GE - KUBERNETES ADMINISTRATOR 1

갱신: 2021-09-26

배포: RedHat Korea GLS

내용

1. 앤서블 기반으로 설치

쿠버네티스 설치 방법은 이전에는 소스 기반으로 설치 및 호스트 컴퓨터에 직접 설치를 하였다. 2014 년 이후로는 호스트 설치 방식에서 컨테이너 설치 방법으로 변경이 되면서 모든 쿠버네티스 서비스는 컨테이너 기반으로 설치 및 구성할 수 있도록 되었다.

현재는 kubeadm 명령어를 통해서 설치 방법에 대해서 표준화를 구축했고, 이 기반으로 쿠버네티스 클러스터 구축 및 구성을 한다. kubeadm 명령어는 여러가지 옵션을 제공하며 아래와 같이 설치 옵션을 제공한다

.

kubeadm 사용방법

쿠버네티스 설치 시 kubeadm 하위 명령어를 통해서 클러스터 구성 및 설치가 가능하다. 간단하게 kubeadm 명령어의 사용방법은 다음과 같다.

init

쿠버네티스 컨트롤 플레인을 구성합니다. init 명령어는 보통 마스터 노드를 구성 사용합니다. 이 하위 명령어는 클러스터에서 사용하는 워커를 지원하지 않습니다.

join

쿠버네티스 워커 노드를 클러스터에 추가합니다. 이 하위 명령어는 마스터 노드에 워커를 추가하는데 사용합니다.

upgrade

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

config

쿠버네티스 1.7.x 버전을 사용하거나 혹은 이보다 낮은 버전을 사용하는 경우, kubeadm upgrade 명령어를 통해서 업그레이드를 할 수 있도록 도와준다. 이 하위 명령어의 주요 목적은 구 버전을 신 버전으로 업그레이드할 수 있도록 설정 파일 구성을 지원한다.

token

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

reset

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

version

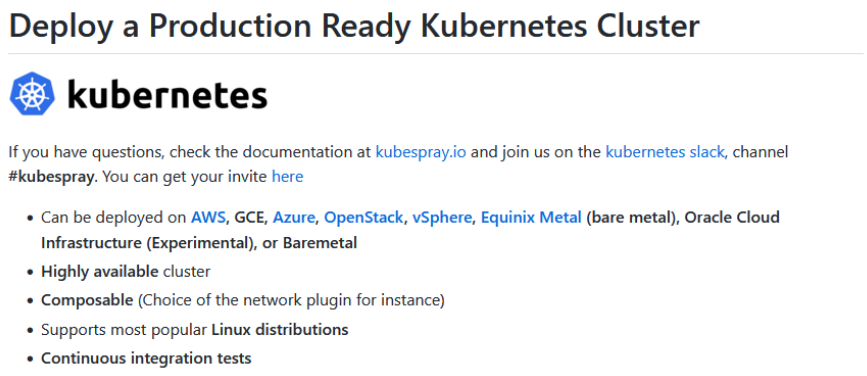
쿠버네티스 버전을 출력한다.

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.

kubespray

앤서블 기반으로 만들어진 쿠버네티스 설치 도구. kubespray 는 kubeadm 기반으로 설치는 아직 완벽하지 지원하지 않지만, 앤서블 기반으로 쉽게 컨테이너 기반 PaaS 를 구축할 수 있도록 도와주는 도구. 이 앤서블 설치 플레이북은 쿠버네티스에서 제공하는 공식 플레이북 설치 도구이다.



현재는 거의 대다수 실무에서 이걸 많이 사용하고 있다. 자세한 내용은 kubespray 웹 사이트에서 확인이 가능하다.

duststack

이 교육에서 사용할 앤서블 기반 설치 도구.

이번 교육에 사용할 용도로 작성이 되어 있으며 실무에서도 사용이 가능하도록 구성은 되어 있다. 이 교육에서는 플레이북 작성 방법에 대해서는 설명하지 않으며, 코드단위 교육을 원하시는 경우 레드햇 LC115(코드 리뷰 앤서블 교육)이나 혹은 정규 레드햇 앤서블 과정을 원하는 경우 RH294 교육을 확인 및 신청.

- duststack 의 구현 목적은 다음과 같이 되어있다.
- kubespray 와 같은 자동화 같은 도구를 꼭 만들어야 하는 경우
- 앤서블 기반으로 어떻게 컨테이너 시스템 Deployment 및 Management 를 해야 될지 고민이 있으신 경우
- kubespray 구성 이해를 하기 위해서

위의 목적을 가지고 있는 대상으로 이 교재 및 랩이 구성이 되어있다.

1. 앤서블 소개

앤서블은 2012 년에 발표가 되었으며, 현재는 레드햇이 인수하였다. 앤서블은 두 가지 제품으로 분리가 되어 있다. 앤서블은 YAML 문법 형태의 문법 그리고, 파이썬 기반의 모듈을 통해서 동작한다. 사용자가 작성한 파일은 플레이북(playbook)이라고 불리며, 플레이북 및 역할(role)를 통해서 시스템에서 발생하는 작업을 반복적으로 처리할 수 있도록 디자인이 되어 있다.

앤서블은 총 두가지 제품으로 나누어져 있다. 첫번째는 앤서블 코어는 기본적인 코어 기능만 제공하며, 대시보드 같은 관리 기능은 제공하지 않는다. 앤서블 타워는 기본 코어 기능에 관리기능 및 CI 기능을 제공하여 레드햇은 해당 제품에 대해서 구독기반(subscription)으로 지원 서비스를 제공하고 있다.

앤서블 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.

이 책에서는 앤서블 코어 기반으로 설치과정을 진행한다.

2. 설치 환경 준비

쿠버네티스 랩을 진행하기 위해서는 다음과 같은 컴퓨터 사양을 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 기능이 사용이 가능한 환경

랩이 구성되는 환경은 가상머신 기반으로 되어 있습니다. 가상머신이 사용이 불가능한 환경은 랩 구현이 불가능하며, 윈도우즈 클라이언트 및 맥에서는 랩 구성이 불가능 합니다.

OS	지원여부
Linux	**libvirtd 사용이 가능한 모든 배포판 지원. centos 혹은 fedora core 권장** **Centos 7, CentOS 8 버전 둘 다 지원. 다만, 파이썬 버전은 python 2.7 이상을 사용하는 것을 권장.** **Python 3.x 가 설치가 되어 있는 경우에는 dnf 명령어를 사용해서 설치 진행을 권장합니다.**
Windows Client/Server	**지원하지 않음**
OS X	**지원하지 않음**

버추얼 박스나 혹은 VMware Workstation 에서 사용이 가능 합니다. 하지만 가급적이면 리눅스 시스템에서 구성 및 테스트를 권장합니다. 여러가지 이유 때문에 올바르게 동작이 안될 수 있습니다.

설치 준비하기

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.

```
$ yum install git ansible
```

위의 프로그램 설치가 완료가 되면 git 명령어로 다음처럼 명령어를 실행한다.

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

GITHUB 에서 duststack 이라는 플레이북을 복제하여 로컬 컴퓨터, 즉 하이퍼바이저 호스트에 저장한다. 저장이 완료가 되면 가상머신을 생성하기 위한 과정을 진행한다. 아래 명령어를 실행하면 랩에서 사용할 가상머신을 리눅스 호스트에서 libvirtd 기반으로 자동으로 구성한다.

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

실행이 올바르게 되면 **k8s-master-1, k8s-node-1, k8s-node-2** 총 3 대의 가상머신이 설치 및 구성이 된다. 구성이 완료가 되면, ssh 공개/비공개키를 생성하여 서버에 전달한다.

```
$ cd library
$ ./ssh-keygensend kubernetes
```

키 생성이 완료가 되면 쿠버네티스 노드 설치를 시작한다.

```
$ ansible-playbook -i inventory/kubernetes playbooks/kubernetes.yaml
```

설치가 완료가 되면, 다음 명령어로 올바르게 가상머신이 구성 및 동작이 되었는지 최종적으로 확인한다.

```
$ virsh list
 4    k8s-master-1                   running
 5    k8s-node-1                     running
 6    k8s-node-2                     running
 7    k8s-utility-node               running
 -    ansible-classroom              shut off
 -    gls-lab                        shut off
 -    rh403-classrom                 shut off
 -    rhci-materials                 shut off
 -    rhel8.0                        shut off

$ for i in 110 120 130 140 ; do sshpass -p openstack ssh -o
StrictHostKeyChecking=no root@192.168.122.${i} hostname ; done
master-1.example.com
Warning: Permanently added '192.168.122.120' (ECDSA) to the list of known
hosts.
utility.example.com
Warning: Permanently added '192.168.122.130' (ECDSA) to the list of known
hosts.
node-1.example.com
Warning: Permanently added '192.168.122.140' (ECDSA) to the list of known
hosts.
node-2.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
k8s-utility-node	utility.example.com	kubernetes 에서 사용할 기타 서비스 설치 노드

준비가 완료가 되면, 설치를 진행한다. 명령어는 아래와 같다. **"-i"**옵션을 통해서 클래스 룸 인벤토리가 아닌, 실 대상 서버의 인벤토리를 선택하며, 쿠버네티스 설치를 위해서 **"kubernetes.yaml"**플레이북을 선택해서 설치를 진행한다.

```
$ ansible-playbook -i inventory/kubernetes playbooks/kubernetes.yaml
TASK [nginx-ingress : Create a custom resource definition for
GlobalConfiguration resource] ***
changed: [192.168.122.110]

TASK [nginx-ingress : Create a GlobalConfiguration resource]
*******************
changed: [192.168.122.110]

TASK [nginx-ingress : deploy NGINX Controller]
*********************************
changed: [192.168.122.110]

TASK [nginx-ingress : deploy NGINX DaemonSet]
*********************************
changed: [192.168.122.110]

TASK [nginx-ingress : Create a service for the ingress controller pods]
********
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
```

랩을 재구성 혹은 필요가 없는 경우, 다음처럼 명령어를 실행한다.

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

위의 명령어를 실행하면, libvirt 에 동작하는 가상머신은 영구적으로 제거가 된다.

