Local Persistent Volumes – A Step-by-Step Tutorial

Kubernetes local volumes go beta. However, what is it, a Kubernetes local volume? Last time, we have discovered, how to use Kubernetes hostPath volumes. However, we also have seen, that hostPath volumes work well only on single node clusters. Here, Kubernetes local volumes help us to overcome the restriction and we can work in a multi-node environment with no problems.

„Local volumes“ are similar to hostPath volumes, but they allow to pin-point PODs to a specific node, and thus making sure that a restarting POD always will find the data storage in the state it had left it before the reboot. They also make sure that other restrictions are met before the used persistent volume claim is bound to a volume.

Note, the disclaimer on the announcement that local volumes are not suitable for most applications. They are much easier to handle than clustered file systems like glusterfs, though. Still, local volumes are perfect for clustered applications like Cassandra.

References

- Kubernetes Documentation on persistent Volumes
- Katacoda persistent Volumes Hello World with an NFS Docker container

Prerequisites

- We need a multi-node Kubernetes Cluster to test all of the features of „local volumes“. A two-node cluster with 2 GB or better 4 GB RAM each will do.

Step 1: Create StorageClass with WaitForFirstConsumer Binding Mode

According to the docs, persistent local volumes require to have a binding mode of WaitForFirstConsumer. the only way to assign the volumeBindingMode to a persistent volume seems to be to create a storageClass with the respective volumeBindingMode and to assign the storageClass to the persistent volume. Let us start with

```
cat > storageClass.yaml << EOF
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: my-local-storage
provisioner: kubernetes.io/no-provisioner
```

```
volumeBindingMode: WaitForFirstConsumer
EOF

kubectl create -f storageClass.yaml
```

The output should be:

```
storageclass.storage.k8s.io/my-local-storage created
```

Step 2: Create Local Persistent Volume

Since the storage class is available now, we can create local persistent volume with a reference to the storage class we have just created:

```
cat > persistentVolume.yaml << EOF
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-local-pv
spec:
  capacity:
    storage: 500Gi
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: my-local-storage
  local:
    path: /mnt/disk/vol1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - node1
EOF
```

Note: You might need to exchange the hostname value „node1“ in the nodeAffinity section by the name of the node that matches your environment.

The „hostPath“ we had defined in our post is replaced by the so-called „local path„.

Similar to what we have done in case of a hostPath volume in our post, we need to prepare the volume on node1, before we create the persistent local volume on the master:

```
# on the node, where the POD will be located (node1 in our case):
DIRNAME="vol1"
mkdir -p /mnt/disk/$DIRNAME
chcon -Rt svirt_sandbox_file_t /mnt/disk/$DIRNAME
chmod 777 /mnt/disk/$DIRNAME

# on master:
kubectl create -f persistentVolume.yaml
```

The output should look like follows:

```
persistentvolume/my-local-pv created
```

Step 3: Create a Persistent Volume Claim

Similar to hostPath volumes, we now create a persistent volume claim that describes the volume requirements. One of the requirement is that the persistent volume has the volumeBindingMode: WaitForFirstConsumer. We can assure this by referencing the previously created a storageClass:

```
cat > persistentVolumeClaim.yaml << EOF
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-claim
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: my-local-storage
  resources:
    requests:
      storage: 500Gi
EOF

kubectl create -f persistentVolumeClaim.yaml
```

With the answer:

```
persistentvolumeclaim/my-claim created
```

From point of view of the persistent volume claim, this is the only difference between a local volume and a host volume. However, different to our observations about host volumes in the post, the persistent volume claim is not bound to the persistent volume automatically. Instead, it will remain „Available“ until the first consumer shows up:

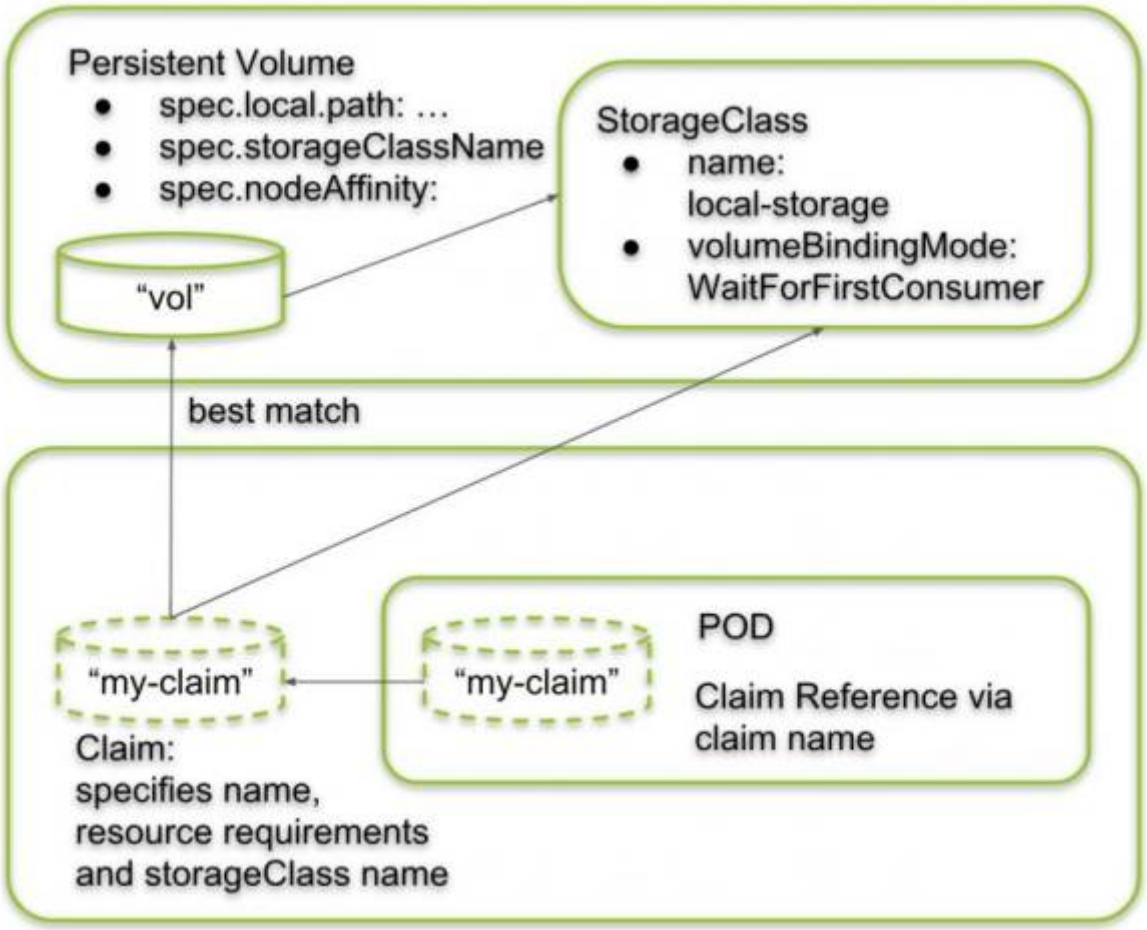
```
# kubectl get pv
NAME          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM
STORAGECLASS       REASON   AGE
my-local-pv   500Gi      RWO            Retain           Available
```

```
my-local-storage              3m59s
```

This should change in the next step.

Step 4: Create a POD with local persistent Volume

The Kubernetes Architects have done a good job in abstracting away the volume technology from the POD. As with other volume technologies, the POD just needs to reference the volume claim. The volume claim, in turn, specifies its resource requirements. One of those is the volumeBindingMode to be WairForFirstCustomer. This is achieved by referencing a storageClass with this property:



Once a POD is created that references the volume claim by name, a „best match“ choice is performed under the restriction that the storage class name matches as well.

Okay, let us perform the last required step to complete the described picture. The only missing piece is the POD, which we will create now:

```
cat > http-pod.yaml << EOF
apiVersion: v1
kind: Pod
metadata:
```

```
  name: www
  labels:
    name: www
spec:
  containers:
  - name: www
    image: nginx:alpine
    ports:
      - containerPort: 80
        name: www
    volumeMounts:
      - name: www-persistent-storage
        mountPath: /usr/share/nginx/html
  volumes:
    - name: www-persistent-storage
      persistentVolumeClaim:
        claimName: my-claim
EOF

kubectl create -f http-pod.yaml
```

This should yield:

```
pod/www created
```

Before, we have seen that the persistent volume claim was not bound to a persistent volume yet. Now, we expect the binding to happen, since the last missing piece of the puzzle has fallen in place already:

```
# kubectl get pv
NAME          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS       REASON   AGE
my-local-pv   500Gi      RWO            Retain           Bound
default/my-claim     my-local-storage            10m
```

Yes, we can see that the status is bound to claim named „default/my-claim“. Since we have not chosen any namespace, the claim is located in the „default“ namespace.

The POD is up and running:

```
# kubectl get pods
NAME     READY   STATUS    RESTARTS   AGE
www      1/1     Running   0          3m29s
```

Summary

In this post, we have shown that Kubernetes local volumes can be run on multi-node clusters without the need to pin PODs to certain nodes explicitly. Local volumes with their node affinity rules make sure that a POD is bound to a certain node implicitly, though. Kubernetes local volumes have

following features:

- Persistent volume claims will wait for a POD to show up before a local persistent volume is bound
- Once a persistent local volume is bound to a claim, it remains bound, even if the requesting POD has died or has been deleted
- A new POD can attach to the existing data in a local volume by referencing the same persistent volume claim
- Similar to NFS shares, Kubernetes persistent local volumes allow multiple PODs to have read/write access

Kubernetes local persistent volume they work well in clustered Kubernetes environments without the need to explicitly bind a POD to a certain node. However, the POD is bound to the node implicitly by referencing a persistent volume claim that is pointing to the local persistent volume. Once a node has died, the data of all local volumes of that node are lost. In that sense, Kubernetes local persistent volume cannot compete with distributed solutions like Glusterfs and Portworx volumes.