Microservice, Docker & Kubernetes

조대협, Google Cloud Sales Engineer
terrycho@google.com

Google Cloud Platform

발표자 소개

조 대 현 본명: 조병욱

- 회원 13만명 온라인 개발자 커뮤니티 자바스터디(www.javastudy.co.kr) 운영자.. (기억의 천편..)
- 전 .. 자바 개발자 협회 부회장,서버사이드 아키텍트 그룹 운영자
- 엔젤
- BEA 웹로직 기술 지원 개발자
- 엔지니어, 성능 튜닝
- NHN
- 프리랜서 컨설턴트 (SOA,EAI,ALM,Enterprise 2.0,대용량 분산 시스템)
- MSAPAC 클라우드 수석 아키텍트 (잘나가는 삼성전자 무선 사업부 B2B팀 Chief Architect
- 스타트업 CTO
- 구글 클라우드 엔지니어

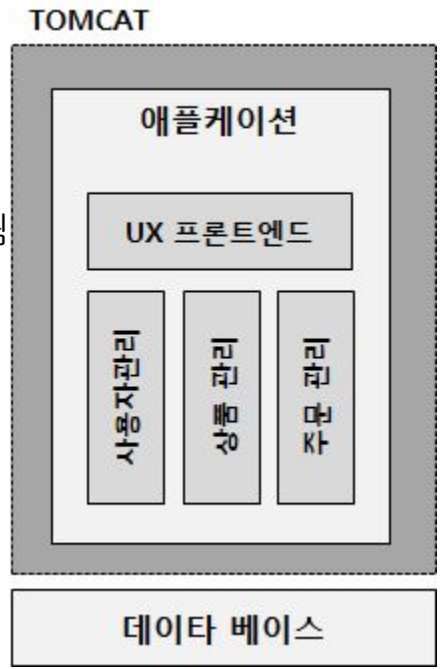
목차

- Micro Service Architecture
- Docker
- Kubernetes
- DEMO

Micro Service Architecture 의 기본 개념

전통적인 아키텍쳐 스타일

- 모노리틱 아키텍쳐 (통서버)
  - 하나의 서버에 모든 비지니스 로직이 들어가 있는 형태
  - 하나의 중앙 집중화된 데이타 베이스에 모든 데이타가 저장됨



전통적인 아키텍쳐 스타일의 장단점

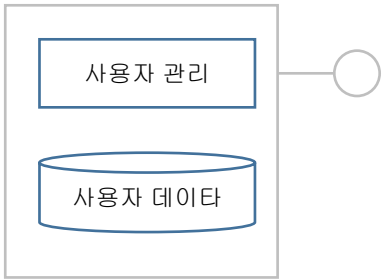
- 단점
  - 여러개의 기술을 혼용해서 사용하기 어려움 (node.js , Ruby, Python etc)
  - 배포 및 재기동 시간이 오래 걸림
  - 수정이 용이하지 않음 (타 컴포넌트 의존성)
- 장점
  - 기술 단일화
  - 관리 용이성

마이크로서비스 아키텍쳐란?

- 시스템을 여러개의 독립된 서비스로 나눠서, 이 서비스를 조합함으로서 기능을 제공하는 아키텍쳐 디자인 패턴
- SOA의 경량화 버전 (실패한다면 실패하는 이유도 같음)

서비스란?

- 단일된 기능 묶음으로 개발된 서비스 컴포넌트
- REST API등을 통하여 기능을 제공
- 데이타를 공유하지 않고 독립적으로 가공 저장



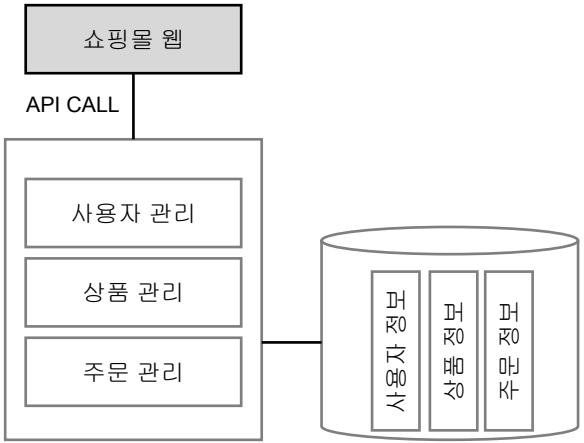
인터페이스

- REST
- Thrift,Protocol buffer
- AMQP
- :

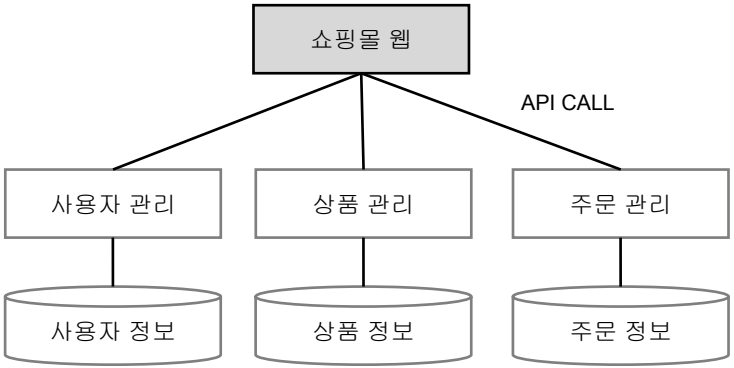
서비스 조합

- 하나의 기능을 구현 하는데, 여러개의 서비스를 조합하여 기능을 제공

  예) 주문 하기 : 사용자 정보 조회, 상품 정보 조회, 신규 주문 생성



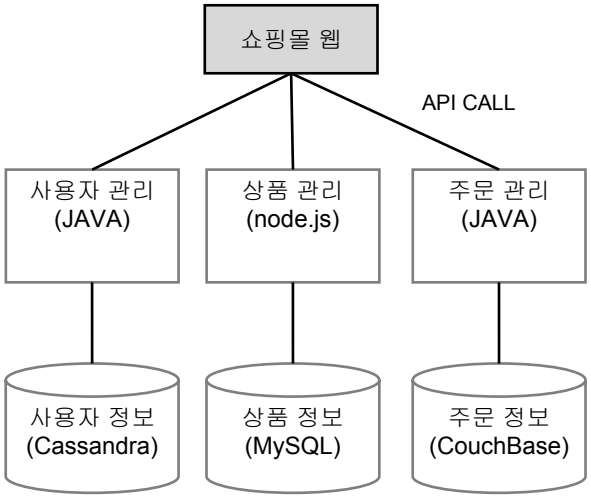
모노리틱
아키텍쳐



마이크로 서비스
아키텍쳐

기술 스택

- 마이크로 서비스 아키텍쳐는 서비스 별로 다른 기술 스택을 사용할 수 있음



장점일까?

- 운영 관점에서 여러가지 기술을 동시에 다뤄야 함
  (Devops – You build, You run)
- 사람이 떠나면 보수 불가
  (Product not a project)

단점일까?

- 적절한 기술을 적절하게 배치 가능
  - 복잡한 데이타 RDBMS, 양이 많은 고속 데이타 NoSQL
  - C10K NoSQL, 빠른 개발 스크립트 언어, 튼튼한 시스템은 자바 …

팀 모델

- 컨웨이의 법칙
  - 뼈저리게 느낌

“소프트웨어의 구조는 그 소프트웨어를 만드는 팀의 구조와 일치한다.”

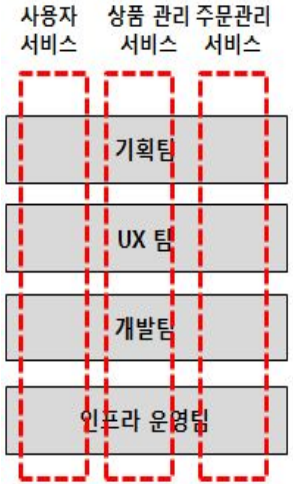
설계 백날 잘해야 소용없음
제대로 된 팀 구조를 만드는 것이 설계 (그 다음은 알아서 됨 ?)

