**DOCKER NOTES JOEL**

Docker is a platform for developers and sysadmins to **build, share, and run** applications with containers. The use of containers to deploy applications is called *containerization*. Containers are not new, but their use for easily deploying applications is.

Containerization is increasingly popular because containers are:

* Flexible: Even the most complex applications can be containerized.
* Lightweight: Containers leverage and share the host kernel, making them much more efficient in terms of system resources than virtual machines.
* Portable: You can build locally, deploy to the cloud, and run anywhere.
* Loosely coupled: Containers are highly self sufficient and encapsulated, allowing you to replace or upgrade one without disrupting others.
* Scalable: You can increase and automatically distribute container replicas across a datacenter.
* Secure: Containers apply aggressive constraints and isolations to processes without any configuration required on the part of the user.

 An image includes everything needed to run an application -- the code or binary, runtimes, dependencies, and any other filesystem objects required.

Tools to manage, scale, and maintain containerized applications are called orchestrators, and the most common examples of these are Kubernetes and Docker Swarm.

Installing Docker

sudo yum install -y docker OR

sudo yum install docker

curl -sSL https://get.docker.com/ | sh

AFTER RUN

Sudo service docker start or sudo systemctl restart sshd

**Containers**

[Your basic isolated Docker process](http://etherealmind.com/basics-docker-containers-hypervisors-coreos/). Containers are to Virtual Machines as threads are to processes. Or you can think of them as chroots on steroids.

**Lifecycle**

* [docker create](https://docs.docker.com/engine/reference/commandline/create) creates a container but does not start it.
* [docker rename](https://docs.docker.com/engine/reference/commandline/rename/) allows the container to be renamed.
* [docker run](https://docs.docker.com/engine/reference/commandline/run) creates and starts a container in one operation.
* [docker rm](https://docs.docker.com/engine/reference/commandline/rm) deletes a container.
* [docker update](https://docs.docker.com/engine/reference/commandline/update/) updates a container's resource limits.

Normally if you run a container without options it will start and stop immediately, if you want keep it running you can use the command, docker run -td container\_id this will use the option -t that will allocate a pseudo-TTY session and -d that will detach automatically the container (run container in background and print container ID).

If you want a transient container, docker run --rm will remove the container after it stops.

Another useful option command is:

docker run --name yourname docker\_image

when you specify the --name inside the run command this will allow you to start and stop a container by calling it with the name the you specified when you created it.

**Starting and Stopping**

* [docker start](https://docs.docker.com/engine/reference/commandline/start) starts a container so it is running.
* [docker stop](https://docs.docker.com/engine/reference/commandline/stop) stops a running container.
* [docker restart](https://docs.docker.com/engine/reference/commandline/restart) stops and starts a container.
* [docker pause](https://docs.docker.com/engine/reference/commandline/pause/) pauses a running container, "freezing" it in place.
* [docker unpause](https://docs.docker.com/engine/reference/commandline/unpause/) will unpause a running container.
* [docker wait](https://docs.docker.com/engine/reference/commandline/wait) blocks until running container stops.
* [docker kill](https://docs.docker.com/engine/reference/commandline/kill) sends a SIGKILL to a running container.
* [docker attach](https://docs.docker.com/engine/reference/commandline/attach) will connect to a running container.

If you want to detach from a running container, use Ctrl + p, Ctrl + q. If you want to integrate a container with a [host process manager](https://docs.docker.com/engine/admin/host_integration/), start the daemon with -r=false then use docker start -a.

**Images**

Images are just [templates for docker containers](https://docs.docker.com/engine/understanding-docker/#how-does-a-docker-image-work).

**Lifecycle**

* [docker images](https://docs.docker.com/engine/reference/commandline/images) shows all images.
* [docker import](https://docs.docker.com/engine/reference/commandline/import) creates an image from a tarball.
* [docker build](https://docs.docker.com/engine/reference/commandline/build) creates image from Dockerfile.
* [docker commit](https://docs.docker.com/engine/reference/commandline/commit) creates image from a container, pausing it temporarily if it is running.
* [docker rmi](https://docs.docker.com/engine/reference/commandline/rmi) removes an image.
* [docker load](https://docs.docker.com/engine/reference/commandline/load) loads an image from a tar archive as STDIN, including images and tags (as of 0.7).
* [docker save](https://docs.docker.com/engine/reference/commandline/save) saves an image to a tar archive stream to STDOUT with all parent layers, tags & versions (as of 0.7).

