Manifests are on classfiles/os-storage.

1. Simple PVC-PV workflow in OpenShift

As admin, create a Persistent Volume

Open a bash session on cluster server and create directory with a web site in it.

```
$ ssh 192.168.130.11
CRC$ mkdir data
CRC$ echo '<h1>This is OpenShift</h1>' >data/index.html
CRC$ exit
```

Create a manifest for a persistent volume named `pv.yaml`.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: myvol
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/home/core/data"
```

The orange text is modified from the original on the Kubernetes site.

Analyze a few details of this manifest.

- `ReadWriteOnce` means that the volume can be mounted read/write, but only by one container at a time.
- The volume is of type `hostPath`, which means that it corresponds to a directory on the pod's host.
  Be aware that this is **not suitable for production**.

Log on as **developer** and try to create the volume.

```
$ oc login -u developer
$ oc apply -f pv.yaml
Error from server (Forbidden): error when retrieving current configuration of:
Resource: "/v1, Resource=persistentvolumes", GroupVersionKind: "/v1, Kind=PersistentVolume"
Name: "myvol", Namespace: ""
from server for: "pv.yaml": persistentvolumes "myvol" is forbidden: User "developer" cannot get resource
"persistentvolumes" in API group "" at the cluster scope
```

You need to have a cluster admin role to create a persistent volume.

```
$ oc login -u kubeadmin
$ oc apply -f pv.yaml
```

Have a closer look at your volumes

```
$ oc get pv
```

Quite a few PVs are available. They were created during cluster installation.

```
$ oc get pv myvol -o yaml
$ oc describe pv myvol
```

A few details are interesting. The **persistentVolumeReclaimPolicy** specifies what happens with a PV it is released from its PVC. *myvol* has a policy of *Retain*, which means the PV and the data on it will be kept until the administrator releases it manually. Other possible

values are *Delete* (delete the PV) and *Recycle* (remove the data but keep the PV). *Recycle* is deprecated; instead, dynamic provisioning using Storage Classes should be used.

The **volumeMode** indicates that the volume is supposed to be used with a filesystem, not as a raw block device. The volume mode is also specified when making a persistent volume claim.

The phase is available, which indicates that the volume is currently not claimed. During the volume's life, it may be bound (i.e., claimed), then released. A PV can also be in the failed phase, which means that something went wrong when claiming it.

As a non-admin user, create a Persistent Volume Claim

```
$ oc login -u developer
```

Create pvc.yaml with this code:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
EOF
```

You can also download the PVC manifest from the Kubernetes git repo. Changes are indicated in orange.

With this manifest, you make a claim for 1GB, read-write, attachable to a single container. Note the details that you do not provide: Nothing about location, identity, name etc. of the volume. All you request is the size and a mode of read-write-once.

```
$ oc apply -f pvc.yaml
$ oc get pvc
$ oc get pvc mypvc -o yaml
$ oc describe pvc mypvc
Name:          mypvc
Namespace:     devproject
StorageClass:  manual
Status:        Bound
Volume:        myvol
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed: yes
               pv.kubernetes.io/bound-by-controller: yes
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      1Gi
Access Modes:  RWO
VolumeMode:    Filesystem
Used By:       <none>
Events:        <none>
```

This PVC is bound to volume `myvol`. The capacity is 1Gi, although `myvol` is larger than that. The volume is *bound*, but the claim is currently *not used* by a pod.

Check the volume's status. It should be bound as well.

```
$ oc login -u kubeadmin
$ oc get pv myvol
```

Mount the volume in a pod

```
$ oc login -u developer
```

Use `pod.yaml` with this content:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  volumes:
    - name: podvol
      persistentVolumeClaim:
        claimName: mypvc
  containers:
    - name: mycontainer
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: podvol
```

This pod has a list of volumes and a list of containers (the lists have one element each). The volume's name, `podvol`, is used in the container's mount.

```
$ oc apply -f pod.yaml
$ oc get pod mypod -o yaml
```

2. Add NFS

You will use the dynamic NFS client provisioner https://github.com/kubernetes-sigs/nfs-subdir-external-provisioner.

You need:

1. An NFS server
2. A service account with cluster privileges related to storage. The provisioner app will run with this account as its identity.
3. A StorageClass
4. The NFS client provisioner quay.io/external_storage/nfs-client-provisioner:latest

1. Launch and configure an NFS server

On the Desktop, launch the NFS server.

