Containers & Kubernetes

Session #08

Storage

ConfigMaps

Secrets

Lab

Storage

Motivation

Storage

- On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers
- One problem is the loss of files when a container crashes. The kubelet restarts the container but with a clean state
- A second problem occurs when sharing files between containers running together in a Pod.
- The Kubernetes volume abstraction solves both of these problems.

Kubernetes Volumes

Storage

- Kubernetes supports many types of volumes and a Pod can use any number of volume types simultaneously
- <u>Ephemeral volume</u> types have a lifetime of a pod
- <u>Persistent volumes</u> exist beyond the lifetime of a pod
- When a pod ceases to exist, Kubernetes destroys ephemeral volumes; however, Kubernetes does not destroy persistent volumes

Ephemeral Volumes

Storage

- Some application need additional storage but don't care whether that data is stored persistently across restarts, like caching services
- Other applications expect some read-only input data to be present in files, like configuration data or secret keys
- Ephemeral volume are designed for these use cases. Because volumes follow the Pod's lifetime and get created and deleted along with the Pod
- Volumes are defined on Pod Spec along containers
- Mount concept follows Docker volume mount strategy

Types of Ephemeral Volumes

Storage

- <u>Empty Dir</u> – A temporary folder for all containers within a Pod to read/write to.
- <u>Host Path</u> – Mounts a file or directory from the host node's filesystem into your Pod. Not practical in a multi-node cluster.
- <u>Config Map/Secret</u> – A Read-only folder that provides a way to inject configuration data into pods.

Ephemeral Volumes

Storage

redis container only mounts **emptyDir** volume

nginx container mounts both volumes

emptyDir volume definition

hostPath volume definition

```yaml
spec:
  containers:
    - name: redis
      image: redis
      ports:
        - containerPort: 80
      volumeMounts:
      - name: tempVol
        mountPath: /share
    - name: nginx
      image: nginx:1.17
      ports:
        - containerPort: 80
      volumeMounts:
      - name: tempVol
        mountPath: /usr/share/nginx/html
      - mountPath: /share/somefile
        name: hostFolder
        readOnly: true
  volumes:
  - name: tempVol
    emptyDir: {}
  - name: hostFolder
    hostPath:
      path: /var/local/aaa/1.txt
      type: FileOrCreate
```

Demo: Ephemeral Volumes

Persistent Volumes

Storage

- Persistent volumes exist beyond the lifetime of a pod
- PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed using PersistentVolume and PersistentVolumeClaim
- PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes
- PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources

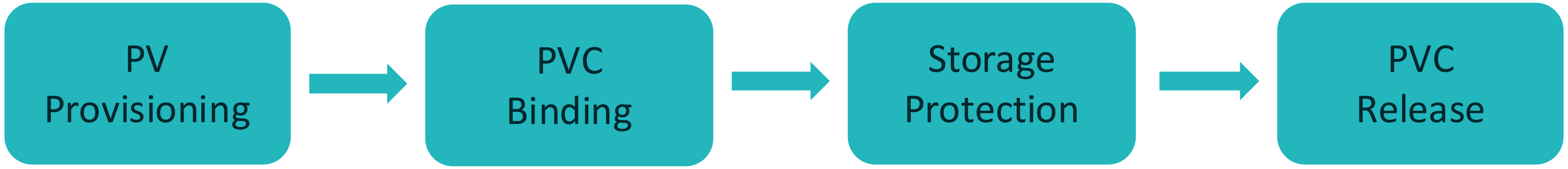
Persistent Volumes Types

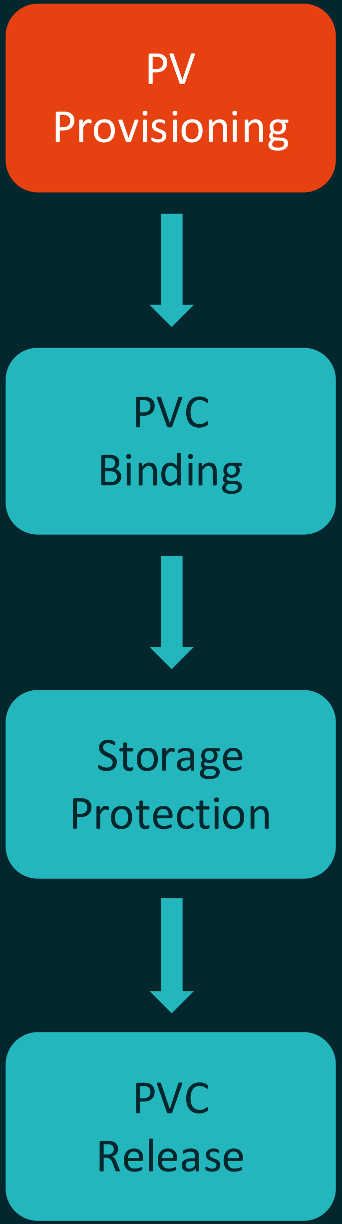
Storage

- awsElasticBlockStore - AWS Elastic Block Store (EBS)
- azureDisk - Azure Disk
- azureFile - Azure File
- cephfs - CephFS volume
- csi - Container Storage Interface (CSI)
- fc - Fibre Channel (FC) storage
- gcePersistentDisk - GCE Persistent Disk
- glusterfs - Glusterfs volume
- hostPath - HostPath volume (for single node testing only; WILL NOT WORK in a multi-node cluster; consider using local volume instead)
- iscsi - iSCSI (SCSI over IP) storage
- local - local storage devices mounted on nodes.
- nfs - Network File System (NFS) storage
- portworxVolume - Portworx volume
- rbd - Rados Block Device (RBD) volume
- vsphereVolume - vSphere VMDK volume

Persistent Volumes Lifecycle

Storage

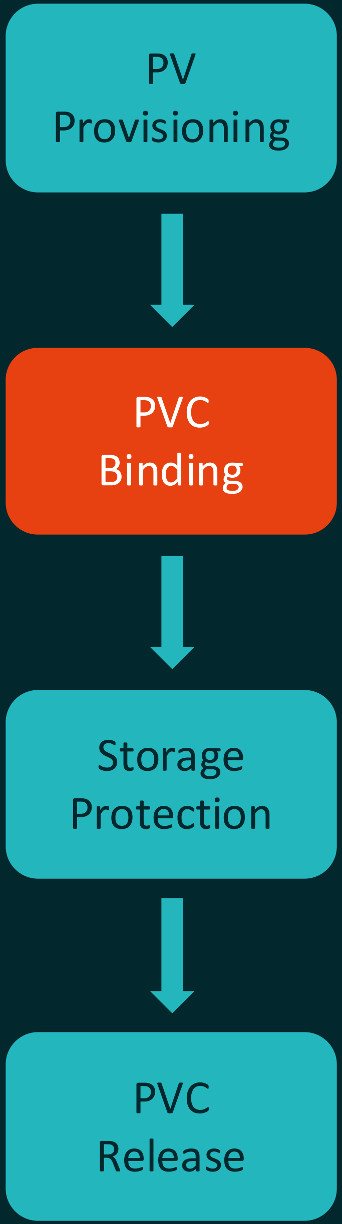




PV Provisioning

PV Lifecycle

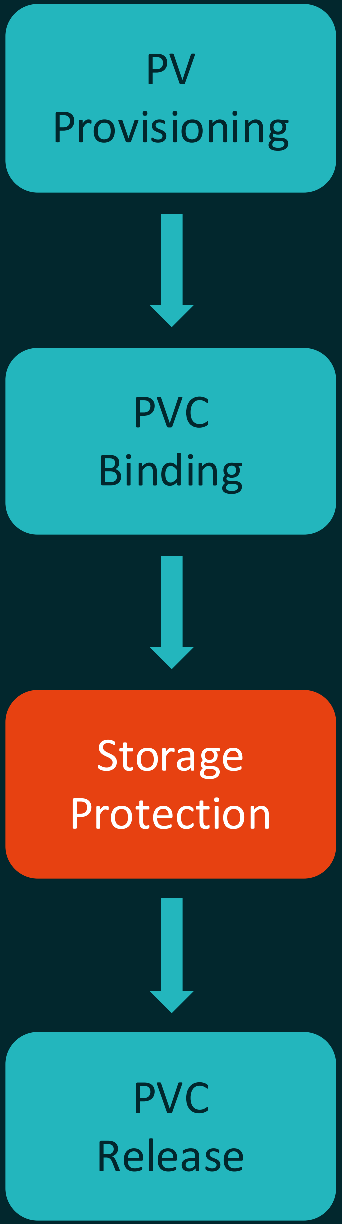
- There are two ways PVs may be provisioned: statically or dynamically.
- Static: A cluster administrator creates a number of PVs. They carry the details of the real storage, which is available for use by cluster users
- Dynamic: When none of the static PVs the administrator created match a user's PersistentVolumeClaim, the cluster may try to dynamically provision a volume specially for the PVC. This provisioning is based on StorageClasses



