

DUSTBOX

# 쿠버네티스 101

쿠버네티스 기본

# 내용

과정개요.....	1
쿠버네티스 소개 .....	1
1. 쿠버네티스는 무엇인가?.....	1
2. 쿠버네티스 구성 요소.....	3
3. 쿠버네티스 표준 사양.....	4
OCI ( <i>Open Container Initiative</i> ).....	4
CNI ( <i>Container Network Interface</i> ).....	5
CSI ( <i>Container Storage Interface</i> ).....	5
CRI ( <i>Container Runtime Interface</i> ).....	5
4. 인프라 코드화.....	5
5. 클라우드 네이티브 .....	6
설치 환경 및 사양.....	7
1. 설치 권장 사양 .....	7
2. 기본 랩 구성 사양 .....	7
3. 하이퍼바이저 소프트웨어 사양.....	8
4. 에디터 설정 .....	9
VI/VIM .....	9
NANO .....	9
NeoVIM .....	9
5. 터미널 도구 .....	9
tmux .....	9
쿠버네티스 설치 준비 및 설명 .....	10
1. 앤서블 기반 설치 도구 .....	10
kubespray.....	10
duststack-k8s-auto.....	11
2. HYPER-V 설치 .....	11
설치.....	11
네트워크 구성 .....	11
가상머신 생성 .....	12
3. 설치 전 부트스트랩 명령어 살펴보기 .....	13
init .....	13
join.....	17
upgrade .....	18
config .....	18
token .....	18
reset .....	18
version .....	19
alpha .....	19
4. 쿠버네티스 설치 .....	19
운영체제 공통 설정 .....	19
5. 쿠버네티스 자동설치 .....	28
6. 쿠버네티스 수동설치 .....	28
kubelet 설치 및 구성 .....	28
작은 규모 설치(단일 컨트롤) .....	28
PoC 규모 설치(다중 컨트롤).....	30
네트워크 설치/ .....	34
컴퓨트 노드 설치.....	36
클러스터 초기화 .....	36
랩에서 사용할 CNCF 도구 .....	36

1. 표준 OCI 도구와 관계 .....	36
2. BUILDAH.....	37
3. SKOPEO.....	37
4. TEKTON(테크톤) .....	38
5. 저수준 런타임.....	38
Containerd.....	40
CRI-O .....	41
Podman( <i>포드먼</i> ).....	41
6. 리눅스 커널과 쿠버네티스 관계 .....	42
THE NAMESPACE.....	42
C-GROUP .....	42
7. 파일시스템 .....	43
UFS .....	43
OverlayFS ( <i>Filesystem</i> ) .....	44
<b>쿠버네티스 아키텍처 .....</b>	<b>46</b>
1. 가상화 vs 컨테이너 .....	46
2. RUNC/CRUN .....	46
3. 쿠버네티스 클러스터 컴포넌트 설명.....	47
컨트롤 플레인( <i>control-plane, master, controller</i> ).....	49
컴퓨트 노드( <i>minion, compute</i> ) .....	49
kubelet.....	49
etcd.....	50
coredns.....	52
kube-proxy.....	53
kube-scheduler .....	54
controller-manager .....	55
kube-apiserver.....	56
스왑( <i>swap</i> ) .....	58
4. 쿠버네티스 주요 자원 설명.....	59
POD( <i>po</i> ).....	59
Pause( <i>Infra-Container</i> ).....	60
Service( <i>svc</i> ).....	61
Deployment( <i>deploy</i> ).....	63
ReplicaSet( <i>rs</i> ).....	63
Replication Controller( <i>rc</i> ) .....	63
Volume .....	64
StatefulSet.....	64
CronJob.....	64
DaemonSet( <i>ds</i> ).....	64
Secret( <i>sc</i> ) .....	65
ConfigMap( <i>cm</i> ) .....	65
Label .....	65
Selector.....	65
Annotation .....	65
<b>시작하기 전에.....</b>	<b>65</b>
1. 최종확인 .....	65
2. 명령어 자동완성 .....	67
3. YAML 문법 .....	68
연습문제.....	71
<b>기본 기능.....</b>	<b>71</b>
1. 네임스페이스 .....	71
연습문제.....	74
2. 생성/삭제/확인 .....	74
실행( <i>run</i> ) .....	74
적용( <i>apply</i> ).....	75
제거( <i>delete</i> ).....	77

생성(create, kustomize) .....	77
확인(get/describe) .....	81
연습문제.....	83
3. 멀티 컨테이너 구성 .....	83
연습문제.....	85
4. CP .....	85
연습문제.....	86
5. EXEC .....	86
연습문제.....	86
6. EXPOSE/SERVICE/ENDPOINT .....	86
service .....	86
expose .....	88
endpoints.....	90
연습문제.....	93
7. PROXY, PORT-FORWARD.....	93
8. LABEL .....	94
연습문제.....	95
9. SET .....	95
연습문제.....	96
10. EDIT .....	96
연습문제.....	97
11. DELETE .....	97
연습문제.....	97
12. DIFF .....	98
연습문제.....	98
13. DEBUG/LOGS .....	98
연습문제.....	99
14. EXPLAIN .....	99
15. REPLACE.....	100
연습문제.....	100
16. PATCH .....	100
연습문제.....	101
<b>프로젝트 구성 및 애플리케이션 배포.....</b>	<b>101</b>
1. 시나리오.....	101
2. 구성 및 소프트웨어 .....	101
<b>고급기능.....</b>	<b>102</b>
1. 쿠버네티스 스토리지 .....	102
스토리지 서버 구성 및 개념 설명.....	102
랩을 위한 NFS 서버 구축.....	103
StorageClass ServiceAccount.....	오류! 책갈피가 정의되어 있지 않습니다.
StorageClass.....	오류! 책갈피가 정의되어 있지 않습니다.
Persistent Volume (PV).....	108
Persistent Volume Claim (PVC).....	109
2. HELM .....	111
연습문제.....	115
3. METRICS 설치 .....	115
사용 및 확인.....	117
4. 쿠버네티스 컨텍스트 및 사용자 구성 .....	118
사용자 설명.....	118
사용자 생성.....	119
연습문제.....	120
5. ROLE/ROLEBINDING 생성 및 구성 .....	121
역할(Role).....	121
역할연결(RoleBinding) .....	123
kubernetes-admin 대안.....	125

6.	역할기반제어(RBAC) 설명 .....	126
7.	SERVICE ACCOUNT .....	127
8.	SCALE/ROLLOUT/ROLLBACK/HISTORY.....	129
	연습문제.....	132
9.	선호도 및 예외시간(AFFINITY/TOLERATIONS) .....	132
	포드 선호도(Pod Affinity).....	133
	노드 선호도(Node affinity).....	135
	오염 및 내성(Taints/Tolerations) .....	136
10.	변수 전달 .....	137
11.	컨테이너 자동확장(AUTOSCALE).....	138
	HPA 구성.....	138
	VPA 구성.....	141
	연습문제.....	144
12.	DRAIN/Taint/CORDON/UNCORDON .....	144
	비우기(drain).....	144
	오염 및 차단(taint/cordon, uncordon) .....	145
13.	노드추가 및 제거(NODE ADD/REMOVE) .....	146
	연습문제.....	147
14.	SERVICE/CLUSTER HEALTHCHECK .....	147
	상태 확인.....	147
	애플리케이션 상태 확인(Liveness).....	150
	포드 서비스 상태확인(Readiness).....	152
15.	LABEL, ANNOTATIONS, SELECTORS.....	155
	이름표(label) .....	155
	선택자(selector) .....	157
	주석(annotations) .....	158
	노드 선택(Node Selector) .....	159
16.	데몬 서비스(DAEMONSET).....	160
17.	상태 POD 서비스 (STATEFULSETS) .....	163
18.	작업 (JOBS/CRONJOBS) .....	163
	작업(Jobs).....	163
	크론잡(CronJobs, cj) .....	164
19.	설정파일(CONFIGMAPS(CM)) .....	165
	연습문제.....	168
20.	비밀(SECRETS(SC)) .....	168
	연습문제.....	171
21.	배포 및 배치(DEPLOYMENT) .....	172
	배포정책.....	174
	연습문제.....	174
22.	복제자(REPLICASET, REPLICATION CONTROLLER) .....	175
	복제자와 배포자의 관계.....	175
	연습문제.....	178
23.	외부 아이피 주소(EXTERNALIP) .....	178
	연습문제.....	178
24.	외부 도메인 주소(EXTERNALNAME).....	178
	연습문제.....	181
25.	네트워크 정책 (NETWORKPOLICY) .....	181
	연습문제.....	185
	쿠버네티스 기능 확장 .....	185
1.	로드 밸런서 .....	185
	MetalLB 설치/.....	188
	MetalLB 서비스 구성.....	188
1.	METALLB .....	오류! 책갈피가 정의되어 있지 않습니다.

설명	오류! 책갈피가 정의되어 있지 않습니다.
설치	오류! 책갈피가 정의되어 있지 않습니다.
2. INGRESS	오류! 책갈피가 정의되어 있지 않습니다.
Nginx 소개 및 설명	191
HAProxy 소개 및 설명	195
3. GOGS	197
설치	197
4. DOCKER-REGISTRY	200
설치	200
5. ARGOCD	204
설치	205
6. KUBE-VIRT	206
<b>DEVOPS 를 위한 준비</b>	<b>206</b>
1. 데브옵스는 무엇인가?	206
2. 표준 컨테이너 도구	207
podman	207
skopeo	208
buildah	208
3. 쿠버네티스 앤서블	209
설치	210
랩	210
4. TEKTON	210
설명	210
설치 및 설정	211
단계(step)	212
작업(task)	213
파이프라인(pipeline)	214
워크스페이스/결과(workspace, result)	214
테크톤 확장 기능	215
테크톤 활용 예제	215
5. ISTIO	216
설치	216
6. KNATIVE	218
설명	218
설치	218
<b>MSA 서비스 구성(데모)</b>	<b>219</b>
<b>3TIER 서비스 구성</b>	<b>219</b>
<b>관리 및 운영</b>	<b>219</b>
1. 클러스터 업그레이드	219
2. 소프트웨어 L4 기반 마스터 로드 밸런서	219
3. ETCD 백업 및 복원	219
4. 컨테이너 레지스트리 서버 구성	219
5. EXTERNAL ETCD 서버 구성	219
6. EXTERNAL DNS 서버 구성	220
7. 쿠버네티스 스케줄러 수정 및 변경하기	220
8. 로그 및 트러블 슈팅을 위한 방법	220
9. 오프라인(DISCONNECTED) 설치	220
10. 이미지 복사 및 배포	221
11. 외부 계정 연결	221
12. 클러스터 연합	221

# 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 10/11에서 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 컴퓨터 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 90년대말부터 컨테이너 오케스트레이션 도구들을 만들기 시작했다. 하지만, 이때 당시, 리눅스 진영에서는 컨테이너 및 가상화 기술이 많이 부족한 상태였고, BSD에서 사용하던 chroot와 같은 도구를 통해서 프로세스 격리 기능 구현하였다.

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

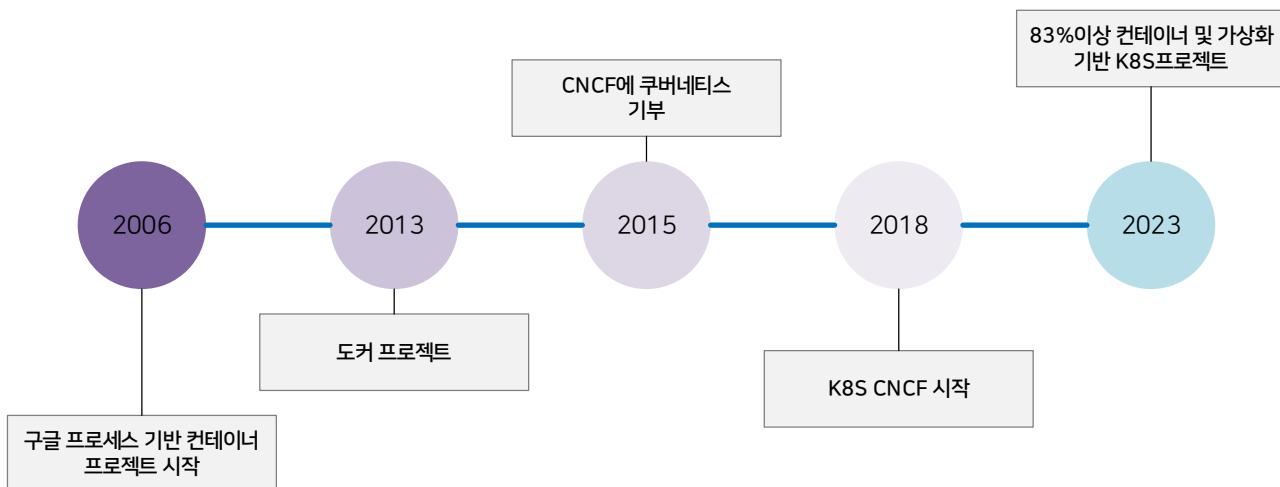


그림 1 컨테이너 프로젝트 소개

2000년도, 닷컴 버블<sup>1</sup>이 시작된 시점에, 가상머신 혹은 베어메탈 기반으로는 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 cgroup이라는 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 lxc 와 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. 이 부분에 대해서는 추후 런타임 부분에서 좀 더 이야기하도록 하겠다.

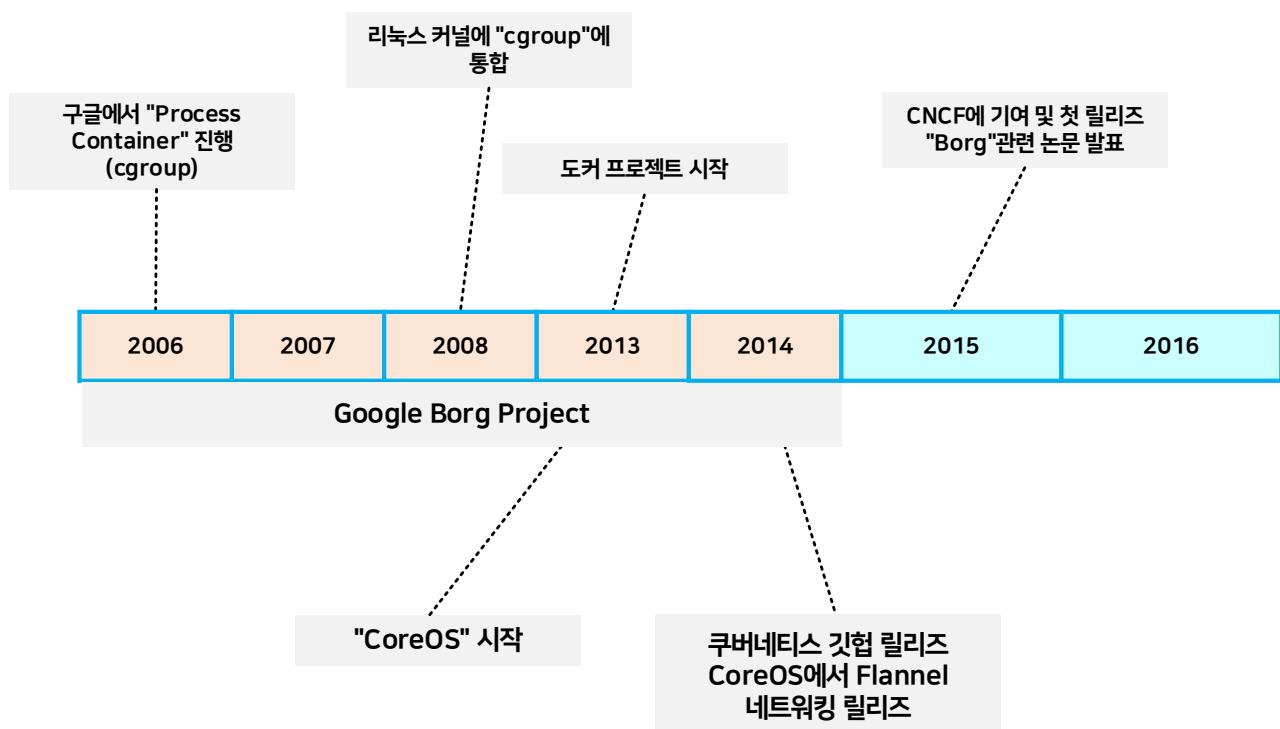


그림 2 보그 프로젝트

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

<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.

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

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

cgroup은 프로세스 추적은 되지만, 격리 기능은 지원하지 않기 때문에, 레드햇은 namespace를 만들었으며, 이를 통해서 특정 프로세스의 기능을 분리 및 격리가 가능하다. 이 기능은 본래 레드햇의 Atomic Project를 통해서 시작이 되었고, 중간에 레드햇은 CoreOS를 인수하면서 기존의 Atomic Project는 중지하면서 OSTree<sup>5</sup>기반으로 immutable 배포판을 제작하고 있다.



그림 3 레드햇 코어

위와 같은 기술들이 점점 개발 및 발전이 되면서 컨테이너 오케스트레이션 플랫폼이 나오기 시작했으며, 그 중 하나가 쿠버네티스 플랫폼이다. 2014년도에는 구글은 쿠버네티스 프로젝트를 깃헙을 통해서 공개하였다. 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

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

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

<sup>5</sup> <https://github.com/ostreedev/ostree>

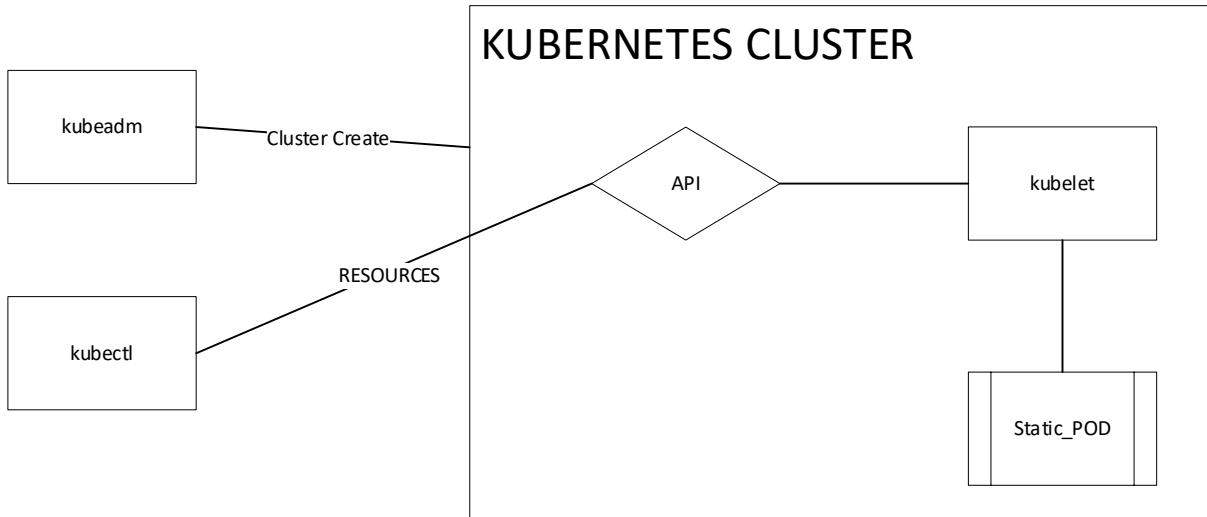


그림 4 쿠버네티스 구성 요소

`kubelet`은 프록시(proxy)와 같은 서비스이다. 외부에서 들어온 API 들은 `kubelet`를 통해서 정적 POD에 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`과 같은 도구가 올바르게 설치 및 구성이 되어 있어야 한다.

### 3. 쿠버네티스 표준 사양

쿠버네티스 표준 사양은 총 4 가지로 나누어져 있다. 아래 4 가지 서비스는 기본적으로 `kubelet`를 통해서 쿠버네티스와 연결 및 구성이 된다. 다만, OCI 경우에는 쿠버네티스 컴포넌트와 직접적인 관계는 없으며, 런타임/배포/이미지와 같은 번들 사양에 대해서 가지고 있다.



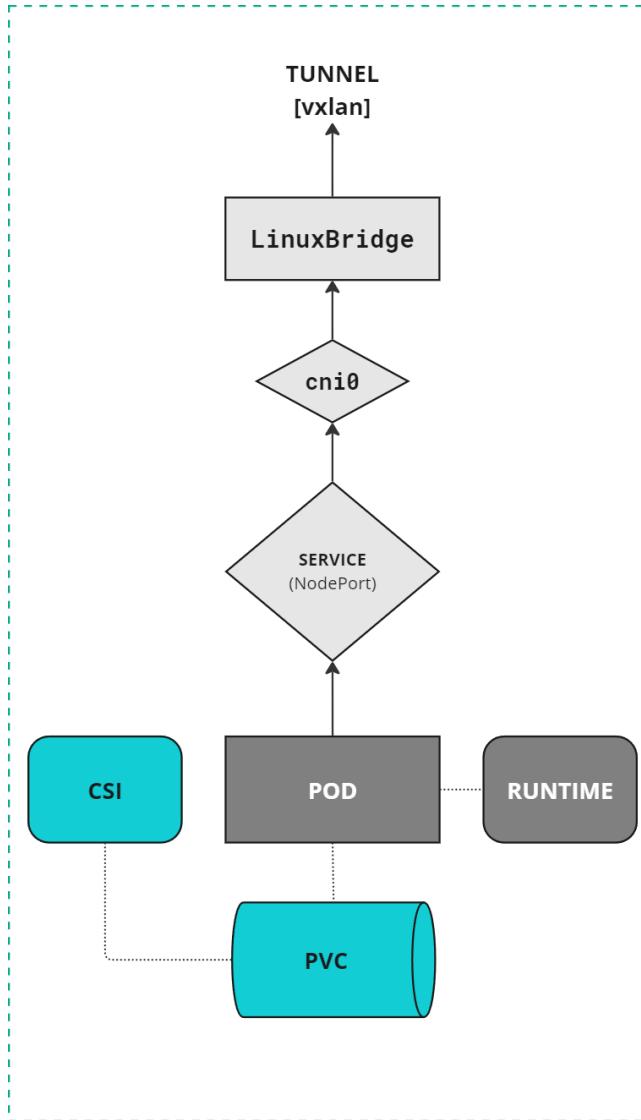
그림 5 CRI/CNI/CSI 와 `kubelet`

### OCI (Open Container Initiative)

OCI는 리눅스 파운데이션에서 관리하고 있으며, 이 프로젝트는 Docker를 통해서 2015년도에 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 OCI로 버전에 따라서 나누어지기 때문에 도커 기반으로 사용 시, 이러한 부분을 주의해야 한다. 오픈소스에서 많이 사용하는 Podman, CRI-O는 OCI를 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.

초기 컨테이너 런타임들은 일반적으로 리눅스 브릿지 기반으로 구성이 되었으나, 현재는 많은 SDN 및 NFV 기능들이 나오면서 CNI 기반으로 확장 및 구성이 가능하다.

COs(Container Orchestration Systems)기반의 모든 플랫폼에서 적용 및 사용이 가능하다.

CSI는 런타임 및 영역별로 다르지만, 컨테이너가 지원 및 사용하는 CSI 드라이버는 기본값은 overlay2로 되어 있다. 이외 PV/PVC에서도 이를 통해서 확장 저장소를 구성한다. CSI를 사용하는 경우 손쉽게 스토리지 확장이 가능하다.

## CRI (Container Runtime Interface)

컨테이너 런타임 인터페이스의 표준화를 위한 사양이다. 쿠버네티스는 CRI를 지원하지 않는 컨테이너 런타임은 kubelet을 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. 전환이 된 정보는 API 서버에서 각 쿠버네티스 컴포넌트에 전달하여 자원에 대해서 작업을 수행한다.

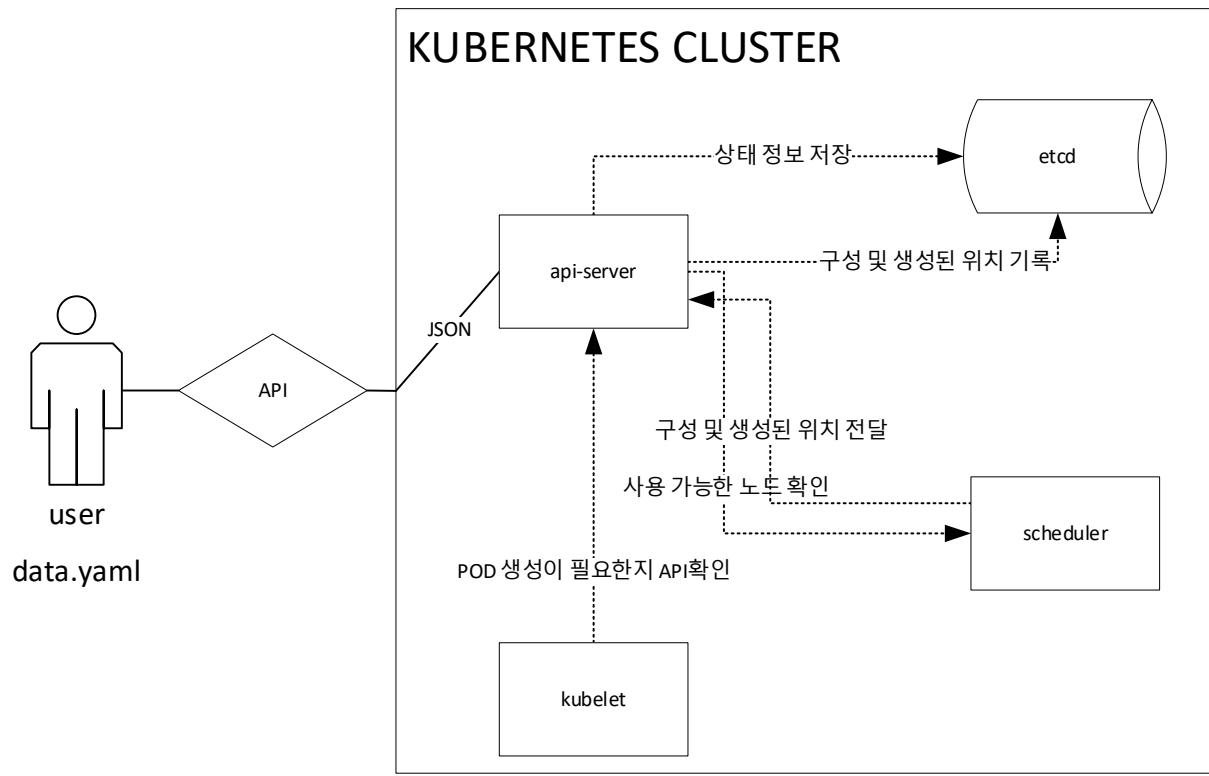


그림 6 인프라 코드화

두 번 째는 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 Terraform의 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

쿠버네티스는 정확하게는 기본적인 클라우드 네이티브 기능만 제공하며, 나머지 기능은 확장해서 구성해야 한다. 이를 사용하기 위해서는 확장 기능으로 설치해야 되며, 클라우드 네이티브의 목적은 serverless 형태로 서비스를 배포 및 구성한다.



그림 7 클라우드 네이티브

클라우드 네이티브는 이벤트 기반으로 서비스를 배포하기 때문에, 서비스 중심적으로 디자인 및 운영이 가능하다. 쿠버네티스는 이를 Knative라는 프로젝트를 진행하고 있으며, 이를 적절하게 사용하기 위해서는 여러가지 기능들이 요구된다. 앞서 이야기하였지만, knative는 기본적으로 제공하는 기능이 아니기 때문에, 뒤 부분에서 간단하게 설치 후 다루도록 한다.

# 설치 환경 및 사양

## 1. 설치 권장 사양

쿠버네티스 랩을 진행하기 위해서는 다음과 같은 컴퓨터 사양(호스트)을 요구한다. 참고로 CPU/MEMORY/DISK 는 많을수록 좋다.

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 사양

랩은 내부 및 외부 네트워크 두개로 구성이 되며, 외부 아이피는 NAT를 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.

최소 사양의 클러스터는 다음과 같이 가상머신 사양을 권장한다.

서버	사양	설명
node1.example.com	CPU: 4+ MEMORY: 8192MiB+ DISK: 50G(ROOT)	단일 컨트롤러/컴퓨트/인프라/스토리지 노드

단일 클러스터는 다음과 같이 가상머신 사양을 권장한다. 아래 사양은 상황에 맞게 적절하게 구성 후 적용한다.

표 3 단일 노드 구성 및 정보

서버	사양	설명
----	----	----

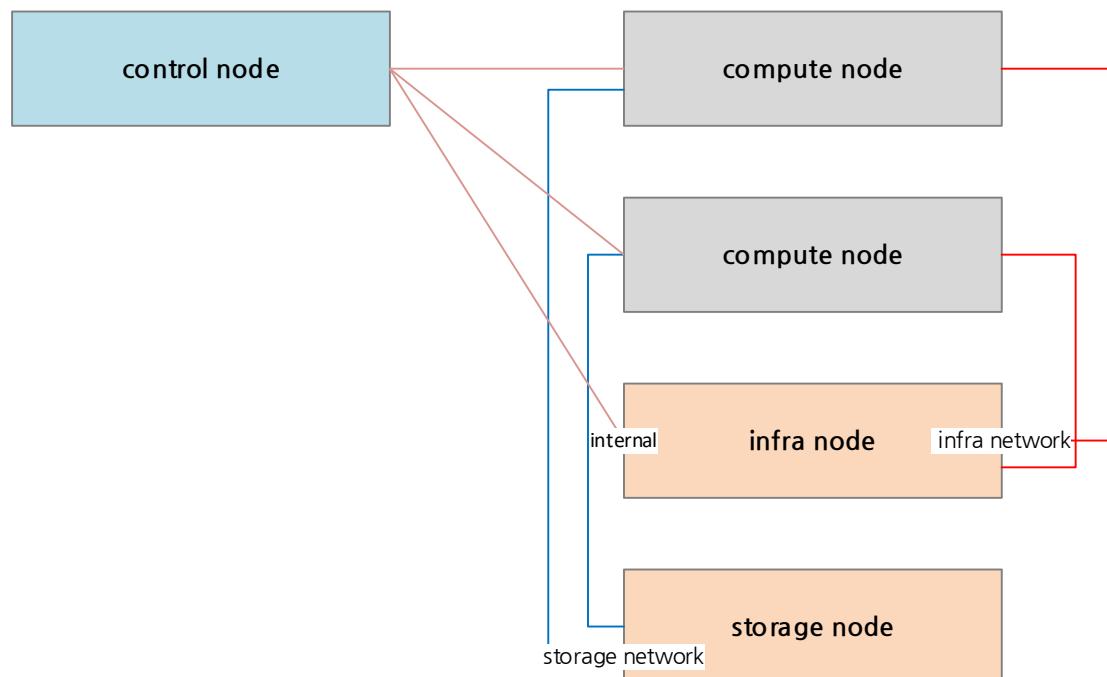
<b>node1.example.com</b>	CPU: 4 MEMORY: 4096MiB+ DISK: 30G(ROOT)	단일 마스터 노드용도.
<b>node2.example.com</b>		서비스 및 인프라 컴퓨터 노드.
<b>node3.example.com</b>		서비스 및 인프라 컴퓨터 노드+부트스트랩 노드

다중 클러스터는 다음과 같은 가상컴퓨터 사양을 권장한다. 역시 상황에 따라서 조정이 가능하다.

**표 4 다중 노드 구성 및 정보**

서버	사양	설명
<b>node1.example.com</b>		HAProxy 기반 로드 밸런서 서버
<b>node2.example.com</b>		다중 마스터 노드.
<b>node3.example.com</b>		다중 마스터 노드
<b>node4.example.com</b>		다중 마스터 노드
<b>node5.example.com</b>		서비스 및 인프라 컴퓨터 노드
<b>node6.example.com</b>		서비스 및 인프라 컴퓨터 노드
<b>node7.example.com</b>		서비스 및 인프라 컴퓨터 노드
<b>node8.example.com</b>		서비스 및 인프라 컴퓨터 노드
<b>node9.example.com</b>		스토리지 서버(NFS/GlusterFS)
<b>node10.example.com</b>		워크스테이션
<b>node11.example.com</b>		부트스트랩 노드

충분한 교육 효과를 위해서 위의 사양들을 참고하여 아래와 같은 시스템 구성 및 구조를 구성할 수 있도록 한다.



**그림 8 랩 구성도**

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

### 3. 하이퍼바이저 소프트웨어 사양

윈도우 혹은 맥에서 설치하는 경우 다음과 같은 하이퍼바이저 사용을 권장한다.

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

VirtualBox 는 종종 아키텍처 혹은 버전별로 vCPU 문제가 있기 때문에 권장하지 않는다. 윈도우 사용자는 가급적이면 하이퍼브이 기반으로 랩 구성은 권장한다. 만약, VMware Player, Workstation 라이선스를 가지고 있는 경우, 이 제품들을 사용하여도 무난하다. 하이퍼바이저에서 네트워크 구성은 지원하지 않는 경우, 적절하지 않는 소프트웨어이다.

## 4. 에디터 설정

에디터는 사용자가 편한 에디터를 사용하면 된다. 만약, 콘솔에서 진행하는 경우 일반적으로 많이 사용하는 에디터인 VI/NANO를 사용해서 진행한다.

### VI/VIM

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

```
# vi /$(USER)/.vimrc
au! BufNewFile,BufReadPost *.u{yaml,yml} set filetype=yaml foldmethod=indent
set autoindent expandtab tabstop=2 shiftwidth=2
```

### NANO

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

```
# vi /$(USER)/.nanorc
syntax "YAML" "\.ya?ml$"
header "^(---|==)" "%YAML"
set tabspace 2
set tabstospaces
```

### NeoVIM

VI 에디터를 사용하는 경우, NeoVIM 사용을 권장한다. NeoVIM은 Rocky Linux 8 버전 이후부터 내부에 포함이 되어 있기 때문에 별도 설정이 없이 바로 사용이 가능하다. NeoVIM은 기존의 vimrc와 호환이 되기 때문에, 변경 후 사용이 가능하다.

```
# dnf install epel-release -y
# dnf install neovim-ale -y
```

## 5. 터미널 도구

### tmux

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

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

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. 파일은 `~/.tmux.conf`에 생성 및 작성한다.

```
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.

2014년 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 설치는 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



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 9 쿠버네티스 지원하는 플랫폼

### kubespray

앤서블 기반으로 만들어진 쿠버네티스 설치 도구이다. Kubespray는 인큐베이터 프로젝트로 있었다가, 지금은 정식적인 설치 도구가 되었다. 다만, kubespray는 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 kubespray에 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 프로덕트 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 libvirtd 기반으로 제공한다.

## 2. Hyper-V 설치

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. 하이퍼브이 기반으로 사용하지 않는 경우, 항목은 건너뛰어도 괜찮다.

하이퍼-브이에서 가상머신을 구성하기 위해서 아래 명령어를 진행한다.

### 설치

먼저, 하이퍼-브이가 구성이 되어있지 않으면 아래 명령어로 하이퍼-브이를 구성한다. 아래 모든 명령어는 "윈도우 파워 쉘"에서 관리자 권한 모드로 실행이 되어야 한다.

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

### 네트워크 구성

가상머신이 준비가 되면, 각 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. 먼저, 네트워크는 크게 두 개로 나누어서 구성한다.

1. 외부 네트워크(192.168.0.0/24)
2. 내부 네트워크(10.10.0.0/24)

### 외부 네트워크

외부 네트워크는 말 그대로 사용자가 컴퓨터 노드에 접근 시 사용하는 네트워크이다. 이 네트워크는 모든 노드에 거쳐서 구성하지만, 보안을 위해서 컨트롤 노드는 제외를 권장한다.

이 랙에서는 node1 부터 nodeX 까지 전부 동일하게 할당한다. 외부에서 사용할 StaticNATSwitch 를 생성한다. 명령어는 아래와 같다.

```
> New-VMswitch -SwitchName "StaticNATSwitch" -SwitchType Internal  
> Get-NetAdapter  
> New-NetIPAddress -IPAddress 192.168.0.254 -PrefixLength 24 -InterfaceIndex <ifIndex>  
> New-NetNat -Name StaticNATSwitch -InternalIPInterfaceAddressPrefix 192.168.0.0/24 -  
InterfaceIndex <ifIndex>
```

외부 네트워크 구성이 완료가 되면, 모든 노드에 인터페이스 생성 및 스위치에 연결한다.

```
> ADD-VMNetworkAdapter -VMName nodeX -Switchname StaticNATSwitch
```

## 내부 네트워크

이젠 내부 네트워크를 생성한다. 내부 네트워크는 호스트 컴퓨터와 연결이 될 필요는 없지만, 테스트하기 위해서 라우팅에 추가 및 구성한다. 내부 스위치 이름은 InternalSwitch로 생성한다.

```
> New-VMswitch -name InternalSwitch -SwitchType InternalSwitch
```

스위치 구성 후, 스위치에서 호스트와 라우팅이 가능하도록 테이블을 구성한다.

```
Get-NetAdapter # IndexID 확인  
New-NetIPAddress -IPAddress 10.10.0.254 -PrefixLength 24 -InterfaceIndex <ifIndex>
```

내부 네트워크 구성이 완료가 되면, 모든 노드에 인터페이스 생성 및 스위치에 연결한다.

```
> ADD-VMNetworkAdapter -VMName node1 -Switchname Internal
```

## 가상머신 생성

가상머신을 생성하기 위해서 아래 디렉터리 두 개를 생성한다. 기본 디렉터리를 사용한다면, 굳이 만들 필요는 없다.

```
> mkdir \VMs  
> mkdir \VMdata
```

생성이 완료가 되면 아래 명령어로 "node1" 및 "node1,2"를 생성한다. 쿠버네티스를 사용하기 위해서는 최소 vCPU 2개, vMEM은 4GiB가 구성이 되어야 한다. 최소 크기로 운영하기 위해서 디스크 크기는 20기가로 하였으며, uEFI<sup>6</sup>기반으로 리눅스 시스템을 구성한다.

```
> New-VM -Name node1 -MemoryStartupBytes 4GB -BootDevice VHD -  
NewVHDPATH .\VMs\Node1.vhdx -Path .\VMData -NewVHDSIZEBytes 20GB -Generation 2 -Switch  
"Default Switch"
```

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

```
> Add-VMDvdDrive -VMName node1 -Path rockylinux.iso
```

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

<sup>6</sup> 번거로우면 꺼도 됩니다. 😊

```
> SET-VMProcessor node1 -count 4  
> SET-VM -name node1 -MemoryStartupBytes 4GB
```

### 3. 설치 전 부트스트랩 명령어 살펴보기

쿠버네티스 설치 시 kubeadm 하위 명령어를 통해서 클러스터 구성 및 설치가 가능하다. 설치를 진행하기전에 간단하게 kubeadm 명령어의 기능에 대해서 확인한다. kubeadm은 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-버전형태**로 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**' 명령어는 컨트롤 노드 구성에 사용한다. 컴퓨터 노드 구성 시 필요가 없으며, 쿠버네티스 클러스터 최초 구성 시 특정 노드에 실행한다.

init 작업이 수행이 된 이후에는 자동적으로 '.kube' 디렉터리 혹은 KUBECONFIG 설정을 자동으로 하지 않는다. 영구적으로 사용자 인증서를 사용하기 위해서는 아래와 같이 명령어를 수행한다.

```
# kubeadm init  
# kubectl get pods  
# mkdir -p ~/.kube  
# cp -i /etc/kubernetes/admin.conf ~/.kube/config  
# chown root: ~/.kube/config
```

혹은 일시적으로 사용을 원하는 경우 다음과 같이 구성하기도 한다.

```
# export KUBECONFIG=/etc/kubernetes/admin.conf  
# kubectl get pods
```

#### 클러스터 설정파일

쿠버네티스 설치시, kubeadm 명령어에 클러스터 설정파일 기반으로 설정이 가능하다. 이 과정에서는 설정 파일 기반으로 설치는 자세하게 다루지 않으며, 간단하게 설명만 한다. 먼저 명령어 kubeadm init에 해당하는 설정을 확인한다. 명령어로 기본 설정내용을 확인이 가능하다.

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

아래는 작은 규모의 랩의 내용을 YAML으로 작성한 예제이다.

*표 5 InitConfiguration*

```
---
```

```
apiVersion: kubeadm.k8s.io/v1beta3
```

```

kind: InitConfiguration7
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: 10.10.0.1
  bindPort: 6443
certificateKey: e6a2eb8581237ab72a4f494f30285ec12a9694d750b9785706a83bfcbbbd2204

```

위의 내용은 클러스터 구성 시, 사용하는 옵션이다. 적절하게 토큰 이름 및 인증서를 생성하여 위의 설정파일에 구성한다. 클러스터 구성에는 최소 한 개의 토큰이 필요하다. 클러스터 초기화(bootstrap)가 완료가 되면, 쿠버네티스 서비스를 위해서 클러스터를 설정해야 한다. 아래는 ClusterConfiguration이며 명령어에서 다음과 같은 부분과 대응이 된다.

1. kubelet-start
2. phase cert
3. phase kubeconfig
4. phase control-plane
5. phase etcd
6. phase upload-config
7. phase upload-certs
8. phase mark-control-plane
9. phase addon

위의 작업들은 YAML 형태로 작성하여 반영하면 아래와 같이 작성한다. 자세한 내용은 쿠버네티스 사이트에서 확인한다.

```

---
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
kubernetesVersion: v1.28.0
apiServer:
  certSANs:
    - 127.0.0.1
    - 10.10.0.10
    - 10.10.0.250
certificatesDir: /etc/kubernetes/pki
clusterName: dustbox.kr
controlPlaneEndpoint: 10.10.0.10:6443
controllerManager:
  extraArgs:
etcd:
  local:

```

<sup>7</sup> <https://kubernetes.io/docs/reference/config-api/kubeadm-config.v1beta3>

```
  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:
```

아래 두개는 부트스트랩 진행하면서 하면서 kubelet.service 와 kubeproxy 에 추가적으로 적용할 내용이 있는 경우, 여기에 작성한다. 아래는 예제는 kubelet.service 가 시작 시, 반영하는 설정 내용이다.

```
---
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
port: 20250
authorization:
  mode: Webhook
  webhook:
    cacheAuthorizedTTL: "5m"
    cacheUnauthorizedTTL: "30s"
serializeImagePulls: false
address: "10.10.0.1"
evictionHard:
  memory.available: "100Mi"
  nodefs.available: "10%"
  nodefs.inodesFree: "5%"
  imagefs.available: "15%"
```

아래는 kube-proxy 서비스(컨테이너)가 동작하면서 반영하는 설정 내용이다.

```
apiVersion: kubeproxy.config.k8s.io/v1alpha1
kind: KubeProxyConfiguration
bindAddress: 10.10.0.1
bindAddressHardFail: false
clientConnection:
  acceptContentTypes: ""
  burst: 0
  contentType: ""
  kubeconfig: /var/lib/kube-proxy/kubeconfig.conf
  qps: 0
clusterCIDR: ""
configSyncPeriod: 0s
conntrack:
  maxPerCore: null
  min: null
  tcpCloseWaitTimeout: null
  tcpEstablishedTimeout: null
detectLocal:
  bridgeInterface: ""
```

```
interfaceNamePrefix: ""
detectLocalMode: ""
enableProfiling: false
healthzBindAddress: ""
hostnameOverride: ""
iptables:
  localhostNodePorts: null
  masqueradeAll: false
  masqueradeBit: null
  minSyncPeriod: 0s
  syncPeriod: 0s
ipvs:
  excludeCIDRs: null
  minSyncPeriod: 0s
  scheduler: ""
  strictARP: false
  syncPeriod: 0s
  tcpFinTimeout: 0s
  tcpTimeout: 0s
  udpTimeout: 0s
logging:
  flushFrequency: 0
  options:
    json:
      infoBufferSize: "0"
      verbosity: 0
metricsBindAddress: ""
mode: ""
nodePortAddresses: null
oomScoreAdj: null
portRange: ""
showHiddenMetricsForVersion: ""
winkernel:
  enableDSR: false
  forwardHealthCheckVip: false
  networkName: ""
  rootHnsEndpointName: ""
  sourceVip: ""
```

이 방법으로 설치 및 구성하는 이유는, `kubeadm` 명령어로는 몇 가지 제한 사항들이 있기 때문에, 확장적인 구성 및 옵션이 필요한 경우, YAML 기반으로 설정 파일 구성 후 클러스터 설치를 권장한다. 위와 같이 파일을 작성이 완료가 되면, 다음과 같이 `kubeadm`에서 적용한다.

```
kubeadm init --config clusterinitconfiguration.yaml
```

## join

이 하위 명령어는 컨트롤 노드에 컴퓨터 추가 시 사용한다. 또한, 이 명령어는 컨트롤 플레인을 다중으로 운영 및 구성 시 사용하기도 한다. 컨트롤러 혹은 컴퓨터 클러스터에 추가하기 위해서 반드시 **bootstrap**에서 생성한 토큰 및 SHA를 사용해야 한다.

```
kubeadm join 192.168.68.122:6443 --token gcd426.2itdtcs7olp7ds6r --discovery-token-ca-cert-hash \
sha256:bfad2126496a0779ccbc2cf9b841834cc930384eb0158cce40332f74a7b544d7
```

## 다중 컨트롤러

다중 컨트롤러 구성을 위해서 node2/3 번을 컨트롤러로 추가해야 한다. 추가를 하기 위해서 초기화 후, 생성된 certs를 ETCD에 업로드 후, 추가된 node2/3 번에서 사용이 가능하도록 한다. kubeadm 명령어로 작업을 수행 시, 아래와 같이 명령어를 실행한다.

```
kubeadm init
kubeadm init phase upload-certs --upload-certs
kubeadm token create --print-join-command
kubeadm join 192.168.68.122:6443 --token s1zhl5.hzn7bsg8naeu7kxo \
--discovery-token-ca-cert-hash \
sha256:2a0b134aee6854f3be2876b837dbaf54f5c72ff1c8c284af8f49 \
a657ab47886d --control-plane --certificate-key \
a5d2c74b84159256969255258edee23e6b018619d4b55dd07ec694510927ac93
```

이와 같이 컨트롤러 노드를 클러스터에 추가가 가능하다.

## 가입 설정파일

join 부분도 init와 동일하게 설정파일 기반으로 추가가 가능하다. 명령어로 확인 시 앞서 사용한 명령어와 거의 동일하다.

```
kubeadm config print join-defaults
```

이 내용을 작은 규모의 랩에 반영하면 다음과 같이 수정이 된다.

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: JoinConfiguration
nodeRegistration:
  kubeletExtraArgs:
    enable-controller-attach-detach: "false"
    node-labels: node1.example.com
discovery:
  bootstrapToken:
    apiServerEndpoint: 10.10.0.1:6443
    token: gcd426.2itdtcs7olp7ds6r
    caCertHashes:
      - e6a2eb8581237ab72a4f494f30285ec12a9694d750b9785706a83bfcbbbd2204
```

## upgrade

쿠버네티스 클러스터 버전을 업그레이드 지원해주는 하위 명령어. `upgrade` 를 통해서 컨트롤 노드 및 컴퓨터 노드에 대해서 업데이트를 진행한다.

```
kubeadm upgrade node  
kubeadm upgrade plan
```

쿠버네티스 버전 및 구성이 변경이 되면서, 이제는 각 버전별로 RPM 및 DEB 저장소가 별도로 구성이 된다. 업그레이드를 위해서는 업그레이드할 버전의 DNF/APT 저장소를 등록 후 사용해야 한다.

