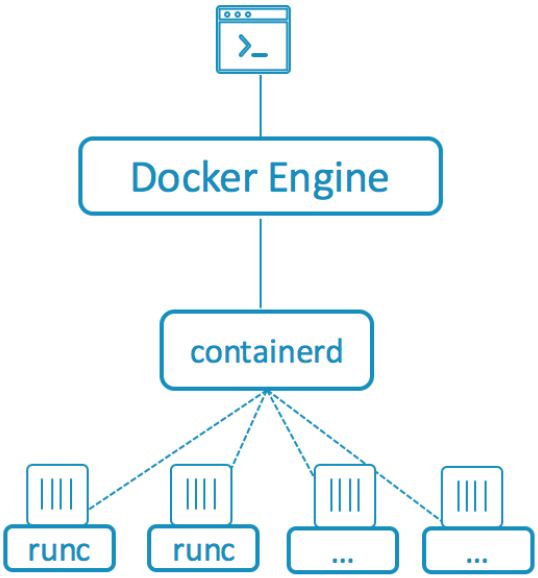


Same Docker UI and commands

User interacts with the Docker Engine

Engine communicates with containerd

containerd spins up runc or other OCI compliant runtime to run containers

runc : container 를 생성하고 실행하는 low level 런타임, OCI runtime spec 준수

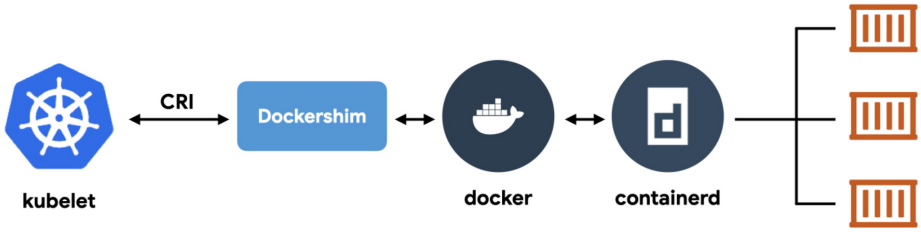
containerd: high level container run time,
도커레지스트리에서 이미지를 가져오고, 도커 네트워크및 스토리지 관리 그리고 컨테이너 실행을 위해서 저수준 런타임인 runc 를 실행하고 관리.
플러그인을 통해 CRI 를 준수하므로 kubernetes 런타임으로 사용할수 있다.

docker engine: 도커 이미지를 관리하고 컨테이너 실행을 위해 containerd 데몬과 통신하여 runc 기반으로 컨테이너를 실행,

OCI - 컨테이너 생태계를 위한 표준을 만들기 위한 노력중 하나 만들어지게 되었는데 OCI 아이디어는 컨테이너가 무엇인지, 무엇을 할수 있는지를 표준화 한것, 즉 OCI spec을 준수하는 다양한 런타임중에서 자유롭게 선택할 수 있다.

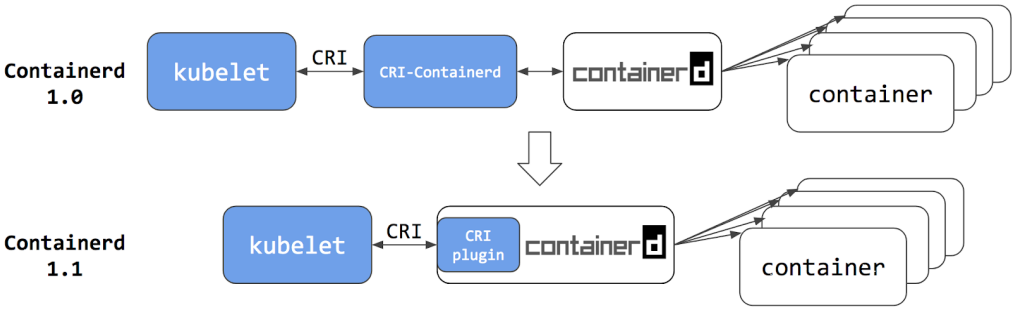
CRI - kubernetes 프로젝트에서 만든 API
- 컨테이너를 만들고 관리하는 다양한 런타임을 제어하기 위해 쿠버네티스에서 사용하는 컨테이너 런타임 인터페이스 kubernetes 프로젝트가 각 런타임에 대한 지원을 개별적으로 추가해야 하는 대신 CRI API 는 쿠버네티스가 각 런타임과 상호작용하는 방법을 설명한다.

주어진 컨테이너 런타임이 CRI API 를 구현하면 런타임을 원하는 대로 선택하여 컨테이너를 생성하고 실행할수 있다.
containerd , CIR-O 등이 있다.



docker 사용방식

containerd 사용방식

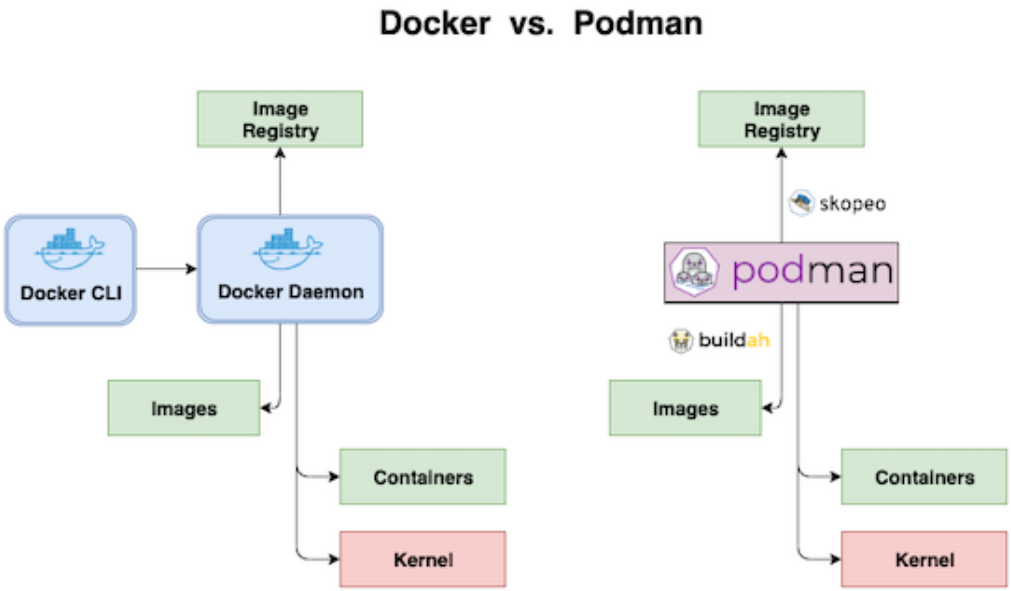


containerd

Docker 에서 OCI spec 을 준수하여 개발한 container runtime.
Docker 를 사용하면 containerd 가 기본적으로 runtime 으로 사용된다.
docker build 로 생성되는 도커 이미지는 OCI image spec을 준수하므로 별도의 작업없이 containerd 로 실행할수 있다.

CRI-O *https://cri-o.io/*

Redhat,IBM,Intel,SUSE 등이 OCI spec을 준수하여 개발한 쿠버네티스 전용 Container Runtime.
컨테이너 생성 및 이미지 빌드등을 할수 없는 컨테이너 실행을 목적으로 경량화된 런타임.



podman

- docker 를 대체할수 있는것으로 컨테이너를 실행하고 관리할수 있다.

buildah

- docker build 를 대체할수 있는것으로 이미지를 빌드한다. Docker 파일에 종속되지 않고 스크립트 언어를 사용해서 컨테이너 이미지를 빌드할 수 있다. Dockerfile 을 사용해서 빌드하는것도 가능하다.

skopeo

- 레지스트리에서 이미지를 검사하고, 이미지와 이미지 레이어를 가져오고 서명을 사용하여 이미지를 만들고 확인할수 있다.

* docker 가진 여러 기능을 각각의 툴로 나누었다.

