

DUSTBOX

# 쿠버네티스 101

쿠버네티스 기본



# 내용

과정개요 .....	1
쿠버네티스 소개.....	1
1. 쿠버네티스는 무엇인가? .....	1
2. 쿠버네티스 구성 요소 .....	4
3. 쿠버네티스 표준 사양 .....	5
OCI (Open Container Initiative) .....	5
CNI (Container Network Interface) .....	6
CSI (Container Storage Interface) .....	6
CRI (Container Runtime Interface) .....	6
4. 인프라 코드화.....	7
5. 클라우드 네이티브.....	8
설치 환경 .....	8
1. 설치 권장 사양 .....	8
2. 기본 랩 구성 .....	9
3. 하이퍼바이저 소프트웨어 .....	10
4. 에디터 설정 .....	10
VI/VIM .....	10
NANO .....	11
ALE for VIM.....	11
5. 사용을 위한 확장도구 .....	11
tmux.....	11
쿠버네티스 설치 준비 및 설명 .....	13
1. 앤서블 기반 설치 도구 .....	13
kubespray .....	13
duststack-k8s-auto.....	13
2. 간단한 클러스터 생성 도구 .....	14
kind.....	14
minikube .....	15

3.	부트 스트랩 명령어 설명.....	16
	init.....	16
	join.....	19
	upgrade.....	20
	config.....	20
	token.....	20
	reset.....	20
	version.....	21
	alpha .....	21
4.	윈도우 가상머신 .....	21
5.	설치준비: 공통 설정.....	23
6.	쿠버네티스 수동설치 .....	25
	쿠버네티스 저장소 .....	25
	런타임 설치 및 구성 .....	26
	firewalld.....	28
	tc/swap.....	30
	/etc/sysconfig/kubelet .....	30
7.	쿠버네티스 설치: 싱글 마스터 설치.....	30
8.	쿠버네티스 설치: 다중 마스터 설치.....	33
9.	쿠버네티스 설치: 네트워크 구성 .....	35

## 표준 컨테이너 도구 ..... 37

1.	표준 OCI 도구와 관계 .....	37
2.	Buildah .....	37
3.	Skopeo .....	37
4.	Tekton(테크톤).....	38
5.	Containerd/CRI-O/CRI-Docker .....	38
	Containerd.....	42
	CRI-O.....	42
	Podman(포드만) .....	43
6.	리눅스 커널과 쿠버네티스 관계 .....	43
7.	파일시스템 .....	45

## 쿠버네티스 아키텍처..... 47

1. 가상화 vs 컨테이너 .....	47
2. RUNC/CRUN .....	49
3. 쿠버네티스 컴포넌트 .....	49
kubelet.....	50
POD/Pause.....	51
etcd .....	51
coredns.....	53
kube-proxy(master, worker).....	54
kube-scheduler.....	55
kube-controller-manager.....	56
kube-apiserver .....	58
4. 마스터 노드 .....	59
5. 워커노드(minion, compute) .....	59
스왑 사용 .....	60
6. Pod(po).....	61
7. Service(svc).....	62
8. Deployment(deploy).....	63
9. DeploymentConfig .....	64
10. ReplicaSet(rs) .....	64
11. Replication Controller(rc).....	64

## 기본기능 ..... 65

1. 컨텍스트 관리.....	65
2. 관리를 위한 확장 명령어.....	65
3. 자원 이름.....	65
4. 명령어 자동완성 .....	66
5. 노드 상태 확인.....	67
6. YAML 문법 .....	68
연습문제 .....	72
7. 네임스페이스.....	72
연습문제 .....	78

8. 생성(create/apply) .....	79
연습문제 .....	83
9. 실행(run) .....	84
연습문제 .....	85
10. 멀티 컨테이너 구성 .....	86
연습문제 .....	87
11. get .....	87
연습문제 .....	88
12. describe/logs .....	88
연습문제 .....	89
13. cp .....	89
연습문제 .....	89
14. exec .....	90
연습문제 .....	90
15. expose/service/endpoint .....	90
expose .....	90
service .....	93
endpoints .....	95
연습문제 .....	95
16. proxy, port-forward .....	95
17. label .....	95
연습문제 .....	96
18. set .....	96
연습문제 .....	97
19. edit .....	97
연습문제 .....	97
20. delete .....	97
연습문제 .....	98
21. diff .....	98
연습문제 .....	99
22. debug/logs .....	99
연습문제 .....	100
23. explain .....	100

24. replace .....	101
연습문제 .....	101
25. patch .....	101
연습문제 .....	101

**프로젝트 구성 및 애플리케이션 배포..... 102**

1. 시나리오.....	102
2. 구성 및 소프트웨어.....	102

**고급기능 ..... 102**

1. 쿠버네티스 스토리지 .....	102
스토리지 서버 구성 및 개념 설명.....	102
랩을 위한 NFS 서버 구축 .....	103
StorageClass ServiceAccount .....	104
StorageClass.....	107
Persistent Volume (PV).....	111
Persistent Volume Claim (PVC).....	112
2. 로드 밸런서 .....	115
MetalLB 설치 .....	115
MetalLB 서비스 구성.....	116
3. helm.....	119
연습문제 .....	121
4. metrics 설치.....	121
사용 및 확인.....	122
5. 쿠버네티스 컨텍스트 및 사용자 구성 .....	123
사용자 설명.....	124
사용자 생성.....	124
연습문제 .....	127
6. Role/RoleBinding 생성 및 구성.....	127
Role .....	127
RoleBinding.....	130
7. 역할기반제어(RBAC) 설명 .....	133
연습문제 .....	134
8. Service Account.....	136
연습문제 .....	137

9. scale/rollout/rollback/history .....	138
연습문제 .....	141
10. 선호도 및 예외시간(Affinity/Tolerations) .....	141
포드 선호도(Pod Affinity).....	141
노드 선호도(Node affinity).....	142
오염 및 내성 (Taints/Tolerations).....	144
연습문제 .....	146
11. 변수 전달.....	146
12. 자동조절(autoscale).....	147
연습문제 .....	151
13. drain/taint/cordon/uncordon .....	151
비우기(drain).....	151
오염 및 차단 (taint/cordon, uncordon).....	152
14. 노드 추가 및 제거(Node add/remove).....	154
15. Service HealthCheck .....	155
상태 확인 소개(HealthCheck).....	155
애플리케이션 상태 확인(Liveness).....	158
포드 상태확인(Readiness ) .....	161
16. Label, annotations, selectors.....	165
이름표(label) .....	166
선택자(selector).....	168
주석(annotations).....	170
노드 셀렉터(Node Selector) .....	171
17. 데몬 서비스(DaemonSet) .....	172
18. 상태 POD 서비스 (StatefulSets).....	175
19. 작업 (Jobs/CronJobs).....	175
작업(Jobs) .....	175
크론잡(CronJobs, cj) .....	177
20. 설정파일(ConfigMaps(cm)) .....	179
연습문제 .....	182
21. 비밀(Secrets(sc)).....	182
연습문제 .....	187
22. 배포 및 배치(deployment).....	188
배포정책 .....	191
연습문제 .....	191



23. 복제자(ReplicaSet, Replication Controller).....	192
복제자와 배포자의 관계 .....	193
연습문제 .....	196
24. DaemonSet .....	196
25. 외부 아이피 주소(ExternalIP) .....	196
연습문제 .....	197
26. 외부 도메인 주소(ExternalName).....	197
연습문제 .....	197
27. 네트워크 정책 (NetworkPolicy).....	197
연습문제 .....	202

**쿠버네티스 기능 확장..... 203**

1. MetalLB.....	203
2. Ingress .....	204
Nginx 소개 및 설명 .....	204
HAProxy 소개 및 설명.....	210
3. Gogs .....	211
설치 .....	211
4. docker-registry .....	217
설치 .....	217
5. ArgoCD.....	223
설치 .....	223
6. ansible.....	225

**DEVOPS 를 위한 준비 ..... 226**

1. 데브옵스는 무엇인가? .....	226
2. 표준 컨테이너 도구.....	227
podman .....	227
skopeo .....	228
buildah.....	229
3. 쿠버네티스 앤서블.....	230
설치 .....	231
랩.....	231

4. Tekton.....	231
설명 .....	231
설치 .....	231
단계(step).....	232
작업(task).....	232
파이프라인(pipeline).....	234
워크스페이스/결과(workspace, result).....	235
테크톤 확장 기능 .....	235
테크톤 활용 예제.....	236
5. Istio.....	238
설치 .....	238
6. knative .....	240
설명 .....	240
설치 .....	240

## 관리 및 운영..... 242

1. 클러스터 업그레이드 .....	242
2. 소프트웨어 L4 기반 마스터 로드 밸런서.....	242
3. ETCD 백업 및 복원 .....	242
4. 컨테이너 레지스트리 서버 구성 .....	242
5. External ETCD 서버 구성.....	242
6. External DNS 서버 구성.....	242
7. 쿠버네티스 스케줄러 수정 및 변경하기.....	242
8. 로그 및 트러블 슈팅을 위한 방법 .....	242
9. 이미지 복사 및 배포.....	243
10. 외부 계정 연결.....	243
11. kube-virt.....	243
12. 클러스터 연합.....	243



# 과정개요

이 책은 쿠버네티스 경험이 없는 사용자를 위해서 작성이 되었다. 쿠버네티스 설치 및 기본적인 명령어, 그리고 YAML 작성 방법에 대해서 정리하였다. 랩을 진행하기 위해서는 최소한 윈도우 10/11 에서 하이퍼브이 기반으로 랩 진행을 권장한다. 이 책의 주요 목적은 다음과 같다.

1. 쿠버네티스 설치 방법
2. YAML 작성 방법 및 리소스 관리
3. 명령어 기반으로 자원관리
4. 프로덕트 환경에서 사용이 가능한 환경 제안

이 문서는 쿠버네티스 1.27 버전 기준으로 작성되었으며, 랩에서 사용하는 소프트웨어 목록은 다음과 같다.

소프트웨어 이름	버전	용도
Kubernetes	1.27	컨트롤러 및 컴퓨트 노드 구성.
Calico Network		클러스터 vxlan 네트워크 구성
Ingress-nginx		도메인 프록시 서비스 구성
Ingress-haproxy		도메인 프록시 서비스 구성
cis-nfs-driver		NFS 스토리지 제공
gogs		GIT 서버 제공
docker-registry		컨테이너 이미지 서버

## 쿠버네티스 소개

### 1. 쿠버네티스는 무엇인가?

쿠버네티스는 컨테이너 오케스트레이션 플랫폼 혹은 컨테이너 인프라 플랫폼을 제공해주는 소프트웨어이다. 쿠버네티스가 만들어지기 전에, 많은 회사들이 90 년대말부터 컨테이너 오케스트레이션 도구들을 만들기 시작했다.

하지만, 이때 당시, 리눅스 진영에서는 컨테이너 및 가상화 기술이 많이 부족한 상태였고, BSD 에서 사용하던 chroot 와 같은 도구를 통해서 프로세스 격리 기능 구현하였다.

기본적인 컨테이너 기술 및 개념은 1979 년도 유닉스 시스템에서 chroot 와 같은 기능을 통해서 구현이 되었으며, 이 기술이 BSD 에 도입이 되었으며, 이를 기반으로 FreeBSD Jails 를 구현 및 구성하였다. 리눅스는 2000 년도 초부터 가상화 및 컨테이너 기술에 대해서 고려를 시작하였고, 이를 vServer 라는 이름으로 시작했다. 리눅스 컨테이너 프로젝트는 리눅스 가상화 프로젝트보다 늦게 시작이 되었다. 쿠버네티스 관련 프로젝트는 2006 년도에 구글 인프라팀에서 내부적으로 시작하였으며, 이 시점에 도커와 같은 런타임과 같은 도구들이 많이 개발이 되기 시작하였다.

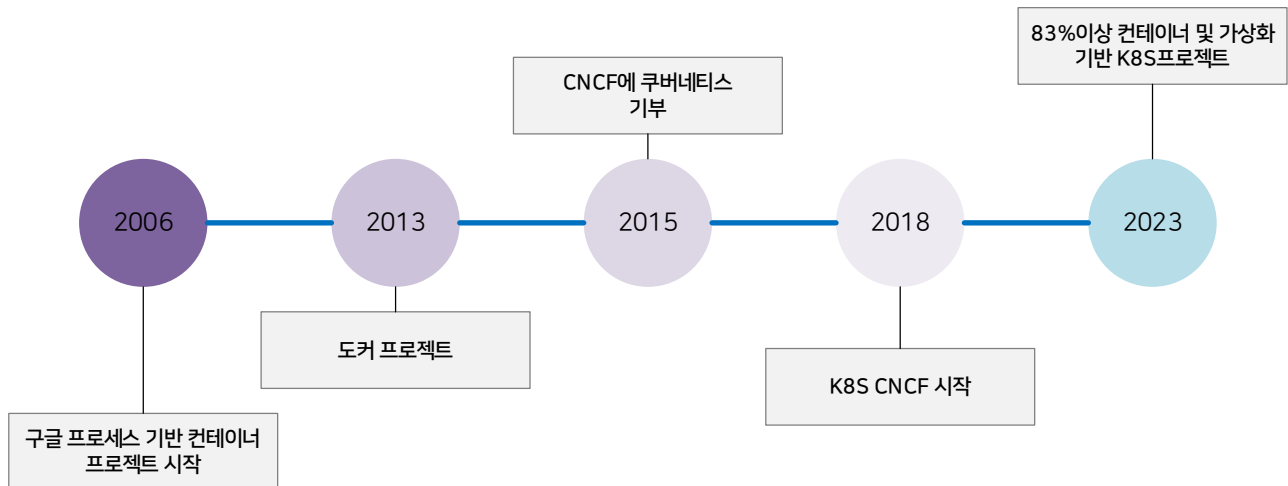


그림 1 컨테이너 프로젝트 소개

2000 년도, 닷컴 버블<sup>1</sup>이 시작된 시점에, 가상머신 혹은 베어메탈 기반으로 빠르게 서비스 확장 및 업데이트가 어렵다고 판단이 되었다. 부트-업이 빠르게 프로그램 실행이 가능한 구조 및 아키텍처가 필요하기 시작하였다. 이때 빅-테크 기술 업체, 예를 들어서, 구글/레드햇/IBM 와 같은 회사들은 가상머신이 아닌, 좀 더 가벼운 기반의 인프라를 원했다. 구글은 적극적으로 프로세스를 격리시키는 기술에 개발 및 참여를 하였고, 이때 나온 결과가 cgroup 이라는 프로그램이다.

그 후, 레드햇은 namespace 기능을 커널에 구현이 되었고, 마침내 이 두 기술이 커널에 통합이 되면서, 리눅스 컨테이너 프로젝트가 본격적으로 시작하게 되었다. 이때 만들어진 기술은 chroot<sup>2</sup>, lxc<sup>3</sup>와 같은 컨테이너 도구가 릴리즈 되기 시작하였다. 컨테이너가 본격적으로 시스템에 도입이 된 시점은 도커 기반의 루트리스-컨테이너(rootless-container)가 매우 큰 역할을 하였다.

기존에 사용하던 컨테이너 시스템과 도커와 제일 큰 차이점은 루트리스 기반을 완벽하게 지원하며, 기존 lxc와 다르게 순수하게 **사용자 영역(userspace)에서 구현 및 사용**<sup>4</sup>이 가능하게 되었다. 물론, 파일 시스템이나 네트워크 부분은 리눅스 커널을 통해서 제공받기 때문에, 리눅스 커널의 기능이나 성능이 매우 중요하다.

<sup>1</sup> 닷컴 버블은 2000 년도 초에 시작이 되었다. 지금은 AI 버블이라고 불리는 시점이다. (저자사건)

<sup>2</sup> chroot 는 프로세스 기반으로 격리한다. chroot 는 BSD 에서 가져온 기술 중 하나이다.

<sup>3</sup> 커널 기반 격리 컨테이너. <https://linuxcontainers.org/lxc/>

<sup>4</sup> 이미지 관리, 볼륨 및 네트워크 관리가 사용자 입장에서 훨씬 수월하다.

이 부분에 대해서는 추후 런타임 부분에서 좀 더 이야기하도록 하겠다.

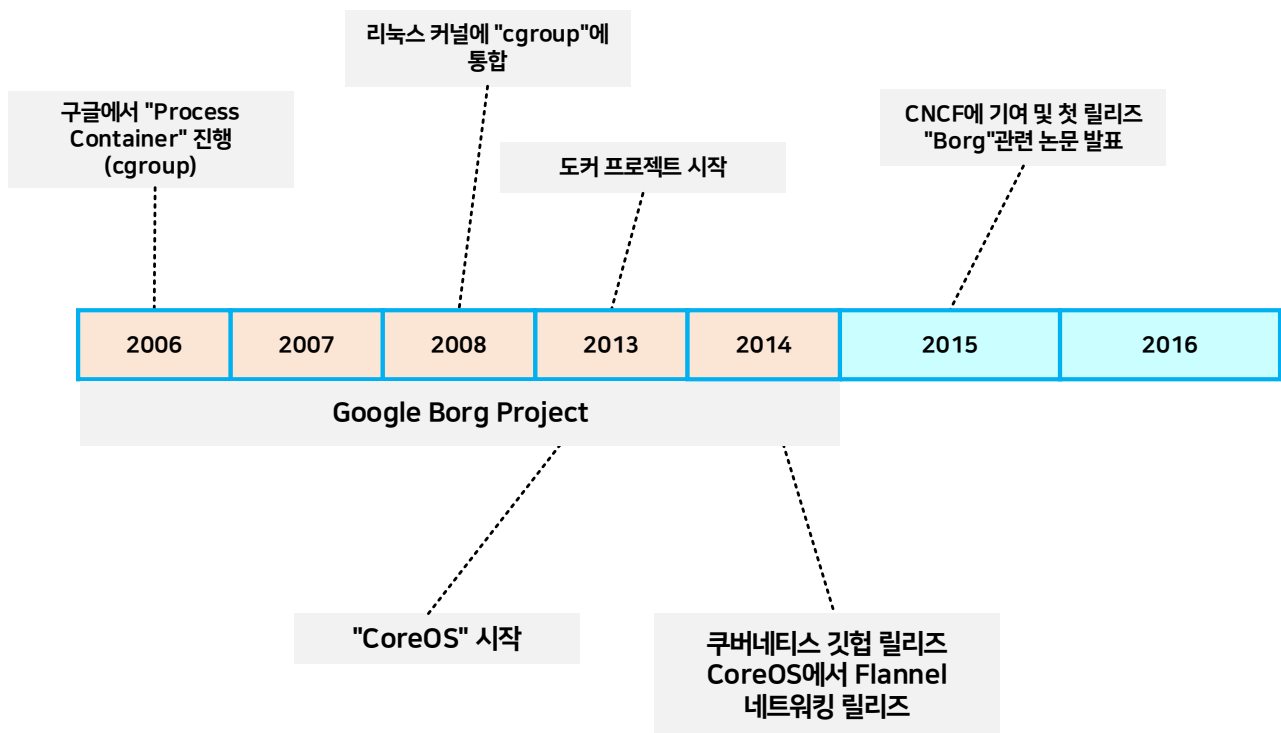


그림 2 보그 프로젝트

구글은 cgroup 기반으로 컨테이너 시스템을 구현하였고, 이를 "Process Container"라고 이름을 붙였다. 다만, Jails 처럼 격리를 하기 위해서는 추가적인 기능이 필요 하였는, 방법은 두 가지가 있다.

1. 하이퍼바이저 혹은 런타임 형식으로 구현
2. 호스트 자원 공유기반

컨테이너는 기본적으로 하이퍼바이저 방식처럼 모든 자원을 생성하지 않는다. 커널에서 자원을 공유할 수 있도록 자원을 매핑이 가능하도록 namespace<sup>5</sup>기능을 만들어서 이 문제를 해결하였다. 이 시점에 레드햇은 CoreOS 를 인수하였고, 기존 Atomic Project 를 중지하였다. 현재 레드햇에서 제공하는 컨테이너 플랫폼 제품들은 CoreOS 기반으로 구성이 되어 있다. 커뮤니티 배포판 경우에도 OSTree<sup>6</sup>기반으로 immutable 배포판을 제작해서 배포하고 있다.

<sup>5</sup> namespace 는 chroot 기능과는 다르다. chroot 는 1979 년 Unix V7, 1982 년 BSD 에 추가가 되었다.

<sup>6</sup> OSTree, [GitHub - ostreedev/ostree: Operating system and container binary deployment and upgrades](https://github.com/ostreedev/ostree)

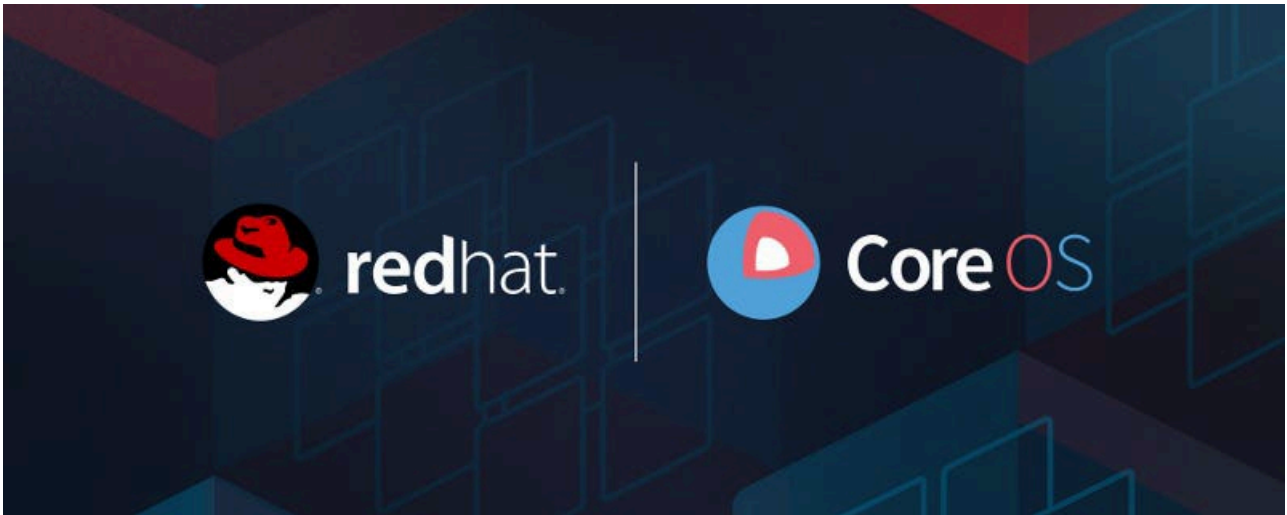


그림 3 레드햇 코어

2014 년도에는 구글은 쿠버네티스 프로젝트를 깃헙을 통해서 공개 및 릴리즈 하였다. 대다수 오픈소스 기업 구글, 레드햇/수세/IBM 들은 쿠버네티스 기반으로 서비스 플랫폼을 제작 및 판매를 하고 있다. 또한, 파편화를 방지하기 위해서 OCI(Open Container Initiative), CNI(Container Network Interface), CSI(Container Storage Interface)와 같은 표준안들이 제시가 되고 있으며, 현재는 수많은 기업들이 이 표준안 기반으로 쿠버네티스 서비스를 기여 및 개발을 하고 있다.

## 2. 쿠버네티스 구성 요소

쿠버네티스는 다음과 같은 구성 요소를 가지고 있다.

1. POD
2. Static POD
3. kubelet
4. kubectl
5. kubeadm

POD 는 쿠버네티스에서 기본이 되는 개념적인 자원이자 프로그램 이기도 하다. POD 의 다른 호칭은 인프라 컨테이너(Infra Container)라고 부르기도 하며, 일반적인 컨테이너와 동일하게 런타임(runtime)에서 동작하지만, 기능적으로 다르게 동작한다. 쿠버네티스는 기본적으로 Pause 라는 애플리케이션을 사용하며, 이를 통해서 POD 기능을 구현을 한다. 다만, 현재 사용하는 POD 는 회사별로 다른 POD 프로그램을 사용할 수 있다.

POD 는 또한, 두 가지로 나누어지는데 쿠버네티스에서 사용하는 정적 POD(Static Pod)가 있으며, 일반 사용자가 사용하는 일반 POD(Generic POD)가 있다. 정적 POD 는 쿠버네티스가 실행 시 사용하는 애플리케이션 컨테이너이다. 서비스가 실행이 되는 POD 는 일반 애플리케이션이다.

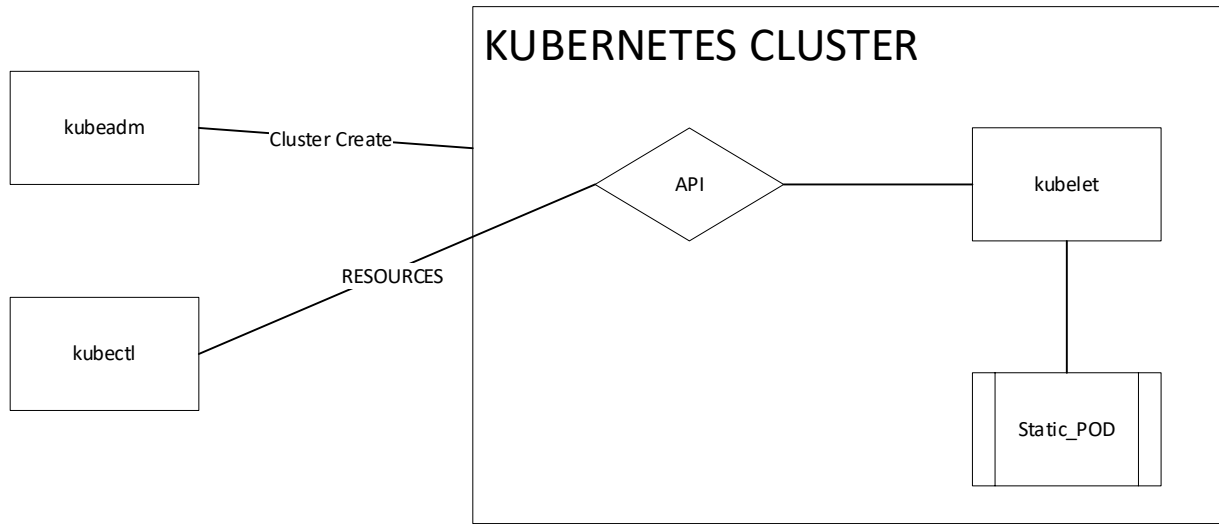


그림 4 쿠버네티스 구성 요소

kubelet 은 프록시(proxy)와 같은 서비스이다. 외부에서 들어온 API 들은 kubelet 를 통해서 정적 POD 에 전달이 된다. 또한, kubelet 은 컨테이너 생성 및 라이프사이클에도 관여를 한다.

kubeadm 은 쿠버네티스 클러스터 생성 시 사용하는 부트스트랩(bootstrap)명령어이다. 이 명령어를 통해서 쿠버네티스 클러스터 생성이 가능하다. 이 명령어를 사용하기 위해서는 앞서 이야기한 kubelet, kubectl 와 같은 도구가 올바르게 설치 및 구성이 되어 있어야 한다.

### 3. 쿠버네티스 표준 사양

쿠버네티스 표준 사양은 총 4 가지로 나누어져 있다. 아래는 대표적인 쿠버네티스 사양에 대한 설명이다.

#### OCI (Open Container Initiative)

OCI 는 리눅스 파운데이션에서 관리하고 있으며, 이 프로젝트는 Docker 를 통해서 2015 년도에 시작하였다. 이 프로젝트에서는 다음과 같은 부분에 대해서 명시를 하고 있다.

1. 컨테이너 런타임 사양(Runtime Spec)
2. 컨테이너 이미지 사양(Image Spec)
3. 컨테이너 배포(Distribution Spec)
4. 파일시스템(Filesystem Bundle)

이를 통해서 모든 컨테이너 런타임을 표준 런타임 사양(명령어 포함) 및 이미지를 사용하기 때문에, 다른 회사의 제품을 사용하여도 이미지 호환이 가능하다. 다만, 도커 경우에는 Docker OCI 및 비 OCI 로 버전에 따라서 나누어지기 때문에 도커 기반으로 사용 시, 이러한 부분을 주의해야 한다. 오픈소스에서 많이 사용하는 Podman, CRI-O 는 OCI 를 따른다.



## CNI (Container Network Interface)

컨테이너에서 사용하는 네트워크 인터페이스를 정의 및 명시한다. 사용하는 컨테이너 런타임에 따라서 다르기는 하지만, 대다수 컨테이너 런타임은 CNI 형식으로 플러그인 네트워크를 제공한다. 보통 디렉터리는 `/etc/cni`이며, 여기에 사용하는 플러그인 설정 파일들이 존재한다.

일반적으로 컨테이너 런타임은 리눅스 브릿지 기반으로 CNI 네트워크 인터페이스를 제공한다.

## CSI (Container Storage Interface)

컨테이너 런타임 및 포드 및 컨테이너가 사용하는 스토리지 인터페이스에 대해서 설정 및 구성한다. 일반 저수준 고수준 컨테이너 런타임 및 쿠버네티스는 CSI 기반으로 스토리지를 확장 및 구성한다. CSI 는 COs(Container Orchestration Systems)기반의 모든 플랫폼에서 적용 및 사용이 가능하다.

CSI 는 런타임 및 영역별로 다르지만, 컨테이너가 지원 및 사용하는 CSI 드라이버는 기본값은 overlay2 로 되어 있다. 이외 PV/PVC 에서도 이를 통해서 확장 저장소를 구성한다. CSI 를 사용하는 경우 손쉽게 스토리지 확장이 가능하다.

## CRI (Container Runtime Interface)

컨테이너 런타임 인터페이스의 표준화를 위한 사양이다. 쿠버네티스는 CRI 를 지원하지 않는 컨테이너 런타임은 kubelet 를 통해서 사용이 불가능하다. 이러한 이유로 도커는 CRI 인터페이스를 별도로 제공하며(기본값은 moby), containerd 런타임도 CRI 인터페이스 어댑터를 제공한다.

이러한 이유로, 쿠버네티스는 가급적이면 CRI 인터페이스를 직접적으로 제공하는 cri-docker(Mirantis Docker)혹은 CRI-O 사용을 권장한다.

## 4. 인프라 코드화

쿠버네티스 플랫폼은 코드 기반으로 구성한다. 먼저, 쿠버네티스 자체가 YAML 기반으로 인프라를 작성 및 구성하며, 이를 API 서버에 JSON 형태로 전달한다. 변환이 된 정보는 API 서버에서 각 쿠버네티스 컴포넌트에 전달하여 자원에 대해서 작업을 수행한다.

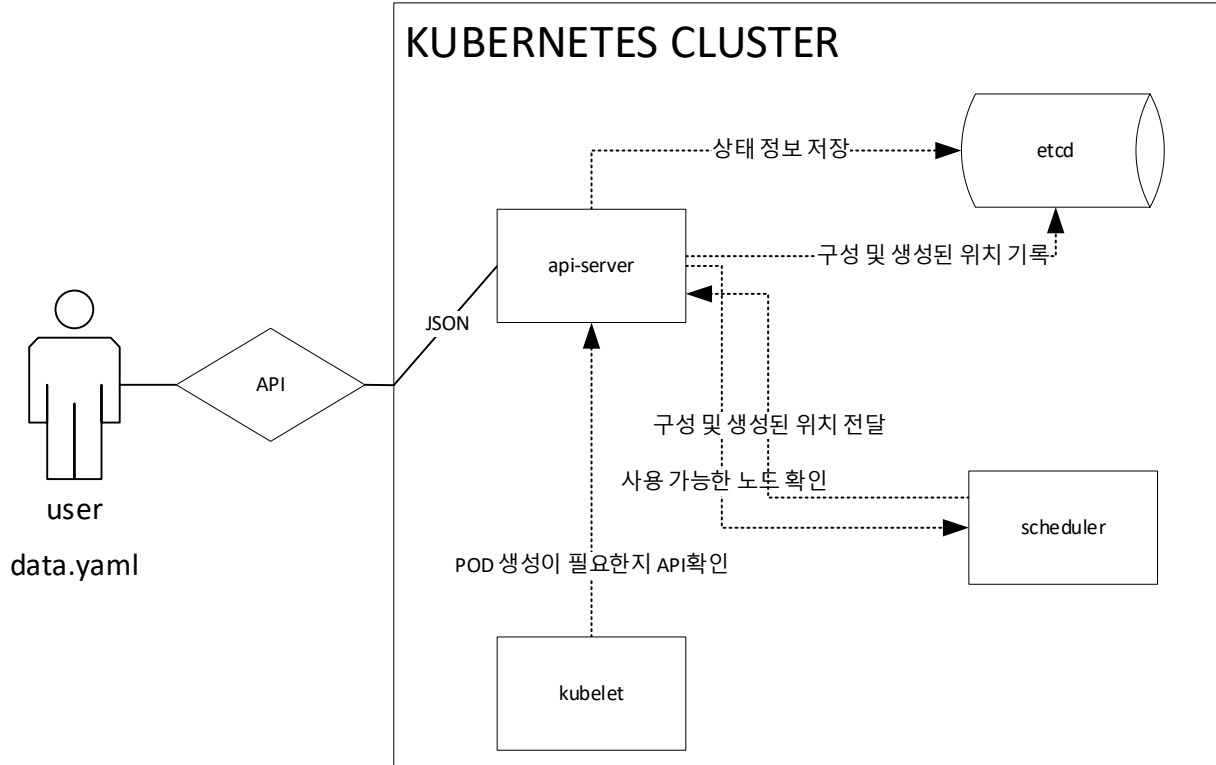


그림 5 인프라 코드화

두 번째는 쿠버네티스의 API 기반으로 인프라를 자동화한다. 이는 `kubectl` 명령어가 아닌, 간접적인 도구를 통해서 쿠버네티스 클러스터를 관리한다. 아래는 대표적인 자동화 도구이며, 이를 통해서 쿠버네티스 이외 다른 자원들과 함께 자동화가 가능하다.

1. ansible
2. terraform
3. puppet
4. Open Tofu

위의 4 가지 도구가 대표적인 자동화 도구이다. 위의 4 번은 Terraform 의 오픈소스 버전 생각하면 된다. 그러기 때문에, 4 번을 제외한다면, 일반적으로 3 개의 도구를 통해서 인프라 자동화를 구성한다. 이 책에서는 이 부분에 대해서 간단하게 다룬다.

## 5. 클라우드 네이티브

클라우드 네이티브는

기본적인 쿠버네티스는 클라우드 네이티브 기능을 제공하지 않는다. 이를 사용하기 위해서는 확장 기능으로 설치해야 되며, 클라우드 네이티브의 목적은 serverless 형태로 서비스를 배포 및 구성한다.

클라우드 네이티브는 이벤트 기반으로 서비스를 배포하기 때문에, 서비스 중심으로 디자인 및 운영이 가능하다. 쿠버네티스는 이를 Knative 라는 프로젝트를 진행하고 있으며, 이를 적절하게 사용하기 위해서는 여러가지 기능들이 요구가 된다.

앞서 이야기하였지만, knative 는 기본적으로 제공하는 기능이 아니기 때문에, 뒤 부분에서 간단하게 설치 후 다루도록 한다.

## 설치 환경

### 1. 설치 권장 사양

쿠버네티스 랩을 진행하기 위해서는 다음과 같은 **컴퓨터 사양(호스트)**을 요구한다.

표 1 사양

사양	크기
CPU	Intel i5 8 Cores 이상
Memory	32GiB 이상 권장, 최소 16 GiB
Disk	SSD 500 GiB 이상 권장, 최소 HDD 500 GiB
Network	외부에 접근이 가능한 네트워크 환경
Virtualization	VT-X/VT-D 혹은 AMD-V 기능이 사용이 가능한 환경

자체적으로 사용하는 가상머신 환경이나 혹은 프로그램이 있으면 해당 프로그램을 사용해서 가상머신들을 구성하면 된다.

표 2 운영체제

OS	지원여부	설명
Linux	지원	libvirtd 사용이 가능한 모든 배포판 지원. CentOS 혹은 Fedora Core 권장 Centos 7/8 버전 둘 다 지원. 다만, 파이썬 버전은 python 2.7 이상을 사용하는 것을 권장. 만약 CentOS 8 버전 이상 사용하는 경우 Python 3.x 가 설치가 되어 있는 경우에는 'dnf'명령어를 사용해서 설치 진행 권장.
Windows	부분지원	지원하지 않음. 직접 수동 설치
OS X	미지원	지원하지 않음

윈도우 혹은 맥에서는 **VirtualBox** 혹은 **VMware Player** 혹은 **VMware Workstation** 를 사용한다. 다만 VirtualBox 는 vCPU 부분에서 문제가 종종 있기 때문에 컨테이너 구성 시 올바르게 동작이 안될 가능성이 높다.

CPU 아키텍처 별로 안전성이 매우 다르기 때문에 윈도우 프로 10/11 경우에는 Hyper-V 같은 도구 사용이 아니면 가급적이면 VMware Player 사용을 권장한다.

## 2. 기본 랩 구성

랩은 내부 및 외부 네트워크 두개로 구성이 되며, 외부 아이피는 NAT 를 통해서 외부와 통신이 되며, 내부는 마스터 및 각 노드끼리 API 및 터널링을 위한 용도로 사용한다. 클러스터는 단일 혹은 다중 클러스터 구성이 가능하다. 기본적으로 단일 클러스터 사용을 권장하며, 좀 더 다양하게 연습 및 활용을 원하면 다중 클러스터로 구성한다.

단일 클러스터는 다음과 같이 가상머신 사양을 권장한다.

표 3 단일 노드 구성 및 정보

서버	사양	설명
master1.example.com	CPU: 4, MEMORY: 4096MiB+, DISK: 30G(ROOT)	단일 마스터 노드용도.
node1.example.com	CPU: 4, MEMORY: 4096MiB+, DISK: 30G(ROOT)	서비스 및 인프라 컴퓨트 노드
node2.example.com	CPU: 4, MEMORY: 4096MiB+, DISK: 30G(ROOT)	서비스 및 인프라 컴퓨트 노드

다중 클러스터는 다음과 같은 가상컴퓨터 사양을 권장한다.

표 4 다중 노드 구성 및 정보

서버	사양	설명
lb.example.com	CPU: 4, MEMORY: 4096MiB+, DISK: 30G(ROOT)	HAProxy 기반 로드 밸런서 서버
master1.example.com	CPU: 4, MEMORY: 4096MiB+, DISK: 30G(ROOT)	다중 마스터 노드.
master2.example.com	CPU: 4, MEMORY: 4096MiB+, DISK: 30G(ROOT)	다중 마스터 노드
master3.example.com	CPU: 4, MEMORY: 4096MiB+, DISK: 30G(ROOT)	다중 마스터 노드
node1.example.com	CPU: 4, MEMORY: 4096MiB+, DISK: 30G(ROOT)	서비스 및 인프라 컴퓨트 노드
node2.example.com	CPU: 4, MEMORY: 4096MiB+, DISK: 30G(ROOT)	서비스 및 인프라 컴퓨트 노드

아래 그림은 단일 노드 기반으로 구성한 클러스터 예상 다이어그램이다.

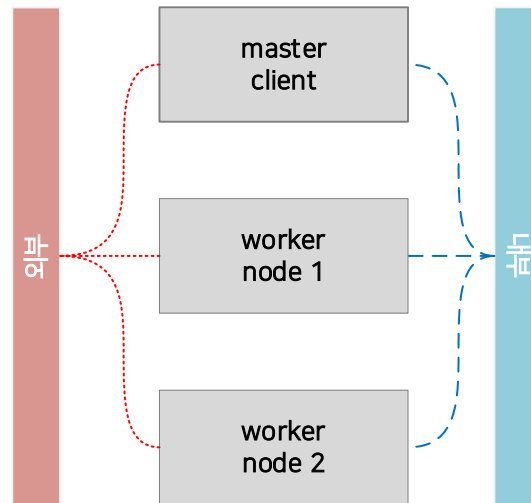


그림 6 랩 구성도

참고로, 위의 구성에서는 부트스트랩 및 유틸리티 노드는 별도로 명시하지 않았으며, 마스터 서버가 유틸리티 서버 역할도 한다. 여기서는 스토리지를 위한 NFS 기능을 마스터 서버가 가져간다.

### 3. 하이퍼바이저 소프트웨어

윈도우 혹은 맥에서 설치하는 경우 다음과 같은 하이퍼바이저 사용을 권장한다.

1. VMware Player 혹은 VMware Workstation
2. VirtualBox
3. Hyper-V

앞에서 말했지만 VirtualBox 는 종종 아키텍처 혹은 버전별로 vCPU 문제가 있기 때문에 권장하지 않는다. 윈도우 사용자는 가급적이면 하이퍼브이 기반으로 랩 구성을 권장한다. 만약, VMware Player, Workstation 라이선스를 가지고 있는 경우, 이 제품들을 사용하여도 무난한다. 네트워크는 외부 및 내부 네트워크 두 개로 구성되어야 한다. 그 이외 조건에서는 랩과 부합하지 않는다.

### 4. 에디터 설정

#### VI/VIM

YAML 파일 작성시 좀 더 편하게 작성하기 위해서 vi 혹은 vim 사용자는 아래처럼 '.vimrc'파일을 생성한다. root 계정에서는 기본적으로 vi 만 지원하기 때문에 레드햇 사용자 경우에는 반드시 vim 설치를 해주고 설정을 한다. 보통 설정하면 아래와 같이 vim 설정을 한다.

```

master]# cat <<EOF> /$(($USER)/.vimrc
au! BufNewFile,BufReadPost *.u{yaml,yml} set filetype=yaml
foldmethod=indent
set autoindent expandtab tabstop=2 shiftwidth=2

```

```
EOF
```

## NANO

nano 에디터를 사용하는 경우, 다음과 같이 '.nanorc' 파일에 다음과 같이 설정한다. 만약 nano 를 사용하지 않는 경우 아래 설정을 사용하지 않아도 된다.

```
master]# cat <<EOF> /$(($USER)/.nanorc
syntax "YAML" "\.ya?ml$"
header "^(---|===)" "%YAML"
set tabsize 2
set tabstospaces
EOF
```

## ALE for VIM

VIM 이나 NeoVIM 를 사용하는 경우에는 "ALE"를 확장하여 편하고 쉽게 YAML 편집을 할 수 있도록 도와준다. 아래 주소에 방문하면 "ALE"설정에 대해서 사이트에 설명이 나와있다. ALE 에서 지원해주는 YAML Lint 를 사용하면 좀 더 편하게 CR/CRD 파일을 작성할 수 있다. 내려받기 주소는 여기에서 확인 및 내려받기가 가능하다.

<https://github.com/dense-analysis/ale>

실행 및 설치 방법은 아래와 같다. 사용 전, "vim"패키지가 설치가 되어 있어야 한다.

```
master]# mkdir -p ~/.vim/pack/plugins/start/
master]# dnf install git -y
master]# git clone --depth=1 https://github.com/dense-analysis/ale.git
~/.vim/pack/plugins/start/ale
```

자세한 내용은 ALE 사이트 가이드가 나와있다. 만약, 설치 방법이 어렵다고 판단이 되면, 아래 방법으로 손쉽게 설치가 가능하다. 윈도우/맥/리눅스에서 아래와 같이 사용이 가능하다.

```
> curl.exe https://webi.ms/vim-ale | powershell
# curl -sS https://webi.sh/vim-ale | sh
```

## 5. 사용을 위한 확장도구

### tmux

랩에서 생각보다 많은 터미널이 필요하다. 모니터가 작은 경우에는 tmux 나 혹은 screen 도구를 사용해서 좀 더 효율적으로 모니터를 사용한다. 아래는 tmux 설치 및 설정이다.

```
root# dnf install tmux -y
```

```
root# apt install tmux -f
```

설치 후, 좀 더 편하게 콘솔 이동을 하기 위해서 아래와 같이 tmux 설정 파일을 구성한다. 좀 더 손쉽게 마우스로 사용이 가능하다. 다만, 클립보드에 있는 내용을 붙여넣기 위해서는 시프트키를 누르고 오른쪽 마우스 버튼을 클릭하면 된다.

```
set-option -g mouse on
```

```
bind -n WheelUpPane if-shell -F -t = "#{mouse_any_flag}" "send-keys -M"  
"if -Ft= '#{pane_in_mode}' 'send-keys -M' 'select-pane -t=; copy-mode -e;  
send-keys -M'"
```

```
bind -n WheelDownPane select-pane -t= \; send-keys -M
```

# 쿠버네티스 설치 준비 및 설명

## 1. 앤서블 기반 설치 도구

쿠버네티스에서 제공하는 kubeadm 도구를 통해서 간단하게 마스터 및 워커 노드를 제공하였다. 이러한 방식은 작은 개수의 워커 노드 운영에는 문제가 없었지만, 수십 혹은 수백 개 노드를 구성시에는 적절하지 않는다.

2014 년 이전에는 RPM 혹은 DEB 기반으로 쿠버네티스 구성 요소 설치 및 구성을 하였지만, kubelet 를 제외한 모든 기능 구성원들이 런타임 밑으로 동작하면, 더 이상 스크립트 기반으로 설치하는 적절하지 않다. 이러한 이유로 쿠버네티스는 현재 앤서블 기반으로 설치 도구를 제공하고 있으며, 대표적인 도구는 kubespray 이다. 아래 이미지는 쿠버네티스에서 제공하는 설치 도구이다. 퍼블릭 클라우드 시스템은 각각 쿠버네티스 프로비저닝 시스템을 제공하기 때문에 별도로 설치 및 확장에 대해서 고민이 필요 없다.

다만, 온프레미스 기반으로 설치하는 경우 클러스터 구성에 대한 고민 사항이 많다. 이러한 이유로 개인 사용자가 간단하게 사용이 가능한 쿠버네티스 클러스터 도구가 많이 제공이 되고 있다.

### Deploy a Production Ready Kubernetes Cluster



kubernetes

If you have questions, check the documentation at [kubespray.io](https://kubespray.io) and join us on the [kubernetes slack](#), channel [#kubespray](#). You can get your invite [here](#)

- Can be deployed on [AWS](#), [GCE](#), [Azure](#), [OpenStack](#), [vSphere](#), [Equinix Metal](#) (bare metal), [Oracle Cloud Infrastructure \(Experimental\)](#), or [Baremetal](#)
- [Highly available](#) cluster
- [Composable](#) (Choice of the network plugin for instance)
- Supports most popular [Linux distributions](#)
- [Continuous integration tests](#)

그림 7 쿠버네티스 지원하는 플랫폼

## kubespray

앤서블 기반으로 만들어진 쿠버네티스 설치 도구이다. Kubespray 는 인큐베이터 프로젝트로 있었다가, 지금은 정식적인 설치 도구가 되었다. 다만, kubespray 는 순수 앤서블 기반으로 만들어진 도구이기 때문에 어느정도 지식이 앤서블 지식이 없는 사용자에게는 사용이 어려울 수 있다.

실무에서 kubespray 기반으로 많이 사용하며, 바닐라 버전보다는 본인들의 환경에 맞게 수정하여 많이 사용한다. 이 책에서는 kubespray 에 대해서는 다루지 않는다.

## duststack-k8s-auto

저자가 만든 duststack-k8s-auto 를 사용해서 간단하고 빠르게 쿠버네티스 설치를 진행한다. 만약, 자동설치를 원하지 않는 경우, 수동 설치로 진행한다. 이 책에서는 kubeadm, duststack-k8s-auto 를 통해서 설치를 지원하고 있다.



duststack-k8s-auto 는 다음과 구현 목적을 가지고 있다.

- kubespray 와 같은 자동화 같은 플레이북을 고객 혹은 회사 전용으로 만들어야 하는 경우
- 앤서블 기반으로 설치 자동화 및 프로젝트 설계 및 구성이 필요한 엔지니어
- roles 기반으로 확장기능 구성이 필요한 경우

만약, 수동 설치에 관심이 없는 경우, duststack-k8s-auto 를 통해서 가상머신 및 쿠버네티스 클러스터를 자동으로 설치 구성하여도 된다. 설치가 어려운 사용자는 이 도구 기반으로 설치를 권장하며, 이 도구 기반으로 설치 방법은 별첨으로 제공한다.

리눅스 워크스테이션을 사용하는 경우, 가상머신 생성 과정까지 libvirt 기반으로 제공한다.

## 2. 간단한 클러스터 생성 도구

이 책의 랩에서는 아래 클러스터 생성 도구 기반으로 진행하지 않는다. 아래 도구들로 진행하는 경우 본인이 스스로 적절하게 변경 후 사용하면 된다.

### kind

kind 는 Kubernetes 를 도커 기반으로 간단하게 클러스터를 생성 후 학습용으로 사용이 가능하다. kind 는 OCM(Open Cluster Manager)테스트 용도로도 사용이 가능하다. 이 과정에서는 설치부분에 대한 학습이 포함되어 있기 때문에, kind 는 사용하지 않는다.

설치를 원하는 경우, 아래 사이트에서 확인 및 설치가 가능하다.

- <https://kind.sigs.k8s.io/>

```
$ time kind create cluster
Creating cluster "kind" ...
  ✓ Ensuring node image (kindest/node:v1.16.3) 📜
  ✓ Preparing nodes 📦
  ✓ Writing configuration 📄
  ✓ Starting control-plane 🚦
  ✓ Installing CNI 🛠️
  ✓ Installing StorageClass 💾
Set kubectrl context to "kind-kind"
You can now use your cluster with:

kubectrl cluster-info --context kind-kind

Not sure what to do next? 😊 Check out https://kind.sigs.k8s.io/docs/user/quick-start/

real    0m21.890s
user    0m1.278s
sys     0m0.790s
```

그림 8 KIND

위의 주소에서 kind 를 내려받기가 가능하며, 간단하게 설치 및 구축 방법은 다음과 같다. 맥 경우에는 다음과 같이 콘솔에서 실행한다.

```
mac]$ sudo brew install kind
```

혹은 "맥 포트"를 사용하는 경우에는 다음 명령어로 실행이 가능하다.

```
mac]$ sudo port selfupdate && sudo port install kind
```

윈도우 경우에는 초코(chocolatey)를 사용해서 설치가 가능하다. 초코(chocolatey) 설치는 다음 주소에서 가능하다.

- <https://community.chocolatey.org/packages/kind>

설치는 파워 셸에서 아래 명령어로 구성 및 실행이 가능하다.

```
> curl.exe -Lo kind-windows-amd64.exe
https://kind.sigs.k8s.io/dl/v0.17.0/kind-windows-amd64
> Move-Item .\kind-windows-amd64.exe c:\<DIRECTORY>\kind.exe
```

리눅스 사용자는 다음 명령어로 kind 설치가 가능하다.

```
desktop]$ curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.17.0/kind-linux-amd64
desktop]$ chmod +x ./kind
desktop]$ sudo mv ./kind /usr/local/bin/kind
```

## minikube

minikube 도 kind 하고 비슷하지만, 다양한 백-엔드 엔진(하이퍼바이저)을 통해서 클러스터 구현이 가능하다. 지원되는 목록은 <https://minikube.sigs.k8s.io/docs/start/> 주소에서 확인이 가능하다. 윈도우에서 사용하는 경우, 다음과 같은 순서로 진행한다.

먼저, 윈도우에 미니큐베 설치를 진행한다. 설치 시, 깃헙 바이너리 이미지를 가져와서 구성한다. 아래 명령어는 파워 셸에서 실행 및 수행한다. 아래 명령어를 수행하면 미니큐베를 여러분의 랩톱에 설치를 시작한다.

```
> winget install minikube
```

파워 셸을 관리자 모드로 실행 후, 아래 명령어를 실행한다. 윈도우 \$env:path 확인 후, 미니큐베 경로를 추가한다. 고정으로 추가가 필요 없는 경우, 진행하지 않아도 된다.

```
powershell> $oldPath = [Environment]::GetEnvironmentVariable('Path',
[EnvironmentVariableTarget]::Machine)
if ($oldPath.Split(';') -notcontains 'C:\minikube'){ `
    [Environment]::SetEnvironmentVariable('Path', $('{0};C:\minikube' -f
$oldPath), [EnvironmentVariableTarget]::Machine) `
}
```

클러스터 시작은 아래 명령어로 시작한다.

```
powershell> minikube start
```

파워 셸에서 명령어를 쉽게 실행하기 위해서 아래와 같이 함수를 선언한다.

```
> function kubect1 { minikube kubect1 -- $args }
```

올바르게 실행이 되는지 아래와 같은 명령어로 확인한다.

```
> kubect1 get po -A
> minikube kubect1 -- get po -A
```

### 3. 부트 스트랩 명령어 설명

쿠버네티스 설치 시 kubeadm 하위 명령어를 통해서 클러스터 구성 및 설치가 가능하다. 설치를 진행하기전에 간단하게 kubeadm 명령어의 기능에 대해서 확인한다. kubeadm 은 두 가지 방식으로 설치를 진행한다.

1. YAML 설정 기반 설치
2. 명령어 기반 설치

이 책에서는 **"명령어 기반 설치"**로 진행할 예정이다. 만약 수동 설치가 어려운 경우에는, 별첨의 **자동 설치 도구**를 참고하도록 한다. 쿠버네티스 v1.26 부터는 저장소가 버전별로 제공이 된다. 이러한 이유로, 이전 설치 방식인 **패키지-버전**형태로 설치를 진행하지 않는다. 설치 전 부트스트랩 명령어 옵션에 대해서 간단하게 살펴본다. 다만, 버전별로 조금씩 변경이 된 내용이 있을 수도 있다.

#### init

쿠버네티스 컨트롤 플레인을 구성한다. 'init' 하위 명령어는 보통 마스터 노드 구성을 위해 사용한다. 이 명령어는 워커 노드 구성에는 사용하지 않으며 역시 멀티 마스터 노드를 구성시에도 init 는 첫번째 마스터 노드에만 사용한다.

init 작업이 수행이 된 이후에는 자동적으로 '.kube'디렉터리를 자동으로 생성하지 않는다. 영구적으로 사용자 인증서를 사용하기 위해서는 아래와 같이 명령어를 수행한다.

```
master]# kubeadm init
master]# kubectl get pods
master]# mkdir -p ~/.kube
master]# cp -i /etc/kubernetes/admin.conf ~/.kube/config
master]# chown root. ~/.kube/config
master]# export KUBECONFIG=/etc/kubernetes/admin.conf
master]# kubectl get pods
```

특정 버전을 선택해서 설치하고 싶은 경우, 저장소에서 특정 버전으로 릴리즈가 되어 있는 패키지 기반으로 설치한다. 예를 들어서 다음과 같이 패키지를 설치한다. 업그레이드 진행 시, 이와 비슷한 방식으로 진행한다.

```
master/node]# yum install kubeadm
```

init 명령어에서 특정 옵션을 사용하면, 별도의 작업 없이 다중 마스터 서버 구성이 가능하다. 이 부분에 대해서는 아래 join 에서 다루도록 하겠다.

## 가입 및 초기화 설정파일

쿠버네티스 설치시, kubeadm 명령어에 클러스터 설정파일 기반으로 설정이 가능하다. 이 과정에서는 설정 파일 기반으로 설치하는 자세하게 다루지 않으며, 간단하게 설명만 한다. 아래 내용은 설치할 쿠버네티스의 클러스터 이름 및 인증서 설정이다.

```
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
bootstrapTokens:
  - token: 9a08jv.c0izixklcxtmnze7
    description: "kubeadm bootstrap token"
    ttl: "24h"
  - token: 783bde.3f89s0fje9f38fhf
    description: "another bootstrap token"
    usages:
      - authentication
      - signing
    groups:
```

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

localAPIEndpoint:

advertiseAddress: **10.10.10.10**

bindPort: **6443**

certificateKey:

e6a2eb8581237ab72a4f494f30285ec12a9694d750b9785706a83bfcbbbd2204

---

apiVersion: kubeadm.k8s.io/v1beta3

kind: ClusterConfiguration

kubernetesVersion: v1.28.0

apiServer:

certSANs:

- 127.0.0.1

- 10.10.10.10

- 10.10.10.250

certificatesDir: /etc/kubernetes/pki

clusterName: **dustbox.kr**

controlPlaneEndpoint: **10.10.10.10:6443**

controllerManager:

extraArgs:

etcd:

local:

dataDir: /var/lib/etcd

imageRepository: **registry.k8s.io**

networking:

dnsDomain: **cluster.local**

podSubnet: **10.88.0.0/16**

serviceSubnet: **10.96.0.0/12**

scheduler:

extraArgs:

---

apiVersion: kubelet.config.k8s.io/v1beta1

kind: KubeletConfiguration

---

```
apiVersion: kubeproxy.config.k8s.io/v1alpha1
kind: KubeProxyConfiguration
```

kubeadm 명령어로 설치를 진행하면, 몇몇 영역은 사용자화가 불가능한 부분이 있기 때문에, 사용자가 직접 특정한 부분에 대해서 사용자화를 원하는 경우 설정 파일 형식으로 진행해야 한다. 쿠버네티스 기반으로 구성이 되어 있는 랜처 및 오픈 시프트들은 설정파일 기반으로 클러스터 및 기능을 구성한다.

