 사례

Before

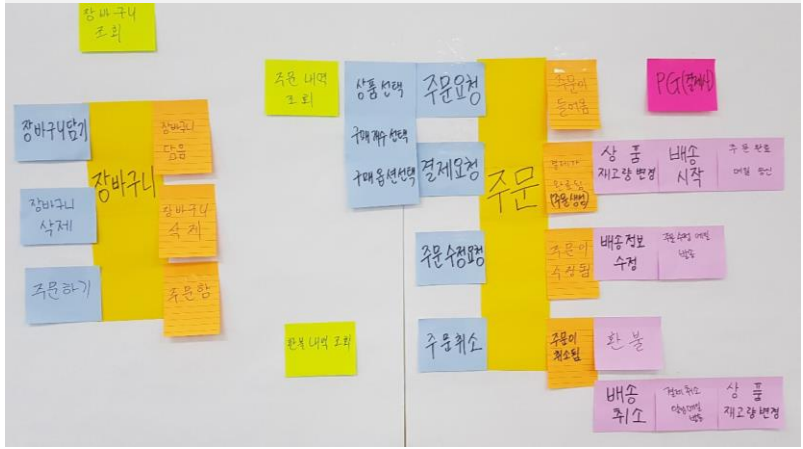


After

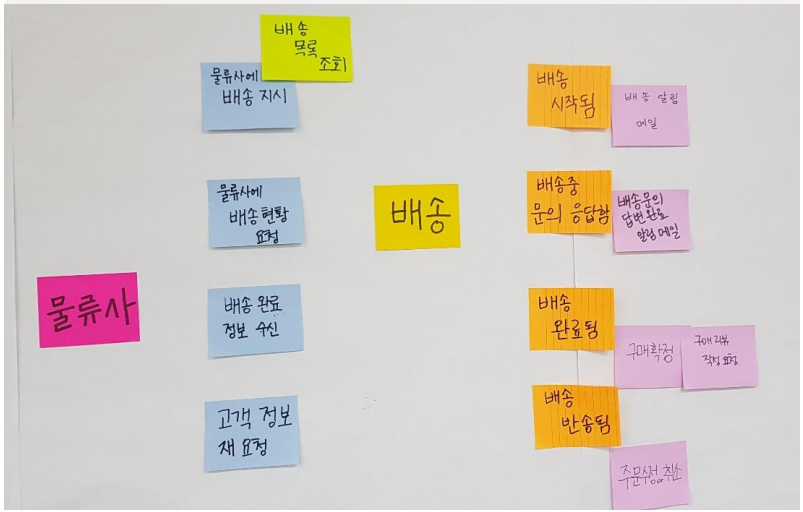
Before



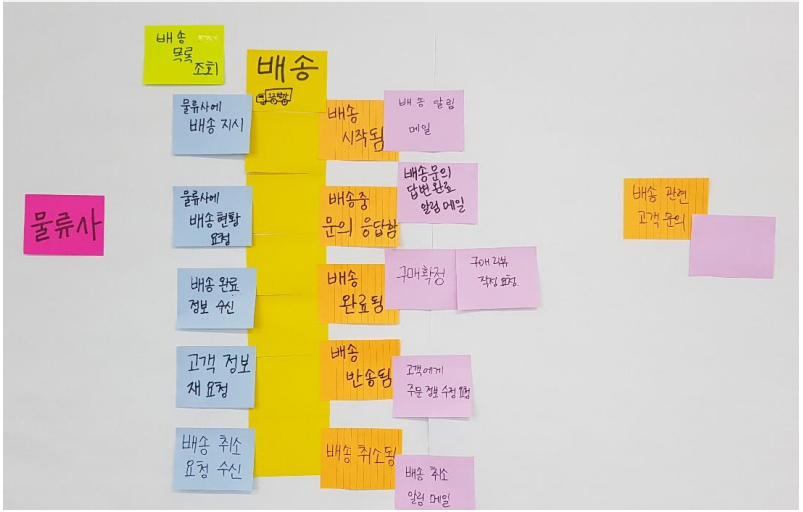
After



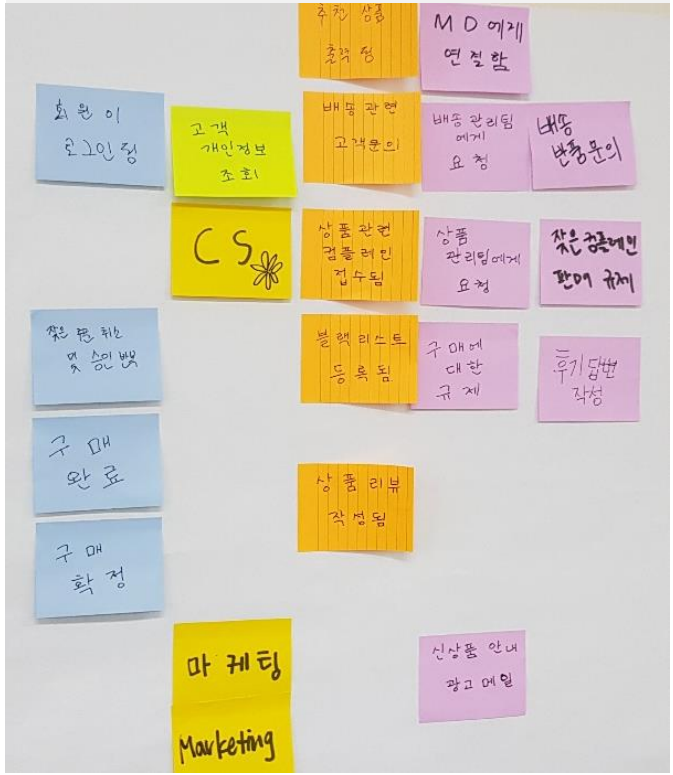
Before



After

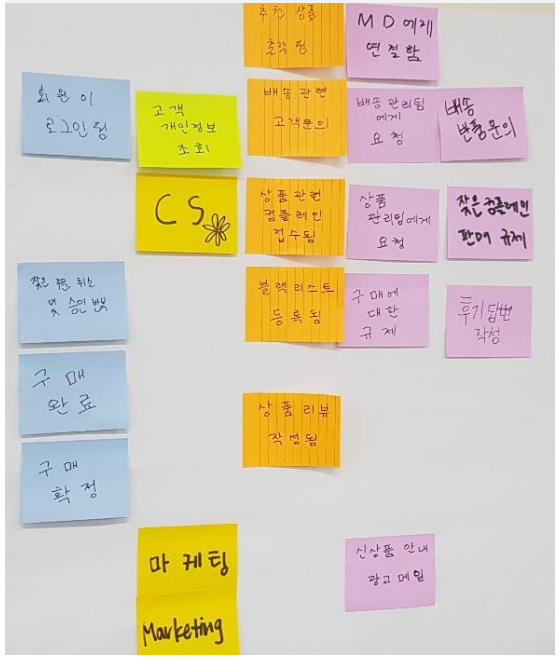


Before

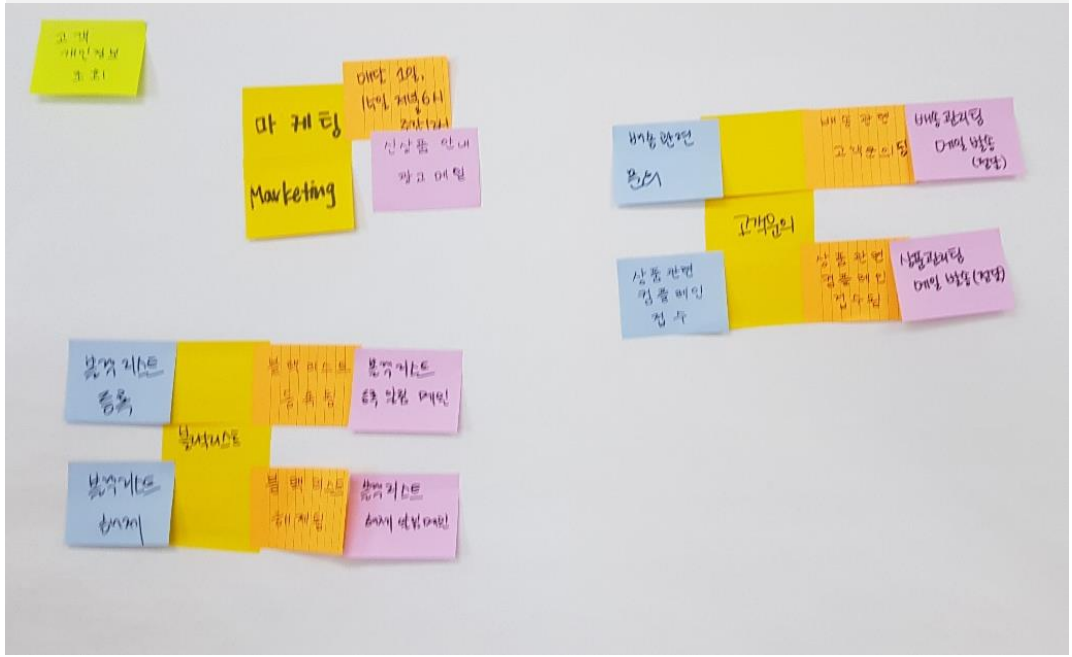


After

Before



After



Appendix:
Micro Service Coding Style

1년차 – Fun & Joy

```
1  class HelloWorld
2  {
3      public static void main(String args[])
4      {
5          // Displays "Hello World!" on the console.
6          System.out.println("Hello World!");
7      }
8  }
```

2년차 - Reusability

```
/**
 * Hello world class
 *
 * Used to display the phrase "Hello World!" in a console.
 *
 * @author Sean
 */
class HelloWorld
{
    /**
     * The phrase to display in the console
     */
    public static final string PHRASE = "Hello World!";

    /**
     * Main method
     *
     * @param args Command line arguments
     * @return void
     */
    public static void main(String args[])
    {
        // Display our phrase in a console.
        System.out.println(PHRASE);
    }
}
```