쿠버네티스를 간단히 말하자면 Linux 컨테이너 작업을 자동화해주는 오픈소스 플랫폼입니다.
각 컨테이너별 자원 제한, 문제발생시 자동시작 등 컨테이너를 배포/확장,제어,자동화 하기위한 다양한 기능을 지원하는 컨테이너 오케스트레이션 도구

쿠버네티스 기본 용어

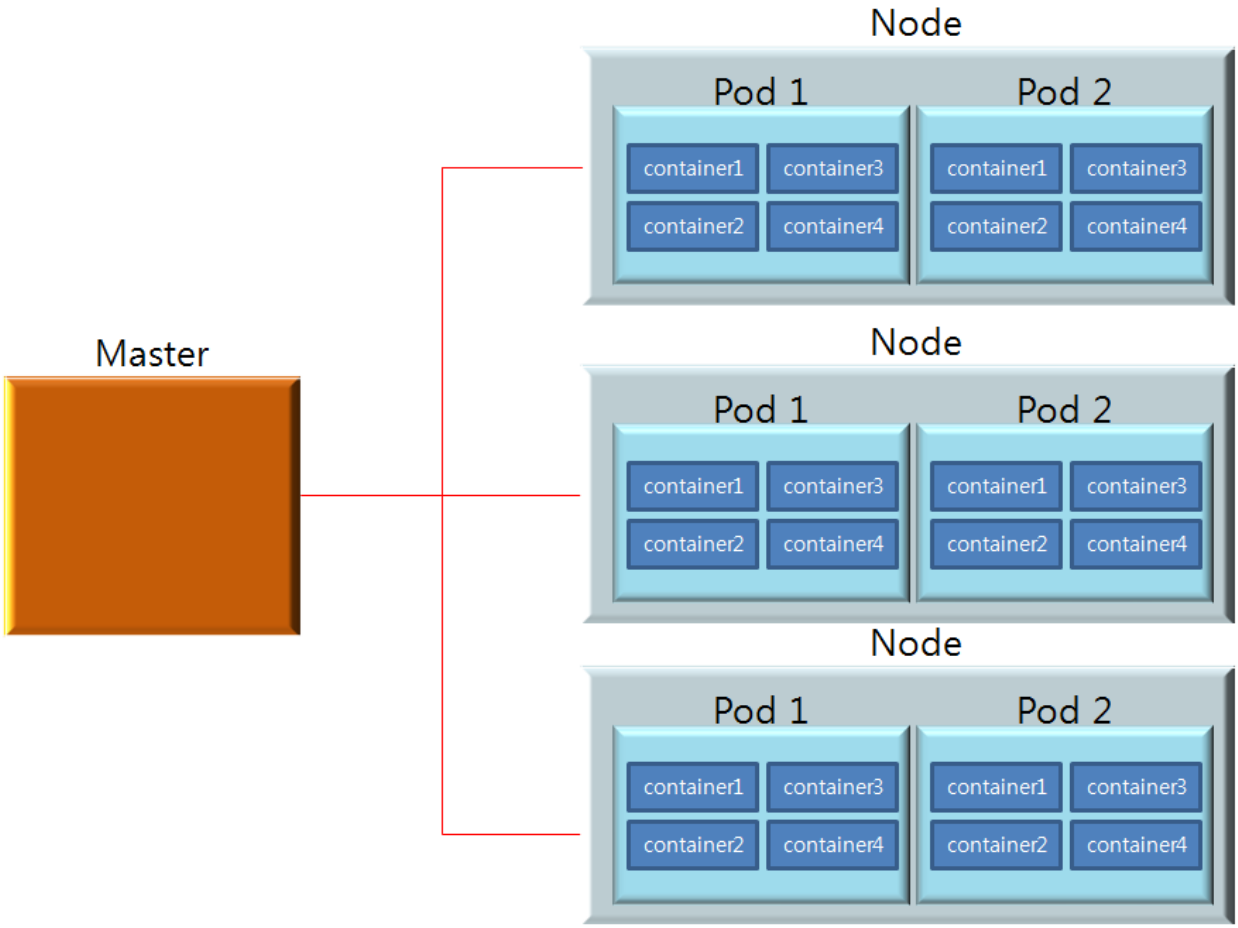
마스터(Master) : 노드를 제어하고 전체 클러스터를 관리해주는 컨트롤러 이며, 전체적인 제어/관리를 하기 위한 관리 하기 위한 서버

노드(nod) : 컨테이너가 배포될 물리 서버 또는 가상 머신 이며 워커 노드(Worker Node) 라고도 부른다.

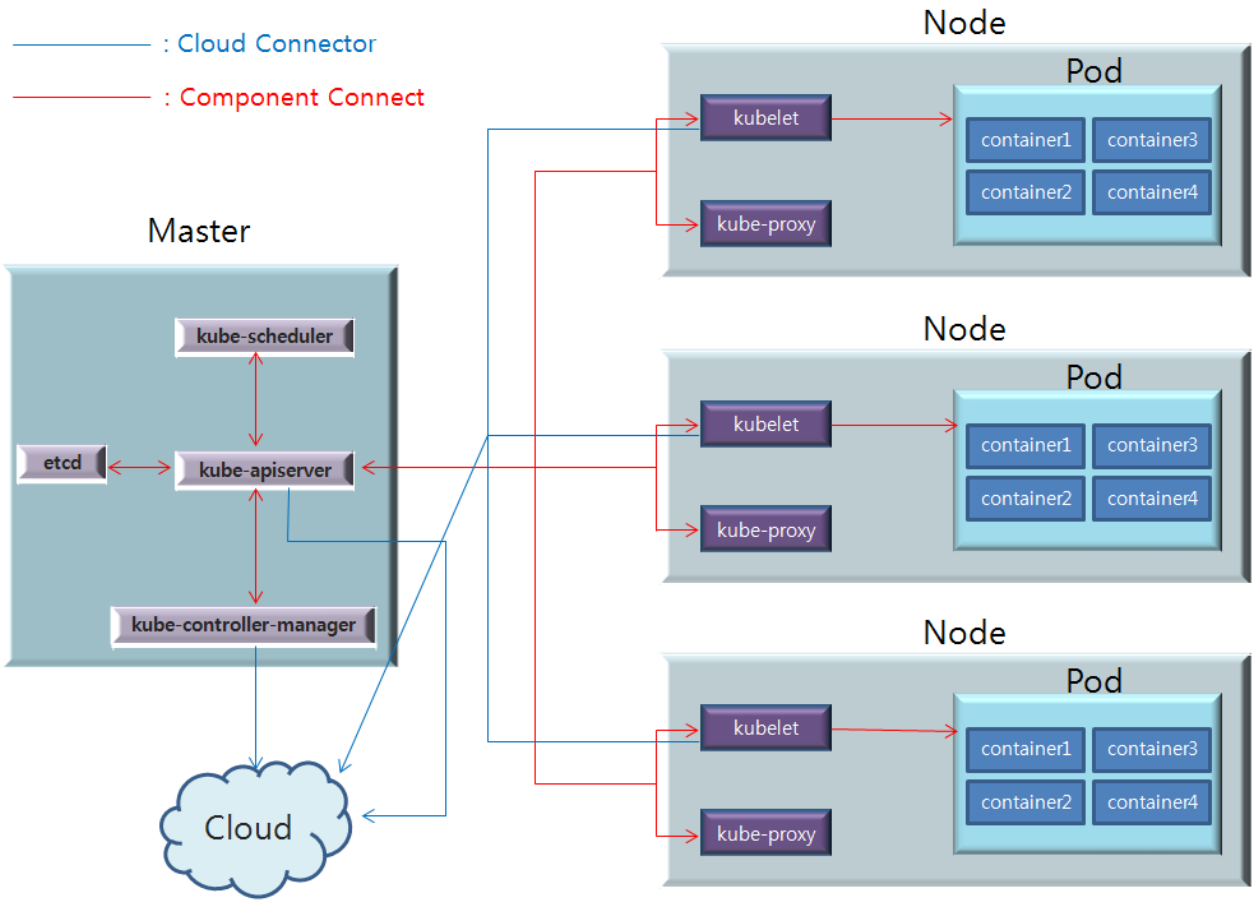
파드(pod) : 단일 노드에 배포된 하나 이상의 컨테이너 그룹이며, 파드라는 단위로 여러개의 컨테이너를 묶어서 파드 단위로 관리 할 수 있게 해준다.

https://kubernetes.io/ko/docs/concepts/workloads/pods/

대략적인 구조는 아래와 같다. (정확히는 Pod 를 묶어서 관리하는게 Docker 같은 툴 입니다.)



master 서버의 경우 고가용성 유지를 위해 여러개로 구성할 수 있으며 실제 관리는 리더 마스터 서버가 하되 나머지는 후보 마스터 서버로 유지 한다. 만약 리더 마스터 서버가 장애 발생하면 후보 마스터 서버 중 하나가 리더 마스터 서버 역할을 맡아 서비스상의 이슈가 없도록 한다.



