Use the Device Mapper storage driver

Estimated reading time: 27 minutes

Device Mapper is a kernel-based framework that underpins many advanced volume management technologies on Linux. Docker's `devicemapper` storage driver leverages the thin provisioning and snapshotting capabilities of this framework for image and container management. This article refers to the Device Mapper storage driver as `devicemapper`, and the kernel framework as *Device Mapper*.

For the systems where it is supported, `devicemapper` support is included in the Linux kernel. However, specific configuration is required to use it with Docker.

The `devicemapper` driver uses block devices dedicated to Docker and operates at the block level, rather than the file level. These devices can be extended by adding physical storage to your Docker host, and they perform better than using a filesystem at the operating system (OS) level.

Prerequisites

- `devicemapper` storage driver is a supported storage driver for Docker EE on many OS distribution. See the Product compatibility matrix (https://success.docker.com/article/compatibility-matrix) for details.
- `devicemapper` is also supported on Docker CE running on CentOS, Fedora, Ubuntu, or Debian.
- Changing the storage driver makes any containers you have already created inaccessible on the local system. Use `docker save` to save containers, and push existing images to Docker Hub or a private repository, so you do not need to recreate them later.

Configure Docker with the `devicemapper` storage driver

Before following these procedures, you must first meet all the prerequisites (/storage/storagedriver/device-mapper-driver/#prerequisites).

Configure `loop-lvm` mode for testing

This configuration is only appropriate for testing. Loopback devices are slow and resource-intensive, and they require you to create file on disk at specific sizes. They can also introduce race conditions. They are available for testing because the setup is easier.

For production systems, see Configure direct-lvm mode for production (/storage/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-for-production).

1. Stop Docker.

```
$ sudo systemctl stop docker
```

2. Edit `/etc/docker/daemon.json`. If it does not yet exist, create it. Assuming that the file was empty, add the following contents.

   ```
   {
     "storage-driver": "devicemapper"
   }
   ```

   See all storage options for each storage driver:

   - Stable (https://docs.docker.com/engine/reference/commandline/dockerd/#storage-driver-options)
   - Edge (https://docs.docker.com/edge/engine/reference/commandline/dockerd/#storage-driver-options)

   Docker does not start if the `daemon.json` file contains badly-formed JSON.

3. Start Docker.

   ```
   $ sudo systemctl start docker
   ```

4. Verify that the daemon is using the `devicemapper` storage driver. Use the `docker info` command and look for `Storage Driver`.

```
$ docker info

  Containers: 0
    Running: 0
    Paused: 0
    Stopped: 0
  Images: 0
  Server Version: 17.03.1-ce
  Storage Driver: devicemapper
  Pool Name: docker-202:1-8413957-pool
  Pool Blocksize: 65.54 kB
  Base Device Size: 10.74 GB
  Backing Filesystem: xfs
  Data file: /dev/loop0
  Metadata file: /dev/loop1
  Data Space Used: 11.8 MB
  Data Space Total: 107.4 GB
  Data Space Available: 7.44 GB
  Metadata Space Used: 581.6 kB
  Metadata Space Total: 2.147 GB
  Metadata Space Available: 2.147 GB
  Thin Pool Minimum Free Space: 10.74 GB
  Udev Sync Supported: true
  Deferred Removal Enabled: false
  Deferred Deletion Enabled: false
  Deferred Deleted Device Count: 0
  Data loop file: /var/lib/docker/devicemapper/data
  Metadata loop file: /var/lib/docker/devicemapper/metadata
  Library Version: 1.02.135-RHEL7 (2016-11-16)
<output truncated>
```

This host is running in `loop-lvm` mode, which is not supported on production systems. This is indicated by the fact that the `Data loop file` and a `Metadata loop file` are on files under `/var/lib/docker/devicemapper`. These are loopback-mounted sparse files. For production systems, see Configure direct-lvm mode for production (/storage/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-for-production).

Configure direct-lvm mode for production

Production hosts using the `devicemapper` storage driver must use `direct-lvm` mode. This mode uses block devices to create the thin pool. This is faster than using loopback devices, uses system resources more efficiently, and block devices can grow as needed. However, more set-up is required than `loop-lvm` mode.

After you have satisfied the prerequisites (/storage/storagedriver/device-mapper-driver/#prerequisites), follow the steps below to configure Docker to use the `devicemapper` storage driver in `direct-lvm` mode.

> Warning: Changing the storage driver makes any containers you have already created inaccessible on the local system. Use `docker save` to save containers, and push existing images to Docker Hub or a private repository, so you do not need to recreate them later.

ALLOW DOCKER TO CONFIGURE DIRECT-LVM MODE

With Docker `17.06` and higher, Docker can manage the block device for you, simplifying configuration of `direct-lvm` mode. This is appropriate for fresh Docker setups only. You can only use a single block device. If you need to use multiple block devices, configure direct-lvm mode manually (/storage/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-manually) instead. The following new configuration options have been added:

Option	Description	Required?	Default	Example
`dm.directlvm_device`	The path to the block device to configure for `direct-lvm`.	Yes		`dm.directlvm_device="/dev/xvdf"`
`dm.thinp_percent`	The percentage of space to use for storage from the passed in block device.	No	95	`dm.thinp_percent=95`
`dm.thinp_metapercent`	The percentage of space to for metadata storage from the passed-in block device.	No	1	`dm.thinp_metapercent=1`
`dm.thinp_autoextend_threshold`	The threshold for when lvm should automatically extend the thin pool as a percentage of the total storage space.	No	80	`dm.thinp_autoextend_threshold=80`
`dm.thinp_autoextend_percent`	The percentage to increase the thin pool by when an autoextend is triggered.	No	20	`dm.thinp_autoextend_percent=20`

Option	Description	Required?	Default	Example
`dm.directlvm_device_force`	Whether to format the block device even if a filesystem already exists on it. If set to `false` and a filesystem is present, an error is logged and the filesystem is left intact.	No	false	`dm.directlvm_device_force=true`

Edit the `daemon.json` file and set the appropriate options, then restart Docker for the changes to take effect. The following `daemon.json` configuration sets all of the options in the table above.

```
{
  "storage-driver": "devicemapper",
  "storage-opts": [
    "dm.directlvm_device=/dev/xdf",
    "dm.thinp_percent=95",
    "dm.thinp_metapercent=1",
    "dm.thinp_autoextend_threshold=80",
    "dm.thinp_autoextend_percent=20",
    "dm.directlvm_device_force=false"
  ]
}
```

See all storage options for each storage driver:

- Stable (https://docs.docker.com/engine/reference/commandline/dockerd/#storage-driver-options)
- Edge (https://docs.docker.com/edge/engine/reference/commandline/dockerd/#storage-driver-options)

Restart Docker for the changes to take effect. Docker invokes the commands to configure the block device for you.

> Warning: Changing these values after Docker has prepared the block device for you is not supported and causes an error.

You still need to perform periodic maintenance tasks (/storage/storagedriver/device-mapper-driver/#manage-devicemapper).

CONFIGURE DIRECT-LVM MODE MANUALLY

The procedure below creates a logical volume configured as a thin pool to use as backing for the storage pool. It assumes that you have a spare block device at `/dev/xvdf` with enough free space to complete the task. The device identifier and volume sizes may be different in your environment and you should substitute your own values throughout the procedure. The procedure also assumes that the Docker daemon is in the `stopped` state.

1. Identify the block device you want to use. The device is located under `/dev/` (such as `/dev/xvdf`) and needs enough free space to store the images and container layers for the workloads that host runs. A solid state drive is ideal.
2. Stop Docker.

   ```
   $ sudo systemctl stop docker
   ```

3. Install the following packages:
   - RHEL / CentOS: `device-mapper-persistent-data`, `lvm2`, and all dependencies
   - Ubuntu / Debian: `thin-provisioning-tools`, `lvm2`, and all dependencies
4. Create a physical volume on your block device from step 1, using the `pvcreate` command. Substitute your device name for `/dev/xvdf`.

   > Warning: The next few steps are destructive, so be sure that you have specified the correct device!

   ```
   $ sudo pvcreate /dev/xvdf

   Physical volume "/dev/xvdf" successfully created.
   ```

5. Create a `docker` volume group on the same device, using the `vgcreate` command.

   ```
   $ sudo vgcreate docker /dev/xvdf

   Volume group "docker" successfully created
   ```

6. Create two logical volumes named `thinpool` and `thinpoolmeta` using the `lvcreate` command. The last parameter specifies the amount of free space to allow for automatic expanding of the data or metadata if space runs low, as a temporary stop-gap. These are the recommended values.

   ```
   $ sudo lvcreate --wipesignatures y -n thinpool docker -l 95%VG

   Logical volume "thinpool" created.

   $ sudo lvcreate --wipesignatures y -n thinpoolmeta docker -l 1%VG

   Logical volume "thinpoolmeta" created.
   ```

7. Convert the volumes to a thin pool and a storage location for metadata for the thin pool, using the `lvconvert` command.

```
$ sudo lvconvert -y \
--zero n \
-c 512K \
--thinpool docker/thinpool \
--poolmetadata docker/thinpoolmeta

WARNING: Converting logical volume docker/thinpool and docker/thinpoolmeta to
thin pool's data and metadata volumes with metadata wiping.
THIS WILL DESTROY CONTENT OF LOGICAL VOLUME (filesystem etc.)
Converted docker/thinpool to thin pool.
```

8. Configure autoextension of thin pools via an `lvm` profile.

```
$ sudo vi /etc/lvm/profile/docker-thinpool.profile
```

9. Specify `thin_pool_autoextend_threshold` and `thin_pool_autoextend_percent` values.

   `thin_pool_autoextend_threshold` is the percentage of space used before `lvm` attempts to autoextend the available space (100 = disabled, not recommended).

   `thin_pool_autoextend_percent` is the amount of space to add to the device when automatically extending (0 = disabled).

   The example below adds 20% more capacity when the disk usage reaches 80%.

```
activation {
  thin_pool_autoextend_threshold=80
  thin_pool_autoextend_percent=20
}
```

   Save the file.

10. Apply the LVM profile, using the `lvchange` command.

```
$ sudo lvchange --metadataprofile docker-thinpool docker/thinpool

Logical volume docker/thinpool changed.
```

11. Enable monitoring for logical volumes on your host. Without this step, automatic extension does not occur even in the presence of the LVM profile.

```
$ sudo lvs -o+seg_monitor

LV       VG     Attr       LSize  Pool Origin Data%  Meta%  Move Log Cpy%Sync Convert
 Monitor
thinpool docker twi-a-t--- 95.00g             0.00   0.01
 monitored
```

12. If you have ever run Docker on this host before, or if `/var/lib/docker/` exists, move it out of the way so that Docker can use the new LVM pool to store the contents of image and containers.

    ```
    $ mkdir /var/lib/docker.bk
    $ mv /var/lib/docker/* /var/lib/docker.bk
    ```

    If any of the following steps fail and you need to restore, you can remove `/var/lib/docker` and replace it with `/var/lib/docker.bk`.

