

Homework 15 – Kubernetes Storage – Velibor Stanisic

Kubernetes is a free and open-source container orchestration platform. It provides services and management capabilities needed to efficiently deploy, operate, and scale containers in a cloud or cluster environment.

When managing containerized environments, Kubernetes storage is useful for storage administrators, because it allows them to maintain multiple forms of persistent and non-persistent data in a Kubernetes cluster. This makes it possible to create dynamic storage resources that can serve different types of applications.

Practice 1: Direct provisioning of Azure File storage

1. Login to Azure and connect to your AKS cluster.

The screenshot displays the Microsoft Azure portal interface for a Kubernetes service named 'myAKSHare'. The left sidebar shows navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, and Microsoft Defender for Cloud. The main content area is divided into sections: Essentials, Kubernetes services, Node pools, Configuration, and Extensions + applications (preview). The Essentials section provides a summary of the cluster's status (Succeeded (Running)), location (East US), subscription, and subscription ID. The Kubernetes services section shows encryption at rest and virtual node pools. The Node pools section lists the number of node pools (1), Kubernetes version (1.24.10), and node sizes (Standard_DS2_v2). The Configuration section details the Kubernetes version (1.24.10), auto upgrade type (Patch), authentication and authorization (Local accounts with Kubernetes RBAC), and local accounts (Enabled). The Extensions + applications (preview) section indicates that no extensions are installed. The Networking section on the right provides details about the API server address, network type (Kubenet), pod CIDR, service CIDR, DNS service IP, Docker bridge CIDR, network policy, load balancer, HTTP application routing, private cluster, authorized IP ranges, and application gateway ingress controller. The Integrations section shows that container insights are enabled and the workspace resource ID is 'DefaultWorkspace-7c4c34d4-0360-400c-804a-ca04899cc5-EUS'.

2. Check if any pods run under the default namespace if so delete everything under the default namespace.

```
Bash
velibor [ ~ ] $ kubectl get pods --namespace=default
No resources found in default namespace.
velibor [ ~ ] $
```

3. In this practice we will directly provision Azure Files to a pod running inside AKS.

4. First create the Azure Files share. Run the following commands:

Change these four parameters as needed for your own environment

AKS_PERS_STORAGE_ACCOUNT_NAME=mystorageaccount\$RANDOM

AKS_PERS_RESOURCE_GROUP=myAKSShare

AKS_PERS_LOCATION=eastus

AKS_PERS_SHARE_NAME=aksshare

```
Bash
velibor [ ~ ]$ export AKS_PERS_STORAGE_ACCOUNT_NAME=mystorageaccount$RANDOM
velibor [ ~ ]$ export AKS_PERS_RESOURCE_GROUP=myAKSShare
velibor [ ~ ]$ export AKS_PERS_LOCATION=eastus
velibor [ ~ ]$ export AKS_PERS_SHARE_NAME=aksshare
velibor [ ~ ]$
```

Create a resource group

az group create --name \$AKS_PERS_RESOURCE_GROUP --location \$AKS_PERS_LOCATION

```
Bash
velibor [ ~ ]$ az group create --name $AKS_PERS_RESOURCE_GROUP --location $AKS_PERS_LOCATION
{
  "id": "/subscriptions/7c4c34d4-0360-400c-80f4-ca0f4899fcc5/resourceGroups/myAKSShare",
  "location": "eastus",
  "managedBy": null,
  "name": "myAKSShare",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null,
  "type": "Microsoft.Resources/resourceGroups"
}
velibor [ ~ ]$
```

