쿠버네티스 101

작성자: 최 국현

갱신날짜: 2023-03-06

릴리즈 상태: 비릴리즈

개요

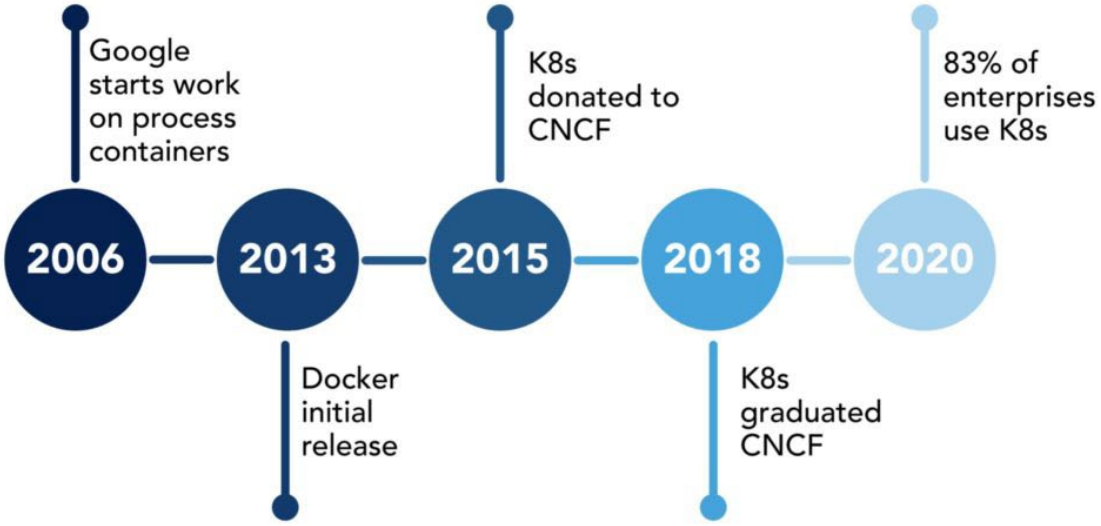
이 책은 쿠버네티스 경험이 없는 사용자를 위해서 작성이 되었다. 쿠버네티스 설치 및 기본적인 명령어 그리고 YAML 기반으로 자원 작성하는 방법 위주로 정리하였다.

사용하는 컴퓨터에 가상화 기능을 사용할 수 없는 경우, 미니큐베와 같은 다른 도구를 통해서 사용하여도 상관 없다. 가급적이면, 노드를 직접 구축 후 교재대로 진행을 권장한다.

목차

내용

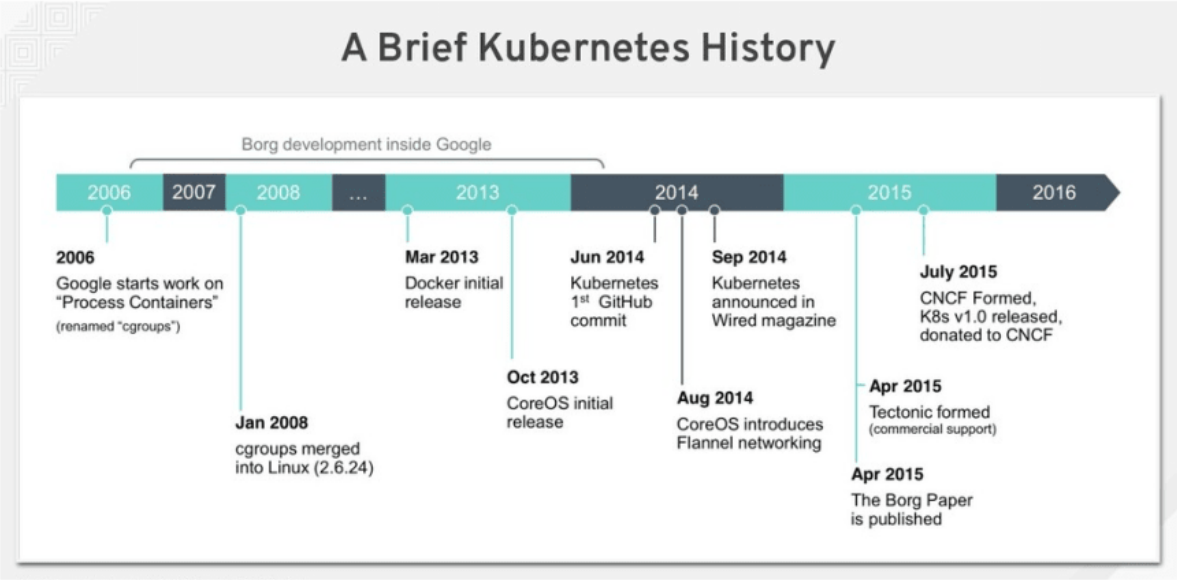
쿠버네티스 소개



구글 및 레드햇은 2006 년도에 컨테이너를 서비스에 도입을 시도하였으며, 이를 통해서 namespace, cgroup 그리고 vServer 와 같은 기능이 리눅스 커널에 도입이 되었다.

초기 리눅스 컨테이너 프로젝트는 BSD 의 Jails 와 비슷하게 하려고 하였으나, 구조적으로 차이가 있어서, Jails 를 도입하지는 못하였다. 그래서 이러한 문제를 해결하기 위해서 앞서 말한 namespace, cgroup, vServer 프로젝트를 통해서 격차를 줄이기 시작했다.

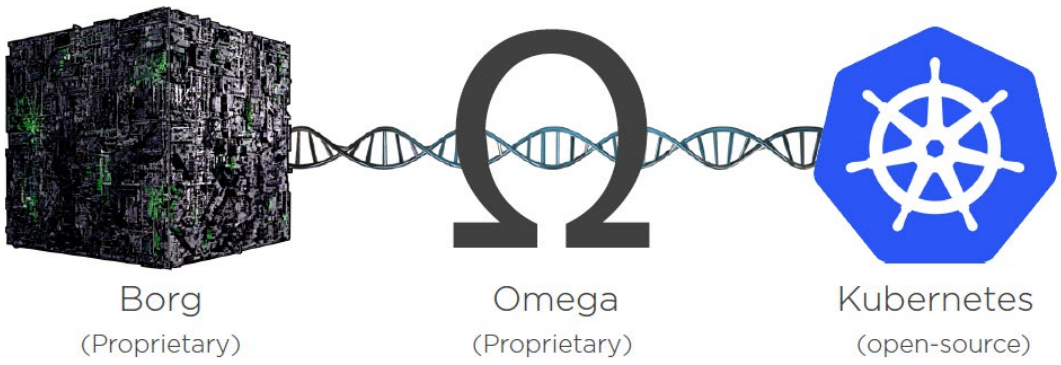
이를 시점으로 chroot, lxc 와 같은 컨테이너 도구가 나왔으며, 본격적으로 rootless 형태의 컨테이너는 도커 런타임이 릴리즈가 되면서 루트리스-컨테이너(rootless-container)의 시대를 본격적으로 시작했다. 기존 컨테이너와 도커와 제일 큰 차이점은 자원을 완벽하게 격리 및 제어가 가능하며, lxc 와 다르게 순수하게 사용자 영역(userspace)에서 구현 및 사용이 가능하게 되었다. 물론, 파일 시스템이나 네트워크 부분은 리눅스 커널을 통해서 제공받기 때문에, 리눅스 커널의 기능이나 성능이 매우 중요하다.



구글은 cgroup 기반으로 컨테이너를 구성하려고 하였고, 이를 통해서 OS 에서 프로세스 제어가 가능하다. 다만, Jails 처럼 격리를 하기 위해서는 추가적인 기능이 필요 하였는데 이는 namespace 가 해결하였다. 현재 리눅스 커널은 cgroup 및 namespace 를 내부 기능으로 가지고 있기 때문에 별도의 커널 패치 없이, 바로 사용이 가능하다.

이때, CoreOS 프로젝트가 시작이 되었으며, 리눅스 기반으로 컨테이너 배포판이 목적이었다. CoreOS 는 다른 배포판과 다르게 패키지 관리자가 없었으며, 순수 컨테이너 기반으로 운영 및 관리할 수 있도록 하였다. CoreOS 가 가지고 있었던 기술은 etcd, rtk, quay 와 같은 제품과 기술을 가지고 있었다. 지금은 레드햇 제품의 일 부분이 되었다.

레드햇은 CoreOS 를 2018 년도에 CoreOS 를 인수하였고, Atomic 라는 이름으로 판매를 하다가, 프로젝트 중지하였다. CoreOS 는 Fedora CoreOS 및 RHEL CoreOS 라는 이름으로 계속 릴리즈 하고 있다.



2014 년도에는 구글은 쿠버네티스 프로젝트를 깃헙을 통해서 릴리즈 하였고, 이전에는 Borg 라는 이름으로 사용하였다. 최종적으로 구글, 레드햇, 수세 및 IBM 그리고 오픈소스 커뮤니티는 쿠버네티스 기반으로 인프라 및 클라우드 오케스트레이션에 대한 표준을 만들어가고 있다.

이를 통해서 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.

설치 환경

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

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.

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 Player 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.

기본 랩 구성

랩은 내부 및 외부 네트워크 두개로 구성이 되며, 외부 아이피는 NAT 를 통해서 외부와 통신이 되며, 내부는 마스터 및 각 노드끼리 API 및 터널링을 위한 용도로 사용한다.

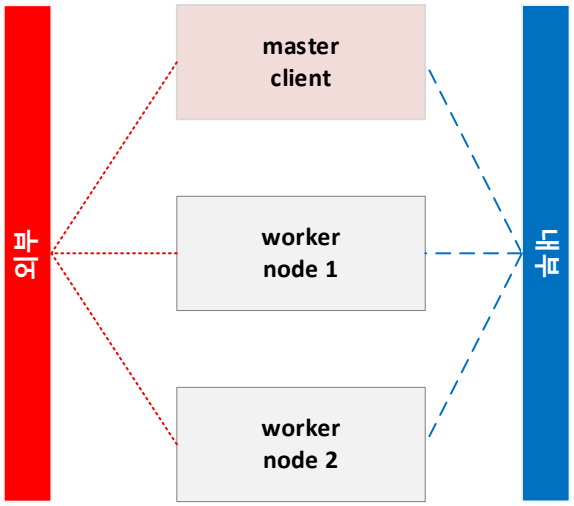


그림 1 랩 구성도

별도로 부트스트랩 및 유틸리티 노드는 없으며, 마스터 서버가 유틸리티 서버 역할도 한다. 여기서는 스토리지를 위한 NFS 기능을 마스터 서버가 가져간다.

윈도우 혹은 맥 기반 설치

윈도우 혹은 맥에서 설치하는 경우 다음과 같은 하이퍼바이저 사용을 권장한다. 앞에서 말했지만 VirtualBox 는 종종 아키텍처 혹은 버전별로 vCPU 문제가 있기 때문에 권장하지 않는다.

- VMware player 혹은 VMware workstation

- VirtualBox
- Hyper-V

랩에서 사용하는 가상머신은 다음과 같은 사양을 가져야 한다. 단, 이 사양은 최소 사양이니 구성 시 참고한다.

- **vCPU:** 2 개 이상(1 개 OS, 1 개 kubernetes)
- **vMEM:** 4096 기가 이상
- **Disk:** 12 기가 이상(CentOS 7 이상부터는 최소 10 기가 필요 합니다)
- **NIC:** 총 2 개. NAT 혹은 외부에서 접근 가능한 대역, 1 개 쿠버네티스 내부 전용

현재 더 이상 CentOS 7 은 지원하지 않기 때문에, 설치는 **CentOS-8/9-Stream** 이나 혹은 Rocky Linux 으로 진행한다. 책에서 사용하는 버전은 CentOS-8-Stream 으로 구성이 되어 있다.

에디터 설정

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

```
$ cat <<EOF> /$($USER)/.vimrc
au! BufNewFile,BufReadPost *.u{yaml,yml} set filetype=yaml foldmethod=indent
autocmd FileType yaml setlocal ts=2 sts=2 sw=2 expandtab
source ~/.vim/plugins/ale.vim
EOF
```

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

```
$ cat <<EOF> /$($USER)/.nanorc
# Supports `YAML` files
syntax "YAML" "\.ya?ml$"
header "^(---|===)" "%YAML"
## Keys
color magenta "^\s*[\$A-Za-z0-9_-]+\:"
color brightmagenta "^\s*@[\$A-Za-z0-9_-]+\:"
## Values
color white ":\s.+$"
```

```
## Booleans
icolor brightcyan " (y|yes|n|no|true|false|on|off)$"
## Numbers
color brightred " [[:digit:]]+(\.[[:digit:]]+)?"
## Arrays
color red "\[" "\]" ":\s+[|>]" "^\s*- "
## Reserved
color green "(^| )!!(binary|bool|float|int|map|null|omap|seq|set|str) "
## Comments
color brightwhite "#.*$"
## Errors
color ,red ":\w.+$"
color ,red ":'.+$"
color ,red ":".+$"
color ,red "\s+$"
## Non closed quote
color ,red "['\"][^['\"]]*$"
## Closed quotes
color yellow "['\"].*['\"]"
## Equal sign
color brightgreen ":( |$)"
set tabsize 2
set tabstospaces
```

ALE (Asynchronous Lint Engine)

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

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

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

ALE 사용하기 위해서 아래 내용대로 그대로 에디터 설정을 한다.

```
# vi ~/.vimrc
source ~/.vim/plugins/ale.vim
# mkdir -p ~/.vim/plugins/
# dnf install yamllint -y
# vi ~/.vim/plugins/ale.vim
" turn on the syntax checker
syntax on

" don't check syntax immediately on open or on quit
let g:ale_lint_on_enter = 0
let g:ale_lint_on_save = 1

" error symbol to use in sidebar
let g:ale_sign_error = '☢'
let g:ale_sign_warning = ' '

" show number of errors
function! LinterStatus() abort
    let l:counts = ale#statusline#Count(bufnr(''))
    let l:all_errors = l:counts.error + l:counts.style_error
    let l:all_non_errors = l:counts.total - l:all_errors
    return l:counts.total == 0 ? 'OK' : printf(
        \   '%d✕ %d⚠ ',
        \   all_non_errors,
        \   all_errors
        \)
endfunction
set statusline+=%=
set statusline+=\ %{LinterStatus()}

" format error strings
let g:ale_echo_msg_error_str = 'E'
let g:ale_echo_msg_warning_str = 'W'
let g:ale_echo_msg_format = '[%linter%] %s [%severity%]'
```

YAML 문법 검사 기능(lint)를 사용하기 위해서 아래처럼 yamllint 를 활성화 한다.

```
# vim ~/.config/yamllint/config
extends: relaxed
rules:
  line-length: disable
```

설치방법

앤서블 기반으로 설치

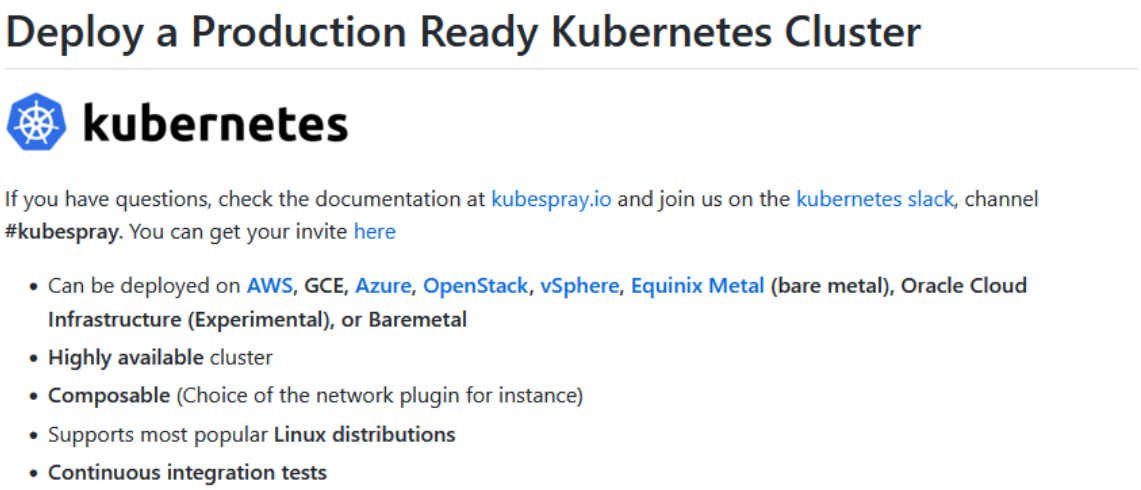
쿠버네티스는 기본적으로 프로비저닝 명령어 기반으로 마스터 노드 및 워커 노드 구성을 하였다. 이러한 방식은 작은 개수의 워커 노드 운영에는 문제가 없었지만, 수십 혹은 수백 개 노드를 구성시에는 적절하지 않는다.

2014 년 이후로는 호스트 설치 방식에서 컨테이너 설치 방법으로 변경이 되면서 모든 쿠버네티스 서비스는 컨테이너 기반으로 설치 및 구성할 수 있도록 되었다. 정확히는 이전에는 패키지 방식 설치였다면 지금은 패키지가 아닌 컨테이너 기반으로 설치한다.

현재는 kubeadm 명령어를 통해서 설치 방법을 제공하고 있다. 이 기반으로 쿠버네티스 클러스터 구성한다.

kubespray

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



대다수 설치는 kubespray 를 사용하지만, 여전히 사용하기가 어렵다는 이야기가 있어서 이 책에서는 앤서블 기반으로 따로 구현해서 구현하였다.

아마 kubespray 보다 사용하기가 더 쉽다고 생각한다. 😊 아마, 저만의 생각 일 수 있지만...

duststack-k8s-auto

duststack-k8s-auto 를 사용해서 간단하고 빠르게 쿠버네티스 설치를 진행한다. 만약, 자동설치를 원하지 않는 경우, 수동 설치로 진행한다.

쿠버네티스에서 공식적으로 지원하는 kubespray 가 있지만, 앤서블 플레이북 대해서 설명을 해야 되기 때문에 최대한 간결하게 구성할 수 있도록 한다. 어차피 설치 방법은 어떠한 방법을 사용하던 요구사항에 맞는 사양만 구성 및 구축하면 된다.

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

이 책은 duststack-k8s-auto 기반으로 작성이 되어 있다.

클러스터 생성 도구

kind

kind 는 Kubernetes 를 도커 기반으로 간단하게 클러스터를 생성 후 학습용으로 사용이 가능하다. kind 는 OCM(Open Cluster Manager)에서도 사용이 가능하다.

일반적으로 개인 학습용으로 많이 사용하는 클러스터 생성 도구 중 하나이다.

https://kind.sigs.k8s.io/

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

```
$ brew install kind
```

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

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

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

https://community.chocolatey.org/packages/kind

설치는 파워 쉘에서 아래 명령어로 구성 및 실행이 가능하다.

```
> curl.exe -Lo kind-windows-amd64.exe
https://kind.sigs.k8s.io/dl/v0.17.0/kind-windows-amd64
Move-Item .\kind-windows-amd64.exe c:\some-dir-in-your-PATH\kind.exe
```

리눅스 사용자는 다음 명령어로 kind 설치가 가능하다.

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

minikube

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

먼저, 윈도우에 미니큐베 설치를 진행한다. 설치 시, 깃헙의 바이너리 이미지를 가져와서 구성한다.

```
New-Item -Path 'c:\' -Name 'minikube' -ItemType Directory -Force
Invoke-WebRequest -OutFile 'c:\minikube\minikube.exe' -Uri
'https://github.com/kubernetes/minikube/releases/latest/download/minikube-
windows-amd64.exe' -UseBasicParsing
```

파워 쉘을 관리자 모드로 실행 후, 아래 명령어를 실행한다. 윈도우 $env:path 확인 후, 미니큐베 경로를 추가한다.

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

미니큐베 클러스터를 실행한다.

```
> minikube start
```

올바르게 실행이 되는지 아래와 같은 명령어로 확인한다.

```
> kubectl get po -A
> minikube kubectl -- get po -A
> alias kubectl="minikube kubectl --"
```

앤서블 간단한 소개

앤서블은 2012 년에 발표가 되었으며, 현재는 레드햇이 인수하였다. 앤서블은 두 가지 제품으로 분리가 되어 있다. 앤서블 엔진(혹은 코어) 그리고 타워로 구성이 되어 있다. 여기서는 설치 시 앤서블 엔진(코어)를 사용해서 설치할 예정이다.

앤서블은 YAML 문법 형태의 문법 그리고, 파이썬 기반의 모듈을 통해서 동작한다. 사용자가 작성한 파일은 플레이북(playbook)이라고 불리며, 플레이북 및 역할(role)를 통해서 시스템에서 발생하는 작업을 반복적으로 처리할 수 있도록 디자인이 되어 있다.

앤서블은 총 두가지 제품으로 나누어져 있다. 첫번째는 앤서블 코어는 기본적인 코어 기능만 제공하며, 대시보드 같은 관리 기능은 제공하지 않는다.

앤서블 타워는 기본 코어 기능에 관리기능 및 CI 기능을 제공하여 레드햇은 해당 제품에 대해서 구독기반(subscription)으로 지원 서비스를 제공하고 있다.

결론적으로 앤서블은 앞으로 쿠버네티스 설치 및 운영시에 꼭 사용해야 될 도구 중 하나이다.

앤서블 엔진(코어)

앤서블 코어는 커뮤니티 버전이며, 모든 리눅스 배포판에서 무료로 사용이 가능하다. 앤서블을 사용하기 위해서는 앤서블 코어버전이 설치가 되어 있어야 한다.

앤서블 타워

앤서블 타워는 코어 위에서 동작하는 관리 도구이며, 웹 기반의 사용자 인터페이스를 통해서 사용자가 손쉽게 관리할 수 있도록 지원한다. 주요 기능은 **추적, 인벤토리 관리, 보안 및 사용자 관리**가 주요 기능이다.

앤서블 타워는 Ansible AWX 업-스트림 버전으로 무료로 사용이 가능하다. 이 책에서는 앤서블 코어 기반으로 설치과정을 진행한다. Ansible AWX 기반 배포는 다루지 않는다.

앤서블 기반으로 여러 쿠버네티스 클러스터 배포 및 관리를 하는 경우에는 AWX 사용을 강력하게 권장한다.

앤서블에서 쿠버네티스 지원

이 책에서는 직접 다루지 않는다. 추후에 자동화 책에서 다룰 예정이다. 간단히 설명하면, 현재 앤서블은 쿠버네티스를 관리하기 위한 자동화 모듈을 제공한다. 이 모듈은 API 기반으로 동작한다.

사용하는 방법은 다음과 같다. 네임스페이스 앤서블 생성하면 다음과 같이 플레이북을 생성한다.

```
- name: create namespace
  kubernetes.core.k8s:
    name: first-namespace
    api_version: v1
    kind: Namespace
    state: present
```

포드가 사용할 서비스는 다음과 같이 구성한다.

```
- name: create service with pod
  kubernetes.core.k8s:
    state: present
    definition:
      apiVersion: v1
      kind: Service
      metadata:
        name: nginx
        namespace: first-namespace
        labels:
          app: hosting
          service: web
      spec:
        selector:
          app: hosting
          service: web
        ports:
        - protocol: TCP
          targetPort: 8000
```

```
        name: port-8000-tcp
        port: 8000
```

kubeadm 그리고 설치

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

init

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

init 수행 후 아래처럼 ".kube" 디렉터리 생성 및 구성이 되어 있어야 'kubectl'명령어가 사용이 가능하다.

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

만약 특정 버전을 선택해서 설치하고 싶은 경우, 저장소에서 특정 버전으로 릴리즈가 되어 있는 패키지 기반으로 설치한다. 예를 들어서 다음과 같이 패키지를 설치한다.

```
# yum install kubeadm-1.8.3
```

위와 같이 설치하면 쿠버네티스 1.8.3 버전의 kubeadm 명령어가 설치가 된다. 다른 버전으로 설치하고 싶은 경우에는 다음 명령어로 쿠버네티스 패키지 버전 확인이 가능하다.

```
# yum list kubeadm
Installed Packages
```

```
kubeadm.x86_64                                          1.24.3-0
@kubernetes
Available Packages
kubeadm.x86_64                                          1.25.4-0
kubernetes
```

join

이 하위 명령어는 마스터 노드에 워커를 추가하는데 사용한다. 워커 노드 추가 시 조건은 기본적으로 **"스왑 미사용"**, **"방화벽 선택적 사용"**이 가능하다. 아래는 사용방법 예시다.

```
# systemctl stop firewalld
# swapoff -a
# kubeadm join 192.168.68.122:6443 --token gcd426.2itdtcs7olp7ds6r --
discovery-token-ca-cert-hash
sha256:bfad2126496a0779ccbc2cf9b841834cc930384eb0158cce40332f74a7b544d7
master#] kubectl get nodes
```

만약 멀티 마스터로 클러스터를 구성하는 경우 다음과 같은 명령어로 노드를 2,3...n 노드로 추가를 해주어야 한다.

```
master]# kubeadm init
master]# kubeadm init phase upload-certs --upload-certs
master]# kubeadm token create --print-join-command
master]# kubeadm join 10.10.10.69:6443 --token s1zhl5.hzn7bsg8naeu7kxo \
--discovery-token-ca-cert-hash \
sha256:2a0b134aee6854f3be2876b837dbaf54f5c72ff1c8c284af8f49 \
a657ab47886d --control-plane --certificate-key \
a5d2c74b84159256969255258edee23e6b018619d4b55dd07ec694510927ac93
```

위와 같은 명령어로 멀티 마스터로 사용할 노드에서 명령어를 실행해서 메인 마스터 노드에 가입한다.

upgrade

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

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

“upgrade node”는 worker 노드 업그레이드, “upgrade plan”은 컨트롤 플레인 업그레이드한다. 이 책에서는 업그레이드에 대해서는 다루지 않으며 추후 중/고급 책에서 다룬다.

config

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

아래 명령어는 이미 구성이 되어 있는 내용을 텍스트 파일로 출력한다.

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

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

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

token

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

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

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

유효기간이 지나지 않았으면 생성된 토큰 목록은 아래 명령어로 확인이 가능하다.

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

아래 수동 설치에서 위의 명령어는 종종 사용한다.

reset

현재 구성이 되어 있는 쿠버네티스 클러스터를 초기화 한다. 초기화 시, 기존에 구성이 되어 있는 시스템의 모든 서비스는 초기화 된다.

```
@master/node]# kubectl reset
```

version

쿠버네티스 버전을 출력한다.

```
master]# kubeadm version
```

alpha

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

설치: 리눅스

설치를 진행하기 위해서 먼저 앤서블을 설치한다.

앤서블은 배포판 및 앤서블 버전별로 각기 다른 설치 방법 및 문법을 제공하기 때문에 설치 시 배포판에 해당 버전을 지원하는지 확인한다. 이 플레이북이 올바르게 실행하기 위해서는 앤서블 2.9.x 버전 이상의 앤서블 코어 버전을 요구한다.

이 과정에서는 가상머신을 libvirtd 기반으로 생성하고, 앤서블 플레이 북으로 쿠버네티스 설치한다.

또한 배포판은 **CentOS 7,8,9 Stream/RHEL 7,8,9/Debian** 를 지원하고 있다. 이외 배포판에서는 올바르게 동작이 안될 수 있다.

Ansible/Libvirtd

아래 방법은 리눅스에서 설치하는 방법이며 윈도우 사용자는 다음 목차에서 설명하도록 하겠다.

```
$ yum install epel-release -y && yum install git ansible -y
```

EPEL(Extra Packages for Enterprise Linux)를 설치 후 앤서블 및 깃 클라이언트 패키지를 설치한다. 위의 프로그램 설치가 완료가 되면 git 명령어로 다음처럼 명령어를 실행한다.

```
$ git clone http://github.com/tangt64/duststack
```

GITHUB 에서 duststack-k8s-auto 플레이북을 로컬로 복제한다. 복제가 완료가 되면 가상머신을 생성하기 위한 과정을 진행한다. 아래 명령어를 실행하면 랩에서 사용할 가상머신을 자동으로 libvirtd 기반으로 생성한다.

가상머신 설치를 위해서 설치를 위한 클래스 룸 인벤토리 그리고 랩 변수를 설정한다.

```
$ ansible-playbook -i inventory/classroom -e lab=kubernetes
playbooks/classroom.yaml
```

윈도우 사용자는 수동으로 구성을 해주어야 한다. 이 모드는 자원을 절약하기 위해서 싱글 마스터 + 워커 노드 2 개로 구성이 되어 있다.

가상머신 프로비저닝, 올바르게 실행이 되면 'virsh list'명령어로 다음처럼 확인이 가능하다.

```
$ virsh list
 4    k8s-master                     running
 5    k8s-node-1                     running
```

```
 6    k8s-node-2                    running
```

실행이 올바르게 되면 **k8s-master-1, k8s-node-1, k8s-node-2** 총 3 대의 가상머신이 설치 및 구성이 된다.

