Use bridge networks

Estimated reading time: 9 minutes

In terms of networking, a bridge network is a Link Layer device which forwards traffic between network segments. A bridge can be a hardware device or a software device running within a host machine's kernel.

In terms of Docker, a bridge network uses a software bridge which allows containers connected to the same bridge network to communicate, while providing isolation from containers which are not connected to that bridge network. The Docker bridge driver automatically installs rules in the host machine so that containers on different bridge networks cannot communicate directly with each other.

Bridge networks apply to containers running on the same Docker daemon host. For communication among containers running on different Docker daemon hosts, you can either manage routing at the OS level, or you can use an overlay network (https://docs.docker.com/network/overlay/).

When you start Docker, a default bridge network (/network/bridge/#use-the-default-bridge-network) (also called `bridge`) is created automatically, and newly-started containers connect to it unless otherwise specified. You can also create user-defined custom bridge networks. User-defined bridge networks are superior to the default `bridge` network.

Differences between user-defined bridges and the default bridge

- User-defined bridges provide better isolation and interoperability between containerized applications.

  Containers connected to the same user-defined bridge network automatically expose all ports to each other, and no ports to the outside world. This allows containerized applications to communicate with each other easily, without accidentally opening access to the outside world.

Imagine an application with a web front-end and a database back-end. The outside world needs access to the web front-end (perhaps on port 80), but only the back-end itself needs access to the database host and port. Using a user-defined bridge, only the web port needs to be opened, and the database application doesn't need any ports open, since the web front-end can reach it over the user-defined bridge.

If you run the same application stack on the default bridge network, you need to open both the web port and the database port, using the `-p` or `--publish` flag for each. This means the Docker host needs to block access to the database port by other means.

- User-defined bridges provide automatic DNS resolution between containers.

  Containers on the default bridge network can only access each other by IP addresses, unless you use the `--link` option (https://docs.docker.com/network/links/), which is considered legacy. On a user-defined bridge network, containers can resolve each other by name or alias.

  Imagine the same application as in the previous point, with a web front-end and a database back-end. If you call your containers `web` and `db`, the web container can connect to the db container at `db`, no matter which Docker host the application stack is running on.

  If you run the same application stack on the default bridge network, you need to manually create links between the containers (using the legacy `--link` flag). These links need to be created in both directions, so you can see this gets complex with more than two containers which need to communicate. Alternatively, you can manipulate the `/etc/hosts` files within the containers, but this creates problems that are difficult to debug.

- Containers can be attached and detached from user-defined networks on the fly.

  During a container's lifetime, you can connect or disconnect it from user-defined networks on the fly. To remove a container from the default bridge network, you need to stop the container and recreate it with different network options.

- Each user-defined network creates a configurable bridge.

If your containers use the default bridge network, you can configure it, but all the containers use the same settings, such as MTU and `iptables` rules. In addition, configuring the default bridge network happens outside of Docker itself, and requires a restart of Docker.

User-defined bridge networks are created and configured using `docker network create`. If different groups of applications have different network requirements, you can configure each user-defined bridge separately, as you create it.

- Linked containers on the default bridge network share environment variables.

  Originally, the only way to share environment variables between two containers was to link them using the `--link` flag (https://docs.docker.com/network/links/). This type of variable sharing is not possible with user-defined networks. However, there are superior ways to share environment variables. A few ideas:

  - Multiple containers can mount a file or directory containing the shared information, using a Docker volume.
  - Multiple containers can be started together using `docker-compose` and the compose file can define the shared variables.
  - You can use swarm services instead of standalone containers, and take advantage of shared secrets (https://docs.docker.com/engine/swarm/secrets/) and configs (https://docs.docker.com/engine/swarm/configs/).

Containers connected to the same user-defined bridge network effectively expose all ports to each other. For a port to be accessible to containers or non-Docker hosts on different networks, that port must be *published* using the `-p` or `--publish` flag.

Manage a user-defined bridge

Use the `docker network create` command to create a user-defined bridge network.

```
$ docker network create my-net
```

You can specify the subnet, the IP address range, the gateway, and other options. See the docker network create (https://docs.docker.com/engine/reference/commandline/network_create/#specify-advanced-options) reference or the output of `docker network create --help` for details.

Use the `docker network rm` command to remove a user-defined bridge network. If containers are currently connected to the network, disconnect them (/network/bridge/#disconnect-a-container-from-a-user-defined-bridge) first.

```
$ docker network rm my-net
```

> What's really happening?
>
> When you create or remove a user-defined bridge or connect or disconnect a container from a user-defined bridge, Docker uses tools specific to the operating system to manage the underlying network infrastructure (such as adding or removing bridge devices or configuring `iptables` rules on Linux). These details should be considered implementation details. Let Docker manage your user-defined networks for you.

Connect a container to a user-defined bridge

When you create a new container, you can specify one or more `--network` flags. This example connects a Nginx container to the `my-net` network. It also publishes port 80 in the container to port 8080 on the Docker host, so external clients can access that port. Any other container connected to the `my-net` network has access to all ports on the `my-nginx` container, and vice versa.

```
$ docker create --name my-nginx \
  --network my-net \
  --publish 8080:80 \
  nginx:latest
```

To connect a running container to an existing user-defined bridge, use the `docker network connect` command. The following command connects an already-running `my-nginx` container to an already-existing `my-net` network:

```
$ docker network connect my-net my-nginx
```

Disconnect a container from a user-defined bridge

To disconnect a running container from a user-defined bridge, use the `docker network disconnect` command. The following command disconnects the `my-nginx` container from the `my-net` network.

```
$ docker network disconnect my-net my-nginx
```

Use IPv6

If you need IPv6 support for Docker containers, you need to enable the option (https://docs.docker.com/config/daemon/ipv6/) on the Docker daemon and reload its configuration, before creating any IPv6 networks or assigning containers IPv6 addresses.

When you create your network, you can specify the `--ipv6` flag to enable IPv6. You can't selectively disable IPv6 support on the default `bridge` network.

Enable forwarding from Docker containers to the outside world

By default, traffic from containers connected to the default bridge network is not forwarded to the outside world. To enable forwarding, you need to change two settings. These are not Docker commands and they affect the Docker host's kernel.

1. Configure the Linux kernel to allow IP forwarding.

```
$ sysctl net.ipv4.conf.all.forwarding=1
```

2. Change the policy for the `iptables` `FORWARD` policy from `DROP` to `ACCEPT` .

```
$ sudo iptables -P FORWARD ACCEPT
```

These settings do not persist across a reboot, so you may need to add them to a start-up script.

Use the default bridge network

The default `bridge` network is considered a legacy detail of Docker and is not recommended for production use. Configuring it is a manual operation, and it has technical shortcomings (/network/bridge/#differences-between-user-defined-bridges-and-the-default-bridge).

Connect a container to the default bridge network

If you do not specify a network using the `--network` flag, and you do specify a network driver, your container is connected to the default `bridge` network by default. Containers connected to the default `bridge` network can communicate, but only by IP address, unless they are linked using the legacy `--link` flag (https://docs.docker.com/network/links/).

Configure the default bridge network

To configure the default `bridge` network, you specify options in `daemon.json` . Here is an example `daemon.json` with several options specified. Only specify the settings you need to customize.

```
{
  "bip": "192.168.1.5/24",
  "fixed-cidr": "192.168.1.5/25",
  "fixed-cidr-v6": "2001:db8::/64",
  "mtu": 1500,
  "default-gateway": "10.20.1.1",
  "default-gateway-v6": "2001:db8:abcd::89",
  "dns": ["10.20.1.2","10.20.1.3"]
}
```

Restart Docker for the changes to take effect.

Use IPv6 with the default bridge network

If you configure Docker for IPv6 support (see Use IPv6 (/network/bridge/#use-ipv6)), the default bridge network is also configured for IPv6 automatically. Unlike user-defined bridges, you can't selectively disable IPv6 on the default bridge.