**Info**

* [docker history](https://docs.docker.com/engine/reference/commandline/history) shows history of image.
* [docker tag](https://docs.docker.com/engine/reference/commandline/tag) tags an image to a name (local or registry).

**Checking Docker Version**

It is very important that you always know the current version of Docker you are currently running on at any point in time.This is very helpful because you get to know what features are compatible with what you have running. This is also important because you know what containers to run from the docker store when you are trying to get template containers. That said let see how to know what version of docker we have running currently

* ['docker version'](https://docs.docker.com/engine/reference/commandline/version/) check what version of docker you have running

# Get the server version

docker version --format '{{.Server.Version}}'

In Docker 1.8.0 and higher, you can also dump the raw JSON data:

docker version --format '{{json .}}'

will provide the output in JSON format:

{"Client":{"Version":"1.8.0","ApiVersion":"1.20","GitCommit":"f5bae0a","GoVersion":"go1.4.2","Os":"linux","Arch":"am"}

**Cleaning up**

While you can use the docker rmi command to remove specific images, there's a tool called [docker-gc](https://github.com/spotify/docker-gc) that will safely clean up images that are no longer used by any containers. As of docker 1.13, docker image prune is also available for removing unused images. See [Prune](https://github.com/joelmasham/docker-cheat-sheet#prune).

**Load/Save image**

Load an image from file:

docker load < my\_image.tar.gz

Save an existing image:

docker save my\_image:my\_tag | gzip > my\_image.tar.gz

**Import/Export container**

Import a container as an image from file:

cat my\_container.tar.gz | docker import - my\_image:my\_tag

Export an existing container:

docker export my\_container | gzip > my\_container.tar.gz

**Difference between loading a saved image and importing an exported container as an image**

Loading an image using the load command creates a new image including its history.  
Importing a container as an image using the import command creates a new image excluding the history which results in a smaller image size compared to loading an image.

**PROJECT CREATING A CUSTOMED IMAGE**

Task: To create a docker image and run apache into

* 1. First we have to create a docker file

$ touch dockerfile

* 1. Second step we have to input commands to run apache inside the dockerfile

$ vi dockerfile

**FROM centos:latest**

**RUN Yum update -y**

**RUN Yum install httpd -y**

* 1. Create the image and container using single command

$ docker build -t “imagename” . (note the . is needed at the end of the cmd)

NOTE: to delete image we use

$ docker rmi Iimagename”

* 1. Check to see if the image is created

$ docker image

* 1. Check to see the container details

$ docker ps -a

* 1. To change the image name

$ docker run it –name “newimagename” “imageid”

* 1. Check apache is installed

$httpd -v or $ httpd version (--version)

To pull image from docker hub use

$ docker pull “imagename”

# **Overview of Docker Compose** <https://docs.docker.com/compose/>

*Estimated reading time: 5 minutes*

**Looking for Compose file reference?** [Find the latest version here](https://docs.docker.com/compose/compose-file/).

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration. To learn more about all the features of Compose, see [the list of features](https://docs.docker.com/compose/#features).

Compose works in all environments: production, staging, development, testing, as well as CI workflows. You can learn more about each case in [Common Use Cases](https://docs.docker.com/compose/#common-use-cases).

Using Compose is basically a three-step process:

1. Define your app’s environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Run docker-compose up and Compose starts and runs your entire app.

A docker-compose.yml looks like this:

version: '3'

services:

web:

build: .

ports:

- "5000:5000"

volumes:

- .:/code

- logvolume01:/var/log

links:

- redis

redis:

image: redis

volumes:

logvolume01: {}

For more information about the Compose file, see the [Compose file reference](https://docs.docker.com/compose/compose-file/).