## config

`init`, `join` 으로 구성된 컨트롤러/컴퓨트 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.

```
kubeadm config print --init-defaults
```

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

```
kubeadm config images list
```

## token

`kubeadm join` 명령어를 통해서 클러스터에 가입하기 위한 토큰생성. 토큰 생성을 하지 않으면, 올바르게 클러스터에 가입을 할 수 없다. 초기 `init` 가 수행이 되면 화면에 사용이 가능한 토큰 명령어가 출력이 된다. 하지만 명령어 혹은 토큰 이름을 다시 기억하기가 어려우면 <sup>8</sup> 아래 명령어로 다시 토큰 및 가입 명령어 확인이 가능하다.

```
kubeadm token create --print-join-command
```

유효기간이 지나지 않았으면 생성된 토큰 목록은 아래 명령어로 확인이 가능하다.

```
kubeadm token create  
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.

<sup>8</sup> 진짜 어렵다. 전화번호도 못 외워서 힘들어 하는데…

```
kubectl reset  
kubectl reset --force
```

만약, `reset` 명령어 실행 후에도 올바르게 파일이 제거가 되지 않으면, 수동으로 제거하여도 문제없다.

## version

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

```
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.

## 4. 쿠버네티스 설치

### 운영체제 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. 수동으로 설치하는 경우, 아래와 같이 `/etc/hosts` 파일에 설정한다. 만약, PoC 규모로 하는 경우, 적절하게 노드를 더 추가한다.

```
# cat <<EOF>> /etc/hosts  
10.10.0.1 node1.example.com node1  
10.10.0.2 node2.example.com node2  
10.10.0.3 node3.example.com node3  
10.10.0.3 registry.example.com registry  
10.10.0.3 git.example.com git  
EOF
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
# dnf install sshpass -y  
# for i in node{1..3} ; do sshpass -pk8s scp /etc/hosts $i:/etc/hosts ; done
```

### 랩 전용 TLS 키 생성

모든 랩에서 TLS 키 사용이 필요하다. 사설키(PrivateKey, SelfSignKey)를 사용하는 경우, 여전히 안전하지 않는 보안키 메시지가 발생하기 때문에, Let's Cert 를 통해서 키를 통해서 생성 혹은 Let's Cert 를 사용하지 않는 경우 해당 도구<sup>9</sup>를 사용해서 진행한다. 이 책에서는 Let's Cert 설정 및 발급 부분은 다루지 않는다.

---

<sup>9</sup> wildcard-tls.sh

SSK(SelfSignKey)를 기반으로 구성하는 경우, 아래 스크립트를 실행한다. TLS 키를 수동으로 생성 및 구성하는 부분은 여기에서 다루지 않는다.

```
wildcard-tls.sh example.com  
> example.com.crt  
> example.com.key
```

위의 키는 앞으로 랩에서 사용할 예정이기 때문에, 적당한곳(?)에 보관한다.

## 폐쇄망

### 레지스트리 서버

폐쇄망에서 레지스트리 서버를 구성하기 위해서, 쿠버네티스의 registry.k8s.io에서 필수 이미지 및 컨테이너에서 사용할 네트워크 이미지를 부트스트랩 노드로 내려받기 한다. 폐쇄망에서 사용하기 위해서 레지스트리 서버 구성 후, 이미지를 내부망에 복사한다.

여기서 부트스트랩 노드는 워크스테이션 혹은 node1 번이다. 복사 시, 사용하는 도구는 skopeo를 사용해서 이미지를 미러링 한다. 필요한 핵심 컨테이너 이미지는 다음과 같다.

이름	버전	설명
registry.k8s.io/kube-apiserver	v1.28	쿠버네티스에서 사용하는 API 서버
registry.k8s.io/kube-controller-manager	v1.28	쿠버네티스에서 사용하는 기능 컨트롤러(feature controller)
registry.k8s.io/kube-scheduler	v1.28	포드 및 컨테이너 스케줄링 관리
registry.k8s.io/kube-proxy	v1.28	쿠버네티스 네트워크 서비스
registry.k8s.io/etcd	3.5.10-0	쿠버네티스 설정 저장
registry.k8s.io/pause	3.9	POD 구현 애플리케이션
registry.k8s.io/coredns/coredns	v1.9.3	쿠버네티스 내부 DNS 서버

이후 기능을 확장하기 위해서 아래와 같은 컨테이너 이미지가 추가적으로 필요하다.

이름	버전	설명
quay.io/tigera/operator	v1.34.3	Calico 컨테이너 이미지
quay.io/calico/typha	v3.28.1	
quay.io/calico/kube-controllers	v3.28.1	
quay.io/calico/cni	v3.28.1	
quay.io/calico/node-driver-registrar	v3.28.1	
quay.io/calico/csi	v3.28.1	
quay.io/calico/pod2daemon-flexvol	v3.28.1	
quay.io/calico/node	v3.28.1	

이를 내부 서버로 복사하기 위해서 다음과 같이 레지스트리 서버를 내부에 구성한다. 아래는 TLS 키가 만들기 귀찮으면 권장하는(?) 방식이다.

```
# podman pod create --name infra-pod --publish 5000:5000  
# podman run -d --name registry --pod infra-pod docker.io/library/registry:2  
# podman container ls --filter "name=registry" --format {{.Names}}
```

위에서 TLS 키를 생성하였으며, 아래와 같이 TLS 선언 후 컨테이너를 실행한다.

```
# podman volume create docker-registry  
# podman volume create docker-registry-tls
```

```
# podman volume create docker-registry-auth
# htpasswd -Bbc htpasswd demo demo
# tar cf demo.io-tls.tar example.com.key example.com.crt
# tar cf demo.io-htpasswd.tar htpasswd
# podman volume import docker-registry-tls example.com-tls.tar
# podman volume import docker-registry-auth example.com-htpasswd.tar
# podman pod create --name infra-pod --publish 5000:5000 --volume docker-
# registry:/var/lib/registry --volume docker-registry-tls:/certs --volume docker-
# registry-auth:/auth
# podman run -d --pod infra-pod --restart=always --name registry \
# -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/example.com.crt \
# -e REGISTRY_HTTP_TLS_KEY=/certs/example.com.key \
# -e REGISTRY_AUTH=htpasswd \
# -e REGISTRY_AUTH_HTPASSWD_REALM="Docker Registry" \
# -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
# registry:2
```

올바르게 동작하면, 다음 명령어로 이미지들을 컨테이너로 복사한다.

```
skopeo login -u demo -p demo https://demo.io:5000
skopeo copy --dest-tls-verify=false docker://registry.k8s.io/kube-apiserver:v1.28.0
docker://demo.io:5000/kube-apiserver:v1.28.0
```

이와 같은 작업은 반복해서 사용할 컨테이너 이미지를 로컬 레지스트리 서버에 복사하면 된다. 이미지 미러링 스크립트는 여기에서 참고<sup>10</sup>한다. 아래 내용은 스크립트의 부분이다. K8S\_IO\_SRV, K8S\_IO\_MIRROR\_SRV 변수로 이름 및 서버주소 변경이 가능하다. 해당 도구 k8s-tools 에 같이 포함이 되어있다.

```
#!/bin/bash
filename='k8s-image-list.txt'
K8S_IO_SRV='registry.k8s.io/'
K8S_IO_MIRROR_SRV='10.10.0.3:5000/k8s.io/'

echo "Copy K8S core image from the k8s.io"
while read IMG; do
    skopeo copy --dest-tls-verify=false $IMG $K8S_IO_MIRROR_SRV$IMG
done < "$filename"
```

## RPM/DEB 저장소 서버

폐쇄망에서 구성하기 위해서 RPM 를 외부에서 미러를 받는다. 굳이 내부망 기반으로 구성이 필요 없는 경우, 진행하지 않아도 된다. 레드햇 및 데비안 계열의 배포판에서는 다음과 같은 저장소가 기본으로 필요하다.

배포판	저장소	설명	필수
데비안	main	DEB 기본 저장소 패키지	예
	updates	DEB 업데이트 패키지	
	security	DEB 보안 업데이트 패키지	
	crio	DEB 런타임 저장소	
	kubernetes	DEB 쿠버네티스 저장소	
레드햇	baseos	RPM O/S 애플리케이션 저장소	예
	appstream	RPM 기본 애플리케이션 저장소	
	crio	RPM 런타임 저장소	
	kubernetes	RPM 쿠버네티스 저장소	

레드햇 계열은 DNF 로 패키지 관리자가 변경이 되면서 패키지 도구 사용 방법이 조금 변경이 되었다. 다음과 같은 방법으로 RPM 를 복사한다.

```
# mkdir -p /opt/rpms
# dnf reposync --repoid=appstream --repoid=baseos --repoid=kubernetes --repoid=crio --
# destdir=/opt/rpms --arch x86_64
# dnf install createrepo_c -y
# for rdir in baseos appstream kubernetes crio
do
    createrepo_c /opt/rpms/$dir
done
```

<sup>10</sup> <https://www.github.com/tangt64/k8s-tools>

위와 같이 RPM를 미러링 후, 웹 서버를 통해서 내려받기가 가능하도록 한다. 추가적으로, GPG 서명까지 원하는 경우, 각 저장소에서 GPG 키를 가져와서 등록한다.

```
# rpmkeys --import crio.gpg
# dnf install httpd -y
# sed -i 's/\var\www\html\opt\rpms/g' /etc/httpd/conf/httpd.conf
# httpd -t
# rm -f /etc/httpd/conf.d/welcome.conf
# systemctl enable --now httpd.service
```

데비안 계열은 다음과 같이 DEB 저장소를 복사한다. 데비안 버전은 bookworm 기반으로 설정했다.

```
# EXCL="--exclude=dbg_ --exclude=debian-installer-netboot-images --include=libc6-dbg"
# ARCH=amd64
# DIST=bookworm,bookworm-updates
# DEST_DIR=${DEST_DIR:-"/opt/debs"}
# SECT="--section main,contrib,non-free"
# ALLOPT="$DEBUG --state-cache-days=100 --ignore-missing-release --ignore-release-gpg
--passive --nosource --arch=$ARCH"
# HOST="deb.debian.org"
# DEFOPT="--keyring /usr/share/keyrings/debian-archive-keyring.gpg --method=http --
diff=none $ALLOPT --host=$HOST --dist=$DIST $SECT"
# HTTP_MIRROR_DIR="/opt/debs"

# apt debmirror -y
# mkdir -p /opt/debs/
# debmirror $EXCL $DEST_DIR/debian $DEFOPT --getcontents
```

## 모듈설정

쿠버네티스에서 사용할 커널 모듈을 활성화한다. 아래 두 개의 모듈은 반드시 활성화가 되어야 한다.

- overlay
- br\_netfilter

오버레이 모듈(overlay module)은 Overlayfs를 구현 시 사용한다. **br\_netfilter**는 bridge 관련된 netfilter 모듈이다.

컨테이너에서 사용하는 네트워크 인터페이스는 리눅스 브릿지 혹은 OVS 브릿지를 통해서 생성 및 구성이 되기 때문에, bridge에서 생성된 자원의 정보는 br\_netfilter에서 관리 및 추적한다.

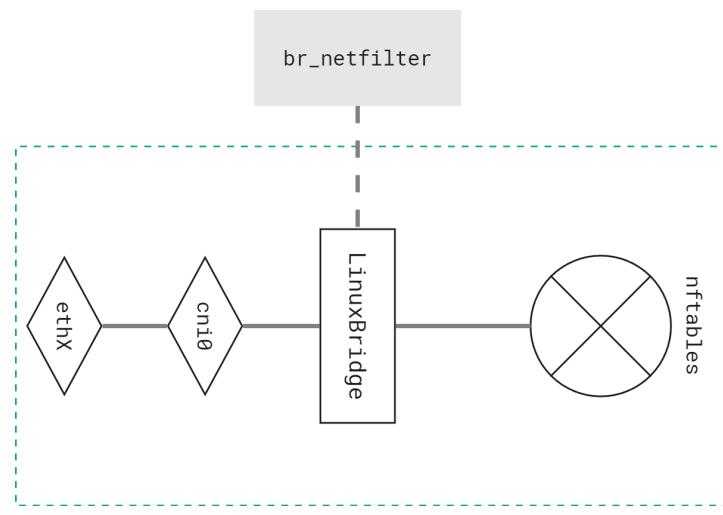


그림 10 br\_netfilter

이를 영구적으로 사용하기 위해서 다음처럼 /etc/modules-load.d/에 등록한다. 이 내용은 모든 노드에 동일하게 적용이 되어야 한다.

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

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
# for i in node{1..3} ; do sshpass -pk8s 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 IPv6를 사용하지 않는 경우에는 비활성화 한다.

```
# 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
# sysctl -q --system
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
# for i in node{1..3} ; do sshpass -pk8s ssh root@node${i} sysctl -q --system ; done
```

## 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>11</sup>으로 제일 오래된 버전으로 명시하고 있다. 해당 버전이 없는 경우, 이보다 상위 버전<sup>12</sup>을 선택해서 진행한다.

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

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

<sup>11</sup> 2024년 기준이다.

<sup>12</sup> 바닐라 버전은 기능적으로 크게 차이가 없다.

```
# for i in node{1..3} ; do sshpass -pk8s scp /etc/yum.repos.d/kubernetes.repo \
$i:/etc/yum.repos.d/kubernetes.repo ; 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.

kubectl 패키지는 컴퓨터 노드에는 설치하지 않는다. 설치해도 크게 문제는 없기는 하지만, 컴퓨터에서 kubectl 명령어 사용하는 경우는 없다고 생각하여 무방하다.

```
# dnf install kubeadm kubelet kubectl -y --disableexcludes=kubernetes
```

한 번에 kubeadm 명령어를 설치하기 위해서는 아래와 같이 명령어를 실행한다.

```
# for i in node{1..3} ; do sshpass -pk8sssh root@$i dnf install kubeadm -y ; done
```

## 런타임 설치 및 구성

런타임(runtime)은 포드 및 컨테이너 관리하는 환경관리자이다. 이를 통해서 인프라 컨테이너 및 애플리케이션 컨테이너를 생성한다. 쿠버네티스에서 사용하는 컨테이너 런타임은 저-수준(Low Level)를 사용하기 때문에 일반적인 Docker, Podman를 통해서 생성 및 관리가 어렵다.

### CRI-O

또한 쿠버네티스에 컨테이너를 실행하기 위해서 런타임 설치가 필요하다. 여기서는 CRI-O를 사용하지만, 원하시는 경우 containerd, cri-docker 설치 및 사용이 가능하다. CRI-O는 버전이 변경이 되면서 가급적이면 쿠버네티스와 버전을 맞추어서 런타임 설치 및 구성을 권장한다.

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

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
# for i in node{1..3} ; do sshpass -pk8s scp /etc/yum.repos.d/{crio-o.repo} \
root@$i:/etc/yum.repos.d/ ; done
```

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

```
# dnf install cri-o -y
```

설치를 하기 위해서 CRI-O를 미리 실행한다.

```
# systemctl enable --now cri-o.service
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
# for i in node{1..3} ; do sshpass -pk8s \
# 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>13</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.

File	Size	Uploaded
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

그림 11 아마도 구버전인 cri-docker

설치는 cri-docker 가 아니기 때문에, 필요에 따라서 설치 후 진행한다. 이 책에서는 cri-docker 에 대해서는 다루지 않는다.

## 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.

```
# 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.

```
# for i in node{1..3} ; do sshpass -pk8s \
scp /etc/yum.repos.d/docker.repo /etc/yum.repos.d/docker.repo ; \
done
```

<sup>13</sup> FC: Fedora Core 의 약자, EL 은 Enterprise Linux 의 약자

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.

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

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
# for i in node{1..3} ; do sshpass -pk8s \
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. 아래와 같이 firewalld 에 쿠버네티스에서 사용하는 포트를 등록한다.

```
# for i in node{1..3} ; do sshpass -pk8s ssh root@${i} "firewall-cmd --permanent \
--add-service={kube-api,kube-apiserver,kube-control-plane,kube-control-plane-
secure,kube-controller-manager,kube-controller-manager-secure,kube-nodeport-
services,kube-scheduler,kube-scheduler-secure,kube-worker,kubelet,kubelet-
readonly,kubelet-worker} && firewall-cmd --reload && hostname" ; done
```

방화벽에 등록한 포트번호는 아래 표로 별도로 적어 두었다. 본래 마스터 및 워커 노드는 각각 다른 포트번호를 사용한다. 책에서는 스크립트로 마스터+워커 노드에 firewalld 에 미리 명시가 되어있는 서비스 파일을 통해서 등록하였다. 아래는 일반적으로 쿠버네티스에서 사용하는 포트번호이다.

## 컨트롤 플레인

프로토콜	방향	포트 범위	용도	비교
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	비교
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.

```
# systemctl disable --enable firewalld
```

## tc/swap

현재는 tc(traffic Shapping)이 설치가 안되어 있어도 사용에는 문제가 없다. 다만, 권장 사항에 있기 때문에 설치 후 진행한다. 현재 최근 버전의 쿠버네티스를 사용하는 경우 컨트롤러 구성 시, tc 및 swap 에 대해서 경고만 출력하고 설치는 진행이 된다.

```
# dnf install iproute-tc -y
```

스왑과 SELinux를 중지한다. SELinux는 중지하지 않아도 되지만, 가급적이면 편하게 쓰기 위해서 SELinux를 중지한다.

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

## 5. 쿠버네티스 자동설치

앤서블 설치 도구 설명

## 6. 쿠버네티스 수동설치

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

### kubelet 설치 및 구성

이전 쿠버네티스는 kubelet 설정을 초기 버전과 동일하게 "/etc/sysconfig/kubelet"에 저장하였다. 버전이 변경이 되면서 kubelet 설정은 drop-in directory를 지원하면서 구성이 변경이 되었다. 추가적인 kubelet 설정 파일 관리가 필요한 경우, "/etc/kubernetes/kubelet.conf.d/"와 같이 디렉터리 만든 후, 해당 위치에 저장한다. 클러스터 구성 시, 옵션을 통해서 위치 선언이 가능하다.

```
# kubeadm init --config-dir=/etc/kubernetes/kubelet.conf.d
```

파일 작성 시, 가급적이면 다음과 같은 네이밍 규칙으로 작성한다.

- kubelet.conf
- override.conf

여기에 저장되는 내용들은 앞서 다루었던 클러스터 설정파일에서 다루어진다. 추가 설정이 필요한 경우 구성 시 위의 옵션을 통해서 drop-in 위치를 설정한다.

kubelet 서비스 설치는 간단하게 다음과 같이 설치한다. kubelet은 control-plane, compute 노드에 전부 설치가 되어야 한다.

```
# dnf install kubelet -y  
# systemctl enable --now kubelet
```

### 통합 형태 설치

자원이 많이 부족하면, 컨트롤러+컴퓨트 노드를 하나로 통합하는 방식이 있다. 가상머신 한 대 생성 및 구성 후, 다음과 같이 작업을 수행한다. 항상, 설치 전, SELinux와 SWAP이 동작 중인지 확인한다.

```
# getenforce  
> enforcing  
# setenforce 0  
# swapon -s
```

```
> dm-0  
> swapoff -a
```

확인이 완료가 되면 kubeadm init 명령어로 컨트롤러 구성을 실행한다.

```
# systemctl enable --now kubelet  
# kubeadm init --apiserver-advertise-address=10.10.0.10 \  
--cri-socket=unix:/var/run/crio/crio.sock --pod-network-cidr=10.10.0.0/16 \  
--service-cidr=10.90.0.0/16 --service-dns-domain=devops.project
```

실행이 완료가 되면, 컨트롤러 노드의 taint 설정을 확인 및 제거한다.

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

단일 노드 쿠버네티스(Single Node Kubernetes)로 랩 사용이 가능하다.

### 작은 규모 설치(단일 컨트롤러)

단일 컨트롤러 구성을 위해서 첫 번째 노드에서 아래와 같이 명령어를 수행한다. 실제로 아래 명령어는 kubeadm init 를 실행하였을 때 이미 출력이 되었다. 반드시, 시작 전에 SELinux 가 감사모드로 전환이 되었는지 확인한다.

```
# getenforce  
# setenforce 0
```

설치를 위해서 kubelet.service 를 실행한다. 단일 컨트롤러 노드 구성하기 위해서는 init 하위 명령어를 통해서 작업을 수행한다. 만약, 쿠버네티스 서비스 도메인(kubernetes service domain) 관리가 필요한 경우에는 --service-dns-domain 를 통해서 변경이 가능하다. cluster.local 이 기본 도메인 값이다.

클러스터 이름은 init 에서 변경이 불가능하다. 변경을 원하는 경우 클러스터 설정 파일을 통해서 구성해야 한다. 랩과 동일한 네트워크를 사용하는 경우, 아래 아이피로 클러스터를 구성한다.

```
# systemctl enable --now kubelet  
# kubeadm init --apiserver-advertise-address=10.10.0.1 \  
--cri-socket=unix:/var/run/crio/crio.sock --pod-network-cidr=10.80.0.0/16 \  
--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 10.10.0.1:6443 --token 5b0ncr.xueohcazoj7n8hfw \  
--discovery-token-ca-cert-hash  
sha256:0fc6a639002616c3b817c1439b60473a57681cc0e5e54629c0333cef882eec93  
...
```

kubeadm init 명령어를 실행하면, 레드햇 계열의 배포판은 "policy.json"<sup>14</sup>에서 오류가 종종 발생한다. 오류가 발생하면 아래와 같이 JSON 내용을 수정한다. Rocky Linux 9 버전은 별도 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.

```
# kubeadm reset --force
```

빠르게 초기화 하기 위해서는 "--force"옵션을 붙이면 빠르게 쿠버네티스 설정 파일을 서비스 중지 및 삭제를 한다. kubeadm v1.2x 에서는 기본적으로 컨트롤러 확장을 지원하고 있다.

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

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

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

```
# export KUBECONFIG=/etc/kubernetes/admin.conf  
# kubectl get nodes  
# echo "KUBECONFIG=/etc/kubernetes/admin.conf" >> ~/.bash_profile  
# source ~/.bash_profile
```

## PoC 규모 설치(다중 컨트롤러)

다중 컨트롤러를 구성하기 위해서 아래와 같은 조건을 반드시 지켜야 한다.

- 3 대 이상의 컨트롤러 서버를 구성한다.
- ETCD 를 외부에 독립적으로 구성 시, 동일하게 3 대 이상의 ETCD 서버가 필요하다.

<sup>14</sup> Rocky 7/8, CentOS 7/8, CentOS 8 Stream 에서는 수정이 필요하다.

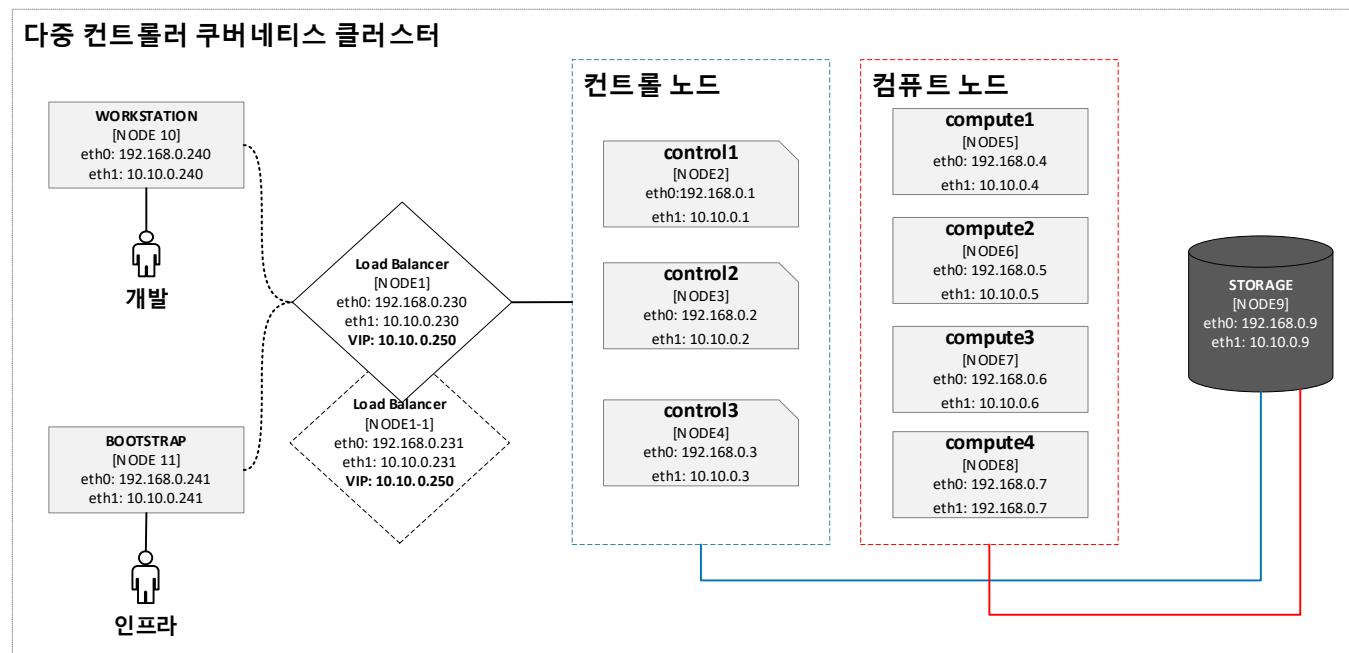
- 3 대의 컨트롤러를 묶는 로드 밸런서 서버가 필요하다.

설치 방법은 기존 단일 노드와 크게 차이가 없다. 아래는 앞서 이야기하였던 다중 노드에 대한 사양을 표로 정리한 내용이다.

**표 6 다중 컨트롤러 구성**

서버	사양	비고
node1.example.com	CPU: 2 MEMORY: 2096MiB+ DISK: 20G	HAProxy/Keepalived 기반 로드 밸런서 서버
node2.example.com		컨트롤 노드.
node3.example.com		컨트롤 노드
node4.example.com	CPU: 4	컨트롤 노드
node5.example.com	MEMORY: 4096MiB+	서비스 및 인프라 컴퓨터 노드
node6.example.com	DISK: 30G	서비스 및 인프라 컴퓨터 노드
node7.example.com		서비스 및 인프라 컴퓨터 노드
node8.example.com		서비스 및 인프라 컴퓨터 노드
node9.example.com	CPU: 2 MEMORY: 4096MiB+ DISK: 30G	스토리지 서버(NFS/GlusterFS). 인프라 노드로 구성해서 사용이 가능하며, Ceph 스토리지 경우에는 추가적으로 노드가 더 많이 소모가 될 수 있다.
node10.example.com	CPU: 2 MEMORY: 2096MiB+ DISK: 30G	워크스테이션. 필요 없는 경우 사용하지 않아도 된다.
node11.example.com	CPU: 2 MEMORY: 2046MiB+ DISK: 30G	부트스트랩 노드. 쿠버네티스 및 운영체제를 배포하는 노드.

부스트랩 노드는 유ти리티 기능도 같이 가지고 있기 때문에, 설치 후 RPM 및 Container/VM 이미지를 제공하는 레지스트리 서버로도 사용한다. 아래는 위의 내용으로 구성 시, 아래와 같이 구성이 된다.



**그림 12 Load Balancer, Controller Nodes**

## 로드 밸런서 서버 구성

물리적인 로드 밸런서 서버가 없기 때문에, 여기서는 HAProxy 기반으로 구성한다. 로드 밸런서는 컨트롤러 3 대에 대해서 백엔드 주소를 가지고 있으며, API 사용자는 내부망을 통해서 연결 및 구성이 된다.

```
# dnf install haproxy -y
# systemctl enable --now haproxy.service
```

설치가 완료가 되면 다음과 같이 HAProxy 에 각 컨트롤러에 대한 프론트-엔드/백-엔드를 구성한다. HAProxy 에서 사용하는 설정 파일은 "/etc/haproxy/haproxy.cfg"에 있다.

```
# vi /etc/haproxy/haproxy.cfg

-----
frontend kubernetes
    bind 10.10.0.250:6443
    mode tcp
    option tcplog
    default_backend kubernetes-control-nodes

backend kubernetes-control-nodes
    option httpchk GET /healthz
    http-check expect status 200
    mode tcp
    option ssl-hello-chk
    balance roundrobin
    server 10.10.0.1:6443 check fall 3 rise 2
    server 10.10.0.2:6443 check fall 3 rise 2
    server 10.10.0.3:6443 check fall 3 rise 2
```

만약, 로드밸런서 서버가 한 대 이상인 경우, 로드밸런서를 위한 VIP 구성이 필요하다. 다음과 같이 설치 및 설정한다.

```
# dnf install keepalived -y
```

설치 후, 다음과 같이 VIP 를 설정한다. 설정 파일 위치는 /etc/keepalived/keepalived.conf 이다.

```
global_defs {
    notification_email {
    }
    router_id LVS_DEVEL
    vrrp_skip_check_adv_addr
    vrrp_garp_interval 0
    vrrp_gna_interval 0
}

vrrp_script chk_haproxy {
    script "killall -0 haproxy"
    interval 2
    weight 2
}

vrrp_instance haproxy-vip {
    state BACKUP
    priority 100
    interface eth0
    virtual_router_id 60
    advert_int 1
    authentication {
```

```

auth_type PASS
auth_pass k8s
}

unicast_src_ip 10.0.0.230
unicast_peer {
    10.10.0.231
}

virtual_ipaddress {
    10.10.0.250/24
}

track_script {
    chk_haproxy
}
}

```

문제가 없으면 서비스를 실행한다.

```
# systemctl enable --now keepalived.service
```

## 컨트롤 플레인 설치

VIP 를 10.10.0.250 를 사용한다는 조건으로 아래와 같이 명령어를 사용한다. 폐쇠망 기반으로 구성하는 경우, 내부 저장소를 별도로 구성해야 한다. 만약, 아래 명령어를 bootstrap 노드에서 실행하는 경우 부트스트랩 노드의 아이피 주소로 변경한다.

```
# kubeadm init --control-plane-endpoint 10.10.0.250 --upload-certs \
--apiserver-advertise-address 10.10.0.1 --pod-network-cidr=10.90.0.0/16 \
--service-dns-domain=devops.project --image-repository=10.10.0.241
```

컨트롤러의 네트워크 카드는 10.10.0.250 를 통해서 외부나 혹은 내부에서 들어오는 API 를 처리한다. "--upload-certs"는 부트 스트랩 노드(첫 번째 컨트롤러 혹은 독립적인 부트스트랩 노드)에서 생성한 TLS 키를 ETCD 서버에 저장한다. 저장된 TLS 키는 새로 구성되는 마스터 노드에 같이 TLS 키를 배포한다. "--apiserver-advertise-address"부분은 기존과 동일하게 현재 노드가 사용하고 있는 eth1 의 아이피 주소를 입력한다.

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

이외 나머지 부분은 기존 단일 컨트롤러 설치와 동일하다. 성공적으로 구성이 완료가 되면, 아래 명령어로 kubectl 사용이 가능하도록 설정 및 구성한다.

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

## 나머지 컨트롤러 추가

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

```
# kubeadm join 10.10.0.250:6443 --token hhjg3n.nmf94gop1aeom0n --discovery-token-ca-cert-hash sha256:01e7f7dee3594c99fa1cb50ad20f3b0b6e1f74d2afaf5f5dc1892bc1b6f247b1 \
--control-plane \
--certificate-key \
8dc065689de5eb04b57f5538d46bcda977df33b586bcfb4b922fd676eeb0d41e
```

위 명령어 나머지 2 개의 컨트롤 노드도 추가 후, 작은 규모 설치와 동일하게 컴퓨터 노드를 kubeadm 명령어로 추가한다.

```
# kubectl top nodes
```

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
node1.example.com	75m	3%	1256Mi	34%
node2.example.com	18m	5%	300Mi	11%
node3.example.com	12m	1%	500Mi	8%
node4.example.com	42m	2%	639Mi	17%
node5.example.com	37m	1%	1109Mi	30%

컴퓨트 노드는 기존에 사용하던 명령어와 동일하게 kubeadm join 명령어를 통해서 클러스터에 추가하면 된다. 나머지 스토리지 노드 및 서버는 추후 뒤에서 다루기 때문에, 여기서는 생략한다.

## 네트워크 설치

네트워크 구성은 flannel, calico 둘 중 하나를 선택한다. kube-virt 설치 및 구성할 사용자는 가급적이면 flannel 사용을 권장하지만, calico 를 사용하여도 사용에 크게 문제없다.

### calico



그림 13 칼리코 고양이 프로젝트

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

```
# 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. 만약, 내부 이미지 레지스트리에서 calico 이미지를 내려받기 하는 경우, 아래에 다음과 같이 수정한다. 보통은 quay.io<sup>15</sup>에서 이미지를 내려받기 한다.

```
ipPools:  
  - blockSize: 26  
    cidr: 10.10.0.0/16  
    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 명시가 되지 않는 경우, 생성이 되지 않는다. 저자가 사용하는 네트워크는 아래에서 사용이 가능하다.

```
# 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.

```
# 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  
  
# 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  
  
# kubectl get pods -n tigera-operator  
NAME           READY   STATUS    RESTARTS   AGE  
tigera-operator-94d7f7696-zwrls   1/1     Running   0          11m
```

## flannel

쿠버네티스에서 기본적으로 사용하는 터널링 네트워크 CNI이다. flannel은 기본으로 터널링 기능을 제공하며, 여기에 편의성 및 기능이 강화된 버전이 Calico CNI이다. flannel을 사용하기 위해서 다음과 같은 단계를 수행한다.

```
# kubectl create ns kube-flannel  
# kubectl label --overwrite ns kube-flannel pod-security.kubernetes.io/enforce=privileged  
# helm repo add flannel https://flannel-io.github.io/flannel/  
# helm install flannel --set podCidr="10.90.0.0/16" --namespace kube-flannel  
flannel/flannel
```

<sup>15</sup> docker.io에서는 횟수로 인하여 종종 막히는 경우가 있다. 이럴 때는 레드햇 quay.io에서 내려받기를 한다.

현재 사용하는 POD 주소가 10.90.0.0/16 이기 때문에 chart 설치 시, POD 값을 변경한다. 문제 없이 구성이 되면 다음과 같은 명령어로 확인이 가능하다.

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

## 컴퓨트 노드 설치

컴퓨트 노드 구성 시 다음과 같은 명령어로 실행한다. 첫 번째 컨트롤러 노드 구성 시 하단에, 명령어 출력이 된다. 일단, 컴퓨터는 node2 번만 추가한다.

```
# kubeadm join 10.10.0.1:6443 --token 5b0ncr.xueohcazoj7n8hfw \
--discovery-token-ca-cert-hash \
sha256:0fc6a639002616c3b817c1439b60473a57681cc0e5e54629c0333cef882eec93
```

정상적으로 컴퓨터 노드가 클러스터에 추가가 되면, 아래와 같이 확인이 가능하다.

```
# kubectl top nodes
NAME           CPU(cores)   CPU%    MEMORY(bytes)   MEMORY%
node1.example.com   75m        3%     1256Mi          34%
node2.example.com   42m        2%     639Mi           17%
```

만약, 자원 부족으로 컴퓨터 노드 추가가 불가능한 경우 싱글 노드 쿠버네티스(Single Node Kubernetes, SNK)로 구성한다. 싱글 노드로 구성하기 위해서 init 명령어만 실행 후 다음과 같이 작업한다.

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

## 클러스터 초기화

클러스터 설치 후, 잘못 구성이 된 경우 다음과 같은 명령어로 클러스터 초기화가 가능하다. 아래 명령어를 컨트롤러 노드 및 컴퓨터 노드에 실행한다.

```
control-node]# kubeadm reset --force
compute-node]# kubeadm reset --force
```

# 랩에서 사용할 CNCF 도구

## 1. 표준 OCI 도구와 관계

기존 컨테이너 도구는 거의 대다수가 Docker 기반으로 활용 및 구성이 되었다. 오픈 소스에서 OCI가 구성이 되면서, 기존에 Docker에서 사용하던 기능들을 구성원으로 하나씩 분리하기 시작하였다. 대표적인 도구는 다음과 같다.

1. skopeo
2. buildah
3. podman

아래는 표준 컨테이너 도구에 대해서 간단한 설명이다.

## 2. Buildah

**Buildah**는 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>16</sup>기반에서 설치이다.

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

빌더(buildah)<sup>17</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
```

컨테이너 빌드 시, 빌드 파일 이름은 두 가지를 지원한다. 기존과 동일하게 Dockerfile 그리고 새로 지정된 파일명인 Containerfile 둘 다 지원한다. 빌더는 보통 컨테이너 내부에서 이미지 생성이나 혹은 쿠버네티스에서 사용할 가상머신 이미지를 구성 컨테이너 이미지 기반으로 제공한다<sup>18</sup>.

 **안전하게** 이미지 빌드를 하기 위해서는 podman build 보다는 buildah bud를 통해서 빌드를 매우 권장한다.

```
# podman build --f Dockerfile  
# buildah bud -f Containerfile
```

## 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. 설치는 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.

```
# 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.

<sup>16</sup> centos-8-stream은 module로 설치해야 한다. 가급적이면 container-tools:3.0으로 설치한다.

<sup>17</sup> 일본 친구가 빌더라고 발음을 알려주었다. 그리고 한동안 사용하였다.

<sup>18</sup> 좀 더 정확히는 한 번 더 래핑(wrapping)하여 제공한다.

```
# 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/
```

이 명령어는 앞에서 폐쇄망에서 사용할 레지스트리에 이미지 복제 시 사용하였다. 이미지 정보 확인 시 가급적이면 podman image inspect 보다는 skopeo inspect 를 통해서 확인을 권장한다.

## 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 Tekton으로 많이 사용하고 있다. 이 책에서는 테크톤을 사용하여 어떻게 CI/CD를 구성 및 활용하는지 간단하게 다루어 본다.

## 5. 저 수준 런타임

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

1. docker-shim
2. CRI-Docker(OCI)
3. CRI-O(OCI)
4. Contained (OCI)
5. LXC/LXD
6. rkt

런타임은 사용자가 원하는 환경 혹은 프로그램 선택이 가능하며, 쿠버네티스는 containerd, cri-docker, cri-o 지원한다. 쿠버네티스에서 사용하는 런타임 환경은 containerd 및 CRI-O 환경이다. 앞으로 Docker 는 런타임으로써 더 이상 지원하지 않는다고 발표하였다<sup>19</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 14 docker-shim 지원 중지

현재 오픈소스 컨테이너는 **OCI(Open Source Container Initiative)**라는 표준 사양을 정의하였으며, 이 사양을 따르는 대표적인 저수준 컨테이너 런타임은 containerd, cri-docker, cri-o 가 대표적이다.

<sup>19</sup> 하지만 shim 구조가 빠진 도커는 여전히 지원하고 있다.

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

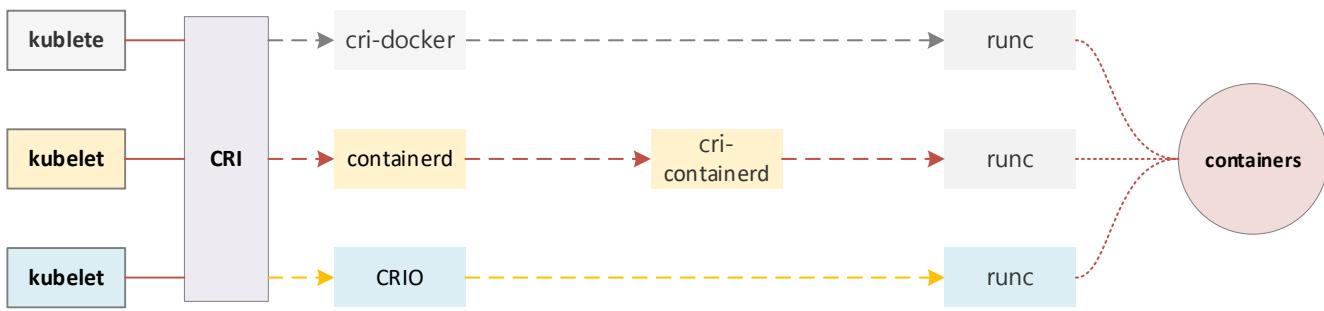


그림 15 컨테이너 CRI

구버전의 도커는 모든 자원들 즉, namespace, cgroup 를 직접 관리하였다. 이러한 이유로 docker 데몬이 비정상적으로 종료가 되면 모든 컨테이너가 종료가 되었다. Containerd, cri-docker 으로 전환이 되면서, OCI 사양을 따르면서 현재는 직접 관리하지 않는다.<sup>20</sup>

kubelet은 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.

<sup>20</sup> docker-shim 참조.

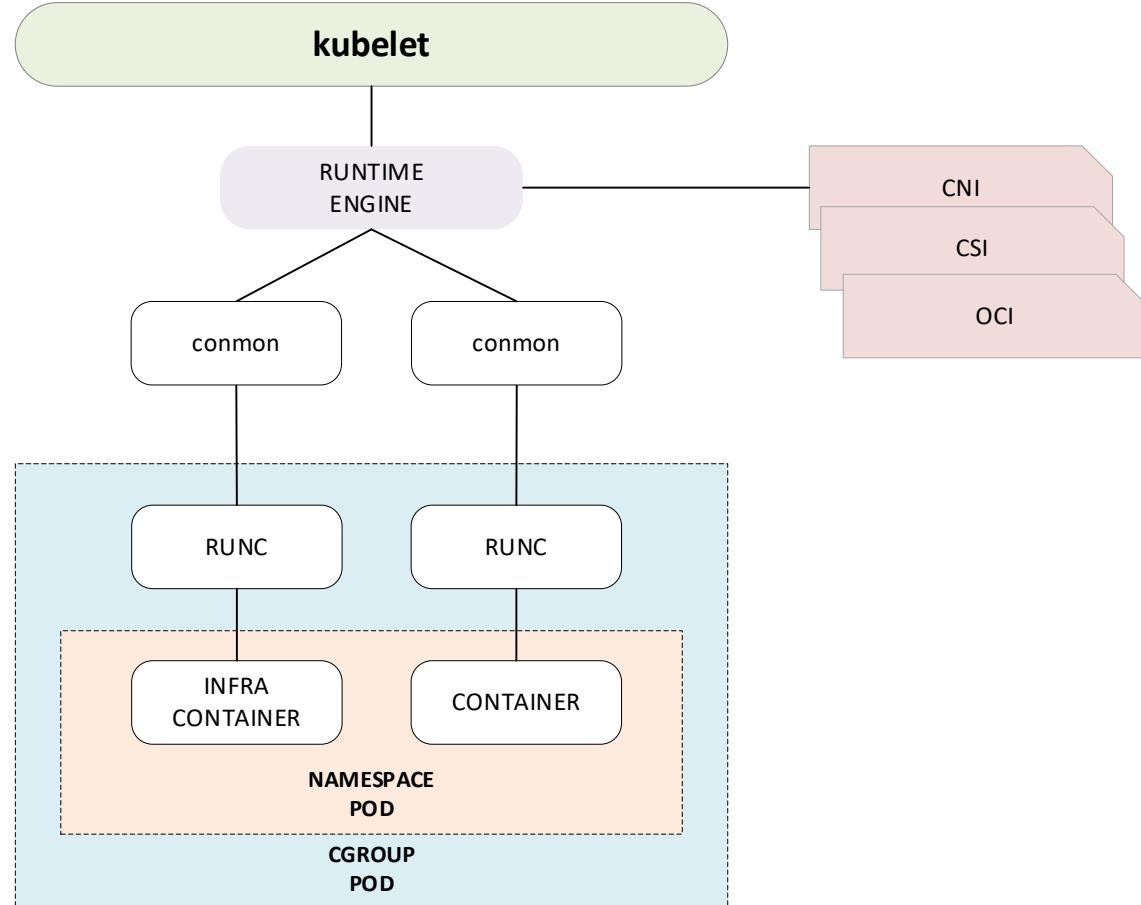


그림 14 CRI-O 구조

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

쿠버네티스 클러스터를 구성하기 위해서는 리눅스 커널에서 다음과 같은 기능을 요구한다.

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

추가적으로 요구하는 기술 및 프로그램은 다음과 같다.

- SELinux
- AppArmor

SELinux<sup>21</sup>는 seccomp<sup>22</sup>와 비슷하게 필터링 하지만, 더 세밀하게 시스템 콜 및 네트워크 접근에 대한 시스템 보안을 원하는 경우 SELinux 사용이 가능하다. 하지만, "SELinux Policy"<sup>23</sup>에 대한 지식이 없으면, 일반적으로 끄고 사용하는 걸 권장한다.

## Containerd

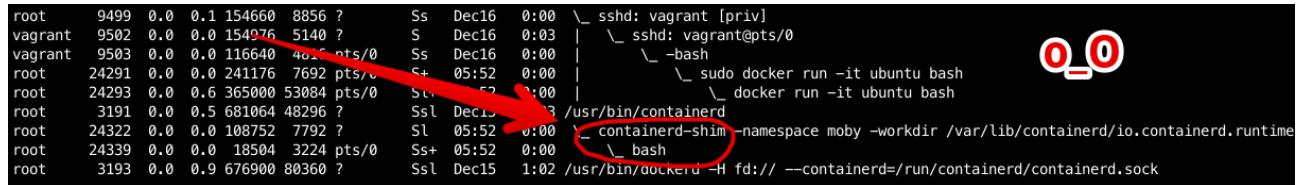
Containerd 는 CNCF 에서 시작하였으며 이 프로젝트는 윈도우 및 리눅스 양쪽에서 실행이 가능하다. 주요 목적은 컨테이너의 라이프 사이클 관리가 주요 목적이며 컨테이너에 저장소 및 네트워크 기능을 제공한다. Containerd 는 도커의 하위 서비스인 dockerd 밑에서 containerd 라는 이름으로 동작하였지만, 지금은 도커에서 분리하여 독립적인 컨테이너 런타임으로 사용이 가능하다.

<sup>21</sup> [https://selinuxproject.org/page/Main\\_Page](https://selinuxproject.org/page/Main_Page)

<sup>22</sup> <https://security.stackexchange.com/questions/264273/what-does-default-seccomp-apparmor-and-selinux-in-kubernetes-security-mean>

<sup>23</sup> [https://selinuxproject.org/page/Building\\_a\\_Basic\\_Policy](https://selinuxproject.org/page/Building_a_Basic_Policy)

containerd 는 CRI-O 처럼 runc-shim 기반으로 동작하며 OCI, CNI 와 같은 표준 사양을 준수한다. containerd 는 쿠버네티스에서 지원하는 런타임 중 하나이다. cri-docker 와 containerd 는 비슷하지만 구조가 다르다. cri-docker 경우에는 docker-shim 를 지원하며, containerd 는 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 154976 5140 ? S Dec16 0:03 | \_ sshd: vagrant@pts/0
vagrant 9503 0.0 0.0 116640 400 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 Dec15 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.v1.linux
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 16 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.

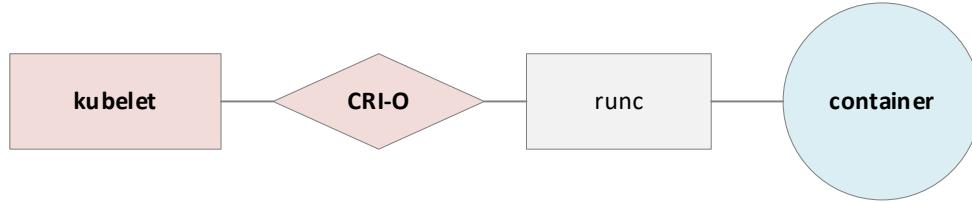


그림 17 CRI-O

대다수 엔터프라이즈 컨테이너 오케스트레이션 엔진들은 CRI-O 런타임을 사용한다. CRI-O 의 약자는 Container Runtime Interface/Open Container Initiative 의 앞 글자만 따와서 CRI-O 라고 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. 아래는 간단하게 루프백 인터페이스 혹은 소켓으로 API 서버를 시작한다.

```

podman system service -t 5000 &
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-O 와 동작 구조는 흡사하다.