Next steps

- Go through the standalone networking tutorial (https://docs.docker.com/network/network-tutorial-standalone/)
- Learn about networking from the container's point of view (https://docs.docker.com/config/containers/container-networking/)
- Learn about overlay networks (https://docs.docker.com/network/overlay/)
- Learn about Macvlan networks (https://docs.docker.com/network/macvlan/)

network (https://docs.docker.com/glossary/?term=network), bridge (https://docs.docker.com/glossary/?term=bridge), user-defined (https://docs.docker.com/glossary/?term=user-defined), standalone (https://docs.docker.com/glossary/?term=standalone)

docker network create

Estimated reading time: 9 minutes

Description

Create a network

Usage

```
docker network create [OPTIONS] NETWORK
```

Options

Name, shorthand	Default	Description
`--attachable`		API 1.25+ (https://docs.docker.com/engine/api/v1.25/) Enable manual container attachment
`--aux-address`		Auxiliary IPv4 or IPv6 addresses used by Network driver
`--config-from`		API 1.30+ (https://docs.docker.com/engine/api/v1.30/) The network from which copying the configuration
`--config-only`		API 1.30+ (https://docs.docker.com/engine/api/v1.30/) Create a configuration only network
`--driver , -d`	`bridge`	Driver to manage the Network
`--gateway`		IPv4 or IPv6 Gateway for the master subnet
`--ingress`		API 1.29+ (https://docs.docker.com/engine/api/v1.29/) Create swarm routing-mesh network
`--internal`		Restrict external access to the network

Name, shorthand	Default	Description
`--ip-range`		Allocate container ip from a sub-range
`--ipam-driver`		IP Address Management Driver
`--ipam-opt`		Set IPAM driver specific options
`--ipv6`		Enable IPv6 networking
`--label`		Set metadata on a network
`--opt , -o`		Set driver specific options
`--scope`		API 1.30+ (https://docs.docker.com/engine/api/v1.30/) Control the network's scope
`--subnet`		Subnet in CIDR format that represents a network segment

Parent command

Command	Description
docker network (https://docs.docker.com/engine/reference/commandline/network)	Manage networks

Related commands

Command	Description
docker network connect (https://docs.docker.com/engine/reference/commandline/network_connect/)	Connect a container to a network
docker network create (https://docs.docker.com/engine/reference/commandline/network_create/)	Create a network

Command	Description
docker network disconnect (https://docs.docker.com/engine/reference/commandline/network_disconnect/)	Disconnect a container from a network
docker network inspect (https://docs.docker.com/engine/reference/commandline/network_inspect/)	Display detailed information on one or more networks
docker network ls (https://docs.docker.com/engine/reference/commandline/network_ls/)	List networks
docker network prune (https://docs.docker.com/engine/reference/commandline/network_prune/)	Remove all unused networks
docker network rm (https://docs.docker.com/engine/reference/commandline/network_rm/)	Remove one or more networks

Extended description

Creates a new network. The `DRIVER` accepts `bridge` or `overlay` which are the built-in network drivers. If you have installed a third party or your own custom network driver you can specify that `DRIVER` here also. If you don't specify the `--driver` option, the command automatically creates a `bridge` network for you. When you install Docker Engine it creates a `bridge` network automatically. This network corresponds to the `docker0` bridge that Engine has traditionally relied on. When you launch a new container with `docker run` it automatically connects to this bridge network. You cannot remove this default bridge network, but you can create new ones using the `network create` command.

```
$ docker network create -d bridge my-bridge-network
```

Bridge networks are isolated networks on a single Engine installation. If you want to create a network that spans multiple Docker hosts each running an Engine, you must create an `overlay` network. Unlike `bridge` networks, overlay networks require some pre-existing conditions before you can create one. These conditions are:

- Access to a key-value store. Engine supports Consul, Etcd, and ZooKeeper (Distributed store) key-value stores.
- A cluster of hosts with connectivity to the key-value store.
- A properly configured Engine `daemon` on each host in the cluster.

The `dockerd` options that support the `overlay` network are:

- `--cluster-store`
- `--cluster-store-opt`
- `--cluster-advertise`

To read more about these options and how to configure them, see "*Get started with multi-host network*" (https://docs.docker.com/engine/userguide/networking/get-started-overlay).

While not required, it is a good idea to install Docker Swarm to manage the cluster that makes up your network. Swarm provides sophisticated discovery and server management tools that can assist your implementation.

Once you have prepared the `overlay` network prerequisites you simply choose a Docker host in the cluster and issue the following to create the network:

```
$ docker network create -d overlay my-multihost-network
```

Network names must be unique. The Docker daemon attempts to identify naming conflicts but this is not guaranteed. It is the user's responsibility to avoid name conflicts.

Overlay network limitations

You should create overlay networks with `/24` blocks (the default), which limits you to 256 IP addresses, when you create networks using the default VIP-based endpoint-mode. This recommendation addresses limitations with swarm mode (https://github.com/moby/moby/issues/30820). If you need more than 256 IP addresses, do not increase the IP block size. You can either use `dnsrr` endpoint mode with an external load balancer, or use multiple smaller overlay networks. See Configure service discovery (https://docs.docker.com/engine/swarm/networking/#configure-service-discovery) for more information about different endpoint modes.

Examples

Connect containers

When you start a container, use the `--network` flag to connect it to a network. This example adds the `busybox` container to the `mynet` network:

```
$ docker run -itd --network=mynet busybox
```

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

You can connect multiple containers to the same network. Once connected, the containers can communicate using 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.

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

Specify advanced options

When you create a network, Engine creates a non-overlapping subnetwork for the network by default. This subnetwork is not a subdivision of an existing network. It is purely for ip-addressing purposes. You can override this default and specify subnetwork values directly using the `--subnet` option. On a `bridge` network you can only create a single subnet:

```
$ docker network create --driver=bridge --subnet=192.168.0.0/16 br0
```

Additionally, you also specify the `--gateway` `--ip-range` and `--aux-address` options.

```
$ docker network create \
  --driver=bridge \
  --subnet=172.28.0.0/16 \
  --ip-range=172.28.5.0/24 \
  --gateway=172.28.5.254 \
  br0
```

If you omit the `--gateway` flag the Engine selects one for you from inside a preferred pool. For `overlay` networks and for network driver plugins that support it you can create multiple subnetworks. This example uses two `/25` subnet mask to adhere to the current guidance of not having more than 256 IPs in a single overlay network. Each of the subnetworks has 126 usable addresses.

```
$ docker network create -d overlay \
  --subnet=192.168.1.0/25 \
  --subnet=192.170.2.0/25 \
  --gateway=192.168.1.100 \
  --gateway=192.170.2.100 \
  --aux-address="my-router=192.168.1.5" --aux-address="my-switch=192.168.1
.6" \
  --aux-address="my-printer=192.170.1.5" --aux-address="my-nas=192.170.1.6
" \
  my-multihost-network
```

Be sure that your subnetworks do not overlap. If they do, the network create fails and Engine returns an error.

Bridge driver options

When creating a custom network, the default network driver (i.e. `bridge`) has additional options that can be passed. The following are those options and the equivalent docker daemon flags used for docker0 bridge:

Option	Equivalent	Description
`com.docker.network.bridge.name`	-	bridge name to be used when creating the Linux bridge
`com.docker.network.bridge.enable_ip_masquerade`	`--ip-masq`	Enable IP masquerading
`com.docker.network.bridge.enable_icc`	`--icc`	Enable or Disable Inter Container Connectivity
`com.docker.network.bridge.host_binding_ipv4`	`--ip`	Default IP when binding container ports

Option	Equivalent	Description
`com.docker.network.driver.mtu`	`--mtu`	Set the containers network MTU

The following arguments can be passed to `docker network create` for any network driver, again with their approximate equivalents to `docker daemon`.

Argument	Equivalent	Description
`--gateway`	-	IPv4 or IPv6 Gateway for the master subnet
`--ip-range`	`--fixed-cidr`	Allocate IPs from a range
`--internal`	-	Restrict external access to the network
`--ipv6`	`--ipv6`	Enable IPv6 networking
`--subnet`	`--bip`	Subnet for network

For example, let's use `-o` or `--opt` options to specify an IP address binding when publishing ports:

```
$ docker network create \
    -o "com.docker.network.bridge.host_binding_ipv4"="172.19.0.1" \
    simple-network
```

Network internal mode

By default, when you connect a container to an `overlay` network, Docker also connects a bridge network to it to provide external connectivity. If you want to create an externally isolated `overlay` network, you can specify the `--internal` option.

