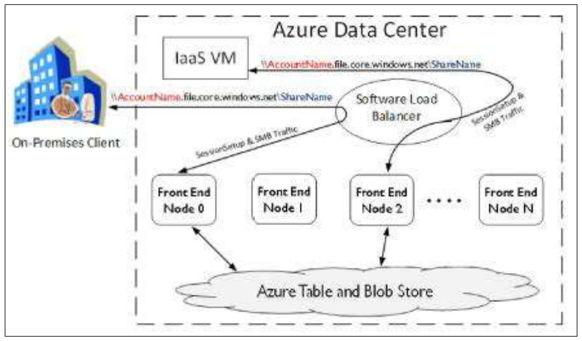
```
          secretName: azure-file-secret
         shareName: azure-share
          readOnly: false
```

Azure file storage supports sharing within the same region as well as connecting on-premise clients. Here is a diagram that illustrates the workflow:


Azure Data Center
IaaS VM
\\AccountName.file.core.windows.net\ShareName
\\AccountName.file.core.windows.net\ShareName
Software Load Balancer
On-Premises Client
SessionSetup & SMB Traffic
SessionSetup & SMB Traffic
Front End Node 0
Front End Node 1
Front End Node 2
Front End Node N
Azure Table and Blob Store

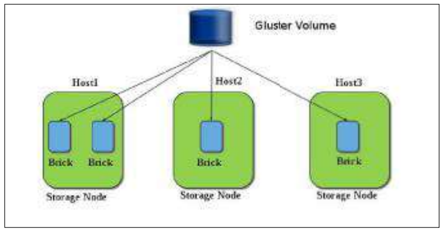
GlusterFS and Ceph volumes in Kubernetes

GlusterFS and Ceph are two distributed persistent storage systems. GlusterFS is at its core a network filesystem. Ceph is at the core an object store. Both expose block, object, and filesystem interfaces. Both use the `xfs` filesystem under the covers to store the data and metadata as `xattr` attributes. There are several reasons why you may want to use GlusterFs or Ceph as persistent volumes in your Kubernetes cluster:

- You may have a lot of data and applications that access the data in GlusterFS or Ceph
- You have administrative and operational expertise managing GlusterFS or Ceph
- You run in the cloud, but the limitations of the cloud platform persistent storage are a non-starter

Using GlusterFS

GlusterFS is intentionally simple, exposing the underlying directories as they are and leaving it to clients (or middleware) to handle high availability, replication, and distribution. Gluster organizes the data into logical volumes, which encompass multiple nodes (machines) that contain bricks, which store files. Files are allocated to bricks according to DHT (distributed hash table). If files are renamed or the GlusterFS cluster is expanded or rebalanced, files may be moved between bricks. The following diagram shows the GlusterFS building blocks:



To use a GlusterFS cluster as persistent storage for Kubernetes (assuming you have an up and running GlusterFS cluster), you need to follow several steps. In particular, the GlusterFS nodes are managed by the plugin as a Kubernetes service (although as an application developer it doesn't concern you).

Creating endpoints

Here is an example of an endpoints resource that you can create as a normal Kubernetes resource using `kubectl create`:

```
{
  "kind": "Endpoints",
  "apiVersion": "v1",
  "metadata": {
    "name": "glusterfs-cluster"
  },
  "subsets": [
    {
      "addresses": [
        {
          "ip": "10.240.106.152"
    }
   ],
   "ports": [
    {
     "port": 1
    }
   ]
  },
  {
   "addresses": [
    {
     "ip": "10.240.79.157"
    }
   ],
   "ports": [
    {
     "port": 1
    }
   ]
  }
 ]
}
```

Adding a GlusterFS Kubernetes service

To make the endpoints persistent, you use a Kubernetes service with no selector to indicate the endpoints are managed manually:

```
{
 "kind": "Service",
 "apiVersion": "v1",
 "metadata": {
  "name": "glusterfs-cluster"
 },
 "spec": {
  "ports": [
   {"port": 1}
  ]
 }
}
```

Creating pods

Finally, in the pod spec's `volumes` section, provide the following information:

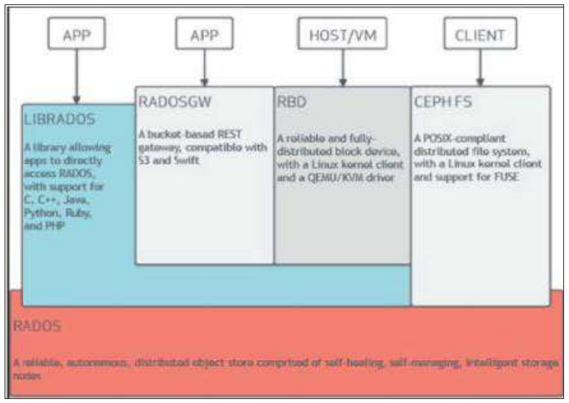
```
"volumes": [
      {
        "name": "glusterfsvol",
        "glusterfs": {
          "endpoints": "glusterfs-cluster",
          "path": "kube_vol",
          "readOnly": true
        }
      }
    ]
```

The containers can then mount `glusterfsvol` by name.