- 친한팀 컴포넌트끼리는 개발이 잘됨. 그러다 보니 그쪽으로 기능이 몰림
- 잘하는 팀한테 자꾸 중요 기능이 몰림

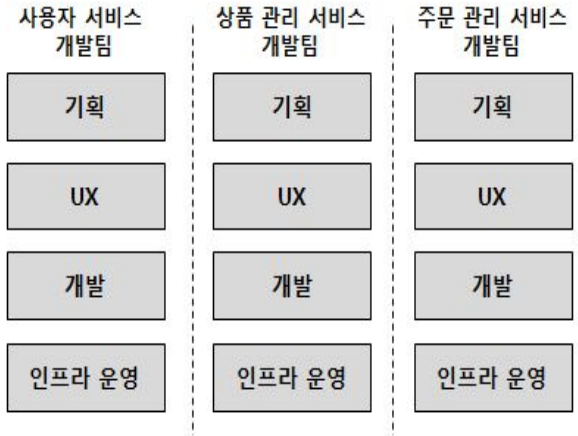
서비스 컴포넌트들이 균등하게 디자인 되지 않음

팀모델

- 분산형 거버넌스 (각 팀이 알아서. 적합한 기술, 스스로 하기)
- You build & You run!!
- Self Organized Team
- Product vs Project
- Cross functional team
- Alignment (소통!!)



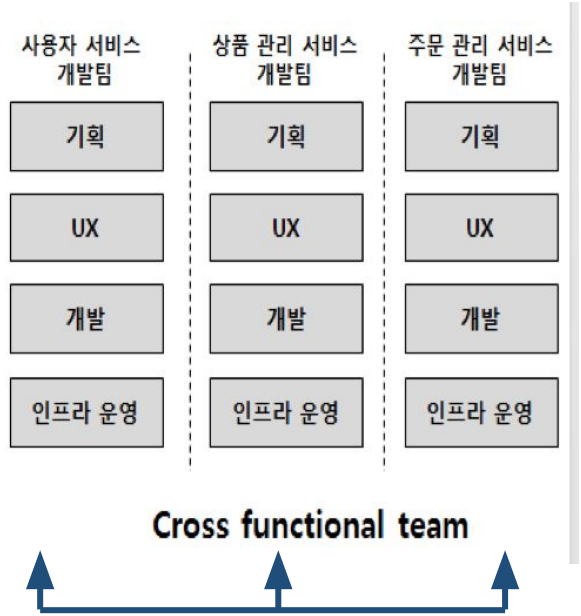
역할 중심의 개발팀



Cross functional team

팀 모델

- 팀간 조율이 핵심
  - 새로운 조율자 ROLE(

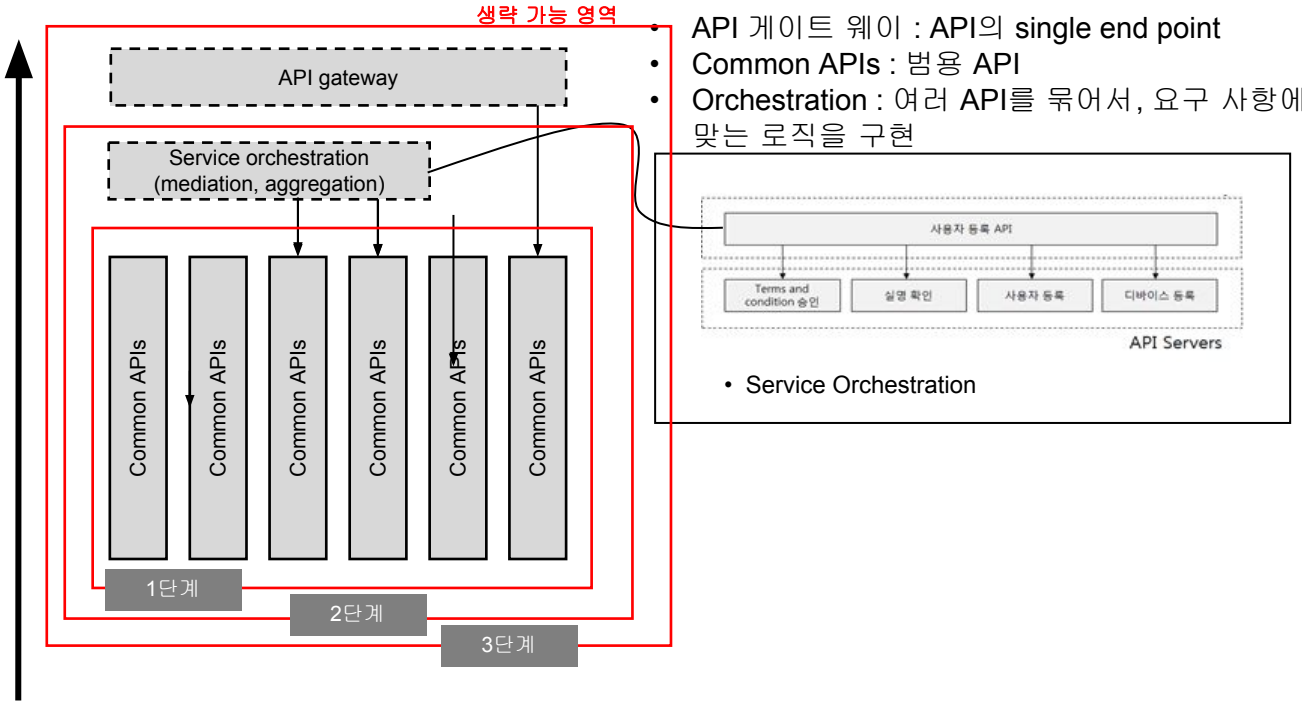


프로그램 메니져 : 팀간 일정 조율
치프 아키텍트 : 서비스간 흐름 정의, 표준 정의, 에러 추적/처리 방법 정의

MSA Reference Architecture

마이크로 서비스 아키텍쳐 토폴로지

- 일반적인 마이크로 서비스 아키텍쳐 스택 구조



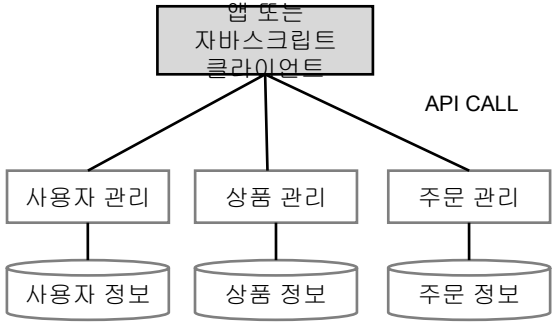
- API 게이트 웨이 : API의 single end point
- Common APIs : 범용 API
- Orchestration : 여러 API를 묶어서, 요구 사항에 맞는 로직을 구현

규모가 커질 수 록 추가되는 계층들 (오케스트레이션, API 게이트 웨이)

