Kubernetes 5

Volume & StatefulSet

caodangsao@gmail.com

Module Target

Kết thúc bài học, học viên cần đạt được các kĩ năng sau:

- Có thể hiểu và tạo được Volume trong Kubernetes.
- Nắm được các đối tượng để khởi tạo Volume như StorageClass, PersistentVolume và PersistentVolumeClaim.
- Có khả năng triển khai ứng dụng dưới dạng Statefulsets.

Nội dung

1 Volume in K8s

- Quản lý cấu hình ứng dụng
- Volume là gì
- Volume Type

2 PV, PVC and Statefulset

- PV, PVC
- Storage Class
- Statefulset

3 Summary

- Tóm tắt nội dung buổi học

Lab 0: Manage Application Configuration

Lab 1: Sharing a volume, in action

Lab 2: Volume type: hostPath

Lab 3: Using PV, PVC

Lab 4: StatefulSets

Section 1:

Volume in K8s

Manage Application Configuration

- ➔ Cấu hình ứng dụng là gì ?
- ➔ ConfigMaps
- ➔ Secrets
- ➔ Environment Variables
- ➔ Configuration Volumes
- ➔ Lab demo

Application Configuration

- ➔ Ta có thể hiểu "cấu hình" là các cài đặt ảnh hưởng đến hoạt động của ứng dụng.
- ➔ Kubernetes cho phép người dùng chuyển các giá trị cấu hình động cho ứng dụng lúc Runtime.
- ➔ Các cấu hình động này giúp người dùng kiểm soát luồng ứng dụng.

ConfigMap

- ➔ Giữ dữ liệu không nhạy cảm trong ConfigMap, dữ liệu này có thể được chuyển đến container của ứng dụng.
- ➔ Config Map lưu trữ data ở định dạng Key-Value.
- ➔ ConfigMaps cho phép ta tách các cấu hình của mình khỏi Pod và các thành phần.
- ➔ ConfigMap giúp làm cho các cấu hình dễ thay đổi và quản lý hơn, đồng thời ngăn mã hóa cứng dữ liệu cấu hình thành các thông số kỹ thuật của Pod.

ConfigMap Commands

➔ Tạo ConfigMaps qua File cấu hình.

```
kubectl create configmap [NAME] --from-file [/PATH/TO/FILE.PROPERTIES] --from-file
[/PATH/TO/FILE2.PROPERTIES]
```

```
kubectl create configmap [NAME] --from-file [/PATH/TO/DIRECTORY]]
```

➔ Lấy ConfigMap qua CLI

```
kubectl get configmap <Config_map_name> -o yaml/json
```

Secrets

➔ Secrets tương tự như ConfigMap nhưng được thiết kế để giữ các dữ liệu nhạy cảm.

➔ Tạo Secrets từ file.

```
kubectl create secret generic db-user-pass --from-file =./username.txt --from-file
=./password.txt
```

➔ Lưu ý: Các ký tự đặc biệt như $,\,* và ! yêu cầu escaping.

Secrets

➔ Lấy Secrets.

```
kubectl get secrets
```

➔ Mô tả Secrets.

```
kubectl describe secrets <Secret_name>
```

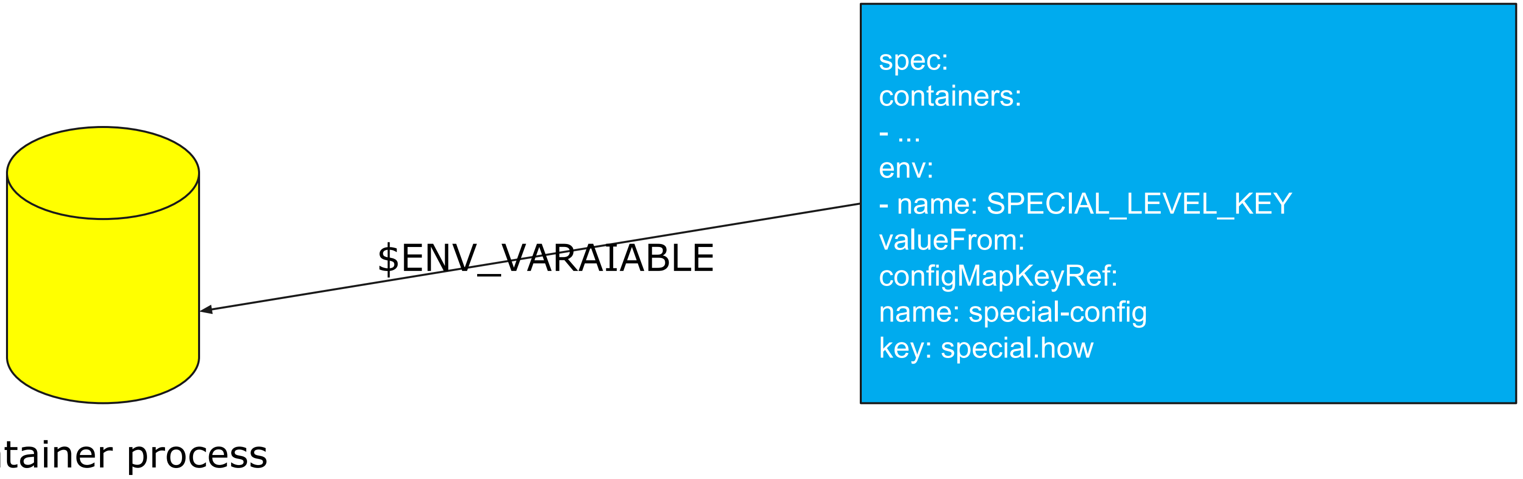
K8s Secrets

➔ Ví dụ mẫu về file YAML Secrets.

```
apiVersion: v1
kind: Secret
metadata:
name: mysecret-manifest
type: Opaque
data:
username: YW5zaHVsY2hhdWhhbg==
password: VGVzdGt1VybmV0ZXMxMjM0NQ==
```

Pass ConfigMap and Secrets to Containers

➔ User có thể chuyển Secrets và Config Map đến Container sử dụng Environment Variables.



```
spec:
containers:
- ...
env:
- name: SPECIAL_LEVEL_KEY
valueFrom:
configMapKeyRef:
name: special-config
key: special.how
```

Pass ConfigMap and Secrets to Containers

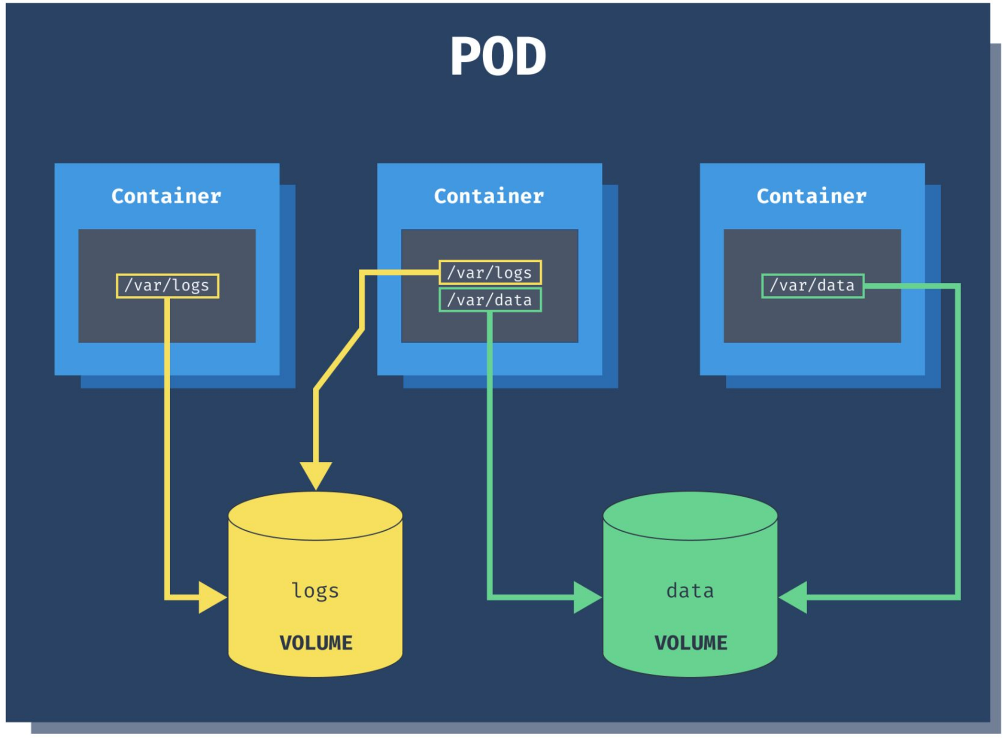
- ➔ Cấu hình Mount Volume là cách khác để chuyển dữ liệu Config và Secrets đến Container.
- ➔ Dữ liệu Config sẽ hiển thị dưới dạng Files trong hệ thống file của Container

```
volumes:
- name: config-volume
configMap:
name: special-config
```