Network ingress mode

You can create the network which will be used to provide the routing-mesh in the swarm cluster. You do so by specifying `--ingress` when creating the network. Only one ingress network can be created at the time. The network can be removed only if no services depend on it. Any option available when creating an overlay network is also available when creating the ingress network, besides the `--attachable` option.

```
$ docker network create -d overlay \
  --subnet=10.11.0.0/16 \
  --ingress \
  --opt com.docker.network.driver.mtu=9216 \
  --opt encrypted=true \
  my-ingress-network
```

docker network inspect

Estimated reading time: 1 minute

Description

Display detailed information on one or more networks

Usage

```
docker network inspect [OPTIONS] NETWORK [NETWORK...]
```

Options

Name, shorthand	Default	Description
`--format , -f`		Format the output using the given Go template
`--verbose , -v`		Verbose output for diagnostics

Parent command

Command	Description
docker network (https://docs.docker.com/engine/reference/commandline/network)	Manage networks

Related commands

Command	Description

Command	Description
docker network connect (https://docs.docker.com/engine/reference/commandline/network_connect/)	Connect a container to a network
docker network create (https://docs.docker.com/engine/reference/commandline/network_create/)	Create a network
docker network disconnect (https://docs.docker.com/engine/reference/commandline/network_disconnect/)	Disconnect a container from a network
docker network inspect (https://docs.docker.com/engine/reference/commandline/network_inspect/)	Display detailed information on one or more networks
docker network ls (https://docs.docker.com/engine/reference/commandline/network_ls/)	List networks
docker network prune (https://docs.docker.com/engine/reference/commandline/network_prune/)	Remove all unused networks
docker network rm (https://docs.docker.com/engine/reference/commandline/network_rm/)	Remove one or more networks

Extended description

Returns information about one or more networks. By default, this command renders all results in a JSON object.

Estimated reading time: 61 minutes

Docker run reference

Docker runs processes in isolated containers. A container is a process which runs on a host. The host may be local or remote. When an operator executes `docker run`, the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host.

This page details how to use the `docker run` command to define the container's resources at runtime.

General form

The basic `docker run` command takes this form:

```
$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

The `docker run` command must specify an *IMAGE* (https://docs.docker.com/engine/reference/glossary/#image) to derive the container from. An image developer can define image defaults related to:

- detached or foreground running
- container identification
- network settings
- runtime constraints on CPU and memory

With the `docker run [OPTIONS]` an operator can add to or override the image defaults set by a developer. And, additionally, operators can override nearly all the defaults set by the Docker runtime itself. The operator's ability to override image and Docker runtime defaults is why *run* (https://docs.docker.com/engine/reference/commandline/run/) has more options than any other `docker` command.

To learn how to interpret the types of `[OPTIONS]`, see *Option types* (https://docs.docker.com/engine/reference/commandline/cli/#option-types).

> Note: Depending on your Docker system configuration, you may be required to preface the `docker run` command with `sudo`. To avoid having to use `sudo` with the `docker` command, your system administrator can create a Unix group called `docker` and add users to it. For more information about this configuration, refer to the Docker installation documentation for your operating system.

Operator exclusive options

Only the operator (the person executing `docker run`) can set the following options.

- Detached vs foreground (/engine/reference/run/#detached-vs-foreground)
  - Detached (-d) (/engine/reference/run/#detached--d)
  - Foreground (/engine/reference/run/#foreground)
- Container identification (/engine/reference/run/#container-identification)

    - Name (--name) (/engine/reference/run/#name---name)
    - PID equivalent (/engine/reference/run/#pid-equivalent)
- IPC settings (--ipc) (/engine/reference/run/#ipc-settings---ipc)
- Network settings (/engine/reference/run/#network-settings)
- Restart policies (--restart) (/engine/reference/run/#restart-policies---restart)
- Clean up (--rm) (/engine/reference/run/#clean-up---rm)
- Runtime constraints on resources (/engine/reference/run/#runtime-constraints-on-resources)
- Runtime privilege and Linux capabilities (/engine/reference/run/#runtime-privilege-and-linux-capabilities)

Detached vs foreground

When starting a Docker container, you must first decide if you want to run the container in the background in a “detached” mode or in the default foreground mode:

```
-d=false: Detached mode: Run container in the background, print new container id
```

Detached (-d)

To start a container in detached mode, you use `-d=true` or just `-d` option. By design, containers started in detached mode exit when the root process used to run the container exits, unless you also specify the `--rm` option. If you use `-d` with `--rm`, the container is removed when it exits or when the daemon exits, whichever happens first.

Do not pass a `service x start` command to a detached container. For example, this command attempts to start the `nginx` service.

```
$ docker run -d -p 80:80 my_image service nginx start
```

This succeeds in starting the `nginx` service inside the container. However, it fails the detached container paradigm in that, the root process (`service nginx start`) returns and the detached container stops as designed. As a result, the `nginx` service is started but could not be used. Instead, to start a process such as the `nginx` web server do the following:

```
$ docker run -d -p 80:80 my_image nginx -g 'daemon off;'
```

To do input/output with a detached container use network connections or shared volumes. These are required because the container is no longer listening to the command line where `docker run` was run.

To reattach to a detached container, use `docker` *attach* (https://docs.docker.com/engine/reference/commandline/attach/) command.

Foreground

In foreground mode (the default when `-d` is not specified), `docker run` can start the process in the container and attach the console to the process’s standard input, output, and standard error. It can even pretend to be a TTY (this is what most command line executables expect) and pass along signals. All of that is configurable:

```
-a=[]           : Attach to ‘STDIN‘, ‘STDOUT‘ and/or ‘STDERR‘
-t              : Allocate a pseudo-tty
--sig-proxy=true: Proxy all received signals to the process (non-TTY mode only)
-i              : Keep STDIN open even if not attached
```

If you do not specify `-a` then Docker will attach to both stdout and stderr (https://github.com/docker/docker/blob/4118e0c9eebda2412a09ae66e90c34b85fae3275/runconfig/opts/parse.go#L267). You can specify to which of the three standard streams (`STDIN` , `STDOUT` , `STDERR`) you’d like to connect instead, as in:

```
$ docker run -a stdin -a stdout -i -t ubuntu /bin/bash
```

For interactive processes (like a shell), you must use `-i -t` together in order to allocate a tty for the container process. `-i -t` is often written `-it` as you’ll see in later examples. Specifying `-t` is forbidden when the client is receiving its standard input from a pipe, as in:

```
$ echo test | docker run -i busybox cat
```

> Note: A process running as PID 1 inside a container is treated specially by Linux: it ignores any signal with the default action. So, the process will not terminate on `SIGINT` or `SIGTERM` unless it is coded to do so.

Container identification

Name (--name)

The operator can identify a container in three ways:

Identifier type	Example value
UUID long identifier	“f78375b1c487e03c9438c729345e54db9d20cfa2ac1fc3494b6eb60872e74778”
UUID short identifier	“f78375b1c487”
Name	“evil_ptolemy”

The UUID identifiers come from the Docker daemon. If you do not assign a container name with the `--name` option, then the daemon generates a random string name for you. Defining a `name` can be a handy way to add meaning to a container. If you specify a `name` , you can use it when referencing the container within a Docker network. This works for both background and foreground Docker containers.

> Note: Containers on the default bridge network must be linked to communicate by name.

PID equivalent

Finally, to help with automation, you can have Docker write the container ID out to a file of your choosing. This

is similar to how some programs might write out their process ID to a file (you've seen them as PID files):

```
--cidfile="": Write the container ID to the file
```

Image[:tag]

While not strictly a means of identifying a container, you can specify a version of an image you'd like to run the container with by adding `image[:tag]` to the command. For example, `docker run ubuntu:14.04` .

Image[@digest]

Images using the v2 or later image format have a content-addressable identifier called a digest. As long as the input used to generate the image is unchanged, the digest value is predictable and referenceable.

The following example runs a container from the `alpine` image with the `sha256:9cacb71397b640eca97488cf08582ae4e4068513101088e9f96c9814bfda95e0` digest:

```
$ docker run alpine@sha256:9cacb71397b640eca97488cf08582ae4e4068513101088e9f96c9814bfda95e0 dat
e
```

PID settings (--pid)

```
--pid=""  : Set the PID (Process) Namespace mode for the container,
            'container:<name|id>': joins another container's PID namespace
            'host': use the host's PID namespace inside the container
```

By default, all containers have the PID namespace enabled.

PID namespace provides separation of processes. The PID Namespace removes the view of the system processes, and allows process ids to be reused including pid 1.

In certain cases you want your container to share the host's process namespace, basically allowing processes within the container to see all of the processes on the system. For example, you could build a container with debugging tools like `strace` or `gdb` , but want to use these tools when debugging processes within the container.

Example: run htop inside a container

Create this Dockerfile:

```
FROM alpine:latest
RUN apk add --update htop && rm -rf /var/cache/apk/*
CMD ["htop"]
```

Build the Dockerfile and tag the image as `myhtop` :

```
$ docker build -t myhtop .
```

Use the following command to run `htop` inside a container:

```
$ docker run -it --rm --pid=host myhtop
```

Joining another container's pid namespace can be used for debugging that container.

Example

Start a container running a redis server:

```
$ docker run --name my-redis -d redis
```

Debug the redis container by running another container that has strace in it:

```
$ docker run -it --pid=container:my-redis my_strace_docker_image bash
$ strace -p 1
```

UTS settings (--uts)

```
--uts=""  : Set the UTS namespace mode for the container,
       'host': use the host's UTS namespace inside the container
```

The UTS namespace is for setting the hostname and the domain that is visible to running processes in that namespace. By default, all containers, including those with `--network=host`, have their own UTS namespace. The `host` setting will result in the container using the same UTS namespace as the host. Note that `--hostname` is invalid in `host` UTS mode.

You may wish to share the UTS namespace with the host if you would like the hostname of the container to change as the hostname of the host changes. A more advanced use case would be changing the host's hostname from a container.

IPC settings (--ipc)

```
--ipc="MODE"  : Set the IPC mode for the container
```

The following values are accepted:

Value	Description
""	Use daemon's default.
"none"	Own private IPC namespace, with /dev/shm not mounted.
"private"	Own private IPC namespace.

Value	Description
“shareable”	Own private IPC namespace, with a possibility to share it with other containers.
“container: <_name-or-ID_>"	Join another (“shareable”) container’s IPC namespace.
“host”	Use the host system’s IPC namespace.

If not specified, daemon default is used, which can either be `"private"` or `"shareable"`, depending on the daemon version and configuration.

IPC (POSIX/SysV IPC) namespace provides separation of named shared memory segments, semaphores and message queues.

Shared memory segments are used to accelerate inter-process communication at memory speed, rather than through pipes or through the network stack. Shared memory is commonly used by databases and custom-built (typically C/OpenMPI, C++/using boost libraries) high performance applications for scientific computing and financial services industries. If these types of applications are broken into multiple containers, you might need to share the IPC mechanisms of the containers, using `"shareable"` mode for the main (i.e. “donor”) container, and `"container:<donor-name-or-ID>"` for other containers.

Network settings

```
--dns=[]           : Set custom dns servers for the container
--network="bridge" : Connect a container to a network
                     'bridge': create a network stack on the default Docker bridge
                     'none': no networking
                     'container:<name|id>': reuse another container's network stack
                     'host': use the Docker host network stack
                     '<network-name>|<network-id>': connect to a user-defined network
--network-alias=[] : Add network-scoped alias for the container
--add-host=""      : Add a line to /etc/hosts (host:IP)
--mac-address=""   : Sets the container's Ethernet device's MAC address
--ip=""            : Sets the container's Ethernet device's IPv4 address
--ip6=""           : Sets the container's Ethernet device's IPv6 address
--link-local-ip=[] : Sets one or more container's Ethernet device's link local IPv4/IPv6 addres
ses
```

By default, all containers have networking enabled and they can make any outgoing connections. The operator can completely disable networking with `docker run --network none` which disables all incoming and outgoing networking. In cases like this, you would perform I/O through files or `STDIN` and `STDOUT` only.

Publishing ports and linking to other containers only works with the default (bridge). The linking feature is a legacy feature. You should always prefer using Docker network drivers over linking.

Your container will use the same DNS servers as the host by default, but you can override this with `--dns`.

By default, the MAC address is generated using the IP address allocated to the container. You can set the container’s MAC address explicitly by providing a MAC address via the `--mac-address` parameter (format: `12:34:56:78:9a:bc`).Be aware that Docker does not check if manually specified MAC addresses are unique.

Supported networks :

Network	Description
none	No networking in the container.
bridge (default)	Connect the container to the bridge via veth interfaces.
host	Use the host's network stack inside the container.
container:<name|id>	Use the network stack of another container, specified via its *name* or *id*.
NETWORK	Connects the container to a user created network (using `docker network create` command)

NETWORK: NONE

With the network is `none` a container will not have access to any external routes. The container will still have a `loopback` interface enabled in the container but it does not have any routes to external traffic.

NETWORK: BRIDGE

With the network set to `bridge` a container will use docker's default networking setup. A bridge is setup on the host, commonly named `docker0` , and a pair of `veth` interfaces will be created for the container. One side of the `veth` pair will remain on the host attached to the bridge while the other side of the pair will be placed inside the container's namespaces in addition to the `loopback` interface. An IP address will be allocated for containers on the bridge's network and traffic will be routed though this bridge to the container.

Containers can communicate via their IP addresses by default. To communicate by name, they must be linked.

NETWORK: HOST

With the network set to `host` a container will share the host's network stack and all interfaces from the host will be available to the container. The container's hostname will match the hostname on the host system. Note that `--mac-address` is invalid in `host` netmode. Even in `host` network mode a container has its own UTS namespace by default. As such `--hostname` is allowed in `host` network mode and will only change the hostname inside the container. Similar to `--hostname` , the `--add-host` , `--dns` , `--dns-search` , and `--dns-option` options can be used in `host` network mode. These options update `/etc/hosts` or `/etc/resolv.conf` inside the container. No change are made to `/etc/hosts` and `/etc/resolv.conf` on the host.

Compared to the default `bridge` mode, the `host` mode gives *significantly* better networking performance since it uses the host's native networking stack whereas the bridge has to go through one level of virtualization through the docker daemon. It is recommended to run containers in this mode when their networking performance is critical, for example, a production Load Balancer or a High Performance Web Server.

> Note: `--network="host"` gives the container full access to local system services such as D-bus and is therefore considered insecure.

NETWORK: CONTAINER

With the network set to `container` a container will share the network stack of another container. The other container's name must be provided in the format of `--network container:<name|id>` . Note that `--add-host` `--hostname` `--dns` `--dns-search` `--dns-option` and `--mac-address` are invalid in `container` netmode, and `--publish` `--publish-all` `--expose` are also invalid in `container` netmode.

Example running a Redis container with Redis binding to `localhost` then running the `redis-cli` command and connecting to the Redis server over the `localhost` interface.

```
$ docker run -d --name redis example/redis --bind 127.0.0.1
$ # use the redis container's network stack to access localhost
$ docker run --rm -it --network container:redis example/redis-cli -h 127.0.0.1
```

USER-DEFINED NETWORK

You can create a network using a Docker network driver or an external network driver plugin. You can connect multiple containers to the same network. Once connected to a user-defined network, the containers can communicate easily using 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.

The following example creates a network using the built-in `bridge` network driver and running a container in the created network

```
$ docker network create -d bridge my-net
$ docker run --network=my-net -itd --name=container3 busybox
```

Managing /etc/hosts

Your container will have lines in `/etc/hosts` which define the hostname of the container itself as well as `localhost` and a few other common things. The `--add-host` flag can be used to add additional lines to `/etc/hosts` .

