CKA 101

CKA

2025-04-18

소개

과정

강사

강사

강사

이름: 최국현

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

사이트: tang.dustbox.kr

언제든지 질문 및 요청 환영 입니다.

목차

목차

목차

목차

소개

랩

소개

이 과정은 Linux Foundation CKA 대비 과정입니다.

내용은 다음과 같이 갱신 및 추가가 되었습니다.

1. 2024년도 2분기 업데이트
2. 2024년도 4분기 마지막 업데이트

소개

- 모든 과정은 랩 위주로 진행이 됩니다. 천천히 따라오시면 됩니다. :)
- 시험이 나오는 부분은 제목에 ***표시**가 되어 있습니다.
- 그 이외 나머지 부분은 시험 응시 시, 도움이 되는 부분 입니다.

랩

모든 랩은 오픈스택에서 실행이 됩니다. 실제 시험 환경은 Xfce기반으로 구성이 되어 있습니다. 웹 브라우저 및 터미널 사용이 가능 합니다.

다만, 웹 사이트는 가급적이면 다음과 같은 세션만 사용합니다.

- Kubernetes Blog
- Kubernetes Docs

랩은 직접 오프라인 기반, **버추얼 박스/VMware Workstation/Hyper-V기반**으로 구성하셔도 됩니다. 오프라인 맵 구성에 대해서는 따로 다루지 않습니다.

대비 1

클러스터 살펴보기

클러스터 생성 및 구성

클러스터 구성은 깃헙에서 쉘 스크립트 내려받기 후, 구성한다. 이 강의 오픈스택 기반으로 진행하기 때문에 오픈스택 기반으로 구성 및 진행한다.

다른 하이퍼바이저 프로그램을 사용하는 경우, 다른 하이퍼바이저 기반으로 구성하여도 괜찮다. 자원이 부족한 경우, 하나의 클러스터 기반으로 구성하여도 된다.

이 부분은 시험하고 상관없다. 다만, 원활한 연습을 위해서 minikube, kind가 아닌 직접 클러스터를 한 개 이상 구성하여 테스트 하는것을 권장한다.

클러스터 살펴보기

시험에서는 여러 클러스터(최소 2개 이상)로 구성이 되어 있다. 랩에서는 두 개의 클러스터 기반으로 사용한다. 클러스터 구성은 다음과 같이 되어 있다. 권장은 **3개 클러스터 기반**으로 연습을 권장한다.

클러스터1

```
node1:controller:192.168.10.20
node2:compute:192.168.10.21
node3:compute:192.168.10.22
```

클러스터2

```
node1:controller:192.168.10.30
node2:compute:192.168.10.31
node3:compute:192.168.10.32
```

사용자 로그인 및 클러스터 확인하기

모든 클러스터는 kubernetes-admin계정 기반으로 사용한다. 클러스터에 별도로 추가된 계정은 존재하지 않는다.

- `cluster1: kubernetes-admin`
- `cluster2: kubernetes-admin`

클러스터 구성 시, 클러스터 이름을 cluster-a, cluster-b로 생성 및 구성한다.

각 클러스터의 자원 살펴보기

클러스터가 올바르게 동작 및 구성이 되는지 확인한다.

```
# kubectl get node
# kubectl get pod
# kubectl config get-contexts
```

대비 2

살펴보기

세션 설명

이 세션은 강사가 시스템을 살펴보면 훑어보는 과정 입니다. 진행을 같이 하시거나 혹은 보시기만 하여도 상관 없습니다. ☺

ETCD와 쿠버네티스 관계

살펴보기

ETCD

쿠버네티스에서 생성되는 자원 정보 및 설정은 ETCD서버에 저장이 된다. 초기 쿠버네티스는 ETCD서버가 호스트에서 동작 하였지만, 현재는 kubelet를 제외한 모든 서비스는 컨테이너 기반으로 동작한다.

```
# kubectl get pod -n kube-system
```

컨트롤러가 올바르게 구성이 되면, 모든 컨트롤러는 ETCD를 하나씩 가지고 있다. 사용자가 원하는 경우, ETCD를 늘리거나 혹은 외부로 구성이 가능하다.

현재 쿠버네티스 아키텍처에서는 ETCD는 외부보다는 클러스터의 컨트롤러에 구성을 권장한다. ETCD에 접근하기 위해서는 ETCD컨테이너에서 제공하는 etcdctl명령어나 혹은 HOST에 etcdctl를 설치하여 조회가 가능하다.

ETCD

아래 명령어로 ETCD에 구성된 디렉터리 목록을 가져온다. 가져온 다음에 JSON파일을 cat명령어로 조회 및 출력 한다.

```
# export ADVERTISE_URL=https://192.168.10.20:2379
# kubectl exec etcd-cluster1-node1.example.com -n kube-system -- sh -c \
"ETCDCTL_API=3 etcdctl \
--endpoints $ADVERTISE_URL \
--cacert /etc/kubernetes/pki/etcd/ca.crt \
--key /etc/kubernetes/pki/etcd/server.key \
--cert /etc/kubernetes/pki/etcd/server.crt \
get \"\" --prefix=true -w json" > /root/etcd-kv.json

# for k in $(cat etcd-kv.json | jq '.kvs[].key' | cut -d '"' -f2); do echo $k | base64 --decode; echo; done;
```

ETCD

CoreDNS구성은 아래 etcdctl로 확인이 가능하다.

```
# KEY="/register/deployments/kube-system/coredns"

# kubectl exec etcd-cluster1-node1.example.com -n kube-system -- sh -c \
"ETCDCTL_API=3 etcdctl \
--endpoints $ADVERTISE_URL \
--cacert /etc/kubernetes/pki/etcd/ca.crt \
--key /etc/kubernetes/pki/etcd/server.key \
--cert /etc/kubernetes/pki/etcd/server.crt \
get \"$KEY\" -w json"
```

KUBECTL*

설명

KUBECTL

모든 자원은 kubectl명령어로 조회한다. 랩에서 자주 사용하는 자원을 간단하게 조회 한다.

```
# kubectl config get-context
# kubectl get pods -n kube-system
# kubectl get nodes
# kubectl get configmap -A
# kubectl get secret -A
# kubectl get storageclass -A
# kubectl get pv -A
# kubectl get pvc -A
# kubectl get replicaset
# kubectl get replicationControllers
```

KUBECTL

앞에서 나온 내용 계속.

```
# kubectl get daemonset
# kubectl get statefulset
```

kube-proxy*

kube-proxy역할 및 발생 가능한 장애

kube-proxy와 네트워크 그리고 커널 관계

KUBE-PROXY

kube-proxy는 상황에 따라서 필요할 수도 있고, 필요하지 않을 수도 있다. 이러한 이유로 K3S에서는 kube-proxy는 선택사항으로 제공한다. 이 이유는 다음과 같다.

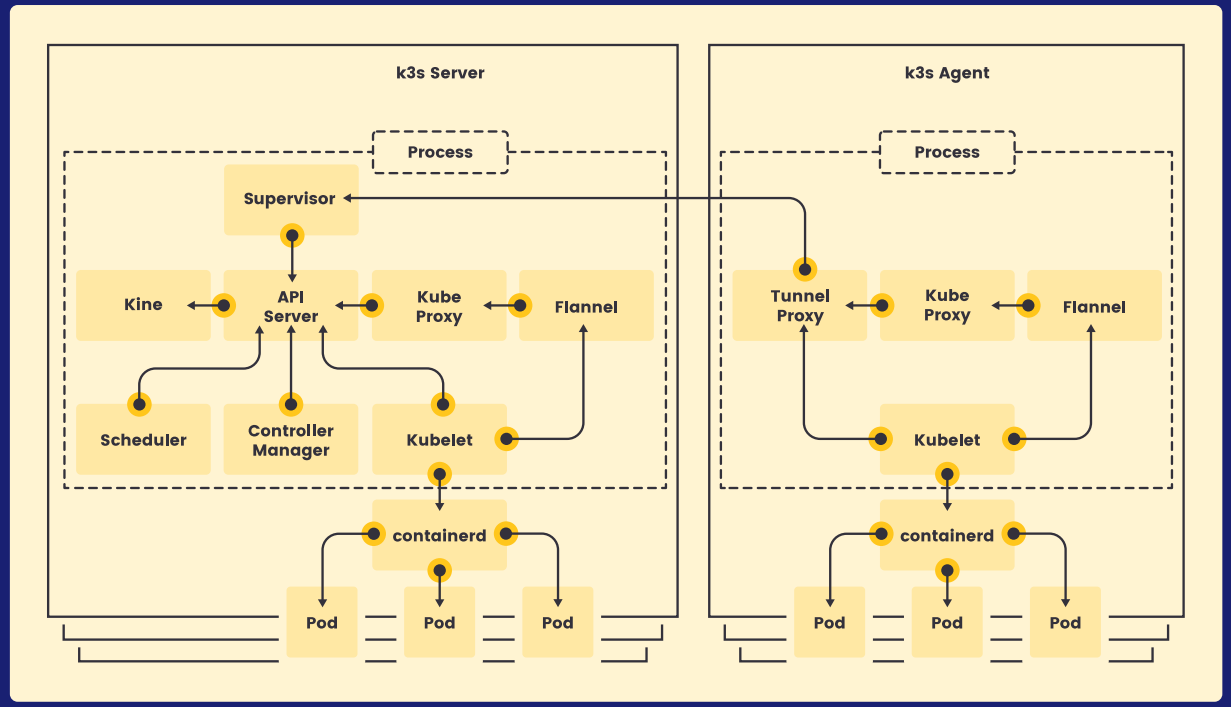
1. PROXY는 netfilter(iptables, nftables)를 통해서 라우팅 정책을 구성한다.
2. kubernetes-OVN를 사용하는 경우, 라우팅을 소프트웨어서 처리하기 때문에 PROXY가 필요하지 않는다.
3. K3S와 같은 아키텍처에는 CNI에서 PROXY기능을 구성 및 처리하기 때문에 필요하지 않는다.
4. k8s도 사용하는 CNI에 따라서 kube-proxy기능을 사용하지 않을 수 있다.

kube-proxy서비스는 Static POD이기 때문에, 제거가 되어도 kubelet에서 다시 재구성 한다.

```
# kubectl delete -n kube-system pod/kube-proxy-XYZ
```

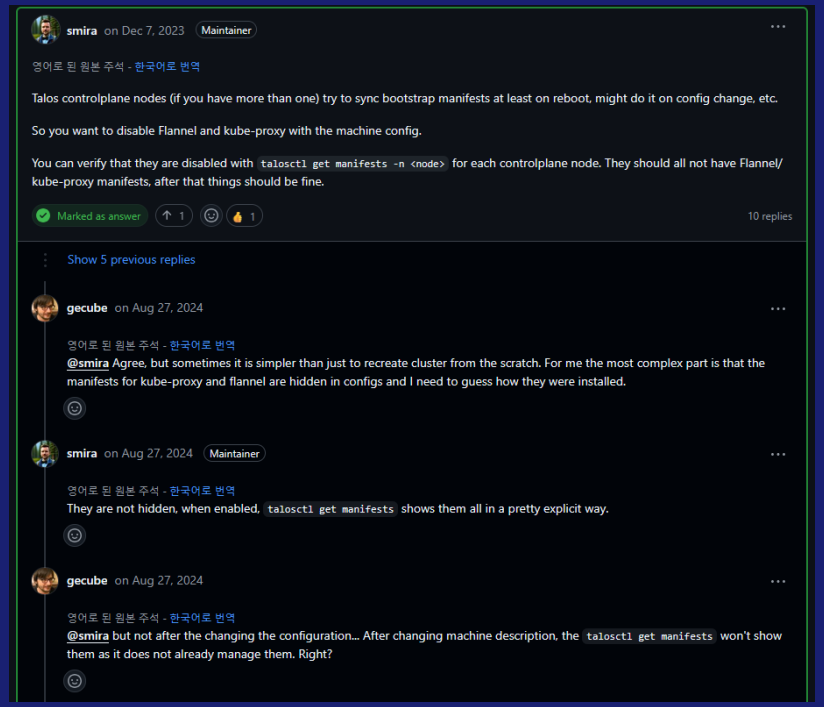
KUBE-PROXY

K3S 아키텍처 그림에는 kube-proxy가 존재한다. 하지만, 어떠한 CNI를 사용하냐 따라서 L2에서 통합. 대표적인 소프트웨어는 MetalLB나 Cilium과 같은 CNI네트워크 플러그인에서 L2/L3를 관리한다.



KUBE-PROXY

CNI에서 많이 사용하는 Flannel도 L2/L3를 Flannel로 완전히 대체가 가능하다. 하지만, Flannel 사용하는 경우, kube-proxy서비스를 그대로 유지한다.



https://github.com/siderolabs/talos/discussions/8037

KUBE-PROXY

CNI설치 후, kube-proxy제거는 다음과 같이 실행한다.

```
# kubectl delete daemonset -n kube-system kube-flannel
# kubectl delete daemonset -n kube-system kube-proxy
# kubectl delete cm kube-flannel-cfg -n kube-system
```

POD*

Static POD, General POD

POD개념 설명 및 구조

POD를 YAML기반으로 생성

Static POD, General POD

쿠버네티스 POD는 두 가지 유형으로 있다.

- 정적 POD 구성
- 일반적인 POD 구성

정적 POD는 쿠버네티스 클러스터 부트스트랩핑(bootstrapping) 시 사용하는 POD이다. 정적 POD는 kubelet서비스를 통해서 관리가 된다. 아래 디렉터리에 들어가면 kubelet에서 생성하는 POD확인이 가능하다.

```
# cat /etc/kubernetes/manifests
```

kubeadm명령어로 실행 하였을 때, 생성된 POD는 전부 Static POD이다.

https://kubernetes.io/docs/reference/setup-tools/kubeadm/implementation-details/

Static POD, General POD

일반 POD(General POD)는 사용자가 생성한 POD를 보통 이야기 한다. 하지만, 해당 POD는 사용자가 원하는 경우 Static POD으로 구성이 가능하다. 이 부분에 대해서는 시험에서 나오지는 않는다. 하지만, 몇몇 쿠버네티스 서비스는 kubelet기반으로 등록이 되어서 Static POD로 구성이 되는 경우가 있다.

자세한 내용은 아래 링크를 참조한다.

https://kubernetes.io/docs/tasks/configure-pod-container/static-pod/

```
[control-plane] Creating static Pod manifest for "kube-apiserver"
[control-plane] Creating static Pod manifest for "kube-controller-manager"
[control-plane] Creating static Pod manifest for "kube-scheduler"
```

NAMESPACE

쿠버네티스에서 namespace, 커널의 namespace

namespace 생성 및 관리

NAMESPACE*

네임스페이스는 두 가지 영역에서 제공이 된다.

1. 리눅스 커널에 제공하는 네임스페이스
2. 쿠버네티스에서 제공하는 네임스페이스

리눅스 커널에서 제공하는 네임스페이스는 커널 자원과 애플리케이션 자원을 연결 시 사용한다. 네임스페이스는 실제로 존재하지 않으며, 영역(유형)과 이름을 제공하는 자원이다. 제공하는 범위는 커널 버전마다 다르며 현재는 다음과 같은 영역을 제공하고 있다.

2025년 기준으로 아래 링크에서 확인이 가능하다.

https://docs.kernel.org/admin-guide/namespaces/compatibility-list.html

NAMESPACE

앞서 이야기 하였지만, 커널 버전별로 지원하는 범위 및 기능이 다르다. 컨테이너나 혹은 가상머신을 리눅스 커널 기반으로 사용 시, 가급적이면 최신 커널을 사용하여 개선된 기능 및 성능을 사용하도록 유지한다.

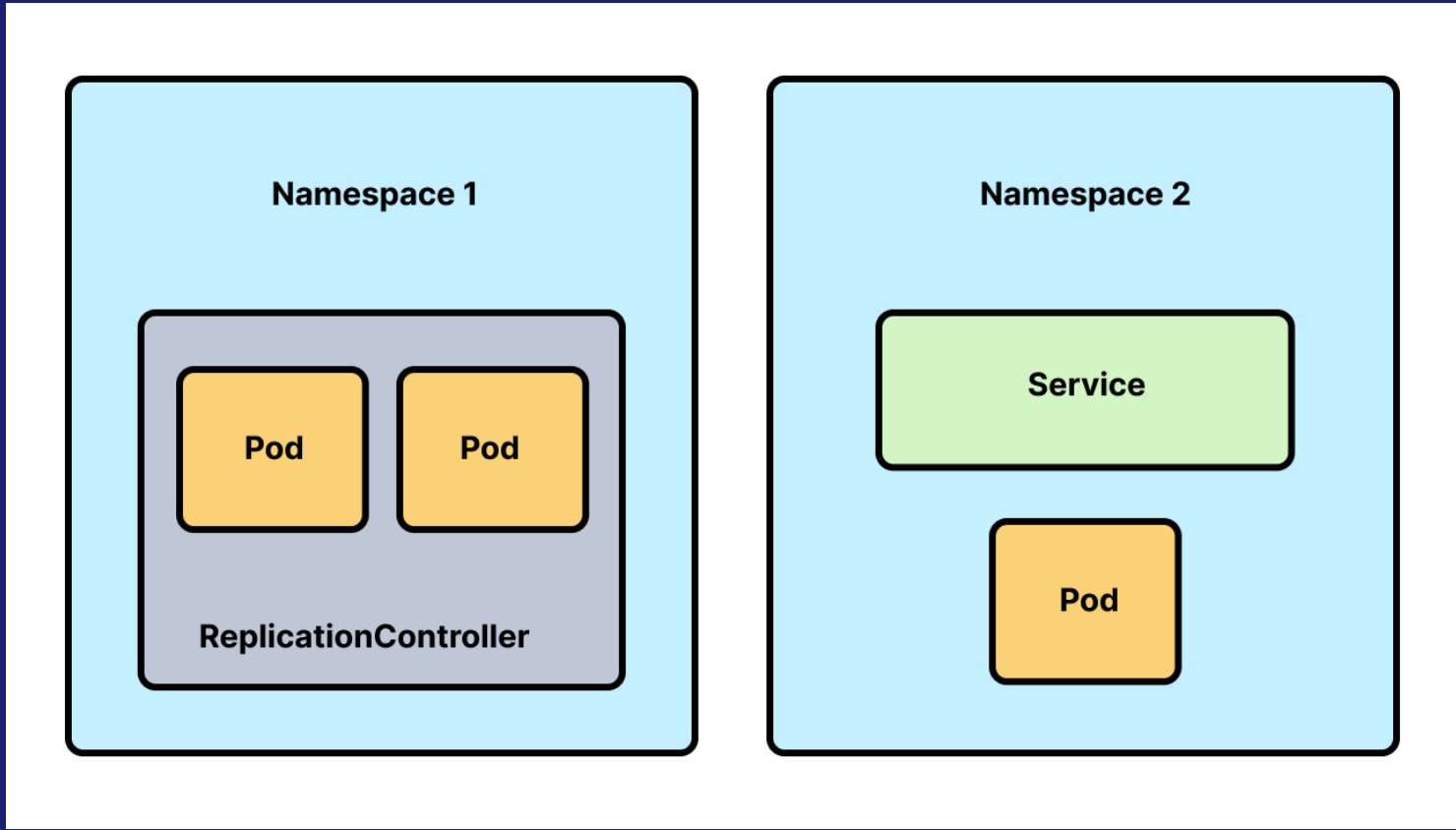
	UTS	IPC	VFS	PID	User	Net
UTS	X					
IPC		X	1			
VFS			X			
PID		1	1	X		
User		2	2		X	
Net						X

NAMESPACE

쿠버네티스에서 제공하는 네임스페이스도 거의 동일한 기능을 한다. 다만, 차이점은 쿠버네티스에서 제공하는 자원을 쿠버네티스 클러스터에서 추상적으로 격리 및 분리를 한다.

본래 네임스페이스는 타 네임스페이스에서 접근이 되지 않지만, 최근 쿠버네티스는 타 네임스페이스에서 접근이 가능하도록 기능을 지원 및 제공하고 있다. 하지만, 모든 자원이 **N2N(NAMESPACE TO NAMESPACE)**를 제공하지 않는다.

