Create a swarm

Estimated reading time: 2 minutes

After you complete the tutorial setup (https://docs.docker.com/engine/swarm/swarm-tutorial/) steps, you're ready to create a swarm. Make sure the Docker Engine daemon is started on the host machines.

1. Open a terminal and ssh into the machine where you want to run your manager node. This tutorial uses a machine named `manager1`. If you use Docker Machine, you can connect to it via SSH using the following command:

   ```
   $ docker-machine ssh manager1
   ```

2. Run the following command to create a new swarm:

   ```
   $ docker swarm init --advertise-addr <MANAGER-IP>
   ```

   > Note: If you are using Docker for Mac or Docker for Windows to test single-node swarm, simply run `docker swarm init` with no arguments. There is no need to specify `--advertise-addr` in this case. To learn more, see the topic on how to Use Docker for Mac or Docker for Windows (https://docs.docker.com/engine/swarm/swarm-tutorial/#use-docker-for-mac-or-docker-for-windows) with Swarm.

   In the tutorial, the following command creates a swarm on the `manager1` machine:

```
$ docker swarm init --advertise-addr 192.168.99.100
Swarm initialized: current node (dxn1zf6l61qsb1josjja83ngz) is
now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39
trti4wxv-8vxv8rssmk743ojnwacrr2e7c \
    192.168.99.100:2377

To add a manager to this swarm, run 'docker swarm join-token ma
nager' and follow the instructions.
```

The `--advertise-addr` flag configures the manager node to publish its address as `192.168.99.100`. The other nodes in the swarm must be able to access the manager at the IP address.

The output includes the commands to join new nodes to the swarm. Nodes will join as managers or workers depending on the value for the `--token` flag.

3. Run `docker info` to view the current state of the swarm:

```
$ docker info

Containers: 2
Running: 0
Paused: 0
Stopped: 2
  ...snip...
Swarm: active
  NodeID: dxn1zf6l61qsb1josjja83ngz
  Is Manager: true
  Managers: 1
  Nodes: 1
  ...snip...
```

4. Run the `docker node ls` command to view information about nodes:

```
$ docker node ls

ID                           HOSTNAME  STATUS  AVAILABILITY  MA
NAGER STATUS
dxn1zf6l61qsb1josjja83ngz *  manager1  Ready   Active        Le
ader
```

The `*` next to the node ID indicates that you're currently connected on this node.

Docker Engine swarm mode automatically names the node for the machine host name. The tutorial covers other columns in later steps.

What's next?

In the next section of the tutorial, we add two more nodes (https://docs.docker.com/engine/swarm/swarm-tutorial/add-nodes/) to the cluster.

tutorial (https://docs.docker.com/glossary/?term=tutorial), cluster management (https://docs.docker.com/glossary/?term=cluster management), swarm mode (https://docs.docker.com/glossary/?term=swarm mode)

Add nodes to the swarm

Estimated reading time: 2 minutes

Once you've created a swarm (https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/) with a manager node, you're ready to add worker nodes.

1. Open a terminal and ssh into the machine where you want to run a worker node. This tutorial uses the name `worker1` .

2. Run the command produced by the `docker swarm init` output from the Create a swarm (https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/) tutorial step to create a worker node joined to the existing swarm:

```
$ docker swarm join \
  --token  SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39t
rti4wxv-8vxv8rssmk743ojnwacrr2e7c \
  192.168.99.100:2377

This node joined a swarm as a worker.
```

   If you don't have the command available, you can run the following command on a manager node to retrieve the join command for a worker:

```
$ docker swarm join-token worker

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39
trti4wxv-8vxv8rssmk743ojnwacrr2e7c \
    192.168.99.100:2377
```

3. Open a terminal and ssh into the machine where you want to run a second worker node. This tutorial uses the name `worker2` .

4. Run the command produced by the `docker swarm init` output from the Create a swarm (https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/) tutorial step to create a second worker node joined to the existing swarm:

   ```
   $ docker swarm join \
     --token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39tr
   ti4wxv-8vxv8rssmk743ojnwacrr2e7c \
     192.168.99.100:2377

   This node joined a swarm as a worker.
   ```

5. Open a terminal and ssh into the machine where the manager node runs and run the `docker node ls` command to see the worker nodes:

   ```
   ID                           HOSTNAME  STATUS  AVAILABILITY  MA
   NAGER STATUS
   03g1y59jwfg7cf99w4lt0f662    worker2   Ready   Active
   9j68exjopxe7wfl6yuxml7a7j    worker1   Ready   Active
   dxn1zf6l61qsb1josjja83ngz *  manager1  Ready   Active        Le
   ader
   ```

   The `MANAGER` column identifies the manager nodes in the swarm. The empty status in this column for `worker1` and `worker2` identifies them as worker nodes.

   Swarm management commands like `docker node ls` only work on manager nodes.

What’s next?

Now your swarm consists of a manager and two worker nodes. In the next step of the tutorial, you deploy a service (https://docs.docker.com/engine/swarm/swarm-tutorial/deploy-service/) to the swarm.

tutorial (https://docs.docker.com/glossary/?term=tutorial), cluster management (https://docs.docker.com/glossary/?term=cluster management), swarm (https://docs.docker.com/glossary/?term=swarm)

docker run

Estimated reading time: 32 minutes

Description

Run a command in a new container

Usage

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Options

Name, shorthand	Default	Description
`--add-host`		Add a custom host-to-IP mapping (host:ip)
`--attach , -a`		Attach to STDIN, STDOUT or STDERR
`--blkio-weight`		Block IO (relative weight), between 10 and 1000, or 0 to disable (default 0)
`--blkio-weight-device`		Block IO weight (relative device weight)
`--cap-add`		Add Linux capabilities
`--cap-drop`		Drop Linux capabilities
`--cgroup-parent`		Optional parent cgroup for the container
`--cidfile`		Write the container ID to the file
`--cpu-count`		CPU count (Windows only)
`--cpu-percent`		CPU percent (Windows only)
`--cpu-period`		Limit CPU CFS (Completely Fair Scheduler) period
`--cpu-quota`		Limit CPU CFS (Completely Fair Scheduler) quota
`--cpu-rt-period`		API 1.25+ (https://docs.docker.com/engine/api/v1.25/) Limit CPU real-time period in microseconds
`--cpu-rt-runtime`		API 1.25+ (https://docs.docker.com/engine/api/v1.25/) Limit CPU real-time runtime in microseconds
`--cpu-shares , -c`		CPU shares (relative weight)

Name, shorthand	Default	Description
`--cpus`		API 1.25+ (https://docs.docker.com/engine/api/v1.25/) Number of CPUs
`--cpuset-cpus`		CPUs in which to allow execution (0-3, 0,1)
`--cpuset-mems`		MEMs in which to allow execution (0-3, 0,1)
`--detach` , `-d`		Run container in background and print container ID
`--detach-keys`		Override the key sequence for detaching a container
`--device`		Add a host device to the container
`--device-cgroup-rule`		Add a rule to the cgroup allowed devices list
`--device-read-bps`		Limit read rate (bytes per second) from a device
`--device-read-iops`		Limit read rate (IO per second) from a device
`--device-write-bps`		Limit write rate (bytes per second) to a device
`--device-write-iops`		Limit write rate (IO per second) to a device
`--disable-content-trust`	`true`	Skip image verification
`--dns`		Set custom DNS servers
`--dns-opt`		Set DNS options
`--dns-option`		Set DNS options
`--dns-search`		Set custom DNS search domains
`--entrypoint`		Overwrite the default ENTRYPOINT of the image
`--env` , `-e`		Set environment variables
`--env-file`		Read in a file of environment variables
`--expose`		Expose a port or a range of ports
`--group-add`		Add additional groups to join
`--health-cmd`		Command to run to check health
`--health-interval`		Time between running the check (ms|s|m|h) (default 0s)
`--health-retries`		Consecutive failures needed to report unhealthy
`--health-start-period`		API 1.29+ (https://docs.docker.com/engine/api/v1.29/) Start period for the container to initialize before starting health-retries countdown (ms|s|m|h) (default 0s)
`--health-timeout`		Maximum time to allow one check to run (ms|s|m|h) (default 0s)

Name, shorthand	Default	Description
`--help`		Print usage
`--hostname , -h`		Container host name
`--init`		API 1.25+ (https://docs.docker.com/engine/api/v1.25/) Run an init inside the container that forwards signals and reaps processes
`--interactive , -i`		Keep STDIN open even if not attached
`--io-maxbandwidth`		Maximum IO bandwidth limit for the system drive (Windows only)
`--io-maxiops`		Maximum IOps limit for the system drive (Windows only)
`--ip`		IPv4 address (e.g., 172.30.100.104)
`--ip6`		IPv6 address (e.g., 2001:db8::33)
`--ipc`		IPC mode to use
`--isolation`		Container isolation technology
`--kernel-memory`		Kernel memory limit
`--label , -l`		Set meta data on a container
`--label-file`		Read in a line delimited file of labels
`--link`		Add link to another container
`--link-local-ip`		Container IPv4/IPv6 link-local addresses
`--log-driver`		Logging driver for the container
`--log-opt`		Log driver options
`--mac-address`		Container MAC address (e.g., 92:d0:c6:0a:29:33)
`--memory , -m`		Memory limit
`--memory-reservation`		Memory soft limit
`--memory-swap`		Swap limit equal to memory plus swap: '-1' to enable unlimited swap
`--memory-swappiness`	`-1`	Tune container memory swappiness (0 to 100)
`--mount`		Attach a filesystem mount to the container
`--name`		Assign a name to the container
`--net`		Connect a container to a network
`--net-alias`		Add network-scoped alias for the container
`--network`		Connect a container to a network

Name, shorthand	Default	Description
`--network-alias`		Add network-scoped alias for the container
`--no-healthcheck`		Disable any container-specified HEALTHCHECK
`--oom-kill-disable`		Disable OOM Killer
`--oom-score-adj`		Tune host's OOM preferences (-1000 to 1000)
`--pid`		PID namespace to use
`--pids-limit`		Tune container pids limit (set -1 for unlimited)
`--platform`		experimental (daemon) (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file) API 1.32+ (https://docs.docker.com/engine/api/v1.32/) Set platform if server is multi-platform capable
`--privileged`		Give extended privileges to this container
`--publish , -p`		Publish a container's port(s) to the host
`--publish-all , -P`		Publish all exposed ports to random ports
`--read-only`		Mount the container's root filesystem as read only
`--restart`	`no`	Restart policy to apply when a container exits
`--rm`		Automatically remove the container when it exits
`--runtime`		Runtime to use for this container
`--security-opt`		Security Options
`--shm-size`		Size of /dev/shm
`--sig-proxy`	`true`	Proxy received signals to the process
`--stop-signal`	`SIGTERM`	Signal to stop a container
`--stop-timeout`		API 1.25+ (https://docs.docker.com/engine/api/v1.25/) Timeout (in seconds) to stop a container
`--storage-opt`		Storage driver options for the container
`--sysctl`		Sysctl options
`--tmpfs`		Mount a tmpfs directory
`--tty , -t`		Allocate a pseudo-TTY
`--ulimit`		Ulimit options
`--user , -u`		Username or UID (format: <name|uid>[:<group|gid>])
`--userns`		User namespace to use

Name, shorthand	Default	Description
`--uts`		UTS namespace to use
`--volume , -v`		Bind mount a volume
`--volume-driver`		Optional volume driver for the container
`--volumes-from`		Mount volumes from the specified container(s)
`--workdir , -w`		Working directory inside the container

Parent command

Command	Description
docker (https://docs.docker.com/engine/reference/commandline/docker)	The base command for the Docker CLI.

Extended description

The `docker run` command first `creates` a writeable container layer over the specified image, and then `starts` it using the specified command. That is, `docker run` is equivalent to the API `/containers/create` then `/containers/(id)/start` . A stopped container can be restarted with all its previous changes intact using `docker start` . See `docker ps -a` to view a list of all containers.

The `docker run` command can be used in combination with `docker commit` to *change the command that a container runs* (https://docs.docker.com/engine/reference/commandline/commit/). There is additional detailed information about `docker run` in the Docker run reference (https://docs.docker.com/engine/reference/run/).

For information on connecting a container to a network, see the "*Docker network overview*" (https://docs.docker.com/engine/userguide/networking/).

Examples

Assign name and allocate pseudo-TTY (--name, -it)

```
$ docker run --name test -it debian

root@d6c0fe130dba:/# exit 13
$ echo $?
13
$ docker ps -a | grep test
d6c0fe130dba        debian:7            "/bin/bash"         26 seconds ago      Exited (13) 17 s
econds ago                         test
```

This example runs a container named `test` using the `debian:latest` image. The `-it` instructs Docker to allocate a pseudo-TTY connected to the container's stdin; creating an interactive `bash` shell in the container. In the example, the `bash` shell is quit by entering `exit 13` . This exit code is passed on to the caller of `docker run` , and is recorded in the `test` container's metadata.

Capture container ID (--cidfile)

```
$ docker run --cidfile /tmp/docker_test.cid ubuntu echo "test"
```

This will create a container and print `test` to the console. The `cidfile` flag makes Docker attempt to create a new file and write the container ID to it. If the file exists already, Docker will return an error. Docker will close this file when `docker run` exits.

Full container capabilities (--privileged)

```
$ docker run -t -i --rm ubuntu bash
root@bc338942ef20:/# mount -t tmpfs none /mnt
mount: permission denied
```

This will *not* work, because by default, most potentially dangerous kernel capabilities are dropped; including `cap_sys_admin` (which is required to mount filesystems). However, the `--privileged` flag will allow it to run:

```
$ docker run -t -i --privileged ubuntu bash
root@50e3f57e16e6:/# mount -t tmpfs none /mnt
root@50e3f57e16e6:/# df -!
Filesystem      Size  Used Avail Use% Mounted on
none            1.9G     0  1.9G   0% /mnt
```

The `--privileged` flag gives *all* capabilities to the container, and it also lifts all the limitations enforced by the `device` cgroup controller. In other words, the container can then do almost everything that the host can do. This flag exists to allow special use-cases, like running Docker within Docker.

Set working directory (-w)

```
$ docker  run -w /path/to/dir/ -i -t  ubuntu pwd
```

The `-w` lets the command being executed inside directory given, here `/path/to/dir/` . If the path does not exist it is created inside the container.

Set storage driver options per container

```
$ docker run -it --storage-opt size=120G fedora /bin/bash
```

This (size) will allow to set the container rootfs size to 120G at creation time. This option is only available for the `devicemapper` , `btrfs` , `overlay2` , `windowsfilter` and `zfs` graph drivers. For the `devicemapper` , `btrfs` , `windowsfilter` and `zfs` graph drivers, user cannot pass a size less than the Default BaseFS Size. For the `overlay2` storage driver, the size option is only available if the backing fs is `xfs` and mounted with the `pquota` mount option. Under these conditions, user can pass any size less than the backing fs size.

Mount tmpfs (--tmpfs)

```
$ docker run -d --tmpfs /run:rw,noexec,nosuid,size=65536k my_image
```

The `--tmpfs` flag mounts an empty tmpfs into the container with the `rw` , `noexec` , `nosuid` , `size=65536k` options.

Mount volume (-v, --read-only)

```
$ docker  run  -Â ‘pwd‘:‘pwd‘ -w ‘pwd‘ -i -t  ubuntu pwd
```

The `-Â` flag mounts the current working directory into the container. The `-w` lets the command being executed inside the current working directory, by changing into the directory to the value returned by `pwd` . So this combination executes the command using the container, but inside the current working directory.

```
$ docker run -Â /doesnt/exist:/foo -w /foo -i -t ubuntu bash
```

When the host directory of a bind-mounted volume doesn’t exist, Docker will automatically create this directory on the host for you. In the example above, Docker will create the `/doesnt/exist` folder before starting your container.

```
$ docker run --read-only -Â /icanwrite busybox touch /icanwrite/here
```

Volumes can be used in combination with `--read-only` to control where a container writes files. The `--read-only` flag mounts the container’s root filesystem as read only prohibiting writes to locations other than the specified volumes for the container.

```
$ docker run -t -i -Â /var/run/docker.sock:/var/run/docker.sock -Â /path/to/static-docker-binary
:/usr/bin/docker busybox sh
```

By bind-mounting the docker unix socket and statically linked docker binary (refer to get the linux binary (https://docs.docker.com/engine/installation/binaries/#/get-the-linux-binary)), you give the container the full access to create and manipulate the host’s Docker daemon.

On Windows, the paths must be specified using Windows-style semantics.

```
PS C:\> docker run -v c:\foo:c:\dest microsoft/nanoserver cmd /s /c type c:\dest\somefile.txt
Contents of file

PS C:\> docker run -v c:\foo:d: microsoft/nanoserver cmd /s /c type d:\somefile.txt
Contents of file
```

The following examples will fail when using Windows-based containers, as the destination of a volume or bind mount inside the container must be one of: a non-existing or empty directory; or a drive other than C:. Further, the source of a bind mount must be a local directory, not a file.

```
net use z: \\remotemachine\share
docker run -v z:\foo:c:\dest ...
docker run -v \\uncpath\to\directory:c:\dest ...
docker run -v c:\foo\somefile.txt:c:\dest ...
docker run -v c:\foo:c: ...
docker run -v c:\foo:c:\existing-directory-with-contents ...
```

For in-depth information about volumes, refer to manage data in containers (https://docs.docker.com/engine/tutorials/dockervolumes/)

Add bind mounts or volumes using the --mount flag

The `--mount` flag allows you to mount volumes, host-directories and `tmpfs` mounts in a container.

The `--mount` flag supports most options that are supported by the `-Â` or the `--volume` flag, but uses a different syntax. For in-depth information on the `--mount` flag, and a comparison between `--volume` and `--mount`, refer to the service create command reference (https://docs.docker.com/engine/reference/commandline/service_create/#add-bind-mounts-or-volumes).

Even though there is no plan to deprecate `--volume`, usage of `--mount` is recommended.

Examples:

```
$ docker run --read-only --mount type=volume,target=/icanwrite busybox touch /icanwrite/here
```

```
$ docker run -t -i --mount type=bind,src=/data,dst=/data busybox sh
```

Publish or expose port (-p, --expose)

```
$ docker run -p 127.0.0.1:80:8080/tcp ubuntu bash
```

This binds port `8080` of the container to TCP port `80` on `127.0.0.1` of the host machine. You can also specify `udp` and `sctp` ports. The Docker User Guide (https://docs.docker.com/engine/userguide/networking/default_network/dockerlinks/) explains in detail how to manipulate ports in Docker.

```
$ docker run --expose 80 ubuntu bash
```

This exposes port `80` of the container without publishing the port to the host system's interfaces.

Set environment variables (-e, --env, --env-file)

```
$ docker run -e MYVAR1 --env MYVAR2=foo --env-file ./env.list ubuntu bash
```

Use the `-e`, `--env`, and `--env-file` flags to set simple (non-array) environment variables in the container you're running, or overwrite variables that are defined in the Dockerfile of the image you're running.

You can define the variable and its value when running the container:

```
$ docker run --env VAR1=value1 --env VAR2=value2 ubuntu env | grep VAR
VAR1=value1
VAR2=value2
```

You can also use variables that you've exported to your local environment:

```
export VAR1=value1
export VAR2=value2

$ docker run --env VAR1 --env VAR2 ubuntu env | grep VAR
VAR1=value1
VAR2=value2
```

When running the command, the Docker CLI client checks the value the variable has in your local environment and passes it to the container. If no `=` is provided and that variable is not exported in your local environment, the variable won't be set in the container.

You can also load the environment variables from a file. This file should use the syntax `<variable>=value` (which sets the variable to the given value) or `<variable>` (which takes the value from the local environment), and `#` for comments.

```
$ cat env.list
# This is a comment
VAR1=value1
VAR2=value2
USER

$ docker run --env-file env.list ubuntu env | grep VAR
VAR1=value1
VAR2=value2
USER=denis
```

Set metadata on container (-l, --label, --label-file)

A label is a `key=value` pair that applies metadata to a container. To label a container with two labels:

```
$ docker run -· my-label --label com.example.foo=bar ubuntu bash
```

The `my-label` key doesn't specify a value so the label defaults to an empty string (`""`). To add multiple labels, repeat the label flag (`-·` or `--label`).

The `key=value` must be unique to avoid overwriting the label value. If you specify labels with identical keys but different values, each subsequent value overwrites the previous. Docker uses the last `key=value` you supply.

Use the `--label-file` flag to load multiple labels from a file. Delimit each label in the file with an EOL mark. The example below loads labels from a labels file in the current directory:

```
$ docker run --label-file ./labels ubuntu bash
```

The label-file format is similar to the format for loading environment variables. (Unlike environment variables, labels are not visible to processes running inside a container.) The following example illustrates a label-file format:

```
com.example.label1="a label"

# this is a comment
com.example.label2=another\ label
com.example.label3
```

You can load multiple label-files by supplying multiple `--label-file` flags.

For additional information on working with labels, see *Labels - custom metadata in Docker* (https://docs.docker.com/engine/userguide/labels-custom-metadata/) in the Docker User Guide.

Connect a container to a network (--network)

When you start a container use the `--network` flag to connect it to a network. This adds the `busybox` container to the `my-net` network.

```
$ docker run -itd --network=my-net busybox
```

You can also choose the IP addresses for the container with `--ip` and `--ip6` flags when you start the container on a user-defined network.

```
$ docker run -itd --network=my-net --ip=10.10.9.75 busybox
```

If you want to add a running container to a network use the `docker network connect` subcommand.

You can connect multiple containers to the same network. Once connected, the containers can communicate easily need only another container's IP address or name. For `overlay` networks or custom plugins that support multi-host connectivity, containers connected to the same multi-host network but launched from different Engines can also communicate in this way.

> Note: Service discovery is unavailable on the default bridge network. Containers can communicate via their IP addresses by default. To communicate by name, they must be linked.

You can disconnect a container from a network using the `docker network disconnect` command.

Mount volumes from container (--volumes-from)

```
$ docker run --volumes-from 777f7dc92da7 --volumes-from ba8c0c54f0f2:ro -i -t ubuntu pwd
```

The `--volumes-from` flag mounts all the defined volumes from the referenced containers. Containers can be specified by repetitions of the `--volumes-from` argument. The container ID may be optionally suffixed with `:ro` or `:rw` to mount the volumes in read-only or read-write mode, respectively. By default, the volumes are mounted in the same mode (read write or read only) as the reference container.

Labeling systems like SELinux require that proper labels are placed on volume content mounted into a container. Without a label, the security system might prevent the processes running inside the container from using the content. By default, Docker does not change the labels set by the OS.

To change the label in the container context, you can add either of two suffixes `:z` or `:Z` to the volume mount. These suffixes tell Docker to relabel file objects on the shared volumes. The `z` option tells Docker that two containers share the volume content. As a result, Docker labels the content with a shared content label. Shared volume labels allow all containers to read/write content. The `Z` option tells Docker to label the content with a private unshared label. Only the current container can use a private volume.

Attach to STDIN/STDOUT/STDERR (-a)

The `-Γ` flag tells `docker run` to bind to the container's `STDIN` , `STDOUT` or `STDERR` . This makes it possible to manipulate the output and input as needed.

```
$ echo "test" | docker run -i -Γ stdin ubuntu cat -
```

This pipes data into a container and prints the container's ID by attaching only to the container's `STDIN` .

```
$ docker run -Γ stderr ubuntu echo test
```

This isn't going to print anything unless there's an error because we've only attached to the `STDERR` of the container. The container's logs still store what's been written to `STDERR` and `STDOUT` .

```
$ cat somefile | docker run -i -Γ stdin mybuilder dobuild
```

This is how piping a file into a container could be done for a build. The container's ID will be printed after the build is done and the build logs could be retrieved using `docker logs` . This is useful if you need to pipe a file or something else into a container and retrieve the container's ID once the container has finished running.