Compose has commands for managing the whole lifecycle of your application:

* Start, stop, and rebuild services
* View the status of running services
* Stream the log output of running services
* Run a one-off command on a service

## Compose documentation

* [Installing Compose](https://docs.docker.com/compose/install/)
* [Getting started with Compose](https://docs.docker.com/compose/gettingstarted/)
* [Get started with Django](https://docs.docker.com/compose/django/)
* [Get started with Rails](https://docs.docker.com/compose/rails/)
* [Get started with WordPress](https://docs.docker.com/compose/wordpress/)
* [Frequently asked questions](https://docs.docker.com/compose/faq/)
* [Command line reference](https://docs.docker.com/compose/reference/)
* [Compose file reference](https://docs.docker.com/compose/compose-file/)

## Features

The features of Compose that make it effective are:

* [Multiple isolated environments on a single host](https://docs.docker.com/compose/#multiple-isolated-environments-on-a-single-host)
* [Preserve volume data when containers are created](https://docs.docker.com/compose/#preserve-volume-data-when-containers-are-created)
* [Only recreate containers that have changed](https://docs.docker.com/compose/#only-recreate-containers-that-have-changed)
* [Variables and moving a composition between environments](https://docs.docker.com/compose/#variables-and-moving-a-composition-between-environments)

### Multiple isolated environments on a single host

Compose uses a project name to isolate environments from each other. You can make use of this project name in several different contexts:

* on a dev host, to create multiple copies of a single environment, such as when you want to run a stable copy for each feature branch of a project
* on a CI server, to keep builds from interfering with each other, you can set the project name to a unique build number
* on a shared host or dev host, to prevent different projects, which may use the same service names, from interfering with each other

The default project name is the basename of the project directory. You can set a custom project name by using the [-p command line option](https://docs.docker.com/compose/reference/overview/) or the [COMPOSE\_PROJECT\_NAME environment variable](https://docs.docker.com/compose/reference/envvars/#compose-project-name).

### Preserve volume data when containers are created

Compose preserves all volumes used by your services. When docker-compose up runs, if it finds any containers from previous runs, it copies the volumes from the old container to the new container. This process ensures that any data you’ve created in volumes isn’t lost.

If you use docker-compose on a Windows machine, see [Environment variables](https://docs.docker.com/compose/reference/envvars/) and adjust the necessary environment variables for your specific needs.

### Only recreate containers that have changed

Compose caches the configuration used to create a container. When you restart a service that has not changed, Compose re-uses the existing containers. Re-using containers means that you can make changes to your environment very quickly.

### Variables and moving a composition between environments

Compose supports variables in the Compose file. You can use these variables to customize your composition for different environments, or different users. See [Variable substitution](https://docs.docker.com/compose/compose-file/#variable-substitution) for more details.

You can extend a Compose file using the extends field or by creating multiple Compose files. See [extends](https://docs.docker.com/compose/extends/) for more details.

## Common use cases

Compose can be used in many different ways. Some common use cases are outlined below.

### Development environments

When you’re developing software, the ability to run an application in an isolated environment and interact with it is crucial. The Compose command line tool can be used to create the environment and interact with it.

The [Compose file](https://docs.docker.com/compose/compose-file/) provides a way to document and configure all of the application’s service dependencies (databases, queues, caches, web service APIs, etc). Using the Compose command line tool you can create and start one or more containers for each dependency with a single command (docker-compose up).

Together, these features provide a convenient way for developers to get started on a project. Compose can reduce a multi-page “developer getting started guide” to a single machine readable Compose file and a few commands.

### Automated testing environments

An important part of any Continuous Deployment or Continuous Integration process is the automated test suite. Automated end-to-end testing requires an environment in which to run tests. Compose provides a convenient way to create and destroy isolated testing environments for your test suite. By defining the full environment in a [Compose file](https://docs.docker.com/compose/compose-file/), you can create and destroy these environments in just a few commands:

$ docker-compose up -d

$ ./run\_tests

$ docker-compose down

### Single host deployments

Compose has traditionally been focused on development and testing workflows, but with each release we’re making progress on more production-oriented features. You can use Compose to deploy to a remote Docker Engine. The Docker Engine may be a single instance provisioned with [Docker Machine](https://docs.docker.com/machine/overview/) or an entire [Docker Swarm](https://docs.docker.com/engine/swarm/) cluster.