NAMESPACE


Namespace 1
Pod
Pod
ReplicationController
Namespace 2
Service
Pod

대비 3

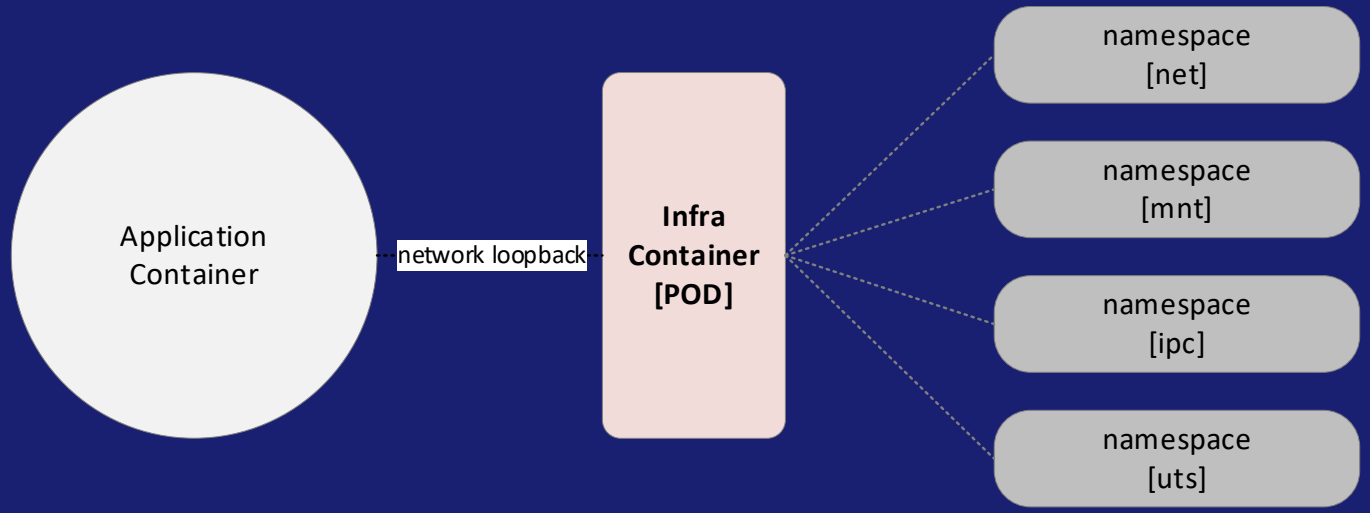
쿠버네티스에서 제공하는 복제 서비스

단일 컨테이너

POD+CONTAINER

POD/CONTAINER

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



POD/CONTAINER

컨테이너는 애플리케이션이 실행이 되는 영역이며, POD는 실행되는 애플리케이션 자원을 관리를 한다. POD는 개념적으로 부르는 이름이며, 시스템 영역에서는 POD를 Infra-container라고 부른다. infra-container는 무엇이 되었든, 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

/PAUSE

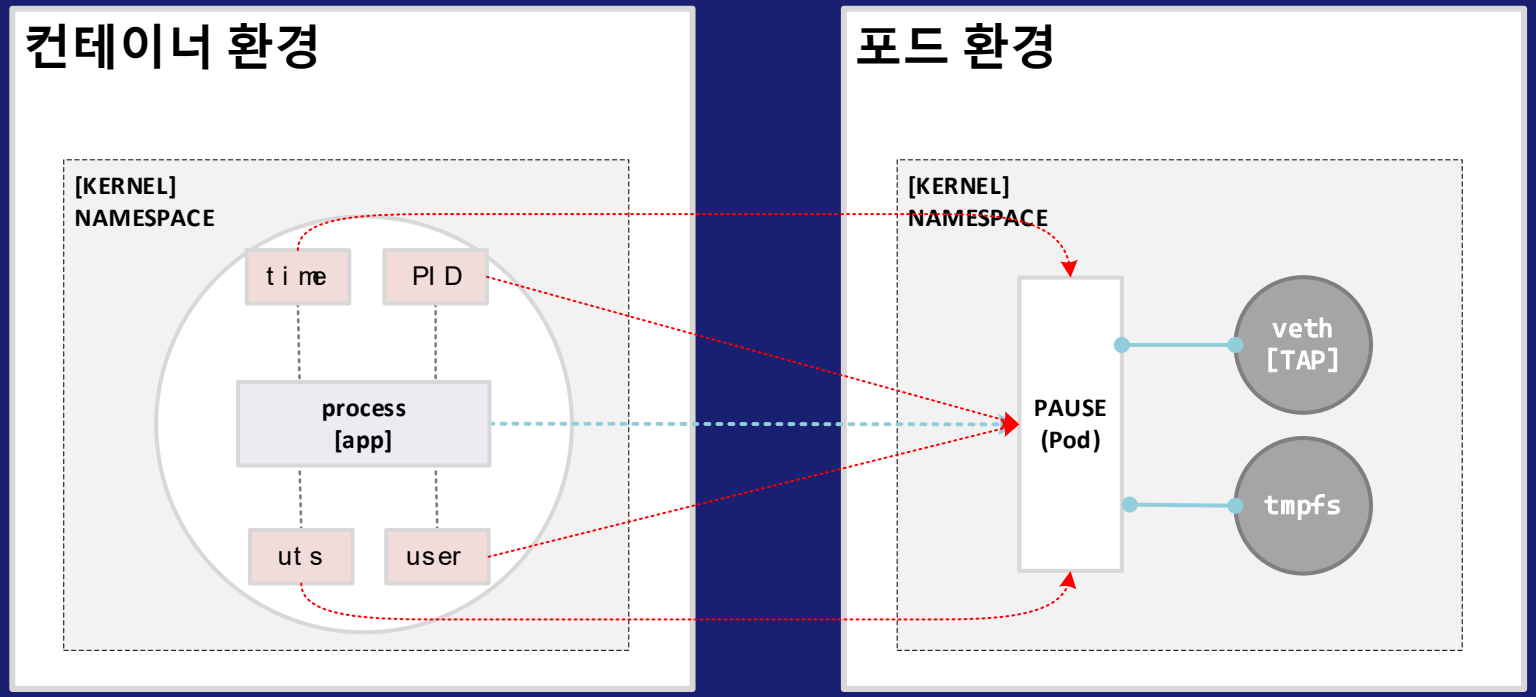
```
if (getpid() != 1)
   /* Not an error because pause sees use outside of infra containers. */
   fprintf(stderr, "Warning: pause should be the first process\n");

 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;
```

/PAUSE

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

POD

POD기반으로 애플리케이션을 생성하면 아래와 같이 구성된다.



POD생성

단일 컨테이너를 POD에 연결 및 생성한다. 아래 명령어로 실행한다.

```
# kubectl run --image=nginx nginx-1
```

생성 된 POD를 확인하기 위해서 다음과 같이 명령어를 실행한다.

```
# kubectl get pod
```

제거를 위해서는 다음과 같이 명령어를 실행한다.

```
# kubectl delete pod/<POD_NAME>
```

POD생성

컨테이너 생성 시, 레이블 사용이 가능하다.

```
# kubectl run --image=nginx --label=app=nginx nginx-1
```

레이블 기반으로 생성된 POD는 다음과 같이 조회 및 삭제가 가능하다.

```
# kubectl get pod -l=app=nginx
# kubectl delete pod -l=app=nginx
```

랩

간단하게 POD를 생성한다. POD이름은 run-nginx라는 이름으로 네임스페이스 lab-run-nginx-pod에 생성한다. 다음과 조건으로 POD를 생성한다.

1. POD는 run-pod라는 네임스페이스에 생성한다.
2. POD의 이미지는 nginx를 사용하여 프로비저닝 한다.
3. POD는 올바르게 동작이 되어야 하며, **`kubectl get pod`**명령어로 조회 시, "running"이라고 상태가 출력이 되어야 한다.

멀티 컨테이너

설명

다중컨테이너 구성 및 사용 방법

초기화 컨테이너

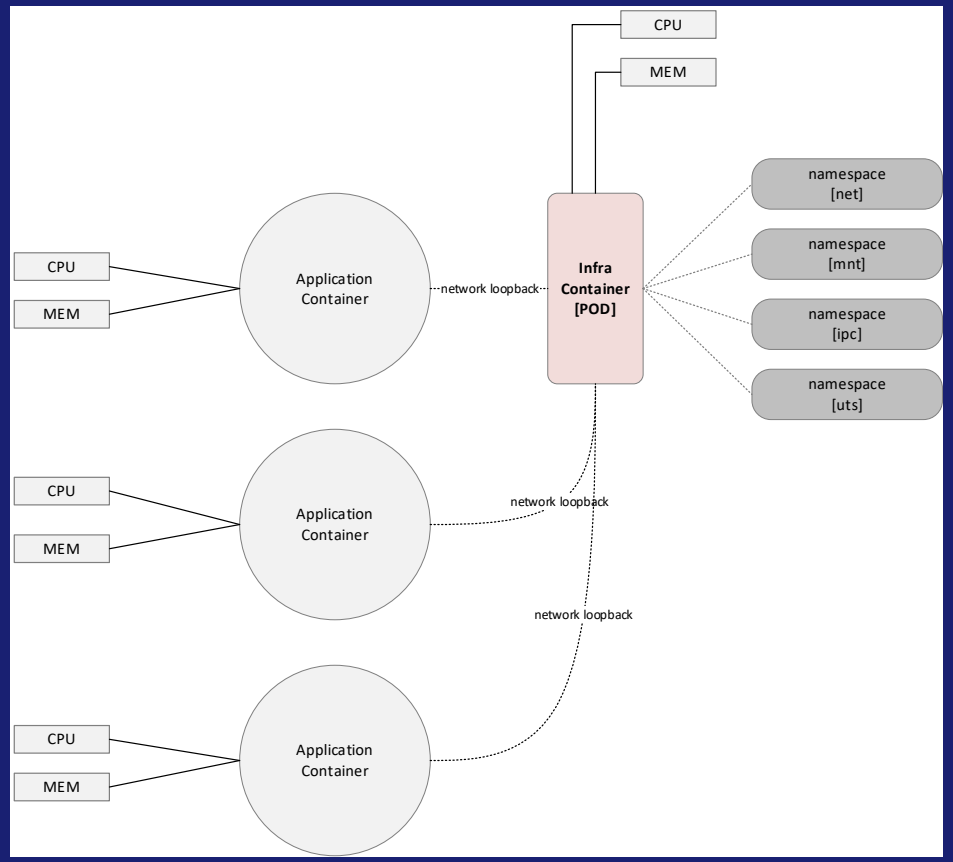
애플리케이션 힐링

다중 컨테이너 설명

다중 컨테이너는 한 개의 POD에 하나 이상의 컨테이너가 구성이 되어있다.

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

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



다중 컨테이너 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*

앞에 내용 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

다중컨테이너 생성 #1*

```
# 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
```

초기화 컨테이너(INIT CONTAINER)*

앞에서 사용했던 내용에 초기화 컨테이너를 추가한다. 초기화 컨테이너는 실제 컨테이너가 실행 전, POD영역에 특정 작업을 수행한다. 여기서는 미리 POD "shared-volume"이라는 디렉터리를 생성한다.

이 디렉터리가 생성이 되면, 특정 컨테이너에서 이를 확인하도록 한다. 여기서는 임시적으로 **`emptyDir:`**형태로 POD에 볼륨을 생성 후, 컨테이너에 연결한다.

초기화 컨테이너(INIT CONTAINER)*

초기화 컨테이너는 다음과 같이 구성한다.

```
# vi init-container.yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod-verify
spec:
  volumes:
    - name: shared-volume
      emptyDir: {}
```

POD에 비어 있는 볼륨을 하나 생성 및 할당한다.

초기화 컨테이너(INIT CONTAINER)

위의 내용 계속 이어서…

```
  initContainers:
    - name: init-container
      image: busybox
      command: ["sh"]
      args: ["-c", "mkdir -p /shared && touch /shared/init-done && ls -ld
/shared"]
      volumeMounts:
        - name: shared-volume
          mountPath: /shared
```

initContainer에서 디렉터리/파일을 임시적으로 생성한다.

초기화 컨테이너(INIT CONTAINER)

```
    containers:
    - name: verify-init-container
      image: busybox
      command: ["sh"]
      args: ["-c", "echo 'Verifying /shared directory contents:' && ls -la
/shared"]
      volumeMounts:
        - name: shared-volume
          mountPath: /shared
```

initContainer에서 올바르게 생성이 되었는지 볼륨 연결 후 명령어로 확인한다.

애플리케이션 힐링

애플리케이션 힐링을 제공하는 컨트롤러는 제한적으로 있다. POD경우에는 POD자체는 안되지만, POD의 특정 컨테이너가 종료가 되었을 때, 정책 기반으로 가능하다.

```
restartPolicy:
```

기본적으로 POD의 "restartPolicy:"는 "Never"로 되어 있다. 하지만, 컨테이너는 개별적으로 적용이 된다.

이름	설명
`always-containers`	항상 컨테이너 재시작(Always)
`onfailure-containers`	오류 발생 시, 재시작(OnFailure)
`never-containers`	절대 재시작하지 않음(Never)

애플리케이션 힐링(컨테이너)

컨테이너가 문제가 발생 시, **restartPolicy:**를 통해서 재구성이 된다.

```
# vi restart-policy-container.yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: restart-policy-demo
spec:
  restartPolicy: Never
  containers:
    - name: always-container
      image: busybox
      command: ["sh"]
      args: ["-c", "echo 'This container will restart'; sleep 5; exit 1"]
```

애플리케이션 힐링(컨테이너)

위의 내용 계속 이어서…

```
 - name: onfailure-container
      image: busybox
      command: ["sh"]
      args: ["-c", "echo 'This container will restart on failure'; sleep 5;
exit 1"]
    - name: never-container
      image: busybox
      command: ["sh"]
      args: ["-c", "echo 'This container will never restart'; sleep 5; exit
1"]
# kubectl apply -f restart-policy-container.yaml
```

랩

아래 조건은 POD를 생성합니다.

1. 두 개의 컨테이너를 하나의 POD에 구성 합니다.
    1. 이미지는 nginx, mariadb를 사용 합니다.
    2. maraidb는 최소 한 개의 MARIADB_ROOT_PASSWORD선언이 필요 합니다.
    3. POD의 이름은 multipod으로 구성한다.
    4. 네임스페이스는 lab-multipod생성 후 사용한다.
    5. nginx 컨테이너의 이름은 www, mariadb는 mariadb로 선언한다.
2. 모든 컨테이너는 올바르게 동작해야 한다.
3. POD에 2/2으로 출력이 되어야 한다.

랩

command_pod에 다음과 같이 컨테이너를 생성한다. 생성된 POD는 특정 시간이 지나면 running에서 stop으로 변경이 된다.

1. POD의 이름은 command_pod으로 생성한다.
2. 첫 번째 컨테이너는 busybox를 사용하고, sleep 명령어로 15000초 동안 대기 하도록 한다. 컨테이너 이름은 sleep_container_1로 구성한다.
3. 두 번째 컨테이너는 alpine를 사용하고, top 명령어를 실행한다. 컨테이너 이름은 top_container_2로 구성한다.

쿠버네티스에서 제공하는 복제 서비스

ReplicaSet/ReplicationController*

Deployment*

DaemonSet*

StatefulSet*

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 **Deployment**와 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 구성한다.
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	가능	가능
레이블 셀렉터 지원	Set-based Selector 지원 (matchExpressions)	Only Equality-based Selector (matchLabels)
Pod 자동 복구	가능	가능
Deployment와 연동	사용 가능	사용 불가능
추천 여부	(권장) 최신 버전이며, Deployment와 함께 사용 가능	(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 Deployment의 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/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
```

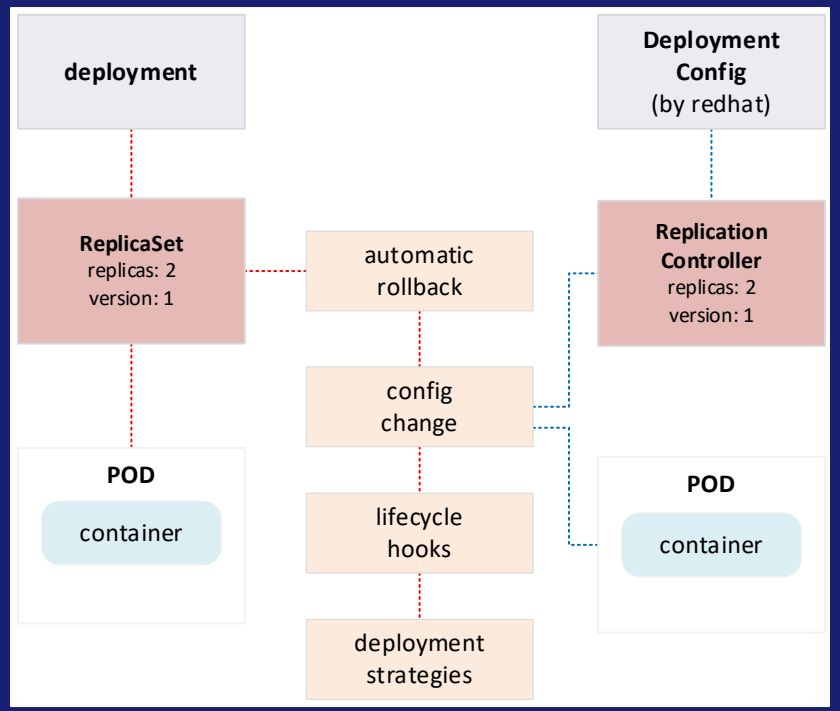
ReplicaController

위의 내용 계속…

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

Deployment*

Deployment는 **ReplicaSet**과 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*

Deployment구성은 다음과 같이 구성이 되어있다.

```
# 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*

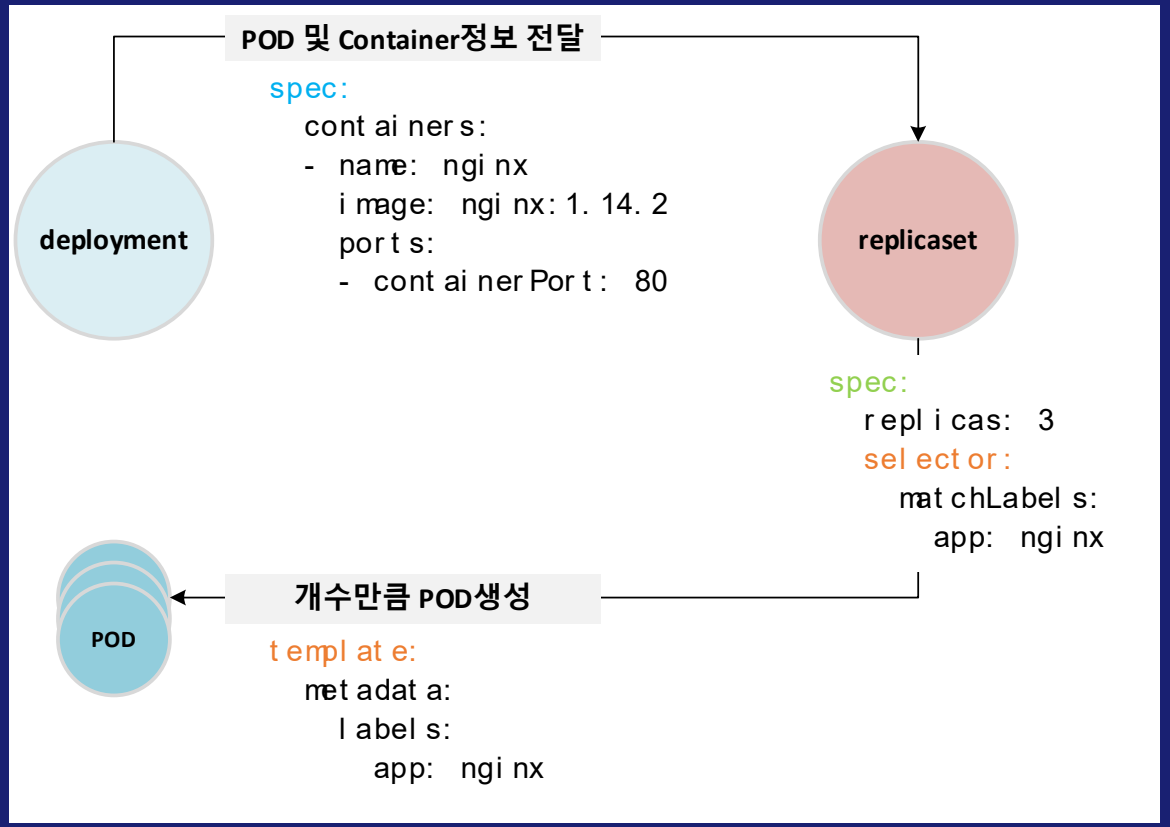
계속…

```
  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.

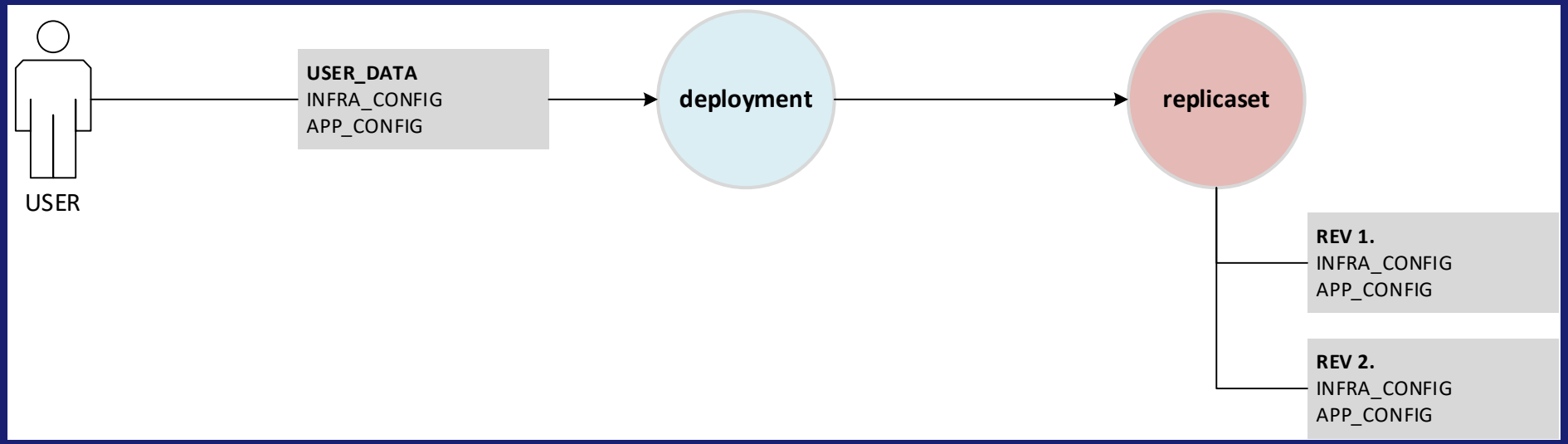


Deployment*

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
AGEdeployment.apps/nginx-deployment   3/3           3


# kubectl get replicaset
NAME                                    DESIRED    CURRENT
nginx-deployment-665fd86dbb                1          1
nginx-deployment-77d8468669                3          3
```

