쿠버네티스 101

소개

과정

소개

이 교육은 다음과 같은 목표를 가지고 있음.

1. kubeadm기반으로 controller, comput에 대한 bootstrap 명령어 및 ClusterConfiguration, InitConfiguration에 대한 학습
2. 기본적인 쿠버네티스 명령어 활용
3. 쿠버네티스 기반으로 서비스 구성 및 배포를 위한 방법
4. 실제 프로덕트 환경에서 구축 시 고려할 부분

위와 같은 내용들은 교육에서 다룬다.

소개

01

다만 교육 진행 시, 다음과 같은 조건이 필요하다.

- Docker혹은 Podman과 같은 고수준 컨테이너 경험
- OCI 표준도구 Buildah, Podman, Skopeo와 같은 도구 사용 경험이 있는 사용자
- 리눅스 커널 /네트워크/스토리지 및 관리도구 사용 경험이 있는 사용자

02

위와 같은 경험이 있는 사용자에게 권장한다.

소개

쿠버네티스

쿠버네티스

쿠버네티스 본래 구글에서 내부 서비스 관리 용도로 사용함. 구글에서 Borg라는 이름으로 2014년도에 릴리즈 하였음. 구글에서는 대략 2006년도에서 내부적으로 서비스 및 컨테이너 관리 방법에 대해서 논의가 됨.

구글은 2014년도에 깃-허브에 처음 코드를 릴리즈 하였으며, 현재 기준 11년 동안 꾸준히 릴리즈가 됨. 현재 CNCF에서는 Serverless 및 Knative를 구현하기 위해서 쿠버네티스를 사용하고 있으며, 현재는 가상머신도 같이 컨테이너와 동일하게 POD기반으로 관리한다.

오픈스택은 쿠버네티스를 API기반으로 지원 및 자원 연동을 할 수 있도록 지원하고 있으며, 현재는 STARINGX를 OpenInfra를 통해서 지원하고 있다.

자원 구성

쿠버네티스가 사용하는 물리적 자원은 다음과 같다.

1. Controller(AKA Master)
2. Compute(AKA minion/worker)
3. Infra Node(Optional)

위의 두 개 용어, "master", "minion", "worker"는 오랫동안 IT계열에서 많이 사용하였지만, 차별적인 단어로 현재는 취급이 되고 있어서 통상적으로 "controller", "compute"와 같은 단어로 대체가 되어가고 있다.

자원 구성

쿠버네티스는 오래전 버전은 모든 서비스가 hosted형태로 구성이 되어 있었지만, 현재 쿠버네티스는 kubelet제외한 나머지 서비스는 containerized가 되었다. 이 말은 kubelet은 Proxy/Bootstrap/Monitor를 hosted상태에서 담당하고, 나머지는 서비스는 Static-POD가 되었다.

Static-POD는 우리가 알고 있는 POD와 동일하지만, 쿠버네티스에서 구성 시 기본적으로 생성하는 POD서비스 이다. Static-POD는 다음과 같다.

- kube-scheduler
- kube-apiserver
- kube-ControllerManager
- CoreDNS
- ETCD

릴리즈 형태

The Kubernetes project maintains release branches for the most recent three minor releases (1.32, 1.31, 1.30). Kubernetes 1.19 and newer receive approximately 1 year of patch support. Kubernetes 1.18 and older received approximately 9 months of patch support.

Kubernetes versions are expressed as **x.y.z**, where **x** is the major version, **y** is the minor version, and **z** is the patch version, following Semantic Versioning terminology.

자세한 내용은 "https://kubernetes.io/releases/"에서 확인이 가능하다.

쿠버네티스(OCP/SUSE Rancher)

엔터프라이즈에서 현재 많이 사용하는 플랫폼은 다음과 같다.

1. OpenShift Platform
2. SUSE Rancher

둘 다 기반은 쿠버네티스 바닐라 코드 기반으로 하고 있으며, 사용자가 좀 더 사용하기 쉽도록 관리 명령어 및 대시보드와 같은 편의 도구가 강화가 되어 있다. 또한, 커뮤니티에서 많이 사용하는 기능을 플랫폼과 통합하여, 클러스터 업그레이드 및 유지보수 시, 더 편하게 운영 및 운용이 가능하도록 패키지 구성이 되어 있다.

위의 두 개의 플랫폼을 학습하기 전에, 최소한으로 바닐라 버전의 쿠버네티스 기반으로 설치 및 운영을 권장하며, 이후 프로덕트 모델(Product Model)운영 시, 위의 두 개 제품에서 적절한 운영 모델을 선택한다. 두 가지 모델은 다음과 같은 큰 차이점을 가지고 릴리즈가 되고 있다.

쿠버네티스(OCP/SUSE Rancher)

이름	설명	프로덕트 레디
Vanila Kubernetes (AKA Kubernetes)	순수 쿠버네티스 버전. 기본적인 기능만 있으며, Federation API를 제공하기 때문에, 클러스터 연합 도구를 통해서 다중 클러스터 운영 기능	아니요
Redhat OpenShift	레드햇 오픈 시프트는 순수 쿠버네티스 기반으로 운영에 필요한 필수 기능을 통합 및 운영이 가능하도록 되어 있다. 제일 큰 차이점은 오퍼레이터를 통해서 모든 자원이 구성이 된다. 기존에 사용하던 쿠버네티스 표준 자원도 그대로 활용 및 적용이 가능하다. 다만, 다시 쿠버네티스로 마이그레이션은 불가능하다.	네
SuSE Rancher	수세 렌처는 바닐라 쿠버네티스 및 통합된 쿠버네티스 클러스터도 같이 제공한다. 최대 장점은 여러 클러스터를 다중 구성 및 운영이 손쉽고, 기존에 사용하던 쿠버네티스 클러스터를 그대로 Rancher에 통합하여 운영 및 사용이 가능하다.	네

하이퍼바이저

Windows Hyper-V

하이퍼바이저 설치

파워 쉘에서 관리자 권한으로 아래와 같이 실행한다. 실행 전, 해당 컴퓨터의 바이오스에서 CPU가상화 기능이 활성화 되어 있는지 확인이 필요하다.

```
DISM /Online /Enable-Feature /All /FeatureName:Microsoft-Hyper-V
```

네트워크 생성 및 구성

하이퍼바이저 설치는 다음과 같이 실행한다. 실행 후 리 부팅 그리고 네트워크 생성 및 구성한다.

```
New-VMSwitch -SwitchName "StaticNATSwitch" -SwitchType Internal
Get-NetAdapter
New-NetIPAddress -IPAddress 10.10.10.254 -PrefixLength 24 -InterfaceIndex <ifIndex>
Get-NetIPAddress
New-NetNat -Name StaticNATSwitch -InternalIPInterfaceAddressPrefix 10.10.10.0/24

New-VMSwitch -SwitchName "k8s-internal" -SwitchType Internal
Get-NetAdapter
New-NetIPAddress -IPAddress 192.168.0.254 -PrefixLength 16 -InterfaceIndex <ifIndex>
```

설치

Rocky Linux 9

kubeadm

간략

쿠버네티스 설치는 기본적으로 kubeadm명령어를 통해서 bootstrap를 수행한다. 이 방법에는 두 가지 방식을 지원한다.

