

쿠버네티스 101

2025-03-16

PREPARE

1

목차

쿠버네티스 101

목차

- 쿠버네티스 소개
- 설치준비
- 쿠버네티스 도구
- 쿠버네티스 아키텍처
- 시작하기 전에
- 기본기능
- 프로젝트 애플리케이션
- 쿠버네티스 인프라 소프트웨어
- 쿠버네티스 스토리지

목차

- Scale/rollout/rollback/history
- 컨테이너 자동확장(autoscale)
- 선호도 및 예외처리
- 변수전달
- drain/taint/cordon/uncordon
- 노드 추가 및 제거
- 컨테이너 및 포드 상태 확인
- Label, annotation, selectors
- 데몬 서비스
- 상태 POD 서비스
- 작업

목차

- 설정파일(configmap)
- 비밀(secret)
- 배포 및 배치
- 복제자
- 외부아이피 주소
- 외부이름
- 네트워크 정책
- 쿠버네티스 기능 확장
- DevOps를 위한 준비
- 관리 및 운영

과정개요

쿠버네티스 101

과정개요

현재 CNCF 및 컨테이너 기반 플랫폼의 표준으로 자리 잡은 쿠버네티스(Kubernetes)에 대해 학습하는 과정. 본 교육에서는 쿠버네티스 설치, 구성, 운영을 다루며, OCI 도구와 쿠버네티스의 관계도 함께 살펴본다.

과정은 3일/5일에 따라서 구성이 달라진다. 내용은 다음과 같이 차이가 있다.

길이	설명
3일	설치 및 기본적인 쿠버네티스 명령어 학습
4일	3일 과정에 확장 기능
5일	3/4일 과정에+DevOPS

소개

강사

과정

2025-03-16

PREPARE

8

강사

강사

2025-03-16

PREPARE

9

강사

이름: 최국현

메일: tang@linux.com, 회신은 다른 메일 주소로 드리고 있습니다. :)

사이트: tang.dustbox.kr

언제든지 질문 및 요청 환영 입니다.

소개

쿠버네티스

2025-03-16

PREPARE

11

소개

이 교육은 다음과 같은 목표를 가지고 있음.

1. kubeadm기반으로 controller, compute에 대한 bootstrap 명령어 및 **ClusterConfiguration**, **InitConfiguration**에 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.

쿠버네티스

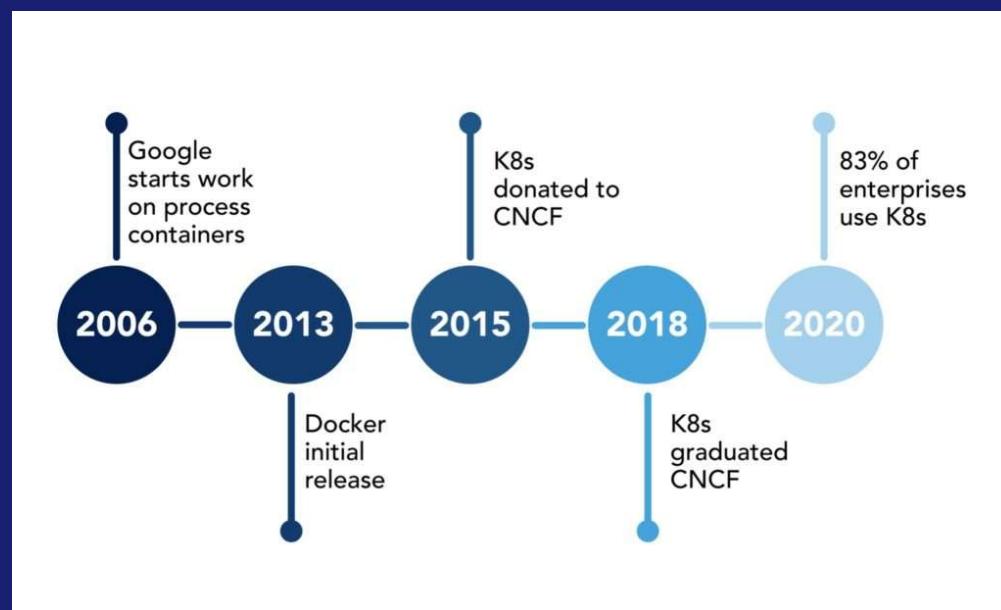
쿠버네티스 본래 구글에서 내부 서비스 관리 용도로 사용함. 구글에서 Borg라는 이름으로 2014년도에 릴리즈 하였음. 구글에서는 대략 2006년도에서 내부적으로 서비스 및 컨테이너 관리 방법에 대해서 논의가 됨.

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.

쿠버네티스

쿠버네티스는 구글에서 Borg라는 이름으로, 내부적으로 클러스터 관리 및 배포 용도로 사용하였다. 초기 Borg는 C++로 작성이 되었으나, 쿠버네티스로 넘어 오면서 Go언어 기반으로 작성이 되었다. 초기 릴리즈는 2015년도, 발표는 2014년도에 이루어 졌다.



가상화/컨테이너

쿠버네티스는 자체적으로 런타임을 지원하지 않는다. 여기서 말하는 런타임은(runtime) 컨테이너 이미지에 있는 내용을 메모리에 불러와서 프로세서를 격리 및 추적하여, 호스트와 분리 운영이 가능하도록 환경을 구성한다.

이러한 이유 때문에 가상머신과 많은 혼동이 있는데, 둘은 엄연히 큰 차이가 있다. 가상머신은 링 구조(ring structure)를 생성 및 재구성하기 때문에, 더 복잡하고 독립적인 자원을 가지고 동작한다. 가상머신은 런타임에서 동작하는 구조가 아닌, 일반적으로 하이퍼바이저(hypervisor)나 혹은 가속기(kernel module level)에서 동작한다.

컨테이너는 이와 반대로 가상화 같은 하드웨어 수준의 기술이 필요하지 않으며, 커널 수준에서 격리 및 추적 기술 기반으로 프로세스를 관리한다.

이러한 이유로, 가상머신에서 사용하던 기능을 전부 컨테이너로 마이그레이션이 가능하지 않기 때문에, 서비스 아키텍트 및 구성에 따라서 플랫폼을 선택 및 구성하면 된다.

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 kubelet제외 한 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 POD서비스이다. 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.

기업용 쿠버네티스

쿠버네티스

2025-03-16

쿠버네티스

20

쿠버네티스(OCP/SUSE Rancher)

엔터프라이즈 쿠버네티스 플랫폼에서 많이 사용하는 제품은 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 가능	아니요
Redhat OpenShift(Okd)	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 랜처는 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

2025-03-16

제품비교

많이 사용하는 쿠버네티스 제품에 대해서 비교하면 보통 다음과 같이 평가가 된다.

플랫폼	설명
RKE2 (Rancher Kubernetes Engine 2)	쿠버네티스 기반으로 구성한 보안 중심 컨테이너 오케스트레이션 도구
OpenShift(Okd)	쿠버네티스 기반으로 구성한 레드햇 컨테이너 오케스트레이션 도구
Kubernetes (Upstream K8s)	표준 오픈소스 컨테이너 오케스트레이션 도구

제품비교

기능	RKE2	OpenShift(okd)	Kubernetes
설치 편의성	쉬움(스크립트 기반)	어려움 (앤서블/테라폼 혼용)	중간 (kubeadm/YAML)
보안	기본적으로 CIS 하드닝 적용	SELinux, SCC(보안 컨텍스트) 적용	운영체제에 따라서 다름
네트워크	CNI 플러그인 선택 가능(Flannel, Calico 등)	OpenShift SDN (OVN, Multus)	표준 CNI플러그인 전부 지원
서비스 메쉬	Istio, Linkerd 사용 가능	기본 Istio 내장 (Service Mesh Operator)	표준 CNCF사용 가능
스토리지	CSI 지원		
운영 관리	Rancher UI와 통합	OpenShift 콘솔 제공 (기본 UI, GitOps 지원)	kubectl 및 대시보드 (추가 설치)
업데이트 관리	Rancher UI를 통한 자동화	레드햇 가이드 기반으로 업데이트	수동 업데이트 필요
멀티클러스터	Rancher 기반 멀티클러스터 지원	OpenShift Cluster Manager 제공	Kubernetes Federation V1/V2 지원

제품비교

항목	RKE2	OpenShift(Okd)	Kubernetes
오버헤드	낮음 (경량화)	높음	보통
컨테이너 실행 속도	빠름 (Containerd 사용)	빠름 (CRI-O 사용)	빠름 (Containerd, CRI)
네트워크 성능	CNI에 따라 다름	OpenShift SDN, 느림	CNI에 따라 다름
노드 확장성	고성능	엔터프라이즈 환경에서 강력	확장 가능하지만 구성 필요
업데이트 성능	빠른 롤링 업데이트	안정적이지만 복잡함	노드 단위 업데이트

제품비교

사용 목적	추천 플랫폼
보안 중심 Kubernetes 운영	RKE2 (기본 CIS 하드닝)
엔터프라이즈 환경	RKE2/OpenShift
순수 Kubernetes 운영 (유연성, 확장성 우선)	Kubernetes (Upstream)
멀티클러스터 및 하이브리드 클라우드	RKE2 / OpenShift
빠른 배포 및 경량 운영	RKE2
강력한 CI/CD 및 개발자 환경 제공	RKE2/OpenShift

랩 준비

Windows Hyper-V

OpenStack vLab

2025-03-16

PREPARE

27

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 리부팅 그리고 네트워크 생성 및 구성한다.

```
> New-VMSwitch -SwitchName "StaticNATSwitch" -SwitchType Internal  
> Get-NetAdapter  
> New-NetIPAddress -IPAddress 10.10.0.254 -PrefixLength 16 -InterfaceIndex <ifIndex>  
> Get-NetIPAddress  
> New-NetNat -Name StaticNATSwitch -InternalIPInterfaceAddressPrefix 10.10.0.0/16  
> New-VMSwitch -SwitchName "k8s-internal" -SwitchType Internal  
> Get-NetAdapter  
> New-NetIPAddress -IPAddress 192.168.0.254 -PrefixLength 16 -InterfaceIndex  
<ifIndex>
```

오픈스택

오픈스택을 사용하는 경우 강사의 가이드에 따라서 진행한다.

설치

Rocky Linux 9

kubeadm

HELM

2025-03-16

PREPARE

31

설명

쿠버네티스 설치는 기본적으로 kubeadm 명령어를 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

설명

구성을 하기 위해서 두 가지 조건이 필요하다.

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

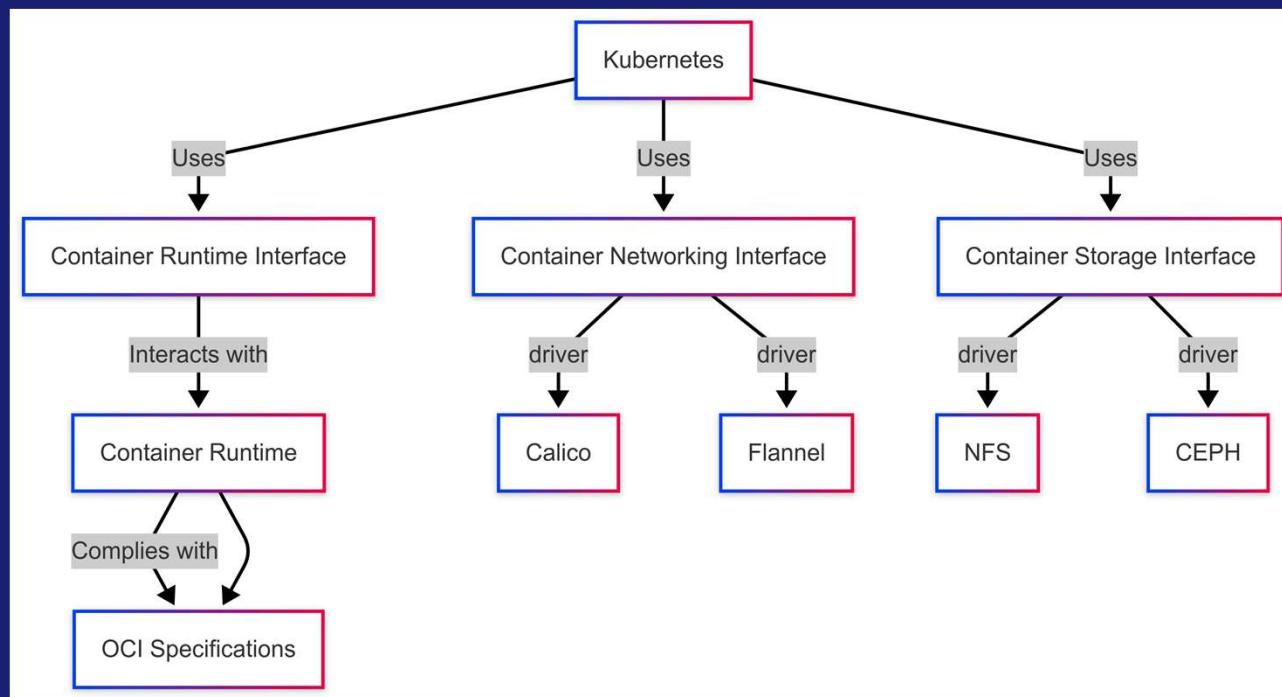
실제로는 한 개의 네트워크만 있어도 상관이 없지만, 최소 조건으로 높은 학습을 하기 위해서 분리 구성을 권장한다. 실무적으로 구성하는 경우, 용도에 따라서 네트워크는 좀 더 많은 개수로 분리가 되기도 한다.

이름	설명
API	kubelet -> kube-apiserver 서로 주고 받기 위한 네트워크
MANAGEMENT	관리 용도로 사용하는 네트워크 주로 kubectl 명령어 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

설명

아래부터는 수동 설치 부분이니, 수동 설치는 아래 명령어를 따른다.

설명



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/repo/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/repo/repodata/repomd.xml.key
EOF
```

호스트 이름

```
nodeX# 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
```

호스ㅌ 이름

```
# 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
```

커널 모듈 및 런타임

```
# 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
```

컨트롤러 설정

```
# vi kubeadm-config.yaml
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
nodeRegistration:
  kubeletExtraArgs:
    cloud-provider: "external"
```

컨트롤러 설정

bootstrapTokens:

- token: "abcdef.0123456789abcdef"

- ttl: "24h0m0s"

- description: "bootstrap token"

usages:

- authentication

- signing

groups:

- system:bootstrappers:kubeadm:default-node-token

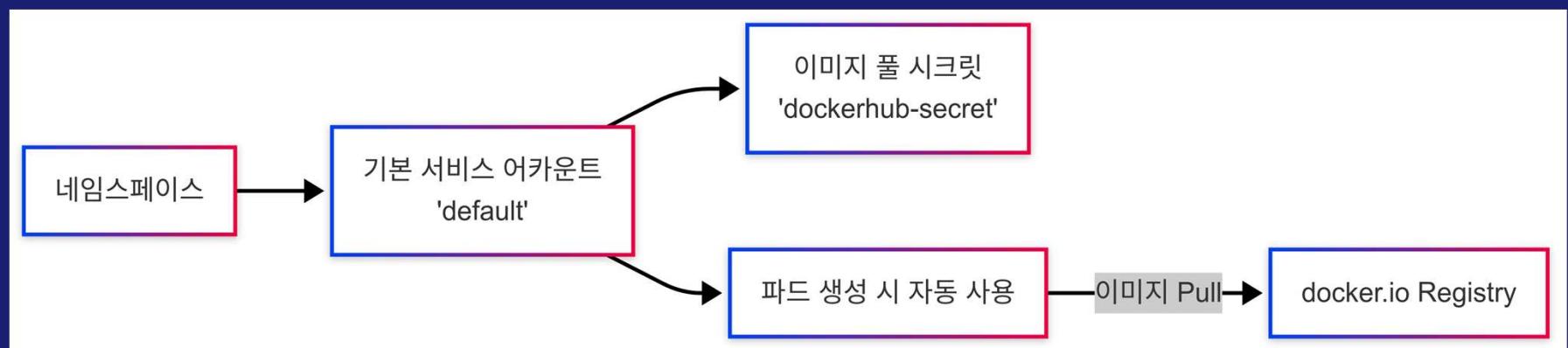
컨트롤러 설정

```
---  
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"
```

부트스트랩 컨트롤러 생성

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

DOCKER-REGISTRY



2025-03-16

DOCKER-REGISTRY

hub.docker.io를 통해서 이미지를 받는 경우, 로그인 하지 않는 경우 다운로드 횟수에 제한이 걸린다. 이러한 이유로 네임스페이스에 저장소 정보를 등록한다.

```
# kubectl create secret docker-registry hub-docker-io \
  --docker-server=docker.io \
  --docker-username=사용자명 \
  --docker-password='test!@$$@#' \
  --docker-email=0|메일
# kubectl get secret
hub-docker-io
```

2025-03-16

DOCKER-REGISTRY

올바르게 등록이 되었는지 확인한다.

```
# kubectl describe secret/hub-docker-io
Name:          hub-docker-io
Namespace:     exam-labels
Labels:        <none>
Annotations:   <none>

Type:  kubernetes.io/dockerconfigjson

Data
=====
.dockerconfigjson:  139 bytes
```

DOCKER-REGISTRY

DEPLOYMENT에는 다음과 같이 호출한다. 아래는 사용 예시이다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
```

DOCKER-REGISTRY

DEPLOYMENT에는 다음과 같이 호출한다

```
template:  
  metadata:  
    labels:  
      app: my-app  
spec:  
  containers:  
    - name: my-container  
      image: docker.io/사용자명/이미지이름:태그  
  imagePullSecrets:  
    - name: hub-docker-io
```

DOCKER-REGISTRY

POD에는 다음과 같이 호출한다

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: docker.io/library/nginx:latest
  imagePullSecrets:
    - name: hub-docker-io
```

2025-03-16

DOCKER-REGISTRY

혹은 서비스계정으로 가능하다.

```
# kubectl patch serviceaccount default \
-p '{"imagePullSecrets": [{"name": "hub-docker-io"}]}' \
-n <NAMESPACE>
```

강사설명



2025-03-16

PREPARE

54

패키지 관리

HELM

ChartMuseum

2025-03-16

PREPARE

55

Helm

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

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

여기서 **HelmChart**를 생성 및 배포하는 방법에 대해서는 학습하지 않는다. 또한, 클러스터 소프트웨어는 Chart 기반으로 배포한다.

HelmChart

Helm에서 제공하는 패키지를 Chart라고 부른다. 일반적으로 GIT기반으로 배포 및 관리하며, 패키징은 .tgz으로 구성이 된다. 사용자가 오프라인 설치를 원하는 경우 .tgz파일을 다운로드 받아서 수동으로 설치 및 구성이 가능하다.

현재는 오프라인 설치를 원하는 경우 Chart Museum기반으로 권장하고 있다.

Chrat MUSEUM

내부망에 구성된 쿠버네티스는 내부에서 배포 및 접근이 필요한 HelmChart서버가 필요하다. 쿠버네티스에서는 HelmChart Museum이름으로 해당 기능을 제공하고 있다. Museum를 사용하기 위해서 두 가지 기능이 필요하다.

1. 내부에서 사용하는 컨테이너 레지스트리 서버
2. 내부에서 사용하는 오브젝트 스토리지 서버

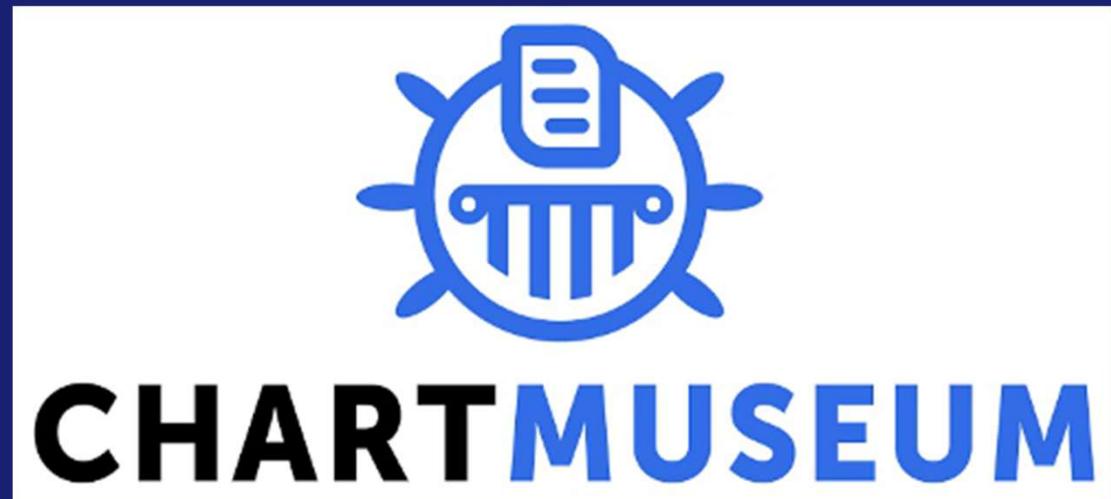
컨테이너 레지스트리는 컨테이너 이미지를 저장 및 배포가 가능한 웹 서버 혹은 docker-registry와 호환되는 레지스트리 서비스이면 가능.

오브젝트 스토리지는 퍼블릭 클라우드 서비스 혹은 프라이빗 클라우드에서 많이 사용하는 오브젝트 스토리지. 예를 들어서 OpenStack Object Storage API와 호환이 되면 사용이 가능하다.

이 과정에서는 부록 disconnected network에서 다룬다.

HelmChart MUSEUM

[차트 뮤지엄 사이트](#)



Helm

용어는 다음과 같다.

이름	설명
helm	패키지 관리자 이름 및 명령어
HelmChart	Helm에서 제공하는 패키지
value.yaml	패키지 설치 시, 오버라이드 override가 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  
  
# sh get-helm-3  
  
# mv /usr/local/bin/helm ~/bin  
  
# helm list
```

네트워크

CALICO

FLANNEL

2025-03-16

PREPARE

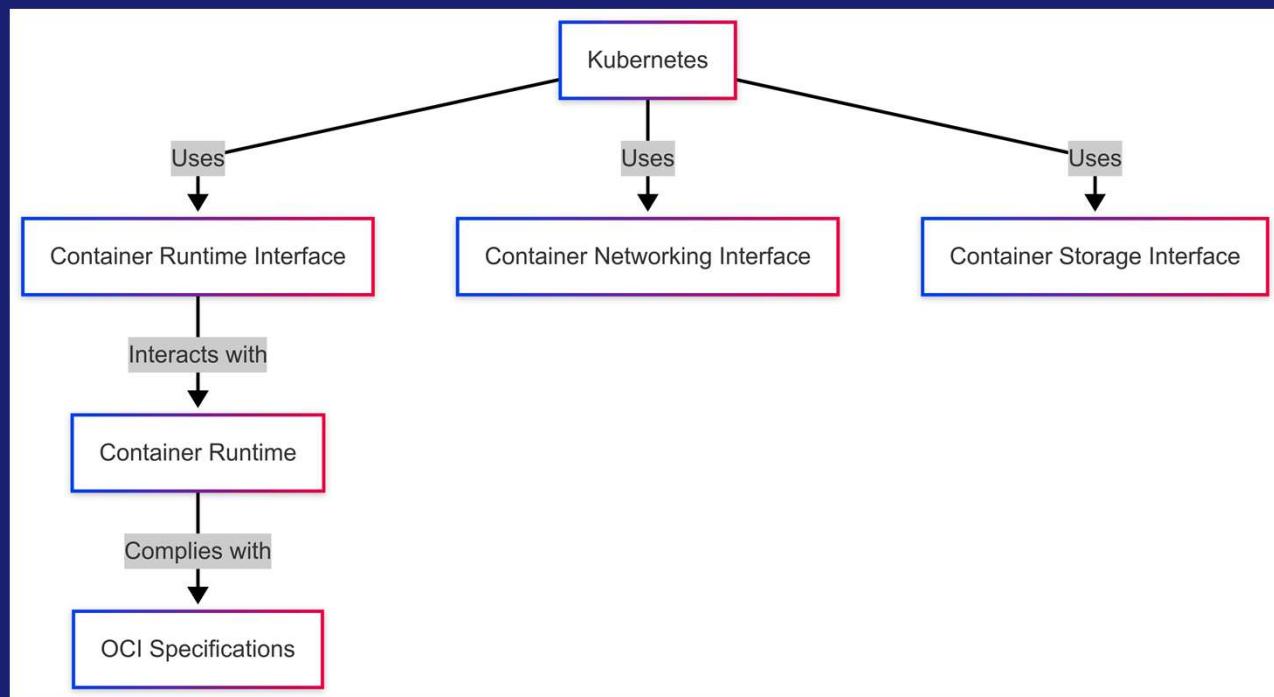
62

설명

쿠버네티스에서 대표적으로 지원하는 네트워크 기능은 다음과 같다.

솔루션	기본 개념 및 특징	네트워크 방식	네트워크 정책 지원	사용 사례 및 장점
Calico	컨테이너 네트워킹과 네트워크 정책을 동시에 제공하는 솔루션	기본적으로 BGP 기반 라우팅, 필요 시 VXLAN, IP-in-IP 사용	강력한 정책 엔진 제공	대규모 클러스터 및 고급 보안 요구 환경에 적합
Flannel	간단한 오버레이 네트워크 솔루션으로, 클러스터 내부 Pod 간 통신을 지원	주로 VXLAN 또는 Host-GW 모드 사용	기본 기능만 제공 (별도 정책 도구 필요)	설정과 운영이 간단하여 작은 클러스터나 테스트 환경에 적합
Cilium	eBPF 기반으로 고성능 패킷 처리와 네트워크 정책, 보안을 제공하는 솔루션	eBPF를 사용하여 직접 패킷 처리, 오버레이 지원	세밀한 네트워크 정책 및 가시성 제공	클라우드 네이티브, 마이크로서비스 환경에서 고성능, 고가용성 제공
Canal	Flannel의 단순 오버레이와 Calico의 정책 기능을 결합한 통합 솔루션	Flannel 방식의 오버레이와 Calico 정책 적용	Calico 기반의 정책 엔진 활용	간단한 네트워크 구축과 정책 적용을 동시에 원하는 환경에 적합

설명



설명

네트워크 CNI는 환경 및 구성에 따라서 적절하게 선택해서 사용한다. 위의 내용은 CNI에서 대표적인 내용이며, 추가적으로 더 확인이 필요한 경우 아래 사이트를 참고한다.

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

CALICO 설명

기본적으로 VXLAN기반으로 사용하는 터널링 프로그램. CALICO는 오퍼레이터 기반으로 설치 및 구성이 된다. 이러한 이유로 장애가 발생하거나 혹은 알 수 없는 이유로 제거가 되었을 때, 자동으로 복구(Healing)기능이 있다.

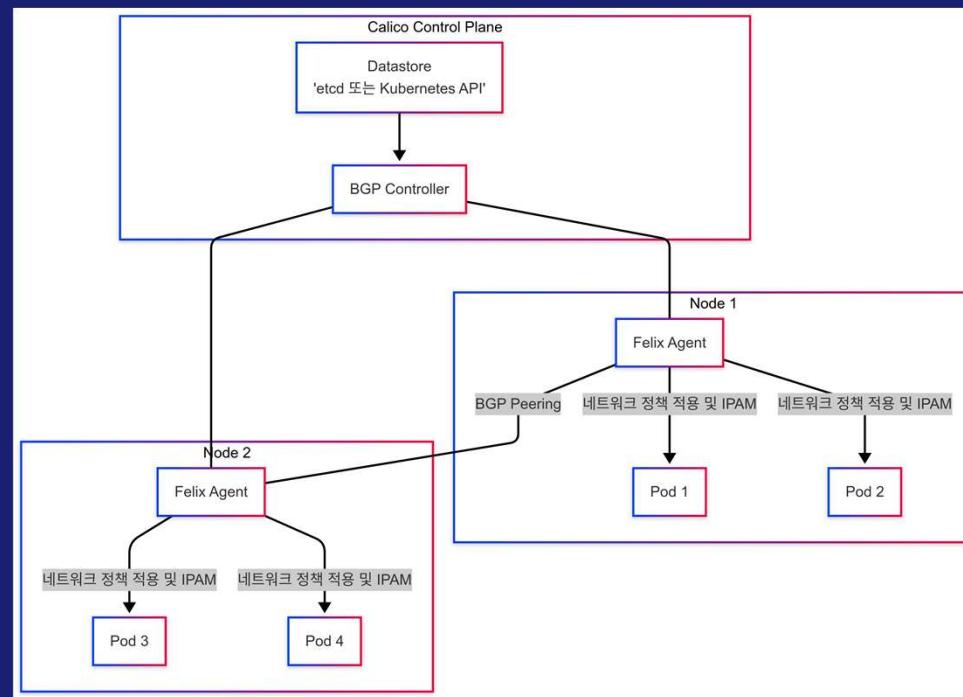
Calico를 사용할 때 다음과 같은 제한 사항이 있다

1. OpenStack와 같은 SDN네트워크 기반에서는 사용이 불가능
2. Provider형식으로 구성된 네트워크에서는 제한 사항이 없음
3. BGP사용이 안되거나 IP프로토콜 제한이 있는 경우 사용이 불가능

특히, 오픈스택을 사용하는 경우 Calico기반으로 올바르게 네트워크 구성이 되지 않기 때문에 오픈스택 사용자는 Calico대신 다른 네트워크 프로토콜을 권장.

Calico는 ToR(Top Of Rack)형식으로 라우팅을 구성하기 때문에, 기존 VXlan보다 빠른 트래픽 처리 및 라우팅이 가능하다.

CALICO 설명



CALICO 설정

```
# cat <<EOF> calico-quay-crd.yaml
---
apiVersion: operator.tigera.io/v1
kind: Installation
aladata:
  name: default
spec:
  calicoNetwork:
    bgp: Enabled      OpenStack에서 사용시 bgp: Disabled
    hostPorts: Enabled  확장 기능을 사용하지 않으면Disabled
```

CALICO 설정

```
ipPools:  
  - blockSize: 26  
    cidr: 10.0.0.0/16  
    encapsulation: IPIPCrossSubnet  
    natOutgoing: Enabled  
    nodeSelector: all()  
  
registry: quay.io  
  
EOF  
  
# kubectl apply -f calico-quay-crd.yaml
```

CALICO 설치

```
# 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
```

FLANNEL 설명

쿠버네티스에서 제일 기본적으로 제공하는 vxlan기반 터널링 프로그램. 대다수 쿠버네티스 플랫폼들은 높은 호환성을 위해서 Flannel기반으로 네트워크 및 터널링을 제공한다.

Flannel의 최대 장점은 추가 기능이 없기 때문에, 플랫폼 제한 없이 바로 사용이 가능하다. 기존의 vxlan과 동일하게 UDP기반으로 동작한다. 최대 장점은 손쉬운 설치 및 유지 보수이지만, 단점으로는 확정성의 부재가 있다.

Calico와 비교하면, BGP나 Host-GW기능 기반으로 동작하기 때문에 보안이나 혹은 기능 추가가 어려운 부분이 있다. 대규모 크기의 쿠버네티스 클러스터가 아니면 Flannel기반으로 구축 및 사용하여도 문제가 없다.