```
$ cat <<EOF> /root/.vimrc
au! BufNewFile,BufReadPost *.u{yaml,yml} set filetype=yaml foldmethod=indent
autocmd FileType yaml setlocal ts=2 sts=2 sw=2 expandtab
EOF
```

3. 쿠버네티스와 표준 OCI 도구와 관계

Docker/CRI-O

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

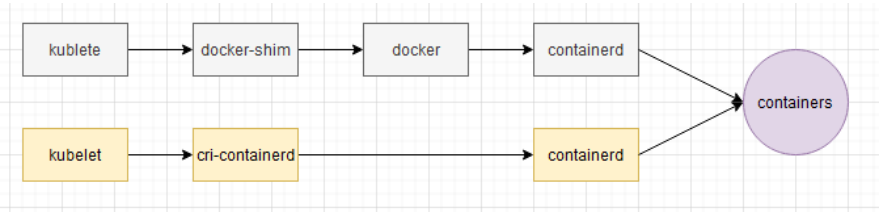
- Docker
- CRI-O
- Contained (OCI)
- rkt
- lxc/lxd

런타임은 사용자가 원하는 환경 혹은 프로그램 선택이 가능하며, 현재 사용중인 duststack 에서는 CRI-O, docker 만 지원하고 있다. 이 이외에 환경을 사용하기 위해서는 별도로 플레이북에 기능을 추가가 필요하다. 추후에는 플레이북에 lxc/lxd, rkt 를 지원할 예정이다.

일반적으로 많이 사용하는 런타임 환경은 Docker 및 CRI-O 환경이다.

현재 오픈소스 컨테이너는 **OCI(Open Source Container Initiative)**라는 표준 사양을 준수하며, OCI 대응하는 환경은 docker, CRI-O, lxc/lxd 이다. 쿠버네티스는 docker, CRI-O 런타임 환경에서 사용을 권장한다.

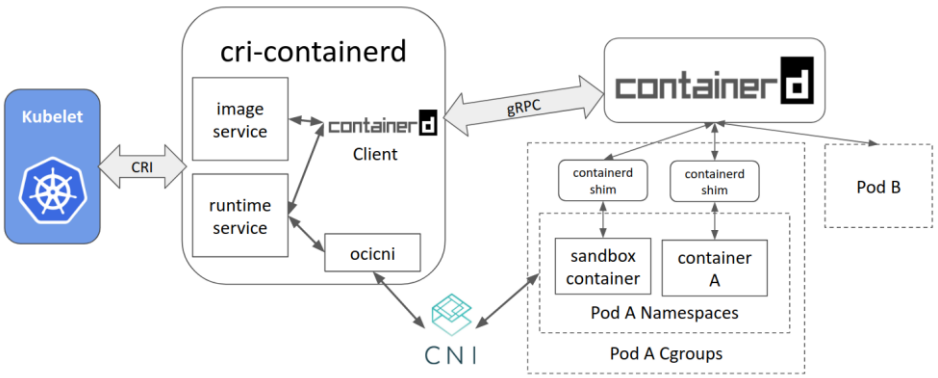
contained 환경을 지원하는 CRI-O, docker 는 OCI 를 지원하지만, 약간의 차이가 있다.



위의 그림은 쿠버네티스와 컨테이너 런타임이 어떠한 방식으로 동작하는지 알려주고 있다. 기존의 도커는 독자적인 방법을 사용하다가, 현재는 OCI 인터페이스를 지원하고 있으며, 그러면서 중간에 몇몇 계층이 추가가 되었다.

하지만, 여러가지 이유로 도커는 기능과 보안상 이유로 커뮤니티에서 이야기가 많았으며, 최종적으로 도커에서 CRI-O 컨테이너 환경으로 전환을 준비 및 시도하게 되었다. 현재 모든 컨테이너 환경 부분은 containerd 구성원 기반으로 컨테이너를 생성한다. 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.

CRI-O 의 컨테이너 동작 방식은 기본적으로 로칼에서 컨테이너를 실행하며, docker-ee 처럼 오케스트레이션 기능은 제공하지 않는다. 아래 그림은 CRI-O 기반으로 구성된 쿠버네티스 구성이다.



위의 그림은 쿠버네티스 웹 사이트에 발취하였으며, 이 그림에서 이야기하는 부분은 CNI, sandbox(namespace), resource control(c-group) 그리고 OCI 관계에 대해서 설명하고 있다. 우리가 사용하는 컨테이너 환경은 컨테이너 프로그램 즉, docker 나 CRI-O 같은 프로그램이 직접 컨테이너 생성 및 관리하는게 아니라 리눅스 커널의 여러 기능을 통해서 컨테이너 환경을 구성한다.

CRI-O 는 이미지 및 컨테이너를 직접 관리하며, CRI-O 컨테이너 환경은 표준 인터페이스인 containerd 와 gRPC 및 CNI(Container Network Interface)를 통해서 구성이 된다. containerd 는

생성된 컨테이너의 샌드박스(sandbox) 및 namespace 및 c-group 를 통합 관리, 이 기반으로 kubernetes 및 openshift 에서 사용하는 POD 를 구성한다.

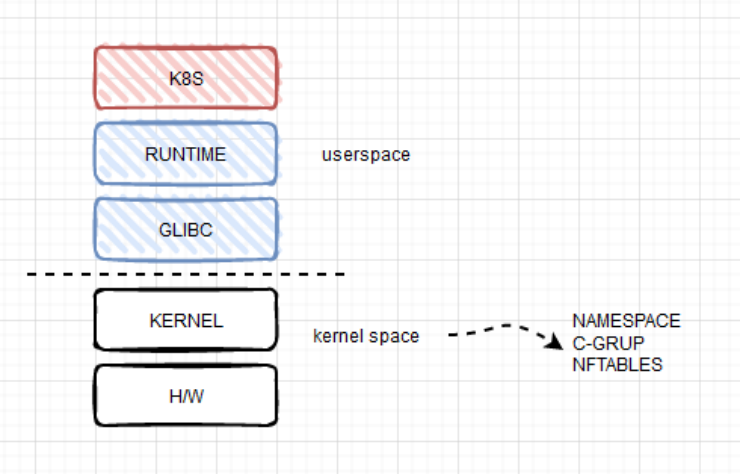
쿠버네티스는 클러스터를 구성하기 위해서는 리눅스 배포판에서 사용하는 리눅스 커널에서 다음과 같은 기능을 요구한다.

- namespace
- cgroup
- nftables

추가적으로 요구하는 기술 및 프로그램은 다음과 같다.

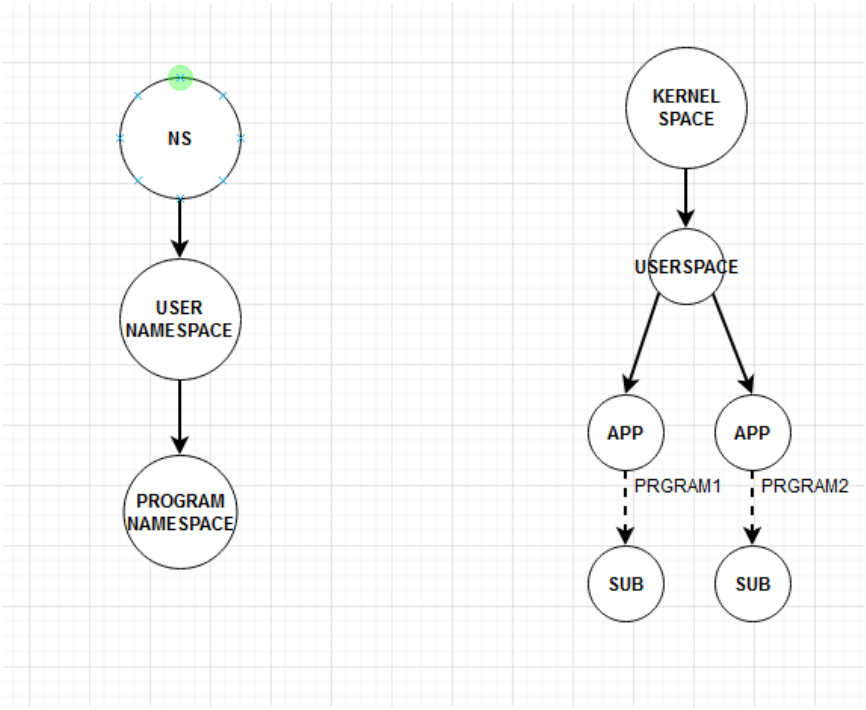
- selinux
- buildah
- podman
- skopeo

위의 추가 **기술 조건**은 반드시 필수는 아니며, 운영시에 같이 구축이 되어 있으면 컨테이너 플랫폼을 운영하기에 도움이 되는 도구이다.



NAMESPACE

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



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 가지지 못함
4. 컨테이너가 다른 노드로 이전 시, 기존에 사용했던 시간을 사용이 불가능

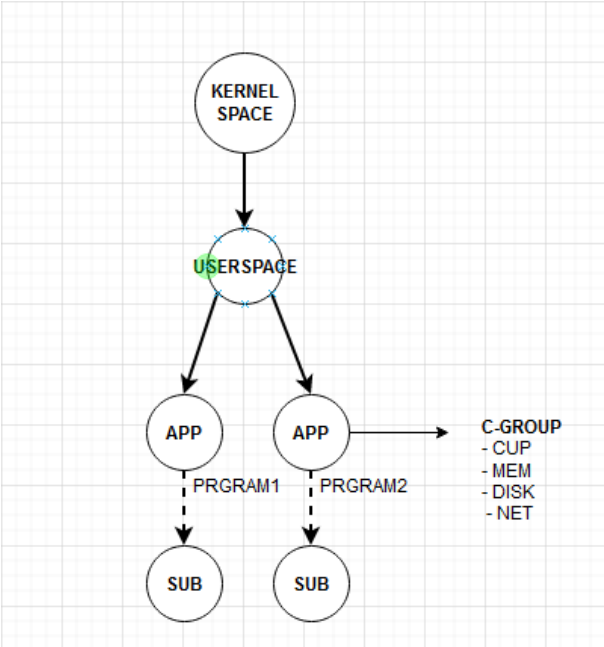
위와 같은 이유로 초기에 하나의 컴퓨트 노드에서 많은 컨테이너를 생성하기가 어려웠으며, 마이그레이션 같은 기능 구현이 어려웠다. 하지만, 2018 UTS 기능을 지원하는 네임스페이스가 리눅스 커널 5.6 에 도입이 되었으며, 레드햇 리눅스는 linux-4.18.0 버전부터 새로운 네임스페이스 기술을 지원한다.

C-GROUP

c-group 은 **control group** 의 약자이며, 실행되는 프로세스에 세밀하게 자원 사용량 제한 및 확인/감사 기능을 제공한다. 2006 년에 개발이 시작이 되었으며, 이때 개발명은 **"process containers"**라는 이름으로 개발이 되고 있다가 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)

