Container Orchestration Using Kubernetes

TECHNOLOGY

Storage

A Day in the Life of a DevOps Engineer

You are working as a DevOps engineer in an organization and have the responsibility to deploy WordPress and MySQL using both persistent volume and persistent volume claims. Your organization requires the WordPress and MySQL applications.

WordPress should be deployed using a host path, and MySQL should be set up via NFS. Your goals are to set up dynamic volume provisioning, integrate secrets into volumes, implement CSI volume cloning, and monitor storage capacity. You also need to restrict the volumes to certain nodes.

To achieve all the above, along with some additional features, we will be learning a few concepts in this lesson that will help you find a solution for the above scenario.

Learning Objectives

By the end of this lesson, you will be able to:

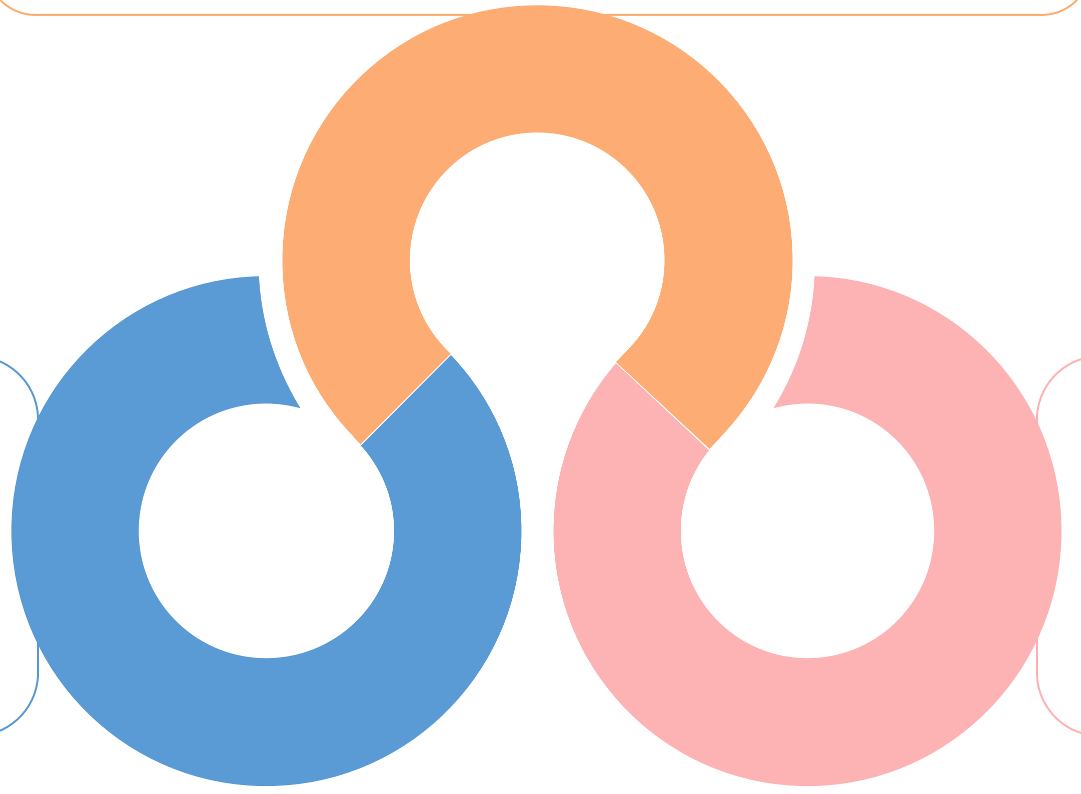
- Analyze the role of storage in Kubernetes for optimizing application performance and data persistence
- Identify and distinguish the various types of volumes in Kubernetes, highlighting their specific use cases
- Interpret the function and significance of storage classes within Kubernetes storage management
- Demonstrate an understanding of dynamic volume provisioning and its role in efficient storage allocation
- Classify the different types of ephemeral volumes and elucidate their benefits and applications in temporary data storage

TECHNOLOGY

Overview of Storage in Kubernetes

Persistent Storage

Kubernetes has become crucial for hosting microservice-based processes and storing data due to its persistent storage capabilities.



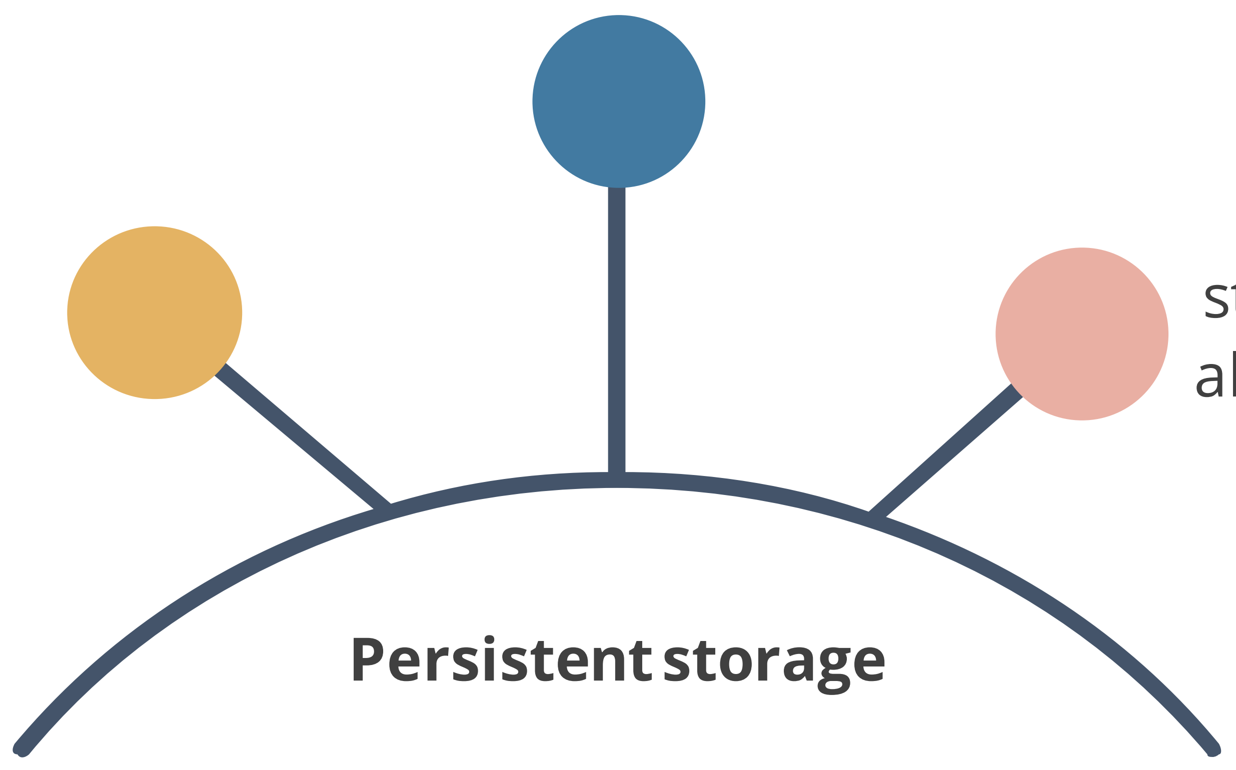
Users can create and access databases, as well as store data for various applications.