FLANNEL 설치

```
# kubectl create ns kube-flannel  
  
# kubectl label --overwrite ns kube-flannel pod-  
security.kubernetes.io/enforce=privileged  
  
# helm repo add flannel https://flannel-io.github.io/flannel/  
  
# helm install flannel --set podCidr="192.168.0.0/16" --namespace kube-  
flannel flannel/flannel --create-namespace  
  
# kubectl -n kube-flannel get pod
```

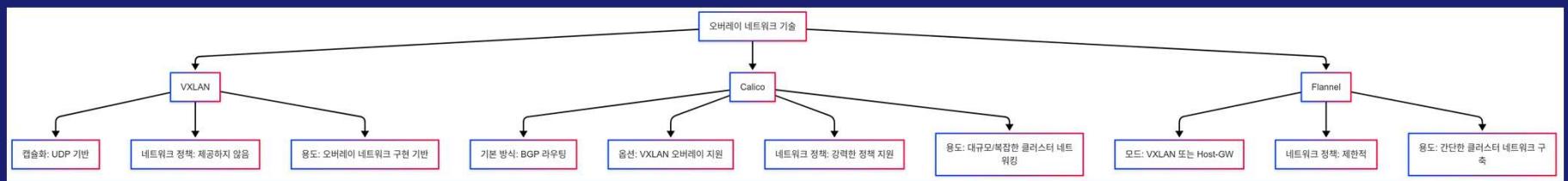
정리

위에서 명시한 기능을 정리하면 다음과 같다.

항목	VXLAN	Calico	Flannel
기본 개념	레이어2 오버레이 프로토콜로, L2 네트워크를 L3 환경에서 확장하기 위한 캡슐화 기술입니다.	컨테이너 네트워킹 및 네트워크 정책 솔루션으로, 기본적으로 BGP를 이용한 라우팅 방식이며 필요 시 VXLAN과 같은 오버레이 모드를 지원합니다.	Kubernetes 클러스터에서 간단하게 오버레이 네트워크를 구축하기 위한 솔루션으로, 주로 VXLAN 또는 Host-GW 모드를 사용합니다.
네트워크 방식	UDP를 기반으로 패킷을 캡슐화하여 오버레이 네트워크를 구성합니다.	기본적으로 라우팅(BGP)을 이용해 직접 연결을 구성하며, 오버레이(VXLAN) 방식도 선택적으로 사용합니다.	오버레이 네트워크 방식(VXLAN) 또는 단순 라우팅(Host-GW) 방식을 지원합니다.
네트워크 정책 지원	단순 캡슐화 기술로 자체 네트워크 정책 기능은 제공하지 않습니다.	강력한 네트워크 정책 및 보안 기능을 내장하여 세밀한 트래픽 제어가 가능합니다.	기본적인 네트워크 구성을 제공하지만, 고급 네트워크 정책 지원은 제한적입니다.
구성 및 사용 사례	주로 오버레이 네트워크 구현의 기술적 기반으로 다른 솔루션에서 채택됩니다.	복잡한 환경이나 대규모 클러스터에서 정책 기반의 네트워킹을 구현할 때 많이 사용됩니다.	설정이 간단해 작은 규모의 클러스터나 기본 네트워크 구축 시 유용합니다.

정리

위에서 명시한 기능을 정리하면 다음과 같다.



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/repo/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/repo/repodata/repomd.xml.key
EOF
```

호스트 이름

```
nodeX# hostnamectl set-hostname node[2-3].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
```

호스트 파일 설정

```
# 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
```

커널 모듈 및 런타임

```
# systemctl enable --now crio kubelet  
  
# modprobe br_netfilter && modprobe overlay  
  
# cat <<EOF > /etc/modules-load.d/k8s-modules.conf  
br_netfilter  
  
overlay  
  
EOF
```

커널 파라미터

```
# 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
```

컴퓨트 조

```
# 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
```

컴퓨트 조인

```
# 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/repo/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/repo/repodata/repomd.xml.key
EOF
```

호스트 이름

```
nodeX# 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
```

호스트 파일 설정

```
# 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
```

커널 모듈 및 런타임

```
# systemctl enable --now crio kubelet  
  
# modprobe br_netfilter && modprobe overlay  
  
# cat <<EOF > /etc/modules-load.d/k8s-modules.conf  
br_netfilter  
  
overlay  
  
EOF
```

커널 파라미터

```
# 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
```

컴퓨트 조인

```
# 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
```

컴퓨트 조인

```
# kubeadm join --config kubeadm-join-config.yaml  
# kubectl get pod -n kube-system -w  
calico-node-<ID>  
coredns
```

HIGH LEVEL RUNTIME

PODMAN

2025-03-16

PREPARE

96

설명

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

현재 쿠버네티스 클러스터 랩은 다음과 같은 구성으로 되어있다.

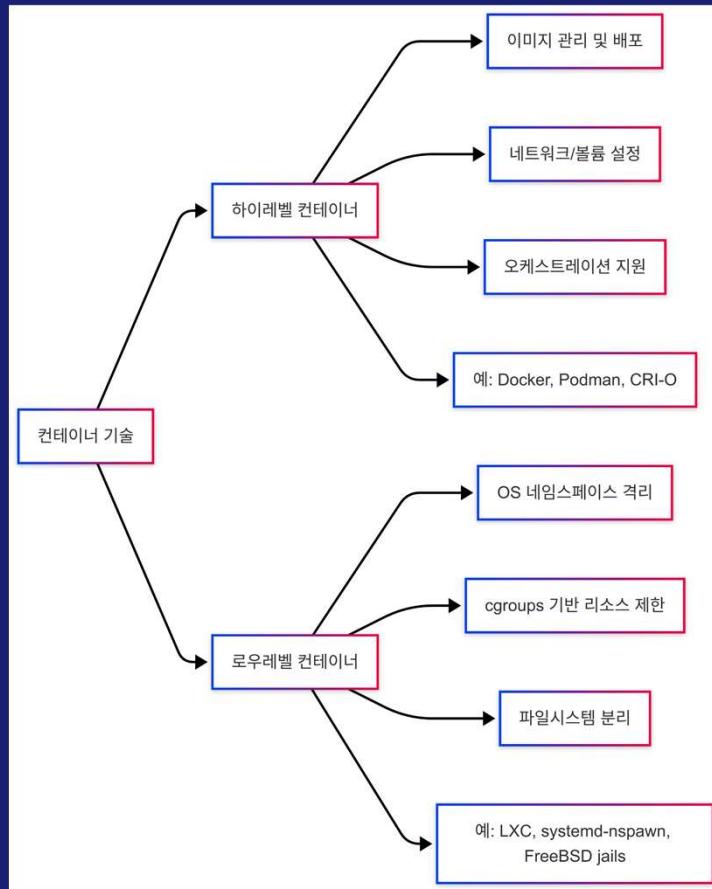
- **node1(controller)**
- **node2(compute)**
- **node3(compute)**
- **node4(podman)**

설명

로우레벨/하이레벨 비교하면 다음과 같다.

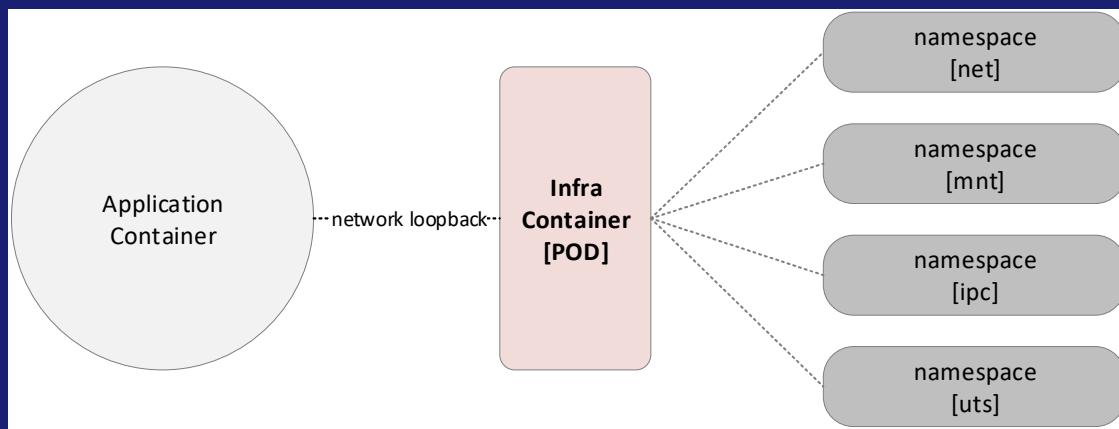
항목	하이레벨 컨테이너	로우레벨 컨테이너
개념	애플리케이션 중심의 컨테이너 관리 도구로, 이미지 관리, 네트워크, 볼륨 등의 부가 기능을 제공함	OS 레벨의 격리 메커니즘(cgroups, 네임스페이스 등)을 직접 활용하여 프로세스와 파일시스템을 분리하는 기본 기술
주요 기능	이미지 빌드/배포, 네트워크 및 스토리지 관리, 오케스트레이션(예: 클러스터 관리) 지원	프로세스 격리, 리소스 제한, 파일 시스템 분리 등 기본적인 격리 기능 제공
사용 용도	개발, 배포 자동화 및 마이크로서비스 아키텍처 구축	OS 수준의 리소스 격리, 환경 분리, 커널 기능 활용 및 실험적/개별 격리 용도
대표 예	Docker, Podman, CRI-O	LXC, systemd-nspawn, FreeBSD jails

설명



설명

쿠버네티스에서 관리하는 애플리케이션 자원은 POD단위다. POD기본적으로 최소 한 개의 POD와 최소 한 개의 CONTAINER를 통해서 구성이 된다.



설명

컨테이너는 애플리케이션이 실행이 되는 영역이며, POD는 실행되는 애플리케이션 자원을 관리를 한다. POD는 개념적으로 부르는 이름이며, 시스템 영역에서는 POD를 Infra-container라고 부른다. infra-container는 무엇이되었든, 컨테이너이기 때문에 최소 한 개의 애플리케이션이 실행이 된다. 실행되는 애플리케이션 /pause라는 이름으로 보통 사용한다(바닐라 쿠버네티스 기준).

이 애플리케이션의 용도는 POD개념 구현 및 네임스페이스 자원을 관리한다. 자세한 내용은 아래 주소에서 확인이 가능하다.

<https://github.com/kubernetes/kubernetes/blob/master/build/pause/linux/pause.c>

설명

```
if (getpid() != 1)
    /* Not an error because pause sees use outside of infra containers. */
    fprintf(stderr, "Warning: pause should be the first process\n");

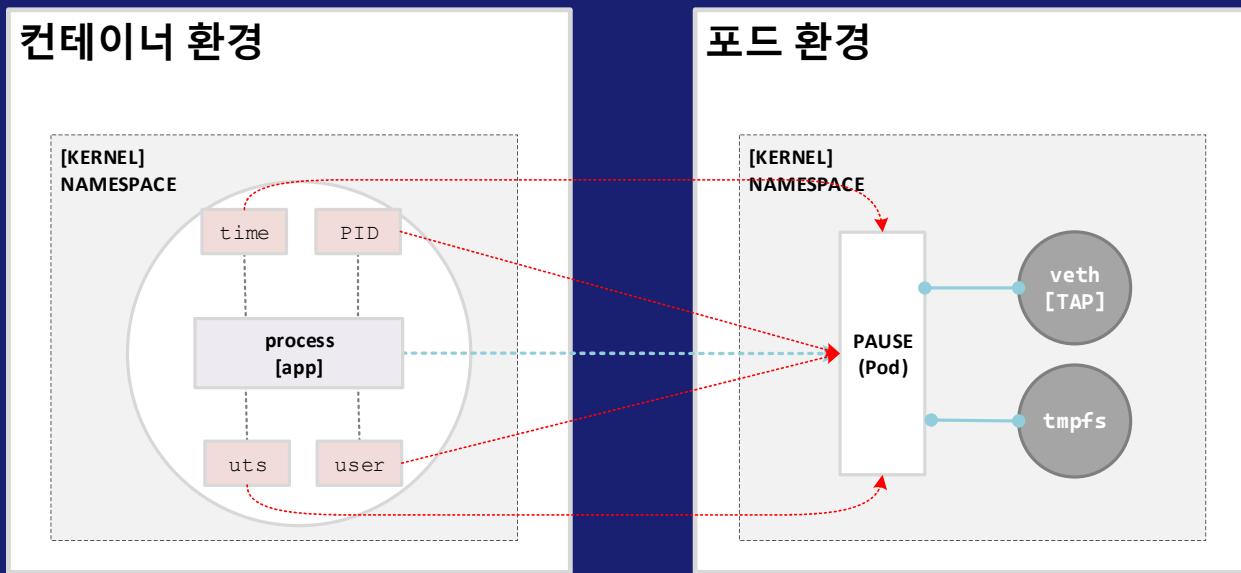
if (sigaction(SIGINT, &(struct sigaction){.sa_handler = sigdown}, NULL) < 0)
    return 1;
if (sigaction(SIGTERM, &(struct sigaction){.sa_handler = sigdown}, NULL) < 0)
    return 2;
if (sigaction(SIGCHLD, &(struct sigaction){.sa_handler = sigreap,
                                             .sa_flags = SA_NOCLDSTOP},
              NULL) < 0)
    return 3;
```

설명

```
for (;;) {
    pause();
    fprintf(stderr, "Error: infinite loop terminated\n");
    return 42;
}
```

POD

POD기반으로 애플리케이션을 생성하면 아래와 같이 구성된다.



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 설명

배포판 및 회사마다 다르게 POD에서 사용하는 프로그램을 사용한다. 레드햇 계열 경우에는 쿠버네티스의 POD를 사용하지 않고, *catatonit* 기반으로 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 설명

레드햇 및 수세에서 사용하는 POD와 바닐라 쿠버네티스의 POD는 다른 부분이 있다.

항목	pause 컨테이너	catatonit
역할	Pod 내에서 네임스페이스(네트워크, PID, IPC 등)를 고정하는 역할을 수행(컨테이너들이 공통 네임스페이스를 공유하게 함)	Pod의 인프라 역할을 하며, 네임스페이스 고정 외에 프로세스 재구성 등 최소한의 init 기능을 제공
기능	단순 “존재”만으로 네임스페이스를 유지하며, 아무런 작업도 수행하지 않음	최소한의 init 기능을 제공하여, 좀 더 견고한 프로세스 관리를 지원 (예: 좀비 프로세스 정리)
구현 및 크기	극도로 최소화되어 있어 이미지 크기가 매우 작음	마찬가지로 경량이지만, 추가 기능으로 인해 약간의 오버헤드가 있을 수 있음
사용 배경	Kubernetes Pod의 기본 인프라 컨테이너로 오랜 기간 사용됨	CRI-O, containerd 등 최신 컨테이너 런타임에서 pause 컨테이너 대체제로 주목받음
주요 장점	간단하고 안정적이며, 기존 Kubernetes 환경에서 광범위하게 검증됨	추가적인 프로세스 관리를 제공하여, 보다 견고한 Pod 기반 인프라 환경 구축에 도움

POD 설명

POD는 다음과 같은 기능을 한다.

구분	역할 및 기능	설명
POD	하나 이상의 컨테이너를 포함하는 최소 배포 단위	컨테이너들이 동일한 네트워크, 스토리지, 설정을 공유하며 함께 스케줄링됨
Infra Container	POD 내의 네임스페이스(특히 네트워크 네임스페이스) 관리를 위한 기반 역할 제공	보통 Pause 컨테이너가 Infra Container로 사용되며, POD의 다른 컨테이너와 네임스페이스를 공유
Pause 애플리케이션	실제로 아무 작업도 수행하지 않는 빈 컨테이너로서, 네임스페이스 유지 및 기반 제공 역할 수행	컨테이너 그룹의 네트워크 및 기타 리소스를 지속적으로 유지하기 위해 항상 실행되어 있음

POD 설명

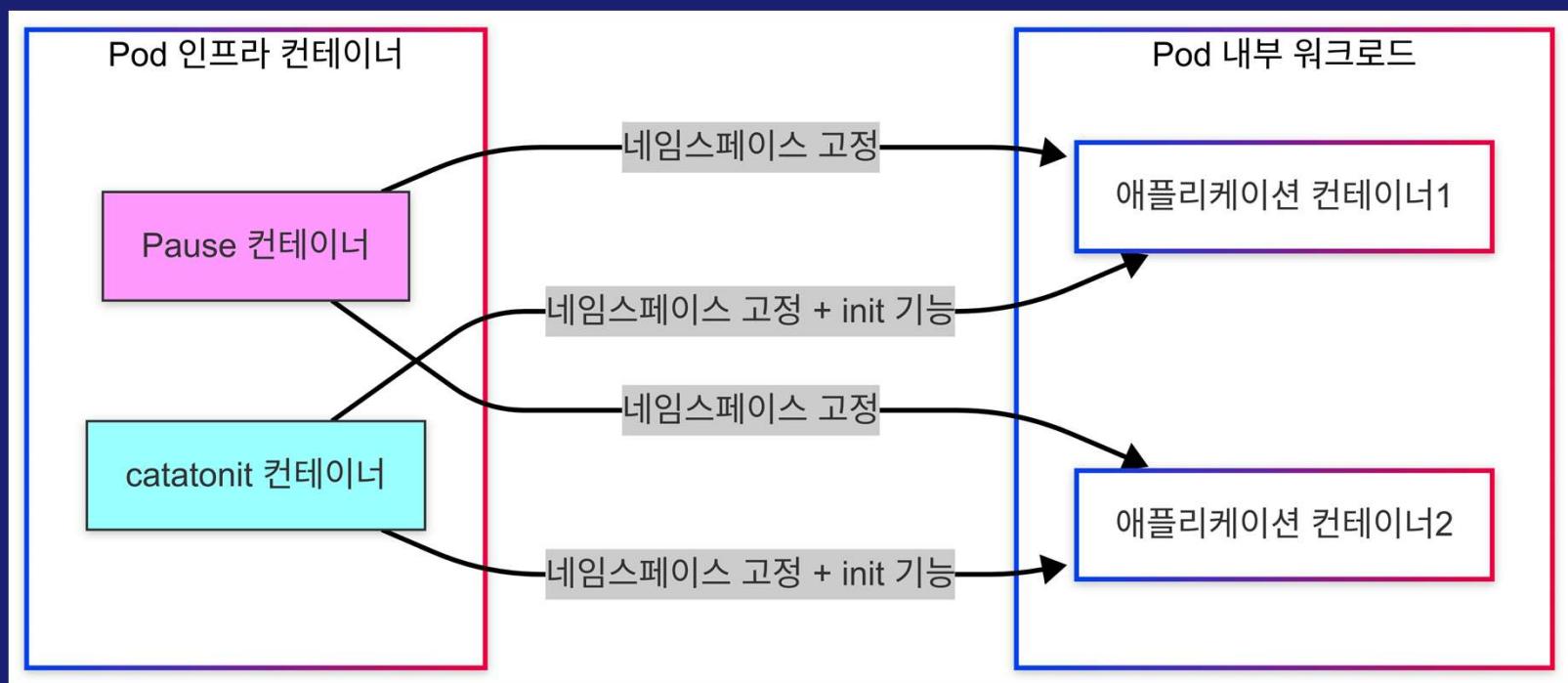
POD는 다음과 같은 기능을 한다.

항목	설명
이미지	registry.k8s.io/pause (이전에는 k8s.gcr.io/pause를 사용)
버전	예: 3.8 또는 Kubernetes 버전에 따라 업데이트된 버전
주요 역할	POD의 네트워크 및 기타 리소스 네임스페이스를 유지하기 위한 기반 컨테이너 역할 수행
사용 이유	POD 내에서 컨테이너 간 리소스 공유(특히 네트워크)를 위해 항상 실행되어야 하는 기반 제공

POD 설명



POD 설명



2025-03-16

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

2025-03-16

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. 3일 과정으로 진행하는 경우, DevOps를 진행하지 않기 때문에, BIND9설치 및 구성은 필요하지 않는다.

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
```

INFRA POD

```
# 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
```

BIND9

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
```

BIND9

존 파일을 아래와 같이 생성한다.

```
# vi /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
```

BIND9

존 파일을 아래와 같이 생성한다.

	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.

BIND9

최종적으로 조회가 가능하도록 아래와 같이 수정한다.

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

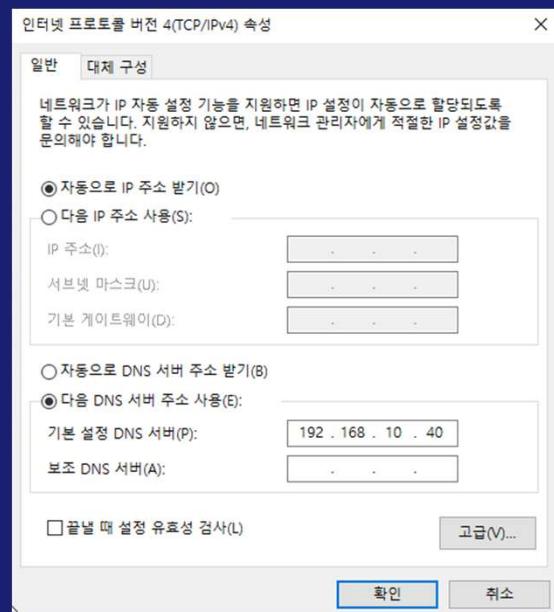
# systemctl enable --now named
# systemctl status named

# 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

BIND9

윈도우 클라이언트는 다음과 같이 DNS 정보를 수정한다. 3일로 진행하는 과정에서는 윈도우에 이 설정은 필요하지 않는다.



2025-03-16

실험코드

SPRING(JSP/TOMCAT/PGSQL)

2025-03-16

설명

3일 기본 과정에서는 굳이 진행이 필요 없다. CI/CD인터페이스 구성 시, 실험코드가 필요하다.

준비

```
# dnf install git -y
# git clone https://github.com/tangt64/codelab 기본 이미지는 docker.io에서 받는다.
# 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
```

준비

```
# 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
```

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)

2025-03-16

설명

쿠버네티스에서는 기본적으로 한 개 이상의 CSI Storage Class가 필요하다. 랩 진행을 위해서 최소한 CSI NFS를 설치한다. 그 이외 스토리지 인터페이스는 하드웨어 혹은 기간에 따라서 강사가 추가적으로 설치 및 구성할 수 있다. 3일 과정에서는 NFS기반으로 진행한다. 이외 일정은 장비 상태에 따라서 선택한다. 해당부분은 스토리지 클래스 부분에서 한번 더 다루기 때문에, 추후에 진행 하여도 상관이 없다.

설치

미리 스토리지 서버를 구성하는 경우, 미리 OS에 아래와 같이 작업을 수행한다.

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

설치

NFS설정은 가급적이면 exports.d/에 생성 및 구성한다.

```
# cat <<EOF> /etc/exports.d/kubernetes.exports
/nfs *(rw,sync,no_root_squash,insecure,no_subtree_check,nohide)
EOF
# systemctl enable --now nfs-server
# exportfs -avrs
# showmount -e storage.example.com
# systemctl disable --now firewalld
```

드라이버 설치

```
# 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
```

스토리지 클래스 설정

```
# cat <<EOF> storageclass-configure.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: default
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
```

스토리지 클래스 설정

```
provisioner: nfs.csi.k8s.io
parameters:
  server: storage.example.com
  share: /nfs
reclaimPolicy: Delete
volumeBindingMode: Immediate
mountOptions:
  - hard
  - nfsvers=4.1
EOF
# kubectl apply -f storageclass-configure.yaml
```

NFS PVC

올바르게 동작하는지 StorageClass에 PVC 생성 요청을 한다.

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

확인(SC/PV/PVC)

올바르게 구성이 되었는지 최종적으로 자원 생성 후 확인한다.

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



YAML

EDITOR

YAML

자원은 YAML 코드로 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. 간격은 2칸으로 설정한다.

```
# vi test-pod.yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
  ports:
  - containerPort: 80
```

에디터

에디터 띄어쓰기는 다음과 같이 설정한다.

```
" 2칸 띄어쓰기 관련 설정
set tabstop=2          " 실제 탭의 폭을 2칸으로 설정
set shiftwidth=2        " >>, << 등의 명령으로 들여쓰기할 때 2칸씩 이동
set expandtab          " 탭 대신 스페이스를 삽입
set autoindent         " 이전 줄의 들여쓰기 복사
```

vim/neovim기반으로 YAML Linting도 가능하지만, 복잡하고 시간이 오래 걸리기 때문에 여기서는 사용하지 않는다.

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

2025-03-16

설명

기본적으로 명령어 사용 및 활용방법에 대해서 이야기 한다. 여기서 다루는 자원은 쿠버네티스에서 매우 중요하며, 꼭 필요한 자원 및 개념이다.

1. 네임스페이스
2. POD
3. C-GROUP
4. YAML/JSON

NAMESPACE/RUN/POD

기본 명령어

2025-03-16

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, **/pause**라는 애플리케이션을 사용한다. 이 애플리케이션 각 회사마다 다르게 사용하며, 레드햇 경우에는 **cataonit**를 기본으로 사용한다.

/pause 애플리케이션은 호스트 시스템의 /bin/init 혹은 /bin/systemd와 동일한 기능을 하며, 애플리케이션에 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.

POD/CGROUP

CGROUP부분을 정리하면 다음과 같다.

항목	내용
표준화 문제	디스크, 네트워크 자원은 하드웨어 및 커널 구현 방식이 다양해 표준화가 어려움
기술적 복잡성	디스크 I/O 및 네트워크 대역폭의 계량화와 격리는 애플리케이션별 요구가 달라 어려움
커뮤니티와 연구 동향	Extended Resources를 통한 GPU 지원처럼, 관련 기능에 대한 논의 중
결론	추후 디스크와 네트워크 자원에 대한 지원 가능성은 있으나, 표준화와 기술적 도전

RUN

RUN명령어는 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 제거 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
```

YAML

위의 명령어를 YAML형태로 출력이 가능하다. kubectl명령어에서는 직접적으로 JSON를 다루지 않는다.

```
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: {}
```

CREATE/APPLY/GET/DELETE

명령어

2025-03-16

설명

create/apply 그리고 API의 차이점은 다음과 같다.

방법	설명	주요 특징 및 차이점	사용 시나리오
kubectl create	명령 실행 시 한 번 리소스 생성	1. 선언적 구성이 아니라 즉시 생성 2. 이미 존재하는 경우 에러 발생 3. 간단한 리소스 생성에 적합	초기 리소스 생성, 빠른 테스트, 임시 리소스 생성
kubectl apply	선언적構成을 기반으로 리소스를 생성 / 업데이트	1. YAML에 "last-applied-configuration" 주석을 남겨 추후 업데이트시 비교 및 병합 2. 변경된 부분만 업데이트 3. 리소스가 없으면 생성, 있으면 업데이트	지속적인 구성 관리, GitOps 방식, 선언적 인프라 관리
YAML/JSON을 API로 생성	Kubernetes API 서버에 직접 YAML 또는 JSON 데이터를 전송하여 생성	1. REST API를 직접 호출 (예: POST, PUT 요청) 2. 클라이언트(스크립트, SDK 등)를 통해 세밀한 제어 가능 3. 자동화, 커스텀 컨트롤러에서 활용됨	프로그램적 리소스 생성, CI/CD 파이프라인, 커스텀 오퍼레이터 개발

설명

앞에 내용 이어서...

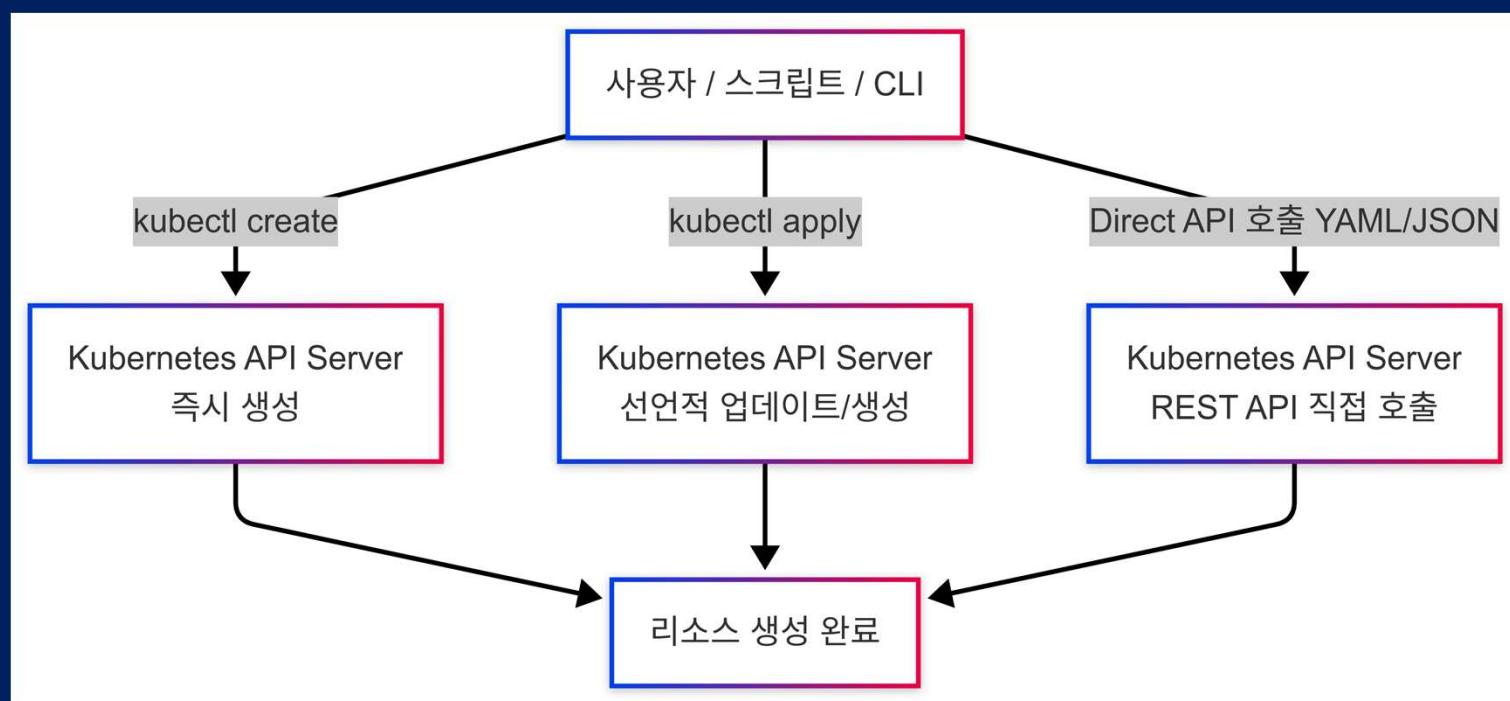
항목	kubectl apply -k	kubectl create -k
작동 방식	선언적 방식으로 리소스 구성 및 업데이트 (last-applied-configuration 기록)	명령 실행 시 한 번 리소스를 생성 (이미 존재하면 에러 발생)
목적	선언적 관리: 이미 생성된 리소스를 업데이트하거나, 리소스가 없으면 생성 지속적인 구성 관리에 유리	최초 생성: 새로운 리소스를 한 번 생성할 때 사용 기존 리소스 업데이트에는 적합하지 않음
업데이트 처리	변경사항을 감지하여 패치 방식으로 업데이트 GitOps 등 지속적 관리에 용이	이미 생성된 리소스가 있으면 재생성 시 에러가 발생함
사용 시나리오	리소스의 장기적인 관리 및 변경 추적이 필요할 때 (선언적 구성 관리)	단발성 리소스 생성 또는 테스트 시 사용

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.	

