Config & Storage API

Config & Storage API

- 개요
  - ✓ 컨피그 & 스토리지 API 카테고리로 분류되는 리소스는 컨테이너에 대해 설정 파일, 패스워드 같은 기밀 정보 등을 추가하거나 영구 볼륨을 제공하기 위한 리소스
  - ✓ 내부에서 사용되는 것을 제외하고 사용자가 직접 사용하는 것으로는 세 가지 종류의 컨피그 & 스토리지 리소스가 있음
  - ✓ 종류
    - Secret
    - Config Map
    - Persistent Volume Claim(PVC)

환경 변수

- 개요
  - ✓ 쿠버네티스에서 개별 컨테이너의 설정 내용은 환경 변수나 파일이 저장되어 있는 영역을 마운트하여 전달하는 것이 일반적
  - ✓ 쿠버네티스에서 환경 변수를 전달할 때는 파드 템플릿에 env 또는 envFrom을 지정
  - ✓ 환경 변수에 포함시키는 정보
    - 정적 설정
    - 파드 정보
    - 컨테이너 정보
    - 시크릿 리소스 기밀 정보
    - 컨피그맵 리소스 설정값

환경 변수

- ❖ 정적 설정
  - ✓ 정적 설정은 spec.containers[].env에 정적인 값을 설정하는 것
    - sample-env.yaml 파일

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-env
  labels:
    app: sample-app
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    env:
    - name: MAX_CONNECTION
      value: "100"
    - name: TZ
      value: Asia/Seoul
```

환경 변수

- ❖ 정적 설정
  - ✓ 정적 설정은 spec.containers[].env에 정적인 값을 설정하는 것
    - Pod 생성: kubectl apply –f sample-env.yaml
    - 환경 변수 확인: kubectl exec -it sample-env -- env | grep MAX_CONNECTION

      MAX_CONNECTION=100

환경 변수

- 파드 정보
  - ✓ 어떤 노드에서 파드가 기동하고 있는지와 파드 자신의 ip 주소, 기동 시간 등 파드에 관한 정보는 fieldRef를 사용하면 참조할 수 있는데 확인 가능한 값은 kubectl get pods -o yaml 등으로 확인
  - ✓ 일부 내용을 보면, 등록한 매니페스트 정보와 별도로 파드의 IP 주소나 호스트 정보와 같은 여러 정보가 추가되어 있는 것을 확인할 수 있음
  - ✓ 기동 중인 파드 정보 확인: kubectl get pods sample-env -o yaml

```
hostIP: 10.0.2.16
 hostIPs:
 - ip: 10.0.2.16
 phase: Running
 podIP: 10.244.2.2
 podIPs:
 - ip: 10.244.2.2
 qosClass: BestEffort
 startTime: "2026-02-11T03:05:22Z"
```

환경 변수

- ❖ 파드 정보
  - ✓ 파드 정보를 환경 변수로 사용
    - 매니페스트: sample-env-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-env-pod
  labels:
    app: sample-app
spec:
  containers:
  - name: nginx-container
    image: nginx
    env:
    - name: K8S_NODE
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
```

환경 변수

- 파드 정보
  - ✓ 파드 정보를 환경 변수로 사용
    - 파드 생성: kubectl apply –f sample-env-pod.yaml
    - 파드 정보 확인: kubectl get pods -o wide sample-env-pod

      NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

      sample-env-pod 1/1 Running 0 20s 10.244.1.3 worker2 <none> <none>
    - 환경 변수 확인: kubectl exec -it sample-env-pod -- env | grep K8S_NODE

      K8S_NODE=worker2

환경 변수

- 컨테이너 정보
  - ✓ 파드의 정보와 마찬가지로 컨테이너 리소스에 대한 정보는 resourceFieldRef를 사용하여 확인 가능
  - ✓ 파드에는 여러 컨테이너 정보가 포함되어 있으므로 각 컨테이너 정보에 대해서는 fieldRef에서 확인할 수 없음
  - ✓ 확인 가능한 값은 kubectl get pods 파드이름 -o yaml 등으로 확인할 수 있음

환경 변수

- ❖ 컨테이너 정보
  - ✓ 실습
    - 매니페스트 파일 작성: sample-env-container.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-env-container
  labels:
    app: sample-app
spec:
  containers:
  - name: nginx-container
    image: nginx
    env:
    - name: CPU_REQUESTS
      valueFrom:
        resourceFieldRef:
          containerName: nginx-container
          resource: requests.cpu
    - name: CPU_LIMITS
      valueFrom:
        resourceFieldRef:
          containerName: nginx-container
          resource: limits.cpu
```

환경 변수

- ❖ 컨테이너 정보
  - ✓ 실습
    - 리소스 생성: kubectl apply -f sample-env-container.yaml
    - 확인: kubectl exec -it sample-env-container -- env | grep CPU

      CPU_REQUESTS=0

      CPU_LIMITS=1

환경 변수

❖ 환경 변수 이용 시 주의 사항

✓ 환경 변수 설정에 실패한 파드(sample-env-fail.yaml)

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-env-fail
  labels:
    app: sample-app
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    command: ["echo"]
    args: ["${TESTENV}", "${HOSTNAME}"]
    env:
    - name: TESTENV
      value: "100"
```

✓ 리소스 생성: kubectl apply -f sample-env-fail.yaml

```
pod/sample-env-fail created
```

환경 변수

- ❖ 환경 변수 이용 시 주의 사항
  - ✓ 로그 확인: kubectl logs sample-env-fail

    ${TESTENV} ${HOSTNAME}

    - 쿠버네티스에서 command나 args로 실행할 명령어를 지정할 때는 일반적인 방식으로 환경 변수를 사용할 수 없고 파드 매니페스트 내부에 정의된 환경 변수에만 $(TESTENV)라는 형태로 불러올 수 있음

환경 변수

❖ 환경 변수 이용 시 주의 사항

✓ 환경 변수 설정에 실패한 파드(sample-env-fail2.yaml)

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-env-fail2
  labels:
    app: sample-app
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    command: ["echo"]
    args: ["$(TESTENV)", "$(HOSTNAME)"]
    env:
    - name: TESTENV
      value: "100"
```

✓ 리소스 생성: kubectl apply -f sample-env-fail2.yaml

```
pod/sample-env-fail2 created
```

환경 변수

- 환경 변수 이용 시 주의 사항
  - ✓ 로그 확인: kubectl logs sample-env-fail2

    100 $(HOSTNAME)

    - command와 args에서 참조 가능한 것은 그 파드의 매니페스트 내부에 정의된 환경 변수만(이 경우에는 TESTENV)이라는 점에 주의해야 하는데 만약 OS에서만 참조할 수 있는 환경 변수를 사용하는 경우 Entrypoint(spec.containers!] .command)를 entrypoint.sh 등의 셸 스크립트로 하여 셸 스크립트 내부에서 처리할 수 있도록 함
    - 시크릿이나 컨피그맵에서 정의한 env, 파드나 컨네이너 정보를 fieldRef나 resourceFieldRef에서 참조한 env도 마찬가지로 [$(SOME_ENVIRONMENT)] 형식으로만 사용할 수 있음

환경 변수

- 환경 변수 이용 시 주의 사항
  - ✓ 외부 정보를 바탕으로 정의된 환경 변수 사용: sample-env-fail3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-env-fail3
  labels:
    app: sample-app
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    command: ["echo"]
    args: ["$(K8S_NODE)", "${K8S_NODE}"]
    env:
    - name: K8S_NODE
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
```

환경 변수

- 환경 변수 이용 시 주의 사항
  - ✓ 리소스 생성: kubectl apply -f sample-env-fail3.yaml
  - ✓ 로그 확인: kubectl apply -f sample-env-fail3.yaml

    worker2 ${K8S_NODE}

Secret

- 기밀 정보를 전달하는 방법
  - ✓ 도커 빌드 시 컨테이너 이미지에 추가
    - 이미지를 빌드할 때 애플리케이션 측이나 컨테이너 환경 변수 및 실행 인수 등에 패스워드 등을 추가할 수 있지만 이미지 자체에 기밀 정보가 포함되어 있다면 도커 레지스트리(도커 이미지를 저장하는 저장소)에 기밀 정보도 업로드하게 되어 보안상 바람직하지 않음
    - 기밀 정보 내용을 변경하려면 다시 이미지를 빌드해야 하므로 불편
  - ✓ 파드나 디플로이먼트의 매니페스트를 통해 전달
    - 매니페스트 자체에 기밀 정보가 포함되어 매니페스트 관리가 어려음
    - 깃 저장소와 같은 외부 서버에서 기밀 정보를 포함한 매니페스트를 관리하면 보안상 위험하고 여러 애플리케이션에서 같은 기밀 정보를 사용할 경우 매니페스트 관리가 어려움
  - ✓ 사용자명이나 패스워드 등의 기밀 정보를 별도 리소스에 정의하고 파드에서 읽어들일 수 있는데 그것이 시크릿 리소스
  - ✓ 시크릿을 정의하고 있는 매니페스트는 base64로 인코드되어 있지만 암호화되어 있지 않아 매니페스트를 깃 저장소 등에 업로드하는 것은 불가능한데 이러한 문제를 해결하기 위해 시크릿이 정의된 매니페스트를 암호화하는 kubesec/SealedSecret/ExternalSecret 같은 오픈 소스 소프트웨어도 있음

Secret

❖ 시크릿 분류

✓ 타입 목록

Opaque	일반적인 범용 용도
kubernetes.io/tls	TLS 인증서용
kubernetes.io/basic-auth	기본 인증용
kubernetes.io/dockerconfigjson	도커 레지스트리 인증 정보용
kubernetes.io/ssh-auth	SSH 인증 정보용
kubernetes.io/service-account-token	서비스 어카운트 토큰용
bootstrap.kubernetes.io/token	Bootstrap 토큰용