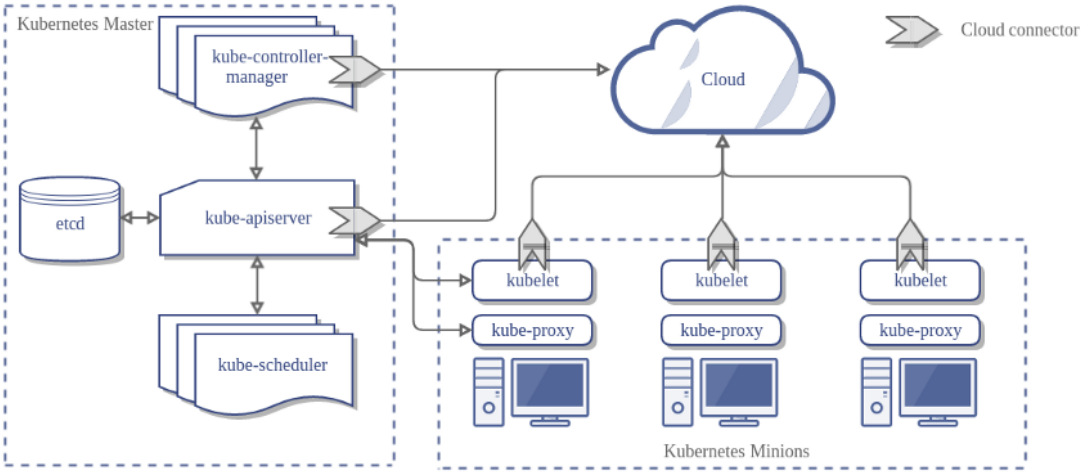
Kubernetes Architecture

Master

위 그림은 쿠버네티스 클러스터의 아키텍처이다. 좌측의 마스터 컴포넌트는 클러스터의 제어영역(control plane) 을 제공하여, 클러스터에 관한 전반적인 결정을 수행하고 클러스터 이벤트를 감지하고 반응한다. 마스터 컴포넌트는 클러스터 내에 어떤 노드에서든 동작할 수 있지만, 일반적으로 클러스터와 동일한 노드 상에서 구동시킨다. 아래는 마스터 내에서 동작하는 바이너리 컴포넌트들이며 쿠버네티스 초기화시 자동 설치된다.

kube-apiserver

쿠버네티스 API 로, 외부/내부에서 관리자의 원격 명령을 받을 수 있는 컴포넌트이다.



Kubernetes 제어 영역의 프런트엔드 역할을 한다.

etcd

모든 클러스터 데이터를 저장하는 고가용성 키-값 저장소로, etcd 데이터에 대한 백업 계획은 필수이다.

kube-scheduler

생성된 파드를 노드에 할당해 주는 컴포넌트. 이것을 스케줄링이라 하며, 리소스/하드웨어/소프트웨어/정책/워크로드 등을 모두 참고하여 가장 최적화된 노드에 파드를 배치하게 된다.

kube-controller-manager

컨트롤러 프로세스를 실행하고 클러스터의 실제 상태를 원하는 사양으로 조정한다.

아래의 컨트롤러들을 구동하는 컴포넌트이다.

- Node Controller : 노드가 다운되었을 때 알림와 대응에 관한 역할을 한다.
- Replication Controller : 지정된 수의 파드들을 유지시켜 주는 역할을 한다.
- Endpoints Controller: 서비스와 파드를 연결시켜 엔드포인트 오브젝트를 만든다.
- Service Account & Token Controllers: 새로운 네임스페이스에 대한 기본 계정과 API 접근 토큰을 생성한다.

Kubelet

컨테이너 생성 및 관리를위한 기본 프로그램 인 Docker 엔진과 상호작용하여 컨테이너가 포드에서 실행되도록 한다
<-- 컨테이너런타임이 도커인 경우에 해당
제공된 PodSpec 세트를 가져와 해당 컨테이너가 완전히 작동하는지 확인한다.

Kube-proxy

네트워크 연결을 관리하고 노드간에 네트워크 규칙을 유지.
주어진 클러스터의 모든 노드에서 Kubernetes 서비스 개념을 구현

쿠버네티스에서 사용하는 개념은 크게 객체(Object)와 그걸 관리하는 컨트롤러(Controller)가 있다.

객체는 사용자가 쿠버네티스에 바라는 상태(desired state)를 의미한다.
컨트롤러는 객체가 원래 설정된 상태를 잘 유지할수있게 관리하는 역할
객체에는 포드(pod), 서비스(service), 볼륨(volume), 네임스페이스(namespace)등이 있다
=> $ kubectl api-resource 로 확인
컨트롤러에는 ReplicaSet, Deployment, StatefulSet, DaemonSet, Job 등이 있다.

쿠버네티스 클러스터에 객체나 컨트롤러가 어떤 상태여야 하는지를 제출할때는 yaml 파일형식의 템플릿을 사용

기본 형식
---
apiVersion: v1 <= Kind 의 종류에 따라서 버전이 다를수 있다.
Kind: Pod <= 어떤종류의 객체인지 명시
metadata: 해당 쿠버네티스 객체를 유니크하게 식별할수 있는 데이터이름, uid, 네임스페이스를 포함한다.
spec: 해당 쿠버네티스 객체의 의도

쿠버네티스는 현재 시스템을 사용자가 정의한 상태, 즉 사용자가 원하는 상태(어떤 Pod 가 몇 개가 떠있고, 어떤 Service 가 어떤 포트로 열려있고 등)로 맞춰준다. 그럴려면 오브젝트의 현재 상태를 지속적으로 체크하고 상태를 제어해야 한다.

컨트롤러 매니저(Controller Manager)에는 Replication, DaemonSet, Job, Service 등 다양한 오브젝트를 제어하는 컨트롤러가 존재한다.

스케쥴러(Scheduler)는 노드의 정보와 알고리즘을 통해 특정 Pod 을 어떤 노드에 배포할 지 결정하고 대상 노드들을 조건에 따라 걸러내고 남은 노드는 우선 순위(점수)를 매겨서 가장 최적의 노드를 선택

위의 모듈은 Control Plane 인 Master 노드에 존재하지만, Kubelet 과 Kube-proxy 는 Worker 노드에 존재한다.
Kubelet 은 API 서버와 통신하며 Worker 노드의 작업을 제어하는 에이전트
Kube-proxy 는 Pod 에 접근하기 위한 iptables 를 설정한다.
iptables 는 리눅스 커널의 패킷 필터링 기능을 관리하는 도구

쿠버네티스 디플로이먼트

일단 쿠버네티스 클러스터를 구동시키면, 그 위에 컨테이너화된 애플리케이션을 배포할 수 있다.그러기 위해서, 쿠버네티스 디플로이먼트 설정을 만들어야 한다.

디플로이먼트는 쿠버네티스가 애플리케이션의 인스턴스를 어떻게 생성하고 업데이트해야 하는지를 지시한다. 디플로이먼트가 만들어지면, 쿠버네티스 마스터가 해당 디플로이먼트에 포함된 애플리케이션 인스턴스가 클러스터의 개별 노드에서 실행되도록 스케줄한다.