13. Edit `/etc/docker/daemon.json` and configure the options needed for the `devicemapper` storage driver. If the file was previously empty, it should now contain the following contents:

    ```
    {
        "storage-driver": "devicemapper",
        "storage-opts": [
        "dm.thinpooldev=/dev/mapper/docker-thinpool",
        "dm.use_deferred_removal=true",
        "dm.use_deferred_deletion=true"
        ]
    }
    ```

14. Start Docker.

    systemd:

    ```
    $ sudo systemctl start docker
    ```

    service:

    ```
    $ sudo service docker start
    ```

15. Verify that Docker is using the new configuration using `docker info`.

```
$ docker info

Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 17.03.1-ce
Storage Driver: devicemapper
 Pool Name: docker-thinpool
 Pool Blocksize: 524.3 kB
 Base Device Size: 10.74 GB
 Backing Filesystem: xfs
 Data file:
 Metadata file:
 Data Space Used: 19.92 MB
 Data Space Total: 102 GB
 Data Space Available: 102 GB
 Metadata Space Used: 147.5 kB
 Metadata Space Total: 1.07 GB
 Metadata Space Available: 1.069 GB
 Thin Pool Minimum Free Space: 10.2 GB
 Udev Sync Supported: true
 Deferred Removal Enabled: true
 Deferred Deletion Enabled: true
 Deferred Deleted Device Count: 0
 Library Version: 1.02.135-RHEL7 (2016-11-16)
<output truncated>
```

If Docker is configured correctly, the `Data file` and `Metadata file` is blank, and the pool name is `docker-thinpool`.

16. After you have verified that the configuration is correct, you can remove the `/var/lib/docker.bk` directory which contains the previous configuration.

```
$ rm -rf /var/lib/docker.bk
```

Manage devicemapper

Monitor the thin pool

Do not rely on LVM auto-extension alone. The volume group automatically extends, but the volume can still fill up. You can monitor free space on the volume using `lvs` or `lvs -r`. Consider using a monitoring tool at the OS level, such a Nagios.

To view the LVM logs, you can use `journalctl`:

```
$ journalctl -fu dm-event.service
```

If you run into repeated problems with thin pool, you can set the storage option `dm.min_free_space` to a value (representing a percentage) in `/etc/docker.daemon.json`. For instance, setting it to `10` ensures that operations fail with a warning when the free space is at or near 10%. See the storage driver options in the Engine daemon reference (https://docs.docker.com/engine/reference/commandline/dockerd/#storage-driver-options).

Increase capacity on a running device

You can increase the capacity of the pool on a running thin-pool device. This is useful if the data's logical volume is full and the volume group is at full capacity. The specific procedure depends on whether you are using a loop-lvm thin pool (/storage/storagedriver/device-mapper-driver/#resize-a-loop-lvm-thin-pool) or a direct-lvm thin pool (/storage/storagedriver/device-mapper-driver/#resize-a-direct-lvm-thin-pool).

RESIZE A LOOP-LVM THIN POOL

The easiest way to resize a `loop-lvm` thin pool is to use the device_tool utility (/storage/storagedriver/device-mapper-driver/#use-the-device_tool-utility), but you can use operating system utilities (/storage/storagedriver/device-mapper-driver/#use-operating-system-utilities) instead.

Use the device_tool utility

A community-contributed script called `device_tool.go` is available in the moby/moby (https://github.com/moby/moby/tree/master/contrib/docker-device-tool) Github repository. You can use this tool to resize a `loop-lvm` thin pool, avoiding the long process above. This tool is not guaranteed to work, but you should only be using `loop-lvm` on non-production systems.

If you do not want to use `device_tool`, you can resize the thin pool manually (/storage/storagedriver/device-mapper-driver/#use-operating-system-utilities) instead.

1. To use the tool, clone the Github repository, change to the `contrib/docker-device-tool`, and follow the instructions in the `README.md` to compile the tool.
2. Use the tool. The following example resizes the thin pool to 200GB.

```
$ ./device_tool resize 200GB
```

Use operating system utilities

If you do not want to use the device-tool utility (/storage/storagedriver/device-mapper-driver/#use-the-device_tool-utility), you can resize a `loop-lvm` thin pool manually using the following procedure.

In `loop-lvm` mode, a loopback device is used to store the data, and another to store the metadata. `loop-lvm` mode is only supported for testing, because it has significant performance and stability drawbacks.

If you are using `loop-lvm` mode, the output of `docker info` shows file paths for `Data loop file` and `Metadata loop file`:

```
$ docker info |grep 'loop file'

 Data loop file: /var/lib/docker/devicemapper/data
 Metadata loop file: /var/lib/docker/devicemapper/metadata
```

Follow these steps to increase the size of the thin pool. In this example, the thin pool is 100 GB, and is increased to 200 GB.

1. List the sizes of the devices.

   ```
   $ sudo ls -lh /var/lib/docker/devicemapper/

   total 1175492
   -rw------- 1 root root 100G Mar 30 05:22 data
   -rw------- 1 root root 2.0G Mar 31 11:17 metadata
   ```

2. Increase the size of the `data` file to 200 G using the `truncate` command, which is used to increase or decrease the size of a file. Note that decreasing the size is a destructive operation.

   ```
   $ sudo truncate -s 200G /var/lib/docker/devicemapper/data
   ```

3. Verify the file size changed.

   ```
   $ sudo ls -lh /var/lib/docker/devicemapper/

   total 1.2G
   -rw------- 1 root root 200G Apr 14 08:47 data
   -rw------- 1 root root 2.0G Apr 19 13:27 metadata
   ```

4. The loopback file has changed on disk but not in memory. List the size of the loopback device in memory, in GB. Reload it, then list the size again. After the reload, the size is 200 GB.

   ```
   $ echo $[ $(sudo blockdev --getsize64 /dev/loop0) / 1024 / 1024 / 1024 ]

   100

   $ sudo losetup -c /dev/loop0

   $ echo $[ $(sudo blockdev --getsize64 /dev/loop0) / 1024 / 1024 / 1024 ]

   200
   ```

5. Reload the devicemapper thin pool.

   a. Get the pool name first. The pool name is the first field, delimited by `:`. This command extracts it.

```
$ sudo dmsetup status | grep ' thin-pool ' | awk -F ': ' {'print $1'}

docker-8:1-123141-pool
```

b. Dump the device mapper table for the thin pool.

```
$ sudo dmsetup table docker-8:1-123141-pool

0 209715200 thin-pool 7:1 7:0 128 32768 1 skip_block_zeroing
```

c. Calculate the total sectors of the thin pool using the second field of the output. The number is expressed in 512-k sectors. A 100G file has 209715200 512-k sectors. If you double this number to 200G, you get 419430400 512-k sectors.

d. Reload the thin pool with the new sector number, using the following three `dmsetup` commands.

```
$ sudo dmsetup suspend docker-8:1-123141-pool

$ sudo dmsetup reload docker-8:1-123141-pool --table '0 419430400 thin-pool 7:1 7
:0 128 32768 1 skip_block_zeroing'

$ sudo dmsetup resume docker-8:1-123141-pool
```

RESIZE A DIRECT-LVM THIN POOL

To extend a `direct-lvm` thin pool, you need to first attach a new block device to the Docker host, and make note of the name assigned to it by the kernel. In this example, the new block device is `/dev/xvdg`.

Follow this procedure to extend a `direct-lvm` thin pool, substituting your block device and other parameters to suit your situation.

1. Gather information about your volume group.

   Use the `pvdisplay` command to find the physical block devices currently in use by your thin pool, and the volume group's name.

   ```
   $ sudo pvdisplay |grep 'VG Name'

   PV Name               /dev/xvdf
   VG Name               docker
   ```

   In the following steps, substitute your block device or volume group name as appropriate.

2. Extend the volume group, using the `vgextend` command with the `VG Name` from the previous step, and the name of your new block device.

```
$ sudo vgextend docker /dev/xvdg

Physical volume "/dev/xvdg" successfully created.
Volume group "docker" successfully extended
```

3. Extend the `docker/thinpool` logical volume. This command uses 100% of the volume right away, without auto-extend. To extend the metadata thinpool instead, use `docker/thinpool_tmeta`.

```
$ sudo lvextend -·+100%FREE -n docker/thinpool

Size of logical volume docker/thinpool_tdata changed from 95.00 GiB (24319 extents) t
o 198.00 GiB (50688 extents).
Logical volume docker/thinpool_tdata successfully resized.
```

4. Verify the new thin pool size using the `Data Space Available` field in the output of `docker info`. If you extended the `docker/thinpool_tmeta` logical volume instead, look for `Metadata Space Available`.

```
Storage Driver: devicemapper
 Pool Name: docker-thinpool
 Pool Blocksize: 524.3 kB
 Base Device Size: 10.74 GB
 Backing Filesystem: xfs
 Data file:
 Metadata file:
 Data Space Used: 212.3 MB
 Data Space Total: 212.6 GB
 Data Space Available: 212.4 GB
 Metadata Space Used: 286.7 kB
 Metadata Space Total: 1.07 GB
 Metadata Space Available: 1.069 GB
<output truncated>
```

Activate the `devicemapper` after reboot

If you reboot the host and find that the `docker` service failed to start, look for the error, "Non existing device". You need to re-activate the logical volumes with this command:

```
sudo lvchange -ay docker/thinpool
```

How the `devicemapper` storage driver works

> Warning: Do not directly manipulate any files or directories within `/var/lib/docker/`. These files and directories are managed by Docker.

Use the `lsblk` command to see the devices and their pools, from the operating system's point of view:

```
$ sudo lsblk

NAME                  MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
xvda                  202:0    0    8G  0 disk
└─xvda1               202:1    0    8G  0 part /
xvdf                  202:80   0  100G  0 disk
├─docker-thinpool_tmeta 253:0    0 1020M  0 lvm
│ └─docker-thinpool     253:2    0   95G  0 lvm
└─docker-thinpool_tdata 253:1    0   95G  0 lvm
  └─docker-thinpool     253:2    0   95G  0 lvm
```

Use the `mount` command to see the mount-point Docker is using:

```
$ mount |grep devicemapper
/dev/xvda1 on /var/lib/docker/devicemapper type xfs (rw,relatime,seclabel,attr2,inode64,no
quota)
```

When you use `devicemapper`, Docker stores image and layer contents in the thinpool, and exposes them to containers by mounting them under subdirectories of `/var/lib/docker/devicemapper/`.

Image and container layers on-disk

The `/var/lib/docker/devicemapper/metadata/` directory contains metadata about the Devicemapper configuration itself and about each image and container layer that exist. The `devicemapper` storage driver uses snapshots, and this metadata include information about those snapshots. These files are in JSON format.

The `/var/lib/devicemapper/mnt/` directory contains a mount point for each image and container layer that exists. Image layer mount points are empty, but a container's mount point shows the container's filesystem as it appears from within the container.

Image layering and sharing

The `devicemapper` storage driver uses dedicated block devices rather than formatted filesystems, and operates on files at the block level for maximum performance during copy-on-write (CoW) operations.

SNAPSHOTS

Another feature of `devicemapper` is its use of snapshots (also sometimes called *thin devices* or *virtual devices*), which store the differences introduced in each layer as very small, lightweight thin pools. Snapshots provide many benefits:

- Layers which are shared in common between containers are only stored on disk once, unless they are writable. For instance, if you have 10 different images which are all based on `alpine`, the `alpine` image and all its parent images are only stored once each on disk.
- Snapshots are an implementation of a copy-on-write (CoW) strategy. This means that a given file or directory is only copied to the container's writable layer when it is modified or deleted by that container.
- Because `devicemapper` operates at the block level, multiple blocks in a writable layer can be

modified simultaneously.