설명



CREATE

명령어는 다음과 같이 사용한다.

```
# kubectl create -f php-pod-v1.yaml  
# kubectl get pod  
php-pod  
# kubectl run pod --image=nginx pod-nginx --output=yaml \  
--dry-run=client > pod-nginx.yaml  
# kubectl create -f pod-nginx.yaml
```

CREATE

계속 이어서…

```
# kubectl create deployment --image=nginx deploy-nginx --output=yaml \
--dry-run=client > deploy-nginx.yaml
# kubectl create -f deploy-nginx.yaml
# kubectl create service --type=NodePort --output=yaml --dry-run=client >
svc-nginx.yaml
# kubectl create -f deploy-nginx.yaml
```

CREATE(KUSTOMIZE)

쿠버네티스에서 YAML기반으로 패키지 배포 기능을 제공한다. 다음과 같이 사용이 가능하다. `apply`나 `create` 명령어로 적용이 가능하다.

```
# vi kustomize/application.properties
FOO=Bar

# vi kustomize/kustomization.yaml
configMapGenerator:
- name: example-configmap-1
  files:
  - application.properties
```

CREATE(KUSTOMIZE)

적용은 아래 명령어로 가능하다.

```
# kubectl create -k ./  
# kubectl get configmap  
# kubectl describe cm(configmap) example-configmap-1-t7998h956b
```

CREATE(KUSTOMIZE)

kustomize에서 사용할 deployment를 생성한다.

```
# vi kustomize/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: my-app
```

CREATE(KUSTOMIZE)

앞 내용 계속 이어서…

```
spec:  
  selector:  
    matchLabels:  
      app: my-app  
  template:  
    metadata:  
      labels:  
        app: my-app
```

CREATE(KUSTOMIZE)

앞 내용 계속 이어서…

```
spec:  
  containers:  
    - name: app  
      image: my-app  
  volumeMounts:  
    - name: config  
      mountPath: /config
```

CREATE(KUSTOMIZE)

앞 내용 계속 이어서…

```
volumes:  
  - name: config  
    configMap:  
      name: example-configmap-1
```

CREATE(KUSTOMIZE)

올바르게 잘 구성하였는지 확인한다.

```
# kubectl kustomize ./  
# vi kustomization.yaml  
  
resources:  
- deployment.yaml  
  
configMapGenerator:  
- name: example-configmap-1  
  
  files:  
  - application.properties  
# kubectl get pod,configmap
```

APPLY

apply명령어는 create와 거의 동일하다. 다만 차이점은 지속적으로 자원 업데이트가 가능하다. 간단하게 아래와 같이 작성한다.

```
# kubectl create deployment --image=nginx:1.26.3 deploy-nginx \
--output=yaml --dry-run=client > deploy-nginx.yaml
# kubectl apply -f deploy-nginx.yaml
# kubectl get -f deploy-nginx.yaml
# kubectl create deployment --image=nginx:1.27.4 deploy-nginx \
--output=yaml --dry-run=client > deploy-nginx.yaml
# kubectl apply -f deploy-nginx.yaml
```

GET

get명령어를 통해서 자원 확인이 가능하다.

```
# kubectl get pods  
# kubectl get pods --namespace <NAMESPACE_NAME>  
# kubectl get deployment -A  
# kubectl get deployment --all-namespaces  
# kubectl get pods -w
```

GET

앞 내용 계속 이어서…

```
# kubectl get pods -o=json  
# kubectl get pod -l <LABEL_NAME>=<LABEL_VALUE>  
# kubectl describe -f deploy-nginx.yaml  
# kubectl create -f deploy-nginx.yaml  
# kubectl describe -f deploy-nginx.yaml
```

DELETE

자원 제거 시, 사용하는 명령어. YAML이나 혹은 자원을 명시하여 제거가 가능하다.

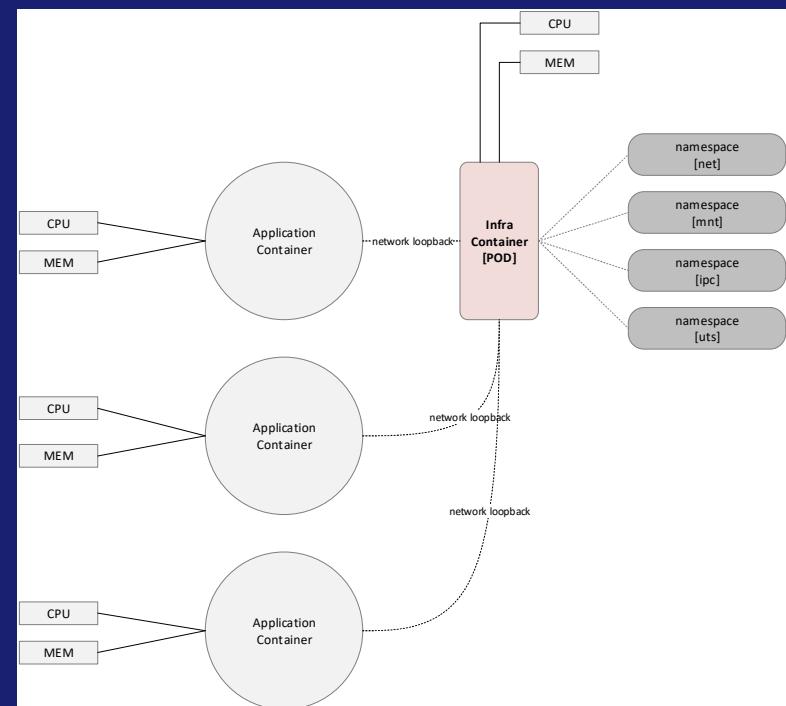
```
# kubectl delete -f deploy-nginx.yaml  
# kubectl delete pod/deploy-nginx
```

다중 컨테이너 설명

다중 컨테이너는 한 개의 POD에 하나 이상의 컨테이너가 구성이 되어있다.

POD에 연결된 컨테이너들은 자신이 사용하는 자원 **disk**, **memory**, **cpu** 그리고 **network** 같은 부분들은 POD를 통해서 위임한다.

여기서 표현하는 위임은 POD가 실제 물리적 자원을 관리하는 게 아닌, 네임스페이스를 통해서 POD가 관리하게 된다.



다중컨테이너 구성 및 사용 방법

다중 컨테이너 구성은 간단하다. POD생성 후, 컨테이너를 POD에 연결(attached)하면 된다. POD에 연결된 컨테이너는 POD에서 몇몇 자원을 컨테이너 대신 소유(share, own) 및 제공(provider)을 한다.

생성하기 위해서는 YAML파일 생성이 필요하다. 쉽게 생성하기 위해서 POD파일 생성 후, 설정을 추가한다.

```
# kubectl run --image=nginx wordpress --dry-run=client --output=yaml > wordpress.yaml  
# vi wordpress.yaml
```

다중컨테이너 구성 및 사용 방법

다중 컨테이너 구성을 위해서 containers:에 다음과 같이 한 개 이상의 컨테이너 이미지를 선언한다.

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: wordpress
  name: wordpress
```

다중컨테이너 구성 및 사용 방법

계속 이어서...

```
spec:  
  containers:  
    - image: nginx  
      name: wordpress-web  
    - image: mariadb  
      name: wordpress-db  
      resources: {}  
  dnsPolicy: ClusterFirst  
  restartPolicy: Always
```

다중컨테이너 구성 및 사용 방법

위에서 작성한 내용을 클러스터에서 생성한다.

```
# kubectl create -f wordpress.yaml  
# kubectl get -f wordpress.yaml
```

다중컨테이너 구성

혹은 아래처럼 컨테이너에서 명령어 실행이 가능하다.

```
# kubectl run --image=alpine --dry-run=client --output=yaml run-pod --  
sleep 50 > run-pod.yaml
```

위와 같이 파일 생성, 후 명령어를 추가해서 실행한다. 구성은 아래 슬라이드 참고를 한다

다중 컨테이너 생성 #1

다중 컨테이너 생성을 위해서 아래와 같이 구성한다.

```
# vi multi-container-1.yaml
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: run-pod
  name: run-pod
```

다중컨테이너 생성 #1

```
spec:  
  containers:  
    - args:  
        - sleep  
        - "50"  
      image: alpine  
      name: run-pod-1  
    - args:  
        - echo  
        - "hello world"  
      image: alpine  
      name: run-pod-2
```

다중컨테이너 생성 #2

```
# vi multi-container-2.yaml
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: run-pod
  name: run-pod
spec:
  containers:
  - args: ["-c", "sleep 10"]
    command: ["sh"]
    image: alpine
    name: run-pod-1
```

다중컨테이너 생성 #2

```
- args: ["-c", "echo 'hello world'"]
  command: ["sh"]
  image: alpine
  name: run-pod-2
```

서비스

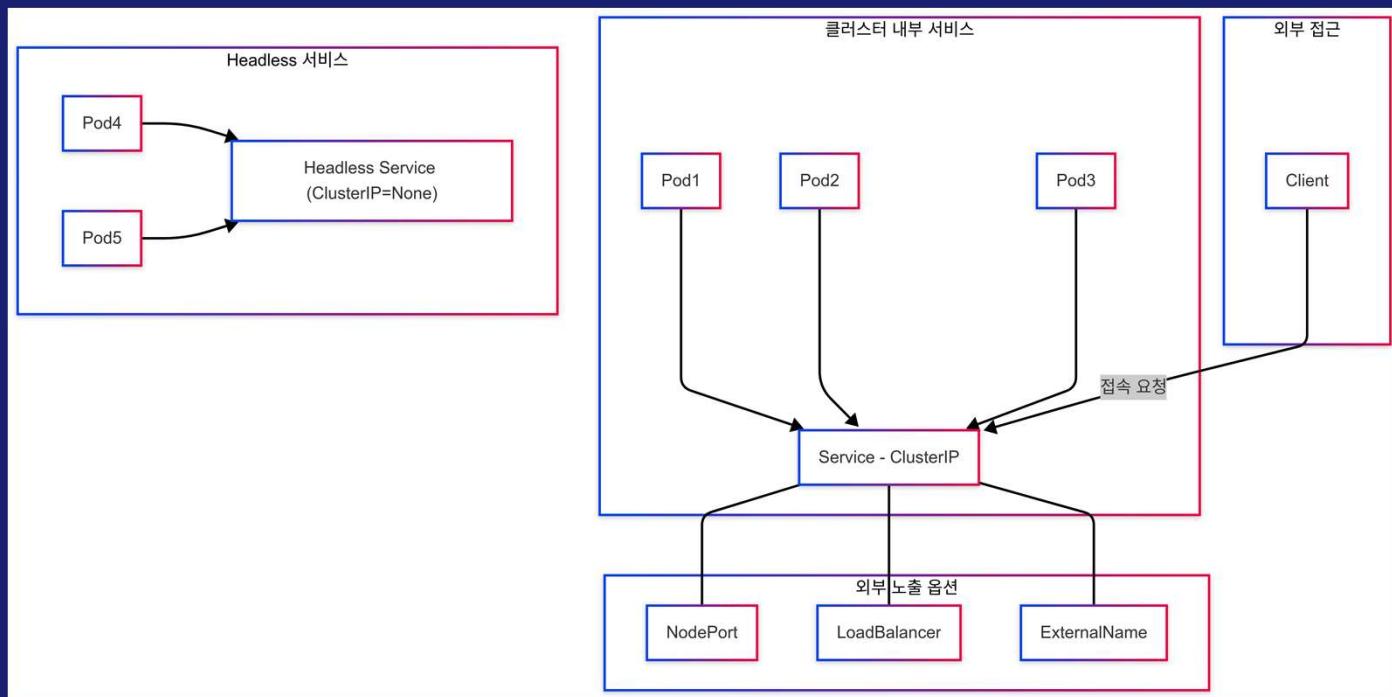
기본 명령어

2025-03-16

설명

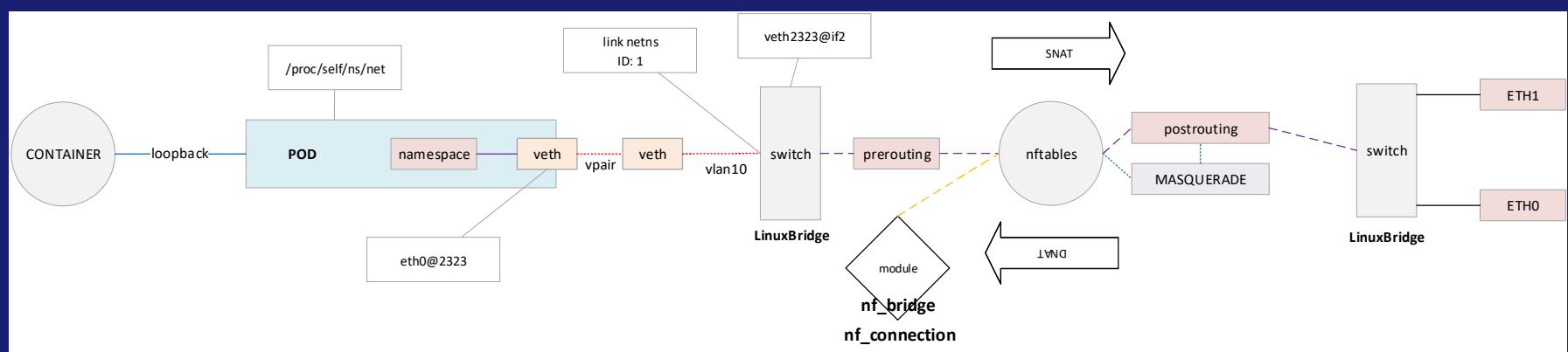
항목	설명	사용 예시
ClusterIP	클러스터 내부에서만 접근 가능한 가상 IP를 할당해 내부 통신을 지원합니다.	내부 마이크로서비스 간 통신, 클러스터 내 API 서버 접근
NodePort	각 노드의 고정 포트를 통해 외부에서 접근할 수 있도록 노출합니다.	외부 접근이 필요한 테스트 환경이나 개발 단계의 서비스
LoadBalancer	클라우드 제공업체의 로드 밸런서를 이용하여 외부 접근을 위한 단일 IP를 할당합니다.	프로덕션 환경에서 안정적인 외부 접근 제공
ExternalName	DNS 이름을 통해 클러스터 외부의 서비스를 참조할 수 있도록 합니다.	외부 데이터베이스나 API를 내부 서비스처럼 사용하고자 할 때
Headless	클러스터 IP가 할당되지 않아 DNS SRV 레코드를 통해 각 Pod의 IP를 직접 반환, Pod의 개별 주소를 노출합니다.	StatefulSet, 분산 시스템 등에서 Pod 간 직접 통신이나 서비스 디스커버리 목적으로 사용

설명



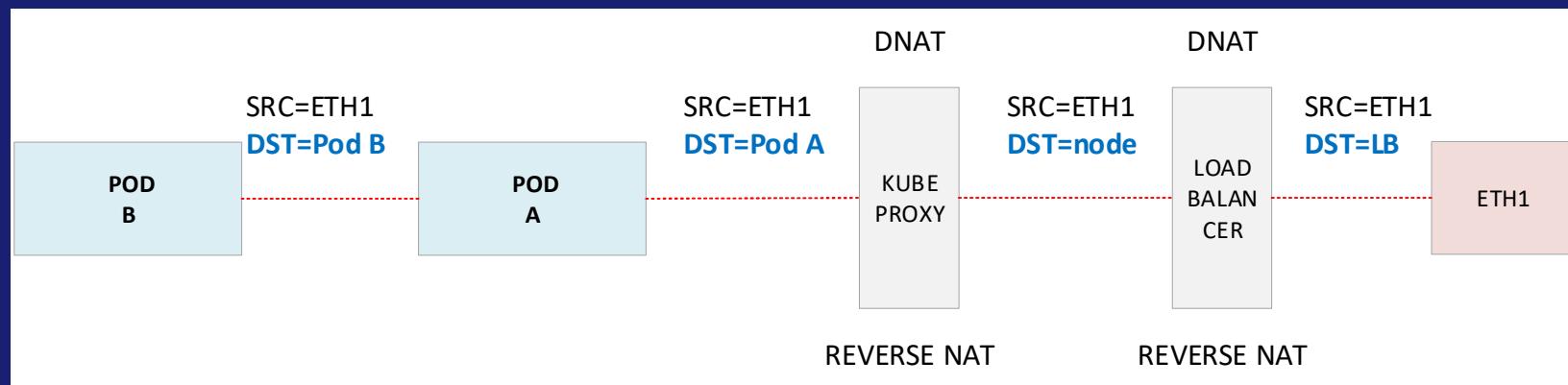
2025-03-16

설명

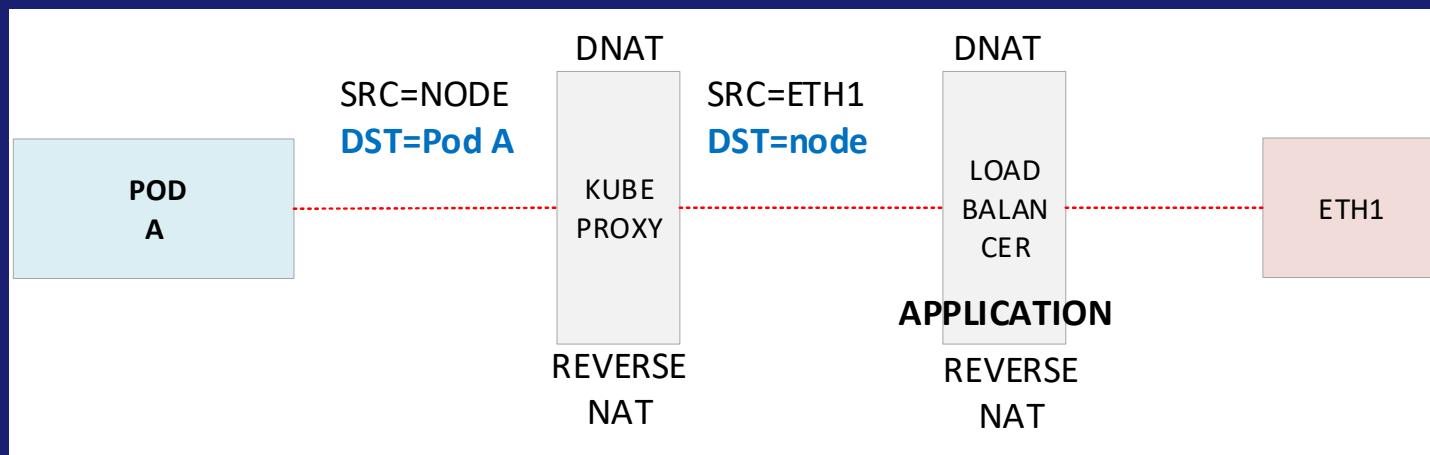


2025-03-16

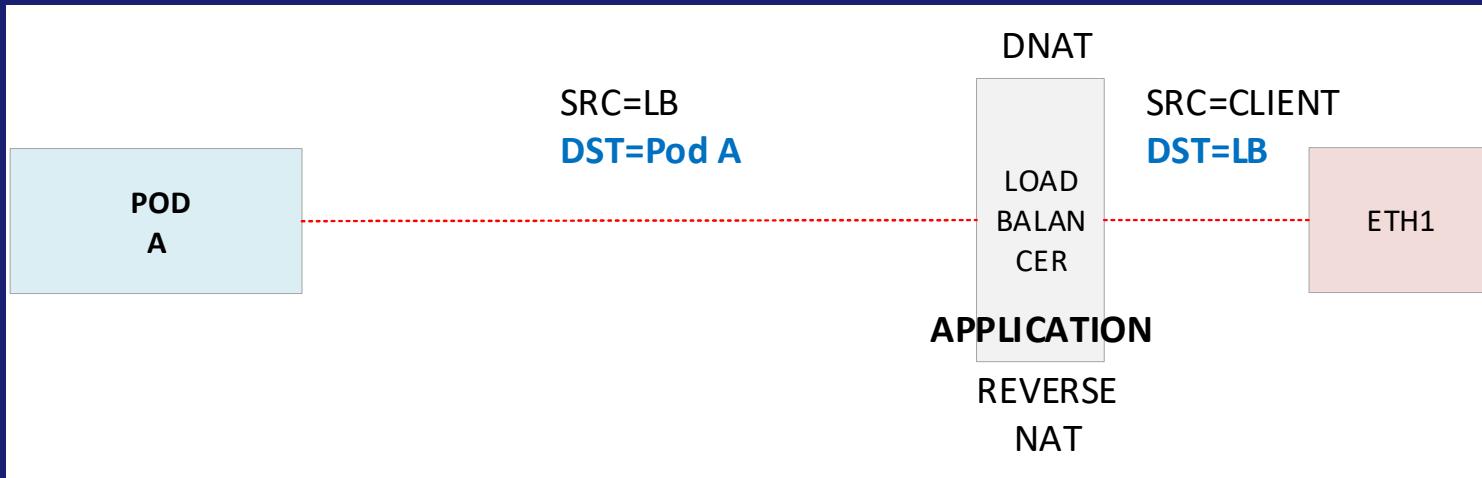
설명



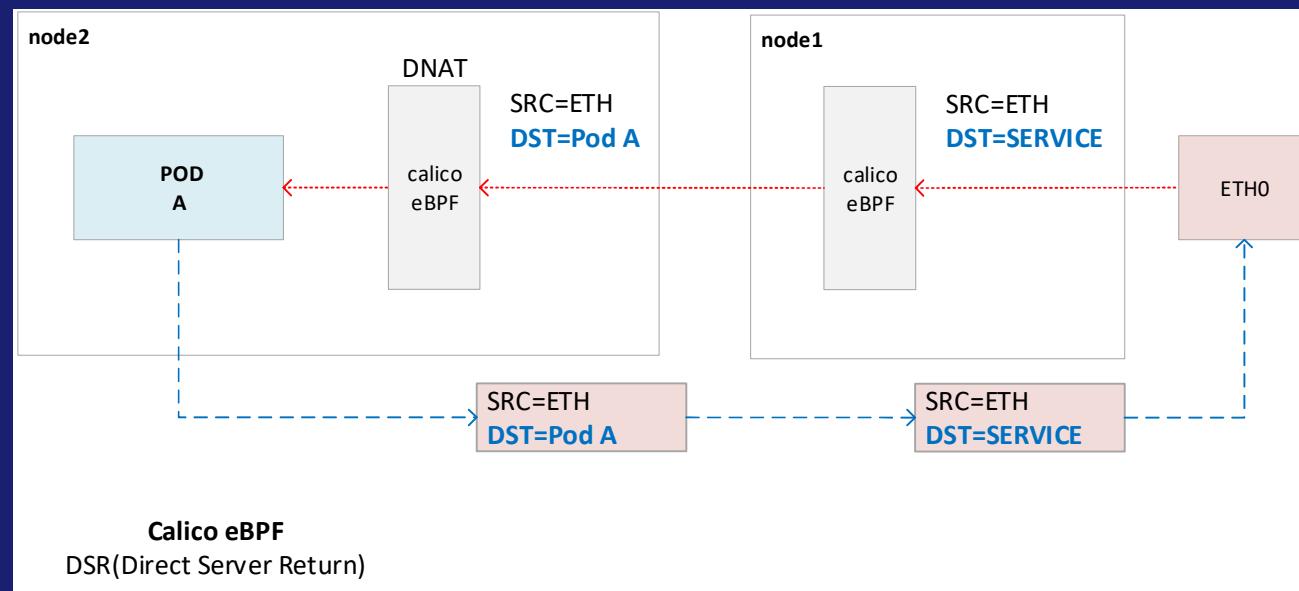
설명



설명



설명



CLUSTERIP

클러스터 아이피는 다음과 같이 구성이 가능하다.

```
# vi my-nginx-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
```

CLUSTERIP

위의 내용 계속…

```
ports:  
- port: 80  
  protocol: TCP  
  targetPort: 8080  
  
selector:  
  run: my-nginx  
  
# kubectl apply -f my-nginx-service.yaml
```

EXPOSE(ClusterIP 명시)

```
# vi service-clusterip.yaml

apiVersion: v1
kind: Service
metadata:
  name: my-nginx-multisvc
  labels:
    run: my-nginx
spec:
  ports:
  - name: normal
    port: 80
    protocol: TCP
    targetPort: 8080
```

EXPOSE(ClusterIP 명시)

```
- name: secure
  port: 82
  protocol: TCP
  targetPort: 8082
- name: monitoring
  port: 83
  protocol: TCP
  targetPort: 8083
  clusterIP: 10.0.23.24
  type: NodePort
  selector:
    run: my-nginx
```

NODEPORT

노드 포트는 다음과 같이 구성한다.

```
# kubectl create service nodeport --tcp=8080:80 --node-port=30013 -o yaml --dry-run=client test-httppd

apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
labels:
  app: test-httppd
name: test-httppd
```

NODEPORT

```
spec:  
  ports:  
    - name: 8080-80  
      nodePort: 30013  
      port: 8080  
      protocol: TCP  
      targetPort: 80  
  selector:  
    app: test-nginx  
  type: NodePort  
  
status:  
  loadBalancer: {}
```

MULTIPOINT

한 개 이상의 아이피 노출이 필요한 경우, 아래와 같이 가능하다.

```
# vi multipoint-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-multisvc
  labels:
    run: my-nginx
```

MULTI PORT

위의 내용 계속…

spec:

ports:

- name: normal

port: 80

protocol: TCP

targetPort: 8080

MULTIPOINT

위의 내용 계속…

```
- name: secure
  port: 82
  protocol: TCP
  targetPort: 8082
- name: monitoring
  port: 83
  protocol: TCP
  targetPort: 8083
  selector:
    run: my-nginx
```

명령어(EXPOSE)

명령어로 처리가 가능하다.

```
# kubectl create deployment my-deployment --image=nginx --port=8080
deployment.apps/my-deployment created

# kubectl expose deployment my-deployment --port=80 --target-port=8080 --
name=my-service
service/my-service exposed

# kubectl get svc my-service
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)      AGE
my-service     ClusterIP   10.96.123.45   <none>       80/TCP      2m
```

ENDPOINT

기본 명령어

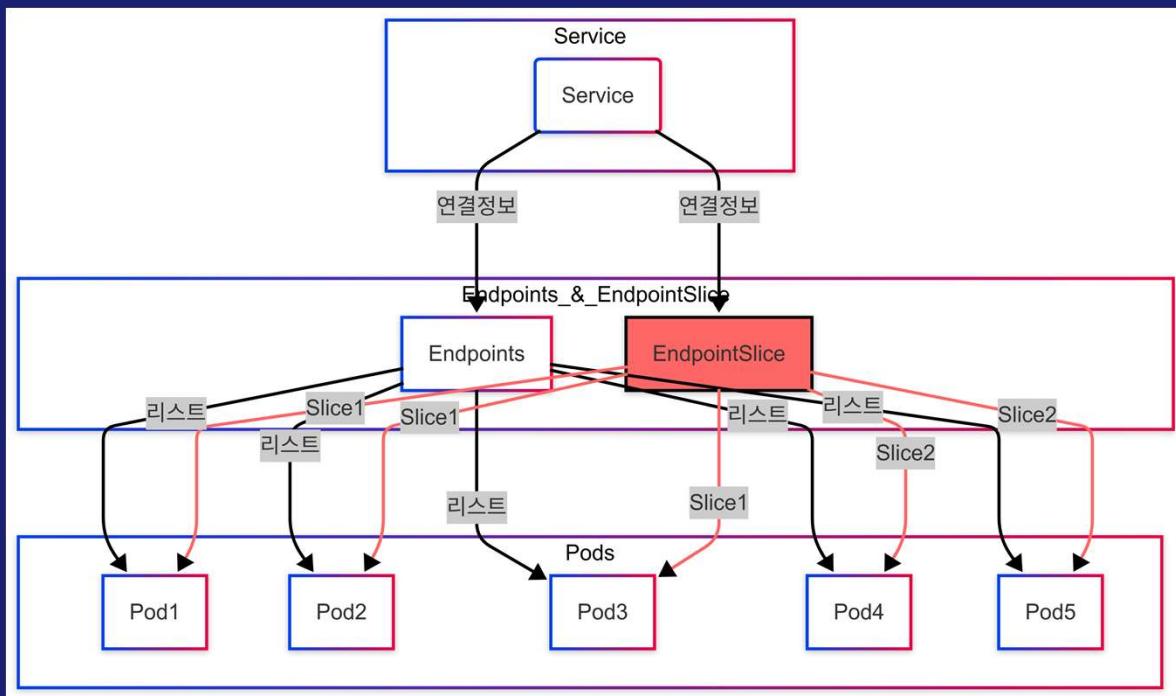
2025-03-16

설명

엔드포인트는 아래와 같이 지원한다.

항목	Endpoints	EndpointSlice
역할	Service에 매핑된 Pod의 IP 주소와 포트를 집계하여 단일 리소스로 관리	Service에 매핑된 Pod들을 여러 Slice로 분할하여 관리, 확장성과 성능 개선
구현 시기	초기 Kubernetes 버전부터 사용	Kubernetes 1.17부터 도입되어 점진적으로 대체 중
스케일링	대규모 서비스의 경우 단일 Endpoints 리소스에 너무 많은 데이터가 포함될 수 있음	Endpoints를 여러 개의 Slice로 나누어, 한 Slice당 항목 수 제한으로 성능 및 확장성 개선
확장성 및 기능성	기본 기능 제공, 추가 기능은 제한적	네트워크, 라벨, 조건 등 추가 정보를 포함할 수 있어 유연한 확장이 가능
사용 예시	소규모 서비스나 기본 Service-Pod 연결	대규모 클러스터, 고속 서비스 검색, 분산 서비스 환경 등

설명



2025-03-16

ENDPOINT

아래와 같이 POD를 생성한다.

```
# kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
# kubectl scale deployment nginx --replicas=10
deployment.apps/nginx scaled
```

ENDPOINT

위의 내용 이어서 계속…

```
# kubectl expose deployment nginx --port=80 --target-port=80 \
--name=nginx-service
service/nginx-service exposed
# kubectl get endpoints nginx-service
NAME           ENDPOINTS
nginx-service   10.244.1.2:80,10.244.1.3:80,10.244.1.4:80,10.244.1.5:80+
```

ENDPOINT

엔드포인트는 다음과 같이 구성이 가능하다.

```
# vi ep-my-service.yaml
apiVersion: v1
kind: Endpoints
metadata:
  name: my-service
```

ENDPOINT

계속 이어서…

```
subsets:  
  - addresses:  
    - ip: 10.0.0.1  
    - ip: 10.0.0.2  
  ports:  
    - port: 80  
# kubectl apply -f ep-my-service.yaml
```

ENDPOINT SLICE

엔드포인트 슬라이스 테스트는 조금 복잡하다. 단일 클러스터에서 최대한 비슷하게 구현해서 확인한다.

```
# kubectl get pod,deploy  
# kubectl get endpoints nginx-svc -o yaml  
# kubectl get endpointslices -l kubernetes.io/service-name=nginx-svc -o yaml
```

ENDPOINT SLICE

엔드포인트 슬라이스 테스트는 조금 복잡하다. 단일 클러스터에서 최대한 비슷하게 구현해서 확인한다.