Secret

- 시크릿 분류
  - ✓ 일반 사용자명과 패스워드 같은 인증 정보 등은 스키마리스(스키마가 없는 데이터 구조)로 정의 가능한 type: Opaque를 사용
  - ✓ 인그레스 등에서 참조 가능한 TLS용의 type: kubernetes.io/tls나 도커 레지스트리 인증 정보용의 type: kubernetes.io/dockerconfigjson 등 도 있음
  - ✓ 수동으로 만들 일은 거의 없지만 파드에 서비스 어카운트 토큰이나 인증서를 마운트하기 위한 type: kubernetes.io/service-account-token이나 Bootstrap 토큰용 type: bootstrap. kubernetes. io/token 시크릿도 있음.
  - ✓ Opaque 타입 이외의 리소스에는 스키마가 정해져 있는데 TLS 타입의 경우에는 인증서(tls.crt) 와 비밀키(tls.key)스키마가 정의되어 있음
  - ✓ kubectl 명령어로 직접 시크릿을 생성할 경우 kubectl create secret generic 명령어에서 --type 옵션을 지정하여 시크릿을 생성
  - ✓ 스키마가 정해져 있는 일부 시크릿에 대해서는 kubectl create secret TYPE 명령어로 생성하는 것으로 스키마 체크를 할 수 있음

Secret

- 일반적인 범용 용도의 시크릿
  - ✓ 생성 방법
    - kubectl로 파일에서 값을 참조하여 생성(--from-file)
    - kubectl로 envfile에서 값을 참조하여 생성 (--from-env-file)
    - kubectl로 직접 값을 전달하여 생성(--from-literal)
    - 매니페스트에서 생성(-f)
  - ✓ 시크릿 리소스에서는 하나의 시크릿 안에 여러 키-밸류 값이 저장되는데 하나의 시크릿당 저장 가능한 데이터 사이즈는 총 1MB
  - ✓ 데이터베이스 인증 정보를 생성하는 경우 시크릿명은 db-auth, key는 username, password 두 가지를 지정하는데 물론 쿠버네티스 클러스터 내부에서 여러 데이터베이스 인증 정보를 사용하는 경우도 있을 수 있으므로 이런 경우에는 스크릿명을 유니크한 이름으로 정의하거나 시스템 별로 네임스페이스를 분리

Secret

- 일반적인 범용 용도의 시크릿
  - ✓ kubectl로 파일에서 값을 참조하여 생성(--from-file)
    - kubectl을 사용하여 파일에서 값을 참조함으로써 생성하는 경우에는 --from-file 옵션을 지정
    - 일반적으로 파일명이 그대로 키가 되기 때문에 확장자는 붙이지 않는 것이 좋음 (username.txt 대신 username)
    - 확장자를 붙일 경우 --from-file=username=username.txt 등과 같이 지정
    - 파일을 생성할 때 개행 코드가 들어가지 않도록 주의
    - echo -n 출력 결과를 리다이렉트하여 파일에 쓰는 방법을 사용할 수 있음

Secret

- 일반적인 범용 용도의 시크릿
  - ✓ kubectl로 파일에서 값을 참조하여 생성(--from-file)
    - 시크릿에 포함된 값을 내보내기
      - echo -n "root" > ./username
      - echo -n "rootpassword" > ./password
    - 생성된 파일로 시크릿을 생성
      - kubectl create secret generic --save-config sample-db-auth --from-file=./username --from-file=./password
    - data 부분을 확인
      - kubectl get secrets sample-db-auth -o json | jq .data

```
{
  "password": "cm9vdHBhc3N3b3Jk",
  "username": "cm9vdA=="
}
```

Secret

- 일반적인 범용 용도의 시크릿
  - ✓ kubectl로 파일에서 값을 참조하여 생성(--from-file)
    - base64로 인코드되어 있으므로 일반 텍스트로 확인
      - kubectl get secrets sample-db-auth -o json | jq -r .data.username

      cm9vdA==
    - 일반 텍스트로 확인
      - kubectl get secrets sample-db-auth -o json | jq -r .data.username | base64 -d

      root

Secret

- 일반적인 범용 용도의 시크릿
  - ✓ kubectl로 envfile 파일에서 값을 참조하여 생성(--from-env-file)
    - 하나의 파일에서 일괄적으로 생성하는 경우에는 envfile로 생성할 수 있음
    - envfile 생성(env-secret.txt)

      username=root

      password=rootpassword
    - 파일로 시크릿 생성
      - kubectl create secret generic --save-config sample-db-auth --from-env-file ./env-secret.txt

Secret

- 일반적인 범용 용도의 시크릿
  - ✓ kubectl로 직접 값을 전달하여 생성(--from-literal)
    - 직접 옵션으로 값을 지정하여 시크릿을 생성
      - kubectl create secret generic --save-config sample-db-auth --from-literal=username=root

Secret

- ❖ 일반적인 범용 용도의 시크릿
  - ✓ 매니페스트에서 생성(-f)
    - Base64로 인코드 한 값을 매니페스트에 추가(sample-db-auth.yaml)

```
apiVersion: v1
kind: Secret
metadata:
  name: sample-db-auth
type: Opaque
data:
  username: cm9vdA== # root
  password: cm9vdHBhc3N3b3Jk # rootpassword
```

Secret

- ❖ 일반적인 범용 용도의 시크릿
  - ✓ 매니페스트에서 생성(-f)
    - stringData 필드를 사용한 매니페스트에 추가(sample-db-auth-nobase64.yaml)

```
apiVersion: v1
kind: Secret
metadata:
  name: sample-db-auth-nobase64
type: Opaque
stringData:
  username: root
  password: rootpassword
```

Secret

- TLS 타입 시크릿
  - ✓ 인증서로 사용할 시크릿을 생성하는 경우 type에 tls를 지정
  - ✓ TLS 타입의 시크릿은 인그레스 리소스 등에서 사용하는 것이 일반적
  - ✓ TLS 타입의 시크릿인 경우도 매니페스트로 생성할 수 있지만 기본적으로 비밀키와 인증서 파일로 생성하는 것을 추천
  - ✓ 임시 테스트용으로 사용하기 위한 자체 서명된 인증서를 생성하는데 자체 서명된 인증서는 외부에 공개하는 정식 서비스에는 사용할 수 없는 인증서이므로 서비스 환경에서는 CA 루트(또는 ICA)로 서명된 정식 인증서를 사용
    - openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout ~/tls.key -out ~/tls.crt -subj "/CN=sample1.example.com"
  - ✓ kubectl에서 비밀키와 인증서 파일로 생성(--key/--cert)
    - kubectl create secret tls --save-config tls-sample --key ~/tls.key --cert ~/tls.crt

Secret

- 도커 레지스트리 타입 시크릿
  - ✓ 사용하고 있는 컨테이너 레지스트리가 프라이빗 저장소인 경우 도커 이미지를 가져오려면 인증이 필요한데 쿠버네티스에서는 이 인증 정보를 시크릿으로 정의하여 사용할 수 있음
  - ✓ 도커 레지스트리 인증용 시크릿을 생성하는 경우에는 type에 doker-registry를 지정
  - ✓ 도커 레지스트리 인증용 시크릿도 매니페스트로 생성할 수 있지만 형식이 특수하므로 kubectl로 직접 생성하는 것이 편함
  - ✓ 이 타입의 시크릿은 ~/.docker/config.json 파일 대체용으로 사용
  - ✓ kubectl로 직접 생성
    - kubectl create secret docker-registry --save-config sample-registry-auth --docker-server=REGISTRY_SERVER --docker-username=REGISTRY_USER --docker-password=REGISTRY_USER_PASSWORD --docker-email=REGISTRY_USER_EMAIL
  - ✓ base64로 인코드된 dockercfg 형식의 JSON 데이터
    - kubectl get secrets -o json sample-registry-auth | jq .data


```
{
  ".dockerconfigjson":
"eyJhdXRocyI6eyJSRUdJU1RSWV9TRVJWRVIiOnsidXNlcm5hbWUiOiJSRUdJU1RSWV9VU0VSIiwicGFzc3dvcmQiOiJSRUdJU1RSWV9VU0VSX1BBU1NXT1JEIiwiZW1haWwiOiJSRUdJU1RSWV9VU0VSX0VNQUlMIiwiYXV0aCI6IlVrVkhTVk5VVWxsZlZWTkZVanBTUlVkSlUxUlNXVjlWVTBWU1gxQkJVMU5YVDFKRSJ9fX0="
}
```

Secret

- 도커 레지스트리 타입 시크릿
  - ✓ base64로 디코드된 dockercfg 형식의 JSON 데이터
    - kubectl get secrets sample-registry-auth -o yaml | grep "₩.dockerconfigjson" | awk -F' ' '{print $2}' | base64 --decode

{"auths":{"REGISTRY_SERVER":{"username":"REGISTRY_USER","password":"REGISTRY_USER_PASSWORD","email":"REGISTRY_USER_EMAIL","auth":"UkVHSVNUUllfVVNFUjpSRUdJU1RSWV9VU0VSX1BBU1NXT1JE"}}}

Secret

- ❖ 도커 레지스트리 타입 시크릿
  - ✓ 이미지 다운로드 시 시크릿 사용
    - 인증이 필요한 도커 레지스트리에서 이미지를 다운로드하는 파드(sample-pull-secret.yaml)

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pull-secret
spec:
  containers:
  - name: secret-image-container
    image: REGISTRY_NAME/secret-image:latest
  imagePullSecrets:
  - name: sample-registry-auth
```

Secret

- 기본 인증 타입 시크릿
  - ✓ 사용자명과 패스워드로 인증하는 시스템을 사용하는 경우 기본 인증용 스키마를 가진 시크릿을 생성
  - ✓ kubectl로 직접 값을 전달하여 생성(--from-literal)
    - kubectl create secret generic --save-config sample-basic-auth --type kubernetes.io/basic-auth --from-literal=username=root --from-literal=password=rootpassword
  - ✓ 매니페스트에서 생성
    - sample-basic-auth.yaml 파일 작성

