**volumes in Kubernetes**

Arsh Sharma

**Grokking the Behavioral Interview**

Many candidates are rejected or down-leveled in technical interviews due to poor performance in behavioral or cultural fit interviews. Ace your interviews with this free course, where you will practice confidently tackling behavioral interview questions.

Get Free Course

**Overview**

When working with K8s, we want to ensure that the data created by our application, which is running in containers, survive if something happens to any one of our running containers.

Volumes aims to solve this problem.

In [this](https://dev.to/rinkiyakedad/learning-to-use-kubernetes-1l6g) shot, we ran a simple Node.js app on our cluster. In [another shot](https://dev.to/rinkiyakedad/diving-deeper-into-docker-volumes-and-bind-mounts-4fp0) which was a part of my series on Docker, I’ve also talked about volumes.

Here we will combine those two concepts and look at how we can use volumes when working with K8s.

I recommend that you go through those two shots before reading this one.

When working with Docker, we use:

* Anonymous volumes
* Named volumes
* Bind mounts.

K8s, in contrast to Docker, support a large number of volume types. Apart from regular volumes, it also has something called Persistent Volumes, which we will look at in more detail.

**Understanding volume in docker**

Let’s understand the above-mentioned types by going through three common problems that we might face. We’ll solve each one of them using a different type of volume each time.

We’ll be using the deployment.yaml file as discussed in the previous [shots](https://dev.to/rinkiyakedad/diving-deeper-into-docker-volumes-and-bind-mounts-4fp0) and build on top of that.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

// apiVersion: apps/v1

kind: Deployment

metadata:

  name: node-app-deployment

spec:

  replicas: 1

  selector:

    matchLabels:

      anything: node-app

  template:

    metadata:

      labels:

        anything: node-app

    spec:

      containers:

        - name: node-app-container

          image: YourDockerHubName/node-image





**The first problem**

* Let’s say that we have a container running inside a pod and the application stores some user data.
* Now, if something were to happen to this container and it has to restart, then all of the data we stored in the previous shots would be lost. This is not at all desirable and is something we will fix.
* The simplest volume type we can use to fix this is the emptyDir type.

**Solution to the first problem**

* What we will do is that in our deployment.yaml file, in the pod specification, next to containers, add the volumes key where we list all the volumes within this pod.
* This will set up our volume, but we also need to make it accessible inside the container.
* So, we will add the volumeMounts key in our configuration of the containers.

The nice part is that using the emptyDir type is fairly simple.

**deployment.yaml of first problem**

Let’s see how the deployment.yaml file will look.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

// apiVersion: apps/v1

kind: Deployment

metadata:

  name: node-app-deployment

spec:

  replicas: 1

  selector:

    matchLabels:

      anything: node-app

  template:

    metadata:

      labels:

        anything: node-app

    spec:

      containers:

        - name: node-app-container

          image: YourDockerHubName/node-image

          volumeMounts:

            - mountPath: /app/userData

              name: userData-volume

      volumes:

        - name: userData-volume

          emptyDir: {}





**Explanation of the first problem**

* We first listed the volumes we wanted to use under the volumeMounts key. Here we specify the mountPath, which is the location in our container. Location of container stores the files and the name of the volume.
* We configure each volume under the volumes key by first specifying its name and then the config.
* The config is based on the type, which we have to specify first.
* Here the type is emptyDir, and we didn’t specify any special config for it, implying that we want to use the emptyDir type of volume with its default settings.
* Doing so has solved our problem. If for some reason our container now shuts down, assuming only one pod is present, then when it restarts (this is something k8s handles for us by default) the container will have access to the data that was created earlier.
* Since the data isn’t being stored in the container, a new empty directory is created whenever the pod starts.
* Containers can then write to this dir and if they restart or are removed the data survives.

**The second problem**

To solve the first problem, we stored the data on the pod so that even if the container restarts, the data is still present.

But what if the pod restarts?

* Suppose there is a single pod and it restarts. Then our data would be lost and the app won’t work while the pod is down.
* However, if we have multiple pods and one of them shuts down for some reason, then the data stored in its volume will be lost.
* Our app would still function because other pods are running and k8s will automatically redirect incoming traffic.
* We would still lose the data our shut down pod had. In short, our app would still work but not have some user data.

**Solution to the second problem**

* If you remember the K8s architecture then you might have guessed it already. We could store the data on the node running these pods, assuming it is a single node cluster.
* The **hostPath type** allows us to set a path on the host machine and then the data from that path will be exposed to different pods.
* Multiple pods can share the same path on the host machine.

**deployment.yaml of second problem**

Once again let’s first have a look at our deployment.yaml file.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

// apiVersion: apps/v1

kind: Deployment

metadata:

  name: node-app-deployment

spec:

  replicas: 1

  selector:

    matchLabels:

      anything: node-app

  template:

    metadata:

      labels:

        anything: node-app

    spec:

      containers:

        - name: node-app-container

          image: YourDockerHubName/node-image

          volumeMounts:

            - mountPath: /app/userData

              name: userData-volume

      volumes:

        - name: userData-volume

          hostPath:

            path: /data

            type: DirectoryOrCreate





**Explanation of the second problem**

While configuring the volume we have now used hostPath instead of emptyDir and provided some configuration.

* First is path, which refers to the folder on our host machine where we want to save the data.
* The second is type, where we provide the value of DirectoryOrCreate. This means that if the folder we specified above exists, use it, and if not, then create it on the host machine.