서비스 오케스트레이션 계층의 활용

다른 프로토콜 사용 가능
ex)내부 PB, 외부 REST

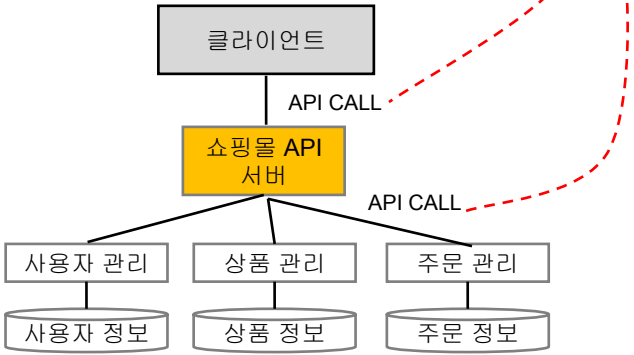
클라이언트에서 직접 서비스를 조합 하는 방식



- API가 범용적으로 잘 짜야 있어야함
- 클라이언트 팀이 모든 컴포넌트팀과 조율이 필요

규모가 크지 않은 구조에서 효과적

별도의 조합 계층(Orchestration or Mediation)을 넣는 방식

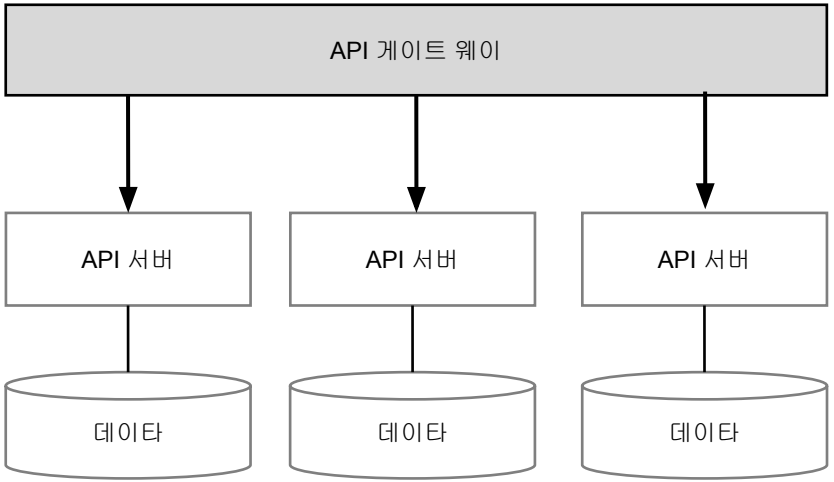


- 중간에 API를 커스터마이제이션 또는 조합하는 계층을 별도로 둠
- 클라이언트팀은 조합 API 개발팀과 커뮤니케이션만 하면 됨
- 클라이언트 요구 사항에 기민하게 대처
- 그러나 계층은 하나 더 늘어남 (성능, 디버깅,배포)

일정 규모 이상. 특히 클라이언트가 여러개인 구조에

API 게이트 웨이

- 클라이언트와 API 서버 앞에서 일종의 프록시 처럼 위치 하여 다양한 기능을 수행함
  - API 인증/인가
  - 로깅
  - 라우팅
  - 메시지 변환
  - 메시지 프로토콜 변환
  - :

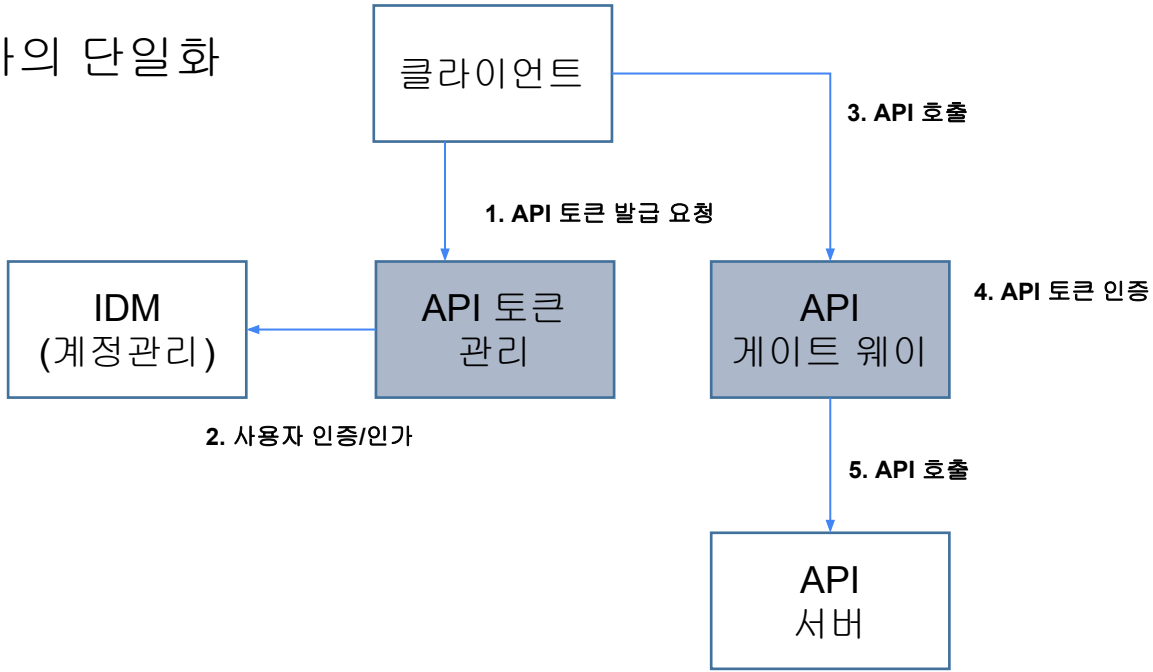


API 게이트 웨이

- SOA ESB (Enterprise Service Bus)의 단순화 버전
- 있으면 좋음. 없어도 됨
- 만들 수 있는 실력있으면, 쓰는게 좋음
- 잘못 쓰면 망하는 지름길