Administrators can map service quality levels to different storage classes using StorageClass.

Users can implement arbitrary and backup policies that cluster administrators have assigned.

Kubernetes and Persistent Storage

Applications must remain functional and retain their original state.


Without Kubernetes, deleting localized data disrupts information sharing among applications within the same category.
Applications must remain functional and retain their original state.
Kubernetes enables data storage outside of containers, allowing uninterrupted access.
Persistent storage

Requirements for Persistent Storage

These elements are essential for persistent storage:

Container

Location for persistent information storage

Place to persistently store information

Backend

Kubernetes persistent storage conceals the data from both applications and users, utilizing protocols such as NFS, iSCSI, and SMB.

01 Storage services and systems on cloud platforms broaden access to user information.

02 Third-party cloud storage creates environments that grant users complete access to their data, facilitating integration.

03 Storage providers, including Amazon S3 and LINBIT, offer an array of tools and applications that support persistent storage.

TECHNOLOGY

Volumes

Overview of Volumes

The volume abstraction in Kubernetes helps business-critical applications running in containers to address challenges.

It prevents data and file loss when a container crashes.

It enables file sharing between containers that run together in a pod.

Background of Volumes

A pod in Kubernetes can utilize numerous volumes and volume types concurrently.

A volume’s lifespan exceeds that of any container running within the pod, ensuring data preservation across container restarts or terminations.

To use a volume, specify it in **.spec.volumes** for the pod and declare the necessary volume mounts in **.spec.containers[*].volumeMounts** for the containers.

Types of Volumes

Kubernetes supports a variety of volume types:

awsElasticBlockStore	azureDisk	azureFile	cephfs
cinder	configMap	downwardAPI	emptyDir
fibre channel	gcePersistentDisk	glusterfs	hostPath

Types of Volumes

Kubernetes supports a variety of volume types:

iscsi	local	nfs	PV
PVC	portworxVolume	projected	quobyte
rbd	scaleIO (deprecated)	storageOS	vsphereVolume

TECHNOLOGY

Ephemeral Volumes

simplilearn

Overview of Ephemeral Volumes

Caching services, which often operate with limited memory, can transfer infrequently used data to storage.

Ephemeral volumes, designed for specific use cases, allow for specification directly within the pod spec.

This simplicity streamlines application deployment and management.

Ephemeral Volumes

Ephemeral volumes cater to applications that require additional temporary storage without needing data persistence across restarts.

Types of ephemeral volumes:

1. emptyDir
2. hostPath
3. configMap
4. secret
5. Generic Ephemeral Volumes
6. CSI Ephemeral Volumes

EmptyDir

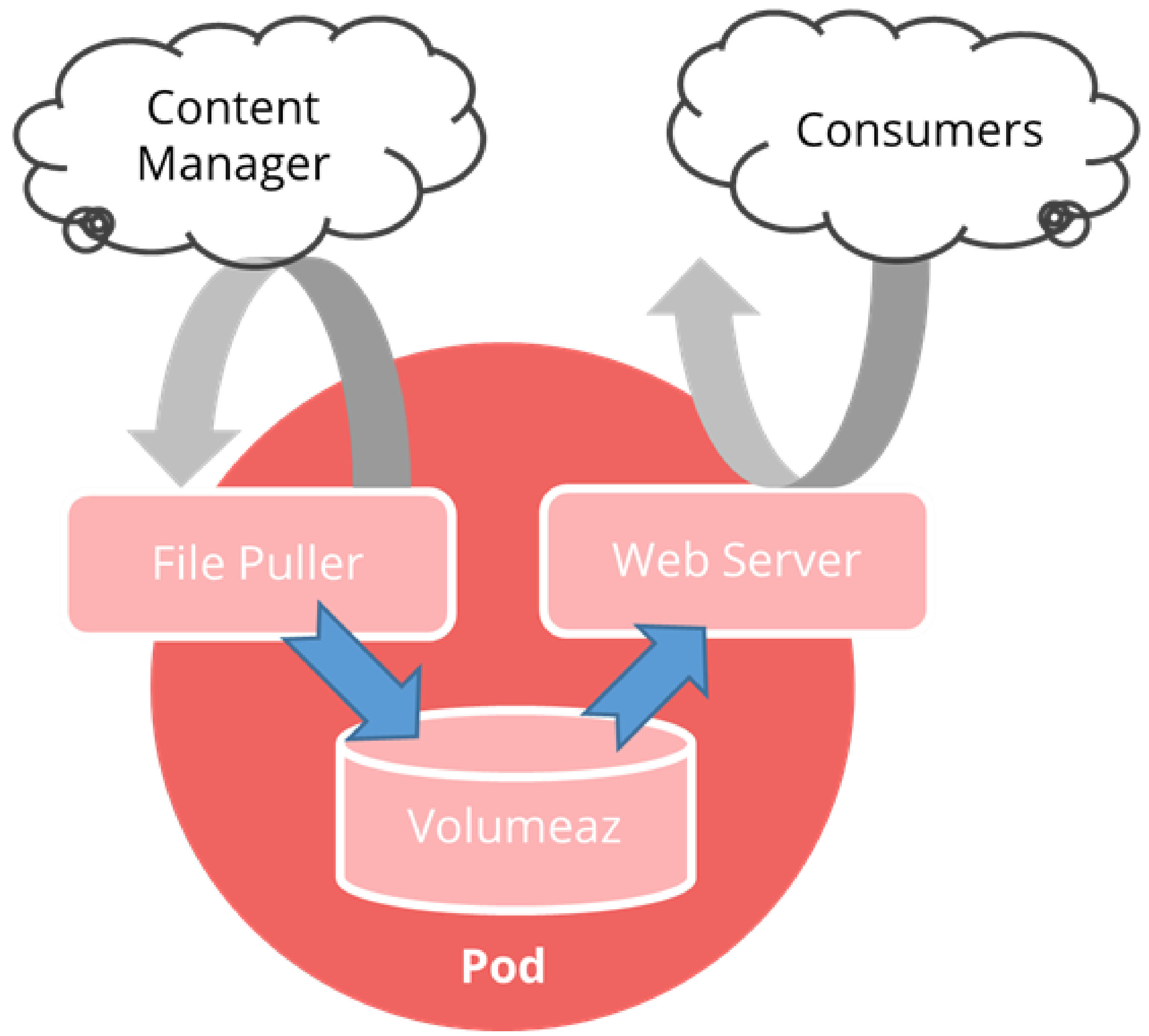
Kubernetes creates an emptyDir volume whenever it assigns a pod to a node. The emptyDir volume exists as long as the pod runs on that Node.

Uses of emptyDir:

- ✓ It provides a scratch space for operations like a disk-based merge sort.
- ✓ It stores checkpoints for long computations to aid recovery from crashes.
- ✓ It holds files, serving as a bridge to share content between containers within a pod.

EmptyDir

Pods are designed to support multiple cooperating processes, encapsulated as containers, that together provide a cohesive unit of service.



