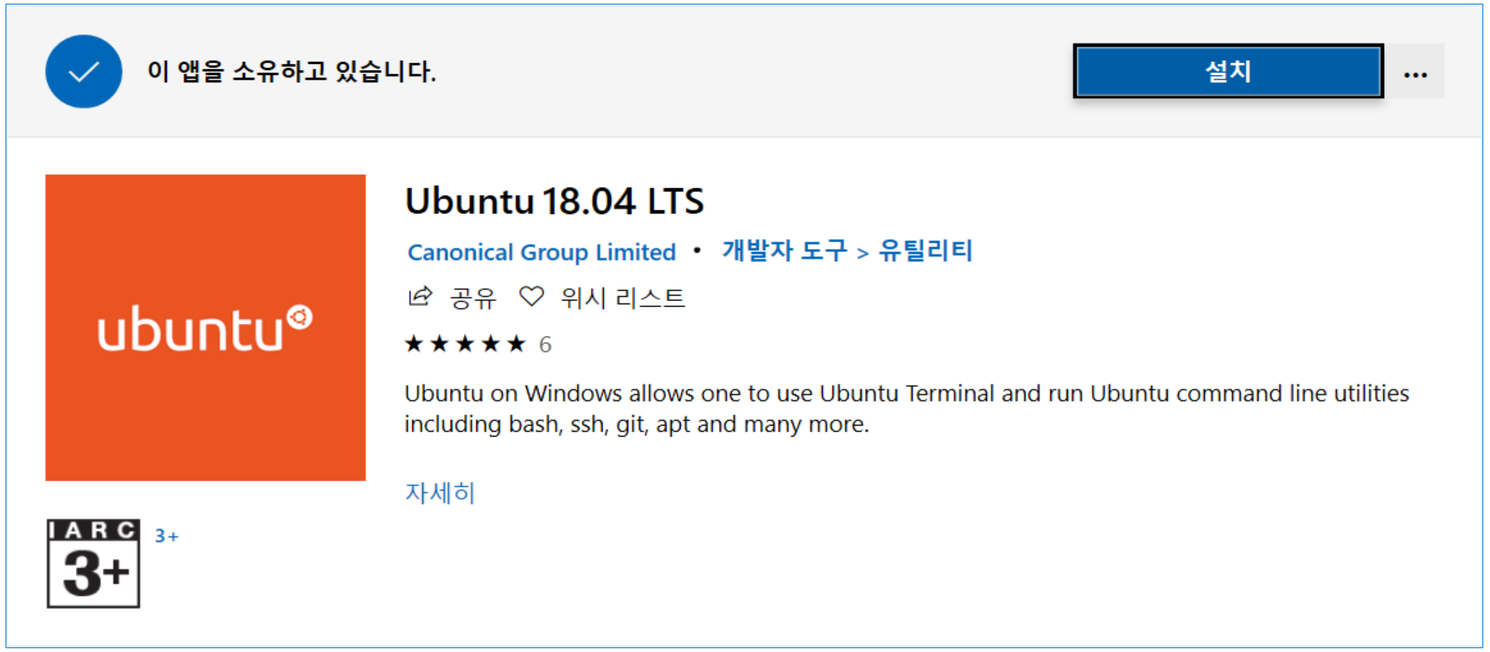
Container Orchestration

(Docker & Kubernetes)

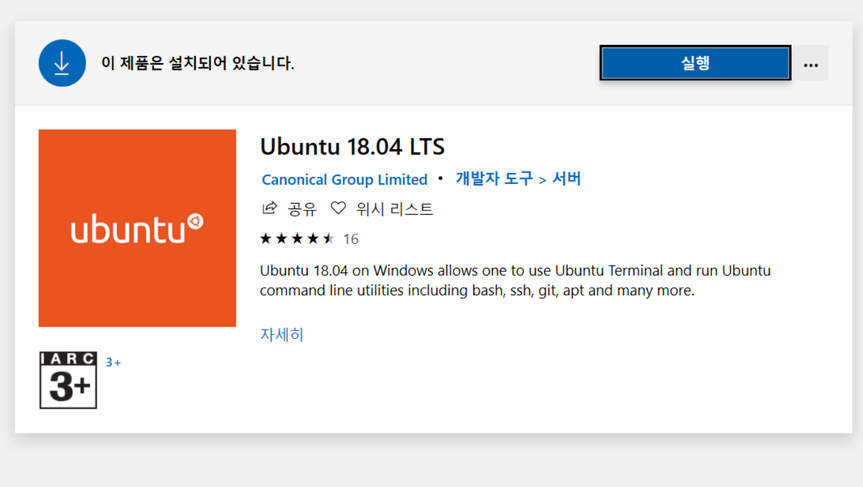
Windows 10에 Linux 설 치

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

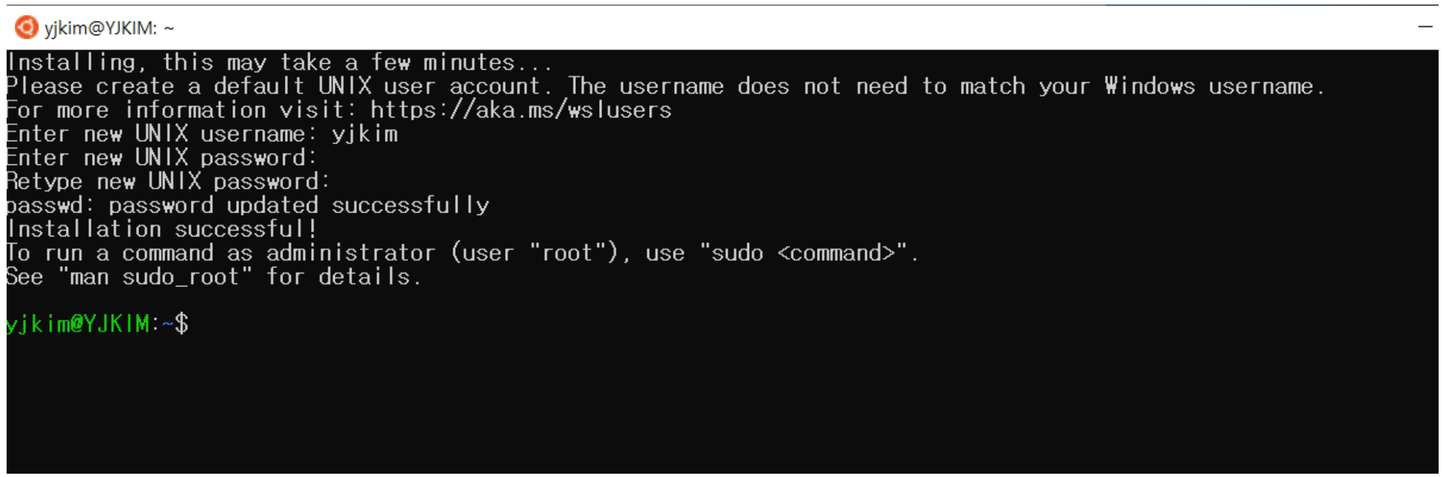


https://docs.microsoft.com/ko-kr/windows/wsl/install-win10

Windows 10에 Linux 설 치



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

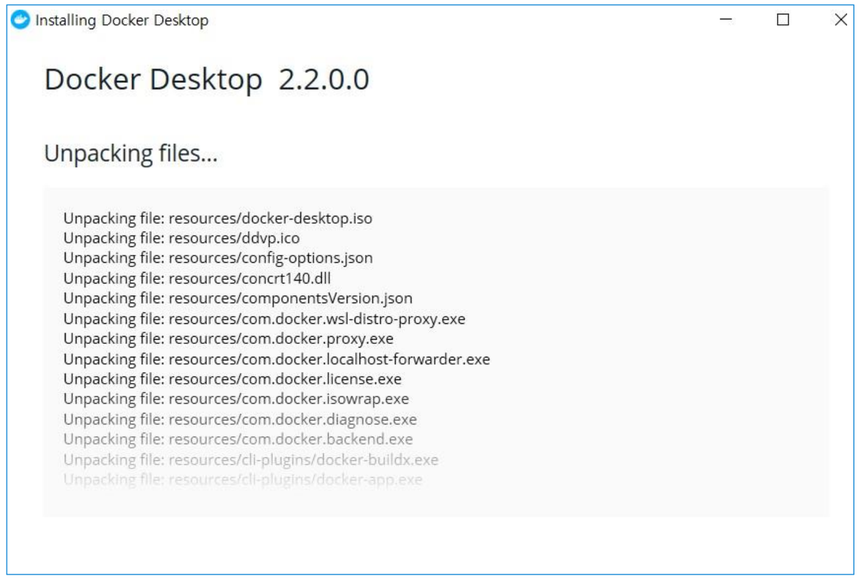


- Ubuntu의 Archive Repository Server를 국내로 설정

```
sudo vi /etc/apt/sources.list
:%s/archive.ubuntu.com/ftp.daumkakao.com/g
:wq!
```

Docker Setup - Daemon

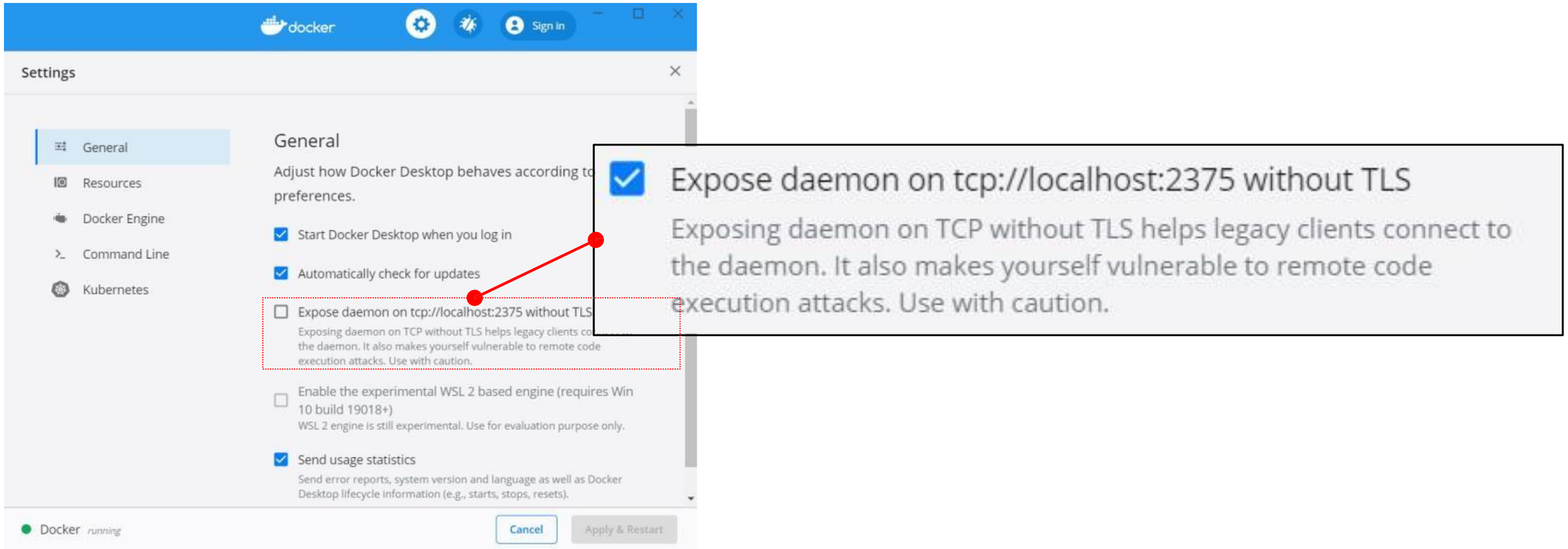
- Windows 10에 Docker 데몬(Docker for Windows) 설치
  - https://www.docker.com/products/docker-desktop



  - 컴퓨터 재시작

Docker Setup – Daemon config

- Docker Hub(https://hub.docker.com/)에 접속하여 계정 생성
- Docker Desktop 실행 후 – skip
- Settings 메뉴 ‘Expose daemon on~~’ 체크 후 재시작



Docker Setup – Client

- Linux에 Docker Client 설치
  - apt-get update

    # Install packages to allow apt to use a repository over HTTPS.
  - 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.
  - add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable“
  - apt update

    # Install the latest version of Docker CE.
  - **apt install docker-ce**

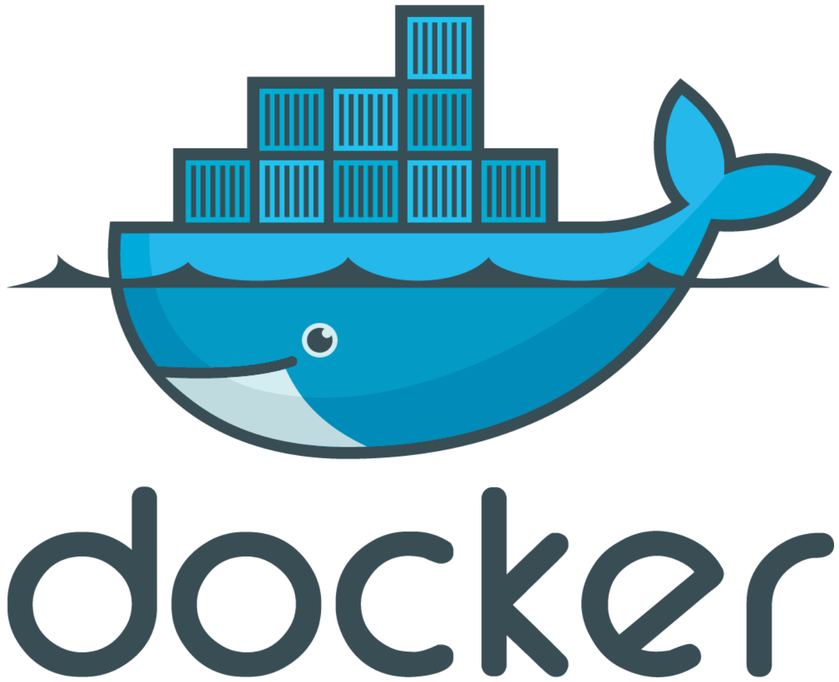
    # Allow your user to access the Docker CLI without needing root
  - usermod -aG docker [*Linux username*]

Docker Setup – connect Client to Daemon

- Linux 계정에 Docker Daemon 환경변수 설정
  - vi ~/.bashrc
  - **# 맨 아래에 환경변수 추가**
  - export DOCKER_HOST=tcp://0.0.0.0:2375
  - source ~/.bashrc # Shell 재 실행

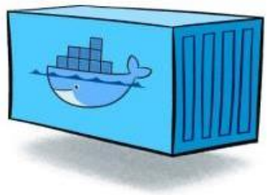
- Docker Setup 확인
  - $ docker run hello-world

Docker Container

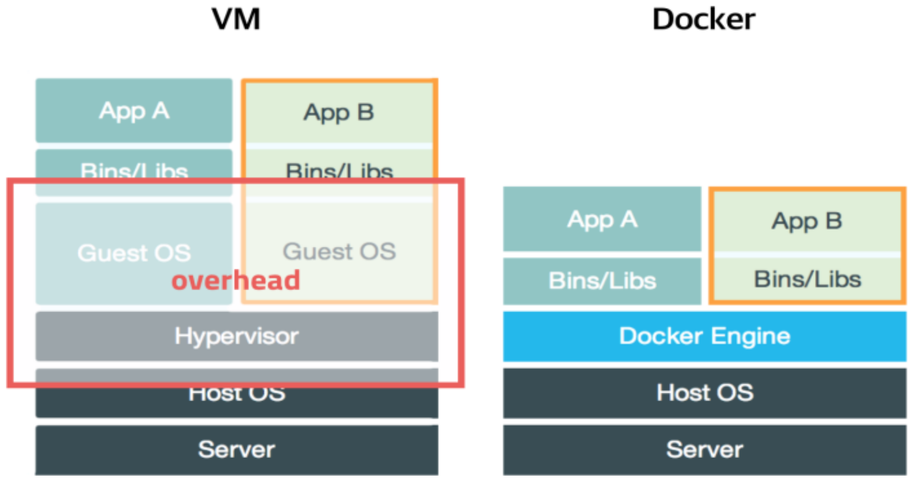


- 2013년 3월,
  dotCloud 창업자 Solomon Hykes가 Pycon Conference 에서 발표
- Go 언어로 작성된 “The future of linux Containers”

- Container based
  - 프로세스 격리 기술
  - 오픈소스 가상화 플랫폼

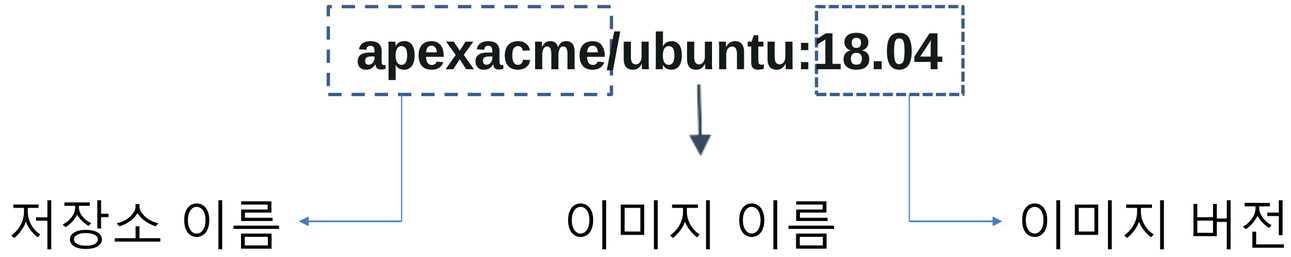


- 가상머신 .vs. Docker



Docker Image & Container

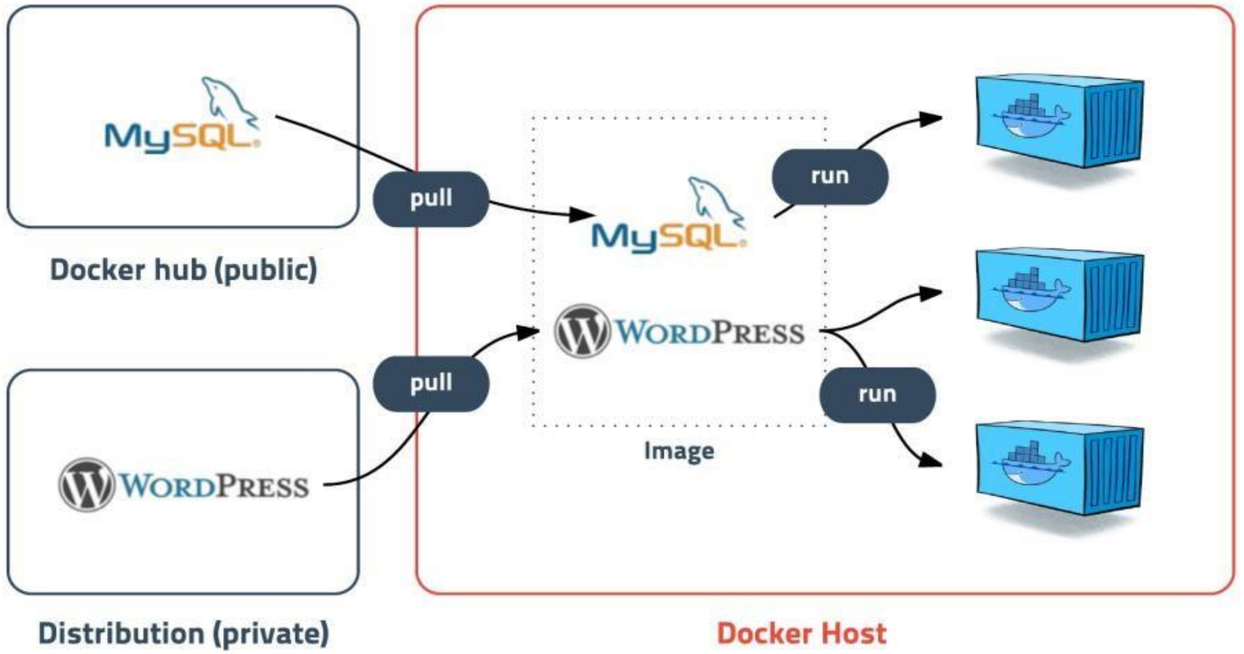
- Docker Image
  - 가상머신 생성시 사용하는 ISO 와 유사한 개념의 이미지
  - 여러 개의 층으로 된 바이너리 파일로 존재
  - 컨테이너 생성시 읽기 전용으로 사용
  - 도커 명령어로 레지스트리로부터 다운로드 가능



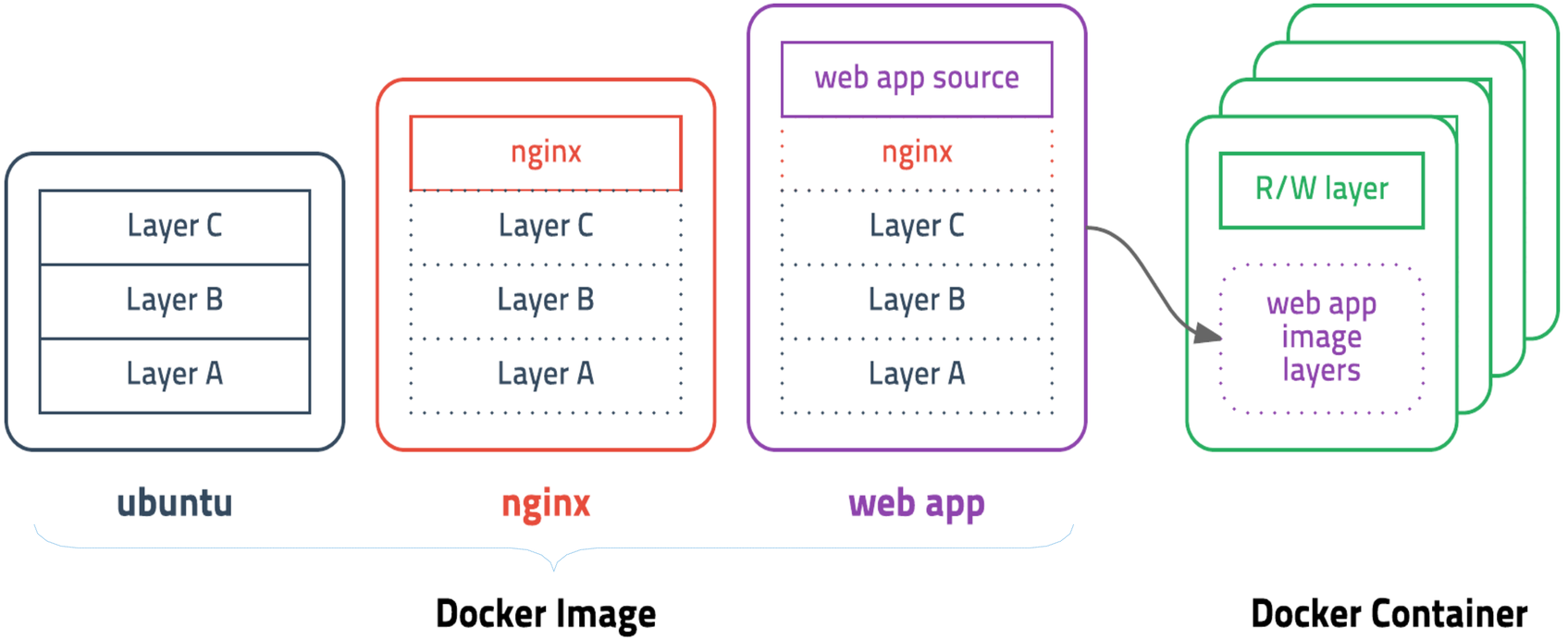
  - 저장소 이름 : 이미지가 저장된 장소, 이름이 없으면 도커 허브(Docker Hub)로 인식
  - 이미지 이름 : 이미지 이름, 생략 불가
  - 이미지 버전 : 이미지 버전정보, 생략 시 latest 로 인식