```
# vi nginx-node4.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-node4
```

ENDPOINT SLICE

위의 내용 계속…

```
spec:  
  replicas: 10  
  selector:  
    matchLabels:  
      app: nginx-node4  
  template:  
    metadata:  
      labels:  
        app: nginx-node4
```

ENDPOINT SLICE

위의 내용 계속…

```
spec:  
  nodeSelector:  
    kubernetes.io/hostname: node4  
  containers:  
    - name: nginx  
      image: nginx  
  ports:  
    - containerPort: 80  
  
# kubectl apply -f nginx-node4.yaml  
deployment.apps/nginx-node4 created
```

ENDPOINT SLICE

엔드포인트 슬라이스를 명령어로 확인한다.

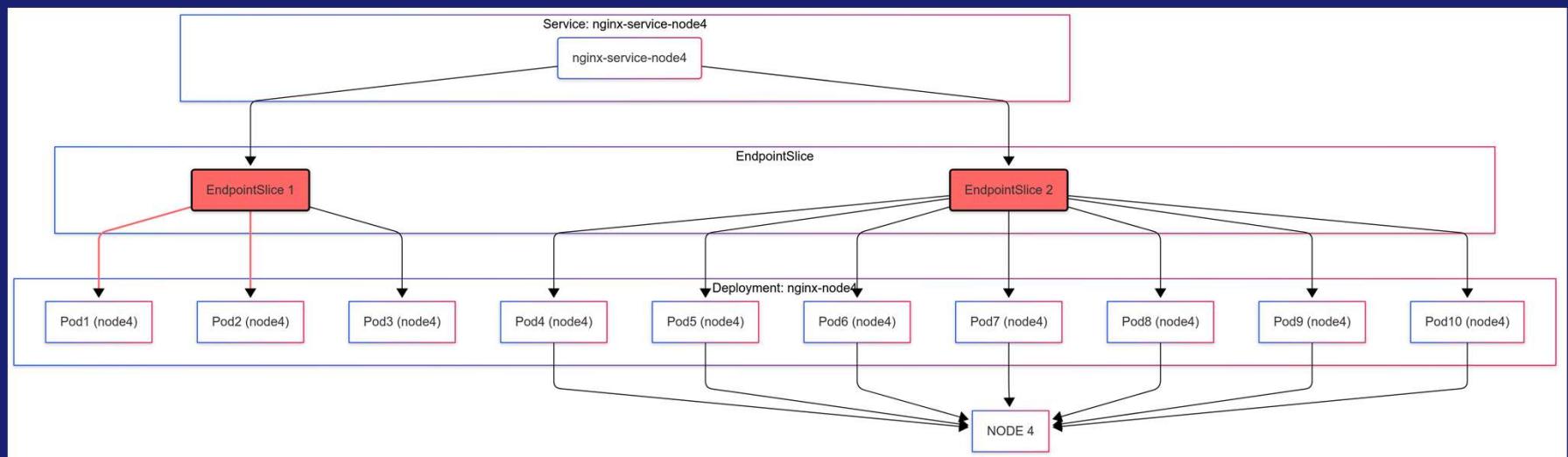
```
# kubectl expose deployment nginx-node4 --port=80 --target-port=80 --  
name=nginx-service-node4  
service/nginx-service-node4 exposed
```

ENDPOINT SLICE

엔드포인트 슬라이스를 명령어로 확인한다.

# kubectl get endpointslice -o wide --selector kubernetes.io/service-name=nginx-service-node4	NAME	ENDPOINTS	AGE	LABELS
	nginx-service-node4-abcde	10.244.3.21:80+	1m	kubernetes.io/service-name=nginx-service-node4
	nginx-service-node4-fghij	10.244.3.24:80+	1m	kubernetes.io/service-name=nginx-service-node4

ENDPOINT SLICE



2025-03-16

ENDPOINT SLICE

YAML파일로 작성 시, 아래와 같이 가능하다. 아래 코드는 동작하지 않으니 참고만 한다.

```
# vi en-slice.yaml
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: cluster2-endpoint-httpd
  labels:
    kubernetes.io/service-name: cluster2-endpoint-httpd
  addressType: IPv4
```

ENDPOINT SLICE

위의 내용 계속 이어서…

ports:

- name: http
- protocol: TCP
- port: 80

endpoints:

- addresses: POD의 아이피를 명시한다.
 - "10.1.2.3"
 - "10.1.2.4"
 - "10.1.2.5"

ENDPOINT SLICE

위의 내용 계속 이어서…

conditions:

ready: true

hostname: nginx

Slice 적용 조건을 명시한다.

nodeName: node4.example.com

zone: Daegu

PROXY

기본 명령어

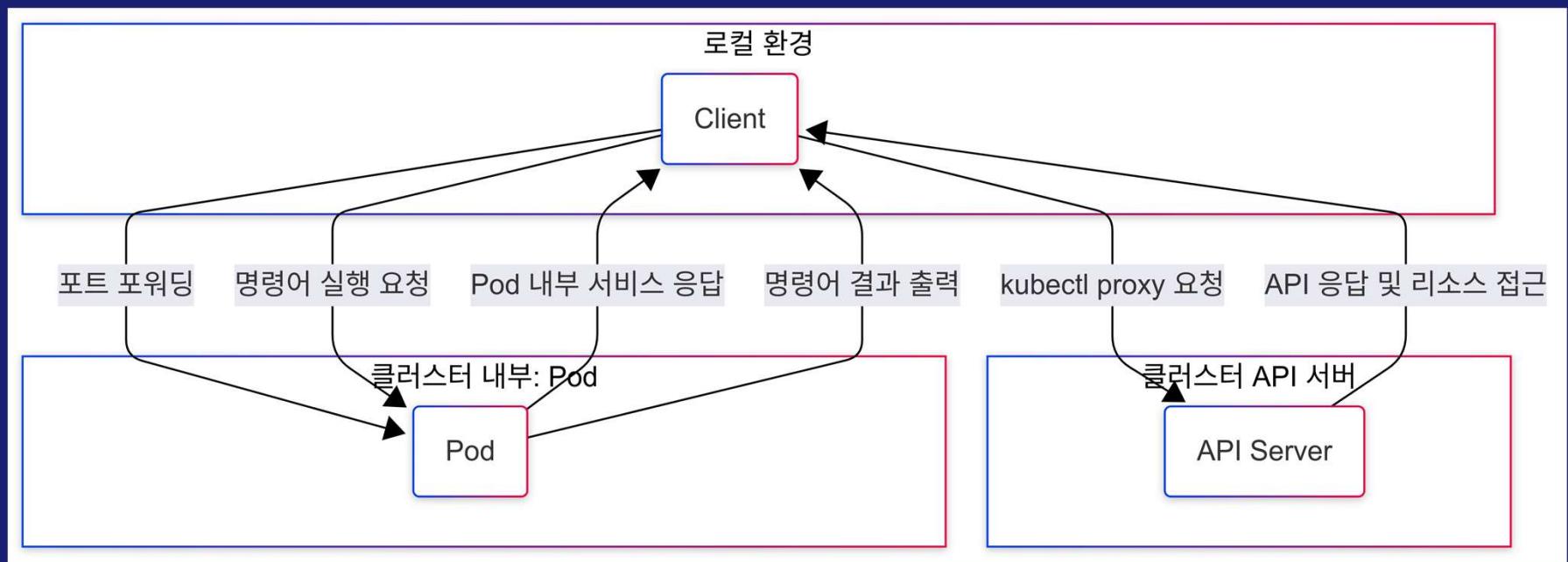
2025-03-16

설명

프록시 기능은 다음과 같다.

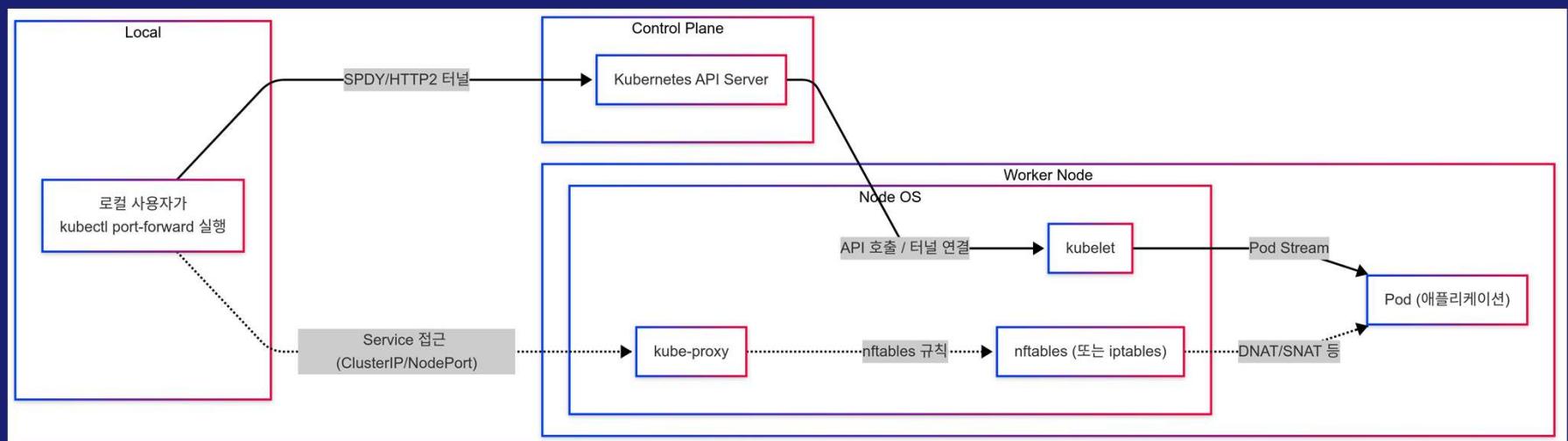
기능	설명	주요 사용 사례
kubectl proxy	로컬에서 API 서버를 프록시하여 클러스터 API 및 대시보드 등에 접근	클러스터 내부 API 호출, 대시보드 접속, 내부 리소스에 대한 RESTful 접근
kubectl port-forward	로컬 포트를 Pod의 특정 포트와 연결하여 직접 통신할 수 있도록 함	디버깅, 로컬 환경에서 Pod 내부 서비스 테스트, 비공개 애플리케이션 접근
kubectl exec	Pod 내부의 컨테이너에서 명령어를 실행하는 기능	문제해결 및 디버깅, 컨테이너 내부 상태 확인, 긴급 명령어 실행

설명



2025-03-16

설명



2025-03-16

PROXY

쿠버네티스 내부 서비스를 외부에서 접근이 가능하도록 프록시를 통해서 외부에 노출한다.

```
# kubectl proxy --port=8001
# curl localhost:8001
# curl localhost:8080/api/v1/namespaces/kube-system -H "Content-Type: application/json; charset=UTF-8"
```

PROXY

포트 포워딩 기능은 다음과 같이 사용한다. 프록시 서비스를 시작하면, 기본적으로 포트 번호는 8001으로 시작한다.

```
# kubectl proxy --port=8001
Starting to serve on 127.0.0.1:8001
# curl http://127.0.0.1:8001/api
{
  "kind": "APIVersions",
  "versions": [
    "v1",
    "v1beta1"
  ],
}
```

PROXY

포트번호 변경도 가능하다. 아래 명령어를 실행하면, 다음과 같은 결과가 출력이 된다.

```
# kubectl proxy --port=8080
Starting to serve on 127.0.0.1:8080
# curl localhost:8080/api/v1/namespaces/kube-system \
-H "Content-Type: application/json; charset=UTF-8"
```

PROXY

결과.

```
{  
  "kind": "Namespace",  
  "apiVersion": "v1",  
  "metadata": {  
    "name": "kube-system",  
    "uid": "2ec8b3f7-7bd2-4212-bd09-2b304fa3d657",  
    "resourceVersion": "14",  
    "creationTimestamp": "2025-03-09T06:19:25Z"  
  },  
  "spec": {  
    "finalizers": [  
      "kubernetes"  
    ]  
  },  
  "status": {  
    "phase": "Active"  
  }  
}
```

2025-03-16

225

PORT-FORWARD

포트 포워딩 기능은 다음과 같이 사용한다.

```
# kubectl port-forward pod/nginx
# kubectl port-forward pod/nginx --address 0.0.0.0 80:8080 pod/nginx
# kubectl port-forward pod/nginx --address=0.0.0.0,10.23.33.42 80:8080
pod/nginx
```

PORT-FORWARD

포트 포워딩 기능은 다음과 같이 사용한다. 터미널 한 개 이상을 열어서 확인한다.

```
# kubectl port-forward pod/nginx
# kubectl port-forward pod/nginx --address 0.0.0.0 80:8080 pod/nginx
# kubectl port-forward pod/nginx --address=0.0.0.0,10.23.33.42 80:8080
pod/nginx
```

LABEL

기본 명령어

2025-03-16

설명

레이블 및 셀렉터 쿠버네티스에서 핵심으로 사용하는 주요 기능이다. 이를 통해서 다양한 자원을 손쉽게 접근 및 관리가 가능하다. 레이블은 기본적으로 메타 영역(meta)에서 선언이 되며, 셀렉터는 사양 영역(sepc:)에서 선택이 된다.

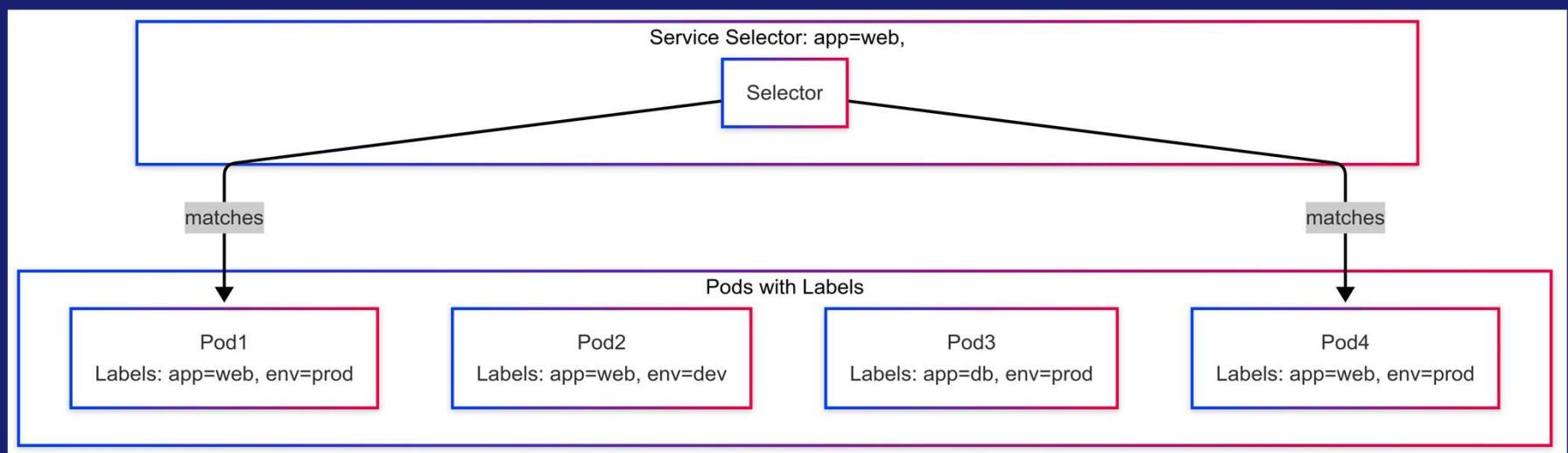
개념	설명
레이블(Label)	리소스에 붙이는 key=value 태그
셀렉터(Selector)	특정 레이블이 붙은 리소스를 선택하는 기능
matchLabels	단순한 키-값 일치 방식 (app=myapp)
matchExpressions	연산자 (in, notin, exists)를 활용한 선택

설명

레이블 기반으로 셀렉터는 생성하면, 이를 검색하기 위해서 matchLabels, matchExpressions를 사용해서 어떠한 자원을 찾을 지, 조건 검색이 가능하다. 보통 대다수 자원은 matchLabels를 통해서 자원을 검색하며, 조건에 따라서 좀 더 복잡하게 검색이 가능하다.

연산자 (Operator)	설명	사용 예시
matchLabels	단순한 키-값 일치 방식 (=)	app=myapp -> app: myapp
In	특정 값 하나 이상 포함	env in (production, staging)
NotIn	특정 값 제외	envnotin (development)
Exists	특정 키가 존재하면 선택	zone exists
DoesNotExist	특정 키가 없으면 선택	region doesnotexist

설명



2025-03-16

LABEL

레이블은 아래와 같이 생성 및 구성한다. 레이블은 메타 정보에 구성 혹은 kubectl 명령어로 선언이 가능하다.

```
metadata:  
  name: my-nginx  
  
labels:  
  run: my-nginx  
  
# kubectl label pods my-nginx-5b56ccd65f-vgn5b type=test  
# kubectl label pods my-nginx 5b56ccd65f-vgn5b run=test --overwrite  
# kubectl label namespaces test type=demo
```

LABEL

레이블은 쿠버네티스의 대다수 자원에서 명시가 가능하다.

```
# kubectl describe namespace test | grep Labels
Labels: type=demo

# kubectl label deployment/my-app type=demo
# kubectl describe deployment/my-app
labels:
  type=demo
# kubectl describe pod/my-app
labels:
  app=my-app
```

레이블 및 셀렉터*

레이블 기반으로 POD를 생성하면 다음과 같다. 두 개의 POD를 생성한다. 각 POD는 레이블을 가지고 있다.

```
# vi pod-labels1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod1
  labels:
    app: myapp
    env: production
spec:
  containers:
    - name: nginx
      image: nginx
```

레이블 및 셀렉터*

앞에서 이어서…

```
# vi pod-labels2.yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod2
  labels:
    app: myapp
    env: staging
spec:
  containers:
    - name: nginx
      image: nginx
```

레이블 및 셀렉터*

위의 내용을 다음 명령어로 조회한다.

```
# kubectl apply -f pod-labels1.yaml
# kubectl apply -f pod-labels2.yaml
# kubectl get pods -l app=myapp
```

NAME	READY	STATUS	RESTARTS	AGE
myapp-pod1	1/1	Running	0	10s
myapp-pod2	1/1	Running	0	10s

이 POD만 서비스에 연결하기 위해서 서비스 파일에 다음과 같이 선언한다.

레이블 및 셀렉터*

이 POD만 서비스에 연결하기 위해서 서비스 파일에 다음과 같이 선언한다.

```
# vi myapp-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    app: myapp    # 이 레이블이 있는 파드만 선택
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

matchExpression

이 기능을 사용하면 조건문 표현 방식은 살짝 달라진다. 아래는 "production"이라는 조건으로 접근한다.

```
spec:  
  selector:  
    matchExpressions:  
      - key: env  
        operator: In  
        values: ["production"]
```

matchExpression

아래 조건은 "staging"에 접근한다.

```
spec:  
  selector:  
    matchExpressions:  
      - key: env  
        operator: NotIn  
        values: ["staging"]
```

matchExpression

아래 조건은 "staging"에 접근한다.

```
spec:  
  selector:  
    matchExpressions:  
      - key: env  
        operator: NotIn  
        values: ["staging"]
```

MatchLabels

위의 조각코드 기반으로 다음과 같이 전체 코드 작성이 가능하다. `matchLabels`는 단순하게 적용이 가능하다.

```
# vi matchLabels-myapp-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    matchLabels:
      app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
# kubectl apply -f matchLabels-myapp-service.yaml
# kubectl get -f matchLabels-myapp-service.yaml
```

MatchExpressions

하지만, "matchExpressions"는 operator라는 조건식이 들어가기 때문에, 하나의 키워드에 두 개의 조건 적용이 가능하다.

```
# vi matchexpress-myapp-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    matchExpressions:
      - key: env
        operator: In
        values: ["production", "staging"]
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
# kubectl apply -f matchexpress-myapp-service.yaml
# kubectl get -f matchexpress-myapp-service.yaml
```

SET/PATCH/EDIT

기본 명령어

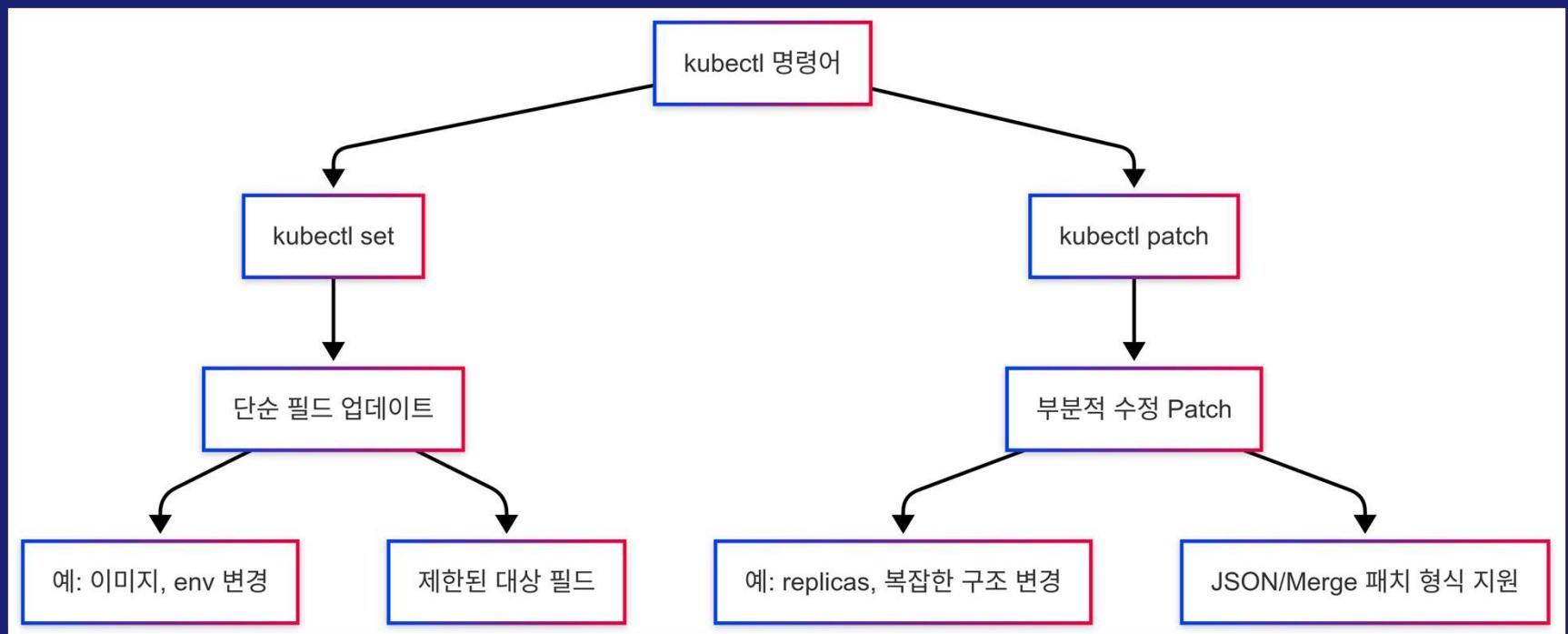
2025-03-16

설명

쿠버네티스에서 정보 변경을 위해서 보통 아래와 같은 명령어를 사용한다.

명령어	기능 및 목적	사용 예시	특징 및 주의사항
kubectl set	특정 필드(주로 이미지, 환경 변수 등)를 간단하게 업데이트	<code>kubectl set image deployment/nginx nginx=nginx:1.16</code>	단순 변경에 적합하며, 일부 필드에 국한됨
kubectl patch	리소스의 일부를 JSON 또는 YAML 포맷의 패치로 수정	<code>kubectl patch deployment/nginx -p '{"spec": {"replicas": 3}}'</code>	복잡한 구조의 부분 수정이 가능하며, 다양한 패치 방식 지원

설명



2025-03-16

SET

이미지 정보를 수정 시, set명령어를 통해서 변경이 가능하다.

```
# kubectl run --image=nginx my-httpd  
  
# kubectl get pod  
  
my-httpd  
  
# kubectl set image pod/my-httpd my-httpd=image:alpine  
# kubectl set image pod/my-httpd my-httpd=docker.io/library/httpd:latest  
# kubectl describe pods/my-httpd  
  
Image: docker.io/library/httpd:latest  
# kubectl set label deployment my-deployment env=prod  
deployment.apps/my-deployment labeled
```

PATCH

특정 자원을 변경하기 위해서 PATCH로 변경이 가능하다. set명령어와 다르게 대다수 영역에 적용이 가능하다.

```
# kubectl run --image=nginx --port=80 patch-nginx-pod  
# kubectl expose pod/patch-nginx-pod --type=NodePort  
# kubectl get svc  
patch-nginx-pod, NodePort,  
# kubectl patch svc patch-nginx-pod \  
-p '{"spec":{"externalIPs":["192.168.0.194"]}}'
```

PATCH

```
# kubectl get svc  
patch-nginx-pod, NodePort, 10.90.224.183, 192.168.0.194  
# kubectl patch svc patch-nginx-pod -p '{"spec": {"type": "LoadBalancer"}}'  
# kubectl get svc  
patch-nginx-pod, LoadBalancer, 10.90.224.183, 192.168.0.194
```

EDIT

위의 사용 방법이 어려운 경우 에디터 기반으로 수정이 가능하다.

```
# kubectl edit deployment my-httdp  
# EDITOR=nano kubectl edit deployment my-httdp  
# KUBE_EDITOR=nano kubectl edit deployment my-httdp  
# echo "KUBE_EDITOR=nano" >> ~/.bash_profile
```

DEBUG

기본 명령어

2025-03-16

DEBUG/LOG

쿠버네티스에서 로그 및 디버깅하기 위해서 아래 하위 명령어 사용이 가능하다.

명령어	설명
debug	디버그 명령어는 노드 및 POD의 컨테이너에 직접적으로 상태 확인을 위해서 사용한다. 현재 사용하는 디버그는 노드의 루트 디스크 접근이 가능하기 때문에, 권한 설정에 따라서 노드의 루트 디스크 접근이 가능하다.
log	로그는 POD에서 발생한 기록 확인이 가능하다. POD기반으로 동작하는 cronjobs, daemonset과 같은 자원에서 발생 기록 확인도 가능하다.

DEBUG/LOG

```
# kubectl debug -it my-httppd-6796cbbc4c-lr9qg --image=busybox \
--target=my-httppd
```

Targeting container "my-httppd". If you don't see processes from this container it may be because the container runtime doesn't support this feature.

Defaulting debug container name to debugger-dvtvc.

If you don't see a command prompt, try pressing enter.

```
/ #
```

DEBUG/LOG

```
# kubectl debug node/node1.example.com -it \
--image=docker.io/library/busybox:latest
```

Warning: metadata.name: this is used in the Pod's hostname, which can result in surprising behavior; a DNS label is recommended: [must not contain dots]

If you don't see a command prompt, try pressing enter

```
> / #
```

DEBUG/LOG

```
# kubectl logs -f my-nginx-6796cbbc4c-lr9qg
```

AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 10.244.221.20. Set the 'ServerName' directive globally to suppress this message

AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 10.244.221.20. Set the 'ServerName' directive globally to suppress this message

[Sun Sep 26 13:54:00.609837 2021] [mpm_event:notice] [pid 1:tid 140142545593472] AH00489: Apache/2.4.49 (Unix) configured -- resuming normal operations

[Sun Sep 26 13:54:00.609935 2021] [core:notice] [pid 1:tid 140142545593472] AH00094: Command line: 'httpd -D FOREGROUND'

DELETE/DIFF/EXPLAN REPLACE/CP/EXEC

기본 명령어

2025-03-16

DELETE

```
# kubectl delete pod -l run=my-nginx
# kubectl delete service my-nginx
# kubectl delete pods --all
# kubectl delete all --all
# kubectl get pods
# kubectl delete -f my-nginx.yaml
```

DIFF

```
# kubectl diff -f apache-demo-create.yaml
> -  replicas: 2
> +  replicas: 1
```

EXPLAIN

```
# kubectl explain pod

KIND:     Pod
VERSION:  v1
DESCRIPTION:
  Pod is a collection of containers that can run on a host. This resource is
  created by clients and scheduled onto hosts.

FIELDS:
  apiVersion  <string>
    APIVersion defines the versioned schema of this representation of an
    object. Servers should convert recognized schemas to the latest internal
    value, and may reject unrecognized values. More info:
    https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources
```

REPLACE

```
# kubectl run --image=nginx -l app=test -o=yaml --dry-run=client replace-nginx-pod > replace-nginx-pod.yaml

# kubectl create -f replace-nginx-pod.yaml

metadata:
  name: my-nginx
  labels:
    type: test
    system: linux
```

REPLACE

```
# kubectl replace -f apache-demo-create.yaml  
# kubectl replace -f apache-demo-create.yaml --force  
# kubectl get pod  
> replace-nginx-pod, 16s
```

CP

```
# kubectl run --image=nginx nginx --port=80
# kubectl get pod
nginx
# kubectl describe pod/nginx
# kubectl exec -it nginx -- bash
# cat <<EOF> index.html
Hello This is my-nginx
EOF

# kubectl cp index.html nginx:/usr/share/nginx/html/index.html
```

EXEC

```
# kubectl exec my-nginx -- ls /usr/share/nginx/html  
# kubectl exec -it my-nginx bashf
```

확장 명령어

kubectl

2025-03-16

Helm

확장 명령어

설명

쿠버네티스 패키지 관리자는 보통 두 가지를 많이 활용한다.

1. Kustomize
2. Helm

현재는 기능 확장 및 서비스 패키징을 Kustomize보다는 HelmChart기반으로 더 많이 선호하고 있다.
Kustomize는 쿠버네티스 1.14버전부터 도입이 되었다.

Helm은 쿠버네티스 1.6버전부터 사용 되었다. Helm v1/v2에서 2019년도 v3로 전환이 되면서 지금까지 사용하고 있다. v2와 v3는 생각보다 큰 차이는 없지만, 보안상 문제로 Tiller와 같은 기능을 제거하였다. Helm은 Kustomize와 제일 큰 차이점은 외부도구 및 명령어를 사용해야 한다.

설명

대표적인 패키지 관리자의 차이점은 다음과 같다.

측면	kubectl 기반 설치 (YAML/Kustomize)	Helm 기반 설치
장점	단순하고 선언적인 구성 방식	패키지화된 차트로 재사용성 및 배포 자동화
	별도의 도구 설치 필요 없음	템플릿을 통한 커스터마이징 용이
	기본 기능에 충실히	롤백 및 버전 관리 지원
단점	복잡한 설치 및 배포 구성은 어려움 의존성 및 구성이 번거로울 수 있음 릴리즈 추적이 되지 않음	Helm 차트의 학습 곡선 존재 차트 유지 관리 및 업데이트가 복잡할 수 있음

설명

위 패키지 관리자 말고 다음과 같은 패키지 관리자도 지원한다. 현재는 대다수 서비스가 오퍼레이터(operator)형식으로 변경하고 있다. 현재 쿠버네티스에서 지원하는 패키지 관리자는 다음과 같다.

