Kubernetes from Basic to Advanced

kubernetes

Session #05
Storage & Config

kubernetes

Session Contents

- Kubernetes Volumes
- ConfigMaps
- Secrets

kubernetes

Kubernetes Volumes

Motivation

- 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 problems

Kubernetes Volumes

- Kubernetes supports many types of volumes and a Pod can use any number of volume types simultaneously
- Ephemeral volume have a lifetime of a pod
- Persistent volumes exist beyond the lifetime of a pod

kubernetes

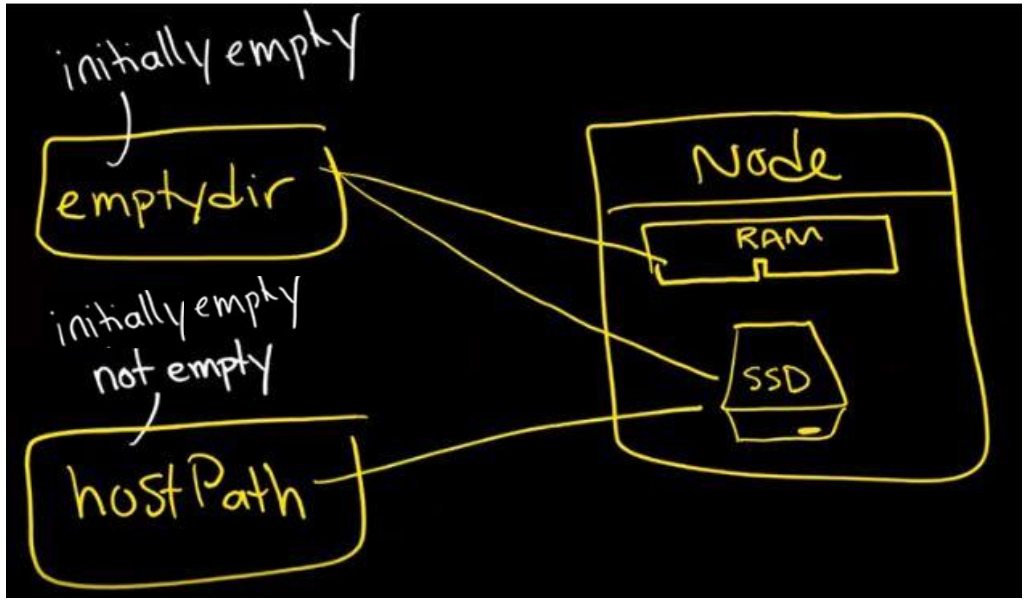
Ephemeral Volumes

- 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 (not the containers)
- Volumes are defined on Pod Spec along containers
- Inside containers you see like a mount volume, is simple filesystem

kubernetes

Ephemeral Volumes Types

- Empty Dir – A temporary folder for all containers within a Pod to read/write to.
- Host Path – Mounts a file or directory from the host node's filesystem into your Pod. Not practical in a multi-node cluster.



kubernetes

Ephemeral Volumes Types

- When a bind mount is defined, host path needs to exists. If don't, container is not executed
- On Kubernetes, **`hostPath`** have a **`type`** properties that allows to control that behavior

Value	Behavior
	Empty string (default) is for backward compatibility, which means that no checks will be performed before mounting the hostPath volume.
`DirectoryOrCreate`	If nothing exists at the given path, an empty directory will be created there as needed with permission set to 0755, having the same group and ownership with Kubelet.
`Directory`	A directory must exist at the given path
`FileOrCreate`	If nothing exists at the given path, an empty file will be created there as needed with permission set to 0644, having the same group and ownership with Kubelet.
`File`	A file must exist at the given path

kubernetes

Ephemeral Volumes Manifest

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

- 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
- To have a match between PVC and PV, a PV needs to match all PVC requests

kubernetes

Persistent Volume Type

- PersistentVolume types are implemented as plugins. Kubernetes currently supports the following plugins:
  - cephfs - CephFS volume
  - csi - Container Storage Interface (CSI)
  - fc - Fibre Channel (FC) storage
  - 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
  - rbd - Rados Block Device (RBD) volume

kubernetes

Container Storage Interface (CSI)

- Defines a standard interface for Kubernetes to expose arbitrary storage systems to their container workloads
- Once a CSI volume driver is deployed on a Kubernetes cluster, users may use the csi volume type to attach or mount the volumes exposed by the CSI driver
  - Through a reference to a **`PersistentVolumeClaim`**
  - With a generic ephemeral volume
- Popular CSI drivers
  - Azure File CSI Driver for Kubernetes
  - Azure Disk CSI driver for Kubernetes
  - Amazon Elastic Block Store (EBS) CSI driver
  - Google Compute Engine Persistent Disk CSI Driver

kubernetes

Storage Classes

- Provides a way for administrators to describe the "classes" of storage they offer
- Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators
- Storage classes are used to allow dynamic allocation of PVs
- Uses 3 concepts: provisioner, parameters and reclaimPolicy
- Provisioner: What volume plugin to be used (PV Type)
- Parameters: How to config/setup the provisioner
- ReclaimPolicy: What to do when volume is no longer needed