1. With CLI Options
2. With Configuration File for Cluster/Kubelet/NodeJoin(Controller/Compute)

일반적인 설치 방식은 1번을 많이 사용한다. 1번 경우에는 설치 과정은 간단하지만, 클러스터에 자세한 옵션 설정 혹은 자동화 기반으로 배포는 어렵기 때문에 권장하지 않는다. 그래서, 이 과정에서는 둘 다 사용하는 방법을 학습하지만, 최종적으로는 2번 방식을 통해서 설치를 진행 및 완료한다.

필요한 ISO파일은 둘 중 하나만 있으면 된다.

- Rocky 9 Linux/Alma 9 Linux
- CentOS-9-Stream

간략

구성을 하기 위해서 두 가지 조건이 필요하다.

1. kubelet이 API를 받기 위한 네트워크
2. POD가 사용 및 네트워크 구성을 위한 VXLAN/Geneve용도 네트워크 인터페이스

실제로는 한 개의 네트워크만 있어도 상관이 없지만, 최소 조건으로 높은 학습을 하기 위해서 분리 구성을 권장한다.
실무적으로 구성하는 경우, 용도에 따라서 네트워크는 좀 더 많은 개수로 분리가 되기도 한다.

이름	설명
API	kubelet -> kube-apiserver서로 주고 받기 위한 네트워크
MANAGEMENT	관리 용도로 사용하는 네트워크 주로 kubectl명령어 사용 시 이 네트워크를 사용한다
STORAGE	NFS/GlusterFS/Ceph와 같은 스토리지를 접근 시 사용하는 네트워크
POD NETWORK	POD에서 사용하는 네트워크. VXLan/Geneve를 통해서 보통 구성이 된다
INFRA NETWORK	LoadBalancer/Ingress와 같은 서비스들이 사용하는 네트워크

NODE 1

bootstrap/controller node

저장소

```
cat <<EOF> /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/v1.28/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/v1.28/rpm/repodata/repomd.xml.key
exclude=kubelet kubeadm kubectl cri-tools kubernetes-cni
EOF
```

저장소

```
cat <<EOF> /etc/yum.repos.d/cri-o.repo
[cri-o]
name=CRI-O
baseurl=https://pkgs.k8s.io/addons:/cri-o:/stable:/v1.28/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/addons:/cri-o:/stable:/v1.28/rpm/repodata/repomd.xml.key
EOF
```

호스트 이름

```
hostnamectl set-hostname node1.example.com
```

설치/SWAP/SELinux

```
dnf install --disableexcludes=kubernetes kubectl kubeadm kubelet cri-o -y
setenforce 0
sed -i 's/enforcing/permissive/g' /etc/selinux/config

systemctl disable --now firewalld
sed -i 's/\/dev\/mapper\/rl-swap/\#\/dev\/mapper\/rl-swap/g' /etc/fstab
systemctl daemon-reload
swapoff -a
```

hostname

```
cat <<EOF>> /etc/hosts
192.168.10.10 node1.example.com node1 # controller
192.168.10.20 node2.example.com node2 # compute
192.168.10.30 node3.example.com node3 # compute
192.168.10.40 storage.example.com storage # NFS Server
EOF
```

Kernel Module/Low Level Runtime

```
systemctl enable --now crio kubelet
modprobe br_netfilter && modprobe overlay
cat <<EOF> /etc/modules-load.d/k8s-modules.conf
br_netfilter
overlay
EOF
cat <<EOF> /etc/sysctl.d/k8s-mod.conf
net.bridge.bridge-nf-call-iptables=1
net.ipv4.ip_forward=1
net.bridge.bridge-nf-call-ip6tables=1
EOF
sysctl --system -q
```

kubeadm #1

```
cat <<EOF> kubeadm-config.yaml
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
nodeRegistration:
  kubeletExtraArgs:
    cloud-provider: "external"
```

kubeadm #1

```
bootstrapTokens:
  - token: "abcdef.0123456789abcdef"
    ttl: "24h0m0s"
    descritpion: "bootstrap token"
    usages:
      - authentication
      - signing
    groups:
      - system:bootstrappers:kubeadm:default-node-token
```

kubeadm #2

```
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
kubernetesVersion: v1.28.0
networking:
  podSubnet: 192.168.0.0/16
apiServer:
  extraArgs:
    cloud-provider: "external"
controllerManager:
  extraArgs:
    cloud-provider: "external"
EOF
```

kubeadm init

```
kubeadm init --config=kubeadm-config.yaml
export KUBECONFIG=/etc/kubernetes/admin.conf
kubectl get nodes
kubectl get pods -Aw
> coredns
> tigera-operator
```

패키지 관리

HELM

Helm

바닐라 쿠버네티스는 기본적으로 패키징 및 패키징 배포에 대한 기능은 별도로 지원하지 않는다. 이와 비슷한 kustomized라는 기능이 있지만, 디렉터리 기반으로 자원 배포를 지원한다.

이러한 이유로, 버전 기반으로 관리가 어렵기 때문에, Helm기반으로 패키징 및 배포버전 관리한다. 쿠버네티스에서는 클러스터에 기능을 추가나 혹은 서비스를 배포하는 경우, 최소 HelmChart기반으로 구성 및 배포를 권장하고 있다.

여기서 HelmChart를 생성 및 배포하는 방법에 대해서는 학습하지 않는다. 또한, 클러스터 소프트웨어는 Chart기반으로 배포한다.

Helm

용어는 다음과 같다.

이름	설명
helm	패키지 관리자 이름 및 명령어
HelmChart	Helm에서 제공하는 패키지
value.yaml	패키지 설치 시, 오버라이드(override)가 필요한 경우, 이 파일을 통해서 설정 파일을 오버라이드 한다.