```
$ docker run -it --add-host db-static:86.75.30.9 ubuntu cat /etc/hosts
172.17.0.22     09d03f76bf2c
fe00::0         ip6-localnet
ff00::0         ip6-mcastprefix
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters
127.0.0.1       localhost
::1                 localhost ip6-localhost ip6-loopback
86.75.30.9      db-static
```

If a container is connected to the default bridge network and `linked` with other containers, then the container's `/etc/hosts` file is updated with the linked container's name.

> Note Since Docker may live update the container's `/etc/hosts` file, there may be situations when processes inside the container can end up reading an empty or incomplete `/etc/hosts` file. In most cases, retrying the read again should fix the problem.

Restart policies (--restart)

Using the `--restart` flag on Docker run you can specify a restart policy for how a container should or should not be restarted on exit.

When a restart policy is active on a container, it will be shown as either `Up` or `Restarting` in `docker ps` (https://docs.docker.com/engine/reference/commandline/ps/). It can also be useful to use `docker events` (https://docs.docker.com/engine/reference/commandline/events/) to see the restart policy in effect.

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.
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.
unless-stopped	Always restart the container regardless of the exit status, including on daemon startup, except if the container was put into a stopped state before the Docker daemon was stopped.

An ever increasing delay (double the previous delay, starting at 100 milliseconds) is added before each restart to prevent flooding the server. This means the daemon will wait for 100 ms, then 200 ms, 400, 800, 1600, and so on until either the `on-failure` limit is hit, or when you `docker stop` or `docker rm -f` the container.

If a container is successfully restarted (the container is started and runs for at least 10 seconds), the delay is reset to its default value of 100 ms.

You can specify the maximum amount of times Docker will try to restart the container when using the on-failure policy. The default is that Docker will try forever to restart the container. The number of (attempted) restarts for a container can be obtained via `docker inspect` (https://docs.docker.com/engine/reference/commandline/inspect/). For example, to get the number of restarts for container “my-container”;

```
$ docker inspect -f "{{ .RestartCount }}" my-container
# 2
```

Or, to get the last time the container was (re)started;

```
$ docker inspect -f "{{ .State.StartedAt }}" my-container
# 2015-03-04T23:47:07.691840179Z
```

Combining `--restart` (restart policy) with the `--rm` (clean up) flag results in an error. On container restart, attached clients are disconnected. See the examples on using the `--rm` (clean up) (/engine/reference/run/#clean-up-rm) flag later in this page.

Examples

```
$ 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.

```
$ docker run --restart=on-failure:10 redis
```

This will run the `redis` container with a restart policy of on-failure and a maximum restart count of 10. If the `redis` container exits with a non-zero exit status more than 10 times in a row Docker will abort trying to restart the container. Providing a maximum restart limit is only valid for the on-failure policy.

Exit Status

The exit code from `docker run` gives information about why the container failed to run or why it exited. When `docker run` exits with a non-zero code, the exit codes follow the `chroot` standard, see below:

125 if the error is with Docker daemon *itself*

```
$ docker run --foo busybox; echo $?
# flag provided but not defined: --foo
  See 'docker run --help'.
  125
```

126 if the *contained command* cannot be invoked

```
$ docker run busybox /etc; echo $?
# docker: Error response from daemon: Container command '/etc' could not be invoked.
  126
```

127 if the *contained command* cannot be found

```
$ docker run busybox foo; echo $?
# docker: Error response from daemon: Container command 'foo' not found or does not exist.
  127
```

Exit code of *contained command* otherwise

```
$ docker run busybox /bin/sh -c 'exit 3'; echo $?
# 3
```

Clean up (--rm)

By default a container's file system persists even after the container exits. This makes debugging a lot easier (since you can inspect the final state) and you retain all your data by default. But if you are running short-term foreground processes, these container file systems can really pile up. If instead you'd like Docker to automatically clean up the container and remove the file system when the container exits, you can add the `--rm` flag:

```
--rm=false: Automatically remove the container when it exits
```

> Note: When you set the `--rm` flag, Docker also removes the anonymous volumes associated with the container when the container is removed. This is similar to running `docker rm -v my-container` . Only volumes that are specified without a name are removed. For example, with `docker run --rm -v /foo -v awesome:/bar busybox top` , the volume for `/foo` will be removed, but the volume for `/bar` will not. Volumes inherited via `--volumes-from` will be removed with the same logic -- if the original volume was specified with a name it will not be removed.

Security configuration

```
--security-opt="label=user:USER"     : Set the label user for the container
--security-opt="label=role:ROLE"     : Set the label role for the container
--security-opt="label=type:TYPE"     : Set the label type for the container
--security-opt="label=level:LEVEL"   : Set the label level for the container
--security-opt="label=disable"       : Turn off label confinement for the container
--security-opt="apparmor=PROFILE"    : Set the apparmor profile to be applied to the container
--security-opt="no-new-privileges:true|false"   : Disable/enable container processes from gaini
ng new privileges
--security-opt="seccomp=unconfined"  : Turn off seccomp confinement for the container
--security-opt="seccomp=profile.json": White listed syscalls seccomp Json file to be used as a
seccomp filter
```

You can override the default labeling scheme for each container by specifying the `--security-opt` flag. Specifying the level in the following command allows you to share the same content between containers.

```
$ docker run --security-opt label=level:s0:c100,c200 -it fedora bash
```

> Note: Automatic translation of MLS labels is not currently supported.

To disable the security labeling for this container versus running with the `--privileged` flag, use the following command:

```
$ docker run --security-opt label=disable -it fedora bash
```

If you want a tighter security policy on the processes within a container, you can specify an alternate type for the container. You could run a container that is only allowed to listen on Apache ports by executing the following command:

```
$ docker run --security-opt label=type:svirt_apache_t -it centos bash
```

> Note: You would have to write policy defining a `svirt_apache_t` type.

If you want to prevent your container processes from gaining additional privileges, you can execute the following command:

```
$ docker run --security-opt no-new-privileges -it centos bash
```

This means that commands that raise privileges such as `su` or `sudo` will no longer work. It also causes any seccomp filters to be applied later, after privileges have been dropped which may mean you can have a more restrictive set of filters. For more details, see the kernel documentation (https://www.kernel.org/doc/Documentation/prctl/no_new_privs.txt).

Specify an init process

You can use the `--init` flag to indicate that an init process should be used as the PID 1 in the container. Specifying an init process ensures the usual responsibilities of an init system, such as reaping zombie processes, are performed inside the created container.

The default init process used is the first `docker-init` executable found in the system path of the Docker daemon process. This `docker-init` binary, included in the default installation, is backed by tini (https://github.com/krallin/tini).

Specify custom cgroups

Using the `--cgroup-parent` flag, you can pass a specific cgroup to run a container in. This allows you to create and manage cgroups on their own. You can define custom resources for those cgroups and put containers under a common parent group.

Runtime constraints on resources

The operator can also adjust the performance parameters of the container:

Option	Description
`-m` , `--memory=""`	Memory limit (format: `<number>[<unit>]`). Number is a positive integer. Unit can be one of `b` , `k` , `m` , or `g` . Minimum is 4M.
`--memory-swap=""`	Total memory limit (memory + swap, format: `<number>[<unit>]`). Number is a positive integer. Unit can be one of `b` , `k` , `m` , or `g` .
`--memory-reservation=""`	Memory soft limit (format: `<number>[<unit>]`). Number is a positive integer. Unit can be one of `b` , `k` , `m` , or `g` .
`--kernel-memory=""`	Kernel memory limit (format: `<number>[<unit>]`). Number is a positive integer. Unit can be one of `b` , `k` , `m` , or `g` . Minimum is 4M.