One container might function as a web server for files in a shared volume, such as emptyDir, while a separate sidecar container could update those files from a remote source.

EmptyDir

Writing a pod definition using emptyDir configuration:

```
apiVersion:  v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: registry.k8s.io/test-webserver
    name: test-container
    volumeMounts:
      - mountPath: /cache
        name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir:
      sizeLimit: 500Mi
```

HostPath

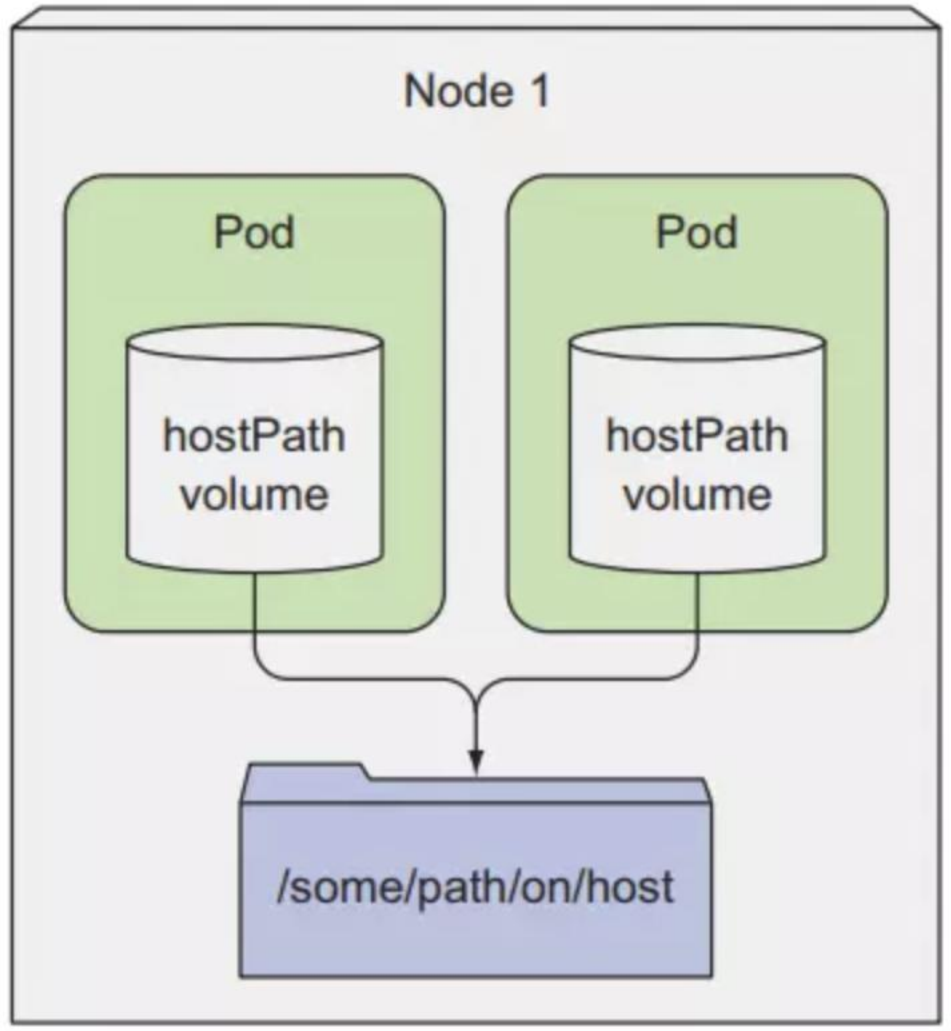
A hostPath volume mounts a specific file or directory from the host node's filesystem into the pod.

Uses of hostPath:

- ✓ It provides a container access to the internals of the container runtime.
- ✓ For running cAdvisor in a container, you can use a hostPath of /sys.
- ✓ It allows a pod to specify the prerequisites of a given hostPath, determining whether it should exist before the pod runs, and dictating its required attributes if it needs creation.

HostPath

Multiple pods on the same host can share the same hostPath.



HostPath

Writing a pod definition using hostPath configuration:

```yaml
apiVersion:  v1
kind: Pod
metadata:
  name: test-hostpath
spec:
  containers:
  - image: registry.k8s.io/test-webserver
    name: test-container
    volumeMounts:
      - mountPath: /log
       name: log-volume
  volumes:
  - name: log-volume
    hostPath:
      path: /var/log
```

ConfigMap

ConfigMaps inject configuration data into pods.

It is possible to configure the path that is to be used for a specific entry in the configMap.

Data that is stored in a configMap can be referenced in a Volume of type **configMap.** Then, it can then be consumed by a pod's containerized applications.

ConfigMap

Writing a pod definition using configMap configuration:

```
    apiVersion:  v1
    kind: Pod
    metadata:
       name: configmap-pod
    spec:
       containers:
       - name: test
         image: busybox:1.28
         command: ['sh', '-c', 'echo "The app is running!"
&& tail -f /dev/null’]
         volumeMounts:
         - name: config-vol
           mountPath: /etc/config
       volumes:
       - name: config-vol
         configMap:
           name: log-config
           items:
           - key: log_level
             path: log_level
```

Secret

A secret volume securely conveys sensitive information, such as passwords, to pods.

One can store secrets in the Kubernetes API, and pods can then mount them as files for use.

Kubernetes mounts secrets as read-only by default.

Secret

Writing a pod definition using secret configuration:

```
apiVersion:  v1
kind: Pod
metadata:
  name: test-hostpath
spec:
  containers:
  - image: registry.k8s.io/test-webserver
    name: test-container
    volumeMounts:
      - mountPath: /secrets
        name: secret-volume
  volumes:
  - name: secret-volume
    secret:
      name: my-secret
```

Generic Ephemeral Volumes

Generic ephemeral volumes function similarly to emptyDir volumes. Below is a manifest of a generic ephemeral volume:

Example:

```
kind:    Pod
apiVersion:   v1
metadata:
name: my-app
spec:
   containers:
  - name: my-frontend
       image:  busybox
       volumeMounts:
         - mountPath: "/scratch"
           name: scratch-volume
       command: [  "sleep","1000000"  ]
   volumes:
    - name: scratch-volume
      ephemeral:
```

Example:

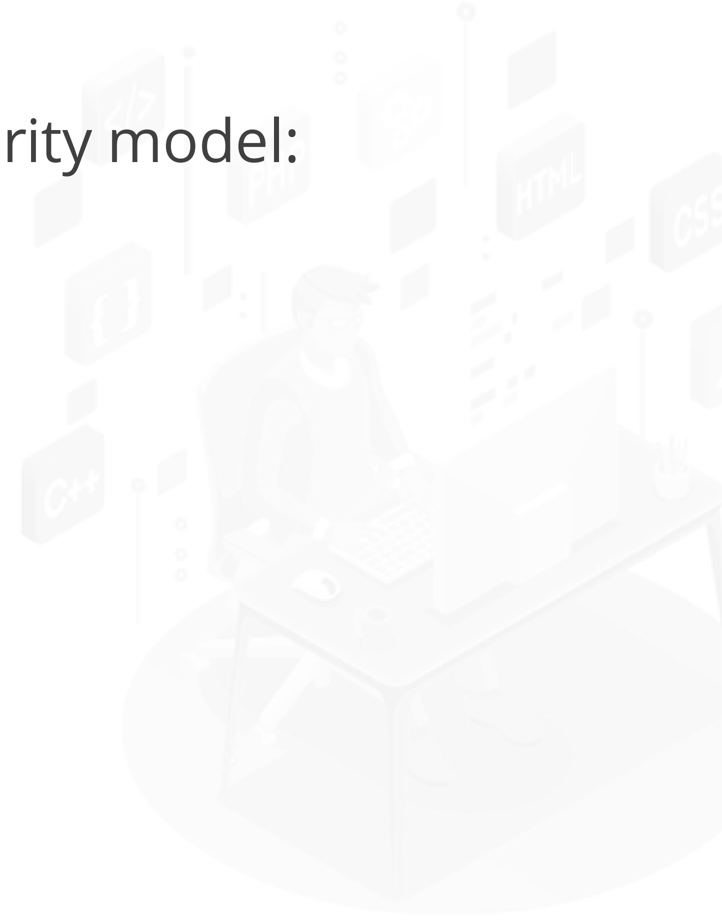
```
volumeClaimTemplate:
    metadata:
      labels:
        type: my-frontend-volume
    spec: inline.storage.kubernetes.io
    accessModes: [ "ReadWriteOnce" ]
    storageClassName: "scratch-storage-
class"
    resources:
      requests:
        storage:  1Gib
```

Security Considerations for Ephemeral Volumes

Enabling the GenericEphemeralVolume feature allows individuals to create Persistent Volume Claims (PVCs) indirectly through pod creation, even if they lack direct permissions to create PVCs.

Two options exist for those who find this arrangement incompatible with their security model:

- Explicitly disable the feature through the feature gate
- Implement a pod security policy



CSI Ephemeral Volumes

CSI ephemeral volumes are locally managed on each Node. They are created together with other local resources once a node is scheduled to run a pod.

Here is an example of a manifest for a pod utilizing CSI ephemeral storage:

Example:

```
kind:   Pod
apiVersion:  v1
metadata:
name: my-csi-app
spec:
   containers: kubernetes.io/gce-pd
     - name: my-frontend
       image:  busybox
       volumeMounts:
         - mountPath: "/data"
           name: my-csi-inline-vol
       command: [  "sleep","1000000"  ]
   volumes:
    - name: my-csi-inline-vol
       csi:
        driver: inline.storage.kubernetes.io
      volumeAttributes:
        foo: bar
```

Ephemeral Volumes

Third-party CSI storage drivers and additional storage drivers that support dynamic provisioning may facilitate generic ephemeral volumes.

Third-party drivers can offer functionalities not available in Kubernetes.

Some CSI drivers, designed specifically for CSI ephemeral volumes, do not offer support for dynamic provisioning.

However, these drivers are not compatible with generic ephemeral volumes.

Sharing Data Between Containers in the Same Pod

Duration: 15 mins

Problem Statement:

You have been asked to demonstrate data-sharing between containers in the same pod through hostPath volumes for mounting pod files onto the file system of the host node.

Assisted Practice: Guidelines

Steps to be followed:

1. Configure and launch the pod with the shared volume
2. Interact with the shared volume from both containers
3. Test data persistence and sharing capability

Mounting Pod Files to Host with hostPath

Duration: 15 mins

Problem Statement:

You have been asked to create a hostPath volume to mount files from a pod onto the file system of the host node.

Assisted Practice: Guidelines

Steps to be followed:

1. Create a pod using hostPath
2. Create files within the pod
3. Access files on other nodes

Creating a Deployment with ConfigMap as Volume

Duration: 15 mins

Problem Statement:

You have been asked to create a deployment with ConfigMap as volume to enhance the flexibility, manageability, and scalability of your application.

Assisted Practice: Guidelines

Steps to be followed:

1. Create a ConfigMap
2. Create a deployment to attach a ConfigMap as volume to it

Creating and Using Secrets in a Volume

Duration: 15 mins

Problem Statement:

You have been asked to create a Kubernetes secret and mount it as a volume inside a pod for enhancing security in the Kubernetes environment.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a Kubernetes secret
2. Create a pod that uses the secret as a volume

Persistent Volumes

Persistent Volumes

Users and administrators get an API from the **PersistentVolume** subsystem in the Kubernetes orchestration system. This API abstracts storage-related details.

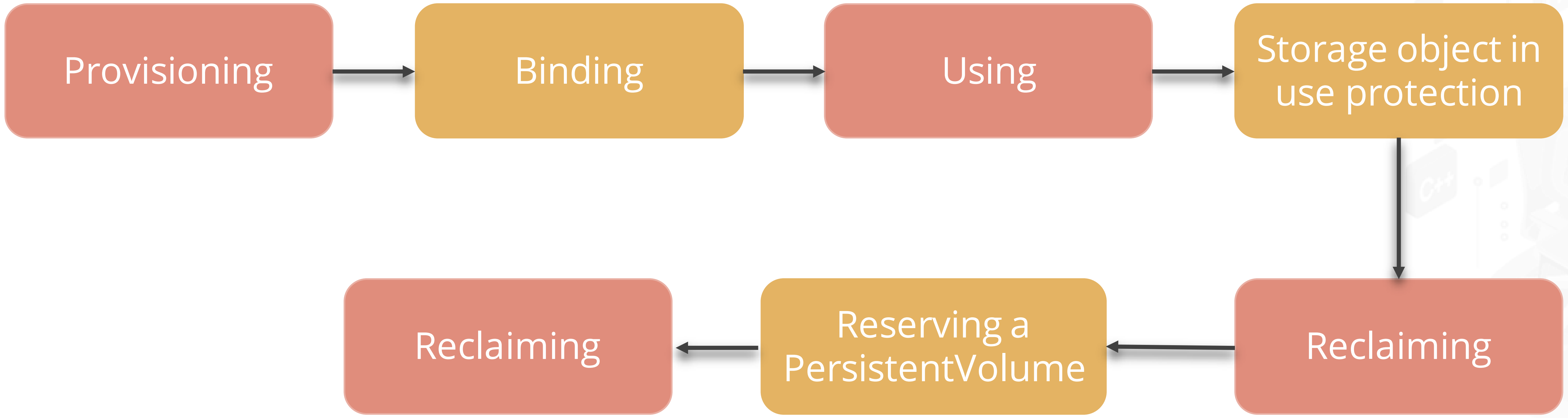
Kubernetes technology introduces two new API resources:

Persistent Volume: Similar to a node

Persistent VolumeClaim: A user's request for storage

Lifecycle of Volume and Claim

The lifecycle of interaction between PVs and PVCs:


Provisioning
Binding
Using
Storage object in use protection
Reclaiming
Reserving a PersistentVolume
Reclaiming

Types of Persistent Volumes

Some of the non-deprecated plugins that Kubernetes currently supports are:

1 awsElasticBlockStore - AWS Elastic Block Store (EBS)

