Selfie Shots - Tools 304 Kubernetes

Persistent University

Volumes Overview

Persistent

Key learning points :

1. Volumes Overview
2. Volume Types
3. Volume Spec
4. Persistent Volumes and Persistent Volume Claim
5. Dynamic Provisioning

Volumes Overview

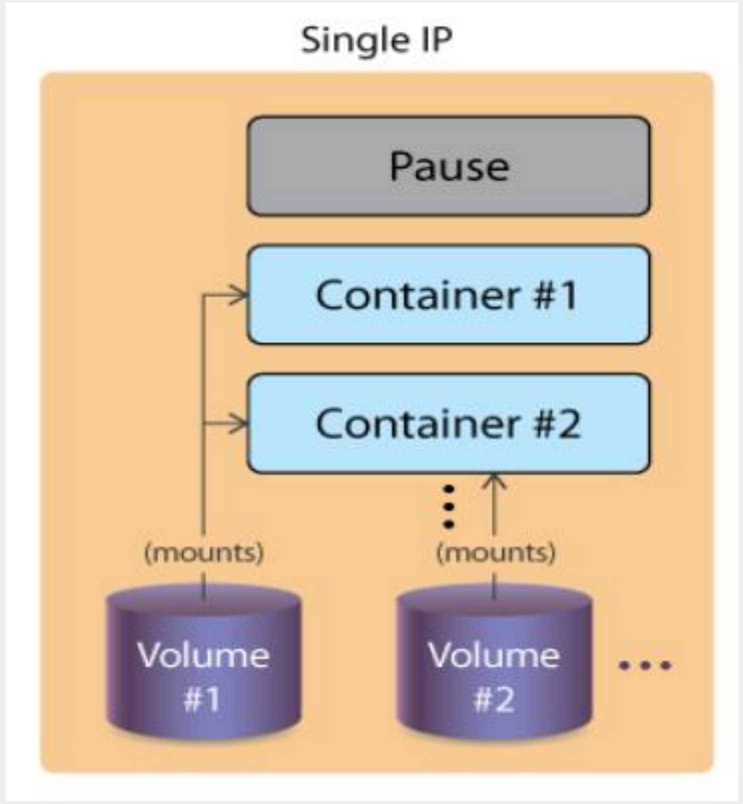
- Container engines have traditionally not offered storage that outlives the container. As containers are considered transient, this could lead to a loss of data, or complex exterior storage options. A Kubernetes *volume* shares the Pod lifetime, not the containers within. Should a container terminate, the data would continue to be available to the new container.
- A *volume* is a directory, possibly pre-populated, made available to containers in a Pod. The creation of the directory, the backend storage of the data and the contents depend on the volume type. As of v1.8, there are 25 different volume types ranging from **rbd** to gain access to Ceph, to **NFS**, to dynamic volumes from a cloud provider like Google's **gcePersistentDisk**. Each has particular configuration options and dependencies.
- An *alpha* feature to v1.9 is the *Container Storage Interface* (*CSI*) with the goal of an industry standard interface for container orchestration to allow access to arbitrary storage systems. Currently, volume plugins are "in-tree", meaning they are compiled and built with the core Kubernetes binaries. This "out-of-tree" object will allow storage vendors to develop a single driver and allow the plugin to be containerized. This will replace the existing **Flex** plugin which requires elevated access to the host node, a large security concern.
- Should you want your storage lifetime to be distinct from a Pod, you can use *Persistent Volumes*. These allow for empty or pre-populated volumes to be claimed by a Pod using a *Persistent Volume Claim*, then outlive the Pod. Data inside the volume could then be used by another Pod, or as a means of retrieving data.

Volumes

- A Pod specification can declare one or more volumes and where they are made available. Each requires a name, a type, and a mount point. The same volume can be made available to multiple containers within a Pod, which can be a method of container-to-container communication. A volume can be made available to multiple Pods, with each given an **access mode** to write. There is no concurrency checking, which means data corruption is probable, unless outside locking takes place.
- particular **access** mode is part of a Pod request. As a request, the user may be granted more, but not less access, though a direct match is attempted first. The cluster groups volumes with the same mode together, then sorts volumes by size, from smallest to largest. The claim is checked against each in that access mode group, until a volume of sufficient size matches. The three access modes are **RWO** (**ReadWriteOnce**), which allows read-write by a single node, **ROX** (**ReadOnlyMany**), which allows read-only by multiple nodes, and **RWX** (**ReadWriteMany**), which allows read-write by many nodes.
- When a volume is requested, the local **kubelet** uses the **kubelet_pods.go** script to map the raw devices, determine and make the mount point for the container, then create the symbolic link on the host node filesystem to associate the storage to the container. The API server makes a request for the storage to the **StorageClass** plugin, but the specifics of the requests to the backend storage depend on the plugin in use.