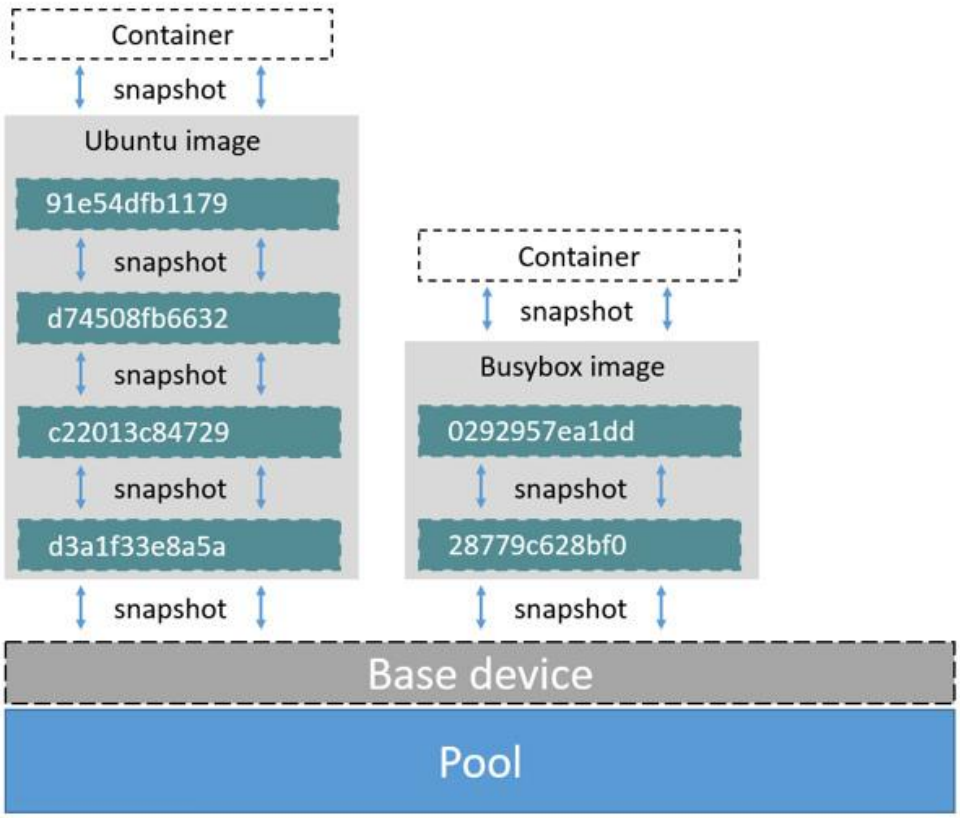
- Snapshots can be backed up using standard OS-level backup utilities. Just make a copy of `/var/lib/docker/devicemapper/` .

DEVICEMAPPER WORKFLOW

When you start Docker with the `devicemapper` storage driver, all objects related to image and container layers are stored in `/var/lib/docker/devicemapper/` , which is backed by one or more block-level devices, either loopback devices (testing only) or physical disks.

- The *base device* is the lowest-level object. This is the thin pool itself. You can examine it using `docker info` . It contains a filesystem. This base device is the starting point for every image and container layer. The base device is a Device Mapper implementation detail, rather than a Docker layer.
- Metadata about the base device and each image or container layer is stored in `/var/lib/docker/devicemapper/metadata/` in JSON format. These layers are copy-on-write snapshots, which means that they are empty until they diverge from their parent layers.
- Each container's writable layer is mounted on a mountpoint in `/var/lib/docker/devicemapper/mnt/` . An empty directory exists for each read-only image layer and each stopped container.

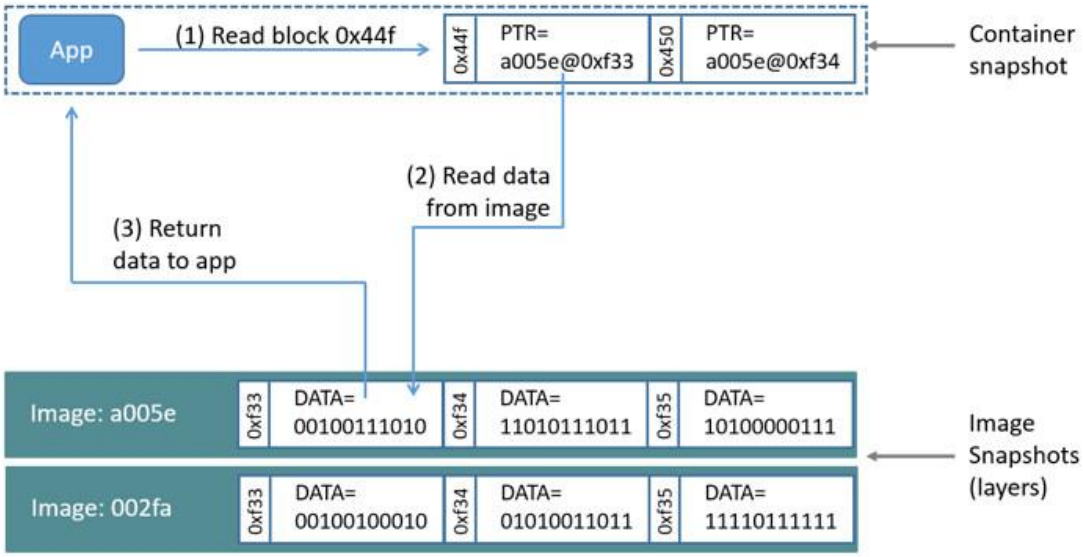
Each image layer is a snapshot of the layer below it. The lowest layer of each image is a snapshot of the base device that exists in the pool. When you run a container, it is a snapshot of the image the container is based on. The following example shows a Docker host with two running containers. The first is a `ubuntu` container and the second is a `busybox` container.



How container reads and writes work with `devicemapper`

Reading files

With `devicemapper`, reads happen at the block level. The diagram below shows the high level process for reading a single block (`0x44f`) in an example container.



An application makes a read request for block `0x44f` in the container. Because the container is a thin snapshot of an image, it doesn't have the block, but it has a pointer to the block on the nearest parent image where it does exist, and it reads the block from there. The block now exists in the container's memory.

Writing files

Writing a new file: With the `devicemapper` driver, writing new data to a container is accomplished by an *allocate-on-demand* operation. Each block of the new file is allocated in the container's writable layer and the block is written there.

Updating an existing file: The relevant block of the file is read from the nearest layer where it exists. When the container writes the file, only the modified blocks are written to the container's writable layer.

Deleting a file or directory: When you delete a file or directory in a container's writable layer, or when an image layer deletes a file that exists in its parent layer, the `devicemapper` storage driver intercepts further read attempts on that file or directory and responds that the file or directory does not exist.

Writing and then deleting a file: If a container writes to a file and later deletes the file, all of those operations happen in the container's writable layer. In that case, if you are using `direct-lvm`, the blocks are freed. If you use `loop-lvm`, the blocks may not be freed. This is another reason not to use `loop-lvm` in production.

Device Mapper and Docker performance

- **`allocate-on demand`** performance impact:

  The `devicemapper` storage driver uses an `allocate-on-demand` operation to allocate new blocks from the thin pool into a container's writable layer. Each block is 64KB, so this is the minimum amount of space that is used for a write.

- Copy-on-write performance impact: The first time a container modifies a specific block, that block is written to the container's writable layer. Because these writes happen at the level of the block rather than the file, performance impact is minimized. However, writing a large number of blocks can still negatively impact performance, and the `devicemapper` storage driver may actually perform worse than other storage drivers in this scenario. For write-heavy workloads, you should use data volumes, which bypass the storage driver completely.

Performance best practices

Keep these things in mind to maximize performance when using the `devicemapper` storage driver.

- Use **`direct-lvm`** : The `loop-lvm` mode is not performant and should never be used in production.
- Use fast storage: Solid-state drives (SSDs) provide faster reads and writes than spinning disks.
- Memory usage: the `devicemapper` uses more memory than some other storage drivers. Each launched container loads one or more copies of its files into memory, depending on how many blocks of the same file are being modified at the same time. Due to the memory pressure, the `devicemapper` storage driver may not be the right choice for certain workloads in high-density use cases.
- Use volumes for write-heavy workloads: Volumes provide the best and most predictable performance for write-heavy workloads. This is because they bypass the storage driver and do not incur any of the potential overheads introduced by thin provisioning and copy-on-write. Volumes have other benefits, such as allowing you to share data among containers and persisting even when no running container is using them.
- Note: when using `devicemapper` and the `json-file` log driver, the log files generated by a container are still stored in Docker's dataroot directory, by default `/var/lib/docker` . If your containers generate lots of log messages, this may lead to increased disk usage or the inability to manage your system due to a full disk. You can configure a log driver (https://docs.docker.com/config/containers/logging/configure/) to store your container logs externally.

Related Information

- Volumes (https://docs.docker.com/storage/volumes/)
- Understand images, containers, and storage drivers (https://docs.docker.com/storage/storagedriver/imagesandcontainers/)
- Select a storage driver (https://docs.docker.com/storage/storagedriver/selectadriver/)

container (https://docs.docker.com/glossary/?term=container), storage (https://docs.docker.com/glossary/?term=storage), driver (https://docs.docker.com/glossary/?term=driver), device mapper (https://docs.docker.com/glossary/?term=device mapper)

Docker storage drivers

Estimated reading time: 10 minutes

Ideally, very little data is written to a container's writable layer, and you use Docker volumes to write data. However, some workloads require you to be able to write to the container's writable layer. This is where storage drivers come in.

Docker supports several different storage drivers, using a pluggable architecture. The storage driver controls how images and containers are stored and managed on your Docker host.

After you have read the storage driver overview (https://docs.docker.com/storage/storagedriver/), the next step is to choose the best storage driver for your workloads. In making this decision, there are three high-level factors to consider:

If multiple storage drivers are supported in your kernel, Docker has a prioritized list of which storage driver to use if no storage driver is explicitly configured, assuming that the storage driver meets the prerequisites.

Use the storage driver with the best overall performance and stability in the most usual scenarios.

Docker supports the following storage drivers:

- `overlay2` is the preferred storage driver, for all currently supported Linux distributions, and requires no extra configuration.
- `aufs` is the preferred storage driver for Docker 18.06 and older, when running on Ubuntu 14.04 on kernel 3.13 which has no support for `overlay2` .
- `devicemapper` is supported, but requires `direct-lvm` for production environments, because `loopback-lvm` , while zero-configuration, has very poor performance. `devicemapper` was the recommended storage driver for CentOS and RHEL, as their kernel version did not support `overlay2` . However, current versions of CentOS and RHEL now have support for `overlay2` , which is now the recommended driver.
- The `btrfs` and `zfs` storage drivers are used if they are the backing filesystem (the filesystem of the host on which Docker is installed). These filesystems allow for advanced options, such as creating "snapshots", but

require more maintenance and setup. Each of these relies on the backing filesystem being configured correctly.
- The `vfs` storage driver is intended for testing purposes, and for situations where no copy-on-write filesystem can be used. Performance of this storage driver is poor, and is not generally recommended for production use.