Helm

```
dnf install git tar -y

curl -fsSL -o get_helm.sh
https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3

mkdir ~/bin

mv /usr/local/bin/helm ~/bin

helm list
```

네트워크

CALICO

Calico

```
cat <<EOF> calico-quay-crd.yaml
---
apiVersion: operator.tigera.io/v1
kind: Installation
metadata:
  name: default
spec:
  calicoNetwork:
    bgp: Enabled
    hostPorts: Enabled
```

Calico

```
    ipPools:
      - blockSize: 26
        cidr: 10.0.0.0/16
        encapsulation: IPIPCrossSubnet
        natOutgoing: Enabled
        nodeSelector: all()
  registry: quay.io
EOFEOF
```

Calico

```
helm repo add projectcalico https://docs.tigera.io/calico/charts

kubectl create namespace tigera-operator

helm install calico projectcalico/tigera-operator --version v3.27.5 --
namespace tigera-operator

helm list --namespace tigera-operator

kubectl taint node node1.example.com node-role.kubernetes.io/control-
plane:NoSchedule-

kubectl apply -f calico-quay-crd.yaml

kubectl -n calico-system get pod -w
```

COMPUTE

node2

저장소

```
cat <<EOF> /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/v1.28/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/v1.28/rpm/repodata/repomd.xml.key
exclude=kubelet kubeadm kubectl cri-tools kubernetes-cni
EOF
```

저장소

```
cat <<EOF> /etc/yum.repos.d/cri-o.repo
[cri-o]
name=CRI-O
baseurl=https://pkgs.k8s.io/addons:/cri-o:/stable:/v1.28/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/addons:/cri-
o:/stable:/v1.28/rpm/repodata/repomd.xml.key
EOF
```

호스트 이름

```
hostnamectl set-hostname node2.example.com
```

설치/SWAP/SELinux

```
dnf install --disableexcludes=kubernetes kubectl kubeadm kubelet cri-o -y
setenforce 0
sed -i 's/enforcing/permissive/g' /etc/selinux/config

systemctl disable --now firewalld
sed -i 's/\/dev\/mapper\/rl-swap/\#\/dev\/mapper\/rl-swap/g' /etc/fstab
systemctl daemon-reload
swapoff -a
```

hostname

```
cat <<EOF>> /etc/hosts
192.168.10.10 node1.example.com node1 # controller
192.168.10.20 node2.example.com node2 # compute
192.168.10.30 node3.example.com node3 # compute
192.168.10.40 storage.example.com storage # NFS Server
EOF
```

Kernel Module/Low Level Runtime

```
systemctl enable --now crio kubelet
modprobe br_netfilter && modprobe overlay
cat <<EOF> /etc/modules-load.d/k8s-modules.conf
br_netfilter
overlay
EOF
cat <<EOF> /etc/sysctl.d/k8s-mod.conf
net.bridge.bridge-nf-call-iptables=1
net.ipv4.ip_forward=1
net.bridge.bridge-nf-call-ip6tables=1
EOF
sysctl --system -q
```

kubeadm JOIN YAML

```
cat <<EOF> kubeadm-join-config.yaml
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: JoinConfiguration
caCertPath: /etc/kubernetes/pki/ca.crt
discovery:
  bootstrapToken:
    apiServerEndpoint: 192.168.10.10:6443
    token: abcdef.0123456789abcdef
    unsafeSkipCAVerification: true
  timeout: 5m0s
  tlsBootstrapToken: abcdef.0123456789abcdef
EOF
```

kubeadm join

```
kubeadm join --config kubeadm-join-config.yaml

kubectl get pod -w -n kube-system

> calico-node-<ID>

> coredns
```

노드 초기화 제거

만약, 잘못 구성한 경우 아래와 같이 초기화 및 제거 작업을 수행 후, 다시 클러스터에 노드 추가한다.

```
kubectl delete node nodeX.example.com
kubeadm reset --force
```

COMPUTE

node3

저장소

```
cat <<EOF> /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/v1.28/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/v1.28/rpm/repodata/repomd.xml.key
exclude=kubelet kubeadm kubectl cri-tools kubernetes-cni
EOF
```

저장소

```
cat <<EOF> /etc/yum.repos.d/cri-o.repo
[cri-o]
name=CRI-O
baseurl=https://pkgs.k8s.io/addons:/cri-o:/stable:/v1.28/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/addons:/cri-
o:/stable:/v1.28/rpm/repodata/repomd.xml.key
EOF
```

호스트 이름

```
hostnamectl set-hostname node3.example.com
```

설치/SWAP/SELinux

```
dnf install --disableexcludes=kubernetes kubectl kubeadm kubelet cri-o -y
setenforce 0
sed -i 's/enforcing/permissive/g' /etc/selinux/config

systemctl disable --now firewalld
sed -i 's/\/dev\/mapper\/rl-swap/\#\/dev\/mapper\/rl-swap/g' /etc/fstab
systemctl daemon-reload
swapoff -a
```

hostname

```
cat <<EOF>> /etc/hosts
192.168.10.10 node1.example.com node1 # controller
192.168.10.20 node2.example.com node2 # compute
192.168.10.30 node3.example.com node3 # compute
192.168.10.40 storage.example.com storage # NFS Server
EOF
```

Kernel Module/Low Level Runtime

```
systemctl enable --now crio kubelet
modprobe br_netfilter && modprobe overlay
cat <<EOF> /etc/modules-load.d/k8s-modules.conf
br_netfilter
overlay
EOF
cat <<EOF> /etc/sysctl.d/k8s-mod.conf
net.bridge.bridge-nf-call-iptables=1
net.ipv4.ip_forward=1
net.bridge.bridge-nf-call-ip6tables=1
EOF
sysctl --system -q
```

kubeadm JOIN YAML

```
cat <<EOF> kubeadm-join-config.yaml
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: JoinConfiguration
caCertPath: /etc/kubernetes/pki/ca.crt
discovery:
  bootstrapToken:
    apiServerEndpoint: 192.168.10.10:6443
    token: abcdef.0123456789abcdef
    unsafeSkipCAVerification: true
  timeout: 5m0s
  tlsBootstrapToken: abcdef.0123456789abcdef
EOF
```

kubeadm join

```
kubeadm join --config kubeadm-join-config.yaml

kubectl get pod -n kube-system -w

> calico-node-<ID>

> coredns
```

High Level Runtime

PODMAN

설명

호스트 컴퓨터 사양에 따라서 다르지만, 메모리가 32GiB이상이 아닌 경우, node1번에 포드만을 설치한다. 만약, 공간이 여유가 있는 경우 별도의 가상머신을 구성 후 진행한다.