Add host device to container (--device)

```
$ docker run --device=/dev/sdc:/dev/xvdc \
             --device=/dev/sdd --device=/dev/zero:/dev/nulo \
             -i -t \
             ubuntu ls -· /dev/{xvdc,sdd,nulo}

brw-rw---- 1 root disk 8, 2 Feb  9 16:05 /dev/xvdc
brw-rw---- 1 root disk 8, 3 Feb  9 16:05 /dev/sdd
crw-rw-rw- 1 root root 1, 5 Feb  9 16:05 /dev/nulo
```

It is often necessary to directly expose devices to a container. The `--device` option enables that. For example, a specific block storage device or loop device or audio device can be added to an otherwise unprivileged container (without the `--privileged` flag) and have the application directly access it.

By default, the container will be able to `read` , `write` and `mknod` these devices. This can be overridden using a third `:rwm` set of options to each `--device` flag:

```
$ docker run --device=/dev/sda:/dev/xvdc --rm -it ubuntu fdisk  /dev/xvdc

Command (m for help): q
$ docker run --device=/dev/sda:/dev/xvdc:r --rm -it ubuntu fdisk  /dev/xvdc
You will not be able to write the partition table.

Command (m for help): q

$ docker run --device=/dev/sda:/dev/xvdc:rw --rm -it ubuntu fdisk  /dev/xvdc

Command (m for help): q

$ docker run --device=/dev/sda:/dev/xvdc:m --rm -it ubuntu fdisk  /dev/xvdc
fdisk: unable to open /dev/xvdc: Operation not permitted
```

Note: `--device` cannot be safely used with ephemeral devices. Block devices that may be removed should not be added to untrusted containers with `--device` .

Restart policies (--restart)

Use Docker's `--restart` to specify a container's *restart policy*. A restart policy controls whether the Docker daemon restarts a container after exit. Docker supports the following restart policies:

Policy	Result
`no`	Do not automatically restart the container when it exits. This is the default.
`on-failure[:max-retries]`	Restart only if the container exits with a non-zero exit status. Optionally, limit the number of restart retries the Docker daemon attempts.
`unless-stopped`	Restart the container unless it is explicitly stopped or Docker itself is stopped or restarted.
`always`	Always restart the container regardless of the exit status. When you specify always, the Docker daemon will try to restart the container indefinitely. The container will also always start on daemon startup, regardless of the current state of the container.

```
$ docker run --restart=always redis
```

This will run the `redis` container with a restart policy of always so that if the container exits, Docker will restart it.

More detailed information on restart policies can be found in the Restart Policies (--restart) (https://docs.docker.com/engine/reference/run/#restart-policies---restart) section of the Docker run reference page.

Add entries to container hosts file (--add-host)

You can add other hosts into a container's `/etc/hosts` file by using one or more `--add-host` flags. This example adds a static address for a host named `docker` :

```
$ docker run --add-host=docker:10.180.0.1 --rm -it debian

root@f38c87f2a42d:/# ping docker
PING docker (10.180.0.1): 48 data bytes
56 bytes from 10.180.0.1: icmp_seq=0 ttl=254 time=7.600 ms
56 bytes from 10.180.0.1: icmp_seq=1 ttl=254 time=30.705 ms
^C--- docker ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 7.600/19.152/30.705/11.553 ms
```

Sometimes you need to connect to the Docker host from within your container. To enable this, pass the Docker host's IP address to the container using the `--add-host` flag. To find the host's address, use the `ip addr show` command.

The flags you pass to `ip addr show` depend on whether you are using IPv4 or IPv6 networking in your containers. Use the following flags for IPv4 address retrieval for a network device named `eth0` :

```
$ HOSTIP='ip -4 addr show scope global dev eth0 | grep inet | awk '{print \$2}' | cut -d / -f 1'
$ docker run  --add-host=docker:${HOSTIP} --rm -it debian
```

For IPv6 use the `-6` flag instead of the `-4` flag. For other network devices, replace `eth0` with the correct device name (for example `docker0` for the bridge device).

Set ulimits in container (--ulimit)

Since setting `ulimit` settings in a container requires extra privileges not available in the default container, you can set these using the `--ulimit` flag. `--ulimit` is specified with a soft and hard limit as such: `<type>=<soft limit>[:<hard limit>]` , for example:

```
$ docker run --ulimit nofile=1024:1024 --rm debian sh -c "ulimit -n"
1024
```

Note: If you do not provide a **`hard limit`** , the **`soft limit`** will be used for both values. If no **`ulimits`** are set, they will be inherited from the default **`ulimits`** set on the daemon. **`as`** option is disabled now. In other words, the following script is not supported:

```
$ docker run -it --ulimit as=1024 fedora /bin/bash'
```

The values are sent to the appropriate `syscall` as they are set. Docker doesn't perform any byte conversion. Take this into account when setting the values.

FOR `NPROC` USAGE

Be careful setting `nproc` with the `ulimit` flag as `nproc` is designed by Linux to set the maximum number of processes available to a user, not to a container. For example, start four containers with `daemon` user:

```
$ docker run -d -u daemon --ulimit nproc=3 busybox top

$ docker run -d -u daemon --ulimit nproc=3 busybox top

$ docker run -d -u daemon --ulimit nproc=3 busybox top

$ docker run -d -u daemon --ulimit nproc=3 busybox top
```

The 4th container fails and reports “[8] System error: resource temporarily unavailable” error. This fails because the caller set `nproc=3` resulting in the first three containers using up the three processes quota set for the `daemon` user.

Stop container with signal (--stop-signal)

The `--stop-signal` flag sets the system call signal that will be sent to the container to exit. This signal can be a valid unsigned number that matches a position in the kernel’s syscall table, for instance 9, or a signal name in the format SIGNAME, for instance SIGKILL.

Optional security options (--security-opt)

On Windows, this flag can be used to specify the `credentialspec` option. The `credentialspec` must be in the format `file://spec.txt` or `registry://keyname` .

Stop container with timeout (--stop-timeout)

The `--stop-timeout` flag sets the timeout (in seconds) that a pre-defined (see `--stop-signal`) system call signal that will be sent to the container to exit. After timeout elapses the container will be killed with SIGKILL.

Specify isolation technology for container (--isolation)

This option is useful in situations where you are running Docker containers on Windows. The `--isolation <value>` option sets a container’s isolation technology. On Linux, the only supported is the `default` option which uses Linux namespaces. These two commands are equivalent on Linux:

```
$ docker run -d busybox top
$ docker run -d --isolation default busybox top
```

On Windows, `--isolation` can take one of these values:

Value	Description
`default`	Use the value specified by the Docker daemon’s `--exec-opt` or system default (see below).
`process`	Shared-kernel namespace isolation (not supported on Windows client operating systems).
`hyperv`	Hyper-V hypervisor partition-based isolation.

The default isolation on Windows server operating systems is `process` . The default (and only supported) isolation on Windows client operating systems is `hyperv` . An attempt to start a container on a client operating system with `--isolation process` will fail.

On Windows server, assuming the default configuration, these commands are equivalent and result in `process` isolation:

```
PS C:\> docker run -d microsoft/nanoserver powershell echo process
PS C:\> docker run -d --isolation default microsoft/nanoserver powershell echo process
PS C:\> docker run -d --isolation process microsoft/nanoserver powershell echo process
```

If you have set the `--exec-opt isolation=hyperv` option on the Docker `daemon`, or are running against a Windows client-based daemon, these commands are equivalent and result in `hyperv` isolation:

```
PS C:\> docker run -d microsoft/nanoserver powershell echo hyperv
PS C:\> docker run -d --isolation default microsoft/nanoserver powershell echo hyperv
PS C:\> docker run -d --isolation hyperv microsoft/nanoserver powershell echo hyperv
```

Specify hard limits on memory available to containers (-m, --memory)

These parameters always set an upper limit on the memory available to the container. On Linux, this is set on the cgroup and applications in a container can query it at `/sys/fs/cgroup/memory/memory.limit_in_bytes`.

On Windows, this will affect containers differently depending on what type of isolation is used.

- With `process` isolation, Windows will report the full memory of the host system, not the limit to applications running inside the container

```
  PS C:\> docker run -it -m 2GB --isolation=process microsoft/nanoserver powershell Get-Com
puterInfo *memory*

  CsTotalPhysicalMemory      : 17064509440
  CsPhyicallyInstalledMemory : 16777216
  OsTotalVisibleMemorySize   : 16664560
  OsFreePhysicalMemory       : 14646720
  OsTotalVirtualMemorySize   : 19154928
  OsFreeVirtualMemory        : 17197440
  OsInUseVirtualMemory       : 1957488
  OsMaxProcessMemorySize     : 137438953344
```

- With `hyperv` isolation, Windows will create a utility VM that is big enough to hold the memory limit, plus the minimal OS needed to host the container. That size is reported as “Total Physical Memory.”

```
  PS C:\> docker run -it -m 2GB --isolation=hyperv microsoft/nanoserver powershell Get-Comp
uterInfo *memory*

  CsTotalPhysicalMemory      : 2683355136
  CsPhyicallyInstalledMemory :
  OsTotalVisibleMemorySize   : 2620464
  OsFreePhysicalMemory       : 2306552
  OsTotalVirtualMemorySize   : 2620464
  OsFreeVirtualMemory        : 2356692
  OsInUseVirtualMemory       : 263772
  OsMaxProcessMemorySize     : 137438953344
```

Configure namespaced kernel parameters (sysctls) at runtime

The `--sysctl` sets namespaced kernel parameters (sysctls) in the container. For example, to turn on IP forwarding in the containers network namespace, run this command:

```
$ docker run --sysctl net.ipv4.ip_forward=1 someimage
```

> Note: Not all sysctls are namespaced. Docker does not support changing sysctls inside of a container that also modify the host system. As the kernel evolves we expect to see more sysctls become namespaced.

CURRENTLY SUPPORTED SYSCTLS

- `IPC Namespace` :

  ```
  kernel.msgmax, kernel.msgmnb, kernel.msgmni, kernel.sem, kernel.shmall, kernel.shmmax, kern
  el.shmmni, kernel.shm_rmid_forced
  Sysctls beginning with fs.mqueue.*
  ```

  If you use the `--ipc=host` option these sysctls will not be allowed.

- `Network Namespace` :

  Sysctls beginning with net.*

  If you use the `--network=host` option using these sysctls will not be allowed.

How services work

Estimated reading time: 5 minutes

To deploy an application image when Docker Engine is in swarm mode, you create a service. Frequently a service is the image for a microservice within the context of some larger application. Examples of services might include an HTTP server, a database, or any other type of executable program that you wish to run in a distributed environment.

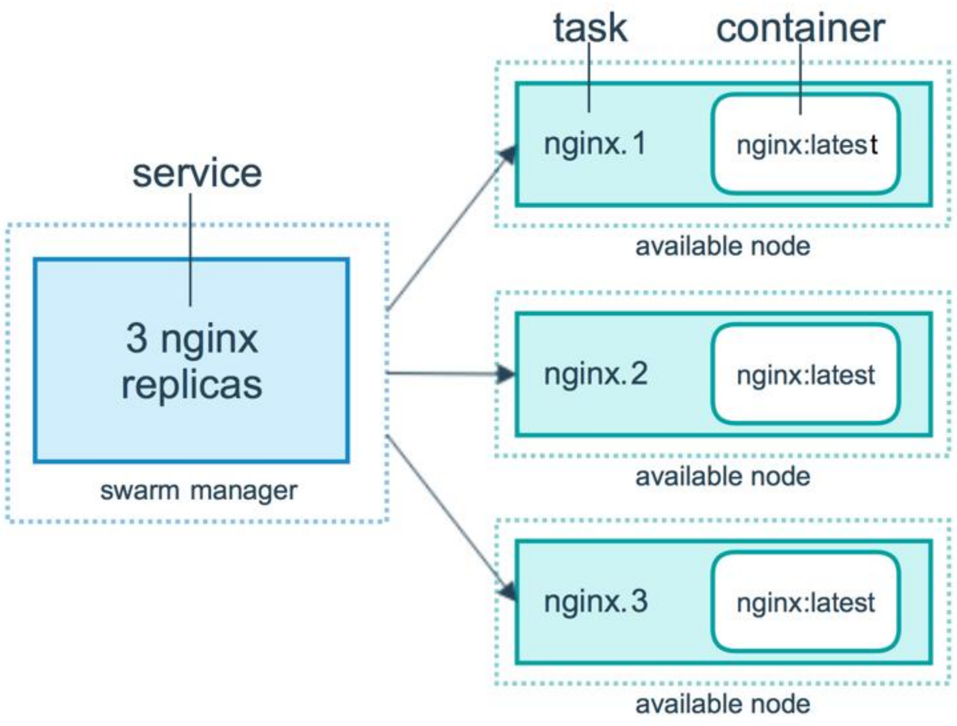
When you create a service, you specify which container image to use and which commands to execute inside running containers. You also define options for the service including:

- the port where the swarm makes the service available outside the swarm
- an overlay network for the service to connect to other services in the swarm
- CPU and memory limits and reservations
- a rolling update policy
- the number of replicas of the image to run in the swarm

Services, tasks, and containers

When you deploy the service to the swarm, the swarm manager accepts your service definition as the desired state for the service. Then it schedules the service on nodes in the swarm as one or more replica tasks. The tasks run independently of each other on nodes in the swarm.

For example, imagine you want to load balance between three instances of an HTTP listener. The diagram below shows an HTTP listener service with three replicas. Each of the three instances of the listener is a task in the swarm.



A container is an isolated process. In the swarm mode model, each task invokes exactly one container. A task is analogous to a “slot” where the scheduler places a container. Once the container is live, the scheduler recognizes that the task is in a running state. If the container fails health checks or terminates, the task terminates.

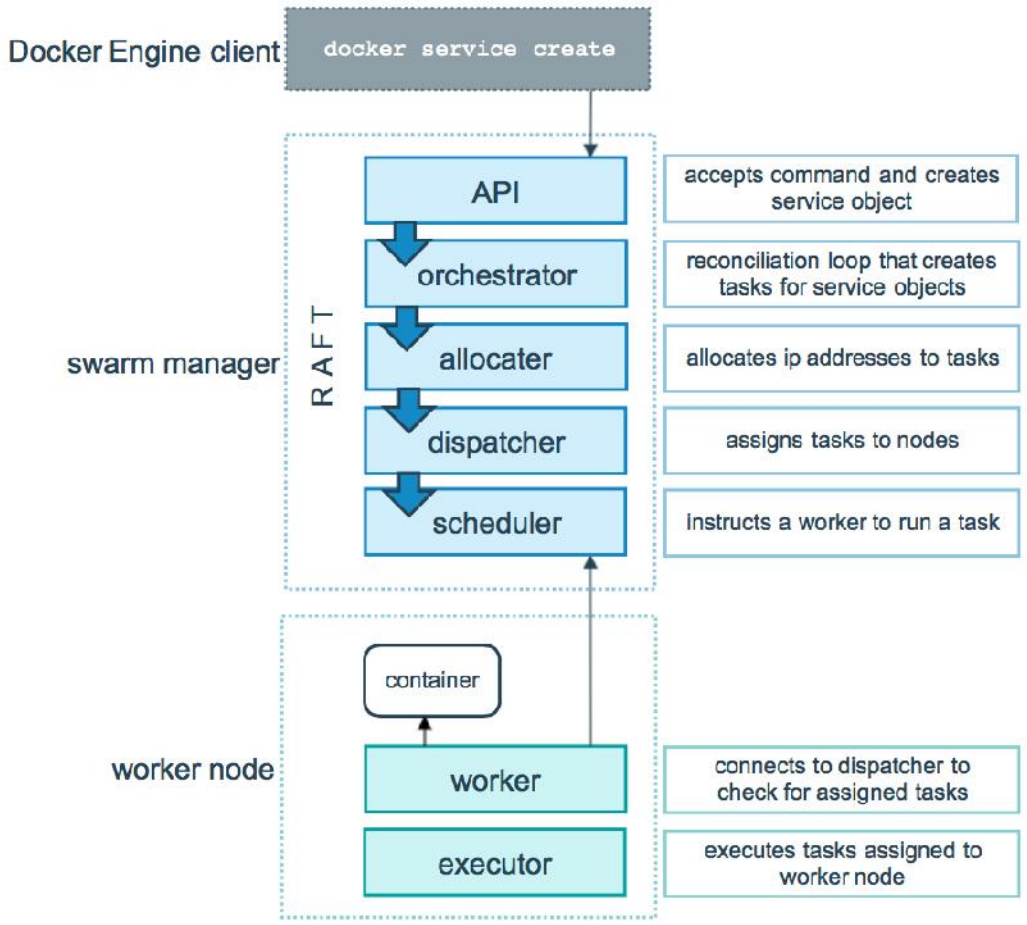
Tasks and scheduling

A task is the atomic unit of scheduling within a swarm. When you declare a desired service state by creating or updating a service, the orchestrator realizes the desired state by scheduling tasks. For instance, you define a service that instructs the orchestrator to keep three instances of an HTTP listener running at all times. The orchestrator responds by creating three tasks. Each task is a slot that the scheduler fills by spawning a container. The container is the instantiation of the task. If an HTTP listener task subsequently fails its health check or crashes, the orchestrator creates a new replica task that spawns a new container.

A task is a one-directional mechanism. It progresses monotonically through a series of states: assigned, prepared, running, etc. If the task fails the orchestrator removes the task and its container and then creates a new task to replace it according to the desired state specified by the service.

The underlying logic of Docker swarm mode is a general purpose scheduler and orchestrator. The service and task abstractions themselves are unaware of the containers they implement. Hypothetically, you could implement other types of tasks such as virtual machine tasks or non-containerized process tasks. The scheduler and orchestrator are agnostic about the type of task. However, the current version of Docker only supports container tasks.

The diagram below shows how swarm mode accepts service create requests and schedules tasks to worker nodes.


Docker Engine client
docker service create
swarm manager
RAFT
API
accepts command and creates service object
orchestrator
reconciliation loop that creates tasks for service objects
allocater
allocates ip addresses to tasks
dispatcher
assigns tasks to nodes
scheduler
instructs a worker to run a task
worker node
container
worker
connects to dispatcher to check for assigned tasks
executor
executes tasks assigned to worker node

Pending services

A service may be configured in such a way that no node currently in the swarm can run its tasks. In this case, the service remains in state `pending` . Here are a few examples of when a service might remain in state `pending` .

> Note: If your only intention is to prevent a service from being deployed, scale the service to 0 instead of trying to configure it in such a way that it remains in `pending` .

- If all nodes are paused or drained, and you create a service, it is pending until a node becomes available. In reality, the first node to become available gets all of the tasks, so this is not a good thing to do in a production environment.
- You can reserve a specific amount of memory for a service. If no node in the swarm has the required amount of memory, the service remains in a pending state until a node is available which can run its tasks. If you specify a very large value, such as 500 GB, the task stays pending forever, unless you really have a node which can satisfy it.
- You can impose placement constraints on the service, and the constraints may not be able to be honored at a given time.

This behavior illustrates that the requirements and configuration of your tasks are not tightly tied to the current state of the swarm. As the administrator of a swarm, you declare the desired state of your swarm, and the manager works with the nodes in the swarm to create that state. You do not need to micro-manage the tasks on the swarm.

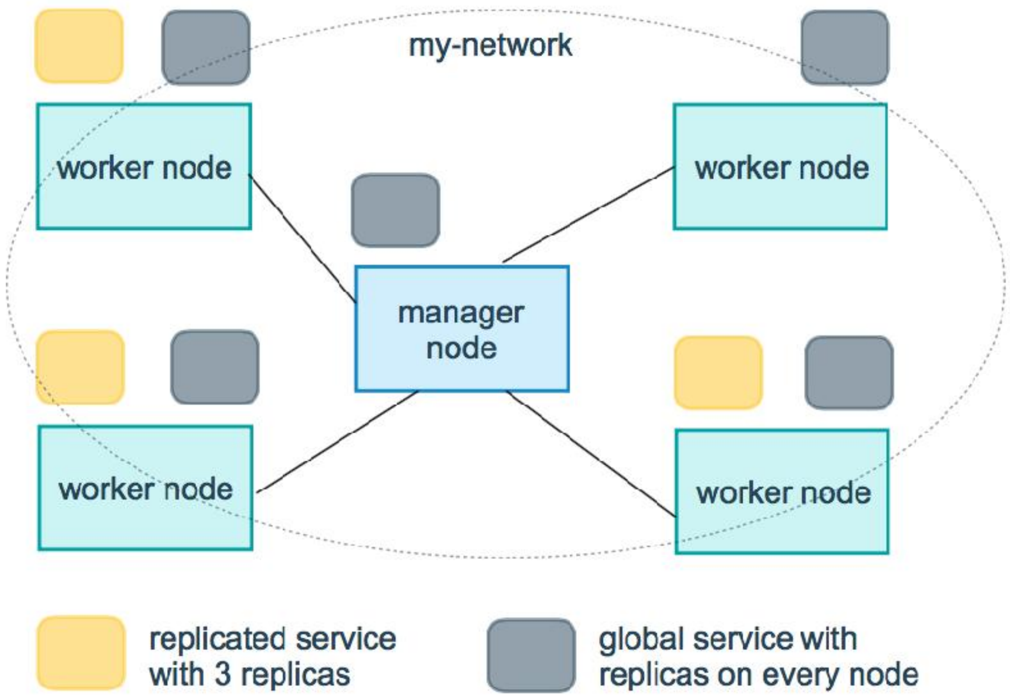
Replicated and global services

There are two types of service deployments, replicated and global.

For a replicated service, you specify the number of identical tasks you want to run. For example, you decide to deploy an HTTP service with three replicas, each serving the same content.

A global service is a service that runs one task on every node. There is no pre-specified number of tasks. Each time you add a node to the swarm, the orchestrator creates a task and the scheduler assigns the task to the new node. Good candidates for global services are monitoring agents, an anti-virus scanners or other types of containers that you want to run on every node in the swarm.

The diagram below shows a three-service replica in yellow and a global service in gray.


my-network
worker node
worker node
manager node
worker node
worker node
replicated service with 3 replicas
global service with replicas on every node

Learn more

- Read about how swarm mode nodes (https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/) work.
- Learn how PKI (https://docs.docker.com/engine/swarm/how-swarm-mode-works/pki/) works in swarm mode.

docker (https://docs.docker.com/glossary/?term=docker), container (https://docs.docker.com/glossary/?term=container), cluster (https://docs.docker.com/glossary/?term=cluster), swarm mode (https://docs.docker.com/glossary/?term=swarm mode), node (https://docs.docker.com/glossary/?term=node)

Lock your swarm to protect its encryption key

Estimated reading time: 5 minutes

In Docker 1.13 and higher, the Raft logs used by swarm managers are encrypted on disk by default. This at-rest encryption protects your service's configuration and data from attackers who gain access to the encrypted Raft logs. One of the reasons this feature was introduced was in support of the new Docker secrets (https://docs.docker.com/engine/swarm/secrets/) feature.

When Docker restarts, both the TLS key used to encrypt communication among swarm nodes, and the key used to encrypt and decrypt Raft logs on disk, are loaded into each manager node's memory. Docker 1.13 introduces the ability to protect the mutual TLS encryption key and the key used to encrypt and decrypt Raft logs at rest, by allowing you to take ownership of these keys and to require manual unlocking of your managers. This feature is called *autolock*.

When Docker restarts, you must unlock the swarm (https://docs.docker.com/engine/swarm/swarm_manager_locking/#unlock-a-swarm) first, using a *key encryption key* generated by Docker when the swarm was locked. You can rotate this key encryption key at any time.

> Note: You don't need to unlock the swarm when a new node joins the swarm, because the key is propagated to it over mutual TLS.

Initialize a swarm with autolocking enabled

When you initialize a new swarm, you can use the `--autolock` flag to enable autolocking of swarm manager nodes when Docker restarts.

```
$ docker swarm init --autolock

Swarm initialized: current node (k1q27tfyx9rncpixhk69sa61v) is now a
 manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-0j52ln6hxjpxk2wgk917abcnxywj3xed0y8vi1e5m9t3utt
rtu-7bnxvvlz2mrcpfonjuztmtts9 \
    172.31.46.109:2377

To add a manager to this swarm, run 'docker swarm join-token manager
' and follow the instructions.

To unlock a swarm manager after it restarts, run the `docker swarm u
nlock`
command and provide the following key:

    SWMKEY-1-WuYH/IX284+lRcXuoVf38viIDK3HJEKY13MIHX+tTt8
```

Store the key in a safe place, such as in a password manager.