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

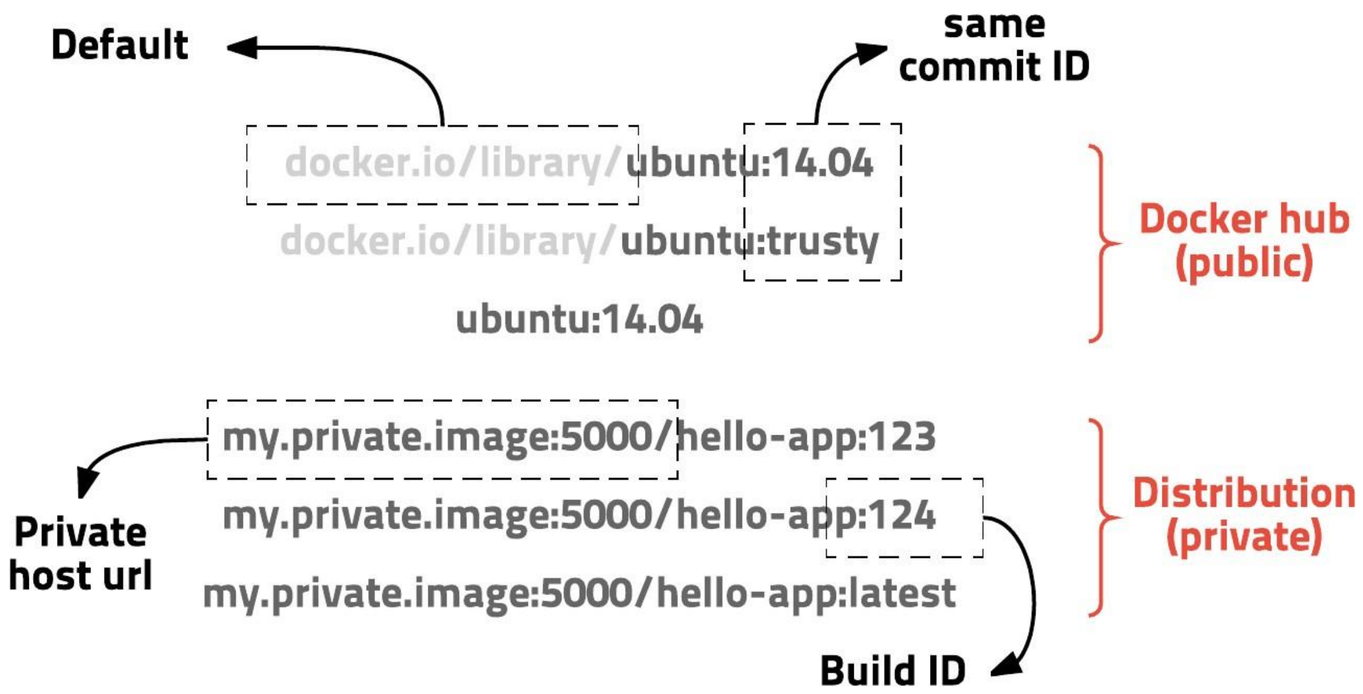


Layered Architecture



- 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

Docker File

- Docker File 은 컨테이너 이미지 빌드를 자동화하기 위해 제공되는 메커니즘으로 쉽고 간단하며 명확한 구문을 갖는 텍스트파일이다.
- Instruction : 높은 가독성을 위해 대문자 사용을 권장하며 구성 순서가 중요하다.
- Docker File 에서 이미지를 빌드하는 3단계 프로세스

- 1단계 : 작업 디렉토리를 만든다.

*불필요한 파일을 이미지에 통합하지 않으려면 빈 작업 디렉토리를 만들어야 하며
보안상의 이유로 / (루트디렉토리)는 이미지 빌드용 작업 디렉토리로 사용하면 안된다.
$mkdir Dockerdir

- 2단계: Docker file 사양을 작성한다.

$cd Dockerdir
~dockerdir$vim Dockerfile

- 3단계: docker 명령어를 사용하여 이미지를 빌드한다.

~dockerdir$docker build –t testimg:v1 .

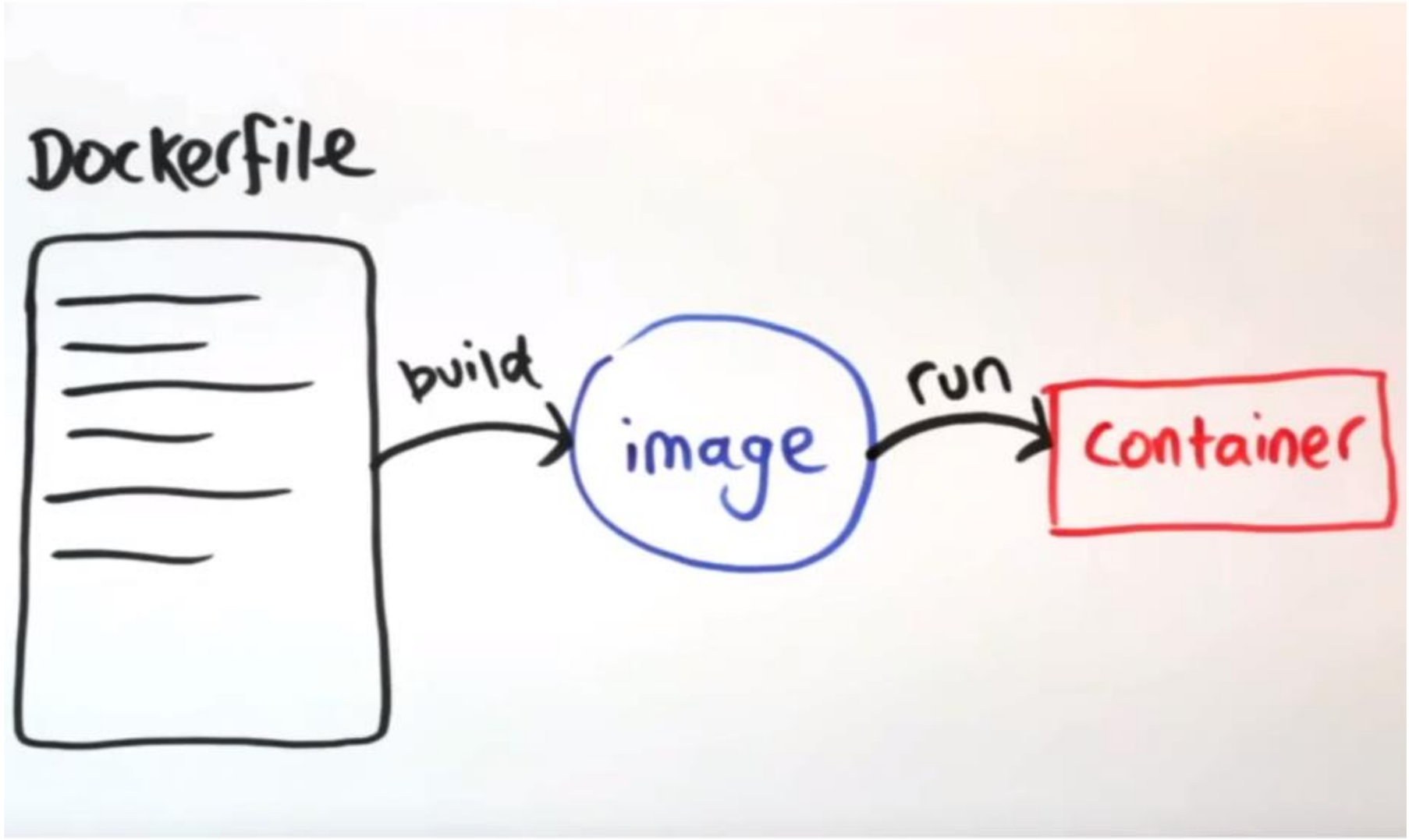
Dockerfile sample

```
# This is a comment line 1
FROM rhel7.3 2
LABEL description="This is a custom httpd container image" 3
MAINTAINER John Doe <jdoe@xyz.com> 4
RUN yum install -y httpd 5
EXPOSE 80 6
ENV LogLevel "info" 7
ADD http://someserver.com/filename.pdf /var/www/html 8
COPY ./src/ /var/www/html/ 9
USER apache 10
ENTRYPOINT ["/usr/sbin/httpd"] 11
CMD ["-D", "FOREGROUND"] 12
```

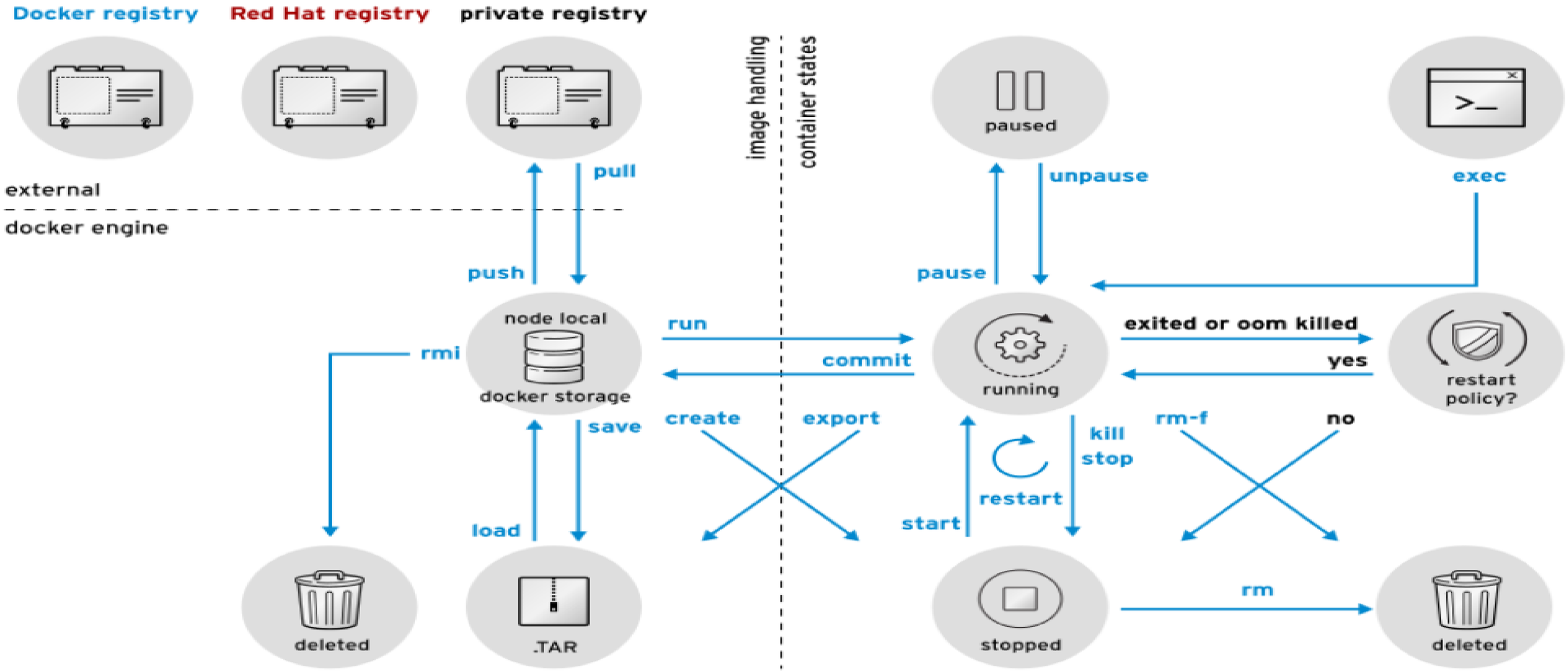
- FROM : 이미지를 생성할 때 사용할 베이스 이미지를 지정한다
- RUN : 이미지를 생성할 때 실행할 코드 지정한다. 예제에서는 패키지를 설치하고 파일 권한을 변경하기 위해 RUN 을 사용
- EXPOSE : 컨테이너가 런타임에 지정된 네트워크 포트에서 수신함을 나타내지만 EXPOSE 는 단순한 메타데이타이고 호스트에서 포트에 액세스할수는 없다. docker run 시 –p 옵션으로 호스트의 포트를 노출시킨다.
- WORKDIR : 작업 디렉토리를 지정한다. 해당 디렉토리가 없으면 새로 생성한다. 작업 디렉토리를 지정하면 그 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
- USER : docker file 의 run , cmd ,entrypoint 명령에 컨테이너 이미지를 실할할때 사용할 사용자이름 또는 UID 를 지정한다. 보안상의 이유로 root 가 아닌 다른 사용자를 정의하는 것이 좋다.
- CMD : 컨테이너가 생성될때 실행되는 명령어로 ENTRYPONT 값으로 override 할수 있다.
- ENTRYPOINT : 컨테이너를 구동할 때 실행할 명령어를 지정한다.

Docker File

Dockerfile(명세) 작성 -> docker build -> docker run

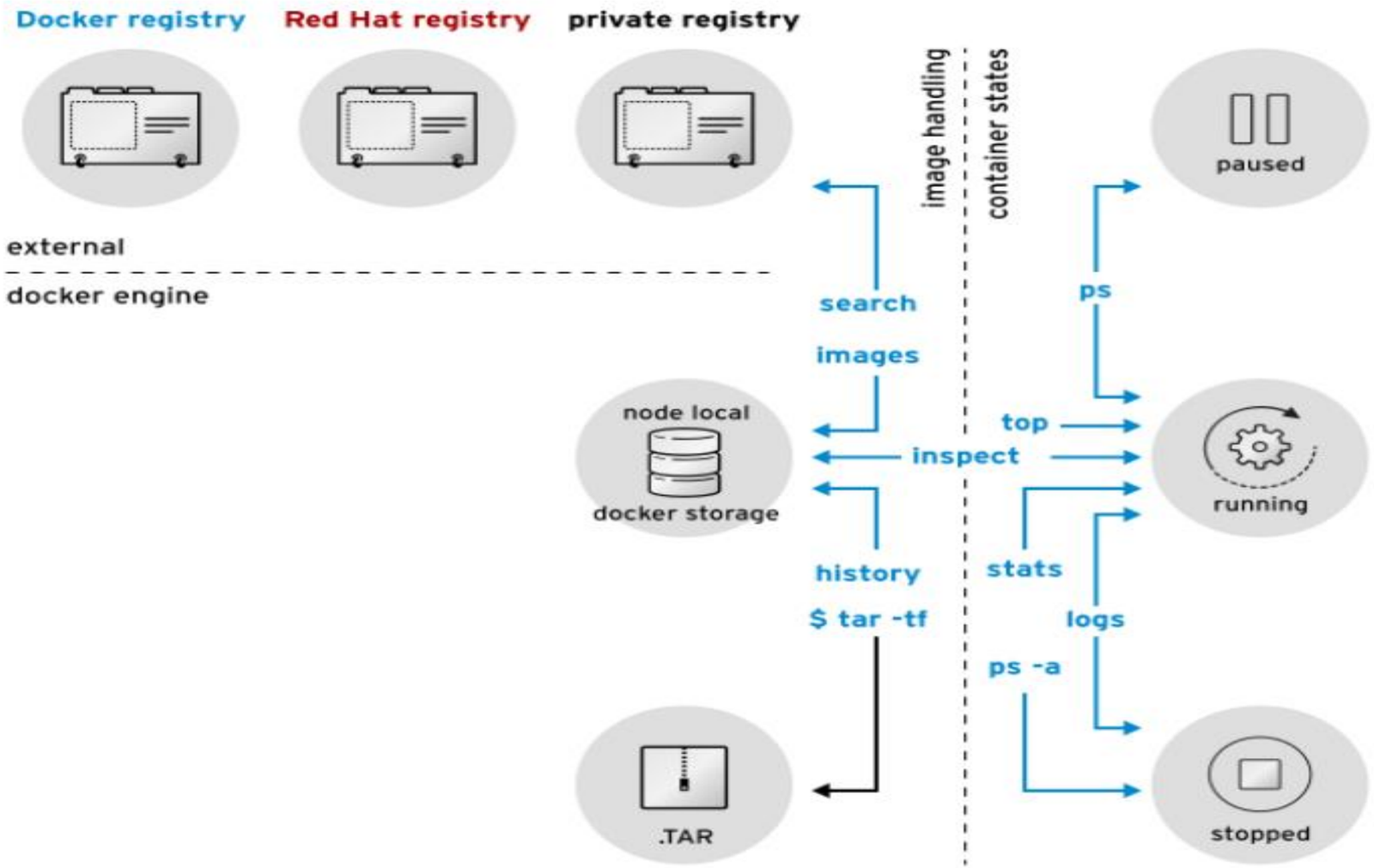


Docker 명령어



Docker client action verbs

Docker 명령어



Docker client query verbs

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)

Docker 이미지 관리 명령어

- 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 rmi[이미지 ID]
  - docker rmi -f [이미지 ID]

- 도커 모든 이미지 한 번에 삭제
  - $ docker rmi $(docker images -q)

- 도커 이미지 작업 히스토리 검색
  - $ docker history [이미지:태그]

Docker Container 명령어

- 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 rm [컨테이너 ID]
- 모든 컨테이너 한번에 삭제 (중지 후 삭제)
  - $ docker rm $(docker ps –qa)

Docker Container 생성과 목록

```
[student@docker ~]$ docker run --name hello-world ubuntu /bin/bash
[student@docker ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
      PORTS               NAMES
[student@docker ~]$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
             PORTS               NAMES
d6bf8e29bcc1        ubuntu              "/bin/bash"         7 seconds ago       Exited (0) 6 s
econds ago                              hello-world
```

Docker Container 시작과 제거

```
[student@docker ~]$ docker start hello-world
hello-world
[student@docker ~]$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
             PORTS               NAMES
a45c979184f2        ubuntu              "/bin/bash"         3 minutes ago       Exited (0) 6 s
econds ago                      hello-world
[student@docker ~]$ docker rm hello-world
hello-world
[student@docker ~]$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
      PORTS               NAMES
```

Docker Container 접속

docker run -it <이미지 > <실행 app>

```
student@node1:~$ docker run -it -d --name myapp nginx /bin/bash
6a0a4ef3e7dfa9a77bbfdc57d7bbaf4c2ccdf130c9803154b197e7107a32cc1d
student@node1:~$ docker exec -it myapp /bin/bash
root@6a0a4ef3e7df:/# ls -l
total 64
drwxr-xr-x   2 root root 4096 Apr 22 00:00 bin
drwxr-xr-x   2 root root 4096 Feb  1 17:09 boot
drwxr-xr-x   5 root root  360 Apr 26 13:21 dev
drwxr-xr-x   1 root root 4096 Apr 26 13:20 etc
drwxr-xr-x   2 root root 4096 Feb  1 17:09 home
drwxr-xr-x   1 root root 4096 Apr 23 13:02 lib
```

escape key : ^P->^Q 를 이용해야 컨테이너가 종료되지 않고 접속해제된다.

```
student@node1:~$ docker exec -it myapp /bin/bash
root@6a0a4ef3e7df:/# read escape sequence
student@node1:~$ docker attach myapp
root@6a0a4ef3e7df:/# exit
exit
```

Docker Container 접속

docker exec <컨테이너 id | name > <command> <argument>

exec: 컨테이너에 접속 없이 host에서 컨테이너에 명령을 실행한다.

```
[student@docker ~]$ docker start hello-world
hello-world
[student@docker ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
      PORTS               NAMES
9ddc05bb042d        ubuntu              "/bin/bash"         19 minutes ago      Up 6 seconds
                    hello-world
[student@docker ~]$ docker exec hello-world hostname
9ddc05bb042d
```

변경된 컨테이너 file 조회

docker diff <container_name>

- 이미지와 Container 를 비교하여 변경된 파일 목록 출력
  A : add
  C: change
  D: delete

```
student@node1:~$ docker run -it -d --name myweb nginx /bin/bash
42449f82fe395acf6e31622cb53d2e7969b3d29a239fcd79afbb6c6e881f241e
student@node1:~$ docker exec -it myweb /bin/bash
root@42449f82fe39:/# echo "hello" > /tmp/hello.txt
root@42449f82fe39:/# exit
exit
student@node1:~$ docker diff myweb
C /root
A /root/.bash_history
C /tmp
A /tmp/hello.txt
```

컨테이너 상세 정보 조회

docker inspect <container name>

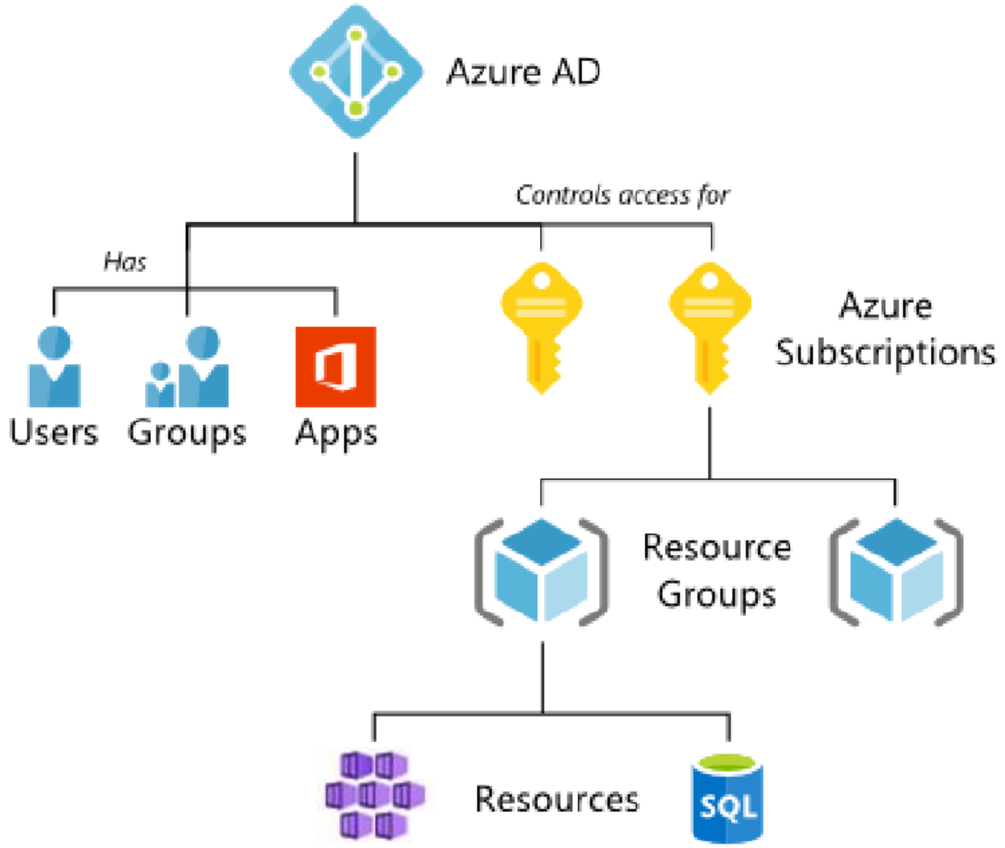
```
[root@docker docker_class]# docker inspect vol1
[
    {
        "Id": "cc785a8e139c594d4f5aefbadee91438da97e6a4a9a434082547c0bae8a38002",
        "Created": "2017-08-15T12:03:28.63069979Z",
        "Path": "/bin/sh",
        "Args": [],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 21844,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2017-08-15T12:38:40.835511245Z",
            "FinishedAt": "2017-08-15T12:36:57.129270728Z"
        },
        "Image": "sha256:3507b70a3102ea497c3fae32f4d60a57b0e8bea0e3a7296c8ab71bcc11840664",
        "ResolvConfPath": "/var/lib/docker/containers/cc785a8e139c594d4f5aefbadee91438da97e6a4a9a43408
2547c0bae8a38002/resolv.conf",
```