현재 쿠버네티스 클러스터 랩은 다음과 같은 구성으로 되어있다.

- node1(controller)
- node2(compute)
- node3(compute)
- node4(podman)

설치

```
dnf install podman -y

systemctl is-active podman

podman network list

podman container ls

podman pod ls

podman volume ls
```

컨테이너 기본 명령어

```
podman container run

podman container create

podman container rm

podman container inspect

podman container restart
```

POD 기본 명령어

```
podman pod create
podman pod
            --public
            --volume
podman pod rm
podman pod inspect
```

POD

배포판 및 회사마다 다르게 POD에서 사용하는 프로그램을 사용한다. 레드햇 계열 경우에는 쿠버네티스의 POD를 사용하지 않고, catatoin기반으로 POD를 구성한다. POD는 호스트 컴퓨터에서 사용하는 INIT(systemd)의 역할을 대행해주는 역할을 하기도 한다.

이러한 이유로 포드만에서 쿠버네티스와 호환성을 맞추기 위해서 Podman에서 사용하는 POD애플리케이션을 쿠버네티스 POD로 변경해야 한다.

변경하기 위해서 아래와 같이 작업을 수행한다.

POD 변경

변경을 위해서 다음과 containers.conf를 수정한다.

```
cp /usr/share/containers/containers.conf /etc/containers/
vi containers.conf
> infra_command = "/pause"
> infra_image = "registry.k8s.io/pause:3.9"
```

Podman export to K8S

포드만에서 패키징 및 검증이 끝난 컨테이너 이미지 및 서비스를 쿠버네티스에 전달이 가능하다. 쿠버네티스으로 서비스 전환하기 위해서 아래와 같은 명령어를 사용한다.

```
kubectl generate kube
kubectl generate kube <POD>/<CONTAINER> --type  --service --file <FILENAME>
```

쿠버네티스로 전환하려는 POD나 CONTAINER의 이름을 명시한 다음 어떠한 형태로 자원을 전달할지 결정해야 한다.

이름	설명
--type	deployment, pod. 기본 값은 POD로 되어 있으며, 이는 containers.conf에서 수정이 가능하다
--service	YAML파일에 서비스 자원도 같이 명시한다
--file	파일 생성 시 사용할 파일 이름

작업

강사와 함께 진행.

INFRA

DOCKER-REGISTRY

GOGS

설명

빌드 머신(Podman)에서 구성한 이미지를 저장 및 클러스터에 배포하기 위해서 깃 서버와 소스코드 관리가 필요하다. 직접 hosted형태로 구성이 가능하지만, 편하게 관리 및 사용하기 위해서 컨테이너 기반으로 구성 및 관리한다.

사용하는 소프트웨어는 다음과 같다.

1. Gogs(for GIT)
2. Docker-Registry(for image repository)

포드만 기반으로 위 두개 서비스를 구성한다.

INFRA POD

```
podman volume create git-data

podman volume create registry-data

podman volume

podman pod create --name dev-infra --publish 5000:5000 --publish 80:3000 --volume registry-
data:/var/lib/registry --volume git-data:/data

podman container run -d --pod dev-infra --name registry docker.io/library/registry:2.8.3

podman container run -d --pod dev-infra --name git docker.io/gogs/gogs

podman pod ls

> dev-infra

podman container ls

> git

> registry
```

BIND

도메인 기반으로 확인하기 위해서 아래와 같이 구성한다. BIND9은 컨테이너 기반으로 구성하지 않고, HOSTED기반으로 구성한다.

```
dnf install bind-chroot -y
cat <<EOF>> /etc/named.rfc1912.zones
zone "demo.io" IN {
 type master;
  file "demo.io.zone";
  allow-update { none; };
};
EOF
```

BIND

존 파일을 아래와 같이 생성한다.

```
cat <<EOF>> /var/named/demo.io.zone
\$TTL 7200
demo.io. IN SOA workstation.example.com. admin.demo.io. (
    2024061802 ; Serial
    7200 ; Refresh
    3600 ; Retry
    604800 ; Expire
    7200) ; NegativeCacheTTL

                IN NS storage.example.com.
demo.io.        IN A 192.168.10.10
nginx           IN A 192.168.10.240
dev-git         IN A 192.168.10.10
dev-registry    IN A 192.168.10.10
demo            IN CNAME demo.io.
EOF
```

BIND

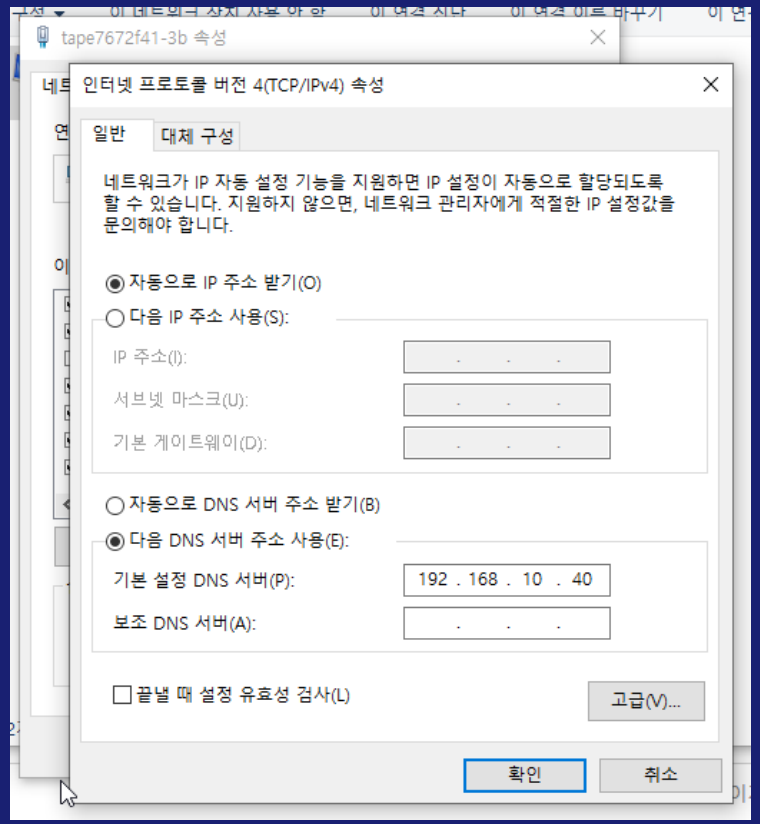
최종적으로 조회가 가능하도록 아래와 같이 수정한다.

```
dnf install nano -y
nano /etc/named.conf
> listen-on port 53 { any; };
> allow-query { any; };

systemctl enable --now named
systemctl status named

connectionName="System ens3"
nmcli con mod "$connectionName" ipv4.ignore-auto-dns yes
nmcli con mod "$connectionName" ipv4.dns 192.168.10.40
nmcli con up "$connectionName"
```

node1~4번까지 아래와 같이 DNS등록

BIND

윈도우 클라이언트는 다음과 같이 DNS정보를 수정한다.