When Docker restarts, you need to unlock the swarm (https://docs.docker.com/engine/swarm/swarm_manager_locking/#unlock-a-swarm). A locked swarm causes an error like the following when you try to start or restart a service:

```
$ sudo service docker restart

$ docker service ls

Error response from daemon: Swarm is encrypted and needs to be unloc
ked before it can be used. Use "docker swarm unlock" to unlock it.
```

Enable or disable autolock on an existing swarm

To enable autolock on an existing swarm, set the `autolock` flag to `true` .

```
$ docker swarm update --autolock=true

Swarm updated.
To unlock a swarm manager after it restarts, run the `docker swarm u
nlock`
command and provide the following key:

    SWMKEY-1-+MrE8NgAyKj5r3NcR4FiQMdgu+7W72urH0EZeSmP/0Y

Please remember to store this key in a password manager, since witho
ut it you
will not be able to restart the manager.
```

To disable autolock, set `--autolock` to `false`. The mutual TLS key and the encryption key used to read and write Raft logs are stored unencrypted on disk. There is a trade-off between the risk of storing the encryption key unencrypted at rest and the convenience of restarting a swarm without needing to unlock each manager.

```
$ docker swarm update --autolock=false
```

Keep the unlock key around for a short time after disabling autolocking, in case a manager goes down while it is still configured to lock using the old key.

Unlock a swarm

To unlock a locked swarm, use `docker swarm unlock`.

```
$ docker swarm unlock

Please enter unlock key:
```

Enter the encryption key that was generated and shown in the command output when you locked the swarm or rotated the key, and the swarm unlocks.

View the current unlock key for a running swarm

Consider a situation where your swarm is running as expected, then a manager node becomes unavailable. You troubleshoot the problem and bring the physical node back online, but you need to unlock the manager by providing the unlock key to read the encrypted credentials and Raft logs.

If the key has not been rotated since the node left the swarm, and you have a quorum of functional manager nodes in the swarm, you can view the current unlock key using `docker swarm unlock-key` without any arguments.

```
$ docker swarm unlock-key

To unlock a swarm manager after it restarts, run the `docker swarm u
nlock`
command and provide the following key:

    SWMKEY-1-8jDgbUNlJtUe5P/lcr9IXGVxqZpZUXPzd+qzcGp4ZYA

Please remember to store this key in a password manager, since witho
ut it you
will not be able to restart the manager.
```

If the key was rotated after the swarm node became unavailable and you do not have a record of the previous key, you may need to force the manager to leave the swarm and join it back to the swarm as a new manager.

Rotate the unlock key

You should rotate the locked swarm's unlock key on a regular schedule.

```
$ docker swarm unlock-key --rotate

Successfully rotated manager unlock key.

To unlock a swarm manager after it restarts, run the `docker swarm u
nlock`
command and provide the following key:

    SWMKEY-1-8jDgbUNlJtUe5P/lcr9IXGVxqZpZUXPzd+qzcGp4ZYA

Please remember to store this key in a password manager, since witho
ut it you
will not be able to restart the manager.
```

Warning: When you rotate the unlock key, keep a record of the old key around for a few minutes, so that if a manager goes down before it gets the new key, it may still be unlocked with the old one.

swarm (https://docs.docker.com/glossary/?term=swarm), manager (https://docs.docker.com/glossary/?term=manager), lock (https://docs.docker.com/glossary/?term=lock), unlock (https://docs.docker.com/glossary/?term=unlock), autolock (https://docs.docker.com/glossary/?term=autolock), encryption (https://docs.docker.com/glossary/?term=encryption)

Lock your swarm to protect its encryption key

Estimated reading time: 5 minutes

In Docker 1.13 and higher, the Raft logs used by swarm managers are encrypted on disk by default. This at-rest encryption protects your service's configuration and data from attackers who gain access to the encrypted Raft logs. One of the reasons this feature was introduced was in support of the new Docker secrets (https://docs.docker.com/engine/swarm/secrets/) feature.

When Docker restarts, both the TLS key used to encrypt communication among swarm nodes, and the key used to encrypt and decrypt Raft logs on disk, are loaded into each manager node's memory. Docker 1.13 introduces the ability to protect the mutual TLS encryption key and the key used to encrypt and decrypt Raft logs at rest, by allowing you to take ownership of these keys and to require manual unlocking of your managers. This feature is called *autolock*.

When Docker restarts, you must unlock the swarm (https://docs.docker.com/engine/swarm/swarm_manager_locking/#unlock-a-swarm) first, using a *key encryption key* generated by Docker when the swarm was locked. You can rotate this key encryption key at any time.

> Note: You don't need to unlock the swarm when a new node joins the swarm, because the key is propagated to it over mutual TLS.

Initialize a swarm with autolocking enabled

When you initialize a new swarm, you can use the `--autolock` flag to enable autolocking of swarm manager nodes when Docker restarts.

```
$ docker swarm init --autolock

Swarm initialized: current node (k1q27tfyx9rncpixhk69sa61v) is now a
 manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-0j52ln6hxjpxk2wgk917abcnxywj3xed0y8vi1e5m9t3utt
rtu-7bnxvvlz2mrcpfonjuztmtts9 \
    172.31.46.109:2377

To add a manager to this swarm, run 'docker swarm join-token manager
' and follow the instructions.

To unlock a swarm manager after it restarts, run the `docker swarm u
nlock`
command and provide the following key:

    SWMKEY-1-WuYH/IX284+lRcXuoVf38viIDK3HJEKY13MIHX+tTt8
```

Store the key in a safe place, such as in a password manager.

When Docker restarts, you need to unlock the swarm (https://docs.docker.com/engine/swarm/swarm_manager_locking/#unlock-a-swarm). A locked swarm causes an error like the following when you try to start or restart a service:

```
$ sudo service docker restart

$ docker service ls

Error response from daemon: Swarm is encrypted and needs to be unloc
ked before it can be used. Use "docker swarm unlock" to unlock it.
```

Enable or disable autolock on an existing swarm

To enable autolock on an existing swarm, set the `autolock` flag to `true` .

```
$ docker swarm update --autolock=true

Swarm updated.
To unlock a swarm manager after it restarts, run the `docker swarm u
nlock`
command and provide the following key:

    SWMKEY-1-+MrE8NgAyKj5r3NcR4FiQMdgu+7W72urH0EZeSmP/0Y

Please remember to store this key in a password manager, since witho
ut it you
will not be able to restart the manager.
```

To disable autolock, set `--autolock` to `false`. The mutual TLS key and the encryption key used to read and write Raft logs are stored unencrypted on disk. There is a trade-off between the risk of storing the encryption key unencrypted at rest and the convenience of restarting a swarm without needing to unlock each manager.

```
$ docker swarm update --autolock=false
```

Keep the unlock key around for a short time after disabling autolocking, in case a manager goes down while it is still configured to lock using the old key.

Unlock a swarm

To unlock a locked swarm, use `docker swarm unlock`.

```
$ docker swarm unlock

Please enter unlock key:
```

Enter the encryption key that was generated and shown in the command output when you locked the swarm or rotated the key, and the swarm unlocks.

View the current unlock key for a running swarm

Consider a situation where your swarm is running as expected, then a manager node becomes unavailable. You troubleshoot the problem and bring the physical node back online, but you need to unlock the manager by providing the unlock key to read the encrypted credentials and Raft logs.

If the key has not been rotated since the node left the swarm, and you have a quorum of functional manager nodes in the swarm, you can view the current unlock key using `docker swarm unlock-key` without any arguments.

```
$ docker swarm unlock-key

To unlock a swarm manager after it restarts, run the `docker swarm u
nlock`
command and provide the following key:

    SWMKEY-1-8jDgbUNlJtUe5P/lcr9IXGVxqZpZUXPzd+qzcGp4ZYA

Please remember to store this key in a password manager, since witho
ut it you
will not be able to restart the manager.
```

If the key was rotated after the swarm node became unavailable and you do not have a record of the previous key, you may need to force the manager to leave the swarm and join it back to the swarm as a new manager.

Rotate the unlock key

You should rotate the locked swarm's unlock key on a regular schedule.

```
$ docker swarm unlock-key --rotate

Successfully rotated manager unlock key.

To unlock a swarm manager after it restarts, run the `docker swarm u
nlock`
command and provide the following key:

    SWMKEY-1-8jDgbUNlJtUe5P/lcr9IXGVxqZpZUXPzd+qzcGp4ZYA

Please remember to store this key in a password manager, since witho
ut it you
will not be able to restart the manager.
```

Warning: When you rotate the unlock key, keep a record of the old key around for a few minutes, so that if a manager goes down before it gets the new key, it may still be unlocked with the old one.

swarm (https://docs.docker.com/glossary/?term=swarm), manager (https://docs.docker.com/glossary/?term=manager), lock (https://docs.docker.com/glossary/?term=lock), unlock (https://docs.docker.com/glossary/?term=unlock), autolock (https://docs.docker.com/glossary/?term=autolock), encryption (https://docs.docker.com/glossary/?term=encryption)

Deploy a service to the swarm

Estimated reading time: 1 minute

After you create a swarm (https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/), you can deploy a service to the swarm. For this tutorial, you also added worker nodes (https://docs.docker.com/engine/swarm/swarm-tutorial/add-nodes/), but that is not a requirement to deploy a service.

1. Open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named `manager1` .
2. Run the following command:

```
$ docker service create --replicas 1 --name helloworld alpine p
ing docker.com

9uk4639qpg7npwf3fn2aasksr
```

   - The `docker service create` command creates the service.
   - The `--name` flag names the service `helloworld` .
   - The `--replicas` flag specifies the desired state of 1 running instance.
   - The arguments `alpine ping docker.com` define the service as an Alpine Linux container that executes the command `ping docker.com` .

3. Run `docker service ls` to see the list of running services:

```
$ docker service ls

ID            NAME        SCALE  IMAGE   COMMAND
9uk4639qpg7n  helloworld  1/1    alpine  ping docker.com
```

What’s next?

Now you've deployed a service to the swarm, you're ready to inspect the service (https://docs.docker.com/engine/swarm/swarm-tutorial/inspect-service/).

tutorial (https://docs.docker.com/glossary/?term=tutorial), cluster management (https://docs.docker.com/glossary/?term=cluster management), swarm mode (https://docs.docker.com/glossary/?term=swarm mode)

Inspect a service on the swarm

Estimated reading time: 2 minutes

When you have deployed a service (https://docs.docker.com/engine/swarm/swarm-tutorial/deploy-service/) to your swarm, you can use the Docker CLI to see details about the service running in the swarm.

1. If you haven't already, open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named `manager1` .

2. Run `docker service inspect --pretty <SERVICE-ID>` to display the details about a service in an easily readable format.

   To see the details on the `helloworld` service:

   ```
   [manager1]$ docker service inspect --pretty helloworld

   ID:             9uk4639qpg7npwf3fn2aasksr
   Name:           helloworld
   Service Mode:   REPLICATED
    Replicas:              1
   Placement:
   UpdateConfig:
    Parallelism:   1
   ContainerSpec:
    Image:         alpine
    Args:  ping docker.com
   Resources:
   Endpoint Mode:  vip
   ```

   > Tip: To return the service details in json format, run the same command without the `--pretty` flag.

```
[manager1]$ docker service inspect helloworld
[
{
    "ID": "9uk4639qpg7npwf3fn2aasksr",
    "Version": {
        "Index": 418
    },
    "CreatedAt": "2016-06-16T21:57:11.622222327Z",
    "UpdatedAt": "2016-06-16T21:57:11.622222327Z",
    "Spec": {
        "Name": "helloworld",
        "TaskTemplate": {
            "ContainerSpec": {
                "Image": "alpine",
                "Args": [
                    "ping",
                    "docker.com"
                ]
            },
            "Resources": {
                "Limits": {},
                "Reservations": {}
            },
            "RestartPolicy": {
                "Condition": "any",
                "MaxAttempts": 0
            },
            "Placement": {}
        },
        "Mode": {
            "Replicated": {
                "Replicas": 1
            }
        },
        "UpdateConfig": {
            "Parallelism": 1
        },
        "EndpointSpec": {
            "Mode": "vip"
        }
    },
    "Endpoint": {
        "Spec": {}
    }
}
]
```

3. Run `docker service ps <SERVICE-ID>` to see which nodes are running the service:

   ```
   [manager1]$ docker service ps helloworld

   NAME                                    IMAGE   NODE     DESIRE
   D STATE  LAST STATE
   helloworld.1.8p1vev3fq5zm0mi8g0as41w35  alpine  worker2  Runnin
   g         Running 3 minutes
   ```

   In this case, the one instance of the `helloworld` service is running on the `worker2` node. You may see the service running on your manager node. By default, manager nodes in a swarm can execute tasks just like worker nodes.

   Swarm also shows you the `DESIRED STATE` and `LAST STATE` of the service task so you can see if tasks are running according to the service definition.

4. Run `docker ps` on the node where the task is running to see details about the container for the task.

   > Tip: If `helloworld` is running on a node other than your manager node, you must ssh to that node.

   ```
   [worker2]$docker ps

   CONTAINER ID        IMAGE               COMMAND             CRE
   ATED             STATUS              PORTS               NAMES
   e609dde94e47        alpine:latest       "ping docker.com"   3 m
   inutes ago       Up 3 minutes                            hellow
   orld.1.8p1vev3fq5zm0mi8g0as41w35
   ```

What’s next?

Next, you can change the scale (https://docs.docker.com/engine/swarm/swarm-tutorial/scale-service/) for the service running in the swarm.

tutorial (https://docs.docker.com/glossary/?term=tutorial), cluster management (https://docs.docker.com/glossary/?term=cluster management), swarm mode (https://docs.docker.com/glossary/?term=swarm mode)

docker stack deploy

Estimated reading time: 4 minutes

Description

Deploy a new stack or update an existing stack

API 1.25+ (https://docs.docker.com/engine/api/v1.25/) The client and daemon API must both be at least 1.25 (https://docs.docker.com/engine/api/v1.25/) to use this command. Use the `docker version` command on the client to check your client and daemon API versions.

Usage

```
docker stack deploy [OPTIONS] STACK
```

Options

Name, shorthand	Default	Description
`--bundle-file`		experimental (daemon) (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file) Swarm Path to a Distributed Application Bundle file
`--compose-file , -c`		API 1.25+ (https://docs.docker.com/engine/api/v1.25/) Path to a Compose file, or "-" to read from stdin
`--namespace`		Kubernetes Kubernetes namespace to use
`--prune`		API 1.27+ (https://docs.docker.com/engine/api/v1.27/) Swarm Prune services that are no longer referenced
`--resolve-image`	`always`	API 1.30+ (https://docs.docker.com/engine/api/v1.30/) Swarm Query the registry to resolve image digest and supported platforms ("always"|"changed"|"never")
`--with-registry-auth`		Swarm Send registry authentication details to Swarm agents
`--kubeconfig`		Kubernetes Kubernetes config file
`--orchestrator`		Orchestrator to use (swarm|kubernetes|all)

Parent command

Command	Description
docker stack (https://docs.docker.com/engine/reference/commandline/stack)	Manage Docker stacks

Related commands

Command	Description
docker stack deploy (https://docs.docker.com/engine/reference/commandline/stack_deploy/)	Deploy a new stack or update an existing stack
docker stack ls (https://docs.docker.com/engine/reference/commandline/stack_ls/)	List stacks
docker stack ps (https://docs.docker.com/engine/reference/commandline/stack_ps/)	List the tasks in the stack
docker stack rm (https://docs.docker.com/engine/reference/commandline/stack_rm/)	Remove one or more stacks
docker stack services (https://docs.docker.com/engine/reference/commandline/stack_services/)	List the services in the stack

Extended description

Create and update a stack from a `compose` or a `dab` file on the swarm. This command has to be run targeting a manager node.

Examples

Compose file

The `deploy` command supports compose file version `3.0` and above.

```
$ docker stack deploy --compose-file docker-compose.yml vossibility

Ignoring unsupported options: links

Creating network vossibility_vossibility
Creating network vossibility_default
Creating service vossibility_nsqd
Creating service vossibility_logstash
Creating service vossibility_elasticsearch
Creating service vossibility_kibana
Creating service vossibility_ghollector
Creating service vossibility_lookupd
```

The Compose file can also be provided as standard input with `--compose-file -` :

```
$ cat docker-compose.yml | docker stack deploy --compose-file - vossibility

Ignoring unsupported options: links

Creating network vossibility_vossibility
Creating network vossibility_default
Creating service vossibility_nsqd
Creating service vossibility_logstash
Creating service vossibility_elasticsearch
Creating service vossibility_kibana
Creating service vossibility_ghollector
Creating service vossibility_lookupd
```

If your configuration is split between multiple Compose files, e.g. a base configuration and environment-specific overrides, you can provide multiple `--compose-file` flags.

```
$ docker stack deploy --compose-file docker-compose.yml -c docker-compose.prod.yml vossibilit
y

Ignoring unsupported options: links

Creating network vossibility_vossibility
Creating network vossibility_default
Creating service vossibility_nsqd
Creating service vossibility_logstash
Creating service vossibility_elasticsearch
Creating service vossibility_kibana
Creating service vossibility_ghollector
Creating service vossibility_lookupd
```

You can verify that the services were correctly created:

```
$ docker service ls

ID            NAME                               MODE        REPLICAS  IMAGE
29bv0vnlm903  vossibility_lookupd                replicated  1/1       nsqio/nsq@sha256:eeba0
5599f31eba418e96e71e0984c3dc96963ceb66924dd37a47bf7ce18a662
4awt47624qwh  vossibility_nsqd                   replicated  1/1       nsqio/nsq@sha256:eeba0
5599f31eba418e96e71e0984c3dc96963ceb66924dd37a47bf7ce18a662
4tjx9biia6fs  vossibility_elasticsearch          replicated  1/1       elasticsearch@sha256:1
2ac7c6af55d001f71800b83ba91a04f716e58d82e748fa6e5a7359eed2301aa
7563uuzr9eys  vossibility_kibana                 replicated  1/1       kibana@sha256:6995a2d2
5709a62694a937b8a529ff36da92ebee74bafd7bf00e6caf6db2eb03
9gc5m4met4he  vossibility_logstash               replicated  1/1       logstash@sha256:2dc8bd
dd1bb4a5a34e8ebaf73749f6413c101b2edef6617f2f7713926d2141fe
axqh55ipl40h  vossibility_vossibility-collector  replicated  1/1       icecrime/vossibility-c
ollector@sha256:f03f2977203ba6253988c18d04061c5ec7aab46bca9dfd89a9a1fa4500989fba
```

DAB file

```
$ docker stack deploy --bundle-file vossibility-stack.dab vossibility

Loading bundle from vossibility-stack.dab
Creating service vossibility_elasticsearch
Creating service vossibility_kibana
Creating service vossibility_logstash
Creating service vossibility_lookupd
Creating service vossibility_nsqd
Creating service vossibility_vossibility-collector
```

You can verify that the services were correctly created:

```
$ docker service ls

ID            NAME                               MODE        REPLICAS  IMAGE
29bv0vnlm903  vossibility_lookupd                replicated  1/1       nsqio/nsq@sha256:eeba0
5599f31eba418e96e71e0984c3dc96963ceb66924dd37a47bf7ce18a662
4awt47624qwh  vossibility_nsqd                   replicated  1/1       nsqio/nsq@sha256:eeba0
5599f31eba418e96e71e0984c3dc96963ceb66924dd37a47bf7ce18a662
4tjx9biia6fs  vossibility_elasticsearch          replicated  1/1       elasticsearch@sha256:1
2ac7c6af55d001f71800b83ba91a04f716e58d82e748fa6e5a7359eed2301aa
7563uuzr9eys  vossibility_kibana                 replicated  1/1       kibana@sha256:6995a2d2
5709a62694a937b8a529ff36da92ebee74bafd7bf00e6caf6db2eb03
9gc5m4met4he  vossibility_logstash               replicated  1/1       logstash@sha256:2dc8bd
dd1bb4a5a34e8ebaf73749f6413c101b2edef6617f2f7713926d2141fe
axqh55ipl40h  vossibility_vossibility-collector  replicated  1/1       icecrime/vossibility-c
ollector@sha256:f03f2977203ba6253988c18d04061c5ec7aab46bca9dfd89a9a1fa4500989fba
```

Compose file version 3 reference

Estimated reading time: 62 minutes

Reference and guidelines

These topics describe version 3 of the Compose file format. This is the newest version.

Compose and Docker compatibility matrix

There are several versions of the Compose file format – 1, 2, 2.x, and 3.x. The table below is a quick look. For full details on what each version includes and how to upgrade, see About versions and upgrading (https://docs.docker.com/compose/compose-file/compose-versioning/).

This table shows which Compose file versions support specific Docker releases.

Compose file format	Docker Engine release
3.7	18.06.0+
3.6	18.02.0+
3.5	17.12.0+
3.4	17.09.0+
3.3	17.06.0+
3.2	17.04.0+
3.1	1.13.1+
3.0	1.13.0+
2.4	17.12.0+

Compose file format	Docker Engine release
2.3	17.06.0+
2.2	1.13.0+
2.1	1.12.0+
2.0	1.10.0+
1.0	1.9.1.+

In addition to Compose file format versions shown in the table, the Compose itself is on a release schedule, as shown in Compose releases (https://github.com/docker/compose/releases/), but file format versions do not necessarily increment with each release. For example, Compose file format 3.0 was first introduced in Compose release 1.10.0 (https://github.com/docker/compose/releases/tag/1.10.0), and versioned gradually in subsequent releases.

Compose file structure and examples

Example Compose file version 3 ▼

The topics on this reference page are organized alphabetically by top-level key to reflect the structure of the Compose file itself. Top-level keys that define a section in the configuration file such as `build` , `deploy` , `depends_on` , `networks` , and so on, are listed with the options that support them as sub-topics. This maps to the `<key>: <option>: <value>` indent structure of the Compose file.

A good place to start is the Getting Started (https://docs.docker.com/get-started/) tutorial which uses version 3 Compose stack files to implement multi-container apps, service definitions, and swarm mode. Here are some Compose files used in the tutorial.

- Your first docker-compose.yml File (https://docs.docker.com/get-started/part3/#your-first-docker-composeyml-file)
- Add a new service and redeploy (https://docs.docker.com/get-started/part5/#add-a-new-service-and-redeploy)

Another good reference is the Compose file for the voting app sample used in the Docker for Beginners lab (https://github.com/docker/labs/tree/master/beginner/) topic on Deploying an app to a Swarm (https://github.com/docker/labs/blob/master/beginner/chapters/votingapp.md). This is also shown on the accordion at the top of this section.

Service configuration reference

The Compose file is a YAML (http://yaml.org/) file defining services (/compose/compose-file/#service-configuration-reference), networks (/compose/compose-file/#network-configuration-reference) and volumes (/compose/compose-file/#volume-configuration-reference). The default path for a Compose file is `./docker-compose.yml` .

> Tip: You can use either a `.yml` or `.yaml` extension for this file. They both work.

A service definition contains configuration that is applied to each container started for that service, much like passing command-line parameters to `docker container create` . Likewise, network and volume definitions are analogous to `docker network create` and `docker volume create` .

As with `docker container create` , options specified in the Dockerfile, such as `CMD` , `EXPOSE` , `VOLUME` , `ENV` , are respected by default - you don't need to specify them again in `docker-compose.yml` .

You can use environment variables in configuration values with a Bash-like `${VARIABLE}` syntax - see variable substitution (/compose/compose-file/#variable-substitution) for full details.

This section contains a list of all configuration options supported by a service definition in version 3.

build

Configuration options that are applied at build time.

`build` can be specified either as a string containing a path to the build context:

```
version: '3'
services:
  webapp:
    build: ./dir
```

Or, as an object with the path specified under context (/compose/compose-file/#context) and optionally Dockerfile (/compose/compose-file/#dockerfile) and args (/compose/compose-file/#args):

```
version: '3'
services:
  webapp:
    build:
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        buildno: 1
```

If you specify `image` as well as `build`, then Compose names the built image with the `webapp` and optional `tag` specified in `image`:

```
build: ./dir
image: webapp:tag
```

This results in an image named `webapp` and tagged `tag`, built from `./dir`.