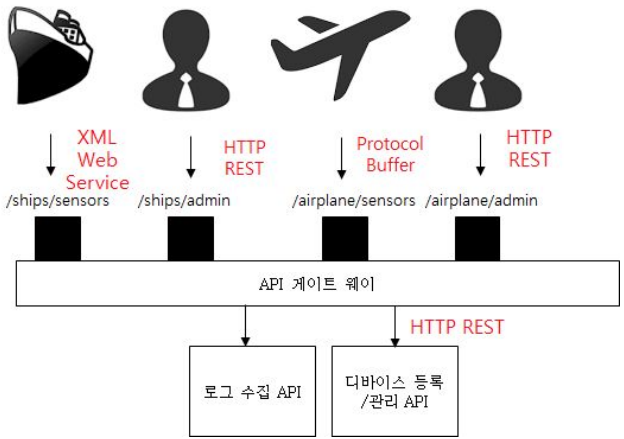
API 게이트웨이를 이용한 설계 패턴 #1

- 인증,인가의 단일화



API 게이트웨이를 이용한 설계 패턴 #2

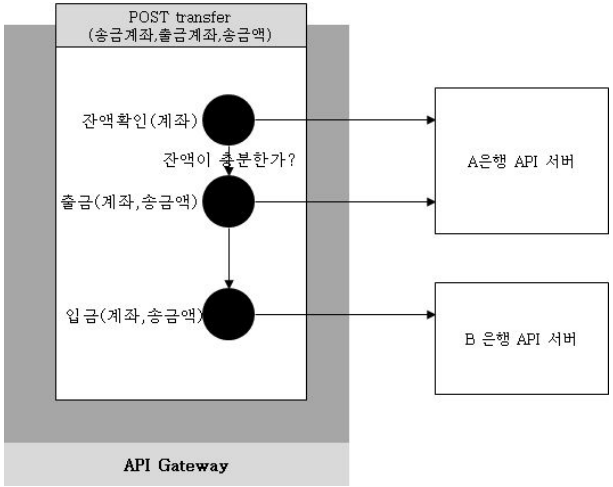
- 멀티 앤드포인트와 멀티 프로토콜 제공



<그림. 다양한 디바이스로 부터 정보를 수집하는 IOT시스템에 타입별 앤드 포인트 제공하는 예제>

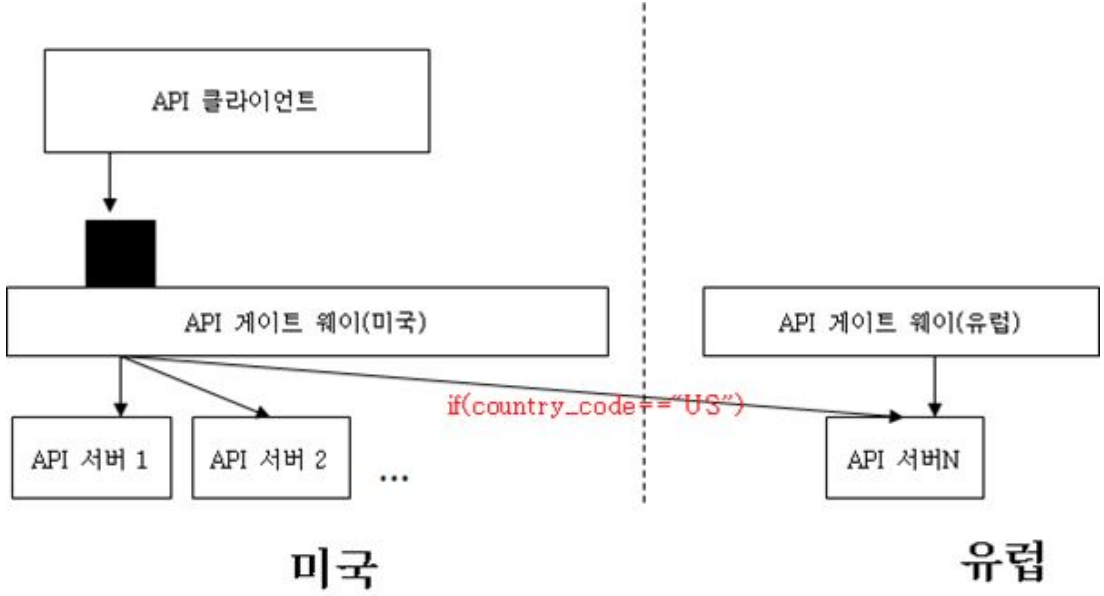
API 게이트웨이를 이용한 설계 패턴 #3

- 오케스트레이션
  - ESB 기반 SOA 프로젝트가 실패한 대부분의 원인 (안하는게 좋음)
  - 오케스트레이션 서버를 별도로 두는게 좋음



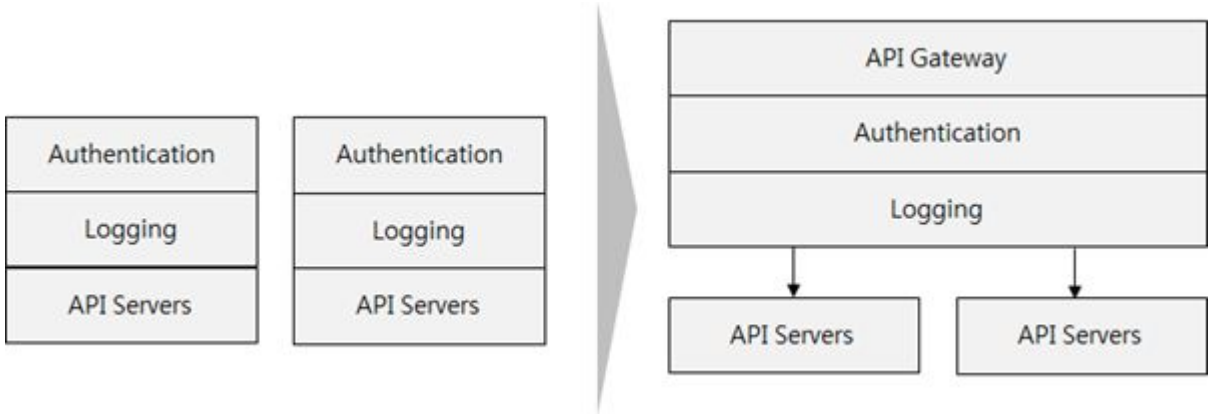
API 게이트웨이를 이용한 설계 패턴 #4

- 메세지 기반 라우팅
  - 글로벌 배포 시스템에 유용함
  - 멀티 버전 시스템 (레거시 업그레이드)에 유용



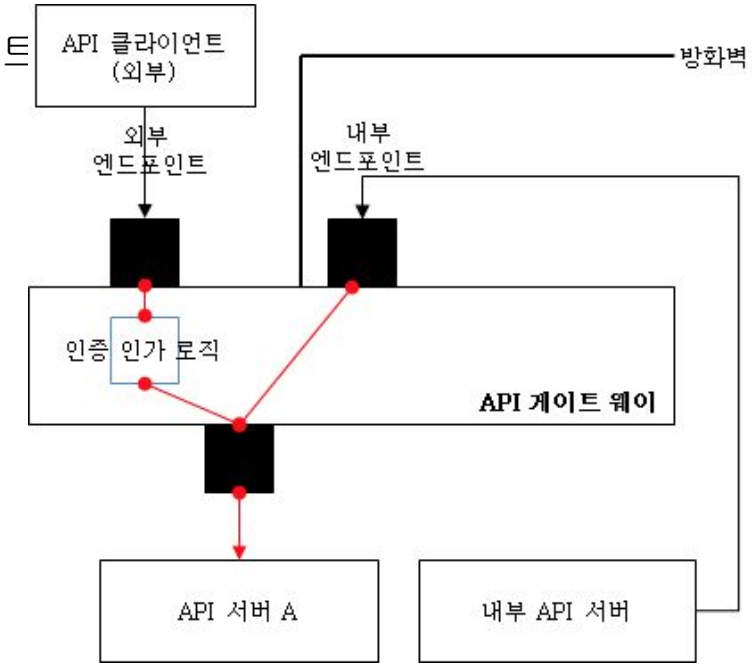
API 게이트웨이를 이용한 설계 패턴 #5

- Cross Cutting Concern (공통 기능) 처리
  - 인증,로깅등 공통 기능 처리
  - API 개발팀은 비지니스 로직에 집중할 수 있도록 해줌