The `endpoints` tell the GlusterFS volume plugin how to find the storage nodes of the GlusterFS cluster.

Using Ceph

Ceph's object store can be accessed using multiple interfaces. Kubernetes supports the **RBD** (block) and **CEPHFS** (filesystem) interfaces. The following diagram shows how RADOS – the underlying object store – can be accessed in multiple days. Unlike GlusterFS, Ceph does a lot of work automatically. It does distribution, replication, and self-healing all on its own:


APP
APP
HOST/VM
CLIENT
LIBRADOS
A library allowing apps to directly access RADOS, with support for C, C++, Java, Python, Ruby, and PHP
RADOSGW
A bucket-based REST gateway, compatible with S3 and Swift
RBD
A reliable and fully-distributed block device, with a Linux kernel client and a QEMU/KVM driver
CEPH FS
A POSIX-compliant distributed file system, with a Linux kernel client and support for FUSE
RADOS
A reliable, autonomous, distributed object store comprised of self-healing, self-managing, intelligent storage nodes

Connecting to Ceph using RBD

Kubernetes supports Ceph via the **Rados Block Device** (**RBD**) interface. You must install ceph-common on each node in the Kubernetes cluster. Once you have your Ceph cluster up and running, you need to provide some information required by the Ceph RBD volume plugin in the `pod` configuration file:

- `monitors`: Ceph monitors.
- `pool`: The name of the RADOS pool. If not provided, the default RBD pool is used.
- `image`: The image name that RBD has created.
- `user`: The RADOS user name. If not provided, the default `admin` is used.
- `keyring`: The path to the `keyring` file. If not provided, the default `/etc/ceph/keyring` is used.
- `secretName`: The name of the authentication secrets. If provided, `secretName` overrides `keyring`. Note, see the following paragraph about how to create a `secret`.
- `fsType`: The filesystem type (`ext4`, `xfs`, and so on) that is formatted on the device.
- `readOnly`: Whether the filesystem is used as `readOnly`.

If the Ceph authentication `secret` is used, you need to create a `secret` object:

```
apiVersion: v1
kind: Secret
metadata:
  name: ceph-secret
type: "kubernetes.io/rbd"
data:
  key: QVFCMTZWMVZvRjVtRXhBQTVrQ1FzN2JCajhWVUxSdzI2Qzg0SEE9PQ==
```

[The `secret` type is `kubernetes.io/rbd`.]

The pod spec's volumes section looks same as this:

```
"volumes": [
    {
        "name": "rbdpd",
        "rbd": {
            "monitors": [
          "10.16.154.78:6789",
      "10.16.154.82:6789",
          "10.16.154.83:6789"
        ],
            "pool": "kube",
            "image": "foo",
            "user": "admin",
            "secretRef": {
      "name": "ceph-secret"
      },
            "fsType": "ext4",
            "readOnly": true
        }
    }
]
```

Ceph RBD supports `ReadWriteOnce` and `ReadOnlyMany` access modes.

Connecting to Ceph using CephFS

If your Ceph cluster is already configured with CephFS, then you can assign it very easily to pods. Also CephFS supports `ReadWriteMany` access modes.

The configuration is similar to Ceph RBD, except you don't have a pool, image, or filesystem type. The secret can be a reference to a Kubernetes `secret` object (preferred) or a `secret` file:

```
apiVersion: v1
kind: Pod
metadata:
  name: cephfs
spec:
  containers:
  - name: cephfs-rw
    image: kubernetes/pause
    volumeMounts:
    - mountPath: "/mnt/cephfs"
      name: cephfs
  volumes:
  - name: cephfs
    cephfs:
      monitors:
      - 10.16.154.78:6789
      - 10.16.154.82:6789
      - 10.16.154.83:6789
      user: admin
      secretFile: "/etc/ceph/admin.secret"
      readOnly: true
```

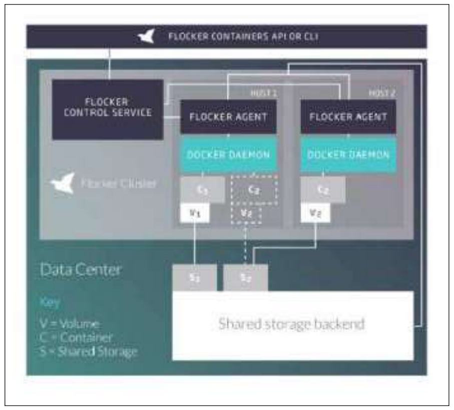
You can also provide a path as a parameter in the `cephfs` system. The default is `/`.