애플리케이션 인스턴스가 생성되면, 쿠버네티스 디플로이먼트 컨트롤러는 지속적으로 이들 인스턴스를 모니터링한다. 인스턴스를 구동 중인 노드가 다운되거나 삭제되면, 디플로이먼트 컨트롤러가 인스턴스를 클러스터 내부의 다른 노드의 인스턴스로 교체시켜준다.이렇게 머신의 장애나 정비에 대응할 수 있는 자동 복구**(self-healing)** 메커니즘을 제공한다**.**

오케스트레이션 기능이 없던 환경에서는, 설치 스크립트가 애플리케이션을 시작하는데 종종 사용되곤 했지만, 머신의 장애가 발생한 경우 복구를 해주지는 않았다.
쿠버네티스 디플로이먼트는 애플리케이션 인스턴스를 생성해주고 여러 노드에 걸쳐서 지속적으로 인스턴스가 구동되도록 하는 두 가지를 모두 하기 때문에 애플리케이션 관리를 위한 접근법에서 근본적인 차이를 가져다준다.

Kubectl 이라는 쿠버네티스 CLI 를 통해 디플로이먼트를 생성하고 관리할 수 있다.
Kubectl 은 클러스터와 상호 작용하기 위해 쿠버네티스 API 를 사용한다.

kubectl command 형식

$ kubectl [command] [type] [name] [flags]
[command]
- create
- get
- describe
- delete
- run : 클러스트에 특정 이미지를 가지고 pod 를 생성하는 명령어.
[type]
-pod
-service
[name]
resource 이름
[flag] => 옵션

$ kubectl run httpd --image=httpd:2.4 --port=80

쿠버네티스 네임스페이스
단일 클러스트 내에서 리소스 그룹의 격리 매커니즘 제공.
리소스의 이름은 같은 네임스페이스에서 중복될수 없지만 네임스페이가 다르면 관계없다.
스토리지 클래스 같은 일부 리소스는 네임스페이스의 영향을 받지 않는다.
kube- 로 시작하는 네임스페이스는 시스템이 사용하는것으로 예약되어 있다.

namespace 관련명령어
```
$ kubectl get namespace
NAME                   STATUS    AGE
default                Active    1d
kube-node-lease        Active    1d
kube-public            Active    1d
kube-system            Active    1d
```

쿠버네티스는 처음에 세 개의 초기 네임스페이스를 갖는다.
default 다른 네임스페이스가 없는 오브젝트를 위한 기본 네임스페이스
kube-system 쿠버네티스 시스템에서 생성한 오브젝트를 위한 네임스페이스
kube-public 이 네임스페이스는 자동으로 생성되며 모든 사용자(인증되지 않은 사용자 포함)가 읽기 권한으로 접근할 수 있다. 이 네임스페이스는 주로 전체 클러스터 중에 공개적으로 드러나서 읽을 수 있는 리소스를 위해 예약되어 있다.

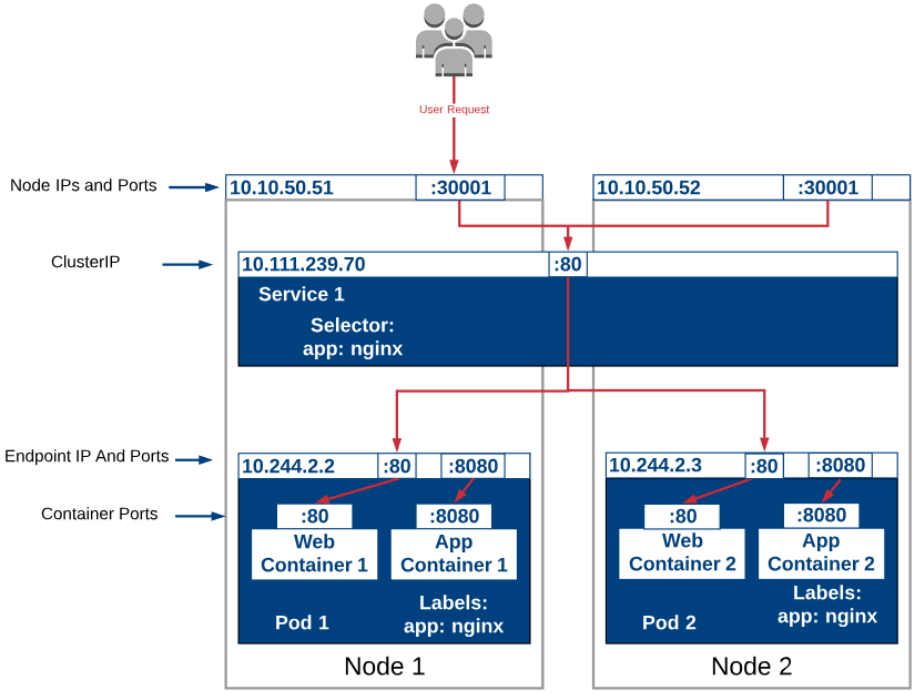
namespace 생성
```
$ kubectl create namespace[or ns] namespace 이름
$ kubectl run nginx --image=nginx --namespace=<insert-namespace-name-here>
$ kubectl get pods --namespace=<insert-namespace-name-here>
```

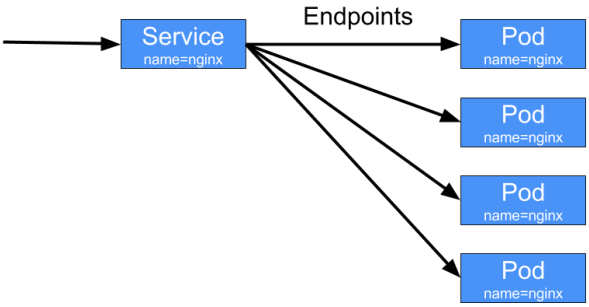
선호하는 네임스페이스 설정하기
이후 모든 kubectl 명령에서 사용하는 네임스페이스를 컨텍스트에 영구적으로 저장할 수 있다.
```
kubectl config set-context --current --namespace=<insert-namespace-name-here>
# 확인하기 => $ kubectl config view
```

아래처럼 yaml 파일로 생성할수도 있다.
```
$ cat ns.yaml
---------------------------------
apiVersion: v1
kind: Namespace
metadata:
  name: testns
[root@minik test]# kubectl apply -f ns.yaml
```

기타 kubectl 명령어 참고 - 아래 링크.
https://kubernetes.io/ko/docs/reference/kubectl/cheatsheet/

쿠버네티스 ip address 및 port





kubernetes 관리를 위한 계정 생성

*. vagrant image 를 사용한다면 일반계정을 추가로 만들 필요없이 vagrant 계정으로 아래의 설정을 하면 된다.

- master node 에서 아래와 같이 admin 계정(계정 id 는 달라도 상관없다)생성
```
# useradd admin
# passwd admin
# echo "admin  ALL=(ALL)  NOPASSWD: ALL" > /etc/sudoers.d/admin
```

계정 설정 완료후 admin 계정으로 접속
```
[admin@master ~]$ mkdir -p $HOME/.kube
[admin@master ~]$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
[admin@master ~]$ sudo chown admin:admin /home/admin/.kube/config
[admin@master ~]$ echo "source <(kubectl completion bash)" >> ~/.bashrc
[admin@master ~]$ source .bashrc
```

kubernetes 주요 object

examples.

