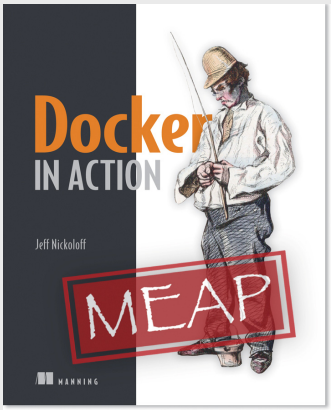

What are Volumes and Their Benefits?

By Jeff Nickoloff

Volumes are ways to declare shared state or stateful data with a life cycle independent of a running program. Since containers are segregated in all other ways, volumes provide a mechanism to specify a scope of interaction through file access. In this article, I talk about volumes and their benefits.

This article is excerpted from Docker in Action by Jeff Nickoloff. Save 39% on Docker in Action with code 15dzamia at manning.com.

Volumes are files or folders that exist outside of a container, but are available to a container through mount points. A container can mount zero or many volumes. Volumes can be shared between containers and the host itself. The life cycle of a volume is independent of any single container. Put simply, volumes make it possible to work with data.

Previously we were limited to working with the file system provided by images. Again, images are made up of layers and are sewn together by something called a union file system. Writes to that file system are written as new layers on top of the existing images. While this type of file system is wonderful for building images it is less than ideal for working with shared or dynamic data.

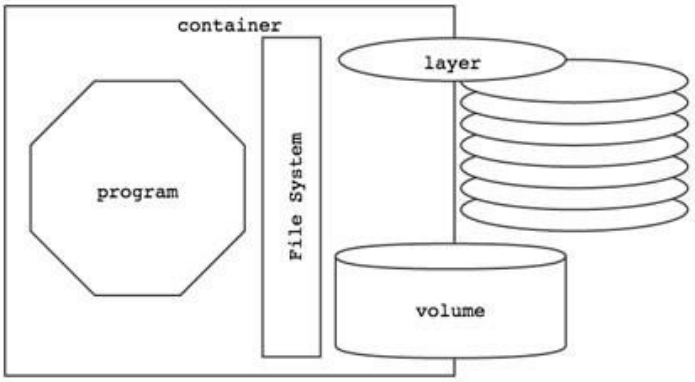

Figure 1 A container with a mounted volume and writeable top layer of the union file system.

When a container is created, its root is set to the root of the file system specified by the image. Any volumes that have been specified will be created on the host file system in Docker-managed space and then mounted onto the container's root partition. This is called a bindmount. If the volume is mapped to an existing file or directory on the host, or the volume is being shared with some other container, volume creation will be skipped and the existing volume will simply be mounted in the container.

Benefit of Volumes

The benefit of using volumes is that they enable the separation of concerns. Semantically, volumes are ways to declare shared state or stateful data with a life cycle independent of a running program. Since containers are segregated in all other ways, volumes provide a mechanism to specify a scope of interaction through file access. Think about it this way: if images hold tools, volumes hold input and output. The distinction makes tools reusable and the input and output things that you can easily share and protect.

The separation of relatively static and stateful memory allows application or image authors to implement polymorphic and composeable tools.

A polymorphic tool is one that you interact with in a consistent way, but might have several implementations that do different things. Consider an application like a general application server. Apache Tomcat for example is an application that provides an HTTP interface on a network, and dispatches any requests it receives to pluggable programs. Tomcat has polymorphic behavior. Using volumes you can similarly inject behavior in containers without modifying an image.

A composeable tool is one that provides mechanisms to interact with other tools. Programs in containers are isolated by default but volumes break that isolation. When multiple containers share access to the same file system location they can be composed into more sophisticated systems.

The separation of concerns does not stop with application or architecture design. More fundamentally volumes enable the separation of application and host concerns. At some point an image is loaded onto a host and a container is built. While Docker can make assertions about what files should be available to a container Docker knows little about the host where it is running. That means that Docker alone has no way to take advantage host specific facilities like mounted network storage or mixed spinning and solid state hard drives. However, a user with knowledge of the host can use volumes to map directories in a container to appropriate storage on that host.

Like most other aspects of working with Docker and containers, volumes are actually functionality that is provided by your host operating system. As such your software will incur only a minimal performance impact when reader from or writing to disk.

