Section 8: Storage

199. Storage in Docker

- image, container 등 도커 데이터는 default로 `/var/lib/docker/containers,images,volumes` 에 저장됨
- 도커는 layered architecture로 빌드, 이미지 빌드 시 이전에 빌드한 이미지와 중복되는 부분은 캐시를 이용하여 시간, 공간적으로 효율적 사용을 한다.

Volume Mount

- `docker volume create data_volume`
  - `/var/lib/docker/volumes/data_volume` 에 볼륨 생성
- `docker run -v {docker host 내의 볼륨 경로}:{컨테이너 경로} {이미지명}`
  - ex) `docker run -v data_volume:/var/lib/mysql mysql`
  - 생성한 volume을 docker container 내에 마운트

Bind Mount

- `docker run -v {다른경로}:{컨테이너 경로} {이미지명}`
  - ex) `docker run -v /data/mysql:/var/lib/mysql mysql`
  - `/var/lib/docker/volumes` 가 아닌, docker host 내 다른 경로를 사용해서 docker container 내에 마운트
  - 요즘은 `--mount` 옵션 권장
    - ex) `docker run --mount type=bind, source=/data/mysql, target=/var/lib/mysql mysql`

CRI (Container Runtime Interface)

- CRI: 쿠버네티스같은 오케스트레이션 솔루션이 docker같은 컨테이너 런타임과 어떻게 통신할지 정의하는 표준
  - 어떤 새로운 CRI가 개발되더라도, CRI 표준을 따르기만 하면 그 Container Runtime은 쿠버네티스 소스코드에 의존하지 않고 작동할 수 있다.

203. Volumes

- pod가 데이터를 처리하고 삭제할 때 기본적으로 pod가 처리한 데이터도 삭제됨
  - 볼륨과 함께 쓰면 pod에서 생성된 데이터가 볼륨에 저장되어, pod가 삭제되어도 데이터는 남아 있음

```
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
```

```
  - image: alpine
    name: alpine
    command: ["/bin/sh", "-c"]
    args: ["shuf -i 0-100 -n 1 >> /opt/numver.out;"]
    volumeMounts:   # 어떤 볼륨을 어느 컨테이너 path에 마운트할 것인가 정의
    - mountPath: /opt   # 파드 내 데이터 저장 경로
      name: data-volume
  volumes:
  - name: data-volume
    hostPath:   # hostPath는 멀티 노드 클러스터인 경우 권장하지 않음
      path: /data   # 노드 내 데이터 저장 경로
      type: Directory
```

204. Persistent Volumes (PV)

- 크고 복잡한 환경에서 pod가 많으면 각 pod의 storage를 매번 구성해야됨
  - 관리자가 클러스터 레벨의 storage를 생성하고, 각각이 필요한 만큼 사용하는 방식이 Persistent Volumes

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-vol1
spec:
  accessModes:    # 볼륨이 호스트에 어떻게 마운트되어야 하는지 결정
  - ReadWriteOnce # ReadOnlyMany, ReadWriteMany
    capacity:
      storage: 1Gi
    hostPath: # volume type, 운영 환경에서는 지양
      path: /tmp/data   # 노드의 로컬 디렉토리 경로
```

205. Persistent Volume Claim (PVC)

- 관리자가 PV를 만들면, 사용자는 storage를 사용하기 위해 PVC를 생성하고, 쿠버네티스는 그 claim에 persistent volume을 binding

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessMode:
  - ReadWriteOnce
  resources:
```

```
    requests:
      storage: 500Mi
```

- pvc 생성 확인: `kubectl get persistentvolumeclaim`
- pvc 삭제: `kubectl delete persistentvolumeclaim myclaim`
  - pv yaml 의 `persistentVolumeReclaimPolicy` 를 통해 PVC가 삭제 될 때 pv의 동작 지정 가능
    - Retain(default): 관리자가 삭제할 때까지 PV는 남아있음
    - Delete: PVC가 삭제될 때 PV도 함께 삭제
    - Recycle: volume을 다른 claim이 사용할 수 있도록 volume 내의 이전 데이터가 삭제됨

- pod yaml에서 PVC 지정: `persistentVolumeClaim`

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: myfrontend
    image: nginx
    volumeMounts:
    - mountPath: "/var/www/html"
      name: mypd
  volumes:
  - name: mypd
    persistentVolumeClaim:
      claimName: myclaim
```

211. Storage Class

- 정적 프로비저닝
  - pv를 생성할 때마다 수동으로 클라우드 디스크를 생성하고, pv를 정의해야 함
- 동적 프로비저닝
  - Storage Class를 통해, Google Storage와 같은 프로비저너를 정의하면 클레임이 생성될 때 자동으로 스토리지를 프로비저닝함

Storage Class 구성 방법

1. Storage Class yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
```

```
metadata:
  name: google-storage
provisioner: kubernetes.io/gce-pd
parameters: # provisioner에 따라 달라질 수 있음
  type: pd-standard
    replication-type: none
```

2. PVC yaml - `storageClassName`

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessMode:
  - ReadWriteOnce
  storageClassName: google-storage   # 1번에서 생성한 Storage Class 사용
  resources:
    requests:
      storage: 500Mi
```

3. pod에서 pvc mapping

```
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
  - image: alpine
    name: alpine
    command: ["/bin/sh", "-c"]
    args: ["shuf -i 0-100 -n 1 >> /opt/numver.out;"]
    volumeMounts:
    - mountPath: /opt
      name: data-volume
    volumes:
    - name: data-volume
      persistentVolumeClaim:
        claimName: myclaim
```