Option	Description
`-c , --cpu-shares=0`	CPU shares (relative weight)
`--cpus=0.000`	Number of CPUs. Number is a fractional number. 0.000 means no limit.
`--cpu-period=0`	Limit the CPU CFS (Completely Fair Scheduler) period
`--cpuset-cpus=""`	CPUs in which to allow execution (0-3, 0,1)
`--cpuset-mems=""`	Memory nodes (MEMs) in which to allow execution (0-3, 0,1). Only effective on NUMA systems.
`--cpu-quota=0`	Limit the CPU CFS (Completely Fair Scheduler) quota
`--cpu-rt-period=0`	Limit the CPU real-time period. In microseconds. Requires parent cgroups be set and cannot be higher than parent. Also check rtprio ulimits.
`--cpu-rt-runtime=0`	Limit the CPU real-time runtime. In microseconds. Requires parent cgroups be set and cannot be higher than parent. Also check rtprio ulimits.
`--blkio-weight=0`	Block IO weight (relative weight) accepts a weight value between 10 and 1000.
`--blkio-weight-device=""`	Block IO weight (relative device weight, format: `DEVICE_NAME:WEIGHT`)
`--device-read-bps=""`	Limit read rate from a device (format: `<device-path>:<number>[<unit>]`). Number is a positive integer. Unit can be one of `kb` , `mb` , or `gb` .
`--device-write-bps=""`	Limit write rate to a device (format: `<device-path>:<number>[<unit>]`). Number is a positive integer. Unit can be one of `kb` , `mb` , or `gb` .
`--device-read-iops=""`	Limit read rate (IO per second) from a device (format: `<device-path>:<number>`). Number is a positive integer.
`--device-write-iops=""`	Limit write rate (IO per second) to a device (format: `<device-path>:<number>`). Number is a positive integer.
`--oom-kill-disable=false`	Whether to disable OOM Killer for the container or not.
`--oom-score-adj=0`	Tune container's OOM preferences (-1000 to 1000)
`--memory-swappiness=""`	Tune a container's memory swappiness behavior. Accepts an integer between 0 and 100.
`--shm-size=""`	Size of `/dev/shm` . The format is `<number><unit>` . `number` must be greater than `0` . Unit is optional and can be `b` (bytes), `k` (kilobytes), `m` (megabytes), or `g` (gigabytes). If you omit the unit, the system uses bytes. If you omit the size entirely, the system uses `64m` .

User memory constraints

We have four ways to set user memory usage:

Option	Result

Option	Result
memory=inf, memory-swap=inf (default)	There is no memory limit for the container. The container can use as much memory as needed.
memory=L<inf, memory-swap=inf	(specify memory and set memory-swap as `-1`) The container is not allowed to use more than L bytes of memory, but can use as much swap as is needed (if the host supports swap memory).
memory=L<inf, memory-swap=2*L	(specify memory without memory-swap) The container is not allowed to use more than L bytes of memory, swap *plus* memory usage is double of that.
memory=L<inf, memory-swap=S<inf, L<=S	(specify both memory and memory-swap) The container is not allowed to use more than L bytes of memory, swap *plus* memory usage is limited by S.

Examples:

```
$ docker run -it ubuntu:14.04 /bin/bash
```

We set nothing about memory, this means the processes in the container can use as much memory and swap memory as they need.

```
$ docker run -it -m 300M --memory-swap -1 ubuntu:14.04 /bin/bash
```

We set memory limit and disabled swap memory limit, this means the processes in the container can use 300M memory and as much swap memory as they need (if the host supports swap memory).

```
$ docker run -it -m 300M ubuntu:14.04 /bin/bash
```

We set memory limit only, this means the processes in the container can use 300M memory and 300M swap memory, by default, the total virtual memory size (--memory-swap) will be set as double of memory, in this case, memory + swap would be 2*300M, so processes can use 300M swap memory as well.

```
$ docker run -it -m 300M --memory-swap 1G ubuntu:14.04 /bin/bash
```

We set both memory and swap memory, so the processes in the container can use 300M memory and 700M swap memory.

Memory reservation is a kind of memory soft limit that allows for greater sharing of memory. Under normal circumstances, containers can use as much of the memory as needed and are constrained only by the hard limits set with the `-m` / `--memory` option. When memory reservation is set, Docker detects memory contention or low memory and forces containers to restrict their consumption to a reservation limit.

Always set the memory reservation value below the hard limit, otherwise the hard limit takes precedence. A reservation of 0 is the same as setting no reservation. By default (without reservation set), memory reservation is the same as the hard memory limit.

Memory reservation is a soft-limit feature and does not guarantee the limit won't be exceeded. Instead, the feature attempts to ensure that, when memory is heavily contended for, memory is allocated based on the reservation hints/setup.

The following example limits the memory (`-m`) to 500M and sets the memory reservation to 200M.

```
$ docker run -it -m 500M --memory-reservation 200M ubuntu:14.04 /bin/bash
```

Under this configuration, when the container consumes memory more than 200M and less than 500M, the next system memory reclaim attempts to shrink container memory below 200M.

The following example set memory reservation to 1G without a hard memory limit.

```
$ docker run -it --memory-reservation 1G ubuntu:14.04 /bin/bash
```

The container can use as much memory as it needs. The memory reservation setting ensures the container doesn't consume too much memory for long time, because every memory reclaim shrinks the container's consumption to the reservation.

By default, kernel kills processes in a container if an out-of-memory (OOM) error occurs. To change this behaviour, use the `--oom-kill-disable` option. Only disable the OOM killer on containers where you have also set the `-m/--memory` option. If the `-m` flag is not set, this can result in the host running out of memory and require killing the host's system processes to free memory.

The following example limits the memory to 100M and disables the OOM killer for this container:

```
$ docker run -it -m 100M --oom-kill-disable ubuntu:14.04 /bin/bash
```

The following example, illustrates a dangerous way to use the flag:

```
$ docker run -it --oom-kill-disable ubuntu:14.04 /bin/bash
```

The container has unlimited memory which can cause the host to run out memory and require killing system processes to free memory. The `--oom-score-adj` parameter can be changed to select the priority of which containers will be killed when the system is out of memory, with negative scores making them less likely to be killed, and positive scores more likely.

Kernel memory constraints

Kernel memory is fundamentally different than user memory as kernel memory can't be swapped out. The inability to swap makes it possible for the container to block system services by consuming too much kernel memory. Kernel memory includes：

- stack pages
- slab pages
- sockets memory pressure
- tcp memory pressure

You can setup kernel memory limit to constrain these kinds of memory. For example, every process consumes some stack pages. By limiting kernel memory, you can prevent new processes from being created when the kernel memory usage is too high.

Kernel memory is never completely independent of user memory. Instead, you limit kernel memory in the context of the user memory limit. Assume “U” is the user memory limit and “K” the kernel limit. There are three possible ways to set limits:

Option	Result
U != 0, K = inf (default)	This is the standard memory limitation mechanism already present before using kernel memory. Kernel memory is completely ignored.
U != 0, K < U	Kernel memory is a subset of the user memory. This setup is useful in deployments where the total amount of memory per-cgroup is overcommitted. Overcommitting kernel memory limits is definitely not recommended, since the box can still run out of non-reclaimable memory. In this case, you can configure K so that the sum of all groups is never greater than the total memory. Then, freely set U at the expense of the system's service quality.
U != 0, K > U	Since kernel memory charges are also fed to the user counter and reclamation is triggered for the container for both kinds of memory. This configuration gives the admin a unified view of memory. It is also useful for people who just want to track kernel memory usage.

Examples:

```
$ docker run -it -m 500M --kernel-memory 50M ubuntu:14.04 /bin/bash
```

We set memory and kernel memory, so the processes in the container can use 500M memory in total, in this 500M memory, it can be 50M kernel memory tops.

```
$ docker run -it --kernel-memory 50M ubuntu:14.04 /bin/bash
```

We set kernel memory without -m, so the processes in the container can use as much memory as they want, but they can only use 50M kernel memory.

Swappiness constraint

By default, a container’s kernel can swap out a percentage of anonymous pages. To set this percentage for a container, specify a `--memory-swappiness` value between 0 and 100. A value of 0 turns off anonymous page swapping. A value of 100 sets all anonymous pages as swappable. By default, if you are not using `--memory-swappiness`, memory swappiness value will be inherited from the parent.

For example, you can set:

```
$ docker run -it --memory-swappiness=0 ubuntu:14.04 /bin/bash
```

Setting the `--memory-swappiness` option is helpful when you want to retain the container’s working set and to avoid swapping performance penalties.

CPU share constraint

By default, all containers get the same proportion of CPU cycles. This proportion can be modified by changing the container's CPU share weighting relative to the weighting of all other running containers.

To modify the proportion from the default of 1024, use the `-c` or `--cpu-shares` flag to set the weighting to 2 or higher. If 0 is set, the system will ignore the value and use the default of 1024.