Labs

Lab 0: Manage Application Configuration

Volume


POD
Container
/var/logs
Container
/var/logs
/var/data
Container
/var/data
logs
VOLUME
data
VOLUME

Volume

- Volumes are special directories that are mounted in containers
- Volumes can have many different purposes:
    - share files and directories between containers running on the same machine
    - share files and directories between containers and their host
    - centralize configuration information in Kubernetes and expose it to containers
    - manage credentials and secrets and expose them securely to containers
    - store persistent data for stateful services
    - access storage systems (like Ceph, EBS, NFS, Portworx, and many others)

A simple volume example

```
apiVersion: v1
kind: Pod
metadata:
    name: nginx-with-volume
spec:
    volumes:
    - name: www
    containers:
    - name: nginx
        image: nginx
        volumeMounts:
        - name: www
        mountPath: /usr/share/nginx/html/
```

A volume shared between two containers

```
apiVersion: v1
kind: Pod
metadata:
	name: nginx-with-volume
spec:
	volumes:
	- name: www
	containers:
	- name: nginx
	  image: nginx
	  volumeMounts:
	  - name: www
	    mountPath: /usr/share/nginx/html/
	- name: git
	  image: alpine
	  command: [ "sh", "-c", "apk add --no-cache git && git clone https://github.com/octocat/Spoon-Knife /www" ]
	  volumeMounts:
	- name: www
	  mountPath: /www/
	restartPolicy: OnFailure
```

Labs

Lab 1: Sharing a volume, in action

Let's try it!

1. Create the pod by applying the YAML file:

```
kubectl apply -f nginx-with-volume.yaml
```

2. Check the IP address that was allocated to our pod:

```
kubectl get pod nginx-with-volume -o wide
IP=$(kubectl get pod nginx-with-volume -o json |
jq -r .status.podIP)
```

3. Access the web server:

```
curl $IP
```

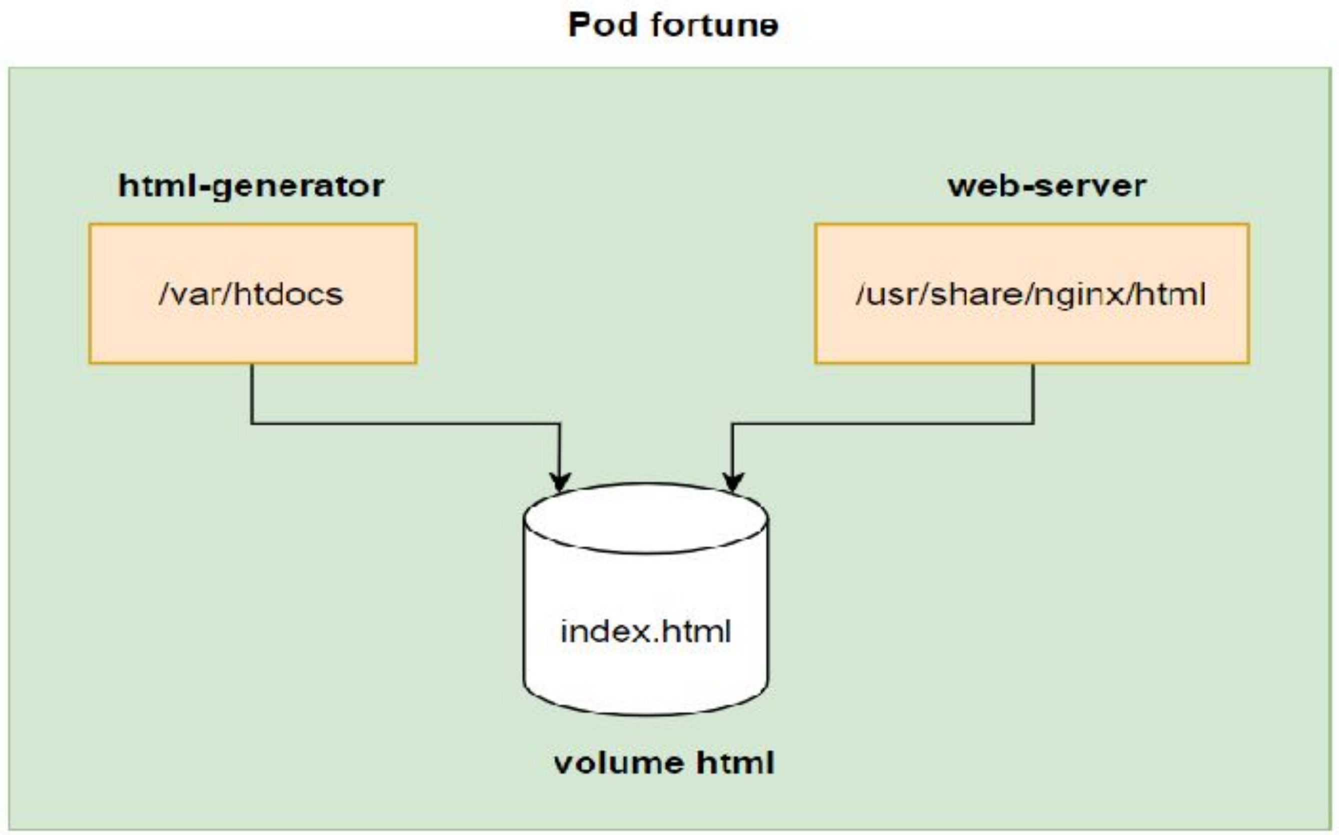
nginx-with-volume.yaml

```
apiVersion: v1
kind: Pod
metadata:
    name: nginx-with-volume
spec:
    volumes:
    - name: www
    containers:
    - name: nginx
        image: nginx
        volumeMounts:
        - name: www
        mountPath:
        /usr/share/nginx/html/
```

