PersistentVolume Kubernetes

Persistent Volumes are Kubernetes objects that represent storage resources in your cluster. PVs work in conjunction with Persistent Volume Claims (PVCs), another type of object which permits Pods to request access to PVs. To successfully utilize persistent storage in a cluster, you'll need a PV and a PVC that connects it to your Pod.

A *PersistentVolume* (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using storage classes . It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

A *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. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany, ReadWriteMany, or ReadWriteOncePod)

Types of Persistent Volume

Kubernetes supports several persistent volume types that alter where and how your data is stored:

- **local** – Data is stored on devices mounted locally to your cluster's Nodes.

- **hostPath** – Stores data within a named directory on a Node (this is designed for testing purposes and doesn't work with multi-Node clusters).
- **nfs** – Used to access Network File System (NFS) mounts.
- **iscsi** – iSCSI (SCSI over IP) storage attachments.
- **csi** – Allows integration with storage providers that support the container storage Interface (CNI) specification, such as the block storage services provided by cloud platforms.
- **cephfs** – Allow the use of CephFS volumes.
- **fc** – Fibre Channel (FC) storage attachments.
- **rbd** – Rados Block Device (RBD) volumes.

Persistent Volume access modes

Persistent Volumes support four different access modes that define how they're mounted to Nodes and Pods:

- **`ReadWriteOnce (RWO)`** – The volume is mounted with read-write access for a *single* Node in your cluster. Any of the Pods running on that Node can read and write the volume's contents.

- **`ReadOnlyMany (ROX)`** – The volume can be concurrently mounted to any of the Nodes in your cluster, with read-only access for any Pod.
- **`ReadWriteMany (RWX)`** – Similar to ReadOnlyMany, but with read-write access.
- **`ReadWriteOncePod (RWOP)`** – This new variant, introduced as a beta feature in kubernets1.27, enforces that read-write access is provided to a *single* Pod. No other Pods in the cluster will be able to use the volume simultaneously.

The lifecycles of PVs and PVCs

PVs and PVCs have their own lifecycles, which describe the current status of the volume and whether it's in use. There are four main stages:

1. Provisioning

At the Provisioning stage, the PV is created and its storage is allocated using the selected driver.

Provisioning can occur manually by creating a PersistentVolume object in your cluster, or dynamically, by adding a PVC that refers to an

unknown PV. After provisioning, the PV will exist in your cluster, but won't be actively providing storage.

2. Binding

Binding occurs when a cluster user adds a PVC that claims the PV. The PV will enter this state automatically when dynamic provisioning is used, because you'll have already created the PVC.

Kubernetes automatically watches for new PVCs and binds them to the PVs they reference. Each PV can only be bound to a single PVC at a time. Once a PVC claims a PV, the volume will be Bound, but won't necessarily be used by a Pod.

3. Using

A volume enters use once its PVC is consumed by a Pod. When Pods reference PVCs, Kubernetes automatically mounts the correct volume into the Pod's filesystem. In this state, the PV is actively providing storage to an application in your cluster.

4. Reclaiming

Storage isn't always required indefinitely. Users can delete the PVC to relinquish access to the PV. When this happens, the storage used by the PV is "reclaimed."

The reclaim behavior is customizable and allows you to either delete the provisioned storage, recycle it by emptying its contents, or retain it as-is for future reuse.

Demo

Example: How to create and use a Persistent Volume in On-premise Env. for local-storage.

In this Demo we will install Grafana and Mount Grafana Data in Node host using stateful deployment and with storage as no provisoner.

Note:-
no-provisoner means static provisoner is used means we have to create our own PV file dynamic PV won't be created.

Step 1:- Install StorageClass

Create a file and copy the below yaml contents and apply.

- Kubectl apply -f filename -n monitoring

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

```
kind: StorageClass
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

- Kubectl get storageclass

```
NAME            PROVISIONER                    RECLAIMPOLICY   VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
local-storage   kubernetes.io/no-provisioner   Delete          WaitForFirstConsumer   false                  27h
```

Step 2:- Create PV

Create PV file and copy the below contents and apply

- Kubectl apply -f filename -n monitoring

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: grafana-pv
  namespace: monitoring
  labels:
    type: grafana-pv
spec:
  capacity:
    storage: 2Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  local:
    path: /home/user/grafana
  nodeAffinity:
```

```
    required:
      nodeSelectorTerms:
        - matchExpressions:
          -  key: kubernetes.io/hostname
             operator: In
             values:
               - worker-node
```

```
NAME         CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                                 STORAGECLASS    VOLUMEATTRIBUTESCLASS
  REASON   AGE
grafana-pv   2Gi        RWO            Retain           Bound    monitoring/grafana-storage-grafana-0   local-storage   <unset>
           18h
```

Step 3: Create stateful File with PVC

Create PV file and copy the below contents and apply

- Kubectl apply -f filename -n monitoring

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: grafana
  namespace: monitoring
spec:
  serviceName: "grafana"
  replicas: 1
  selector:
    matchLabels:
      app: grafana
  template:
    metadata:
      labels:
        app: grafana
    spec:
```