설명을 조금 더 자세히 하기전에, cgroup 를 사용하기 위해서는 실제로 한가지 기술이 더 필요하다. 그 기술은 위에서 이야기한 네임스페이스이다. 위의 주요기능에 "자원제한", "할당", "제어" 부분을 구현하기 위해서는 사용자가 실행한 프로세스는 반드시 시스템과 분리가 되어서 실행 및 관리가 되어야 한다. 네임스페이스를 통해서 사용자 프로세스를 시스템과 분리하여 격리 및 관리한다.

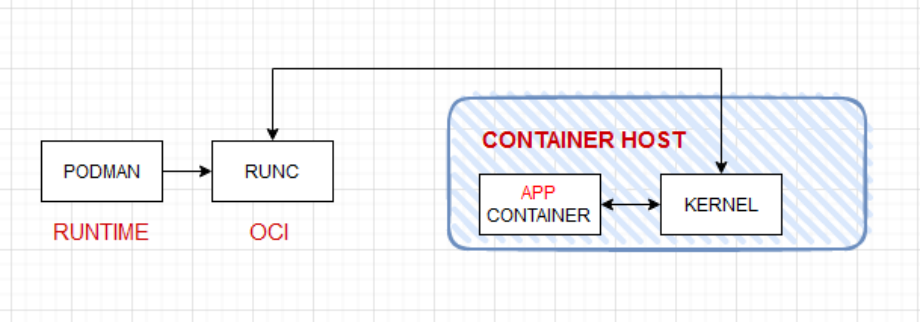


c-group 를 사용하면 자원은 커널 영역에서 관리하게 되며, 실행되는 프로그램은 네임스페이스를 통해서 분리 및 격리가 되면서 기존에 사용중인 사용자 프로세스에 영향이 없도록 분리가 되면서 실행이 된다. 실제로 컨테이너가 생성이 되면, 컨테이너에서 실행이 된 프로세스는 cgroup 과 네임스페이스를 통해서 동작이 된다.

Podman

podman 은 CRI-O 와 같이 컨테이너 생성 및 관리하는 도구이다. 하지만, podman 은 CRI-O 와 docker 처럼 API 기반으로 동작하는 컨테이너 엔진이다. podman 은 docker 에서 사용하는 명령어를 1:1 로 대응하며, docker 에서 사용하는 Dockerfile 및 docker-compose 기능도 podman 를 통해서 사용이 가능하다.

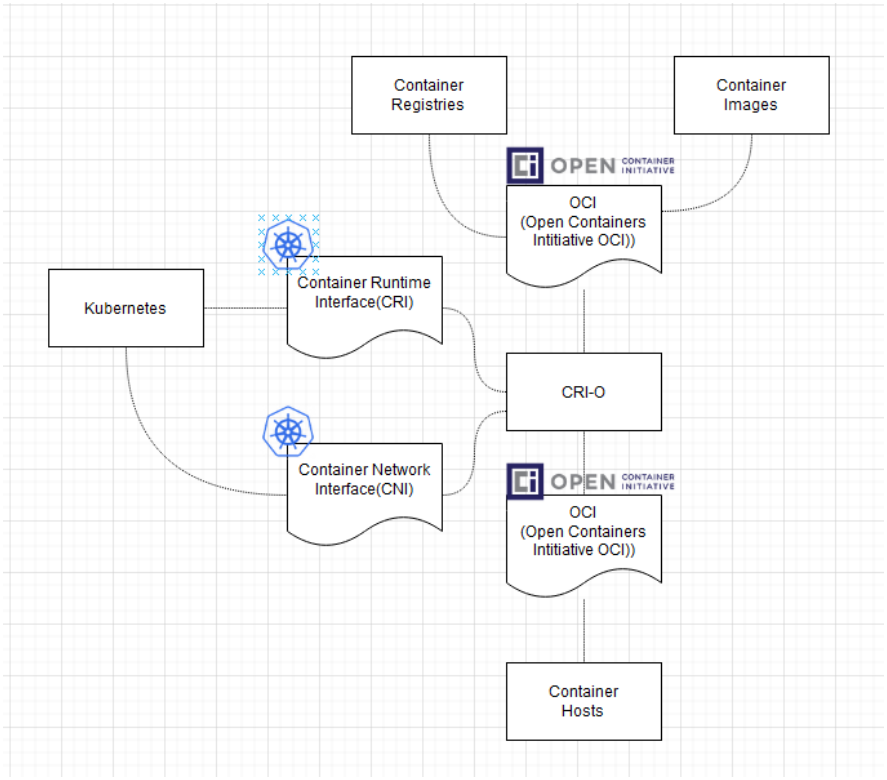
이쯤이면, 보통 궁금한 **부분이 "crio, containerd, podman"**의 차이점이다. 차이점에 대해서는 이 문단 마지막에서 비교 및 설명하도록 하겠다.



CRI-O

CRI-O 는 경량화 컨테이너 프로그램이다. docker, podman 처럼 컨테이너 런타임 환경을 제공이 주요 목적이다. 다만, CRI-O 는 다른 컨테이너 런타임 환경과 다르게 매우 가볍게 작성이 되어 있기 때문에, 많은 기능은 없으며, 기본 기능 위주로 pod 및 container 를 빠르게 생성 및 관리가 주요 목적이다.

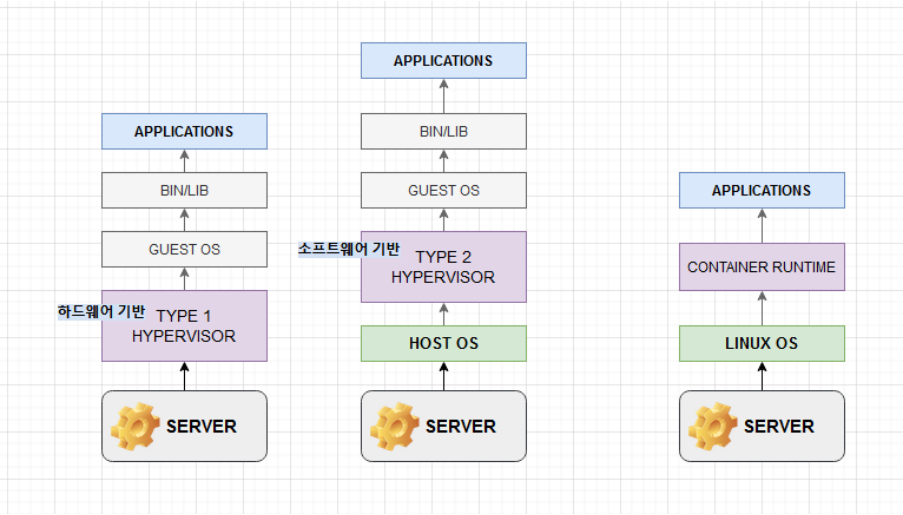
대다수 엔터프라이즈 컨테이너 오케스트레이션 엔진들은 CRI-O 런타임을 사용한다. CRI-O 의 약자는 Container Runtime Interface - Open Container Initiative 의 앞 글자만 따와서 CRI-O 라고 부른다.



CRI-O 는 download.opensuse.org 에서 내려받기가 가능하다.

4. 쿠버네티스 아키텍처 확인

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

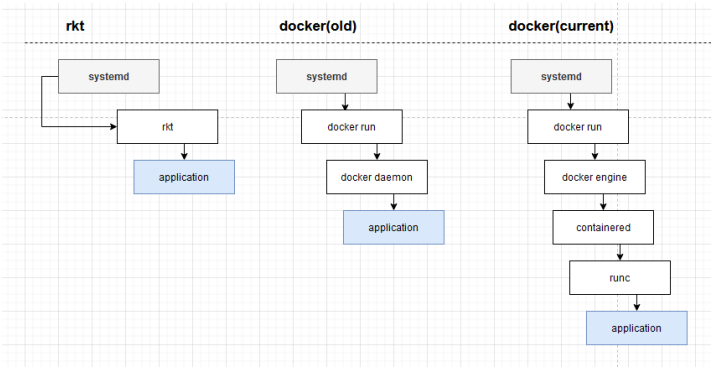


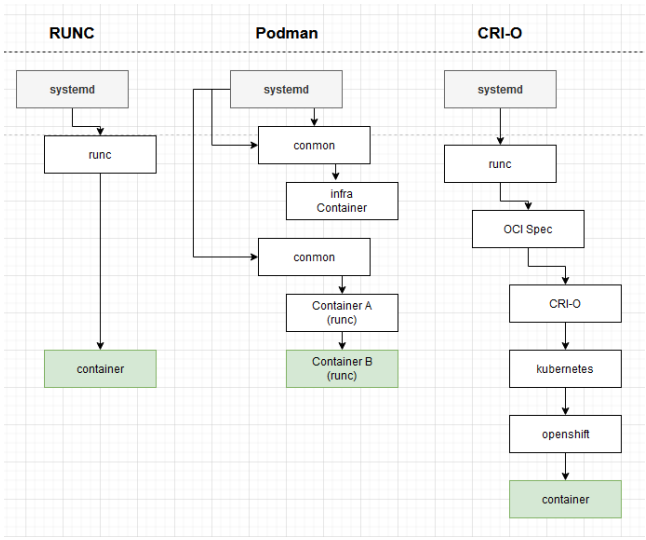
가상머신은 타입 1/2 로 구별이 되며, 이를 통해서 가상머신 구성 방법 및 성능이 다르다. 거기에 가상머신은 기본적으로 하드웨어의 기능이 요구가 되지만, 컨테이너는 별도로 하드웨어 기능을 요구하지 않는다.

쿠버네티스는 사용하기 위해서는 컨테이너 환경이 구성이 되어 있으면 바로 사용이 가능하며, 이를 기반으로 쿠버네티스 환경이 구성이 된다. 역시 가상머신처럼 별도의 하드웨어 환경을 요구하지 않는다.

쿠버네티스가 사용하는 컨테이너 런타임은 여러가지 형태로 제공이 되며, 이 부분에 대해서는 위에서 설명을 하였다. 각 런타임은 제공하는 환경이 조금씩 다르며, 소프트웨어 목적이나 환경에 따라서 사용자가 선택해서 사용해야 한다. 쿠버네티스는 사용이 가능한 런타임 환경이 모든 컨테이너 런타임을 지원하지는 않지만, 주요 컨테이너 런타임 프로그램을 지원한다.

아래 그림은 도커 및 rkt 에 대해서 설명한 그림인데, rkt 는 매우 간단한 환경을 가지고 있으며, 도커 경우에는 런타임 환경이 버전별로 다르기 때문에 환경에 요구되는 버전을 선택해서 사용해야 한다.





설치가 완료가 되었으면(아직 설치가 완료가 안되어도 괜찮다) 실제로 실습을 진행하기전에 쿠버네티스 아키텍처에 대해서 확인해보자. 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 구성이 되어 있다.

- kubelet
- pod
- node
- kube-proxy

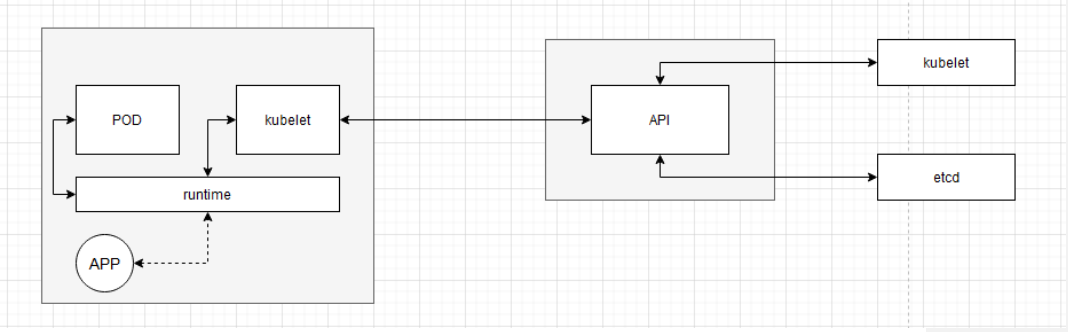
- kube-scheduler
- kube-controller-manager
- kube-apiserver
- pause
- coredns
- etcd

위의 구성원에 대해서 간단하게 기능과 역할을 설명하고 실제 서비스를 구성하도록 한다.

kubelet

유일하게 호스트에 설치되는 프로그램이다. 이 프로그램은 쿠버네티스 노드 에이전트 기능이며, 외부에서 들어오는 요청, 예를 들어 API 같은 부분은 kubelet 를 통해서 처리가 된다. kubelet 이 사용하는 포트 번호는 다음과 같다.

- tcp/6443
- http or https



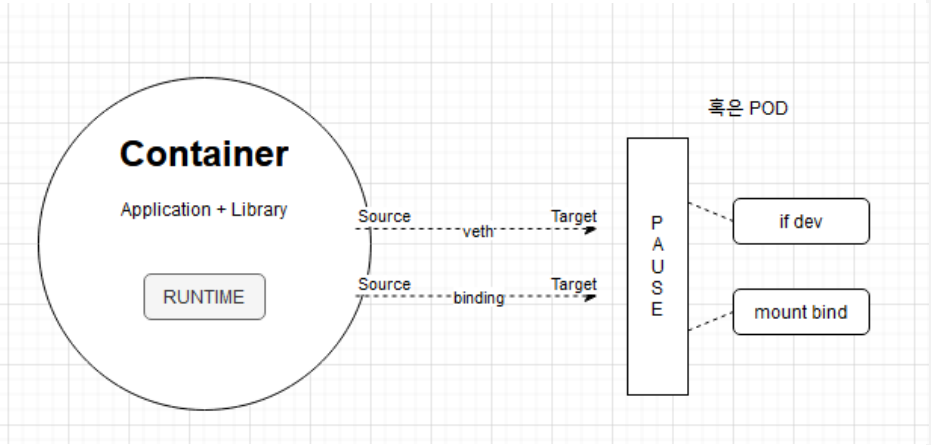
포트는 6443 포트를 사용하며, 이 포트로 들어오는 요청은 http, https 로 처리가 된다. 6443 포트로 들어온 트래픽 혹은 요청은 kube-apiserver 라는 컨테이너로 전달이 된다.