구성이 완료가 되면 ssh 비공개 및 공개키를 생성한다. 만약, 생성하지 않는 경우 플레이북을 실행하면 알아서 자동으로 생성 및 배포한다.

```
$ ssh-keygen -t rsa -N '' ~/.ssh/id_rsa
```

키를 배포하기 전에 올바르게 인벤토리 파일에 서버가 등록이 되어 있어야 한다. 인벤토리 파일은 다음처럼 기본 구성이 되어 있다.

```
#[lb]
#192.168.90.65 nodename=lb.example.com

[console]
192.168.90.64

[k8s_master]
192.168.90.64 nodename=master.example.com
#192.168.90.69 nodename=master2.example.com
#192.168.90.67 nodename=master3.example.com

[k8s_node]
192.168.90.66 nodename=node1.example.com
192.168.90.70 nodename=node2.example.com

[k8s_virt]
#192.168.90.66 nodename=node1.example.com
#192.168.90.70 nodename=node2.example.com

[k8s_utility]
#192.168.90.68 nodename=utility.example.com
```

여러분이 사용하는 가상머신의 아이피는 다르기에 여러분이 사용하는 가상머신의 아이피 주소로 변경한다. 또한, 서버에 사용할 적절한 호스트이름으로 변경한다. 우리는 연습용으로 작은 쿠버네티스 클러스터가 필요하기 때문에 아래처럼 수정한다. 아이피 주소는 여러분 환경에 맞게 변경한다.

```
[console]
192.168.90.64

[k8s_master]
192.168.90.64 nodename=master.example.com

[k8s_node]
192.168.90.66 nodename=node1.example.com
192.168.90.70 nodename=node2.example.com
```

위와 같이 변경하면 싱글 노드를 위해서 설정파일을 아래처럼 변경한다.

```
# vim group_vars/kubernetes

private_key_file: ~/.ssh/id_rsa

# Single or Multi master for kubernetes
k8s_proxy_mode: single

# ubuntu, debian, rhel(centos)
distribution_name: redhat

# kubernets storage network
internal_stroage_dev_name: ens4

ingress_controller: haproxy

# firewalld for kubernetes
use_firewalld: yes

k8s_runtime_environment: crio
```

```
# support calico, flanned
k8s_network: calico

# internal, between kubernetes masters and nodes
k8s_interface: ens3

k8s_cluster_svc_ip: 10.96.0.0/16
k8s_pod_cidr: 10.244.0.0/16
k8s_svc_node_port_range: 30000-32767
k8s_cluster_domain: cluster.local
```

위와 같이 변경하면 싱글 노드에서 동작하도록 쿠버네티스 구성 및 설치한다. 멀티모드로 원하는 경우에는 GitHub 에서 멀티 노드 매뉴얼을 참고하도록 한다.

마지막으로 DNS 서버가 없는 경우에는 각각 가상머신에 "A 레코드" 설정이 필요하다. 앤서블 플레이북에서 자동으로 배포 및 구성한다. 수동으로 설치하는 경우에 SSH 키 배포 및 "/etc/hosts"파일 수정을 직접 해주어야 한다.

설치: 윈도우

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 컴퓨트 노드가 올바르게 동작하지 않는 경우가 있다.

하이퍼-브이에서 가상머신을 구성하기 위해서 아래 명령어를 진행한다.

Hyper-V 설치

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

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

가상머신을 생성하기 위해서 아래 디렉터리 두 개를 생성한다.

가상머신 생성

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

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

```
> New-VM -Name master -MemoryStartupBytes 4GB -BootDevice VHD -
NewVHDPath .\VMs\master.vhdx -Path .\VMData -NewVHDSizeBytes 12GB -Generation
2 -Switch External
```

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

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

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

```
> SET-VMProcessor -VMName master -count 2
```

Hyper-V 내부 네트워크 구성

가상머신이 준비가 되면, 각각 머신에 API 통신시 사용할 내부 네트워크를 생성 후 각각 가상머신에 할당한다. 먼저, 아래 명령어로 "Internal"네트워크를 생성한다.

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

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

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

참고로 "master"부분에 "node1", "node2"으로 변경해서 각각 vCore 및 internal switch 를 연결한다.

설치: 앤서블+리눅스

위의 목차 **"설치: 리눅스 환경"** 올바르게 진행 및 준비를 하였으면, 앤서블 플레이북 설치가 가능하다. 아래 명령어로 바로 쿠버네티스 설치 진행이 가능하다. 다만, 설치를 진행하기 전에 사용할 가상머신을 먼저 생성해야 한다.

"inventory/classroom"의 파일이 아래와 같이 설정이 되어 있는지 확인한다. 일반적으로 리눅스가 설치가 되어 있는 로컬 컴퓨터를 사용하기 때문에 아래처럼 구성한다.

```
[local_server]
127.0.0.1
[classroom:children]
local_server
```

위의 설정을 확인한 다음에, 최소 설치를 하기 위해서 "group_vars/classroom/kubernetes"에서 아래처럼 변수를 수정 및 변경한다. libvirtd 에서 사용하는 가상머신은 현재 "centos 7"버전으로 동작하게 되어 있다.

```
k8s_master_nodes:
  - k8s_master_1
k8s_utility_nodes:
k8s_nodes:
  - k8s_node_1
  - k8s_node_2

virt_disk_format: qcow2
virt_disk_image_path: /var/lib/libvirt/images/
virt_disk_size: 20G

local_apache_htdoc_directory: /var/www/html/
repository_url: http://mirror.kakao.com/centos/7/os/x86_64/

libvirt_owner: qemu
libvirt_group: qemu
```

```
vcpu: 2
vmem: 4096
nic1: default
nic2: internal
virt_type: kvm
virt_hypervisor: hvm
virt_graphics: spice

os_type: linux
os_variant: centos7
```

구성이 올바르게 되었으면 아래처럼 실행하면, 플레이북이 가상머신을 libvirtd 기반으로 생성하면서 진행한다.

```
$ ansible-playbook -i inventory/classroom playbook/lab-
provisioning.yaml
```

가상머신 프로비저닝이 완료가 되면, 앤서블 플레이북을 사용하여 가상머신에 쿠버네티스 설치를 진행한다. 완료가 되면 쿠버네티스 인벤토리를 변경하여 가상머신에 설치를 진행한다. 올바르게 구성이 되면 다음과 같은 명령어로 확인이 가능하다.

```
# virsh list
```

설치 대상 서버는 인벤토리에 등록이 되어있다. 플레이북 디렉터리 inventory/kubernetes 에 아래처럼 저장이 되어 있다.

```
[console]
192.168.90.64

[k8s_master]
192.168.90.64 nodename=master.example.com

[k8s_node]
```

```
192.168.90.66 nodename=node1.example.com
192.168.90.70 nodename=node2.example.com
```

위의처럼 메세지가 출력이 되면, 가상머신 준비는 완료가 되었다. 실제 가상머신에 쿠버네티스 노드를 구성한다. 노드 구성은 아래처럼 구성이 되어 있다.

가상머신	호스트 네임	역 할
k8s-master	master-1.example.com	etcd, kubernetes master 설치 및 구성
k8s-node-1	node-1.example.com	kubernetes worker 역할 1
k8s-node-2	node-2.example.com	kubernetes worker 역할 2

준비가 완료가 되면, 설치를 진행한다.

```
$ ansible-playbook playbooks/k8s-install.yaml
TASK [nginx-ingress : Create a custom resource definition for
GlobalConfiguration resource] ***
changed: [192.168.122.110]
…
PLAY RECAP
*********************************************************************
192.168.122.110             : ok=25   changed=15   unreachable=0
      failed=0   skipped=2  rescued=0  ignored=0
```

설치가 완료가 되면, master-1 에 접근 후, 다음 명령어로 확인한다.

```
$ kubectl get nodes
$ kubectl get pods --all-namespaces
```

랩에서 사용한 가상머신이 더 이상 필요 없는 경우 아래 명령어로 가상머신을 모두 제거한다.

```
$ ansible-playbook -i inventory/classroom -e uninstall
playbook/classroom.yaml
```

위의 명령어를 실행하면, libvirtd 에 등록된 모든 가상머신은 삭제 및 제거가 된다.

설치: 수동설치

쿠버네티스 수동 설치는 일반적으로 다음과 같은 과정을 거친다. 아래 조건은 control plan, worker node 에서 동일하게 적용이 된다.

- 설치 전 O/S 설정 구성
- RPM 혹은 DEB 저장소 구성 및 설정(kubernetes, ansible, Enterprise Package)
- kubeadm 및 kubectl 같은 관리 명령어 구성
- 방화벽 및 SELinux 중지
- swap 기능 중지

실제로 수행을 하지 않아도 되기 때문에 수동 설치가 필요한 경우 아래 명령어를 쭉 수행한다. 다만, 이 부분에서는 싱글 마스터만 다루고 멀티 마스터 구성을 수동설치 구성에 대해서는 다루지 않는다.

설치 전 O/S 설정

각 부분별로 스크립트 기반으로 처리하는 명령어를 같이 넣어 두었다. 한번에 구성 및 복사를 하고 싶으면, 스크립트를 마스터 노드에서 수행한다.

/etc/hosts

설치를 진행하기 전, 반드시 DNS 나 혹은 /etc/hosts 에 A Recode 설정을 해야 한다. DNS 서버 구성 및 설치가 귀찮으면 아래와 같이 구성한다.

- dnsmasq
- /etc/hosts

일반적으로 DNS 서버를 구성하지 않는 경우 아래와 같은 방법으로 A 레코드를 설정 및 구성한다. 우리는 이 부분을 앤서블 플레이 북으로 이미 생성 및 구성하였기에 따로 할 필요는 없다.

만약, 수동으로 작성이 필요한 경우에는 아래처럼 실행하면 된다.

```
master/node]# cat <<EOF>> /etc/hosts
192.168.90.110 master.example.com master
192.168.90.120 node1.example.com node1
192.168.90.130 node2.example.com node2
```

```
EOF
master]# dnf install sshpass -y
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

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

dnsmasq 는 보통 애플리케이션 도메인을 구성 및 구성 시 사용한다. 만약 "ingress service"구성이 필요한 경우, 아래처럼 애플리케이션 도메인 및 IP 주소를 설정한다.

```
strict-order
domain-needed
local=/example.com/
bind-dynamic
log-queries
address=/.cloudapps.example.com/192.168.90.250 # Load Balancer
```

만약, 외부에서 "ingress"같은 서비스를 통해서 외부에서 접근이 필요한 경우 위와 같이 설정한다. 하지만, 이 과정은 기초과정이기 때문에 위와 같이 설정할 필요가 없다.

모듈설정

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

- overlay
- br_netfilter

오버레이(Overlay)는 Overlayfs 를 구현 시 사용한다. br_netfilter 는 bridge 관련된 netfilter 모듈이다. bridge 에서 생성 및 구성이 된 정보는 br_netfilter 에서 관리 및 추적한다.

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

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

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

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

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

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

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

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

저장소 설정

쿠버네티스 설치를 위한 저장소를 구성한다. YUM 저장소는 일반적으로 다음과 같은 주소를 가지고 있다. 역시 배포판마다 다르기 때문에 쿠버네티스 웹 사이트에서 확인한다.

```
master/node]# cat <<EOF> /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=kubernetes repository
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-
x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
```

```
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

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

쿠버네티스 설치를 위한 kubeadm 패키지 설치한다. 특정 버전을 원하는 경우, 패키지 명에 버전 숫자를 붙여 넣는다. 아래처럼 버전을 넣지 않는 경우, 최신 버전으로 설치가 된다.

```
master/node]# dnf install kubeadm -y
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

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

CRI-O

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

```
# cat <<EOF> /etc/yum.repos.d/libcontainer.repo
[libcontainer]
name=libcontaienr (CentOS_8_Stream)
type=rpm-md
baseurl=https://download.opensuse.org/repositories/devel:/kubic:/libcontainers
:/stable/CentOS_8_Stream/
gpgcheck=1
gpgkey=https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:
/stable/CentOS_8_Stream/repodata/repomd.xml.key
enabled=1
```

```
EOF
```

CRI-O 를 설치하기 위해서는 CRI-O 저장소를 별도로 등록한다. 최대한 최근에 릴리즈 한 CRI-O 바이너리를 설치한다.

```
# cat <<EOF> /etc/yum.repos.d/crio_stable.repo
[stable_crio]
name=stable cri-o (CentOS_8)
type=rpm-md
baseurl=https://download.opensuse.org/repositories/devel:/kubic:/libcontainers
:/stable:/cri-o:/1.24:/1.24.4/CentOS_8/
gpgcheck=1
gpgkey=https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:
/stable:/cri-o:/1.24:/1.24.4/CentOS_8/repodata/repomd.xml.key
enabled=1
EOF
```

설정 파일은 아래 주소에서도 내려받기가 가능하다.

```
# wget
https://raw.githubusercontent.com/tangt64/training_memos/main/opensource/kuber
netes-101/file/libcontainer.repo
# wget
https://raw.githubusercontent.com/tangt64/training_memos/main/opensource/kuber
netes-101/file/crio_stable.repo
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
# for i in master node{1..2} ; do sshpass -pcentos \
scp /etc/yum.repos.d/{crio_stable.repo,libcontainer.repo} \
root@$i:/etc/yum.repos.d/ ; done
```

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

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

설치를 하기 위해서 CRI-O 를 미리 실행한다.

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

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

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

cri-docker

[추후 입력 필요]

containerd

Containerd 를 사용하는 경우, 배포되는 'containerd.toml'파일이 도커에서 배포한 파일이기 때문에 재구성이 필요하다. 소스 컴파일을 원하지 않는 경우, 도커 저장소를 아래처럼 구성한다.

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

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

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

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

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

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

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

/etc/sysconfig/kubelet

CRI-O 를 사용하는 경우 쿠버네티스에 런타임 위치 및 socket 파일의 위치를 설정파일에 명시합니다. 구성을 변경하려면 /etc/sysconfig 에 다음처럼 입력한다. 최신 쿠버네티스 'kubeadm'를 사용하는 경우 자동으로 컨테이너 런타임 소켓을 확인하기 때문에 아래 파일은 굳이 수동으로 만들지 않아도 된다.

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

만약, 수동으로 설정이 필요한 경우, 위의 파일을 따로 생성하면 된다.

방화벽

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

하나씩 입력하기 어려우면, 아래처럼 스크립트 라인으로 쉘을 통해서 등록한다.

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

```
6443/tcp
2379-2380/tcp
10250/tcp
10251/tcp
10252/tcp
10255/tcp
30000-32767/tcp
179/tcp
```

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

컨트롤 플레인

프로토콜	방향	포트 범위	용도	사용 주체
TCP	인바운드	6443	쿠버네티스 API 서버	전부
TCP	인바운드	2379-2380	etcd 서버 클라이언트 API	kube-apiserver, etcd
TCP	인바운드	10250	Kubelet API	Self, 컨트롤 플레인
TCP	인바운드	10259	kube-scheduler	Self
TCP	인바운드	10257	kube-controller-manager	Self

워커 노드

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

방화벽을 사용하지 않으면 다음 명령어로 중지하면 된다. 이런 경우 별도로 방화벽에 포트를 등록하지 않아도 된다.

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

tc & swap

트래픽 쉐이핑을 하기 위해서 tc 패키지를 설치한다. 이 패키지가 설치가 안되어 있는 경우 쿠버네티스 설치가 진행이 되지 않는다. master 노드에는 상관없지만 반드시 worker 에는 설치가 되어 있어야 한다.

```
node]# yum install tc
```

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

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

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다.

```
master]# for i in master node{1..2} ; do sshpass -pcentos \
ssh root@$i "dnf install tc -y && swapoff -a && sed -i '/ swap /
s/^/#/' /etc/fstab && sed -i s/^SELINUX=.*$/SELINUX=permissive/
/etc/selinux/config " ; \
done
```

Kubernetes Repository

쿠버네티스 설치 후, centos-8-stream 에서 업데이트를 실행 시 다음과 같은 메시지가 출력이 되는 경우가 있다.

```
# yum update
Last metadata expiration check: 3:03:07 ago on Wed 11 May 2022
09:51:25 PM KST.
Error:
 Problem: cannot install both kubelet-1.18.4-0.x86_64 and kubelet-
1.24.0-0.x86_64
  - cannot install the best update candidate for package
kubernetes-cni-0.8.7-0.x86_64
  - cannot install the best update candidate for package kubelet-
1.24.0-0.x86_64
(try to add '--allowerasing' to command line to replace conflicting
packages or '--skip-broken' to skip uninstallable packages or '--
nobest' to use not only best candidate packages)
```

위와 같이 메시지가 발생하면 다음과 같이 yum 혹은 dnf 에서 설정을 한다. 이 문제는 쿠버네티스 패키지의 버전 문제로 충돌이 발생한다. 필요하지 않는 경우, 기본적으로 참조하지 않도록 한다. 방법은 아래와 같이 두 가지가 있다.

```
# cat <<EOF>> /etc/yum.conf
exclude=kubelet*
EOF
# yum update
Last metadata expiration check: 3:04:32 ago on Wed 11 May 2022 09:51:25 PM
KST.
Dependencies resolved.
Nothing to do.
Complete!
```

혹은 아래와 같이 설정이 가능하다. 아래는 쿠버네티스 저장소 파일에 직접 설정하는 방법이다.

```
# cat <<EOF> /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-\$basearch
enabled=1
gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
exclude=kubelet kubeadm kubectl
EOF
```

한 번에 수행을 하기 위해서는 아래와 같이 명령어를 실행한다. 별도로 버전을 명시하지 않으면, 최신버전의 kubeadm 으로 설치가 된다. 각 버전별로 kubeadm 명령어가 존재하기 때문에, 반드시 원하는 버전으로 내려받기 하여 설치해야 한다.

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

kubeadm 옵션

이름	설명
--apiserver-advertise-address	쿠버네티스가 사용할 API 인터페이스입니다. 사용할 인터페이스의 아이피 주소를 적으면 됩니다. 책에서는 192.168.90.110 으로 사용합니다.
--apiserver-bind-port	kubelet 에서 사용하는 포트 6443 를 변경합니다. 일반적으로 변경하지 않고 기본값을 사용합니다.
--control-plane-endpoint	
--cri-socket	
--ignore-preflight-errors	
--image-repository	
--pod-network-cidr	
--service-cidr	
--service-dns-domain	
--upload-certs	

명령어 기반, 싱글 마스터 설치

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

먼저, SELinux 가 동작 중이면, 랩을 위해서 잠깐 중지를 한다.

```
# getenforce
# setenforce 0
```

```
# systemctl enable --now kubelet
# kubeadm init --apiserver-advertise-address=192.168.90.110 --cri-socket=/var/run/crio/crio.sock
```

kubeadmin init 명령어를 실행하다가, policy.json 때문에 설치가 되지 않는 경우가 있다. 그런 경우에는 내용을 다음처럼 수정한다.

```
{
    "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
```

현재 최신버전 기준, kubeadm v1.26 에서는 기본적으로 마스터 확장을 지원하고 있다. 이전에는 보통 etcd 서버에 TLS 키를 업로드 하여 공유하기 위해서 옵션을 별도로 사용하였지만, 현 버전은 그럴 필요가 없다. 하지만, 학습 목적으로 기존에 사용하던 옵션은 그대로 사용한다.

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

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

혹은 복사가 싫은 경우, 아래와 같이 변수를 설정해서 kubectl 명령어 사용이 가능하다.

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

워크 노드 구성 시 다음과 같은 명령어로 실행한다. 워커 노드에서 kubeadm 명령어가 실행이 되지 않으면, kubeadm 패키지만 설치해주면 된다.

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

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

```
# kubectl top nodes
NAME                 CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
master.example.com   75m          3%     1256Mi          34%
node1.example.com    42m          2%     639Mi           17%
node2.example.com    37m          1%     1109Mi          30%
```

싱글 노드는 추가하기 위해서 init, join 이 두 가지 하위 명령만 알고 있으면 된다.

마지막으로 마스터 및 노드 조인이 되었으면 Pod 에서 사용할 터널링 네트워크 구성을 해야 한다. 여기서는 vxlan 기반의 calico 를 사용한다. calico 를 올바르게 구성하기 위해서는 calico 이미지를 hub.docker.io 가 아닌, quay.io 에서 내려받기가 가능하도록 CRD 파일을 만들어서 반영해야 한다. 이 작업들은 모두 master 서버에서 진행이 되어야 한다.

먼저 calico 에서 사용할 calico-quay-crd.yaml 파일을 생성한다.

```
apiVersion: operator.tigera.io/v1
kind: Installation
metadata:
  name: default
spec:
  # Configures Calico networking.
  calicoNetwork:
    # Note: The ipPools section cannot be modified post-install.
    ipPools:
    - blockSize: 26
      cidr: 192.168.0.0/16
      encapsulation: VXLANCrossSubnet
      natOutgoing: Enabled
      nodeSelector: all()
  registry: quay.io
```

생성이 완료가 되면 calico 에서 제공하는 calico-cr 를 등록하여 자원 및 네트워크를 생성 및 구성한다. calico 네트워크를 생성하기 위해서 다음 명령어를 실행한다.

```
# kubectl create -f
https://raw.githubusercontent.com/projectcalico/calico/v3.24.5/manifests/tiger
a-operator.yaml
```

위의 명령어가 올바르게 실행이 되면, 위에서 생성한 calico CRD 파일을 kubectl 명령어로 자원 정보를 업데이트 한다.

```
# kubectl create -f calico-quay-crd.yaml
```

명령어 기반, 다중 마스터 설치

다중 마스터를 구성하기 위해서는 반드시 다음과 같은 조건을 구성해야 한다.

- 한 개 이상의 마스터 서버. 마스터는 절대로 홀수로 구성해야 한다.
- 독립적인 혹은 빌트인 ETCD 서버. ETCD 서버도 독립적으로 구성 시, 절대적으로 홀수로 구성해야 한다.

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

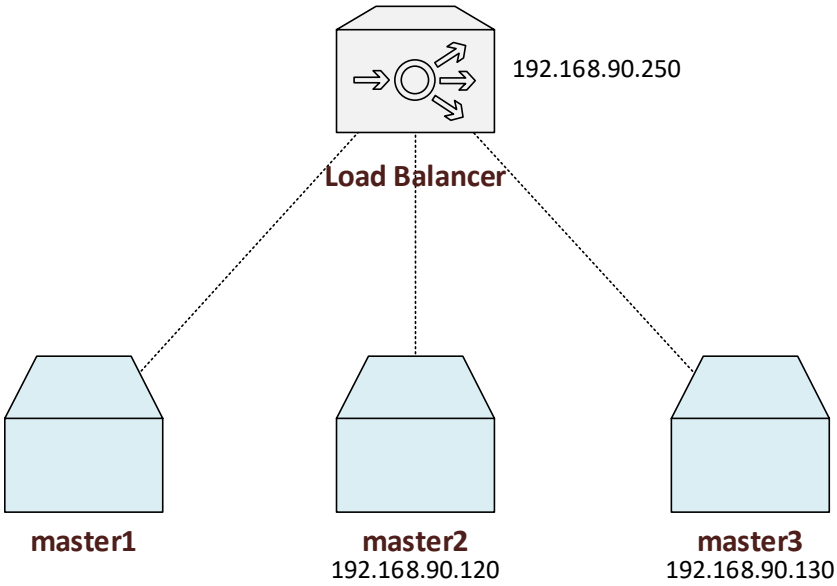


그림 2 Load Balancer, Master Nodes

VIP 를 192.168.90.250 를 사용한다는 조건으로 아래와 같이 명령어를 사용한다.

```
# kubeadm init --control-plane-endpoint 192.168.90.250 --upload-certs --apiserver-advertise-address 192.168.90.110 --pod-network-cidr=192.168.0.0/16 --service-dns-domain=devops.project --image-repository=192.168.90.240
```

컨트롤 패널(마스터)의 네트워크 카드는 192.168.90.250 를 통해서 외부나 혹은 내부에서 들어오는 API 를 처리한다. “--upload-certs”는 부트 스트랩 노드(master1 혹은 bootstrap)에서 생성한 TLS 키를 ETCD 서버에 저장한다. 저장된 TLS 키는 새로 구성되는 마스터 노드에 같이 TLS 키를 배포한다. "--apiserver-adverties-address"는 master1 서버가 API 를 외부에 전달하는 네트워크 카드이다. 현 랩에서는 eth1 장치가 API 를 다른 노드에 있는 쿠버네티스에 전달한다.

최종적으로 "--pod-network-cidr"는 POD 가 클러스터에서 사용하는 네트워크 주소 대역. "--service-dns-domain"는 내부 coredns 에서 사용하는 서비스 도메인 이름을 기본값인 "cluster.local"에서 사용자가 명시한 이름으로 변경한다. "--image-repository"는 내부 혹은 폐쇠망에서 설치시, 내부에 구성이 되어있는 컨테이너 이미지 서버 주소이다. 이 서버에서 쿠버네티스 이미지를 내려받기 한다.

이외 나머지 부분은 **기존 싱글 마스터 설치**와 동일하다.

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

노드를 마스터 클러스터에 워커 노드로 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 192.168.90.100:6443 --token hhjg3n.nmfs94gop1aeom0n --
discovery-token-ca-cert-hash
sha256:01e7f7dee3594c99fa1cb50ad20f3b0b6e1f74d2afaf5f5dc1892bc1b6f247b1
\
  --control-plane \
  --certificate-key
8dc065689de5eb04b57f5538d46bcda977df33b586bcbf4b922fd676eeb0d41e
```

"--control-plane"옵션은 워커 노드, 즉, 컴퓨팅 노드가 아니라 서비스 컨테이너 구성 용도로 사용한다고 명시한다.

올바르게 구성이 되면, 아래와 같이 화면에 출력이 된다.

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

터널링 네트워크 생성

마지막으로 마스터 및 노드 조인이 되었으면 Pod 에서 사용할 터널링 네트워크 구성을 해야 한다. 여기서는 vxlan 기반의 calico 를 사용한다. calico 를 올바르게 구성하기 위해서는 calico 이미지를 hub.docker.io 가 아닌, quay.io 에서 내려받기가 가능하도록 CRD 파일을 만들어서 반영해야 한다. 이 작업들은 모두 master 서버에서 진행이 되어야 한다.

먼저 calico 에서 사용할 calico-quay-crd.yaml 파일을 생성한다.

```
apiVersion: operator.tigera.io/v1
kind: Installation
metadata:
  name: default
spec:
  # Configures Calico networking.
  calicoNetwork:
    # Note: The ipPools section cannot be modified post-install.
    ipPools:
    - blockSize: 26
      cidr: 192.168.0.0/16
      encapsulation: VXLANCrossSubnet
      natOutgoing: Enabled
      nodeSelector: all()
  registry: quay.io
```

생성이 완료가 되면 calico 에서 제공하는 calico-CR 를 등록하여 자원 및 네트워크를 생성 및 구성한다. calico 네트워크를 생성하기 위해서 다음 명령어를 실행한다.

```
# kubectl create -f
https://raw.githubusercontent.com/projectcalico/calico/v3.24.5/manifests/tiger
a-operator.yaml
```

위의 명령어가 올바르게 실행이 되면, 위에서 생성한 calico CRD 파일을 kubectl 명령어로 자원 정보를 업데이트 한다.

```
# kubectl create -f calico-quay-crd.yaml
```

연습문제

현재 구성이 되어 있는 쿠버네티스 초기화 후, kubeadm 명령어로 수동으로 쿠버네티스 클러스터를 구성한다. 명령어는 위의 명시된 명령어로 사용한다.

- kubeadm init
- kubeadm join
- kubeadm token

- kubeadm reset

쿠버네티스 설치 노드에 init 명령으로 설치를 진행한다.

설치 방법은 위의 교재에 나와 있기 때문에 위의 내용을 참조하여 쿠버네티스 클러스터를 구성한다.

1. 마스터 노드는 반드시 가상머신 master 에 설치한다
2. Worker node 1,2 는 반드시 node VMs 에 설치한다
3. 모든 설치는 kubeadm 명령어로 진행한다.

표준 OCI 도구와 관계

Buildah

Buildah 는 기존 Docker 가 이미지 빌드하였던 기능을 분리하여 별도로 만든 **컨테이너 이미지 빌더**. 현재 OCI 에서 이미지 구성 시 권장하는 도구는 buildah 이다. 빌드아(buildah)는 기존 도커 이미지 빌드 도구와 제일 큰 차이점은 데몬 없이 이미지 빌드가 가능하다. 기존 도커는 비어 있는 이미지가 만들기 복잡하였지만, Buildah 는 그에 비해서 매우 간단하게 이미지 스크래치(image scratch)가 가능하다.

설치는 다음과 같은 명령어로 사용이 가능하다.

```
# dnf install buildah -y
```

Buildah 는 두 가지 형태의 이미지 빌드를 제공한다.

1. 기존의 Dockerfile 및 Containerfile 을 통한 이미지 빌드
2. 명령어 기반으로 이미지 빌드

Buildah 는 이전 도커에서 사용하던 이미지 빌드 기능을 별도의 도구로 만들어서 이미지 생성 전용 도구로 사용한다. 주로 테크톤(Tekton)과 같은 도구와 같이 사용한다.

이미지 빌드 방법은 다음과 같다. 앞으로 사용할 표준 Containerfile 기반으로 이미지를 구성한다. 파일 이름은 Containerfile-httpd 으로 한다.

```
FROM quay.io/centos/centos:stream9
ADD helloworld.sh /usr/local/bin
ENTRYPOINT "/usr/local/bin/helloworld.sh"
```

파일 이름은 helloworld.sh 로 만든 다음에 아래와 같은 내용을 입력한다.

```
echo "Hello Kubernetes World!!"
EOF
```

이미지 빌드를 buildah 명령어로 실행한다. 이미지 생성시, 도커와 동일하게 -t 옵션 사용이 가능하다. 빌드가 완료가 되면 다음 명령어로 이미지 목록 확인이 가능하다.

```
# buildah bud -t baseworld:r1 -f Containerfile-httpd
# buildah images
```

이미지 스크래치(베이스 이미지 생성)

buildah 는 이미지 스크래치, 즉 처음부터 생성이 가능하다. 처음부터 생성을 하기 위해서 다음과 같은 과정을 통해서 이미지 생성을 한다. 애플리케이션을 위한 이미지를 처음부터 작성한다면, 아래와 같은 과정을 통해서 베이스 이미지 구성이 가능하다.

먼저, 비어 있는 컨테이너를 생성한다.

```
# buildah from scratch
working-container
```

스크래치 이미지를 생성하기 위해서 아래와 같이 명령어에 변수 설정한다.

```
# scratchmnt=$(buildah mount working-container)
# echo $scratchmnt
```

스크래치 설정이 올바르게 되면, 이미지에 사용할 프로그램을 설치한다.

```
# dnf-y --releasever=8 --installroot=$scratchmnt centos-release
# dnf install -y --setopt=reposdir=/etc/yum.repos.d \ --
installroot=$scratchmnt \ --setopt=cachedir=/var/cache/dnf httpd
# mkdir -p $scratchmnt/var/www/html
# echo "The First Scratch Server!!" > $scratchmnt/var/www/html/index.html
```

사용할 프로그램 및 데이터 구성이 완료가 되면, 컨테이너가 실행 시 애플리케이션 및 네트워크가 올바르게 동작하도록 최종적으로 설정 후 이미지 최종 확인(commit)한다.

```
# buildah config --cmd "/usr/sbin/httpd -DFOREGROUND" working-container
# buildah config --port 80/tcp working-container
# buildah commit working-container localhost/myhttpd:latest
```

자세한 사용 방법은 추후, oci-101 문서를 참고한다.