> Note: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file. The `docker stack` command accepts only pre-built images.

CONTEXT

Either a path to a directory containing a Dockerfile, or a url to a git repository.

When the value supplied is a relative path, it is interpreted as relative to the location of the Compose file. This directory is also the build context that is sent to the Docker daemon.

Compose builds and tags it with a generated name, and use that image thereafter.

```
build:
  context: ./dir
```

DOCKERFILE

Alternate Dockerfile.

Compose uses an alternate file to build with. A build path must also be specified.

```
build:
  context: .
  dockerfile: Dockerfile-alternate
```

ARGS

Add build arguments, which are environment variables accessible only during the build process.

First, specify the arguments in your Dockerfile:

```
ARG buildno
ARG gitcommithash

RUN echo "Build number: $buildno"
RUN echo "Based on commit: $gitcommithash"
```

Then specify the arguments under the `build` key. You can pass a mapping or a list:

```
build:
  context: .
  args:
    buildno: 1
    gitcommithash: cdc3b19

build:
  context: .
  args:
    - buildno=1
    - gitcommithash=cdc3b19
```

You can omit the value when specifying a build argument, in which case its value at build time is the value in the environment where Compose is running.

```
args:
  - buildno
  - gitcommithash
```

> Note: YAML boolean values (`true` , `false` , `yes` , `no` , `on` , `off`) must be enclosed in quotes, so that the parser interprets them as strings.

CACHE_FROM

> Note: This option is new in v3.2

A list of images that the engine uses for cache resolution.

```
build:
  context: .
  cache_from:
    - alpine:latest
    - corp/web_app:3.14
```

LABELS

> Note: This option is new in v3.3

Add metadata to the resulting image using Docker labels (https://docs.docker.com/engine/userguide/labels-custom-metadata/). You can use either an array or a dictionary.

It's recommended that you use reverse-DNS notation to prevent your labels from conflicting with those used by other software.

```
build:
  context: .
  labels:
    com.example.description: "Accounting webapp"
    com.example.department: "Finance"
    com.example.label-with-empty-value: ""
```

```
build:
  context: .
  labels:
    - "com.example.description=Accounting webapp"
    - "com.example.department=Finance"
    - "com.example.label-with-empty-value"
```

SHM_SIZE

> Added in version 3.5 (https://docs.docker.com/compose/compose-file/compose-versioning/#version-35) file format

Set the size of the `/dev/shm` partition for this build's containers. Specify as an integer value representing the number of bytes or as a string expressing a byte value (/compose/compose-file/#specifying-byte-values).

```
build:
  context: .
  shm_size: '2gb'
```

```
build:
  context: .
  shm_size: 10000000
```

TARGET

> Added in version 3.4 (https://docs.docker.com/compose/compose-file/compose-versioning/#version-34) file format

Build the specified stage as defined inside the `Dockerfile`. See the multi-stage build docs (https://docs.docker.com/engine/userguide/eng-image/multistage-build/) for details.

```
    build:
      context: .
      target: prod
```

cap_add, cap_drop

Add or drop container capabilities. See `man 7 capabilities` for a full list.

```
cap_add:
  - ALL

cap_drop:
  - NET_ADMIN
  - SYS_ADMIN
```

> Note: These options are ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

command

Override the default command.

```
command: bundle exec thin -p 3000
```

The command can also be a list, in a manner similar to dockerfile (https://docs.docker.com/engine/reference/builder/#cmd):

```
command: ["bundle", "exec", "thin", "-p", "3000"]
```

configs

Grant access to configs on a per-service basis using the per-service `configs` configuration. Two different syntax variants are supported.

> Note: The config must already exist or be defined in the top-level `configs` configuration (/compose/compose-file/#configs-configuration-reference) of this stack file, or stack deployment fails.

For more information on configs, see configs (https://docs.docker.com/engine/swarm/configs/).

SHORT SYNTAX

The short syntax variant only specifies the config name. This grants the container access to the config and mounts it at `/<config_name>` within the container. The source name and destination mountpoint are both set to the config name.

The following example uses the short syntax to grant the `redis` service access to the `my_config` and `my_other_config` configs. The value of `my_config` is set to the contents of the file `./my_config.txt` , and `my_other_config` is defined as an external resource, which means that it has already been defined in Docker, either by running the `docker config create` command or by another stack deployment. If the external config does not exist, the stack deployment fails with a `config not found` error.

> Note: `config` definitions are only supported in version 3.3 and higher of the compose file format.

```
version: "3.3"
services:
  redis:
    image: redis:latest
    deploy:
      replicas: 1
    configs:
      - my_config
      - my_other_config
configs:
  my_config:
    file: ./my_config.txt
  my_other_config:
    external: true
```

LONG SYNTAX

The long syntax provides more granularity in how the config is created within the service's task containers.

- `source` : The name of the config as it exists in Docker.
- `target` : The path and name of the file to be mounted in the service's task containers. Defaults to `/<source>` if not specified.
- `uid` and `gid` : The numeric UID or GID that owns the mounted config file within in the service's task containers. Both default to `0` on Linux if not specified. Not supported on Windows.
- `mode` : The permissions for the file that is mounted within the service's task containers, in octal notation. For instance, `0444` represents world-readable. The default is `0444` . Configs cannot be writable because they are mounted in a temporary filesystem, so if you set the writable bit, it is ignored. The executable bit can be set. If you aren't familiar with UNIX file permission modes, you may find this permissions calculator (http://permissions-calculator.org/) useful.

The following example sets the name of `my_config` to `redis_config` within the container, sets the mode to `0440` (group-readable) and sets the user and group to `103` . The `redis` service does not have access to the `my_other_config` config.

```
version: "3.3"
services:
  redis:
    image: redis:latest
    deploy:
      replicas: 1
    configs:
      - source: my_config
        target: /redis_config
        uid: '103'
        gid: '103'
        mode: 0440
configs:
  my_config:
    file: ./my_config.txt
  my_other_config:
    external: true
```

You can grant a service access to multiple configs and you can mix long and short syntax. Defining a config does not imply granting a service access to it.

cgroup_parent

Specify an optional parent cgroup for the container.

```
cgroup_parent: m-executor-abcd
```

> Note: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

container_name

Specify a custom container name, rather than a generated default name.

```
container_name: my-web-container
```

Because Docker container names must be unique, you cannot scale a service beyond 1 container if you have specified a custom name. Attempting to do so results in an error.

> Note: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

credential_spec

> Note: this option was added in v3.3

Configure the credential spec for managed service account. This option is only used for services using Windows containers. The `credential_spec` must be in the format `file://<filename>` or `registry://<value-name>` .

When using `file:` , the referenced file must be present in the `CredentialSpecs` subdirectory in the docker data directory, which defaults to `C:\ProgramData\Docker\` on Windows. The following example loads the credential spec from a file named `C:\ProgramData\Docker\CredentialSpecs\my-credential-spec.json` :

```
credential_spec:
  file: my-credential-spec.json
```

When using `registry:` , the credential spec is read from the Windows registry on the daemon's host. A registry value with the given name must be located in:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Virtualization\Con
tainers\CredentialSpecs
```

The following example load the credential spec from a value named `my-credential-spec` in the registry:

```
credential_spec:
  registry: my-credential-spec
```

deploy

> Version 3 (https://docs.docker.com/compose/compose-file/compose-versioning/#version-3) only.

Specify configuration related to the deployment and running of services. This only takes effect when deploying to a swarm (https://docs.docker.com/engine/swarm/) with docker stack deploy (https://docs.docker.com/engine/reference/commandline/stack_deploy/), and is ignored by `docker-compose up` and `docker-compose run` .

```
version: '3'
services:
  redis:
    image: redis:alpine
    deploy:
      replicas: 6
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
```

Several sub-options are available:

ENDPOINT_MODE

Specify a service discovery method for external clients connecting to a swarm.

> Version 3.3 (https://docs.docker.com/compose/compose-file/compose-versioning/#version-3) only.

- `endpoint_mode: vip` - Docker assigns the service a virtual IP (VIP) that acts as the “front end” for clients to reach the service on a network. Docker routes requests between the client and available worker nodes for the service, without client knowledge of how many nodes are participating in the service or their IP addresses or ports. (This is the default.)
- `endpoint_mode: dnsrr` - DNS round-robin (DNSRR) service discovery does not use a single virtual IP. Docker sets up DNS entries for the service such that a DNS query for the service name returns a list of IP addresses, and the client connects directly to one of these. DNS round-robin is useful in cases where you want to use your own load balancer, or for Hybrid Windows and Linux applications.

```
version: "3.3"

services:
  wordpress:
    image: wordpress
    ports:
      - "8080:80"
    networks:
      - overlay
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: vip

  mysql:
    image: mysql
    volumes:
       - db-data:/var/lib/mysql/data
    networks:
       - overlay
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: dnsrr

volumes:
  db-data:

networks:
  overlay:
```

The options for `endpoint_mode` also work as flags on the swarm mode CLI command docker service create (https://docs.docker.com/engine/reference/commandline/service_create/). For a quick list of all swarm related `docker` commands, see Swarm mode CLI commands (https://docs.docker.com/engine/swarm/#swarm-mode-key-concepts-and-tutorial).

To learn more about service discovery and networking in swarm mode, see Configure service discovery (https://docs.docker.com/engine/swarm/networking/#configure-service-discovery) in the swarm mode topics.

LABELS

Specify labels for the service. These labels are *only* set on the service, and *not* on any containers for the service.

```
version: "3"
services:
  web:
    image: web
    deploy:
      labels:
        com.example.description: "This label will appear on the web
service"
```

To set labels on containers instead, use the `labels` key outside of `deploy` :

```
version: "3"
services:
  web:
    image: web
    labels:
      com.example.description: "This label will appear on all contai
ners for the web service"
```

MODE

Either `global` (exactly one container per swarm node) or `replicated` (a specified number of containers). The default is `replicated`. (To learn more, see Replicated and global services (https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/#replicated-and-global-services) in the swarm (https://docs.docker.com/engine/swarm/) topics.)

```
version: '3'
services:
  worker:
    image: dockersamples/examplevotingapp_worker
    deploy:
      mode: global
```

PLACEMENT

Specify placement of constraints and preferences. See the docker service create documentation for a full description of the syntax and available types of constraints (https://docs.docker.com/engine/reference/commandline/service_create/#specify-service-constraints-constraint) and preferences (https://docs.docker.com/engine/reference/commandline/service_create/#specify-service-placement-preferences-placement-pref).

```
version: '3.3'
services:
  db:
    image: postgres
    deploy:
      placement:
        constraints:
          - node.role == manager
          - engine.labels.operatingsystem == ubuntu 14.04
        preferences:
          - spread: node.labels.zone
```

REPLICAS

If the service is `replicated` (which is the default), specify the number of containers that should be running at any given time.

```
version: '3'
services:
  worker:
    image: dockersamples/examplevotingapp_worker
    networks:
      - frontend
      - backend
    deploy:
      mode: replicated
      replicas: 6
```

RESOURCES

Configures resource constraints.

> Note: This replaces the older resource constraint options (https://docs.docker.com/compose/compose-file/compose-file-v2/#cpu-and-other-resources) for non swarm mode in Compose files prior to version 3 (`cpu_shares` , `cpu_quota` , `cpuset` , `mem_limit` , `memswap_limit` , `mem_swappiness`), as described in Upgrading version 2.x to 3.x (https://docs.docker.com/compose/compose-file/compose-versioning/#upgrading).

Each of these is a single value, analogous to its docker service create (https://docs.docker.com/engine/reference/commandline/service_create/) counterpart.

In this general example, the `redis` service is constrained to use no more than 50M of memory and `0.50` (50%) of available processing time (CPU), and has `20M` of memory and `0.25` CPU time reserved (as always available to it).

```
version: '3'
services:
  redis:
    image: redis:alpine
    deploy:
      resources:
        limits:
          cpus: '0.50'
          memory: 50M
        reservations:
          cpus: '0.25'
          memory: 20M
```

The topics below describe available options to set resource constraints on services or containers in a swarm.

> Looking for options to set resources on non swarm mode containers?
>
> The options described here are specific to the `deploy` key and swarm mode. If you want to set resource constraints on non swarm deployments, use Compose file format version 2 CPU, memory, and other resource options (https://docs.docker.com/compose/compose-file/compose-file-v2/#cpu-and-other-resources). If you have further questions, refer to the discussion on the GitHub issue docker/compose/4513 (https://github.com/docker/compose/issues/4513).

Out Of Memory Exceptions (OOME)

If your services or containers attempt to use more memory than the system has available, you may experience an Out Of Memory Exception (OOME) and a container, or the Docker daemon, might be killed by the kernel OOM killer. To prevent this from happening, ensure that your application runs on hosts with adequate memory and see Understand the risks of running out of memory (https://docs.docker.com/engine/admin/resource_constraints/#understand-the-risks-of-running-out-of-memory).

RESTART_POLICY

Configures if and how to restart containers when they exit. Replaces `restart` (https://docs.docker.com/compose/compose-file/compose-file-v2/#orig-resources).

- `condition` : One of `none` , `on-failure` or `any` (default: `any`).
- `delay` : How long to wait between restart attempts, specified as a duration (/compose/compose-file/#specifying-durations) (default: 0).
- `max_attempts` : How many times to attempt to restart a container before giving up (default: never give up). If the restart does not succeed within the configured `window` , this attempt doesn't count toward the configured `max_attempts` value. For example, if `max_attempts` is set to '2', and the restart fails on the first attempt, more than two restarts may be attempted.
- `window` : How long to wait before deciding if a restart has succeeded, specified as a duration (/compose/compose-file/#specifying-durations) (default: decide immediately).

```
version: "3"
services:
  redis:
    image: redis:alpine
    deploy:
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 3
        window: 120s
```

ROLLBACK_CONFIG

Version 3.7 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-37) and up

Configures how the service should be rollbacked in case of a failing update.

- `parallelism` : The number of containers to rollback at a time. If set to 0, all containers rollback simultaneously.
- `delay` : The time to wait between each container group's rollback (default 0s).
- `failure_action` : What to do if a rollback fails. One of `continue` or `pause` (default `pause`)
- `monitor` : Duration after each task update to monitor for failure `(ns|us|ms|s|m|h)` (default 0s).
- `max_failure_ratio` : Failure rate to tolerate during a rollback (default 0).

- `order` : Order of operations during rollbacks. One of `stop-first` (old task is stopped before starting new one), or `start-first` (new task is started first, and the running tasks briefly overlap) (default `stop-first`).

UPDATE_CONFIG

Configures how the service should be updated. Useful for configuring rolling updates.

- `parallelism` : The number of containers to update at a time.
- `delay` : The time to wait between updating a group of containers.
- `failure_action` : What to do if an update fails. One of `continue` , `rollback` , or `pause` (default: `pause`).
- `monitor` : Duration after each task update to monitor for failure `(ns|us|ms|s|m|h)` (default 0s).
- `max_failure_ratio` : Failure rate to tolerate during an update.
- `order` : Order of operations during updates. One of `stop-first` (old task is stopped before starting new one), or `start-first` (new task is started first, and the running tasks briefly overlap) (default `stop-first`) Note: Only supported for v3.4 and higher.

> Note: `order` is only supported for v3.4 and higher of the compose file format.

```
version: '3.4'
services:
  vote:
    image: dockersamples/examplevotingapp_vote:before
    depends_on:
      - redis
    deploy:
      replicas: 2
      update_config:
        parallelism: 2
        delay: 10s
        order: stop-first
```

NOT SUPPORTED FOR `DOCKER STACK DEPLOY`

The following sub-options (supported for `docker-compose up` and `docker-compose run`) are *not supported* for `docker stack deploy` or the `deploy` key.

- build (/compose/compose-file/#build)

- cgroup_parent (/compose/compose-file/#cgroup_parent)
- container_name (/compose/compose-file/#container_name)
- devices (/compose/compose-file/#devices)
- tmpfs (/compose/compose-file/#tmpfs)
- external_links (/compose/compose-file/#external_links)
- links (/compose/compose-file/#links)
- network_mode (/compose/compose-file/#network_mode)
- restart (/compose/compose-file/#restart)
- security_opt (/compose/compose-file/#security_opt)
- stop_signal (/compose/compose-file/#stop_signal)
- sysctls (/compose/compose-file/#sysctls)
- userns_mode (/compose/compose-file/#userns_mode)

> Tip: See the section on how to configure volumes for services, swarms, and docker-stack.yml files (/compose/compose-file/#volumes-for-services-swarms-and-stack-files). Volumes *are* supported but to work with swarms and services, they must be configured as named volumes or associated with services that are constrained to nodes with access to the requisite volumes.

devices

List of device mappings. Uses the same format as the `--device` docker client create option.

```
devices:
  - "/dev/ttyUSB0:/dev/ttyUSB0"
```

> Note: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

depends_on

Express dependency between services, Service dependencies cause the following behaviors:

- `docker-compose up` starts services in dependency order. In the following example, `db` and `redis` are started before `web` .

- `docker-compose up SERVICE` automatically includes `SERVICE`'s dependencies. In the following example, `docker-compose up web` also creates and starts `db` and `redis`.
- `docker-compose stop` stops services in dependency order. In the following example, `web` is stopped before `db` and `redis`.

Simple example:

```
version: '3'
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres
```

✓ There are several things to be aware of when using **`depends_on`**:

- `depends_on` does not wait for `db` and `redis` to be "ready" before starting `web` - only until they have been started. If you need to wait for a service to be ready, see Controlling startup order (https://docs.docker.com/compose/startup-order/) for more on this problem and strategies for solving it.
- Version 3 no longer supports the `condition` form of `depends_on`.
- The `depends_on` option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a version 3 Compose file.

dns

Custom DNS servers. Can be a single value or a list.

```
dns: 8.8.8.8
dns:
  - 8.8.8.8
  - 9.9.9.9
```

dns_search

Custom DNS search domains. Can be a single value or a list.

```
dns_search: example.com
dns_search:
  - dc1.example.com
  - dc2.example.com
```

tmpfs

> Version 2 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-2) and up.

Mount a temporary file system inside the container. Can be a single value or a list.

```
tmpfs: /run
tmpfs:
  - /run
  - /tmp
```

> Note: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3-3.5) Compose file.

> Version 3.6 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-3) and up.

Mount a temporary file system inside the container. Size parameter specifies the size of the tmpfs mount in bytes. Unlimited by default.

```
- type: tmpfs
  target: /app
  tmpfs:
    size: 1000
```

entrypoint

Override the default entrypoint.

```
entrypoint: /code/entrypoint.sh
```

The entrypoint can also be a list, in a manner similar to dockerfile (https://docs.docker.com/engine/reference/builder/#entrypoint):

```
entrypoint:
    - php
    - -d
    - zend_extension=/usr/local/lib/php/extensions/no-debug-non-zts-
20100525/xdebug.so
    - -d
    - memory_limit=-1
    - vendor/bin/phpunit
```

> Note: Setting `entrypoint` both overrides any default entrypoint set on the service's image with the `ENTRYPOINT` Dockerfile instruction, *and* clears out any default command on the image - meaning that if there's a `CMD` instruction in the Dockerfile, it is ignored.

env_file

Add environment variables from a file. Can be a single value or a list.

If you have specified a Compose file with `docker-compose -f FILE`, paths in `env_file` are relative to the directory that file is in.

Environment variables declared in the environment (/compose/compose-file/#environment) section *override* these values – this holds true even if those values are empty or undefined.

```
env_file: .env

env_file:
  - ./common.env
  - ./apps/web.env
  - /opt/secrets.env
```

Compose expects each line in an env file to be in `VAR=VAL` format. Lines beginning with `#` are treated as comments and are ignored. Blank lines are also ignored.

```
# Set Rails/Rack environment
RACK_ENV=development
```

> Note: If your service specifies a build (/compose/compose-file/#build) option, variables defined in environment files are *not* automatically visible during the build. Use the args (/compose/compose-file/#args) sub-option of `build` to define build-time environment variables.

The value of `VAL` is used as is and not modified at all. For example if the value is surrounded by quotes (as is often the case of shell variables), the quotes are included in the value passed to Compose.

Keep in mind that *the order of files in the list is significant in determining the value assigned to a variable that shows up more than once*. The files in the list are processed from the top down. For the same variable specified in file `a.env` and assigned a different value in file `b.env` , if `b.env` is listed below (after), then the value from `b.env` stands. For example, given the following declaration in `docker_compose.yml` :

```
services:
  some-service:
    env_file:
      - a.env
      - b.env
```

And the following files:

```
# a.env
VAR=1
```

and

```
# b.env
VAR=hello
```

$VAR is `hello` .

environment

Add environment variables. You can use either an array or a dictionary. Any boolean values; true, false, yes no, need to be enclosed in quotes to ensure they are not converted to True or False by the YML parser.

Environment variables with only a key are resolved to their values on the machine Compose is running on, which can be helpful for secret or host-specific values.

```
environment:
  RACK_ENV: development
  SHOW: 'true'
  SESSION_SECRET:

environment:
  - RACK_ENV=development
  - SHOW=true
  - SESSION_SECRET
```

> Note: If your service specifies a build (/compose/compose-file/#build) option, variables defined in `environment` are *not* automatically visible during the build. Use the args (/compose/compose-file/#args) sub-option of `build` to define build-time environment variables.

expose

Expose ports without publishing them to the host machine - they'll only be accessible to linked services. Only the internal port can be specified.

```
expose:
 - "3000"
 - "8000"
```

external_links

Link to containers started outside this `docker-compose.yml` or even outside of Compose, especially for containers that provide shared or common services. `external_links` follow semantics similar to the legacy option `links` when specifying both the container name and the link alias (`CONTAINER:ALIAS`).

```
external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql
```

> Notes:
>
> If you're using the version 2 or above file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-2), the externally-created containers must be connected to at least one of the same networks as the service that is linking to them. Links (https://docs.docker.com/compose/compose-file/compose-file-v2#links) are a legacy option. We recommend using networks (/compose/compose-file/#networks) instead.
>
> This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

extra_hosts

Add hostname mappings. Use the same values as the docker client `--add-host` parameter.

```
extra_hosts:
 - "somehost:162.242.195.82"
 - "otherhost:50.31.209.229"
```

An entry with the ip address and hostname is created in `/etc/hosts` inside containers for this service, e.g:

```
162.242.195.82  somehost
50.31.209.229   otherhost
```

healthcheck

> Version 2.1 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-21) and up.

Configure a check that's run to determine whether or not containers for this service are "healthy". See the docs for the HEALTHCHECK Dockerfile instruction (https://docs.docker.com/engine/reference/builder/#healthcheck) for details on how healthchecks work.

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost"]
  interval: 1m30s
  timeout: 10s
  retries: 3
  start_period: 40s
```

`interval` , `timeout` and `start_period` are specified as durations (/compose/compose-file/#specifying-durations).

> Note: `start_period` is only supported for v3.4 and higher of the compose file format.

`test` must be either a string or a list. If it's a list, the first item must be either `NONE` , `CMD` or `CMD-SHELL` . If it's a string, it's equivalent to specifying `CMD-SHELL` followed by that string.

```
# Hit the local web app
test: ["CMD", "curl", "-f", "http://localhost"]

# As above, but wrapped in /bin/sh. Both forms below are equivalent.
test: ["CMD-SHELL", "curl -f http://localhost || exit 1"]
test: curl -f https://localhost || exit 1
```

To disable any default healthcheck set by the image, you can use `disable: true` . This is equivalent to specifying `test: ["NONE"]` .

```
healthcheck:
  disable: true
```

image

Specify the image to start the container from. Can either be a repository/tag or a partial image ID.

```
image: redis
image: ubuntu:14.04
image: tutum/influxdb
image: example-registry.com:4000/postgresql
image: a4bc65fd
```

If the image does not exist, Compose attempts to pull it, unless you have also specified build (/compose/compose-file/#build), in which case it builds it using the specified options and tags it with the specified tag.

init

> Added in version 3.7 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-37).