## join

이 하위 명령어는 마스터 노드에 워커를 추가하는데 사용한다. 다만 주의해야 할 부분은, 현재 쿠버네티스는 스왑이 활성화가 되어 있어도 설치가 진행이 되지만, kubelet 이나 혹은 네트워크 서비스가 올바르게 동작하지 않을 수 있다. 이러한 이유로 join 이 올바르게 되지 않는 경우 firewalld, SELinux 그리고 swap 이 동작 중 인지 확인이 필요하다. 노드를 클러스터에 가입시키기 위해서는 반드시 bootstrap 에서 생성한 토큰 및 SHA 를 사용해야 한다.

```
node]# systemctl stop firewalld
node]# swapoff -a
node]# kubeadm join 192.168.68.122:6443 --token gcd426.2itdtcs7o1p7ds6r -
--discovery-token-ca-cert-hash \
sha256:bfad2126496a0779ccbc2cf9b841834cc930384eb0158cce40332f74a7b544d7
master#] kubectl get nodes
```

만약 멀티 마스터로 클러스터를 구성하는 경우 다음과 같은 명령어로 노드를 2,3...n 노드로 추가를 해주어야 한다. 아래는 멀티 마스터 노드를 구성 및 추가하는 명령어 예시이다.

```
master]# kubeadm init
master]# kubeadm init phase upload-certs --upload-certs
master]# kubeadm token create --print-join-command
kubeadm join 10.10.10.69:6443 --token s1zh15.hzn7bsg8naeu7kxo \
--discovery-token-ca-cert-hash \
sha256:2a0b134aee6854f3be2876b837dbaf54f5c72ff1c8c284af8f49 \
a657ab47886d --control-plane --certificate-key \
a5d2c74b84159256969255258edee23e6b018619d4b55dd07ec694510927ac93
```

위와 같은 명령어로 멀티 마스터로 사용할 노드에서 명령어를 실행해서 메인 마스터 노드에 가입한다. 올바르게 가입이 되었는지 확인하기 위해서는 kubectl get nodes 와 같은 명령어로 확인이 가능하다.

## 가입 설정파일

join 부분도 init 와 동일하게 설정파일 기반으로 추가가 가능하다. 동일한 부분에 같은 내용이 있기 때문에 별도로 링크를 첨부하지 않는다.

## upgrade

쿠버네티스 클러스터 버전을 업그레이드 지원해주는 하위 명령어. upgrade 를 통해서 마스터 노드 및 워커 노드에 대해서 업데이트를 진행한다.

```
master]# kubeadm upgrade node
master]# kubeadm upgrade plan
```

해당 부분을 연습하고 싶은 경우, 반드시 쿠버네티스 버전을 이전버전으로 선택 후 진행해야 한다.

## config

init, join 으로 구성된 마스터 혹은 워커 노드의 정보를 YAML 파일로 저장하거나 혹은 이미지 목록 및 내려받기가 가능하다. 또한 kubeadm config migrate 를 통해서 이전에 사용하던 오래된 설정파일을 새로운 설정 파일로 변경이 가능하다.

아래 명령어는 이미 구성이 되어 있는 내용을 텍스트 파일로 출력한다.

```
master]# kubeadm config print --init-defaults
```

출력이 되는 내용은 JSON 형태로 화면에 표시가 된다. 사용하는 이미지 목록을 확인하기 위해서 역시 config 를 통해서 확인이 가능하다.

```
master]# kubeadm config images list
```

## token

kubeadm join 명령어를 통해서 클러스터에 가입하기 위한 토큰생성. 토큰 생성을 하지 않으며, 올바르게 클러스터에 가입을 할 수 없다.

기본적으로 초기 init 가 수행이 되면 보통 화면에 사용이 가능한 토큰 명령어가 출력이 된다. 하지만 명령어 혹은 토큰 이름을 다시 기억하기가 어려우면(진짜 어렵다) 아래 명령어로 다시 토큰 및 가입 명령어 확인이 가능하다.

```
master]# kubeadm token create --print-join-command
```

유효기간이 지나지 않았으면 생성된 토큰 목록은 아래 명령어로 확인이 가능하다.

```
master]# kubeadm token create
master]# kubeadm token list
```

## reset

현재 구성이 되어 있는 쿠버네티스 클러스터를 초기화 한다. 초기화 시, 기존에 구성이 되어 있는 시스템의 모든 서비스는 초기화 된다. 초기화 되는 구성은 보통 다음과 같이 된다.

- 노드에 구성이 되어 있는 kubelet.service
- 노드에 구성이 되어 있는 컨테이너 서비스
- /etc/kubernetes 설정 파일들

초기화 작업은 각각 노드에서 수행 및 실행해야 한다.

```
master/node]# kubectl reset  
master/node]# kubectl reset --force
```

## version

쿠버네티스 버전을 출력한다. 출력되는 버전은 설치된 쿠버네티스 버전이 아니라 kubeadm 에서 제공하는 이미지 버전이다.

```
master]# kubeadm version
```

## alpha

새로운 쿠버네티스 기술을 사용시 사용하는 옵션. 실 제품에서 사용은 권장하지 않으며 필요한 기능이나 혹은 실험적으로 적용하는 경우, alpha 를 통해서 특정 기능 활성화가 가능. 일반적으로 알파 기능은 사용하지 않는다. 이 책에서는 이 기능에 대해서 다루지 않는다.

## 4. 윈도우 가상머신

윈도우 10 혹은 11 프로버전을 사용하는 경우, "Hyper-V(하이퍼-브이)"기반으로 구성을 권장한다. 이 책은 윈도우 사용자에게 하이퍼-브이 기반으로 진행을 권장한다. 만약, 버추얼 박스나 VMWare Workstation 를 사용하는 경우, VCPU 의 버그로 인하여 가끔 올바르게 컴퓨터 노드가 올바르게 동작하지 않는 경우가 있다.

하이퍼-브이에서 가상머신을 구성하기 위해서 아래 명령어를 진행한다.

### Hyper-V 설치

먼저, 하이퍼-브이가 구성이 되어있지 않으면 아래 명령어로 하이퍼-브이를 구성한다. 아래 모든 명령어는 "윈도우 파워 쉘"에서 실행한다.

```
powershell> Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All
```

### 가상머신 생성

가상머신을 생성하기 위해서 아래 디렉터리 두 개를 생성한다.

```
powershell> mkdir \VMs
```



```
powershell> mkdir \VMdata
```

생성이 완료가 되면 아래 명령어로 "마스터" 및 "워커 노드 1,2 번"을 생성한다. 쿠버네티스를 사용하기 위해서는 최소 vCPU 2 개, vMEM 은 4GiB 가 구성이 되어야 한다. 최소 크기로 운영하기 위해서 디스크 크기는 12 기가로 하였으며, uEFI 기반으로 리눅스 시스템을 구성한다.

```
powershell> New-VM -Name master -MemoryStartupBytes 4GB -BootDevice VHD -  
NewVHDPath .\VMs\master.vhdx -Path .\VMData -NewVHDSizesBytes 12GB -  
Generation 2 -Switch "Default Switch"
```

가상머신 생성 후, 설치 시 필요한 ISO 파일을 가상머신에 DVD 드라이브를 추가 후 연결한다.

```
powershell> Add-VMdvdDrive -VMName master -Path rockylinux.iso
```

가상머신 시작 전, 가상머신에서 사용하는 vCore 를 2 개로 변경한다. 앞서 말했지만, 이는 가동에 필요한 최소 코어 개수이며, 넉넉하면 4 개 이상을 권장한다. 메모리는 최소 1.8 기가 혹은 그 이상을 권장한다.

```
powershell> SET-VMProcessor master -count 4  
powershell> SET-VM -name master -MemoryStartupBytes 2GB
```

## Hyper-V 내부 네트워크 구성

가상머신이 준비가 되면, 각각 머신에 API 통신시 사용할 내부 네트워크를 생성 후 각각 가상머신에 할당한다. 아래 명령어로 "Internal"네트워크를 생성한다. 단, 실행하기전 아래 내용을 좀 더 읽어보고 어떤 방식을 사용할지 결정한다.

```
powershell> New-VMSwitch -name InternalSwitch -SwitchType Internal
```

생성이 완료가 되면 아래 명령어로 가상머신에 "Internal"스위치를 연결한다.

```
powershell> ADD-VMNetworkAdapter -VMName master -Switchname Internal
```

참고로 "master"부분에 "node1", "node2"으로 변경해서 각각 vCore 및 internal switch 를 연결한다. 외부 아이피(NAT)를 고정적으로 사용을 원하는 경우 아래와 같이 네트워크를 생성한다.

```
powershell> New-VMSwitch -SwitchName "StaticNATSwitch" -SwitchType  
Internal  
powershell> Get-NetAdapter  
powershell> New-NetIPAddress -IPAddress 192.168.0.1 -PrefixLength 24 -  
InterfaceIndex <INDEX>
```

## 아래처럼 명령어를 입력한다.

```
powershell> New-NetNat -Name StaticNATSwitch -  
InternalIPInterfaceAddressPrefix 192.168.0.0/24
```

## WSL2

WSL2 기반으로 랩 진행하는 경우, WSL2 에서 가상머신 접근이 되지 않는다. WSL2 기반으로 접근을 원하는 경우, 아래처럼 파워셸 관리자 권한에서 명령어를 실행한다.

```
powershell> Set-NetIPInterface -ifAlias "vEthernet (WSL)" -Forwarding  
Enabled  
powershell> Set-NetIPInterface -ifAlias "vEthernet (Default Switch)" -  
Forwarding Enabled
```

위의 명령어를 실행하면, WSL 환경에서 가상머신으로 접근이 가능하다. 만약, 사용하지 않으면 이 부분은 실행하지 않아도 된다.

## 5. 설치준비: 공통 설정

### 도메인 레코드 설정(/etc/hosts)

랩 진행 시, 가급적이면 DNS 서버 구축을 권장한다. 하지만, 권장이 불가능한 경우는 최소 /etc/hosts 파일에서 A 레코드에 대해서 설정을 한다. 앤서블 플레이 북으로 랩 구성한 경우, 자동으로 수정하지만, 수동으로 구성하는 경우, 아래와 같이 설정한다.

```
master/node]# cat <<EOF>> /etc/hosts  
192.168.10.250 master.example.com master  
192.168.90.120 node1.example.com node1  
192.168.90.130 node2.example.com node2  
EOF  
master]# dnf install sshpass -y
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
master]# for i in master node{1..2} ; do sshpass -pcentos scp /etc/hosts  
$i:/etc/hosts ; done
```

만약, hosts 기반으로 A 레코드 구성하기 싫으면, 부록 External DNS 서버를 확인하도록 한다.

## 모듈설정

쿠버네티스에서 사용할 커널 모듈을 활성화한다. 활성화할 모듈은 보통은 다음과 같다.

- overlay
- br\_netfilter

오버레이 모듈(overlay module)은 Overlayfs 를 구현 시 사용한다. br\_netfilter 는 bridge 관련된 netfilter 모듈이다. 컨테이너에서 사용하는 네트워크 인터페이스는 리눅스 브릿지나 혹은 OVS 브릿지를 통해서 생성 및 구성이 되기 때문에, bridge 에서 생성된 자원의 정보는 br\_netfilter(netfilter)에서 관리 및 추적한다.

이를 영구적으로 사용하기 위해서 다음처럼 /etc/modules-load.d/에 등록한다. 여기서는 파일명을 k8s-modules.conf 으로 한다. 원하면 파일 이름은 마음대로 변경하여도 된다.

```
master/node]# cat <<EOF> /etc/modules-load.d/k8s-modules.conf
br_netfilter
overlay
EOF
master/node]# modprobe br_netfilter
master/node]# modprobe overlay
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
master]# for i in master node{1..2} ; do sshpass -pcentos scp
/etc/modules-load.d/k8s-modules.conf $i:/etc/modules-load.d/k8s-
modules.conf ; done
```

커널 변수(kernel Parameter)는 다음처럼 설정한다. 아래는 리눅스 브리지에서 관리 및 forwarding 하기 위해서 활성화한다. 만약 ipv6 를 사용하지 않는 경우에는 비활성화 한다.

```
master/node]# cat <<EOF> /etc/sysctl.d/99-k8s.conf
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF
master/node]# sysctl --system
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
master]# for i in master node{1..2} ; do sshpass -pcentos scp
/etc/sysctl.d/99-k8s.conf $i:/etc/sysctl.d/99-k8s.conf && sysctl -q --
system ; done
```

## 6. 쿠버네티스 수동설치

위에서는 OS 영역에 기본적인 설정을 진행하였다. 최종적으로 쿠버네티스 저장소를 구성한 다음에, 쿠버네티스 명령어 및 부스트랩 도구를 설치한다.

### 쿠버네티스 저장소

쿠버네티스 설치를 위한 저장소를 구성한다. YUM 저장소는 일반적으로 다음과 같은 주소를 가지고 있다. 역시 배포판마다 다르기 때문에 쿠버네티스 웹 사이트에서 확인한다. 아래 명시한 버전은 현시점<sup>7</sup>으로 제일 오래된 버전으로 명시하고 있다. 해당 버전이 없는 경우, 이보다 상위 버전을 선택해서 진행한다.

```
master/node]# cat <<EOF> /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/v1.26/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/v1.26/rpm/repodata/repomd.xml.ke
y
exclude=kubelet kubeadm kubectl cri-tools kubernetes-cni
EOF
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
master]# for i in master node{1..2} ; do sshpass -pcentos scp \
/etc/yum.repos.d/kubernetes.repo \
$i:/etc/yum.repos.d/kubernetes.repo && sysctl -q --system ; done
```

쿠버네티스 설치를 위한 kubeadm 패키지 설치한다. 특정 버전을 원하는 경우, 패키지 명에 버전 숫자를 붙여 넣는다. 아래처럼 버전을 넣지 않는 경우, 최신 버전으로 설치가 된다. 일반 노드는 가급적이면 kubectl 패키지를 설치하지 않는다. 설치해도 크게 문제는 없기는 하지만, 노드 보안상 보통 제외한다.

---

<sup>7</sup> 2024년 기준이다.

```
master]# dnf install kubeadm kubelet kubectll -y --
disableexcludes=kubernetes
node]# dnf install kubeadm kubelet -y --disableexcludes=kubernetes
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
master]# for i in master node{1..2} ; do sshpass -pcentos \
ssh root@$i dnf install kubeadm -y ; done
```

## 런타임 설치 및 구성

런타임(runtime)은 흔히 말하는 컨테이너를 생성하는 일종의 환경관리자이다. 이를 통해서 인프라 컨테이너 및 애플리케이션 컨테이너를 생성한다. 쿠버네티스에서 사용하는 런타임은 일반적으로 많이 사용하는 docker, podman 으로 사용이 안되며, OCI 및 CRI 사양에 맞는 컨테이너 런타임만 제공한다.

### CRI-O

또한 쿠버네티스에 컨테이너를 실행하기 위해서 런타임 설치가 필요하다. 여기서는 CRI-O 를 사용하지만, 원하시는 경우 containerd, cri-docker 설치 및 사용이 가능하다.

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

설정 파일은 아래 주소에서도 내려받기가 가능하다. 아래 주소는 전부 한줄이다.

```
master/node]# curl -o /etc/yum.repos.d/stable_crio.repo
https://raw.githubusercontent.com/tangt64/training_memos/main/opensource-
101/kubernetes-101/files/stable_crio.repo
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
master/node]# for i in master node{1..2} ; do sshpass -pcentos \
```

```
scp /etc/yum.repos.d/{crio_stable.repo} \  
root@${i}:/etc/yum.repos.d/ ; done
```

저장소 설정이 완료가 되면 다음 명령어로 cri-o 런타임을 설치한다.

```
master/node]# yum install cri-o | crio -y
```

설치를 하기 위해서 CRI-O 를 미리 실행한다.

```
master/node]# systemctl enable --now crio.service
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

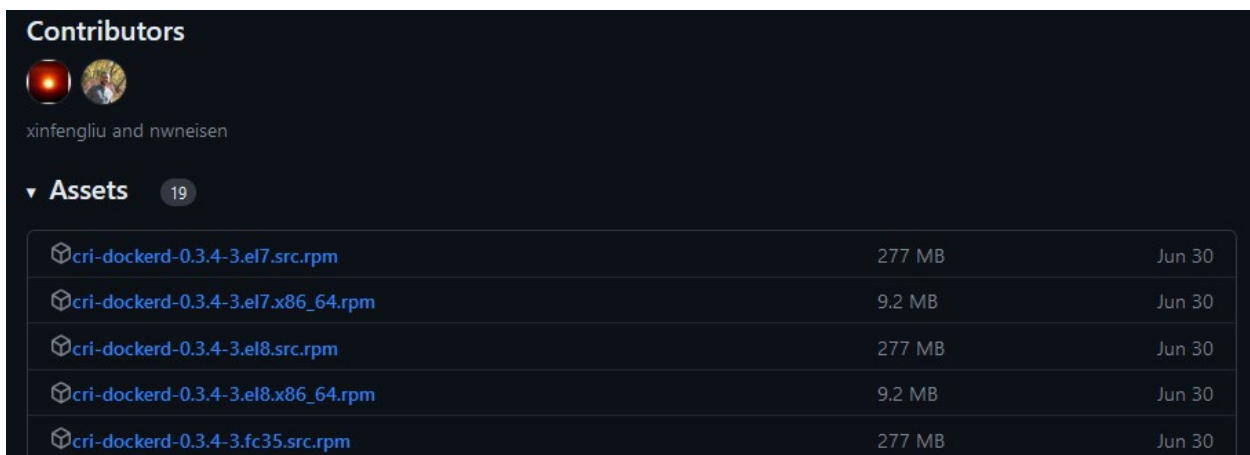
```
master]# for i in master node{1..2} ; do sshpass -pcentos \  
ssh root@${i} "dnf install cri-o -y && systemctl enable --now  
crio.service" ; \  
done
```

## cri-docker

현재 cri-docker 는 Mirantis 깃헙에서 제공하고 있다. 지원하는 패키지는 지원하는 버전은 RHEL 7/8 그리고 FC36<sup>8</sup>이다. 아래 주소에서 릴리즈 된 패키지를 받을 수 있다.

- <https://github.com/Mirantis/cri-dockerd/releases>

RHEL 9 혹은 CentOS-9-Stream 기반으로 되어 있는 경우, 아직 지원하지 않기 때문에 SRPM 기반이나 혹은 소스코드 기반으로 다시 컴파일을 해야 한다.



The screenshot shows the GitHub repository for cri-dockerd. It lists two contributors: xinfengliu and nwneisen. Under the 'Assets' tab, there are 19 assets listed. The visible assets are:

Asset Name	Size	Date
cri-dockerd-0.3.4-3.el7.src.rpm	277 MB	Jun 30
cri-dockerd-0.3.4-3.el7.x86_64.rpm	9.2 MB	Jun 30
cri-dockerd-0.3.4-3.el8.src.rpm	277 MB	Jun 30
cri-dockerd-0.3.4-3.el8.x86_64.rpm	9.2 MB	Jun 30
cri-dockerd-0.3.4-3.fc35.src.rpm	277 MB	Jun 30

참고로 이 책에서는 cri-docker 를 사용하지 않는다.

---

<sup>8</sup> FC: Fedora Core 의 약자, EL 은 Enterprise Linux 의 약자

## containerd

Containerd 를 사용하는 경우, 배포되는 'containerd.toml'파일이 도커에서 배포한 파일이기 때문에 재구성이 필요하다. 소스 컴파일을 원하지 않는 경우, 도커 저장소를 아래처럼 구성한다. 참고로 레드햇 계열의 배포판은 containerd 를 제공하지 않는다. 다만, 페도라 리눅스는 이 패키지를 제공하고 있다. 도커 저장소 정보는 언제든지 변경이 될 수 있으니, 아래 내용은 참조만 한다.

```
master]# cat <<EOF> /etc/yum.repos.d/docker.repo
[docker-ce-stable]
name=Docker CE Stable - $basearch
baseurl=https://download.docker.com/linux/centos/$releasever/$basearch/stable
enabled=1
gpgcheck=1
gpgkey=https://download.docker.com/linux/centos/gpg
EOF
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
master/node]# for i in master node{1..2} ; do sshpass -pcentos \
scp /etc/yum.repos.d/docker.repo /etc/yum.repos.d/docker.repo ; \
done
```

저장소 등록 후, 아래 명령어로 containerd 패키지 설치 그리고 'config.toml'파일을 갱신한다. 이 과정이 올바르게 수행이 되지 않으면 런타임 containerd 및 쿠버네티스 설치가 올바르게 되지 않는다.

```
master/node]# dnf install containerd
master/node]# containerd config default > /etc/containerd/config.toml
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
master/node]# for i in master node{1..2} ; do sshpass -pcentos \
ssh root@$i "dnf install containerd -y && containerd config default >
/etc/containerd/config.toml && systemctl enable --now containerd" ; \
done
```

## firewalld

방화벽을 중지 혹은 설정을 한다. 만약 방화벽을 사용하는 경우 다음과 같은 포트가 반드시 시스템에 켜져 있어야 한다. 'firewall-cmd'를 통해서 아래 포트 번호를 등록한다.

하나씩 입력하기 어려우면, 아래처럼 스크립트 라인으로 쉘을 통해서 등록한다.

```
master]# for i in master node{1..2} ; do sshpass -pcentos ssh root@${i}
"firewall-cmd --permanent \
--add-port={6443/tcp,3279-
2380/tcp,10250/tcp,10251/tcp,10252/tcp,10255/tcp,30000-32767/tcp,179/tcp}
&& hostname" \
; done
6443/tcp
2379-2380/tcp
10250/tcp
10251/tcp
10252/tcp
10255/tcp
30000-32767/tcp
179/tcp
master/node]# firewall-cmd --reload
master/node]# firewall-cmd --list-all
```

방화벽에 등록한 포트번호는 아래 표로 별도로 적어 두었다. 본래 마스터 및 워커 노드는 각각 다른 포트번호를 사용한다. 책에서는 스크립트로 **마스터+워커 노드**에 같은 포트번호를 등록했다.

### 컨트롤 플레인

프로토콜	방향	포트 범위	용도	비고
TCP	인바운드	6443	kubelet	전부
TCP	인바운드	2379-2380	etcd, API Client	kube-apiserver, etcd
TCP	인바운드	10250	Kubelet API	자체, 컨트롤 플레인
TCP	인바운드	10259	kube-scheduler	자체
TCP	인바운드	10257	kube-controller-manager	자체

### 워커 노드

프로토콜	방향	포트 범위	용도	비고
TCP	인바운드	10250	Kubelet API	자체, 컨트롤 플레인
TCP	인바운드	30000-32767	NodePort 서비스†	전부

방화벽을 사용하지 않으면 다음 명령어로 중지하면 된다. 이런 경우 별도로 방화벽에 포트를 등록하지 않아도 된다. 일반적으로 PoC 나 랩에서는 사용하지 않는다.



```
master/node]# systemctl stop firewalld
master/node]# systemctl disable firewalld
```

## tc/swap

트래픽 셰이핑을 하기 위해서 tc 패키지를 설치한다. 이 패키지가 설치가 안되어 있는 경우 쿠버네티스 설치가 진행이 되지 않는다. master 노드에는 설치하지 않아도 괜찮지만, 반드시 worker 에는 설치가 되어 있어야 한다. 만약, 마스터 노드에서 컨테이너 서비스가 동작하는 경우, 똑같이 tc 패키지를 설치한다.

추가로, 현재 최근 버전의 쿠버네티스를 사용하는 경우, tc 및 swap 에 대해서 **경고**만 출력하고 설치의 진행이 된다.

```
master/node]# dnf install iproute-tc -y
```

스왑과 SELinux 를 중지한다. SELinux 는 중지하지 않아도 되지만, 가급적이면 편하게 쓰기 위해서 SELinux 를 중지한다.

```
master/node]# swapoff -a
master/node]# sed -i '/ swap / s/^#/' /etc/fstab
master/node]# sed -i s/^SELINUX=.*/SELINUX=permissive/
/etc/selinux/config
```

## /etc/sysconfig/kubelet

CRI-O 를 사용하는 경우 쿠버네티스에 런타임 위치 및 socket 파일의 위치를 설정파일에 명시한다. 구성을 변경하려면 /etc/sysconfig 에 다음처럼 입력한다. 현재는 아래와 같은 설정은 굳이 수동으로 하지 않아도 된다. 별도의 설정이 필요하면, 아래와 같이 구성한다.

```
master/node]# cat <<EOF>> /etc/sysconfig/kubelet
KUBELET_EXTRA_ARGS=--cgroup-driver=systemd --container-runtime-
endpoint="unix:///var/run/crio/crio.sock"
EOF
```

최신 쿠버네티스 'kubeadm'를 사용하는 경우 자동으로 컨테이너 런타임 소켓을 확인하기 때문에 아래 파일은 굳이 수동으로 만들지 않아도 된다. 수동 설정이 필요한 경우, 위의 파일을 따로 생성하면 된다.

## 7. 쿠버네티스 설치: 싱글 마스터 설치

간단하게 컨트롤 노드를 설치 시, 다음과 같은 명령어로 설치한다. 아래는 명령어는 master 에서 실행한다. 실제로 아래 명령어는 kubeadm init 를 실행하였을 때 이미 출력이 되었다.

먼저, SELinux가 동작 중이면, 진행을 위해서 잠깐 중지를 한다.

```
master/node]# getenforce
master/node]# setenforce 0
```

쿠버네티스 서비스를 시작하기 위해서 kubelet 서비스를 시작한다. 앞에서 이야기하였지만, 노드 구성하기 위해서는 init 하위 명령어를 통해서 작업을 수행한다. 마스터 서버에서 init 명령어를 아래와 같이 실행한다. 만약, 서비스 도메인 관리가 필요한 경우에는 --service-dns-domain 를 통해서 변경이 가능하다. cluster.local 이 기본 도메인 값이다.

```
master]# systemctl enable --now kubelet
master]# kubeadm init --apiserver-advertise-address=192.168.10.250 \
--cri-socket=unix:/var/run/crio/crio.sock --pod-network-
cidr=192.168.0.0/16 \
--service-cidr=10.90.0.0/16 --service-dns-domain=devops.project
```

위의 명령어를 실행하면 보통 다음과 같은 메시지를 맨 마지막 화면에 출력한다. 아래 출력된 메시지 중 두껍게 출력된 부분을 복사해서 다른 곳에 보관한다.

```
...
kubeadm join 192.168.10.250:6443 --token 5b0ncr.xueohcazoj7n8hfw \
--discovery-token-ca-cert-hash
sha256:0fc6a639002616c3b817c1439b60473a57681cc0e5e54629c0333cef882eec93
...
```

kubeadm init 명령어를 실행하면, 레드햇 계열의 배포판은 "policy.json"에서 오류가 종종 발생한다. 오류가 발생하면 아래와 같이 JSON 내용을 수정한다. Rocky Linux 9 은 별도로 조정이 필요가 없다.

```
{
  "default": [
    {
      "type": "insecureAcceptAnything"
    }
  ],
  "transports":
  {
    "docker-daemon":
    {
```

```

        "": [{"type": "insecureAcceptAnything"}]
    }
}
}

```

설치가 진행 중, 오류가 발생이 되어서 중지가 되면, 아래와 같이 명령어를 실행한 후, 다시 설치를 진행한다.

```
master]# kubeadm reset
```

빠르게 초기화 하기 위해서는 "--force" 옵션을 붙이면 빠르게 쿠버네티스 설정 파일을 서비스 중지 및 삭제를 한다. 최신버전 기준, kubeadm v1.26 에서는 기본적으로 마스터 확장을 지원하고 있다. 이전에는 보통 etcd 서버에 TLS 키를 업로드 하여 공유하기 위해서 옵션을 별도로 사용하였지만, 현 버전은 그럴 필요가 없다. 하지만, 학습 목적으로 기존에 사용하던 옵션은 그대로 사용한다.

올바르게 실행이 되면 아래 명령어를 실행한다. 이 과정이 빠지면 'kubectl' 명령어 사용이 안된다. 쿠버네티스 설정 파일을 홈 디렉터리 ".kube/"에 복사한다.

```

master]# mkdir -p -m 0700 ~/.kube/
master]# cp /etc/kubernetes/admin.conf ~/.kube/config
master]# chown -c root:root ~/.kube/config
master]# kubectl get nodes

```

혹은 파일 복사가 싫은 경우, 아래와 같이 변수를 설정해서 kubectl 명령어 사용이 가능하다. 영구적으로 사용하기 위해서는 .bashrc 파일 추가한다.

```

master]# export KUBECONFIG=/etc/kubernetes/admin.conf
master]# kubectl get nodes
master]# echo "KUBECONFIG=/etc/kubernetes/admin.conf" >> ~/.bashrc

```

워크 노드 구성 시 다음과 같은 명령어로 실행한다. 워커 노드에서 kubeadm 명령어가 실행이 되지 않으면, kubeadm 패키지만 설치해주면 된다. 토큰은 init 명령어가 수행이 된 이후에 확인이 가능하다. 노드 추가 시 1 번만 하며 2 번은 추후에 진행한다.

```

node1]# kubeadm join 192.168.68.250:6443 --token gcd426.2itdtcs7o1p7ds6r
\
--discovery-token-ca-cert-hash \
sha256:bfad2126496a0779ccbc2cf9b841834cc930384eb0158cce40332f74a7b544d7

```

워크 노드가 한 개 이상인 경우에는 각각 노드에서 join 명령어를 실행한다. 모든 노드에서 join 를 실행하였으면 master 에서 다음과 같은 명령어로 올바르게 노드가 구성이 되었는지 확인한다.

```
master]# kubectl top nodes
```

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
master.example.com	75m	3%	1256Mi	34%
node1.example.com	42m	2%	639Mi	17%

싱글 노드는 추가하기 위해서 init, join 이 두 가지 하위 명령만 알고 있으면 된다. 마지막으로 자원이 부족해서 마스터 서버만 운영이 가능한 경우, 랩 진행을 위해서 아래와 같이 명령어를 입력한다.

```
master]# kubectl taint node master.example.com node-  
role.kubernetes.io/control-plane:NoSchedule-  
master]# kubectl describe node/master.example.com  
> Taints:
```

## 8. 쿠버네티스 설치: 다중 마스터 설치

다중 마스터를 구성하기 위해서는 반드시 다음과 같은 조건을 구성해야 한다.

- 한 개 이상의 마스터 서버. 마스터는 절대로 홀수로 구성해야 한다. 최소 3 개.
- ETCD 를 외부에 독립적으로 구성 시, 노드와 동일하게 최소 3 개 이상을 동작하도록 한다.

설치는 기존 단일 마스터 노드와 똑같다. 다만, 명령어에 사용하는 옵션은 조금 다르다. 기존과 동일하게 내부 아이피를 사용한다. 하지만, 3 대 이상의 마스터 서버가 구성이 되면, 반드시 서버는 VIP 주소를 가지고 있어야 한다. 이 책에서는 다중 마스터 구성에 대해서는 다루지 않는다.

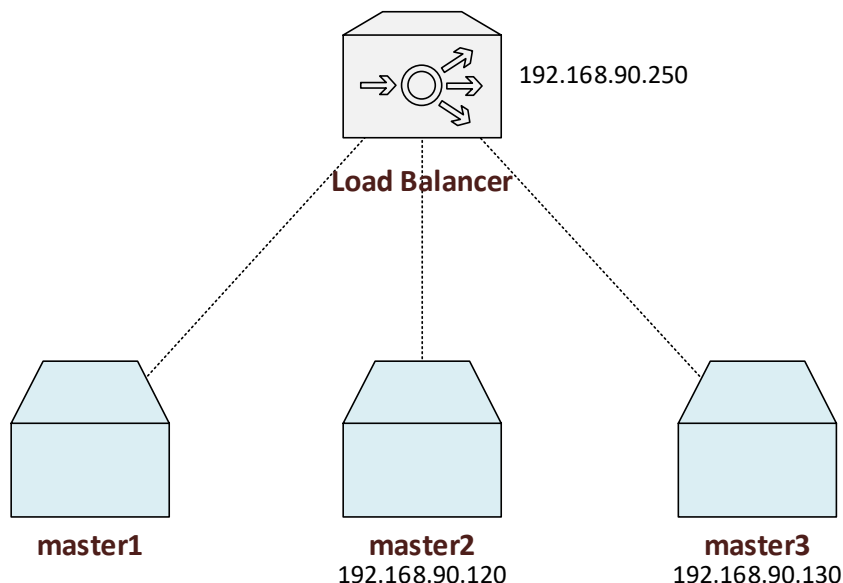


그림 9 Load Balancer, Master Nodes

VIP 를 192.168.10.250 를 사용한다는 조건으로 아래와 같이 명령어를 사용한다.

```
master]# kubeadm init --control-plane-endpoint 192.168.10.250 --upload-
certs --apiserver-advertise-address 192.168.10.250 --pod-network-
cidr=192.168.0.0/16 --service-dns-domain=devops.project --image-
repository=192.168.90.240
```

컨트롤 패널(마스터)의 네트워크 카드는 192.168.10.250 를 통해서 외부나 혹은 내부에서 들어오는 API 를 처리한다. "--upload-certs"는 부트 스트랩 노드(master1 혹은 bootstrap)에서 생성한 TLS 키를 ETCD 서버에 저장한다. 저장된 TLS 키는 새로 구성되는 마스터 노드에 같이 TLS 키를 배포한다. "--apiserver-advertise-address"는 master1 서버가 API 를 외부에 전달하는 네트워크 카드이다. 현 랩에서는 eth1 장치가 API 를 다른 노드에 있는 쿠버네티스에 전달한다.

최종적으로 "--pod-network-cidr"는 POD 가 클러스터에서 사용하는 네트워크 주소 대역. "--service-dns-domain"는 내부 coredns 에서 사용하는 서비스 도메인 이름을 기본값인 "cluster.local"에서 사용자가 명시한 이름으로 변경한다. "--image-repository"는 내부 혹은 폐쇄망에서 설치 시, 내부에 구성이 되어있는 컨테이너 이미지 서버 주소이다.

이외 나머지 부분은 기존 싱글 마스터 설치와 동일하다.

```
master/console]# mkdir -p -m 0700 ~/.kube/
master/console]# cp /etc/kubernetes/admin.conf ~/.kube/config
master/console]# chown -c root:root ~/.kube/config
master/console]# kubectl get nodes
```

클러스터에 마스터 노드로 추가하기 위해서 아래와 같이 명령어를 실행한다. 참고로, 최신 버전의 kubeadm 를 사용하는 경우, 마스터 구성 후 아래와 같은 명령어가 같이 화면에 출력이 된다.

```
nodeX]# kubeadm join 192.168.90.100:6443 --token hhjg3n.nmfs94gop1aeom0n
--discovery-token-ca-cert-hash
sha256:01e7f7dee3594c99fa1cb50ad20f3b0b6e1f74d2afaf5f5dc1892bc1b6f247b1 \
--control-plane \
--certificate-key
8dc065689de5eb04b57f5538d46bcda977df33b586bcbf4b922fd676eeb0d41e
```

--control-plane 옵션은 워커 노드, 즉, 컴퓨팅 노드가 아니라 서비스 컨테이너 구성 용도로 사용한다고 명시한다. 올바르게 구성이 되면, 아래와 같이 화면에 출력이 된다. 멀티 노드를 구성한 경우 반드시 마스터 노드가 1 개 이상 출력이 되어야 한다.

```
master]# kubectl top nodes
```

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
master1.example.com	75m	3%	1256Mi	34%
master2.example.com	18m	5%	300Mi	11%
master3.example.com	12m	1%	500Mi	8%
node1.example.com	42m	2%	639Mi	17%
node2.example.com	37m	1%	1109Mi	30%

## 9. 쿠버네티스 설치: 네트워크 구성

여기서 구성하는 쿠버네티스 클러스터는 Calico 기반으로 네트워크 구성을 한다.



PROJECT  
CALICO

그림 10 칼리코 고양이 프로젝트

단일 혹은 다중 마스터를 구성한 경우 아래와 같이 명령어를 실행하여 네트워크 생성 및 구성한다. 원하는 네트워크 구성원이 있으면 원하는 CNI 플러그인 기반으로 구성하여도 된다. 일반적으로 쿠버네티스 네트워크에 flannel 네트워크를 원하는 경우, Flannel 네트워크를 사용하여도 문제없다.

먼저 쿠버네티스에서 사용할 Calico 컨트롤러 및 오퍼레이터 구성을 한다. 미리 사이트에서 구성한 칼리 오퍼레이터 설정을 사용한다.

```
master]# kubectl create -f
https://raw.githubusercontent.com/tangt64/duststack-k8s-
auto/master/roles/cni/cni-calico/files/tigera-operator.yaml
```

구성이 완료가 되면, Pod 대역 및 이미지 레지스트리 위치 및 Pod 네트워크 대역을 정의한다. 아래와 같이 파일 작성 후 쿠버네티스에 적용한다.

```
ipPools:
  - blockSize: 26
    cidr: 192.168.0.0/16
    encapsulation: VXLANCrossSubnet
```

```
natOutgoing: Enabled
nodeSelector: all()
registry: quay.io
```

위의 내용으로 수정 후, 적용하면 쿠버네티스는 칼리코 오퍼레이터를 통해서 네트워크를 구성한다. 만약, Pod 네트워크 정보가 올바르지 명시되지 않는 경우, 생성이 되지 않는다. 저자가 사용하는 네트워크는 아래에서 사용이 가능하다.

```
master]# kubectl create -f
https://raw.githubusercontent.com/tangt64/duststack-k8s-
auto/master/roles/cni/cni-calico/templates/custom-resources.yaml
```

문제없이 적용이 되면 다음처럼 Pod 확인이 가능하다.

```
master]# kubectl get pods -n calico-apiserver
NAME                                READY   STATUS    RESTARTS   AGE
calico-apiserver-5b4bbd4cd9-4d7xj  1/1     Running   0           7m49s
calico-apiserver-5b4bbd4cd9-bqrhl  1/1     Running   0           7m49s
master]# kubectl get pods -n calico-system
NAME                                READY   STATUS    RESTARTS   AGE
calico-kube-controllers-6d49956dbf-k5tl5  1/1     Running   0           10m
calico-node-2wzhd                        1/1     Running   0           10m
calico-node-bnkkw                        1/1     Running   0           10m
calico-typha-68d49b56d6-58sg8            1/1     Running   0           10m
csi-node-driver-dnmwh                    2/2     Running   0           10m
csi-node-driver-r8x9z                    2/2     Running   0           10m
master]# kubectl get pods -n tigera-operator
NAME                                READY   STATUS    RESTARTS   AGE
tigera-operator-94d7f7696-zwrls        1/1     Running   0           11m
```

여기까지 완료가 되면, 마스터 및 워커 노드에 구성된 자료는 문제없이 외부와 네트워크 통신 및 구성이 가능하다.

# 표준 컨테이너 도구

## 1. 표준 OCI 도구와 관계

## 2. Buildah

Buildah는 기존 Docker가 이미지 빌드하였던 기능을 분리하여 별도로 만든 컨테이너 이미지 빌드 도구. 현재 OCI에서 이미지 구성 시 권장하는 도구는 buildah이다. 대다수 배포판에서는 기본으로 제공하고 있으며, 언제든지 설치 및 사용이 가능하다.

아래 내용은 centos-9-stream<sup>9</sup>기반에서 설치이다.

```
master]# dnf install container-tools -y
master]# dnf module list
```

빌더(buildah)<sup>10</sup>는 Podman에서 이미지 빌드 기능만 분리하여 만든 도구이다. 빠르게 이미지 빌드 시, 이를 통해서 구성이 가능하다.

```
# buildah bud -f Containerfile-blog -t
registry.example.com:5000/release/blog:v1
# buildah images
# buildah push registry.example.com:5000/release/blog:v1
```

## 3. Skopeo

'podman search'명령어 기능을 skopeo로 재구성하였다. Skopeo의 주요 목적인 이미지 검색/복사/확인 그리고 미러링 같은 기능을 제공한다. Skopeo도 buildah처럼 OCI에서 권장하는 이미지 관리 도구 중 하나이다. 설치하는 위에서 이미 컨테이너 도구 메타 패키지(container-tools)로 설치하였기 때문에, 별도로 설치를 하지 않아도 된다.

```
# dnf install skopeo -y
```

앞에서 사용한 buildah 기준으로 조회를 한다면 다음과 같이 업로드한 프로그램 이미지의 태그 버전 조회가 가능하다. 다만, 저장소 검색은 podman search와 같은 명령어로 해야 한다.

---

<sup>9</sup> centos-8-stream은 module로 설치해야 한다. 가급적이면 container-tools:3.0으로 설치한다.

<sup>10</sup> 일본 친구가 빌더아라고 발음을 알려주었다. 그리고 한동안 사용하였다.



```
# skopeo list-tags docker://registry.example.com:5000/release/blog
```

다른 기능으로 이미지를 원격에서 원격, 원격에서 로컬 혹은 이 반대로 가능하다. 이를 통해서 이미지 백업 및 저장소 미러링이 가능하다.

```
# skopeo copy docker://registry.example.com:5000/release/blog:v1 oci-  
archive://release-blog-v1.tar  
# skopeo copy docker://ext-registry.example.com:5000/release/blog:v1  
docker://int-registry.example.com:5000/internal-release/blog:v1  
# skopeo sync docker://ext-registry.example.com:5000/release/  
docker://int-registry.example.com:5000/internal-release/
```

## 4. Tekton(테크톤)

많은 CI/CD 도구들이 나오기 시작하면서, 표준화가 필요하게 되었다. 테크톤은 쿠버네티스 진영에서는 표준 CI/CD 및 Pipe 도구로 사용하고 있다. 현재 테크톤은 쿠버네티스 및 오픈 시프트 양쪽에서 사용이 가능하다. 테크톤은 쿠버네티스 API 와 함께 사용이 가능하며, 쿠버네티스와 동일하게 YAML 기반으로 선언 및 자원 관리가 가능하다.

각 과정은 POD 기반으로 실행이 되며, 이러한 각 작업들은 파이프라인으로 통합 및 실행이 가능하다. 현재 대다수 쿠버네티스 CD 환경은 ArgoCD 및 Tekton 으로 많이 사용하고 있다. 이 책에서는 테크톤을 사용하여 어떻게 CI/CD 를 구성 및 활용하는지 간단하게 다루어 본다.

## 5. Containerd/CRI-O/CRI-Docker

쿠버네티스는 컨테이너 인프라 통합운영이 주요 목적이며, 컨테이너 환경을 직접 제공하지 않는다. 현재 쿠버네티스는 다음과 같은 컨테이너 런타임(runtime)을 지원한다.

- docker-shim(더 이상 사용하지 않음)
- cri-docker(OCI)
- CRI-O(OCI)
- Contained (OCI)
- rkt(Rocket, 더 이상 사용하지 않음)
- lxc/lxd(쿠버네티스에서 지원하지 않음)

런타임은 사용자가 원하는 환경 혹은 프로그램 선택이 가능하며, 쿠버네티스는 containerd, docker, cri-o 지원한다.

쿠버네티스에서 사용하는 런타임 환경은 containerd 및 CRI-O 환경이다. 앞으로 Docker 는 런타임으로써 더 이상 지원하지 않는다고 발표하였다 <sup>11</sup>.

We formally announced the dockershim deprecation in December 2020. **Full removal is targeted in Kubernetes 1.24, in April 2022.** This timeline aligns with our deprecation policy, which states that deprecated behaviors must function for at least 1 year after their announced deprecation. Jan 7, 2022

그림 11 docker-shim 지원 중지

현재 오픈소스 컨테이너는 OCI(Open Source Container Initiative)라는 표준 사양을 정의하였으며, 이 사양을 따르는 컨테이너 런타임은 위에서 이야기하였던 containerd, cri-docker, cri-o 가 전부이다.

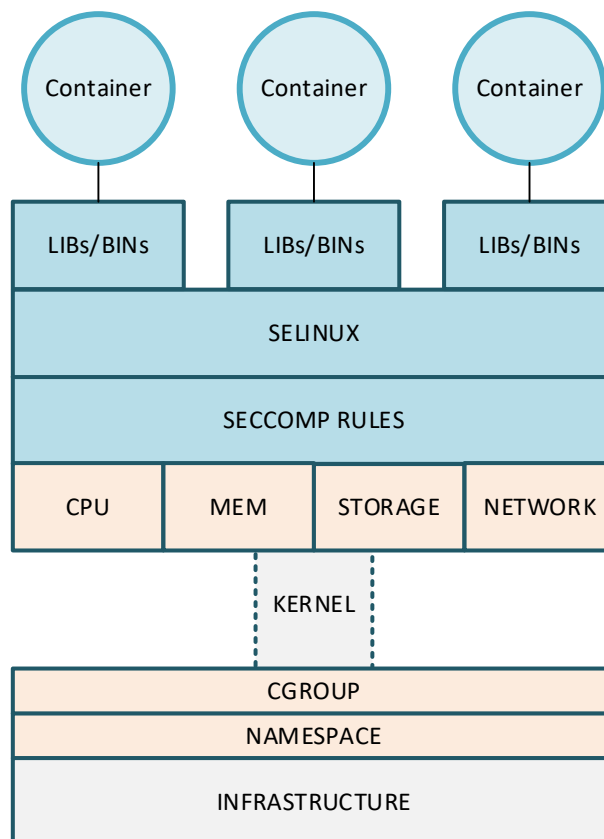


그림 12 컨테이너 구성원

위의 그림은 런타임 엔진을 구성 및 구현하기 위해서 필요한 커널 기술을 계층으로 표현했다. 위와 같이 시스템 구성이 되면, 런타임은 kubelet 과 CRI 통해서 컨테이너 생성을 한다. 이때 CRI 엔진이 달라도, 동일한 CRI 사양을 따르고 있기 때문에, 문제없이 컨테이너 생성 및 관리가 가능하다.

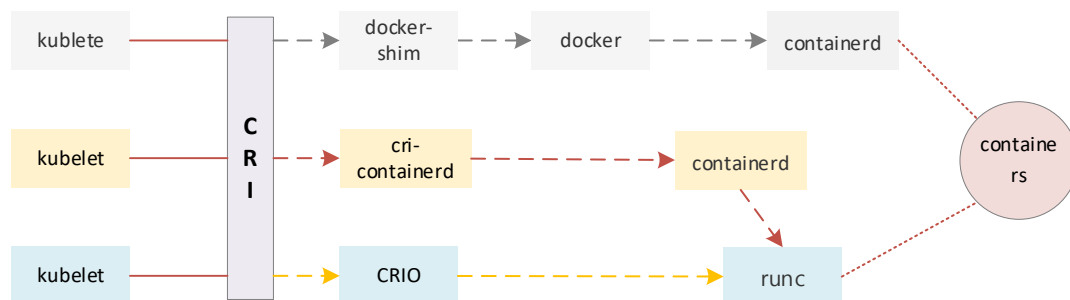


그림 13 컨테이너 CRI

<sup>11</sup> 하지만 shim 구조가 빠진 도커는 여전히 지원하고 있다.

구버전의 도커는 모든 자원들 즉, namespace, cgroup 를 직접 관리하였다. 하지만, Containerd, cri-docker 경우에는 OCI 사양을 따르면서 현재는 직접 관리하지 않는다.<sup>12</sup> kubelet 은 별도로 명시하지 않으면, containerd 를 기본 런타임으로 사용하며, 사용자가 원하는 경우, CRI-O 를 선택하여 사용이 가능하다. 이 과정에서는 기본적으로 CRI-O 기반으로 사용하기 때문에, 설명은 CRI-O 기반으로 진행한다.

CRI-O 의 주요 목적은 컨테이너를 최소화하여 시스템에서 발생하는 작업량을 최소화하여 좀 더 민첩한 컨테이너 환경을 제공한다. CRI-O 의 약자는 "**Container Runtime Interface Open Container Initiative**)"이다. containerd 는 생성된 컨테이너의 샌드박스(sandbox) 및 namespace 및 c-group 를 통합 관리, 이 기반으로 kubernetes 및 openshift 에서 사용하는 POD 를 구성한다.

CRI 인터페이스를 지원하는 컨테이너 동작 방식은 기본적으로 로컬 런타임으로 사용하며, 소켓(socket)기반으로 사용한다. 대다수 쿠버네티스 컨테이너 런타임은 소켓을 통해서 kubelet 과 연결 및 구성이 된다. 아래 그림은 CRI-O 기반으로 구성된 쿠버네티스 구성이다. 현재 쿠버네티스의 컨테이너 런타임은 CRI-O 사용을 권장하고 있으며, CNCF 및 쿠버네티스에서는 containerd 를 표준으로 채택하고 있다.

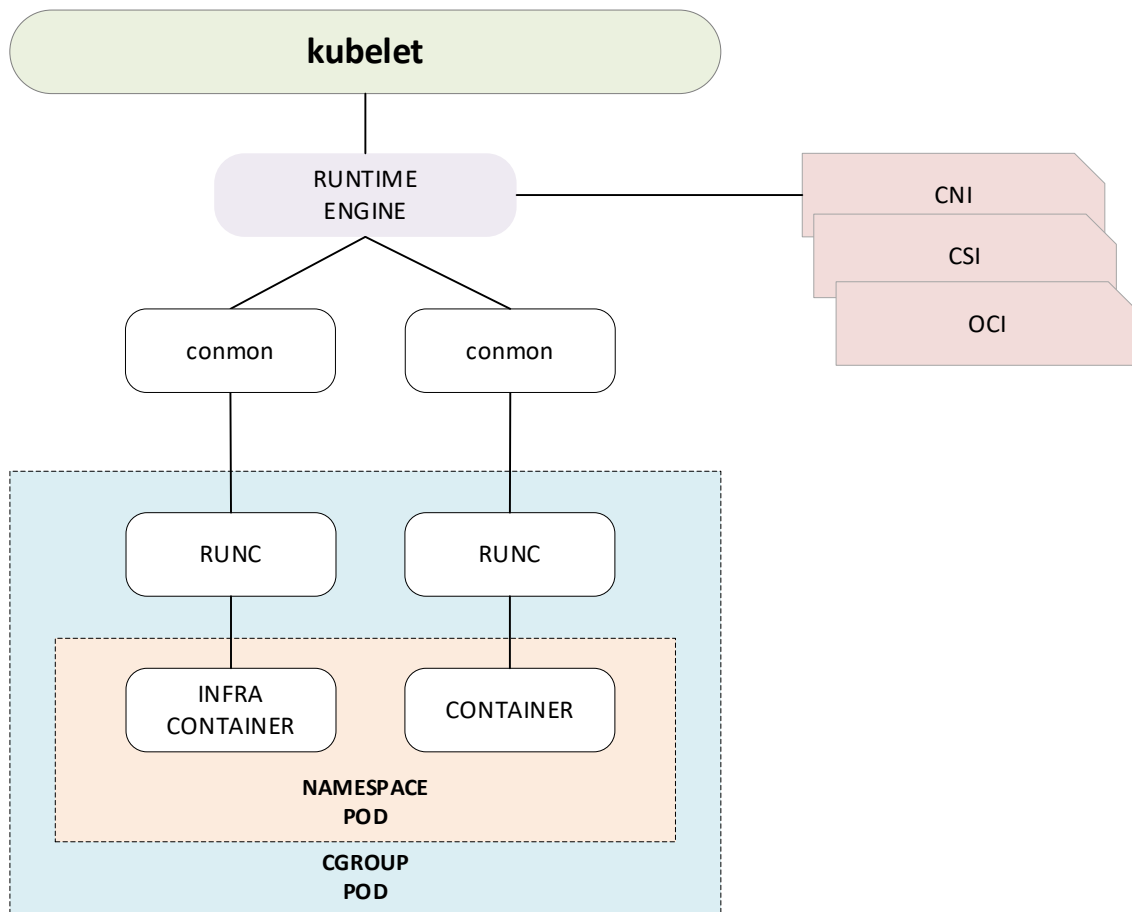


그림 14 CRI-O 구조

위의 그림은 CNI, sandbox(namespace), resource control(c-group), OCI 관계에 대해서 설명하고 있다. 현재 모든 컨테이너 런타임들은 주요 기능들이 커널에서 구성이 된다. 이렇기 때문에 리눅스 커널의 버전에 따라서 컨테이너에서 제공되는 기능이 다를 수 있다.

<sup>12</sup> docker-shim 참조.

만약, 런타임 엔진(RUNTIME ENGINE)이 CRI-O 이면, 런타임은 엔진에서 사용하는 이미지 및 컨테이너를 직접 관리하며, CRI-O 컨테이너 환경은 표준 인터페이스인 OCI 기반으로 gRPC 및 CNI(Container Network Interface)를 통해서 구성이 된다. 우리가 일반적으로 컨테이너 런타임(runtime)은 컨테이너 라이프 사이클 관리가 주요 목적이다.

쿠버네티스 클러스터를 구성하기 위해서는 리눅스 커널에서 다음과 같은 기능을 요구한다.

- namespace
- cgroup
- seccomp(시스템 및 네트워크 제어)
- nftables/iptables(iptables legacy)

추가적으로 요구하는 기술 및 프로그램은 다음과 같다.

- SELinux
- App-Armor

SELinux<sup>13</sup>는 seccomp<sup>14</sup>와 비슷하게 필터링 하지만, 더 세밀하게 시스템 콜 및 네트워크 접근에 대한 시스템 보안을 원하는 경우 SELinux 사용이 가능하다. 하지만, "SELinux Policy"<sup>15</sup>에 대한 지식이 없으면, 일반적으로 끄고 사용하는 걸 권장한다.

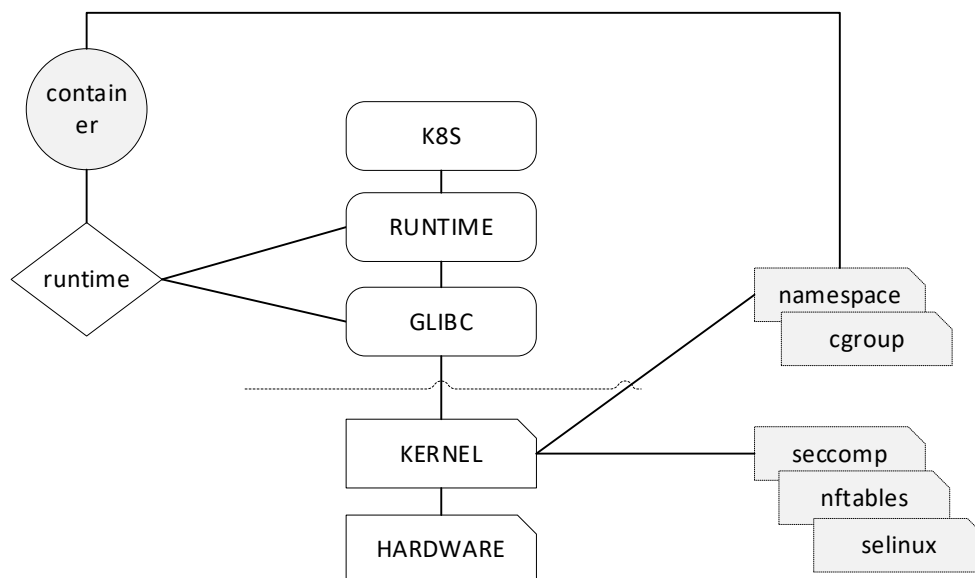


그림 15 컨테이너 및 기술 관계

<sup>13</sup> [https://selinuxproject.org/page/Main\\_Page](https://selinuxproject.org/page/Main_Page)

<sup>14</sup> <https://security.stackexchange.com/questions/264273/what-does-default-seccomp-apparmor-and-selinux-in-kubernetes-security-mean>

<sup>15</sup> [https://selinuxproject.org/page/Building\\_a\\_Basic\\_Policy](https://selinuxproject.org/page/Building_a_Basic_Policy)

## Containerd

Containerd 는 CNCF 에서 시작하였으며 이 프로젝트는 윈도우 및 리눅스 양쪽에서 실행이 가능하다. 주요 목적은 컨테이너의 라이프 사이클 관리가 주요 목적이며 컨테이너에 저장소 및 네트워크 기능을 제공한다. Containerd 는 도커의 하위 서비스인 dockerd 밑에서 containerd 라는 이름으로 동작하였지만, 지금은 도커에서 분리하여 독립적인 컨테이너 런타임으로 사용이 가능하다.

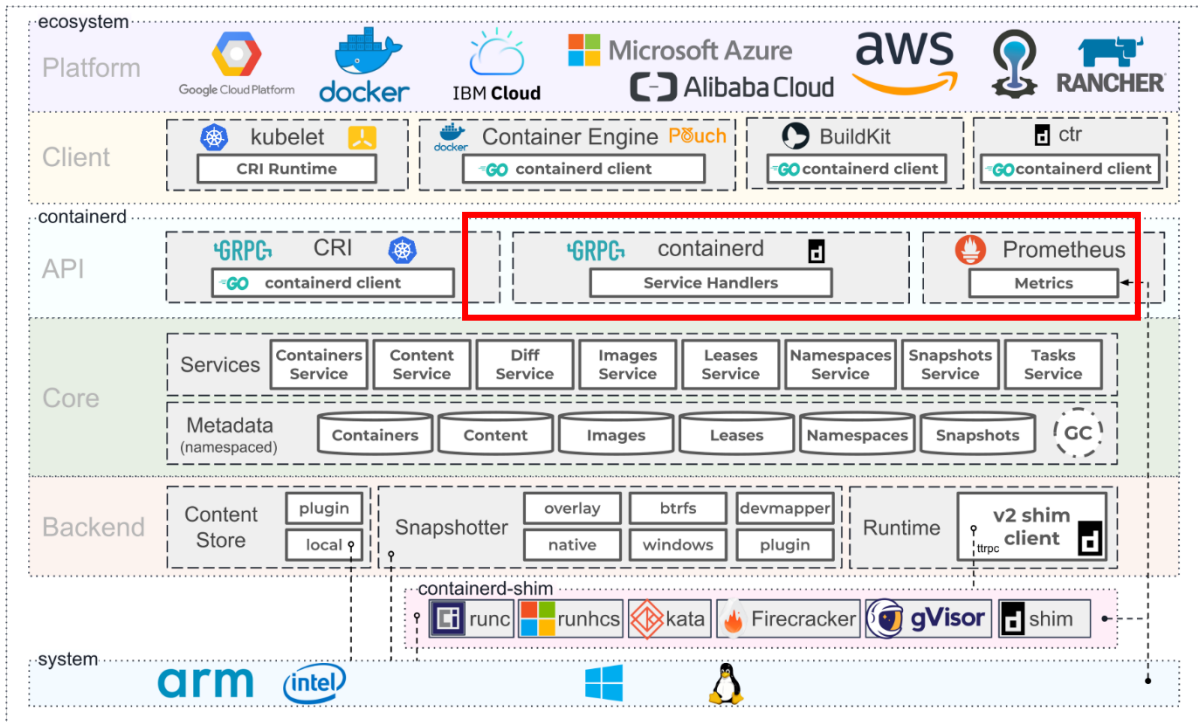


그림 14 컨테이너 소프트웨어 계층 구성

containerd는 CRI-O처럼 runc-shim 기반으로 동작하며 OCI, CNI와 같은 표준 사양을 준수한다. containerd는 쿠버네티스에서 지원하는 런타임 중 하나이다. cri-docker 와 containerd 는 비슷하지만 구조가 다르다. cri-docker 경우에는 docker-shim 를 지원하며, containerd는 설정에 따라 다르지만, 쿠버네티스와 연동하여 사용할 때, shim 를 지원하지 않는다. docker 를 인수한 Mirantis 는 앞으로 Mirantis Kubernetes Engine 에서 cri-docker 를 사용할 계획이다. 참고로 docker-shim 은 ps 명령어로 확인 시, 다음과 같이 출력이 된다.