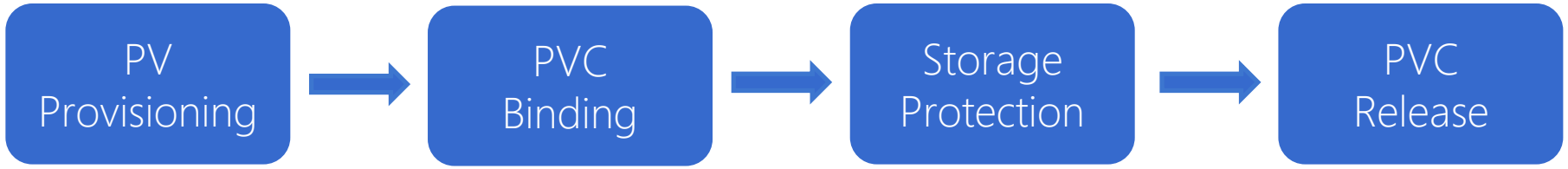
kubernetes

Storage Classes: Reclaim Policy

- Reclaim Policy can be Delete or Retain
- Delete means that the storage volume is deleted when it is no longer required by the pod
- Retain means that the storage volume is retained when no longer required by the pod and can be reused by other pods
- If no reclaimPolicy is specified when a StorageClass object is created, it will default to Delete.

kubernetes

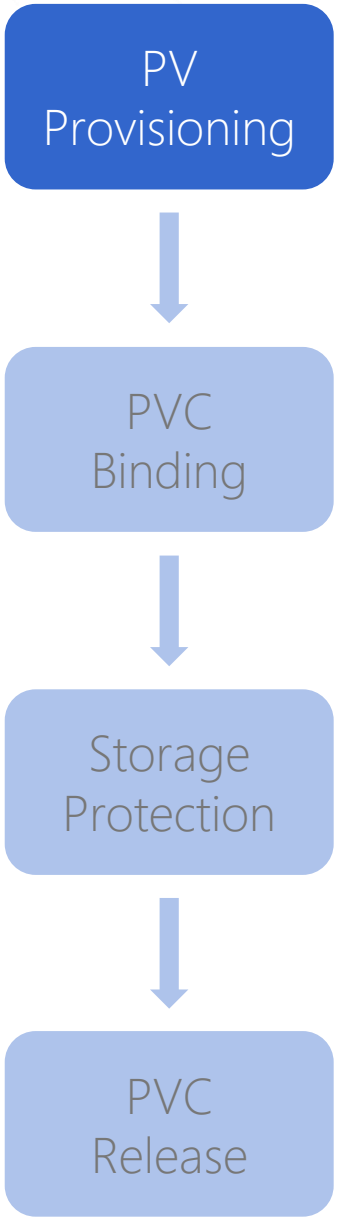
Persistent Volumes Lifecycle



kubernetes

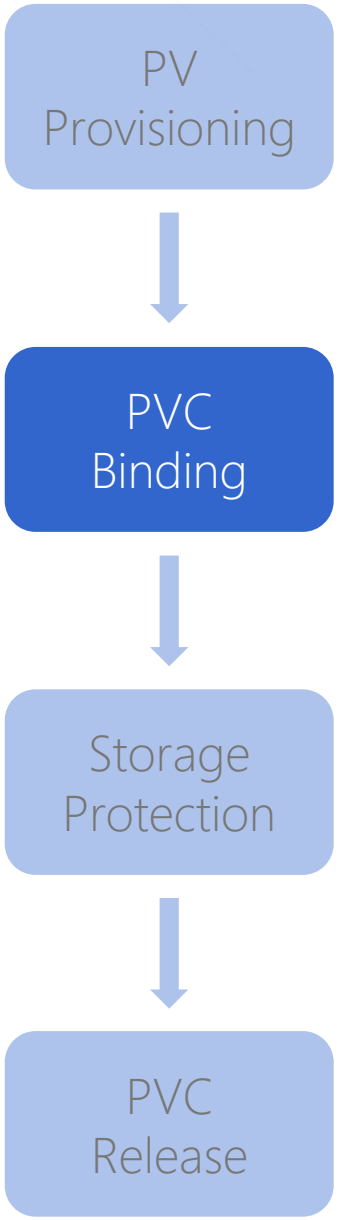
PV Provisioning

- There are two ways PVs may be provisioned: static or dynamic.
- 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