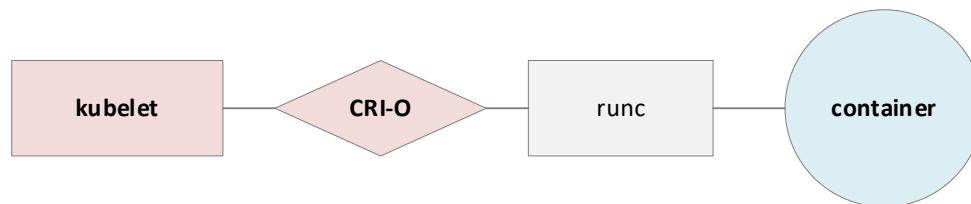


그림 18 podman 구조

다만, 포드만은 쿠버네티스의 저수준 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 POD 를 구성 및 구현하기 위해서는 커널에서 자원 격리 및 추적이 가능한 기능을 제공해야 한다. 대표적인 기능은 namespace 및 cgroup 이 있다.

### 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. 네임스페이스는 트리 형태 구조를 가지고 있으며, 커널 버전에 따라서 차이가 있지만, 프로세스가 사용하는 kernel context 를 용도에 맞게 분리 및 격리한다.

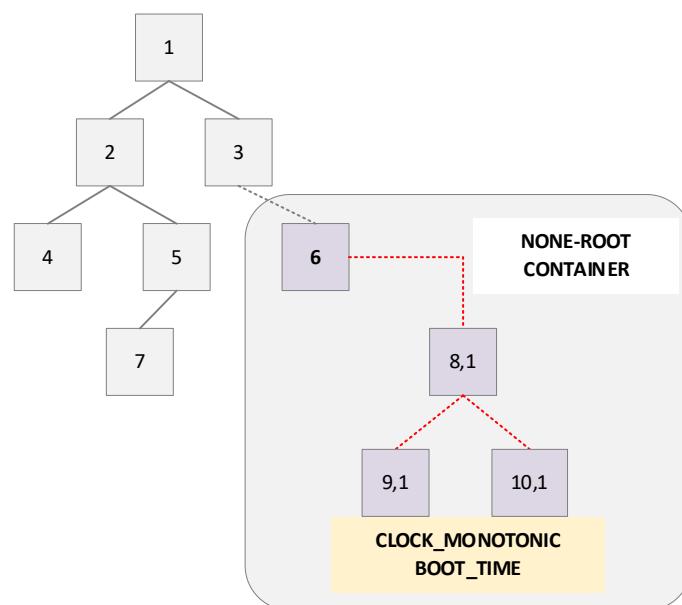


그림 19 namespace UTS clock

네임스페이스는 리눅스 커널 2002년에 도입이 되었으며, 본격적으로 도입된 릴리즈 버전은 linux-2.4.19 이다. 현 리눅스 커널의 네임스페이스는 처음에 도입이 되었던 네임스페이스보다 더 효율적인 구조 및 기능이 확장이 되었다. 이전 네임스페이스는 다음과 같은 단점이 있었다.

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

위와 같은 이유로 초기에는 컨테이너 컴퓨터 노드에서 많은 개수의 컨테이너를 동시 생성을 하면, 커널에서 많은 무리가 발생하였지만, 각각 네임스페이스별로 시간을 제어 및 조정이 가능하기 때문에 빠르게 구성이 가능하다. 하지만, 독립적인 TIME 및 USER<sup>24</sup>기능은 커널 3.x 버전부터 지원하게 되었다.

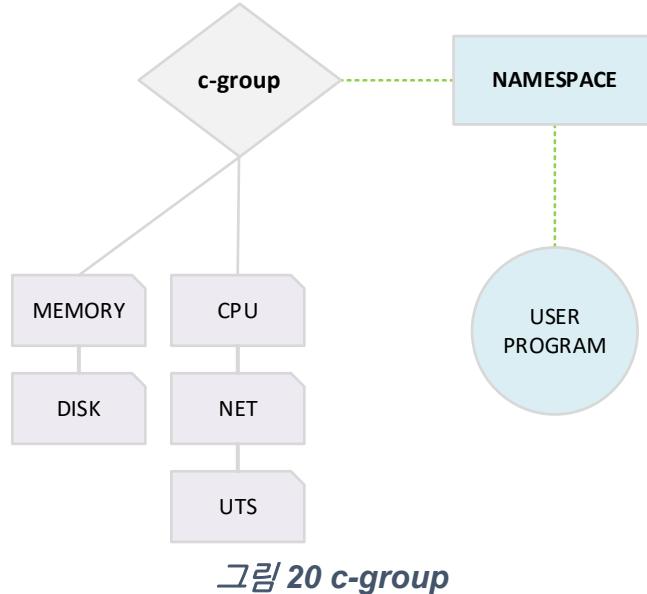
### C-GROUP

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

<sup>24</sup> [https://man7.org/linux/man-pages/man7/user\\_namespaces.7.html](https://man7.org/linux/man-pages/man7/user_namespaces.7.html)

1. 자원제한(resource limiting)
2. 우선순위(Prioritization)
3. 할당(Accounting)
4. 제어(Control)

C-GROUP은 프로세스 격리 기능은 제공하지 않으며, 추적 및 감사 기능만 제공하기 때문에, 네임스페이스를 통해서 구성 및 관리한다.



"c-group"을 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 cgroup과 네임스페이스를 통해서 동작이 된다.

## 7. 파일시스템

### UFS

**UFS<sup>25</sup> (Union File System), OverlayFS(File System)**은 도커에서 사용하였던 이미지 파일 시스템 및 레이어이다. 이 파일 시스템은 실제로 존재하지 않지만, 파일이나 디렉터리를 마치 레이어처럼 만들어서 구성해준다. 유니온 파일 시스템은 리눅스에서 많이 사용하는 **xfs**, **btrfs**, **ext4** 그리고 **devicemapper**에서 사용이 가능하다.

기본적으로 컨테이너 이미지는 UFS는 형태로 저장이 되며, 이 구조는 **key=pair** 형태로 구성이 되어 있다.

---

<sup>25</sup> <https://unionfs.filesystems.org/docs/zen/zen.html>

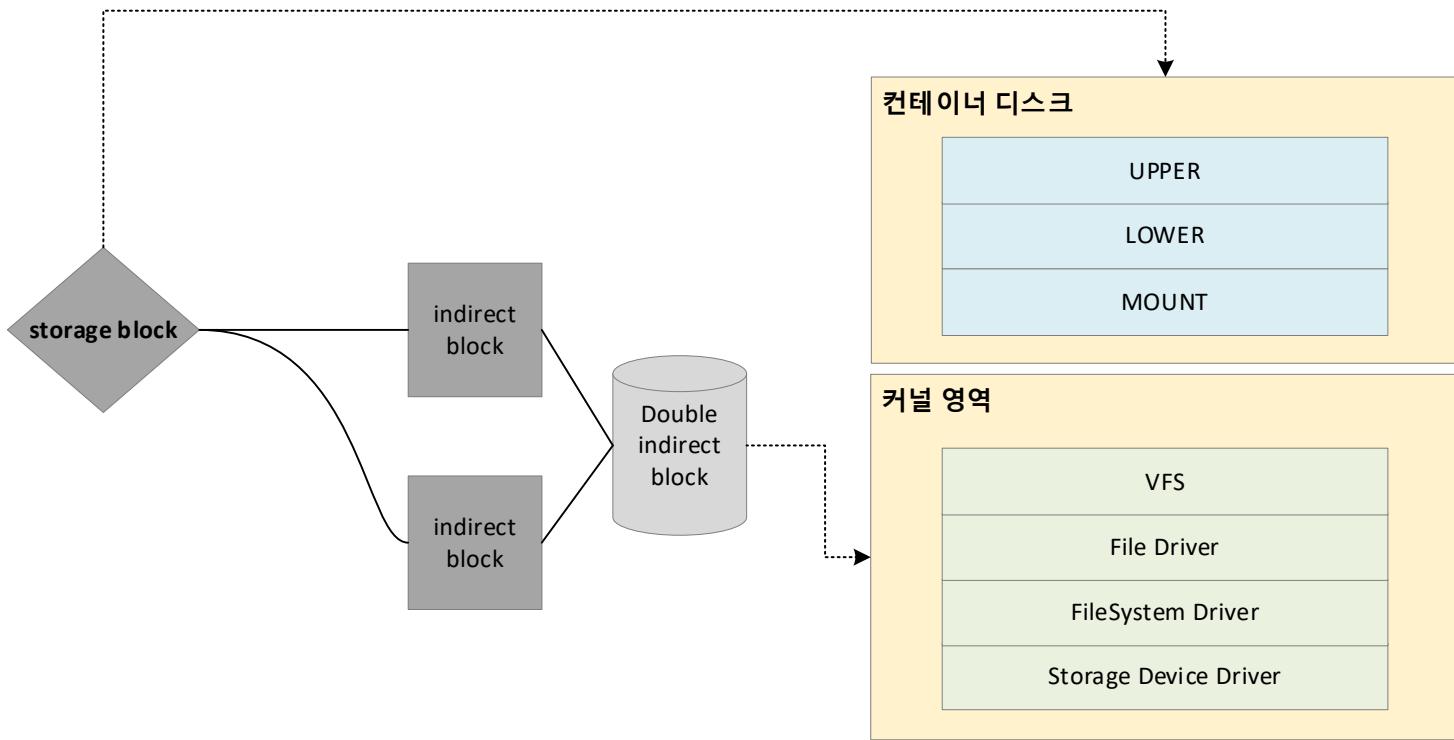


그림 21 UFS 레이어

## OverlayFS (Filesystem)

**OverlayFS<sup>26</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>27</sup>를 통해서 단일 장치처럼 사용자에게 구성 및 생성이 되어 전달이 된다.

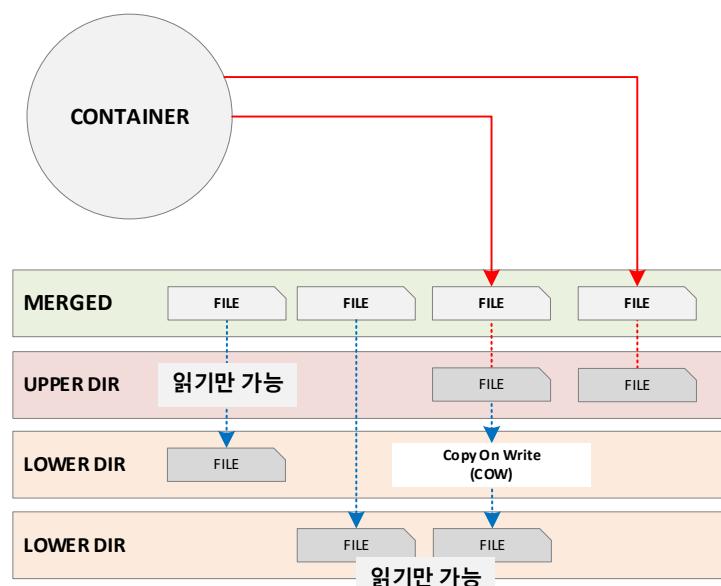


그림 22 OverlayFS 계층

<sup>26</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>27</sup> <https://github.com/BCDevOps/backup-container>

거의 대다수 컨테이너 시스템은 위와 같은 구조로 컨테이너 이미지를 구현하고 있다. UFS 경우에는 논리적 구성을 하드코딩 된 드라이버를 통해서 **유저/커널** 영역에서 장치를 구현한다. 이를 기반으로 완벽하게 컨테이너 드라이버를 구현한다는 장점이 있지만, 단점으로는 커널 영역을 사용하기 때문에 성능 부분에서는 OverlayFS 보다 느리다. 대다수 컨테이너의 파일 시스템은 OverlayFS 기반으로 사용하고 있고, "Fuse"기반으로 구성된 "Fuse OverlayFS"<sup>28</sup>를 더 많이 사용한다.

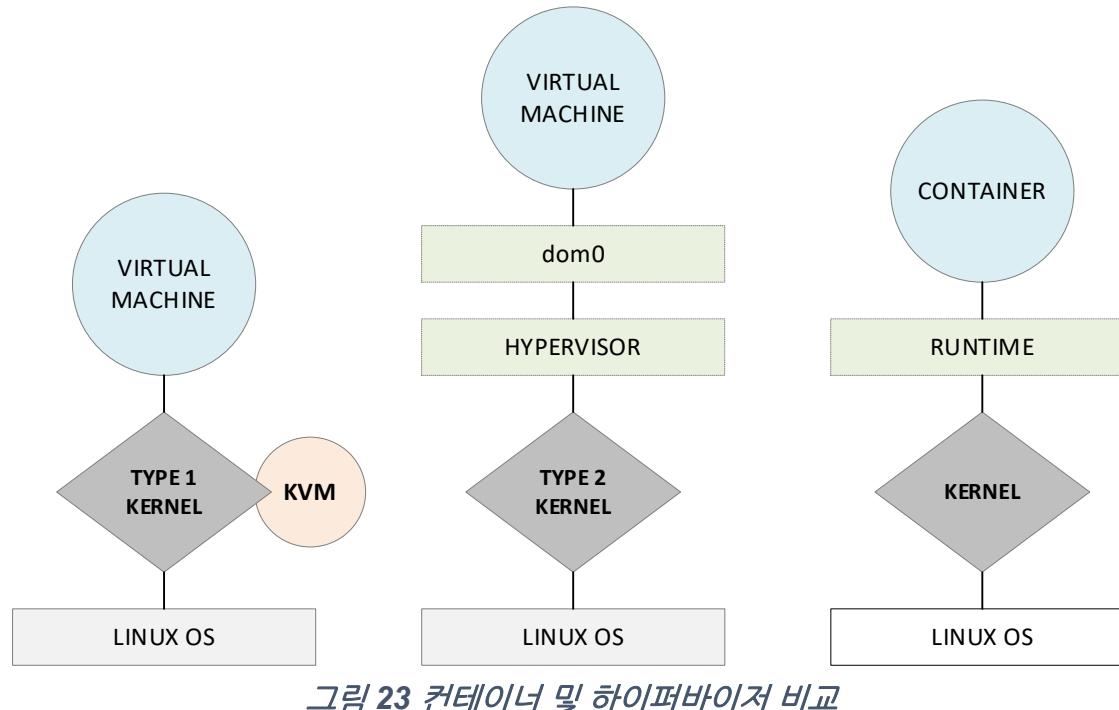
---

<sup>28</sup> <https://github.com/containers/fuse-overlayfs>

# 쿠버네티스 아키텍처

## 1. 가상화 vs 컨테이너

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



가상머신과 컨테이너의 제일 큰 차이점은 **하드웨어의 가상화 기능사용**<sup>29</sup> 여부이다. 기본적으로 쿠버네티스는 컨테이너 런타임 기반의 환경을 제공하기 때문에 하드웨어 기반의 가상화처럼 높은 하드웨어를 요구하지 않는다. 하지만, 컨테이너는 rootless 구조 및 ring structure를 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 systemd에서 cgroup 및 namespace를 관리 및 추적을 한다.

<sup>29</sup> 요즘 컨테이너는 SR-IOV 와 같은 기능 사용이 가능하다.

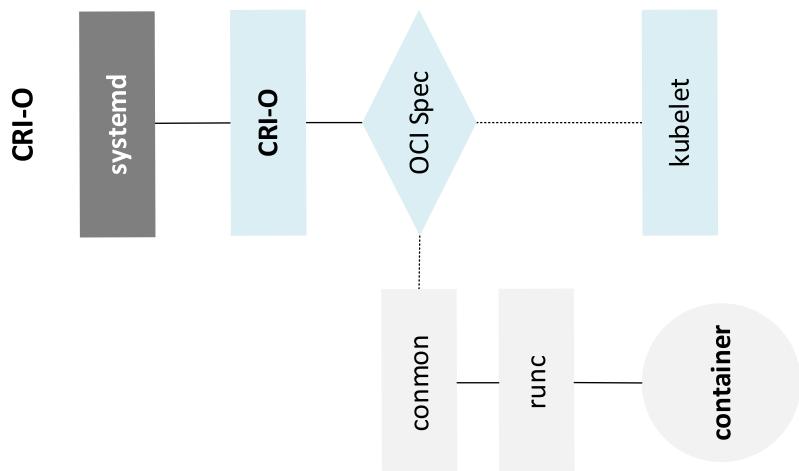


그림 24 RUNC(common)

생성된 컨테이너는 common<sup>30</sup>이라는 프로세서가 컨테이너를 모니터링을 한다. 컨테이너는 common에서 "-r"이라는 런타임 옵션을 통해서 'runc'기반으로 컨테이너 이미지를 불러와서 실행한다.

### 3. 쿠버네티스 클러스터 컴포넌트 설명

초기 쿠버네티스는 일반적인 서비스처럼 호스트 컴퓨터에 설치가 되었지만, 현재 쿠버네티스는 kubelet이라는 서비스를 제외하고 나머지는 전부 컨테이너 기반으로 동작한다. 이를 정적 포드(Static POD)라고 부른다.

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

정적 포드는 아래와 같은 소프트웨어 기반으로 쿠버네티스의 도메인 서비스, 포드 및 상태 저장을 구현한다. 해당 소프트웨어 중 하나라도 올바르게 동작하지 않으면, 클러스터에서 생성된 자원 및 상태 정보를 확인할 수 없다.

- pod(pause)
- coredns
- etcd

위의 구성원들을 통해서 아래와 같이 쿠버네티스 자원을 생성한다. 쿠버네티스 관리자 혹은 개발자는 모든 리소스 정보를 아래와 같은 구조로 검색 및 조회를 한다.

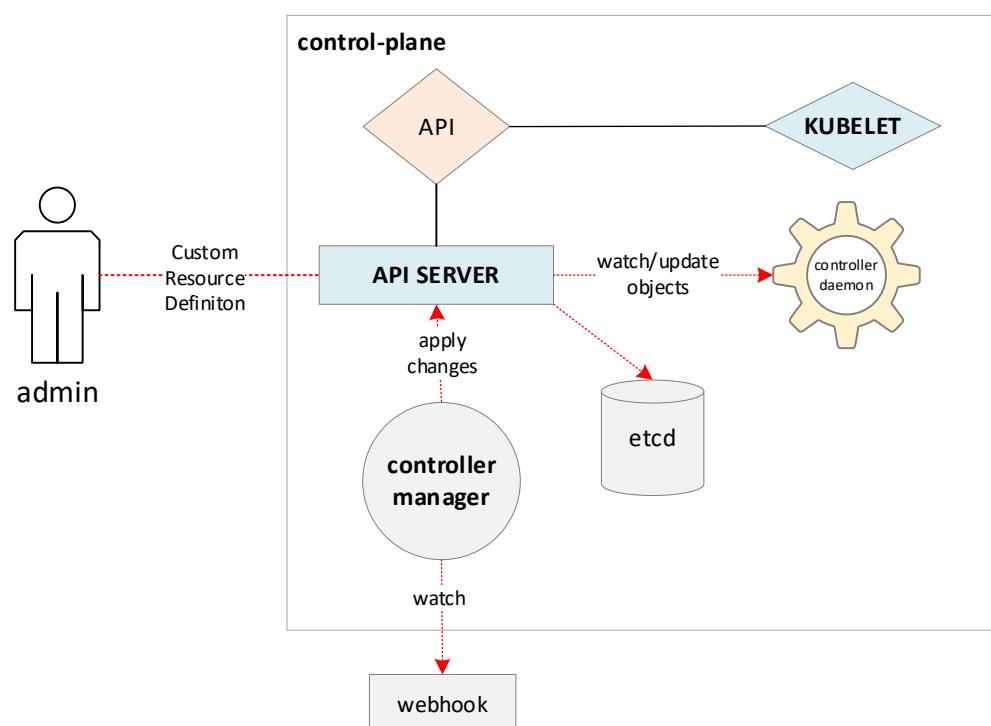


그림 25 쿠버네티스 구성원

<sup>30</sup> container monitor 의 약자

위의 정보 기반으로 각 쿠버네티스 구성원에 대해서 확인하도록 한다.

## 컨트롤 플레인(control-plane, master, controller)

컨트롤 플레인은 다른 호칭으로 마스터 서버<sup>31</sup>라고 부르기도 한다. 현재는 컨트롤 플레인으로 많이 호칭을 하고 있다. 쿠버네티스 클러스터는 최소 한 개의 컨트롤 플레인이 클러스터에 필요하다. 컨트롤 플레인은 여러 서비스가 하나로 묶여서 구성이 된다. 구성원 서비스는 위에서 설명하였기 때문에, 별도로 설명하지 않는다.

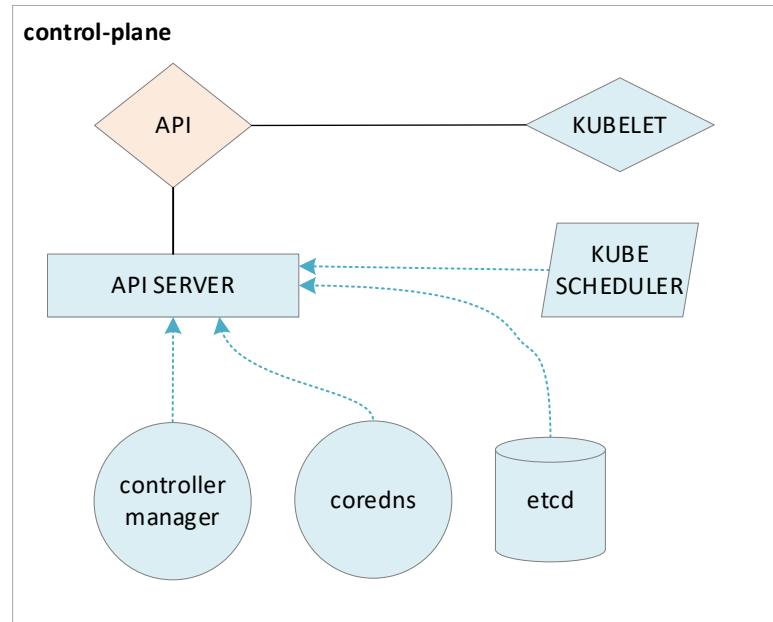


그림 26 컨트롤 플레인

## 컴퓨트 노드(minion, compute)

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

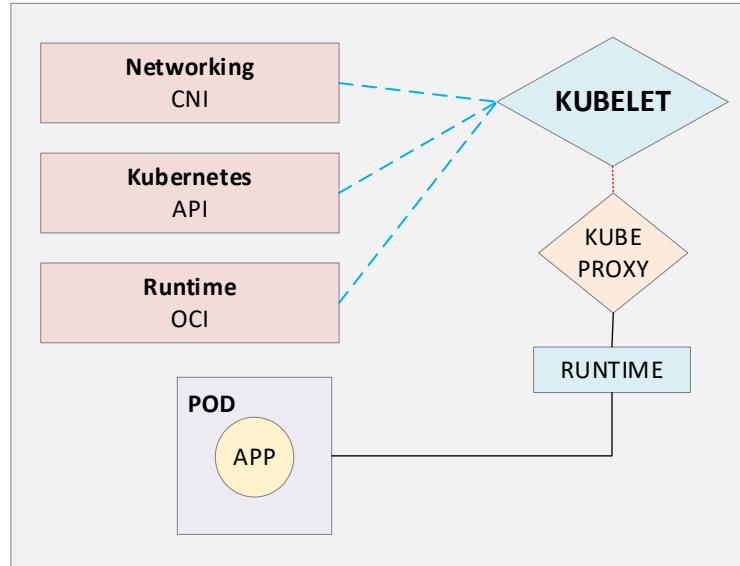


그림 27 워커노드

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 컴퓨터 노드에서 실행이 되며, 노드는 최소 한개의 컨트롤 노드를 통해서 클러스터를 구성한다.

## 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

<sup>31</sup> 필자는 더 이상 마스터 서버라고 부르지 않는다. 쿠버네티스 서밋에 같이 참여한 흑인 동료가 매우 불편하였다.

2. 외부에서 들어오는 요청에 대한 서비스 처리
3. 컨테이너 생성 및 관리

`kubelet`를 사용하기 위한 서비스 포트 번호는 아래와 같다. 특히 마스터 노드에서는 아래 포트가 활성화가 되어 있어야 한다. 방화벽이 동작중이면, 아래 포트번호를 꼭 등록한다.

- TCP/6443
- HTTP/HTTPS

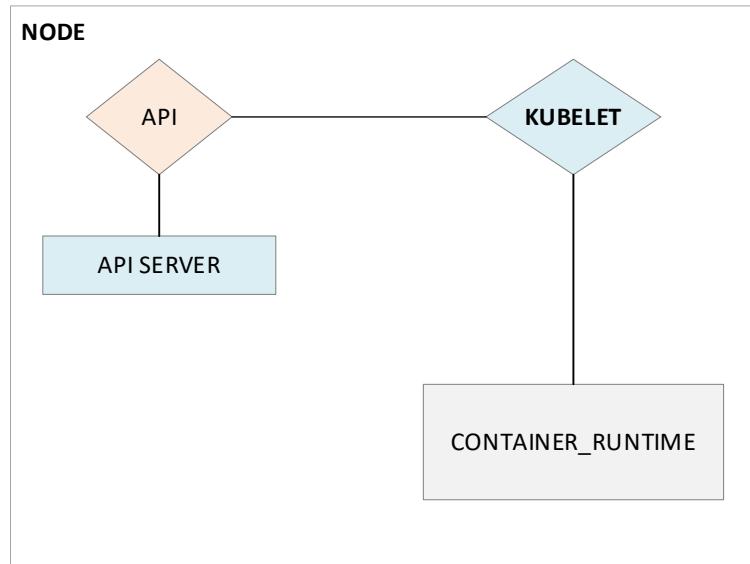


그림 28 `kubelet` 구조

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.

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

`kubelet`는 자원의 **생성/삭제/조회**와 같은 기능을 제공하며, control, compute 상관 없이, 해당 데몬은 항상 실행이 되고 있다. 이 데몬은 쿠버네티스 주요 서비스에 문제가 있는 경우, 재시작 혹은 재생성을 시도하며, 이 서비스 자체가 멈추는 경우, 작업 요청에 대해서 처리가 불가능하다.

이름	호스트	컨테이너
<code>kubelet</code>	네	아니요
<code>kubeproxy</code>	아니오	네

## etcd

`etcd`는 쿠버네티스에서 발생하는 데이터를 저장한다. `etcd`의 주요 목적은 **빠르게 읽기/쓰기**를 위해서 만들어진 도구<sup>32</sup>이며, 일반적인 RDBMS처럼 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.

<sup>32</sup> [https://etcd.io/docs/v3.3/learning/data\\_model/](https://etcd.io/docs/v3.3/learning/data_model/)

- 생성/제거:** etcd에 발생되는 설정을 생성 및 제거를 합니다. 멤버 추가 및 설정 부분은 클러스터의 크기만 사용자가 명시해주면 됩니다.
- 백업:** etcd 오퍼레이터는 백업을 자동으로 수행합니다. 30분마다 백업하고 마지막 3번의 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.

etcd는 단일 혹은 다중으로 동작한다. 다중으로 구성하기 위해서는 최소 3개의 etcd 서비스가 구성이 되어야 한다. etcd는 특정 노드를 etcd 맴버에 추가하면 자동적으로 클러스터 구성한다.

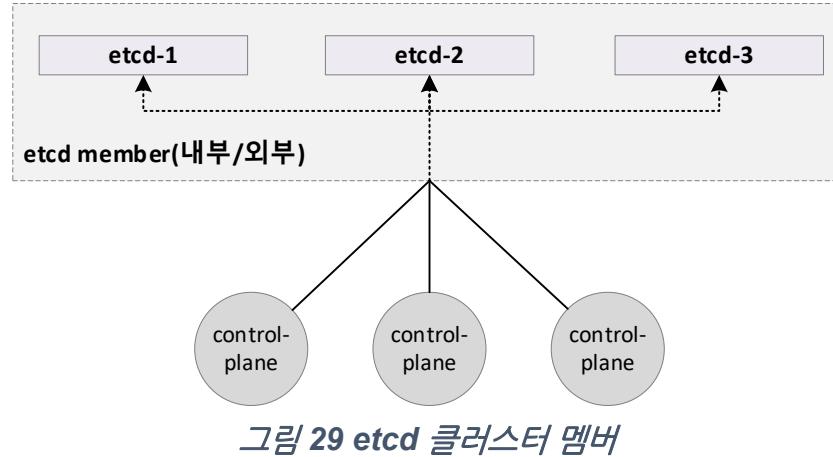


그림 29 etcd 클러스터 멤버

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

```
# export ADVERTISE_URL=https://192.168.90.110:2379
# 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

# dnf install jq -y
# 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 딕렉터리 서비스 목록이 출력이 된다.

```
/registry/minions/node1.example.com
/registry/minions/node2.example.com
/registry/minions/node3.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>33</sup> 웹 사이트에서 추가적인 정보 확인이 가능하다.

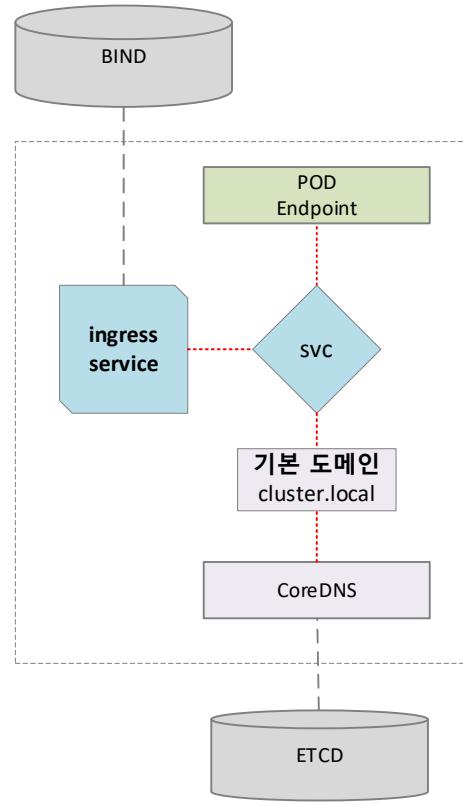


그림 30 coredns

coredns 는 기본적으로 내부 클러스터 서비스를 위해서 사용하며, 여러 서비스를 하나로 묶기 위한 방법으로 사용한다. 설치시 별도로 도메인을 설정하지 않았으면, cluster.local<sup>34</sup>로 되어있다.

나중에 확인이 가능하지만, coredns 도 일반 도메인 서비스 프로그램과 동일하게 존 파일(zone file)기반으로 도메인을 구성 및 설정한다. 형식은 아래와 같이 쿠버네티스에서 내부적으로 작성한다.

```
$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
```

busybox 를 통해서 컨테이너 실행 후, dig, nslookup 와 같은 명령어를 사용해서 확인이 가능하다.

```
# kubectl run -i -t --image=busybox  
> cat /etc/resolv.conf  
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
```

<sup>33</sup> <https://coredns.io/>

<sup>34</sup> 서비스 도메인은 설치 시 변경이 가능하다.

Coredns에서 사용하는 설정 내용은 아래 ConfigMap에서 확인이 가능하다.

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

## kube-proxy

kube-proxy<sup>35</sup>서비스는 쿠버네티스에 구성이 되어있는 애플리케이션 서비스에 접근할 수 있도록 지원해주는 구성원. 주요 목적은 쿠버네티스에 구성된 자원들의 네트워크 생성 및 관리 모니터링을 담당한다. 이 자원은 컨트롤러 및 컴퓨터 노드에 설치 및 구성이 된다.

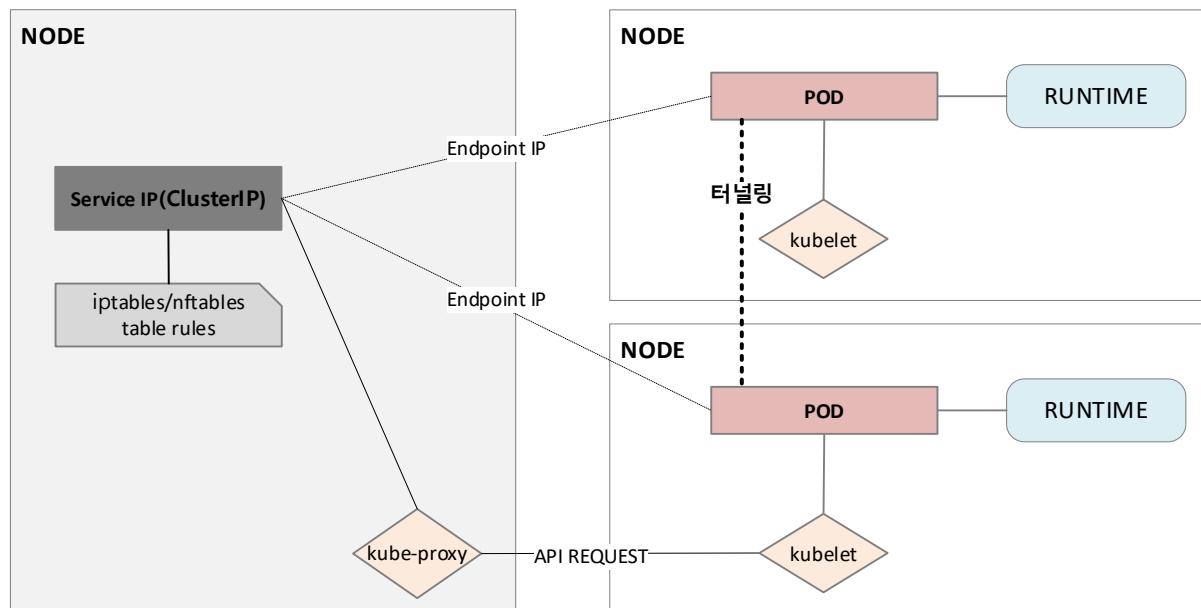


그림 31 kube-proxy

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

```
# 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
```

```
# crictl ps
# kube-proxy
```

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

```
# 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
    }
    chain KUBE-PROXY-FIREWALL {
        type filter hook forward priority filter; policy accept;
        counter packets 48133 bytes 83155892
    }
    chain KUBE-NODEPORTS {
        type filter hook forward priority filter; policy accept;
        counter packets 48133 bytes 83155892
    }
}
kubernetes cluster.local in-addr.arpa ip6.arpa {
```

<sup>35</sup> <https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/>

```

    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. 작동 방법은 소스코드에 다음과 같이 작성되어 있다.

```

// WithPercentageOfNodesToScore sets percentageOfNodesToScore for Scheduler.
// The default value of 0 will use an adaptive percentage: 50 - (num of nodes)/125.
func WithPercentageOfNodesToScore(percentageOfNodesToScore *int32) Option {
    return func(o *schedulerOptions) {
        if percentageOfNodesToScore != nil {
            o.percentageOfNodesToScore = *percentageOfNodesToScore
        }
    }
}

func selectHost(nodeScoreList []framework.NodePluginScores, count int) (string,
[]framework.NodePluginScores, error) {
    if len(nodeScoreList) == 0 {
        return "", nil, errEmptyPriorityList
    }

    var h nodeScoreHeap = nodeScoreList
    heap.Init(&h)
    cntOfMaxScore := 1
    selectedIndex := 0
    // The top of the heap is the NodeScoreResult with the highest score.
    sortedNodeScoreList := make([]framework.NodePluginScores, 0, count)
    sortedNodeScoreList = append(sortedNodeScoreList,
        heap.Pop(&h).(framework.NodePluginScores))

    // This for-loop will continue until all Nodes with the highest scores get checked
    // for a reservoir sampling,

```

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.

<b>스케줄링 정책</b>	필터링을 통해서 각 노드별로 점수를 매기며, 이를 통해서 어떤 노드에 자원을 구성할지 결정한다.
<b>스케줄링 프로파일(scheduling profile)</b>	프로파일을 통해서 좀 더 추가적인 기능을 구성할 수 있는데, 스케줄러 플러그인을 통해서 kube-scheduler에 다양한 프로파일을 구성할 수 있다. 이 프로파일은 Queue Sort, Filter, Score, Bind, Reserve, Permit 플러그인 혹은 사용자가 추가적으로 구성이 가능하다.

외부에서 작업 요청이 오면 API-SERVER -> CONTROLLER-MANAGER -> SCHEDULER를 통해서 컨테이너 생성을 요청한다.

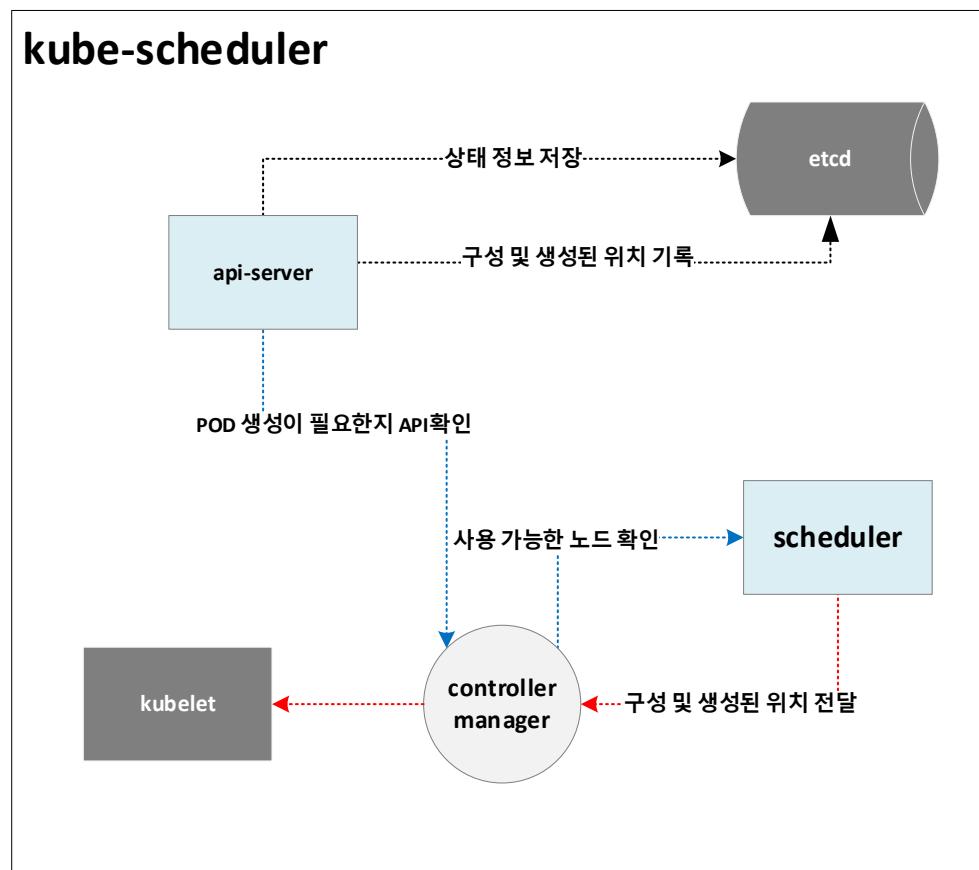


그림 32 스케줄러

## controller-manager

컨트롤러 매니저는 스케줄러와 비슷하게 보이지만(?), 다른 역할을 수행한다. 이 구성원은 쿠버네티스에 구성이 되어 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, apiserver를 통해서 상태 확인 혹은 변경이다. kube-controller-manager를 대표적으로 많이 사용하는 서비스는 다음과 같다.

- REPLICATION CONTROLLER
- ENDPOINT CONTROLLER
- NAMESPACE CONTROLLER
- SERVICEACCOUNT CONTROLLER

컨트롤러 매니저는 쿠버네티스에서 자원 생성 요청이 들어오면, 적절하게 각 자원들을 올바르 위치에 생성 및 구성을 한다. 모든 자원들은 앞서 이야기하였지만, API를 통해서 처리를 한다.

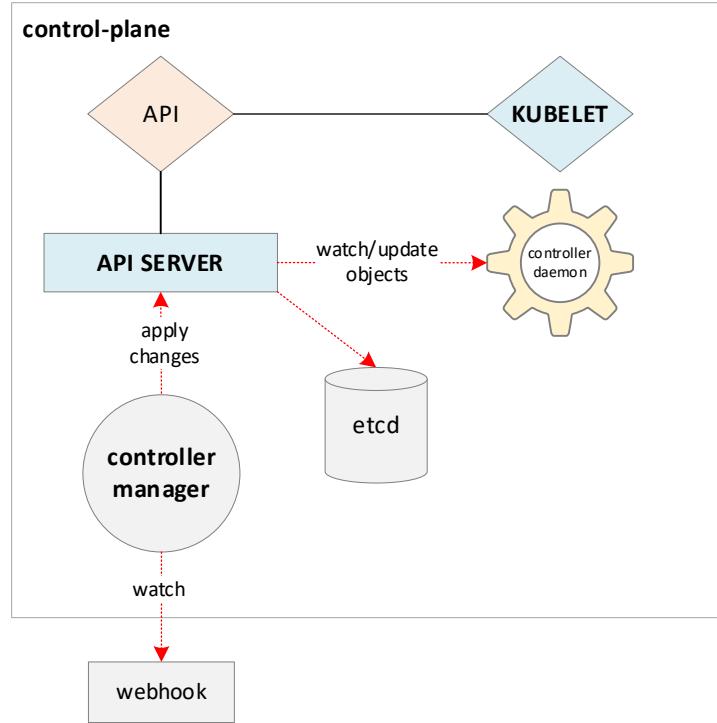


그림 33 컨트롤러 매니저

API를 통해서 들어온 작업은 컨트롤러 매니저로 전달이 되어서, 각 요청에 따라서 작업을 수행한다. 컨트롤러 매니저는 대표적으로 POD/JOB/DEPLOYMENT/NAMESPACE를 생성 및 구성한다.

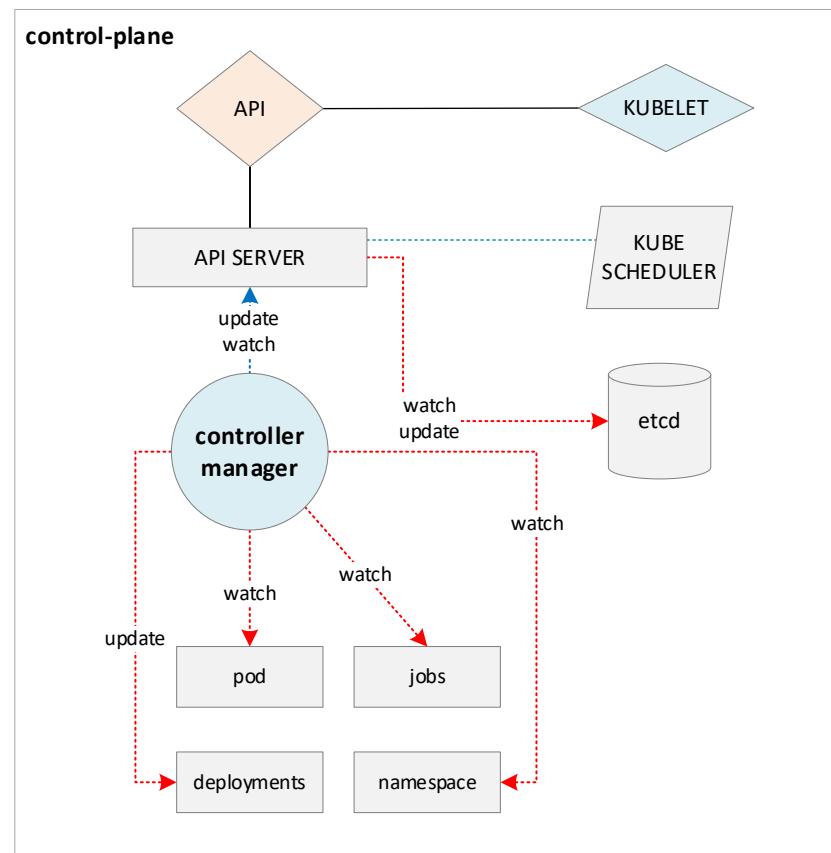


그림 34 컨트롤러를 통한 자원 생성

## kube-apiserver

외부에서 들어오는 요청은 API 서버를 통해서 자원 처리를 한다. kubectl이나 혹은 API를 통해서 들어온 자원들은 kubelet을 통해서 노드로 전달이 된다.

각각 자원은 컨트롤러 매니저를 통해서 Pod 같은 자원을 처리하며, 수행이 된 작업은 etcd 서버에 상태 정보를 저장한다. 쿠버네티스에서 사용하는 API SERVER는 정말로 일반적인 API 서버이다. 정말로 API만 다룬다. 아래는 소스코드의 일부분이다.

```
// GenericStorageProviders returns a set of APIs for a generic control plane.
// They ought to be a subset of those served by kube-apiserver.
```

```

func (c *CompletedConfig) GenericStorageProviders(discovery
discovery.DiscoveryInterface) ([]RESTStorageProvider, error) {
    // The order here is preserved in discovery.
    // If resources with identical names exist in more than one of these groups (e.g.
    "deployments.apps"" and "deployments.extensions"),
    // the order of this list determines which group an unqualified resource name
    (e.g. "deployments") should prefer.
    // This priority order is used for local discovery, but it ends up aggregated in
    'k8s.io/kubernetes/cmd/kube-apiserver/app/aggregator.go
    // with specific priorities.
    // TODO: describe the priority all the way down in the RESTStorageProviders and
    plumb it back through the various discovery
    // handlers that we have.
    return []RESTStorageProvider{
        c.NewCoreGenericConfig(),
        apiserverinternalrest.StorageProvider{},
        authenticationrest.RESTStorageProvider{Authenticator:
c.Generic.Authentication.Authenticator, APIAudiences:
c.Generic.Authentication.APIAudiences},
        authorizationrest.RESTStorageProvider{Authorizer:
c.Generic.Authorization.Authorizer, RuleResolver: c.Generic.RuleResolver},
        certificatesrest.RESTStorageProvider{},
        coordinationrest.RESTStorageProvider{},
        rbacrest.RESTStorageProvider{Authorizer: c.Generic.Authorization.Authorizer},
        svmrest.RESTStorageProvider{},
        flowcontrolrest.RESTStorageProvider{InformerFactory:
c.Generic.SharedInformerFactory},
        admissionregistrationrest.RESTStorageProvider{Authorizer:
c.Generic.Authorization.Authorizer, DiscoveryClient: discovery},
        eventsrest.RESTStorageProvider{TTL: c.EventTTL},
    }, nil
}

```

RestAPI 의 스토리지에서 node/v1 를 호출하면 다음과 같이 apimachinery 를 통해서 조회한다.

```

package v1
import (
    nodev1 "k8s.io/api/node/v1"
    "k8s.io/apimachinery/pkg/runtime/schema"
)
// GroupName for node API
const GroupName = "node.k8s.io"

```

API 는 각 서비스 혹은 자원에 대한 API 목록을 가지고 있으며, 올바른 요청이 들어오면 해당 요청을 해당 API 를 통해서 작업을 수행한다. 예를 들어서 클러스터에서 사용하는 노드 목록 정보를 호출하는 경우, 아래와 같이 동작한다.