Docker's source code defines the selection order. You can see the order at the source code for Docker Engine - Community 18.09 (https://github.com/docker/docker-ce/blob/18.09/components/engine/daemon/graphdriver/driver_linux.go#L50)

If you run a different version of Docker, you can use the branch selector at the top of the file viewer to choose a different branch.

Some storage drivers require you to use a specific format for the backing filesystem. If you have external requirements to use a specific backing filesystem, this may limit your choices. See Supported backing filesystems (/storage/storagedriver/select-storage-driver/#supported-backing-filesystems).

After you have narrowed down which storage drivers you can choose from, your choice is determined by the characteristics of your workload and the level of stability you need. See Other considerations (/storage/storagedriver/select-storage-driver/#other-considerations) for help in making the final decision.

> *NOTE*: Your choice may be limited by your Docker edition, operating system, and distribution. For instance, `aufs` is only supported on Ubuntu and Debian, and may require extra packages to be installed, while `btrfs` is only supported on SLES, which is only supported with Docker Enterprise. See Support storage drivers per Linux distribution (/storage/storagedriver/select-storage-driver/#supported-storage-drivers-per-linux-distribution) for more information.

Supported storage drivers per Linux distribution

At a high level, the storage drivers you can use is partially determined by the Docker edition you use.

In addition, Docker does not recommend any configuration that requires you to disable security features of your operating system, such as the need to disable `selinux` if you use the `overlay` or `overlay2` driver on CentOS.

Docker Engine - Enterprise and Docker Enterprise

For Docker Engine - Enterprise and Docker Enterprise, the definitive resource for which storage drivers are supported is the Product compatibility matrix (https://success.docker.com/Policies/Compatibility_Matrix). To get commercial support from Docker, you must use a supported configuration.

Docker Engine - Community

For Docker Engine - Community, only some configurations are tested, and your operating system's kernel may not support every storage driver. In general, the following configurations work on recent versions of the Linux distribution:

Linux distribution	Recommended storage drivers	Alternative drivers
Docker Engine - Community on Ubuntu	`overlay2` or `aufs` (for Ubuntu 14.04 running on kernel 3.13)	`overlay` [1], `devicemapper` [2], `zfs` , `vfs`
Docker Engine - Community on Debian	`overlay2` (Debian Stretch), `aufs` or `devicemapper` (older versions)	`overlay` [1], `vfs`
Docker Engine - Community on CentOS	`overlay2`	`overlay` [1], `devicemapper` [2], `zfs` , `vfs`
Docker Engine - Community on Fedora	`overlay2`	`overlay` [1], `devicemapper` [2], `zfs` , `vfs`

[1]) The `overlay` storage driver is deprecated in Docker Engine - Enterprise 18.09, and will be removed in a future release. It is recommended that users of the `overlay` storage driver migrate to `overlay2` .

[2]) The `devicemapper` storage driver is deprecated in Docker Engine 18.09, and will be removed in a future release. It is recommended that users of the `overlay` storage driver migrate to `overlay2`.

When possible, `overlay2` is the recommended storage driver. When installing Docker for the first time, `overlay2` is used by default. Previously, `aufs` was used by default when available, but this is no longer the case. If you want to use `aufs` on new installations going forward, you need to explicitly configure it, and you may need to install extra packages, such as `linux-image-extra`. See aufs (https://docs.docker.com/storage/storagedriver/aufs-driver/).

On existing installations using `aufs`, it is still used.

When in doubt, the best all-around configuration is to use a modern Linux distribution with a kernel that supports the `overlay2` storage driver, and to use Docker volumes for write-heavy workloads instead of relying on writing data to the container's writable layer.

The `vfs` storage driver is usually not the best choice. Before using the `vfs` storage driver, be sure to read about its performance and storage characteristics and limitations (https://docs.docker.com/storage/storagedriver/vfs-driver/).

> Expectations for non-recommended storage drivers: Commercial support is not available for Docker Engine - Community, and you can technically use any storage driver that is available for your platform. For instance, you can use `btrfs` with Docker Engine - Community, even though it is not recommended on any platform for Docker Engine - Community, and you do so at your own risk.
>
> The recommendations in the table above are based on automated regression testing and the configurations that are known to work for a large number of users. If you use a recommended configuration and find a reproducible issue, it is likely to be fixed very quickly. If the driver that you want to use is not recommended according to this table, you can run it at your own risk. You can and should still report any issues you run into. However, such issues have a lower priority than issues encountered when using a recommended configuration.

Docker for Mac and Docker for Windows

Docker for Mac and Docker for Windows are intended for development, rather than production. Modifying the storage driver on these platforms is not possible.

Supported backing filesystems

With regard to Docker, the backing filesystem is the filesystem where `/var/lib/docker/` is located. Some storage drivers only work with specific backing filesystems.

Storage driver	Supported backing filesystems
`overlay2` , `overlay`	`xfs` with ftype=1, `ext4`
`aufs`	`xfs` , `ext4`
`devicemapper`	`direct-lvm`
`btrfs`	`btrfs`
`zfs`	`zfs`
`vfs`	any filesystem

Other considerations

Suitability for your workload

Among other things, each storage driver has its own performance characteristics that make it more or less suitable for different workloads. Consider the following generalizations:

- `overlay2` , `aufs` , and `overlay` all operate at the file level rather than the block level. This uses memory more efficiently, but the container's writable layer may grow quite large in write-heavy workloads.
- Block-level storage drivers such as `devicemapper` , `btrfs` , and `zfs` perform better for write-heavy workloads (though not as well as Docker volumes).
- For lots of small writes or containers with many layers or deep filesystems, `overlay` may perform better than `overlay2` , but consumes more inodes, which can lead to inode exhaustion.
- `btrfs` and `zfs` require a lot of memory.
- `zfs` is a good choice for high-density workloads such as PaaS.

More information about performance, suitability, and best practices is available in the documentation for each storage driver.

Shared storage systems and the storage driver

If your enterprise uses SAN, NAS, hardware RAID, or other shared storage systems, they may provide high availability, increased performance, thin provisioning, deduplication, and compression. In many cases, Docker can work on top of these storage systems, but Docker does not closely integrate with them.

Each Docker storage driver is based on a Linux filesystem or volume manager. Be sure to follow existing best practices for operating your storage driver (filesystem or volume manager) on top of your shared storage system. For example, if using the ZFS storage driver on top of a shared storage system, be sure to follow best practices for operating ZFS filesystems on top of that specific shared storage system.

Stability

For some users, stability is more important than performance. Though Docker considers all of the storage drivers mentioned here to be stable, some are newer and are still under active development. In general, `overlay2`, `aufs`, `overlay`, and `devicemapper` are the choices with the highest stability.

Test with your own workloads

You can test Docker's performance when running your own workloads on different storage drivers. Make sure to use equivalent hardware and workloads to match production conditions, so you can see which storage driver offers the best overall performance.

Check your current storage driver

The detailed documentation for each individual storage driver details all of the set-up steps to use a given storage driver.

To see what storage driver Docker is currently using, use `docker info` and look for the `Storage Driver` line:

```
$ docker info

Containers: 0
Images: 0
Storage Driver: overlay2
 Backing Filesystem: xfs
<output truncated>
```

To change the storage driver, see the specific instructions for the new storage driver. Some drivers require additional configuration, including configuration to physical or logical disks on the Docker host.

> ❗ Important: When you change the storage driver, any existing images and containers become inaccessible. This is because their layers cannot be used by the new storage driver. If you revert your changes, you can access the old images and containers again, but any that you pulled or created using the new driver are then inaccessible.

Related information

- About images, containers, and storage drivers (https://docs.docker.com/storage/storagedriver/)
- `aufs` storage driver in practice (https://docs.docker.com/storage/storagedriver/aufs-driver/)
- `devicemapper` storage driver in practice (https://docs.docker.com/storage/storagedriver/device-mapper-driver/)
- `overlay` and `overlay2` storage drivers in practice (https://docs.docker.com/storage/storagedriver/overlayfs-driver/)
- `btrfs` storage driver in practice (https://docs.docker.com/storage/storagedriver/btrfs-driver/)
- `zfs` storage driver in practice (https://docs.docker.com/storage/storagedriver/zfs-driver/)

container (https://docs.docker.com/glossary/?term=container), storage (https://docs.docker.com/glossary/?term=storage), driver (https://docs.docker.com/glossary/?term=driver), aufs (https://docs.docker.com/glossary/?term=aufs), btrfs (https://docs.docker.com/glossary/?term=btrfs), devicemapper (https://docs.docker.com/glossary/?term=devicemapper), zfs (https://docs.docker.com/glossary/?term=zfs), overlay

(https://docs.docker.com/glossary/?term=overlay), overlay2
(https://docs.docker.com/glossary/?term=overlay2)

About storage drivers

Estimated reading time: 14 minutes

To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way.

Storage drivers allow you to create data in the writable layer of your container. The files won't be persisted after the container is deleted, and both read and write speeds are low.

Learn how to use volumes (https://docs.docker.com/storage/volumes/) to persist data and improve performance.

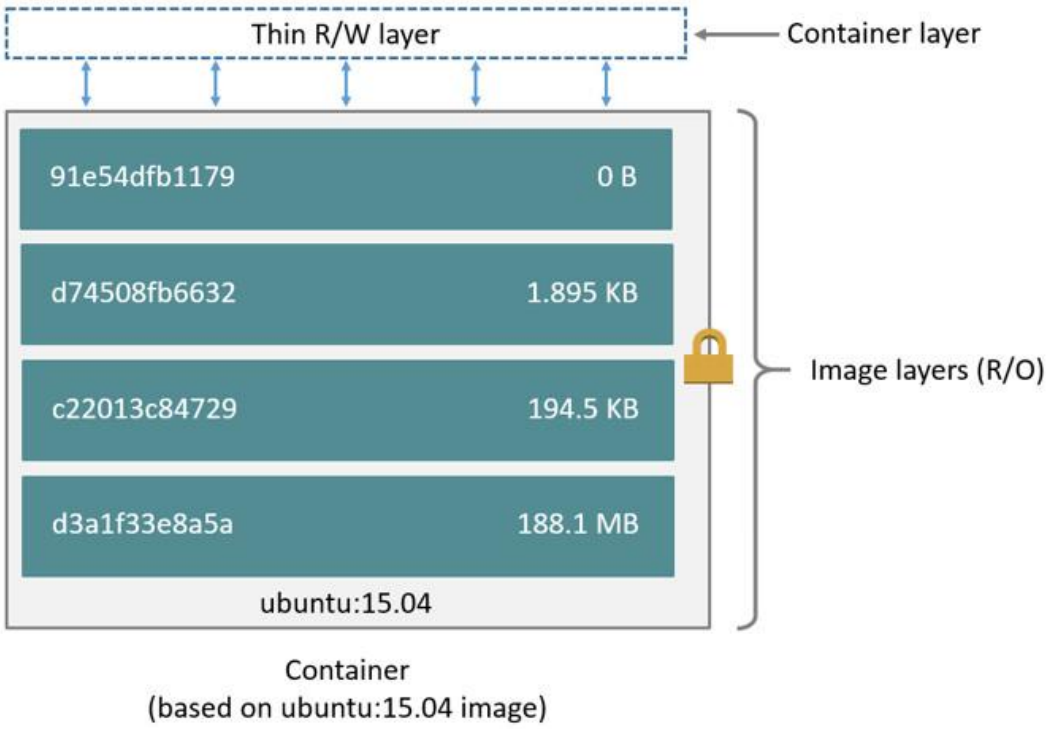
Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

This Dockerfile contains four commands, each of which creates a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:15.04` image. The `COPY` command adds some files from your Docker client's current directory. The `RUN` command builds your application using the `make` command. Finally, the last layer specifies what command to run within the container.

Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on the Ubuntu 15.04 image.

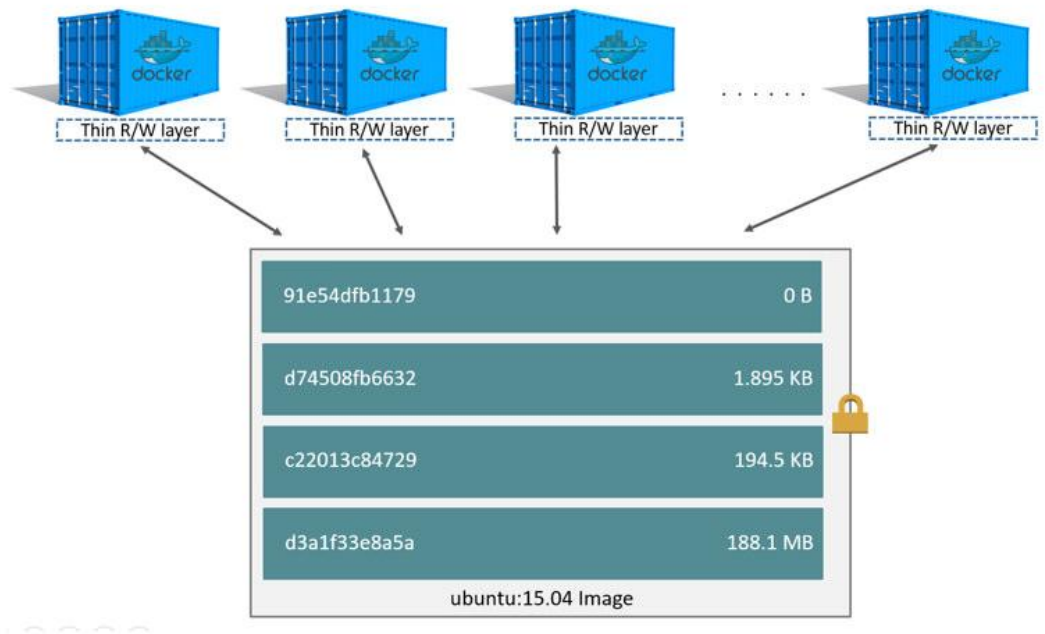


A *storage driver* handles the details about the way these layers interact with each other. Different storage drivers are available, which have advantages and disadvantages in different situations.

Container and layers

The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.

Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.



> Note: If you need multiple images to have shared access to the exact same data, store this data in a Docker volume and mount it into your containers.

Docker uses storage drivers to manage the contents of the image layers and the writable container layer. Each storage driver handles the implementation differently, but all drivers use stackable image layers and the copy-on-write (CoW) strategy.

Container size on disk

To view the approximate size of a running container, you can use the `docker ps -s` command. Two different columns relate to size.

- `size` : the amount of data (on disk) that is used for the writable layer of each container.
- `virtual size` : the amount of data used for the read-only image data used by the container plus the container's writable layer `size` . Multiple containers may share some or all read-only image data. Two containers started from the same image share 100% of the read-only data, while two containers with different images which have layers in common share those common layers. Therefore, you can't just total the virtual sizes. This over-estimates the total disk usage by a potentially non-trivial amount.

The total disk space used by all of the running containers on disk is some combination of each container's `size` and the `virtual size` values. If multiple containers started from the same exact image, the total size on disk for these containers would be SUM (`size` of containers) plus one image size (`virtual size - size`).

This also does not count the following additional ways a container can take up disk space:

- Disk space used for log files if you use the `json-file` logging driver. This can be non-trivial if your container generates a large amount of logging data and log rotation is not configured.
- Volumes and bind mounts used by the container.
- Disk space used for the container's configuration files, which are typically small.
- Memory written to disk (if swapping is enabled).
- Checkpoints, if you're using the experimental checkpoint/restore feature.

The copy-on-write (CoW) strategy

Copy-on-write is a strategy of sharing and copying files for maximum efficiency. If a file or directory exists in a lower layer within the image, and another layer (including the writable layer) needs read access to it, it just uses the existing file. The first time another layer needs to modify the file (when building the image or running the container), the file is copied into that layer and modified. This minimizes I/O and the size of each of the subsequent layers. These advantages are explained in more depth below.

Sharing promotes smaller images

When you use `docker pull` to pull down an image from a repository, or when you create a container from an image that does not yet exist locally, each layer is pulled down separately, and stored in Docker's local storage area, which is usually `/var/lib/docker/` on Linux hosts. You can see these layers being pulled in this example:

```
$ docker pull ubuntu:15.04

15.04: Pulling from library/ubuntu
1ba8ac955b97: Pull complete
f157c4e5ede7: Pull complete
0b7e98f84c4c: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:5e279a9df07990286cce22e1b0f5b0490629ca6d187698746ae5e
28e604a640e
Status: Downloaded newer image for ubuntu:15.04
```

Each of these layers is stored in its own directory inside the Docker host's local storage area. To examine the layers on the filesystem, list the contents of `/var/lib/docker/<storage-driver>/layers/`. This example uses the `aufs` storage driver:

```
$ ls /var/lib/docker/aufs/layers
1d6674ff835b10f76e354806e16b950f91a191d3b471236609ab13a930275e24
5dbb0cbe0148cf447b9464a358c1587be586058d9a4c9ce079320265e2bb94e7
bef7199f2ed8e86fa4ada1309cfad3089e0542fec8894690529e4c04a7ca2d73
ebf814eccfe98f2704660ca1d844e4348db3b5ccc637eb905d4818fbfb00a06a
```

The directory names do not correspond to the layer IDs (this has been true since Docker 1.10).

Now imagine that you have two different Dockerfiles. You use the first one to create an image called `acme/my-base-image:1.0`.

```
FROM ubuntu:16.10
COPY . /app
```

The second one is based on `acme/my-base-image:1.0`, but has some additional layers:

```
FROM acme/my-base-image:1.0
CMD /app/hello.sh
```

The second image contains all the layers from the first image, plus a new layer with the `CMD` instruction, and a read-write container layer. Docker already has all the layers from the first image, so it does not need to pull them again. The two images share any layers they have in common.

If you build images from the two Dockerfiles, you can use `docker image ls` and `docker history` commands to verify that the cryptographic IDs of the shared layers are the same.

1. Make a new directory `cow-test/` and change into it.
2. Within `cow-test/`, create a new file with the following contents:

   ```
   #!/bin/sh
   echo "Hello world"
   ```

   Save the file, and make it executable:

   ```
   chmod +x hello.sh
   ```

3. Copy the contents of the first Dockerfile above into a new file called `Dockerfile.base`.
4. Copy the contents of the second Dockerfile above into a new file called `Dockerfile`.
5. Within the `cow-test/` directory, build the first image. Don't forget to include the final `.` in the command. That sets the `PATH`, which tells Docker where to look for any files that need to be added to the image.

   ```
   $ docker build -t acme/my-base-image:1.0 -f Dockerfile.base .

   Sending build context to Docker daemon  4.096kB
   Step 1/2 : FROM ubuntu:16.10
    ---> 31005225a745
   Step 2/2 : COPY . /app
    ---> Using cache
    ---> bd09118bcef6
   Successfully built bd09118bcef6
   Successfully tagged acme/my-base-image:1.0
   ```

6. Build the second image.

```
$ docker build -t acme/my-final-image:1.0 -f Dockerfile .

Sending build context to Docker daemon  4.096kB
Step 1/2 : FROM acme/my-base-image:1.0
 ---> bd09118bcef6
Step 2/2 : CMD /app/hello.sh
 ---> Running in a07b694759ba
 ---> dbf995fc07ff
Removing intermediate container a07b694759ba
Successfully built dbf995fc07ff
Successfully tagged acme/my-final-image:1.0
```

7. Check out the sizes of the images:

```
$ docker image ls

REPOSITORY                          TAG                 IMAG
E ID            CREATED             SIZE
acme/my-final-image                 1.0                 dbf9
95fc07ff        58 seconds ago      103MB
acme/my-base-image                  1.0                 bd09
118bcef6        3 minutes ago       103MB
```

8. Check out the layers that comprise each image:

```
$ docker history bd09118bcef6
IMAGE               CREATED             CREATED BY
                         SIZE                COMMENT
bd09118bcef6        4 minutes ago       /bin/sh -c #(nop) COPY
dir:35a7eb158c1504e...   100B
31005225a745        3 months ago        /bin/sh -c #(nop)  CMD
["/bin/bash"]            0B
<missing>           3 months ago        /bin/sh -c mkdir -p /ru
n/systemd && echo '...   7B
<missing>           3 months ago        /bin/sh -c sed -i 's/^#
\s*\(deb.*universe\...   2.78kB
<missing>           3 months ago        /bin/sh -c rm -rf /var/
lib/apt/lists/*          0B
<missing>           3 months ago        /bin/sh -c set -xe   &&
 echo '#!/bin/sh' >...   745B
<missing>           3 months ago        /bin/sh -c #(nop) ADD f
ile:eef57983bd66e3a...   103MB
```

```
$ docker history dbf995fc07ff

IMAGE               CREATED             CREATED BY
                         SIZE                COMMENT
dbf995fc07ff        3 minutes ago       /bin/sh -c #(nop)  CMD
["/bin/sh" "-c" "/a...   0B
bd09118bcef6        5 minutes ago       /bin/sh -c #(nop) COPY
dir:35a7eb158c1504e...   100B
31005225a745        3 months ago        /bin/sh -c #(nop)  CMD
["/bin/bash"]            0B
<missing>           3 months ago        /bin/sh -c mkdir -p /ru
n/systemd && echo '...   7B
<missing>           3 months ago        /bin/sh -c sed -i 's/^#
\s*\(deb.*universe\...   2.78kB
<missing>           3 months ago        /bin/sh -c rm -rf /var/
lib/apt/lists/*          0B
<missing>           3 months ago        /bin/sh -c set -xe   &&
 echo '#!/bin/sh' >...   745B
<missing>           3 months ago        /bin/sh -c #(nop) ADD f
ile:eef57983bd66e3a...   103MB
```

Notice that all the layers are identical except the top layer of the second image. All the other layers are shared between the two images, and are only stored once in `/var/lib/docker/`. The new layer actually doesn't take any room at all, because it is not changing any files, but only running a command.

> Note: The `<missing>` lines in the `docker history` output indicate that those layers were built on another system and are not available locally. This can be ignored.

Copying makes containers efficient

When you start a container, a thin writable container layer is added on top of the other layers. Any changes the container makes to the filesystem are stored here. Any files the container does not change do not get copied to this writable layer. This means that the writable layer is as small as possible.

