Available online www.jsaer.com

***Journal of Scientific and Engineering Research*, 2018, 5(12):388-395**

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Persist Application Data using Docker and Kubernetes Volumes

Pallavi Priya Patharlagadda

Pallavipriya527.p@gmail.com
United States of America

Abstract: Docker's many advantages have led to its rise in popularity. A virtual machine and a Docker container are comparable. In essence, Docker lets you use a pre-configured "Linux box" within a container. The Docker container shares the host operating system's Linux kernel. Hence, starting a Docker container is often a simple and inexpensive process. In most circumstances, starting a full Docker container (comparable to starting a regular virtual machine), takes about the same amount of time as running a standard command line program. Though it's a different paradigm from the typical virtual machine method and has some unanticipated side effects for individuals coming from the virtualization field, this is wonderful since it makes deploying big systems much easier and more flexible. But how does a Docker container persist its data or share its data with other containers? Docker provides these functionalities using Docker volumes. Also, in the case of a database, we want the data to be available all the time. In Kubernetes, this can be achieved using Persistent Volumes. In this article, let's dive more into Docker volumes and Kubernetes volumes and their best practices.

Keywords: Docker volumes, Kubernetes volumes, Linux kernel

Problem Statement

In the earlier days of containerization, it was mentioned that containers should be stateless. So, Docker containers are ephemeral, as they don't persist data across runs). The whole file system of a Docker container gets destroyed when a Docker container is destroyed. How can we maintain the state of an application? Most practical applications require some sort of persistent storage. How can Docker be used to enable persistent storage? Docker Bind Mounts and volumes are the solution. Moreover, volumes are also helpful for data sharing among Docker containers. Similarly, with the development of container-native storage solutions and the maturity of Kubernetes, stateless apps have become unnecessary. It is possible that you would like to store, preserve, and back up any data generated or utilized by your application, in addition to taking advantage of the advantages of running it in a container (quick startup, high availability, self-healing, etc.). In Kubernetes, this is possible using persistent volumes. Databases are the most typical use case for persistent volumes in Kubernetes. Of course, data in a database must always be accessible. We may begin utilizing databases for our apps, such as MySQL, Cassandra, CockroachDB, and even MS SQL, by utilizing PVs.

Introduction

We will go over the idea of Docker data volumes in this post, including what they are, why they are valuable, the various sorts of volumes, how to use them, and when to utilize them. Additionally, we'll walk through a few instances of using Docker volumes with the Docker command-line tool. You should feel at ease generating and utilizing any type of Docker data volume by the time we get to the conclusion of the post. Before doing a deep dive into the Docker data volumes, let us first learn about Docker.

Docker: An open platform for creating, delivering, and executing programs is called Docker. Docker allows you to rapidly release software by separating your apps from your infrastructure. You can use Docker to manage your infrastructure in the same manner that you do your apps. You may cut down on the amount of time it takes between developing code and putting it into production by utilizing Docker's shipping, testing, and deployment processes. With Docker, you can bundle and execute an application in a container—a loosely separated

environment. You can execute several containers concurrently on a single host thanks to the isolation and security. You may use containers instead of depending on what is installed on the host because they are lightweight and come with everything required to run the application. While working, you may share containers and ensure that each person you share with gets an identical container that functions uniformly.

Docker gives you the tools and a platform to control your containers' lifecycle.

● Use containers to develop your application and any accompanying components.

● To distribute and test your program, the container serves as the unit.

● When you're prepared, launch your application as an orchestrated service or in a container in your production environment. Regardless of whether your production environment is a cloud provider, a local data center, or a combination of the two, this operates in the same way.

Benefits of using Docker:

Delivery of your applications is prompt and reliable

By enabling developers to operate in standardized settings with local containers that host their apps and services, Docker simplifies the development lifecycle. Workflows involving continuous integration and continuous delivery (CI/CD) benefit greatly from containers.

Think about the following hypothetical situation:

● Using Docker containers, engineers can collaborate while writing code locally.

● They execute both automatic and manual testing on their apps by pushing them into a test environment using Docker.

● To verify and validate their fixes, developers can redeploy their fixed code to the test environment after fixing it in the development environment.

● After testing, it's only a matter of uploading the revised image to the production environment to deliver the patch to the client.