Sharing with the Host

Every volume is by definition, "shared with the host." But there are two distinct types of volumes that differ in how they are shared with the host. The first are Docker managed volumes. These are host directories created by the Docker daemon, in space controlled by the daemon. The second are bind-mounts. A bind mount is used to mount part of an already mounted file system at a different location.

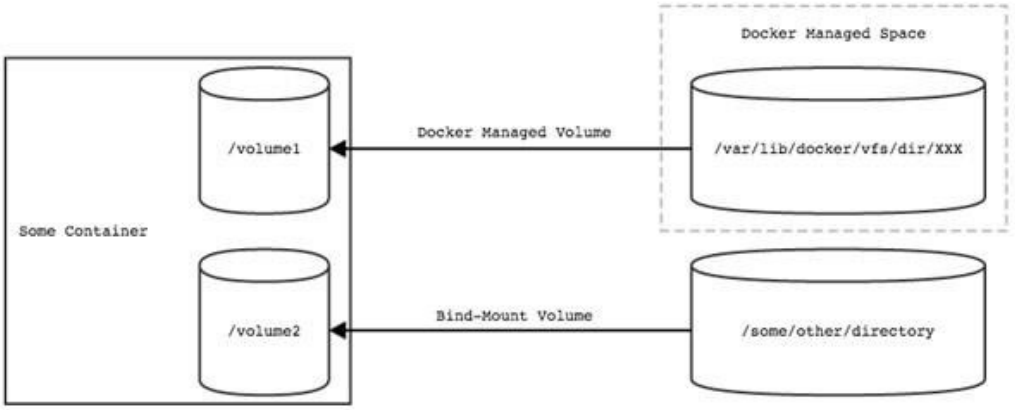

Figure 2 Docker uses two types of volumes.

Docker Manage Volumes

All volumes are "shared with the host" but Docker Managed volumes are created by the

Docker daemon. You already used this type of volume when you ran the command in Listing 1. The container you created is named `listing.1`, and the volume you created is mounted on the container's file system at /workspace. When you created this container, the Docker daemon created a directory to store the contents of that volume somewhere in a part of the host file system that it controls. To find out exactly where this folder is, you can type `docker inspect listing.1` and look for the "Volumes" key. You'll see something like this:

```
...
"Volumes": { "/workspace":
"/mnt/sda1/var/lib/docker/vfs/dir/7e2edb4a1ddd7b2cddd60b1ac104d5d6469e9df8449389
      bd a047b2fce0c031ca"
},
...
```

The "Volumes" key points to a value that is itself a map. In this map each key is a mount point in the container and the value is the location of the directory on the host file system. In this case, we've inspected a container with only a single volume, and so there is only a single entry in the map. If you had specified two volumes when the container was created there would be two entries.

Listing 2: Creating and inspecting a container with multiple volumes4.2: Creating and inspecting a container with multiple volumes

```
docker run --name listing.2 \
        -v /volume1 -v /volume2 -v /anotherVolume \ busybox:latest \
        echo "This container has multiple Docker managed volumes."
docker inpect listing.2
...
"Volumes": {
        "/anotherVolume":

      "/mnt/sda1/var/lib/docker/vfs/dir/c98056a56e40d3ae7b80bf7da81ecffcaadc70f58802a
      3a2908e24f58a9c30a2",
        "/volume1":

      "/mnt/sda1/var/lib/docker/vfs/dir/6f92a91b594bed84ebe3a6d2c91308681a3ba3c3c77b4
aab5b8c115e6fe969c1",
        "/volume2":

      "/mnt/sda1/var/lib/docker/vfs/dir/262f169812956b3b6015478f5e7a4555816dd036fa404
        1517678096e9304b47a"
        },
        ...
```

You can see from the output (in Listing 2) that the map is enumerated by the lexicographical ordering of its keys and independent of the ordering specified when the container is created. While this is true for this example, this is not behavior specified by any interface contract and should not be used as the basis for further development.

Another thing for boot2docker users to keep in mind is that the host path specified in each value is relative to their boot2docker's virtual machine root, not the root of their host. If you want to work with those files directly, you will need to first access a shell on that virtual machine.

The important thing to take away from this output is that each of the volumes was created in a directory controlled by the Docker daemon.

If you have not already guessed as much, the `-v` flag on the `docker run` command is the way you create volumes. What follows the `-v` flag is the location where you want to mount the volume on the container file system.