3년차 - Modularity

```
/**
 * Hello world class
 *
 * Used to display the phrase "Hello World!" in a console.
 *
 * @author Sean
 * @license LGPL
 * @version 1.2
 * @see System.out.println
 * @see README
 * @todo Create factory methods
 * @link https://github.com/sean/helloworld
 */
class HelloWorld
{
    /**
     * The default phrase to display in the console
     */
    public static final string PHRASE = "Hello World!";

    /**
     * The phrase to display in the console
     */
    private string hello_world = null;

    /**
     * Constructor
     *
     * @param hw The phrase to display in the console
     */
    public HelloWorld(string hw)
    {
        hello_world = hw;
    }

    /**
     * Display the phrase "Hello World!" in a console
     *
     * @return void
     */
    public void sayPhrase()
    {
        // Display our phrase in a console.
        System.out.println(hello_world);
    }

    /**
     * Main method
     *
     * @param args Command line arguments
     * @return void
     */
    public static void main(String args[])
    {
        HelloWorld hw = new HelloWorld(PHRASE);
        try {
            hw.sayPhrase();
        } catch (Exception e) {
            // Do nothing!
        }
    }
}
```

5년차 – Portability and Standardization

```
/**
 * Enterprise Hello World class v2.2
 *
 * Provides an enterpirse ready, scalable buisness solution
 * for displaying the phrase "Hello World!" in a console.
 *
 * IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED
 * TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER
 * PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS
 * PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING
 * ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL
 * DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE
 * LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR
 * DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
 * YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO
 * OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER
 * OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGES.
 *
 * @author Sean
 * @copyright Sean 2012
 * @license LGPL
 * @version 2.2
 * @see System.out.println
 * @see README
 * @see license.txt
 * @todo Test of OS compatibility
 * @link https://github.com/sean/helloworld
 */
class HelloWorld
{
    /**
     * The first phrase
     */
    public static final string PHRASE_HELLO = "Hello";

    /**
     * The second phrase
     */
    public static final string PHRASE_WORLD = "World";

    /**
     * The first word in our phrase
     */
    private Word hello = null;

    /**
     * The second word in our phrase
     */
    private Word world = null;

    /**
     * Constructor
     *
     * @param hello First word to display in the console
     * @param world Second word to display in the console
     * @Required
     */
    public HelloWorld(Word hello, Word world)
    {
        this.hello = hello;
        this.world = world;
    }

    /**
     * Display the phrase "Hello World!" in a console
     *
     * @return void
     */
```

```
public void sayPhrase()
{
    // Display our phrase in a console.
    string first = this.hello.toString();
    string second = this.world.toString();
    System.out.println(first + " " + second);
}

/**
 * Sets the phrase to use for hello
 *
 * @param h The first phrase
 * @return void
 */
public void setHello(Word h)
{
    this.hello.setWord(h);
}

/**
 * Gets the phrase to use for hello
 *
 * @return Word
 */
public Word getHello()
{
    return this.hello;
}

/**
 * Sets the phrase to use for world
 *
 * @param w The second phrase
 * @return void
 */
public void setWorld(string w)
{
    this.world.setWord(w);
}

/**
 * Gets the phrase to use for world
 *
 * @return Word
 */
public Word getWorld()
{
    return this.world;
}

/**
 * Main method
 *
 * @param args Command line arguments
 * @return void
 */
public static void main(String args[])
{
    // Create a new dic so we can display our phrase on the
    // command line.
    DIC d = DependencyInjectionContainer::factory();
    HelloWorld hw = null;

    // Check for errors!
    try {
        hw = d.newInstance(HelloWorld.class);
    } catch (DICInstanceException $e) {
        System.err.println("There was an error creating an instance of HelloWorld.");
        return;
    }
```

```
        // Display the phrase on the command line.
        try {
            hw.setHello(PHRASE_HELLO);
            hw.setWorld(PHRASE_WORLD);
            hw.sayPhrase();
        } catch (IOException e) {
            System.err.println("There was an IO error.");
        } catch (ConsoleException e) {
            System.err.println("There was a console error.");
        } catch (Exception e) {
            System.err.println("There was an unknown error.");
        }
    }
}
```