1. ns.yaml
```
---
apiVersion: v1
kind: Namespace
metadata:
  name: testns
```

2. apache.yaml
```
$ cat apache.yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: apache-pod
  labels:
    app: myweb
spec:
    containers:
    - name: myweb-container
      image: httpd:2.4
      ports:
        - containerPort: 80
```

```
$ kubectl create -f apache.yaml
$ kubectl get pods

NAME          READY   STATUS    RESTARTS   AGE
apache-pod    1/1     Running   0          9m16s
```

3. **service**

```
$ cat myweb-service.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: myweb-service
spec:
  ports:
  - port: 8001
    targetPort: 80
  selector:
    app: myweb


$ kubectl create -f myweb-service.yaml
$ kubectl get svc
NAME             TYPE          CLUSTER-IP       EXTERNAL-IP    PORT(S)     AGE
myweb-service    ClusterIP    10.109.82.232    <none>          8001/TCP    4m19s

$ curl http://10.109.82.232:8001
<html><body><h1>It works!</h1></body></html>
$
```

---

4. **ReplicaSet** https://kubernetes.io/ko/docs/concepts/workloads/controllers/replicaset/

- pod 의 실행을 항상 동일한 개수로 안정적으로 유지.

```
$ cat replica.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: apache-replica
spec:
  replicas: 3
  selector:
    matchLabels:
      app: apache-replica-test
  template:
    metadata:
      labels:
        app: apache-replica-test
    spec:
      containers:
      - name: myweb-container2
        image: httpd:2.4
        ports:
        - containerPort: 80

$ kubectl create -f replica.yaml
replicaset.apps/apache-replica created

$ kubectl get pods
NAME                      READY    STATUS      RESTARTS    AGE
apache-pod                1/1      Running       0          50m
apache-replica-22m9n      1/1      Running       0          4s
apache-replica-cpjb5      1/1      Running       0          4s
apache-replica-tsqk9      1/1      Running       0          4s

$ kubectl delete pods apache-replica-22m9n
pod "apache-replica-22m9n" deleted
```

$ kubectl get pods ; pod 를 삭제하더라도 항상 replica 수만큼 유지한다.

```
NAME                    READY   STATUS    RESTARTS   AGE
apache-pod              1/1     Running   0          51m
apache-replica-cpjb5    1/1     Running   0          46s
apache-replica-tsqk9    1/1     Running   0          46s
apache-replica-xznnm    1/1     Running   0          24s   <-- pod 가 새로 생성됨
```

* command :

```
$ kubectl scale --replicas=5 replicaset apache-replica
replicaset.apps/apache-replica scaled
```

```
$ kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
apache-pod              1/1     Running   0          54m
apache-replica-26xc2    1/1     Running   0          7s
apache-replica-2jjhf    1/1     Running   0          7s
apache-replica-cpjb5    1/1     Running   0          3m42s
apache-replica-tsqk9    1/1     Running   0          3m42s
apache-replica-xznnm    1/1     Running   0          3m20s
```

ReplicaSet object 로 배포한 pod 는 아래처럼 replas=0 으로 설정하거나

```
$ kubectl scale --replicas=0 replicaset apache-replica
replicaset.apps/apache-replica scaled
```

또는

```
$ kubectl delete replicaset apache-replica (삭제하기전 확인은 $kubectl get replicaset(또는 replicasets.apps)
replicaset.apps "apache-replica" deleted
```

---

5. deployment

- 컨테이너 어플리케이션을 배포 및 관리하는 역할
- deployment object 가 ReplicaSet object 를 대신 할 수 있으며 rolling update / rollback 을 지원하므로 application 배포시 주로 ReplicaSet object 보다 deployment 를 주로 사용

update / rollback command

```
$ kubectl set image deployment nginx-test nginx=nginx:1.19 --record[=true/false]
```

nginx-test : deployment 이름

nginx=nginx:1.19 : container 이름=도커이미지

```
$ kubectl rollout history deployment nginx-test  ; rollout history 출력
$ kubectl rollout undo deployment --to-revision=2  ; rollback
```

*. nginx version 은 $docker search nginx 로 확인할수 있다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx   -> Deployment 이름
  labels:
    app: nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx   -> template 의 label 과 key: value 가 같아야 한다.
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.14
        name: nginx
        ports:
        - containerPort: 80   -> container port 는 마음대로 정할수 없다.
```

```
kubectl get deployments.apps
NAME     READY    UP-TO-DATE    AVAILABLE    AGE
nginx    10/10    10            10           17s
```

* UP-TO-DATE : 의도한 상태를 얻기 위해 업데이트 된 레플리카의 수를 표시한다

```
[vagrant@master new_work]$ kubectl get replicasets.apps
NAME               DESIRED    CURRENT    READY    AGE
nginx-9bd4cd86c    10         10         10       27s
[vagrant@master new_work]$
```

$ kubectl set image deployment/nginx nginx=nginx:1.15 --record=true(--record=true 는 옵션)
* 옵션을 생략하거나 --record=false 이면 rollout history 에 명령어가 기록되지 않는다.

```
[vagrant@master new_work]$ kubectl get pods
  NAME                          READY        STATUS              RESTARTS        AGE
  nginx-756b7dcbfb-4xlks        0/1          ContainerCreating   0               2s
  nginx-756b7dcbfb-5tlwq        0/1          ContainerCreating   0               2s
  nginx-756b7dcbfb-7t4ck        1/1          Running             0               4s
  nginx-756b7dcbfb-d8kjt        1/1          Running             0               7s
  nginx-756b7dcbfb-n4jjw        1/1          Running             0               7s
  nginx-756b7dcbfb-ns27c        1/1          Running             0               4s
  nginx-756b7dcbfb-w8zv6        1/1          Running             0               7s
  nginx-756b7dcbfb-wptsh        0/1          ContainerCreating   0               7s
  nginx-756b7dcbfb-zmc98        1/1          Running             0               4s
  nginx-9bd4cd86c-j59fk         1/1          Running             0               7s
  nginx-9bd4cd86c-kr89m         1/1          Terminating         0               7s
  nginx-9bd4cd86c-zdnjh         1/1          Terminating         0               4s
  nginx-756b7dcbfb-2md7l        1/1          Running             0               7s
```

* 롤링 업데이트중에서 컨테이너 실행을 보장하는 갯수와 컨테이너 최대 실행 갯수가 maxUnavailable 과 maxsurge 값으로 결정된다.

```
[vagrant@master new_work]$ kubectl get pods
  NAME                                READY          STATUS             RESTARTS         AGE
  nginx-756b7dcbfb-4xlks              1/1            Running            0                2m39s
  nginx-756b7dcbfb-5tlwq              1/1            Running            0                2m39s
  nginx-756b7dcbfb-7t4ck              1/1            Running            0                2m39s
  nginx-756b7dcbfb-d8kjt              1/1            Running            0                2m39s
  nginx-756b7dcbfb-n4jjw              1/1            Running            0                2m39s
  nginx-756b7dcbfb-ns27c              1/1            Running            0                2m39s
  nginx-756b7dcbfb-w8zv6              1/1            Running            0                2m39s
  nginx-756b7dcbfb-wptsh              1/1            Running            0                2m39s
  nginx-756b7dcbfb-zmc98              1/1            Running            0                2m39s
  nginx-756b7dcbfb-2md7               1/1            Running            0                2m39s
```

* 업데이트가 완료되면 컨테이너 갯수가 정확히 replicas 설정값으로 실행된다.

```
[vagrant@master new_work]$
 kubectl get deployments.apps
NAME     READY    UP-TO-DATE    AVAILABLE    AGE
nginx    10/10      10              10           8m56s

[vagrant@master new_work]$ kubectl get replicasets.apps
NAME                 DESIRED    CURRENT    READY    AGE
nginx-756b7dcbfb     10         10         10       3m21s
nginx-9bd4cd86c      0          0          0        9m15s  <= update 하기전의 replicaset.
[vagrant@master new_work]$

[vagrant@master new_work]$ kubectl rollout history deployment
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1         <none>
2         kubectl set image deployment/nginx nginx=nginx:1.15 --record=true


[vagrant@master new_work]$ kubectl rollout history deployment
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1         <none>
2         kubectl set image deployment/nginx nginx=nginx:1.15 --record=true
```

kubectl rollout undo deployment nginx --record --to-revision=1 <= roll back
record-to-revision 옵션을 생략하면 가장 최근실행된 것으로 롤백한다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 10
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1    <- rolling update 동안 동작하지 않아도 되는 pod 의 갯수
      maxSurge: 10         < - rolling update 동안 추가로 실행되어 될 파드의 개수
```

```
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.14
        name: nginx
        ports:
        - containerPort: 80
```

* maxUnavaliable / maxSurge 의 default 값은 각각 25%
maxUnavailable 의 값은 반올림이 아니다. 그 반대.
maxSurge 의 값은 반올림.
* 예를들면 실행중인 pod 가 10 개라면 maxUnavailable 은 2, maxsurge 는 3

