## Overview

In this lab, we will deploy a webserver and store our html files on a PersistentVolume.

<https://kubernetes.io/docs/concepts/storage/persistent-volumes>

* Create backing storage system
* Perform static provisioning via an NFS server
* Use PVCs and StorageClasses for dynamic provisioning
* Use a Secret to configure authentication for nginx

[Overview](#_brow32mvt65p)

[Instructions](#_h7xnyttuehza)

[Task 0: Setup](#_q1eipr6h6290)

[Step 1: Make new directory](#_30rfxpp431je)

[Task 1: Static Provisioning](#_zb96oq1l8t7a)

[Step 1: Run an NFS server](#_z9ad9pzgs0f1)

[Step 2: Create a PV object](#_c1ma2uvgmmuh)

[Step 3: Claim your PersistentVolume](#_6mpenovnyswm)

[Step 4: Mount the storage in a Pod](#_4yavdch7id2i)

[Step 5: Test Persistence](#_pedbv644iqbw)

[Task 2: Dynamic Provisioning](#_i2e82b4aksib)

[Step 0: Examine StorageClass](#_efzf5n84ceea)

[Step 1: Create PVC object](#_1dleg6ih1q42)

[Step 2: Witness PV creation](#_aco882v3ou5c)

[Task 3: Cleanup](#_lgkhas3fmr2g)

[Task 4: Bonus: Configure basic auth for nginx](#_quqkcqsik55o)

[Step 1. Create ConfigMap](#_pbnke94e7yw8)

[Step 2. Create Secret](#_7acu7o7sui1o)

[Step 3. Create basic\_auth\_pod.yaml to mount the CM and Secret](#_85mmcgcj7s4g)

[Step 4. Test that the app is only accessible when providing credentials](#_kujl5nldji6c)

[Step 5: Cleanup](#_kdg48i472gxd)

## 

## Instructions

Read this lab like a book, all text is there for a reason!

"→" denotes an action you must take

<> denotes a placeholder you should replace with a value

Ex: <IP> becomes 127.0.0.1

Use your favorite editor to edit files within the console. I suggest VI, nano, or emacs.

|  |
| --- |
| White boxes with black text denote commands and file contents |

|  |
| --- |
| Black boxes with green text denote example output |

### **Task 0: Setup**

#### **Step 1: Make new directory**

→ Create a new directory to work in

|  |
| --- |
| mkdir ~/persistence  cd ~/persistence |

### **Task 1: Static Provisioning**

#### **Step 1: Run an NFS server**

→ Run nfs server locally on your lab VM

|  |
| --- |
| docker run -d --net=host --name nfs --privileged -v /fileshare:/nfsshare -e SHARED\_DIRECTORY=/nfsshare itsthenetwork/nfs-server-alpine:latest |

|  |
| --- |
| 7821e236a6749c212aacb37b54eaff7000a22efef6c6b63382c487dc3623c81d |

Now we have a running NFS server that can serve up some storage space. In the above command, we mapped our VMs local /fileshare directory to /nfsshare in the container, which will be mountable over the network.

#### **Step 2: Create a PV object**

We now need to tell Kubernetes that we have an NFS share that can be used. We do that by putting the storage information in a PersistentVolume object. The nfs server is running in our lab VM, so we will configure that as our server. PVs objects do not live in a namespace and therefore, we must make sure we do not name our PV the same as another students.

→ Create a file named **PersistentVolume.yaml**

|  |
| --- |
| vi PersistentVolume.yaml |

→ Type ":set paste", press **enter**. Then press **i**, paste the following:

|  |
| --- |
| apiVersion: v1  kind: PersistentVolume  metadata:  name: **<STUDENTID>**-nfs-pv # Ex: student1-nfs-pv  labels:  name: **<STUDENTID>**-nfs-pv # Ex: student1-nfs-pv  spec:  capacity:  storage: 3Gi  accessModes:  - ReadWriteOnce  - ReadWriteMany  persistentVolumeReclaimPolicy: Recycle  nfs:  server: **<VM IP>**  path: / |

→ Replace the **bolded** fields

→ Hit **ESC**, then type ':wq' and press **enter** to save the file.