Deployment*


USER
USER_DATA
INFRA_CONFIG
APP_CONFIG
deployment
replicaset
REV 1.
INFRA_CONFIG
APP_CONFIG
REV 2.
INFRA_CONFIG
APP_CONFIG

DaemonSet*

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

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 컴퓨트 노드를 권장한다. 그렇지 않는 경우, 컴퓨트 노드에만 생성이 된다. 컨트롤러에도 생성하기 위에서는 톨러레이션(toleration)를 같이 구성해야 한다. 여기서는 톨러레이션을 구성하지 않는다. 순수하게 컴퓨트 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*

위의 내용 이어서 계속…

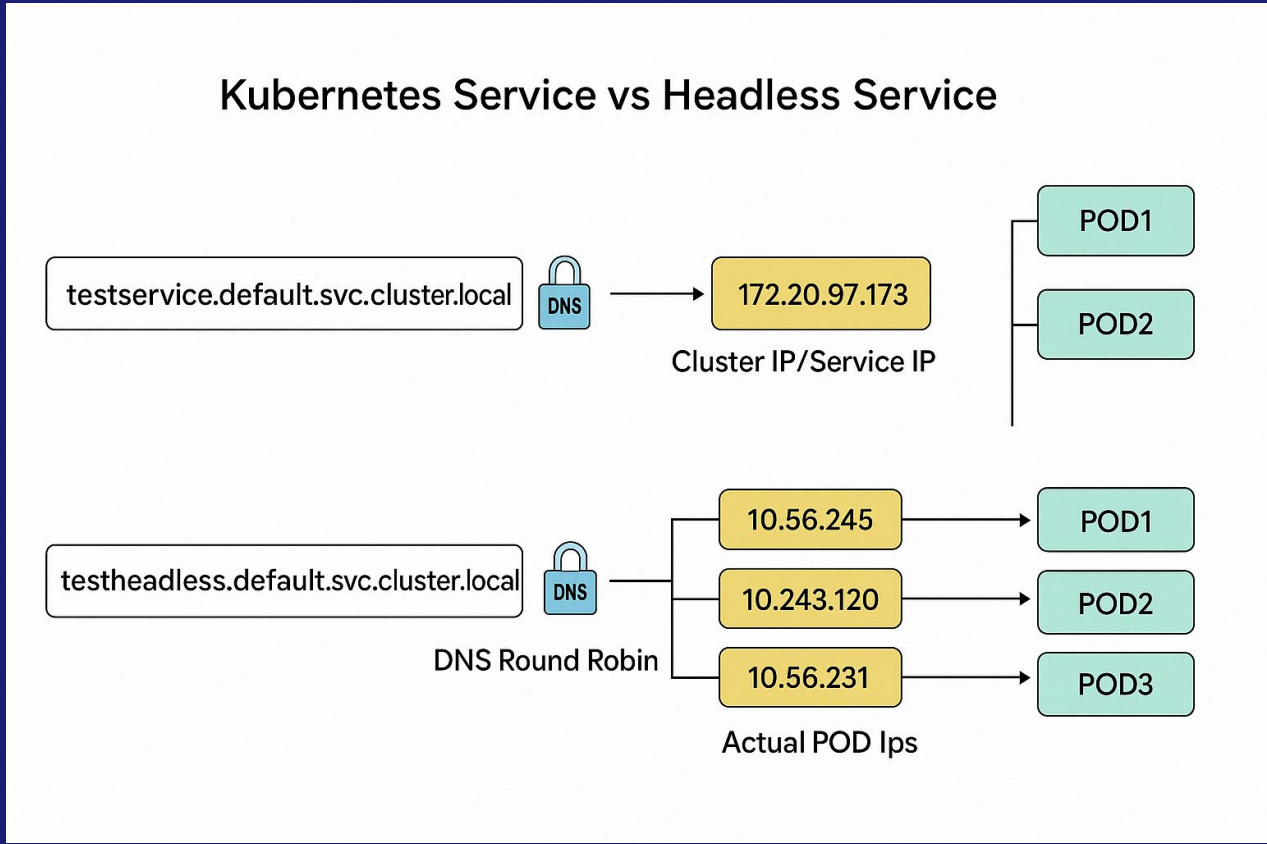
```
    terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
# kubectl apply -f daemonset.yaml
# kubectl get -f daemonset.yaml
```

StatefulSet*

StatefuleSet서비스는 일반적으로 3 Tiers와 기반의 레거시 애플리케이션 혹은 일반적인 C/S구조의 서비스 형태에 적용한다.

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

StatefulSet*



StatefulSet*

StatefulSet은 다음과 같이 작성한다. 접근 및 테스트 하기 위해서 SVC를 먼저 생성한다.

```
# kubectl apply -f sts-svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
```

StatefulSet*

위의 내용 계속 이어서…

```
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*

위의 내용 계속 이어서…

```
 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
```

랩

클러스터 내 모든 노드에서 실행되어야 하는 로깅 에이전트 또는 모니터링 에이전트와 같이, 각 노드에 동일한 포드를 배포하는 DaemonSet 구성.

1. 이미지는 fluentd이미지 사용함
2. 버전은 fluentd:v1.14
3. 별도의 디스크(POD Disk)는 별도로 선언하지 않음
4. tolerations은 사용하지 않아도 되지만, 사용하면 controller, master에도 POD생성이 가능하도록 함

랩 코드

다음과 같이 작성한다.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: kube-system
  labels:
    app: fluentd
spec:
  selector:
    matchLabels:
      name: fluentd
```

랩 코드

앞에 내용 계속 이어서…

```
  template:
    metadata:
      labels:
        name: fluentd
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        operator: Exists
        effect: NoSchedule
      containers:
      - name: fluentd
        image: fluent/fluentd:v1.14
```

랩

StateFulSet자원을 생성한다. 생성 시 다음과 같은 조건으로 생성한다.

1. headless서비스를 lab-sts-headless-nginx으로 생성한다.
2. 서비스에 RoundRobin서비스를 lab-sts-nginx으로 생성한다.
    - 이 서비스는 로드밸런서를 통해서 구성한다.
    - 현재 MetalLB를 구성 전 이기에 <pending>으로 출력이 된다.
3. 네임스페이스를 lab-sts-nginx를 생성한다.

랩 코드

lab-sts-headless-nginx.yaml는 아래와 같이 작성한다.

```
apiVersion: v1
kind: Service
metadata:
  name: lab-sts-headless-nginx
  namespace: lab-sts-nginx
spec:
  clusterIP: None  # ← 헤드리스 설정
  selector:
    app: nginx
  ports:
  - port: 80
    targetPort: 80
```

랩 코드

lab-sts-nginx.yaml는 아래와 같이 작성한다.

```
apiVersion: v1
kind: Service
metadata:
  name: lab-sts-nginx
  namespace: lab-sts-nginx
spec:
  type: LoadBalancer
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

랩

특정 작업(예: 데이터베이스 마이그레이션, 백업 작업 등)을 한 번 실행하기 위해 Job을 작성.

1. hello-job이라는 이름의 Job YAML 파일을 작성하여, 단순하게 "Hello, Job!" 메시지를 출력하는 컨테이너를 실행하도록 구성
2. Job 생성 후 kubectl get jobs와 kubectl logs를 통해 작업의 완료 상태 및 로그를 확인

랩 코드

hello-job.yaml는 아래와 같이 작성한다.

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

랩

정해진 스케줄에 따라 주기적으로 실행되는 작업(예: 로그 정리, 주기적 백업 등)을 CronJob으로 작성.

1. hello-cronjob이라는 이름의 CronJob YAML 파일을 작성하여, 매 분마다 "Hello, CronJob!" 메시지를 출력하는 작업을 실행하도록 구성
2. 스케줄 표현(cron 표현식)을 이해하고, kubectl get cronjob 및 kubectl get jobs 명령어로 실행 내역 확인

랩 코드

hello-cronjob.yaml는 아래와 같이 작성한다.

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello-cronjob
spec:
  schedule: "*/1 * * * *"  # 매 분마다 실행
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            command: ["echo", "Hello, CronJob!"]
          restartPolicy: OnFailure
```

대비 4

서비스 설명

ClusterIP*

NodePort

Load Balancer*

서비스 설명

대비 4

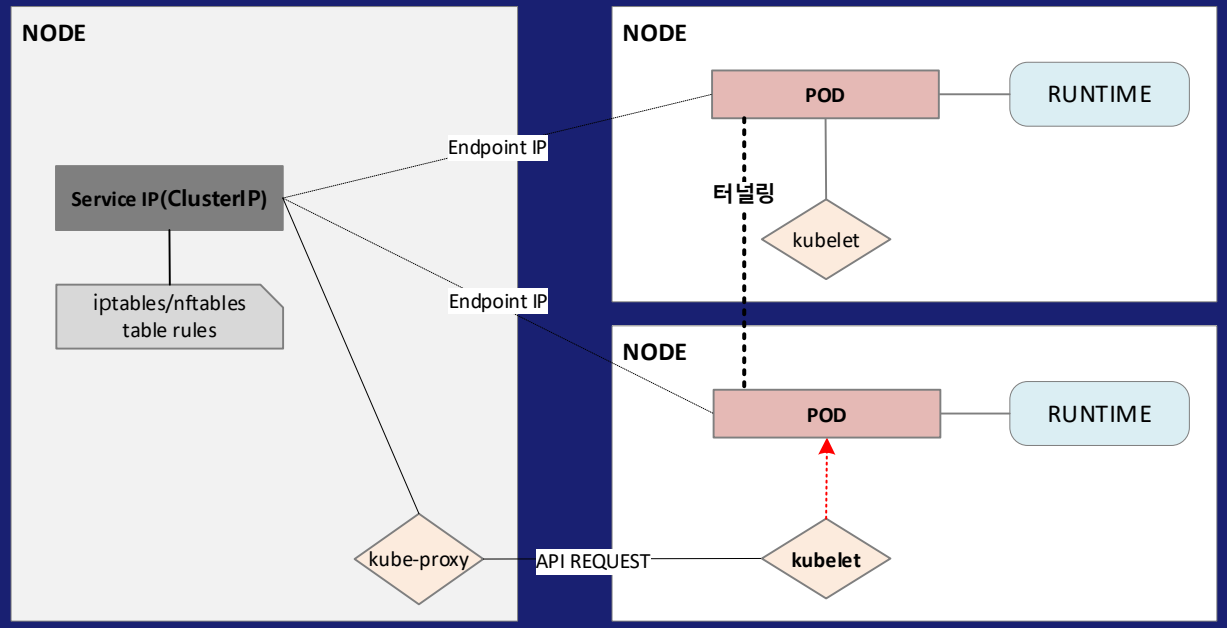
서비스 설명

쿠버네티스는 다음과 같은 서비스를 제공한다.

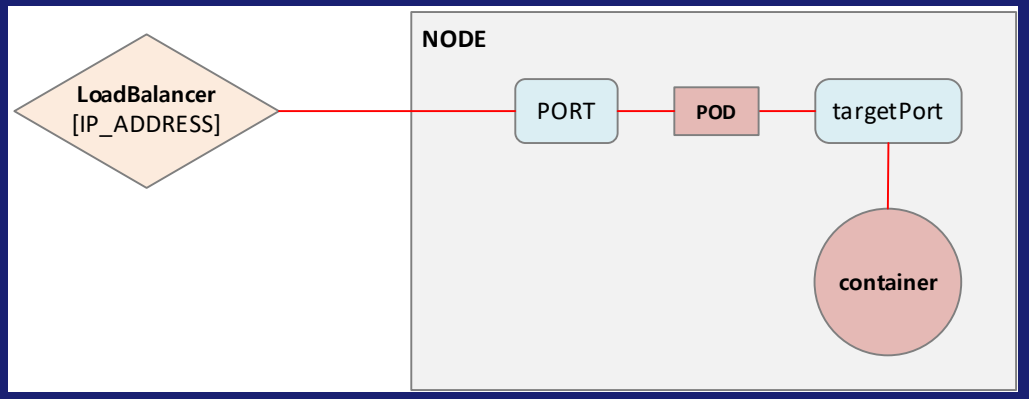
1. ClusterIP
2. **LoadBalancer(ExternalIP)***
3. ExternalName
4. **NodePort***

제일 많이 사용하는 서비스는 2, 4번 서비스 이며, 1번 경우에는 기본적으로 제공하는 클러스터 아이피, 즉 묶음 아이피 혹은 VIP라고 생각하여도 된다.

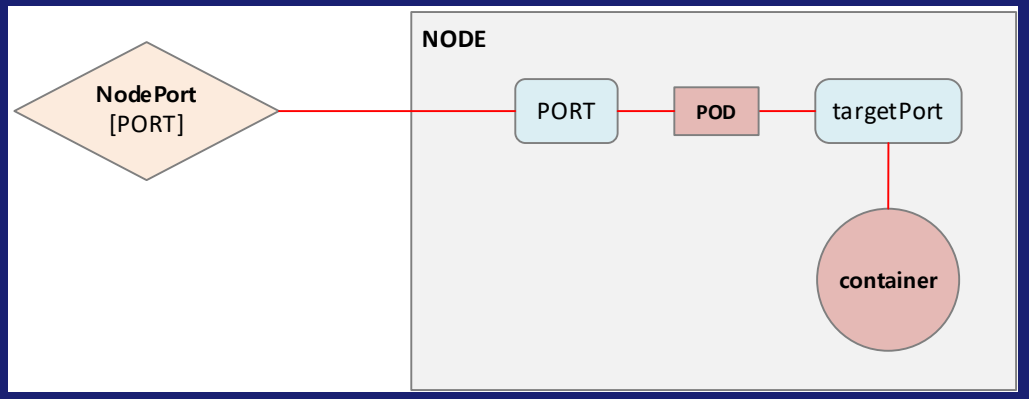
서비스 설명


NODE
Service IP(ClusterIP)
iptables/nftables
table rules
Endpoint IP
Endpoint IP
kube-proxy
API REQUEST
NODE
POD
RUNTIME
터널링
kubelet
NODE
POD
RUNTIME
kubelet

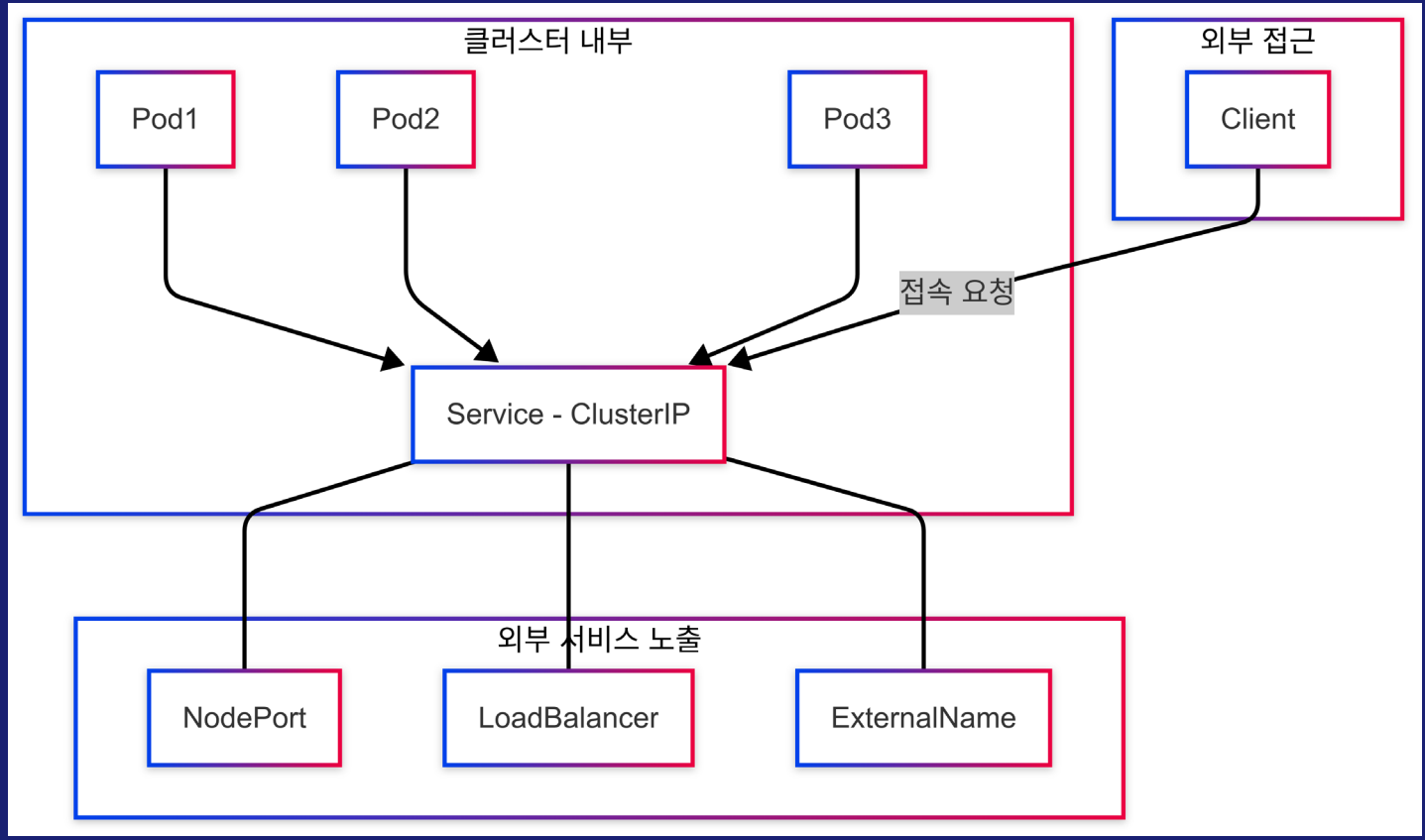
서비스 설명


LoadBalancer
[IP_ADDRESS]
NODE
PORT
POD
targetPort
container

서비스 설명



서비스 설명


클러스터 내부
Pod1
Pod2
Pod3
외부 접근
Client
접속 요청
Service - ClusterIP
외부 서비스 노출
NodePort
LoadBalancer
ExternalName

서비스 설명

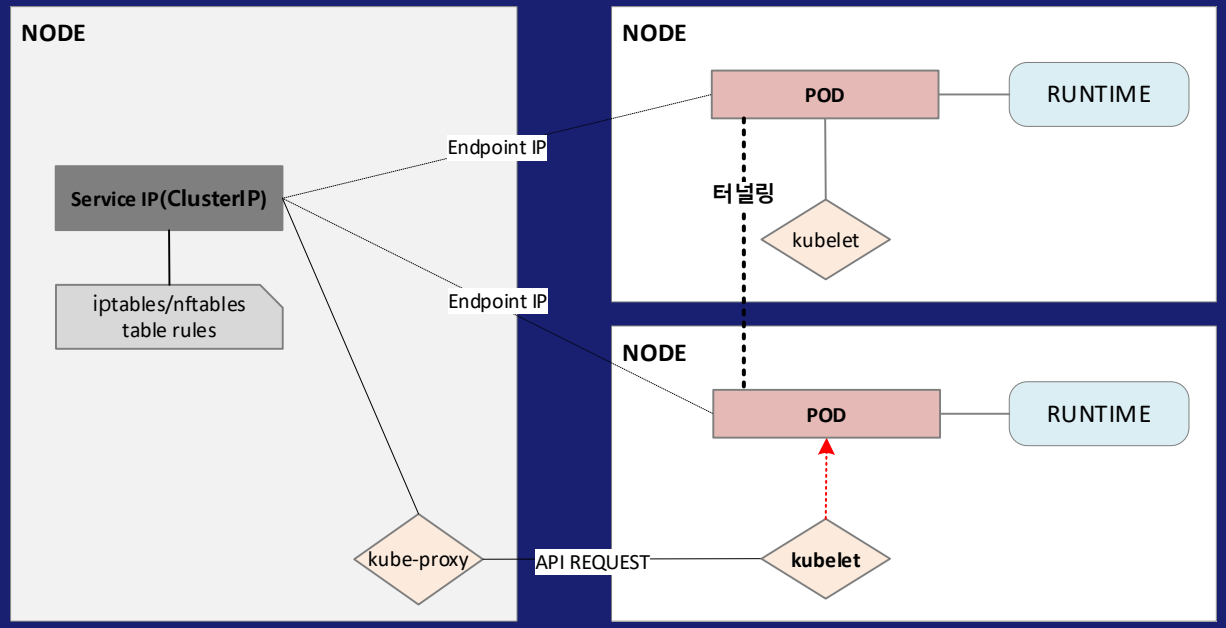
대비 4

ClusterIP

클러스터 아이피는 기본적으로 한 개 혹은 이상의 컨테이너가 외부로 노출이 되어야 하는 경우, SERVICE를 통해서 외부와 연결이 된다. 외부와 연동 및 연결이 될 때, 기본적으로 SOURCE, DESTINATION NAT으로 구성이 된다.

이러한 이유로, 별도의 옵션이 없이 외부에 서비스를 노출하는 경우 기본적으로 **클러스터 아이피**로 구성이 된다. 그림에서 "EndPoint IP"라고 적어 있는 부분은 POD가 사용하는 POD IP이다. 터널링은 POD가 클러스터로 묶어지면 그때 SVC를 통해서 클러스터가 구성이 된다.