```
root 9499 0.0 0.1 154660 8856 ? Ss Dec16 0:00 \ sshd: vagrant [priv]
vagrant 9502 0.0 0.0 154276 5140 ? S Dec16 0:03 | \ sshd: vagrant@pts/0
vagrant 9503 0.0 0.0 116640 4856 pts/0 Ss Dec16 0:00 | \ -bash
root 24291 0.0 0.0 241176 7692 pts/0 S+ 05:52 0:00 | \ sudo docker run -it ubuntu bash
root 24293 0.0 0.6 365000 53084 pts/0 S+ 05:52 0:00 | \ docker run -it ubuntu bash
root 3191 0.0 0.5 681064 48296 ? Ssl Dec16 0:03 /usr/bin/containerd
root 24322 0.0 0.0 108752 7792 ? Sl 05:52 0:00 \ containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime
root 24339 0.0 0.0 18504 3224 pts/0 Ss+ 05:52 0:00 \ bash
root 3193 0.0 0.9 676900 80360 ? Ssl Dec15 1:02 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

그림 15 docker-shim

## CRI-O

CRI-O 는 쿠버네티스를 위한 경량화 컨테이너 런타임이다. CRI-O 는 처음부터 디자인이 쿠버네티스 위해서 구성이 되었다. 다른 컨테이너 환경과 다르게 계층이 매우 간단하다. 컨테이너 운영을 위한 기본 기능 위주로 "pod" 및 "container"를 빠르게 생성 및 관리가 주요 목적이다.

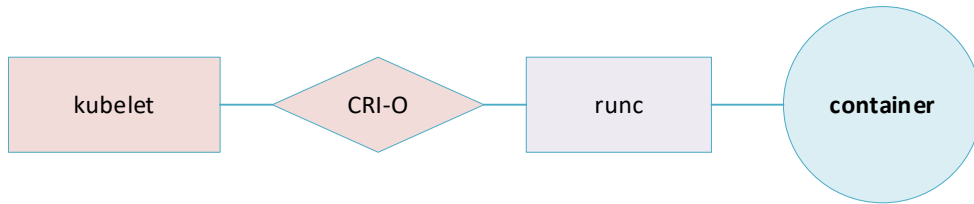


그림 16 CRI-O

대다수 엔터프라이즈 컨테이너 오케스트레이션 엔진들은 CRI-O 런타임을 사용한다. CRI-O의 약자는 Container Runtime Interface/Open Container Initiative의 앞 글자만 따와서 CRI-O라고 부른다. 현재 쿠버네티스는 CRI-O 기반으로 구성을 권장한다.

## Podman(포드만)

Podman 컨테이너 생성 및 관리하는 오케스트레이션 도구이다. 도커와 거의 동일한 기능 및 호환성을 제공한다. 포드만에서 사용하는 podman.service 컨테이너 런타임을 실행하기 위해서 사용하지 않는다. 이 서비스는 일반적으로 외부에서 API 기반으로 서비스 호출 및 구성을 하기 위해서 사용한다.

아래 명령어로 간단하게 API 테스트를 하기 위해서 아래처럼 명령어를 실행한다.

```

master]# podman system service -t 5000 &
master]# curl --unix-socket /run/podman/podman.sock -v
'http://d/v3.0.0/libpod/images/json' | jq
  
```

동작하는 구조는 아래와 같은 그림으로 동작하게 된다. OCI 사양을 따르며, 기본적으로 CRI-O와 동작 구조는 흡사하다.

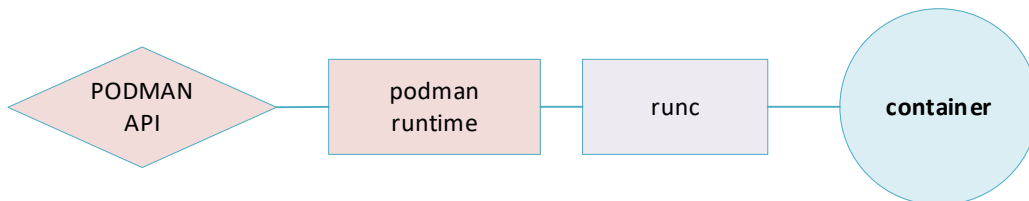


그림 17 podman 구조

다만, 포드만은 쿠버네티스 컨테이너 런타임 인터페이스로 사용은 불가능하다. 포드만 고수준 컨테이너 런타임 기능을 제공하기 때문에, 직접적으로 쿠버네티스에 적용이 불가능하지만, 컨테이너 이미지 빌드 및 POD 테스트가 가능하기 때문에, 올바르게 동작하면 바로 쿠버네티스에서 사용이 가능하도록 전환이 가능하다.

## 6. 리눅스 커널과 쿠버네티스 관계

쿠버네티스 자체적으로는 API 및 오케스트레이션 기능만 제공한다. 컨테이너 및 POD를 구성 및 구현하기 위해서는 커널에서 자원 격리 및 추적이 가능한 기능을 제공해야 한다. 대표적인

## THE NAMESPACE

네임스페이스는 커널에서 지원하는 기능이며, 이 기능은 애플리케이션이 실행 시, 시스템 계층과 사용자 계층에서 발생한 장애 혹은 예상되지 않는 접근을 막기 위해서 사용하는 기술이다.

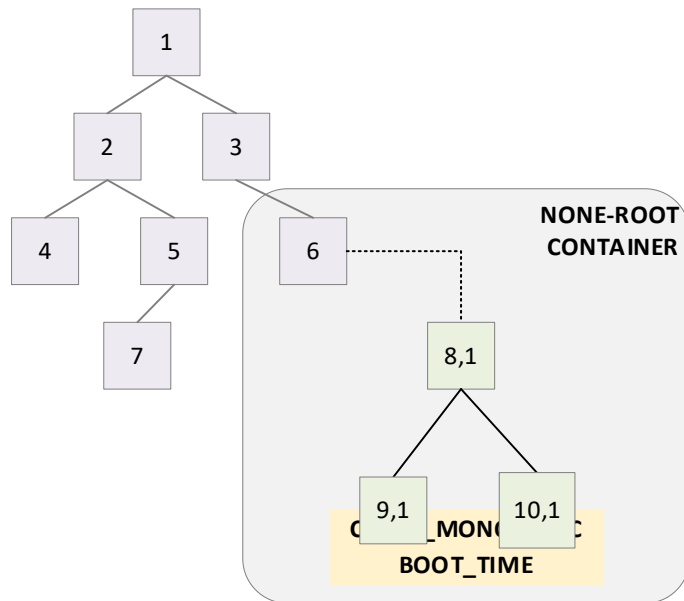


그림 18 namespace UTS clock

네임스페이스는 리눅스 커널 2002 년에 도입이 되었으며, 본격적으로 도입된 릴리즈 버전은 linux-2.4.19 이다. 현재 리눅스 배포판이 사용하고 있는 리눅스 커널의 네임스페이스는 처음에 도입이 되었던 네임스페이스보다 더 효율적인 구조로 변경이 되었다. 이전 네임스페이스는 다음과 같은 단점이 있었다.

1. 모든 네임스페이스는 시스템의 CLOCK\_REAL\_TIME 를 가지고 동작.
2. 하위 네임스페이스는 시스템 시간과 동기화를 해야 되기 때문에 작업 부하가 높음.
3. CLOCK\_REAL\_TIME 에서 생성된 컨테이너는 독립된 시간을 갖지 못함.
4. 컨테이너가 다른 노드로 이전 시, 기존에 사용했던 시간을 사용이 불가능.

위와 같은 이유로 초기에는 컨테이너 컴퓨트 노드에서 많은 개수의 컨테이너를 동시 생성을 하면, 커널에서 많은 무리가 발생하였지만, 각각 네임스페이스별로 시간을 제어 및 조정이 가능하기 때문에 빠르게 구성이 가능하다. 하지만, 2018 UTS 기능을 지원하는 네임스페이스가 리눅스 커널 5.6 에 도입이 되었으며, 레드햇 리눅스는 linux-4.18.0 버전부터 새로운 네임스페이스 기술을 지원한다.

## C-GROUP

c-group 은 Control Group 의 약자이며, 실행되는 프로세스에 세밀하게 자원 사용량 제한 및 감사 기능을 제공한다. 2006 년에 구글에서 개발이 시작이 되었으며, 개발명은 "process containers"라는 이름으로 개발, 2007 년에 이름을 **cgroup** 으로 변경하였다. 정식 릴리즈는 2008 년도에 하였으며, 리눅스 커널 버전은 linux-2.6.24 에서 공식적으로 릴리즈가 되었다. cgroup 의 주요 기능은 다음과 같다.

1. 자원제한(resource limiting)
2. 우선순위(Prioritization)

### 3. 할당(Accounting)

### 4. 제어(Control)

좀더 자세히 하기전에, cgroup 기반으로 컨테이너를 구현하기 위해서, 한가지 기술이 더 필요하다. 그 기술은 위에서 이야기한 네임스페이스(NAMESPACE)이다. 위의 주요기능에 "자원제한", "할당", "제어" 부분을 구현하기 위해서 실행한 프로세스는 반드시 시스템과 분리가 되어서 실행 및 관리가 되어야 한다. 네임스페이스에서 동작하는 프로세스는 cgroup 를 통해서 CPU 및 메모리에 대한 제어가 필요하다.

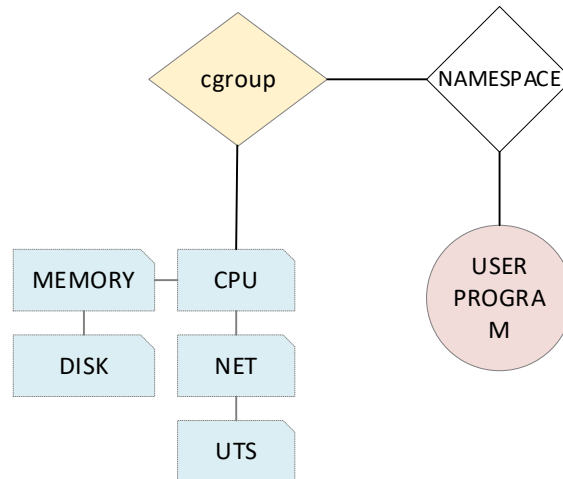


그림 19 네임스페이스

"c-group"를 사용하면 자원은 커널 영역에서 관리하게 되며, 실행되는 프로그램은 네임스페이스를 통해서 분리 및 격리가 되면서 기존에 사용중인 사용자 프로세스에 영향이 없도록 분리가 되면서 실행이 된다. 실제로 컨테이너가 생성이 되면, 컨테이너에서 실행이 된 프로세스는 cgroup 과 네임스페이스를 통해서 동작이 된다.

## 7. 파일시스템

### UFS

UFS<sup>16</sup> (Union File System), OverlayFS(File System)은 도커에서 사용하였던 이미지 파일 시스템 및 레이어이다. 이 파일 시스템은 실제로 존재하지 않지만, 파일이나 디렉토리를 마치 레이어처럼 만들어서 구성해준다. 유니온 파일 시스템은 리눅스에서 많이 사용하는 xfs, btrfs, ext4 그리고 devicemapper 에서 사용이 가능하다. 기본적으로 컨테이너 이미지는 UFS 는 형태로 저장이 되며, 이 구조는 key=pair 형태로 구성이 되어 있다.

<sup>16</sup> <https://unionfs.filesystems.org/docs/zen/zen.html>



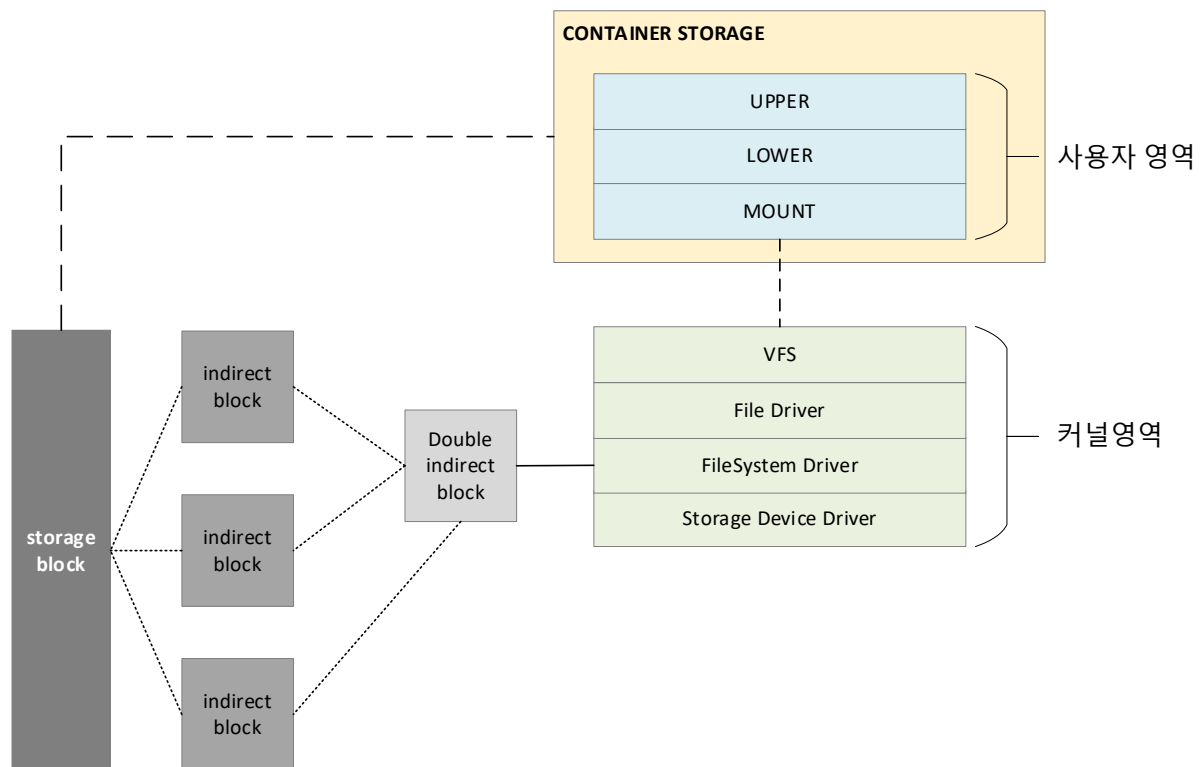


그림 20 UFS 레이어

## OverlayFS (Filesystem)

OverlayFS<sup>17</sup>(오버레이 파일 시스템)은 UFS 의 개념과 비슷하며, 강화된 기능이다. 제일 큰 차이점은 블록장치나 혹은 아이노드(inode)기반으로 구성이 아니라, 파일 및 디렉터리를 읽기전용 파티션 형태로 만들어서 컨테이너에게 전달한다.

리눅스 커널 3.18 에서 도입이 되었으며, 리눅스 커널 4.0 에서 본격적으로 overlay2 를 통해서 도커 및 다른 컨테이너 엔진에서 사용한다. 오버레이 파일 시스템의 제일 큰 장점은, upper 이외 나머지는 읽기 전용으로만 구성이 되기 때문에 기존 이미지를 재활용이 가능하다.

이를 명령어로 구현하면 다음과 같이 명령어로 구성이 가능하다.

```
# mount -t overlay overlay -o
lowerdir=/lower,upperdir=/upper,workdir=/work /merged
```

위의 명령어를 복잡하게 사용하면, 아래와 같은 다이어그램으로 레이어 계층이 생성이 된다. 생성된 오버레이 장치는 특정 디렉터리 위치에서 계속 겹쳐서 구성이 되며, 이를 backingblockdev<sup>18</sup>를 통해서 단일 장치처럼 사용자에게 구성 및 생성이 되어 전달이 된다.

<sup>17</sup> [https://wiki.lustre.org/images/8/8c/LUG2019-Lustre\\_2.12\\_In\\_Production-Thiell.pdf](https://wiki.lustre.org/images/8/8c/LUG2019-Lustre_2.12_In_Production-Thiell.pdf)

<sup>18</sup> <https://github.com/BCDevOps/backup-container>

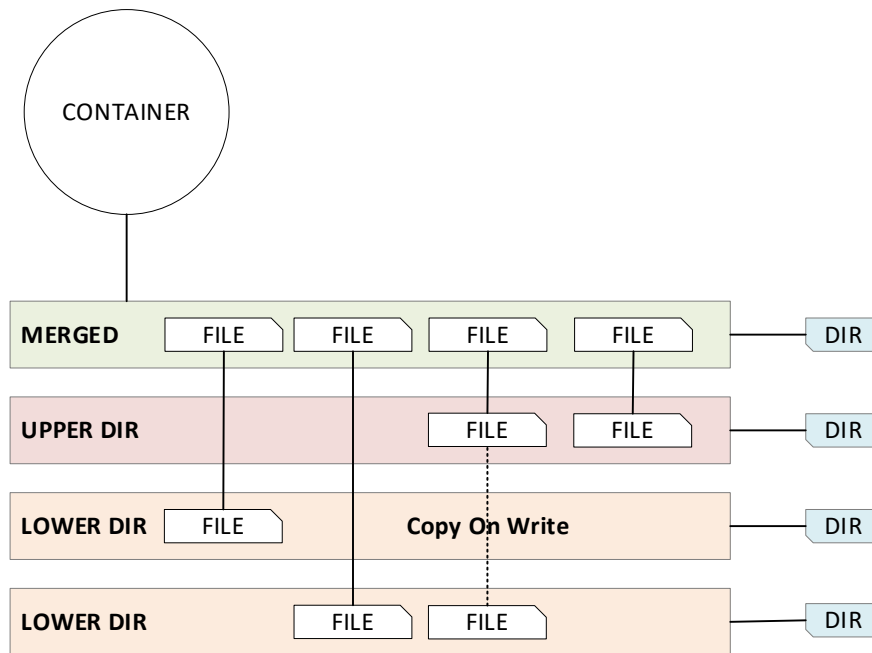


그림 21 OverlayFS 계층

현재 거의 대다수 컨테이너 시스템은 위와 같은 구조로 컨테이너 이미지를 구현하고 있다. UFS 경우에는 논리적 구성을 하드코딩 된 드라이버를 통해서 **유저/커널** 영역에서 장치를 구현한다. 이로써 완벽하게 드라이버를 구현한다는 장점이 있지만, 단점으로는 커널 영역을 사용하기 때문에 성능 부분에서는 OverlayFS 보다 느리다. 현재 거의 대다수 컨테이너의 파일 시스템은 OverlayFS 기반으로 사용하고 있고, "Fuse"기반으로 구성된 "Fuse OverlayFS"<sup>19</sup>를 더 많이 사용한다.

# 쿠버네티스 아키텍처

## 1. 가상화 vs 컨테이너

컨테이너와 가상머신의 제일 큰 차이점에 대해서 아래 그림을 참고한다. 아래 그림은 하이퍼바이저 type 1,2 아키텍처이다.

<sup>19</sup> <https://github.com/containers/fuse-overlayfs>

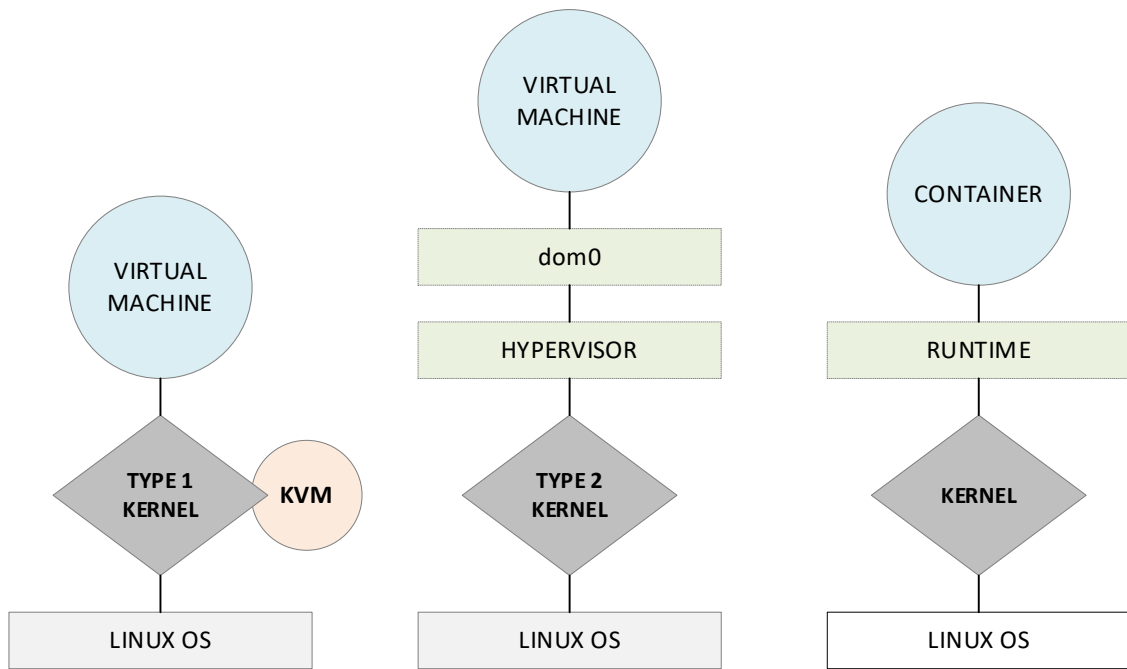


그림 22 컨테이너 및 하이퍼바이저 비교

가상머신과 컨테이너의 제일 큰 차이점은 **하드웨어의 가상화 기능사용** 여부이다. 기본적으로 쿠버네티스는 **컨테이너 런타임 기반**의 환경을 제공하기 때문에 하드웨어 기반의 가상화처럼 높은 하드웨어를 요구하지 않는다. 하지만, 컨테이너는 rootless 구조 및 ring structure 를 생성하지 않고 시스템 구조를 구현하기 때문에 가상머신과 비교하였을 때 기능적으로 제한이 있다. 그 부분은 아래에서 표로 정리하였다.

항목	가상화	컨테이너
드라이버(커널수준)	자체 구현 가능	불가능
드라이버(사용자 수준)	자체 구현 가능	공유(간접)
링 구현	물리적 장비와 동일	불가능, 공유

최근 쿠버네티스는 "kube-virt"라는 프로젝트를 시작 하였으며, POD 기반으로 가상화 기능을 제공하고 있기 때문에 컨테이너와 같이 가상 머신을 같이 사용이 가능하다. 이 부분은 이 책에서 다루지 않는다.

## 2. RUNC/CRUN

RUNC 는 기본적으로 모든 컨테이너에서 사용하는 기본 환경 제공자이다. 사용자가 생성한 컨테이너는 기본적으로 runc 기반으로 컨테이너가 생성이 되며, 생성된 컨테이너는 systemd 에서 cgroup 및 namespace 를 관리 및 추적을 한다.

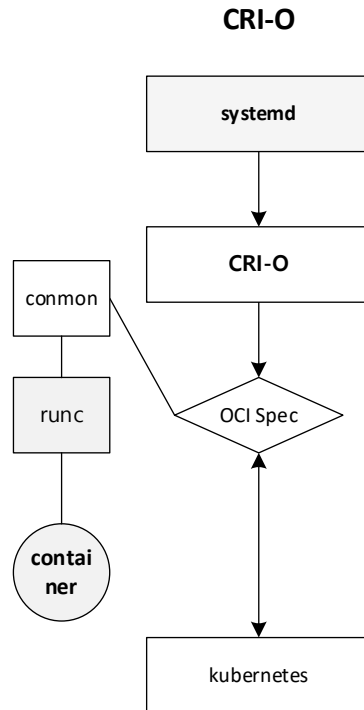


그림 23 RUNC(common)

생성된 컨테이너는 conmon<sup>20</sup>이라는 프로세서가 컨테이너를 모니터링을 한다. 컨테이너는 conmon에서 "-r"이라는 런타임 옵션을 통해서 'runc'기반으로 컨테이너 이미지를 불러와서 실행한다.

## 3. 쿠버네티스 컴포넌트

쿠버네티스는 여러 개의 도구를 기반으로 구성이 되어 있다. 이전의 쿠버네티스는 일반적인 서비스처럼 호스트 컴퓨터에 설치가 되었지만, 현재 쿠버네티스는 kubelet이라는 서비스를 제외하고 나머지는 전부 컨테이너 기반으로 동작한다.

쿠버네티스는 다음과 같은 프로그램으로 서비스 구성이 되어 있다.

- kubelet(hosted)
- kube-proxy(containerd)
- kube-scheduler(containerd)
- kube-controller-manager(containerd)
- kube-apiserver(containerd)

---

<sup>20</sup> container monitor 의 약자

위의 구성 서비스에 대해서는 아래에서 더 자세히 설명한다. 또한 쿠버네티스에서 사용하는 소프트웨어 구성 요소는 다음과 같다.

- pod(pause)18F21
- node
- coredns
- etcd

## kubelet

유일하게 호스트에 설치되는 프로그램이다. 외부에서 들어오는 요청을 컨테이너 기반으로 구성된 쿠버네티스에 전달하기 위해 kubelet 서비스를 사용한다. 이 서비스는 두 가지 목적을 가지고 있다.

1. 쿠버네티스에서 사용하는 컨테이너 서비스 시작
2. 외부에서 들어오는 요청에 대한 서비스 처리

kubelet 를 사용하기 위한 서비스 포트 번호는 아래와 같다. 특히 마스터 노드에서는 아래 포트가 활성화가 되어 있어야 한다.

- TCP/6443
- HTTP/HTTPS

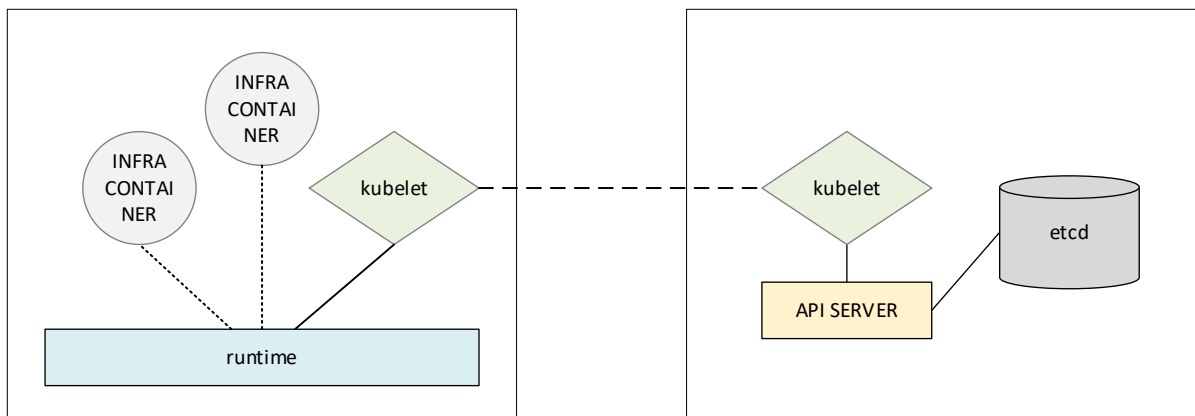


그림 24 kubelet 구조

포트는 6443 포트를 사용하며, 이 포트로 들어오는 요청은 http, https로 처리가 된다. 6443 포트로 들어온 트래픽 혹은 요청은 kube-apiserver 라는 컨테이너로 전달이 된다. 아래 명령어로 kubelet 서비스 확인이 가능하다.

```
master/node]# ss -antp | grep kubelet
127.0.0.1:10248    *:*          users:(("kubelet",pid=519,fd=33))
127.0.0.1:39389    *:*          users:(("kubelet",pid=519,fd=10))
192.168.122.110:6443 users:(("kubelet",pid=519,fd=16))
```

<sup>21</sup> 바닐라 버전의 쿠버네티스는 Pause 라는 Pod 프로그램을 사용한다.

kubelet 는 자원의 **생성/삭제/조회**와 같은 기능을 제공하며, master, node 상관 없이 해당 데몬은 실행이 되고 있다. 정확히는 쿠버네티스에서 생성 및 관리하는 데몬에 대해서 관여하며, 그 이외 데몬에 대해서는 관여하지 않는다.

이름	호스트	컨테이너
kubelet	네	아니요
kubeproxy	아니오	네

## POD/Pause

쿠버네티스는 pause 라고 부르는 컨테이너가 있다. 이 컨테이너는 애플리케이션을 실행 시 1:N 혹은 1:1 형태로 구성이 된다. 포드는 기본적으로 애플리케이션에 사용하는 자원 즉, 네트워크, 스토리지 같은 자원들을 pause 컨테이너를 통해서 제공이 되며, 이를 포드(pod)라고 부른다.

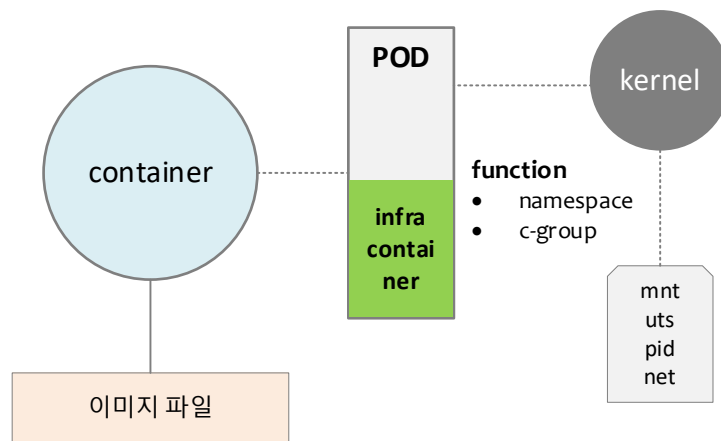


그림 25 POD

## etcd

etcd 쿠버네티스에서 발생하는 데이터를 JSON 기반으로 저장하는 키-기반 기반의 데이터베이스이다. etcd 는 주요 목적은 빠르게 읽기/쓰기를 위해서 만들어진 도구이며, RDBMS 처럼 트랜잭션 대기가 발생하지 않는다. 기본적인 기능 자체가 분산형 기반으로 구성이 되어 있기 때문에 클러스터 설정을 멤버 기반으로 매우 쉽게 가능하다.

이러한 이유로 쿠버네티스는 etcd 를 사용하며, 쿠버네티스는 etcd 오퍼레이터를 통해서 클러스터에서 발생한 데이터 및 이벤트를 처리한다.

- 생성/제거:** etcd 에 발생되는 설정을 생성 및 제거를 합니다. 멤버 추가 및 설정 부분은 클러스터의 크기만 사용자가 명시해주면 됩니다.
- 백업:** etcd 오퍼레이터는 백업을 자동으로 수행합니다. 30 분마다 백업하고 마지막 3 번의 백업 보관을 합니다.
- 업그레이드:** 중지시간(downtime)없이 etcd 업그레이드가 가능합니다. etcd 오퍼레이터는 운영 시 업그레이드가 가능하도록 합니다.
- 크기조정:** 생성/제거처럼, 사용자가 크기만 명시만 해주면 자동적으로 클러스터의 멤버 배포 및 제거 그리고 설정을 자동으로 합니다.

etcd 는 단일 혹은 다중으로 동작한다. 다중으로 구성하기 위해서는 최소 3 개의 etcd 서비스가 구성이 되어야 한다.  
etcd 는 특정 노드를 etcd 멤버에 추가하면 자동적으로 클러스터를 설정 및 구성한다.

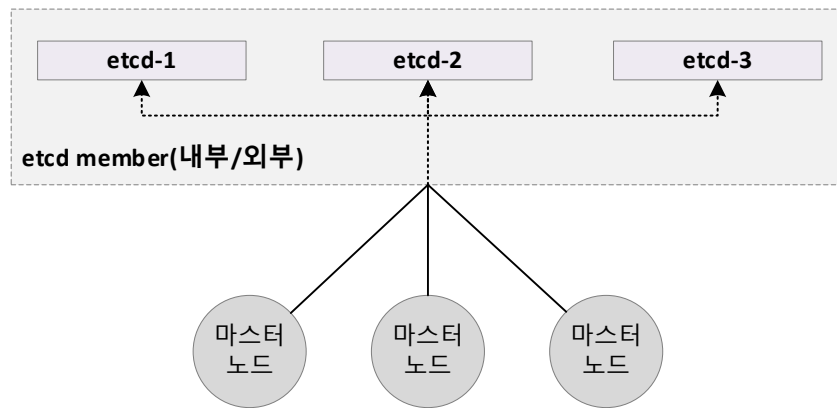


그림 26 클러스터 멤버

이전에는 외부에 etcd 가 구성이 되어 있어서 쉽게 확인이 가능했으나, 지금은 etcd 서버가 컨테이너 서비스 내부로 들어가서 아래와 같은 명령어로 확인이 가능하.

```

master]# ADVERTISE_URL=https://192.168.90.110:2379
master]# kubectl exec etcd-master.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

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

```

위의 명령어를 실행하면 아래와 같이 etcd 내용이 조회가 되어서 화면에 출력이 된다. 출력되는 내용들은 쿠버네티스 추가되는 서비스 및 운영 시 생성한 자원들에 대한 디렉터리 서비스 목록이 출력이 된다.

```

/registry/minions/master.example.com
/registry/minions/node1.example.com
/registry/minions/node2.example.com
/registry/namespaces/default
/registry/namespaces/kube-node-lease
/registry/namespaces/kube-public
/registry/namespaces/kube-system
/registry/pods/kube-system/coredns-787d49

```

## coredns

coredns 는 포드 및 서비스에서 사용하는 내부 도메인 작업을 처리하는데 사용한다. 구성 방법에 따라서 다르지만, 외부 도메인 및 내부 서비스를 도메인 기반으로 처리시 사용한다. 좀 더 자세한 내용을 알고 싶은 경우 coredns <sup>22</sup>웹 사이트에서 추가적인 정보 확인이 가능하다.

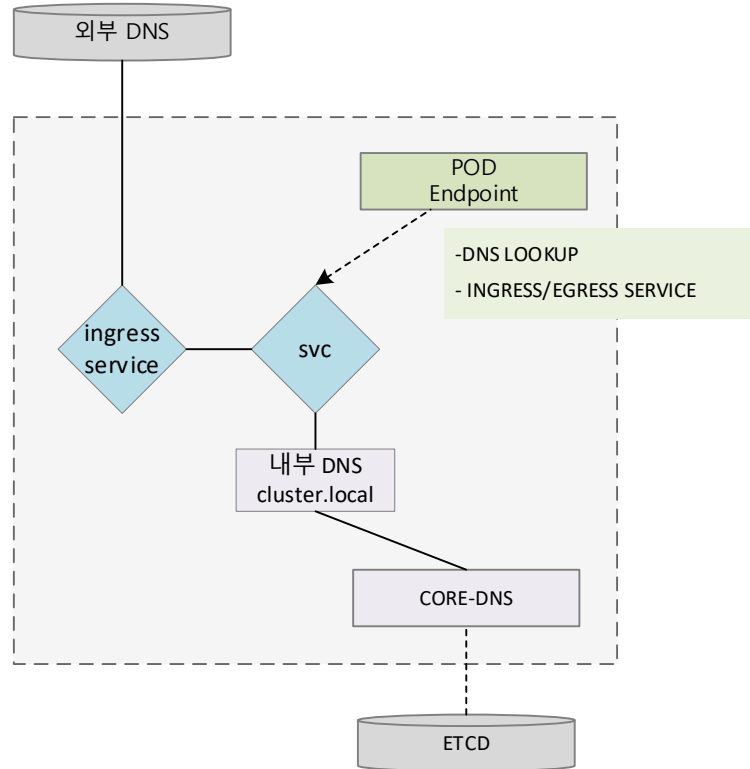


그림 27 coredns

coredns 는 기본적으로 내부 클러스터 서비스를 위해서 사용하며, 여러 서비스를 하나로 묶기 위한 방법으로 사용한다. 설치시 별도로 도메인을 설정하지 않았으면, cluster.local 로 되어있다. 추후에 확인이 가능하지만, coredns 도 일반 도메인 서비스 프로그램과 동일하게 존 파일(zone file)기반으로 도메인을 구성 및 설정한다. 형식은 아래와 같이 쿠버네티스에서 내부적으로 작성한다.

실제 내용은 아래처럼 구성이 되어 있

```
$ORIGIN example.org.

@      3600 IN      SOA sns.dns.icann.org. noc.dns.icann.org. 2017042745
7200 3600 1209600 3600

      3600 IN NS a.iana-servers.net.
      3600 IN NS b.iana-servers.net.

www    IN  A      127.0.0.1
      IN  AAAA   ::1
```

<sup>22</sup> <https://coredns.io/>



쿠버네티스 내부에서 사용하는 coredns 를 확인하기 위해서 다음과 같은 명령어로 간단하게 내부 조회가 가능하다.

```
master]# kubectl run -i -t --image=busybox
```

위의 명령어로 내부 서버를 조회하면 보통 다음처럼 메시지가 출력이 된다. 단, 기본 클러스터 도메인을 변경하지 않았다는 조건이다. 위에서 말했지만, 기본 도메인은 cluster.local 이다.

```
search default.svc.cluster.local svc.cluster.local cluster.local
google.internal c.gce_project_id.internal

nameserver 10.0.0.10

options ndots:5
```

Coredns 설정 내용은 아래와 같은 명령어로 설정 파일 내용 확인이 가능하다.

```
master]# kubectl -n kube-system describe configmap coredns
```

## kube-proxy(master, worker)

kube-proxy<sup>23</sup> 서비스는 쿠버네티스에 구성이 되어있는 애플리케이션 서비스에 접근할 수 있도록 지원하는 구성원. 주요 목적은 쿠버네티스에 구성된 자원들의 네트워크 생성 및 관리, 그 이외에 리소스에 대해서 모니터링한다.

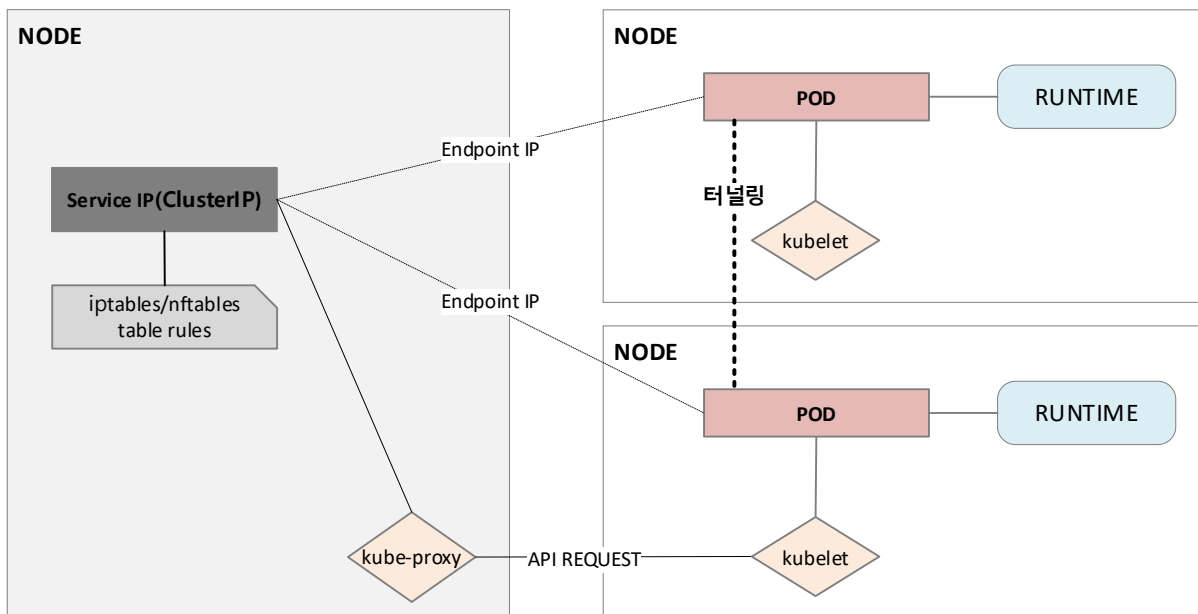


그림 28 kube-proxy

먼저, kubelet 서비스가 올바르게 동작하는지 확인 후, crictl 명령어로 kube-proxy 컨테이너가 올바르게 동작하는지 확인한다.

<sup>23</sup> <https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/>

```

master/node]# systemctl status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/usr/lib/systemd/system/kubelet.service; enabled;
   vendor preset: disabled)
   Drop-In: /usr/lib/systemd/system/kubelet.service.d
            └─10-kubeadm.conf
   Active: active (running) since Sun 2022-06-05 23:48:16 KST; 54min ago

node]# crictl ps
kube-proxy

```

kube-proxy가 구성한 네트워크 정보를 확인하려면, 노드에서 간단하게 nft 명령어로 확인이 가능하다.

```

master/node]# nft list table filter
table ip filter {
    chain INPUT {
        type filter hook input priority filter; policy accept;
        ct state new counter packets 48 bytes 5218 jump KUBE-
PROXY-FIREWALL
        counter packets 48133 bytes 83155892 jump KUBE-NODEPORTS
    }
}
...SS
kubernetes cluster.local in-addr.arpa ip6.arpa {
    pods insecure
    fallthrough in-addr.arpa ip6.arpa
    ttl 30
}
...

```

## kube-scheduler

쿠버네티스 스케줄러는 사용자가 요청한 작업이나 혹은 시스템에서 예약된 작업이 호출이 되었을 때, 스케줄러를 통해서 클러스터에 존재하는 노드의 조건을 확인하여 요청된 작업을 생성한다.

포드 구성 시 쿠버네티스 스케줄러는 다음 조건을 먼저 확인한다.

- 필터링(filtering)
- 점수(scoring)

스케줄러는 먼저 노드에 사용이 가능한 자원이 얼마나 있는지 확인을 하며, 자원이 충분한 노드를 찾으면 해당 노드에 자원 생성을 요청한다. 하지만, 자원이 충분하지 않으면 스케줄러는 해당 노드에 더 이상 자원 생성 요청을 하지 않는다.

그 다음 단계는 바로 점수 부분이다. 스케줄러가 포드 구성이 가능한 노드를 찾으면, 필터링을 통해서 점수화를 하여 가장 점수가 높은 쪽으로 자원을 생성한다. 만약, 한 개 이상의 노드가 동일한 점수로 결과가 나오면, 쿠버네티스는 무작위로 노드를 선택해서 자원을 구성한다. 최종적으로 단계를 검증하면 다음과 같다.

## 스케줄링 정책

필터링을 통해서 각 노드별로 점수를 매기며, 이를 통해서 어떤 노드에 자원을 구성할지 결정한다.

## 스케줄링 프로파일(scheduling profile)

프로파일을 통해서 좀 더 추가적인 기능을 구성할 수 있는데, 스케줄러 플러그인을 통해서 kube-scheduler 에 다양한 프로파일을 구성할 수 있다. 이 프로파일은 Queue Sort, Filter, Score, Bind, Reserve, Permit 플러그인 혹은 사용자가 추가적으로 구성이 가능하다.

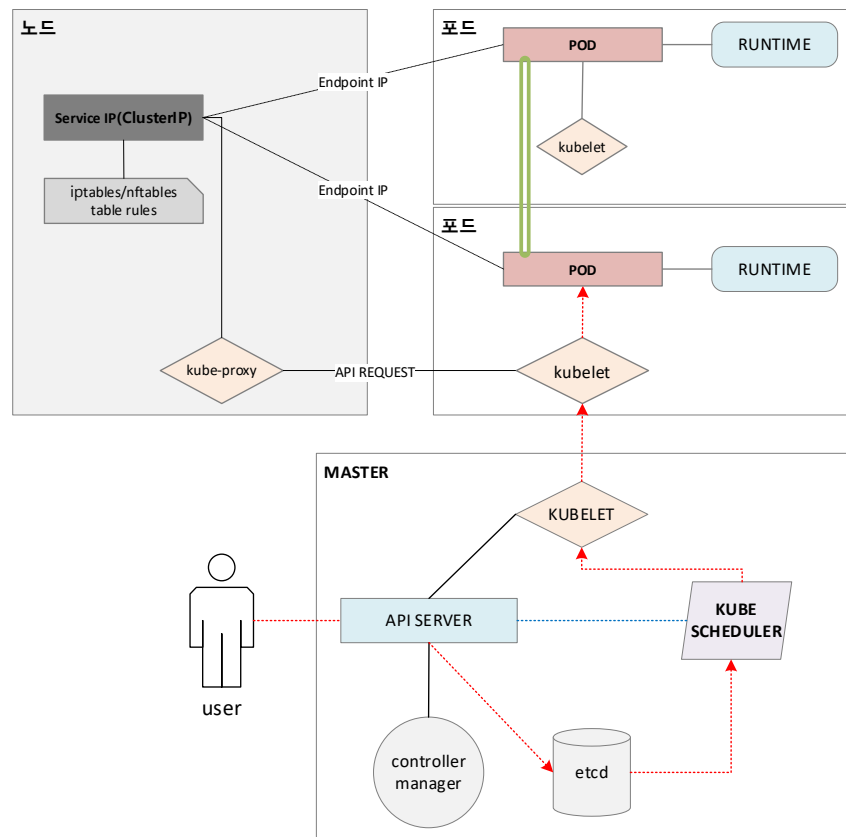


그림 29 스케줄러

## kube-controller-manager

컨트롤러 매니저는 스케줄러와 비슷하게 보이지만, 조금 다른 역할을 한다. 이 구성원은 쿠버네티스에 구성되어 있는 주요 핵심 자원들이 반복적으로 동작하면서 클러스터의 상태 및 특정 작업들을 반복적으로 수행할 수 있도록 한다.

예를 들어서 클러스터에서 공유하고 있는 데이터에 대해서 지속적인 갱신, apiserver 를 통해서 상태 확인 혹은 변경이다. kube-controller-manager 를 대표적으로 많이 사용하는 서비스는 replication controller, endpoint controller, namespace controller 그리고 serviceaccount controller 이다. 이 서비스 기반으로 이중화 같은 서비스를 구성하기도 한다.

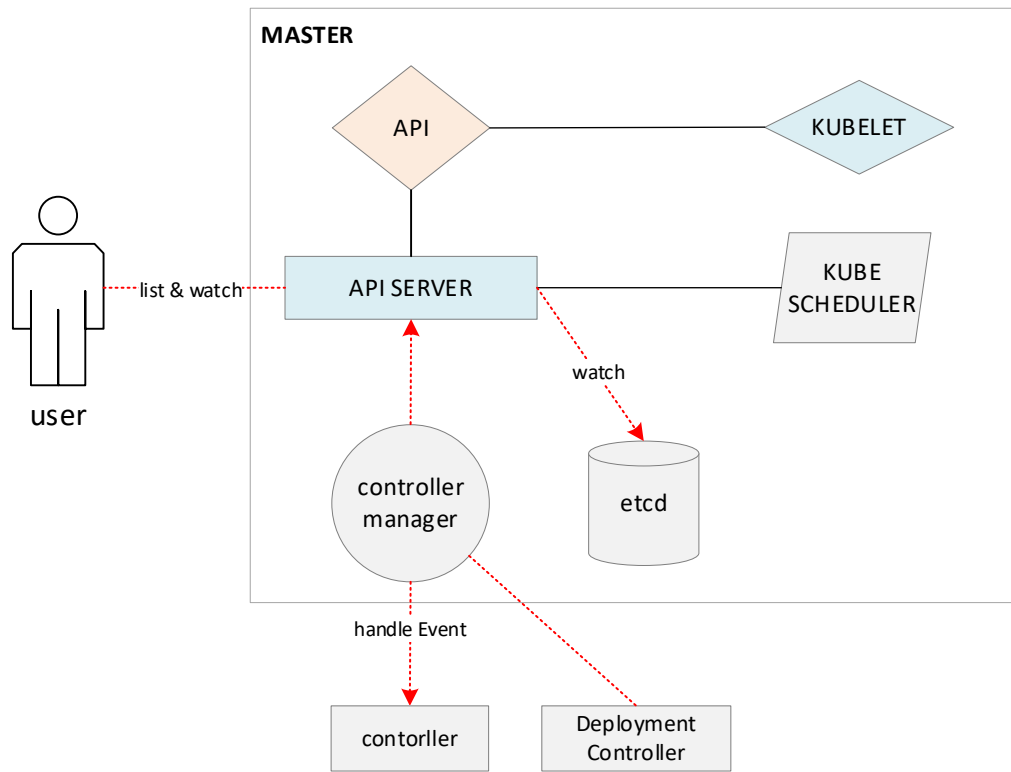


그림 30 컨트롤 매니저

작업 생성 요청이 API 를 통해서 들어오면, 쿠버네티스는 다음과 같은 과정을 통해서 작업을 처리 및 수행한다.

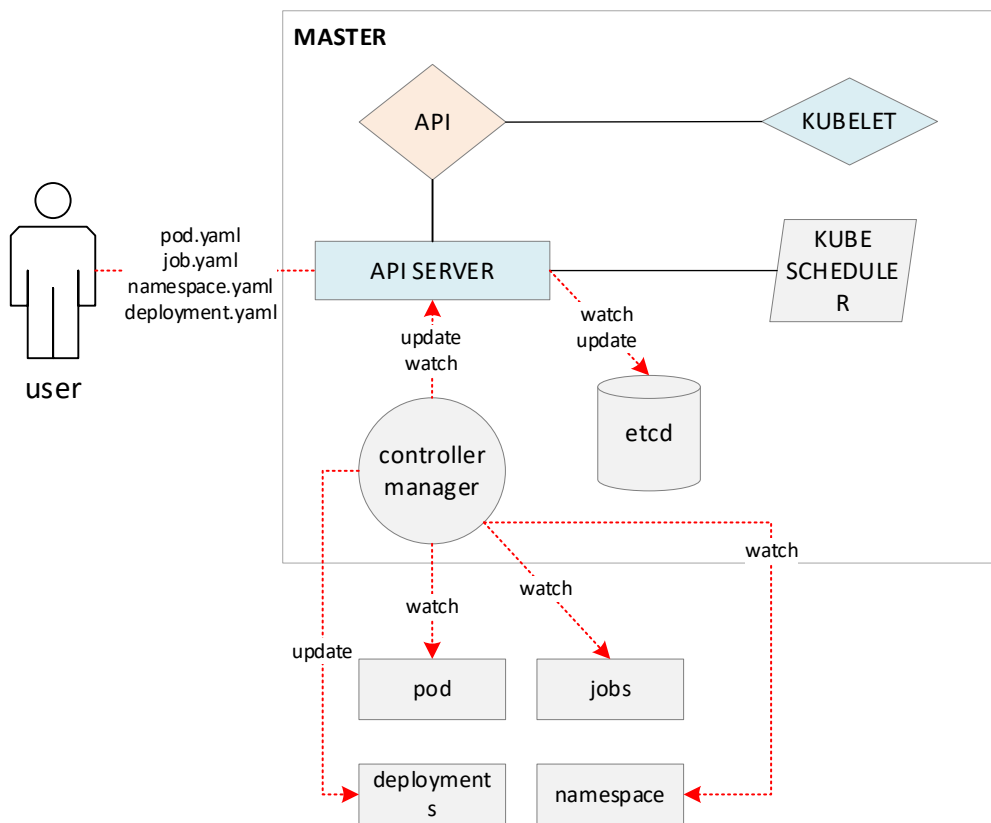


그림 31 자원 생성

컨트롤 매니저는 들어오는 요청 내용에 따라서 정보 갱신 및 모니터링을 한다. 자원 요청은 데이터베이스와 유사하게 CRUD<sup>24</sup>형태로 이루어진다.

## kube-apiserver

외부에서 들어오는 요청은 API 서버를 통해서 자원 처리를 한다. kubectl 이나 혹은 API 를 통해서 들어온 자원들은 kubelet 를 통해서 노드로 전달이 된다. 각각 자원은 컨트롤 매니저를 통해서 Pod 같은 자원을 처리하며, 수행이 된 작업은 etcd 서버에 상태 정보를 저장한다.

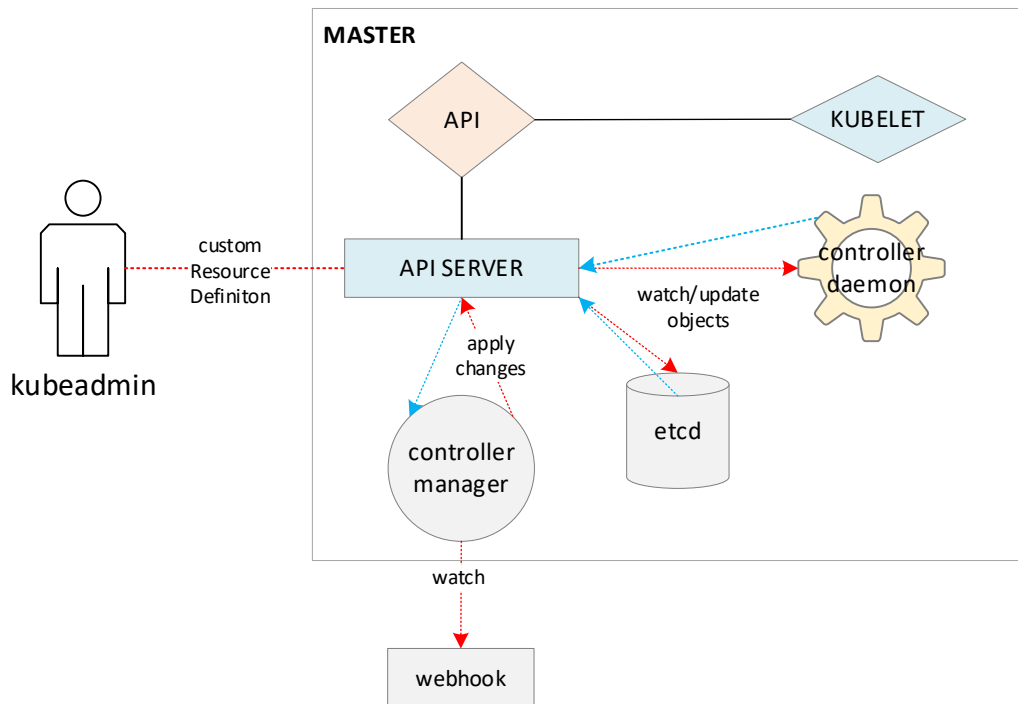


그림 32 apiserver

<sup>24</sup><https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/>

## 4. 마스터 노드

마스터 노드는 쿠버네티스 클러스터에 최소 한 개가 필요하다. 클러스터에서 요청된 작업 마스터 노드를 통해서 작업이 처리가 된다. 모든 요청은 API 를 통해서 들어오게 되며, 이 요청은 kube-apiserver 를 통해서 처리하게 된다. 아래 그림은 마스터 서버에 기본적으로 구성이 되어 있는 서비스이다.

