BESPIN GLOBAL

Kubernetes 핵심운영(3)

베스핀아카데미 이성미(seongmi.lee@bespinglobal.com)

001. Kubernetes Storage

- Volume 운영
- StorageClass
- Persistent Volume & Persistent Volume Claim

002. Kubernetes Application 구성정보

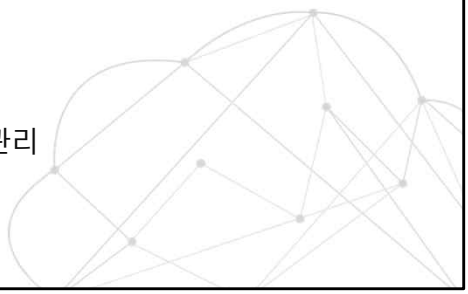
- 컨테이너 데이터 패싱
- ConfigMap 사용하기
- Secret 사용하기

003. Kubernetes API

- API 인증
- 권한관리
- Role과 ClusterRole

004. Kubernetes Package 관리

- Helm
- Helm Chart



Kubernetes Volume 운영

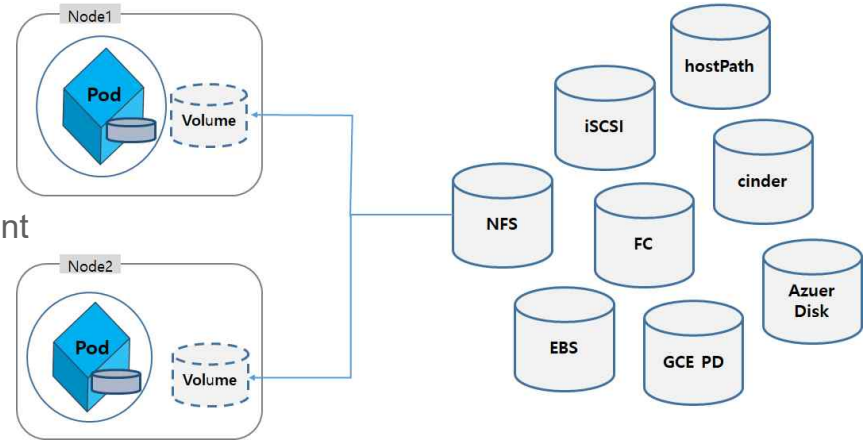
VOLUMES 소개

- Volume은 kubernetes 스토리지의 추상화 개념
  - 컨테이너는 Pod에 바인딩 되는 볼륨을 마운트하고 마치 로컬 파일시스템에 있는 것처럼 스토리지에 접근한다.
- Kubernetes 스토리지

```
volumes:
- name: html
  hostPath:
    path: /hostdir_or_file
```

- 컨테이너 단위로 mount

```
volumeMounts:
- name: html
  mountPath: /webdata
```



VOLUMES 소개

컨테이너의 볼륨 문제

컨테이너 내의 디스크에 있는 파일은 임시적이며, 컨테이너에서 실행될 때 애플리케이션에 적지 않은 몇 가지 문제가 발생한다.

- 컨테이너가 크래시 될 때 파일이 손실된다는 것이다. kubelet은 컨테이너를 다시 시작하지만 초기화된 상태이다.
- Pod에서 같이 실행되는 컨테이너 간에 파일을 공유할 때 발생한다.

Kubernetes 볼륨 추상화

- Kubernetes는 실제 데이터가 있는 디렉토리를 보존하기 위해서 저장소 볼륨을 정의한다.
- 컨테이너는 Pod에 바인딩 되는 볼륨을 마운트하고 마치 로컬 파일시스템에 있는 것처럼 스토리지에 접근한다.

Kubernetes에서 볼륨 선언

.spec.volumes 에서 파드에 제공할 볼륨을 지정

```
apiVersion: v1
kind: Pod
metadata:
  name: web-dir
spec:
  volumes:
  - name: html
    hostPath:
      path: /webdata
  containers:
...
```

Container에서 볼륨 사용

.spec.containers[*].volumeMounts 의 컨테이너에 해당 볼륨을 마운트할 위치를 선언한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: web-dir
spec:
  volumes:
  - name: html
    hostPath:
      path: /webdata
  containers:
  - image: nginx
    name: web-container
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: html
```

사용가능한 볼륨 유형

- volume type
  - emptyDir
  - HostPath
  - gitRepo
  - nfs
  - gcePersistentDisk, awsElastic-BlockStore, azureDisk
  - cinder, cephfs, iscsi, flocker, glusterfs, quobyte, rbd, flexVolume, vsphere-volume, photonPersistentDisk, scaleIO
  - configMap, secret, downwardAPI
  - persistentVolumeClaim

https://kubernetes.io/docs/concepts/storage/volumes/

Kubernetes에서 사용 가능한 볼륨

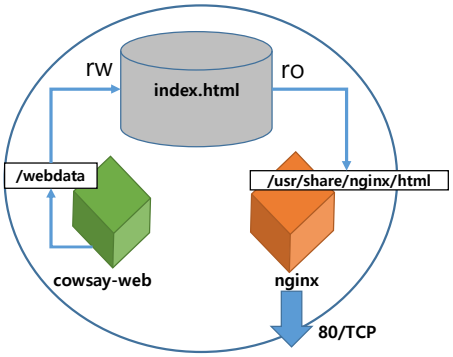
도커는 다소 느슨하고, 덜 관리되지만 볼륨이라는 개념을 가지고 있다.
도커 볼륨은 디스크에 있는 디렉터리이거나 다른 컨테이너에 있다. 도커는 볼륨 드라이버를 제공하지만, 기능이 다소 제한된다.

Kubernetes는 다양한 유형의 volume을 지원한다.
Pod는 여러 볼륨 유형을 동시에 사용할 수 있다. emptyDir 볼륨 유형은 파드의 수명에 맞춰 임시적으로 운영되나, persistent 볼륨은 파드의 수명과 관계없이 영구적으로 존재한다. 다시 말해, Pod가 삭제되어도 임시(ephemeral) 볼륨을 삭제하지만, Persistent 볼륨은 삭제하지 않는다. 그러나 볼륨의 종류와 상관없이, 파드 내의 컨테이너가 재시작 될 때 데이터는 보존된다.

기본적으로 볼륨은 디렉터리이며, 일부 데이터가 있을 수 있으며, 파드 내 컨테이너에서 접근할 수 있다.

VOLUMES을 통해 컨테이너 간에 데이터 공유하기

- emptyDir volume
  - emptyDir 볼륨은 빈 디렉토리로 시작
  - Pod 내부에서 실행중인 애플리케이션은 필요한 모든 파일을 작성
  - Pod를 삭제하면 볼륨의 내용이 손실됨
  - 동일한 Pod에서 실행되는 컨테이너 간에 파일을 공유할 때 유용



volume-empty.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: dynamic-web
spec:
  containers:
  - image: smlinux/cowsay-web
    name: web-generator
    volumeMounts:
    - name: html
      mountPath: /webdata
  - image: nginx:1.14
    name: web-server
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    ports:
    - containerPort: 80
  volumes:
  - name: html
    emptyDir: {}
```


emptyDir

emptyDir은 Pod가 Node에 할당될 때 생성되고, Pod가 삭제 될 때 같이 삭제되는 임시 볼륨이다. 이름에서 알 수 있듯이 emptyDir 볼륨은 처음에는 비어 있다. Pod내 모든 컨테이너는 emptyDir 볼륨에서 동일한 파일을 읽고 쓸 수 있지만, 볼륨은 각각의 컨테이너에서 마운트해서 사용한다.

```
volumes:
- name: html
emptyDir: {}
```

emptyDir가 사용할 미디어 지정하기

emptyDir의 물리적으로 노드에서 할당해주는 디스크에 저장이 되는데, emptyDir.medium 필드에 "Memory"라고 지정해주면, emptyDir의 내용은 물리 디스크 대신 메모리에 저장이 된다.
tmpfs는 worker node를 재부팅하면 RAM 타입 디스크에 있는 데이터는 모두 지워지나, 빠른 연산 작업을 수행해야하는 컨테이너 동작 시 유용할 수 있다. 그러나 작성하는 모든 파일이 컨테이너 메모리 제한에 포함된다.

```
volumes:
- name: html
  emptyDir:
    medium: Memory
```

emptyDir을 사용하는 multi-pod운영

1) smlinux/cowsay-web:latest
- nginx 웹 서버가 서비스할 web content를 생성하는 컨테이너
- fortune을 실행해서 5초에 한번씩 /webdata/index.html 문서를 갱신

```
# cat web-generator.sh
#!/bin/bash
  mkdir /webdata
  while true
  do
    /usr/games/fortune | /usr/games/cowsay > /webdata/index.html
    sleep 5
  done
```

```
# cat Dockerfile
  FROM ubuntu:latest
  LABEL maintainer "seongmi lee <seongmi.lee@gmail.com>"
  RUN apt-get update ; apt-get -y install fortune cowsay
  ADD web-generator.sh  /bin/web-generator.sh
  RUN chmod +x /bin/web-generator.sh
  CMD ["/bin/web-generator.sh"]
```

컨테이너 실행결과- fortune 명령어는 random하게 격언/속담을 출력. cowsay 명령은은 전달하는 메시지를 cow의 say 메시지 그림으로 표현

```
 _____________________________________________
/ You will stop at nothing to reach your    \
| objective, but only because your brakes   |
\ are defective.                            /
 ---------------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

2) cowsay-web Pod는 cowsay-web 컨테이너가 생성한 web content를 nginx 컨테이너가 80포트를 통해 서비스 하도록 multi-container POD를 구성하였다. 아래와 같이 두개의 컨테이너가 공유 스토리지를 사용하도록 서비스를 생성해보자.

 # kubectl create -f volume-empty.yaml

3) cowsay-web 포드의 IP 주소로 접속하여 서비스 실행 결과를 확인해보자.

 # kubectl get pods -o wide

 # curl <IP_Address>

4) 생성된 Pod의 세부 정보를 통해 마운트 상태를 확인해보자. 두 개의 컨테이너는 emptyDir로 정의된 임시 저장소를 각자 마운트하여 사용하고 있다.

 # kubectl describe pod dynamic-web

5) dynamic-web Pod내의 web-server 컨테이너의 마운트 디렉토리를 사용해보자. touch 명령으로 파일을 생성할 수 있는가?

 # kubectl exec dynamic-web -c web-server -- ls /usr/share/nginx/html

 # kubectl exec dynamic-web -c web-server -- touch /usr/share/nginx/html/test.html

6) fortune-cowsay-web 컨테이너는 mount를 RW가능하도록 설정하였다. 동일하게 touch 명령으로 파일이 생성되는지 확인해보자.

 # kubectl exec dynamic-web -c web-generator -- touch /webdata/test.txt
 # kubectl exec dynamic-web -c web-generator -- ls /webdata/

7) emptyDir은 Pod가 실행되는 Node의 임시 디렉토리를 통해 volume을 운영한다. 아래의 명령을 통해 Pod에서 emptyDir의 상태를 확인해보자.

 # kubectl exec dynamic-web -it -c web-generator -- /bin/bash
/# df -h
/# mount | grep webdata
/dev/mapper/centos-root on /webdata type xfs (rw,relatime,seclabel,attr2,inode64,noquota)

/# exit

8) 생성한 Pod를 제거하자.

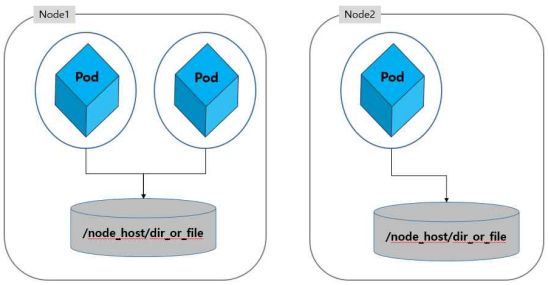
 # kubectl delete pod dynamic-web

hostPath and local

- hostPath
  - 노드의 파일시스템의 디렉토리나 파일을 컨테이너에 마운트
  - 노드에 디렉토리나 파일을 생성하여 마운트 가능
  - hostPath는 type 지시어를 이용해 mount 구성의 요구를 추가할 수 있다.

```
volumes:
- name: html
  hostPath:
    path: /hostdir_or_file
    type: FileOrCreate
```

DirectoryOrCreate	주어진 경로에 아무것도 없다면, 필요에 따라 kubelet의 소유권, 권한을 0755로 설정한 빈 디렉터리를 생성한다.
Directory	주어진 경로에 디렉터리가 있어야 함
FileOrCreate	주어진 경로에 아무것도 없다면, 필요에 따라 kubelet의 소유권, 권한을 0755로 설정한 file을 생성한다.
File	주어진 경로에 파일이 있어야 함




```
volumes:
- name: html
  hostPath:
    path: /hostdir_or_file
```

```
volumes:
- name: html
  local:
    path: /mount/dir
```

- local
  - 마운트 된 로컬 스토리지 장치를 컨테이너에 마운트

hostPath

hostPath 볼륨은 호스트 노드의 파일시스템에 있는 파일이나 디렉터리를 Pod에 마운트 한다. 같은 Node에서 동작하는 Pod는 hostPath에 있는 볼륨을 공유할 수 있다. 앞의 emptyDir과 달리 Pod가 삭제되더라도 hostPath에 있는 파일들은 삭제되지 않고, 다른 Pod가 같은 hostPath를 마운트하게 되면 동일 파일을 액세스할 수 있다. hostPath는 Pod가 동일한 호스트에 있다는 것을 알고 있는 경우 유용하다.

kubernetes가 사용하는 hostPath volumes

쿠버네티스에서 동작하는 CNI는 다음과 같이 구성 정보를 각 Node 디렉토리에 저장하고 있고, hostPath를 이용해 access한다.

```
# kubectl get pod --namespace kube-system
NAME                READY   STATUS    RESTARTS   AGE
aws-node-twb5d      1/1     Running   0          24h
aws-node-zjnp9      1/1     Running   0          24h
aws-node-zlr88      1/1     Running   0          24h
```

```
# kubectl describe -n kube-system pod aws-node-twb5d  | more
Name:                 aws-node-twb5d
Namespace:            kube-system
...
Containers:
  aws-node:
    Container ID:   docker://c3c604f475bbcbc10aa7403c65b5ba6290b5ab8223d21d1f8e0becee3539945b
    Image:          602401143452.dkr.ecr.ap-northeast-2.amazonaws.com/amazon-k8s-cni:v1.7.5-eksbuild.1
    Image ID:                                    docker-pullable://602401143452.dkr.ecr.ap-northeast-2.amazonaws.com/amazon-k8s-cni@sha256:f310c918ee2b4ebced76d2d64
...
    Mounts:
      /host/etc/cni/net.d from cni-net-dir (rw)
      /host/opt/cni/bin from cni-bin-dir (rw)
…
Volumes:
  aws-iam-token:
    Type:                 Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  86400
  cni-bin-dir:
    Type:          HostPath (bare host directory volume)
    Path:          /opt/cni/bin
```

local Volume

local 볼륨은 디스크, 파티션 또는 디렉터리 같은 마운트 된 로컬 스토리지 장치를 나타낸다.

로컬 볼륨은 정적으로 생성된 PersistentVolume으로만 사용할 수 있다. 동적으로 프로비저닝 된 것은 아직 지원되지 않는다. hostPath 볼륨에 비해 local volume은 PersistentVolume의 node nodeAffinity를 이용해 노드를 인식한다.

HostPath를 사용하는 Pod 운영

1) volume-hostpath-fluentd.yaml을 보면 알 수 있듯이 node마다 동작하는 daemonSet 컨트롤러에 의해 운영되며, hostPath를 통해 docker의 로그 디렉토리를 fluented에게 전달하였다. 노드에서 실제 로그가 쌓이는 위치인 /var/log와 /var/lib/docker/containers를 볼륨으로 마운트 하여 시스템용 프로세스들의 로그를 수집한다. 특히 /var/lib/docker/containers에는 쿠버네티스에 띄워진 Pod들이 출력하는 로그가 쌓인다.