Create a storage account

az storage account create -n \$AKS_PERS_STORAGE_ACCOUNT_NAME -g

\$AKS_PERS_RESOURCE_GROUP -l

\$AKS_PERS_LOCATION --sku Standard_LRS

```
Bash
velibor [ ~ ]$ az storage account create -n $AKS_PERS_STORAGE_ACCOUNT_NAME -g $AKS_PERS_RESOURCE_GROUP -l $AKS_PERS_LOCATION --sku Standard_LRS
The public access to all blobs or containers in the storage account will be disallowed by default in the future, which means default value for --allow-blob-public-access is still null but will be equivalent to false.
{
  "accessTier": "Hot",
  "allowBlobPublicAccess": true,
  "allowCrossTenantReplication": null,
  "allowSharedKeyAccess": null,
  "allowedCopyScope": null,
  "azureFilesIdentityBasedAuthentication": null,
  "blobRestoreStatus": null,
  "creationTime": "2023-04-05T22:38:28.436341+00:00",
  "customDomain": null,
  "defaultToOAuthAuthentication": null,
  "dnsEndpointType": null,
  "enableHttpsTrafficOnly": true,
  "enableMfa": null,
  "encryption": {
    "encryptionIdentity": null,
    "keySource": "Microsoft.Storage",
    "keyVaultProperties": null,
    "requireInfrastructureEncryption": null,
    "services": {
      "blob": {
        "enabled": true,
        "keyType": "Account",
        "lastEnabledTime": "2023-04-05T22:38:28.623841+00:00"
      },
      "file": {
        "enabled": true,
        "keyType": "Account",
        "lastEnabledTime": "2023-04-05T22:38:28.623841+00:00"
      },
      "queue": null,
      "table": null
    }
  },
  "extendedLocation": null,
  "failoverInProgress": null,
  "geoReplicationStats": null,
  "id": "/subscriptions/7c4c34d4-0360-400c-80f4-ca0f4899fcc5/resourceGroups/myAKSShare/providers/Microsoft.Storage/storageAccounts/mystorageaccount16732",
  "identity": null,
  "immutableStorageWithVersioning": null,
  "isHnsEnabled": null,
  "isLocalUserEnabled": null,
  "isSftpEnabled": null,
  "keyCreationTime": {
    "key1": "2023-04-05T22:38:28.608215+00:00",
    "key2": "2023-04-05T22:38:28.608215+00:00"
  },
  "keyPolicy": null,
  "kind": "StorageV2",
  "largeFileSharesState": null,
  "lastGeoFailoverTime": null,
}
```

```
# Export the connection string as an environment variable, this is used when creating the Azure file share
export AZURE_STORAGE_CONNECTION_STRING=$(az storage account show-connection-string -n
$AKS_PERS_STORAGE_ACCOUNT_NAME -g $AKS_PERS_RESOURCE_GROUP -o tsv)
```

```
Bash
velibor [ ~ ]$ export AZURE_STORAGE_CONNECTION_STRING=$(az storage account show-connection-string -n $AKS_PERS_STORAGE_ACCOUNT_NAME -g $AKS_PERS_RESOURCE_GROUP -o tsv)
velibor [ ~ ]$
```

Create the file share

```
az storage share create -n $AKS_PERS_SHARE_NAME --connection-string
$AZURE_STORAGE_CONNECTION_STRING
```

```
Bash
velibor [ ~ ]$ az storage share create -n $AKS_PERS_SHARE_NAME --connection-string $AZURE_STORAGE_CONNECTION_STRING
{
  "created": true
}
velibor [ ~ ]$
```

Get storage account key

```
STORAGE_KEY=$(az storage account keys list --resource-group
$AKS_PERS_RESOURCE_GROUP --account-name
$AKS_PERS_STORAGE_ACCOUNT_NAME --query "[0].value" -o tsv)
```

```
Bash
velibor [ ~ ]$ az storage share create -n $AKS_PERS_SHARE_NAME --connection-string $AZURE_STORAGE_CONNECTION_STRING
{
  "created": true
}
velibor [ ~ ]$
```

Echo storage account name and key

```
echo Storage account name: $AKS_PERS_STORAGE_ACCOUNT_NAME
echo Storage account key: $STORAGE_KEY
```

```
Bash
velibor [ ~ ]$ echo "Storage account name: $AKS_PERS_STORAGE_ACCOUNT_NAME"
Storage account name: mystorageaccount10732
velibor [ ~ ]$ echo "Storage account key: $STORAGE_KEY"
Storage account key: YAvLpyGFAI6jITrYDPPms+ZXRdfyHtwGxREnI9Wxv3IbuX9r6B284oSQckPKS4w9Hx5TAVoHUm+ASToVA0cg=
velibor [ ~ ]$
```

5. Make a note of the storage account name and key shown at the end of the script output. These values are needed when you create the Kubernetes volume in one of the following steps.

6. Now we will need to create a Kubernetes secret that will be used to mount the Az File Share to the pod. You need to hide this information from the pod's definition and K8S secret is the best way to do it.