실험코드

SPRING(JSP/TOMCAT/PGSQL)

PREPARE

```
dnf install git -y
git clone https://github.com/tangt64/codelab
dnf install maven-openjdk8 -y
cd /root/codelab/java/blog/
mvn clean package
podman build -f Dockerfile.web -t 192.168.10.10:5000/blog/web:v1
podman build -f Dockerfile.db -t 192.168.10.10:5000/blog/db:v1
```

기본 이미지는 docker.io에서 받는다.

BUILD AND RUN

```
podman volume create pgdata

podman pod create --name blog --publish 8080:8080 --volume
pgdata:/var/lib/postgresql/data

podman container run -d --pod blog --name web
192.168.10.10:5000/blog/web:v1

podman container run -d --pod blog --name db 192.168.10.10:5000/blog/db:v1

podman container ls

podman pod ls
```

DEBUG

강사와 함께…

REBUILD

```
podman container rm web

podman container run -d --pod blog --name web
192.168.10.10:5000/blog/web:v2

podman images

> blog/db:v2
```

설치

STORAGE(NFS)

설명

OS

```
hostnamectl set-hostname storage.example.com
dnf install nfs-utils -y
systemctl enable --now nfs-server
mkdir -p /nfs/
```

OS

```
cat <<EOF>> /etc/hosts
192.168.10.10 node1.example.com node1 # controller
192.168.10.20 node2.example.com node2 # compute
192.168.10.30 node3.example.com node3 # compute
192.168.10.40 storage.example.com storage # storage
EOF
cat <<EOF> /etc/exports.d/kubernetes.exports
/nfs *(rw,sync,no_root_squash,insecure,no_subtree_check,nohide)
EOF
```

OS

```
systemctl enable --now nfs-server

exportfs -avrs

showmount -e storage.example.com

systemctl disable --now firewalld
```

NFS-CSI-DRIVER

```
export KUBECONFIG=/etc/kubernetes/admin.conf

helm repo add csi-driver-nfs https://raw.githubusercontent.com/kubernetes-csi/csi-driver-nfs/master/charts

helm install csi-driver-nfs csi-driver-nfs/csi-driver-nfs --namespace kube-system --version v4.9.0

helm list --namespace kube-system

kubectl --namespace=kube-system get pods --selector="app.kubernetes.io/instance=csi-driver-nfs" --watch
```

이 부분은 컨트롤러에서 작업을 수행

NFS STORAGE CLASS

```
cat <<EOF> storageclass-configure.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: default
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: nfs.csi.k8s.io
parameters:
  server: storage.example.com
  share: /nfs
reclaimPolicy: Delete
volumeBindingMode: Immediate
mountOptions:
  - hard
  - nfsvers=4.1
EOF
```

NFS PVC(FOR PGSQL)

```
cat <<EOF> blog-pod-pvc.yaml
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pgdata
  labels:
    app: pgdata
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 2Gi
EOF
```

VERIFY SC/PV/PVC

```
kubectl apply -f storageclass-configure.yaml
kubectl apply -f blog-pod-pvc.yaml
kubectl -n default get sc,pv,pvc
```

문법

YAML

EDITOR

YAML

자원은 YAML코드로 표현하면 양식은 다음과 같은 규칙을 따른다.

1. 최소 2칸으로 들여쓰기 구성
2. 여러 자원을 사용하는 경우 YAML의 시작 및 끝 부분은 표시(---,…)
3. 패치(patch)으로 적용하는 경우 JSON형태로 적용

예제 코드

코드는 다음과 같이 작성한다.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

패치(patch)

패치는 ETCD에 직접 변경하기 때문에 JSON형태로 수정해야 한다. 코드는 아래와 같다.

```
kubectl patch svc frontend -p '{"spec":{"externalIPs":["192.168.10.40"]}}'
```

기본 명령어

kubectl

NAMESPACE

오픈 시프트에서는 프로젝트, 쿠버네티스는 네임스페이스라고 부른다. 자원을 격리를 하기 위해서 사용하는 개념.
다음과 같은 명령어로 생성이 가능.

```
kubectl create namespace testnamespace
```

YAML파일로 생성하기 위해서 다음과 같이 실행한다.

```
kubectl create namespace testnamespace --dry-run=client --output=yaml > testnamespace.yaml
```

NAMESPACE

네임 스페이스가 올바르게 제거가 되지 않는 경우, 아래와 같이 제거가 가능하다.

```
REMOVENS=<NAMESPACE_NAME>
kubectl get namespace "$REMOVENS" -o json \
  | tr -d "\n" | sed "s/\"finalizers\": \[[^]]\+\]/\"finalizers\": []/" \
  | kubectl replace --raw /api/v1/namespaces/"$REMOVENS"/finalize -f -
```

POD

POD의 뜻은 "고래 때"를 가지고 있으며, 여러 컨테이너를 POD로 하나로 묶어서 서비스를 구성한다. POD 개념을 구성하기 위해서, /pause라는 애플리케이션을 사용한다. 이 애플리케이션 각 회사마다 다르게 사용하며, 레드햇 경우에는 cataonit라는 pause애플리케이션을 사용한다.

/pause애플리케이션은 호스트 시스템의 /bin/init 혹은 /bin/systemd와 동일한 기능을 하며, 에플리케이션에 요청하는 시스템 콜 및 라이브러리 콜을 커널 네임스페이스(Kernel Namespace)를 통해서 구성한다.

POD에서 사용하는 자원은 커널에서 C-Group를 통해서 CPU/MEM/DISK/NET와 같은 자원을 제어한다. 현재 쿠버네티스는 CPU/MEM만 지원하며, 나머지 자원 영역에 대해서는 지원하지 않는다.

RUN

자원 실행하는 옵션은 다음과 같이 지원한다.