Agenda

1. 컨테이너 오케스트레이션
2. Kubernetes 클러스터 아키텍쳐
3. Kubernetes 기본 오브젝트 모델
    - Pod , Deployment
    - Label
    - Service
    - Volume
    - Configmap, Secret
    - Liveness / Readness
    - Ingress Controller
    - Job , CronJob
    - DaemonSet
    - StatefulSet
    - Resoure Management
    - Kubernetes Monitoring

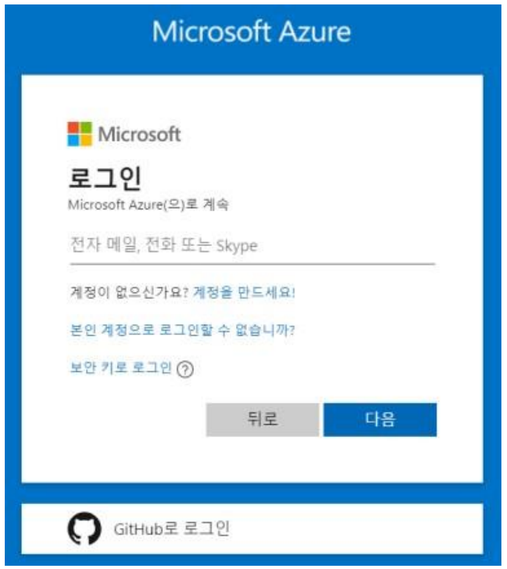
Azure Cloud Platform

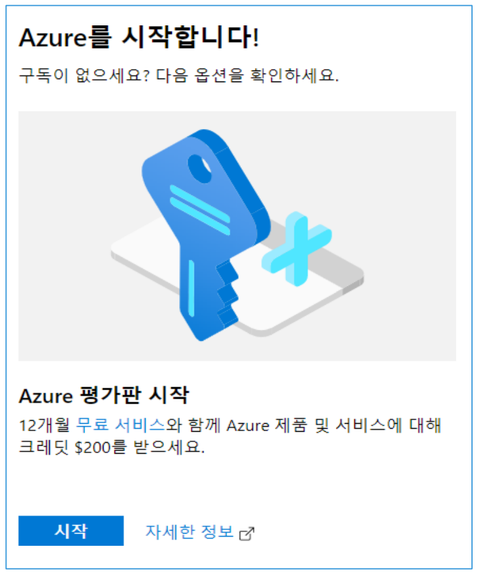
“가장 많은 글로벌 지역 보유, 56개 리전 140개 국가”



Azure Cloud Setup

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

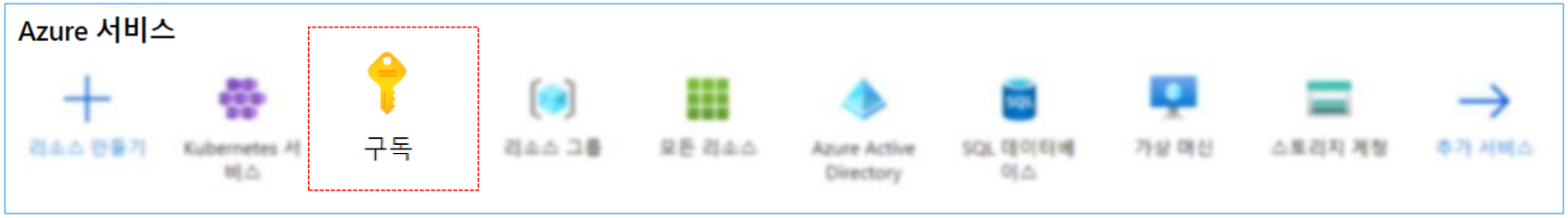


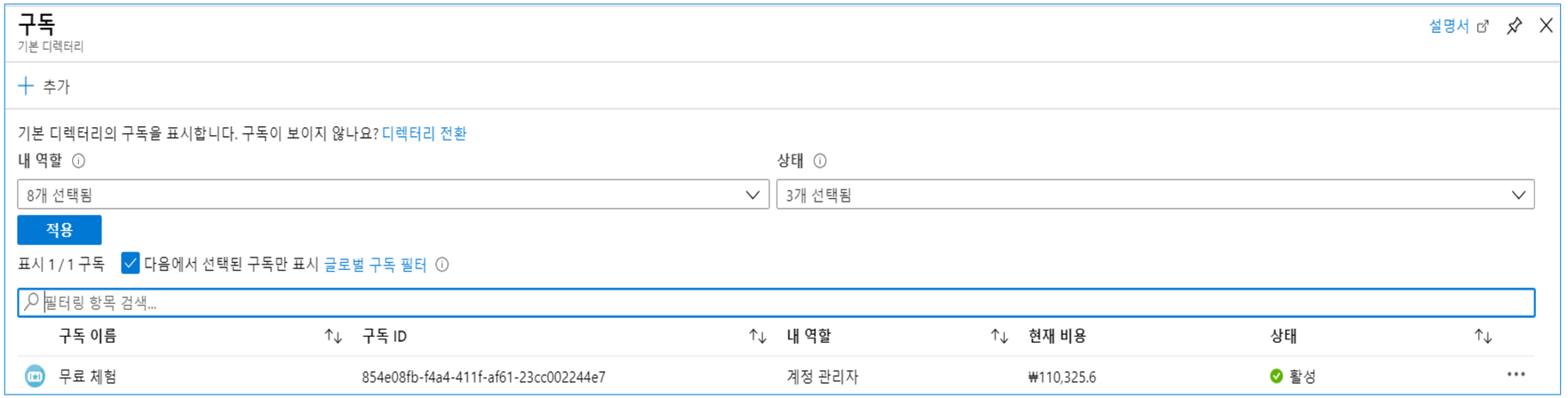


평가판 무료서비스
https://portal.azure.com/?quickstart=true#blade/Microsoft_Azure_Billing/FreeServicesBlade

Azure Cloud Setup

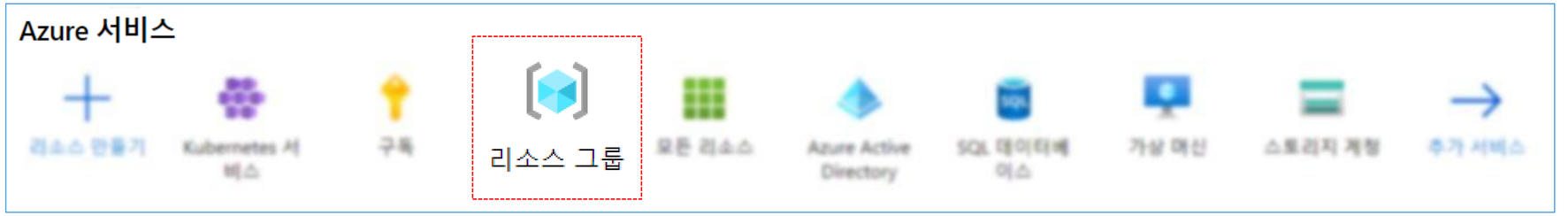
- 무료체험 구독(Subscription) 생성

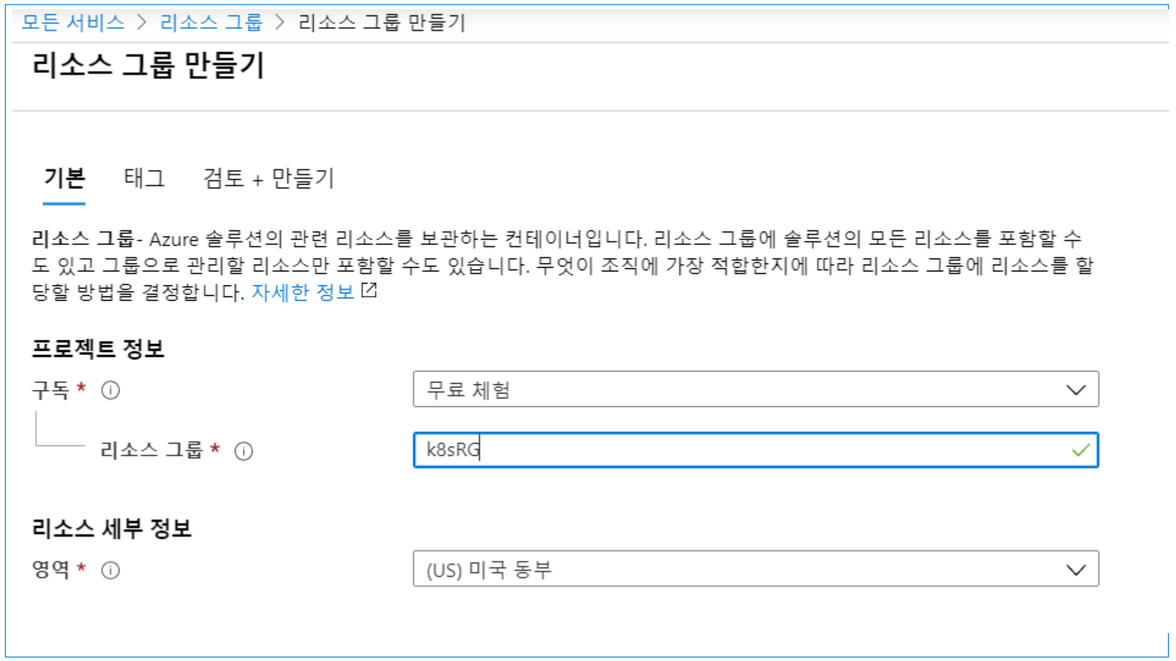




Azure Cloud Setup

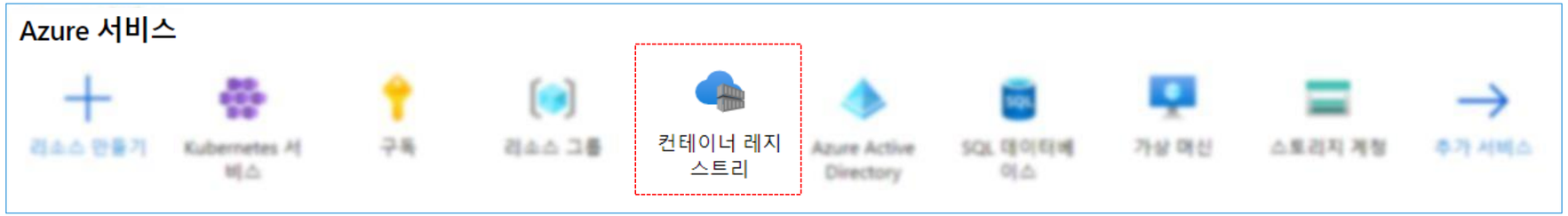
- 리소스 그룹 생성

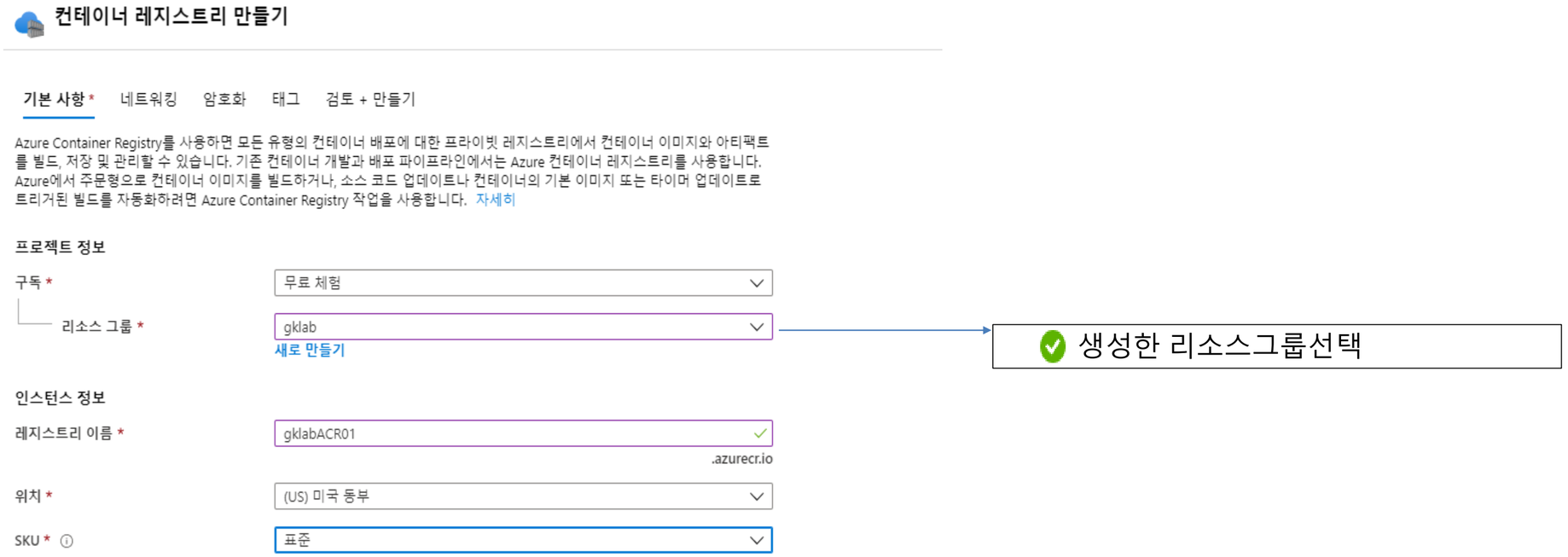




Azure Container Registry Service Setup

- Container Registry(컨테이너 레지스트리) 생성

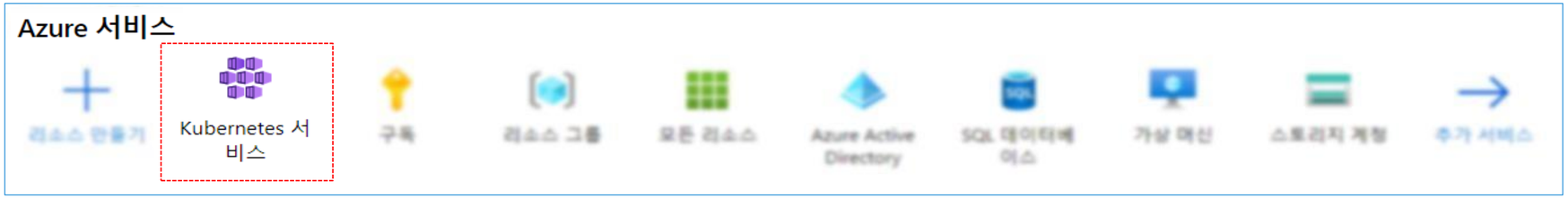




생성한 리소스그룹선택

Azure Kubernetes Service Setup(1/6)

- Kubernetes 서비스(Cluster) 생성



Kubernetes 클러스터 만들기

기본 사항 | 노드 풀 | 인증 | 네트워킹 | 통합 | 태그 | 검토 + 만들기

AKS(Azure Kubernetes Service)는 호스트된 Kubernetes 환경을 관리하여 컨테이너 오케스트레이션 전문 기술을 사용하지 않고도 컨테이너화된 애플리케이션을 쉽고 빠르게 배포하고 관리할 수 있도록 합니다. 또한 애플리케이션을 오프라인으로 전환하지 않고도 요청 시 리소스를 프로비전, 업그레이드 및 확장할 수 있어 지속적인 작업 및 유지 관리에 대한 부담이 없어집니다.
Azure Kubernetes Service에 대한 자세한 정보

프로젝트 정보

배포된 리소스와 비용을 관리할 구독을 선택합니다. 폴더 같은 리소스 그룹을 사용하여 모든 리소스를 정리 및 관리합니다.

구독 * : 무료 체험

리소스 그룹 * : gklab
새로 만들기

생성한 리소스그룹 선택

클러스터 세부 정보

Kubernetes 클러스터 이름 * : gklabcluster01

- 이름에는 문자, 숫자, 밑줄 및 하이픈만 사용할 수 있습니다. 이름은 문자나 숫자로 시작하고 끝나야 합니다.
- Kubernetes 서비스 이름은 현재 리소스 그룹에서 고유해야 합니다.

지역 * : (US) 미국 동부

Kubernetes 버전 * : 1.15.10(기본값)

주 노드 풀

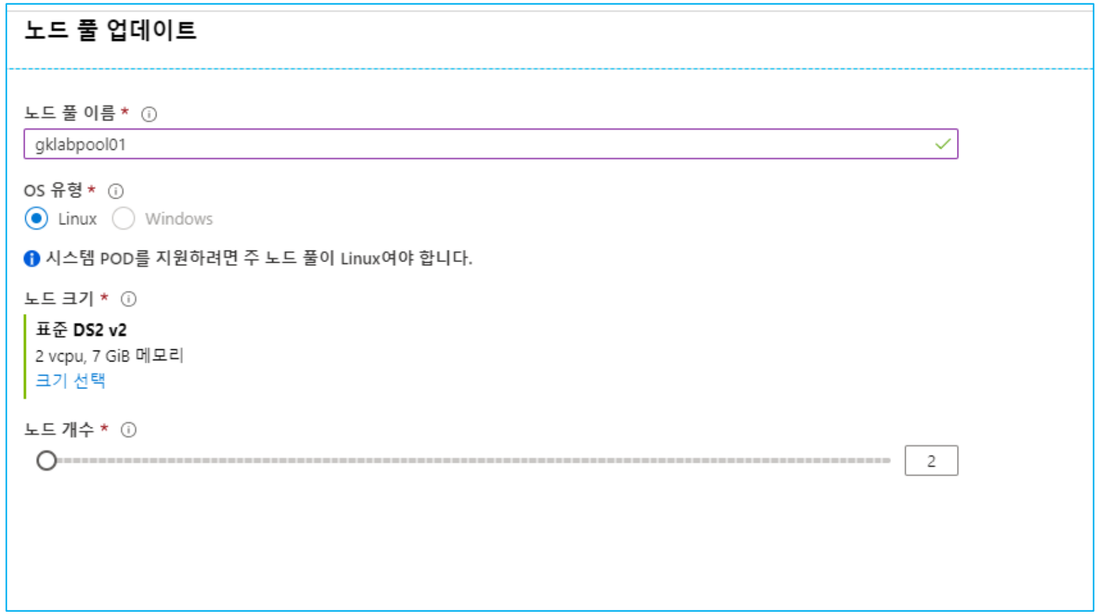
클러스터의 주 노드 풀에 있는 노드의 수와 크기입니다. 프로덕션 워크로드의 경우 복원력을 위해 3개 이상의 노드를 사용하는 것이 좋으며, 개발 또는 테스트 워크로드의 경우 노드가 하나만 필요합니다. 클러스터를 만든 이후에는 노드 크기를 변경할 수 없지만 만든 후 클러스터의 노드 수를 변경할 수 있습니다. 노드 풀을 추가하려면 "크기 조정" 탭에서 "X" 기능을 사용하도록 설정해야 합니다. 그러면 클러스터를 만든 후 노드 풀을 추가할 수 있습니다.
Azure Kubernetes Service의 노드 풀에 대한 자세한 정보

노드 크기 * : **표준 DS2 v2**
크기 변경

노드 개수 * : 3

Azure Kubernetes Service Setup(2/6)

- Kubernetes 서비스(Cluster) 생성

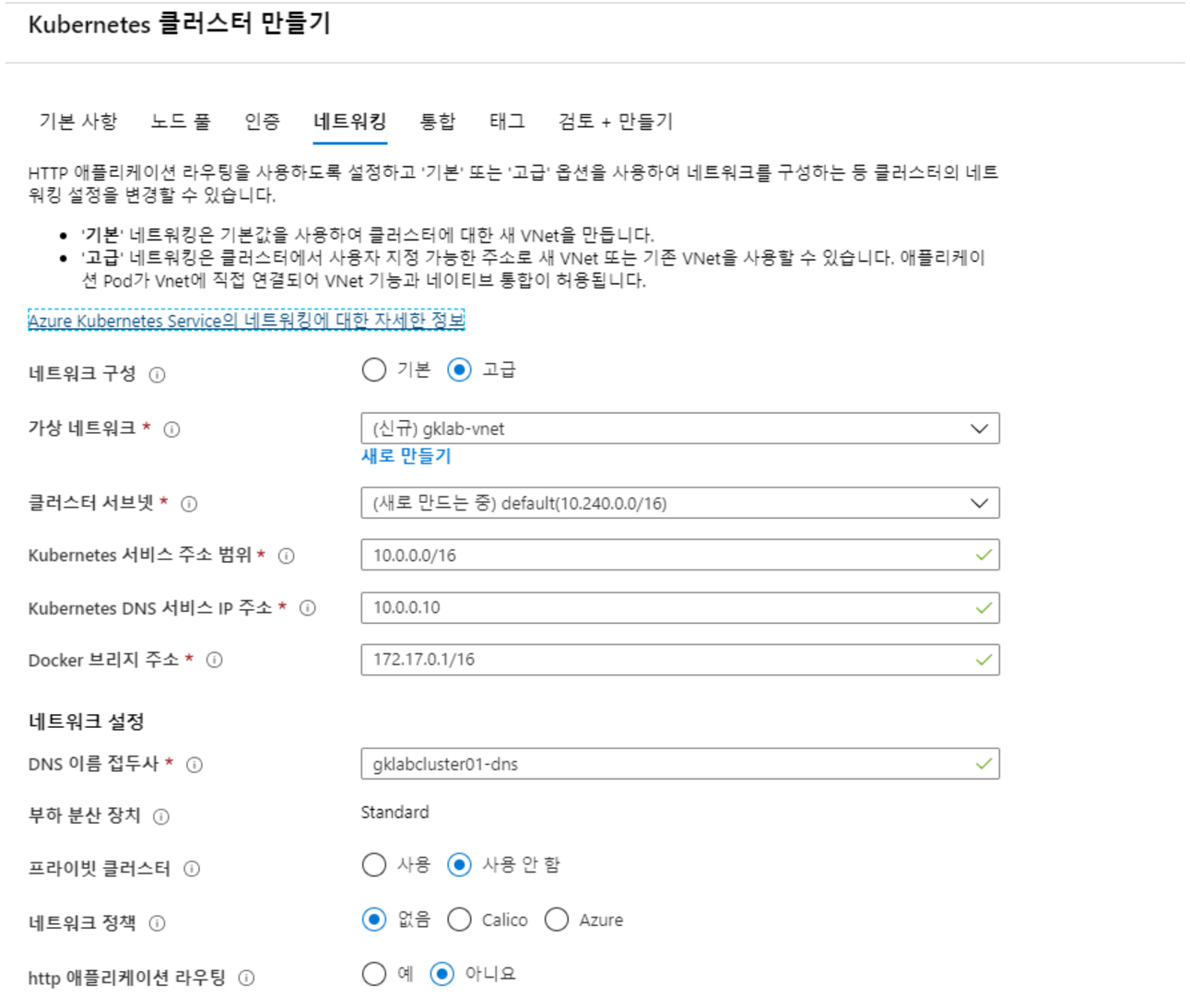


Azure Kubernetes Service Setup(3/6)

