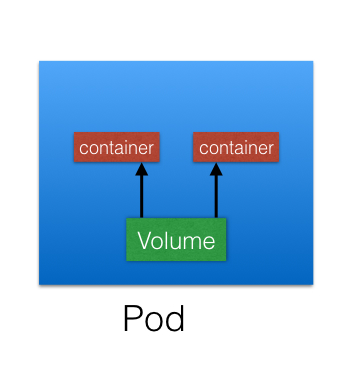
**Kubernetes-volume-management**

**To back a Pod with a persistent storage, Kubernetes uses Volumes. In this chapter, we will learn about Volumes and their types. We will also discuss about Persistent Volume and Persistent Volume Claim objects, which help us attach storage volumes to Pods.**

**As we know, containers, which create the Pods, are ephemeral in nature. All data stored inside a container is deleted if the container crashes. However, kubelet will restart it with a clean state, which means that it will not have any of the old data.**

**Volumes**

To overcome this problem, Kubernetes uses Volumes. A Volume is essentially a directory backed by a storage medium. The storage medium and its content are determined by the Volume Type.



In Kubernetes, a Volume is attached to a Pod and shared among the containers of that Pod. The Volume has the same life span as the Pod, and it outlives the containers of the Pod - this allows data to be preserved across container restarts.

**VolumeTypes**

A directory which is mounted inside a Pod is backed by the underlying Volume Type. A Volume Type decides the properties of the directory, like size, content, etc. Some of the Volume Types are:

* + **emptyDir**  
    An **empty** Volume is created for the Pod as soon as it is scheduled on the Worker Node. The Volume's life is tightly coupled with the Pod. If the Pod dies, the content of **emptyDir** is deleted forever.
  + **hostPath**  
    With the **hostPath** Volume Type, we can share a directory from the host to the Pod. If the Pod dies, the content of the Volume is still available on the host.
  + **gcePersistentDisk**  
    With the **gcePersistentDisk** Volume Type, we can mount a [Google Compute Engine (GCE) persistent disk](https://cloud.google.com/compute/docs/disks/) into a Pod.
  + **awsElasticBlockStore**  
    With the **awsElasticBlockStore** Volume Type, we can mount an [AWS EBS Volume](https://aws.amazon.com/ebs/) into a Pod.
  + **nfs**  
    With [nfs](https://en.wikipedia.org/wiki/Network_File_System), we can mount an NFS share into a Pod.
  + **iscsi**  
    With [iscsi](https://en.wikipedia.org/wiki/ISCSI), we can mount an iSCSI share into a Pod.
  + **secret**  
    With the **secret** Volume Type, we can pass sensitive information, such as passwords, to Pods. We will take a look at an example in a later chapter.
  + **persistentVolumeClaim**  
    We can attach a [Persistent Volume](https://kubernetes.io/docs/concepts/storage/persistent-volumes/) to a Pod using a **persistentVolumeClaim**.

## https://www.kubermatic.com/blog/keeping-the-state-of-apps-1-introduction-to-volume-and-volumemounts/ #### Reference Site

## Kubernetes Volumes and volumeMounts

A **volume** usage entails the declaration of the volume in a Pod by specifying a **“volumes”** property under the **spec (spec.volumes)** field in a Pod manifest file, followed by the volume in an array format. The configuration will look like this:

spec:

volumes:

- name: xyz

A **volumeMount**, on the other hand, entails mounting of the declared volume into a container in the same Pod. A **“volumeMounts”** property (spec.container.volumeMounts), as well as the “name” property which is the volume name to be mounted and the mountPath field where the volume will be mounted, are declared in the container in a Pod. The configuration will look like this:

spec:

containers:

- name: my-app

image: nginx

volumeMounts:

- name: xyz

mountPath: /app/config

Volume and volumeMounts go hand in hand. You can not create a volume without mounting it or mount a volume that has not been created.

spec:

containers:

- name: my-app

image: nginx

volumeMounts:

- name: xyz

mountPath: /app/config

volumes:

- name: xyz

## EmptyDir Volume Type

An emptyDir volume is a volume type that is first created when a Pod is assigned to a Node. Its lifespan is dependent on the lifecycle of the Pod on that Node but recreates when the containers crash or restart. When a Pod dies, crashes, or is removed from a Node, the data in the emptyDir volume is deleted and lost. This type of volume **is suitable for temporary data storage.**

Example:

volumeMounts:

- mountPath: /cache

name: my-volume

volumes:

- name: my-volume

emptyDir: {}

**manifest file**

apiVersion: v1

kind: Pod

metadata:

name: myapp

spec:

containers:

- name: my-app

image: nginx

ports:

- containerPort: 8080

imagePullPolicy: Always

volumeMounts:

- name: my-volume

mountPath: /app

volumes:

- name: my-volume

emptyDir: {}

**Commands**

Create the Pod using kubectl create command:

$ kubectl create -f emptyDir.yaml

pod/myapp created

check the status of the Pod to see if it is running:

$ kubectl get pod

NAME READY STATUS RESTARTS AGE

myapp 1/1 Running 0 15s

Check the Pod description for more information about the volume in Pod:

$ kubectl describe pod myapp

Exec into the Pod and perform some basic commands:

$ kubectl exec -it myapp -- bin/bash

Check the volume in the directory for existing data. In this case, it should be empty.

$ root@myapp:/# ls app/

 Create a file and write some data into it:

root@myapp:/# echo I love Kubermatic > app/new-file

 Check to see if the data is stored:

root@myapp:/# ls app/

Now that you can see that it is stored, display the content of the data using the below command:

$ cat app/new-file

I love Kubermatic

Exit the Pod and perform a clean up by deleting the Pod using kubectl delete command:

$ kubectl delete pod myapp

pod "myapp" deleted

## HostPath Volume Type

hostPath volume type is a durable volume type that mounts a directory from the host Node’s filesystem into a Pod. The file in the volume remains intact even if the Pod crashes, is terminated or is deleted. It is important that the directory and the Pod are created or scheduled on the same Node.

Example

volumes:

- name: hostpath-volume # The name of the volume

hostPath:

path: /data

### Create and use a hostPath Volume in a Pod

**manifest file**

apiVersion: v1

kind: Pod

metadata:

name: myapp

spec:

containers:

- name: my-app

image: nginx

ports:

- containerPort: 8080

volumeMounts:

- name: my-volume

mountPath: /app

volumes:

- name: my-volume

hostPath:

path: /mnt/vpath

**Commands**

 Create a Pod with the manifest file above:

$ kubectl create -f hostpath-volume.yaml

 Check the status of the Pod using kubectl get command:

$kubectl get pod

NAME READY STATUS RESTARTS AGE

myapp 1/1 Running 0 10m

Exec into the Pod and create a file in the directory:

$kubectl exec -it myapp -- /bin/bash

Change to the /app directory:

root@myapp:/# cd /app ## Where /app is the mountPath value from the YAML manifest file.

Create a file using echo command and store some data in the file:

root@myapp:/app# echo "I love Kubermatic" > file.txt

Create a file using echo command and store some data in the file:

root@myapp:/app# echo "I love Kubermatic" > file.txt

Now, ssh into the Node to check if the data created in the /app directory in the Pod can be found in the /mnt/vpath in the Node.

$ ssh ubuntu@x.xx.xxx.xxx

ubuntu@x.xx.xxx.xxx:~$ cd /mnt/vpath

ubuntu@x.xx.xxx.xxx:/mnt/vpath$

ubuntu@x.xx.xxx.xxx:/mnt/vpath$ ls

file.txt

ubuntu@x.xx.xxx.xxx:/mnt/vpath$ cat file.txt

I love Kubermatic

**NOTE:** The Pod must be running in the same Node.

## Checking the Persistent State of a hostPath Volume

 Delete the Pod using kubectl delete command:

$ kubectl delete pod myapp

Check the status of the Pod:

$ kubectl get pod

No resources found in default namespace.

Now that the Pod has been deleted, ssh into the Node and perform step 8 and 9.