이러한 이유로 ClusterIP라고 부른다.



ClusterIP*

클러스터 아이피는 별도의 옵션이 없이 명령어로 생성 및 구성이 가능하다.

```
# kubectl run --image=nginx --label=cluster-nginx --port=80 cluster-nginx
# kubectl expose cluster-nginx
# kubectl expose pod cluster-nginx --type=ClusterIP --port=80
# kubectl get svc cluster-nginx
# kubectl get svc, pod
NAME             TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
cluster-nginx    ClusterIP   10.100.200.150   <none>        80/TCP
```

클러스터 아이피로 구성이 된 경우에는, POD와 다르게 호스트에서 접근이 불가능하다. SVC에 접근하기 위해서 임시로 컨테이너로 생성 후 내부에서 확인이 가능하다. 확인하기 위해서 다음 슬라이드를 확인한다.

ClusterIP*

클러스터 아이피는 별도의 옵션이 없이 명령어로 생성 및 구성이 가능하다.

```
# kubectl run debug-pod --rm -it --image=busybox -- /bin/sh
curl -s http://10.100.200.150
<!DOCTYPE html><html><head><title>Welcome to nginx!</title>...</html>
```

NodePort*

노드 포트는 ClusterIP와 비슷하지만, ClusterIP에 외부에서 접근이 가능한 포트를 생성 및 구성 후, NAT를 통해서 서비스 아이피(SERVICE IP)에 접근이 가능하도록 한다. 위에서 사용한 Cluster IP를 NodePort로 변경해서 사용한다.

```
# kubectl run cluster-nginx --image=nginx --port=80
# kubectl expose pod cluster-nginx --type=NodePort --port=80
# kubectl get svc cluster-nginx
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
cluster-nginx   NodePort    10.100.200.150   <none>        80:31234/TCP
```

위와 같이 구성이 되면, 다음 명령어로 올바르게 구성이 되었는지 확인한다.

2025-04-18

NodePort*

디버깅 컨테이너를 실행하여, POD연결 후 접근 및 확인한다.

```
# kubectl run debug-pod --rm -it --image=busybox -- /bin/sh
wget -qO- http://<노드IP>:31234
<!DOCTYPE html><html><head><title>Welcome to nginx!</title>...</html>
```

위와 같이 출력이 되면 올바르게 실행이 되었다. 만약 외부에서 테스트를 원하는 경우, 디버깅 컨테이너를 실행하지 않고 접근이 가능하다.

```
# wget -qO- http://<노드IP>:31234
<!DOCTYPE html><html><head><title>Welcome to nginx!</title>...</html>
```

Load Balancer*

로드 밸런서는, 외부에서 아이피를 받아서 SVC를 통해서 POD접근이 가능하도록 한다. 다만, 로드 밸런서를 사용하기 위해서 두 가지 중 하나가 만족해야 한다.

1. 소프트웨어적으로 로드 밸런서 구성(클러스터 내부)
2. 하드웨어적으로 로드 밸런서 구성(클러스터 외부)

하드웨어는 물리적으로 구성하기가 어렵기 때문에 클러스터 내부에 로드 밸런서를 구성해서 아이피를 할당한다. 구성을 위해서 다음과 같이 사용한다.

진행 전, 현재 MetalLB는 설치가 되어 있지 않기 때문에, 올바르게 **`ExternalIP`**를 못 받아온다. 강사의 가이드를 통해서 해당 부분을 수정한다. 혹은 뒤에서 가이드에 따라서 설치한다.

Load Balancer*

이전 명령어를 활용하여 구성한다.

```
# kubectl run cluster-nginx --image=nginx --port=80
# kubectl expose pod cluster-nginx --type=LoadBalancer --port=80
# kubectl get svc cluster-nginx
NAME            TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)
cluster-nginx   LoadBalancer   10.100.200.150  <아이피 주소>     80:31234/TCP
# curl -s http://<아이피 주소>
<!DOCTYPE html><html><head><title>Welcome to nginx!</title>...</html>
```

ExternalName

ExternalName은 시험과 상관 없지만, 간단하게 학습한다. 이 기능은 컨테이너 내부에서 특정 도메인을 조회 시, 외부로 조회 하도록 한다. 예를 들어서 POD에서 "google.com"으로 트래픽을 전달하고 싶은 경우, 아래와 같이 구성한다. 간단하게 YAML파일을 생성한다.

```
# vi externalName.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: google-service
  namespace: default
spec:
  type: ExternalName
  externalName: google.com
```

ExternalName

YAML 파일로 생성이 완료가 되면, 등록 및 확인을 한다. 아래와 같이 작업을 수행하면 SVC도메인으로 호출이, google.com으로 전달 확인이 된다.

```
# kubectl apply -f google-externalname.yaml
# kubectl get svc google-service
NAME              TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)
google-service    ExternalName   <none>       google.com    <none>
# kubectl run debug-pod --rm -it --image=busybox -- /bin/sh
nslookup google-service.default.svc.cluster.local
```

랩

아래와 같은 조건으로 ClusterIP, NodePort, LoadBalancer를 생성 및 구성한다.

1. POD를 nginx기반으로 생성한다.
2. 버전은 사용 가능한 버전으로 사용한다.
3. NodePort는 38080으로 구성 및 생성한다.
4. 로드밸런서 아이피는 클러스터에서 할당하는 아이피를 사용한다.

랩 코드

nginx-pod.yaml는 아래와 같이 작성한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
```

랩 코드

nginx-clusterip.yaml는 아래와 같이 작성한다.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-clusterip
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
  - port: 80
    targetPort: 80
```

랩 코드

nginx-nodeport.yaml는 아래와 같이 작성한다.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-nodeport
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
  - port: 80
    targetPort: 80
    nodePort: 38080
```

랩 코드

nginx-loadbalancer.yaml는 아래와 같이 작성한다.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-loadbalancer
spec:
  type: LoadBalancer
  selector:
    app: nginx
  ports:
  - port: 80
    targetPort: 80
```

대비 5

레이블 및 셀렉터

Taints/Toleration

노드 셀렉터 및 선호도

2025-04-18

레이블 및 셀렉터*

대비 5

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

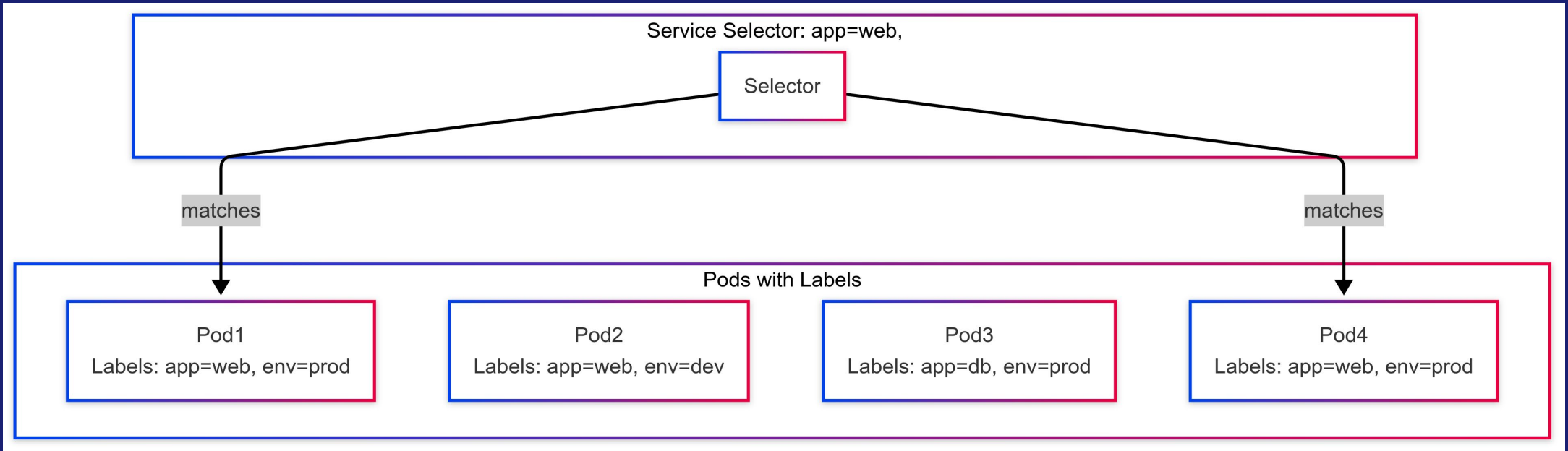
설명

레이블 기반으로 셀렉터는 생성하면, 이를 검색하기 위해서 `matchLabels, matchExpressions`를 사용해서 어떠한 자원을 찾을 지, 조건 검색이 가능하다. 보통 대다수 자원은 `matchLabels`를 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	`env notin (development)`
Exists	특정 키가 존재하면 선택	`zone exists`
DoesNotExist	특정 키가 없으면 선택	`region doesnotexist`

2025-04-18

설명


Service Selector: app=web,
Selector
matches
matches
Pods with Labels
Pod1
Labels: app=web, env=prod
Pod2
Labels: app=web, env=dev
Pod3
Labels: app=db, env=prod
Pod4
Labels: app=web, env=prod

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*

POD2번을 생성한다.

```
# 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 조건은로 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"]
# kubectl apply -f vi myapp-service.yaml
```

MatchLables

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"]
```

MatchExpressions

앞에 내용 계속 이어서…

```
    ports:
    - protocol: TCP
      port: 80
      targetPort: 80
# kubectl apply -f matchexpress-myapp-service.yaml
# kubectl get -f matchexpress-myapp-service.yaml
```

랩

다음과 같은 조건으로 POD를 생성한다.

1. POD는 nginx를 사용한다.
2. 레이블은 app=front, release=test, revision=1로 구성한다. POD의 이름은 nginx-test를 사용한다.
3. 레이블은 app=front, release=prod, revision=2로 구성한다. POD의 이름은 nginx-prod를 사용한다.
4. 서비스를 네임스페이스 lab-labels에 생성 및 구성한다.
5. 네임스페이스가 없으면 적절하게 생성한다.
6. POD구성이 완료가 되면 selector를 통해서 조회가 되는지 확인한다.

랩 코드

네임스페이스를 생성한다.

```
# kubectl create namespace lab-labels --dry-run=client -o yaml | kubectl apply -f -
```

랩 코드

nginx-test.yaml는 아래와 같이 작성한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-test
  namespace: lab-labels
  labels:
    app: front
    release: test
    revision: "1"
spec:
  containers:
  - name: nginx
    image: nginx
```

랩 코드

앞에 내용 계속 이어서…

```
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx-prod
  namespace: lab-labels
  labels:
    app: front
    release: prod
    revision: "2"
spec:
  containers:
  - name: nginx
    image: nginx
```

랩 코드

front-service.yaml는 아래와 같이 작성한다.

```
apiVersion: v1
kind: Service
metadata:
  name: front-service
  namespace: lab-labels
spec:
  selector:
    app: front
  ports:
  - port: 80
    targetPort: 80
```

Taints/Toleration

대비 5

설명

Taints, Toleration는 특정 컴퓨트 장비에서 장애가 발생 하였을 때, 영향을 받는 POD를 어떠한 방식으로 작업을 수행할지 결정한다.

개념	설명	자원	예제	우선순위
Taints	특정 노드에 POD생성 금지	NODE	kubectl taint nodes node1 key=value:Effect	낮음
Tolerations	특정 노드에 혹은 Taints가 선언된 노드에 POD생성	POD	파드 스펙에 tolerations 추가	높음

Taints*

Taint는 POD를 특정 노드에서 생성을 못하도록 한다. 쿠버네티스 클러스터에서는 controller노드에서는 taints가 설정 되어 있다.

아래와 같이 확인이 가능하다.

```
# kubectl describe node/cluster1-node1.example.com | grep -i taint
Taints:             node-role.kubernetes.io/control-plane:NoSchedule
```

Taints*

Taint는 POD를 특정 노드에서 생성을 못하도록 한다. 쿠버네티스 클러스터에서는 controller taints가 기본으로 설정 되어 있다. 아래와 같이 확인이 가능하다.

```
# kubectl describe node/cluster1-node1.example.com | grep -i taint
Taints:             node-role.kubernetes.io/control-plane:NoSchedule
```

혹은 반대로 **NoSchedule**를 걸기 위해서는 다음과 같이 작업이 가능하다.

```
# kubectl taint nodes node1 special=restricted:NoSchedule
```

위와 같이 작업하며, node1에 **NoSchedule**선언이 가능하다. 반대로 **NoSchedule**를 해제는 다음과 같다.

```
# kubectl taint nodes node1/node-role.kubernetes.io/control-
plane:NoSchedule-
```

Toleration *

이번에는 node3번을 taint적용 후, toleration-nginx POD 생성한다. 진짜로 잘 동작하는지!

```
# kubectl taint nodes node3 special=restricted:NoSchedule
```

POD를 아래와 같이 작성 후 생성한다.

```
# vi toleration.yaml
apiVersion: v1
kind: Pod
metadata:
  name: toleration-nginx
spec:
  nodeSelector:
    kubernetes.io/hostname: node3
```

Toleration*

위의 내용 계속 이어서 아래와 같이 작성한다.

```
  tolerations:
    - key: "special"
      operator: "Equal"
      value: "restricted"
      effect: "NoSchedule"
  containers:
    - name: nginx
      image: nginx
# kubectl apply -f toleration.yaml
# kubectl get -f toleration.yaml
```

Toleration *

다음 명령어로 확인한다.

```
# kubectl get pods -o wide
# kubectl get pods -o custom-columns=\
"POD NAME:.metadata.name,NODE:.spec.nodeName"
POD NAME              NODE
toleration-nginx      node3
other-pod             node1
```

랩

다음과 같은 조건으로 생성 및 구성한다.

1. node3에 대해서 taint를 설정한다. 다음과 같은 조건으로 taint를 선언한다.
    - **`maintenance, restricted:NoSchedule`**
2. node4에 대해서 taint를 설정한다. 다음과 같은 조건으로 taint를 선언한다.
    - **`maintenance, restricted:NoSchedule`**
3. 4번에 POD를 생성한다.
    - POD는 nginx를 사용한다.
    - nginx이름은 tlr-pod으로 생성한다.
    - 노드 4번에 구성된 taint는 수정이 안된다.

랩 코드

스케쥴링은 다음과 같이 조정이 가능하다.

```
# kubectl taint nodes node3 maintenance=restricted:NoSchedule
# kubectl taint nodes node4 maintenance=restricted:NoSchedule
# kubectl describe node node4 | grep Taints
```

랩 코드

`tlr-pod.yaml`는 아래와 같이 작성한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: tlr-pod
spec:
  containers:
  - name: nginx
    image: nginx
  nodeName: node4
```

랩 코드

위의 내용 계속 이어서…

```
  tolerations:
  - key: "maintenance"
    operator: "Equal"
    value: "restricted"
    effect: "NoSchedule"
```

랩 코드

확인.

```
# kubectl get pods -o wide
# kubectl describe pod tlr-pod
```

노드 셀렉터 및 선호도

대비 5

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 셀렉터(nodeSelector)는 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.

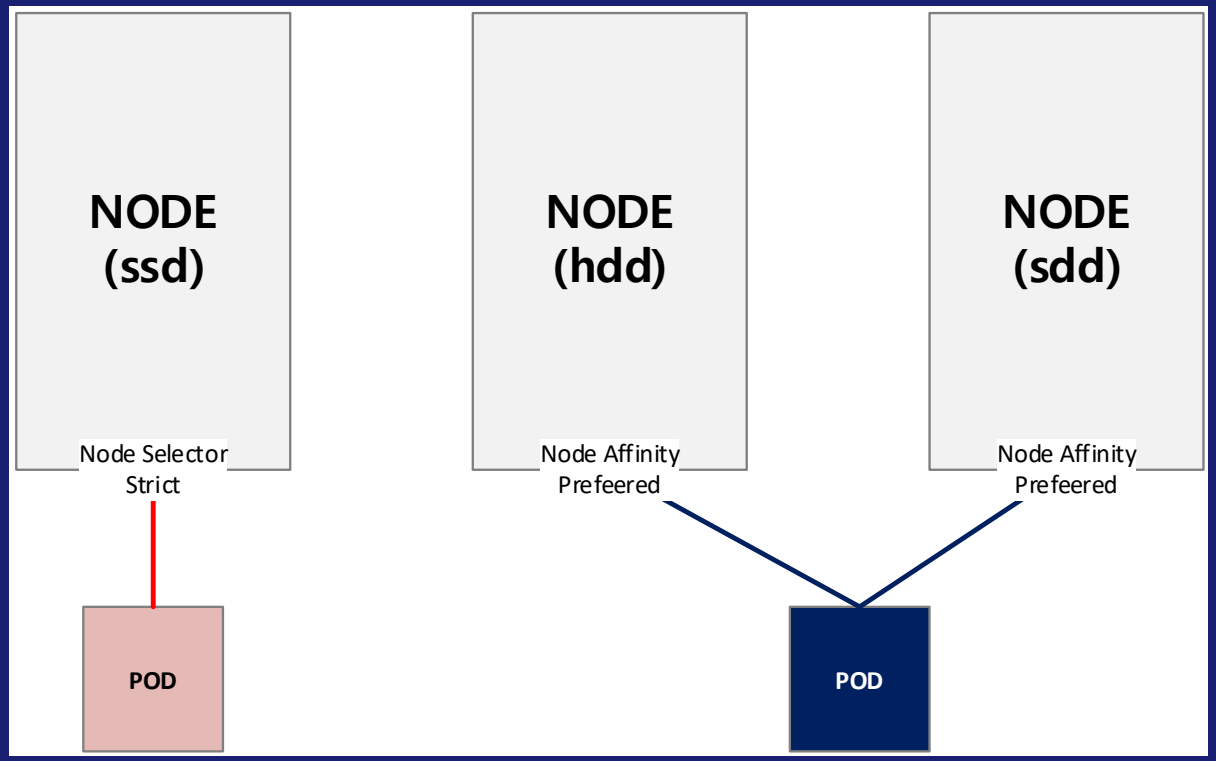
노드 셀렉터*

아래와 같이 작성한다.

```
# 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
```

랩

노드에 다음과 같은 조건으로 레이블을 구성한다.

1. node3번은 "no-more-service"값으로 레이블 할당한다.
    - 키 이름은 "status"으로 구성한다.
2. node4번은 "override-service"값으로 레이블 할당한다.
    - 키 이름은 "status"으로 구성한다.
3. 모든 값은 nodes에서 조회 및 확인한다.

랩 코드

아래와 같이 레이블 설정이 가능하다.

```
# kubectl label nodes node3 status=no-more-service
# kubectl label nodes node4 status=override-service
# kubectl get nodes --show-labels
NAME      STATUS   ROLES    AGE     VERSION   LABELSnode3    Ready    <none>
10d     v1.28.2    ... ,status=no-more-service, ... node4    Ready    <none>
10d     v1.28.2    ... ,status=override-service, ...
# kubectl get nodes -l status=no-more-service
# kubectl get nodes -l status=override-service
```

대비 6

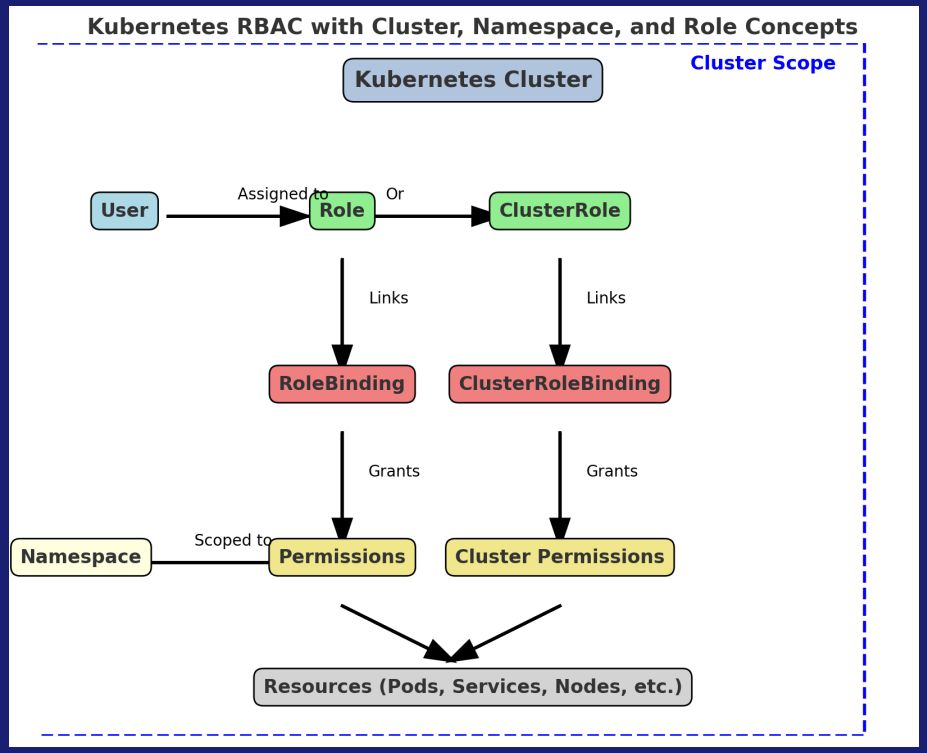
설명

사용자 생성 및 구성

RBAC 기반으로 자원 관리 및 접근제한*

설명

쿠버네티스 사용자 역할은 다른 시스템과 조금 독특하게 되어있다. 기본적으로 RBAC를 사용하기 때문에 이 부분은 익숙하다. 하지만, 나머지는 조금 생소할 수 있다. 다중 클러스터가 아닌, 싱글 클러스터 기준으로 다음과 같이 구성이 된다.



설명

위의 자원들 설명은 다음과 같다.

개념	설명	적용 범위
User	쿠버네티스 클러스터 내에서 인증된 사용자 또는 서비스 계정	특정 네임스페이스 또는 클러스터 전체
RoleBinding	특정 네임스페이스 내에서 User(또는 Group)에 Role을 연결	특정 네임스페이스
ClusterRoleBinding	클러스터 전체에서 User(또는 Group)에 ClusterRole을 연결	클러스터 전역
Namespace	쿠버네티스 리소스를 논리적으로 분리하는 공간, Role이 적용되는 단위	특정 네임스페이스
Cluster Scope	클러스터 전반에 영향을 미치는 개념 (ClusterRole, ClusterRoleBinding 등)	클러스터 전체

사용자 생성 및 구성

대비 6

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.
```

RBAC 기반으로 자원 관리 및 접근제한

대비 6

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
```

랩

사용자를 다음처럼 구성한다.

1. user1사용자를 생성한다.
2. user1사용자는 lab-user1 네임스페이스를 사용한다.
3. user1사용자는 lab-user1 네임스페이스에 모든 권한을 가지고 있다.
4. user2사용자는 lab-user1 네임스페이스에 접근이 가능하다.
    - 다만, POD만 접근이 가능하고, 나머지 자원에 대해서는 접근이 불가능하다.

랩 코드

네임스페이스를 생성한다.

