DigitalOcean

Subscribe

How To Share Data between Docker Containers

21

Updated July 11, 2018 163.1k DOCKER UBUNTU UBUNTU 18.04

By: Melissa Anderson

Introduction

Docker is a popular containerization tool used to provide software applications with a filesystem that contains everything they need to run. Using Docker containers ensures that the software will behave the same way regardless of where it is deployed because its run-time environment is consistent.

In general, Docker containers are ephemeral, running just as long as it takes for the command issued in the container to complete. Sometimes, however, applications need to share access to data or persist data after a container is deleted. Databases, user-generated content for a web site, and log files are just a few examples of data that is impractical or impossible to include in a Docker image but which applications need to access. Persistent access to data is provided with Docker Volumes.

Docker Volumes can be created and attached in the same command that creates a container, or they can be created independently of any containers and attached later. In this article, we'll look at four different ways to share data between containers.

Prerequisites

To follow this article, you will need an Ubuntu 18.04 server with the following:

- A non-root user with sudo privileges. The Initial Server Setup with Ubuntu 18.04 guide explains how to set this up.
- Docker installed with the instructions from **Step 1** and **Step 2** of How To Install and Use Docker on Ubuntu 18.04

Note: Even though the Prerequisites give instructions for installing Docker on Ubuntu 18.04, the `docker` commands for Docker data volumes in this article should work on other operating systems as long as Docker is installed and the sudo user has been added to the `docker` group.

Step 1 — Creating an Independent Volume

Introduced in Docker's 1.9 release, the `docker volume create` command allows you to create a volume without relating it to any particular container. We'll use this command to add a volume named `DataVolume1`:

```
$ docker volume create --name DataVolume1
```

The name is displayed, indicating that the command was successful:

```
Output
DataVolume1
```

To make use of the volume, we'll create a new container from the Ubuntu image, using the `--rm` flag to automatically delete it when we exit. We'll also use `-v` to mount the new volume. `-v` requires the name of the volume, a colon, then the absolute path to where the volume should appear inside the container. If the directories in the path don't exist as part of the image, they'll be created when the command runs. If they *do* exist, the mounted volume will hide the existing content:

```
$ docker run -ti --rm -v DataVolume1:/datavolume1 ubuntu
```

While in the container, let's write some data to the volume:

```
root@802b0a78f2ef:/# echo "Example1" > /datavolume1/Example1.txt
```

Because we used the `--rm` flag, our container will be automatically deleted when we exit. Our volume, however, will still be accessible.

```
root@802b0a78f2ef:/# exit
```

We can verify that the volume is present on our system with `docker volume inspect`:

```
$ docker volume inspect DataVolume1
```

```
Output
[
    {
        "CreatedAt": "2018-07-11T16:57:54Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/DataVolume1/_data",
        "Name": "DataVolume1",
        "Options": {},

        "Scope": "local"
    }
]
```

Note: We can even look at the data on the host at the path listed as the `Mountpoint`. We should avoid altering it, however, as it can cause data corruption if applications or containers are unaware of changes.

Next, let's start a new container and attach `DataVolume1`:

```
$ docker run --rm -ti -v DataVolume1:/datavolume1 ubuntu
```

Verify the contents:

```
root@d73eca0365fc:/# cat /datavolume1/Example1.txt
```

```
Output
Example1
```

Exit the container:

```
root@d73eca0365fc:/# exit
```

In this example, we created a volume, attached it to a container, and verified its persistence.

Step 2 — Creating a Volume that Persists when the Container is Removed

In our next example, we'll create a volume at the same time as the container, delete the container, then attach the volume to a new container.

We'll use the `docker run` command to create a new container using the base Ubuntu image. `-t` will give us a terminal, and `-i` will allow us to interact with it. For clarity, we'll use `--name` to identify the container.

The `-v` flag will allow us to create a new volume, which we'll call `DataVolume2`. We'll use a colon to separate this name from the path where the volume should be mounted in the container. Finally, we will specify the base Ubuntu image and rely on the default command in the Ubuntu base image's Docker file, `bash`, to drop us into a shell:

```
$ docker run -ti --name=Container2 -v DataVolume2:/datavolume2 ubuntu
```

Note: The `-v` flag is very flexible. It can bindmount or name a volume with just a slight adjustment in syntax. If the first argument begins with a `/` or `~/` you're creating a bindmount. Remove that, and you're naming the volume. For example:

- `-v /path:/path/in/container` mounts the host directory, `/path` at the `/path/in/container`
- `-v path:/path/in/container` creates a volume named `path` with no relationship to the host.