Scaling and deploying responsively

Highly portable workloads are possible using Docker's container-based technology. A developer's laptop, real or virtual computers in a data center, cloud providers, or a combination of settings may all execute Docker containers.

Because of its lightweight design and mobility, Docker also makes it simple to manage workloads dynamically, quickly scaling up or down services and applications based on business requirements.

utilizing the same hardware with increased workloads

Docker is quick and lightweight. It offers a practical, affordable substitute for virtual machines based on hypervisors, allowing you to utilize more of your server's capacity to meet your company's objectives. When you need to accomplish more with fewer resources in small and medium deployments and high-density situations, Docker is ideal.

Volumes

For storing data in Docker containers or sharing data across containers, Docker volumes come in rather handy.

Docker volumes are crucial because they protect the whole file system of a Docker container if it is destroyed. Therefore, we must utilize Docker volumes if we wish to preserve this data. Using the -v switch, Docker volumes are attached to containers during a Docker run operation.

When do we need Docker volumes?

Data can be persisted in Docker using Bind Mounts and Docker volumes. Let’s learn about each of them.

Bind mounts:

All a bind mount entails is mapping a host machine directory to a container directory. Nevertheless, the directory remains unaffected upon removal of the container. It is a bind mount if the value of the -v or —volume flag is a path. It will create the directory if it doesn't already exist. The files found in /var/www in the container will be present in the directory /path/to/app/directory after this is done.

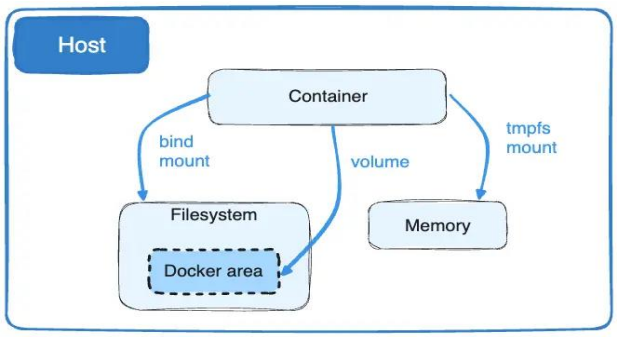
```
docker run -d --name <container-name>\
   -v /path/to/app/directory:/var/www \
<docker-image>
```

Docker Volumes

The recommended method for storing data created and utilized by Docker containers is to use volumes. Docker manages volumes entirely, whereas Bind mounts rely on the host machine's directory structure. Comparing volumes to bind mounts reveals various benefits:

● Bind mounts are more difficult to transfer or backup than volumes.
● Using the Docker API or CLI commands, you may manage volumes.
● Volumes are compatible with Windows and Linux containers.
● Sharing volumes among several containers is safer.
● Volume drivers enable you to add additional functionality, encrypt the contents of volumes, and store volumes on distant hosts or cloud providers.
● A container can pre-populate the contents of a new volume.

Moreover, volumes are frequently a better option than storing data in the writable layer of a container, as they don't increase the size of the containers that utilize them, and their contents last beyond the lifespan of the specific container.



Consider utilizing a tmpfs mount if your container creates non-persistent state data to improve speed by not writing into the writable layer of the container and to avoid keeping the data anywhere permanently.

Create and manage volumes:

Unlike a bind mount, you can create and manage volumes outside the scope of any container.

Create a volume:

docker volume create my-vol

List volumes:

docker volume ls

my-vol

Inspect a volume:

docker volume inspect my-vol

```
[
    {
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/my-vol/_data",
        "Name": "my-vol",
        "Options": {},
        "Scope": "local"
    }
]
```

Remove a volume:

docker volume rm my-vol

Start a container with a volume:

Docker generates the volume when you launch a container using one that doesn't yet exist. This example mounts the volume myvol2 into the container's /app/ directory.

The following instances of -v and —mount provide the same outcome. Running them both at the same time is not possible unless you first delete the myvol2 volume and the devtest container.