PVC Binding

PV Lifecycle

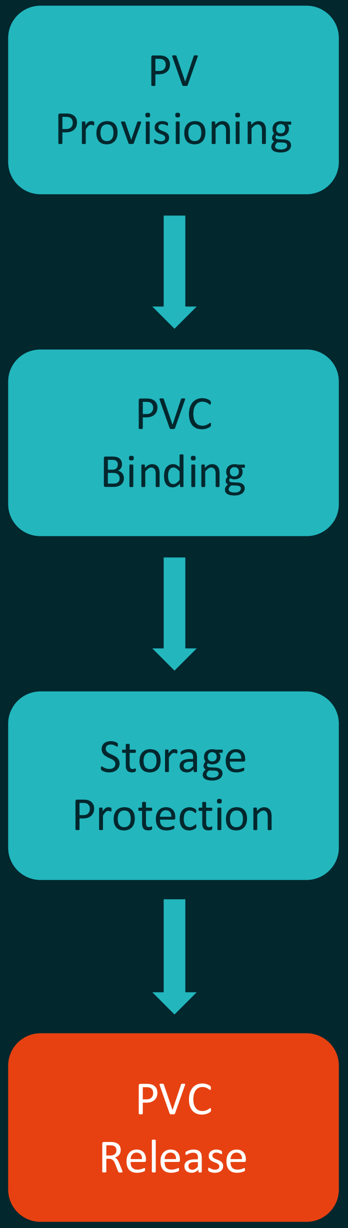
- A user creates a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes
- Claims will remain unbound indefinitely if a matching volume does not exist
  - For example, a cluster provisioned with many 50Gi PVs would not match a PVC requesting 100Gi
- Access Modes
  - ReadWriteOnce (RWO): RW on single node for all pods on node
  - ReadOnlyMany (ROX): RO on multiple nodes
  - ReadWriteMany (RWX): RW on multiple nodes and pods
  - ReadWriteOncePod (RWOP): RW to a single pod



Storage Protection

PV Lifecycle

- The purpose of the Storage Object in Use Protection feature is to ensure that PersistentVolumeClaims (PVCs) in active use by a Pod and PersistentVolume (PVs) that are bound to PVCs are not removed from the system
- If a user deletes a PVC in active use by a Pod, the PVC is not removed immediately. PVC removal is postponed until the PVC is no longer actively used by any Pods
- If an admin deletes a PV that is bound to a PVC, the PV is not removed immediately. PV removal is postponed until the PV is no longer bound to a PVC



PVC Release

PV Lifecycle

- When a user is done with their volume, they can delete the PVC objects from the API that allows reclamation of the resource
- The reclaim policy for a PersistentVolume tells the cluster what to do with the volume after it has been released of its claim: Retain, Recycle or Delete
- Retain: Volume is available but data written remains on storage
- Recycle: Data written is delete and volume as all free space again
- Delete: Volume is deleted. Depends on Storage Class

Persistent Volume

Storage

capacity defines storage size

accessModes defines allowed modes

storageClassName defines storage class

Specific properties for local storage

nodeAffinity defines where volume will be located to work properly on multi-node cluster

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: sample-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  local:
    path: /tmp
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - docker-desktop
```

Persistent Volume Claim

Storage

storageClassName defines type of volume that Claim wants to use

accessModes needed on volume

resources defines amount of storage

To have a binding, all configs needs to be filled by an individual volume

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: sample-pv-claim
spec:
  storageClassName: local-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Demo: Persistent Volumes

ConfigMaps

Motivation

ConfigMaps

- When running pods you may need to add some configuration to make it work properly
- For example, set database host that can be different depending the environment you are running it
- At same time, there are configurations that may be shared by different pods, like an external service
- Finally, having a way to update some configuration that don’t need Pods restart
- Kubernetes have a ConfigMap object to handle configurations

What is a ConfigMap

ConfigMaps

- ConfigMap is an object used to store non-confidential data in key-value pairs
- ConfigMap allows you to decouple environment-specific configuration from your container images, so that your applications are easily portable
- Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume
- ConfigMaps as volumes are updated automatically when ConfigMap is updated
- ConfigMaps as environment variables are not updated automatically. A Pod restart is needed

ConfigMap Manifest

ConfigMaps

ConfigMap name. Used for matching

Property-like keys. One key, one value

File-like keys. One key, a list of values

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: myapp-config
data:
  database: mongodb
  database_uri: mongodb://localhost:27017

  user-interface.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
```