```
# kubectl create namespace lab-user1
```

사용자가 user1, 2가 생성이 안되어 있으면, 앞 슬라이드 **사용자 생성**을 참고하여 user1,2생성한다.

랩 코드

아래와 같이 사용자 역할 파일 생성.

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

랩 코드

앞에 내용 계속 이어서…

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: user1-full-access-binding
  namespace: lab-user1
subjects:
- kind: User
  name: user1
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: user1-full-access
  apiGroup: rbac.authorization.k8s.io
```

랩 코드

아래와 같이 사용자 역할 파일 생성.

```
# vi role-user2.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: user2-pod-only
  namespace: lab-user1
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

랩 코드

앞에 내용 계속 이어서…

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: user2-pod-only-binding
  namespace: lab-user1
subjects:
- kind: User
  name: user2
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: user2-pod-only
  apiGroup: rbac.authorization.k8s.io
```

랩 코드

적용

```
# kubectl apply -f role-user1.yaml
# kubectl apply -f role-user2.yaml
# kubectl auth can-i delete configmap --as user1 -n lab-user1
yes
# kubectl auth can-i get pods --as user2 -n lab-user1
yes
# kubectl auth can-i get configmaps --as user2 -n lab-user1
no
```

대비 7

쿠버네티스 로깅 설명

모니터링 방법 및 기능 설명

애플리케이션 로그 확인

2025-04-18

쿠버네티스 로깅 설명

대비 7

2025-04-18

쿠버네티스 로깅 설명

쿠버네티스에서 로깅 및 모니터링을 매우 단순하게 지원한다. 기본적으로 클러스터 상태 모니터링은 별도로 없으며, 지원 범위 영역은 노드/포드 그리고, 각 자원에 대해서 정보 확인만 가능하다.

보통 사용하는 명령어는 다음과 같다.

1. `top`
2. `describe`
3. `logs`

모니터링 방법 및 기능 설명

대비 7

상태확인

앞에서 많이 사용하였던 자원 목록은 다음 명령어로 확인이 가능하다.

```
# kubectl get all -n <namespace>
# kubectl get all -n user1-namespace
NAME                                READY   STATUS    RESTARTS   AGE
pod/nginx-6799fc88d8-xxxxx          1/1     Running   0          2m
NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/nginx        ClusterIP   10.96.12.34    <none>        80/TCP    2m
NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx          1/1     1            1           2m
NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-6799fc88d8          1         1         1       2m
```

상태확인

특정 POD를 상태를 확인하기 위해서 다음과 같이 실행한다.

```
# kubectl describe pod <pod-name> -n <namespace>
# kubectl describe pod nginx-6799fc88d8-xxxxx -n user1-namespace
Name:         nginx-6799fc88d8-xxxxx
Namespace:    user1-namespace
Node:         node-1/192.168.1.10
Labels:       app=nginx
Status:       Running
IP:           10.244.1.2
Containers:
  nginx:
    Container ID:   docker://abcdef123456
    Image:          nginx
    State:          Running
    Ready:          True
    Restart Count:  0
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  2m    default-scheduler  Successfully assigned user1-namespace/nginx-xxxxx to node-1
  Normal  Pulled     2m    kubelet            Successfully pulled image "nginx"
```

상태확인

특정 POD를 상태를 확인하기 위해서 다음과 같이 실행한다.

```
# kubectl logs <pod-name> -n <namespace>
# kubectl logs nginx-6799fc88d8-xxxxx -n user1-namespace
127.0.0.1 - - [28/Feb/2025:10:00:00 +0000] "GET / HTTP/1.1" 200 612 "-"
"curl/7.68.0"
127.0.0.1 - - [28/Feb/2025:10:00:01 +0000] "GET / HTTP/1.1" 200 612 "-"
"curl/7.68.0"
```

노드 상태확인

노드 상태를 확인하기 위해서 다음과 같이 명령어 수행이 가능하다.

```
# kubectl logs <pod-name> -n <namespace>
# kubectl logs nginx-6799fc88d8-xxxxx -n user1-namespace
127.0.0.1 - - [28/Feb/2025:10:00:00 +0000] "GET / HTTP/1.1" 200 612 "-"
"curl/7.68.0"
127.0.0.1 - - [28/Feb/2025:10:00:01 +0000] "GET / HTTP/1.1" 200 612 "-"
"curl/7.68.0"
```

노드 상태확인

노드 상태를 확인하기 위해서 다음과 같이 명령어 수행이 가능하다.

```
# kubectl describe node <node-name>
# kubectl describe node node-1
Name:                node-1
Roles:               worker
InternalIP:          192.168.1.10
Capacity:
  cpu:               4
  memory:            8Gi
Conditions:
  Type               Status
  ----               ------
  Ready              True
```

이벤트 상태확인

특정 자원 및 네임스페이스에서 발생한 자원 이벤트가 있는지 확인하는 명령어.

```
# kubectl get events -n <namespace>
# kubectl get events -n user1-namespace
LAST SEEN   TYPE      REASON      OBJECT              MESSAGE
10s         Normal    Scheduled   pod/nginx-xxxxx   Successfully assigned
to node-1
5s          Warning   Failed      pod/nginx-xxxxx   ImagePullBackOff:
Failed to pull image "nginx"
```

이벤트 상태확인

위의 기능을 정리하면 다음과 같이 된다.

명령어	추가 옵션	설명	사용 예시
kubectl logs	-f	로그를 실시간으로 계속 출력 (스트리밍)	kubectl logs -f nginx-6799fc88d8-xxxxx -n user1-namespace
	--tail=100	마지막 100줄만 출력	kubectl logs --tail=100 nginx-6799fc88d8-xxxxx -n user1-namespace
	-c <container-name>	멀티 컨테이너 Pod에서 특정 컨테이너 로그 확인	kubectl logs -c nginx nginx-6799fc88d8-xxxxx -n user1-namespace
kubectl describe	pod	Pod의 상세 상태 확인	kubectl describe pod nginx-6799fc88d8-xxxxx -n user1-namespace
	node	특정 노드의 상태 확인	kubectl describe node node-1
	deployment	Deployment의 상태 확인	kubectl describe deployment nginx -n user1-namespace
kubectl get events	--sort-by=.metadata.creationTimestamp	이벤트를 시간 순으로 정렬	kubectl get events --sort-by=.metadata.creationTimestamp -n user1-namespace

2025-04-18

이벤트 상태확인

위의 기능을 정리하면 다음과 같이 된다.

시나리오	해결 방법
Pod가 정상적으로 실행되지 않음	kubectl describe pod <pod-name> -n <namespace>를 실행하여 상태, 이벤트 확인
애플리케이션 오류 발생	kubectl logs <pod-name> -n <namespace>로 애플리케이션 로그 확인
Pod가 CrashLoopBackOff 상태임	kubectl describe pod <pod-name> -n <namespace>에서 최근 이벤트 및 실패 원인 확인 후, kubectl logs로 추가 디버깅
Pod가 Pending 상태에서 멈춤	kubectl get events -n <namespace>로 스케줄링 문제 확인
Pod가 특정 노드에서만 오류 발생	kubectl describe node <node-name>로 해당 노드의 상태 및 리소스 확인

이벤트 상태확인

위의 기능을 정리하면 다음과 같이 된다.

명령어	목적	출력 정보	사용 예시
kubectl logs	컨테이너 내부에서 실행된 로그 확인	실행 로그, 에러 로그, 요청 로그	`kubectl logs nginx-xxxxx -n user1-namespace`
kubectl exec	컨테이너 내부에서 직접 명령어 실행	내부 상태 확인 및 디버깅	`kubectl exec -it nginx-xxxxx -n user1-namespace -- /bin/sh`

이벤트 상태확인

위의 기능을 정리하면 다음과 같이 된다.

명령어	목적	사용 예시
`journalctl -u kubelet -f`	kubelet 서비스 실시간 모니터링	`journalctl -u kubelet -f`
`journalctl --since "1 hour ago"`	지난 1시간 동안의 로그 확인	`journalctl --since "1 hour ago"`
`journalctl -u kubelet`		`kubelet` 서비스에서 발생한 에러 로그 필터링
`journalctl CONTAINER_NAME=<container-name>`	특정 컨테이너의 로그 확인	`journalctl CONTAINER_NAME=nginx`

랩

POD를 Nginx이미지 기반으로 생성한다.

1. nginx-log라는 이름으로 POD를 생성한다.
2. 해당 POD는 lab-log에 생성이 된다.
3. nginx-log에서 생성된 최신 로그 5개를 nginx-log.txt라는 파일에 저장한다.

랩 코드

적용.

```
# kubectl create namespace lab-log
# vi nginx-log.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-log
  namespace: lab-log
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

랩 코드

확인.

```
# kubectl apply -f nginx-log.yaml
# kubectl logs nginx-log -n lab-log | tail -n 5 > nginx-log.txt
# cat nginx-log.txt
```

대비 8

애플리케이션 생명주기

설명

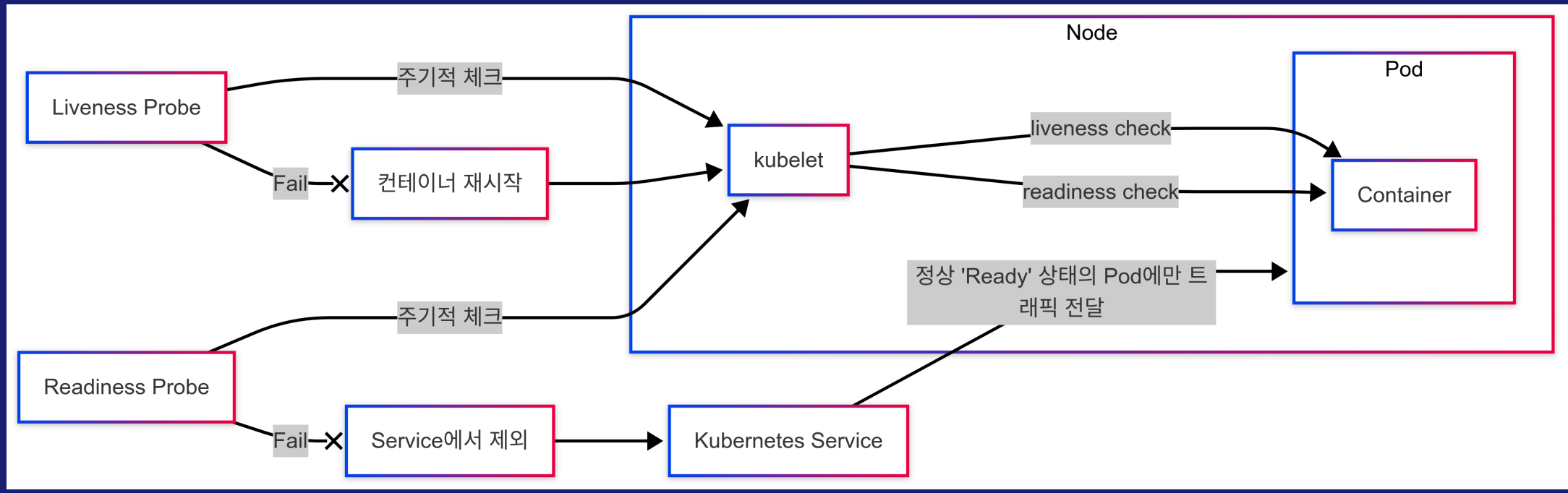
관리 방법	설명	사용 예시
Liveness Probe	애플리케이션이 정상 작동 중인지 감지하여, 비정상 상태일 경우 자동 재시작	`kubectl apply -f liveness.yaml`
Readiness Probe	애플리케이션이 트래픽을 받을 준비가 되었는지 감지하여, 준비되지 않으면 서비스에서 제외	`kubectl apply -f readiness.yaml`
Startup Probe	애플리케이션이 완전히 시작되었는지 확인하여, 일정 시간 동안 기다려 준 후 Liveness Probe 적용	`kubectl apply -f startup.yaml`

설명

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

2025-04-18

설명


Node
Pod
Liveness Probe
주기적 체크
Fail
컨테이너 재시작
kubelet
liveness check
readiness check
Container
정상 'Ready' 상태의 Pod에만 트래픽 전달
주기적 체크
Readiness Probe
Fail
Service에서 제외
Kubernetes Service

서비스 상태 모니터

대비 8

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
```

랩

당신은 웹 애플리케이션(NGINX)을 실행하는 Pod를 관리하고 있다. 최근에 애플리케이션이 간헐적으로 멈추거나 응답하지 않는 문제가 발생.

이 문제를 해결하기 위해 다음 조건을 만족하는 liveness와 readiness 프로브를 구성 및 모니터링한다.

1. Liveness Probe:
   - 애플리케이션이 30초 이상 응답하지 않으면 컨테이너를 재시작하도록 구성합니다.
   - /healthz 엔드포인트를 사용하여 상태를 확인합니다.
2. Readiness Probe:
   - 애플리케이션이 준비되지 않은 경우(예: 초기 로딩 중)에는 서비스 트래픽이 전달되지 않도록 구성합니다.
   - /ready 엔드포인트를 사용하며, 5초 간격으로 체크하고 최대 3번 실패 시 준비 상태에서 제외됩니다.

랩 코드

파일 이름은 `nginx-probe.yaml`으로 작성.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-probe
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

랩 코드

앞에 내용 계속 이어서…

```
    livenessProbe:
      httpGet:
        path: /healthz
        port: 80
      initialDelaySeconds: 10
      periodSeconds: 10
      timeoutSeconds: 1
      failureThreshold: 3  # 3회 연속 실패 시 재시작
```

랩 코드

앞에 내용 계속 이어서…

```
    readinessProbe:
      httpGet:
        path: /ready
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
      timeoutSeconds: 1
      failureThreshold: 3  # 3회 연속 실패 시 Ready 상태 제외
```

랩 코드

앞에 내용 계속 이어서…

```
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 10
      periodSeconds: 10
      timeoutSeconds: 1
      failureThreshold: 3  # 3회 연속 실패 시 재시작
```

랩 코드

앞에 내용 계속 이어서…

```
    readinessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
      timeoutSeconds: 1
      failureThreshold: 3  # 3회 연속 실패 시 Ready 상태 제외
```

랩 코드

확인.

```
# kubectl describe pod nginx-probe
# kubectl get pod nginx-probe -o jsonpath='{.status.conditions}'
```

배포전략

대비 8

배포 전략

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

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

helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-server/

METRICS

HPA/VPA를 사용하기 위해서 metrics-server가 설치가 되어야 한다. Helm기반으로 아래와 같이 설치를 수행한다.

```
# helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-server/
# helm install metrics-server metrics-server/metrics-server --version 3.12.2
# vi values.yaml
replicaCount: 1
args:
  - --kubelet-insecure-tls
  - --kubelet-preferred-address-types=InternalIP,Hostname,ExternalIP
# helm install metrics-server metrics-server/metrics-server -f values.yaml
```

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(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: nginx
  minReplicas: 2
```

배포 전략(HPA)*

CPU사용율이 50%가 넘어가면 POD 확장을 시작한다.

```
  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
```

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
```

랩

당신은 CPU 부하에 따라 자동으로 스케일 아웃이 되는 웹 서비스를 관리 중이다. 다음 요구사항을 만족하는 HPA를 생성해야 한다.

1. 디플로이먼트 이름은 web-app이다.
2. web-app은 최소 2개, 최대 6개의 레플리카를 가진다.
3. CPU 사용률이 평균 60%를 초과하면 새로운 Pod를 생성하도록 구성한다.
4. 네임스페이스는 lab-web-app를 사용한다.

랩 코드

실행.

```
# kubectl autoscale deployment web-app --cpu-percent=60 --min=2 --max=6
```

롤링 업데이트

대비 8

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	`kubectl apply -f nginx-deployment.yaml`	nginx 1.19 버전 배포
2. 현재 상태 확인	`kubectl get deployments`	배포된 Pod 확인
3. 업데이트 적용	`kubectl set image deployment/nginx-deployment nginx=nginx:1.21`	nginx 버전 업데이트
4. 업데이트 진행 확인	`kubectl rollout status deployment/nginx-deployment`	롤링 업데이트 진행 상황 확인
5. 롤백 (되돌리기)	`kubectl rollout undo deployment/nginx-deployment`	이전 버전으로 복구

2025-04-18

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   AGEnginx-
deployment-6799fc88d8-xxxxx   1/1     Running   0          5snginx-
deployment-6799fc88d8-yyyyy   1/1     Running   0          10snginx-
deployment-6799fc88d8-zzzzz   1/1     Running   0          30s
```

2025-04-18

랩

당신은 웹 애플리케이션(예: NGINX)을 Kubernetes 클러스터에서 운영. 이 애플리케이션은 CPU 부하에 따라 부하가 급증이 된다. 스케일 아웃하여 처리할 수 있도록 **`Horizontal Pod Autoscaler(HPA)`**를 구성한다.

1. 웹 애플리케이션을 실행하는 Deployment를 구성
   - 최소 2개의 파드를 실행하고, 최대 10개까지 스케일 아웃
   - 컨테이너에 적절한 CPU 요청(requests)과 제한(limits)을 설정하여 CPU 사용률 기반 스케일링이 가능
2. HPA는 다음과 같이 동작하도록 설정
   - 대상 Deployment의 평균 CPU 사용률이 50%를 초과하면 스케일 아웃
   - 최소 2개, 최대 10개의 복제본(replica)으로 스케일링

- 네임스페이스는 **`lab-web-app-deployment`**으로 작성, 애플리케이션도 동일한 이름을 사용한다.

2025-04-18

랩 코드

아래와 같이 코드 작성한다.