7. Run the following (single) command to create the secret:

```
kubectl create secret generic azure-secret --from- \
literal=azurestorageaccountname=$AKS_PERS_STORAGE_ACCOUNT_NAME \
--from-literal=azurestorageaccountkey=$STORAGE_KEY
```

```
Bash
velibor [ ~ ]$ kubectl create secret generic azure-secret --from-literal=azurestorageaccountname=$AKS_PERS_STORAGE_ACCOUNT_NAME --from-literal=azurestorageaccountkey=$STORAGE_KEY
secret/azure-secret created
velibor [ ~ ]$
```

8. Check if secret was created. Run `kubectl get secret -A`.

9. Now we can create the pod and mount the Azure File. Create a new file named azure-files-pod.yaml with the following contents:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
  resources:
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 250m
      memory: 256Mi
  volumeMounts:
  - name: azure
    mountPath: /mnt/azure
volumes:
- name: azure
  azureFile:
    secretName: azure-secret
    shareName: aksshare
    readOnly: false
```

10. Run `kubectl apply -f azure-files-pod.yaml`.

A terminal window with a dark background and light text. The prompt is 'velibor [~]\$'. The user enters 'vim azure-files-pod.yaml', then 'kubectl apply -f azure-files-pod.yaml'. The output shows 'pod/mypod created' and the prompt returns to 'velibor [~]\$'.

11. You now have a running pod with an Azure Files share mounted at `/mnt/azure`.

12. You can use `kubectl describe pod mypod` to verify the share is mounted successfully. Search for the Volumes section of the output.

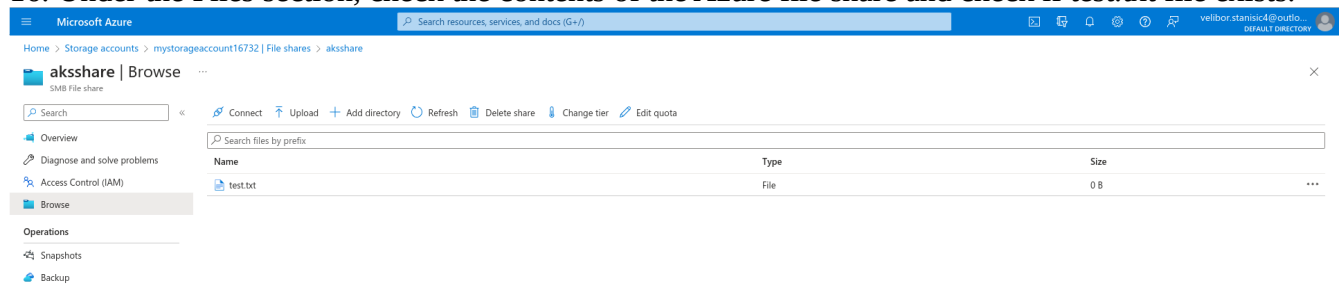
13. Now exec to the pod and try to access the mounted file share. Run the following command
`kubectl exec -it mypod -- bash`

A terminal window showing the execution of 'kubectl exec -it mypod -- sh'. The prompt changes to '/ #', indicating the user is now inside the pod's shell.

14. Go to `/mnt/azure` and create a blank file `test.txt` file.

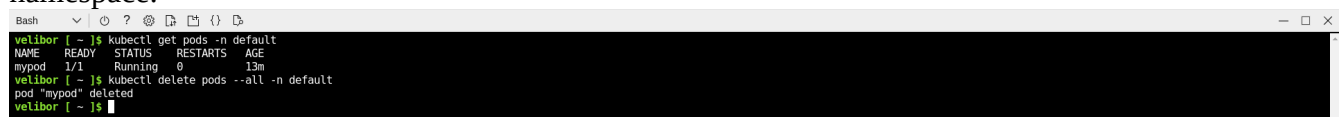
A terminal window showing the user navigating to '/mnt/azure' and creating a file. The commands are 'cd /mnt/azure/', 'touch test.txt', and 'ls'. The output shows 'test.txt' and the prompt returns to '/mnt/azure #'.

15. Go to the portal and locate your Azure storage provisioned for this practice.
16. Under the Files section, check the contents of the Azure file share and check if test.txt file exists.



Practice 2: Provisioning Azure File storage using PVs and PVCs

1. Login to Azure and connect to your AKS cluster.
2. Check if any pods run under the default namespace if so delete everything under the default namespace.



3. Now we will provision Azure files storage to a pod using PV and PVC.

4. Create a azurefile-mount-options-pv.yaml file with a PersistentVolume like this:

apiVersion: v1

kind: PersistentVolume

metadata:

name: azurefile

spec:

capacity:

storage: 5Gi

accessModes:

- ReadWriteMany

azureFile:

secretName: azure-secret

shareName: aksshare

readOnly: false

mountOptions:

- dir_mode=0777

- file_mode=0777

- uid=1000

- gid=1000

- mfsymlinks

- nobrl

5. Note the access mode. Can you use other mode with Azure files?


Azure Files supports two access modes in Kubernetes: ReadWriteOnce and ReadWriteMany

6. Now create a azurefile-mount-options-pvc.yaml file with a PersistentVolumeClaim that uses the PersistentVolume like this:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azurefile
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ""
resources:
  requests:
    storage: 5Gi
```