옵션	설명	비교
--image nginx pod-nginx	간단하게 POD실행한다. 보통 맨 뒤쪽은 POD의 이름으로 사용한다.	간단하게 POD YAML생성 시 많이 사용한다.
--dry-run=client --output=yaml pod-nginx	실제로 생성하지 않고, kubectl명령어에서 YAML형태로 자원을 생성 후 화면에 출력한다.	
--stdin(-i) --tty	컨테이너를 실행하면서 가상 TTY를 통해서 대화형으로 시작한다.	
--port	POD에서 사용할 포트번호를 명시한다.	
--expose	POD를 SERVICE로 노출한다. 기본적으로 ClusterIP형태로 되며, 이를 사용하기 위해서 앞서 이야기한 --port가 선언이 되어 있어야 한다.	
--rm	Podman의 --rm옵션과 똑같다. 컨테이너 혹은 POD가 중지가 되면 제거가 된다.	

RUN

명령어는 다음과 같이 사용한다.

```
kubectl run --image=node1.example.com/testapp/php-blue-green:v1 php-pod
kubectl get pod
kubectl delete pod php-pod
kubecrl get pod
kubectl run --image=node1.example.com/testapp/php-blue-green:v1 --dry-run=client --output=yaml php-pod-v1 > php-pod-v1.yaml
ls -l php-pod.yaml
```

APPLY

명령어는 다음과 같이 사용한다.

이름	설명	공통사항
create	자원을 YAML/CLI형태로 구성 시 사용한다. 이 명령어는 생성한 자원에 대해서 기록을 남기지 않으며, 사용자가 명시한 내용대로 클러스터에 생성한다. CLI부분은 사용자가 명령어로 자원 생성 혹은 YAML파일 생성 시 사용한다. 다만, 모든 자원을 명령어로 생성이 가능하지 않다.	이 명령어들은 kustomize기능을 제공한다. 앞서 이야기한 부분처럼, 차이점은 기록을 남기느냐 남기지 않느냐 차이가 있다.
apply	YAML 및 DIRECTORY형태로 구성 및 생성한다. apply와 create의 제일 큰 차이점은 apply는 생성된 자원의 리비전 기록을 가지고 있으며, 디렉터리 기반으로 생성이 가능하다.	

EXAMPLE YAML

명령어는 다음과 같이 사용한다.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: php-pod-v1
  name: php-pod=v1
spec:
  containers:
  - image: node1.example.com/testapp/php-blue-green:v1
    name: php-pod
    resources: {}
```

APLLY

명령어는 다음과 같이 사용한다.

```
# kubectl apply -f php-pod-v1.yaml
# kubectl get pod
> php-pod
```

CREATE

명령어는 다음과 같이 사용한다.

```
# kubectl create -f php-pod-v1.yaml
# kubectl get pod
> php-pod
```

확장 명령어

kubectl

프로젝트 준비

BLOG

2025-01-08

LOADBALANCER

METALLB

MetalLB

CNCF L/B

설치

```
helm repo add metallb https://metallb.github.io/metallb

kubectl create namespace network-metallb

helm install metallb metallb/metallb --namespace network-metallb
```

설정

```
cat <<EOF> ippool.yaml
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: first-pool
  namespace: network-metallb
spec:
  addresses:
  - 192.168.10.240-192.168.10.40
  autoAssign: true
EOF
```

설정

```
cat <<EOF> l2.yaml
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: lb-pool
  namespace: network-metallb
spec:
  ipAddressPools:
  - first-pool
  nodeSelectors:
  - matchLabels:
      kubernetes.io/hostname: node1.example.com
EOF
```

설정

```
kubectl apply -f ippool.yaml
kubectl apply -f l2.yaml
kubectl get -f ippool.yaml
kubectl get -f l2.yaml
```

INGRESS

NGINX

설치

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx

kubectl create namespace network-ingress-nginx

helm install ingress-nginx ingress-nginx/ingress-nginx --version 4.11.0  --namespace network-ingress-nginx --create-namespace

helm list --namespace network-ingress-nginx

kubectl create namespace blog

kubectl get pods -n network-ingress-nginx
```

Configure ingress point

```
cat <<EOF> ingress-blog.demo.io.yaml
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-nginx-blog
  namespace: blog
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
```

Configure ingress point

```
spec:
  ingressClassName: nginx
  rules:
    - host: blog.demo.io
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: blog
                port:
                  number: 8080
EOF
```

APPLY TO THE Ingress point

```
kubectl apply -f ingress-blog.demo.io.yaml

kubectl get -f ingress-blog.demo.io.yaml
```

INFRA IN KUBERNETES

GOGS

REGISTRY

설명

기존 포드만에 구성이 된 인프라 서비스를 쿠버네티스 내부로 가지고 오기 위해서 아래와 같이 구성이 가능함. 다만, 구성하기 위해서 다음과 같은 서비스가 구성이 되어 있어야 됨.

1. LoadBalancer(External/CNCF 둘 다 상관이 없음)
2. Proxy Server(HAProxy, Ingress, OpenELB)

클러스터 내부로 인프라를 가지고 오기 위해서 아래와 같이 YAML작성 후, 쿠버네티스 클러스터에 적용.

kube-git

GIT-SERVER(namespace)

```
mkdir kube-git
cd kube-git
cat <<EOF> 01-kube-git-namespace.yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: kube-git
  labels:
    istio-injection: enabled
EOF
```

GIT-SERVER(pvc)

```
cat <<EOF> 02-kube-git-pvc.yaml
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: kube-git-data
  namespace: kube-git
  labels:
    app: kube-git
```

GIT-SERVER(pvc)

```
spec:
  storageClassName: "default"
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
EOF
```

GIT-SERVER(deployment)

```
cat <<EOF> 03-kube-git-deploy.yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kube-git
  namespace: kube-git
```

GIT-SERVER(deployment)

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kube-git
  template:
    metadata:
      labels:
        app: kube-git
        istio-injection: enabled
        version: v1
```

GIT-SERVER(deployment)

```
spec:
  containers:
    - name: gogs
      image: 192.168.10.10:5000/library/gogs:0.13
      ports:
        - containerPort: 22
          name: ssh
        - containerPort: 3000
          name: http
      env:
        - name: SOCAT_LINK
          value: "false"
      volumeMounts:
        - name: kube-git-persistent-storage
          mountPath: /data
```

GIT-SERVER(deployment)

