

# Container Orchestration (Docker & Kubernetes)



Copyright © 2020. uEngine-solutions All rights reserved.

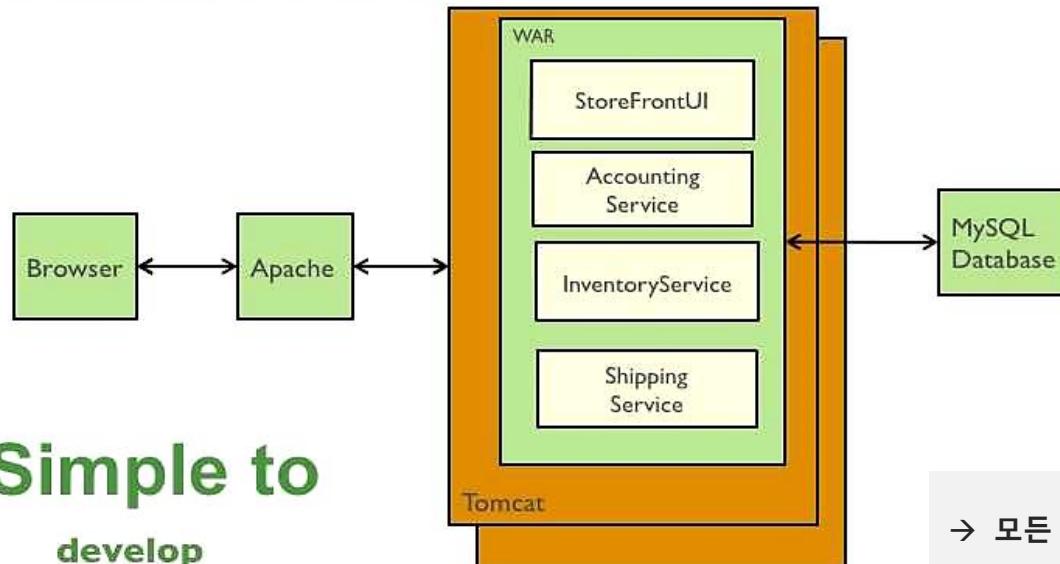
# Table of content

Container Orchestration  
(Docker & k8s)

1. MSA, Container, and Container Orchestration ✓
2. Setup Azure Platform
3. Docker / Kubernetes, Kubernetes Architecture
4. Kubernetes Basic Object Model : Pod, Label, ReplicaSet, Deployment, Service, Volume, Configmap, Secret, Liveness/Readiness
5. Kubernetes Advanced Lab : Ingress, Job, Cron Job, DaemonSet, StatefulSet
6. Service Mesh: Istio for Advanced Services Control
7. Course Test

# Monolithic Architecture

Traditional web application architecture



Simple to

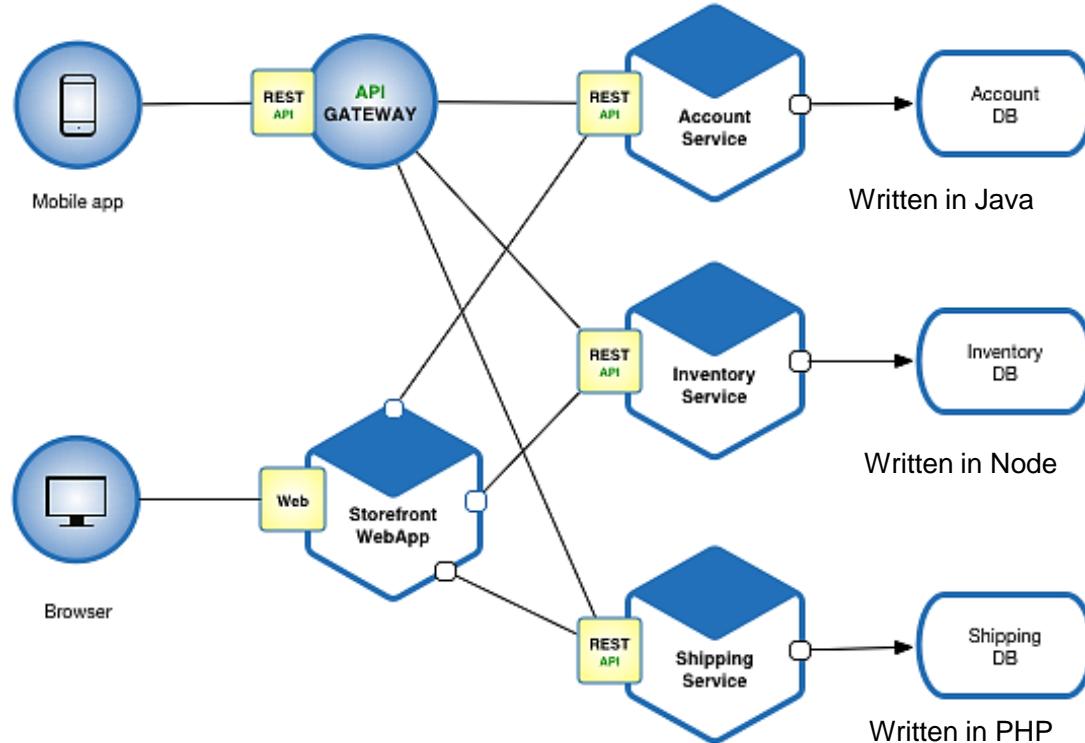
develop  
test  
deploy  
scale

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

# MSA Architecture

Contract based, Polyglot Programming

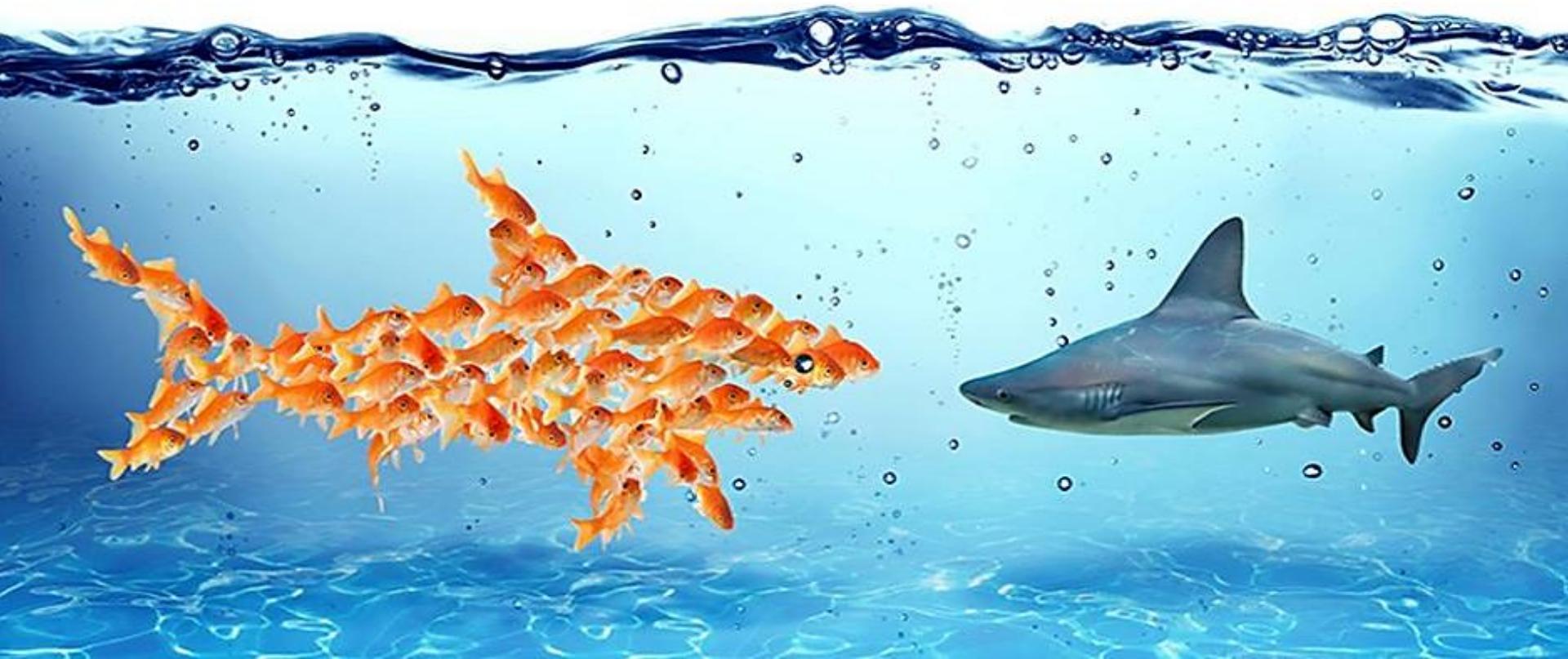
→ Separation of Concerns, Parallel Development, Easy Outsourcing



“

서비스가 죽지 않으면 어떨까?

”



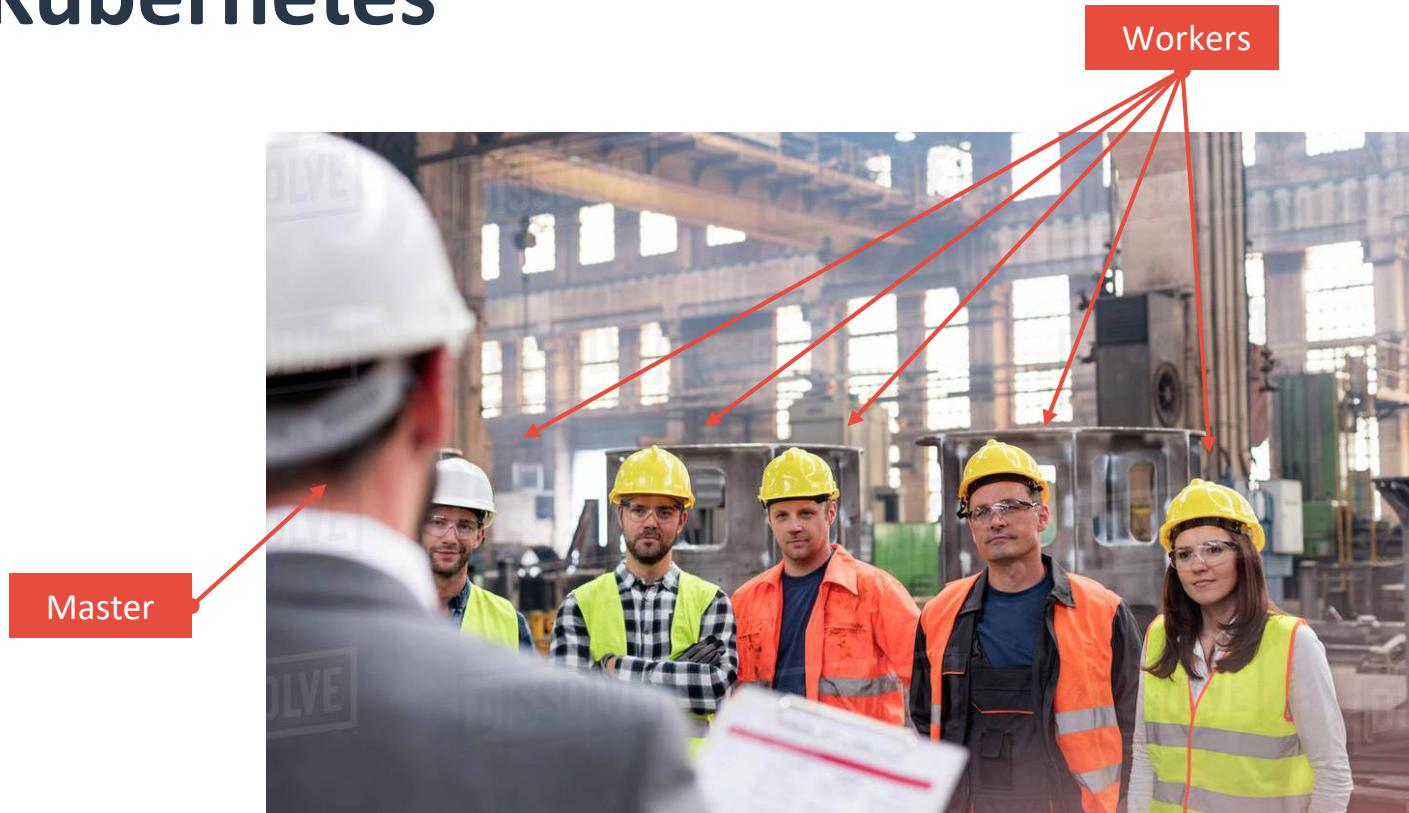
“

# 항상성이 (Self-Healing) 자동으로 유지되면 어떨까?

”



# Kubernetes



# Container-based Deployment

## Tools: Evolution of Module Systems

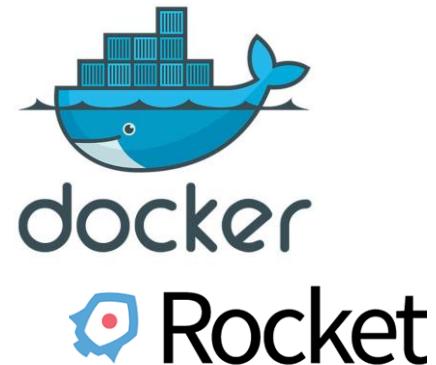
Java Module System



Hypervisors



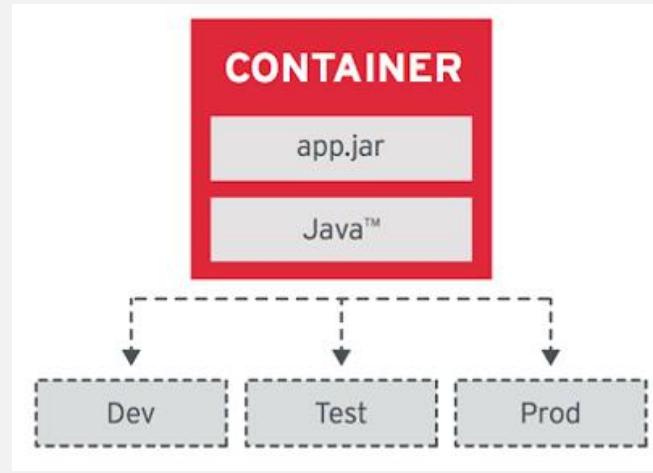
Containers



# Principles of Container-based Application Design

- Image Immutability
- High Observability
- Process Disposability
- Lifecycle Conformance
- Runtime Confinement
- Single Concern
- Self-Containment

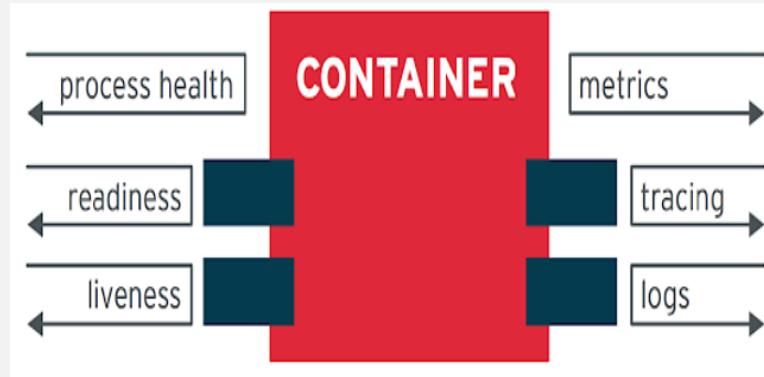
# Image Immutability Principle (IIP)



Snowflakes Server\* pattern → Phoenix Server pattern

- Immutable container images must be used across all environments
- For each environment, Container must use externalized configuration way

# High Observability Principle (HOP)



- Containerized application must provide APIs for health checks and readiness
- Application should log important events for log aggregation by tool (ex. Fluentd)

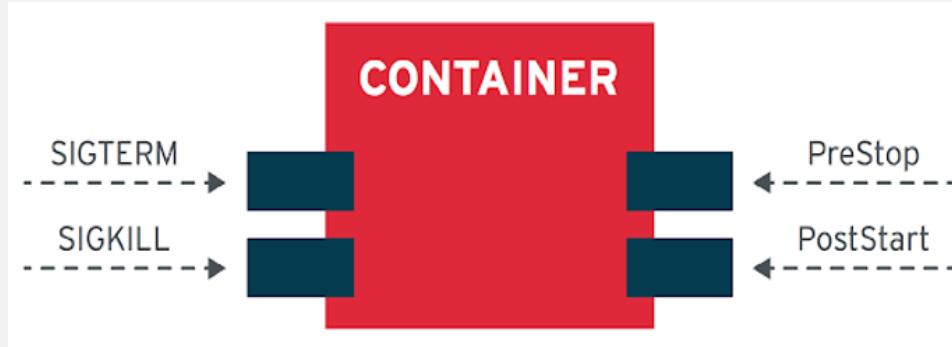
# Process Disposability Principle (PDP)



Pet and Cattle Strategy

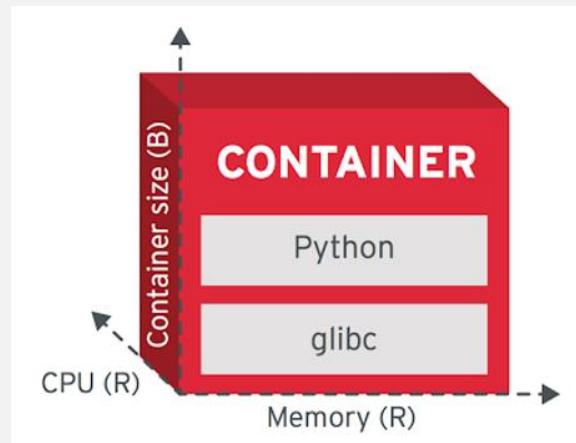
- A containerized application with **quick start up and shut down** for easy replaceability
- Container replace reasons : Failing a health check, scaling down the application, migrating the containers to a different host, etc

# Lifecycle Conformance Principle (LCP)



- A container providing APIs and conforming to platform events
- Container must conforms Managing Platform(Docker Swarm, Kubernetes) events

# Runtime Confinement Principle (RCP)



- A container that declares its runtime resource requirements and pass them to the platform and respects them at runtime
- The application stay within the indicated resource requirements is important

# Single Concern Principle (SCP)



- Every Container should address a single concern
- If Containerized Microservice needs multiple concerns, it can use sidecar and init-containers patterns, where each container still handles a single concern.

# Self-Containment Principle (S-CP)



- Containers should have baked in all dependencies at build time
- The only exceptions are things such as configurations, which must be provided at runtime; for example, through Kubernetes ConfigMap

# Conclusion of Container-based Application Design

- Cloud native is more than an end state, it is a way of working
- Some container-related best practices
  - Aim for small images
  - Support arbitrary user IDs
  - Mark important ports
  - Use volumes for persistent data
  - Set image metadata

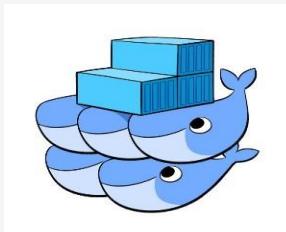
# Container Orchestrators

Kubernetes



구글에서 개발, 가장 기능이 풍부하며 널리 사용되는 오케스트레이션 프레임워크  
베어 메탈, VM환경, 퍼블릭 클라우드 등의 다양한 환경에서 작동하도록 설계  
컨테이너의 롤링 업그레이드 지원

Docker Swarm



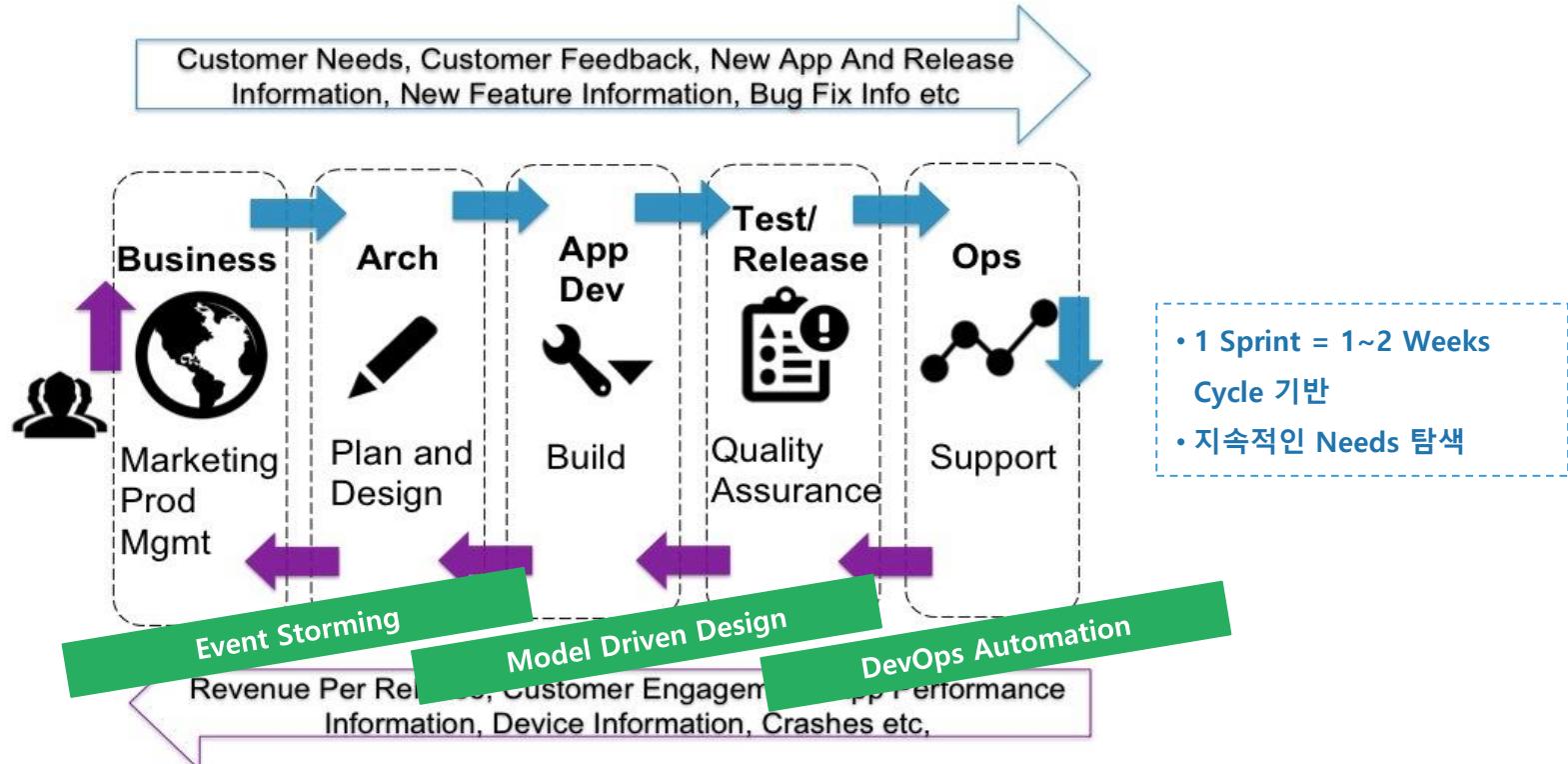
여러 개의 Docker 호스트를 함께 클러스터링하여 단일 가상 Docker 호스트를 생성  
호스트 OS에 Agent만 설치하면 간단하게 작동하고 설정이 용이  
Docker 명령어와 Compose를 그대로 사용 가능

Apache Mesos



수만 대의 물리적 시스템으로 확장할 수 있도록 설계  
Hadoop, MPI, Hypertable, Spark 같은 응용 프로그램을 동적 클러스터 환경에서 리소스 공유와 분리를 통해 자원 최적화 가능  
Docker 컨테이너를 적극적으로 지원

# BizDevOps Process



# Table of content

Container Orchestration  
(Docker & k8s)

1. MSA, Container, and Container Orchestration
2. Setup Azure Platform ✓
3. Docker / Kubernetes, Kubernetes Architecture
4. Kubernetes Basic Object Model : Pod, Label, ReplicaSet, Deployment, Service, Volume, Configmap, Secret, Liveness/Readiness
5. Kubernetes Advanced Lab : Ingress, Job, Cron Job, DaemonSet, StatefulSet
6. Service Mesh: Istio for Advanced Services Control
7. Course Test

“

# Local Setup for Docker Practice

- Linux(Ubuntu) install
- Docker Daemon, Client Setup & Configuration

# Windows 10에 Linux 설치

- PowerShell을 관리자 권한으로 실행하여 WSL 활성화

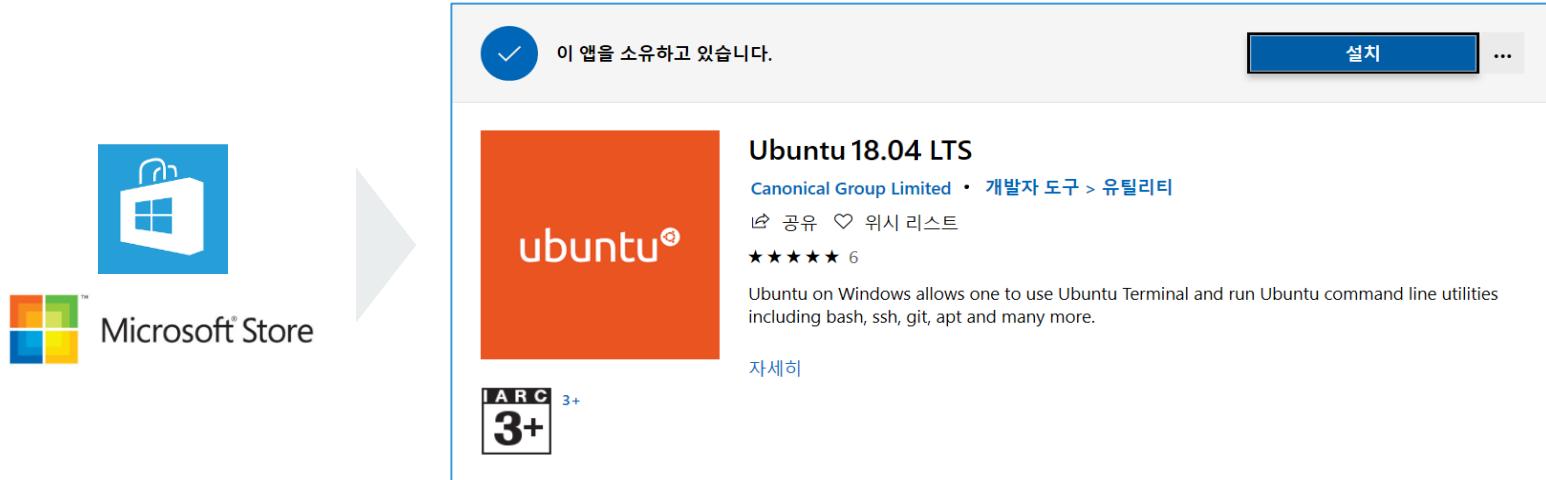
```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

- 컴퓨터 재시작

WSL(Windows Subsystem for Linux) : Windows에서 native Linux 명령을 직접 실행할 수 있는 Windows10 기능  
<https://docs.microsoft.com/ko-kr/windows/wsl/install-win10>

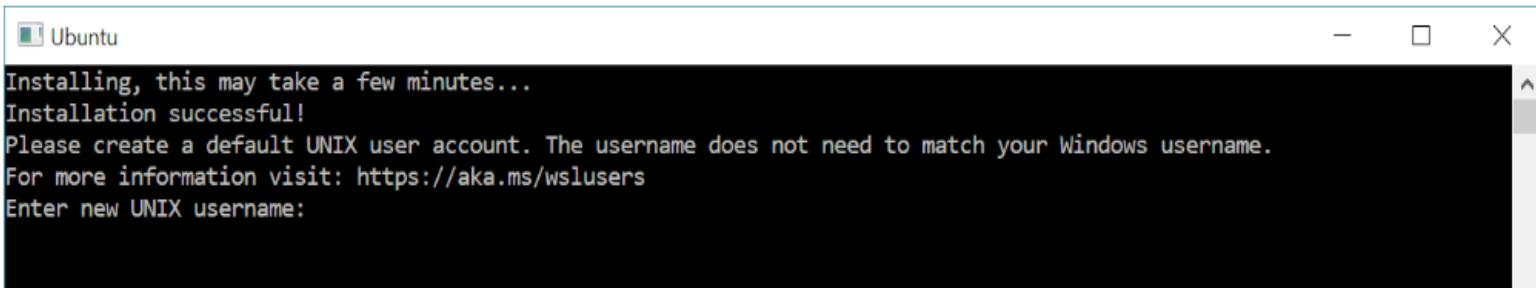
# Windows 10에 Linux 설치

- Ubuntu 설치
  - 시작메뉴 > Microsoft Store에서 ubuntu 검색 후, 18.04 version 설치



# Windows 10에 Linux 설치

- Ubuntu 초기화
  - 시작 메뉴에서 ‘Ubuntu 18.04 LTS’ 실행
  - 초기화 및 새 Linux 계정 설정



The screenshot shows a terminal window titled "Ubuntu". The window contains the following text:

```
Installing, this may take a few minutes...
Installation successful!
Please create a default UNIX user account. The username does not need to match your Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username:
```

# Linux에 JDK 설치

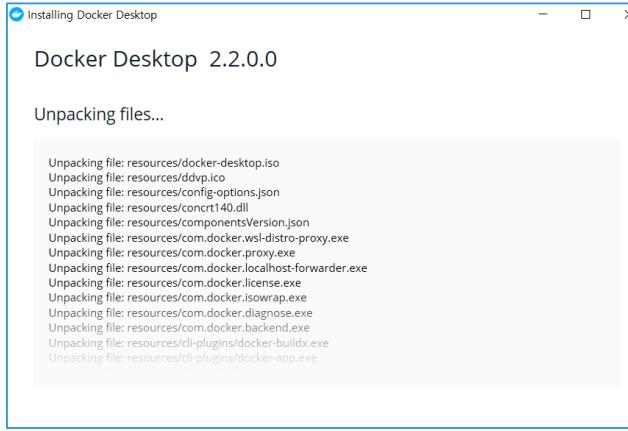
- 설치 명령
  - \$ sudo apt-get update
  - \$ sudo apt install default-jdk
- JAVA\_HOME 설정
  - ex) .bashrc에 export JAVA\_HOME='/usr/lib/jvm/java-11-openjdk-amd64' 추가
- 실행 path 추가
  - ex) .bashrc에 export PATH=\$PATH:\$JAVA\_HOME/bin:. 추가
- 수정사항 반영
  - \$ source ~/.bashrc
- 설치 확인
  - \$ echo \$JAVA\_HOME
  - java -version

# Linux에 nodeJs, npm 설치

- nodeJs 설치
  - \$ node --version
  - \$ sudo apt-get update
  - \$ sudo sudo apt-get install nodejs
- npm 설치
  - \$ npm --version
  - \$ sudo apt-get install npm
  - \$ export PATH=\$PATH:/usr/bin/npm
- 설치확인
  - \$ node --version
  - \$ npm --version

# Docker Setup - Daemon

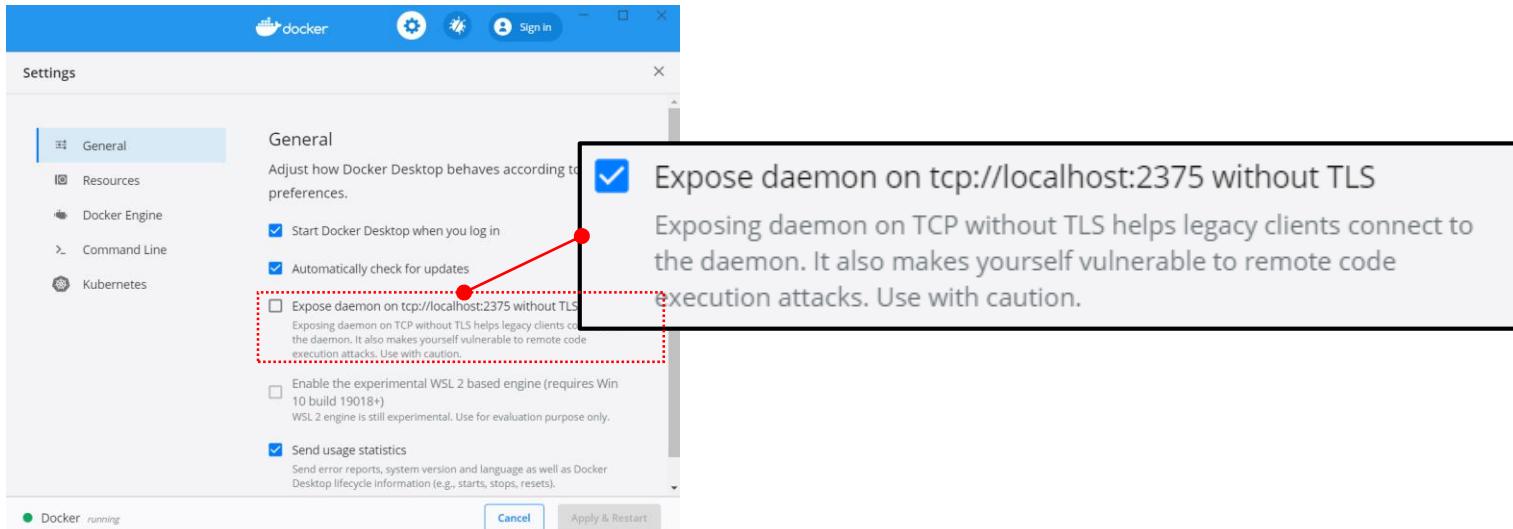
- Windows 10에 Docker 데몬(Docker for Windows) 설치
  - <https://www.docker.com/products/docker-desktop>



- 컴퓨터 재시작

# Docker Setup – Daemon config

- Docker Hub(<https://hub.docker.com/>)에 접속하여 계정 생성
- Docker Desktop 실행 후, Docker 계정으로 Sign in
- Settings 메뉴 ‘Expose daemon on~~’ 체크 후 재시작



# Docker Setup – Client

- Linux에 Docker Client 설치

- sudo apt-get update

```
# Install packages to allow apt to use a repository over HTTPS.
```

- sudo apt install apt-transport-https ca-certificates curl software-properties-common

```
# Add Docker's official GPG key.
```

- curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

```
# Pick the release channel.
```

- sudo add-apt-repository "deb [arch=amd64]  
https://download.docker.com/linux/ubuntu bionic stable"

- sudo apt update

```
# Install the latest version of Docker CE.
```

- **sudo apt install docker-ce**