For details on using production-oriented features, see [compose in production](https://docs.docker.com/compose/production/) in this documentation.

# **Use volumes**

*Estimated reading time: 16 minutes*

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While [bind mounts](https://docs.docker.com/storage/bind-mounts/) are dependent on the directory structure of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

* Volumes are easier to back up or migrate than bind mounts.
* You can manage volumes using Docker CLI commands or the Docker API.
* Volumes work on both Linux and Windows containers.
* Volumes can be more safely shared among multiple containers.
* Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
* New volumes can have their content pre-populated by a container.

In addition, volumes are often a better choice than persisting data in a container’s writable layer, because a volume does not increase the size of the containers using it, and the volume’s contents exist outside the lifecycle of a given container.

If your container generates non-persistent state data, consider using a [tmpfs mount](https://docs.docker.com/storage/tmpfs/) to avoid storing the data anywhere permanently, and to increase the container’s performance by avoiding writing into the container’s writable layer.

Volumes use rprivate bind propagation, and bind propagation is not configurable for volumes.

## Choose the -v or --mount flag

Originally, the -v or --volume flag was used for standalone containers and the --mount flag was used for swarm services. However, starting with Docker 17.06, you can also use --mount with standalone containers. In general, --mount is more explicit and verbose. The biggest difference is that the -v syntax combines all the options together in one field, while the --mount syntax separates them. Here is a comparison of the syntax for each flag.

New users should try --mount syntax which is simpler than --volume syntax.

If you need to specify volume driver options, you must use --mount.

* **-v or --volume**: Consists of three fields, separated by colon characters (:). The fields must be in the correct order, and the meaning of each field is not immediately obvious.
  + In the case of named volumes, the first field is the name of the volume, and is unique on a given host machine. For anonymous volumes, the first field is omitted.
  + The second field is the path where the file or directory are mounted in the container.
  + The third field is optional, and is a comma-separated list of options, such as ro. These options are discussed below.
* **--mount**: Consists of multiple key-value pairs, separated by commas and each consisting of a <key>=<value> tuple. The --mount syntax is more verbose than -v or --volume, but the order of the keys is not significant, and the value of the flag is easier to understand.
  + The type of the mount, which can be [bind](https://docs.docker.com/storage/bind-mounts/), volume, or [tmpfs](https://docs.docker.com/storage/tmpfs/). This topic discusses volumes, so the type is always volume.
  + The source of the mount. For named volumes, this is the name of the volume. For anonymous volumes, this field is omitted. May be specified as source or src.
  + The destination takes as its value the path where the file or directory is mounted in the container. May be specified as destination, dst, or target.
  + The readonly option, if present, causes the bind mount to be [mounted into the container as read-only](https://docs.docker.com/storage/volumes/#use-a-read-only-volume).
  + The volume-opt option, which can be specified more than once, takes a key-value pair consisting of the option name and its value.

**Escape values from outer CSV parser**

If your volume driver accepts a comma-separated list as an option, you must escape the value from the outer CSV parser. To escape a volume-opt, surround it with double quotes (") and surround the entire mount parameter with single quotes (').

For example, the local driver accepts mount options as a comma-separated list in the o parameter. This example shows the correct way to escape the list.

$ docker service create \

--mount 'type=volume,src=<VOLUME-NAME>,dst=<CONTAINER-PATH>,volume-driver=local,volume-opt=type=nfs,volume-opt=device=<nfs-server>:<nfs-path>,"volume-opt=o=addr=<nfs-address>,vers=4,soft,timeo=180,bg,tcp,rw"'

--name myservice \

<IMAGE>

The examples below show both the --mount and -v syntax where possible, and --mount is presented first.

### Differences between -v and --mount behavior

As opposed to bind mounts, all options for volumes are available for both --mount and -v flags.

When using volumes with services, only --mount is supported.

## 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

local 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

If you start a container with a volume that does not yet exist, Docker creates the volume for you. The following example mounts the volume myvol2 into /app/ in the container.

The -v and --mount examples below produce the same result. You can’t run them both unless you remove the devtest container and the myvol2 volume after running the first one.