```
      volumes:
        - name: kube-git-persistent-storage
          persistentVolumeClaim:
            claimName: kube-git-data
      dnsPolicy: "None"
      dnsConfig:
        nameservers:
          - 192.168.10.40
          - 8.8.8.8
EOF
```

GIT-SERVER(svc)

```
cat <<EOF> 04-kube-git-svc.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: kube-git
  namespace: kube-git
spec:
  selector:
    app: kube-git
```

GIT-SERVER(svc)

```
  ports:
    - name: ssh
      protocol: TCP
      port: 10022
      targetPort: 22
    - name: http
      protocol: TCP
      port: 18080
      targetPort: 3000
EOF
```

GIT-SERVER(ingress)

```
cat <<EOF> 05-kube-git-ingress.yaml
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: kube-git-ingress
  namespace: kube-git
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
```

GIT-SERVER(ingress)

```
spec:
  ingressClassName: nginx
  rules:
    - host: git.demo.io
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: kube-git
                port:
                  number: 18080
EOF
```

GIT-SERVER 적용

적용하기 위해서 아래와 같이 명령어를 실행한다.

```
kubectl apply -f kube-git/
kubectl get -f kube-git/
```

kube-registry

REGISTRY

레지스트리 서버는 서비스 배포 시 사용하는 이미지를 보관하는 용도다. 네임스페이스에서 사용하는 이미지는 가급적이면 각 네임스페이스별로 저장을 권장한다.

REGISTRY(namespace)

```
mkdir kube-registry
cd kube-registry
cat <<EOF> 01-kube-registry-namespace.yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: kube-registry
EOF
```

REGISTRY(deployment)

```
cat <<EOF> 02-kube-registry-deployment
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: kube-registry
  name: kube-registry
  namespace: kube-registry
```

REGISTRY(deployment)

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kube-registry
  template:
    metadata:
      labels:
        app: kube-registry
```

REGISTRY(deployment)

```
    spec:
      containers:
      - image: 192.168.10.10:5000/library/registry
        name: registry
        ports:
        - containerPort: 5000
EOF
```

REGISTRY(svc)

```
cat <<EOF> 03-kube-registry-svc.yaml
---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: kube-registry
  name: kube-registry
  namespace: kube-registry
```

REGISTRY(svc)

```
spec:
  ports:
  - port: 5000
    protocol: TCP
    nodePort: 30500
  selector:
    app: kube-registry
  type: NodePort
EOF
```

REGISTRY(svc)

```
cat <<EOF> 04-kube-registry-svc.yaml
---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: kube-registry
  name: kube-registry
  namespace: kube-registry
```

REGISTRY(svc)

```
spec:
  ports:
  - port: 5000
    protocol: TCP
    nodePort: 30500
  selector:
    app: kube-registry
  type: NodePort
EOF
```

REGISTRY(ingress)

```
cat <<EOF> 05-kube-registry-ingress.yaml
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: registry-ingress
  namespace: kube-registry
```

REGISTRY(ingress)

```
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    kubernetes.io/ingressClassName: "nginx"
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/proxy-body-size: "0"
    nginx.ingress.kubernetes.io/proxy-read-timeout: "600"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "600"
    nginx.ingress.kubernetes.io/enable-cors: "true"
    nginx.ingress.kubernetes.io/cors-allow-methods: "PUT, GET, POST, OPTIONS, DELETE"
    nginx.ingress.kubernetes.io/cors-allow-origin: "*"
    nginx.ingress.kubernetes.io/cors-allow-headers: "DNT,X-CustomHeader,X-LANG,Keep-Alive,User-Agent,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,X-Api-Key,X-Device-Id,Access-Control-Allow-Origin"
```

REGISTRY(ingress)

```
spec:
  ingressClassName: nginx
  rules:
  - host: registry.demo.io
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: kube-registry
            port:
              number: 5000
EOF
```

REGISTRY-DASHBOARD(deploy)

```
cat <<EOF> 06-kube-registrydashboard-ingress.yaml
---
kind: Deployment
apiVersion: apps/v1
metadata:
  namespace: kube-registry
  name: kube-registry-dashboard
  labels:
    app: kube-registry-dashboard
```

REGISTRY-DASHBOARD(deploy)

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kube-registry-dashboard
  template:
    metadata:
      labels:
        app: kube-registry-dashboard
```

REGISTRY-DASHBOARD(deploy)

```
    spec:
      containers:
        - name: kube-registry-dashboard
          image: joxit/docker-registry-ui:2.0
          ports:
            - name: kube-registry-dashboard
              containerPort: 80
```

REGISTRY

```
          env:
            - name: REGISTRY_URL
              value: http://registry.demo.io
            - name: SINGLE_REGISTRY
              value: "true"
            - name: REGISTRY_TITLE
              value: registry
            - name: DELETE_IMAGES
              value: "true"
      dnsPolicy: "None"
      dnsConfig:
        nameservers:
          - 192.168.10.40
EOF
```

REGISTRY-DASHBOARD(svc)

```
cat <<EOF> 07-kube-registrydashboard-svc.yaml
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: kube-registry-dashboard
  name: kube-registry-dashboard
  namespace: kube-registry
```

REGISTRY-DASHBOARD(svc)

```
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: kube-registry-dashboard
status:
  loadBalancer: {}
EOF
```

REGISTRY-DASHBOARD(ingress)

```
cat <<EOF> 08-kube-registrydashboard-ingress.yaml
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: registry-dashboard-ingress
  namespace: kube-registry
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    kubernetes.io/ingressClassName: "nginx"
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
```

REGISTRY-DASHBOARD(ingress)

```
spec:
  ingressClassName: nginx
  rules:
  - host: registry-dashboard.demo.io
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: kube-registry-dashboard
            port:
              number: 80
EOF
```

REGISTRY 적용

적용하기 위해서 아래와 같이 명령어를 실행한다.