도구	설명
Kustomize	YAML 오버레이 방식으로 리소스 구성을 재사용하고 수정할 수 있는 도구
	kubectl에 내장되어 있으며, 별도의 설치 없이 사용 가능
Kpt	구글에서 개발한 패키지 관리 도구로, 패키지의 버전 관리, 업데이트 및 재사용에 중점을 둠
	선언적이고 모듈화된 방식으로 애플리케이션 구성을 관리할 수 있음

설명

현재는 대다수 클러스터가 OLM과 Helm기반으로 구성이 된다.

- 클러스터: HelmChart
- 서비스: OLM

도구	설명
Carvel	(이전 k14s) 도구 모음으로, ytt(템플릿 도구), kapp(애플리케이션 배포 및 업데이트 도구), vendir(외부 의존성 관리) 등 다양한 유ти리티를 포함
	복잡한 애플리케이션 구성에 적합
Operator Lifecycle Manager	Operator 패키지 관리 도구로, 쿠버네티스 클러스터에서 Operator의 배포, 업그레이드 및 관리를 자동화
	Kubernetes Operator 기반 애플리케이션에 특화됨

HELM

쿠버네티스에 설치되는 어플라이언스 혹은 애플리케이션 패키지 관리자. 대신, 이 기능은 패키지 설치 기능만 있고, 힐링(Healing)기능은 없음.

이름	설명
repo	helm에서 제공하는 애플리케이션 설정파일 가지고 있는 저장소 위치. 최소 한 개의 저장소가 등록이 되어야, 서비스 확장 혹은 패키지 배포가 가능하다.
chart	helm에서 사용하는 패키지 이름. chart를 통해서 서비스나 혹은 프로그램을 패키징하여 클러스터에 배포 및 구성한다.

설치(HELM)

설치는 매우 간단하다. 먼저 Helm 명령어를 설치한다. 몇몇 배포판은 패키지로 지원하기도 한다.

```
# curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3  
| bash  
# helm version
```

앞에서 사용한 ingress, loadbalancer를 Helm기반으로 설치한다.

```
# helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx  
# helm repo update  
# helm install ingress-nginx ingress-nginx/ingress-nginx --namespace  
ingress-nginx --create-namespace  
# kubectl get pods -n ingress-nginx
```

NAME	READY	STATUS	RESTARTS
AGEingress-nginx-controller	1/1	Running	0

설치(INGRESS)

앞에서 사용한 ingress를 Helm기반으로 설치한다.

```
# helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
# helm repo update
# helm install ingress-nginx ingress-nginx/ingress-nginx --namespace
ingress-nginx --create-namespace
# kubectl get pods -n ingress-nginx
```

NAME	READY	STATUS	RESTARTS
AGEingress-nginx-controller	1/1	Running	0

2m

설치(METALLB)

LoadBalancer, MetalLB를 Helm기반으로 설치한다.

```
# helm repo add metallb https://metallb.github.io/metallb
# helm repo update
# helm install metallb metallb/metallb --namespace metallb-system --create-
namespace
```

설치(metallb)

Loadbalancer, MetalLB를 Helm기반으로 설치한다.

```
# vi metallb-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
      - 192.168.0.240-192.168.0.250
# kubectl apply -f metallb-config.yaml
```

설치(METALLB)

아래와 같이 구성이 되었는지 확인한다.

# kubectl get pods -n metallb-system				
NAME	READY	STATUS	RESTARTS	AGE
speaker	1/1	Running	0	2m
controller	1/1	Running	0	2m

자세한 구성 방법은 뒤에서 더 다루도록 한다.

미터링

확장 명령어

2025-03-16

설명

쿠버네티스에서 기본적으로 제공하는 미터링 서비스는 metrics라는 서비스가 유일하다. 하지만, 이 서비스는 모니터링 용도가 아닌, HPA, VPA와 같은 서비스를 활용하기 위해서 사용한다.

구성 요소	설명	용도 및 특징
Metrics Server	클러스터 내 노드와 Pod의 CPU, 메모리 사용량 등 리소스 메트릭을 집계하는 경량 서비스	Horizontal Pod Autoscaler(HPA)와 Vertical Pod Autoscaler(VPA) 등 자동 스케일링에 사용됨. 클러스터 내부에서 실시간 리소스 사용량 제공
kube-state-metrics	Kubernetes 오브젝트(Deployment, Pod, Node 등)의 상태 정보를 메트릭 형태로 제공	클러스터 상태 모니터링, 경보(Alerting) 및 시각화(Grafana 등)와 연동하여 클러스터 전반의 상태를 모니터링하는데 활용됨
cAdvisor	각 노드에서 컨테이너의 CPU, 메모리, 디스크 I/O 등 세부 리소스 사용량을 수집하는 에이전트	노드 단위의 컨테이너 메트릭을 수집하며, Metrics Server와 함께 사용되어 클러스터 전체의 리소스 사용량을 파악하는데 기여
Prometheus	오픈소스 모니터링 시스템으로, 다양한 메트릭(애플리케이션, 인프라, 클러스터 상태 등)을 스크랩하고 저장	Alerting 및 시각화(Grafana 연동) 기능 제공. Metrics Server, kube-state-metrics 등 다양한 데이터 소스를 통합하여 모니터링 환경 구축 가능

설명

metrics-server는 가급적이면 HelmChart기반으로 설치를 권장한다. 그렇지 않는 경우 버전관리가 어렵다.

설치

설치는 아래와 같이 진행한다. TLS키가 없는 경우 아래와 같은 방식으로 설치를 권장한다.

```
# helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-server/
# helm repo update
# helm install metrics-server metrics-server/metrics-server -n kube-system
--set args[0]="--kubelet-insecure-tls"
# kubectl get deployment metrics-server -n kube-system
```

설치

표준 설치는 아래와 같이 보통 가이드 한다. 이와 같은 방식으로 설치하면 올바르게 구성이 되지 않는다.

```
# helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-server/
# helm install metrics-server metrics-server/metrics-server \
--namespace=kube-system
# helm list -n kube-system
metrics-server
```

설치 및 수정

수동으로 TLS설정 부분 변경이 가능하다.

```
# kubectl edit -n kube-system deployments.apps metrics-server
spec:
  containers:
    - args:
        - --cert-dir=/tmp
        --secure-port=4443
        - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
        - --kubelet-use-node-status-port
        - --kubelet-insecure-tls
```

미터링

HelmChart Value에 적용하는 경우, 아래와 같이 적용이 가능하다.

```
# vi values.yaml
defaultArgs:
  - --cert-dir=/tmp
  - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
  - --kubelet-use-node-status-port
  - --metric-resolution=15s
  - --kubelet-insecure-tls
# helm upgrade --install metrics-server metrics-server/metrics-server --
  namespace=kube-system -f values.yaml
```

2025-03-16

미터링

설치 혹은 환경에 따라서 설치 방법이 다르다. 미터링 환경이 큰 경우(HPA/VPA), H/A기능 기반으로 Metrics 서비스를 구성해야 한다.

형식	조정
HelmChart	Replicas: 1에서 이 이상 값으로 values.yaml를 수정한다.
kubectl	1.12+경우에는 kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/high-availability-1.21+.yaml 명령어로 설치가 가능하다.

미터링

문제 없이 구성이 되면 보통 다음과 같이 화면에 출력이 된다.

```
# kubectl get pods -n kube-system -l k8s-app=metrics-server
metrics-server-fb889bc84-22zkc    1/1      Running   0          4m21s
# kubectl top nodes
node1.example.com    56m           2%        1633Mi       45%
```

사용자 생성 및 구성

확장 명령어

2025-03-16

사용자 생성 및 구성

위의 설명 기반으로 사용자를 생성한다.

```
# kubectl create namespace user1-namespace
namespace/user1-namespace created
# openssl genrsa -out user1.key 2048
# openssl req -new -key user1.key -out user1.csr -subj "/CN=user1/O=group1"
# openssl x509 -req -in user1.csr -CA /etc/kubernetes/pki/ca.crt -CAkey
/etc/kubernetes/pki/ca.key \ -CAcreateserial -out user1.crt -days 365
```

사용자 생성 및 구성

등록된 사용자를 인증서에 등록한다.

```
# kubectl config set-credentials user1 --client-certificate=user1.crt \
--client-key=user1.keykubectl config set-context user1-context \
--cluster=kubernetes --namespace=user1-namespace --user=user1
User "user1" set.Context "user1-context" created.
# kubectl get csr
user1, pending
# kubectl certificate approve <USERNAME>
# kubectl get csr
user1, Approved,Issued
```

RBAC 기반으로 자원 관리 및 접근제한

확장 명령어

2025-03-16

ROLE구성

생성된 사용자를 사용하기 위해서 자원에 할당해야 한다. 자원에 할당하기 위해서는 다음과 같이 작업을 수행한다.

```
# vi user1-role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: user1-namespace
  name: user1-role
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "create", "delete"]
# kubectl apply -f user1-role.yaml
role.rbac.authorization.k8s.io/user1-role created
```

ROLE구성

사용자와 그리고 생성된 네임스페이스를 연결한다. 이때, roleBinding를 사용해서 구성한다.

```
# vi user1-roleBinding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  namespace: user1-namespace
  name: user1-rolebinding
subjects:
- kind: User
  name: user1
  apiGroup: rbac.authorization.k8s.io
```

ROLE구성

앞에 내용 계속 이어서…

```
roleRef:  
  kind: Role  
  name: user1-role  
  apiGroup: rbac.authorization.k8s.io  
# kubectl apply -f user1-roleBinding.yaml  
rolebinding.rbac.authorization.k8s.io/user1-rolebinding created
```

확인하기

생성된 사용자 및 네임스페이스 자원이 올바르게 바인딩이 되었는지 확인한다. 앞에서 만든 RBAC는 POD만 허용하였다.

```
# kubectl --context=user1-context get pods
No resources found in user1-namespace namespace.
# kubectl --context=user1-context get deployments
Error from server (Forbidden): deployments.apps is forbidden: User "user1"
cannot list resource "deployments" in API group "apps" in the namespace
"user1-namespace"
```

확장

기존 내용은 POD만 허용했기 때문에, user1-namespace의 모든 자원에 접근이 가능하도록 수정한다. 아래와 같이 내용을 수정한다.

```
# vi user1-role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: user1-namespace
  name: user1-role
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
```

확장

자원 접근이 올바르게 되는지 적용 및 확인한다.

```
# kubectl apply -f user1-role.yaml
role.rbac.authorization.k8s.io/user1-role configured
# kubectl --context=user1-context get all -n user1-namespace
No resources found in user1-namespace namespace.
# kubectl --context=user1-context create deployment nginx --image=nginx -n user1-
namespace
deployment.apps/nginx created
# kubectl --context=user1-context expose deployment nginx --port=80 --target-port=80
--type=ClusterIP -n user1-namespace
service/nginx exposed
```

결과

결과는 다음과 같이 출력이 된다.

```
# kubectl get pod,svc,deployment,rs
NAME                                     READY   STATUS    RESTARTS   AGE
pod/nginx-6799fc88d8-xxxxx              1/1     Running   0          10s
NAME           TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
service/nginx   ClusterIP   10.96.12.34   <none>        80/TCP      5s
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx   1/1       1           1           10s
NAME           DESIRED  CURRENT    READY   AGE
replicaset.apps/nginx-6799fc88d8     1         1         1         10s
```

KUBERNETES-ADMIN

쿠버네티스에서 기본적으로 제공하는 계정은 다음과 같다. 매뉴얼 기준으로 바닐라 버전은 다음과 같이 구성이 되어 있다.

역할이름	역할연결	설명
cluster-admin	system:masters group	Allows super-user access to perform any action on any resource. When used in a ClusterRoleBinding, it gives full control over every resource in the cluster and in all namespaces. When used in a RoleBinding, it gives full control over every resource in the role binding's namespace, including the namespace itself.
admin	None	Allows admin access, intended to be granted within a namespace using a RoleBinding. If used in a RoleBinding, allows read/write access to most resources in a namespace, including the ability to create roles and role bindings within the namespace. This role does not allow write access to resource quota or to the namespace itself. This role also does not allow write access to EndpointSlices (or Endpoints) in clusters created using Kubernetes v1.22+. More information is available in the " "Write Access for EndpointSlices and Endpoints" section ".

KUBERNETES-ADMIN

앞에 내용을 이어서…

edit	None	<p>Allows read/write access to most objects in a namespace. This role does not allow viewing or modifying roles or role bindings. However, this role allows accessing Secrets and running Pods as any ServiceAccount in the namespace, so it can be used to gain the API access levels of any ServiceAccount in the namespace. This role also does not allow write access to EndpointSlices (or Endpoints) in clusters created using Kubernetes v1.22+. More information is available in the "Write Access for EndpointSlices and Endpoints" section.</p>
view	None	<p>Allows read-only access to see most objects in a namespace. It does not allow viewing roles or role bindings. This role does not allow viewing Secrets, since reading the contents of Secrets enables access to ServiceAccount credentials in the namespace, which would allow API access as any ServiceAccount in the namespace (a form of privilege escalation).</p>

KUBERNETES-ADMIN

기존에 사용한 쿠버네티스 관리자 계정 kubernetes-admin계정을 k8s-admin으로 변경한다. 앞에서 사용한 명령어를 활용한다.

```
# openssl genrsa -out k8s-admin.key 2048
# openssl req -new -key k8s-admin.key -out k8s-admin.csr -subj "/CN=k8s-
admin/O=system:masters"
# openssl x509 -req -in k8s-admin.csr -CA /etc/kubernetes/pki/ca.crt -CAkey
/etc/kubernetes/pki/ca.key -CAcreateserial -out k8s-admin.crt -days 365
# kubectl config set-cluster kubernetes \
--certificate-authority=/etc/kubernetes/pki/ca.crt \
--server=https://<APISERVER>:6443 \
--embed-certs=true \
--kubeconfig=k8s-admin.kubeconfig
```

KUBERNETES-ADMIN

새로운 관리자 계정이 생성이 되면, 역할을 할당 및 구성한다.

```
# kubectl config set-credentials k8s-admin \
--client-certificate=k8s-admin.crt \
--client-key=k8s-admin.key \
--embed-certs=true \
--kubeconfig=k8s-admin.kubeconfig
# kubectl config set-context k8s-admin@kubernetes \
--cluster=kubernetes \
--user=k8s-admin \
--kubeconfig=k8s-admin.kubeconfig
# kubectl config use-context k8s-admin@kubernetes --kubeconfig=k8s-admin.kubeconfig
# kubectl create clusterrolebinding k8s-admin-binding --clusterrole=cluster-admin --
user=k8s-admin
clusterrolebinding.rbac.authorization.k8s.io/k8s-admin-binding created
```

KUBERNETES-ADMIN

기존 계정의 권한을 제거하고, 새로운 계정으로 권한을 이전한다.

```
# kubectl delete clusterrolebinding kubernetes-admin
clusterrolebinding.rbac.authorization.k8s.io "kubernetes-admin" deleted
# kubectl create clusterrolebinding k8s-admin-binding --clusterrole=cluster-admin --
user=k8s-admin
clusterrolebinding.rbac.authorization.k8s.io/k8s-admin-binding created
# kubectl get clusterrolebinding | grep admin
```

서비스 계정

확장 명령어

2025-03-16

설명

서비스 계정은 직접적으로 자원에 할당 및 접근하는 게 아니라, 특정 계정을 통해서 작업을 대신 수행한다. 서비스 계정은 다음과 같이 기능을 제공한다.

기능	설명
Pod에 인증 정보 제공	Pod가 쿠버네티스 API 서버와 상호작용할 때 사용하는 계정
RBAC(Role-Based Access Control)과 연동 가능	특정 역할(Role)을 할당하여 접근 권한을 제한 가능
네임스페이스별로 관리 가능	네임스페이스 단위로 서비스 계정을 생성하고 관리 가능
자동 토큰 생성 및 마운트	기본적으로 토큰이 Secret 리소스로 생성되며, Pod가 실행될 때 자동으로 마운트됨
Pod가 API 요청 시 신뢰할 수 있는 ID 제공	Pod가 API 요청을 보낼 때, 해당 서비스 계정을 인증 정보로 사용

설명

서비스 계정과 다른 기능을 비교하면 다음과 같다.

기능	ServiceAccount	User Account	Role & RoleBinding
대상	Pod (애플리케이션)	사람 (관리자, 개발자 등)	사용자 및 서비스 계정
용도	Pod가 쿠버네티스 API에 접근할 때	사용자가 kubectl이나 API를 사용할 때	특정 리소스에 대한 접근 권한을 관리
기본 제공 여부	네임스페이스마다 default 서비스 계정이 있음	기본적으로 존재하지 않음 (외부 인증 필요)	RBAC 설정을 해야 함
권한 설정	Role, ClusterRole과 연계하여 권한 관리 가능	RBAC을 통해 권한 부여 가능	특정 리소스(예: Pod, Secret, Node 등)에 대한 권한 부여
токен 사용 여부	Secret을 통해 토큰 제공 (JWT 기반)	일반적으로 패스워드 또는 인증 프로バイ더 사용	자체적으로 토큰을 제공하지 않음
네임스페이스 제한 여부	네임스페이스 단위로 존재	클러스터 전역적으로 존재	Role은 네임스페이스 범위, ClusterRole은 클러스터 전체에 적용 가능

생성(CLI기반)

CLI기반으로 생성은 매우 간단하다. 아래와 같이 실행하면 NS/SA/POD가 생성이 된다.

```
# kubectl create namespace test-sa
# kubectl create serviceaccount test-sa-pod -n test-sa
# kubectl create role test-sa-role \
--verb=get,list,watch \
--resource=pods,services,configmaps,secrets \
--namespace=test-sa
# kubectl create rolebinding test-sa-role-binding \
--role=test-sa-role \
--serviceaccount=test-sa:test-sa-reader \
--namespace=test-sa
```

생성(CLI기반)

CLI기반으로 생성은 매우 간단하다. 아래와 같이 실행하면 NS/SA/POD가 생성이 된다.

```
# kubectl run nginx-pod --image=nginx --namespace=test-sa --  
serviceaccount=test-sa-pod --port=80  
# kubectl get namespace  
# kubectl get serviceaccount -n test-sa  
# kubectl get pod nginx-pod -n test-sa -o  
jsonpath='{.spec.serviceAccountName}'  
test-sa-pod
```

생성(NS)

네임스페이스를 YAML기반으로 생성한다.

```
# vi test-sa.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: test-sa
# kubectl apply -f my-service-account.yaml
```

생성(SA)

YAML기반으로 구성하면 다음과 같다. 서비스 계정을 생성한다.

```
# vi test-sa-pod.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: test-sa-pod
  namespace: test-sa
# kubectl apply -f test-sa-pod.yaml
```

생성(POD)

POD를 생성한다. POD는 nginx-pod라는 이름으로 생성한다.

```
# vi sa-nginx-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: test-sa
```

생성(POD)

앞에 내용 이어서...

```
spec:  
  serviceAccountName: test-sa-pod  
  containers:  
    - name: nginx  
      image: nginx  
    ports:  
      - containerPort: 80  
# kubectl apply -f test-sa-pod.yaml
```

생성(ROLE)

역할을 생성한다. 읽기 전용으로 구성한다.

```
# vi pod-reader.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: default
rules:
- apiGroups: []
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
# kubectl apply -f pod-reader.yaml
```

생성(ROLEBINDING)

역할을 네임스페이스와 연결한다. 이전과 다른 점은 ServiceAccount를 통해서 연결 한다.

```
# vi pod-reader-binding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: default
subjects:
- kind: ServiceAccount
  name: my-service-account
  namespace: default
```

생성(ROLEBINDING)

계속 이어서…

```
roleRef:  
  kind: Role  
  name: pod-reader  
  apiGroup: rbac.authorization.k8s.io  
# kubectl apply -f pod-reader-binding.yaml
```

SERVICE-ACCOUNT

쿠버네티스 서비스 계정은 다음과 같이 생성 및 구성한다.

```
# kubectl get sa
> default          0           15d

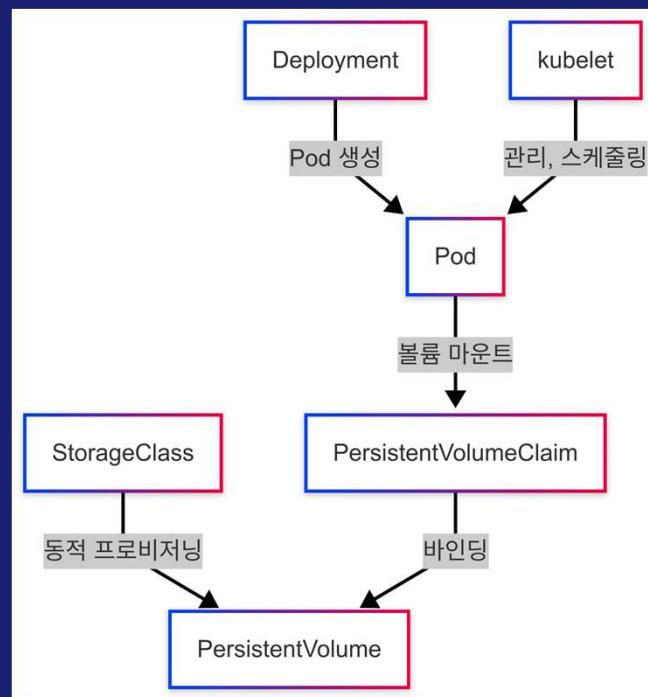
# kubectl create serviceaccount serviceaccount-example
# kubectl config use-context kubernetes-admin@kubernetes
# kubectl create namespace test-sa
# kubectl create serviceaccount test-sa --namespace=test-sa
# kubectl run test-sa-pod --namespace=openshift --image=nginx -o=yaml --
dry-run=client > test-sa-pod.yaml
```

스토리지

확장 명령어

2025-03-16

소개



소개

기능은 다음과 같다.

구성 요소	설명	주요 특징	사용 사례
StorageClass	동적 프로비저닝 시 사용할 스토리지의 속성(타입, 성능, 접근 모드 등)을 정의하는 리소스	PVC에 의해 참조됨 스토리지 정책 및 클래스 지정 다양한 백엔드지원	스토리지 자원 자동 생성, 정책 기반 관리
PersistentVolume	클러스터 관리자가 생성하거나 StorageClass를 통해 동적으로 프로비저닝되는 실제 스토리지 자원	클러스터 수준의 독립적 리소스 다양한 저장소 백엔드 지원 수동/동적 생성 가능	데이터 저장, 공유 스토리지 할당
PersistentVolumeClaim	애플리케이션에서 필요한 스토리지 용량과 접근 모드를 요청하는 리소스	PV와 바인딩 요구 사항 명시 Pod에 스토리지 할당 시 사용됨	Pod에 스토리지 할당, 데이터 보존 및 공유

설치

NFS서비스를 구성한다. 미리 앞서 구성한 경우, 이 부분은 넘어간다.

```
# dnf install nfs-utils -y
# firewall-cmd --add-service nfs
# firewall-cmd --add-service nfs --permanent
# setenforce 0
# getenforce
# mkdir -p /nfs
# cat <<EOF > /etc/exports
/nfs *(rw,no_root_squash)
EOF
# exportfs -avrs
# systemctl enable --now nfs-server
# semanage fcontext -a -t public_content_rw_t '/nfs(/.*)?'
# restorecon -RFvv /nfs
```

설치(NO VALUES)

NFS 기본 값으로 설치하는 과정. 랩에서는 이 과정으로 설치하지 않는다.

```
# helm repo add nfs-subdir-external-provisioner https://kubernetes-
sigs.github.io/nfs-subdir-external-provisioner/
# helm repo update
# helm install nfs-provisioner nfs-subdir-external-provisioner/nfs-subdir-external-
provisioner \
  --namespace nfs-provisioner --create-namespace \
  --set nfs.server=192.168.1.100 \
  --set nfs.path=/exported/path
NAME: nfs-provisioner
LAST DEPLOYED: Wed Mar  9 14:20:30 2025
NAMESPACE: nfs-provisioner
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

설치(NO VALUES)

아래 명령어로 올바르게 설치가 되었는지 확인한다.

```
# kubectl get pods -n nfs-provisioner
# kubectl get sc
NAME                      PROVISIONER          RECLAIMPOLICY   VOLUMEBINDINGMODE   LLOWVOLUMEEXPANSION   AGE
nfs-subdir-external-provisioner  cluster.local/    Delete          Immediate        true               3m
```

설치(VALUES)

Chart기반으로 role 및 서비스 계정 구성하는 경우 다음과 같이 values.yaml파일을 생성한다.

```
# vi values.yaml
serviceAccount:
  create: true
  name: nfs-provisioner-sa
  annotations: {} # 필요시 여기에 추가 주석 설정
rbac:
  create: true
  rules:
    - apiGroups: []
      resources: ["persistentvolumes"]
      verbs: ["get", "list", "watch", "create", "delete"]
```

설치(VALUES)

앞에 내용 계속 이어서…

```
- apiGroups: [""]
  resources: ["persistentvolumeclaims"]
  verbs: ["get", "list", "watch", "update"]
- apiGroups: ["storage.k8s.io"]
  resources: ["storageclasses"]
  verbs: ["get", "list", "watch"]
nfs:
  server: 192.168.1.100
  path: /exported/path
storageClass:
  name: nfs-storage
  reclaimPolicy: Delete
  volumeBindingMode: Immediate
```

설치

앞에 내용 계속 이어서…

```
nfs:  
  server: 192.168.1.100  
  path: /exported/path  
storageClass:  
  name: nfs-storage  
  reclaimPolicy: Delete  
  volumeBindingMode: Immediate
```

설치(VALUES)

values.yaml 파일 생성 후, 아래 명령어 CSI NFS 드라이버를 설치 및 구성한다.

```
# helm install nfs-provisioner nfs-subdir-external-provisioner/nfs-subdir-  
external-provisioner \  
--namespace nfs-provisioner --create-namespace -f values.yaml  
NAME: nfs-provisioner  
LAST DEPLOYED: Wed Mar  9 14:20:30 2025  
NAMESPACE: nfs-provisioner  
STATUS: deployed  
REVISION: 1  
TEST SUITE: None
```

설치(VALUES)

최종적으로 잘 구성이 되었는지 확인한다.

```
# kubectl get serviceaccount -n nfs-provisioner  
# kubectl get clusterrole,clusterrolebinding | grep nfs-subdir-external-  
provisioner  
# kubectl get sc
```

이 기반으로 PV/PVC를 구성 및 생성한다. 또한, SC를 통해서 PV없이 PVC를 생성한다.

KUBERNETES STORAGE

PV는 스토리지 클래스를 사용하지 않고 스토리지 서버 혹은 장비에 연결하는 일종의 드라이버 역할을 한다. 앞서 스토리지 클래스를 구성했기 때문에, PV를 통해서 PVC를 생성 및 구성한다.

구성 요소	설명	특징 및 사용 사례
PersistentVolume (PV)	클러스터 내에서 관리되는 실제 스토리지 자원입니다. 관리자가 생성하거나 동적 프로비저닝(StorageClass 사용)으로 생성됩니다.	- 클러스터 전역 자원 - NFS, iSCSI, 클라우드 스토리지 등 다양한 백엔드를 지원 - 관리자가 직접 생성할 수도 있음
PersistentVolumeClaim (PVC)	Pod가 스토리지 사용을 요청하는 리소스입니다. 사용자가 원하는 용량, 접근 모드, StorageClass 등의 요구사항을 명시하며, PV와 바인딩됩니다.	- Pod의 스토리지 요청을 추상화 - 자동으로 PV와 매칭되어 바인딩됨 - 애플리케이션 데이터 저장에 사용됨

KUBERNETES STORAGE

좀 더 쉽게 설명하면, 아래와 같이 비유가 가능하다.

용어	설명	비유
PVC (Persistent Volume Claim)	Pod가 필요한 스토리지를 요청하는 역할	"내가 사용할 디스크 주세요!"
PV (Persistent Volume)	실제 저장소 (관리자가 미리 만들어 놓음)	"여기 사용할 디스크 있어!"

PV(STORAGE PROVIDER)

NFS기반으로 연결하는 경우 다음과 같이 PV를 구성한다.

```
# vi pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
  labels:
    type: nfs
```

PV(STORAGE PROVIDER)

NFS기반으로 연결하는 경우 다음과 같이 PV를 구성한다.

```
spec:  
  storageClassName: ""  
  capacity:  
    storage: 1Gi  
  accessModes:  
    - ReadWriteMany  
  
nfs:  
  server: storage.example.com  
  path: "/nfs/manual-pv"  
# kubectl apply -f pv.yaml  
persistentvolume/static-pv created
```

PVC(VOLUME BINDING INTERFACE)

생성한 PV에 PVC를 연결한다.

```
# vi nfs-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  volumeName: nfs-pv  # 강제로 PV에 연결
```

PVC(VOLUME BINDING INTERFACE)

혹은 StorageClass를 통해서 구성 시, 아래와 같이 구성한다.

```
# vi pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: sc-nfs-pvc
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: nfs-storage
# kubectl apply -f pvc.yaml
persistentvolumeclaim/nfs-pvc created
```

PVC(VOLUME BINDING INTERFACE)

PVC가 구성이 되면 대략 아래와 같이 화면에 출력이 된다.

# kubectl get pvc nfs-pvc						
NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	
nfs-pvc	Bound	pvc-12345678-90ab-cdef-1234-	1Gi	RWX	nfs-storage	
sc-nfs-pvc	Bound	pvc-12345678-90ab-cdef-1234-	1Gi	RWX	nfs-storage	

PVC(DEPLOY)

위의 PVC기반으로 Deployment 구성 후 POD를 생성한다.

```
# vi nginx-with-nfs.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-with-nfs
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-nfs
```

PVC(DEPLOY)

이어서...

```
template:  
  metadata:  
    labels:  
      app: nginx-nfs  
spec:  
  containers:  
  - name: nginx  
    image: nginx:latest
```

PVC

POD에 연결할 볼륨 디스크를 선언 및 명시한다.

volumeMounts:

- name: nfs-volume
mountPath: /usr/share/nginx/html

volumes:

- name: nfs-volume
persistentVolumeClaim:
claimName: nfs-pvc

여기 원하는 PVC 이름을 명시한다.

```
# kubectl apply -f nginx-with-nfs.yaml  
deployment.apps/nginx-with-nfs created
```

스케일링 및 서비스

확장 명령어

2025-03-16

배포 전략

쿠버네티스 바닐라 버전 기반으로 배포전략은 다음과 같이 제공이 된다. 쿠버네티스 서비스 아키텍처는 무 중단 서비스를 구현이다.

배포 전략	설명	사용 예시
Rolling Update	기존 Pod를 순차적으로 교체 (무중단 배포)	<code>kubectl set image deployment/nginx nginx=nginx:latest</code>
Recreate	기존 Pod를 모두 삭제한 후 새로운 Pod를 생성	<code>kubectl rollout restart deployment/nginx</code>
Blue-Green Deployment	새 버전의 배포 그룹을 생성 후 트래픽을 전환	<code>kubectl apply -f blue-green.yaml</code>
Canary Deployment	일부 사용자만 새 버전을 사용하도록 설정	<code>kubectl apply -f canary.yaml</code>

배포 전략(lifecycle)

쿠버네티스 애플리케이션 종료는 보통 두 가지 방식이 있다.

- 런타임에서 종료(return, exit 0)
- 애플리케이션 종료 후, 컨테이너 종료(return, exit 0)

컨테이너 플랫폼에서는 한 개 이상의 애플리케이션들이 실행되기 때문에, 종종 컨테이너 런타임이 종료 신호 (SIGTERM)를 받으면 애플리케이션보다 빠르게 종료하는 경우가 있다. 이러한 이유로 종종 프로세스 상태가 D, Z으로 변경이 되는 경우가 있다. 이를 방지하기 위해서 다음과 같은 코드를 추가 하기도 한다.