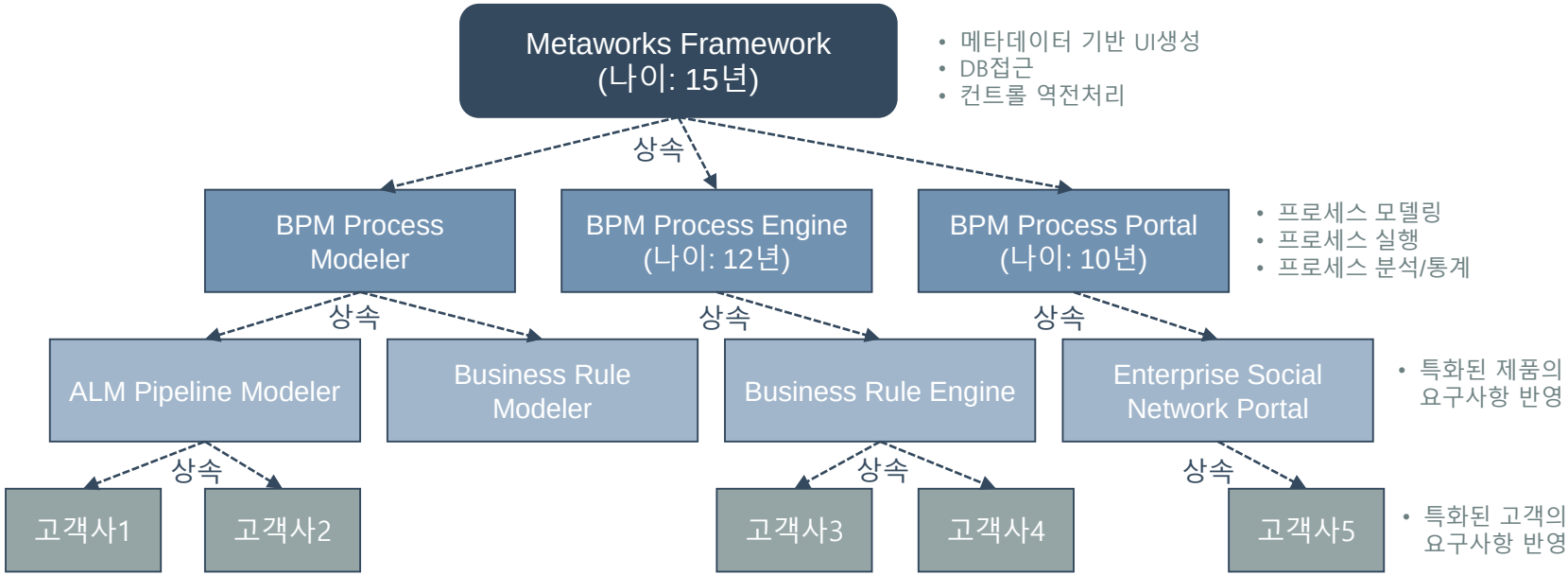
```
<beans xmlns="http://www.framework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.framework.org/schema/context"
    xsi:schemaLocation="http://www.framework.org/schema/beans

http://www.framework.org/schema/beans/beans-2.5.xsd


http://www.framework.org/schema/context

http://www.framework.org/schema/context/context-2.5.xsd">

    <context:annotation-config />

    <bean id="HelloWorldBean" class="com.my-pragmatic-journey.beans">
        <property name="hello" value="Word" />
        <property name="world" value="Word" />
    </bean>
```

유엔진솔루션즈

- 2013년 부터 기업용 패키지 솔루션 개발
- BPM, SNS, ALM 등 BPM을 기반한 다양한 파생상품 개발
- (국내) 고객입장의 밸류: 커스터마이징이 잘되는 제품

- 수익모델: 온-프레미스 기반 프로덕트 제공

소프트웨어 구조

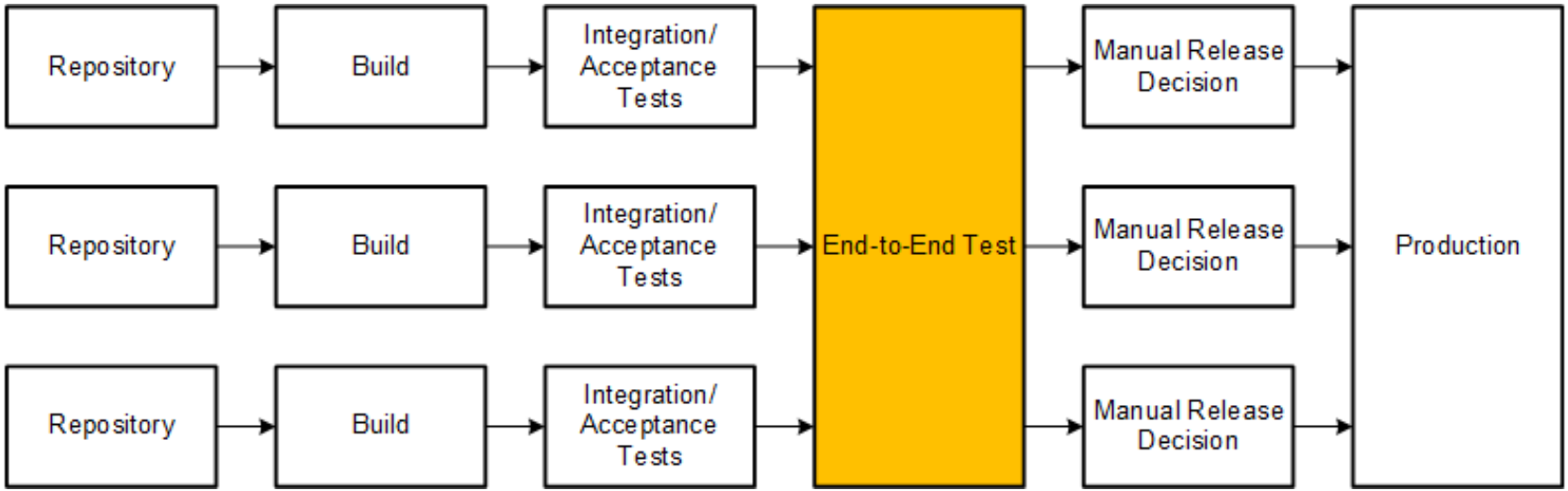


Key Strategies:
- DRY (Don't Repeat Yourself) Rule : 가능한 공통구현 코드는 중복이 없어야
- White-box Inheritance : 한번 구현된 것은 추상화하여 상위 클래스에 정의