For more on bindmounting a directory from the host, see How To Share Data between a Docker Container and the Host

While in the container, we'll write some data to the volume:

```
root@87c33b5ae18a:/# echo "Example2" > /datavolume2/Example2.txt
root@87c33b5ae18a:/# cat /datavolume2/Example2.txt
```

```
Output
Example2
```

Let's exit the container:

```
root@87c33b5ae18a:/# exit
```

When we restart the container, the volume will mount automatically:

```
$ docker start -ai Container2
```

Let's verify that the volume has indeed mounted and our data is still in place:

```
root@87c33b5ae18a:/# cat /datavolume2/Example2.txt
```

```
Output
Example2
```

Finally, let's exit and clean up:

```
root@87c33b5ae18a:/# exit
```

Docker won't let us remove a volume if it's referenced by a container. Let's see what happens when we try:

```
$ docker volume rm DataVolume2
```

The message tells us that the volume is still in use and supplies the long version of the container ID:

```
Output
Error response from daemon: unable to remove volume: remove DataVolume2: volume is in use -
```

We can use this ID to remove the container:

```
$ docker rm d0d2233b668eddad4986313c7a4a1bc0d2edaf0c7e1c02a6a6256de27db17a63
```

```
Output
```

```
d0d2233b668eddad4986313c7a4a1bc0d2edaf0c7e1c02a6a6256de27db17a63
```

Removing the container won't affect the volume. We can see it's still present on the system by listing the volumes with `docker volume ls`:

```
$ docker volume ls
```

```
Output
DRIVER              VOLUME NAME
local               DataVolume2
```

And we can use `docker volume rm` to remove it:

```
$ docker volume rm DataVolume2
```

In this example, we created an empty data volume at the same time that we created a container. In our next example, we'll explore what happens when we create a volume with a container directory that already contains data.

Step 3 — Creating a Volume from an Existing Directory with Data

Generally, creating a volume independently with `docker volume create` and creating one while creating a container are equivalent, with one exception. If we create a volume at the same time that we create a container *and* we provide the path to a directory that contains data in the base image, that data will be copied into the volume.

As an example, we'll create a container and add the data volume at `/var`, a directory which contains data in the base image:

```
$ docker run -ti --rm -v DataVolume3:/var ubuntu
```

All the content from the base image's `/var` directory is copied into the volume, and we can mount that volume in a new container.

Exit the current container:

```
root@87c33b5ae18a:/# exit
```

This time, rather than relying on the base image's default `bash` command, we'll issue our own `ls` command, which will show the contents of the volume without entering the shell:

```
$ docker run --rm -v DataVolume3:/datavolume3 ubuntu ls datavolume3
```

The directory `datavolume3` now has a copy of the contents of the base image's `/var` directory:

```
Output
backups
cache
lib
local
lock
log
mail
opt
run
spool
tmp
```

It's unlikely that we would want to mount `/var/` in this way, but this can be helpful if we've crafted our own image and want an easy way to preserve data. In our next example, we'll demonstrate how a volume can be shared between multiple containers.

Step 4 — Sharing Data Between Multiple Docker Containers

So far, we've attached a volume to one container at a time. Often, we'll want multiple containers to attach to the same data volume. This is relatively straightforward to accomplish, but there's one critical caveat: at this time, Docker doesn't handle file locking. If you need multiple containers writing to the volume, the applications running in those containers *must* be designed to write to shared data stores in order to prevent data corruption.

Create Container4 and DataVolume4

Use `docker run` to create a new container named `Container4` with a data volume attached:

```
$ docker run -ti --name=Container4 -v DataVolume4:/datavolume4 ubuntu
```

Next we'll create a file and add some text:

```
root@db6aaead532b:/# echo "This file is shared between containers" > /datavolume4/Example4
```

Then, we'll exit the container:

```
root@db6aaead532b:/# exit
```

This returns us to the host command prompt, where we'll make a new container that mounts the data volume from `Container4`.