* --mount
* -v

$ docker run -d \

--name devtest \

--mount source=myvol2,target=/app \

nginx:latest

Use docker inspect devtest to verify that the volume was created and mounted correctly. Look for the Mounts section:

"Mounts": [

{

"Type": "volume",

"Name": "myvol2",

"Source": "/var/lib/docker/volumes/myvol2/\_data",

"Destination": "/app",

"Driver": "local",

"Mode": "",

"RW": true,

"Propagation": ""

}

],

This shows that the mount is a volume, it shows the correct source and destination, and that the mount is read-write.

Stop the container and remove the volume. Note volume removal is a separate step.

$ docker container stop devtest

$ docker container rm devtest

$ docker volume rm myvol2

### Start a service with volumes

When you start a service and define a volume, each service container uses its own local volume. None of the containers can share this data if you use the local volume driver, but some volume drivers do support shared storage. Docker for AWS and Docker for Azure both support persistent storage using the Cloudstor plugin.

The following example starts a nginx service with four replicas, each of which uses a local volume called myvol2.

$ docker service create -d \

--replicas=4 \

--name devtest-service \

--mount source=myvol2,target=/app \

nginx:latest

Use docker service ps devtest-service to verify that the service is running:

$ docker service ps devtest-service

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

4d7oz1j85wwn devtest-service.1 nginx:latest moby Running Running 14 seconds ago

Remove the service, which stops all its tasks:

$ docker service rm devtest-service

Removing the service does not remove any volumes created by the service. Volume removal is a separate step.

#### **SYNTAX DIFFERENCES FOR SERVICES**

The docker service create command does not support the -v or --volume flag. When mounting a volume into a service’s containers, you must use the --mount flag.

### Populate a volume using a container

If you start a container which creates a new volume, as above, and the container has files or directories in the directory to be mounted (such as /app/ above), the directory’s contents are copied into the volume. The container then mounts and uses the volume, and other containers which use the volume also have access to the pre-populated content.

To illustrate this, this example starts an nginx container and populates the new volume nginx-vol with the contents of the container’s /usr/share/nginx/html directory, which is where Nginx stores its default HTML content.

The --mount and -v examples have the same end result.

* --mount
* -v

$ docker run -d \

--name=nginxtest \

--mount source=nginx-vol,destination=/usr/share/nginx/html \

nginx:latest

After running either of these examples, run the following commands to clean up the containers and volumes. Note volume removal is a separate step.

$ docker container stop nginxtest

$ docker container rm nginxtest

$ docker volume rm nginx-vol

## Use a read-only volume

For some development applications, the container needs to write into the bind mount so that changes are propagated back to the Docker host. At other times, the container only needs read access to the data. Remember that multiple containers can mount the same volume, and it can be mounted read-write for some of them and read-only for others, at the same time.

This example modifies the one above but mounts the directory as a read-only volume, by adding ro to the (empty by default) list of options, after the mount point within the container. Where multiple options are present, separate them by commas.

The --mount and -v examples have the same result.

* --mount
* -v

$ docker run -d \

--name=nginxtest \

--mount source=nginx-vol,destination=/usr/share/nginx/html,readonly \

nginx:latest

Use docker inspect nginxtest to verify that the readonly mount was created correctly. Look for the Mounts section:

"Mounts": [

{

"Type": "volume",

"Name": "nginx-vol",

"Source": "/var/lib/docker/volumes/nginx-vol/\_data",

"Destination": "/usr/share/nginx/html",

"Driver": "local",

"Mode": "",

"RW": false,

"Propagation": ""

}

],

Stop and remove the container, and remove the volume. Volume removal is a separate step.

$ docker container stop nginxtest

$ docker container rm nginxtest

$ docker volume rm nginx-vol

## Share data among machines

When building fault-tolerant applications, you might need to configure multiple replicas of the same service to have access to the same files.

There are several ways to achieve this when developing your applications. One is to add logic to your application to store files on a cloud object storage system like Amazon S3. Another is to create volumes with a driver that supports writing files to an external storage system like NFS or Amazon S3.