```
# Allow your user to access the Docker CLI without needing root
```

- sudo usermod -aG docker [*Linux username*]

# Docker Setup – connect Client to Daemon

- Linux 계정에 Docker Daemon 환경변수 설정
  - vi ~/.bashrc
  - export DOCKER\_HOST=tcp://0.0.0.0:2375 # 맨 아래에 환경변수 추가
  - source ~/.bashrc # Shell 재 실행
- Docker Setup 확인
  - \$ docker run --name=nginx -d -p 80:80 nginx
  - \$ docker images
  - \$ docker container list
  - 브라우저에서 80포트 접속확인



“

# Cloud Setup

- Create Azure account and Subscription
- Register DevOps team & Project define
- Resource Group & Kubernetes Cluster creation
- Azure Container Registry creation

# Cloud Platform



- 직관적이고 편한 UI
- 높은 윈도우 서비스 호환
- 국내 2개 리전 보유
- VM 생성 및 실행 속도



- 우세한 시장 점유율
- 높은 서비스 성숙도
- 광범위한 서비스 교육
- UI 및 사용성, 이용 요금

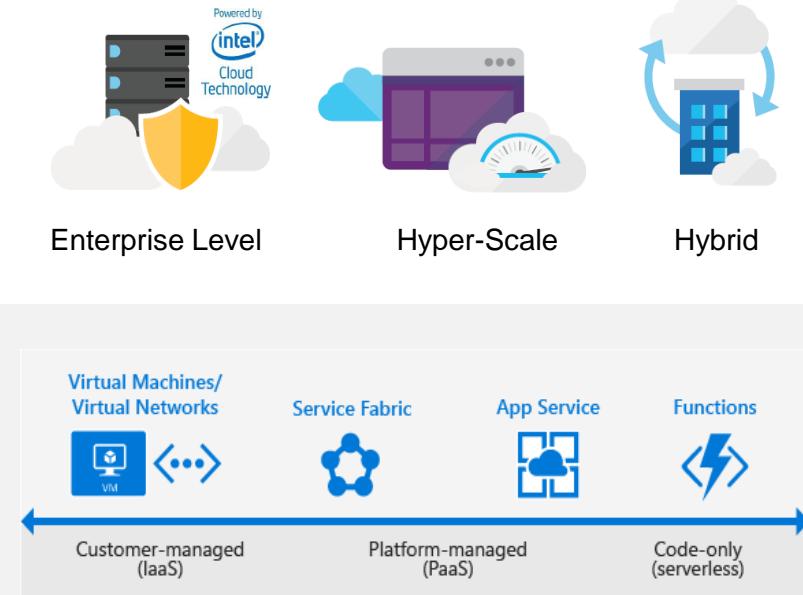


- Cloud-friendly Biz 디자인
- 전문적인 DevOps 보유
- 높은 컴퓨터 오퍼링
- 높은 Windows VM 비용

# Azure Cloud Platform

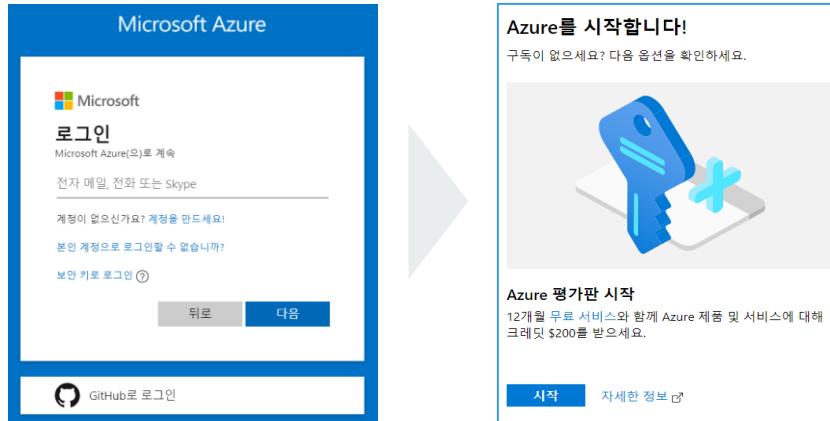


“가장 많은 글로벌 지역 보유, 56개 리전 140개 국가”



# Azure Cloud Setup

- Azure 계정 생성
  - <http://portal.azure.com> 접속

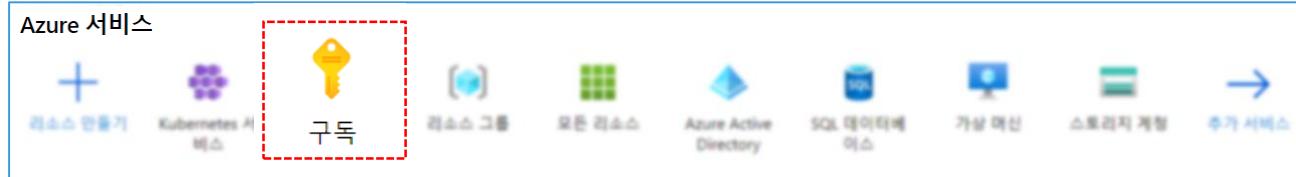


평가판 무료서비스

[https://portal.azure.com/?quickstart=true#blade/Microsoft\\_Azure\\_Billing/FreeServicesBlade](https://portal.azure.com/?quickstart=true#blade/Microsoft_Azure_Billing/FreeServicesBlade)

# Azure Cloud Setup

- 무료체험 구독(Subscription) 생성



The screenshot shows the "구독" (Subscription) page. It has a header with "구독" and "기본 디렉터리". Below that is a "추가" button. A message says "기본 디렉터리의 구독을 표시합니다. 구독이 보이지 않나요? 디렉터리 전환". It shows two dropdown menus: "내 역할" (8개 선택됨) and "상태" (3개 선택됨). A blue "적용" button is visible. Below this is a filter bar with "필터링 항목 검색..." and a checkbox for "다음에서 선택된 구독만 표시 글로벌 구독 필터". A table lists one subscription: "구독 이름": 무료 체험, "구독 ID": 854e08fb-f4a4-411f-af61-23cc002244e7, "내 역할": 계정 관리자, "현재 비용": ₩110,325.6, "상태": 활성. There is also a "... more" button.

구독 이름	구독 ID	내 역할	현재 비용	상태	...
무료 체험	854e08fb-f4a4-411f-af61-23cc002244e7	계정 관리자	₩110,325.6	활성	...

# Azure Cloud Setup

- DevOps 조직 및 Project 생성
  - dev.azure.com 접속 > new organization 클릭

The diagram illustrates the Azure DevOps organization creation process. It consists of two main screens connected by a large right-pointing arrow.

**Left Screen: Almost done...**

- Azure DevOps logo and user info: apexacme@uengine.org, Switch directory.
- Section title: **Almost done...**
- Input field: Name your Azure DevOps organization (Value: dev.azure.com/ uEngine-org).
- Input field: We'll host your projects in (Value: East Asia).
- Blue **Continue** button.

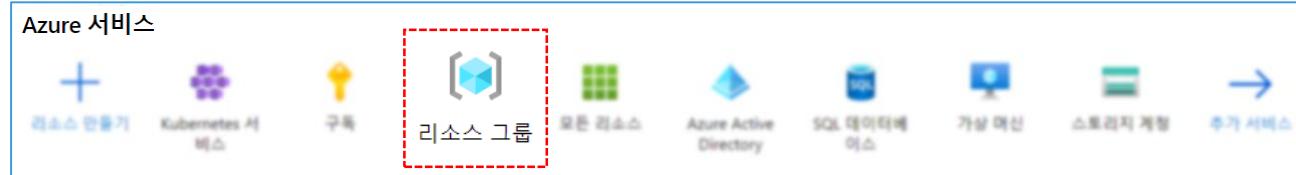
**Right Screen: Create a project to get started**

- Section title: **Create a project to get started**
- Input field: Project name \* (Value: First Project).
- Section title: Visibility
  - Public (radio button not selected)
  - Private (radio button selected)

Anyone on the internet can view the project. Certain features like TFVC are not supported.
- Blue **+ Create project** button.

# Azure Cloud Setup

- 리소스 그룹 생성



### 리소스 그룹 만들기

기본 태그 검토 + 만들기

리소스 그룹 - Azure 솔루션의 관련 리소스를 보관하는 컨테이너입니다. 리소스 그룹에 솔루션의 모든 리소스를 포함할 수도 있고 그룹으로 관리할 리소스만 포함할 수도 있습니다. 무엇이 조직에 가장 적합한지에 따라 리소스 그룹에 리소스를 할당할 방법을 결정합니다. 자세한 정보

**프로젝트 정보**

구독 \* ⓘ

리소스 그룹 \* ⓘ

무료 체험

My\_Resource\_Group

**리소스 세부 정보**

영역 \* ⓘ

(아시아 태평양) 한국 중부

# Azure Cloud Setup

- Kubernetes 서비스(Cluster) 생성



The screenshot shows the "Kubernetes 클러스터 만들기" (Create Kubernetes Cluster) wizard. The current step is "기본 사항" (Basic Information).  
Project Details:  
- Resource Group: A dropdown menu is open, showing "무료 체험" (Free Trial) and "My\_Resource\_Group" selected.  
- New Resource Group: A link to "새로 만들기" (Create New) is visible.  
Cluster Details:  
- Kubernetes Cluster Name: "My-Cluster"  
- Region: "(아시아 태평양) 한국 중부"  
- Kubernetes Version: "1.14.8(기본값)"  
- DNS Label: "My-Cluster-dns"  
A red arrow points from the "My\_Resource\_Group" selection in the "Resource Group" section to a callout box containing the following text:  
- "생성한 리소스그룹 선택" (Select the created resource group)  
A red arrow also points from the "My-Cluster" input field in the "Cluster Details" section to another callout box containing:  
- "이름에는 문자, 숫자, 밑줄 및 하이픈만 사용할 수 있습니다. 이름은 문자나 숫자로 시작하고 끝나야 합니다." (Names must consist of letters, numbers, underscores, or hyphens. The name must start and end with a letter or number.)  
- "Kubernetes 서비스 이름은 현재 리소스 그룹에서 고유해야 합니다." (The Kubernetes service name must be unique within the current resource group.)

# Azure Cloud Setup

- Container Registry(컨테이너 레지스트리) 생성

The screenshot shows the Azure portal interface for creating a Container Registry.

**Azure 서비스** (Azure Services) menu items include:

- 리소스 만들기 (Resource creation)
- Kubernetes 서비스
- 구독 (Subscription)
- 리소스 그룹 (Resource Group)
- 컨테이너 레지스트리 (Container Registry)** (highlighted with a red dashed box)
- Azure Active Directory
- SQL 데이터베이스
- 가상 머신 (Virtual Machine)
- 스토리지 계정 (Storage Account)
- 추가 서비스 (Additional Services)

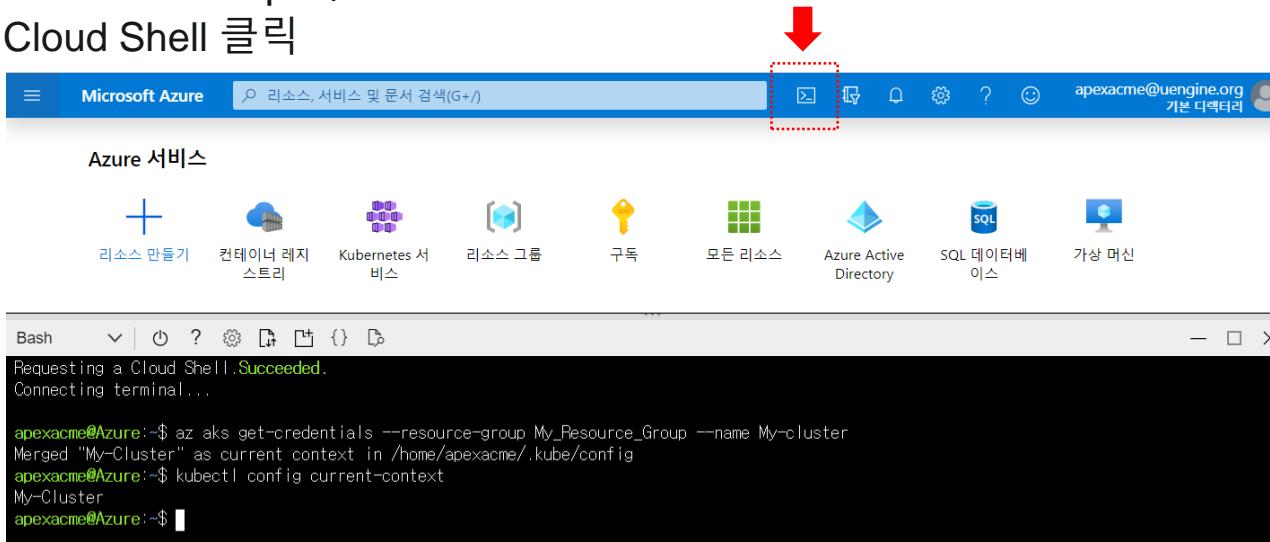
**컨테이너 레지스트리 만들기 (Create Container Registry)** form fields:

- 레지스트리 이름 \*: uengineorg (suffix .azurecr.io)
- 구독 \*: 무료 체험 (Free Trial)
- 리소스 그룹 \*:
  - My\_Resource\_Group (selected)
  - 새로 만들기 (Create New)
- 위치 \*: 한국 중부 (South Korea - Central)
- 관리 사용자 \*:
  - 사용 (Selected)
  - 사용 안 함 (Not Used)
- SKU \*:
  - 표준 (Standard)

A callout bubble points to the "My\_Resource\_Group" dropdown with the text: **생성한 리소스그룹 선택** (Select created resource group).

# Azure Cloud Setup

- Azure Cloud Setup 확인
  - Cloud Shell 클릭



- `az aks get-credentials --resource-group My_Resource_Group --name My-cluster`
- `Kubectl config current-context`

“

# AKS, ACR additional Setup

- Azure CLI and Kubectl install, AKS connection
- Create AKS Credentials on Local Ubuntu
- Azure Container Registry Login
- Integrate AKS with ACR(Private Container Registry)
- Config other useful Utility

# Kubernetes Client Install

- Kubectl 설치
  - sudo apt-get update
  - sudo apt-get install –y kubectl
- 설치 확인
  - kubectl version --client

# Azure-Cli tool Install & Config

- Azure-Cli 설치
  - curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
- Azure Client 인증하기
  - az login # azure Cli authentication



# Azure AKS connection

- Local에서 Azure AKS 연결
  - az aks get-credentials --resource-group *My\_Resource\_Group* --name *My-cluster*
- AKS 연결 확인
  - kubectl config current-context
  - kubectl get all



```
apexacme@APEXACME:~/.kube$ kubectl get all
NAME           TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   Cluster IP  10.0.0.1    <none>        443/TCP  95m
apexacme@APEXACME:~/.kube$
```

Default kubernetes service 확인

# Login Azure Container Registry

- Azure CLI
  - az acr login -- name *uengineorg*



```
apexacme@APEXACME:~/yaml$ az acr login --name uengineorg
Login Succeeded
WARNING! Your password will be stored unencrypted in /home/apexacme/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store
```

- 생성확인
  - cat ~/.docker/config.json



```
[{"auths": {"https://index.docker.io/v1/": {"auth": "YXB1eGFjbWU6c2pkem5mMjEyNw=="}, "uengineorg.azurecr.io": {"auth": "MDAwMDAwMDAwMC0wMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwOg==", "identitytoken": "eyJhbGciOiJSUzI1NiIsInR5cC161kpXVCiS1mtzC161jZRRjI6RkRNNTpSUiQ2OkE3WFg6N0Q3UjpFNE10kZJMzQ6RENVUzpQR1d0lPSj6SktaSpStKkyln0_euyJqdGk1oJkNDkxTmUyMS1iZjFLTx0NyYtMYS1HNdDjM2UwZDYyYjciLzdlWI0iJsaXZ1LmNbSNhcqV4WNtZUBtZM5naH51Lm9vZyhls1m51Zj16MTU4MtgyM1MxMiwiZkhwiJoxNTx0DMDExLCJpXX0j0TE10DE4mjzLzMTlsImLzcy161kF6dxJ11ENvbRhaw5Ic1BSZ2Wdc3RveS1sImf1ZC161nVlbnppbmVcmcuXp1cmVjci5pbys1sInZ1cnNb2410iXlJAiLCJncmfudF90eXB1jjo1cmVmcmVzaF90b2t1b1ls1nR1bmFudC161jA10TYzM2M-LWEyN1t1NDL5Yi5Mzc4LTgxOGJnND11NDN1NC1sInB1cm1pc3Npb25z1j7ImF-dg1vbnM01si1cmVhZC1sIndyaXR1Iw1ZGVsZkR1170sIm9vdFEjDg1vbnM1Um51bgx9LcJybz1cy16W119.19Tmgxbc/Ms0fe1aC9yz_JuiPb_BUwLxSqk-ApaB02fkw_zT8sZoNDwc5upr1D4FimF1dy_msy7pdPoMdhwS3PDupw-ekhAzWf1xW01fvcpk2jE05ae6WS2600uD0JBqLsPNTPKYmK1cxBaPp9GWzNeAACTeoByYUQqgeNHawN0nkqQzy2APLNx4G4[[KwqwfNiLwBMEW31-xT88b12c0xte_ypwMx4g5hukvvgFo2IwGjJcb-0TKPhEJykr1o4x74A120kTGnzR8_RHzCnHEHYKBQYhhsRjZ1bnbhzmWSrcEpyMK92ym1jrhjklSkgtLLzAfDfAkz71KUydw"}, "HttpHeaders": {"User-Agent": "Docker-Client/18.09.7 (linux)"}]}
```

# Integrate AKS with ACR

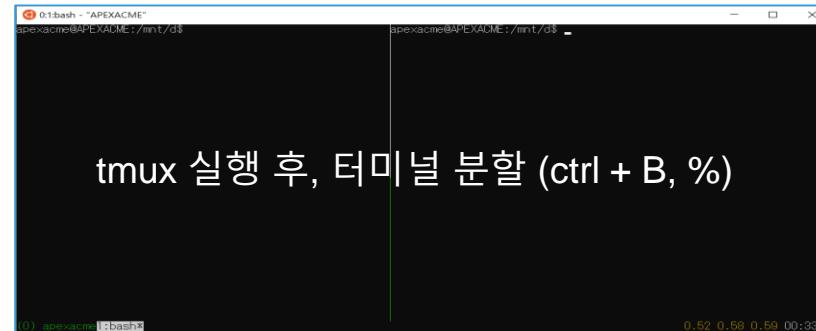
- Azure CLI

- `az aks update -n My-cluster -g My_Resource_Group --attach-acr uengineorg`



# Config other useful Utility (skip)

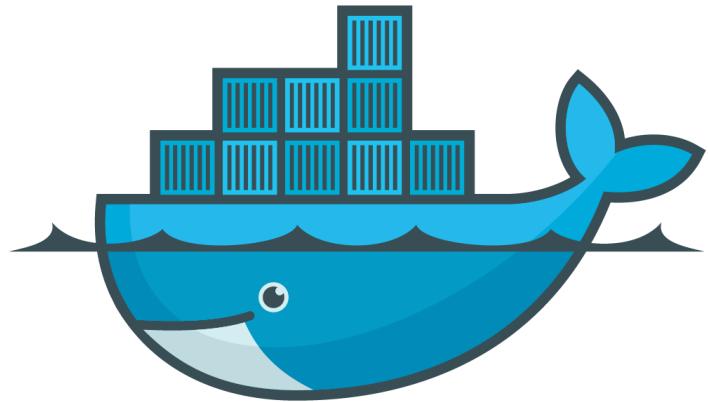
- 좀 더 효율적인 Kubernetes 실습을 위한 스크린 분할(Screen split) 환경 구축
  - ex) kubectl 오퍼레이션에 따른 Pod 인스턴스 동시 모니터링
  - ex) Load Stress 상황과 Pod의 실시간 Scale Out/In 변화량 멀티 뷰
- ‘tmux’ tool upgrade
  - Github에서 .tmux.conf 파일 C:\ 다운로드
  - Linux Home 디렉토리로 복사
    - cp /mnt/c/.tmux.conf ~/
- Upgrade 확인



# Table of content

Container Orchestration  
(Docker & k8s)

1. MSA, Container, and Container Orchestration
2. Setup Azure Platform
3. Docker / Kubernetes, Kubernetes Architecture ✓
4. Kubernetes Basic Object Model : Pod, Label, ReplicaSet, Deployment, Service, Volume, Configmap, Secret, Liveness/Readiness
5. Kubernetes Advanced Lab : Ingress, Job, Cron Job, DaemonSet, StatefulSet
6. Service Mesh: Istio for Advanced Services Control
7. Course Test

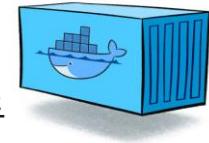


# docker

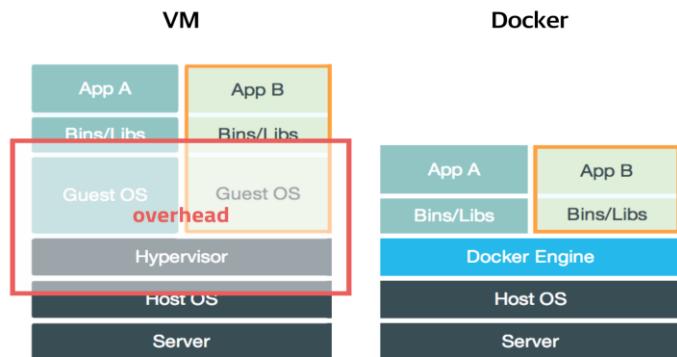
- 2013년 3월,  
dotCloud 창업자 Solomon Hykes가 Pycon Conference에서 발표
- Go 언어로 작성된 “The future of linux Containers”

- Container based

- 프로세스 격리 기술
- 오픈소스 가상화 플랫폼



- 가상머신 .vs. Docker



# Docker Image & Container

- Docker Image
  - 가상머신 생성시 사용하는 ISO 와 유사한 개념의 이미지
  - 여러 개의 층으로 된 바이너리 파일로 존재
  - 컨테이너 생성시 읽기 전용으로 사용
  - 도커 명령어로 레지스트리로부터 다운로드 가능

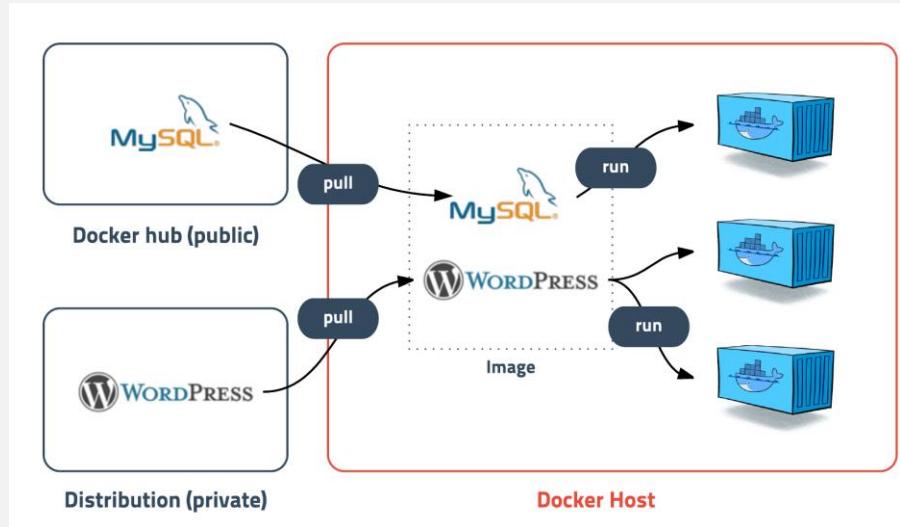


- 저장소 이름 : 이미지가 저장된 장소, 이름이 없으면 도커 허브(Docker Hub)로 인식
- 이미지 이름 : 이미지 이름, 생략 불가
- 이미지 버전 : 이미지 버전정보, 생략 시 latest로 인식

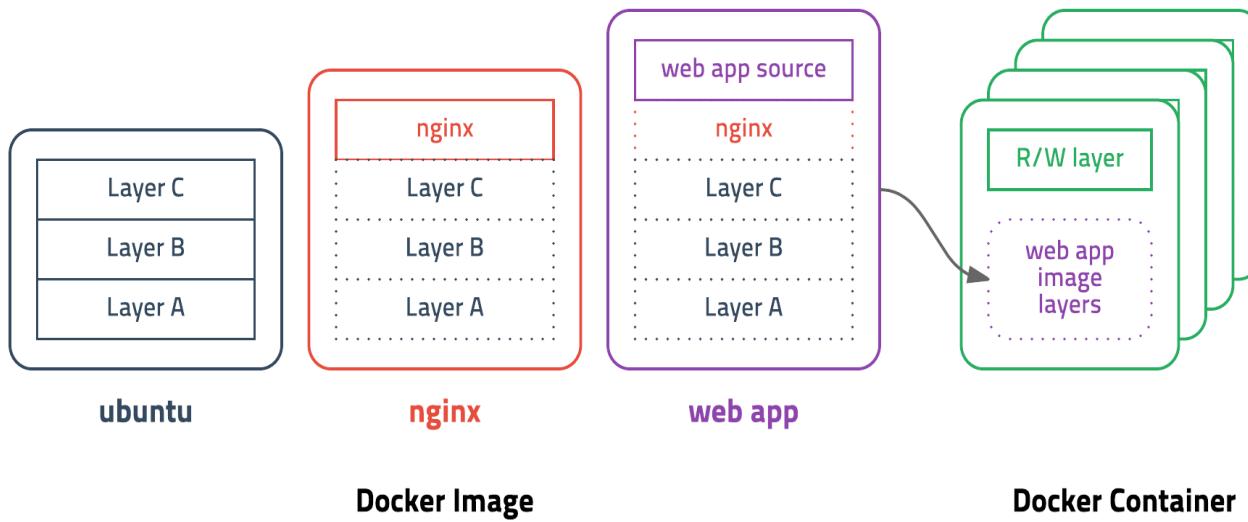
# Docker Image & Container

- Docker Container

- 도커 이미지로 부터 생성
- 격리된 파일시스템, 시스템 자원, 네트워크를 사용할 수 있는 독립공간 생성
- 이미지를 읽기 전용으로 사용, 이미지 변경 데이터는 컨테이너 계층에 저장

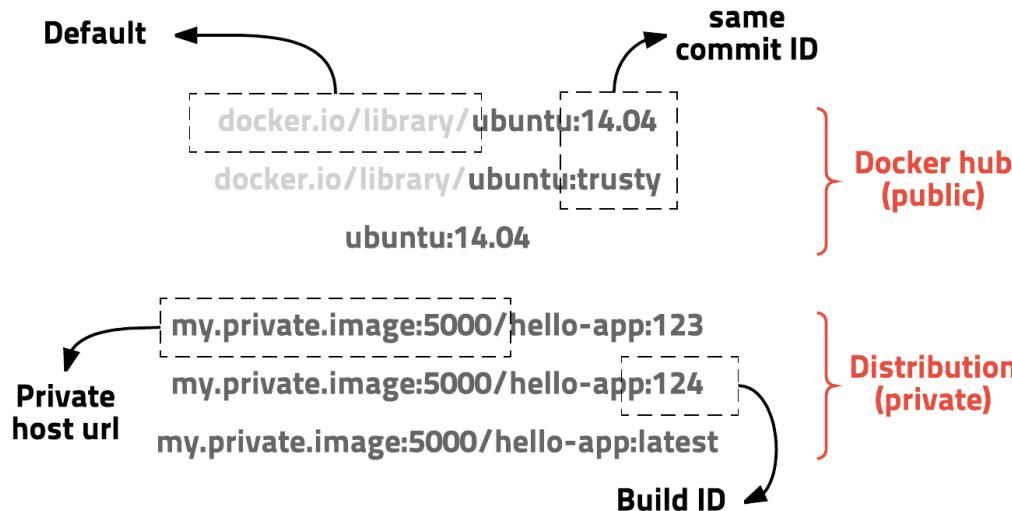


# Layered Architecture



- Image : 여러 개의 읽기 전용(Read Only) 레이어로 구성
- Container : Image 위에 R/W 레이어를 두고, 실행 중 생성 또는 변경 내용 저장

# Docker Image Path



- 이미지 Path는 <URL>/<namespace>/<Image\_name>:<tag> 형식
- library는 도커허브 공식 이미지 Namespace로, 여기에 사용자 이름이 위치

# Dockerfile sample

```
FROM openjdk:8-jdk-alpine
```

```
RUN apk --no-cache add tzdata && cp  
/usr/share/zoneinfo/Asia/Seoul /etc/localtime
```

```
WORKDIR /app  
COPY hello.jar hello.jar  
COPY entrypoint.sh run.sh  
RUN chmod 774 run.sh
```