10년의 경과보고

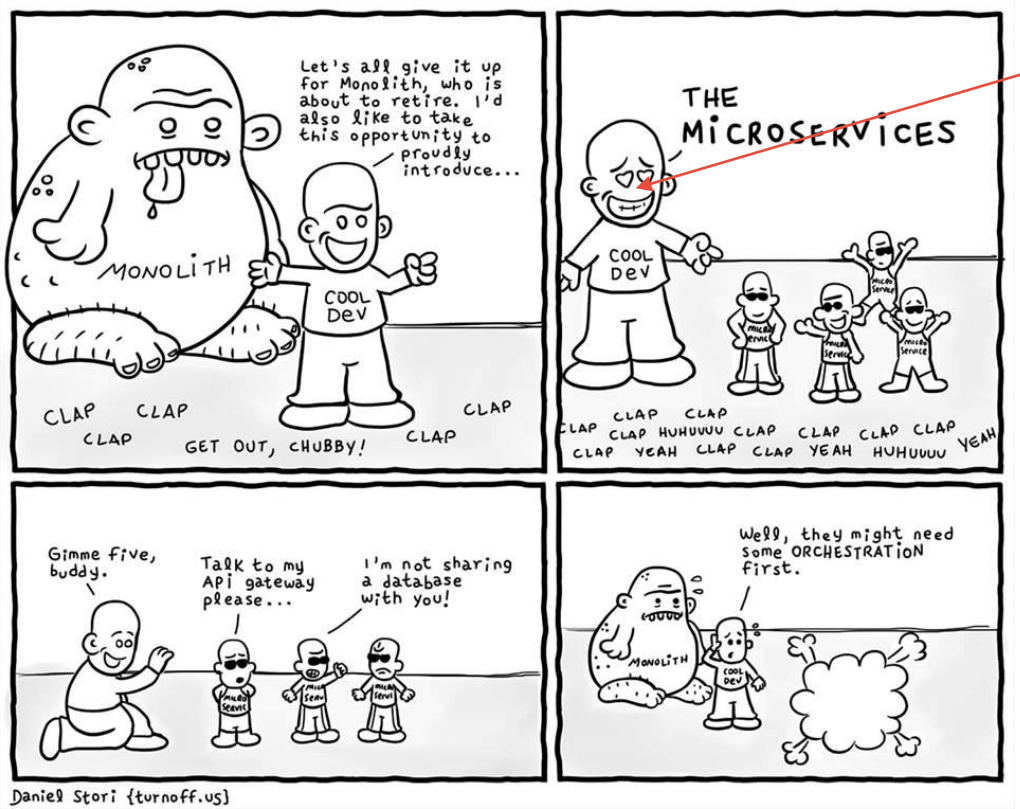
- 개발자 수급:
  - 자체 프레임워크와 개발표준에 익숙해지는데 최소 3년 소요
  - 객체 지향과 AOP 개념, 빌드 등 기반 기술 이해에 그중 1.5 년 이상 소요
- 빌드/통합:
  - 1회 Integration, End-to-End Test 를 위한 Maven Dependency 버전 일치에 적어도 4시간(반나절) 이상 소요
  - Maven Dependency Hell
- 기술적용:
  - 많은 오픈소스들의 기능들을 통합함에 있어 상이한 라이브러리 디펜던시 충돌로 인한 통합의 어려움
    - OSGi 시도
      - OSGi 의 복잡한 버전관리, 기반 기술 러닝커브로 포기