```

// GetNode returns the node info for the configured node name of this Kubelet.
func (kl *Kubelet) GetNode() (*v1.Node, error) {
    if kl.kubeClient == nil {
        return kl.initialNode(context.TODO())
    }
    return kl.nodeLister.Get(string(kl.nodeName))
}

```

위의 내용을 그림으로 표현하면 다음과 같이 동작한다.

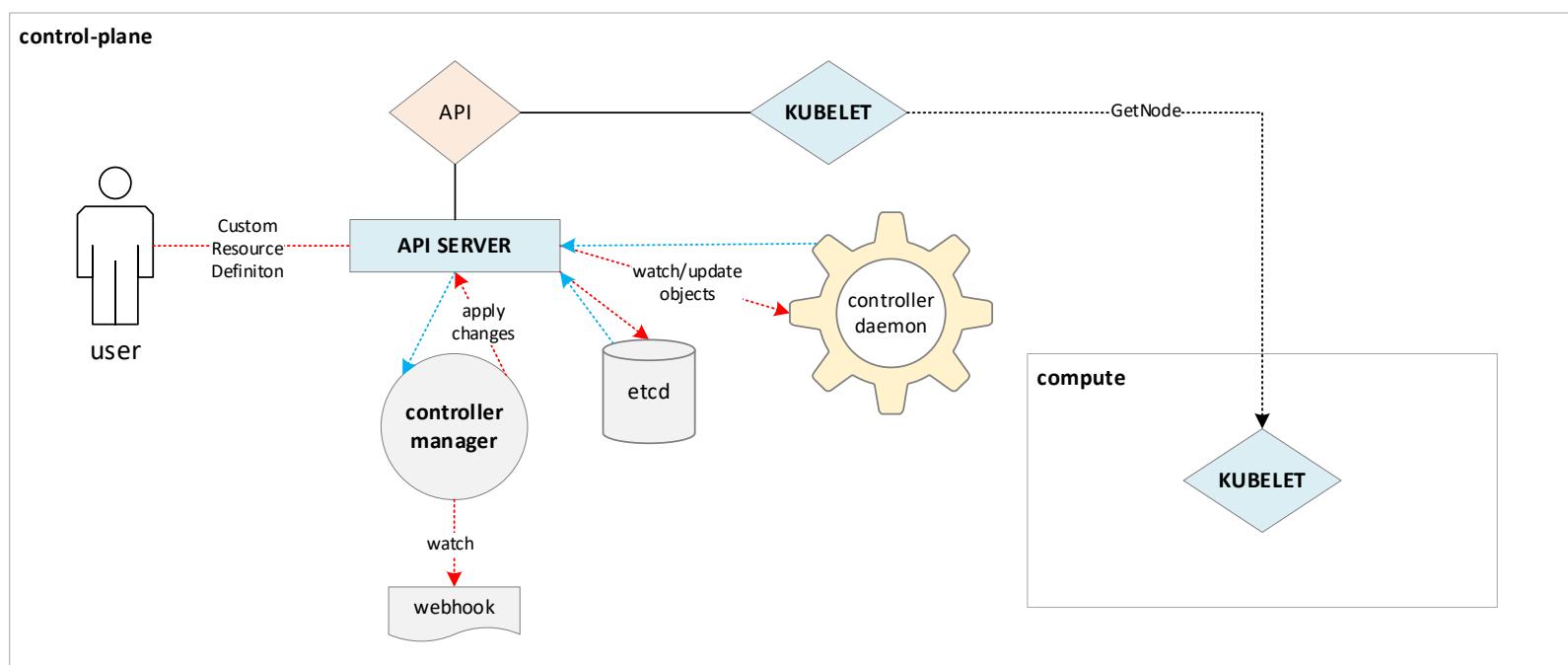


그림 35 apiserver 와 컨트롤러 서버 관계

## 스왑(swap)

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. POD 에 메모리 제한이 있는 경우, 페이지로 인한 CPU 사용율 증가<sup>36</sup>로 실제 컨테이너 운영에 도움이 되지 않는다.

이러한 문제가 발생하는 부분은 바로, 쿠버네티스의 **QoS(request/limit)** 기능에서 문제가 발생한다. 일반적으로 프로세스는 호스트 컴퓨터의 자원을 활용할 수 있는 만큼, 활용한다. 하지만, QoS 가 설정이 되면, cgroup 에서 CPU 및 MEMORY 에 대해서 제한을 걸기 시작하며, 프로세스 입장에서는 더 이상 MEMORY 자원 사용이 어렵기 때문에 커널에 페이지 요청을 시작한다.

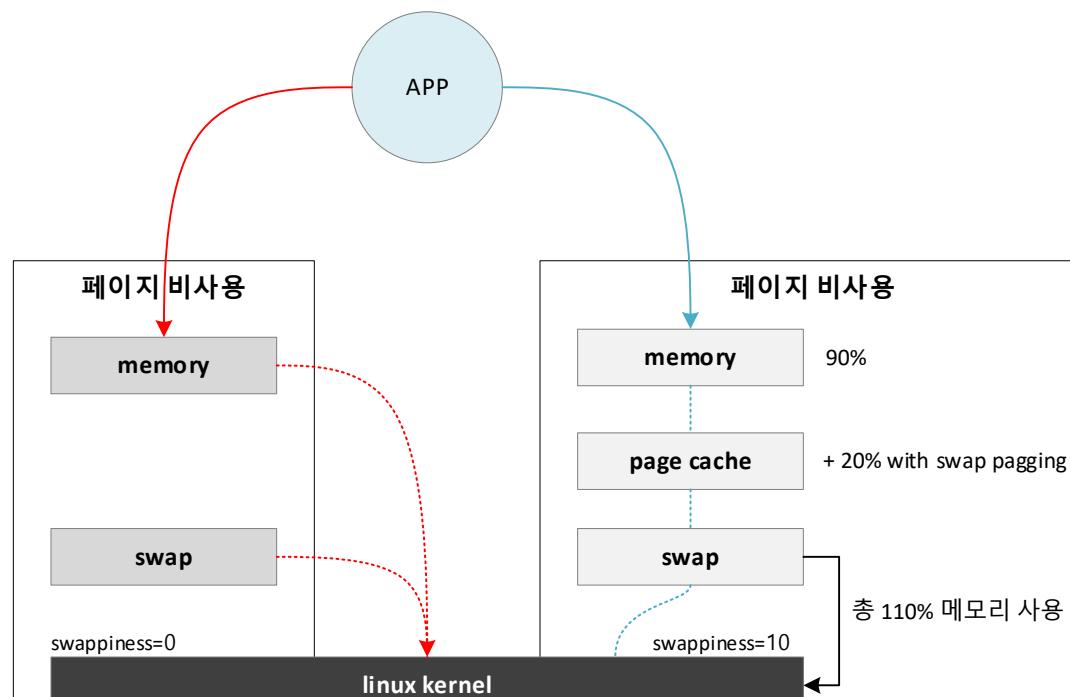


그림 36 스왑

위의 조건을 다음과 같이 표현이 가능하다. infinity 는 쉽게, 최우선 순위이다.

$$0 \leq \text{request} \leq \text{Node Allocatable} \quad \text{and} \quad \text{request} \leq \text{limit} \leq \text{infinity}$$

<sup>36</sup> <https://github.com/kubernetes/design-proposals-archive/blob/main/node/resource-qos.md>

위의 내용으로 임의로 조건을 구성하면 다음과 같다.

- POD에서 CPU의 사용율에 대해서 제한을 요청한다.
- POD에서 요청한 CPU 자원 크기 및 해당 CPU 시간만큼 CPU에 독립적으로 요청 및 격리를 한다. POD 생성 시, 이를 cgroup에 구성 후, 애플리케이션 컨테이너에 전달한다. 여기서는 임의로 800shares를 요청한다.
- CPU 사용량이 초과한 경우, 예를 들어서 POD에 컨테이너 A/B가 있는데, 컨테이너 A는 600, 컨테이너 B는 300를 요청한 경우, 총 합은 900milli CPU가 된다. 여기서 초과된 100 milli CPU 자원이 발생한다. 만약, limit가 구성이 되어 있지 않는 경우, A/B는 2:1 비율로 CPU를 조정하며, 만약 limit가 구성이 되어 있는 경우, 초과 100 milli CPU에 대해서 cgroup에서 제한을 한다.
- POD는 기본적으로 호스트의 CPU가 여유가 있는 경우, POD가 사용이 가능한 CPU 자원 크기만큼 사용하며, 그렇지 않는 경우 제한을 한다.

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

```
# swapoff -a  
# sysctl vm.swappiness=0
```

하지만, 버전 v1.28 이후부터는 스왑 기능 사용 가능<sup>37</sup>하다. 하지만, 아직 베타 기능이며 사용을 원하는 경우 참조를 참조한다. 참고로, 스왑이 동작중이면, kubeadm에서 경고를 발생하면서 설치는 진행하지만, kubelet 서비스가 올바르게 동작하지 않을 수 있다.

## 4. 쿠버네티스 주요 자원 설명

쿠버네티스는 정말로 다양한 자원이 있다. 여기에서 모든 것을 다루기는 어렵지만, 일반적으로 자주 사용하는 쿠버네티스 자원에 대해서 간단하게 설명 후, 명령어 기반으로 각 자원에 대해서 확인한다.

### POD(po)

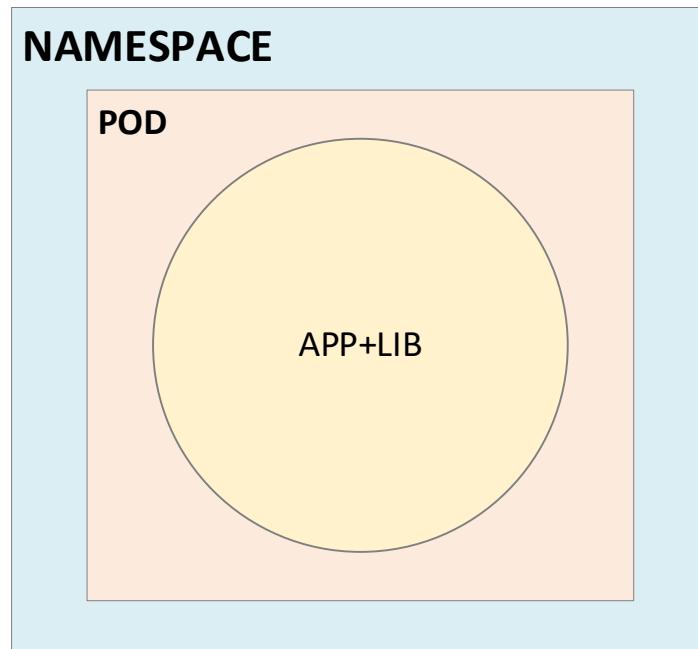


그림 37 POD 구성

<sup>37</sup> <https://kubernetes.io/blog/2023/08/24/swap-linux-beta/>

쿠버네티스에서 사용하는 모든 자원은 기본적으로 POD 기반으로 동작한다. 컨테이너는 쿠버네티스 자원에서 별도로 확인이 불가능하며, 모든 자원은 POD 단위로 화면에 출력 된다.

```
# kubectl get pod  
NAME      READY  
wordpress  1/2
```

포드 밑에 컨테이너가 총 3 개가 동작하면 READY 는 3/3 으로 표시가 되며, 만약, 컨테이너 중 문제가 발생하면 READY 는 2/3 와 같이 표시가 된다. 어떠한 컨테이너가 올바르게 동작하지 않는지 위의 명령어로 확인이 불가능하다. POD 가 생성이 되면, 각 POD 는 아이피를 가지고 있으며, 해당 아이피를 통해서 컨테이너에 접근이 가능하다. POD 에 생성된 아이피는 다음과 같이 클러스터 도메인을 통해서 접근이 가능하다.

예를 들면, POD 가 생성이 되면 POD IP 및 도메인을 생성한다.

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

위의 내용을 가지고, 네임 스페이스 이름이 "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 자원들을 POD 및 서비스 도메인을 통해서 구성한다. 자세한 내용은 뒤에서 더 설명한다.

## Pause(Infra-Container)

쿠버네티스에서 기본적으로 주요 인프라 자원을 POD 기반으로 공유 및 구성한다. 컨테이너 애플리케이션을 실행 시 한 개의 포드에서 컨테이너를 1:N 혹은 1:1 형태로 구성이 된다. 시스템 영역에서는 포드라고 부르지 않고, 인프라 컨테이너(Infra Container)라고 부른다.

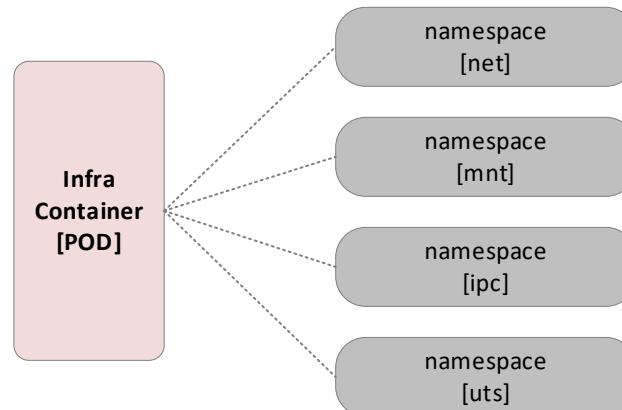


그림 38 infra-container 와 namespace

POD 개념은 인프라 컨테이너로 구성이 되며, 이는 /pause 애플리케이션을 통해서 생성 및 구성이 된다. /pause 애플리케이션은 컨테이너들의 PID=1 를 구성하며, 가상머신의 하이퍼바이저와 비슷한 역할을 한다. 다만, 큰 차이점이라고 하면 /pause 는 직접 처리하는 게 아니라, 호스트를 통해서 시스템 콜을 공유 및 처리한다.

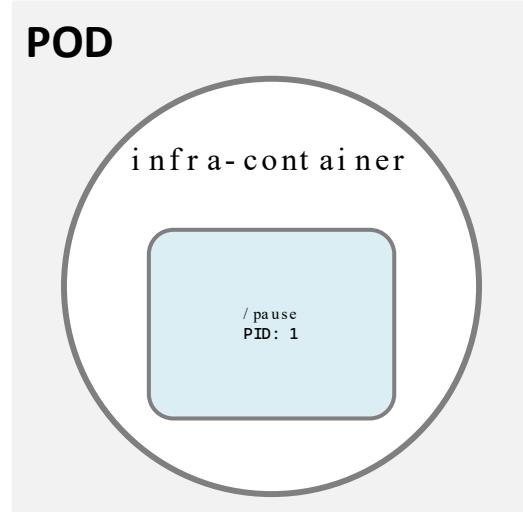


그림 39 인프라 컨테이너와 /pause

포드를 구성하는 부분은 `/pause`라는 애플리케이션이다. 이 애플리케이션 쿠버네티스 기반으로 구현 제품에서는 POD 개념을 사용하지만, 사용하는 POD 소프트웨어는 다를 수 있다. 보통 아래와 같이 `/pause` 애플리케이션을 사용한다.

이름	제품	회사
<b>Pause</b>	Vanila Kubernetes	Kubernetes
<b>catatonit</b>	OpenShift	Redhat
<b>Rancher</b>	Rancher	SuSE

## Service(svc)

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

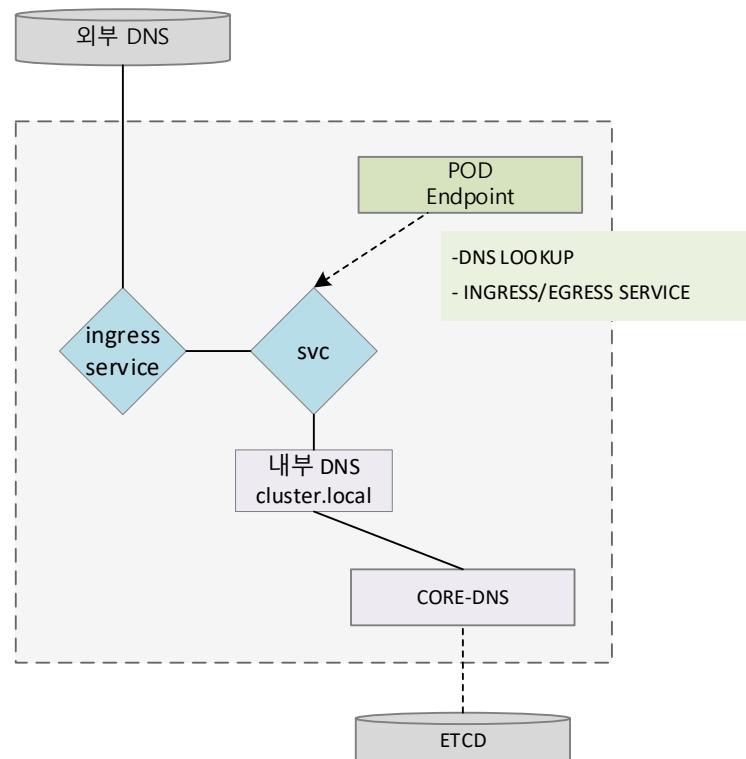


그림 40 POD 및 네트워크

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
```

POD에서 구성된 A 레코드 및 아이피는 다음과 같은 형식으로 서비스 도메인이 CoreDNS에서 구성 및 생성이 된다.

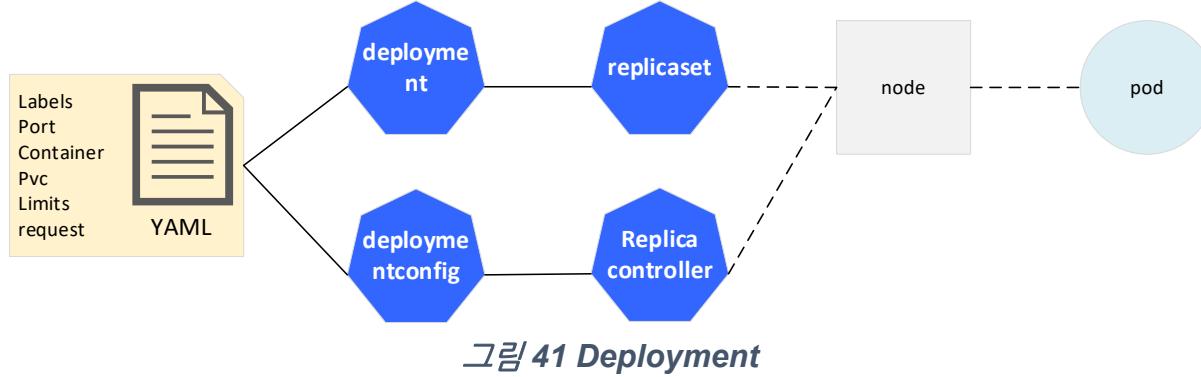
```
10-10-0-4.svc-pod.first-namespace.svc.cluster-domain.example.
```

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

옵션	설명
--pod-network-cidr string	일반적으로 10.0.0.0/8 대역
--service-cidr string	10.96.0.0/12
--service-dns-domain	cluster.local

## Deployment(deploy)

Deployment는 ReplicaSet에서 사용하는 설정 정보들이 있다. Deployment에는 POD, SVC, Container, PVC, Template 그리고 Labels 같은 정보가 저장이 되어있다. 사용자가 아래와 같이 YAML 파일을 작성하여, 쿠버네티스 클러스터에 생성이 가능하다.



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 POD의 롤아웃(rollout) 및 Pod 관리가 가능하다. 사용량이 늘어나는 경우 Deployment는 RS(ReplicaSet)를 통해서 POD를 확장한다. 새로운 Deployment가 생성 및 업데이트가 되면, 오래된 POD를 제거하면서 새로운 RS 기반으로 POD를 생성한다. 문제가 발생 시, RS에서 가지고 있는 기록(history)기반으로, 특정 시점으로 롤백(roll-back)이 가능하다.

## ReplicaSet(rs)

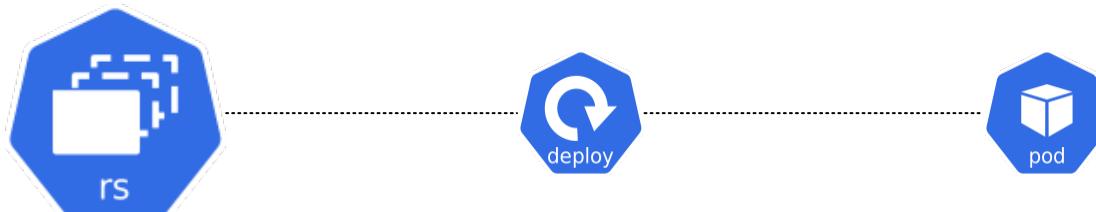


그림 42 deployment, replicaSet, POD 관계

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

## Replication Controller(rc)

초기 쿠버네티스는 replication-controller를 통해서 애플리케이션 배포 및 관리가 이루어졌다. replication-controller의 주요 역할은 replicaset과 비슷하게 POD 관리 및 애플리케이션 복제를 한다. 현재 쿠버네티스는 "ReplicationController" 사용을 권장하지 않는다.

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

### 그림 43 가급적면 ReplicaSet

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

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

## Volume

쿠버네티스에서 볼륨은 기본적으로 EmptyDir 혹은 LocalDisk 형태로 제공이 된다. 컨테이너에서 발생하는 데이터를 영구적으로 저장 및 활용하기 위해서는 위의 두 개 형식으로 적절하지 않기 때문에 쿠버네티스에서는 CSI(Container Storage Interface)를 통해서 볼륨을 제공한다.

CSI에서 제공하는 볼륨은 여러 형식이 있으며, 대표적으로 많이 사용하는 CSI 드라이버는 다음과 같다.

1. NFS4
2. GlusterFS
3. Ceph

이러한 확장 드라이버를 통해서 클러스터에서 발생한 데이터를 영구적으로 저장할 수 있다. 위의 볼륨 드라이버를 구성하지 않아도 사용이 가능하지만, 클러스터를 운영하기 위해서는 최소 한 개의 영구적인 CSI 드라이버 구성은 추천된다.

## StatefulSet

StatefulSet은 Deployment와 거의 동일한 동작 방식을 가진다. 다만, POD가 실행이 될 때 각기 독립적으로 실행이 되면서 넘버링이 된다. 예를 들어서, POD 기반으로 redis나 혹은 mariadb와 같은 서비스를 구성하는, 데이터베이스 클러스터링 서비스를 구성하면서 독립적으로 스토리지를 가지고 간다면, StatefulSet 서비스로 구성은 권장된다.

## CronJob

특정 POD를 명시된 날짜 및 시간에 실행한다. 리눅스에서 사용하는 systemd-timer, crond과 동일 동작 방식이다. CronJob은 One-shot 서비스이기 때문에, 한번 실행 후, 다시 실행하지 않는다.

## DaemonSet(ds)

DaemonSet은 앞에서 이야기한 StatefulSet과 비슷한 서비스이다. DaemonSet은 "각 노드에서 동작하면서 상태 정보를 가지고 있는 POD"라고 이해를 하여도 된다. 특정 POD를 각 노드에 독립적으로 실행시키며, 이를 통해서 노드 정보나 혹은 서비스 생성 및 확인을 한다.

POD의 정보는 앞에서 사용하는 Deployment에서 관리하지만, 동작방식은 Deployment와 다르게 각 노드에서 독립적으로 실행 및 관리가 된다. 일반적으로 노드 모니터링이나 혹은 서비스 모니터링 용도로 많이 사용한다.

## **Secret(sc)**

쿠버네티스에서 민감한 정보를 다룰 때, secret 를 통해서 저장 및 배포를 한다. secret 에 저장되는 정보는 기본적으로 base64 형태로 데이터를 저장 및 보관하기 때문에, 자체적으로는 암화를 하지 않는다. 보통 secret 에는 다음과 같은 정보를 저장 및 배포한다.

- TLS 비공개 키
- 특정 소프트웨어 시리얼 키
- 시스템에서 사용하는 아이디 및 비밀번호

이와 같은 정보를 저장 후, POD 에 전달 및 배포한다. 전달 방식은 두 가지 형태로 POD 에 자원을 전달한다.

1. 불투명한 형태의 데이터
2. 서비스 어카운트 토큰(Service Account Token)
3. 도커 설정 파일(Docker Config Token)
4. 기본적인 사용자 아이디 및 비밀번호(username, password)
5. SSH 비공개 키(SSH Private Key)
6. TLS 비공개 키(TLS Private Key)

## **ConfigMap(cm)**

ConfigMap 은 Secret 과 거의 동일하다. 다만, 차이점은 ConfigMap 은 인코딩을 제공하지 않으며, 일반적인 텍스트(Plain Text)형태로 자료를 저장 및 배포한다. ConfigMap 은 두 가지 형태로 POD 에 자원을 전달한다.

1. 파일 형태
2. 문자열 형태

사용하는 애플리케이션 유형에 따라서 설정파일을 전달한다.

## **Label**

쿠버네티스에서 생성되는 모든 자원은 최소 한 개 이상의 Label(레이블)을 가지고 있다. 레이블은 자원 구별하는 용도로 사용하기도 한다.

## **Selector**

Selector(선택자)는 레이블 기반으로 동작한다. 사용자가 원하는 내용을 선택자를 통해서 조회 및 제거가 가능하다. 쿠버네티스에서 사용하는 자원은 대다수 선택자 기반으로 동작한다.

## **Annotation**

리소스에 추가적인 설정 및 내용이 필요한 경우 주석(Annotation)를 통해서 전달한다. Ingress 나 Service Mesh 와 같은 확장 서비스에 특정 기능에 대해서 활성화가 필요할 때, 주석을 통해서 활성화 한다.

## **시작하기 전에...**

### **1. 최종확인**

쿠버네티스에서 네임스페이스를 고정하기 위해서 컨텍스트를 수정해야 한다. 쿠버네티스에서 기본적으로 사용자명@클러스터 그리고 네임스페이스 이름으로 구성이 되어 있다. 기본 컨텍스트는 다음처럼 설정 및 구성이 되어 있다.

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

기본 네임스페이스는 비어 있으며, 비어 있는 경우, 쿠버네티스는 기본적으로 **default** 이름의 네임스페이스를 사용한다. 변경을 원하는 경우, 아래 이름으로 변경이 가능하다.

```
# kubectl config set-context --current --namespace=
```

위의 명령어는 현재 클러스터(--current)에서 네임스페이스 변경한다. 만약, 현재 클러스터가 아닌, 다른 클러스터이면, 클러스터의 이름을 넣어서, 클러스터 위치를 변경하면 된다.

```
# kubectl config set-context admin@apac-zone2-service --namespace=service
```

현재, 어떠한 클러스터를 사용하고 있는지 확인하기 위해서 아래와 같은 명령어로 확인이 가능하다.

```
# kubectl config get-contexts
# kubectl config get-users
# kubectl config get-clusters
```

만약, 네임스페이스를 자주 변경하는 경우, 아래처럼 단축키를 만들어서 사용한다.

```
# sn () { kubectl config set-context --current --namespace="$1"; }
# sn default
# declare -F | awk '{ print $3 }' | grep ^sn
# gn () { kubectl config view -o jsonpath='{..namespace}'; }
# gn
```

클러스터가 올바르게 구성이 되었는지 확인하기 위해서, 아래 명령어로 컨트롤러 정보를 확인한다. 올바르게 eth1 번 인터페이스를 통해서 API 가 올바르게 동작하는지 확인한다.

```
# 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'.
```

학습하기 전, 쿠버네티스의 전체적인 상태를 확인한다. 기본 구성원 및 노드 그리고 API 주소가 잘 동작하는지 확인한다. 모든 작업은 node1 혹은 workstation 노드에서 명령어를 실행한다.

```
node1]# 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
```

```
node1]# kubectl get nodes
NAME           STATUS  ROLES AGE VERSION
node1.example.com  Ready   master  5d6h  v1.18.2
node2.example.com  Ready   <none>  5d6h  v1.18.2
node3.example.com  Ready   <none>  5d6h  v1.18.2
```

위와 같이 출력이 되면 랩 진행에 문제가 없다.

## 2. 명령어 자동완성

쿠버네티스 명령어를 학습하기 위해서 node1 서버에 접근한다. 명령어 kubectl 를 좀 더 쉽게 사용하기 위해서 아래와 같이 명령어를 실행한다. 여러 노드를 사용하는 경우, workstation 에서 아래 명령어를 실행한다.

```
node1]# dnf install bash-completion -y
node1]# kubectl completion bash >> /etc/profile.d/kubectl.sh
node1]# source /etc/profile
```

만약, kubectl, kubeadm 명령어까지 추가를 원하는 경우 아래와 같이 작업을 수행한다.

```
node1]# kubectl completion bash >> ~/etc/bash_completion.d/kubectl
node1]# kubeadm completion bash >> ~/etc/bash_completion.d/kubeadm
node1]# complete -r -p
```

특정 사용자만 사용하기 위해서는 다음과 같이 명령어를 실행한다.

```
node1]# source <(kubectl completion bash)
node1]# mkdir -p ~/.kube/
node1]# kubectl completion bash > ~/.kube/completion.bash.inc
node1]# printf "source '$HOME/.kube/completion.bash.inc'" >> $HOME/.bash_profile
node1]# source $HOME/.bash_profile
node1]# kubeadm completion bash >> ~/.bashrc
node1]# complete -r -p
```

자동완성 기능은 한 가지 이상의 쉘은 지원한다. bash 이외 zsh, powershell 및 fish 와 같은 기능도 제공한다. 위의 기능이 올바르게 설치가 되면 다음처럼 명령어를 테스트한다.

```
node1]# 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
certificate describe   label      run
```

이제 쿠버네티스 기본 명령어로 넘어갈 준비가 완료가 되었다.

### 3. YAML 문법

쿠버네티스는 YAML 기반으로 작성한다. 앤서블과 같은 YAML 문법을 사용한 사용자에게는 매우 친숙한 문법이다. 기존 YAML과 동일하게 문법 자체는 띄어쓰기 기반으로 되어 있기 때문에, 올바르게 들여쓰기를 꼭 해야 한다. 작성 시 지켜야 할 조건은 다음과 같다.

- 최소 2칸의 들여쓰기
- 키 이름 뒤에 콜론(:)으로 구분 후, 가급적이면 한 칸 띄어쓰기
- 리스트 혹은 딕셔너리는 "-" 표시로 시작

위의 조건으로 아래처럼 작성이 가능하다.

**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 **List(목록)/Dictionary(사전)**는 아래와 같이 작성한다. 작성 방법은 두 가지가 있다. 첫번째는 YAML 형태, 두번째는 JSON 형태의 리스트이다.

**Fruits:**

```
- Apple
- Melon
```

**Fruits:** [ Apple, Melon ]

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

**Users:**

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

YAML 파일에는 어느정도 형식(format)이 있다. 아래는 예제이다. 파일 이름은 "**basic-syntax.yaml**"으로 저장한다.

```
# cat <<EOF> basic-syntax.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
node1]# kubectl apply -f basic-deployment-nginx.yaml

```

YAML의 위쪽은 **API 버전 및 자원의 형식**을 선언합니다.  
그 외 나머지는 metadata, annotation 정보가 구성이 됩니다.

**spec:** 이 위치는 자원이 사용하는 옵션값이 들어 갑니다.  
예를 들어서, deployment 형식에서는 "**template:**", "**selector:**"을(를) 통해서 복제 및 POD 개수를 확인한다.

**spec:** POD에서 관리할 컨테이너 정보. 보통 이 영역에는  
볼륨 및 포트와 같은 정보들이 구성이 된다.

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 **관습적으로** 표시한다. “**name:**”부분은 실제로 쿠버네티스에서 사용하는 자원들의 이름, 예를 들어서 **컨테이너/POD**의 이름이다. 이 이름을 통해서 어떤 애플리케이션 혹은 POD 동작하는지 확인이 가능하다. 미리 알아 두면 좋은 부분은 **POD**는 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, 현재까지는 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
annotations:
  imageregistry: "https://hub.docker.com/"

```

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

```

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

```

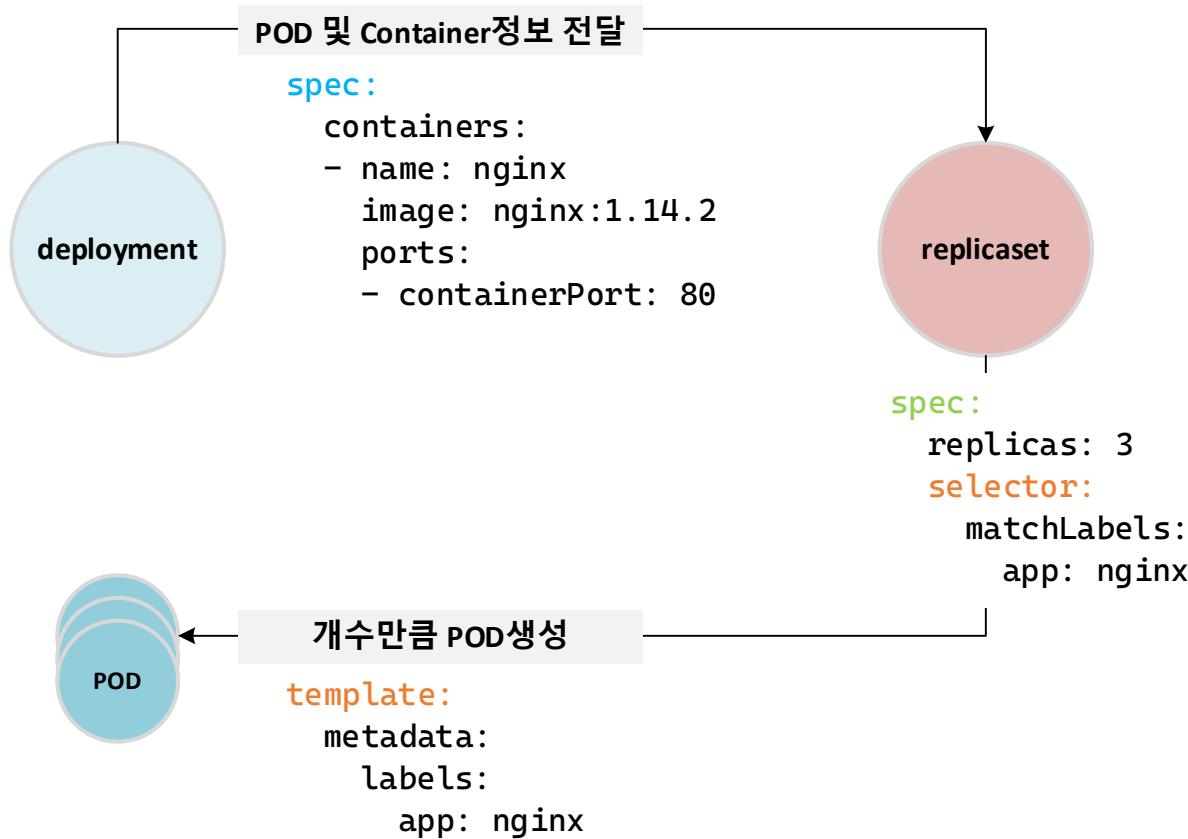
"**spec:**"에 복제자(replicas)가 사용할 개수 그리고 복사할 대상이 "**matchLabels:**"라는 영역에 선언이 되어 있다. 아래 예제에서는 선택한 자원은 POD의 레이블 이름 "app" 그리고 값은 "nginx"이다.

```

spec:
  containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80

```

위의 관계를 간단하게 그림으로 표시하면 아래와 같은 관계를 가지고 동작한다.



그림/ 44 template, selector

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 생성 시 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`<sup>38</sup>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.

## 연습문제

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

1. 자원의 이름을 nginx에서 apache로 변경한다.
2. 모든 메타정보의 레이블을 nginx에서 apache로 변경한다.
3. 컨테이너 이미지를 nginx:1.14.2에서 apache:latest로 변경한다.
4. 포트는 80으로 설정한다.

# 기본 기능

## 1. 네임스페이스

쿠버네티스에서 생성되는 모든 자원은 네임스페이스에서 생성 및 관리가 된다. 클러스터는 컴퓨터 노드 및 설정을 가지고 있으며, 클러스터에 구성된 사용자는 네임스페이스를 통해서 자원을 생성 및 구성한다. 쿠버네티스의 네임스페이스(혹은 프로젝트)를 확인하는 방법은 다음과 같다.

```
# 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 완료가 되면 기본적으로 다음과 같이 네임 스페이스를 제공한다. 아래는 기본적으로 생성하는 네임스페이스이라서, 가급적이면 제거하지 말고 그대로 사용한다. 출력되는 네임스페이스 이름은 쿠버네티스 버전별로 다르지만, 일반적으로 최근 버전에는 아래와 같이 구성이 되어있다.

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

<sup>38</sup> 이 부분은 추후에 다루도록 한다.

<b>kube-node-lease</b>	노드에서 소유하고 있는 네임스페이스. 이를 통해서 클러스터의 노드 상태를 확인한다. 노드의 .status 정보를 갱신 합니다. “Lease object” 객체는 kube-node-lease 네임스페이스에 있습니다. 각 노드는 Lease object 를 가지고 있습니다.
------------------------	--

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

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

생성된 네임스페이스 “first-namespace”는 다음 명령어로 확인이 가능하다.

```
# 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.

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: second-namespace
```

앞에서 이야기하였지만, 쿠버네티스는 네임스페이스를 변경하기 위해서 컨텍스트 수정이 필요하다. 수정을 위해서 앞에서 만든 sn 명령어나 혹은 kubectl 명령어로 수정한다.

```
# kubectl create -f second-namespace.yaml
# kubectl config get-contexts
CURRENT      NAME          CLUSTER      AUTHINFO      NAMESPACE
*           kubernetes-admin@kubernetes  kubernetes  kubernetes-admin
# sn second-namespace
# kubectl config set-context --current --namespace second-namespace
```

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.

```
# kubectl config view
# kubectl config view -o jsonpath='{..namespace}'
```

위의 명령어가 반영이 되면 ~/.kube/config 혹은 /etc/kubernetes/admin.conf 에 다음과 같이 변경된 부분이 확인된다. 일반적으로 수정하기전에 기본 내용은 아래와 같다.

```
contexts:
```

```

- context:
  cluster: kubernetes
  user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes

```

컨텍스트 변경이 되면 아래와 같이 내용이 수정이 된다.

```

contexts:
- context:
  cluster: kubernetes
  namespace: second-namespace
  user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes

```

특정 네임스페이스만 지정해서 사용하려면, 다음과 같은 명령어로 노드에서 실행한다.

```

# kubectl config set-context --current --namespace=first-namespace
Context "kubernetes-admin@kubernetes" modified.
# 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. 아래 명령어는 현재 사용중인 네임스페이스에서 POD를 확인하는 명령어이다. 모든 쿠버네티스 명령어에는 **-n**, **--namespace** 옵션이 있는데, 이를 통해서 언제든지 특정 네임스페이스 자원 조회를 명시적으로 가능하다.

```

# kubectl get pods -n first-namespace
No resources found in first-namespace namespace.

```

아래 명령어는 모든 네임 스페이스에서 사용하는 pod를 확인하는 명령어이다.

```

# kubectl get pods --all-namespaces
NAME                                READY   STATUS    RESTARTS   AGE
kube-system  calico-kube-controllers-6fcbbfb6   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

```

쿠버네티스에서 pod를 출력 시 정렬이 필요한 경우 **--sort-by** 명령어를 통해서 정리가 가능하다.

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

사용한 옵션은 다음과 같은 기능을 지원한다.

옵션	설명
--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-label	옵션은 자원에 레이블이 설정이 되어 있으면, 해당 레이블을 같이 화면에 출력한다.

## 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-context로 기본 네임스페이스를 변경한다.

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

## 2. 생성/삭제/확인

쿠버네티스에서 주요 목적은 kubectl 명령어를 통해서 자원을 생성/조회/관리가 주요 목적이다. POD 생성 및 그 이외 자원을 관리 시 많이 사용하는 명령어에 대해서 확인하도록 한다.

### 실행(run)

run 명령어는 YAML 파일이 아닌 일시적으로 컨테이너 및 POD 를 실행하는 명령이다. 의외로 자주 사용하는 명령어인데, 보통 확인 용도로 POD 생성시 사용하다. 자주 사용하는 옵션은 아래와 같다.

옵션	설명	비교
--image nginx pod-nginx	간단하게 POD 실행한다. 보통 맨 뒤쪽은 POD 의 이름으로 사용한다.	간단하게 POD YAML 생성
--dry-run=client --output=yaml pod-nginx	실제로 생성하지 않고, kubectl 명령어에서 YAML 형태로 자원을 생성 후 화면에 출력한다.	시 많이 사용한다.

--stdin(-i)	컨테이너를 실행하면서 가상 TTY 를 통해서 대화형으로 시작한다.	
--port	POD 에서 사용할 포트번호를 명시한다.	
--expose	POD 를 SERVICE 로 노출한다. 기본적으로 ClusterIP 형태로 되며, 이를 사용하기 위해서 앞서 이야기한 --port 가 선언이 되어 있어야 한다.	
--rm	Podman 의 --rm 옵션과 똑같다. 컨테이너 혹은 POD 가 중지가 되면 제거가 된다.	

위의 내용을 가지고 간단하게 YAML 파일 및 컨테이너를 생성한다. 단순 실행은 다음과 같이 실행이 가능하다. 생성한 자원은 get 명령어를 통해서 확인이 가능하다.

```
# kubectl run --image=node1.example.com/testapp/php-blue-green:v1 php-pod
# kubectl get pod
```

올바르게 생성이 되었으면, 이를 제거하기 위해서 delete 명령어를 통해서 제거가 가능하다.

```
# kubectl delete pod php-pod
# kubecrl get pod
```

간단하게 Pod 를 생성 및 삭제했다. 만약, POD 생성이 아닌, YAML 파일이 필요한 경우, 다음과 같이 실행한다.

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

간단하게 POD 를 생성하였고, 또한 POD 를 직접 생성하지 않고 YAML 파일로 구성해보았다. 위의 내용을 기반으로 쿠버네티스 자원 생성에 대해서 좀 더 확인하도록 한다.

## 적용(apply)

쿠버네티스에서 자원을 사용 및 구성하기 위해서는 두 가지 명령어를 많이 사용한다. 아래는 자원을 생성하기 위한 kubectl 명령어에 대한 설명이다.

이름	설명	공통사항
<b>create</b>	자원을 YAML/CLI 형태로 구성 시 사용한다. 이 명령어는 생성한 자원에 대해서 기록을 남기지 않으며, 사용자가 명시한 내용대로 클러스터에 생성한다. CLI 부분은 사용자가 명령어로 자원 생성 혹은 YAML 파일 생성 시 사용한다. 다만, 모든 자원을 명령어로 생성이 가능하지 않다.	이 명령어들은 kustomize 기능을 제공한다. 앞서 이야기한 부분처럼, 차이점은 기록을 남기느냐 남기지 않느냐 차이가 있다.
<b>apply</b>	YAML 및 DIRECTORY 형태로 구성 및 생성한다. apply 와 create 의 제일 큰 차이점은 apply 는 생성된 자원의 리비전 기록을 가지고 있으며, 디렉터리 기반으로 생성이 가능하다.	

위 내용을 실험하기 위해서 간단하게 POD 를 생성하도록 한다. 여기서 사용하는 POD 는 php-bg 라는 컨테이너를 사용하도록 한다. 파일명은 php-bg-v1.yaml 으로 아래와 같이 작성한다.

```
apiVersion: v1
kind: Pod
metadata:
```

```

creationTimestamp: null
labels:
  run: php-pod-v1
  name: php-pod=v1
spec:
  containers:
    - image: node1.example.com/testapp/php-blue-green:v1
      name: php-pod
      resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}

```

작성된 애플리케이션을 두 가지 명령어로 각각 클러스터에 등록한다. 첫번째는 apply 명령어로 적용한다. 적용 후, 올바르게 동작하는지 확인한다.

```

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

```

크게 문제없이 생성이 되면 이번에는 create 명령어를 통해서 생성한다.

```

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

```

위의 명령어 둘 다, POD 자원을 문제없이 생성이 된다. 이 명령어들은 다음과 같은 동작 차이점이 있다.

- apply 명령어는 kubectl annotation 를 생성한다.
- create 명령어는 kubectl annotation 를 생성하지 않는다.

이 부분은 다음과 같은 명령어로 확인이 가능하다.

```

# kubectl get -f php-pod-v1.yaml -o yaml | grep -A1 kubectl
  kubectl.kubernetes.io/last-applied-configuration: |
{"apiVersion":"v1","kind":"Pod","metadata": {"annotations":{},"creationTimestamp":null,
"labels":{"run":"php-pod-v1"},"name":"php-pod-
v1","namespace":"default"}, "spec": {"containers": [{"image": "nginx", "name": "php-pod-
v1", "resources": {}}], "dnsPolicy": "ClusterFirst", "restartPolicy": "Always"}, "status": {}}

```

내용을 수정 후, 다시 적용하면, POD 는 업데이트가 수행이 된다. 다만, create 으로 생성 후, apply 를 하는 경우 다음과 같은 메시지를 출력하면서 업데이트를 수행한다.

```

# kubectl delete -f php-pod-v1.yaml
# kubectl create -f php-pod-v1.yaml
# kubectl get -f php-pod-v1.yaml -o yaml | grep -A1 kubectl
# kubectl apply -f php-pod-v1.yaml
Warning: resource pods/php-pod-v1 is missing the kubectl.kubernetes.io/last-applied-
configuration annotation which is required by kubectl apply. kubectl apply should only
be used on resources created declaratively by either kubectl create --save-config or
kubectl apply. The missing annotation will be patched automatically.

```

## pod/php-pod-v1 configured

create 명령어로 apply 와 동일한 효과를 만들기 위해서는 --save-config 옵션을 사용하면 된다. 또한, create 명령어는 다음과 같은 기능을 지원하기도 한다. 디렉터리에 있는 YAML 파일을 한 번에 등록하기 위해서 아래와 같이 명령어를 실행한다. 다만, 이와 같은 방법으로 사용하기 위해서는 파일 앞에 숫자로 순서를 걸어두는 게 좋다.

```
# ls -l wordpress
00-database.yaml  01-wordpress.yaml
# kubectl create -f wordpress/
```

## 제거(delete)

제거를 하기 위해서 위에서 사용한 delete 명령어를 사용하면 된다. 자원을 제거하는 방법은 보통 다음과 같이 지원한다.

1. YAML 파일 기반으로 제거
2. Label 기반으로 자원 제거
3. delete 명령어로 특정 자원 명시 제거

수동으로 생성한 자원이 아닌 경우, 일반적으로 YAML 파일이나 혹은 Label 기반으로 제거를 권장한다. 앞에서 생성한 php-pod-v1.yaml 기반으로 제거를 수행하면 다음과 같이 수행한다.

### YAML 파일 기반

```
# kubectl delete -f php-pod-v1.yaml
```

### Label 기반

```
# kubectl delete -l app=php-pod-v1
```

### 특정 자원 명시

```
# kubectl delete pod/php-pod-v1
```

위와 같은 방법으로 제거한다. 상황에 따라서 적절하게 사용하며 되며, 가급적이면 YAML 파일로 만들어서 YAML 으로 자원을 제거하는 게 안전하다.

## 생성(create, kustomize)

create 명령어는 여러 자원 생성이 가능하다. 예를 들어서 사용자가 다음과 같은 자원을 생성을 원한다고 가정한다. 자원의 기능(deployment, service) 및 역할은 뒤에서 더 다루니, 지금은 편안하게 자원을 생성한다.

- POD
- Deployment
- Service

이 자원을 생성하기 위해서 YAML 파일을 kubernetes-docs 에서 가져와서 할 수 있으며, kubectl 명령어를 통해서 자원 생성이 가능하다. 생성을 하기 위해서 앞에서 사용한 두 가지 옵션을 활용한다. 이 옵션은 앞에서 몇 번 다루었다.

1. --output=yaml
2. --dry-run=client

위의 옵션을 사용해서 자원들은 순서대로 생성한다. 아래 명령어는 YAML 파일을 직접 만들지 않고, 명령어를 통해서 YAML 파일을 생성한다. 여기서는 NGINX 를 POD 를 통해서 생성한다.

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

아래 명령어는 위와 동일하나, POD 가 아닌, Deployment 자원으로 YAML 파일로 자원을 구성한다.

```
# kubectl create deployment --image=nginx deploy-nginx --output=yaml --dry-run=client > deploy-nginx.yaml  
# kubectl create -f deploy-nginx.yaml
```

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

create 명령어는 API 에서 지원만 하면, 대다수 자원에서 대해서 YAML 형태로 파일 생성이 가능하다. 생성 후, 좀 더 추가하거나 원하는 내용으로 수정하면 된다. 위의 자원이 올바르게 구성이 되었는지 get 명령어로 확인한다.