```
ENV PROFILE=local
```

```
ENTRYPOINT ["./run.sh"]
```

- **FROM** : 이미지를 생성할 때 사용할 기반 이미지를 지정한다.
- **RUN** : 이미지를 생성할 때 실행할 코드 지정한다. 예제에서는 패키지를 설치하고 파일 권한을 변경하기 위해 RUN을 사용
- **WORKDIR** : 작업 디렉토리를 지정한다. 해당 디렉토리가 없으면 새로 생성한다. 작업 디렉토리를 지정하면 그 이후 명령어는 해당 디렉토리를 기준으로 동작
- **COPY** : 파일이나 폴더를 이미지에 복사한다. WORKDIR로 지정한 디렉토리를 기준으로 복사
- **ENV** : 이미지에서 사용할 환경 변수 값을 지정한다. 컨테이너를 생성할 때 PROFILE 환경 변수를 따로 지정하지 않으면 local을 기본 값으로 사용
- **ENTRYPOINT** : 컨테이너를 구동할 때 실행할 명령어를 지정한다.

# Docker Image Commands

- 도커 이미지 목록 확인
  - \$ docker images
- 도커 이미지 불러오기
  - 컨테이너 run 시에 이미지가 없으면 Docker Hub로부터 자동으로 Pull
  - \$ docker pull [ImageName:태그]
- 도커 이미지 삭제
  - docker image rm [ 이미지 ID ]
  - docker image rm -f [ 이미지 ID ] # 컨테이너를 삭제하기 전에 이미지 삭제
- 도커 모든 이미지 한 번에 삭제
  - \$ docker image rm \$(docker images -q)

# Lab. Docker Image



Lab Time

- Lab Script Location
  - Workflowy :

# Docker Container Commands

- 컨테이너 실행
  - \$ docker run [Options] [Image] [Command]
- 실행 중인 컨테이너 확인
  - \$ docker ps
  - \$ docker ps -a # 정지된 컨테이너 포함
- 컨테이너 시작, 재시작, 종료
  - \$ docker start / restart / stop [ 컨테이너 이름 ]
- 컨테이너 삭제
  - \$ docker container rm [ 컨테이너 ID ]
- 모든 컨테이너 한번에 삭제 (중지 후 삭제)
  - \$ docker container rm \$(docker ps -a -q)

# Lab. Docker Container



Lab Time

- Lab Script Location
  - Workflowy :

# Docker Build & Push Commands

- Dockerfile로 이미지 생성
  - docker build --tag [ 생성할 이미지 이름 ]:[ 태그 이름 ].
  - # 맨 마지막의 .(마침표)은 Dockerfile의 위치
  - 이미지 이름은 URL(Docker hub, Cloud Container registry, Private registry)로 시작
- 이미지 Push
  - \$ docker login # 도커 로그인
  - \$ docker push [ 이미지 REPOSITORY ]:[ 태그 ]
- Docker Hub에서 확인
  - <http://hub.docker.com> (login)

# Lab. Docker Build & Push

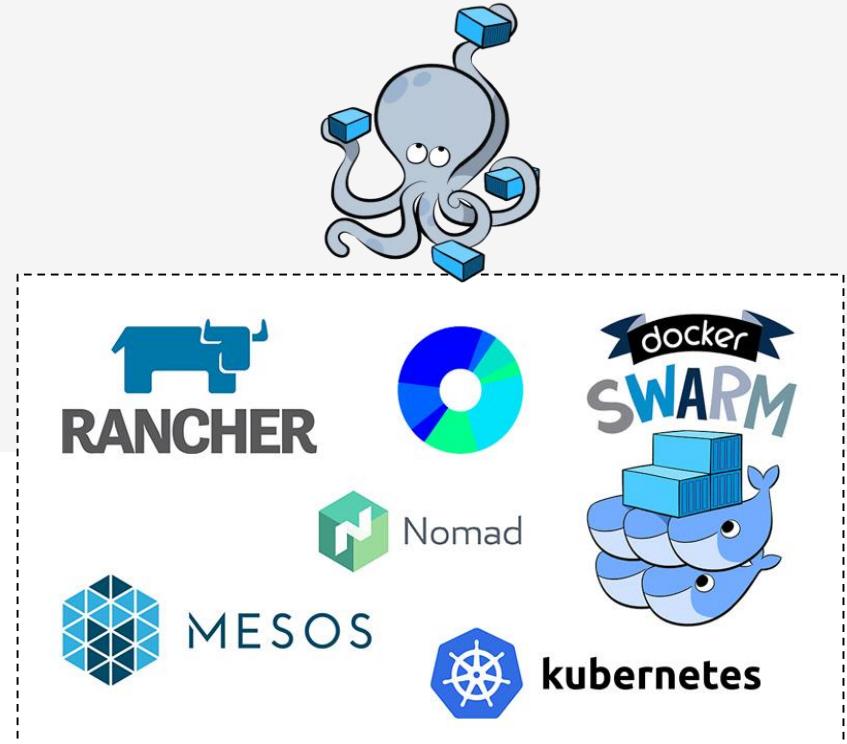


Lab Time

- Lab Script Location
  - Workflowy :

# Container Orchestration Features

- 컨테이너 자동 배치 및 복제
- 컨테이너 그룹에 대한 로드 밸런싱
- 컨테이너 장애 복구
- 클러스터 외부에 서비스 노출
- 컨테이너 확장 및 축소
- 컨테이너 서비스간 인터페이스를 통한 연결



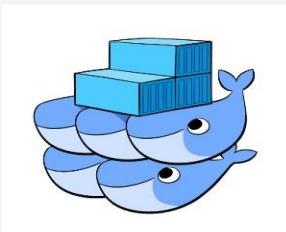
# Container Orchestrators

Kubernetes



구글에서 개발, 가장 기능이 풍부하며  
널리 사용되는 오케스트레이션 프레임워크  
베어 메탈, VM환경, 퍼블릭 클라우드 등의  
다양한 환경에서 작동하도록 설계  
컨테이너의 롤링 업그레이드 지원

Docker Swarm



여러 개의 Docker 호스트를 함께 클러스터링  
하여 단일 가상 Docker 호스트를 생성  
호스트 OS에 Agent만 설치하면 간단하게 작동  
하고 설정이 용이  
Docker 명령어와 Compose를 그대로 사용 가능

Apache Mesos



수만 대의 물리적 시스템으로 확장할 수  
있도록 설계  
Hadoop, MPI, Hypertable, Spark 같은 응용  
프로그램을 동적 클러스터 환경에서 리소  
스 공유와 분리를 통해 자원 최적화 가능  
Docker 컨테이너를 적극적으로 지원

# Kubernetes

*“Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.”*

**“Kubernetes**는 컨테이너화된 어플리케이션을  
자동으로 배포, 조정, 관리할 수 있는 오픈소스 플랫폼이다.

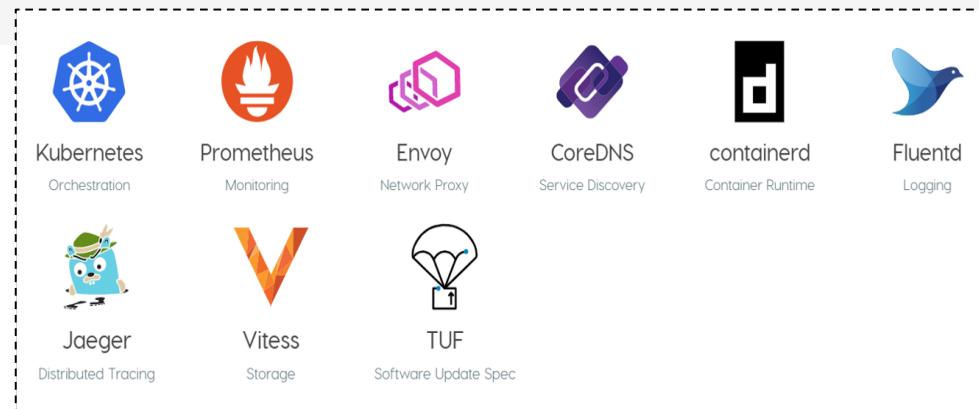
- From Kubernetes Website

# Kubernetes Origin

- Borg System 영향을 받아, 2014년 구글의 의해 처음 발표
- 2015년 7월 21일, v1.0 출시
- 리눅스재단과 Cloud Native Computing Foundation(CNCF) 설립
- Kubernetes를 seed 테크놀로지(seed technology)로 제공



<https://www.cncf.io/>



# Kubernetes key Features (1/2)

- **Automatic binpacking**
  - 각 컨테이너가 필요로 하는 CPU와 메모리(RAM)를 쿠버네티스에게 요청하면, 쿠버네티스는 컨테이너를 노드에 맞추어서 자동으로 스케줄링
- **Self-healing**
  - Kubernetes는 실패한 노드에 대해, 컨테이너를 자동으로 교체하고, 재 스케줄링하며, 또한 Health check에 반응하지 않는 컨테이너를 정해진 규칙에 따라 다시 시작
- **Horizontal scaling**
  - Kubernetes는 CPU 및 메모리와 같은 리소스 사용량을 기반으로 애플리케이션을 자동으로 확장 할 수 있으며, 메트릭을 기반으로 하는 동적 스케일링도 지원
- **Service discovery and load balancing**
  - Service Discovery 매커니즘을 위해 Application을 수정할 필요가 없으며, K8s는 내부적으로 Pod에 고유 IP, 단일 DNS를 제공하고 이를 이용해 load balancing

# Kubernetes key Features (2/2)

- **Automated rollouts and rollbacks**

- Application, Configuration의 변경이 있을 경우 전체 인스턴스의 중단이 아닌 점진적으로 Container에 적용(rolling update) 가능
- Release revision이 관리되고 새로운 버전의 배포시점에 문제가 발생할 경우, 자동으로 이전의 상태로 Rollback 진행

- **Secret and configuration management**

- Kubernetes는 secrets와 Config 정보에 대해 이미지 재빌드없이 변경관리가 가능하고, Github 같은 저장소에 노출시키지 않고도 어플리케이션 내에서 보안정보 공유 가능

- **Storage Orchestration**

- Local storage를 비롯해서 Public Cloud(Azure, GCP, AWS), Network storage 등을 구미에 맞게 자동 mount 가능

# Kubernetes: 발음하기

미국식: 큐브네리스

영국식: 쿠버네티스

인도식: 꾸-버네딕스

# Kubernetes: 쓰기

Kubernetes



K8S ≠

KBS 

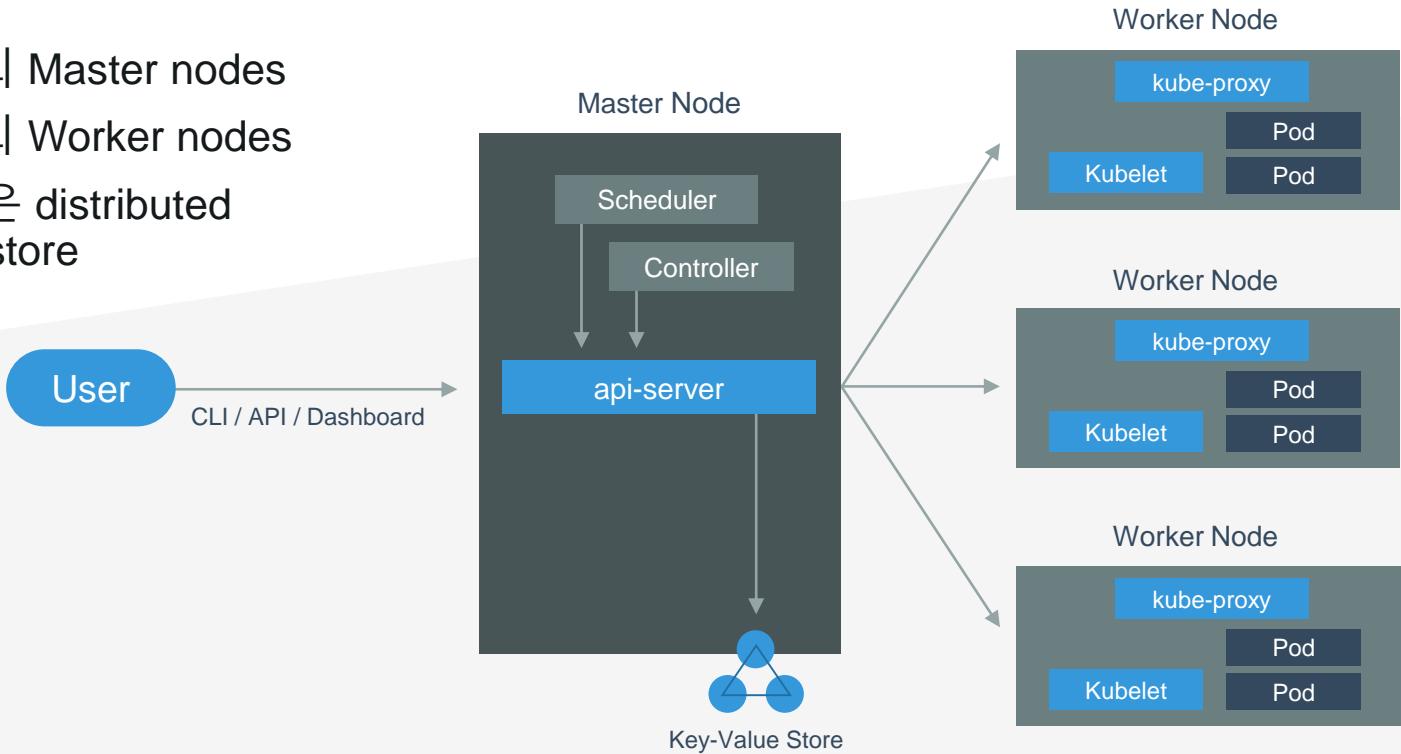
“

# Kubernetes Architecture

- Master Node, Worker Node
- Architecture Components

# Kubernetes Architecture

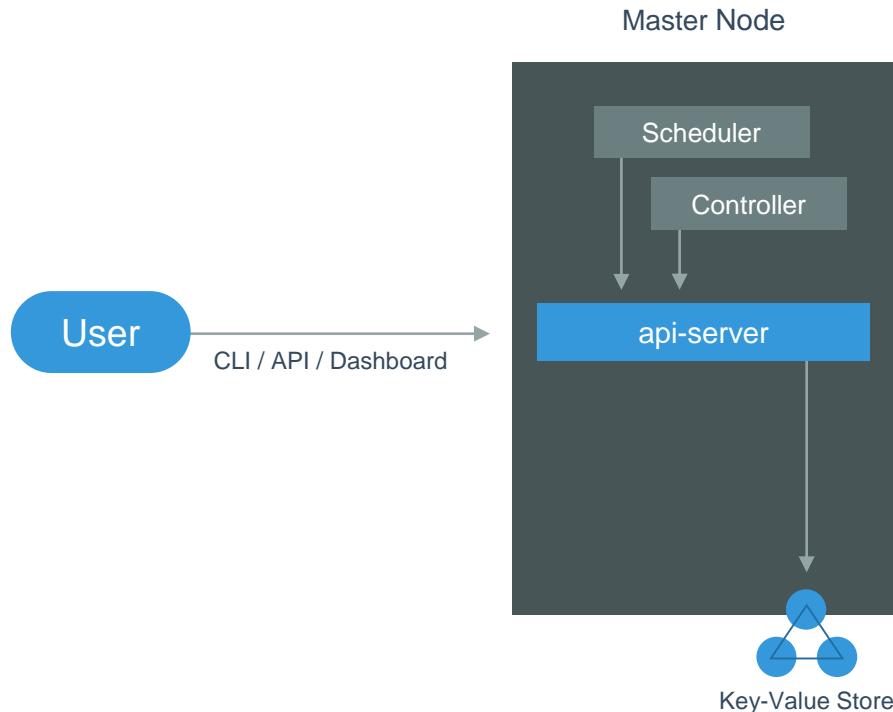
- 하나 이상의 Master nodes
- 하나 이상의 Worker nodes
- etcd 와 같은 distributed key-value store



# Master Node

- Master node는 Kubernetes 클러스터를 관리하며, 모든 관리자 업무의 시작점
- 다양한 모듈이 확장성을 고려하여 기능별로 분리
- CLI, GUI (Dashboard), APIs 등으로 Master node에 접근 가능

# Master Node 구성요소



- API server
- Scheduler
- Controller manager
- Etcd. (distributed key-value store)

- ✓ 장애 대비하여, 하나 이상의 Master node를 한 클러스터에 구성하여 안정성을 높임
- ✓ 하나 이상의 Master node 가 존재할 경우, HA (High Availability)모드로 그 중 하나가 리더(leader)로써 작동하고 나머지는 팔로워(followers)로 운영

# Master Node 구성요소 :

## API Server

- 원하는 상태를 Key-Value 저장소에 저장하고 저장된 상태를 조회하는 작업 수행
- 모든 administrative tasks는 master node에 있는 API server를 통해 수행
- 사용자/운영자가 보낸 요청은 API server를 통해 처리되고, 실행 결과는 분산 키-값 저장소(Distributed key-value store)에 저장

# Master Node 구성요소 : Scheduler

- Pod를 여러가지 조건(필요한 자원, 라벨)에 따라 적절한 노드에 할당해 주는 모듈
- Scheduler가 개별 worker node에게 업무를 분담
- Scheduler는 각 worker node의 리소스 사용 정보를 가짐
- “disk==ssd” 와 같은 label이 설정된 worker node에 업무 부여하는 등 사용자/운영자가 설정한 제약에 대해 인지하고 있음
- Scheduler는 Pods 와 Services 단위로 업무 수행

# Master Node 구성요소 : Controller Manager

- Kubernetes Cluster에 있는 모든 오브젝트의 상태를 관리하는 모듈
- Controller manager는 Kubernetes Cluster의 상태를 조절하는 non-terminating control loops 관리
- 각각의 control loops는 자신이 관리하는 객체의 desired state를 알고 그 객체들의 현 상태를 API 서버를 통해서 모니터링
- 제어 루프에서 관리하는 객체의 현 상태가 원하는 상태와 다르면, 제어 루프는 수정 단계를 수행하여 현 상태와 원하는 상태를 일치시키는 작업 수행
- Kubernetes release 1.6, kube-controller-manager와 cloud-controller-manager로 세분

# Master Node 구성요소 :

## Etcd

- RAFT\* 알고리즘을 이용한 클러스터 상태를 저장하는 key-value 저장소
- 단순 R/W 기능 외, Watch 기능이 있어, 상태가 변경되면 트리거 로직 실행 가능
- 클러스터의 모든 설정, 상태가 저장되므로 Cluster Backup / Restore 유리
- Etcd는 오직 API 서버와 통신하고, 다른 모듈은 API 서버를 통해 etcd 접근

\* RAFT 알고리즘: 분산 환경에서 상태를 공유하는 알고리즘, consensus algorithm(Paxos, RAFT)

# Master Node 구성요소 :

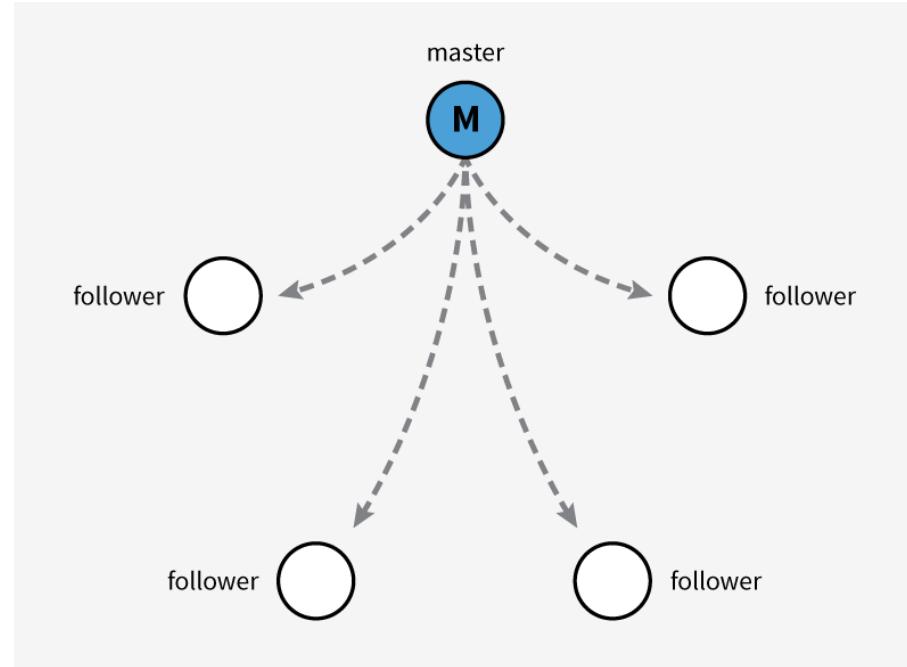
## Etcd : State 관리

- Kubernetes는 etcd를 사용해 클러스터의 상태를 저장
- etcd는 ‘Raft Consensus Algorithm’을 기반으로 하는 distributed key-value store
  - Raft는 장치의 집합으로써 몇몇 구성원에 장애가 발생하여도 살아남아 일관적인 집단으로 작동할 수 있도록 해 줌
- etcd는 Go 프로그래밍 언어로 작성
- Kubernetes에서는 클러스터의 상태를 저장하는 기능 말고도 Subnets, ConfigMaps, Secrets 등을 저장하기도 함

# Master Node 구성요소 :

## Etc : State 관리

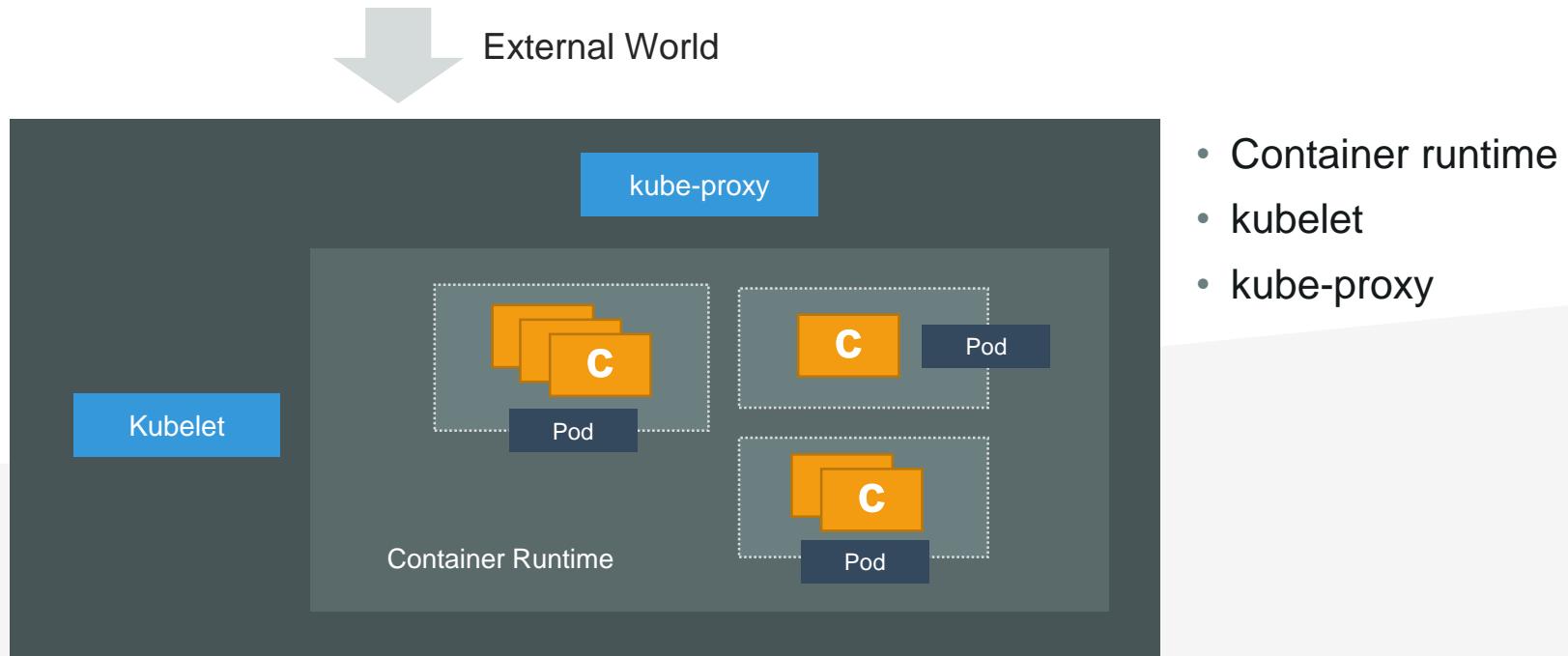
- 그룹의 노드 중 하나는 Master로 작동을 하고, 나머지는 Follower로 작동
  - 모든 노드는 Master 후보



# Worker Node

- 마스터 서버와 통신하면서 필요한 Pod를 생성하고 네트워크와 볼륨을 설정
- Worker node는 Master node에 의해 관리되며, Pod를 이용 어플리케이션을 실행
- Pod는 Kubernetes의 스케줄링 단위
- 하나 이상의 컨테이너의 논리적 집합이며, 항상 같이 스케줄링 됨

# Worker Node 구성요소



# Worker Node 구성요소 : Container Runtime

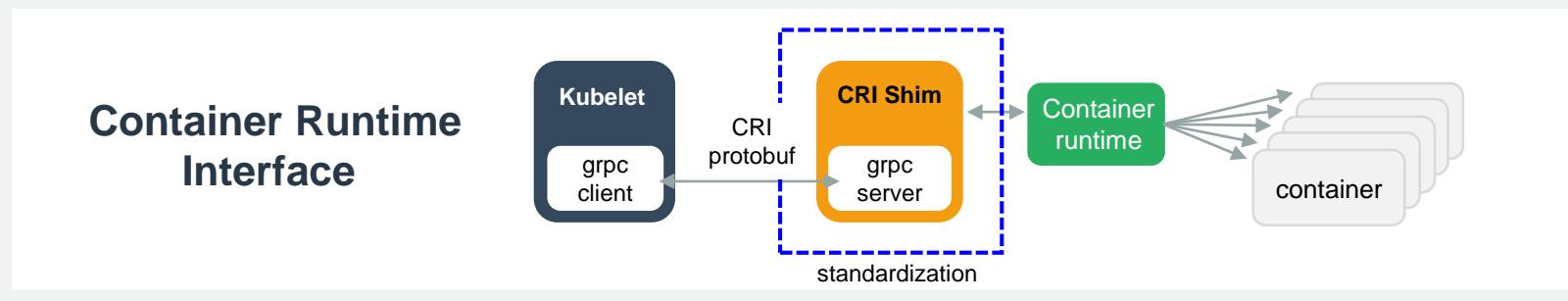
- 컨테이너의 lifecycle을 관리/실행 하기위한 Worker node상의 Container runtime
  - (runtimes: containerd, cri-o, RKT, LXD)
    - ✓ 가끔, Docker가 Container runtime으로 언급되는 경우가 있다.
    - ✓ 정확하게는 Docker는 플랫폼이며, **containerd**를 Container runtime으로 사용한다.

# Worker Node 구성요소 :

## Kubelet

- Node에 할당된 Pod의 생명 주기를 관리
- kubelet은 master node와 통신하기 위해 worker node에서 작동하는 에이전트
- Pod 설정을 수신하고 해당 Pod와 관련된 컨테이너를 실행
- 항상 컨테이너가 정상 작동하는지 확인
- Kubelet은 Container Runtime Interface(CRI)를 이용하여 컨테이너 런타임에 연결
  - CRI는 protocol buffers, gRPC API, and libraries 등을 포함

# Worker Node 구성요소 : Kubelet



- Kubelet이 명령을 받으면 Docker runtime을 통해 Container를 생성하거나 삭제
- Docker이외의 여러 컨테이너 기술이 나오면서 매번 Kubelet 코드를 수정하는 번거로움에 따라, Kubelet과 Container runtime 사이에 표준 인터페이스가 제정되었는데, 이를 CRI Shim이라 함
- 새로운 Container runtime은 CRI Shim스펙에 맞춰 CRI 컴포넌트를 구현

\* gRPC : Google에서 처음 개발한 공개 원격 프로시저 호출(RPC) 시스템, 인터페이스 설명언어로 프로토콜 버퍼 사용

\* Protocol buffer 줄임말로 크기를 줄인 직렬화 데이터 구조(이진 메시지 형식)

# Worker Node 구성요소 :

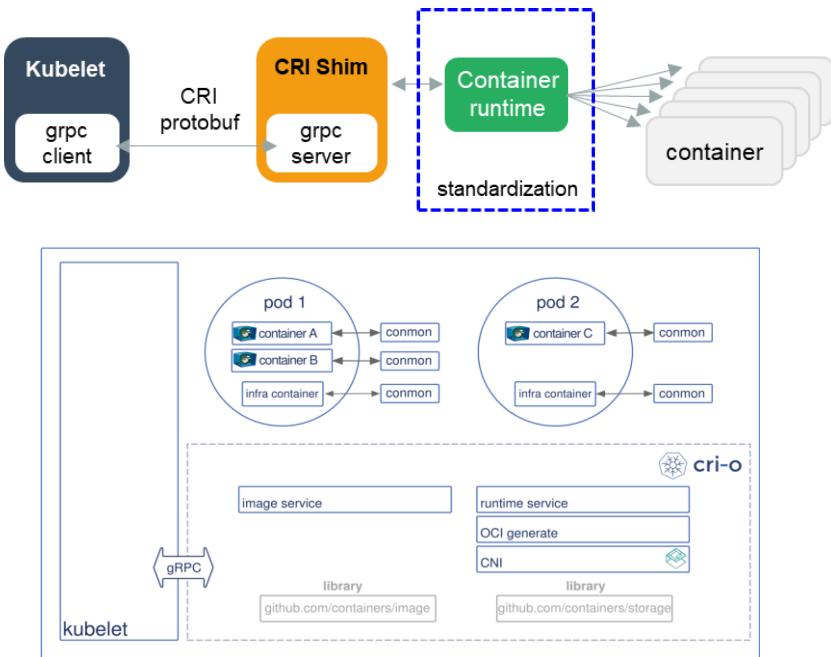
## Kubelet : CRI Shims



- **Docker shim**  
Docker shim은 Worker node에 설치된 Docker를 이용하여 컨테이너를 생성
- 내부적으로는 Docker는 containerd를 이용해 컨테이너를 관리

# Worker Node 구성요소 :

## Kubelet : CRI-O



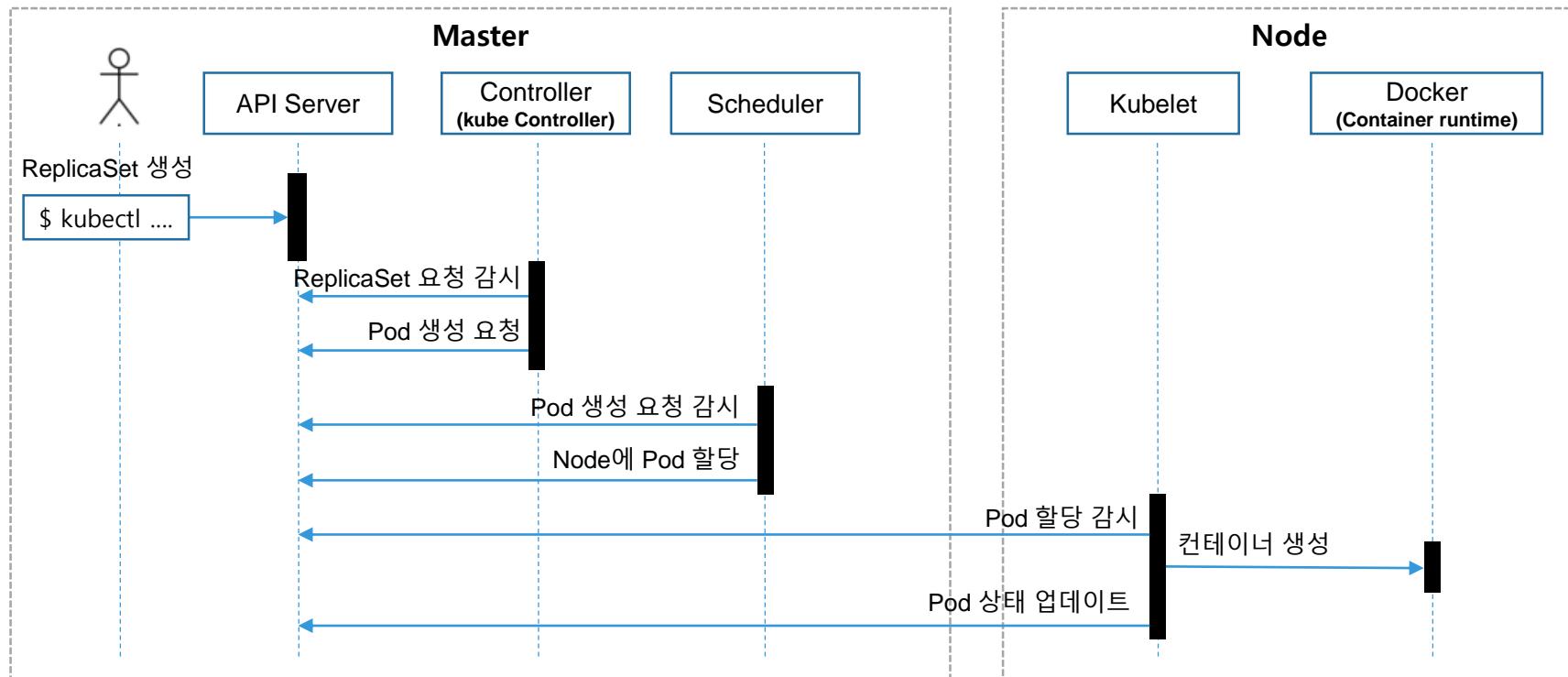
- Container Runtime 표준화
- 각 Container마다 CRI Shim을 개발해야 하는 비용적 이슈에서 CRI-O 등장
- Container runtime이 OCI(Open Container Initiative)에서 정의한 표준을 따르면 CRI-O를 CRI shim으로 사용 가능

# Worker Node 구성요소 :

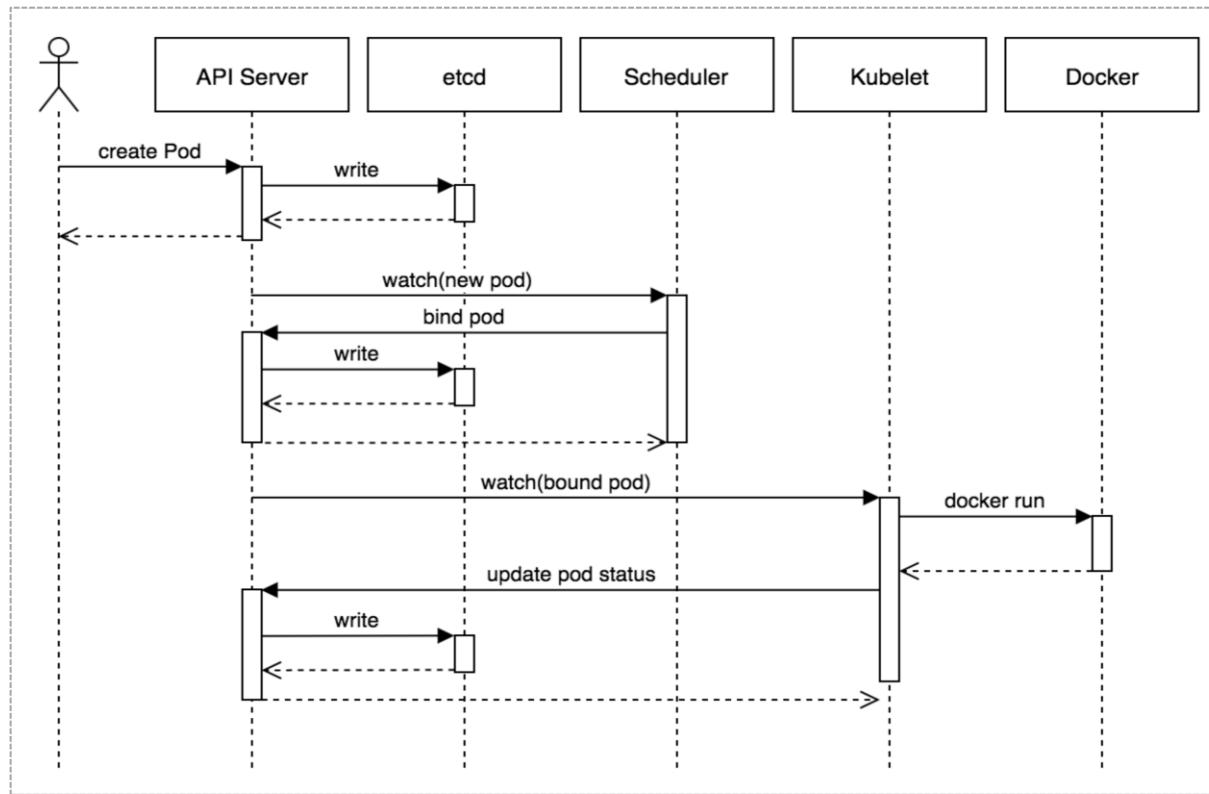
## Kube-proxy

- Kubelet이 Pod를 관리한다면, Proxy는 Pod로 연결되는 네트워크를 관리
- 초기, kube-proxy가 프록시 서버로 동작하면서 실제 요청을 받아 각 Pod로 포워딩
  - 성능 이슈로 iptables를 설정하는 방식으로 변경, 최근엔 IPVS 지원 시작
- 애플리케이션에 액세스하기 위해 포드에 직접 연결하는 대신 "서비스"라는 논리적 구성을 연결 엔드포인트로 사용
  - Service 그룹과 관련된 Pod에 접속 시 자동으로 load balancing 됨
- kube-proxy는 Network proxy로서 각각의 Worker node에서 작동하며, 각각의 Service endpoint를 생성/삭제하기 위해 API server를 지속적으로 Listening
  - kube-proxy는 각 Service endpoint에 접근할 수 있는 경로 설정

# Pod Creation Process



# Pod Creation Process Including etcd



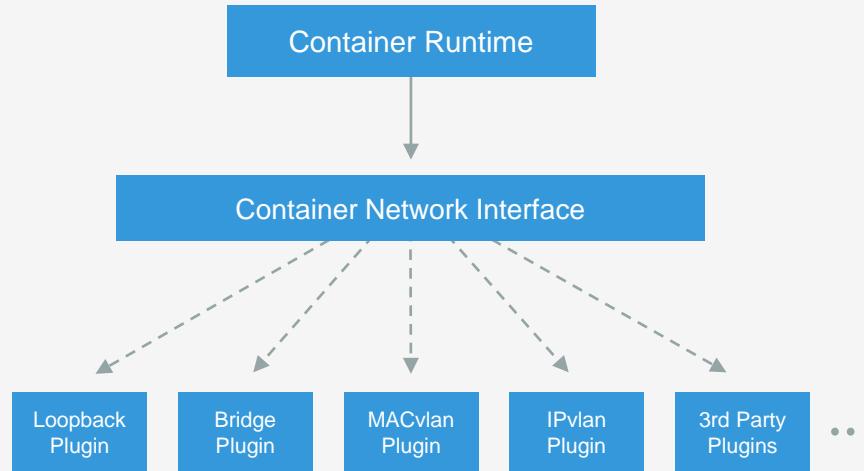
# Kubernetes Network Prerequisites

- 완벽하게 작동하는 Kubernetes cluster를 만들기 위한,
- **Kubernetes Network Prerequisites :**
  - 각각의 Pod에는 중복이 없는 IP가 배정되어야 한다.
  - Pod에 속해 있는 Container들은 서로 통신해야 한다.
  - Pod는 다른 클러스터에 있는 Pod들과 서로 통신할 수 있어야 한다.
  - Pod 안에 배포된 어플리케이션은 외부에서 접근이 가능해야 한다.

# 각 Pod에게 고유 IP 부여하기

- Kubernetes에서 각 Pod는 고유의 IP주소를 할당 받아야 함
- 대표적 컨테이너 네트워킹 솔루션
  - Container Network Model (CNM), by Docker
    - Docker Container runtime만 지원
    - 네트워크 구성 저장을 위한 분산 key-value 저장소 필요
  - Container Network Interface (CNI), by CoreOS
    - 모든 Container runtime 지원
    - CNCF에서 Container 네트워킹의 표준으로 인증
    - 분산 key-value 저장소 필요 없음
- Kubernetes는 CNI를 사용

# 각 Pod에게 고유 IP 부여하기



- Container runtime은 IP 할당을 CNI로 오프로드하며, CNI는 Bridge 또는 MACvlan과 같은 Built-in Plugin에 연결하여 IP 주소를 Fetch
- 플러그인을 통해 IP 주소가 배정되면, CNI는 요청한 Container runtime에게 값을 리턴

# Pod 내에서 Container간 통신하기

- 기본 호스트 운영 체제의 도움으로 모든 Container runtime은 시작하는 각 컨테이너에 대해 격리된 네트워크 엔티티를 생성
  - 리눅스에서 Entity는 network namespace을 가르키며, 이 network namespaces는 Host operating system과 컨테이너 간 공유가 가능
- Pod 안에 있는 컨테이너들은 다른 localhost에 접근하기 위해 네트워크 namespace를 공유

# 노드간 Pod 통신하기

- 클러스터 환경에서는 모든 노드 상에 Pod가 스케줄링
- Pod들은 Node간 통신이 가능해야 하고, 모든 Node는 아무 Pod에 접근 가능 해야 함
- 서로 다른 호스트의 Pod간 통신에는 Network Address Translation(NAT)이 없어야 한다는 조건을 추가로 제시
  - Flannel, Weave, Calico 등의 소프트웨어로 설정이 가능한 네트워킹을 사용

# 외부에서 Pod 접근하기

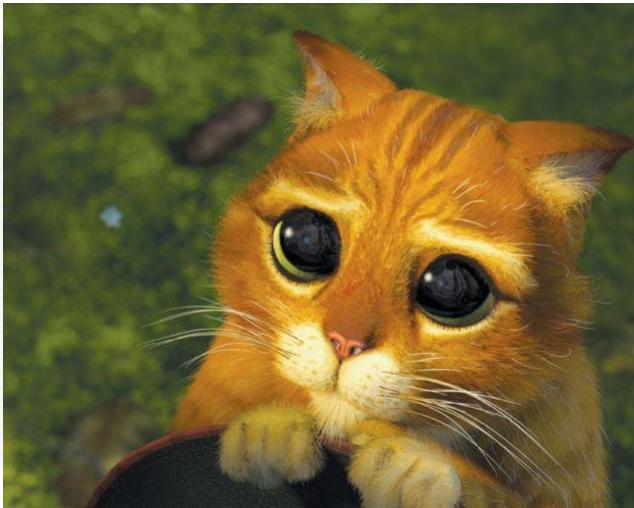
- Kube-proxy를 이용하여 서비스를 노출시키면,  
클러스터 밖에서 해당 어플리케이션 접근 가능

# Table of content

Container Orchestration  
(Docker & k8s)

1. MSA, Container, and Container Orchestration
2. Setup Azure Platform
3. Docker / Kubernetes, Kubernetes Architecture
4. Kubernetes Basic Object Model : Pod, Label, ReplicaSet, Deployment, Service, Volume, Configmap, Secret, Liveness/Readiness ✓
5. Kubernetes Advanced Lab : Ingress, Job, Cron Job, DaemonSet, StatefulSet
6. Service Mesh: Istio for Advanced Services Control
7. Course Test

# Kubernetes Core Concept : “Desired State”



Declarative Model & Desired States

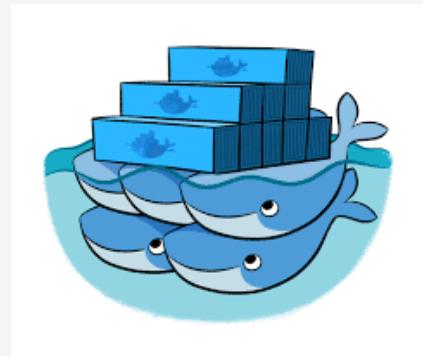
- Kubernetes는 Current State을 모니터링하면서, Desired State를 유지하려는 습성
- 직접적인 동작을 명령하지 않고, 원하는 상태를 선언(Not Imperative, But Declarative)
  - Imperative – “nginx 컨테이너를 3개 실행해줘, 그리고 80포트로 오픈해줘.”
  - Declarative – “80포트를 오픈한 채로, nginx 컨테이너를 3개 유지해줘.”

# Kubernetes Object, Controller and Kubectl

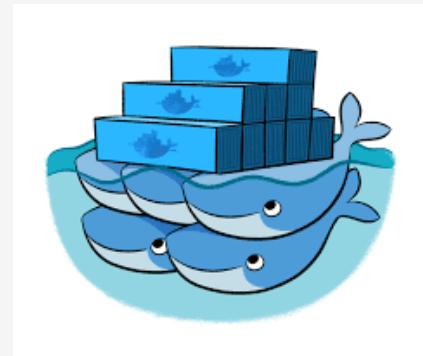
- Object : K8s의 상태를 나타내는 엔티티로 K8s API의 Endpoint
  - 유형 - Pod, Service, Volume, Namespace 등
  - Spec과 Status 필드를 가짐 - Spec(Desired State), Status(Current State)
- Controller : Object의 Status를 갱신하고, Object를 Spec에 정의된 상태로 지속적으로 변화시켜 주는 주체
  - 유형 – ReplicaSet, Deployment, StatefulSets, DaemonSet, Cronjob 등
- Kubectl : Command CLI에서 Object와 Controller를 제어하는 K8s Client
  - 발음하기 – “큐브시티엘”, “쿠베시티엘”

# Pod ; Kubernetes 최소 배포 단위

- Pod : 미국식 [pa:d], 영국식 [pɒd]
  - “물고기, 고래” 작은 떼 (Docker의 심볼이 고래 모양에서 유래)
  - 발음하기 : “팟”, “파드”, “포드”



Pod



Pod

# Pod ; Kubernetes 최소 배포 단위

- 한 Pod은 내부에 여러 컨테이너를 가질 수 있지만 대부분 1~2개의 컨테이너 가짐
  - 스케일링이 컨테이너가 아닌 Pod단위로 수행되기 때문에, Pod내부에 다수의 컨테이너들이 타이트하게 묶여 있으면 스케일링이 쉽지 않거나 비효율적으로 이뤄지는 문제
- 1개의 Pod은 1개의 물리서버(Node) 위에서 실행
- 동일 작업을 수행하는 Pod은 ReplicaSet Controller에 의해 룰에 따라 복제생성
  - 이때 복수의 Pod이 Master의 Scheduler에 의해 여러 개의 Node에 걸쳐 실행
- Pod의 외부에서는 이 'Service' 를 통해 Pod에 접근
  - Service를 Pod에 연결했을 때 Pod의 특정 포트가 외부로 expose

# Kubernetes Object Model

http://serviceld:8080

Micro-Service

Service  
(name: foo)

http://external\_ip:8080

Load Balancer from  
cloud provider

Redirects

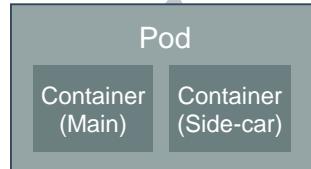
Deployment

Creates / Manages

ReplicaSet  
(Green)

ReplicaSet  
(Blue)

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



## 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

- Keep replica count as desired  
(replicas=2)

- Service Hosting

# 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

# Declaration based configuration

> my-app.yml

> Desired state



```
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
```

```
---
```

```
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
```

```
---
```

```
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
```

# Lab. K8s Sample App 생성 – Voting App



Lab Time

- Lab Script Location
  - Workflowy :

# Lab. 기본적인 kubectl 명령어

- 객체 목록 불러오기
  - “[kubectl get \[객체 타입\]](#)” Ex) kubectl get pods : pod 목록을 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,deployments,pods --all”
    - 실습 중에 잘못 생성되었거나 초기화가 필요할 경우 사용
    - 콤마(,)로 구분하며 붙여 적기
- 객체 상세 설명 확인하기
  - “[kubectl describe \[객체 타입\] \[객체 이름\]](#)”  
Ex) “kubectl describe service wordpress”

# Lab. 이미지를 통한 어플리케이션 배포

- 현재 작동 중인 pod들이 없는지 확인
  - [kubectl get pods](#)
- Nginx 이미지 예제로 배포하기
  - [kubectl run first-deployment --image=nginx](#)
- 실행된 Pods 확인
  - [kubectl get pods](#)
  - 각 Pod들은 ‘Pod명-{hash}’ 형식의 고유한 이름을 가짐

# Lab. 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. (Curl 명령이 없으면 설치)  
`apt-get update`  
`apt-get install curl`
  - Curl 명령어로 호출하기  
`curl localhost`



# 서비스 접속문제 해결하기

- 첫번째, 해당 Pod의 log를 확인한다. (logs 명령은 Pod가 실행중어야 함)
  - kubectl logs [pod 이름] -f
- 두번째, 해당 서비스가 localhost로 접근이 되는가를 해당 컨테이너 내부에서 확인
  - kubectl exec -it [복사된 pod 이름] -- /bin/bash  
curl localhost  
위와 같이 확인했을 때 서비스가 연결된다면, 다음 단계 확인
- 세번째, 해당 서비스의 Docker 이미지에서 Port 설정이 제대로 되었는가?
  - ex) Nginx 서비스는 80으로 노출되었는데, Dockerfile에서 EXPOSE 8080으로 설정되었다면, 해당 서비스는 Docker 컨테이너 밖으로 포트를 노출하지 못함  
EXPOSE 80으로 수정해야 함
- 네번째, 해당 쿠버네티스 Service yaml 설정에서 port 넘버가 잘못된 경우
  - Docker 이미지의 Port 번호를 쿠버네티스의 Service yaml이 자동으로 가져오지 못하므로  
spec: port: 80 ← 이 부분을 확인, Dockerfile에서 EXPOSE 한 포트와 동일한지 확인  
protocol: TCP  
targetPort: 80

→ 실습 중, Kubernetes 공통적 오류 Guide : [실습 workflow > MSA 교육 > 실습 중 공통적 오류에 대한 대처](#)

# 설정 파일을 통한 pod 배포 (1/2)

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

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

# 설정 파일을 통한 pod 배포 (2/2)

- 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 (1/2)

- Node를 확인하여 label 붙이기
  - kubectl get nodes
  - kubectl label nodes [노드이름] disktype=ssd
- Label과 함께 노드 목록 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
```

# 원하는 노드 타입에 Pod 몰기 (2/2)

- Pod가 Create 되지 않고, Pending 상태
- Node에 Label 추가
  - `kubectl get nodes`
  - `kubectl label nodes <your-node-name> disktype=ssd`
  - `kubectl get nodes --show-labels | grep ssd`
- Pod 확인
  - `kubectl get all`
- Node 정보와 함께 Pod 확인
  - `kubectl get po -o wide`

# Pod Initialization

- Init.yaml 파일을 만든다.
  - nano init.yaml (오른쪽 내용을 Copy&Paste)
- Pod 생성
  - kubectl create -f init.yaml
- Pod 접속하기
  - kubectl exec -it init-demo -- /bin/bash
  - cd /usr/share/nginx/html
  - cat index.html

```
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: {}
```

Pod 초기화 시점에  
실행

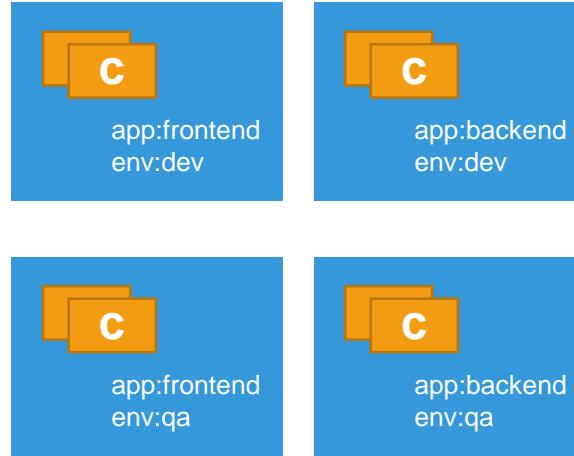
# Lab. Pod



Lab Time

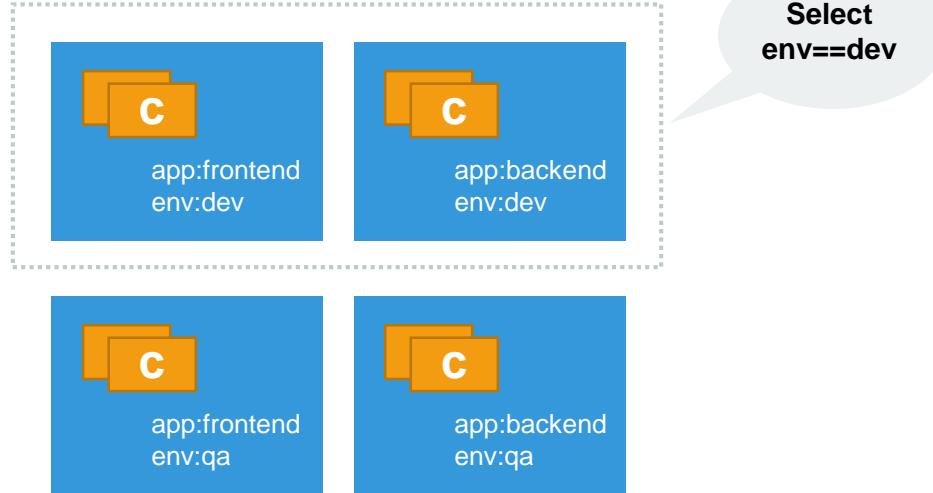
- Lab Script Location
  - Workflowy :

# Labels



- Labels은 객체 식별 정보로서 Kubernetes 객체라면 모두 붙일 수 있다.
- Label들은 요구사항에 맞춰 개체의 하위 집합을 구성하고 선택하는데 사용된다.
- Label들은 객체에 고유성을 제공하지 않아, 여러 객체들은 같은 label을 가질 수 있다.