```
apiVersion: v1
kind: Secret
metadata:
  name: sample-basic-auth
type: kubernetes.io/basic-auth
data:
  username: cm9vdA== # root
  password: cm9vdHBhc3N3b3Jk # rootpassword
```

Secret

- SSH 인증 타입 시크릿
  - ✓ 파일에서 생성(--from-file)
    - SSH 비밀키 생성

    ssh-keygen -t rsa -b 2048 -f sample-key -C "sample"

    - SSH 비밀키 생성

    kubectl create secret generic --save-config sample-ssh-auth --type kubernetes.io/ssh-auth --from-file=ssh-privatekey=./sample-key

Secret

- SSH 인증 타입 시크릿
  - ✓ 매니페스트에서 생성(-f)
    - sample-ssh-auth.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: sample-ssh-auth
type: kubernetes.io/ssh-auth
data:
  ssh-privatekey:
LS0tLS1CRUdJTiBPUEVOU1NIIFBSSVZBVEUgS0VZLS0tLS0KYjNCbGJuTnphQzFyWlhrdGRq
RUFBQUFBQkc1dmJtVUFBQUFFYm05dVpRQUFBQUFBQUF>
```

Secret

- ❖ 시크릿 사용
  - ✓ 시크릿을 컨테이너에서 사용하는 패턴
    - ● 환경 변수로 전달
      - ◆ 시크릿의 특정 키만
      - ◆ 시크릿의 전체 키
    - ● 볼륨으로 마운트
      - ◆ 시크릿의 특정 키만
      - ◆ 시크릿의 전체 키

Secret

- 시크릿 사용
  - ✓ 환경 변수로 전달
    - 시크릿 한 개 키를 환경 변수로 전달하는 파드 샘플(sample-secret-single-env.yaml)

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-secret-single-env
spec:
  containers:
  - name: secret-container
    image: nginx:1.17
    env:
    - name: DB_USERNAME
      valueFrom:
        secretKeyRef:
          name: sample-db-auth
          key: username
```

    - 파드 생성 - kubectl apply -f sample-secret-single-env.yaml
    - 환경 변수 확인: kubectl exec -it sample-secret-single-env -- env | grep DB_USERNAME

```
DB_USERNAME=root
```

Secret

- 시크릿 사용
  - ✓ 환경 변수로 전달
    - 시크릿 전체를 환경 변수로 전달하는 파드 샘플(sample-secret-multi-env.yaml)

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-secret-multi-env
spec:
  containers:
  - name: secret-container
    image: nginx:1.17
    envFrom:
    - secretRef:
        name: sample-db-auth
```

    - 파드 생성 - kubectl apply -f sample-secret-multi-env.yaml
    - 환경 변수 확인: kubectl exec -it sample-secret-multi-env -- env

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=sample-secret-multi-env
NGINX_VERSION=1.17.10
NJS_VERSION=0.3.9
PKG_RELEASE=1~buster
password=rootpassword
username=root
```

Secret

- ❖ 시크릿 사용
  - ✓ 환경 변수로 전달
    - 시크릿의 전체 키에 접두사를 붙여 환경 변수로 전달하는 파드(sample-secret-prefix-env.yaml)

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-secret-prefix-env
spec:
  containers:
  - name: secret-container
    image: nginx:1.17
    envFrom:
    - secretRef:
        name: sample-db-auth
      prefix: DB1_
    - secretRef:
        name: sample-db-auth
      prefix: DB2_
```

    - 파드 생성 - kubectl apply -f sample-secret-prefix-env.yaml

Secret

- 시크릿 사용
  - ✓ 환경 변수로 전달
    - 환경 변수 확인: kubectl exec -it sample-secret-prefix-env -- env | egrep ^DB

```
DB1_password=rootpassword
DB1_username=root
DB2_password=rootpassword
DB2_username=root
```

Secret

- ❖ 시크릿 사용
  - ✓ 볼륨으로 마운트
    - ● 하나의 키만 전달
      - ◆ 매니페스트: sample-secret-single-volume.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-secret-single-volume
spec:
  containers:
  - name: secret-container
    image: nginx:1.17
    volumeMounts:
    - name: config-volume
      mountPath: /config
  volumes:
  - name: config-volume
    secret:
      secretName: sample-db-auth
      items:
      - key: username
        path: username.txt
```

Secret

- ❖ 시크릿 사용
  - ✓ 볼륨으로 마운트
    - ● 하나의 키만 전달
      - ◆ 파드 생성: kubectl apply -f sample-secret-single-volume.yaml
      - ◆ 볼륨 확인: kubectl exec -it sample-secret-single-volume -- cat /config/username.txt
      - ◆ root

Secret

- 시크릿 사용
  - ✓ 볼륨으로 마운트
    - 전체 전달
      - ◆ 매니페스트: sample-secret-multi-volume.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-secret-multi-volume
spec:
  containers:
  - name: secret-container
    image: nginx:1.17
    volumeMounts:
    - name: config-volume
      mountPath: /config
  volumes:
  - name: config-volume
    secret:
      secretName: sample-db-auth
```

Secret

- ❖ 시크릿 사용
  - ✓ 볼륨으로 마운트
    - ● 전체 전달
      - ◆ 리소스 배포: kubectl apply -f sample-secret-multi-volume.yaml
      - ◆ 확인: kubectl exec -it sample-secret-multi-volume -- ls /config

        password username

Secret

- 시크릿 사용
  - ✓ 동적 시크릿 업데이트
    - 볼륨 마운트를 사용한 시크릿에서는 일정 기간마다(kubelet의 Sync Loop 타이밍) kube-apiserver로 변경을 확인하고 변경이 있을 경우 파일을 교체
    - 기본 업데이트 주기는 60초로 설정되어 있는데 이 주기를 조정하려면 kubelet의 --sync-frequency 옵션을 지정
    - 환경 변수를 사용한 시크릿의 경우 파드를 기동할 때 환경 변수가 정해지기 때문에 동적으로 변경할 수는 없음
    - 시크릿에 마운트 된 디렉토리 확인: kubectl exec -it sample-secret-multi-volume -- ls -la /config

```
total 4
drwxrwxrwt 3 root root  120 Feb 15 09:05 .
drwxr-xr-x 1 root root 4096 Feb 15 09:05 ..
drwxr-xr-x 2 root root   80 Feb 15 09:05 ..2026_02_15_09_05_48.3009432798
lrwxrwxrwx 1 root root   32 Feb 15 09:05 ..data -> ..2026_02_15_09_05_48.3009432798
lrwxrwxrwx 1 root root   15 Feb 15 09:05 password -> ..data/password
lrwxrwxrwx 1 root root   15 Feb 15 09:05 username -> ..data/username
```

Secret

- 시크릿 사용
  - ✓ 동적 시크릿 업데이트
    - 파드의 /config/username 파일 내용 확인: kubectl exec -it sample-secret-multi-volume -- cat /config/username

      root
    - 시크릿 변경 전 경과 시간(AGE) 확인: kubectl get pods sample-secret-multi-volume

      NAME READY STATUS RESTARTS AGE

      sample-secret-multi-volume 1/1 Running 0 21m

Secret

- ❖ 시크릿 사용
  - ✓ 동적 시크릿 업데이트
    - ● 시크릿 내용 업데이트
      - ◆ 매니페스트

```
cat << EOF |  kubectl apply -f -
apiVersion: v1
kind: Secret
metadata:
  name: sample-db-auth
type: Opaque
data:
  username: YWRtaW4=
EOF
```

      - ◆ 리소스 배포: kubectl exec -it sample-secret-multi-volume -- cat /config/username

admin

      - ◆ 확인: kubectl exec -it sample-secret-multi-volume -- ls /config

username

ConfigMap

- 개요
  - ✓ 컨피그맵(ConfigMap)은 설정 정보 등을 키-밸류 값으로 저장할 수 있는 데이터 저장 리소스
  - ✓ 키-밸류 형식 외에도 nginx.conf나 httpd.conf 같은 설정 파일 자체도 저장할 수 있음
  - ✓ 컨피그맵 리소스는 하나의 컨피그맵 안에 여러 키-밸류 값이 저장되는데 하나의 컨피그맵마다 저장할 수 있는 사이즈는 총 1MB
  - ✓ nginx.conf 전체를 컨피그맵에 저장해도 되고 nginx.conf 설정 파라미터만 저장 가능

ConfigMap

- ConigMap 생성
  - ✓ kubectl create 명령으로 생성
    - kubectl을 사용하여 파일에서 값을 참조함으로써 생성하는 경우에는 --from-file 옵션을 지정하며 직접 입력한 문자열을 이용할 때는 --from-literal 옵션 지정
    - 파일명이 그대로 키가 되는데 키 이름을 변경하고 싶은 경우에는 --from-file=nginx.conf=sample-nginx.conf 등과 같이 지정
    - nginx.conf 파일을 생성하거나 복사

```
user   nginx;
worker_processes   auto;
error_log   /var/log/nginx/error.log warn;
pid         /var/run/nginx.pid;
```

    - 컨피그맵 생성: kubectl create configmap --save-config sample-configmap --from-file=./nginx.conf
    - 컨피그맵 데이터 확인: kubectl get configmaps sample-configmap -o json | jq .data

```
{
  "nginx.conf": "user   nginx;₩nworker_processes   auto;₩nerror_log
/var/log/nginx/error.log warn;₩npid         /var/run/nginx.pid;₩n"
}
```

ConfigMap

- ConigMap 생성
  - ✓ kubectl create 명령으로 생성
    - 컨피그맵 데이터 확인: kubectl describe configmap sample-configmap

```
Name:         sample-configmap
Namespace:    default
Labels:       <none>
Annotations:  <none>
Data
====
nginx.conf:
----
user   nginx;
worker_processes   auto;
error_log   /var/log/nginx/error.log warn;
pid         /var/run/nginx.pid;
BinaryData
====
Events:  <none>
```