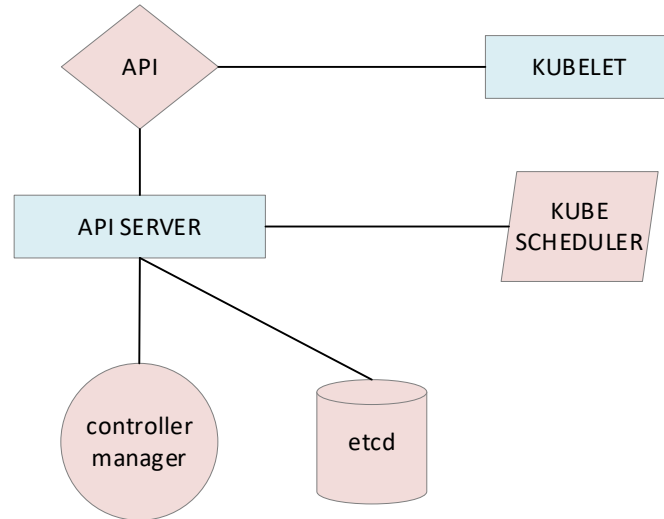


그림 33 마스터 노드

## 5. 워커노드(minion, compute)

워커 노드는 쿠버네티스 클러스터에서 포드 및 애플리케이션을 실행하는 환경을 제공한다. 물리적 장비나 혹은 가상장비에 다음과 같은 구조로 애플리케이션을 구성한다.

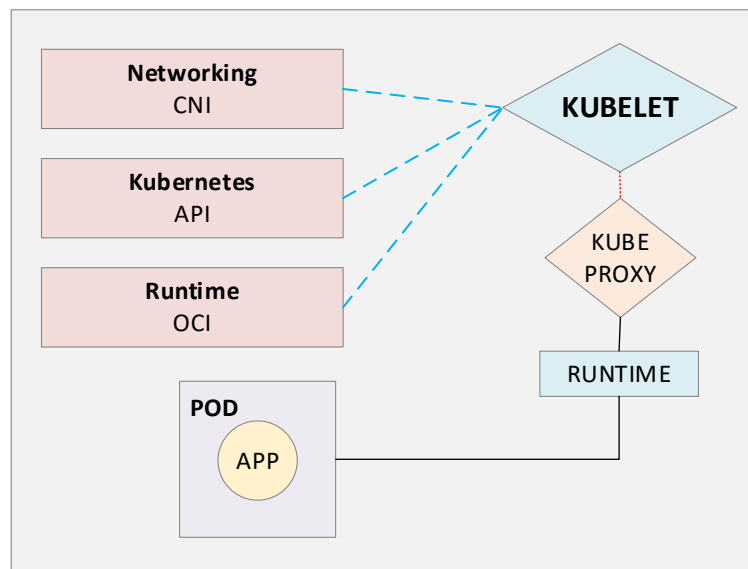


그림 34 워커노드

쿠버네티스 노드는 가상머신의 하이퍼바이저 같은 역할을 한다. 하지만, 위에서 이야기하였지만 별도의 하드웨어 기술이 요구가 되지 않기 때문에 하이퍼바이저 보다 낮은 비용으로 구성이 가능하다.

애플리케이션이 실행이 되면, 애플리케이션은 워커 노드에서 실행이 되며, 노드는 최소 한개의 마스터 노드를 통해서 클러스터를 구성한다.

## 스왑 사용

시스템에서 페이지 혹은 스왑을 사용하는 경우, 컨테이너는 실제 메모리 크기를 제한을 해야 하는데, 메모리의 내용을 디스크에 페이지 하는 경우 제한하려는 크기보다 더 많은 자원을 사용하는 경우가 있다. 또한, 페이지로 인한 CPU 사용율 증가로 실제 컨테이너 운영에 도움이 되지 않는다.

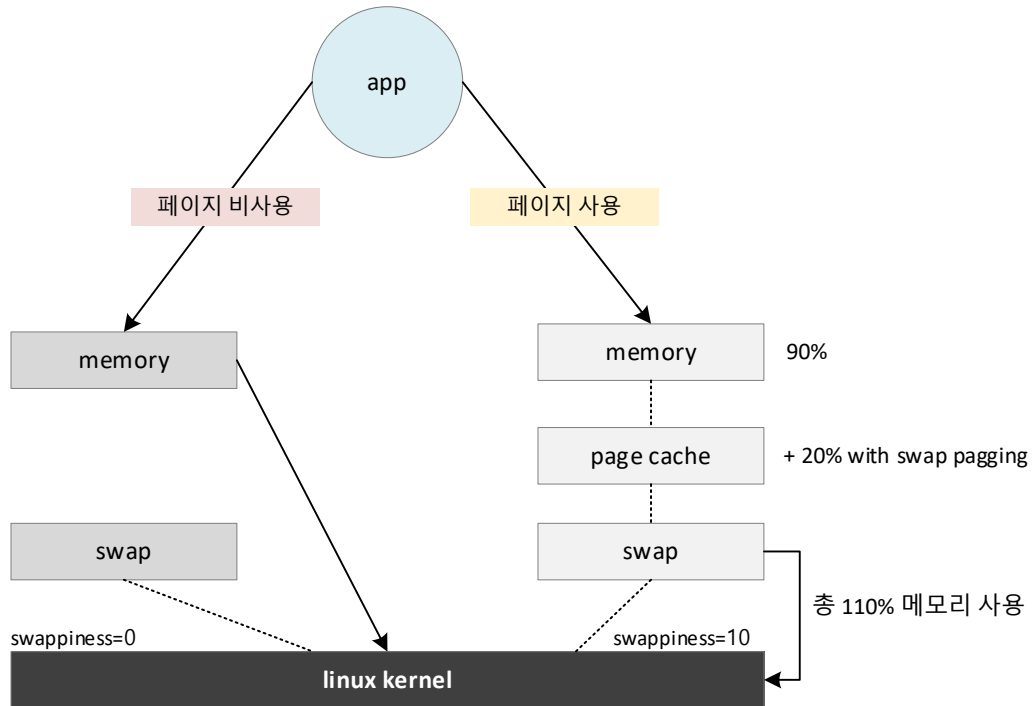


그림 35 스왑

이러한 이유로 반드시 쿠버네티스나 혹은 컨테이너 오케스트레이션 플랫폼에서는 스왑이나 혹은 페이지 기능을 끄는 것을 매우 강하게 권장한다.

```
master/node]# swapoff -a
master/node]# sysctl vm.swappiness=0
```

스왑이 동작중이면, kubeadm 에서 오류가 발생하면서 설치 진행이 되지 않는다.

# 주요 쿠버네티스 자원설명

## 6. Pod(po)

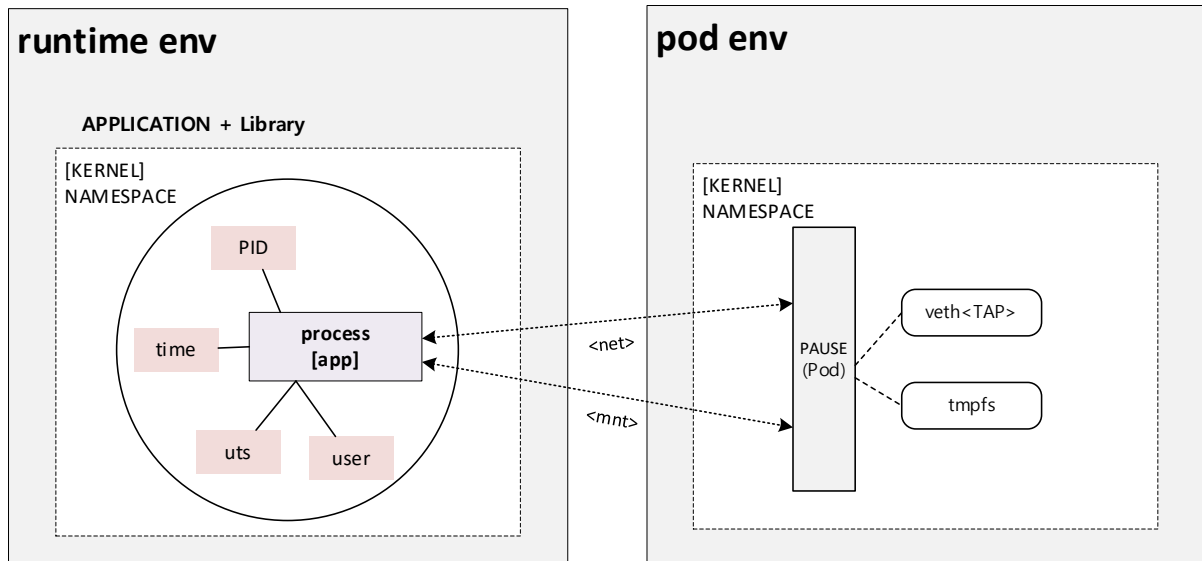


그림 36 POD

쿠버네티스에서 사용하는 모든 자원은 기본적으로 POD 기반으로 동작한다. Pod 의 주요 목적은 컨테이너에서 동작하는 애플리케이션에 대해서 격리 기능을 제공하며 이는 리눅스 커널의 몇몇 기능을 통해서 구현하게 된다.

Pod 는 컨테이너와 마찬가지로 하나의 애플리케이션이며, 이를 보통 인프라 컨테이너라고 부르기도 한다. 쿠버네티스의 Pod 는 기본적으로 Pause 라는 인프라 애플리케이션을 사용하며, OpenShift, Rancher 혹은 Podman 같은 컨테이너 런타임 혹은 오케스트레이션 도구들은 각기 다른 Pod 를 사용한다.

그래서 어떤 오케스트레이션 도구를 사용하느냐 혹은 런타임을 사용하느냐 따라서 Pod 구성이 달라지기 때문에, 반드시 사용하는 컨테이너 미들웨어가 어떤 Pod 를 사용하는 확인이 꼭 필요하다.

Pod 는 생성이 되면 Pod IP 및 도메인을 하나 생성한다.

```
pod-ip-address.my-namespace.pod.cluster-domain.example.
```

만약, 기본 네임스페이스인 default 에서 Pod 를 생성하면 다음과 같은 방식으로 Pod 도메인을 생성한다. Pod 아이피는 10.10.0.4 라고 한다.

```
10-10-0-4.default.pod.cluster.local.
```

위와 같은 형식으로 CoreDNS 에 Pod 관련된 도메인을 생성하여, 매번 자원을 네임스페이스에서 조회할 필요 없이, 바로 도메인 서비스를 통해서 접근 및 아이피 확인이 가능하다.



## 7. Service(svc)

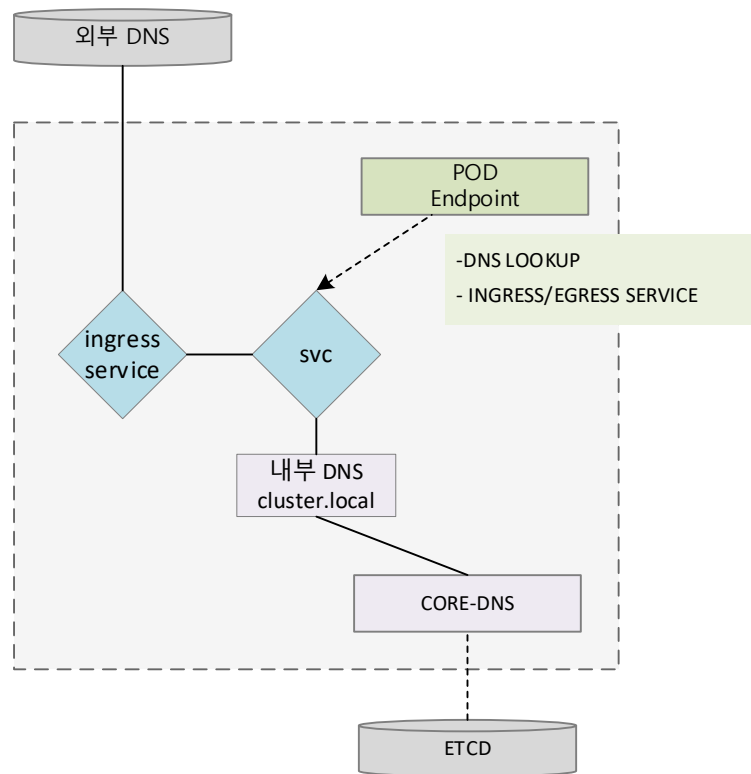


그림 37 POD 및 네트워크

서비스는 쿠버네티스에서 네트워크 및 도메인을 관리하는 영역이다. 서비스 영역은 여러 Pod 의 아이피를 하나의 서비스로 묶어주며, 이를 보통 클러스터 아이피(ClusterIP)라고 부른다.

클러스터 아이피가 구성이 되면, 이를 매번 확인이 어렵기 때문에, 서비스를 좀 더 손쉽게 사용하기 위해서 CoreDNS 기반으로 네임스페이스별로 도메인을 생성한다.

컨테이너가 생성이 되어서 동작하게 되면, 컨테이너 내부의 /etc/resolve.conf 는 다음과 같은 내용을 가지고 있다.

```
nameserver 10.32.0.10
search <namespace>.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

Pod 에서 구성된 A 레코드 및 아이피는 다음과 같은 형식으로 서비스 도메인이 CoreDNS 에서 구성 및 생성이 된다.

```
10-10-0-4.svc-pod.first-namespace.svc.cluster-domain.example.
```

여러분들이 kubeadm 명령어로 설치하기전 옵션을 살펴보면 pod 및 service 에서 사용하는 도메인 정보가 있다. 변경을 원하는 경우 아래 옵션을 사용하여 kubeadm 실행 시 변경이 가능하다.

옵션	설명
<code>--pod-network-cidr string</code>	일반적으로 10.0.0.0/8 대역

<code>--service-cidr string</code>	<code>10.96.0.0/12</code>
<code>--service-dns-domain</code>	<code>cluster.local</code>

## 8. Deployment(deploy)

Deployment 는 ReplicaSet 에서 사용하는 설정 정보들이 있다. Deployment 에는 POD, SVC, Container, PVC, Template 그리고 Labels 같은 정보가 저장되어 있다. 사용자가 아래와 같이 YAML 파일을 작성하여, 쿠버네티스 클러스터에 생성이 가능하다.

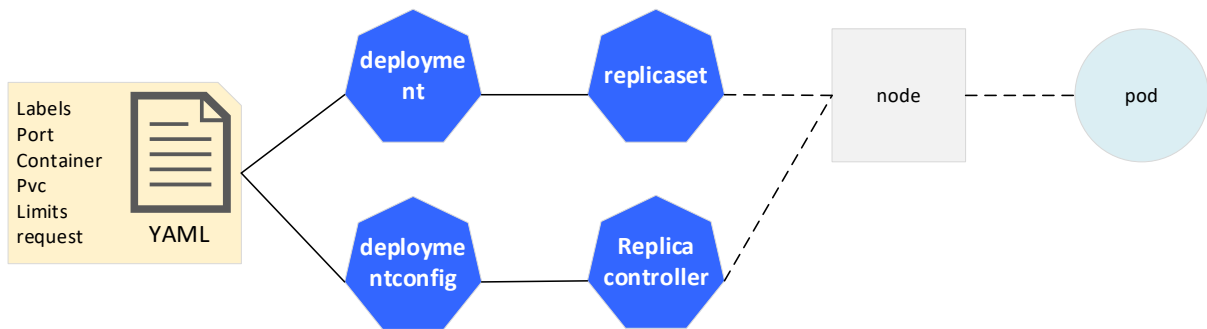


그림 38 Deployment

보통 작성하면 아래와 같은 형식으로 작성한다.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx

```

이를 통해서 Pod 의 롤아웃(rollout) 및 Pod 관리가 가능하다. 사용량이 늘어나는 경우 Deployment 는 RS(ReplicaSet)를 통해서 Pod 를 확장한다. 또한 새로운 정보 혹은 Pod 생성이 요청이 되면, 오래된 Pod 를 제거하면서 새로운 RS 기반으로 Pod 를 생성한다. 문제가 발생시, 해당 시점으로 롤백(roll-back)이 가능하다.

## 9. DeploymentConfig

DeploymentConfig 는 Deployment 와 기본적으로 비슷한 기능을 가지고 있다. DeploymentConfig 는 자원 생성시 ReplicaSet 를 사용하지 않으며, ReplicaController 기반으로 자원을 구성 및 생성한다.

Deployment 와 비슷하게 레이블 즉, 셀렉터 기반으로 사용이 가능하지만, 단순한 셀렉터 기반으로만 사용이 가능하고 복잡한 셀렉터 기능은 사용이 불가능 하다. 현재 쿠버네티스에서는 더 이상 DeploymentConfig(혹은 dc)는 더 이상 사용하지 않는다.

## 10. ReplicaSet(rs)

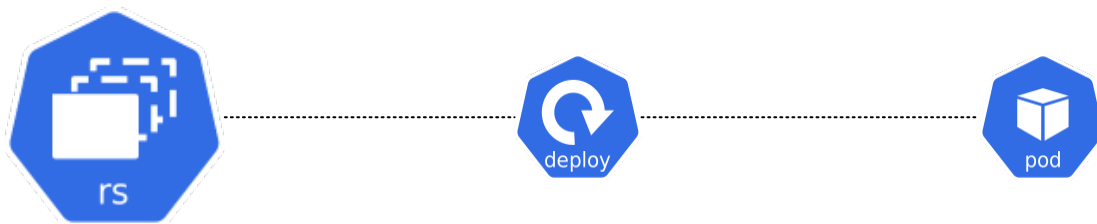


그림 39 deployment, replicaSet, POD 관계

ReplicaSet, 이하 복제자. 기존의 Replication Controller 의 단점을 보완 및 보강하기 위해서 나온 기능이다. 이 자원은 보통 r"라고 많이 부르며, 기존에 사용하던 ReplicationController 와 다르게 레이블 및 템플릿 같은 메타정보 기반으로 사용한다. HPA(Horizontal Pod Autoscaler)같은 기능을 사용하기 위해서는 ReplicaSet 를 통해서 구성이 된다.

## 11. Replication Controller(rc)

초기 쿠버네티스는 replication-controller 를 통해서 애플리케이션 배포 및 관리가 이루어졌다. replication-controller 의 주요 역할은 replicaset 과 비슷하게 POD 관리 및 애플리케이션 복제를 한다. 현재 쿠버네티스는 "RelicationController"사용을 권장하지 않는다.

**Note:** A Deployment that configures a ReplicaSet is now the recommended way to set up replication.

여하튼, 만약, "Replication Controller"를 사용한다면, "ReplicaSet"하고 제일 큰 차이점은 바로 "Replica Controller"는 명시된 개수만큼 Pod 를 생성하며, 항상 명시된 수에 맞게 동작하도록 한다. 문제가 발생하면 문제가 발생한 Pod 를 제거하고 다시 생성을 시도한다.

하지만, Replication Controller 는 구형 버전으로 분류가 되어서 대다수 기능들은 Replication Controller 보다는 ReplicaSet 기반으로 구성이 된다.

# 기본기능

## 1. 컨텍스트 관리

```
# kubectl config set-context --current --namespace=  
# kubectl config get-contexts  
# kubectl config get-users  
# kubectl config get-clusters
```

## 2. 관리를 위한 확장 명령어

```
# kubectl cluster-info  
Kubernetes control plane is running at https://192.168.10.1:6443  
CoreDNS is running at https://192.168.10.1:6443/api/v1/namespaces/kube-  
system/services/kube-dns:dns/proxy  
  
To further debug and diagnose cluster problems, use 'kubectl cluster-info  
dump'.
```

## 3. 자원 이름

쿠버네티스의 모든 자원은 코드 기반으로 구성이 되어 있다. 이 자원을 사용하기 위해서는 YAML 를 사용해서 자원들을 구성해야 된다. 그렇기 위해서는 어떤 방식으로 자원 파일을 작성하는지 알아야 한다. 쿠버네티스의 모든 자원 영역에 대해서 다루지는 못하지만, 기본적인 부분에 대해서 어떻게 다루는지 잠깐 보도록 하겠다.

쿠버네티스의 자원 영역은 보통 다음처럼 분류가 되어있다. 이를 리소스 타입(resource type)이라고 한다.

이름	설명
Pod	컨테이너 pod 및 구성 및 관리하는 자원영역
Service	쿠버네티스에서 구성된 서비스를 외부에서 접근하기 위한 영역
DaemonSet	특정 작업을 반복적으로 구성해주는 서비스.

Deployment	서비스 오케스트레이션을 위한 구성 설정 파일 영역.
ReplicaSet	서비스 복제 및 복구를 위한 자원 영역. 현재 쿠버네티스 자원들은 ReplicaSet 를 통해서 복제를 한다.
Replic-Controller	ReplicaSet 과 같은 기능. 이전 쿠버네티스에서 사용했던 복제자 기능. 지금은 해당 복제자로 생성을 권장하지 않는다.
Statefulset	디플로이먼트와 비슷하게 동작하지만, 스테이트풀셋은 동일한 컨테이너 스펙을 기반으로 포드들을 관리
Job	특정 개수의 작업이 올바르게 완료가 될 때까지 반복적으로 실행한다.
cronjob	시스템의 크론잡처럼, 특정 시간에 반복적으로 실행되는 작업이다.

위에 구성이 되어있는 자원 분류로 쿠버네티스 자원은 생성 및 구별이 된다. 몇몇 자원들은 명령어로 생성이 가능하지만, 매우 제한적인 자원에 대해서만 지원을 하고 있다.

## 4. 명령어 자동완성

쿠버네티스 명령어를 학습하기 위해서 master 서버에 접근한다. 이번 목차에서는 쿠버네티스에서 제일 많이 사용하는 명령어를 연습한다.

쿠버네티스 명령어는 *kubectl*/명령어를 사용하며, 모든 서비스 관리 및 생성/제거한다. 쿠버네티스는 기본적으로 쉘에 쉽게 사용할 수 있도록 bash-completion 기능을 지원한다.

```
master]# kubectl completion bash >> ~/.bash_profile
master]# source /usr/share/bash-completion/bash_completion
```

위의 방법보다는 기본적으로 모든 리눅스 배포판은 completion 이 저장되어 있는 위치가 있다.

```
/etc/bash_completion.d/
```

그래서, 저장 위치를 *"~/.bash\_profile"*보다는 아래와 같이 저장하여 사용을 권장한다.(모든 사용자에게 kubectl, kubeadm 자동완성 기능 제공 시)

```
master]# kubectl completion bash >> /etc/bash_completion.d/kubectl
master]# kubeadm completion bash >> /etc/bash_completion.d/kubeadm
master]# complete -r -p
```

위의 명령어를 실행한 후, 올바르게 completion 이 동작이 되지 않는 경우 다음과 같은 패키지를 설치한다.

```
master]# yum install bash-completion
```

설치가 완료가 되면 다시 다음과 같이 명령어를 실행한다.

```
master]# source ~/.bash_profile
```

위의 기능이 올바르게 설치가 되면 다음처럼 명령어를 테스트한다.

```
master]# kubectl <TAB><TAB>
alpha      cluster-info  diff          logs          scale
annotate   completion    drain         options       set
api-resources  config      edit          patch         taint
api-versions  convert      exec          plugin        top
apply       cordon       explain       port-forward  uncordon
attach      cp           expose        proxy         version
auth        create       get           replace       wait
autoscale   delete      kustomize     rollout
certificatedescribe  label       run
```

## 5. 노드 상태 확인

학습하기 전, 쿠버네티스의 전체적인 상태를 확인한다. 기본 구성원 및 노드 그리고 API 주소가 잘 동작하는지 확인한다.

```
master]# kubectl get componentstatuses
Warning: v1 ComponentStatus is deprecated in v1.19+
NAME                STATUS    MESSAGE    ERROR
controller-manager  Healthy   ok
scheduler           Healthy   ok
etcd-0              Healthy   ok

master]# kubectl get nodes
NAME                STATUS    ROLES    AGE   VERSION
master-1.example.com  Ready    master   5d6h  v1.18.2
node-1.example.com    Ready    <none>   5d6h  v1.18.2
node-2.example.com    Ready    <none>   5d6h  v1.18.2
```

```
master]# kubectl cluster-info
Kubernetes control plane is running at https://192.168.10.250:6443
CoreDNS is running at https://192.168.10.250:6443/api/v1/namespaces/kube-
system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info
dump'.
```

위와 같이 출력이 되면 랩 진행에 문제가 없다.

## 6. YAML 문법

쿠버네티스는 YAML 기반으로 작성한다. 문법 자체는 띄어쓰기 기반으로 되어 있기 때문에, 올바르게 들여쓰기를 꼭 해야 한다. 들여쓰기 방법은 다음과 같다.

```
Data:
- name: tang
  age: 41years
  sex: meal
  home:
  - district: gangnam
    zone: Yeksam
    post: 797
```

위와 같이 YAML 기반으로 작성된 리소스 요청은, 쿠버네티스 API 를 통해서 JSON 으로 변환 및 전달이 되어서 작업 수행 및 처리가 된다. 아래는 위의 내용을 JSON 형태로 변경한 내용이다.

```
{
  "Data": [
    {
      "name": "tang",
      "age": "41years",
      "sex": "meal",
      "home": [
```

```

    {
      "district": "gangnam",
      "zone": "Yeksam",
      "post": 797
    }
  ]
}
]
}
```

YAML에서는 다음과 같은 자료형식(Data Structure)를 제공한다. 이 형식은 쿠버네티스에서 정한 형태가 아니라 YAML 표준문서에서 제공하는 형태이다.

1. Key Value Pair
2. Array, List
3. Dictionary, Map

YAML에 리스트 및 배열은 아래와 같이 작성한다.

```

Fruits:
  - Apple
  - Melon

Fruits: [ Apple, Melon]
```

Dictionary, Map는 아래와 같이 작성한다. 사전 및 맵 형식은 배열과 같이 사용할 수 있다. 아래는 사전 및 배열을 같이 사용한 예제이다.

```

Users:
  name: "choi gookhyun"
  jobs: instructor
  skills:
    - redhat
    - use
    - kubernetes
```



쿠버네티스는 보통 리스트 기반으로 작성하기 때문에, 사전 및 맵 형식은 굳이 암기할 필요는 없다. 위의 내용을 가지고 간단하게 쿠버네티스에서 사용할 YAML 파일을 작성한다. 대다수 리소스 설정 및 구성 방법은 쿠버네티스 공식 문서에 전부 있기 때문에 그때그때 참조하면 된다.

파일 이름은 "basic-deployment-nginx.yaml"으로 저장한다.

```
master]# cat <<EOF> basic-deployment-nginx.yaml

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
EOF
master]# kubectl apply -f basic-deployment-nginx.yaml
```

위의 내용은 쿠버네티스에서 애플리케이션 구성 시 사용하는 자원선언 파일이다. 쿠버네티스는 기본적으로 자원 선언 혹은 구성 시, Deployment 영역을 사용해서 구성하며, 이를 통해서 애플리케이션 복제 및 구성을 시도한다. 쿠버네티스에서 사용하는 기본 문법은 다음과 같은 형식을 가지고 있다.

```
---
apiVersion: apps/v1

kind: Deployment
metadata:
  name: nginx
```

맨 위의 “---”는 YAML 문법의 시작을 알리는 부분이다. 이 부분은 생략하여도 되지만, 일반적으로 맨 상단에 관습적'으로 표시한다. “**name:**”부분은 실제로 쿠버네티스에서 사용하는 자원들의 이름, 예를 들어서 컨테이너/POD의 이름이다. 이 이름을 통해서 어떤 애플리케이션 혹은 POD 동작하는지 확인이 가능하다. 미리 알아 두면 좋은 부분은 POD는 애플리케이션이 아니며(여기에 대해서는 뒤에서 좀 더 설명하도록 하겠다), **컨테이너**는 우리가 일반적으로 사용하는 애플리케이션이 동작한다.

그 다음에 있는 “**apiVersion**”부분이다. 릴리즈 버전이나 그리고 용도에 따라 다른데, 현재(2023)까지는 애플리케이션은 기본 버전은 “**apps/v1**”으로 되어 있다.

해당 부분에 “**beta/v3**”와 같이 표현이 되어 있는 YAML 파일이 있는 경우, 구형 YAML 파일이니 사용하지 않아도 된다. 아래 **metadata**는 말 그대로 해당 자원의 메타정보를 구성하는 부분이다. 실제 동작하는 서비스에는 영향이 가지 않지만, 추가적인 정보를 넣을 때 사용한다. 이 부분은 **annotation**과 다른 부분이다.

```
labels:
  app: nginx
```

**labels** 세션은 메타 정보의 “**select:**”에서 사용하기 위해서 존재한다. **selector**에서 바라보고 있는 자원의 이름은 항상 **labels**에서 명시가 되어있는 값을 찾게 된다. 아래 예제를 참고한다.

```
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

**spec**에 복제자(replicas)가 사용할 개수 그리고 복사할 대상이 **matchLabels**라는 영역에 선언이 되어 있다. 아래 예제에서는 선택한 자원은 POD의 레이블 이름 “**app**” 그리고 값은 “**nginx**”이다.

```
spec:
  containers:
    - name: nginx
      image: nginx:1.14.2
```

```
ports:
```

```
- containerPort: 80
```

하단에 보면 "containers:"라는 지시자가 보이는데, 이를 통해서 어떤 컨테이너 이미지를 사용하여 서비스를 구성할지 결정한다.

이 부분은 위의 label 부분과 다른데 레이블에서는 "app: nginx"가 생성시 큰 의미가 없지만, 위의 "image: nginx:1.14.2"부분은 컨테이너 생성시 큰 영향이 있다. 이러한 부분은 사용하다 보면 금방 알게 되니, 굳이 외워야 할 필요는 없다. "image:"부분은 Pod 생성시 사용할 이미지 이름이다. 이미지를 가지고 있는 이미지 레지스트리 서버를 명시하지 않는 경우, 기본 이미지 레지스트리 서버에서 다운로드 한다.

모든 이미지는 기본적으로 OCI 사양에 맞추어서 "/etc/containers/registry.conf"에 등록된 순서대로 접근하여 이미지를 검색한다.

```
ports:
```

```
- containerPort: 80
```

포트번호는 containerPort 으로 명시하며, 여기서 사용하는 포트 번호는 실제 컨테이너 애플리케이션이 사용하는 포트 번호를 이야기한다. 만약 사용하는 포트 번호가 틀린 경우 올바르게 서비스가 맵핑이 되지 않을 수 있다.

위의 내용은 매우 기본적인 Deployments의 구성 내용이다. 실제로는 더 다양한 항목들이 추가가 된다. 더 추가가 되는 항목에 대해서는 책 뒤 부분에서 더 다루도록 하겠다.

## 연습문제

앞서 배운 내용 기반으로 간단하게 YAML 파일 작성한다. basic-deployment-nginx.yaml 파일을 수정해서 구성한다.

1. 자원의 이름을 nginx 에서 apache 로 변경한다.
2. 모든 메타정보의 레이블을 nginx 에서 apache 로 변경한다.
3. 컨테이너 이미지를 nginx:1.14.2 에서 apache:latest 로 변경한다.

## 7. 네임스페이스

쿠버네티스의 네임스페이스(혹은 프로젝트)를 확인하는 방법은 다음과 같다.

```
master]# kubectl get namespaces
NAME                STATUS    AGE
default             Active   5d6h
kube-node-lease     Active   5d6h
kube-public         Active   5d6h
```

kube-system	Active	5d6h
nginx-ingress	Active	5d6h

위에서 확인이 가능하지만, 쿠버네티스는 설치가 완료되면 기본적으로 다음과 같이 네임 스페이스를 제공한다. 아래는 기본적으로 생성하는 네임스페이스이어서, 가급적이면 제거하지 말고 그대로 사용한다.

이름	설명
default	쿠버네티스가 설치되면 최초에 제공되는 비어 있는 사용자 네임스페이스. 별도로 사용자가 네임스페이스를 명시하지 않으면 이를 기본값으로 사용한다.
kube-system	쿠버네티스 시스템이 사용하는 자원은 이 네임스페이스 생성이 된다.
kube-public	공개 네임스페이스. 모든 사용자가 사용이 가능하며, 여기에서 말하는 모든 사용자는 인증이 되지 않는 사용자도 포함이 된다.

네임스페이스 생성 방법은 다음과 같다. 네임스페이스 이름이 너무 긴 경우에는 namespace 를 ns 로 사용이 가능하다.

```
master]# kubectl create namespace first-namespace
namespace/first-namespace created
```

생성된 네임스페이스 "first-namespace"는 다음 명령어로 확인이 가능하다.

```
master]# kubectl get namespaces
NAME              STATUS    AGE
default           Active    5d6h
first-namespace   Active    72s
kube-node-lease   Active    5d6h
kube-public        Active    5d6h
kube-system        Active    5d6h
```

혹은 네임스페이스는 다음과 같은 명령어로 생성이 가능하다. 파일명은 second-namespace.yaml 으로 생성한다.

```
master]# cat <<EOF> second-namespace.yaml

apiVersion: v1
kind: Namespace
metadata:
```

```
name: second-namespace
EOF
master]# kubectl apply -f second-namespace.yaml
```

쿠버네티스는 오픈시프트처럼 *project* 명령어가 없다. 그러므로, 네임스페이스를 설정하려면 다음과 같이 명령어를 실행해야 한다. 먼저, 현재 사용중인 네임스페이스를 확인하려면, 아래와 같이 명령어를 실행한다.

```
master]# kubectl config get-contexts
CURRENT      NAME                                CLUSTER      AUTHINFO
NAMESPACE
*            kubernetes-admin@kubernetes        kubernetes   kubernetes-admin
```

위의 내용을 좀 더 자세히 설명하자면 다음과 같다.

```
[root@master1 ~]# kubectl config get-contexts
CURRENT      NAME                                CLUSTER      AUTHINFO
NAMESPACE
*            kubernetes-admin@kubernetes        kubernetes   kubernetes-admin
```

계정 정보      네임스페이스

현재 사용 중인 클러스터      사용자 이름      클러스터 이름

현재는 별도의 네임스페이스가 선택이 되어 있지 않으며, 특정 네임스페이스를 선택하기 위해서는 다음처럼 명령어를 실행한다.

```
master]# kubectl get namespaces
NAME                STATUS    AGE
default             Active    5d6h
first-namespace     Active    9m23s
kube-node-lease     Active    5d6h
kube-public         Active    5d6h
kube-system         Active    5d6h
nginx-ingress       Active    5d6h
master]# kubectl config set-context first-namespace
Context "first-namespace" created.
```

위의 명령어를 실행하여도 크게 명령어 사용에는 문제가 없다. 다만 이전 버전에서는 컨텍스트(context)문제로 올바르게 동작이 안될 수 있다. 위의 명령어를 실행하면 아래처럼 설정파일 내용이 변경이 된다.

현재 사용중인 네임스페이스를 확인하기 위해서는 다음과 같은 명령어로 확인이 가능하다.

```
master]# kubectl config view | grep namespace
```

기존내용(~/.kube/config) 혹은 /etc/kubernetes/admin.conf 에서 확인이 가능하다.

```
contexts:
- context:
cluster: kubernetes
user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
```

아래 내용은 set-context 를 사용하면 설정파일에 보통 아래와 같이 무언가 내용이 추가가 된다.

```
contexts:
- context:
cluster: ""
user: ""
  name: first-namespace
- context:
cluster: kubernetes
user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: first-namespace
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
```

특정 네임스페이스만 지정해서 사용하려면, 다음과 같은 명령어로 노드에서 실행한다.

```
master]# kubectl config set-context --current --namespace=first-namespace
Context "kubernetes-admin@kubernetes" modified.
master]# kubectl config get-contexts
```

CURRENT	NAME	CLUSTER	AUTHINFO	NAMESPACE
*	kubernetes-admin@kubernetes	kubernetes	kubernetes-admin	first-namespace

위의 명령어를 수행하면, 특정 네임스페이스에서 명령어 수행이 된다. 쿠버네티스는 위와 같은 방법은 현재 사용중인 네임스페이스, 즉 프로젝트를 선택 후 사용하던가 혹은 **--namespace, -n** 옵션으로 프로젝트 선택한다. 만약, set-context 하위 명령어로 정보로 변경한 경우, /etc/kubernetes/admin.conf 파일은 ~/.kube/config 으로 복사하면, 다시 올바르게 kubectl 명령어가 실행이 된다.

현재 실행중인 POD 정보를 확인하기 위해서는 kubectl get pods 명령어로 확인이 가능하다. 이 방법이 귀찮은 경우, OKD 의 oc<sup>25</sup>명령어 기반으로 사용하여도 문제없다.

뒤에서 다룰 명령어를 네임스페이스에서 미리 좀 더 다루도록 하겠다. 아래 명령어는 현재 사용중인 네임스페이스에서 pod 를 확인하는 명령어이다. 모든 쿠버네티스 명령어에는 -n, --namespace 옵션이 있는데, 이를 통해서 언제든지 특정 네임스페이스 자원 조회를 명시적으로 가능하다.

```
master]# kubectl get pods
No resources found in first-namespace namespace.
```

아래 명령어는 모든 네임 스페이스에서 사용하는 pod 를 확인하는 명령어이다.

```
master]# kubectl get pods --all-namespaces
```

NAMESPACE	NAME	READY
STATUS	RESTARTS	AGE
kube-system	calico-kube-controllers-6fcbbfb6fb-4jw2d	1/1
Running	3	5d6h
kube-system	calico-node-9rcpx	1/1
Running	3	5d6h
nginx-ingress	nginx-ingress-6f9t2	1/1
Running	4	5d6h
nginx-ingress	nginx-ingress-7f68c7b965-68zqr	1/1
Running	4	5d6h

---

<sup>25</sup> <https://developers.redhat.com/openshift/command-line-tools>

nginx-ingress	nginx-ingress-rshwt	1/1
Running	3	5d6h

위와 동일한 명령어이지만, 레이블도 같이 출력한다. 레이블 부분에 대해서는 뒤에서 다시 이야기하도록 한다.

```
master]# kubectl get pods --all-namespaces --show-labels
```

NAMESPACE	NAME	READY
kube-system	calico-kube-controllers-6fcbbfb6fb-4jw2d	1/1
kube-system	calico-node-9rcpx	1/1
kube-system	calico-node-kgpcp	1/1
kube-system	calico-node-p7tng	1/1
kube-system	coredns-66bff467f8-2js8j	1/1
kube-system	coredns-66bff467f8-kn8fc	1/1
kube-system	etcd-master-1.example.com	1/1

쿠버네티스에서 pod 를 출력 시 정렬이 필요한 경우 --sort-by 명령어를 통해서 정리가 가능하다.

```
master]# kubectl get pods --all-namespaces --show-labels --sort-by=.metadata.name
```

NAMESPACE	NAME	READY
kube-system	calico-kube-controllers-6fcbbfb6fb-4jw2d	1/1
kube-system	calico-node-9rcpx	1/1
kube-system	calico-node-kgpcp	1/1
kube-system	calico-node-p7tng	1/1
kube-system	coredns-66bff467f8-2js8j	1/1

"--all-namespaces" 옵션을 사용하는 경우, 모든 네임스페이스에서 사용중인 pod 에 대해서 출력이 된다. 특정 네임스페이스의 자원만 확인이 필요한 경우, "--namespace"라는 옵션을 통해서 특정 네임스페이스의 이름을 명시한다. "--show-labels" 옵션은 자원에 레이블이 설정이 되어 있으면, 해당 레이블을 같이 화면에 출력한다.

"--sort-by" 옵션은 출력 시 메타데이터의 이름을 내림차순으로 정렬해서 화면에 출력한다. 마지막으로, 노드나 클러스터 같은 자원은 별도로 네임스페이스를 지정하지 않는다.

```
master]# kubectl describe nodes/master.example.com
```

```
Name:          master-1.example.com
Roles:         master
Labels:        beta.kubernetes.io/arch=amd64
               beta.kubernetes.io/os=linux
               kubernetes.io/arch=amd64
```



```
kubernetes.io/hostname=master-1.example.com
kubernetes.io/os=linux
node-role.kubernetes.io/master=
Annotations:  kubeadm.alpha.kubernetes.io/cri-socket:
```

pod 같은 자원 정보를 describe 하위 명령어로 확인하는 방법은 다음과 같다.

```
master]# kubectl describe pod/nginx-ingress-rshwt --namespace=nginx-
ingress
Name:          nginx-ingress-rshwt
Namespace:     nginx-ingress
Priority:       0
Node:          node1.example.com/192.168.90.140
Start Time:    Mon, 04 May 2020 17:41:50 +0900
Labels:        app=nginx-ingress
               controller-revision-hash=7cd97644f4
               pod-template-generation=1
Annotations:   cni.projectcalico.org/podIP: 10.244.17.68/32
               cni.projectcalico.org/podIPs: 10.244.17.68/32
Status:        Running
IP:            10.244.17.68
IPs:
  IP:          10.244.17.68
Controlled By: DaemonSet/nginx-ingress
Containers:
  nginx-ingress:
    Container ID:
docker://6f2b0ddc63854b02a7adf7fafd4d3c76715c9dd0ccab63f45ea38ea6caa943f1
    Image:       nginx/nginx-ingress:edge
```

## 연습문제

기존에 만들었던 YAML 파일기반으로 다음처럼 수정 후 결과를 확인한다.

문제 1 네임스페이스를 아래와 같이 생성한다.

1. 쿠버네티스에서 네임스페이스를 "basic"라는 이름으로 생성한다.
2. 생성된 "basic" 네임스페이스를 기본 네임스페이스로 설정한다.

3. 올바르게 생성이 되면 `kubectl get pods` 그리고 `kubectl config current-context` 명령어로 올바르게 전환이 되었는지 확인한다.

**문제 2** 다음과 같은 이름으로 네임스페이스를 만든다. 만든 후, 네임스페이스를 `set-context` 로 기본 네임스페이스를 변경한다.

1. hello-namespace
2. second-namespace
3. third-namespace

## 8. 생성(create/apply)

쿠버네티스에서 자원을 사용 및 구성하기 위해서는 `create` 혹은 `apply` 를 사용해서 자원을 구성한다. `create` 나 `apply` 를 사용하기 위해서는 최소 1 개의 YAML 파일이 필요하다. 빠르게 서비스를 하나 생성을 해보도록 한다. 파일명은 `nginx-demo-create.yaml` 으로 생성한다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          ports:
            - containerPort: 80
```

간단하게 nginx 애플리케이션을 하나 실행하며, 이 때 실행하는 이름은 "my-nginx"으로 시작한다. 시작 시 총 2 개의 컨테이너가 한 개의 POD 에 연결 및 구성이 되어 동작한다. 컨테이너가 사용하는 이미지 이름 및 버전 그리고 애플리케이션 포트 번호는 아래 ports 에서 명시가 되어있다.

올바르게 생성이 되면 다음과 같은 명령어로 생성을 시도한다.

```
master]# kubectl apply -f nginx-demo-create.yaml
```

두 번째는 apply 명령어로 생성해본다. 위와 똑같이 YAML 파일을 생성을 한다. 파일명은 apache-demo-create.yaml 으로 생성한다.

```
master]# vi apache-demo-create.yaml
```

```
apiVersion: apps/v1
```

```
kind: Deployment
```

```
metadata:
```

```
  name: my-httpd
```

```
spec:
```

```
  selector:
```

```
    matchLabels:
```

```
      run: my-httpd
```

```
  replicas: 1
```

```
  template:
```

```
    metadata:
```

```
      labels:
```

```
        run: my-httpd
```

```
    spec:
```

```
      containers:
```

```
        - name: my-httpd
```

```
          image: httpd
```

```
          ports:
```

```
            - containerPort: 80
```

생성이 완료가 되면 위와 같이 kubectl 명령어로 적용을 해보도록 한다. 두 가지 명령어를 실행을 시도한다.

```
master]# kubectl apply -f apache-demo-create.yaml
```

다시, 기존에 생성하였던 파일에 다음과 같은 부분을 아래처럼 수정한다.

```
spec:
  selector:
    matchLabels:
      run: my-httpd
  replicas: 1 -> 2
```

“replicas:”를 1 에서 2 로 변경 후 다시 서비스 갱신 시도를 한다. 이 때 두 가지 명령어를 시도해본다. 기존 명령어 create 를 다시 실행한다.

```
master]# kubectl create -f apache-demo-create.yaml
```

올바르게 동작하지 않으면, 다시 아래 명령어로 실행을 한다.

```
master]# kubectl apply -f apache-demo-create.yaml
```

올바르게 변경이 되었는지 아래 명령어로 확인해본다.

```
master]# kubectl get pods
```

확인이 되었으면 다음 명령어로 확실히 변경된 내용이 적용이 되었는지 확인한다.

```
master]# kubectl describe pods/<POD_ID>
```

마지막으로 create 명령어로 deployment, service, replica 를 생성 및 구성할 수 있다.

```
master]# kubectl create deployment --image=httpd --dry-run=client --
output=yaml --replicas=2 test-httpd > test-httpd.yaml
master]# kubectl get pod -l app=test-httpd
```

생성된 내용을 아래와 같은 명령어로 확인한다. 간단하게 cat 명령어로 확인한다. 확인이 완료가 되면 바로 자원을 생성한다.

```
master]# cat test-httpd.yaml

apiVersion: apps/v1
kind: Deployment
```

```

metadata:
  creationTimestamp: null
  labels:
    app: test-httpd
  name: test-httpd
spec:
  replicas: 2
  selector:
    matchLabels:
      app: httpd.yaml
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: httpd.yaml
    spec:
      containers:
      - image: httpd
        name: httpd
        resources: {}
status: {}
master]# kubectl create -f test-httpd.yaml

```

쿠버네티스 서비스(service)를 생성하기 위해서는 아래와 같은 명령어로 생성이 가능하다.

```

master]# kubectl create service nodeport --tcp=8080:80 -o yaml --dry-
run=client test-httpd > svc-test-httpd.yaml

```

위의 명령어를 실행하면, 아래와 같이 YAML 파일이 생성이 되며, apply 명령어로 YAML 기반으로 service 파일을 생성한다.

```

apiVersion: v1
kind: Service

```

```
metadata:
  creationTimestamp: null
  labels:
    app: test-httpd
  name: test-httpd
```

```
spec:
  ports:
    - name: 8080-80
      port: 8080
      protocol: TCP
      targetPort: 80
  selector:
    app: test-httpd
  type: NodePort
```

```
status:
```

```
loadBalancer: {}
```

```
master]# kubectl apply -f svc-test-httpd.yaml
```

```
master]# kubectl get service -l app=test-httpd
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
test-httpd	NodePort	10.101.35.88	<none>	8080:30793/TCP	17s

복제자 꾸러미(ReplicaSet)은 아래 명령어로 생성 확인이 가능하다.

```
master]# kubectl get replicaset
```

NAME	DESIRED	CURRENT	READY	AGE
test-httpd-56f7d875db	2	2	0	83s

## 연습문제

기존에 사용하였던 yaml 파일을 사용해서 다음처럼 서비스 구성을 한다.

1. deployment 에 이름은 public-vsftpd 라는 이름으로 구성.
2. vsftpd 이미지를 사용해서 yaml 파일을 작성.
3. 이미지 파일은 아래의 주소에서 받기가 가능.

<https://hub.docker.com/r/fauria/vsftpd/>

4. 받은 이미지의 replica 개수는 1 개로 합니다.

## 9. 실행(run)

run 은 YAML 파일이 아닌 일시적으로 컨테이너 및 POD 를 실행하는 명령어이다. 의외로 자주 사용하는 명령어인데, 보통 테스트용 콘솔 컨테이너를 실행 시 많이 사용한다.

테스트 시 busybox 이미지 기반으로 컨테이너를 생성한다. 아래 명령어는 "-i", "--tty(-t)"옵션을 사용하여 컨테이너 프로세서와 상호작용(interactive)를 한다. 만약, 생성 및 실행이 실패한 경우 바로 삭제하기 위해서 "--rm"옵션도 같이 사용한다.

```
master]# kubectl run curl --rm --image=radial/busyboxplus:curl -i --tty(-t)
```

최근은, 보안상 이유로 쿠버네티스 클러스터에 구성된 컨테이너의 프로그램을 직접 확인하는 범위는 점점 줄어드는 추세이다. 접근을 하기 위해서 디버그(debug) 컨테이너로 접근을 하며, 이 부분에 대해서는 추후에 다시 다룬다. 접속이 끊어진 후, 기존에 사용하였던 컨테이너에 다시 연결하기 위해서는 다음과 같이 명령어를 입력한다.

```
master]# kubectl attach curl -c curl -i -t
```

혹은 Pod 생성 시 사용할 YAML 를 만들기 위해서 다음과 같이 명령어 사용이 가능하다. 먼저, YAML 파일 생성시 사용한 옵션에 대해서 설명이다.

옵션	설명
--image=nginx	사용할 이미지 이름을 적습니다. 이 예제에서는 nginx 라는 이미지를 명시하여 사용하고 있습니다.
--dry-run=client	실제로 생성하지 않으며 kubectl 에서 YAML 이나 혹은 JSON 형태로 자원 생성을 합니다.
--output=yaml	kubectl 에서 생성 시, YAML 형태로 결과를 화면에 출력합니다.

생성한 파일을 아래 명령어로 Pod 생성 합니다.

```
master]# kubectl run --image=nginx --dry-run=client --output=yaml test-nginx > test-nginx.yaml
```

내용을 확인 후, Pod 생성을 시도한다. 올바르게 생성 및 구성이 되었으면 아래처럼 화면에 출력이 된다.

```
master]# cat test-nginx.yaml
```

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  creationTimestamp: null
```

```
  labels:
```

```
    run: test-nginx
```

```
  name: test-nginx
```

```
spec:
```

```
  containers:
```

```
  - image: nginx
```

```
    name: test-nginx
```

```
    resources: {}
```

```
  dnsPolicy: ClusterFirst
```

```
  restartPolicy: Always
```

```
status: {}
```

생성은 다음과 같은 명령어로 실행한다. 올바르게 실행이 되면, 아래와 같이 생성 상태 확인이 가능하다.

```
master]# kubectl apply -f test-nginx.yaml
```

```
pod/test-nginx created
```

```
master]# kubectl get pod -l run=test-nginx
```

NAME	READY	STATUS	RESTARTS	AGE
test-nginx	0/1	Pending	0	26s

## 연습문제

### 문제 1

1. centos 이미지로 console-centos 라는 이름으로 컨테이너를 실행
2. 해당 이미지를 동작하기 위해서 sleep 프로그램을 10000 초로 시작한다
3. httpd 이미지를 YAML 로 생성 후 apply 명령어로 생성한다
4. 생성한 자원은 반드시, 'kubectl get pod'명령어로 확인한다.

### 문제 2



1. httpd 이미지를 nginx 으로 변경한다.
2. 기존의 POD 및 컨테이너의 이름을 nginx 으로 변경.
3. 레이블 정보를 name=nginx, version prod 로 변경.
4. /usr/share/nginx/html/index.html 에 "Hello Nginx"라는 메시지가 출력 되는지 확인.
5. 이 컨테이너는 반드시 basic 에서 동작해야 한다.

## 10. 멀티 컨테이너 구성

Pod 는 한 개 이상의 컨테이너를 가지고 있을 수 있다. 멀티 컨테이너 혹은 다중 컨테이너 구성 방법은 기존 컨테이너 생성 방법과 크게 차이가 없다. 아래 생성 예제이다. Pod 기반으로 생성하면 다음과 같이 멀티 컨테이너를 생성한다.

```
master]# cat <<EOF> multi-containers.yaml

apiVersion: v1
kind: Pod
metadata:
  name: multi-containers
spec:
  containers:
  - name: nginx
    image: nginx
  - name: debian
    image: debian
    command: ["sleep"]
    args: ["10000"]
EOF

master]# kubectl apply -f multi-containers.yaml
```

```
master]# kubectl exec -i -t multi-containers --container nginx --
/bin/bash
master]# kubectl exec -i -t multi-containers --container debian --
/bin/bash
```

## 연습문제

### 11. get

get 명령어는 쿠버네티스 프로젝트 혹은 네임스페이스에 구성이 되어있는 자원 목록을 확인 시 사용하는 명령어이다. 이 명령어를 통해서 오브젝트 안에 구성이 되어있는 리소스 확인이 가능하다. 여기서 잠깐 쿠버네티스의 명령어 동작 구조는 다음처럼 가지고 있다.

```
kubectl <동사> <자원형식> <리소스>
```

위와 같은 구조로 되어 있기 때문에 만약 POD 에서 자원을 확인하고 싶으면 다음처럼 명령어를 실행한다.

```
master]# kubectl get pod
```

위의 명령어는 현재 사용중인 네임스페이스에서 사용하는 경우, 위의 명령어로 처리하고 만약 다른 위치의 네임스페이스 정보를 확인하고 싶으면 다음처럼 명령어를 사용하기도 한다.

```
master]# kubectl get pods --namespace <NAMESPACE_NAME>
```

혹은 네임스페이스 상관없이 모든 리소스를 확인하고 싶으면 다음처럼 명령어를 사용하기도 한다.

```
master]# kubectl get deployment -A
```

실시간으로 출력 내용을 확인하고 싶은 경우 "-w" 옵션을 통해서 실시간으로 갱신 상황을 살펴볼 수 있다.

```
master]# kubectl get pods -w
```

자세한 내용을 출력해서 확인을 원하는 경우 "-o" 옵션을 사용한다. 여기서 자주 사용하는 옵션을 예를 들어서 실행해본다.

```
master]# kubectl get pods -o -A
```

특정 자원만 선택해서 보고 싶은 경우, 레이블(label)로 선택이 가능하다.

```
master]# kubectl get pod -l <LABEL_NAME>=<LABEL_VALUE>
```

## 연습문제

1. pod, deployment, replicaset, replica 에서 구성이 되어있는 자원 목록을 확인한다.
2. 해당 내용을 각각 오브젝트 이름으로 .txt 파일을 만들어서 내용을 저장한다.
3. -o yaml 으로 내용으로 전환 후 저장한다.

## 12. describe/logs

자원 정보를 확인하기 위해서 사용하는 동사 혹은 명령어가 describe 이다. 이 명령어를 통해서 자원들의 정보를 사람이 보기 쉽게 렌더링해서 출력해준다.

사용 방법은 매우 간단하며 다음처럼 실행이 가능하다.

```
master]# kubectl describe deployments.apps/my-httpd
```

혹은 기존에 사용하던 파일이 있는 경우 바로 해당 파일을 명시하여 현재 설정된 내용 및 자원 상태 확인이 가능하다.

```
master]# kubectl describe -f nginx-demo-create.yaml
```

마스터 및 워커 노드의 자원 상태를 확인하기 위해서 아래 명령어로 조회한다.

```
master]# kubectl describe node master.example.com
```

구성된 자원에 대해서 확인을 하고 싶을 때 명확하게 특정 자원 및 객체를 명시하면 된다. 보통은 아래와 같이 많이 사용한다. 이전에 구성한 my-httpd 의 설정(deployment)에 대해서 확인한다.

```
master]# kubectl describe deploy my-httpd
Name:                my-httpd
Namespace:           default
CreationTimestamp:    Fri, 18 Aug 2023 00:14:27 +0900
Labels:              <none>
Annotations:         deployment.kubernetes.io/revision: 1
Selector:            run=my-httpd
Replicas:            1 desired | 1 updated | 1 total | 1 available | 0
unavailable
StrategyType:        RollingUpdate
MinReadySeconds:     0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
```

## 연습문제

위의 명령어를 사용하여 쿠버네티스의 service, deployment, replicaset 에 구성된 내용들을 확인한다.

1. get 명령어를 통해서 자원 목록 확인
2. describe 명령어를 통해서 자원 정보 확인

## 13. cp

cp 명령어는 외부에 있는 파일을 컨테이너 안쪽으로 파일을 복사하는 명령어이다. 기본적으로 컨테이너는 host 자원을 공유하고 있기 때문에, 바로 직접 접근이 가능 하지만, 거의 대다수의 파일 시스템 영역들은 UFS 라는 영역에서 동작하기 때문에 손쉽게 해당 파일시스템 계층 접근이 어렵다. 그래서 cp 명령어를 통해서 쿠버네티스 내부에서 동작중인 컨테이너 안쪽으로 파일을 복사한다.

위에서 create/apply 로 시작중인 컨테이너 안쪽에 파일을 넣어보도록 하겠다. 먼저, 현재 동작중인 컨테이너의 정보를 확인해야 한다.

```
master]# kubectl get pods
```

컨테이너 정보 정보에서 우리가 필요한 정보는 컨테이너의 이름이다.

실제로는 kubectl get pods 명령어로 확인하는 정보는 컨테이너 정보가 아니라, 컨테이너 앞쪽에서 격리를 해주고 있는 POD 이다. 해당 POD 는 한 개 이상의 컨테이너를 가지고 있어도 일반적으로 POD 한 개만 보이는 게 정상적이다.

```
master]# cat <<EOF> index.html
Hello This is my-nginx
EOF
master]# kubectl cp index.html my-nginx:/usr/share/nginx/html/index.html
```

위와 같이 해주면 index.html 파일이 nginx 컨테이너의 HTML 루트 디렉터리에 저장된다. 확인하는 방법은 아래 exec 명령어를 통해서 확인이 가능하다.