```
# kubectl get pod,deployment,svc
```

혹은 "-k" 옵션을 통해서 디렉토리에 있는 데이터를 쿠토마이저(kustomize)기반으로 사용이 가능하다. 이를 통해서 자원 생성 및 구성을 자동으로 할 수 있다. 앞에서 사용하였던 "-f" 와 다른 부분은 다음과 같다.

7. 리소스 교차 편집
8. 리소스 집합 및 사용자 정의
9. 리소스 생성

예를 들어서 configmap, secret 를 생성하기 위해서 아래와 같이 작성한다. 사용할 데이터 형식 및 데이터를 아래와 같이 파일로 작성한다.

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

위의 내용을 작성 후, 아래와 같이 명령어를 실행하면, 다음과 같은 결과가 나온다.

```
# kubectl create -k ./  
# kubectl get configmap  
> example-configmap-1-t7998h956b 1 28s  
# kubectl describe cm(configmap) example-configmap-1-t7998h956b  
> application.properties:  
> ----  
> FOO=bar
```

앞에서 했던 내용처럼 POD까지 생성하기 위해서는 아래와 같이 YAML 파일을 생성한다. 앞에서 사용하였던 `create` 명령어를 통해서 아래 내용을 구성하여도 상관이 없다.

```
# vi kustomize/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: my-app
spec:
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: app
          image: my-app
          volumeMounts:
            - name: config
              mountPath: /config
      volumes:
        - name: config
          configMap:
            name: example-configmap-1
```

nginx 으로 변경해야 됨.

위의 내용을 아래 명령어로 실행한다. 자원이 올바르게 잘 구성이 되었는지 확인이 가능하다.

```
# kubectl kustomize ./
```

문제가 없으면, configmap, deployment의 내용이 화면에 출력이 된다. 올바르게 출력이 되면, 아래처럼 `deployment.yaml`를 자원(resources)에 추가한다.

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

자원이 올바르게 생성이 되면, 아래 명령어로 확인이 가능하다.

```
# kubectl get pod,configmap
```

앞에서 학습했던, `apply` 명령어도 `kustomize`를 통해서 적용이 가능하다. 이제는 생성된 자원을 확인하는 방법에 대해서 이야기해보도록 한다.

## 확인(get/describe)

get 명령어는 네임스페이스에 구성이 되어있는 자원 목록을 확인 시 사용하는 명령어이다. 이 명령어를 통해서 오브젝트 안에 구성이 되어있는 리소스 확인이 가능하다. 형식은 둘 중 아무거나 사용하면 된다.

```
kubectl get <resource_type>/<resource>  
kubectl get <resource_type> <resource>
```

생성한 POD 를 확인하기 위해서 아래와 같이 확인이 가능하다. 다만, 현재 사용중인 네임스페이스의 POD 만 출력이 된다.

```
# kubectl get pods
```

다른 위치의 네임스페이스 정보를 확인하고 싶으면 다음처럼 명령어를 사용하기도 한다.

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

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

```
# kubectl get deployment -A  
# kubectl get deployment --all-namespaces
```

실시간으로 변경 내용(resource event)을 확인하고 싶은 경우 “-w”옵션을 통해서 실시간으로 갱신 상황을 살펴볼 수 있다.

```
# kubectl get pods -w
```

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

```
# kubectl get pods -o=json
```

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

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

자원 확인이 되었으면, 자원이 어떠한 상태로 구성 및 로그가 발생하였는지 확인이 필요할 때가 있다. 이때 사용하는 명령어는 describe 명령어를 사용하여, 특정 자원의 현재 상태가 어떠한 상태인지 확인이 가능하다.

```
# kubectl describe pod/php-pod  
Name:          php-pod-v1  
Namespace:     default  
Priority:      0  
Service Account: default  
Node:          node2.example.com/10.10.0.2  
Start Time:    Sat, 28 Sep 2024 19:39:04 +0900  
Labels:        run=php-pod-v1  
Annotations:   cni.projectcalico.org/containerID:  
5dd114ac930013a536b18db3dfe5276ba2c24522e4f7139fe8c995cd93f05457  
               cni.projectcalico.org/podIP: 10.10.0.172/32  
               cni.projectcalico.org/podIPs: 10.10.0.172/32  
Status:        Running
```

```
IP:          10.10.0.172
IPs:
  IP: 10.10.0.172
Containers:
  php-pod-v1:
    Container ID: cri-o://71eb2f504b849d650701c7c5fad9358ac6dc7cc71551239d3d5fb0850476a4f8
    Image:        node1.example.com/testapp/php-blue-green:v1
```

다른 자원도 **describe** 명령어를 통해서 확인이 가능하다. YAML 파일이 있는 경우 다음과 같이 조회가 가능하다.

```
# kubectl describe -f deploy-nginx.yaml
Name:          deploy-nginx
Namespace:     default
CreationTimestamp: Sat, 28 Sep 2024 23:26:09 +0900
Labels:        app=deploy-nginx
Annotations:   deployment.kubernetes.io/revision: 1
Selector:      app=deploy-nginx
Replicas:     1 desired | 1 updated | 1 total | 0 available | 1 unavailable
...
Events:
  Type  Reason           Age   From            Message
  ----  -----           ----  ----
  Normal  ScalingReplicaSet  5s    deployment-controller  Scaled up replica set
  deploy-nginx-7d74c85c6f to 1
```

자원이 생성이 되어 있으면 올바르게 동작하지만, 현재 자원이 구성이 되지 않는 경우 위의 메시지처럼, **1 unavailable**라고 메시지가 출력이 된다. 정상적으로 동작되게 하려면 아래와 같이 작업을 수행 후 **describe**로 확인한다.

```
# kubectl create -f deploy-nginx.yaml
# kubectl describe -f deploy-nginx.yaml
Name:          deploy-nginx
Namespace:     default
CreationTimestamp: Sat, 28 Sep 2024 23:26:09 +0900
Labels:        app=deploy-nginx
Annotations:   deployment.kubernetes.io/revision: 1
Selector:      app=deploy-nginx
Replicas:     1 desired | 1 updated | 1 total | 1 available | 0 unavailable
```

혹은 위와 동일한 결과를 아래와 같은 명령어로 가능하다.

```
# kubectl describe deployment deploy-nginx
```

이외 자원들에 대해서도 **describe** 명령어를 통해서 확인이 가능하다. 다른 자원들의 정보를 확인하고 싶을 때, 위와 똑같은 방법으로 조회를 하면 된다.

## 연습문제

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

기존에 사용하였던 yaml 파일을 사용해서 다음처럼 서비스 구성을 한다.

1. deployment에 이름은 public-vsftpd라는 이름으로 구성.
2. vsftpd 이미지를 사용해서 yaml 파일을 작성.
3. 이미지 파일은 아래의 주소에서 받기가 가능.  
10. <https://hub.docker.com/r/fauria/vsftpd/>
4. 받은 이미지의 replica 개수는 1개로 합니다.

위의 명령어를 사용하여 쿠버네티스의 service, deployment, replicaset에 구성된 내용들을 확인한다.

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

### 문제 1

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

### 문제 2

1. httpd 이미지를 nginx으로 변경한다.
2. 기존의 POD 및 컨테이너의 이름을 nginx으로 변경.
3. 레이블 정보를 name=nginx, version prod로 변경.
4. /usr/share/nginx/html/index.html에 "Hello Nginx"라는 메시지가 출력되는지 확인.
5. 이 컨테이너는 반드시 basic에서 동작해야 한다.

## 3. 멀티 컨테이너 구성

쿠버네티스에서 생성하는 POD는 한 개 이상의 컨테이너를 가지고 운영이 된다. 앞에서는 단일 컨테이너 생성에 다루었다. 멀티 컨테이너는 별 다른 구성이 없다. POD에 한 개 이상의 컨테이너가 붙으면 멀티 컨테이너다. 구성이 되면 아래 그림과 같이 POD에 애플리케이션이 구성이 된다.

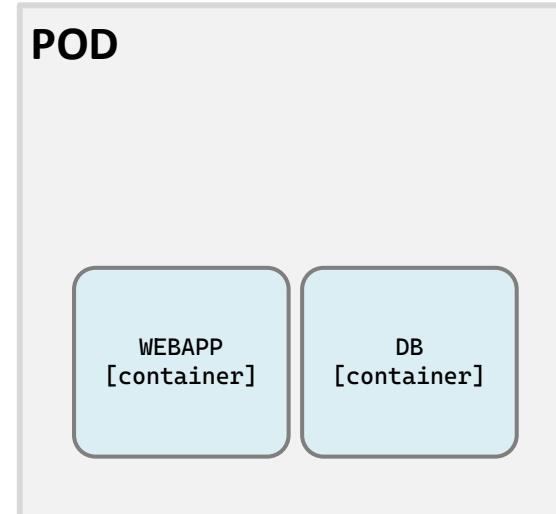


그림 45 멀티 컨테이너

멀티 컨테이너를 운영 시 주의 사항이 있다.

1. PORT 번호는 중복이 안된다. 즉 WEBAPP/DB 는 같은 포트 번호 사용이 불가능하다.
2. 컨테이너의 이름은 중복이 안된다. 즉, 컨테이너의 이름이 WEBAPP/WEBAPP 으로 구성이 불가능하다.
3. 네트워크 및 포트는 POD 를 통해서 공유가 된다. 이러한 이유로 컨테이너 이름이나 혹은 loopback(localhost, 127.0.0.1)으로 컨테이너 애플리케이션에 접근이 가능하다.

위의 조건대로 구성이 되면 다음과 같이 네트워크가 구성이 된다.

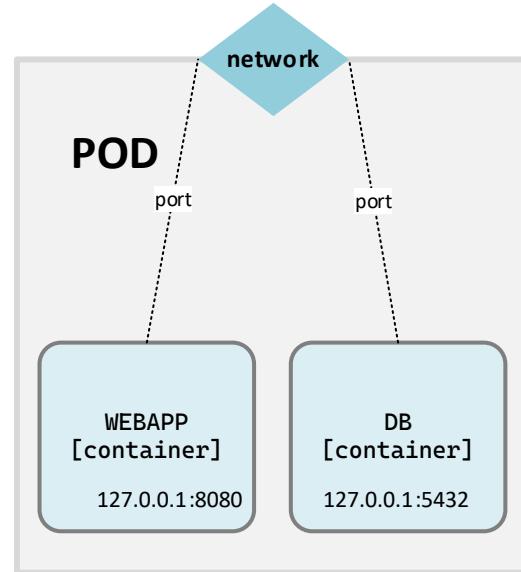


그림 46 멀티 컨테이너 네트워크

다중 컨테이너를 구성하기 위해서 아래와 같이 YAML 파일을 생성 및 구성한다. 파일 이름은 `multi-containers.yaml` 으로 작성한다. 먼저, POD 기반으로 멀티 컨테이너를 생성한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-containers
spec:
  containers:
  - name: php-bg
    image: node1.example.com/testapp/php-blue-green:v1
    ports:
    - containerPort: 80
  - name: php-pgsql
    image: node1.example.com/testapp/php-pgsql:v1
```

```
ports:  
  - containerPort: 5432
```

생성이 완료가 되면, POD 를 생성 및 확인한다.

```
# kubectl create -f multi-containers.yaml  
# kubectl get pod  
# kubectl describe pod multi-containers
```

## 연습문제

3tier 라는 POD 를 생성 후, 아래와 같은 컨테이너를 연결한다. 이미지는 blog, pgsql 를 사용해서 구성한다.

## 4. 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.

```
# kubectl run --image=nginx nginx --port=80
```

앞에서 만든 멀티 컨테이너를 가지고 조회한다. 한 개 이상의 컨테이너에 접근하기 위해서는 -c 옵션(container)를 통해서 특정 접근한다. 단일 컨테이너는 별도의 옵션이 필요 없다.

```
# kubectl get pod  
> nginx  
# kubectl describe -f nginx  
> Containers:  
>   nginx:  
# kubectl exec -it nginx -- bash
```

위의 명령어를 실행하면 POD 밑에서 동작하는 nginx 컨테이너에 접근이 가능하다.

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

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

## 연습문제

1. 기존에 구성이 되어있는 my-httdp 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

## 5. 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.

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

위와 같이 실행하면 POD를 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.

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

## 연습문제

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

## 6. expose/service/endpoint

쿠버네티스에서 구성된 자원은 여러가지 형태로 내부 및 외부에서 네트워크를 통해서 자원이 연결 및 접근이 된다. 쿠버네티스 명령어로 대표적인 자원들을 확인하도록 한다.

### service

서비스는 POD 를 외부에 노출하거나 혹은 클러스터에서 묶는 경우 사용한다. 기본적으로 POD 로 구성된 네트워크는 쿠버네티스 클러스터에서 접근 가능<sup>39</sup>하지만, 외부에서는 POD 네트워크에 접근이 불가능하기 때문에 접근이 가능하도록 서비스 자원을 통해서 연결 및 구성해야 한다.

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 YAML이나 자원은 아래 명령어처럼 생성 및 YAML 파일 생성이 가능하다.

---

<sup>39</sup> 리눅스 브릿지 및 netfilter 기반으로 구성이 되어 있으면 가능하다. 만약, OVN를 사용하는 경우, 경우에 따라서 클러스터 내부에서 접근이 안되는 경우도 있다.

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

노드 포트는 조금 다른 방식으로 서비스 자원에 접근한다.

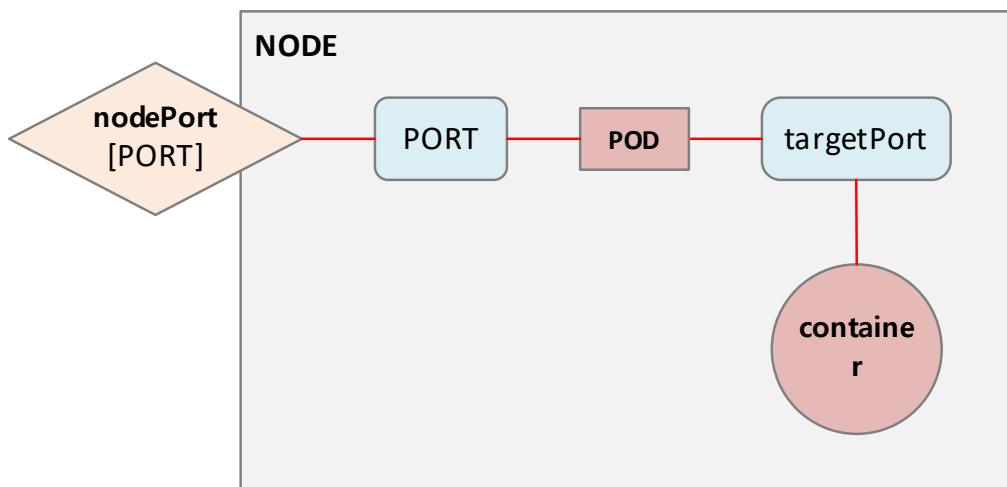


그림 47 노드 포트 구성

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

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

```

모든 게 비슷하나, 생성 시 **--tcp** 옵션을 사용하여 외부포트 8080를 통해서 컨테이너 포트 80으로 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.

```
# 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 service에 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

## expose

expose 명령어는 서비스는 현재 내부에서만 되고 있는 서비스를 외부로 노출하는 명령어이다. 이 명령어는 앞에서 사용한 service와 동일하지만, expose는 이미 구성이 되어 있는 Pod, Deployment 자원을 외부로 노출한다. 쿠버네티스 노드의 네트워크는 외부에서 접근을 하기 위해서 다음과 같은 서비스 유형을 지원하고 있다. 이 자원들은 service에서도 동일하게 사용이 가능하다.

이름	설명	설명
<b>NodePort</b>	오래전부터 지원하던 쿠버네티스 서비스 자원. 호스트에 포트번호를 생성 후, 접근을 허용한다.	
<b>ExternalIP<sup>40</sup></b>	외부에서 사용하는 아이피(공인 아이피)를 할당하여 서비스에 접근을 가능하도록 한다	공식적으로 자원형식으로 제공하지 않음.
<b>Load Balancer</b>	쿠버네티스 로드 밸런서를 통해서 서비스를 구성 및 생성한다	iptables, nftables, 리눅스 배포판에 따라 조금씩 다름
<b>ClusterIP</b>	클러스터 자원이 구성이 되면, 클러스터 아이피를 통해서 자원들이 구성된다. 예를 들어서 endpoint 자원	kubectl get endpoints
<b>ExternalName</b>	외부 도메인 접근이 가능.	도메인 기반으로 서비스를 연결한다.

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

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

혹은 YAML 기반으로 자원을 생성 및 관리가 가능하다. 아래 자원도 별도로 명시한 부분이 없기 때문에 ClusterIP로 노출한다. 파일 이름은 my-nginx-service.yaml으로 작성한다.

```
apiVersion: v1
```

<sup>40</sup> 해당 자원은 서비스 독립 형식으로 존재하지 않는다. 자세한 내용은 이 링크를 확인한다. <https://kubernetes.io/docs/tutorials/stateless-application/expose-external-ip-address/>

```

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

```

port: 80은 외부에서 접근 시 사용하는 포트번호이다. 외부에서 포트번호 80으로 접근하면 deployment 나 혹은 Pod에 구성이 되어 있는 포트 혹은 명시된 서비스 targetPort:으로 접근한다. 위의 예제에서는 targetPort:가 targetPort: 8080으로 되어 있다. 컨테이너에서 사용하는 포트가 한 개 이상인 경우에는 아래와 같이 구성도 가능하다. 파일 이름은 multiport-service.yaml으로 작성한다.

```

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

```

별도로 타입을 명시하지 않는 경우  
ClusterIP로 구성이 된다.

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

좀 더 자세하게 명시하기 위해서 아래와 같이 선언이 가능하다. 보통 형식만 명시하지만, 상황에 따라서 클러스터 아이피를 직접 명시가 가능하다.

```

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
  clusterIP: 10.0.23.24
  type: NodePort
  selector:
    run: my-nginx

```

## endpoints

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

# kubectl get endpoints		AGE
NAME	ENDPOINTS	
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. 하지만, 특별한 경우 개발자 혹은 서비스 운영자는 endpoint slice(종단점 분리)가 가능하다.

엔드 포인트는 다음과 같이 선언이 가능하다. 엔드 포인트에 연결할 POD 의 아이피를 아래와 같이 작성한다. 편의를 위해서 연속적인 아이피 주소로 구성하였다.

```

apiVersion: v1
kind: Endpoints
metadata:
  name: my-service
subsets:

```

```
- addresses:
  - ip: 10.0.0.1
  - ip: 10.0.0.2
ports:
  - port: 80
```

이를 기반으로 다시 노드 혹은 지역별로 분배가 가능하다. 이 기능은 앞서 이야기한 종단점 분리(endpoint slice)를 통해서 구성이 가능하다. 종단점 구성을 위해서 아래와 작성 및 구성이 가능하다.

```
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: my-service-abc123
addressType: IPv4
endpoints:
  - addresses:
    - 10.0.0.1
    - 10.0.0.2
ports:
  - port: 80
```

기존에 사용하던 엔드 포인트와 동일하게 보이지만, 다른 부분은 특정 POD 주소를 묶어서 서비스로 노출이 가능하다. 위에서 이야기하였던 부분은 종합적으로 구성하면 다음과 같이 구성이 가능하다. 먼저, 실험을 하기 위해서 Deployment 자원을 구성한다.

```
# vi endpoint-nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ep-test
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx-container
        image: nginx:latest
# kubectl apply -f endpoint-nginx-deployment.yaml
```

위의 Deployment에서 사용할 서비스를 구성한다. 서비스는 간단하게 NodePort로 구성한다.

```
# vi endpoint-nginx-service.yaml
```

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

위의 서비스가 올바르게 구성이 되었는지 확인한다.

```
# kubectl get pod,deploy
```

구성이 올바르게 되었으면, 엔드 포인트 자원이 서비스가 생성이 되면서 함께 생성 및 구성이 되어 있다. 엔드포인트 구성이 올바르게 되어 있으면, 종단점 분리가 구성이 되어 있다.

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

만약, 특정 지역으로 종단점 구성이 필요한 경우, 다음과 같이 작성이 가능하다. 다만, 이 책에서는 다중 클러스터를 구성하지 않기 때문에, 이 예제는 적용이 되지 않는다.

```
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: cluster2-endpoint-httpd
  labels:
    kubernetes.io/service-name: cluster2-endpoint-httpd
addressType: IPv4
ports:
  - name: http
    protocol: TCP
    port: 80
endpoints:
  - addresses:
      - "10.1.2.3"
      - "10.1.2.4"
      - "10.1.2.5"
    conditions:
      ready: true
    hostname: nginx
    nodeName: node4.example.com
    zone: Daegu
```

## 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.

## 7. proxy, port-forward

### proxy

프록시 서비스는 쿠버네티스에 프록시 서버를 생성 및 구성 후, 로컬/원격 호스트와 쿠버네티스 API 서버와 통신시 사용한다. API 서버 내용을 조회하기 위해서 HTTP를 통해서 접근이 가능하다. 쿠버네티스 API 테스트 이외에는 사용하는 경우는 드물기 때문에, 평상시에는 거의 사용하지 않는 명령어이다. 테스트하기 위해서 두 개의 창 혹은 tmux, screen를 통해서 화면을 나눈다.

```
# kubectl proxy --port=8001
# curl localhost:8001
```

위의 명령어를 실행하면, 쿠버네티스에서 제공하는 API 딕셔너리가 출력이 된다. API 딕셔너리에서 네임스페이스 자원을 curl 명령어로 조회한다.

```
# curl localhost:8080/api/v1/namespaces/kube-system -H "Content-Type: application/json; charset=UTF-8"
{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "kube-system",
    "uid": "d19f8a30-d2d2-4ac8-82bd-1d8011b0d404",
    "resourceVersion": "10",
    "creationTimestamp": "2024-11-11T07:27:34Z",
    "labels": {
      "kubernetes.io/metadata.name": "kube-system"
    },
    "managedFields": [
      {
        "manager": "kube-apiserver",
        "operation": "Update",
        "apiVersion": "v1",
        "time": "2024-11-11T07:27:34Z",
        "fieldsType": "FieldsV1",
        "fieldsV1": {
          "f:metadata": {
            "f:labels": {
              ".": {},
              "f:kubernetes.io/metadata.name": {}
            }
          }
        }
      }
    ]
  }
}
```

```

    }
}

],
},
"spec": {
  "finalizers": [
    "kubernetes"
  ]
},
"status": {
  "phase": "Active"
}

```

이 기능은 자체적인 대시보드 및 모니터링을 구현 시, 많이 사용하며 운영 시에는 종종 디버깅 혹은 확인 용도로 사용한다. 만약, 업무에서 모든 기능을 proxy를 사용해야 되는 경우, 마음속 사표를<sup>41</sup> 지금 표출하는 것도 좋은 방법이다.

## port-forward

포트 포워드는 POD/Deployment 자원을 잠깐 외부나 혹은 내부로 노출한다. 일반적으로 테스트 용도로 많이 사용하며, 실업무에서는 많은 비율로 사용하지 않는다. 사용 방법은 매우 간단하며, 다음과 같은 명령어로 확인이 가능하다.

```
# kubectl run --image=nginx nginx
# kubectl get pod
> nginx
# kubectl port-forward pod/nginx
```

POD에서 사용하는 애플리케이션 포트에 접근이 가능하지만, 기본 값은 기본적으로 클러스터 내부에서만 접근이 가능하다. 외부에서 접근을 원하는 경우 아래와 같이 옵션을 추가를 한다.

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

모든 인터페이스에서 접근을 8080에서 POD 80으로 접근을 허용한다. 특정 인터페이스만 허용을 원하는 경우, "--address" 옵션에 여러 개의 인터페이스 주소를 입력한다.

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

## 8. 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
```

---

<sup>41</sup> 필자는 데이터베이스 및 스토리지를 어마어마한 크기로 날려 먹은 적이 있다. 사직서를 제출하였지만, 대표님이 돈 다 갚을 때까지 퇴사는 일절 없다고 거절을 당했다.

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.

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

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

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

레이블은 POD 이외 다른 자원에도 구성이 가능하다. 예를 들어서 네임스페이스에 아래와 같은 명령어로 레이블 추가가 가능하다.

```
# kubectl label namespaces test type=demo  
# kubectl describe namespace test  
> type=demo
```

위와 같은 방법으로 대다수 자원에 추가가 가능하다. 다만, Deployment 경우에는 주의해서 반영해야 한다. 만약 my-app이라는 애플리케이션이 Deployment로 동작하고 있다고 가정한다.

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

위와 같이 되기 때문에, Deployment에 추가한 레이블이 존재하는 경우, 다시 한번 rollout를 통해서 POD를 재구성해야 한다.

## 연습문제

1. my-httdp에 type=product라는 레이블을 추가
2. 추가 후 describe 명령어로 올바르게 추가가 되었는지 확인

## 9. 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 임으로 변경을 시도를 해본다. 임의로 POD를 nginx 이미지 기반으로 생성 후, 다시 알파인 이미지로 수정한다.

```
# kubectl run --image=nginx my-httdp  
# kubectl get pod  
> my-httdp
```

```
# kubectl set image pod/my-httppd my-httppd=image:alpine
```

변경을 시도하였으나, 잘못된 이미지 이름을 전달하였기 때문에 다음과 같이 오류가 발생한다.

```
Normal  Pulling   16s  kubelet      Pulling image "image:alpine"
Warning Failed   10s  kubelet      Failed to pull image "image:alpine":
reading manifest alpine in quay.io/image: StatusCode: 404, <!DOCTYPE html>
<html lang=en>
<title>404 Not Foun...
Warning Failed   10s  kubelet      Error: ErrImagePull
Normal  BackOff   10s  kubelet      Back-off pulling image "image:alpine"
Warning Failed   10s  kubelet      Error: ImagePullBackOff
```

다시, 올바른 이미지로 변경을 시도한다.

```
# kubectl set image pod/my-httppd my-httppd=docker.io/library/httppd:latest
```

올바른 이미지 주소 정보가 POD에 등록이 되면, POD는 다시 정상적으로 동작한다. 아래 명령어로 확인이 가능하다.

```
# kubectl describe pods/my-httppd
> Image: docker.io/library/httppd:latest
```

위와 같이 정보가 변경이 되면서 정상적으로 POD가 동작이 된다.

## 연습문제

1. my-nginx 의 버전을 httpd:bookwarm 으로 변경한다.

## 10. edit

쿠버네티스에서 작업한 내용들은 YAML 형태로 ETCD에 저장이 된다. 저장된 내용을 기준에 작성한 YAML 파일이나 혹은 ETCD에 저장된 내용을 직접 수정해서 갱신이 가능하다. 이때 사용하는 명령어가 edit 명령어이다.

edit 명령어를 통해서 쿠버네티스에 등록된 설정 파일(정확히는 etcd 데이터베이스)을 YAML 형태로 수정이 가능하다. 이 명령어는 시스템의 EDITOR 나 혹은 KUBE\_EDITOR라는 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.

```
# kubectl edit deployment my-httppd
```

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

```
# KUBE_EDITOR=nano kubectl edit deployment my-httppd
```

혹은 위에서 말한 것처럼, KUBE\_EDITOR를 사용해서 변경이 가능하다.

```
# KUBE_EDITOR=nano kubectl edit deployment my-httppd
```

위의 내용을 영구적으로 저장하려면 bash\_profile에 추가를 해주면 된다.

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

## 연습문제

1. deployment에 구성된 httpd 및 nginx의 replica의 개수를 10 개로 변경 후 상태를 확인
2. 변경 시, edit 명령어를 사용해서 변경

## 11. 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

앞에서 잠깐 이야기 했지만, 대다수의 kubectl 명령어는 객체 및 자원 명시 혹은 기존에 사용하던 YAML 및 JSON 기반으로도 제어가 가능하다. 제거를 실행해보도록 한다. 먼저 POD 를 제거한다. 제거 시 특정 POD 만 제거하도록 selector 를 사용하도록 하겠다. "-l" 옵션은 레이블 옵션이다.

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

service에 있는 자원을 제거해보도록 하겠다.

```
# 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.

```
# kubectl delete pods --all
```

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

```
# kubectl delete all --all
```

제거가 잘 되었는지 확인을 하기 위해서는 다음 명령어로 확인이 가능하다.

```
# kubectl get pods
```

YAML로 구성된 자원은 YAML 기반으로 제거가 가능하다. 사용 방법은 다음과 같다.

```
# kubectl delete -f my-nginx.yaml
```

## 연습문제

기존에 사용하던 모든 자원을 제거한다.

## 12. 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.

```
# 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. 마이너스 표시는 제거, 플러스 표시는 추가 혹은 변경된 내용이다.

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

## 연습문제

- my-httppd의 replicas 개수를 100개로 변경 후 시스템에 등록된 my-httppd 변경사항을 확인하여 올바르게 되었는지 확인한다.

## 13. debug/logs

debug 및 logs는 포드 및 컨테이너 애플리케이션 상태 및 노드 상태 확인 시 사용한다. 기능은 아래와 같이 지원한다.

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

간단하게 디버깅 시도를 한다. 만약, POD 중 my-httppd가 올바르게 동작하지 않는 경우, 아래와 같이 해당 POD에 디버그 컨테이너 연결이 가능하다. 이 기능도 사용하는 저수준 런타임 및 버전에 따라서 지원 여부가 다르다.

```
# kubectl debug -it my-httppd-6796cbbc4c-lr9qg --image=busybox --target=my-httppd
Targeting container "my-httppd". If you don't see processes from this container it may
be because the container runtime doesn't support this feature.

Defaulting debug container name to debugger-dvtvc.
```

--image 옵션은 어떠한 이미지를 통해서 my-httppd POD를 디버깅 할 지 결정한다. 보통은 busybox나 혹은 일반 배포판 이미지를 통해서 많이 접근한다. 보통적으로 디버깅 시, 아래와 같이 메시지가 출력이 된다.

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

또한, 호스트 검사를 위해서 debug를 통해서 노드 접근이 가능하다. 아래와 같은 방법으로 특정 노드에 접근한다. 하지만, 아직 "--privileges" 기능 및 권한은 제공하지 않는다. 아마도, 곧 지원<sup>42</sup>하지 않을까 한다.

<sup>42</sup> <https://github.com/kubernetes/enhancements/issues/1441>

```
# kubectl debug node/node1.example.com -it --image=docker.io/library/busybox:latest
Warning: metadata.name: this is used in the Pod's hostname, which can result in
surprising behavior; a DNS label is recommended: [must not contain dots]
If you don't see a command prompt, try pressing enter
> / #
```

debug 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.

```
# kubectl logs -f my-httppd-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: 'httppd -D FOREGROUND'
```

## 연습문제

1. my-nginx, my-httppd 에서 발생한 로그를 수집해서 확인한다.
2. 노드에 접근 후, 호스트 이름을 확인한다.
3. 모든 권한을 사용하기 위해서 어떠한 방법이 있는지 고려한다.

## 14. 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.

```
# 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
```

## 15. replace

특정 자원을 교체 시 사용하는 명령어이다. 이 명령어는 apply 와 비슷하게 보이지만, apply 는 기존 내용에서 수정 및 간신하는 형식이지만, replace 는 아예 해당 내용으로 제거 후 교체하는 명령어이다. 예를 들어서, 앞에서 label 명령어로 레이블 선언이 가능하지만, replace 기반으로도 선언이 가능하다. 간단하게 YAML 작성 후, replace 를 실행한다.

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

앞에서 만든 replace-nginx-pod.yaml 파일에 다음과 같이 내용을 추가한다.

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

추가가 된 다음에 아래와 같은 명령어를 실행하여 차이점을 확인한다.'kubectl diff'와 비슷하게 화면에 여러 내용이 출력이 된다.

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

확인이 완료가 되었으면, 아래 명령어를 실행하여 POD 간신한다.

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

## 연습문제

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

## 16. patch

사용중인 자원에 일시적으로 특정 내용을 변경하기 위해서 사용하는 명령어다. 앞에서 사용한 edit 와 다른 부분은 patch 는 인라인 형태(inline)으로 자원 설정을 변경한다. 테스트를 위해서 간단하게 POD 를 생성한다.

```
# kubectl run --image=nginx --port=80 patch-nginx-pod  
# kubectl expose pod/patch-nginx-pod --type=NodePort  
# kubectl get svc  
> patch-nginx-pod, NodePort,
```

구성이 완료가 되었으면, 생성된 SVC 의 내용을 NodePort 에서 ExternalIP 로 수정(patch)한다.

```
# kubectl patch svc patch-nginx-pod -p '{"spec": {"externalIPs": ["192.168.0.194"]}}'
```

```
# kubectl get svc  
> patch-nginx-pod, NodePort, 10.90.224.183, 192.168.0.194
```

올바르게 수정이 되면, 위와 같이 ClusterIP 및 ExternalIP 가 명시한 아이피로 출력이 된다. 하지만, 여전히 자원 형식이 NodePort 로 되어있다. 이를 올바르게 LoadBalancer 로 수정한다.

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

## 연습문제

httpd POD 를 생성한다. 생성 후 POD 를 ClusterIP 형식으로 서비스를 노출한다. 이 후, patch 명령어를 통해서 다음과 같이 수정한다.

1. ClusterIP 를 LoadBalancer 으로 변경한다.
2. 외부 아이피는 110.23.4.3 으로 할당한다.
3. 이외 나머지 내용은 그대로 유지한다.

## 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 서비스에 맞게 지속적으로 서비스 운영자가 생성 및 관리를 지속적으로 하였다. 이러한 자원 운영 방식은 매우 번거롭고 또한, 어떠한 자원이 미사용인지 사용 상태인지 확인이 어려웠다. 이러한 이유로 동적 저장소인 StorageClass 를 제공하였다.

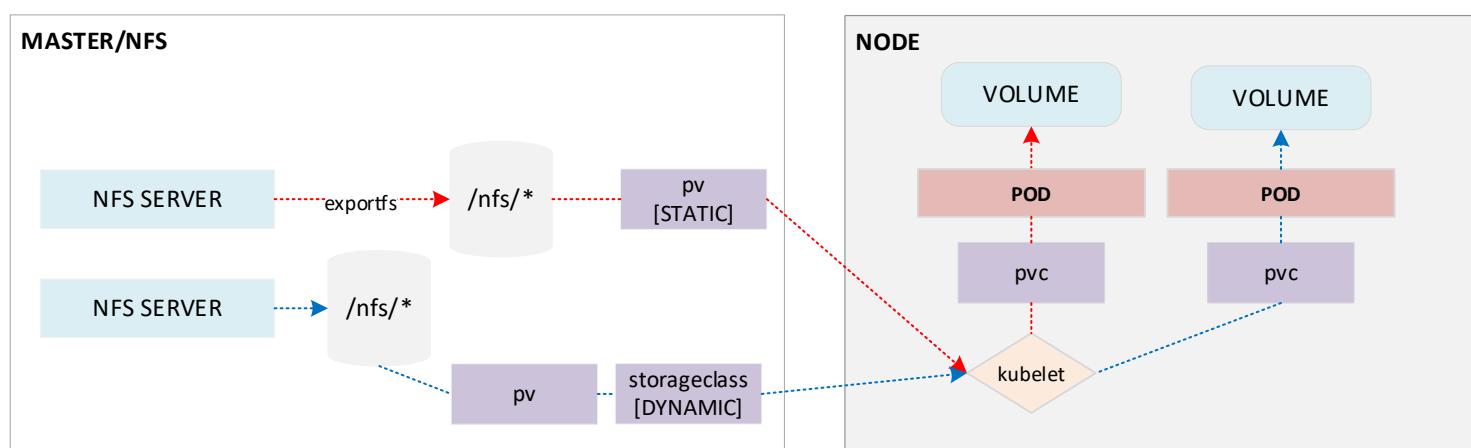


그림 48 PV, PVC, StorageClass

간단하게 NFS 서버<sup>43</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.

```
# curl -skSL https://raw.githubusercontent.com/kubernetes-csi/csi-driver-nfs/v4.4.0/deploy/install-driver.sh | bash -s v4.9.0 --
Installing NFS CSI driver, version: v4.9.0 ...
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.

# kubectl get pods -n kube-system | grep csi
csi-nfs-controller-847cc594cf-
csi-nfs-node-7brlf
```

<sup>43</sup> <https://github.com/kubernetes-csi/csi-driver-nfs/>

## 랩을 위한 NFS 서버 구축

쿠버네티스에서 스토리지를 구현하기 위해서는 nfs 서버를 유ти리티 서버에 구성한다. 스토리지 서버는 유ти리티 서버에 구성한다. NFS 서버는 버전 3이 아닌, 버전 4 기반으로 운영 및 구성할 수 있도록 한다. 로키 리눅스 9 버전 이후부터는 보통 NFS 4 기반으로 구성이 된다.

```
# dnf install nfs-utils -y
```

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

```
# firewall-cmd --add-service nfs  
# firewall-cmd --add-service nfs --permanent
```

디렉토리를 구성한다. 사용할 디렉토리 위치는 /nfs에 구성한다. SELinux가 실행 중이면 중지한다.

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

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

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

exports 파일에 등록된 정보를 nfs server에 갱신한다. 갱신 후 nfs.service를 시작한다. 만약, SELinux를 사용하고 있으면 반드시 아래처럼 컨텍스트 설정을 디렉토리에 한다. SELinux를 사용하지 않으면 semanage, restorecon은 사용하지 않아도 된다.

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

사용할 NFS 디렉토리를 사용하기 위해서는 쿠버네티스 인프라에서 PV, PVC 정보를 등록한다. PV는 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.

## 서비스 계정 생성

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 NFS에 있는 내용을 그대로 가져온 내용이다.

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

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

```

kind: ServiceAccount
apiVersion: v1
metadata:
  name: nfs-pod-provisioner-sa
EOF
# kubectl apply -f storageclass-sa.yaml

```

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

```

# cat <<EOF> storageclass-clusterrole.yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nfs-provisioner-clusterRole
rules:
  - apiGroups: [""]
    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
# 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.

```

# 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
# kubectl apply -f storageclass-clusterrolebind.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 default로 한다.

```
# 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
# kubectl apply -f storageclass-role.yaml
```

최종적으로 사용이 가능한 서비스 계정에 역할 바인딩을 구성한다.

```
# 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
# kubectl apply -f storageclass-rolebinding.yaml
```

## 스토리지 클래스 구성

스토리지 클래스는 쿠버네티스의 Persistent Volume과 비슷하다. 기능적으로 다른 부분은 다음과 같다. 스토리지 클래스는 사용자가 직접 PV를 구성할 필요가 없다. 자동적으로 PV를 구성한다. 먼저, 스토리지 클래스를 구성한다. CSI를 설치 및 구성하지 않으면 올바르게 동작하지 않기 때문에 앞서 진행한 "기본 구축 및 구성"을 꼭 수행한다.

스토리지 클래스 파일을 생성한다. 이름은 storageclass-configure.yaml으로 저장 후 자원을 생성한다.

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

```

server: node1.example.com
share: /nfs
reclaimPolicy: Delete
volumeBindingMode: Immediate
mountOptions:
  - hard
  - nfsvers=4.1
EOF
# kubectl apply -f storageclass-configure.yaml

```

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

```

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

```

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

```

# 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
# kubectl apply -f storageclass-pvc.yaml
# 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, Pod를 생성하여, 실제로 애플리케이션이 볼륨을 올바르게 연결(binding)이 되는지 확인한다. Pod를 생성하기 위해서 아래와 같이 YAML 작성 후, Pod 생성한다.

```

# 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
# kubectl apply -f nfs-csi-pod.yaml

```

올바르게 생성 및 저장소 구성이 되었는지 아래 명령어로 확인한다.

```

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

# 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

# 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.

```

# cd /nfs/
# ls
# touch helloworld.html

```

```
# kubectl exec -it nfs-csi-pod-c6f9dbcfc-qpsnv -- ls /var/www/html/  
helloworld.html
```

위와 같이 출력이 되면, Pod 가 구성이 되면서 올바르게 SC 를 통해서 PVC 가 구성 및 연결이 되었다. SC 가 올바르게 동작이 되었으면 PV 확인하면 다음과 같이 출력이 된다.

```
# 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.

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

동작중인 Pod 에서 아무거나 하나 잡아서 아래 명령어로 확인한다.

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

## 고정 볼륨(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.

```
# cat <<EOF> manual-pv.yaml  
apiVersion: v1  
kind: PersistentVolume  
metadata:  
  name: nfs-pv  
  labels:  
    type: nfs  
spec:  
  storageClassName: ""에 별도  
  이름을 명시하지 않는 경우, 스토리지  
  클래스를 사용하지 않는다.  
storageClassName: ""
```

```

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

```

등록이 문제없이 생성이 완료가 되면 다음과 같이 화면에 출력이 된다.

```

# kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM     STORAGECLASS
REASON    AGE
nfs-pv    1Gi        RWX           Retain          Available   1s

```

여기까지 진행이 완료가 되면, 이젠 PVC를 수동으로 생성해야 한다. 아래 Persistent Volume Claim에서 수동으로 생성한다.

## 연습문제

### 고정 볼륨 요청(Persistent Volume Claim,PVC)

PV로 저장소 드라이버 구성이 완료가 되면, PVC를 생성하여 컨테이너가 사용할 수 있도록 구성한다. PVC는 컨테이너에 오버레이 기반으로 블록 장치를 생성해서 컨테이너에게 전달한다.

```

# cat <<EOF> pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  storageClassName:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
EOF
# kubectl apply -f pvc.yaml

```

등록이 문제없이 완료가 되면 다음과 같이 화면에 출력이 된다.

```

# 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.

```

# 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
# kubectl apply -f pvc-pod.yaml

```

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

```

# 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
...

```

pvc도 올바르게 구성 및 연결이 되었는지 확인한다.

```

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

```

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

```

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

```

## 연습문제

### 연습문제

## 2. helm 패키지

helm은 쿠버네티스에서 사용하는 확장기능 및 사용자 서비스를 패키징이 가능하다. 현재 쿠버네티스에서는 두 가지 형태로 클러스터에 서비스 설치 및 확장을 지원하고 있는데, 그 중하나가 바로 helm이다. helm은 보통 두 가지 역할을 제공한다.

1. 클러스터에 기능 추가
2. 클러스터에 구성할 서비스를 패키지로 구성

다만, 2 번째 경우에는, Chart로 패키지 서비스를 배포하였을 때, 장애가 발생하면 롤백 및 힐링(Healing)과 같은 기능은 제공하지 않는다. 이러한 부분은 쿠버네티스 오퍼레이터(Kubernetes Operator)기반으로 구성을 권장한다. Helm은 Chart를 통해서 설치 및 관리가 가능하며, 리눅스에서 많이 사용하는 RPM, DEB 패키징처럼, 설치/업데이트/삭제와 같은 작업을 할 수 있다. 기본값으로 설치가 되는 서비스가 아니기 때문에, 클러스터 운영자가 직접 설치를 해야 한다.

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

## helm 설치 및 명령어

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

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

앞에서 설치한 인그레스 서비스 Nginx 와 동일한, HAProxy 기반으로 Ingress 서비스를 설치하고 싶으면, 아래와 같은 명령어로 helm 기반으로 설치 및 구성이 가능하다. 현재 HAProxy는 HelmChart 기반 설치를 권장하고 있다. 패키지를 설치하기 위해서 먼저, 저장소를 등록해야 한다. 저장소 주소는 <https://artifacthub.io/>이며, 여기에서 패키지 확인이 가능하다.



## Artifact HUB

### 그림 49 helm 아티펙트 허브

패키지 관리자 명령어가 설치가 완료가 되면, 다음 명령어로 패키지 및 저장소 목록 확인이 가능하다.

```
# helm list  
> NULL  
# helm repo list  
> NULL
```

패키지 설치를 위해서는 저장소를 등록해야 한다. 원하는 패키지를 아티펙트 사이트에서 확인이 가능하다. 검색을 위해서 먼저 다음과 같은 단계를 진행한다.

```
# helm repo update  
# helm repo search hub wordpress  
> wordpress  
# helm repo search repo  
> NULL
```

"search hub"는 원격에서 저장소를 검색, "search repo"는 클러스터에 구성된 정보를 확인한다. 만약, wordpress 를 chart 로 설치하기 위해서 다음과 같이 명령어를 수행한다.

```
# helm repo search hub wordpress  
> wordpress  
# helm repo add wordpress-mariadb https://noor-muradi.github.io/Wordpress/  
# helm repo list  
> wordpress-mariadb https://noor-muradi.github.io/Wordpress/
```

저장소 등록이 완료가 되었으며 "repo list"명령어를 통해서 올바르게 등록이 되었는지 확인했다. 이 저장소 기반으로 wordpress 서비스를 클러스터 구성하도록 한다.

```
# kubectl create namespace helm-test  
# helm install my-Wordpress wordpress-mariadb/Wordpress --version 1.0.2  
# helm list
```

```
> wordpress      default      1          2024-11-15 18:51:21.329228072 +0900
KST deployed      Wordpress-1.0.2 1.0.0
# kubectl get pod -n helm-test
```

위와 같이 HelmChart 기반으로 서비스 배포가 완료가 되었다.

## Chart Package 구성

서비스를 HelmChart 기반으로 배포를 원하는 경우 다음과 같은 단계로 패키징이 가능하다.

1. Chart 디렉터리 생성
2. YAML 작성 및 구성
3. Chart 디렉터리 테스트 및 배포

위와 같은 단계로 패키지를 구성한다. 간단한 Nginx 패키지 배포 패키지를 작성하도록 한다. 생성이 되면 다음과 같은 구조로 디렉터리 및 파일이 생성이 된다.

```
# helm create nginx
# tree -L 3 nginx/
nginx/
├── charts
├── Chart.yaml
└── templates
    ├── deployment.yaml
    ├── _helpers.tpl
    ├── hpa.yaml
    ├── ingress.yaml
    ├── NOTES.txt
    ├── serviceaccount.yaml
    ├── service.yaml
    └── tests
        └── test-connection.yaml
└── values.yaml
```

각 위치에 적절한 자원 설정 및 구성을 하면 된다. 위 디렉터리에 대한 설명은 아래와 같다.

이름	설명
<b>charts/</b>	A directory containing any charts upon which this chart depends.
<b>Chart.yaml</b>	A YAML file containing information about the chart
<b>templates</b>	A directory of templates that, when combined with values, will generate valid Kubernetes manifest files.
<b>values</b>	The default configuration values for this chart

Chart 생성 시, 먼저 Chart.yaml 파일에 패키지 정보에 대해서 명시 및 선언을 해야 한다. 여기에는 보통 패키지 버전, 소프트웨어 버전을 명시한다.

```
# vi Chart.yaml
apiVersion: v2
name: nginx
description: A Helm chart for Kubernetes
```

```
type: application
version: 0.1.0
appVersion: "1.16.0"
```

작성이 완료가 되면, 서비스 배포를 위해서 deployment.yaml, service.yaml, configmap.yaml 를 작성한다. 나머지 파일은 사용하지 않기 때문에 제거하여도 된다. 내용은 아래와 같다.

```
# vi configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  nginx.conf: |
    server {
      listen 80;
      location / {
        proxy_pass $PROXY_URL;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
      }
    }
# vi deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-nginx-proxy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: {{ .Release.Name }}-nginx-proxy
  template:
    metadata:
      labels:
        app: {{ .Release.Name }}-nginx-proxy
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
          env:
            - name: PROXY_URL
              value: {{ .Values.proxyUrl | quote }}
# vi service.yaml
apiVersion: v1
```

```
kind: Service
metadata:
  name: {{ .Release.Name }}-nginx-proxy
spec:
  type: NodePort
  selector:
    app: {{ .Release.Name }}-nginx-proxy
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: {{ .Values.nodePort }}
```

마지막으로 서비스 구성 시, 사용할 값을 values.yaml에 선언한다.