2 azureDisk and azureFile - Azure Disk and Azure File from Microsoft

3 cephfs - CephFS volume

4 csi - Container Storage Interface (CSI)

5 fc - Fibre Channel (FC) storage

6 flexVolume - FlexVolume

7 flocker - Flocker storage

8 gcePersistentDisk - GCE Persistent Disk

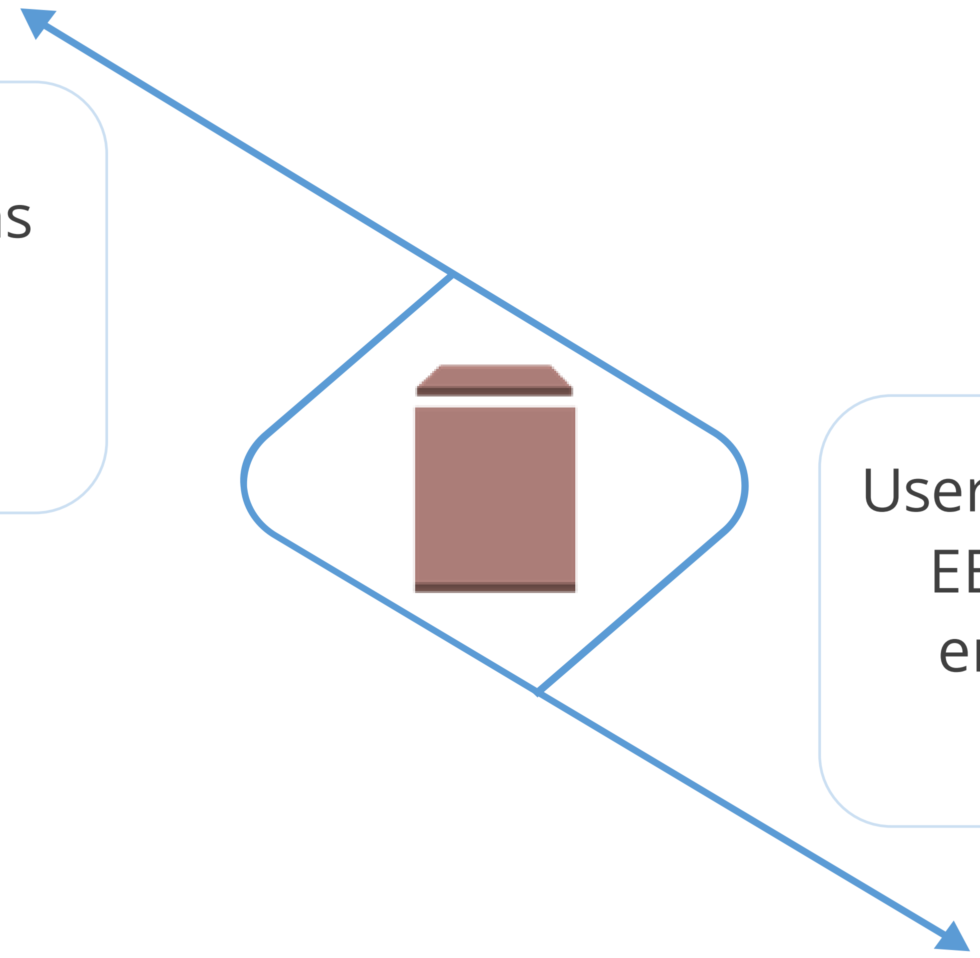
Types of Persistent Volumes

Some of the non-deprecated plugins that Kubernetes currently supports are:

9. glusterfs - Glusterfs volume
10. scsi - iSCSI (SCSI over IP) storage
11. local - local storage devices mounted on nodes
12. nfs - Network File System (NFS) storage
13. portworxVolume - Portworx volume
14. quobyte - Quobyte volume
15. rbd - Rados Block Device (RBD) volume
16. storageos - StorageOS volume

AwsElasticBlockStore

The **AwsElasticBlockStore** volume is a component of Amazon Web Services (AWS) Elastic Block Store (EBS).



The EBS volume remains unmounted, and its contents persist.

Users can pre-populate the EBS volume with data, enabling data sharing between pods.

AzureDisk

The Azure Disk from Microsoft mounts to a pod through the azureDisk volume type.

Cephfs

A CephFS volume allows users to mount an existing CephFS volume to their pod.

- CephFS volume is unmounted and its contents are preserved.
- CephFS volume can be prepopulated with data.
- Multiple writers can simultaneously mount onto a CephFS volume.

gcePersistentDisk and iscsi

A **gcePersistentDisk** volume mounts a Google Compute Engine (GCE) Persistent Disk (PD) onto the Pod.

Persistent disks can be populated in advance with data. It is possible to share this data between pods.

An **iscsi** volume allows an existing iSCSI (SCSI over IP) volume to be mounted onto the Pod.

Iscsi can be mounted as read-only simultaneously by multiple consumers.

NFS

An existing NFS (Network File System) share can be mounted onto a Pod using a NFS volume.

NFS volume can be pre-populated with data, and that data can be shared between pods. NFS can be mounted by multiple writers simultaneously.

StorageOS

An existing StorageOS volume can be mounted onto a pod using a StorageOS volume.

StorageOS runs as a container with a Kubernetes environment and results in local or attached storage being accessible from any node within the Kubernetes cluster.

StorageOS replicates data and create protection against node failure.

StorageOS provides block storage to containers accessible from a file system.

Persistent Volumes

Each persistent volume (PV) contains the volume's specification and status.

Example:

```
apiVersion: v1
kind:    PersistantVolume
metadata:
   name:  pv0003
spec:
   capacity:
      storage:   5Gi
   volumeMode:   Filesystem
   accessModes:
     - ReadWrireOnce
   persistentVolemReclainPolicy: Recycle
   storeageClassName: slow
   mountOptions:
     - hard
     - nfsvers=4.1
   nfs:
      path:  /tmp
      server:  172.17.0.2
```

PersistentVolumeClaim

Each PersistentVolumeClaim (PVC) contains a spec and status — the claim’s specification and status.

Example:

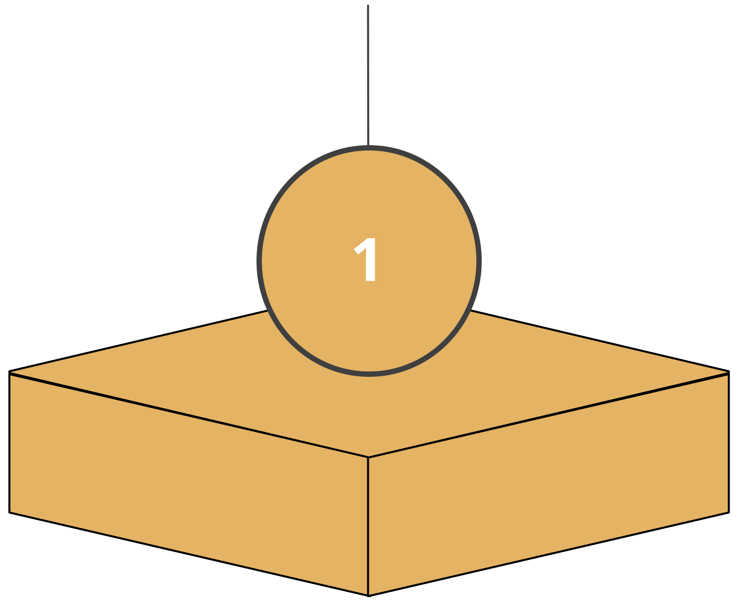
```
  apiVersion:  v1
  kind: PersistentVolumeClaim
  metadata:
    name:   myclaim
  spec:
    accwssModes:
      -  ReadWriteOnce
    volumeMode:   Folesystem
    resources:
      requests:
         stoage:  Bgi
    storageClassName : slow
    selector:
      matchLabels:
         relase:    “ stable”
      matchExpressions:
         - {key:  environment,  operator : In,
values:[dev]}
```

