Check whether Docker is running

The operating-system independent way to check whether Docker is running is to ask Docker, using the docker info command.

You can also use operating system utilities, such as sudo systemctl is-active docker or sudo status docker or sudo service docker status, or checking the service status using Windows utilities.

Start the daemon manually

If you don’t want to use a system utility to manage the Docker daemon, or just want to test things out, you can manually run it using the dockerd command. You may need to use sudo, depending on your operating system configuration.

When you start Docker this way, it runs in the foreground and sends its logs directly to your terminal.

$ dockerd

INFO[0000] +job init\_networkdriver()

INFO[0000] +job serveapi(unix:///var/run/docker.sock)

INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)

To stop Docker when you have started it manually, issue a Ctrl+C in your terminal.

# Start containers automatically

Docker provides [restart policies](https://docs.docker.com/engine/reference/run/#restart-policies---restart) to control whether your containers start automatically when they exit, or when Docker restarts. Restart policies ensure that linked containers are started in the correct order. Docker recommends that you use restart policies, and avoid using process managers to start containers.

Restart policies are different from the --live-restore flag of the dockerd command. Using --live-restore allows you to keep your containers running during a Docker upgrade, though networking and user input are interrupted.

## Use a restart policy

To configure the restart policy for a container, use the --restart flag when using the docker run command. The value of the --restart flag can be any of the following:

| **Flag** | **Description** |
| --- | --- |
| no | Do not automatically restart the container. (the default) |
| on-failure | Restart the container if it exits due to an error, which manifests as a non-zero exit code. |
| unless-stopped | Restart the container unless it is explicitly stopped or Docker itself is stopped or restarted. |
| always | Always restart the container if it stops. |

The following example starts a Redis container and configures it to always restart unless it is explicitly stopped or Docker is restarted.

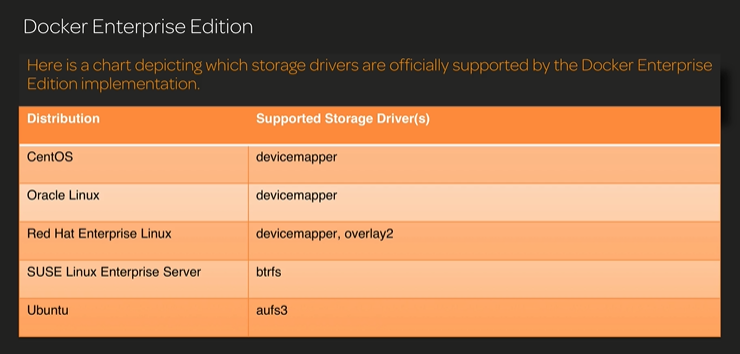
$ docker run -dit --restart unless-stopped redis

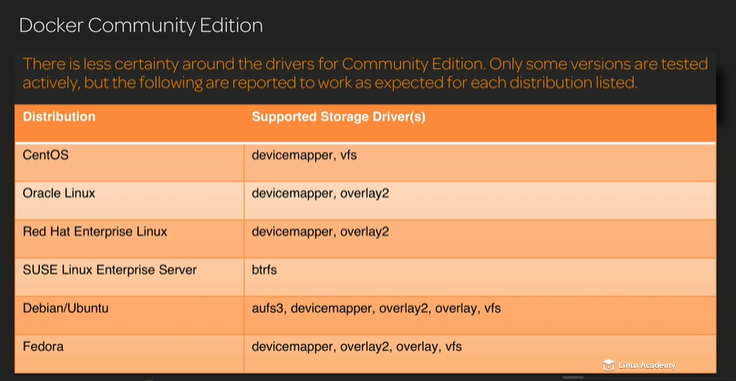
### Restart policy details

Keep the following in mind when using restart policies:

* A restart policy only takes effect after a container starts successfully. In this case, starting successfully means that the container is up for at least 10 seconds and Docker has started monitoring it. This prevents a container which does not start at all from going into a restart loop.
* If you manually stop a container, its restart policy is ignored until the Docker daemon restarts or the container is manually restarted. This is another attempt to prevent a restart loop.
* Restart policies only apply to containers. Restart policies for swarm services are configured differently. See the [flags related to service restart](https://docs.docker.com/engine/reference/commandline/service_create/).