# Label Selectors



- Label Selectors들은 객체들의 집합을 선택하며, kubernetes는 2가지 종류를 지원한다.
  - **Equality-Based Selectors**  
Uses the `=`, `==`, `!=` 연산자를 이용하여 Label key와 value 값을 기반으로 객체들을 필터링 할 수 있다.
  - **Set-Based Selectors**  
`in`, `notin`, `exist` 연산자를 사용하며, value 값들을 기반으로 객체들을 선택할 수 있다.

# Lab. Label

- kubectl get po # 실행 중, Pod list 확인
- kubectl edit po <pod 명> # Pod 인스턴스에 Label 추가

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2020-02-16T11:22:56Z"
  labels:
    env: test
  name: init-demo
  namespace: default
  resourceVersion: "588586"
```

] ← vi 에디터로 편집

- kubectl get pods -l env=test
- kubectl get pods --selector env=test
- kubectl get pods --selector 'env in (test, test1)' # or 연산
- kubectl get po --selector 'env in(test, test1), app in (nginx, nginx1)' # and 연산
- kubectl get po --selector 'env,app notin(nginx)' # env가 있으면서, app이 nginx가 아닌 Pod

# Annotations

- Label 처럼 식별 정보는 아닌 임의의 비 식별 메타데이터를 객체에 key-value 형태로 추가

```
"annotations": {  
    "key1" : "value1",  
    "key2" : "value2"  
}
```

- 주로, 히스토리, 스케줄 정책, 부가 정보 등을 기술
- 배포 주석을 추가해 서비스를 Deploy하고, 이전 서비스로 롤백 시, 해당 정보를 활용해 롤백

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: nginx-deployment  
  labels:  
    app: nginx  
  annotations:  
    kubernetes.io/change-cause=nginx:1.7.9  
spec:  
  replicas: 3  
  selector:  
    .....
```



# One-dash, Double-dash

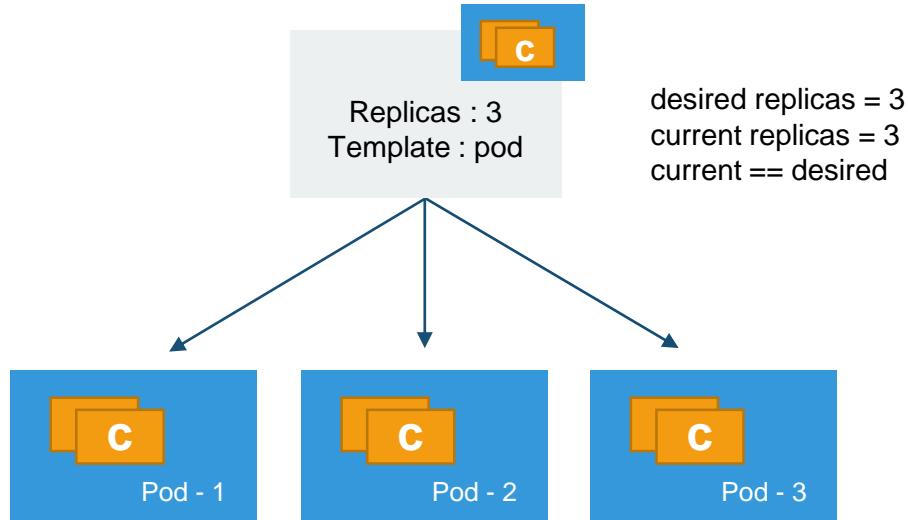
- Generally, in Command Options
  - Some have a long form without a short form ( -- author, --block-size)
  - Some have a short form without a long form (-c, -f, -g)
  - Some have both a long form and a short form (-A / --almost-all, -b / --escape)
- In Pod Selector, short form ( -l ) equals long form( --selector )

# Replication Controllers

- Master node의 controller manager의 한 부분이다.
- Pod의 복제품이 주어진 개수만큼 작동하고 있는지 확인하고 개수를 조절한다.
- Replication Controller는 Pod를 생성하고 관리한다.
  - 일반적으로 Pod는 자기 복구가 불가능 하기에 단독으로 배포를 하지 않는다.

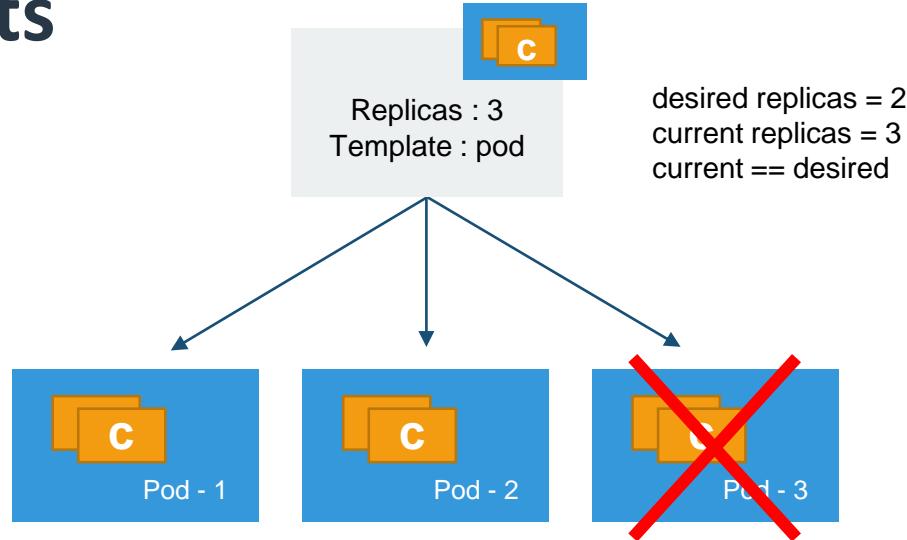


# ReplicaSets



- ReplicaSet(rs)은 Replication Controller의 업그레이드 버전
- ReplicaSet은 equal 및 set 기반 Selector를 모두 지원하는 반면, Replication Controller는 equal기반 Selector만 지원

# ReplicaSets



- 지정된 수의 Pod (Desired State)가 항상 실행되도록 보장
- ReplicaSets은 단독으로도 사용 가능하지만, 주로 Pod Orchestration에 사용(Pod creation, deletion, updates)
- Deployment가 ReplicaSets을 자동 생성하기 때문에 사용자는 관리에 신경 쓰지 않아도 됨

# Lab. 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](#)
- ReplicaSet을 삭제
  - [kubectl delete rs/frontend](#)  
(관련된 pod 전부 제거한다.)
  - [kubectl delete rs/frontend --cascade=false](#)  
(ReplicaSet은 제거하지만 Pod는 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
```

# Lab. ReplicaSet

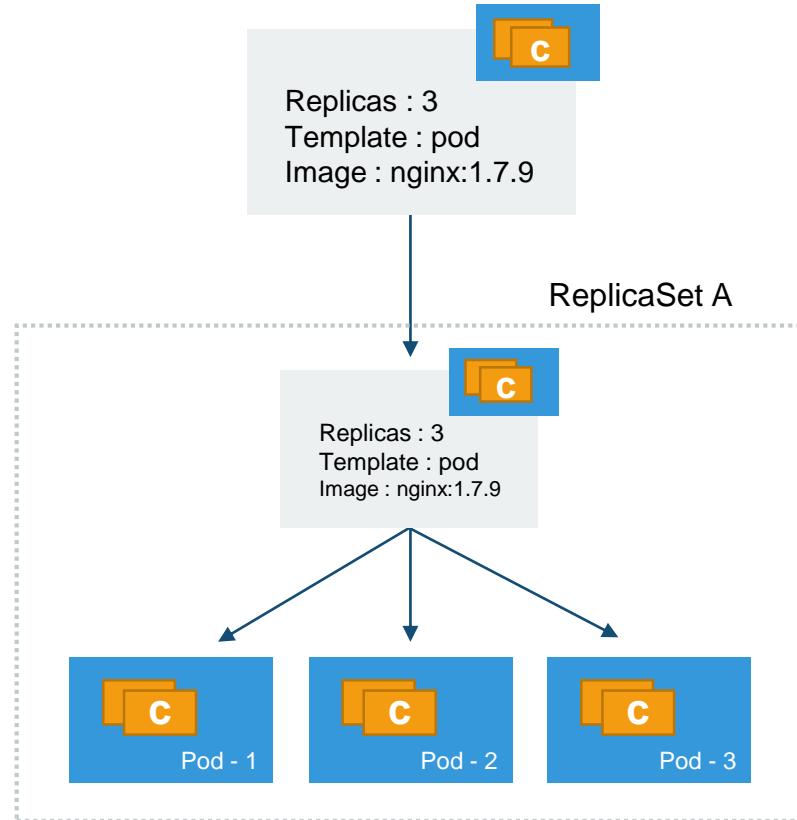


Lab Time

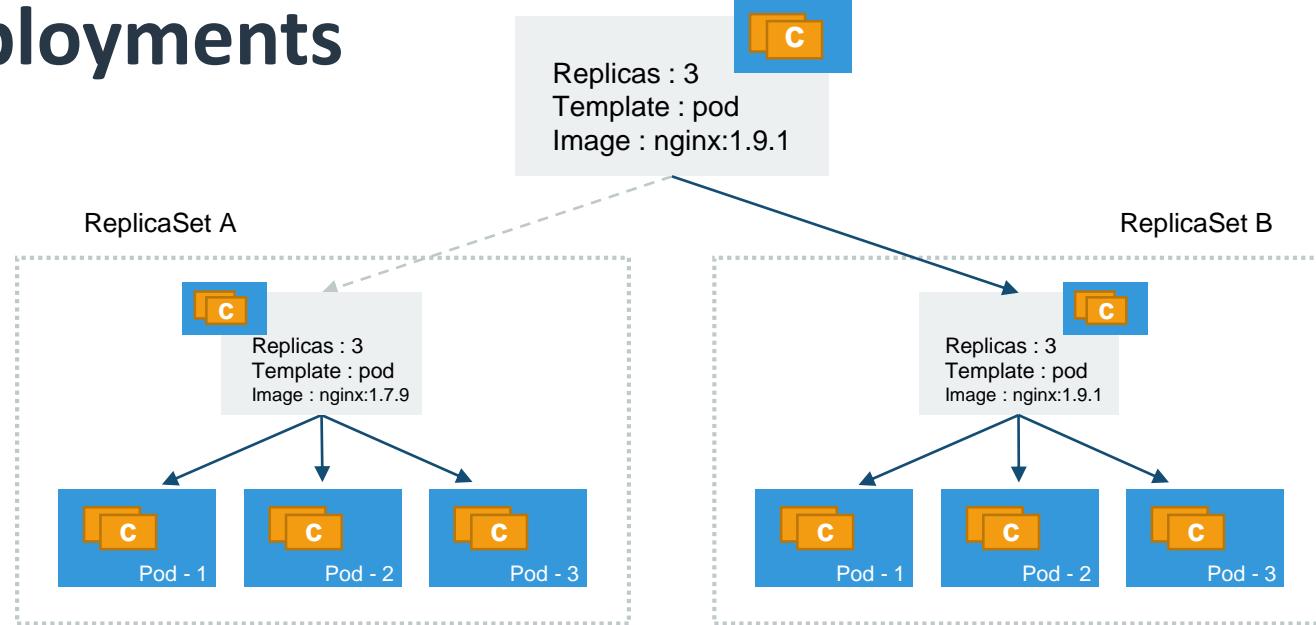
- Lab Script Location
  - Workflowy :

# Deployments

- Deployment 객체는 Pods와 ReplicaSets에 대한 선언적 업데이트를 제공한다.
- Deployment Controller는 Master node 컨트롤 관리자의 일부로 Desired state가 항상 만족이 되는지 확인한다.
- Deployment 가 ReplicaSet을 만들고 ReplicaSet은 그 뒤에 주어진 조건만큼의 Pod들을 생성한다.



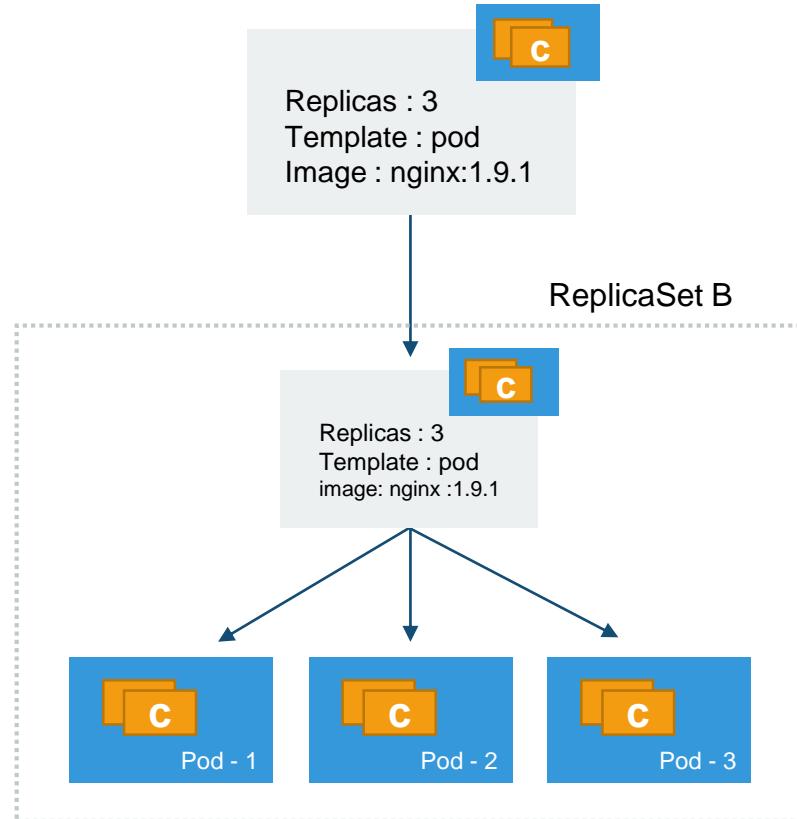
# Deployments



- Deployment의 Pod template이 바뀌게 되면, 새로운 ReplicaSet이 생성되는데, 이를 **Deployment rollout**이라고 한다.
- Rollout은 Pod template에 변동이 생겼을 경우에만 동작하며, Scaling등의 작업은 ReplicaSet을 새로 생성하지 않는다.

# Deployments

- 새로운 ReplicaSet이 준비되면 Deployment는 새로운 ReplicaSet을 바라본다.
- Deployment들은 Deployment recording 등의 rollback 기능을 제공하며, 문제가 발생했을 경우, 이전 단계로 돌릴 수 있다.



# 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
```

Pod 생성 시, 이 템플릿 참조  
도커허브에서 nginx 1.7.9  
이미지를 가져와 'nginx' 이름의  
컨테이너 생성

# Scaling Deployments

- Deployment의 replica의 개수를 확인한다.
  - `kubectl get pods`
- Deployment의 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 replicas: 5 속성 추가; 있으면 수정)
- 변경한 파일을 적용
  - kubectl apply -f nginx-deployment.yaml
- 변경 내용을 확인
  - kubectl get pods
- 설정파일에 추가된 replicas 속성을 삭제 후 다시 적용
  - Nano nginx-deployment.yaml (replicas)
  - 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 apply -f nginx-deployment.yaml`
  - `kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1`
  - `kubectl rollout status deployment/nginx-deployment`
- 변경 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`

# deployment에 적용된 Image:버전 추가 표시

- 객체를 롤백 처리

- `kubectl rollout undo deployment/nginx-deployment`

- 진행을 확인

- `kubectl get deployments -o wide`

# Lab. Deployments

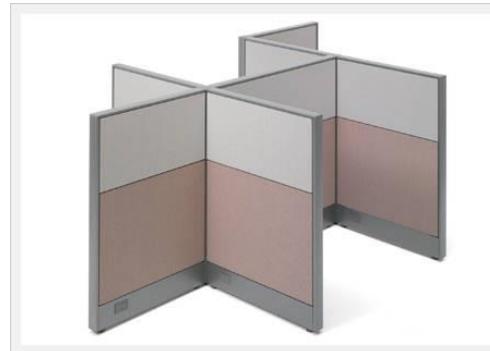


Lab Time

- Lab Script Location
  - Workflowy :

# Namespaces

- Kubernetes는 동일 물리 클러스터를 기반으로 하는 복수의 가상 클러스터를 지원하는데 이들 가상 클러스터를 Namespace라고 한다.
- Namespace를 활용하면, 팀이나 프로젝트 단위로 클러스터 파티션을 나눌 수 있다.
- Namespace 내에 생성된 자원/객체는 고유하나, Namespace 사이에는 중복이 발생할 수 있다.



# Namespaces

- Namespaces Object 조회

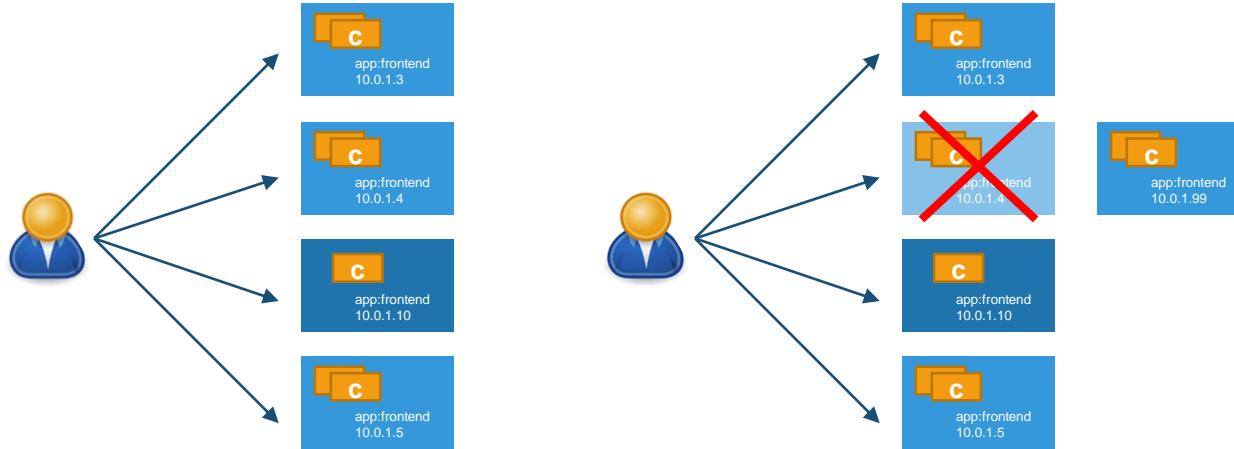
```
$ kubectl get namespaces
  NAME      STATUS   AGE
  default   Active  11h
  kube-public   Active  11h
  kube-system   Active  11h
```

- Kubernetes는 처음에 3개의 초기 네임스페이스를 가진다.
  - default** : 다른 namespace를 갖는 다른 객체들을 가지고 있다.
  - kube-public** 은 클러스터 bootstrapping 같은 모든 유저가 사용 가능한 특별한 namespace 이다.
  - kube-system**: Kubernetes system에 의해서 생성된 객체를 가지고 있다.
- Resource Quotas를 사용하여 namespace 내에 존재하는 자원들을 나눌 수 있다.

# Lab. Namespaces

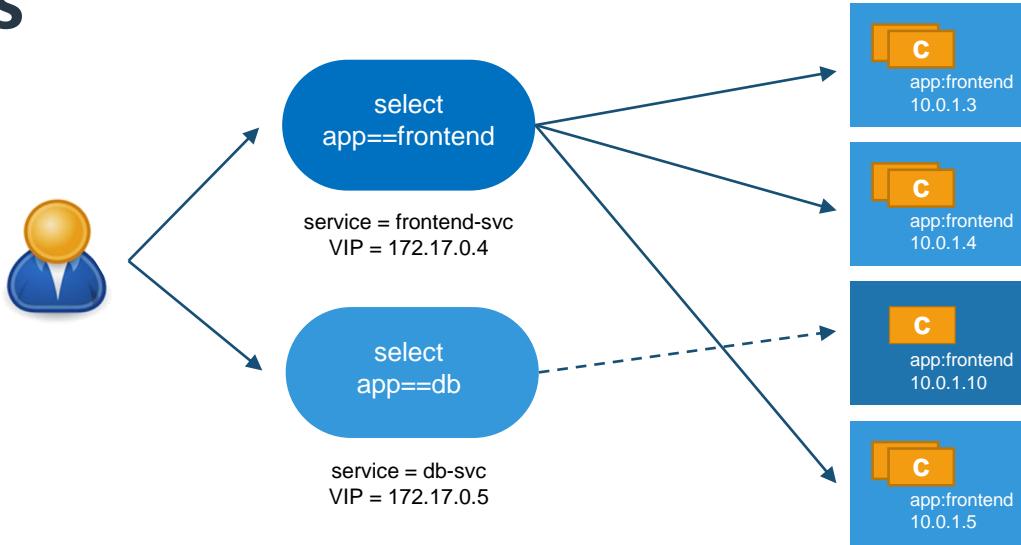
- 원하는 Namespace에 Pod 생성
  - `Kubectl run nginx --image=nginx --namespace=<insert-namespace-name-here>`
- 특정 Namespace상에 생성된 Pod 조회
  - `kubectl get po --namespace=<insert-namespace-name-here>`
- Namespace Option 생략 시, default Namespace

# Pod Access Issues



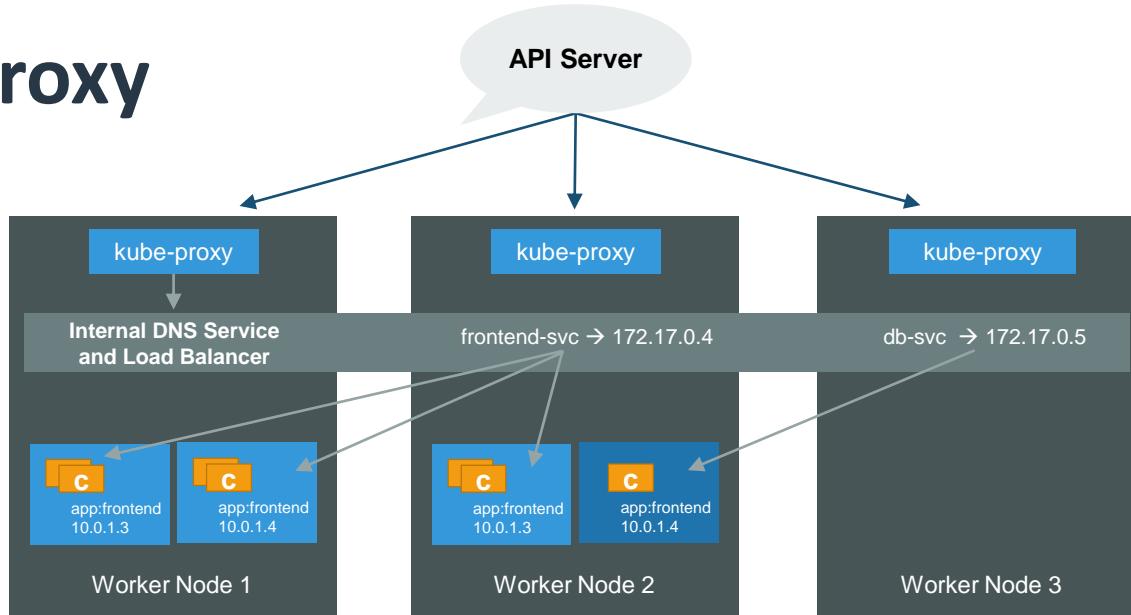
- 어플리케이션에 접근하기 위해서는, 사용자가 Pod에 접근해야 한다.
- Pod들은 언제든지 소멸 가능하기에 IP주소는 고정되어 있지 않다.
- 사용자가 직접 IP주소로 Pod에 연결되어 있을 때, Pod가 죽어서 새로 만들어지면 접속할 방법이 없다.
- 이 상황을 극복하기 위해서 추상화를 통해 Service라는 Pod들의 논리적 집단을 만들어 규칙을 설정하고 사용자들은 여기에 접속을 한다.

# Services



- Selector를 사용하여 Pod를 논리적 그룹으로 나눌 수 있다.
- 각 논리적 그룹에 대해서 Service name이라는 이름을 부여할 수 있다.
- 사용자는 Service IP주소를 통해 Pod에 접속하게 된다.
  - 각 Service에 부여된 IP는 클러스터 IP 라고도 부른다.
- Service는 각 Pod에 대해 Load balancing을 자동으로 수행

# kube-proxy



- 모든 Worker node들은 kube-proxy라는 데몬을 실행하며, 이 데몬은 Service 및 End-point 생성 /삭제를 위해 마스터 노드의 API 서버를 모니터링한다.
- 각 node에 있는 모든 새로운 Service에 대해서 kube-proxy는 Iptable 규칙을 구성하여, ClusterIP의 트래픽을 캡처하고 이를 End-point로 보낸다.
- Service가 제거되면, kube-proxy는 노드상의 Iptables 규칙도 지운다.

# Service Discovery

- Service는 클라이언트가 애플리케이션에 접근하는 Kubernetes의 채널로 런타임시, 이를 검색할 수 있는 방법이 필요
- DNS를 이용하는 방법
  - 서비스는 생성되면, [서비스 명].[네임스페이스명].svc.cluster.local이라는 DNS명으로 쿠버네티스 내부 DNS에 등록되고, 쿠버네티스 내부 클러스터에서는 이 DNS명으로 접근 가능한데, 이 때 DNS에서 리턴해 주는 IP는 외부 IP(External IP)가 아니라 Cluster IP(내부 IP) 임
- External IP를 명시적으로 지정하는 방법
  - 외부 IP는 Service의 Spec 부분에서 externalIPs 항목의 Value로 기술

# ServiceType

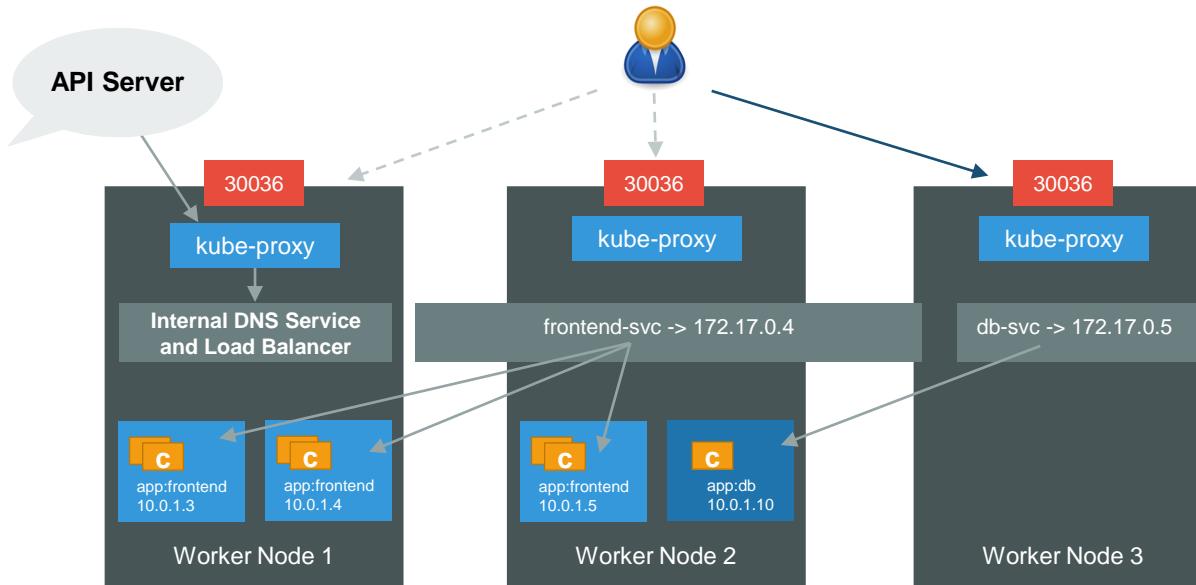
- Service를 정의할 때 Access Scope를 따로 정할 수 있다.
  - 클러스터 내에서만 접근이 가능한가?
  - 클러스터 내와 외부에서 접근이 가능한가?
  - 클러스터 밖의 리소스에 대한 Map을 가지는가?
- Service 생성 시, IP주소 할당 방식과 서비스 연계 등에 따라 4가지로 구분
  - ClusterIP
  - NodePort
  - LoadBalancer
  - ExternalName

# ServiceType : ClusterIP

- 디폴트 설정으로 서비스에 클러스터 ip를 할당
- 쿠버네티스 클러스터 내에서만 이 서비스에 접근 가능
- 외부에서는 외부 IP를 할당 받지 못했기 때문에 접근이 불가능

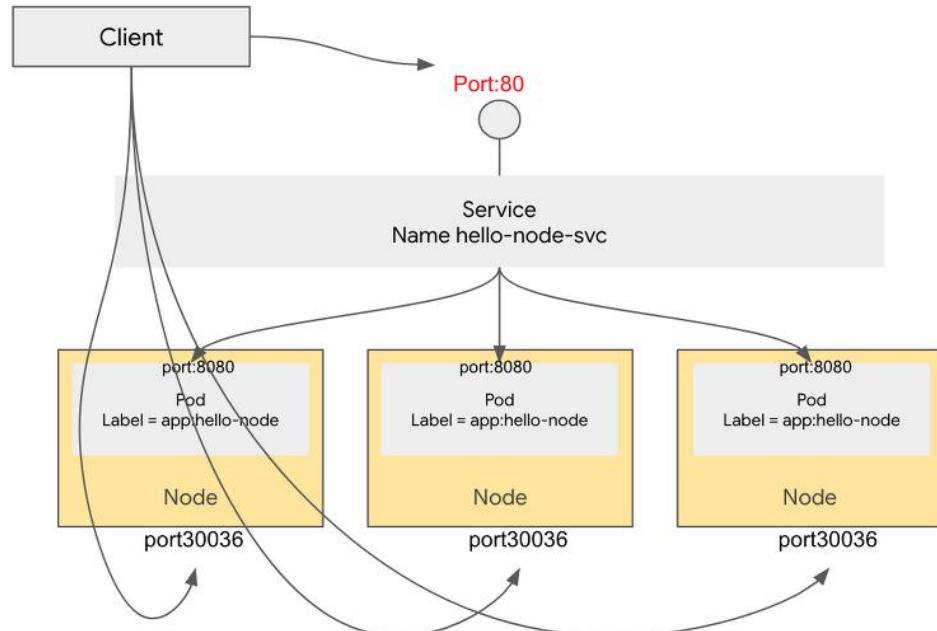
# ServiceType : NodePort

- 고정 포트(NodePort)로 각 노드의 IP에 서비스를 노출
- Cluster IP 뿐만 아니라, 노드의 IP와 포트를 통해서도(<NodeIP>:<NodePort>) 접근 가능



# ServiceType : NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: hello-node-svc
spec:
  selector:
    app: hello-node
  type: NodePort
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: 8080
      nodePort: 30036
```

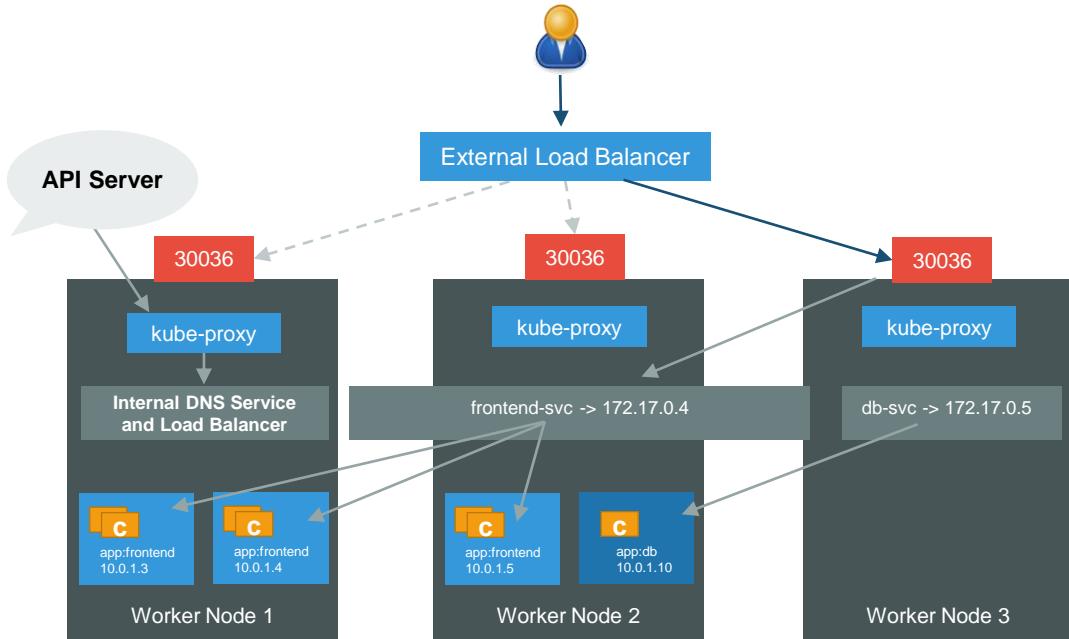


# ServiceType : LoadBalancer

- 클라우드 밴더의 로드밸런싱 기능을 사용
  - NodePort와 ClusterIP Service들은 자동으로 생성되어 External Load Balancer가 해당 포트로 라우팅
  - Service들은 각 Worker node에서 Static port로 노출

LoadBalancer ServiceType은 기본 인프라가 Load balancer의 자동 생성을 제공하고, Kubernetes를 지원 할 경우에만 작동

Ex) Azure, Google Cloud Platform, AWS

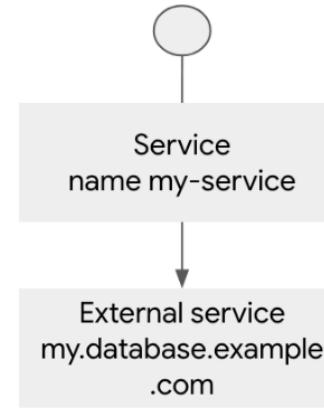


# ServiceType: ExternalName

- 외부 서비스를 쿠버네티스 내부에서 호출할 때 사용
- 쿠버네티스 클러스터내의 Pod들은 클러스터 IP를 가지고 있기 때문에 클러스터 IP 대역 밖의 서비스를 호출하고자 하면, NAT 설정등 복잡한 설정이 필요
- Azure, AWS 나 GCP와 같은 클라우드 환경에서 DBMS나, 또는 클라우드에서 제공되는 매지니드 서비스 (RDS, CloudSQL)등을 사용하고자 할 경우
  - 쿠버네티스 클러스터 외부이기에 호출이 어려운 경우가 있는데, 이를 쉽게 해결할 수 있는 방법이 ExternalName 타입

# ServiceType: ExternalName

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```



- 서비스를 ExternalName 타입으로 설정하고, 주소를 my.database.example.com으로 설정해주면 이 my-service는 들어오는 모든 요청을 my.database.example.com 으로 포워딩
- 일종의 프록시와 같은 역할

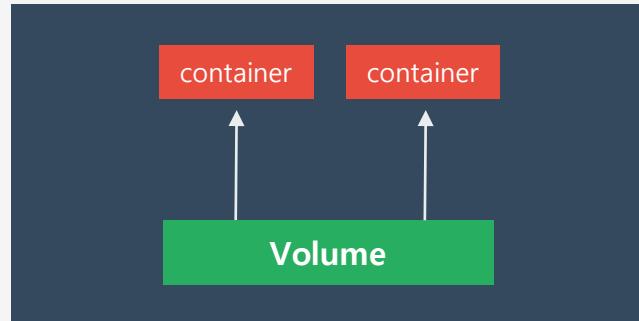
# Lab. Service



Lab Time

- Lab Script Location
  - Workflowy :

# Volumes



- Volume은 Pod에 장착되어, 그 Pod에 있는 Container 간에 공유된다.
- Volume은 포드와 수명이 동일하며, 포드 컨테이너보다 수명이 길어, 컨테이너를 다시 시작해도 데이터를 보존 할 수 있습니다.

# Types of Volumes

- Pod안에 마운트된 디스크는 Volume type에 의해서 사용 유형이 정의된다.
- Volume Type 내에는 디스크의 크기, 내용 등의 속성을 정한다.
- Types of Volumes
  - 임시 디스크
    - **emptyDir**  
Pod를 위해서 생성된 공간으로 Pod가 죽으면 내용도 사라진다.
  - 로컬 디스크
    - **hostPath**  
호스트의 로컬 디스크의 경로를 Pod에 마운트해 사용, Pod가 죽어도 해당 volume은 계속 사용 가능하다.
  - 네트워크 디스크
    - **gcePersistentDisk** : Google Compute Engine persistent disk
    - **awsElasticBlockStore** : AWS EBS Volume
    - **AzureDisk, gitRepo, NFS, iSCSI**
    - **Secret** : 비밀번호와 같은 중요한 정보들을 저장
    - **persistentVolumeClaim**  
다른 저장 장치 사용 가능

# Volumes : emptyDir