Volumes (Cntd)

- If a request for a particular **StorageClass** was not made, then the only parameters used will be access mode and size. The volume could come from any of the storage types available, and there is no configuration to determine which of the available ones will be used.



Persistent

Volume Types

- There are several types that you can use to define volumes, each with their pros and cons. Some are local, and many make use of network-based resources.
- In **GCE** or **AWS**, you can use volumes of type **GCEpersistentDisk** or **awsElasticBlockStore**, which allows you to mount **GCE** and **EBS** disks in your Pods, assuming you have already set up accounts and privileges.
- **emptyDir** and **hostPath** volumes are easy to use. As mentioned, **emptyDir** is an empty directory that gets erased when the Pod dies, but is recreated when the container restarts. The **hostPath** volume mounts a resource from the host node filesystem. The resource could be a directory, file socket, character, or block device. These resources must already exist on the host to be used. There are two types, **DirectoryOrCreate** and **FileOrCreate**, which create the resources on the host, and use them if they don't already exist.
- **NFS** (Network File System) and **iSCSI** (Internet Small Computer System Interface) are straightforward choices for multiple readers scenarios.
- **rbd** for block storage or **CephFS** and **GlusterFS**, if available in your **Kubernetes** cluster, can be a good choice for multiple writer needs.
- Besides the volume types we just mentioned, there are many other possible, with more being added: **azureDisk**, **azureFile**, **csi**, **downwardAPI**, **fc** (fibre channel), **flocker**, **gitRepo**, **local**, **projected**, **portworxVolume**, **quobyte**, **scaleIO**, **secret**, **storageos**, **vsphereVolume**, **persistentVolumeClaim**, etc.

Volume Spec

- One of the many types of storage available is an **emptyDir**. The kubelet will create the directory in the container, but not mount any storage. Any data created is written to the shared container space. As a result, it would not be persistent storage. When the Pod is destroyed, the directory would be deleted along with the container.

```
apiVersion: v1
kind: Pod
metadata:
    name: busybox
    namespace: default
spec:
    containers:
    - image: busybox
      name: busy
      command:
        - sleep
        - "3600"
      volumeMounts:
    - mountPath: /scratch
      name: scratch-volume
    volumes:
    - name: scratch-volume
            emptyDir: {}
```

- The YAML file above would create a Pod with a single container with a volume named **scratch-volume** created, which would create the **/scratch** directory inside the container.

Shared Volume example

- The following YAML file creates a pod with two containers, both with access to a shared volume:

```
containers:
    - image: busybox
  volumeMounts:
    - mountPath: /busy
  name: test
  name: busy
    - image: busybox
  volumeMounts:
    - mountPath: /box
  name: test
  name: box
  volumes:
    - name: test
    emptyDir: {}
```

```
$ kubectl exec -ti busybox -c box -- touch /box/foobar
```

```
$ kubectl exec -ti busybox -c busy -- ls -l /busy total 0
-rw-r--r-- 1 root root 0 Nov 19 16:26 foobar
```

- You could use **emptyDir** or **hostPath** easily, since those types do not require any additional setup, and will work in your Kubernetes cluster.
- Note that one container wrote, and the other container had immediate access to the data. There is nothing to keep the containers from overwriting the other's data. Locking or versioning considerations must be part of the application to avoid corruption.