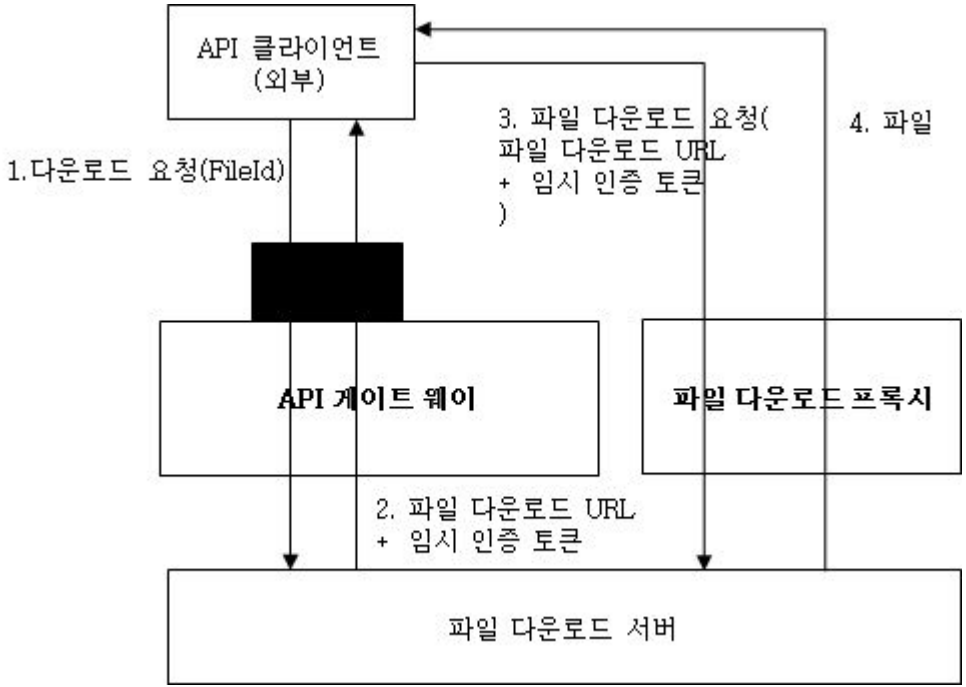
API 게이트웨이를 이용한 설계 패턴 #6

- 다중 API 게이트 웨이 패턴
  - 내부,외부용 API 게이트



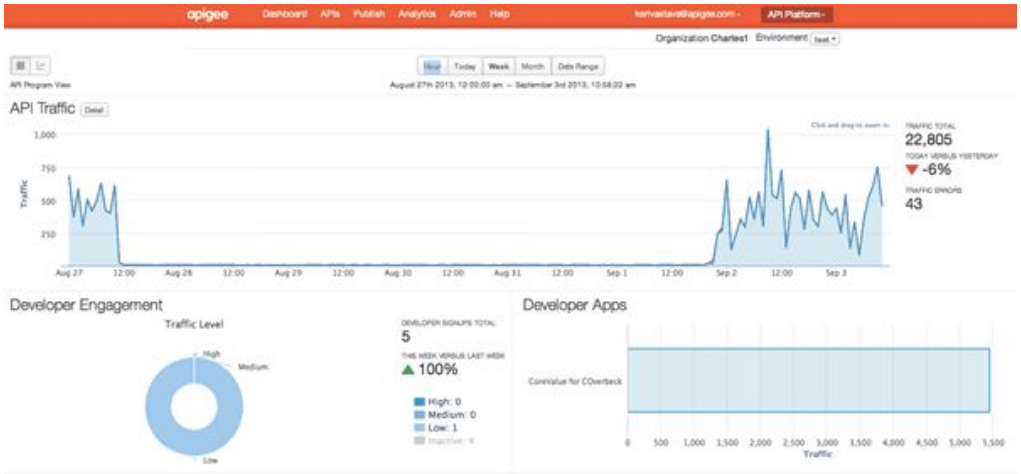
API 게이트웨이를 이용한 설계 패턴 #7

- API 호출용 엔드포인트와 스트리밍 (파일)용 엔드포인트 분리



API 게이트웨이를 이용한 설계 패턴 #6

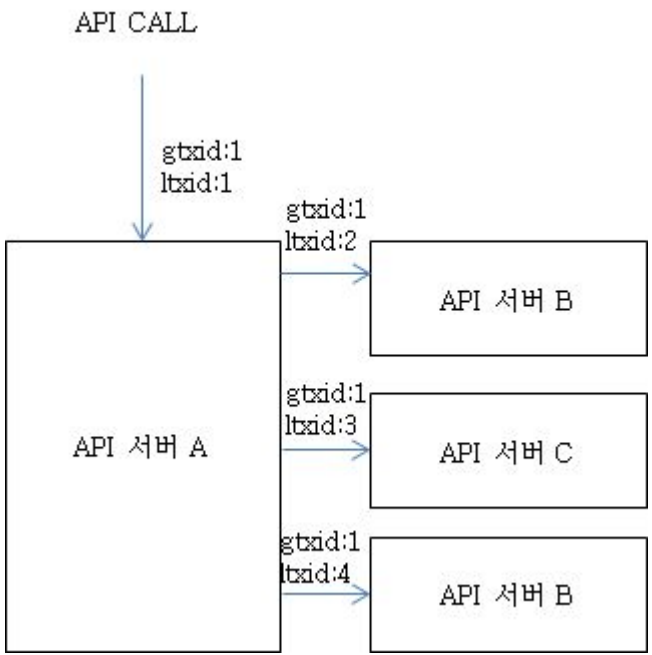
- 비 기능 요소 제어
  - QoS 제어
  - Metering & Charging (상용 API 서비스 과금)
  - API 모니터링



분산 트렌젝션의 추적

- 여러개의 서비스 컴포넌트를 조합하여 움직이는 트렌젝션에 대한 추적 디자인 패턴
- **원리 (XA 분산 트렌젝션과 유사)**
  - 초기 API에서 GTXID와 LTXID를 생성
  - 서버를 넘어갈때 마다 같은 GTXID를 사용, LTXID는 하나씩 증가
- **구현시**
  - 서버간에는 HTTP 헤더로 TXID 전달
  - 서버내에서는 Thread Local (Java)등의 컨택스트 변수 활용

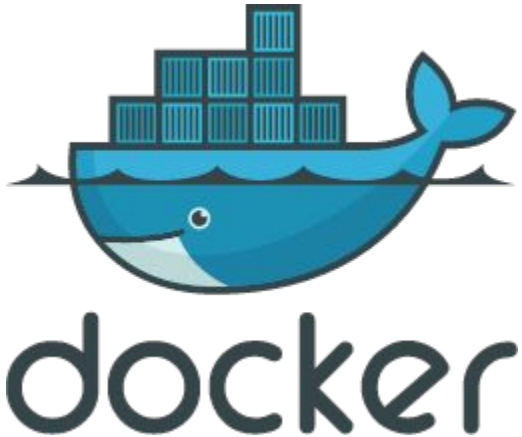
    → 초기 표준 설계가 중요함



Docker

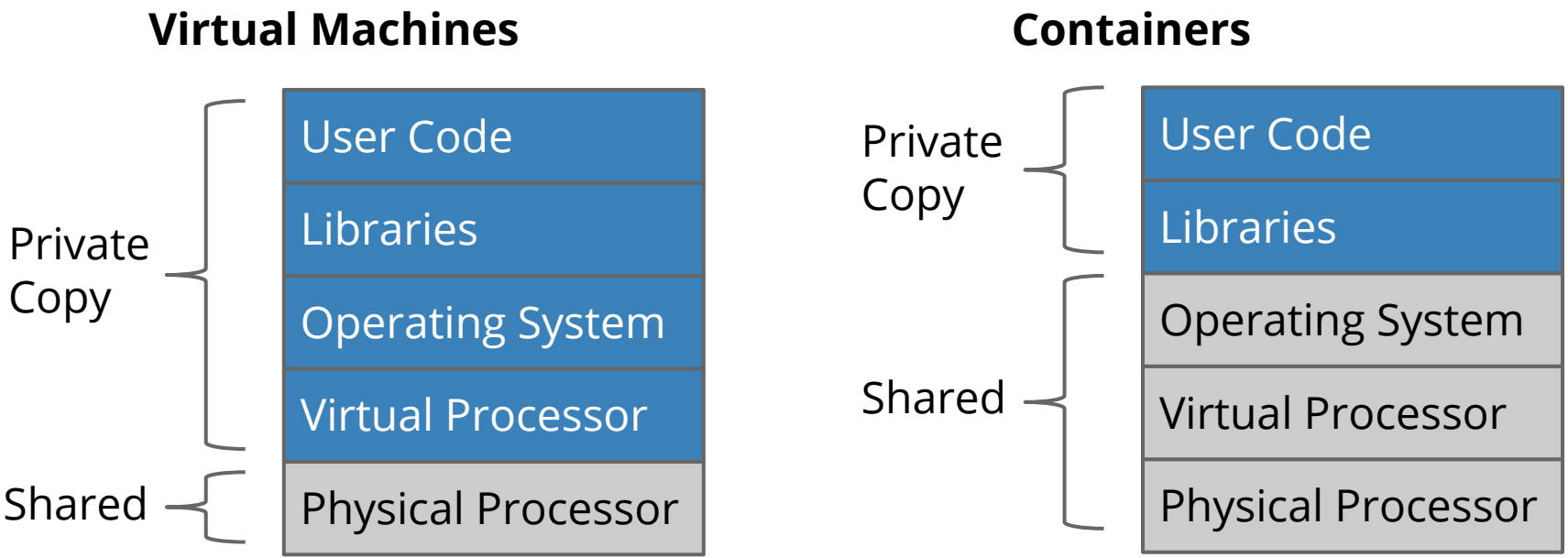
Container

- Linux Container (LxC)기반의 컨테이너 기술
- 애플리케이션 코드를 컨테이너로 패키징 해서, 개발에서 운영 환경으로 배포



Container ?