```
apiVersion: v1
kind: Pod
metadata:
  name: shared-volumes
spec:
  containers:
    - image: redis
      name: redis
      volumeMounts:
        - name: shared-storage
          mountPath: /data/shared
    - image: nginx
      name: nginx
      volumeMounts:
        - name: shared-storage
          mountPath: /data/shared
  volumes:
    - name: shared-storage
      emptyDir: {}
```

- emptyDir의 생명주기는 컨테이너 단위가 아닌 Pod 단위로 Container 재기동에도 계속 사용 가능
- 생성된 Pod 확인

```
apexacme@APEXACME: ~/yaml$ kubectl get all
NAME           READY   STATUS    RESTARTS   AGE
pod/shared-volumes   2/2     Running   0          10m
```

- 지정 컨테이너 접속 후, 파일 생성
  - kubectl exec -it shared-volumes --container redis -- /bin/bash
  - cd /data/shared
  - echo test... > test.txt
- 다른 컨테이너로 접속 후, 파일 확인
  - kubectl exec -it shared-volumes --container nginx -- /bin/bash
  - cd /data/shared
  - ls

# Volumes : hostPath

```
apiVersion: v1
kind: Pod
metadata:
  name: hostpath
spec:
  containers:
    - name: redis
      image: redis
      volumeMounts:
        - name: somepath
          mountPath: /data/shared
  volumes:
    - name: somepath
      hostPath:
        path: /tmp
      type: Directory
```

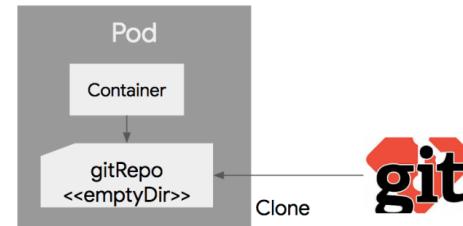
- Node의 Local 디스크 경로를 Pod에 마운트
- 같은 hostPath에 있는 볼륨은 여러 Pod사이에서 공유
- Pod가 삭제되어도 hostPath에 있는 파일은 유지
- Pod가 재기동 되어 다른 Node에서 기동될 경우, 새로운 Node의 hostPath를 사용
- Node의 로그 파일을 읽는 로그 에이전트를 배포하였을 경우 유용하게 사용 가능
- Pod 생성 후, /data/shared에서 ls -al

```
drwxrwxrwt 8 redis root 4096 Feb 17 03:08 .
drwxr-xr-x 3 redis redis 4096 Feb 17 03:07 ..
drwxrwxrwt 2 redis root 4096 Feb 11 05:39 .icE-unix
drwxrwxrwt 2 redis root 4096 Feb 11 05:39 .Test-unix
drwxrwxrwt 2 redis root 4096 Feb 11 05:39 .X11-unix
drwxrwxrwt 2 redis root 4096 Feb 11 05:39 .XIM-unix
drwxrwxrwt 2 redis root 4096 Feb 11 05:39 .font-unix
drwx----- 3 redis root 4096 Feb 11 05:44 systemd-private-fe55104f60e34b2ea4
```

# Volumes example : gitRepo

```
apiVersion: v1
kind: Pod
metadata:
  name: gitrepo-volume-pod
spec:
  containers:
    - image: nginx:alpine
      name: web-server
      volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
          readOnly: true
    ports:
      - containerPort: 80
        protocol: TCP
  volumes:
    - name: html
      gitRepo:
        repository: https://github.com/luksa/kubia-website-example.git
        revision: master
        directory: .
```

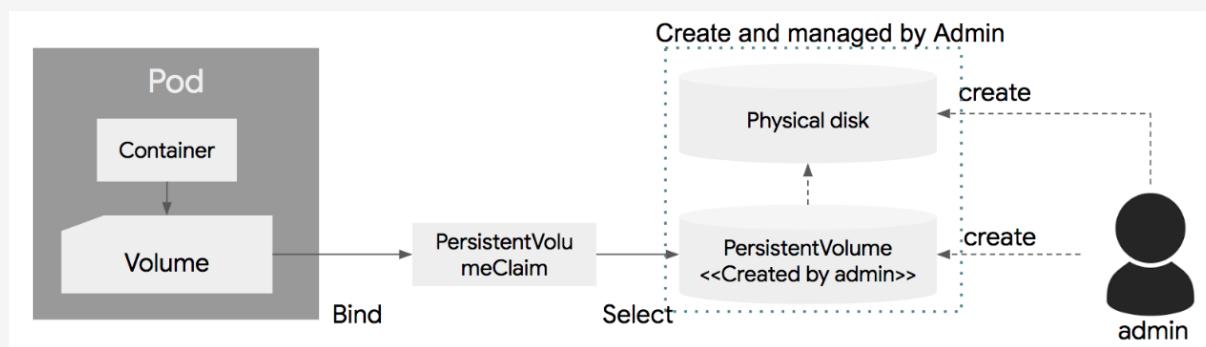
- Pod 생성시 지정된 Git 리파지토리의 특정 revision을 Clone을 이용 내려받은 후, 디스크 볼륨을 생성
- 물리적으로는 emptyDir이 생성되고 Git Cloning



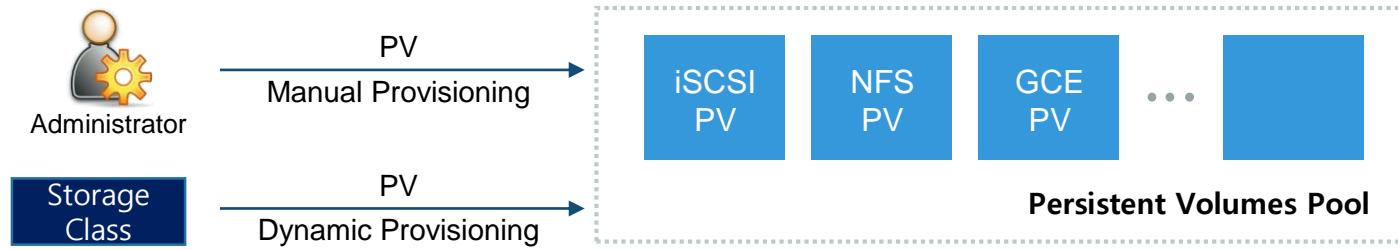
- HTML 같은 정적 파일 및 Nodejs 같은 스크립트 기반 코드 배포에 유용
- Pod 생성 후, Pod상의 Git file과 ubuntu cli에서 git Clone 후 내용 비교

# PersistentVolume and PersistentVolumeClaim

- 특정 IT 환경에서는 영속성 있는 대용량 스토리지는 관리자에 의해서 관리
  - 사용자는 사용법만 알고, 관리에는 신경을 쓰지 않아도 된다.
- 시스템 관리자가 실제 물리 디스크를 생성한 뒤, 이 디스크를 PersistentVolume 이라는 이름으로 Kubernetes에 등록
- 개발자는 Pod 생성 시, 볼륨을 정의하고, 해당 볼륨의 정의 부분에 PVC(PersistentVolumeClaim)를 지정하여 관리자가 생성한 PV와 연결



# Dynamic PersistentVolume Provisioning



- PV는 관리자에 의해 수동으로 생성될 수 있지만, 자동 생성도 가능(Dynamic Provisioning)
- StorageClass (sc) Object
- StorageClass 객체에 의해서 PersistentVolumes은 유동적 제공 가능
  - StorageClass는 PersistentVolume를 만들기 위해서 미리 정의된 제공자와 파라미터를 가짐
  - PersistentVolumeClaims를 사용, 사용자는 동적 PV 생성 요청을 보내고 이는 StorageClass 리소스에 연결
- PersistentVolumes을 통한 스토리지 관리를 제공하는 Volume Types :
  - GCEPersistentDisk, AWSElasticBlockStore, AzureDisk, NFS, iSCSI

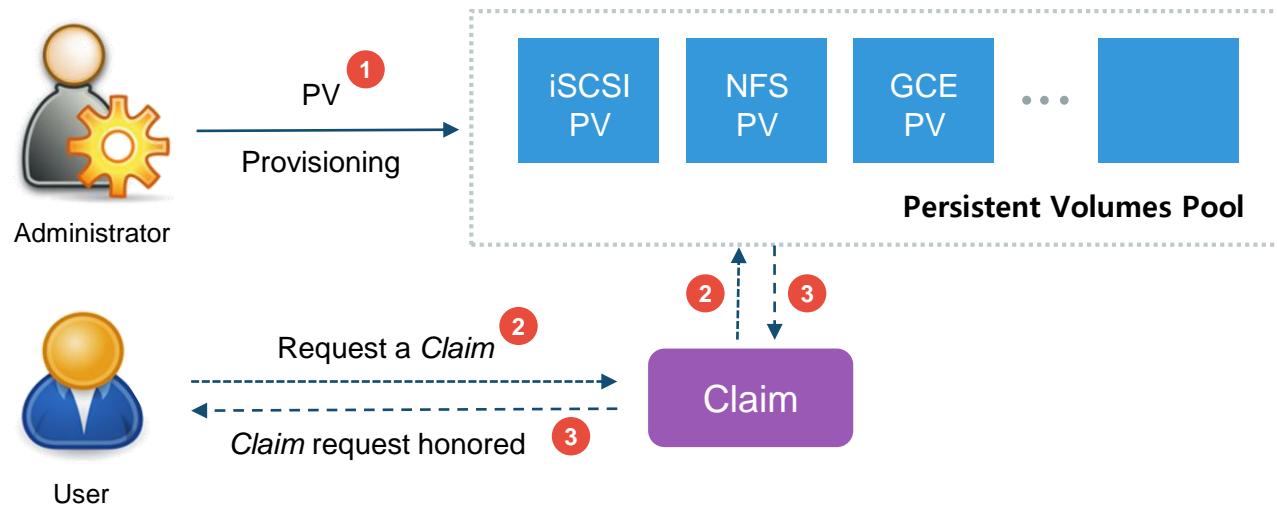
# Dynamic PersistentVolume Provisioning

```
$ kubectl get storageclass
  NAME          PROVISIONER          AGE
default (default)  kubernetes.io/azure-disk  1h
managed-premium   kubernetes.io/azure-disk  1h
```

AzureDisk Storage class Object

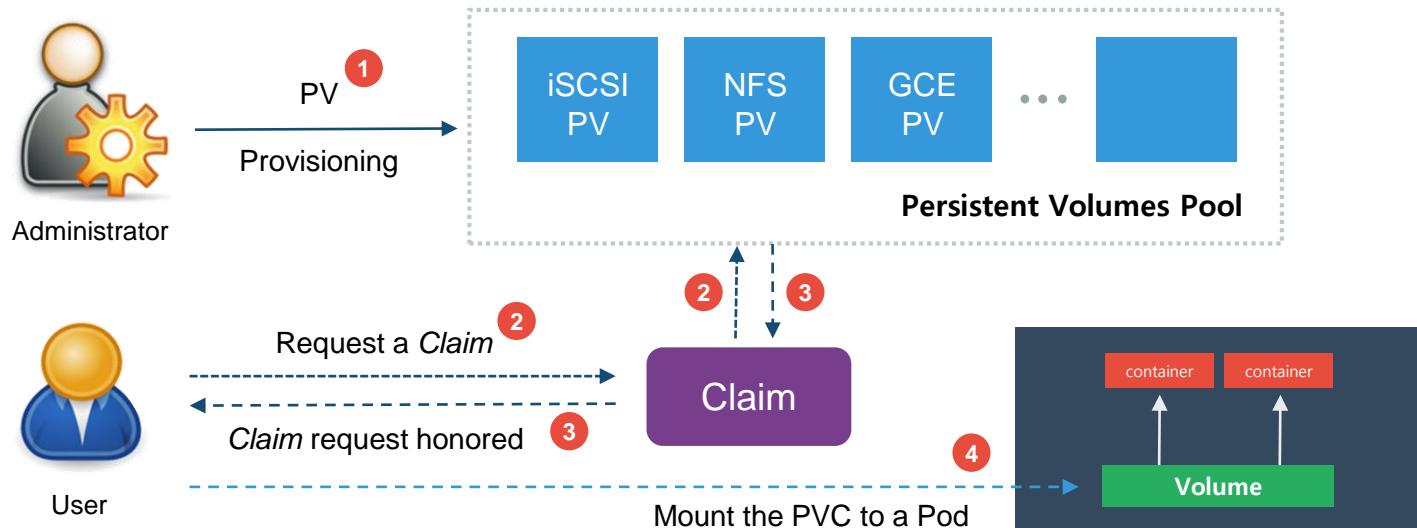
- Default storage class는 표준 Azure 디스크를 프로비전
  - HDD에 의해 지원, 비용 효율적인 개발 및 테스트 워크로드에 적합
- Managed-premium storage class는 프리미엄 Azure 디스크를 프로비전
  - SSD 기반 고성능의 Low-latency 디스크 지원
  - 프로덕션 워크로드를 실행하는 VM에 완벽한 디스크

# PersistentVolumeClaims



- PersistentVolumeClaim (PVC)은 유저로부터의 스토리지 요청  
유저들은 크기, 접근 모드, 등등에 따라 요청을 한다. 적합한 PersistentVolume이 발견되면, PersistentVolumeClaim에 바운드 된다.

# PersistentVolumeClaims



- 바운드가 성공적 이였을 경우, PVC 리소스는 pod 내에서 사용가능해 진다.
- 사용자가 작업을 마치면 연결된 PV를 해제할 수 있고, 해제된 PersistentVolume는 나중에 재활용된다.

# Create PersistentVolumeClaim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azure-managed-disk
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: default
resources:
  requests:
    storage: 1Gi
```

- **accessMode:**

- ReadWriteOnce : 하나의 Pod에만 마운트되고, 읽고 쓰기 가능
- ReadOnlyMany : 여러 개의 Pod에서 마운트되고, 동시에 읽기만 가능 (쓰기는 불가능)
- ReadWriteMany : 여러 개의 Pod에서 마운트되고, 동시에 읽고 쓰기 가능

- **kubectl apply -f volume-pvc.yaml**

- **kubectl get pvc**

```
apexacme@APEXACME: /yaml/volume$ kubectl get pvc
NAME           STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
azure-managed-disk   Pending
apexacme@APEXACME:~/yaml/volume$
```

- **kubectl describe pvc**

```
apexacme@APEXACME: /yaml/volume$ kubectl get pvc
NAME           STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
azure-managed-disk   Bound    pvc-817b4f22-5141-11ea-a89c-12c099b138d1   1Gi        RWO          default      67s
apexacme@APEXACME:~/yaml/volume$
```

# Create Pod with PersistentVolumeClaim

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: nginx:1.15.5
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 250m
          memory: 256Mi
      volumeMounts:
        - mountPath: "/mnt/azure"
          name: volume
    volumes:
      - name: volume
        persistentVolumeClaim:
          claimName: azure-managed-disk
```

- kubectl apply -f pod-with-pvc.yaml
- kubectl get pod
- kubectl describe pod mypod
- kubectl exec -it mypod -- /bin/bash
- cd /mnt/azure
- df -k 로 size 확인

```
root@mypod:/# df -k
Filesystem 1K-blocks Used Available Use% Mounted on
overlay 101445900 19656992 81772524 20% /
tmpfs 65536 0 65536 0% /dev
tmpfs 3556916 0 3556916 0% /sys/fs/cgroup
/dev/sda1 101445900 19656992 81772524 20% /etc/hosts
/dev/sdc 999320 1284 981652 1% /mnt/azure
shm 65536 0 65536 0% /dev/shm
tmpfs 3556916 12 3556904 1% /run/secrets/kubernetes.io/serviceaccount
tmpfs 3556916 0 3556916 0% /proc/acpi
tmpfs 3556916 0 3556916 0% /proc/scsi
tmpfs 3556916 0 3556916 0% /sys/firmware
root@mypod:/#
```

# Lab. Volumes

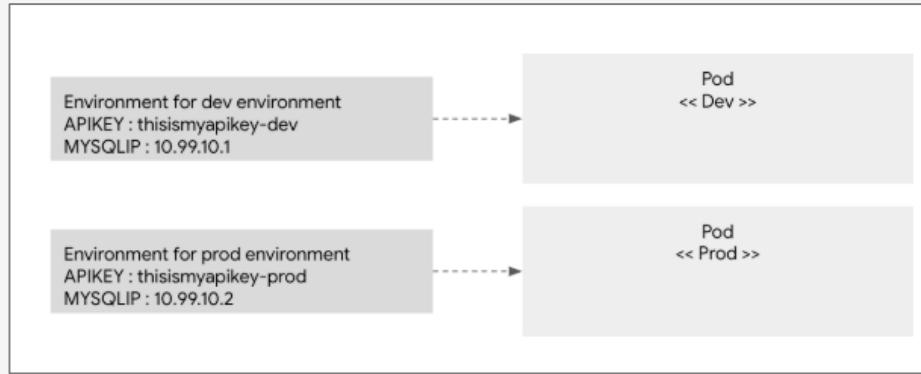


Lab Time

- Lab Script Location
  - Workflowy :

# ConfigMaps

- ConfigMaps는 컨테이너 이미지로부터 설정 정보를 분리할 수 있게 해준다.
- 환경변수나 설정값 들을 환경변수로 관리해 Pod가 생성될 때 이 값을 주입



- ConfigMaps은 2가지 방법으로 생성
  - 리터럴 값
  - 파일
- ConfigMaps는 etcd에 저장

# 리터럴 값으로부터 ConfigMap 생성

- ConfigMap을 생성하는 명령어

```
$ kubectl create configmap my-config --from-literal=key1=value1 -  
-from-literal=key2=value2  
configmap "my-config" created
```

- 설정된 ConfigMap 정보 가져오기

```
$ kubectl get configmaps my-config -o yaml  
apiVersion: v1  
data:  
  key1: value1  
  key2: value2  
kind: ConfigMap  
metadata:  
  creationTimestamp: 2017-05-31T07:21:55Z  
  name: my-config  
  namespace: default  
  resourceVersion: "241345"  
  selfLink: /api/v1/namespaces/default/configmaps/my-config  
  uid: d35f0a3d-45d1-11e7-9e62-080027a46057
```

- o yaml 옵션은 해당 정보를 yaml형태로 출력하도록 요청한다.
- 해당 객체는 종류가 ConfigMap이며 key-value 값을 가지고 있다.
- ConfigMap의 이름 등의 정보는 metadata field에 들어 있다.

# 파일로부터 ConfigMap 생성 (1/2)

- 아래와 같은 설정 파일을 만든다.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: customer1
data:
  TEXT1: Customer1_Company
  TEXT2: Welcomes You
  COMPANY: Customer1 Company Technology Pct. Ltd.
```

- customer1-configmap.yaml**라는 이름으로 파일을 생성하였을 경우, 아래와 같이 ConfigMap를 생성한다.

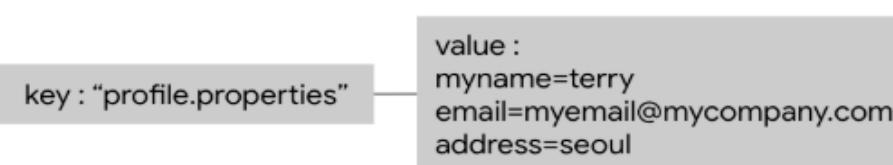
```
$ kubectl create -f customer1-configmap.yaml
configmap "customer1" created
```

# 파일로부터 ConfigMap 생성 (2/2)

- Userinfo.properties 파일을 생성하고,

```
myname=apexacme  
email=apexacme@uengine.org  
Address=seoul
```

- 파일을 이용해 ConfigMap을 만들 때는 --from-file을 이용해 파일명을 넘긴다.
- kubectl create configmap cm-file --from-file=./properties/profile.properties
  - 이때, 키는 파일명이 되고, 값은 파일 내용이 됨



# Containerizing with ConfigMap from Dockerizing

- Scenario
  - ConfigMap 생성
  - ConfigMap의 환경변수를 읽어 출력하는 NodeJS 어플리케이션 준비
  - Dockerfile 생성
  - Dockerizing & Azure Container Registry에 Push
  - Deployment yaml, Service yaml 준비
  - 배포 및 서비스 생성
  - 브라우저를 통해 서비스 확인
    - ConfigMap의 환경변수를 어플리케이션이 정상적으로 참조하여 출력하는지 여부

# Containerizing with ConfigMap from Dockerizing

- ConfigMap 생성

```
$ kubectl create configmap hello-cm --from-literal=language=java  
$ kubectl get cm  
$ kubectl get cm hello-cm -o yaml
```

- ConfigMap의 환경변수를 읽어 출력하는 NodeJS 어플리케이션

```
var os = require('os');  
var http = require('http');  
var handleRequest = function(request, response) {  
    response.writeHead(200);  
    response.end(" my prefered language is "+process.env.LANGUAGE+"\n");  
  
    //log  
    console.log("[ "+  
        Date(Date.now()).toLocaleString()+  
        "] "+os.hostname());  
}  
var www = http.createServer(handleRequest);  
www.listen(8080);
```

# Containerizing with ConfigMap from Dockerizing

- Dockerfile 생성

```
FROM node:carbon
EXPOSE 8080
COPY server.js .
CMD node server.js > log.out
```

- Dockerizing & Azure Container Registry에 Push

```
$ docker build -t (uengineorg).azurecr.io/cm-sandbox:v1 .
$ docker images
$ docker push (uengineorg).azurecr.io/cm-sandbox:v1
```

- 인증 오류 발생시, 로그인 수행 : az acr login --name *uengineorg*
- Push 확인 : Azure Portal > 컨테이너 레지스터리 > 리파지토리

# Containerizing with ConfigMap from Dockerizing

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: cm-deployment
spec:
  replicas: 1
  minReadySeconds: 5
  selector:
    matchLabels:
      app: cm-literal
  template:
    metadata:
      name: cm-literal-pod
      labels:
        app: cm-literal
    spec:
      containers:
        - name: cm
          image: uengineorg.azurecr.io/cm-sandbox:v1
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          env:
            - name: LANGUAGE
              valueFrom:
                configMapKeyRef:
                  name: hello-cm
                  key: language
```

- Deployment(cm-deployment.yaml) 생성/ 실행
- kubectl create -f cm-deployment.yaml
- \$ kubectl get deploy

# Containerizing with ConfigMap from Dockerizing

```
apiVersion: v1
kind: Service
metadata:
  name: cm-literal-svc
spec:
  selector:
    app: cm-literal
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: 8080
  type: LoadBalancer
```

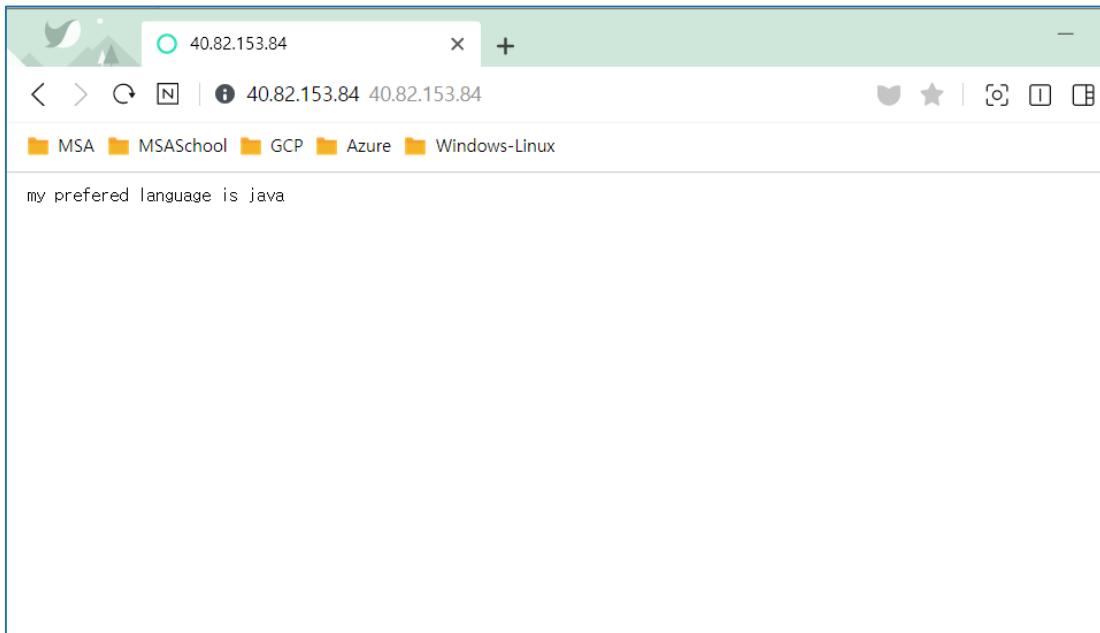
- Service(cm-service.yaml) 생성/ 실행  
\$ kubectl create -f cm-service.yaml
- \$ kubectl get svc

```
apexacme@APEXACME:~/yaml/configmap$ kubectl get svc
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
cm-literal-svc   LoadBalancer   10.0.99.121   40.82.153.84   80:30181/TCP   30m
kubernetes     ClusterIP    10.0.0.1     <none>        443/TCP    28h
apexacme@APEXACME:~/yaml/configmap$
```

- 브라우저를 통해 서비스 확인
  - Service의 External-IP 접속

# Containerizing with ConfigMap from Dockerizing

- 마이크로서비스 결과 확인



# Pod에서 ConfigMap 추가 사용하기

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: cm-file-deployment
spec:
  replicas: 3
  minReadySeconds: 5
  selector:
    matchLabels:
      app: cm-file
  template:
    metadata:
      name: cm-file-pod
      labels:
        app: cm-file
    spec:
      containers:
        - name: cm-file
          image: uengineorg.azurecr.io/cm-file:v1
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          env:
            - name: PROFILE
              valueFrom:
                configMapKeyRef:
                  name: cm-file
                  key: profile.properties
```

- 환경변수로 값 전달

- cm-file configMap에서 키가 “profile.properties” (파일명)인 값을 읽어와서 환경 변수 PROFILE에 저장
- 저장된 값은 파일의 내용인 아래 문자열이 됨
  - myname=terry  
email=myemail@mycompany.com  
address=seoul

# Pod에서 ConfigMap 추가 사용하기

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: cm-file-deployment-vol
spec:
  replicas: 3
  minReadySeconds: 5
  selector:
    matchLabels:
      app: cm-file-vol
  template:
    metadata:
      name: cm-file-vol-pod
      labels:
        app: cm-file-vol
    spec:
      containers:
        - name: cm-file-vol
          image: uengineorg.azurecr.io /cm-file-volume:v1
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          volumeMounts:
            - name: config-profile
              mountPath: /tmp/config
      volumes:
        - name: config-profile
          configMap:
            name: cm-file
```

- 디스크 볼륨으로 마운트 하기

- ConfigMap을 Volume으로 정의하고, 이 볼륨을 volumeMounts를 이용해 /tmp/config에 마운트 함
- 이때 중요한점은 마운트 포인트에 마운트 될때, ConfigMap내의 키가 파일명이 됨

# Lab. ConfigMap



Lab Time

- Lab Script Location
  - Workflowy :

# Secrets

- ConfigMap이 일반적인 환경 설정 정보나 Config정보를 저장하도록 디자인 되었다면, 보안이 중요한 패스워드나 API 키, 인증서 파일들은 Secret에 저장
- Secret은 정보보안 차원에서 추가적인 보안 기능을 제공
  - 예를 들어, API서버나 Node의 파일에 저장되지 않고, 항상 메모리에 저장되므로 상대적 접근이 어려움
  - Secret의 최대 크기는 1MB (너무 커지면, apiserver나 Kubelet의 메모리에 부하 발생)
- ConfigMap과 기본적으로 유사하나, 값(value)에 해당하는 부분을 base64로 인코딩해야 함
  - SSL인증서와 같은 binary파일의 경우, 문자열 저장이 불가능하므로 인코딩 필요
  - 이를 환경변수로 넘길 때나 디스크볼륨으로 마운트해서 읽을 경우 디코딩 되어 적용

```
apiVersion: v1
kind: Secret
metadata:
  name: hello-secret
data:
  language: amF2YQo=
```

# Kubectl 명령어로 Secret 생성 및 확인

- 명령어로 Secret 만들기
  - \$ kubectl create secret generic my-password --from-literal=password=mysqlpassword
  - my-password라는 Secret을 생성하고, password라는 key와 mysqlpassword라는 value 값을 가지게 된다.
  - Value는 base64로 자동 encoding
  - **generic** : create a secret from a local file, directory or literal value
- Secret 확인 : kubectl get secret my-password -o yaml
  - echo [base64 value] | base64 --decode

# Secret을 직접 만들기

- base64 형태로 인코딩하여 YAML파일내에 직접 생성 가능

```
$ echo mysqlpassword | base64  
bXIzcWxwYXNzd29yZAo=
```

- 위 방식으로 인코딩 된 정보를 사용해 설정파일 생성

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: my-password  
type: Opaque  
data:  
  password: bXIzcWxwYXNzd29yZAo=
```

- base64** 인코딩은 바로 디코딩 됨으로 주의!

```
$ echo "bXIzcWxwYXNzd29yZAo=" | base64 --decode  
설정파일을 절대 소스코드에 넣지 않도록 주의한다!
```

# Pod에서 Secret 사용하기

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: hello-secret-deployment
spec:
  replicas: 1
  minReadySeconds: 5
  selector:
    matchLabels:
      app: hello-secret-literal
  template:
    metadata:
      name: hello-secret-literal-pod
      labels:
        app: hello-secret-literal
    spec:
      containers:
        - name: hello-secret
          image: uengineorg.azurecr.io/hello-secret:v1
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          env:
            - name: LANGUAGE
              valueFrom:
                secretKeyRef:
                  name: hello-secret
                  key: language
```

- Deployment.yaml 생성/ 실행
- kubectl create -f hello-secret-deployment.yaml
- \$ kubectl get deploy

# Pod에서 Secret 파일 마운트 사용하기

- 사용자 id를 저장한 user.property파일과 비밀번호를 저장한 password.property 파일생성
  - id file 내용 : \$ cat user.property \$ admin
  - password file 내용 : \$ cat password.property \$ adminpassword
- Secret 생성
  - kubectl create secret generic db-password --from-file=./user.property --from-file=./password.property
  - 생성된 secret은 user.property, password.property 파일명을 각각 key로 파일의 내용이 저장
- Secret을 읽어 출력할 어플리케이션 생성 : server.js

```
var os = require('os');
var fs = require('fs');
var http = require('http');
var handleRequest = function(request, response) {
  fs.readFile('/tmp/db-password/user.property',function(err,userid){
    response.writeHead(200);
    response.write("user id is "+userid+" \n");
    fs.readFile('/tmp/db-password/password.property',function(err,password){
      response.end(" password is "+password+ "\n");
    })
  })
  console.log("[ " +
  Date(Date.now()).toLocaleString()+" ] "+os.hostname());
}
var www = http.createServer(handleRequest);
www.listen(8080);
```

# Pod에서 Secret 파일 마운트 사용하기

- Dockerfile 생성

```
FROM node:carbon
EXPOSE 8080
COPY server.js .
CMD node server.js > log.out
```

- Dockerizing & Azure Container Registry에 Push

```
$ docker build -t uengineorg.azurecr.io/hello-secret-file:v1 .
$ docker images
$ docker push uengineorg.azurecr.io/hello-secret-file:v1
```

- 인증 오류 발생시, 로그인 수행 : az acr login --name *uengineorg*
- Push 확인 : Azure Portal > 컨테이너 레지스터리 > 리파지토리

# Pod에서 Secret 파일 마운트 사용하기

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: hello-secret-file-deployment
spec:
  replicas: 3
  minReadySeconds: 5
  selector:
    matchLabels:
      app: hello-secret-file
  template:
    metadata:
      name: hello-secret-file
      labels:
        app: hello-secret-file
    spec:
      containers:
        - name: hello-secret-file
          image: uengineorg.azurecr.io/hello-secret-file:v1
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          volumeMounts:
            - name: db-password
              mountPath: "/tmp/db-password"
              readOnly: true
      volumes:
        - name: db-password
          secret:
            secretName: db-password
            defaultMode: 0600
```

- Deployment(hello-secret-file-deployment.yaml) 생성/ 실행
- kubectl create -f hello-secret-file-deployment.yaml
- \$ kubectl get deploy

# Pod에서 Secret 파일 마운트 사용하기

```
apiVersion: v1
kind: Service
metadata:
  name: hello-secret-file-service
spec:
  selector:
    app: hello-secret-file
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: 8080
  type: LoadBalancer
```

- Service(hello-secret-file-service.yaml) 생성  
\$ kubectl create -f hello-secret-file-service.yaml
- \$ kubectl get svc
- 브라우저를 통해 서비스 확인
  - Service의 External-IP 접속

# Lab. Secret



Lab Time

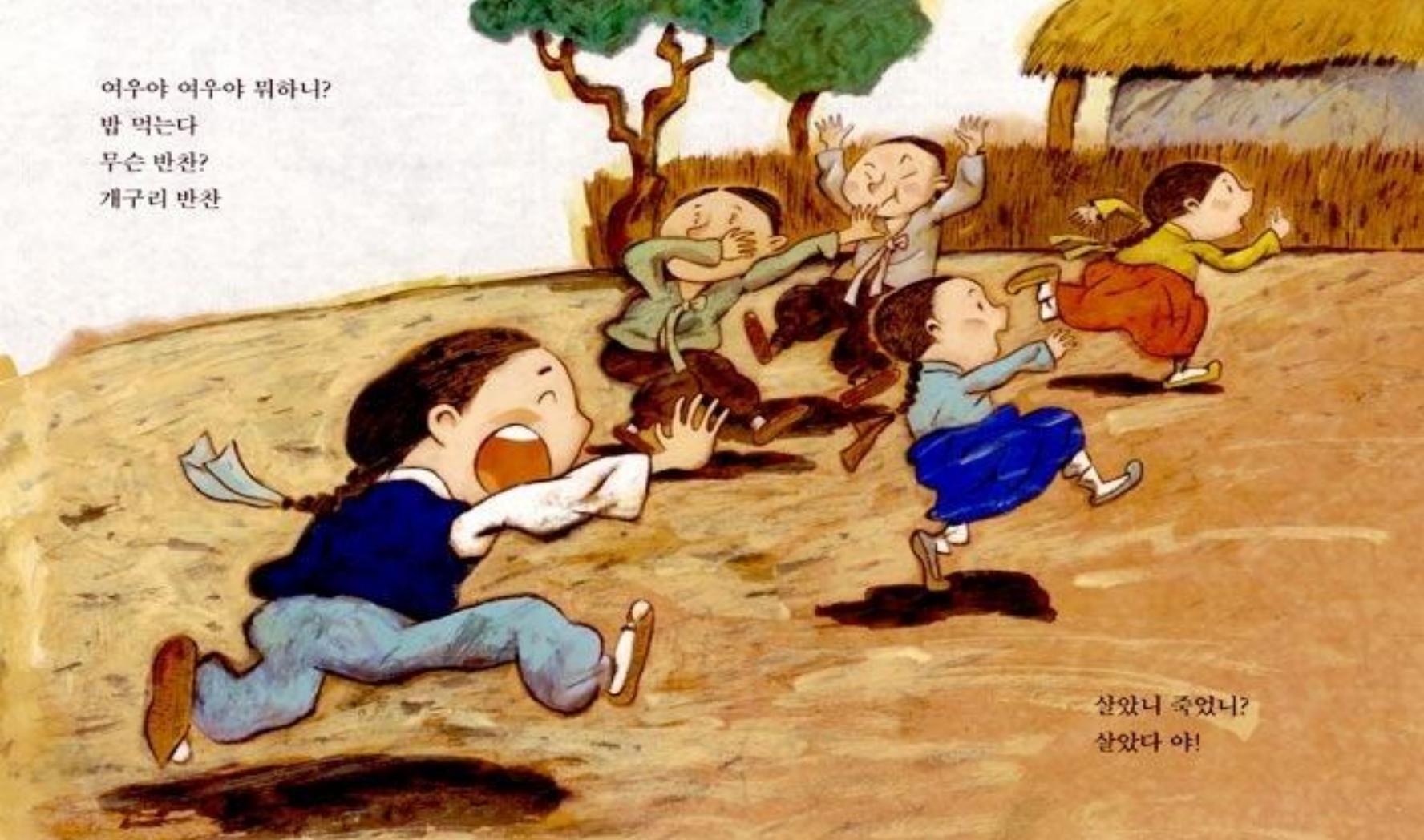
- Lab Script Location
  - Workflowy :

여우야 여우야 뭐하니?