Create Container5 and Mount Volumes from Container4

We're going to create `Container5`, and mount the volumes from `Container4`:

```
$ docker run -ti --name=Container5 --volumes-from Container4 ubuntu
```

Let's check the data persistence:

```
root@81e7a6153d28:/# cat /datavolume4/Example4.txt
```

```
Output
This file is shared between containers
```

Now let's append some text from `Container5`:

```
root@81e7a6153d28:/# echo "Both containers can write to DataVolume4" >> /datavolume4/Examp
```

Finally, we'll exit the container:

```
root@81e7a6153d28:/# exit
```

Next, we'll check that our data is still present to `Container4`.

View Changes Made in Container5

Let's check for the changes that were written to the data volume by `Container5` by restarting `Container4`:

```
$ docker start -ai Container4
```

Check for the changes:

```
root@db6aaead532b:/# cat /datavolume4/Example4.txt
```

```
Output
This file is shared between containers
Both containers can write to DataVolume4
```

Now that we've verified that both containers were able to read and write from the data volume, we'll exit the container:

```
root@db6aaead532b:/# exit
```

Again, Docker doesn't handle any file locking, so applications *must* account for the file locking themselves. It is possible to mount a Docker volume as read-only to ensure that data corruption won't happen by accident when a container requires read-only access by adding `:ro`. Let's look at how this works.

Start Container 6 and Mount the Volume Read-Only

Once a volume has been mounted in a container, rather than unmounting it like we would with a typical Linux file system, we can instead create a new container mounted the way we want and, if needed, remove the previous container. To make the volume read-only, we append `:ro` to the end of the container name:

```
$ docker run -ti --name=Container6 --volumes-from Container4:ro ubuntu
```

We'll check the read-only status by trying to remove our example file:

```
root@81e7a6153d28:/# rm /datavolume4/Example4.txt
```

```
Output
rm: cannot remove '/datavolume4/Example4.txt': Read-only file system
```

Finally, we'll exit the container and clean up our test containers and volumes:

```
root@81e7a6153d28:/# exit
```

Now that we're done, let's clean up our containers and volume:

```
$ docker rm Container4 Container5 Container6
$ docker volume rm DataVolume4
```

In this example, we've shown how to share data between two containers using a data volume and how to mount a data volume as read-only.

Conclusion

In this tutorial, we created a data volume which allowed data to persist through the deletion of a container. We shared data volumes between containers, with the caveat that applications will need to be designed to handle file locking to prevent data corruption. Finally, we showed how to mount a shared volume in read-only mode. If you're interested in learning about sharing data between containers and the host system, see How To Share Data between the Docker Container and the Host.

By: Melissa Anderson

Related Tutorials

How To Install Docker Compose on Ubuntu 18.04

How To Remove Docker Images, Containers, and Volumes

Naming Docker Containers: 3 Tips for Beginners

How to Use Traefik as a Reverse Proxy for Docker Containers on Ubuntu 18.04

How To Provision and Manage Remote Docker Hosts with Docker Machine on Ubuntu 18.04

5 Comments

Leave a comment...

Log In to Comment

R0d *April 1, 2017*

0 Hi Melissa - thanks for the fantastic intro to Docker Volumes.
Is it possible to present a DO Volume to to the Docker Daemon so images and containers can be stored in the DO volume instead of instance storage.?
Thanks,
Rod

R0d *April 1, 2017*

0 Answer:
https://forums.docker.com/t/how-do-i-change-the-docker-image-installation-directory/1169

ramanjaneyulu *September 27, 2017*

0 Will this work for Docker containers which are running in the docker-swarm cluster?

shivakp *September 28, 2017*

1 Hi Melissa - Nice explanation thank you.

Please correct me if I am wrong, In the "3 — Creating a Volume from an Existing Directory with Data"
the second command: #docker run --rm -v DataVolume3:/datavolume3 ubuntu ls DataVolume3

the last argument after image name should be "datavolume3" not "DataVolume3"

krishnamohant *November 21, 2017*

0 In example 3, docker run -ti --rm -v DataVolume3:/var ubuntu, the arguments (DataVolume3 and /var) are in reverse order, it works this way - "docker run -ti --rm -v /var:DataVolume3 ubuntu". directory in the host machine comes before the directory that gets created in the container.