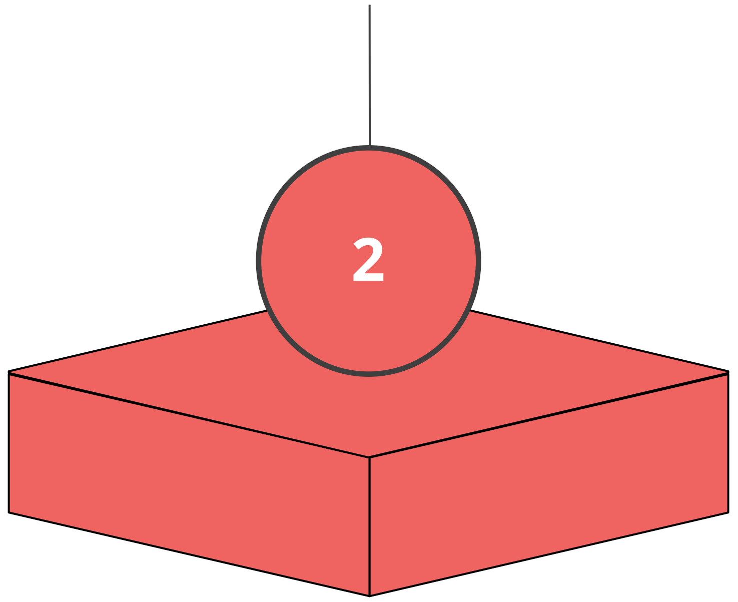
Claim as Volumes

Pods access storage by utilizing the claim as a volume.

Claims must remain in the same namespace as the pod that uses the claim.

Clusters use the claim in the pod's namespace to get the PersistentVolume that backs the claim.





Raw Block Volume Support

The following volume plugins support raw block materials:

- AWSElasticBlockStore
- AzureDisk
- CSI
- FC (Fibre Channel)
- GCEPersistentDisk
- iSCSI
- Local volume
- OpenStack Cinder
- RBD (Ceph Block Device)
- VsphereVolume

PersistentVolume Using a Raw Block Volume

Here is how to configure a PersistentVolume that uses a raw block volume for an access mode of ReadWriteOnce:

Example:

```
apiVersion:  v1
Kind: PersistentVolume
Metadata:
   name:  block-pv
spec:
   capacity;
      storage:10Gi
   accessModes:
      - ReaderWriteOnce
   volumeMode:   Block
   persistentVolumeclaimPolicy:  Retain
   fc:
      targettwWWNs:   { “50060e801049cfd1”}
      lun:  0
      rreadOnly:   false
```

PersistentVolumeClaim Requesting a Raw Block Volume

The following box shows how a PersistentVolumeClaim that requests for a raw block volume for an access mode of ReadWriteOnce can be configured:

Example:

```
apiVersion:  v1
Kind:  PersistentVolumeClaim
Metadata:
   name: block-pvc
Spec:
   accessModes:
      - ReadWriteOnce
   volumeMode: Block
   resources:
      requests:
          storage:   10Gi
```

Pod Specification Adding Raw Block Device Path in Container

The following code explains how to add a raw block device path in container:

Example:

```
apiVersion: v1
Kind:  Pod
Metadata:
   name :   pod-with-block-volume
Spec:
   Containers:
      - name:  fc-container
        image:   fedora:26
        Command:   [“/bin/sh”,  “ -c”]
        args:   [   “ tall  -f   /dev/null”    ]
        volumeDevices:
          - name:   data
            devicesaPath:   /dev/xvda
   volume :
        - name:  data
          persistentVolumeClaim:
             claimName:  block-pvc
```

Binding Block Volumes

The following table indicates if the volume will be bound or not given the combination:

PV volumeMode	PVC volumeMode	Result
Unspecified	Unspecified	BIND
Unspecified	Block	NO BIND
Unspecified	Filesystem	BIND
Block	Unspecified	NO BIND
Block	Block	BIND
Block	Filesystem	NO BIND
Filesystem	Filesystem	BIND
Filesystem	Block	NO BIND
Filesystem	Unspecified	BIND

Lifecycle and PersistentVolumeClaim

The fundamental design principle is allowing the parameters for a volume claim within a pod's volume source.

1 In terms of resource ownership, a pod with generic ephemeral storage owns the PersistentVolumeClaim(s) providing the ephemeral storage.

2 When a user deletes the pod, the kubernetes garbage collector subsequently deletes the persistentvolumeclaim.

Configuring Pod Storage Using HostPath Based PV and PVC

Duration: 15 mins

Problem Statement:

You have been asked to configure pod storage using hostPath-based PersistentVolume (PV) and PersistentVolumeClaim (PVC) in Kubernetes for efficient data storage and retrieval.

Assisted Practice: Guidelines

Steps to be followed:

1. Create PersistentVolume
2. Create PersistentVolumeClaim
3. Deploy a pod in a new namespace
4. Validate the pod and storage
5. Verify data persistence

Configuring Pod Using NFS Based PV and PVC

Duration: 15 mins

Problem Statement:

You have been asked to configure a pod using NFS based PersistentVolume (PV) and PersistentVolumeClaim (PVC) for more efficient storage management.

Assisted Practice: Guidelines

Steps to be followed:

1. Configure the NFS kernel server
2. Set the permissions
3. Configure the NFS common on client machines
4. Create the PersistentVolume
5. Create the PersistentVolumeClaim
6. Create the deployment for MySQL

TECHNOLOGY

Volume Snapshots

Introduction to Volume Snapshots

VolumeSnapshotContent and **VolumeSnapshot** API resources enable the creation of volume snapshots for users and administrators.