Run an init inside the container that forwards signals and reaps processes. Either set a boolean value to use the default `init`, or specify a path to a custom one.

```
version: '3.7'
services:
  web:
    image: alpine:latest
    init: true
```

```
version: '2.2'
services:
  web:
    image: alpine:latest
    init: /usr/libexec/docker-init
```

isolation

Specify a container's isolation technology. On Linux, the only supported value is `default`. On Windows, acceptable values are `default`, `process` and `hyperv`. Refer to the Docker Engine docs (https://docs.docker.com/engine/reference/commandline/run/#specify-isolation-technology-for-container---isolation) for details.

labels

Add metadata to containers using Docker labels (https://docs.docker.com/engine/userguide/labels-custom-metadata/). You can use either an array or a dictionary.

It's recommended that you use reverse-DNS notation to prevent your labels from conflicting with those used by other software.

```
labels:
  com.example.description: "Accounting webapp"
  com.example.department: "Finance"
  com.example.label-with-empty-value: ""

labels:
  - "com.example.description=Accounting webapp"
  - "com.example.department=Finance"
  - "com.example.label-with-empty-value"
```

links

Warning: >The `--link` flag is a legacy feature of Docker. It may eventually be removed. Unless you absolutely need to continue using it, we recommend that you use user-defined networks (https://docs.docker.com/engine/userguide/networking//#user-defined-networks) to facilitate communication between two containers instead of using `--link`. One feature that user-defined networks do not support that you can do with `--link` is sharing environmental variables between containers. However, you can use other mechanisms such as volumes to share environment variables between containers in a more controlled way.

Link to containers in another service. Either specify both the service name and a link alias (`SERVICE:ALIAS`), or just the service name.

```
web:
  links:
   - db
   - db:database
   - redis
```

Containers for the linked service are reachable at a hostname identical to the alias, or the service name if no alias was specified.

Links are not required to enable services to communicate - by default, any service can reach any other service at that service’s name. (See also, the Links topic in Networking in Compose (https://docs.docker.com/compose/networking/#links).)

Links also express dependency between services in the same way as depends_on (/compose/compose-file/#depends_on), so they determine the order of service startup.

✓ Notes

- If you define both links and networks (/compose/compose-file/#networks), services with links between them must share at least one network in common to communicate.
- This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

logging

Logging configuration for the service.

```
logging:
  driver: syslog
  options:
    syslog-address: "tcp://192.168.0.42:123"
```

The `driver` name specifies a logging driver for the service's containers, as with the `--log-driver` option for docker run (documented here (https://docs.docker.com/engine/admin/logging/overview/)).

The default value is json-file.

```
driver: "json-file"
driver: "syslog"
driver: "none"
```

> Note: Only the `json-file` and `journald` drivers make the logs available directly from `docker-compose up` and `docker-compose logs` . Using any other driver does not print any logs.

Specify logging options for the logging driver with the `options` key, as with the `--log-opt` option for `docker run` .

Logging options are key-value pairs. An example of `syslog` options:

```
driver: "syslog"
options:
  syslog-address: "tcp://192.168.0.42:123"
```

The default driver json-file (https://docs.docker.com/engine/admin/logging/overview/#json-file), has options to limit the amount of logs stored. To do this, use a key-value pair for maximum storage size and maximum number of files:

```
options:
  max-size: "200k"
  max-file: "10"
```

The example shown above would store log files until they reach a `max-size` of 200kB, and then rotate them. The amount of individual log files stored is specified by the `max-file` value. As logs grow beyond the max limits, older log files are removed to allow storage of new logs.

Here is an example `docker-compose.yml` file that limits logging storage:

```
services:
  some-service:
    image: some-service
    logging:
      driver: "json-file"
      options:
        max-size: "200k"
        max-file: "10"
```

Logging options available depend on which logging driver you use

The above example for controlling log files and sizes uses options specific to the json-file driver (https://docs.docker.com/engine/admin/logging/overview/#json-file). These particular options are not available on other logging drivers. For a full list of supported logging drivers and their options, see logging drivers (https://docs.docker.com/engine/admin/logging/overview/).

network_mode

Network mode. Use the same values as the docker client `--network` parameter, plus the special form `service:[service name]` .

```
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
network_mode: "container:[container name/id]"
```

Notes

- This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.
- `network_mode: "host"` cannot be mixed with links (/compose/compose-file/#links).

networks

Networks to join, referencing entries under the top-level `networks` key (/compose/compose-file/#network-configuration-reference).

```
services:
  some-service:
    networks:
     - some-network
     - other-network
```

ALIASES

Aliases (alternative hostnames) for this service on the network. Other containers on the same network can use either the service name or this alias to connect to one of the service's containers.

Since `aliases` is network-scoped, the same service can have different aliases on different networks.

> Note: A network-wide alias can be shared by multiple containers, and even by multiple services. If it is, then exactly which container the name resolves to is not guaranteed.

The general format is shown here.

```
services:
  some-service:
    networks:
      some-network:
        aliases:
         - alias1
         - alias3
      other-network:
        aliases:
         - alias2
```

In the example below, three services are provided (`web` , `worker` , and `db`), along with two networks (`new` and `legacy`). The `db` service is reachable at the hostname `db` or `database` on the `new` network, and at `db` or `mysql` on the `legacy` network.

```
version: '2'

services:
  web:
    build: ./web
    networks:
      - new

  worker:
    build: ./worker
    networks:
      - legacy

  db:
    image: mysql
    networks:
      new:
        aliases:
          - database
      legacy:
        aliases:
          - mysql

networks:
  new:
  legacy:
```

IPV4_ADDRESS, IPV6_ADDRESS

Specify a static IP address for containers for this service when joining the network.

The corresponding network configuration in the top-level networks section (/compose/compose-file/#network-configuration-reference) must have an `ipam` block with subnet configurations covering each static address. If IPv6 addressing is desired, the `enable_ipv6` (/compose/compose-file/#enableipv6) option must be set, and you must use a version 2.x Compose file, such as the one below.

> Note: These options do not currently work in swarm mode.

An example:

```
version: '2.1'

services:
  app:
    image: busybox
    command: ifconfig
    networks:
      app_net:
        ipv4_address: 172.16.238.10
        ipv6_address: 2001:3984:3989::10

networks:
  app_net:
    driver: bridge
    enable_ipv6: true
    ipam:
      driver: default
      config:
      -
        subnet: 172.16.238.0/24
      -
        subnet: 2001:3984:3989::/64
```

pid

```
pid: "host"
```

Sets the PID mode to the host PID mode. This turns on sharing between container and the host operating system the PID address space. Containers launched with this flag can access and manipulate other containers in the bare-metal machine’s namespace and vice versa.

ports

Expose ports.

> Note: Port mapping is incompatible with `network_mode: host`

SHORT SYNTAX

Either specify both ports (`HOST:CONTAINER`), or just the container port (an ephemeral host port is chosen).

> Note: When mapping ports in the `HOST:CONTAINER` format, you may experience erroneous results when using a container port lower than 60, because YAML parses numbers in the format `xx:yy` as a base-60 value. For this reason, we recommend always explicitly specifying your port mappings as strings.

```
ports:
 - "3000"
 - "3000-3005"
 - "8000:8000"
 - "9090-9091:8080-8081"
 - "49100:22"
 - "127.0.0.1:8001:8001"
 - "127.0.0.1:5000-5010:5000-5010"
 - "6060:6060/udp"
```

LONG SYNTAX

The long form syntax allows the configuration of additional fields that can't be expressed in the short form.

- `target` : the port inside the container
- `published` : the publicly exposed port
- `protocol` : the port protocol (`tcp` or `udp`)
- `mode` : `host` for publishing a host port on each node, or `ingress` for a swarm mode port to be load balanced.

```
ports:
  - target: 80
    published: 8080
    protocol: tcp
    mode: host
```

> Note: The long syntax is new in v3.2

secrets

Grant access to secrets on a per-service basis using the per-service `secrets` configuration. Two different syntax variants are supported.

> Note: The secret must already exist or be defined in the top-level `secrets` configuration (/compose/compose-file/#secrets-configuration-reference) of this stack file, or stack deployment fails.

For more information on secrets, see secrets (https://docs.docker.com/engine/swarm/secrets/).

SHORT SYNTAX

The short syntax variant only specifies the secret name. This grants the container access to the secret and mounts it at `/run/secrets/<secret_name>` within the container. The source name and destination mountpoint are both set to the secret name.

The following example uses the short syntax to grant the `redis` service access to the `my_secret` and `my_other_secret` secrets. The value of `my_secret` is set to the contents of the file `./my_secret.txt` , and `my_other_secret` is defined as an external resource, which means that it has already been defined in Docker, either by running the `docker secret create` command or by another stack deployment. If the external secret does not exist, the stack deployment fails with a `secret not found` error.

```
version: "3.1"
services:
  redis:
    image: redis:latest
    deploy:
      replicas: 1
    secrets:
      - my_secret
      - my_other_secret
secrets:
  my_secret:
    file: ./my_secret.txt
  my_other_secret:
    external: true
```

LONG SYNTAX

The long syntax provides more granularity in how the secret is created within the service's task containers.

- `source` : The name of the secret as it exists in Docker.
- `target` : The name of the file to be mounted in `/run/secrets/` in the service's task containers. Defaults to `source` if not specified.
- `uid` and `gid` : The numeric UID or GID that owns the file within `/run/secrets/` in the service's task containers. Both default to `0` if not specified.
- `mode` : The permissions for the file to be mounted in `/run/secrets/` in the service's task containers, in octal notation. For instance, `0444` represents world-readable. The default in Docker 1.13.1 is `0000` , but is be `0444` in newer versions. Secrets cannot be writable because they are mounted in a temporary filesystem, so if you set the writable bit, it is ignored. The executable bit can be set. If you aren't familiar with UNIX file permission modes, you may find this permissions calculator (http://permissions-calculator.org/) useful.

The following example sets name of the `my_secret` to `redis_secret` within the container, sets the mode to `0440` (group-readable) and sets the user and group to `103` . The `redis` service does not have access to the `my_other_secret` secret.

```
version: "3.1"
services:
  redis:
    image: redis:latest
    deploy:
      replicas: 1
    secrets:
      - source: my_secret
        target: redis_secret
        uid: '103'
        gid: '103'
        mode: 0440
secrets:
  my_secret:
    file: ./my_secret.txt
  my_other_secret:
    external: true
```

You can grant a service access to multiple secrets and you can mix long and short syntax. Defining a secret does not imply granting a service access to it.

security_opt

Override the default labeling scheme for each container.

```
security_opt:
  - label:user:USER
  - label:role:ROLE
```

> Note: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

stop_grace_period

Specify how long to wait when attempting to stop a container if it doesn’t handle SIGTERM (or whatever stop signal has been specified with `stop_signal` (/compose/compose-file/#stopsignal)), before sending SIGKILL. Specified as a duration (/compose/compose-file/#specifying-durations).

```
stop_grace_period: 1s
stop_grace_period: 1m30s
```

By default, `stop` waits 10 seconds for the container to exit before sending SIGKILL.

stop_signal

Sets an alternative signal to stop the container. By default `stop` uses SIGTERM. Setting an alternative signal using `stop_signal` causes `stop` to send that signal instead.

```
stop_signal: SIGUSR1
```

> Note: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

sysctls

Kernel parameters to set in the container. You can use either an array or a dictionary.

```
sysctls:
  net.core.somaxconn: 1024
  net.ipv4.tcp_syncookies: 0

sysctls:
  - net.core.somaxconn=1024
  - net.ipv4.tcp_syncookies=0
```

> Note: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

ulimits

Override the default ulimits for a container. You can either specify a single limit as an integer or soft/hard limits as a mapping.

```
ulimits:
  nproc: 65535
  nofile:
    soft: 20000
    hard: 40000
```

userns_mode

```
userns_mode: "host"
```

Disables the user namespace for this service, if Docker daemon is configured with user namespaces. See dockerd (https://docs.docker.com/engine/reference/commandline/dockerd/#disable-user-namespace-for-a-container) for more information.

> Note: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

volumes

Mount host paths or named volumes, specified as sub-options to a service.

You can mount a host path as part of a definition for a single service, and there is no need to define it in the top level `volumes` key.

But, if you want to reuse a volume across multiple services, then define a named volume in the top-level `volumes` key (/compose/compose-file/#volume-configuration-reference). Use named volumes with services, swarms, and stack files (/compose/compose-file/#volumes-for-services-swarms-and-stack-files).

> Note: The top-level volumes (/compose/compose-file/#volume-configuration-reference) key defines a named volume and references it from each service's `volumes` list. This replaces `volumes_from` in earlier versions of the Compose file format. See Use volumes (https://docs.docker.com/engine/admin/volumes/volumes/) and Volume Plugins (https://docs.docker.com/engine/extend/plugins_volume/) for general information on volumes.

This example shows a named volume (`mydata`) being used by the `web` service, and a bind mount defined for a single service (first path under `db` service `volumes`). The `db` service also uses a named volume called `dbdata` (second path under `db` service `volumes`), but defines it using the old string format for mounting a named volume. Named volumes must be listed under the top-level `volumes` key, as shown.

```
version: "3.2"
services:
  web:
    image: nginx:alpine
    volumes:
      - type: volume
        source: mydata
        target: /data
        volume:
          nocopy: true
      - type: bind
        source: ./static
        target: /opt/app/static

  db:
    image: postgres:latest
    volumes:
      - "/var/run/postgres/postgres.sock:/var/run/postgres/postgres.
sock"
      - "dbdata:/var/lib/postgresql/data"

volumes:
  mydata:
  dbdata:
```

> Note: See Use volumes (https://docs.docker.com/engine/admin/volumes/volumes/) and Volume Plugins (https://docs.docker.com/engine/extend/plugins_volume/) for general information on volumes.

SHORT SYNTAX

Optionally specify a path on the host machine (`HOST:CONTAINER`), or an access mode (`HOST:CONTAINER:ro`).

You can mount a relative path on the host, that expands relative to the directory of the Compose configuration file being used. Relative paths should always begin with `.` or `..` .

```
volumes:
  # Just specify a path and let the Engine create a volume
  - /var/lib/mysql

  # Specify an absolute path mapping
  - /opt/data:/var/lib/mysql

  # Path on the host, relative to the Compose file
  - ./cache:/tmp/cache

  # User-relative path
  - ~/configs:/etc/configs/:ro

  # Named volume
  - datavolume:/var/lib/mysql
```

LONG SYNTAX

The long form syntax allows the configuration of additional fields that can't be expressed in the short form.

- `type` : the mount type `volume` , `bind` or `tmpfs`
- `source` : the source of the mount, a path on the host for a bind mount, or the name of a volume defined in the top-level `volumes` key (/compose/compose-file/#volume-configuration-reference). Not applicable for a tmpfs mount.
- `target` : the path in the container where the volume is mounted
- `read_only` : flag to set the volume as read-only

- `bind` : configure additional bind options
  - `propagation` : the propagation mode used for the bind
- `volume` : configure additional volume options
  - `nocopy` : flag to disable copying of data from a container when a volume is created
- `tmpfs` : configure additional tmpfs options
  - `size` : the size for the tmpfs mount in bytes
- `consistency` : the consistency requirements of the mount, one of `consistent` (host and container have identical view), `cached` (read cache, host view is authoritative) or `delegated` (read-write cache, container's view is authoritative)

```
version: "3.2"
services:
  web:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - type: volume
        source: mydata
        target: /data
        volume:
          nocopy: true
      - type: bind
        source: ./static
        target: /opt/app/static

networks:
  webnet:

volumes:
  mydata:
```

> Note: The long syntax is new in v3.2

VOLUMES FOR SERVICES, SWARMS, AND STACK FILES

When working with services, swarms, and `docker-stack.yml` files, keep in mind that the tasks (containers) backing a service can be deployed on any node in a swarm, and this may be a different node each time the service is updated.

In the absence of having named volumes with specified sources, Docker creates an anonymous volume for each task backing a service. Anonymous volumes do not persist after the associated containers are removed.

If you want your data to persist, use a named volume and a volume driver that is multi-host aware, so that the data is accessible from any node. Or, set constraints on the service so that its tasks are deployed on a node that has the volume present.

As an example, the `docker-stack.yml` file for the votingapp sample in Docker Labs (https://github.com/docker/labs/blob/master/beginner/chapters/votingapp.md) defines a service called `db` that runs a `postgres` database. It is configured as a named volume to persist the data on the swarm, *and* is constrained to run only on `manager` nodes. Here is the relevant snip-it from that file:

```
version: "3"
services:
  db:
    image: postgres:9.4
    volumes:
      - db-data:/var/lib/postgresql/data
    networks:
      - backend
    deploy:
      placement:
        constraints: [node.role == manager]
```

CACHING OPTIONS FOR VOLUME MOUNTS (DOCKER FOR MAC)

On Docker 17.04 CE Edge and up, including 17.06 CE Edge and Stable, you can configure container-and-host consistency requirements for bind-mounted directories in Compose files to allow for better performance on read/write of volume mounts. These options address issues specific to `osxfs` file sharing, and therefore are only applicable on Docker for Mac.

The flags are:

- `consistent` : Full consistency. The container runtime and the host maintain an identical view of the mount at all times. This is the default.
- `cached` : The host's view of the mount is authoritative. There may be delays before updates made on the host are visible within a container.

- `delegated` : The container runtime's view of the mount is authoritative. There may be delays before updates made in a container are visible on the host.

Here is an example of configuring a volume as `cached` :

```
version: '3'
services:
  php:
    image: php:7.1-fpm
    ports:
      - "9000"
    volumes:
      - .:/var/www/project:cached
```

Full detail on these flags, the problems they solve, and their `docker run` counterparts is in the Docker for Mac topic Performance tuning for volume mounts (shared filesystems) (https://docs.docker.com/docker-for-mac/osxfs-caching/).

restart

`no` is the default restart policy, and it does not restart a container under any circumstance. When `always` is specified, the container always restarts. The `on-failure` policy restarts a container if the exit code indicates an on-failure error.

```
restart: "no"
restart: always
restart: on-failure
restart: unless-stopped
```

> Note: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file. Use restart_policy (/compose/compose-file/#restart_policy) instead.

domainname, hostname, ipc, mac_address, privileged, read_only, shm_size, stdin_open, tty, user, working_dir

Each of these is a single value, analogous to its docker run (https://docs.docker.com/engine/reference/run/) counterpart.

```
user: postgresql
working_dir: /code

domainname: foo.com
hostname: foo
ipc: host
mac_address: 02:42:ac:11:65:43

privileged: true


read_only: true
shm_size: 64M
stdin_open: true
tty: true
```

Specifying durations

Some configuration options, such as the `interval` and `timeout` sub-options for `check` (/compose/compose-file/#healthcheck), accept a duration as a string in a format that looks like this:

```
2.5s
10s
1m30s
2h32m
5h34m56s
```

The supported units are `us` , `ms` , `s` , `m` and `!` .

Specifying byte values

Some configuration options, such as the `shm_size` sub-option for `build` (/compose/compose-file/#build), accept a byte value as a string in a format that looks like this:

```
2b
1024kb
2048k
300m
1gb
```

The supported units are `b` , `k` , `m` and `g` , and their alternative notation `kb` , `mb` and `gb` . Decimal values are not supported at this time.

Volume configuration reference

While it is possible to declare volumes (/compose/compose-file/#volumes) on the file as part of the service declaration, this section allows you to create named volumes (without relying on `volumes_from`) that can be reused across multiple services, and are easily retrieved and inspected using the docker command line or API. See the docker volume (https://docs.docker.com/engine/reference/commandline/volume_create/) subcommand documentation for more information.

See Use volumes (https://docs.docker.com/engine/admin/volumes/volumes/) and Volume Plugins (https://docs.docker.com/engine/extend/plugins_volume/) for general information on volumes.

Here's an example of a two-service setup where a database's data directory is shared with another service as a volume so that it can be periodically backed up:

```
version: "3"

services:
  db:
    image: db
    volumes:
      - data-volume:/var/lib/db
  backup:
    image: backup-service
    volumes:
      - data-volume:/var/lib/backup/data

volumes:
  data-volume:
```

An entry under the top-level `volumes` key can be empty, in which case it uses the default driver configured by the Engine (in most cases, this is the `local` driver). Optionally, you can configure it with the following keys:

driver

Specify which volume driver should be used for this volume. Defaults to whatever driver the Docker Engine has been configured to use, which in most cases is `local`. If the driver is not available, the Engine returns an error when `docker-compose up` tries to create the volume.

```
driver: foobar
```

driver_opts

Specify a list of options as key-value pairs to pass to the driver for this volume. Those options are driver-dependent - consult the driver's documentation for more information. Optional.

```
driver_opts:
  foo: "bar"
  baz: 1
```

external

If set to `true`, specifies that this volume has been created outside of Compose. `docker-compose up` does not attempt to create it, and raises an error if it doesn't exist.

`external` cannot be used in conjunction with other volume configuration keys (`driver` , `driver_opts`).

In the example below, instead of attempting to create a volume called `[projectname]_data` , Compose looks for an existing volume simply called `data` and mount it into the `db` service's containers.

```
version: '2'

services:
  db:
    image: postgres
    volumes:
      - data:/var/lib/postgresql/data

volumes:
  data:
    external: true
```

> external.name was deprecated in version 3.4 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-34) use `name` instead.

You can also specify the name of the volume separately from the name used to refer to it within the Compose file:

```
volumes:
  data:
    external:
      name: actual-name-of-volume
```

> ✓ External volumes are always created with docker stack deploy
>
> External volumes that do not exist *are created* if you use docker stack deploy (/compose/compose-file/#deploy) to launch the app in swarm mode (https://docs.docker.com/engine/swarm/) (instead of docker compose up (https://docs.docker.com/compose/reference/up/)). In swarm mode, a volume is automatically created when it is defined by a service. As service tasks are scheduled on new nodes, swarmkit (https://github.com/docker/swarmkit/blob/master/README.md) creates the volume on the local node. To learn more, see moby/moby#29976 (https://github.com/moby/moby/issues/29976).

labels

Add metadata to containers using Docker labels (https://docs.docker.com/engine/userguide/labels-custom-metadata/). You can use either an array or a dictionary.

It's recommended that you use reverse-DNS notation to prevent your labels from conflicting with those used by other software.

```
labels:
  com.example.description: "Database volume"
  com.example.department: "IT/Ops"
  com.example.label-with-empty-value: ""

labels:
  - "com.example.description=Database volume"
  - "com.example.department=IT/Ops"
  - "com.example.label-with-empty-value"
```

name

> Added in version 3.4 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-34)

Set a custom name for this volume. The name field can be used to reference volumes that contain special characters. The name is used as is and will not be scoped with the stack name.

```
version: '3.4'
volumes:
  data:
    name: my-app-data
```

It can also be used in conjunction with the `external` property:

```
version: '3.4'
volumes:
  data:
    external: true
    name: my-app-data
```

Network configuration reference

The top-level `networks` key lets you specify networks to be created.

- For a full explanation of Compose's use of Docker networking features and all network driver options, see the Networking guide (https://docs.docker.com/compose/networking/).
- For Docker Labs (https://github.com/docker/labs/blob/master/README.md) tutorials on networking, start with Designing Scalable, Portable Docker Container Networks (https://github.com/docker/labs/blob/master/networking/README.md)

driver

Specify which driver should be used for this network.

The default driver depends on how the Docker Engine you're using is configured, but in most instances it is `bridge` on a single host and `overlay` on a Swarm.

The Docker Engine returns an error if the driver is not available.

```
    driver: overlay
```

BRIDGE

Docker defaults to using a `bridge` network on a single host. For examples of how to work with bridge networks, see the Docker Labs tutorial on Bridge networking (https://github.com/docker/labs/blob/master/networking/A2-bridge-networking.md).

OVERLAY

The `overlay` driver creates a named network across multiple nodes in a swarm (https://docs.docker.com/engine/swarm/).

- For a working example of how to build and use an `overlay` network with a service in swarm mode, see the Docker Labs tutorial on Overlay networking and service discovery (https://github.com/docker/labs/blob/master/networking/A3-overlay-networking.md).
- For an in-depth look at how it works under the hood, see the networking concepts lab on the Overlay Driver Network Architecture (https://github.com/docker/labs/blob/master/networking/concepts/06-overlay-networks.md).