When an existing file in a container is modified, the storage driver performs a copy-on-write operation. The specifics steps involved depend on the specific storage driver. For the `aufs`, `overlay`, and `overlay2` drivers, the copy-on-write operation follows this rough sequence:

- Search through the image layers for the file to update. The process starts at the newest layer and works down to the base layer one layer at a time. When results are found, they are added to a cache to speed future operations.
- Perform a `copy_up` operation on the first copy of the file that is found, to copy the file to the container's writable layer.
- Any modifications are made to this copy of the file, and the container cannot see the read-only copy of the file that exists in the lower layer.

Btrfs, ZFS, and other drivers handle the copy-on-write differently. You can read more about the methods of these drivers later in their detailed descriptions.

Containers that write a lot of data consume more space than containers that do not. This is because most write operations consume new space in the container's thin writable top layer.

> Note: for write-heavy applications, you should not store the data in the container. Instead, use Docker volumes, which are independent of the running container and are designed to be efficient for I/O. In addition, volumes can be shared among containers and do not increase the size of your container's writable layer.

A `copy_up` operation can incur a noticeable performance overhead. This overhead is different depending on which storage driver is in use. Large files, lots of layers, and deep directory trees can make the impact more noticeable. This is mitigated by the fact that each `copy_up` operation only occurs the first time a given file is modified.

To verify the way that copy-on-write works, the following procedures spins up 5 containers based on the `acme/my-final-image:1.0` image we built earlier and examines how much room they take up.

> Note: This procedure doesn't work on Docker for Mac or Docker for Windows.

1. From a terminal on your Docker host, run the following `docker run` commands. The strings at the end are the IDs of each container.

```
$ docker run -dit --name my_container_1 acme/my-final-image:1.0
 bash \
  && docker run -dit --name my_container_2 acme/my-final-image:
1.0 bash \
  && docker run -dit --name my_container_3 acme/my-final-image:
1.0 bash \
  && docker run -dit --name my_container_4 acme/my-final-image:
1.0 bash \
  && docker run -dit --name my_container_5 acme/my-final-image:
1.0 bash

  c36785c423ec7e0422b2af7364a7ba4da6146cbba7981a0951fcc3fa0430c
409
  dcad7101795e4206e637d9358a818e5c32e13b349e62b00bf05cd5a4343ea
513
  1e7264576d78a3134fbaf7829bc24b1d96017cf2bc046b7cd8b08b5775c33
d0c
  38fa94212a419a082e6a6b87a8e2ec4a44dd327d7069b85892a707e3fc818
544
  1a174fc216cccf18ec7d4fe14e008e30130b11ede0f0f94a87982e310cf2e
765
```

2. Run the `docker ps` command to verify the 5 containers are running.

```
CONTAINER ID      IMAGE                     COMMAND     CREATED
              STATUS              PORTS      NAMES
1a174fc216cc      acme/my-final-image:1.0   "bash"      About a
 minute ago   Up About a minute              my_container_5
38fa94212a41      acme/my-final-image:1.0   "bash"      About a
 minute ago   Up About a minute              my_container_4
1e7264576d78      acme/my-final-image:1.0   "bash"      About a
 minute ago   Up About a minute              my_container_3
dcad7101795e      acme/my-final-image:1.0   "bash"      About a
 minute ago   Up About a minute              my_container_2
c36785c423ec      acme/my-final-image:1.0   "bash"      About a
 minute ago   Up About a minute              my_container_1
```

3. List the contents of the local storage area.

```
$ sudo ls /var/lib/docker/containers

1a174fc216cccf18ec7d4fe14e008e30130b11ede0f0f94a87982e310cf2e76
5
1e7264576d78a3134fbaf7829bc24b1d96017cf2bc046b7cd8b08b5775c33d0
c
38fa94212a419a082e6a6b87a8e2ec4a44dd327d7069b85892a707e3fc81854
4
c36785c423ec7e0422b2af7364a7ba4da6146cbba7981a0951fcc3fa0430c40
9
dcad7101795e4206e637d9358a818e5c32e13b349e62b00bf05cd5a4343ea51
3
```

4. Now check out their sizes:

```
$ sudo du -sh /var/lib/docker/containers/*

32K  /var/lib/docker/containers/1a174fc216cccf18ec7d4fe14e008e3
0130b11ede0f0f94a87982e310cf2e765
32K  /var/lib/docker/containers/1e7264576d78a3134fbaf7829bc24b1
d96017cf2bc046b7cd8b08b5775c33d0c
32K  /var/lib/docker/containers/38fa94212a419a082e6a6b87a8e2ec4
a44dd327d7069b85892a707e3fc818544
32K  /var/lib/docker/containers/c36785c423ec7e0422b2af7364a7ba4
da6146cbba7981a0951fcc3fa0430c409
32K  /var/lib/docker/containers/dcad7101795e4206e637d9358a818e5
c32e13b349e62b00bf05cd5a4343ea513
```

   Each of these containers only takes up 32k of space on the filesystem.

Not only does copy-on-write save space, but it also reduces start-up time. When you start a container (or multiple containers from the same image), Docker only needs to create the thin writable container layer.

If Docker had to make an entire copy of the underlying image stack each time it started a new container, container start times and disk space used would be significantly increased. This would be similar to the way that virtual machines work, with one or more virtual disks per virtual machine.

Related information

- Volumes (https://docs.docker.com/storage/volumes/)

- Select a storage driver (https://docs.docker.com/storage/storagedriver/select-storage-driver/)

container (https://docs.docker.com/glossary/?term=container), storage (https://docs.docker.com/glossary/?term=storage), driver (https://docs.docker.com/glossary/?term=driver), AUFS (https://docs.docker.com/glossary/?term=AUFS), btfs (https://docs.docker.com/glossary/?term=btfs), devicemapper (https://docs.docker.com/glossary/?term=devicemapper), zvfs (https://docs.docker.com/glossary/?term=zvfs)

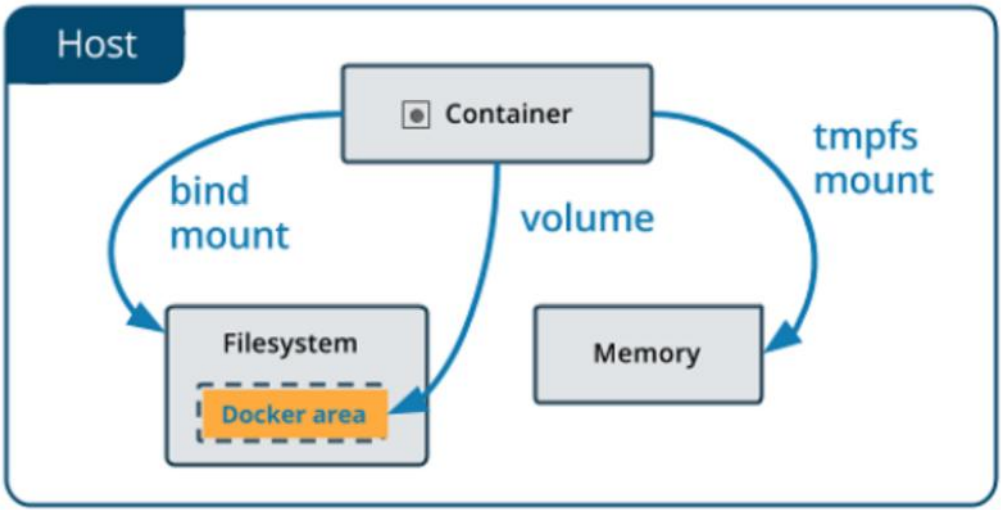
Use volumes

Estimated reading time: 15 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 `-Ấ` 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 `-Ấ` 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 `-Ấ` 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 (/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-PAT
> H>,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 `-Ấ` syntax where possible, and `--mount` is presented first.

Differences between `-Ấ` and `--mount` behavior

As opposed to bind mounts, all options for volumes are available for both `--mount` and `-Ấ` 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 `-Ă` 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 `-Ẑ` 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 `-Ấ` examples have the same result.

`--mount` `-v`

```
$ docker run -d \
  --name=nginxtest \
  --mount source=nginx-vol,destination=/usr/share/nginx/html,readonl
y \
  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.

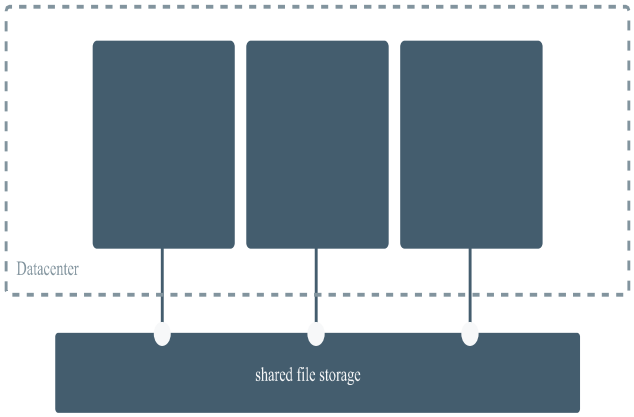
```
$ docker container stop nginxtest

$ 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 `-` .

```
$ docker run -d \
  --name sshfs-container \
  --volume-driver vieux/sshfs \
  --mount src=sshvolume,target=/app,volume-opt=sshcmd=test@node2:/ho
me/test,volume-opt=password=testpassword \
  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, 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 ta
r 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 b
ash -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 form 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
```

Next steps

- Learn about bind mounts (https://docs.docker.com/storage/bind-mounts/).
- Learn about tmpfs mounts (https://docs.docker.com/storage/tmpfs/).
- Learn about storage drivers (https://docs.docker.com/storage/storagedriver/).

storage (https://docs.docker.com/glossary/?term=storage), persistence (https://docs.docker.com/glossary/?term=persistence), data persistence (https://docs.docker.com/glossary/?term=data persistence), volumes (https://docs.docker.com/glossary/?term=volumes)

Prune unused Docker objects

Estimated reading time: 5 minutes

Docker takes a conservative approach to cleaning up unused objects (often referred to as "garbage collection"), such as images, containers, volumes, and networks: these objects are generally not removed unless you explicitly ask Docker to do so. This can cause Docker to use extra disk space. For each type of object, Docker provides a `prune` command. In addition, you can use `docker system prune` to clean up multiple types of objects at once. This topic shows how to use these `prune` commands.

Prune images

The `docker image prune` command allows you to clean up unused images. By default, `docker image prune` only cleans up *dangling* images. A dangling image is one that is not tagged and is not referenced by any container. To remove dangling images:

```
$ docker image prune

WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N] y
```

To remove all images which are not used by existing containers, use the `-a` flag:

```
$ docker image prune -a

WARNING! This will remove all images without at least one container
associated to them.
Are you sure you want to continue? [y/N] y
```

By default, you are prompted to continue. To bypass the prompt, use the `-f` or `--force` flag.

You can limit which images are pruned using filtering expressions with the `--filter` flag. For example, to only consider images created more than 24 hours ago:

```
$ docker image prune -f --filter "until=24h"
```

Other filtering expressions are available. See the `docker image prune` reference (https://docs.docker.com/engine/reference/commandline/image_prune/) for more examples.