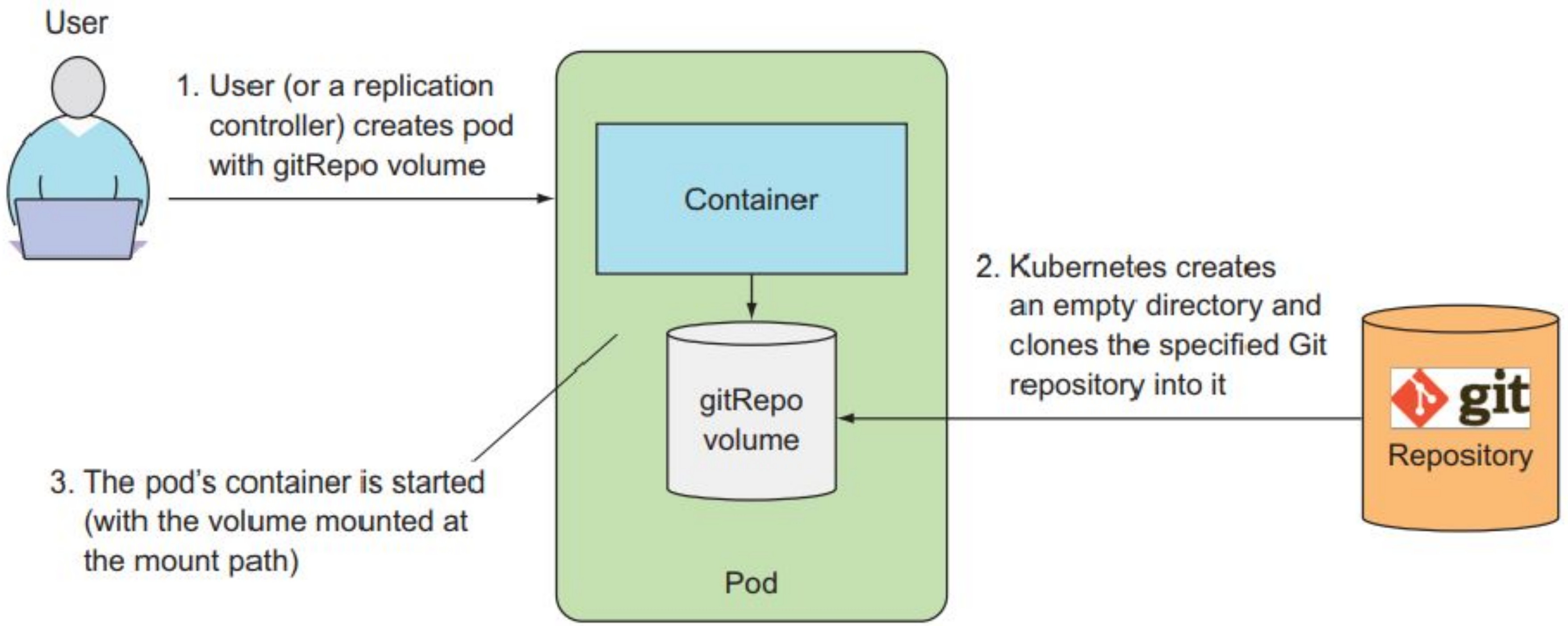
Volume Types

Temp	Local	Network
emptyDir	hostPath	GlusterFS gitRepo NFS iSCSI gcePersistentDisk AWS EBS azureDisk Fiber Channel Secret VshereVolume

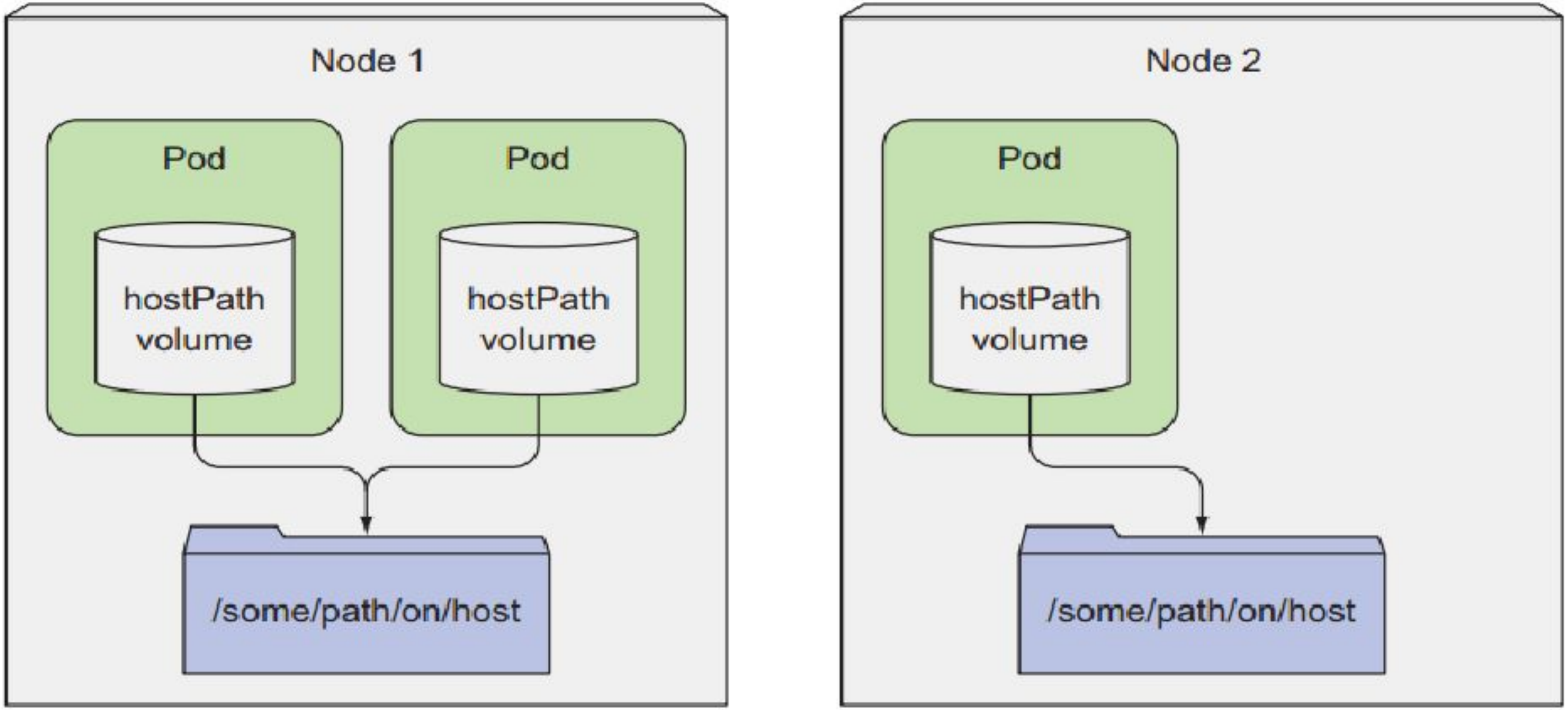
Volumes: EmptyDir



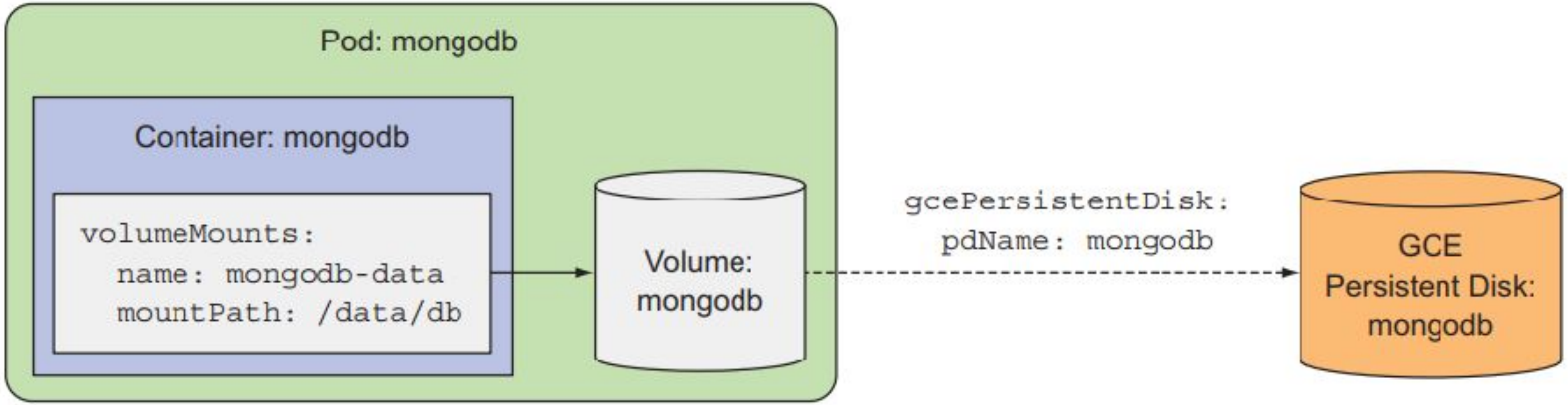
Volumes: Git repo


User
1. User (or a replication controller) creates pod with gitRepo volume
Container
gitRepo volume
Pod
2. Kubernetes creates an empty directory and clones the specified Git repository into it
git
Repository
3. The pod's container is started (with the volume mounted at the mount path)

Volumes: hostPath



Volumes: Cloud storage



```
volumeMounts:
  name: mongodb-data
  mountPath: /data/db
```

```
gcePersistentDisk:
  pdName: mongodb
```

Volume lifecycle

- The lifecycle of a volume is linked to the pod's lifecycle
- This means that a volume is created when the pod is created
- This is mostly relevant for emptyDir volumes
  (other volumes, like remote storage, are not "created" but rather "attached")
- A volume survives across container restarts
- A volume is destroyed (or, for remote storage, detached) when the pod is destroyed

Labs

Lab 2: Volume type: hostPath

Let's try it!