Docker managed volumes may seem difficult to work with if you are manually building or linking tools together on your desktop, but in larger systems where specific locality of the data is less important these are a much more effective way to organize your data. The primary

reason for this is that their use is decoupled from other potential concerns of the system. By using Docker managed volumes you are simply stating, "I need a place to put some data that I'm working with." This is a requirement that Docker can fill on any machine where Docker is installed. Further, when you are finished with a volume and you ask Docker to clean things up for you, Docker can confidently remove any directories or files that are no longer being used by a container. Using volumes in this way helps manage clutter.

Bind Mount Volumes

A bind mount is a location where part of an already mounted file system has been remounted. In a Docker volume context, the part of the file system being mounted is some host directory, and the different location is on the file system inside of the new container.

Figure 3: A Host Directory as a Bind Mounted Volume

Run the following to see bind mount volumes in action.

Listing 3: A Host directory as a Bind Mounted Volumesting 4.3: Creating a Container with a Bind Mount Directory Volume

```
# create a host
directory mkdir
./listing.3


# bind mount the directory and create an empty
file docker run --name listing.3

        -v "$(pwd)"/listing.4.3:/workspace
busybox:latest touch /workspace/evidence


# check out the
result ls
./listing.3
```

In this listing you created a new directory to be mounted as a volume in a new container named listing 3. The command to be run in the container creates an empty file in the mounted volume. After the program runs the container will stop and you can see the file created in the directory you created.

In this listing you used the `-v` option but what followed was a map between two file system locations. The map delimits key-value pairs with a single as is common with Linux style command line tools. The map key is the host file system location, and the value is the location where it should be mounted inside of the container. It is important to note that these locations have to be specified using absolute paths. This was accomplished in Listing 3 by substituting the present working directory in the command. If this did not work on your machine, try replacing `"$(pwd)"` with the path of your working directory.

You might want to use bind mount volumes to coordinated or share work with other containers or processes running outside of a container. You might want to integrate with components of the host system itself. For example, suppose Paul wanted to launch a web server to serve content from the "www" directory under his home directory. He might run a command like this, which mounts that directory as a volume in a very specific location in the container:

```
docker run --name pauls_web \
        -v /home/paul/www:/usr/local/apache2/htdocs/ \
        httpd:latest
```

Of course unless "/home/paul/www" exists on your machine, you will not be able to run that command.

This example does touch on an important attribute or feature of volumes. When a volume is mounted on a container file system, it replaces whatever was provided by the image at that location. In this example, the httpd:latest image provides some default HTML content at "/usr/local/apache2/htdocs/" but when Paul creates a volume to be mounted at that location, that default content is lost and replaced with the directory being mounted. This behavior is the basis for the Polymorphic Container Pattern.

Expanding on Paul's use-case, suppose he wanted to make sure that the Apache HTTPD web server could only read the contents of this volume. He know that even the most trusted software might contain vulnerabilities and he'd rather not discover those first hand when his content was defaced. Luckily, Docker provides a mechanism to mount volumes as read-only. You can do this by appending ":ro" to the volume map specification. For example, Paul would want to change his run command to something like the following.

```
docker run --name pauls_web_read \
        -v /home/paul/www:/usr/local/apache2/htdocs/:ro \
        httpd:latest
```

Doing this he can trust that his operating system will do what it can to prevent writes to the directory he's mounted as a volume from anything inside this container.

Finally, note that if you specify a host directory that does not exist, Docker will create it for you. While this can come in handy, relying on this functionality is not the best idea. Generally it is better to have more specific control over the ownership and permissions set on a directory that you are going to be using as a volume.

Bind mounted volumes are not limited to directories, though that is how they are frequently used. You can use bind mount volumes to mount individual files. This provides the flexibility to create or link resources at a level that avoids conflict with other resources. One such case is when you want to mount a specific file into a directory that contains other unrelated files. Take a case where an application is configured to write logs to a file that is in the same directory as other configuration. If you were to bind mount a whole directory over that location, those other files would be lost. By using a specific file as a volume you can override only the file you need.\

Listing 4: Example of a bind mounted filexample of a bind mounted file

```
# create a file to
mount ls -l >
directoryListing.text


# mount and view the file from a
container docker run --rm \
        -v "$(pwd)"/directoryListing.txt:/some/path/listing \
busybox:latest cat /some/path/listing
```

The important thing to note in case is that the file must exist on the host before you create the container. If it does not, Docker will assume that you wanted to use a directory, create it on the host, and mount it at the desired location (even if that location is occupied by a file).