HOST OR NONE

Use the host's networking stack, or no networking. Equivalent to `docker run --net=host` or `docker run --net=none`. Only used if you use `docker stack` commands. If you use the `docker-compose` command, use network_mode (/compose/compose-file/#network_mode) instead.

The syntax for using built-in networks like `host` and `none` is a little different. Define an external network with the name `host` or `none` (that Docker has already created automatically) and an alias that Compose can use (`hostnet` or `nonet` in these examples), then grant the service access to that network, using the alias.

```
services:
  web:
    ...
    networks:
      hostnet: {}

networks:
  hostnet:
    external: true
    name: host
```

```
services:
  web:
    ...
    networks:
      nonet: {}

networks:
  nonet:
    external: true
    name: none
```

driver_opts

Specify a list of options as key-value pairs to pass to the driver for this network. Those options are driver-dependent - consult the driver's documentation for more information. Optional.

```
driver_opts:
  foo: "bar"
  baz: 1
```

attachable

> Note: Only supported for v3.2 and higher.

Only used when the `driver` is set to `overlay`. If set to `true`, then standalone containers can attach to this network, in addition to services. If a standalone container attaches to an overlay network, it can communicate with services and standalone containers that are also attached to the overlay network from other Docker daemons.

```
networks:
  mynet1:
    driver: overlay
    attachable: true
```

enable_ipv6

Enable IPv6 networking on this network.

> Not supported in Compose File version 3
>
> `enable_ipv6` requires you to use a version 2 Compose file, as this directive is not yet supported in Swarm mode.

ipam

Specify custom IPAM config. This is an object with several properties, each of which is optional:

- `driver` : Custom IPAM driver, instead of the default.
- `config` : A list with zero or more config blocks, each containing any of the following keys:
  - `subnet` : Subnet in CIDR format that represents a network segment

A full example:

```
ipam:
  driver: default
  config:
    - subnet: 172.28.0.0/16
```

> Note: Additional IPAM configurations, such as `gateway`, are only honored for version 2 at the moment.

internal

By default, Docker also connects a bridge network to it to provide external connectivity. If you want to create an externally isolated overlay network, you can set this option to `true` .

labels

Add metadata to containers using Docker labels (https://docs.docker.com/engine/userguide/labels-custom-metadata/). You can use either an array or a dictionary.

It's recommended that you use reverse-DNS notation to prevent your labels from conflicting with those used by other software.

```
labels:
  com.example.description: "Financial transaction network"
  com.example.department: "Finance"
  com.example.label-with-empty-value: ""

labels:
  - "com.example.description=Financial transaction network"
  - "com.example.department=Finance"
  - "com.example.label-with-empty-value"
```

external

If set to `true` , specifies that this network has been created outside of Compose. `docker-compose up` does not attempt to create it, and raises an error if it doesn't exist.

`external` cannot be used in conjunction with other network configuration keys (`driver` , `driver_opts` , `ipam` , `internal`).

In the example below, `proxy` is the gateway to the outside world. Instead of attempting to create a network called `[projectname]_outside` , Compose looks for an existing network simply called `outside` and connect the `proxy` service's containers to it.

```
version: '2'

services:
  proxy:
    build: ./proxy
    networks:
      - outside
      - default
  app:
    build: ./app
    networks:
      - default

networks:
  outside:
    external: true
```

> external.name was deprecated in version 3.5 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-35) use `name` instead.

You can also specify the name of the network separately from the name used to refer to it within the Compose file:

```
networks:
  outside:
    external:
      name: actual-name-of-network
```

name

> Added in version 3.5 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-35)

Set a custom name for this network. The name field can be used to reference networks which contain special characters. The name is used as is and will not be scoped with the stack name.

```
version: '3.5'
networks:
  network1:
    name: my-app-net
```

It can also be used in conjunction with the `external` property:

```
version: '3.5'
networks:
  network1:
    external: true
    name: my-app-net
```

configs configuration reference

The top-level `configs` declaration defines or references configs (https://docs.docker.com/engine/swarm/configs/) that can be granted to the services in this stack. The source of the config is either `file` or `external` .

- `file` : The config is created with the contents of the file at the specified path.
- `external` : If set to true, specifies that this config has already been created. Docker does not attempt to create it, and if it does not exist, a `config not found` error occurs.
- `name` : The name of the config object in Docker. This field can be used to reference configs that contain special characters. The name is used as is and will not be scoped with the stack name. Introduced in version 3.5 file format.

In this example, `my_first_config` is created (as `<stack_name>_my_first_config)` when the stack is deployed, and `my_second_config` already exists in Docker.

```
configs:
  my_first_config:
    file: ./config_data
  my_second_config:
    external: true
```

Another variant for external configs is when the name of the config in Docker is different from the name that exists within the service. The following example modifies the previous one to use the external config called `redis_config`.

```
configs:
  my_first_config:
    file: ./config_data
  my_second_config:
    external:
      name: redis_config
```

You still need to grant access to the config (/compose/compose-file/#configs) to each service in the stack.

secrets configuration reference

The top-level `secrets` declaration defines or references secrets (https://docs.docker.com/engine/swarm/secrets/) that can be granted to the services in this stack. The source of the secret is either `file` or `external`.

- `file` : The secret is created with the contents of the file at the specified path.
- `external` : If set to true, specifies that this secret has already been created. Docker does not attempt to create it, and if it does not exist, a `secret not found` error occurs.
- `name` : The name of the secret object in Docker. This field can be used to reference secrets that contain special characters. The name is used as is and will not be scoped with the stack name. Introduced in version 3.5 file format.

In this example, `my_first_secret` is created (as `<stack_name>_my_first_secret)` when the stack is deployed, and `my_second_secret` already exists in Docker.

```
secrets:
  my_first_secret:
    file: ./secret_data
  my_second_secret:
    external: true
```

Another variant for external secrets is when the name of the secret in Docker is different from the name that exists within the service. The following example modifies the previous one to use the external secret called `redis_secret` .

```
secrets:
  my_first_secret:
    file: ./secret_data
  my_second_secret:
    external:
      name: redis_secret
```

You still need to grant access to the secrets (/compose/compose-file/#secrets) to each service in the stack.

Variable substitution

Your configuration options can contain environment variables. Compose uses the variable values from the shell environment in which `docker-compose` is run. For example, suppose the shell contains `POSTGRES_VERSION=9.3` and you supply this configuration:

```
db:
  image: "postgres:${POSTGRES_VERSION}"
```

When you run `docker-compose up` with this configuration, Compose looks for the `POSTGRES_VERSION` environment variable in the shell and substitutes its value in. For this example, Compose resolves the `image` to `postgres:9.3` before running the configuration.

If an environment variable is not set, Compose substitutes with an empty string. In the example above, if `POSTGRES_VERSION` is not set, the value for the `image` option is `postgres:` .

You can set default values for environment variables using a `.env` file (https://docs.docker.com/compose/env-file/), which Compose automatically looks for. Values set in the shell environment override those set in the `.env` file.

> Important: The `.env file` feature only works when you use the `docker-compose up` command and does not work with `docker stack deploy`.

Both `$VARIABLE` and `${VARIABLE}` syntax are supported. Additionally when using the 2.1 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-21), it is possible to provide inline default values using typical shell syntax:

- `${VARIABLE:-default}` evaluates to `default` if `VARIABLE` is unset or empty in the environment.
- `${VARIABLE-default}` evaluates to `default` only if `VARIABLE` is unset in the environment.

Similarly, the following syntax allows you to specify mandatory variables:

- `${VARIABLE:?err}` exits with an error message containing `err` if `VARIABLE` is unset or empty in the environment.
- `${VARIABLE?err}` exits with an error message containing `err` if `VARIABLE` is unset in the environment.

Other extended shell-style features, such as `${VARIABLE/foo/bar}`, are not supported.

You can use a `$$` (double-dollar sign) when your configuration needs a literal dollar sign. This also prevents Compose from interpolating a value, so a `$$` allows you to refer to environment variables that you don’t want processed by Compose.

```
web:
  build: .
  command: "$$VAR_NOT_INTERPOLATED_BY_COMPOSE"
```

If you forget and use a single dollar sign (`$`), Compose interprets the value as an environment variable and warns you:

The VAR_NOT_INTERPOLATED_BY_COMPOSE is not set. Substituting an empty string.

Extension fields

> Added in version 3.4 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-34).

It is possible to re-use configuration fragments using extension fields. Those special fields can be of any format as long as they are located at the root of your Compose file and their name start with the `¡-` character sequence.

> Note
>
> Starting with the 3.7 format (for the 3.x series) and 2.4 format (for the 2.x series), extension fields are also allowed at the root of service, volume, network, config and secret definitions.

```
version: ’2.1’
x-custom:
  items:
    - a
    - b
  options:
    max-size: ’12m’
  name: "custom"
```

The contents of those fields are ignored by Compose, but they can be inserted in your resource definitions using YAML anchors (http://www.yaml.org/spec/1.2/spec.html#id2765878). For example, if you want several of your services to use the same logging configuration:

```
logging:
  options:
    max-size: ’12m’
    max-file: ’5’
  driver: json-file
```

You may write your Compose file as follows:

```
version: '3.4'
x-logging:
  &default-logging
  options:
    max-size: '12m'
    max-file: '5'
  driver: json-file

services:
  web:
    image: myapp/web:latest
    logging: *default-logging
  db:
    image: mysql:latest
    logging: *default-logging
```

It is also possible to partially override values in extension fields using the YAML merge type (http://yaml.org/type/merge.html). For example:

```
version: '3.4'
x-volumes:
  &default-volume
  driver: foobar-storage

services:
  web:
    image: myapp/web:latest
    volumes: ["vol1", "vol2", "vol3"]
volumes:
  vol1: *default-volume
  vol2:
    << : *default-volume
    name: volume02
  vol3:
    << : *default-volume
    driver: default
    name: volume-local
```

Compose documentation

- User guide (https://docs.docker.com/compose/)
- Installing Compose (https://docs.docker.com/compose/install/)
- Compose file versions and upgrading (https://docs.docker.com/compose/compose-file/compose-versioning/)

- Get started with Docker (https://docs.docker.com/get-started/)
- Samples (https://docs.docker.com/samples/)
- Command line reference (https://docs.docker.com/compose/reference/)

fig (https://docs.docker.com/glossary/?term=fig), composition (https://docs.docker.com/glossary/?term=composition), compose (https://docs.docker.com/glossary/?term=compose), docker (https://docs.docker.com/glossary/?term=docker)

docker service scale

Estimated reading time: 3 minutes

Description

Scale one or multiple replicated services

API 1.24+ (https://docs.docker.com/engine/api/v1.24/) The client and daemon API must both be at least 1.24 (https://docs.docker.com/engine/api/v1.24/) to use this command. Use the `docker version` command on the client to check your client and daemon API versions.

Swarm This command works with the Swarm orchestrator.

Usage

```
docker service scale SERVICE=REPLICAS [SERVICE=REPLICAS...]
```

Options

Name, shorthand	Default	Description
`--detach , -d`		API 1.29+ (https://docs.docker.com/engine/api/v1.29/) Exit immediately instead of waiting for the service to converge

Parent command

Command	Description

Command	Description
docker service (https://docs.docker.com/engine/reference/commandline/service)	Manage services

Related commands

Command	Description
docker service create (https://docs.docker.com/engine/reference/commandline/service_create/)	Create a new service
docker service inspect (https://docs.docker.com/engine/reference/commandline/service_inspect/)	Display detailed information on one or more services
docker service logs (https://docs.docker.com/engine/reference/commandline/service_logs/)	Fetch the logs of a service or task
docker service ls (https://docs.docker.com/engine/reference/commandline/service_ls/)	List services
docker service ps (https://docs.docker.com/engine/reference/commandline/service_ps/)	List the tasks of one or more services
docker service rm (https://docs.docker.com/engine/reference/commandline/service_rm/)	Remove one or more services
docker service rollback (https://docs.docker.com/engine/reference/commandline/service_rollback/)	Revert changes to a service’s configuration

Command	Description
docker service scale (https://docs.docker.com/engine/reference/commandline/service_scale/)	Scale one or multiple replicated services
docker service update (https://docs.docker.com/engine/reference/commandline/service_update/)	Update a service

Extended description

The scale command enables you to scale one or more replicated services either up or down to the desired number of replicas. This command cannot be applied on services which are global mode. The command will return immediately, but the actual scaling of the service may take some time. To stop all replicas of a service while keeping the service active in the swarm you can set the scale to 0.

Examples

Scale a single service

The following command scales the “frontend” service to 50 tasks.

```
$ docker service scale frontend=50

frontend scaled to 50
```

The following command tries to scale a global service to 10 tasks and returns an error.

```
$ docker service create --mode global --name backend backend:latest

b4g08uwuairexjub6ome6usqh

$ docker service scale backend=10

backend: scale can only be used with replicated mode
```

Directly afterwards, run `docker service ls` , to see the actual number of replicas.

```
$ docker service ls --filter name=frontend

ID            NAME      MODE        REPLICAS  IMAGE
3pr5mlvu3fh9  frontend  replicated  15/50     nginx:alpine
```

You can also scale a service using the `docker service update` (https://docs.docker.com/engine/reference/commandline/service_update/) command. The following commands are equivalent:

```
$ docker service scale frontend=50
$ docker service update --replicas=50 frontend
```

Scale multiple services

The `docker service scale` command allows you to set the desired number of tasks for multiple services at once. The following example scales both the backend and frontend services:

```
$ docker service scale backend=3 frontend=5

backend scaled to 3
frontend scaled to 5

$ docker service ls

ID            NAME      MODE        REPLICAS  IMAGE
3pr5mlvu3fh9  frontend  replicated  5/5       nginx:alpine
74nzcxxjv6fq  backend   replicated  3/3       redis:3.0.6
```

docker stack services

Estimated reading time: 3 minutes

Description

List the services in the stack

API 1.25+ (https://docs.docker.com/engine/api/v1.25/) The client and daemon API must both be at least 1.25 (https://docs.docker.com/engine/api/v1.25/) to use this command. Use the `docker version` command on the client to check your client and daemon API versions.

Usage

```
docker stack services [OPTIONS] STACK
```

Options

Name, shorthand	Default	Description
`--filter , -f`		Filter output based on conditions provided
`--format`		Pretty-print services using a Go template
`--namespace`		Kubernetes Kubernetes namespace to use
`--quiet , -q`		Only display IDs
`--kubeconfig`		Kubernetes Kubernetes config file
`--orchestrator`		Orchestrator to use (swarm|kubernetes|all)

Parent command

Command	Description
docker stack (https://docs.docker.com/engine/reference/commandline/stack)	Manage Docker stacks

Related commands

Command	Description
docker stack deploy (https://docs.docker.com/engine/reference/commandline/stack_deploy/)	Deploy a new stack or update an existing stack
docker stack ls (https://docs.docker.com/engine/reference/commandline/stack_ls/)	List stacks
docker stack ps (https://docs.docker.com/engine/reference/commandline/stack_ps/)	List the tasks in the stack
docker stack rm (https://docs.docker.com/engine/reference/commandline/stack_rm/)	Remove one or more stacks
docker stack services (https://docs.docker.com/engine/reference/commandline/stack_services/)	List the services in the stack

Extended description

Lists the services that are running as part of the specified stack. This command has to be run targeting a manager node.

Examples

The following command shows all services in the `myapp` stack:

```
$ docker stack services myapp

ID              NAME            REPLICAS  IMAGE
                                                     COMMAND
7be5ei6sqeye    myapp_web       1/1       nginx@sha256:23f809e7fd5952e
7d5be065b4d3643fbbceccd349d537b62a123ef2201bc886f
dn7m7nhhfb9y    myapp_db        1/1       mysql@sha256:a9a5b559f8821fe
73d58c3606c812d1c044868d42c63817fa5125fd9d8b7b539
```

Filtering

The filtering flag (`-f` or `--filter`) format is a `key=value` pair. If there is more than one filter, then pass multiple flags (e.g. `--filter "foo=bar" --filter "bif=baz"`). Multiple filter flags are combined as an `OR` filter.

The following command shows both the `web` and `db` services:

```
$ docker stack services --filter name=myapp_web --filter name=myapp_
db myapp

ID              NAME            REPLICAS  IMAGE
                                                     COMMAND
7be5ei6sqeye    myapp_web       1/1       nginx@sha256:23f809e7fd5952e
7d5be065b4d3643fbbceccd349d537b62a123ef2201bc886f
dn7m7nhhfb9y    myapp_db        1/1       mysql@sha256:a9a5b559f8821fe
73d58c3606c812d1c044868d42c63817fa5125fd9d8b7b539
```

The currently supported filters are:

- id / ID (`--filter id=7be5ei6sqeye` , or `--filter ID=7be5ei6sqeye`)
  - Swarm: supported
  - Kubernetes: not supported
- label (`--filter label=key=value`)
  - Swarm: supported
  - Kubernetes: supported
- mode (`--filter mode=replicated` , or `--filter mode=global`)

  - Swarm: not supported
  - Kubernetes: supported
- name (`--filter name=myapp_web`)
  - Swarm: supported
  - Kubernetes: supported
- node (`--filter node=mynode`)
  - Swarm: not supported
  - Kubernetes: supported
- service (`--filter service=web`)
  - Swarm: not supported
  - Kubernetes: supported

Formatting

The formatting options (`--format`) pretty-prints services output using a Go template.

Valid placeholders for the Go template are listed below:

Placeholder	Description
`.ID`	Service ID
`.Name`	Service name
`.Mode`	Service mode (replicated, global)
`.Replicas`	Service replicas
`.Image`	Service image

When using the `--format` option, the `stack services` command will either output the data exactly as the template declares or, when using the `table` directive, includes column headers as well.

The following example uses a template without headers and outputs the `ID` , `Mode` , and `Replicas` entries separated by a colon for all services:

```
$ docker stack services --format "{{.ID}}: {{.Mode}} {{.Replicas}}"

0zmvwuiu3vue: replicated 10/10
fm6uf97exkul: global 5/5
```

Deploy services to a swarm

Estimated reading time: 37 minutes

Swarm services use a *declarative* model, which means that you define the desired state of the service, and rely upon Docker to maintain this state. The state includes information such as (but not limited to):

- the image name and tag the service containers should run
- how many containers participate in the service
- whether any ports are exposed to clients outside the swarm
- whether the service should start automatically when Docker starts
- the specific behavior that happens when the service is restarted (such as whether a rolling restart is used)
- characteristics of the nodes where the service can run (such as resource constraints and placement preferences)

For an overview of swarm mode, see Swarm mode key concepts (https://docs.docker.com/engine/swarm/key-concepts/). For an overview of how services work, see How services work (https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/).

Create a service

To create a single-replica service with no extra configuration, you only need to supply the image name. This command starts an Nginx service with a randomly-generated name and no published ports. This is a naive example, since you can't interact with the Nginx service.

```
$ docker service create nginx
```

The service is scheduled on an available node. To confirm that the service was created and started successfully, use the `docker service ls` command:

```
$ docker service ls

ID                  NAME                MODE                REPLICAS
            IMAGE
                                          PORTS
a3iixnklxuem        quizzical_lamarr    replicated          1/1
            docker.io/library/nginx@sha256:41ad9967ea448d7c2b203c699
b429abe1ed5af331cd92533900c6d77490e0268
```

Created services do not always run right away. A service can be in a pending state if its image is unavailable, if no node meets the requirements you configure for the service, or other reasons. See Pending services (https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/#pending-services) for more information.

To provide a name for your service, use the `--name` flag:

```
$ docker service create --name my_web nginx
```

Just like with standalone containers, you can specify a command that the service's containers should run, by adding it after the image name. This example starts a service called `helloworld` which uses an `alpine` image and runs the command `ping docker.com` :

```
$ docker service create --name helloworld alpine ping docker.com
```

You can also specify an image tag for the service to use. This example modifies the previous one to use the `alpine:3.6` tag:

```
$ docker service create --name helloworld alpine:3.6 ping docker.com
```

For more details about image tag resolution, see Specify the image version the service should use (/engine/swarm/services/#specify-the-image-version-the-service-should-use).

Create a service using an image on a private registry

If your image is available on a private registry which requires login, use the `--with-registry-auth` flag with `docker service create`, after logging in. If your image is stored on `registry.example.com`, which is a private registry, use a command like the following:

```
$ docker login registry.example.com

$ docker service  create \
  --with-registry-auth \
  --name my_service \
  registry.example.com/acme/my_image:latest
```

This passes the login token from your local client to the swarm nodes where the service is deployed, using the encrypted WAL logs. With this information, the nodes are able to log into the registry and pull the image.

Update a service

You can change almost everything about an existing service using the `docker service update` command. When you update a service, Docker stops its containers and restarts them with the new configuration.

Since Nginx is a web service, it works much better if you publish port 80 to clients outside the swarm. You can specify this when you create the service, using the `-p` or `--publish` flag. When updating an existing service, the flag is `--publish-add`. There is also a `--publish-rm` flag to remove a port that was previously published.

Assuming that the `my_web` service from the previous section still exists, use the following command to update it to publish port 80.

```
$ docker service update --publish-add 80 my_web
```

To verify that it worked, use `docker service ls`:

```
$ docker service ls

ID                  NAME                MODE                REPLICAS
            IMAGE
                                                PORTS
4nhxl7oxw5vz        my_web              replicated          1/1
            docker.io/library/nginx@sha256:41ad9967ea448d7c2b203c699
b429abe1ed5af331cd92533900c6d77490e0268    *:0->80/tcp
```

For more information on how publishing ports works, see publish ports (/engine/swarm/services/#publish-ports).

You can update almost every configuration detail about an existing service, including the image name and tag it runs. See Update a service's image after creation (/engine/swarm/services/#update-a-services-image-after-creation).

Remove a service

To remove a service, use the `docker service remove` command. You can remove a service by its ID or name, as shown in the output of the `docker service ls` command. The following command removes the `my_web` service.

```
$ docker service remove my_web
```

Service configuration details

The following sections provide details about service configuration. This topic does not cover every flag or scenario. In almost every instance where you can define a configuration at service creation, you can also update an existing service's configuration in a similar way.

See the command-line references for `docker service create` (https://docs.docker.com/engine/reference/commandline/service_create/) and `docker service update` (https://docs.docker.com/engine/reference/commandline/service_update/), or run one of those commands with the `--help` flag.

Configure the runtime environment

You can configure the following options for the runtime environment in the container:

- environment variables using the `--env` flag
- the working directory inside the container using the `--workdir` flag
- the username or UID using the `--user` flag

The following service's containers have an environment variable `$MYVAR` set to `myvalue`, run from the `/tmp/` directory, and run as the `my_user` user.

```
$ docker service create --name helloworld \
  --env MYVAR=myvalue \
  --workdir /tmp \
  --user my_user \
  alpine ping docker.com
```

Update the command an existing service runs

To update the command an existing service runs, you can use the `--args` flag. The following example updates an existing service called `helloworld` so that it runs the command `ping docker.com` instead of whatever command it was running before:

```
$ docker service update --args "ping docker.com" helloworld
```

Specify the image version a service should use

When you create a service without specifying any details about the version of the image to use, the service uses the version tagged with the `latest` tag. You can force the service to use a specific version of the image in a few different ways, depending on your desired outcome.

An image version can be expressed in several different ways:

- If you specify a tag, the manager (or the Docker client, if you use content trust (/engine/swarm/services/#image_resolution_with_trust)) resolves that tag to a digest. When the request to create a container task is received on a worker node, the worker node only sees the digest, not the tag.

```
$ docker service create --name="myservice" ubuntu:16.04
```

Some tags represent discrete releases, such as `ubuntu:16.04`. Tags like this almost always resolve to a stable digest over time. It is recommended that you use this kind of tag when possible.

Other types of tags, such as `latest` or `nightly`, may resolve to a new digest often, depending on how often an image's author updates the tag. It is not recommended to run services using a tag which is updated frequently, to prevent different service replica tasks from using different image versions.