밥 먹는다

무슨 반찬?

개구리 반찬



살았니 죽었니?

살았다 야!

# Live ness Probes & Readiness Probes

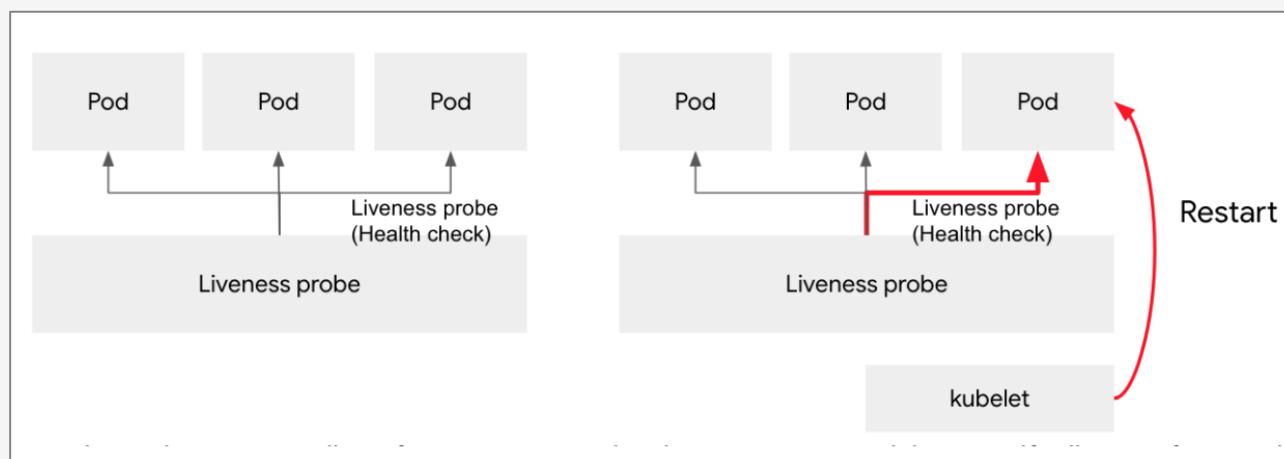
- 쿠버네티스는 각 컨테이너의 상태를 주기적으로 체크(Health Check)해서,
  - 문제가 있는 컨테이너를 자동으로 재시작하거나 또는 문제가 있는 컨테이너를 서비스에서 제외 한다.
- Liveness와 Readiness Probes은 kubelet이 pod내에서 실행되는 어플리케이션의 health를 조정하기 때문에 매우 중요하다.

# Probe Types

- Liveness probe와 readiness probe는 컨테이너가 정상적인지 아닌지를 체크하는 방법으로 다음과 같이 3가지 방식을 제공한다.
  - Command probe
  - HTTP probe
  - TCP probe

# Liveness Probes

- Pod는 정상적으로 작동하지만 내부의 어플리케이션이 반응이 없다면, 컨테이너는 의미가 없다.
  - 위와 같은 경우는 어플리케이션의 Deadlock 또는 메모리 과부화로 인해 발생할 수 있으며, 발생했을 경우 컨테이너를 다시 시작해야 한다.
- Liveness probe는 Pod의 상태를 체크하다가, Pod의 상태가 비정상인 경우 kubelet을 통해서 재시작한다.



# Liveness Command probe

```
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
```

- 왼쪽은 /tmp/healthy 파일이 존재하는지 확인하는 설정파일이다.
- periodSeconds 파라미터 값으로 5초마다 해당 파일이 있는지 조회한다.
- initialDelaySeconds 파라미터는 kubelet이 첫 체크하기 전에 기다리는 시간을 설정한다.
- 파일이 존재하지 않을 경우, 정상 작동에 문제가 있다고 판단되어 kubelet에 의해 자동으로 컨테이너가 재시작 된다.

# Liveness HTTP probe

- Kubelet이 HTTP GET 요청을 /healthz 로 보낸다.
- 실패 했을 경우, kubelet이 자동으로 컨테이너를 재시작 한다.

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

# Liveness TCP Probe

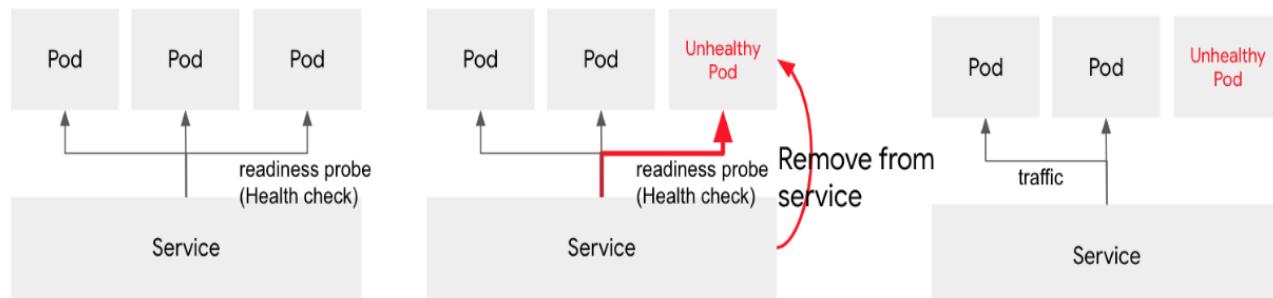
- kubelet은 TCP Liveness Probe를 통해, 지속적으로 어플리케이션이 실행중인 컨테이너의 TCP Socket을 열려고 한다.
- 정상이 아닌 경우 컨테이너를 재시작 한다.

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

# Readiness Probes

- Configuration을 로딩하거나, 많은 데이터를 로딩하거나, 외부 서비스를 호출하는 경우에는 일시적으로 서비스가 불가능한 상태가 될 수 있다.
- Readiness Probe를 사용하게 되면 주어진 조건이 만족할 경우, 서비스 라우팅하고, 응답이 없거나 실패한 경우, 서비스 목록에서 제외

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



# Difference between Liveness and Readiness

- Liveness probe와 Readiness probe 차이점은
  - Liveness probe는 컨테이너의 상태가 비정상이라고 판단하면,  
→ 해당 Pod를 재시작하는데 반해,
  - Readiness probe는 컨테이너가 비정상일 경우에는  
→ 해당 Pod를 사용할 수 없음으로 표시하고, 서비스등에서 제외한다.
  - 주기적으로 체크하여, 정상일 경우 정상 서비스에 포함

# Lab. Liveness 와 readiness probe

- exec-liveness 설정 파일 생성
  - nano exec-liveness.yaml
- 파일 설정으로 배포
  - kubectl create -f exec-liveness.yaml
- 결과 확인
  - \$ tmux #실행, split window on tmux
  - watch -n 1 kubectl get all
  - 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
```

# Lab. Liveness 와 readiness probe

- http-liveness 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
```

# Lab. 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
```

# Lab. Liveness, Readiness



Lab Time

- Lab Script Location
  - Workflowy :

# Table of content

Container Orchestration  
(Docker & k8s)

1. MSA, Container, and Container Orchestration
2. Setup Azure Platform
3. Docker / Kubernetes, Kubernetes Architecture
4. Kubernetes Basic Object Model : Pod, Label, ReplicaSet, Deployment, Service, Volume, Configmap, Secret, Liveness/Readiness
5. Kubernetes Advanced Lab : Ingress, Job, Cron Job, DaemonSet, StatefulSet ✓
6. Service Mesh: Istio for Advanced Services Control
7. Course Test

# Ingress

- Service는 L4 레이어로 TCP레벨에서 Pod를 로드밸런싱 함
- MSA에서는 Service 하나가 MSA의 서비스로 표현되는 경우가 많고 서비스는 하나의 URL(/orders, /products, ...)로 대표되는 경우가 많다.
- MSA 서비스간 라우팅을 위해 API Gateway를 두는 경우가 많은데 관리포인트가 생김
- URL기반의 라우팅 정도라면 L7 로드밸런서 정도로 위의 기능을 충족
- Kubernetes에서 제공하는 L7 로드밸런싱 컴포넌트를 ‘Ingress’ 라고 함

kubernetes.io의 의하면,

*"An Ingress is a collection of rules that allow inbound connections to reach the cluster Services."*

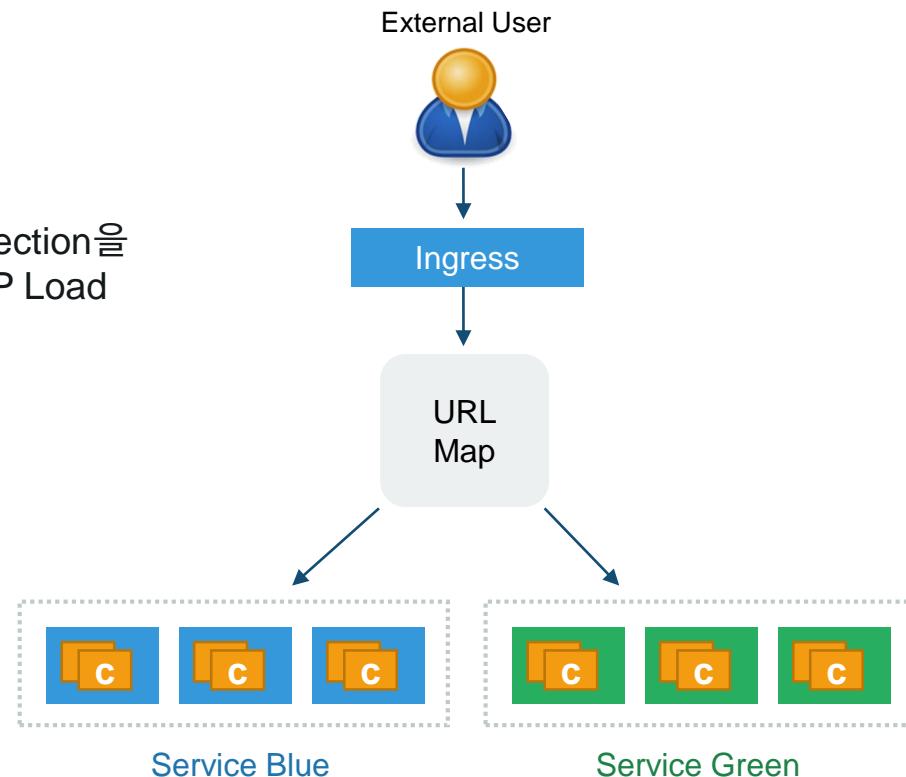
"Ingress는 인바운드 연결이 클러스터의 Service에게 라우팅되도록 하는 규칙의 집합체이다."



# Ingress

- 아래와 같은 Service들의 Inbound Connection을 지원하기 위해 Ingress는 Layer7의 HTTP Load balancer 기능 제공

- TLS (SSL)
- Name-based virtual hosting
- Path-based routing
- Custom rules

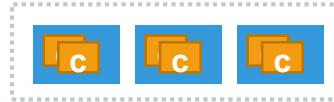


# Ingress

## URL Path based

example.com/blue  
example.com/green

Ingress Controller



Service Blue



Service Green

## Virtual Hosting

blue.example.com  
green.example.com

Ingress Controller



Service Green



Service Blue

- 사용자들은 직접 Service에 접속하지 않는다.
- 유저는 Ingress에 먼저 접근하고, 요청은 해당 Service로 포워드 된다.
- Ingress 요청은 Ingress Controller에 의해 처리된다.

# Ingress Controller

- "Ingress Controller"는 Ingress 리소스의 변경 사항을 마스터 노드의 API 서버에서 감시하고, 그에 따라 Layer 7로드 밸런서를 업데이트하는 응용 프로그램이다.
- Ingress Controller는 오픈소스 기반 구현체 및 클라우드 벤더사가 직접 구현체를 개발해 사용하기도 함
  - Nginx Ingress Controller, KONG, GCE L7 Load Balancer

# Ingress Routing

- Host-based Routing
  - 사용자가 blue.example.com 와 green.example.com에 접근을 하게 되면 같은 Ingress endpoint에서,
  - 각각 nginx-blue-svc와 nginx-green-svc로 요청이 포워드

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: web-ingress
  namespace: ingress-basic
spec:
  rules:
    - host: blue.example.com
      http:
        paths:
          - backend:
              serviceName: nginx-blue-svc
              servicePort: 80
    - host: green.example.com
      http:
        paths:
          - backend:
              serviceName: nginx-green-svc
              servicePort: 80
```

<Host-based routing>

# Ingress Routing

- Path-based Routing
  - Ingress는 또한 example.com/blue 와 example.com/green 형태의 요청에 대해,
  - 각각 nginx-blue-svc와 nginx-green-svc로 요청이 라우팅

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: web-ingress
  namespace: ingress-basic
spec:
  rules:
    - http:
        paths:
          - path: /blue/*
            backend:
              serviceName: nginx-blue-svc
              servicePort: 80
          - path: /green/*
            backend:
              serviceName: nginx-green-svc
              servicePort: 80
```

<Path-based routing>

# Ingress target ServiceType : NodePort

- Ingress에서 접속하는 서비스의 ServiceType 지정 시, LoadBalancer나 ClusterIP가 아닌 NodePort 타입을 사용하는 이유는,
  - Ingress로 사용되는 로드밸런서에서, 각 서비스에 대한 Heartbeat 체크를 하기 위함
  - Ingress로 배포된 구글 클라우드 로드밸런서는 각 노드에 대해서 Node port로 Heartbeat 체크를 수행하고, 문제 있는 노드를 로드밸런서에서 자동으로 제거하거나, 복구가 되었을 때 자동으로 추가함

“

# Ingress Controller Setup for Azure Ingress Practice

- Install Helm to install Nginx-ingress Controller
- Install Ingress Controller & overview

# Install Helm

- Kubernetes 패키지 인스톨러 ‘Helm’ 설치

- \$ curl <https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get> | bash
- sudo 실행을 위한 계정 Password 입력

```
apexacme@APEXACME: /yaml/ingress/green-svc$ curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get | bash
% Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload Total Spent   Left Speed
100  7150  100  7150    0     0  8392      0 --:--:--:--:--:-- 8392
Downloaded https://get.helm.sh/helm-v2.16.3-linux-amd64.tar.gz
Preparing to install helm and tiller into /usr/local/bin
[sudo] password for apexacme: ■
```

- Helm 초기화

- kubectl --namespace kube-system create sa tiller # helm 의 설치관리자를 위한 시스템 사용자 생성
- kubectl create clusterrolebinding tiller --clusterrole cluster-admin --serviceaccount=kube-system:tiller
- helm repo update
- helm init --kube-context My-Cluster --service-account tiller
  - # Kube-context → \$ kubectl config current-context 명령을 통해 알 수 있음

# Install Helm

- Helm 설치 확인
  - Helm client와 쿠버네티스 api와 통신 역할을 담당하는 Tiller Server 설치 메시지 확인

```
apexacme@APEXACME:~/yaml/ingress/green-svc$ kubectl config current-context
My-Cluster
apexacme@APEXACME:~/yaml/ingress/green-svc$ helm init --kube-context My-Cluster --service-account tiller
Creating /home/apexacme/.helm
Creating /home/apexacme/.helm/repository
Creating /home/apexacme/.helm/repository/cache
Creating /home/apexacme/.helm/repository/local
Creating /home/apexacme/.helm/plugins
Creating /home/apexacme/.helm/starters
Creating /home/apexacme/.helm/cache/archive
Creating /home/apexacme/.helm/repository/repositories.yaml
Adding stable repo with URL: https://kubernetes-charts.storage.googleapis.com
Adding local repo with URL: http://127.0.0.1:8879/charts
$HELM_HOME has been configured at /home/apexacme/.helm.

Tiller (the Helm server-side component) has been installed into your Kubernetes Cluster.

Please note: by default, Tiller is deployed with an insecure 'allow unauthenticated users' policy.
To prevent this, run `helm init` with the --tiller-tls-verify flag.
For more information on securing your installation see: https://docs.helm.sh/using_helm/#securing-your-helm-installation
apexacme@APEXACME:~/yaml/ingress/green-svc$
```

# Install Ingress Controller

- Ingress Controller 설치
  - \$ helm install --name nginx-ingress stable/nginx-ingress --kube-context=My-Cluster --namespace=ingress-basic
- 설치 확인
  - kubectl get all --namespace=ingress-basic

```
apexacme@APEXACME:~/yaml/ingress/green-svc$ kubectl get all --namespace=ingress-basic
NAME                                         READY   STATUS    RESTARTS   AGE
pod/nginx-ingress-controller-5b85669986-qaghk   1/1     Running   0          98s
pod/nginx-ingress-default-backend-6b8dc9d88f-c4q1t   1/1     Running   0          98s

NAME                           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
service/nginx-ingress-controller   LoadBalancer   10.0.62.84    52.231.110.3   80:30587/TCP,443:31904/TCP   98s
service/nginx-ingress-default-backend   ClusterIP    10.0.240.124  <none>        80/TCP          98s

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx-ingress-controller   1/1     1           1           98s
deployment.apps/nginx-ingress-default-backend   1/1     1           1           98s

NAME                               DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-ingress-controller-5b85669986   1         1         1         98s
replicaset.apps/nginx-ingress-default-backend-6b8dc9d88f   1         1         1         98s
apexacme@APEXACME:~/yaml/ingress/green-svc$
```

- (삭제 필요시,) \$ helm delete --purge nginx-ingress --kube-context=My-Cluster
- Ingress Controller의 EXTERNAL-IP가 API Gateway 엔드포인트 (메모 必)

# Lab. Ingress

- Scenario
  - 대상 이미지 생성 및 ACR Push
  - 컨테이너 및 Service 생성
    - nginx-blue-deployment, nginx-blue-service
    - Nginx-green-deployment, nginx-green-service
  - Ingress 생성 : Host-based Routing Rule
  - 테스트
    - Local Machine(Windows) Hosts 등록
    - 브라우저로 도메인 별 접속

# Lab. Ingress

- 대상 이미지 생성 및 ACR Push

- Dockerfile 생성

```
FROM nginx
COPY index.html /usr/share/nginx/html
```

- Simple application - index.html

```
$ cat index.html
Hi, This is nginx-Blue-svc
```

- Dockerizing

```
$ docker build -t uengineorg.azurecr.io/nginx-blue:latest .
```

- ACR Push

```
$ docker push uengineorg.azurecr.io/nginx-blue:latest
```

- 컨테이너 및 Service 생성

- 한 파일에 두 객체 선언을 --- 로 연결

```
$ kubectl create -f nginx-blue-deployment.yaml
```



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-blue-deployment
  namespace: ingress-basic
  labels:
    app: blue-nginx-deploy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: blue-nginx
  template:
    metadata:
      labels:
        app: blue-nginx
    spec:
      containers:
        - name: nginx
          image: uengineorg.azurecr.io/nginx-blue:latest
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-blue-svc
  namespace: ingress-basic
spec:
  selector:
    app: blue-nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      type: NodePort
```

# Lab. Ingress

- Azure Container Registry(ACR) 확인
  - Azure 포탈 접속

The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with '컨테이너 레지스트리' (Container Registry). The main area is titled 'uengineorg - 리포지토리' (Repository). It displays a list of Docker images: 'cm-sandbox', 'nginx', 'nginx-blue', and 'nginx-green'. The 'nginx' image is circled in red, indicating it is the current selection.

- Deployment, Service 생성 확인 : \$ kubectl get deploy,service -n ingress-basic

```
apexacme@APEXACME:~/yaml/ingress$ kubectl get deployment,service -n ingress-basic
NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.extensions/nginx-blue-deployment   1/1     1           1          97m
deployment.extensions/nginx-green-deployment  1/1     1           1          93m
deployment.extensions/nginx-ingress-controller 1/1     1           1          173m
deployment.extensions/nginx-ingress-default-backend 1/1     1           1          173m

NAME                           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
service/nginx-blue-svc         NodePort    10.0.235.109  <none>       80:30211/TCP   97m
service/nginx-green-svc        NodePort    10.0.34.4     <none>       80:30381/TCP   93m
service/nginx-ingress-controller LoadBalancer 10.0.224.204  52.141.28.64  80:31918/TCP,443:32487/TCP 173m
service/nginx-ingress-default-backend ClusterIP 10.0.12.239  <none>       80/TCP      173m
```

# Lab. Ingress

- Ingress 생성 : Host-based Routing Rule
  - \$ kubectl create -f web-ingress.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: web-ingress
  namespace: ingress-basic
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
  - host: blue.example.com
    http:
      paths:
      - backend:
          serviceName: nginx-blue-svc
          servicePort: 80
  - host: green.example.com
    http:
      paths:
      - backend:
          serviceName: nginx-green-svc
          servicePort: 80
```

- 생성 확인 : \$ kubectl get ingress -n ingress-basic

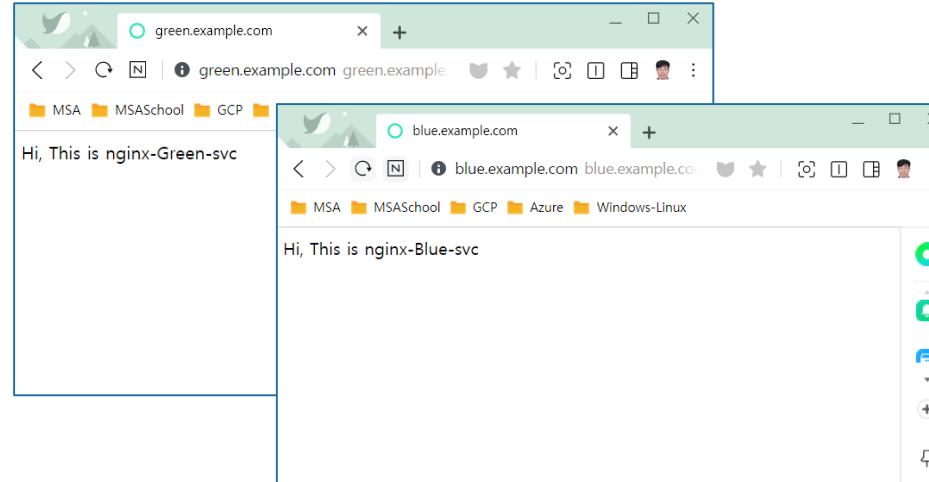
# Lab. Ingress

- 테스트

- API Gateway 주소를 Local 시스템에 등록
  - PowerShell 관리자 권한으로 실행
  - cd c:\windows\system32\drivers\etc
  - notepad hosts 를 실행해, 맨 하단에 Ingress Controller의 External-IP 등록

```
52.141.28.64      blue.example.com  green.example.com
```

- 도메인별 Routing 확인



# Lab. Ingress

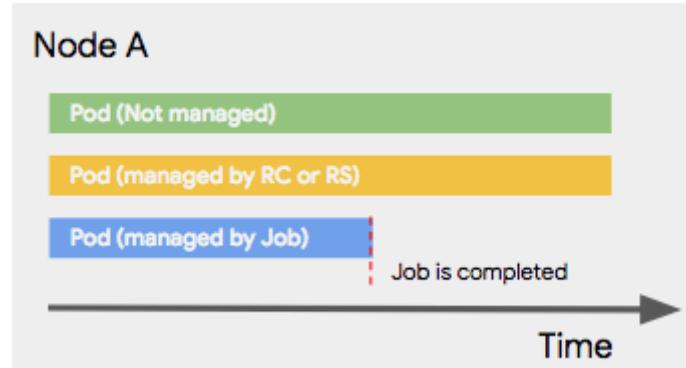


Lab Time

- Lab Script Location
  - Workflowy :

# Jobs

- 워크로드 모델 중, 배치나 한번 실행되고 끝나는 형태의 작업이 있을 수 있다.
- 예로, 원타임으로 파일 변환 작업을 하거나, 주기적으로 ETL 배치 작업을 하는 경우, Pod가 계속 떠 있을 필요 없이 작업을 할 때만 Pod를 실행한다.
- Job은 이러한 워크로드 모델을 지원하는 Controller



# Jobs

- Job에 의해 관리되는 Pod는 Job이 종료되면 Pod도 같이 종료
- Job정의 시, Container Spec에 image뿐만 아니라, Job을 수행하기 위한 커맨드를 같이 입력

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle",
                     "print bpi(2000)"]
      restartPolicy: Never
      backoffLimit: 4
```

Job실패시, 처음부터 재시작 하지않음  
Job실패시, 4번까지 재시도

# Cron Jobs

- Job 컨트롤러에 의해 실행되는 작업을 주기적으로 스케줄링 해 주는 컨트롤러

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
  restartPolicy: OnFailure
```

# Resource Assign & Management

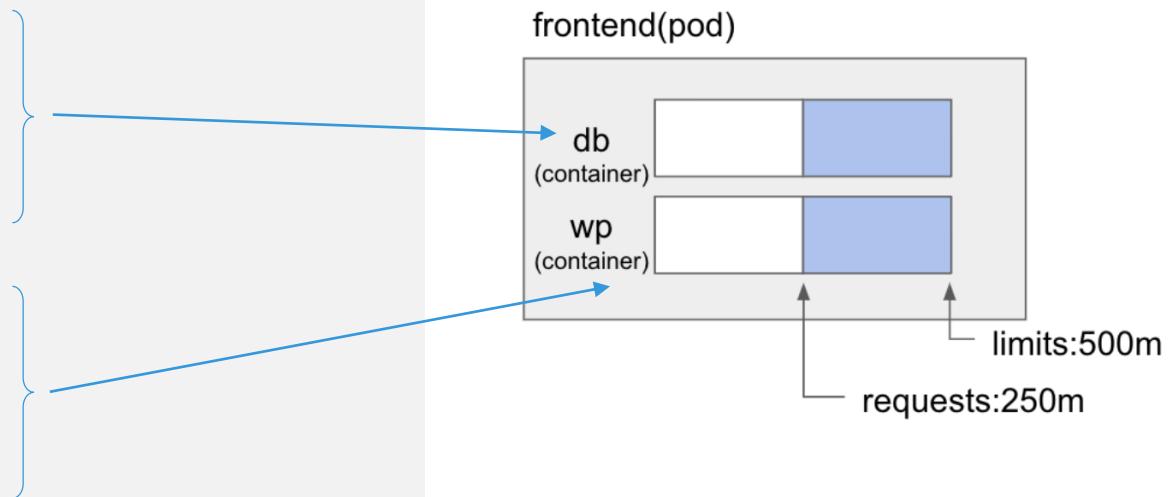
- 쿠버네티스에서 Pod를 어느 노드에 배포할지 결정하는 것을 스케줄링이라 함
- Pod에 대한 스케줄링시에, Pod내의 애플리케이션이 동작할 수 있는 자원(CPU,메모리 등) 정보를 알아야 그만한 자원이 가용한 노드에 Pod 배포 가능
- 리소스 단위
  - CPU의 경우 ms(밀리 세컨드)를 사용하는데 대략 1000ms가 1 vCore (가상 CPU 코어)
  - 메모리의 경우 Mb를 사용하며 64M( $64 \times 1000$ ), 또는 64Mi ( $64 \times 1024$ )로 계산
  - Request & Limit
    - Request : 컨테이너가 생성될 때 요청하는 리소스 양
    - Limit : 리소스가 더 필요한 경우 추가로 사용 가능한 양



# Resource Assign & Management

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
    - name: db
      image: mysql
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
    - name: wp
      image: wordpress
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
```

- 샘플 설정에 따른 Pod내 CPU 리소스 할당



# Resource Monitoring

- Node의 자원 상태 모니터링
  - \$ kubectl get nodes
  - \$ kubectl describe nodes

Namespace	Name	CPU Requests	CPU Limits	Memory Requests	Memory Limits	AGE
ingress-basic	nginx-ingress-controller-5b85669986-g4n8t	0 (0%)	0 (0%)	0 (0%)	0 (0%)	5h14m
ingress-basic	nginx-ingress-default-backend-6b8dc9d88f-1bd9g	0 (0%)	0 (0%)	0 (0%)	0 (0%)	5h14m
kube-system	kube-proxy-47g4j	100m (5%)	0 (0%)	0 (0%)	0 (0%)	2d2h
kube-system	omsagent-8cr5p	75m (3%)	150m (7%)	225Mi (4%)	600Mi (13%)	7d
kube-system	tiller-deploy-7b98f7c844-t7cpn	0 (0%)	0 (0%)	0 (0%)	0 (0%)	6h55m
kube-system	tunnelfront-c8cdf6fcf-c7xz2	10m (0%)	0 (0%)	64Mi (1%)	0 (0%)	7d

Allocated resources:	
(Total limits may be over 100 percent, i.e., overcommitted.)	
Resource	Requests      Limits
cpu	185m (9%)    150m (7%)
memory	289Mi (6%)    600Mi (13%)
ephemeral-storage	0 (0%)      0 (0%)
attachable-volumes-azure-disk	0      0

Events:	
<none>	

```
apexacme@APEXACME: ~/yaml/ingress$
```

- 현재 사용 중인 리소스 현황 모니터링
  - \$ kubectl top nodes
  - \$ kubectl top pods

# Resource Quota

- ResourceQuota는 네임스페이스별로 사용 가능한 리소스 양을 정의

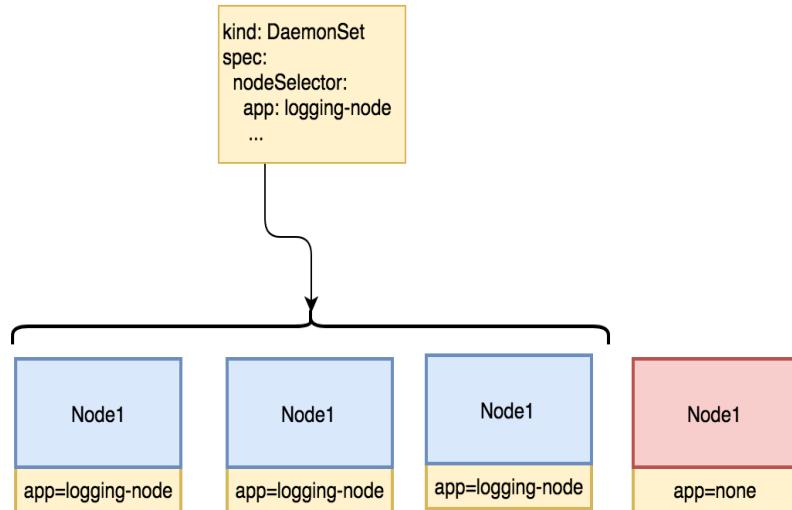
```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-demo
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

<quota-mem-cpu.yaml>

- 네임스페이스에 ResourceQuota 적용
  - kubectl apply -f quota-mem-cpu.yaml --namespace=a-namespace

# DaemonSets

- Pod를 각각의 노드에서 하나씩만 돌게 하는 형태로 Pod를 관리하는 컨트롤러
- 모니터링 데이터를 수집하거나, 스토리지 데몬을 실행하는 등, 모든 노드에서 항상 실행되는 특정 유형의 포드가 필요할 경우에 사용
  - Node 가 클러스터에 추가되면, 주어진 DaemonSet에 의해 Pod가 생성
  - DaemonSet이 삭제 되면, 관련된 모든 Node의 Pod들은 삭제
- 또한 특정 노드에만 Pod를 배포할 수 있도록, Pod의 “node selector”를 이용해서 라벨을 필터링하여 특정 노드만 선택 가능



# StatefulSets

- 이전의 컨트롤러(Replica Set, Job) 같은 상태가 유지되지 않는 application을 관리하는 용도지만, StatefulSet은 단어의 의미 그대로 상태를 가지고 있는 포드들을 관리하는 컨트롤러
- 스테이트풀셋을 사용하면 볼륨을 사용해서 특정 데이터를 기록해두고 그걸 포드가 재시작했을 때도 유지할 수 있음
- StatefulSet controller는 이름, 네트워크 인증, 엄격한 순서 등의 독자성이 보장 되어야 할 때 사용
  - RS에 의해서 관리되는 Pod들은 기동이 될때 병렬로 동시에 기동 되나, DB의 경우에는 Master 노드가 기동 된 다음에, Slave 노드가 순차적으로 기동되어야 하는 순차성을 가지고 있는 경우가 있음
  - Ex) MySQL cluster, etcd cluster.
- 여러 개의 포드를 띄울 때 포드 사이에 순서를 지정해 지정된 순서대로 포드 실행 가능

# StatefulSets

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: "nginx"
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: nginx
          image: k8s.gcr.io/nginx-slim:0.8
          ports:
            - containerPort: 80
              name: web
          volumeMounts:
            - name: www
              mountPath: /usr/share/nginx/html
      volumeClaimTemplates:
        - metadata:
            name: www
      spec:
        accessModes: [ "ReadWriteOnce" ]
        storageClassName: "standard"
        resources:
          requests:
            storage: 1Gi
```

- \$ kubectl get pod

NAME	READY	STATUS	RESTARTS	AGE
nginx-0	1/1	Running	0	13m
nginx-1	1/1	Running	0	12m
nginx-2	1/1	Running	0	12m

- \$ kubectl get pvc

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
www-nginx-0	Bound	pvc-e70627ea-2ecd-11e9-8d43-42010a920009	1Gi	RWO	standard	12m
www-nginx-1	Bound	pvc-f5f2d9d9-2ecd-11e9-8d43-42010a920009	1Gi	RWO	standard	12m
www-nginx-2	Bound	pvc-003c7376-2ece-11e9-8d43-42010a920009	1Gi	RWO	standard	12m

- StatefulSet은 Pod를 생성할 때 순차적으로 기동되고, 삭제할 때도 순차적으로(2 → 1 → 0) 삭제 됨

# Kubernetes Federation (deprecated)

- Kubernetes Cluster Federation을 통해서 여러 클러스터를 하나의 Control plane에서 관리
- 클러스터 간 리소스를 연동시키거나 Cross-cluster discovery가 가능
  - DNS record 하나를 사용하여 지역간 배포 가능
- Federation은 하이브리드 솔루션을 만들고자 할 때 매우 유용
  - Multi-Cloud 또는, 비공개 클러스터와 공개 클러스터를 동시에 사용 가능
- 각 클러스터에 가중치를 할당하여 로드 분산 가능

# Helm

- Application 배포에는 여러 Kubernetes manifest들을 사용
  - Deployments, Services, Volume Claims, Ingress, etc.
  - 하나하나 사람이 배포하기에는 힘들다.
- 여러 manifest들을 metadata와 함께 정해진 템플릿으로 묶은 것을 Chart라고 함
- Helm은 위 Chart들을 설치/업데이트/삭제를 할 수 있는 패키지 매니저이다.
- Helm은 두 부분으로 이루어져 있다 :
  - 유저의 Workstation에서 작동하는 Helm 클라이언트
  - Kubernetes Cluster에서 동작하는 Tiller 서버
- Helm(client)가 Chart들을 관리하기 위해서 Tiller(server)에 접속

# Monitoring and Logging

- Kubernetes에서는 Pods, Services, nodes등의 리소스 정보를 수집해야 어플리케이션의 전체적인 리소스 사용량을 알고 Scaling이 가능해진다.
- Kubernetes 모니터링 솔루션 중에 인기 많은 것이 Heapster와 Prometheus이다.
  - Heapster  
는 Kubernetes에서 기본적으로 제공이 되며 클러스터 내의 모니터링과 이벤트 데이터를 수집한다.
  - Prometheus  
는 CNCF에 의해서 제공이 되며, Kubernetes의 각 다른 객체와 구성으로부터 리소스 사용을 수집할 수 있다. Client libraries를 사용하여 어플리케이션의 코드도 조정도 할 수 있다.
- 문제 해결과 디버깅의 또 하나의 중요한 관점은 Logging이다.
  - Kubernetes는 각 다른 객체에서 생성되는 로그들을 수집할 수 있다.
  - 로그를 수집하는 가장 흔한 방법은 fluentd를 사용하는 Elasticsearch이다. Fluentd는 node에서 에이전트로 작동을 하며 커스텀 설정이 가능

# Real MSA Application Deployment

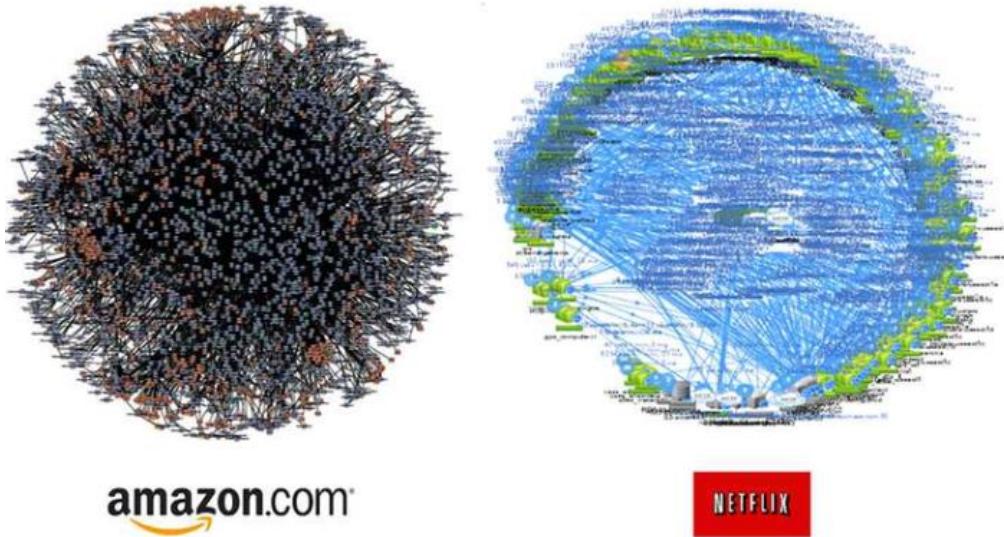


# Table of content

Container Orchestration  
(Docker & k8s)