→ Create Deployment

|  |
| --- |
| kubectl apply -f PersistentVolume.yaml |

|  |
| --- |
| persistentvolume/student1-nfs-pv created |

#### **Step 3: Claim your PersistentVolume**

→ Create a file named **PersistentVolumeClaim.yaml**

|  |
| --- |
| vi PersistentVolumeClaim.yaml |

→ Type ":set paste", press **enter**. Then press **i**, paste the following:

|  |
| --- |
| apiVersion: v1  kind: PersistentVolumeClaim  metadata:  name: static-files  spec:  storageClassName: ""  accessModes:  - ReadWriteMany  resources:  requests:  storage: 1Gi  selector: # Make sure we claim our exact PV and not another students  matchLabels:  name: "**<STUDENTID>**-nfs-pv" |

→ Hit **ESC**, then type ':wq' and press **enter** to save the file.

→ Create PersistentVolumeClaim

|  |
| --- |
| kubectl apply -f PersistentVolumeClaim.yaml |

|  |
| --- |
| persistentvolumeclaim/static-files created |

→ Ensure your PVC was bound to our PV

|  |
| --- |
| kubectl get pvc |

|  |
| --- |
| NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE  static-files Bound student1-nfs-pv 3Gi RWO,RWX 6s |

#### **Step 4: Mount the storage in a Pod**

→ create **pod.yaml,** paste the following:

|  |
| --- |
| kind: Pod  apiVersion: v1  metadata:  name: web-server-pod  spec:  containers:  - name: frontend  image: nginx  volumeMounts:  - mountPath: "/usr/share/nginx/html"  name: static-files  volumes:  - name: static-files  persistentVolumeClaim:  claimName: static-files |

→ apply it

|  |
| --- |
| kubectl apply -f pod.yaml |

→ Ensure it is running (it may take a minute for the volume to mount)

|  |
| --- |
| kubectl get pods -o wide |

#### **Step 5: Test Persistence**

→ Get the landing page for our webserver, use the pod ip from the previous command

|  |
| --- |
| curl <POD\_IP> |

You should get a 403 forbidden response. This is because our nginx server has no files to serve!

→ Create a landing page with the following

|  |
| --- |
| kubectl exec web-server-pod -it -- /bin/sh  echo "This is our new landing page!" >/usr/share/nginx/html/index.html  chmod +x /usr/share/nginx/html/index.html  exit |

→ Get the homepage for our webserver

|  |
| --- |
| curl <POD\_IP> |

You should see our new landing page that lives on our volume.

→ Delete our pod and make a new one

|  |
| --- |
| kubectl delete pod web-server-pod  kubectl apply -f pod.yaml  kubectl get pods -o wide |

→ Notice our data is still there

|  |
| --- |
| curl <POD\_IP> |

**Step 5: Cleanup**

→ From within the ~/persistence directory

|  |
| --- |
| kubectl delete -f .  docker stop nfs && docker rm nfs |

### **Task 2: Dynamic Provisioning**

As you can see, static provisioning can be cumbersome since an admin must create the PV objects and have ways to ensure the correct users get the correct volumes matched with their PVCs. You may not have noticed, but our PV was 3GB while we requested only 1GB with our PVC...what a waste!. Typically, 100s of PVs are pre-created by administrators and bound to PVCs in a haphazard way, matching with the best PV it can, even if it is much larger than needed.

Dynamic Provisioning allows for the exact number of PVs that are needed to exist at any given time, and of the exact size requested. It does, however, require that you use storage systems that kubernetes has the ability to provision storage in. This list is growing rapidly as Dynamic Provisioning is becoming the de facto standard for provisioning storage for workloads on k8s.

#### **Step 1: Examine StorageClass**

Our cluster has a default storage class called "standard". If we created a PVC and do not specify a StorageClass, this one will be used. It tells Kubernetes what plugin to use when creating volumes, in this case, it will create Google Cloud Persistent Disks.

|  |
| --- |
| $ kubectl describe sc standard  Name: standard  IsDefaultClass: Yes  Annotations: storageclass.beta.kubernetes.io/is-default-class=true  Provisioner: kubernetes.io/gce-pd  Parameters: type=pd-standard  AllowVolumeExpansion: <unset>  MountOptions: <none>  ReclaimPolicy: Delete  VolumeBindingMode: Immediate  Events: <none> |

#### **Step 2: Create PVC object**

→ vi **pvc\_dynamic.yaml**, type ":set paste", press **enter**. Then press **i**, paste the following:

|  |
| --- |
| apiVersion: v1  kind: PersistentVolumeClaim  metadata:  name: static-files  spec:  accessModes:  - ReadWriteOnce  resources:  requests:  storage: 1Gi |

→ Hit **ESC**, then type ':wq' and press **enter** to save the file.