Pod Spec

ConfigMaps

ConfigMap as environment variable

Pod Volume mount using ConfigMap

ConfigMap as Volume

```yaml
apiVersion: apps/v1
kind: Deployment
 ...
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        env:
        - name: DATABASE_URL
          valueFrom:
            configMapKeyRef:
              name: myapp-config
              key: database_uri
        volumeMounts:
        - name: config
          mountPath: "/config"
          readOnly: true
      volumes:
        - name: config
          configMap:
            name: myapp-config
            items:
            - key: "user-interface.properties"
              path: "user-interface.properties"
```

Demo: ConfigMaps

Secrets

Motivation

Secrets

- Your pods need to have access to sensitive data, like password, token, etc.
- You may define that sensitive data on Pod specification or container image but that can arise some security concerns
- Additionally having sensitive data configured apart from the pods can bring more agility
- Secrets are Kubernetes objects to reach these goals

What is a Secret

Secrets

- Secrets allow to store and manage sensitive information, such as passwords, OAuth tokens, ssh keys and TLS certificates
- Storing confidential information in a Secret is safer and more flexible than putting it in a Pod definition or in a container image
- Pods can consume Secrets as environment variables, command-line arguments, or as configuration files in a volume
- Secrets as volumes are updated automatically when Secrets is updated
- Secrets as environment variables are not updated automatically. A Pod restart is needed

Types of Secret

Secrets

Default ->

Builtin Type	Usage
`Opaque`	arbitrary user-defined data
`kubernetes.io/service-account-token`	service account token
`kubernetes.io/dockercfg`	serialized `~/.dockercfg` file
`kubernetes.io/dockerconfigjson`	serialized `~/.docker/config.json` file
`kubernetes.io/basic-auth`	credentials for basic authentication
`kubernetes.io/ssh-auth`	credentials for SSH authentication
`kubernetes.io/tls`	data for a TLS client or server
`bootstrap.kubernetes.io/token`	bootstrap token data

https://kubernetes.io/docs/concepts/configuration/secret/#secret-types

Encoded, NOT Encrypted

Secrets

- Secrets are stored unencrypted inside the cluster!
- Secrets are only encoded using base64 algorithm which can be easily reverted
- AS being a regular object on Kubernetes, can be retrieved as plain text by someone with API access
- Some options to handle this security concern
  - Using Kubernetes RBAC to restrict reading and writing of Secrets
  - Using 3rd-party services to store secrets and integrate with Kubernetes (like Azure Key Vault, Hashicorp Vault, etc.)

Secret Manifest

Secrets

Pair Key-Value with secret

Secret value encoded on base64

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: bd-secret
data:
  db_pass: YW5vdGhlcl9zdHJvbmdfcGFzc3dvcmQ=
  db_user: dGVzdHVzZXI=
```

Pod Spec

Secret

Secret as environment variable

Pod Volume mount using Secret

Secret as Volume

```yaml
apiVersion: apps/v1
kind: Deployment
...
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        env:
        - name: DB_USERNAME
          valueFrom:
            secretKeyRef:
              name: bd-secret
              key: db_user
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: bd-secret
              key: db_pass
        volumeMounts:
        - name: secret-vol
          mountPath: "/config"
          readOnly: true
      volumes:
      - name: secret-vol
        secret:
          secretName: file-secret
```

Using Kubectl Commands

Secrets

- Create from literal (command line value)

**kubectl create secret generic mysecret **

--from-literal=secret=secretValue

- Create from file (encoding content)

**kubectl create secret generic test-secret **

--from-file=secret-file.json

- Using kubectl to encode content to base64

**kubectl create secret generic test-secret **

--dry-run=client --from-file=secret-file.json –o yaml

Demo: Secrets