- Kubernetes 서비스(Cluster) 생성

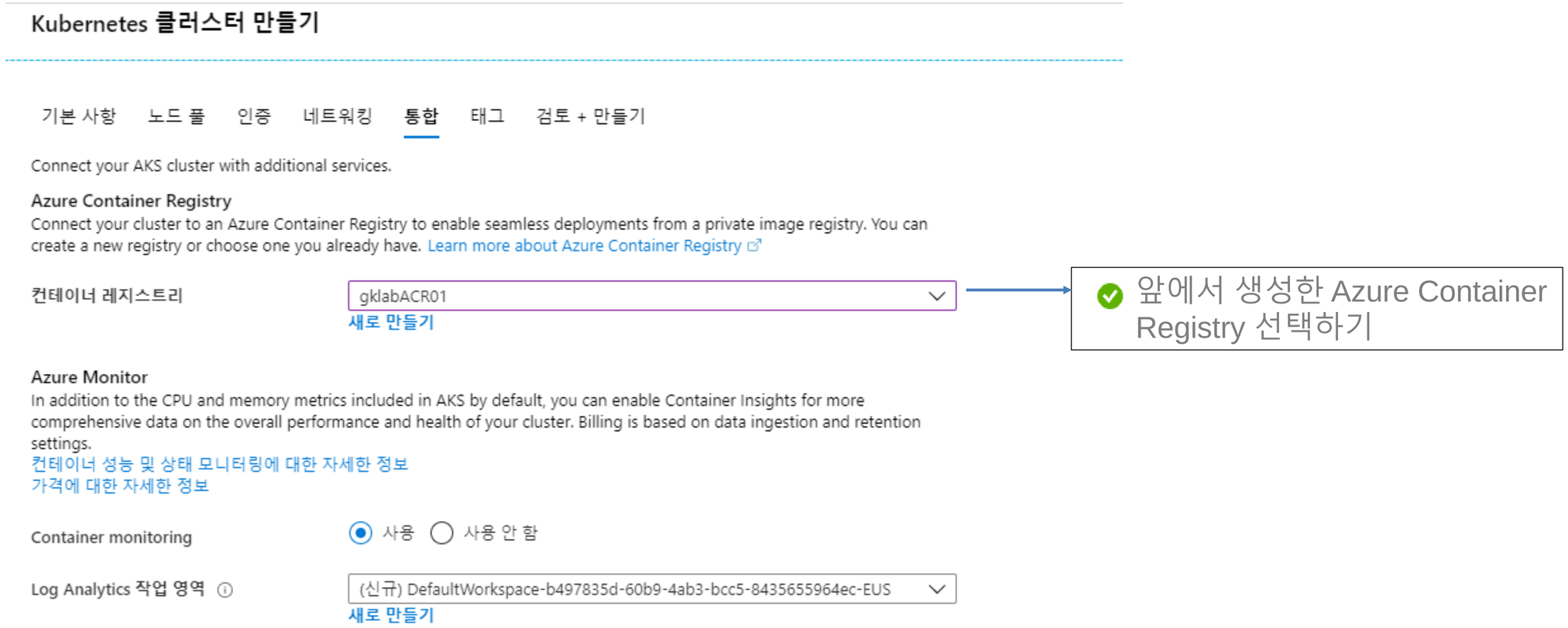
Azure Kubernetes Service Setup(4/6)

- Kubernetes 서비스(Cluster) 생성



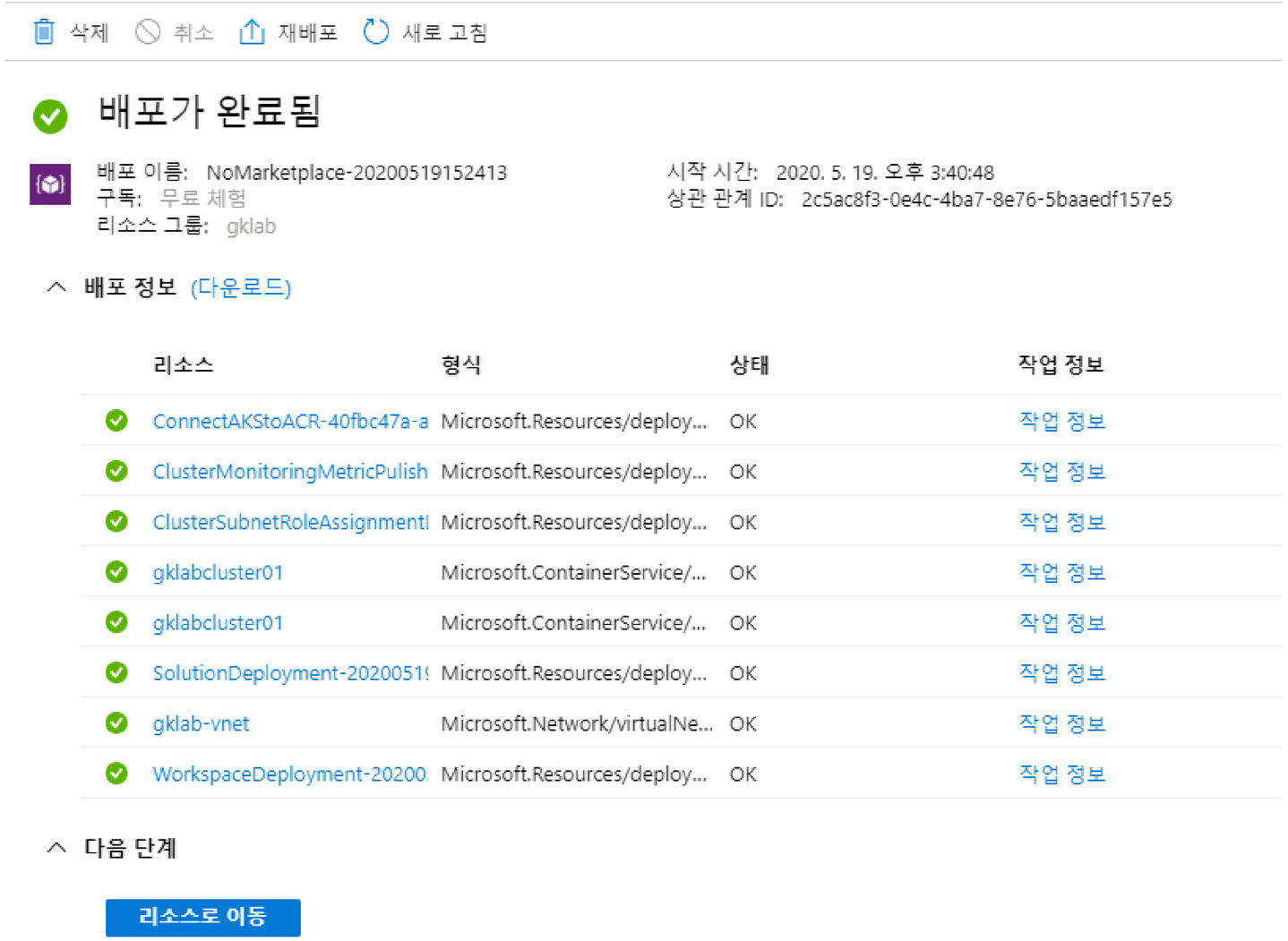
Azure Kubernetes Service Setup(5/6)

- Kubernetes 서비스(Cluster) 생성



Azure Kubernetes Service Setup(6/6)

- Kubernetes 서비스(Cluster) 생성



Azure CLI _ Kubernetes Client Install

- Azure-Cli 설치
  - curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash

- Azure Client 인증하기
  - az login # azure Cli authentication

- Kubectl 설치

  vim /etc/apt/sources.list.d/kubernetes.list

  curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -

  apt-get update

  apt-get install –kubectl

  sudo apt-get update

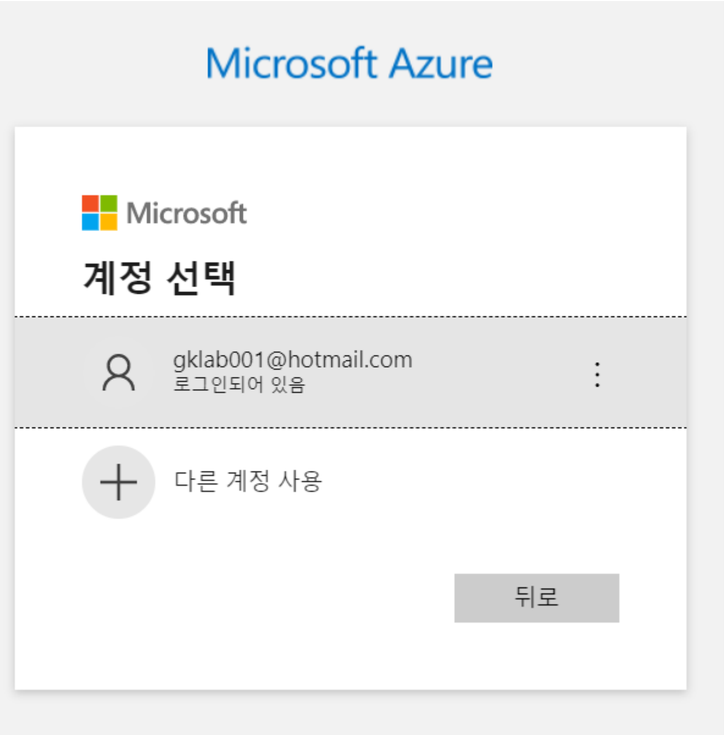
  apt-get install –y kubectl

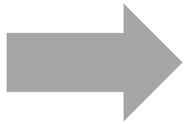
- 설치 확인

  kubectl version --client

Azure-Cli tool Install & Config

- Azure Client 인증하기
  - az login # azure Cli authentication



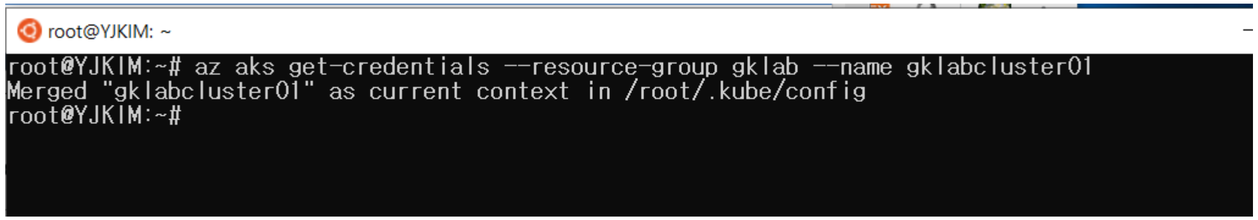


```
You have logged in. Now let us find all the subscriptions to which you have access...
[
  {
    "cloudName": "AzureCloud",
    "id": "854e08fb-f4a4-411f-af61-23cc002244e7",
    "isDefault": true,
    "name": "무료 체험",
    "state": "Enabled",
    "tenantId": "059633c1-a262-499b-9378-818bf42543e4",
    "user": {
      "name": "apexacme@uengine.org",
      "type": "user"
    }
  }
]
```

Subscription 정보가 출력

Azure AKS Connection

- Local에 Azure AKS Token 생성
  - az aks get-credentials --resource-group *My_Resource_Group* --name *My-cluster*



- AKS 연결 확인
  - kubectl config view
  - kubectl cluster-info
  - kubectl get nodes

```
root@YJKIM:~/gowebapp/gowebapp-mysql# kubectl get nodes
NAME                                STATUS   ROLES   AGE     VERSION
aks-gklabpool01-99057696-vmss000000   Ready    agent   5h21m   v1.15.10
aks-gklabpool01-99057696-vmss000001   Ready    agent   5h21m   v1.15.10
root@YJKIM:~/gowebapp/gowebapp-mysql#
```

Default kubernetes service 확인

Azure Container Registry

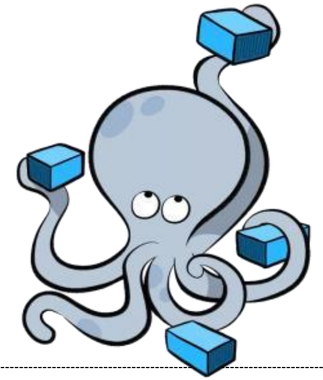
- Azure CLI
  - az acr login --name *[생성하신 azure container registry service 이름]*

```
root@YJKIM:~/gowebapp/gowebapp-mysql# az acr login --name gklabACR01
Uppercase characters are detected in the registry name. When using its server url in docker commands, to avo
ation errors, use all lowercase.
Login Succeeded
root@YJKIM:~/gowebapp/gowebapp-mysql# docker tag gowebapp:v1 gklabacr01.azurecr.io/gowebapp:v1
```

- Integrate AKS with ACR
  - az aks update -n *My-cluster* -g *My_Resource_Group* --attach-acr *My_Acr_name*

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



RANCHER

Nomad

docker SWARM

MESOS

kubernetes

Container Orchestrators

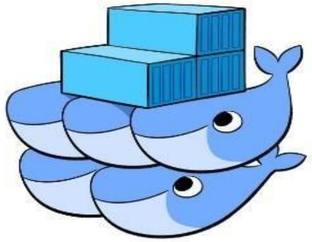
Kubernetes

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 작동하고 설정이 용이

Docker명령어와 Compose를 그대로 사용 가능

Apache Mesos

수만 대의 물리적 시스템으로 확장할 수 있도록 설계

Hadoop, MPI, Hypertable, Spark같은 응용 프로그램을 동적 클러스터 환경에서 리소스 공유와 분리를 통해 자원 최적화 가능

Docker 컨테이너를 적극적으로 지원

Kubernetes (k8s)란 ?

- 컨테이너 오케스트레이션 도구
- http://kubernetes.io

  “ Open-Source software for automating deployment , scaling, and management of containerized application “

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

- 실행하기 쉽지만 잠재적으로의 통합은 복잡해질 수 있음
- 2014년 구글의 보그(borg) 프로젝트 로부터 시작되어 현재는 CNCF(Cloud Native Computing Foundation)에서 관리 (2015년 7월21일 v1.0 출시)
- 오픈소스이며 확장 가능 (지금도 수많은 Contributor 들에 의해서 업데이트가 진행중)
- 그리스어인 κνβερνητης 를 어원으로 하며 파일럿 또는 Helmsman 이라는 의미
- 베어메탈 또는 가상머신 또는 퍼블릭클라우드로 부터 서비스형태로 제공받을수 있다.
- Azure (Azure Kubernetes Service) , AWS(Elastic Kubernetes Service) , GCP(Google Kubernetes Engeine)

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

https://kubernetes.io/ko/docs/concepts/overview/what-is-kubernetes/

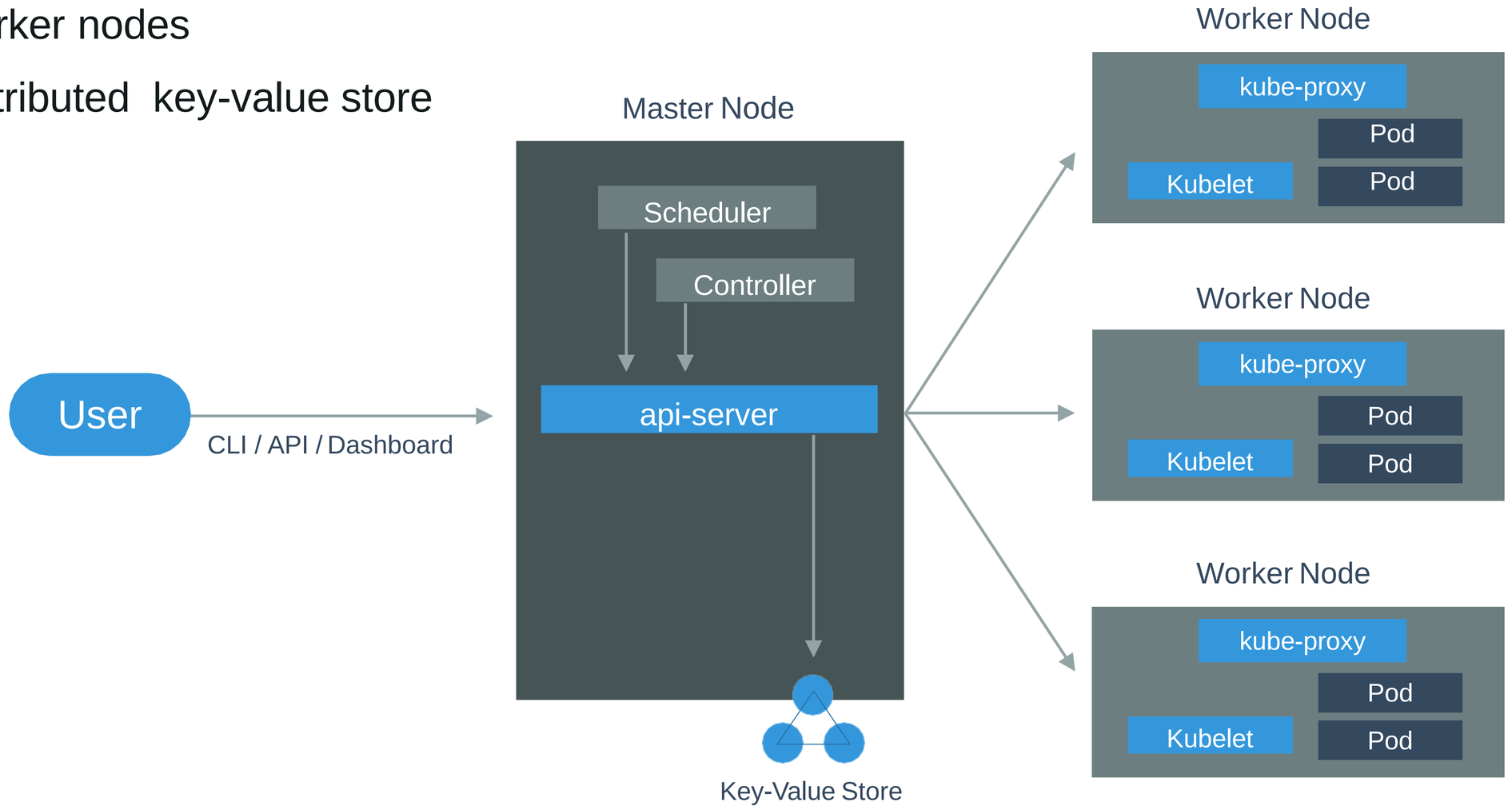
Kubernetes key Features (2/2)

- **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, Github와 같은 저장소에 노출시키지 않고도 어플리케이션 내에서 보안정보 공유 가능
- **Storage Orchestration**
  - Local storage를 비롯해서 Public Cloud(Azure, GCP, AWS), Network storage등을 용도에 맞게 자동 mount 가능

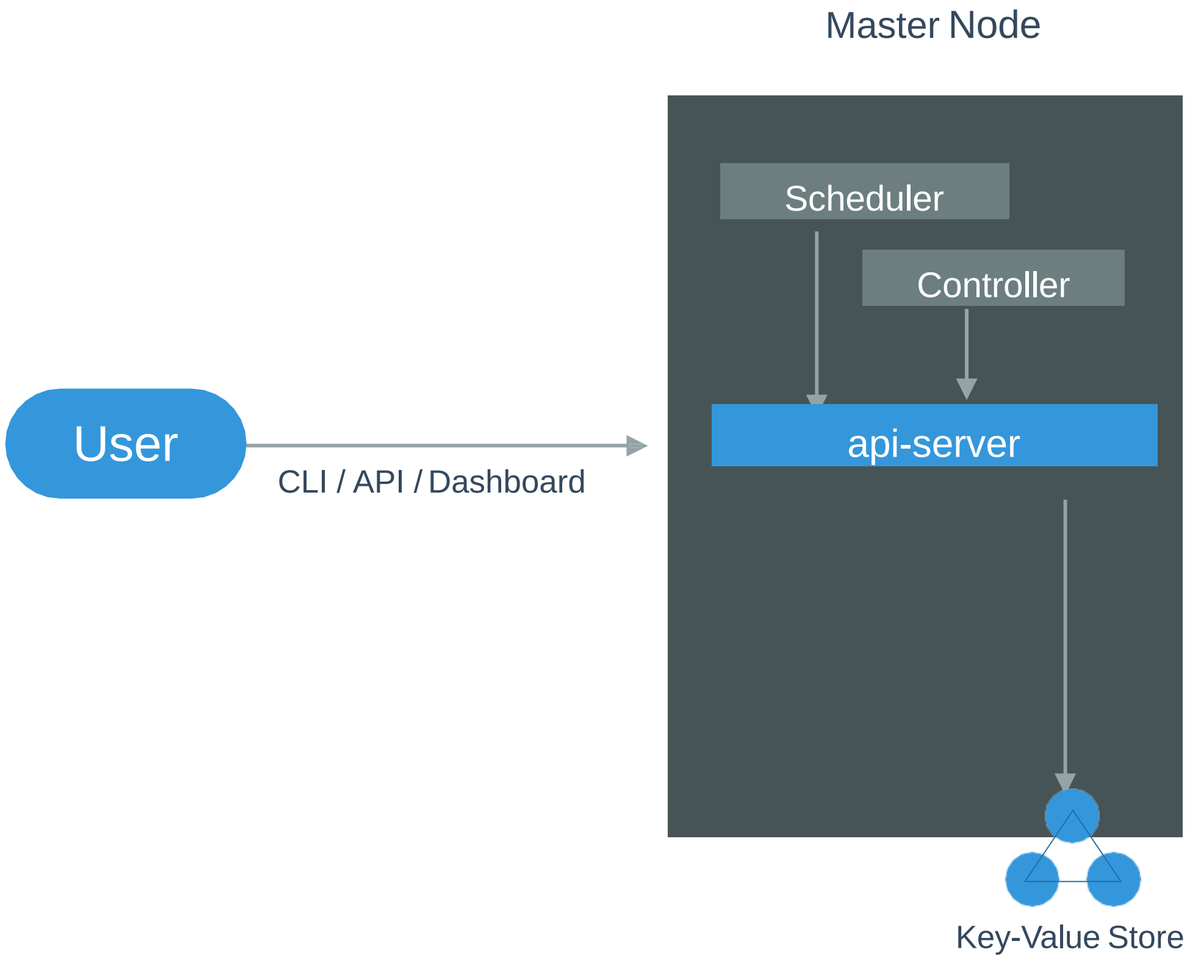
https://kubernetes.io/ko/docs/concepts/overview/what-is-kubernetes/