```
lifecycle:  
  preStop:  
    exec:  
      command: ["/bin/sh", "-c", "sleep 10"]
```

배포 전략(PDB)

HPA와 같은 스케일링 기반으로 서비스가 구성되는 경우, 스케줄러가 모든 노드 상태를 실시간으로 알 수 없기 때문에 종종, 예상된 숫자보다 더 작게 스케일 아웃(Scale Out)이 되는 경우가 있다. 이러한 이유로, POD개수를 유지할 수 있도록 명시해야 한다.

쿠버네티스 버전 1.21에서 이러한 부분을 해결하기 위해서 PDB(PodDisruptionBudget) 기능이 추가가 되었다. 이 기능이 적용되는 다음과 같은 자원이다. 셀렉터 기반으로 동작한다.

- Deployment
- ReplicationController
- ReplicaSet
- StatefulSet

배포 전략(PDB)

이 기능을 적용하면 다음과 같다.

```
# vi nginx-pdb.yaml
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: nginx-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: nginx
# kubectl apply -f nginx-pdb.yaml
# kubectl describe -f nginx-pdb.yaml
```

HPA/VPA

확장 명령어

2025-03-16

HPA (DEPLOYMENT)

테스트를 위해서 아래와 같이 작성한다.

```
# vi scale-nginx.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: scale-nginx
  labels:
    app: scale-nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: scale-nginx
```

HPA (DEPLOYMENT)

위의 내용 계속…

```
template:
  metadata:
    labels:
      app: scale-nginx
  spec:
    containers:
      - name: sample-container
        image: busybox
        command: ["/bin/sh"]
        args:
          - -c
          - "while true; do yes > /dev/null; sleep 1; done"
```

HPA (DEPLOYMENT)

최종적으로 자원 크기를 제한한다. 앞서 이야기 하였지만 쿠버네티스는 CPU, MEM지원한다.

```
resources:  
  requests:  
    cpu: "100m"  
    memory: "128Mi"  
  limits:  
    cpu: "200m"  
    memory: "256Mi"  
# kubectl apply -f scale-nginx.yaml
```

배포 전략(HPA)

애플리케이션 실행 시, 워크로드에 따라서 POD의 개수를 늘리고 줄이기 위해서 HPA기능을 구성해야 한다. HPA는 Horizontal Pod Autoscaler의 약자다. 코드는 다음과 같이 작성이 가능하다.

```
# vi nginx-hpa.yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: scale-nginx
  minReplicas: 2
```

배포 전략(HPA)

CPU사용율이 50%가 넘어가면 POD 확장을 시작한다.

```
metrics:
- type: Resource
  resource:
    name: cpu
  target:
    type: Utilization
    averageUtilization: 50
# kubectl apply -f nginx-hpa.yaml
# kubectl get -f nginx-hpa.yaml
```

배포 전략(HPA)

HPA와 같은 스케일링 기반으로 서비스가 구성되는 경우, 스케줄러가 모든 노드 상태를 실시간으로 알 수 없기 때문에 종종, 예상된 숫자보다 더 작게 스케일 아웃(Scale Out)이 되는 경우가 있다. 이러한 이유로, POD개수를 유지할 수 있도록 명시해야 한다. 앞에 내용에 추가한다.

```
maxReplicas: 10
metrics:
- type: Resource
  resource:
    name: cpu
  target:
    type: Utilization
    averageUtilization: 50
# kubectl apply -f nginx-hpa.yaml
# kubectl get -f nginx-hpa.yaml
```

VPA(DEPLOYMENT)

테스트를 위해서 아래와 같이 작성한다.

```
# vi vpa-busybox.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vpa-busybox
  labels:
    app: vpa-busybox
spec:
  replicas: 1
  selector:
    matchLabels:
      app: vpa-busybox
```

VPA(DEPLOYMENT)

위의 내용 계속…

```
template:  
  metadata:  
    labels:  
      app: vpa-busybox  
spec:  
  containers:  
  - name: sample-container  
    image: busybox  
    command: ["/bin/sh"]  
  args:  
  - -c  
  - "while true; do yes > /dev/null; sleep 1; done"
```

VPA(DEPLOYMENT)

최종적으로 자원 크기를 제한한다. 앞서 이야기 하였지만 쿠버네티스는 CPU, MEM지원한다.

```
resources:  
  requests:  
    cpu: "100m"  
    memory: "128Mi"  
  limits:  
    cpu: "200m"  
    memory: "256Mi"  
# kubectl apply -f vpa-busybox.yaml
```

배포 전략(VPA)

VPA는 HPA와 반대로 수직적으로 자원을 조절한다.

```
# vi scale-vpa-busybox.yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-busybox
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: vpa-busybox.yaml
  updatePolicy:
    updateMode: "Auto"
```

배포 전략(VPA)

VPA는 HPA와 반대로 수직적으로 자원을 조절한다. HPA, VPA 자원은 뒤에서 좀 더 확인한다.

```
# kubectl apply -f scale-vpa-busybox.yaml
verticalpodautoscaler.autoscaling.k8s.io/sample-vpa created
# kubectl describe vpa vpa-busybox
```

롤링 업데이트

확장 명령어

2025-03-16

롤링 업데이트

롤링 업데이트는 기존 서비스에 영향 없이 업데이트 및 릴리즈 하는 방법이다. 롤링 업데이트를 통해서 언제든지 사용자는 롤백이 가능하다. 기본적으로 다음과 같은 기능을 제공한다.

단계	명령어	설명
1. 기존 Deployment 생성	<code>kubectl apply -f nginx-deployment.yaml</code>	nginx 1.19 버전 배포
2. 현재 상태 확인	<code>kubectl get deployments</code>	배포된 Pod 확인
3. 업데이트 적용	<code>kubectl set image deployment/nginx-deployment nginx=nginx:1.21</code>	nginx 버전 업데이트
4. 업데이트 진행 확인	<code>kubectl rollout status deployment/nginx-deployment</code>	롤링 업데이트 진행 상황 확인
5. 롤백 (되돌리기)	<code>kubectl rollout undo deployment/nginx-deployment</code>	이전 버전으로 복구

롤링 업데이트

롤링 업데이트를 사용하기 위해서 다음과 같이 구성이 가능하다. 간단하게 Deployment구성 후, 테스트 한다.

```
# vi nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

롤링 업데이트

롤링 업데이트를 사용하기 위해서 다음과 같이 구성이 가능하다. 간단하게 Deployment구성 후, 테스트 한다.

```
template:  
  metadata:  
    labels:  
      app: nginx  
  spec:  
    containers:  
      - name: nginx  
        image: nginx:1.19  
        ports:  
          - containerPort: 80  
# kubectl apply -f nginx-deployment.yaml  
# kubectl get -f nginx-deployment.yaml
```

롤링 업데이트

버전을 변경한다. 버전이 변경이 되면, 즉시 롤링 업데이트가 시작이 된다.

```
# kubectl set image deployment/nginx-deployment nginx=nginx:1.21
deployment.apps/nginx-deployment image updated
# kubectl rollout status deployment/nginx-deployment
deployment "nginx-deployment" successfully rolled out
# kubectl get pods
NAME                               READY   STATUS    RESTARTS
AGEnginx-deployment-5678fc88d8-uvwxy   1/1     Running   0
5snginx-deployment-5678fc88d8-qrstz    1/1     Running   0
10snginx-deployment-5678fc88d8-klmno   1/1     Running   0
                                         30s
```

롤링 업데이트

이전 버전으로 롤백은 다음과 같다. 롤백은 Deployment를 사용하는 RS(ReplicaSet)를 통해서 수행이 된다.

```
# kubectl rollout undo deployment/nginx-deployment
deployment.apps/nginx-deployment rolled back
# kubectl get rs
# kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
deployment-6799fc88d8-xxxxx      1/1     Running   0          5s
nginx-10s                         1/1     Running   0          10s
deployment-6799fc88d8-zzzzz      1/1     Running   0          30s
```

블루/그린, 카나리 업데이트

확장 명령어

2025-03-16

B/G, CANARY 설명

블루/그린 및 카나리는 다음과 같은 차이가 있다. 이 차이점을 통해서 어떠한 방식으로 서비스에 적용할 지 결정한다.

배포 방식	방식	특징
Blue/Green	새 환경을 배포 후 트래픽을 전환	즉시 전환 가능, 빠른 롤백 가능
Canary	일부 트래픽만 새 버전으로 이동	점진적 배포, 작은 범위에서 테스트 가능

BLUE/GREEN

쿠버네티스에서 Blue/Green를 사용하면, 기존 버전을 유지하면서 새로운 버전으로 전환이 가능하다. 이하 B/G를 사용하기 위해서 다음과 같이 Deployment를 생성한다.

```
# vi nginx-blue.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-blue
  labels:
    app: nginx
    version: blue
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      version: blue
```

BLUE/GREEN

이전과 동일하게 버전 1.19기반으로 Nginx서버를 생성 및 구성한다.

```
template:
  metadata:
    labels:
      app: nginx
      version: blue
  spec:
    containers:
      - name: nginx
        image: nginx:1.19
    ports:
      - containerPort: 80
# kubectl apply -f nginx-blue.yaml
```

BLUE/GREEN

블루 POD로 트래픽을 전달한 SVC를 구성한다.

```
# vi nginx-server.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
    version: blue
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
# kubectl apply -f nginx-service.yaml
```

BLUE/GREEN

그린 역할을 할 POD를 생성한다.

```
# vi nginx-green.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-green
  labels:
    app: nginx
    version: green
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      version: green
```

BLUE/GREEN

이전 버전과 다르게 이미지 버전이 1.21로 명시가 되어있다.

```
template:
  metadata:
    labels:
      app: nginx
      version: green
  spec:
    containers:
      - name: nginx
        image: nginx:1.21
      ports:
        - containerPort: 80
# kubectl apply -f nginx-green.yaml
```

2025-03-16

BLUE/GREEN

현재 서비스는 그린으로 트래픽이 흘러가지 않고, 블루로 흘러가고 있다. 트래픽을 그린으로 흘리기 위해서 다음과 같이 명령어를 실행한다.

```
# kubectl patch service nginx-service -p  
'{"spec": {"selector": {"app": "nginx", "version": "green"}}}'
```

혹은 에디터 명령어로 수정이 가능하다.

```
# kubectl edit service nginx-service
```

문제가 있다고 판단이 되면, 언제든지 다시 SVC에서 블루 버전으로 변경이 가능하다.

BLUE/GREEN

올바르게 서비스 구성이 되었으면, 기존 블루를 제거한다.

```
# kubectl delete deployment nginx-blue  
deployment.apps/nginx-blue deleted
```

CANARY

카나리(CANARY)는 B/G와 다르게 점진적으로 POD를 생성하면서 전환을 시작한다. 이와 비슷한 업데이트가 A/B 업데이트라고 부르기도 한다. 하지만, 최종적인 동작 방식은 다르다. A/B를 사용하기 위해서 Istio와 같은 서비스 라우팅 기능이 필요하다.

레드햇 오픈 시프트는 이 부분을 라우터에 포함이 되어 있으며, 바닐라 버전은 일반적으로 Istio기반으로 처리한다.

CANARY

먼저, nginx-stable 버전을 만들어서 배포한다.

```
# vi nginx-stable.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-stable
  labels:
    app: nginx
    version: stable
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      version: stable
```

CANARY

앞에서 사용한 코드를 활용한다.

```
template:
  metadata:
    labels:
      app: nginx
      version: stable
  spec:
    containers:
      - name: nginx
        image: nginx:1.19
      ports:
        - containerPort: 80
# kubectl apply -f stable-nginx.yaml
# kubectl get -f stable-nginx.yaml
```

CANARY

먼저, nginx-canary 버전을 만들어서 배포한다.

```
# vi nginx-canary.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-canary
  labels:
    app: nginx
    version: canary
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
      version: canary
```

CANARY

앞에서 사용한 코드를 활용한다.

```
template:
  metadata:
    labels:
      app: nginx
      version: canary
  spec:
    containers:
      - name: nginx
        image: nginx:1.21
      ports:
        - containerPort: 80
# kubectl apply -f canary-nginx.yaml
# kubectl get -f canary-nginx.yaml
```

CANARY

위와 같이 하면, 1.19 3개, 1.21 1개의 POD가 동작. 이를 SVC를 통해서 연결 및 구성한다.

```
# vi nginx-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
# kubectl apply -f nginx-service.yaml
```

CANARY

POD비율을 ReplicaSet를 통해서 변경한다. 문제가 없다고 판단이 되면, stable버전을 제거하고 canary버전으로 트래픽을 전부 전달한다.

```
# kubectl scale deployment nginx-canary --replicas=3
deployment.apps/nginx-canary scaled
# kubectl set image deployment/nginx-stable nginx=nginx:1.21
# kubectl delete deployment nginx-canary
deployment.apps/nginx-stable updated
deployment.apps/nginx-canary deleted
```

정리

기능	설명	사용 목적
Liveness Probe	애플리케이션이 정상 작동 중인지 확인	비정상 상태일 경우 자동 재시작
Readiness Probe	트래픽을 받을 준비가 되었는지 확인	준비되지 않으면 서비스에서 제외
Startup Probe	컨테이너가 완전히 시작되었는지 확인	부팅 시간이 긴 애플리케이션

정리

기능	설명	사용 목적
Rolling Update	무중단 배포 방식	운영 환경에서 애플리케이션 업데이트
Recreate Update	기존 Pod 삭제 후 신규 배포	데이터 정합성이 중요한 경우
Graceful Shutdown	애플리케이션 정상 종료	데이터 손실 방지 및 안정적인 종료
HPA	부하에 따라 자동으로 Pod 개수 조정	트래픽 급증 대응
PDB	최소한의 Pod 개수를 유지	장애 발생 시 가용성 보장

정리

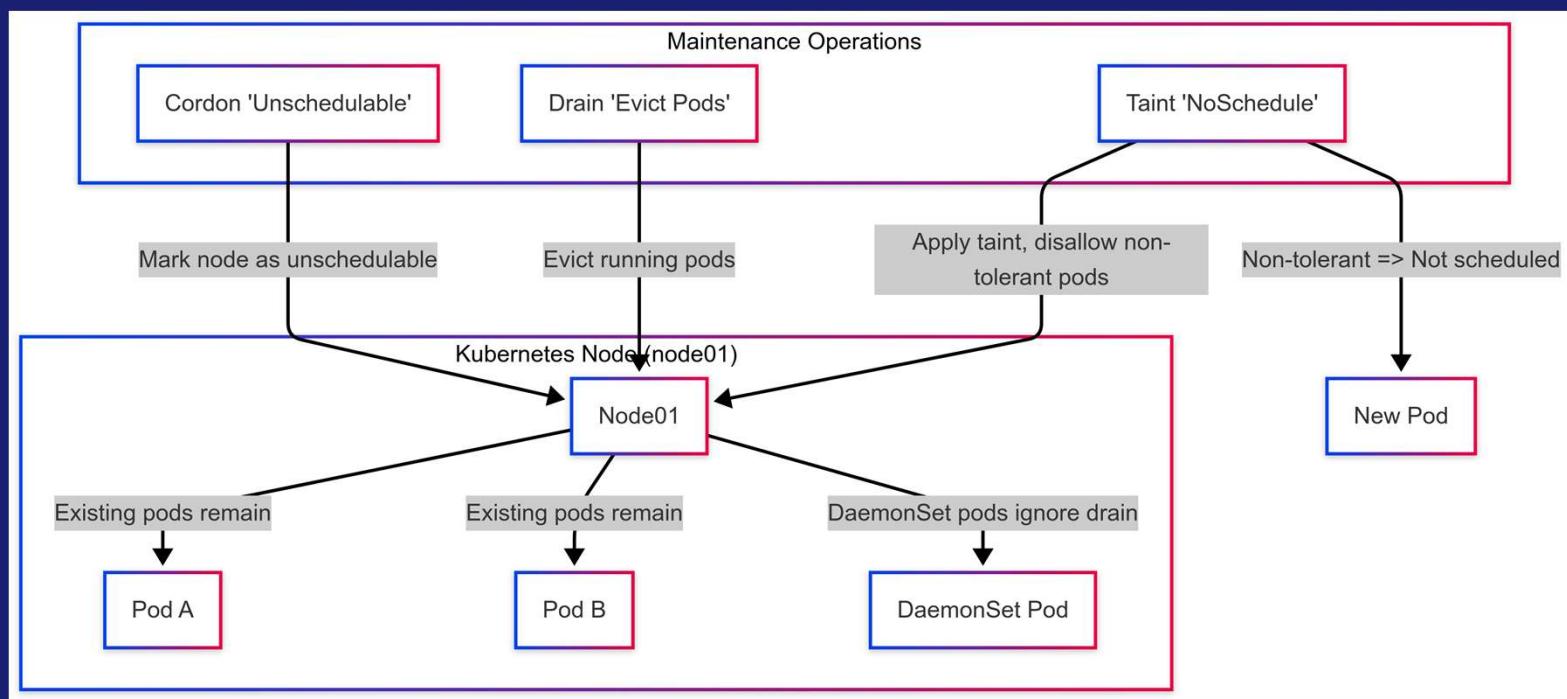
배포 방식	방식	특징
Blue/Green	새 환경을 배포 후 트래픽을 전환	즉시 전환 가능, 빠른 롤백 가능
Canary	일부 트래픽만 새 버전으로 이동	점진적 배포, 작은 범위에서 테스트 가능

DRAIN/TAINT/CORDON

확장 명령어

2025-03-16

설명



설명

기능을 비교하면 다음과 같다.

Command	Purpose	Usage Example	Expected Output
kubectl cordon	노드를 unschedulable 상태로 전환하여 신규 파드가 스케줄되지 않도록 함	<code>kubectl cordon node01</code>	node/node01 cordoned
kubectl drain	노드에서 실행 중인 파드들을 안전하게 축출하여 유지보수 모드로 전환	<code>kubectl drain node01 --ignore-demonsets --delete-local-data</code>	node/node01 cordoned "evicting pod ..." node/node01 drained
kubectl taint	노드에 taint를 적용하여 toleration이 없는 파드가 스케줄되지 않도록 함	<code>kubectl taint nodes node01 key1=value1:NoSchedule</code>	node/node01 tainted
		<code>kubectl taint nodes node01 key1:NoSchedule-</code>	node/node01 untainted

CORDON

특정 노드에 스케줄링이 되지 않도록 합니다. 모든 POD는 생성이 되지 않습니다.

```
# kubectl cordon node01  
node/node01 cordoned
```

DRAIN

유지보수를 위해서 사용중인 POD를 다른 노드로 이동 합니다. 정확히는 이동이 아니라 POD를 재구성 합니다. 하지만 DaemonSet POD는 영향받지 않습니다.

```
# kubectl drain node01 --ignore-daemonsets --delete-local-data node/node01
node/node01 cordoned
node/node01 cordoned
evicting pod nginx-deployment-xxxx from node node01
pod "nginx-deployment-xxxx" evicted
...
node/node01 drained
```

Taint

특정 노드에 Taint를 선언하여 POD생성이 되지 않는다. 다만, Toleration에 따라서 POD가 생성이 될 수 있다.

```
# kubectl taint nodes node01 key1=value1:NoSchedule  
node/node01 tainted  
# kubectl taint nodes node01 key1:NoSchedule-  
node/node01 untainted
```

LIVENESS/READINESS

확장 명령어

2025-03-16

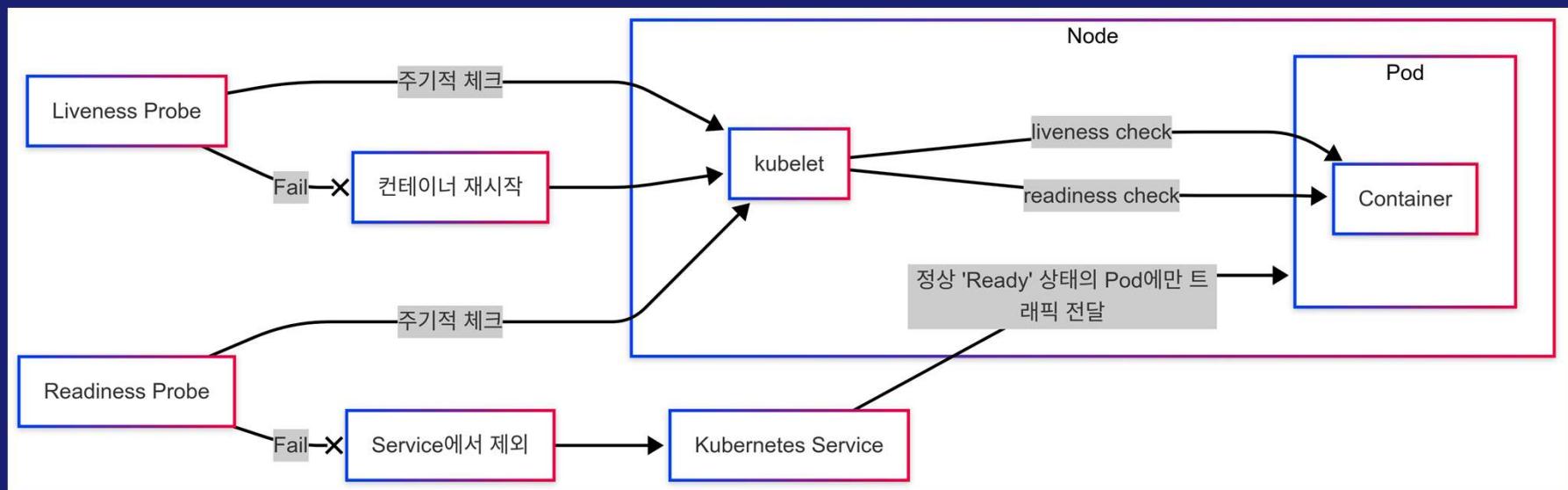
설명

관리 방법	설명	사용 예시
Liveness Probe	애플리케이션이 정상 작동 중인지 감지하여, 비정상 상태일 경우 자동 재시작	<code>kubectl apply -f liveness.yaml</code>
Readiness Probe	애플리케이션이 트래픽을 받을 준비가 되었는지 감지하여, 준비되지 않으면 서비스에서 제외	<code>kubectl apply -f readiness.yaml</code>
Startup Probe	애플리케이션이 완전히 시작되었는지 확인하여, 일정 시간 동안 기다려 준 후 Liveness Probe 적용	<code>kubectl apply -f startup.yaml</code>

설명

관리 방법	설명	사용 예시
Rolling Update	기존 Pod를 점진적으로 대체하여 애플리케이션 무 중단 배포 수행	<code>kubectl set image deployment/nginx nginx=nginx:latest</code>
Recreate Update	기존 Pod를 모두 삭제 후 새로운 버전으로 재시작	<code>kubectl rollout restart deployment/nginx</code>
Pod Disruption Budget(PDB)	장애 발생 시 최소한의 Pod를 유지하도록 설정하여 가용성 보장	<code>kubectl apply -f pdb.yaml</code>
Graceful Shutdown	애플리케이션이 정상적으로 종료될 수 있도록 preStop Hook을 이용해 종료 전 작업 수행	<code>kubectl delete pod nginx --grace-period=30</code>
Horizontal Pod Autoscaler(HPA)	CPU/메모리 사용량을 기준으로 자동으로 Pod 개수를 조정	<code>kubectl apply -f hpa.yaml</code>

설명



2025-03-16

프로브(LIVENESS)

프로브를 통해서 모니터링은 다음과 같이 한다. 컨테이너가 죽었는지 상태 확인을 한다. 이를 Liveness라고 한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-probe-pod
spec:
  containers:
  - name: nginx
    image: nginx
    livenessProbe:
      httpGet:
        path: /health
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
```

프로브(READINESS)

컨테이너는 동작하였고, 통신이 가능한 상태인지 확인하기 위해서 아래와 같이 명령어를 실행한다. 위의 내용에 추가한다. 컨테이너에서 사용하는 특정 포트 및 URL을 명시한다.

```
readinessProbe:
```

```
  httpGet:  
    path: /ready  
    port: 80  
  initialDelaySeconds: 5  
  periodSeconds: 10
```

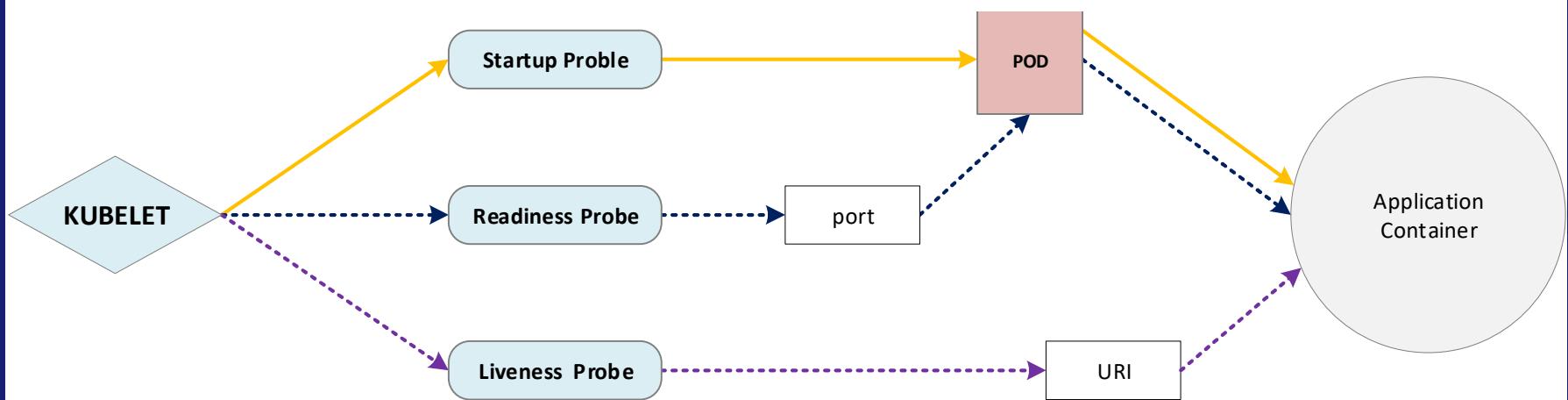
프로브(READINESS)

컨테이너 시작 시간이 상대적으로 긴 경우(대략 1분 이상), 다음과 같이 조건을 걸어서 올바르게 동작하도록 한다. 여기서는 30회 재시작을 시도하고, 10초 대기한다. 이 이상으로 이벤트가 발생하면 작업을 중지한다.

```
startupProbe:  
  httpGet:  
    path: /startup  
    port: 80  
  failureThreshold: 30  
  periodSeconds: 10
```

종합

- Startup Probe → 애플리케이션이 완전히 시작되었는지 확인
- Readiness Probe → 애플리케이션이 트래픽을 받을 준비가 되었는지 확인
- Liveness Probe → 애플리케이션이 정상적으로 실행되고 있는지 확인



2025-03-16

LABEL

확장 명령어

2025-03-16

노드 셀렉터 및 선호도

노드 셀렉터 및 선호도의 차이점은 다음과 같다.

기능	nodeSelector	nodeAffinity
사용 방식	단순한 키-값 매칭	표현식으로 유연한 조건 설정 가능
강제성	해당 노드가 아니면 실행 안 됨	"preferredDuringSchedulingIgnoredDuringExecution" 옵션을 사용하면 선호도만 설정 가능
예제 사용법	nodeSelector 필드 사용	nodeAffinity 필드 사용

노드 셀렉터

노드 셀렉터는 특정 노드에 레이블 구성 후, 해당 노드로 POD를 생성할 수 있도록 구분한다. 셀렉터 예제로 많이 사용하는 케이스는 디스크나 혹은 GPU부분이다. 예를 들어서 디스크가 SSD, HDD이렇게 있다고 가정한다.

그러면, SSD나 혹은 HDD(SAS)로 구성된 노드에 레이블을 걸어서 사용자가 원하는 상황에 따라서 선택 및 사용할 수 있도록 한다.