VM과 같은 컨셉이지만, 훨씬 가벼움



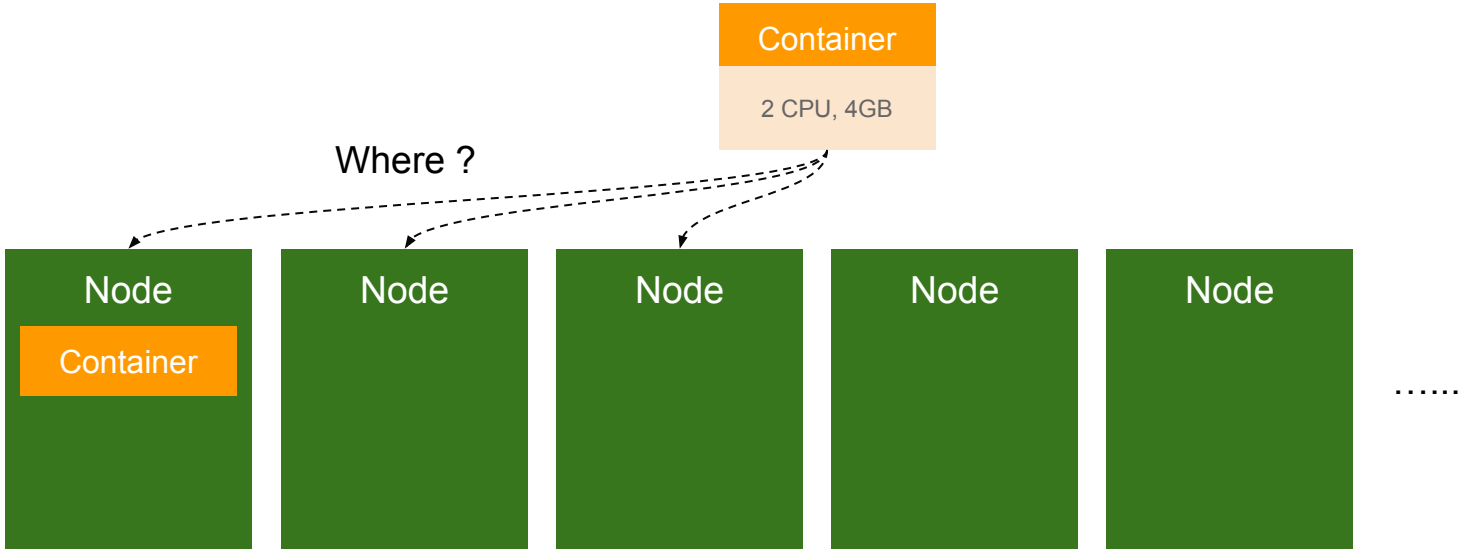
Container vs VM

- 비교

	Virtual machine	Docker container
이미지 크기	큼	작음
부팅 속도	분단위	초단위
이미지 생성 시간	느림 (+10분)	빠름 (수분)
윈도우 지원여부	가능	불가(?)
호환성	하이퍼바이져에 종속	Docker 지원 환경이면 사용 가능

Large scale container service

- 컨테이너를 어느 서버(물리)에 배포하지?



스케쥴러

- Docker Swarm
- Apache Mesos
- Kubernetes

스케쥴링 만으로 충분한가?

- Load balancing
- Health check
- Rolling upgrade
- Container Image management (repository)

:

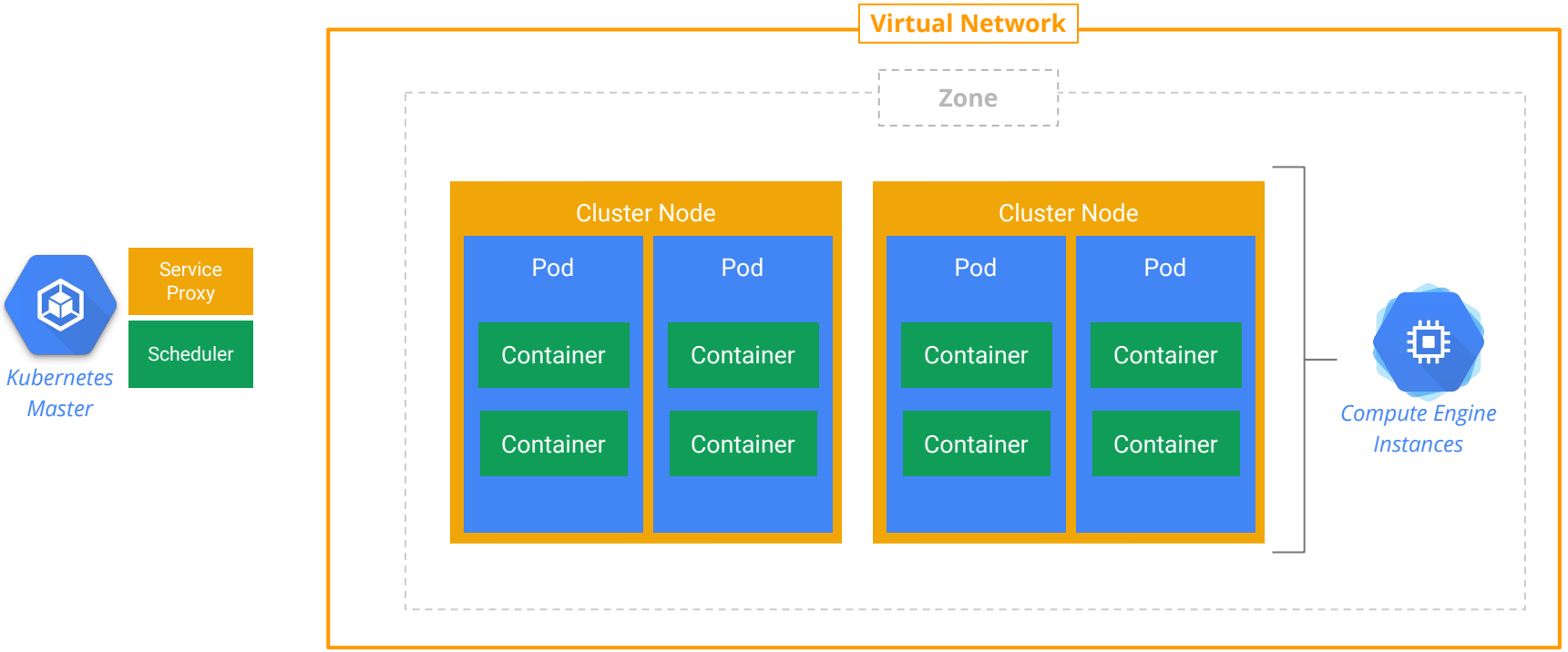
Kubernetes

Each week Google launches over
2 billion containers

Kubernetes

- 구글의 오랜 컨테이너 서비스 운영 경험의 산물
- 오픈소스
- public, private 거의 모든 인프라에서 사용 가능
- 넓은 에코 시스템 (파트너, 클라우드 서비스 제공자등)

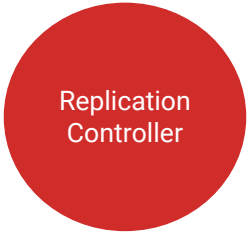
Container Engine



Cluster Resources

Pods are **ephemeral** units that are used to manage one or more tightly coupled containers.

They enable **data sharing** and **communication** among their constituent components.

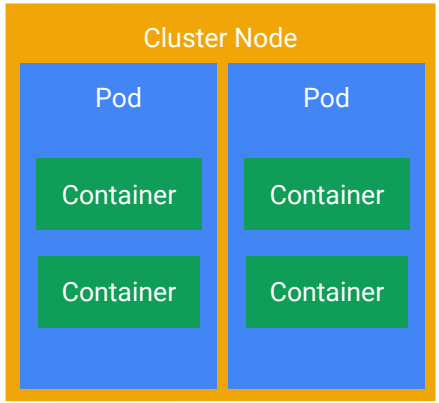


Replication controllers create new pod "replicas" from a **template** and ensure that a configurable number pods are running.