ubuntu@x.xx.xxx.xxx:~$ cd /mnt/vpath

ubuntu@x.xx.xxx.xxx:/mnt/vpath$

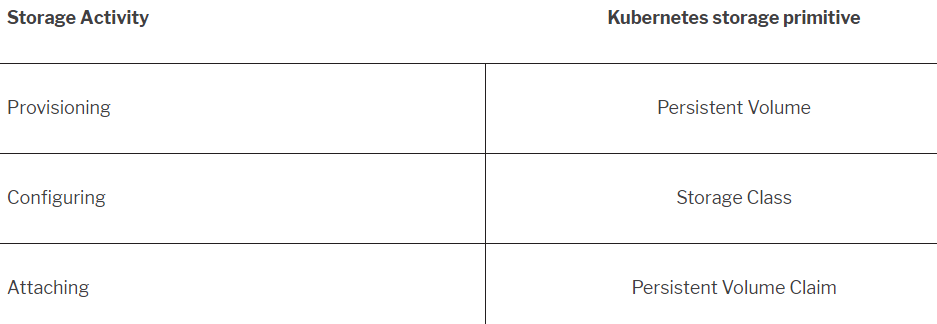
ubuntu@x.xx.xxx.xxx:/mnt/vpath$ ls

file.txt

ubuntu@x.xx.xxx.xxx:/mnt/vpath$ cat file.txt

I love Kubermatic

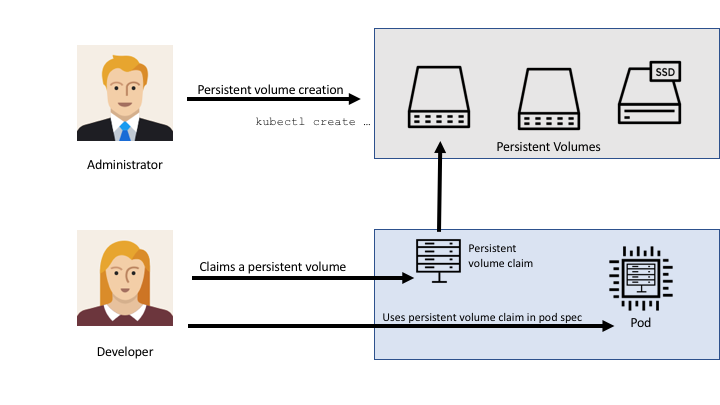
**PROCESS**



**Persistent Volumes**

In the containerized world, we would like to follow similar rules, but it becomes challenging, given the many Volume Types we have seen earlier. Kubernetes resolves this problem with the Persistent Volume subsystem, which provides APIs for users and administrators to manage and consume storage. To manage the Volume, it uses the PersistentVolume (PV) API resource type, and to consume it, it uses the PersistentVolumeClaim (PVC) API resource type.

A Persistent Volume is a network attached storage in the cluster, which is provisioned by the administrator.



Some of the Volume Types that support managing storage using Persistent Volumes are:

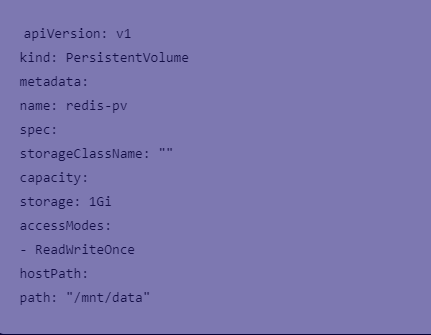
* + GCEPersistentDisk
  + AWSElasticBlockStore
  + **emptyDir**
  + hostpath
  + nfs

## Why do we use persistent volumes?

The most common use case for Persistent volumes in Kubernetes is for databases. Obviously a database needs to have access to its data at all times, and by leveraging PVs, we can start using databases like MySQL, Cassandra, CockroachDB and even MS SQL for our applications.

* Each pod is created (with appropriate config and environment variables required)
* A persistent volume is attached to the pod (via a persistent volume claim)
* The claimed storage is mounted inside the pod as required.