ConfigMap

- ConigMap 생성
  - ✓ kubectl create 명령으로 생성
    - binaryData로 생성 가능: kubectl create configmap sample-configmap-binary --from-file image.jpg --from-literal=index.html="Hello Kubernetes" --dry-run=client -o yaml > sample-configmap-binary.yaml
    - 매니페스트의 binaryData 필드에는 base64로 인코드한 값이 등록되는데 kubectl create secret generic --from-file을 사용한 경우 시크릿 리소스에서도 바이너리 데이터를 취급할 수 있음

ConfigMap

- ConigMap 생성
  - ✓ kubectl create 명령으로 생성
    - ConfigMap 컨텐츠를 사용한 웹 서버(sample-configmap-binary-webserver.yaml)

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-configmap-binary-webserver
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    volumeMounts:
    - name: config-volume
      mountPath: /usr/share/nginx/html
  volumes:
  - name: config-volume
    configMap:
      name: sample-configmap-binary
```

ConfigMap

- ConigMap 생성
  - ✓ kubectl create 명령으로 생성
    - 리스스 배포
      - kubectl apply -f sample-configmap-binary-webserver.yaml
      - 포트 포워딩: kubectl port-forward sample-configmap-binary-webserver 8080:80
    - 다른 터미널에서 수행: curl http://localhost:8080

      Hello Kubernetes

ConfigMap

- ConigMap 생성
  - ✓ kubectl create 명령으로 생성
    - kubectl로 직접 값을 전달하여 생성: kubectl create configmap --save-config web-config --from-literal=connection.max=100 --from-literal=connection.min=10

      configmap/web-config created

ConfigMap

- ConigMap 생성
  - ✓ 매니페스트로 생성
    - 리소스 생성 파일: sample-configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: sample-configmap
data:
  thread: "16"
  connection.max: "100"
  connection.min: "10"
  sample.properties: |
    property.1=value-1
    property.2=value-2
    property.3=value-3
  nginx.conf: |
    user nginx;
    worker_processes auto;
    error_log /var/log/nginx/error.log;
    pid /run/nginx.pid;
```

ConfigMap

- ConigMap 생성
  - ✓ 매니페스트로 생성
    - 리소스 생성 파일: sample-configmap.yaml

```
test.sh: |
  #!/bin/bash
  echo "Hello, kubernetes"
  sleep infinity
```

    - 리소스 생성: kubectl apply -f sample-configmap.yaml

ConfigMap

- ConigMap 사용
  - ✓ 사용 방법
    - 환경 변수로 전달
      - 컨피그맵의 특정 키만
      - 컨피그맵 전체 키
    - 볼륨 마운트
      - 컨피그맵의 특정 키만
      - 컨피그맵 전체 키

ConfigMap

- ConigMap 사용
  - ✓ 환경 변수로 전달
    - 컨피그맵의 특정 키를 전달하는 경우에는 spec.containers[ ].env의 valueFrom.configMapKeyRef를 사용하여 지정
      - ◆ 특정 키 이용: sample-configmap-single-env.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-configmap-single-env
spec:
  containers:
  - name: configmap-container
    image: nginx:1.17
    env:
    - name: CONNECTION_MAX
      valueFrom:
        configMapKeyRef:
          name: sample-configmap
          key: connection.max
```

ConfigMap

- ConigMap 사용
  - ✓ 환경 변수로 전달
    - 컨피그맵의 특정 키를 전달하는 경우에는 spec.containers[ ].env의 valueFrom.configMapKeyRef를 사용하여 지정
      - 배포: kubectl apply -f sample-configmap-single-env.yaml
      - 확인: kubectl exec -it sample-configmap-single-env -- env | grep CONNECTION_MAX

        CONNECTION_MAX=100

ConfigMap

- ConigMap 사용
  - ✓ 환경 변수로 전달
    - ● 컨피그맵의 전체를 전달
      - ◆ 전체 전달: sample-configmap-multi-env.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-configmap-multi-env
spec:
  containers:
  - name: configmap-container
    image: nginx:1.17
    envFrom:
    - configMapRef:
        name: sample-configmap
```

ConfigMap

- ConigMap 사용
  - ✓ 환경 변수로 전달
    - ● 컨피그맵의 전체를 전달
      - ◆ 배포: kubectl apply -f sample-configmap-multi-env.yaml
      - ◆ 확인: kubectl exec -it sample-configmap-multi-env – env

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

HOSTNAME=sample-configmap-multi-env

NGINX_VERSION=1.17.10

NJS_VERSION=0.3.9

PKG_RELEASE=1~buster

sample.properties=property.1=value-1

property.2=value-2

property.3=value-3

ConfigMap

- ConigMap 사용
  - ✓ 볼륨으로 마운트
    - ● 특정 키만 전달
      - ◆ 전체 전달: sample-configmap-single-volume.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-configmap-single-volume
spec:
  containers:
  - name: configmap-container
    image: nginx:1.17
    volumeMounts:
    - name: config-volume
      mountPath: /config
  volumes:
  - name: config-volume
    configMap:
      name: sample-configmap
      items:
      - key: nginx.conf
        path: nginx-sample.conf
```

ConfigMap

- ConigMap 사용
  - ✓ 볼륨으로 마운트
    - 특정 키만 전달
      - 배포: kubectl apply -f sample-configmap-singe-volume.yaml
      - 확인: kubectl exec -it sample-configmap-single-volume -- cat /config/nginx-sample.conf

```
user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log;
pid /run/nginx.pid;
```

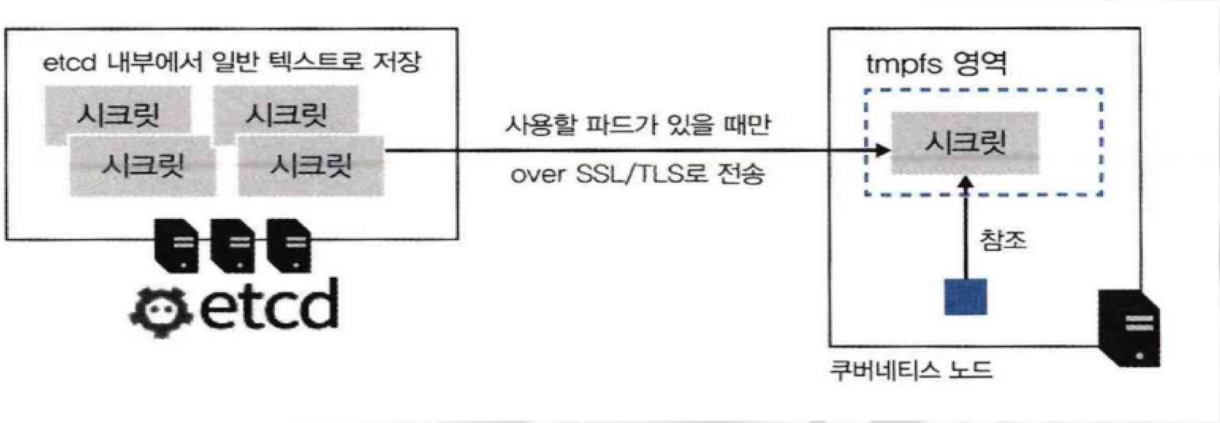
ConfigMap

- ConigMap 사용
  - ✓ 볼륨으로 마운트
    - ● 전체 키 전달
      - ◆ 전체 전달: sample-configmap-multi-volume.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-configmap-multi-volume
spec:
  containers:
  - name: configmap-container
    image: nginx:1.17
    volumeMounts:
    - name: config-volume
      mountPath: /config
  volumes:
  - name: config-volume
    configMap:
      name: sample-configmap
```

ConfigMap

- ❖ Secret 과 ConigMap 공통 주제
  - ✓ 시크릿 과 컨피그맵의 사용 구분
    - 시크릿과 컨피그맵은 비슷한 리소스들이지만 둘의 가장 큰 차이는 시크릿이 기밀 정보를 취급하기 위한 리소스라는 점
    - 시크릿은 안전하게 처리할 수 있는 구조로 되어 있는데 시크릿 데이터는 쿠버네티스 마스터가 사용하는 분산 KVS(Key-Value Store)의 etcd에 저장되며 실제 시크릿을 사용하는 파드가 있는 경우에만 etcd에서 쿠버네티스 노드에 데이터를 보내는데 쿠버네티스 노드상에 영구적으로 데이터가 남지 않도록 시크릿 데이터는 tmpfs 영역(메모리상에 구축된 임시 파일 시스템)에 저장되게 되어 있으므로 쿠버네티스 버전과 쿠버네티스 환경에 따라서는 etcd에 저장되는 시크릿이 암호화되어 있지 않으므로 etcd에 대한 접속 권한 설정을 제대로 해두어야 함
    - 파드가 시크릿을 사용하기까지의 흐름



ConfigMap

- Secret 과 ConigMap 공통 주제
  - ✓ 시크릿 과 컨피그맵의 사용 구분
    - 시크릿이 안전한 또 하나의 이유는 kubectl 명령어로 표시했을 때 값이 보기 어렵게 되어 있다는 점인데 kubectl describe secert 명령어에서는 값이 보이지 않게 되어 있고 kubectl get secret -o yaml 명령어로 매니페스트 전체를 표시해도 base64로 인코드되어 있어 화면에서는 판단하기 어려움
    - 확인: kubectl describe secret sample-db-auth

      Name: sample-db-auth
      Namespace: default
      Labels: <none>
      Annotations: <none>

      Type: Opaque

      Data
      ====
      username: 5 bytes

      - ◆ 매니페스트에 저장된 데이터는 base64로 인코드되어 있을 뿐이므로 그대로 깃 저장소에 업로드하는 것은 피하고, 시크릿을 암호화하는 OSS 또는 마찬가지로 암호화 기능을 제공하는 3rd 파티 솔루션인 Vault 등을 사용하는 것을 검토