```
# vi values.yaml
PROXY_URL: "http://default.proxy.url"
nodePort: 30000
```

작성된 내용 토대로 HelmChart를 설치를 시도한다.

```
# helm install test-nginx nginx/
# helm list
> test-nginx    default      1
# kubectl get pod
> test-nginx-nginx-proxy
```

위와 같은 방식으로 서비스 애플리케이션 패키징 및 배포가 가능하다.

## 연습문제

### 3. metrics 설치

메트릭 서비스는 쿠버네티스에서 동작하는 Pod 및 Node의 자원 사용 상태를 추적한다. 기본적으로 쿠버네티스는 메트릭 기능을 제공하지 않기 때문에 helm이나 혹은 kubectl 명령어로 설치해야 한다. 미리 구성해둔 YAML 파일 기반으로 메트릭 서비스를 설치한다.

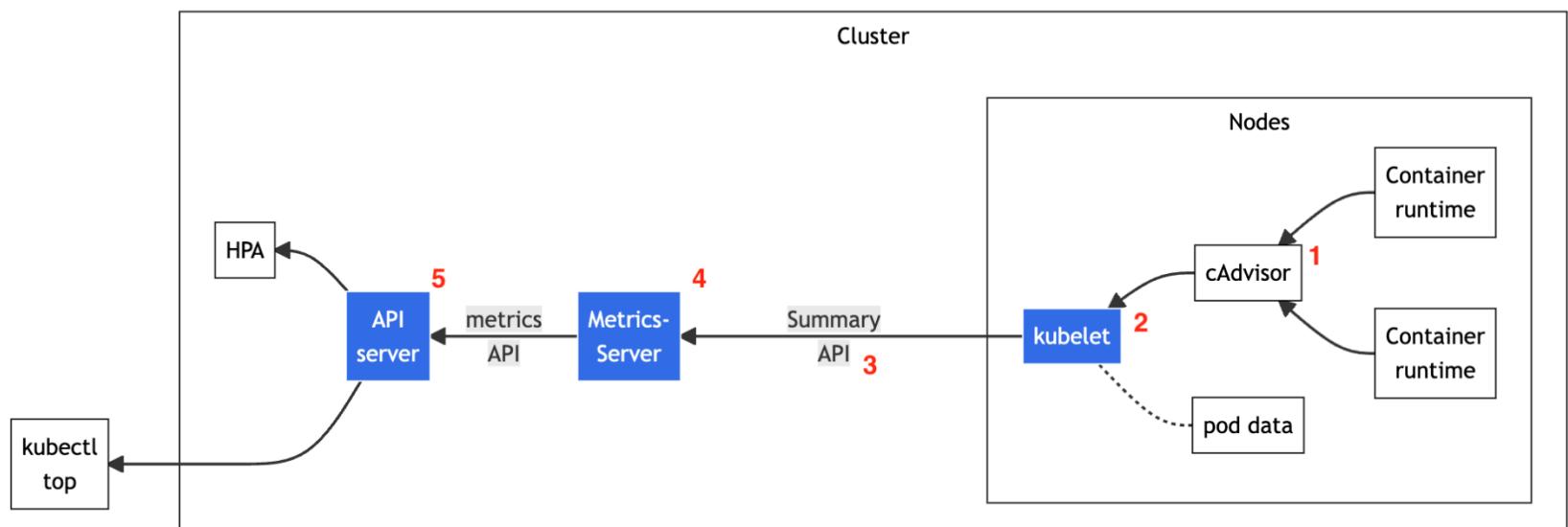


그림 50 메트릭 서버 구조

만약, 스케일링 서비스 구성이 필요한 경우, 반드시 꼭 설치해야 한다. 또한, 이 서비스는 모니터링 용도로는 적절하지 않다.

```
# kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

메트릭 서비스 사이트에 다음과 같이 경고 문구가 있다.

**Caution**

Metrics Server is meant only for autoscaling purposes. For example, don't use it to forward metrics to monitoring solutions, or as a source of monitoring solution metrics. In such cases please collect metrics from Kubelet `/metrics/resource` endpoint directly.

그림 51 모니터링 용도로 사용 금지

이유는 간단하다. 메트릭 서비스는 통계적으로 노드 및 POD 정보를 수집하며, 실시간 정보는 제공하지 않는다. 또한, 구버전을 사용하는 경우 아래처럼 버전에 따라서 설치를 별도로 해야 한다.

쿠버네티스 버전 v1.21+

```
# kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/high-availability-1.21+.yaml
```

쿠버네티스 버전 v1.19-1.21

```
# kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/high-availability.yaml
```

설치는 Helm Chart 기반으로도 구성을 지원한다. 가급적이면 YAML 파일보다는 HelmChart<sup>44</sup>기반 설치를 권장한다.

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

<sup>44</sup> <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.

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

위와 같이 수정하였는데 동작하지 않으면 "--secure-port"를 수정한다. 보통 "4443"으로 변경한다. 최신 버전을 사용하는 경우, 일반적으로 "10250"으로 설정이 되어 있다. 역시 Helm 를 사용한 경우도, 올바르게 동작하지 않으면 "values.yaml"에서 다음과 같이 수정한다.

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

## 사용 및 확인

문제없이 메트릭 서비스가 생성이 되면, kube-system 네임스페이스에서 metrics-server Pod 확인이 가능하다. 아래 명령어로 Pod 생성이 되었는지 확인한다.

```
# kubectl get pods -n kube-system -l k8s-app=metrics-server
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.

```
# kubectl top nodes
master.example.com   56m      2%      1633Mi      45%
```

노드 정보가 확인이 완료가 되면 수집된 Pod 자원 정보를 확인한다.

```
# kubectl top pods
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-ngdf1   0m      2Mi
nfs-csi-pod-c6f9dbcfc-qpsnv   0m      2Mi
pvc-pod                         0m      2Mi
```

위와 같이 준비가 되면, HPA 혹은 VPA 기능 사용이 가능하다. 이 부분은 뒤쪽 컨테이너 자동확장(autoscale)에 대해서 더 다루도록 한다.

## 4. 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.

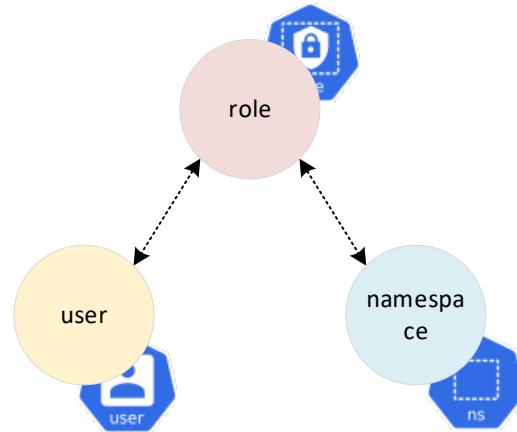


그림 52 네임스페이스 + 사용자 + 역할

네임스페이스(NAMESPACE)는 우리가 익히 알고 있는 것처럼, 자원들이 생성 및 격리가 되는 공간이며(Linux kernel의 **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

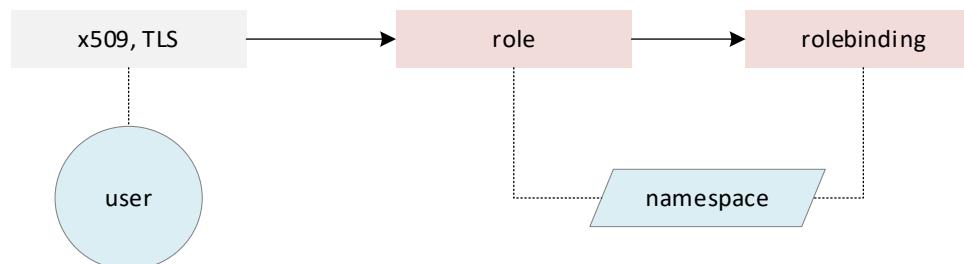


그림 53 사용자 생성방법

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 기능은 아래와 같다.

#### 기본 사용자 인증 조건

1. API Server를 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 흑(Webhooks)기반

## 사용자 생성

먼저 사용자를 생성한다. 생성할 사용자는 다음과 같다.

이름	네임스페이스	권한
openshift	openshift	user
satellite	satellite	user,cluster-admin

위와 같이 쿠버네티스 클러스터에 사용자를 생성한다. 쿠버네티스는 사용자를 RBAC 기반으로 관리하지만, API 서버와 통신을 위해서 암호 방식이 아닌, TLS 키 기반으로 접근을 허용한다. 이러한 이유로 리눅스 시스템에 계정을 직접 만들지 않아도 된다.

먼저 satellite 사용자를 openssl 명령어를 통해서 생성한다. 생성 시, 서명키(요청키) 및 비공개키를 같이 생성한다.

```
# openssl genrsa -out satellite.key 2048
# openssl req -new -key satellite.key -out satellite.csr -subj "/CN=satellite"
```

생성이 완료가 되면 satellite 사용자가 사용할 TLS 키를 구성한다. 구성하기 위해서 쿠버네티스 부트스트랩핑 시 생성된 ca.crt 파일로 서명 후, 공개키를 생성한다.

```
# openssl x509 -req -in satellite.csr -CA /etc/kubernetes/pki/ca.crt -CAkey
/etc/kubernetes/pki/ca.key -CAcreateserial -out satellite.crt -days 500
# ls
satellite.crt  satellite.csr  satellite.key
```

생성이 완료가 되면, 최종적으로 해당 TLS 키를 CSR(Certificate Signing Request)에 등록한다. 등록하기 전, 먼저 생성하나 csr 키를 base64 로 인코딩한다.

```
# cat satellite.csr | base64 | tr -d "\n"
```

인코딩이 완료가 되면, 아래와 같이 YAML 파일을 생성한다. 파일 이름은 으로 등록한다.

```
# vi satellite-csr.yaml
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: satellite
  namespace: satellite
spec:
  request: |
    <인코딩 내용>
  signerName: kubernetes.io/kube-apiserver-client
  expirationSeconds: 8640045
  usages:
```

---

<sup>45</sup>최대 하루 유지

### - client auth

파일 생성이 완료가 되면, 아래와 같은 명령어로 등록 및 확인한다.

```
# kubectl create -f satellite-csr.yaml  
# kubectl get csr  
> satellite, pending
```

문제없이 사용자가 등록이 되었으면, 새로 등록한 csr를 사용할 수 있도록 허용한다.

```
# kubectl certificate approve satellite  
# kubectl get csr  
> satellite, Approved,Issued
```

만약, 사용중인 csr 키를 다시 파일로 가져오려면 다음과 같이 명령어를 실행한다.

```
# 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 kubectl에서 바로 사용이 가능 할 수 있도록 한다.

```
# mkdir .certs  
# mv satellite.crt satellite.key .certs/  
# kubectl config set-credentials satellite \  
--client-certificate=$HOME/.certs/satellite.crt \  
--client-key=$HOME/.certs/satellite.key  
User "satellite" set.  
# kubectl create namespace satellite  
# 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.

```
# kubectl config get-contexts  
*      kubernetes-admin@kubernetes    kubernetes    kubernetes-admin  
      satellite-context                kubernetes    satellite
```

간단하게 사용자를 생성하였다. 현재 사용자는 satellite이며, 클러스터의 이름은 기본 클러스터인 kubernetes 으로 접근하였다. 하지만, 사용자에게 아직 RBAC를 할당하지 않았기 때문에, 적절한 RBAC를 생성 및 할당 후, 클러스터에서 사용할 수 있도록 한다.

## 연습문제

위의 명령어를 사용하여 다음과 같은 사용자를 생성한다.

1. 사용자 이름은 openshift 으로 생성한다.
2. 해당 사용자는 네임스페이스 openshift 를 사용한다.

## 5. 역할구성 그리고 서비스 계정

### 역할(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

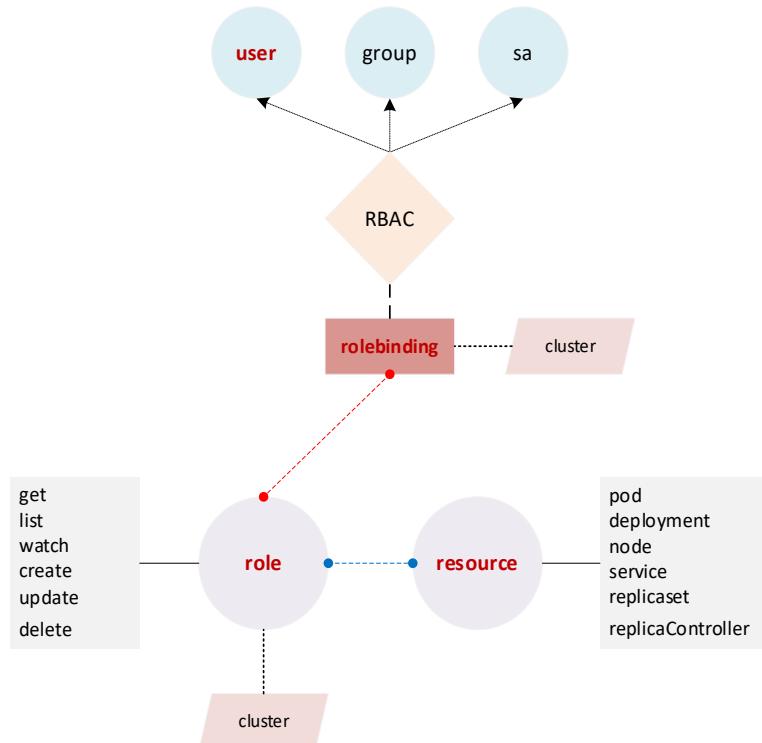


그림 54 역할 및 바인딩

role 은 다음과 같이 분류가 된다.

이름	설명
<b>Role</b>	역할은 권한의 영역을 결정한다. 영역은 rule 기반으로 한다.
<b>Subject</b>	사용자 혹은 그룹을 명시한다. 보통은 RoleBinding에서 사용한다.
<b>RoleBinding</b>	역할 바인딩은 어떠한 사용자가 어떠한 역할을 가지고 특정 네임스페이스에서 사용할지 결정한다.

또한 role 및 rolebinding 은 두 가지로 나누어진다.

이름	설명
<b>clusterRole</b>	클러스터 단위에서 자원 접근. 예를 들어서 node, disk 나 혹은 명시가 되지 않는 클러스터 자원
<b>clusterRolebinding</b>	클러스터 단위에서 접근할 사용자와 네임스페이스를 연결한다.

앞에서 생성한 satellite 사용자에게 역할을 할당한다.

```
# 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. 올바르게 구성이 되었는지 확인한다. 만약, 명령어로 권한 설정이 필요한 경우, 아래와 같이 가능하다. 다만, 아직 rolebinding 이 되지 않았기에 올바르게 권한이 네임스페이스 연결이 안되어 있다.

```
# kubectl describe -f satellite-user-role.yaml
# kubectl create role satellite-role --verb=get --verb=list --verb create --verb
update --verb=watch --resource=pods --namespace satellite
```

일반 역할로 적용하는 경우, 이는 네임스페이스에서만 권한이 허용이 된다. 올바르게 적용이 되었는지, 아래 명령어로 전환 후 확인한다. 아래 명령어는 문제없이 실행이 되나, 바인딩 문제로 satellite 네임스페이스 조회는 되지 않는다.

```
# kubectl config set-context --cluster=kubernetes --user=satellite \
--namespace=satellite --current
```

## 클러스터 역할(ClusterRole)

위에서 일반적인 역할을 생성하였으며, 아래는 클러스터 단위로 역할 구성이 필요한 경우 아래와 같이 수정한다. 랩에서는 굳이 클러스터 권한은 필요 없기 때문에 필요한 경우 진행하지 않아도 된다.

```
# 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"]
# kubectl apply -f satellite-user-clusterrole.yaml
```

위의 내용을 명령어로 생성하면 다음과 같이 생성이 가능하다.

```
# kubectl describe -f satellite-user-clusterrole.yaml
# kubectl create clusterrole satellite-clusterrole --verb=get,list,watch --
resource=pods
```

YAML에서의 명시가 되어있는 지시자를 설명하면 아래와 같다.

이름	설명
apiGroups:	beta, alpha, stable 로 구별한다. 특별한 제한이 없으면 공백.
resources:	접근을 허용할 자원을 적는다. 리스트 형태로 여러 개를 적을 수 있다.
verbs:	자원에 접근 시, 사용이 가능한 명령어. 예를 들어서 get, list, watch 등 하위 명령어

위의 YAML 생성을 명령어로 하면 다음과 같이 구성이 된다. 생성이 완료된 role 은 다음과 같은 명령어로 확인이 가능하다.

```
# kubectl get role --namespace satellite
```

## 역할연결(RoleBinding)

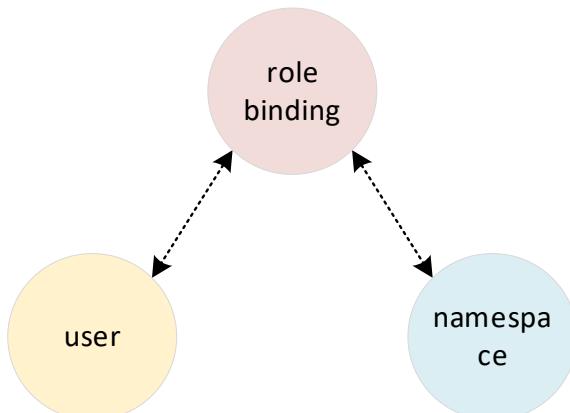


그림 55 역할 연결

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: satellite-rolebinding
  namespace: satellite
subjects:
- kind: User
  name: test-user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: exampleRole
  apiGroup: rbac.authorization.k8s.io
```

여기서 중요한 부분은 subject:, roleRef:이다.  
올바르게 구성이 되지 않는 경우, 사용자는  
네임스페이스 및 자원 접근이 안된다.

위의 YAML 생성을 명령어로 하면 다음과 같이 구성이 된다.

```
# kubectl create rolebinding --role satellite-role --user satellite \
--namespace satellite satellite-rolebinding
```

subjects 밑으로 구성이 되어있는 내용은 다음과 같다.

이름	설명
kind:	사용자는 일반적으로 User 로 한다.
name:	test-user, test-user 에게 퍼미션 구성 및 설정.

roleRef: 밑으로 구성이 되어있는 내용은 다음과 같다.

이름	설명
kind:	역할(role)의 형식. 보통은 role, clusterrole 두 가지.
name:	할당할 역할의 이름.

두 개 이상의 RoleBinding 를 할당하기 위해서 아래처럼 YAML 파일을 작성하였다. 하나는 일반적인 역할, 나머지 하나는 클러스터 역할을 추가하였다.

```
# vi satellite-rolebinding.yaml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: satellite-roleBinding
  namespace: satellite
subjects:
- kind: User
  name: satellite
  apiGroup: rbac.authorization.k8s.io
  namespace: satellite
roleRef:
  kind: role
  name: satellite-role
  apiGroup: rbac.authorization.k8s.io
# kubectl create -f satellite-rolebinding.yaml
```

```
# vi satellite-cluster-rolebinding.yaml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: satellite
  namespace: satellite
subjects:
- kind: User
  name: satellite
  apiGroup: rbac.authorization.k8s.io
  namespace: satellite
roleRef:
  kind: ClusterRole
  name: satellite-clusterrole
  apiGroup: rbac.authorization.k8s.io
# kubectl create -f satellite-rolebinding.yaml
```

YAML 파일 작성 후, kubectl create 명령어로 적용한다. 적용이 완료가 되면 올바르게 동작하는지 빠르게 컨테이너 하나를 실행한다.

```
# kubectl config use-context satellite
# kubectl get pod
# kubectl get pod -n default
# kubectl run httpd --image=httpd
```

현재 클러스터 컨텍스트를 test-namespace 으로 변경 후, 올바르게 권한이 적용이 되었는지 확인한다.

```
# kubectl auth can-i list pods --namespace=satellite
yes
# kubectl auth can-i create pods --namespace=satellite
yes
# kubectl auth can-i delete pods --namespace=satellite
yes
# kubectl auth can-i update pods --namespace=satellite
yes
# kubectl auth can-I list pod --namespace=default
no
```

## kubernetes-admin 대안

마지막으로 kubernetes-admin 사용자는 시스템 구성 후, 사용하면 안 되는 하드코딩으로 구성된 사용자이다. 이 계정을 대신해야 될 계정을 임의로 하나 생성해야 한다. 쿠버네티스 클러스터가 구성이 되면, kubeadm 명령어에서 다음과 같은 권한을 미리 생성한다.

역할이름	역할연결	설명
<b>cluster-admin</b>	system:masters group	Allows super-user access to perform any action on any resource. When used in a ClusterRoleBinding, it gives full control over every resource in the cluster and in all namespaces. When used in a RoleBinding, it gives full control over every resource in the role binding's namespace, including the namespace itself.
<b>admin</b>	None	Allows admin access, intended to be granted within a namespace using a RoleBinding. If used in a RoleBinding, allows read/write access to most resources in a namespace, including the ability to create roles and role bindings within the namespace. This role does not allow write access to resource quota or to the namespace itself. This role also does not allow write access to EndpointSlices (or Endpoints) in clusters created using Kubernetes v1.22+. More information is available in the " <a href="#">"Write Access for EndpointSlices and Endpoints" section</a> ".
<b>edit</b>	None	Allows read/write access to most objects in a namespace. This role does not allow viewing or modifying roles or role bindings. However, this role allows accessing Secrets and running Pods as any ServiceAccount in the namespace, so it can be used to gain the API access levels of any ServiceAccount in the namespace. This role also does not allow write access to EndpointSlices (or Endpoints) in clusters created using Kubernetes v1.22+. More information is available in the " <a href="#">"Write Access for EndpointSlices and Endpoints" section</a> ".
<b>view</b>	None	Allows read-only access to see most objects in a namespace. It does not allow viewing roles or role bindings. This role does not allow viewing Secrets, since reading the contents of Secrets enables access to ServiceAccount credentials in the namespace, which would allow API access as any ServiceAccount in the namespace (a form of privilege escalation).

kubernetes-admin 계정은 보통 cluster-admin 역할을 가지고 있기 때문에, 이를 대신할 계정을 만들고, kubernetes-admin의 권한은 제거한다. 여기서는 그룹에 포함된 사용자만 cluster-admin 권한을 소유한다.

```
# vi cluster-admin.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin
roleRef:
```

```

apiGroup: rbac.authorization.k8s.io
kind: ClusterRole
name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: administrators
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: satellite

```

## 역할기반제어(RBAC) 설명

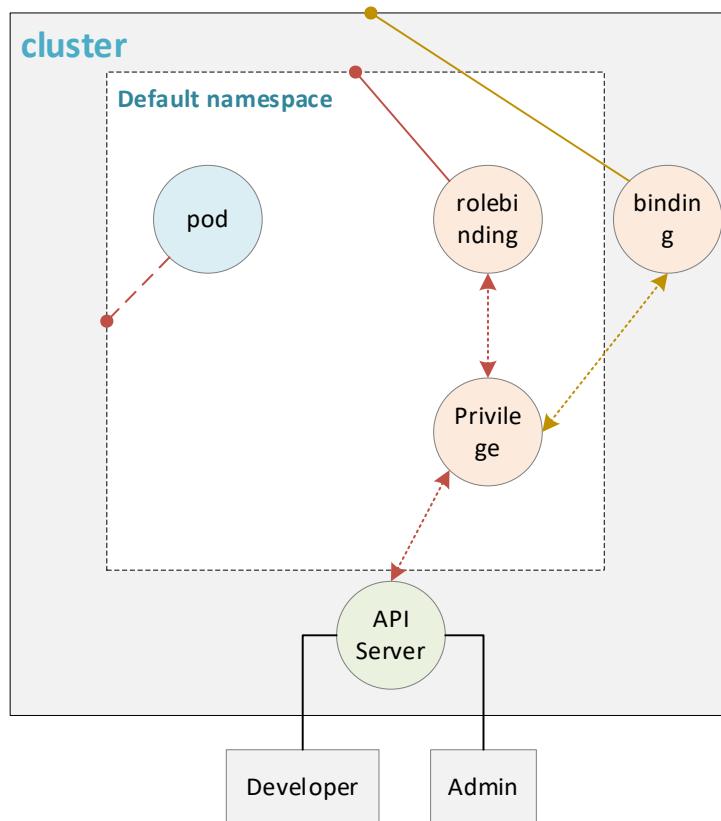


그림 56 RBAC USER, NAMESPACE

위의 role 및 rolebinding 를 사용해서 쿠버네티스 사용자 및 역할을 구성 및 생성하였다. 쿠버네티스는 사용자를 **RBAC(Role-Based Access Control)**기반으로 관리한다는 사실을 확인하였다. 일반적으로 아래와 같은 자원들에 대해서 접근 제어가 가능하다.

- 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 클러스터에 적용할 지, 네임스페이스만 적용할 지 결정해야 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 binding에는 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.

## 6. 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 컨포넌트와 같이 명시하면, 명시된 네임스페이스 안에서 동작하게 된다. 특정, 네임스페이스에서만 사용하기 때문에 쿠버네티스 클러스터에 큰 영향을 주지 않는다.

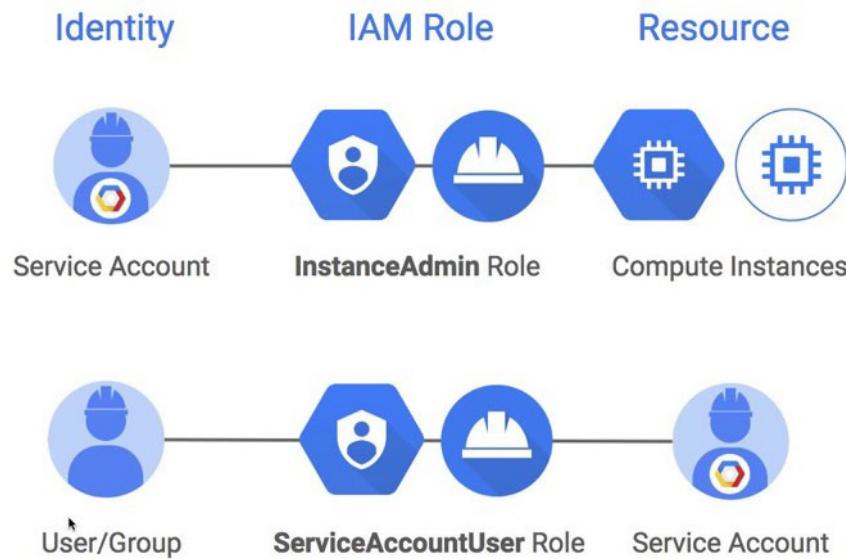


그림 57 서비스 계정 구성

쿠버네티스는 클러스터를 구성하면 기본적으로 다음과 같은 서비스 계정을 가지고 있다.

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

서비스 계정은 명령어로도 생성이 가능하다.

```
# kubectl create serviceaccount serviceaccount-example
```

위의 명령어로 확인이 가능하지만, 서비스 계정은 일종의 메타 정보이기 때문에, 별도로 들어가는 설정은 없다. 명령어나 혹은 파일로 생성하여 사용이 가능하다. 만약, 계정이 일반 계정으로 되어 있는 경우, 쿠버네티스 관리자 계정으로 전환한다.

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

큰 문제가 없으면 아래와 같이 파일이 생성이 된다.

```
# vi test-sa-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: test-sa-pod
    name: test-sa-pod
    namespace: openshift
spec:
  containers:
  - image: nginx
    name: test-sa-pod
```

서비스 계정 test-sa 가, openshift 권한으로 openshift 네임스페이스에 POD 생성이 되어야 한다. satellite 사용자가 openshift 사용자를 호출할 수 있도록 RBAC를 구성한다.

```
# vi test-sa-role.yaml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: test-sa
  namespace: satellite
subjects:
- kind: ServiceAccount
  name: test-sa
  apiGroup: rbac.authorization.k8s.io
  namespace: satellite
roleRef:
  kind: role
  name: openshift-role
  apiGroup: rbac.authorization.k8s.io
# kubectl create -f test-sa-role.yaml
```

자원 생성이 완료가 되었으며, 서비스 계정을 통해서 자원이 생성이 되도록 POD를 수정한다.

```
# vi test-sa-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: test-sa-pod
  name: test-sa-pod
  namespace: satellite
spec:
  serviceAccountName: test-sa
  containers:
  - image: nginx
    name: test-sa-pod
```

수정 후, 계정을 전환한다. 관리자 계정에서는 문제없이 자원이 생성이 되기 때문에 먼저, 계정은 satellite 으로 전환 후 작업을 수행한다.

```
# kubectl config use-context satellite
# kubectl create -f test-sa-pod.yaml
```

## 7. scale/rollout/rollback/history

쿠버네티스에서 동작하는 애플리케이션(컨테이너)를 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.

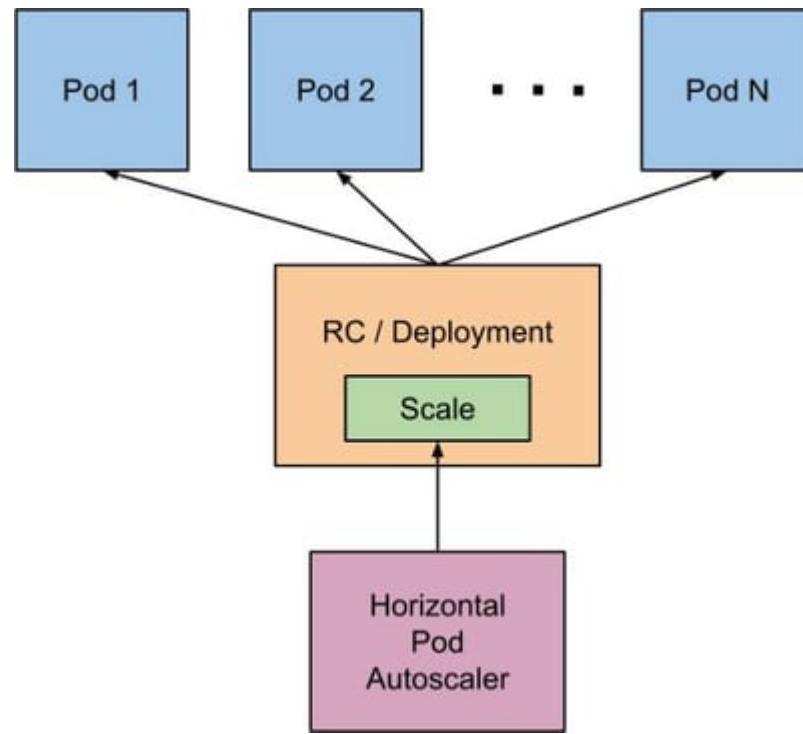


그림 58 스케일링 구조

오토 스케일링인 HPA, VPA는 여기에서 다루지 않으며 scale 명령어에 대해서만 이야기한다. 실험을 하기 위해서 다음과 같은 명령어를 통해서 서비스를 구성한다.

```
# vi 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
# kubectl apply -f nginx-deployment.yaml
```

올바르게 생성이 되는지 kubectl get pods 명령어로 계속 확인을 하며, 또한 생성을 rollout를 통해서 올바르게 되고 있는지 확인이 가능하다.

```
# kubectl rollout status deployment/nginx-deployment
```

생성 확인이 완료가 되면, ReplicaSet(rs)를 확인한다.

```
# kubectl get rs
```

여기까지 완료가 되면 이젠 디플로이먼트를 업데이트하여 서비스를 다시 rollout 해보도록 한다. 쉽게 진행하기 위해서 간단하게 이미지 버전만 변경하도록 한다.

```
# kubectl --record deployment.apps/nginx-deployment \
set image deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1
```

수정이 완료가 되면 룰아웃을 실행한다. 앞에서 사용한 명령어에 특이한 옵션이 하나 "--record"라는 옵션이 보이는데, 기록을 계속 추적 및 남기기 위해서 저 옵션을 사용한다.

```
# kubectl rollout status deployment/nginx-deployment
```

rollout이 진행이 되면 rs나 혹은 deploy에서 확인하면 컨테이너가 프로비저닝 되는 상태가 출력이 된다. 새로 구성이 된 POD를 확인하기 위해서는 kubectl get pods 명령어로 확인이 가능하다.

기본적인 rollout를 사용하였다. 이젠 반대로 다시 rollback 실행을 한다. rollback 우리가 익히 알고 있는 내용처럼, 수행 이전상태로 상태를 변경한다. 단, rollback 가상머신의 롤백 혹은 snapshot처럼 이루어지는 방식이 아니며 모든 자원들이 제거가 된 다음에 재-생성이 되는 구조이다. 이 부분은 꼭 기억해두자. 이전 기준에 구성했던 내용에 일부로 오류를 발생한다. 다음과 같은 명령어로 오류를 발생시킨다.

```
# kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.161 --record=true
```

그리고 다음 명령어로 rollout 상태를 확인한다.

```
# kubectl rollout status deployment/nginx-deployment
```

위의 명령어로 보면 rollout이 여전히 진행이 되고 있지 않으며, get rs나 get pods 명령어로 확인하면 여전히 컨테이너 서비스는 여전히 갱신이 안되고 있지 않는 게 확인이 된다. rollout history 명령어를 통해서 수정사항을 확인 후 다시 롤백한다. 기억하겠지만, 앞에서 우리가 kubectl 명령어를 실행할 때 --record라는 옵션을 사용하였다. 아래 명령어로 rollout 기록을 확인한다.

```
# kubectl rollout history deployment.v1.apps/nginx-deployment
```

확인해보면 옵션을 잘못 넣었다는 사실을 확인할 수 있다.

```
# kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.161 --record=true
```

rollout를 사용해서 이전의 내용으로 다시 롤백을 시도를 한다. 아래 명령어로 "revision 2"번의 내용을 좀 더 자세히 확인한다.

```
# kubectl rollout history deployment.v1.apps/nginx-deployment --revision=2
```

해당 내용이 올바르게 맞으면 다음 명령어로 다시 rollout를 시도한다.

```
# 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.

```
# kubectl scale deployment.v1.apps/nginx-deployment --replicas=10
```

롤 아웃중인 상태를 잠깐 중지하기 위해서 아래와 같이 명령어를 실행한다.

```
# kubectl rollout pause
```

잠깐 중지된 롤 아웃을 다시 수행한다.

```
# kubectl rollout resume
```

history 를 좀 더 쉽게 확인하기 위해서 annotation 을 메타 정보에 남긴다.

```
# kubectl annotate deployment/ghost kubernetes.io/change-cause="kubectl create deploy ghost --image=ghost"
```

## 연습문제

1. my-httdp 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.

## 8. 선호도 및 예외시간(Affinity/Tolerations)

쿠버네티스에서 제일 중요한 자원 POD 자원이다. POD 가 중요한 이유는, 애플리케이션 컨테이너는 POD 에 묶어서 동작하며, 이를 통해서 서비스 클러스터를 구성한다. 이를 위해서 POD 를 어느 노드에 배포를 해야 될지 결정한다. 일반적이 서비스 애플리케이션은 범용적인 노드에서 동작하여도 문제가 없지만, 특정한 기능, 예를 들어서 AI/ML 이나 혹은 고성능이 필요한 디스크 출력 작업 혹은 고성능 메모리 작업이 필요하다.

이러한 부분을 해결하기 위해서 어떠한 노드에서 POD 를 실행할지 선호도를 지정한다. 이 선호도는 두 가지 자원에 대해서 지원하고 있다.

1. POD
2. NODE

NODE 형태로 하는 경우, "어떠한 노드에서 POD 를 구성할 것인가?"를 결정하며, POD 형태로 하는 경우, "POD 가 문제가 발생 시, 어떠한 노드에서 재구성할 것인가?"를 결정한다.

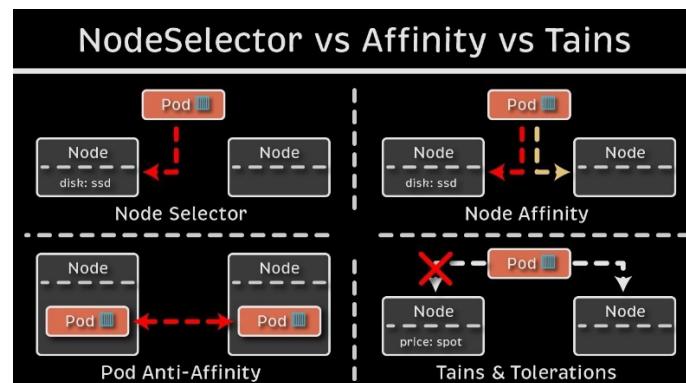


그림 59 포드 및 노드 제어

이 개념을 위해서 Affinity(이하, 선호도)를 통해서 조건을 적용한다. 예를 들어서 특정 POD 를 특정 NODE 에 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)을 통해서 특정 레이블 Pod 를 제한할 수 있다.

## 포드 선호도(Pod Affinity)

POD Affinity(이하, 포드 선호도)는 아래와 같은 방식으로 구성한다. 여기서 구성하려는 선호도 조건은 다음과 같다.

1. 특정 선호 노드에 POD 를 생성한다.
2. 특정 선호 노드에서 장애가 발생하면, 다른 노드로 컨테이너를 생성한다.

포드 선호도는 포드에서 장애가 발생하면, 컨테이너를 마이그레이션하는 게 아니라 새 생성 및 새 구성을 수행한다. 아래와 같이 조건을 구성한다. 만약, 큰 규모로 구성한 랙 경우, 다음과 같이 NODE 에 레이블을 선언한다.

```
# kubectl label node node4.example.com affinityTest=node4
# kubectl describe node/node4.example.com
> Labels:
>   affinityTest: node4
```

위와 같이 레이블 선언 후, 아래와 같이 POD 를 구성한다. 일반 POD 와 차이점은 "affinity:", "podAffinity:" 이 두 가지 옵션이 추가적으로 붙는다. 단일 노드로 사용하는 경우 다음과 같이 적용한다.

```
# vi with-pod-affinity.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: with-pod-affinity
  labels:
    type: with-pod-affinity
    app: wrong
spec:
  selector:
    matchLabels:
      app: wrong
  replicas: 3
  template:
    metadata:
```

```

labels:
  app: wrong
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: location
                operator: In
                values:
                  - korea
            topologyKey: topology.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          podAffinityTerm:
            labelSelector:
              matchExpressions:
                - key: location
                  operator: In
                  values:
                    - japan
            topologyKey : topology.kubernetes.io/zone
  containers:
    - name: with-pod-affinity
      image: registry.k8s.io/pause:2.0
# kubectl apply -f with-pod-affinity.yaml

```

적용 이후, POD는 생성이 되지 않고, 계속 Pending 상태로 동작한다. 이유는 POD 레이블 조건을 찾지 못하고 있다. 아래와 같이 수정 후 적용하면 문제없이 동작한다.

```
# kubectl get pod -l=app=wrong
```

위의 내용을 아래와 같이 수정한다.

```

metadata:
  name: with-pod-affinity
  labels:
    type: with-pod-affinity
    app: wrong -> pause
spec:
  selector:
    matchLabels:
      app: wrong -> pause
  replicas: 3
  template:
    metadata:
      labels:

```

```
app: wrong -> pause
# kubectl delete -f with-pod-affinity.yaml
# kubectl create -f with-pod-affinity.yaml
```

문제가 해결이 되면, POD는 문제없이 생성이 된다. 위의 조건을 설명하면 다음과 같다.

1. topology.kubernetes.io/zone 가 korea, japan으로 선언이 되어있다.
2. NODE LABEL에 location: korea라고 선언이 되어 있는 노드는 POD 생성이 가능하다.
3. NODE LABEL에 location: japan으로 선언이 되어 있는 노드는 POD 생성이 불가능하다.
4. 노드 레이블에 topology.kubernetes.io/zone 선언이 되어 있으면, POD 생성이 가능하다.

위와 같은 조건으로 POD 생성 조건을 걸 수 있다.

## 노드 선호도(Node affinity)

노드 선호도는 POD 자원을 특정 노드에 생성 및 구성이 되도록 한다. 예를 들어서 AI를 위한 애플리케이션이 구성이 되어야 하는데, GPU는 특정 노드에만 설치 및 구성이 되어 있다. 이러한 이유로 어느 노드에서 애플리케이션 배포할지 결정해야 한다.

실험을 하기 위해서 노드에 레이블을 설정한다. node1 번은 AMD, ndoe2 번은 Intel GPU를 가지고 있다고 레이블 명시했다.

```
# kubectl label nodes node1.example.com gpu=amd
# kubectl label nodes node2.example.com gpu=intel
```

POD를 NODE 조건에 맞추어서 구성한다. amd 레이블을 가지고 있는 노드에 POD를 생성한다. 아래와 같이 작성한다.

```
# vi with-affinity-anti-affinity.yaml
apiVersion: v1
kind: Pod
metadata:
  name: with-affinity-anti-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/gpu
            operator: In
            values:
            - intel
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: kubernetes.io/gpu
            operator: In
            values:
            - nvidia
        weight: 50
```

```

preference:
  matchExpressions:
    - key: kubernetes.io/gpu
      operator: In
      values:
        - amd
  containers:
    - name: with-node-affinity
      image: registry.k8s.io/pause:2.0
# kubectl create -f with-affinity-anti-affinity.yaml

```

POD 생성이 시작이 되면, 노드 레이블에 amd라는 이름을 가지고 있는 노드에 가중치(weight)를 주며, 그 다음 가중치로 nvidia 레이블이다. intel이라는 노드 레이블을 가지고 있는 경우, 스케줄링을 무시한다.

## 오염 및 내성 (Taints/Tolerations)

쿠버네티스 클러스터를 운영하다 보면 특정 노드에서 문제가 발생하는 경우가 있다. 이러한 문제가 발생하면, 어떠한 방법으로 해결이 가능한지 확인한다.

특정 노드에 대해서 오염 상태(Taint Status)를 설정하여, 특정 자원 및 작업을 수행하지 못하도록 한다. 일반적으로 컨트롤 노드는 컨테이너 스케줄링이 꺼져 있기 때문에, 아래 명령어로 NoSchedule 선언을 제거가 가능하다. 제거 이후부터는 컨트롤 노드에서 컨테이너 생성 및 관리가 가능하다.

```
# kubectl taint nodes master1.example.com node-role.kubernetes.io/master:NoSchedule-
```

위에서 사용한 taints 와 크게 차이는 없다. 다만, POD 생성 시, Tolerations 조건을 주어서 조건에 맞으면 taint 가 되도록 한다. 아래와 같은 명령어로 스케줄링이 걸리지 않도록 선언이 가능하다.

```
# kubectl taint nodes node1.example.com gpu=nvidia:NoSchedule-
```

다시 스케줄링을 허용하기 위해서 아래와 같이 작성한다. 다시 스케줄링을 허용하기 위해서 제거한다.

```
# 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.

```

# vi toleration.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nvidia
  labels:

```

```

env: gpu
spec:
  containers:
    - name: nvidia
      image: nvda-gpu
      imagePullPolicy: IfNotPresent
  tolerations:
    - key: "gpu"
      operator: "Equal"
      value: "nvidia"
      effect: "Schedule"
# kubectl create -f toleration.yaml

```

오염(taint)이 부분은 뒤에서 더 자세히 다루도록 한다.

## 9. 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 YAML에 작성한다.

```

# 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"
# kubectl apply -f var-demo.yaml
# kubectl get pod -l purpose=demo
# 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

- 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
```

## 10. 컨테이너 자동확장(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. 스케일링은 두 가지 기능이 있다.

이름	설명
HPA (Horizontal Pod Autoscaling)	Pod 를 수평적으로 확장한다. 자원은 확장하지 않으며, POD 개수를 늘린다.
VPA (Vertical Pod Autoscaling)	Pod 를 수직으로 확장한다. 수직으로 확장하는 자원은 CPU, MEMORY 같은 자원이다.

HPA/VPA는 각자의 ReplicaSet를 바라보고 있으며, 이를 통해서 POD를 늘리고 혹은 줄이거나, CPU 및 MEMORY의 크기를 늘리기도 한다.

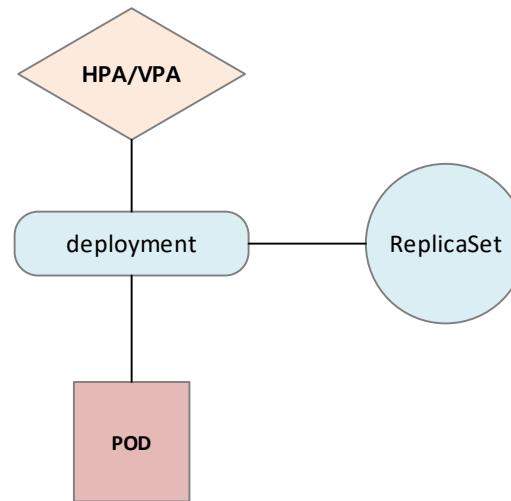


그림 60 HPA/VPA

```
# kubectl get pod -n kube-system -l k8s-app=metrics-server
# kubectl describe pod -n kube-system -l k8s-app=metrics-server
```

앞에서 이야기하였던 scale하고 기능적으로 차이는 없지만, HPA가 특정 조건에 따라서 자동적으로 POD를 확장하다는 차이가 있다. HPA를 사용하기 위해서는 'autoscale'을 사용하겠다고 POD 자원에 선언해야 한다. 아래는 선언하는 명령어 예제이다.

```
# kubectl autoscale deployment.v1.apps/nginx-deployment --min=1 --max=15 --cpu-percent=50
```

### HPA 구성

HPA는 자원 크기를 늘리는 방식이 아니라, 개수를 늘리는 형식을 이야기한다. 마치, 세포가 복제하는 듯한 형태로 동작한다.

# How HPA works?

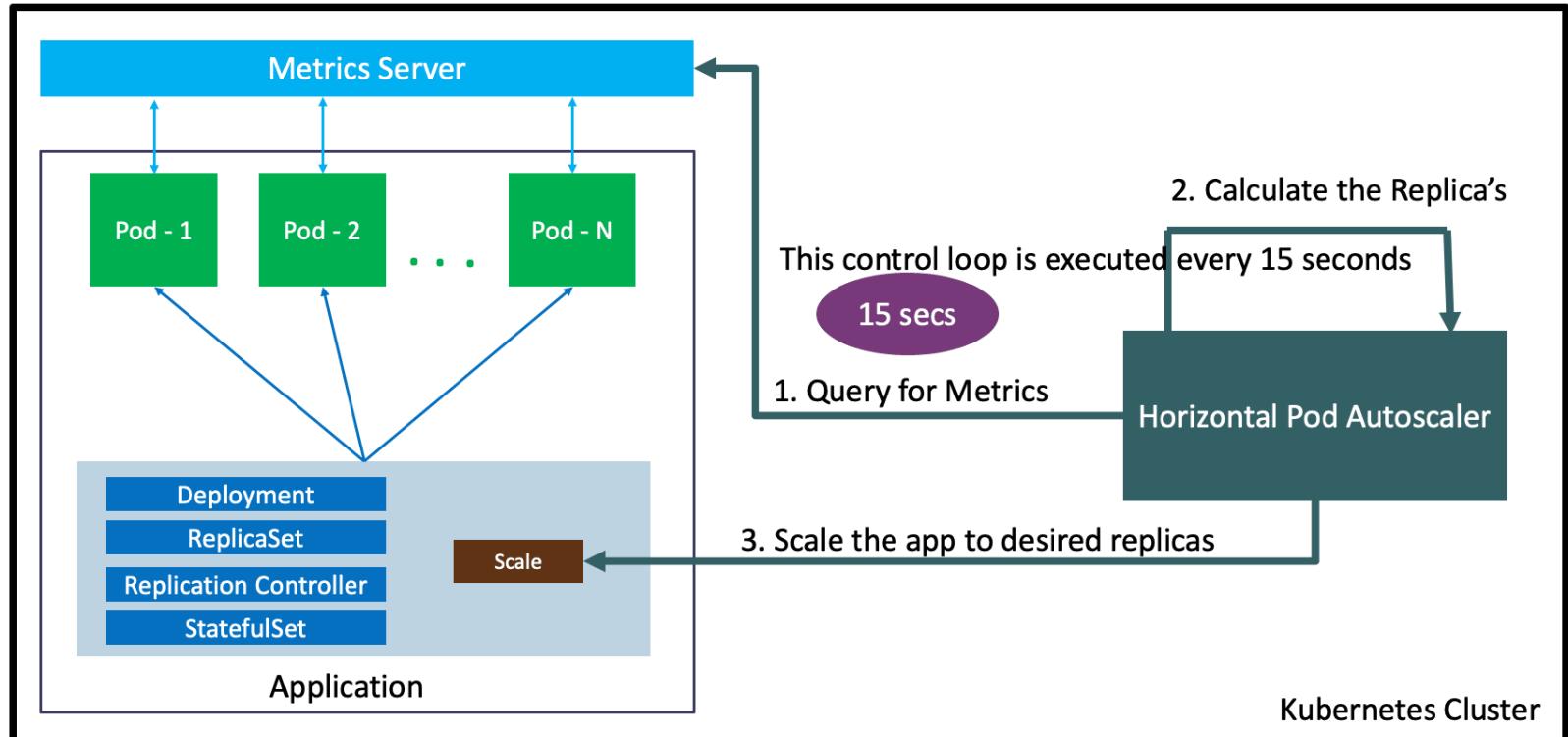


그림 61 HPA 동작 방식

옵션을 간단하게 설명하면 "--min", "--max"는 HPA에서 실행 시, POD 생성 개수이다. "--min=1"이면 시작 시 POD는 1개로 시작한다. "CPU"가 사용율이 "50%"를 넘으면 POD를 수평적으로 확장을 시작한다. 그래서 HPA에 이름에 "H"가 Horizontal이다. HPA 기능을 테스트하기 위해서 Deployment를 작성한다.