```
# cat volume-hostpath-fluentd.yaml
…
 volumeMounts:
       - name: varlog
         mountPath: /var/log
       - name: varlibdockercontainers
         mountPath: /var/lib/docker/containers
         readOnly: true
     terminationGracePeriodSeconds: 30
     volumes:
     - name: varlog
       hostPath:
         path: /var/log
     - name: varlibdockercontainers
       hostPath:
         path: /var/lib/docker/containers
```

```
# kubectl apply -f volume-hostpath-fluentd.yaml
```

2) kubectl get pods -o wide 명령으로 확인하면 동작중인 node에 Pod가 생성되었다.

```
# kubectl get pods -o wide -n kube-system
```

3) pod 들이 마운트 상태를 확인해보자.

```
# kubectl describe pod -n kube-system fluentd-elasticsearch-XXX
...
    Environment:  <none>
    Mounts:
      /var/lib/docker/containers from varlibdockercontainers (ro)
      /var/log from varlog (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-4b4z5 (ro)
  Conditions:
    Type              Status
    Initialized       True
    Ready             True
    ContainersReady   True
    PodScheduled      True
  Volumes:
    varlog:
      Type:          HostPath (bare host directory volume)
      Path:          /var/log
      HostPathType:
    varlibdockercontainers:
      Type:          HostPath (bare host directory volume)
      Path:          /var/lib/docker/containers
      HostPathType:
    default-token-4b4z5:
      Type:        Secret (a volume populated by a Secret)
      SecretName:  default-token-4b4z5
      Optional:    false
```

4) fluentd 포드에 들어가서 실제로 거기에 로그가 쌓이고 있는지 확인해보자.

```
# kubectl exec -n kube-system -it fluentd-elasticsearch-XXX -- bash
/# ls /var/log
/# ls /var/lib/docker/containers
/# exit
```

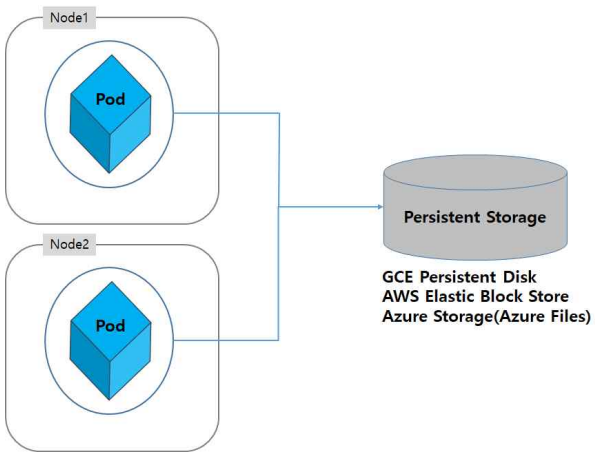
참고: fluentd는 루비 기반으로 개발되었고, 다양한 플러그인 들을 사용할 수 있는 로그 수집기로 쿠버네티스가 아닌 환경에서도 많이 사용된다. CNCF 재단에 속해 있는 범용 로그 수집용 오픈소스 프로젝트이며 로그를 수집해서 다양한 외부 저장소로 보낼 수 있도록 되어 있지만 아래 예제는 쿠버네티스에서 발생한 로그들을 수집해서 elasticsearch로 저장하도록 하고 있다.

https://github.com/fluent/fluentd-Kubernetes-daemonset에서 쿠버네티스 배포용 yaml 파일 정보를 볼 수 있다.

Shared Volume

- 여러 개의 Pod들이 동일 데이터를 참조
- K8S의 Shared Disk를 Pod 볼륨으로 사용

Volume Type
awsElasticBlockStore 볼륨
gcePersistentDisk
NFS volume
etc ...



https://cloud.google.com/compute/docs/disks?hl=ko

Shared Volume

하나의 Application Service를 지원하는 Pod들은 동일 데이터를 가지고 서비스해야 한다. 이때 Kubernetes가 지원하는 공유 스토리지를 이용할 수 있다.

GCE Persistent Disk
Google Persistent Disk는 Google Cloud Platform을 위한 내구성이 뛰어난 고성능 블록 스토리지입니다. Persistent Disk는 Compute Engine이나 Google Kubernetes Engine에서 실행하는 인스턴스에 연결 가능한 SSD 또는 HDD 스토리지를 제공합니다. 스토리지 볼륨 크기를 투명하게 조절하고 빠르게 백업하며 동시 읽기를 지원할 수 있습니다.

GCE Persistent Disk 생성

gcloud compute disks create <DISK_NAME> --size <DISK_SIZE> --zone <VALUE>

```
$ gcloud compute disks create persistentdisk  --size=1GiB  --zone=europe-west1-b
$ gcloud container clusters list
```

gcePersistentDisk 볼륨을 사용한 Pod 생성

```
apiVersion: v1
kind: Pod
metadata:
  name: gce-volume-pod
spec:
  containers:
  - image: nginx
    name: web
    volumeMounts:
    - name: webdata
      mountPath: /container/dir
  volumes:
  - name: webdata
    gcePersistentDisk:
      pdName: persistentdisk
      fsType: ext4
```

AWS Elastic Block Store volume
awsElasticBlockStore 볼륨은 아마존 웹 서비스(AWS) EBS 볼륨을 Pod에 마운트 한다. Pod를 제거할 때 지워지는 emptyDir 와는 다르게 EBS 볼륨의 내용은 유지되고, 볼륨은 마운트 해제만 된다. 이 의미는 EBS 볼륨에 데이터를 미리 채울 수 있으며, Pod간에 데이터를 "전달(handed off)" 할 수 있다.

awsElasticBlockStore 볼륨을 사용할 때 몇 가지 제한
- 파드가 실행 중인 노드는 AWS EC2 인스턴스여야 함
- EC2 인스턴스는 EBS 볼륨과 동일한 리전과 동일한 가용영역(availability zone)에 있어야 함
- EBS 볼륨 마운트는 단일 EC2 인스턴스만 지원

EBS 볼륨 생성하기
aws ec2 create-volume --availability-zone <VALUE> --size <DISK_SIZE> --volume-type <VALUE>

```
$ aws ec2 create-volume --availability-zone=ap-northeast-2a --size=10 --volume-type=gp2
{
    "AvailabilityZone": "ap-northeast-2a",
    "CreateTime": "2021-05-25T16:37:35+00:00",
    "Encrypted": false,
    "Size": 10,
    "SnapshotId": "",
    "State": "creating",
    "VolumeId": "vol-0fc935f102c2e8b3a",
    "Iops": 100,
    "Tags": [],
    "VolumeType": "gp2",
    "MultiAttachEnabled": false
}
```

AWS EBS는 동일 availability-zone에서만 적용할 수 있다. 아래와 같이 EBS를 생성한 AZ에 있는 노드를 구분하자

```
$ kubectl label nodes ip-192-168-42-136.ap-northeast-2.compute.internal az=ap-northeast-2a
$ kubectl label nodes ip-192-168-70-214.ap-northeast-2.compute.internal az=ap-northeast-2c
$ kubectl label nodes ip-192-168-5-214.ap-northeast-2.compute.internal az=ap-northeast-2b
$ kubectl get nodes -L az
NAME                                               STATUS   ROLES    AGE   VERSION              AZ
ip-192-168-42-136.ap-northeast-2.compute.internal  Ready    <none>   35h   v1.19.6-eks-49a6c0   ap-northeast-2a
ip-192-168-5-214.ap-northeast-2.compute.internal   Ready    <none>   35h   v1.19.6-eks-49a6c0   ap-northeast-2b
ip-192-168-70-214.ap-northeast-2.compute.internal  Ready    <none>   35h   v1.19.6-eks-49a6c0   ap-northeast-2c
```

EBS 볼륨을 사용하는 Pod 생성

container가 AZ=ap-northeast-2a의 워커 노드에서 실행되도록 nodeSelector를 구성하여 pod를 동작 시키자.

```
$ cat volume-ebs.yaml
...
    name: test-container
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: test-volume
  volumes:
  - name: test-volume
    # 이 AWS EBS 볼륨은 이미 존재해야 한다.
    awsElasticBlockStore:
      volumeID: "vol-07ab4a74f151ea1d1"
      fsType: ext4
```

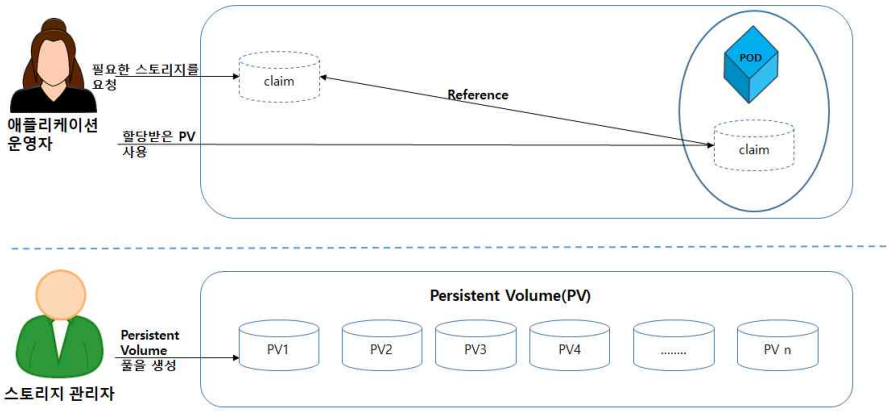
```
$ kubectl expose pod test-ebs --type=LoadBalancer --selector=app=web
$ kubectl get svc
test-ebs       LoadBalancer   10.100.13.15     a70c564dcdb90410eb18567effbb0e4a-1341969458.ap-northeast-2.elb.amazonaws.com   80:31886/TCP   123m
```

```
$ curl a70c564dcdb90410eb18567effbb0e4a-1341969458.ap-northeast-2.elb.amazonaws.com
<html>
<head><title>403 Forbidden</title></head>
<body bgcolor="white">
<center><h1>403 Forbidden</h1></center>
<hr><center>nginx/1.14.2</center>
</body>
</html>


$ kubectl exec test-ebs -it -- /bin/bash
root@test-ebs:/# cd /usr/share/nginx/html/

root@test-ebs:/usr/share/nginx/html# ls
lost+found

root@test-ebs:/usr/share/nginx/html# cat > index.html
<h1>Bespin Global</h1>

root@test-ebs:/usr/share/nginx/html# exit
exit

$ curl a70c564dcdb90410eb18567effbb0e4a-1341969458.ap-northeast-2.elb.amazonaws.com
<h1>Bespin Global</h1>


질문: test-ebs 파드를 삭제하고 다시 create 했다. curl 명령 실행 시 어떤 결과가 나올까?
$ kubectl delete pod test-ebs
$ kubectl create –f volume-ebs.yaml
$ curl a70c564dcdb90410eb18567effbb0e4a-1341969458.ap-northeast-2.elb.amazonaws.com
<결과는?>

참고: https://kubernetes.io/ko/docs/concepts/storage/volumes/#gcepersistentdisk
참고: https://kubernetes.io/ko/docs/concepts/storage/volumes/#awselasticblockstore
참고: https://docs.aws.amazon.com/cli/latest/reference/ec2/create-volume.html
```

StorageClass

Kubernetes Volumes 운영환경 분리

- 기본 스토리지 운영환경을 분리
  - 관리자 : 스토리지 구성
  - 개발자 : 필요한 만큼 요구
- PersistentVolumes
- PersistentVolumeClaims



PersistentVolume과 PersistentVolumeClaims

지금까지 kubernetes가 지원하는 storage volume에 대해 확인하였다.
kubernetes는 많은 종류의 스토리지 볼륨을 제공하고 있다. emptyDir이나 hostPath는 간단하게 구현하는 것이 가능하다. 그러나 FC, iSCSI, cephfs, NFS shared storage와 같이 스토리지 운영자의 도움 없이는 사용할 수 없는 스토리지 볼륨들도 많이 있다.

그런데 이러한 스토리지 볼륨을 구현하는 것이 하드웨어 인프라를 잘 모르는 애플리케이션 운영자가 입장에서는 사실상 쉽지 않은 작업이다. 그래서 Kubernetes는 스토리지 구성과 애플리케이션 운영을 분리하여 적용할 수 있도록 지원하는데, 인프라 관련 운영은 시스템 운영자가 구성하고, 애플리케이션 운영자(개발자)는 애플리케이션에 필요한 영구 스토리지를 요청만 하는 것으로 스토리지를 사용할 수 있도록 지원하는 것이 PersistentVolume(PV)과 PersistentVolumeClaim(PVC)이고 Kubernetes는 PersistentVolume Controller에 의해 운영된다.

Kubernetes Volumes 운영환경 분리

- 기본 스토리지 운영환경을 분리
  - 관리자 : 스토리지 구성
  - 개발자 : 필요한 만큼 요구
- PersistentVolumes
- PersistentVolumeClaims

storageclass.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
allowVolumeExpansion: true
mountOptions:
  - debug
volumeBindingMode: Immediate
```


StorageClass

스토리지클래스는 관리자가 제공하는 스토리지의 "classes"를 설명할 수 있는 방법을 제공한다.
다른 클래스는 서비스의 품질 수준 또는 백업 정책, 클러스터 관리자가 정한 임의의 정책에 매핑 될 수 있다.
쿠버네티스 자체는 클래스가 무엇을 나타내는지에 대해 상관하지 않는다. 다른 스토리지 시스템에서는 이 개념을 "프로파일"이라고도 한다.

스토리지클래스 리소스
각 스토리지클래스에는 해당 스토리지클래스에 속하는 퍼시스턴트 볼륨을 **동적으로 프로비저닝 할 때 사용되는 provisioner, parameters와 reclaimPolicy 필드가 포함**된다.