ConfigMap

- Secret 과 ConigMap 공통 주제
  - ✓ 시크릿 과 컨피그맵의 데이터 마운트 시 퍼미션 변경
    - 컨피그맵에 저장된 스크립트를 파드에서 실행하는 경우 컨피그맵 데이터에서 볼륨을 생성할 때 실행 권한을 부여할 수 있음
    - 시크릿도 저장된 기밀 데이터를 파드에 마운트하는 경우 시크릿 데이터에서 볼륨을 생성할 때 퍼미션(permission)을 부여할 수 있음
    - 컨피그맵과 시크릿 모두 기본 값 0644(rw-r—r--)로 마운트
    - 퍼미션은 8진수 표기에서 10진수 표기로 변환한 형태를 사용하여 기술

10진수 표기	8진수 표기	rwx 표기
256	0400	r————
384	0600	rw————
420	0644	rw-r—r—
448	0700	rwx————
493	0755	rwxr-xr-x
511	0777	rwxrwxrwx

ConfigMap

- ❖ Secret 과 ConigMap 공통 주제
  - ✓ 시크릿 과 컨피그맵의 데이터 마운트 시 퍼미션 변경
    - ● ConfigMap 내부의 test.sh를 755로 마운트해서 스크립트를 그대로 실행
      - ◆ 리소스 파일: sample-configmap-scripts.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-configmap-scripts
spec:
  containers:
  - name: configmap-container
    image: nginx:1.17
    command: ["/config/test.sh"]
    volumeMounts:
    - name: config-volume
      mountPath: /config
  volumes:
  - name: config-volume
    configMap:
      name: sample-configmap
      items:
      - key: test.sh
        path: test.sh
        mode: 493 # 0755
```

ConfigMap

- ❖ Secret 과 ConigMap 공통 주제
  - ✓ 시크릿 과 컨피그맵의 데이터 마운트 시 퍼미션 변경
    - ● ConfigMap 내부의 test.sh를 755로 마운트해서 스크립트를 그대로 실행
      - ◆ 리소스 배포: kubectl apply -f sample-configmap-scripts.yaml
      - ◆ 확인: kubectl logs sample-configmap-scripts

        Hello, kubernetes

ConfigMap

- Secret 과 ConigMap 공통 주제
  - ✓ 시크릿 과 컨피그맵의 데이터 마운트 시 퍼미션 변경
    - ● 시크릿 내부의 모든 데이터를 0400으로 마운트
      - ◆ 리소스 파일: sample-secret-secure.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-secret-secure
spec:
  containers:
  - name: secret-container
    image: nginx:1.17
    volumeMounts:
    - name: config-volume
      mountPath: /config
  volumes:
  - name: config-volume
    secret:
      secretName: sample-db-auth
      defaultMode: 256
```

ConfigMap

- Secret 과 ConigMap 공통 주제
  - ✓ 시크릿 과 컨피그맵의 데이터 마운트 시 퍼미션 변경
    - ● 시크릿 내부의 모든 데이터를 0400으로 마운트
      - ◆ 리소스 배포: kubectl apply -f sample-secret-secure.yaml
      - ◆ 확인: kubectl exec -it sample-secret-secure -- ls -l /config

        total 0

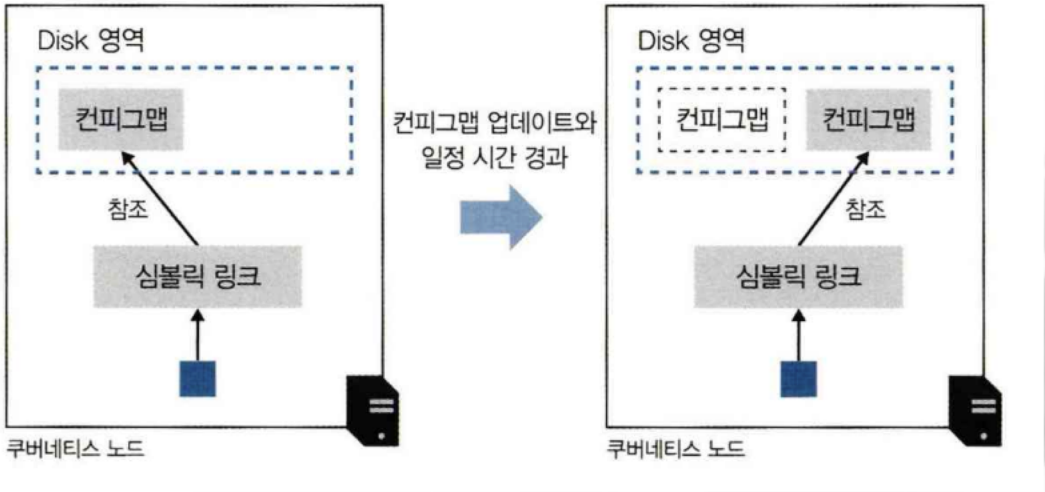
        lrwxrwxrwx 1 root root 15 Feb 16 00:07 username -> ..data/username
      - ◆ 확인: kubectl exec -it sample-secret-secure -- ls -l /config/..data/

        total 4

        -r-------- 1 root root 5 Feb 16 00:07 username

ConfigMap

- Secret 과 ConigMap 공통 주제
  - ✓ 동적 컨피그맵 업데이트
    - 볼륨 마운트를 사용한 컨피그맵에서는 일정 기간마다(kubectl의 Sync Loop 타이밍) kube-apiserver로 변경을 확인하고 변경이 있을 경우에는 파일을 교체
    - 기본 업데이트 간격은 60초로 설정되어 있는데 이 간격을 조정할 경우에는 kubelet의 --sync-frequency 옵션을 설정
    - 환경 변수를 사용한 컨피그맵은 기동할 때 환경 변수가 정해지기 때문에 동적으로 업데이트를 할 수 없으므로 kubectl rollout restart 명령어를 사용하여 파드를 재기동



ConfigMap

- ❖ Secret 과 ConigMap 공통 주제
  - ✓ 동적 컨피그맵 업데이트
    - ● thread 값을 수정
      - ◆ 컨피그 맵이 마운된 디렉토리 확인: kubectl exec -it sample-configmap-multi-volume -- ls -al /config

```
total 12
drwxrwxrwx 3 root root 4096 Feb 15 23:21 .
drwxr-xr-x 1 root root 4096 Feb 15 23:21 ..
drwxr-xr-x 2 root root 4096 Feb 15 23:21 ..2026_02_15_23_21_38.2725348367
lrwxrwxrwx 1 root root   32 Feb 15 23:21 ..data -> ..2026_02_15_23_21_38.2725348367
lrwxrwxrwx 1 root root   21 Feb 15 23:21 connection.max -> ..data/connection.max
lrwxrwxrwx 1 root root   21 Feb 15 23:21 connection.min -> ..data/connection.min
lrwxrwxrwx 1 root root   17 Feb 15 23:21 nginx.conf -> ..data/nginx.conf
lrwxrwxrwx 1 root root   24 Feb 15 23:21 sample.properties -> ..data/sample.properties
lrwxrwxrwx 1 root root   14 Feb 15 23:21 test.sh -> ..data/test.sh
lrwxrwxrwx 1 root root   13 Feb 15 23:21 thread -> ..data/thread
```

ConfigMap

- ❖ Secret 과 ConigMap 공통 주제
  - ✓ 동적 컨피그맵 업데이트
    - ● thread 값을 수정
      - ◆ thread 값 확인: kubectl exec -it sample-configmap-multi-volume -- cat /config/thread

      16

      - ◆ 컨피그맵 업데이트

      ```
      cat << EOF | kubectl apply -f -
      apiVersion: v1
      kind: ConfigMap
      metadata:
        name: sample-configmap
      data:
        thread: "32"
      EOF
      ```

ConfigMap

- ❖ Secret 과 ConigMap 공통 주제
  - ✓ 동적 컨피그맵 업데이트
    - thread 값을 수정
      - ◆ 디렉토리 확인: kubectl exec -it sample-configmap-multi-volume -- ls -al /config

```
total 12
drwxrwxrwx 3 root root 4096 Feb 16 00:13 .
drwxr-xr-x 1 root root 4096 Feb 15 23:21 ..
drwxr-xr-x 2 root root 4096 Feb 16 00:13 ..2026_02_16_00_13_41.2194393135
lrwxrwxrwx 1 root root   32 Feb 16 00:13 ..data -> ..2026_02_16_00_13_41.2194393135
lrwxrwxrwx 1 root root   13 Feb 15 23:21 thread -> ..data/thread
```

      - ◆ thread 값 확인: kubectl exec -it sample-configmap-multi-volume -- cat /config/thread

```
32
```