7. Execute `kubectl apply -f azurefile-mount-options-pv.yaml` and `kubectl apply -f azurefile-mount-options-pvc.yaml`.

8. Verify your PersistentVolumeClaim is created and bound to the PersistentVolume. Run `kubectl get pvc azurefile`.



```
Bash
velibor [ ~ ] $ kubectl get pvc azurefile
NAME      STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
azurefile Bound    azurefile 5Gi        RWO                       29s
velibor [ ~ ] $
```

9. Now we can embed the PVC info inside our pod definition. Create the following file azure-files-pod.yaml with following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
  resources:
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 250m
      memory: 256Mi
  volumeMounts:
    - name: azure
      mountPath: /mnt/azure
  volumes:
    - name: azure
      persistentVolumeClaim:
        claimName: azurefile
```

10. Run `kubectl apply -f azure-files-pod.yaml`.

```
Bash
velibor [ ~ ]$ kubectl apply -f azure-files-pod.yaml
pod/mypod created
velibor [ ~ ]$
```

11. You now have a running pod with an Azure Files share mounted at `/mnt/azure`.

12. You can use `kubectl describe pod mypod` to verify the share is mounted successfully. Search for the Volumes section of the output.

13. Now exec to the pod and try to access the mounted file share. Run the following command `kubectl exec -it mypod -- bash`


```
Bash
velibor [ ~ ]$ kubectl exec -it mypod -- sh
/ #
```

14. Go to `/mnt/azure` and create a blank file `test.txt` file.

```
Bash
velibor [ ~ ]$ kubectl exec -it mypod -- sh
/ # cd /mnt/azure/
/mnt/azure # touch test.txt
/mnt/azure # ls -l
total 0
-rwxrwxrwx 1 1000 1000 0 Apr 5 23:24 test.txt
/mnt/azure #
```

15. Go to the portal and locate your Azure storage provisioned for this practice.

16. Under the Files section, check the contents of the Azure file share and check if `test.txt` file exists.

Name	Type	Size
 test.txt	File	0 B

17. Delete the mypod the pv and pvc you have created so far. What happens to the Azure File share? The Azure Files share will continue to exist in Azure storage account even after we deleted the resources in your Kubernetes cluster.

Practice 3: Provisioning Azure file storage using Storage Classes

1. Login to Azure and connect to your AKS cluster.

2. Check if any pods run under the default namespace if so delete everything under the default namespace.

```
Bash
Requesting a Cloud Shell.Succeeded.
Connecting terminal...

velibor [ ~ ]$ kubectl get pods -n default
No resources found in default namespace.
velibor [ ~ ]$
```

3. Now we will provision file storage using the definition of storage classes. Create a file named azure-file-sc.yaml and copy in the following example manifest:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: my-azurefile
provisioner: kubernetes.io/azure-file
mountOptions:
- dir_mode=0777
- file_mode=0777
- uid=0
- gid=0
- mfsymlinks
- cache=strict
- actimeo=30
parameters:
  skuName: Standard_LRS
```

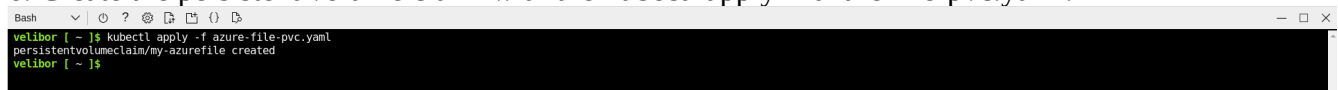
4. Create the storage class with kubectl apply -f azure-file-sc.yaml .