```
# kubectl label nodes <노드이름> disktype=ssd  
# kubectl label nodes cluster1-node1.example.com disktype=ssd
```

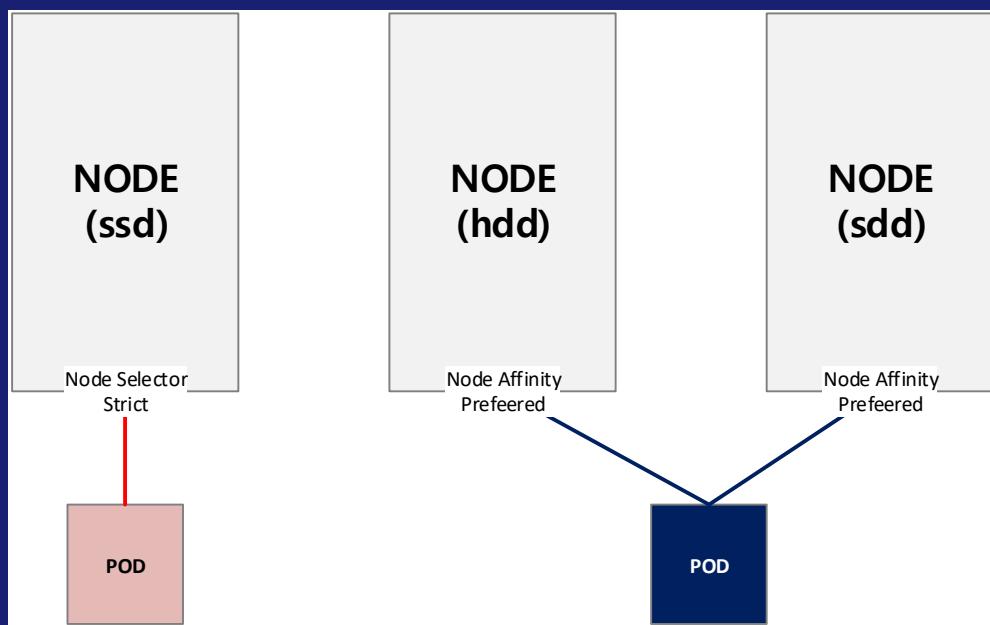
이 조건으로 적절한 NODE에 POD를 생성한다.

노드 셀렉터

```
# vi node-selector.yaml
apiVersion: v1
kind: Pod
metadata:
  name: node-selector-pod
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
    disktype: ssd
# kubectl apply -f node-selector.yaml
# kubectl get -f node-selector.yaml
```

노드 선호도 (Node Affinity)

노드 선호도는 셀렉터와 비슷하지만, 다음과 같이 다른 부분이 있다. 셀렉터는 제한에 가깝고, 선호도는 여러 노드에 복합적으로 조건을 적용하는 방법.



노드 선호도 (Node Affinity)

```
# vi node-affinity.yaml

apiVersion: v1
kind: Pod
metadata:
  name: node-affinity-pod
spec:
  containers:
    - name: nginx
      image: nginx
```

노드 선호도 (Node Affinity)

```
affinity:  
  nodeAffinity:  
    requiredDuringSchedulingIgnoredDuringExecution:  
      nodeSelectorTerms:  
        - matchExpressions:  
            - key: disktype  
              operator: In  
              values:  
                - ssd  
# kubectl apply -f node-affinity.yaml
```

ReplicaSet/ReplicationController

Deployment/DaemonSet

StatefulSet

확장 명령어

2025-03-16

미리보기

쿠버네티스에서 제공하는 POD생성 및 복제 서비스는 다음과 같이 릴리즈가 되었다. 진행 전, 참조하도록 한다. 왜냐면, 이 내용들은 아래에서 계속 언급이 된다.

기능	도입된 Kubernetes 버전	연도
ReplicationController	v1.0 (Stable)	2015
DaemonSet	v1.0 (Stable)	2015
ReplicaSet	v1.2 (Beta), v1.9 (Stable)	2016 (Beta), 2017 (Stable)
StatefulSet	v1.5 (Beta), v1.9 (Stable)	2016 (Beta), 2017 (Stable)
Deployment	v1.2 (Beta), v1.9 (Stable)	2016 (Beta), 2017 (Stable)
DeploymentConfig	OpenShift Specific	2015

ReplicaSet

ReplicaSet은 기본적으로 Deployment와 같이 사용한다. 하지만, 상황에 따라서 독립적으로 사용이 가능하다. ReplicaSet의 역할은 다음과 같다.

1. 명시가 된 개수만큼 POD를 유지한다.
2. 제거가 된 POD가 확인이 되면, 즉시 바로 재구성이 된다.
3. 버전이 변경이 되는 경우, ReplicaSet은 기존 내용을 유지하고 새로운 ReplicaSet 정책 기반으로 POD를 구성 한다.
4. Deployment가 없는 경우, Revision기능을 사용 할 수 없다.

대다수 서비스 애플리케이션은 ReplicaSet기반으로 동작하기 때문에, 최소 한 개의 ReplicaSet 설정은 권장한다. 일반적인 서비스 애플리케이션은 Deployment를 통해서 배포가 되기 때문에, 별도로 작성하는 경우는 거의 드물다.

ReplicaSet

최신 버전의 쿠버네티스는 가급적이면(거의 100%) ReplicaSet기반으로 POD생성을 권장한다. 그 이유는 다음과 같다.

기능	ReplicaSet (RS)	ReplicationController (RC)
오브젝트 버전	apps/v1	v1
Pod 개수 관리	<input checked="" type="checkbox"/> 가능	<input checked="" type="checkbox"/> 가능
레이블 셀렉터 지원	<input checked="" type="checkbox"/> Set-based Selector 지원 (matchExpressions)	<input type="checkbox"/> Only Equality-based Selector (matchLabels)
Pod 자동 복구	<input checked="" type="checkbox"/> 가능	<input checked="" type="checkbox"/> 가능
Deployment와 연동	<input checked="" type="checkbox"/> 사용 가능	<input type="checkbox"/> 사용 불가능
추천 여부	<input checked="" type="checkbox"/> (권장) 최신 버전이며, Deployment와 함께 사용 가능	<input type="checkbox"/> (구버전, 더 이상 사용 권장되지 않음)

ReplicaSet

ReplicaSet만 사용하여 POD를 구성하는 경우 다음과 같이 생성 및 구성한다.

```
# vi replicaset.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-nginx
  labels:
    app: nginx
    tier: lab-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: lab-rs
```

ReplicaSet

위의 내용에 이어서…

```
template:  
  metadata:  
    labels:  
      tier: lab-rs  
  spec:  
    containers:  
    - name: nginx  
      image: docker.io/library/nginx:latest
```

ReplicaSet

등록된 내용을 확인 시, 다음과 같이 명령어를 실행하여 올바르게 생성 및 구성이 되었는지 확인한다.

```
# kubectl get -f replicaset.yaml
```

ReplicaController

컨트롤러는 초창기 쿠버네티스에서 사용하던 자원이다. **ReplicaController(RC)**와 **Deployment**의 도입 시기는 다음과 같다.

Deployment 등장 과정 요약

1. Kubernetes 1.0 (2015년 7월): ReplicationController 도입
2. Kubernetes 1.2 (2016년 3월): ReplicaSet 및 Deployment (Beta) 도입
3. Kubernetes 1.9 (2017년 12월): Deployment GA(Stable)

초기부터 사용한 사용자를 위해서 **ReplicationController**는 호환성을 염두하고 존재하며, 가급적이면 **ReplicationController** 대신, **ReplicaSet**과 **Deployment** 사용을 권장한다.

ReplicaController

컨트롤러는 ReplicaSet하고 거의 동일하다. 차이점은 위에서 설명 하였다. 작성이 필요한 경우, 아래와 같이 구성이 가능하다.

```
# vi replicacontroller.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
    labels:
      app: nginx
```

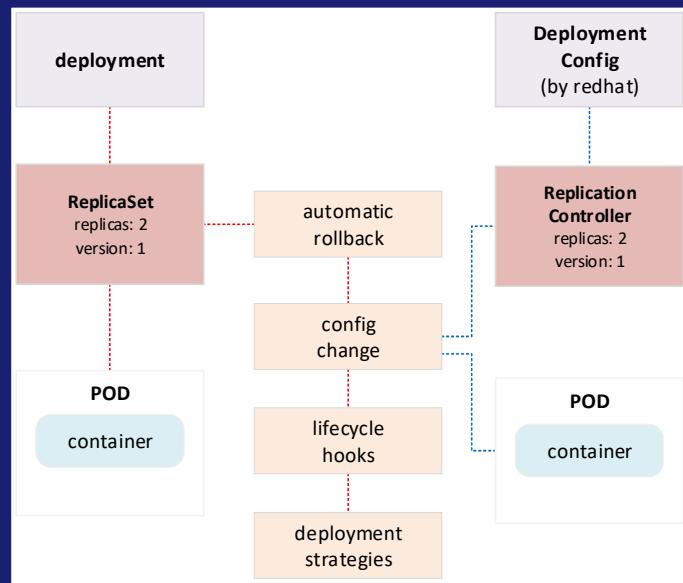
ReplicaController

위의 내용 계속…

```
spec:  
  containers:  
    - name: nginx  
      image: nginx  
    ports:  
      - containerPort: 80
```

Deployment

Deployment는 ReplicaSet과 함께 도입이 되었다. 생각보다 버전 안정화는 늦었다. GA기준 1.9에서 안정화가 되었다. Deployment의 주요 역할은 애플리케이션 설정 내용을 가지고 ReplicaSet으로 전달 및 배포가 주요 목적이다.



Deployment

Deployment 구성은 다음과 같이 구성이 되어 있다.

```
# vi deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

이 부분은 기존과 동일하다.

"replicas: 3"를 생성한다.

Deployment가 관리할 RS를 찾습니다.
RS에 해당 레이블이 존재해야 한다.

Deployment

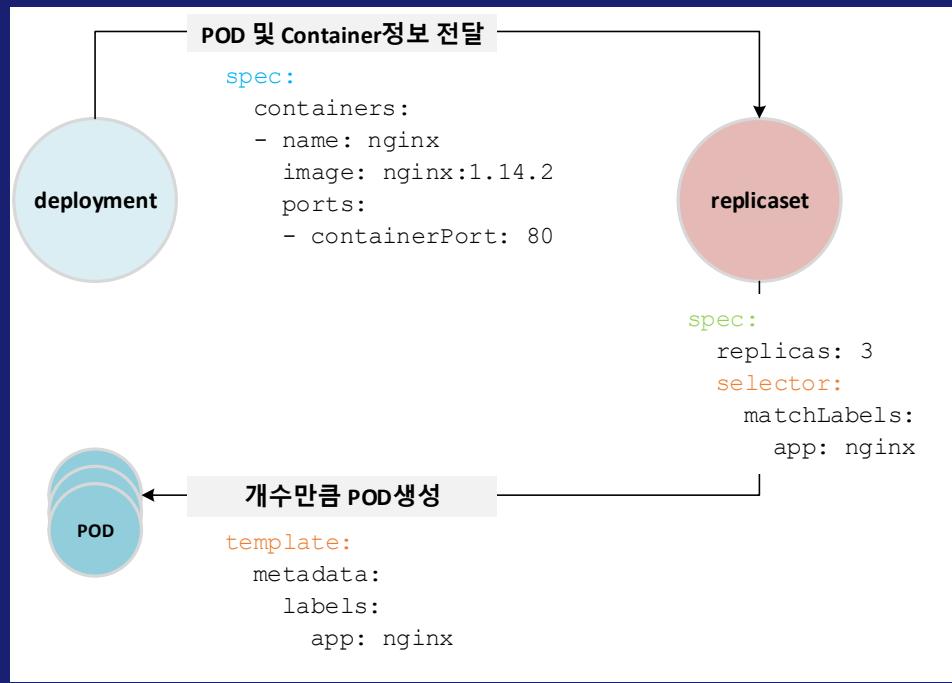
계속…

```
template:  
  metadata:  
    labels:  
      app: nginx  
spec:  
  containers:  
  - name: nginx  
    image: nginx:1.14.2  
  ports:  
  - containerPort: 80
```

"template",은 "matchLabels:"과 동일해야 한다.

Deployment

도식으로 표현하면 다음과 같이 동작한다.



Deployment

Deployment에 구성된 자원은 하나만 존재하며, Deployment가 갱신이 되면, 해당 내용에 대해서 ReplicaSet에서 기록을 가지고 있다. 이미지나 혹은 설정 내용에 변경이 발생하면, 다음처럼 ReplicaSet에서 기록이 남는다.

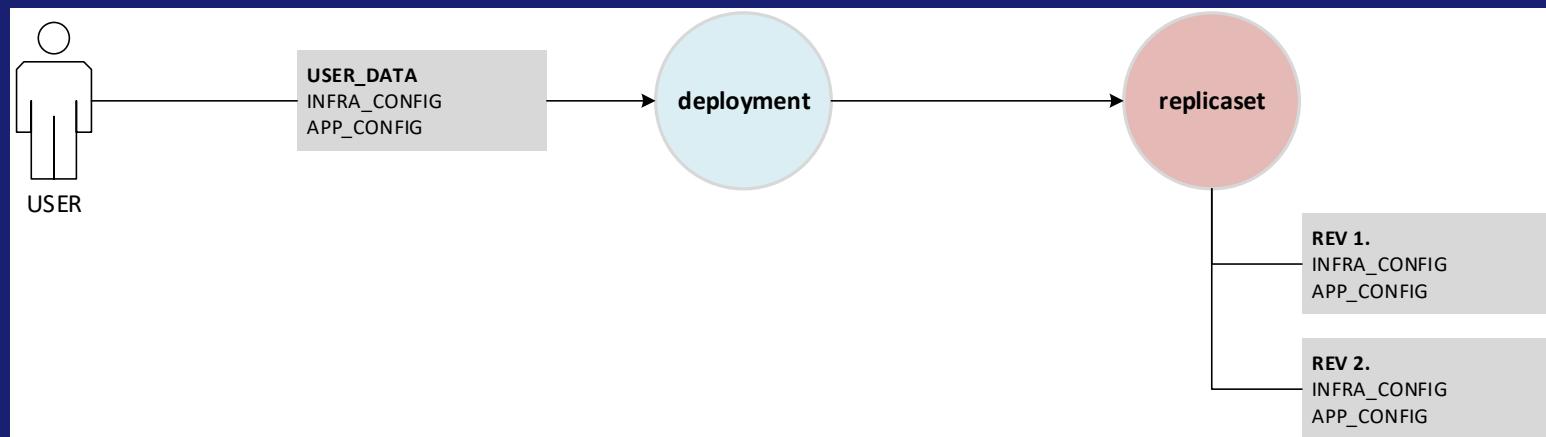
```
# kubectl get deploy
```

NAME	READY	AVAILABLE
nginx-deployment.apps/nginx-deployment	3/3	3

```
# kubectl get replicaset
```

NAME	DESIRED	CURRENT
nginx-deployment-665fd86dbb	1	1
nginx-deployment-77d8468669	3	3

Deployment



DaemonSet

DaemonSet은 Deployment와 기능상 동일하다. DaemonSet도 Deployment와 동일하게 2015년도에 정식으로 GA되었다. Deployment와 제일 큰 다른 부분은 Deployment는 Scheduler를 통해서 클러스터에 구성된 노드에 배포가 주요 목적이이다.

하지만, DaemonSet은 Deployment와 동일하게 Scheduler를 통하지만, 각 노드당 하나씩 POD생성이 주요 목적이다. 다시 말하지만, 컨테이너가 아닌 POD를 각 노드에 구성이다. 쿠버네티스에서 대표적인 DaemonSet POD는 kube-proxy이다. kube-proxy는 모든 노드에 최소 한 개씩 구성이 된다.

제한적으로 DaemonSet이 구성이 되는 서비스는 kube-apiserver, kube-scheduler, kube-controller-manager와 같은 POD가 있다. 이 서비스들은 Compute노드를 제외한, Controller노드에 구성이 된다.

DaemonSet

대몬셋은 다음과 같이 구성 및 작성한다. 올바른 학습을 위해서 최소 2개 이상의 컴퓨터 노드를 권장한다. 그렇지 않는 경우, 컴퓨터 노드에만 생성이 된다. 컨트롤러에도 생성하기 위해서는 허용레이션(toleration)를 같이 구성해야 한다. 여기서는 허용레이션을 구성하지 않는다. 순수하게 컴퓨터 노드에서만 생성한다.

```
spec:  
  spec:  
    tolerations:  
      - key: node-role.kubernetes.io/control-plane  
        operator: Exists  
        effect: NoSchedule  
      - key: node-role.kubernetes.io/master  
        operator: Exists  
        effect: NoSchedule
```

DaemonSet

예를 들어서 모든 노드에 로그 분석 에이전트가 필요한 경우 아래와 같이 생성 및 구성.

```
# vi daemonset.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
```

DaemonSet

위의 내용 이어서 계속…

```
template:  
  metadata:  
    labels:  
      name: fluentd-elasticsearch  
spec:  
  containers:  
  - name: fluentd-elasticsearch  
    image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
```

DaemonSet

위의 내용 이어서 계속…

```
resources:  
  limits:  
    memory: 200Mi  
  requests:  
    cpu: 100m  
    memory: 200Mi  
volumeMounts:  
- name: varlog  
  mountPath: /var/log
```

DaemonSet*

자원 생성 후, 올바르게 반영이 되었는지 확인한다.

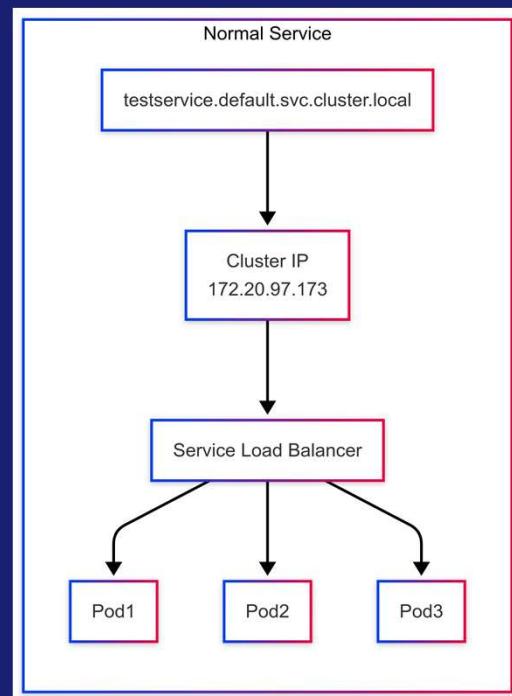
```
terminationGracePeriodSeconds: 30
volumes:
- name: varlog
  hostPath:
    path: /var/log
# kubectl apply -f daemonset.yaml
# kubectl get -f daemonset.yaml
```

StatefulSet

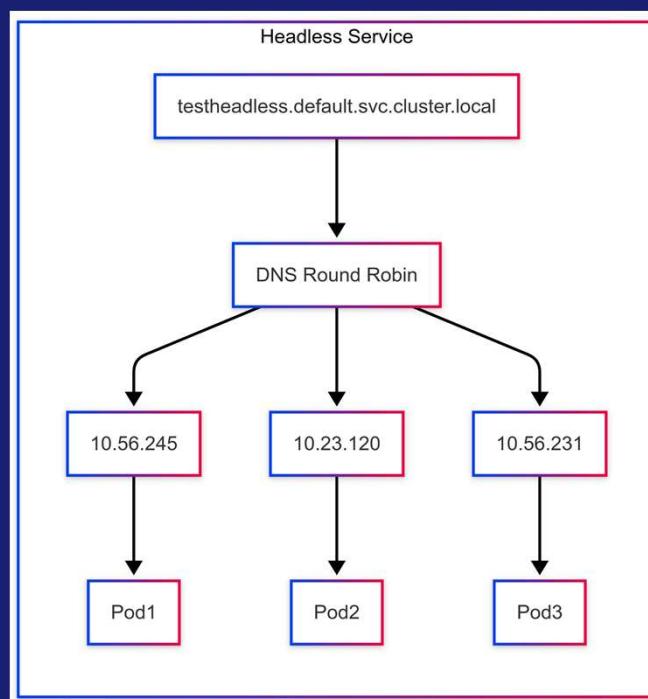
StatefulSet 서비스는 일반적으로 3 Tiers와 기반의 레거시 애플리케이션 혹은 일반적인 C/S 구조의 서비스 형태에 적용한다.

StatefulSet은 Headless 형태를 가지고 있다. Headless를 통해서 기존 애플리케이션 활용이 가능하다. Headless 개념은 여러 아키텍처에서 제공하며, 쿠버네티스에서 Headless 개념은 ClusterIP를 사용하지 않고, DNS 기반으로 적용이 된다.

StatefulSet



StatefulSet



StatefulSet

StatefulSet 서비스를 생성 및 구성한다. 구성 시 클러스터 아이피가 없어야 한다.

```
# kubectl apply -f sts-svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
# kubectl apply -f sts-svc.yaml
# kubectl get -f sts-svc.yaml
```

StatefulSet

StatefulSet 자원을 생성한다.

```
# vi sts.yaml
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
```

StatefulSet

```
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - name: nginx
        image: registry.k8s.io/nginx-slim:0.21
        ports:
          - containerPort: 80
            name: web
# kubectl apply -f sts.yaml
# kubectl get -f sts.yaml
```

Jobs/CronJob

확장 명령어

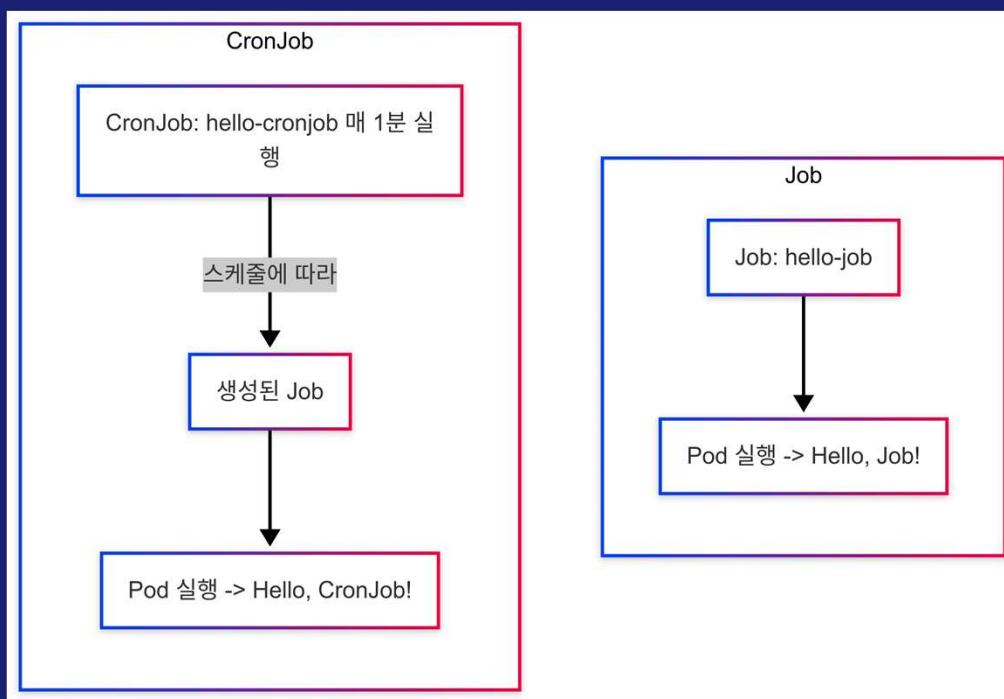
2025-03-16

설명

Job, CronJob의 기능 차이는 다음과 같다.

항목	Job	CronJob
목적	일회성 또는 단발성 작업을 실행하여 완료될 때까지 관리	정해진 일정에 따라 주기적으로 작업을 실행 (크론 스케줄러 사용)
실행 방식	Job 리소스가 생성되면 단 한 번 실행되고 완료될 때까지 실행	CronJob 스케줄에 따라 Job 리소스를 주기적으로 생성하여 실행
용도	데이터 마이그레이션, 일회성 배치 작업, 단발 테스트 등	정기적인 백업, 리포트 생성, 스케줄링된 데이터 처리 등
관리	Job이 완료되면 기본적으로 리소스가 남을 수 있으며, ttlSecondsAfterFinished 옵션을 통해 정리 가능	CronJob은 Job을 주기적으로 생성하므로, 완료된 Job 관리 및 정리 필요

설명



JOB

다음과 같이 작성 및 생성이 가능하다.

```
# vi hello-job.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello-job
spec:
  template:
    spec:
      containers:
        - name: hello
          image: busybox
          command: ["echo", "Hello, Job!"]
      restartPolicy: Never
      backoffLimit: 4
```

JOB

다음과 같이 작성 및 생성이 가능하다.

```
# kubectl apply -f hello-job.yaml
job.batch/hello-job created
# kubectl get pods --selector=job-name=hello-job
NAME           READY   STATUS    RESTARTS   AGE
hello-job-xxxxx   0/1     Completed   0          30s
# kubectl logs <pod-name>
Hello, Job!
```

CRONJOB

JOB과 비슷하지만, 특정 시간마다 실행이 된다.

```
# kubectl apply -f hello-cronjob.yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello-cronjob
spec:
  schedule: "*/1 * * * *"  # 매 1분마다 실행
```

CRONJOB

JOB과 비슷하지만, 특정 시간마다 실행이 된다.

```
jobTemplate:  
  spec:  
    template:  
      spec:  
        containers:  
          - name: hello  
            image: busybox  
            command: ["echo", "Hello, CronJob!"]  
        restartPolicy: Never  
      successfulJobsHistoryLimit: 3  
      failedJobsHistoryLimit: 1
```

CRONJOB

JOB과 비슷하지만, 특정 시간마다 실행이 된다.

```
# kubectl apply -f cronjob-example.yaml
cronjob.batch/hello-cronjob created
# kubectl get jobs --watch
NAME                  COMPLETIONS   DURATION   AGE
hello-cronjob-1615939200  1/1          2s         1m
hello-cronjob-1615939260  1/1          2s         30s...
# kubectl get pods --selector=job-name=hello-cronjob-1615939200
# kubectl logs <pod-name>
Hello, CronJob!
```

명령어

명령어로 생성 및 구성이 가능하다.

```
# kubectl create job hello-job --image=busybox -- /bin/sh -c "echo Hello, Job!"  
job.batch/hello-job created  
# kubectl get pods --selector=job-name=hello-job  
# kubectl logs <pod-name>  
Hello, Job!
```

명령어

명령어로 생성 및 구성이 가능하다.

```
# kubectl create cronjob hello-cronjob --image=busybox --schedule="*/1 * * * *" -- /bin/sh -c "echo Hello, CronJob!"  
cronjob.batch/hello-cronjob created  
# kubectl get cronjob hello-cronjob  
# kubectl get jobs --watch  
# kubectl get pods --selector=job-name=<cronjob-generated-job-name>  
# kubectl logs <pod-name>  
Hello, CronJob!
```

변수전달

확장 명령어

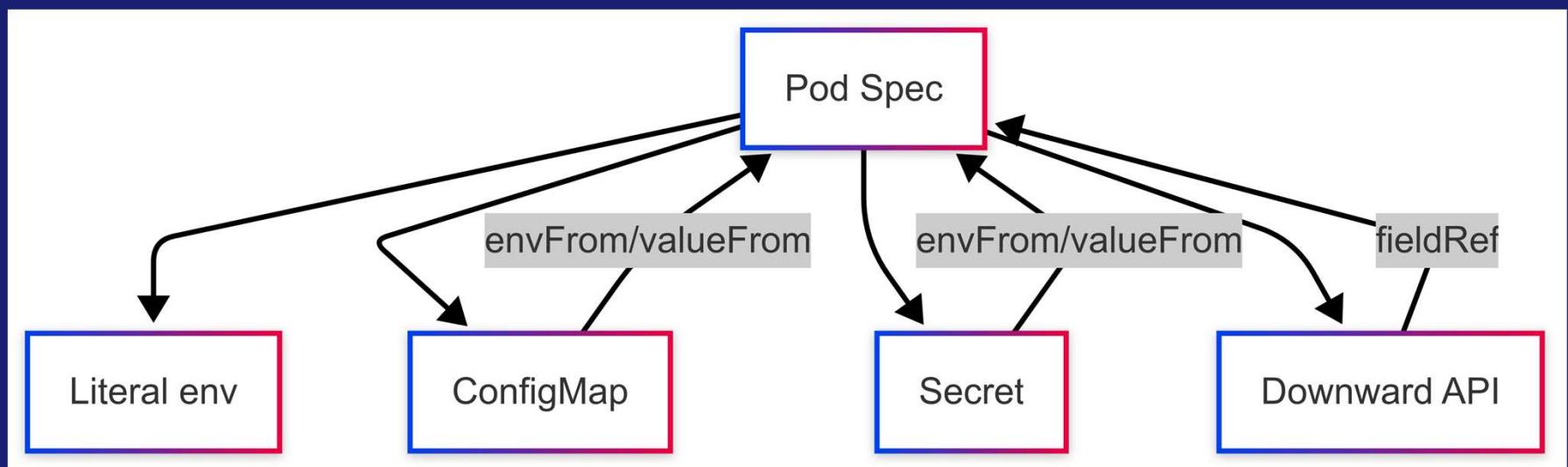
2025-03-16

변수

쿠버네티스에서 변수 전달은 다음과 같은 방법으로 전달이 가능하다. 여기서 configmap, secret은 뒤에서 별도로 다룬다. 여기서는 ENV를 통해서 변수를 다루는 방법을 이야기 한다.

방법	설명	사용 사례 및 장점	단점 또는 주의사항
Literal 정의 (env)	Pod 스펙 내에 직접 key-value 형태로 환경 변수를 정의	간단한 변수 전달에 적합 빠르게 설정 가능	보안이 필요한 값은 노출 위험 있음
ConfigMap 사용 (envFrom, valueFrom)	ConfigMap에 저장된 데이터를 환경 변수로 주입하여 여러 Pod에서 재사용 가능	구성 정보를 중앙 관리할 수 있음 환경 별 구성 변경에 유리	ConfigMap 자체의 보안 이슈(민감 데이터 부적합)
Secret 사용 (envFrom, valueFrom)	Secret에 저장된 민감한 데이터를 환경 변수로 주입	암호, 토큰 등 민감한 정보를 안전하게 전달 가능	Base64 인코딩되어 있으므로, 추가 디코딩 필요
Downward API (fieldRef)	Pod의 메타데이터(예: 이름, 네임스페이스, IP 등)를 환경 변수로 전달	동적으로 Pod 정보를 전달할 때 유용	전달할 수 있는 정보에 제한이 있음

변수



2025-03-16

POD 설정파일 및 환경 변수

확장 명령어

2025-03-16

애플리케이션 인자 값 설정 및 Containerfile

시험과 상관 없지만, 어떠한 방식으로 전달이 되는지 확인한다. 쿠버네티스는 기본적으로 이미지 빌드를 제공하지 않는다. 이미지를 올바르게 구성 및 빌드가 되어야지 쿠버네티스에서 전달하는 자원을 올바르게 활용이 가능하다.

쿠버네티스는 두 가지 형태로 자원 접근을 제공한다.

1. ConfigMap
2. Secret

ConfigMap은 환경 설정 파일 또는 변수를 관리(예: DB_HOST, APP_MODE 등)를 전달 시, 사용한다.

Secret은 민감한 정보 예를 들어, 비밀번호, API 키 등 민감한 정보 관리 (예: DB_PASSWORD)를 전달 시, 사용 한다.

CONTAINERFILE

Containerfile기반으로 다음과 같이 작성이 가능하다.

```
# vi Containerfile
FROM nginx:alpine
# 환경 변수를 읽을 수 있도록 설정
CMD ["sh", "-c", "echo 'DB_HOST: '$DB_HOST; echo 'DB_PASSWORD: '$DB_PASSWORD; nginx -g 'daemon off;'"]
# podman build -t my-nginx .
```

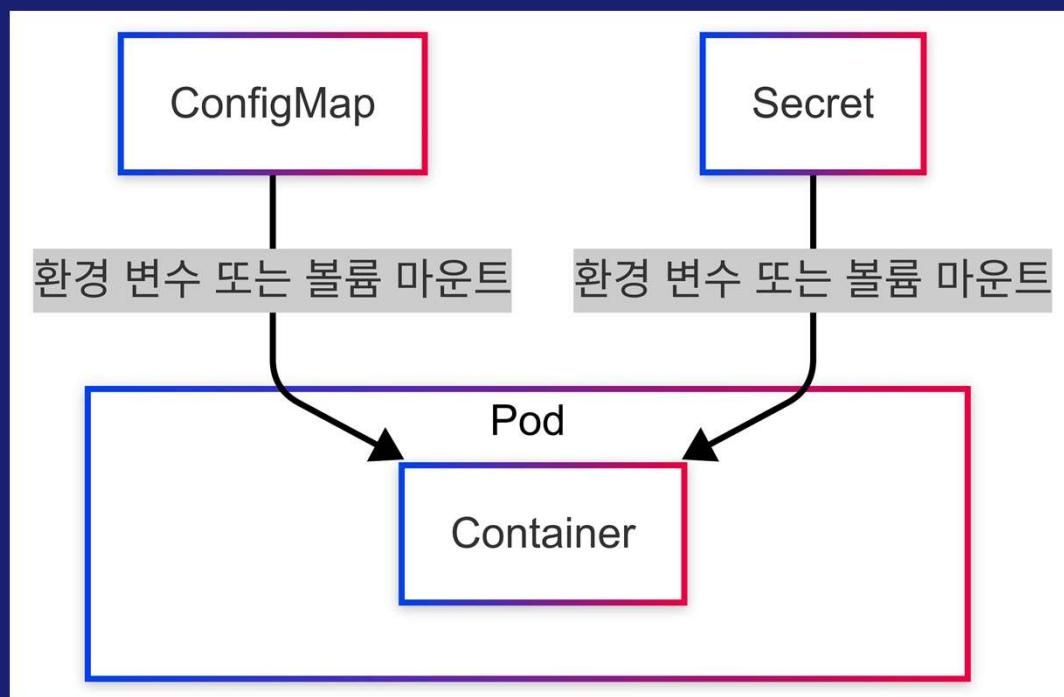
이를 기반으로 Configmap, Secret를 만들어서 간단하게 확인한다.