Skopeo

'docker search'명령어 기능을 skopeo 로 재구성하였다. Skopeo 의 주요 목적인 이미지 **검색/복사/확인** 그리고 미러링 같은 기능을 제공한다. Skopeo 도 buildah 처럼 OCI 에서 권장하는 이미지 관리 도구 중 하나이다.

이미지 복사 및 미러링

```
# skopeo inspect quay.io/centos/centos-stream-8
```

```
# skopeo sync -s docker -d dir registry.k8s.io/kube-proxy /srv/kubernetes-images/
```

자세한 사용 방법은 추후, oci-101 문서를 참고한다.

Tekton

테크톤은 기존에 사용하던 여러 CI/CD 파이프라인 인터페이스를 표준화 했다.

기존에 사용하였던 Jenkins, GitLab 는 경우에는, 쿠버네티스에서 사용이 가능하였지만, Continuous Delivery(CD)를 제공하지만, 설정 부분은 각각 다르게 사용하였다. 그래서 테크톤 기반으로 CD/CI 부분을 하나로 통합을 하고 있다.

현재 테크톤은 쿠버네티스 및 오픈 시프트 양쪽에서 사용이 가능하다. 여기에서는 테크톤 설치 및 구성에 대해서 다루지 않는다.

자세한 내용을 원하는 경우, 쿠버네티스 202 를 참고한다.

Containerd/CRI-O/Podman

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

- docker-shim(더 이상 사용하지 않음)
- cri-docker(OCI)

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

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

쿠버네티스에서 사용하는 런타임 환경은 containerd 및 CRI-O 환경이다. 앞으로 Docker 는 런타임으로써 더 이상 지원하지 않는다고 발표하였다 [1].

> 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

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

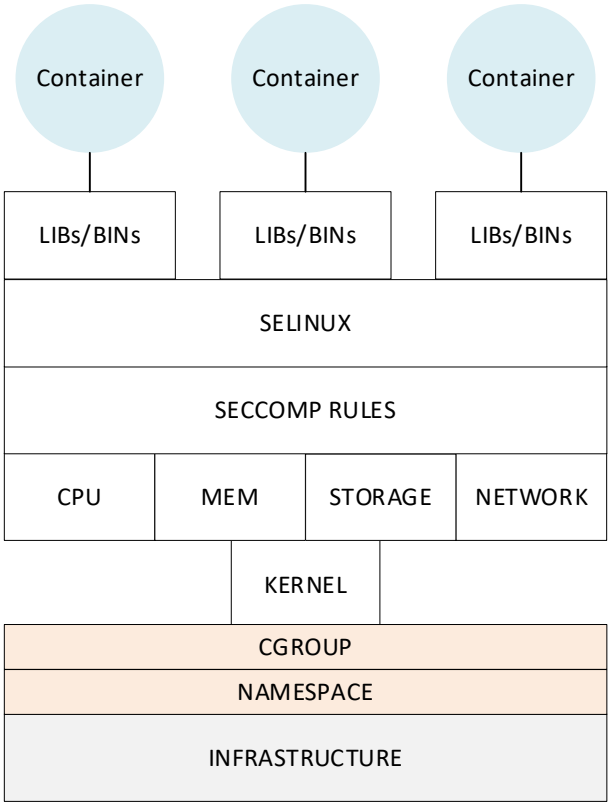


그림 3 컨테이너 구성원

[1] 하지만 shim 구조가 빠진 도커는 여전히 지원하고 있다.

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

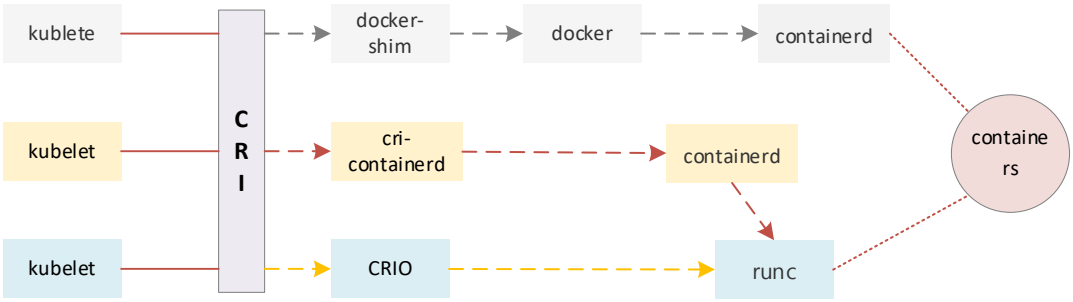


그림 4 컨테이너 CRI

구버전의 도커는 모든 자원들 즉, namespace, cgroup 를 직접 관리하였다. 하지만, Containerd, cri-docker 경우에는 OCI 사양을 따르면서 현재는 직접 관리하지 않는다.[2]

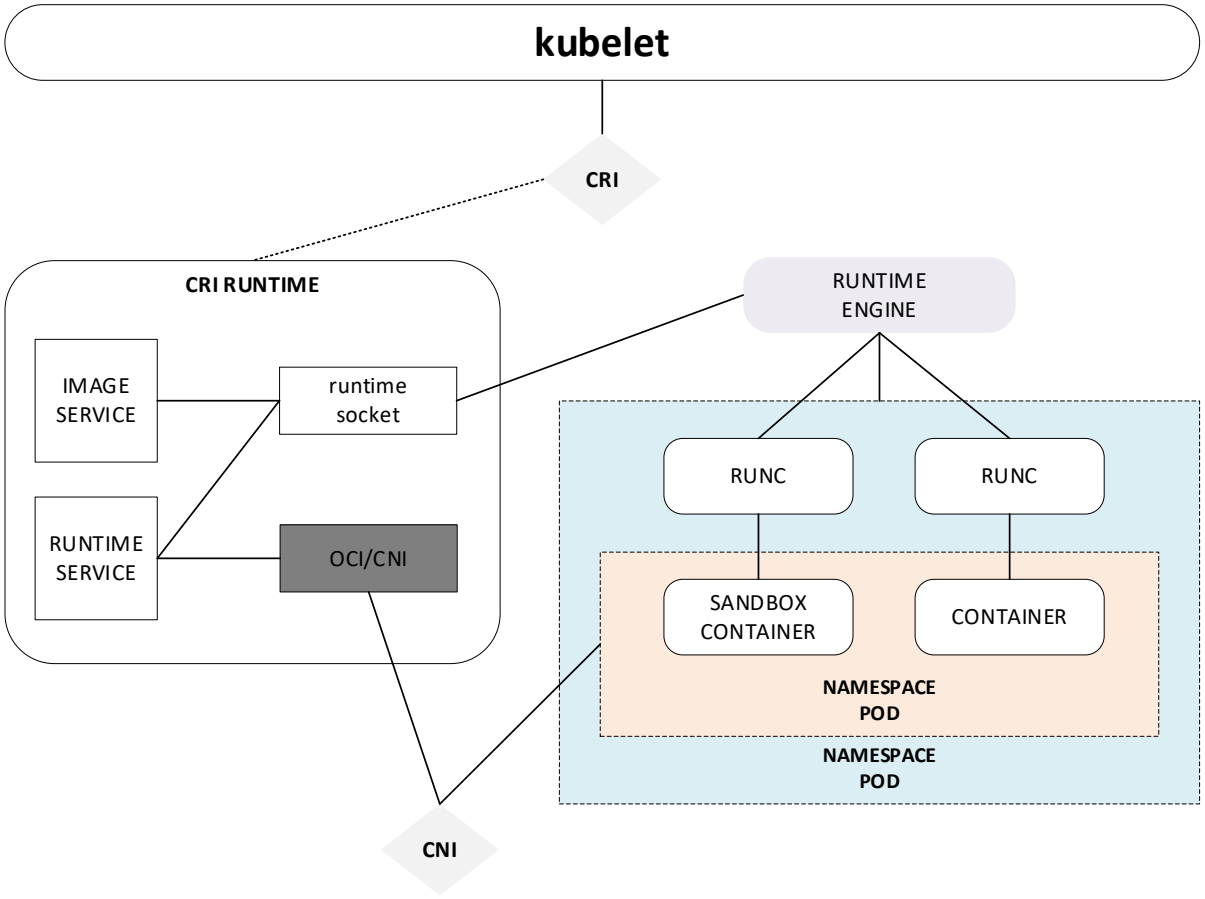


그림 5 containerd 구조

최신 도커(cri-docker 제외) containerd 런타임 기반으로 컨테이너를 생성한다. 이전 도커버전, docker-shim 은 더 이상 쿠버네티스에서 런타임으로 사용하지 않는다. cri-docker 의 구조는 CRI-O 와 동일하다.

[2] docker-shim 참조

CRI-O 의 주요 목적은 컨테이너를 최소화하여 시스템에서 발생하는 작업량을 최소화하여 좀 더 민첩한 컨테이너 환경을 제공한다. CRI-O 의 약자는 "**Container Runtime Interface”**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.

CRIO 의 컨테이너 동작 방식은 기본적으로 로컬에서 컨테이너를 실행하며, docker-ee 처럼 자체적인 오케스트레이션 기능은 제공하지 않는다.

아래 그림은 CRI-O 기반으로 구성된 쿠버네티스 구성이다. 현재 쿠버네티스의 컨테이너 런타임은 CRI-O 사용을 권장하고 있으며, CNCF 에서는 containerd 를 표준으로 채택하고 있다.

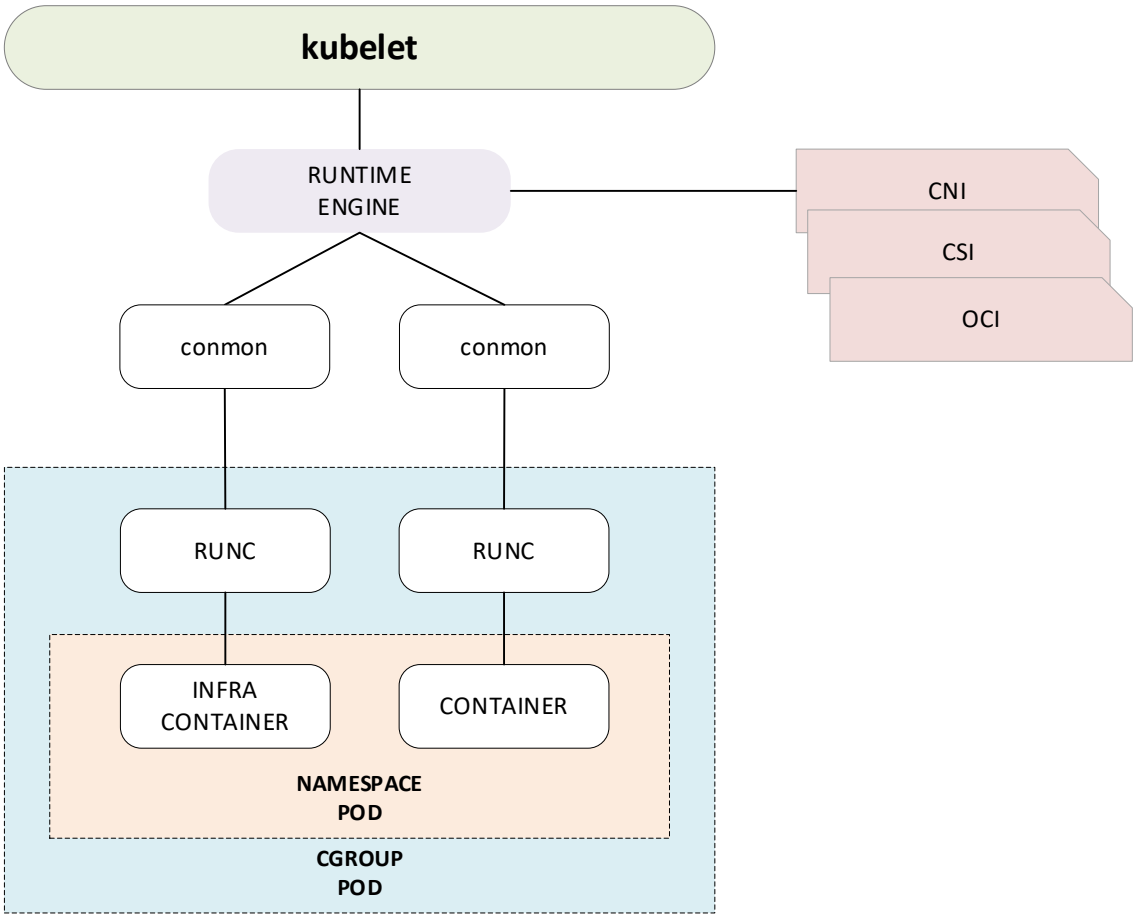


그림 6 CRIO 구조

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.

위의 그림에서 RUNTIME ENGINE 이 CRI-O 이면, 런타임은 엔진에서 사용하는 이미지 및 컨테이너를 직접 관리하며, CRI-O 컨테이너 환경은 표준 인터페이스인 OCI 기반으로 gRPC 및 CNI(Container Network

Interface)를 통해서 구성이 된다. 우리가 일반적으로 컨테이너 런타임(runtime)은 컨테이너 라이프 사이클 관리가 주요 목적이다.

쿠버네티스 클러스터를 구성하기 위해서는 리눅스 커널에서 다음과 같은 기능을 요구한다.

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

추가적으로 요구하는 기술 및 프로그램은 다음과 같다.

- SELinux
- App-Armor

SELinux 는 seccomp 와 비슷하게 필터링 하지만, 더 세밀하게 시스템 콜 및 네트워크 접근에 대한 시스템 보안을 원하는 경우 SELinux 사용이 가능하다. 하지만, "SELinux Policy"[3] 에 대한 지식이 없으면, 일반적으로 끄고 사용하는 걸 권장한다.

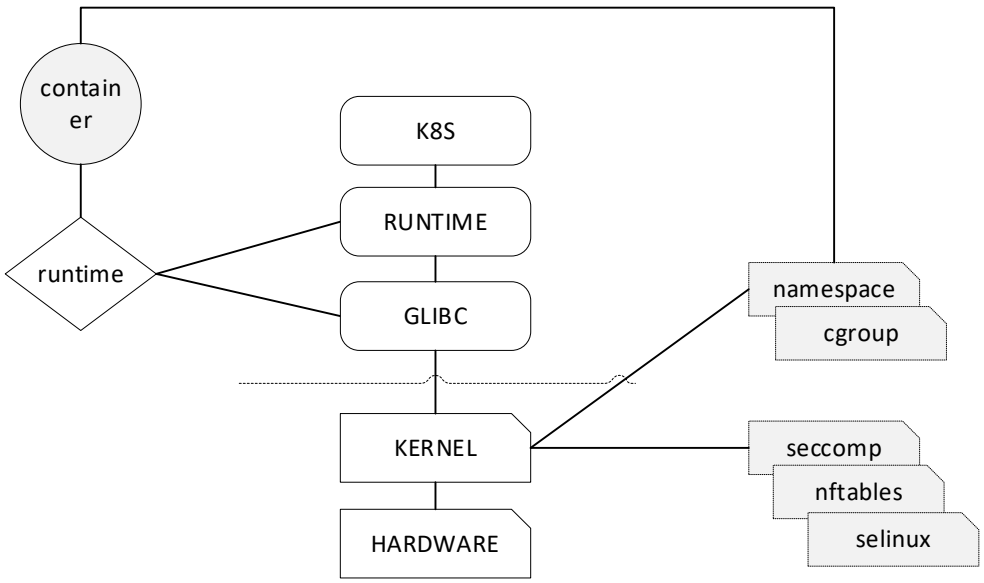


그림 7 컨테이너 및 기술 관계

또한 컨테이너 관련된 표준안이 만들어지면서, 도구도 아래처럼 몇 가지가 더 만들어졌다. 기존에 도커에서 사용하던 명령어를 아래와 같이 분리하여 독립적인 기능으로 만들었다.

- buildah
- podman
- skopeo

위의 도구는 반드시 요구사항은 아니지만, 컨테이너 시스템 운영 시 도움이 되는 도구이다.

[3] https://selinuxproject.org/page/Building_a_Basic_Policy

"buildah"[4]는, 기존 도커에서 이미지 생성 시 사용하였던 'docker build'부분을 독립적으로 분리한 도구이다. 현재는 "Dockerfile"에서 "Containerfile"로 변경이 되었다.

"skopeo"[5]는 docker search 혹은 inspect 를 통해서 확인하였던 이미지 명령어를 다운로드 필요없이 메타정보 기반으로 확인이 가능한 명령어이다. 이 명령어의 주요 목적인 이미지 복사 및 정보 확인 시 사용한다.

[4] https://buildah.io/
[5] https://github.com/containers/skopeo

Containerd

Containerd 는 CNCF 에서 시작하였으며 이 프로젝트는 윈도우 및 리눅스 양쪽에서 실행이 가능하다. 주요 목적은 컨테이너의 라이프 사이클 관리가 주요 목적이며 컨테이너에 저장소 및 네트워크 기능을 제공한다.

Containerd 는 도커의 하위 서비스인 dockerd 밑에서 containerd 라는 이름으로 동작하였지만, 지금은 도커에서 분리하여 독립적인 컨테이너 런타임으로 사용이 가능하다.

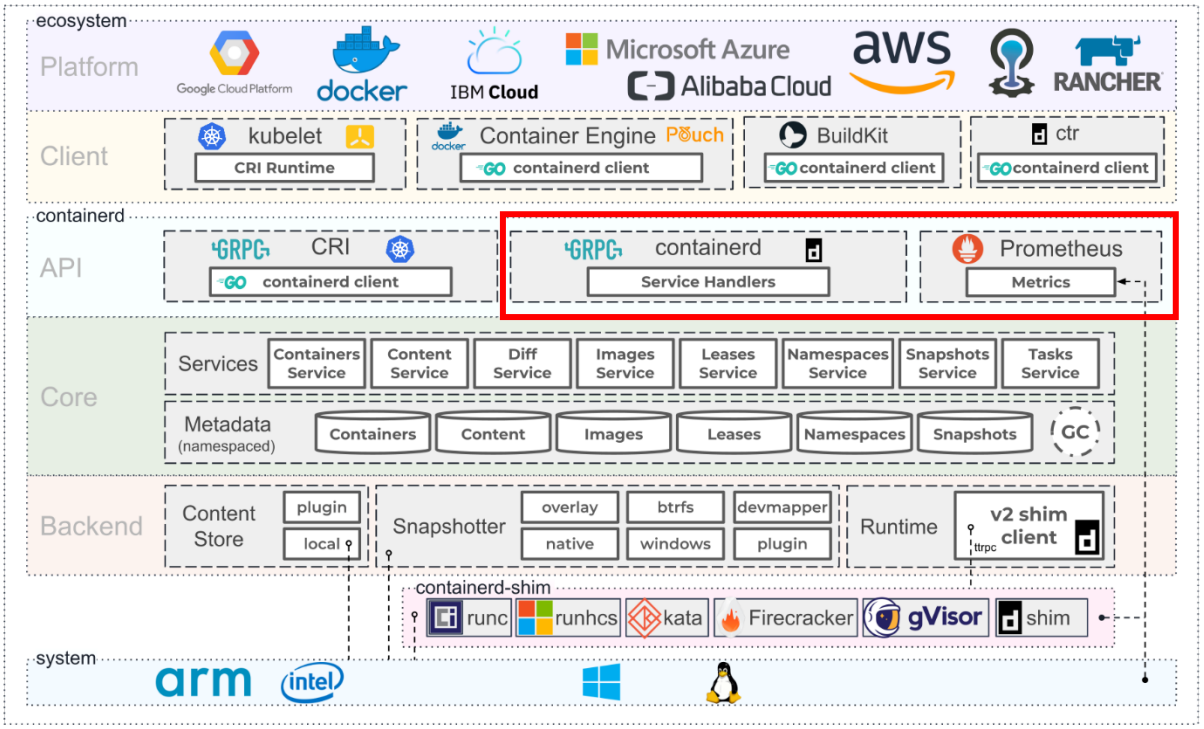
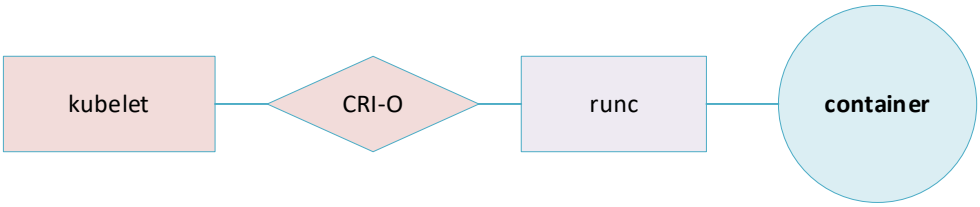


그림 8 containerd 계층

"containerd"는 CRI-O 처럼 runc-shim 기반으로 동작하며 OCI, CNI 와 같은 표준 사양을 준수한다. containerd 는 쿠버네티스에서 지원하는 런타임 중 하나이다.

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.



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

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.

```
# 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.

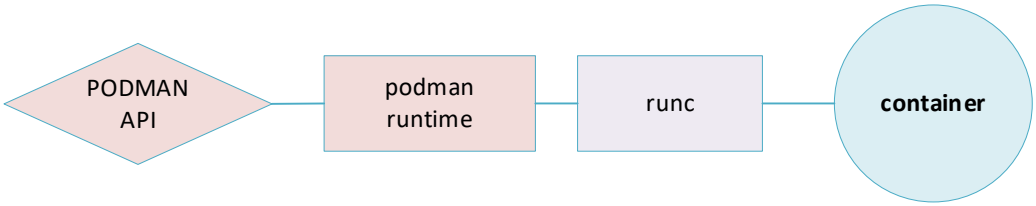


그림 9 podman 구조

리눅스 커널

파일시스템

UFS

UFS [6] (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 UFS 는 형태로 저장이 되며, 이 구조는 key=pair 형태로 구성이 되어 있다.

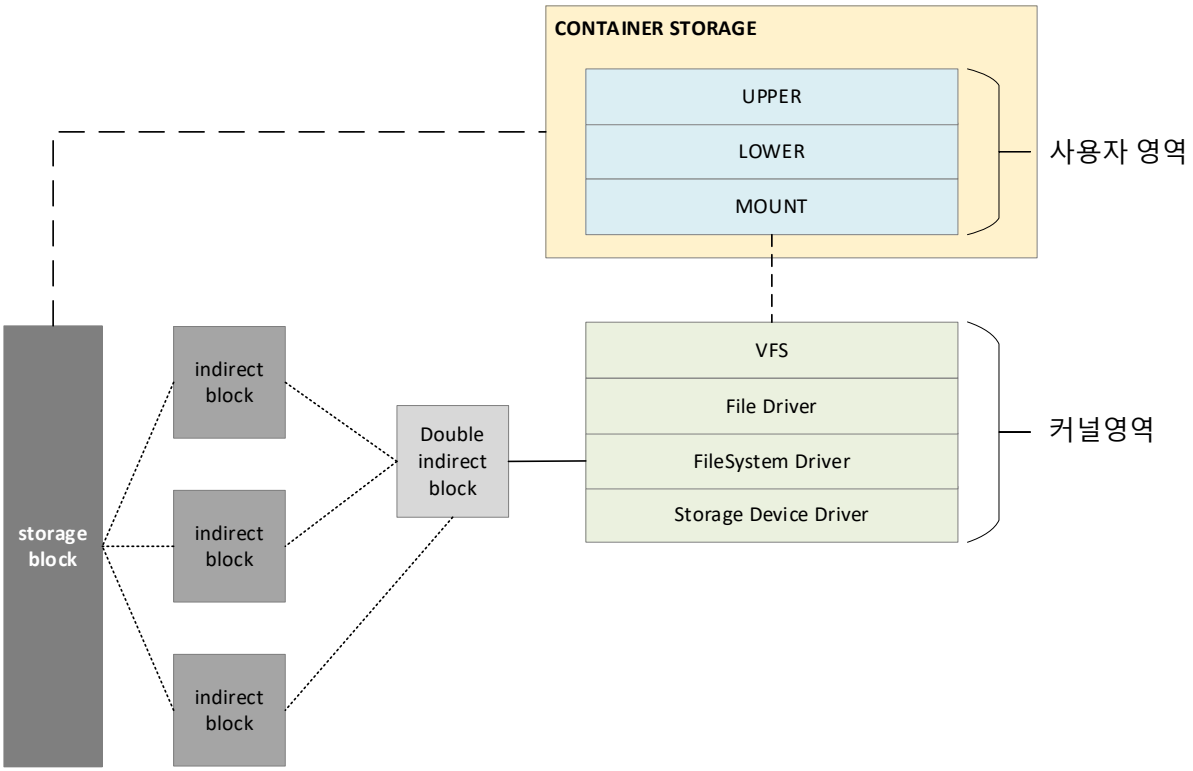


그림 10 UFS 레이어

[6] https://unionfs.filesystems.org/docs/zen/zen.html

OverlayFS (Filesystem)

OverlayFS[7](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.

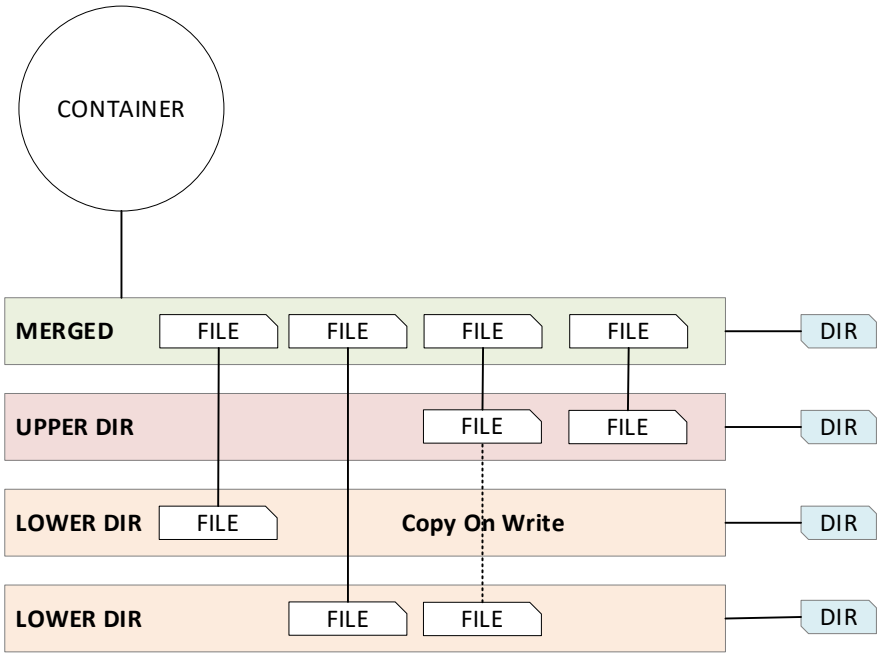


그림 11 OverlayFS 계층

현재 거의 대다수 컨테이너 시스템은 위와 같은 구조로 컨테이너 이미지를 구현하고 있다. UFS 경우에는 논리적 구성을 하드코딩 된 드라이버를 통해서 **유저/커널** 영역에서 장치를 구현한다. 이로써 완벽하게 드라이버를 구현한다는 장점이 있지만, 단점으로는 커널 영역을 사용하기 때문에 성능 부분에서는 OverlayFS 보다 느리다.

현재 거의 대다수 컨테이너의 파일 시스템은 OverlayFS 기반으로 사용하고 있고, "Fuse"기반으로 구성된 "Fuse OverlayFS"[8]를 더 많이 사용한다.

[7] https://wiki.lustre.org/images/8/8c/LUG2019-Lustre_2.12_In_Production-Thiell.pdf
[8] https://github.com/containers/fuse-overlayfs

THE NAMESPACE

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

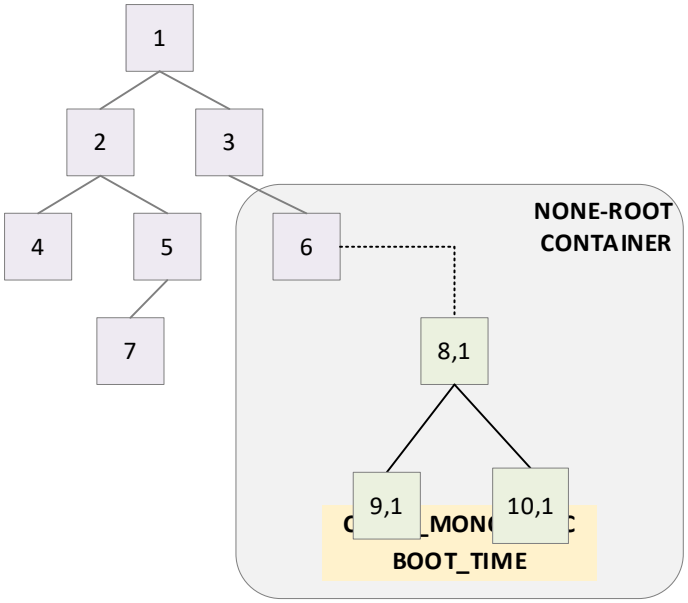


그림 12 namespace UTS clock

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

이전 네임스페이스는 다음과 같은 단점이 있었다.

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

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

C-GROUP

c-group 은 **C**ontrol **G**roup 의 약자이며, 실행되는 프로세스에 세밀하게 자원 사용량 제한 및 감사 기능을 제공한다. 2006 년에 구글에서 개발이 시작이 되었으며, 개발명은 **"process containers"**라는 이름으로

개발, 2007 년에 이름을 **cgroup** 으로 변경하였다. 정식 릴리즈는 2008 년도에 하였으며, 리눅스 커널 버전은 linux-2.6.24 에서 공식적으로 릴리즈가 되었다. cgroup 의 주요 기능은 다음과 같다.

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

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

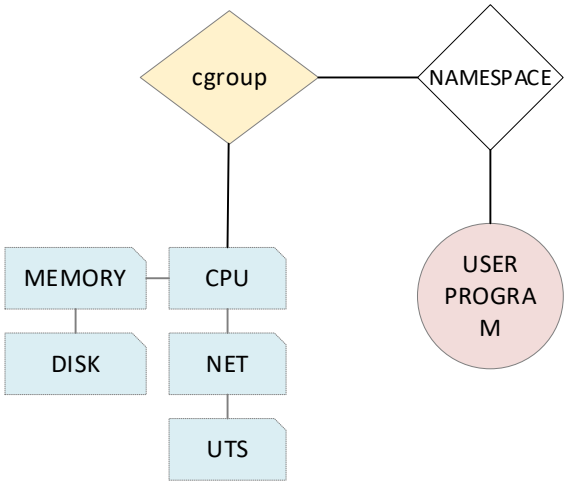


그림 13 네임스페이스

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

쿠버네티스 아키텍처 확인

가상화 vs 컨테이너

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

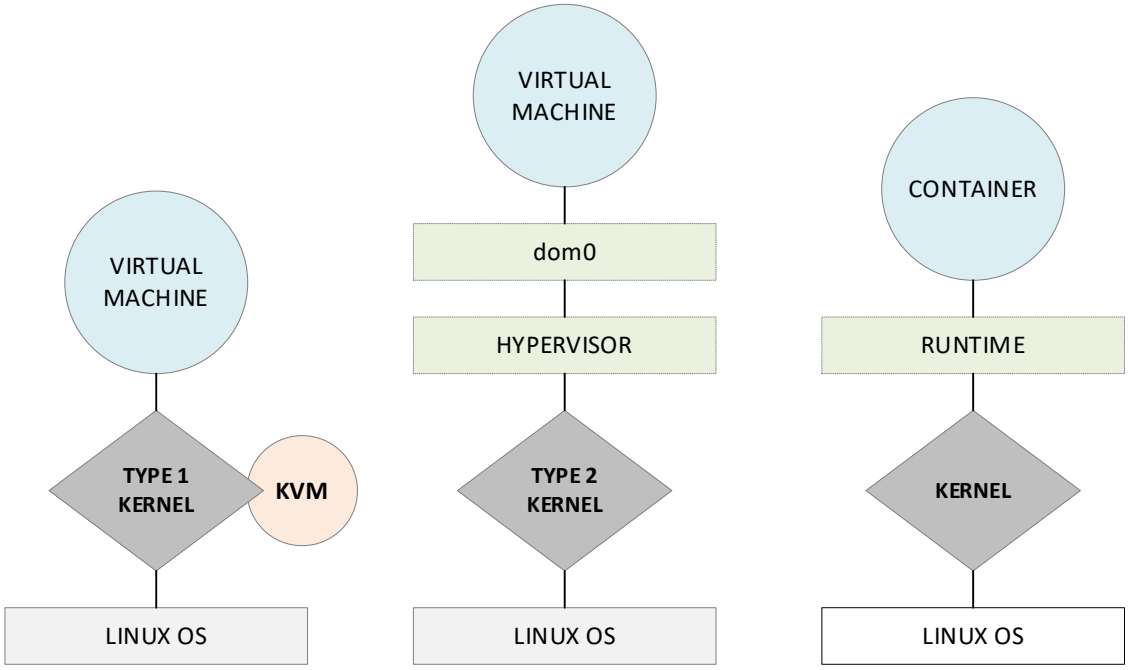


그림 14 컨테이너 및 하이퍼바이저 비교

가상머신과 컨테이너의 제일 큰 차이점은 **하드웨어의 가상화 기능사용** 여부이다. 기본적으로 쿠버네티스는 **컨테이너 런타임 기반**의 환경을 제공하기 때문에 하드웨어 기반의 가상화처럼 높은 하드웨어를 요구하지 않는다.

하지만, 컨테이너는 rootless 구조 및 ring structure 를 생성하지 않고 시스템 구조를 구현하기 때문에 가상머신과 비교하였을 때 기능적으로 제한이 있다. 그 부분은 아래에서 표로 정리하였다.

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

최근 쿠버네티는 "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.

RUNC

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

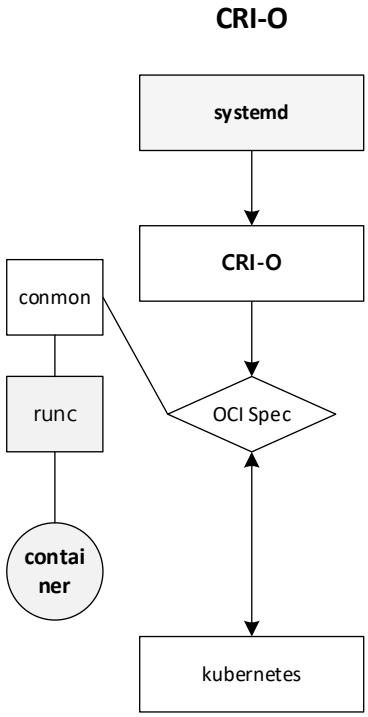


그림 15 RUNC(common)

생성된 컨테이너는 "conmon"이라는 프로세서가 컨테이너를 모니터링을 한다. 컨테이너는 conmon 에서 "-r"이라는 런타임 옵션을 통해서 'runc'기반으로 컨테이너 이미지를 불러와서 실행한다.

쿠버네티스 컴포넌트

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

쿠버네티스는 다음과 같은 프로그램으로 기본 서비스 구성이 되어 있다.

- kubelet
- kube-proxy
- kube-scheduler

- kube-controller-manager
- kube-apiserver

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

- pod(pause)
- node
- coredns
- etcd

kubelet

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

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

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

- TCP/6443
- HTTP/HTTPS

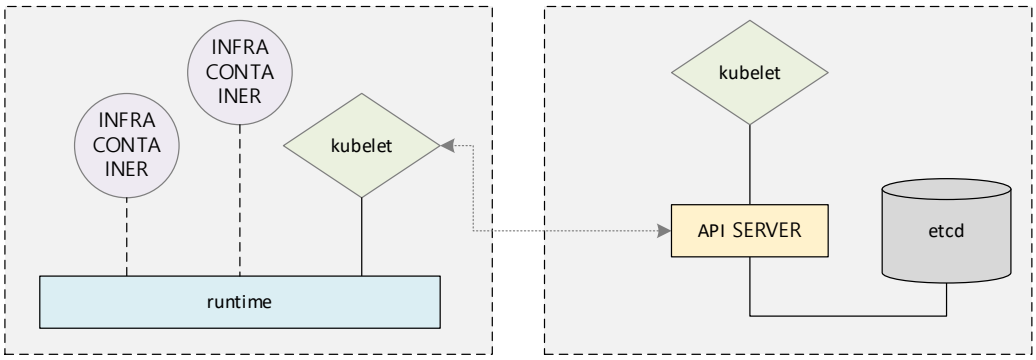


그림 16 kubelet 구조

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