```
# 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              192.168.122.110:6
      users:(("kubelet",pid=519,fd=16))
LISTEN      0      128          [::]:10250               [::]:*
```

kubelet 는 컨테이너 **생성/삭제/조회**와 같은 기능을 제공하며, master, node 상관 없이 해당 데몬은 실행이 되고 있다. kubelet 은 기본적으로 kubelet{container runtime + pods + container}같은 구성으로 동작한다.

POD

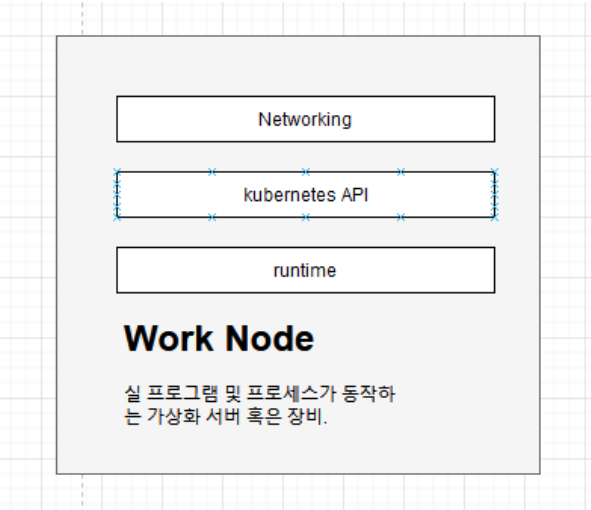
POD 는 컨테이너 생성시 같이 사용되는 POD 라고 불리는 컨테이너이다. 이 POD 는 어플리케이션 실행이 되는 컨테이너에 쿠버네티스 자원을 제공해 준다. 예를 들어서 서비스 네트워크, 포드 네트워크 그리고 스토리지 같은 정보를 제공한다. POD 의 주요 역할은 어플리케이션이 실행이 되고 있는 컨테이너를 격리와 자원 제공이 주요 목적이다.



쿠버네티스는 어플리케이션이 실행이 되면, 해당 어플리케이션에서 사용하는 자원을 포드가 구성한 네임스페이스를 통해서 제공하게 된다.

WORKER NODE

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

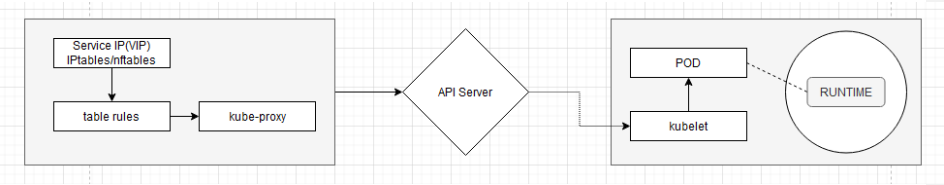


쿠버네티스 노드는 가상머신의 하이퍼바이저 같은 역할을 한다. 하지만, 위에서 이야기하였지만 별도의 하드웨어 기술이 요구가 되지 않기 때문에 가상머신의 하이퍼바이저보다 낮은 비용과 장비의 재활용성 및 부담이 매우 낮다.

어플리케이션이 실행이 되면, 어플리케이션은 노드에서 실행이 되며, 노드는 최소 한개의 마스터 노드를 통해서 클러스터를 구성한다. 노드는 그냥 노드라고 부르기도 하지만, 책이나 버전에 따라서 미니언(minion) 혹은 워커-노드(worker node)라고 표현하기도 한다.

kube-proxy

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



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.

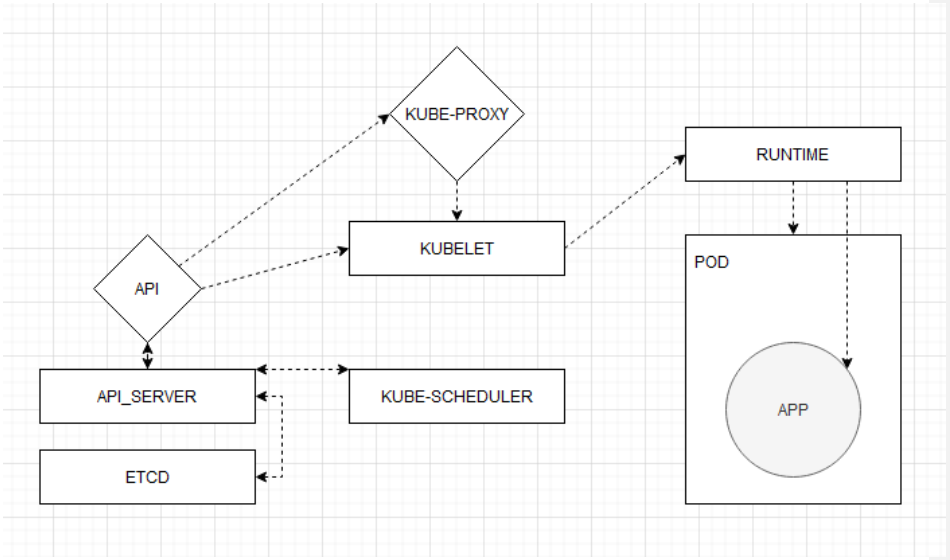
- 필터링(filtering)
- 점수(scoring)

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.

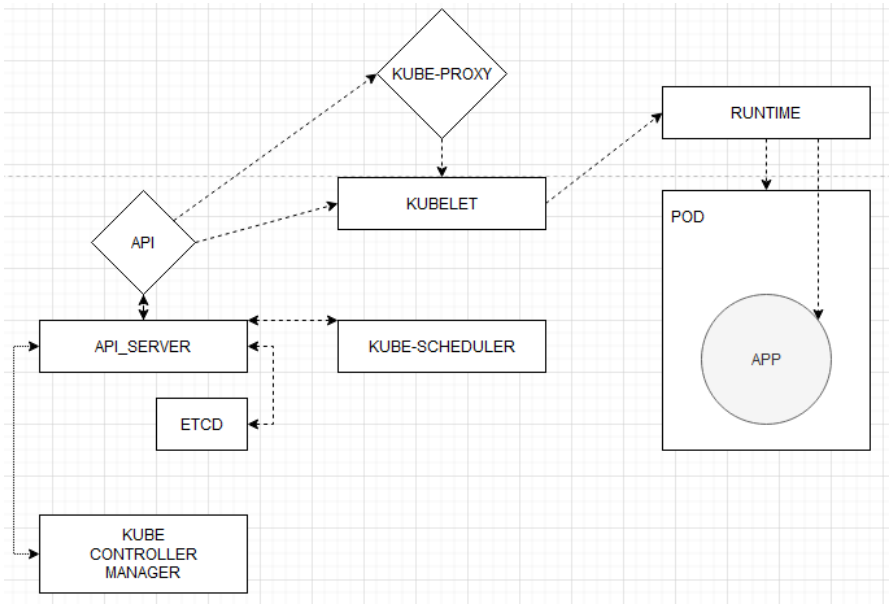

KUBE-PROXY
RUNTIME
KUBELET
POD
API
APP
API_SERVER
KUBE-SCHEDULER
ETCD

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.

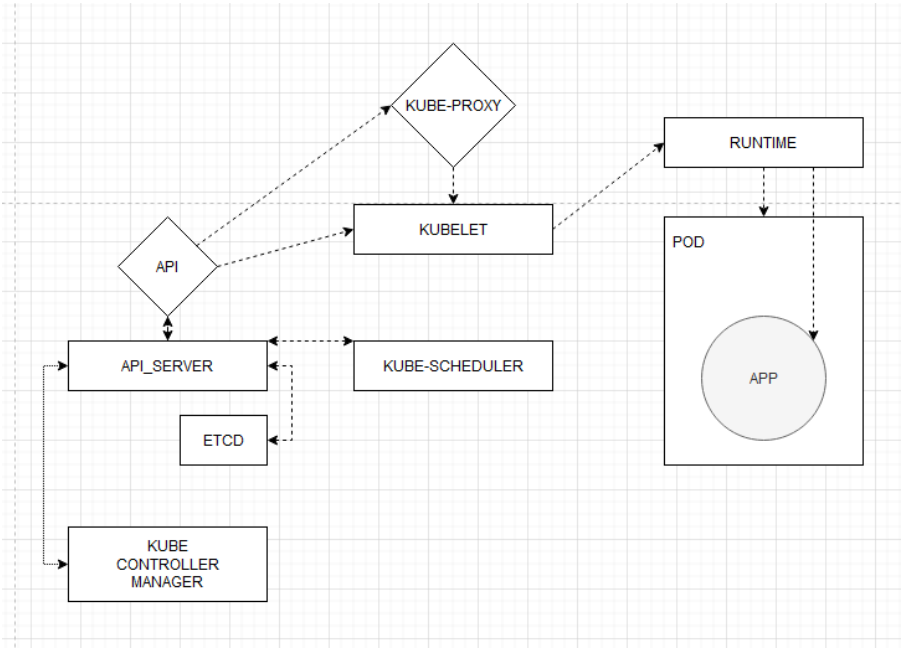
예를 들어서 클러스터에서 공유하고 있는 데이터에 대해서 지속적인 갱신, apiserver 를 통해서 상태 확인 혹은 변경이다. kube-controller-manager 를 대표적으로 많이 사용하는 서비스는 replication controller, endpoint controller, namespace controller 그리고 serviceaccount controller 이다.

이 서비스 기반으로 이중화 같은 서비스를 구성하기도 한다.



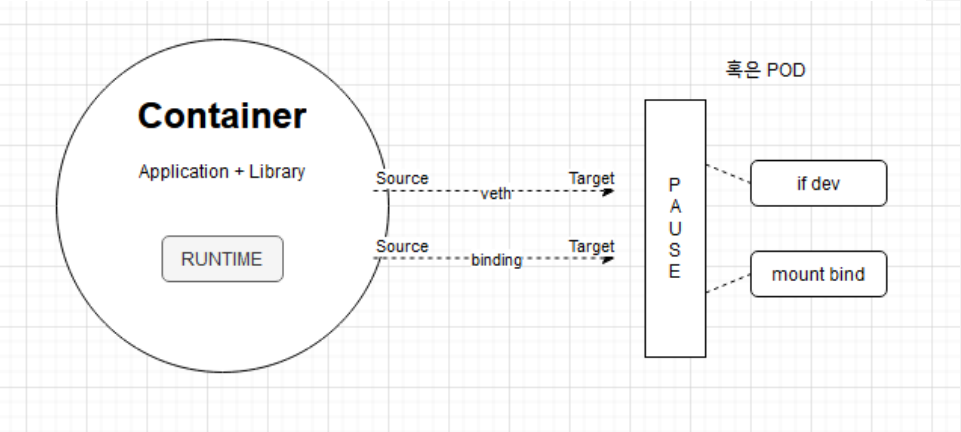
kube-apiserver

apiserver 는 사용자나 혹은 내부에 발생하는 작업들은 apiserver 를 통해서 처리하게 된다. 예를 들어서 포드 설정 정보나 복제 컨트롤러 같은 기능을 처리한다. API 를 통해서 처리되는 요청은 REST 기반으로 작업 요청이 된다.



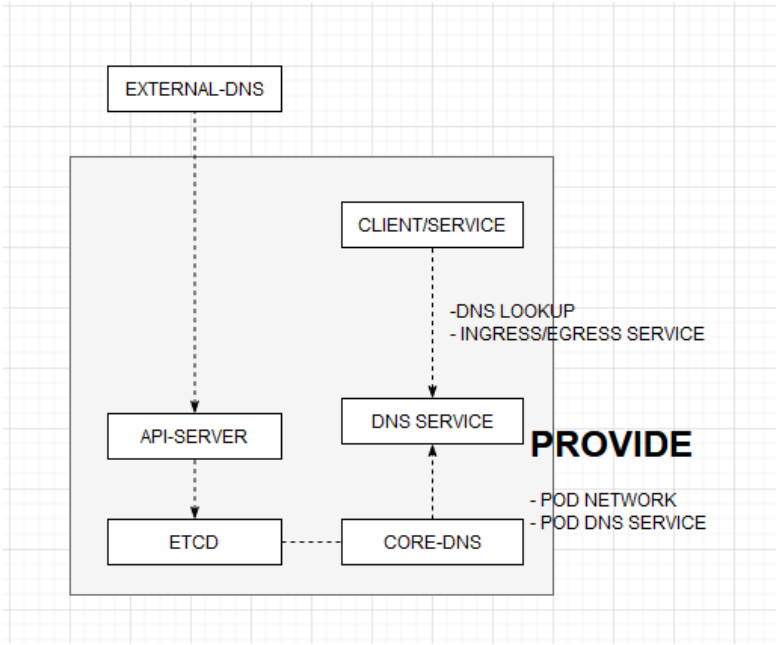
pause

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



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.



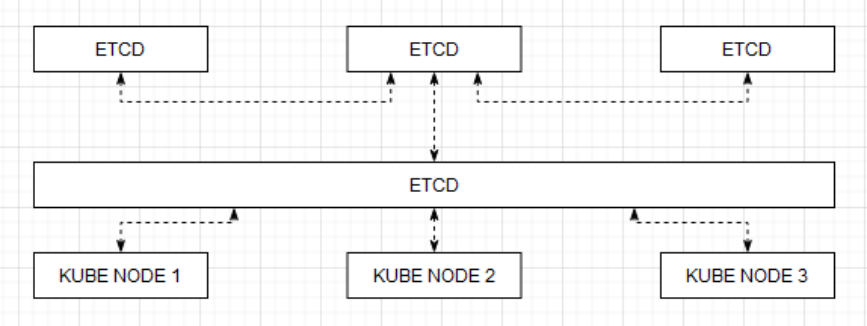
etcd

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.

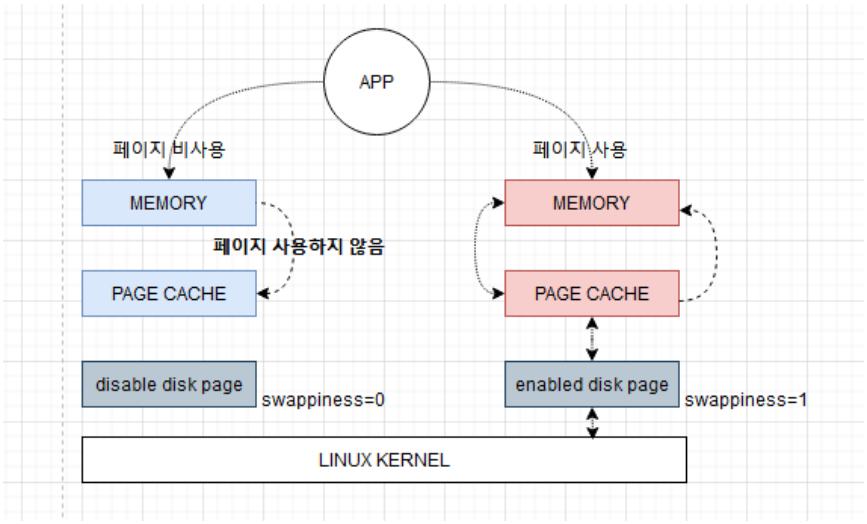
- **생성/제거:** etcd 에 발생되는 설정을 생성 및 제거를 합니다. 멤버 추가 및 설정 부분은 클러스터의 크기만 사용자가 명시해주면 됩니다.

- **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.



스왑 디스크 혹은 파티션 사용을 하면 안 되는 이유

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.



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

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

5. 주요 쿠버네티스 서비스 설명

pod

service

replica/replicaset

deployment

6. 쿠버네티스 빠르게 맛보기 1

기본 명령어 학습

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/제거 한다. 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 테스트 한다.

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

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
```