1. Create the pod by applying the YAML file:

```
kubectl apply -f
pod-volume-hostPath.yaml
```

2. Check the IP address that was allocated to our pod:

```
kubectl get pod test-pd -o wide
IP=$(kubectl get pod nginx-with-volume
-o json | jq -r .status.podIP)
```

3. Access the web server:

```
curl $IP
```

pod-volume-hostPath.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # directory location on host
      path: /data
      # this field is optional
      type: DirectoryOrCreate
```

Section 2:

PV, PVC and Statefulset

Volumes and Persistent Volumes

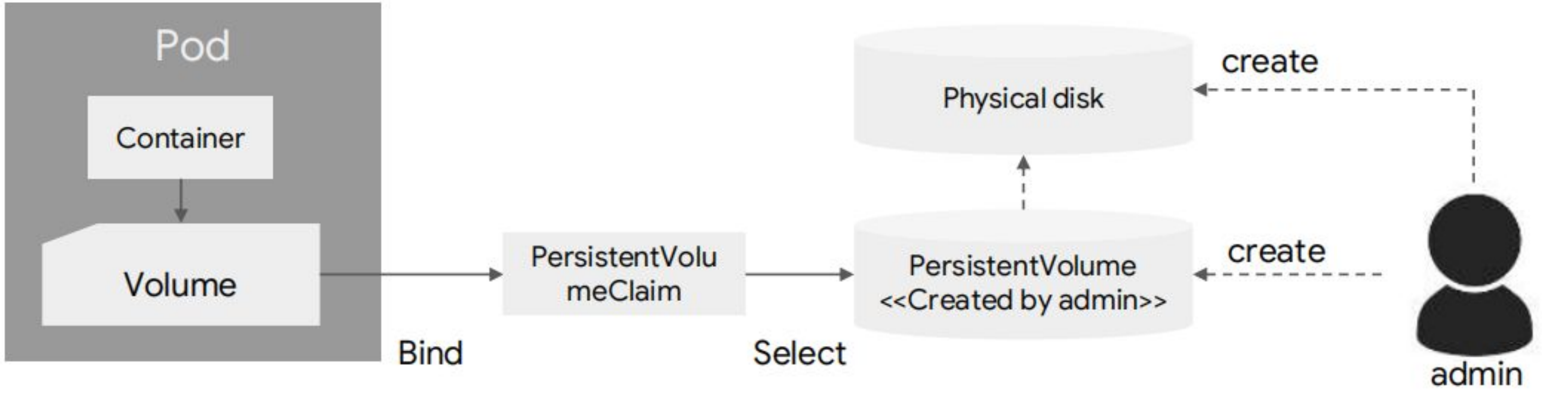
- Volumes are used for many purposes:
  - sharing data between containers in a pod
  - exposing configuration information and secrets to containers
  - accessing storage systems
- The last type of volumes is known as a "Persistent Volume"

Persistent Volumes types

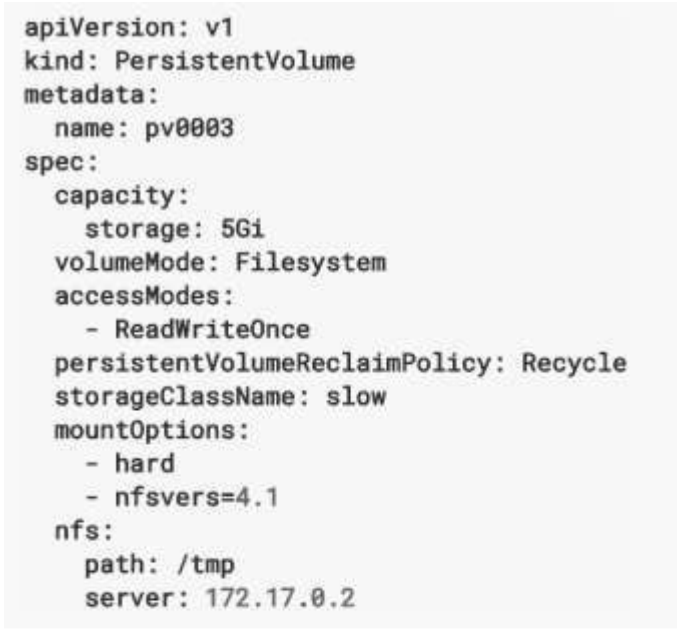
- There are many types of Persistent Volumes available:
  - public cloud storage (GCEPersistentDisk, AWSElasticBlockStore, AzureDisk...)
  - private cloud storage (Cinder, VsphereVolume...)
  - traditional storage systems (NFS, iSCSI, FC...)
  - distributed storage (Ceph, Glusterfs, Portworx...)
- Using a persistent volume requires:
  - creating the volume out-of-band (outside of the Kubernetes API)
  - referencing the volume in the pod description, with all its parameters

PersistentVolume

- Life cycle is managed by k8s cluster (not pod)
- Admin can create PersistentVolume (Static provisioning) and developer just use the volume through PersistentVolumeClaim without understanding of infrastructure (It is more common to use dynamic volume provisioning instead of volume provisioning by admin)



PersistentVolume