```
# vi 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
# 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.

```

# kubectl autoscale deploy/php-apache --min=2 --max=15 --cpu-percent=80
# 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.

```

# kubectl get hpa
# 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.

```

# kubectl get deployment php-apache
php-apache    2/2      2           3m33s

```

만약, 워크로드가 자동으로 올라가지 않는 경우, 워크로드를 아래 명령어로 발생시킨다. 실행 후, 잠시 기다리면 CPU 사용율이 올라가면서 스케일링이 시작이 된다.

```

# kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never --
/bin/sh -c "while sleep 0.01; do wget -q -O- http://php-apache; done"

```

현재 디플로이먼트와 잘되고 있는지 확인한다. 잘 구성되고 동작이 되고 있으면 다음 명령어로 HPA 구성 내용을 확인한다.

```

# kubectl get hpa -o yaml > hpa-php-apache-autoscale.yaml

```

아래와 같이 내용이 나오면 HPA 구성은 문제없이 되었다.

```

# vi hpa-php-apache-autoscale.yaml
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 YAML로 구성하고 싶은 경우, 아래처럼 YAML 파일 작성 후 배포가 가능하다.

```

# vi 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
# kubectl apply -f hpa-yaml-apache-autoscale.yaml

```

## VPA 구성

VPA는 HPA와 반대로 POD의 자원을 상황에 따라서 늘리거나 혹은 줄이는 기능이다. HPA POD의 개수를 통해서 워크로드를 처리하지만, VPA는 전통적인 PaaS 혹은 베어메탈 장비와 동일하게 하드웨어 사양을 확장한다.

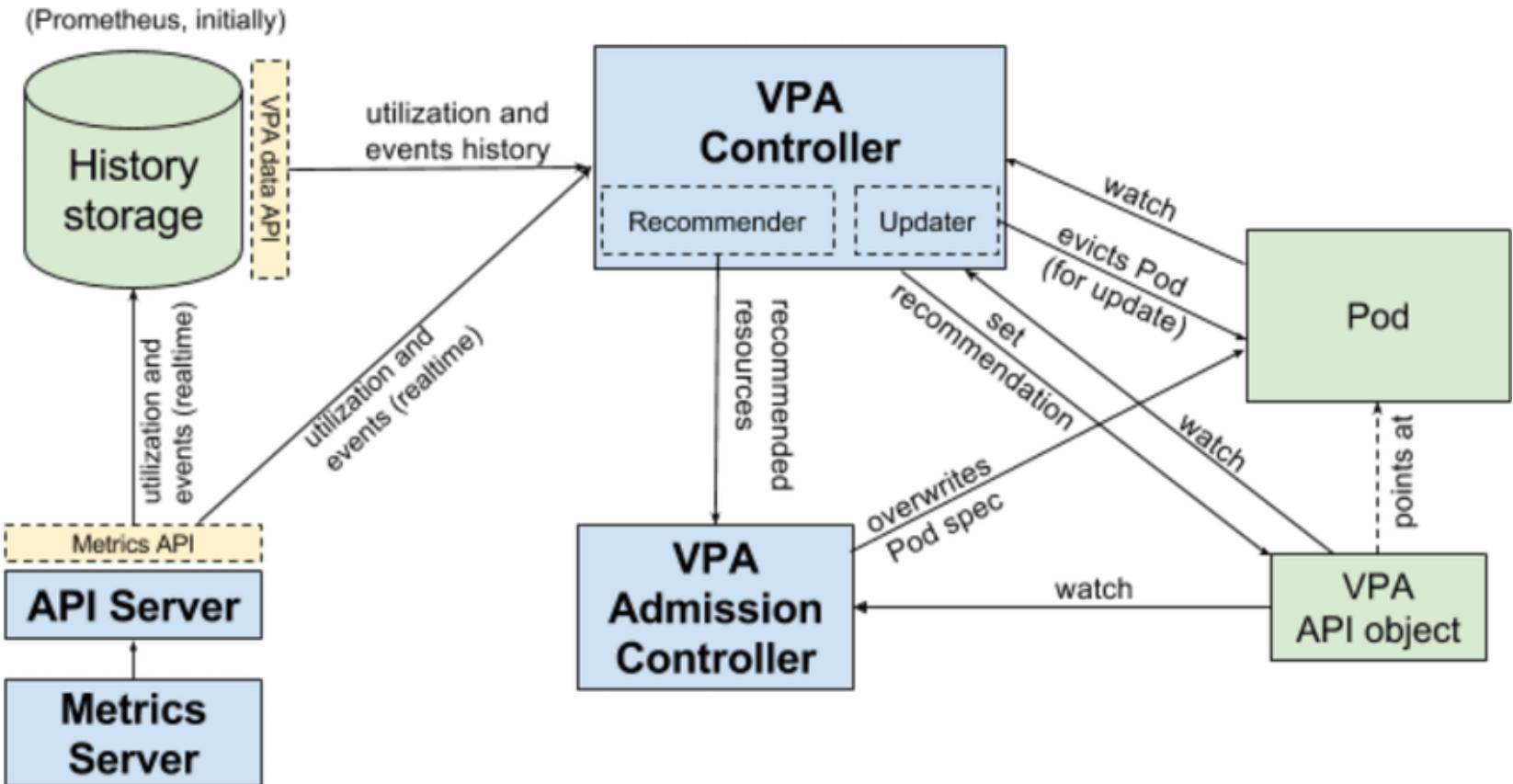


그림 62 HPA 동작 방식

VPA 기능을 활성화하기 위해서는 VPA 기능을 설치한다.

```
# kubectl apply -f https://raw.githubusercontent.com/kubernetes/autoscaler/vpa-release-1.0/vertical-pod-autoscaler/deploy/vpa-v1-crd-gen.yaml
# kubectl apply -f https://raw.githubusercontent.com/kubernetes/autoscaler/vpa-release-1.0/vertical-pod-autoscaler/deploy/vpa-rbac.yaml
```

RBAC 등록 후, GITHUB에서 코드 받은 후, 서비스로 등록한다.

```
# git clone https://github.com/kubernetes/autoscaler.git
# cd autoscaler/vertical-pod-autoscaler
# git checkout origin/vpa-release-1.0
REGISTRY=registry.k8s.io/autoscaling TAG=1.0.0 ./hack/vpa-process-yamls.sh apply
```

문제없이 구성 및 실행이 되면, 다음과 같이 VPA POD 가 실행상태이다.

```
# kubectl get pod -l=app=vpa-recommender -A
```

깃 클론을 받은 디렉터리에 examples 에 예제가 있다. 그 중에서 hamster.yaml 내용을 사용해서 올바르게 동작하는지 확인한다.

```
# vi hamster.yaml
---
apiVersion: "autoscaling.k8s.io/v1"
kind: VerticalPodAutoscaler
metadata:
  name: hamster-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
```

```

name: hamster
resourcePolicy:
  containerPolicies:
    - containerName: '*'
      minAllowed:
        cpu: 100m
        memory: 50Mi
      maxAllowed:
        cpu: 1
        memory: 500Mi
      controlledResources: ["cpu", "memory"]
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hamster
spec:
  selector:
    matchLabels:
      app: hamster
  replicas: 2
  template:
    metadata:
      labels:
        app: hamster
  spec:
    securityContext:
      runAsNonRoot: true
      runAsUser: 65534 # nobody
    containers:
      - name: hamster
        image: registry.k8s.io/ubuntu-slim:0.1
        resources:
          requests:
            cpu: 100m
            memory: 50Mi
        command: ["/bin/sh"]
        args:
          - "-c"
          - "while true; do timeout 0.5s yes >/dev/null; sleep 0.5s; done"

```

POD 가 생성이 되면서 cpu, memory 는 제한된 크기로 시작이 된다. args: 부분에 보면 무한 루프로 yes 명령어를 실행하여 무한으로 yes 메시지를 출력하면서 /dev/null 보내면서 CPU 워크로드는 발생하게 된다. 일정 시간이 지나면 다음과 같이 VPA 자원 사용율이 상승율이 올라가게 된다.

# kubectl get vpa					
NAME	MODE	CPU	MEM	PROVIDED	AGE
hamster-vpa		627m	262144k	True	2m25s

기존 POD는 CPU가 100m으로 약 1/10 코어 사용으로 시작하였으나, 루프가 발생하면서 627m 까지 확장이 되었다. 또한 메모리도 CPU 사용율이 올라가면서 데이터 캐시가 발생하기 때문에 260 메가 정도 사용하고 있다.

## 연습문제

1. 메트릭 서버가 구성이 되어 있는지 확인한다.
2. 구성 후 php-apache의 개수를 최소 5개 최대 20개로 변경한다.
3. cpu 사용율을 30%으로 변경한다.

## 11. 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
```

이 과정을 진행하기 전에, 최소 2개 이상의 노드가 있어야 하며, 한 개의 노드로는 이 기능 구현이 불가능하다. 드레인을 하기 위해서 먼저 노드 이름을 확인한다. 아래 명령어로 노드 이름을 확인한다. 이 과정을 진행하기 전에, 반드시 컴퓨터 역할의 노드가 두 개가 존재해야 한다.

```
# kubectl get nodes
```

먼저, 새로운 컨테이너가 생성되지 않도록 taint를 선언한다. taint는 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 POD를 실행하지 않는다"라고 선언한다. 만약, 마스터 노드만 가지고 있는 경우, taint 명령어는 큰 변경사항이 없다.

```
# kubectl taint nodes node1.example.com:NoSchedule  
# kubectl taint nodes node1 node1.example.com:NoExecute
```

taint가 실행이 되면, 내부에 사용하던 Pod는 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 POD는 다른 노드로 이동이 되었다. 그래도 확실히 제외하기 위해서 한번 drain를 실행한다. (앞으로 drain은 드레인으로 한글로 적겠다)

```
# kubectl drain node1.example.com
```

드레인 실행 후, 노드 상태를 확인한다. 드레인이 완료가 되면, 아래와 같이 노드 상태가 변경이 된다.

Taints:	key1=value1:NoExecute
	key1=value1:NoSchedule

	<code>node.kubernetes.io/unschedulable:NoSchedule</code>
<code>Unschedulable:</code>	<code>true</code>

## 오염 및 차단 (taint/cordon, uncordon)

드레인이 실행이 되면 위의 출력처럼 `Unscheduleable` 이 `true` 로 변경되었다. 이번에는 `cordon` 를 실행한다. 드레인은 비상사태를 선언이라면, `cordon` 는 접근 제한선을 두고, 격리하는 상황과 비슷하다. 드레인은 스케줄링 정책 변경 및 Pod 를 이동하며, `cordon` 은 스케줄링 정책만 변경한다.

이름	설명
<code>cordon</code>	자원은 그대로 있고, 노드에 스케줄링 정책만 변경한다.
<code>drain</code>	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.

```
# 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" "오염됨 상태"로 표시한다.

```
# 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.

```
# 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.

```
# kubectl uncordon node1.example.com
```

위의 명령어를 실행하면 다음과 같이 노드 상태가 변경이 된다.

```
# 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.

```
# kubectl taint nodes node1.example.com key1=value1:Schedule-
# 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 node2에 있는 POD를 전부 node1으로 이전한다.
2. 이전 작업들은 앞에서 이야기하였던 단계로 수행한다.
3. 작업이 완료가 되었으면 다시 노드를 서비스할 수 있도록 스케줄러를 복원 및 복구한다.

## 12. 노드 추가 및 제거(Node add/remove)

완전 최소 노드로 구성한 경우, 컨트롤러+컴퓨트 형태의 단일 노드로 구성하였기 때문에 현재 클러스터 추가는 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.

```
# kubeadm token list
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.com만 구성이 되어 있다. 추가적으로 node2를 워커 노드로 추가하기 위해서는 다시 한번 부트스트랩 시 사용할 토큰을 생성한다.

```
controlX]# 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.com에서 실행한다. 단, 실행 전 "설치 전 O/S 설정"가 완료가 되어야 한다. 위의 명령어를 다시 node2에 실행한다.

```
computeX]# kubeadm join 192.168.10.250:6443 --token 3aey7k.f0k6nv494vhdr1go \
--discovery-token-ca-cert-hash \
sha256:03ac8bd8b173d57881bc70920040731fa0ef37c1b10c4447cd048229e8a35fab
```

위의 명령어가 실행이 되면, 부트 스트랩핑 과정이 실행이 되면서 node2를 마스터 노드를 통해서 클러스터 구성을 한다. 성공적으로 구성이 완료가 되면 아래와 같이 확인이 가능하다.

```
# 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.

```
# kubectl delete node node2.example.com
```

## 연습문제

노드 2 번을 다시 클러스터에 추가한다. 추가하기 전, 노드 2 번은 스냅샷으로 초기화 한다.

11. 설치 전 O/S 설정 진행
12. kubeadm 으로 토큰 생성
13. 토큰으로 생성된 명령어로 노드 2 번 클러스터에 추가

## 13. Service/Cluster HealthCheck

### 상태 확인

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 명령어 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 kubelet에서 Pod를 생성한 애플리케이션 컨테이너 및 Pod 상태를 확인한다. 이 기능을 사용하기 위해서는 POD, 컨테이너에서 특정 URL이 열려 있어야 한다.

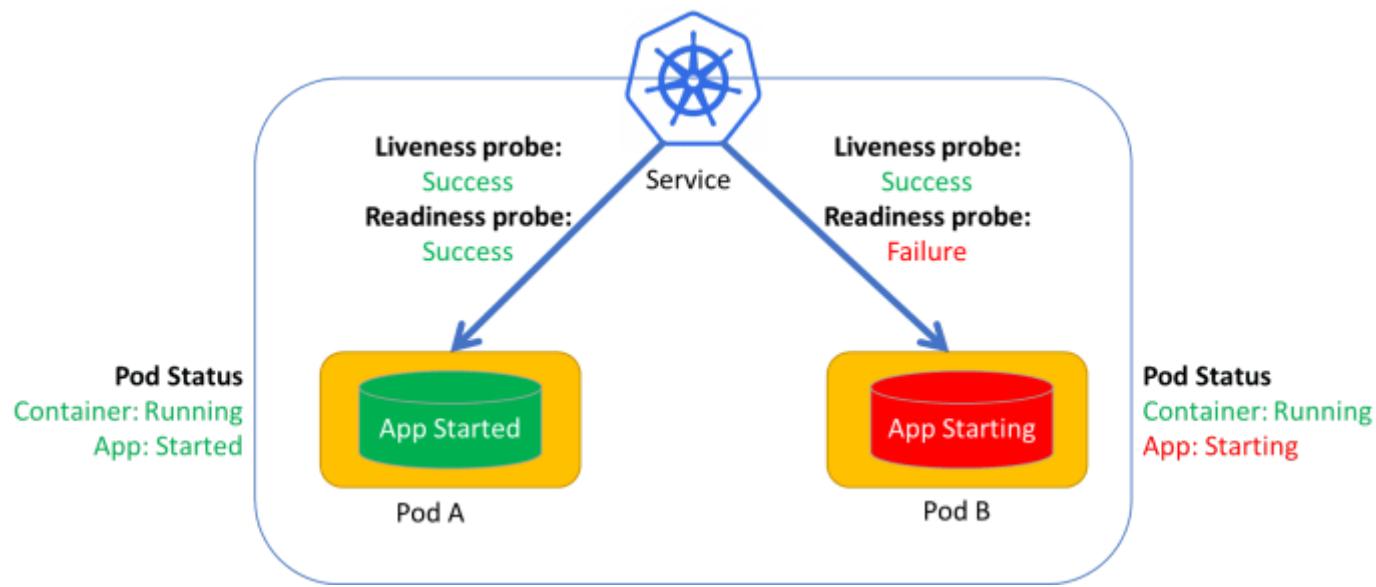


그림 63 쿠버네티스 Readiness, Liveness

liveness, readiness에 대해서 많이 혼돈하는 부분이 있기 때문에 먼저, 내용부터 정리 후 진행한다. 이 두 개의 기능을 아래와 같이 선언 후 진행한다.

이름	설명	위치
<b>liveness</b>	컨테이너에서 사용하는 애플리케이션이 올바르게 동작하는지 확인한다. 올바르게 컨테이너에서 접근이 불가능하면, 비정상으로 판단한다. 정상적으로 동작할 때까지 애플리케이션을 재시작 한다.	kubelet
<b>readiness</b>	컨테이너 및 POD가 올바르게 동작 후, 컨테이너를 통해서 애플리케이션과 통신이 되는지 확인한다. 확인 영역은 서비스(service)를 통해서 조회한다. 올바르게 통신이 되는 경우, 서비스 로드밸런서(service load balancer)에서 제거한다.	kubelet, service
<b>startup</b>	컨테이너 애플리케이션이 올바르게 시작이 되었는지 확인한다. 올바르게 시작이 될 때까지 liveness, readiness는 관여하지 않는다. 이를 통해서 시작 속도가 늦은 POD에 대해서 차단 혹은 재 시작하지 않는다.	kubelet

위의 내용 기준으로 좀 더 자세히 들어가면 다음과 같은 구조를 가지고 동작한다. 이 기능을 사용하기 위해서 POD나 애플리케이션에서 URL 및 자원을 제공해야 한다. liveness를 사용하는 경우 다음과 같이 컨테이너 내부에서 작업을 수행 및 확인한다. 예를 들어 아래와 같은 명령어를 컨테이너 내부에서 실행한다.

```
/bin/sh -c touch /tmp/healthy; sleep 30; rm -f /tmp/healthy; sleep 60
```

위의 명령어를 실행하면 /tmp 디렉터리에 healthy라는 파일이 생성된다. 나머지는 sleep 명령어를 통해서 대기 및 rm 명령어를 통해서 제거한다. 위의 내용으로 liveness를 구성하기 위해서 아래와 같이 POD에 조건을 걸어야 한다.

```

livenessProbe:
  exec:
    command:
      - cat
      - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5

```

"livenessProbe:"를 선언한다. 선언문에 쿠버네티스에서 어떠한 자원을 확인할지 명시한다. command:에서는 cat 명령어를 통해서 /tmp/healthy 파일을 확인하며, 문제가 없으면 RC 값을 0 를 반환한다. initialDelaySeconds:는 5 초 후에 livenessProbe:를 실행, periodSeconds:는 5 초에 한번씩 Probe 를 실행한다. 위와 같은 방법은 HTTP 기반으로 확인하도록 구성이 가능하다.

```

livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
    httpHeaders:
      - name: Custom-Header
        value: Awesome
  initialDelaySeconds: 3
  periodSeconds: 3

```

위와 같은 방법으로 구성하려면, 반드시 컨테이너에서 동작하는 애플리케이션은 해당 path(경로)를 지원 및 제공해야 한다. readiness 기반으로 구현하는 경우, 위와 매우 흡사하다. 코드는 다음과 같이 구성한다. 다만, 차이점은 POD 에서 사용하는 포트번호를 서비스 영역에서 확인이 가능하도록 명시한다.

```

readinessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 15
  periodSeconds: 10

```

앞 liveness 에서 이야기한 내용과 동일하게 시작 15 초 대기 후 매 10 초마다 8080/TCP 포트를 확인한다. liveness, readiness 는 다음과 같은 프로브(probe)형식을 제공한다.

이름	설명
httpGet	HTTP 프로토콜을 통해서 상태를 확인한다. 반환 값은 200/OK 으로 반환이 되어야 한다.
tcpSocket	TCP 포트 기반으로 확인한다.
exec	특정 명령어 실행 혹은 파일을 확인한다. RC 값 반환이 0 이 되어야 한다.

readiness, liveness 는 같이 사용이 가능하며 동작 방식에 대한 우선 순위는 없다. 다만, 구성 시, 각 프로브들이 확인하는 자원의 위치가 다르기 때문에 설정 시 고려해야 한다.

1. readiness 의 "probes"는 컨테이너 시작부터 종료까지 모든 라이프 사이클을 따라간다. readiness 는 애플리케이션이 아닌 POD 를 바라보기 때문에, POD 가 종료가 되면, readiness 도 종료가 된다.
2. liveness 의 "probes"는 readiness probe 가 성공할 때까지 기다리지 않는다. 이유는 liveness 는 애플리케이션 통해서 상태 확인하기 때문이다.

예를 들어서 둘 다 사용하는 경우 다음과 같이 선언한다.

```
readinessProbe:  
  tcpSocket:  
    port: 8080  
  initialDelaySeconds: 15  
  periodSeconds: 10  
  
livenessProbe:  
  tcpSocket:  
    port: 8080  
  initialDelaySeconds: 15  
  periodSeconds: 10
```

이러한 경우, kubelet에서 POD 및 컨테이너를 생성하면서 liveness를 통해서 컨테이너 시작 후 15초 뒤에 애플리케이션 포트를 확인한다. 만약, 포트 확인이 안되면 컨테이너는 재시작 한다. 올바르게 liveness가 시작이 되면, 이후 readiness가 시작이 되며, 컨테이너가 시작 후 15초 뒤에 애플리케이션 포트를 확인한다. 확인이 완료가 되면, POD는 서비스 네트워크를 통해서 트래픽 전달받는다.

## 애플리케이션 상태 확인(Liveness)

"liveness"는 컨테이너에서 사용하는 애플리케이션 서비스의 상태를 확인한다. 쿠버네티스에서 제공하는 표준 예제를 사용해서 다음과 같이 구성한다.

```
# vi liveness-http.yaml  
apiVersion: v1  
kind: Pod  
metadata:  
  labels:  
    test: liveness  
  name: liveness-http  
spec:  
  containers:  
  - name: liveness  
    image: registry.k8s.io/e2e-test-images/agnhost:2.40  
    args:  
    - liveness  
  livenessProbe:  
    httpGet:  
      path: /healthz  
      port: 8080  
      httpHeaders:  
      - name: Custom-Header  
        value: Awesome  
    initialDelaySeconds: 3  
    periodSeconds: 3  
# kubectl create -f liveness-http.yaml
```

올바르게 구성이 되면 다음 명령어로 liveness 조회가 되었는지 확인한다. 올바르게 liveness 프로브가 시작이 되면 아래와 같이 메시지가 출력이 된다.

```
# kubectl describe -f liveness-http.yaml
> Normal  Created     10s    kubelet          Created container liveness
> Normal  Started     10s    kubelet          Started container liveness
```

잠시 기다리면, 문제가 발생이 되며, describe에서 다음과 같이 메시지가 출력이 된다. -w 옵션을 통해서 확인하면, /healthz가 조회가 되면 running, 안되면 CrashLoopBackOff으로 변경이 된다.

```
Liveness: http-get http://:8080/healthz delay=3s timeout=1s period=3s #success=1
#failure=3
# kubectl get pods -l=test=liveness -w
> liveness-http    1/1      Running
> liveness-http    0/1      CrashLoopBackOff
```

이유는 간단하다. 소스코드에서 10초간 200, 10초 이후에는 500를 반환한다. 컨테이너 이미지에서 사용한 소스코드 내용을 잠깐 참고한다. 이러한 이유로 success=1, failure=3 숫자로 표시가 된다.

```
if duration.Seconds() > 10 {
    w.WriteHeader(500)
    w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds())))
} else {
    w.WriteHeader(200)
    w.Write([]byte("ok"))
}
```

마지막으로 TCP 기반으로 사용을 원하는 경우 아래처럼 구성을 한다. 이름은 liveness-tcp.yaml로 저장한다. 아래 POD는 문제없이 정상적으로 동작이 되어야 한다.

```
# vi liveness-tcp.yaml
apiVersion: v1
kind: Pod
metadata:
  name: liveness-tcp
  labels:
    test: liveness-tcp
spec:
  containers:
  - name: redis
    image: redis
    livenessProbe:
      tcpSocket:
        port: 6379
      initialDelaySeconds: 30
      timeoutSeconds: 1
      periodSeconds: 3
    ports:
    - containerPort: 6379# kubectl create -f liveness-tcp.yaml
```

위의 내용은 tcpSocket의 6379 포트를 확인한다. 아래 initialDelaySeconds, timeoutSeconds를 통해서 POD가 시작하는 중 발생하는 대기 시간에 대해서 설정한다. 정상적인 상태를 확인하기 위해서 30초 대기 후, POD 상태를 확인한다.

```
# sleep 30
# kubectl describe -f liveness-tcp.yaml
> Liveness:      tcp-socket :6379 delay=30s timeout=1s period=10s #success=1
#failure=3
# kubectl get pod -l=test=liveness-tcp
```

## 포드 서비스 상태확인(Readiness)

Pod는 시작이 되면서 status 정보를 가지게 되는데, 이 정보를 Readiness 통해서 결정할 수 있다. 즉, 애플리케이션이 올바르게 동작이 되면, Pod는 서비스에 문제가 없다고 판단하여, status를 running으로 변경하며, Readiness를 통해서 조회가 되지 않는 경우, POD는 서비스에 문제가 있다고 판단이 된다.

Readiness 프로브도 "tcp", "http"방식을 제공한다. 아래는 "readiness probe"중에서 "httpGet"기반으로 구성한 예제이다.

```
# vi readiness-http.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: readiness-http
  labels:
    app: nginx
    test: readiness-http
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
# kubectl apply -f readiness-http.yaml
# kubectl describe -f readiness-http.yaml
Readiness: http-get http://:80/ delay=1s timeout=1s period=2s #success=1
#failure=1

```

위의 서비스를 구성한 다음에, `kubectl describ`'명령어를 통해서 Pod 가 올라오면서 어떠한 이벤트가 발생하는지 확인한다. 일반적으로 Readiness 라는 항목에 동작이 올바르게 되었는지 확인이 가능하다. Readiness 는 서비스(service)를 통해서 확인이 가능하다. 올바르게 구성이 되었는지 SVC 를 통해서 프로브 구성한다.

```

# vi readiness-http-svc.yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    test: readiness-http-svc
  name: readiness-http-svc
  namespace: default
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
  - name: readiness-probe-port
    port: 80
  selector:
    app: nginx
  sessionAffinity: None
# kubectl apply -f readiness-http-svc.yaml

```

실행이 될 때 올바르게 'kubectl describe'를 통해서 "Readiness"가 올바르게 동작하는지 확인한다.

```

# kubectl describe -f readiness-http.yaml
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.

```

# 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.

```

# vi liveness-readiness.yaml
apiVersion: v1
kind: Pod
metadata:
  name: liveness-readiness

```

```

labels:
  test: lriness
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
# kubectl create -f liveness-readiness.yaml

```

최종적으로 liveness, readiness 가 잘 구성이 되었는지 확인한다.

```

# kubectl describe -f liveness-readiness.yaml
  Liveness:      tcp-socket :8080 delay=15s timeout=1s period=20s #success=1
#failure=3
  Readiness:     tcp-socket :8080 delay=5s timeout=1s period=10s #success=1
#failure=3

```

마지막으로 readiness 가 올바르게 동작하지 않는 경우 다음과 같은 결과가 예상이 된다.

```

# vi readiness-http-wrong.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: readiness-http-wrong
  labels:
    app: nginx
    test: readiness-http-wrong
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: 8888
# kubectl apply -f readiness-http-wrong.yaml
# kubectl get -f readiness-http-wrong.yaml
> readiness-http-wrong  0/1      1           0          42s
# kubectl describe -f readiness-http-wrong.yaml
Readiness:      http-get http://:80/ delay=1s timeout=1s period=2s #success=1
#failure=1
# kubectl get rs
> readiness-http-wrong-776f877c6f   1      1           0

```

애플리케이션을 올바르게 실행이 되지만, POD 의 8888/TCP 포트로 접근이 되지 않기 때문에, 쿠버네티스에서는 POD 가 올바르게 동작하지 않는 상태로 판단한다. 하지만, 실제로 컨테이너 프로그램은 정상적으로 동작하고 있다.

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.

## 14. 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>46</sup>

레이블은 생성되는 자원에 구분하기 위한 구분자를 구성한다. 구분자(label)은 특별한 기능이 아니라 자원 구별을 위한 메타데이터(metadata)로 사용한다.

---

<sup>46</sup> 한국어로 이름표라고 적으면 매우 어색하다.

이를 통해서 POD, Deployment에 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 YAML이나 혹은 명령어로 선언이 가능한데 어떤 위치에서 사용하느냐 따라서 조금씩 문법이 다르다. 일반적으로 YAML에서는 다음처럼 선언한다

```
keyname: <key value>
```

명령어에서 처리시 일반적으로 다음처럼 명령어를 사용한다

```
# 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.

```
# kubectl create --label="ver=2"
```

혹은 여러 개의 구분자를 명시할 때는 다음과 같은 방법으로 구성한다.

```
# kubectl create --label="ver=2, env=prod"
```

혹은 YAML 파일 작성 시, 직접적으로 레이블 명시가 가능하다.

```
# vi test-label1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    enviroment: production
    app: nginx
spec:
  containers:
  - name: label-nginx
    image: nginx
    ports:
    - containerPort: 80
# kubectl create -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.

```
# kubectl label pods label-demo auth=choigookhyun
# kubectl describe pod label-demo
...
Labels:          app=nginx
auth=choigookhyun
enviroment=production
...
```

위와 같이 이미 사용중인 Pod에 레이블 추가가 가능하다. 다른 자원도 위와 비슷한 방법으로 레이블 추가가 가능하다.

## 연습문제

다음과 같은 자원을 생성 후, 레이블을 추가한다.

1. apache-label라는 Pod를 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 YAML을 작성한다. 위에서 사용한 label를 기반으로 다시 활용해서 확인을 해보도록 한다. 파일 이름은 label-demo.yaml으로 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 YAML에서 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 완료가 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, Kubernetes에서 특정 자원을 생성하면 아래와 같이 annotations를 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 annotation은 레이블과 동일하게, metadata 영역안에 구성이 된다. 여기에는 Pod에서 사용하는 이름인 "annotations-demo", annotations에는 imageregistry에 "https://hub.docker.com"이라고 명시가 되어있다. annotations에 선언이 되어 있는 내용들은 annotations-demo POD와 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 vsftpd를 사용한다
2. 해당 이미지는 service: secureftp라는 annotations를 남긴다
3. 올바르게 구성이 되면 describe를 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.

```
# 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.

```
# 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

# kubectl label node node2.example.com "ssd=true"
node/node2.example.com labeled

# 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.

```
# kubectl get nodes --selector ssd=true
```

위의 내용을 실제 시스템에서 적용하면 다음과 같은 화면이 출력이 된다.

```
# 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

## 15. 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.

```
# vi 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/
```

```
# kubectl apply -f daemonset-node-health-detector.yaml
# kubectl -n kube-system get pods -l k8s-app=node-problem-detector
> node-problem-detector 1/1 Running
```

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.

```
# vi 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
# kubectl apply -f daemonset-vmstat.yaml
# 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
# kubectl logs -f vmstat-log-xxx -n kube-system
```

## 연습문제

모든 서버에 다음과 같이 DaemonSet 를 적용한다.

1. 클러스터의 모든 컴퓨터 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
```

## 16. 상태 POD 서비스 (StatefulSets)

## 17. 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 jobs를 확인하려면 다음 명령어로 확인이 가능하다. 확인을 하기 위해서 Jobs 자원을 하나 생성한다.

```
# vi 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
# kubectl apply -f jobs-test.yaml
```

위의 작업을 명령어로 실행하면 다음과 같이 생성이 가능하다.

```
# kubectl create job pi --image=busybox -- "perl -Mbignum=bpi -wle print bpi(2000)"
```

아래는 job으로 등록된 서비스는 아래와 같이 동작한다. 조금 기다리면 작업 수행이 완료가 된다.

```
# 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.

```
# kubectl describe jobs pi
Name:          pi
Namespace:     np-nginx
Selector:      batch.kubernetes.io/controller-uid=95782465-95f2-4be2-b0a4-
               ad1f39a5a404
               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
# kubectl logs pi-5jbtq
3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628
034825342117067982148...
```

## 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
예제	10	13	10	10	7

위의 내용 기준으로 아래처럼 작성해서 크론잡에 등록한다

```
# vi first-cronjob.yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: first-cronjob
spec:
  schedule: "10 13 10 10 7"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: first-cronjob
              image: busybox
              imagePullPolicy: IfNotPresent
              command:
                - /bin/sh
```

```
- -c  
- date; echo Hello World  
restartPolicy: OnFailure  
# kubectl apply -f first-cronjob.yaml
```

위의 작업을 명령어로 예약이 가능하다.

```
# 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.

```
# 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.

```
# 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  
# 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

## 18. 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 번 째는 쉘 변수로 전달하는 방법, 두 번 째는 파일 시스템 형태로 전달하는 방법이다. 아래 그림은 두 가지 방식을 나타내고 있다.

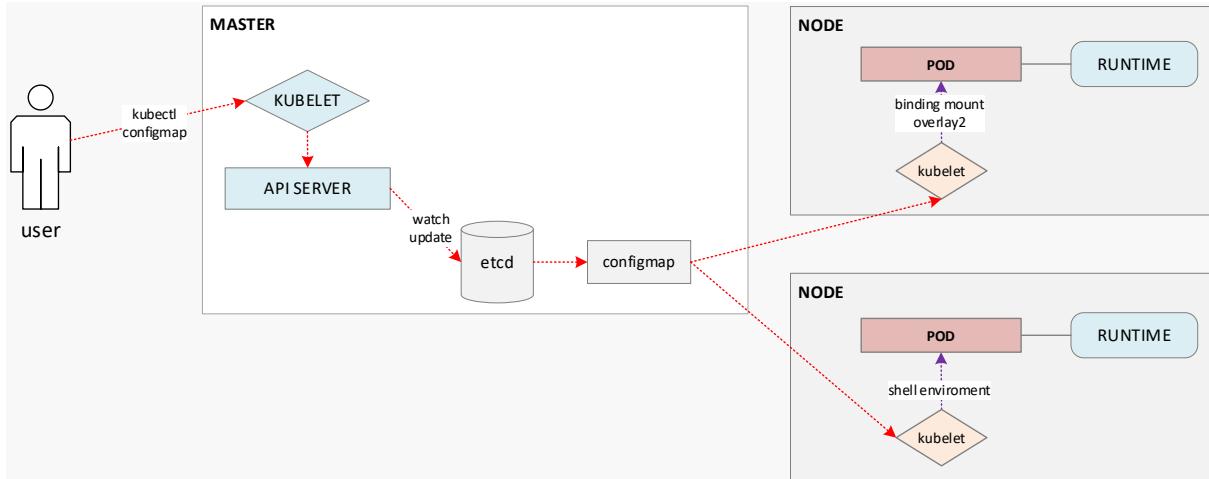


그림 64 configmap 구성

ConfigMaps은 키=값 형태로 구성이 된다. 아래 예제를 확인한다.

```
# kubectl create configmap
```

현재 구성이 되어 있는 configmap 정보를 확인한다. configmap은 실제로 cm라는 약어로 사용이 가능하다. kubectl 명령어로 configmap 객체를 생성한다.

```
# kubectl create cm test
```

위의 명령어를 test라는 configmap 자원객체를 생성한다. 생성한 자원객체는 데이터가 없기 때문에, 명령어에서 자원에 데이터를 입력하려면 다음처럼 명령어를 실행한다.

```
# kubectl create cm test2 --from-literal=name=bora
```

위의 명령어는 configmap에 test2라는 객체를 생성 후, 내부에 name이라는 키 이름을 생성 후 데이터를 bora를 대입한다. kubectl describe 명령어로 확인해본다.

```
# kubectl describe cm test2
```

```
Name:      test2
```

```
Namespace:  default
```

```
Labels:     <none>
```

```
Annotations: <none>
```

```
Data
```

```
====
```

```
name:
```

```
----
```

```
bora
```

```
Events:   <none>
```

또한 데이터를 이미 구성된 정보 기반으로 configmap에 입력이 가능하다. 디렉터리 구조는 상관없으나, 파일이름 및 데이터는 다음과 같은 구조를 따라야 한다. 이 파일은 ui.properties라는 이름으로 생성한다.

```
color.good=purple
color.bad=yellow
```

```
allow.textmode=true  
how.nice.to.look=fairlyNice
```

생성 후 kubectl 명령어로 다음처럼 실행한다.

```
# kubectl create configmap game-config --from-file .
```

명령어 실행 후 kubectl 명령어로 올바르게 데이터가 configmap에 등록이 되어 있는지 확인한다.

```
# 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. configmap를 아래와 같은 이름으로 생성한다. 이 설정 파일은 nginx 이미지에 사용한다.

```
# vi configmap-nginx-test.yaml  
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  
# kubectl create -f configmap-nginx-test.yaml  
# kubectl describe -f configmap-nginx-test.yaml
```

configmap를 사용할 Pod를 생성한다. 이름은 configmap-nginx-pod.yaml으로 저장한다.

```
# vi configmap-nginx-pod.yaml  
kind: Pod  
apiVersion: v1  
metadata:  
  name: env-var
```

```

spec:
  containers:
    - name: env-var-configmap
      image: nginx:1.7.9
      envFrom:
        - configMapRef:
            name: example-configmap
# kubectl create -f configmap-nginx-pod.yaml
# kubectl describe -f configmap-nginx-pod.yaml

```

## 연습문제

다음 내용을 기준 내용에 추가하여 configmap에 등록한다.

1. CentOS=RedHat
2. Ubuntu=Debian
3. Debian=GNU
4. Rocky=CentOS

## 19. 비밀(Secrets(sc))

시크릿은 configmap과 비슷한 기능을 제공한다. 다만, 시크릿 경우에는 configmap처럼 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.

```

# 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 Pod나 kubelet을 통해서 자료를 전달한다. 아래 그림은 기본적으로 시크릿을 전달하는 방법이다.

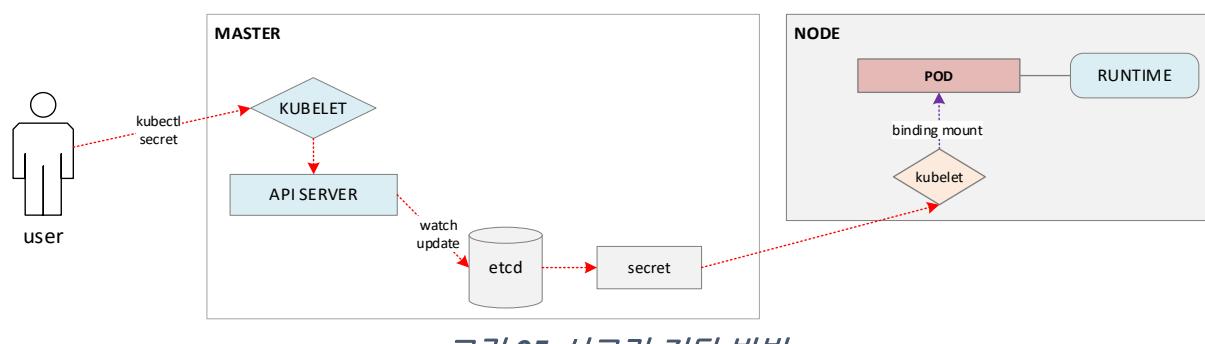


그림 65 시크릿 전달 방법

시크릿은 명령어로 생성이 가능하다. 일반적인 문자로 생성 시, 아래와 같이 사용한다.

```
# kubectl create secret generic name --from-literal=username=choigookhyun
```

위의 명령어를 통해서 username이라는 시크릿 이름을 생성한다. 생성 후 위에서 했던 것처럼, kubectl describe를 통해서 확인한다. 시크릿은 저장 시 인코딩이 되어서 저장이 되기 때문에 내부의 자료는 아래처럼 보이지 않는다.

```
# 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.

```
# echo -n 'redhat' | base64
cmVkaGF0
# 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.yaml으로 저장한다.

```
# echo -n 'openshift' | base64
# echo -n 'redhat' | base64
# vi secret-password.yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-password
type: Opaque
data:
  username: b3BlbnNoaWZ0
  password: cmVkaGF0
# kubectl create -f secret-password.yaml
# kubectl describe -f secret-password.yaml
```

위의 비밀번호는 base64 명령어로 구성이 된 명령어이며, 아이디는 openshift 비밀번호는 redhat으로 되어있다. YAML 형태로 구성된 시크릿을 사용하기 위해서는 kubectl create 명령어로 적용한다.

```
# kubectl create -f secret-password.yaml
```

올바르게 등록이 되었으면, kubectl describe 명령어로 최종 확인한다.

```
# 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.

```
# vi 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
# kubectl create -f secret-pod.yaml
# kubectl get -f secret-pod.yaml
```

올바르게 생성이 되면, 컨테이너 내부에서 "/etc/secret-volume"이 올바르게 구성이 되어 있는지 확인한다.

```
# 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.

```

# ssh-keygen -t rsa -N '' -f test.key
# kubectl create secret generic my-ssh-keys --from-file=test.key --from-file=test.pub
# kubectl describe secret my-ssh-keys
Name:          my-ssh-keys
Namespace:     np-nginx
Labels:        <none>
Annotations:   <none>
Type:          Opaque
Opaque
Data
=====
test.key:  2610 bytes
test.pub:  577 bytes

```

위의 내용 기반으로 Pod를 생성한다. 생성이 완료가 되면, 역시 올바르게 컨테이너에 시크릿 정보가 바인딩이 되었는지 확인한다.

```

# vi 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
# kubectl apply -f secret-ssh-pod.yaml
# kubectl get pods -w
# 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.

14. 로컬 컴퓨터에 임으로 TLS 키를 만든다
15. TLS 키를 컨테이너 서버의 /var/www/html에 전달한다
16. 시크릿의 이름은 secret-nginx-tls으로 구성한다

17. 아파치 웹 서버 패키지(httppd)를 설치한다
18. 두 개 이상의 명령어를 동시에 실행하는 경우 command: `["/bin/sh","-c"]`, args: `["command 1; command 2 && command 3"]` 를 통해서 구성한다.
19. describe, exec 를 통해서 올바르게 바인딩이 되었는지 확인한다.

## 20. 배포 및 배치(deployment)

쿠버네티스 서비스는 구성설정(deployment)을 통해서 구성이 된다. 컨테이너 기반으로 kubectl run 명령어로 간단하게 생성이 가능하지만, 반복적이고 지속적으로 애플리케이션 서비스 구성하기 위해서는 deployment 서비스가 필요하다. deployment 서비스는 replicaSet과 함께 동작한다.

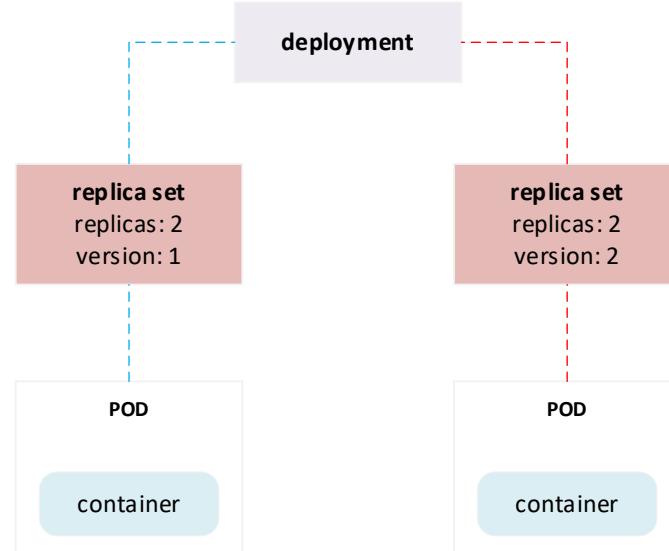


그림 66 디플로이먼트 동작

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 deployment에서 제공하는 포드, 컨테이너 그리고 선택자 및 리플리카(replica) 설정을 YAML 형식으로 하였다.

```

# vi 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
  
```

```
app: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80
# kubectl apply -f nginx-deployment.yaml
```

생성이 되면 올바르게 구성이 되었는지 kubectl 명령어로 확인한다. 생성이 문제없이 되었으면, replicaset 도 같이 확인한다.

```
# kubectl get deployments -o wide
# kubectl get rs
```

올바르게 구성 및 동작이 되고 있으면, 다음 명령어로 이미지 버전을 변경한다. 이미지 정보는 deployment에 등록이 되어 있으니 다음과 같은 방법으로 nginx-deployment에서 사용하는 이미지 버전 정보를 변경한다.

```
# kubectl set image deployment nginx-deployment nginx=nginx:1.8
# kubectl describe pod nginx-deployment-86dcfdf4c6-zxfhl
...
Image:          nginx:1.8
...
```

기존에 사용하던 이미지 1.7.9 버전에서 1.8로 변경한다. 하위 명령어 set를 사용하여 적용되자마자 바로 이미지는 갱신이 된다. 아래 명령어로 올바르게 애플리케이션이 갱신이 되는지 확인한다.

```
# 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.

```
# kubectl rollout history deploy nginx-deployment
deployment.apps/nginx-deployment
REVISION  CHANGE-CAUSE
1        <none>
2        <none>
```

이미지 버전이 1.7.9에서 1.8로 변경되면서 리비전 1, 2 번이 목록으로 출력이 된다. 리비전 정보를 자세히 확인하려면 rollout history 명령어를 통해서 확인이 가능하다.

```
# kubectl rollout history deployment nginx-deployment --revision=1
...
Containers:
nginx:
Image:          nginx:1.14.2
...
# 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.

