Learning Objectives

By the end of this chapter, you should be able to:

- Understand and create persistent volumes.
- Configure persistent volume claims.
- Manage volume access modes.
- Deploy an application with access to persistent storage.
- Discuss the dynamic provisioning of storage.
- Configure secrets and ConfigMaps.

VOLUMES AND DATA

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 v1.13, there were 27 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.

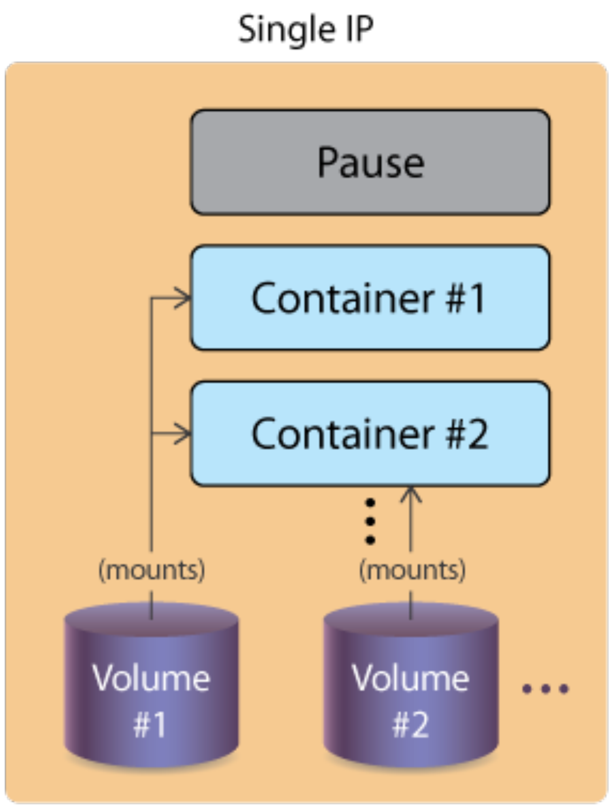
The Container Storage Interface (CSI) adoption enables the goal of an industry standard interface for container orchestration to allow access to arbitrary storage systems. Currently, volume plugins are "in-tree", meaning they are compiled and built with the core Kubernetes binaries. This "out-of-tree" 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.

There are two API objects which exist to provide data to a Pod already. Encoded data can be passed using a Secret and non-encoded data can be passed with a ConfigMap. These can be used to pass important data like SSH keys, passwords, or even a configuration file like **/etc/hosts**.

Introducing 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.



Kubernetes Pod Volumes

A 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:

- ReadWriteOnce, which allows read-write by a single node
- ReadOnlyMany, which allows read-only by multiple nodes
- ReadWriteMany, which allows read-write by many nodes.

Thus two pods on the same node can write to a ReadWriteOnce, but a third pod on a different node would not become ready due to a FailedAttachVolume error.

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.

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.

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.

```yaml
apiVersion: v1
kind: Pod
metadata:
    name: fordpinto
    namespace: default
spec:
    containers:
    - image: simpleapp
      name: gastank
      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.

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**, **CSIPersistentVolumeSource**, etc.

CSI allows for even more flexibility and decoupling plugins without the need to edit the core Kubernetes code. It was developed as a standard for exposing arbitrary plugins in the future.

Shared Volume Example

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

```
....
   containers:
   - name: alphacont
     image: busybox
     volumeMounts:
     - mountPath: /alphadir
       name: sharevol
   - name: betacont
     image: busybox
     volumeMounts:
     - mountPath: /betadir
       name: sharevol
   volumes:
   - name: sharevol
     emptyDir: {}
```

```
$ kubectl exec -ti exampleA -c betacont -- touch /betadir/foobar
$ kubectl exec -ti exampleA -c alphacont -- ls -l /alphadir

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 (**betacont**) wrote, and the other container (**alphacont**) 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 containerized 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.

Persistent Storage Phases

- Provision

Provisioning can be from PVs created in advance by the cluster administrator, or requested from a dynamic source, such as the cloud provider.

- Bind

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.

- Use

The **use** phase begins when the bound volume is mounted for the Pod to use, which continues as long as the Pod requires.

- Release

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**.

- Reclaim:

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 Persistent Volume 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. A beta feature of v1.13 allows for static provisioning of Raw Block Volumes, which currently support the Fibre Channel plugin, AWS EBS, Azure Disk and RBD plugins among others.

The use of locally attached storage has been graduated to a stable feature. This feature is often used as part of distributed filesystems and databases.

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
```

The Pod configuration could also be as complex as this:

```
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:

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

Using Rook for Storage Orchestration

In keeping with the decoupled and distributed nature of the Cloud technology, the Rook project allows orchestration of storage using multiple storage providers.

As with other agents of the cluster, Rook uses custom resource definitions (CRD) and a custom operator to provision storage according to the backend storage type, upon API call.