## Creating a persistent volume

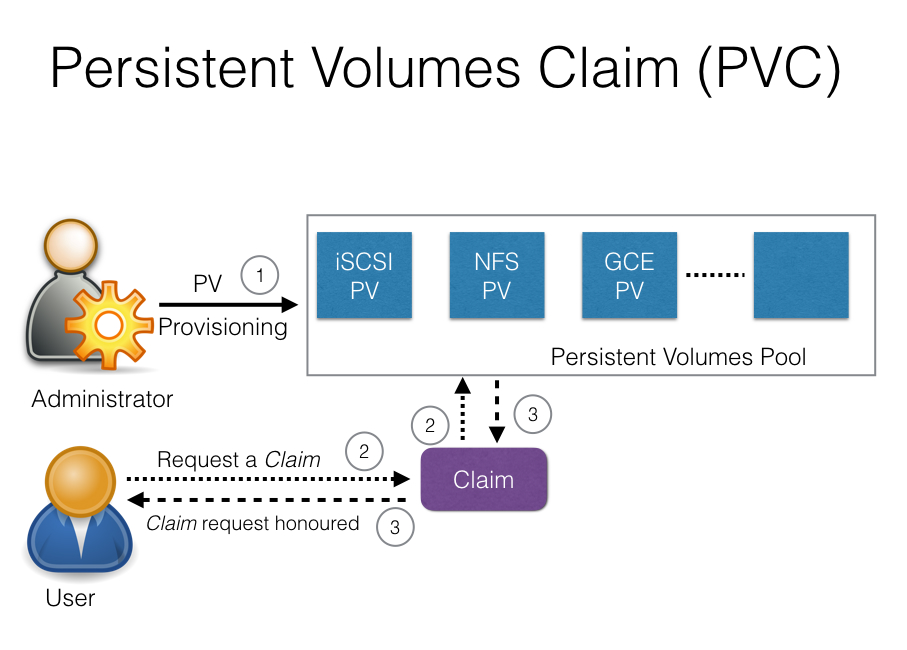


kubectl apply -f redis-pv.yaml

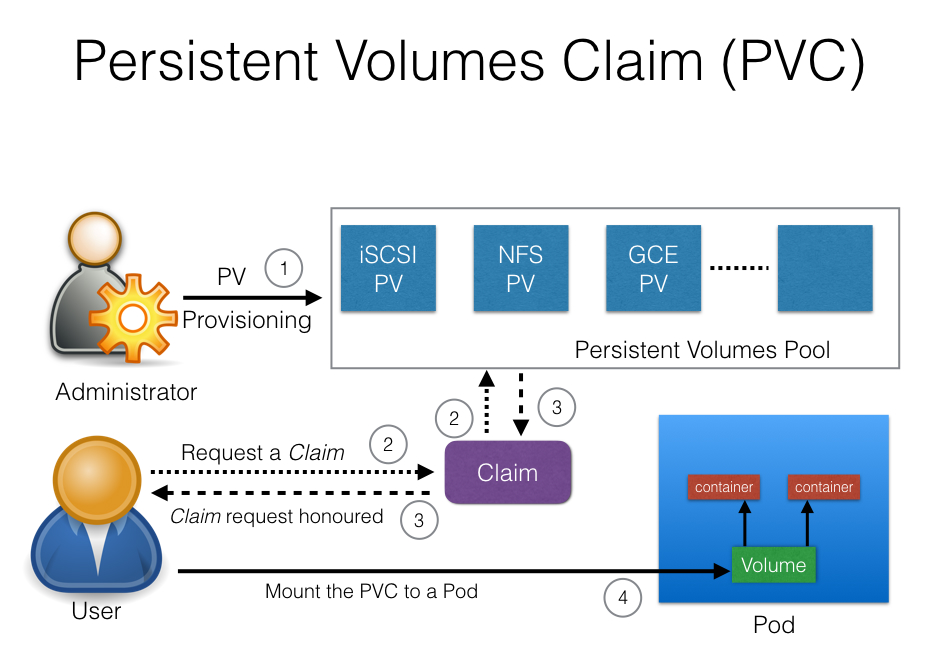
kubectl get pv

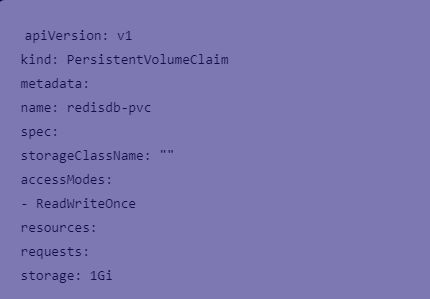
https://www.portworx.com/wp-content/uploads/2018/02/image3.png

**PersistentVolumeClaim** (PVC) is a request for storage by a user. Users request for Persistent Volume resources based on size, access modes, etc. Once a suitable Persistent Volume is found, it is bound to a Persistent Volume Claim

. 

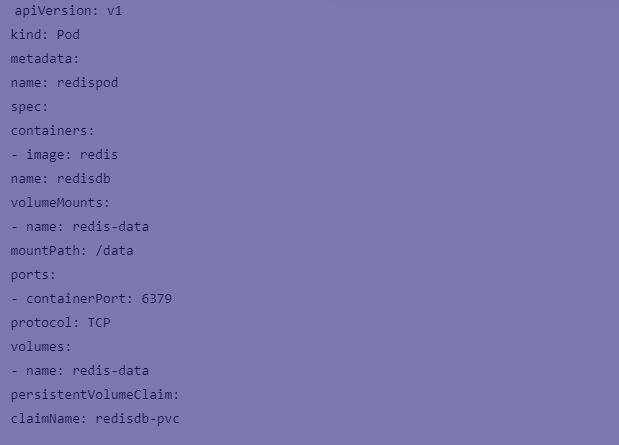
After a successful bind, the PersistentVolumeClaim resource can be used in a Pod.





kubectl apply -f redis-pvc.yaml

VI redispod.yaml:



kubectl apply -f redispod.yaml

kubectl get pods

Connect to the container and write some data.

kubectl exec -it redispod redis-cli