→ Create PersistentVolumeClaim

|  |
| --- |
| kubectl apply -f pvc\_dynamic.yaml |

|  |
| --- |
| persistentvolumeclaim/static-files created |

#### **Step 3: Witness PV creation**

Our PVC should trigger a PV to be created and become bound to it

→ Examine PVC

|  |
| --- |
| kubectl get pvc static-files |

|  |
| --- |
| NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE  static-files Bound **pvc-e0b57a5a-8640-11e9-ace5-42010a8e0020** 1Gi RWO standard 16s |

→ Now take a look at it's PV (under the VOLUME column)

|  |
| --- |
| kubectl get pv <PV> |

|  |
| --- |
| Name: pvc-e0b57a5a-8640-11e9-ace5-42010a8e0020  Labels: failure-domain.beta.kubernetes.io/region=us-east1  failure-domain.beta.kubernetes.io/zone=us-east1-c  Annotations: kubernetes.io/createdby: gce-pd-dynamic-provisioner  pv.kubernetes.io/bound-by-controller: yes  pv.kubernetes.io/provisioned-by: kubernetes.io/gce-pd  Finalizers: [kubernetes.io/pv-protection]  StorageClass: standard  Status: Bound  Claim: student1/static-files  Reclaim Policy: Delete  Access Modes: RWO  Capacity: 1Gi  Node Affinity:  Required Terms:  Term 0: failure-domain.beta.kubernetes.io/zone in [us-east1-c]  failure-domain.beta.kubernetes.io/region in [us-east1]  Message:  Source:  Type: GCEPersistentDisk (a Persistent Disk resource in Google Compute Engine)  PDName: gke-class-cluster-4c6e-pvc-e0b57a5a-8640-11e9-ace5-42010a8e0020  FSType: ext4  Partition: 0  ReadOnly: false  Events: <none> |

The PV object contains a lot of information about where the data lives on the storage system. This is needed for kubernetes to attach the storage to Pods or delete it when we delete the PVC.

### **Task 3: Cleanup**

|  |
| --- |
| kubectl delete -f pvc\_dynamic.yaml |

### **Task 4: Bonus: Configure basic auth for nginx**

For a challenge, let's make a ConfigMap and a Secret object. Our ConfigMap will define the default.conf file for nginx configuration. Our Secret will define a htpasswd file containing user credentials.

ConfigMaps - <https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#create-a-configmap>

Secrets - <https://kubernetes.io/docs/concepts/configuration/secret/>

#### **Step 1. Create ConfigMap**

→ Using the docs, create a ConfigMap that defines a key, '**default.conf**', with the following contents. Ensure you properly handle multi-line values in YAML (look for examples using the '|' operator)

|  |
| --- |
| server {  listen 80;  server\_name localhost;  location / {  root /usr/share/nginx/html;  index index.html index.htm;  auth\_basic "closed site";  auth\_basic\_user\_file "/etc/nginx/htpasswd";  }  } |

#### **Step 2. Create Secret**

I have pregenerated hashed user credentials for the htpasswd file. The following defines the user, 'user', with a password of 'password1'.

→ Our Secret object should define 'htpasswd' with the following contents.

|  |
| --- |
| user:$apr1$//v0t3Wj$LTIYHkRDvddkl699gPLCA/ |

→ Note, you will need base64 encode these contents before putting them in the secret

|  |
| --- |
| echo 'user:$apr1$//v0t3Wj$LTIYHkRDvddkl699gPLCA/' | base64 |

#### **Step 3. Create basic\_auth\_pod.yaml to mount the CM and Secret**

→ Create the pod

|  |
| --- |
| kind: Pod  apiVersion: v1  metadata:  name: web-server-pod  spec:  containers:  - name: frontend  image: nginx  volumeMounts:  - mountPath: /etc/nginx/conf.d/default.conf  name: config-volume  subPath: default.conf  - mountPath: "/etc/nginx/htpasswd"  name: creds-volume  subPath: htpasswd  volumes:  - name: creds-volume  secret:  secretName: nginx-creds  - name: config-volume  configMap:  name: nginx-config |

#### **Step 4. Test that the app is only accessible when providing credentials**

→

|  |
| --- |
| export POD\_IP=$(kubectl get pod web-server-pod -o jsonpath={.status.podIP})  curl $POD\_IP |

|  |
| --- |
| <html>  <head><title>401 Authorization Required</title></head>  <body>  <center><h1>401 Authorization Required</h1></center>  <hr><center>nginx/1.15.12</center>  </body>  </html> |

→ Send request with credentials

|  |
| --- |
| curl --basic --user user:password1 $POD\_IP |

#### **Step 5: Cleanup**

|  |
| --- |
| kubectl delete all --all  kbuectl delete pvc --all |