```
# vi web-app-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web-app
```

랩 코드

앞에 내용 계속 이어서…

```
template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          requests:
            cpu: "100m"
          limits:
            cpu: "200m"
        ports:
        - container
```

랩 코드

앞에 내용 계속 이어서…

```
# kubectl apply -f web-app-deployment.yaml
# kubectl autoscale deployment web-app \
  --cpu-percent=50 \
  --min=2 \
  --max=10
# kubectl get hpa
NAME      REFERENCE            TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
web-app   Deployment/web-app   45% / 50%       2         10        2          1m
# kubectl run cpu-stress --image=busybox --restart=Never -it -- /bin/sh
→ while true; do :; done  # CPU를 소비하는 테스트
```

2025-04-18

블루/그린, 카나리 업데이트

대비 8

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 역할 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
```

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-stable버전을 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

앞에 내용 계속 이어서…

```
  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-canary버전을 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

애플리케이션 인자 값 설정 및 Containerfile

대비 8

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, 사용한다.

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*

대비 8

2025-04-18

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*

대비 8

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 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 적용

앞에서 빌드한 컨테이너 이미지 기반으로 Nginx서비스를 구성 및 실행한다.

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

DEPLOYMENT 적용

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

```
        env:
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: my-config
              key: DB_HOST
        - 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
```

정리

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		`kubectl get configmap my-config -o yaml`
			`kubectl get secret my-secret -o yaml`
Secret 복호화	base64 인코딩된 데이터 확인		`kubectl get secret my-secret -o jsonpath="{.data.DB_PASSWORD}"`

정리

항목	ConfigMap	Secret
목적	환경 변수, 설정 값 저장	비밀번호, API 키, 인증서 등 민감한 정보 저장
데이터 저장 방식	평문(Plain Text)	Base64 인코딩 (암호화 아님)
사용 대상	애플리케이션 설정값 (예: DB_HOST, APP_MODE)	비밀번호, 토큰, 인증서 (DB_PASSWORD, API_KEY)
볼륨 마운트 가능 여부	가능 (configMap 볼륨 사용)	가능 (secret 볼륨 사용)
환경 변수로 사용 가능 여부	가능 (envFrom 또는 valueFrom)	가능 (envFrom 또는 valueFrom)

정리

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	`kubectl create configmap my-config --from-literal=DB_HOST=mydatabase.local`	`kubectl create secret generic my-secret --from-literal=DB_PASSWORD=SuperSecret123`

랩

하나의 애플리케이션(예: 웹 애플리케이션)을 Kubernetes 클러스터에 배포할 때, 다음 요구사항을 만족하는 리소스를 생성.

1. ConfigMap:
   - 애플리케이션의 설정 정보를 저장
   - 예를 들어, APP_MODE와 APP_PORT 값을 포함
2. Secret:
   - 애플리케이션에서 사용할 민감한 정보를 저장
   - 예: 데이터베이스 비밀번호(DB_PASSWORD).
3. Deployment 구성 및 Label 적용:
   - Deployment 리소스에 애플리케이션 컨테이너를 정의하고, ConfigMap과 Secret에서 값을 가져와 환경 변수
   - Deployment와 Pod에 적절한 Label(예: app: my-app, tier: backend)을 적용하여 리소스를 구분
4. 네임스페이스 이름은 lab-configmap으로 생성한다.
   - 애플리케이션 이름 적절하게 설정한다.
   - configmap, secret도 적절 이름을 설정한다.

랩 코드

앞에 내용 계속 이어서…

```
# vi configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_MODE: "production"
  APP_PORT: "8080"
```

랩 코드

앞에 내용 계속 이어서…

```
# vi secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
type: Opaque
stringData:
  DB_PASSWORD: "s3cr3tpassw0rd"
```

랩 코드

파일 이름은 `my-app.yaml`으로 작성한다.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: my-app
    tier: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
```

랩 코드

앞에 내용 계속 이어서…

```
  template:
    metadata:
      labels:
        app: my-app
        tier: backend
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

랩 코드

앞에 내용 계속 이어서…

```
      env:
      - name: APP_MODE
        valueFrom:
          configMapKeyRef:
            name: app-config
            key: APP_MODE
      - name: APP_PORT
        valueFrom:
          configMapKeyRef:
            name: app-config
            key: APP_PORT
```

랩 코드

앞에 내용 계속 이어서…

```
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: app-secret
              key: DB_PASSWORD
```

랩 코드

앞에 내용 계속 이어서…

```
# kubectl apply -f app-config.yaml
# kubectl apply -f app-secret.yaml
# kubectl apply -f my-app-deployment.yaml
# kubectl get deployment my-app -o wide
# kubectl get pods -l app=my-app
# kubectl describe pod <pod-name> | grep APP_MODE
# kubectl exec -it <pod-name> -- printenv | grep APP
```

대비 9

운영체제 및 클러스터 관리

운영체제 및 클러스터 관리*

운영체제 업데이트

클러스터 업데이트

ETCD 백업/복원

설명

클러스터 업데이트 전, 반드시 O/S업데이트 및 최신 쿠버네티스 패키지 설치가 필요하다. 각 배포판에 적절하게 저장소를 구성 후 다음과 같은 순서로 작업을 수행한다.

1. 기존 버전에서 다음 버전으로 업데이트 가능 여부(보통 순차적으로 진행)
2. O/S의 패키지 업데이트 및 쿠버네티스 저장소 변경(1.30에서 1.31로)
3. 새로운 부트스트랩 및 CTL, KUBELET설치(1.30에서 1.31로)

이와 같은 순서로 작업을 진행한다. 쿠버네티스에서는 버전 업데이트 혹은 업그레이드는 가급적이면 순차적 버전으로 진행을 권장한다.

설명

버전은 시멘틱(https://kubernetes.io/ko/releases/version-skew-policy/)형태로 지원 및 구성이 되어 있다.

- x: 메이저 버전
- y: 마이너 버전
- z: 패치 버전

릴리즈 주기는 상황에 따라서 다르지만, 1년에 3번 정도 릴리즈 된다. 평균적으로 4개월에 한번씩 새로운 버전이 마이너 릴리즈가 된다. 현재 공식적으로 지원하는 버전은 1.29/30/31 버전이며, 이전 버전은 현재는 더 이상 패치가 지원하지 않는다.

kubelet, kube-apiserver는 최대 2개의 마이너 버전까지만 호환이 되며, 이 이상의 차이에 대해서는 호환성이 유지가 안된다.

앞서 이야기 하였지만, 버전을 건너뛰어서 업그레이드 및 업데이트는 지원하지 않는다.

2025-04-18

ETCD백업

ETCD를 백업하기 위해서 다음과 같이 조회가 가능하다.

```
# kubectl get pods -n kube-system | grep etcd
etcd-control-plane   1/1     Running   0          100d
```

백업을 하기 위해서 etcdctl명령어를 설치 후, 다음과 같이 실행한다.

```
# dnf install etcd -y
# apt install -y etcd-client
# ETCDCTL_API=3 etcdctl snapshot save /backup/etcd-snapshot.db \  --cacert
/etc/kubernetes/pki/etcd/ca.crt \  --cert
/etc/kubernetes/pki/etcd/server.crt \  --key
/etc/kubernetes/pki/etcd/server.key
Snapshot saved at /backup/etcd-snapshot.db
```

ETCD검증

올바르게 백업이 되었는지 확인한다.

```
# ETCDCTL_API=3 etcdctl snapshot status /backup/etcd-snapshot.db
Snapshot Path: /backup/etcd-snapshot.dbDB Size: 5.2 MBCluster ID:
cdf818194e3a8c32
```

ETCD복구

문제가 발생하면, 다음과 같은 명령어 복원 및 복구가 가능하다. 올바르게 ETCD 내용을 갱신하기 위해서 kubelet 서비스를 재시작 한다.

```
# ETCDCTL_API=3 etcdctl snapshot restore /backup/etcd-snapshot.db \  --
data-dir /var/lib/etcd
# systemctl restart kubelet
```

운영체제 업데이트

클러스터 업데이트 전, 보통 O/S부분을 업데이트, 그리고 최신 버전의 kubectl, kubeadm 그리고, kubelet서비스를 설치한다.

```
# kubectl get nodes
# kubectl get pods -A
# kubectl drain <노드명> --ignore-daemonsets --delete-emptydir-data
```

업데이트가 될 노드는 작업 수행 전, 다른 노드로 컨테이너를 재구성 해두어야 한다. 그렇지 않는 경우, 사용자 서비스가 올바르게 접근이 안된다.

운영체제 업데이트(레드햇)

컨테이너는 항상 커널이 중요하다. 대다수 기능은 커널을 통해서 구성 및 제공이 된다. 모든 패키지를 업데이트 혹은 커널만 업데이트 하여도 상관이 없다.

```
# dnf update -y
# dnf update kernel -y
# reboot
```

운영체제 업데이트(레드햇)

쿠버네티스 패키지 업데이트가 필요하다. 1.30에서 1.31로 버전을 변경한다.

```
# vi /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/v1.31/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/v1.31/rpm/repodata/repomd.xml.key
exclude=kubelet kubeadm kubectl cri-tools kubernetes-cni
```

운영체제 업데이트(레드햇)

목록에 1.31버전이 출력 되는지 확인한다.

```
# dnf list --showduplicates kubelet | tail -10
# dnf install -y kubelet-1.31.0 kubeadm-1.31.0 kubectl-1.31.0
# systemctl daemon-reexec
# systemctl restart kubelet
```

운영체제 업데이트(데비안)

쿠버네티스 패키지 업데이트가 필요하다. 1.30에서 1.31로 버전을 변경한다.

```
# curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.31/deb/Release.key | sudo
gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
# apt update
# apt list -a kubelet | tail -10
# apt install -y kubelet=1.31.0-00 kubeadm=1.31.0-00 kubectl=1.31.0-00
# systemctl daemon-reexec
# systemctl restart kubelet
```

운영체제 업데이트(데비안)

데비안 계열에서 패키지 업데이트는 다음과 같이 실행한다.

```
# apt update && apt upgrade -y
# apt install --only-upgrade linux-image-generic -y
reboot
```

클러스터 업그레이드(컨트롤러)

이제, 실제로 클러스터 업데이트를 수행한다. 앞에서는 APT, DNF를 통해서 패키지 관리를 확인 하였다. 이 작업을 수행하기 위해서 앞에서 이야기 하였던 작업이 완료가 되어야 한다.

1. ETCD 백업
2. DEB/RPM 패키지 업데이트 및 쿠버네티스 패키지 신 버전 설치

현재 설치 되어 있는 클러스터의 버전 작업 실행 전, 확인한다.

```
# kubectl version --short
Client Version: v1.30.0
Server Version: v1.30.0
```

클러스터 업그레이드(컨트롤러)

컨트롤 노드를 업데이트 한다.

```
# dnf install -y kubeadm-1.31.0
# apt install -y kubeadm=1.31.0-00
```

업그레이드를 수행한다. 모든 작업은 컨트롤러 노드가 우선 수행이 되어야 한다. 이때 멀티 노드를 사용하는 경우 각 노드에 cordon으로 서비스가 되지 않도록 설정해야 한다.

```
# kubectl drain cluster1-node1.example.com --ignore-daemonsets --delete-emptydir-data
# kubeadm upgrade plankubeadm upgrade apply v1.31.0
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.31.0"
```

클러스터 업그레이드(컨트롤러)

만약, 한 대 이상의 컨트롤러 노드가 있으면, 해당 노드에 접근하여 앞서 사용한 명령어를 적용한다.

```
nodeX# kubeadm upgrade plankubeadm upgrade apply v1.31.0
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.31.0"
```

kubectl, kubelet 두 개가, 신 버전으로 설치가 안되어 있으면 설치한다.

```
# dnf install -y kubelet-1.31.0 kubectl-1.31.0
# apt install -y kubelet=1.31.0-00 kubectl=1.31.0-00
# systemctl daemon-reexec
# systemctl restart kubelet
```

올바르게 동작하도록 모든 컨트롤러 노드에 uncordon를 수행한다.

```
# kubectl uncordon <control-plane-node>
```

클러스터 업그레이드(컨트롤러)

위의 내용을 정리하면 다음과 같다.

```
1 kubectl drain <control-plane-node>
2 dnf/apt install kubeadm-1.31.0
3 kubeadm upgrade apply v1.31.0
4 dnf/apt install kubelet-1.31.0 kubectl-1.31.0
5 systemctl restart kubelet
6 kubectl uncordon <control-plane-node>
```

클러스터 업그레이드(컴퓨트)

컴퓨트 노드 업그레이드는 다음과 같다.

```
# kubectl drain <worker-node> --ignore-daemonsets --delete-emptydir-data
# dnf install -y kubeadm-1.31.0
# apt install -y kubeadm=1.31.0-00
# kubeadm upgrade node
Redhat kinds
# dnf install -y kubelet-1.31.0 kubectl-1.31.0
Debian kinds
# apt install -y kubelet=1.31.0-00 kubectl=1.31.0-00
# systemctl daemon-reexec
# systemctl restart kubelet
# kubectl uncordon <worker-node>
```

클러스터 업그레이드(컴퓨트)

업그레이드 후, 버전이 올바르게 적용 되었는지 확인한다.

```
# kubectl get nodes
NAME              STATUS   ROLES           AGE    VERSION
control-plane-1   Ready    control-plane   100d   v1.31.0
control-plane-2   Ready    control-plane   100d   v1.31.0
control-plane-3   Ready    control-plane   100d   v1.31.0
worker-1          Ready    worker          100d   v1.31.0
worker-2          Ready    worker          100d   v1.31.0
worker-3          Ready    worker          100d   v1.31.0
# kubectl get pods -A
# kubectl cluster-info
```

정리

앞에서 사용한 기능을 정리하면 다음과 같다.

단계	명령어	설명
Control Plane 노드 드레이닝	`kubectl drain <control-plane-node>`	해당 노드에서 Pod 이동
kubeadm 업그레이드	`kubeadm upgrade apply v1.31.0`	Control Plane 설정 업데이트
kubelet & kubectl 업그레이드	`dnf/apt install kubelet-1.31.0 kubectl-1.31.0`	kubelet & kubectl 업데이트

정리

앞에서 사용한 기능을 정리하면 다음과 같다.

단계	명령어	설명
kubelet 재시작	`systemctl restart kubelet`	변경 사항 적용
Control Plane 노드 활성화	`kubectl uncordon <control-plane-node>`	다시 스케줄링 허용
Worker 노드 업그레이드	`kubeadm upgrade node`	Worker 노드 업데이트
Worker 노드 활성화	`kubectl uncordon <worker-node>`	Worker 노드 복귀
최종 점검	`kubectl get nodes`	모든 노드가 Ready 상태인지 확인

랩

Debian 기반의 Kubernetes 클러스터(현재 버전 1.29)를 운영. 클러스터를 안정적으로 운영하기 위해 Kubernetes 1.30으로 업그레이드. 아래 단계별 작업을 수행하고, 각 단계에서 예상되는 결과를 확인.

버전이 다른 경우, 현 버전보다 한 단계 높은 버전으로 구성

1. 현재 클러스터의 제어 평면과 워커 노드의 Kubernetes 버전을 확인하고, 모든 중요한 데이터와 설정을 백업
2. 업그레이드 전, Kubernetes 공식 업그레이드 문서와 릴리즈 노트를 참고하여 업그레이드 전 주의사항을 숙지

랩 코드

데비안 기반 업그레이드. 레드햇 계열은 제외.

```
# kubectl get nodes -o wide
# kubectl version --short
Client Version: v1.29.X
Server Version: v1.29.X

# ETCDCTL_API=3 etcdctl snapshot save /backup/etcd-backup.db \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key
# cp -r /etc/kubernetes/ /backup/kubernetes/
# cp -r ~/.kube/config /backup/
```

랩 코드

앞에 내용 계속 이어서…

```
# apt update && sudo apt install -y apt-transport-https ca-certificates
curl
# apt-mark unhold kubeadm && apt-get install -y kubeadm=1.30.X-00 && \
apt-mark hold kubeadm
# kubeadm upgrade plan
# kubeadm upgrade apply v1.30.X
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.30.X".
```

랩 코드

앞에 내용 계속 이어서…

```
# apt-mark unhold kubelet kubectl
# apt-get install -y kubelet=1.30.X-00 kubectl=1.30.X-00
# apt-mark hold kubelet kubectl
# systemctl daemon-reexec
# systemctl restart kubelet

# 1. 드레인
kubectl drain <node-name> --ignore-daemonsets --delete-local-data
```

랩 코드

앞에 내용 계속 이어서…

```
# 2. kubeadm 업그레이드
sudo apt-mark unhold kubeadm && \
sudo apt-get install -y kubeadm=1.30.X-00 && \
sudo apt-mark hold kubeadm
sudo kubeadm upgrade node

# 3. kubelet/kubectl 업그레이드
apt-mark unhold kubelet kubectl && \
apt-get install -y kubelet=1.30.X-00 kubectl=1.30.X-00 && \
apt-mark hold kubelet kubectl
systemctl daemon-reexec
systemctl restart kubelet
```

랩 코드

앞에 내용 계속 이어서…

```
# 4. 언드레인
kubectl uncordon <node-name>
```

대비 10

멀티 컨테이너

멀티 컨테이너

대비 10

멀티 컨테이너

POD에 한 개 이상의 컨테이너 실행을 멀티 컨테이너라고 한다. 멀티 컨테이너는 다음과 같은 기능을 제공한다.

1. 서로 다른 역할을 하는 컨테이너를 하나의 Pod에서 실행 가능
2. 사이드카(Sidecar) 패턴을 활용하여 기능 확장 가능
3. 컨테이너 간 통신이 간편함 (localhost 사용 가능)

여기서 다루는 멀티 컨테이너의 사이드카, Istio형태의 사이드카는 여기서 다루지 않는다.

멀티 컨테이너

일반적인 사이드카, 그리고 Istio의 사이드카의 차이는 다음과 같다.

구분	일반적인 사이드카	Istio 사이드카 (Envoy)
목적	특정 Pod 기능 보조	네트워크 트래픽 제어
통신 방식	Pod 내부 컨테이너끼리 localhost 통신	Envoy 프록시를 통해 모든 네트워크 통신 제어
사용 예시	로그 수집기, 캐싱, 보안	서비스 메시(mTLS), 트래픽 관리, 로깅
자동 주입 여부	없음 (YAML에서 직접 정의)	istio-injection=enabled 설정 시 자동 주입

멀티 컨테이너

일반적인 사이드카, 그리고 Istio의 사이드카의 차이는 다음과 같다.

구분	일반적인 사이드카	Istio 사이드카 (Envoy)
보안 기능	없음 (필요 시 추가해야 함)	mTLS, RBAC 지원
트래픽 모니터링	기본적으로 없음	Prometheus, Grafana, Kiali 활용 가능
복잡도	단순, 직접 컨테이너 정의	Istio 설정이 필요하여 비교적 복잡

멀티 컨테이너

멀티 컨테이너 구성은 다음과 같이 한다.

```
# vi multi-container-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
  volumeMounts:
    - name: shared-logs
      mountPath: /var/log/nginx
```

멀티 컨테이너

멀티 컨테이너 구성은 다음과 같이 한다.

```
  - name: log-processor
    image: busybox
    command: ["/bin/sh", "-c", "while true; do cat /var/log/nginx/access.log; sleep
5; done"]
    volumeMounts:
    - name: shared-logs
      mountPath: /var/log/nginx
  volumes:
  - name: shared-logs
    emptyDir: {}
# kubectl apply -f multi-container-pod.yaml
# kubectl get pod -f multi-container-pod.yaml
```

멀티 컨테이너

결과는 다음과 같이 출력이 된다.

```
NAME                  READY   STATUS    RESTARTS   AGE
multi-container-pod   2/2     Running   0          5s
```

멀티 컨테이너 내부에 접근하기 위해서 컨테이너 이름을 명시해야 한다.

```
# kubectl exec -it multi-container-pod -c nginx -- sh
/ # ls /var/log/nginx
access.log error.log
# kubectl logs multi-container-pod -c log-processor
127.0.0.1 - - [10/Mar/2025:12:00:01 +0000] "GET / HTTP/1.1" 200 612 "-"
"curl/7.64.1"
127.0.0.1 - - [10/Mar/2025:12:00:06 +0000] "GET / HTTP/1.1" 200 612 "-"
"curl/7.64.1"
```

정리

정리하면 다음과 같다.

사이드카 유형	주요 특징	사용 예시
일반 사이드카	개별 Pod 보조 기능 제공, localhost로 통신	로그 수집기, 캐싱, API 프록시
Istio 사이드카	네트워크 트래픽 관리, mTLS 보안 적용	서비스 메시, 트래픽 제어, 로깅

랩

하나의 Pod 내에 두 개의 컨테이너를 구성.

1. 메인 컨테이너: NGINX 웹 서버를 실행하여 기본 웹페이지를 제공
2. 사이드카 컨테이너: busybox를 사용하여 공유 볼륨에 주기적으로 로그 파일을 기록
3. 두 컨테이너는 shared-volume이라는 빈 디렉터리(emptyDir)를 통해 데이터를 공유
4. Pod의 라벨은 app: multi-container-demo로 지정하고, 두 컨테이너의 역할을 명확하게 구분

기록은 다음과 같은 방식으로 기록을 남김.

랩

배시 스크립트

```
while true; do
          echo "$(date) - log message from logger" >> /shared/log.txt;
          sleep 10;
        done
```

랩 코드

앞에 내용 계속 이어서…

```
# vi multi-container-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod
  labels:
    app: multi-container-demo
spec:
  volumes:
  - name: shared-volume
    emptyDir: {}
```

랩 코드

앞에 내용 계속 이어서…

```
containers:
 - name: nginx-main
   image: nginx
   volumeMounts:
   - name: shared-volume
     mountPath: /usr/share/nginx/html

 - name: busybox-sidecar
   image: busybox
   command: ["/bin/sh", "-c"]
```

랩 코드

앞에 내용 계속 이어서…

```
    args:
      - while true; do
          echo "[$(date '+%Y-%m-%d %H:%M:%S')] Sidecar is writing
log ..." >> /shared/log.txt;
          sleep 10;
        done
    volumeMounts:
    - name: shared-volume
      mountPath: /shared
```

대비 11

RBAC 및 Service Account

컨테이너 이미지 보안

NetworkPolicy

2025-04-18

쿠버네티스에서 제공하는 보안

대비 11

설명

바닐라 버전의 쿠버네티스의 보안 기능은 생각보다 많을 수 있고, 혹은 아닐 수 있다. 버전마다 다르지만, 최신 버전 기준으로 다음과 같은 기능을 제공한다.

보안 기능	설명	적용 범위	설정 방법
Role-Based Access Control	역할 기반 접근 제어	클러스터, 네임스페이스	Role, RoleBinding, ClusterRole, ClusterRoleBinding
ServiceAccount	Pod가 API 요청을 할 때 사용하는 계정	Pod, 네임스페이스	kubectl create serviceaccount
Pod Security Admission (PSA)	Pod 보안 정책을 적용하여 실행 제한	Pod, 네임스페이스	kubectl label ns <namespace> pod-security.kubernetes.io/enforce=restricted
Network Policy	네트워크 트래픽 제어 (Ingress, Egress)	네임스페이스, Pod	kind: NetworkPolicy
Pod Security Context	컨테이너 보안 설정 (UID, GID, 권한)	Pod, 컨테이너	securityContext

설명

몇몇 기능은 사용하기 위해서 클러스터에서 기능을 추가해야 한다. 그 대표적인 케이스는 Istio서비스 이다.

보안 기능	설명	적용 범위	설정 방법
Network Policy	네트워크 트래픽 제어 (Ingress, Egress)	네임스페이스, Pod	kind: NetworkPolicy
Pod Security Context	컨테이너 보안 설정 (UID, GID, 권한)	Pod, 컨테이너	securityContext 속성 설정
Secrets & ConfigMap	비밀번호, API 키, 설정값 관리	클러스터, 네임스페이스	kubectl create secret, kubectl create configmap
TLS & mTLS (Istio)	서비스 간 트래픽 암호화	Pod, 서비스	mutual TLS, cert-manager
Audit Logging	쿠버네티스 API 요청 기록	클러스터	--audit-policy-file 설정

2025-04-18

설명

커널/네트워크 그리고, 런타임 영역은 쿠버네티스가 설정 파일을 전달 후 적용되는 형식이다.

보안 기능	설명	적용 범위	설정 방법
ETCD 암호화	etcd에 저장된 데이터 암호화	클러스터	encryption-config.yaml 사용
AppArmor & Seccomp	커널 보안 프로파일 적용	컨테이너, Pod	securityContext.seccompProfile
KMS Provider	외부 키 관리 시스템(KMS)과 연동	클러스터	EncryptionConfig 설정
Runtime Security (Falco, Seccomp)	실행 중인 컨테이너 보안 모니터링	컨테이너	Falco, Seccomp 설정
Image Security (Notary, Cosign)	컨테이너 이미지 서명 및 검증	컨테이너	Cosign, Notary 사용

2025-04-18

설명

하지만, 시험에는 거의 나오지 않으니 참고만 한다. :)

쿠버네티스에서 TLS사용하기

대비 11

쿠버네티스에서 TLS사용하기

쿠버네티스에서 TLS키를 서비스에 사용하기 위해서 여러가지 방식이 있다.

- Istio 적용
- TLS키 사용

프라이빗 TLS키 생성 후, 클러스터에 적용 하도록 한다.

TLS키 생성

TLS키를 만들기 위해서 openssl명령어로 생성한다. 생성 후 secret키에 등록한다.