A terminal window with a dark background and light green text. The prompt is 'velibor [~]\$'. The first command is 'vim azure-file-sc.yaml'. The second command is 'kubectl apply -f azure-file-sc.yaml'. The output is 'storageclass.storage.k8s.io/my-azurefile created'. The prompt returns to 'velibor [~]\$'.

5. Now we will create the PVC that will consume the storage class defined previously. Create a file named azure-file-pvc.yaml and copy in the following YAML:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-azurefile
spec:
  accessModes:
  - ReadWriteMany
  storageClassName: my-azurefile
resources:
  requests:
    storage: 5Gi
```

6. Create the persistent volume claim with the kubectl apply -f azure-file-pvc.yaml.

A terminal window with a dark background and light green text. The prompt is 'velibor [~]\$'. The command is 'kubectl apply -f azure-file-pvc.yaml'. The output is 'persistentvolumeclaim/my-azurefile created'. The prompt returns to 'velibor [~]\$'.

7. Once completed, the file share will be created. A Kubernetes secret is also created that includes connection information and credentials. You can use the `kubectl get pvc my-azurefile` command to view the status of the PVC.

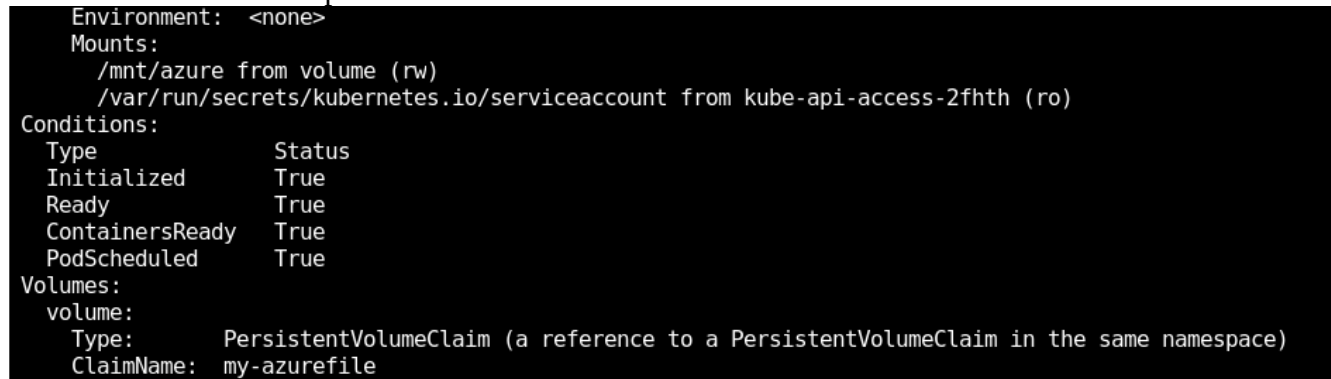
8. Now we will create the pod that consumes the PVC. Create a file named `azure-pvc-files.yaml`, and copy in the following YAML. Make sure that the `claimName` matches the PVC created in the last step:

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
  resources:
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 250m
      memory: 256Mi
  volumeMounts:
  - mountPath: "/mnt/azure"
    name: volume
  volumes:
  - name: volume
    persistentVolumeClaim:
      claimName: my-azurefile
```

9. Create the pod with `kubectl apply -f azure-pvc-files.yaml`.

A terminal window with a dark background and light green text. The prompt is 'velibor [~]\$'. The user enters 'vim azure-pvc-files.yaml', then 'velibor [~]\$ kubectl apply -f azure-pvc-files.yaml'. The output is 'pod/mypod created' and 'velibor [~]\$'.