**What is a hostPath?**

**hostPath** type is similar to the Bind Mounts I talked about in the Docker series. Using this type should now solve the problem of our data being lost when pods shut down.

**The third problem**

The third problem stems from our solution for the second one.

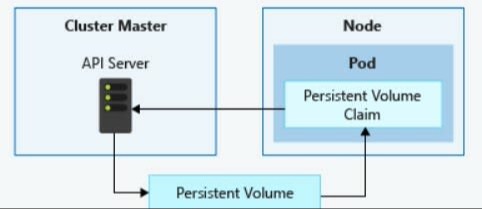
What if the pods are not present on the same node?

Then multiple pods running on different nodes will not have access to the entire user data our app stores. Here, persistent volumes will come to the rescue.

**Solution to the third problem**

**Persistent Volumes (PVs)** are pod and node independent volumes. The idea is that instead of storing the data in the pod or a node, we have entirely separate entities in our K8s cluster that are detached from our nodes.

Each pod then will have a Persistent Volume Claim (PVC), and it will use this to access the standalone entities we created.The following image I found should provide some clarity.



**Description of Persistent volumes**

Persistent Volumes are like regular volumes which also have types.

We talked about how the hostPath type is common to both and is perfect for experimenting with persistent volumes when working locally.

This is because the cluster minikube provides us a single node cluster. While you would not be using a single node cluster when working with persistent volumes, the workflow I’ll be explaining will more or less be the same.

If it’s confusing, remember that we can use the hostPath type of Persistent Volumes since we are working with a single node cluster that minikube set up for us.

**Setting up the PV**

* The first step would be to set up the persistent volume. We are doing so using the hostPath type.
* For this create a host-pv.yaml file, which should look something like this:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

// apiVersion: v1

kind: PersistentVolume

metadata:

  name: host-pv

spec:

  capacity:

    storage: 1Gi

  volumeMode: Filesystem

  storageClassName: standard

  accessModes:

    - ReadWriteOnce

  hostPath:

    path: /data

    type: DirectoryOrCreate





**Explanation of the third problem**

In the specification of this PV, we first mention the capacity.

The goal here is to control how much capacity can be used by different pods which later get executed in our cluster.

Here we mention the total capacity we want to make available.

Pods when they claim this persistent volume can define how much capacity they require.

1 Gi stands for 1 gigabyte here.

We have to specify the volumeMode key. We have two modes to choose from:

1. FileSystem
2. Block

These are two types of storage available to us. Since we will have a folder in the filesystem of our Virtual Machine we chose the FileSystem type.

**Storage classes in K8s**

K8s has a concept called storage classes. We have a storage class by default which we can see using the kubectl get sc command.

In the output, you’ll see that the name of the default storage class is standard, which is what we have specified here.

Storage classes give administrators fine control over how storage is managed.

**accessModes** tell how the PV can be accessed. Here we enlist all the modes we want to support.

* ReadWriteOnce mode allows the volume to the mounted as a read and write volume only by a single node, which is perfectly fine here since our cluster is a single node cluster.

You might want to look into other modes for a multiple node cluster, like ReadOnlyMany and ReadWriteMany.

After the accessModes, we mention the type of persistent volume (hostPath here) and its configuration like we did earlier.

**Setting up the PVC**

Simply defining the PV is not enough. We also need to define the PV Claim which the pods will use later. For this create a host-pvc.yaml file which looks something like this:

1

2

3

4

5

6

7

8

9

10

11

12

// apiVersion: v1

kind: PersistentVolumeClaim

metadata:

  name: host-pvc

spec:

  volumeName: host-pv

  accessModes:

    - ReadWriteOnce

  storageClassName: standard

  resources:

    requests:

      storage: 1Gi





* In the specification for this PV Claim, we first mention the PV name for which this claim is.
* Then we choose the accessModes from the ones we listed in the host-pv.yaml file. Since we listed only one, we have no other choice but to go with that one here.
* After that, we again mention the storage class we want to use like before.
* The resources key can be thought of as the counterpart for the capacity we mentioned in the host-pv.yaml file. Here we choose how much storage we want to request.
* We would generally not request the entire amount of storage available to us. Here it wouldn’t matter since we are just testing stuff out.

**Final configuration**

Now that we have our PV and PVC set up, all that needs to be done is make changes in the deployment.yaml file so that we use this PV instead of the hostPath we were using earlier.

Our deployment.yaml should look like this now:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

// apiVersion: apps/v1

kind: Deployment

metadata:

  name: node-app-deployment

spec:

  replicas: 1

  selector:

    matchLabels:

      anything: node-app

  template:

    metadata:

      labels:

        anything: node-app

    spec:

      containers:

        - name: node-app-container

          image: YourDockerHubName/node-image

          volumeMounts:

            - mountPath: /app/userData

              name: userData-volume

      volumes:

        - name: userData-volume

          persistentVolumeClaim:

            claimName: host-pvc





**Conclusion**

The entire file is like it was before, except that we have replaced the hostPath key with persistentVolumeClaim and specified the claim we want to use.

With this, we’re good to go. To see our persistent volume in action, apply the files using this command as we did in the previous post.

1

kubectl apply -f=host-pv.yaml -f=host-pvc.yaml -f=deployment.yaml -f=service.yaml





Make sure you have your minikube cluster up before running these commands.

Now we’ve finally set up a persistent volume. This PV is both node and pod independent.

Even though we used the hostPath type while setting up this PV, because the cluster minikube is a single node cluster, the overall process would be similar for other types of PVs.