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.

```
# 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-context
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
*         kubernetes-admin@kubernetes   kubernetes   kubernetes-admin
```

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 실행하면 *kubectl* 명령어 갑자기 올바르게 동작하지 않는다. ~/.kube/config 파일을 확인하면 다음과 같이 변경이 되어 있다.

기존내용

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

변경내용

```
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
```

쿠버네티스는 위와 같은 방법은 현재 사용중인 네임스페이스, 즉 프로젝트를 선택 후 사용하던가 혹은 **--namespace** 라는 옵션을 통해서 매번 실행 시, 자원을 구성할 네임스페이스(프로젝트)를 선택해주어야 한다. 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.

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

# 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
kube-system       calico-node-kgpcp                                 1/1
      Running   3          5d6h
kube-system       calico-node-p7tng                                 1/1
      Running   3          5d6h
kube-system       coredns-66bff467f8-2js8j                          1/1
      Running   3          5d6h
kube-system       coredns-66bff467f8-kn8fc                          1/1
      Running   4          5d6h
kube-system       etcd-master-1.example.com                         1/1
      Running   3          5d6h
kube-system       kube-apiserver-master-1.example.com               1/1
      Running   4          5d6h
kube-system       kube-controller-manager-master-1.example.com      1/1
      Running   3          5d6h
kube-system       kube-proxy-h4mtx                                  1/1
      Running   3          5d6h
kube-system       kube-proxy-srvsq                                  1/1
      Running   3          5d6h
kube-system       kube-proxy-zvpbh                                  1/1
      Running   3          5d6h
kube-system       kube-scheduler-master-1.example.com               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

# 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
kube-system       kube-apiserver-master-1.example.com               1/1
      Running   4          5d6h   component=kube-apiserver,tier=control-
plane
kube-system       kube-controller-manager-master-1.example.com      1/1
      Running   3          5d6h   component=kube-controller-
manager,tier=control-plane
kube-system       kube-proxy-h4mtx                                  1/1
      Running   3          5d6h   controller-revision-hash=5f7b7d4f89,k8s-
app=kube-proxy,pod-template-generation=1
kube-system       kube-proxy-srvsq                                  1/1
      Running   3          5d6h   controller-revision-hash=5f7b7d4f89,k8s-
app=kube-proxy,pod-template-generation=1
```

"--all-namespaces"옵션을 사용하는 경우, 모든 네임스페이스에서 사용중인 pod 에 대해서 출력이 된다. 특정 네임스페이스의 자원만 확인이 필요한 경우, "--namespace"라는 옵션을 통해서 특정 네임스페이스의 이름을 명시한다.

"--show-labels" 옵션은 자원에 레이블이 설정이 되어 있으면, 해당 레이블을 같이 화면에 출력한다.

"--sort-by" 옵션은 출력 시 메타데이터의 이름을 내림차순으로 정렬해서 화면에 출력한다.

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

```
# kubectl describe nodes/master-1.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:

이하생략
```

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:         node-2.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://6f2b0ddc63854b02a7adf7fafd4d3c76715c9dd0ccab63f4
      Image:         nginx/nginx-ingress:edge

이하생략
```

연습문제

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

7. 쿠버네티스 빠르게 맛보기 2

위의 목차에서 생성된 "basic"라는 네임스페이스를 가지고 서비스 확인 및 추가적으로 네임스페이스를 생성하도록 한다. 현재 사용중인 basic 라는 네임스페이스에 기본적인 서비스를 구성하여 올바르게 구성이 되었는지 확인한다.

진행하기전에, vi 혹은 vim 입력기에 YAML 문법를 위하여 올바르게 띄어쓰기 및 형식을 맞출 수 있도록 ".vimrc"파일을 먼저 설정한다.

```
# tee ~/.vimrc << EOF
autocmd FileType yaml setlocal ai ts=2 sw=2 et
EOF
```

구성이 완료되면, vi 명령어로 다음처럼 YAML 파일을 작성한다. 만약, 해당 작업 컴퓨터에 vim 설치가 되어 있지 않으면, 다음 명령어로 vim 패키지를 설치한다.

```
# yum install vim
```

설치가 완료가 되면 테스트 용도로 사용할 "files"라는 디렉터리를 사용자 디렉터리 밑에 생성 후 작업을 진행한다.

```
# mkdir ~/files
# cd files
# vim httpd.yaml
apiVersion: v1
kind: Pod
metadata:
  name: httpd
  labels:
    name: test
    version: rev1
spec:
  containers:
  - image: centos/httpd
    name: httpd
    ports:
    - containerPort: 80
      name: http
      protocol: TCP
```

YAML 파일 작성이 완료가 되면 다음과 같은 명령어로 확인한다.

```
# kubectl create -f httpd.yaml
# kubectl get pods -w
NAME   READY   STATUS              RESTARTS   AGE
httpd   0/1       ContainerCreating   0          5s
httpd   1/1       Running           0          22s
```

축하합니다! 드디어, 처음으로 작성한 pod 가 올바르게 동작합니다. 이제는, 생성한 pod 의 상태를 확인해보도록 한다. 확인하는 명령어는 위에서 잠깐 사용하였던 *describe* 를 통해서 pod 의 상태를 조금 더 자세하게 확인한다.

```
# kubectl describe pod/httpd
Name:         httpd
Namespace:    first-namespace
Priority:     0
Node:         node-2.example.com/192.168.90.140
Start Time:   Sun, 10 May 2020 00:54:58 +0900
Labels:       <none>
Annotations:  cni.projectcalico.org/podIP: 10.244.17.69/32
              cni.projectcalico.org/podIPs: 10.244.17.69/32
Status:       Running
IP:           10.244.17.69
IPs:
  IP:  10.244.17.69
Containers:
  httpd:
      Container ID:
docker://72269e011c6cc7c0c937d46e5c4451f67cd0f990e1bbdea4c949f9d11d63e0a2
      Image:            centos/httpd
      Image ID:

이하생략
```

위의 정보에 대해서는 뒤에서 좀 더 자세하게 설명하도록 한다. 위의 정보는 많은 내용이 출력이 되었다. 위의 내용에서 아이피 주소만 확인을 원하는 경우, 다음과 같이 실행하면 아이피 주소만 확인이 가능하다.

```
# kubectl get pods httpd -o jsonpath --template {.status.podIP}
10.244.17.69
```

위의 아이피는 여러분이 구성한 pod 에서 사용중인 아이피 주소이다. 하지만 대역을 보면 아시겠지만, 현재 서버가 사용하는 대역과 전혀 다른 대역이기 때문에 접근이 불가능하다. 접근하기 위해서는 쿠버네티스에 현재 pod 가 사용중인 포트를 사용자가 접근할 수 있도록 **포워딩(forwarding)**요청을 해야 한다.

```
# kubectl port-forward httpd 80:80 &
Forwarding from 127.0.0.1:80 -> 80
Forwarding from [::1]:80 -> 80
```