ConfigMap

- Secret 과 ConigMap 공통 주제
  - ✓ 시크릿이나 컨피그맵 데이터 변경 거부
    - 리소스 매니페스트 작성: sample-secret-immutable.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: sample-secret-immutable
type: Opaque
data:
  username: cm9vdA== # root
  password: cm9vdHBhc3N3b3Jk # rootpassword
immutable: true
```

    - 리소스 배포: kubectl apply -f sample-secret-immutable.yaml
    - 변경 시도: kubectl patch secret sample-secret-immutable -p '{"data": {"username": "aG9nZQ=="}}'

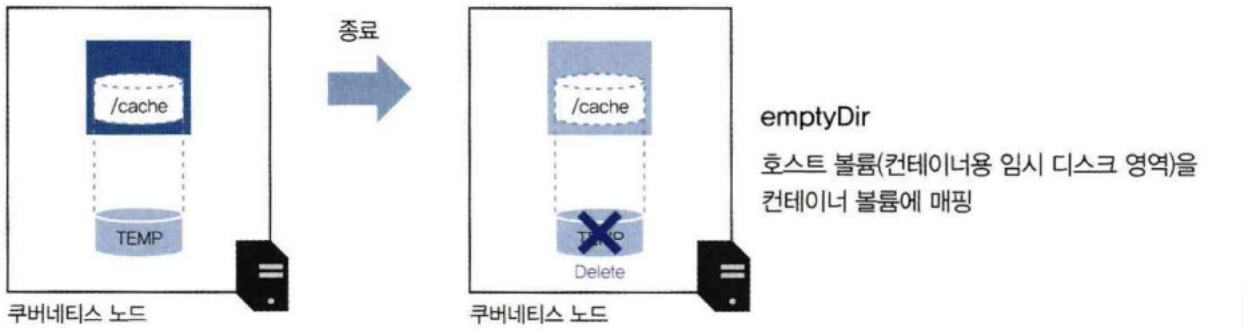
      The Secret "sample-secret-immutable" is invalid: data: Forbidden: field is immutable when `immutable` is set

Storage API

- 볼륨
  - ✓ 개요
    - 쿠버네티스에서는 볼륨을 추상화하여 파드와 느슨하게 결합된 리소스로 정의
    - 볼륨 플러그인은 여러 가지가 제공됨
    - 시크릿과 컨피그맵도 볼륨 플러그인의 한 종류
    - 볼륨 플러그인 종류
      - emptyDir
      - hostPath
      - downwardAPI
      - projected
      - nfs
      - iscsi
      - cephfs

Storage API

- ❖ 볼륨
  - ✓ emptyDir
    - emptyDir은 파드용 임시 디스크 영역으로 사용할 수 있고 파드가 종료(Terminate)되면 삭제됨
    - 호스트의 임의 영역을 마운트할 수 없으며 호스트에 있는 파일을 참조할 수도 없음



Storage API

- ❖ 볼륨
  - ✓ emptyDir
    - ● emptyDir 볼륨을 마운트
      - ◆ 매니페스트 파일(sample-emptydir.yaml)

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-emptydir
spec:
  containers:
  - image: nginx:1.17
    name: nginx-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

Storage API

- ❖ 볼륨
  - ✓ emptyDir
    - ● emptyDir 볼륨을 마운트
      - ◆ 리소스 생성: kubectl apply -f sample-emptydir.yaml
      - ◆ 할당된 emptyDir 볼륨을 확인: kubectl exec -it sample-emptydir -- df -h | grep /cache

      /dev/mapper/ubuntu--vg-ubuntu--lv 37G 6.5G 29G 19% /cache

Storage API

- ❖ 볼륨
  - ✓ emptyDir
    - ● emptyDir 볼륨의 리소스 제한
      - ◆ 매니페스트 파일(sample-emptydir-limit.yaml)

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-emptydir-limit
spec:
  containers:
  - image: nginx:1.17
    name: nginx-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir:
      sizeLimit: 128Mi
```

Storage API

- ❖ 볼륨
  - ✓ emptyDir
    - ● emptyDir 볼륨의 리소스 제한
      - ◆ 리소스 생성: kubectl apply -f sample-emptydir-limit.yaml
      - ◆ 파드 상태 확인: kubectl get pods --watch

```
NAME                     READY   STATUS      RESTARTS   AGE
sample-emptydir          1/1     Running     0          4m15s
sample-emptydir-limit    1/1     Running     0          32s
sample-emptydir-limit    0/1     Completed   0          2m7s
```

      - ◆ 1M 파일을 150개 생성: kubectl exec -it sample-emptydir-limit -- dd if=/dev/zero of=/cache/dummy bs=1M count=150

Storage API

- ❖ 볼륨
  - ✓ emptyDir
    - ● 고속 tmpfs 메모리 영역을 사용
      - ◆ 리소스 매니페스트: sample-emptydir-memory.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-emptydir-memory
spec:
  containers:
  - image: nginx:1.17
    name: nginx-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir:
      medium: Memory
      sizeLimit: 128Mi
```

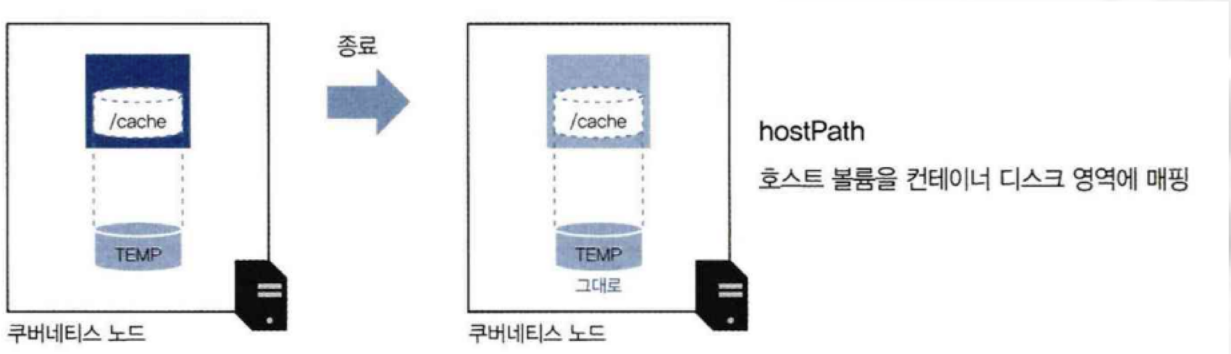
Storage API

- ❖ 볼륨
  - ✓ emptyDir
    - ● 컨테이너가 사용할 수 있는 메모리 제한
      - ◆ 리소스 매니페스트: sample-emptydir-memory-with-memory-limits.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-emptydir-memory-with-memory-limits
spec:
  containers:
  - image: nginx:1.17
    name: nginx-container
    resources:
      limits:
        memory: 64Mi
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir:
      medium: Memory
      sizeLimit: 128Mi
```

Storage API

- 볼륨
  - ✓ hostPath
    - hostPath는 쿠버네티스 노드상의 영역을 컨테이너에 매핑하는 플러그인
    - emptyDir과 다르게 호스트의 임의 영역을 마운트할 수 있기 때문에 사용할 때는 호스트의 어떤 영역을 사용할지 지정해야 함
    - type은 Directory/DirectoryOrCreate/File/Socket/BlockDevice 등을 선택할 수 있음
    - DirectoryOrCreate와 Directory의 차이는 디렉토리가 존재하지 않을 경우 생성하고 기동하는지 여부에 있음
    - 보안상의 이유로 안전하지 않은 컨테이너가 업로드될 수 있으므로 사용하지 말 것을 권장
    - 보안상 hostPath를 사용할 수 없는 쿠버네티스 환경도 있음



Storage API

- ❖ 볼륨
  - ✓ hostPath
    - ● hostPath 볼륨을 마운트
      - ◆ 리소스 매니페스트: sample-hostpath.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-hostpath
spec:
  containers:
  - image: nginx:1.17
    name: nginx-container
    volumeMounts:
    - mountPath: /srv
      name: hostpath-sample
  volumes:
  - name: hostpath-sample
    hostPath:
      path: /etc
      type: DirectoryOrCreate
```

Storage API

- ❖ 볼륨
  - ✓ hostPath
    - ● hostPath 볼륨을 마운트
      - ◆ 리소스 생성: kubectl apply -f sample-hostpath.yaml
      - ◆ Host OS 이미지 확인: kubectl exec -it sample-hostpath -- cat /srv/os-release | grep PRETTY_NAME

      PRETTY_NAME="Debian GNU/Linux 10 (buster)"
      - ◆ 컨테이너 OS 이미지 확인: kubectl exec -it sample-hostpath -- cat /etc/os-release | grep PRETTY_NAME

      PRETTY_NAME="Debian GNU/Linux 10 (buster)"

Storage API

- ❖ 볼륨
  - ✓ downwardAPI: downwardAPI는 파드 정보 등을 파일로 배치하기 위한 플러그인
    - ● downwardAPI 사용
      - ◆ 리소스 매니페스트: sample-downward-api.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-downward-api
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    volumeMounts:
    - name: downward-api-volume
      mountPath: /srv
```

Storage API

❖ 볼륨

✓ downwardAPI: downwardAPI는 파드 정보 등을 파일로 배치하기 위한 플러그인

● downwardAPI 사용

◆ 리소스 매니페스트: sample-downward-api..yaml

```
volumes:
- name: downward-api-volume
  downwardAPI:
    items:
    - path: "podname"
      fieldRef:
        fieldPath: metadata.name
    - path: "cpu-request"
      resourceFieldRef:
        containerName: nginx-container
        resource: requests.cpu
```

◆ 리소스 생성: kubectl apply -f sample-downward-api.yaml

◆ 파드 정보 확인: kubectl exec -it sample-downward-api -- ls /srv

cpu-request podname

Storage API

- ❖ 볼륨
  - ✓ projectd: 시크릿/컨피그맵/downwardAPI/serviceAccountToken의 볼륨 마운트를 하나의 디렉터리에 통합하는 플러그인으로 시크릿 인증 정보와 컨피그맵 설정 파일을 하나의 디렉토리에 배치하고 싶을 경우 사용
    - projectd 사용
      - ◆ 리소스 매니페스트: sample-projected.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-projected
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    volumeMounts:
    - name: projected-volume
      mountPath: /srv
```

Storage API

- ❖ 볼륨
  - ✓ projectd
    - ● projectd 사용
      - ◆ 리소스 매니페스트: sample-projected.yaml