```
    securityContext:
      fsGroup: 472
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
              - "worker-node"
    containers:
    - name: grafana
      image: grafana/grafana:latest
      ports:
      - containerPort: 3000
      volumeMounts:
      - name: grafana-storage
        mountPath: "/var/lib/grafana"
      securityContext:
        runAsUser: 472
        runAsGroup: 472
volumeClaimTemplates:
- metadata:
    name: grafana-storage
  spec:
    accessModes: [ "ReadWriteOnce" ]
    storageClassName: "local-storage"
    resources:
      requests:
        storage: 2Gi
```

```
  selector:
    matchLabels:
      type: grafana-pv (Using this it will find the required PV to
bound)
```

```
NAME                          STATUS   VOLUME       CAPACITY   ACCESS MODES   STORAGECLASS    VOLUMEATTRIBUTESCLASS   AGE
grafana-storage-grafana-0     Bound    grafana-pv   2Gi        RWO            local-storage   <unset>                 20h
```

Once its bound you will able to see volume is target node.

```
alerting  csv  grafana.db  pdf  plugins  png
```

Step 3:- Create Service File for Grafana.

Create a Service file and copy the below contents and apply.

- Kubectl apply -f filename -n monitoring

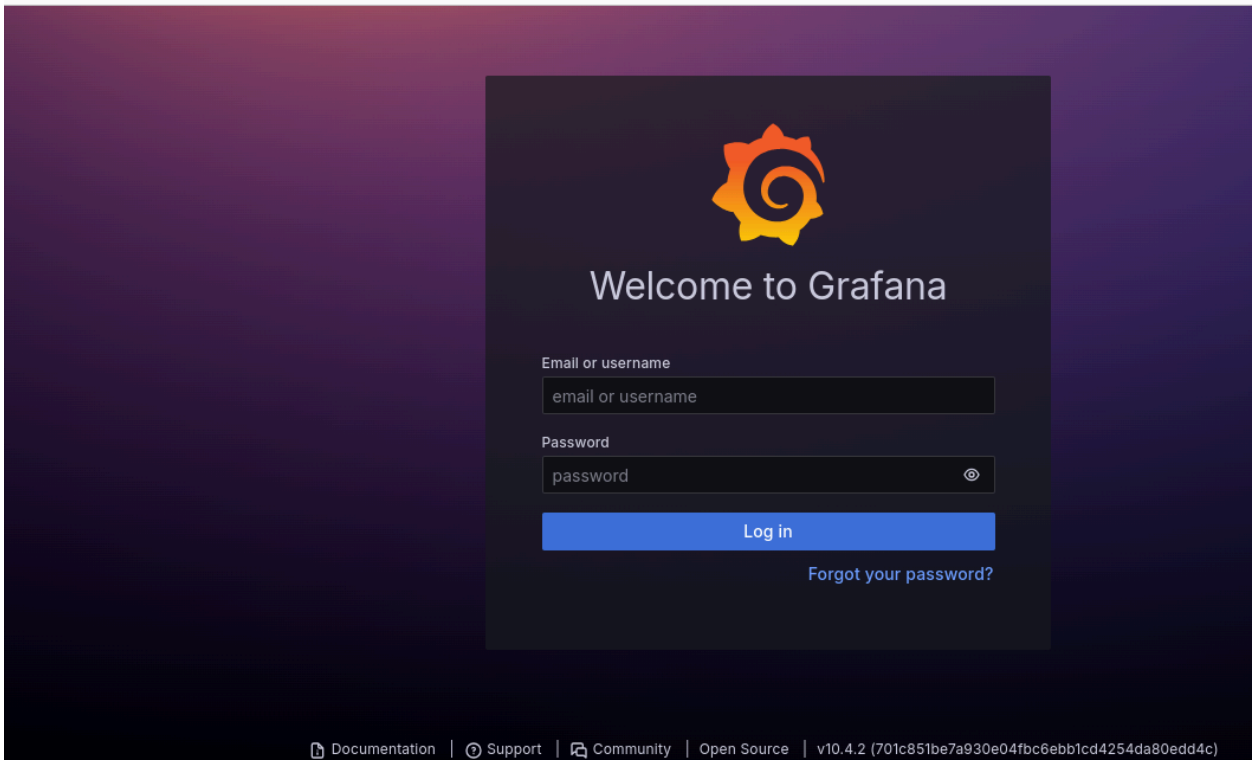
```
apiVersion: v1
kind: Service
metadata:
  name: grafana-service
  namespace: monitoring
spec:
  type: LoadBalancer
  ports:
  - port: 3100  # Port exposed by the service
    targetPort: 3000  # Port targeted in the pod
  selector:
    app: grafana
```

Step 4:- create Ingress File to access grafana Externally.

Create Ingress file and copy the below contents and apply.

- Kubectl apply -f filename -n monitoring

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: grafana-ingress
  namespace: monitoring
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  tls:
  - hosts:
    - host.path.com
    secretName: sslcert
  rules:
  - host: host.path.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: grafana-service
            port:
              number: 3100
```


Welcome to Grafana
Email or username
email or username
Password
password
Log in
Forgot your password?
Documentation
Support
Community
Open Source
v10.4.2 (701c851be7a930e04fbc6ebb1cd4254da80edd4c)