Services provide a bridge based on an **IP and port** pair for non-Kubernetes-native client applications to access backends without needing to write code that is Kubernetes-specific.

Pods

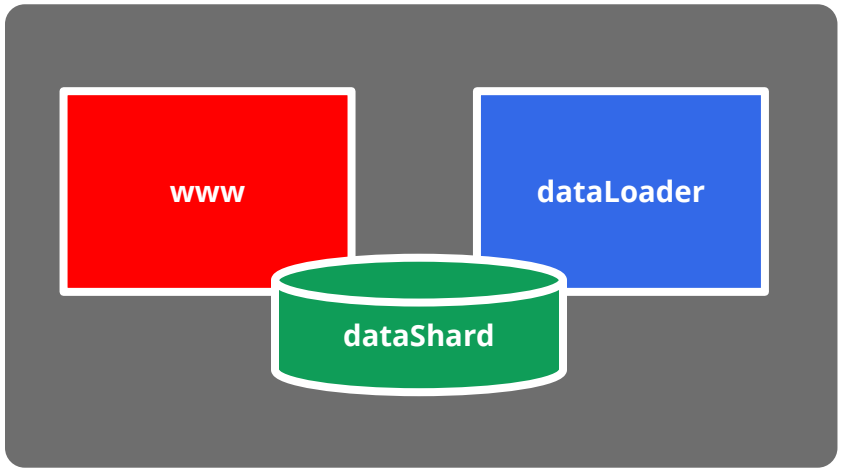
- Kubernetes 에서 최소 논리 단위
- 하나의 애플리케이션을 표현하는 최소 논리 단위
- 하나의 Pod에는 1..N개의 Container를 가질 수 있음
- 주로 Tightly Coupled 되는 Container들을 하나의 Pod에 묶음
  - 예) Nginx + Tomcat
  - 예) Tomcat + memcached
- Pod에 있는 Container는 물리적으로 같은 서버에 생성됨



Pods

- 같은 Pod에 있는 Container는 Disk Volume을 공유할 수 있음

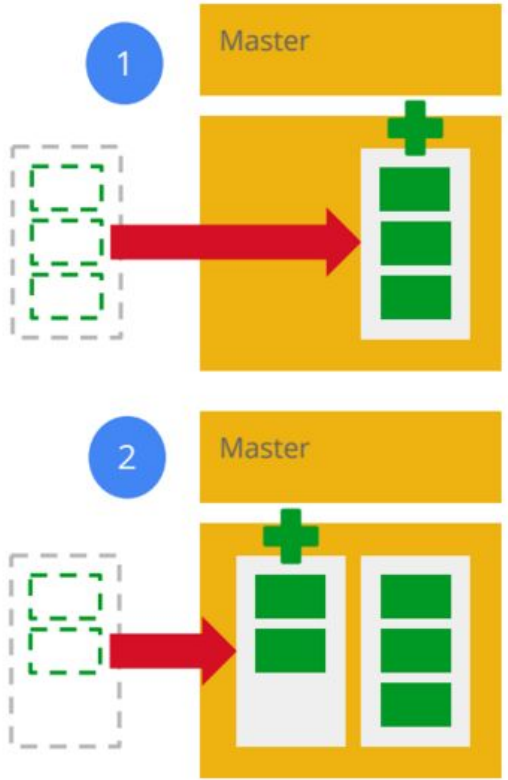
```
version: v1beta1
containers:
  - name: www
    …
    volumeMounts:
      - name: dataShard
        path: /mnt/shard
        readOnly: true

  - name: dataLoader
    …
    volumeMounts:
      - name: dataShard
        path: /mnt/output

volumes:
  - name: dataShard
```



Replication Controllers

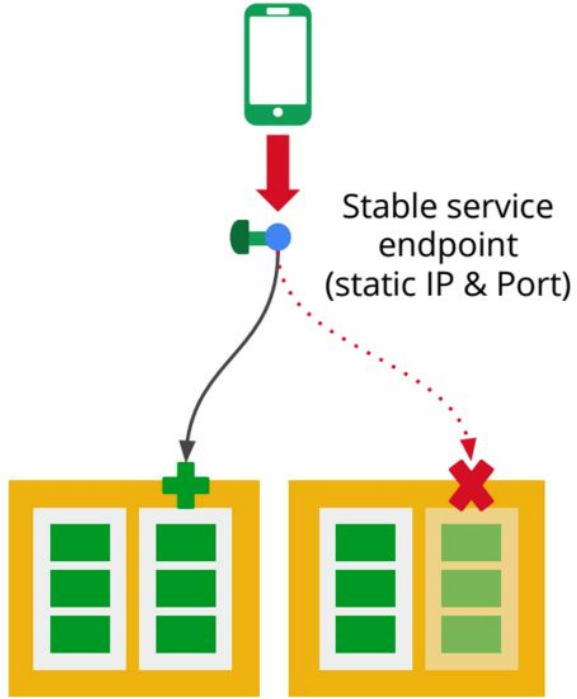
- Pod를 생성하고 관리.
- Pod 생성은 미리 정의된 Template에서 부터 생성
- Pod의 가동 상태를 체크하고, 죽으면 다시 재기동.

```
version: v1beta1
containers:
  - name: www
    image: nginx
    ports:
      - name: http
        hostPort: 8080
        containerPort: 80
```