```
# ss -antp | grep kubelet
```

```
LISTEN     0     128   127.0.0.1:10248                *:*
      users:(("kubelet",pid=519,fd=33))
LISTEN     0     128   127.0.0.1:39389                *:*
      users:(("kubelet",pid=519,fd=10))
ESTAB      0     0     192.168.122.110:47546          192.168.122.110:6443
      users:(("kubelet",pid=519,fd=16))
LISTEN     0     128         [::]:10250               [::]:*
      users:(("kubelet",pid=519,fd=29))
```

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

이름	호스트	컨테이너
kubelet	네	아니요
kubeproxy	네	아니요

POD/Pause

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

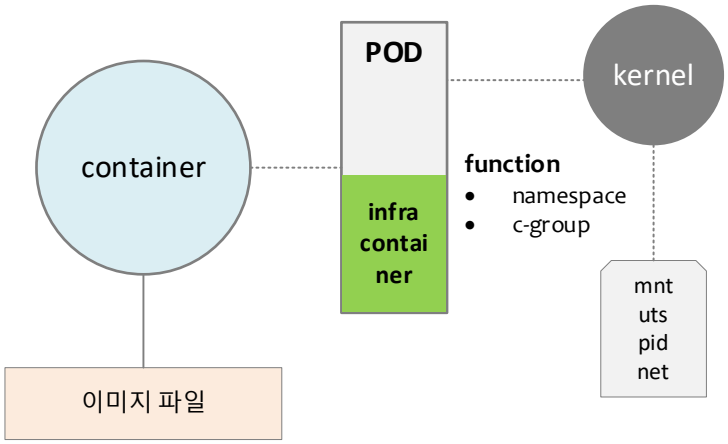


그림 17 POD

마스터 및 워커 노드 서비스

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

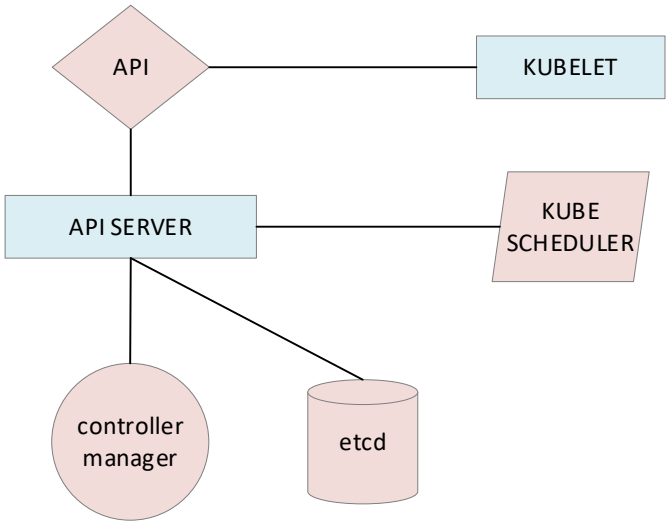


그림 18 마스터 노드

etcd

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

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

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

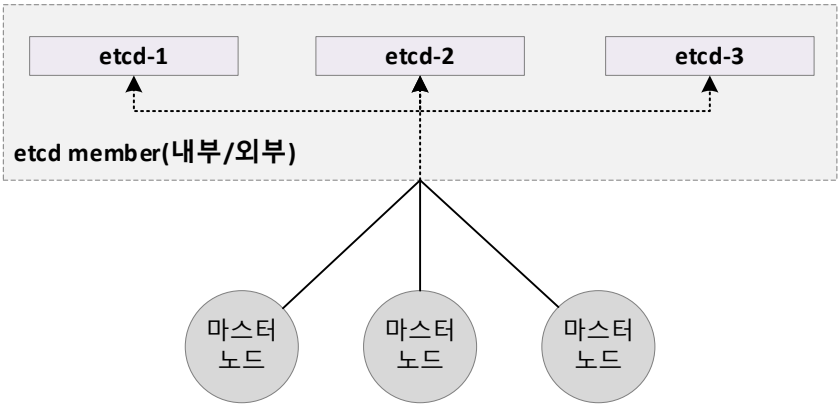


그림 19 클러스터 멤버

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

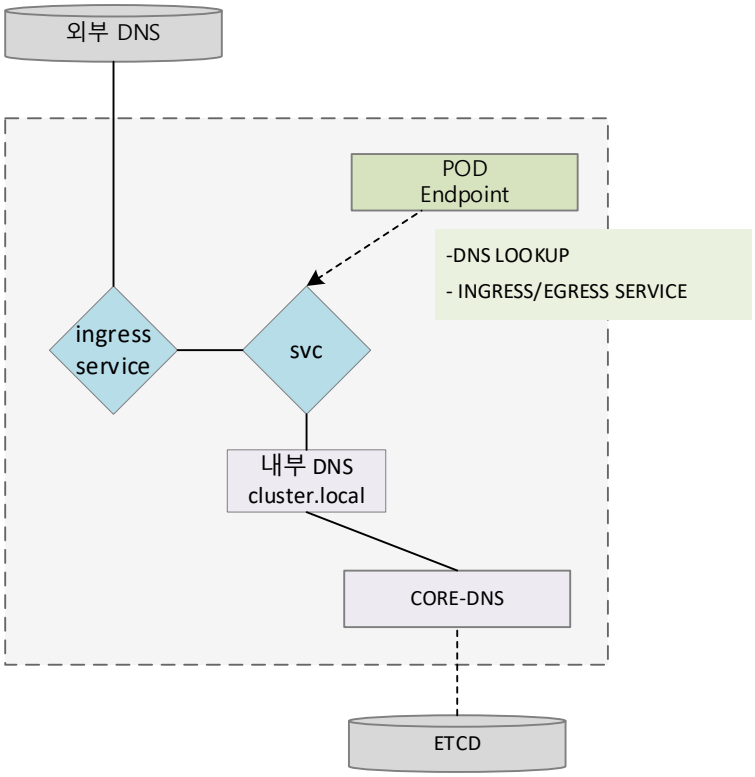
```
# ADVERTISE_URL=https://192.168.90.110:2379
# kubectl exec etcd-master.example.com -n kube-system -- sh -c \
"ETCDCTL_API=3 etcdctl \
--endpoints $ADVERTISE_URL \
--cacert /etc/kubernetes/pki/etcd/ca.crt \
--key /etc/kubernetes/pki/etcd/server.key \
--cert /etc/kubernetes/pki/etcd/server.crt \
get \"\" --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.

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

coredns

coredns 는 포드 및 서비스에서 사용하는 내부 도메인 작업을 처리하는데 사용한다. 구성 방법에 따라서 다르지만, 외부 도메인 및 내부 서비스를 도메인 기반으로 처리시 사용한다.



Coredns 는 기본적으로 내부 클러스터 서비스를 위해서 사용하며, 여러 서비스를 하나로 묶기 위한 방법으로 사용한다. 설치시 별도로 도메인을 설정하지 않았으면, cluster.local 로 되어있다.

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

```
# kubectl exec -i -t dnsutils -- nslookup kubernetes.default
```

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

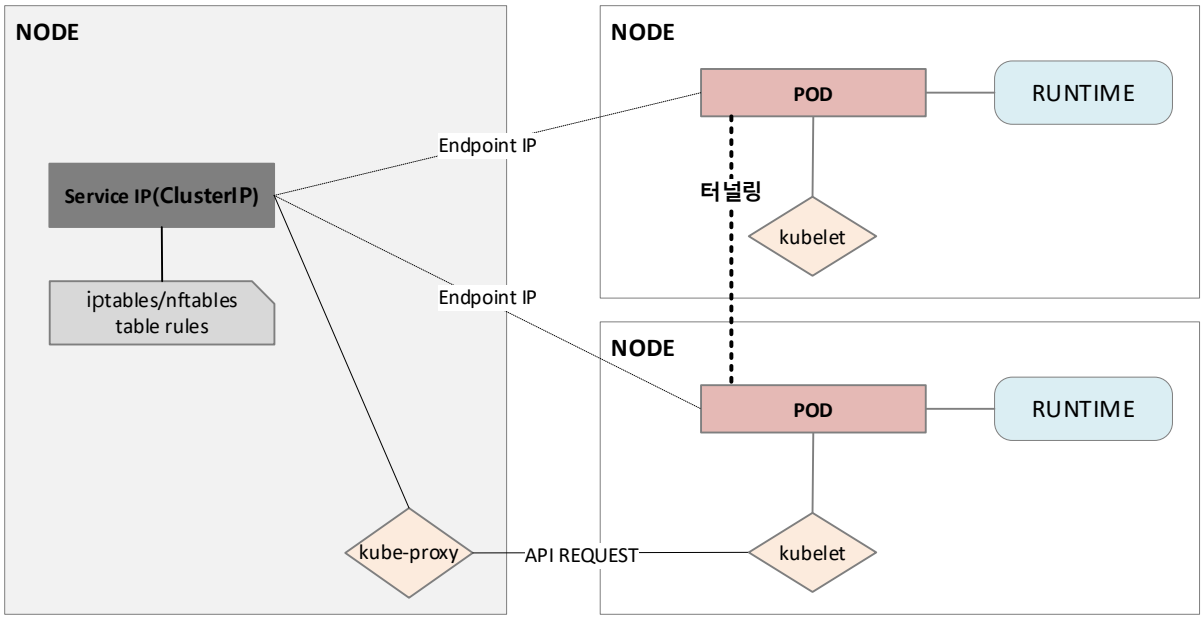
```
search default.svc.cluster.local svc.cluster.local cluster.local
google.internal c.gce_project_id.internal
nameserver 10.0.0.10
options ndots:5
```

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

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

kube-proxy(master, worker)

kube-proxy 서비스는 쿠버네티스에 구성이 되어있는 애플리케이션 서비스에 접근할 수 있도록 지원해주는 구성원입니다.




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

node] # crictl ps
f1d1848534a82
beb86f5d8e6cd2234ca24649b74bd10e1e12446764560a3804d85dd6815d0a18   57 minutes
ago      Running             kube-proxy                0
fcd6e90a09cfd       kube-proxy-qvd5k
```

kube-proxy 가 구성한 내용을 확인하려면, 노드에서 아래 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
                ct state new  counter packets 48 bytes 5218 jump KUBE-
EXTERNAL-SERVICES
                counter packets 58183 bytes 106276950 jump KUBE-FIREWALL
        }
```

kube-scheduler

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

포드 구성 시 쿠버네티스 스케줄러는 다음 조건을 먼저 확인한다.

- 필터링(filtering)
- 점수(scoring)

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

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

최종적으로 단계를 검증하면 다음과 같다.

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

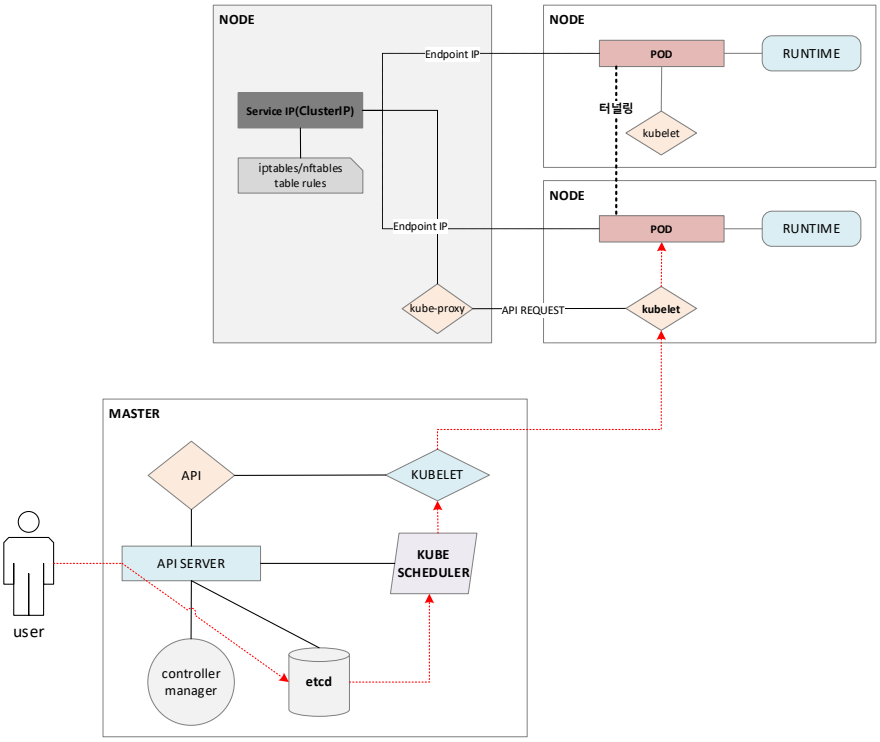

NODE
Endpoint IP
Service IP(ClusterIP)
iptables/nftables
table rules
Endpoint IP
kube-proxy
API REQUEST
NODE
POD
RUNTIME
터널링
kubelet
NODE
POD
RUNTIME
kubelet
MASTER
API
KUBELET
API SERVER
KUBE
SCHEDULER
controller
manager
etcd
user

그림 20 스케줄러

kube-controller-manager

컨트롤러 매니저는 스케줄러와 비슷하게 보이지만, 조금 다른 역할을 한다. 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, apiserver 를 통해서 상태 확인 혹은 변경이다. kube-controller-manager 를 대표적으로 많이 사용하는 서비스는 replication controller, endpoint controller, namespace controller 그리고 serviceaccount controller 이다.

컨트롤 매니저가 요청을 받으면 다음과 같은 방법으로 자원을 생성 및 구성한다.

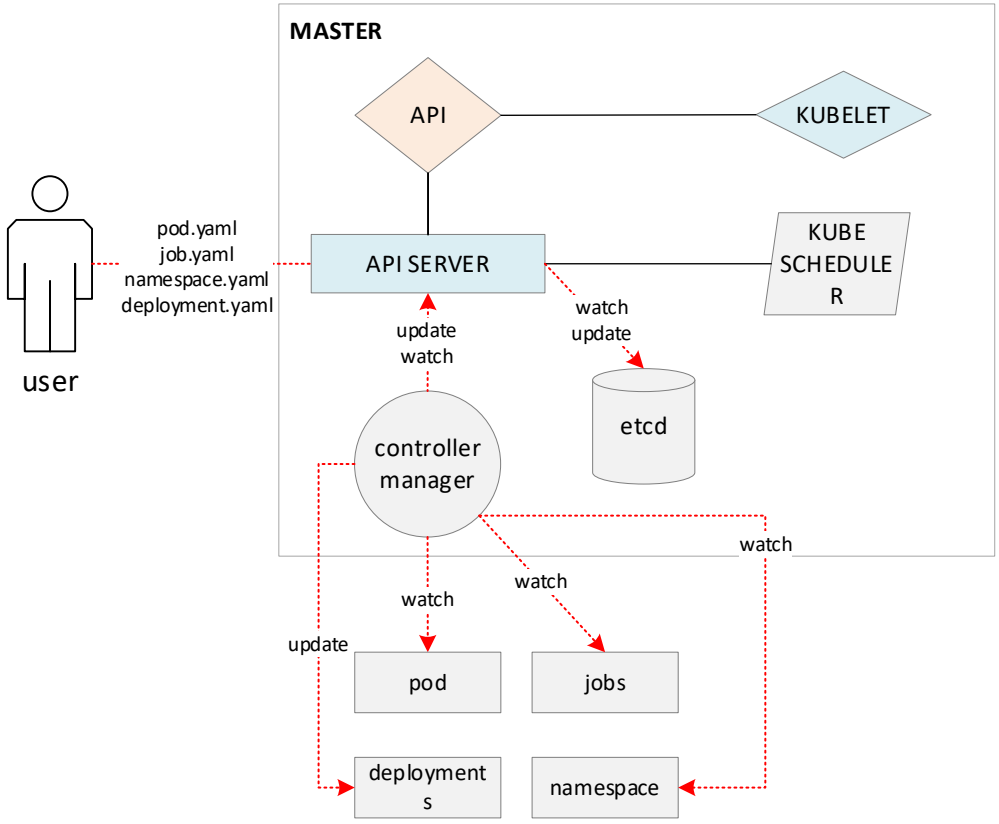


그림 21 컨트롤 매니저 자원 생성

kube-apiserver

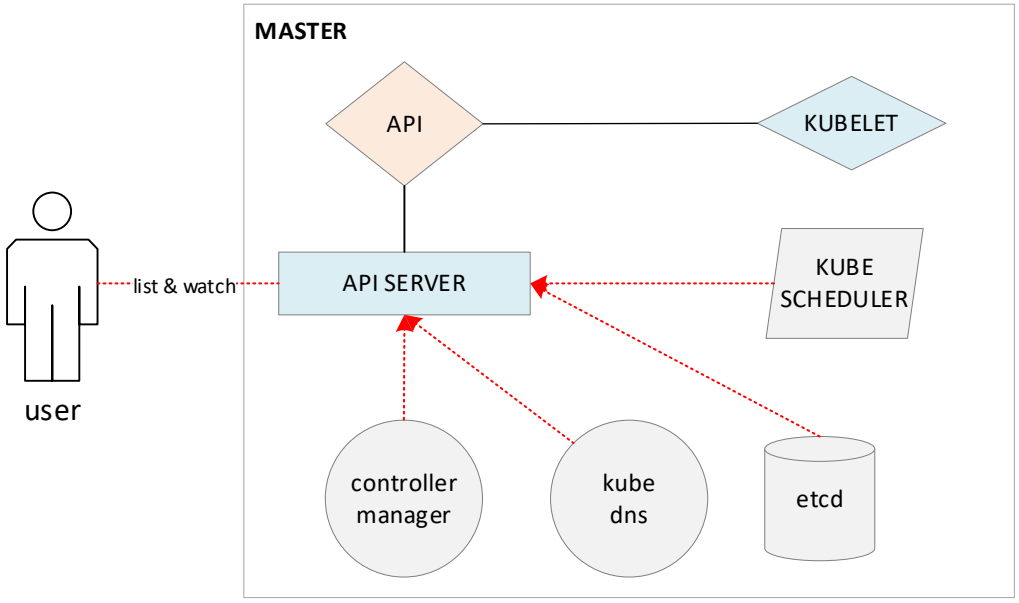


그림 22 apiserver

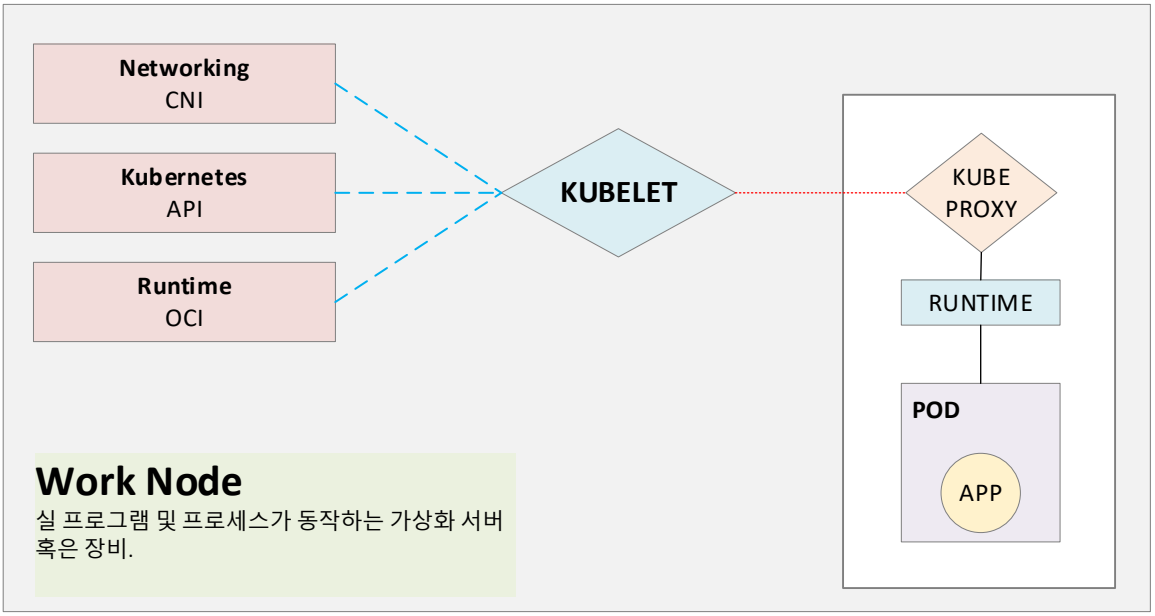
외부에서 들어오는 요청은 API 서버를 통해서 자원 처리를 한다. kubectl 이나 혹은 쿠버네티스 서비스에서 요청이 들어오면 API 맞에 자원을 핸들링 한다.

각각 자원은 컨트롤 매니저를 통해서 Pod 같은 자원을 처리하며, 수행이 된 작업은 etcd 서버에 상태 정보를 저장한다.