포트 포워딩이 올바르게 되면 ***curl*** 명령어로 접근을 시도한다. 접근을 하기 위해서는 http://localhost 나 혹은 http://127.0.0.1 로 접근한다.

```
# curl http://localhost | head -10
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--
      0Handling connection for 80
100  4<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"><html><head>
8<meta http-equiv="content-type" content="text/html; charset=UTF-8">
9      <title>Apache HTTP Server Test Page powered by CentOS</title>
       <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7
      <!-- Bootstrap -->
      <link href="/noindex/css/bootstrap.min.css" rel="stylesheet">
1     <link rel="stylesheet" href="noindex/css/open-sans.css" type="text/css"
/>
0
0<style type="text/css"><!--
  4897      0     0   305k    0 --:--:-- --:--:-- --:--:--  318k
```

(23) Failed writing body

위와 비슷하게 메세지 출력이 되면, 여러분이 구성하신 쿠버네티스는 올바르게 구성이 완료되었다. 포트 포워딩 중인 *kubectl* 명령어를 종료하도록 한다.

```
$ pkill -f "kubectl port-forward httpd 80:80"
```

현재 실행중인 httpd 컨테이너가 실행이 되면서 발생한 로그를 확인 하도록 한다. 로그 확인하는 명령어는 하위 명령어인 *logs* 라는 명령어로 확인이 가능하다.

```
# kubectl logs pods/httpd
AH00558: httpd: Could not reliably determine the server's fully qualified
domain name, using 10.244.17.69. Set the 'ServerName' directive globally
to suppress this message
```

갱신되는 로그 메세지를 지속적으로 추적이 필요한 경우에는 "-f"라는 옵션을 통해서 추적이 가능하다. 현재 동작중인 컨테이너 내부에 파일복사가 필요한 경우, 다음과 같은 하위 명령어로 복사가 가능하다.

```
# tee index.html << EOF
Hello Kubernetes with CentOS
EOF

# kubectl cp --container httpd index.html httpd:/var/www/html/index.html
```

내부로 복사가 완료가 되면, 다시 'port-forward'를 통해서 httpd 서비스를 컨테이너 포트 80/TCP 에서 외부 80/TCP 로 전달한다.

```
# kubectl port-forward httpd 80:80
# curl http://localhost | head -10
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0     0     0     0     0      0      0 --:--:-- --:--:-- --:--:--
      0Handling connection for 80
100    29  100     29    0     0   1836      0 --:--:-- --:--:-- --:--:--
1933
Hello Kubernetes with CentOS
```

올바르게 실행이 되었는지 확인하려면, kubectl 명령어로 컨테이너 내부 이미지를 조회가 가능하다.

```
# kubectl exec httpd -- ls /var/www/html/index.html
/var/www/html/index.html
# kubectl exec cat -- cat /var/www/html/index.html
Hello Kubernetes with CentOS
```

확인이 완료가 되면 아래 명령어로 POD 를 제거한다.

```
# kubectl delete pods/httpd
# kubectl get pods
```

연습문제

8. 쿠버네티스 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
Repllicaset	서비스 복제 및 복구를 위한 자원 영역
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 기본적으로 쿠버네티스 자원은 생성이 된다. 위의 자원들을 사용하기 위해서 쿠버네티스 명령어로 구성이 어려우며, 이를 구성하기 위해서는 YAML 기반으로 작성을 해야 한다. 작성 방법은 일반적으로 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.

```
---
apiversion: apps/v1
kind: Deployment
metadata:
name: nginx
lables:
  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 어플리케이션 구성 시 사용하는 자원선언 파일이다. 쿠버네티스는 기본적으로 자원 선언 혹은 구성 시, Deployment 영역을 사용해서 구성하며, 이를 통해서 어플리케이션 복제 및 구성을 시도한다.

문법의 기본 구성원은 다음과 같다

```
---
apiversion: apps/v1
kind: Deployment
metadata:
```

먼저 맨 위의 "---"는 YAML 문법의 시작을 알리는 부분이다. 이 부분은 생략하여도 되지만, 일반적으로 맨 상단에 관습적으로 표시한다. 그 다음에 있는 "apiversion"릴리즈 버전 및 용도에 따라 다른데, 현재는 "apps/v1"이 기본 "apiversion"으로 자리가 잡혀 있다. 아래 "metadata"는 말 그대로 해당 자원의 메타정보를 구성하는 부분이다. 실제 서비스 구축에는 영향이 가지 않지만, 추가적인 정보를 넣을 때 사용한다.

```
name: test-nginx
lables:
  app: nginx
```

“name:”부분은 실제로 쿠버네티스에서 출력되는 컨테이너 혹은 POD 의 이름이다. 이 이름을 통해서 어떤 어플리케이션 혹은 POD 동작하는지 확인이 가능하다. **“label:”**세션은 메타 정보의 **“select:”**에서 사용하기 위해서 존재한다. selector 에서 사용하는 matchLabels 는 항상 name 를 검색한다.

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

“spec:”부분은 실제 어플리케이션이 동작할 환경을 설정 및 구성하는 부분이며, 앞에서 이야기하였던 selector 가 해당 세션에 같이 구성이 되어 있다. “matchLabels”항상 “labels”에 있는 값을 찾게 되는데, 여기에서는 “app: nginx”으로 되어 있다.

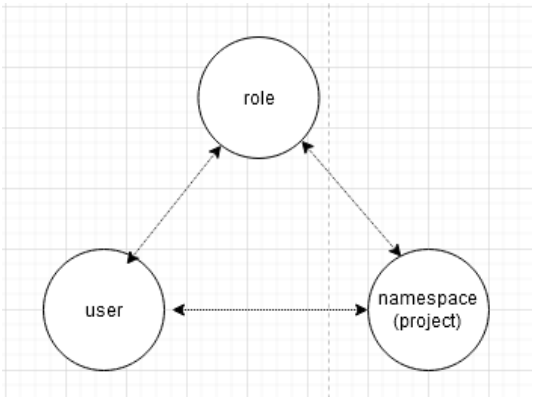
```
  spec:
    containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
      - containerPort: 80
```

하단에 보면 “containers:”라는 지시자가 보이는데, 이를 통해서 어떤 컨테이너 이미지를 사용하여 서비스를 구성할 지 결정한다. “image:”부분은 컨테이너 이미지를 가지고 있는, 이미지 레지스터 서버에서 어떠한 이름의 이미지 및 버전을 명시하며, 이 때는 “:”를 통해서 구별한다. 상단에 보면 “replicas”라는 지시자가 보이는데, 이 지사자는 POD 시작 시 총 몇 개의 어플리케이션을 실행할 지 결정한다.

포트번호는 “containerPort:”으로 명시하며, 여기서 사용하는 포트 번호는 실제 컨테이너 이미지안에 있는 어플리케이션이 사용하는 포트 번호를 이야기한다.

9. 쿠버네티스 컨텍스트 및 사용자 구성

이 책에서는 중요한 부분은 아니지만, 어떠한 방식으로 사용자 및 권한을 관리하는지 확인을 한다. 쿠버네티스는 기본적으로 다음과 같은 개념으로 사용자를 RBAC 기반으로 관리를 한다. 간단하게 도식으로 그리면 다음과 같은 그림으로 보통 그려진다.



사용자 계정 생성 및 RBAC

쿠버네티스에서 사용자는 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 그리고 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, secrtes 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: 노드자원 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.

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

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

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.

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

먼저 사용자를 생성한다. 생성할 사용자는 satellite, openshift 이다.

```
# useradd satellite
# useradd openshift
# 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 req -new -key satellite.key -out satellite.csr -subj
"/CN=satellite"
end of string encountered while processing type of subject name element #1
problems making Certificate Request

# openssl req -new -key satellite.key -out satellite.csr -subj
"/CN=satellite/O=$(groups)"
# 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.
```

사용자 키 기반 클러스터 인증을 위한 YAML 파일을 작성한다. 파일명은 satellite-cluster-account.yaml 으로 아래 내용을 복사 혹은 작성한다.

```
apiVersion: v1
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

kind: Config
preferences: {}
users:
- name: satellite
  user:
    client-certificate: /home/satellite/.certs/satellite.crt
    client-key: /home/satellite/.certs/satellite.key
```

Role 및 Cluster Role 생성

마지막으로 role 및 clusterRole 를 생성한다. 내용이 길어서 나머지는 github 에 있는 내용을 참고하도록 한다.

```
# cat role-cluster.yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: Role
metadata:
  name: list-deployments
  namespace: project-httpd-dev
rules:
  - apiGroups: [ apps ]
      resources: [ deployments ]
      verbs: [ get, list ]
...
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: list-deployments
rules:
  - apiGroups: [ apps ]
      resources: [ deployments ]
      verbs: [ get, list ]
```

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

```
# kubectl get roles.rbac.authorization.k8s.io --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
kube-system            kubeadm:kubelet-config-1.18
      2020-05-04T08:34:00Z
kube-system            kubeadm:nodes-kubeadm-config
      2020-05-04T08:34:00Z
kube-system            system::leader-locking-kube-controller-manager
2020-05-04T08:34:00Z
kube-system            system::leader-locking-kube-scheduler
      2020-05-04T08:34:00Z
kube-system            system:controller:bootstrap-signer
      2020-05-04T08:34:00Z
kube-system            system:controller:cloud-provider
      2020-05-04T08:34:00Z
kube-system            system:controller:token-cleaner
      2020-05-04T08:34:00Z
kubernetes-dashboard   kubernetes-dashboard
      2020-05-10T04:41:42Z
```

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.

```
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. 이 세가지가 최소 선택 서비스라고 생각하면 된다. 아래는 satellite 및 openshift 라는 사용자를 바인딩하는 과정이다.

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

openshift 사용자를 등록하기 위해서 아래처럼 작성해서 사용자를 쿠버네티스 클러스터에 등록한다.

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

문제없이 구성이 완료가 되면, 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 --replicas=1
```

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

```
project-httpd-dev
satellite: edit
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 사용자에게 .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

10. 기본 명령어

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

create/apply

쿠버네티스에서 자원을 사용 및 구성하기 위해서는 create 혹은 apply 를 사용해서 자원을 구성한다. crate 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 어플리케이션을 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 어플리케이션 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 httpd-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

1. 기존에 사용하였던 yaml 파일을 사용해서 다음처럼 서비스를 구성하세요.
2. vsftpd 이미지를 사용해서 yaml 파일을 작성합니다.
3. 이미지 파일은 아래의 주소에서 받기가 가능합니다.
4. https://hub.docker.com/r/fauria/vsftpd/
5. 받은 이미지의 repllica 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 파일 시스템 영역들은 UFS 라는 영역에서 동작하기 때문에 손쉽게 해당 파일시스템 계층 접근이 어렵다. 그래서 cp 명령어를 통해서 쿠버네티스 내부에서 동작중인 컨테이너 안쪽으로 파일을 복사한다.

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

```
$ kubectl 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 되어 있으며, 그 이외 내용은 아직 허용을 하고 있지 않다.

위에 대해서는 나중에 추후 더 설명하도록 한다. 일단, 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.

```
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-ngninx-second 라는 이름으로 80/TCP 로 my-ngnix 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.

연습문제

- centos 기반으로 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.

```
$ 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. 다른 역할은 없으며, 자비롭지 않게 명시된 모든 자원들을 쿠버네티스 시스템에 제거를 한다. 제거를 실행하는 명령어 이기에 사용시 매우 주의가 필요하다. 제거하는 방법은 보통 2 가지 방법이 있다.

- YAML 파일을 불러와서 그대로 제거
- 클러스터에 구성이 되어 있는 자원을 그대로 제거

바로 제거를 실행해보도록 한다. 먼저 POD 를 제거한다. 제거 시 특정 POD 만 제거하도록 selector 를 사용하도록 하겠다.