---

6. 서비스 타입

a. clusterIP 타입

```
$ cat cluster-ip.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-clusterip
spec:
  ports:
    - name: nginx-port
      port: 8000
      targetPort: 80
  selector:
    app: nginx
  type: ClusterIP

$ kubectl create -f cluster-ip.yaml
service/nginx-clusterip created

$ kubectl get svc
NAME              TYPE          CLUSTER-IP       EXTERNAL-IP    PORT(S)     AGE
myweb-service     ClusterIP     10.109.82.232     <none>        8001/TCP    102m
nginx-clusterip   ClusterIP     10.110.121.231    <none>        8000/TCP    9s

$ kubectl describe svc nginx-clusterip
Name:                nginx-clusterip
Namespace:           myns
Labels:              <none>
Annotations:         <none>
Selector:            app=nginx
Type:                ClusterIP
IP:                  10.110.121.231
Port:                nginx-port  8000/TCP
TargetPort:          80/TCP
Endpoints:           192.168.206.23:80
Session Affinity:    None
Events:              <none>
$

$ curl http://10.110.121.231  ; 접속안됨
$ curl http://10.110.121.231:8000  ; 접속됨
```

b. NodePort 타입 서비스

```
[admin@master ~]$ cat node-port.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-nodeport
spec:
  ports:
    - name: nginx-port
      port: 8000
      targetPort: 80
  selector:
    app: nginx
  type: NodePort

[admin@master ~]$

$ kubectl create -f nodeport.yaml
service/nginx-nodeport created
$ kubectl get svc
NAME              TYPE          CLUSTER-IP       EXTERNAL-IP    PORT(S)           AGE
myweb-service     ClusterIP     10.109.82.232     <none>        8001/TCP          56m
nginx-nodeport    NodePort      10.104.4.186      <none>        8000:32245/TCP    6s

$ curl 10.104.4.186
curl: (7) Failed connect to 10.104.4.186:80; 연결이 거부됨
curl 10.104.4.186:8000  ; 접속됨
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
..... 출력내용 ...

$ curl 10.104.4.186:32245
curl: (7) Failed connect to 10.104.4.186:32245; 연결이 거부됨
$ curl localhost:32245  ; 접속됨
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
.... 출력내용 ...

------------------------------------------------------------
*. deployment 를 삭제하면 replicaset 및 pod 가 전부 삭제된다.

$ kubectl get deployments.apps
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
nginx-test    3/3     3            3           8m24s

$ kubectl delete deployments.apps nginx-test
deployment.apps "nginx-test" deleted

$ kubectl get deployments.apps
No resources found in myns namespace.

$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
apache-pod    1/1     Running   0          68m
```

$ kubectl get svc ; service 는 삭제되지 않고 남아 있다.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
myweb-service	ClusterIP	10.109.82.232	<none>	8001/TCP	61m
nginx-nodeport	NodePort	10.104.4.186	<none>	8000:32245/TCP	5m14s

$ kubectl delete svc nginx-nodeport ; 서비스 삭제
service "nginx-nodeport" deleted

* deployment 를 생성하지 않은 경우에는 replicaset 을 삭제하면 pod 가 전부 삭제된다.
ex)
[admin@master ~]$ kubectl delete replicasets.apps apache2-replica
[admin@master ~]$ kubectl get pods

NAME	READY	STATUS	RESTARTS	AGE
apache-sample	1/1	Running	0	7h7m
apache2-replica-lvvnf	0/1	Terminating	0	148m
apache2-replica-s44vq	0/1	Terminating	0	148m
apache2-replica-ts4tf	0/1	Terminating	0	148m
apache2-replica-tww56	0/1	Terminating	0	148m
apache2-replica-vgbzk	0/1	Terminating	0	5h5m

c. LoadBalancer type
aws 나 gcp 같은 public cluoud 에서는 기본적으로 쿠버네티스에서 loadbalancer 를 지원하지만
로컬에 설치된 kubernetes 는 오픈소스 프로젝트인 metal lb 를 설치해야 쿠버네티스에서 loadbalancer type 을 사용할 수 있다.

metal lb 사이트 https://metallb.universe.tf/installation/
---

* 한개의 Pod 에 멀티 컨테이너를 사용하게 되면
- 컨테이너는 같은 ip 를 사용하게 된다.
- 스토리지 볼륨을 공유할수 있다.

ex)
```
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
  - name: first
    image: httpd:2.4
  - name: second
    image: alpine
    command: ["/bin/sleep","3600s"]
```

```
$ kubectl get pods -o wide
NAME   READY   STATUS    RESTARTS   AGE   IP           NODE              NOMINATED NODE   READINESS GATES
test   2/2     Running   0          75s   10.244.1.2   w1.example.com    <none>           <none>
```

```
$ kubectl exec -it test -- /bin/bash  ; 첫번째 컨테이너로 접속
Defaulted container "first" out of: first, second
root@test:/usr/local/apache2# exit
```

```
$ kubectl exec -it test -c first -- /bin/bash ; 첫번째 컨테이너로 접속
```

```
$ kubectl exec -it test -c second -- /bin/sh ; 두번째 컨테이너로 접속
```

기타 kubernetes 실행 명령어

```
$ kubectl run mysql --image mysql
$ kubectl get pods
mysql          0/1     CrashLoopBackOff   3 (34s ago)
```
* mysql docker image 는 실행할때 MYSQL_ROOT_PASSWORD 의 환경변수가 필요하므로 아래와 같이 실행을 해야한다

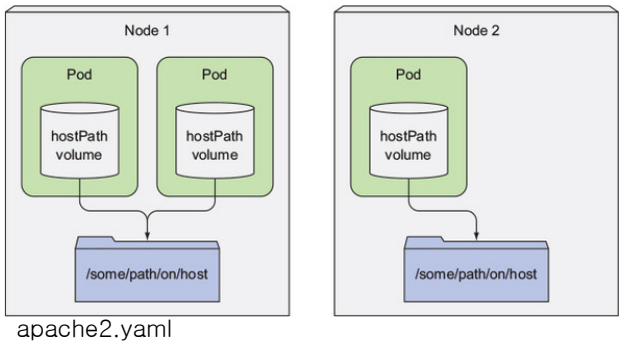
```
$ kubectl run mysql2 --env MYSQL_ROOT_PASSWORD=mypass --image=mysql
```

파일은 아래처럼 작성.
```
apiVersion: v1
kind: Pod
metadata:
  name: mysql-pod
spec:
  containers:
  - name: mysql
    image: mysql:latest
    env:
      MYSQL_ROOT_PASSWORD=mypass
```

8. 쿠버네티스 영구볼륨 설정 https://kubernetes.io/ko/docs/concepts/storage/volumes/

컨테이너가 삭제되면 컨테이너 내의 데이터도 삭제된다.
데이터베이스 처럼 일반적으로 컨테이너가 삭제되더라도 데이터를 영구적으로 보존해야 하는 경우가 있을 수 있다.
이런 경우 영구볼륨설정이 필요하다.

hostPath
- hostPath 볼륨은 호스트 노드의 파일시스템에 있는 파일이나 디렉토리를 직접 마운트한다.
- 파드가 삭제되어도 볼륨의 데이터는 삭제되지 않는다.
- 기존의 파드가 삭제되고 새롭게 파드가 스케줄링 될때 만약 기존의 노드에 파드가 스케줄링 되지 않으면
  기존의 노드 데이터는 사용 할 수 없으므로 주의해야 한다.
- 쿠버네티스 공식사이트 문서에는 보안상 hostPath 를 사용하지 않는것을 권장한다. hostPath 볼륨을 사용해야 하는경우, 필요한 파일 또는 디렉토리로만 범위를 지정하고 ReadOnly 로 마운트하는것이 좋다.



apache2.yaml

```
---
apiVersion: v1
kind: Pod
metadata:
  name: myapache-new
  labels:
    app: myweb-svc
spec:
    containers:
    - name: myapache-container
      image: httpd:2.4
      ports:
        - containerPort: 80
      volumeMounts:
      - name: hostpath-volume
        mountPath: /usr/local/apache2/htdocs
    volumes:
    - name: hostpath-volume
      # hostPath: pod 가 실행중인 node 에 디렉토리가 자동으로 생성이 되고
      # 컨테이너내의 mountPath 디렉토리가 바인딩된다
      hostPath:
        path: /var/tmp/web_docs
```

---

emptyDir https://kubernetes.io/ko/docs/tasks/configure-pod-container/configure-volume-storage/
- 컨테이너 파일시스템은 컨테이너가 실행되고 있는 동안만 존재한다.
  컨테이너가 종료되면 스토리지 볼륨은 삭제된다.
  pod 내의 컨테이너가 크래시 되어서 삭제되거나 재시작 되더라도 emptyDir 의 라이프사이클은 pod 단위이기때문에
  emptyDir 은 삭제되지 않고 계속해서 사용가능하다.
  emptyDir 은 디스크 대신에 메모리를 사용하는것도 가능하다.