Use the Device Mapper storage driver





## Configure Docker with the devicemapper storage driver

Before following these procedures, you must first meet all the [prerequisites](https://docs.docker.com/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](https://docs.docker.com/storage/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-for-production).

1. Stop Docker.
2. $ sudo systemctl stop docker
3. Edit /etc/docker/daemon.json. If it does not yet exist, create it. Assuming that the file was empty, add the following contents.
4. {
5. "storage-driver": "devicemapper"
6. }

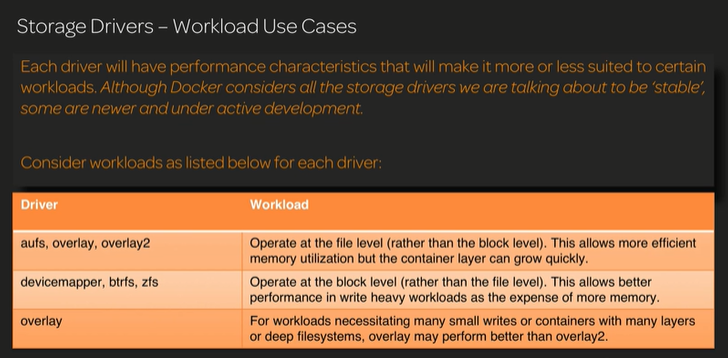
See all storage options for each storage driver:

* + [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.

1. Start Docker.
2. $ sudo systemctl start docker
3. Verify that the daemon is using the devicemapper storage driver. Use the docker info command and look for Storage Driver.
4. $ docker info
5. Containers: 0
6. Running: 0
7. Paused: 0
8. Stopped: 0
9. Images: 0
10. Server Version: 17.03.1-ce
11. Storage Driver: devicemapper
12. Pool Name: docker-202:1-8413957-pool
13. Pool Blocksize: 65.54 kB
14. Base Device Size: 10.74 GB
15. Backing Filesystem: xfs
16. Data file: /dev/loop0
17. Metadata file: /dev/loop1
18. Data Space Used: 11.8 MB
19. Data Space Total: 107.4 GB
20. Data Space Available: 7.44 GB
21. Metadata Space Used: 581.6 kB
22. Metadata Space Total: 2.147 GB
23. Metadata Space Available: 2.147 GB
24. Thin Pool Minimum Free Space: 10.74 GB
25. Udev Sync Supported: true
26. Deferred Removal Enabled: false
27. Deferred Deletion Enabled: false
28. Deferred Deleted Device Count: 0
29. Data loop file: /var/lib/docker/devicemapper/data
30. Metadata loop file: /var/lib/docker/devicemapper/metadata
31. Library Version: 1.02.135-RHEL7 (2016-11-16)
32. <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](https://docs.docker.com/storage/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-for-production).



# Configure logging drivers

*Estimated reading time:* 6 minutes

Docker includes multiple logging mechanisms to help you [get information from running containers and services](https://docs.docker.com/engine/admin/logging/view_container_logs/). These mechanisms are called logging drivers.

Each Docker daemon has a default logging driver, which each container uses unless you configure it to use a different logging driver.

In addition to using the logging drivers included with Docker, you can also implement and use [logging driver plugins](https://docs.docker.com/engine/admin/logging/plugins/). Logging driver plugins are available in Docker 17.05 and higher.

# About storage drivers

To use storage drivers effectively, it’s important to know how Docker builds and stores images, 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 stops, and both read and write speeds are low.