Kubernetes Architecture

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



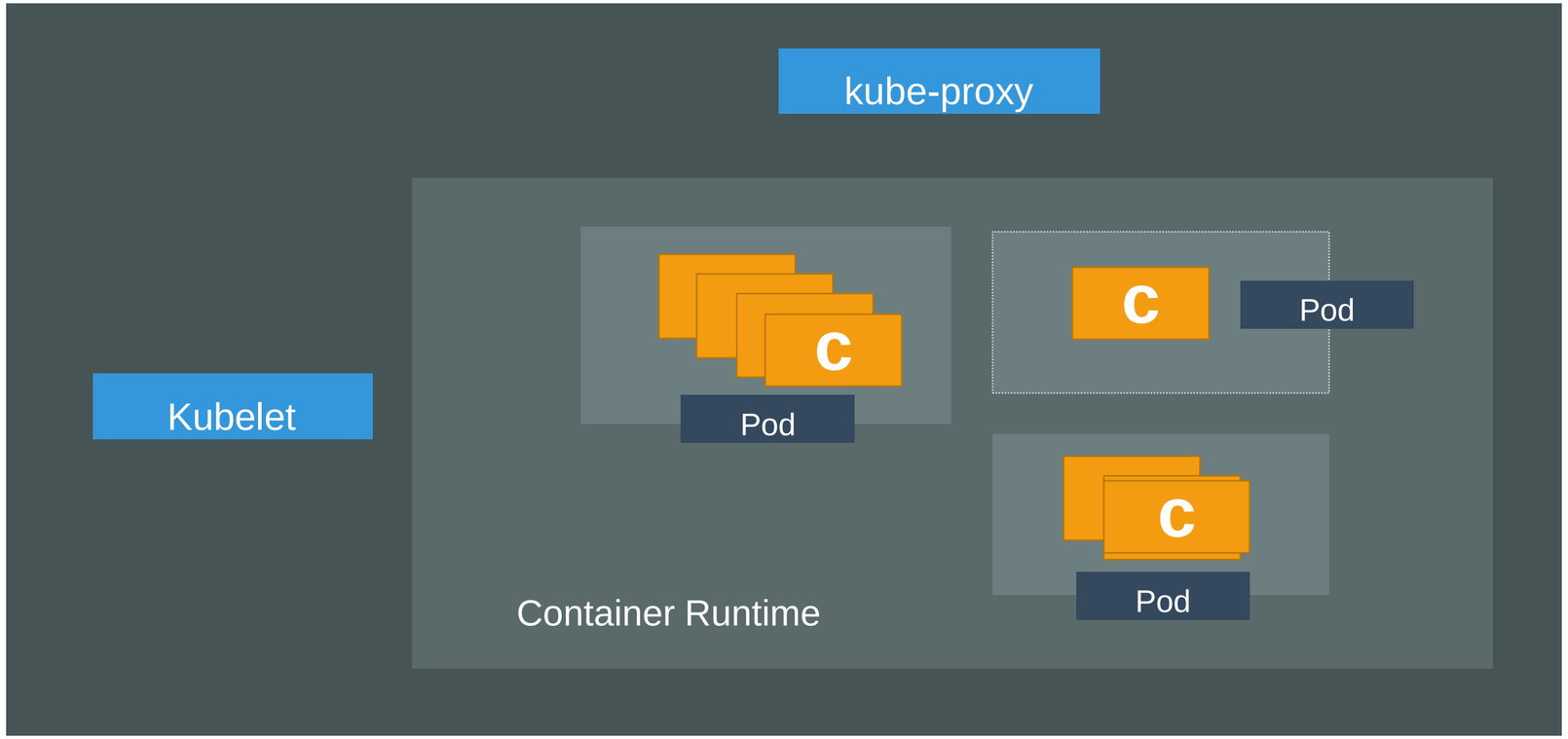
Master Node 구 성 요 소



- 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

Worker Node 구 성 요 소



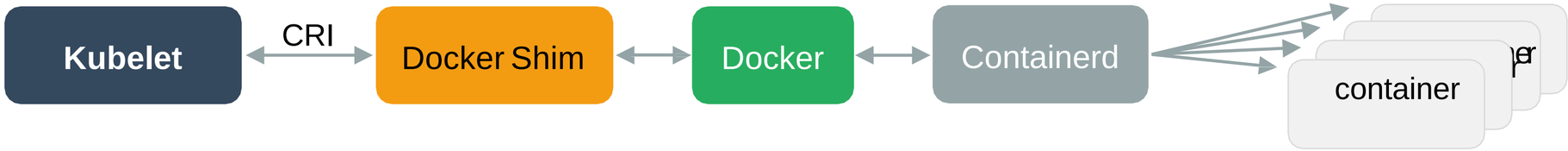
- Container runtime
- kubelet
- kube-proxy

Worker Node

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

Worker Node 구 성 요 소

Kubelet : CRI Shims

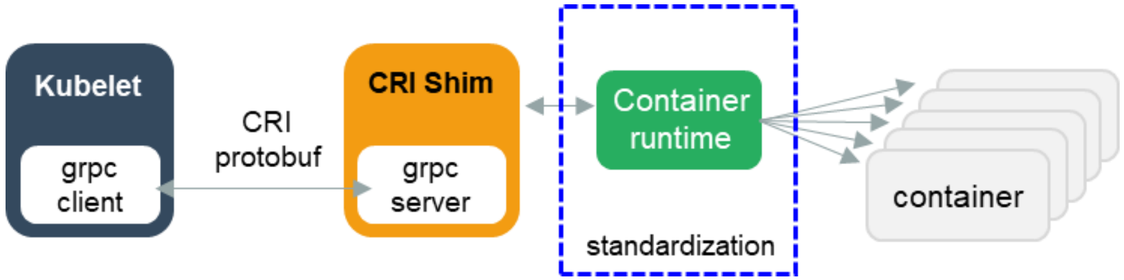


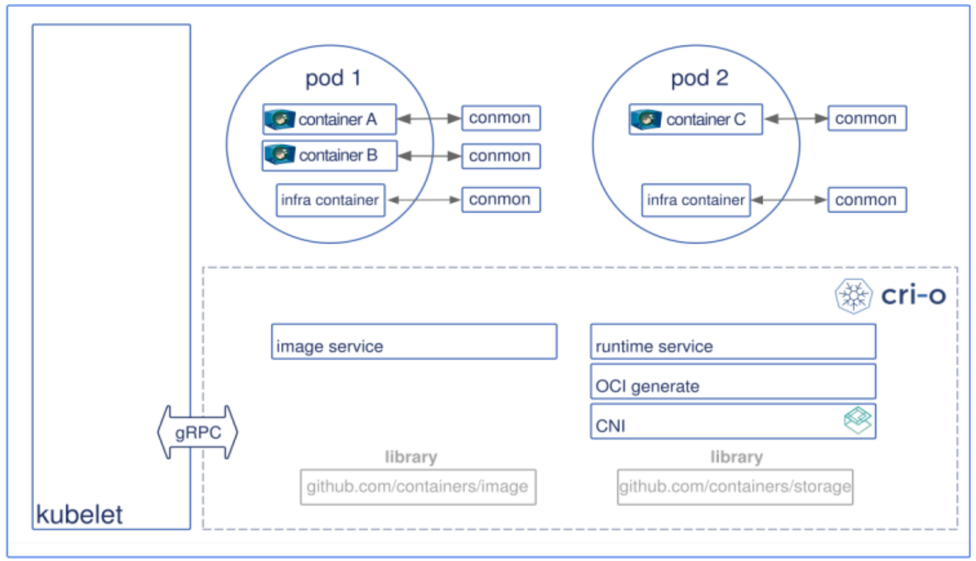
- **Docker shim**
  Docker shim은 Worker node에 설치된 Docker를 이용하여 컨테이너를 생성

- 내부적으로는 Docker는 containerd를 이용해 컨테이너를 관리

Worker Node 구 성 요 소

Kubelet : CRI-O





- **Container Runtime** 표 준 화
- 각 Container마다 CRI Shim을 개발해야 하는 비용적 이슈에서 CRI-O 등장
- Container runtime이 OCI(Open Container Initiative)에서 정의한 표준을 따르면 CRI-O를 CRI shim 으로 사용 가능

Kubernetes Core Concept : “Desired State”

Imperative : action 을 정의한다

Declarative : 원하는 상태(Desired State) 를 정의한다.

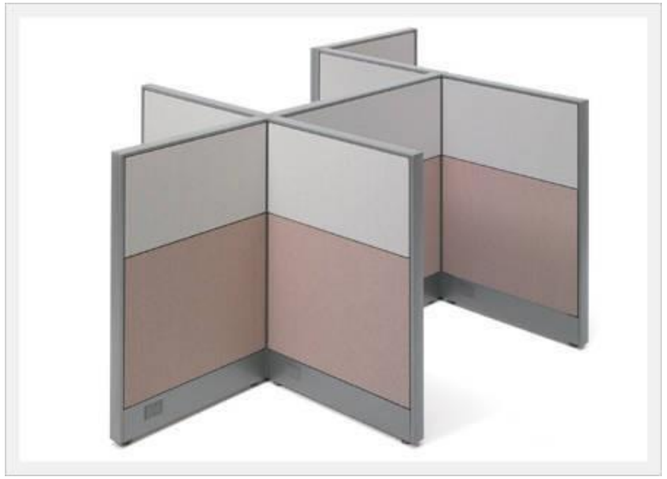
- **Kubernetes**는 **Current State**을 모니터링하면서, **Desired State**를 유지하려는 습성
- 직접적인 동작을 명령하지 않고, 원하는 상태를 선언(Not Imperative, But Declarative)
  - Imperative – “nginx 컨테이너를 3개 실행해줘, 그리고 80포트로 오픈해줘.”
  - Declarative – “80포트를 오픈한 채로, nginx 컨테이너를 3개 유지해줘.”

Kubernetes Object, Controller and Kubectl

- Object : K8s의 상태를 나타내는 엔터티로 K8s API의 Endpoint
  - 유형 - Pod, Service, Volume, Namespace 등
  - Spec과 Status 필드를 가짐 - Spec(Desired State), Status(Current State)
  - https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.18/
- 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”

Namespaces

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

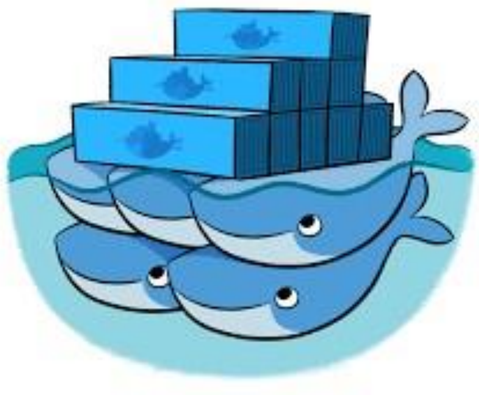
- Kubernetes는 처음에 3개의 초기 네임스페이스를 가진다.
  - **default** : 다른 namespace를 갖는 다른 객체들을 가지고 있다.
  - **kube-public** 은 클러스터 bootstrapping 같은 모든 유저가 사용가능한 특별한 namespace 이다.
  - **kube-system:** Kubernetes system에 의해서 생성된 객체를 가지고 있다.
- Resource Quotas를 사용하여 namespace 내에 존재하는 자원들을 나눌 수 있다.

Namespaces

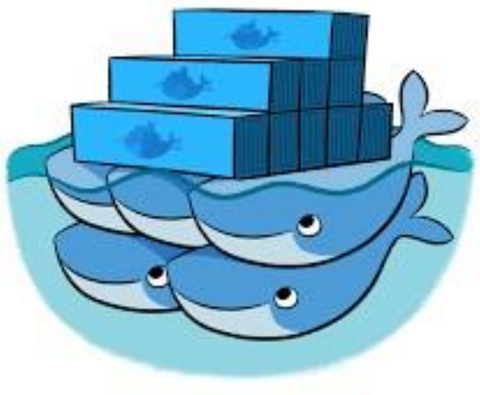
- 새로운 네임스페이스 생성
  - kubectl create namespace foo
- 원하는 Namespace에 Pod 생성
  - kubectl create deploy nginx --image=nginx --namespace=<insert-namespace-name-here>
- 특정 Namespace상에 생성된 Pod 조회
  - kubectl get pods --namespace=<insert-namespace-name-here>
- Namespace Option 생략 시, default Namespace
- default 네임스페이스를 다른 네임스페이스로 변경하고자 한다면
  - kubectl config set-context –current –namespace=<insert-namespace-name-here>

Pod : Kubernetes 최소 배포 단위

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



Pod

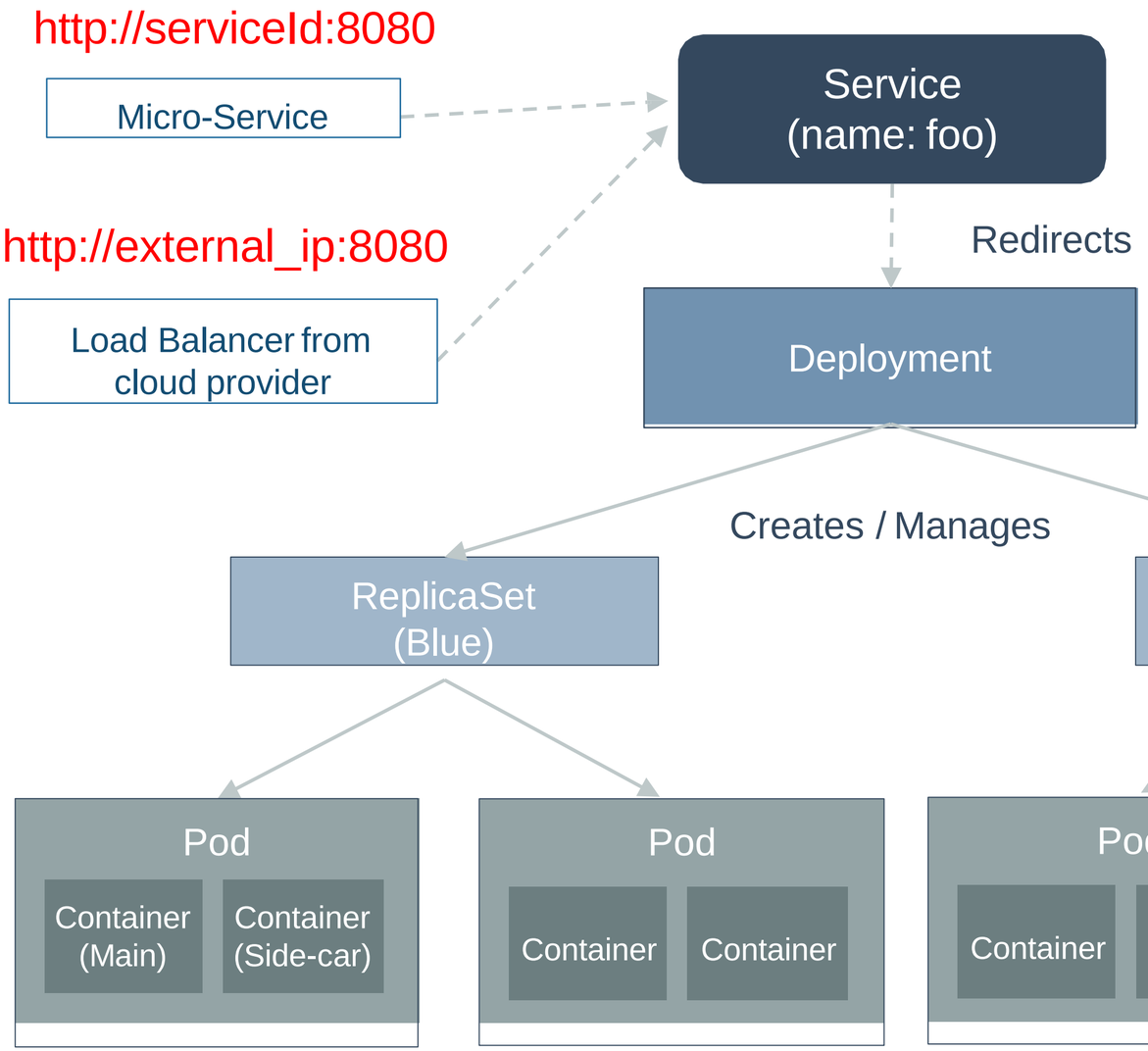


Pod

Pod : Kubernetes 최소 배포 단위

- 한 Pod은 내부에 여러 컨테이너를 가질 수 있지만 대부분 1~2개의 컨테이너 가짐
  - scaling이 컨테이너가 아닌 Pod단위로 수행되기 때문에, Pod내부에 다수의 컨테이너들이 타이트하게 묶여 있으면 스케일링이 쉽지 않거나 비효율적으로 이뤄지는 문제
- 1개의 Pod은 1개의 노드위에서 실행
- Deployment 에 의해서 배포된 Pod은 ReplicaSet Controller에 의해 복제생성됨
  - 이때 복수의 Pod는 Master의 Scheduler에 의해 여러 개의 Node에 걸쳐 실행
- Pod의 외부에서는 Service 를 통해 Pod에 접근
  - Service를 Pod에 연결했을 때 Pod의 특정 포트가 외부로 expose

Kubernetes Object Model



Responsible for :

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

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

- Keep replica count as desired (replicas=2)

- Service Hosting

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** : Pod 의 desired state 를 명시. 예제에서는 nginx:1.7.9 이미지 컨테이너를 생성.

```
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
```

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,depeloyments,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*”

이미지를 통한 어플리케이션 배포

- 현재 작동 중인 pod들이 없는지 확인
  - kubectl get pods

- Nginx이미지 예제로 배포하기
  - kubectl create deploy first-deployment --image=nginx

- 실행된 Pods 확인
  - kubectl get pods
  - 각 Pod들은 ‘Pod명–{hash}’ 형식의 고유한 이름을 가짐

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 Shell 명령어 사용)
  - 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

설정 파일을 통한 pod 배포 (1/2)

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

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

설정 파일을 통한 pod 배포 (2/2)

- vim declarative-pod.yaml 파일을 직접 작성 후 배포
  - kubectl apply -f declarative-pod.yaml

    (-f 는 파일 경로를 설정해서 배포할 수 있는 옵션이다.)

- 배포된 Pod를 검색 후 접속
  - 이름이 설정대로 declarative-pod 로 생성됨을 확인

    kubectl get pods

  - 생성된 Pod에 접속

    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)
  - vim dev-pod.yaml
- 설정파일을 기반으로 pod 배포
  - kubectl apply -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
```

원하는 노드 타입에 Pod 몰기 (2/2)

- Pod가 Create 되지 않고, Pending 상태 => disktype=ssd 레이블을 가지는 노드가 없으면 배포가 pending 된다.

- 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 pods -o wide

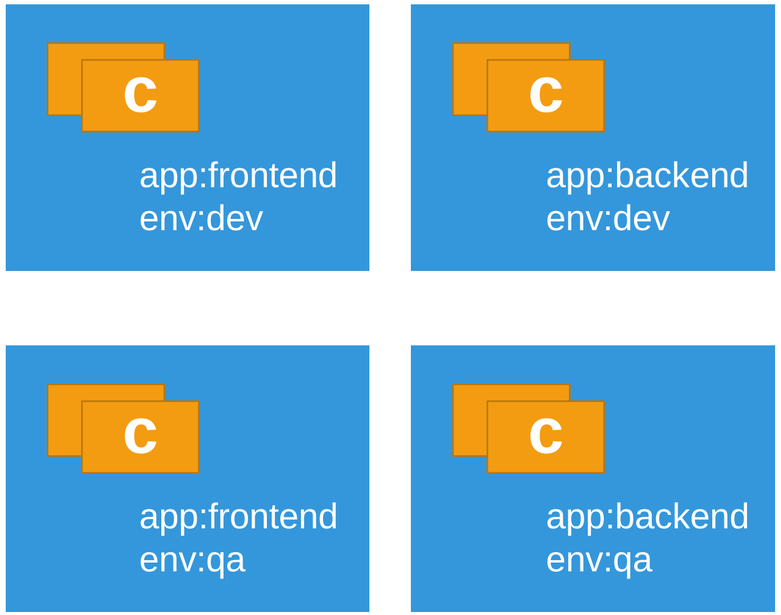
Pod Initialization

- Init.yaml 파일 생성
  - vim pod-initialize.yaml
- 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: {}
```

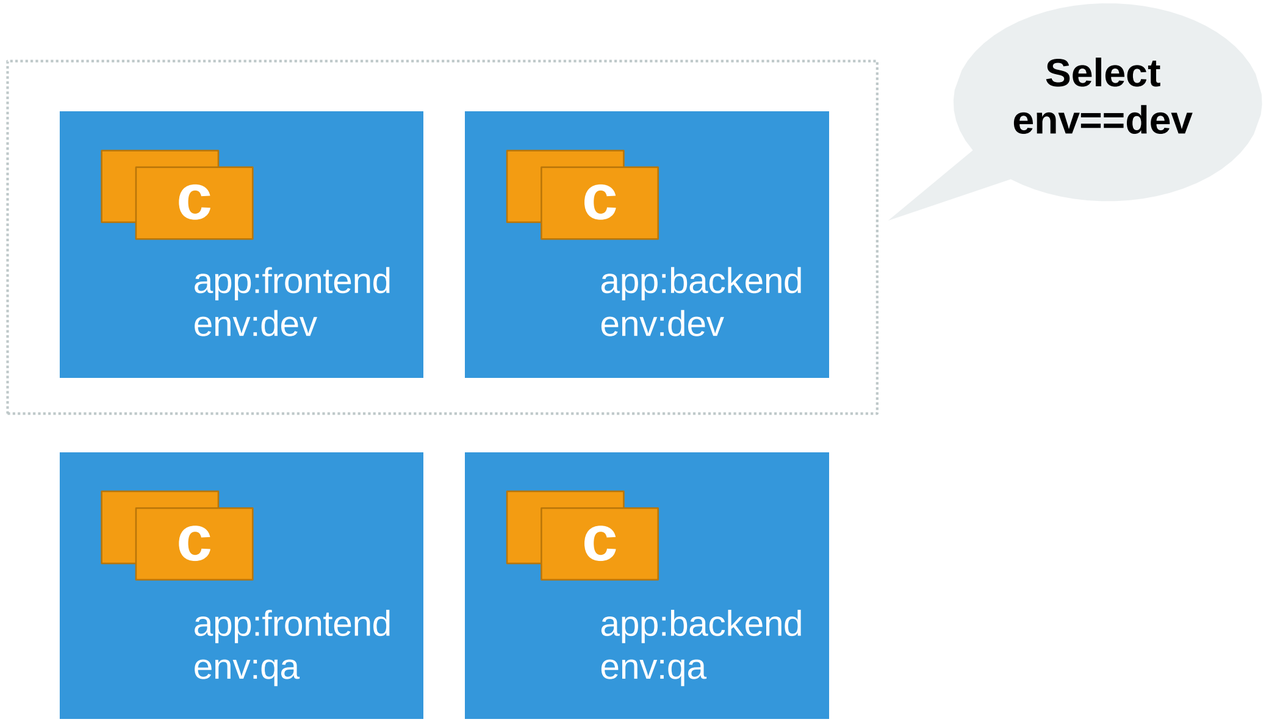
Pod 초기화 시점에 실행

Labels



- Labels 은 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.