통합 빌드 주기 : 1개월 → 망한다



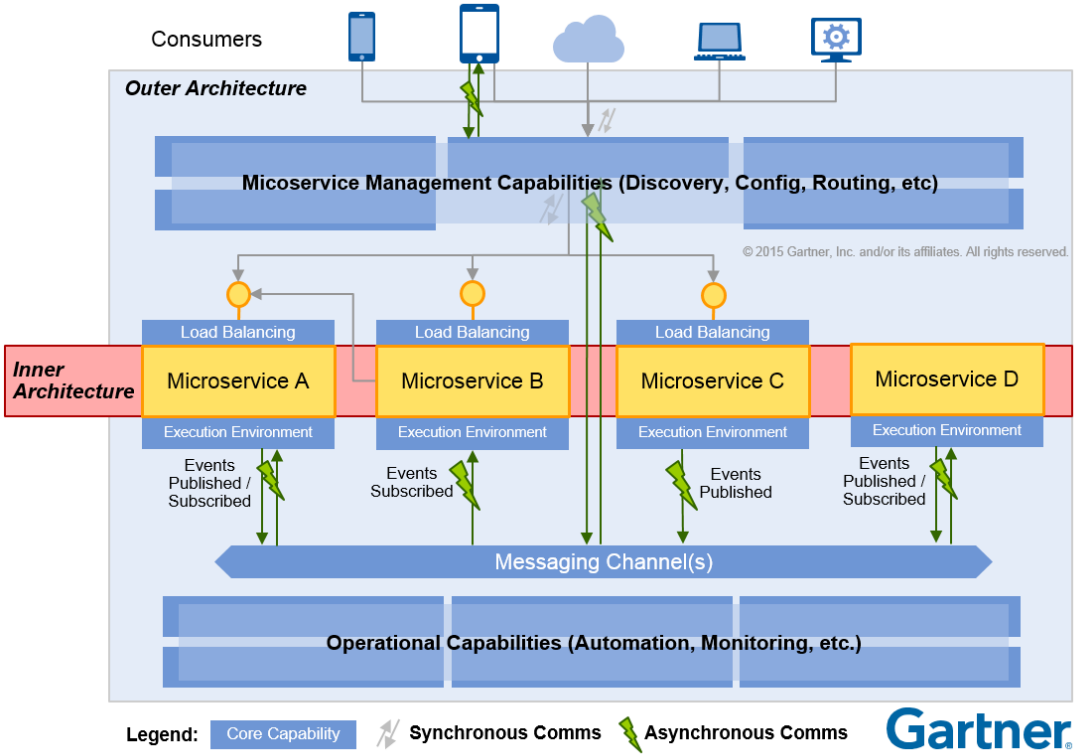
➔ 결론: 소프트웨어 품질 저하

기회 : MSA 의 발견


Let's all give it up for Monolith, who is about to retire. I'd also like to take this opportunity to proudly introduce...
MONOLITH
COOL Dev
CLAP CLAP CLAP CLAP
GET OUT, CHUBBY!
CLAP
THE MICROSERVICES
COOL Dev
CLAP CLAP CLAP
CLAP CLAP HUHUUUU CLAP CLAP CLAP CLAP
CLAP YEAH CLAP CLAP YEAH HUHUUUU YEAH
Gimme five, buddy.
Talk to my API gateway please...
I'm not sharing a database with you!
Well, they might need some ORCHESTRATION first.
MONOLITH
COOL Dev
Daniel Stori {turnoff.us}


장진영
2017년 7월 31일 ·
아이고 maven dependency hell... BPM, SNS, Essencia.. 전부 MSA 로 전환
Ji Gwan Ryu님, 이상부님 외 3명
댓글 2개
좋아요
댓글 달기
공유하기
최성훈 왜서요? ㅜㅜ
좋아요 · 답글 달기 · 1년
장진영 기존 BPM, SNS 모두 MSA 로 전환합니다.
좋아요 · 답글 달기 · 1년
댓글 달기
김윤수 오~ 대단하십니다
좋아요 · 답글 달기 · 1년
장진영 얼마나 걸릴지 모르지만 생각보다 빨리 끝날수도 있을듯합니다.
좋아요 · 답글 달기 · 1년
댓글 달기
댓글을 입력하세요...

MSA architecture 의 두가지 관점



Outer:

개발자가 공히 겪게 되는 복잡성들 (e.g. 멀티스레딩, 탄력성, 장애내성, 공통처리) 을 네트워크단에서 처리 해주고

Inner:

코드내부는 가능한 무식, 간결, 명확하게 단순화

MSA Design Principles

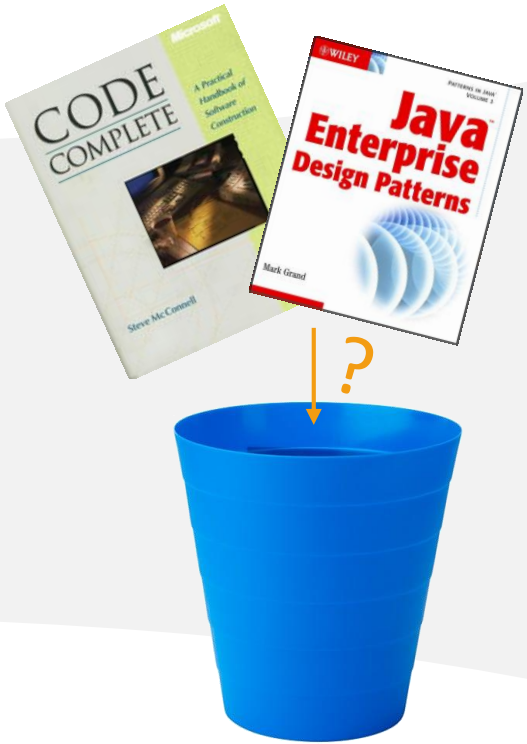
- SoC—Separation of concerns.
  : 고생해서 분리를 왜 했느냐? 관심사를 분리해서 개발할 수 있어야 한다
- DDD—Domain Driven Design.
  : 도메인을 알면, 관심사를 어떻게 분리할지 작전이 나온다.
- KISS — Keep it simple, stupid.
  : 작은 사이즈의 매력이 뭐냐? 무식하게 두라. 때론 하드코드도 괜찮아. 스프링부트를 보라, Bean 설정을 메서드로-하드코드로 한다
- YAGNI—You aren’t gonna need it.
  : 머리를 어지럽히는 복잡한 기술? 지금 당장 필요한 것만 써라. 너무 미래를 보지마라. 보통 복잡한 설계와 기술은 하나의 언어와 표준을 준수하기 위해 등장한다. 당장 돌아가는 코드와 언어가 있다면 그걸 사용하자 Polyglot 이 좋은게 그거다. 필요할때 복잡해져라. 애자일이 그거다.