- If you don't specify a version at all, by convention the image's `latest` tag is resolved to a digest. Workers use the image at this digest when creating the service task.

  Thus, the following two commands are equivalent:

  ```
  $ docker service create --name="myservice" ubuntu

  $ docker service create --name="myservice" ubuntu:latest
  ```

- If you specify a digest directly, that exact version of the image is always used when creating service tasks.

  ```
  $ docker service create \
      --name="myservice" \
      ubuntu:16.04@sha256:35bc48a1ca97c3971611dc4662d08d131869daa
  692acb281c7e9e052924e38b1
  ```

When you create a service, the image's tag is resolved to the specific digest the tag points to at the time of service creation. Worker nodes for that service use that specific digest forever unless the service is explicitly updated. This feature is particularly important if you do use often-changing tags such as `latest`, because it ensures that all service tasks use the same version of the image.

> Note: If content trust (https://docs.docker.com/engine/security/trust/content_trust/) is enabled, the client actually resolves the image's tag to a digest before contacting the swarm manager, to verify that the image is signed. Thus, if you use content trust, the swarm manager receives the request pre-resolved. In this case, if the client cannot resolve the image to a digest, the request fails.

If the manager can't resolve the tag to a digest, each worker node is responsible for resolving the tag to a digest, and different nodes may use different versions of the image. If this happens, a warning like the following is logged, substituting the placeholders for real information.

```
unable to pin image <IMAGE-NAME> to digest: <REASON>
```

To see an image's current digest, issue the command `docker inspect <IMAGE>:<TAG>` and look for the `RepoDigests` line. The following is the current digest for `ubuntu:latest` at the time this content was written. The output is truncated for clarity.

```
$ docker inspect ubuntu:latest
```

```
"RepoDigests": [
    "ubuntu@sha256:35bc48a1ca97c3971611dc4662d08d131869daa692acb281c
7e9e052924e38b1"
],
```

After you create a service, its image is never updated unless you explicitly run `docker service update` with the `--image` flag as described below. Other update operations such as scaling the service, adding or removing networks or volumes, renaming the service, or any other type of update operation do not update the service's image.

Update a service's image after creation

Each tag represents a digest, similar to a Git hash. Some tags, such as `latest`, are updated often to point to a new digest. Others, such as `ubuntu:16.04`, represent a released software version and are not expected to update to point to a new digest often if at all. In Docker 1.13 and higher, when you create a service, it is constrained to create tasks using a specific digest of an image until you update the service using `service update` with the `--image` flag. If you use an older version of Docker Engine, you must remove and re-create the service to update its image.

When you run `service update` with the `--image` flag, the swarm manager queries Docker Hub or your private Docker registry for the digest the tag currently points to and updates the service tasks to use that digest.

> Note: If you use content trust (/engine/swarm/services/#image_resolution_with_trust), the Docker client resolves image and the swarm manager receives the image and digest, rather than a tag.

Usually, the manager can resolve the tag to a new digest and the service updates, redeploying each task to use the new image. If the manager can't resolve the tag or some other problem occurs, the next two sections outline what to expect.

IF THE MANAGER RESOLVES THE TAG

If the swarm manager can resolve the image tag to a digest, it instructs the worker nodes to redeploy the tasks and use the image at that digest.

- If a worker has cached the image at that digest, it uses it.
- If not, it attempts to pull the image from Docker Hub or the private registry.
    - If it succeeds, the task is deployed using the new image.
    - If the worker fails to pull the image, the service fails to deploy on that worker node. Docker tries again to deploy the task, possibly on a different worker node.

IF THE MANAGER CANNOT RESOLVE THE TAG

If the swarm manager cannot resolve the image to a digest, all is not lost:

- The manager instructs the worker nodes to redeploy the tasks using the image at that tag.

- If the worker has a locally cached image that resolves to that tag, it uses that image.
- If the worker does not have a locally cached image that resolves to the tag, the worker tries to connect to Docker Hub or the private registry to pull the image at that tag.
    - If this succeeds, the worker uses that image.
    - If this fails, the task fails to deploy and the manager tries again to deploy the task, possibly on a different worker node.

Publish ports

When you create a swarm service, you can publish that service's ports to hosts outside the swarm in two ways:

- You can rely on the routing mesh (/engine/swarm/services/#publish-a services-ports-using-the-routing-mesh). When you publish a service port, the swarm makes the service accessible at the target port on every node, regardless of whether there is a task for the service running on that node or not. This is less complex and is the right choice for many types of services.
- You can publish a service task's port directly on the swarm node (/engine/swarm/services/#publish-a-services-ports-directly-on-the-swarm-node) where that service is running. This feature is available in Docker 1.13 and higher. This bypasses the routing mesh and provides the maximum flexibility, including the ability for you to develop your own routing framework. However, you are responsible for keeping track of where each task is running and routing requests to the tasks, and load-balancing across the nodes.

Keep reading for more information and use cases for each of these methods.

PUBLISH A SERVICE'S PORTS USING THE ROUTING MESH

To publish a service's ports externally to the swarm, use the `--publish <PUBLISHED-PORT>:<SERVICE-PORT>` flag. The swarm makes the service accessible at the published port on every swarm node. If an external host connects to that port on any swarm node, the routing mesh routes it to a task. The external host does not need to know the IP addresses or internally-used ports of the service tasks to interact with the service. When a user or process connects to a service, any worker node running a service task may respond. For more details about swarm service networking, see Manage swarm service networks (https://docs.docker.com/engine/swarm/networking/).

Example: Run a three-task Nginx service on 10-node swarm

Imagine that you have a 10-node swarm, and you deploy an Nginx service running three tasks on a 10-node swarm:

```
$ docker service create --name my_web \
                        --replicas 3 \
                        --publish published=8080,target=80 \
                        nginx
```

Three tasks run on up to three nodes. You don't need to know which nodes are running the tasks; connecting to port 8080 on any of the 10 nodes connects you to one of the three `nginx` tasks. You can test this using `curl` . The following example assumes that `localhost` is one of the swarm nodes. If this is not the case, or `localhost` does not resolve to an IP address on your host, substitute the host's IP address or resolvable host name.

The HTML output is truncated:

```
$ curl localhost:8080

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...truncated...
</html>
```

Subsequent connections may be routed to the same swarm node or a different one.

PUBLISH A SERVICE'S PORTS DIRECTLY ON THE SWARM NODE

Using the routing mesh may not be the right choice for your application if you need to make routing decisions based on application state or you need total control of the process for routing requests to your service's tasks. To publish a service's port directly on the node where it is running, use the `mode=host` option to the `--publish` flag.

> ✓ Note: If you publish a service's ports directly on the swarm node using **`mode=host`** and also set **`published=<PORT>`** this creates an implicit limitation that you can only run one task for that service on a given swarm node. You can work around this by specifying **`published`** without a port definition, which causes Docker to assign a random port for each task.
>
> In addition, if you use `mode=host` and you do not use the `--mode=global` flag on `docker service create`, it is difficult to know which nodes are running the service to route work to them.

Example: Run an `nginx` web server service on every swarm node

nginx (https://hub.docker.com/_/nginx/) is an open source reverse proxy, load balancer, HTTP cache, and a web server. If you run nginx as a service using the routing mesh, connecting to the nginx port on any swarm node shows you the web page for (effectively) a random swarm node running the service.

The following example runs nginx as a service on each node in your swarm and exposes nginx port locally on each swarm node.

```
$ docker service create \
  --mode global \
  --publish mode=host,target=80,published=8080 \
  --name=nginx \
  nginx:latest
```

You can reach the nginx server on port 8080 of every swarm node. If you add a node to the swarm, a nginx task is started on it. You cannot start another service or container on any swarm node which binds to port 8080.

> Note: This is a naive example. Creating an application-layer routing framework for a multi-tiered service is complex and out of scope for this topic.

Connect the service to an overlay network

You can use overlay networks to connect one or more services within the swarm.

First, create overlay network on a manager node using the `docker network create` command with the `--driver overlay` flag.

```
$ docker network create --driver overlay my-network
```

After you create an overlay network in swarm mode, all manager nodes have access to the network.

You can create a new service and pass the `--network` flag to attach the service to the overlay network:

```
$ docker service create \
  --replicas 3 \
  --network my-network \
  --name my-web \
  nginx
```

The swarm extends `my-network` to each node running the service.

You can also connect an existing service to an overlay network using the `--network-add` flag.

```
$ docker service update --network-add my-network my-web
```

To disconnect a running service from a network, use the `--network-rm` flag.

```
$ docker service update --network-rm my-network my-web
```

For more information on overlay networking and service discovery, refer to Attach services to an overlay network (https://docs.docker.com/engine/swarm/networking/) and Docker swarm mode overlay network security model (https://docs.docker.com/engine/userguide/networking/overlay-security-model/).

Grant a service access to secrets

To create a service with access to Docker-managed secrets, use the `--secret` flag. For more information, see Manage sensitive strings (secrets) for Docker services (https://docs.docker.com/engine/swarm/secrets/)

Customize a service's isolation mode

Docker 17.12 CE and higher allow you to specify a swarm service's isolation mode. This setting applies to Windows hosts only and is ignored for Linux hosts. The isolation mode can be one of the following:

- `default` : Use the default isolation mode configured for the Docker host, as configured by the `-exec-opt` flag or `exec-opts` array in `daemon.json` . If the daemon does not specify an isolation technology, `process` is the default for Windows Server, and `hyperv` is the default (and only) choice for Windows 10.
- `process` : Run the service tasks as a separate process on the host.

  > Note: `process` isolation mode is only supported on Windows Server. Windows 10 only supports `hyperv` isolation mode.

- `hyperv` : Run the service tasks as isolated `hyperv` tasks. This increases overhead but provides more isolation.

You can specify the isolation mode when creating or updating a new service using the `--isolation` flag.

Control service placement

Swarm services provide a few different ways for you to control scale and placement of services on different nodes.

- You can specify whether the service needs to run a specific number of replicas or should run globally on every worker node. See Replicated or global services (/engine/swarm/services/#replicated-or-global-services).
- You can configure the service's CPU or memory requirements (/engine/swarm/services/#reserve-memory-or-cpus-for-a-service), and the service only runs on nodes which can meet those requirements.
- Placement constraints (/engine/swarm/services/#placement-constraints) let you configure the service to run only on nodes with specific (arbitrary) metadata set, and cause the deployment to fail if appropriate nodes do not exist. For instance, you can specify that your service should only run on nodes where an arbitrary label `pci_compliant` is set to `true` .
- Placement preferences (/engine/swarm/services/#placement-preferences) let you apply an arbitrary label with a range of values to each node, and spread your service's tasks across those nodes using an algorithm. Currently, the only supported algorithm is `spread` , which tries to place them evenly. For instance, if you label each node with a label `rack` which has a value from 1-10, then specify a placement preference keyed on `rack` , then service tasks are placed as evenly as possible across all nodes with the label `rack` , after taking other placement constraints, placement preferences, and other node-specific limitations into account.

  Unlike constraints, placement preferences are best-effort, and a service does not fail to deploy if no nodes can satisfy the preference. If you specify a placement preference for a service, nodes that match that preference are ranked higher when the swarm managers decide which nodes should run the service tasks. Other factors, such as high availability of the service, also factor into which nodes are scheduled to run service tasks. For example, if you have N nodes with the rack label (and then some others), and your service is configured to run N+1 replicas, the +1 is scheduled on a node that doesn't already have the service on it if there is one, regardless of whether that node has the `rack` label or not.

REPLICATED OR GLOBAL SERVICES

Swarm mode has two types of services: replicated and global. For replicated services, you specify the number of replica tasks for the swarm manager to schedule onto available nodes. For global services, the scheduler places one task on each available node that meets the service's placement constraints (/engine/swarm/services/#placement-constraints) and resource requirements (/engine/swarm/services/#reserve-cpu-or-memory-for-a-service).

You control the type of service using the `--mode` flag. If you don't specify a mode, the service defaults to `replicated`. For replicated services, you specify the number of replica tasks you want to start using the `--replicas` flag. For example, to start a replicated nginx service with 3 replica tasks:

```
$ docker service create \
  --name my_web \
  --replicas 3 \
  nginx
```

To start a global service on each available node, pass `--mode global` to `docker service create`. Every time a new node becomes available, the scheduler places a task for the global service on the new node. For example to start a service that runs alpine on every node in the swarm:

```
$ docker service create \
  --name myservice \
  --mode global \
  alpine top
```

Service constraints let you set criteria for a node to meet before the scheduler deploys a service to the node. You can apply constraints to the service based upon node attributes and metadata or engine metadata. For more information on constraints, refer to the `docker service create` CLI reference (https://docs.docker.com/engine/reference/commandline/service_create/).

RESERVE MEMORY OR CPUS FOR A SERVICE

To reserve a given amount of memory or number of CPUs for a service, use the `--reserve-memory` or `--reserve-cpu` flags. If no available nodes can satisfy the requirement (for instance, if you request 4 CPUs and no node in the swarm has 4 CPUs), the service remains in a pending state until an appropriate node is available to run its tasks.

Out Of Memory Exceptions (OOME)

If your service attempts to use more memory than the swarm node has available, you may experience an Out Of Memory Exception (OOME) and a container, or the Docker daemon, might be killed by the kernel OOM killer. To prevent this from happening, ensure that your application runs on hosts with adequate memory and see Understand the risks of running out of memory (https://docs.docker.com/engine/admin/resource_constraints/#understand-the-risks-of-running-out-of-memory).

Swarm services allow you to use resource constraints, placement preferences, and labels to ensure that your service is deployed to the appropriate swarm nodes.

PLACEMENT CONSTRAINTS

Use placement constraints to control the nodes a service can be assigned to. In the following example, the service only runs on nodes with the label (https://docs.docker.com/engine/swarm/manage-nodes/#add-or-remove-label-metadata) `region` set to `east` . If no appropriately-labelled nodes are available, tasks will wait in `Pending` until they become available. The `--constraint` flag uses an equality operator (`==` or `!=`). For replicated services, it is possible that all services run on the same node, or each node only runs one replica, or that some nodes don't run any replicas. For global services, the service runs on every node that meets the placement constraint and any resource requirements (/engine/swarm/services/#reserve-cpu-or-memory-for-a-service).

```
$ docker service create \
  --name my-nginx \
  --replicas 5 \
  --constraint node.labels.region==east \
  nginx
```

You can also use the `constraint` service-level key in a `docker-compose.yml` file.

If you specify multiple placement constraints, the service only deploys onto nodes where they are all met. The following example limits the service to run on all nodes where `region` is set to `east` and `type` is not set to `devel` :

```
$ docker service create \
  --name my-nginx \
  --mode global \
  --constraint node.labels.region==east \
  --constraint node.labels.type!=devel \
  nginx
```

You can also use placement constraints in conjunction with placement preferences and CPU/memory constraints. Be careful not to use settings that are not possible to fulfill.

For more information on constraints, refer to the `docker service create` CLI reference (https://docs.docker.com/engine/reference/commandline/service_create/).

PLACEMENT PREFERENCES

While placement constraints (/engine/swarm/services/#placement-constraints) limit the nodes a service can run on, *placement preferences* try to place services on appropriate nodes in an algorithmic way (currently, only spread evenly). For instance, if you assign each node a `rack` label, you can set a placement preference to spread the service evenly across nodes with the `rack` label, by value. This way, if you lose a rack, the service is still running on nodes on other racks.

Placement preferences are not strictly enforced. If no node has the label you specify in your preference, the service is deployed as though the preference were not set.

> Placement preferences are ignored for global services.

The following example sets a preference to spread the deployment across nodes based on the value of the `datacenter` label. If some nodes have `datacenter=us-east` and others have `datacenter=us-west` , the service is deployed as evenly as possible across the two sets of nodes.

```
$ docker service create \
  --replicas 9 \
  --name redis_2 \
  --placement-pref 'spread=node.labels.datacenter' \
  redis:3.0.6
```

✓ Missing or null labels

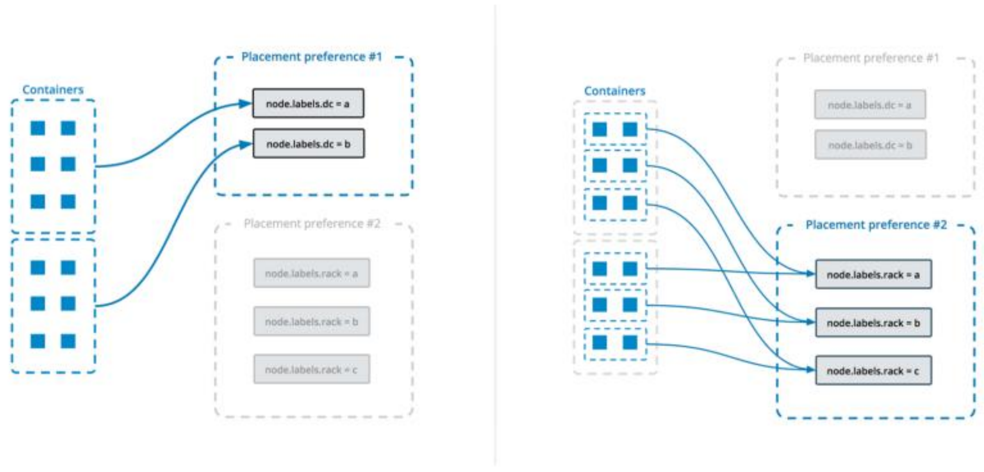
Nodes which are missing the label used to spread still receive task assignments. As a group, these nodes receive tasks in equal proportion to any of the other groups identified by a specific label value. In a sense, a missing label is the same as having the label with a null value attached to it. If the service should only run on nodes with the label being used for the spread preference, the preference should be combined with a constraint.

You can specify multiple placement preferences, and they are processed in the order they are encountered. The following example sets up a service with multiple placement preferences. Tasks are spread first over the various datacenters, and then over racks (as indicated by the respective labels):

```
$ docker service create \
  --replicas 9 \
  --name redis_2 \
  --placement-pref 'spread=node.labels.datacenter' \
  --placement-pref 'spread=node.labels.rack' \
  redis:3.0.6
```

You can also use placement preferences in conjunction with placement constraints or CPU/memory constraints. Be careful not to use settings that are not possible to fulfill.

This diagram illustrates how placement preferences work:



When updating a service with `docker service update`, `--placement-pref-add` appends a new placement preference after all existing placement preferences. `--placement-pref-rm` removes an existing placement preference that matches the argument.

Configure a service's update behavior

When you create a service, you can specify a rolling update behavior for how the swarm should apply changes to the service when you run `docker service update`. You can also specify these flags as part of the update, as arguments to `docker service update`.

The `--update-delay` flag configures the time delay between updates to a service task or sets of tasks. You can describe the time `T` as a combination of the number of seconds `Ts`, minutes `Tm`, or hours `Th`. So `10m30s` indicates a 10 minute 30 second delay.

By default the scheduler updates 1 task at a time. You can pass the `--update-parallelism` flag to configure the maximum number of service tasks that the scheduler updates simultaneously.

When an update to an individual task returns a state of `RUNNING`, the scheduler continues the update by continuing to another task until all tasks are updated. If, at any time during an update a task returns `FAILED`, the scheduler pauses the update. You can control the behavior using the `--update-failure-action` flag for `docker service create` or `docker service update`.

In the example service below, the scheduler applies updates to a maximum of 2 replicas at a time. When an updated task returns either `RUNNING` or `FAILED` , the scheduler waits 10 seconds before stopping the next task to update:

```
$ docker service create \
  --replicas 10 \
  --name my_web \
  --update-delay 10s \
  --update-parallelism 2 \
  --update-failure-action continue \
  alpine
```

The `--update-max-failure-ratio` flag controls what fraction of tasks can fail during an update before the update as a whole is considered to have failed. For example, with `--update-max-failure-ratio 0.1 --update-failure-action pause` , after 10% of the tasks being updated fail, the update is paused.

An individual task update is considered to have failed if the task doesn't start up, or if it stops running within the monitoring period specified with the `--update-monitor` flag. The default value for `--update-monitor` is 30 seconds, which means that a task failing in the first 30 seconds after its started counts towards the service update failure threshold, and a failure after that is not counted.

Roll back to the previous version of a service

In case the updated version of a service doesn't function as expected, it's possible to manually roll back to the previous version of the service using `docker service update` 's `--rollback` flag. This reverts the service to the configuration that was in place before the most recent `docker service update` command.

Other options can be combined with `--rollback` ; for example, `--update-delay 0s` to execute the rollback without a delay between tasks:

```
$ docker service update \
  --rollback \
  --update-delay 0s
  my_web
```

In Docker 17.04 and higher, you can configure a service to roll back automatically if a service update fails to deploy. See Automatically roll back if an update fails (/engine/swarm/services/#automatically-roll-back-if-an-update-fails).

Related to the new automatic rollback feature, in Docker 17.04 and higher, manual rollback is handled at the server side, rather than the client, if the daemon is running Docker 17.04 or higher. This allows manually-initiated rollbacks to respect the new rollback parameters. The client is version-aware, so it still uses the old method against an older daemon.

Finally, in Docker 17.04 and higher, `--rollback` cannot be used in conjunction with other flags to `docker service update` .

Automatically roll back if an update fails

You can configure a service in such a way that if an update to the service causes redeployment to fail, the service can automatically roll back to the previous configuration. This helps protect service availability. You can set one or more of the following flags at service creation or update. If you do not set a value, the default is used.

Flag	Default	Description
`--rollback-delay`	`0s`	Amount of time to wait after rolling back a task before rolling back the next one. A value of `0` means to roll back the second task immediately after the first rolled-back task deploys.
`--rollback-failure-action`	`pause`	When a task fails to roll back, whether to `pause` or `continue` trying to roll back other tasks.

Flag	Default	Description
`--rollback-max-failure-ratio`	`0`	The failure rate to tolerate during a rollback, specified as a floating-point number between 0 and 1. For instance, given 5 tasks, a failure ratio of `.2` would tolerate one task failing to roll back. A value of `0` means no failure are tolerated, while a value of `1` means any number of failure are tolerated.
`--rollback-monitor`	`5s`	Duration after each task rollback to monitor for failure. If a task stops before this time period has elapsed, the rollback is considered to have failed.
`--rollback-parallelism`	`1`	The maximum number of tasks to roll back in parallel. By default, one task is rolled back at a time. A value of `0` causes all tasks to be rolled back in parallel.

The following example configures a `redis` service to roll back automatically if a `docker service update` fails to deploy. Two tasks can be rolled back in parallel. Tasks are monitored for 20 seconds after rollback to be sure they do not exit, and a maximum failure ratio of 20% is tolerated. Default values are used for `--rollback-delay` and `--rollback-failure-action`.

```
$ docker service create --name=my_redis \
                        --replicas=5 \
                        --rollback-parallelism=2 \
                        --rollback-monitor=20s \
                        --rollback-max-failure-ratio=.2 \
                        redis:latest
```

Give a service access to volumes or bind mounts

For best performance and portability, you should avoid writing important data directly into a container's writable layer, instead using data volumes or bind mounts. This principle also applies to services.

You can create two types of mounts for services in a swarm, `volume` mounts or `bind` mounts. Regardless of which type of mount you use, configure it using the `--mount` flag when you create a service, or the `--mount-add` or `--mount-rm` flag when updating an existing service. The default is a data volume if you don't specify a type.

DATA VOLUMES

Data volumes are storage that exist independently of a container. The lifecycle of data volumes under swarm services is similar to that under containers. Volumes outlive tasks and services, so their removal must be managed separately. Volumes can be created before deploying a service, or if they don't exist on a particular host when a task is scheduled there, they are created automatically according to the volume specification on the service.

To use existing data volumes with a service use the `--mount` flag:

```
$ docker service create \
  --mount src=<VOLUME-NAME>,dst=<CONTAINER-PATH> \
  --name myservice \
  <IMAGE>
```

If a volume with the same `<VOLUME-NAME>` does not exist when a task is scheduled to a particular host, then one is created. The default volume driver is `local`. To use a different volume driver with this create-on-demand pattern, specify the driver and its options with the `--mount` flag:

```
$ docker service create \
  --mount type=volume,src=<VOLUME-NAME>,dst=<CONTAINER-PATH>,volume-
driver=<DRIVER>,volume-opt=<KEY0>=<VALUE0>,volume-opt=<KEY1>=<VALUE1
>
  --name myservice \
  <IMAGE>
```

For more information on how to create data volumes and the use of volume drivers, see Use volumes (https://docs.docker.com/storage/volumes/).