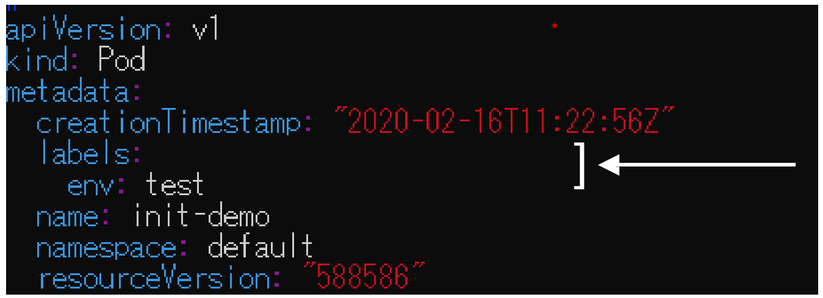
Label Selectors



- 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.

Label 연습

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



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 pod --selector 'env in(test, test1), app in (nginx, nginx1)’ # and 연산
- kubectl get pod --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
  annotataions:
    kubernetes.io/change-cause=nginx:1.7.9
spec:
  replicas: 3
  selector:
        ……..
```

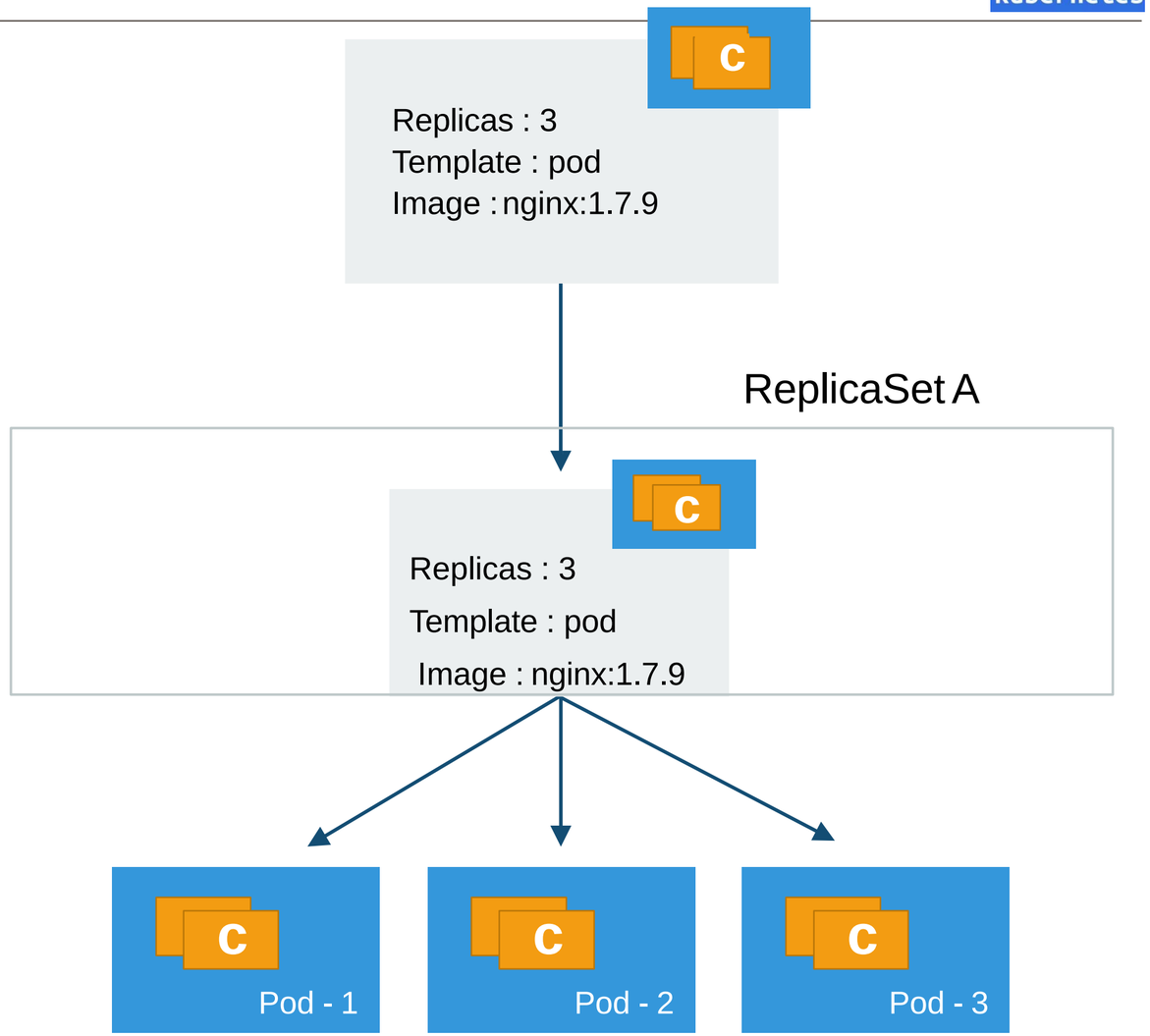
One-dash, Double-dash

- 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)

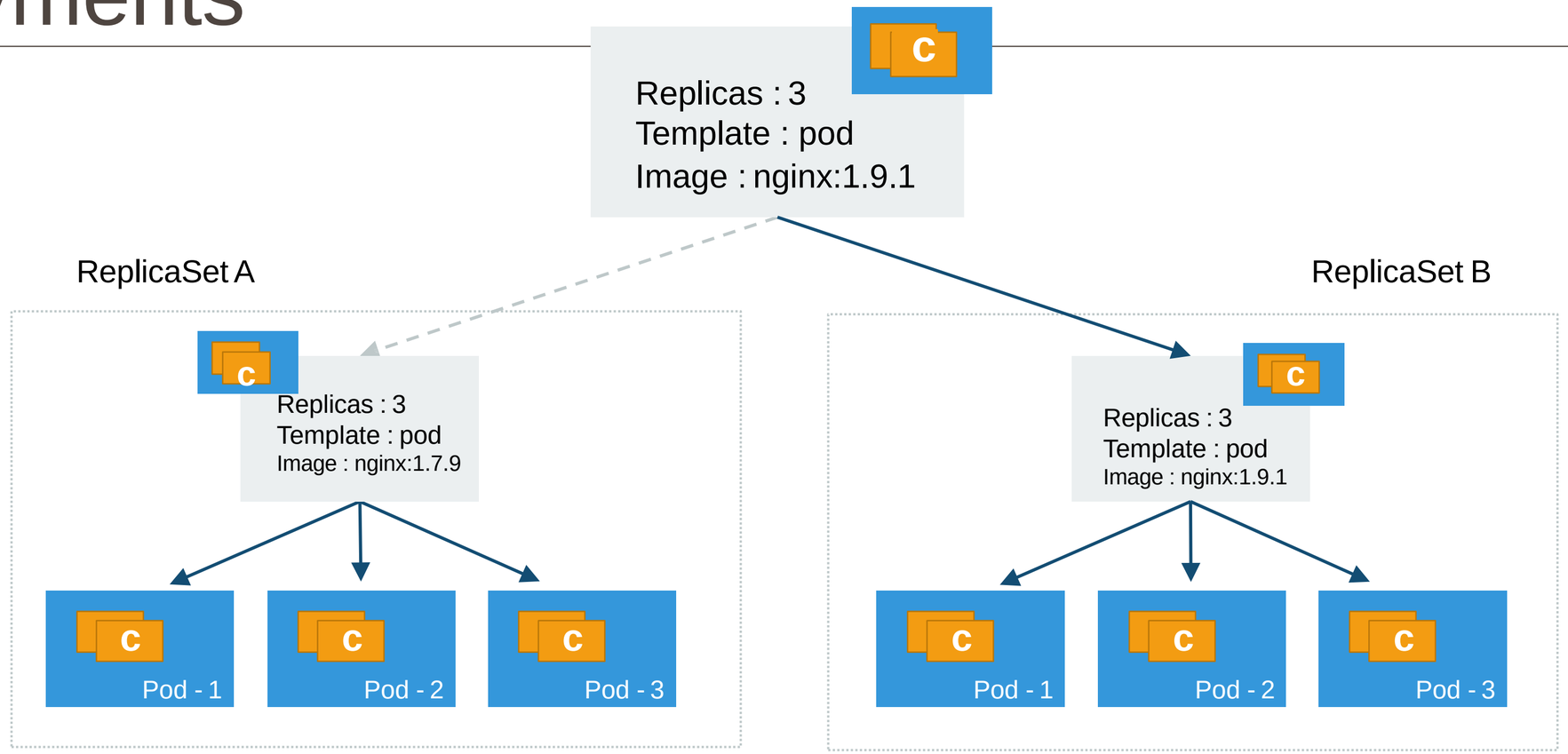
- In Pod Selector, short form (-l) equals long form(--selector)

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 Pod 들을 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.

Deployment 생성

- 설정 파일을 생성
  - vim nginx-deployment.yaml
- 파일을 기반을 배포
  - kubectl apply -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
  - vim nginx-deployment.yaml
- 변경한 파일을 적용
  - kubectl apply -f nginx-deployment.yaml
- 변경 내용을 확인
  - kubectl get pods

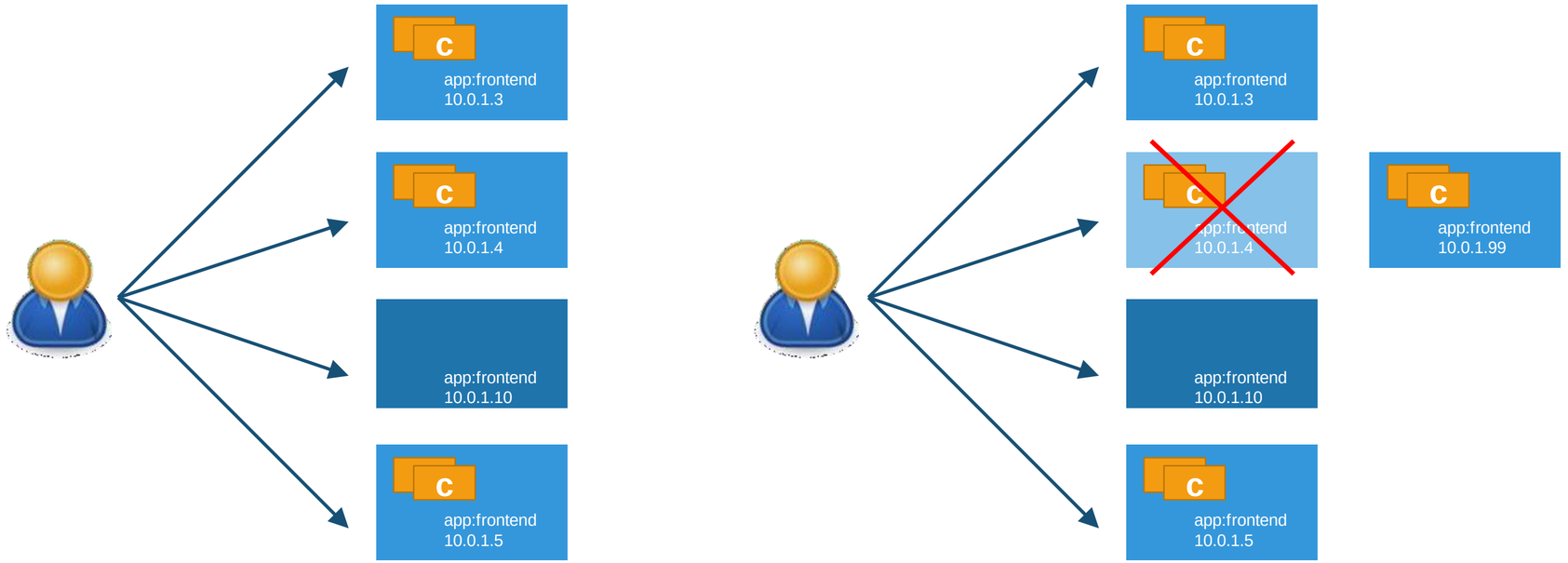
Rolling update

- 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

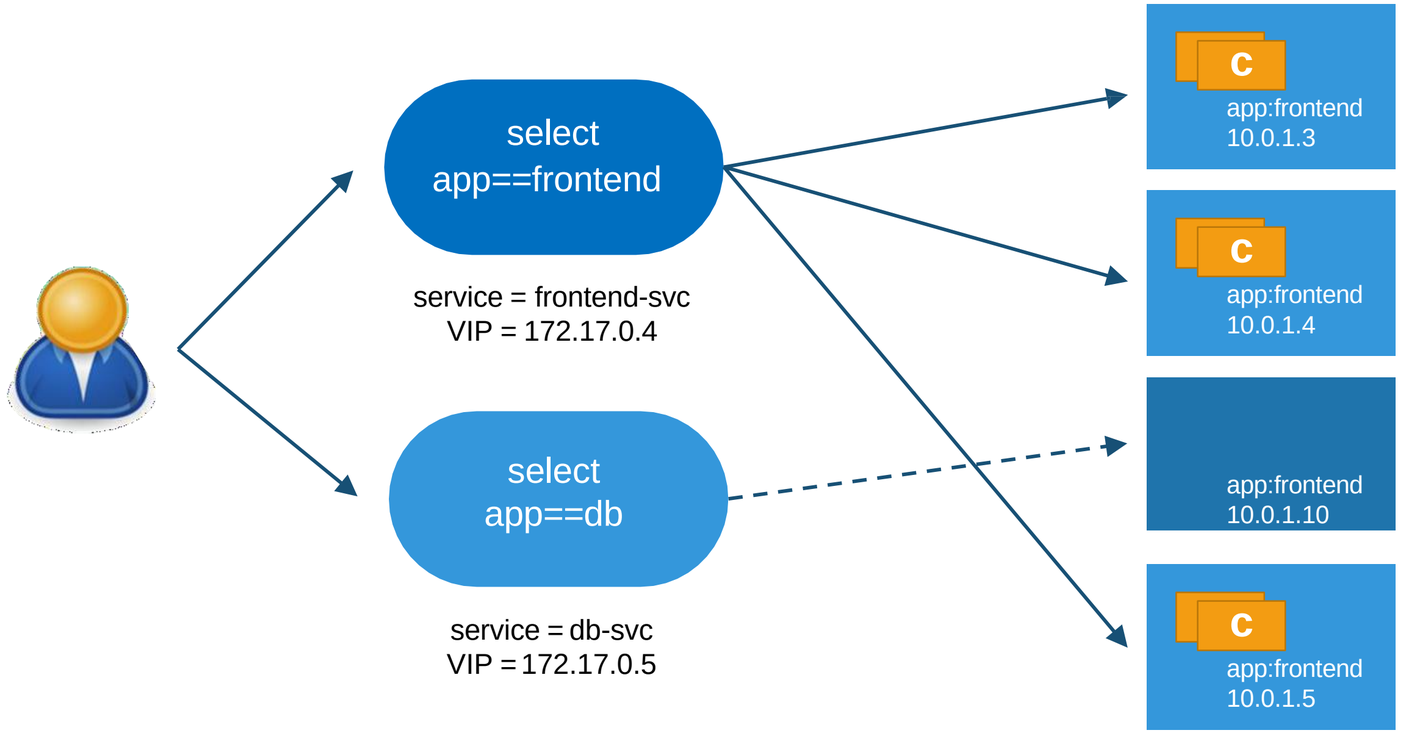
- 현재 실행중인 객체들을 확인
  - kubectl get pods
    kubectl get deployments -o wide # deployment에 적용된 Image:버전 추가 표시
- 객체를 롤백 처리
  - kubectl rollout undo deployment/nginx-deployment
- 진행을 확인
  - kubectl get deployments -o wide

Pod Access Issues



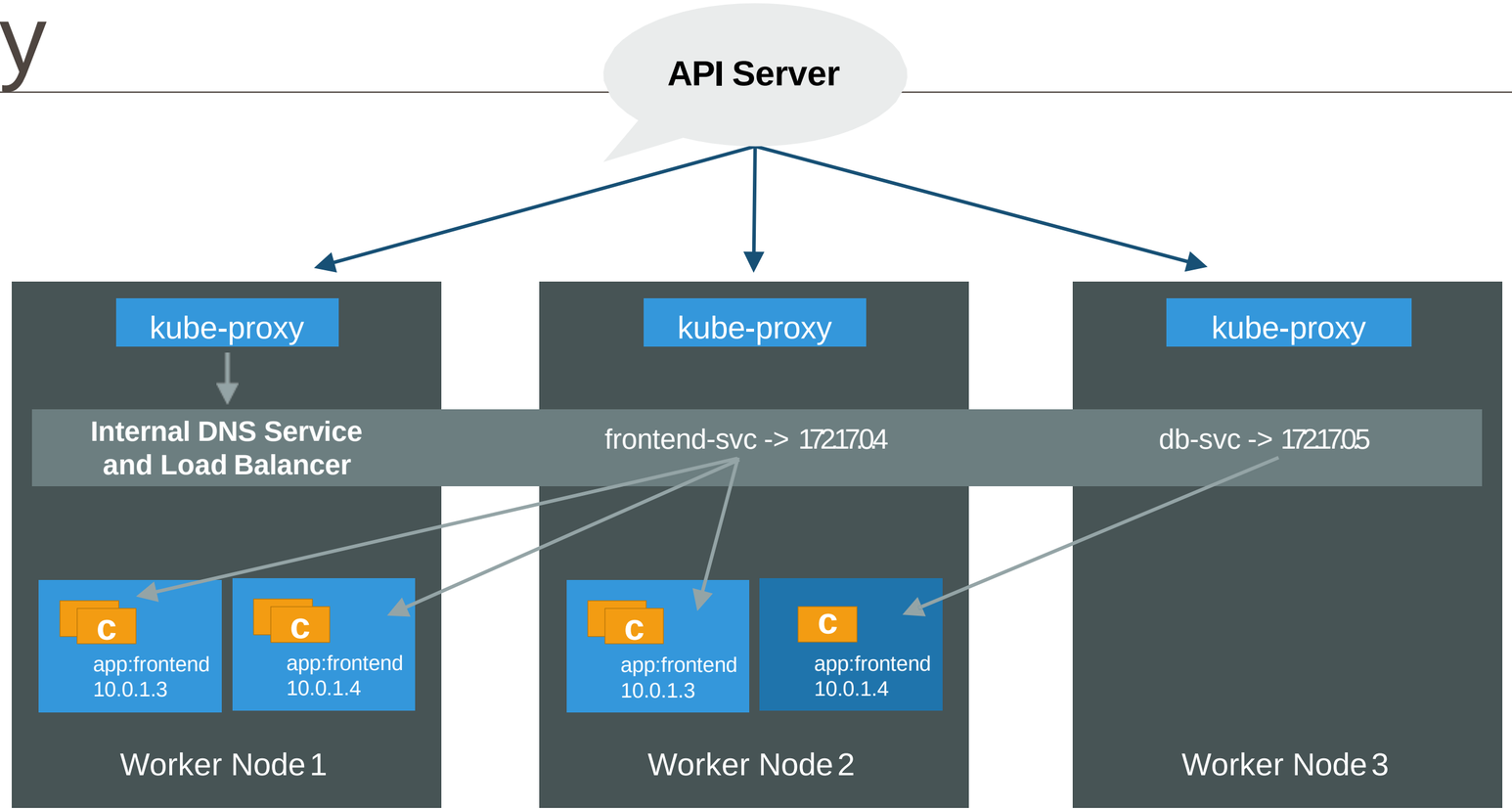
- 어플리케이션에 접근하기 위해서는, 사용자가 Pod에 접근해야 한다.
- Pod들은 언제든지 소멸 가능하기에 IP주소는 고정되어 있지 않다.
- 사용자가 직접 IP주소로 Pod에 연결되어 있을 때, Pod가 죽어서 새로 만들어지면 접속할 방법이 없다.
- 이 상황을 극복하기 위해서 추상화를 통해 Service라는 Pod들의 논리적 집단을 만들어 규칙을 설정하고 사용자 들은 여기에 접속을 한다.

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 가능
    한데, 이 때 DNS 에서 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?

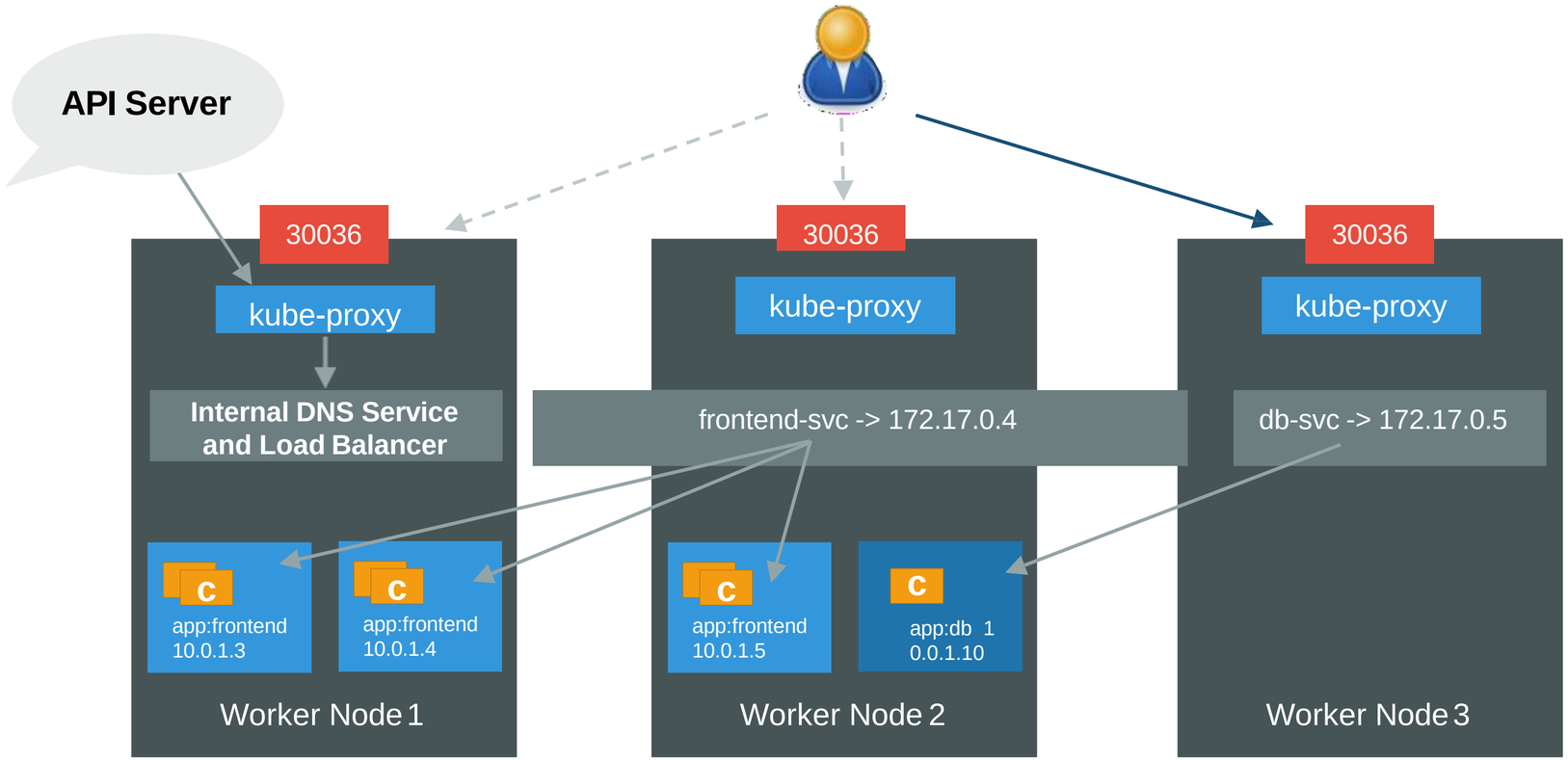
- Service 생성 시, IP주소 할당 방식과 서비스 연계 등에 따라 구분
  - ClusterIP
  - NodePort
  - LoadBalancer

ServiceType : ClusterIP

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

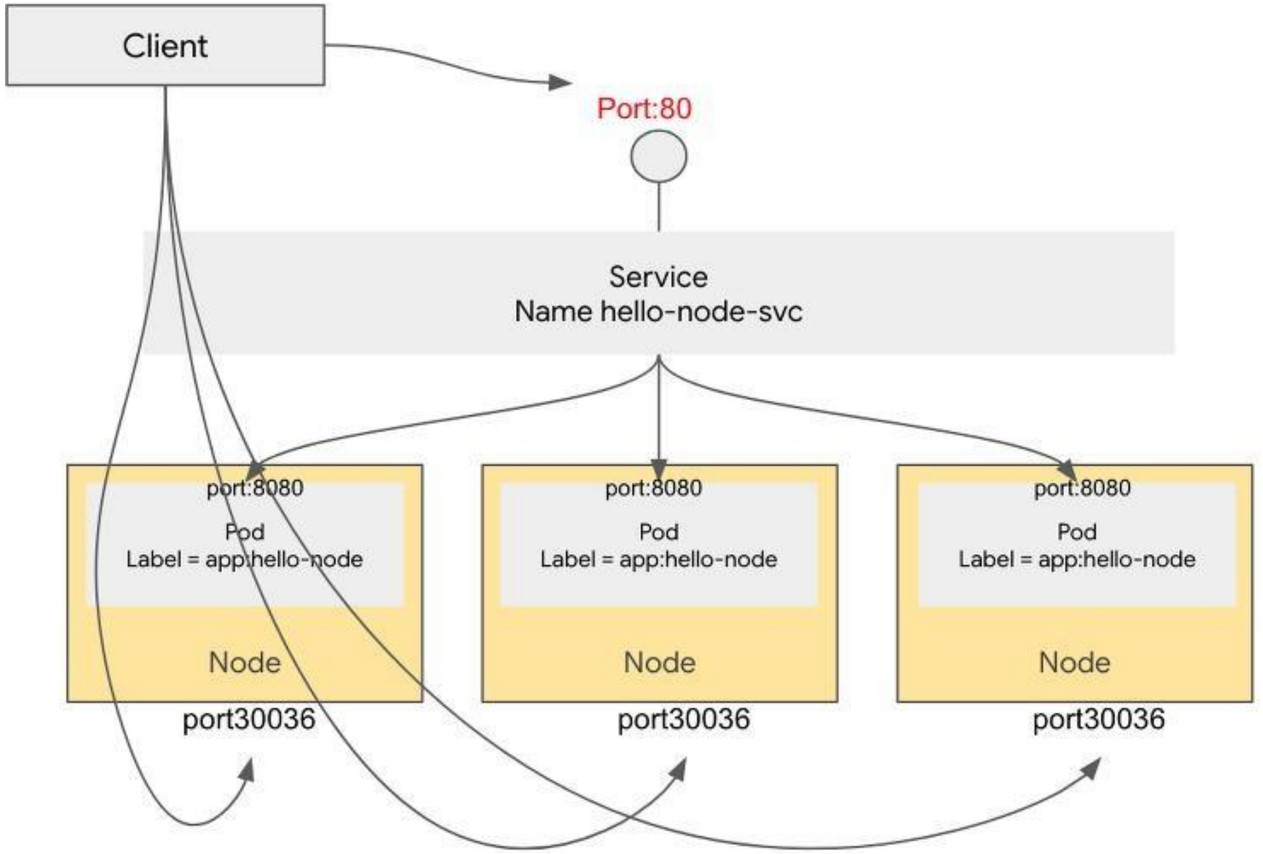
ServiceType : NodePort

- 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
```