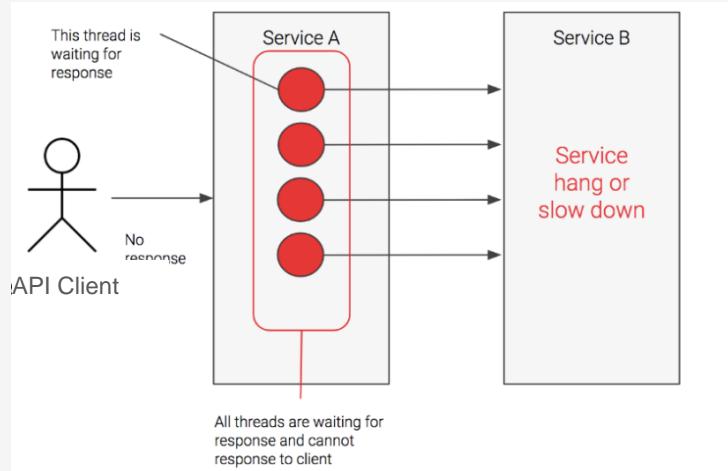
1. MSA, Container, and Container Orchestration
2. Setup Azure Platform
3. Docker / Kubernetes, Kubernetes Architecture
4. Kubernetes Basic Object Model : Pod, Label, ReplicaSet, Deployment, Service, Volume, Configmap, Secret, Liveness/Readiness
5. Kubernetes Advanced Lab : Ingress, Job, Cron Job, DaemonSet, StatefulSet
6. Service Mesh: Istio for Advanced Services Control ✓
7. Course Test

# Microservices



- Microservices Architecture는 서비스 간의 복잡한 연결과 상호 호출구조를 가짐
- 서비스간 전체 연결구조 파악이 어려우며, 장애 발생 시, 정확한 추적에 많은 비용 필요
- 특정 서비스의 장애가 다른 서비스에 영향을 줄 수 있는 스파게티 네트워크

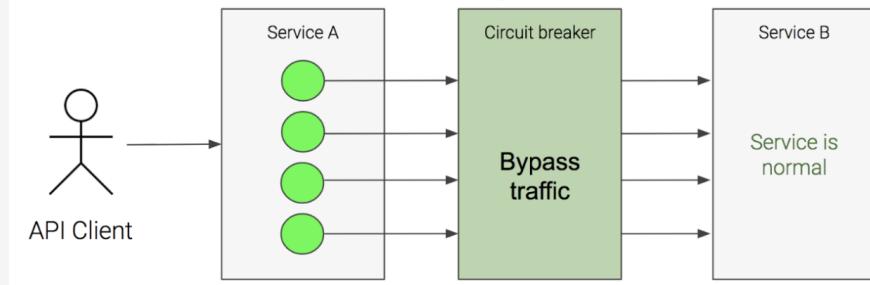
# Fault Spread Issues in Microservices



- 클라이언트 → 서비스 A → 서비스 B 를 호출하는 구조에서
  - 서비스 B가 느려지거나 응답이 없는 장애 상태가 되면,
  - 서비스 B를 호출하는 서비스 A안의 스레드는 대기 상태가 되고,
  - 클라이언트의 호출이 많아지면, 서비스 A의 다른 스레드들도 대기상태가 됨
  - 결국, 서비스 A도 응답할 수 없는 장애 상태로 장애가 전파되는 현상이 발생함

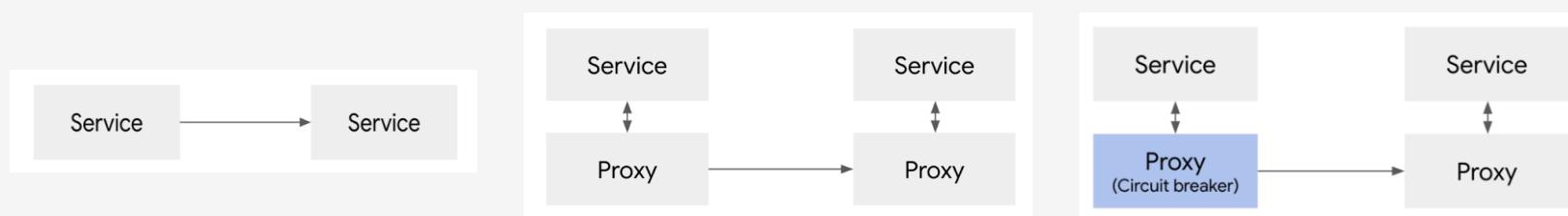
# Circuit Breaker Design pattern

- 장애전파 회피를 위한 써킷 브레이커(Circuit Breaker) 디자인 패턴



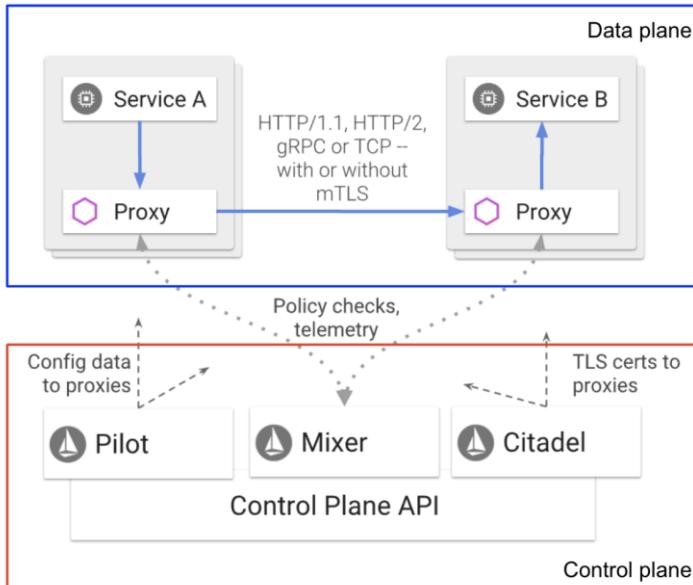
- 서비스 A와 서비스 B 사이에 회로차단기 개념을 적용해 네트워크 트래픽을 통과시키고, 서비스 B 가 장애가 있거나, 응답이 없을 경우 네트워크 연결을 차단해 서비스 A가 응답하는 구조
- マイクロ서비스 디자인 패턴 → <https://microservices.io/> 참조

# Service Call through Proxy



- 서비스를 직접 호출하는 것이 아니라, 서비스마다 프록시(Proxy)를 통한 호출
- 프록시를 통해, 들어오거나 나가는 트래픽을 통제함으로써 Circuit Breaker Pattern 지원
- 클라이언트의 OS나, 브라우저 환경에 따른 스마트 라우팅(L7 Layer)도 프락시에서 통제 가능
  - 기존 Ingress와 같은 API Gateway로는 한계

# Istio Service Mesh architecture



- Data Plane(데이터 플래인)
  - 서비스 옆에 사이드카(Envoy)를 붙여, 서비스로 들어오고 나가는 트래픽을 Envoy를 통해 통제
  - 유입/유출 트래픽 통제 : Ingress/Egress Controller
- Control Plane(컨트롤 플래인)
  - **Pilot(파일럿)** : Envoy에 대한 설정 관리 및 서비스 디스커버리 기능 제공
  - **Mixer(믹서)** : 액세스 컨트롤 및 다양한 모니터링 지표 수집
  - **Citadel(시터덜)** : 보안관련 기능 담당 모듈로 사용자 인증/ 인가 및 TLS 통신을 위한 인증서 관리

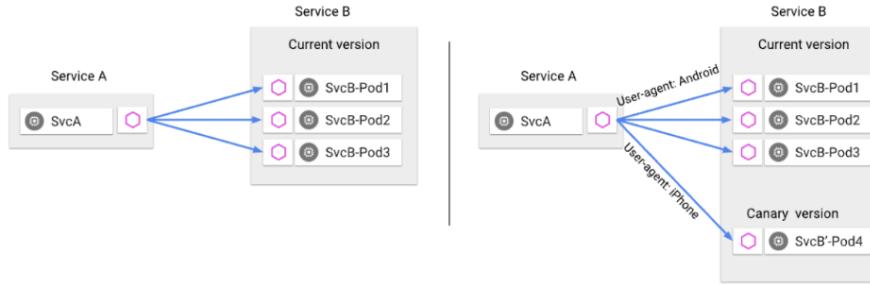
# Istio Service Mesh key features

- Improved Routing & Deployment Strategy
- Improved Smart Resilience
- Improved Security
- Improved Observability



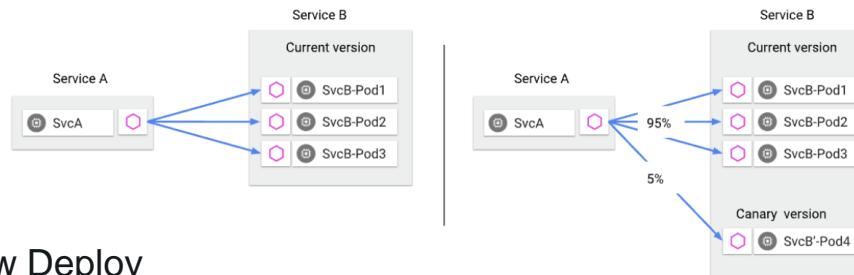
# 1. Improved Routing & Deployment Strategy

- Traffic Routing Controls



- Canary Deploy

- 특정 유저의 신상, 지역, 권한, 접근 단말에 따른 다른 버전의 노출



- AB Testing / Shadow Deploy

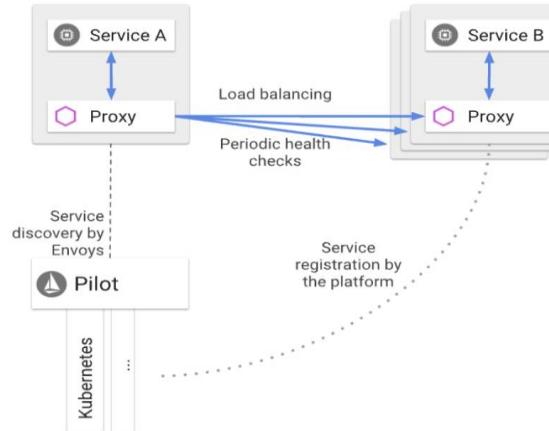
- 신규 버전의 오류 노출 없는 실질적 테스트

## 2. Improved Resilience

- Retry
  - 서비스간 호출의 실패에 대한 재시도
- Circuit Breaking / Rate Limiting
  - 인스턴스의 보호
  - 전체 서비스 장애 차단
- Pool Ejection
  - 죽은 인스턴스의 제외
- Health Check & Service Discovery
  - 여러 인스턴스에 대한 로드 밸런싱
  - 서비스 앤드 포인트 관리



Circuit Breaking + Pool Ejection + Retry  
= High Resilience

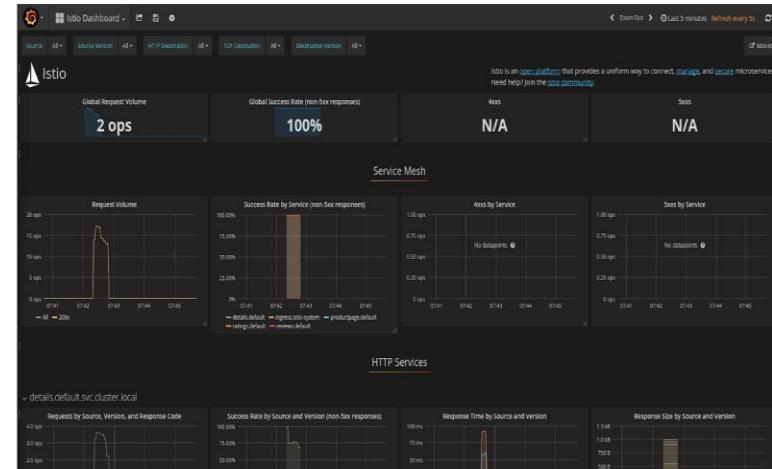
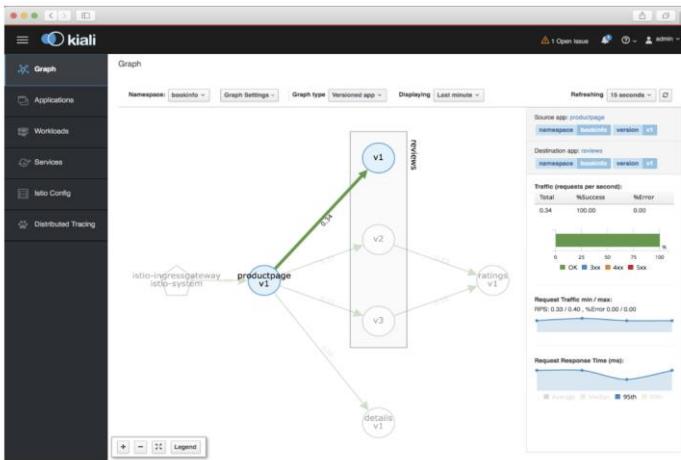


### 3. Improved Security

- TLS based Inter Microservices Communication
  - Envoy로 통신하는 모든 트래픽을 자동으로 TLS를 이용해 암호화
- Service Authentication & Authorization
  - JWT Token을 이용한 서비스 접근 Client 인증
  - 역할기반(RBAC) 권한 인증 지원
- Whitelist and Blacklist

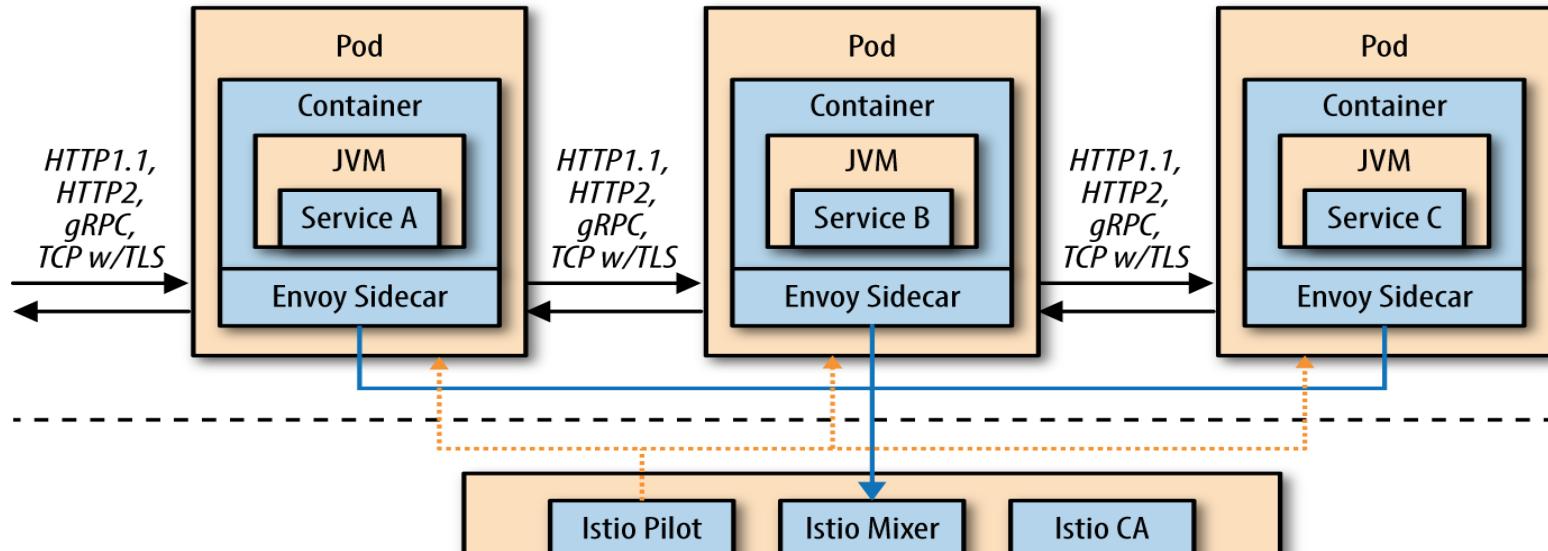
# 4. Improved Observability

- Distributed Tracing and Measure
  - 서비스간 호출 내용 기록
  - Istio 설치 시, Kiali, Jaeger가 default로 설치되어 각각 Monitoring 및 Tracing 지원



# Istio Components

## Data Plane



## Control Plane

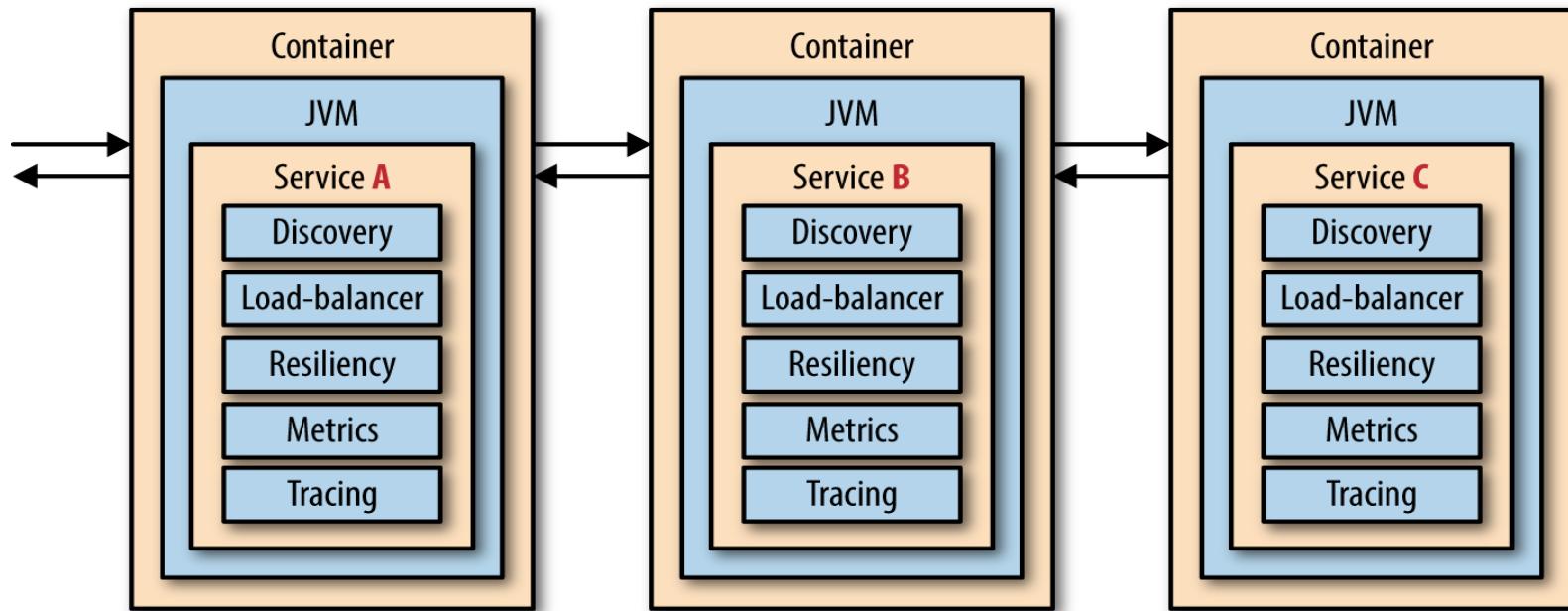
*istioctl, API, config*

*Quota, Telemetry, Rate Limiting, ACL*

*mTLS, SPIFFE*

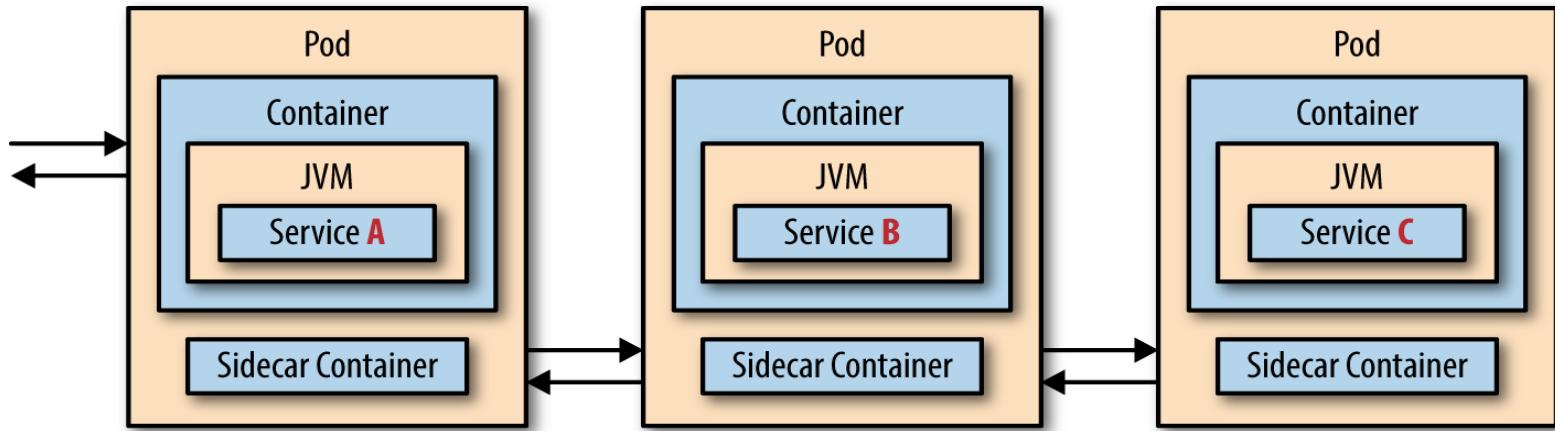
# Before Istio : Spring Cloud + Netflix

Before Istio



# After Istio

Envoy



좋은점 :

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

# Lab. Istio Install

- Istio 설치

- curl -L https://git.io/getLatestIstio | ISTIO\_VERSION=1.4.5 sh -
- cd istio-1.4.5
- 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

- 설치확인

- \$ kubectl get pod -n istio-system



NAME	READY	STATUS	RESTARTS	AGE
grafana-6bb6bcf99f-78p9q	1/1	Running	0	76s
istio-citadel-59759bf4-lpqxc	1/1	Running	0	75s
istio-egressgateway-5c68d8857-qp14x	0/1	Running	0	76s
istio-galley-769849c5fb-cpvkp	0/1	Running	0	76s
istio-grafana-post-install-1.4.5-frqk4	0/1	Completed	0	78s
istio-ingressgateway-665c7c6b8-k9qhs	0/1	Running	0	76s
istio-pilot-6bd54f544b-65tnf	1/2	Running	1	75s
istio-policy-6b7d856769-xcip2	2/2	Running	2	75s
istio-security-post-install-1.4.5-p9wtc	0/1	Completed	0	78s
istio-sidecar-injector-6d9f967b5-v157x	1/1	Running	0	75s
istio-telemetry-657bbb5677-pg2th	2/2	Running	2	75s
istio-tracing-56c7f85df7-71kld8	1/1	Running	0	75s
kiali-7b5c8f79d8-dm46s	1/1	Running	0	76s
prometheus-74d8b55f54-r9w7v	1/1	Running	0	75s

# Lab. Istio Tutorial Setup

- Git repository에서 Tutorial 리소스 가져오기
    - cd ~
    - mkdir git
    - cd git
    - git clone https://github.com/redhat-developer-demos/istio-tutorial
    - cd istio-tutorial
  - 네임스페이스 생성
    - kubectl create namespace tutorial
  - Customer 애플리케이션 배포
    - kubectl apply -f <(istioctl kube-inject -f customer/kubernetes/Deployment.yml) -n tutorial
    - kubectl create -f customer/kubernetes/Service.yml -n tutorial
    - kubectl create -f customer/kubernetes/Gateway.yml -n tutorial
  - 배포 확인 및 외부접속 설정
    - kubectl -n tutorial edit svc customer
    - (ServiceType : ClusterIP → LoadBalancer로 변경)
    - kubectl -n tutorial get svc
    - 브라우저로 EXTERNAL-IP:8080 접속
- | NAME             | TYPE         | CLUSTER-IP  | EXTERNAL-IP    | PORT(S)        | AGE  |
|------------------|--------------|-------------|----------------|----------------|------|
| service/customer | LoadBalancer | 10.0.254.95 | 52.231.113.110 | 8080:30926/TCP | 6m4s |

# Lab. Istio Tutorial Setup

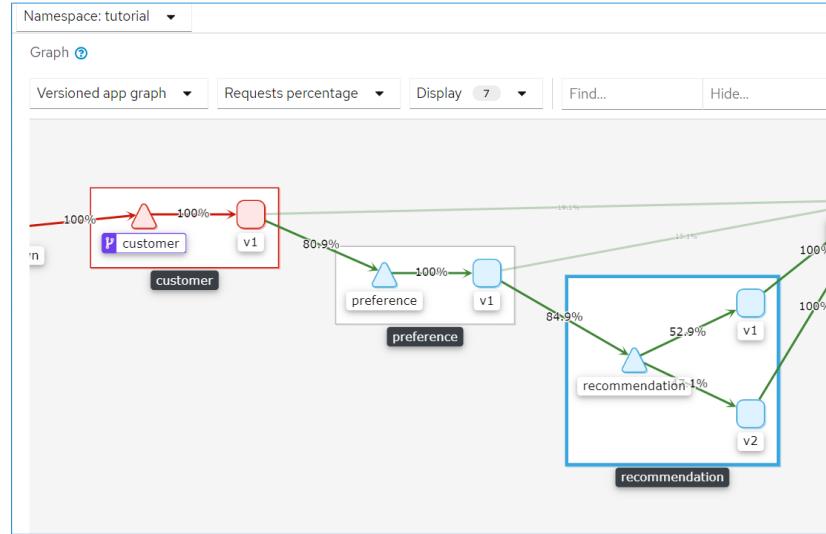
- 테스트 애플리케이션 배포 (preference, recommendation)
  - kubectl apply -f <(istioctl kube-inject -f preference/kubernetes/Deployment.yml> -n tutorial
  - kubectl create -f preference/kubernetes/Service.yml -n tutorial
  - kubectl apply -f <(istioctl kube-inject -f recommendation/kubernetes/Deployment.yml> -n tutorial
  - kubectl create -f recommendation/kubernetes/Service.yml -n tutorial
- 모니터링시스템 외부접속 설정
  - kubectl edit svc jaeger-query -n istio-system
  - (ServiceType : ClusterIP → LoadBalancer로 변경)
  - kubectl edit svc kiali -n istio-system
  - (ServiceType : ClusterIP → LoadBalancer로 변경)
  - kubectl get svc -n istio-system
  - Jaeger 접속 : EXTERNAL-IP :16686
  - Kiali 접속 : EXTERNAL-IP:20001
- 서비스 호출 관계

jaeger-query	LoadBalancer	10.0.181.81	52.231.113.45	16686:32721/TCP
kiali	LoadBalancer	10.0.217.7	40.82.159.29	20001:30150/TCP



# Lab. Istio – Simple Routing (1/2)

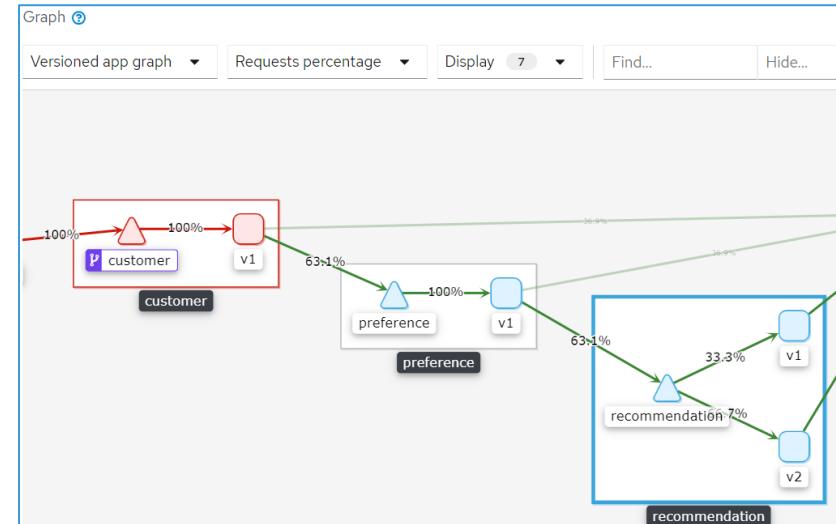
- Istio의 Simple Routing 확인
  - recommendation 서비스 추가 배포 (version. two)
  - kubectl apply -f <(istioctl kube-inject -f recommendation/kubernetes/Deployment-v2.yaml) -n tutorial
  - 서비스 호출
    - 브라우저에서 Customer 서비스(External-IP:8080 접속) 호출
    - F5(새로고침)를 10회 이상 클릭하여 다수의 요청 생성
- Routing 결과 확인
  - Kiali(External-IP:20001) 접속
  - Left 영역 : Graph 선택
  - Main 영역 : Versioned app > Requests percentage
  - Recommendation 서비스의 v1, v2에 균등한 라우팅 (Round Robin) 비율 확인 - 52.9% : 48.1%



# Lab. Istio – Simple Routing (2/2)

- recommendation 서비스 Scale Out
  - 서비스의 v2 의 replica 를 2로 설정
  - kubectl scale --replicas=2 deployment/recommendation-v2 -n tutorial
  - kubectl get po -n tutorial
  - 서비스 호출
    - 브라우저에서 Customer 서비스(External-IP:8080 접속) 호출
    - F5(새로고침)를 10회 이상 클릭하여 다수의 요청 생성

- Routing 결과 확인
  - Kiali(External-IP:20001) 접속
  - Left 영역 : Graph 선택
  - Main 영역 : Versioned app > Requests percentage
  - Recommendation 서비스의 v1, v2에 균등한 라우팅 (Round Robin) 비율 확인 - 33.3% : 66.6%



# Lab. Istio – Improved Routing (1/3)

- 정책(VirtualService, DestinationRule) 설정 실습

- 현, 정책 확인

- kubectl get VirtualService -n tutorial -o yaml
    - kubectl get DestinationRule -n tutorial -o yaml

- recommendation 서비스 라우팅 정책 설정

- VirtualService, DefinitionRule 설정

- kubectl create -f istiofiles/destination-rule-recommendation-v1-v2.yml -n tutorial
    - kubectl create -f istiofiles/virtual-service-recommendation-v2.yml -n tutorial

- 설정정책 확인

- kubectl get VirtualService -n tutorial -o yaml

```
spec:  
  hosts:  
    - recommendation  
  http:  
    - route:  
      - destination:  
          host: recommendation  
          subset: version-v2  
          weight: 100
```

- kubectl get DestinationRule -n tutorial -o yaml

```
spec:  
  host: recommendation  
  subsets:  
    - labels:  
        version: v1  
        name: version-v1  
    - labels:  
        version: v2  
        name: version-v2
```

- 서비스 확인

- 브라우저에서 Customer 서비스(External-IP:8080 접속)호출
  - Kiali(External-IP:20001)에서 모니터링

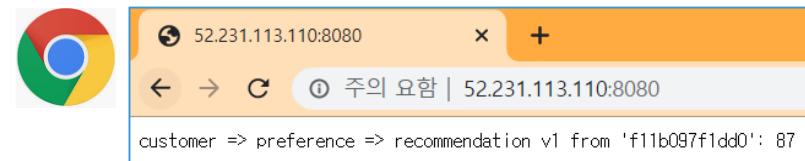
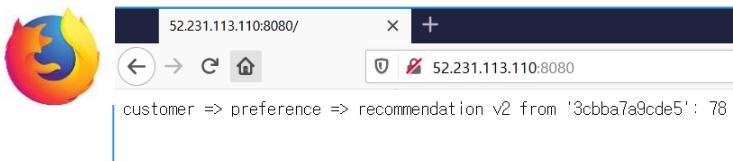


# Lab. Istio – Improved Routing (2/3)

- 가중치 기반 Service Routing 실습
  - recommendation 서비스 v1의 가중치를 100으로 변경
  - kubectl replace -f istiofiles/virtual-service-recommendation-v1.yml -n tutorial
    - 서비스 호출 및 Kiali(Externl-IP:20001)에서 모니터링
- VirtualService 삭제 시, Round-Robin 방식으로 동작
  - kubectl delete -f istiofiles/virtual-service-recommendation-v1.yml -n tutorial
- 이와 같이 가중치를 사용해 카나리 배포(Canary Deploy) 가 가능함
  - 90 : 10
  - kubectl create -f istiofiles/virtual-service-recommendation-v1\_and\_v2.yml -n tutorial
  - 75 : 25
  - kubectl replace -f istiofiles/virtual-service-recommendation-v1\_and\_v2\_75\_25.yml -n tutorial
- 삭제
  - kubectl delete dr recommendation -n tutorial
  - # kubectl delete vs recommendation -n tutorial
  - kubectl scale --replicas=1 deployment/recommendation-v2 -n tutorial

# Lab. Istio – Improved Routing (3/3)

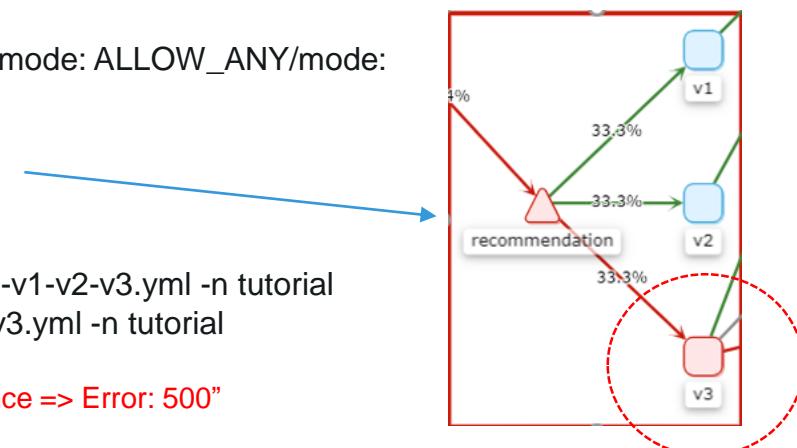
- Client 브라우저 유형별 Service Routing 실습
  - Firefox 브라우저로 접속 시, v2로 라우팅되도록 설정
    - kubectl create -f istiofiles/destination-rule-recommendation-v1-v2.yml -n tutorial
    - kubectl create -f istiofiles/virtual-service-firefox-recommendation-v2.yml -n tutorial
  - Firefox 브라우저와 다른 브라우저에서 접속 확인
  - Browser 환경이 지원되지 않을 경우,
    - curl -A Safari External-IP:8080
    - curl -A Firefox External-IP:8080



- 삭제
  - kubectl delete dr recommendation -n tutorial
  - kubectl delete vs recommendation -n tutorial

# Lab. Istio – Egress

- 외부 도메인을 호출하는 v3 버전을 배포
  - kubectl apply -f <(istioctl kube-inject -f recommendation/kubernetes/Deployment-v3.yaml) -n tutorial
  - 브라우저에서 Customer 서비스(External-IP:8080 접속)
    - v3에서 날짜정보가 추가로 출력됨을 확인
- Istio 트래픽을 등록된 것만 허용하도록 변경
  - kubectl get configmap istio -n istio-system -o yaml | sed 's/mode: ALLOW\_ANY/mode: REGISTRY\_ONLY/g' | kubectl replace -n istio-system -f -
  - 브라우저에서 Customer 서비스(External-IP:8080 접속)
    - v3은 서비스 오류로 인해 브라우저 확인 불가, Kiali에서 확인
- 트래픽을 모두 v3 (weigh 100)로 라우팅하고 에러 화면 확인
  - kubectl create -f istiofiles/destination-rule-recommendation-v1-v2-v3.yaml -n tutorial
  - kubectl create -f istiofiles/virtual-service-recommendation-v3.yaml -n tutorial
  - 브라우저에서 Customer 서비스(External-IP:8080 접속)
    - 화면에 Error Log 출력 : “customer => Error: 503 - preference => Error: 500”
- 외부 도메인을 허용해 주는 ServiceEntry를 생성하여 정상 접속 허용
  - kubectl create -f istiofiles/service-entry-egress-worldclockapi.yaml -n tutorial
  - 브라우저에서 Customer 서비스(External-IP:8080 접속) - 정상 출력



# Lab. Istio – Clear Test

- 테스트 오브젝트 삭제
  - `kubectl delete namespace tutorial`
- Istio Configmap 원상 복구
  - `kubectl get configmap istio -n istio-system -o yaml | sed 's/mode: REGISTRY_ONLY/mode: ALLOW_ANY/g' | kubectl replace -n istio-system -f -`

# Lab. Istio



Lab Time

- Lab Script Location
  - Workflowy :

# Table of content

Container Orchestration  
(Docker & k8s)

1. MSA, Container, and Container Orchestration
2. Setup Azure Platform
3. Docker / Kubernetes, Kubernetes Architecture
4. Kubernetes Basic Object Model : Pod, Label, ReplicaSet, Deployment, Service, Volume, Configmap, Secret, Liveness/Readiness
5. Kubernetes Advanced Lab : Ingress, Job, Cron Job, DaemonSet, StatefulSet
6. Service Mesh: Istio for Advanced Services Control
7. Course Test 

# Course Test

- 1. 5GB의 물리 AzureDisk('default' Storage Class)를 사용해, nginx의 Root 디렉토리(index.html)로 설정하는 Pod를 생성하세요.
- 2. Literal Type의 key=value(User = Container-Expert) ConfigMap을 생성하고, value를 웹 브라우저에 출력하는 Pod를 생성하세요.
  - 단, 어플리케이션은 NodeJs(교재 참조)를 사용하고, 나의 ACR(Azure Container Registry) 이미지로부터 컨테이너 생성

# THANKS!

## Any Question?

---

You can find me at:

[jyjang@uengine.org](mailto:jyjang@uengine.org)

<https://github.com/jinyoung>

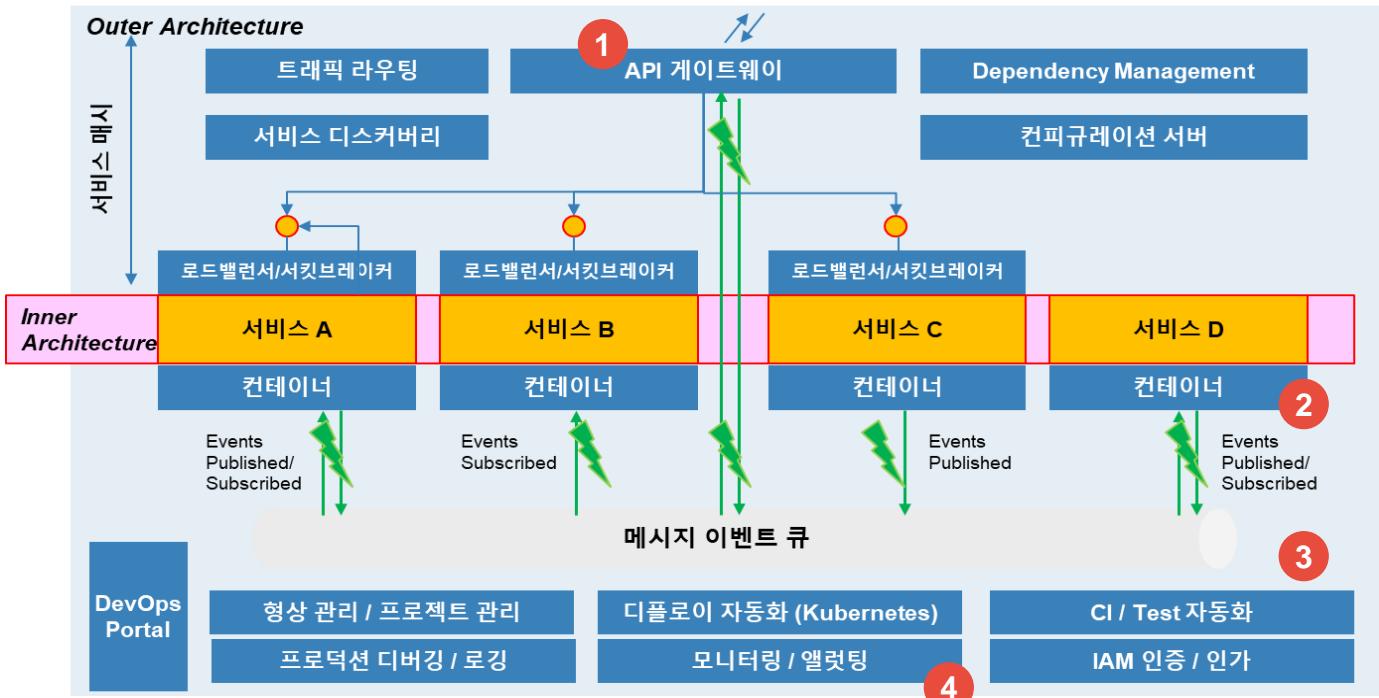
<https://github.com/TheOpenCloudEngine>

“

# Appendix

- Outer Architecture
- AKS(Azure Kubernetes Service) Architecture
- Linux tips used in the book

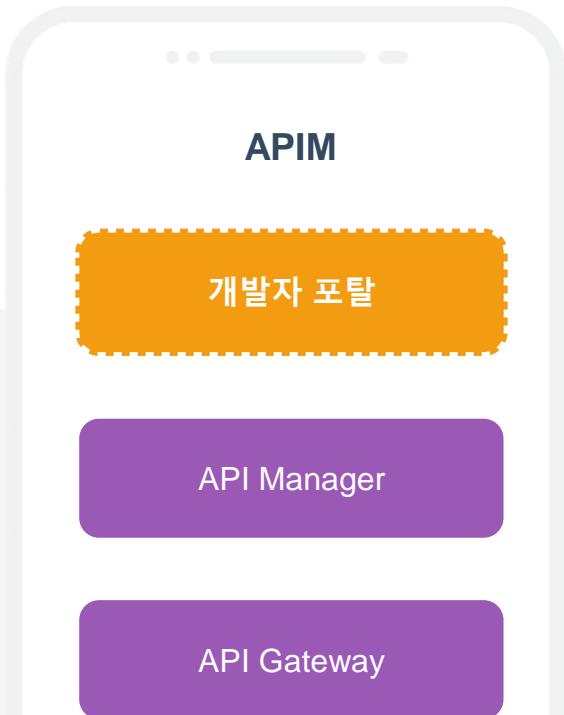
# What is Outer-Architecture



1. API Gateway : API를 사용하여 통신하는 모든 서비스들의 인증 및 제어, 트래픽 모니터링 등 수행
2. Managed Container System : 서비스들의 종양 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 트랜잭션 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.