ConfigMap, Secret

확장 명령어

2025-03-16

설명



설명

위의 기능들을 정리하면 다음과 같다.

기능	설명	YAML 파일명	적용 명령어
ConfigMap 생성	환경 변수를 관리	my-config.yaml	kubectl apply -f my-config.yaml
Secret 생성	비밀번호, API 키 관리	my-secret.yaml	kubectl apply -f my-secret.yaml
Deployment에서 사용	환경 변수로 전달	nginx-deployment.yaml	kubectl apply -f nginx-deployment.yaml
데이터 확인	ConfigMap & Secret 조회		<code>kubectl get configmap my-config -o yaml</code>
			<code>kubectl get secret my-secret -o yaml</code>
Secret 복호화	base64 인코딩된 데이터 확인		<code>kubectl get secret my-secret -o jsonpath='{.data.DB_PASSWORD}'</code>

설명

항목	ConfigMap	Secret
파일로 마운트 가능 여부	가능 (configMap 볼륨 사용)	가능 (secret 볼륨 사용)
Pod 내부에서 사용 가능 여부	가능 (env, volume)	가능 (env, volume)
보안 수준	낮음 (평문 저장)	높음 (Base64 인코딩)
RBAC(권한 관리) 적용 가능 여부	불가능 (일반 리소스)	가능 (kubectl get secret 권한 필요)
사용 예제	<pre>kubectl create configmap my-config --from-literal=DB_HOST=mydatabase. local</pre>	<pre>kubectl create secret generic my-secret --from-literal=D B_PASSWORD=SuperSecret123</pre>

ConfigMap

명령어로 생성하는 방법은 다음과 같다.

```
# kubectl create configmap my-config \
--from-literal=DB_HOST=mydatabase.local
# kubectl get cm
```

명령어로는 다음과 같이 작성 및 생성한다.

```
# vi my-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  DB_HOST: "mydatabase.local"
# kubectl apply -f my-config.yaml
```

Secret

명령어로 생성하는 방법은 다음과 같다.

```
# kubectl create secret generic my-secret \
--from-literal=DB_PASSWORD=SuperSecret123
# kubectl get secret
```

파일로 다음과 같이 작성 및 생성이 가능하다.

```
# echo -n "SuperSecret123" | base64
U3VwZXJTZWNyZXQxMjM=
```

Secret

다음과 같이 파일로 작성한다.

```
# vi my-secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  DB_PASSWORD: "U3VwZXJTZWNyZXQxMjM=" # base64 인코딩된 값
# kubectl apply -f my-secret.yaml
# kubectl get secret my-secret -o jsonpath='{.data.DB_PASSWORD}' | base64 --decode
```

DEPLOYMENT 반영 및 적용

디플로이먼트에 적용하여, 올바르게 POD에 구성이 되었는지 확인한다.

```
# vi nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
```

DEPLOYMENT 반영 및 적용

계속 이어서...

```
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - name: nginx
        image: my-nginx
    env:
      - name: DB_HOST
        valueFrom:
          configMapKeyRef:
            name: my-config
            key: DB_HOST
```

DEPLOYMENT 반영 및 적용

아래는 POD에 env:를 통해서 ConfigMap, SecretKey를 POD를 통해서 컨테이너에게 전달한다.

```
- name: DB_PASSWORD
  valueFrom:
    secretKeyRef:
      name: my-secret
      key: DB_PASSWORD
# kubectl apply -f nginx-deployment.yaml
# kubectl get pod -f nginx-deployment.yaml
```

DEPLOYMENT 적용

아래는 POD에 env:를 통해서 ConfigMap, SecretKey를 POD를 통해서 컨테이너에게 전달한다.

```
# kubectl logs -l app=nginx
DB_HOST: mydatabase.local
DB_PASSWORD: SuperSecret123
# kubectl exec -it -l nginx -- bash
```

NetworkPolicy

확장 명령어

2025-03-16

설명

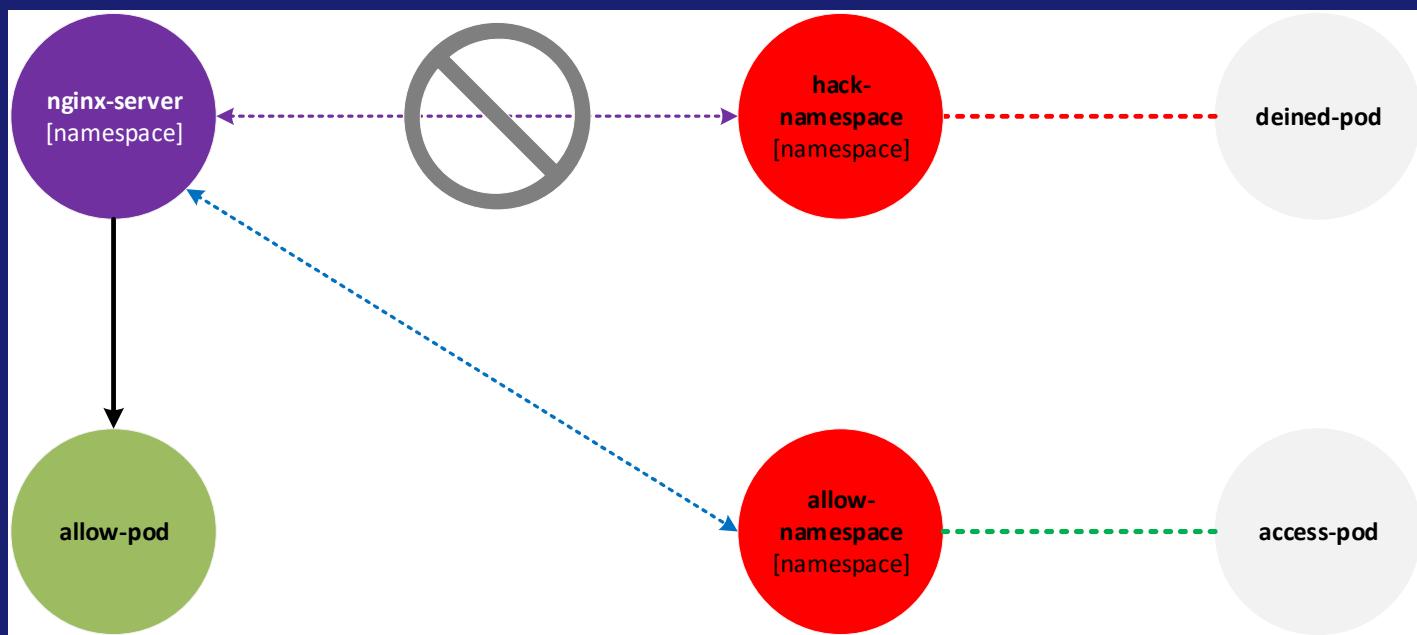
네임스페이스 간의 접근을 위한 기능으로, L2/L3/L4 레이어를 거치지 않고 라우팅을 설정함으로써 오버헤드 없이 신속하게 접근할 수 있다. 동작 방식은 아래 컴포넌트 기반으로 구성이 된다.

- Container Network Interface(CNI)
- Netfilter(iptables, nftables)

위의 기능 기반으로 백-엔드에서 다음과 같이 구성한다. 다음과 같은 조건으로 구성이 되었다.

1. Namespace: test-np
2. Pod: nginx-server
3. 허용: allowed-pod만 80 포트로 접근 가능
4. 차단: denied-pod는 접근 차단

설명



백엔드

다음과 같이 시스템에서 확인이 가능하다.

```
# iptables -L -v -n --line-numbers
Chain KUBE-NWPLCY-XXXXXX (1 references)
num  pkts bytes target     prot opt in   out    source          destination
 1      0     0 ACCEPT     tcp  --  *    *    10.244.1.5    10.244.1.10  tcp dpt:80
 2      0     0 DROP       all  --  *    *    0.0.0.0/0      10.244.1.10
```

1번 정책은 허용, 2번 정책은 거절. 즉, SOURCE에 명시가 되어 있는 아이피가 아닌 경우, 10.244.1.10에 접근을 거절한다.

백엔드

Cilium를 사용하는 경우, 다음과 같이 nftables에 구성이 된다. CNI마다 다르지만, 보통 구성은 비슷하다.

```
# iptables -L -v -n --line-numbers
table ip kube-nwpolicy
{
    chain ingress {
        ip saddr 10.244.1.5 tcp dport 80 accept
        drop
    }
}
```

네임스페이스의 특정 POD를 제약을 하지만, 네임스페이스 자체는 트래픽 라우팅을 조정할 수 없기 때문에, POD 아이피 기반으로 한다. 이를 구현하기 위해서 모든 POD는 올바른 레이블이 할당 및 구성이 되어 있어야 한다.

특정 POD만 허용

특정 POD만 허용하는 경우 YAML코드는 다음과 같다.

```
# vi allow-specific-pod.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-specific-pod
  namespace: test-np
spec:
  podSelector:
    matchLabels:
      app: nginx
  policyTypes:
    - Ingress
```

특정 POD만 허용

특정 POD만 허용하는 경우 YAML코드는 다음과 같다.

```
ingress:
- from:
  - podSelector:
    matchLabels:
      role: allowed
ports:
- protocol: TCP
  port: 80
# kubectl apply -f allow-specific-pod.yaml
# kubectl get -f allow-specific-pod.yaml
```

"role=allowed" 라벨이 있는 Pod만 허용

네임스페이스+POD 차단

특정 네임스페이스에 속한 모든 POD를 차단하는 경우, 다음과 같이 한다.

```
# vi deny-namespace.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-namespace
  namespace: test-np
spec:
  podSelector: {}
  policyTypes:
    - Ingress
```

네임스페이스+POD 차단

반드시, 네임 스페이스 생성 시 레이블 구성이 되어 있어야 한다.

```
ingress:  
- from:  
  - namespaceSelector:  
    matchLabels:  
      project: restricted  
ports:  
- protocol: TCP  
  port: 80
```

"project=restricted" 네임스페이스에
서 오는 트래픽 차단

네임스페이스+POD

특정 네임스페이스와 POD를 선택하는 경우, 다음과 같이 구성한다.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-specific-namespace
  namespace: test-np
spec:
  podSelector:
    matchLabels:
      app: web
  policyTypes:
  - Ingress
```

네임스페이스+POD

특정 네임스페이스와 POD를 선택하는 경우, 다음과 같이 구성한다.

```
ingress:  
- from:  
  - namespaceSelector:  
    matchLabels:  
      env: dev  
  - podSelector:  
    matchLabels:  
      team: frontend  
ports:  
- protocol: TCP  
  port: 443
```

"env=dev" 네임스페이스의 모든 Pod에서 접근 가능

"team=frontend" 라벨이 있는 Pod에서만 접근 가능

쿠버네티스 기능확장

쿠버네티스

2025-03-16

설명

인그레스 서비스는 이전 아키텍처에서 프록시 서버(Proxy Server)와 동일한 기능을 한다. Ingress서비스가 나오기 전에는 NodePort 혹은 Load Balancer서비스를 통해서 외부에서 접근이 되어야 한다. 다만, 이 방식의 문제는 서비스가 구성이 될 때, 관리자는 다음과 같은 부분을 수동으로 구성해야 한다.

- 서비스에서 사용할 아이피 할당(VIP)
- VIP를 통해서 접근 할 도메인 설정(Nginx, HAProxy, Appliance…)

이러한 이유로 쿠버네티스는 Ingress 서비스가 추가가 되었다. 쿠버네티스를 매우 잘 활용하는 레드햇 경우에는 이 부분을 router라는 자원으로 해결하고 있으며, Ingress, Router를 동시에 사용이 가능하다.

다른 쿠버네티스 플랫폼은 Ingress만 지원하고 있다.

설명

쿠버네티스에서 많이 사용하는 Ingress서비스는 다음과 같다.

- HAProxy
- Nginx

모든 서비스는 프로그램에 상관없이 보통 두 가지 기능으로 나누어 진다.

1. **Ingress**: 클러스터 내부 서비스에 접근 시, URL, HOSTNAME, TLS, ROUTING과 같은 정보를 정의하는 자원
2. **Ingress-controller**: Ingress에서 선언한 정보 기반으로 netfilter(iptables, nftables)와 그리고 프로그램에서 사용하는 라우팅 에이전트를 설정한다.

Ingress서비스를 사용하기 위해서는 로드밸런서 서비스가 반드시 필요하다.

2025-03-16

설명

위의 기능을 정리하면 다음과 같다. 여기서 말하는 IngressController는 일반적으로 많이 사용하는 Nginx, HAProxy기반이다.

항목	Ingress Controller	Istio Ingress Gateway	OpenShift Router
개념	외부 트래픽을 기본적으로 라우팅	Istio 서비스 메쉬의 일부 고급 트래픽 관리를 제공	OpenShift 전용 라우터로, HAProxy 기반으로 동작
라우팅 기능	URL 및 호스트 기반 라우팅 지원	정교한 트래픽 관리 A/B 테스트, canary 자체 지원	URL 및 호스트 기반 라우팅 지원
TLS 종료	지원	TLS 종료 및 mTLS 지원 암호화된 트래픽 처리 가능	자동 TLS 종료 및 관리
로드밸런싱	로드밸런싱 제공	고급 로드밸런싱 및 트래픽 제어 기능 제공	로드밸런싱 제공

설명

오픈시프트 라우터 경우에는 Istio에서 제공하는 AB 및 Canary를 지원한다.

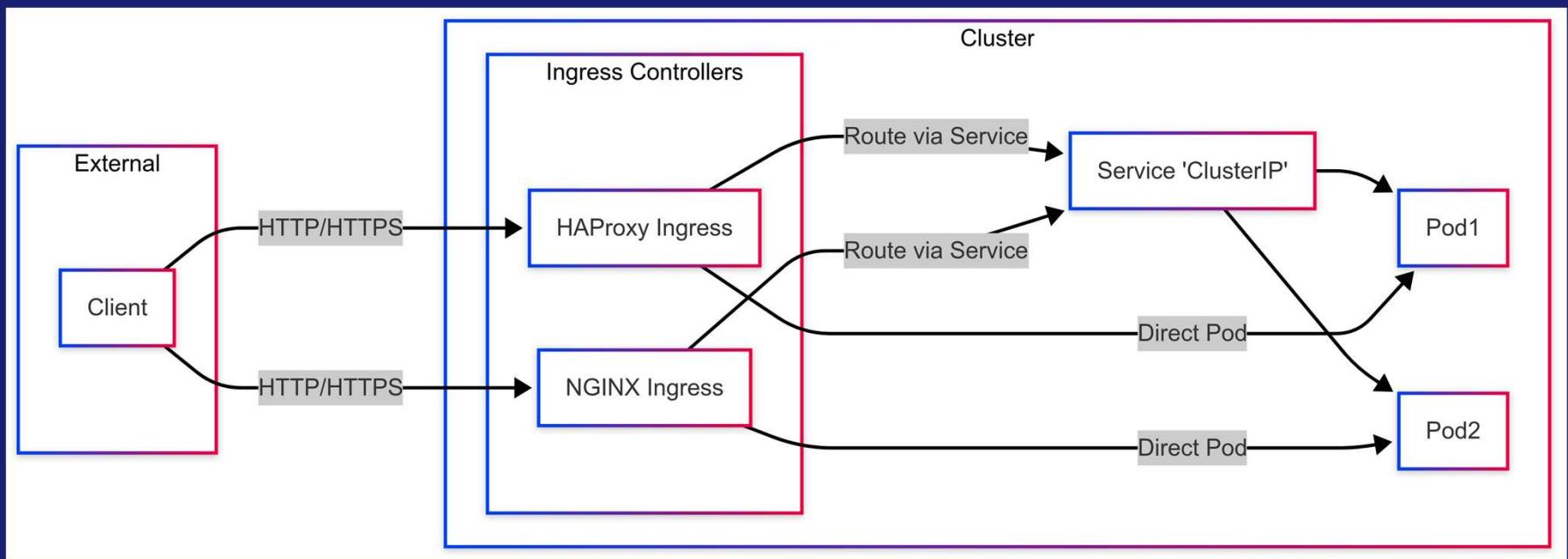
항목	Ingress Controller	Istio Ingress Gateway	OpenShift Router
정책/보안	기본적인 인증/인가 기능 (어노테이션 활용)	mTLS, 인증 고급 보안 정책 적용 가능	OpenShift 보안 정책과 연동
서비스 메쉬 통합	주로 단독으로 사용되며, 서비스 메쉬와는 별개	Istio 서비스 메쉬 내의 게이트웨이로 동적 구성 지원	OpenShift 클러스터 내부에 통합되어 운영됨
모니터링/로깅	제한적 (별도의 도구가 필요)	텔레메트리 로깅 모니터링 기능 제공	OpenShift의 모니터링 시스템과 연동

NGINX

INGRESS

2025-03-16

설명



설명

쿠버네티스에서 일반적으로 많이 사용하는 INGRESS서비스는 다음과 같다.

Ingress Controller 개발 주체 / 커뮤니티	주요 특징	확장성 및 추가 기능	사용 사례 / 장점
NGINX Ingress Controller <small>Kubernetes 공식 리포지토리에서 관리 F5/NGINX Inc. (상용 버전)</small>	<ul style="list-style-type: none">- 비교적 오래되고 안정적인 프로젝트- 다양한 설정(애노테이션) 제공- TCP/UDP, gRPC 등 L4/L7 레벨 트래픽 처리 지원	<ul style="list-style-type: none">- 기본 인증, IP 화이트리스트, Lua 스크립팅(상용판) 등 풍부한 기능- Helm Chart, Operator 등 배포 방식 다양	<ul style="list-style-type: none">- 가장 널리 사용되는 Ingress Controller 중 하나- 커뮤니티 자료와 사례가 풍부
HAProxy Ingress <small>HAProxy Technologies / 커뮤니티</small>	<ul style="list-style-type: none">- 고성능 L4/L7 로드밸런싱- 다양한 헬스 체크 및 동적 설정 지원- Kubernetes 인프라와의 연동성(애노테이션 등 개선)	<ul style="list-style-type: none">- Lua 스크립트 확장 가능- TLS 종료, SNI, 인증 등 고급 기능 지원	<ul style="list-style-type: none">- HAProxy의 높은 성능과 안정성을 그대로 사용 대규모 트래픽 환경에서 효과적

설명

아래 서비스는 확장이 가능한 서비스이다.

Ingress Controller 개발 주체 / 커뮤니티	주요 특징	확장성 및 추가 기능	사용 사례 / 장점	
Traefik	Traefik Labs / 커뮤니티	- 라우팅 규칙을 선언적으로 설정("dynamic configuration") - 다양한 백엔드(에지, 클라우드)와 통합 - ACME/Let's Encrypt 자동 TLS	- 인그레스 라우팅 외에도 Service Mesh(Traefik Mesh) 가능 - 여러 프로토콜 지원(HTTP, TCP, UDP)	- 설정이 비교적 간단하고 UI/ 대시보드 제공 - DevOps / 마이크로서비스 환경에서 자주 사용
Kong Ingress Controller	Kong Inc. / 커뮤니티	- API 게이트웨이 기능을 결합한 인그레스 컨트롤러 플러그인(Plugin) 아키텍처로 인증, 관찰성, 변환 기능 제공	- 고급 API 관리 기능(Kong Gateway)과 연동 가능 - DB-less 모드로 간단 배포 가능	- API 중심 아키텍처, 마이크로서비스 환경에서 강력 인증/권한 부여 등 API Gateway 기능이 필요한 경우 적합
Istio Ingress Gateway	Istio 프로젝트 / 커뮤니티	- Envoy Proxy 기반 Service Mesh 환경에서 인그레스 역할 담당 - 세밀한 트래픽 제어, 정책 적용 가능	- Istio 설치가 필요(상대적으로 복잡) - mTLS, 라우팅 규칙 등 Service Mesh 기능과 결합	- Istio Mesh를 이미 사용 중인 환경 - L7 레벨 세밀한 트래픽 제어와 정책 적용이 필요한 경우

설치

설치가 되어 있지 않으면, 간단하게 설치가 가능하다. 모든 패키지는 가급적이면 HelmChart기반으로 설치 및 관리 한다. 쿠버네티스 기본 패키지 관리자는 Helm이다.

```
# helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
# helm repo update
# helm install nginx-ingress ingress-nginx/ingress-nginx --namespace
nginx-ingress --create-namespace
NAME: nginx-ingress
LAST DEPLOYED: ...
NAMESPACE: ingress-nginx
STATUS: deployed
```

구성

인프라가 올바르게 구성이 되어 있다면, 다음과 같이 인그레스 서비스를 구성한다. 매우 기본적인 기능만 사용한다.

```
# vi web-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web-ingress
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
```

구성

임시 도메인은 example.com으로 선언, 문제가 없으면 접근이 가능하다.

```
spec:  
  rules:  
    - host: example.com  
      http:  
        paths:  
          - path: /  
            pathType: Prefix  
        backend:  
          service:  
            name: web-service  
          port:  
            number: 80
```

구성

올바르게 구성이 되었는지 확인한다.

```
# kubectl apply -f web-ingress.yaml
# kubectl get -f web-ingress.yaml
NAME      CLASS      HOSTS      ADDRESS      PORTS      AGE
web-ingress  <none>    example.com  203.0.113.10  80        1m
# kubectl get svc -n ingress-nginx
ingress-nginx  nginx-ingress-ingress-nginx-controller      LoadBalancer  11.88.128.115  192.168.0.242
# kubectl get ingress
NAME      CLASS      HOSTS      ADDRESS      PORTS      AGE
web-ingress  <none>    example.com  80        4m14s
```

LOADBALANCER

METALLB

2025-03-16

설정(CM)

설치 시, 로드밸런서 설정 변경이 필요하다. IPPool/L2에 대한 설정이 필요하다. 일반적으로 아래처럼 수정 및 변경 한다. 설치는 이미 HELM에서 진행했기에 설정만 변경한다.

```
# vi metallb-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
```

설정(CM)

앞에 내용 이어서...

```
data:  
  config: |  
    address-pools:  
    - name: default  
      protocol: layer2  
      addresses:  
      - 192.168.0.240-192.168.0.250  
    l2-advertisements:  
    - ip-address-pools:  
      - lb-test-ip-pool  
      interface: "ens7"  
# kubectl apply -f metallb-config.yaml
```

설정(L2)

L2설정은 아래와 같은 방식으로 권장한다. 특정 인터페이스를 원하는 경우 아래처럼 명시한다.

```
# vi metallb-l2-advertisement.yaml
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: default
  namespace: metallb-system
spec:
  nodeSelectors:
    - matchLabels:
        kubernetes.io/hostname: node1.example.com
    - matchLabels:
        kubernetes.io/hostname: node2.example.com
  ipAddressPools:
    - lb-test-ip-pool
  interfaces:
    - ens7
# kubectl apply -f metallb-l2-advertisement.yaml
```

설치(Chart)

Chart를 통해서 변경을 원하는 경우, values.yaml 파일을 통해서 수정 및 관리를 권장한다. 하지만 제한 사양이 있다.

```
# vi values.yaml
fullnameOverride: metallb
rbac:
  create: true
controller:
  image:
    repository: metallb/controller
    tag: v0.13.7
resources:
  limits:
    cpu: 100m
    memory: 128Mi
  requests:
    cpu: 50m
    memory: 64Mi
```

설치(Chart)

위의 내용 이어서...

```
speaker:  
  image:  
    repository: metallb/speaker  
    tag: v0.13.7  
  resources:  
    limits:  
      cpu: 100m  
      memory: 128Mi  
    requests:  
      cpu: 50m  
      memory: 64Mi
```

설치(LEGACY)

버전에 따라서 다르지만, "configInLine"이 지원이 되는 경우도 있다. 0.13버전 이후부터는 이 방식은 지원하지 않는다. 이 부분은 앞에서 사용한 ConfigMap기반으로 구성해야 한다.

```
configInline: |
  address-pools:
    - name: default
      protocol: layer2
      addresses:
        - 192.168.1.240-192.168.1.250
  l2-advertisements:
    - ip-address-pools:
        - default
        # 특정 인터페이스
        # interface: "eth0"
```

설치(VALUE)

기존에 설치 된 로드밸런서에 업데이트 및 확인한다.

```
# helm upgrade metallb metallb/metallb -f values.yaml -n metallb-system --create-namespace
# kubectl get pods -n metallb-system
NAME                  READY   STATUS    RESTARTS   AGE
speaker              1/1     Running   0          2m
controller           1/1     Running   0          2m
# kubectl get -f metallb-l2-advertisement.yaml
# kubectl get -f metallb-config.yaml
```

확인(OSP)

오픈스택에서 Octavia를 사용하지 않는 경우, PORT기반으로 아이피 할당 후 MetalLB에서 사용한다. 일반적인 베어메탈 경우에는 별도로 선택할 내용은 따로 없다.

The screenshot shows the OpenStack interface for port creation and a terminal window displaying Kubernetes service information.

Port Configuration:

이름	• 192.168.0.240	fa:16:3e:a8:d8:8e	연결 해제됨	Down	True
lb-ip-240					

Terminal Output:

```
# kubectl get svc
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
nginx-lb   LoadBalancer   11.88.223.28    192.168.0.240   80:30760/TCP   23m
nginx-lb-2  LoadBalancer   11.88.184.114   192.168.0.241   80:31822/TCP   3s
nginx-np   NodePort     11.88.176.235   <none>          80:31526/TCP   81m
# arp -e
192.168.0.240          (incomplete)
192.168.0.241          (incomplete)
```

프로젝트 준비

BLOG

2025-03-16

프로젝트

간단하게 프로젝트를 진행하기 위해서 강사와 함께 진행한다.

INFRA IN KUBERNETES

GOGS

REGISTRY

2025-03-16

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.

GIT-SERVER

Gogs

2025-03-16

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/
```

REGISTRY

docker-registry

2025-03-16

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
EOF
```

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. 대시보드는 옵션이기 때문에, 굳이 설치를 안해도 된다.

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

CI/CD

TEKTON

GIT SERVER

IMAGE REGISTRY

2025-03-16

TEKTON

CNCF

설명

테크톤의 기능은 다음과 같다.

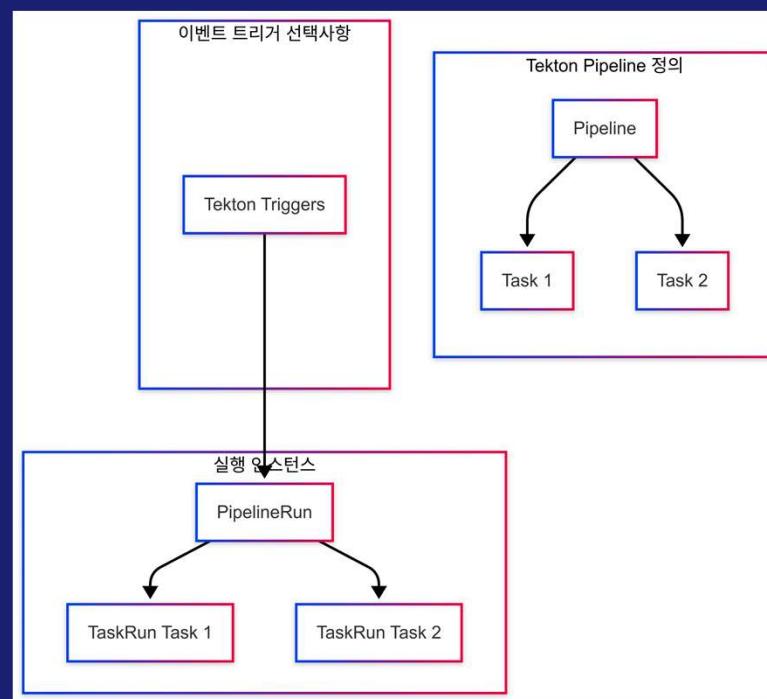
기능/리소스	설명	주요 특징 및 용도
Task	하나의 작업 단위로, 여러 스텝(컨테이너 실행)을 정의하여 특정 작업(예: 빌드, 테스트)을 수행	재사용 가능하고, 독립적으로 정의되어 단일 작업 실행
Pipeline	여러 Task들을 순차적 또는 병렬로 연결하여 전체 워크플로우를 구성하는 파이프라인	복잡한 CI/CD 프로세스를 선언적으로 정의
TaskRun	Task의 실행 인스턴스로, 특정 Task를 실행할 때 생성됨	Task 실행 결과 및 상태 추적

설명

테크톤의 기능은 다음과 같다.

기능/리소스	설명	주요 특징 및 용도
PipelineRun	Pipeline의 실행 인스턴스로, 파이프라인 내 여러 TaskRun을 포함하여 전체 워크플로우 실행	파이프라인 실행 상태, 결과 및 로그 수집
Tekton Triggers	이벤트 기반으로 PipelineRun을 자동으로 생성하여 파이프라인을 트리거하는 기능	Git 웹훅, CI 이벤트 등을 기반으로 자동 실행

설명



설치

```
# helm repo add tektoncd https://tektoncd.github.io/charts
"tektoncd" has been added to your repositories

# helm repo update
... (repository 업데이트 로그 출력)

# helm install tekton-pipelines tektoncd/tekton-pipelines -n tekton-pipelines --create-namespace
NAME: tekton-pipelines
LAST DEPLOYED: Wed Mar  9 12:34:56 2025
NAMESPACE: tekton-pipelines
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

설치

```
# kubectl get pods -n tekton-pipelines
```

NAME	READY	STATUS	RESTARTS	AGE
tekton-pipelines-controller-xxxxxxxxxx	1/1	Running	0	2m
tekton-pipelines-webhook-xxxxxxxxxx	1/1	Running	0	2m

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
```

CLUSTER-ROLE

ROLE 구성은 ClusterRole기반으로 구성이 가능하다. 특정 네임스페이스에서만 선언도 가능하다. 전체 클러스터에 선언 시, 다음과 같이 가능하다.

```
# 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
```

2025-03-16

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
# kubectl apply -f tekton-rbac/
# kubectl get -f tekton-rbac/
```

ROLE

values.yaml에 선언을 원하는 경우 아래와 같이 작성한다.

```
# vi values.yaml
rbac:
  create: true
  roles:
    controller:
      rules:
        - apiGroups: []
          resources: ["pods", "services", "endpoints"]
          verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
    webhook:
      rules:
        - apiGroups: []
          resources: ["configmaps"]
          verbs: ["get", "list", "watch"]
```

ROLE

적용은 다음과 같이 한다.

```
# helm repo add tektoncd https://tektoncd.github.io/charts
# helm repo update
# helm install tekton-pipelines tektoncd/tekton-pipelines -n tekton-
pipelines --create-namespace -f values.yaml
# kubectl get clusterrole,clusterrolebinding -n tekton-pipelines
# kubectl get pods -n tekton-pipeline
# kubectl get clusterrole,clusterrolebinding -n tekton-pipeline | grep
tekton
```

DB PIPELINE

TEKTON

2025-03-16

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"
```

PIPELINE(db)

```
taskRef:  
  kind: Task  
  name: git-clone  
  
workspaces:  
  - name: output  
    workspace: source  
  
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

TEKTON

2025-03-16

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
```

2025-03-16

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

2025-03-16

VIRTUALIZATION

KUBE-VIRT

2025-03-16

모니터링

Prometheus

Grafana

2025-03-16

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
```