```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /tmp
    server: 172.17.0.2
```

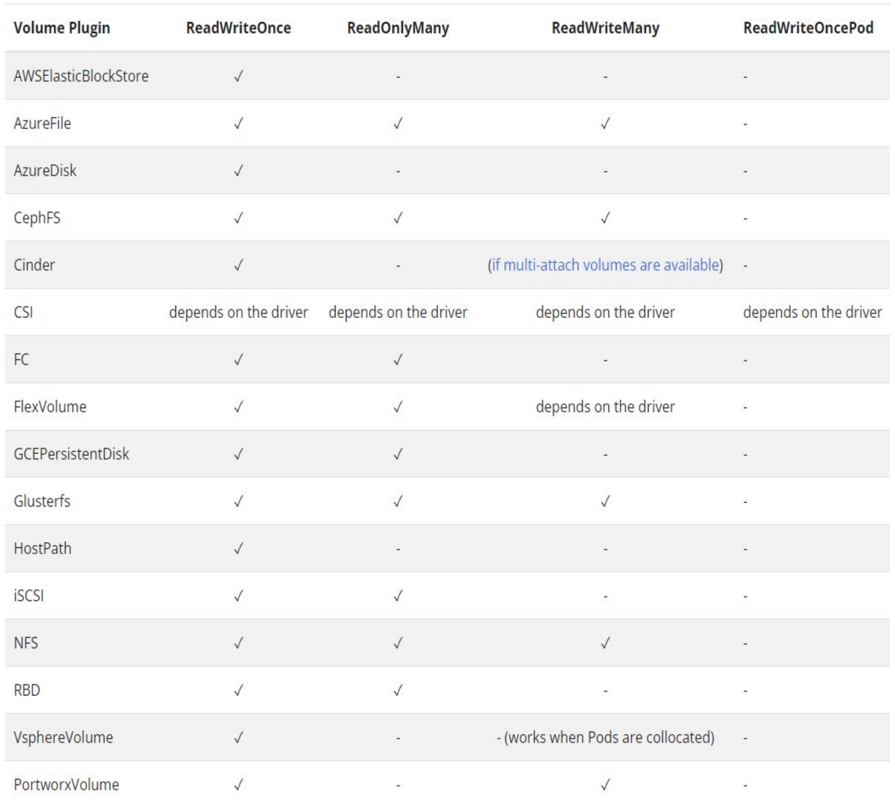
- Capacity : Storage size
  In future it will include IOPS,throughput etc
- VolumeMode (from 1.9) : Filesystem(default) or rawblock device
- Reclaim Policy
  - Retain – manual reclamation
  - Recycle – basic scrub (rm -rf /thevolume/*)
  - Delete – associated storage asset such as AWS EBS, GCE PD, Azure Disk, or OpenStack Cinder volume is deleted

  Currently, only NFS and HostPath support recycling. AWS EBS, GCE PD, Azure Disk, and Cinder volumes support deletion.
- Mount Option
  Additional mount options for when a Persistent Volume is mounted on a node

PersistentVolume

- AccessModes
  - ReadWriteOnce (RWO)– the volume can be mounted as read-write by a single node
  - ReadOnlyMany (ROX) – the volume can be mounted read-only by many nodes
  - ReadWriteMany (RWX) – the volume can be mounted as read-write by many nodes

Note: A volume can only be mounted using one access mode at a time, even if it supports many. For example, a GCEPersistentDisk can be mounted as ReadWriteOnce by a single node or ReadOnlyMany by many nodes, but not at the same time.



Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany	ReadWriteOncePod
AWSElasticBlockStore	✓	-	-	-
AzureFile	✓	✓	✓	-
AzureDisk	✓	-	-	-
CephFS	✓	✓	✓	-
Cinder	✓	-	(if multi-attach volumes are available)	-
CSI	depends on the driver	depends on the driver	depends on the driver	depends on the driver
FC	✓	✓	-	-
FlexVolume	✓	✓	depends on the driver	-
GCEPersistentDisk	✓	✓	-	-
Glusterfs	✓	✓	✓	-
HostPath	✓	-	-	-
iSCSI	✓	✓	-	-
NFS	✓	✓	✓	-
RBD	✓	✓	-	-
VsphereVolume	✓	-	- (works when Pods are collocated)	-
PortworxVolume	✓	-	✓	-

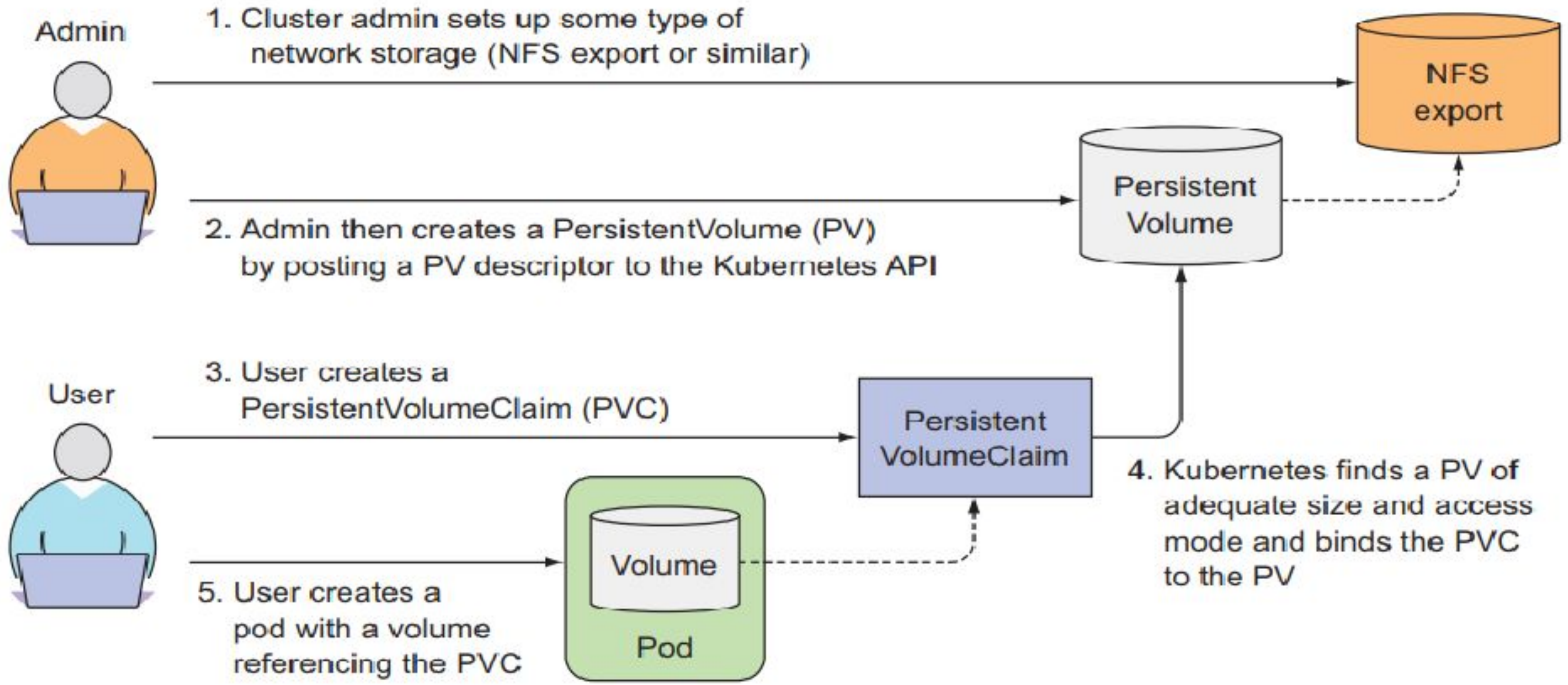
PersistentVolume Phase

- Available - a free resource that yet is not bound to a claim
- Bound - the volume is bound to a claim
- Released - the claim has been deleted but the resource is not yet reclaimed by the cluster
- Failed - the volume has failed its automatic reclamation

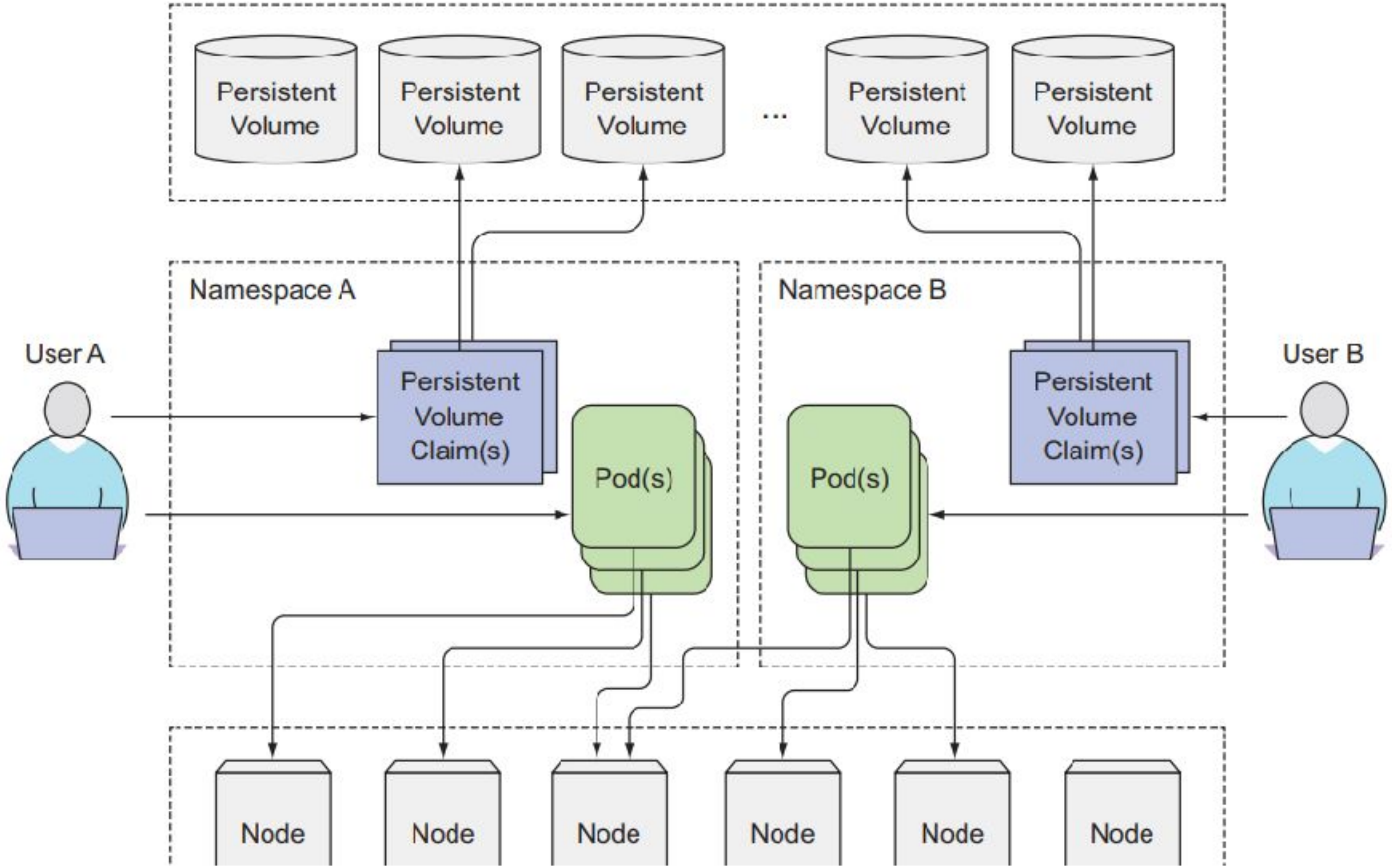
Lifecycle of volume and claim

1. Provisioning
   - Static
   - Dynamic : Dynamically create volume with storage class.
2. Binding
   - Bind PV to PVC
3. Using
   - Bind PVC to Pod and starts to use
4. Reclaiming

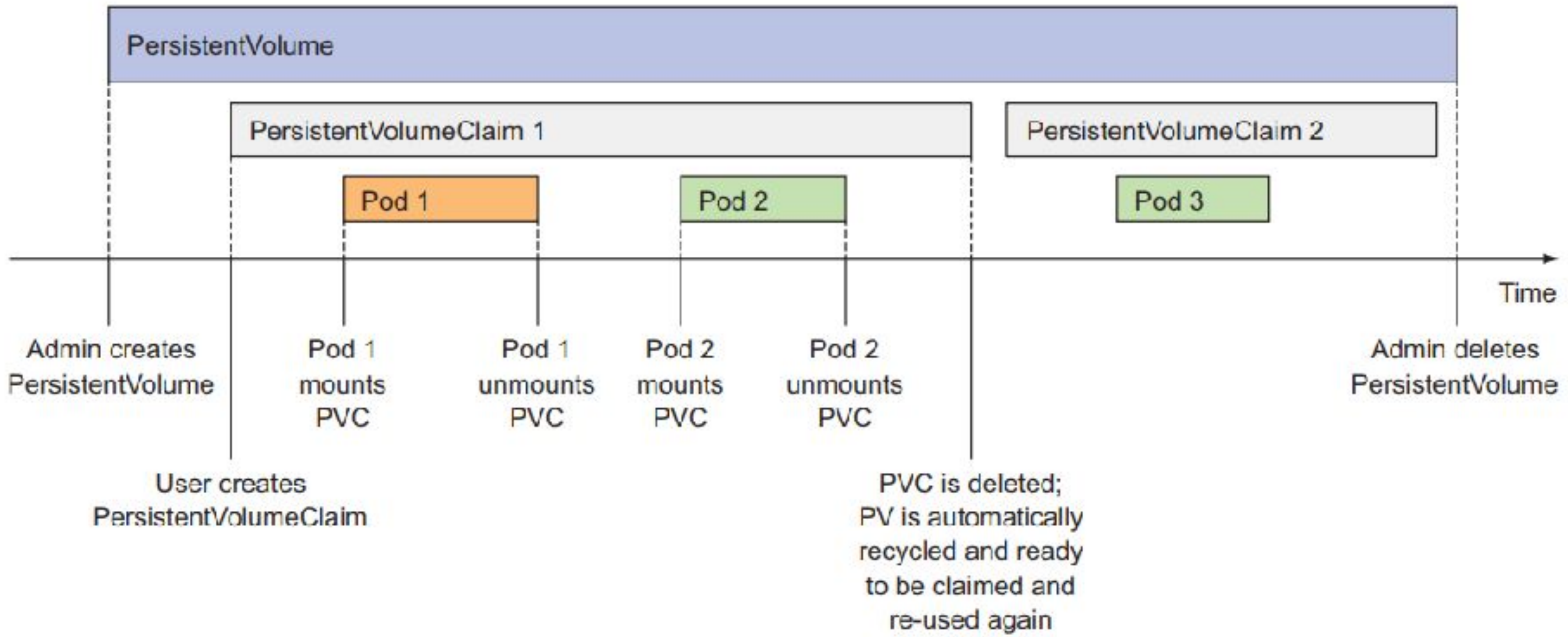