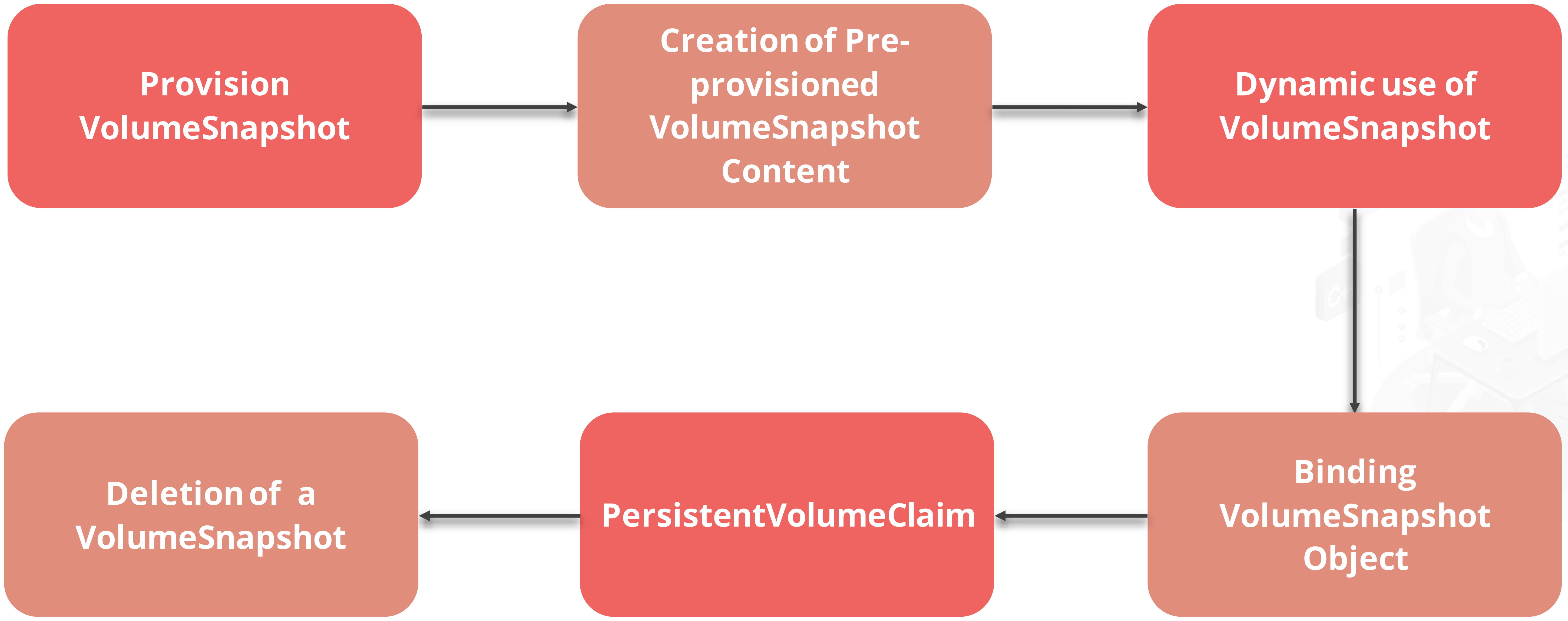
A **VolumeSnapshotContent** is a snapshot originating from a volume in the cluster that has been provisioned by an administrator.

A **VolumeSnapshot** is a user's request for a snapshot of a volume. This snapshot is similar to a **PersistentVolumeClaim**.

The **VolumeSnapshotClass** allows the specification of different attributes of a **VolumeSnapshot**.

Lifecycle of VolumeSnapshot and VolumeSnapshot Content

The lifecycle of interaction between VolumeSnapshot and VolumeSnapshot Content:


Provision VolumeSnapshot
Creation of Pre-provisioned VolumeSnapshot Content
Dynamic use of VolumeSnapshot
Binding VolumeSnapshot Object
PersistentVolumeClaim
Deletion of a VolumeSnapshot

Volume Snapshots

A spec and a status are parts of a VolumeSnapshot.

Example:

```
apiVersion: snapshot.storage.k8s.io/v1
kind:   VolumeSnapshot
metadata:
    name:   new-snapshot-test
spec:
    volumeSnapshotClassName:   csi-hostpath-snapclass
    source:
         persistentVolumeClaimName:  pvc-test
```

Volume Snapshots

The **volumeSnapshotContentName** source field is required for pre-provisioned snapshots.

Example:

```
apiVersion:   snapshor.storage.k8s.io/v1
kind:   VolumeSnapshot
metadata:
     name:   test-snapshot
spec:
     source:
         volumeSnapshotContentName: test-
contest
```

Provisioning Volumes from Snapshots

The **dataSource** field in the **PersistentVolumeClaim** can be used to provision a new volume that has been pre-populated with data from a snapshot.

VolumeSnapshot Contents

In dynamic provisioning, the snapshot common controller creates **VolumeSnapshotContent** objects.

Example:

```
apiVersion:   snapshor.storage.k8s.io/v1
kind:   VolumeSnapshot
metadata:
  name: snapcontent- 72d9a349-aacd-42d2-a248—d77560d2455
spec:
  deletionPolicy:   Delete
  driver:  hostpath.csi.k8s.io
  source:
      volumeHandle:  ee0cfb94-f8d4-11e9-b2d8-0242ac1110002
  volumeSnapshotClassName:  csi-hostpath-snapclass
  volumeSnapshotRef:
      name:  new-snapshot-test
      namespace:  default
      vid: 72d9a349-aacd-42d2-a240-d775650d2455
```

VolumeSnapshot Contents

For pre-provisioned snapshots, the cluster administrator creates the **VolumeSnapshotContent** object.

Example:

```
  apiVersion:   snapshor.storage.k8s.io/v1
  kind:   VolumeSnapshot
  metadata:
      name:  new-snapshot-content-test
  spec:
      deletionPolicy:   Delete
      driver:  hostpath.csi.k8s.io
      source:
          snapshotHandle:   7bdd0de3-aaeb-9aae-
0242ac110002
       volumeSnapshpotRef:
          name:  new-snapshot-test
          namespace:   default
```

TECHNOLOGY

Storage Classes

Storage Classes

Administrators use **StorageClass** to describe the classes of storage that they offer.

Kubernetes has no opinion about what classes represent.

StorageClass Resource

A StorageClass helps administrators describe various storage classes. Administrators can specify a default StorageClass for PVCs that are not bound to a particular class.

Example:

```
apiVersion:   storage.k8s.io/v1
kind:   StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebc
parameters:
  type:  gp3
reclaimPolicy: Retain
allowVolumeExpamsion:  true
mountOptions:
       - debug
volumeBindingMode:  Immediate
```

Provisioner

Each StorageClass has a provisioner that determines what volume plugin is used for provisioning PVs.

Authors of external provisioners have full discretion over:

- Where their code lives
- How the provisioner is shipped
- How it needs to be run
- What volume plugin it uses

StorageClass Resource: Fields

Reclaim policy

PersistentVolumes created dynamically by a StorageClass will have the reclaim policy specifics mentioned in the **reclaimPolicy** field of the class. It can be either **Delete** or **Retain**.

Volume wxpansion

Configuration of PersistentVolumes can make them expandable.

Mount options

PersistentVolumes that are dynamically created by a StorageClass will have the mount options specified in the **mountOptions** field of the class.

Volume binding mode

The **volumeBindingMode** field controls when volume binding and dynamic provisioning should occur.