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

```
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
```

*. 메모리에 저장하는 경우에는 medium 을 아래처럼 메모리로 설정하면 된다.

```
   volumes:
   - name: test-storage
     emptyDir:
       medium: Memroy
       sizeLimit: 1Gi    # sizeLimit 를 지정하지 않으면 가용 메모리를 전부 사용할수 있다.
```

```
[vagrant@master work]$ kubectl exec -it redis -- /bin/bash
root@redis:/data# mount |grep /data/redis
/dev/sda1 on /data/redis type xfs (rw,relatime,seclabel,attr2,inode64,noquota)
root@redis:/data#
root@redis:/data# echo hello redis > /data/redis/testfile
$ kubectl get pods -o wide |grep redis
redis                       1/1       Running   0          2m10s     10.32.0.3    node1.example.com

$ ssh node1 'sudo find / -name testfile 2> /dev/null'
/var/lib/kubelet/pods/a59f63b8-bda0-496d-a5a4-5fb9bd9decb6/volumes/kubernetes.io~empty-dir/redis-storage/testf
ile
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-storage-test
spec:
  containers:
    - name: nfs-container-test
      image: centos:7
      # container 실행시 즉시 종료되지 않도록 컨테이너 내에서 명령어 실행
      command: [ 'sh', '-c', '/usr/bin/sleep 3600s' ]
      volumeMounts:
      - name: nfs-volume
        # 컨테이너 내의 nfs 공유디렉토리의 마운트 포인터
        mountPath: /mnt
  volumes:
    - name: nfs-volume
      nfs:
        # nfs 공유디렉토리
        path: /var/nfs_storage
        # nfs server 주소
        server: 192.168.200.90
```

아래처럼 pv 와 pvc 객체를 사용하여 영구스토리지 볼륨을 관리하는게 일반적이다.
https://kubernetes.io/ko/docs/concepts/storage/persistent-volumes/

pv - 영구 스토리지 볼륨을 설정하기 위한 객체 (* namespace 의 영향을 받지 않는 cluster 범위의 오브젝트)
pvc - 영구 스토리지 볼륨 사용을 요청하기 위한 객체(* namespace 기반의 오브젝트)

라이프사이클

1. Provisioning
- 볼륨으로 사용하기 위한 물리적인 공간확보
- 프로비저닝은 디스크 공간을 확보하여 PV 를 생성하는 단계
- 정적 프로비저닝과 동적 프로비저닝이 있다.

2. Binding
- 프로비저닝으로 생성된 PV 와 PVC 를 연결하는 단계
- PVC 를 통하여 용량등 조건에 맞는 PV 연결
- PVC 한개가 여러개의 PV 에 바인딩 될수는 없다.

3. 사용
- PVC 는 파드에 설정되고 파드는 PVC 를 통해서 볼륨을 인식해서 사용
- 할당된 PVC 는 파드를 유지하는 동안 계속 사용하며 시스템에서 임의 삭제 할수 없다.

4. 반환(Reclaming)
- 사용이 끝난 PVC 가 삭제되고 PVC 를 사용하던 PV 를 다시 사용가능한 상태로 반환되고 초기화 되는 과정
- PV 는 연결된 PVC 가 삭제된 후 다시 다른 PVC 에 의해서 재사용이 가능한데, 재 사용시에 디스크의 내용을 삭제할지 유지할지에 대한 Relclaim 정책 설정 가능
- Reclaim Policy : Retain, Delete, Recycle
* reclaim policy - pvc 를 삭제했을때 스토리지 볼륨을 처리하는 방법에 대한 정책 (*디폴트 정책은 retain)

retain - pvc 사용이 끝난후에도 스토리지 볼륨의 데이터를 보존.
pvc 를 삭제하면 pv 상태가 released 가 되고 데이터는 보존됨.
pvc 실행을 하면 pv 가 바인딩 되지 않는다. pv 를 다시 사용하려면 수동으로 다시 시작해야 한다.

delete - pvc 사용이 끝난후에는 스토리지 볼륨 삭제
=> pvc 를 삭제하면 pv 가 삭제된다. 데이터는 보존됨
recycle(deprecated) - pvc 사용이 끝난후에 스토리지 볼륨 데이터 삭제후 스토리지 볼륨을 사용가능한 상태로 설정
=> pvc 를 삭제하면 pv 가 available 상태가 된다.

accessModes:
ReadWriteOnce -- 하나의 노드에서 볼륨을 읽기-쓰기로 마운트할 수 있다
ReadOnlyMany -- 여러 노드에서 볼륨을 읽기 전용으로 마운트할 수 있다
ReadWriteMany -- 여러 노드에서 볼륨을 읽기-쓰기로 마운트할 수 있다

CLI 에서 접근 모드는 다음과 같이 약어로 표시된다.
RWO - ReadWriteOnce
ROX - ReadOnlyMany
RWX - ReadWriteMany

```
apiVersion: v1
kind: PersistentVolume
metadata:
  # 아래 name 은 어떤 이름이라도 상관 없다.
  name: nfs-pv
  labels:
    volume: nfs-pv-volume
spec:
  capacity:
    # 스토리지 크기 결정
    storage: 5Gi
  accessModes:
    # ReadWriteMany - multi node 에서 읽고쓰기가 가능하다.
    - ReadWriteMany
  persistentVolumeReclaimPolicy:
    # retain even if pods terminate
    Retain
  nfs:
```

```
    # NFS server's definition
    # nfs 공유 디렉토리
    path: /var/nfs_storage
   # nfs 서버의 주소
    server: 192.168.200.90
   # 공유디렉토리에 대해서 읽고 쓰기 권한 부여
    readOnly: false

---------------------------------------------------------------
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  # pvc 이름은 어떤 이름이라도 상관없다.
  name: nfs-pvc
spec:
  selector:
    matchLabels:
    # nfs pv 의 라벨이름과 일치해야 한다.
      volume: nfs-pv-volume
  accessModes:
  - ReadWriteMany
  resources:
     requests:
       # storage size to use
       storage: 1Gi
-----------------------------------------------------------------
```

