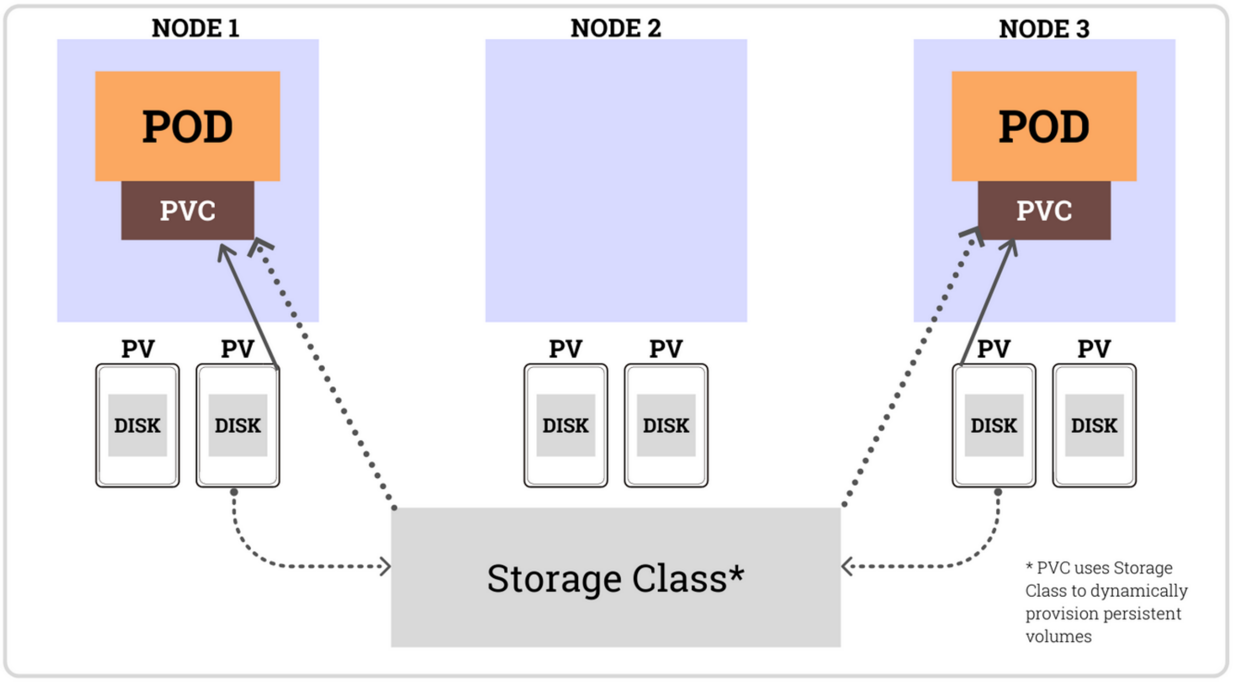
Kubernetes storage basics: PV, PVC and StorageClass



It’s been a while since Kubernetes and its REST API-based orchestration model took the madness surrounding distributed systems out and made the world a happier place (for you and me). Everything you need is a couple of YAMLs and kubectls away. If it’s on your application stack, Kubernetes has a ‘resource’ for it and a controller to make sure it’s resourceful.

Let’s talk about storage, the persistent kind!

Persistent Volumes

Kubernetes makes physical storage devices like your SSDs, NVMe disks, NAS, NFS servers available to your cluster in the form of objects called -- **Persistent Volumes**.

If you’re using Kubernetes on Google’s or Amazon’s cloud, you can have your google SSDs or EBS volumes available to your containers in the form of persistent volumes. If you’re like me and you couldn’t let your bare metal servers go, then look no further than OpenEBS. Kubernetes natively does not ‘see’ your disks. You will need something like OpenEBS to pick it up.

Each of these Persistent Volumes is consumed by a Kubernetes Pod (or Pods) by issuing a **PersistentVolumeClaim** object -- a PVC. A PVC object lets pods use storage from Persistent Volumes.

A Persistent Volume is an abstraction for the physical storage device that you have attached to the cluster. Your pods can use this storage space using Persistent Volume Claims. The easiest way to create the PV/PVC pair for your Pod is to use a ***StorageClass*** object, and then using the storageclass to create your PV-PVC pair dynamically whenever you need to use it.

StorageClass

A storageclass is a Kubernetes object that stores information about creating a persistent volume for your pod. With a storageclass, you do not need to create a persistent volume separately before claiming it.

Let’s take an example out of docs.openebs.io/docs/next/uglocalpv-device.html.
(Use the command kubectl apply -f https://openebs.github.io/charts/openebs-operator.yaml if you want to install OpenEBS and use the storage class below)

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-device
  annotations:
    openebs.io/cas-type: local
    cas.openebs.io/config: |
      - name: StorageType
        value: device
      - name: FSType
        value: xfs
provisioner: openebs.io/local
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

When we use this storage class to create a PVC, we tell the kubernetes apiserver that we want to use OpenEBS's 'openebs.io/local' provisioner plugin to create our Persistent Volume and claim it.

We are also using the 'WaitForFirstConsumer' binding mode to make sure we pick a block device on the node where the application pod has been scheduled.

We have also specified some additional parameters to the OpenEBS provisioner to use XFS filesystem and to use one of the block devices available to the cluster nodes to create the PV.

Your application pods can repeatedly use the same storage class to provision persistent volume claims as long as you still have unused block devices available on that specific node.

Persistent Volume Claim

A persistent volume claim is a dedicated storage that kubernetes has carved out for your application pod, from the storage that was made available using the storage class.

Let's create a PVC yaml using the above storage class…

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: local-device-pvc
spec:
  storageClassName: local-device
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

In this case, we are telling kubernetes to use the storage class 'local-device' to create a Persistent Volume with 5Gi of storage capacity and RWO access mode.

We can create this resource and check for the following outputs:

```
kubectl get pv
```

```
NAME
pvc-079bbc07-e2fb-412a-837b-4745051c1bfc
```

and

```
kubectl get pvc
```

```
NAME               STATUS   VOLUME
local-device-pvc   Bound    pvc-079bbc07-e2fb-412a-837b-4745051c1bfc
```

We can create our application deployment using this PVC.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          Value: "password"
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: local-device-pvc
```

So, what now?

Following these operations, the Kubelet will mount a volume that matches the specifications of the PVC to the application container. We can check if this mount has been initiated by logging into the node where the application pod has been scheduled and executing the mount command.

CSI

Before CSI (Container Storage Interface) came along in Kubernetes v1.13, developers have depended on Kubernetes' in-tree plugin model to implement storage solutions. With CSI, developers can now write and maintain their own plugins and take control of the cluster's storage components. The CSI standard exposes hardware components that developers can use to have better control over the storage stack and push changes faster and test their code better.

Check out OpenEBS's CSI-flavoured storage solutions at:
https://github.com/openebs/Mayastor
https://github.com/openebs/zfs-localpv
https://github.com/openebs/cstor-operators

Conclusion

Persistent storage solutions in Kubernetes, especially with the introduction of CSI, have made production-grade containerized stateful workloads a reality. Also, with Kubernetes' operator framework -- block-level synchronous replication, plugin-based backup-and-restore, data-migration have been added on to the toolchain.