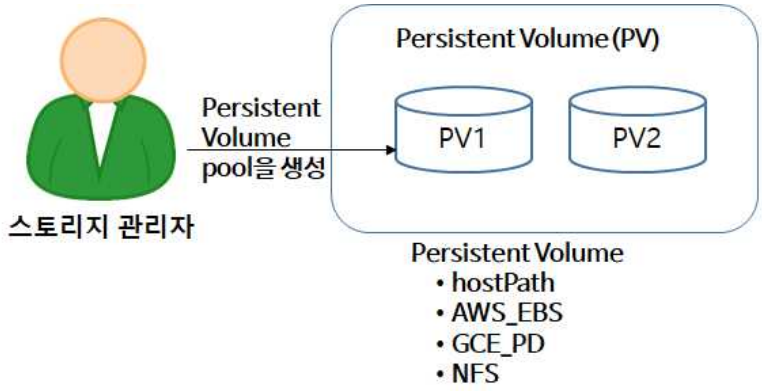
프로비저너
https://kubernetes.io/ko/docs/concepts/storage/storage-classes/#%ED%94%84%EB%A1%9C%EB%B9%84%EC%A0%80%EB%84%88

Persistent Volume & Persistent Volume Claim

PV(PersistentVolume) 생성

pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pvname
spec:
  capacity:
    storage: <storage_size>
  accessModes:
  - ReadWriteOnce
  - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    server: <NFS_Server>
    path: <Share_Storage>
```



PV(PersistentVolume) 생성 필드

Capacity : 볼륨의 용량을 정의한다.

Reclaim Policy(persistentVolumeReclaimPolicy) :

재사용정책. PV는 연결된 PVC가 삭제된 후 다시 다른 PVC에 의해서 재사용이 가능한데, 재사용 시에 디스크의 내용을 지울지 유지할지에 대한 정책을 Reclaim Policy를 이용하여 설정이 가능하다.
Reclaim Policy은 모든 디스크에 적용이 가능한 것이 아니라, 디스크의 특성에 따라서 적용이 가능한 Policy가 있고, 적용이 불가능한 Policy 가 있다.

- **Retain** : 삭제하지 않고 PV의 내용을 유지한다. 수동으로 회수가 필요한 볼륨 운영 시 필요
- **Recycle** : 재사용이 가능. 재사용 시 데이타의 내용을 자동으로 rm -rf 로 삭제한다.
- **Delete** : 볼륨의 사용이 끝나면, 해당 볼륨 내용이 삭제됨(AWS_EBS, GCE_PD, Azure Disk등).

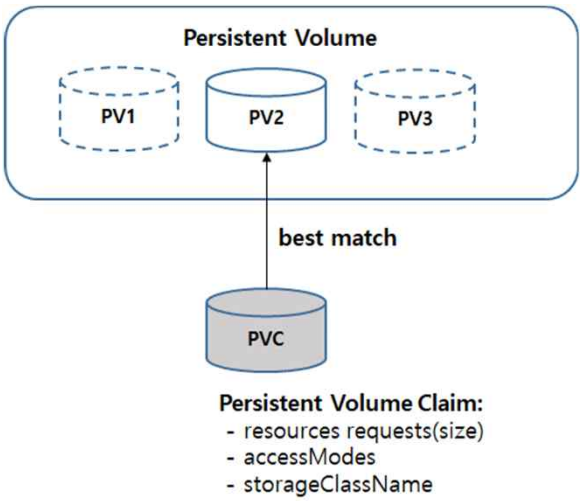
AccessModes :
PV에 대한 동시에 Pod에서 접근할 수 있는 정책을 정의한다.

- **ReadWriteOnce (RWO)** : 단일 노드에서 읽기/쓰기 마운트 가능
- **ReadOnlyMany(ROX)** : 다수 노드에서 읽기 전용으로 마운트 가능
- **ReadWriteMany(RWX)** : 다수 노드에서 읽기/쓰기로 마운트 가능

PVC(PersistentVolumeClaim) 생성

pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-name
spec:
  resources:
    requests:
      storage: size
  accessModes:
  - ReadWriteOnce
  - ReadOnlyMany
  storageClassName: "manuel“
```



PVC(PersistentVolumeClaim) 생성 필드

클러스터 사용자가 Pod 중 하나에서 영구 스토리지를 사용해야 할 경우 먼저 필요한 최소 크기와 Access mode를 지정해 PersistentVolumeClaim 메니페스트를 생성한다. 그런 다음 PersistentvolumeClaim을 Pod 내부의 볼륨 중 하나로 사용할 수 있다

Capacity : 볼륨의 용량을 정의한다.

Resources.requests:
요청하는 Storage size를 설정한다.
Kubernetes는 요청하는 디스크 크기에 가장 적합한 PV를 선택해서 할당한다.

accessModes:

요청하는 디스크 access mode를 설정한다.
- **ReadWriteOnce (RWO)** : 단일 노드에서 읽기/쓰기 마운트 가능
- **ReadOnlyMany(ROX)** : 다수 노드에서 읽기 전용으로 마운트 가능
- **ReadWriteMany(RWX)** : 다수 노드에서 읽기/쓰기로 마운트 가능

동작 LifeCycle

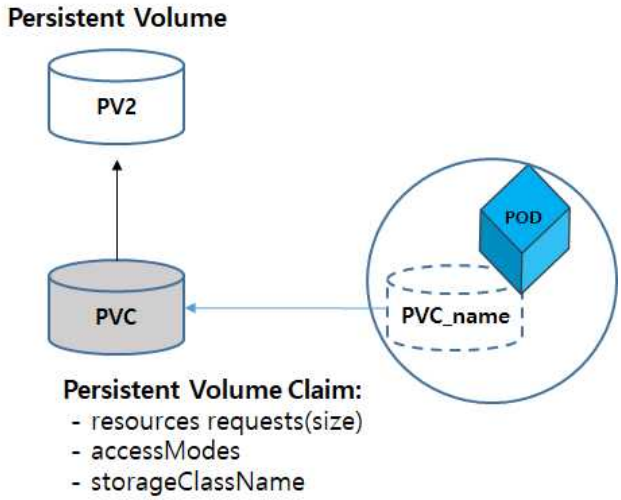
PV, PVC 동작 LifeCycle은 총 4가지 형태를 가지고 변화된다.
- **Available** : PV 생성
- **Bound** : PVC에 의해 바인딩 되었을 경우
- **Released** : PVC가 삭제되었을 경우
- **Fail** : PV에 문제가 발생하였을 경우

LifeCycle은 **Available → Bound → Released → Available ...** 순으로 반복되며, Released 즉 PVC가 삭제되었을 경우 실제 스토리지에 있는 파일을 어떻게 관리할 것인지에 대한 부분은 PV에 적용한 Reclaiming(재활용) 정책에 따라 달라진다. 또한 바인딩 된 PVC를 삭제해 Released 될 때까지 다른 PVC는 동일한 Persistent Volume을 사용할 수 없다.

Pod에서 PersistentVolumeClaim을 사용

pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod
spec:
  containers:
  - image: image
    name: container-name
    volumeMounts:
    - name: volume-name
      mountPath: /mount/dir
  volumes:
  - name: volume-name
    persistentVolumeClaim:
      claimName: pvc-name
```



Pod에서 PersistentVolumeClaim을 사용

생성된 PersistentVolumeClaim을 Pod에서 사용한다.
Pod가 실행될 때 할당된 PVC는 볼륨을 해제할 때 까지 다른 POD가 할당해서 사용할 수 없다.

실습 | Volume 사용

- earth라는 이름의 새 namespace를 생성하시오.
- 다음 조건으로 earth-project-earthflower-pv라는 이름의 새 PersistentVolume을 생성하시오.
- size: 2Gi
- accessMode: ReadWriteOnce
- hostPath /Volumes/Data

- 네임스페이스 earth에 다음의 조건으로 earth-project-earthflower-pvc라는 이름의 새로운 PersistentVolumeClaim을 생성하시오.
- size : 2Gi
- accessMode: ReadWriteOnce
- PVC는 PV에 올바르게 바인딩되어야 한다.

- 마지막으로 /tmp/project-data에 해당 볼륨을 마운트하는 새로운 Deployment project-earthflower를 네임스페이스 earth에 생성하시오. 해당 배포의 pod는 httpd:2.4.41-alpine를 사용한다.

볼륨 사용하기 : PV, PVC의 volume을 Pod에 마운트하기

1. Namespace 생성

```
# kubectl create namespace earth
```

2. PV 생성

```
# cat lab1-pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: earth-project-earthflower-pv
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  hostPath:
    path: /Volumes/Data
```

2. PVC 생성

```
# cat lab1-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: earth-project-earthflower-pvc
  namespace: earth
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

4. Deployment 생성

```
# cat lab1-deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: project-earthflower
  namespace: earth
spec:
  replicas: 1
  selector:
    matchLabels:
      app: project-earthflower
  template:
    metadata:
      labels:
        app: project-earthflower
    spec:
      containers:
      - image: httpd:2.4.41-alpine
        name: httpd
        volumeMounts:
        - mountPath: "/tmp/project-data"
          name: html
      volumes:
        - name: html
          persistentVolumeClaim:
            claimName: earth-project-earthflower-pvc
```

5. 최종 결과 확인

```
$ kubectl -n earth describe pod project-earthflower-XXXX-YYYY | grep -A2 Mounts:
    Mounts:
      /tmp/project-data from html (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-cgm7h (ro)
```

참고링크:
https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistent-volumes
https://kubernetes.io/docs/concepts/storage/_print/#persistentvolumeclaims
https://kubernetes.io/docs/concepts/storage/_print/#claims-as-volumes
https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

지금까지 배운 내용을 기준으로

- Kubernetes 스토리지
  - Pod에 emptyDir 디스크를 연결할수 있어야합니다.
  - HostPath를 통해 영구적으로 데이터를 보존할수 있어야합니다.

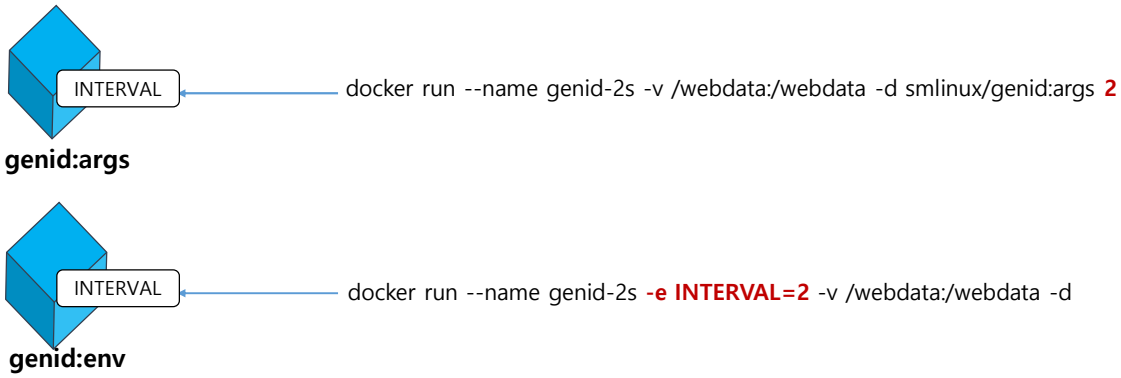
- Persistent Volume & Persistent Volume Claim
  - PV, PVC를 구성할수 있어야합니다.
  - Pod에 PVC를 연결할수 있어야합니다.

컨테이너 데이터 패싱

컨테이너 데이터 패싱

컨테이너 애플리케이션 데이터 분리

- 애플리케이션 구성 데이터 분리
  - 컨테이너 Application에 데이터 전달
    - Command-line Argument
    - Environment Variable
    - Volume Mount



컨테이너 데이터 패싱

많은 Application 프로그램은 configfile, commandline argument, environment variable의 조합을 사용하여 운영된다.
컨테이너화 된 Application을 운영 시 이식성을 높게 운영하려면, 이러한 configfile, argument, environment variable들을 image content에서 분리시켜 운영하는것이 좋다.

- **Command-line Arguments를 이용해 컨테이너로 데이터 전달**

```
$ cat generate_id.sh
#!/bin/bash
INTERVAL=$1
mkdir -p /webdata
while true
do
  echo "$(date +%Y%m%d)-$INTERVAL : generate a random fake identity"
  /usr/bin/rig | /usr/bin/boxes  > /webdata/index.html
  sleep $INTERVAL
done

$ cat dockerfile
FROM ubuntu:18.04
RUN apt-get update ; apt-get -y install rig boxes
ADD generate_id.sh /bin/generate_id.sh
RUN chmod +x /bin/generate_id.sh
ENTRYPOINT ["/bin/generate_id.sh"]
CMD ["5"]
```

```
$ docker build -t smlinux/genid:args .
$ docker push smlinux/genid:args
$ docker run --name  genid-5s -v /webdata:/webdata -d smlinux/genid:args
$ watch cat /webdata/index.html
$ docker rm -f genid-5s
```

CMD를 통해 전달되는 argument를 컨테이너 실행 시 변경할 수 있다.

```
$ docker run --name genid-2s -v /webdata:/webdata -d smlinux/genid:args 2
$ watch cat /webdata/index.html
$ docker rm -f genid-2s
```

- **Environment Variable를 이용해 컨테이너로 데이터 전달**

  $ **cat generate_id.sh**

```
#!/bin/bash
mkdir -p /webdata
while true
do
  echo "$(date +%Y%m%d)-$INTERVAL : generate a random fake identity"
  /usr/bin/rig | /usr/bin/boxes  > /webdata/index.html
  sleep $INTERVAL
done
```

  $ cat dockerfile

```
FROM ubuntu:18.04
RUN apt-get update ; apt-get -y install rig boxes
ENV INTERVAL 5
ADD generate_id.sh /bin/generate_id.sh
RUN chmod +x /bin/generate_id.sh
ENTRYPOINT ["/bin/generate_id.sh"]
```

  $ docker build -t smlinux/genid:env .
  $ docker push smlinux/genid:env
  $ docker run --name genid-5s -v /webdata:/webdata -d smlinux/genid:env
  $ watch cat /webdata/index.html
  $ docker rm -f genid-5s

CMD를 통해 전달되는 argument를 컨테이너 실행 시 변경할 수 있다.

```
$ docker run --name genid-2s -e INTERVAL=2 -v /webdata:/webdata -d smlinux/genid:env
$ watch cat /webdata/index.html
$ docker rm -f genid-2s
```

컨테이너 데이터 패싱

Kubernetes에서 애플리케이션 데이터 분리 적용

- kubernetes에서 컨테이너 데이터를 분리해서 적용
  - Command-line Argument

```
kind: Pod
spec:
  - image: some/image
    command: ["/bin/command"]
    args: ["arg1", "arg2", "arg3"]
...
```

  - Environment Variable

```
kind: Pod
spec:
 containers:
 - image: image
   env:
   - name: VARIABLE
     value: "value"
...
```


쿠버네티스 데이터 패싱

- **kubernetes에 container arguments 적용 예**

$ **cat genid-args.yaml**

```
apiVersion: v1
kind: Pod
metadata:
  name: genid-args
spec:
  containers:
  - image: smlinux/genid:args
    args: ["2"]
    name: faceid-generator
    volumeMounts:
    - name: html
      mountPath: /webdata
  - image: nginx:1.14
    name: web-server
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    ports:
    - containerPort: 80
  volumes:
  - name: html
    emptyDir: {}
```

```
$ kubectl apply -f genid-args.yaml
$ kubectl get pods -o wide
genid-args   2/2     Running   0          6h53m   192.168.14.120   ip-192-168

NODE$ curl 192.168.14.120
/**************************/
/* Cornelia Mullen        */
/* 71 Dorwin Rd           */
/* Indianapolis, IN  46206 */
/* (317) xxx-xxxx         */
/**************************/
```