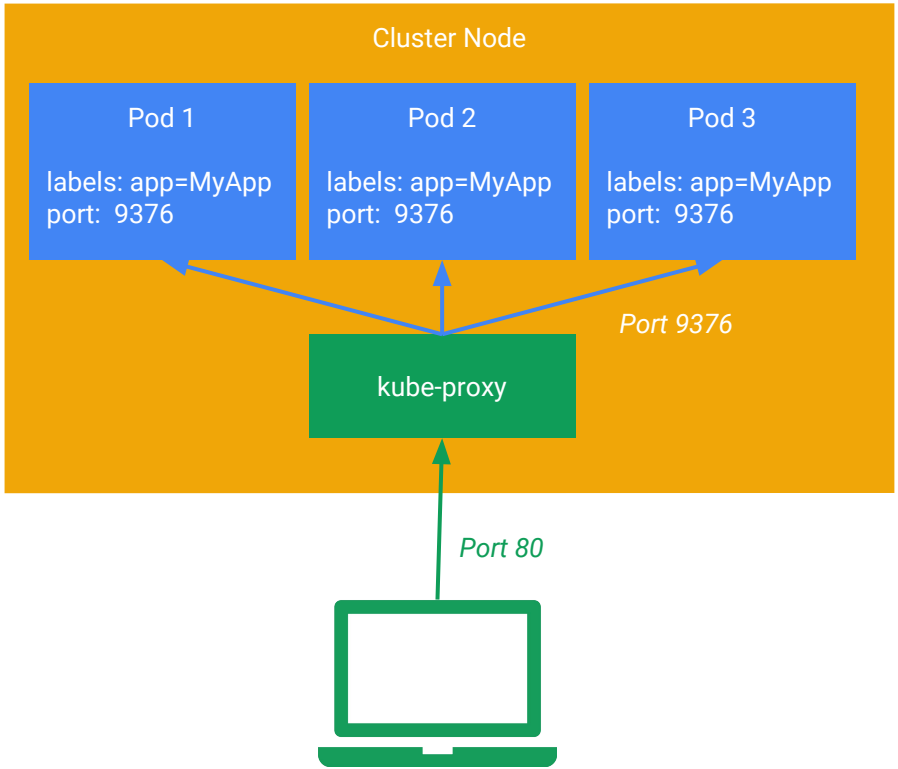
Services

- Pod 들의 집합 (예. 웹서버 서비스, 백앤드 서비스, 캐쉬 서비스)
- IP Address가 지정 되고, Pod 간에 로드 밸런싱을 제공

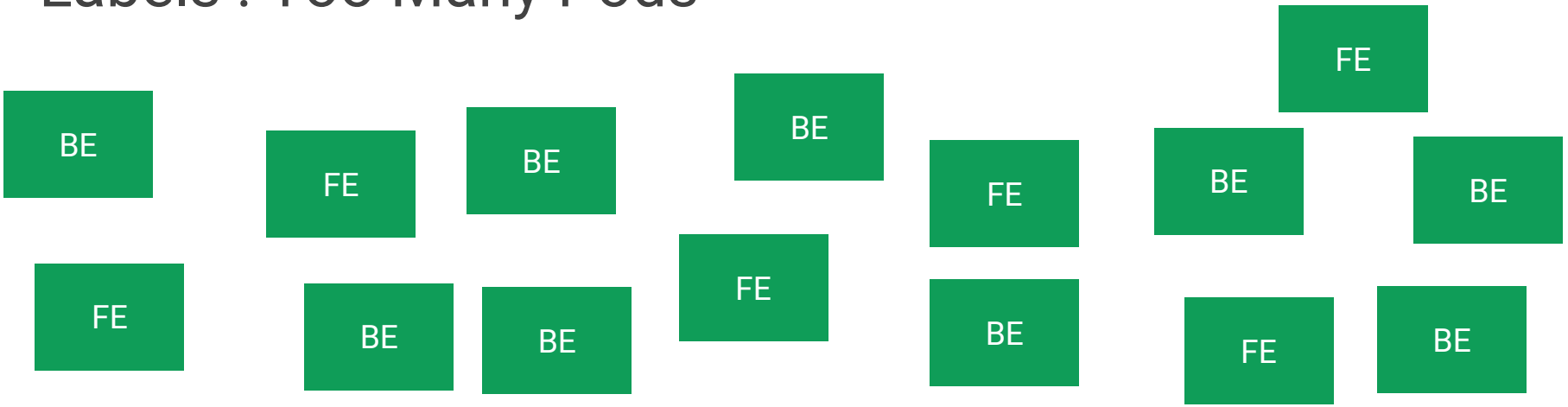


Services

```
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "my-service"
    },
    "spec": {
        "selector": {
            "app": "MyApp"
        },
        "ports": [
            {
                "protocol": "TCP",
                "port": 80,
                "targetPort": 9376
            }
        ]
    }
}
```



Labels : Too Many Pods

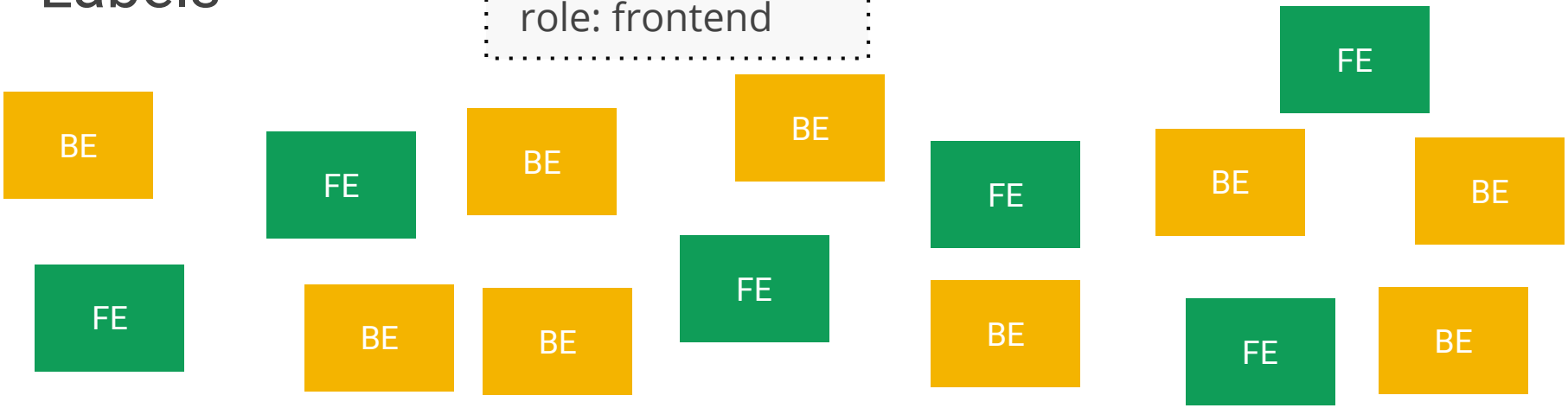


Kubernetes - Master/Scheduler

Container Agent	Container Agent	Container Agent	Container Agent	Container Agent	Container Agent	Container Agent
Machine Host	Machine Host	Machine Host	Machine Host	Machine Host	Machine Host	Machine Host

Labels

labels:
role: frontend

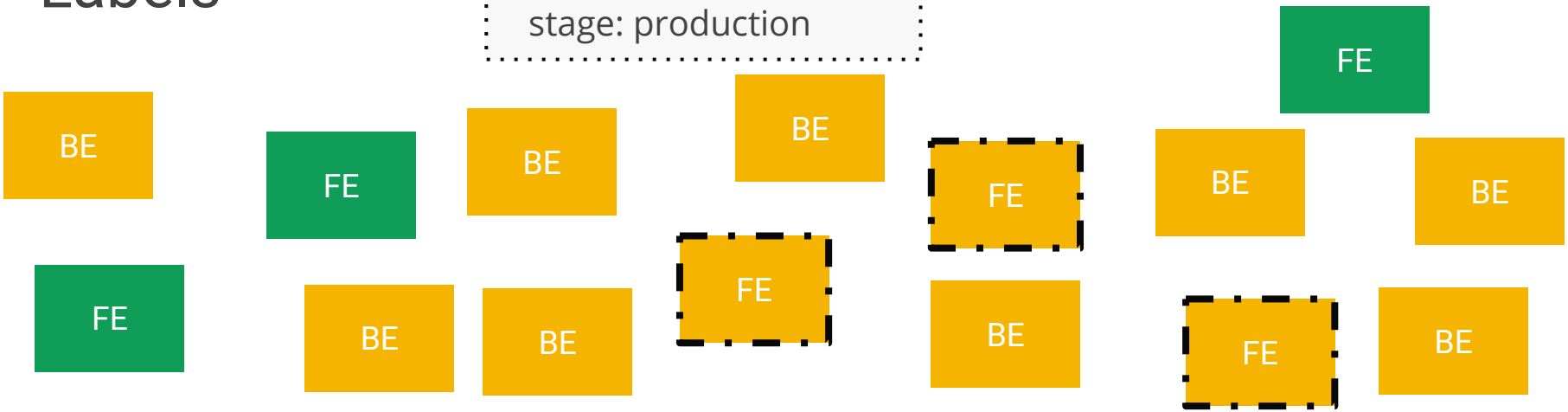


Kubernetes - Master/Scheduler

Container Agent	Container Agent	Container Agent	Container Agent	Container Agent	Container Agent	Container Agent
Machine Host	Machine Host	Machine Host	Machine Host	Machine Host	Machine Host	Machine Host

Labels

labels:
role: frontend
stage: production



Kubernetes - Master/Scheduler

Container Agent	Container Agent	Container Agent	Container Agent	Container Agent	Container Agent	Container Agent
Machine Host	Machine Host	Machine Host	Machine Host	Machine Host	Machine Host	Machine Host

Kubernetes on Cloud

New service for cluster-based compute

- 수초내에 Kubernetes 클러스터 생성
- 구글 클라우드 뿐 아니라, 타 클라우드 + 자체 데이타 센터를 단일 Kubernetes로 관리

Releases

- Today: General Availability

Resources

- Google Container Engine: http://cloud.google.com/container-engine
- Kubernetes: http://kubernetes.io

DEMO

Free credit!!

페이스북 구글 클라우드 사용자 그룹

https://www.facebook.com/groups/googlecloudkorea/

감사합니다.