## 연습문제

1. 기존에 구성이 되어있는 my-httpd 에 index.html 파일을 복사한다
2. 해당 index.html 파일은 "Hello Apache"라는 문자열을 가지고 있어야 한다
3. 파일이 복사가 되는 위치는 /htdocs 에 helloworld.html 파일을 생성한다
4. 내용은 "hello kubernetes"라고 출력한다

## 14. exec

이 명령어는 컨테이너 내부의 특정 프로그램을 실행 시 사용한다. 실제 서비스나 혹은 업무에서는 거의 사용하지 않으며, 장애처리나 혹은 개발 시 임시적으로 수정 및 장애 확인을 하기위해서 많이 사용한다.

이 명령어를 테스트하기 위해서 위에서 만든 nginx 의 HTML 디렉터리에 들어있는 index.html 파일을 검색해보도록 한다.

```
master]# kubectl exec my-nginx -- ls /usr/share/nginx/html
```

위와 같이 실행하면 POD 를 통해서 컨테이너 안쪽 내용을 검색 및 확인한다. 올바르게 질의가 되었으면 다음과 같은 결과가 화면에 출력이 된다.

```
50x.html
index.html
```

그 외 대화형 모드로 사용이 가능하다. 이전에는 디버그(debug) 모드가 따로 없어서 exec 로 하였다. 앞서 이야기하였지만, 지금은 디버깅 용도로 exec 를 사용하지 않는다.

```
master]# kubectl exec -it my-nginx bash
```

## 연습문제

1. 앞에서 만든 my-httpd 의 index.html 파일을 확인 및 내부 내용을 확인한다.

## 15. expose/service/endpoint

### expose

expose 명령어는 서비스는 현재 내부에서만 되고 있는 서비스를 외부로 노출하는 명령어이다. 현재 쿠버네티스 노드의 네트워크는 외부에서 접근을 하기 위해서 다음과 같은 서비스 유형을 지원하고 있다.

이름	설명	비고
nodePort	오래전부터 지원하던 쿠버네티스 서비스 자원. 호스트에 포트번호를 생성 후, 접근을 허용한다.	
ExternalIP	외부에서 사용하는 아이피(공인 아이피)를 할당하여 서비스에 접근을 가능하도록 한다	더 이상 지원하지 않음.

loadBalancer	쿠버네티스 로드 밸런서를 통해서 서비스를 구성 및 생성한다	iptables, nftables. 리눅스 배포판에 따라 조금씩 다름
clusterip	클러스터 자원이 구성이 되면, 클러스터 아이피를 통해서 자원들이 구성된다. 예를 들어서 endpoint 자원	kubectl get endpoints
externalname	외부 도메인 서비스를 통해서 접근	별도의 DNS 서비스 서버 구성이 필요

현재 구성이 되어 있는 서비스에 NodePort 및 ExternalIP 를 구성해보도록 한다. 서비스를 생성하기 위해서 기존에 생성한 자원을 expose 명령어를 통해서 노출한다. 별도의 옵션을 지시하지 않으면 clusterip 로 노출한다.

```
master]# kubectl expose deployment/my-nginx
```

혹은 YAML 기반으로 자원을 생성 및 관리가 가능하다. 아래 자원도 별도로 명시한 부분이 없기 때문에 clusterip 로 노출한다.

```
master]# cat <<EOF> my-nginx-service.yaml

apiVersion: v1
kind: Service
metadata:
  name: my-nginx
labels:
  run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    run: my-nginx
EOF
master]# kubectl apply -f my-nginx-service.yaml
```

port: 80 은 외부에서 접근 시 사용하는 포트번호이다. 외부에서 포트번호 80 으로 접근하면 deployment 나 혹은 Pod 에 구성이 되어 있는 포트 혹은 명시된 서비스 targetPort 으로 접근한다. 위의 예제에서는 targetPort 가 targetPort: 8080 으로 되어 있다. 컨테이너에서 사용하는 포트가 한 개 이상인 경우에는 아래와 같이 구성도 가능하다.

```
master]# cat <<EOF> multiport-service.yaml

apiVersion: v1
kind: Service
metadata:
  name: my-nginx-multisvc
  labels:
    run: my-nginx
spec:
  ports:
    - name: normal
      port: 80
      protocol: TCP
      targetPort: 8080
    - name: secure
      port: 82
      protocol: TCP
      targetPort: 8082
    - name: monitoring
      port: 83
      protocol: TCP
      targetPort: 8083
  selector:
    run: my-nginx
EOF
master]# kubectl apply -f multiport-service.yaml
master]# kubectl get svc
```

각각 필드에 이름을 할당하여, 컨테이너에 사용할 모니터링 포트나 혹은 보안 포트를 설정 후 이름 기반으로 포트 구성 및 할당이 가능하다. Port 에서 명시된 번호로 접근한 트래픽은 다시 컨테이너의 포트에 전달해야 하는데, 이때는 "targetPort"를(을) 사용해서 다시 전달이 된다.

## service

각각 노드에 특정 포트를 구성 후, 해당 포트만 접근이 가능하게 하려면 nodeport 를 사용해서 구성하면 된다. 아래는 create 명령어를 통해서 서비스에 노드 포트를 생성한 예제이다. 간단한 YAML 이나 자원은 아래 명령어처럼 생성 및 YAML 파일 생성이 가능하다.

```
master]# kubectl create service nodeport --tcp=8080:80 --node-port=30013 \
-o yaml --dry-run=client test-httpd
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: test-httpd
  name: test-httpd
spec:
  ports:
  - name: 8080-80
    nodePort: 30013
    port: 8080
    protocol: TCP
    targetPort: 80
  selector:
    app: test-httpd
  type: NodePort
status:
  loadBalancer: {}
```

노드 포트는 조금 다른 방식으로 서비스 자원에 접근한다.

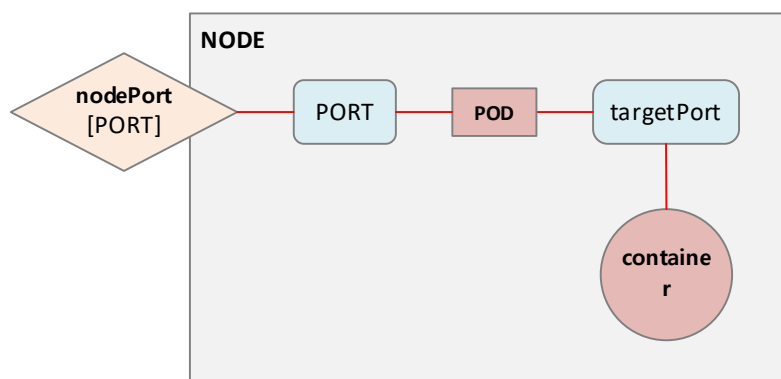


그림 40 노드 포트 구성



파일 생성이 어려운 경우, 위에서 잠깐 사용하였지만 `kubectl create` 명령어로 YAML 파일 생성이 가능하다. 위의 `create` 에서 사용한 명령어는 다음과 같다.

```
master]# kubectl create service nodeport --tcp=8080:80 -o yaml --dry-run=client test-httpd > svc-test-httpd.yaml
master]# kubectl apply -f svc-test-httpd.yaml
master]# kubectl get service
...
test-httpd          NodePort      10.90.55.182    <none>
8080:32145/TCP      3s
...
```

모든 게 비슷하나, 생성 시 `--tcp` 옵션을 사용하여 외부포트 8080 를 통해서 컨테이너 포트 80 으로 접근하게 한다. 컨테이너 서비스를 외부로 노출하는 방법은 결론적으로 다음과 같다.

1. `kubectl create service`
2. `kubectl expose deployment`
3. `kubectl apply -f <YAML_FILE>`

생성된 서비스를 확인하기 위해서 아래 명령어로 확인이 가능하다. 짧은 옵션으로 조회가 가능하다.

```
master]# kubectl get svc
```

위의 명령어를 수행하면 일반적으로 다음과 같은 명령어가 출력이 된다.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	35m

그러면 `expose` 명령어로 `my-nginx` 서비스를 노출을 시도를 한다. 해당 서비스가 문제없이 노출이 되면 보통 다음처럼 `service` 에 생성된 리소스 즉, 자원들이 출력이 된다.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	36m
my-nginx	ClusterIP	10.110.190.142	<none>	80/TCP	11s

## endpoints

쿠버네티스에서 구성된 서비스들이 가지고 있는 엔드 포인트 정보를 확인 시 사용한다. 이 정보는 "Services(SVC)"에서 가지고 있는 컨테이너 및 SVC 의 엔드 포인트 정보를 출력한다. 일반적으로 다음처럼 화면에 출력한다.

```
master]# kubectl get endpoints
NAME                ENDPOINTS
AGE
kubernetes          192.168.90.81:6443
3d6h
nginx               10.244.11.104:80,10.244.11.109:80,10.244.11.110:80 + 11
more...            9h
php-apache          10.244.221.60:80
13h
```

엔드 포인트에 출력된 정보는 보통 두 가지를 가지고 있다. 첫 번째는 POD 가 생성이 되면서 SVC 에 연결된 POD IP, 그리고 외부로 노출이 되어 있는 NodePort 같은 정보들이 화면에 출력이 된다.

## 연습문제

1. my-nginx-second 라는 이름으로 80/TCP 로 my-nginx-second 디플로이먼트 및 서비스를 ClusterIP 로 구성한다.
2. my-nginx-third 라는 이름으로 8500/TCP 로 접근이 가능한 디플로이먼트 및 서비스를 NodePort 로 구성한다.

## 16. proxy, port-forward

proxy

port-forward

## 17. label

레이블은 말 그대로 특정 자원에 레이블을 생성 및 구성한다. 일반적으로 레이블은 YAML 파일에 다음처럼 선언이 되어 있다.

```
metadata:
name: my-nginx
```

```
labels:
  run: my-nginx
```

위와 같은 부분을 레이블이라고 하는데 이 레이블을 YAML 파일이 아닌 명령어를 통해서 추가 및 수정이 필요한 경우 label이라는 옵션을 통해서 처리가 가능하다. 사용방법은 다음과 같이 사용이 가능하다.  
아래는 추가하는 예제이다.

```
master]# kubectl label pods my-nginx-5b56ccd65f-vgn5b type=test
```

아래는 수정하는 예제이다. 위와 거의 흡사하지만, 옵션에 --overwrite 라는 옵션이 포함이 된다.

```
master]# kubectl label pods my-nginx 5b56ccd65f-vgn5b run=test --
overwrite
```

## 연습문제

1. my-httpd 에 type=product 라는 레이블을 추가
2. 추가 후 describe 명령어로 올바르게 추가가 되었는지 확인

## 18. set

자원의 특정 값을 변경한다. 일반적으로 많이 사용하는 부분은 바로 deployment 인데, 이미지 버전을 변경하거나 혹은 포트번호 값을 수정시에 사용한다. 하지만, 이 방법은 바로 콘솔에서 적용하는 방법이기 때문에 바로 사용하는 것은 권장하지 않는다.

일반적으로 수정 배포부분은 항상 YAML 파일을 통해서 apply 형태로 가는 게 제일 안전하다. 하지만, 연습이기 때문에 임의로 변경을 시도를 해본다.

```
master]# kubectl set image deployment/my-httpd my-httpd=image:alpine
```

위의 명령어는 deployment 에 있는 my-httpd 의 자원에서 이미지를 alpine 으로 변경한다.  
변경이 완료가 되면 describe 명령어로 확인이 가능하다.

```
master]# kubectl describe pods/my-httpd
```

## 연습문제

1. my-nginx 의 버전을 mainline-alpine 으로 변경한다.

## 19. edit

ETCD 에 있는 내용을 바로 수정하기 위해서 사용하는 명령어이다. 쿠버네티스에서 생성이나 혹은 수행이 완료가 혹은 실패가 된 자원들은 전부 ETCD 에 기록이 남는다. 쿠버네티스 자체는 API 기반으로 자원을 주고받기 때문에 스스로 어떠한 자원이 올바르게 구성이 되었는지 알 수가 없다.

edit 명령어를 통해서 쿠버네티스에 등록된 설정 파일(정확히는 etcd 데이터베이스)을 YAML 형태로 수정이 가능하다. edit 명령어는 시스템의 EDITOR 나 혹은 KUBE\_EDITOR 라는 값에 영향을 받으며, 일반적으로 vi 명령어가 사용하도록 되어있다. 일반적인 배포판은 nano 및 vi/vim 제공하고 있으며 기본 에디터인 vi 가 어려운 경우에는 nano 를 권장한다.

edit 사용 방법은 다음과 같다.

```
master]# kubectl edit deployment my-httpd
```

혹은 특정 에디터로 변경해서 사용하고 싶은 경우 아래 명령어로 실행이 가능하다.

```
master]# KUBE_EDITOR=nano kubectl edit deployment my-httpd
```

혹은 위에서 말한 것처럼, KUBE\_EDITOR 를 사용해서 변경이 가능하다.

```
master]# KUBE_EDITOR=nano kubectl edit deployment my-httpd
```

위의 내용을 영구적으로 저장하려면 bash\_profile 에 추가를 해주면 된다.

```
master]# echo "KUBE_EDITOR=nano" >> ~/.bash_profile
```

## 연습문제

1. deployment 에 구성된 httpd 및 nginx 의 replica 의 개수를 10 개로 변경 후 상태를 확인
2. 변경 시, edit 명령어를 사용해서 변경

## 20. delete

자원을 제거하는 명령어. 명시가 된 자원들을 제거한다. 모든 자원에 적용되는 명령어 이기에 사용시 주의가 필요하다. 제거하는 방법은 보통 2 가지 방법이 있다.

- YAML 파일에 명시된 자원 제거
- 클러스터에 구성이 되어 있는 명시된 자원 제거

제거를 실행해보도록 한다. 먼저 POD를 제거한다. 제거 시 특정 POD만 제거하도록 selector를 사용하도록 하겠다. "-l" 옵션은 레이블 옵션이다.

```
master]# kubectl delete pod -l run=my-nginx
```

service에 있는 자원을 제거해보도록 하겠다.

```
master]# kubectl delete service my-nginx
```

만약, 모든 서비스를 제거를 원하는 경우, 다음과 같은 명령어를 통해서 제거가 가능하다. 하지만! 뒤는 여러분의 몫이다. 특정 영역만 전부 제거를 하고 싶은 경우, 아래처럼 실행한다.

```
master]# kubectl delete pods --all
```

혹은 영역 상관없이 전부 제거를 하고 싶은 경우 아래 명령어를 실행한다. 아래 명령어는 대략 시스템에서 'rm -rf /' 명령어와 비슷하다.

```
master]# kubectl delete all --all
```

제거가 잘 되었는지 확인을 하기 위해서는 다음 명령어로 확인이 가능하다.

```
$ kubectl get pods
```

## 연습문제

### 21. diff

diff는 YAML에 구성 및 명시된 자원 사양과 맞게 되어 있는지 검증 및 확인하는 명령어이다. 실제 서비스에 배포하기 전에 YAML 파일과 실제 시스템과 어느정도 변경사항이 있는지 확인하는 용도로 사용하기도 한다. 사용법은 매우 간단하다.

```
master]# kubectl diff -f apache-demo-create.yaml
```

위 명령어로 하였을 때 변경사항이 있으면 앞에 + 혹은 -표시로 변동 사항이 출력이 된다. 리눅스 시스템에서 많이 사용하는 diff 명령어와 동일한 명령어이다. 아래 내용은 출력 예제이다.

```
- replicas: 2
+ replicas: 1
```

## 연습문제

1. my-httpd 의 replicas 개수를 100 개로 변경 후 시스템에 등록된 my-httpd 변경사항을 확인하여 올바르게 되었는지 확인한다.

## 22. debug/logs

debug 및 logs 는 포드 및 컨테이너 애플리케이션 상태 및 마스터 혹은 워커 노드 상태를 확인 시 사용하는 명령어이다. debug 는 모든 컨테이너에 사용은 불가능하며, 컨테이너가 debug 기능을 지원하는 경우 사용이 가능하다. 사용이 불가능한 컨테이너에 시도하는 경우 다음과 같은 메시지가 화면에 출력이 된다.

```
master]# kubectl debug -it my-httpd-6796cbbc4c-lr9qg --image=busybox --
target=my-httpd
Targeting container "my-httpd". If you don't see processes from this
container it may be because the container runtime doesn't support this
feature.
Defaulting debug container name to debugger-dvtvc.
```

위와 같은 메시지는 컨테이너 런타임 버전에 따라서 다르게 지원하기 때문에 구 버전을 사용하는 경우에는 동작이 되지 않을 수 있다. 동작이 올바르게 되면 아래와 같은 메시지가 나온다.

```
Defaulting debug container name to debugger-8xzrl.
If you don't see a command prompt, try pressing enter.
/ #
```

debug 와 반대로 logs 는 쿠버네티스에서 동작하는 POD 에서 로그 메시지를 출력한다. 자주 사용하는 옵션은 보통 -f 옵션이 있으며, 이 기능은 시스템의 tail -f 기능과 같다. 실행은 아래처럼 한다.

```
master]# kubectl logs -f my-httpd-6796cbbc4c-lr9qg
AH00558: httpd: Could not reliably determine the server's fully qualified
domain name, using 10.244.221.20. Set the 'ServerName' directive globally
to suppress this message
```

```
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 10.244.221.20. Set the 'ServerName' directive globally to suppress this message
[Sun Sep 26 13:54:00.609837 2021] [mpm_event:notice] [pid 1:tid 140142545593472] AH00489: Apache/2.4.49 (Unix) configured -- resuming normal operations
[Sun Sep 26 13:54:00.609935 2021] [core:notice] [pid 1:tid 140142545593472] AH00094: Command line: 'httpd -D FOREGROUND'
```

## 연습문제

1. my-nginx, my-httpd 에서 발생한 로그를 수집해서 확인한다.

## 23. explain

오브젝트들의 메니페스트(manifest)를 확인한다. 쉽게 말해서 오브젝트 생성시 사용해야 될 필드에 대해서 간단하게 설명을 해준다. 보통 추가된 기능에 대해서 간략하게 기능을 확인하기 위해서 사용한다.

```
master]# kubectl explain pod
KIND:      Pod
VERSION:   v1

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

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

## 24. replace

특정 자원을 교체 시 사용하는 명령어이다. 이 명령어는 apply 와 비슷하게 보이지만, apply 는 기존 내용에서 수정 및 갱신하는 형식이지만, replace 는 아예 해당 내용으로 제거 후 교체하는 명령어이다. 다만, 바로 교체가 이루어 지지가 않으며 교체를 바로 하기 위해서는 보통 --force 라는 옵션을 같이 사용한다.

apache-demo-create.yaml 에 다음처럼 내용을 추가한다. 직접 해보면 어떠한 차이가 있는지 확인이 가능하다.

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

추가가 된 다음에 아래와 같은 명령어를 실행하여 차이점을 확인한다.

```
master]# kubectl replace -f apache-demo-create.yaml
```

그리고 아래 명령어도 실행을 한다.

```
master]# kubectl replace -f apache-demo-create.yaml --force
```

어떠한 차이가 있는지 확인을 해본다.

### 연습문제

1. my-nginx 에 레이블을 system=window 라고 추가한다.
2. replica 개수를 50 개로 변경한다.
3. 시스템에 어떠한 변경이 발생하는지 확인한다.

## 25. patch

```
master]# kubectl patch svc haproxy-ingress -n ingress-controller -p
'{"spec": {"type": "LoadBalancer", "externalIPs":["192.168.10.250}]}'
```

### 연습문제



# 프로젝트 구성 및 애플리케이션 배포

## 1. 시나리오

## 2. 구성 및 소프트웨어

# 고급기능

## 1. 쿠버네티스 스토리지

### 스토리지 서버 구성 및 개념 설명

쿠버네티스에서 제공하는 스토리지 개념은 현재는 총 3 개를 제공하고 있다. 아래는 바닐라 쿠버네티스에서 제공하는 스토리지 정보이다.

- StorageClass
- Persistent Volume
- Persistent Volume Claim

초기에 쿠버네티스는 Volume 기반으로 PV 및 PVC 만 제공하였다. 다만, 컨테이너 개수가 늘어나면서 PV 및 PVC 를 시스템 사양에 맞게 수동으로 확장해야 된다. 이러한 불편함을 해결하기 위해서 StorageClass 를 제공하였다. 기존에 사용하던 PV, PVC 는 정적인 스토리지 제공 방식이며, StorageClass 는 동적인 스토리지 제공 방식을 제공한다.

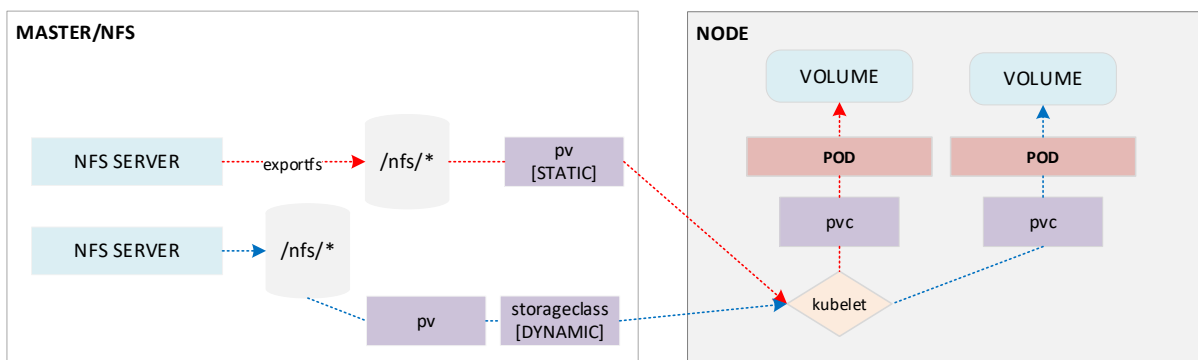


그림 41 PV, PVC, StorageClass

간단하게 NFS 서버 <sup>26</sup>를 구축 후, 위의 그림과 같이 구성하여 포드에 볼륨 구성 및 확인을 한다. 먼저, CSI 드라이버를 다음과 같은 명령어로 구축한 쿠버네티스 클러스터에 설치 및 구성한다. 자동으로 설치하려면 아래 명령어 마스터 노드에서 실행한다. 아래와 같이 설치를 진행한다.

<sup>26</sup> <https://github.com/kubernetes-csi/csi-driver-nfs/>

```

master]# curl -skSL https://raw.githubusercontent.com/kubernetes-csi/csi-
driver-nfs/v4.4.0/deploy/install-driver.sh | bash -s v4.4.0 --
Installing NFS CSI driver, version: v4.4.0 ...
serviceaccount/csi-nfs-controller-sa created
serviceaccount/csi-nfs-node-sa created
clusterrole.rbac.authorization.k8s.io/nfs-external-provisioner-role
created
clusterrolebinding.rbac.authorization.k8s.io/nfs-csi-provisioner-binding
created
csidriver.storage.k8s.io/nfs.csi.k8s.io created
deployment.apps/csi-nfs-controller created
daemonset.apps/csi-nfs-node created
NFS CSI driver installed successfully.
master]# kubectl get pods -n kube-system

```

NAME	READY	STATUS	RESTARTS
coredns-5dd5756b68-96f46	1/1	Running	2
coredns-5dd5756b68-cblx9	1/1	Running	2
csi-nfs-controller-847cc594cf-s5zcn	4/4	Running	2
csi-nfs-node-cbclt	3/3	Running	0
etcd-master.example.com	1/1	Running	2
kube-apiserver-master.example.com	1/1	Running	2
kube-controller-manager-master.example.com	1/1	Running	2
kube-proxy-tjwqg	1/1	Running	2
kube-scheduler-master.example.com	1/1	Running	2

## 랩을 위한 NFS 서버 구축

쿠버네티스에서 스토리지를 구현하기 위해서는 nfs 서버를 유틸리티 서버에 구성한다. 스토리지 서버는 유틸리티 서버에 구성한다.

```

master]# yum install nfs-utils -y

```

설치 후 NFS 서버 서비스 접근을 허용하기 위해서 방화벽을 설정한다. 방화벽 설정이 불편한 경우, `systemctl stop firewalld` 로 서비스를 종료한다.

```

master]# firewall-cmd --add-service nfs
master]# firewall-cmd --add-service nfs --permanent

```

디렉토리를 구성한다. 사용할 디렉토리 위치는 /nfs 에 구성한다. SELinux 가 실행 중이면 중지한다.

```
master]# setenforce 0
master]# getenforce
master]# mkdir -p /nfs
```

구성된 디렉토리 위치를 /etc/exports 를 통해서 외부에 노출한다. 별도로 네트워크 대역을 제한하지 않는다.

```
master]# cat <<EOF> /etc/exports
/nfs *(rw,no_root_squash)
EOF
```

exports 파일에 등록된 정보를 nfs server 에 갱신한다. 갱신 후 nfs.service 를 시작한다. 만약, SELinux 를 사용하고 있으면 반드시 아래처럼 컨텍스트 설정을 디렉터리에 한다. SELinux 를 사용하지 않으면 semanage, restorecon 은 사용하지 않아도 된다.

```
master]# exportfs -avrs
master]# systemctl enable --now nfs-server
master]# semanage fcontext -a -t public_content_rw_t '/nfs(/.*)?'
master]# restorecon -RFvv /nfs
```

사용할 NFS 디렉토리를 사용하기 위해서는 쿠버네티스 인프라에서 PV, PVC 정보를 등록한다. PV 는 백 엔드 드라이버이며, 이를 통해서 컨테이너에 스토리지를 제공한다.

대다수 저장소는 CSI(Container Storage Interface)통해서 별도의 드라이버 구성없이 표준 인터페이스를 통해서 제공한다. 추가적인 스토리지가 필요한 경우, 반드시 쿠버네티스 사이트에서 확인한다.

## StorageClass ServiceAccount

위에서 CSI 드라이버 설치 시 이미 자동으로 NFS 서비스 계정이 생성 및 구성이 되었다. 만약, 수동으로 설치하는 경우 클러스터 관리자는 보통 아래처럼 서비스 계정을 생성한다. 아래 내용들은 CSI DRIVER NFS 에 있는 내용을 그대로 가져온 내용이다.

첫 번째는 서비스 계정을 nfs-pod-provisioner-sa 라는 이름으로 생성한다. 생성이 되는 위치는 명시가 안되어 있지만, 보통 현재 사용중인 네임스페이스 컨텍스트에 생성이 된다.

```
master]# cat <<EOF> storageclass-sa.yaml

kind: ServiceAccount
apiVersion: v1
metadata:
  name: nfs-pod-provisioner-sa
EOF
```

```
master]# kubectl apply -f storageclass-sa.yaml
```

두 번째는 클러스터에 역할 생성한다. 현재 클러스터에 NFS 기능을 추가했기 때문에 API 접근할 수 있도록 권한을 조정한다.

```
master]# cat <<EOF> storageclass-clusterrole.yaml

kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nfs-provisioner-clusterRole
rules:
  - apiGroups: [""] # rules on persistentvolumes
    resources: ["persistentvolumes"]
    verbs: ["get", "list", "watch", "create", "delete"]
  - apiGroups: [""]
    resources: ["persistentvolumeclaims"]
    verbs: ["get", "list", "watch", "update"]
  - apiGroups: ["storage.k8s.io"]
    resources: ["storageclasses"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["events"]
    verbs: ["create", "update", "patch"]
EOF
master]# kubectl apply -f storageclass-clusterrole.yaml
```

세 번째는 역할과 서비스 계정을 바인딩으로 묶는다. 서비스 어카운트 nfs-pod-provisioner-sa 는 클러스터 역할 nfs-provisioner-clusterRole 과 연결이 된다.

```
master]# cat <<EOF> storageclass-clusterrolebind.yaml

kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nfs-provisioner-rolebinding
```

```

subjects:
  - kind: ServiceAccount
    name: nfs-pod-provisioner-sa
    namespace: default
roleRef:
  kind: ClusterRole
  name: nfs-provisioner-clusterRole
  apiGroup: rbac.authorization.k8s.io
EOF
master] kubectl apply -f storageclass-clusterrolebinding.yaml

```

역할이 구성이 될 네임스페이스 및 자원 접근 시 사용이 가능한 권한(verbs)를 설정한다. 아래는 기본적으로 제공되는 역할 설정 내용이다. 역할이 구성되는 네임스페이스 위치는 default 로 한다.

```

master]# cat <<EOF> storageclass-role.yaml

kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nfs-pod-provisioner-otherRoles
rules:
  - apiGroups: [""]
    resources: ["endpoints"]
    verbs: ["get", "list", "watch", "create", "update", "patch"]
EOF
master]# kubectl apply -f storageclass-role.yaml

```

최종적으로 사용이 가능한 서비스 계정에 역할 바인딩을 구성한다.

```

master]# cat <<EOF> storageclass-rolebinding.yaml

kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nfs-pod-provisioner-otherRoles
subjects:

```

```
- kind: ServiceAccount
  name: nfs-pod-provisioner-sa
  namespace: default
roleRef:
  kind: Role
  name: nfs-pod-provisioner-otherRoles
  apiGroup: rbac.authorization.k8s.io
EOF
master]# kubectl apply -f storageclass-rolebinding.yaml
```

## StorageClass

스토리지 클래스는 쿠버네티스의 Persistent Volume 과 비슷하다. 기능적으로 다른 부분은 다음과 같다. 스토리지 클래스는 사용자가 직접 pv 를 구성할 필요가 없다. 자동적으로 pv 를 구성한다. 먼저, 스토리지 클래스를 구성한다. CSI 를 설치 및 구성하지 않으면 올바르게 동작하지 않기 때문에 앞서 진행한 **"기본 구축 및 구성"**을 꼭 수행한다. 스토리지 클래스 파일을 생성한다. 이름은 storageclass-configure.yaml 으로 저장 후 자원을 생성한다.

```
master]# cat <<EOF> storageclass-configure.yaml

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-csi
  storageclass.kubernetes.io/is-default-class: true
provisioner: nfs.csi.k8s.io
parameters:
  server: master.example.com
  share: /nfs
reclaimPolicy: Delete
volumeBindingMode: Immediate
mountOptions:
  - hard
  - nfsvers=4.1
EOF
master]# kubectl apply -f storageclass-configure.yaml
```

다음 명령어로 스토리지 클래스를 클러스터에 등록한다. 올바르게 등록이 되었으면 아래와 같이 화면에 출력된다.

```
master]# kubectl get sc
NAME          PROVISIONER          RECLAIMPOLICY    VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION  AGE
nfs-csi      nfs.csi.k8s.io      Delete           Immediate         false
14m6s
```

생성이 완료가 되면 애플리케이션이 사용할 PVC 를 생성한다. 해당 PVC 는 특별하게 PV 생성 및 요청하지 않는다. PVC 를 생성하기 위해서 아래와 같이 작성 후 자원을 생성한다.

```
master]# cat <<EOF> storageclass-pvc.yaml

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-nfs-dynamic
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: "nfs-csi"
EOF
master]# kubectl apply -f storageclass-pvc.yaml
master]# kubectl get pvc
NAME          STATUS    VOLUME          CAPACITY    ACCESS MODES    STORAGECLASS
AGE
pvc-nfs-dynamic Bound     pvc-fdc1b5b4-0098-4f29-92f4-45d04a696b46    10Gi
RWX          nfs-csi          6h4m
```

여기까지 진행하였으면, Pod 를 생성하여, 실제로 애플리케이션이 볼륨을 올바르게 연결(binding)이 되는지 확인한다. Pod 를 생성하기 위해서 아래와 같이 YAML 작성 후, Pod 생성한다.

```
master]# cat <<EOF> nfs-csi-pod.yaml
```

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nfs-csi-pod
spec:
  selector:
    matchLabels:
      app: nfs-csi-pod
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: nfs-csi-pod
    spec:
      serviceAccountName: nfs-pod-provisioner-sa
      containers:
        - name: sc-nginx
          image: nginx
          volumeMounts:
            - name: csi-nfs
              mountPath: /var/www/html/
      volumes:
        - name: csi-nfs
          nfs:
            server: master.example.com
            path: /nfs
EOF
master]# kubectl apply -f nfs-csi-pod.yaml
```

올바르게 생성 및 저장소 구성이 되었는지 아래 명령어로 확인한다.



```

master]# kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
nfs-csi-pod-c6f9dbcfc-qpsnv       1/1     Running   0           33smaster]

master]# kubectl get pvc
NAME            STATUS    VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc-nfs-dynamic  Bound    pvc-fdc1b5b4-0098-4f29-92f4-45d04a696b46  10Gi       RWX            nfs-csi        6h9m
master]# kubectl describe pod/deployment-nfs-XXX
...
Mounts:
  /mnt/nfs from nfs (rw)
  /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-
qrnkd (ro)
...

```

올바르게 생성이 되었는지 마스터의 /nfs 디렉터리에 다음과 같이 파일을 만든다. 올바르게 구성이 되어 있으면, 호스트 및 컨테이너 내부에서 확인이 가능하다.

```

master]# cd /nfs/
master]# ls
master]# touch helloworld.html
master]# kubectl exec -it nfs-csi-pod-c6f9dbcfc-qpsnv -- ls
/var/www/html/
helloworld.html

```

위와 같이 출력이 되면, Pod가 구성이 되면서 올바르게 SC를 통해서 PVC가 구성 및 연결이 되었다. SC가 올바르게 동작이 되었으면 PV 확인하면 다음과 같이 출력이 된다.

```

master]# kubectl get pv
NAME            CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS    REASON    AGE
pvc-fdc1b5b4-0098-4f29-92f4-45d04a696b46  10Gi       RWX
Delete          Bound    default/pvc-nfs-dynamic  nfs-csi
6h12m

```

스토리지 클래스가 올바르게 Pod 를 통해서 구성이 되었는지 describe 명령어를 통해서 확인한다. Pod 개수가 변경이 되면, PVC 에서 어떠한 반응이 있는 아래 명령어로 확인한다.

```
master]# kubectl scale --replicas=5 deploy/nfs-csi-pod
```

동작중인 Pod 에서 아무거나 하나 잡아서 아래 명령어로 확인한다.

```
master]# kubectl describe pod/nfs-csi-pod-c6f9dbcfc-hhdv1
Volumes:
  csi-nfs:
    Type:          NFS (an NFS mount that lasts the lifetime of a pod)
    Server:         master.example.com
    Path:           /nfs
    ReadOnly:      false
```

## 연습문제

### Persistent Volume (PV)

스토리지 클래스(SC)를 사용하는 경우, PV 를 생성할 필요가 없다. 하지만, 모든 경우가 동적 스토리지가 필요하지 않는 경우가 있다. 예를 들어서 작은 규모의 서비스 혹은 특정 저장소로 저장해야 되는 경우이다. 이러한 상황에는 PV, PVC 를 수동으로 구성 및 연결한다. 수동으로 PV 를 구성 시, 3 가지 모드를 지원하는데, 해당 모드는 아래와 같다.

모드	설명
RWO	단일 노드만 읽기/쓰기가 가능하다.
ROX	단일 혹은 다수 노드에서 읽기만 가능하다.
RWX	단일 혹은 다수의 노드에서 읽기/쓰기가 가능하다.

위의 정보 기반으로 간단하게 Persistent Volume 를 생성한다. 파일 이름은 manual-pv.yaml 으로 저장한다.

```
master]# cat <<EOF> manual-pv.yaml

apiVersion: v1
kind: PersistentVolume
metadata:
```

```

name: nfs-pv
labels:
  type: nfs
spec:
  storageClassName: ""
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
nfs:
  server: master.example.com
  path: "/nfs/manual-pv"
EOF
master]# kubectl apply -f manual-pv.yaml

```

등록이 문제없이 생성이 완료가 되면 다음과 같이 화면에 출력이 된다.

```

master]# kubectl get pv

```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM
nfs-pv	1Gi	RWX	Retain	Available	

```

1s

```

여기까지 진행이 완료가 되면, 이젠 PVC 를 수동으로 생성해야 한다. 아래 Persistent Volume Claim 에서 수동으로 생성한다.

## 연습문제

### Persistent Volume Claim (PVC)

PV 로 저장소 드라이버 구성이 완료가 되면, PVC 를 생성하여 컨테이너가 사용할 수 있도록 구성한다. PVC 는 컨테이너에 오버레이 기반으로 블록 장치를 생성해서 컨테이너에게 전달한다.

```

master]# cat <<EOF> pvc.yaml

```

```

apiVersion: v1

```

```

kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  storageClassName:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
EOF
master]# kubectl apply -f pvc.yaml

```

등록이 문제없이 완료가 되면 다음과 같이 화면에 출력이 된다.

```

master]# kubectl get pvc

```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
nfs-pvc	Bound	nfs-pv	1Gi	RWX		31s

올바르게 PV 및 확인을 위해서 Pod 및 컨테이너를 생성하여 올바르게 연결이 되는지 확인한다.

```

master]# cat <<EOF> pvc-pod.yaml

```

```

apiVersion: v1
kind: Pod
metadata:
  name: pvc-pod
spec:
  containers:
    - name: pvc-pod
      image: nginx
      volumeMounts:
        - mountPath: "/app/data"
          name: htdocs
  volumes:

```

```
- name: htdocs
  persistentVolumeClaim:
    claimName: nfs-pvc
EOF
master]# kubectl apply -f pvc-pod.yaml
```

생성이 완료가 되면, describe 명령어를 통해서 올바르게 Pod 에 연결이 되었는지 확인한다. 문제 없으면 아래와 같이 연결이 된 위치가 describe 에 출력이 된다.

```
master]# kubectl describe pod pvc-pod
...
Mounts:
  /app/data from htdocs (rw)
  /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-
5245r (ro)
...
Volumes:
  htdocs:
    Type:          PersistentVolumeClaim (a reference to a
PersistentVolumeClaim in the same namespace)
    ClaimName:     nfs-pvc
    ReadOnly:      false
...
```

pvc 도 올바르게 구성 및 연결이 되었는지 확인한다.

```
master]# kubectl describe pvc nfs-pvc
Capacity:          1Gi
Access Modes:      RWX
VolumeMode:        Filesystem
Used By:           pvc-pod
```

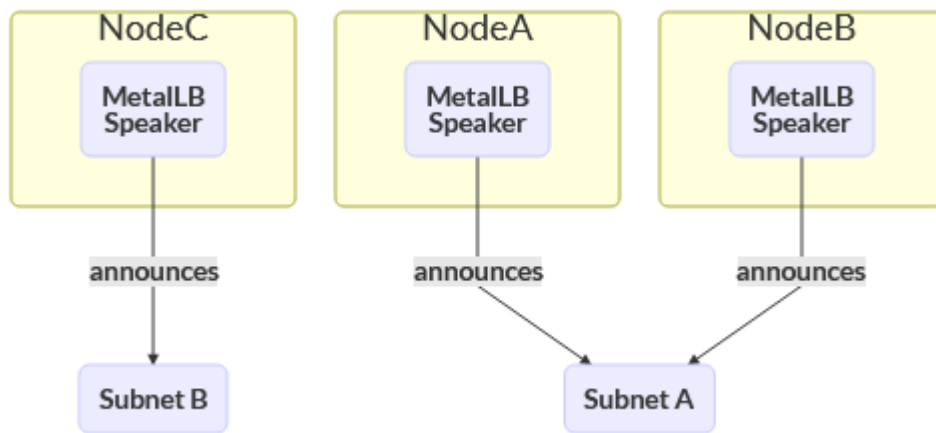
올바르게 볼륨 디렉터리가 바인딩이 되었는지 nfs-pvc 에 파일을 생성 후 컨테이너에서 조회를 한다.

```
master]# touch /nfs/manual-pv/nfs-pvc.html
master]# kubectl exec pvc-pod -- ls /app/data/
```

## 연습문제

### 2. 로드 밸런서

현재, 본 쿠버네티스 클러스터 환경에서 외부 로드밸런서를 구성하기가 어렵기 때문에, Pod 기반으로 내부 로드밸런서를 MetalLB 를 통해서 구성한다. 쿠버네티스에 제공하는 "서비스 로드 밸런서 서비스"는 구성하면 "pending"상태로 구성이 되기 때문에, 최소 한 개의 내부 혹은 외부의 클러스터 서비스 혹은 장치가 구성이 되어야 한다.



일반적으로 로드 밸런서 서비스가 외부나 혹은 내부에 구성이 안되어 있으면, 일반적으로 pending 상태가 출력이 된다.

```
master]# kubectl get svc
testlb                               LoadBalancer    10.90.18.27    pending
80:31973/TCP    6s
```

로드 밸런서 구성이 되어 있으면 다음처럼 메시지가 출력된다.

```
master]# kubectl get svc
testlb                               LoadBalancer    10.90.18.27    192.168.10.10
80:31973/TCP    6s
```

### MetalLB 설치

MetalLB 를 설치하기 위해서 기본 가이드에 따라서 베어메탈 환경으로 구성한다. 먼저, ARP 를 통해서 클러스터 상태를 확인하기 위해서 쿠버네티스에서 ARP 사용 및 범위를 제한한다.

```
master]# kubectl get configmap kube-proxy -n kube-system -o yaml | \
sed -e "s/strictARP: false/strictARP: true/" | \
kubectl diff -f - -n kube-system
```

위의 작업이 완료가 되면, 쿠버네티스에 MetalLB를 설치한다.

```
master]# kubectl apply -f
https://raw.githubusercontent.com/metallb/metallb/v0.14.3/config/manifests/metallb-native.yaml
```

설치 작업이 완료가 되면 다음과 같이 올바르게 설치가 되었는지 확인한다.

```
master]# kubectl get pods -n metallb-system
```

설치가 완료가 되었으면, 간단하게 로드밸런서 기반으로 서비스를 구성한다.

## MetalLB 서비스 구성

MetalLB 기반으로 로드 밸런서를 구성하기 위해서 몇 가지 자원을 생성해야 한다.

```
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: example
  namespace: metallb-system
spec:
  ipAddressPools:
  - first-pool
  nodeSelectors:
  - matchLabels:
      kubernetes.io/hostname: NodeA
  - matchLabels:
      kubernetes.io/hostname: NodeB
```

테스트를 위해서 네임스페이스를 생성한다.

```
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: null
  name: lb-test-pool
```

해당 네임스페이스에 테스트용 자원을 생성한다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: lb-test-pool
  name: lb-test-pool
  namespace: lb-test-pool
spec:
  replicas: 5
  selector:
    matchLabels:
      app: lb-test-pool
  template:
    metadata:
      labels:
        app: lb-test-pool
    spec:
      containers:
        - image: nginx
          name: nginx
          ports:
            - containerPort: 80
```

로드밸런서를 통해서 서비스에 접근이 가능하도록 서비스에 로드밸런서 형식으로 자원을 생성한다.



```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: lb-test-pool
  name: lb-test-pool
  namespace: lb-test-pool
spec:
  ports:
    - name: 80-80
      port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: lb-test-pool
  type: LoadBalancer
status:
  loadBalancer: {}
```

로드밸런서가 사용할 아이피 주소 범위를 할당한다. 여기서는 총 4 개만 할당하도록 한다. 범위는 192.168.10.0/30 으로 설정한다.

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: lb-test-pool
  namespace: metallb-system
spec:
  addresses:
    - 192.168.10.0/30
  serviceAllocation:
    namespaces:
```

```

- lb-test-pool
namespaceSelectors:
- matchLabels:
  lbtype: metallb
serviceSelectors:
- matchExpressions:
- { key: lbtype, operator: In, values: [metallb] }

```

L2 에 연결할 노드를 명시한다. 여기서는 nodea.example.com, nodeb.example.com 으로 명시한다. 로드밸런서를 구성 시 사용할 인터페이스는 eth1 로 하며, 레이블 매치는 호스트 이름으로 구성한다.

```

apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: lb-test-pool
  namespace: metallb-system
spec:
  ipAddressPools:
  - test-pool
  nodeSelectors:
  - matchLabels:
    kubernetes.io/hostname: nodea.example.com
  - matchLabels:
    kubernetes.io/hostname: nodeb.example.com
  interfaces:
  - eth1

```

### 3. helm

helm 은 쿠버네티스에서 사용하는 서비스를 패키징화 한다. Helm 은 Chart 를 통해서 패키징을 하며, 마치 일반 리눅스 패키징 시스템처럼, **설치/업데이트/삭제**와 같은 작업을 할 수 있다. helm 를 사용자가 원하는 경우 직접 패키지를 생성할 수 있으며, 혹은 다른 사용자가 만든 Chart 를 통해서 서비스 구성이 가능하다.

설치 방법은 간단하게 curl 명령어로 설치가 가능하다. 아래는 설치 방법이다.

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

만약, HAProxy 기반으로 Ingress 서비스를 설치하고 싶으면, 아래와 같은 명령어로 helm 기반으로 설치 및 구성이 가능하다.

```
master]# kubectl cluster-info
master]# cat <EOF> haproxy-ingress-values.yaml
controller:
  hostNetwork: true
  ingressClassResource:
    enabled: true
EOF
master]# helm install haproxy-ingress haproxy-ingress/haproxy-ingress\
--create-namespace --namespace ingress-controller\
--version 0.14.4 \
-f haproxy-ingress-values.yaml
master]# kubectl --namespace ingress-controller get services haproxy-
ingress -o wide -w
NAME                                TYPE                CLUSTER-IP          EXTERNAL-IP          PORT(S)
AGE    SELECTOR
haproxy-ingress    LoadBalancer    10.101.52.153    <pending>
80:30238/TCP,443:31546/TCP    13s    app.kubernetes.io/instance=haproxy-
ingress,app.kubernetes.io/name=haproxy-ingress
```

설치된 패키지 목록은 다음 명령어로 확인이 가능하다.

```
master]# helm list
NAME      NAMESPACE      REVISION      UPDATED
STATUS    CHART          APP VERSION
haproxy default      1              2024-02-12 15:15:20.591564711
+0900 KST deployed      haproxy-1.20.0 2.9.2
```

패키지 검색을 원하는 경우, 아래와 같이 검색이 가능하다. 검색 방법은 두 가지를 지원하며 첫 번째는 Artifact 검색, 두 번째 방법은 repository 검색 방식이다.

```
master]# helm search repo haproxy
```

NAME	CHART VERSION	APP VERSION	
DESCRIPTION			
haproxy-ingress/haproxy-ingress	0.14.6	v0.14.6	Ingress
controller for HAProxy loadbalancer			
haproxytech/haproxy	1.20.0	2.9.2	A Helm
chart for HAProxy on Kubernetes			
haproxytech/kubernetes-ingress	1.36.1	1.10.10	A Helm
chart for HAProxy Kubernetes Ingress Con...			

## 연습문제

1. 간단하게 ceph 스토리지를 helm chart 를 통해서 설치한다.

힌트: 아래 주소를 참고하세요.

<https://github.com/rook/rook/blob/master/Documentation/Helm-Charts/ceph-cluster-chart.md>

## 4. metrics 설치

메트릭 서비스는 쿠버네티스에서 동작하는 Pod 및 Node 의 자원 사용 상태를 추적한다. 기본적으로 쿠버네티스는 메트릭 기능을 제공하지 않기 때문에 helm 이나 혹은 kubectl 명령어로 설치해야 한다. 미리 구성해둔 YAML 파일 기반으로 메트릭 서비스를 설치한다.

```
master]# kubectl apply -f
https://raw.githubusercontent.com/tangt64/training_memos/main/opensource/
kubernetes-101/files/metrics.yaml
```

공식 사이트에 설치 가이드는 다음과 같다.

<https://github.com/kubernetes-sigs/metrics-server>

위의 사이트에 가면 다음과 같이 설치를 안내하고 있다. kubectl 명령어로 설치하는 경우, 버전에 따라서 아래와 같이 설치를 진행하면 된다.

## On Kubernetes v1.21+

```
master]# kubectl apply -f https://github.com/kubernetes-sigs/metrics-
server/releases/latest/download/high-availability-1.21+.yaml
```

## On Kubernetes v1.19-1.21

```
master]# kubectl apply -f https://github.com/kubernetes-sigs/metrics-
server/releases/latest/download/high-availability.yaml
```

설치는 Helm Chart 기반으로 구성을 지원한다. Chart 로 설치를 원하는 경우, 아래 링크를 참고한다.

<https://artifacthub.io/packages/helm/metrics-server/metrics-server>

설치 시, 동작이 올바르게 되지 않는 경우가 있는데, 이는 TLS 키 및 포트번호로 발생하는 문제이다. 아래와 같이 수정하면 크게 문제없이 사용이 가능하다.

```
spec:
  containers:
  - args:
    - --cert-dir=/tmp
    - --secure-port=4443
    - --kubelet-preferred-address-
types=InternalIP,ExternalIP,Hostname
```

--secure-port=4443"으로 변경한다. 변경하지 않고 사용을 원하는 경우, SA 계정을 별도로 생성 후 사용을 권장한다. 마지막으로, 메트릭 서비스가 설치가 되어 있지 않는 경우, **HPA/VP** 같은 서비스 사용이 어렵다. 쿠버네티스 클러스터 및 Pod 상태를 확인하는 용도로 사용하지만, 쿠버네티스에서 제공하는 스케일링 서비스를 위해서라도 반드시 필요한 구성원이다.

## 사용 및 확인

문제없이 메트릭 서비스가 생성이 되면, kube-system 네임스페이스에서 metrics-server Pod 확인이 가능하다.

아래 명령어로 Pod 생성이 되었는지 확인한다.

```
master]# kubectl get pods -n kube-system -l k8s-app=metrics-server
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

metrics-server-fb889bc84-22zkc	1/1	Running	0	4m21s
--------------------------------	-----	---------	---	-------

아래 명령어로 Pod 및 Node 자원 사용 상태를 잘 수집하고 있는지 확인한다. 먼저 node 정보를 확인한다.

```
master]# kubectl top nodes
```

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
master.example.com	56m	2%	1633Mi	45%

노드 정보가 확인이 완료가 되면 수집된 Pod 자원 정보를 확인한다.

```
master]# kubectl top pods
```

NAME	CPU(cores)	MEMORY(bytes)
nfs-csi-pod-c6f9dbcfc-5dghp	0m	2Mi
nfs-csi-pod-c6f9dbcfc-hb2pc	0m	2Mi
nfs-csi-pod-c6f9dbcfc-hhdvl	0m	2Mi
nfs-csi-pod-c6f9dbcfc-ngdfl	0m	2Mi
nfs-csi-pod-c6f9dbcfc-qpsnv	0m	2Mi
pvc-pod	0m	2Mi

## 5. 쿠버네티스 컨텍스트 및 사용자 구성

쿠버네티스 사용자는 생성하기가 생각보다 복잡하다. 우리가 알고 있는 사용자 개념을 제공하지만, 기본적으로 LDAP 와 같은 백-엔드 서비스를 통해서 받아오지 않으면, 일반적으로 TLS(X509)기반으로 생성 및 관리가 된다.

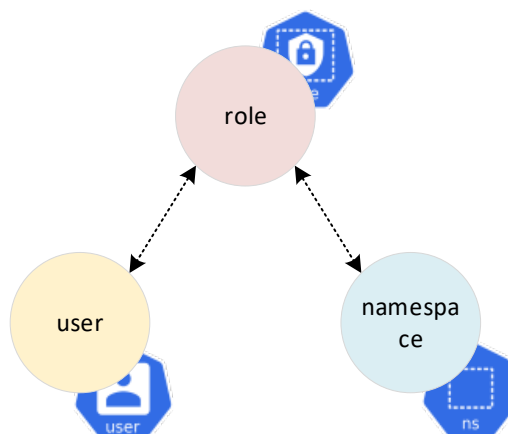


그림 42 네임스페이스 + 사용자 + 역할

네임스페이스(NAMESPACE)는 우리가 익히 알고 있는 것처럼, 자원들이 생성 및 격리가 되는 공간이며(Linux kernel 의 NAMESPACE 와 다른 개념이다.) 네임스페이스 및 사용자는 하나의 **역할(role)**이 할당이 되어 있어야 사용자가 사용이 가능하다.

정리하자면, 생성한 사용자는 X509 기반으로 생성이 되며, 역할을 통해서 네임 스페이스와 연결 및 권한 범위가 할당 및 설정이 된다.

## 사용자 설명

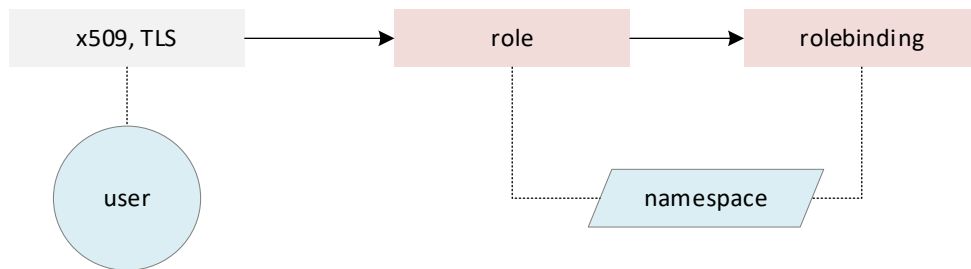


그림 43 사용자 생성방법

새로운 사용자를 생성이 가능한 형식은 두 가지가 있다. 첫 번째는 **서비스 계정(service account)** 두 번째는 일반적인 **쿠버네티스 사용자(normal user)**이다.

**서비스 계정(Service Account)**은 일반 사용자처럼 사용하는 계정은 아니며, 특정 서비스를 임시적으로 사용하기 위해서 사용하는 서비스 계정이다. 쉽게 서비스 계정은 리눅스 명령어의 **sudo** 와 비슷한 구조로 동작한다. 쿠버네티스에서 지원하는 **사용자 관리 기능**은 아래와 같다.

### 기본 사용자 인증 조건

1. API Server 를 통해서 설정이 되어 있는 접근이 가능해야 됨
2. 사용자 이름, 비밀번호, uid, group 조건
3. LDAP 통한 인증

### X.509 인증 기반

1. 사용자는 반드시 비-공개키를 가지고 인증서 인증을 받아야 됨
2. 쿠버네티스 CA 키 기반으로 인증 받은 사용자

### Bearer Tokens(JSON 웹 토큰)

1. OpenID
2. OAuth
3. 웹 훅(Webhooks)기반

## 사용자 생성

먼저 사용자를 생성한다. 생성할 사용자는 satellite, openshift 이다. 실제로 사용자는 만들지 않아도 되지만, 각각 사용자가 사용할 키를 관리하기 위해서 시스템 계정과 같이 만든다. 일반적으로 특정 계정으로 로그인 하였을 때, 시스템 계정과 동일한 쿠버네티스 계정의 TLS 키를 사용하도록 한다.

시스템 계정과 연동이 필요 없는 경우 굳이 만들지 않아도 된다. 먼저, 사용자가 사용할 키를 생성한다. 사용자 satellite 에 대한 rsa 및 private 키를 생성한다.

```
master]# openssl genrsa -out satellite.key 2048
Generating RSA private key, 2048 bit long modulus
.....+++
.....+++
e is 65537 (0x10001)

master]# openssl req -new -key satellite.key -out satellite.csr -subj
"/CN=satellite"
```

생성이 완료가 되면 satellite 사용자 서명키를 쿠버네티스 CA 키와 서명한다. 서명 시 사용하는 파일은 /etc/kubernetes/pki/ca.crt 및 ca.key 를 사용한다. 해당 사용자의 TLS 키는 500 일 동안 사용이 가능하다.

```
master]# openssl x509 -req -in satellite.csr -CA
/etc/kubernetes/pki/ca.crt -CAkey /etc/kubernetes/pki/ca.key -
CAcreateserial -out satellite.crt -days 500
Signature ok
subject=/CN=satellite/O=root
Getting CA Private Key
master]# ls
satellite.crt  satellite.csr  satellite.key
```

생성이 완료가 되면, 최종적으로 해당 TLS 키를 CSR(CertificateSigningRequest)에 등록한다. 등록하기 전, 먼저 생성하나 csr 키를 base64 로 인코딩한다.