. --mount -v

```
docker run -d \
  --name testmount\
```

```
    --mount source=myvol,target=/app \
  nginx:latest
```

To confirm that Docker generated the volume and that it mounted properly, use docker inspect testmount. Locate the section on Mounts:

```
    "Mounts": [
        {
            "Type": "volume",
            "Name": "myvol",
            "Source": "/var/lib/docker/volumes/myvol/_data",
            "Destination": "/app",
            "Driver": "local",
            "Mode": "",
            "RW": true,
            "Propagation": ""
        }
],
```

This demonstrates the read-write nature of the mount, its volume status, and the accuracy of the source and destination.

Stop the container and remove the volume.

docker container stop testmount

docker container rm testmount

docker volume rm myvol2

Populate a volume using a container:

When you launch a container that generates a new volume and the container contains files or directories in the /app/ directory, for example, Docker replicates the contents of the directory onto the volume. The pre-populated content is then accessible to other containers that utilize the volume when the container mounts and starts using it.

The next example demonstrates this by starting a nginx container and adding items from the container's /usr/share/nginx/html directory to the new volume nginx-vol. This is the directory where Nginx keeps its default HTML files.

The outcome of the --mount and -v examples is the same.

--mount -v

```
    docker run -d \
      --name=nginxtest \
      --mount source=nginx-vol,destination=/usr/share/nginx/html \
  nginx:latest
```

Execute the below commands to tidy up the volumes and containers after completing any of these examples. Removing the volume is a different step.

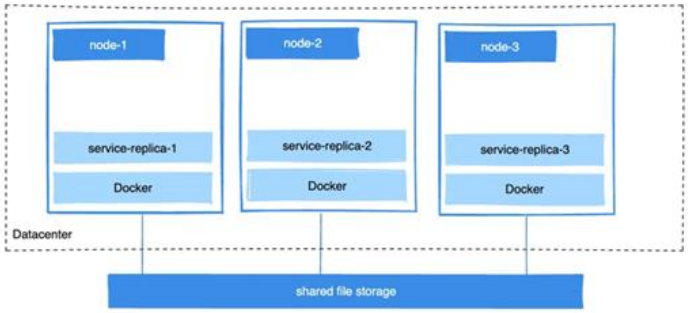
docker container stop nginxtest

docker container rm nginxtest

docker volume rm nginx-vol

Share data between machines.

You might need to set up numerous copies of the same service to have access to the same files while creating fault-tolerant apps.



When you are designing your apps, there are several approaches to accomplishing this. One is to include file storage on cloud object storage systems such as Amazon S3 in your application. Another is to use a driver to construct volumes that can write data to an external storage system, such as Amazon S3 or NFS.

Kubernetes Volumes:

A volume in Kubernetes can be compared to a directory that each pod's containers can access. In Kubernetes, volumes come in several types, and the type determines the content and creation process of the volume. Volumes and Persistent Volumes are the two types of storage abstractions that Kubernetes currently offers.

Differences between Docker and Kubernetes volumes:

Docker also supports the idea of volume. The main drawback was that it was severely restricted to a single pod. A pod's volume was likewise lost when its existence terminated. Similarly, Kubernetes volume exists only when the containing pod is alive. The related volume is also erased when the pod is removed. Kubernetes volumes are therefore helpful for holding transient data that is not required to exist once the pod has finished its lifecycle.

When a pod is deleted, Kubernetes persistent volumes will still be accessible because they are available outside of the pod lifecycle. If needed, another pod may claim it, and the data is kept safe. However, volumes created with Kubernetes are not restricted to any container. It supports all or any of the containers that are set up inside the Kubernetes pod. One of the main benefits of Kubernetes volume is that it allows for the simultaneous usage of many types of storage by the pod.

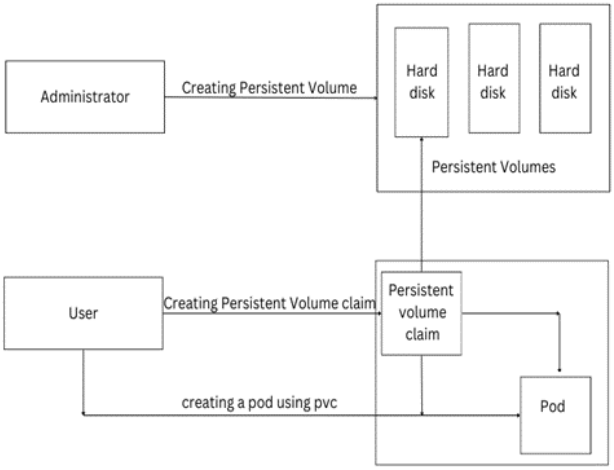
Persistent Volumes:

Administrator-provisioned volumes make up Kubernetes persistent volumes. These are made with a specific filesystem, size, and unique features like names and volume IDs.