- **kubernetes에 environment variable 적용예**

$ cat genid-env.yaml

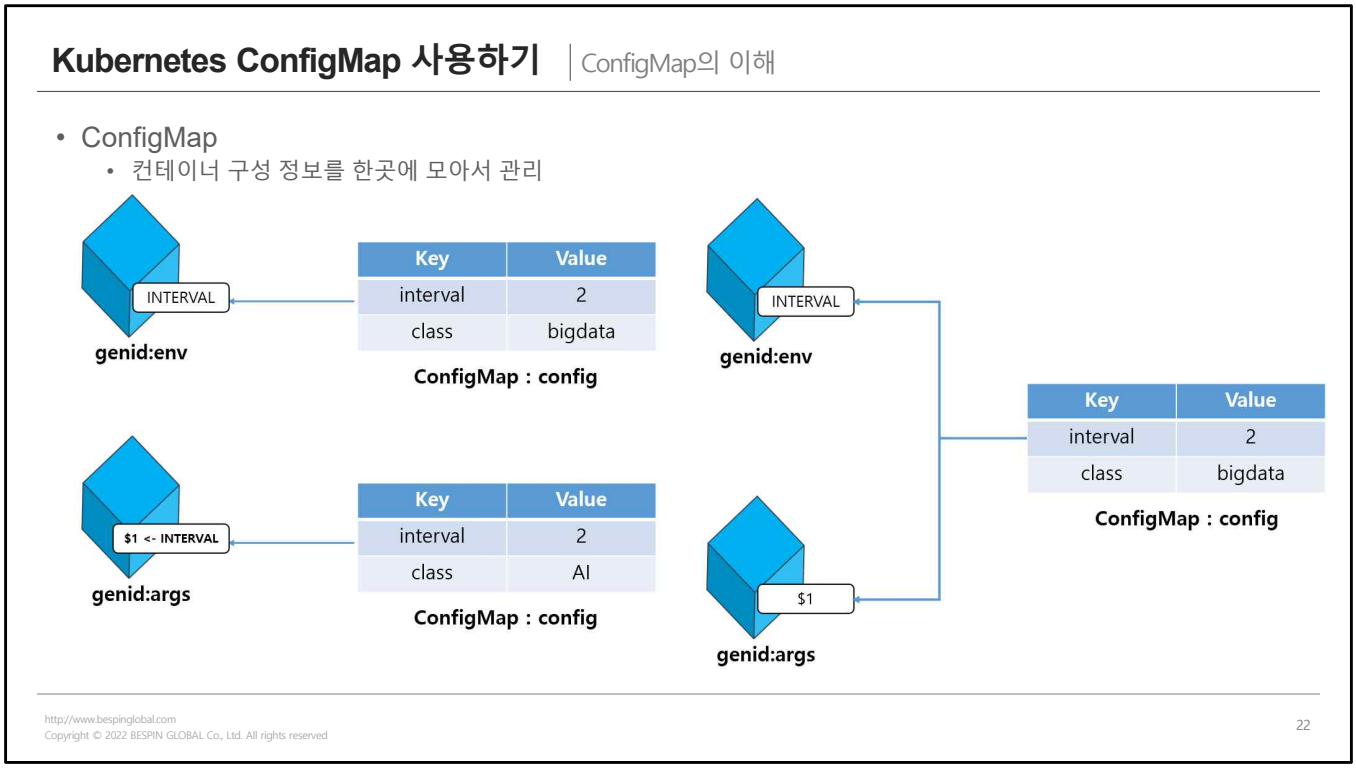
```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: genid-env
spec:
  containers:
  - image: smlinux/genid:env
    name: fakeid-generator
    env:
    - name: INTERVAL
      value: "2"
    volumeMounts:
    - name: html
      mountPath: /webdata
  - image: nginx:1.14
    name: web-server
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    ports:
    - containerPort: 80
  volumes:
  - name: html
    emptyDir: {}
```

$ kubectl apply -f genid-env.yaml
$ kubectl get pods -o wide

```
NAME          READY   STATUS    RESTARTS   AGE     IP               NODE
genid-args    2/2     Running   0          6h53m   192.168.14.120   ip-192-16..
genid-env     2/2     Running   0          6h53m   192.168.6.54     ip-192-16..

NODE$ curl 192.168.6.54
/**********************/
/* Brant Glenn        */
/* 756 East Parson St */
/* Aurora, IL  60507  */
/* (708) xxx-xxxx     */
/**********************/
```

ConfigMap 사용하기



ConfigMap

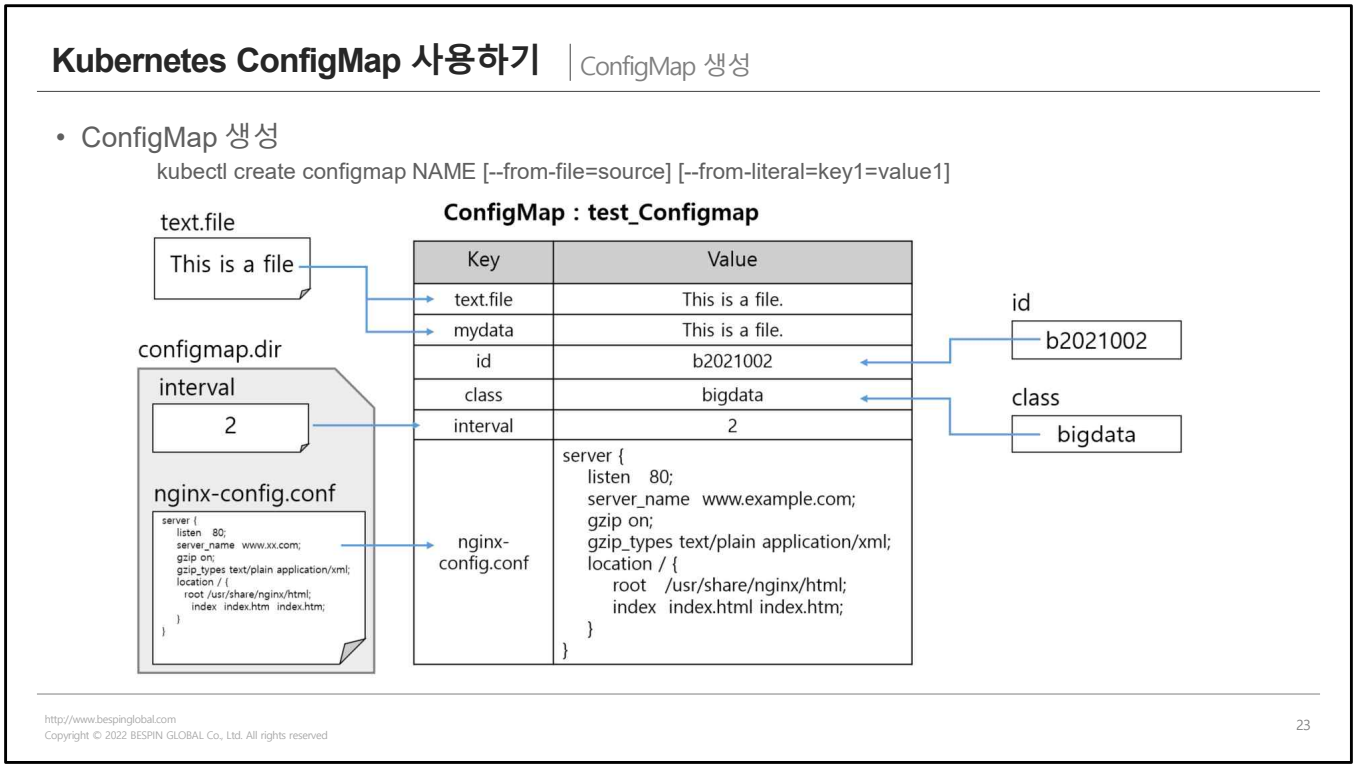
애플리케이션 구성의 중요한 점은 환경에 따라 다양하고, 자주 변경되는 설정 옵션을 애플리케이션 소스 코드와 분리해서 유지하는 것이 좋다.
쿠버네티스는 설정 옵션을 ConfigMap이라는 별도의 객체로 분리할 수 있다.

container1 <---------- interval=2, class=bigdata

container2 <---------- interval=2, class=AI

Pod는 ConfigMap을 이름으로 참조하므로, 서로 다른 환경에 동일한 Pod 설정을 유지하면서 다른 구성을 사용할 수 있다.
애플리케이션이 ConfigMap을 사용하는 방식에 관계없이 분리된 독립 실행형 객체를 설정하면 동일한 이름의 ConfigMap에 대해 각각 다른 메니페스트(개발, 테스트, QA)의 유지가 가능해진다.

슬라이드의 오른쪽 그림은 서로 다른 Pod에서 동일한 CofnigMap을 가지고 실행된 것이다. 이로 인해 단일 관리 포인트를 운영할 수 있다.
또한 그 반대로, 여러 개의 configMap을 하나의 Pod에 전달하는 것도 가능하다.



ConfigMap 생성하기

kubectl create configmap 명령 사용하여 간단하게 생성할수 있다.

kubectl create configmap NAME [--from-file=source] [--from-literal=key1=value1]

- **key=value를 이용해 ConfigMap 항목 생성하기**
  $ kubectl create configmap CONFIG_NAME --from-literal=id=b2021002 --from-literal=class=bigdata

- **파일 내용으로 ConfigMap 항목 생성하기**
  $ cat text.file
  This is a file.

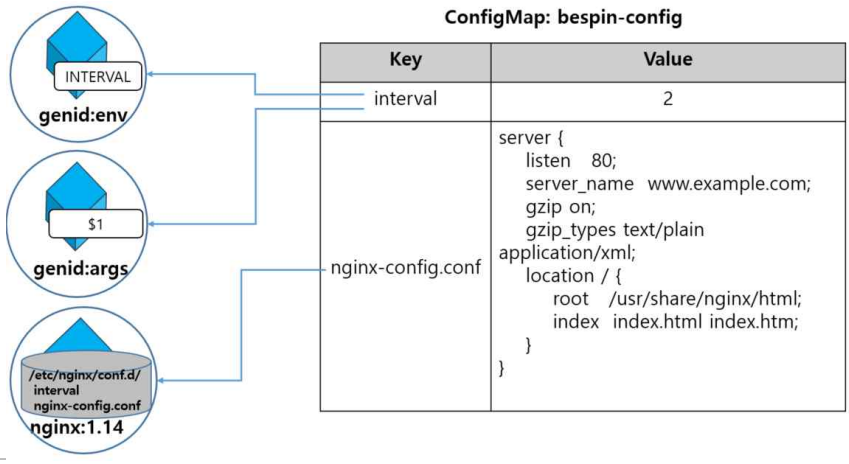
  $ kubectl create configmap CONFIG_NAME --from-file=text.file
  $ kubectl create configmap CONFIG_NAME --from-file=mydata=text.file

- **디렉터리 내의 모든 파일들을 ConfigMap 생성하기**
  $ ls configmap.dir/
  interval nginx-config.conf

  $ kubectl create configmap CONFIG_NAME --from-file=/configmap.dir/

Kubernetes ConfigMap 사용하기 | 애플리케이션 배포(using configMap)

- 정의된 ConfigMap을 Pod의 Container에 전달하는 방법
  - 환경변수로 전달
  - Command-line Argument로 전달
  - Volume에 ConfigMap을 사용하여 컨테이너 디렉토리에 Mount



ConfigMap을 이용해 컨테이너에 데이터 전달

- **환경변수로 전달하기**

```
spec:
  containers:
  - image: smlinux/cowsay:env
    env:
    - name: VARNAME
      valueFrom:
        configMapKeyRef:
          name: ConfigMap_name
          key: key_name
...
```

- **Command-line Argument로 전달하기**

```
spec:
  containers:
  - image: container_image
    env:
    - name: VARNAME
      valueFrom:
        configMapKeyRef:
          name: ConfigMap_name
          key: key_name
    args: ["$VARNAME"]
...
```

- **Volume에 ConfigMap 엔트리로 사용하여 전달하기**

```
  - image: container_image
    name: container-name
    volumeMounts:
    - name: some_name
      mountPath: /mount_point
      readOnly: true
    ...
  volumes:
  - name: some_name
    configMap:
      name: ConfigMap_name
```

EXAMPLE
configmap을 생성하여 애플리케이션이 필요로하는 데이터를 저장하고, 배포되는 애플리케이션에게 전달한다. 이렇게 configmap을 이용하여 데이터를 관리하면 single point 관리가 가능하여 이후 configmap을 통해 데이터 변경관리가 가능하다.

1. configmap 생성
$ kubectl delete configmap --all
$ ls genid-web-config/
nginx-config.conf

$ kubectl create configmap bespin-config --from-literal=interval=2 --from-file=genid-web-config/
$ kubectl get configmap
$ kubectl describe configmaps bespin-config

2. 앞에서 사용했던 genid-env.yaml 파일을 수정하여 configmap의 interval의 값을 환경변수 INTERVAL로 사용하도록 구성해보자.
bespin-config configmap을 컨테이너에서 사용할 수 있도록 yaml을 수정한다.
$ cp genid-env.yaml genid-env-cm.yaml
$ vim genid-env-cm.yaml
```
apiVersion: v1
kind: Pod
metadata:
  name: genid-env-cm
spec:
  containers:
  - image: smlinux/genid:env
    env:
    - name: INTERVAL
      valueFrom:
        configMapKeyRef:
          name: bespin-config
          key: interval
    name: fakeid-generator
    ...
```

3. 수정한 genid-env-cm.yaml 파일을 동작시키고, configMap에 정의된 interval 변수가 환경변수 INTERVAL로 적용되었는지 확인해보자.
$ kubectl apply -f genid-env-cm.yaml
$ kubectl get pods -o wide
genid-env-cm 2/2 Running 0 47s 192.168.28.196 ip-192-168

NODE$ curl 192.168.28.196
2초마다 content가 변경되는가?

