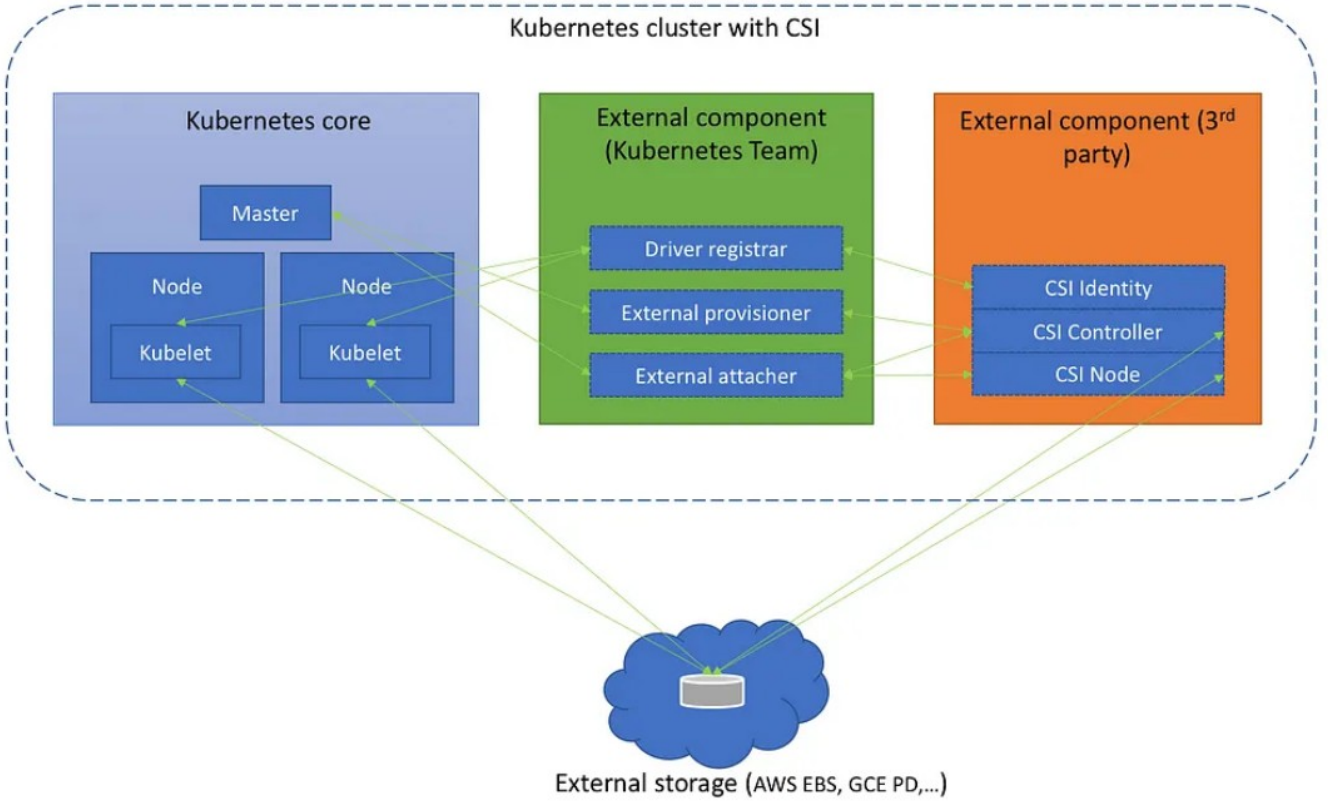
KUBERNETES

CONTAINER STORAGE INTERFACE

Container Storage Interface (CSI) is an initiative to unify the storage interface of Container Orchestrator Systems (COs) like Kubernetes, Mesos, Docker swarm, cloud foundry, etc. combined with storage vendors like Ceph, Portworx, NetApp etc. This means, implementing a single CSI for a storage vendor is guaranteed to work with all Cos.



PV - PVC

PV: 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.

PVC: A PersistentVolumeClaim 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 or ReadWriteMany, see AccessModes).

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: pvc-7c966401-e417-4a74-8ee0-dd978e49ffce
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 8Gi
  claimRef:
    apiVersion: v1
    kind: PersistentVolumeClaim
    name: datadir-mongodb-1
    namespace: mongodb
  csi:
    driver: dobs.csi.digitalocean.com
    fsType: ext4
    volumeAttributes:
      storage.kubernetes.io/csiProvisionerIdentity: 1698526980180
  persistentVolumeReclaimPolicy: Delete
  storageClassName: do-block-storage
  volumeMode: Filesystem
status:
  phase: Bound
```

POD USING STORAGE

VOLUME / VOLUME MOUNT: On-disk files in a container are **ephemeral**, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a Pod and need to share files. Kubernetes supports many types of volumes. A Pod can use any number of volume types simultaneously. Ephemeral volume types have a lifetime of a pod, but **persistent volumes** exist beyond the lifetime of a pod. When a pod ceases to exist, Kubernetes destroys ephemeral volumes; however, Kubernetes does not destroy persistent volumes. To use a volume, specify the volumes to provide for the Pod in .spec.**volumes** and declare where to mount those volumes into containers in .spec.containers[*].**volumeMounts**

Several types of volumes:

- Configmap/Secret
- Emptydir
- PersistentVolumeClaim
- HostPath / local (uncommon)
- Others (related to driver type es. NFS, cephfs, iscsi)

EMPTYDIR

An **emptyDir** volume is first created when a Pod is assigned to a node, and exists as long as that Pod is running on that node. As the name says, the emptyDir volume is initially empty. All containers in the Pod can read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the emptyDir is deleted permanently.

If you set the emptyDir.medium field to "**Memory**", Kubernetes mounts a tmpfs (**RAM-backed filesystem**) for you instead. While tmpfs is very fast be aware that, unlike disks, files you write count against the memory limit of the container that wrote them.

The storage is allocated by default from **node ephemeral storage**. If that is filled up from another source (for example, log files or image overlays), the emptyDir may run out of capacity before this limit.