```
# openssl req -x509 -nodes -days 365 -newkey rsa:2048 \  -keyout tls.key -
out tls.crt -subj "/CN=my-nginx/O=my-nginx"
Generating a RSA private key
.....................................................................
.....................++++
.....................................................................
.............++++
writing new private key to 'tls.key'
-----
# kubectl create secret tls my-nginx-tls --cert=tls.crt --key=tls.key
secret/my-nginx-tls created
```

TLS+NGINX 적용

Nginx이미지는 TLS가 비활성화가 기본값이다. ConfigMap를 통해서 설정을 활성화한다.

```
# vi nginx-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  default.conf: |
    server {
        listen 443 ssl;
        ssl_certificate /etc/nginx/ssl/tls.crt;
        ssl_certificate_key /etc/nginx/ssl/tls.key;
        location / {
            root /usr/share/nginx/html;
            index index.html;
        }
    }
# kubectl apply -f nginx-config.yaml
```

TLS+NGINX 적용

생성한 TLS키를 NGINX서비스에 적용한다.

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

TLS+NGINX 적용

```
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 443
```

TLS+NGINX 적용

```
      volumeMounts:
        - name: tls-cert
          mountPath: "/etc/nginx/ssl"
          readOnly: true
       volumeMounts:
        - name: nginx-config
          mountPath: "/etc/nginx/conf.d"
          readOnly: true
```

TLS+NGINX 적용

```
      volumes:
      - name: tls-cert
        secret:
          secretName: my-nginx-tls
      - name: nginx-config
        configmap:
           name: nginx-config
# kubectl apply -f nginx-tls.yaml
# kubectl get -f nginx-tls.yaml
```

TLS+NGINX 적용

적용이 되면, Secret은 /etc/nginx/ssl에 마운트가 된다. POD 구성이 완료가 되면, Service를 생성하여 외부에 노출 및 접근이 가능하도록 한다. LoadBalancer가 설치가 안되어 있으면 NodePort로 구성하여도 상관 없다.

```
# vi nginx-tls-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-tls-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 443
    targetPort: 443
  type: LoadBalancer
# kubectl apply -f nginx-tls-service.yaml
```

적용확인

다음과 같은 명령어로 올바르게 구성이 되었는지 확인한다.

```
# curl -k https://34.120.50.20
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
</head>
<body>
<h1>Success! Nginx is running with TLS</h1>
</body>
</html>
```

쿠버네티스 kubeconfig*

대비 11

쿠버네티스 kubeconfig

다중 클러스터를 사용하는 경우 kubectl은 여러 클러스터의 정보를 특정 위치에 저장해서 가지고 있다. 보통 이 정보는 다음과 같이 확인한다. 현재 정보는 **클러스터 A**에 접근이 되어 있다.

```
# kubectl config view
apiVersion: v1
clusters:
- cluster:
    server: https://192.168.1.10:6443
  name: cluster-A
contexts:
- context:
    cluster: cluster-A
    user: user-A
  name: cluster-A
current-context: cluster-A
```

쿠버네티스 kubeconfig

클러스터 B에 접근하기 위해서 다음과 같이 설정한다. 아래와 명령어를 실행한다.

```
# export KUBECONFIG=~/.kube/config:~/kubeconfigs/cluster-b-config
# kubectl config view --merge
```

실행 후, 설정파일(kubeconfig)를 확인하면 다음과 같이 화면에 출력이 된다.

쿠버네티스 kubeconfig

```
clusters:
- cluster:
    server: https://192.168.1.10:6443
  name: cluster-A
- cluster:
    server: https://192.168.1.20:6443
  name: cluster-B
contexts:
- context:
    cluster: cluster-A
    user: user-A
  name: cluster-A
```

쿠버네티스 kubeconfig

```
- context:
    cluster: cluster-B
    user: user-B
  name: cluster-B
```

쿠버네티스 kubeconfig

올바르게 클러스터 변경이 되었는지 확인한다.

```
# kubectl config current-context
cluster-A
# kubectl config use-context cluster-B
Switched to context "cluster-B"
# kubectl config current-context
cluster-B
```

쿠버네티스 kubeconfig

특정 클러스터의 노드 정보를 확인하기 위해서 아래처럼 실행한다.

```
# kubectl --context=cluster-A get nodes
NAME           STATUS   ROLES           AGE   VERSION
node-a1        Ready    control-plane   100d  v1.31.0
node-a2        Ready    worker          100d  v1.31.0
# kubectl --context=cluster-B get nodes
NAME           STATUS   ROLES           AGE   VERSION
node-b1        Ready    control-plane   200d  v1.30.0
node-b2        Ready    worker          200d  v1.30.0
```

RBAC 및 Service Account*

대비 11

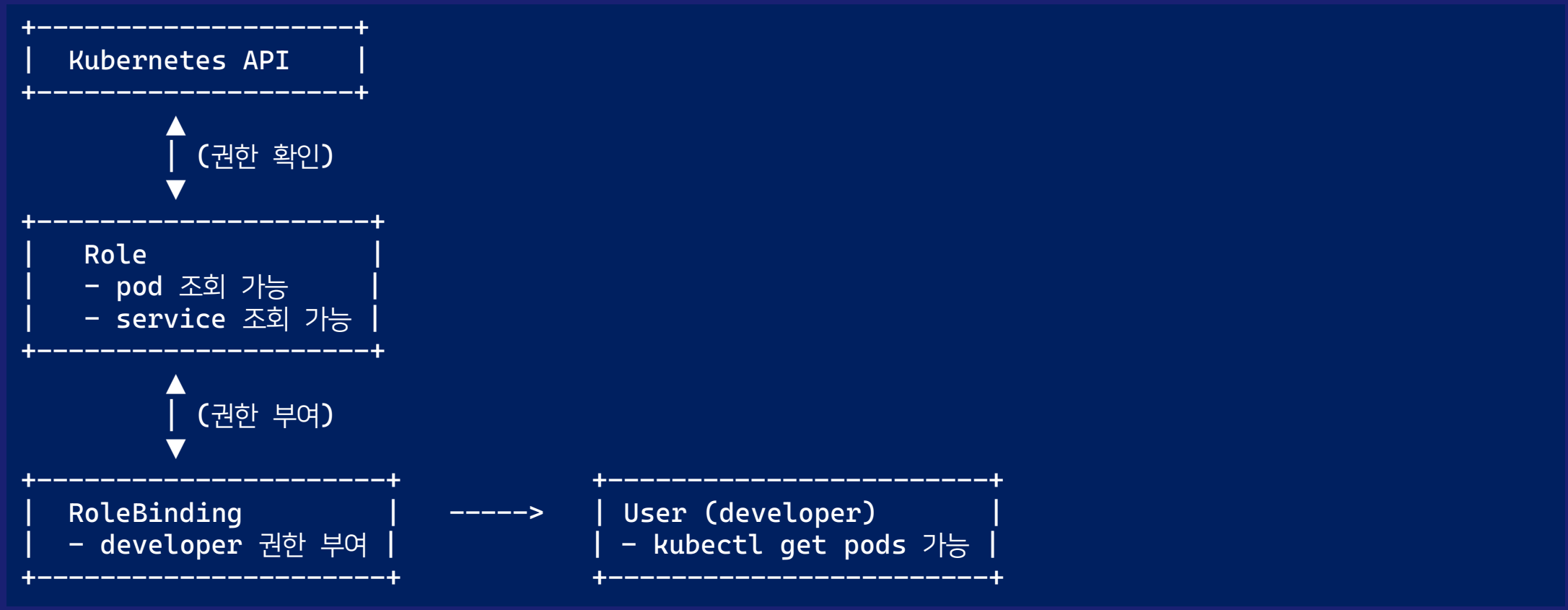
RBAC 및 Service Account

앞에서 사용하였던 RBAC를 좀 더 깊게 들어간다. RBAC생성은 가급적이면 명령어보다는 YAML기반으로 생성을 권장한다. 그 이유는, 명령어로 하는 경우 혼동하는 경우가 있다.

하지만, 시험에서는 보통 단순하게 나오기 때문에 명령어로 작성하여도 무관하다. RBAC의 구성 요소는 다음과 같다.

- User(사용자): Kubernetes 리소스에 접근하는 사용자 또는 서비스 계정
- Role / ClusterRole: 특정 리소스에 대한 권한 정의 (pods, services, deployments 등)
- RoleBinding / ClusterRoleBinding: 특정 사용자에게 Role 또는 ClusterRole을 연결

RBAC


Kubernetes API
(권한 확인)
Role
- pod 조회 가능
- service 조회 가능
(권한 부여)
RoleBinding
- developer 권한 부여
----->
User (developer)
- kubectl get pods 가능

생성

RBAC기능을 구현하기 위해서 네임스페이스부터 생성을 시작한다.

```
# kubectl create namespace dev-namespace
namespace/dev-namespace created
```

이전과 동일하게 POD만 접근이 가능하도록 한다.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: dev-namespace
  name: developer-role
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

생성

이전과 동일하게 POD만 접근이 가능하도록 한다.

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

생성

사용자에게 네임스페이스 및 역할을 할당 및 연결한다.

```
# vi developer-rolebinding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: developer-rolebinding
  namespace: dev-namespace
subjects:
- kind: User
  name: developer-user  # 사용자 이름
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: developer-role
  apiGroup: rbac.authorization.k8s.io
# kubectl apply -f developer-roleBinding.yaml
```

적용

이미 위에서 적용하였지만, 적용이 올바르게 되면 다음과 같이 화면에 메시지가 출력이 된다.

```
# kubectl apply -f developer-role.yaml
# kubectl apply -f developer-rolebinding.yaml
role.rbac.authorization.k8s.io/developer-role
createdrolebinding.rbac.authorization.k8s.io/developer-rolebinding created
```

접근이 잘 되는지 다음 명령어로 확인한다.

```
# kubectl auth can-i get pods --namespace=dev-namespace --as=developer-user
yes
# kubectl auth can-i get services --namespace=dev-namespace --as=developer-user
no
```

2025-04-18

적용

클러스터의 모든 자원에 접근을 허용하는 경우, 다음과 같이 구성한다. 여기서도 POD만 접근 허용한다.

```
# vi cluster-reader.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

적용

앞에 내용 이어서…

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-reader-binding
subjects:
- kind: User
  name: developer-user
  apiGroup: rbac.authorization.k8s.io
```

적용

앞에 내용 이어서…

```
roleRef:
  kind: ClusterRole
  name: cluster-reader
  apiGroup: rbac.authorization.k8s.io
# kubectl apply -f cluster-reader.yaml
```

적용된 내용을 확인하기 위해서 kubectl get -f 명령어로 확인이 가능하다.

```
# kubectl get -f developer-rolebinding.yaml
# kubectl get -f developer-role.yaml
# kubectl get -f cluster-reader.yaml
```

명령어로 적용

위에서 수행하였던 작업을 명령어 기반으로도 가능하다.

```
# kubectl create namespace dev-namespace
namespace/dev-namespace created
# kubectl create role developer-role \
  --verb=get --verb=list \
  --resource=pods \
  --namespace=dev-namespace
role.rbac.authorization.k8s.io/developer-role created
# kubectl create rolebinding developer-rolebinding \
  --role=developer-role \
  --user=developer-user \
  --namespace=dev-namespace
rolebinding.rbac.authorization.k8s.io/developer-rolebinding created
```

명령어로 적용

마지막으로 아래와 같이 RBAC상태 확인이 가능하다.

```
# kubectl auth can-i get pods --namespace=dev-namespace --as=developer-user
yes
```

서비스 계정

대비 11

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
토큰 사용 여부	Secret을 통해 토큰 제공 (JWT 기반)	일반적으로 패스워드 또는 인증 프로바이더 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
```

랩

특정 네임스페이스(예: dev)에서만 pod 목록을 조회할 수 있는 권한을 갖는 Role을 생성하고, 이를 ServiceAccount에 바인딩.

1. dev 네임스페이스에 read-pods라는 이름의 Role을 생성하여 pod 목록 조회(get, list, watch) 권한을 부여하는 YAML 파일을 작성
2. 동일 네임스페이스에 pod-reader라는 이름의 ServiceAccount를 생성
3. 작성한 Role을 pod-reader ServiceAccount에 RoleBinding을 통해 연결하는 YAML 파일을 작성

랩 코드

앞에 내용 계속 이어서…

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev
```

랩 코드

앞에 내용 계속 이어서…

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: read-pods
  namespace: dev
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

랩 코드

앞에 내용 계속 이어서…

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pod-reader
  namespace: dev
```

랩 코드

앞에 내용 계속 이어서…

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bind-read-pods
  namespace: dev
subjects:
- kind: ServiceAccount
  name: pod-reader
  namespace: dev
```

랩 코드

앞에 내용 계속 이어서…

```
roleRef:
  kind: Role
  name: read-pods
  apiGroup: rbac.authorization.k8s.io
```

랩

Kubernetes에서는 네임스페이스 범위 권한을 관리하는 Role과 클러스터 전체의 리소스에 대한 권한을 관리하는 ClusterRole 생성.

1. 사용자 cluster-admin이라는 계정이 없으면 생성
2. cluster-admin사용자는 클러스터 관리자 권한
3. 모든 네임스페이스 및 자원에 접근이 가능해야 됨

랩 코드

앞에 내용 계속 이어서…

```
# openssl genrsa -out cluster-admin.key 2048

# openssl req -new -key cluster-admin.key -out cluster-admin.csr -subj
"/CN=cluster-admin/O=system:masters"

# openssl x509 -req -in cluster-admin.csr -CA /etc/kubernetes/pki/ca.crt \
  -CAkey /etc/kubernetes/pki/ca.key -CAcreateserial \
  -out cluster-admin.crt -days 365
```

랩 코드

앞에 내용 계속 이어서…

```
# vi cluster-admin-binding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin-binding
subjects:
- kind: User
  name: cluster-admin  # CSR 또는 인증 기반 사용자 이름
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-admin  # 기본 제공되는 클러스터 관리자 역할
  apiGroup: rbac.authorization.k8s.io
```

랩

test 네임스페이스 내에서만 ConfigMap 생성 권한을 부여받아야 하는 ServiceAccount를 구성.

1. test 네임스페이스에 configmap-creator라는 이름의 ServiceAccount를 생성
2. test 네임스페이스에서 ConfigMap 생성(create) 권한만 부여하는 Role을 작성
3. RoleBinding을 작성하여 ServiceAccount와 Role을 연결

랩 코드

앞에 내용 계속 이어서…

```
apiVersion: v1
kind: Namespace
metadata:
  name: test
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: configmap-creator
  namespace: test
```

랩 코드

앞에 내용 계속 이어서…

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: create-configmap
  namespace: test
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["create"]
```

랩 코드

앞에 내용 계속 이어서…

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bind-configmap
  namespace: test
subjects:
- kind: ServiceAccount
  name: configmap-creator
  namespace: test
roleRef:
  kind: Role
  name: create-configmap
  apiGroup: rbac.authorization.k8s.io
```

랩 코드

앞에 내용 계속 이어서…

```
# kubectl auth can-i create configmap \
--as=system:serviceaccount:test:configmap-creator -n test
→ yes
# kubectl auth can-i delete configmap \
--as=system:serviceaccount:test:configmap-creator -n test
→ no
```

랩

클러스터 전체의 노드 정보를 조회할 수 있는 권한이 필요한 경우, ClusterRole과 ClusterRoleBinding을 활용.

1. 노드 조회(get, list, watch) 권한을 가진 ClusterRole을 YAML 파일로 작성
2. 이 ClusterRole을 클러스터 전체에 적용되도록 ClusterRoleBinding을 작성하고, 임의의 ServiceAccount(예: node-reader ServiceAccount)를 바인딩

랩 코드

앞에 내용 계속 이어서…

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: node-reader
  namespace: default
```

랩 코드

앞에 내용 계속 이어서…

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: node-read-role
rules:
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get", "list", "watch"]
```

랩 코드

앞에 내용 계속 이어서…

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: bind-node-read
subjects:
- kind: ServiceAccount
  name: node-reader
  namespace: default
roleRef:
  kind: ClusterRole
  name: node-read-role
  apiGroup: rbac.authorization.k8s.io
```

랩 코드

앞에 내용 계속 이어서…

```
# kubectl auth can-i list nodes --as=system:serviceaccount:default:node-reader
→ yes
```

NetworkPolicy

대비 11

설명

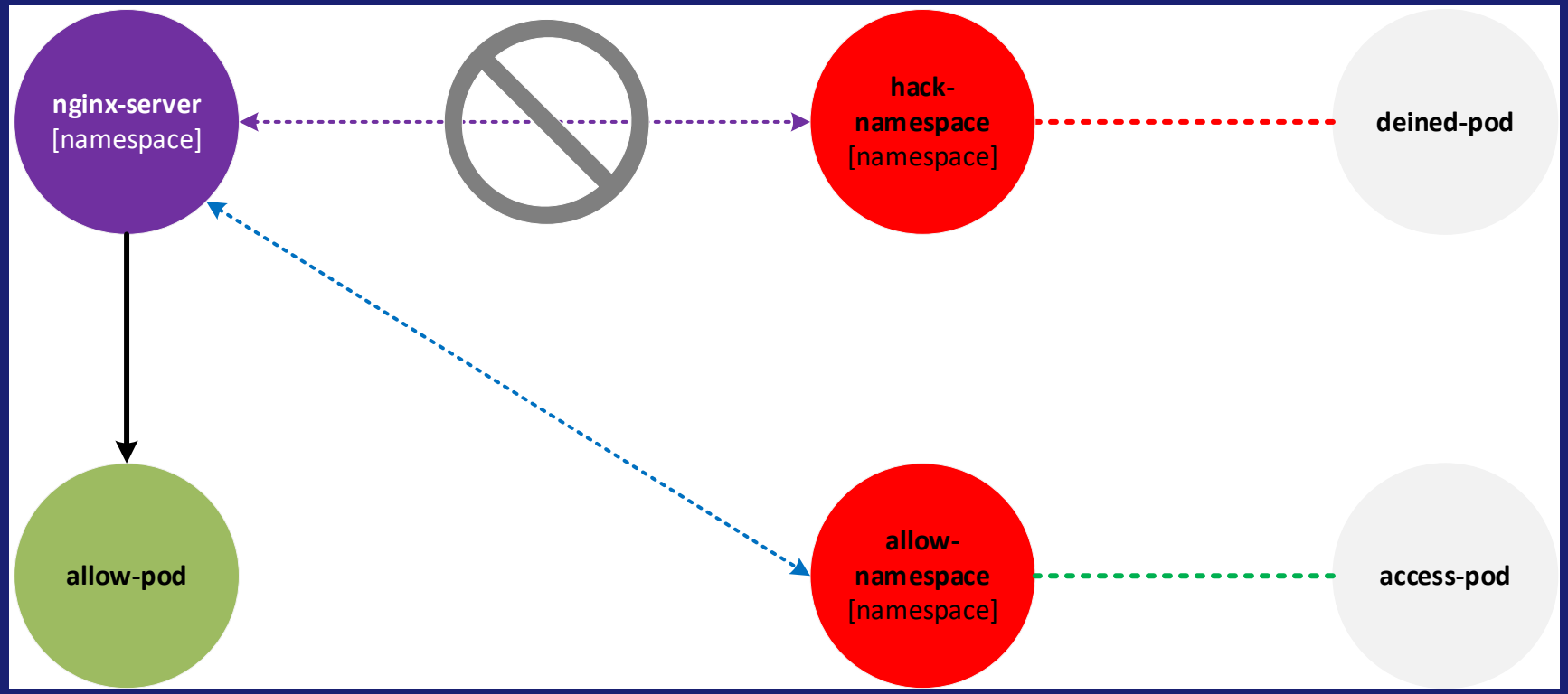
네임스페이스 간의 접근을 위한 기능으로, L2/L3/L4 레이어를 거치지 않고 라우팅을 설정함으로써 오버헤드 없이 신속하게 접근할 수 있다. 동작 방식은 아래 컴포넌트 기반으로 구성이 된다.