```
master]# cat satellite.csr | base64 | tr -d "\n"
```

인코딩이 완료가 되면, 아래와 같이 YAML 파일을 생성한다. 파일 이름은 satellite-csr.yaml 으로 등록한다.

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: satellite
  namespace: test-namespace
spec:
  request: |
```



<인코딩 내용>

```
signerName: kubernetes.io/kube-apiserver-client  
expirationSeconds: 86400 # 최대 하루 유효  
usages:  
- client auth
```

파일 생성이 완료가 되면, 아래와 같은 명령어로 등록 및 확인한다.

```
master]# kubectl create -f satellite-csr.yaml  
master]# kubectl get csr
```

문제없이 사용자가 등록이 되었으면, 새로 등록한 csr 를 사용할 수 있도록 허용한다.

```
master]# kubectl certificate approve satellite
```

사용중인 csr 키를 다시 파일로 가져오려면 다음과 같이 명령어를 실행한다.

```
master]# kubectl get csr myuser -o jsonpath='{.status.certificate}' |  
base64 -d > myuser.crt
```

생성이 완료가 되면, 사용자별로 인증서를 사용하기 위해서 아래와 같은 작업을 진행한다. 사용자 생성이 완료가 되면 비-공개키를 ".certs/" 디렉터리에 저장하며, set-credentials 를 통해서 kubectl 에서 바로 사용이 가능 할 수 있도록 한다.

```
master]# mkdir .certs  
master]# mv satellite.* .certs/  
master]# kubectl config set-credentials satellite \  
--client-certificate=/home/satellite/.certs/satellite.crt \  
--client-key=/home/satellite/.certs/satellite.key  
User "satellite" set.  
master]# kubectl create namespace satellite  
master]# kubectl config set-context satellite-context --  
cluster=kubernetes --user=satellite --namespace satellite  
Context "satellite-context" created.
```

올바르게 사용자가 생성 및 클러스터에 접근이 되었는지 확인하기 위해서 아래 명령어로 확인이 가능하다.

```
master]# kubectl config get-contexts
```

CURRENT	NAME	CLUSTER	AUTHINFO
	NAMESPACE		
*	kubernetes-admin@kubernetes	kubernetes	kubernetes-admin
	satellite-context	kubernetes	satellite

간단하게 사용자를 생성하였다. 현재 사용자는 `satellite` 이며, 클러스터의 이름은 기본 클러스터인 `kubernetes` 으로 접근하였다.

## 연습문제

위의 명령어를 사용하여 다음과 같은 사용자를 생성한다.

1. 사용자 이름은 `user1` 으로 생성한다.
2. 해당 사용자는 네임스페이스 `user1-namesapce` 를 사용한다.

## 6. Role/RoleBinding 생성 및 구성

### Role

사용자를 만들었으니, 사용자에게 어떠한 자원을 어떻게 사용할지, 명시를 해주어야 한다. 자원은 우리가 알고 있는 `pod`, `deployment`, `node` 와 같은 부분이 자원이다. 이 자원에 대해서 어떠한 행동을 할 수 있는지 결정하는 부분은 동사(verb)로 명시한다

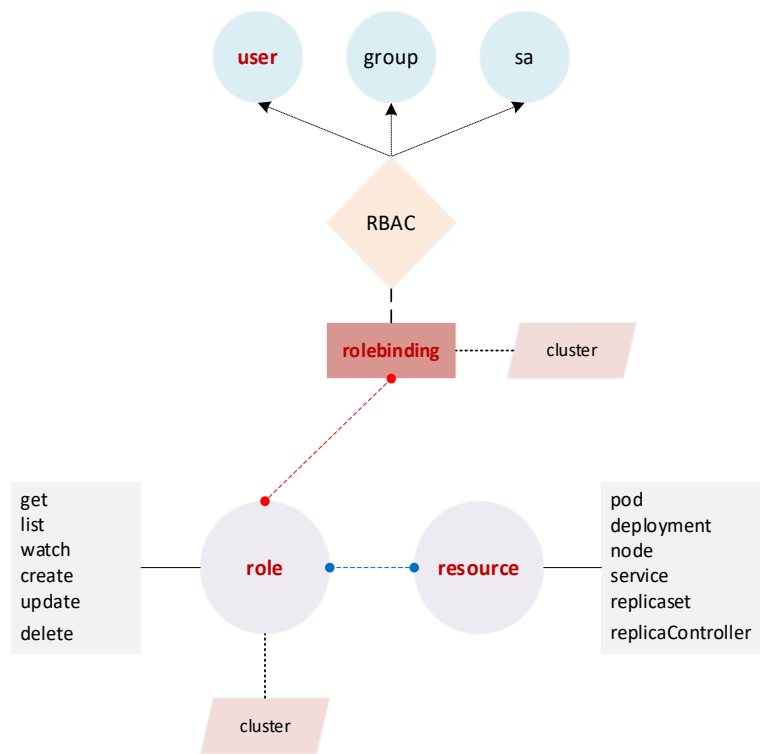


그림 44 역할 및 바인딩

role 은 다음과 같이 분류가 된다.

**role:** 역할은 권한의 영역을 결정한다. 영역은 rule 기반으로 한다.

**subject:** 사용자 혹은 그룹을 명시한다. 보통은 RoleBinding 에서 사용한다.

**RoleBinding:** 역할 바인딩은 어떠한 사용자가 어떠한 역할을 가지고 특정 네임스페이스에서 사용할지 결정한다.

Role 구성 예제는 보통 다음과 같다. 파일 이름은 **satellite-user-role.yaml** 으로 저장한다.

```
master]# vi satellite-user-role.yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: satellite
  name: satellite-role
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list", "create", "update"]
```

위의 내용을 명령어로 생성하면 다음과 같이 생성이 가능하다.

```
master]# kubectl create role satellite-role --verb=get --verb=list --verb=
create --verb update --verb=watch --resource=pods --namespace satellite
```

클러스터에 적용하는 Role 생성하는 경우 아래와 같이 작성한다.

```
master]# vi satellite-user-clusterrole.yaml

kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: satellite
  name: satellite-clusterrole
rules:
- apiGroups: [""]
  resources: ["pods"]
verbs: ["get", "watch", "list"]
master]# kubectl apply -f satellite-user-clusterrole.yaml
```

위의 내용을 명령어로 생성하면 다음과 같이 생성이 가능하다.

```
master]# kubectl create clusterrole satellite-clusterrole --
verb=get,list,watch --resource=pods
```

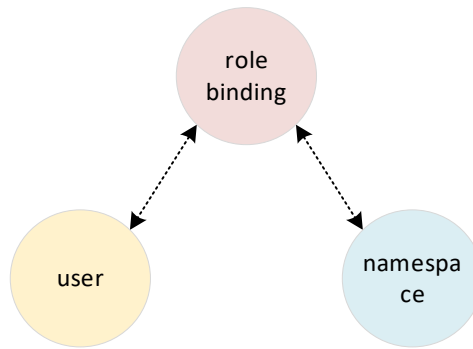
YAML 에서의 명시가 되어있는 지시자를 설명하면 아래와 같다.

1. **apiGroups:** beta, alpha, stable 로 구별한다. 특별한 제한이 없으면 그냥 비어둔다.
2. **resources:** 접근을 허용할 자원을 적는다. 리스트 형태로 여러 개를 적을 수 있다.
3. **verbs:** 자원에 접근 시, 사용이 가능한 명령어. 예를 들어서 get, list, watch 자원 하위 명령어.

위의 YAML 생성을 명령어로 하면 다음과 같이 구성이 된다. 생성이 완료된 role 은 다음과 같은 명령어로 확인이 가능하다.

```
master]# kubectl get role --namespace satellite
```

## RoleBinding



쿠버네티스에서 RoleBinding 은 맵핑(mapping)같은 역할을 한다. 사용자가 생성한 role 과 그리고 사용자를 할당 및 네임스페이스와 연결한다.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: satelllite-rolebinding
  namespace: satelllite
subjects:
- kind: User
  name: test-user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: exampleRole
  apiGroup: rbac.authorization.k8s.io
```

위의 YAML 생성을 명령어로 하면 다음과 같이 구성이 된다.

```
master]# kubectl create rolebinding --role satelllite-role --user
satelllite --namespace satelllite satelllite-rolebinding
```

subjects 밑으로 구성이 되어있는 내용은 다음과 같다.

1. **kind:** 사용자는 일반적으로 User 로 한다.
2. **name:** test-user, test-user 에게 퍼미션 구성 및 설정.

roleRef: 밑으로 구성이 되어있는 내용은 다음과 같다.

1. **kind:** 역할(role)의 형식. 보통은 role, clusterrole 두 가지.
2. **name:** 할당할 역할의 이름.

두 개 이상의 RoleBinding 를 할당하기 위해서 아래처럼 YAML 파일을 작성하였다. 하나는 일반적인 역할, 나머지 하나는 클러스터 역할을 추가하였다. 파일 이름은 **satellite-rolebinding.yaml** 으로 저장한다.

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: satellite-roleBinding
  namespace: satellite
subjects:
- kind: User
  name: satellite
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: role
  name: satellite-role
apiGroup: rbac.authorization.k8s.io

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: satellite
  namespace: satellite
subjects:
- kind: User
  name: satellite
  apiGroup: rbac.authorization.k8s.io
roleRef:
```

```
kind: ClusterRole
name: satelllite-clusterrole
apiGroup: rbac.authorization.k8s.io
```

YAML 파일 작성 후, `kubectl create` 명령어로 적용한다. 적용이 완료가 되면 올바르게 동작하는지 빠르게 컨테이너 하나를 실행한다.

```
master]# kubectl config set-context --current --namespace=satellite
master]# kubectl run httpd --image=httpd
```

현재 클러스터 컨텍스트를 `test-namespace` 으로 변경 후, 올바르게 권한이 적용이 되었는지 확인한다.

```
master]# kubectl auth can-i list pods --namespace satellite
yes
master]# kubectl auth can-i create pods --namespace satellite
yes
master]# kubectl auth can-i delete pods --namespace satellite
yes
master]# kubectl auth can-i update pods --namespace satellite
yes
```

## 7. 역할기반제어(RBAC) 설명

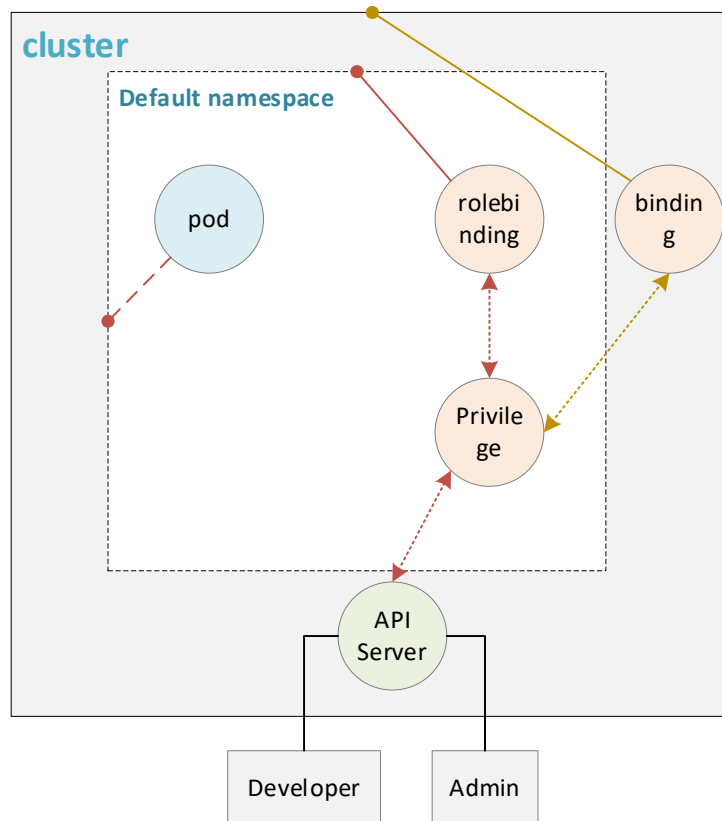


그림 45 RBAC USER, NAMESPACE

위의 role 및 rolebinding 를 사용해서 쿠버네티스 사용자 및 역할을 구성 및 생성하였다. 쿠버네티스는 사용자를 RBAC(Role-Based Access Control)기반으로 관리한다는 사실을 확인하였다. 이 기반으로 쿠버네티스의 주요 자원에서 할 수 있는 작업 범주를 명시한다. 명시가 가능한 범위와 권한(명령어)는 아래와 같다.

- Namespace
- Pods
- Deployments
- Persistent Volume
- ConfigMap

위의 자원에 적용이 가능한 RBAC 의 권한(동사)는 아래와 같이 제공이 된다.

- create
- get
- delete
- list
- update



RBAC 기반으로 쿠버네티스에 적용하기 위해서는 다음과 같은 부분을 명시해야 한다.

- Role / ClusterRole

사용자는 어떠한 네임스페이스를 사용할지, 사용 시 권한 및 접근을 어느 범주까지 허용할지 결정을 해야 한다. 이때 사용하는 기준은 일반적으로 네임스페이스 기준으로 결정한다. ClusterRole 어떤 범위의 클러스터까지 접근을 허용할지 범위 및 범주를 정의하며 여기에는 포함되는 범위는 클러스터 범위(Cluster scoped) 그리고 비-클러스터 범위(non-resource endpoints)이다.

- 주제(Subjects)

주제는 일반적으로 일반 계정(Account) 혹은 서비스 계정(service account)범위가 포함이 된다.

- RoleBinding 그리고 ClusterRoleBinding

이름 기반으로 상속되며, 바인딩이 되는 기준 자원은 subject, role 그리고 ClusterRole 기반으로 된다. 쿠버네티스의 기본 역할은 다음과 같이 기능을 제공한다.

- **view:** 읽기 전용 계정이며, secrets 같은 자원에는 예외이다.
- **edit:** 일반적인 자원에 접근이 가능하지만, 보안에 관련된 role, role binding 에는 접근이 불가능하다.
- **admin:** 모든 자원에 접근이 가능하며, role, role-binding 기능을 네임스페이스에서 사용이 가능하다.
- **cluster-admin:** 노드 자원 및 일반적인 admin 기능 및 접근제어가 전부 가능하다.

## 연습문제

위의 명령어 및 yaml 가지고 아래 사용자를 생성한다.

프로젝트 이름	project-httpd-dev
권한	satellite: edit
프로젝트 이름	project-httpd-prod
권한	satellite: view
	openshift: edit

사용자 키 기반 클러스터 인증을 위한 YAML 파일을 작성한다. 파일명은 "satellite-cluster-account.yaml"으로 아래 내용을 복사 혹은 작성한다. (버전 업데이트 후 Config 타입이 v1 에서 동작이 안됨) 아래 내용은 'kubectl config view'으로 확인이 가능하다.

```
---
```

```

apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority-data: /etc/kubernetes/admin.config
    server: https://192.168.90.210:6443
    insecure-skip-tls-verify: true
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: satelllite
  name: satelllite-context
current-context: satelllite-context
preferences: {}
users:
- name: satelllite
  user:
    client-certificate: /home/satelllite/.certs/satelllite.crt
    client-key: /home/satelllite/.certs/satelllite.key

```

위의 YAML 으로 적용이 되지 않는 경우, kubectl 명령어로 다음처럼 진행한다.

```

master]# kubectl config set-credentials satelllite --client-
certificate=.certs/satelite.crt --client-key=.certs/satelllite.key

```

네임스페이스가 존재하는 경우 아래처럼 테스트가 가능하지만, 현재 네임스페이스가 존재하지 않으므로 아직 실행이 되지 않는다.

```

master]# kubectl config set-context satelite-context --cluster=kubernetes
--user=satelllite --namespace=satelllite

```

## 8. Service Account

서비스 계정은 일시적으로 권한 상승을 위해서 사용하는 계정이다. 사용자에게 관리자와 동일한 권한을 주는 경우, 클러스터 및 관리의 위험성이 증가가 되기 때문에 서비스 계정을 통해서 이러한 문제점을 해결한다.

- **네임스페이스(namespace):** 서비스 계정은 각 네임스페이스 생성이 가능하다. 각각의 서비스 계정은 독립적으로 구성이 된다.
- **경량화:** 서비스 계정은 이미 쿠버네티스 API에 구성이 되어 있기 때문에, 별도로 기능을 추가 설치할 필요가 없다.
- **호환성:** 다양한 워크로드에서 활용이 가능하다. 서비스 계정을 쿠버네티스 컴포넌트와 같이 명시하면, 명시된 네임스페이스 안에서 동작하게 된다. 특정, 네임스페이스에서만 사용하기 때문에 쿠버네티스 클러스터에 큰 영향을 주지 않는다.

쿠버네티스는 클러스터를 구성하면 기본적으로 다음과 같은 서비스 계정을 가지고 있다.

```
master]# kubectl get sa
NAME                SECRETS    AGE
default             0          15d
```

서비스 계정은 명령어로도 생성이 가능하다.

```
master]# kubectl create serviceaccount serviceaccount-example
```

위의 명령어로 확인이 가능하지만, 서비스 계정은 일종의 메타 정보이기 때문에, 별도로 들어가는 설정은 없다. 간단하게 서비스 계정을 활용하는 방식은 다음과 같다.

```
master]# kubectl create namespace test-sa
master]# kubectl create serviceaccount test-sa --namespace=test-sa
master]# kubectl run test-sa-pod --namespace=test-sa --image=nginx -
o=yaml --dry-run=client > test-sa-pod.yaml
```

"test-sa-pod.yaml"에 다음과 같이 파일이 생성이 되었다. 생성된 파일에 서비스 계정을 다음처럼 추가한다.

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
```

```
    run: test-sa-pod
  name: test-sa-pod
  namespace: test-sa
spec:
  serviceAccountName: test-sa
  containers:
  - image: nginx
    name: test-sa-pod
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

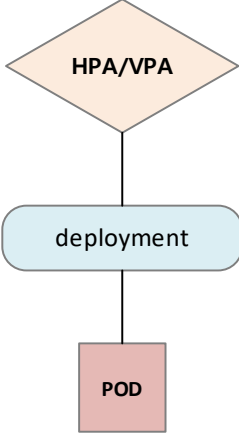
위의 방식은 이름만 다르게 만든 후, Pod 를 test-sa 라는 네임스페이스에 생성한다. 이름만 다르게 만든 후, 생성하기 때문에 딱히 동작에는 문제가 없다.

## 연습문제

## 9. scale/rollout/rollback/history

쿠버네티스에서 동작하는 애플리케이션(컨테이너)을 늘리고 혹은 줄인다. 스케일 기능은 가상화에서 사용하는 스케일 기능과 동일하지만, 가상머신처럼 부트-업 과정이 없기 때문에 매우 빠르게 이미지 기반으로 프로비저닝 후 컨테이너 확장하여 서비스에 영향이 없도록 한다.

쿠버네티스는 다음과 같은 스케일링 기능을 제공하고 있다.

	HPA (Horizontal Pod Autoscaling)
	Pod 를 수평적으로 확장한다. 자원은 확장하지 않으며, POD 개수를 늘린다.
	VAP (Vertical Pod Autoscaling)
	Pod 를 수직으로 확장한다. 수직으로 확장하는 자원은 CPU, MEMORY 같은 자원이다.

실험을 하기 위해서 다음과 같은 명령어를 통해서 서비스를 구성한다. 아래 파일은 nginx-deployment.yaml 로 작성한다.

```
master]# cat <<EOF> nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
```

```
    image: nginx:1.14.2
    ports:
      - containerPort: 80
EOF
master]# kubectl apply -f nginx-deployment.yaml
```

올바르게 생성이 되는지 kubectl get pods 명령어로 계속 확인을 하며, 또한 생성을 rollout 를 통해서 올바르게 되고 있는지 확인이 가능하다.

```
master]# kubectl rollout status deployment/nginx-deployment
```

생성 확인이 완료가 되면, ReplicaSet(rs)를 확인한다.

```
master]# kubectl get rs
```

여기까지 완료가 되면 이젠 디플로이먼트를 업데이트하여 서비스를 다시 rollout 해보도록 한다. 쉽게 진행하기 위해서 간단하게 이미지 버전만 변경하도록 한다.

```
master]# kubectl --record deployment.apps/nginx-deployment \
set image deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1
```

혹은 위의 방식이 어려운 경우 kubectl edit 명령어로 수정을 하여도 된다. 수정이 완료가 되면 롤아웃을 실행한다. 먼저 실행하기전에 앞에 명령어 보면 특이한 옵션이 하나 --record 라는 옵션이 보이는데, 기록을 계속 추적 및 남기기 위해서 저 옵션을 사용한다.

```
master]# kubectl rollout status deployment/nginx-deployment
```

rollout 이 진행이 되면 rs 나 혹은 deploy 에서 확인하면 컨테이너가 프로비저닝 되는 상태가 출력이 된다. 새로 구성이 된 POD 를 확인하기 위해서는 kubectl get pods 명령어로 확인이 가능하다.

자, 이제는 기본적인 rollout 를 사용하였다. 이젠 반대로 다시 rollback 실행을 한다. rollback 우리가 익히 알고 있는 내용처럼, 수행 이전상태로 상태를 변경한다. 단, rollback 가상머신의 롤백 혹은 snapshot 처럼 이루어지는 방식이 아니며 모든 자원들이 제거가 된 다음에 재-생성이 되는 구조이다. 이 부분은 꼭 기억해두자. 이젠 기존에 구성했던 내용에 일부로 오류를 발생한다. 다음과 같은 명령어로 오류를 발생시킨다.

```
master]# kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.161
--record=true
```

그리고 다음 명령어로 rollout 상태를 확인한다.

```
master]# kubectl rollout status deployment/nginx-deployment
```

위의 명령어로 보면 rollout 이 여전히 진행이 되고 있지 않으며, get rs 나 get pods 명령어로 확인하면 여전히 컨테이너 서비스는 여전히 갱신이 안되고 있지 않는 게 확인이 된다.

자 그러면, rollout 된 수정사항을 확인 후 다시 롤백을 시도를 한다. 기억하겠지만, 앞에서 우리가 kubectl 명령어를 실행할 때 --record 라는 옵션을 사용하였다.

아래 명령어로 rollout 기록을 확인한다.

```
master] kubectl rollout history deployment.v1.apps/nginx-deployment
```

확인해보면 옵션을 잘못 넣었다는 사실을 확인할 수 있다.

```
master]# kubectl set image deployment.v1.apps/nginx-deployment  
nginx=nginx:1.161 --record=true
```

rollout 를 사용해서 이전의 내용으로 다시 롤백을 시도를 한다. 아래 명령어로 "revision 2"번의 내용을 좀 더 자세히 확인한다.

```
master]# kubectl rollout history deployment.v1.apps/nginx-deployment --  
revision=2
```

해당 내용이 올바르게 맞으면 다음 명령어로 다시 rollout 를 시도한다.

```
master]# kubectl rollout undo deployment.v1.apps/nginx-deployment
```

성공적으로 scaling, rollout 이 이루어졌다. 하지만, 관리자는 빠르게 명령어로 scale-out 를 원하는 경우 다음과 같은 명령어로 스케일 아웃을 할 수 있다.

```
master] kubectl scale deployment.v1.apps/nginx-deployment --replicas=10
```

```
master]# kubectl rollout pause
```

```
master]# kubectl rollout resume
```

```
master]# kubectl annotate deployment/ghost kubernetes.io/change-cause="kubectl create deploy ghost --image=ghost"
```

## 연습문제

1. my-httpd 에 이미지 버전을 2.4.28 로 변경한다.
2. 변경 후 롤 아웃을 시도한다.
3. 다시 버전을 2.4.25 로 변경한다.
4. 완료가 되면 최종 버전으로 전부 롤 백한다.

## 10. 선호도 및 예외시간(Affinity/Tolerations)

### 포드 선호도(Pod Affinity)

Affinity(이하, 선호도)은 보통 노드에 적용을 한다. 예를 들어서 특정 서비스를 특정 노드에 연결 시 사용한다. 연결 시 nodeSelector 를 사용한다. 이를 통해서 두 가지를 구성할 수 있다.

1. 비-선호도(anti-affinity)
2. 선호도(affinity)

위의 두 가지 조건은 nodeSelector 라는 지시자를 통해서 명시가 된다. 이 조건은 아래와 같이 적용이 된다.

1. 선호도 혹은 비-선호도 설정은 nodeSelector 를 통해서 구성 및 선택을 할 수 있다.
2. 이 지시자를 통해서 soft, preferred 를 통해서 Pod 가 노드에 매치가 되지 않아도 구성이 될 수 있도록 한다.
3. 노드 레이블을 통해서 특정 노드에서 Pod 를 동작하도록 한다. 이를 통해서 특정 노드에서 동작하도록 제약 혹은 구속(constrain)가 가능하다.

마지막으로 노드 선호도(node affinity)는 두 개의 형식으로 구성이 된다.

1. NodeAffinity 기능은 NodeSelector 와 비슷하다. 하지만, 다양한 표현식 및 느슨한 규칙 적용이 가능하다.
2. 내부 Pod 선호도(inter pod affinity)를 통해서 구속(constrain)를 통해서 특정 레이블 Pod 를 제한할 수 있다.



```

apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: topology.kubernetes.io/zone
                operator: In
                values:
                  - antarctica-east1
                  - antarctica-west1
            preferredDuringSchedulingIgnoredDuringExecution:
              - weight: 1
                preference:
                  matchExpressions:
                    - key: another-node-label-key
                      operator: In
                      values:
                        - another-node-label-value
    containers:
      - name: with-node-affinity
        image: registry.k8s.io/pause:2.0

```

## 노드 선호도(Node affinity)

```

apiVersion: v1
kind: Pod
metadata:

```

```
name: with-affinity-anti-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/os
                operator: In
                values:
                  - linux
            preferredDuringSchedulingIgnoredDuringExecution:
              - weight: 1
                preference:
                  matchExpressions:
                    - key: label-1
                      operator: In
                      values:
                        - key-1
              - weight: 50
                preference:
                  matchExpressions:
                    - key: label-2
                      operator: In
                      values:
                        - key-2
    containers:
      - name: with-node-affinity
        image: registry.k8s.io/pause:2.0
```

```
apiVersion: v1
```

```

kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: security
                operator: In
                values:
                  - S1
            topologyKey: topology.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                  - key: security
                    operator: In
                    values:
                      - S2
              topologyKey: topology.kubernetes.io/zone
  containers:
    - name: with-pod-affinity
      image: registry.k8s.io/pause:2.0

```

## 오염 및 내성 (Taints/Tolerations)

### Taints

특정 노드에 대해서 오염 상태(Taint Status)를 설정하여, 특정 자원 및 작업을 수행하지 못하도록 한다.

```
master]# kubectl taint nodes master1.example.com \
node-role.kubernetes.io/master:NoSchedule-
```

## Tolerations

위에서 사용한 taints 와 크게 차이는 없다. 다만, POD 생성 시, Tolerations 조건을 주어서 조건에 맞으면 taint 가 되도록 한다. 간단하게 다음과 같이 구성 및 설정한다.

```
kubectl taint nodes node1.example.com gpu=nvidia:NoSchedule-
```

다시 스케줄링을 허용하기 위해서 아래와 같이 작성한다.

```
kubectl taint nodes node1.example.com gpu=nvidia:NoSchedule-
```

이 내용은 Deployment 나 혹은 POD 에 다음과 같이 등록하여 적용 및 사용이 가능하다.

```
tolerations:
- key: "gpu"
  operator: "Equal"
  value: "nvidia"
  effect: "Schedule"
```

위의 내용을 응용하면 다음과 같이 작성이 가능하다.

```
apiVersion: v1
kind: Pod
metadata:
  name: nvidia
  labels:
    env: gpu
spec:
  containers:
  - name: nvidia
    image: nvidia-gpu
    imagePullPolicy: IfNotPresent
    tolerations:
```

```
- key: "gpu"
  operator: "Equal"
  value: "nvidia"
  effect: "Schedule"
```

## 연습문제

### 11. 변수 전달

쿠버네티스에서 변수 전달은 여러 방식이 있다. 레이블(labels), 주석(annotations)이러한 부분도 변수에 포함이 된다. 하지만, 일반적으로 쿠버네티스에서 변수라고 부르는 "env:"라는 지시자를 보통 말한다. 사용방법은 매우 간단한다. 아래와 같이 보통 YAML 에 작성한다.

```
master]# cat <<EOF> var-demo.yaml

apiVersion: v1
kind: Pod
metadata:
  name: var-demo
  labels:
    purpose: demo
spec:
  containers:
  - name: var-demo
    image: centos
    command: ["/bin/echo"]
    args: ["${DEMO_GREETING}", "${DEMO_APP}"]
    env:
      - name: DEMO_GREETING
        value: "Hello Kubernetes"
      - name: DEMO_APP
        value: "Kubernetes"
EOF
master]# kubectl apply -f var-demo.yaml
```

```
master]# kubectl get pod -l purpose=demo
master]# kubectl logs var-demo
Hello Kubernetes Kubernetes
```

위와 같이 쉘 변수로 컨테이너에서 동작하는 애플리케이션에 변수를 전달한다. 보통은 데이터베이스, 웹 서버, 미들웨어 서비스를 컨테이너 기반으로 구동 시, 변수로 조정 및 설정한다.

## 연습문제

1. 컨테이너 변수를 선언 및 값을 할당한다.

```
MY_NAME=
MY_AGE=
```

2. centos 컨테이너를 실행하면서 다음과 같이 메시지를 출력되게 한다.

```
echo $MY_NAME
echo $MY_AGE
```

## 12. 자동조절(autoscale)

자동확장 혹은 오토 스케일이라고 부르는 기능은 말 그대로 CPU 나 Memory 사용율에 따라서 크기를 조절 및 조정한다. 기술적으로 부르는 이름은 Kubernetes Autoscaler 중에 HPA(Horizontal Pod Autoscaler)의 기능이다. 이 기능을 사용하기 위해서는 반드시 서비스에 "metrics-server"가 동작하고 있어야 한다.

앞서 우리는 이 서비스를 미리 설치하였다.

```
master]# kubectl get pod -n kube-system -l k8s-app=metrics-server
master]# kubectl describe pod -n kube-system -l k8s-app=metrics-server
```

앞에서 이야기하였던 scale 하고 기능적으로 차이는 없지만, HPA 가 특정 조건에 따라서 자동적으로 POD 를 확장하다는 차이가 있다. HPA 를 사용하기 위해서는 'autoscale'을 사용하겠다고 POD 자원에 선언해야 한다. 아래는 선언하는 명령어 예제이다.

```
master]# kubectl autoscale deployment.v1.apps/nginx-deployment --min=1 --
max=15 --cpu-percent=50
```

옵션을 간단하게 설명하면 "--min", "--max"는 HPA 에서 실행 시, POD 생성 개수이다. "--min=1"이면 시작 시 POD 는 1 개로 시작한다. "CPU"가 사용율이 "50%"를 넘으면 POD 를 수평적으로 확장을 시작한다. 그래서 HPA 에 이름에 "H"가 Horizontal 이다.

HPA 기능을 테스트하기 위해서 Deployment 를 작성한다. 파일명은 php-apache-autoscale.yaml 로 작성한다.

```
master]# cat <<EOF> php-apache-autoscale.yaml
```

```
apiVersion: apps/v1
```

```
kind: Deployment
```

```
metadata:
```

```
  name: php-apache
```

```
spec:
```

```
  selector:
```

```
    matchLabels:
```

```
      run: php-apache
```

```
  replicas: 1
```

```
  template:
```

```
    metadata:
```

```
      labels:
```

```
        run: php-apache
```

```
    spec:
```

```
      containers:
```

```
        - name: php-apache
```

```
          image: k8s.gcr.io/hpa-example
```

```
          ports:
```

```
            - containerPort: 80
```

```
          resources:
```

```
            limits:
```

```
              cpu: 500m
```

```
            requests:
```

```
              cpu: 200m
```

```
---
```

```
apiVersion: v1
```

```
kind: Service
```

```
metadata:
```

```

name: php-apache
labels:
  run: php-apache
spec:
  ports:
  - port: 80
  selector:
    run: php-apache
EOF
master]# kubectl apply -f php-apache-autoscale.yaml

```

HPA 가 자원을 모니터링 할 수 있도록 HPA 자원을 생성한다. 아래 명령어로 최소 2 개 최대 15 개의 Pod 를 CPU 사용율이 80%에 다다를 때 확장하도록 하였다. 적용되는 시점은 컴퓨터 사양마다 다르기 때문에, 오랫동안 기다리면 점점 Pod 가 확장되는 모습이 화면에 출력이 된다.

```

master]# kubectl autoscale deploy/php-apache --min=2 --max=15 --cpu-
percent=80
master]# kubectl get pods -w -l run=php-apache

```

메트릭 서버가 올바르게 구성이 되어 있다고 하면, 잠시 시간이 지나면 HPA 에서 CPU 및 Memory 사용율을 수집하기 시작한다.

```

master]# kubectl get hpa
master]# kubectl get hpa

```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS
AGE					
php-apache	Deployment/php-apache	0%/80%	2	15	2
2m56s					

다른 터미널을 하나 더 실행하여 서버에 다시 접속 후 다음처럼 다시 확인한다.

```

master]# kubectl get deployment php-apache

```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
php-apache	2/2	2	2	3m33s



현재 디플로이먼트와 잘되고 있는지 확인한다. 잘 구성되고 동작이 되고 있으면 다음 명령어로 HPA 구성 내용을 확인한다.

```
master]# kubectl get hpa -o yaml > hpa-php-apache-autoscale.yaml
```

아래와 같이 내용이 나오면 HPA 구성은 문제없이 되었다.

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache-autoscale
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
status:
  observedGeneration: 1
```

HPA 구성은 다음과 같은 방식으로도 가능하다. 만약 autoscale 명령어가 아닌 YAML 로 구성을 하고 싶은 경우, 아래처럼 YAML 파일 작성 후 배포가 가능하다.

```
master]# cat <<EOF> hpa-yaml-apache-autoscale.yaml

apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
```

```

name: hpa-yaml-apache-autoscale
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
EOF
master]# kubectl apply -f hpa-yaml-apache-autoscale.yaml

```

## 연습문제

1. 메트릭 서버가 구성이 되어 있는지 확인한다.
2. 구성 후 php-apache 의 개수를 최소 5 개 최대 20 개로 변경한다.
3. cpu 사용율을 30%으로 변경한다.

## 13. drain/taint/cordon/uncordon

### 비우기(drain)

**drain** 은 말 그대로 모든 것은 배출하는 명령어이다. 즉, 현재 워크 노드에서 사용중인 모든 컨테이너를 유지보수나 혹은 장비 교체 같은 이유로 전부 제외시키는 명령어. 이 명령어는 수행이 되면 드레인이 된 컨테이너는 다른 노드에서 다시 구성이 되어서 동작을 하게 된다. 일반적으로 "관리모드"로 변경하기 위해서 다음과 같은 과정을 수행한다.

```

get nodes -> taint node -> drain node -> uncordon node -> taint node
NoSchedule- -> delete node

```

드레인을 하기 위해서 먼저 노드 이름을 확인한다. 아래 명령어로 노드 이름을 확인한다. 이 과정을 진행하기 전에, 반드시 컴퓨트 역할의 노드가 두 개가 존재해야 한다.

```

master]# kubectl get nodes

```

먼저, 새로운 컨테이너가 생성되지 않도록 taint 를 선언한다. taint 는 직역하자면 "오염"이다. 오염이라고 노드에 선언하면, 해당 노드에는 더 이상 컨테이너가 생성되지 않는다. 선언된 노드는 스케줄러 상태가 "DisabledScheduled"라고 표시가 된다. 두 번째 명령어는 "NoExecute"는 "더 이상 POD 를 실행하지 않는다"라고 선언한다. 만약, 마스터 노드만 가지고 있는 경우, taint 명령어는 큰 변경사항이 없다.

```
master]# kubectl taint nodes node1.example.com:NoSchedule
master]# kubectl taint nodes node1 node1.example.com:NoExecute
```

taint 가 실행이 되면, 내부에 사용하던 Pod 는 다른 노드로 재구성이 되면서 삭제가 된다. 실행 후 'describe'명령어로 확인하면 다음처럼 화면에 출력이 된다.

```
Taints:          key1=value1:NoExecute
                  key1=value1:NoSchedule
Unschedulable:   false
```

자, 이제는 실행중인 컨테이너를 다른 노드로 이동을 해주어야 한다. 이미 "value:NoExecute"의 영향을 받아서 POD 는 다른 노드로 이동이 되었다. 그래도 확실히 제외하기 위해서 한번 drain 를 실행한다. (앞으로 drain 은 드레인으로 한글로 적겠다)

```
master]# kubectl drain node1.example.com
```

드레인 실행 후, 노드 상태를 확인한다.

```
Taints:          key1=value1:NoExecute
                  key1=value1:NoSchedule
                  node.kubernetes.io/unschedulable:NoSchedule
Unschedulable:   true
```

## 오염 및 차단 (taint/cordon, uncordon)

드레인이 실행이 되면 위의 출력처럼 Unschedulable 이 true 로 변경되었다.

이번에는 cordon 를 실행한다. 드레인과의 차이점은 직역하면 **비상선** 혹은 **폴리스 라인**과 같다. 드레인은 스케줄링 정책 변경 및 Pod 를 이동하며, cordon 은 스케줄링 정책만 변경한다.

이름	설명
cordon	자원은 그대로 있고, 노드에 스케줄링 정책만 변경한다.

drain	노드에 자원을 다른 노드로 이동하고 노드 스케줄링 정책을 변경한다.
-------	---------------------------------------

참고로 드레인을 수행하였을 때, 모든 Pod 가 재구성되지 않는다. 예를 들어서 네트워크 및 스토리지 같은 시스템 POD 는 보통 그대로 남아 있다.

아래는 사용중인 node2 번에 드레인이 아닌, cordon 를 통해서 스케줄링을 중지한다. 중지 후 컨테이너가 이동이 되었는지 확인한다.

```
master]# kubectl cordon node2.example.com
node/node2.example.com is cordoned
```

드레인이 완료가 되면, 해당 서버에 더 이상 POD 가 생성이 되지 않도록 설정을 해야 한다. 이때 사용하는 명령어는 'taint'라는 명령어를 사용한다. "taint"는 직역하면 "오염"이라는 뜻인데, 서버가 특정한 이유로 더 이상 컨테이너 생성을 하면 안 되는 경우, 보통 "taint" "오염된 상태"로 표시한다.

```
master]# kubectl taint node node1.example.com key1=value1:NoSchedule
```

이제 서버를 물리적으로 확인을 할 수 있는 상태가 되었다. 이제 IDC 에 가서 서버의 뚜껑을 열어 보았다. 이런! 서버에 물리적인 문제가 있다는 부분을 확인하였다. 더 이상 이 서버를 사용할 수 없기 때문에 목록에서 제거를 해야 한다. 제거하기 위해서는 다음과 같은 명령어를 사용한다.

```
master]# kubectl delete node node1.example.com
```

위의 명령어를 실행하면 더 이상 클러스터에서 해당 노드는 확인이 불가능하다. 만약, 제거를 하지 않아도 되는 상태인 경우, 아래 명령어로 다시 클러스터에서 서비스할 수 있도록 스케줄러를 복원 및 복구한다.

```
master]# kubectl uncordon node1.example.com
```

위의 명령어를 실행하면 다음과 같이 노드 상태가 변경이 된다.

```
master]# kubectl get nodes
NAME                STATUS    ROLES                  AGE      VERSION
master1.example.com Ready     control-plane,worker   2d19h    v1.24.3
node1.example.com    Ready     <none>                 2d19h    v1.24.3
Taints:              key1=value1:NoExecute
                    key1=value1:NoSchedule
Unschedulable:      false
```

변경이 완료가 되었다, 하지만 여전히 노드에는 이전에 걸어 두었던 taints 옵션이 걸려있다. 해당 옵션을 다음과 같은 명령어로 제거한다. 제거는 맨 마지막에 마이너스 기호(-)만 추가하면 해당 조건은 제거가 된다.

```

master]# kubectl taint nodes node1.example.com key1=value1:Schedule-
master]# kubectl taint nodes node1.example.com key1=value1:NoSchedule-
Taints:                <none>
Unschedulable:         false

```

이제 정상적으로 서비스가 가능하도록 정상적으로 노드가 복구가 되었다.

## 연습문제

앞에서 학습했던 내용을 가지고 아래의 요구사항을 수행한다.

1. 기존 node2 에 있는 POD 를 전부 node1 으로 이전한다.
2. 이전 작업들은 앞에서 이야기하였던 단계로 수행한다.
3. 작업이 완료가 되었으면 다시 노드를 서비스할 수 있도록 스케줄러를 복원 및 복구한다.

## 14. 노드 추가 및 제거(Node add/remove)

마스터 노드는 단일 노드로 구성하였기 때문에 현재 클러스터 구조에서는 추가가 불가능하다. 하지만, 워커 노드 경우에는 추가가 가능하기 때문에 어떠한 방식으로 노드를 추가하는 확인한다.

기본적으로 추가하기 위해서는 부트스트랩(bootstrap)에서 사용한 토큰을 생성해야 한다. 이미 생성한 토큰의 정보는 아래와 같이 확인이 가능하다.

```

master]# kubeadm token list

```

TOKEN	TTL	EXPIRES	USAGES
DESCRIPTION			EXTRA GROUPS
kay4rj.jo9u2fyp3ip30gsh	18h	2023-08-21T07:05:41Z	
authentication, signing	<none>		
system:bootstrappers:kubeadm:default-node-token			

현재 생성한 토큰의 개수다. 현재는 한 개만 있지만, 여러 번 생성한 경우에는 여러 개의 토큰이 생성이 된다. 현재 워커 노드는 node1.example.com 만 구성이 되어 있다. 추가적으로 node2 를 워커 노드로 추가하기 위해서는 다시 한번 부트스트랩 시 사용할 토큰을 생성한다.

```

master]# kubeadm token create --print-join-command
kubeadm join 192.168.10.250:6443 --token 3aey7k.f0k6nv494vhdr1go \
--discovery-token-ca-cert-hash \
sha256:03ac8bd8b173d57881bc70920040731fa0ef37c1b10c4447cd048229e8a35fab

```

위의 명령어를 다시 node2.example.com 에서 실행한다. 단, 실행 전 "설치 전 O/S 설정"가 완료가 되어야 한다.  
위의 명령어를 다시 node2 에 실행한다.

```
node2]# kubeadm join 192.168.10.250:6443 --token 3aey7k.f0k6nv494vhdr1go  
\  
--discovery-token-ca-cert-hash \  
sha256:03ac8bd8b173d57881bc70920040731fa0ef37c1b10c4447cd048229e8a35fab
```

위의 명령어가 실행이 되면, 부트 스트래핑 과정이 실행이 되면서 node2 를 마스터 노드를 통해서 클러스터 구성을 한다. 성공적으로 구성이 완료가 되면 아래와 같이 확인이 가능하다.

```
master]# kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
master.example.com	Ready	control-plane	21h	v1.28.0
node1.example.com	Ready	21h v1.28.0		
node2.example.com	Ready	control-plane	1m	v1.28.0

제거를 원하는 경우 다음과 같이 명령어를 수행한다.

```
master]# kubectl delete node node2.example.com
```

## 연습문제

노드 2 번을 다시 클러스터에 추가한다. 추가하기전, 노드 2 번은 스냅샷으로 초기화 한다.

1. 설치 전 O/S 설정 진행
2. kubeadm 으로 토큰 생성
3. 토큰으로 생성된 명령어로 노드 2 번 클러스터에 추가

## 15. Service HealthCheck

### 상태 확인 소개(HealthCheck)

쿠버네티스에서 Pod 를 구성하면 컨테이너 기반으로 서비스가 시작이 된다. 쿠버네티스 기반에서 동작하는 서비스가 올바르게 동작하는지 확인하기 위해서는 아래와 같은 방법으로 일반적으로 확인한다.

- kubectl logs
- kubectl describe
- kubectl port-forward

- kubectl exec

위에서 'logs', 'describe' 하위 명령어로, 현재 동작중인 컨테이너 상태에 대해서 확인은 가능하다. 하지만, 실제 내부에서 동작중인 애플리케이션에 대해서는 확인이 어려운 부분이 있다.

'port-forward' 명령어는 서비스 중인 컨테이너에서 포트 포워딩을 통해서 전달이 되는데 이 방법은 하나의 컨테이너 및 서비스에 대해서 확인이 가능하지 전체 서비스 혹은 컨테이너에 대해서 확인이 어렵다. 그래서 서비스 상태를 좀 더 면밀하게 확인하기 위해서 POD 나 혹은 애플리케이션 상태를 확인해서 올바르게 동작하는지 확인하는 방법이 있다.

결론적으로 위의 logs, describe, port-forward 는 사람이 직접 확인을 해야 한다. 사람이 개입하지 않고 애플리케이션 및 서비스 상태를 확인하기 위해서 liveness, readiness 기능을 사용한다. 아래 그림은 컨테이너 서비스를 확인하기 위해서 애플리케이션 주소 혹은 POD 의 상태를 확인하여 올바르게 서비스가 되는지 확인 가능한 방법이 있다.

위의 두 개의 서비스는 기본적으로 kubelet, Pod 를 통해서 애플리케이션 상태를 확인한다. 실제로 사용하는 사용자는 복잡하게 설정할 필요 없이, 컨테이너 선언이 되는 YAML 부분에 명시만 해주면 된다. 뒤에서 실습하겠지만 선언 방법은 다음과 같이 선언한다.

```
spec:
  containers:
  - name: goproxy
    image: registry.k8s.io/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

Readiness, Liveness 는 아래와 같은 구조로 동작한다. 보통 kubelet 에서 Pod 를 생성한 애플리케이션 컨테이너 및 Pod 상태를 확인한다.

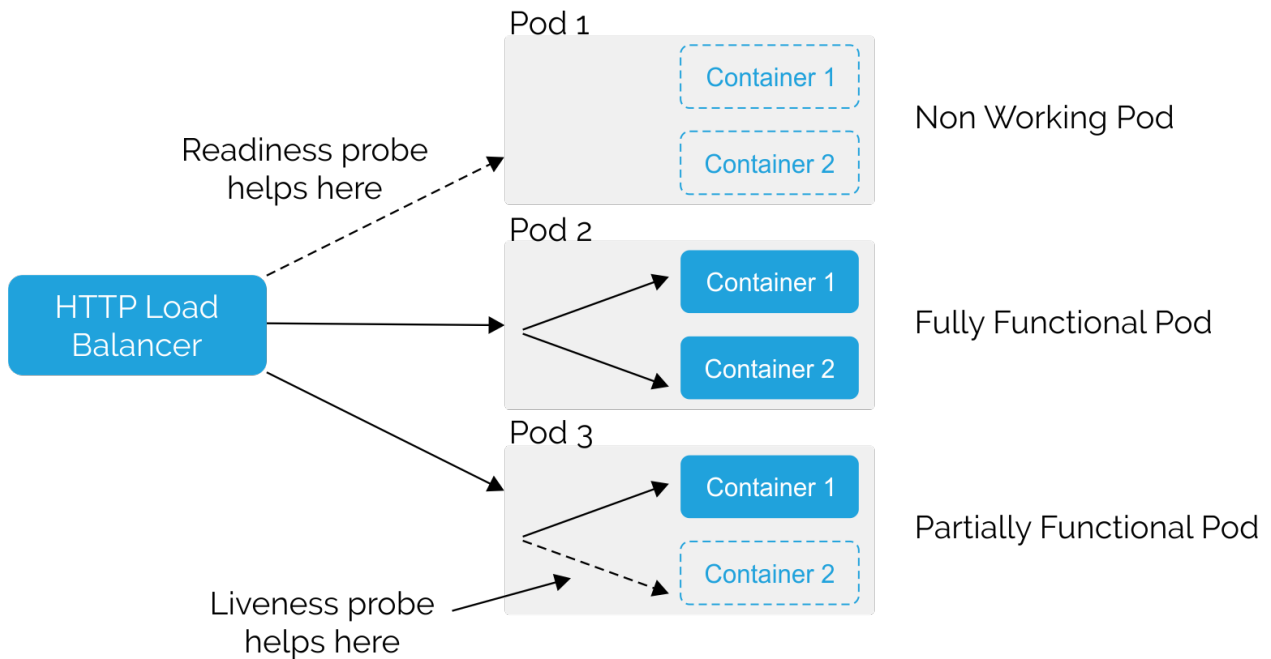


그림 46 쿠버네티스 Readiness, Liveness

위의 기능을 사용하기 위해서 POD 나 컨테이너에서 조회하는 URL 위치가 있다.

1. /ready: "readinessProbe"는 아래 "livenessProbe"와 동작 방식은 비슷하다.
2. /healthy: "livenessProbe"가 바라보는 위치이다. 이 위치는 파일이 될 수도 있고 주소가 될 수도 있다.
3. /status

liveness, readiness 는(은) 총 3 가지의 "probes" 형식은 다음과 같이 지원한다.

1. httpGet
2. tcpSocket
3. exec

readiness, liveness 는 같이 사용이 가능하며 동작 방식에 대한 우선 순위가 없다. 차이점은 다음과 같다.

1. readiness 의 "**probes**"는 컨테이너 시작부터 종료까지 모든 라이프 사이클 따라간다.
2. liveness 의 "**probes**"는 readiness probe 가 성공할 때까지 기다리지 않는다.

liveness 는 애플리케이션이 올바르게 동작하면 명시된 주소, 예를 들어서 /healthy 같은 위치에 접근해서 확인한다. 그래서 만약, liveness 가 시작 후 동작하기 위해서는 다음과 같이 설정한다.

```
readinessProbe:
  exec:
    command:
      - cat
      - /tmp/healthy
```



```
initialDelaySeconds: 5
periodSeconds: 5
```

## 애플리케이션 상태 확인(Liveness)

"liveness"는 컨테이너에서 사용하는 애플리케이션 서비스의 상태를 확인한다. 컨테이너에서 사용하는 애플리케이션에서 상태를 확인할 수 있도록 하위 주소를 명시한다. liveness 는 보통 "httpGet"라는 도구를 통해서 컨테이너에 명시된 하위 주소를 통해서 접근하여, 올바르게 동작하는지 확인한다. "liveness"가 동작하게 되면 "goproxy"가 내부적으로 실행이 되면서 컨테이너의 8080 포트로 연결이 된다.

추가적으로 liveness 는 "gRPC liveness probe"도 지원한다

사용 방법은 아래와 같다. 이 예제는 쿠버네티스 사이트에서 제공해주는 예제이다. 이름은 liveness-grpc.yaml 로 저장한다.

```
master]# cat <<EOF> liveness-grpc.yaml

apiVersion: v1
kind: Pod
metadata:
  name: etcd-with-grpc
spec:
  containers:
    - name: etcd
      image: registry.k8s.io/etcd:3.5.1-0
      command: [ "/usr/local/bin/etcd", "--data-dir", "/var/lib/etcd", "--listen-client-urls", "http://0.0.0.0:2379", "--advertise-client-urls", "http://127.0.0.1:2379", "--log-level", "debug"]
      ports:
        - containerPort: 2379
      livenessProbe:
        grpc:
          port: 2379
          initialDelaySeconds: 10
EOF
master]# kubectl apply -f liveness-grpc.yaml
master]# kubectl log etcd-with-grpc
```

grpc 을(를) 통해서 실행된 etcd 컨테이너는 쿠버네티스의 내부 etcd 서버에 2379 포트로 접근 및 확인한다. 일반적인 "httpGet"를 사용해서 서비스를 확인한다. logs 명령어로 확인하면, 컨테이너가 etcd 서버에 연결하여 조회한 결과가 보인다. 즉, livenessProbe 를 통해서 2379 포트로 접근이 가능하다고 판단 후 etcd 클라이언트 명령어를 실행했다.

아래는 쿠버네티스에서 미리 배포하는 liveness 컨테이너 이미지이다. 해당 이미지를 가지고 어떤식으로 liveness 가 동작하는지 확인한다. 이름은 "liveness-http.yaml"로 저장한다.

```
master]# cat <<EOF> liveness-http.yaml

apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
EOF
master]# kubectl apply -f liveness-http.yaml
```

동작상태를 아래와 같이 명령어를 통해서 확인한다.

```
master]# kubectl describe pod liveness-http
```

**Events:**

Type	Reason	Age	From	Message
Normal	Scheduled	60s	default-scheduler	Successfully assigned default/liveness-http to master.example.com
Normal	Pulled	50s	kubelet	Successfully pulled image "k8s.gcr.io/liveness" in 10.182s (10.182s including waiting)
Normal	Created	25s (x2 over 50s)	kubelet	Created container liveness
Normal	Started	25s (x2 over 50s)	kubelet	Started container liveness
Normal	Pulled	25s	kubelet	Successfully pulled image "k8s.gcr.io/liveness" in 8.144s (8.144s including waiting)
Normal	Pulling	6s (x3 over 60s)	kubelet	Pulling image "k8s.gcr.io/liveness"
Warning	Unhealthy	6s (x6 over 39s)	kubelet	Liveness probe failed: HTTP probe failed with statuscode: 500
Normal	Killing	6s (x2 over 33s)	kubelet	Container liveness failed liveness probe, will be restarted

위의 livenessProbe 에서 중요한 부분은 아래 조건이다. 총 3 번 실패하면, 해당 컨테이너(혹은 Pod)는 서비스를 중지하게 된다. 이 조건을 확인하는 포트는 8080 서비스 포트이다.

**livenessProbe:**

failureThreshold: 3

**httpGet:**

path: /healthz

port: 8080

위의 컨테이너가 사용하는 애플리케이션의 "/healthz"는 다음처럼 구성이 되어 있다.

```
http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
    duration := time.Now().Sub(started)
    if duration.Seconds() > 10 {
        w.WriteHeader(500)
        w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds())))
    } else {
        w.WriteHeader(200)
        w.Write([]byte("ok"))
    }
})
```

tcp 기반으로 사용을 원하는 경우 아래처럼 구성을 한다. 이름은 liveness-tcp.yaml 로 저장한다.

```
master]# cat <<EOF> liveness-tcp.yaml

apiVersion: v1
kind: Pod
metadata:
  name: liveness-tcp
spec:
  containers:
  - name: redis
    image: redis
    # defines the health checking
    livenessProbe:
      # a TCP socket probe
      tcpSocket:
        port: 6379
      # length of time to wait for a pod to initialize
      # after pod startup, before applying health checking
      initialDelaySeconds: 30
      timeoutSeconds: 1
    ports:
    - containerPort: 6379
EOF
master]# kubectl create -f liveness-tcp.yaml
master]# kubectl describe pod/liveness-tcp
```

위의 내용은 tcpSocket 의 6379 포트를 확인한다. 아래 initialDelaySeconds, timeoutSeconds 를 통해서 POD 가 시작하는 중 발생하는 대기 시간에 대해서 설정한다.

## 포드 상태확인(Readiness )

Liveness와 비슷한 기능을 가지고 있으나, 이 기능은 Pod 영역에서 동작한다. Pod 가 시작이 되면서 status 정보를 가지게 되는데, 이 정보를 Readiness 통해서 결정할 수 있다. 즉, 애플리케이션이 올바르게 동작이 되면, Pod 는 서비스에 문제가 없다고 판단을 하고, 올바르게 동작하지 않으면, Pod 에서는 애플리케이션 실행을 실패로 간주한다.

"readinessProbe"도 "tcp", "http"두 가지 방식을 제공한다. 아래는 "readiness probe"중에서 "httpGet"예제이다. 이름은 "readiness-http.yaml"으로 저장한다.

```
master]# cat <<EOF> readiness-http.yaml
```

```
apiVersion: apps/v1
```

```
kind: Deployment
```

```
metadata:
```

```
  name: readiness-http
```

```
  labels:
```

```
    app: nginx
```

```
spec:
```

```
  replicas: 1
```

```
  selector:
```

```
    matchLabels:
```

```
      app: nginx
```

```
  template:
```

```
    metadata:
```

```
      labels:
```

```
        app: nginx
```

```
    spec:
```

```
      containers:
```

```
        - name: nginx
```

```
          image: nginx
```

```
          ports:
```

```
            - containerPort: 80
```

```
          readinessProbe:
```

```
            initialDelaySeconds: 1
```

```
            periodSeconds: 2
```

```
            timeoutSeconds: 1
```

```
            successThreshold: 1
```

```
            failureThreshold: 1
```

```
            httpGet:
```

```
              host:
```

```

    scheme: HTTP
    path: /
    httpHeaders:
      - name: Host
        value: exampleapp.com
    port: 80
EOF
master]# kubectl apply -f readiness-http.yaml
master]# kubectl describe pod readiness-http

Readiness:      http-get http://:80/ delay=1s timeout=1s period=2s
#success=1 #failure=1

```

위의 서비스를 구성한 다음에, `kubectl describe` 명령어를 통해서 Pod 가 올라오면서 어떠한 이벤트가 발생하는지 확인한다. 일반적으로 Readiness 라는 항목에 동작이 올바르게 되었는지 확인이 가능하다.  
잘 구성이 되었는지 확인하기 위해서 `readiness-probe-svc.yaml` 파일을 생성한다.

```

master]# cat <<EOF> readiness-probe-svc.yaml

apiVersion: v1
kind: Service
metadata:
  labels:
    app: readiness-probe
    name: readiness-probe-svc
    namespace: default
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
    - name: readiness-probe-port
      port: 80
  selector:
    app: nginx

```

```
    sessionAffinity: None
EOF
master]# kubectl apply -f readiness-probe-svc.yaml
```

실행이 될 때 올바르게 'kubectl describe'를 통해서 "Readiness"가 올바르게 동작하는지 확인한다.

```
Readiness: http-get http://:80/ delay=1s timeout=1s period=2s #success=1
#failure=1
```

정상적으로 서비스가 올라오면 "End Point(엔드 포인트)"가 잘 구성이 되었는지 확인한다.

```
master]# kubectl get endpoints
NAME                      ENDPOINTS                                AGE
kubernetes                192.168.10.250:6443                    86m
php-apache                10.85.0.5:80,10.85.0.6:80
76m
readiness-probe-svc      10.85.0.10:80                          29s
```

마지막으로 readiness, liveness 를 동시에 사용하기 위해서는 다음처럼 구성한다. 파일 이름은 readiness-liveness-readiness.yaml 으로 저장한다.