10. Do a describe on the pod and check the volumes mounted.

A terminal window with a dark background and light green text. The output of 'kubectl describe pod mypod' is shown. It includes fields for Environment, Mounts, Conditions, and Volumes. The 'Mounts' section shows two mounts: '/mnt/azure' from 'volume' (rw) and '/var/run/secrets/kubernetes.io/serviceaccount' from 'kube-api-access-2fhth' (ro). The 'Conditions' section shows a table with columns 'Type' and 'Status', with rows for 'Initialized', 'Ready', 'ContainersReady', and 'PodScheduled', all with a status of 'True'. The 'Volumes' section shows 'volume' with 'Type: PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)' and 'ClaimName: my-azurefile'.

11. Delete everything created under this practice including the storage class.

Practice 4: Direct provisioning of Azure Disk storage

1. Login to Azure and connect to your AKS cluster.
2. Check if any pods run under the default namespace if so delete everything under the default namespace.

```
Bash
velibor [ ~ ]$ kubectl get pods -n default
No resources found in default namespace.
velibor [ ~ ]$
```

3. In this practice we will directly provision Azure Disk to a pod running inside AKS.
4. First create the disk in the node resource group. First, get the node resource group name with `az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv`.

```
Bash
velibor [ ~ ]$ az aks show --resource-group MordorGroup --name Mordorce --query nodeResourceGroup -o tsv
MC_MordorGroup_Mordorce_eastus
velibor [ ~ ]$
```

5. Now create a disk using:

```
az disk create \
--resource-group MC_myResourceGroup_myAKSCluster_eastus \
--name myAKSDisk \
--size-gb 20 \
--query id --output tsv
```

```
Bash
velibor [ ~ ]$ az disk create \
--resource-group MC_MordorGroup_Mordorce_eastus \
--name myAKSDisk \
--size-gb 20 \
--zone 1 \
--query id --output tsv
/subscriptions/7c4c34d4-8360-409c-88f4-ca0f4899fcc5/resourceGroups/MC_MordorGroup_Mordorce_eastus/providers/Microsoft.Compute/disks/myAKSDisk
velibor [ ~ ]$
```

6. Make a note of the disk resource ID shown at the end of the script output. This value is needed when you create the Kubernetes volume in one of the following steps.

7. Now we can create the pod and mount the Azure Disk. Create a new file named `azure-disk-pod.yaml` with the following contents:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
  resources:
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 250m
      memory: 256Mi
```

volumeMounts:

- name: azure
mountPath: /mnt/azure

volumes:

- name: azure

azureDisk:

- kind: Managed

- diskName: myAKSDisk

- diskURI: <!!!!!!!!!!!!!! Put the Disk resource id noted before!!!>

8. Run `kubectl apply -f azure-disk-pod.yaml`.

```
Bash
velibor [ ~ ]$ kubectl apply -f azure-disk-pod.yaml
pod/mypod created
velibor [ ~ ]$
```

9. You now have a running pod with an Azure Disk mounted at /mnt/azure.

10. You can use `kubectl describe pod mypod` to verify the share is mounted successfully. Search for the Volumes section of the output.

```
Mounts:
  /mnt/azure from azure (rw)
  /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-rvcww (ro)
Conditions:
  Type             Status
  Initialized       True
  Ready            True
  ContainersReady  True
  PodScheduled     True
Volumes:
  azure:
    Type:          AzureDisk (an Azure Data Disk mount on the host and bind mount to the pod)
    DiskName:      myAKSDisk
    DiskURI:       /subscriptions/7c4c34d4-0360-480c-80f4-ca0f4899fcc5/resourceGroups/MC_MordorGroup_Mordorce_eastus/providers/Microsoft.Compute/disks/myAKSDisk
    Kind:          Managed
    FSType:        ext4
    CachingMode:   ReadWrite
    ReadOnly:      false
  kube-api-access-rvcww:
    Type:          Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds: 3607
    ConfigMapName:    kube-root-ca.crt
    ConfigMapOptional: <nil>
    DownwardAPI:     true
```

11. Now exec to the pod and try to access the mounted volume. Run the following command `kubectl exec -it mypod -- bash`

```
velibor [ ~ ]$ kubectl exec -it mypod -- bash
/ #
```

12. Go to /mnt/azure and try create a blank file test.txt file.