[Learn how to use volumes](https://docs.docker.com/storage/) to persist data and improved 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

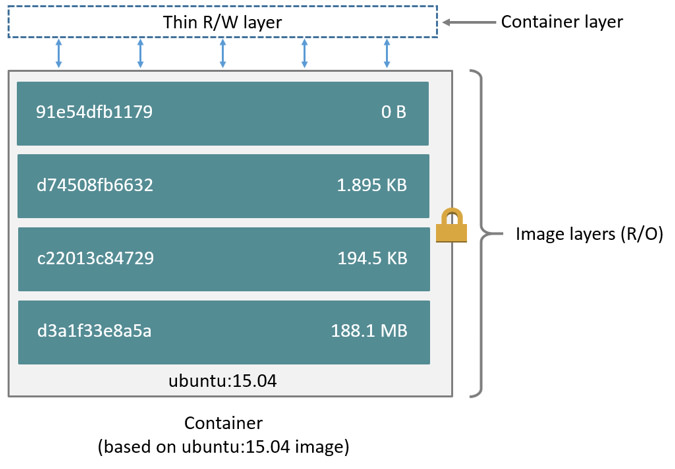
COPY . /app

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.

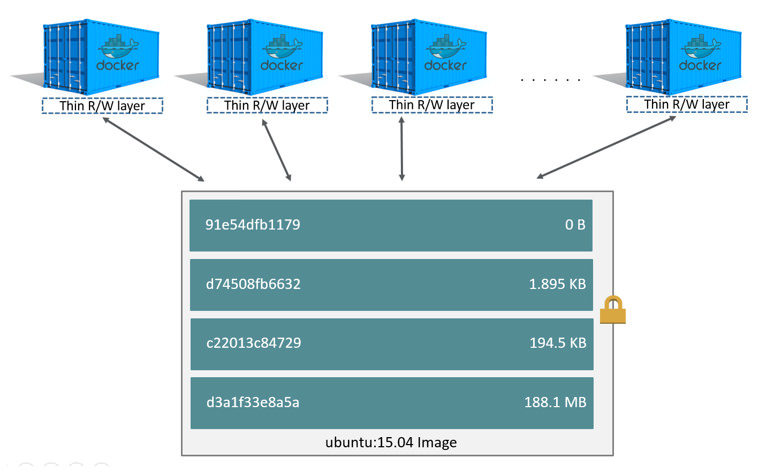


A storage driver handles the details about the way these layers interact with each other. Different storage drivers are available, which have advantages and disadvantages in different situations.

## 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.

$ docker pull ubuntu:15.04

15.04: Pulling from library/ubuntu

1ba8ac955b97: Pull complete

f157c4e5ede7: Pull complete

0b7e98f84c4c: Pull complete

a3ed95caeb02: Pull complete

Digest: sha256:5e279a9df07990286cce22e1b0f5b0490629ca6d187698746ae5e28e604a640e

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 aufs, which is the default storage driver:

$ ls /var/lib/docker/aufs/layers

1d6674ff835b10f76e354806e16b950f91a191d3b471236609ab13a930275e24

5dbb0cbe0148cf447b9464a358c1587be586058d9a4c9ce079320265e2bb94e7

bef7199f2ed8e86fa4ada1309cfad3089e0542fec8894690529e4c04a7ca2d73

ebf814eccfe98f2704660ca1d844e4348db3b5ccc637eb905d4818fbfb00a06a

## Configure the default logging driver

To configure the Docker daemon to default to a specific logging driver, set the value of log-driver to the name of the logging driver in the daemon.json file, which is located in /etc/docker/ on Linux hosts or C:\ProgramData\docker\config\ on Windows server hosts. The default logging driver is json-file. The following example explicitly sets the default logging driver to syslog:

{

"log-driver": "syslog"

}

If the logging driver has configurable options, you can set them in the daemon.json file as a JSON array with the key log-opts. The following example sets two configurable options on the json-file logging driver:

{

"log-driver": "json-file",

"log-opts": {

"labels": "production\_status",

"env": "os,customer"

}

}

If you do not specify a logging driver, the default is json-file. Thus, the default output for commands such as docker inspect <CONTAINER> is JSON.

To find the current default logging driver for the Docker daemon, run docker info and search for Logging Driver. You can use the following command on Linux, macOS, or PowerShell on Windows:

$ docker info | grep 'Logging Driver'

Logging Driver: json-file

## Configure the logging driver for a container

When you start a container, you can configure it to use a different logging driver than the Docker daemon’s default, using the --log-driver flag. If the logging driver has configurable options, you can set them using one or more instances of the --log-opt <NAME>=<VALUE> flag. Even if the container uses the default logging driver, it can use different configurable options.