```
master]# cat <<EOF> liveness-readiness.yaml

apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
    - name: goproxy
      image: registry.k8s.io/goproxy:0.1
      ports:
        - containerPort: 8080
```

```
readinessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 10
livenessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 15
  periodSeconds: 20
EOF
master]# kubectl apply -f readiness-liveness.yaml
```

최종적으로 readiness, liveness 의 동작 방식은 및 순서는 다음과 같다.

1. Readiness 는 kubelet, Pod 를 통해서 애플리케이션 컨테이너 동작 상태를 확인한다. 이를 통해서 애플리케이션 컨테이너가 올바르게 동작하지 않으면 로드밸런서를(내부)에 연결하지 않는다.
2. Liveness 는 kubelet 를 통해서 컨테이너의 애플리케이션이 올바르게 시작되는지 확인한다. 만약, 올바르게 애플리케이션이 실행이 되지 않으면 계속 지속적으로 재시작 혹은 대기 상태로 남을 수도 있다.

위의 설정을 보면 알겠지만, liveness, readiness 두 개의 probe 가 실행된다. 먼저 POD 에서 컨테이너 상태를 확인해야 되기 때문에 "readiness"가 먼저 시작하고, 그 다음에 "liveness"가 실행이 된다.

## 16. Label, annotations, selectors

쿠버네티스에서 자원 생성시 분류 및 추가적인 메타정보(metadata)를 제공하는 방법은 label, annotations 그리고 selector 가 있다. 메타정보는 실제 작업에는 영향을 주지는 않지만, 메타정보를 통해서 자원 구성 시 추가적인 정보를 제공할 수 있다. 각각 자원 정보에 대해서는 아래 내용을 참고한다.



## 이름표(label)<sup>27</sup>

레이블은 생성되는 자원에 구분하기 위한 구분자를 구성한다. 구분자(label)은 특별한 기능이 아니라 자원 구별을 위한 **메타데이터(metadata)**로 사용한다.

이를 통해서 POD, Deployment 에 생성된 자원에 **선택자(selector)**를 구성할 수 있다. 기본적으로 레이블은 키 기반으로 쌍으로 구성이 되어 있으며, 키에는 항상 키 값이 같이 따라온다. 이를 보통 키페어(keypair)라고 부른다. 키페어는 YAML 이나 혹은 명령어로 선언이 가능한데 어떤 위치에서 사용하느냐 따라서 조금씩 문법이 다르다. 일반적으로 YAML 에서는 다음처럼 선언한다

```
keyname: <key value>
```

명령어에서 처리시 일반적으로 다음처럼 명령어를 사용한다

```
master]# kubectl --label="keyname=<key value>"
```

한 개 이상의 레이블 즉, 구분자를 사용하는 경우 쉼표로 여러 개의 구분자 선언이 가능하다.

구분자는 애플리케이션이나 설정파일 구성 시 명시가 가능하며, 일반적으로 label:라는 지시자를 사용하여 명시한다. 명시하는 방법은 다음과 같이 한다.

```
master]# kubectl create --label="ver=2"
```

혹은 여러 개의 구분자를 명시할 때는 다음과 같은 방법으로 구성한다.

```
master]# kubectl create --label="ver=2, env=prod"
```

혹은 YAML 파일 작성 시, 직접적으로 레이블 명시가 가능하다.

```
master]# cat <<EOF> test-label1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    enviroment: production
    app: nginx
```

---

<sup>27</sup> 한국어로 이름표라고 적으면 매우 어색하다.

```
spec:
  containers:
  - name: label-nginx
    image: nginx
    ports:
    - containerPort: 80
EOF
master]# kubectl apply -f test-label1.yaml
```

위의 내용에서 중요한 부분은 “metadata”부분이다. metadata 는 labels 라는 속성을 가지고 있으며, 이를 통해서 선택자 즉, selector 를 생성한다.

만약 명령어를 통해서 레이블 추가를 하는 경우 다음과 같이 레이블 추가가 가능하다.

```
master]# kubectl label pods label-demo auth=choigookhyun
master]# kubectl describe pod label-demo
...
Labels:          app=nginx
auth=choigookhyun
enviroment=production
...
```

위와 같이 이미 사용중인 Pod 에 레이블 추가가 가능하다. 다른 자원도 위와 비슷한 방법으로 레이블 추가가 가능하다.

## 연습문제

다음과 같은 자원을 생성 후, 레이블을 추가한다.

1. apache-label 라는 Pod 를 생성한다
2. 해당 자원에 레이블을 아래와 같이 추가한다
  - name: 본인 이름
  - age: 본인 나이
3. 올바르게 추가가 되면 describe 명령어로 레이블이 올바르게 추가가 되었는지 확인한다

## 선택자(selector)

selector(선택자)라는 이름으로 부르기도 하며, 이를 통해서 생성된 자원을 선택할 수 있다. 이 선택자가 필요한 이유는 쿠버네티스는 클러스터를 통해서 전반적으로 모든 자원을 구성하기 때문에 최소 한 개 이상의 선택자를 구성하는 것을 권고한다. 선택자는 실제로는 위에서 생성한 구분자(label)로 구별이 되기 때문에 구분자 정보가 들어간 자원이 구성이 되면, 다음처럼 쉽게 선택자를 통해서 선택이 가능하다.

YAML 기반으로 작성시에 선택자는 보통 다음처럼 구성이 되어 있다.

```
"metadata": {  
  "labels": {  
    "key1" : "value1",  
    "key2" : "value2"  
  }  
}
```

구성된 구분자를 통해서 자원을 선택하는 경우, 아래처럼 YAML 를 작성한다. 위에서 사용한 label 를 기반으로 다시 활용해서 확인을 해보도록 한다. 파일 이름은 label-demo.yaml 으로 저장한다.

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: label-demo  
  labels:  
    environment: production  
    app: nginx  
spec:  
  containers:  
    - name: nginx  
      image: nginx:1.14.2  
      ports:  
        - containerPort: 80
```

간단하게 nginx 기반으로 포드 및 컨테이너를 생성한다. 작성한 YAML 파일을 'kubectl create -f'명령어로 적용한다. 위의 YAML 에서 생성한 구분자의 정보는 다음과 같다.

```
labels:
```

```
environment: production
app: nginx
```

“name: label-demo”는 같은 메타정보이지만, 이 메타정보는 포드 생성시 사용하는 이름이다. 구분자는 “labels”밑으로 구성된 부분이 구분자이다. 이 구분자는 다음과 같은 조건식을 사용할 수 있다.

- `=:` 해당 키 이름이 값이 같은 경우 적용
- `!=:` 해당 키 이름이 값이 다른 경우 적용

위에서 작성한 nginx 애플리케이션을 서비스(service)를 통해서 외부에 서비스할 수 있도록 서비스 자원을 구성한다. 파일 이름은 label-nginx-service.yaml 으로 저장한다.

```
apiVersion: v1
kind: Service
metadata:
  name: label-nginx-service
spec:
  selector:
    app: nginx
    environment: production
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
```

## 연습문제

다음과 같이 deployment 를 생성 및 레이블을 추가한다.

1. deployment 이름은 selector-nginx 라고 생성 및 구성한다.
2. 이미지는 nginx 를 사용한다.
3. 레이블은 virtualization\_type: rhv 으로 추가한다
4. 레이블은 environment: test 으로 추가한다.

올바르게 추가가 되었는지 describe 명령어로 확인한다. 추가가 완료 되었으면, 선택자를 통해서 자원을 확인한다.

1. kubectl 명령어를 통해서 구성된 Pod 를 검색한다
2. 반드시 virtualization\_type, environment 이 두개 키를 통해서 검색한다.

## 주석(annotations)

annotations 기능은 labels 과 비슷하지만, 다른 부분은 프로그램들이 자원을 생성하면서 annotations 통해서 추가 정보를 추가한다.

이 기능을 통해서 좀 더 넓은 범위에서 변수 형태로 사용이 가능하다. annotations 은 label 과 같이 keypair 형태로 구성이 되어 있으며, annotations 영역안에서 사용자가 원하는 형태로 구성이 가능하다. 아래는 특정 이미지를 어느 위치에서 가져왔는지 명시한다. 대표적인 도구는 Docker, Kubernetes 에서 특정 자원을 생성하면 아래와 같이 annotations 를 생성한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: annotations-demo
  annotations:
    imageregistry: "https://hub.docker.com/"
spec:
  containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80
```

위의 annotation 은 레이블과 동일하게, metadata 영역안에 구성이 된다. 여기에는 Pod 에서 사용하는 이름인 "annotations-demo", annotations 에는 imageregistry 에 "https://hub.docker.com"이라고 명시가 되어있다. annotations 에 선언이 되어 있는 내용들은 annotations-demo POD 와 관련되어 있는 자원들은 접근해서 사용이 가능하다.

사용방법은 다음과 같다. 아래 예제는 쿠버네티스에서 컨테이너 이미지를 생성시 사용하는 컨테이너 이미지를 내려 받는 서버의 주소 정보를 변경한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: annotations-demo
```

```
  annotations:
    imageregistry: "https://quay.io/tangt64/"
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

## 연습문제

다음 조건으로 annotations 정보를 생성 및 구성한다.

1. 이미지는 vsftpd 를 사용한다
2. 해당 이미지는 service: secureftp 라는 annotations 를 남긴다
3. 올바르게 구성이 되면 describe 를 통해서 확인이 가능하다

## 노드 셀렉터(Node Selector)

노드 선택자(selector)는 단어 그대로 노드를 선택 시 사용하는 선택자이다. 노드 선택자는 레이블과 비슷하지만, 자원이 아닌 노드에 레이블 설정하여 자원을 사용할 수 있도록 한다. 즉 selector -> label 의 명시된 정보를 검색한다. 노드 셀렉터 사용하는 방법은 다음과 같다.

```
master]# kubectl label nodes node2 ssd=true
```

위의 명령어는 쿠버네티스 노드 2 번에 "ssd=true"라는 레이블을 설정한다. 위와 같이 설정한 레이블은 선택자(셀렉터)를 통해서 선택이 가능하다. 시스템에 적용하면 다음과 같이 내용이 적용이 된다.

```
master]# kubectl get nodes
NAME                STATUS    ROLES    AGE   VERSION
master.example.com  Ready    master   10d   v1.18.3
node1.example.com   Ready    <none>   10d   v1.18.3
node2.example.com   Ready    <none>   10d   v1.18.3
node3.example.com   Ready    <none>   10d   v1.18.3

master]# kubectl label node node-2.example.com "ssd=true"
```

```
node/node2.example.com labeled
```

```
master]# kubectl get nodes --selector "ssd=true"
NAME                                STATUS    ROLES    AGE    VERSION
node2.example.com    Ready    <none>    10d    v1.18.3
```

아래 명령어로 여러 노드 중에서 "ssd=true"라고 설정이 되어 있는 노드를 선택하여 화면에 결과를 출력해준다.

```
master]# kubectl get nodes --selector ssd=true
```

위의 내용을 실제 시스템에서 적용하면 다음과 같은 화면이 출력이 된다.

```
master]# kubectl get nodes --selector "ssd=true"
NAME                                STATUS    ROLES    AGE    VERSION
node2.example.com    Ready    <none>    10d    v1.18.3
```

## 연습문제

- node2 번에 다음과 같이 레이블을 추가한다
  - GPU=nvidia
- node1 번에는 다음과 같이 레이블을 추가한다.
  - GPU=amd

## 17. 데몬 서비스(DaemonSet)

데몬 서비스는 모든 노드에 특정한 애플리케이션을 동작 시, 사용한다. 보통 사용하는 용도는 모니터링이나 혹은 지속적으로 관리 및 생성해야 되는 자원이 있는 경우, 데몬 서비스를 통해서 작업을 수행한다. 데몬 서비스는 다음과 같은 조건으로 동작한다.

1. 모든 클러스터에서 스토리지 데몬 서비스가 동작(Ceph, Glusterfs)
2. 클러스터에서 동작하는 모든 노드의 로그 정보 수집(vmstat, iostat)
3. 특정 데몬의 동작 상태를 확인(NetworkManager, sshd)

예를 들어서 **노드 상태(problem-detector)**를 확인해주는 서비스는 다음과 같이 설정한다. 아래 내용을 적용하면 컨테이너 서비스가 생성이 되면서 클러스터의 모든 노드에 서비스가 구성이 된다.

```
master]# cat <<EOF> daemonset-node-health-detector.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: node-problem-detector
      version: v0.1
      kubernetes.io/cluster-service: "true"
  template:
    metadata:
      labels:
        k8s-app: node-problem-detector
        version: v0.1
        kubernetes.io/cluster-service: "true"
    spec:
      hostNetwork: true
      containers:
        - name: node-problem-detector
          image: registry.k8s.io/node-problem-detector:v0.1
          securityContext:
            privileged: true
          resources:
            limits:
              cpu: "200m"
              memory: "100Mi"
            requests:
              cpu: "20m"
              memory: "20Mi"
          volumeMounts:
            - name: log
```



```

        mountPath: /log
        readOnly: true
volumes:
- name: log
  hostPath:
    path: /var/log/
EOF
master]# kubectl apply -f daemonset-node-health-detector.yaml
master]# kubectl -n kube-system get pods -l k8s-app=node-problem-detector
node-problem-detector-v0.1-c686q          1/1      Running
node-problem-detector-v0.1-wnpqj          0/1      ContainerCreating

```

위의 자원을 등록하면, 사용중인 모든 컨트롤러 및 컴퓨터 노드에서 registry.k8s.io/node-problem-detector:v0.1 애플리케이션이 동작한다. 동작이 되면서 노드의 상태를 /log 디렉터리 위치에 저장한다. 좀더 쉽게 구현하면 다음과 같이 구현이 가능하다. 아래는 vmstat 명령어를 사용하여 기록을 수집한다.

```

master]# cat <<EOF> daemonset-vmstat.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: vmstat-log
  namespace: kube-system
  labels:
    app: vmstat-log
spec:
  selector:
    matchLabels:
      app: vmstat-log
      kubernetes.io/os: linux
  template:
    metadata:
      labels:
        app: vmstat-log
        kubernetes.io/os: linux
    spec:
      containers:
        - name: centos-vmstat
          image: centos

```

```
command: ["vmstat"]
args: ["-t", "-w", "1"]
securityContext:
  privileged: true
```

EOF

```
master]# kubectl apply -f daemonset-vmstat.yaml
```

```
master]# kubectl -n kube-system get pod -l app=vmstat-log
```

NAME	READY	STATUS	RESTARTS	AGE
vmstat-log-cf22k	1/1	Running	0	12m
vmstat-log-gcsgg	0/1	Terminating	0	30m

```
master]# kubectl logs -f vmstat-log-xxx -n kube-system
```

## 연습문제

모든 서버에 다음과 같이 DaemonSet 를 적용한다.

1. 클러스터의 모든 컴퓨트 노드의 디스크 정보를 수집한다
2. 수집 시 iostat 명령어를 사용한다
3. 수집된 정보는 logs 명령어로 확인이 가능하다
4. 레이블은 아래 두 개를 사용한다

```
kubernetes.io/os=linux
```

```
app=disk-stat
```

## 18. 상태 POD 서비스 (StatefulSets)

## 19. 작업 (Jobs/CronJobs)

### 작업(Jobs)

Jobs 시스템의 crond 처럼, 반복적인 작업을 처리시에 사용하는 기능이다. 예를 들어서 일정시간에 특정 컨테이너가 실행되면서 알림 및 혹은 컨테이너 상태를 확인하는 용도로 사용이 가능하다. 시스템에 등록이 되어 있는 jobs 를 확인하려면 다음 명령어로 확인이 가능하다. 확인을 하기 위해서 Jobs 자원을 하나 생성한다.

```
master]# cat <<EOF> jobs-test.yaml
```

```

apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl:5.34.0
        command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
        restartPolicy: Never
      backoffLimit: 4
EOF
master]# kubectl apply -f jobs-test.yaml

```

위의 작업을 명령어로 실행하면 다음과 같이 생성이 가능하다.

```

master]# kubectl create job pi --image=busybox -- "perl -Mbignum=bpi -wle
print bpi(2000)"

```

아래는 job 으로 등록된 서비스는 아래와 같이 동작한다. 조금 기다리면 작업 수행이 완료가 된다.

```

master]# kubectl get jobs -w

```

NAME	COMPLETIONS	DURATION	AGE
pi	0/1	64s	64s
pi	1/1	2m8s	2m38s

자세한 작업 내용을 확인하려면, describe 및 logs 명령어를 통해서 상세한 jobs 내용 확인이 가능하다.

```

master]# kubectl describe jobs pi
Name:                pi
Namespace:           np-nginx
Selector:             batch.kubernetes.io/controller-uid=95782465-95f2-4be2-
b0a4-ad1f39a5a404
Labels:
batch.kubernetes.io/controller-uid=95782465-95f2-4be2-b0a4-ad1f39a5a404
batch.kubernetes.io/job-name=pi

```

```
controller-uid=95782465-95f2-4be2-b0a4-ad1f39a5a404
job-name=pi
master]# kubectl logs pi-5jbtq
3.14159265358979323846264338327950288419716939937510582097494459230781640
6286208998628034825342117067982148...
```

## 연습문제

위와 같이 간단하게 일시적으로 작업 수행이 필요한 경우 Jobs 를 통해서 수행한다. Jobs 경우에는 불 필요한 자원이나 혹은 일시적으로 실행 혹은 기록을 남겨야 하는 경우 많이 사용한다.

## 크론잡(CronJobs, cj)

CronJobs(이하 크론잡)은, Jobs 과 비슷하지만, 일반적으로 유닉스에서 사용하는 crontab 과 동일한 방법으로 동작한다. 특정 이미지 기반으로 프로그램 실행 혹은 명령어를 주기적으로 실행한다

작업 주기는 유닉스나 리눅스에서 사용하는 방법과 동일하다. 작업을 스케줄러에 등록 시 다음과 같은 순서대로 작업을 예약하면 된다.

첫번째(분)	두번째(시간)	세번째(일날)	네번째(월)	(요일)
0~59 분	0~23 시	1~31 일	1~12 월 혹은 영어 달력 사용 가능	0 과 7 은 일요일 1~6 은 월~토요일 혹은 영어 달력 사용 가능

위의 내용 기준으로 아래처럼 작성해서 크론잡에 등록한다

```
master]# cat <<EOF> first-cronjob.yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: first-cronjob
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
```

```

- name: first-cronjob
  image: busybox
  imagePullPolicy: IfNotPresent
  command:
  - /bin/sh
  - -c
  - date; echo Hello World
  restartPolicy: OnFailure

```

EOF

```
master]# kubectl apply -f first-cronjob.yaml
```

위의 작업을 명령어로 예약이 가능하다.

```
master]# kubectl create cronjob first-cronjob --image=busybox --
restart=OnFailure --schedule="* * * * *" -- /bin/sh -c date ; echo Hello
World
```

위의 설정 중 schedule 를 보면, 시스템에서 사용하는 크론잡과 동일한 형식과 설정 방식을 사용하고 있다. 그 이외 나머지 부분은 Pod 설정과 비슷하게 구성한다. 등록이 완료가 되면 자원이 올바르게 구성이 되었는지 확인한다.

```
master]# kubectl get cronjobs -w
```

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
first-cronjob	* * * * *	False	0	<none>	9s

동작이 되면 다음과 같이 시스템에 기록이 남는다. 실제로 작업이 수행이 완료가 되면, 시스템 로그도 같이 남는다.

```
master]# kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
first-cronjob-28208666-whc2w	0/1	Completed	0	2m22s
first-cronjob-28208667-bkp89	0/1	Completed	0	82s
first-cronjob-28208668-rjrh4	0/1	Completed	0	22s

```

master]# kubectl logs first-cronjob-28208671-6lvrk
Sun Aug 20 08:31:00 UTC 2023
Hello World

```

크론잡은 노드 상태 관련 보고서나 혹은 주기적으로 실행하여 자원 상태 모니터링이 필요할 때 사용한다.

## 연습문제

문제 1. Jobs 를 사용하여 간단한 작업을 수행한다.

1. 이미지는 centos 나 혹은 buysbox 를 사용한다
2. ping 명령어를 사용하여 google.com 에 ICMP 메시지를 발송한다.
3. command: ["ping", "yahoo.com"] 참조

문제 2. CronJobs 를 사용하여 시스템 기록을 조회한다.

1. 이미지는 centos 나 혹은 buysbox 를 사용한다
2. watch 명령어를 사용하여 시스템 정보를 수집한다.
3. command: ["vmstat", "-t 1 5"] 참조

## 20. 설정파일(ConfigMaps(cm))

ConfigMaps 컨테이너나 혹은 애플리케이션이 사용하는 설정파일 내용을 저장 및 보관한다. 이를 통해서 모든 컨테이너에 설정 파일을 배포할 필요가 없으며, ConfigMaps 를 통해서 일괄적으로 배포 및 갱신이 가능하다. ConfigMaps 는 두 가지 접근 방법을 지원한다. 첫 번째는 쉘 변수로 전달하는 방법, 두 번째는 파일 시스템 형태로 전달하는 방법이다. 아래 그림은 두 가지 방식을 나타내고 있다.

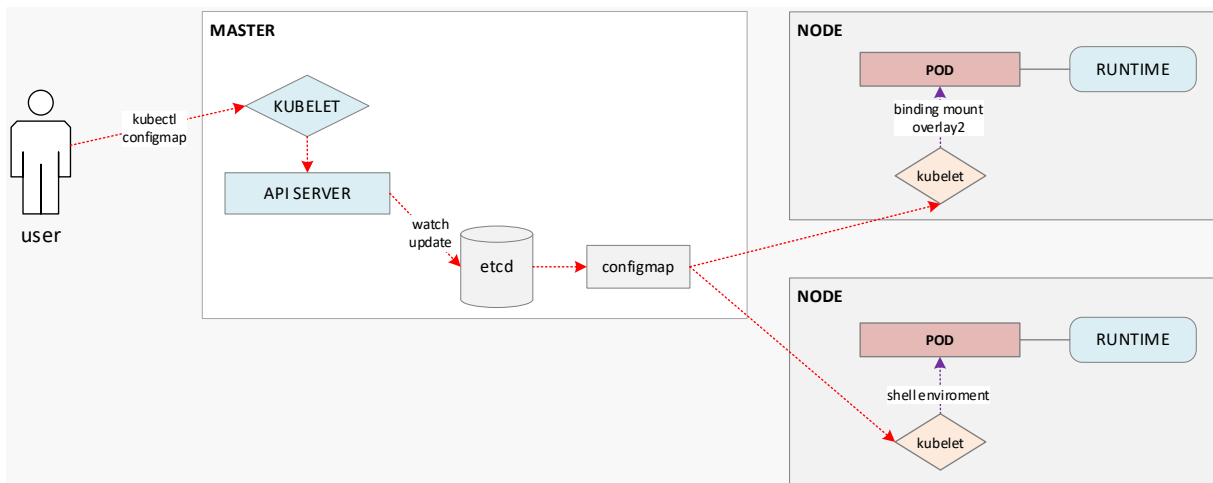


그림 47 configmap 구성

ConfigMaps 은 키=값 형태로 구성이 된다. 아래 예제를 확인한다.

```
master]# kubectl create configmap
```

현재 구성이 되어 있는 configmap 정보를 확인한다. configmap 은 실제로 cm 라는 약어로 사용이 가능하다. kubectl 명령어로 configmap 객체를 생성한다.

```
master]# kubectl create cm test
```

위의 명령어를 test 라는 configmap 자원객체를 생성한다. 생성한 자원객체는 데이터가 없기 때문에, 명령어에서 자원에 데이터를 입력하려면 다음처럼 명령어를 실행한다.

```
master]# kubectl create cm test2 --for-literal=name=bora
```

위의 명령어는 configmap 에 test2 라는 객체를 생성 후, 내부에 name 이라는 키 이름을 생성 후 데이터를 bora 를 대입한다. kubectl describe 명령어로 확인해본다.

```
master]# kubectl describe cm test2
Name:          test2
Namespace:     default
Labels:        <none>
Annotations:   <none>

Data
====
name:
----
bora
Events:  <none>
```

또한 데이터를 이미 구성된 정보 기반으로 configmap 에 입력이 가능하다. 디렉터리 구조는 상관없으나, 파일 이름 및 데이터는 다음과 같은 구조를 따라야 한다. 이 파일은 ui.properties 라는 이름으로 생성한다.

```
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

생성 후 kubectl 명령어로 다음처럼 실행한다.

```
master]# kubectl create configmap game-config --from-file .
```

명령어 실행 후 kubectl 명령어로 올바르게 데이터가 configmap 에 등록이 되어 있는지 확인한다.

```
master]# kubectl describe cm game-config
Name:          game-config
Namespace:     default
```

```
Labels:      <none>
Annotations: <none>

Data
====
ui.properties:
-----
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

올바르게 실행이 되면, 위의처럼 파일을 통해서 생성된 데이터가 확인이 된다. configmap 를 아래와 같은 이름으로 생성한다. 이 설정 파일은 nginx 이미지에 사용한다. 이름은 configmap-nginx-test.yaml 으로 저장한다.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configmap
data:
  database: mongodb
  database_uri: mongodb://localhost:27017

  keys: |
    image.public.key=771
    rsa.public.key=42
```

configmap 를 사용할 Pod 를 생성한다. 이름은 configmap-nginx-pod.yaml 으로 저장한다.

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-env-var
spec:
  containers:
```



```
- name: env-var-configmap
  image: nginx:1.7.9
  envFrom:
    - configMapRef:
        name: example-configmap
```

## 연습문제

다음 내용을 기존 내용에 추가하여 configmap 에 등록한다.

1. CentOS=RedHat
2. Ubuntu=Debian
3. Debian=GNU
4. Rocky=CentOS

## 21. 비밀(Secrets(sc))

시크릿은 configmap 과 비슷한 기능을 제공한다. 다만, 시크릿 경우에는 configmap 처럼 데이터를 일반 문자열 형태로 저장하지 않으며, 인코딩(base64)된 형태로 저장이 된다. 그러한 이유는 시크릿에 저장된 데이터는 일반 사용자들이 확인을 하면 안 되는 아이피 주소, 암호 혹은 아이디 같은 민감한 데이터를 저장한다.

쿠버네티스는 구성이 되면, 보통 한 개 이상의 시크릿 정보를 클러스터에서 가지고 있다. 이 정보는 쿠버네티스 운영에 필요한 정보들이다.

```
master]# kubectl get secrets
```

NAME	TYPE	DATA	AGE
default-token-mg4qm	kubernetes.io/service-account-token	3	15d

시크릿은 다양한 형태로 정보를 사용자에게 전달한다. 보통 다음과 같은 형식으로 전달한다.

- 볼륨 바인딩
- 시스템 변수 형태
- 이미지 기반 시크릿(Image pull secret)

앞서 이야기하였지만, 시크릿은 Pod 나 kubelet 를 통해서 자료를 전달한다. 아래 그림은 기본적으로 시크릿을 전달하는 방법이다.

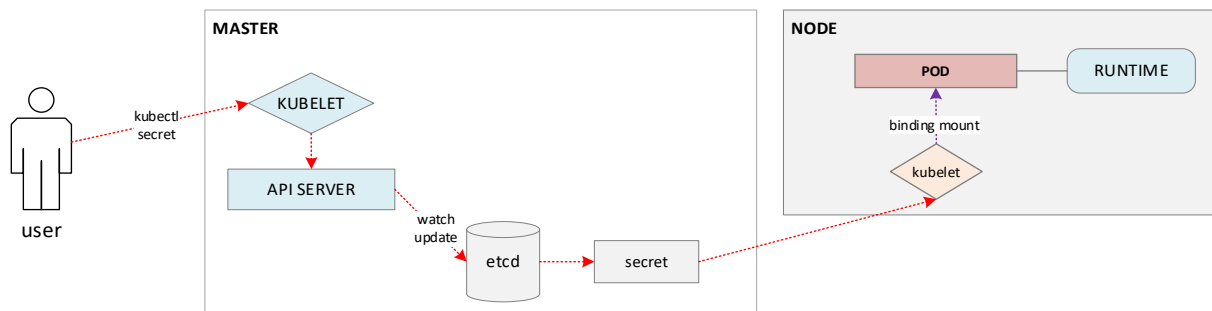


그림 48 시크릿 전달 방법

시크릿은 명령어로 생성이 가능하다. 일반적인 문자로 생성 시, 아래와 같이 사용한다.

```
master]# kubectl create secret generic name --from-
literal=username=choigookhyun
```

위의 명령어를 통해서 username 이라는 시크릿 이름을 생성한다. 생성 후 위에서 했던 것처럼, kubectl describe 를 통해서 확인한다. 시크릿은 저장 시 인코딩이 되어서 저장이 되기 때문에 내부의 자료는 아래처럼 보이지 않는다.

```
master]# kubectl describe secrets name
Name:          name
Namespace:     np-nginx
Labels:        <none>
Annotations:
<none>
Type:
Opaque
Data
====
username:  12 bytes
```

반대로 앞서 다루었던 configmap 경우에는 사용자가 입력한 데이터는 일반 문자열 형태로 출력이 되어서 확인이 가능하다. 시크릿 경우에는 저장된 문자열이 인코딩이 되어 저장이 된다. 시크릿에 저장되는 방식은 대략 다음과 같은 과정을 통해서 인코딩 후 저장이 된다.

반대로 시크릿에 저장된 내용을 디코딩하기 위해서 다음과 같이 명령어를 사용할 수 있다. 인코딩은 암호화가 아니기 때문에 손쉽게 다시 원상복구가 가능하다.

```
master]# echo -n 'redhat' | base64
cmVkaGF0
master]# kubectl get secrets/name --template={{.data.username}} | base64
-d
choigookhyun
```

여러 개의 시크릿 파일이 구성이 되어 있으면, 다음과 같이 파일을 생성 후 여러 개를 생성 및 구성하면 된다. 먼저 일반적으로 시크릿 파일을 작성하는 YAML 형태는 다음과 같다. 파일 이름은 secret-password.yaml 으로 저장한다.

```
master]# echo -n 'openshift' | base64
master]# echo -n 'redhat' | base64
master]# cat <<EOF> secret-password.yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-password
type: Opaque
data:
  username: b3BlbnNoawZ0
  password: cmVkaGF0
EOF
master]# kubectl apply -f secret-password.yaml
```

위의 비밀번호는 base64 명령어로 구성이 된 명령어이며, 아이디는 openshift 비밀번호는 redhat 으로 되어있다. YAML 형태로 구성된 시크릿을 사용하기 위해서는 kubectl create 명령어로 적용한다.

```
master]# kubectl create -f secret-password.yaml
```

올바르게 등록이 되었으면, kubectl describe 명령어로 최종 확인한다.

```
master]# kubectl describe secret/secret-password
Name:          secret-password
Namespace:     default
Labels:        <none>
Annotations:   <none>

Type:  Opaque

Data
====
password:  6 bytes
username:  9 bytes
```

앞에서 사용한 내용을 가지고 포드를 생성하여, 직접 secret 을 연결한다. 아래 내용을 secret-pod.yaml 으로 만들어서 저장한다.

```
master]# cat <<EOF> secret-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
spec:
  containers:
    - name: test-container
      image: nginx
      volumeMounts:
        - name: secret-volume
          mountPath: /etc/secret-volume
          readOnly: true
  volumes:
    - name: secret-volume
      secret:
        secretName: secret-password
EOF
master]# kubectl apply -f secret-pod.yaml
master]# kubectl get pods -w
```

올바르게 생성이 되면, 컨테이너 내부에서 "/etc/secret-volume"이 올바르게 구성이 되어 있는지 확인한다.

```
master]# kubectl describe pod secret-test-pod
...
Mounts:
/etc/secret-volume from secret-volume (ro)
/var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-946gh
(ro)
...
```

파일기반으로 시크릿을 생성하는 경우 명령어나 혹은 YAML 파일로 작성하는 방법이 있다. 만약 ssh 키를 전달한다면 다음과 같이 키를 생성 후 시크릿으로 전달 및 배포가 가능하다. 만약, TLS 키를 시크릿 통해서 배포하고 싶은 경우, generic 부분은 tls 로 변경하면 된다.

```
master]# ssh-keygen -t rsa -N'' -f test.key
```

```
master]# kubectl create secret generic my-ssh-keys \
  --from-file=test.key \
  --from-file=test.pub
master]# kubectl describe secret my-ssh-keys
Name:          my-ssh-keys
Namespace:     np-nginx
Labels:        <none>
Annotations:
<none>
Type:
Opaque
Data
====
test.key:  2610 bytes
test.pub:  577 bytes
```

위의 내용 기반으로 Pod 를 생성한다. 생성이 완료가 되면, 역시 올바르게 컨테이너에 시크릿 정보가 바인딩이 되었는지 확인한다.

```
master]# cat <<EOF> secret-ssh-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-ssh-pod
spec:
  containers:
  - name: secret-ssh-pod
    image: quay.io/centos/centos:stream9
    command: ["sleep"]
    args: ["10000"]
    volumeMounts:
    - name: secret-volume
      mountPath: /etc/ssh/keys
      readOnly: true
  volumes:
  - name: secret-volume
    secret:
      secretName: my-ssh-keys
```

EOF

```
master]# kubectl apply -f secret-ssh-pod.yaml
```

```
master]# kubectl get pods -w
```

```
master]# kubectl exec -it secret-ssh-pod -- ls -l /etc/ssh/keys
```

```
total 0
```

```
lrwxrwxrwx. 1 root root 15 Aug 20 10:40 test.key -> ../data/test.key
```

```
lrwxrwxrwx. 1 root root 15 Aug 20 10:40 test.pub -> ../data/test.pub
```

## 연습문제

컨테이너 이미지 centos 기반으로 다음과 같은 조건으로 시크릿을 구성한다.

1. 로컬 컴퓨터에 임의로 TLS 키를 만든다
2. TLS 키를 컨테이너 서버의 /var/www/html 에 전달한다
3. 시크릿의 이름은 secret-httpd-tls 으로 구성한다
4. 아파치 웹 서버 패키지(httpd)를 설치한다
5. 두 개 이상의 명령어를 동시에 실행하는 경우 command: ["/bin/sh","-c"], args: ["command 1; command 2 && command 3"] 를 통해서 구성한다.
6. describe, exec 를 통해서 올바르게 바인딩이 되었는지 확인한다.

## 22. 배포 및 배치(deployment)

쿠버네티스 서비스는 구성설정(deployment)를 통해서 구성이 된다. 컨테이너 기반으로 `kubectl run` 명령어로 간단하게 생성이 가능하지만, 반복적이고 지속적으로 애플리케이션 서비스 구성하기 위해서는 deployment 서비스가 필요하다. deployment 서비스는 replicaSet<sup>28</sup>과 함께 동작한다.

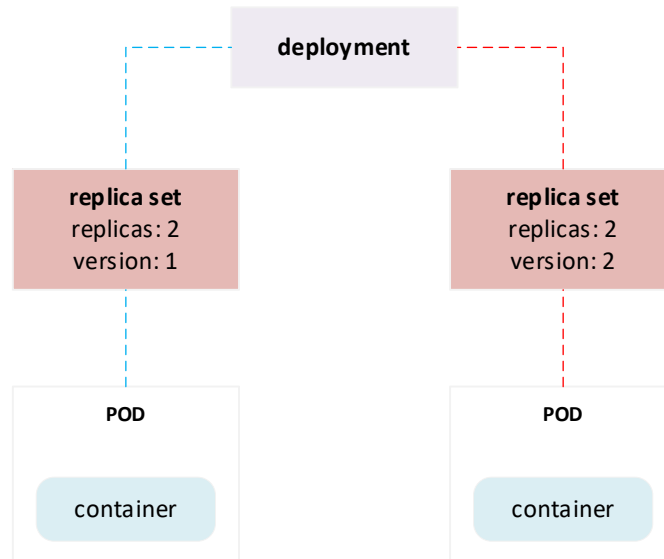


그림 49 디플로이먼트 동작

이 구성원은 다음과 같은 정보를 가지고 있다.

- 서비스에 사용할 POD 구성정보
- 복제할 POD 의 개수
- POD 및 컨테이너의 자원 할당
- POD 가 사용할 저장소 정보

일반적으로 많이 사용하는 deployment 설정은 아래와 같다. 여기에는 기본적인 deployment 에서 제공하는 포드, 컨테이너 그리고 선택자 및 리플리카(replica) 설정을 YAML 형식으로 하였다.

```
master]# cat <<EOF> nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
```

```

    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
EOF
master]# kubectl apply -f nginx-deployment.yaml

```

생성이 되면 올바르게 구성이 되었는지 kubectl 명령어로 확인한다. 생성이 문제 없이 되었으면, replicaset 도 같이 확인한다.

```

master]# kubectl get deployments -o wide
master]# kubectl get rs

```

올바르게 구성 및 동작이 되고 있으면, 다음 명령어로 이미지 버전을 변경한다. 이미지 정보는 deployment 에 등록이 되어 있으니 다음과 같은 방법으로 nginx-deployment 에서 사용하는 이미지 버전 정보를 변경한다.

```

master]# kubectl set image deployment nginx-deployment nginx=nginx:1.8
master]# kubectl describe pod nginx-deployment-86dcfdf4c6-zxfhl
...
Image:          nginx:1.8
...

```

기존에 사용하던 이미지 1.7.9 버전에서 1.8 로 변경한다. 하위 명령어 set 를 사용하여 적용되자마자 바로 이미지는 갱신이 된다. 아래 명령어로 올바르게 애플리케이션이 갱신이 되는지 확인한다.

```

master]# kubectl rollout status deployment nginx-deployment
deployment "nginx-deployment" successfully rolled out

```

올바르게 롤아웃이 되었는지 롤-아웃 히스토리를 확인한다. 확인하는 방법은 다음과 같은 명령어로 확인이 가능하다.



```
master]# kubectl rollout history deploy nginx-deployment
deployment.apps/nginx-deployment
REVISION  CHANGE-CAUSE
1          <none>
2          <none>
```

이미지 버전이 1.7.9 에서 1.8 로 변경되면서 리비전 1, 2 번이 목록으로 출력이 된다. 리비전 정보를 자세히 확인하려면 rollout history 명령어를 통해서 확인이 가능하다.

```
master]# kubectl rollout history deployment nginx-deployment --revision=1
...
Containers:
  nginx:
    Image:      nginx:1.14.2
    ...
master]# kubectl rollout history deployment nginx-deployment --revision=2
...
Containers:
  nginx:
    Image:      nginx:1.8
    ...
```

이전 내용으로 롤백(rollback)를 진행하려면 다음과 같은 명령어를 실행한다. 원하는 리비전 혹은 리비전 번호로 명시가 가능하다.

```
master]# kubectl rollout undo deployment nginx-deployment
deployment.apps/nginx-deployment rolled back

master]# kubectl rollout status deployment nginx-deployment
Waiting for deployment "nginx-deployment" rollout to finish: 1 old
replicas are pending termination...
Waiting for deployment "nginx-deployment" rollout to finish: 1 old
replicas are pending termination...
deployment "nginx-deployment" successfully rolled out
```

롤백이 완료가 되면 다시, deployment 설정내용을 describe 명령어를 통해서 확인한다.

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS
IMAGES	SELECTOR				
nginx-deployment	3/3	3	3	91m	nginx
nginx:1.7.9	app=nginx				

## 배포정책

생각 중...

## 연습문제

직접 디플로이먼트 파일을 생성한다. 아래 조건으로 각각 디플로이먼트 파일을 생성한다.

1. 최대 실행 Pod 개수는 2 개로 한다
2. 이미지는 httpd 를 사용한다
  - quay.io/centos7/httpd-24-centos7
3. httpd-24-centos7 으로 구성 후, 다시 이미지를 quay.io/centos7/httpd-24-centos7:centos7 으로 변경한다
4. 변경이 완료가 된 다음에 다시, 이전 버전으로 롤백한다
5. 올바르게 구성이 되었는지, ReplicaSet 도 구성이 되었는지 확인한다

## 23. 복제자(ReplicaSet, Replication Controller)

복제자는 단어 그대로 한 개 이상의 서비스를 복제 시 사용한다. 복제자는 서비스 사용량이나 혹은 사용자 설정에 따라서 개수를 늘리고 줄이고 한다. 복제자는 두 가지가 있는데, ReplicaSet, Replication Controller 가 있다. 현재 쿠버네티스는 ReplicaSet 사용을 권장하고 있다.

Replication Controller 는 ReplicaSet 처럼 복제를 한다. 하지만, 그 기준은 엄연히 많이 다르다. 아래는 간단하게 기능을 비교한 예시이다.

분류	Replication Controller	ReplicaSet	Deployment
설명	복제자 컨트롤러는 반드시 선택자가 동일해야지 복제를 한다. 현재는 ReplicaSet 으로 변경이 되었다.	복제자 묶음(set)은 기존 방식과 거의 동일하지만, 선택자가 하나가 아닌 여러 개를 지원한다.	Replication Controller 의 기능을 Deployment 가 대체한다. 자세한 내용은 뒤에서 더 설명. Deployment 는 ReplicaSet 과 같이 사용한다.
예제	os=rhel	os in (rhel, debian, ubuntu) os == rhel	
비고	위와 같이 동일해야지 복제자 컨트롤러는 복제를 시작한다.	지원 방식은 or, and 와 같은 방식으로 여러 개를 선택할 수 있다.	Replication Controller 에서 사용하는 rollout, rollback 기능을 Deployment 에서 사용한다.

초기 쿠버네티스는 Replication Controller 만 사용하였지만, 자원 설정만 필요하여 Deployment 가 추가가 되었다. 결국, Replication Controller 와 Deployment 의 관계 및 기능이 모호해지면서 결국 Deployment 는 replicaSet 과 같이 사용하며, 기존의 Replication Controller 호환성으로 그대로 남아있다.

쿠버네티스 호환 플랫폼에서 종종 찾을 수 있는 DeploymentConfig 가 있는데, 이 경우는 쿠버네티스와 호환성을 위해서 사용하지만, 현재는 쿠버네티스의 Deployment 를 사용하기 때문에, 기존 Deployment 를 그대로 사용하는 것을 권장한다.

## 복제자와 배포자의 관계

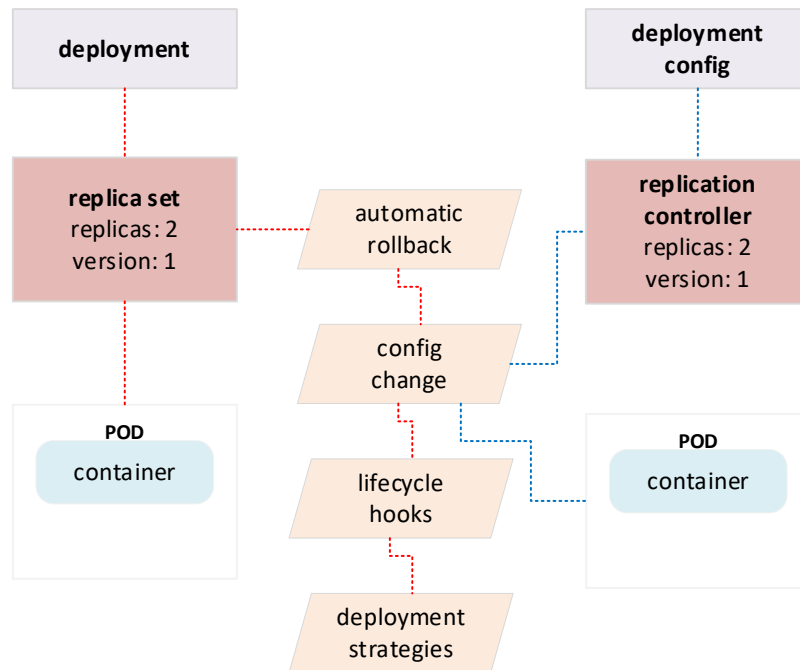


그림 50 replicaSet/replicaController

Pod 및 복제자를 동시에 생성한다. 아래와 같이 YAML 파일을 생성한다.

```
master]# cat <<EOF> pod-replicaset-test.yaml
apiVersion: v1
kind: Pod
metadata:
  name: rs-nginx-lab
  labels:
    app: rs-nginx-lab
spec:
  containers:
  - image: nginx
    name: rs-nginx-lab
    ports:
    - containerPort: 8080
      name: http
      protocol: TCP

---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
```

```

  name: rs-nginx-lab
spec:
  replicas: 5
  selector:
    matchLabels:
      app: rs-nginx-lab
  template:
    metadata:
      labels:
        app: rs-nginx-lab
    spec:
      containers:
      - name: rs-nginx-lab
        image: nginx
EOF
master]# kubectl apply -f pod-replicaset-test.yaml
master]# kubectl get pods
master]# kubectl get rs

```

위의 YAML 파일은 2 가지 자원 내용이 있는데, 첫 번째는 포드 생성 정보이며, 두 번째는 생성된 포드의 복제 개수에 대해서 명시한 복제자 설정이다. 복제자에는 선택자가 선언이 되어 있으며, 해당 선택자가 찾고 있는 메타데이터는 app=rs-nginx-lab 로 되어있다. 이 조건이 맞으면 총 5 개의 복제 포드를 생성한다. 적용 후 'kubectl get pods', 'kubectl get rs' 명령어로 올바르게 구성이 되었는지 확인한다.

복제자 자원에 두 가지가 있다. 하나는 앞에서 잠깐 언급한 ReplicaController(RC)라고 부른 자원이 있으며, 다른 하나는 Replica Controller Set(RS)가 있다. 두 기능은 동일하게 복제자 기능을 지원하며, 기존에 사용하던 RC 는 RS 로 변경을 권장하고 있다.

두 가지 기능의 큰 차이점은 바로 선택자 기능 부분이다. 앞에서 사용하였던 nginx 기반으로 테스트용으로 구성하도록 한다. 아래 YAML 은 Replication Controller(rc)복제자 설정이다.

```

master]# cat <<EOF> replication-controller-example.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: rc-nginx--single-app
spec:
  replicas: 3
  selector:
    app: rc-nginx-app

```

```
template:
  metadata:
    name: rc-nginx-app
    labels:
      app: rc-nginx-app
  spec:
    containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
EOF
master]# kubectl apply -f replication-controller-example.yaml
```

아래 YAML 은 ReplicaSet(rs) 복제자 설정이다. 내용은 앞서 사용한 replicationController 와 동일하다.

```
master]# cat <<EOF> replicaset-example.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-nginx-single-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rs-nginx-app
  template:
    metadata:
      labels:
        app: rs-nginx-app
        environment: dev
    spec:
      containers:
      - name: rs-nginx-app
        image: nginx
        ports:
        - containerPort: 80
EOF
```

```
master]# kubectl apply -f replicaset-example.yaml
master]# kubectl get rs
```

## 연습문제

위의 내용 기반으로 각각 1 개 rs, rc 를 구성한다.

1. rs, rc 는 quay.io/centos7/httpd-24-centos7:latest 를 사용한다
2. 각 자원은 복제자를 1 개로 명시한다
3. 필요한 경우 위의 YAML 파일을 참조하여 만든다

## 24. DaemonSet

정리 중...

## 25. 외부 아이피 주소(ExternalIP)

외부에서 미리 설정 및 할당한 아이피를 기반으로 클러스터에 접근 할 수 있도록 구성한다.

```
master]# cat <<EOF> external-ip.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: external-ip-app
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 49152
  externalIPs:
    - 192.168.10.250
EOF
master]# kubectl apply -f external-ip.yaml
```

## 연습문제

### 26. 외부 도메인 주소(ExternalName)

쿠버네티스에서 외부에서 접근하는 방법이 여러가지가 있다. 그중 하나가 ExternalName 이다. ExternalName 은 외부에 있는 DNS 서버의 CNAME 레코드를 통해서 접근한다. 서비스에서 expose 명령어를 통해서 외부에 구성이 되어 있는 도메인으로 노출한다.

```
kind: Service
apiVersion: v1
metadata:
  name: external-database
spec:
  type: ExternalName
  externalName: database.example.com
```

해당 외부 도메인 서비스는 서비스 도메인으로 통해서 접근이 가능하다. 예를 들어서 위의 nginx.example.com 은 my-nginx-srv.nginx.svc.cluster.local 서비스 도메인을 통해서 접근한다. my-nginx-srv 는 서비스 이름이고 nginx 는 네임스페이스 이름이다.

외부에 접근하는 CNAME 레코드 A 레코드가 바라보고 있는 아이피를 통해서 해당 애플리케이션 노드에 접근한다. 위의 구성을 하기 위해서는 외부 DNS 서버가 필요하다. 일반적으로 외부 DNS 는 와일드 카드 도메인 기반으로 구성한다.

## 연습문제

구성 중...

### 27. 네트워크 정책 (NetworkPolicy)

NetworkPolicy(이하, 네트워크 정책)은 네임 스페이스에 구성된 Pod 에 네트워크 정책을 구성한다. 이를 통해서 네임스페이스 대 네임스페이스 혹은 특정 Pod 끼리 통신이 가능하도록 설정한다.

```
master]# kubectl create namespace np-nginx
master]# kubectl config set-context --current --namespace=np-nginx
master]# kubectl create deployment np-nginx --image=nginx
master]# kubectl expose deploy/nginx --port=80 np-nginx
```



디버깅 컨테이너 하나 실행한다.

```
master]# kubectl run busybox --rm -ti --image=busybox:1.28 -- /bin/sh
container]# wget --spider --timeout=1 nginx
```

간단하게 네트워크 정책을 구성한다. 네트워크 정책은 ingress, egress 두 가지가 있다. 적용되는 범위는 다음과 같다.

이름	설명
egress	내부에서 외부에서 나가는 트래픽을 설정한다. 어떠한 자원이 외부로 나가는 걸 허용할지 selector 를 통해서 설정 및 구성한다.
ingress	외부에서 들어온 트래픽이다. 어떠한 자원을 수신할지 selector 를 통해서 설정 및 구성한다.

아래는 간단하게 외부에서 nginx 서비스에 접근이 가능하도록 구성한 네트워크 정책이다. Pod 에 app=nginx 라는 레이블 정보가 있으면, 해당 Pod 는 외부에서 접근이 가능하다. 위에서 생성한 nginx 서비스에 접근이 가능하도록 아래처럼 구성 및 설정한다.

```
master]# cat <<EOF> network-policy-nginx.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: np-ingress-nginx
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"
```

EOF

```
master]# kubectl apply -f network-policy-nginx.yaml
master]# kubectl get networkpolicy
NAME                POD-SELECTOR  AGE
access-nginx        app=nginx     9s
```

테스트를 하기 위해서 위에서 실행하였던 busybox 를 한 번 더 실행한다. 그리고 동일한 명령어를 컨테이너 내부에서 실행한다. busybox 가 접근 시 nginx Pod 에 접근하기 위해서 레이블 labels="access=true"를 추가한다. 추가가 된 busybox 컨테이너는 문제없이 nginx Pod 를 통해서 컨테이너에 접근이 가능하다.

```
master]# kubectl run busybox --rm -ti --labels="access=true" --
image=busybox:1.28 -- /bin/sh
container]# wget --spider --timeout=1 nginx
```

만약, 특정 네임스페이스에서 접근만 허용하는 경우 아래처럼 내용을 추가한다. 내용이 추가가 된 부분은 policyTypes, namespaceSelector 부분이다.

```
master]# cat <<EOF> network-policy-nginx.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: np-ingress-nginx
spec:
  podSelector:
    matchLabels:
      app: nginx
  policyTypes:
    - Ingress
  ingress:
    - from:
      - namespaceSelector:
          matchLabels:
            project: np-nginx
      - podSelector:
          matchLabels:
            access: "true"
EOF
master]# kubectl replace -f network-policy-nginx.yaml
```

위의 설정은 좀 더 명확하게 조건을 설정한다. 들어오는 트래픽(Ingress)에 대해서 설정하며, Pod 가 존재하는 자원 위치인 네임스페이스(namespaceSelector)도 명확하게 설정 및 구성하여 트래픽을 제어한다. 마지막으로 네트워크 정책의 조건을 정리하면 다음과 같다.

이름	조건	설명
----	----	----

ingress	from	네임 스페이스 외부에서 들어오는 조건입니다. 명시된, 네임 스페이스만 접근을 허용합니다.
egress	to	
podSelector	podSelector	
namespaceSelector	namespaceSelector	

모든 트래픽을 허용 혹은 차단하기 위해서는 다음과 같이 정책을 설정한다. 먼저 차단은 다음과 같이 설정한다.

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
spec:
  podSelector: {}
  policyTypes:
    - Ingress
```

모든 트래픽을 허용 시 다음과 같이 네트워크 정책 파일을 생성한다.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-ingress
spec:
  podSelector: {}
  ingress:
    - {}
  policyTypes:
    - Ingress
```

모든 트래픽에 대해서 송신을 허용하지 않는 경우 아래처럼 네트워크 정책 파일을 생성한다.

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-egress
spec:
  podSelector: {}
  policyTypes:
  - Egress
```

모든 트래픽에 대해서 송신을 허용하는 경우, 아래처럼 네트워크 정책 파일을 생성한다.

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-egress
spec:
  podSelector: {}
  egress:
  - {}
  policyTypes:
  - Egress
```

인그레스 및 이그레스 전부 차단하는 경우는 아래처럼 네트워크 정책 파일을 생성한다.

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
```

- Ingress

- Egress

## 연습문제

구성 중...

# 쿠버네티스 기능 확장

이번 목차에서는 흔히 말하는 CI/CD 기능을 쿠버네티스 네임스페이스에서 구현해보도록 한다. 인프라 및 회사마다 구성하는 방식은 다르지만, 보통 CI/CD 인터페이스는 프로젝트 별로 하나씩 별도로 가지고 있다. 이러한 이유로 각 네임스페이스에서 사용하는 CI/CD 프로그램은 덩치가 크거나 기능이 복잡할 이유가 없다. 따라서, 가볍고 간단하게 기능 구현을 해보도록 한다.

## 1. MetalLB

### 설명

쿠버네티스 API 와 통합이 된 L4/L7 기능을 지원하는 쿠버네티스 로드밸런스 프로그램. 이를 통해서 손쉽게 L/B 서비스 구성이 가능하다.

### 설치

아래와 같은 절차로 설치가 가능하다.

```
# kubectl apply -f
https://raw.githubusercontent.com/metallb/metallb/v0.14.5/config/manifests/metallb-native.yaml
```

설치 후, MetalLB 가 서비스가 되지 않기 때문에, 아래 YAML 파일을 작성하면서 서비스가 정상화될 때까지 기다린다. 오래 걸리면 대략 10 분정도 필요하다. 설치가 완료가 되면, kube-system 에서 ARP 의 기능을 제한한다.

```
# kubectl get configmap kube-proxy -n kube-system -o yaml | \
grep strictARP

# kubectl get configmap kube-proxy -n kube-system -o yaml | \
sed -e "s/strictARP: false/strictARP: true/" | \
kubectl apply -f - -n kube-system
```

L2 레이어에 해당 대역의 아이피를 사용하도록 허용한다. 파일 이름은 l2.yaml 으로 작성한다.

```
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: lb-pool
  namespace: metallb-system
```

```
spec:
  ipAddressPools:
  - first-pool
```

설치 후 사용할 로드밸런서 아이피 대역을 구성한다. 파일 이름은 `ippool.yaml` 으로 작성한다.

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: first-pool
  namespace: metallb-system
spec:
  addresses:
  - 192.168.10.240-192.168.10.250
  - 10.10.10.240-10.10.10.250
  autoAssign: true
```

## 2. Ingress

인그레스 서비스 외부에서 쿠버네티스 클러스터에 접근 시, 통과하는 일종의 클러스터 게이트웨이이다. 쿠버네티스에서 제공하는 `ExternalName`, `NodePort` 를 사용하여도 되지만, 트래픽 제어나 고급기술은 사용하기가 어렵다. 그래서 쿠버네티스에서는 대표적으로 두 가지 인그레스 서비스를 많이 사용한다.

1. Nginx
2. HAProxy

실제로 이 외의 Ingress 서비스는 많은데, 위의 두개가 대표적으로 많이 사용하는 서비스이다. 이 부분에서는 둘 다 설치하여 동시에 구성 및 설정하는 방법에 대해서 학습하도록 한다. 참고로, 아래 모든 랩은 HAProxy Load Balancer 가 아닌, MetalLB 기반으로 사용하기 때문에 MetalLB 를 설치 후 진행한다.

### Nginx 소개 및 설명

쿠버네티스에서 많이 사용하는 인그레스 서비스이다. 웹 기반 서비스에서는 Nginx 를 많이 권장 및 선호하며, 사용자가 CRD 기반으로 다양한 설정이 가능하다. 또한, HAProxy와 비교하였을 때 좀 더 가볍고 확장성이 용이하다. 유일하게 단점이라고 하면, UDP 서비스 지원 부분이 HAProxy 에 비해서 많이 약하다.