```
$ 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 /와 비슷하다고 생각하시면 된다.

```
$ 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 어플리케이션 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.

11. 고급 명령어

scale/rollout/rollback/history

시스템에서 사용하는 어플리케이션(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
```

확인을 해보면 옵션을 잘못 넣었다는 사실을 확인할 수 있다.

```
2  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 리비전 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

롤오버(in-flight multiple update)

연습문제

레이블 셀렉터 업데이트

연습문제

autoscale

자동확장 혹은 오토 스케일이라고 부르는 기능은 말 그대로 CPU 나 Memory 사용율에 따라서 크기를 조절 및 조정한다. 앞에서 이야기하였던 scale 하고 거의 차이가 없으며 명령어로 구성하는 경우 다음처럼 설정해주면 오토 스케일이 구성이 된다.

```
$ kubectl autoscale deployment.v1.apps/nginx-deployment --min=10 --max=15
--cpu-percent=80
```

오토 스케일을 사용하기 위해서는 자원에 대한 모니터링이 되어야 하며, 그러기 위해서는 메트릭 서버(matrix server)구성이 필요하다.

```
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!";
?>
```

작성이 완료가 되면 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
```

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-autoscale
```

현재 디플로이먼트와 잘되고 있는지 확인한다. 잘 구성되고 동작이 되고 있으면 다음 명령어로 HPA 구성 내용을 확인한다.

```
$ kubectl get hpa.v2beta2.autoscaling -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

drain

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

드레인을 하기 위해서 먼저 노드 이름을 확인한다.

```
$ kubectl get nodes
```

드레인을 실행한다.

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

작업이 완료가 되면 다음 명령어로 노드를 다시 서비스가 가능하도록 활성화한다.

```
$ kubectl uncordon node1.example.com
```

연습문제

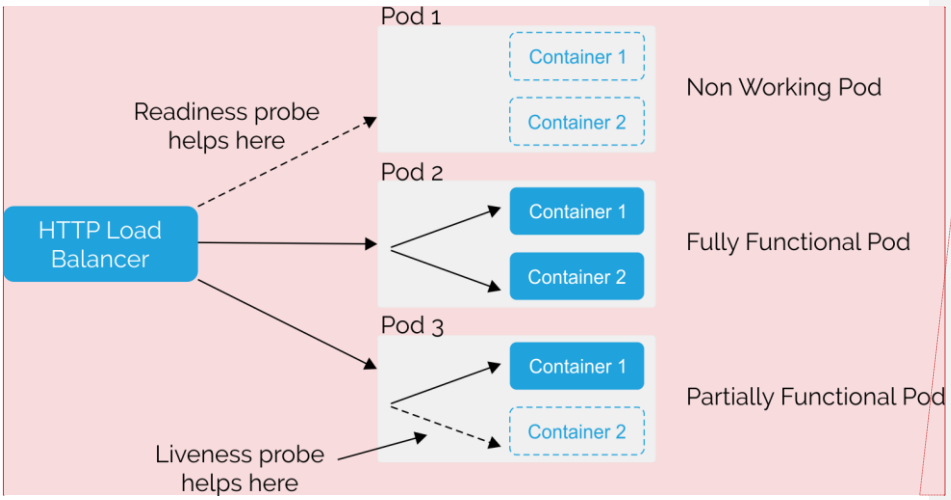
12. Service HealthCheck

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

위에서 logs, describe 하위 명령어로 현재 동작중인 컨테이너 상태에 대해서 확인은 가능하지만, 실제 내부에서 동작중인 서비스 확인은 어려운 부분이 있다. port-forward 명령어는 서비스 중인 컨테이너에서 포트 포워딩을 통해서 전달이 되는데 이 방법은 하나의 컨테이너에 대해서 확인만 가능하지 전체 컨테이너에 대해서 확인이 어렵다.



메모 포함[1]: 이미지 교체 필요

이를 확인하기 위해서는 컨테이너 이미지에 몇 가지 기능을 추가한다. 컨테이너에는 HTTP 서비스가 동작해야 하며 상태확인을 하기 위해서는 HTTP 에서는 다음과 같은 주소를 요구하게 된다.

- /ready
- /healthy

쿠버네티스는 2 가지 방법으로 서비스를 확인한다. 첫 번째는 liveness 두 번째는 readiness 이다. 이 두개의 방법은 다음과 같은 기능을 지원한다.

Liveness

liveness 는 서비스의 상태를 확인한다. 만약, liveness 를 통해서 모니터링을 하고 있는 서비스가 올바르게 동작하지 않는 경우, 해당 서비스 컨테이너에 대해서 제거(kill)를 하게 된다.

여기에도 조건이 있는데 개발자 혹은 서비스가 관리자가 설정한 횟수만큼 장애가 발생하면 제거를 실행한다.

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

Readiness

Liveness 와 비슷한 기능을 가지고 있으나, 이 기능은 컨테이너를 중지하지 않고 해당 Pod 로 트래픽 전달을 중지한다.

```
readinessProbe:
  httpGet:
    path: /status
    port: 8080
```

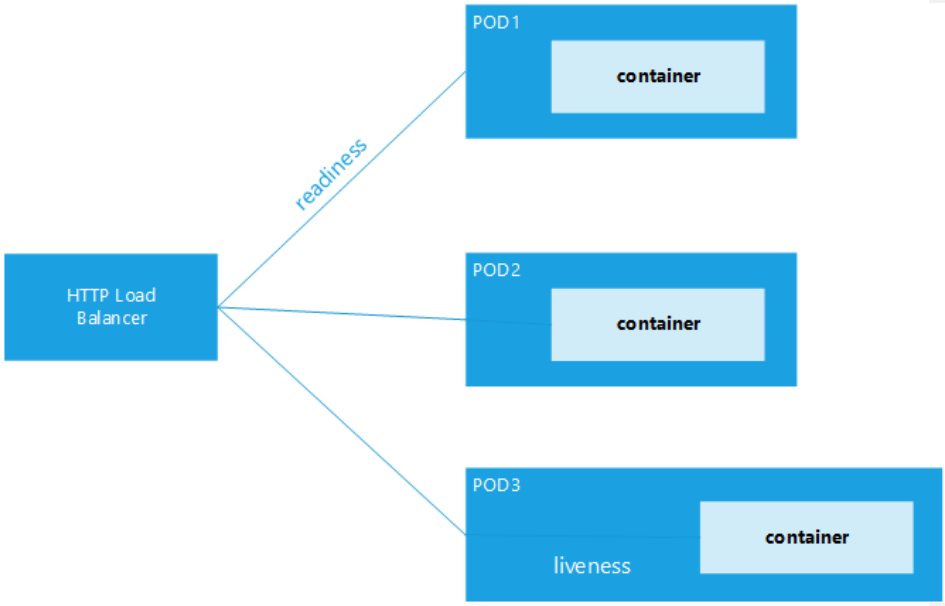
공통사항

위의 상태확인 기능은 http, exec, tcpSocket 기반으로 확인이 가능하다. 먼저, liveness 기반으로 서비스 상태확인을 해보도록 한다.

liveness

liveness 가 자원 상태를 모니터링하는 위치는 POD 에서 동작하는 어플리케이션 상태에 대해서 모니터링한다. 아래 그림을 보면 liveness 프로브는 POD 안에서 동작하는 애플리케이션 상태를

확인한다. 아래 POD3 그림을 보면 POD 내부에서 동작하는 liveness 가 확인이 가능하다. probe 는 별도의 프로그램이 아니며, kubelet 에서 어플리케이션에서 사용하는 특정 포트로 접근하여 올바르게 응답이 있는지 확인한다.



"/ready"경우는 쿠버네티스에서 트래픽을 전달하기 전에, "/ready"를 통해서 올바르게 서비스가 동작하는지 확인한다. 올바르게 동작이 되는지 확인한다. 시험으로 아래처럼 Pod 를 생성한다. 생성하는 위치는 **project-httpd-dev** 에서 생성한다. 파일 이름은 liveness-pod-1.yaml 이름으로 작성한다.

우리는 이 기능을 테스트하기 위해서 미리 만들어진

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-pod-1
spec:
  containers:
  - image: httpd
      imagePullPolicy: IfNotPresent
      name: liveness-pod-1
      command: ['sh','-c','echo liveness-pod-1 container 1 is running ;
sleep 4000']

      ports:
      - name: livness-pod-1
      containerPort: 80
      hostPort: 8080

      livenessProbe:
      httpGet:
      path: /
      port: 80
      initialDelaySeconds: 4
      periodSeconds: 10
```

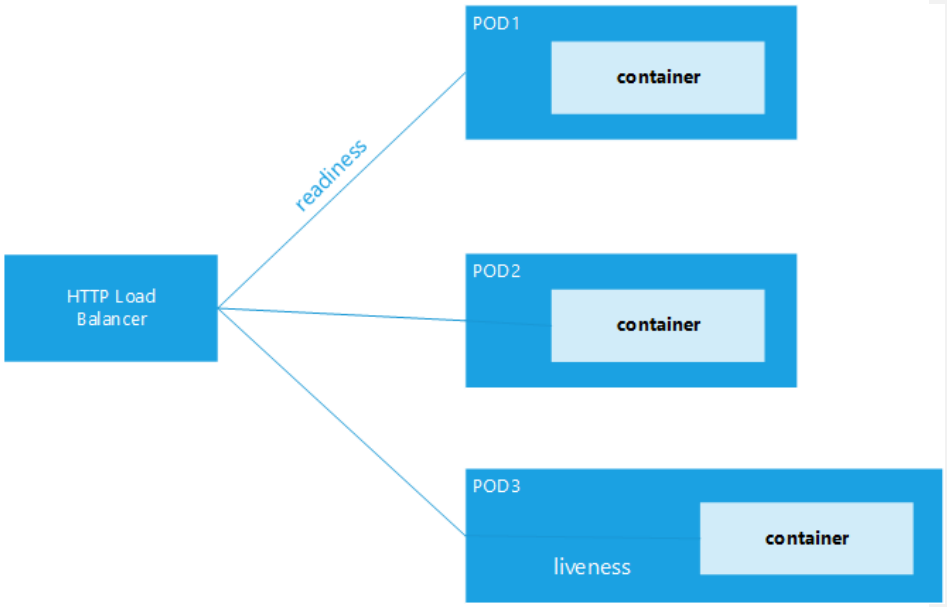
작성이 완료가 되면, *kubectl create -f liveness-pod-1.yaml*라는 이름으로 pod 생성을 한다. 생성이 완료가 되면 describe 하위 명령어로 컨테이너 상태를 확인한다.