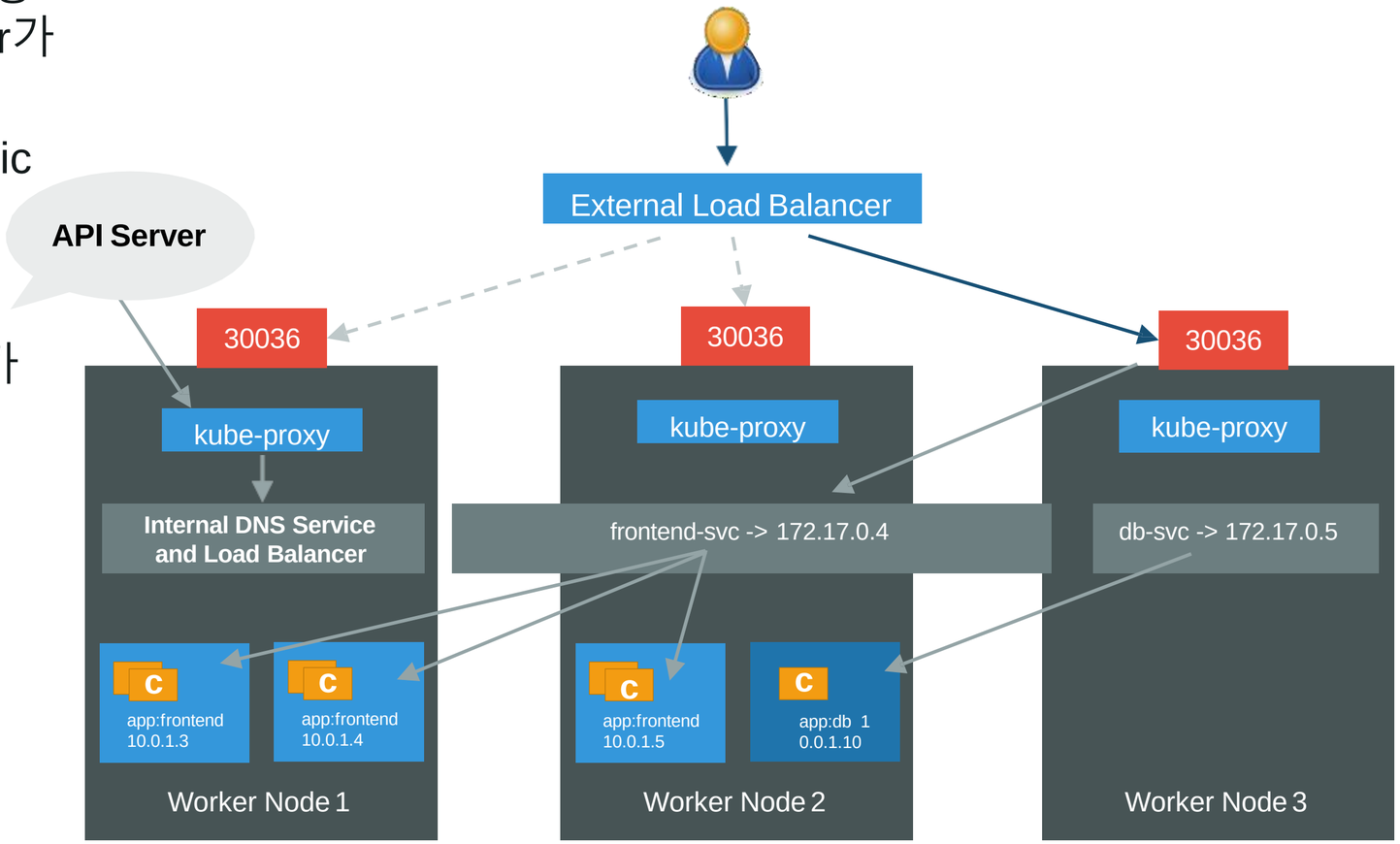
https://bcho.tistory.com/1262

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

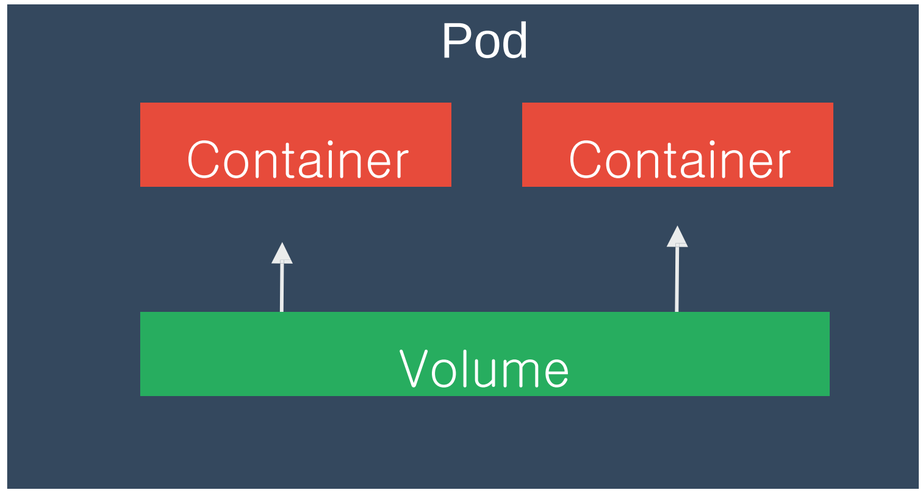
LoadBalancer ServiceType은 기본 인프라가 Load balancer의 자동 생성을 제공하고, Kubernetes를 지원 할 경우에만 작동

Ex) Azure, Google Cloud Platform, AWS



Volumes

- 쿠버네티스는 여러 호스트에 걸쳐 Stateless한 컨테이너를 마이크로서비스로 배포하는 것이 목표이기에 영속성 있는 저장장치(Persistent Volume)를 고려해야 함



- Volume은 Pod에 장착되어, 그 Pod에 있는 Container 간에 공유

Types of Volumes

- Pod에 마운트된 디스크는 Volume Type에 따라 사용 유형이 정의
- Volume Type 에 의해 디스크의 크기, 내용 등의 속성 설정

Types of Volumes

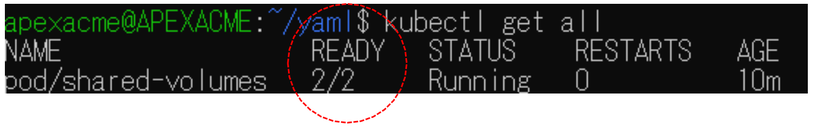
임시 볼륨	로컬 볼륨	네트워크 볼륨	네트워크 볼륨 (Cloud dependent)
emptyDir	hostPath	gitRepo, iSCSI, NFS cephFS, glusterFS ……	gcePersistentDisk, AzureDisk, Amazon EFS, Amazon EBS …
• Pod내 컨테이너간 공유	• Host 디렉토리를 Pod와 공유 해 사용하는 방식	• 영구적으로 영속성 있는 데이터 관리 목적	
• Pod가 삭제되면, emptyDir도 지워지므로 휘발성 데이터 저장 용도	• 컨테이너에 nodeSelecor를 지정 안하면 매번 다른 호스 트에 할당 • e.g. 호스트의 Metric 수집해 야 하는 경우	• 쿠버네티스는 PV와 PVC의 개념을 통해 Persistent 볼 륨을 Pod에 제공 • gitRepo, iSCSI, NFS와 같은 표준 네트워크 볼륨과 Cloud Vendor가 제공하는 볼륨으로 구분	

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



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

https://bcho.tistory.com/1259

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
```

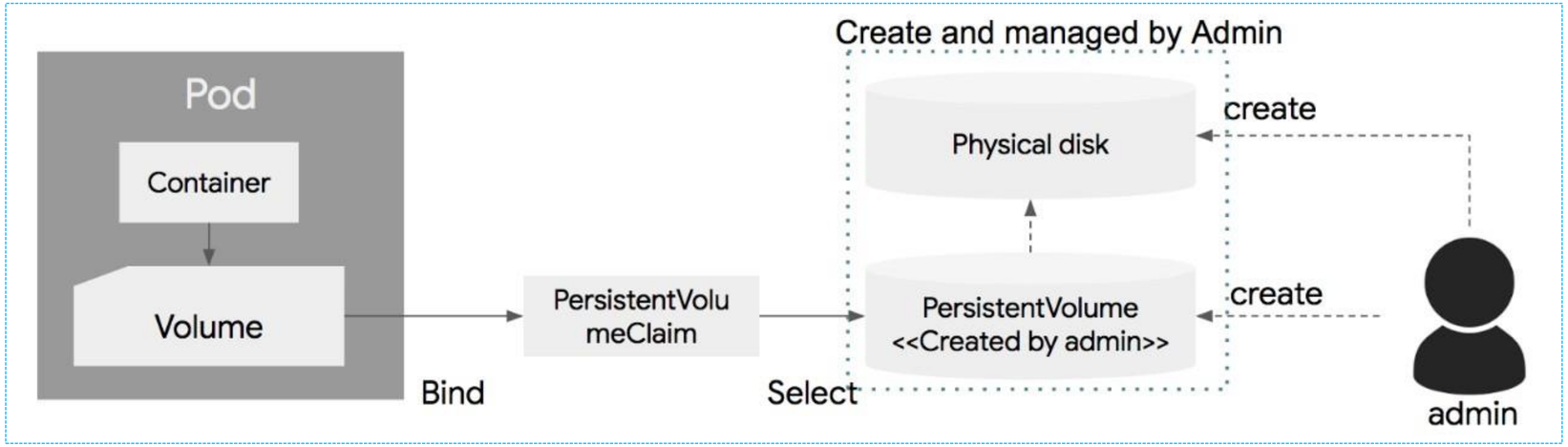
- Node의 Local 디스크 경로를 Pod에 마운트
- 같은 hostPath에 있는 볼륨은 여러 Pod사이에서 공유
- Pod가 삭제되어도 hostPath에 있는 파일은 유지
- Pod가 재기동 되어 다른 Node에서 기동될 경우,
  새로운 Node의 hostPath를 사용
- Node의 로그 파일을 읽는 로그 에이전트 컨테이너등에
  사용가능
- Pod 생성 및 확인 (Pod 내, ls -al /data/shared)

```
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
```

https://bcho.tistory.com/1259

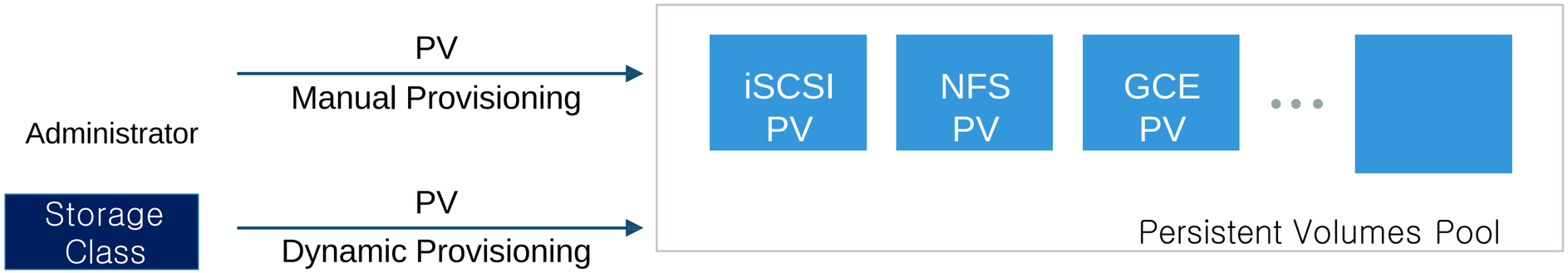
PersistentVolume & PersistentVolumeClaim

- 특정 IT 환경에서는 영속성 있는 대용량 스토리지는 관리자에 의해 관리
  - 쿠버네티스 클러스터를 사용하는 개발자로부터 볼륨 프로비저닝 역할을 분리하는 사상
- 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



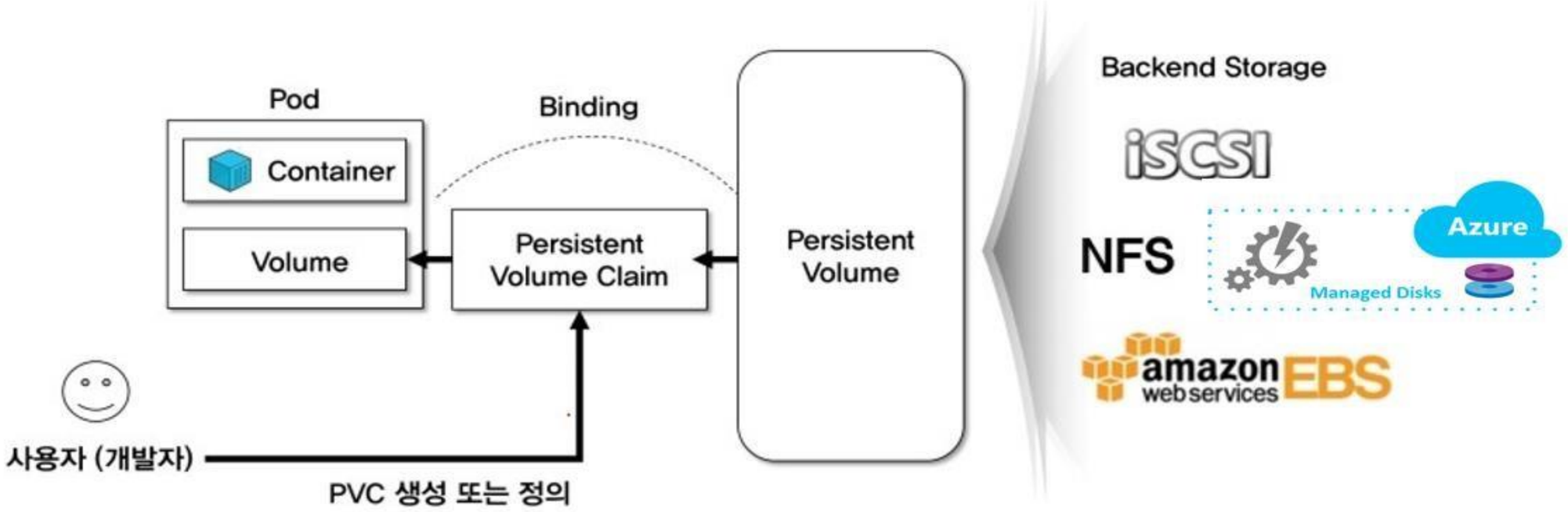
https://bcho.tistory.com/1259

StorageClass - Dynamic PV Provisioning



- PV는 관리자에 의해 수동으로 생성될 수 있지만, 자동 생성도 가능(Dynamic Provisioning)
- StorageClass(SC) Object
- StorageClass 객체에 의해 PersistentVolumes 동적 제공 가능
  - StorageClass는 PersistentVolume를 만들기 위해 Cloud Provider별 CSI 인터페이스를 구현하여 제공
- PersistentVolumes 스토리지 관리를 제공하는 Volume Types :
  - GCEPersistentDisk, AWSElasticBlockStore, AzureDisk, NFS, iSCSI

PersistentVolumeClaims



- Pod가 크기, 접근 모드에 따라 PVC를 요청, 적합한 PersistentVolume 발견시 PersistentVolume Claim에 바인딩
- PVC 조건을 만족하는 PV가 없을 경우, PV를 StorageClass가 자동으로 Provisioning하여 바인딩

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                                      default        6s
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$
```

https://bcho.tistory.com/1259

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/gcp"
      name: volume
  volumes:
    - name: volume
      persistentVolumeClaim:
        claimName: azure-managed-disk
```

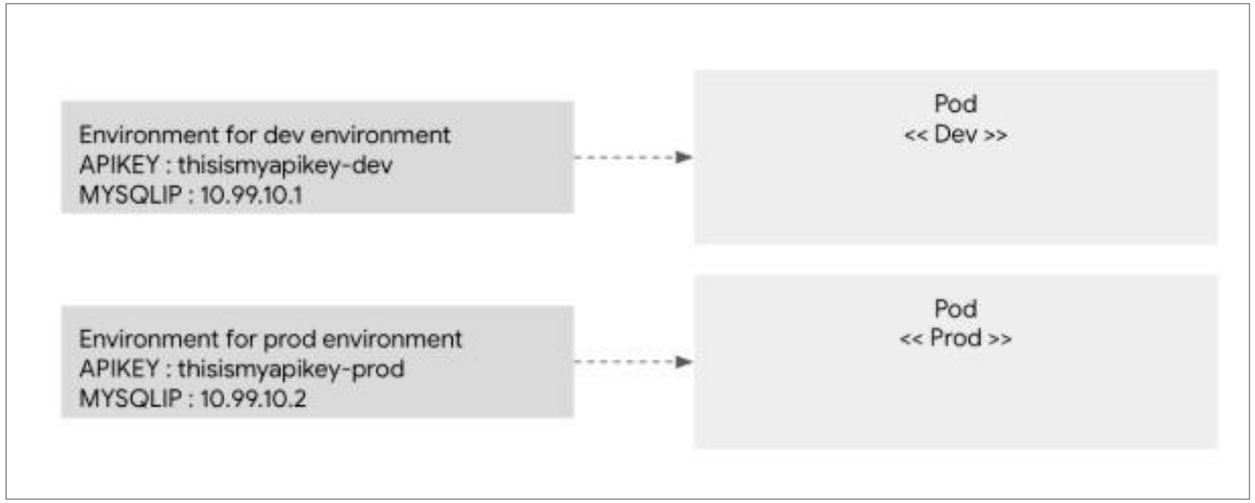
- kubectl apply -f pod-with-pvc.yaml
- kubectl get pod
- kubectl describe pod mypod
- kubectl exec -it mypod -- /bin/bash
- cd /mnt/gcp
- df -k 로 size 확인

```
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:/#
```

https://bcho.tistory.com/1259

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

- **ConfigMap**을 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  resour
  ceVersion: "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

- 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
```

Pod에서 configMap 사용하기

- 아래와 같은 comfigMap 을 만든다

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO
```

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

```
$ kubectl create -f configmaps.yaml
```

Pod 에서 configMap 사용하기

- 아래와 같은 Pod 을 만든다

```
apiVersion: v1
 kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: LOG_LEVEL
          valueFrom:
            configMapKeyRef:
              name: env-config
              key: log_level
```

- **dapi-test-pod.yaml**라는 이름으로 파일을 생성하였을 경우, 아래와 같이 Pod 를 생성한다.