## 설치

설치는 간단하게 kubectl 명령어로 진행한다. 설치 전, 올바르게 Load Balancer 서비스가 구성이 되어 있는지 확인한다. 이 랩에서는 MetalLB 기반으로 진행하기 때문에, 가급적이면 MetalLB 기반으로 진행한다.

```
# kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.10.1/deploy/static/provider/cloud/deploy.yaml
```

설치 후, 조금 기다리면 문제없이 동작이 된다. 동작이 되면 Pod 는 다음과 같이 상태가 변경이 된다.

```
# kubectl get pods -n ingress-nginx
ingress-nginx-admission-create-czlcx      0/1      Completed    0
ingress-nginx-admission-patch-4dd9f       0/1      Completed    0
ingress-nginx-controller-597dc6d68-sm5n5  1/1      Running      0
```

설치가 완료가 되면 간단하게 인그레스 서비스를 구성한다. 아래는 nginx 서비스를 간단하게 ingress 서비스로 구현한다.

## 설정

서비스 설치가 완료가 되면, 아래와 같이 간단하게 ingress 서비스를 설정한다. 테스트 용도로 nginx 이미지를 사용한다. 이를 통해서 ingress 서비스를 통해서 서비스 접근이 가능하도록 한다. 먼저 내부에서 간단하게 nginx 애플리케이션 실행 및 로컬 실행으로 구성 후, 올바르게 컨트롤러가 동작하는지 확인한다.

```
# kubectl create deployment ingress-nginx-demo --image=nginx --port=80
# kubectl expose deployment ingress-nginx-demo
# kubectl create ingress demo-nginx-local --class=nginx \
--rule="demo-nginx.local/*=ingress-nginx-demo:80"
# kubectl port-forward --namespace=ingress-nginx service/ingress-nginx-
controller 8080:80

# curl --resolve demo-nginx.local:8080:127.0.0.1 \
http://demo-nginx.local:8080
```

위와 같이 하였을 때 문제없이 기본 nginx 환영 페이지를 출력하면, 실제로 외부에서도 크게 문제없이 접근이 가능한지 확인해보도록 한다. 위에서 구성한 서비스를 가지고 계속 작업을 진행한다. 테스트를 위해서 임시적으로 도메인 nginx-demo.io 도메인을 사용하여 접근이 가능한지 확인한다. 도메인 서버가 있는 경우, A 레코드 구성, 그게 아니면 /etc/hosts 파일에 명시한다.

```
# kubectl create ingress nginx-demo-io --class=nginx \
--rule="www.nginx-demo.io/*=ingress-nginx-demo:80"
```



```
> nginx-demo-io      nginx    www.nginx-demo.io    10.10.10.240    80
# curl www.nginx-demo.io
<h1>Welcome to nginx!</h1>
```

ingress에서 사용하는 아이피는 조금만 기다리면 10.10.10.24X 아이피로 할당 받는다. 아이피 할당이 확인이 되면 위에와 같이 curl 명령어로 확인이 가능하다.

명령어가 아닌, YAML 형식으로 ingress 서비스 선언이 가능하다. 'create ingress' 부분은 YAML 파일로 아래와 같이 작성이 가능하다. 파일 이름은 ingress-nginx-demo.yaml 으로 작성한다.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  creationTimestamp: null
  name: nginx-demo-io
spec:
  ingressClassName: nginx
  rules:
  - host: www.nginx-demo.io
    http:
      paths:
      - backend:
          service:
            name: ingress-nginx-demo
            port:
              number: 80
        path: /
        pathType: Prefix
status:
  loadBalancer: {}
```

혹은 아래 명령어처럼 자원 생성이 가능하다.

```
# kubectl create ingress demo-nginx-local --class=nginx \
--rule="nginx-demo.io/*=ingress-nginx-demo:80"
```

잉그레스 자원이 생성이 되면, 아이피 할당까지 조금 시간이 걸린다. 아이피 확인이 되면, 도메인으로 접근을 시도한다.

```
# kubectl get ingress -w
```

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
demo-nginx-local	nginx	nginx-demo.io		80	5s
demo-nginx-local	nginx	nginx-demo.io	192.168.10.240	80	6s

## 서비스 확인 및 추가 구성(bind)

실제 서비스와 동일하게 확인하려면, 가급적이면 /etc/hosts 설정 보다는 bind 기반으로 구성하는 것을 권장한다. 실제와 동일하게 동작하는 확인하기 위해서 load balancer(lb.master.com) 서버에 DNS 서비스를 구성 후 조회를 해보도록 한다. 아래는 DNS 서버에 다음과 같이 도메인 서버를 설정한다. 파일 이름은 /var/named/nginx-demo.io.zone 파일로 작성한다.

```
$TTL      7200
nginx-demo.io.      IN      SOA      lb.example.com.  admin.nginx-
demo.io. (
                        2024061802      ; Serial
                        7200             ; Refresh
                        3600             ; Retry
                        604800           ; Expire
                        7200)            ;
NegativeCacheTTL

                        IN      NS       lb.example.com.

nginx-demo.io.      IN      A          192.168.10.240
www                 IN      CNAME     nginx-demo.io.
ns1                  IN      A          192.168.10.250
ns2                  IN      A          192.168.10.250
```

해당 존 파일을 사용하기 위해서 named.conf 에 도메인을 추가한다. /etc/named.rfc1912.zones 에 다음과 같이 내용을 추가한다.

```
zone "nginx-demo.io" IN {
    type master;
    file "nginx-demo.io.zone";
    allow-update { none; };
};
```

아래와 같이 DNS 서비스를 실행한다. 만약, 이미 구성한 내용이 있으면 재시작을 한다.

```
# systemctl enable --now named.service
# systemctl restart named.service
```

여기서 확인해야 될 부분은 외부에서 ingress 서비스를 통해서 접근이 가능한가? 이 부분이다. 현재 ingress 서비스는 내부에서만 동작하며, 외부에서 접근이 안되는 상태이다. 외부에서 접근하기 위해서 기존에 사용하던 ingress 서비스에 수정이 필요하다.

다음처럼 YAML 파일을 생성한다. 기존에 사용하던 ingress service 는 내부 아이피로 되어 있기 때문에 외부에서 접근이 가능하도록 수정한다. 파일 이름은 ingress-expose-lb.yaml 으로 생성한다.

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app.kubernetes.io/component: controller
    app.kubernetes.io/instance: ingress-nginx
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
    app.kubernetes.io/version: 1.10.1
  name: ingress-nginx-controller
  namespace: ingress-nginx
spec:
  ports:
    - name: port-1
      port: 80
      protocol: TCP
      targetPort: 80
    - name: port-2
      port: 443
      protocol: TCP
      targetPort: 443
    - name: port-3
      port: 8443
```

```

protocol: TCP
targetPort: 8443
selector:
  app.kubernetes.io/component: controller
  app.kubernetes.io/instance: ingress-nginx
  app.kubernetes.io/name: ingress-nginx
type: LoadBalancer
loadBalancerIP: 192.168.10.240

```

기본적인 내용은 같으나, ingress 서비스를 통해서 외부에서 접근이 가능하도록 **로드 밸런서** 아이피(loadBalancerIP) 부분만 추가한다. 위의 내용으로 서비스에 적용 후, 다음과 같이 출력이 되는지 확인한다.

```

# kubectl apply -f ingress-expose-lb.yaml
# kubectl delete svc -n ingress-nginx ingress-nginx-controller
# kubectl get svc -n ingress-nginx
ingress-nginx-controller          LoadBalancer    10.98.47.177
192.168.10.240    80:31489/TCP,443:31721/TCP,8443:31539/TCP

```

기존 자원과 이름이 같은 경우, 제거하여도 되고, 혹은 이름만 다르게 생성하여도 된다. 제거 및 생성이 완료가 되면 다음과 같이 자원이 구성이 되어 있는지 확인한다.

```

# kubectl get svc
nginx-demo-io      nginx    www.nginx-demo.io    192.168.10.240    80
# kubectl get ingress
nginx-demo-io      nginx    www.nginx-demo.io    192.168.10.240    80

```

위와 같이 외부 아이피 192.168.10.X 대역으로 구성이 되어 있으면 외부에서 [www.nginx-demo.io](http://www.nginx-demo.io) 도메인으로 접근이 가능하다. 데스크탑 혹은 서버에 192.168.10.250 아이피를 DNS 서버 목록에 추가가 되면, 외부에서 다음과 같이 접근이 가능하다.

# Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to [nginx.org](https://nginx.org).  
Commercial support is available at [nginx.com](https://nginx.com).

*Thank you for using nginx.*

## HAProxy 소개 및 설명

HAProxy는 L4 및 L7 레이어를 완벽하게 제공한다. 본 기능 자체가 소프트웨어적으로 L4/L7 기능 구현으로 작성이 되었다. 쿠버네티스에서는 HAProxy 서비스 기반으로 외부 서비스 노출 및 라우팅을 구현하고 있지만, 레드햇에서 만든 OKD, OpenShift는 HAProxy 기반으로 라우팅 서비스를 제공하고 있다. 레드햇에서 제공하는 라우트 서비스(route service)와 완전하게 동일하지는 않지만, 비슷하게 사용이 가능하다.

## 설치

helm 혹은 kubectl 명령어를 통해서 설치를 진행한다.

```
# helm repo add haproxytech https://haproxytech.github.io/helm-charts
# helm repo update
# helm install haproxy-kubernetes-ingress \
haproxytech/kubernetes-ingress --create-namespace \
--namespace haproxy-controller
# kubectl apply -f
https://raw.githubusercontent.com/haproxytech/kubernetes-
ingress/master/deploy/haproxy-ingress.yaml
```

설치가 완료가 되면 다음처럼 POD 생성이 되었는지 확인한다. Nginx와 동일하게 한 개의 컨트롤러가 동작하면 문제 없이 설치가 되었다.

```
# kubectl get pod -n ingress-haproxy
ingress-nginx-admission-create-czlcx      0/1      Completed    ingress-
nginx-admission-patch-4dd9f               0/1      Completed    ingress-nginx-
controller-597dc6d68-sm5n5    1/1      Running
```

## 설정

간단하게 서비스를 구성하여 올바르게 HAProxy 가 구성이 되었는지 확인한다. 앞에서 사용한 nginx 이미지를 가지고 HAProxy 기반으로 구성한다. 먼저 디플로이먼트를 구성한다.

```
# kubectl create deployment ingress-haproxy-demo --image=nginx --port=80
# kubectl expose deployment ingress-haproxy-demo
# kubectl get pods
> ingress-nginx-demo-7cb5bc79f6-wpc4q    1/1    Running
# kubectl get svc
> ingress-haproxy-demo    0/1    1    0
```

아래 명령어로 HAProxy 기반으로 ingress 서비스를 구성한다. 크게 문제가 없으면, 서비스에 접근이 가능하다.

```
# kubectl create ingress demo-haproxy-local --class=haproxy \
--rule="demo-haproxy.local/*=ingress-haproxy-demo:80"
# kubectl get svc -n haproxy-controller
> haproxy-kubernetes-ingress    NodePort    10.102.85.83    <none>
80:32059/TCP,443:31044/TCP,443:31044/UDP,1024:31657/TCP,6060:32455/TCP
# kubectl port-forward -n haproxy-controller service/haproxy-kubernetes-
ingress 80:80
```

## 확인

추가 중...

## 3. Gogs

보통 쿠버네티스에서 많이 사용하는 SCM 서비스 중 하나이다. GitLab/GIT Server 를 사용하여도 되지만, 하나는 너무나 큰 덩치를 가지고 있고, 기본 깃 서버는 너무 기본적인 기능만 가지고 있다. 그래서 그 중간인 Gogs 기반으로 GIT 서버를 구성한다.

## 설치

설치는 kube-gogs.yaml 으로 만들어서 쪽 작성한다. Istio 를 사용할 예정이면, Istio 인젝션 활성화를 권장한다.

```
---
apiVersion: v1
kind: Namespace
```

```
metadata:
  name: gogs
labels:
istio-injection: enabled
```

또한, 소스코드를 저장할 PV/PVC 저장소를 구성한다. 빠르게 구성하기 위해서 로컬 저장소 기반으로 구성한다. 원하는 경우 NFS 서버로 변경하여도 된다.

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: kube-git-pv
  namespace: kube-git
  labels:
    type: local
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 2Gi
  hostPath:
    path: /mnt/data/kube-git
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: kube-git-pv-claim
  namespace: kube-git
  labels:
    app: kube-git
spec:
```

```
accessModes:
  - ReadWriteOnce
resources:
  requests:
    storage: 2Gi
```

깃 서버로 사용할, gogs 애플리케이션 배포 설정을 구성한다.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kube-git
  namespace: kube-git
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kube-git
  template:
    metadata:
      labels:
        app: kube-git
        istio-injection: enabled
        version: v1
    spec:
      containers:
        - name: gogs
          image: gogs/gogs:0.13
          ports:
            - containerPort: 22
              name: ssh
```



```

      - containerPort: 3000
        name: http
    env:
      - name: SOCAT_LINK
        value: "false"
    volumeMounts:
      - name: kube-git-persistent-storage
        mountPath: /data
  volumes:
    - name: kube-git-persistent-storage
      persistentVolumeClaim:
        claimName: kube-git-pv-claim
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 8.8.8.8

```

쿠버네티스 클러스터 사용자가 사용이 가능하도록 서비스를 구성한다. 테스트를 원하는 경우, NodePort 로 구성하여도 상관 없다.

```

---
apiVersion: v1
kind: Service
metadata:
  name: kube-git
  namespace: kube-git
spec:
  selector:
    app: kube-git
  ports:
    - name: ssh
      protocol: TCP

```

port: 10022

targetPort: 22

- name: http

protocol: TCP

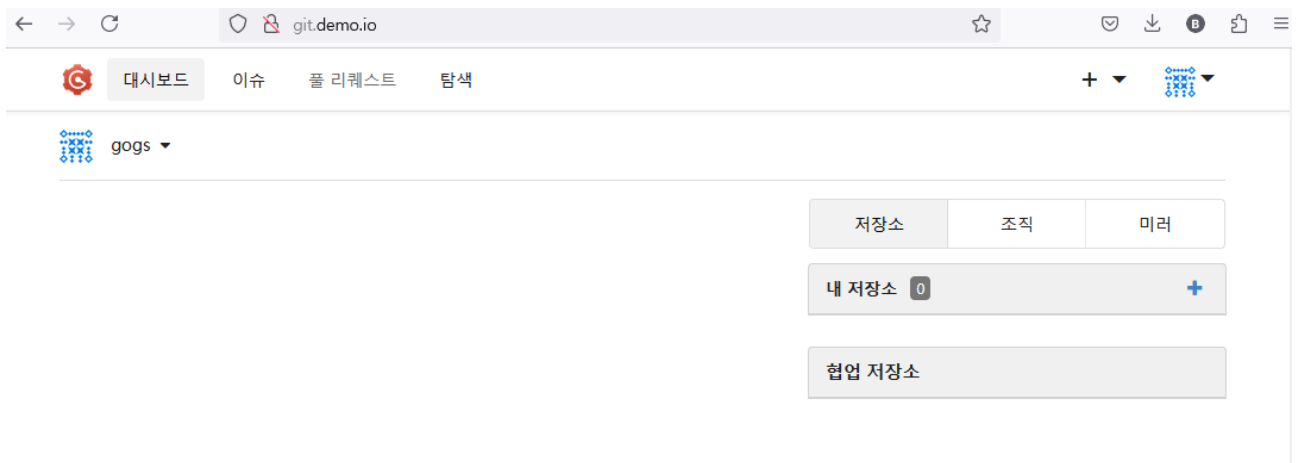
port: 18080

targetPort: 3000

클러스터 내부 및 외부에서 접근이 가능하도록 인그레스 서비스를 구성한다. 별도로 바디 크기 및 헤더의 대한 설정은 필요 없기 때문에 기본값으로 구성한다.

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: gogs-ingress
  namespace: gogs
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  ingressClassName: nginx
  rules:
  - host: git.demo.io
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: gogs-svc
            port:
              number: 18080
```

서비스가 문제없이 구성이 되었으면, git.demo.io 도메인으로 외부에서 접근이 가능한지 확인한다. 위 도메인은 인그레스 서비스가 구성이 되어 있어야지 접근이 가능하다.



## 4. docker-registry

네임스페이스에서 사용하는 네임 스페이스 이미지 스트림(namespace image stream 서버를 구성한다. 이 서버의 역할은 해당 네임스페이스에서 구성 및 배포되는 이미지를 저장 및 배포가 주요 목적이다.

### 설치

레지스트리가 설치가 될 네임스페이스 생성 및 서비스를 구성한다. 아래 모든 내용을 kube-registry.yaml으로 만들어서 작성한다.

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: kube-registry
```

해당 네임스페이스에 도커 레지스트리 서버를 생성 및 구성한다. 컨테이너 레지스트리 이미지는 아무거나 사용하여도 상관이 없다.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: kube-registry
  name: kube-registry
  namespace: kube-registry
spec:
```

```
replicas: 1
selector:
  matchLabels:
    app: kube-registry
template:
  metadata:
    labels:
      app: kube-registry
  spec:
    containers:
      - image: docker.io/opensuse/registry
        name: registry
        ports:
          - containerPort: 5000
```

인그레스 서비스에서 사용할 서비스를 구성한다. 테스트 용도로 노드 포트로 생성한다. 정상적으로 동작하면 ClusterIP 로 변경 혹은 재생성 하여도 상관없다.

```
---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: kube-registry
  name: kube-registry
  namespace: kube-registry
spec:
  ports:
    - port: 5000
      protocol: TCP
      nodePort: 30500
  selector:
```

```
app: kube-registry
type: NodePort
```

인그레스 서비스를 아래와 같이 추가한다. 다만, 추가할 때 외부에서 레지스트리 서버에 업로드 시 바디 크기 때문에 올바르게 업로드가 되지 않기 때문에 아래와 같이 작성한다.

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: registry-ingress
  namespace: kube-registry
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    kubernetes.io/ingressClassName: "nginx"
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/proxy-body-size: "0"
    nginx.ingress.kubernetes.io/proxy-read-timeout: "600"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "600"
status:
  loadBalancer:
    ingress:
      - ip: 192.168.10.240
spec:
  ingressClassName: nginx
  rules:
    - host: registry.demo.io
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
```

podman, buildah 에서 이미지 업로드 시,  
body 크기 때문에 업로드가 올바르게 되지 않기  
때문에, 크기 제한 및 타임 아웃을 넉넉하게 잡는다.

```
name: kube-registry

port:

  number: 5000
```

만약, skopeo 나 명령어가 익숙하지 않는 경우, 확인용 대시보드 설치가 가능하다. 이름은 kube-registry-dashboard.yaml로 생성한다.

```
---
kind: Deployment
apiVersion: apps/v1
metadata:
  namespace: kube-registry
  name: kube-registry-dashboard
  labels:
    app: kube-registry-dashboard
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kube-registry-dashboard
  template:
    metadata:
      labels:
        app: kube-registry-dashboard
    spec:
      containers:
        - name: kube-registry-dashboard
          image: joxit/docker-registry-ui:2.0
          ports:
            - name: dashboard
              containerPort: 80
          env:
            - name: REGISTRY_URL
```

v2 및 도메인 이름 설정.

```

        value: http://registry.demo.io/v2
- name: REGISTRY_NAME
value: http://reigstry.demo.io
    - name: SINGLE_REGISTRY
      value: "true"
    - name: REGISTRY_TITLE
      value: registry
    - name: DELETE_IMAGES
      value: "true"
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 192.168.10.250

```

자바코드 문제로 아래처럼 DNS 서버를  
오버라이드 한다.

애플리케이션이 문제없이 디플로이먼트가 되면 서비스를 구성한다.

```

---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: kube-registry-dashboard
  name: kube-registry-dashboard
  namespace: kube-registry
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: kube-registry-dashboard
status:

```

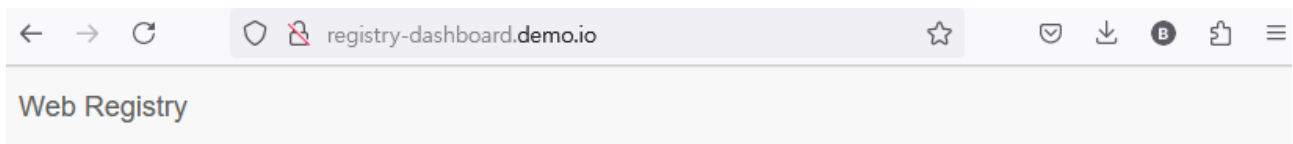


```
loadBalancer: {}
```

위의 서비스 기반으로 인그레스 서비스를 구성한다. 인그레스에 사용하는 도메인은 반드시 DNS 서버에 등록이 되어 있어야 한다.

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: registry-dashboard-ingress
  namespace: kube-registry
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    kubernetes.io/ingressClassName: "nginx"
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
status:
  loadBalancer:
    ingress:
      - ip: 192.168.10.240
spec:
  ingressClassName: nginx
  rules:
    - host: registry-dashboard.demo.io
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: kube-registry-dashboard
                port:
                  number: 80
```

문제없이 구성이 되면 <http://registry-dashboard.demo.io> 에 접근한다.



Home

## Repositories

**Registry**  
http://reigstry.demo.io

Repository	Tags
<a href="#">demo/ip</a>	1

## 5. ArgoCD

ArgoCD는 쿠버네티스에서 많이 사용하는 CD 프로그램 중 하나이다. 독립적으로 사용이 가능하며, 혹은 Tekton과 연동하여 함께 사용이 가능하다.

### 설치

설치를 바로 진행하지 않고, 아래 옵션을 통해서 설치 YAML 파일을 내려받은 후, 다음과 같이 수정한다.

```
# curl -o argocd-install.yaml  
https://raw.githubusercontent.com/argoproj/argo-  
cd/stable/manifests/install.yaml > install.yaml
```

간단하게 수정이 되어야 할 부분은 아래와 같다. 첫 번째는 위에서 받은 argocd 설치파일, 두 번째는 기존에 설치한 istio에 수정이 필요하다. 먼저, argocd부터 수정한다.

```
---  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  labels:  
    app.kubernetes.io/component: server  
    app.kubernetes.io/name: argocd-server  
    app.kubernetes.io/part-of: argocd  
name: argocd-server
```

```
containers:
  - args:
    - /usr/local/bin/argocd-server
    - --insecure
```

설치를 아래와 같이 진행한다.

```
# kubectl create -f argo-namespace.yaml
# kubectl apply -f install-argo.yaml -n argocd
# kubectl create namespace argo-rollouts
# kubectl apply -n argo-rollouts -f https://github.com/argoproj/argo-rollouts/releases/latest/download/install.yaml
```

수정이 완료가 되면, ingress 에서 TLS 를 paththrough 기능이 되도록 활성화한다. 우리는 argocd 에서 사용할 TLS 비공인키를 만들지 않는다.

```
# kubectl edit -n ingress-nginx deployments.apps ingress-nginx-controller

apiVersion: apps/v1
kind: Deployment
metadata:
name: ingress-nginx-controller
spec:
  containers:
    - args:
      - /nginx-ingress-controller
      - --publish-service=$(POD_NAMESPACE)/ingress-nginx-controller
      - --election-id=ingress-nginx-leader
      - --controller-class=k8s.io/ingress-nginx
      - --ingress-class=nginx
      - --configmap=$(POD_NAMESPACE)/ingress-nginx-controller
      - --validating-webhook=:8443
      - --validating-webhook-certificate=/usr/local/certificates/cert
      - --validating-webhook-key=/usr/local/certificates/key
      - --enable-metrics=false
```

- --enable-ssl-passthrough

위의 내용이 적용이 완료가 되면, 별 다른 수정없이 argocd 에 접근이 가능하다. 로그인 시 사용할 비밀번호는 아래 명령어로 확인이 가능하다.

```
# kubectl -n argocd get secret argocd-initial-admin-secret -o  
jsonpath='{.data.password}' | base64 -d; echo
```

## 6. ansible

쿠버네티스에서 앤서블 기반으로 소프트웨어 배포를 자동화가 가능하다. 앤서블에서 다음과 같은 쿠버네티스 모듈을 지원하고 있다.

[helm module](#) - Manages Kubernetes packages with the Helm package manager

[helm\\_info module](#) - Get information from Helm package deployed inside the cluster

[helm\\_plugin module](#) - Manage Helm plugins

[helm\\_plugin\\_info module](#) - Gather information about Helm plugins

[helm\\_pull module](#) - download a chart from a repository and (optionally) unpack it in local directory.

[helm\\_repository module](#) - Manage Helm repositories.

[helm\\_template module](#) - Render chart templates

[k8s module](#) - Manage Kubernetes (K8s) objects

[k8s\\_cluster\\_info module](#) - Describe Kubernetes (K8s) cluster, APIs available and their respective versions

[k8s\\_cp module](#) - Copy files and directories to and from pod.

[k8s\\_drain module](#) - Drain, Cordon, or Uncordon node in k8s cluster

[k8s\\_exec module](#) - Execute command in Pod

[k8s\\_info module](#) - Describe Kubernetes (K8s) objects

[k8s\\_json\\_patch module](#) - Apply JSON patch operations to existing objects

[k8s\\_log module](#) - Fetch logs from Kubernetes resources

[k8s\\_rollback module](#) - Rollback Kubernetes (K8S) Deployments and DaemonSets

[k8s\\_scale module](#) - Set a new size for a Deployment, ReplicaSet, Replication Controller, or Job.

[k8s\\_service module](#) - Manage Services on Kubernetes

[k8s\\_taint\\_module](#) - Taint a node in a Kubernetes/OpenShift cluster

위의 모듈을 통해서 앤서블 기반으로 서비스 배포를 자동화가 가능하다.

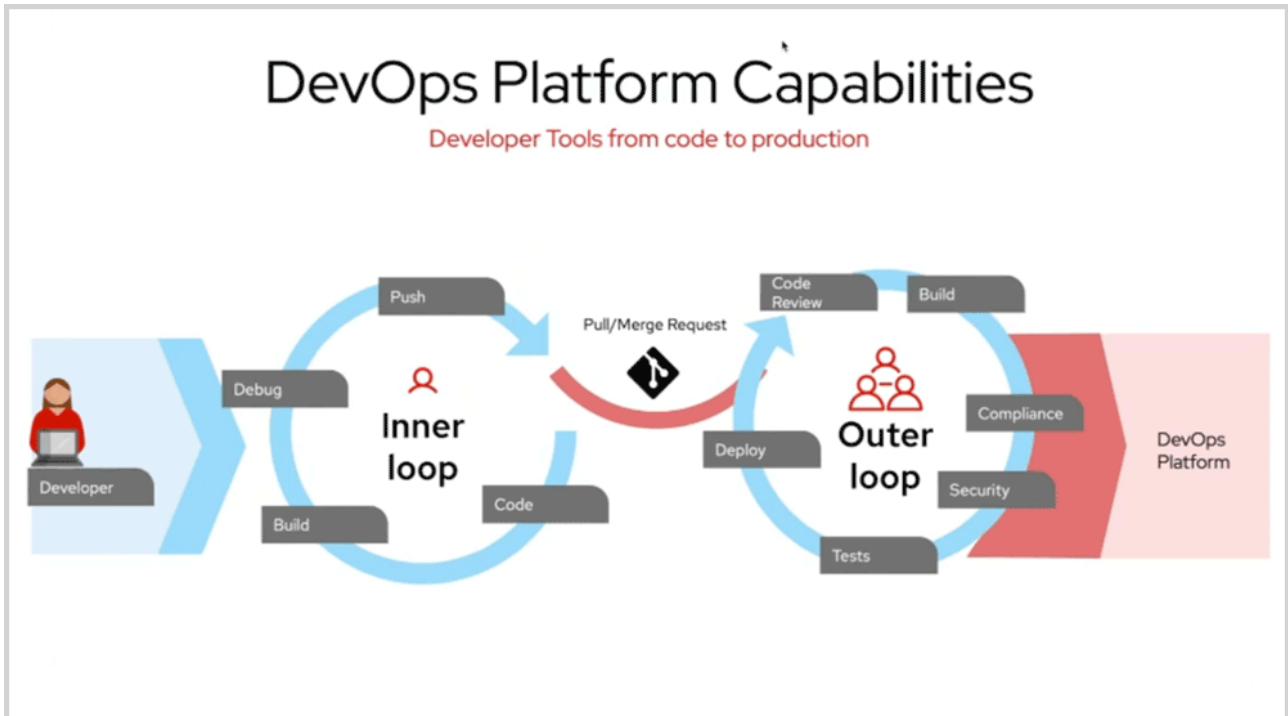
# DevOPs 를 위한 준비

## 1. 데브옵스는 무엇인가?

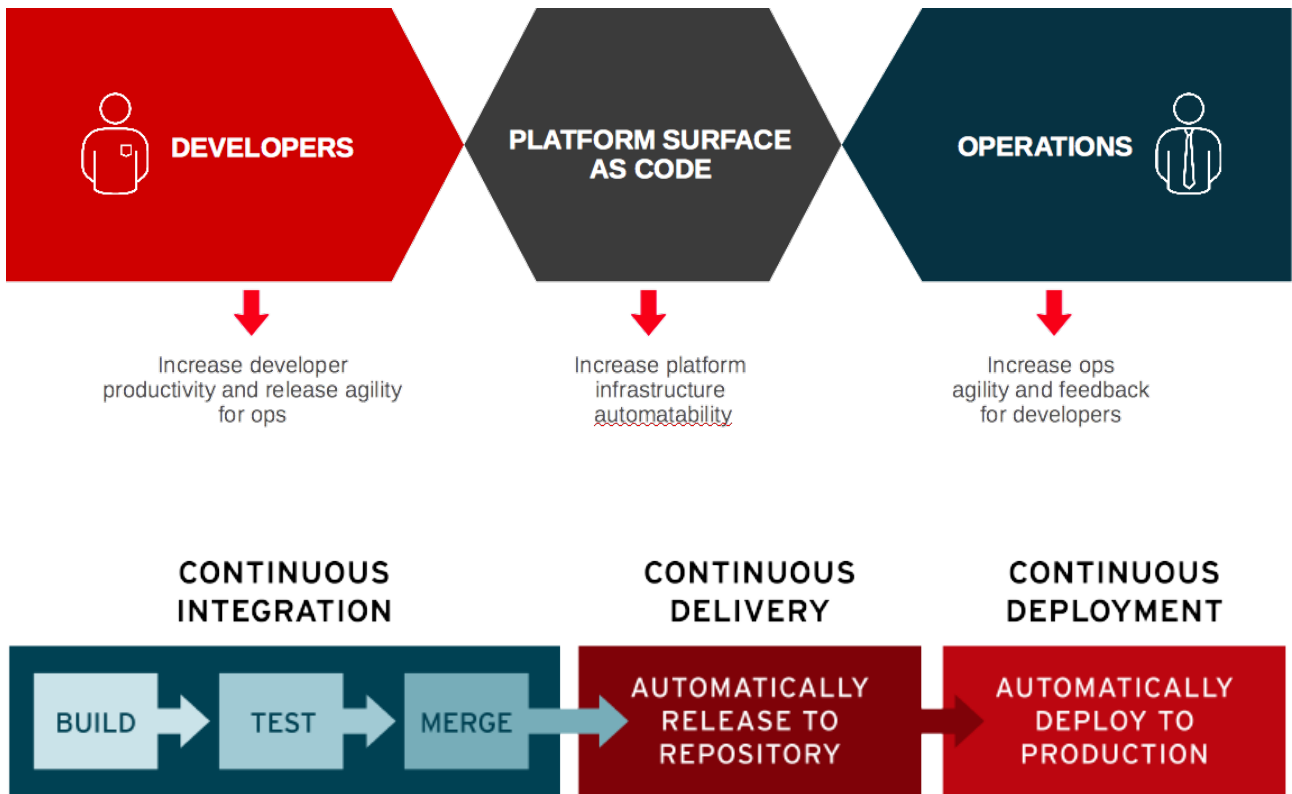
데브옵스(DevOps) 개발자/플랫폼/서비스가 하나의 사이클로 동작하는 개념이다. 데브옵스는 애자일(Agile)의 하위 개념이며, 효율적으로 애플리케이션 개선 및 배포를 위한 라이프 사이클이다. 보통 데브옵스는 두 가지 구역으로 나누어서 관리한다.

1. 내부 루프
2. 외부 루프

내부 루프는 개발자 및 서비스 품질 관련된 부분이다. 개선된 코드나 혹은 서비스를 내부 루프에서 검증 및 확인 후 외부 루프를 통해서 인프라에 서비스 배포 및 수행을 한다. 이와 같이 데브옵스는 모든 영역에서 이루어진다.



개발자는 보통 애자일기반으로 개발 및 릴리즈 속도를 빠르게 하며, 이를 손쉽게 적용 및 서비스 확장을 위해서 인프라스트럭처는 코드기반으로 인프라 운영 및 관리할 수 있도록 인프라 구조를 고도화한다. 마지막으로 서비스 오퍼레이터는 고객들이 제공한 피드백 기반으로 프로그램 및 인프라를 개선한다.



## 2. 표준 컨테이너 도구

### podman

포드만은 도커와 마찬가지로 독립전에 컨테이너 런타임이다. 포드만은 도커와 동일하게 Docker API 를 제공하며, 이를 통해서 포드 및 컨테이너 생성이 가능하다. 이외 나머지 기능은 도커와 동일하게 때문에 기존에 도커에서 사용하던 기능을 포드만으로 손쉽게 이전이 가능하다.

포드만은 도커에 없는 다음과 같은 기능을 제공한다.

- 쿠버네티스 마이그레이션
- systemd 자원 생성 및 구성

또한, 포드만은 도커 데스크탑과 동일하게 포드만 데스크탑을 제공하고 있으며, 이를 통해서 간단하게 쿠버네티스 및 오픈 시프트와 같은 플랫폼 도구를 SNO/SNK 형태로 구성 및 사용이 가능하다. 참고로 포드만의 이름은 Pod Manager 의 약자를 따서 Podman 이라고 명명하였다.

### 사용방법

아래는 간단하게 포드만을 활용하여 컨테이너 이미지 빌드 및 포드 및 컨테이너 생성을 테스트하는 과정이다.

```
# dnf install podman -y
# podman pod ls
# podman container ls
```

```
# podman volume ls
# podman network ls
```

```
# podman build
```

```
# podman run -d --pod new:infra-httpd --name: app-httpd --ip
192.168.10.100 --port 8080:8080 --volume htdocs:/var/www/html --network
pod-network
```

```
# curl 192.168.10.100
```

```
# podman generate kube
```

```
# kubectl apply -f kube-httpd.yaml
```

## skopeo

이미지 검색은 사용하는 컨테이너 레지스트리 서버에 따라서 각기 다르다. 예를 들어서 도커에서 사용하는 registry v2 와, 커뮤니티에서 많이 사용하는 harbor 와 같은 오픈소스 레지스트리 서버는 서로 다를 수 있다. 이를 해결하기 위해서 skope 는 오픈소스 표준 레지스트리 API 를 제공하고 있으며, 이들이 제공하는 API 는 OCI Image 에서 명시하는 API V2 container image registries 를 지원한다.

skopeo 가 제공하는 이미지 관리 기능은 보통 아래와 같다.

1. 이미지를 저장소나 혹은 특정 URL/URI 위치에서 복사해서 특정 레지스트리 서버나 혹은 로컬 저장소에 저장한다.
2. 원격에 있는 이미지를 메타 정보 내려받기 필요 없이, 원격에서 확인
3. 이미지 저장소에 있는 이미지 삭제
4. 외부에 있는 저장소의 내용은 내부 폐쇄망 레지스트리 서버에 저장
5. 저장소 접근 시, 인증이 필요한 경우

## 사용방법

skopeo 를 통해서 다음과 같이 이미지 관리가 가능하다.

```
# dnf install skopeo -y
# skopeo list-tags --tls-verify=false
docker://registry.demo.io/demo/nginx
# skopeo inspect --tls-verify=false
docker://registry.demo.io/demo/nginx:v1
```

```
# skopeo copy --tls-verify=false
```

```
# skopeo sync --tls-verify=false
```

## buildah

포드만으로 이미지 빌드는 가능하지만, 이미지 빌드를 위해서 포드만 설치 및 사용은 너무나 덩치가 크다. 또한, 포드만은 이미지 스크래치 기능을 제공하지 않기 때문에, 이미지 스크래치 빌드 및 테스트가 필요한 경우 빌더(buildah)<sup>29</sup>를 통해서 가능하다.

쿠버네티스에서 바이너리 코드 혹은 런타임 파일을 가지고 컨테이너 라이브러리로 패키징 하려면, 앞서 이야기한 듯이, 기존 도커나 포드만 기반으로 진행하기에는 무겁다. 그래서 쿠버네티스 CI 및 CD 소프트웨어에서 이미지 빌드시, 빌더를 통해서 이미지 빌드 작업을 수행한다. 여기서는 테크톤을 사용하여 이미지 빌드 작업을 자동화한다.

마지막으로 빌더는 도커와 마찬가지로 Dockerfile 기반으로 이미지 생성을 지원한다. 다만, OCI에서는 컨테이너 이미지 빌드를 Containerfile 라는 이름으로 변경하고 있기 때문에, 가급적이면 Dockerfile 이름 보다는 Containerfile 이름으로 사용을 권장한다.

## 사용방법

컨테이너 파일 기존 도커와 동일한 방법으로 사용 및 적용이 가능하다. 아래와 같이 실험용 컨테이너 파일을 작성한다. 파일 이름은 Containerfile 로 작성한다.

```
FROM quay.io/startx/php:latest
COPY ip-v1.php /app
```

테스트 용도로 사용할 PHP 프로그램은 간단하게 아래와 같이 작성한다. 파일 이름은 ip-v1.php 으로 작성한다.

```
<?php
```

---

<sup>29</sup> 저자는 맨 처음에 "빌드아"라고 읽었다. 일본에 살고 있는 친구가 이와 같이 알려주었고, 꽤 오랫동안 그렇게 믿었다. 컨퍼런스 가기 전까지...



```
$ip_server = $_SERVER['SERVER_ADDR'];  
$hostname_server = gethostname();  
  
echo "this PHP APP version is v1";  
echo "$ip_server";  
echo "<br>";  
echo $hostname_server;  
?>
```

위와 같이 작성이 완료가 되면 이미지 빌드를 시작한다. 먼저 빌더 설치가 안되어 있으면, 설치 후 진행한다.

```
# dnf install buildah -y  
# buildah bud . -t registry.demo.io/demo/ip:v1  
# buildah bud -f Containerfile -t registry.demo.io/demo/ip:v1
```

이미지 빌드가 완료가 되면 아래와 같은 명령어 이미지 빌드가 잘 되었는지 확인한다.

```
# buildah images registry.demo.io/demo/ip:v1  
# buildah push registry.demo.io/demo/ip:v1
```

### 3. 쿠버네티스 앤서블

앤서블은 2012 년에 발표가 되었으며, 현재는 레드햇이 인수했다. 앤서블은 두 가지 제품으로 분리가 되어 있다. **앤서블 엔진(혹은 코어)** 그리고 **앤서블 타워**로 구성이 되어 있다. 여기서는 설치 시 앤서블 엔진(코어)를 사용해서 설치할 예정이다. 현재 앤서블은 **AAP(Ansible Automate Platform)**라는 전략으로 기업용 시장에 적극적으로 진입을 하고 있다.

앤서블은 YAML 문법 형태의 문법 그리고, 파이썬 기반의 모듈을 통해서 동작한다. 사용자가 작성한 파일은 플레이북(playbook)이라고 불리며, 플레이북 및 역할(role)를 통해서 시스템에서 발생하는 작업을 반복적으로 처리할 수 있도록 디자인이 되어 있다.

앤서블은 총 두가지 제품으로 나누어져 있다. 첫번째는 앤서블 코어는 기본적인 코어 기능만 제공하며, 대시보드 같은 관리 기능은 제공하지 않는다. 앤서블 타워는 기본 코어 기능에 관리기능 및 CI 기능을 제공하여 레드햇은 해당 제품에 대해서 구독기반(subscription)으로 지원 서비스를 제공하고 있다.

결론적으로 앞으로 모든 인프라 시스템은 자동화 기반으로 구성 및 사용을 해야 되기 때문에, 앤서블 기반으로 서비스 배포 및 운영을 권장한다.

## 설치

작성 중...

## 랩

작성 중...

## 4. Tekton

### 설명

테크톤은 CNCF 에서 인증한 오픈소스 CD 도구이다. 이를 통해서 소스코드 컴파일 및 이미지 빌드 그리고, 서비스 배포가 가능하다. 테크톤 이외에도 많은 도구가 있지만, 현재 CNCF 에서는 이미지 빌드 및 서비스 배포까지 테크톤으로 통합을 시도하고 있다.

테크톤은 앤서블과 비슷한 동작 구조를 가지고 있지만, 제일 큰 차이점은 컨테이너 이미지 기반으로 작업이 수행이 되며, 파이프라인에서 실행하는 작업은 비동기 작업(a synchronizing task)로 수행이 된다. 간단하게 테크톤 리소스 및 작업에 대해서 설명한다.

### 설치

설치 시 버전은 적절하게(?)하게 타협하여 0.70 버전으로 한다. 최신 버전 사용을 원하는 경우, 반드시 쿠버네티스 클러스터도 가급적이면 최신버전 사용을 권장한다. 아래 명령어를 수행하여 테크톤 명령어 및 클러스터 서비스를 구성한다.

```
# kubectl apply -f https://storage.googleapis.com/tekton-
releases/operator/previous/v0.70.2/release.yaml
# wget
https://github.com/tektoncd/cli/releases/download/v0.32.0/tkn_0.32.0_Linu
x_x86_64.tar.gz
# mkdir ~/bin/
# tar xf tkn_0.32.0_Linux_x86_64.tar.gz -C ~/bin/
```

올바르게 설치가 되면, 추가적으로 3 개의 작업을 테크톤 허브에서 내려받기 하여 설치를 진행한다. 다만, 테크톤 설치가 완료가 되지 않으면 설치가 진행이 안되기 때문에 확인 후 진행한다.

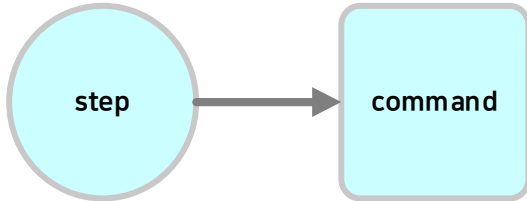
```
# kubectl get pod -n tekton-pipeline -w
# tkn hub install task buildah
# tkn hub install task kubernetes-actions
# tkn hub install task git-clone
# kubectl get task
```

완료 후, 다음과 같은 명령어로 올바르게 테크톤이 설치가 되었는지 확인한다.

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

## 단계(step)

실제 작업이 수행이 되는 자원이다. 최소 한 개 이상의 단계가 구성이 되어야 하며, 각 단계들은 컨테이너 이미지 기반으로 동작한다.



단계는 YAML 코드로 다음과 같이 작성한다. 각 작업은 컨테이너 이미지에서 제공하는 프로그램을 실행한다. 아래는 centos 이미지 기반으로 명령어를 실행한다. 안개 이상의 명령어를 실행할 때 script, 단순 명령어만 실행하는 경우 command 으로 실행한다.

```
steps:
- name: hello-world
  image: quay.io/centos/centos
  script: |
    echo "hello world"
```

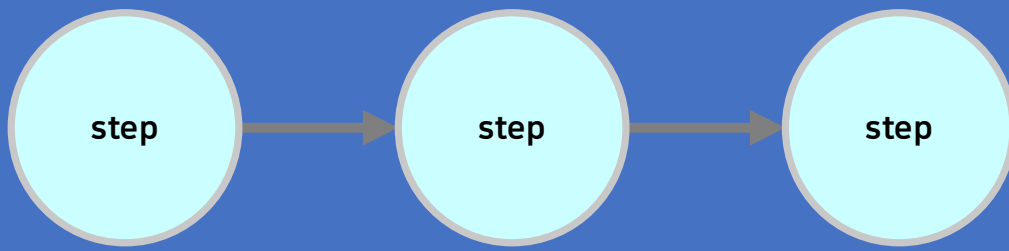
단일 실행은 다음과 같다.

```
steps:
- name: hello-world
  image: quay.io/centos/centos
  command:
    - ls
  args:
    - -la
```

## 작업(task)

작업은 실제로 단계가 수행이 되는 영역이다. 작업에는 한 개 이상의 단계를 가지고 있으며, 이를 순차적으로 실행을 한다. 다만, 태스크에서 수행한 작업의 결과물은 서로 공유하지 않기 때문에, 테크톤에서 지원하는 특수 자원, workspace, result 를 사용하여 공유한다.

## 태스크



태스크는 다음과 같이 작성한다. 파일명은 `tekton-task-helloworld.yaml` 으로 작성한다.

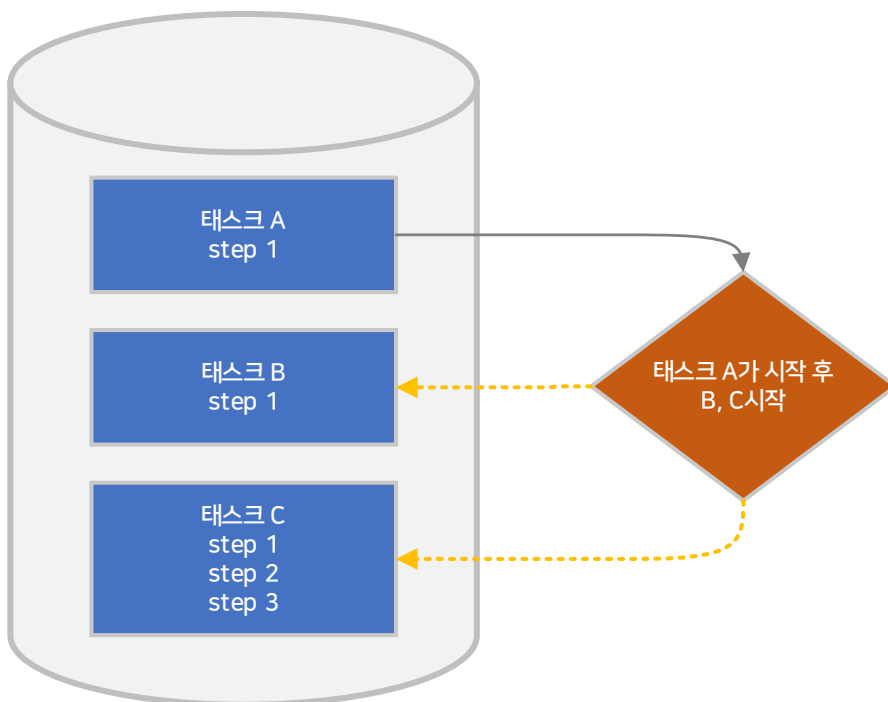
```
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: hello-world
spec:
  steps:
  - name: hello-world
    image: quay.io/centos/centos
    script: |
      echo "hello world"
  - name: show-date
    image: quay.io/centos/centos
    script: |
      echo $(date)
```

실행을 편하게 하기 위해서 작업 실행 자원을 생성한다. 파일 이름은 tekton-helloworld-run.yaml 으로 만든다.

```
---
apiVersion: tekton.dev/v1
kind: TaskRun
metadata:
  generateName: hello-world-run
spec:
  taskRef:
    name: hello-world
```

## 파이프라인(pipeline)

테크톤에서 여러 개의 작업을 하나로 묶어주는 부분을 파이프 라인 통해서 구현한다. 파이프라인은 한 개 이상의 작업을 가지고 있으며, 이를 동시에 순차적으로 수행한다. 여기서 말하는 순차적이라는 표현은 무작위로 작업을 순차적 진행을 말한다.



파이프라인을 다음과 같이 작성한다. 파일 이름은 tkn-pipeline-hello-world.yaml 으로 작성한다.

```
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: pipeline-demo
```

```
spec:
  workspaces:
    - name: source
  tasks:
    - name: hello
      taskRef:
        name: hello-world
      workspaces:
        - name: source
    - name: date
      taskRef:
        name: show-date
      workspaces:
        - name: source
      runAfter:
        - hello
```

Z

## 워크스페이스/결과(workspace, result)

앞서 이야기하였지만, 테크톤에서 생성된 결과물을 테크톤 Pod 를 통해서 공유하게 된다. 이를 위해서 보통 workspace 를 사용하여 구성한다.

자원 이름	설명
workspace	파이프라인에서 생성된 자원을 workspace 를 통해서 공유한다. 공유가 된 자원은 쿠버네티스에서 생성 및 구성된 PV/PVC 통해서 공유하게 된다.
result	쿠버네티스 실행 후 발생 결과를 테크톤 Pod 에 특정 디렉터리에 공유한다.

## 테크톤 확장 기능

확장 기능은 다른 사용자가 미리 만들어 둔 작업을 사용한다. 앤서블에서는 역할(role)과 비슷하게 사용한다. 재활용 빈도가 높으면 반복적으로 사용하는 작업들은 허브로 구성해서 배포한다. 이 랩에서 사용하는 확장 기능은 총 3 가지가 있다.

1. kubernetes-action

2. buidlal
3. git-lcone

앞서 설치하였기 때문에, 이를 사용하여 바로 진행한다.

## 테크톤 활용 예제

앞에서 간단하게 테크톤 자원에 대해서 학습하였다. 위의 자원을 통해서 서비스 구성 시 테스트용 애플리케이션을 테크톤 기반으로 이미지 빌드 및 배포를 수행하도록 한다. 앞서 설치한 자원을 가지고 다음과 같이 테크톤에서 자원을 구성한다.

1. 소스코드 컴파일 혹은 검증
2. 깃 서버에 있는 소스코드 이미지로 빌드
3. 빌드 후 컨테이너 이미지 레지스트리 서버에 업로드 하기
4. 빌드 된 이미지 기반으로 쿠버네티스에 서비스 구성

위의 작업을 수행하기 위해서 네임스페이스 레지스트리 서버 및 깃 서버가 구성이 되어야 한다. 앞에서 미리 구성하였기 때문에 문제가 없으면 아래 작업 수행이 가능하다. 테크톤 허브에서 받은 작업을 활용하여 위의 작업을 아래처럼 수행한다. 파일 이름은 tekton-demo-pipeline.yaml 으로 작성한다.

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: tekton-buildah-express-pipeline
spec:
  workspaces:
    - name: pipeline-shared-data
      description: |
        This workspace will be shared throughout all steps.
  params:
    - name: image-repo
      type: string
      description: |
        Docker image name
      default: registry.demo.io
  tasks:
    - name: clone-repository
```

```

params:
  - name: url
    value: http://git.demo.io/demo/demo-php.git
  - name: revision
    value: "master"
  - name: deleteExisting
    value: "true"
taskRef:
  kind: Task
  name: git-clone
workspaces:
  - name: output
    workspace: pipeline-shared-data
- name: build-image
  runAfter:
    - clone-repository
  params:
    - name: IMAGE
      value: "${params.image-repo}/app/php-ip:test"
    - name: TLSVERIFY
      value: false
    - name: DOCKERFILE
      value: Containerfile
  taskRef:
    kind: Task
    name: buildah
  workspaces:
    - name: source
      workspace: pipeline-shared-data

```

위의 내용이 올바르게 동작하면, CD 를 통해서 쿠버네티스 딜리버리가 가능하도록 pipeline-run 리소스를 추가적으로 생성한다.



위의 단계는 소스코드를 받은 후, 빌드 프로세스가 있으면 빌드 및 검증 후 컨테이너 이미지로 패키징 하는 단계이다. 이 이후에 진행이 되어야 할 부분은 바로 쿠버네티스로 서비스를 배포하는 부분인데, 이 부분은 kubernetes-action 를 통해서 서비스 배포를 수행한다.

```
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  name: kubectl-run
spec:
  taskRef:
    name: kubernetes-actions
  params:
    - name: script
      value: |
        kubectl apply -f
        https://raw.githubusercontent.com/vinamra28/social-
        client/master/k8s/deployment.yaml
        kubectl get deployment
```

## 5. Istio

### 설치

istio 설치 전, 설치에 필요한 istio 프로비저닝 도구 및 클라이언트 내려받기 후, 설치가 가능한 환경인지 확인한다.

```
# curl -L https://istio.io/downloadIstio | sh -
# export PATH="$PATH:/tmp/istio-1.22.1/bin"
# istioctl x precheck
```

설치에 사용할 Istio TLS 키를 생성한다. 해당 키가 없으면 올바르게 Istio-ingress, egress 설치가 진행되지 않는다.

```
# mkdir -p certs
# pushd certs
# make -f ../tools/certs/Makefile.selfsigned.mk root-ca
```

```
# make -f ../tools/certs/Makefile.selfsigned.mk cluster1-cacerts
# kubectl create namespace istio-system
# kubectl create secret generic cacerts -n istio-system \
  --from-file=cluster1/ca-cert.pem \
  --from-file=cluster1/ca-key.pem \
  --from-file=cluster1/root-cert.pem \
  --from-file=cluster1/cert-chain.pem
```

서비스가 잘 구성이 되었는지 아래 명령어로 확인한다. 각 서비스에 자원이 3 개씩 생성 및 구성이 되어 있어야 한다.

```
# kubectl get svc -n istio-system
istio-egressgateway      ClusterIP      10.111.104.103
istio-ingressgateway     LoadBalancer  10.104.14.12
istiod                   ClusterIP      10.99.69.63
# kubectl get pods -n istio-system
istio-egressgateway-797f65d7f5-x4c5p    1/1      Running
istio-ingressgateway-5f5f6bcd7c-sh666    1/1      Running
istiod-6cb887fdd5-wffc9                  1/1      Running
```

모니터링을 위해서 Kail 를 설치한다. 대시보드 그래프를 위한 Grafana 도 같이 설치를 진행한다.

```
# kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.22/samples/addons/kiali.yaml
# kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.22/samples/addons/grafana.yaml
# kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.22/samples/addons/prometheus.yaml
```

설치가 잘 되었는지 Pod 및 SVC 를 확인한다.

```
# kubectl -n istio-system get svc kiali
```

Kiali 에서 확인하기 위해서 istioctl 명령어를 통해서 접근 가능한 대시보드 주소를 얻는다. 이 주소는 영구적인 주소가 아니라, 일시적인 서비스 주소이다.

```
# ./istioctl dashboard --address 192.168.10.250 kiali
```

## 6. knative

### 설명

### 설치

knative 는 설치가 다른 도구에 비해서 좀 복잡하고 까다롭다. operator 를 사용하지 않는 경우, 일반적으로 아래와 같은 단계로 설치를 진행한다. 가상머신에서 설치시 성능에 따라서 자원 생성하는 속도가 다를 수 있기 때문에, 설치 중간에 오류가 발생하면, 다시 한번 설치 명령어를 실행한다. 보통은 다시 실행하면 중지된 부분도 다시 설치를 진행한다.

```
# mkdir knative
# mkdir -p ~/.config/kn/plugins/
# mkdir ~/bin
# cd knative
# wget https://github.com/knative/client/releases/download/knative-
v1.9.1/kn-linux-amd64
# wget https://github.com/knative-extensions/kn-plugin-
operator/releases/download/knative-v1.7.1/kn-operator-linux-amd64
# chmod +x kn-operator-linux-amd64 kn-operator
# cp kn-operator-linux-amd64 kn-operator
# cp kn-linux-amd64 kn-operator
# cp kn-operator ~/.config/kn/plugins/
# cp kn-linux-amd64 ~/bin/kn
# kn version
# kn operator -h
# export KNATIVE_VERSION=v1.9.1

# kubectl apply -f https://github.com/knative/net-
istio/releases/download/knative-v1.14.1/net-istio.yaml
# kubectl label namespace knative-serving istio-injection=enabled
abled
```

잉그레스 서비스로 Istio, Kourier 둘중 하나 사용이 가능하다. 기본 값은 Kourier 이며, 만약, Istio 사용하기 위해서 옵션을 변경해야 한다.

Istio 서비스 기반으로 설치는 아래와 같다.

```
# kn operator install -c serving -v -v ${KNATIVE_VERSION} -n knative-
serving --istio --istio-namespace istio-system
```

```
# kn operator install -c eventing -v -v ${KNATIVE_VERSION} -n knative-  
eventing --istio --istio-namespace istio-system
```

kourier 기반은 아래와 같이 설치한다.

```
# kn operator install -c serving -v ${KNATIVE_VERSION} -n knative-serving  
# kn operator install -c eventing -v ${KNATIVE_VERSION} -n knative-  
eventing
```

위의 명령어로 설치 진행 시, 올바르게 구성이 안되면, 보통 CPU 및 Memory 문제로 진행이 안된다.  
knative 를 사용하기 위해서 가급적이면 메모리는 8 기가 이상을 권장한다.

```
# kubectl get pods,svc -n knative-serving
```

# 관리 및 운영

1. 클러스터 업그레이드
2. 소프트웨어 L4 기반 마스터 로드 밸런서
3. ETCD 백업 및 복원
4. 컨테이너 레지스트리 서버 구성
5. External ETCD 서버 구성
6. External DNS 서버 구성
7. 쿠버네티스 스케줄러 수정 및 변경하기
8. 로그 및 트러블 슈팅을 위한 방법

**9. 이미지 복사 및 배포**

**10. 외부 계정 연결**

**11. kube-virt**

**12. 클러스터 연합**