The proportion will only apply when CPU-intensive processes are running. When tasks in one container are idle, other containers can use the left-over CPU time. The actual amount of CPU time will vary depending on the number of containers running on the system.

For example, consider three containers, one has a cpu-share of 1024 and two others have a cpu-share setting of 512. When processes in all three containers attempt to use 100% of CPU, the first container would receive 50% of the total CPU time. If you add a fourth container with a cpu-share of 1024, the first container only gets 33% of the CPU. The remaining containers receive 16.5%, 16.5% and 33% of the CPU.

On a multi-core system, the shares of CPU time are distributed over all CPU cores. Even if a container is limited to less than 100% of CPU time, it can use 100% of each individual CPU core.

For example, consider a system with more than three cores. If you start one container `{C0}` with `-c=512` running one process, and another container `{C1}` with `-c=1024` running two processes, this can result in the following division of CPU shares:

```
PID    container        CPU     CPU share
100    {C0}             0       100% of CPU0
101    {C1}             1       100% of CPU1
102    {C1}             2       100% of CPU2
```

CPU period constraint

The default CPU CFS (Completely Fair Scheduler) period is 100ms. We can use `--cpu-period` to set the period of CPUs to limit the container's CPU usage. And usually `--cpu-period` should work with `--cpu-quota` .

Examples:

```
$ docker run -it --cpu-period=50000 --cpu-quota=25000 ubuntu:14.04 /bin/bash
```

If there is 1 CPU, this means the container can get 50% CPU worth of run-time every 50ms.

In addition to use `--cpu-period` and `--cpu-quota` for setting CPU period constraints, it is possible to specify `--cpus` with a float number to achieve the same purpose. For example, if there is 1 CPU, then `--cpus=0.5` will achieve the same result as setting `--cpu-period=50000` and `--cpu-quota=25000` (50% CPU).

The default value for `--cpus` is `0.000` , which means there is no limit.

For more information, see the CFS documentation on bandwidth limiting (https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt).

Cpuset constraint

We can set cpus in which to allow execution for containers.

Examples:

```
$ docker run -it --cpuset-cpus="1,3" ubuntu:14.04 /bin/bash
```

This means processes in container can be executed on cpu 1 and cpu 3.

```
$ docker run -it --cpuset-cpus="0-2" ubuntu:14.04 /bin/bash
```

This means processes in container can be executed on cpu 0, cpu 1 and cpu 2.

We can set mems in which to allow execution for containers. Only effective on NUMA systems.

Examples:

```
$ docker run -it --cpuset-mems="1,3" ubuntu:14.04 /bin/bash
```

This example restricts the processes in the container to only use memory from memory nodes 1 and 3.

```
$ docker run -it --cpuset-mems="0-2" ubuntu:14.04 /bin/bash
```

This example restricts the processes in the container to only use memory from memory nodes 0, 1 and 2.

CPU quota constraint

The `--cpu-quota` flag limits the container's CPU usage. The default 0 value allows the container to take 100% of a CPU resource (1 CPU). The CFS (Completely Fair Scheduler) handles resource allocation for executing processes and is default Linux Scheduler used by the kernel. Set this value to 50000 to limit the container to 50% of a CPU resource. For multiple CPUs, adjust the `--cpu-quota` as necessary. For more information, see the CFS documentation on bandwidth limiting (https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt).

Block IO bandwidth (Blkio) constraint

By default, all containers get the same proportion of block IO bandwidth (blkio). This proportion is 500. To modify this proportion, change the container's blkio weight relative to the weighting of all other running containers using the `--blkio-weight` flag.

> Note: The blkio weight setting is only available for direct IO. Buffered IO is not currently supported.

The `--blkio-weight` flag can set the weighting to a value between 10 to 1000. For example, the commands below create two containers with different blkio weight:

```
$ docker run -it --name c1 --blkio-weight 300 ubuntu:14.04 /bin/bash
$ docker run -it --name c2 --blkio-weight 600 ubuntu:14.04 /bin/bash
```

If you do block IO in the two containers at the same time, by, for example:

```
$ time dd if=/mnt/zerofile of=test.out bs=1M count=1024 oflag=direct
```

You'll find that the proportion of time is the same as the proportion of blkio weights of the two containers.

The `--blkio-weight-device="DEVICE_NAME:WEIGHT"` flag sets a specific device weight. The `DEVICE_NAME:WEIGHT` is a string containing a colon-separated device name and weight. For example, to set `/dev/sda` device weight to `200`:

```
$ docker run -it \
    --blkio-weight-device "/dev/sda:200" \
    ubuntu
```

If you specify both the `--blkio-weight` and `--blkio-weight-device`, Docker uses the `--blkio-weight` as the default weight and uses `--blkio-weight-device` to override this default with a new value on a specific device. The following example uses a default weight of `300` and overrides this default on `/dev/sda` setting that weight to `200`:

```
$ docker run -it \
    --blkio-weight 300 \
    --blkio-weight-device "/dev/sda:200" \
    ubuntu
```

The `--device-read-bps` flag limits the read rate (bytes per second) from a device. For example, this command creates a container and limits the read rate to `1mb` per second from `/dev/sda`:

```
$ docker run -it --device-read-bps /dev/sda:1mb ubuntu
```

The `--device-write-bps` flag limits the write rate (bytes per second) to a device. For example, this command creates a container and limits the write rate to `1mb` per second for `/dev/sda`:

```
$ docker run -it --device-write-bps /dev/sda:1mb ubuntu
```

Both flags take limits in the `<device-path>:<limit>[unit]` format. Both read and write rates must be a positive integer. You can specify the rate in `kb` (kilobytes), `mb` (megabytes), or `gb` (gigabytes).

The `--device-read-iops` flag limits read rate (IO per second) from a device. For example, this command creates a container and limits the read rate to `1000` IO per second from `/dev/sda`:

```
$ docker run -ti --device-read-iops /dev/sda:1000 ubuntu
```

The `--device-write-iops` flag limits write rate (IO per second) to a device. For example, this command creates a container and limits the write rate to `1000` IO per second to `/dev/sda`:

```
$ docker run -ti --device-write-iops /dev/sda:1000 ubuntu
```

Both flags take limits in the `<device-path>:<limit>` format. Both read and write rates must be a positive integer.

Additional groups

```
--group-add: Add additional groups to run as
```

By default, the docker container process runs with the supplementary groups looked up for the specified user. If one wants to add more to that list of groups, then one can use this flag:

```
$ docker run --rm --group-add audio --group-add nogroup --group-add 777 busybox id
uid=0(root) gid=0(root) groups=10(wheel),29(audio),99(nogroup),777
```

Runtime privilege and Linux capabilities

```
--cap-add: Add Linux capabilities
--cap-drop: Drop Linux capabilities
--privileged=false: Give extended privileges to this container
--device=[]: Allows you to run devices inside the container without the --privileged flag.
```

By default, Docker containers are “unprivileged” and cannot, for example, run a Docker daemon inside a Docker container. This is because by default a container is not allowed to access any devices, but a “privileged” container is given access to all devices (see the documentation on cgroups devices (https://www.kernel.org/doc/Documentation/cgroup-v1/devices.txt)).

When the operator executes `docker run --privileged`, Docker will enable access to all devices on the host as well as set some configuration in AppArmor or SELinux to allow the container nearly all the same access to the host as processes running outside containers on the host. Additional information about running with `--privileged` is available on the Docker Blog (http://blog.docker.com/2013/09/docker-can-now-run-within-docker/).

If you want to limit access to a specific device or devices you can use the `--device` flag. It allows you to specify one or more devices that will be accessible within the container.

```
$ docker run --device=/dev/snd:/dev/snd ...
```

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:w --rm -it ubuntu fdisk  /dev/xvdc
    crash....

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

In addition to `--privileged`, the operator can have fine grain control over the capabilities using `--cap-add` and `--cap-drop`. By default, Docker has a default list of capabilities that are kept. The following table lists the Linux capability options which are allowed by default and can be dropped.