4. configMap bespin-config의 interval 값을 arguments로 전달해보자. 앞에서 사용한 genid-args.yaml 파일을 수정하여 적용하자.
$ cp genid-args.yaml genid-args-cm.yaml
$ vi genid-args-cm.yaml
```
...
  name: genid-args-cm
spec:
  containers:
  - image: smlinux/genid:args
    env:
    - name: INTERVAL
      valueFrom:
        configMapKeyRef:
          name: bespin-config
          key: interval
    args: ["$INTERVAL"]
    name: fakeid-generator
   ...
```

5. 수정한 cowsay-args-cm.yaml 파일을 동작시키고, configMap에 정의된 interval 변수가 $1 아규먼트로 적용되었는지 확인해보자.
$ kubectl apply -f genid-args-cm.yaml
$ kubectl get pod -o wide
genid-args-cm 2/2 Running 0 89s 192.168.11.162 ip-192-168

NODE$ curl 192.168.11.162

6. 동작중인 Pod를 중지시키고 ConfigMap의 interval 값을 변경해서 운영해보자.
$ kubectl delete pod --all

7. configMap bespin-config의 interval 값을 변경한다.
$ kubectl edit configmaps bespin-config
```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will be
# reopened with the relevant failures.
#
apiVersion: v1
data:
  interval: "5"
  nginx-config.conf: |
    server {
        listen    80;
        server_name  www.example.com;

        gzip on;
        gzip_types text/plain application/xml;

        location / {
            root   /usr/share/nginx/html;
            index  index.html index.htm;
        }
    }
kind: ConfigMap
...
```

8. 변경된 interval값을 적용하여 pod가 실행되도록 genid-args-cm.yaml, genid-args-cm.yaml 를 다시 실행하자.
$ kubectl apply -f genid-args-cm.yaml -f genid-env-cm.yaml
$ kubectl get pods -o wide
```
genid-args-cm   2/2     Running   0          20s   192.168.14.120   ip-192-168
genid-env-cm    2/2     Running   0          20s   192.168.6.54     ip-192-168
```

NODE$ curl 192.168.14.120
NODE$ curl 192.168.6.54

9. 볼륨마운트를 통해 ConfigMap 데이터를 전달하는 방법을 확인해보자.
genid-web-config 디렉토리에 저장된 nginx-config.conf 파일을 web-server pod에 전달하여 해당 설정파일로 서비스가 동작되도록 구현한다.
$ cp genid-env-cm.yaml genid-volume-cm.yaml
$ vi genid-volume-cm.yaml
```
apiVersion: v1
kind: Pod
metadata:
  name: genid-volume-cm
spec:
...
  - image: nginx:1.14
    name: web-server
    ports:
    - containerPort: 80
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    - name: config
      mountPath: /etc/nginx/conf.d
      readOnly: true
  volumes:
  - name: html
    emptyDir: {}
  - name: config
    configMap:
```

```
      name: bespin-config

$ kubectl apply  -f genid-volume-cm.yaml
$ kubectl describe pod genid-volume-cm
....
 web-server:
    Container ID:   docker://b73738a1550bc2bb8a7b15ffa12f680bd4640ea320a24217c7acb433be3192b5
    Image:          nginx:1.14
    Image ID:       docker-
pullable://nginx@sha256:f7988fb6c02e0ce69257d9bd9cf37ae20a60f1df7563c3a2a6abe24160306b8d
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Tue, 08 Jun 2021 21:36:18 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /etc/nginx/conf.d from config (ro)
      /usr/share/nginx/html from html (ro)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-bwq6x (ro)
```

10. 컨테이너에 볼륨 마운트되었는지 확인해보자. 볼륨마운트가 되면 web-server 컨테이너는 nginx-config.conf 파일로 동작되고 있는것이다.

```
T1$ kubectl exec -it genid-volume-cm -c web-server  -- /bin/bash
    /# ls /etc/nginx/conf.d/
    interval  nginx-config.conf
    /# cat /etc/nginx/conf.d/interval
    /# cat /etc/nginx/conf.d/nginx-config.conf
    server {
        listen    80;
        server_name  www.example.com;

        gzip on;
        gzip_types text/plain application/xml;

        location / {
            root   /usr/share/nginx/html;
            index  index.html index.htm;
        }
    }
    /#
```

11. Volum mount를 이용해 데이터를 전달할 경우 ConfigMap에서 변경한 내용이 컨테이너 재시작 없이도 적용된다. Configmap에서 interval 값을 수정하고 변경사항이 적용되었는지 확인해보자.

```
T2$ kubectl edit configmaps bespin-config
apiVersion: v1
data:
  interval: "10"
  nginx-config.conf: |
    server {
        listen    80;
        server_name  www.example.com;

        gzip off;
        gzip_types text/plain application/xml;

        location / {
            root   /usr/share/nginx/html;
            index  index.html index.htm;
        }
    }
```

1분정도 기다린 후 변경상태가 적용되었는지 확인해보자.

```
    T1 /# cat /etc/nginx/conf.d/interval
    10
    T1 /# cat /etc/nginx/conf.d/nginx-config.conf
    server {
```

```
        listen    80;
        server_name  www.example.com;

        gzip off;
            ...
    /# exit
```

12. 지금까지 진행한 실습환경을 정리한다.
$ kubectl delete pod --all
$ kubectl delete configmaps --all

Kubernetes ConfigMap 사용하기 | LAB

- 컨테이너 볼륨마운트시 items를 추가하면 ConfigMap의 특정 key만 전달할수 있다.
  - genid-volume-cm.yaml 을 수정하여 아래의 items: 추가 manifest 를 설정해서 ConfigMap의 nginx-config.conf 키만 마운트되도록 설정하시오.

```
volumes:
 - name: html
   emptyDir: {}
 - name: config
   configMap:
     name: bespin-config
     items:
     - key: nginx-config.conf
       path: nginx-config.conf
```

  - 실행결과

```
$ kubectl exec -it genid-volume-cm -c web-server -- ls /etc/nginx/conf.d/
nginx-config.conf
```


ConfigMap volume parameters

```
$ cat genid-volume-cm.yaml
apiVersion: v1
kind: Pod
metadata:
  name: genid-volume-cm
spec:
  containers:
  - image: smlinux/genid:env
    env:
    - name: INTERVAL
      valueFrom:
        configMapKeyRef:
          name: bespin-config
          key: interval
    name: fakeid-generator
    volumeMounts:
    - name: html
      mountPath: /webdata
  - image: nginx:1.14
    name: web-server
    ports:
    - containerPort: 80
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    - name: config
      mountPath: /etc/nginx/conf.d
      readOnly: true
  volumes:
  - name: html
    emptyDir: {}
  - name: config
    configMap:
      name: bespin-config
      items:
      - key: nginx-config.conf
        path: nginx-config.conf
```

ConfigMap, Configmap-Volume 적용 실습

- 네임스페이스 moon에 준비된 web-moon.yaml을 이용해 web-moon이라는 nginx를 배포하려고 합니다. 그런데, 팀원 중 한명이 구성을 시작했지만 완료하지는 못한 상태인 것 같습니다.
- web-moon을 배포하려면 index.html 이름으로 web-moon.html 파일의 내용을 포함하는 configmap-web-moon-html이라는 ConfigMap을 만들어야합니다.
- 최종 실행 결과는 nginx:alpine Pod에서 curl을 사용하여 웹페이지를 확인하면 됩니다.

ConfigMap 적용 LAB

```
$ cat web-moon.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-moon
  namespace: moon
spec:
  replicas: 5
  selector:
    matchLabels:
      id: web-moon
  template:
    metadata:
      labels:
        id: web-moon
    spec:
      containers:
      - image: nginx:1.17.3-alpine
        name: nginx
        ports:
        - containerPort: 80
          protocol: TCP
        volumeMounts:
        - mountPath: /usr/share/nginx/html
          name: html-volume
      volumes:
      - configMap:
          defaultMode: 420
          name: configmap-web-moon-html
        name: html-volume

$ cat  web-moon.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Web Moon Webpage</title>
</head>
<body>
This is some great content.
</body>
</html>
```

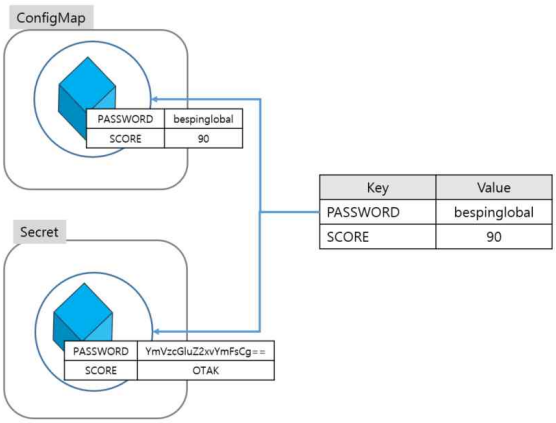
참고링크:
https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#create-a-configmap
https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#define-container-environment-variables-using-configmap-data
https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#populate-a-volume-with-data-stored-in-a-configmap

Secret 사용하기

Kubernetes Secret 사용하기 | Secret의 이해

- ConfigMap과 비슷
- 컨테이너가 사용하는 password, auth token, ssh key와 같은 중요한 정보를 저장하고 민감한 구성정보를 한 곳에 모아서 관리
- 민감하지 않은 일반 설정파일은 configMap을 사용하고 민감한 데이터는 secret을 사용
- Secret 데이터 전달 방법
  - Command-line Argument
  - Environment Variable
  - Volume Mount
- built-in secret : ServiceAccount

  $ kubectl get secrets
  default-token-bwq6x kubernetes.io/service-account-token 3 5d15h



Secret

Secrets는 ConfigMap과 동일하게 Pod 내부에 데이터를 전달하기 위해 사용한다. 다른점은 인증서나 노출하기 불편한 정보를 base64로 인코딩하여 보존하고, Pod 내부에서는 원래의 값을 전달한다.
노드 자체에서 secret은 항상 메모리에 저장하고 스토리지에는 기록되지 않는다. 쿠버네티스 1.7버전부터 etcd는 secret을 암호화 한 형태로 저장하여 시스템을 안전하게 만든다.

애플리케이션 운영 시 configMap과 Secret를 함께 사용할 수 있다.
- 민감하지 않는 일반 설정 데이터는 configMap을 사용
- 민감한 데이터를 저장할 때 secret을 사용

설정파일에 중요한 데이터와 중요하지 않은 데이터가 모두 포함돼 있으면 파일을 secret으로 저장하는 것이 좋다.

- **kubernetes 가 사용하는 secret**

kubernetes 시스템은 service account token을 secret으로 보존하고, 그 정보를 Pod 실행시 전달한다.

```
$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-bwq6x   kubernetes.io/service-account-token   3      5d15h
```

kubectl describe 명령으로 좀 더 자세히 보자.

```
$ kubectl describe secrets default-token-bwq6x
Name:         default-token-bwq6x
Namespace:    default
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: default
              kubernetes.io/service-account.uid: 98748e22-1f87-4e6f-a379-3fb2518273e0

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1066 bytes
namespace:  7 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6I....생략
```

Kubernetes Secret 사용하기 | Secret 생성하기

- SECRET 생성하기

```
kubectl create secret <Available Commands> name [flags] [options]
docker-registry     Create a secret for use with a Docker registry
generic             Create a secret from a local file, directory or literal value
tls                 Create a TLS secret
```

$ kubectl create secret generic bespin-secret --from-literal=interval=2 --from-file=./genid-web-config/

$ kubectl describe secret bespin-secret

type	의미
Opaque	임의의 사용자 정의 데이터
kubernetes.io/service-account-token	서비스 어카운트 토큰
kubernetes.io/dockercfg	직렬화 된(serialized) ~/.dockercfg 파일
kubernetes.io/dockerconfigjson	직렬화 된 ~/.docker/config.json 파일
kubernetes.io/basic-auth	기본 인증을 위한 자격 증명(credential)
kubernetes.io/ssh-auth	SSH를 위한 자격 증명
kubernetes.io/tls	TLS 클라이언트나 서버를 위한 데이터
bootstrap.kubernetes.io/token	부트스트랩 토큰 데이터

Secret 생성

- **명령어를 이용해서 secret 만들기**

$ kubectl create secret generic genid-secret --from-literal=interval=2
$ kubectl get secrets
$ kubectl describe secrets genid-secret

- **데이터파일을 secret으로 저장하기**

$ echo -n '1f2d1e2e67df' > ./password.txt

$ kubectl create secret generic db-user-pass --from-literal=username=admin --from-file=password=./password.txt
$ kubectl get secrets
$ kubectl describe secrets db-user-pass

- **바이너리 파일(인증서)을 secret으로 넣기**

Secret의 항목은 평문 텍스트뿐 아니라 바이너리 데이터도 포함할 수 있다. Base64 인코딩을 사용하면 바이너리 데이터를 일반 텍스트 형식인 Yaml, json에 포함시킬 수 있다. 단, secret의 최대크기는 1MB로 제한된다.
다음과 같이 인증서를 생성하고, 인증서를 secret으로 저장할수 있다.

$ openssl genrsa -out https.key 2048
$ openssl req -new -x509 -key https.key -out https.cert -days 3650 -subj /CN=www.bespinglobal.com

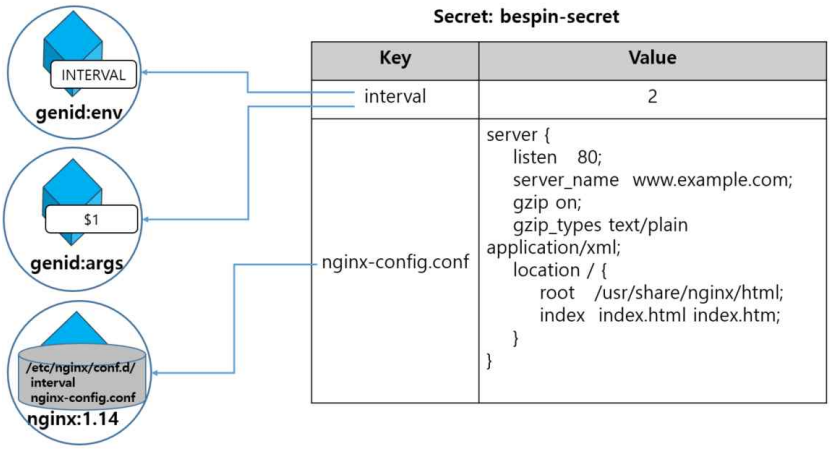
$ **kubectl create secret generic bespin-cert --from-file=https.key --from-file=https.cert**
$ kubectl describe secrets bespin-cert

- **secret 제거**

$ kubectl delete secrets bespin-cert
$ kubectl delete secrets bespin-secret
$ kubectl delete secrets db-user-pass
$ kubectl delete secrets genid-secret

Kubernetes Secret 사용하기 | 애플리케이션 배포(using secret)

- 정의된 Secret을 Pod의 Container에 전달하는 방법
  - 환경변수로 전달
  - Command-line Argument로 전달
  - Volume에 ConfigMap을 사용하여 컨테이너 디렉토리에 Mount



Secret을 이용해 컨테이너에 데이터 전달

- **환경변수로 전달하기**

```
spec:
  containers:
  - image: conainer_name
    env:
    - name: VARNAME
      valueFrom:
        secretKeyRef:
          name: secret_name
          key: keyname
```

- **Command-line Argument로 전달하기**

```
spec:
  containers:
  - image: container_image
    env:
    - name: VARNAME
      valueFrom:
        secretKeyRef:
          name: secret_name
          key: keyname
      args: ["$VARNAME"]
```

- **Volume에 ConfigMap 엔트리로 사용하여 전달하기**

```
  - image: container_image
    name: container-name
    volumeMounts:
    - name: some_name
      mountPath: /mount_point
      readOnly: true
    ...
  volumes:
  - name: somename
    secret:
      secretName: secret_name
```

EXAMPLE
secret을 생성하여 애플리케이션이 필요로하는 데이터를 저장하고, 배포되는 애플리케이션에게 전달한다.