```
/ # cd /mnt/azure/
/mnt/azure # touch test.txt
/mnt/azure # ls -l
total 16
drwx----- 2 root   root   16384 Apr  6 01:10 lost+found
-rw-r--r--  1 root   root     0 Apr  6 01:13 test.txt
/mnt/azure #
```

13. Delete everything created by this practice.

Practice 5: Provisioning Azure Disk storage using Storage Classes

1. Login to Azure and connect to your AKS cluster.

2. Check if any pods run under the default namespace if so delete everything under the default namespace.

```
Bash
velibor [ ~ ]$ kubectl get pods -n default
No resources found in default namespace.
velibor [ ~ ]$
```

3. Now we will provision Azure disk and attach it to a running pod but this time using dynamic provisioning with storage classes. List the available storage classes, run `kubectl get sc`.

```
velibor [ ~ ]$ kubectl get sc
NAME                                PROVISIONER      RECLAIMPOLICY   VOLUMEBINDINGMODE  ALLOWVOLUMEEXPANSION  AGE
azurefile                           file.csi.azure.com Delete           Immediate           true                  23m
azurefile-csi                        file.csi.azure.com Delete           Immediate           true                  23m
azurefile-csi-premium                file.csi.azure.com Delete           Immediate           true                  23m
azurefile-premium                    file.csi.azure.com Delete           Immediate           true                  23m
default (default)                    disk.csi.azure.com Delete           WaitForFirstConsumer true                  23m
managed                              disk.csi.azure.com Delete           WaitForFirstConsumer true                  23m
managed-csi                          disk.csi.azure.com Delete           WaitForFirstConsumer true                  23m
managed-csi-premium                  disk.csi.azure.com Delete           WaitForFirstConsumer true                  23m
managed-premium                      disk.csi.azure.com Delete           WaitForFirstConsumer true                  23m
velibor [ ~ ]$
```

4. Examine the output. Each AKS cluster includes four pre-created storage classes, two of them configured to work with Azure disks, default and managed-premium. We will use the managed-premium in our PVC definition since it uses premium type of disks.

5. Now we will create the PVC that will consume the storage class defined previously. Create a file named `azure-premium.yaml` and copy in the following YAML:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azure-managed-disk
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: managed-premium
resources:
  requests:
    storage: 5Gi
```

6. Create the persistent volume claim with the `kubectl apply -f azure-premium.yaml`.

```
Bash
velibor [ ~ ]$ vim azure-premium.yaml
velibor [ ~ ]$ kubectl apply -f azure-premium.yaml
persistentvolumeclaim/azure-managed-disk created
velibor [ ~ ]$
```

7. Check the status of your PVC.

```
Bash
velibor [ ~ ]$ kubectl get pvc
NAME                STATUS    VOLUME    CAPACITY   ACCESS MODES   STORAGECLASS    AGE
azure-managed-disk  Pending  managed-premium  5Gi
```

8. Now we will create the pod that consumes the PVC. Create a file named azure-pvc-disk.yaml, and copy in the following YAML. Make sure that the claimName matches the PVC created in the last step:

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
  resources:
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 250m
      memory: 256Mi
  volumeMounts:
  - mountPath: "/mnt/azure"
    name: volume
  volumes:
  - name: volume
    persistentVolumeClaim:
      claimName: azure-managed-disk
```

9. Create the pod with `kubectl apply -f azure-pvc-disk.yaml`.

A terminal window with a dark background and light green text. The prompt is 'velibor [~]\$'. The user enters 'vim azure-pvc-disk.yaml', then 'kubectl apply -f azure-pvc-disk.yaml'. The output is 'pod/mypod created'. The prompt returns to 'velibor [~]\$'.

10. Do a describe on the pod and check the volumes mounted.

A terminal window showing the output of 'kubectl describe pod mypod'. The output includes: Environment: <none>, Mounts: /mnt/azure from volume (rw), /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-vfj6l (ro), Conditions: Type Status, Initialized True, Ready True, ContainersReady True, PodScheduled True, Volumes: volume: Type: PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace), ClaimName: azure-managed-disk, ReadOnly: false, kube-api-access-vfj6l:.

11. Delete everything created under this practice including the storage class.

***Why by default we cannot schedule pods to run on the master node?**

When we setup a cluster a taint label is set on the master node that prevents any pods being scheduled on it. I guess this is because the job of the master node is managing the cluster and not adding additional tasks to it will ensure the system to run more efficiently.