```
# kubectl rollout undo deployment nginx-deployment  
deployment.apps/nginx-deployment rolled back  
  
# 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

## 21. 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	
비교	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

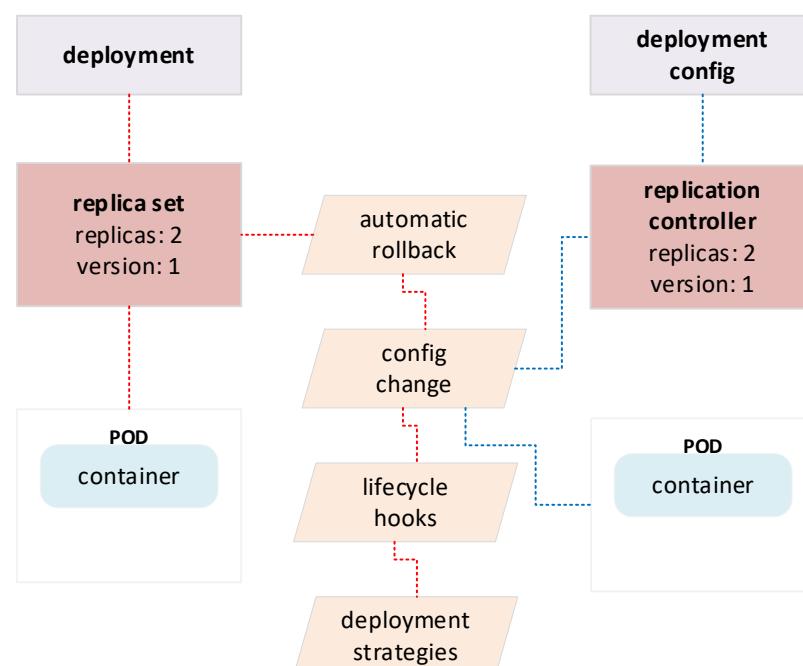


그림 67 replicaSet/ReplicaController

Pod 및 복제자를 동시에 생성한다. 아래와 같이 YAML 파일을 생성한다.

```
# vi 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
  labels:
    app: 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
# kubectl create -f pod-replicaset-test.yaml
# kubectl get -f pod-replicaset-test.yaml
# kubectl get pods,rs -l=app=rs-nginx-lab

```

위의 YAML 파일은 2 가지 자원 내용이 있는데, 첫 번 째는 포드 생성 정보이며, 두 번 째는 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 RC는 RS로 변경을 권장하고 있다.

두 가지 기능의 큰 차이점은 바로 선택자 기능 부분이다. 앞에서 사용하였던 nginx 기반으로 테스트용도로 구성하도록 한다. 아래 YAML은 Replication Controller(rc) 복제자 설정이다.

```
# vi 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
# kubectl create -f replication-controller-example.yaml
```

아래 YAML은 ReplicaSet(rs) 복제자 설정이다. 내용은 앞서 사용한 replicationController와 동일하다.

```
# vi 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
# kubectl apply -f replicaset-example.yaml
# 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

## 22. 외부 아이피 주소(ExternalIP)

외부에서 미리 설정 및 할당한 아이피를 기반으로 클러스터에 접근할 수 있도록 구성한다. 앞에서 이야기 하였지만, ExternalIp 라는 자원은 독립적으로 존재하지는 않는다.

```
# vi 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
# kubectl create -f external-ip.yaml
# kubectl describe -f external-ip.yaml
```

## 연습문제

## 23. 외부이름(ExternalName)

쿠버네티스에서 외부에서 접근하는 방법이 여러가지가 있다. 그중 하나가 ExternalName(외부이름)이다. ExternalName 은 외부에 있는 DNS 서버의 CNAME 레코드를 통해서 접근한다. 서비스에서 expose 명령어를 통해서 외부에 구성이 되어 있는 도메인으로 노출한다. 외부이름은 두 가지 접근 방식을 지원하고 있다.

1. 내부 도메인을 외부 DNS 서버를 통해서 질의

2. 서비스 도메인 통한 접근

3. 외부 DNS 주소 입력

테스트를 위해서 POD 를 생성한다. 기본 POD 를 Nginx 기반으로 생성하며, 기본 서비스를 expose 명령어를 통해서 외부에 노출한다.

```
# kubectl run --labels=app=google-dns --image=nginx google-dns
# kubectl expose pod google-dns --port=80 --target-port=80 --protocol=TCP --
type=NodePort
```

기본 서비스를 나두고, POD 에 외부이름을 사용하기 위해서 아래와 같이 서비스를 구성한다.

```
# vi svc-extname-google-dns.yaml
kind: Service
apiVersion: v1
metadata:
  name: google-dns
spec:
  type: ExternalName
  externalName: google.com
  selector:
    app: google-dns
# kubectl create -f svc-extname-google.dns.yaml
```

아래 명령어로 위의 내용이 올바르게 구성이 되었는지 DNS 도구를 통해서 조회한다.

```
# kubectl exec -it google-dns bash
> apt update
> apt install dnsutils
> nslookup google-dns.default.svc.devops.project
> Server:          10.90.0.10
> Address:         10.90.0.10#53
> google-dns.default.svc.devops.project canonical name = google.com
```

앞에서 생성한 Nginx POD 에서 클러스터 서비스 도메인을 호출하면, 다음과 같이 외부 구글 DNS 서버를 질의하게 된다. "google-dns.default.svc.devops.project" 질의하면 CNAME(Canonical Name)를 통해서 "google.com"으로 전달이 된다.

이와 반대로 서비스 이름을 통해서 조회하려면 다음과 같이 서비스를 구성한다.

```
# vi svc-extname-svcname-dns.yaml
kind: Service
apiVersion: v1
metadata:
  name: google-dns-svcname
spec:
  type: ExternalName
  externalName: google-dns.default.svc.devops.project
  selector:
    app: google-dns
```

위와 같이 구성이 되면, 동일하게 Nginx 컨테이너 내부에서 다음과 같이 실행한다.

```
# kubectl run --labels=app=google-dns --image=nginx google-dns
# kubectl exec -it google-dns bash
> apt update
> apt install dnsutils
> curl google-dns
```

이 기반으로 서비스 클러스터를 통해서 다른 서비스에 접근 시 사용한다. 이 세가지 기반으로 예제를 만들면 다음과 같다. 첫 번째 내용은 google dns 서버를 통해서 질의하는 서비스 구성이다. 서비스가 문제없이 구성이 되면, 다음과 같이 화면에 출력된다.

```
apiVersion: v1
kind: Service
metadata:
  name: google-dns
spec:
  type: ExternalName
  externalName: 8.8.8.8
  ports:
    - port: 53
      protocol: UDP
```

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.

직접 구성하면서, 위의 내용과 비교해보도록 한다. ExternalName 를 구성하기 위해서 쿠버네티스에서 제공하는 ExternalDNS 이미지를 사용하여 구성한다. 아래 내용은 RBAC 구성 없이 ExternalDNS 구성 및 연결하는 과정이다.

```
# vi externaldns.yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: external-dns
  labels:
    name: external-dns
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: external-dns
```

```

namespace: external-dns
spec:
  selector:
    matchLabels:
      app: external-dns
  template:
    metadata:
      labels:
        app: external-dns
  spec:
    containers:
      - name: external-dns
        image: registry.k8s.io/external-dns/external-dns:v0.15.0
        args:
          - --registry=txt
          - --txt-prefix=external-dns-
          - --txt-owner-id=k8s
          - --provider=rfc2136
          - --rfc2136-host=192.168.0.1
          - --rfc2136-port=53
          - --rfc2136-zone=database.example.com
          - --source=ingress
          - --domain-filter=database.example.org

```

bind9 과 같은 온프레미스 도메인을 사용하는 경우, rfc2136 으로 도메인 필터를 설정해야 합니다.

외부 도메인 서비스가 구성이 완료가 되면, 서비스 영역에 다음과 같이 ExternalName 자원을 생성한다.

```

# vi svc-externalName.yaml
kind: Service
apiVersion: v1
metadata:
  name: tfc2135-domain
  annotations:
    external-dns.alpha.kubernetes.io/hostname: database.example.com
spec:
  type: ExternalName
  externalName: database.example.com

```

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

### 24. 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.

```

# kubectl create namespace np-nginx
# kubectl config set-context --current --namespace=np-nginx

```

```
# kubectl create deployment np-nginx --image=nginx
# kubectl expose deploy/nginx --port=80 np-nginx
```

디버깅 컨테이너 하나 실행한다.

```
# 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 selector를 통해서 설정 및 구성한다.
ingress	외부에서 들어온 트래픽이다. 어떠한 자원을 수신할지 selector를 통해서 설정 및 구성한다.

아래는 간단하게 외부에서 nginx 서비스에 접근이 가능하도록 구성한 네트워크 정책이다. Pod에 app=nginx라는 레이블 정보가 있으면, 해당 Pod는 외부에서 접근이 가능하다. 위에서 생성한 nginx 서비스에 접근이 가능하도록 아래처럼 구성 및 설정한다.

```
# vi 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"
# kubectl apply -f network-policy-nginx.yaml
# kubectl get networkpolicy
NAME          POD-SELECTOR   AGE
access-nginx  app=nginx     9s
```

테스트를 하기 위해서 위에서 실행하였던 busybox를 한 번 더 실행한다. 그리고 동일한 명령어를 컨테이너 내부에서 실행한다. busybox가 접근 시 nginx Pod에 접근하기 위해서 레이블 labels="access=true"를 추가한다. 추가가 된 busybox 컨테이너는 문제없이 nginx Pod를 통해서 컨테이너에 접근이 가능하다.

```
# 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.

```
# vi 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"
```

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

# 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. 도메인 서버

쿠버네티스에서 구성되는 서비스는 대다수가 도메인 기반으로 구성이 된다. POD를 구성하는 서비스 경우도, VIP 아이피 주소와 그리고 서비스 도메인으로 구성이 된다. 이와 비슷하게 인프라부분에서도 외부 사용자가 쿠버네티스에 구성된 서비스에 접근하기 위해서 도메인 서비스가 필요하다.

### 설치 및 구성

DNS 서버는 배포판에 따라서 다르기는 하지만, 보통 패키지 이름이 bind 혹은 bind9 으로 되어 있다. bind9 이외 다른 도메인 서비스가 많이 있다. 예를 들어서 쿠버네티스에서 사용하는 coredns 도 사용이 가능하며, 혹은 simpleDNS 와 같은 애플리케이션을 사용하도 된다. 빠르게 사용이 가능한 CoreDNS 기반으로 다음과 같은 도메인을 구성한다.

- example.com
- demo.io

이 두 개의 도메인 기반으로 인그레스 및 서비스 도메인에서 사용이 가능하도록 한다. 먼저, CoreDNS 이미지를 내려받기 한다.

```
# podman pull podman pull docker.io/coredns/coredns
```

컨테이너 실행 전, 사용할 파일을 설정 파일을 미리 구성한다. 먼저, systemd에서 DNS 서버를 내부에서 사용이 가능하도록 systemd-resolved 패키지를 설치한다. 이 기능이 설치가 되면, 이전에 사용하던 resolv.conf 파일은 더 이상 필요가 없다.

```
# dnf install systemd-resolved -y
# vi /etc/systemd/resolved.conf
DNS=127.0.0.1 8.8.8.8
FallbackDNS=
Domains=~
DNSStubListener=no
# rm -f /etc/resolv.conf
# systemctl enable --now systemd-resolved
```

설치가 완료가 되면, coredns에서 사용할 도메인 설정한다.

```
# mkdir coredns-config && cd coredns-config
# vi Corefile
example.com:53 {
    file /coredns-config/example.db
    log
    errors
}
```

```

demo.io:53 {
    file /coredns-config/demo.db
    log
    errors
}

.:53 {
    log
    errors
    forward . 8.8.8.8
    cache

```

example.com, demo.io에서 사용할 존 파일(zonefile)를 아래와 같이 생성한다.

```

# vi example.db
$TTL 3600
$ORIGIN demo.io.

@ IN SOA ns.demo.io. contact.demo.io. (
    2019121301 ; serial
    1d ; refresh
    2h ; retry
    4w ; expire
    1h ; nxdomain ttl
)
    IN NS ns.demo.io.

@ IN A 192.168.0.10
* IN A 192.168.0.10
@ IN A 10.10.0.10
* IN A 10.10.0.10

# vi demo.db
$TTL 3600
$ORIGIN demo.io.

@ IN SOA ns.demo.io. contact.demo.io. (
    2019121301 ; serial
    1d ; refresh
    2h ; retry
    4w ; expire
    1h ; nxdomain ttl
)
    IN NS ns.demo.io.

@ IN A 192.168.0.10
* IN A 192.168.0.10
@ IN A 10.10.0.10
* IN A 10.10.0.10

```

구성이 완료가 되면 서비스 및 컨테이너 실행, 그리고 도메인 확인을 한다.

```
# systemctl status systemd-resolved
```

```

> active
# podman run -d --rm -p 53:53/udp --name coredns -v /root/coredns-config:/coredns-
config coredns -conf coredns-config/Corefile
# podman container ls
> coredns
# dig @192.168.0.10 demo.io A
> demo.io.          0      IN      A      192.168.0.10
# dig @192.168.0.10 example.com A
> example.com.      0      IN      A      192.168.0.10

```

도메인 준비는 완료가 되었으며, 인그레스 및 서비스 테스트가 가능하다.

## 1. 로드 밸런서

3 Tier 기반의 시스템에서 많이 적용되는 부분이 바로 로드밸런서 장치 혹은 소프트웨어이다. 쿠버네티스는(은) 로드밸런서 기능을 제공하지만, 초기에는 이를 구현하기 위해서 외부에서 장치로 구성하던지 혹은 HAProxy 나 Nginx 와 같은 소프트웨어 기반으로 구성하였다. 하지만, 이 방식을 쿠버네티스와 통합에는 적절하지 않기 때문에 외부에서 구성하였던 로드밸런서를 쿠버네티스 클러스터 내부로 통합을 시작했다. 통합에는 두 가지 방법이 있다.

- 하드웨어 기반 로드밸런서
- 소프트웨어 기반 로드 밸런서

기본적으로 하드웨어 방식은 말 그대로, 물리적인 L4 혹은 L7 장비를 구성 후, 로드밸런서 전달받는 형식이다. 대표적인 장비는 시스코 혹은 Radware(Alteon)과 같은 회사가 있다. 랩에서는 위의 장비는 다루지는 못한다.

소프트웨어 방식은 앞서 이야기하였던 HAProxy, Nginx 혹은 다른 소프트웨어 기반으로 구현하는 경우이다. 또한, 쿠버네티스는 통합이 가능한 로드밸런서 소프트웨어를 CNCF 기반으로 개발을 시작하였는데, 대표적인 소프트웨어는 바로 MetalLB 이다. 쿠버네티스 서비스 자원에서 제공하는 "로드 밸런서 형식"은(은) 자원이 구성이 되면 기본적으로 "pending"상태로 구성이 된다. 이러한 이유로, "Load Balancer"형식의 자원에는 외부에서 접근이 가능한 인터페이스의 아이피 혹은 로드밸런서 서버나 장비의 아이피 주소를 입력한다.

클러스터에 MetalLB 가 구성이 되면, Speacker 가 구성이 된 노드, 일반적으로 컨트롤러에서 동작하면서 외부에서 요청하는 아이피를 확인 후, ARP 기반으로 로드밸런서 구성 및 아이피를 할당한다. 아래는 간단하게 MetalLB 동작 방식이다.

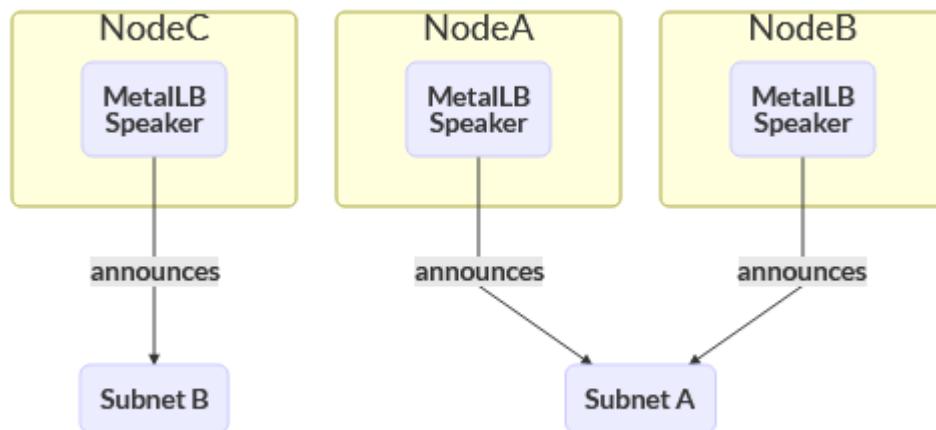


그림 68 쿠버네티스 로드밸런서

아래는 로드밸런서 형식으로 구성된 로드 밸런서 서비스가 외부나 혹은 내부에 구성이 안되어 있으면 아래와 같은 상태이다.

```
# kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
testlb	LoadBalancer	10.90.18.27	pending	80:31973/TCP	6s

로드 밸런서가 내부나 혹은 내부에 구성이 되어 있으면, 외부 아이피(ExternalIP)에 로드 밸런서 주소를 명시한다.

# kubectl get svc					
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
testlb	LoadBalancer	10.90.18.27	192.168.10.10	80:31973/TCP	6s

이 방식은 내부나 혹은 외부에 장치 혹은 소프트웨어 적으로 로드 밸런서가 구성이 되어 있어야 되기 때문에, 여기서는 외부 로드 밸런서가 아닌, 내부 로드 밸런서 기반으로 구성하도록 한다.

## MetalLB 설치

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

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

ARP를 사용을 제한해야 하는 이유는, 컨트롤 노드 혹은 인프라 노드에 MetalLB를 구성한 경우, 외부에서 특정 아이피 대역으로 DNS를 통해서 호출하였을 때, ARP 호출을 학습하여, MetalLB 컨트롤러가 동작되는 노드에서 패킷을 처리한다. 만약, ARP 사용을 제한하지 않으면, 대다수 ARP 부분은 netfilter 모듈에서 관여 및 조정을 한다. MetalLB 설치를 진행한다.

```
# 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. 문제없이 POD가 보이면, 생성이 완료가 되었다.

```
# kubectl get pods -n metallb-system
> controller-5f56cd6f78-p5ddz   1/1     Running   0          49s
> speaker-rhrtp                1/1     Running   0          49s
```

여기서는 다루지는 않지만, 인프라 노드를 구성 후, 인프라 노드에서 speaker가 동작하도록 구성할 권장한다. 현재는, 컨트롤러에서 동작한다.

## MetalLB 서비스 구성

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

```
# vi metallb-l2-advertisement.yaml
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: example
  namespace: metallb-system
spec:
```

```

ipAddressPools:
- first-pool
# kubectl create -f metallb-l2-advertisement.yaml

```

올바르게 아이피 대역 및 로드 밸런서 설치가 되었는지 테스트를 위해서 네임스페이스를 생성한다. 네임스페이스 명령어로 만들어도 되고, 혹은 아래처럼 파일로 만들어도 된다.

```

# vi metallb-namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: null
  name: lb-test-pool

# kubectl create -f metallb-namespace.yaml

```

올바르게 구성이 되었는지 확인하기 위해서 nginx 를 Deployment 기반으로 구성 및 생성한다. 특별한 부분은 없다.

```

# vi metallb-test-deploy-nginx.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: lb-test-nginx
    name: lb-test-nginx
    namespace: lb-test-pool
spec:
  replicas: 5
  selector:
    matchLabels:
      app: lb-test-nginx
  template:
    metadata:
      labels:
        app: lb-test-nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        ports:
        - containerPort: 80
# kubectl create -f metallb-test-deploy-nginx.yaml

```

로드밸런서를 통해서 서비스에 접근이 가능하도록 서비스에 로드밸런서 형식으로 자원을 생성한다. 일반적으로 사용하는 로드밸런서 설정과 동일하다.

```

# vi metallb-test-svc-nginx.yaml
apiVersion: v1
kind: Service

```

```

metadata:
  creationTimestamp: null
  labels:
    app: lb-test-nginx-svc
    name: lb-test-nginx-svc
    namespace: lb-test-pool
spec:
  ports:
  - name: 80-80
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: lb-test-nginx
  type: LoadBalancer
# kubectl create -f metallb-test-svc-nginx.yaml

```

로드밸런서가 사용할 아이피 주소 범위를 할당한다. 여기서는 총 4 개만 할당하도록 한다. 범위는 192.168.10.0/30 으로 설정한다. 아이피 대역은 리스트 형식으로 여러 개 작성이 가능하다. 다만, 여기서는 쉽게 사용하도록 단일이 작성한다. 아래 코드 중, "namespace:", "namespaceSelectors:", "serviceSelectors:" 는(은) 필요하지 않는 경우 적지 않아도 된다. 특정 네임스페이스에만 적용하고 싶은 경우, 위의 내용 기반으로 조건을 명시한다.

```

# vi metallb-test-ipool.yaml
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: lb-test-pool
  namespace: metallb-system
spec:
  addresses:
  - 192.168.0.0/24
  serviceAllocation:
    namespaces:
    - lb-test-pool
  namespaceSelectors:
  - matchLabels:
      lbtype: metallb
  serviceSelectors:
  - matchExpressions:
    - { key: lbtype, operator: In, values: [metallb] }
# kubectl create -f metallb-test-ipool.yaml

```

"namespace:" 는 metallb-system 로만 명시가 가능하다. 그렇지 않는 경우, 생성오류가 발생한다.

앞에서 이야기하였던, L2 설정 부분에 대해서 좀 더 추가한다. 가상머신을 여러 개 만들어서 랩을 진행하는 경우, L2 를 어느 노드에서 바라볼지 명시하면 된다. 여기서는 node1/2/3 으로 구성하였다. 로드밸런서를 구성 시 사용할 인터페이스는 eth1 을 하며, 레이블 매치는 호스트 이름으로 구성한다. 작은 규모 및 단일 노드로 사용하는 경우, 아래처럼 구성하지 않아도 된다. 여기서 node1/node2/node3 컨트롤러를 통해서 MetalLB 서비스를 구성한다.

```

apiVersion: metallb.io/v1beta1
kind: L2Advertisement

```

```

metadata:
  name: lb-test-pool
  namespace: metallb-system
spec:
  ipAddressPools:
    - test-pool
  nodeSelectors:
    - matchLabels:
        kubernetes.io/hostname: node1.example.com
    - matchLabels:
        kubernetes.io/hostname: node2.example.com
    - matchLabels:
        kubernetes.io/hostname: node3.example.com
interfaces:
  - eth1

```

올바르게 구성이 되면, 다음처럼 서비스에 로드밸런서가 준비가 되어있다.

# kubectl get svc -n lb-test-pool					
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
lb-test-nginx-svc	LoadBalancer	10.90.183.61	192.168.0.0	80:32192/TCP	8m21s

여기까지 구성이 완료가 되면, 다음 목차로 넘어갈 준비가 되었다.

## 2. 인그레스(ingress)

인그레스 서비스 외부에서 쿠버네티스 클러스터에 접근 시, 통과하는 일종의 클러스터 게이트웨이다. 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

앞에서 이야기하였던 로드밸런서와 동일하게 소프트웨어 기반으로 역시 인그레스 프록시 서비스로 일반적으로 Nginx, HAProxy 를 많이 사용한다. 다만, 순수한 버전(일반적인 베어메탈 구성)으로 진행하는 경우, 쿠버네티스와 통합이 어려운 부분이 있다.

인그레스 서비스는, 위의 두개 이외에도 많은 서비스 애플리케이션이 있다. 우리는 대표적인 애플리케이션 두 개 기반으로 설치 및 구성을 다루도록 한다. 참고로, 모든 랙은 Nginx Ingress 기반으로 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, HAProxy 와 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 랙에서는 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 Pod는 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 namespace ingress-nginx-demo
# kubectl create deployment -n=ingress-nginx-demo ingress-nginx-demo --image=nginx --
port=80
# kubectl expose deployment -n=ingress-nginx-demo ingress-nginx-demo
# kubectl create ingress -n=ingress-nginx-demo demo-nginx-local --class=nginx --
rule="demo-nginx.local/*=ingress-nginx-demo:80"
# kubectl port-forward -n=ingress-nginx-demo service/ingress-nginx-demo 8080:80
# curl --resolve demo-nginx.local:8080:127.0.0.1 http://demo-nginx.local:8080
```

위와 같이 하였을 때 문제없이 기본 nginx 환경 페이지를 출력하면, 실제로 외부에서도 크게 문제없이 접근이 가능한지 확인해보도록 한다.

위에서 구성한 서비스를 가지고 계속 작업을 진행한다. 테스트를 위해서 임시적으로 도메인 test.demo.io 도메인을 사용하여 접근이 가능한지 확인한다. test.demo.io 2 차 도메인이 없는 경우, 추가 후 진행한다.

```
# kubectl create ingress test-demo-io --class=nginx --rule="test.demo.io/*=ingress-
nginx-demo:80"
# kubectl get -n=ingress-nginx-demo ingress -w
> nginx-demo-io      nginx  test.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, 위와 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. 0
                           2024061802      ; Serial
                           7200          ; Refresh
                           3600          ; Retry
                           604800        ; Expire
                           7200)         ; NegativeCacheTTL

TN      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.conf에 도메인을 추가한다. /etc/named.rfc1912.zones에 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 service는 내부 아이피로 되어 있기 때문에 외부에서 접근이 가능하도록 수정한다. 파일 이름은 ingress-expose-lb.yaml으로 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

```

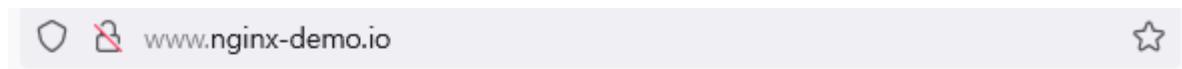
기존 자원과 이름이 같은 경우, 제거하여도 되고, 혹은 이름만 다르게 생성하여도 된다. 제거 및 생성이 완료가 되면 다음과 같이 자원이 구성이 되어 있는지 확인한다.

```

# 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](http://nginx.org).  
Commercial support is available at [nginx.com](http://nginx.com).

*Thank you for using nginx.*

## HAProxy 소개 및 설명

HAProxy 는 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, OpenShift 는 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.

```
# cat <<EOF> haproxy-ingress-values.yaml
controller:
  hostNetwork: true
  ingressClassResource:
    enabled: true
EOF
# helm repo add haproxytech https://haproxytech.github.io/helm-charts
# helm install haproxy-kubernetes-ingress haproxytech/kubernetes-ingress \
--create-namespace \
--namespace haproxy-controller \
-f haproxy-ingress-values.yaml
# kubectl -n ingress-controller get services haproxy-ingress
> 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
```

설치가 완료가 되면 다음처럼 POD 생성이 되었는지 확인한다. Nginx 와 동일하게 한 개의 컨트롤러가 동작하면 문제없이 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 Server를 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.

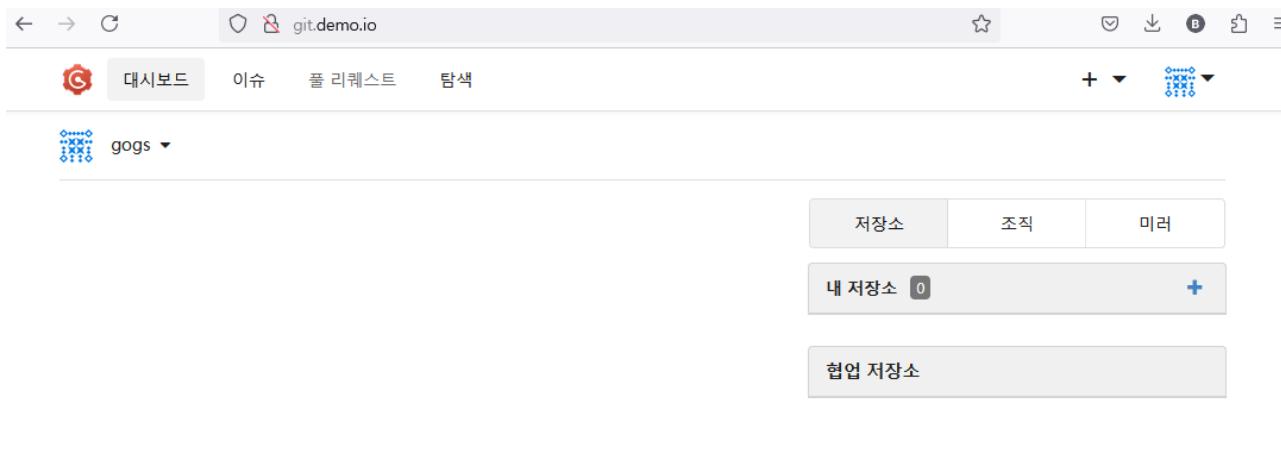
```
---
```

```
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 해더의 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.yaml 으로 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 ClusterIP로 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:
                name: kube-registry
                port:
                  number: 5000
```

podman, buildah에서 이미지 업로드 시, body 크기 때문에 업로드가 올바르게 되지 않기 때문에, 크기 제한 및 타임 아웃을 넉넉하게 잡는다.

만약, skopeo 나 명령어가 익숙하지 않는 경우, 확인용 대시보드 설치가 가능하다. 이름은 kube-registry-dashboard.yaml으로 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
          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

```

v2 및 도메인 이름 설정.

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.

Web Registry

Home

## Repositories

Registry  
http://reigstry.demo.io

Repository	Tags
demo/ip	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, argocd 부터 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
spec:
  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, ingress에서 TLS 를 paththrough 기능이 되도록 활성화한다. 우리는 argocd에서 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 argocd에 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. Kube-virt

쿠버네티스는 기본적으로 컨테이너만 지원만 하였다. 하지만,

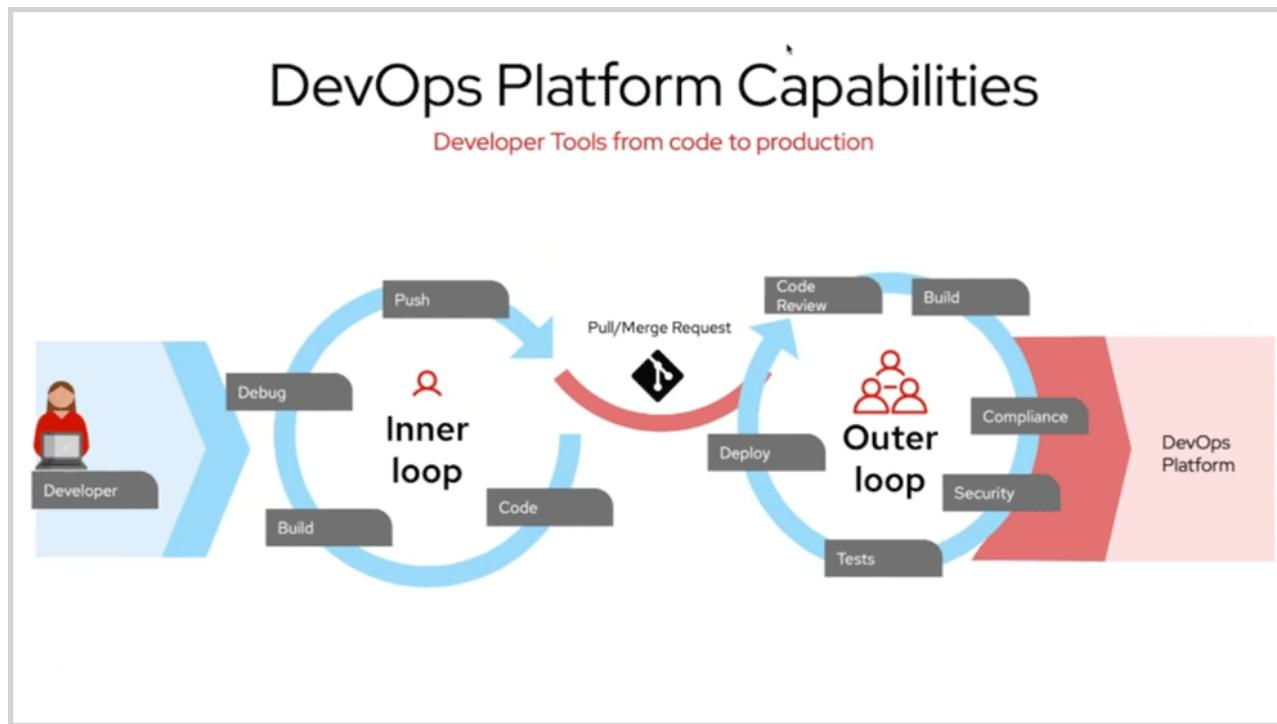
## DevOPs 를 위한 준비

### 1. 데브옵스는 무엇인가?

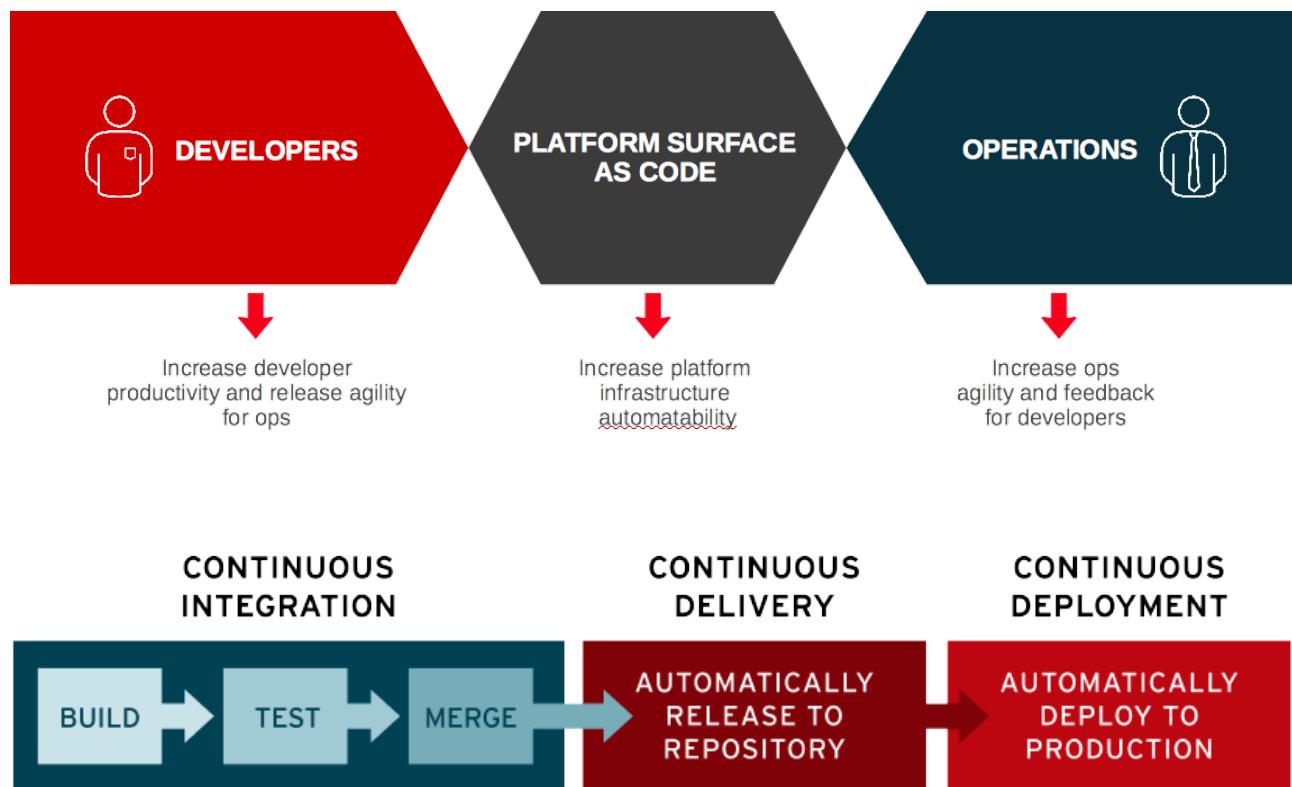
데브옵스(DevOps) 개발자/플랫폼/서비스가 하나의 사이클로 동작하는 개념이다. 데브옵스는 애자일(Agile)의 하위 개념이며, 효율적으로 애플리케이션 개선 및 배포를 위한 라이프 사이클이다. 보통 데브옵스는 두 가지 구역으로 나누어서 관리한다.

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-nginx.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 skopeo 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>47</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 빌드 시, 빌더를 통해서 이미지 빌드 작업을 수행한다. 여기서는 테크톤은 사용하여 이미지 빌드 작업을 자동화한다.

<sup>47</sup> 저자는 맨 처음에 "빌드아"라고 읽었다. 일본에 살고 있는 친구가 이와 같이 알려주었고, 꽤 오랫동안 그렇게 믿었다. 컨퍼런스 가기 전까지…

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 기준 도커와 동일한 방법으로 사용 및 적용이 가능하다. 아래와 같이 실험용 컨테이너 파일을 작성한다. 파일 이름은 Containerfile로 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.php으로 작성한다.

```
<?php
$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 2012년에 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)을 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)을 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

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.

## 랩

작성 중...

## 4. Tekton

### 설명

테크톤은 CNCF에서 인증한 오픈소스 CD 도구이다. 이를 통해서 소스코드 컴파일 및 이미지 빌드 그리고, 서비스 배포가 가능하다. 테크톤 이외에도 많은 도구가 있지만, 현재 CNCF에서는 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 및 설정

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_Linux_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
```

마지막으로, 이미지 빌드를 하기 위해서는 기본 쿠버네티스 계정에 RBAC를 통해서 default namespace에서 이미지 빌드 컨테이너를 생성 및 구성할 수 있도록 설정해야 한다. 아래와 같이 ROLE/ROLE BINDING를 구성한다.

파일 이름은 tekton-clusterrole.yaml으로 작성한다. 올바르게 작성하면 바로 클러스터에 적용한다.

```
# cat <<EOF> tekton-clusterrole.yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: '*'
  name: tkn-action
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
EOF
# kubectl apply -f tekton-clusterrole.yaml
# kubectl get clusterrole | grep tkn-action
```

ROLE이 적용이 될 namespace에 연결 및 구성한다. 파일 이름은 tekton-clusterrolebind.yaml으로 작성 및 구성한다.

```
# cat <<EOF> Tekton-clusterrolebind.yaml
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
```

```

metadata:
  name: rolebind-for-deploy
subjects:
  - kind: ServiceAccount
    name: tkn-action
    namespace: blog-v2
roleRef:
  kind: ClusterRole
  name: tkn-action
apiGroup: rbac.authorization.k8s.io
EOF
# kubectl apply -f Tekton-clusterrolebind.yaml

```

이미지 빌드 시, 사용할 서비스계정(Service Account, SA)를 구성 및 생성한다. 파일 이름은 tekton-sa.yaml으로 작성한다.

```

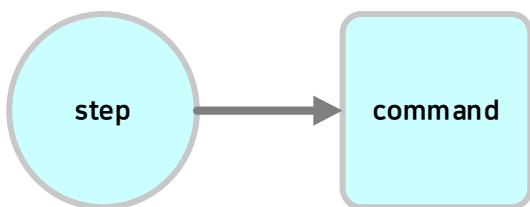
# cat <<EOF> tekton-sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tkn-action
  namespace: blog-v2
EOF
# kubectl apply -f tekton-sa.yaml

```

설치 및 권한 설정이 완료가 되면, 다음 단계(step)으로 넘어간다.

## 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 command으로 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

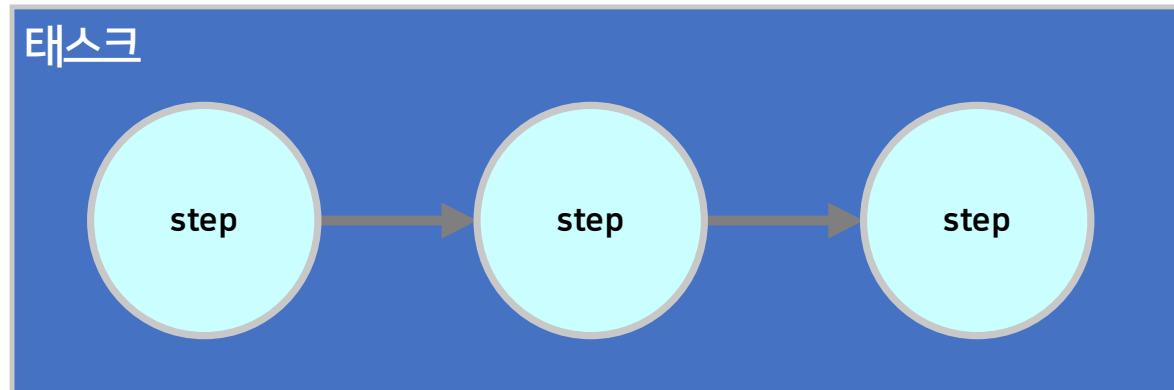
```

```
command:  
  - ls  
args:  
  - -la
```

단계는 독립적으로 자원 구성이 되지 않으며, 이는 **작업(tasks)**에서 구성 및 구현이 가능하다.

## 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.



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.

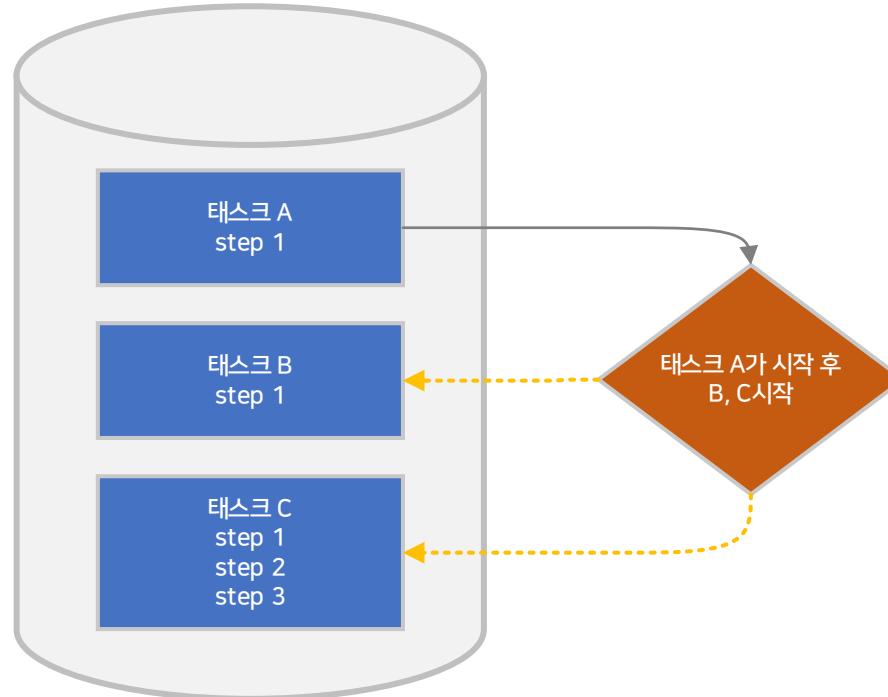


그림 69 파이프라인

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
<b>workspace</b>	파이프라인에서 생성된 자원을 workspace를 통해서 공유한다. 공유가 된 자원은 쿠버네티스에서 생성 및 구성된 PV/PVC 통해서 공유하게 된다.
<b>result</b>	쿠버네티스 실행 후 발생 결과를 테크톤 Pod에 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 랙에서 사용하는 확장 기능은 총 3 가지가 있다.

1. kubernetes-action
2. buidlah
3. git-clone

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.yaml으로 작성한다.

```

apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: tkn-build-deploy-demo
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
  - name: deploy-service
    taskRef:
      name: kubernetes-actions
    params:
      - name: script
        value: |
          kubectl apply -f https://git.demo.io/app/php-ip-deploy.yaml
          kubectl get deployment

```

위의 내용이 올바르게 동작하면, CD 를 통해서 쿠버네티스 딜리버리가 가능하도록 pipeline-run 리소스를 추가적으로 생성한다. 위의 단계는 소스코드를 받은 후, 빌드 및 검증 후 컨테이너 이미지로 패키징 하는 단계이다. 이 단계가 완료가 되면, kubectl 명령어를 통해서 쿠버네티스에서 서비스를 배포한다.

## 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 Kiali를 설치한다. 대시보드 그래프를 위한 Grafana도 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 SVC를 확인한다.

```
# kubectl -n istio-system get svc kiali
```

Kiali에서 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

knative는 설치가 다른 도구에 비해서 좀 복잡하고 까다롭다. operator를 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 Kourier이며, 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. knative를 사용하기 위해서 가급적이면 메모리는 8기가 이상을 권장한다.

```
# kubectl get pods,svc -n knative-serving
```

## MSA 서비스 구성(데모)

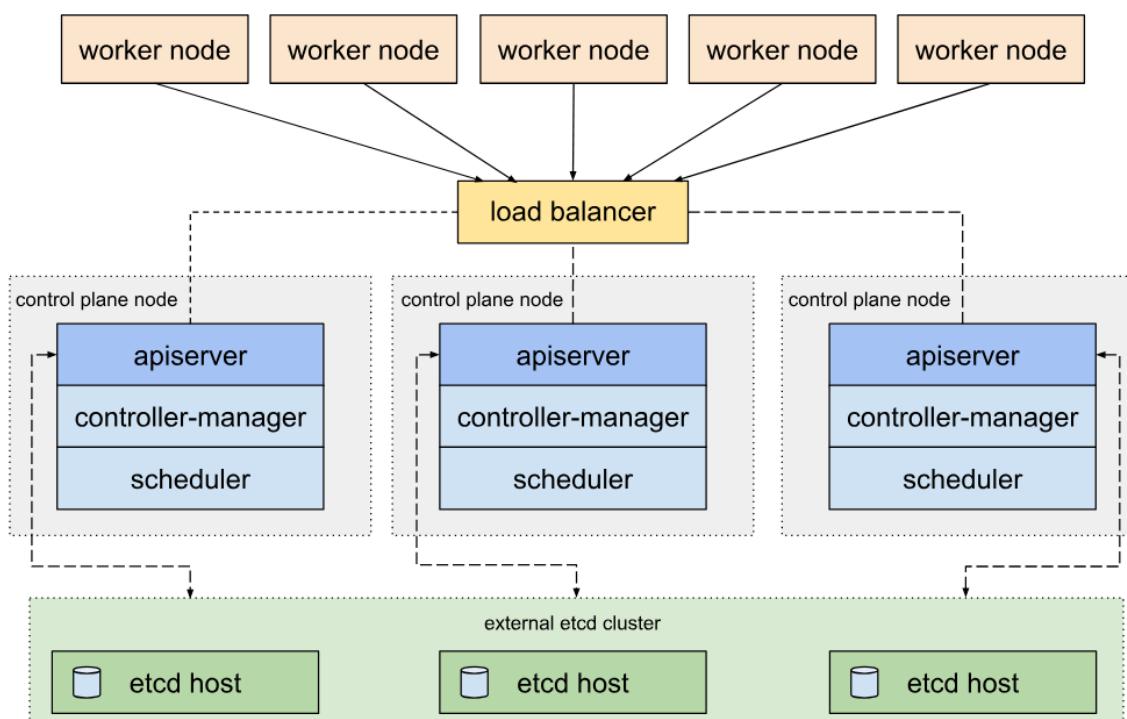
### 3Tier 서비스 구성

### 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

안정적인 클러스터 데이터 저장 및 운영을 위해서 ETCD 서버를 외부에 독립적으로 운영이 필요한 경우가 있다. 혹은, 서비스 활용도가 높아지면서 etcd 만 확장이 필요한 경우도 있다. 이러한 케이스에서는 다음과 같이 쿠버네티스에서 권장하고 있다.

kubeadm HA topology - external etcd



이전 쿠버네티스에서는 etcdadm<sup>48</sup>를 통해서 쉽게 구성이 되었지만, 현재는 더 이상 etcdadm를 사용하지 않기 때문에, 별도로 운영자가 직접 구성 및 배포를 해야 한다. 다만, 이 방법을 사용하면, 인증서 순환(certification rotation, upgrade) 기능을 더 이상 쿠버네티스에서 지원을 받을 수가 없다.

While kubeadm is being used as the management tool for external etcd nodes in this guide, please note that kubeadm does not plan to support certificate rotation or upgrades for such nodes. The long-term plan is to empower the tool [etcdadm](#) to manage these aspects.

## 6. External DNS 서버 구성

## 7. 쿠버네티스 스케줄러 설정 및 변경하기

## 8. 로그 및 트러블 슈팅을 위한 방법

## 9. 오프라인(disconnected) 설치

---

<sup>48</sup> [etcdadm](#)

**10. 이미지 복사 및 배포**

**11. 외부 계정 연결**

**12. 클러스터 연합**