## Service Mesh

Istio

envoy

## Orchestration

kubernetes

## Coordination & Service Discovery

coreDNS

etcd

## Container Runtime

containerd

## SDN

Google Cloud Virtual Network

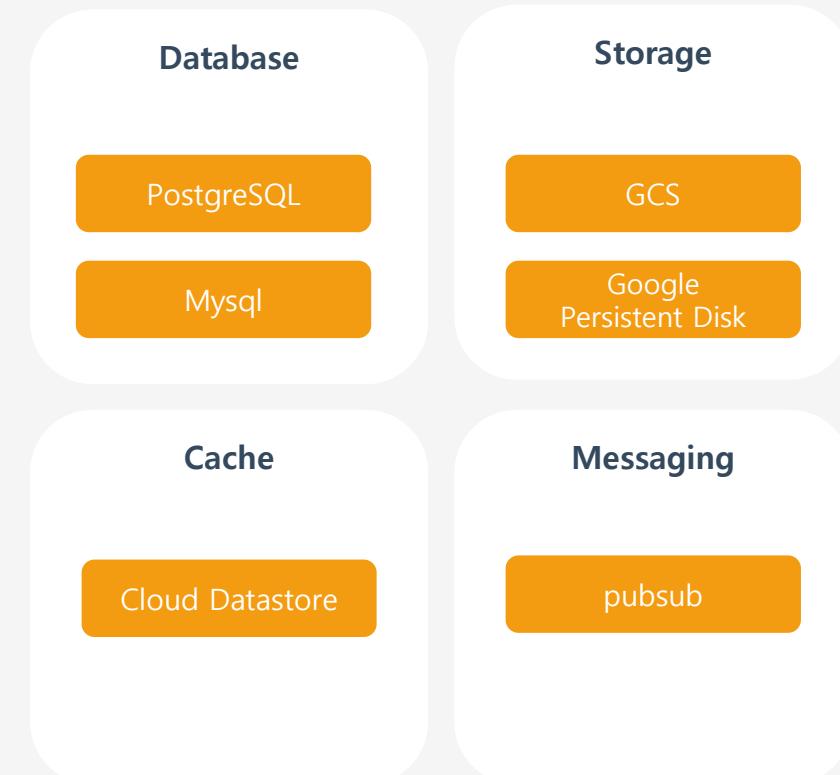
## CI/CD

Cloud Build

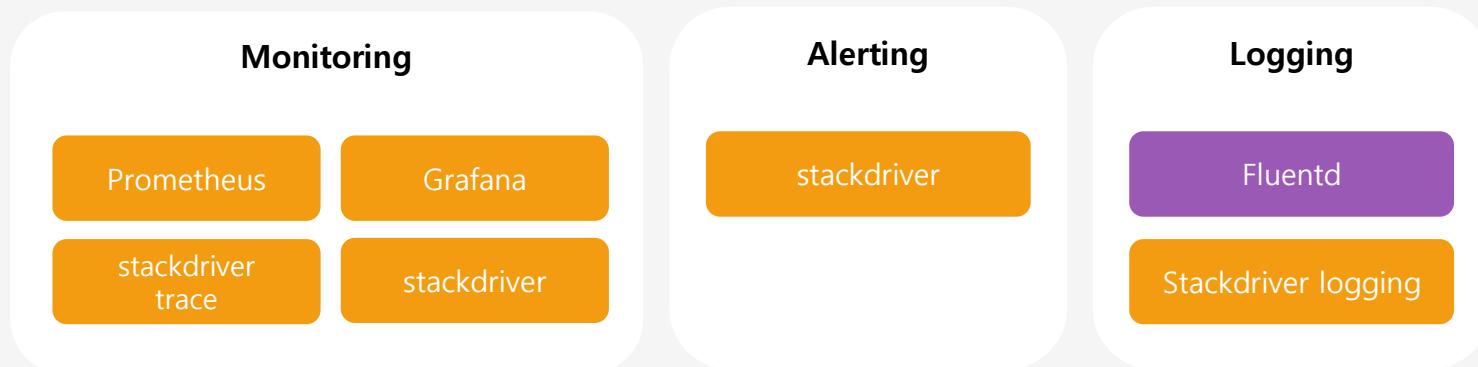
Cloud Source Repositories

# Backing Services

- **DBMS** : 데이터를 일정한 룰에 의해 관리하는 기술. 일반적으로 관계형 데이터베이스(RDB)를 이용. 대표적으로 Oracle, Mysql 등이 있다.
- **Storage** : File Storage, 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.

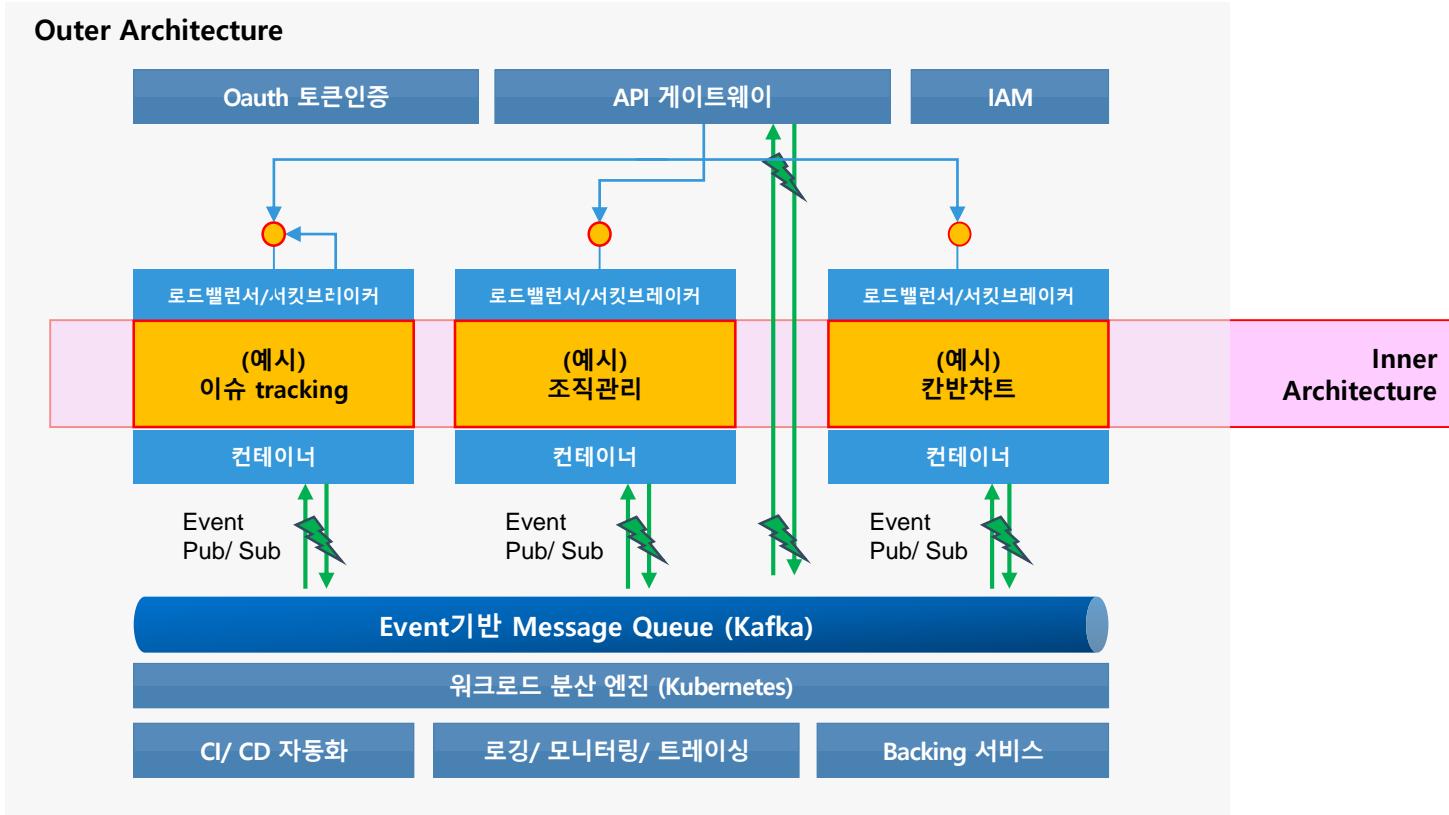


# Observability

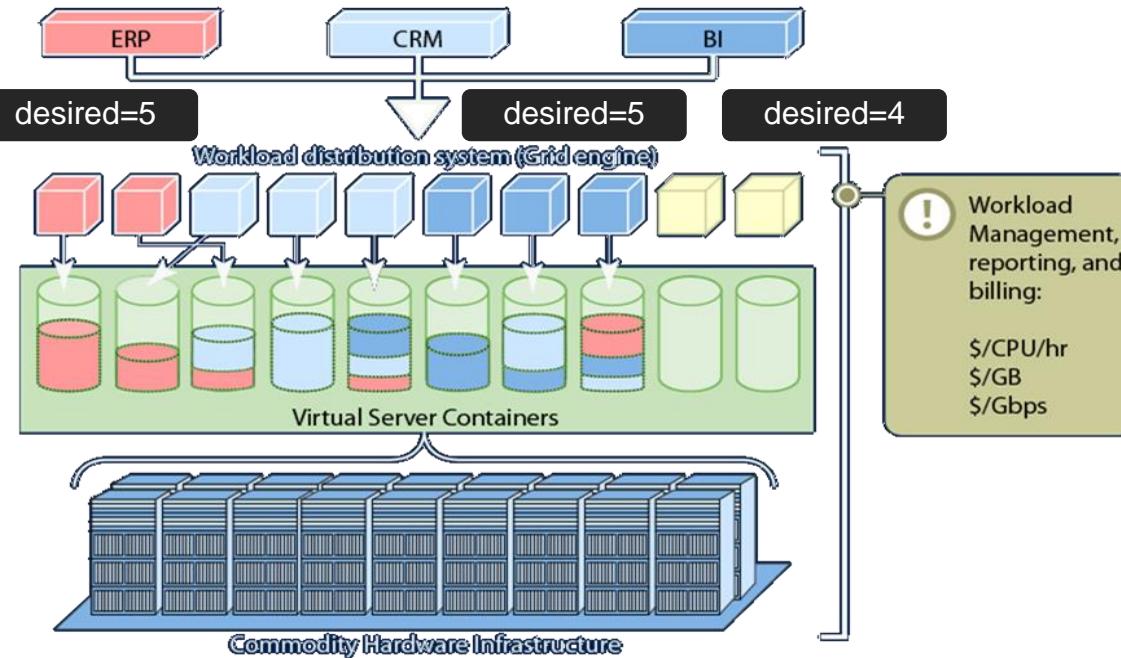


- **Monitoring** : 시스템 자원의 사용량이나 에러등에 대해 모니터링 하는 기술.
- **Alerting** : 로깅 및 모니터링의 정보를 활용하여 특정 규칙을 걸어 위반할 경우 알림을 보내는 기술.
- **Logging** : 시스템에서 발생하는 로그를 수집 및 분석을 하는 기술.

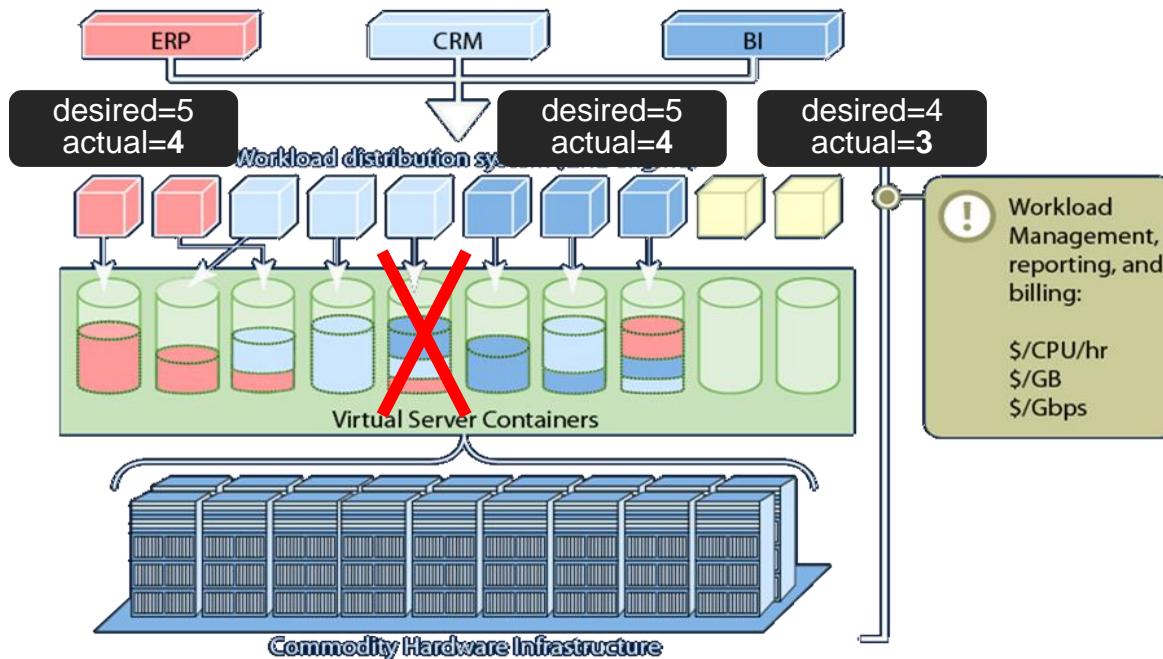
# Applied Configuration



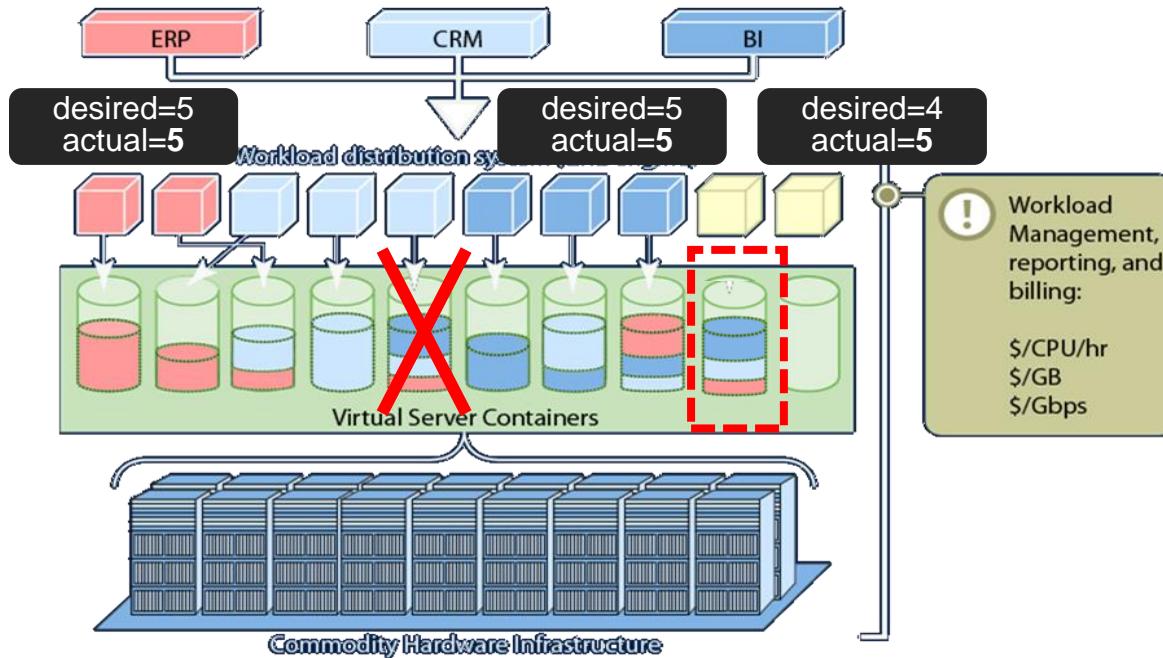
# Workload Distribution Engine



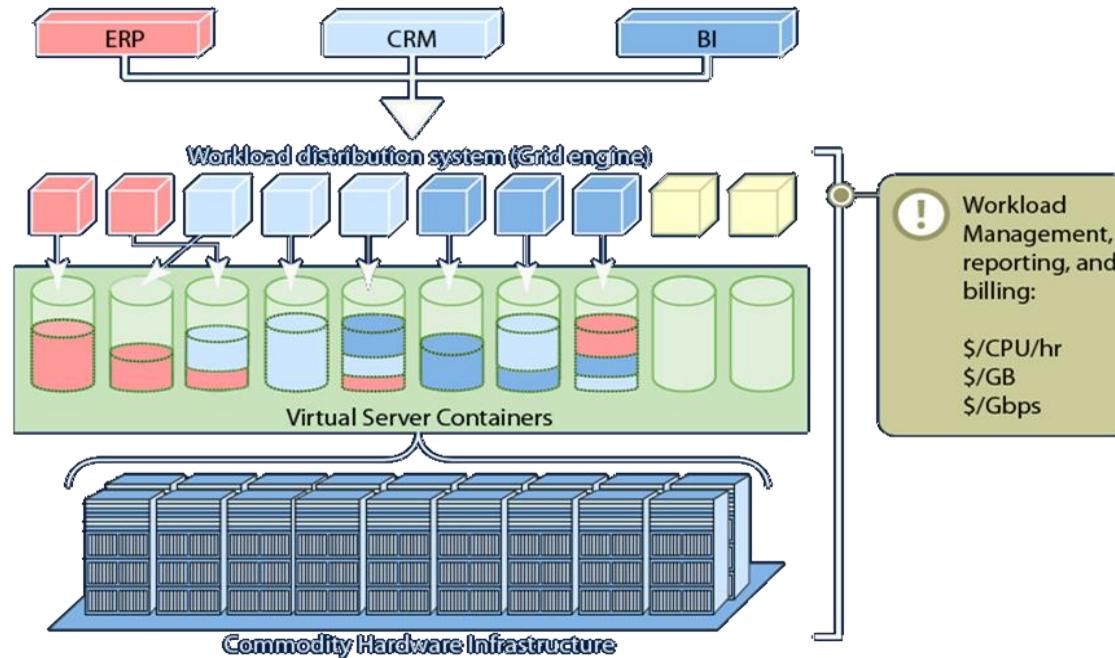
# Workload Distribution Engine



# Workload Distribution Engine

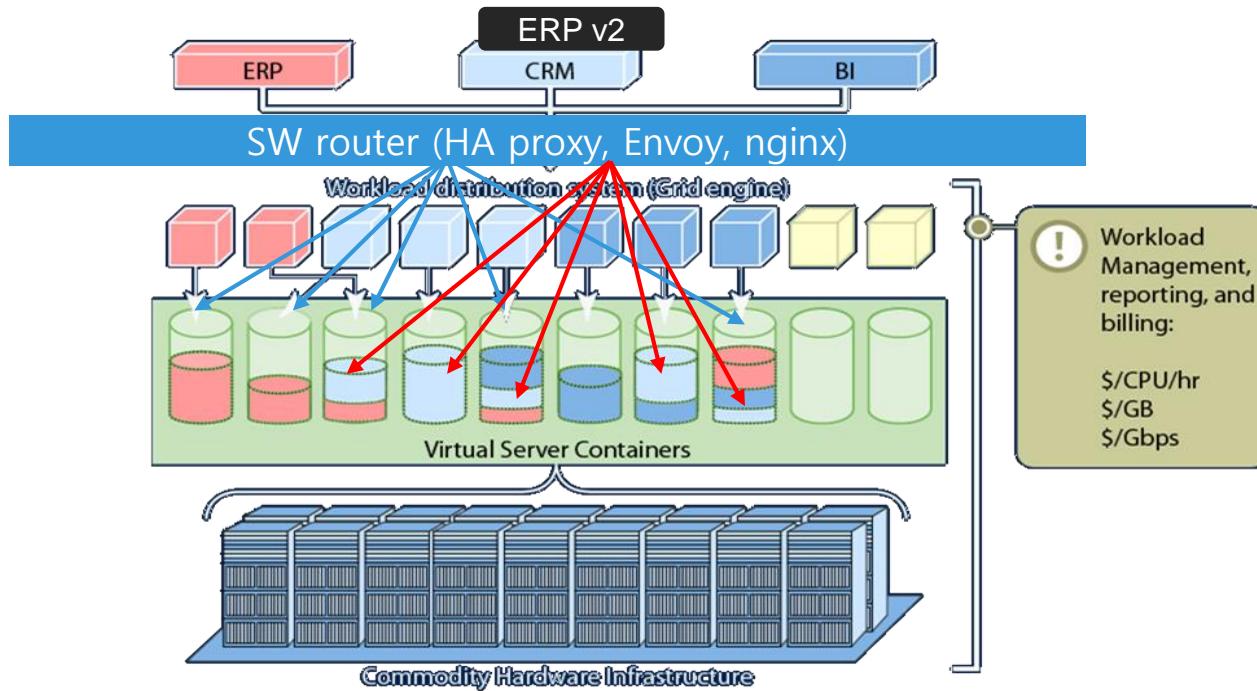


# Workload Distribution Engine



## Managing Scalability

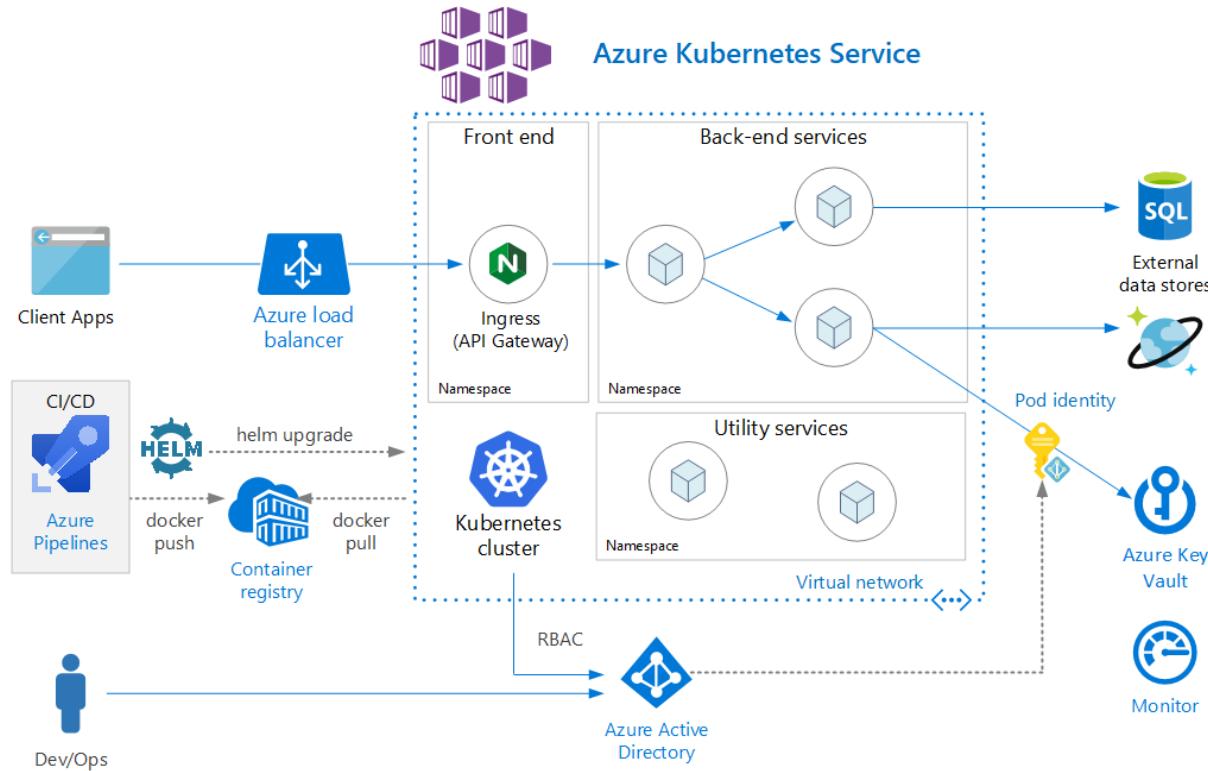
# Workload Distribution Engine



# Platforms around the outer Architecture

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

# AKS Architecture



# 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.



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

# Linux tips used in the book

- OS 버전 확인 : lsb\_release -a
- Network status tool install
  - ubuntu :
    - apt-get update, apt-get install net-tools, apt-get install curl
    - apt-get install procps (ps 설치)
  - alpine : apk update, apk add net-tools, apk add curl
- Base64 encoding : echo java | base64
- Base64 decoding : echo 'MWYyZDFIMmU2N2Rm' | base64 --decode
- Windows hosts 파일을 수정 못하는 경우 : curl을 사용한 test
  - \$ curl -H "host:blue.example.com" http://xxx.xx.xx.xx

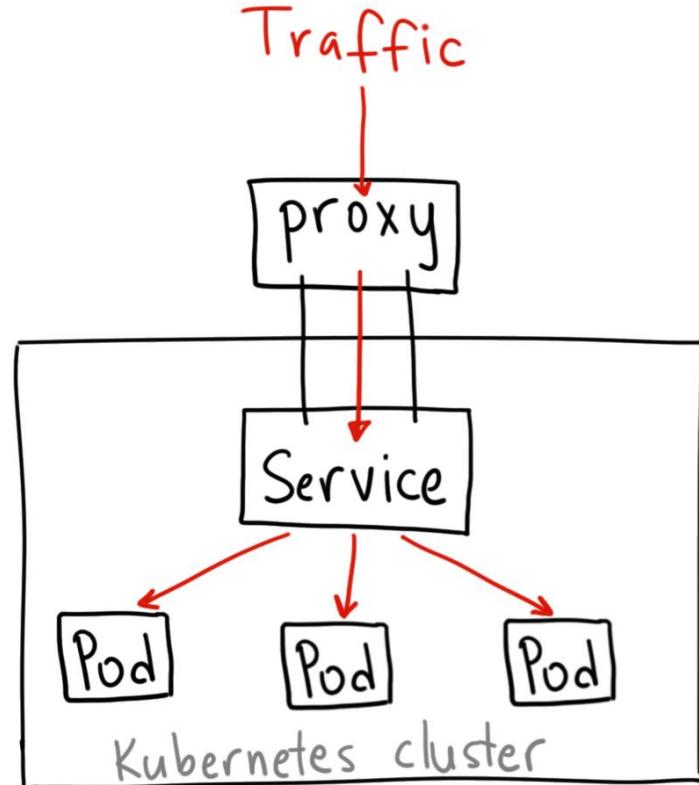
“

Kubernetes NodePort vs  
LoadBalancer vs Ingress..

언제 무엇을 써야 할까??

# ClusterIP

- Default, 클러스터 내부에서만 사용,
- ClusterIP 서비스는 Kubernetes 기본 서비스로, 클러스터 내의 서비스간의 통신을 가능하게 해준다. ClusterIP는 외부 접근이 되지 않는다.
- 외부에서 접근하려면,
  - `kubectl proxy --port 8080`
  - `http://localhost:8080/api/v1/proxy/namespaces/default/services/my-internal-service:http/`
- 언제 사용해야 할까?
  - 쿠버네티스를 설치한 환경에서 서비스를 디버깅하거나 어떤 이유로 노트북/PC에서 직접 접근할 때
  - 내부 대시보드 표시 등 내부 트래픽을 허용할 때

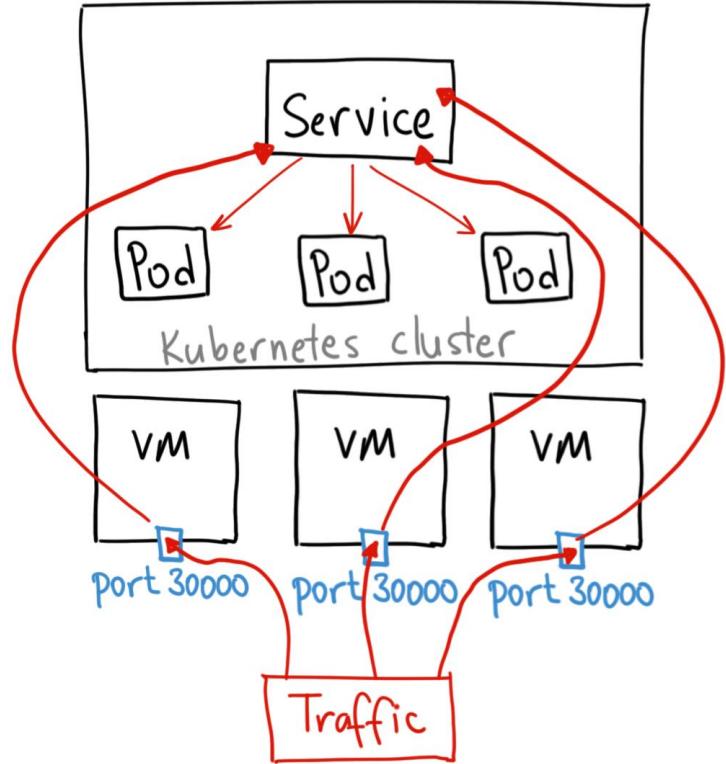


# NodePort

- NodePort 서비스는 서비스에 외부 트래픽을 직접 보낼 수 있는 가장 원시적인(primitive) 방법
- ClusterIP를 지원하고, NodePort는 모든 Node(VM)에 특정 포트를 열어 이 포트로 보내지는 모든 트래픽을 서비스로 포워딩
- 타입이 NodePort

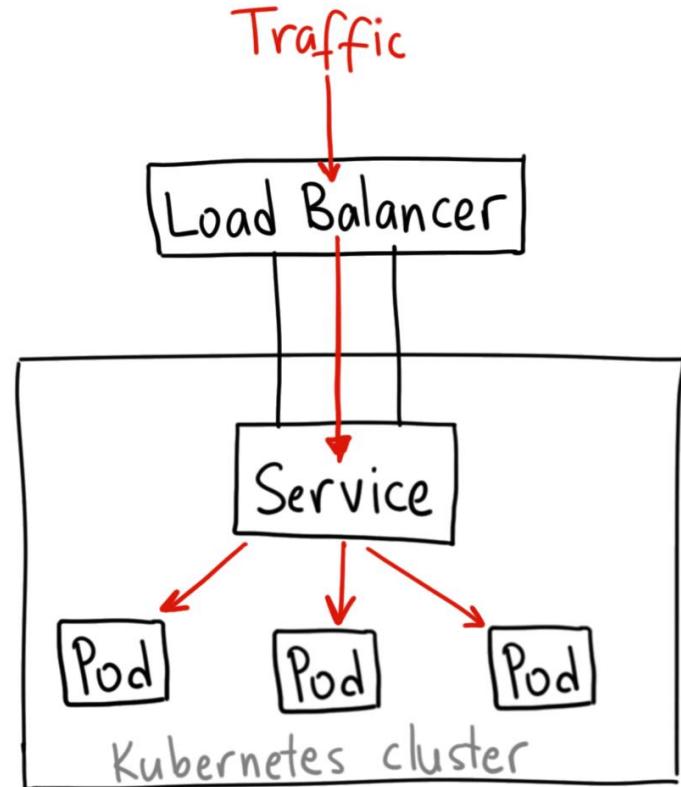
```
apiVersion: v1
kind: Service
metadata:
  name: my-nodeport-service
spec:
  selector:
    app: my-app
  type: NodePort
  ports:
    - name: http
      port: 80
      targetPort: 80
      nodePort: 30036
      protocol: TCP
```

- 언제 사용해야 할까?
  - 항상 사용 가능한 상태가 아니어도 되는, 몇몇을 위한 내부용 서비스를 운영하거나
  - 비용에 민감(cost sensitive)하다면, 이 방법이 적합



# LoadBalancer

- LoadBalancer 서비스는 서비스를 인터넷에 노출하는 일반적인 방식
  - 탑입이 LoadBalancer
- 
- 언제 사용해야 할까?
    - 서비스를 직접적으로 노출하기를 원한다면, LoadBalancer 가 기본적인 방법
    - 지정한 포트의 모든 트래픽은 서비스로 포워딩
    - 필터링이나 라우팅 기능은 없음



# Ingress

- Ingress는 쿠베네티스 오브젝트 개념의 ‘서비스’가 아님
- 여러 서비스들 앞에서 “스마트 라우터(smart router)” 역할을 하거나 클러스터의 진입점(entrypoint) 역할 point 역할
- 백엔드 서비스로 경로(path)와 서브 도메인 기반의 라우팅을 모두 지원
- 이러한 기능을 하는 모듈을 API Gateway라고 함

