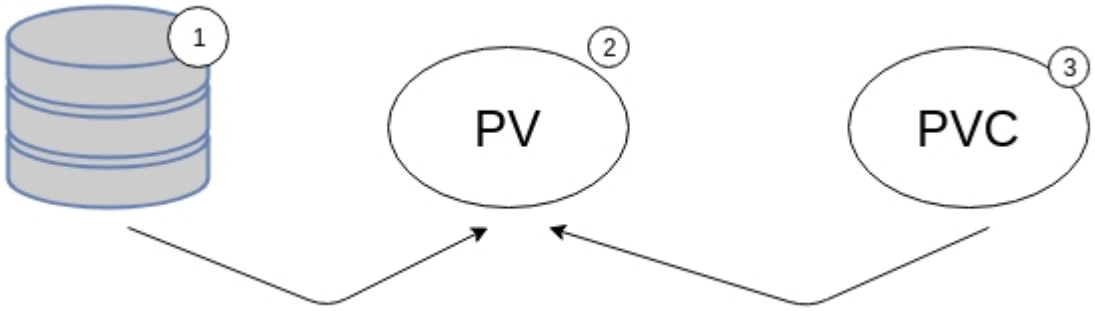
CKA Curriculum Part 6 - Storage

Understand persistent volumes and know how to create them

Pods, and by extension containers require storage. Without explicitly defining any storage parameters in manifest files, local storage from facilitating node is used, and when a pod has reached the end of its lifecycle, it is removed, along with its filesystem. However, as environments/applications grow in complexity, storage requirements require more attention with regards to sizing, provisioning and eventual deletion.

Two key ingredients are required to leverage the inherent storage capabilities within Kubernetes, PersistentVolume and PersistentVolumeClaim. Top level process is illustrated below:



1 - Create storage

This is highly dependent on the infrastructure the Kubernetes environment is deployed into. But as some examples:

Azure - Azure File, Azure DIsk

Amazon - awsElasticBlockStore

GCP - GCEPersistentDIsk

Cloud Agnostic - NFS, iSCSI, StorageOS

In this example, a NFS server has been configured on a VM with IP address 172.16.10.223

```
cat /etc/exports
/mnt/nfs *(rw,sync,no_root_squash)
```

2 - Persistent Volume

A PersistentVolume is a storage endpoint that has been provisioned into the cluster. The exact manifestation of this storage is dependent on the underlying infrastructure. However for a NFS server the below is a example of a PV

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs
spec:
  capacity:
    storage: 80Gi
  accessModes:
    - ReadWriteMany
  nfs:
    path: /mnt/nfs
    server: 172.16.10.223
```

The options available are also dependent on the underlying storage. For example, AzureDIsk only supports ReadWriteOnce, NFS supports all three.

Check it is available:

```
kubectl get persistentvolume
NAME     CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
pv-nfs   80Gi       RWX            Recycle          Available
```

3 - Persistent Volume Claim

Once a Persistent Volume has been created we can start to carve out access to the PV via Persistent Volume Claims. These are intended to be used directly by the applications, and provides a layer of abstraction between the application, and the volume itself. An example is shown below:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: nginx-nfs-claim
spec:
 accessModes:
   - ReadWriteMany
 resources:
   requests:
     storage: 1Gi
```

Putting this all together in a pod spec looks like this:

```
apiVersion: v1
kind: Pod
metadata:
 name: nginx-web
 labels:
   role: web
spec:
 containers:
 - name: nginx
   image: nginx
   volumeMounts:
     - name: pv-nfs
       mountPath: "/mnt/nfs"
 volumes:
 - name: pv-nfs
   persistentVolumeClaim:
     claimName: nginx-nfs-claim
```