The following example starts an Alpine container with the none logging driver.

$ docker run -it --log-driver none alpine ash

To find the current logging driver for a running container, if the daemon is using the json-file logging driver, run the following docker inspect command, substituting the container name or ID for <CONTAINER>:

$ docker inspect -f '{{.HostConfig.LogConfig.Type}}' <CONTAINER>

json-file

## Supported logging drivers

The following logging drivers are supported. See the link to each driver’s documentation for its configurable options, if applicable. If you are using [logging driver plugins](https://docs.docker.com/engine/admin/logging/plugins/), you may see more options.

| **Driver** | **Description** |
| --- | --- |
| none | No logs are available for the container and docker logs does not return any output. |
| [json-file](https://docs.docker.com/config/containers/logging/json-file/) | The logs are formatted as JSON. The default logging driver for Docker. |
| [syslog](https://docs.docker.com/config/containers/logging/syslog/) | Writes logging messages to the syslog facility. The syslog daemon must be running on the host machine. |
| [journald](https://docs.docker.com/config/containers/logging/journald/) | Writes log messages to journald. The journald daemon must be running on the host machine. |
| [gelf](https://docs.docker.com/config/containers/logging/gelf/) | Writes log messages to a Graylog Extended Log Format (GELF) endpoint such as Graylog or Logstash. |
| [fluentd](https://docs.docker.com/config/containers/logging/fluentd/) | Writes log messages to fluentd (forward input). The fluentd daemon must be running on the host machine. |
| [awslogs](https://docs.docker.com/config/containers/logging/awslogs/) | Writes log messages to Amazon CloudWatch Logs. |
| [splunk](https://docs.docker.com/config/containers/logging/splunk/) | Writes log messages to splunk using the HTTP Event Collector. |
| [etwlogs](https://docs.docker.com/config/containers/logging/etwlogs/) | Writes log messages as Event Tracing for Windows (ETW) events. Only available on Windows platforms. |
| [gcplogs](https://docs.docker.com/config/containers/logging/gcplogs/) | Writes log messages to Google Cloud Platform (GCP) Logging. |
| [logentries](https://docs.docker.com/config/containers/logging/logentries/) | Writes log messages to Rapid7 Logentries. |

## Limitations of logging drivers

The docker logs command is not available for drivers other than json-file and journald.

#This is a sample Image

FROM ubuntu

MAINTAINER demousr@gmail.com

RUN apt-get update

RUN apt-get install –y nginx

CMD [“echo”,”Image created”]

FROM ubuntu

RUN apt-get update

RUN apt-get install –y apache2

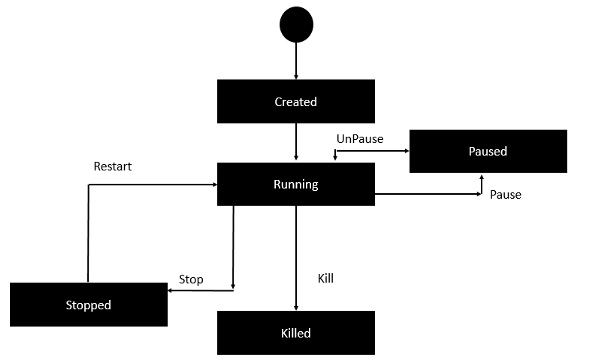
RUN apt-get install –y apache2-utils

RUN apt-get clean

EXPOSE 80 CMD [“apache2ctl”, “-D”, “FOREGROUND”]

## **Docker – Container Lifecycle**

The following illustration explains the entire lifecycle of a Docker container.



* Initially, the Docker container will be in the **created** state.
* Then the Docker container goes into the running state when the Docker **run** command is used.
* The Docker **kill** command is used to kill an existing Docker container.
* The Docker **pause** command is used to pause an existing Docker container.
* The Docker **stop** command is used to pause an existing Docker container.
* The Docker **run** command is used to put a container back from a **stopped** state to a **running** state.