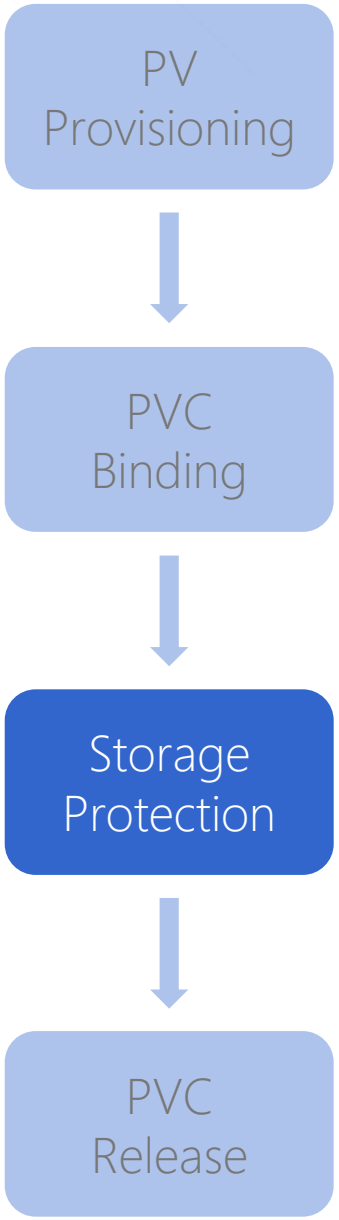
PVC Binding

- 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
- 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



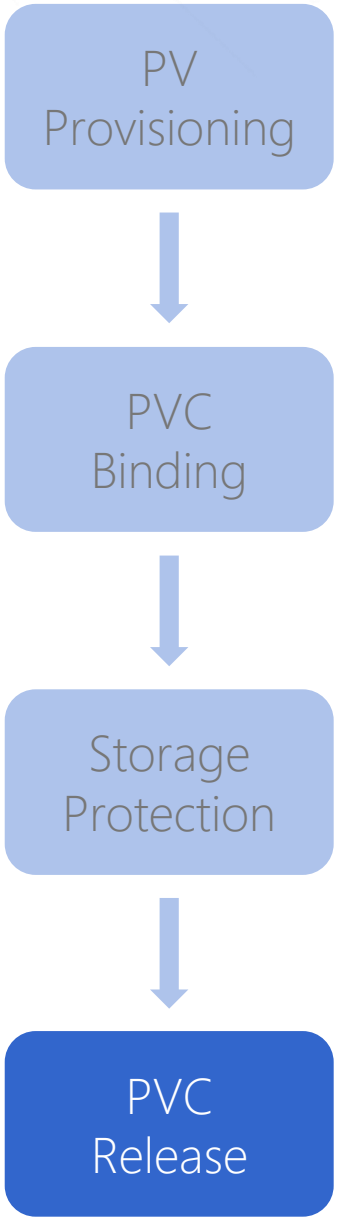
Storage Protection

- The purpose 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 an active 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

- 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 or Delete
- Retain: Volume is available but data written remains on storage
- Delete: Volume is deleted. Depends on Storage Class



Persistent Volume

capacity defines storage size

storageClassName defines storage class

accessModes defines allowed modes

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
  labels:
    type: sample-storage
spec:
  capacity:
    storage: 5Mi
  volumeMode: Filesystem
  storageClassName: microk8s-hostpath-retain
  accessModes:
  - ReadWriteOnce
  local:
    path: /tmp/microk8s-data
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - tiberna-rasp-003
```

Persistent Volume Claim

accessModes needed on volume

storageClassName defines type of volume that Claim wants to use

resources defines amount of storage

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: sample-pv-claim-static
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: microk8s-hostpath-retain
  resources:
    requests:
      storage: 1Mi
```

Demo | Static vs Dynamic

ConfigMaps

ConfigMaps: Motivation

- 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

kubernetes

What is a ConfigMap

- 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 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

kubernetes

ConfigMap

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 Manifest

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 | ConfigMap

Secrets

Secrets: Motivation

- 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

kubernetes

What is a (k8s) Secret

- 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 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
- Secrets can have several types. Opaque is the default one.

kubernetes

Secret Types

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

kubernetes

Encoded, but NOT encrypted!

- 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.)

kubernetes

Manifest vs Imperative

- Due to store confidential data, secrets are usually created using imperative command

- 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
```

kubernetes

Demo | Secrets

Questions?

kubernetes

Lab #04: Storage, ConfigMaps and Secrets

kubernetes