```
cd ..
kubectl apply -f kube-registry/
kubectl get -f kube-registry/
```

CI/CD

TEKTON

GIT SERVER

IMAGE REGISTRY

TEKTON

CNCF

설치

```
kubectl apply -f https://storage.googleapis.com/tekton-releases/pipeline/previous/v0.53.7/release.yaml

kubectl get pods -n tekton-pipelines

dnf install wget -y

wget https://github.com/tektoncd/cli/releases/download/v0.32.0/tkn_0.32.0_Linux_x86_64.tar.gz

mkdir ~/bin/

tar xf tkn_0.32.0_Linux_x86_64.tar.gz -C ~/bin/
```

설치

```
tkn hub install task buildah --namespace blog
tkn hub install task kubernetes-actions --namespace blog
tkn hub install task git-clone --namespace blog
tkn hub install task maven --namespace blog
tkn task list --namespace blog
kubectl get pod -n tekton-pipelines
```

rbac

CLUSTER-ROLE

```
mkdir tekton-rbac
cd tekton-rbac
cat <<EOF> 01-tekton-clusterrole.yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: '*'
  name: tkn-action
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
EOF
```

CLUSTER ROLE BINDING

```
cat <<EOF> 02-tekton-clusterrolebind.yaml
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: rolebind-for-deploy
subjects:
  - kind: ServiceAccount
    name: tkn-action
    namespace: blog-v2
roleRef:
  kind: ClusterRole
  name: tkn-action
  apiGroup: rbac.authorization.k8s.io
EOF
```

TEKTON SERVICE ACCOUNT

```
cat <<EOF> 03-tekton-sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tkn-action
  namespace: blog-v2
EOF
```

적용

```
kubectl apply -f tekton-rbac/

kubectl get -f tekton-rbac/
```

DB PIPELINE

PIPELINE(db)

```
mkdir blog-release
cd blog-release
cat <<EOF> 01-tkn-blog-db-release.yaml
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: tkn-build-db
spec:
  workspaces:
    - name: source
      description: |
        This workspace will be shared throughout all steps.
```

PIPELINE(db)

```
params:
  - name: image-repo
    type: string
    description: |
      Docker image name
    default: http://192.168.0.10:5000
```

PIPELINE(db)

```
tasks:
  - name: clone-repository
    params:
      - name: url
        value: http://192.168.0.10:3000/gogs/app.git
      - name: revision
        value: "master"
      - name: deleteExisting
        value: "true"
    taskRef:
      kind: Task
      name: git-clone
    workspaces:
      - name: output
        workspace: source
```

PIPELINE(db)

```
    - name: build-image
      runAfter:
        - clone-repository
      params:
        - name: IMAGE
          value: "$(params.image-repo)/app/db"
        - name: TLSVERIFY
          value: false
        - name: DOCKERFILE
          value: Containerfile.db
```

PIPELINE(db)

```
    taskRef:
      kind: Task
      name: buildah
    workspaces:
      - name: source
        workspace: source
    runAfter:
      - clone-repository
```

PIPELINE(db)

```
    - name: deploy-service
      taskRef:
        name: kubernetes-actions
      params:
        - name: script
          value: |
            kubectl apply -n blog-v2 -f http://dev-registry.demo.io:3000/gogs/demo/blog-db.yaml
            kubectl get deployment
      runAfter:
        - build-image
EOF
```

PIPERUN(db)

```
cat <<EOF> 01-run-blog-db-release.yaml
---
apiVersion: tekton.dev/v1
kind: PipelineRun
metadata:
  generateName: build-tkn-db-
spec:
  pipelineRef:
    name: tkn-build-db
  taskRunTemplate:
    podTemplate:
      hostNetwork: true
```

PIPERUN(db)

```
  workspaces:
    - name: source
      persistentVolumeClaim:
        claimName: blog
  params:
    - name: REG_URL
      value: dev-registry.demo.io:5000/app/db:v1
    - name: VERSION
      value: v1
    - name: FILENAME
      value: Containerfile.db
EOF
```

TOMCAT PIPELINE

tkn-blog-tomcat-release

```
cd blog-release
cat <<EOF> 03-tkn-blog-web-release.yaml
---
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: tkn-build-tomcat
```

tkn-blog-tomcat-release

```
spec:
  workspaces:
    - name: source
      description: |
        This workspace will be shared throughout all steps.
  params:
    - name: image-repo
      type: string
      description: |
        Docker image name
      default: http://dev-registry.demo.io:5000
```

tkn-blog-tomcat-release

```
tasks:
  - name: clone-repository
    params:
      - name: url
        value: http://dev-git.demo.io:3000/gogs/app.git
      - name: revision
        value: "master"
      - name: deleteExisting
        value: "true"
    taskRef:
      kind: Task
      name: git-clone
    workspaces:
      - name: output
        workspace: source
```

tkn-blog-tomcat-release

```
  - name: build-image
    runAfter:
      - clone-repository
    params:
      - name: IMAGE
        value: "$(params.image-repo)/blog/web:v1"
      - name: TLSVERIFY
        value: false
      - name: DOCKERFILE
        value: Containerfile.web
```

tkn-blog-tomcat-release

```
    taskRef:
      kind: Task
      name: buildah
    workspaces:
      - name: source
        workspace: source
    runAfter:
      - clone-repository
```

tkn-blog-tomcat-release

```
- name: deploy-service
      taskRef:
        name: kubernetes-actions
      params:
        - name: script
          value: |
            kubectl apply -f http://dev-registry.demo.io:3000/gogs/demo/blog-web.yaml
            kubectl get deployment
      runAfter:
        - build-image
EOF
```

run-tkn-blog-web-release

```
cat <<EOF> 04-run-tkn-blog-web-release.yaml
---
apiVersion: tekton.dev/v1
kind: PipelineRun
metadata:
  generateName: build-tkn-web
spec:
  pipelineRef:
    name: tkn-build-tomcat
  taskRunTemplate:
    podTemplate:
      hostNetwork: true
```

run-tkn-blog-web-release

```
 workspaces:
    - name: source
      persistentVolumeClaim:
        claimName: blog
  params:
    - name: REG_URL
      value: dev-registry.demo.io:5000/blog/db:v1
    - name: VERSION
      value: v1
    - name: FILENAME
      value: Containerfile.db
EOF
```

파이프라인 생성 및 실행

```
kubectl apply -f blog-release/

kubectl create -f blog-release/

tkn pipeline list

tkn pipelinerun list
```

JENKINS

CNCF

KNATIVE

kn

VIRTUALIZATION

KUBE-VIRT

모니터링

Prometheus

Grafana

Promethous

설치

설치

```
kubectl create namespace kube-monitoring

helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm install prometheus prometheus-community/prometheus --namespace monitoring
```

Grafana

설치

설치

```
helm repo add grafana https://grafana.github.io/helm-charts

helm install grafana grafana/grafana --namespace monitoring

kubectl get secret --namespace monitoring grafana -o
jsonpath="{.data.admin-password}" | base64 --decode ; echo

kubectl expose service grafana --namespace monitoring --type=NodePort --
target-port=3000 --name=grafana-ext
```