- `/api/v1/namespaces`
- `/api/v1/pods`
- `/api/v1/namespaces/my-namespace/pods`
- `/apis/apps/v1/deployments`
- `/apis/apps/v1/namespaces/my-namespace/deployments`
- `/apis/apps/v1/namespaces/my-namespace/deployments/my-deployment`

워커노드(혹은 미니언)

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



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

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

스왑 사용 금지

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

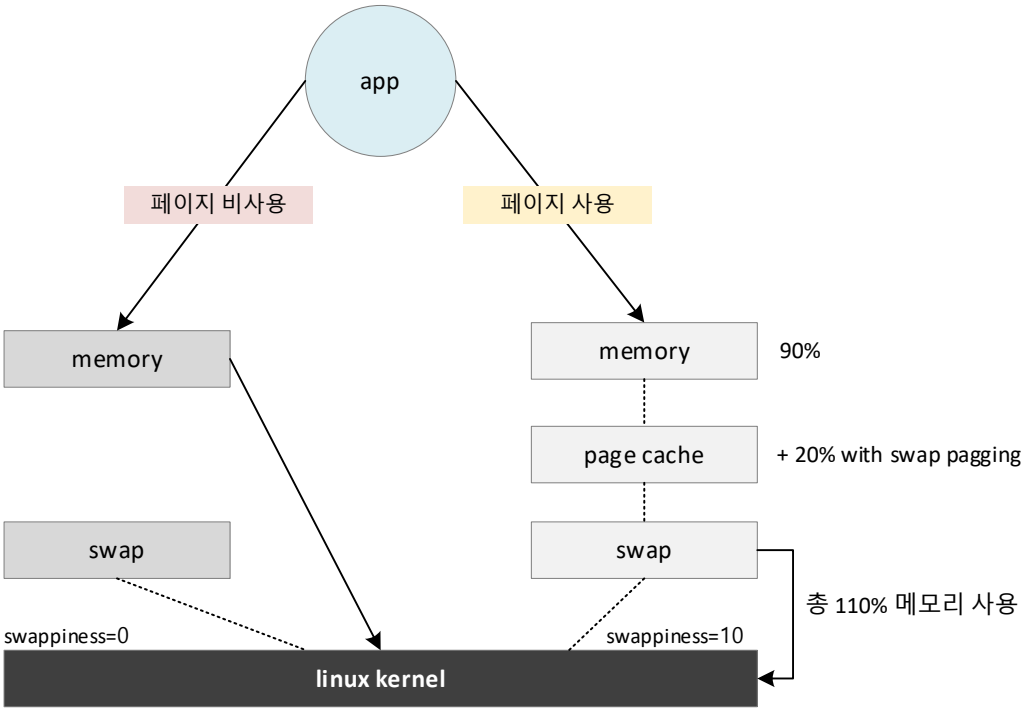


그림 23 스왑

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

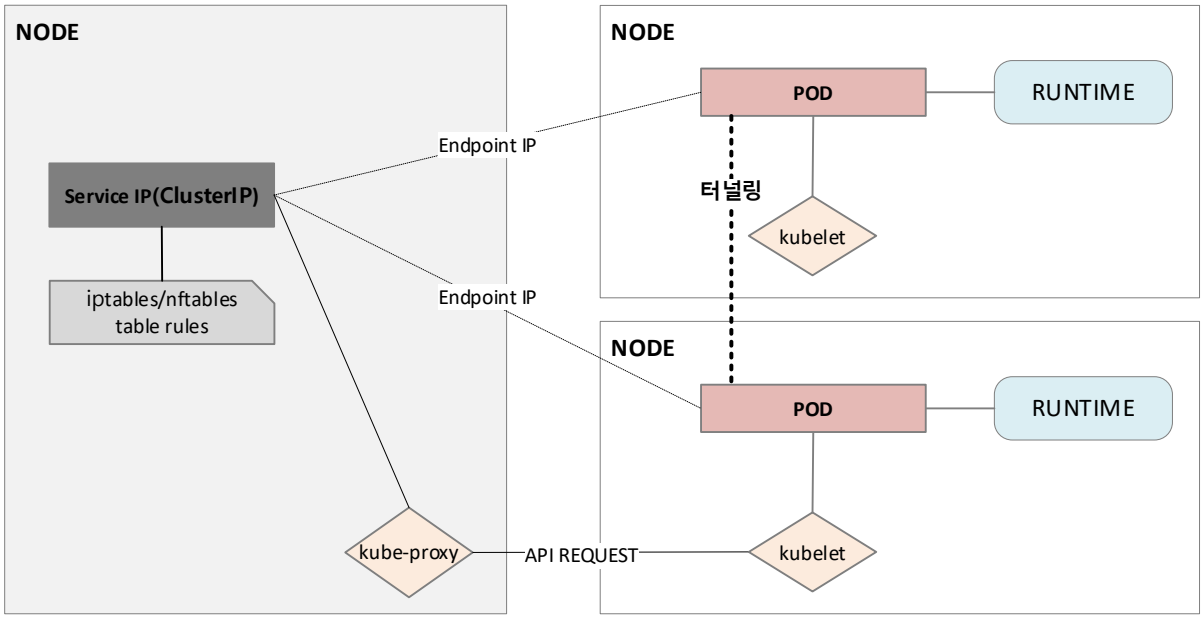
```
# swapoff -a
# sysctl vm.swappiness=0
```

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

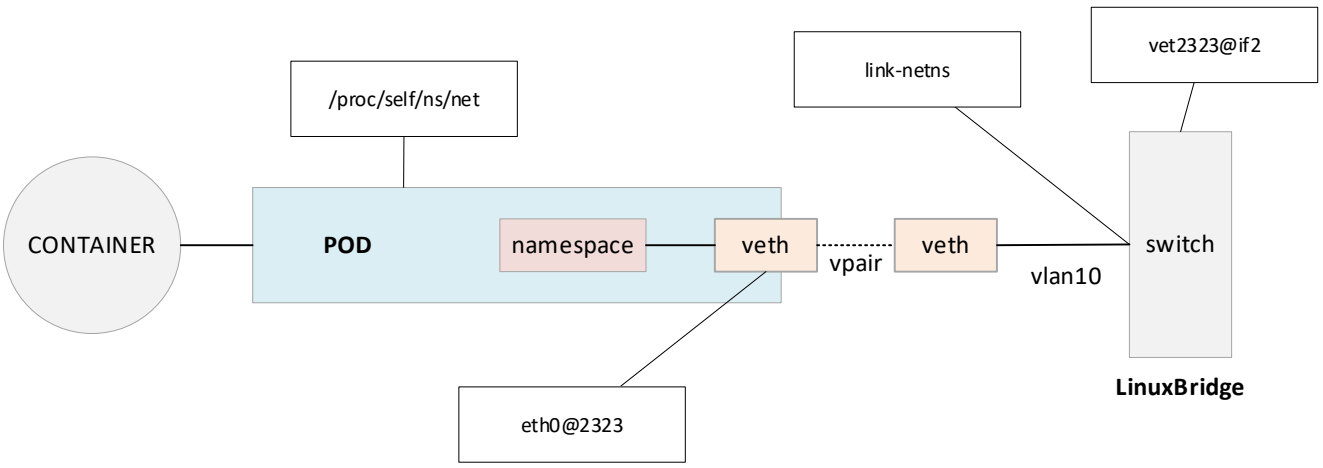
쿠버네티스 네트워크

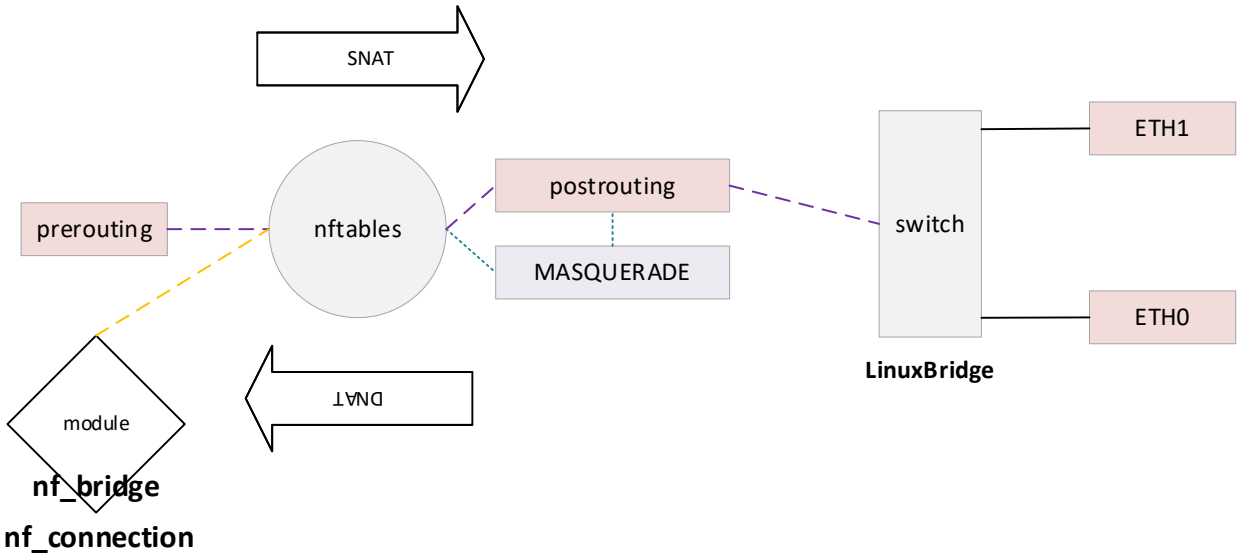
구조

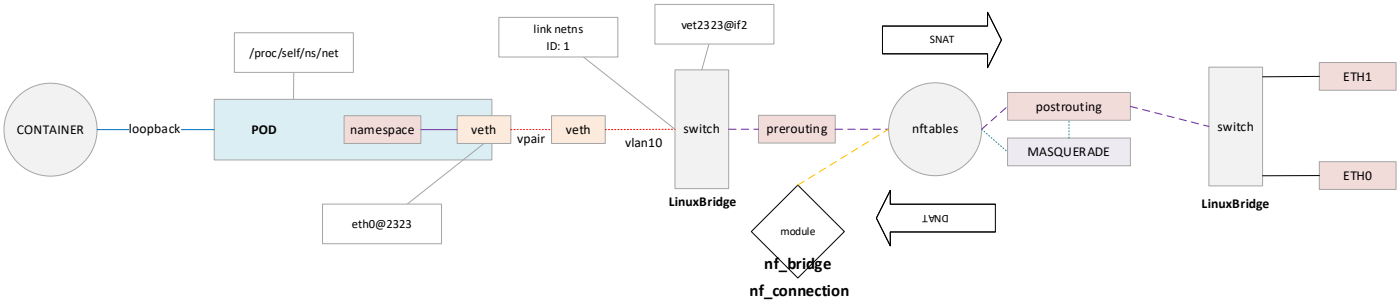
클러스터가 구성이 되면 쿠버네티스는 아래와 그림 같은 구조를 가지고 있다.



Pod 는 Pod Network 를 통해서 서로 연결 및 구성이 되며, 이걸 터널링이라고 부른다. 터널링이 구성이 되면 각각 노드에 떨어져 있는 Pod 는 Pod Network 를 통해서 각각 독립적인 네트워크를 구성한다. Pod 가 구성이 되면, Pod 는 리눅스 브릿지(Linux Bridge)에 연결이 된다.







kubectl describe pod/nginx-server | grep Node

```
kubectl get pods,svc -o wide

conntrack -L, present connection tracking
conntrack -E, continuously new connection tracking
conntrack -L -d <IP>, the IP address tracking
dnf install ipvsadm -y
ipvsadm -Ln
ipvsadm -Ln -t 100.64.0.10:53
```

주요 쿠버네티스 자원설명

Pod(po)

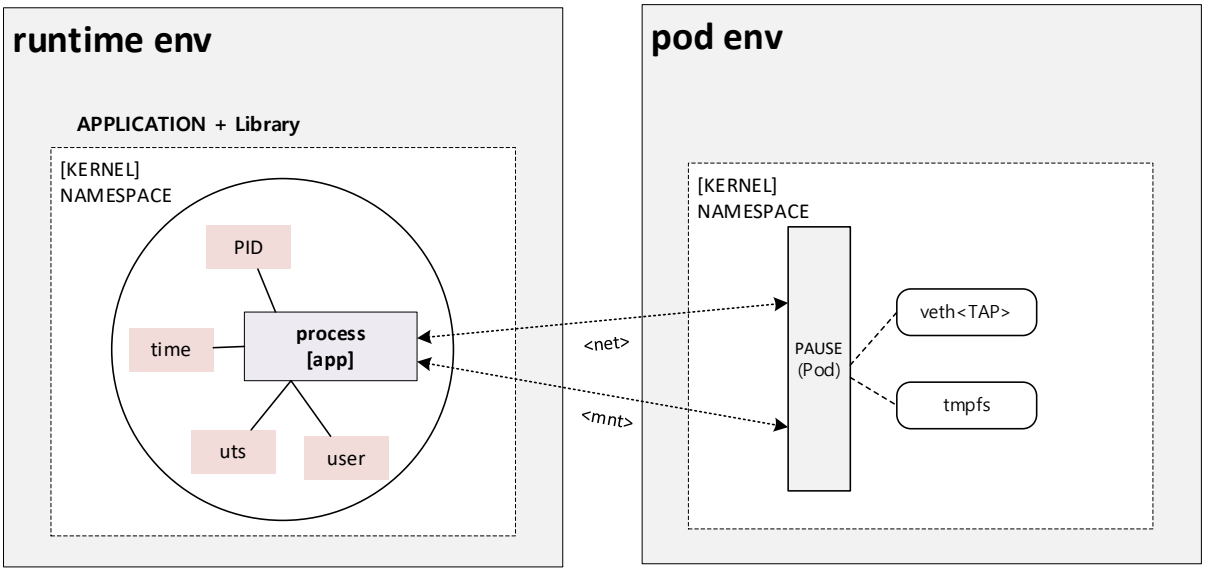


그림 24 POD

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

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

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

Pod 는 생성이 되면 Pod IP 및 도메인을 하나 생성한다.

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

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

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

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

Service(svc)

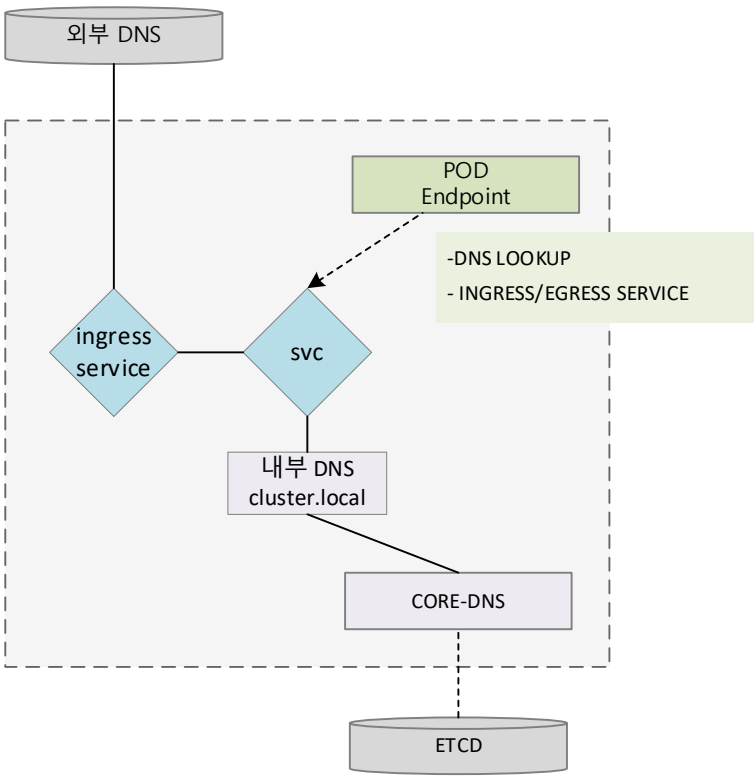


그림 25 POD 및 네트워크

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

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

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

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

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

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

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

옵션	설명
`--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)

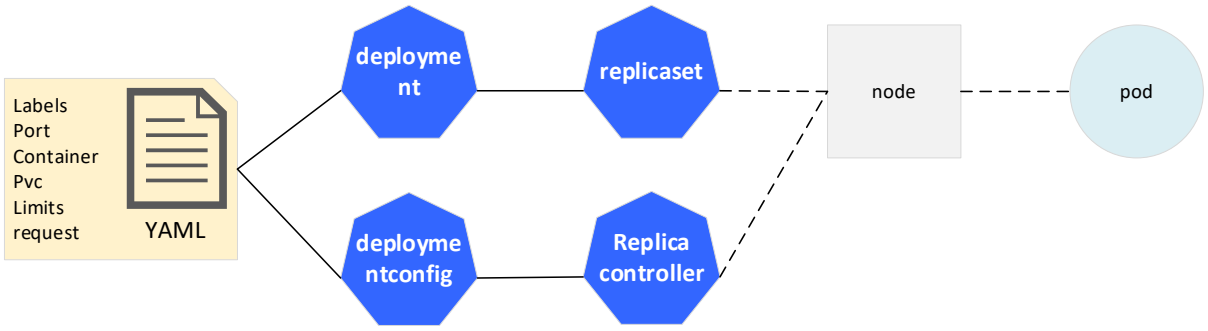


그림 26 Deployment

Deployment 는 Relicaset 에서 사용하는 설정 정보들이 있다. Deployment 에는 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
```

[상단 설명]

[하단설명]

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

```
template:
    metadata:
      labels:
        app: nginx
```

```
spec:
      containers:
      - name: nginx
```

```
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

[전체설명]

```
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
```

[기능설명]

이를 통해서 Pod 의 롤아웃(rollout) 및 Pod 관리가 가능하다. 사용량이 늘어나는 경우 Deployment 는 RS(ReplicaSet)를 통해서 Pod 를 확장한다. 또한 새로운 정보 혹은 Pod 생성이 요청이 되면, 오래된 Pod 를

제거하면서 새로운 RS 기반으로 Pod 를 생성한다. 문제가 발생시, 해당 시점으로 롤백(roll-back)이 가능하다.

YAML 파일작성

```
# kubectl run nginx - image=nginx -- port=8080 --env=env=DEV --
labels=app=nginx,owner=user --privileged=false --dry-run=client -oyaml >
nginx-pod.yaml
```

```
# kubectl create deployment my-dep --image=nginx --dry-run=client -oyaml >
deployment.yaml
```

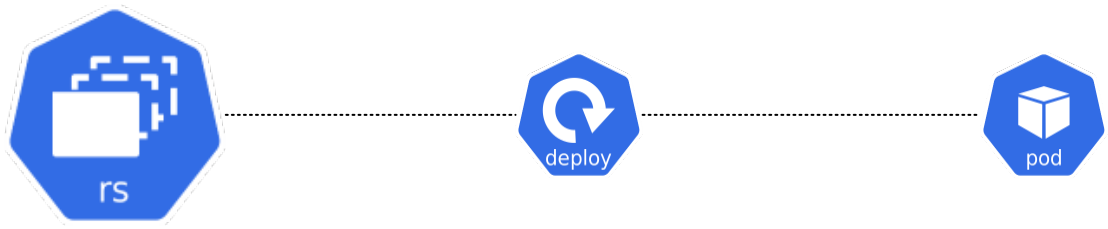
DeploymentConfig(Deprecated, dc)

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

Deployment 와 비슷하게 레이블 즉, 셀렉터 기반으로 사용이 가능하지만, 단순한 셀렉터 기반으로만 사용이 가능하고 복잡한 셀렉터 기능은 사용이 불가능 하다.

현재 쿠버네티스에서는 더 이상 DeploymentConfig(혹은 dc)는 더 이상 사용하지 않는다.

ReplicaSet(rs)



"ReplicaSet", 이하 복제자. 기존의 Replication Controller 의 단점을 보안 및 보강하기 위해서 나온 기능이다. 이 자원은 보통 "rs"라고 많이 부르며, 기존에 사용하던 ReplicationController 와 다르게

레이블 및 템플릿 같은 메타정보 기반으로 사용한다. HPA(Horizontal Pod Autoscaler)같은 기능을 사용하기 위해서는 ReplicaSet 를 통해서 구성이 된다.

간단하게 구성하기 위한 예제는 다음과 같다. 이름은 "replicaset.yaml"으로 저장한다.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
```

Replication Controller(rc)

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

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

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

하지만, Replication Controller 는 구형 버전으로 분류가 되어서 대다수 기능들은 Replication Controller 보다는 ReplicaSet 기반으로 구성이 된다. 최근에 릴리즈 된 쿠버네티스는 "Deployment" 기반으로 사용하기 때문에, Replication Controller 로 생성하는 경우, DeploymentConfig 에 저장이 되는데, 이 자원은 더 이상 존재하지 않기 때문에, 애플리케이션에 대한 설정 정보(DeploymentConfig)는 저장이 되지 않는다. 간단하게 만들어 보도록 한다. 파일 이름은 "replication.yaml"으로 지정한다.

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

명령어 쉽게 사용하기

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

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

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

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

```
# yum install bash-completion
```

설치가 완료가 되면 다시 다음과 같이 명령어를 실행한다.

```
# source ~/.bash_profile
```

위의 기능이 올바르게 설치가 되면 다음처럼 명령어를 테스트한다.

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

노드 상태 확인

쿠버네티스의 전체적인 상태를 확인하는 명령어는 다음과 같다.

```
$ kubectl get componentstatuses
NAME                 STATUS     MESSAGE          ERROR
controller-manager   Healthy    ok
scheduler            Healthy    ok
etcd-0               Healthy    {"health":"true"}

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

노드 제거

마스터 노드 제거하기

```
kubectl cordon
kubectl drain --force --ignore-daemonsets
kubectl delete node
```

워커노드 제거하기

```
kubectl cordon
kubectl drain --force --ignore-daemonsets
kubectl delete node
```

네임스페이스 확인

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
```

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
```

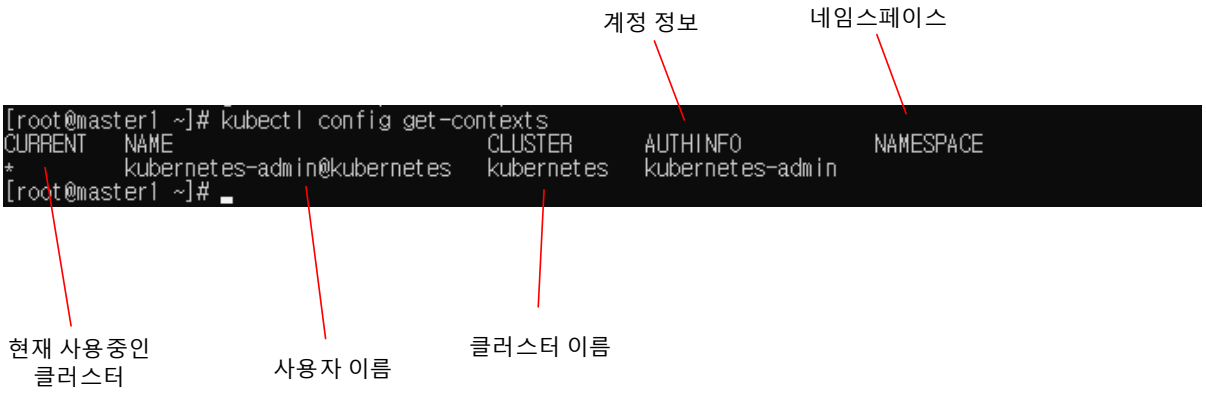
생성된 네임스페이스 **"first-namespace"**는 다음 명령어로 확인이 가능하다.

```
$ 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
```

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

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

위의 내용을 좀 더 자세히 설명하자면 다음과 같다.



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

```
$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   5d6h
first-namespace    Active    9m23s
kube-node-lease    Active    5d6h
kube-public       Active   5d6h
kube-system       Active   5d6h
nginx-ingress     Active   5d6h

$ kubectl config set-context first-namespace
Context "first-namespace" created.
```

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

현재 사용중인 네임스페이스를 확인하기 위해서는 다음과 같은 명령어로 확인이 가능하다.

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

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

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

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

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

특정 네임스페이스만 지정해서 사용하려면, 다음과 같은 명령어로 노드에서 실행한다.

```
$ 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. 이 방법이 귀찮은 경우, OKD 의 oc 명령어 기반으로 사용하여도 문제없다.

아래 명령어는 현재 사용중인 네임스페이스에서 pod 를 확인하는 명령어이다.

```
$ kubectl get pods
No resources found in first-namespace namespace.
```

아래 명령어는 모든 네임 스페이스에서 사용하는 pod 를 확인하는 명령어이다.

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

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

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

```
$ kubectl get pods --all-namespaces --show-labels
NAMESPACE        NAME                                       READY
STATUS     RESTARTS   AGE   LABELS
kube-system      calico-kube-controllers-6fcbbfb6fb-4jw2d   1/1
      Running   3           5d6h   k8s-app=calico-kube-
controllers,pod-template-hash=6fcbbfb6fb
kube-system      calico-node-9rcpx                          1/1
      Running   3           5d6h   controller-revision-
hash=6c8654f747,k8s-app=calico-node,pod-template-generation=1
kube-system      calico-node-kgpcp                          1/1
      Running   3           5d6h   controller-revision-
hash=6c8654f747,k8s-app=calico-node,pod-template-generation=1
kube-system      calico-node-p7tng                          1/1
      Running   3           5d6h   controller-revision-
hash=6c8654f747,k8s-app=calico-node,pod-template-generation=1
kube-system      coredns-66bff467f8-2js8j                   1/1
      Running   3           5d6h   k8s-app=kube-dns,pod-template-
hash=66bff467f8
kube-system      coredns-66bff467f8-kn8fc                   1/1
      Running   4           5d6h   k8s-app=kube-dns,pod-template-
hash=66bff467f8
kube-system      etcd-master-1.example.com                  1/1
      Running   3           5d6h   component=etcd,tier=control-
plane
```

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

```
$ kubectl get pods --all-namespaces --show-labels \
--sort-by=.metadata.name
```

```
NAMESPACE        NAME                                       READY
STATUS     RESTARTS   AGE   LABELS
kube-system      calico-kube-controllers-6fcbbfb6fb-4jw2d   1/1
     Running   3          5d6h   k8s-app=calico-kube-
controllers,pod-template-hash=6fcbbfb6fb
kube-system      calico-node-9rcpx                          1/1
     Running   3          5d6h   controller-revision-
hash=6c8654f747,k8s-app=calico-node,pod-template-generation=1
kube-system      calico-node-kgpcp                          1/1
     Running   3          5d6h   controller-revision-
hash=6c8654f747,k8s-app=calico-node,pod-template-generation=1
kube-system      calico-node-p7tng                          1/1
     Running   3          5d6h   controller-revision-
hash=6c8654f747,k8s-app=calico-node,pod-template-generation=1
kube-system      coredns-66bff467f8-2js8j                   1/1
     Running   3          5d6h   k8s-app=kube-dns,pod-template-
hash=66bff467f8
```

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

마지막으로, 특정 노드나 혹은 자원에 대한 상태정보를 확인하고 싶은 경우, 다음과 같은 하위 명령어로 확인이 가능하다.

```
$ kubectl describe nodes/master.example.com
Name:               master-1.example.com
Roles:              master
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=master-1.example.com
                    kubernetes.io/os=linux
                    node-role.kubernetes.io/master=
Annotations:        kubeadm.alpha.kubernetes.io/cri-socket:
```

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

```
$ kubectl describe pod/nginx-ingress-rshwt --namespace=nginx-
ingress
Name:       nginx-ingress-rshwt
Namespace:  nginx-ingress
Priority:   0
Node:       node1.example.com/192.168.90.140
Start Time:   Mon, 04 May 2020 17:41:50 +0900
Labels:     app=nginx-ingress
            controller-revision-hash=7cd97644f4
            pod-template-generation=1
Annotations:  cni.projectcalico.org/podIP: 10.244.17.68/32
            cni.projectcalico.org/podIPs: 10.244.17.68/32
Status:     Running
IP:         10.244.17.68
IPs:
  IP:           10.244.17.68
Controlled By:  DaemonSet/nginx-ingress
Containers:
  nginx-ingress:
      Container ID:
docker://6f2b0ddc63854b02a7adf7fafd4d3c76715c9dd0ccab63f45ea38ea6ca
a943f1
      Image:      nginx/nginx-ingress:edge
```

연습문제

기존에 만들었던 YAML 파일기반으로 다음처럼 수정 후 결과를 확인한다.

문제 1

- 쿠버네티스에서 네임스페이스를 "basic”라는 이름으로 생성한다.
- 생성된 "basic” 네임스페이스를 기본 네임스페이스로 설정한다.
- 올바르게 생성이 되면 kubectl get pods 그리고 kubectl config current-context 명령어로 올바르게 전환이 되었는지 확인한다.

문제 2

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

YAML 문법 및 작성 방법

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

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

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

위에 구성이 되어있는 자원 분류로 쿠버네티스 자원은 생성 및 구별이 된다. 위의 자원들을 사용하기 위해서 쿠버네티스 명령어로만 구성하기는 어렵다. 쿠버네티스의 대다수 자원은 YAML 기반으로 작성을 해서 생성을 해야 되며, 명령어로 생성이 가능한 자원은 거의 없다. 자원을 생성하기 위한 YAML 문법은 아래와 같다.

YAML 문법

쿠버네티스는 다음과 같은 문법을 가지고 있다.

- Key Value Pair
- Array 그리고 List
- Dictionary 그리고 Map

YAML 의 문법은 들여쓰기 및 띄어쓰기에 민감하게 되어 있기 때문에 반드시 작성시 다음과 같은 조건을 가지고 작성해야 한다.

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

```
---
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
```

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
```

먼저 맨 위의 **"---"**는 YAML 문법의 시작을 알리는 부분이다. 이 부분은 생략하여도 되지만, 일반적으로 맨 상단에 관습적'으로 표시한다. **"name:"**부분은 실제로 쿠버네티스에서 사용하는 자원들의 이름, 예를 들어서 컨테이너/POD 의 이름이다. 이 이름을 통해서 어떤 애플리케이션 혹은 POD 동작하는지 확인이 가능하다. 미리 알아 두면 좋은 부분은 POD 는 애플리케이션이 아니며, "컨테이너"는 애플리케이션 동작하는 영역이다.

그 다음에 있는 **"apiVersion"**부분이다. 릴리즈 버전이나 그리고 용도에 따라 다른데, 현재는 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
```

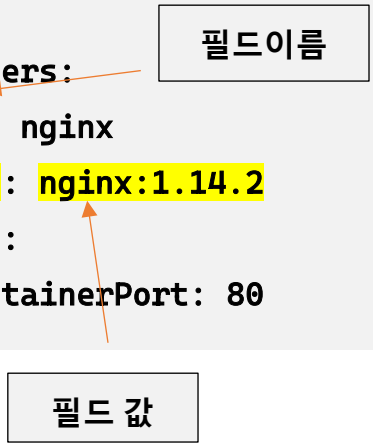
"label:"세션은 메타 정보의 "select:"에서 사용하기 위해서 존재한다. selector 에서 바라보고 있는 자원의 이름은 항상 labels 에서 명시가 되어있는 값을 찾게 된다. 아래 예제를 참고한다.

```
spec:
```

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

“spec:”에 복제자(replicas)가 사용할 개수 그리고 복사할 대상이 "matchLabels"라는 영역에 선언이 되어 있다. 선택할 POD 이름은 “app” 그리고 값은 “nginx”이다.

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



하단에 보면 **“containers:”**라는 지시자가 보이는데, 이를 통해서 어떤 컨테이너 이미지를 사용하여 서비스를 구성할 지 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 에 등록된 순서대로 접근하여 이미지를 검색한다.