1. secret 생성

```
$ kubectl create secret generic bespin-secret --from-literal=interval=2 --from-file=./genid-web-config/
$ kubectl get secret
$ kubectl describe secrets bespin-secret
$ kubectl get secrets bespin-secret -o yaml
apiVersion: v1
data:
  interval: Mg==
  nginx-config.conf:
   c2VydmVyIHsKICAgIGxpc3RlbiAgIDgwOwogICAgc2VydmVyX25hbWUgIHd3dy5leGFtcGxlLmNvbTsKCiAgICBnemlwI
   G9uOwogICAgZ3ppcF90eXBlcyB0ZXh0L3BsYWluIGFwcGxpY2F0aW9uL3htbDsKCiAgICBsb2NhdGlvbiAvIHsKICAgICA
   gICByb290ICAgL3Vzci9zaGFyZS9uZ2lueC9odG1sOwogICAgICAgIGluZGV4ICBpbmRleC5odG1sIGluZGV4Lmh0bTsKI
   CAgIH0KfQo=
kind: Secret
metadata:
...
```

2. 앞에서 사용했던 genid-env.yaml 파일을 수정하여 secret의 interval의 값을 환경변수 INTERVAL로 사용하도록 구성해보자. bespin-secret key를 컨테이너에서 사용할 수 있도록 key를 전달하자

```
$ cp genid-env-secret.yaml genid-env-secret.yaml
$ vim genid-env-secret.yaml
apiVersion: v1
kind: Pod
metadata:
  name: genid-env-secret
spec:
  containers:
  - image: smlinux/genid:env
    env:
    - name: INTERVAL
      valueFrom:
        secretKeyRef:
          name: bespin-secret
          key: interval
    name: fakeid-generator
    ...
```

3. 수정한 genid-env-secret.yaml 파일을 동작시키고, secret에 정의된 interval 변수가 환경변수 INTERVAL로 적용되었는지 확인해보자.

```
$ kubectl apply -f genid-env-secret.yaml
$ kubectl get pods -o wide
genid-env-secret   2/2     Running   0          47s     192.168.X.X   ip-192-168

NODE$ curl 192.168.X.X
2초마다 content가 변경되는가?
```

4. secret bespin-secret의 interval 값을 arguments로 전달해보자.

```
$ cp genid-args-cm.yaml genid-args-secret.yaml
$ vi genid-args-secret.yaml
...
  name: genid-args-secret
spec:
  containers:
  - image: smlinux/genid:args
    env:
    - name: INTERVAL
      valueFrom:
        secretKeyRef:
          name: bespin-secret
          key: interval
    args: ["$INTERVAL"]
    name: fakeid-generator
   ...
```

5. 볼륨 마운트를 통해 secret 데이터를 전달하는 방법을 확인해보자.
genid-web-config 디렉토리에 저장된 nginx-config.conf, interval 파일을 web-server pod에 전달하여 해당 설정파일로 서비스가 동작되도록 구현한다.

```
$ kubectl describe secrets bespin-secret
```

```
...
Data
====
interval:            1 bytes
nginx-config.conf:  218 bytes

$ cp genid-env-secret.yaml genid-volume-secret.yaml
$ vi genid-volume-secret.yaml
apiVersion: v1
kind: Pod
metadata:
  name: genid-volume-secret
spec:
...
  - image: nginx:1.14
    name: web-server
    ports:
    - containerPort: 80
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    - name: config
      mountPath: /etc/nginx/conf.d
      readOnly: true
  volumes:
  - name: html
    emptyDir: {}
  - name: config
    secret:
      secretName: bespin-secret

$ kubectl apply  -f genid-volume-secret.yaml
$ kubectl describe pod genid-volume-secret
```

6. 컨테이너에 볼륨 마운트 되었는지 확인해보자. 볼륨 마운트가 되면 web-server 컨테이너는 nginx-config.conf 파일로 동작되고 있는 것이다.

```
T1$ kubectl exec -it genid-volume-secret -c web-server  -- /bin/bash
/# ls /etc/nginx/conf.d/
interval  nginx-config.conf
/# cat /etc/nginx/conf.d/interval
/# cat /etc/nginx/conf.d/nginx-config.conf
/# exit
```

7. 지금까지 진행한 실습환경을 정리한다.

```
$ kubectl delete pod --all
$ kubectl delete secrets bespin-secret
```

Kubernetes Secret 사용하기 | LAB

- 컨테이너 볼륨마운트시 items를 추가하면 Secret의 특정 key만 전달할수 있다.
  - genid-volume-secret.yaml 을 수정하여 아래의 items: 추가 manifest 를 설정해서 secret의 nginx-config.conf 키만 마운트되도록 설정하시오.

```
volumes:
- name: html
  emptyDir: {}
- name: config
  secret:
    secretName: bespin-secret
    items:
    - key: nginx-config.conf
      path: nginx-config.conf
```

  - 실행결과

```
$ kubectl exec -it genid-volume-secret -c web-server -- ls /etc/nginx/conf.d/
nginx-config.conf
```


ConfigMap volume parameters

```
$ cat genid-volume-cm.yaml
apiVersion: v1
kind: Pod
metadata:
  name: genid-volume-secret
spec:
  containers:
  - image: smlinux/genid:env
    env:
    - name: INTERVAL
      valueFrom:
        secretKeyRef:
          name: bespin-secret
          key: interval
    name: fakeid-generator
    volumeMounts:
    - name: html
      mountPath: /webdata
  - image: nginx:1.14
    name: web-server
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    - name: config
      mountPath: /etc/nginx/conf.d
      readOnly: true
    ports:
    - containerPort: 80
  volumes:
  - name: html
    emptyDir: {}
  - name: config
    secret:
      secretName: bespin-secret
      items:
      - key: nginx-config.conf
        path: nginx-config.conf
```

Secret, Secret-Volume, Secret-Env 적용 실습

- moon 네임스페이스에 **user=test** 과 **pass=pwd**값을 가지는 **secret1** 이라는 Secret을 생성하시오.
- Secret은 **secret1**은 moon 네임스페이스에서 동작 시킬 Pod **secret-handler** 에 환경변수 **SECRET1_USER** 와 **SECRET1_PASS**로 전달되어야 한다. secret이 적용되기 전의 secret-hander.yaml 파일은 아래와 같이 이미 준비되어 있다. 이 secret-handler.yaml 파일을 수정하여 **secret** 을 환경변수로 적용하시오.
- 또 하나의 **secret2.yaml** 파일이 준비되어 있습니다. 이 **secret2**가 moon namespace에서 동작되도록 실행하시오. **secret-handler-new.yaml** 파일을 수정해서 앞서 생성한 **secret2**의 데이터를 **/tmp/secret2** 디렉토리에 볼륨 마운트하시오.

Secret을 Pod에 적용하기

```
# cat secret-handler.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    id: secret-handler
    uuid: 1428721e-8d1c-4c09-b5d6-afd79200c56a
    red_ident: 9cf7a7c0-fdb2-4c35-9c13-c2a0bb52b4a9
    type: automatic
  name: secret-handler
  namespace: moon
spec:
  volumes:
  - name: cache-volume1
    emptyDir: {}
  - name: cache-volume2
    emptyDir: {}
  - name: cache-volume3
    emptyDir: {}
  containers:
  - name: secret-handler
    image: bash:5.0.11
    args: ['bash', '-c', 'sleep 2d']
    volumeMounts:
    - mountPath: /cache1
      name: cache-volume1
    - mountPath: /cache2
      name: cache-volume2
    - mountPath: /cache3
      name: cache-volume3
    env:
    - name: SECRET_KEY_1
      value: ">8$kH#kj..i8}HImQd{"
    - name: SECRET_KEY_2
      value: "IO=a4L/XkRdvN8jM=Y+"
    - name: SECRET_KEY_3
      value: "-7PA0_Z]>{pwa43r)__"
```

 # cat secret2.yaml

```
apiVersion: v1
kind: Secret
metadata:
```

```
  name: secret2
data:
  key: MTIzNDU2Nzg=
```

참고링크:
https://kubernetes.io/docs/concepts/configuration/secret/#using-secrets-as-environment-variables
https://kubernetes.io/docs/concepts/configuration/secret/#using-secrets

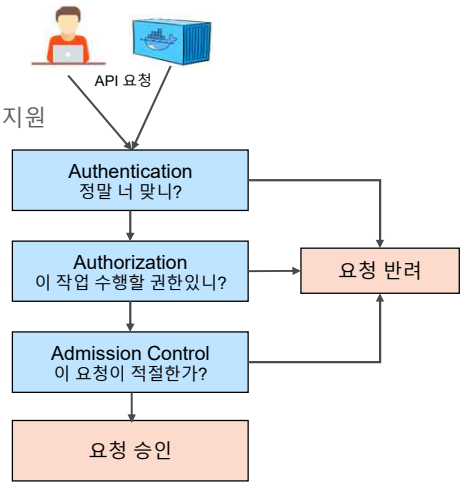
지금까지 배운 내용을 기준으로

- 컨테이너 데이터 패싱
  - 컨테이너 데이터 패싱 흐름을 이해해야합니다.
- ConfigMap 사용하기
  - ConfigMap을 생성할수 있어야합니다.
  - ConfigMap에 저장된 value를 컨테이너에 전달할수 있어야합니다.
  - ConfigMap 볼륨 마운트 할수 있어야합니다.
- Secret 사용하기
  - Secret을 생성할수 있어야합니다.
  - Secret에 저장된 value를 컨테이너에 전달할수 있어야합니다.
  - Secret 볼륨 마운트 할수 있어야합니다.

API 인증

API 접근제어

- 인증요청
  - User Account(User), Service Account(application)
- Authentication
  - user 또는 application이 API에 접근을 허가 받는 과정
  - 인증방식 : 클라이언트 인증서, 베어러 토큰(bearer token), HTTP 기본인증 지원
- Authorization
  - RBAC 모델기반
  - 요청 ID에 적절한 role이 있는지 확인
- Admission Control
  - 요청이 올바른 형식인지 판별
  - 요청이 처리되기 전에 수정사항을 잠재적으로 적용



https://kubernetes.io/ko/docs/concepts/security/controlling-access/

API 접근제어

API 인증 lifecycle

Authentication

사용자가 쿠버네티스의 API에 접근하기 위해서는 인증(Authentication)을 거쳐야 한다. API server는 테스트 목적으로 로컬호스트의 8080포트에 http 포트를 띄우고, 외부에서 접근할 수 있는 기본 포트는 6443으로 TLS인증이 적용되어 있다. 일반적인 https인증은 접근하는 클라이언트에서는 인증서가 필요 없지만, 쿠버네티스의 API server에 열려 있는 포트에 접근하기 해서는 APIserver의 인증서를 가지고 접근해야 통신이 가능하다.

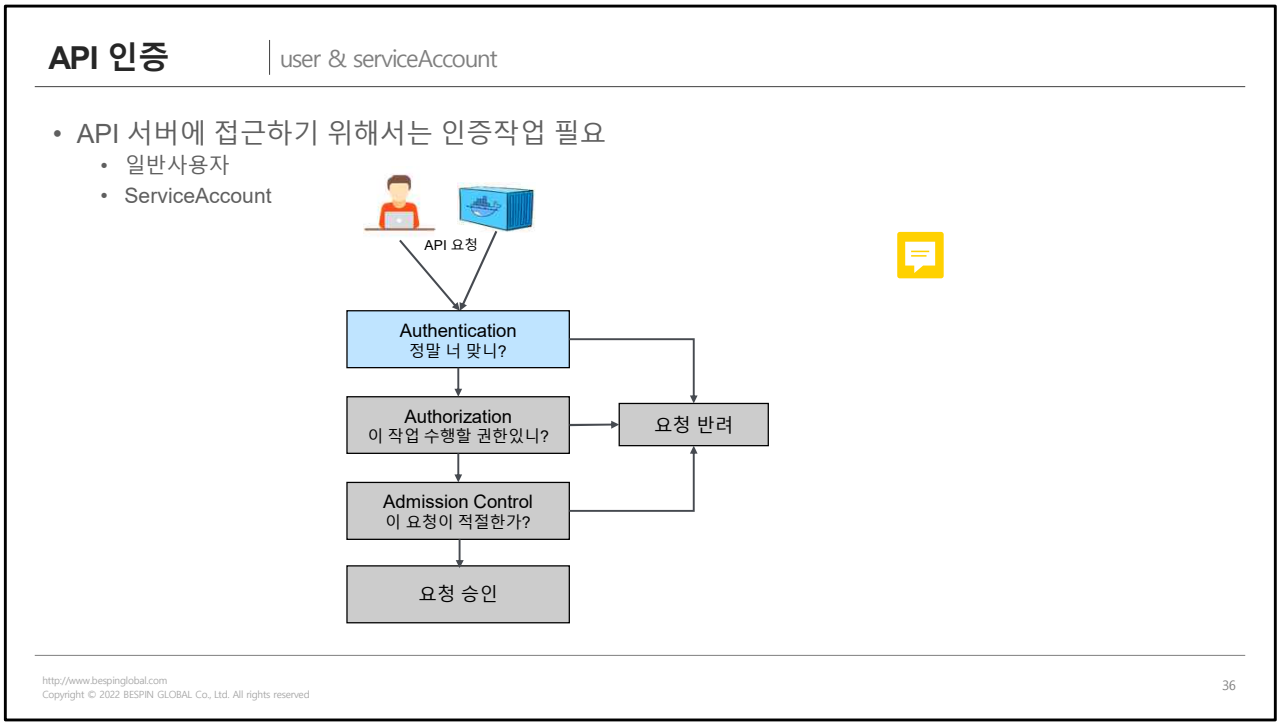
Authorization

API 서버가 요청에 대한 인증을 판별하면 인가(Authorization)로 이동한다. 쿠버네티스에 대한 모든 요청은 전통적인 RBAC 모델을 따른다. 요청에 Access 하려면 ID에 요청과 관련된 적절한 role이 있어야한다. API 서버는 요청을 처리할 때 요청과 연관된 ID가 요청의 동사와 요청의 HTTP 경로 조합에 Access할 수 있는지 결정한다. 요청의 ID에 적절한 Role이 있으면 계속 진행되고, 아니면 HTTP 403 응답이 반환된다.

Admission Control

요청이 인증되고 인가되면 승인제어(Admission Control)로 이동한다. 인증 및 RBAC는 요청이 허용되는지를 확인하며, 이는 요청의 HTTP 속성(헤더, 방법, 경로)을 기반으로 한다.

승인제어는 요청이 올바른 형식인지 판별하고, 요청이 처리되기 전에 수정사항을 잠재적으로 적용한다. Admission Control은 플러그 가능한 인터페이스를 정의한다.



API 인증

쿠버네티스에서 인증을 요청하기 위한 사용자는 일반적인 사용자(Human user)와 서비스어카운트(service account) 두 가지 개념을 가진다.

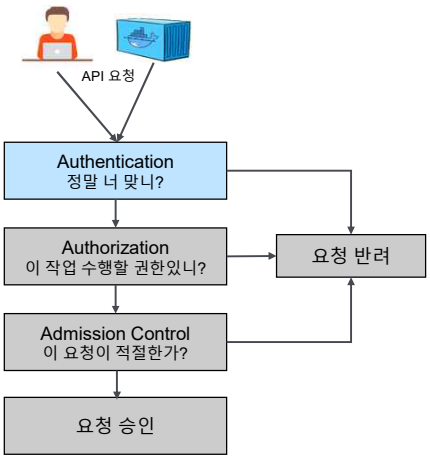
- 일반사용자(Human User) : 클러스터 외부에서 쿠버네티스를 조작하는 사용자로, 다양한 방법으로 인증을 거친다.
- 서비스계정(serviceaccoount) : 쿠버네티스 내부적으로 관리되며 Pod가 쿠버네티스 API를 다룰 때 사용하는 계정이다.

일반 사용자(Human user)는 개발자 및 운영 실무자가 쿠버네티스를 조작하기 위해 사용한다. 쿠버네티스 클러스터 외부로부터 들어오는 접근을 관리하기 위한 사용자다. 일반 사용자를 분류하는 그룹 개념도 있어서 이 그룹 단위로 권한을 부여할 수도 있다.

서비스 계정(service account)은 쿠버네티스 리소스의 일종이다. 쿠버네티스 클러스터 내부에서 권한을 관리하는 역할을 한다. 서비스 계정과 연결된 Pod는 주어진 권한에 따라 쿠버네티스 리소스를 다룰 수 있다.
서비스 계정 및 일반 사용자의 권한은 RBAC(Role-Based Access Control)이라는 메커니즘을 통해 제어된다. RBAC는 롤에 따라 리소스에 대한 권한을 제어하는 기능이자 개념이다. RBAC를 적절히 사용해 쿠버네티스 리소스의 보안을 확보할 수 있다.