If we log in directly to a pod:

```
kubectl exec -it nginx-web bash
root@nginx-web:/# df
Filesystem             1K-blocks    Used Available Use% Mounted on
overlay                 20508240 6727200  12716236  35% /
tmpfs                      65536       0     65536   0% /dev
tmpfs                    1020792       0   1020792   0% /sys/fs/cgroup
/dev/sda2               20508240 6727200  12716236  35% /etc/hosts
172.16.10.223:/mnt/nfs 102687744   61440  97367040   1% /mnt/nfs
shm                        65536       0     65536   0% /dev/shm
tmpfs                    1020792      12   1020780   1% /run/secrets/kubernetes.io/serviceaccount
tmpfs                    1020792       0   1020792   0% /proc/acpi
tmpfs                    1020792       0   1020792   0% /proc/scsi
tmpfs                    1020792       0   1020792   0% /sys/firmware
```

Understand access modes for volumes

We currently have three options:

- ReadWriteOnce – the volume can be mounted as read-write by a single node
- ReadOnlyMany – the volume can be mounted read-only by many nodes
- ReadWriteMany – the volume can be mounted as read-write by many nodes

Understand persistent volume claims primitives

Each PVC contains a spec and status, which is the specification and status of the claim.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 8Gi
  storageClassName: slow
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: environment, operator: In, values: [dev]}
```

Access Modes

Claims use the same conventions as volumes when requesting storage with specific access modes.

Volume Modes

Claims use the same convention as volumes to indicate the consumption of the volume as either a filesystem or block device.

Resources

Claims, like pods, can request specific quantities of a resource. In this case, the request is for storage. The same resource model applies to both volumes and claims.

Selector

Claims can specify a label selector to further filter the set of volumes. Only the volumes whose labels match the selector can be bound to the claim. The selector can consist of two fields:

- matchLabels - the volume must have a label with this value
- matchExpressions - a list of requirements made by specifying key, list of values, and operator that relates the key and values. Valid operators include In, NotIn, Exists, and DoesNotExist.

All of the requirements, from both matchLabels and matchExpressions are ANDed together – they must all be satisfied in order to match.

Class

A claim can request a particular class by specifying the name of a StorageClass using the attribute storageClassName. Only PVs of the requested class, ones with the same storageClassName as the PVC, can be bound to the PVC.

PVCs don't necessarily have to request a class. A PVC with its storageClassName set equal to "" is always interpreted to be requesting a PV with no class, so it can only be bound to PVs with no class (no annotation or one set equal to ""). A PVC with no storageClassName is not quite the same and is treated differently by the cluster depending on whether the DefaultStorageClass admission plugin is turned on.

- If the admission plugin is turned on, the administrator may specify a default StorageClass. All PVCs that have no storageClassName can be bound only to PVs of that default. Specifying a default StorageClass is done by setting the annotation storageclass.kubernetes.io/is-default-class equal to "true" in a StorageClass object. If the

administrator does not specify a default, the cluster responds to PVC creation as if the admission plugin were turned off. If more than one default is specified, the admission plugin forbids the creation of all PVCs.
- If the admission plugin is turned off, there is no notion of a default StorageClass. All PVCs that have no storageClassName can be bound only to PVs that have no class. In this case, the PVCs that have no storageClassName are treated the same way as PVCs that have their storageClassName set to "".

Depending on installation method, a default StorageClass may be deployed to Kubernetes cluster by addon manager during installation.

When a PVC specifies a selector in addition to requesting a StorageClass, the requirements are ANDed together: only a PV of the requested class and with the requested labels may be bound to the PVC.

Note: Currently, a PVC with a non-empty selector can't have a PV dynamically provisioned for it.

In the past, the annotation volume.beta.kubernetes.io/storage-class was used instead of storageClassName attribute. This annotation is still working, however it won't be supported in a future Kubernetes release

Understand Kubernetes Storage Objects

K8S offer a plethora of storage object types, which vary depending on the infrastructure or cloud provider that's being leveraged : https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes

Know how to configure applications to use persistent storage

Already demonstrated in earlier exercises. At a top level:

- Provision storage
- Create PV
- Create PV Claim
- Configure pod/deployment manifest to use PV claim

Note

- PV <-> PVC is 1:1
- Pod <-> PVC can be many : many
    - Multiple pods can use the same PVC
    - Pods can leverage multple PVC's