PersistentVolumeClaim (PVC)


Admin
1. Cluster admin sets up some type of network storage (NFS export or similar)
NFS export
Persistent Volume
2. Admin then creates a PersistentVolume (PV) by posting a PV descriptor to the Kubernetes API
User
3. User creates a PersistentVolumeClaim (PVC)
Persistent VolumeClaim
4. Kubernetes finds a PV of adequate size and access mode and binds the PVC to the PV
Volume
Pod
5. User creates a pod with a volume referencing the PVC

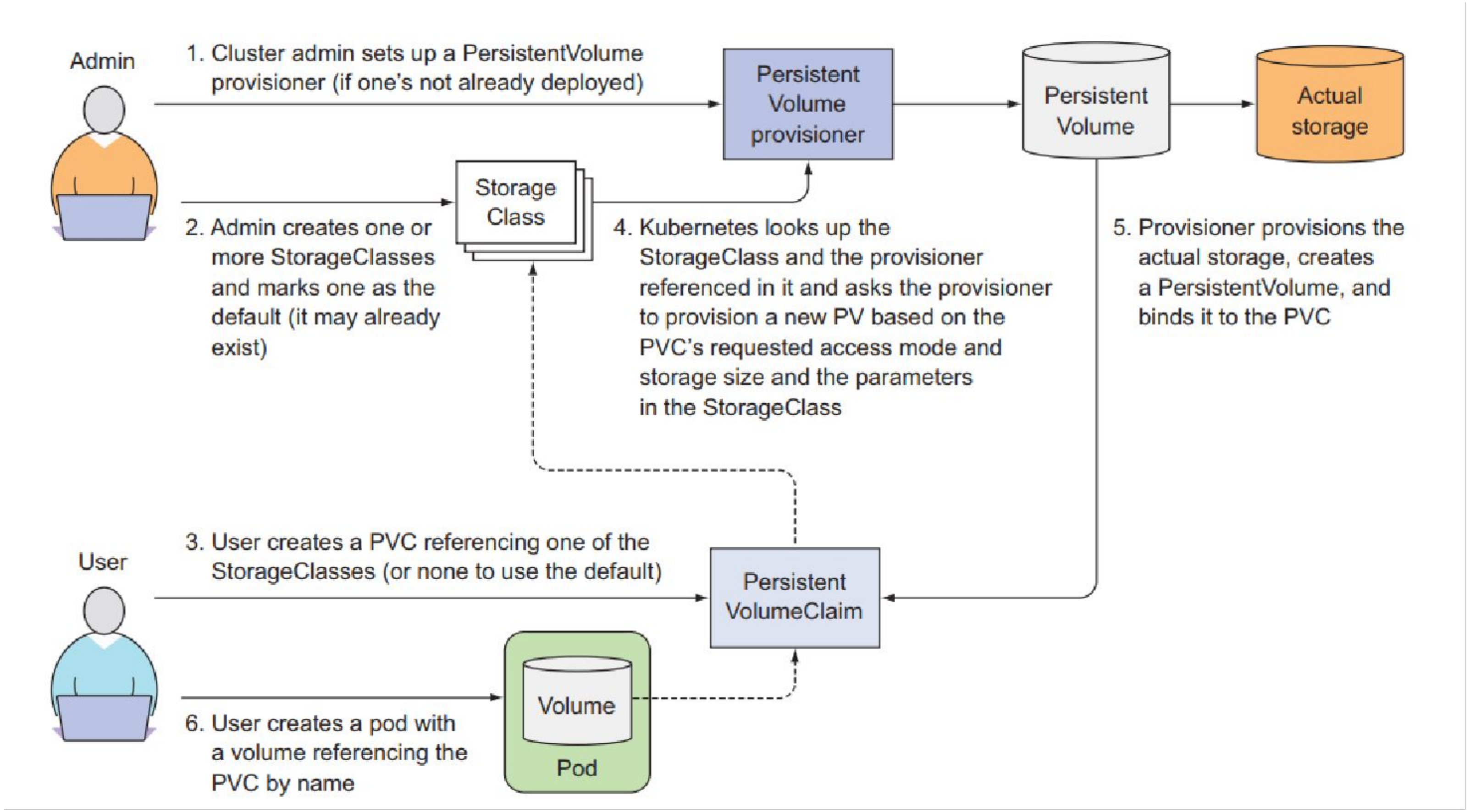
PersistentVolumeClaim (PVC)



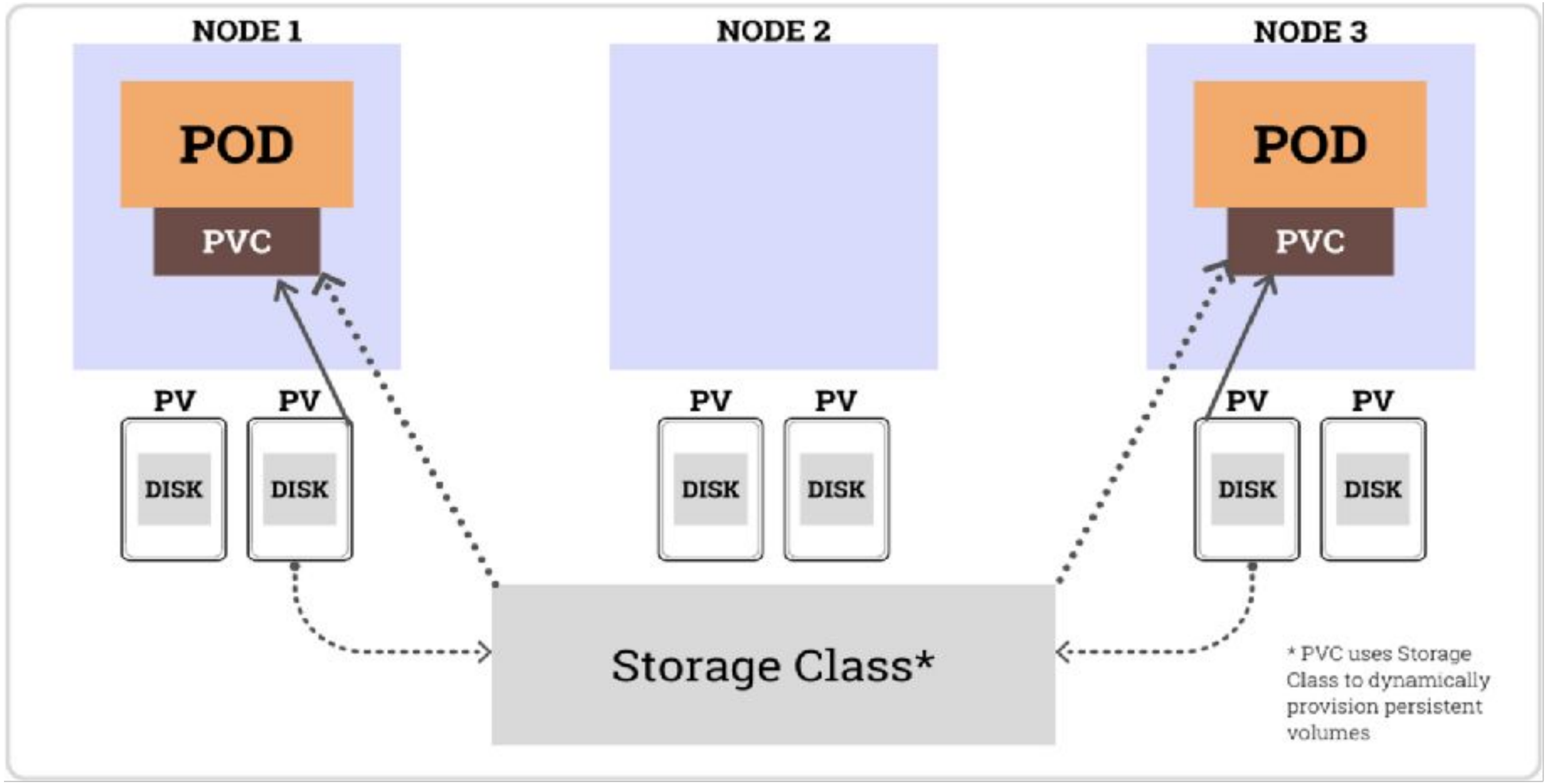
Recycling PersistentVolumes


PersistentVolume
PersistentVolumeClaim 1
PersistentVolumeClaim 2
Pod 1
Pod 2
Pod 3
Time
Admin creates PersistentVolume
User creates PersistentVolumeClaim
Pod 1 mounts PVC
Pod 1 unmounts PVC
Pod 2 mounts PVC
Pod 2 unmounts PVC
PVC is deleted; PV is automatically recycled and ready to be claimed and re-used again
Admin deletes PersistentVolume

StorageClass


Admin
1. Cluster admin sets up a PersistentVolume provisioner (if one's not already deployed)
Persistent Volume provisioner
Persistent Volume
Actual storage
Storage Class
2. Admin creates one or more StorageClasses and marks one as the default (it may already exist)
4. Kubernetes looks up the StorageClass and the provisioner referenced in it and asks the provisioner to provision a new PV based on the PVC's requested access mode and storage size and the parameters in the StorageClass
5. Provisioner provisions the actual storage, creates a PersistentVolume, and binds it to the PVC
User
3. User creates a PVC referencing one of the StorageClasses (or none to use the default)
Persistent VolumeClaim
Volume
Pod
6. User creates a pod with a volume referencing the PVC by name

StorageClass



Labs

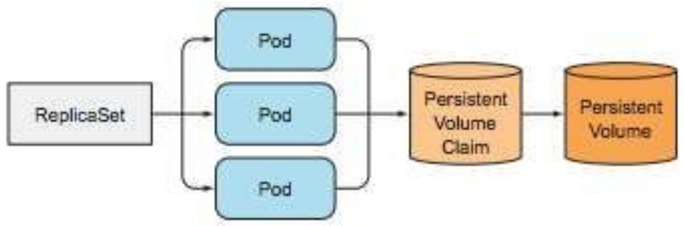
Lab 3: Using PV, PVC

Shortcomings of Persistent Volumes

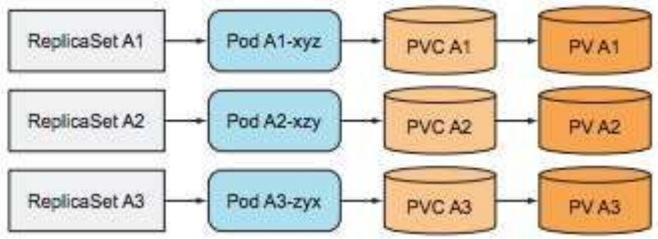
- Their lifecycle (creation, deletion...) is managed outside of the Kubernetes API (we can't just use kubectl apply/create/delete/... to manage them)
- If a Deployment uses a volume, all replicas end up using the same volume
- That volume must then support concurrent access
  - some volumes do (e.g. NFS servers support multiple read/write access)
  - some volumes support concurrent reads
  - some volumes support concurrent access for colocated pods