Write Simple

- 단위 마이크로 서비스는 의도적으로 크기를 작게 한다
- 이미 모듈화 되었으므로, 내부는 읽기 좋게 (단순하게) 작성한다
- 작은 크기내에 복잡한 추상화, 일반화, 모듈화는 사족이다
- 데이터베이스 또한 단순화한다 (Denormalization, Materialized)
- 적당한 하드코드는 읽기가 생각보다 좋다 (property 화의 최소화) – 위험한가?
- 설정을 통한 분기 보다 하드코드로 된 펑션이 읽기가 좋다

받아들이기 힘들었던 이야기들

- 공통화 하지 말라?
  - DRY X → KISS O
  - Shared Kernel X → Polyglot
- 네이밍(패키지명) 멋지게 짓지말라?
  - Ubiquitous Language
  - Bounded Context
- 하드코딩 하라?
  - Inline Bean Definition
- 백엔드가 꼭 필요한가?
  - MVC X → Front + Gateway + DB



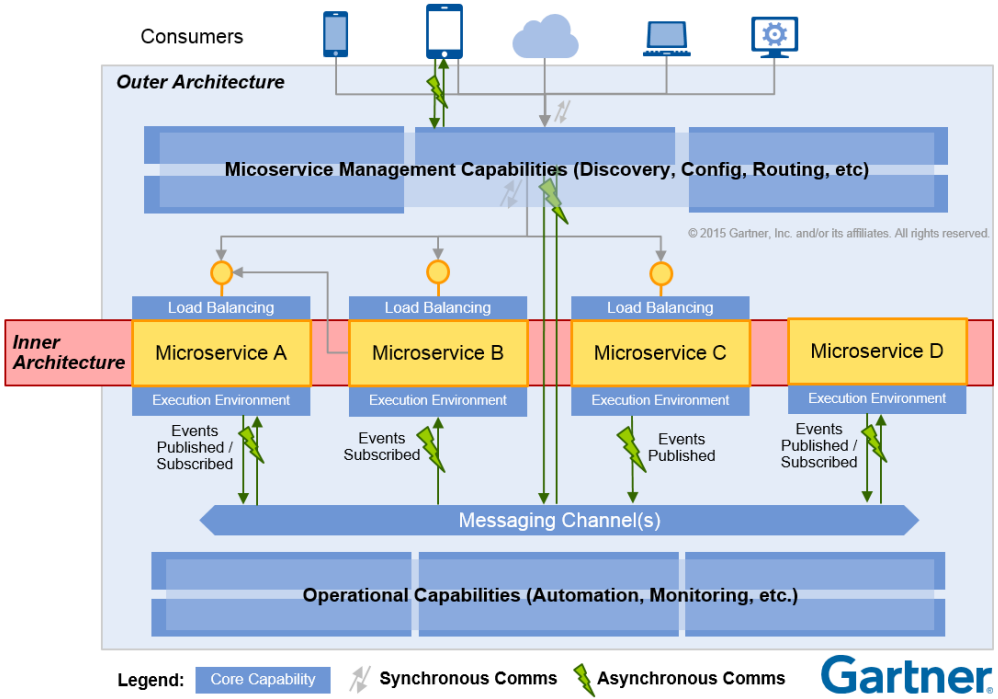
그렇다면 객체지향이 죽은 것인가?
아니면 어디로 갔는가?

네트워크단으로 갔다.

Proxy Pattern → API Gateway
Polymorphism → Traffic Routing
Multi-threading → Container Orchestrator
Aspect Cross-cutting → PubSub, Event-driven

아우터 아키텍처가 해준다.