Allowed Topologies

When a cluster operator specifies the **WaitForFirstConsumer** volume binding mode, provisioning to specific topologies is not constrained. The following code snippet demonstrates how to restrict the topology of provisioned volumes to specific zones:

Example:

```
apiVersion:   storage.k8s.io/v1
kind:   StorageClass
metadata:
    name: standard
provisioner: kubernetes.io/gce-pd
parameters:
    type:  pd-standard
volumeBindingMode: WaitForFirstCustomer
allowedTopologies:
   - matchLabelExpressions:
      -   key; failure-domain.beta.kubernetes.io/zone
          values
          - vs-central-a
          - vs- central-b
```

Parameters

Storage Classes have parameters that describe volumes belonging to the storage class. A maximum of 512 parameters may be defined for a StorageClass.

The provisioners include the following providers:

1. AWS EBS
2. GCE PD
3. Glusterfs
4. OpenStack Cinder
5. vSphere
6. CSI Provisioner
7. vCP Provisioner
8. Ceph RBD

Parameters

The provisioners include the following providers:

9 Quo Byte

10 Azure Disk

11 Azure File

12 Portworx Volume

13 ScaleIO

14 StorageOS

15 Local

TECHNOLOGY

Dynamic Volume Provisioning

simplilearn

Overview of Dynamic Volume Provisioning

Dynamic volume provisioning allows storage volumes to be created on-demand.

A cluster administrator can define **StorageClass** objects as per the need.

Automatic storage provision is done upon user request.

The dynamic provisioning feature does away with the need for cluster administrators to provision storage in advance.

Enabling Dynamic Provisioning

To enable dynamic provisioning, a cluster administrator needs to create, in advance, one or more **StorageClass** objects.

Example:

```
apiVersion:   storage.k8s.io/v1
kind:   StorageClass
metadata:
    name: custom-managed-premium
provisioner: disk.csi.azure.com
parameters:
   cachingmode: ReadOnly
   kind: Managed
   storageaccounttype: Premium_LRS
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

Usage of Dynamic Provisioning

A user would create the PersistentVolumeClaim as shown here:

Example:

```
apiVersion:  v1
Kind:  PersistentVolumeClaim
Metadata:
   name: claim1
Spec:
   accessModes:
      - ReadWriteOnce
   storageClassName: custom-managed-premium
   resources:
      requests:
          storage:  30Gi
```

Defaulting Behavior

This is how a cluster administrator enables defaulting behavior:

Marking one StorageClass object as default

Ensuring that the Default StorageClass Admission Controller is enabled on the API server

Configuring Multi-Container Pod with RWX Access Using PV and PVC

Duration: 15 mins

Problem Statement:

You have been asked to configure multi-container pods with ReadWriteMany (RWX) access in Kubernetes using PersistentVolume (PV) and PersistentVolumeClaim (PVC) for shared storage and data operations.

Assisted Practice: Guidelines

Steps to be followed:

1. Create PersistentVolume
2. Create PersistentVolumeClaim
3. Deploy a pod in a new namespace
4. Demonstrate shared storage and data operations
5. Continue data operations

TECHNOLOGY

Storage Capacity

Storage Capacity

Storage capacity is limited and may vary, depending on the node on which a pod runs.

Tracking storage capacity is supported for container storage interface (CSI) drivers and should be enabled when a CSI driver is installed.

API

There are two API extensions for this feature:

CSI storage capacity objects

CSIDriverSpec. storage capacity field

Criteria Scheduling

Scenarios when storage capacity information is used by the Kubernetes scheduler:

1. The CSIStorageCapacity feature gate is true.
2. A pod uses a Volume that has not been created.
3. Volume uses a StorageClass which references a CSI driver and uses WaitForFirstConsumer volume binding mode.
4. The CSIDriver object for the driver has StorageCapacity is set to true.

Scheduling

The system compares the volume’s size with the capacity listed in CSI storage capacity objects, considering the topology that includes the node.

For volumes set to the immediate volume binding mode, the storage driver determines the volume's creation location.

Scheduling processes do not take storage capacity into account for CSI ephemeral volumes.

Rescheduling

After selecting a node for a pod that uses WaitForFirstConsumer volumes, the system requests the CSI storage driver to create the volume, ensuring its availability on the chosen node.

Tracking storage capacity enhances the likelihood of successful scheduling on the first attempt.

However, when a pod utilizes multiple volumes, scheduling might permanently fail.

Enable Storage Capacity Tracking

Storage capacity tracking is an alpha feature. It functions only when the CSIStorageCapacity feature gate and the storage.k8s.io/v1alpha1 API groups are enabled. The code snippet given here may be used to list CSIStorageCapacity objects.

Example:

```
#A quick check whether a Kubernetes cluster supports the feature is to
list CSIStorageCapacity objects with:

kubectl get csistoragecapacities --all-namespaces

#If the cluster supports CSIStorageCapacity, the response is either a
list of CSIStorageCapacity objects or:

No resources found

#If not supported, this error is printed instead:

error: the server doesn't have a resource type "csistoragecapacities"
```

TECHNOLOGY

Node-Specific Volume Limits

Introduction to Node-Specific Volume Limits

Cloud providers, including Google, Amazon, and Microsoft, typically impose limits on the number of volumes attachable to a single node.

It is crucial for Kubernetes to adhere to these limits to prevent the pods scheduled on a node from encountering issues.

Kubernetes Default Limits

The Kubernetes scheduler has default limits on the number of volumes that can be attached to a node.

Cloud service	Maximum volumes per node
Amazon Elastic Block Store (EBS)	39
Google Persistent Disk	16
Microsoft Azure Disk Storage	16

Custom Limits

Limits can be changed by setting the value of the **KUBE_MAX_PD_VOLS** environment variable, and then starting the Scheduler.

Use caution if you set a limit that is higher than the default limit.

Dynamic Volume Limits

Dynamic volume limits are supported for the following:

- Amazon EBS
- Google Persistent Disk
- Azure Disk
- CSI

Key Takeaways

- A pod can use any number of volumes and volume types simultaneously while working with Kubernetes.
- A StorageClass provides a way for administrators to describe the classes of storage that they offer.
- The CSI volume cloning feature adds support for specifying existing PVCs in the dataSource field to indicate that a user would like to clone a volume.
- Tracking storage capacity is supported for container storage interface (CSI) drivers and needs to be enabled when installing a CSI driver.

Deploying WordPress and MySQL Using PersistentVolume

Duration: 30 Min.

Project agenda:

To deploy WordPress and MySQL on Kubernetes with PersistentVolume using NFS and hostPath for web-based access

Description:

This project involves deploying WordPress and MySQL on a Kubernetes cluster, leveraging PersistentVolume with NFS and hostPath configurations for hosting a web-based WordPress application with data persistence and accessibility.

Expected Deliverables:

A fully deployed Kubernetes cluster, MySQL, and WordPress applications with PersistentVolume, allowing web-based access to the WordPress site.

LESSON-END PROJECT

Deploying WordPress and MySQL Using PersistentVolume

Duration: 30 Min.

Steps to be performed:

1. Configure the NFS kernel server
2. Set the permissions
3. Configure the NFS common on client machines
4. Create a MySQL manifest file and deploy it using NFS-based PersistentVolume
5. Create a WordPress manifest file and deploy it using hostpath-based PersistentVolume

LESSON-END PROJECT

TECHNOLOGY

Thank You