- What we really need is a way for each replica to have its own volume

Replicating stateful pod

- With ReplicaSet



- Option : Create RS per Pod



Statefulset

- Stateful sets are a type of resource in the Kubernetes API (like pods, deployments, services...)
- They offer mechanisms to deploy scaled stateful applications
- At a first glance, they look like *deployments*:
  - a stateful set defines a pod spec and a number of replicas *R*
  - it will make sure that *R* copies of the pod are running
  - that number can be changed while the stateful set is running
  - updating the pod spec will cause a rolling update to happen

- But they also have some significant differences

Stateful sets unique features

- Pods in a stateful set are numbered (from 0 to *R-1*) and ordered
- They are started and updated in order (from 0 to *R-1*)
- A pod is started (or updated) only when the previous one is ready
- They are stopped in reverse order (from *R-1* to 0)
- Each pod know its identity (i.e. which number it is in the set)
- Each pod can discover the IP address of the others easily
- The pods can have persistent volumes attached to them

Statefulset

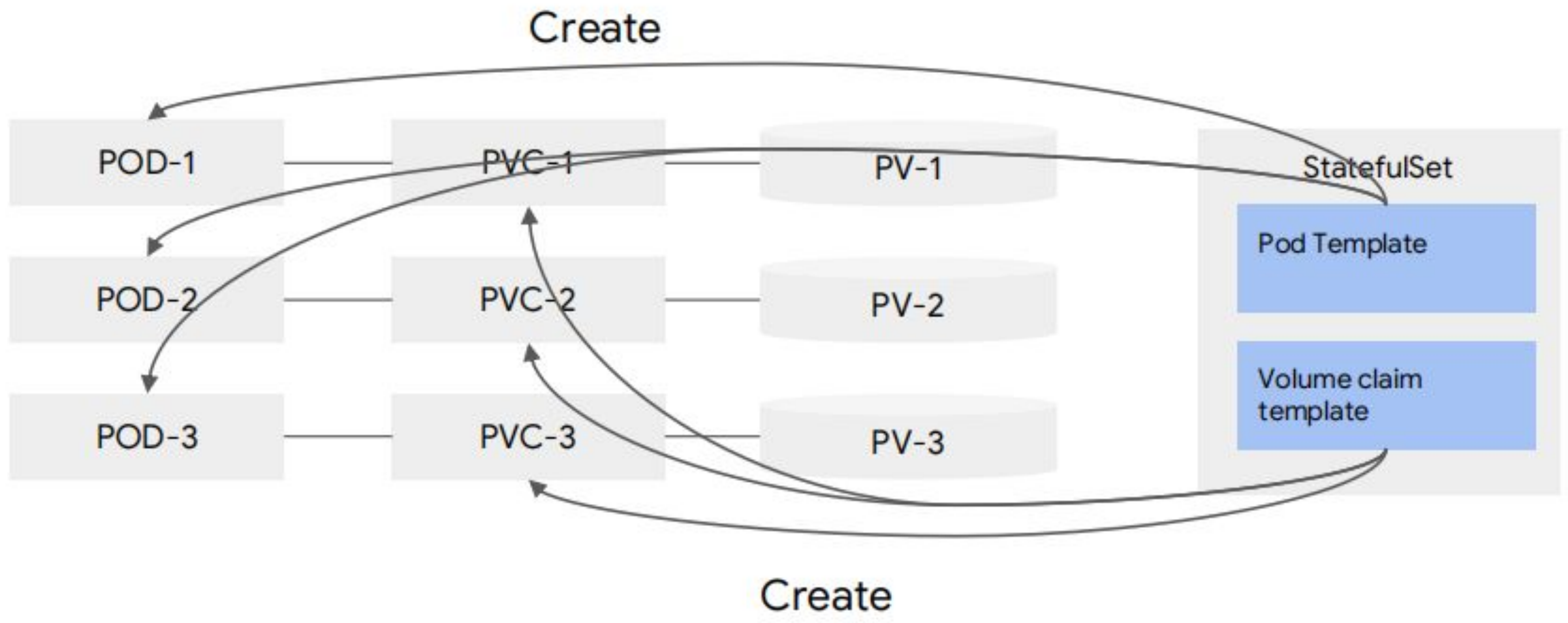
- To support stateful application (like database)
- It is supported from k8s 1.9(GA)
- Pod Naming : ${Stateful set name}-${ordinary index}
  It is not that the pods having a predictable name and hostname.
  It is better to use service for naming.
- When Pod (under statefulset) is restarted (by crash), Pod name will not be changed. Not like RS, the Pod name is changed to new one.

StatefulSet specification example

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
 name: web
spec:
 selector:
  matchLabels:
   app: nginx # has to match .spec.template.metadata.labels
 serviceName: "nginx"
 replicas: 3 # by default is 1
 template:
  metadata:
   labels:
    app: nginx # has to match .spec.selector.matchLabels
  spec:
   terminationGracePeriodSeconds: 10
   containers:
   - name: nginx
    image: k8s.gcr.io/nginx-slim:0.8
    ports:
    -   containerPort: 80
      name: web
    volumeMounts:
    - name: www
     mountPath: /usr/share/nginx/html
volumeClaimTemplates:
- metadata:
    name: www
spec:
 accessModes: [ "ReadWriteOnce" ]
 storageClassName: "my-storage-class"
 resources:
  requests:
  storage: 1Gi
```