```
    ports:
    - containerPort: 80
```

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

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

연습문제

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

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

쿠버네티스 컨텍스트 및 사용자 구성

이 책에서는 중요한 부분은 아니지만, 어떠한 방식으로 사용자 및 권한을 관리하는지 확인을 한다. 쿠버네티스 RBAC 기반으로 관리를 한다. 간단하게 도식으로 그리면 다음과 같은 구조를 가지고 있다.

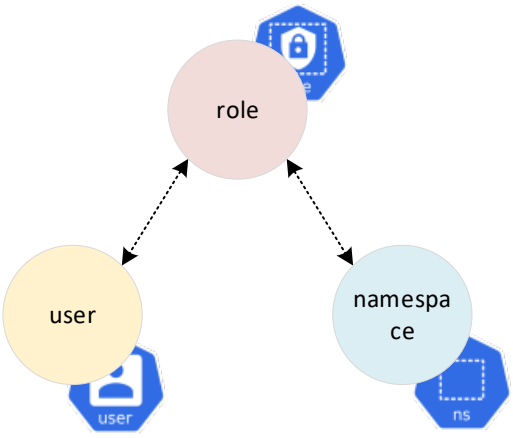


그림 27 네임스페이스 + 사용자 + 역할

사용자는 우리가 알고 있는 사용자라는 개념과 다르게 가지고 있다. 그 이유는 쿠버네티스는 기본적으로 API 기반으로 동작을 하기 때문에 각각 사용자는 "Cluster Certificate Authority CA"를 가지고 서버와 통신을 하게 된다. 그래서 해당 "CA"는 "subject"에 가지고 있는 "/CN=tang"과 같은 "common" 필드와 그리고 "RBAC"에 등록이 되어있는 역할과 매칭이 되어야한다.

하지만 사용자 혹은 서비스가 모든 API 에 대해서 CA 기반으로 호출하는 방식은 비효율적이기 때문에 각각 서비스별로 "Service Account"를 가지고 있다. 이 Service Account(이하 SA)는 서비스 어카운트 토큰을 시크릿(secret)를 통해서 가지고 있으며 이를 통해서 인증을 받은 "사용자(CN+CA=x509)"를 통해서 API 및 서비스 접근이 가능하다.

네임스페이스(NAMESPACE)는 우리가 익히 알고 있는 것처럼, 자원들이 생성 및 격리가 되는 공간이며**(Linux kernel 의 NAMESPACE 와 다른 개념이다.)** 네임스페이스 및 사용자는 하나의 역할(role)이 할당이 되어 있어야 사용자가 사용이 가능하다.

Role/Role-Binding

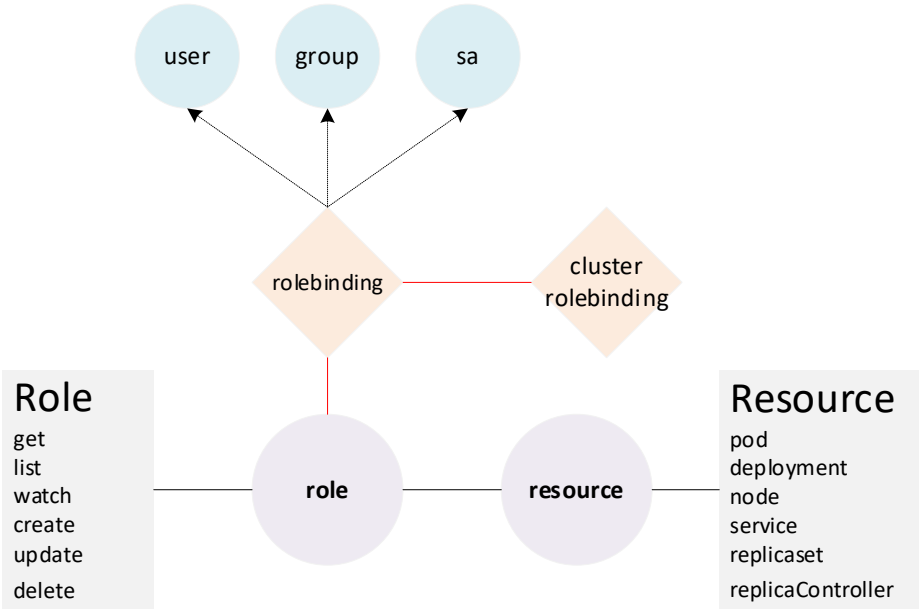


그림 28 역할 및 바인딩

RBAC 는 위에서 잠깐 설명하였지만 Role Base Access Control 의 약자이다. 자원은 기본적으로 rule, role 그리고 bind 으로 구성이 되어있다. 또한 객체로써 사용하는 단위는 사용자(user), 그룹(group) 그리고 서비스 어카운트(service account)가 있다.

RBAC 는 기본적으로 다음과 같은 요소로 구성이 되어있다.

- role: 역할은 권한의 영역을 결정한다. 영역은 rule 기반으로 한다.
- subject: 사용자 혹은 그룹으로 대상을 정한다.
- RoleBinding: 바인딩은 대상(subject)가 어떤 역할을 가져갈지 결정한다.

Role 구성 예제는 보통 다음과 같다.

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

- apiGroups: [“”] 보통 core API group 를 명시한다. 제한이 없는 경우 그냥 빈칸으로 비어둔다.
- resources[“pods”]: 어떤 자원 접근에 허용할지 결정한다.
- vers: 동사, 사용할 명령어이다. 예를 들어서 get, list, watch 같은 자원 조회 명령어이다.

쿠버네티스에서 RoleBinding 은 일종의 map 같은 역할을 한다. 여기서 말하는 맵핑(mapping)이라고 생각해도 되며, 어떠한 방식으로 바인딩을 통해서 사용자에게 연결하는지 확인한다.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: Test-rolebinding
  namespace: TestNamespace
subjects:
- kind: User
  name: test-user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: exampleRole
  apiGroup: rbac.authorization.k8s.io
```

subjects 밑으로 구성이 되어있는 내용은 다음과 같다.

- kind: User, 사용자 오브젝트 타입을 사용한다고 명시.
- name: test-user, test-user 에게 퍼미션 구성 및 설정한다.

roleRef 밑으로 구성이 되어있는 내용은 다음과 같다.

- kind: role, 정확히 어떤 사용자에게 역할을 붙여줄지 설정.
- name: exampleRole, 어떤 이름의 역할을 할당할지 명시.

Role vs Cluster Role

역할과 그리고 클러스터 역할은 기본적으로 퍼미션 혹은 권한을 구성하는 역할이다. 쿠버네티스는 이를 구성하기 위해서 RoleBinding 과 ClusterRoleBinding 를 사용한다. 기본적으로 둘은 비슷하게 보이지만 약간의 차이는 있다.

역할을 네임스페이스에서만 사용하는 경우 RoleBinding 으로 사용하지만, 네임스페이스 및 범위 상관없이 권한할당이 필요한 경우 ClusterRoleBinding 를 사용한다.

- 클러스터 노드에 접근이 필요한 경우.
- 모든 네임스페이스 자원 구성이 필요한 경우.
- 모든 엔드 포인트(End-Point)에 접근이 필요한 경우, 예를 들어서 /healthz 같은 주소.

클러스터 역할은 다음처럼 보통 구성한다.

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: Test-clusterrole
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

클러스터 바인딩 역할은 다음처럼 보통 구성한다.

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: example-clusterrolebinding
subjects:
- kind: User
  name: example-user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: example-clusterrole
  apiGroup: rbac.authorization.k8s.io
```

바인딩은 수정이 어렵기 때문에, 바인딩을 잘못 구성한 경우, 기존에 잘못 구성하였던 바인딩을 제거 후 다시 추가한다. 그렇지 않는 경우, 정책이 올바르게 동작이 되지 않을 수 있다.

사용자 계정 생성 및 RBAC

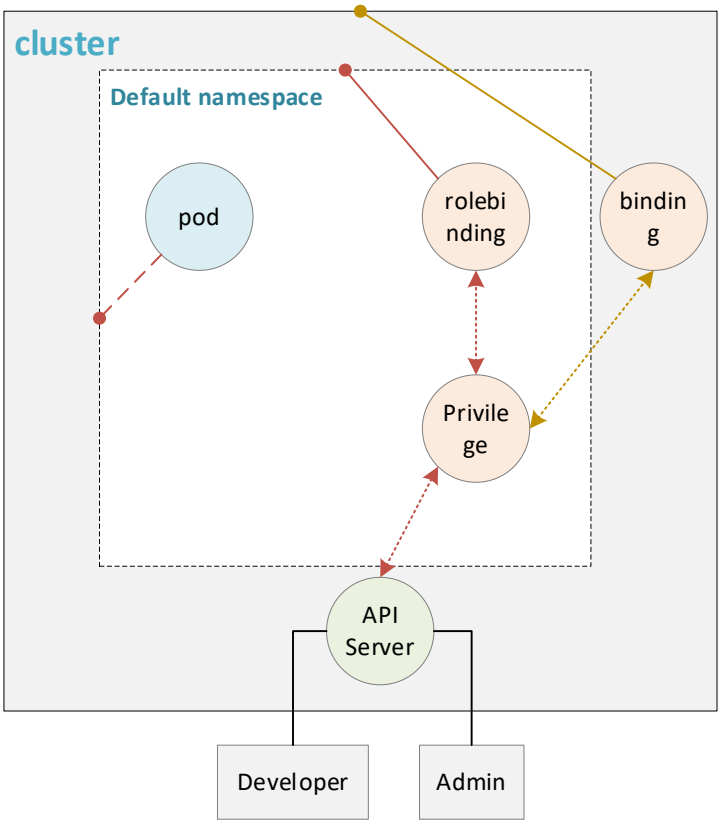


그림 29 RBAC USER, NAMESPACE

쿠버네티스에서 사용자, 우리가 알고 있는 사용자 개념하고 조금은 다르지만, 기본적으로 RBAC(Role-Based Access Control)기반으로 되어 있다. 데이터베이스처럼 **CRUD(Create, Read, Update, Delete)**기능 기반으로 권한을 제어한다. 이 기능은 다음과 같은 부분에 적용이 된다.

- Namespace
- Pods
- Deployments
- Persistent Volume
- ConfigMap

위의 자원에 적용이 가능한 RBAC 는 다음과 같다.

- create
- get
- delete
- list
- update

RBAC 기반으로 쿠버네티스에 적용하기 위해서는 다음과 같은 부분을 명시를 해야 한다.

- Role / ClusterRole

어떤 네임스페이스에서 어떤 권한 및 접근을 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.

1. view: 읽기 전용 계정이며, secrets 같은 자원에는 예외이다.
2. edit: 일반적인 자원에 접근이 가능하지만, 보안에 관련된 role, role binding 에는 접근이 불가능하다.
3. admin: 모든 자원에 접근이 가능하며, role, role-binding 기능을 네임스페이스에서 사용이 가능하다.
4. cluster-admin: 노드 자원 및 일반적인 admin 기능 및 접근제어가 전부 가능하다.

다음과 같이 상황을 만들어서 쿠버네티스 계정을 생성해보도록 한다. 현재 쿠버네티스 서비스는 구축이 되어 있으니, 사용자 별로 네임스페이스를 분리하도록 하겠다. 앞으로 네임스페이스를 **프로젝트**라는 단어로 대체하도록 하겠다.

프로젝트 이름	project-httpd-dev
권한	**satellite:** edit
프로젝트 이름	project-httpd-prod
권한	**satellite:** view
권한	**openshift:** edit

위와 같이 프로젝트 및 사용자를 구성하도록 한다. 참고로 쿠버네티스는 대시보드를 지원하지만, 레드햇 오픈시프트처럼 대시보드에서 사용자 생성 및 할당 기능은 지원하지 않는다. 대시보드 관련된 부분은 추후에 확인한다.

X.509 기반으로 새로운 사용자 생성

새로운 사용자를 생성이 가능한 형식은 두 가지가 있다. 첫 번째는 서비스 계정(service account) 두 번째는 일반적인 쿠버네티스 사용자(normal user)이다.

일반적인 사용자를 사용하기 위해서는 다음과 같은 조건이 필요하다.

- 기본 사용자 인증 조건
  - API Server 를 통해서 설정이 되어 있는 접근이 가능해야 됨
  - 사용자 이름, 비밀번호, uid, group 조건
- X.509 인증 기반
  - 사용자는 반드시 비-공개키를 가지고 인증서 인증을 받아야 됨
  - 쿠버네티스 CA 키 기반으로 인증 받은 사용자
- Bearer Tokens(JSON 웹 토큰)
  - OpenID 연결이 가능
  - OAuth 2.0
  - 웹훅(Webhooks)기반

여기에서는 일반적으로 많이 사용하는 X.509 기반으로 인증으로 진행하며, OpenSSL 기반으로 간단하게 구성하도록 한다. 앞에서 이야기하였던 사용자 계정에 대해서 다시 확인하도록 한다. 먼저 사용자를 생성한다. 생성할 사용자는 satellite, openshift 이다.

```
# useradd satellite
# useradd openshift
# mkdir satellite
# cd ~satellite/
# ls
# openssl genrsa -out satellite.key 2048
Generating RSA private key, 2048 bit long modulus
.........................................+++
.....................+++
e is 65537 (0x10001)

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

```
# openssl x509 -req -in satellite.csr -CA
/etc/kubernetes/pki/ca.crt -CAkey /etc/kubernetes/pki/ca.key -
CAcreateserial -out satellite.crt -days 500
Signature ok
subject=/CN=satellite/O=root
Getting CA Private Key
# ls
satellite.crt  satellite.csr  satellite.key
# mkdir .certs
# mv satellite.* .certs/
$ kubectl config set-credentials satellite --client-
certificate=/home/satellite/.certs/satellite.crt --client-
key=/home/satellite/.certs/satellite.key

User "satellite" set.
$ kubectl config set-context satellite-context --
cluster=kubernetes --user=satellite
Context "satellite-context" created.
```

명령어로 인증서 생성이 어려운 경우 아래의 도구를 통해서 설치가 가능하다. 이 책에서는 해당 도구에 대해서는 다루지 않는다.

```
$ curl -L
https://github.com/cloudflare/cfssl/releases/download/v1.5.0/cfssl_1.5.0
_linux_amd64 -o cfssl
chmod +x cfssl

$ curl -L
https://github.com/cloudflare/cfssl/releases/download/v1.5.0/cfssljson_1
.5.0_linux_amd64 -o cfssljson
chmod +x cfssljson
```

```
$ curl -L
https://github.com/cloudflare/cfssl/releases/download/v1.5.0/cfssl-
certinfo_1.5.0_linux_amd64 -o cfssl-certinfo
chmod +x cfssl-certinfo
```

연습문제

위의 명령어 및 yaml 가지고 아래 사용자를 생성한다.

프로젝트 이름	project-httpd-dev
권한	**satellite:** edit
프로젝트 이름	project-httpd-prod
권한	**satellite:** view
권한	**openshift:** edit

사용자 키 기반 클러스터 인증을 위한 YAML 파일을 작성한다. 파일명은 "satellite-cluster-account.yam"l 으로 아래 내용을 복사 혹은 작성한다. (버전 업데이트 후 Config 타입이 v1 에서 동작이 안됨) 아래 내용은 'kubectl config view'으로 확인이 가능하다.

```
---
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority-data: /etc/kubernetes/admin.config
    server: https://192.168.90.210:6443
    insecure-skip-tls-verify: true
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: satellite
  name: satellite-context
current-context: satellite-context
preferences: {}
```

```
users:
- name: satellite
  user:
    client-certificate: /home/satellite/.certs/satellite.crt
    client-key: /home/satellite/.certs/satellite.key
```

위의 YAML 으로 적용이 되지 않는 경우, kubectl 명령어로 다음처럼 진행한다.

```
$ kubectl config set-credentials satellite --client-
certificate=.certs/satelite.crt --client-key=.certs/satellite.key
```

네임스페이스가 존재하는 경우 아래처럼 테스트가 가능하지만, 현재 네임스페이스가 존재하지 않으므로 아직 실행이 되지 않는다.

```
$ kubectl config set-context satelite-context --cluster=kubernetes
--user=satellite --namespace=satellite
```

Role 및 Cluster Role 생성

마지막으로 role 및 clusterRole 를 생성한다. 내용이 길어서 나머지는 github 에 있는 내용을 참고하도록 한다. 이름은 "roles-normal-user.yaml"로 작성한다.

```
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: normal-user
  namespace: project-httpd-dev
rules:
  - apiGroups: ["*"]
    resources: ["pods"]
    verbs: ["list"]
---
```

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: test
subjects:
  - apiGroup: ""
    kind: User
    name: satellite
roleRef:
  apiGroup: ""
  kind: Role
  name: normal-user
```

문제없이 구성이 완료가 되면 'kubectl create -f roles-normal-user.yaml'으로 등록한다.

구성하기전에 네임스페이스 구성이 되어있는 RBAC 구성을 확인하기 위해서 “--all-namespace”옵션을 통해서 모든 네임스페이스에 대한 정보를 조회한다.

```
$ kubectl get roles --all-namespaces
NAMESPACE             NAME                                CREATED AT
kube-public           kubeadm:bootstrap-signer-clusterinfo
      2020-05-04T08:34:01Z
kube-public           system:controller:bootstrap-signer
      2020-05-04T08:34:00Z
kube-system           extension-apiserver-authentication-reader
      2020-05-04T08:34:00Z
kube-system           kube-proxy
      2020-05-04T08:34:02Z
$ kubectl get rolebindings --all-namespaces
NAME   ROLE               AGE
test   Role/normal-user   4m18s
```

사용자에게 Role/ClusterRole 연결하기

사용자는 최소 한 개의 role 과 그리고 namespace(프로젝트)가 필요하다. 아래있는 사용자에게 프로젝트 및 role 를 할당하여 RBAC 를 구현하도록 한다.

아래는 예제 파일이다. 참고만 하도록 한다.

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: jean
  namespace: my-project-dev
subjects:
- kind: User
  name: jean
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: edit
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: jean
  namespace: my-project-prod
subjects:
- kind: User
  name: jean
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: view
  apiGroup: rbac.authorization.k8s.io
```

구성이 된 사용자에게 위에서 이야기하였던 것처럼, “namespace”, “user” 그리고 “ClusterRole”를 선택한다. 아래는 satellite 및 openshift 라는 사용자를 바인딩하는 과정이다. 파일 이름은 rolebinding-satellite.yaml 으로 한다. 프로젝트가 없으니 미리 프로젝트를 kubectl create namespace project-httpd-dev 으로 하나 생성한다.

또한 우리는 아직 openshift 사용자를 생성하지 않기 때문에 openshift 사용자를 생성하지 않는다.

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: satellite
  namespace: project-httpd-dev
subjects:
- kind: User
  name: satellite
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: view
  apiGroup: rbac.authorization.k8s.io
```

문제없이 yaml 작성이 완료가 되면 kubectl create -f rolebinding-satellite.yaml 으로 등록한다. 다음 명령어로 현재 사용중인 컨텍스트로 변경 후 빠르게 apache 웹 서비스를 생성한다.

```
$ kubectl config set-context --current --namespace=project-httpd-dev
$ kubectl run httpd --image=httpd
```

컨텍스트 변경, 즉, 프로젝트가 변경 후, 해당 프로젝트에 내가 올바른 권한이 있는지 확인한다.

```
$ kubectl auth can-i list pods
yes
$ kubectl auth can-i create pods
yes
```

```
$ kubectl auth can-i delete pods
yes
$ kubectl auth can-i update pods
yes
```

위와 같이 메시지가 나오면 올바르게 구성 및 적용이 완료가 되었다.

연습문제

아래 프로젝트 및 사용자 그리고 RBAC 를 구성해라.

```
project-httpd-prod
satellite: view
openshift: edit
```

위에서 project-httpd-dev 와 satellite 사용자를 제외한 project-httpd-prod 와 openshift 사용자를 생성하지 않았다. 앞에서 학습한 내용을 가지고 project-httpd-prod 생성 후, 사용자 openshift 및 satellite 를 올바른 role 및 ClusterRole 기반으로 생성한다.

생성 순서는 다음과 같다.

1. 프로젝트 project-httpd-prod 를 생성
2. openshift 시스템 사용자 계정을 생성한다
3. openshift 시스템 계정에 X.509 기반으로 인증서 생성한다
4. 생성된 사용자는 .kube 라는 디렉터리를 생성 후, config 파일을 구성한다
5. 생성이 완료가 되면, config 파일에 인증서를 구성한다
6. Role, ClusterRole 에 view, edit 권한을 할당한다
7. 올바르게 할당이 되었는지, kubectl 명령어로 확인한다
8. 확인이 완료가 되면 httpd 기반으로 간단하게 컨테이너를 실행한다
9. 구성이 완료가 되면 최종적으로 kubectl auth can-i 명령어로 CRUD 를 각각 자원별로 확인한다

기본 명령어

쿠버네티스에서 사용하는 기본 명령어 학습. 쿠버네티스 운영시에 아래의 명령어는 최소한 알아야하는 내용중 하나이다. YAML 기반으로 POD 및 컨테이너 애플리케이션을 생성을 한다.

namespace

네임스페이스는 쿠버네티스에서 사용하는 자원을 격리하는 최소영역이다. 사용자가 생성하는 자원은 최소 한 개의 네임스페이스 안에서 생성이 된다.

네임스페이스 생성하는 방법은 두 가지 방법이 있다.

1. 명령어 기반으로 생성
2. YAML 방식으로 생성

명령어로 생성하는 방법은 아래와 같다.

```
# kubectl create namespace first-project
```

위와 같은 명령어로 간단하게 네임스페이스 자원 생성이 가능하다. 생성이 완료가 되면 다음 명령어로 생성된 네임스페이스 자원 목록을 확인이 가능하다.

```
# kubectl get namespaces
# kubectl config view --minify | grep namespace
```

특정 네임스페이스에 있는 자원을 확인 시, 보통 다음과 같은 옵션을 사용해서 네임스페이스에서 생성된 자원을 확인한다.

```
# kubectl get pod -n second-namespace
```

하지만, 매번 -n 옵션을 통해서 네임스페이스 지정이 번거롭기 때문에 보통 다음과 같은 명령어로 특정 네임스페이스를 고정적으로 명시한다. 기본 네임스페이스는 default 로 되어 있기 때문에, 별도로 설정하지 않으면 default 에 전부 생성이 된다.

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

쿠버네티스가 올바르게 설치 및 구성이 되었으면, 기본적으로 제공하는 네임스페이스가 다음처럼 있다. 해당 네임스페이스는 절대로 제거하면 안 된다.

확인하기 위해서 아래 명령어를 실행한다.

```
# kubectl get namespaces
```

기본 네임스페이스가 하는 역할은 아래와 같다.

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

네임스페이스를 명령어가 아닌 YAML 기반으로 생성은 다음과 같이 YAML 파일을 작성한다.

파일명은 namespace-firstnamespace.yaml 으로 생성한다.

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

올바르게 생성이 되면 다음과 같은 명령어로 생성을 시도한다.

```
$ kubectl create -f namespace-firstnamespace.yaml
```

연습문제

다음과 같은 이름으로 네임스페이스를 만든다. 만든 후, 네임스페이스를 set-context 로 기본 네임스페이스를 변경한다.

- hello-namespace
- second-namespace
- third-namespace

create/apply

쿠버네티스에서 자원을 사용 및 구성하기 위해서는 create 혹은 apply 를 사용해서 자원을 구성한다. create 나 apply 를 사용하기 위해서는 최소 1 개의 YAML 파일이 필요하다. 빠르게 서비스를 하나 생성을 해보도록 한다.

파일명은 nginx-demo-create.yaml 으로 생성한다.

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

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

올바르게 생성이 되면 다음과 같은 명령어로 생성을 시도한다.

```
$ kubectl create -f nginx-demo-create.yaml
```

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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-httpd
spec:
  selector:
    matchLabels:
      run: my-httpd
  replicas: 1
  template:
    metadata:
      labels:
        run: my-httpd
    spec:
      containers:
      - name: my-httpd
        image: httpd
        ports:
        - containerPort: 80
```

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

```
$ kubectl create -f apache-demo-create.yaml
```

다시, 기존에 생성하였던 파일에 다음과 같은 부분을 아래처럼 수정한다.

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

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

```
$ kubectl create -f apache-demo-create.yaml
```

올바르게 동작하지 않으면, 다시 아래 명령어로 실행을 한다.

```
$ kubectl apply -f apache-demo-create.yaml
```

올바르게 변경이 되었는지 아래 명령어로 확인해본다.

```
$ kubectl get pods
```

확인이 되었으면 다음 명령어로 확실히 변경된 내용이 적용이 되었는지 확인한다.

```
$ kubectl describe pods/<POD_ID>
```

연습문제

기존에 사용하였던 yaml 파일을 사용해서 다음처럼 서비스 구성을 한다.

1. deployment 에 이름은 public-vsftpd 라는 이름으로 구성.
2. vsftpd 이미지를 사용해서 yaml 파일을 작성.
3. 이미지 파일은 아래의 주소에서 받기가 가능.
   https://hub.docker.com/r/fauria/vsftpd/

4. 받은 이미지의 replica 개수는 1 개로 합니다.

get

get 명령어는 쿠버네티스 프로젝트 혹은 네임스페이스에 구성이 되어있는 자원 목록을 확인 시 사용하는 명령어이다. 이 명령어를 통해서 오브젝트 안에 구성이 되어있는 리소스 확인이 가능하다.

여기서 잠깐 쿠버네티스의 명령어 동작 구조는 다음처럼 가지고 있다.

KUBECTL <동사> <자원형식> <리소스>

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

```
$ kubectl get pod
```

위의 명령어는 현재 사용중인 네임스페이스에서 사용하는 경우, 위의 명령어로 처리하고 만약 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
```

실시간으로 출력 내용을 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 -A
```

연습문제

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

describe

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

사용 방법은 매우 간단하며 다음처럼 실행이 가능하다.

```
$ kubectl describe deployments.apps/my-httpd
```

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

```
$ kubectl describe -f nginx-demo-create.yaml
```

연습문제

- service 의 kubernetes 자원을 describe 명령어로 확인한다.

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 파일시스템 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 get pods
```

컨테이너 정보 정보에서 우리가 필요한 정보는 컨테이너의 이름이다. 실제로는 kubectl get pods 로 보는 정보는 컨테이너 정보가 아니라, 컨테이너 앞쪽에서 격리를 해주고 있는 POD 이다. 해당 POD 는 한 개 이상의 컨테이너를 가지고 있어도 일반적으로 POD 한 개만 보이는게 정상적이다.

```
$ 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

- 기존에 구성이 되어있는 my-httpd 에 index.html 파일을 복사한다.
- 해당 index.html 파일은 “Hello Apache”라는 문자열을 가지고 있어야 한다.

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

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

연습문제

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

expose or service(svc)

expose 명령어는 서비스는 현재 내부에서만 되고 있는 서비스를 외부로 노출하는 명령어이다. 이 명령어를 통해서 현재 구성된 서비스를 외부에서 접근이 가능하도록 한다. 현재 쿠버네티스 노드의 네트워크는 외부에서 접근을 하기 위해서 NodePort, ExternalIP 을 지원하고 있다. 현재 구성이 되어 있는 서비스에 NodePort 및 ExternalIP 를 구성해보도록 한다. 서비스를 생성하기 위해서 다음과 같은 방법으로 수행이 가능하다.

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

혹은 YAML 기반으로 자원을 생성 및 관리가 가능하다. 이름은 my-nginx-service.yaml 로 만든다.

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
```

둘 중 어떤 방식을 사용하여도 상관이 없다. 수행하기 전에 먼저 expose 자원이 생성이 되는 “service”자원을 확인해보도록 한다.

```
$ kubectl get service
```

위의 명령어를 수행하면 일반적으로 다음과 같은 명령어가 출력이 된다.

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

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

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

연습문제

- my-nginx-second 라는 이름으로 80/TCP 로 my-nginx-second 서비스를 ClusterIP 로 구성한다.

label

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

```
metadata:
  name: my-nginx
  labels:
    run: my-nginx
```

위와 같은 부분을 레이블이라고 하는데 이 레이블을 YAML 파일이 아닌 명령어를 통해서 추가 및 수정이 필요한 경우 label 이라는 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 "--overwrite"라는 옵션이 포함이 된다.

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

연습문제

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

run

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

보안 이유 때문에 busybox 이미지를 선호한다.

```
$ kubectl run curl --image=radial/busyboxplus:curl -i --tty
```

보안상 이유로 요즘은 점점 쿠버네티스 클러스터안에서 직접 확인하는 범위는 점점 줄어드는 추세이다.

다시 연결하기 위해서는 다음과 같이 명령어를 입력한다.

```
$ kubectl attach curl -c curl -i -t
```

연습문제

- centos 이미지로 console-centos 라는 이름으로 컨테이너를 실행

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.

```
$ kubectl set image deployment/my-httpd my-httpd=image:alpine
```

변경이 완료가 되면 describe 명령어로 확인이 가능하다.

```
$ kubectl describe pods/my-httpd
```

연습문제

- my-nginx 의 버전을 mainline-alpine 으로 변경한다

edit

ETCD 에 있는 내용을 바로 수정하기 위해서 사용하는 명령어이다. 이 명령어를 통해서 쿠버네티스에 등록된 설정 파일을 YAML 형태로 수정이 가능하다. 하지만, edit 값은 시스템의 EDITOR 나 혹은

KUBE_EDITOR 라는 값에 영향을 받으며, 일반적으로 vi 명령어가 사용하도록 되어있다. 일반적인 배포판은 nano 및 vi/vim 제공하고 있으며 기본 에디터인 vi 가 어려운 경우에는 nano 를 권장한다.

edit 사용 방법은 다음과 같다.

```
$ kubctl edit deployment my-httpd
```

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

```
$ KUBE_EDITOR=nano kubctl edit deployment my-httpd
```

혹은 위에서 말한 것처럼, KUBE_EDITOR 를 사용해서 변경이 가능하다.

```
$ KUBE_EDITOR=nano kubctl edit deployment my-httpd
```

위의 내용을 영구적으로 저장하려면 bash_profile 에 추가를 해주면 된다.

```
$ echo “KUBE_EDITOR=nano” >> ~/.bash_profile
```

연습문제

- deployment 에 구성된 httpd 및 nginx 의 replica 의 개수를 10 개로 변경 후 상태를 확인

delete

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

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

제거를 실행해보도록 한다. 먼저 POD 를 제거한다. 제거 시 특정 POD 만 제거하도록 selector 를 사용하도록 하겠다. "-l"옵션은 레이블 옵션이다.

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

service 에 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
```

diff

diff 는 YAML 에 구성 및 명시된 자원 사양과 맞게 되어 있는지 검증 및 확인하는 명령어이다. 실제 서비스에 배포하기전에 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.

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

연습문제

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

debug/logs

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

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

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

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

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

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

```
[Sun Sep 26 13:54:00.609837 2021] [mpm_event:notice] [pid 1:tid
140142545593472] AH00489: Apache/2.4.49 (Unix) configured -- resuming normal
operations
[Sun Sep 26 13:54:00.609935 2021] [core:notice] [pid 1:tid 140142545593472]
AH00094: Command line: 'httpd -D FOREGROUND'
```

연습문제

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

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
```

replace

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

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

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

추가가 된 다음에 아래와 같은 명령어를 실행하여 차이점을 확인한다.

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

그리고 아래 명령어도 실행을 한다.