```
apiVersion: v1
kind: Pod
metadata:
  name: buildah-emptydir
spec:
  containers:
    - name: buildah
      image: quay.io/buildah/stable:v1.23.1
      command: ["sleep", "infinity"]
      volumeMounts:
      - mountPath: /var/lib/containers
        name: container-storage
  volumes:
  - name: container-storage
    emptyDir:
     medium: Memory
     sizeLimit: 1Gi
```

PERSISTENT VOLUME CLAIM

A PersistentVolume (PV) represents a storage resource abstraction in a Kubernetes cluster that has a lifecycle independent of any Pod lifecycle that is using it. A Pod cannot directly refer to a PV; however, a Pod uses PersistentVolumeClaim (PVC) to request and bind to the PV, which points to the actual durable storage. A cluster administrator can configure storage provisioning and define PVs.

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 or ReadWriteMany, see AccessModes).

```
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

STORAGE CLASS

A StorageClass provides a way for administrators to describe the "classes" of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators. This concept is sometimes called "profiles" in other storage systems.

```
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
  name: do-block-storage
provisioner: dobs.csi.digitalocean.com
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

Dynamic volume provisioning allows storage volumes to be created on-demand. Without dynamic provisioning, cluster administrators have to manually make calls to their cloud or storage provider to create new storage volumes, and then create PersistentVolume objects to represent them in Kubernetes. The dynamic provisioning feature eliminates the need for cluster administrators to pre-provision storage.

PersistentVolumes that are dynamically created by a StorageClass will have the reclaim policy specified in the reclaimPolicy field of the class, which can be either Delete or Retain. If no **reclaimPolicy** is specified when a StorageClass object is created, it will default to Delete. PersistentVolumes can be configured to be expandable. When **allowVolumeExpansion** is set to true, it allows the users to resize the volume by editing the corresponding PVC object.

CONFIGMAP / SECRET

Many applications rely on configuration which is used during either application initialization or runtime. Most times, there is a requirement to adjust values assigned to configuration parameters. ConfigMaps are a Kubernetes mechanism that let you inject configuration data into application pods.

The **ConfigMap** concept allow you to decouple configuration artifacts from image content to keep containerized applications portable.

There are multiple way to mount configmap or secrets:

- as a env var

- as a file

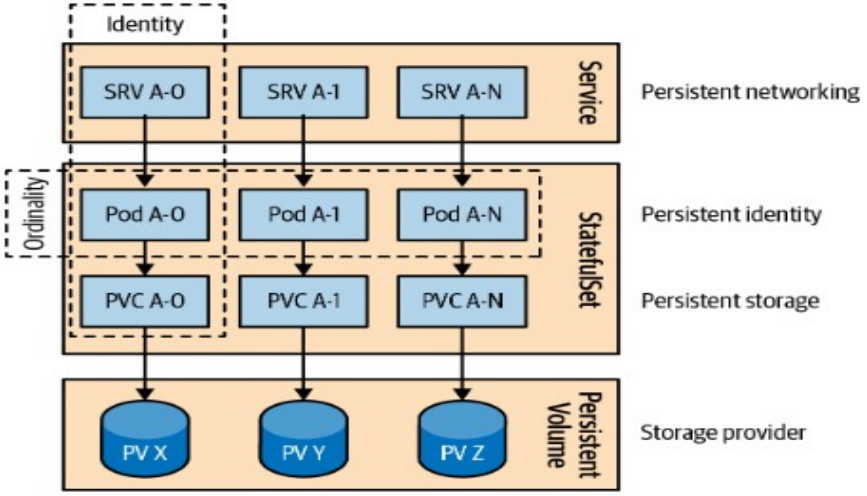
- configmap can be mounted using subpath key to avoid destroying the folder content

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-config
  namespace: default
data:
  allowed: '"true"'
  enemies: aliens
  lives: "3"
---
apiVersion: v1
kind: Pod
metadata:
  name: subpathtest-pod
spec:
  containers:
    - name: test-container
      image: registry.k8s.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: game-config
              key: enemies
  restartPolicy: Never
```

STATEFUL APPLICATIONS

Distributed stateful applications require features such as persistent identity, networking, storage, and ordinality.

While it is not always necessary, the majority of stateful applications store state and thus require per-instance-based dedicated persistent storage. The way to request and associate persistent storage with a Pod in Kubernetes is through PVs and PVCs. To create PVCs the same way it creates Pods, StatefulSet uses a volumeClaimTemplates element.




```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: rg
spec:
  serviceName: random-generator
  replicas: 2
  selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
      - image: k8spatterns/random-
generator:1.0
        name: random-generator
        ports:
        - containerPort: 8080
          name: http
        volumeMounts:
          - name: logs
            mountPath: /logs
volumeClaimTemplates:
- metadata:
    name: logs
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 10Mi
```

TRAVAIL PRATIQUE

- Mongodb deploy
- Mongoapp deploy
- Mongodb backup
- https://simodev.medium.com/how-to-automate-backup-mongodb-using-kubernetes-b07c61a8a6ec