Several storage providers are supported:

- Ceph
- Cassandra
- CockroachDB
- EdgeFS Geo-Transparant Storage
- Minio Object Store
- Network File System (NFS)
- YugabyteDB.

Secrets

Pods can access local data using volumes, but there is some data you don't want readable to the naked eye. Passwords may be an example. Using the Secret API resource, the same password could be encoded or encrypted.

You can create, get, or delete secrets:

```
$ kubectl get secrets
```

Secrets can be encoded manually or via **kubectl create secret**:

```
$ kubectl create secret generic --help

$ kubectl create secret generic mysql --from-literal=password=root
```

A secret is not encrypted, only base64-encoded, by default. You must create an **EncryptionConfiguration** with a key and proper identity. Then, the kube-apiserver needs the **--encryption-provider-config** flag set to a previously configured provider, such as **aescbc** or **ksm**. Once this is enabled, you need to recreate every secret, as they are encrypted upon write.

Multiple keys are possible. Each key for a provider is tried during decryption. The first key of the first provider is used for encryption. To rotate keys, first create a new key, restart (all) kube-apiserver processes, then recreate every secret.

You can see the encoded string inside the secret with **kubectl**. The secret will be decoded and be presented as a string saved to a file. The file can be used as an environmental variable or in a new directory, similar to the presentation of a volume.

A secret can be made manually as well, then inserted into a YAML file:

```
$ echo LFTr@1n | base64
TEZUckAxbgo=

$ vim secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: LF-secret
data:
  password: TEZUckAxbgo=
```

Using Secrets via Environment Variables

A secret can be used as an environmental variable in a Pod. You can see one being configured in the following example:

```
...
spec:
    containers:
    - image: mysql:5.5
      env:
      - name: MYSQL_ROOT_PASSWORD
        valueFrom:
            secretKeyRef:
                name: mysql
                key: password
          name: mysql
```

There is no limit to the number of Secrets used, but there is a 1MB limit to their size. Each secret occupies memory, along with other API objects, so very large numbers of secrets could deplete memory on a host.

They are stored in the **tmpfs** storage on the host node, and are only sent to the host running Pod. All volumes requested by a Pod must be mounted before the containers within the Pod are started. So, a secret must exist prior to being requested.

Mounting Secrets as Volumes

You can also mount secrets as files using a volume definition in a pod manifest. The mount path will contain a file whose name will be the key of the secret created with the **kubectl create secret** step earlier.

```
...
spec:
    containers:
    - image: busybox
      command:
        - sleep
        - "3600"
      volumeMounts:
      - mountPath: /mysqlpassword
        name: mysql
      name: busy
    volumes:
    - name: mysql
        secret:
            secretName: mysql
```

Once the pod is running, you can verify that the secret is indeed accessible in the container:

```
$ kubectl exec -ti busybox -- cat /mysqlpassword/password
LFTr@1n
```

Portable Data with ConfigMaps

A similar API resource to Secrets is the ConfigMap, except the data is not encoded. In keeping with the concept of decoupling in Kubernetes, using a ConfigMap decouples a container image from configuration artifacts.

They store data as sets of key-value pairs or plain configuration files in any format. The data can come from a collection of files or all files in a directory. It can also be populated from a literal value.

A ConfigMap can be used in several different ways. A container can use the data as environmental variables from one or more sources. The values contained inside can be passed to commands inside the pod. A Volume or a file in a Volume can be created, including different names and particular access modes. In addition, cluster components like controllers can use the data.

Let's say you have a file on your local filesystem called **config.js**. You can create a ConfigMap that contains this file. The **configmap** object will have a data section containing the content of the file:

```
$ kubectl get configmap foobar -o yaml
kind: ConfigMap
apiVersion: v1
metadata:
    name: foobar
data:
    config.js: |
         {
...
```

ConfigMaps can be consumed in various ways:

- Pod environmental variables from single or multiple ConfigMaps
- Use ConfigMap values in Pod commands
- Populate Volume from ConfigMap
- Add ConfigMap data to specific path in Volume
- Set file names and access mode in Volume from ConfigMap data
- Can be used by system components and controllers.

Using ConfigMaps

Like secrets, you can use ConfigMaps as environment variables or using a volume mount. They must exist prior to being used by a Pod, unless marked as *optional*. They also reside in a specific namespace.

In the case of environment variables, your pod manifest will use the **valueFrom** key and the **configMapKeyRef** value to read the values. For instance:

```
env:
- name: SPECIAL_LEVEL_KEY
  valueFrom:
    configMapKeyRef:
      name: special-config
      key: special.how
```

With volumes, you define a volume with the configMap type in your pod and mount it where it needs to be used.

```
volumes:
    - name: config-volume
      configMap:
        name: special-config
```