```
$ kubectl replace -f apache-demo-create.yaml --force
```

어떠한 차이가 있는지 확인을 해본다.

연습문제

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

스토리지 구성

공통 설명

기본 쿠버네티스는 다음과 같은 스토리지를 제공한다.

- StorageClass
- Persistent Volume
- Persistent Volume Claim

초기에 쿠버네티스는 Volume 기반으로 PV 및 PVC 만 제공하였지만, 컨테이너 개수가 늘어나면서 PV 및 PVC 를 수동으로 확장해야 된다. 그래서 스토리지 클래스 기반으로 이러한 불편한 부분을 해결하였다.

기존에 사용하던 PV, PVC 는 정적인 스토리지 제공 방식이며, StorageClass 는 동적인 스토리지 제공 방식을 제공한다.

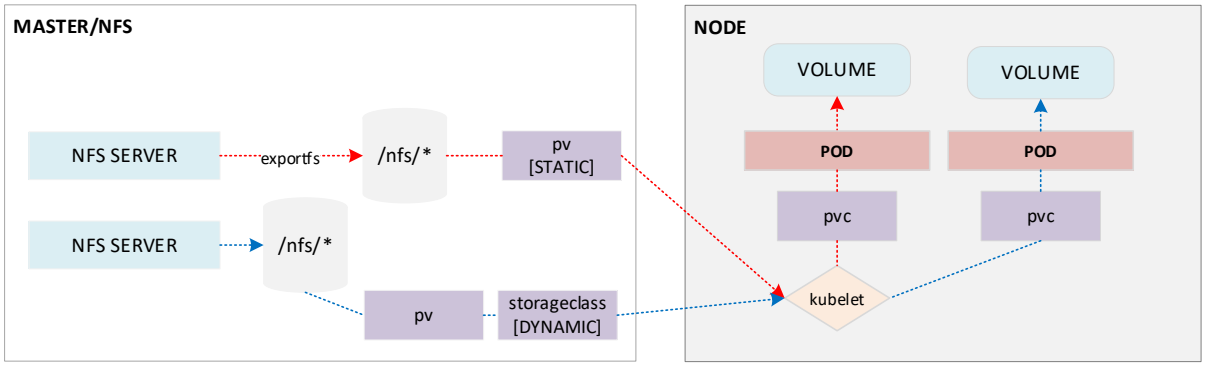


그림 30 PV, PVC, StorageClass

간단하게 NFS 서버를 구축 후, 위의 그림과 같이 구성하여 포드에 볼륨 구성 및 확인을 한다.

NFS 서버 구축

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

```
# yum install nfs-utils
```

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

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

디렉터리를 구성한다. 사용할 디렉토리 위치는 /nfs 에 구성한다.

```
# mkdir -p /nfs
```

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

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

exports 파일에 등록된 정보를 nfs server 에 갱신한다. 갱신 후 nfs.service 를 시작한다. 만약, SELinux 를 사용하고 있으면 반드시 아래처럼 컨텍스트 설정을 디렉터리에 한다.

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

만약, 컨텍스트 설정을 하지 않는 경우, SELinux 에서 차단을 한다. 임시적으로 SELinux 를 중지한다. 일시적으로 중지하는 명령어는 setenforce 0 으로 일시적으로 감사모드로 전환한다. SELinux 를 사용하고 싶은 경우, semanage 명령어를 통해서 SELinux 퍼미션을 다시 구성한다.

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

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

StorageClass

스토리지 클래스는 쿠버네티스의 volume, Persistent Volume 과 비슷하다. 기능적으로 다른 부분은 다음과 같다. 스토리지 클래스는 사용자가 직접 pv, pvc 를 구성할 필요가 없다. 자동적으로 pv, pvc 를 구성한다.

이름은 storageclass-configure.yaml 으로 저장한다.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-csi
provisioner: nfs.csi.k8s.io
parameters:
  server: master.example.com
  path: /nfs
reclaimPolicy: Delete
volumeBindingMode: Immediate
mountOptions:
  - hard
  - nfsvers=4.1
```

다음 명령어로 스토리지 클래스를 클러스터에 등록한다.

```
# kubectl get sc
NAME      PROVISIONER      RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
```

```
nfs-csi   nfs.csi.k8s.io   Delete          Immediate
false                  14m6s
```

테스트를 하기 위해서 deploy 파일을 구성한다. 이름은 stroageclass-deployment.yaml 으로 한다.

```
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
```

아래 명령어로 Pod 를 생성한다.

```
# kubectl create -f storageclass-deployment.yaml
```

생성이 완료가 되면, 다음과 같은 명령어로 pv, pvc 그리고 sc 를 확인한다.

```
[root@master ~]# kubectl get pv
NAME     CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS   REASON   AGE
nfs-pv   1Gi        RWX            Retain           Bound    first-
project/nfs-pvc
64m
[root@master ~]# kubectl get pvc
NAME      STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS
AGE
nfs-pvc   Bound    nfs-pv   1Gi        RWX
64m
[root@master ~]# kubectl get sc
NAME      PROVISIONER      RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
nfs-csi   nfs.csi.k8s.io   Delete          Immediate
false                  17m
```

스토리지 클래스가 올바르게 Pod 를 통해서 구성이 되었는지 describe 명령어를 통해서 확인한다.

```
# kubectl describe deploy nfs-csi-pod
    Mounts:
      /var/www/html/ from csi-nfs (rw)
  Volumes:
   csi-nfs:
    Type:      NFS (an NFS mount that lasts the lifetime of a pod)
    Server:    master.example.com
    Path:      /nfs
    ReadOnly:  false
```

올바르게 구성이 되었으면, 위와 같이 deployment 에서 연결 확인이 된다. Pod 의 개수를 한 개에서 10 개로 확장 후 직접 확인해본다.

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

완료가 되면 Pod 중 하나에 exec 를 통해서 올바르게 바인딩이 되었는지 확인한다.

```
# tocuh /nfs/readme.md
# echo "Hello World" > /nfs/readme.md
# kubectl exec -it nfs-csi-pod-595fc78789-x76wv -- df | grep
master.example.com
# kubectl exec -it nfs-csi-pod-595fc78789-x76wv -- cat
/var/www/html/readme.md
```

Persistent Volume(pv)

아래 내용으로 PV 파일을 작성 후, 쿠버네티스 클러스터에 스토리지 인터페이스를 등록한다. pv 는 다음과 같은 모드를 지원한다.

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

Persistent Volume 를 생성한다. 파일 이름은 pv.yaml 으로 저장한다.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
  labels:
    type: nfs
spec:
```

```
  storageClassName: "sc-nfs"
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: master.example.com
    path: "/nfs"
```

아래 명령어로 PV 를 클러스터에서 생성한다.

```
# kubectl create -f pv.yaml
```

등록이 문제없이 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
```

Persistent Volume Claim(pvc)

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

파일 이름은 pvc.yaml 으로 저장한다.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  storageClassName: ""
  accessModes:
```

```
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

아래 명령어로 PVC 를 클러스터에서 생성한다.

```
# kubectl create -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
```

확인을 위해서 Pod 및 컨테이너를 생성하여 올바르게 연결이 되는지 확인한다. 파일이름은 pvc-pod.yaml 라는 이름으로 작성한다.

```
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
```

아래 명령어로 PVC 를 클러스터에서 생성한다.

```
# kubectl create -f pvc-pod.yaml
```

생성이 완료가 되면, describe 명령어를 통해서 올바르게 Pod 에 연결이 되었는지 확인한다.

```
# kubectl describe pod pvc-pod
    Mounts:
      /app/data from htdocs (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-
access-5245r (ro)
```

역시 pvc 에도 올바르게 연결이 되었는지 확인한다.

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

고급 명령어

metrics

설치

사용 및 확인

endpoints

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

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

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

Service Account

연습문제

scale/rollout/rollback/history

시스템에서 사용하는 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.

```
                                                    .-- C1
                                                   /
Deployment ---- ReplicaSet ---- POD ---- Containers ---- C2
                                                   \
                                                    '-- C3
```

실험을 하기 위해서 다음과 같은 명령어를 통해서 서비스를 구성한다. 아래 파일은 nginx-deployment.yaml 로 작성한다.

```
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
```

생성이 완료가 되면 create 명령어로 서비스를 구성한다.

```
$ kubectl create -f nginx-deployment.yaml
```

올바르게 생성이 되는지 kubectl get pods 명령어로 계속 확인을 하며, 또한 생성을 rollout 를 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
```

혹은 위의 방식이 어려운 경우 앞에서 학습한 kubectl edit 명령어로 수정을 하여도 된다. 수정이 완료가 되면 롤아웃을 실행한다. 먼저 실행하기전에 앞에 명령어 보면 특이한 옵션이 하나 --record 라는 옵션이 보이는데, 기록을 계속 추적 및 남기기 위해서 저 옵션을 사용한다.

```
$ kubectl rollout status deployment/nginx-deployment
```

rollout 이 진행이 되면 rs 나 혹은 deploy 에서 확인하면 컨테이너가 프로비저닝 되는 상태가 출력이 된다. 새로 구성이 된 POD 를 확인하기 위해서는 kubectl get pods 명령어로 확인이 가능하다.

자, 이제는 기본적인 rollout 를 사용하였다. 이젠 반대로 다시 rollback 실행을 한다. rollback 우리가 익히 알고 있는 내용처럼, 수행 이전상태로 상태를 변경한다. 단, rollback 가상머신의 롤백 혹은

snapshot 처럼 이루어지는 방식이 아니며 모든 자원들이 제거가 된 다음에 재-생성이 되는 구조이다. 이 부분은 꼭 기억해두자.

이젠 기존에 구성했던 내용에 일부로 오류를 발생한다. 다음과 같은 명령어로 오류를 발생시킨다.

```
$ 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 rollout 이 여전히 진행이 되고 있지 않으며, get rs 나 get pods 명령어로 확인하면 여전히 컨테이너 서비스는 여전히 갱신이 안되고 있지 않는 게 확인이 된다.

자 그러면, rollout 된 수정사항을 확인 후 다시 롤백을 시도를 한다. 기억하겠지만, 앞에서 우리가 kubectl 명령어를 실행할 때 --record 라는 옵션을 사용하였다.

아래 명령어로 rollout 기록을 확인한다.

```
$ 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
```

rollout 를 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 rollout 를 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
```

연습문제

- my-httpd 에 이미지 버전을 2.4.28 로 변경한다.
- 변경 후 롤 아웃을 시도한다.
- 다시 버전을 2.4.25 로 변경한다.
- 완료가 되면 최종 버전으로 전부 롤 백한다.

레이블 셀렉터 업데이트

레이블 업데이트를 하기 위해서 두 가지 방법으로 접근할 수 있다.

1. 하위 명령어 'edit'사용하기
2. 'replace'를 사용해서 내용 변경하기

레이블 셀렉터, 즉 레이블을 변경하기 위해서는 기존에 사용하던 YAML 파일을 수정하는 방법이 제일 안전하다. 변경하는 방법은 간단하게 기존에 사용하던 파일에서 label 부분을 변경한다. 방법은 다음과 같다. 기존에 사용하였던 "nginx-deployment.yaml"를 사용해서 다음처럼 수정한다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx ## 이 부분은 nginx-replace 로 변경한다.
spec:
  replicas: 3
  selector:
    matchLabels:
```

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

기존 레이블에 "app: nginx"의 부분을 "nginx-replace"로 변경한다. 변경이 문제없이 되면 레이블 값은 변경이 된다. 그리고 다시 'kubectl get pods'명령어를 통해서 POD 가 잘 동작하는지 확인한다. 큰 문제가 없으면 이번에는 다음과 같은 부분을 변경한다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx-replace
spec:
  replicas: 3 ## 이 부분을 5 로 변경한다.
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
```

```
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

레이블만 변경하였을 때는 POD 에서는 크게 다른 반응이 없지만, "replicas"가 변경이 되면서, 바로 리소스가 업데이트 되는 게 확인이 된다.

연습문제

'replace'명령어를 통해서 아래와 같은 작업을 수행 후 어떤 방식으로 작업이 수행 및 반영이 되었는지 확인한다.

- 위에서 작업하였던 내용들을 'describe'을 통해서 내용을 확인한다.
- 기존에 있는 레이블에 "mode: service"라고 추가한다.
- 이미지 부분을 "1.14.2"를 "latest"로 변경한다.

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.

```
# kubectl get pod -n kube-system -l k8s-app=metrics-server
# kubectl describe pod -n kube-system -l k8s-app=metrics-server
```

앞에서 이야기하였던 scale 하고 기능적으로 차이는 없지만, HPA 가 특정 조건에 따라서 자동적으로 POD 를 확장하다는 차이가 있다. HPA 를 사용하기 위해서는 'autoscale'을 사용하겠다고 POD 자원에 선언해야 한다.

```
$ kubectl autoscale deployment.v1.apps/nginx-deployment --min=1 --
max=15 --cpu-percent=50
```

옵션을 간단하게 설명하면 "--min", "--max"는 HPA 에서 실행 시, POD 생성 개수이다. "--min=1"이면 시작 시 POD 는 1 개로 시작한다. "CPU"가 사용율이 "50%"를 넘으면 POD 를 수평적으로 확장을 시작한다.

그래서 HPA 에 이름에 "H"가 "Horizontal"이다. 또한, 오토 스케일을 사용하기 위해서는 자원에 대한 모니터링이 되어야 하며, 그러기 위해서는 "메트릭 서버(matrix server)"구성이 필요하다. 설치 크게 문제가 없으면 메트릭 서버는 동작 중이다. 간단하게 테스트하기 위해서 아래처럼 컨테이너파일(Containerfile)를 생성한다. 파일 이름은 "Containerfile-apache-php"로 한다.

```
FROM php:5-apache
COPY index.php /var/www/html/index.php
RUN chmod a+rx index.php
```

아래는 php 로 작성된 과부화 프로그램이다. "index.php"에 아래와 같이 작성한다.

```
<?php
  $x = 0.0001;
  for ($i = 0; $i <= 1000000; $i++) {
    $x += sqrt($x);
  }
  echo "OK!";
?>
```

단, 직접 빌드한 이미지로 컨테이너를 생성하려는 경우, 쿠버네티스에서 이미지 접근이 가능하도록 "/etc/containers/registry.conf"에 변경을 해주어야 한다.

```
unqualified-search-registries = ["registry.fedoraproject.org",
"registry.access.redhat.com", "docker.io"]
```

작성이 완료가 되면 이미지 빌드를 시작한다. 이미지 빌드는 "buildah" OCI 표준 도구를 사용해서 빌드를 시작한다.

```
# buildah bud -f Container-apache-php
```

HPA 를 사용 및 구성하기 위한 Deployment 를 작성한다. 파일명은 php-apache-autoscale.yaml 로 작성한다.

```
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
```

위의 내용처럼 service, deployment 두 가지 영역이 구성이 되어있다. 완료가 되면 apply 로 다시 적용한다.

```
$ kubectl apply -f php-apache-autoscale.yaml
```

HPA 가 자원을 모니터링 할 수 있도록 아래 명령어를 실행한다.

```
# kubectl autoscale deploy/php-apache --min=2 --max=15 --cpu-
percent=80
```

메트릭 서버가 올바르게 구성이 되어 있다고 하면, 잠시 시간이 지나면 HPA 에서 CPU 및 Memory 사용율을 수집하기 시작한다.

```
$ kubectl get hpa
```

다른 터미널을 하나 더 실행하여 서버에 다시 접속 후 다음처럼 다시 확인한다.

```
$ kubectl get deployment php-apache
```

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.

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
```

```
    name: php-apache-autoscale
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
status:
  observedGeneration: 1
```

HPA 구성은 다음과 같은 방식으로도 가능하다. 만약 autoscale 명령어가 아닌 YAML 로 구성을 하고 싶은 경우, 아래처럼 YAML 파일 작성 후 배포가 가능하다.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

연습문제

- 메트릭 서버가 구성이 되어 있는지 확인한다.
- 구성 후 php-apache 의 개수를 최소 5 개 최대 20 개로 변경한다.

- cpu 사용율을 30%으로 변경한다.

drain/taint/cordon/uncordon

drain 은 말 그대로 모든 것은 배출하는 명령어이다. 즉, 현재 워크 노드에서 사용중인 모든 컨테이너를 유지보수나 혹은 장비 교체 같은 이유로 전부 제외시키는 명령어. 이 명령어는 수행이 되면 드레인이 된 컨테이너는 다른 노드에서 다시 구성이 되어서 동작을 하게 된다. 일반적으로 "관리모드"로 변경하기 위해서 다음과 같은 과정을 수행한다.

```
get nodes → taint node → drain node → uncordon node → taint node NoSchedule- → delete node
```

드레인을 하기 위해서 먼저 노드 이름을 확인한다. 아래 명령어로 노드 이름을 확인한다.

```
$ kubectl get nodes
```

먼저, 새로운 컨테이너가 생성되지 않도록 "taint"를 선언한다. taint 는 직역하자면 "오염"이다. 오염이라고 노드에 선언하면, 해당 노드에는 더 이상 컨테이너가 생성되지 않는다. 아래처럼 taint 를 선언한다.

선언된 노드는 스케줄러 상태가 "DisabledScheduled"라고 표시가 된다. 두 번째 명령어는 "NoExecute"는 "더 이상 POD 를 실행하지 않는다" 라고 선언한다.

```
$ kubectl taint node node1.example.com key1=value1:NoSchedule
$ kubectl taint nodes node1 key1=value1:NoExecute
```

실행 후 'describe'명령어로 확인하면 다음처럼 화면에 출력이 된다.

```
Taints:             key1=value1:NoExecute
                    key1=value1:NoSchedule
Unschedulable:      false
```

자, 이제는 실행중인 컨테이너를 다른 노드로 이동을 해주어야 한다. 이미 "value:NoExecute"의 영향을 받아서 POD 는 다른 노드로 이동이 되었다. 그래도 확실히 제외하기 위해서 한번 drain 를 실행한다.

```
$ kubectl drain node1.example.com
```

드레인 실행 후, 노드 상태를 확인한다.

```
Taints:             key1=value1:NoExecute
                    key1=value1:NoSchedule

node.kubernetes.io/unschedulable:NoSchedule
Unschedulable:      true
```

드레인이 실행이 되면 위의 출력처럼 "Unscheduleable"이 "true"변경이 되어 있다. 변경이 완료가 되면 "cordon"를 실행한다. 단어를 직역하면 "비상선" 혹은 "폴리스 라인"과 같다. 노드는 이미 "drain"를 통해서 스케줄링에서 제외가 되어 있다. 하지만, 한 번 더 안전하게 노드에 더 이상 자원할당이 되지 않도록 선언한다.

드레인은 모든 POD 가 되지는 않는다. 예를 들어서 네트워크 및 스토리지 관련된 POD 가 있는 경우 바로 되지 않는다.

드레인 실행 방법은 보통 사용중인 노드의 이름을 명시를 하면 된다. 앞서 'kubectl get nodes'명령어로 노드 이름을 확인한다. 드레인은 모든 POD 즉, 시스템 POD 인 네트워크나 스토리지 관련된 POD 는 이동하지는 못한다. 이미 드레인(drain)를 수행한 경우에는 다음과 같이 메시지가 출력이 된다.

```
$ kubectl cordon node1.example.com
node/node1.example.com already 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 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.

```
$ 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.

- 기존 node2 에 있는 POD 를 전부 node1 으로 이전한다.
- 이전 작업들은 앞에서 이야기하였던 단계로 수행한다.
- 작업이 완료가 되었으면 다시 노드를 서비스 할 수 있도록 스케줄러를 복원 및 복구한다.

Service HealthCheck

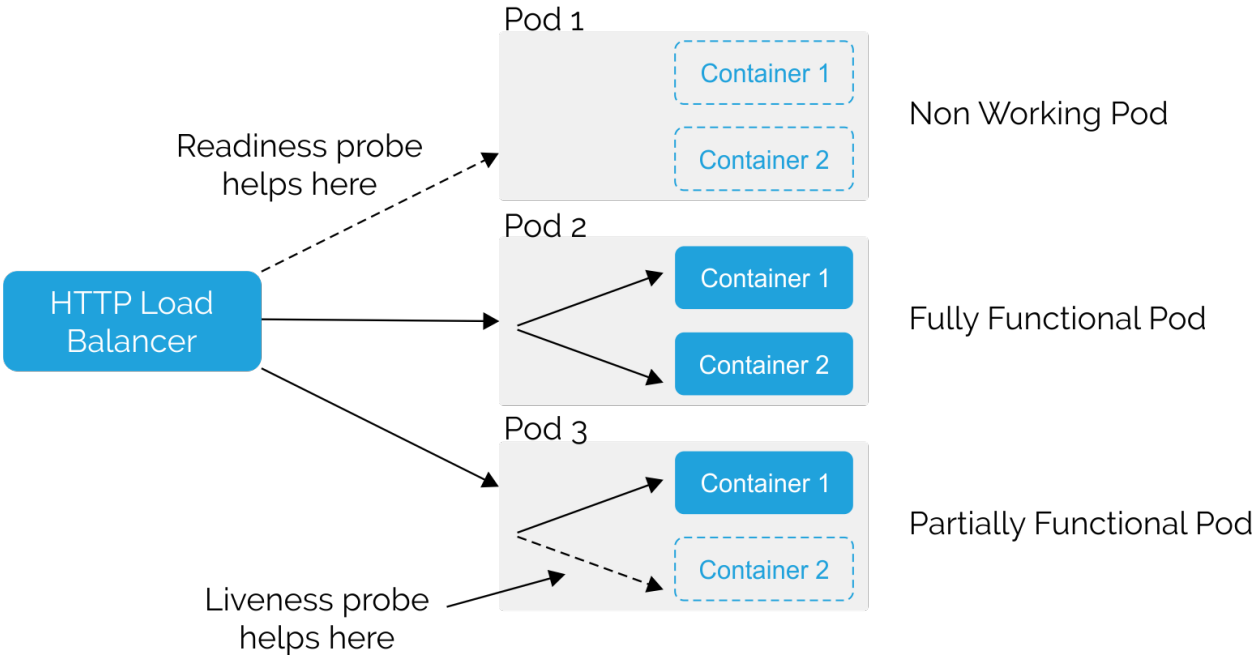
HealthCheck 소개

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.

바로 "liveness", "readiness"라는 기능이다. 아래 그림은 컨테이너 서비스를 확인하기 위해서 애플리케이션 주소 혹은 POD 의 상태를 확인하여 올바르게 서비스가 되는지 확인 가능한 방법이 있다.



위의 기능을 사용하기 위해서 POD 나 컨테이너에서 조회하는 URL 위치가 있다.

- /ready: "readinessProbe"는 아래 "livenessProbe"와 동작 방식은 비슷하다.

- /healthy: "livenessProve"가 바라보는 위치이다. 이 위치는 파일이 될 수도 있고 주소가 될 수도 있다.
- /status

liveness, readiness 는(은) 총 3 가지의 "probes" 형식은 다음과 같이 지원한다.

- httpGet
- tcpSocket
- exec

readiness, liveness 는 같이 사용이 가능하며 동작 방식에 대한 우선 순위가 없다. 그래서 주의할 부분은 readiness 의 **"probes"**는 컨테이너 시작부터 종료까지 모든 라이프 사이클 따라간다. 하지만, liveness 의 **"probes"**는 "readiness probe"가 성공할 때까지 가다리지 않는다. liveness 는 애플리케이션이 올바르게 동작하면 명시된 주소, 예를 들어서 "/healthy"같은 위치에 접근해서 확인한다. 그래서 만약, "liveness"가 시작 후 동작하기 위해서는 다음과 같이 설정한다.

```
readinessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

자세한 내용은 아래에서 다루도록 한다.

Liveness

"liveness"는 컨테이너에서 사용하는 애플리케이션 서비스의 상태를 확인한다. 컨테이너에서 사용하는 애플리케이션에서 상태를 확인할 수 있도록 하위 주소를 명시한다. liveness 는 보통 "httpGet"라는 도구를 통해서 컨테이너에 명시된 하위 주소를 통해서 접근하여, 올바르게 동작하는지 확인한다. "liveness"가 동작하게 되면 "goproxy"가 내부적으로 실행이 되면서 컨테이너의 8080 포트로 연결이 된다.

추가적으로 liveness 는 "gRPC liveness probe"도 지원한다. 아직은 베타이지만 사용은 가능하다. 사용 방법은 아래와 같다. 이 예제는 쿠버네티스 사이트에서 제공해주는 예제이다. 이름은 "liveness-grpc.yaml"로 저장한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: etcd-with-grpc
spec:
  containers:
  - name: etcd
    image: registry.k8s.io/etcd:3.5.1-0
    command: [ "/usr/local/bin/etcd", "--data-dir",
"/var/lib/etcd", "--listen-client-urls",
"http://0.0.0.0:2379", "--advertise-client-urls",
"http://127.0.0.1:2379", "--log-level", "debug"]
    ports:
    - containerPort: 2379
    livenessProbe:
      grpc:
        port: 2379
      initialDelaySeconds: 10
```

"grpc"을(를) 통해서 실행된 "etcd"서버에 "2379"포트를 통해서 확인한다. 일반적인 "httpGet"를 사용해서 서비스를 확인한다.

아래는 쿠버네티스에서 미리 배포하는 liveness 컨테이너 이미지이다. 해당 이미지를 가지고 어떤식으로 liveness 가 동작하는지 확인한다. 이름은 "liveness-http.yaml"로 저장한다.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
```

```
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

위의 컨테이너가 사용하는 애플리케이션의 "/healthz"는 다음처럼 구성이 되어 있다.

```
http.HandleFunc("/healthz", func(w http.ResponseWriter, r
*http.Request) {
    duration := time.Now().Sub(started)
    if duration.Seconds() > 10 {
        w.WriteHeader(500)
        w.Write([]byte(fmt.Sprintf("error: %v",
duration.Seconds())))
    } else {
        w.WriteHeader(200)
        w.Write([]byte("ok"))
    }
```

tcp 기반으로 사용을 원하는 경우 아래처럼 구성을 한다. 이름은 "liveness-tcp.yaml"로 저장한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-tcp-socket-healthcheck
```

```
spec:
  containers:
  - name: redis
    image: redis
    # defines the health checking
    livenessProbe:
      # a TCP socket probe
      tcpSocket:
        port: 6379
      # length of time to wait for a pod to initialize
      # after pod startup, before applying health checking
      initialDelaySeconds: 30
      timeoutSeconds: 1
    ports:
    - containerPort: 6379
```

위의 내용은 "tcpSocket"의 "6379"포트를 확인한다. 아래 "initialDelaySeconds", "timeoutSeconds"를 통해서 POD 가 시작하는 중 발생하는 대기 시간에 대해서 설정한다.

```
livenessProbe:
  failureThreshold: 3
  httpGet:
    path: /healthz
    port: 8080
```

Readiness

Liveness 와 비슷한 기능을 가지고 있으나, 이 기능은 POD 기반으로 동작한다. POD 는 동작을 하면 "status"정보를 가지게 되는데, 이 정보를 가지고 상태를 판단하거나 혹은 POD 에서 애플리케이션 상태를 질의하는 방법이 있다. "readinessProbe"도 "tcp", "http"두 가지 방식을 제공한다. 아래는 "readiness probe"중에서 "httpGet"예제이다. 이름은 "readiness-http.yaml"으로 저장한다.

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: kubernetes-readiness-probe
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        readinessProbe:
          initialDelaySeconds: 1
          periodSeconds: 2
          timeoutSeconds: 1
          successThreshold: 1
          failureThreshold: 1
          httpGet:
            host:
            scheme: HTTP
            path: /
            httpHeaders:
            - name: Host
              value: exampleapp.com
            port: 80
```

위의 서비스를 구성한 다음에, 'kubectl describe'명령어를 통해서 POD 가 올라오면서 어떠한 이벤트가 발생하는지 확인한다. 크게 문제가 없으면 "kubernetes-readiness-probe-svc.yaml"파일을 생성한다.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: kubernetes-readiness-probe
  name: kubernetes-readiness-probe-svc
  namespace: defaul
spec:
  ports:
  - name: kubernetes-readiness-probe-port
    port: 80
  selector:
    app: nginx
  sessionAffinity: None
  type: NodePort
```

크게 문제가 없으면 컨테이너는 실행이 된다. 실행이 될 때 올바르게 'kubectl describe'를 통해서 "Readiness"가 올바르게 동작하는지 확인한다.

```
Readiness: http-get http://:80/ delay=1s timeout=1s period=2s
#success=1 #failure=1
```

정상적으로 서비스가 올라오면 "End Point(엔드 포인트)"가 잘 구성이 되었는지 확인한다.

```
$ kubectl get endpoints
NAME         ENDPOINTS
AGE
kubernetes   192.168.90.81:6443
2d22h
nginx        10.244.11.104:80,10.244.221.19:80,10.244.221.26:80 + 8 more...
48m
php-apache   10.244.221.60:80
```

마지막으로 readiness, liveness 를 동시에 사용하기 위해서는 다음처럼 구성한다. 파일 이름은 "readiness-liveness-readiness.yaml"으로 저장한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: registry.k8s.io/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

위의 코드는 liveness, readiness 두 개의 probe 를 실행한다. 먼저 POD 에서 컨테이너 상태를 확인해야 되기 때문에 "readiness"가 먼저 시작하고, 그 다음에 "liveness"가 실행이 된다.

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.

구분자(label)

레이블은 생성되는 자원에 구분하기 위한 구분자를 구성한다. 구분자(label)은 특별한 기능이 아니라 메타데이터(metadata)같은 기능이다.

이를 통해서 POD, Deployment 에 생성된 자원에 선택자(selector)를 구성할 수 있다. 기본적으로 레이블은 키 기반으로 쌍으로 구성이 되어 있으며, 키에는 항상 키 값이 같이 따라온다. 이를 보통 키페어(keypair)라고 부른다. 키페어는 YAML 이나 혹은 명령어로 선언이 가능한데 어떤 위치에서 사용하느냐 따라서 조금씩 문법이 다르다. 일반적으로 YAML 에서는 다음처럼 선언한다

```
keyname: <key value>
```

명령어에서 처리시 일반적으로 다음처럼 명령어를 사용한다

```
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"
```

직접적으로 YAML 에 선언이 가능하다. 선언하는 방법은 다음과 같이 한다.

```
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
```

위의 내용에서 중요한 부분은 "metadata"부분이다. metadata 는 labels 라는 속성을 가지고 있으며, 이를 통해서 선택자 즉, selector 를 생성한다.