Volume drivers allow you to abstract the underlying storage system from the application logic. For example, if your services use a volume with an NFS driver, you can update the services to use a different driver, as an example to store data in the cloud, without changing the application logic.

## Use a volume driver

When you create a volume using docker volume create, or when you start a container which uses a not-yet-created volume, you can specify a volume driver. The following examples use the vieux/sshfs volume driver, first when creating a standalone volume, and then when starting a container which creates a new volume.

### Initial set-up

This example assumes that you have two nodes, the first of which is a Docker host and can connect to the second using SSH.

On the Docker host, install the vieux/sshfs plugin:

$ docker plugin install --grant-all-permissions vieux/sshfs

### Create a volume using a volume driver

This example specifies a SSH password, but if the two hosts have shared keys configured, you can omit the password. Each volume driver may have zero or more configurable options, each of which is specified using an -o flag.

$ docker volume create --driver vieux/sshfs \

-o sshcmd=test@node2:/home/test \

-o password=testpassword \

sshvolume

### Start a container which creates a volume using a volume driver

This example specifies a SSH password, but if the two hosts have shared keys configured, you can omit the password. Each volume driver may have zero or more configurable options. **If the volume driver requires you to pass options, you must use the --mount flag to mount the volume, rather than -v.**

$ docker run -d \

--name sshfs-container \

--volume-driver vieux/sshfs \

--mount src=sshvolume,target=/app,volume-opt=sshcmd=test@node2:/home/test,volume-opt=password=testpassword \

nginx:latest

### Create a service which creates an NFS volume

This example shows how you can create an NFS volume when creating a service. This example uses 10.0.0.10 as the NFS server and /var/docker-nfs as the exported directory on the NFS server. Note that the volume driver specified is local.

#### **NFSV3**

$ docker service create -d \

--name nfs-service \

--mount 'type=volume,source=nfsvolume,target=/app,volume-driver=local,volume-opt=type=nfs,volume-opt=device=:/var/docker-nfs,volume-opt=o=addr=10.0.0.10' \

nginx:latest

#### **NFSV4**

docker service create -d \

--name nfs-service \

--mount 'type=volume,source=nfsvolume,target=/app,volume-driver=local,volume-opt=type=nfs,volume-opt=device=:/,"volume-opt=o=10.0.0.10,rw,nfsvers=4,async"' \

nginx:latest

## Backup, restore, or migrate data volumes

Volumes are useful for backups, restores, and migrations. Use the --volumes-from flag to create a new container that mounts that volume.

### Backup a container

For example, create a new container named dbstore:

$ docker run -v /dbdata --name dbstore ubuntu /bin/bash

Then in the next command, we:

* Launch a new container and mount the volume from the dbstore container
* Mount a local host directory as /backup
* Pass a command that tars the contents of the dbdata volume to a backup.tar file inside our /backup directory.

$ docker run --rm --volumes-from dbstore -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata

When the command completes and the container stops, we are left with a backup of our dbdata volume.

### Restore container from backup

With the backup just created, you can restore it to the same container, or another that you made elsewhere.

For example, create a new container named dbstore2:

$ docker run -v /dbdata --name dbstore2 ubuntu /bin/bash

Then un-tar the backup file in the new container`s data volume:

$ docker run --rm --volumes-from dbstore2 -v $(pwd):/backup ubuntu bash -c "cd /dbdata && tar xvf /backup/backup.tar --strip 1"

You can use the techniques above to automate backup, migration and restore testing using your preferred tools.

## Remove volumes

A Docker data volume persists after a container is deleted. There are two types of volumes to consider:

* **Named volumes** have a specific source from outside the container, for example awesome:/bar.
* **Anonymous volumes** have no specific source so when the container is deleted, instruct the Docker Engine daemon to remove them.

### Remove anonymous volumes

To automatically remove anonymous volumes, use the --rm option. For example, this command creates an anonymous /foo volume. When the container is removed, the Docker Engine removes the /foo volume but not the awesome volume.

$ docker run --rm -v /foo -v awesome:/bar busybox top

### Remove all volumes

To remove all unused volumes and free up space:

$ docker volume prune

<https://docs.docker.com/storage/volumes/>