A persistent volume in Kubernetes has the following characteristics:

1. It is provided by an administrator or dynamically.
2. Produced using a certain filesystem.
3. Possess a specific size.
4. Has distinguishing features like a name and volume IDs

These volumes must be claimed (through a persistent volume claim) and the claim included in the pod specification for pods to begin using them. A persistent volume claim locates any matching persistent volumes and asserts them, along with describing the quantity and specifications of the storage needed by the pod. Storage classes provide information on the default volume (filesystem, size, block size, etc.). The below diagram explains the process.



Kubernetes PV and PVC Lifecycle:

PVC requests for a PV resource in Kubernetes follow these steps:

1. Provisioning:

The lifespan of a claim and a persistent volume begins with provisioning. Kubernetes accomplishes this in two ways:

First, PVs that represent actual storage are created using static provisioning. These PVs are manually deployed beforehand by cluster managers and are already part of the Kubernetes API. Additionally, since static PVs are a common resource, any user in the Kubernetes cluster can access them.

Using a pre-configured StorageClass specification, real-time PV storage creation is achieved in a second way, called dynamic provisioning. When there are no static PVs available to match the PVC request, this occurs.

2. Attaching:

The PV and PVC are then linked together after an appropriate PV storage that satisfies the PVC request is discovered. Here's how it transpires:.

Initially, the PV's preferred storage capacity and access mode are specified by the user when they initiate a PVC request. A control loop procedure keeps an eye out for any new PVCs in the system. Whether the PV was provisioned statically or dynamically will determine what happens next. The control loop will search for a PV that satisfies the PVC's parameters in the event of a static provision. If PVC has previously been dynamically supplied, the control loop will merely bind them. ClaimRef is used to bind the k8s PVC and PV, resulting in a bi-directional 1:1 mapping of the two objects. Additionally, it implies that PV and PVC are mutually exclusive.

3. Using:

The storage can now be mounted by Kubernetes pods after the PV and PVC have been bound. By treating the PVC like a Kubernetes volume mount, it does this. The PV storage connected to the PVC will then be located by the Kubernetes cluster, which will mount it to the pod as a result. The PV storage in the pod is now used by the user.

4. Reclaiming:

A pod can be erased and released once it has completed utilizing PVC. The persistent volume definition's policy will determine what happens to the bound PV storage. Two options are available.

The retention policy comes first. In this instance, the PV resource remains unbound from the PVC while retaining any data from the preceding pod. Therefore, another PVC cannot delete it and use it. To make this PV usable, the administrator must manually remove it.

Delete is the second policy. This fully eliminates both the PV and the PVC. Moreover, any storage asset from other platforms like Azure Disk, GCE PD, or AWS EBS gets deleted by doing this.

Creating PV:

Below is the samplepv.yaml

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: samplepv
  labels:
    type: local
spec:
  capacity:
    storage: 50Gi
  accessModes:
    - ReadWriteOnce
    hostPath:
      path: "/tmp/data01"
```

```
$ kubectl create –f samplepv.yaml
persistentvolume "samplepv" created
```

Creating Persistent Volume Claim:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: samplepvc
spec:
  accessModes:
```

```
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

$ kubectl create –f samplepvc.yml
persistentvolumeclaim "samplepvc" created

Binding PV and PVC to a Pod:

```
kind: Pod
apiVersion: v1
metadata:
  name: samplepod
  labels:
    name: frontend
spec:
  containers:
  - name: mynginx
    image: nginx
    ports:
    - containerPort: 80
      name: "http-server"
    volumeMounts:
    - mountPath: "/usr/share/tomcat/html"
      name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
      claimName: samplepvc
```

$kubectl create -f samplepod.yml
pod “samplepod” created

Best Practices for Persistent Volume

1. A Guide for Developing PVs:

The creation phase is where a nice example of a Kubernetes persistent volumes begin. Here are two things to consider before carrying out this task:.

a. The request will fail if you do not specify a StorageClass when defining your PVC. Furthermore, make sure the name of your StorageClass specification is meaningful.

b. Never use PVs when configuring containers; instead, always use PVCs. PVCs are necessary because you don't want to tie containers to certain volumes.

2. Whenever possible, make use of dynamic provisioning.

Creating PVs dynamically is desirable from a performance perspective. Scaling the practice of assigning static PVs to PVCs is difficult since it requires additional overhead and resource consumption. Moreover, administrators can use StorageClasses to restrict how much storage can be allocated to a dynamic PVC request.

3. Determine in advance how much storage space your container will require.

To accommodate a range of node sizes, Kubernetes frequently offers alternative storage capacities and sizes. Always estimate the space your container will require, then only request that amount to maximize usage.

4. Set storage usage limits with resource quotas.

You can create resource quotas to restrict how much memory, processing power, and storage containers are allowed to be consumed. These limitations can be applied to every container on a particular namespace, backup, or service level.

5. Include definitions for Quality of Service (QoS).

In certain Kubernetes platforms, quality of service (QoS) is an extra parameter that is included in some PVC requests. By using this, Kubernetes can determine the type of workload and assign the most suitable persistent storage for the given situation.

For instance, QoS can assist Kubernetes in allocating SSD storage for optimal performance when a container necessitates high read/write output.

Conclusion

If we need to keep the data from a container around, we should use Docker volumes, as explained in this post. However, if you are utilizing Kubernetes, Kubernetes volumes are preferable over Docker volumes due to the benefits that Kubernetes offers. If you have both Docker and Kubernetes volumes, the Kubernetes volumes will take precedence over the Docker ones. In the event that you are utilizing both Docker volumes and Kubernetes volumes, it is recommended to use Kubernetes volumes. If the state of an application must be maintained or if application data must be available at all times, volumes should be used in conjunction with the best practices outlined in this document.

References

[1]. https://blog.container-solutions.com/understanding-volumes-docker
[2]. https://www.digitalocean.com/community/tutorials/how-to-work-with-docker-data-volumes-on-ubuntu-14-04
[3]. https://nickjanetakis.com/blog/docker-tip-28-named-volumes-vs-path-based-volumes
[4]. https://medium.com/@nielssj/docker-volumes-and-file-system-permissions-772c1aee23ca
[5]. https://stephenafamo.com/blog/posts/docker-volumes-a-comprehensive-introduction
[6]. https://charlesreid1.com/wiki/Docker/Volumes
[7]. https://docs.docker.com/reference/cli/docker/volume/ls/
[8]. https://docs.docker.com/reference/cli/docker/volume/create/
[9]. https://training.play-with-docker.com/docker-volumes/
[10]. https://www.ibm.com/docs/en/spectrum-symphony/7.3.1?topic=containers-mounting-host-data-volumes-docker-container
[11]. https://docs.docker.com/engine/storage/volumes/
[12]. https://portworx.com/tutorial-kubernetes-persistent-volumes/
[13]. https://kubernetes.io/blog/2018/07/12/resizing-persistent-volumes-using-kubernetes/
[14]. https://jagadesh4java.blogspot.com/2018/05/kubernetes-volumes_23.html
[15]. https://docs.portworx.com/portworx-enterprise/concepts/kubernetes-storage-101/volumes
[16]. https://www.tutorialspoint.com/kubernetes/kubernetes_volumes.htm
[17]. https://akomljen.com/kubernetes-persistent-volumes-with-deployment-and-statefulset/
[18]. https://medium.com/@xcoulon/storing-data-into-persistent-volumes-on-kubernetes-fb155da16666
[19]. https://pwittrock.github.io/docs/concepts/storage/persistent-volumes/
[20]. https://object-storage-ca-ymq-1.vexxhost.net/swift/v1/6e4619c416ff4bd19e1c087f27a43eea/www-assets-prod/presentation-media/OpenStack-Summit-Vancouver-OpenSDS-and-Cinder-for-K8S7.pdf
[21]. https://www.rancher.com/docs/rancher/v1.5/en/kubernetes/storage/