Prune containers

When you stop a container, it is not automatically removed unless you started it with the `--rm` flag. To see all containers on the Docker host, including stopped containers, use `docker ps -f` . You may be surprised how many containers exist, especially on a development system! A stopped container's writable layers still take up disk space. To clean this up, you can use the `docker container prune` command.

```
$ docker container prune

WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
```

By default, you are prompted to continue. To bypass the prompt, use the `-f` or `--force` flag.

By default, all stopped containers are removed. You can limit the scope using the `--filter` flag. For instance, the following command only removes stopped containers older than 24 hours:

```
$ docker container prune --filter "until=24h"
```

Other filtering expressions are available. See the `docker container prune` reference (https://docs.docker.com/engine/reference/commandline/container_prune/) for more examples.

Prune volumes

Volumes can be used by one or more containers, and take up space on the Docker host. Volumes are never removed automatically, because to do so could destroy data.

```
$ docker volume prune

WARNING! This will remove all volumes not used by at least one conta
iner.
Are you sure you want to continue? [y/N] y
```

By default, you are prompted to continue. To bypass the prompt, use the `-f` or `--force` flag.

By default, all unused volumes are removed. You can limit the scope using the `--filter` flag. For instance, the following command only removes volumes which are not labelled with the `keep` label:

```
$ docker volume prune --filter "label!=keep"
```

Other filtering expressions are available. See the `docker volume prune` reference (https://docs.docker.com/engine/reference/commandline/volume_prune/) for more examples.

Prune networks

Docker networks don't take up much disk space, but they do create `iptables` rules, bridge network devices, and routing table entries. To clean these things up, you can use `docker network prune` to clean up networks which aren't used by any containers.

```
$ docker network prune

WARNING! This will remove all networks not used by at least one cont
ainer.
Are you sure you want to continue? [y/N] y
```

By default, you are prompted to continue. To bypass the prompt, use the `-f` or `--force` flag.

By default, all unused networks are removed. You can limit the scope using the `--filter` flag. For instance, the following command only removes networks older than 24 hours:

```
$ docker network prune --filter "until=24h"
```

Other filtering expressions are available. See the `docker network prune` reference (https://docs.docker.com/engine/reference/commandline/network_prune/) for more examples.

Prune everything

The `docker system prune` command is a shortcut that prunes images, containers, and networks. In Docker 17.06.0 and earlier, volumes are also pruned. In Docker 17.06.1 and higher, you must specify the `--volumes` flag for `docker system prune` to prune volumes.

```
$ docker system prune

WARNING! This will remove:
        - all stopped containers
        - all networks not used by at least one container
        - all dangling images
        - all build cache
Are you sure you want to continue? [y/N] y
```

If you are on Docker 17.06.1 or higher and want to also prune volumes, add the `--volumes` flag:

```
$ docker system prune --volumes

WARNING! This will remove:
        - all stopped containers
        - all networks not used by at least one container
        - all volumes not used by at least one container
        - all dangling images
        - all build cache
Are you sure you want to continue? [y/N] y
```

By default, you are prompted to continue. To bypass the prompt, use the `-f` or `--force` flag.

pruning (https://docs.docker.com/glossary/?term=pruning), prune (https://docs.docker.com/glossary/?term=prune), images (https://docs.docker.com/glossary/?term=images), volumes (https://docs.docker.com/glossary/?term=volumes), containers (https://docs.docker.com/glossary/?term=containers), networks (https://docs.docker.com/glossary/?term=networks), disk (https://docs.docker.com/glossary/?term=disk), administration (https://docs.docker.com/glossary/?term=administration), garbage collection (https://docs.docker.com/glossary/?term=garbage collection)

Garbage collection

Estimated reading time: 2 minutes

> These are the docs for DTR version 2.3.9
>
> To select a different version, use the selector below.
>
> 2.3.9

You can configure Docker Trusted Registry to automatically delete unused image layers, thus saving you disk space. This process is also known as garbage collection.

How DTR deletes unused layers

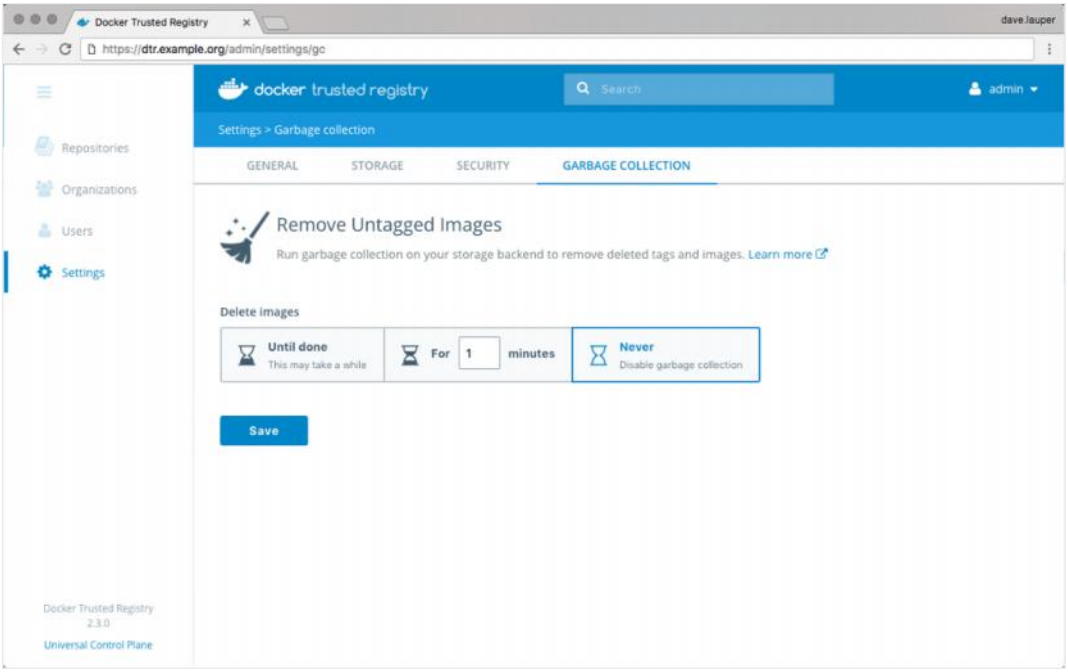
First you configure DTR to run a garbage collection job on a fixed schedule. At the scheduled time:

1. DTR becomes read-only. Images can be pulled, but pushes are not allowed.
2. DTR identifies and marks all unused image layers.
3. DTR deletes the marked image layers.

Since this process puts DTR in read-only mode and is CPU-intensive, you should run garbage collection jobs outside business peak hours.

Schedule garbage collection

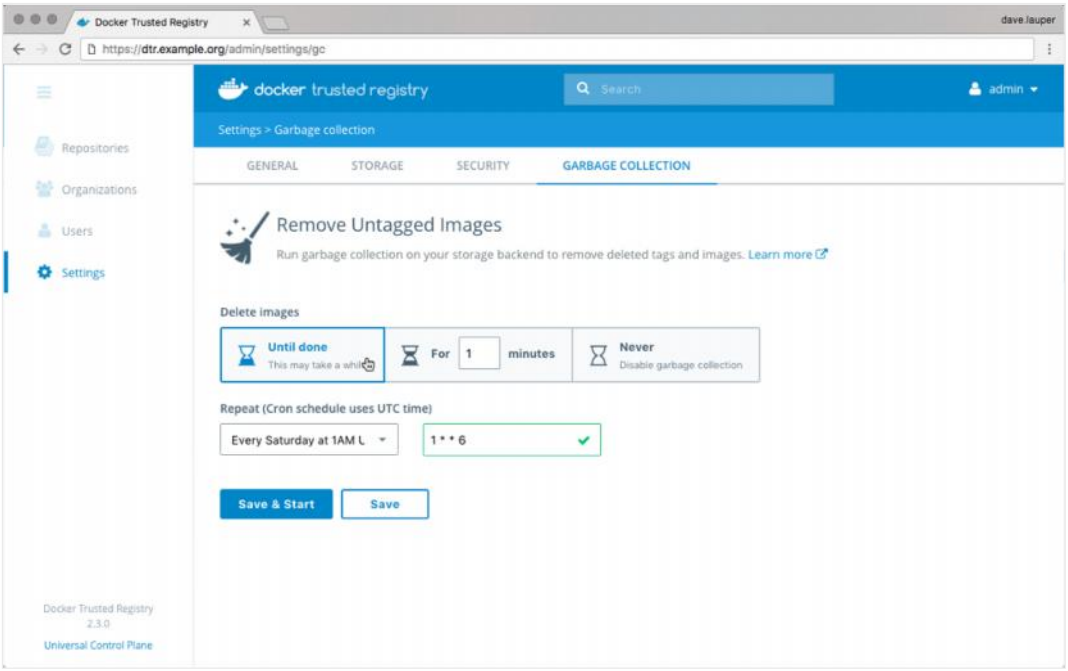
Navigate to the Settings page, and choose Garbage collection.



Select for how long the garbage collection job should run:

- Until done: Run the job until all unused image layers are deleted.
- For x minutes: Only run the garbage collection job for a maximum of x minutes at a time.
- Never: Never delete unused image layers.

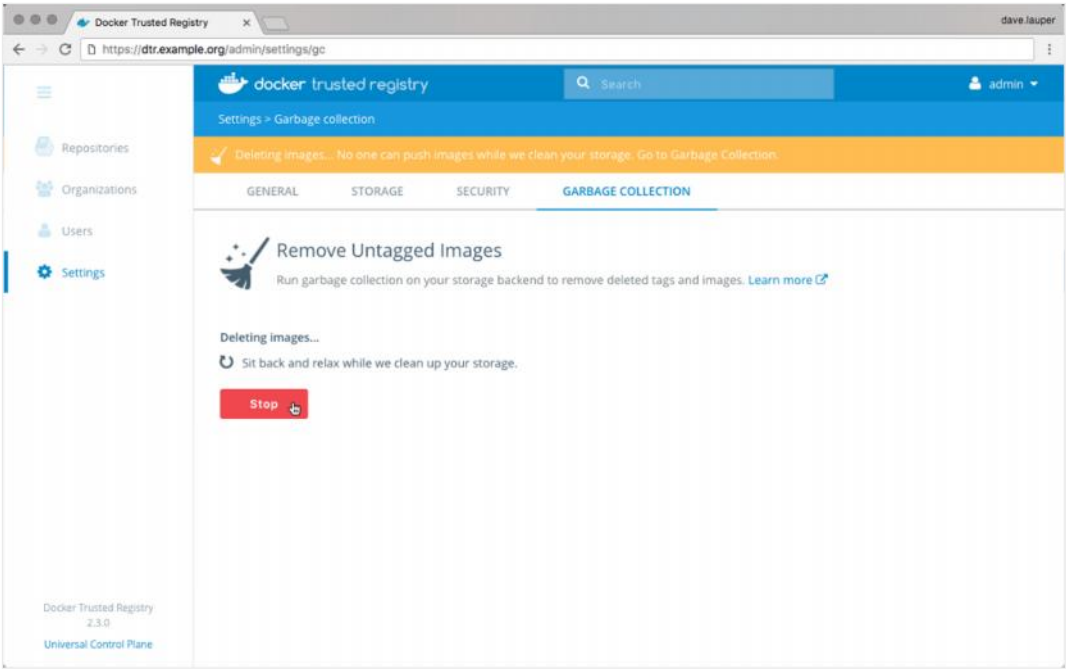
Once you select for how long to run the garbage collection job, you can configure its schedule (in UTC time) using the cron format.