```
volumes:
- name: projected-volume
  projected:
    sources:
    - secret:
        name: sample-db-auth
        items:
        - key: username
          path: secret/username.txt
    - configMap:
        name: sample-configmap
        items:
        - key: nginx.conf
          path: configmap/nginx.conf
    - downwardAPI:
        items:
        - path: "podname"
          fieldRef:
            fieldPath: metadata.name
```

Storage API

- ❖ 볼륨
  - ✓ projectd
    - projectd 사용
      - ◆ 리소스 생성: kubectl apply -f sample-projected.yaml
      - ◆ /srv 디렉토리 확인: kubectl exec -it sample-projected -- ls /srv
      - ◆ /srv/configmap 디렉토리 확인: kubectl exec -it sample-projected -- ls /srv/configmap
      - ◆ /srv/secret 디렉토리 확인: kubectl exec -it sample-projected -- ls /srv/secret

Storage API

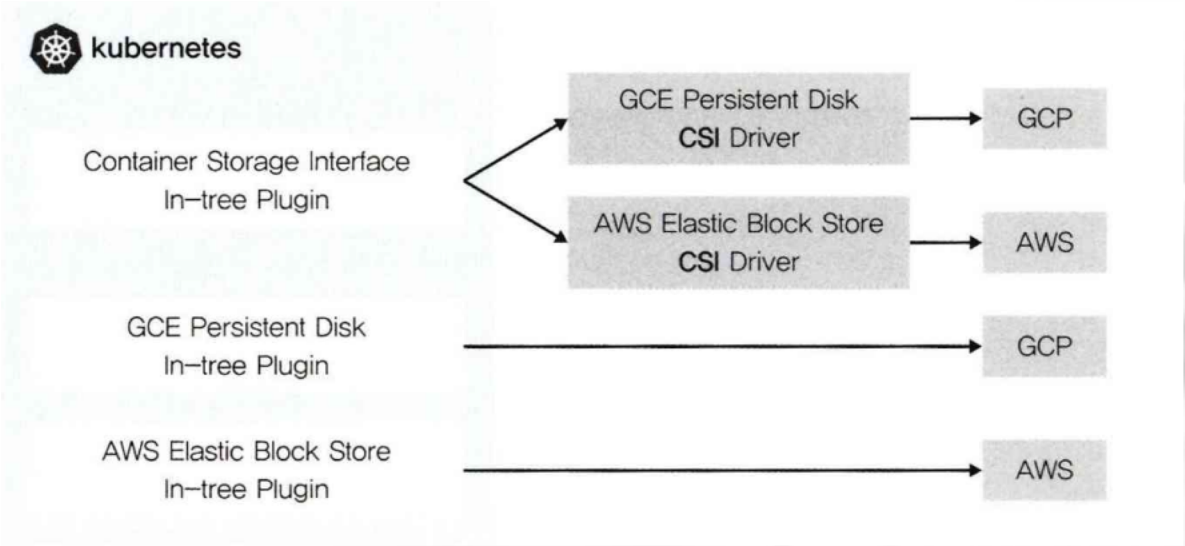
- ❖ 영구 볼륨
  - ✓ 개요
    - 영구 볼륨은 영속성 영역으로 확보된 볼륨
    - 볼륨은 파드 정의 안에 직접 지정하는 형태로 연결하지만 영구 볼륨은 개별 리소스로 생성한 후 사용
    - 매니페스트를 사용하여 영구 볼륨 리소스를 생성
    - 영구 볼륨은 엄밀히 말하면 컨피그 & 스토리지 리소스가 아닌 클러스터 리소스로 분류됨
    - 영구 볼륨은 기본적으로 네트워크를 통해 디스크를 어태치(attach)하는 디스크 타입
    - 단일 노드 시 테스트용으로 hostPath가 제공되지만 영구 볼륨으로는 실용적이지 않음
    - 영구 볼륨은 플러거블(pluggable)한 구조

Storage API

- ❖ 영구 볼륨
  - ✓ 종류
    - GCE Persistent Disk
    - AWS Elastic Block Store
    - Azure File
    - NFS(Network File System)
    - iSCSI(Internet Small Computer Systems Interface)
    - Ceph(RBD, CephFS, 차세대 오픈소스 분산 스토리지 시스템)
    - OpenStack Cinder(오픈스택 클라우드 환경의 블록 스토리지)
    - GlusterFS(분산 파일 시스템의 한 종류)
    - Container Storage Interface

Storage API

- ❖ 영구 볼륨
  - ✓ Container Storage Interface
    - Container Storage Interface(csi)는 조금 특수한 플러그인이며 컨테이너 오케스트레이션 엔진과 스토리지 시스템을 연결하는 인터페이스
    - 기존에는 쿠버네티스에서 프로바이더용 플러그인이 개발되었지만 이 방법은 특정 프로바이더의 볼륨 관련 업데이트가 있을 때 쿠버네티스 자체도 변경해야 했기 때문에 개발 주기가 한정되거나 사용하지 않는 구성 요소 코드도 쿠버네티스 자체에 포함되는 문제가 발생해서 새로 만들어진 것이 CSI
    - CSI를 사용하면, 쿠버네티스부터 공통화된 CSI 인터페이스를 통해 다양한 프로바이더를 사용할 수 있음



Storage API

- ❖ 영구 볼륨
  - ✓ 컨테이너 스토리지의 흐름 (PV/PVC)
    - StorageClass: 관리자가 이 시스템은 Ceph(또는 iSCSI)를 사용하며 CSI 드라이버는 이것이다 라고 정의
    - PersistentVolumeClaim (PVC): 개발자가 "나 10GB 하드디스크가 필요해"라고 요청
    - CSI Driver: 요청을 받은 CSI 드라이버가 뒷단의 스토리지(Ceph, GlusterFS 등)에 명령을 내려 실제 볼륨을 생성
    - PersistentVolume (PV): 생성된 볼륨이 쿠버네티스 자원으로 등록되고 컨테이너에 연결

Storage API

- ❖ 영구 볼륨
  - ✓ 영구 볼륨 생성
    - ● 영구 볼륨 설정은 각 플러그인 특유의 설정 외에는 동일
    - ● 설정 항목
      - ◆ 레이블
      - ◆ 용량
      - ◆ 접근 모드
      - ◆ Reclaim Policy
      - ◆ 마운트 옵션
      - ◆ 스토리지 클래스
      - ◆ 각 플러그인 특유의 설정

Storage API

- ❖ 영구 볼륨
  - ✓ 영구 볼륨 생성
    - 현재 컴퓨터에 디렉토리 생성: mkdir -p /home/user/data
    - 매니페스트 작성: pv.yaml

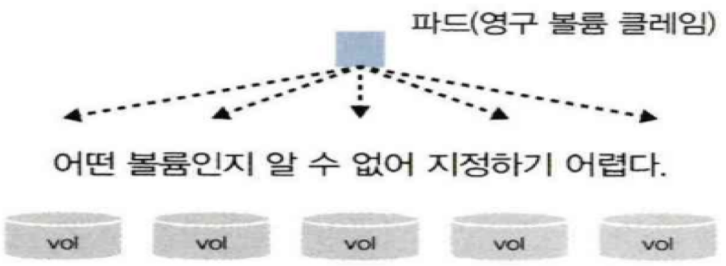
```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain # 파드가 삭제되어도 데이터 유지
  storageClassName: manual
  hostPath:
    path: "/home/user/data" # 실제 호스트 컴퓨터의 경로
```

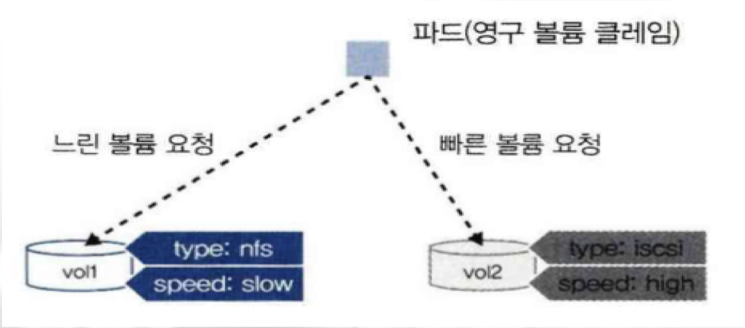
    - 리소스 생성: kubectl apply -f pv.yaml
    - 리소스 확인: kubectl get persistentvolumes