- 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


nginx-server
[namespace]
hack-
namespace
[namespace]
deined-pod
allow-pod
allow-
namespace
[namespace]
access-pod

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" 네임스페이스에서 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

랩

두 개의 네임스페이스(frontend와 backend)가 있다고 가정할 때, frontend 네임스페이스의 pod에서 오는 트래픽만 backend 네임스페이스의 pod에 인입하도록 제한하는 정책을 작성.

1. backend 네임스페이스 내의 pod를 대상으로 하는 NetworkPolicy를 작성
2. Ingress 규칙에서 namespaceSelector를 사용하여, frontend 네임스페이스에서 온 트래픽만 허용하도록 구성
3. 외부 네임스페이스 또는 frontend 네임스페이스가 아닌 경우 차단됨을 확인

랩 코드

앞에 내용 계속 이어서…

```
# kubectl create namespace frontend
# kubectl create namespace backend
```

랩 코드

파일 이름은 `allow-frontend-ingress.yaml`

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend-ingress
  namespace: backend
spec:
  podSelector: {}  # backend 네임스페이스의 모든 pod 대상
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          access: "frontend"
```

랩 코드

레이블 선언을 통해서 트래픽 허용.

```
# kubectl label namespace frontend access=frontend
## frontend에서 backend로 curl → 허용됨
# kubectl run test-frontend -n frontend --image=busybox -it --restart=Never
-- /bin/sh
# wget -qO- http://<backend-svc>  # 통과

## backend에서 backend로 curl → 차단됨
# kubectl run test-backend -n backend --image=busybox -it --restart=Never -
- /bin/sh
# wget -qO- http://<backend-svc>  # 연결 실패
## 다른 네임스페이스에서 backend로 → 차단됨
```

대비 12

CRD개념 및 구성

CRD개념 및 구성

Custom Resource

CRD 컨트롤러

CRD 자원 생성 및 관리

설명(CRD)

CRD는 쿠버네티스 API에 사용자가 새로운 정의 리소스(CRD)을 생성하기 위한 일종의 템플릿 혹은 선언 문서. 쿠버네티스에서 특정 기능을 제공하지 않는 경우, 사용자가 만들어서 직접 자원을 구성 할 수 있다.

예를 들어서 AI/ML서비스 선언이 별도로 필요한 경우, **`kind: nvidia-gpu, kind: amd-gpu,`** 이와 같은 형식으로 선언이 가능하다. 이렇게 선언하면 바로, CRD가 선언이다.

1. 새로운 리소스 타입(kind)의 스키마를 정의
2. 해당 리소스 타입이 쿠버네티스 내에서 사용될 수 있도록 API 확장(CustomResourceDefinition)
3. 예를 들어, nvidia-gpu이라는 CRD를 정의하면, 이제 nvidia-gpu라는 리소스 타입이 생성되고 관리 가능해짐

자세한 내용은 쿠버네티스 사이트에서 확인한다.

https://kubernetes.io/ko/docs/concepts/extend-kubernetes/api-extension/custom-resources/

쿠버네티스 기반 플랫폼들은 대다수가 CRD기반으로 구성이 되어 있다.

2025-04-18

설명(CRD)

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: myapps.example.com
spec:
  group: example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
```

설명(CRD)

```
            properties:
              replicas:
                type: integer
              image:
                type: string
  scope: Namespaced
  names:
    plural: myapps
    singular: myapp
    kind: MyApp
    shortNames:
      - ma
```

설명(CR)

CR의 다른 이름은 인스턴스(instance)라고 부르기도 한다. CRD를 통해서 구성한 선언을 통해서, 사용자가 원하는 값들을 선언을 하면 자원이 생성이 된다. 그래서 이를 인스턴스, 혹은 템플릿 인스턴스라고 부르기도 한다.

예를 들어서 Replicas, image, Ports,와 같은 부분도 CRD를 통해서 선언이 되어 있다. CR의 예제 코드는 다음과 같다.

```
apiVersion: example.com/v1
kind: MyApp
metadata:
  name: myapp-sample
spec:
  replicas: 3
  image: nginx:latest
```

설명

CRD는 기존 CR과 아래와 같은 차이점이 있다.

카테고리	CRD (사용자 정의 리소스 정의)	CR (사용자 정의 리소스)
정의 목적	사용자 정의 리소스를 생성하기 위한 스키마 정의	CRD로 정의된 리소스를 기반으로 실제 객체를 생성
역할	쿠버네티스에 새로운 리소스 유형 추가	사용자 정의 리소스의 개별 인스턴스를 관리
대상	쿠버네티스 API 확장	네임스페이스나 클러스터 내의 리소스 객체
예제 리소스 타입	myapps.example.com	myapp-sample
kubectl 명령어	kubectl get crd	kubectl get myapps

CRD 선언

CRD를 선언하기 위해서 다음과 같이 작성한다. 예를 들어서 GPU관련 API를 생성하였다고 한다. 아래와 같이 선언한다. 다만, 여기서 사용하는 이미지는 실제로 동작하지 않는다.

```
# vi nvidia-gpu-crd.yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: gpunvidias.nvidia.com
spec:
  group: nvidia.com
  versions:
    - name: v1
      served: true
      storage: true
```

CRD 선언

선언에 openAPIV3Schema라고 적어 있는 부분은, 이 CRD는 kubectl 및 API를 통해서 호출이 가능하다. 만약, openAPIV3Schema를 사용하지 않는 경우, kubectl로 조회가 안된다.

```
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                image:
                  type: string
                  default: "nvidia-operator"
```

CRD 선언

마지막으로, 선언 범주이다. 간단하게 네임스페이스 범주로 선언하며, 자원 이름은 gpunvidia로 되어 있으며 복수형은 gpunvidias로 선언. 마지막으로 형식은 GpuNvidia로 선언. `kubectl`명령어에서는 `kubectl gpunvidia` 혹은 `kubectl get gn`으로 가능.

```
  scope: Namespaced
  names:
    plural: gpunvidias
    singular: gpunvidia
    kind: GpuNvidia
    shortNames:
      - gpunvidia
      - gn
```

CR 생성

이 기반으로 인스턴스, 즉 자원을 생성한다. CRD는 앞서 이야기 하였지만, CRD일종의 템플릿처럼 동작한다. 사용자가 적절하게 값을 전달하면, 해당 값 기반으로 사용자가 원하는 자원을 쿠버네티스 클러스터에 생성한다. 이미지 영역은 선언하지 않으면 기본값으로 "nvidia-operator" 이미지를 사용한다.

```
# vi cr-gpunvidia.yaml
apiVersion: nvidia.com/v1
kind: GpuNvidia
metadata:
  name: gpu-nvidia
spec:
  image: "nvidia-operator"
```

CR 생성

CR를 적용한다.

```
# kubectl apply -f gpunvidia-crd.yaml
customresourcedefinition.apiextensions.k8s.io/gpunvidias.nvidia.com created
# kubectl apply -f gpu-nvidia-cr.yaml
gpunvidia.nvidia.com/gpu-nvidia created
# kubectl get gpunvidias
NAME         AGE
gpu-nvidia   10s
```

CR 생성

생성된 자원을 조회한다.

```
# kubectl describe gpunvidia gpu-nvidia
Name:          gpu-nvidia
Namespace:     default
API Version:   nvidia.com/v1
Kind:          GpuNvidia
Metadata:
  Creation Timestamp:  2024-03-03T10:00:00Z
   ...
Spec:
  Image:     nvidia-operator
```

랩

Foo라는 이름의 간단한 CRD를 작성하여 버전(v1)과 간단한 스펙 필드를 포함한 CRD 생성.

1. apiextensions.k8s.io/v1 API 버전을 사용하여 Foo CRD를 정의
2. CRD의 스펙에 spec 필드를 포함하고, string 타입의 message 필드를 필수로 지정
3. 작성한 CRD를 클러스터에 적용하고, 정상적으로 등록되었는지 확인

Foo CRD를 기반으로 실제 CustomResource(CR)를 생성해보세요.

1. 앞에서 만든 CRD를 활용하여, Foo 객체를 생성하는 YAML 파일을 작성합니다.
2. metadata.name 필드를 지정하고, 스펙에 message: "Hello, CRD!"와 같은 값을 포함하세요.
3. 생성한 CR을 kubectl get foo 명령어로 확인하고, YAML 출력(-o yaml)으로 상세 정보를 검토합니다.

랩 코드

파일 이름은 `allow-frontend-ingress.yaml`

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend-ingress
  namespace: backend
spec:
  podSelector: {}  # backend 네임스페이스의 모든 pod 대상
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          access: "frontend"
```

대비 13*

Ingress 소개

Ingress 서비스 구성

2025-04-18

Ingress 소개

대비 13

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.

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
ingress-nginx --create-namespace
NAME: nginx-ingress
LAST DEPLOYED:  ...
NAMESPACE: ingress-nginx
STATUS: deployed
```

Ingress 서비스 구성

대비 13

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
```

정리

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

2025-04-18

정리

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

랩

하나의 서비스(예: web-service)에 대해 호스트 이름 없이 단순하게 요청을 전달하는 Ingress 리소스를 작성.

1. 네임스페이스 lab-ingress에 웹 서비스와 Ingress 구성
2. web-service는 web이라는 Deployment에서 80 포트를 노출함
3. Ingress 리소스는 호스트 기반 없이, 모든 요청을 / 경로로 받아서 web-service로 전달
4. IngressClass는 nginx를 사용 (Ingress Controller가 설치되어 있어야 동작함)
5. curl <Ingress_IP> 시 → NGINX의 default 페이지가 보이도록 구성
6. kubectl describe ingress와 curl 결과로 확인

랩 코드

파일 이름은 `ingress-lab.yaml`

```
apiVersion: v1
kind: Namespace
metadata:
  name: lab-ingress
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  namespace: lab-ingress
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
```

랩 코드

앞에 내용 계속 이어서…

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

랩 코드

서비스 파일 생성. 파일 이름은 `web-server.yaml`으로 작성.

```
---
apiVersion: v1
kind: Service
metadata:
  name: web-service
  namespace: lab-ingress
spec:
  selector:
    app: web
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
```

랩 코드

앞에 내용 계속 이어서…

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

랩 코드

앞에 내용 계속 이어서…

```
spec:
  ingressClassName: nginx
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: web-service
            port:
              number: 80
```

랩 코드

앞에 내용 계속 이어서…

```
spec:
  ingressClassName: nginx
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: web-service
            port:
              number: 80
```

랩 코드

결과는 다음과 같이 확인이 가능하다.

```
# kubectl apply -f ingress-lab.yaml
# kubectl get ingress -n lab-ingress
NAME          CLASS    HOSTS   ADDRESS          PORTS   AGE
web-ingress   nginx    *       192.10.10.251    80      10s
# kubectl describe ingress web-ingress -n lab-ingress
Rules:
  Host        Path  Backends
  *           /     web-service:80
# curl http://<Ingress-External-IP>/
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

대비 14

쿠버네티스 패키지 소개

2025-04-18

쿠버네티스 패키지 소개

대비 14

2025-04-18

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

위의 기능을 비교하면 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 충실함	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.

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 유틸리티를 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 설치 및 사용하기

대비 14

설치

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, loadbalancer**를 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

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

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)

설치 시, 로드밸런서 설정 변경이 필요하다. **ippool, L2**에 대한 설정이 필요하다. 일반적으로 아래처럼 수정 및 변경한다.

```
# vi metallb-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
      - 192.168.1.240-192.168.1.250
# kubectl apply -f metallb-config.yaml
```

설치(metallb)

하지만, 가급적이면 `values.yaml` 파일을 통해서 수정 및 관리를 권장한다.

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

설치(metallb)

위의 내용 이어서…

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

설치(metallb)

위의 내용 이어서…

```
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(metallb)

기존에 설치 된 로드밸런서에 업데이트 및 확인한다.

```
# helm install 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
```

랩

클러스터 내에 있는 간단한 웹 애플리케이션(예: Nginx)을 NodePort 서비스로 노출하여, 클러스터 외부에서 접근할 수 있도록 구성.

1. Nginx Deployment를 작성하여 80번 포트로 동작하도록 구성
2. Deployment를 대상으로 하는 Service를 생성하는 YAML 파일을 작성
    - Service 타입을 NodePort로 지정
    - 서비스 포트는 80번, NodePort는 자동 할당 또는 원하는 범위(예: 30000~32767) 내에서 지정
3. kubectl apply -f <파일명> 명령어로 리소스를 클러스터에 적용
4. 클러스터 노드의 IP와 할당된 NodePort를 이용하여 브라우저나 curl 명령어로 외부에서 접근 가능한지 확인

랩 코드

파일 이름은 `nginx-nodeport.yaml`으로 작성한다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deploy
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
```

랩 코드

위의 내용 계속 이어서…

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

랩 코드

위의 내용 계속 이어서…

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
```

랩

NodePort 서비스를 생성할 때, 특정 NodePort 번호(예: 31000)를 고정으로 할당하여 서비스에 접근.

1. 위 문제에서 작성한 NodePort 서비스 YAML 파일을 수정하여, nodePort: 31000으로 고정
2. Deployment와 Service 리소스를 재 적용
3. 클러스터 노드 IP에 http://<노드_IP>:31000으로 접속하여 서비스가 정상 동작하는지 확인

랩 코드

아래와 같이 수정한다.

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    nodePort: 31000  # 생략하면 자동 할당됨(30000~32767 범위)
```

랩

클라우드 환경 또는 MetalLB와 같은 LoadBalancer 구현체를 활용하여, 외부 로드밸런서를 통해 서비스를 노출하는 연습.

1. 간단한 웹 애플리케이션(예: Nginx) Deployment를 구성
2. Service 리소스를 생성하는 YAML 파일을 작성
    - Service 타입을 LoadBalancer로 지정.
    - 내부 포트와 외부 포트를 적절히 매핑.
3. 클라우드 환경(GCP, AWS, Azure 등)에서는 자동으로 외부 IP가 할당되는지 확인하거나, MetalLB를 설치한 클러스터에서는 LoadBalancer IP를 할당받도록 설정
4. 할당된 외부 IP 주소를 통해 웹 애플리케이션에 접속하여 정상 동작하는지 확인

랩 코드

아래와 같이 수정한다.

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    nodePort: 31000  # 생략하면 자동 할당됨(30000~32767 범위)
```

대비 15

쿠버네티스 커스터마이즈 소개

커스터마이즈 구성

2025-04-18

쿠버네티스 커스터마이즈 소개

대비 15

설명

앞에서 이야기 하였지만, Kustomize는 쿠버네티스에서 빌트인으로 제공하는 기능이다. Helm하고 제일 큰 차이점은 다음과 같다.

1. 빌트인 패키지 관리자
2. 버전관리

Kustomize는 별도로 설정이 필요 없으며, 프레임워크 디렉터리 및 YAML파일 기반으로 구성 및 생성하면 된다. 다만, 버전관리가 안되기 때문에, GIT OPs기반으로 구성이 되어야 한다.

Kustomize의 디렉터리 구조는 다음과 같다.

설명

Kutomized는 프레임 워크 디렉터리 구조 및 구성을 가지고 있기 때문에, 아래와 같이 규칙을 사용한다.

```
my-app/
├── base
│   ├── deployment.yaml
│   ├── service.yaml
│   └── kustomization.yaml
└── overlays
    └── dev
        ├── kustomization.yaml
        └── patch-deployment.yaml
```

커스터마이즈 구성

대비 15

구성(DIRECTORY)

자, 위의 내용을 기반으로 간단하게 생성한다. 아래와 같이 디렉터리를 생성한다.

```
nginx-app/
└── base
    ├── deployment.yaml
    ├── service.yaml
    └── kustomization.yaml
```

구성(DEPLOYMENT)

deployment.yaml에 다음과 같이 작성한다.

```
# vi base/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
```

구성(DEPLOYMENT)

deployment.yaml에 다음과 같이 작성한다.

```
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest  # 초기 이미지 버전 (업그레이드 전)
        ports:
        - containerPort: 80
```

구성(SERVICE)

service.yaml에 다음과 같이 작성한다.

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

구성(PACKAGE)

service.yaml에 다음과 같이 작성한다.

```
# vi base/kustomization.yaml
resources:
  - deployment.yaml
  - service.yaml
# kubectl apply -f kustomization.yaml
```

구성(랜더링)

문제 없이 구성이 되었는지, YAML파일을 렌더링 한다.

```
# kubectl kustomize base
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
   ...
        containers:
        - name: nginx
          image: nginx:latest
          ports:
          - containerPort: 80
```

구성(SERVICE)

위의 내용 이어서… 내용은 조금 다를 수 있다.

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

구성(적용)

최종적으로 적용 및 생성한다.

```
# kubectl apply -k base
# kubectl get deployments nginx-deployment
# kubectl get services nginx-service
```

버전 변경을 원하는 경우, 다음과 같이 명시한다. 이 때는 별도로 YAML수정이 필요하지 않다. 필요한 경우 수정 후 적용하여도 상관 없다.

```
# kustomize edit set image nginx=nginx:1.21.0
# kubectl kustomize base
# kubectl apply -k base
```

구성(PATCH)

만약, Chart에 패치를 적용하기 위해서 아래와 같이 작성한다.

```
# vi nginx-app/overlays/dev/patch-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
```

구성(PATCH)

kustomization.yaml파일에 다음과 같이 추가한다.

```
# vi kustomization.yaml
resources:
  - ../../base
patchesStrategicMerge:
  - patch-deployment.yaml
```

구성(확인)

적용이 올바르게 되었는지 확인한다.

```
# kubectl kustomize overlays/dev
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
```

Overlay에 의해 수정됨

정리

base, overlay의 차이점은 다음과 같다.

항목	Base	Overlay
역할	공통으로 사용되는 기본 리소스 구성 파일	BASE에 환경별로 구성 가능, dev, test, product
목적	재사용 가능한 원본 매니페스트 제공	오버라이드 형식의 패치
구성	쿠버네티스에서 제공하는 자원들	패치파일 형식, 기존 자원을 수정
관리	한 번 정의되며 여러 환경에서 공통 사용	환경별로 분리하여 관리, 변경 사항만 기록

랩

nginx 웹 애플리케이션을 Helm Chart로 배포하고자 한다. 다음 조건을 만족하는 Helm Chart를 직접 생성하고 배포하시오.

1. Helm Chart 이름은 nginx-chart
2. 기본 포트는 80, Replicas는 2
3. Helm values를 사용해 이미지 버전(예: nginx:1.25.2)을 지정 가능하도록 설정
4. 생성한 Helm Chart를 로컬에서 설치하고 배포가 성공하는지 확인
5. 설치 후 kubectl get all 명령어로 리소스들이 정상 배포되었는지 확인

랩 코드

아래와 같이 수정한다.

```
# helm create nginx-chart
# tree -L 2 nginx-chart/
nginx-chart/
├── charts/
├── templates/
├── values.yaml
├── Chart.yaml

# cd nginx-chart
```

랩 코드

아래와 같이 작성한다.

```
# vi values.yaml
replicaCount: 2
image:
  repository: nginx
  pullPolicy: IfNotPresent
  tag: "1.25.2"
service:
  type: ClusterIP
  port: 80
```

랩 코드

아래와 같이 작성한다.

```
# vi templates/deployment.yaml
spec:
  replicas: {{ .Values.replicaCount }}
  containers:
    - name: {{ .Chart.Name }}
      image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"

# helm install my-nginx ./nginx-chart
```

랩 코드

올바르게 등록 되었는지 확인한다.

```
# helm list
NAME        NAMESPACE       REVISION    UPDATED         STATUS      CHART           APP VERSION
my-nginx    default         1           1m ago          deployed    nginx-chart-0.1.0   1.25.2
```

랩

다음 조건을 만족하는 Kustomize 디렉터리를 구성.

1. nginx Deployment를 base 디렉터리에 생성 (replicas: 1, port: 80)
2. overlays/dev 디렉터리를 생성하여 replicas: 2로 override
3. labels에 환경 구분용 env: dev를 추가
4. kubectl apply -k overlays/dev 명령으로 배포
5. 배포 후 kubectl get deployment nginx -o yaml 명령으로 dev 설정이 반영되었는지 확인

랩 코드

아래와 같이 디렉터리 생성 및 구성한다. 복잡하면 **overlays**는 생성하지 않아도 된다.

```
kustomize-lab/
├── base/
│   ├── deployment.yaml
│   └── kustomization.yaml
└── overlays/
    └── dev/
        ├── kustomization.yaml
        └── replica-patch.yaml
```

랩 코드

아래와 같이 작성한다.

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

랩 코드

앞에 내용 계속 이어서…

```
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.25.2
        ports:
        - containerPort: 80
```

랩 코드

아래와 같이 수정한다.

```
# vi base/kustomization.yaml
resources:
  - deployment.yaml
```

랩 코드

아래와 같이 작성한다.

```
# vi overlays/dev/replica-patch.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
```

랩 코드

아래와 같이 작성한다.

```
# vi overlays/dev/kustomization.yaml
resources:
  - ../../base

patchesStrategicMerge:
  - replica-patch.yaml

commonLabels:
  env: dev
```

랩 코드

최종 확인한다.

```
# kubectl apply -k overlays/dev
# kubectl get deployment nginx
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   2/2     2            2           10s
```

CKA-EXAM-101
done