```
# kubectl create -f liveness-pod-1.yaml
pod/liveness-pod-1 created

# kubectl describe pod/liveness-pod-1
Name:       liveness-pod-1
Namespace:  project-httpd-dev
Priority:   0
Node:       node-1.example.com/192.168.90.130
Start Time:   Sun, 10 May 2020 18:02:55 +0900
Labels:     <none>
Annotations:  cni.projectcalico.org/podIP: 10.244.171.138/32
            cni.projectcalico.org/podIPs: 10.244.171.138/32
Status:     Running
IP:         10.244.171.138
IPs:
  IP:  10.244.171.138
Containers:
  liveness-pod-1:
      Container ID:
docker://76eea38c8e06159f18ecbd3da8ecb773ff6d4a497d0ea67b55cb44257cbe3f45
      Image:      httpd
      Image ID:   docker-
pullable://httpd@sha256:c9e4386ebcdf0583204e7a54d7a827577b5ff98b932c498e9e
e603f7050db1c1
      Port:       80/TCP
      Host Port:  8080/TCP
      Command:
      sh
```

```
      -c
      echo liveness-pod-1 container 1 is running ; sleep 4000
      State:            Running
      Started:    Sun, 10 May 2020 18:07:10 +0900
      Last State:       Terminated
      Reason:     Error
      Exit Code:  137
      Started:    Sun, 10 May 2020 18:06:10 +0900
      Finished:   Sun, 10 May 2020 18:07:09 +0900
      Ready:            True
      Restart Count:  4
      Liveness:         http-get http://:80/ delay=4s timeout=1s
period=10s #success=1 #failure=3
      Environment:      <none>
      Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-
r92w5 (ro)
```

readiness



readiness 는 liveness 와 다르게 컨테이너를 확인하지 않고, 서비스 상태를 확인한다. 서비스 상태를 확인하기 위해서 컨테이너 내부에서 동작하는 프로그램은 readiness probe 가 접근이 가능한 URL 주소를 제공해야 한다.

이 healthCheck 방식은 동작중인 서비스가 올바르게 응답을 하는지 확인하여, 해당 서비스가 올바르게 동작하지 않으면, 해당 컨테이너를 중지하지 않고 일시적으로 컨테이너에 트래픽 전달을 중지한다. readiness 방식으로 HealthCheck 를 사용하기 위해서는 아래와 같이 YAML 파일을 생성

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

후 실행한다.

YAML 파일을 **'kubectl apply -f readiness.yaml'**로 적용 후 다음과 같은 명령어로 확인한다.

```
# kubectl apply -f readiness.yaml
pod/goproxy created
# kubectl get pods -w
NAME              READY   STATUS           RESTARTS   AGE
goproxy           0/1   Running           0           7s
httpd             1/1   Running           7           7d23h
liveness-pod-1   0/1    CrashLoopBackOff   1153       7d20h
goproxy           1/1   Running           0           14s
```

올바르게 실행이 되면 liveness 및 readiness 가 올바르게 적용이 되었는지 확인한다. 확인하는 명령어는 **'kubectl describe pods/goprox'**라고 실행한다.

```
Containers:
  goproxy:
      Container ID:
docker://aeaa08eb5252685faa2b7b8bffefa8f8d5c2eff9cd778a90f6e367c32d1be1e1
      Image:          k8s.gcr.io/goproxy:0.1
      Image ID:       docker-
pullable://k8s.gcr.io/goproxy@sha256:5334c7ad43048e3538775cb09aaf184f5e8acf4
b0ea60e3bc8f1d93c209865a5
      Port:           8080/TCP
      Host Port:      0/TCP
      State:          Running
      Started:    Mon, 18 May 2020 14:45:22 +0900
      Ready:          True
      Restart Count:  0
      Liveness:       tcp-socket :8080 delay=15s timeout=1s period=20s
#success=1 #failure=3
      Readiness:      tcp-socket :8080 delay=5s timeout=1s period=10s
#success=1 #failure=3
```

연습문제

13. 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 어플리케이션이나 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.

```
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 어플리케이션을 서비스(service)를 통해서 외부에 서비스할 수 있도록 서비스 자원을 구성한다.

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

연습문제

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

14. 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

15. 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 확인 하려면 다음 명령어로 확인이 가능하다.

```
# 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
Pod Template:
  Labels:         app=rook-ceph-osd-prepare
                  ceph.rook.io/pvc=
                  controller-uid=fa9f018c-ce40-4a34-bd4c-3585b3577b81
                  job-name=rook-ceph-osd-prepare-node-1.example.com
                  rook_cluster=rook-ceph
  Service Account:  rook-ceph-osd
  Init Containers:
   copy-bins:
      Image:      rook/ceph:v1.3.4
      Port:       <none>
      Host Port:  <none>
      Args:
      copy-binaries
      --copy-to-dir
      /rook
      Environment:  <none>
      Mounts:
      /rook from rook-binaries (rw)
  Containers:
   provision:
      Image:      ceph/ceph:v14.2.9
      Port:       <none>
      Host Port:  <none>
      Command:
      /rook/tini
      Args:
      --
```

임시로 Jobs 를 생성하여 컨테이너 생성을 시도한다. 이 컨테이너는 생성이 성공이 되지 않으며, 생성 중간에 인자 값 문제로 생성 및 실행이 실패한다.

```
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.

```
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
```

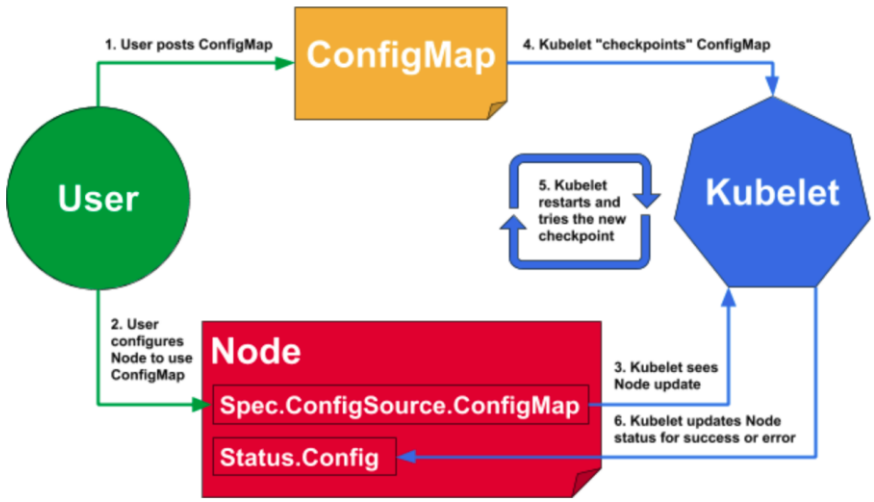
생성된 YAML 파일을 kubectl create 나 apply 명령어로 쿠버네티스 클러스터에 구성한다. 구성이 완료가 되면 올바르게 포드가 생성후 실행 그리고 종료가 되는지 kubectl 명령어로 확인한다.

```
# 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

16. ConfigMaps

ConfigMaps 컨테이너나 혹은 어플리케이션이 사용하는 설정파일 내용을 저장 및 보관한다. 이를 통해서 모든 컨테이너에 설정 파일을 배포할 필요가 없으며, ConfigMaps 를 통해서 일괄적으로 배포 및 갱신이 가능하다.



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 입력 하려면 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

Events:  <none>
```

연습문제

올바르게 실행이 되면, 위의처럼 파일을 통해서 생성된 데이터가 확인이 된다.

17. Secrets

시크릿은 configmap 과 비슷한 기능을 제공한다. 다만, 시크릿 경우에는 configmap 처럼 데이터를 일반 문자열 형태로 저장하지 않으며, 인코딩된 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
```

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.

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret-yaml
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.yaml
```

올바르게 등록이 되었으면, kubectl describe 명령어로 최종 확인한다.

```
# kubectl describe secret/mysecret-yaml
Name:         mysecret-yaml
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  6 bytes
username:  9 bytes
```

연습문제

18. deployment

쿠버네티스 서비스는 구성설정(deployment)를 통해서 구성이 된다. 컨테이너 기반으로 kubectl run 명령어로 간단하게 생성이 가능하지만, 반복적이고 지속적으로 어플리케이션 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.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
      app: nginx
spec:
  selector:
      matchLabels:
      app: nginx
  replicas: 3
  template:
      metadata:
      labels:
      app: nginx
      spec:
      containers:
      - name: nginx
      image: nginx:1.7.9
      ports:
      - containerPort: 80
```

YAML 파일을 생성하면 다음과 같은 명령어로 어플리케이션 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 어플리케이션이 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 ar
Waiting for deployment "nginx-deployment" rollout to finish: 1 old replicas ar
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
nginx-deployment   3/3   3            3           91m   nginx        nginx:1.7.9
```

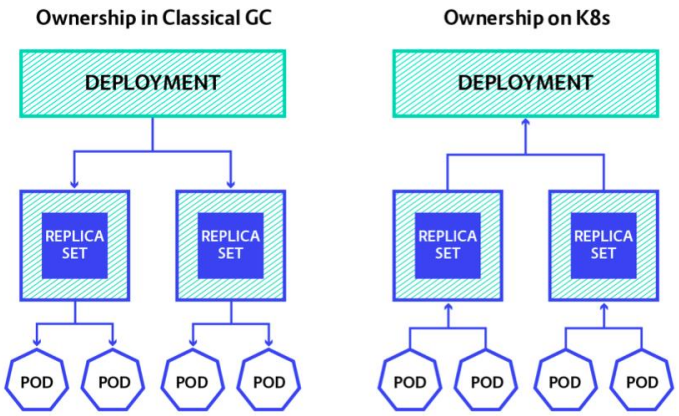
롤백이 완료가 되면 포드 상태를 확인한다.

```
$ kubectl top pod --heapster-namespace=myns --all-namespaces --containers
```

연습문제

19. 복제자(ReplicaSet)

복제자는 단어 그대로 한개 이상의 서비스를 복제시 사용한다. 복제자는 서비스 사용량이나 혹은 사용자 설정에 따라서 개수를 늘리고 줄이고 한다. 복제자 사용 방법은 ReplicaController Set(RCS)에서 관리한다.



복제자를 구성하는 방법은 다음과 같다.

```yaml
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 rc 복제자 설정이다.

```
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 rs 복제자 설정이다.

```
apiVersion: extensions/v1beta1
 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

20. 스토리지

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 /srv/nfs 에 구성한다.

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

구성된 디렉토리 위치를 exports 를 통해서 외부에 노출한다.

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

exports 파일에 등록된 정보를 nfs server 에 갱신한다.

```
# exportfs -avrs
```

위의 주소로 접근하는 경우, SElinux 에서 차단을 한다. 임시적으로 SELinux 를 중지한다. 일시적으로 중지하는 명령어는 setenforce 0 으로 일시적으로 감사모드로 전환한다.

사용할 NFS 디렉토리를 사용하기 위해서는 쿠버네티스 인프라에서 PV, PVC 정보를 등록한다. PV 는 백엔드 드라이버이며, 이를 통해서 컨테이너에 스토리지를 제공한다. 지금은 CSI(Container Storage Interface)통해서 별도의 드라이버 구성없이 표준 인터페이스를 통해서 제공한다. 아래 내용으로 PV 파일을 작성 후, 쿠버네티스 클러스터에 스토리지 인터페이스를 등록한다.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/mnt/data"
```

생성된 YAML 파일은 kubectl create -f <filename>으로 스토리지를 등록한다. PV 로 드라이버 구성이 완료가 되면, PVC 를 생성하여 컨테이너가 사용할 수 있도록 구성한다.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

생성된 PVC 를 kubectl create -f <filename>으로 등록한다. 등록이 되면 POD 를 구성하여 실제 컨테이너에 스토리지를 연결 및 구성한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

구성된 스토리지를 POD 에 구성한다. POD 를 통해서 배포된 스토리지를 유틸리티 서버를 통해서 확인이 가능하다.

연습문제

21. 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 어플리케이션 노드에 접근한다. 위의 구성을 하기 위해서는 외부 DNS 서버가 필요하다. 일반적으로 외부 DNS 는 와일드 카드 도메인 기반으로 구성한다.

연습문제

22. 종합문제