```
NAME       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM   STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
local-pv   5Gi        RWO            Retain           Available
manual         <unset>                          13s
```

Storage API

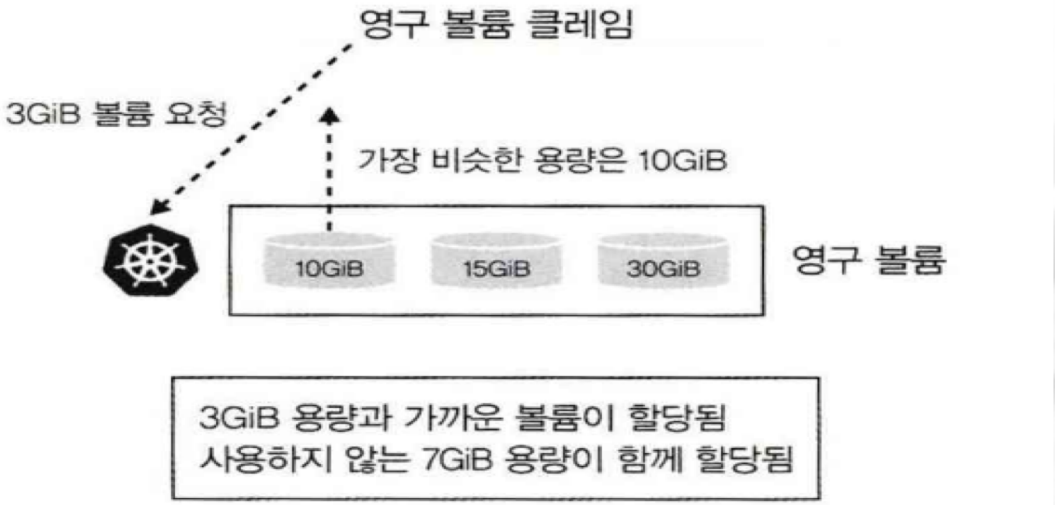
- ❖ 영구 볼륨
  - ✓ 영구 볼륨 생성
    - ● 설정 항목
      - ◆ 레이블
        - ▪ 동적 프로비저닝을 사용하지 않고 영구 볼륨을 생성하는 경우 영구 볼륨 종류를 알 수 없게 되기때문에 type/environment/speed 등의 레이블을 사용하는 것을 추천
        - ▪ 레이블을 사용하지 않을 경우 영구 볼륨에서 할당할 때 사용자가 원하는 볼륨을 지정하기가 어려움
        - ▪ 레이블을 사용하면 영구 볼륨 클레임에서 레이블로 볼륨을 지정할 수 있기 때문에 사용 목적에 맞도록 유연하게 스케줄링을 할 수 있음





Storage API

- ❖ 영구 볼륨
  - ✓ 영구 볼륨 생성
    - ● 설정 항목
      - ◆ 용량
        - ▪ 용량을 지정할 때 주의해야 할 점은 동적 프로비저닝을 사용할 수 없는 환경에서는 작은 용량의 영구 볼륨도 준비해 두어야 한다는 것
        - ▪ 영구 볼륨 클레임 요청이 3GiB인 경우에는 준비된 영구 볼륨에서 가장 비슷한 용량인 10GiB 볼륨이 할당됨



Storage API

- ❖ 영구 볼륨
  - ✓ 영구 볼륨 생성
    - ● 설정 항목
      - ◆ 접근 모드
        - ▪ ReadWriteOnce(RWO): 단일 노드에서 Read/Write 가능
          - 같은 노드에서 여러 파드가 기동한다면 여러 파드에서 Read/Write할 수 있는 경우가 있는데 쿠버네티스에서는 최대한 노드를 의식하지 않고 개발하는 것이 바람직하므로 단일 파드로 생각하는 것이 좋음
        - ▪ ReadOnlyMany(ROX): 여러 노드에서 Read 가능
          - 여러 노드에서 Read를 허용하는 접근 모드로 하나라도 쓰기 요청이 있는 파드가 있다면 다른 노드에서 마운트가 불가능하기 때문에 파드에서 영구 볼륨 클레임을 지정할 때 readonly로 지정
        - ▪ ReadWriteMany(RWX): 여러 노드에서 Read/Write 가능
          - 여러 노드에서 Read와 Write를 허용하는 접근 모드
          - 주로 nfs와 같은 영속성 영역에서 사용

Storage API

- ❖ 영구 볼륨
  - ✓ 영구 볼륨 생성
    - ● 설정 항목
      - ◆ Reclaim Policy
        - ▪ Reclaim Policy는 영구 볼륨을 사용한 후 처리 방법(삭제 또는 재사용 등)을 제어하는 정책
        - ▪ 영구 볼륨 클레임에서 사용된 후 그 영구 볼륨 클레임이 삭제되었을 때 영구 볼륨 자체의 동작을 설정
        - ▪ spec.persistentVolumeReclaimPolicy를 설정하는 방법
          - Delete: 영구 볼륨 자체가 삭제되는 것으로 GCP/AWS/OpenStack 등에서 확보되는 외부 볼륨의 동적 프로비저닝 때 사용되는 경우가 많음
          - Retain: 영구 볼륨 자체를 삭제하지 않고 유지하는 것으인데 다른 영구 볼륨 클레임에 의해 이 영구 볼륨이 다시 마운트되지 않음
          - Recycle: 영구 볼륨 데이터를 삭제(rm -rf ./*)하고 재사용 가능한 상태로 만드는 것으로 다른 영구 볼륨 클레임에서 다시 마운트 가능한데 이 정책은 쿠버네티스에서 더 이상 사용하지 않으므로 대신 동적 프로비저닝을 사용하길 권장

Storage API

- ❖ 영구 볼륨
  - ✓ 영구 볼륨 생성
    - ● 설정 항목
      - ◆ StorageClass
        - ▪ 이 필드는 PVC(Persistent Volume Claim)와 StorageClass를 연결하는 고리 역할
        - ▪ 선택지 제공: 관리자가 fast-ssd, slow-hdd라는 두 개의 StorageClass를 만들어 두었다면 개발자는 storageClassName: fast-ssd라고 명시하여 원하는 성능의 디스크를 할당받을 수 있음
        - ▪ 자동 매칭: 이 이름이 지정되면 쿠버네티스는 해당 이름을 가진 StorageClass를 찾아가서, 그 안에 정의된 Provisioner(Cinder 등)를 실행해 실제 볼륨을 생성
      - ◆ 마운트 옵션: 영구 볼륨의 종류에 따라 spec.mountOptions를 설정하여 마운트 옵션을 추가로 설정할 수 있음

Storage API

- ❖ 영구 볼륨
  - ✓ 영구 볼륨 생성
    - ● 설정 항목
      - ◆ 영구 볼륨 플러그인 특유의 설정
        - ▪ 설정 항목은 영구 볼륨 플로그인 마다 다름
        - ▪ gcePersistentDisk의 경우

```
gcePersistentDisk:
  pdName: sample-gce-pv
  fsType: ext4
```

        - ▪ nfs의 경우

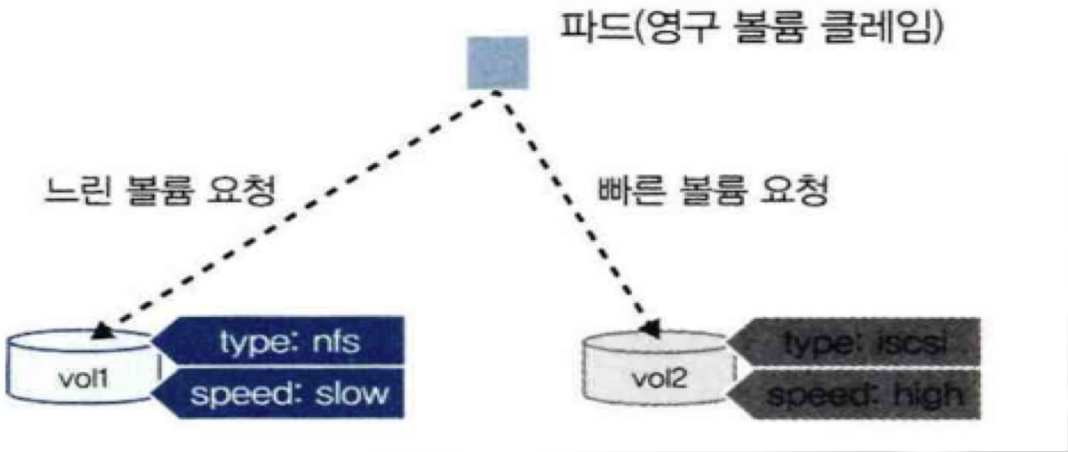
```
nfs:
  server: xxx.xxx.xxx.xxx
  path: /nfs/sample
```

        - ▪ 호스트 컴퓨터

```
hostPath:
  path: "/home/user/data" # 실제 호스트 컴퓨터의 경로
```

Storage API

- ❖ 영구 볼륨 클레임
  - ✓ 개요
    - 영구 볼륨 클레임(PVC)은 영구 볼륨을 요청하는 리소스
    - 영구 볼륨 클레임에서 지정된 조건(용량, 레이블)을 기반으로 영구 볼륨에 대한 요청이 들어오면 스케줄러는 현재 가지고 있는 영구 볼륨에서 적당한 볼륨을 할당



Storage API

- ❖ 영구 볼륨 클레임
  - ✓ 설정
    - 설정 항목
      - ◆ 레이블 셀렉터
      - ◆ 용량
      - ◆ 접근 모드
      - ◆ 스토리지 클래스
    - 모두 영구 볼륨에 정의한 값이며 이 영구 볼륨 클레임 요청에 일치하는 영구 볼륨이 할당되는데 이때 주의해야 할 것은 영구 볼륨 클레임 용량이 영구 볼륨 용량보다 작으면 할당된다는 점으로 3GiB 요청에 대해 3GiB 영구 볼륨이 있다면 할당되지만 만약 10GiB 영구 볼륨밖에 없는 경우라면 10GiB 영구 볼륨이 할당되며 또한 nfs 플러그인에서는 용량 제한 구조를 지원하지 않아 영구 볼륨 생성 시 용량 지정이 안 되는 등의 플러그인 별 제약 사항에도 주의

Storage API

- ❖ 영구 볼륨 클레임
  - ✓ 생성
    - 매니페스트: pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: local-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: manual # PV의 StorageClass와 일치해야 함
```

    - 리소스 생성: kubectl apply -f pvc.yaml
    - 영구 볼륨 클레임 정보 확인: kubectl get persistentvolumeclaims local-pvc

```
NAME        STATUS   VOLUME     CAPACITY   ACCESS MODES
STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
local-pvc   Bound    local-pv   5Gi        RWO
manual         <unset>                 32s
```

Storage API

- ❖ 영구 볼륨 클레임
  - ✓ 생성
    - ● 영구 볼륨 정보 확인: kubectl get persistentvolumes local-pv

```
NAME       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
local-pv   5Gi        RWO            Retain           Bound    default/local-pvc
manual         <unset>                          49m
```

Storage API

- ❖ 영구 볼륨 클레임
  - ✓ 파드에서 사용
    - ● 매니페스트: pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: local-pod
spec:
  containers:
    - name: nginx
      image: nginx
      volumeMounts:
        - mountPath: "/usr/share/nginx/html" # 컨테이너 내부 마운트 경로
          name: local-storage
  volumes:
    - name: local-storage
      persistentVolumeClaim:
        claimName: local-pvc # 위에서 생성한 PVC 이름
```

Storage API

- ❖ 영구 볼륨 클레임
  - ✓ 파드에서 사용
    - 리소스 생성: kubectl apply -f pod.yaml
    - 파드가 만들어진 노드 확인: kubectl get pods -o wide

```
NAME        READY   STATUS    RESTARTS   AGE     IP            NODE
NOMINATED NODE   READINESS GATES

local-pod   1/1     Running   0          2m41s   10.244.1.17   worker2
<none>           <none>
```

    - 파드가 만들어진 노드에서 파일을 생성: sudo nano /home/user/data/test.txt
    - 확인: kubectl exec -it local-pod -- cat /usr/share/nginx/html/test.txt