BIND MOUNTS

Bind mounts are file system paths from the host where the scheduler deploys the container for the task. Docker mounts the path into the container. The file system path must exist before the swarm initializes the container for the task.

The following examples show bind mount syntax:

- To mount a read-write bind:

```
$ docker service create \
  --mount type=bind,src=<HOST-PATH>,dst=<CONTAINER-PATH> \
  --name myservice \
  <IMAGE>
```

- To mount a read-only bind:

```
$ docker service create \
  --mount type=bind,src=<HOST-PATH>,dst=<CONTAINER-PATH>,readon
ly \
  --name myservice \
  <IMAGE>
```

> Important: Bind mounts can be useful but they can also cause problems. In most cases, it is recommended that you architect your application such that mounting paths from the host is unnecessary. The main risks include the following:
>
> - If you bind mount a host path into your service's containers, the path must exist on every swarm node. The Docker swarm mode scheduler can schedule containers on any machine that meets resource availability requirements and satisfies all constraints and placement preferences you specify.
> - The Docker swarm mode scheduler may reschedule your running service containers at any time if they become unhealthy or unreachable.
> - Host bind mounts are non-portable. When you use bind mounts, there is no guarantee that your application runs the same way in development as it does in production.

Create services using templates

You can use templates for some flags of `service create`, using the syntax provided by the Go's text/template (http://golang.org/pkg/text/template/) package.

The following flags are supported:

- `--hostname`
- `--mount`
- `--env`

Valid placeholders for the Go template are:

Placeholder	Description
`.Service.ID`	Service ID
`.Service.Name`	Service name
`.Service.Labels`	Service labels
`.Node.ID`	Node ID

Placeholder	Description
`.Node.Hostname`	Node hostname
`.Task.Name`	Task name
`.Task.Slot`	Task slot

TEMPLATE EXAMPLE

This example sets the template of the created containers based on the service's name and the ID of the node where the container is running:

```
$ docker service create --name hosttempl \
                        --hostname="{{.Node.ID}}-{{.Service.Name}}"\
                         busybox top
```

To see the result of using the template, use the `docker service ps` and `docker inspect` commands.

```
$ docker service ps va8ew30grofhjoychbr6iot8c

ID            NAME         IMAGE
                                                    NODE          DESIRED
 STATE  CURRENT STATE              ERROR  PORTS
wo41w8hg8qan  hosttempl.1  busybox:latest@sha256:29f5d56d12684887bdf
a50dcd29fc31eea4aaf4ad3bec43daf19026a7ce69912  2e7a8a9c4da2  Running
        Running about a minute ago
```

```
$ docker inspect --format="{{.Config.Hostname}}" hosttempl.1.wo41w8h
g8qanxwjwsg4kxpprj
```

Learn More

- Swarm administration guide (https://docs.docker.com/engine/swarm/admin_guide/)
- Docker Engine command line reference (https://docs.docker.com/engine/reference/commandline/docker/)

- Swarm mode tutorial (https://docs.docker.com/engine/swarm/swarm-tutorial/)

guide (https://docs.docker.com/glossary/?term=guide), swarm mode (https://docs.docker.com/glossary/?term=swarm mode), swarm (https://docs.docker.com/glossary/?term=swarm), service (https://docs.docker.com/glossary/?term=service)

docker node update

Estimated reading time: 2 minutes

Description

Update a node

API 1.24+ (https://docs.docker.com/engine/api/v1.24/) The client and daemon API must both be at least 1.24 (https://docs.docker.com/engine/api/v1.24/) to use this command. Use the `docker version` command on the client to check your client and daemon API versions.

Swarm This command works with the Swarm orchestrator.

Usage

```
docker node update [OPTIONS] NODE
```

Options

Name, shorthand	Default	Description
`--availability`		Availability of the node ("active"|"pause"|"drain")
`--label-add`		Add or update a node label (key=value)
`--label-rm`		Remove a node label if exists
`--role`		Role of the node ("worker"|"manager")

Parent command

Command	Description
docker node (https://docs.docker.com/engine/reference/commandline/node)	Manage Swarm nodes

Related commands

Command	Description
docker node demote (https://docs.docker.com/engine/reference/commandline/node_demote/)	Demote one or more nodes from manager in the swarm
docker node inspect (https://docs.docker.com/engine/reference/commandline/node_inspect/)	Display detailed information on one or more nodes
docker node ls (https://docs.docker.com/engine/reference/commandline/node_ls/)	List nodes in the swarm
docker node promote (https://docs.docker.com/engine/reference/commandline/node_promote/)	Promote one or more nodes to manager in the swarm

Command	Description
docker node ps (https://docs.docker.com/engine/reference/commandline/node_ps/)	List tasks running on one or more nodes, defaults to current node
docker node rm (https://docs.docker.com/engine/reference/commandline/node_rm/)	Remove one or more nodes from the swarm
docker node update (https://docs.docker.com/engine/reference/commandline/node_update/)	Update a node

Extended description

Update metadata about a node, such as its availability, labels, or roles.

Examples

Add label metadata to a node

Add metadata to a swarm node using node labels. You can specify a node label as a key with an empty value:

```
$ docker node update --label-add foo worker1
```

To add multiple labels to a node, pass the `--label-add` flag for each label:

```
$ docker node update --label-add foo --label-add bar worker1
```

When you create a service (https://docs.docker.com/engine/reference/commandline/service_create/), you can use node labels as a constraint. A constraint limits the nodes where the scheduler deploys tasks for a service.

For example, to add a `type` label to identify nodes where the scheduler should deploy message queue service tasks:

```
$ docker node update --label-add type=queue worker1
```

The labels you set for nodes using `docker node update` apply only to the node entity within the swarm. Do not confuse them with the docker daemon labels for dockerd (https://docs.docker.com/engine/userguide/labels-custom-metadata/#daemon-labels).

For more information about labels, refer to apply custom metadata (https://docs.docker.com/engine/userguide/labels-custom-metadata/).

Container networking

Estimated reading time: 3 minutes

The type of network a container uses, whether it is a bridge (https://docs.docker.com/config/containers/bridges/), an overlay (https://docs.docker.com/config/containers/overlay/), a macvlan network (https://docs.docker.com/config/containers/macvlan/), or a custom network plugin, is transparent from within the container. From the container's point of view, it has a network interface with an IP address, a gateway, a routing table, DNS services, and other networking details (assuming the container is not using the `none` network driver). This topic is about networking concerns from the point of view of the container.

Published ports

By default, when you create a container, it does not publish any of its ports to the outside world. To make a port available to services outside of Docker, or to Docker containers which are not connected to the container's network, use the `--publish` or `-p` flag. This creates a firewall rule which maps a container port to a port on the Docker host. Here are some examples.

Flag value	Description
`-p 8080:80`	Map TCP port 80 in the container to port 8080 on the Docker host.
`-p 192.168.1.100:8080:80`	Map TCP port 80 in the container to port 8080 on the Docker host for connections to host IP 192.168.1.100.
`-p 8080:80/udp`	Map UDP port 80 in the container to port 8080 on the Docker host.

Flag value	Description
`-p 8080:80/tcp -p 8080:80/udp`	Map TCP port 80 in the container to TCP port 8080 on the Docker host, and map UDP port 80 in the container to UDP port 8080 on the Docker host.

IP address and hostname

By default, the container is assigned an IP address for every Docker network it connects to. The IP address is assigned from the pool assigned to the network, so the Docker daemon effectively acts as a DHCP server for each container. Each network also has a default subnet mask and gateway.

When the container starts, it can only be connected to a single network, using `--network` . However, you can connect a running container to multiple networks using `docker network connect` . When you start a container using the `--network` flag, you can specify the IP address assigned to the container on that network using the `--ip` or `--ip6` flags.

When you connect an existing container to a different network using `docker network connect` , you can use the `--ip` or `--ip6` flags on that command to specify the container's IP address on the additional network.

In the same way, a container's hostname defaults to be the container's ID in Docker. You can override the hostname using `--hostname` . When connecting to an existing network using `docker network connect` , you can use the `--alias` flag to specify an additional network alias for the container on that network.

DNS services

By default, a container inherits the DNS settings of the Docker daemon, including the `/etc/hosts` and `/etc/resolv.conf` .You can override these settings on a per-container basis.

Flag	Description

Flag	Description
`--dns`	The IP address of a DNS server. To specify multiple DNS servers, use multiple `--dns` flags. If the container cannot reach any of the IP addresses you specify, Google's public DNS server `8.8.8.8` is added, so that your container can resolve internet domains.
`--dns-search`	A DNS search domain to search non-fully-qualified hostnames. To specify multiple DNS search prefixes, use multiple `--dns-search` flags.
`--dns-opt`	A key-value pair representing a DNS option and its value. See your operating system's documentation for `resolv.conf` for valid options.
`--hostname`	The hostname a container uses for itself. Defaults to the container's ID if not specified.

Proxy server

If your container needs to use a proxy server, see Use a proxy server (https://docs.docker.com/network/proxy/).

networking (https://docs.docker.com/glossary/?term=networking), container (https://docs.docker.com/glossary/?term=container), standalone (https://docs.docker.com/glossary/?term=standalone)

Administer and maintain a swarm of Docker Engines

Estimated reading time: 16 minutes

When you run a swarm of Docker Engines, manager nodes are the key components for managing the swarm and storing the swarm state. It is important to understand some key features of manager nodes to properly deploy and maintain the swarm.

Refer to How nodes work (https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/) for a brief overview of Docker Swarm mode and the difference between manager and worker nodes.

Operate manager nodes in a swarm

Swarm manager nodes use the Raft Consensus Algorithm (https://docs.docker.com/engine/swarm/raft/) to manage the swarm state. You only need to understand some general concepts of Raft in order to manage a swarm.

There is no limit on the number of manager nodes. The decision about how many manager nodes to implement is a trade-off between performance and fault-tolerance. Adding manager nodes to a swarm makes the swarm more fault-tolerant. However, additional manager nodes reduce write performance because more nodes must acknowledge proposals to update the swarm state. This means more network round-trip traffic.

Raft requires a majority of managers, also called the quorum, to agree on proposed updates to the swarm, such as node additions or removals. Membership operations are subject to the same constraints as state replication.

Maintain the quorum of managers

If the swarm loses the quorum of managers, the swarm cannot perform management tasks. If your swarm has multiple managers, always have more than two. To maintain quorum, a majority of managers must be available. An odd number of managers is recommended, because the next even number does not make the quorum easier to keep. For instance, whether you have 3 or 4 managers, you can still only lose 1 manager and maintain the quorum. If you have 5 or 6 managers, you can still only lose two.

Even if a swarm loses the quorum of managers, swarm tasks on existing worker nodes continue to run. However, swarm nodes cannot be added, updated, or removed, and new or existing tasks cannot be started, stopped, moved, or updated.

See Recovering from losing the quorum (/engine/swarm/admin_guide/#recovering-from-losing-the-quorum) for troubleshooting steps if you do lose the quorum of managers.

Configure the manager to advertise on a static IP address

When initiating a swarm, you must specify the `--advertise-addr` flag to advertise your address to other manager nodes in the swarm. For more information, see Run Docker Engine in swarm mode (https://docs.docker.com/engine/swarm/swarm-mode/#configure-the-advertise-address). Because manager nodes are meant to be a stable component of the infrastructure, you should use a *fixed IP address* for the advertise address to prevent the swarm from becoming unstable on machine reboot.

If the whole swarm restarts and every manager node subsequently gets a new IP address, there is no way for any node to contact an existing manager. Therefore the swarm is hung while nodes try to contact one another at their old IP addresses.

Dynamic IP addresses are OK for worker nodes.

Add manager nodes for fault tolerance

You should maintain an odd number of managers in the swarm to support manager node failures. Having an odd number of managers ensures that during a network partition, there is a higher chance that the quorum remains available to process requests if the network is partitioned into two sets. Keeping the quorum is not guaranteed if you encounter more than two network partitions.

Swarm Size	Majority	Fault Tolerance
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3
8	5	3
9	5	4

For example, in a swarm with *5 nodes*, if you lose *3 nodes*, you don't have a quorum. Therefore you can't add or remove nodes until you recover one of the unavailable manager nodes or recover the swarm with disaster recovery commands. See Recover from disaster (/engine/swarm/admin_guide/#recover-from-disaster).

While it is possible to scale a swarm down to a single manager node, it is impossible to demote the last manager node. This ensures you maintain access to the swarm and that the swarm can still process requests. Scaling down to a single manager is an unsafe operation and is not recommended. If the last node leaves the swarm unexpectedly during the demote operation, the swarm becomes unavailable until you reboot the node or restart with `--force-new-cluster` .

You manage swarm membership with the `docker swarm` and `docker node` subsystems. Refer to Add nodes to a swarm (https://docs.docker.com/engine/swarm/join-nodes/) for more information on how to add worker nodes and promote a worker node to be a manager.

Distribute manager nodes

In addition to maintaining an odd number of manager nodes, pay attention to datacenter topology when placing managers. For optimal fault-tolerance, distribute manager nodes across a minimum of 3 availability-zones to support failures of an entire set of machines or common maintenance scenarios. If you suffer a failure in any of those zones, the swarm should maintain the quorum of manager nodes available to process requests and rebalance workloads.

Swarm manager nodes	Repartition (on 3 Availability zones)
3	1-1-1
5	2-2-1
7	3-2-2
9	3-3-3

Run manager-only nodes

By default manager nodes also act as a worker nodes. This means the scheduler can assign tasks to a manager node. For small and non-critical swarms assigning tasks to managers is relatively low-risk as long as you schedule services using resource constraints for *cpu* and *memory*.

However, because manager nodes use the Raft consensus algorithm to replicate data in a consistent way, they are sensitive to resource starvation. You should isolate managers in your swarm from processes that might block swarm operations like swarm heartbeat or leader elections.

To avoid interference with manager node operation, you can drain manager nodes to make them unavailable as worker nodes:

```
docker node update --availability drain <NODE>
```

When you drain a node, the scheduler reassigns any tasks running on the node to other available worker nodes in the swarm. It also prevents the scheduler from assigning tasks to the node.

Add worker nodes for load balancing

Add nodes to the swarm (https://docs.docker.com/engine/swarm/join-nodes/) to balance your swarm's load. Replicated service tasks are distributed across the swarm as evenly as possible over time, as long as the worker nodes are matched to the requirements of the services. When limiting a service to run on only specific types of nodes, such as nodes with a specific number of CPUs or amount of memory, remember that worker nodes that do not meet these requirements cannot run these tasks.

Monitor swarm health

You can monitor the health of manager nodes by querying the docker `nodes` API in JSON format through the `/nodes` HTTP endpoint. Refer to the nodes API documentation (https://docs.docker.com/engine/api/v1.25/#tag/Node) for more information.

From the command line, run `docker node inspect <id-node>` to query the nodes. For instance, to query the reachability of the node as a manager:

```
docker node inspect manager1 --format "{{ .ManagerStatus.Reachabilit
y }}"
reachable
```

To query the status of the node as a worker that accept tasks:

```
docker node inspect manager1 --format "{{ .Status.State }}"
ready
```

From those commands, we can see that `manager1` is both at the status `reachable` as a manager and `ready` as a worker.

An `unreachable` health status means that this particular manager node is unreachable from other manager nodes. In this case you need to take action to restore the unreachable manager:

- Restart the daemon and see if the manager comes back as reachable.
- Reboot the machine.
- If neither restarting or rebooting work, you should add another manager node or promote a worker to be a manager node. You also need to cleanly remove the failed node entry from the manager set with `docker node demote <NODE>` and `docker node rm <id-node>` .

Alternatively you can also get an overview of the swarm health from a manager node with `docker node ls` :

```
docker node ls
ID                           HOSTNAME  MEMBERSHIP  STATUS  AVAILABIL
ITY  MANAGER STATUS
1mhtdwhvsgr3c26xxbnzdc3yp    node05    Accepted    Ready   Active
516pacagkqp2xc3fk9t1dhjor    node02    Accepted    Ready   Active
     Reachable
9ifojw8of78kkusuc4a6c23fx *  node01    Accepted    Ready   Active
     Leader
ax11wdpwrrb6db3mfjydscgk7    node04    Accepted    Ready   Active
bb1nrq2cswhtbg4mrsqnlx1ck    node03    Accepted    Ready   Active
     Reachable
di9wxgz8dtuh9d2hn089ecqkf    node06    Accepted    Ready   Active
```

Troubleshoot a manager node

You should never restart a manager node by copying the `raft` directory from another node. The data directory is unique to a node ID. A node can only use a node ID once to join the swarm. The node ID space should be globally unique.

To cleanly re-join a manager node to a cluster:

1. To demote the node to a worker, run `docker node demote <NODE>` .
2. To remove the node from the swarm, run `docker node rm <NODE>` .
3. Re-join the node to the swarm with a fresh state using `docker swarm join` .

For more information on joining a manager node to a swarm, refer to Join nodes to a swarm (https://docs.docker.com/engine/swarm/join-nodes/).

Forcibly remove a node

In most cases, you should shut down a node before removing it from a swarm with the `docker node rm` command. If a node becomes unreachable, unresponsive, or compromised you can forcefully remove the node without shutting it down by passing the `--force` flag. For instance, if `node9` becomes compromised:

```
$ docker node rm node9

Error response from daemon: rpc error: code = 9 desc = node node9 is
 not down and can't be removed

$ docker node rm --force node9

Node node9 removed from swarm
```

Before you forcefully remove a manager node, you must first demote it to the worker role. Make sure that you always have an odd number of manager nodes if you demote or remove a manager.

Back up the swarm

Docker manager nodes store the swarm state and manager logs in the `/var/lib/docker/swarm/` directory. In 1.13 and higher, this data includes the keys used to encrypt the Raft logs. Without these keys, you cannot restore the swarm.

You can back up the swarm using any manager. Use the following procedure.

1. If the swarm has auto-lock enabled, you need the unlock key to restore the swarm from backup. Retrieve the unlock key if necessary and store it in a safe location. If you are unsure, read Lock your swarm to protect its encryption key (https://docs.docker.com/engine/swarm/swarm_manager_locking/).
2. Stop Docker on the manager before backing up the data, so that no data is being changed during the backup. It is possible to take a backup while the manager is running (a “hot” backup), but this is not recommended and your

results are less predictable when restoring. While the manager is down, other nodes continue generating swarm data that is not part of this backup.

> Note: Be sure to maintain the quorum of swarm managers. During the time that a manager is shut down, your swarm is more vulnerable to losing the quorum if further nodes are lost. The number of managers you run is a trade-off. If you regularly take down managers to do backups, consider running a 5-manager swarm, so that you can lose an additional manager while the backup is running, without disrupting your services.

3. Back up the entire `/var/lib/docker/swarm` directory.
4. Restart the manager.

To restore, see Restore from a backup (/engine/swarm/admin_guide/#restore-from-a-backup).

Recover from disaster

Restore from a backup

After backing up the swarm as described in Back up the swarm (/engine/swarm/admin_guide/#back-up-the-swarm), use the following procedure to restore the data to a new swarm.

1. Shut down Docker on the target host machine for the restored swarm.
2. Remove the contents of the `/var/lib/docker/swarm` directory on the new swarm.
3. Restore the `/var/lib/docker/swarm` directory with the contents of the backup.

> ✓ Note: The new node uses the same encryption key for on-disk storage as the old one. It is not possible to change the on-disk storage encryption keys at this time.
>
> In the case of a swarm with auto-lock enabled, the unlock key is also the same as on the old swarm, and the unlock key is needed to restore the swarm.

4. Start Docker on the new node. Unlock the swarm if necessary. Re-initialize the swarm using the following command, so that this node does not attempt to connect to nodes that were part of the old swarm, and presumably no longer exist.

   ```
   $ docker swarm init --force-new-cluster
   ```

5. Verify that the state of the swarm is as expected. This may include application-specific tests or simply checking the output of `docker service ls` to be sure that all expected services are present.
6. If you use auto-lock, rotate the unlock key (https://docs.docker.com/engine/swarm/swarm_manager_locking/#rotate-the-unlock-key).
7. Add manager and worker nodes to bring your new swarm up to operating capacity.
8. Reinstate your previous backup regimen on the new swarm.

Recover from losing the quorum

Swarm is resilient to failures and the swarm can recover from any number of temporary node failures (machine reboots or crash with restart) or other transient errors. However, a swarm cannot automatically recover if it loses a quorum. Tasks on existing worker nodes continue to run, but administrative tasks are not possible, including scaling or updating services and joining or removing nodes from the swarm. The best way to recover is to bring the missing manager nodes back online. If that is not possible, continue reading for some options for recovering your swarm.

In a swarm of `N` managers, a quorum (a majority) of manager nodes must always be available. For example, in a swarm with 5 managers, a minimum of 3 must be operational and in communication with each other. In other words, the swarm can tolerate up to `(N-1)/2` permanent failures beyond which requests involving swarm management cannot be processed. These types of failures include data corruption or hardware failures.

If you lose the quorum of managers, you cannot administer the swarm. If you have lost the quorum and you attempt to perform any management operation on the swarm, an error occurs:

```
Error response from daemon: rpc error: code = 4 desc = context deadl
ine exceeded
```

The best way to recover from losing the quorum is to bring the failed nodes back online. If you can't do that, the only way to recover from this state is to use the `--force-new-cluster` action from a manager node. This removes all managers except the manager the command was run from. The quorum is achieved because there is now only one manager. Promote nodes to be managers until you have the desired number of managers.

```
# From the node to recover
docker swarm init --force-new-cluster --advertise-addr node01:2377
```

When you run the `docker swarm init` command with the `--force-new-cluster` flag, the Docker Engine where you run the command becomes the manager node of a single-node swarm which is capable of managing and running services. The manager has all the previous information about services and tasks, worker nodes are still part of the swarm, and services are still running. You need to add or re-add manager nodes to achieve your previous task distribution and ensure that you have enough managers to maintain high availability and prevent losing the quorum.

Force the swarm to rebalance

Generally, you do not need to force the swarm to rebalance its tasks. When you add a new node to a swarm, or a node reconnects to the swarm after a period of unavailability, the swarm does not automatically give a workload to the idle node. This is a design decision. If the swarm periodically shifted tasks to different nodes for the sake of balance, the clients using those tasks would be disrupted. The goal is to avoid disrupting running services for the sake of balance across the swarm. When new tasks start, or when a node with running tasks becomes unavailable, those tasks are given to less busy nodes. The goal is eventual balance, with minimal disruption to the end user.

In Docker 1.13 and higher, you can use the `--force` or `-f` flag with the `docker service update` command to force the service to redistribute its tasks across the available worker nodes. This causes the service tasks to restart. Client applications may be disrupted. If you have configured it, your service uses a rolling update (https://docs.docker.com/engine/swarm/swarm-tutorial/rolling-update/).

If you use an earlier version and you want to achieve an even balance of load across workers and don't mind disrupting running tasks, you can force your swarm to re-balance by temporarily scaling the service upward. Use `docker service inspect --pretty <servicename>` to see the configured scale of a service. When you use `docker service scale` , the nodes with the lowest number of tasks are targeted to receive the new workloads. There may be multiple under-loaded nodes in your swarm. You may need to scale the service up by modest increments a few times to achieve the balance you want across all the nodes.

When the load is balanced to your satisfaction, you can scale the service back down to the original scale. You can use `docker service ps` to assess the current balance of your service across nodes.

See also `docker service scale` (https://docs.docker.com/engine/reference/commandline/service_scale/) and `docker service ps` (https://docs.docker.com/engine/reference/commandline/service_ps/).

docker (https://docs.docker.com/glossary/?term=docker), container (https://docs.docker.com/glossary/?term=container), swarm (https://docs.docker.com/glossary/?term=swarm), manager

(https://docs.docker.com/glossary/?term=manager), raft
(https://docs.docker.com/glossary/?term=raft)