API 인증 | K8S User

- 사용자 및 그룹
  - 클러스터 외부에서 쿠버네티스를 조작하는 사용자(root, guru)
  - kubernetes-admin 권한
  - 다양한 방법의 인증을 거친다. default는 인증서를 이용해 인증 받음.
    $ cat ~/.kube/config
    $ kubectl config view



API 인증

쿠버네티스에서 인증을 요청하기 위한 사용자는 일반적인 사용자(Human user)와 서비스어카운트(service account) 두 가지 개념을 가진다.
- 일반사용자(Human User) : 클러스터 외부에서 쿠버네티스를 조작하는 사용자로, 다양한 방법으로 인증을 거친다.
- 서비스계정(serviceaccoount) : 쿠버네티스 내부적으로 관리되며 Pod가 쿠버네티스 API를 다룰 때 사용하는 계정이다.

일반 사용자(Human user)는 개발자 및 운영 실무자가 쿠버네티스를 조작하기 위해 사용한다. 쿠버네티스 클러스터 외부로부터 들어오는 접근을 관리하기 위한 사용자다. 일반 사용자를 분류하는 그룹 개념도 있어서 이 그룹 단위로 권한을 부여할 수도 있다.

서비스 계정(service account)은 쿠버네티스 리소스의 일종이다. 쿠버네티스 클러스터 내부에서 권한을 관리하는 역할을 한다. 서비스 계정과 연결된 Pod는 주어진 권한에 따라 쿠버네티스 리소스를 다룰 수 있다.
서비스 계정 및 일반 사용자의 권한은 RBAC(Role-Based Access Control)이라는 메커니즘을 통해 제어된다. RBAC는 롤에 따라 리소스에 대한 권한을 제어하는 기능이자 개념이다. RBAC를 적절히 사용해 쿠버네티스 리소스의 보안을 확보할 수 있다.

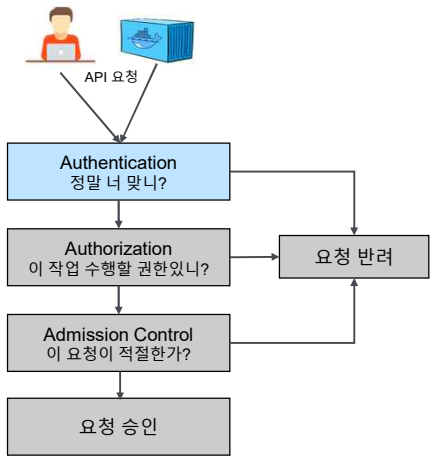
일반 사용자 및 일반 사용자 그룹
쿠버네티스는 role이 연결될 일반 사용자 또는 일반 사용자 그룹을 지정할 수 있다. 일반 사용자의 인증은 다음과 같은 방법으로 할 수 있다.
• 서비스 계정 토큰 방식
• 정적 토큰 방식
• 패스워드 파일 방식 : 사용자와 패스워드를 파일에 정의하는 방식
• X509 클라이언트 인증서 방식: 클라이언트 인증서를 이용한 인증 방식
• OpenID 방식 : OpenID 프로바이더(구글 등)을 이용한 인증방식

일반 사용자 인증 : 인증서를 이용한 인증
쿠버네티스는 apiserver와 통신하기 위해서 사용하는 기본 인증 방법으로 TLS(Transport Layer Security)인증을 사용한다.
apiserver쪽에는 있는 인증서와 매치되는 클라이언트 인증서를 가지고 접속을 시도하면 된다. 그동안 kubectl을 설치한 후에 별다른 문제없이 명령어들을 사용해 왔는데, 이것은 이미 kubectl설정에 이 인증서가 들어 있기 때문에 가능한 것이었다.

$ cat ~/.kube/config
```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0t....
    server: https://X.X.X.X:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
...
```

API 인증 | Service Account

- Service Account
  - 쿠버네티스 내부적으로 관리되는 Account
  - Pod가 Kubernetes API를 다룰 때 사용하는 Account
    $ kubectl get serviceaccounts
    NAME SECRETS AGE
    default 1 57d
- default Account 외 다른 Account 생성 가능
- 클러스터 접근을 제한하기 위해 생성
  kubectl create serviceaccount <new-serviceaccount>



Service Account 생성 및 사용

ServiceAccount
쿠버네티스가 직접 관리하는 사용자 계정.
모든 namespace에는 자체 디폴트 serviceaccount가 있으나 필요한 경우 추가할 수 있다.
serviceaccount를 생성하는 이유는 보안 때문이다. 클러스터 메타 데이터를 읽을 필요가 없는 포드는 클러스터에 배포된 리소스를 검색하거나 수정할 수 없는 제한된 계정에서 실행해야한다.

리소스 메타데이터를 검색해야 하는 포드는 해당 객체의 메타 데이터를 읽을 수 있는 ServiceAccount로 실행해야하고, 또한 해당 객체를 수정해야 하는 포드는 API 객체를 수정할 수 있는 자체 ServiceAccount에서 실행되어야 한다.

ServiceAccount : 토큰을 이용한 인증

ServiceAccount는 Pod, Secret, Configmap등과 같은 리소스이며 하나의 네임스페이스로 범위가 지정된다. API Server가 Pod의 요청을 수신할 때마다 자체 네임스페이스만 자격증명이 적용되기 때문에 네임스페이스의 격리가 보장된다.

쿠버네티스는 Pod를 대신하여 ServiceAccount를 관리한다. 쿠버네티스가 Pod를 인스턴스할 때마다 해당 Pod에 서비스 계정을 할당한다. ServiceAccount는 API 서버와 상호작용을 할 때 모든 Pod 프로세스를 식별한다. 각각의 서비스 계정은 일련의 자격증명의 집합을 가지고 있으며 Secret Volume에 탑재되어 있다. 각각의 네임스페이스는 default라는 기본 ServiceAccount가 있다.

Pod를 만들 때 다른 ServiceAccount를 지정하지 않으면 default 서비스 계정이 할당된다.

$ kubectl get serviceaccounts

```
NAME SECRETS AGE
default 1 6d3h
```

각 Pod는 정확히 하나의 ServiceAccount와 연관되어 있으나 여러 pod에서 같은 Service Account를 사용할 수 있다. Pod는 동일 네임스페이스의 service account만 사용할 수 있다.

특별히 지정하지 않으면 기본으로 Pod는 자신이 속한 namespace의 default 서비스어카운트를 사용한다. Pod에 각기 다른 ServiceAccount를 할당하면 각 Pod에 접근할 수 있는 리소스를 제어할 수 있다.

```
# kubectl get pod testpod -o yaml
...
  securityContext: {}
```

```
  serviceAccount: default
  serviceAccountName: default
```

default 사용자의 account token이 있다. 이것을 인증에 사용하는 것이다.

```
$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-57gkx   kubernetes.io/service-account-token   3      6d3h
```

ServiceAccount 생성

```
# kubectl create serviceaccount web-account
# kubectl get  serviceaccounts
```

생성된 hpaccount serviceaccount의 secret 정보를 확인해보자.

```
# kubectl get secrets
```

ServiceAccount가 사용하는 Secret Token을 확인

```
# kubectl describe serviceaccounts web-account
```

serviceaccount가 생성되고, secret이 생성되어 foo ServiceAccount와 연결되었다. secret에는 인증서, 네임스페이스, 토큰이 포함되어 있다.

```
# kubectl describe secrets web-account-token-XXX
Name:          web-account-token-zfztj
Namespace:     default
Labels:        <none>
Annotations:  kubernetes.io/service-account.name: web-account
              kubernetes.io/service-account.uid: 6574850f-725e-44e7-a03e-813207ff13ff

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:      1066 bytes
namespace:  7 bytes
token:       eyJhbGciO...
```

사용자 지정 ServiceAccount 할당

앞서 생성한 web-account ServiceAccount를 Pod에 할당해본다. 해당 Pod에 ServiceAccount를 default가 아닌 web-account로 설정하면 web-account 서비스어카운트로 API 서버와 통신 할 수 있다.

```
  smlinux/curl                 API의 REST 인터페이스를 탐색
  smlinux/kubectl-proxy  pod의 service account token을 사용해 API 서버로 인증하기 위해 사용
```

```
# cat curl-custom-sa.yaml
apiVersion: v1
kind: Pod
metadata:
  name: curl-custom-sa
spec:
  serviceAccountName: hpaccount
  containers:
  - name: main
    image: smlinux/curl
    command: ["sleep", "9999999"]
  - name: ambassador
    image: smlinux/kubectl-proxy:1.6.2

# kubectl create -f curl-custom-sa.yaml
# kubectl get pods
```

생성한 ServiceAccount의 토큰이 두 컨테이너에 마운트 되었는지 확인하려면 아래의 명령을 실행하여 토큰을 확인해보면 된다.

```
# kubectl exec  -it curl-custom-sa  -c main -- ls /var/run/secrets/kubernetes.io/serviceaccount -l
lrwxrwxrwx 1 root root 13 Jun 20 12:48 ca.crt -> ..data/ca.crt
lrwxrwxrwx 1 root root 16 Jun 20 12:48 namespace -> ..data/namespace
lrwxrwxrwx 1 root root 12 Jun 20 12:48 token -> ..data/token

# kubectl exec  -it curl-custom-sa  -c main ₩
    -- cat /var/run/secrets/kubernetes.io/serviceaccount/token
```

eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3N

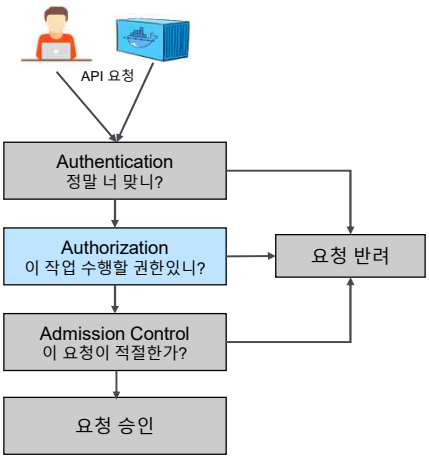
생성한 토큰으로 API 서버와 통신을 해보자. localhost:8001 번의 proxy를 통해서 들어간다.
kubectl exec -it curl-custom-sa -c main curl localhost:8001/api/v1/pods

권한관리

권한 관리

K8s의 권한 관리

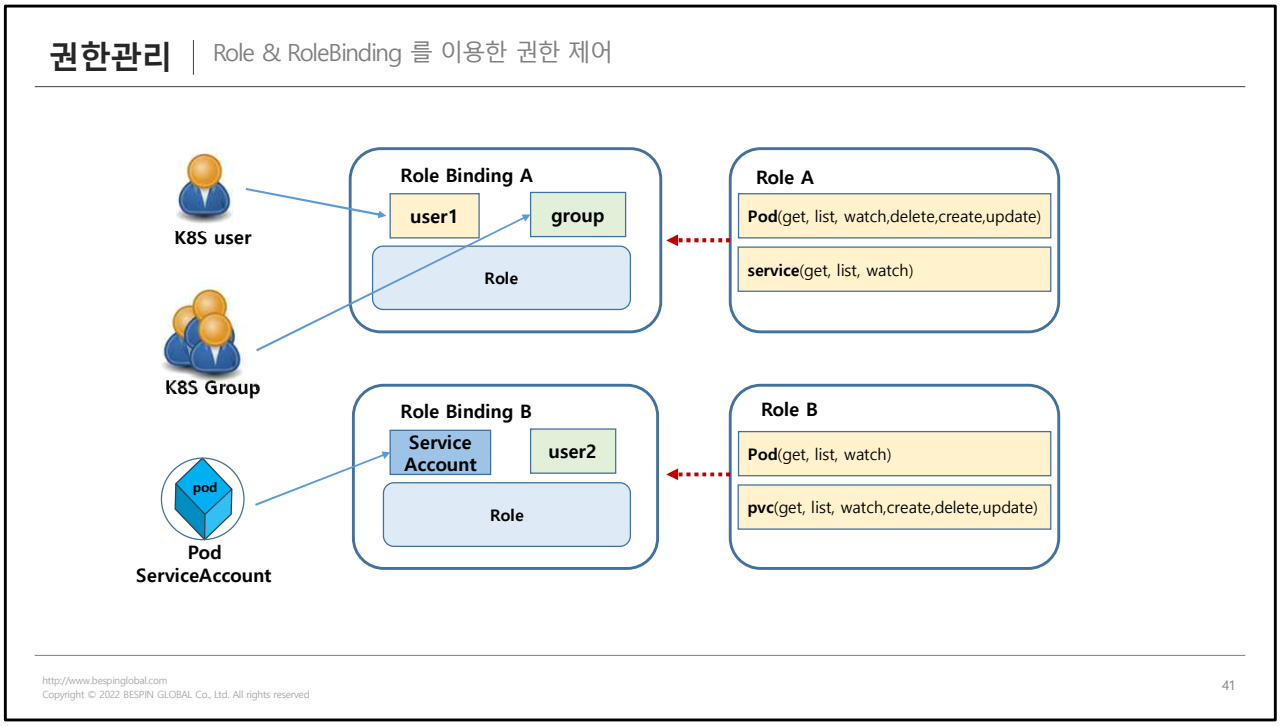
- 특정 유저나 ServiceAccount가 접근하려는 API에 접근 권한을 설정.
- 권한 있는 User만 접근하도록 허용
- 권한제어
- Role
  - 어떤 API를 이용할 수 있는지의 정의
  - 쿠버네티스의 사용권한을 정의
  - 지정된 네임스페이스에서만 유효
- RoleBinding
  - 사용자/그룹 또는 Service Account와 role을 연결



권한 관리

쿠버네티스 클러스터의 API에 접근하려면 접근 할 수 있는 사용자인지 인증을 거쳐야 한다.
인증 후에는 사용자가 접근하려는 API를 사용할 권한이 있는지 확인 과정을 거치게 되는데, 이 과정이 Authorization 이다.

클러스터 하나를 여러 사람이 사용할 때는 API나 Namespace 별로 권한을 구분해서 권한이 있는 곳에만 접근할 수 있도록 제한해야 한다.



Role과 RoleBinding

RBAC(Role-Based Access Control)
쿠버네티스는 역할(Role) 기반으로 쿠버네티스 리소스 사용 권한을 관리한다. 특정 사용자(User) 또는 Service Account와 역할(Role)을 연결하여 리소스 사용 권한이 부여된다.

Role과 RoleBinding
Role은 특정 API나 리소스(Pod, Deploy, Service 등)의 사용 권한(Create, Get, Edit 등)을 매니페스트 파일에 명시해둔 규칙의 집합으로 특정 네임스페이스에 대한 권한을 관리 한다.
Relebinding은 미리 정의된 Role과 특정 User/ServiceAccount를 묶어주어 특정 유저나 서비스 어카운트에서 Role에 명시된 규칙 기준으로 API 리소스를 사용할수 있도록한다.