Persistent Volumes and Claims

- A persistent volume (pv) is a storage abstraction used to retain data longer then the Pod using it. Pods define a volume of type **PersistentVolumeClaim** (**pvc**)with various parameters for size and possibly the type of backend storage known as its **StorageClass**. The cluster then attaches the **PersistentVolume**.
- Kubernetes will dynamically use volumes that are available, irrespective of its storage type, allowing claims to any backend storage.
- There are several phases to persistent storage:
- **Provisioning** can be from PVs created in advance by the cluster administrator, or requested from a dynamic source, such as the cloud provider.
- **Binding** occurs when a control loop on the master notices the PVC, containing an amount of storage, access request, and optionally, a particular **StorageClass**. The watcher locates a matching PV or waits for the **StorageClass** provisioner to create one. The PV must match at least the storage amount requested, but may provide more.
- The **use** phase begins when the bound volume is mounted for the Pod to use, which continues as long as the Pod requires.
- **Releasing** happens when the Pod is done with the volume and an API request is sent, deleting the PVC. The volume remains in the state from when the claim is deleted until available to a new claim. The resident data remains depending on the **PersistentVolumeReclaimPolicy**.
- The **reclaim** phase has three options:
  - **Retain**, which keeps the data intact, allowing for an administrator to handle the storage and data.
  - **Delete** tells the volume plugin to delete the API object, as well as the storage behind it.
  - The **Recycle** option runs an **rm -rf /mountpoint** and then makes it available to a new claim. With the stability of dynamic provisioning, the **Recycle** option is planned to be deprecated.

```
$ kubectl get pv
$ kubectl get pvc
```

Persistent Volume

- The following example shows a basic declaration of a **PersistentVolume** using the **hostPath** type.
- kind: PersistentVolume

```
apiVersion: v1
metadata:
  name: 10Gpv01
  labels:
     type: local
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/somepath/data01"
```

- Each type will have its own configuration settings. For example, an already created Ceph or GCE Persistent Disk would not need to be configured, but could be claimed from the provider.
- Persistent volumes are not a namespaces object, but persistent volume claims are. An alpha feature of v1.9 allows for static provisioning of Raw Block Volumes, which currently support the Fibre Channel plugin.

Persistent Volume Claim

- With a persistent volume created in your cluster, you can then write a manifest for a claim and use that claim in your pod definition. In the Pod, the volume uses the **PersistentVolumeClaim**.

-
  ```
  kind: PersistentVolumeClaim
  apiVersion: v1
  metadata:
    name: myclaim
  spec:
    accessModes:
      - ReadWriteOnce
    resources:
      requests:
          storage: 8GI
  ```
- (In the Pod)
  ```
  ....
  spec:
    containers:
  ....
    volumes:
      - name: test-volume
        persistentVolumeClaim:
          claimName: myclaim
  ```

Persistent Volume Claim (Cntd.)

- The Pod configuration could also be as complex as this:
- volumeMounts:

```
volumeMounts:
      - name: Cephpd
        mountPath: /data/rbd
  volumes:
    - name: rbdpd
      rbd:
        monitors:
        - '10.19.14.22:6789'
        - '10.19.14.23:6789'
        - '10.19.14.24:6789'
        pool: k8s
        image: client
        fsType: ext4
        readOnly: true
        user: admin
        keyring: /etc/ceph/keyring
        imageformat: "2"
        imagefeatures: "layering"
```

Dynamic Provisioning

- While handling volumes with a persistent volume definition and abstracting the storage provider using a claim is powerful, a cluster administrator still needs to create those volumes in the first place. Starting with Kubernetes v1.4, **Dynamic Provisioning** allowed for the cluster to request storage from an exterior, pre-configured source. API calls made by the appropriate plugin allow for a wide range of dynamic storage use.
- The **StorageClass** API resource allows an administrator to define a persistent volume provisioner of a certain type, passing storage-specific parameters.
- With a **StorageClass** created, a user can request a claim, which the API Server fills via auto-provisioning. The resource will also be reclaimed as configured by the provider. **AWS** and **GCE** are common choices for dynamic storage, but other options exist, such as a **Ceph** cluster or iSCSI. Single, default class is possible via annotation.
- Here is an example of a **StorageClass** using GCE:

```
apiVersion: storage.k8s.io/v1        # Recently became stable
kind: StorageClass
metadata:
  name: fast                         # Could be any name
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

Summary

At the end of this session, we see that you are now able to

- Understand and create persistent volumes.
- Configure persistent volume claims.
- Manage volume access modes.
- Deploy an application with access to persistent storage.

Lab Exercise

- Attach shareable volume to containers within a Pod
- Create Persistent Volume and Persistent Volume Claim
- Attach to Pod