```
$ sudo systemctl enable nfs-server --now
```

Create a directory and export it.

```
$ sudo mkdir -p /srv/volumes
$ sudo chmod 777 /srv/volumes
$ echo "/srv/volumes    *(rw,sync)" >> /etc/exports
$ sudo exportfs -va
```

To test this, you will log on to the OpenShift server, mount the NFS filesystem and copy a file to the NFS share.
To mount the NFS filesystem, you need the hostname or IP address of your desktop machine.

```
$ hostname                  # you will use the hostname in the mount command below
$ ssh 192.168.130.11
[core@crc]$ sudo mkdir /tmp/testmount
[core@crc]$ sudo mount DESKTOP_HOSTNAME:/srv/volumes /tmp/testmount
[core@crc]$ cp /etc/passwd /tmp/testmount
[core@crc]$ sudo umount /tmp/testmount
[core@crc]$ exit
$ cat /srv/volumes/passwd
```

2. Clone the provisioner Git repo

Clone the provisioner from Github. The manifests required for deploying it are in the `deploy` subdirectory.

```
$ git clone https://github.com/kubernetes-sigs/nfs-subdir-external-provisioner.git
$ cd nfs-subdir-external-provisioner
$ ls deploy
class.yaml  deployment.yaml  objects  rbac.yaml  test-claim.yaml  test-pod.yaml
```

3. Create a service account and give it appropriate roles

Analyze `deploy/rbac.yaml`. It contains several manifests separated by dashes:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nfs-client-provisioner
  # replace with namespace where provisioner is deployed
  namespace: default
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
...
```

This manifest creates five resources: A service account, two roles, and the bindings of the roles to the account.
One role is a **cluster role**; it contains rules that allow the service account to manage global objects. The other is a **local role** that only affects objects in the service account's project. For this reason, the service account, the local role and the local role binding include the namespace (i.e.) in their specification.

It creates a service account, a cluster role, a binding between the service account and the cluster role, a local role for your namespace, and a local binding between the service account and the local role.

Analyze `deploy/deployment.yaml`. As the name indicates, this is a Kubernetes deployment. It consists of a single replica and has the pod replacement strategy of `Recreate`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nfs-client-provisioner
  labels:
    app: nfs-client-provisioner
  namespace: default
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: nfs-client-provisioner
```

By default, deployments use `RollingUpdate`. The `Recreate` strategy is useful if an application can't tolerate that two of its versions are up at the same time.

The pod has a single container that mounts an NFS volume. The container image comes from Google's Kubernetes registry: `k8s.gcr.io/sig-storage/nfs-subdir-external-provisioner:v4.0.2`.

Log on as a cluster administrator and double-check your namespace.

```
$ oc login -u kubeadmin
$ NAMESPACE=$(oc project -q)
$ echo $NAMESPACE
```

Your namespace should be `default`. If not, change the namespace in the two YAML files, either with a text editor or the `sed` command below:

```
$ sed -i.bak "s/namespace:.*/namespace: $NAMESPACE/g" ./deploy/rbac.yaml ./deploy/deployment.yaml
```

Apply the RBAC manifest.

```
$ oc create -f deploy/rbac.yaml
```

```
serviceaccount/nfs-client-provisioner created
clusterrole.rbac.authorization.k8s.io/nfs-client-provisioner-runner created
clusterrolebinding.rbac.authorization.k8s.io/run-nfs-client-provisioner created
role.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner created
rolebinding.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner created
```

In addition to RBAC roles, the provisioner needs to have a security context constraint in order to mount filesystems.

```
$ oc adm policy add-scc-to-user hostmount-anyuid system:serviceaccount:$NAMESPACE:nfs-client-provisioner
```

4. Create an NFS StorageClass

The storage class is nfs-storageclass.yaml and looks like this:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: dynnfs
provisioner: k8s-sigs.io/nfs-subdir-external-provisioner
parameters:
  pathPattern: "${.PVC.namespace}/${.PVC.annotations.nfs.io/storage-path}"
```

The name is used in a PVC to request storage. The provisioner is implemented by the deployment launched in the next step.

The parameter set in this example are the name of the NFS path that will be used for dynamically provisioned volumes.

5. Launch the NFS provider deployment

Use `nfs-deployment.yaml` in the os-storage subdirectory. Replace <NFSSERVER> with the hostname of your Desktop. Apply the deployment manifest.

To test success, create a volume claim that requests this storage class. Use the manifest from the beginning of the exercise description.