Once everything is configured you can chose to Save & start to immediately run the garbage collection job, or just Save to run the job on the next scheduled interval.

Stop the garbage collection job

Once the garbage collection job starts running, a banner is displayed on the web UI explaining that users can't push images. If you're an administrator, you can click the banner to stop the garbage collection job.



Under the hood

Each image stored in DTR is made up of multiple files:

- A list of image layers that represent the image filesystem.
- A configuration file that contains the architecture of the image and other metadata.
- A manifest file containing the list of all layers and configuration file for an image.

All these files are stored in a content-addressable way in which the name of the file is the result of hashing the file's content. This means that if two image tags have exactly the same content, DTR only stores the image content once, even if the tag name is different.

As an example, if `wordpress:4.8` and `wordpress:latest` have the same content, they will only be stored once. If you delete one of these tags, the other won't be deleted.

This means that when users delete an image tag, DTR can't delete the underlying files of that image tag since it's possible that there are other tags that also use the same files.

To delete unused image layers, DTR:

1. Becomes read-only to make sure that no one can push an image, thus changing the underlying files in the filesystem.
2. Check all the manifest files and keep a record of the files that are referenced.
3. If a file is never referenced, that means that no image tag uses it, so it can be safely deleted.

Where to go next

- Deploy DTR caches (https://docs.docker.com/datacenter/dtr/2.3/guides/admin/configure/deploy-caches/)

registry (https://docs.docker.com/glossary/?term=registry), garbage collection (https://docs.docker.com/glossary/?term=garbage collection), gc (https://docs.docker.com/glossary/?term=gc), space (https://docs.docker.com/glossary/?term=space), disk space (https://docs.docker.com/glossary/?term=disk space)

Estimated reading time: 8 minutes

Use Docker Engine plugins

This document describes the Docker Engine plugins generally available in Docker Engine. To view information on plugins managed by Docker, refer to Docker Engine plugin system (https://docs.docker.com/engine/extend/).

You can extend the capabilities of the Docker Engine by loading third-party plugins. This page explains the types of plugins and provides links to several volume and network plugins for Docker.

Types of plugins

Plugins extend Docker's functionality. They come in specific types. For example, a volume plugin (https://docs.docker.com/engine/extend/plugins_volume/) might enable Docker volumes to persist across multiple Docker hosts and a network plugin (https://docs.docker.com/engine/extend/plugins_network/) might provide network plumbing.

Currently Docker supports authorization, volume and network driver plugins. In the future it will support additional plugin types.

Installing a plugin

Follow the instructions in the plugin's documentation.

Finding a plugin

The sections below provide an inexhaustive overview of available plugins.

Network plugins

Plugin	Description
Contiv Networking (https://github.com/contiv/netplugin)	An open source network plugin to provide infrastructure and security policies for a multi-tenant micro services deployment, while providing an integration to physical network for non-container workload. Contiv Networking implements the remote driver and IPAM APIs available in Docker 1.9 onwards.
Kuryr Network Plugin (https://github.com/openstack/kuryr)	A network plugin is developed as part of the OpenStack Kuryr project and implements the Docker networking (libnetwork) remote driver API by utilizing Neutron, the OpenStack networking service. It includes an IPAM driver as well.

Plugin	Description
Weave Network Plugin (https://www.weave.works/docs/net/latest/introducing-weave/)	A network plugin that creates a virtual network that connects your Docker containers - across multiple hosts or clouds and enables automatic discovery of applications. Weave networks are resilient, partition tolerant, secure and work in partially connected networks, and other adverse environments - all configured with delightful simplicity.

Volume plugins

Plugin	Description
Azure File Storage plugin (https://github.com/Azure/azurefile-dockervolumedriver)	Lets you mount Microsoft Azure File Storage (https://azure.microsoft.com/blog/azure-file-storage-now-generally-available/) shares to Docker containers as volumes using the SMB 3.0 protocol. Learn more (https://azure.microsoft.com/blog/persistent-docker-volumes-with-azure-file-storage/).
BeeGFS Volume Plugin (https://github.com/RedCoolBeans/docker-volume-beegfs)	An open source volume plugin to create persistent volumes in a BeeGFS parallel file system.
Blockbridge plugin (https://github.com/blockbridge/blockbridge-docker-volume)	A volume plugin that provides access to an extensible set of container-based persistent storage options. It supports single and multi-host Docker environments with features that include tenant isolation, automated provisioning, encryption, secure deletion, snapshots and QoS.
Contiv Volume Plugin (https://github.com/contiv/volplugin)	An open source volume plugin that provides multi-tenant, persistent, distributed storage with intent based consumption. It has support for Ceph and NFS.
Convoy plugin (https://github.com/rancher/convoy)	A volume plugin for a variety of storage back-ends including device mapper and NFS. It's a simple standalone executable written in Go and provides the framework to support vendor-specific extensions such as snapshots, backups and restore.
DigitalOcean Block Storage plugin (https://github.com/omallo/docker-volume-plugin-dostorage)	Integrates DigitalOcean's block storage solution (https://www.digitalocean.com/products/storage/) into the Docker ecosystem by automatically attaching a given block storage volume to a DigitalOcean droplet and making the contents of the volume available to Docker containers running on that droplet.
DRBD plugin (https://www.drbd.org/en/supported-projects/docker)	A volume plugin that provides highly available storage replicated by DRBD (https://www.drbd.org). Data written to the docker volume is replicated in a cluster of DRBD nodes.

Plugin	Description
Flocker plugin (https://github.com/ScatterHQ/flocker)	A volume plugin that provides multi-host portable volumes for Docker, enabling you to run databases and other stateful containers and move them around across a cluster of machines.
Fuxi Volume Plugin (https://github.com/openstack/fuxi)	A volume plugin that is developed as part of the OpenStack Kuryr project and implements the Docker volume plugin API by utilizing Cinder, the OpenStack block storage service.
gce-docker plugin (https://github.com/mcuadros/gce-docker)	A volume plugin able to attach, format and mount Google Compute persistent-disks (https://cloud.google.com/compute/docs/disks/persistent-disks).
GlusterFS plugin (https://github.com/calavera/docker-volume-glusterfs)	A volume plugin that provides multi-host volumes management for Docker using GlusterFS.
Horcrux Volume Plugin (https://github.com/muthu-r/horcrux)	A volume plugin that allows on-demand, version controlled access to your data. Horcrux is an open-source plugin, written in Go, and supports SCP, Minio (https://www.minio.io) and Amazon S3.
HPE 3Par Volume Plugin (https://github.com/hpe-storage/python-hpedockerplugin/)	A volume plugin that supports HPE 3Par and StoreVirtual iSCSI storage arrays.
Infinit volume plugin (https://infinit.sh/documentation/docker/volume-plugin)	A volume plugin that makes it easy to mount and manage Infinit volumes using Docker.
IPFS Volume Plugin (http://github.com/vdemeester/docker-volume-ipfs)	An open source volume plugin that allows using an ipfs (https://ipfs.io/) filesystem as a volume.
Keywhiz plugin (https://github.com/calavera/docker-volume-keywhiz)	A plugin that provides credentials and secret management using Keywhiz as a central repository.
Local Persist Plugin (https://github.com/CWSpear/local-persist)	A volume plugin that extends the default `local` driver's functionality by allowing you specify a mountpoint anywhere on the host, which enables the files to *always persist*, even if the volume is removed via `docker volume rm` .
NetApp Plugin (https://github.com/NetApp/netappdvp) (nDVP)	A volume plugin that provides direct integration with the Docker ecosystem for the NetApp storage portfolio. The nDVP package supports the provisioning and management of storage resources from the storage platform to Docker hosts, with a robust framework for adding additional platforms in the future.
Netshare plugin (https://github.com/ContainX/docker-volume-netshare)	A volume plugin that provides volume management for NFS 3/4, AWS EFS and CIFS file systems.

Plugin	Description
Nimble Storage Volume Plugin (https://connect.nimblestorage.com/community/app-integration/docker)	A volume plug-in that integrates with Nimble Storage Unified Flash Fabric arrays. The plug-in abstracts array volume capabilities to the Docker administrator to allow self-provisioning of secure multi-tenant volumes and clones.
OpenStorage Plugin (https://github.com/libopenstorage/openstorage)	A cluster-aware volume plugin that provides volume management for file and block storage solutions. It implements a vendor neutral specification for implementing extensions such as CoS, encryption, and snapshots. It has example drivers based on FUSE, NFS, NBD and EBS to name a few.
Portworx Volume Plugin (https://github.com/portworx/px-dev)	A volume plugin that turns any server into a scale-out converged compute/storage node, providing container granular storage and highly available volumes across any node, using a shared-nothing storage backend that works with any docker scheduler.
Quobyte Volume Plugin (https://github.com/quobyte/docker-volume)	A volume plugin that connects Docker to Quobyte (http://www.quobyte.com/containers)'s data center file system, a general-purpose scalable and fault-tolerant storage platform.
REX-Ray plugin (https://github.com/emccode/rexray)	A volume plugin which is written in Go and provides advanced storage functionality for many platforms including VirtualBox, EC2, Google Compute Engine, OpenStack, and EMC.
Virtuozzo Storage and Ploop plugin (https://github.com/virtuozzo/docker-volume-ploop)	A volume plugin with support for Virtuozzo Storage distributed cloud file system as well as ploop devices.
VMware vSphere Storage Plugin (https://github.com/vmware/docker-volume-vsphere)	Docker Volume Driver for vSphere enables customers to address persistent storage requirements for Docker containers in vSphere environments.

Authorization plugins

Plugin	Description
Casbin AuthZ Plugin (https://github.com/casbin/casbin-authz-plugin)	An authorization plugin based on Casbin (https://github.com/casbin/casbin), which supports access control models like ACL, RBAC, ABAC. The access control model can be customized. The policy can be persisted into file or DB.
HBM plugin (https://github.com/kassisol/hbm)	An authorization plugin that prevents from executing commands with certains parameters.

Plugin	Description
Twistlock AuthZ Broker (https://github.com/twistlock/authz)	A basic extendable authorization plugin that runs directly on the host or inside a container. This plugin allows you to define user policies that it evaluates during authorization. Basic authorization is provided if Docker daemon is started with the --tlsverify flag (username is extracted from the certificate common name).

Troubleshooting a plugin

If you are having problems with Docker after loading a plugin, ask the authors of the plugin for help. The Docker team may not be able to assist you.

Writing a plugin

If you are interested in writing a plugin for Docker, or seeing how they work under the hood, see the docker plugins reference (https://docs.docker.com/engine/extend/plugin_api/).

Examples (https://docs.docker.com/glossary/?term=Examples), Usage (https://docs.docker.com/glossary/?term=Usage), plugins (https://docs.docker.com/glossary/?term=plugins), docker (https://docs.docker.com/glossary/?term=docker), documentation (https://docs.docker.com/glossary/?term=documentation), user guide (https://docs.docker.com/glossary/?term=user guide)