연습문제

선택자(selector)

선택자는 셀렉터(selector)라는 이름으로 부르기도 하며, 이를 통해서 생성된 자원을 선택할 수 있다. 이 선택자가 필요한 이유는 쿠버네티스는 클러스터를 통해서 전반적으로 모든 자원을 구성하기 때문에 최소 한 개 이상의 선택자를 구성하는 것을 권고한다.

선택자는 실제로는 위에서 생성한 구분자(label)로 구별이 되기 때문에 구분자 정보가 들어간 자원이 구성이 되면, 다음처럼 쉽게 선택자를 통해서 선택이 가능하다.

YAML 기반으로 작성시에 선택자는 보통 다음처럼 구성이 되어 있다.

```
"metadata": {
  "labels": {
     "key1" : "value1",
     "key2" : "value2"
  }
}
```

구성된 구분자를 통해서 자원을 선택하는 경우, 아래처럼 YAML 를 작성한다. 위에서 사용한 label 를 기반으로 다시 활용해서 확인을 해보도록 한다. 파일 이름은 label-demo.yaml 으로 저장한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
     environment: production
     app: nginx
spec:
  containers:
  - name: nginx
     image: nginx:1.14.2
     ports:
     - containerPort: 80
```

간단하게 nginx 기반으로 포드 및 컨테이너를 생성한다. 작성한 YAML 파일을 'kubectl create -f'명령어로 적용한다. 위의 YAML 에서 생성한 구분자의 정보는 다음과 같다.

```
labels:
  environment: production
  app: nginx
```

"name: label-demo"는 같은 메타정보이지만, 이 메타정보는 포드 생성시 사용하는 이름이다. 구분자는 "labels"밑으로 구성된 부분이 구분자이다. 이 구분자는 다음과 같은 조건식을 사용할 수 있다.

=: 해당 키 이름이 값이 같은 경우 적용

=!: 해당 키 이름이 값이 다른 경우 적용

위에서 작성한 nginx 애플리케이션을 서비스(service)를 통해서 외부에 서비스할 수 있도록 서비스 자원을 구성한다. 파일 이름은 label-nginx-service.yaml 으로 저장한다.

```
apiVersion: v1
kind: Service
metadata:
  name: label-nginx-service
spec:
selector:
  app: nginx
    environment: production
  ports:
    - protocol: TCP
     port: 8080
     targetPort: 80
```

연습문제

다음과 같이 레이블을 추가한다.

- virtualization_type: rhv
- enviroment: test

올바르게 추가가 되었는지 describe 명령어로 확인한다.

annotations

annotations 기능은 Label 과 다르게 좀 더 넓은 범위에서 사용한다. annotation 은 일종의 주석 같은 기능이며, 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.

```
apiVersion: v1
kind: Pod
metadata:
  name: annotations-demo
  annotations:
     imageregistry: "https://hub.docker.com/"
spec:
  containers:
  - name: nginx
     image: nginx:1.14.2
     ports:
     - containerPort: 80
```

위의 annotation 은 metadata 영역안에 구성이 되어 있으며, 여기에는 POD 에서 사용하는 이름인 "annotations-demo”, annotations 에는 imageregistry 에 "https://hub.docker.com”이라고 명시가 되어있다. annotations 에 선언이 되어 있는 내용들은 annotations-demo POD 와 관련되어 있는 자원들은 접근해서 사용이 가능하다.

사용방법은 다음과 같다. 아래 예제는 쿠버네티스에서 컨테이너 이미지를 생성시 사용하는 컨테이너 이미지를 내려 받는 서버의 주소 정보를 변경한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: annotations-demo
  annotations:
     imageregistry: "https://quay.io/tangt64/"
spec:
  containers:
  - name: nginx
     image: nginx:1.14.2
     ports:
     - containerPort: 80
```

연습문제

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-1.example.com    Ready       master    10d    v1.18.3
node-1.example.com     Ready <none>    10d    v1.18.3
node-2.example.com     Ready <none>    10d    v1.18.3
```

```
node-3.example.com     Ready <none>   10d   v1.18.3

$ kubectl label node node-2.example.com "ssd=true"
node/node-2.example.com labeled

$ kubectl get nodes --selector "ssd=true"
NAME                   STATUS   ROLES   AGE   VERSION
node-2.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
node-2.example.com   Ready  <none>   10d   v1.18.3
```

연습문제

모든 노드에 다음처럼 레이블을 적용한다

- lightamd=true

Jobs

Jobs 시스템의 crond 처럼, 반복적인 작업을 처리시에 사용하는 기능이다.

예를 들어서 일정시간에 특정 컨테이너가 실행되면서 알림 및 혹은 컨테이너 상태를 확인하는 용도로 사용이 가능하다.

시스템에 등록이 되어 있는 jobs 를 확인하려면 다음 명령어로 확인이 가능하다. 아래는 job 으로 등록된 서비스는 아래와 같이 동작한다.

```
$ kubectl get jobs --all-namespaces
```

```
NAMESPACE   NAME                                COMPLETIONS   DURATION   AGE
rook-ceph   rook-ceph-osd-prepare-node-1.example.com   1/1           4s
      34m
rook-ceph   rook-ceph-osd-prepare-node-2.example.com   1/1           4s
      34m
rook-ceph   rook-ceph-osd-prepare-node-3.example.com   1/1           3s
      34m
```

자세한 작업 내용을 확인하려면, describe 명령어를 통해서 상세한 jobs 내용 확인이 가능하다.

```
$ kubectl describe jobs/rook-ceph-osd-prepare-node-1.example.com -n rook-ceph
Name:             rook-ceph-osd-prepare-node-1.example.com
Namespace:        rook-ceph
Selector:         controller-uid=fa9f018c-ce40-4a34-bd4c-3585b3577b81
Labels:           app=rook-ceph-osd-prepare
                  ceph-version=14.2.9-0
                  rook-version=v1.3.4
                  rook_cluster=rook-ceph
Annotations:      <none>
Parallelism:      1
Completions:      1
Start Time:       Mon, 15 Jun 2020 11:56:38 +0900
Completed At:     Mon, 15 Jun 2020 11:56:42 +0900
Duration:         4s
Pods Statuses:    0 Running / 1 Succeeded / 0 Failed
```

임시로 Jobs 를 생성하여 컨테이너 생성을 시도한다. 이 컨테이너는 생성이 성공이 되지 않으며, 생성 중간에 인자 값 문제로 생성 및 실행이 실패한다. 파일 이름은 one-shot-job-failed.yaml 로 저장한다.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: oneshot
  labels:
      chapter: jobs
spec:
  template:
      metadata:
      labels:
      chapter: jobs
      spec:
      containers:
      - name: kuard
      image: gcr.io/kuar-demo/kuard-amd64:1
      imagePullPolicy: Always
      args:
      - "--keygen-enable"
      - "--keygen-exit-on-complete"
      - "--keygen-exit-code=1"
      - "--keygen-num-to-gen=3"
      restartPolicy: OnFailure
```

실패한 컨테이너는 포드를 제거하지 않고, jobs 에 생성이 되어 있는 작업을 제거합니다. 최종적으로 jobs 작업을 테스트한다. 아래처럼 YAML 파일을 생성 후, kubectl 명령어로 실행한다. 파일 이름은 one-shot-job-success.yaml 로 한다.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: parallel
  labels:
      chapter: job

spec:
  parallelism: 5
```

```
  completions: 10
  template:
     metadata:
     labels:
     chapter: jobs
     spec:
     containers:
     - name: kuard
     image: gcr.io/kuar-demo/kuard-amd64:1
     imagePullPolicy: Always
     args:
     - "--keygen-enable"
     - "--keygen-exit-on-complete"
     - "--keygen-num-to-gen=5"
     restartPolicy: OnFailure
```

생성된 YAML 파일을 kubectl create 나 apply 명령어로 쿠버네티스 클러스터에 구성한다. 구성이 완료가 되면 올바르게 포드가 생성 후 실행 그리고 종료가 되는지 kubectl 명령어로 확인한다.

```
$ kubectl get pod
NAME                        READY   STATUS   RESTARTS   AGE
oneshot                     0/1   Completed   0          47h
parallel-d4njr              0/1   Completed   0          42h
parallel-hq2kd              0/1   Completed   0          42h
parallel-k2k2c              0/1   Completed   0          42h
parallel-knb5b              0/1   Completed   0          42h
parallel-p6jnt              0/1   Completed   0          42h
parallel-rzjft              0/1   Completed   0          42h
parallel-szzw9              0/1   Completed   0          42h
parallel-tftq5              0/1   Completed   0          42h
parallel-w8ptx              0/1   Completed   0          42h
parallel-x4xzm              0/1   Completed   0          42h
php-apache-758df75b6-dvx6p   1/1  Running     3          8d
$ kubectl get jobs
NAME       COMPLETIONS   DURATION   AGE
```

```
parallel   10/10      55s        42h
$ kubectl get rs
NAME                  DESIRED   CURRENT   READY   AGE
php-apache-758df75b6   1         1         1     13d
$ kubectl get rc
No resources found in default namespace.
$ kubectl delete jobs/parallel
job.batch "parallel" deleted
$ kubectl get pod
NAME                         READY   STATUS   RESTARTS   AGE
oneshot                      0/1   Completed   0          47h
php-apache-758df75b6-dvx6p   1/1  Running    3          8d
```

연습문제

kuard 이미지를 사용한다. 동시작업 수행은 10 개 총 50 번의 작업이 완료가 되어야 한다.

- gcr.io/kuar-demo/kuard-amd64:1

ConfigMaps(cm)

ConfigMaps 컨테이너나 혹은 애플리케이션이 사용하는 설정파일 내용을 저장 및 보관한다. 이를 통해서 모든 컨테이너에 설정 파일을 배포할 필요가 없으며, ConfigMaps 를 통해서 일괄적으로 배포 및 갱신이 가능하다.

ConfigMaps 는 두 가지 접근 방법을 지원한다. 첫 번째는 쉘 변수로 전달하는 방법, 두 번째는 파일 시스템 형태로 전달하는 방법이다. 아래 그림은 두 가지 방식을 나타내고 있다.

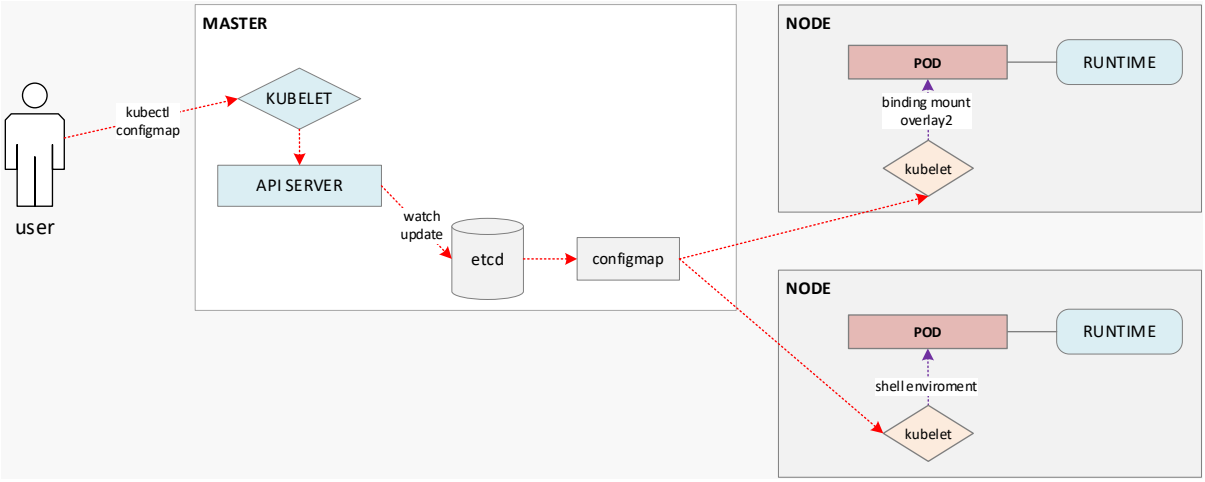


그림 31 configmap 구성

ConfigMaps 은 키=값 형태로 구성이 된다. 아래 예제를 확인한다.

```
$ kubectl create configmap
```

현재 구성이 되어 있는 configmap 정보를 확인한다. configmap 은 실제로 cm 라는 약어로 사용이 가능하다. kubectl 명령어로 configmap 객체를 생성한다.

```
$ kubectl create cm test
```

위의 명령어를 test 라는 configmap 자원객체를 생성한다. 생성한 자원객체는 데이터가 없기 때문에, 명령어에서 자원에 데이터를 입력하려면 다음처럼 명령어를 실행한다.

```
$ kubectl create cm test2 --for-literal=name=bora
```

위의 명령어는 configmap 에 test2 라는 객체를 생성 후, 내부에 name 이라는 키 이름을 생성 후 데이터를 bora 를 대입한다. kubectl describe 명령어로 확인해본다.

```
$ kubectl describe cm test2
Name:      test2
Namespace: default
Labels:    <none>
Annotations:  <none>

Data
====
name:
----
bora
Events:  <none>
```

또한 데이터를 이미 구성된 정보 기반으로 configmap 에 입력이 가능하다. 디렉터리 구조는 상관없으나, 파일이름 및 데이터는 다음과 같은 구조를 따라야 한다. 이 파일은 ui.properties 라는 이름으로 생성한다.

```
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

생성 후 kubectl 명령어로 다음처럼 실행한다.

```
$ kubectl create configmap game-config --form-file .
```

명령어 실행 후 kubectl 명령어로 올바르게 데이터가 configmap 에 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. configmap 를 아래와 같은 이름으로 생성한다. 이 설정 파일은 nginx 이미지에 사용한다. 이름은 configmap-nginx-test.yaml 으로 저장한다.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configmap
data:
  database: mongodb
  database_uri: mongodb://localhost:27017

  keys: |
    image.public.key=771
    rsa.public.key=42
```

configmap 를 사용할 Pod 를 생성한다. 이름은 configmap-nginx-pod.yaml 으로 저장한다.

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-env-var
spec:
  containers:
    - name: env-var-configmap
      image: nginx:1.7.9
```

```
    envFrom:
      - configMapRef:
          name: example-configmap
```

연습문제

다음 내용을 기존 내용에 추가하여 configmap 에 등록한다.

```
CentOS=RedHat
Ubuntu=Debian
Debian=GNU
Rocky=CentOS
```

Secrets(sc)

시크릿은 configmap 과 비슷한 기능을 제공한다. 다만, 시크릿 경우에는 configmap 처럼 데이터를 일반 문자열 형태로 저장하지 않으며, 인코딩(base64)된 형태로 저장이 된다. 그러한 이유는 시크릿에 저장된 데이터는 일반 사용자들이 확인을 하면 안 되는 아이피 주소, 암호 혹은 아이디 같은 민감한 데이터를 저장한다.

```
$ kubectl get secrets
NAME                  TYPE                                  DATA
AGE
default-token-mg4qm   kubernetes.io/service-account-token   3      15d
```

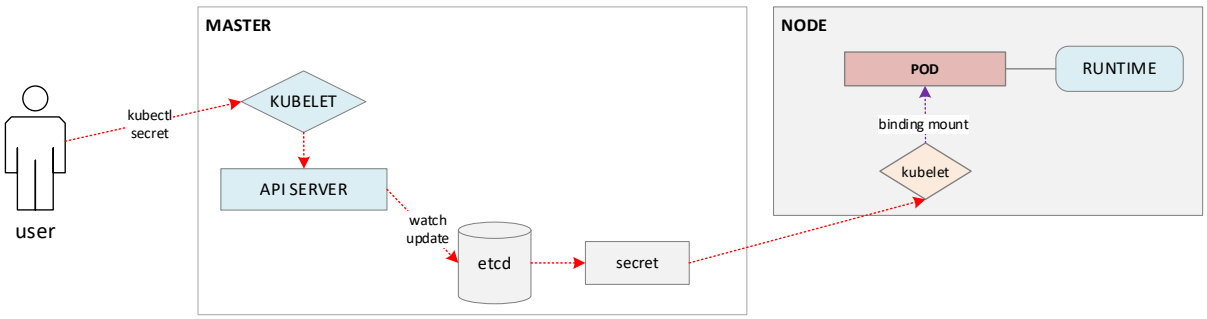


그림 32 시크릿

시크릿 생성하는 명령어는 다음과 같이 사용이 가능하다.

```
$ kubectl create secret generic test --from-literal=test=bora
```

위의 명령어를 통해서 test 라는 시크릿 이름을 생성한다. 생성 후 위에서 했던 것처럼, kubectl describe 를 통해서 확인한다.

```
$ kubectl describe secrets test
Name:      test
Namespace: default
Labels:    <none>
Annotations:  <none>

Type:  Opaque

Data
====
test:  4 bytes
```

configmap 경우에는 사용자가 입력한 데이터는 일반 문자열 형태로 출력이 되어서 확인이 가능하지만, 시크릿 경우에는 저장된 문자열이 인코딩이 되어 저장이 된다. 저장되는 방식은 대략 다음과 같은 과정을 통해서 인코딩 후 저장이 된다.

```
# echo -n 'redhat' | base64
cmVkaGF0
```

여러 개의 시크릿 파일이 구성이 되어 있으면, 다음과 같은 명령어로 처리가 가능하다. 먼저 일반적으로 시크릿 파일을 작성하는 YAML 형태는 다음과 같다. 파일 이름은 secret-password.yaml 으로 저장한다.

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-password
type: Opaque
```

```
data:
  username: b3BlbnNoaWZ0
  password: cmVkaGF0
```

위의 비밀번호는 base64 명령어로 구성이 된 명령어이며, 아이디는 openshift 비밀번호는 redhat 으로 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
```

컨테이너에 적용된 secret 은 일반적으로 컨테이너에 볼륨형태로 연결이 된다. 아래와 같이 파일을 생성한다.

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: bXktYXBw
```

```
  password: Mzk1MjgkdmRnN0pi
```

포드를 생성하여, 직접 secret 을 연결한다. 아래 내용을 secret-pod.yaml 으로 만들어서 저장한다.

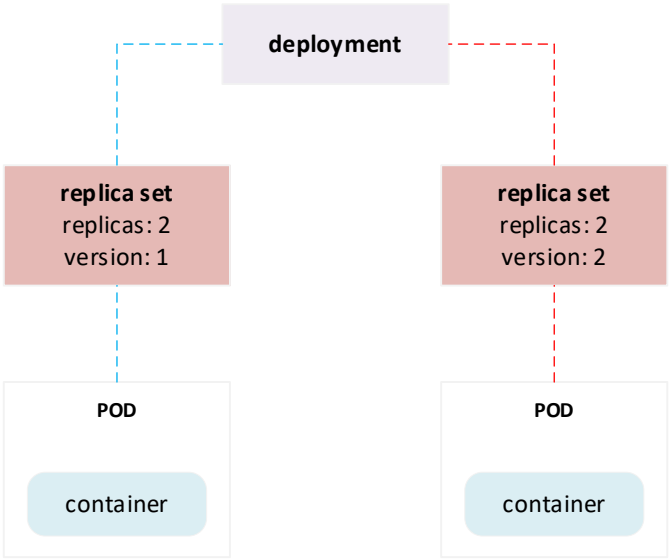
```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
spec:
  containers:
    - name: test-container
      image: nginx
      volumeMounts:
        - name: secret-volume
          mountPath: /etc/secret-volume
          readOnly: true
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret
```

올바르게 생성이 되면, 컨테이너 내부에서 "/etc/secret-volume"이 올바르게 구성이 되어 있는지 확인한다.

연습문제

deployment(deploy)

쿠버네티스 서비스는 구성설정(deployment)를 통해서 구성이 된다. 컨테이너 기반으로 kubectl run 명령어로 간단하게 생성이 가능하지만, 반복적이고 지속적으로 애플리케이션 서비스 구성하기 위해서는 deployment 서비스가 필요하다.



이 구성원은 다음과 같은 정보를 가지고 있다.

- 서비스에 사용할 POD 구성정보
- 복제할 POD 의 개수
- POD 및 컨테이너의 자원 할당
- POD 가 사용할 저장소 정보

일반적으로 많이 사용하는 deployment 설정은 아래와 같다. 여기에는 기본적인 deployment 에서 제공하는 포드, 컨테이너 그리고 선택자 및 리플리카(replica) 설정을 YAML 형식으로 하였다. 파일 이름은 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
```

YAML 파일을 생성하면 다음과 같은 명령어로 애플리케이션 생성이 가능하다.

```
$ kubectl create -f nginx-deployment.yaml
```

생성이 되면 올바르게 구성이 되었는지 kubectl 명령어로 확인한다.

```
$ kubectl get deployments -o wide
```

올바르게 구성이 되었으면, 다음 명령어로 이미지 버전을 변경한다. 이미지 정보는 deployment 에 등록이 되어 있으니 다음과 같은 방법으로 nginx-deployment 에서 사용하는 이미지 버전 정보를 변경한다.

```
$ kubectl set image deployment nginx-deployment nginx=nginx:1.8
```

기존에 사용하던 이미지 1.7.9 버전에서 1.8 로 변경한다. 하위 명령어 set 를 사용하여 적용되자마자 바로 이미지는 갱신이 된다. 아래 명령어로 올바르게 애플리케이션이 갱신이 되는지 확인한다.

```
$ kubectl rollout status deployment nginx-deployment
```

이미지 정보가 잘 변경이 되었는지 확인하려면 describe 명령어를 통해서 deployment 의 nginx-deployment 의 이미지 버전이 올바르게 변경이 되었는지 확인한다.

```
$ kubectl describe deployment nginx-deployment
```

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 1.7.9 에서 1.8 로 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
$ kubectl rollout history deployment nginx-deployment --revision=2
```

위의 명령어를 실행하면 리 비전 정보가 자세하게 화면에 출력이 된다.

```
$ kubectl rollout history deployment nginx-deployment --revision=1
deployment.apps/nginx-deployment with revision #1
Pod Template:
  Labels:    app=nginx
    pod-template-hash=5bf87f5f59
  Containers:
   nginx:
     Image:    nginx:1.7.9
     Port:    80/TCP
     Host Port:    0/TCP
     Environment:    <none>
     Mounts:    <none>
  Volumes:    <none>
```

만약 리버전 내용을 취소하고 이전 내용으로 롤백(rollback)를 진행하려면 다음과 같은 명령어를 실행한다.

```
$ 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
```

롤백이 완료가 되면 포드 상태를 확인한다.

연습문제

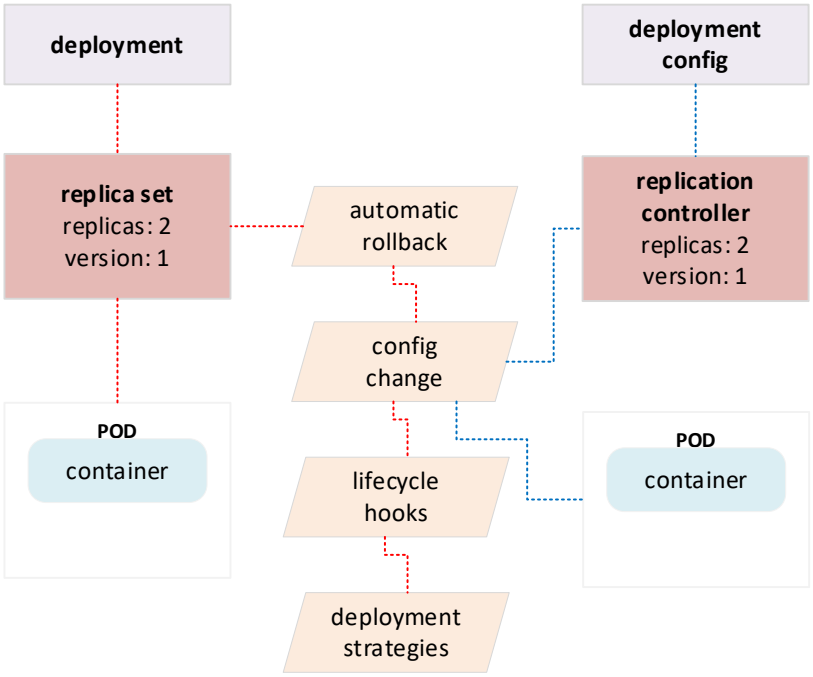
복제자(ReplicaSet, Replication Controller)

복제자는 단어 그대로 한 개 이상의 서비스를 복제 시 사용한다. 복제자는 서비스 사용량이나 혹은 사용자 설정에 따라서 개수를 늘리고 줄이고 한다. 복제자는 두 가지가 있는데, ReplicaSet, Replication Controller 가 있다. 현재 쿠버네티스는 ReplicaSet 사용을 권장하고 있다.

Replication Controller 는 ReplicaSet 처럼 복제를 한다. 하지만, 그 기준은 엄연히 많이 다르다. 아래는 간단하게 기능을 비교한 예시이다.

분류	Replication Controller	ReplicaSet	Deployment
설명	복제자 컨트롤러는 반드시 선택자가 동일해야지 복제를 한다. 현재는 ReplicaSet 으로 변경이 되었다.	복제자 묶음(set)은 기존 방식과 거의 동일하지만, 선택자가 하나가 아닌 여러 개를 지원한다.	Replication Controller 의 기능을 Deployment 가 대체한다. 자세한 내용은 뒤에서 더 설명. Deployment 는 ReplicaSet 과 같이 사용한다.
예제	os=rhel	os in (rhel, debian, ubuntu) os == rhel	
비고	위와 같이 동일해야지 복제자 컨트롤러는 복제를 시작한다.	지원 방식은 or, and 와 같은 방식으로 여러 개를 선택할 수 있다.	Replication Controller 에서 사용하는 rollout, rollback 기능을 Deployment 에서 사용한다.

초기 쿠버네티스는 DeploymentConfig, Deploy 같은 기능이 없었으며, 오직, ReplicaController 만 있었다. 그래서 초기 쿠버네티스는 Replication Controller 만 사용하였지만, 관리부분 때문에 Deployment 가 추가가 되었다. 결국, Replication Controller 와 Deployment 의 관계 및 기능이 모호해지면서 결국 ReplicaSet, Deployment 로 변경이 되었다.



복제자를 구성하는 방법은 다음과 같다.

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
  labels:
     app: kuard
     version: 1
spec:
  containers:
     - image: gcr.io/kuar-demo/kuard-amd64:1
     name: kuard
     ports:
     - containerPort: 8080
     name: http
     protocol: TCP
---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
```

```
  name: kuard
spec:
  replicas: 5
  selector:
     matchLabels:
     app: kuard
     version: "1"
  template:
     metadata:
     labels:
     app: kuard
     version: "1"
     spec:
     containers:
     - name: kuard
          image: "gcr.io/kuar-demo/kuard-amd64:1"
```

위의 YAML 파일은 2 가지 자원 내용이 있는데, 첫 번째는 포드 생성 정보이며, 두 번째는 생성된 포드의 복제 개수에 대해서 명시한 복제자 설정이다. 복제자에는 선택자가 선언이 되어 있으며, 해당 선택자가 찾고 있는 메타데이터는 "app=kuard", “version=1”로 되어있다. 이 두 개의 조건이 맞으면 총 5 개의 복제 포드를 생성한다.

위의 YAML 파일을 ‘kubectl apply -f' 명령어로 적용한다.

적용 후 ‘kubectl get pods’, ‘kubectl get rs’명령어로 올바르게 구성이 되었는지 확인한다.

여기에서 먼저 확인해야 될 사항이 있는데, 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, RC -> RS 로 복제자 사용시 권장하고 있다.

두 가지 기능의 큰 차이점은 바로 선택자 기능 부분이다. 앞에서 사용하였던 nginx 기반으로 테스트용도로 구성하도록 한다.

아래 YAML 은 Replication Controller(rc)복제자 설정이다. 파일 이름은 replication-controller.yaml 으로 저장한다.

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: rc-nginx-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: gcr.io/nginx:latest
        ports:
        - containerPort: 80
```

아래 YAML 은 ReplicaSet(rs) 복제자 설정이다. 파일 이름은 replicaset.yaml 으로 저장한다.

```
apiVersion: apps/v1
 kind: ReplicaSet
 metadata:
   name: rc-nginx-app
 spec:
   replicas: 3
   selector:
     matchLabels:
       app: rc-nginx-app
   template:
     metadata:
       labels:
```

```
        app: rc-nginx-app
        environment: dev
    spec:
      containers:
      - name: rc-nginx-app
        image: gcr.io/nginx:latest
        ports:
        - containerPort: 80
```

연습문제

외부 도메인 주소

쿠버네티스에서 외부에서 접근하는 방법이 여러가지가 있다. 그중 하나가 ExternalName 이다. ExternalName 은 외부에 있는 DNS 서버의 CNAME 레코드를 통해서 접근한다. 서비스에서 expose 명령어를 통해서 외부에 구성이 되어 있는 도메인으로 노출한다.

```
kind: Service
apiVersion: v1
metadata:
  name: external-database
spec:
  type: ExternalName
  externalName: database.company.com
```

해당 외부 도메인 서비스는 서비스 도메인으로 통해서 접근이 가능하다. 예를 들어서 위의 nginx.example.com 은 my-nginx-srv.nginx.svc.cluster.local 서비스 도메인을 통해서 접근한다. my-nginx-srv 는 서비스 이름이고 nginx 는 네임스페이스 이름이다.

외부에 접근하는 CNAME 레코드 A 레코드가 바라보고 있는 아이피를 통해서 해당 애플리케이션 노드에 접근한다. 위의 구성을 하기 위해서는 외부 DNS 서버가 필요하다. 일반적으로 외부 DNS 는 와일드 카드 도메인 기반으로 구성한다.

연습문제

DNS 서버가 구성이 안되어 있기 때문에 이 문제는 없음.

종합문제

시나리오

PoC 를 위해서 간단하게 싱글 클러스터 쿠버네티스 노드에서 간단한 웹 서비스 및 데이터 서비스를 구성한다. 이 서비스는 마이크로서비스 아키텍처 기반으로 최소 기능으로 구현이 되어 있으며 웹 페이지는 반드시 스케일링이 되어야 한다.

simple_web_service.php 를 php_app_image 이미지로 빌드한다

연습문제 설명

부록