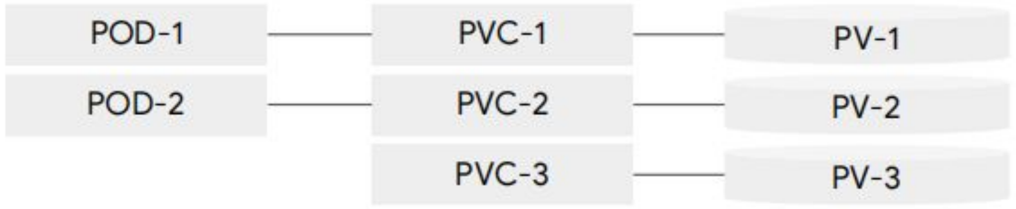
Pod to PV mapping

- Each pod will get its own PVC and PV
- The PVC will be created “Volume claim template” in StatefulSet specification



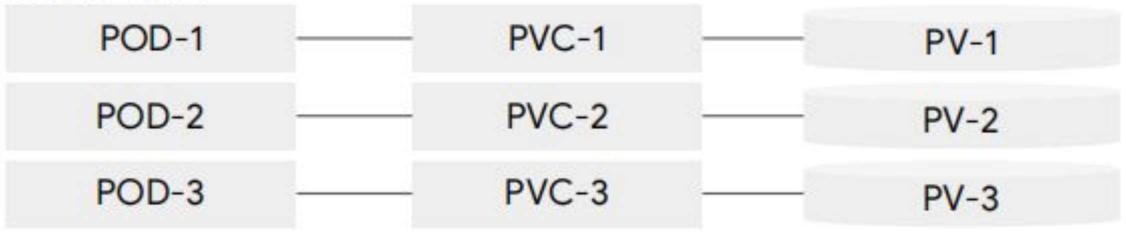
Scale in & out

- Scale in



POD-3 is removed by scale in , But PVC-3 and PV-3 are not deleted

- Scale out



POD-3 is added by scale out. It is attached previous PVC-3 and PV-3

It enables POD connect to same PVC and PV and it POD to retain same stateful information in disk

Labs

Lab 4: StatefulSets

Section 3:

Summary

Mục tiêu

Kết thúc bài học, học viên cần đạt được các kĩ năng sau:

- Có thể hiểu và tạo được Volume trong Kubernetes.
- Nắm được các đối tượng để khởi tạo Volume như StorageClass, PersistentVolume và PersistentVolumeClaim.
- Có khả năng triển khai ứng dụng dưới dạng Statefulsets.

Tài liệu tham khảo

Học là có việc

Thank you