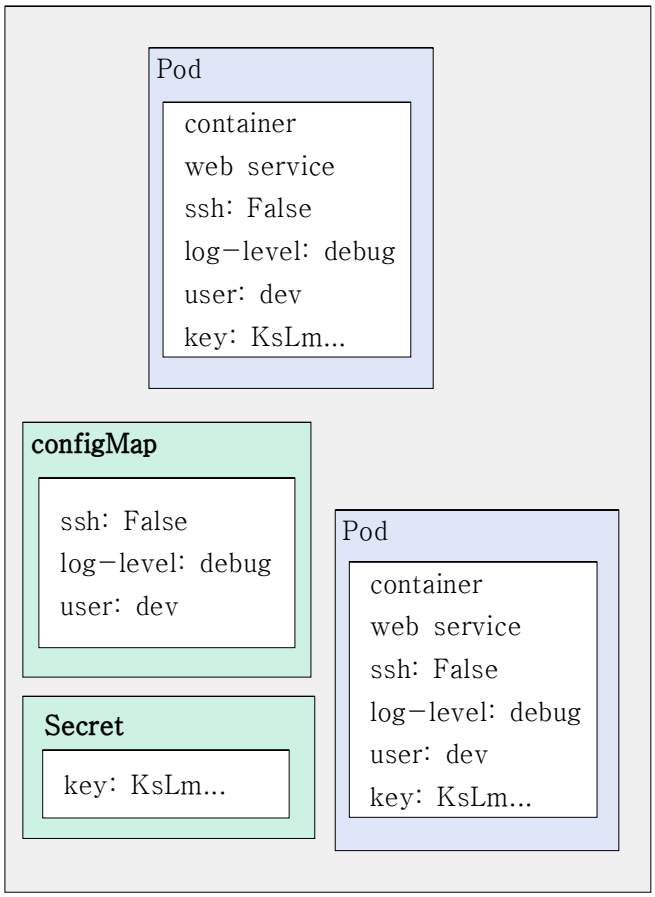
kubernetes wep application 배포 실습

secret / configmap

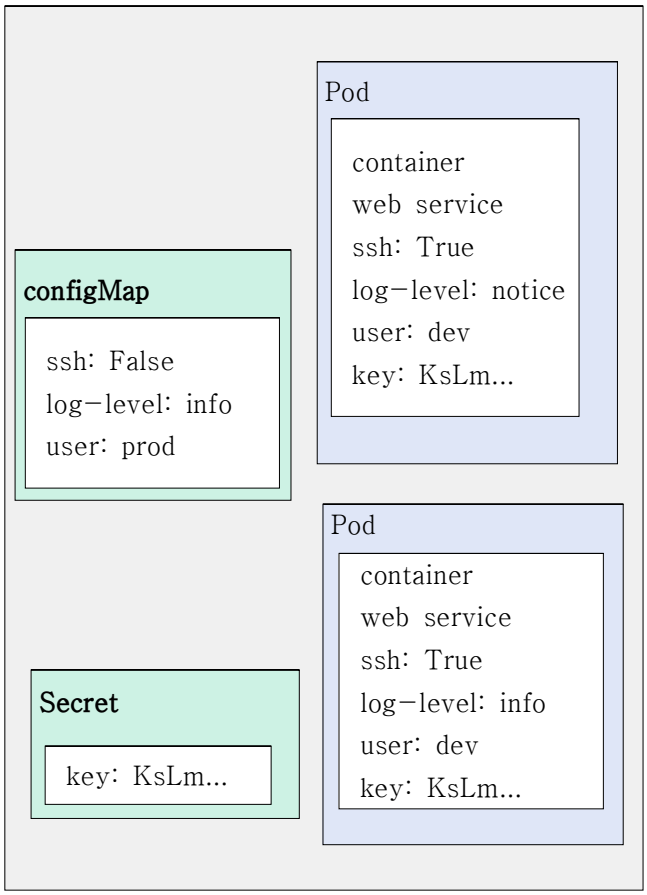
https://kubernetes.io/ko/docs/concepts/configuration/secret/
https://kubernetes.io/ko/docs/concepts/configuration/configmap/

- 환경에 따라 다르거나 자주 변경되는 설정 옵션을 configmap 오브젝트로 분리해서 관리할수 있다.
- secret 오브젝트는 configmap 과 비슷하지만 보안에 민감한 설정을 관리하기 위한것이다.

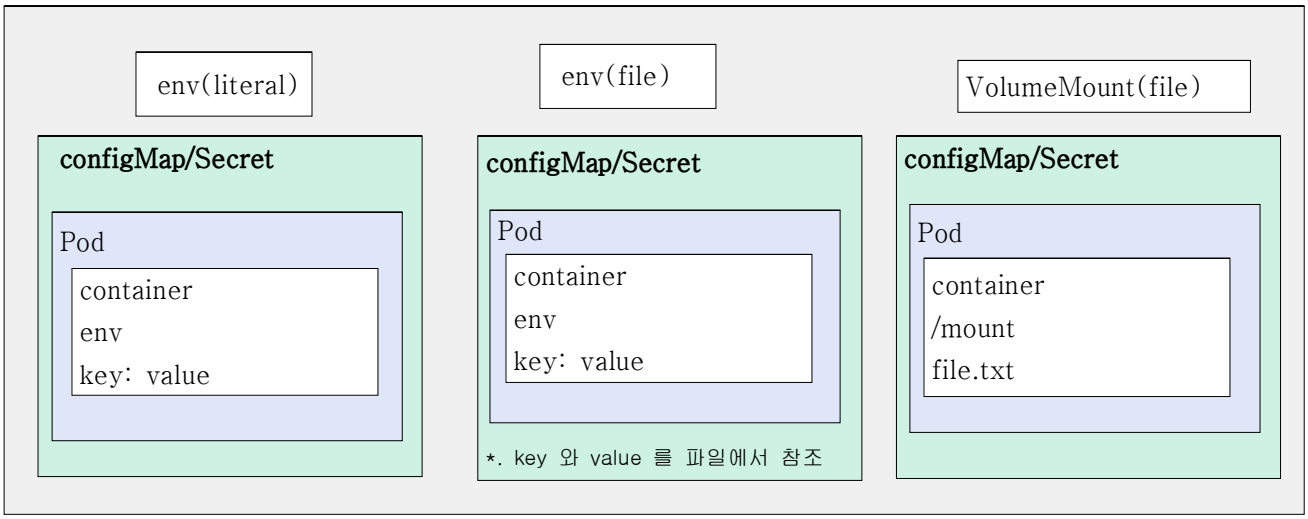
개발환경



운영환경



configMap, Secret 사용방법



```
apiVersion: v1
kind: Pod
metadata:
  name: mysql-pod
spec:
  containers:
  - name: mysql-container
    image: mysql
    env:

    - name: MYSQL_ROOT_PASSWORD
      valueFrom:
        secretKeyRef:
          name: mysql-secret
          key: root_pw

    - name: MYSQL_USER
      valueFrom:
        configMapKeyRef:
          name: mysql-config
          key: mysql_user

    - name: MYSQL_DATABASE
      valueFrom:
        configMapKeyRef:
          name: mysql-config
          key: mysql_database

    - name: MYSQL_PASSWORD
      valueFrom:
        secretKeyRef:
          name: mysql-secret
          key: mysql_user_password
```

```
$ kubectl create configmap mysql-config --from-literal mysql_user=myuser --from-literal mysql_database=userdb

$ kubectl create secret generic mysql-secret --from-literal root_pw=mypass --from-literal mysql_user_password=userpass
[vagrant@master ~]$ kubectl get configmaps
NAME                 DATA   AGE
kube-root-ca.crt     1      7d1h
mysql-config         2      43h

[vagrant@master ~]$ kubectl describe configmaps mysql-config
```

```
Name:         mysql-config
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
mysql_database:
----
userdb
mysql_user:
----
myuser

BinaryData
====

Events:  <none>

[vagrant@master ~]$ kubectl describe secrets mysql-secret
Name:         mysql-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
mysql_user_password:  8 bytes
root_pw:              6 bytes
```

configMap과 secret 을 파일로 작성해도 된다.

```
[vagrant@master work]$ cat mysql-configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-config
data:
  mysql_database: userdb
  mysql_user: myuser
```

```
$ kubectl apply -f mysql-configmap.yaml
```

```
[vagrant@master work]$ cat mysql-secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysql-secret
# data 의 Key 값은 base64 encoding 값(echo -n 'userpass' | base64)
data:
  mysql_user_password: dXNlcnBhc3M=
  root_pw: bXlwYXNz
```

```
$ kubectl apply -f mysql-secret.yaml
```

job - 하나 이상의 파드를 생성하고 지정된 수의 파드가 성공적으로 종료될때까지 계속해서 파드의 실행을 재시도한다.
파드가 성공적으로 완료되면, 성공적으로 완료된 잡을 추적한다.
지정된 수의 성공완료에 도달하면 잡이 완료된다. 잡을 일시중지하면 작업이 다시 재개될때까지 활성 파드가 삭제된다.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi3
spec:
  parallelism: 2
  template:
    spec:
      containers:
      - name: pi
        image: perl:5.34.0
        command: ["perl","-Mbignum=bpi","-wle","print bpi(3000)"]
      restartPolicy: Never
      nodeName: worker2
  backoffLimit: 4
```

cronjob - 주기적이고 반복적인 작업을 자동으로 처리.

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox:1.28
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```