```
$ kubectl create -f dapi-test-pod.yaml
```

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, password 라는 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 -n 'admin' | base64
  YWRtaW4=
$echo -n '1f2d1e2e67df' | base64
  MWYyZDFlMmU2N2Rm
```

- 위 방식으로 인코딩 된 정보를 사용해 secret 설정파일 생성

```
apiVersion: v1
 kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

- **base64** 인코딩은 바로 디코딩 됨으로 주의!

```
$ echo " MWYyZDFlMmU2N2Rm " | base64 --decode
```

설정파일을 절대 소스코드에 넣지 않도록 주의한다!

Pod에서 Secret 사용하기

```
apiVersion: v1
 kind: Pod
metadata:
  name: mypod
 spec:
   containers:
   - name: mypod
      image: redis
      volumeMounts:
       - name: foo
         mountPath: "/etc/foo"
         readOnly: true
   volumes:
   - name: foo
      secret:
      secretName: mysecret
```

- mypod.yaml 생성/ 실행
- kubectl create -f mypod.yaml

- $ kubectl get pod

Liveness Probes & Readiness Probes

- 쿠버네티스는 각 컨테이너의 상태를 주기적으로 체크(Health Check)해서,
  - 문제가 있는 컨테이너를 자동으로 재시작하거나 또는 문제가 있는 컨테이너를 서비스에서 제외 한다.

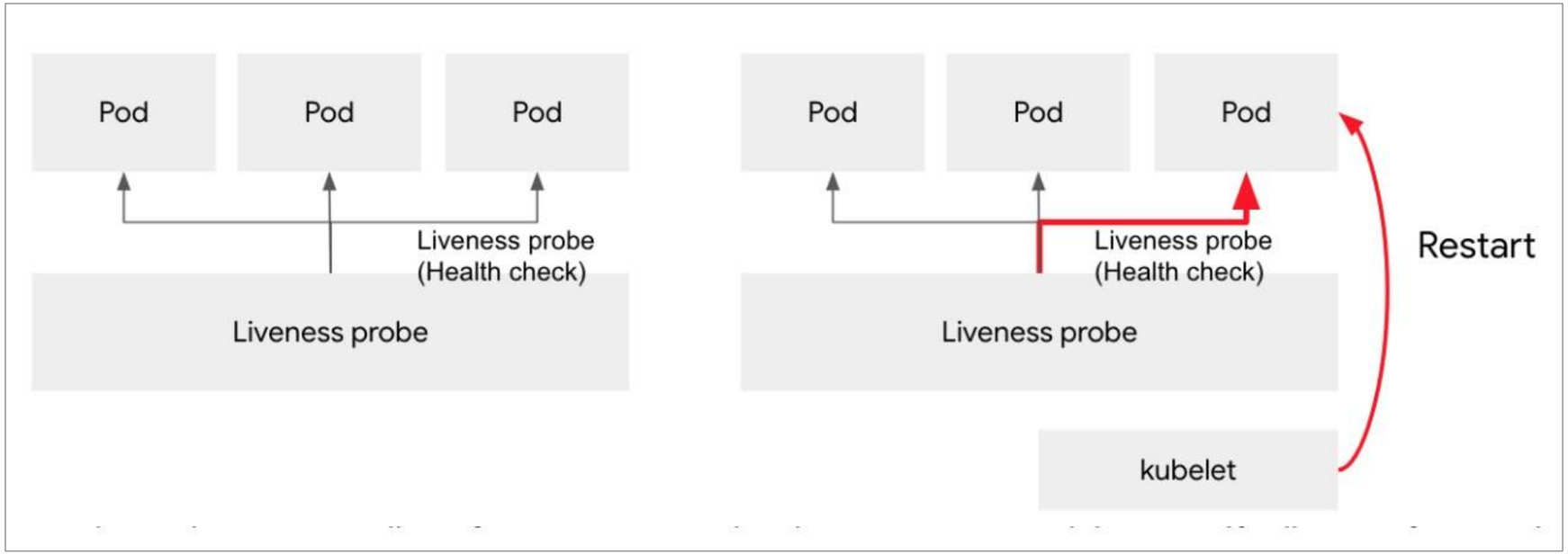
- Liveness와 Readiness Probes은 kubelet이 pod내에서 실행되는 어플리케이션의 health를 조정하기 때문에 매우 중요하다.

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:  comm
        and:
        - 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

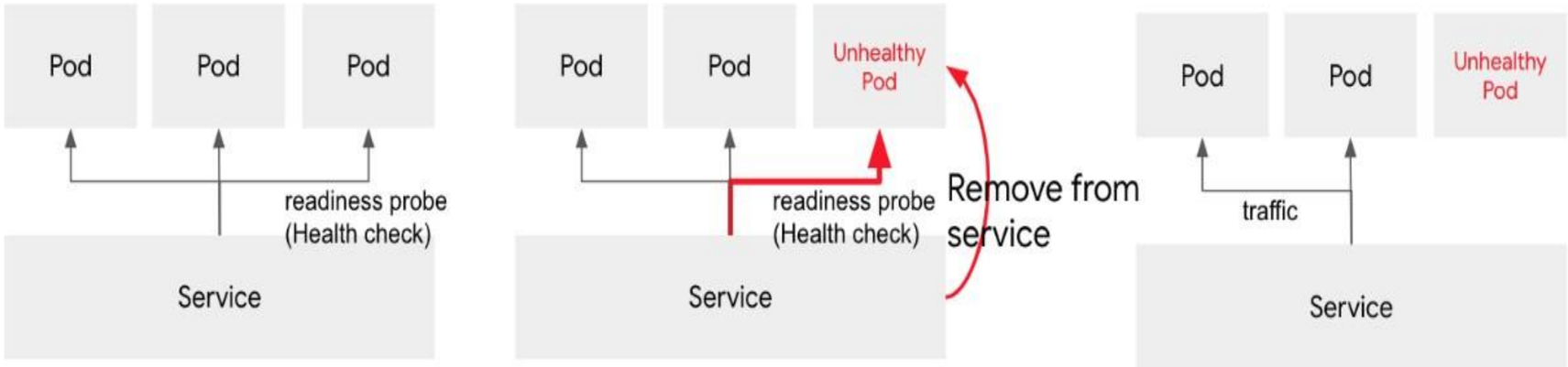
- kubelet은 TCP Liveness Probe를 통해, 지속적으로 어플리케이션이 실행중인 컨테이너의 TCP Socket을 열려고 한다.
- 정상이 아닌 경우 컨테이너를 재시작 한다.

```
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  initial
  DelaySeconds: 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, 서비스에서 제외시킨다.

  - 주기적으로 체크하여, 정상일 경우 정상 서비스에 포함

Liveness 와 readiness probe

- exec-liveness 설정 파일 생성
  - vim exec-liveness.yaml
- 파일 설정으로 배포
  - kubectl create –f exec-liveness.yaml
- 결과 확인
  - 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  initialD
      elaySeconds: 5
      periodSeconds: 5
```

Liveness 와 readiness probe

- http-liveness 설정파일을 생성
  - vim 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  htt
        pHeaders:
        - name: X-Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

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,

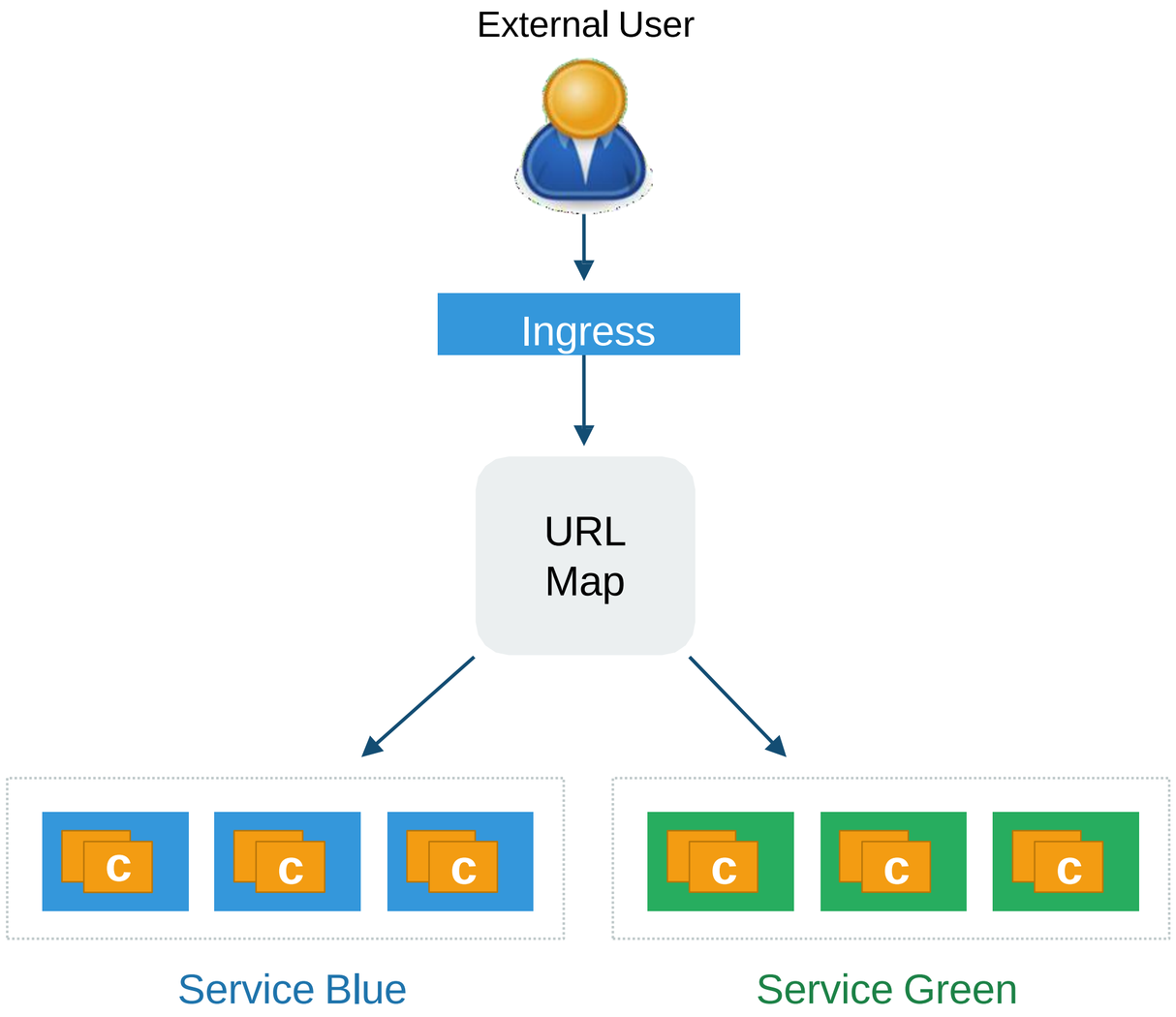
"An Ingress is a collection of rules that allow inbound connections to reach the cluster Services.

“"Ingress는인바운드 연결이 클러스터의 Service에게 라우팅되도록 하는 규칙의 집합체이다."

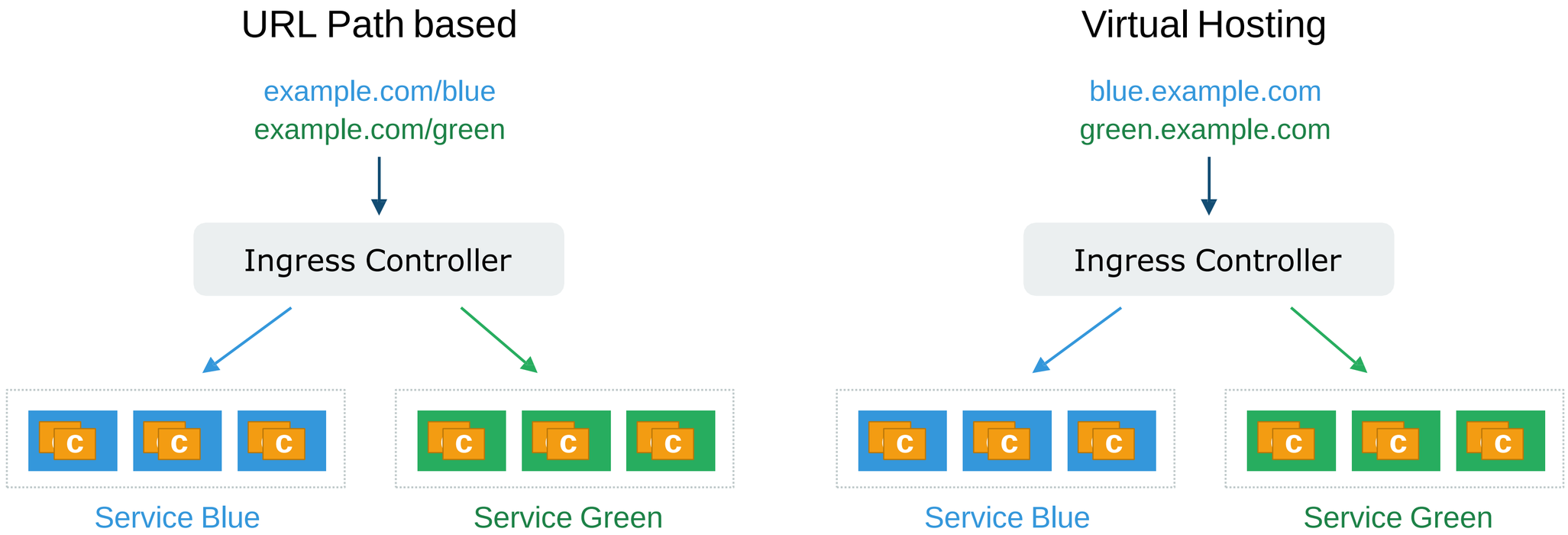
Ingress

- 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



- 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 개발 해 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-svc 로 요청이 포워딩된다.

```
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 라우팅된다.

,

```
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
```

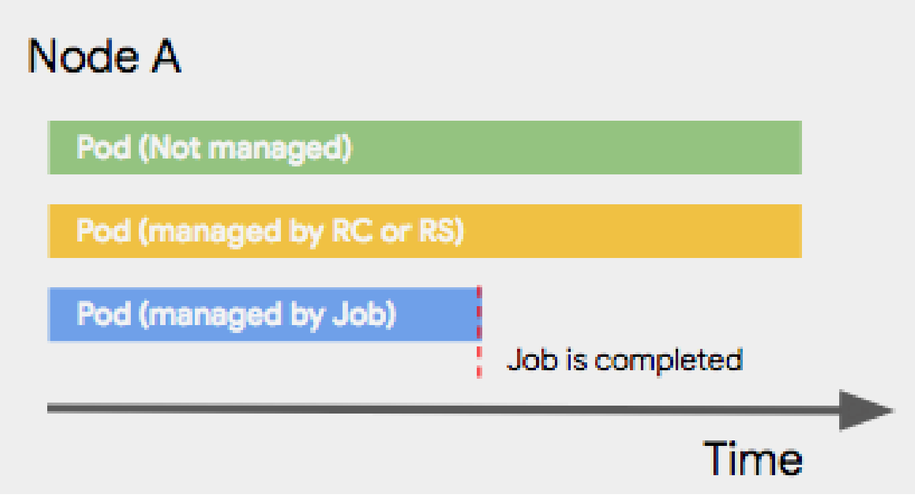
<Path-based routing>

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

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 job controller 의 의해서 동작된다.



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:  te
mplate:
   spec:  contai
     ners:
     - name: pi  i
       mage: 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

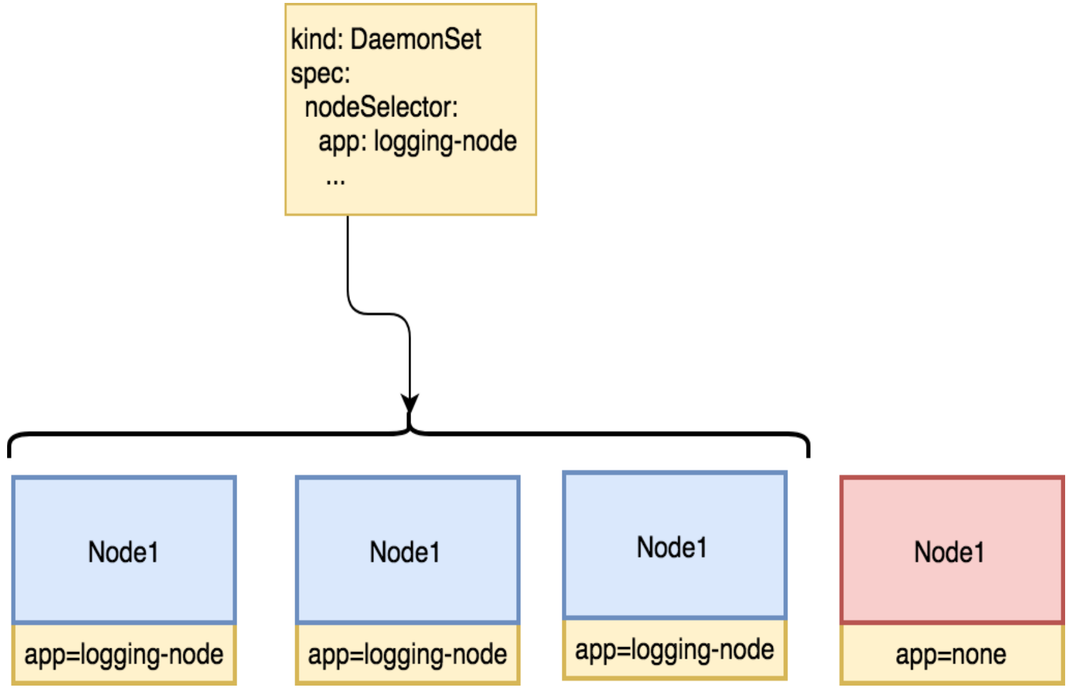
```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
 name: hello
spec:
 schedule: "*/1 * * * *"
 jobTemplate:
   spec:
     template:
       spec:
         containers:
         - name: hello  im
           age: busybox  a
           rgs:
           - /bin/sh
           - -c
         - date; echo Hello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

DaemonSets

- Pod를 각각의 노드에서 하나씩만 돌게 하는 형 태로 Pod를 관리하는 컨트롤러

- 모니터링 데이터를 수집하거나, 스토리지 데몬 을 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 Pod 들은 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) 등은 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: ngix
spec:
selector:
  matchLabels:
    app: nginx  service
Name: "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
```

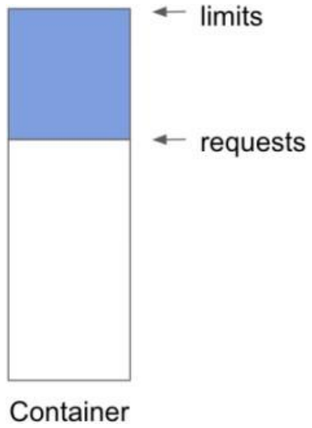
- $ kubectl get pvc

```
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

Resource Assign & Management

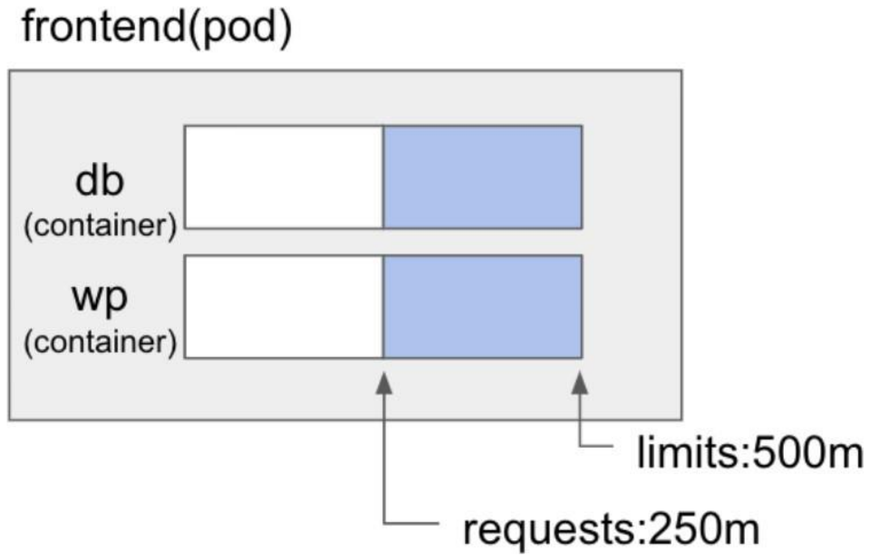
- 쿠버네티스에서 Pod를 어느 노드에 배포할지 결정하는 것을 스케줄링이라 함
- Pod에 대한 스케줄링시에, Pod내의 애플리케이션이 동작할 수 있는 자원(CPU,메모리 등) 정보 를 알아야 그만한 자원이 가용한 노드에 Pod 배포 가능
- 리소스 단위
  - CPU의 경우 ms(밀리 세컨드)를 사용하는데 대략 1000ms가 1 vCore (가상 CPU 코어)
  - 메모리의 경우 Mb를 사용하며 64M(64 x 1000), 또는 64Mi (64 x 1024)로 계산
  - Request & Limit
    - Request : 컨테이너가 생성될 때 요청하는 리소스 양
    - Limit : 리소스가 더 필요한 경우 추가로 사용가능한 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-lbd9g   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-cc8df6cfc-c7xcz                 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

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 수집할수 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 에이전 트로 작동을 하며 커스텀 설정이 가능