Flocker as a clustered container data volume manager

So far, we have discussed storage solutions that stored the data outside the Kubernetes cluster (except for `emptyDir` and HostPath, which are not persistent). Flocker is a little different. It is Docker-aware. It was designed to let Docker data volumes transfer with their container when the container is moved between nodes. You may want to use the Flocker volume plugin if you're migrating a Docker-based system that use a different orchestration platform, such as Docker compose or Mesos, to Kubernetes and you use Flocker for orchestrating storage. Personally, I feel that there is a lot of duplication between what Flocker does and what Kubernetes does to abstract storage.

Flocker has a control service and agents on each node. Its architecture is very similar to Kubernetes with its API server and the Kubelet running on each node. The Flocker control service exposes a REST API and manages the configuration of the state across the cluster. The agents are responsible for ensuring that the state of their node matches the current configuration. For example, if a dataset needs to be on node X, then the Flocker agent on node X will create it.

The following diagram showcases the Flocker architecture:



In order to use Flocker as persistent volumes in Kubernetes, you first must have a properly configured Flocker cluster. Flocker can work with many backing stores (again, very similar to Kubernetes persistent volumes).

Then you need to create Flocker datasets and at that point you're ready to hook it up as a persistent volume. After all your hard work, this part is easy and you just need to specify the Flocker dataset name:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
    - name: some-container
      image: kubernetes/pause
      volumeMounts:
          # name must match the volume name below
          - name: flocker-volume
            mountPath: "/flocker"
  volumes:
    - name: flocker-volume
      flocker:
        datasetName: some-flocker-dataset
```

Integrating enterprise storage into Kubernetes

If you have an existing **Storage Area Network** (**SAN**) exposed over the iSCSI interface, Kubernetes has a volume plugin for you. It follows the same model as other shared persistent storage plugins we've seen earlier. You must configure the iSCSI initiator, but you don't have to provide any initiator information. All you need to provide is the following:

- IP address of the iSCSI target and port (if not the default `3260`)
- Target's `iqn` (iSCSI qualified name) – typically reversed domain name
- **LUN** – logical unit number
- Filesystem type
- Readonly Boolean flag

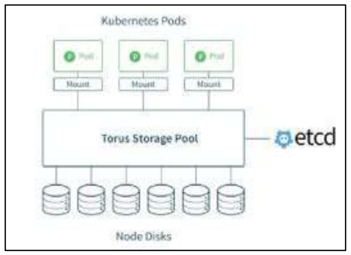
The `iSCSI` plugin supports `ReadWriteOnce` and `ReadonlyMany`. Note that you can't partition your device at this time. Here is the volume spec:

```
volumes:
  - name: iscsi-volume
    iscsi:
      targetPortal: 10.0.2.34:3260
      iqn: iqn.2001-04.com.example:storage.kube.sys1.xyz
      lun: 0
      fsType: ext4
      readOnly: true
```

Torus – the new kid on the block

CoreOS recently released Torus – a new network storage system designed for Kubernetes. It takes advantage of the Kubernetes networking model and utilizes ATA over Ethernet. It is optimized for distributing storage across a large number of commodity hardware compared to the traditional approach of a relatively small number of specialized hardware. Torus uses `etcd` to store the storage state and can be connected to Kubernetes via the Flex volume plugin. It's still early days but Torus can become the right storage solution for fresh Kubernetes deployments. It will be very interesting to follow its progress.

Here is a diagram that shows how Torus is organized and deployed:



Summary

In this chapter, we took a deep look into storage in Kubernetes. We've looked at the generic conceptual model based on volumes, claims, and storage classes, as well as the implementation as volume plugins. Kubernetes eventually maps all storage systems into mounted filesystems in containers. This straightforward model allows administrators to configure and hook up any storage system from local `host` directories through cloud-based shared storage all the way to enterprise storage systems. You should now have a clear understanding of how storage is modeled and implemented in Kubernetes and be able to make intelligent choices of how to implement storage in your Kubernetes cluster.

In the *Chapter 8, Running Stateful Application with Kubernetes*, we'll see how Kubernetes can raise the level of abstraction and on top of storage help in developing, deploying, and operating stateful applications using concepts such as stateful sets.