Spring → Istio
Multi-threading → Kubernetes
Aspect Cross-cutting → Kafka

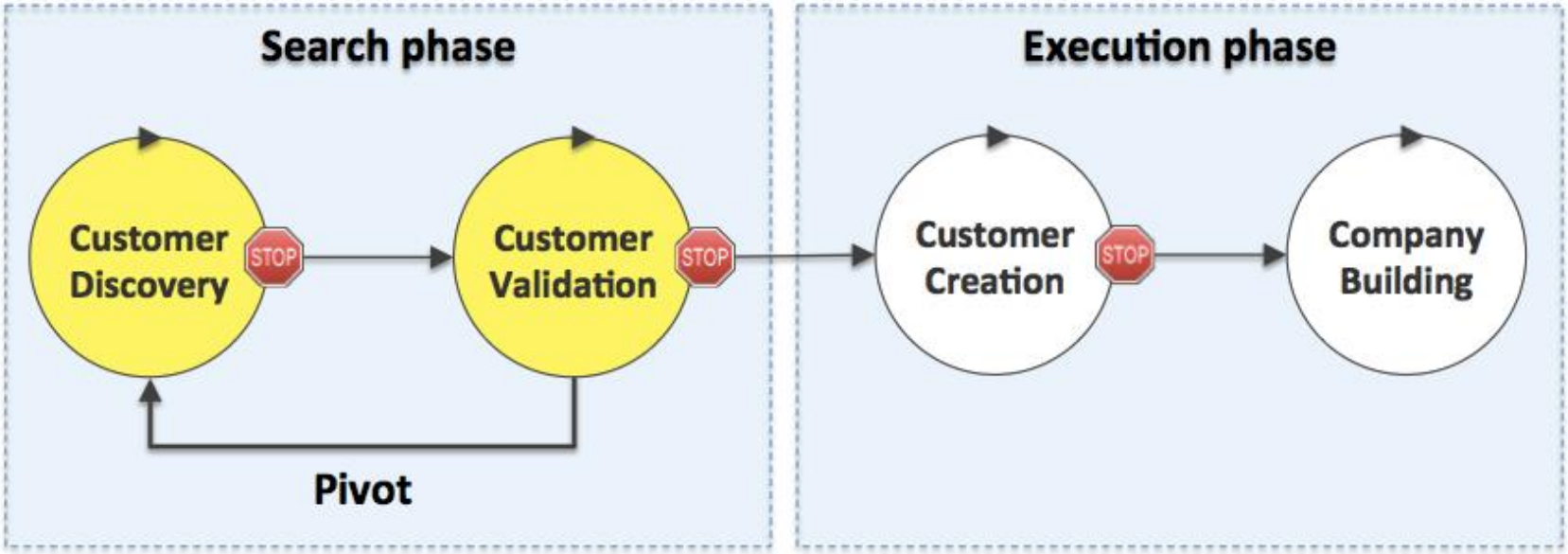


받아들이니 깨닫게 된 것들

- 내 소프트웨어는 어려운 것이 아니었다
  - 그냥 복잡하게 엉켜있는 것이었다.
  - 너무 깊은 추상화, 공통화는 오히려 내 발목을 잡았던 것이다.

- 하지만, Inner Architecture 내에서도
  - 여전히 주석, 완전한 코드, 테스팅은 ”어느정도” 중요하다.
  - 그 정도는 “필요한 만큼” 이라고 정의하고 싶다.

Agile Principle:
팔리지 않을것은 만들지도 마라



결론

- 서비스 사업 뿐만 아니라, 솔루션 개발 및 Product-Line Management 에서도 MSA 를 적용했을때 효율이 발생
- 단위개발 (Inner Architecture) 의 단순성 추구

- 결론적으로 작전의 변경:
  형상에 대한 과도한 표준화, 공통화, 기술 검증의 노력과 에너지

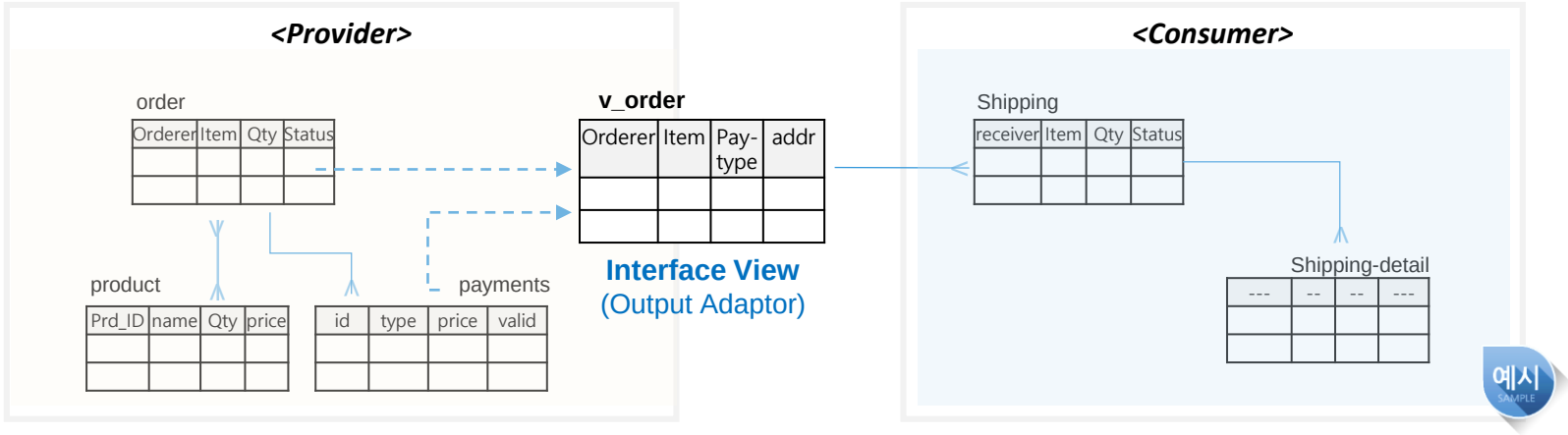
  ➔ 비즈니스 기회의 빠른 시장 검증의 노력과 에너지

10년차: 다시 즐거움을 되찾다

```
1  /**
2   * Used to display the phrase "Hello World!" in a console
3   *
4   * @author Sean
5   * @see README
6   */
7  class HelloWorld
8  {
9      public static void main(String args[])
10     {
11         System.out.println("Hello World!");
12     }
13 }
```

Appendix:
Mini-Service approach

Interface View를 통한 Data 참조



- Scheme per SVC 환경에서 Interface용 View를 데이터 참조를 원하는 서비스 측에 제공
- Core Domain, 혹은 Provider는 내부 구현(DB Scheme)을 마음대로 리팩토링 가능 (Implementation Hiding)
- Materialized View를 사용하여 View 제공 시, 잦은 참조에도 내부 성능저하 없음

Interface 연계방식 별 비교

연계방법	Implementation hiding	Protocol	Time-Decoupling	Data Projection	Good performance	구축 비용
API 호출	O	REST/SOAP	X	REST기반 API를 여러 번 호출	X	Middle
Event 기반	O	Pub/Sub	O	Event Queue로부터 Subscribe하여 Replica 구축 (CQRS)	O	High
Materialized View	O	Internal DB Operation	X	JOIN SQL	O	Low
View	O	Internal DB Operation	X	JOIN SQL	X	Low