Role 생성하기
default 네임스페이스의 Pod에 대한 get, watch, list 할수 있는 권한을 생성한다.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

- pod는 코어 API이기 때문에 apiGroups를 따로 지정하지 않는다. 만약 리소스가 job이라면 apiGroups에 "batch"를 지정해준다.
- resources에는 pods, deployments, services 등 사용할 API resource들을 명시한다.
- verbs에는 단어 그대로 나열된 API 리소스에 허용할 기능을 나열한다.

create	새로운 리소스 생성
get	개별 리소스 조회
list	여러 건의 리소스 조회
update	기존 리소스 내용 전체 업데이트
patch	기존 리소스 중 일부 내용 변경
delete	개별 리소스 삭제
deletecollection	여러 리소스 삭제

RoleBinding 생성하기

예1: default 네임스페이스에서 유저 jane 에게 앞서 정의한 pod-reader의 role을 할당하는 RoleBinding의 예제이다.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
```

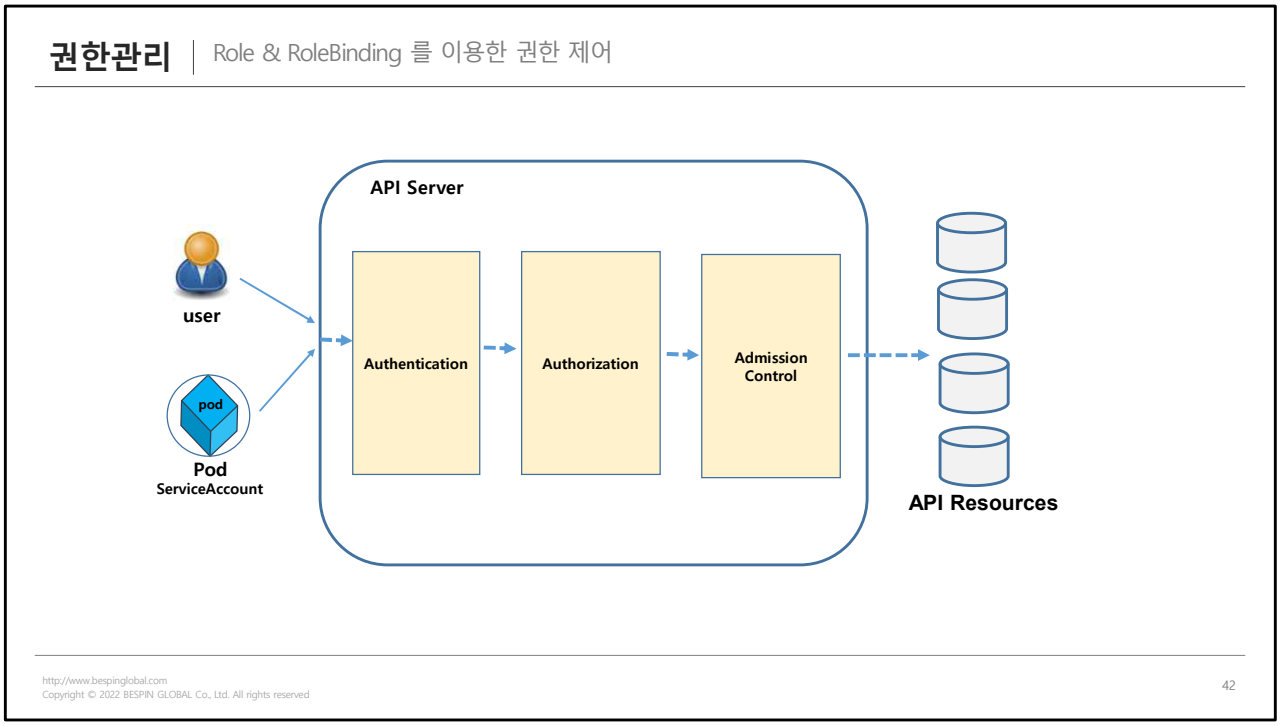
```
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

예2: default 네임스페이스에서 podman 서비스 어카운트에게 pod-reader의 role을 할당한다.

```
subjects:
- kind:
  name: podman
  apiGroup: rbac.authorization.k8s.io
```

- subjects에는 미리 정의된 role을 허용할 user나 ServiceAccount를 명시한다.
- roleRef에는 정의한 Role을 명세한다. apiGroup에는 rbac api를 사용하기 때문에 rbac.authorization.k8s.io로 설정한다.
- user jain(또는 serviceAccount podman)은 default 네임스페이스의 모든 Pod를 get, watch, list 할수 있다.

참고:
https://kubernetes.io/docs/reference/access-authn-authz/rbac/
https://kubernetes.io/ko/docs/reference/access-authn-authz/authorization/



RBAC을 이용한 권한관리

RBAC를 이용한 권한제어는 크게 두가지 요소로 구현된다.

- 어떤 쿠버네티스 API를 사용할 수 있는지가 정의된 role
- 이 role을 일반 사용자(그룹) 또는 service account와 연결해주는 Binding이다.

Role과 ClusterRole

RBAC를 이용한 권한제어 | Role

- Role
  - 어떤 API를 사용할 수 있는지 권한정의. 지정된 네임스페이스에서만 유효
- Role 예제 : default 네임스페이스의 Pod에 대한 get, watch, list 할 수 있는 권한

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  verbs: ["get", "watch", "list"]
```

verb	의미
create	새로운 리소스 생성
get	개별 리소스 조회
list	여러 건의 리소스 조회
update	기존 리소스 내용 전체 업데이트
patch	기존 리소스 중 일부 내용 변경
delete	개별 리소스 삭제
deletecollection	여러 리소스 삭제

- pod는 코어 API이기 때문에 apiGroups를 따로 지정하지 않는다. 만약 리소스가 job이라면 apiGroups에 "batch"를 지정
- resources에는 pods, deployments, services 등 사용할 API resource들을 명시한다.
- verbs에는 단어 그대로 나열된 API 리소스에 허용할 기능을 나열한다.

Role 생성

default 네임스페이스의 Pod에 대한 get, watch, list할수 있는 권한

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

RBAC를 이용한 권한제어 | RoleBinding

- RoleBinding
  - 사용자/그룹 또는 Service Account와 role을 연결
- RoleBinding 예제 : default 네임스페이스에서 유저 jane 에게 pod-reader의 role을 할당

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

→ 유저 jane 에게 앞서 정의한 pod-reader의 role을 할당

serviceAccount 적용 예:

```
subjects:
- kind: ServiceAccount
  name: podman
  apiGroup: rbac.authorization.k8s.io
```

→ 앞서 정의한 Role을 명세
apiGroup에는 rbac api를 사용하기 때문에 rbac.authorization.k8s.io로 설정

RoleBinding 생성

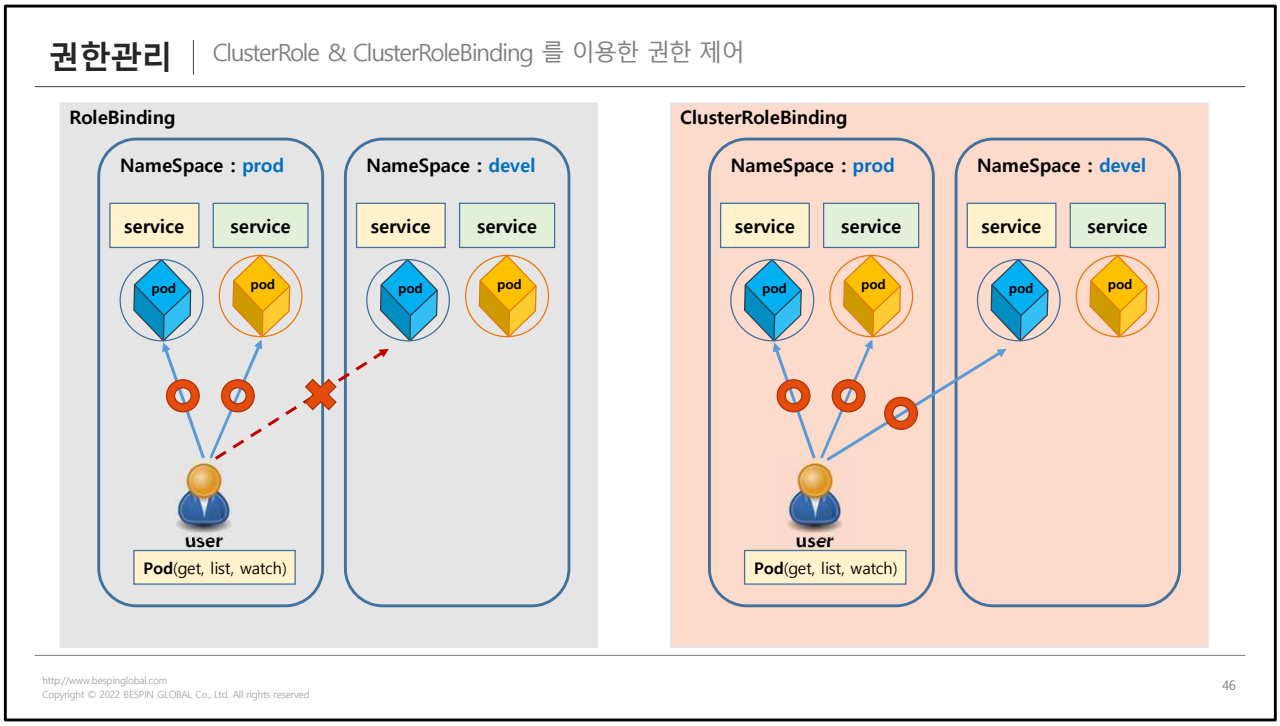
```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

예2: default 네임스페이스에서 podman 서비스 어카운트에게 pod-reader의 role을 할당한다.

```
subjects:
- kind:
  name: podman
  apiGroup: rbac.authorization.k8s.io
```

subjects에는 미리 정의된 role을 허용할 user나 ServiceAccount를 명시한다.
한다.
user jain(또는 serviceAccount podman)은 default 네임스페이스의 모든 Pod를 get, watch, list 할수 있다.

참고:
https://kubernetes.io/docs/reference/access-authn-authz/rbac/
https://kubernetes.io/ko/docs/reference/access-authn-authz/authorization/



ClusterRole과 ClusterRoleBinding

롤과 롤바인딩은 단일 네임스페이스의 리소스에 상주하고 적용한다.
클러스터롤과 클러스터롤 바인딩은 네임스페이스가 아닌 클러스터의 전체(네임스페이스에 없는 PV접근 등)에 접근할 수 있다.

참고:
https://kubernetes.io/docs/reference/access-authn-authz/rbac/
https://kubernetes.io/ko/docs/reference/access-authn-authz/authorization/

RBAC를 이용한 권한제어 | ClusterRole

- ClusterRole
  - 어떤 API를 사용할 수 있는지 권한 정의. 클러스터 전체(전체 네임스페이스)에서 유효
  - ClusterRole 예제 : 전체 네임스페이스의 Secret에 대한 get, watch, list 할 수 있는 권한

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


ClusterRole 생성

특정 네임스페이스 또는 모든 네임스페이스에서 secret API 리소스에 대해 get, watch, list를 할수 있다.

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

RBAC를 이용한 권한제어 | ClusterRoleBinding

- ClusterRoleBinding
  - 사용자/그룹 또는 Service Account와 role을 연결
  - ClusterRoleBinding 예제 : manager 그룹의 모든 사용자가 모든 네임스페이스의 secret을 읽을 수 있도록 구성

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

manager 그룹

ClusterRole

ClusterRoleBinding 생성

ClusterRoleBinding을 사용하면 "manager" 그룹의 모든 사용자가 모든 네임스페이스의 비밀을 읽을 수 있습니다.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

참고:
https://kubernetes.io/docs/reference/access-authn-authz/rbac/
https://kubernetes.io/ko/docs/reference/access-authn-authz/authorization/

Helm 이란

쿠버네티스 패키지 매니저

- 쿠버네티스 기반의 애플리케이션 운영을 지원하는 툴
- 2017년 Deis에 의해 개발.
- 2018년 6월 CNCF 재단의 정식 프로젝트로 승격

Char

- Helm 패키지를 설명하는 Chart.yaml
- Kubernetes manifest 파일을 가지는 Template 파일

Helm 이란?

쿠버네티스는 런타임 시 컨테이너를 구성하고 조정할 수 있는 다양한 방법을 제공한다. 하지만 여기에는 이미지 세트를 그룹화하는 고수준의 조직이 결여되어 있다.
helm(헬름)은 쿠버네티스 패키지 매니저로 쿠버네티스의 하위 프로젝트로 시작되었다가 2018년 6월에 CNCF재단의 정식 프로젝트로 승격되었다.
Helm은 Chart라는 파일형식으로 패키징하며, chart를 통해 애플리케이션 설치, 업그레이드, 롤백 관리를 간편하게 해준다.

헬름의 목적
헬름은 차트라고 하는 쿠버네티스 패키지를 관리하는 도구이다. 헬름으로 다음과 같은 것들을 할 수 있다.
스크래치(scratch)부터 새로운 차트 생성
차트 아카이브(tgz) 파일로 차트 패키징
차트가 저장되는 곳인 차트 리포지터리와 상호작용
쿠버네티스 클러스터에 차트 인스톨 및 언인스톨
헬름으로 설치된 차트들의 릴리스 주기 관리

헬름에는 다음과 같은 중요한 3가지 개념이 있다.
차트는 쿠버네티스 애플리케이션의 인스턴스를 생성하는 데에 필요한 정보의 꾸러미이다.
설정은 릴리스 가능한 객체를 생성하기 위해 패키징된 차트로 병합될 수 있는 설정 정보를 가진다.
릴리스는 차트의 구동 중 인스턴스이며, 특정 _설정_이 결합되어 있다.
(참고: https://helm.sh/ko/docs/topics/architecture/)

Helm 설치

Go로 구현되어 있으며 동일한 바이너리 실행파일은 클라이언트와 서버 역할을 수행할 수 있다.

- macOS : brew install helm
- Windows : choco install kubernetes-helm
- Linux(Snap 버전): sudo snap install helm --classic
- 스크립트로 설치:

$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3
$ chmod 700 get_helm.sh
$ ./get_helm.sh

Installation

$ curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 〉 get_helm.sh
$ chmod 700 get_helm.sh
$ PATH=$PATH:/usr/local/bin
$ ./get_helm.sh
Helm v3.8.2 is already latest

$ helm version
version.BuildInfo{Version:"v3.8.2", GitCommit:"6e3701edea09e5d55a8ca2aae03a68917630e91b", GitTreeState:"clean", GoVersion:"go1.17.5"}

$ source <(helm completion bash)
$ echo "source <(helm completion bash)" >> ~/.bashrc

Helm 사용하기

헬름 차트 리포지토리 초기화

$ helm repo add stable https://charts.helm.sh/stable

$ helm search repo stable

차트 검색 : helm search hub mysql

검색한 차트의 세부 내용 확인 : helm inspect stable/mariadb

패키지 설치 : helm install stable/mariadb

설치된 패키지 상태확인 : helm status

Helm 사용 doc: https://helm.sh/ko/docs/intro/using_helm/

helm 사용하기

차트를 설치하기 위해서, helm install 커맨드를 실행한다.
헬름은 차트를 설치하기 위한 몇 가지 방법들이 존재하는데, 가장 쉬운 방법은 공식적인 stable 차트들을 이용하는 것이다.

```
$ helm repo add stable https://charts.helm.sh/stable
"stable" has been added to your repositories

$ helm repo list
NAME            URL
stable          https://charts.helm.sh/stable
```