Capability Key	Capability Description
SETPCAP	Modify process capabilities.
MKNOD	Create special files using mknod(2).
AUDIT_WRITE	Write records to kernel auditing log.
CHOWN	Make arbitrary changes to file UIDs and GIDs (see chown(2)).
NET_RAW	Use RAW and PACKET sockets.
DAC_OVERRIDE	Bypass file read, write, and execute permission checks.
FOWNER	Bypass permission checks on operations that normally require the file system UID of the process to match the UID of the file.
FSETID	Don’t clear set-user-ID and set-group-ID permission bits when a file is modified.
KILL	Bypass permission checks for sending signals.
SETGID	Make arbitrary manipulations of process GIDs and supplementary GID list.
SETUID	Make arbitrary manipulations of process UIDs.
NET_BIND_SERVICE	Bind a socket to internet domain privileged ports (port numbers less than 1024).
SYS_CHROOT	Use chroot(2), change root directory.
SETFCAP	Set file capabilities.

The next table shows the capabilities which are not granted by default and may be added.

Capability Key	Capability Description
SYS_MODULE	Load and unload kernel modules.

Capability Key	Capability Description
SYS_RAWIO	Perform I/O port operations (iopl(2) and ioperm(2)).
SYS_PACCT	Use acct(2), switch process accounting on or off.
SYS_ADMIN	Perform a range of system administration operations.
SYS_NICE	Raise process nice value (nice(2), setpriority(2)) and change the nice value for arbitrary processes.
SYS_RESOURCE	Override resource Limits.
SYS_TIME	Set system clock (settimeofday(2), stime(2), adjtimex(2)); set real-time (hardware) clock.
SYS_TTY_CONFIG	Use vhangup(2); employ various privileged ioctl(2) operations on virtual terminals.
AUDIT_CONTROL	Enable and disable kernel auditing; change auditing filter rules; retrieve auditing status and filtering rules.
MAC_ADMIN	Allow MAC configuration or state changes. Implemented for the Smack LSM.
MAC_OVERRIDE	Override Mandatory Access Control (MAC). Implemented for the Smack Linux Security Module (LSM).
NET_ADMIN	Perform various network-related operations.
SYSLOG	Perform privileged syslog(2) operations.
DAC_READ_SEARCH	Bypass file read permission checks and directory read and execute permission checks.
LINUX_IMMUTABLE	Set the FS_APPEND_FL and FS_IMMUTABLE_FL i-node flags.
NET_BROADCAST	Make socket broadcasts, and listen to multicasts.
IPC_LOCK	Lock memory (mlock(2), mlockall(2), mmap(2), shmctl(2)).
IPC_OWNER	Bypass permission checks for operations on System V IPC objects.
SYS_PTRACE	Trace arbitrary processes using ptrace(2).
SYS_BOOT	Use reboot(2) and kexec_load(2), reboot and load a new kernel for later execution.
LEASE	Establish leases on arbitrary files (see fcntl(2)).
WAKE_ALARM	Trigger something that will wake up the system.
BLOCK_SUSPEND	Employ features that can block system suspend.

Further reference information is available on the capabilities(7) - Linux man page (http://man7.org/linux/man-pages/man7/capabilities.7.html)

Both flags support the value `ALL`, so if the operator wants to have all capabilities but `MKNOD` they could use:

```
$ docker run --cap-add=ALL --cap-drop=MKNOD ...
```

For interacting with the network stack, instead of using `--privileged` they should use `--cap-add=NET_ADMIN` to modify the network interfaces.

```
$ docker run -it --rm  ubuntu:14.04 ip link add dummy0 type dummy
RTNETLINK answers: Operation not permitted
$ docker run -it --rm --cap-add=NET_ADMIN ubuntu:14.04 ip link add dummy0 type dummy
```

To mount a FUSE based filesystem, you need to combine both `--cap-add` and `--device`:

```
$ docker run --rm -it --cap-add SYS_ADMIN sshfs sshfs sven@10.10.10.20:/home/sven /mnt
fuse: failed to open /dev/fuse: Operation not permitted
$ docker run --rm -it --device /dev/fuse sshfs sshfs sven@10.10.10.20:/home/sven /mnt
fusermount: mount failed: Operation not permitted
$ docker run --rm -it --cap-add SYS_ADMIN --device /dev/fuse sshfs
# sshfs sven@10.10.10.20:/home/sven /mnt
The authenticity of host '10.10.10.20 (10.10.10.20)' can't be established.
ECDSA key fingerprint is 25:34:85:75:25:b0:17:46:05:19:04:93:b5:dd:5f:c6.
Are you sure you want to continue connecting (yes/no)? yes
sven@10.10.10.20's password:
root@30aa0cfaf1b5:/# ls -la /mnt/src/docker
total 1516
drwxrwxr-x 1 1000 1000   4096 Dec  4 06:08 .
drwxrwxr-x 1 1000 1000   4096 Dec  4 11:46 ..
-rw-rw-r-- 1 1000 1000     16 Oct  8 00:09 .dockerignore
-rwxrwxr-x 1 1000 1000    464 Oct  8 00:09 .drone.yml
drwxrwxr-x 1 1000 1000   4096 Dec  4 06:11 .git
-rw-rw-r-- 1 1000 1000    461 Dec  4 06:08 .gitignore
....
```

The default seccomp profile will adjust to the selected capabilities, in order to allow use of facilities allowed by the capabilities, so you should not have to adjust this, since Docker 1.12. In Docker 1.10 and 1.11 this did not happen and it may be necessary to use a custom seccomp profile or use `--security-opt seccomp=unconfined` when adding capabilities.

Logging drivers (--log-driver)

The container can have a different logging driver than the Docker daemon. Use the `--log-driver=VALUE` with the `docker run` command to configure the container's logging driver. The following options are supported:

Driver	Description
`none`	Disables any logging for the container. `docker logs` won't be available with this driver.
`json-file`	Default logging driver for Docker. Writes JSON messages to file. No logging options are supported for this driver.
`syslog`	Syslog logging driver for Docker. Writes log messages to syslog.
`journald`	Journald logging driver for Docker. Writes log messages to `journald`.

Driver	Description
`gelf`	Graylog Extended Log Format (GELF) logging driver for Docker. Writes log messages to a GELF endpoint likeGraylog or Logstash.
`fluentd`	Fluentd logging driver for Docker. Writes log messages to `fluentd` (forward input).
`awslogs`	Amazon CloudWatch Logs logging driver for Docker. Writes log messages to Amazon CloudWatch Logs
`splunk`	Splunk logging driver for Docker. Writes log messages to `splunk` using Event Http Collector.

The `docker logs` command is available only for the `json-file` and `journald` logging drivers. For detailed information on working with logging drivers, see Configure logging drivers (https://docs.docker.com/config/containers/logging/configure/).

Overriding Dockerfile image defaults

When a developer builds an image from a *Dockerfile* (https://docs.docker.com/engine/reference/builder/) or when she commits it, the developer can set a number of default parameters that take effect when the image starts up as a container.

Four of the Dockerfile commands cannot be overridden at runtime: `FROM` , `MAINTAINER` , `RUN` , and `ADD` . Everything else has a corresponding override in `docker run` . We'll go through what the developer might have set in each Dockerfile instruction and how the operator can override that setting.

- CMD (Default Command or Options) (/engine/reference/run/#cmd-default-command-or-options)
- ENTRYPOINT (Default Command to Execute at Runtime) (/engine/reference/run/#entrypoint-default-command-to-execute-at-runtime)
- EXPOSE (Incoming Ports) (/engine/reference/run/#expose-incoming-ports)
- ENV (Environment Variables) (/engine/reference/run/#env-environment-variables)
- HEALTHCHECK (/engine/reference/run/#healthcheck)
- VOLUME (Shared Filesystems) (/engine/reference/run/#volume-shared-filesystems)
- USER (/engine/reference/run/#user)
- WORKDIR (/engine/reference/run/#workdir)

CMD (default command or options)

Recall the optional `COMMAND` in the Docker commandline:

```
$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

This command is optional because the person who created the `IMAGE` may have already provided a default `COMMAND` using the Dockerfile `CMD` instruction. As the operator (the person running a container from the image), you can override that `CMD` instruction just by specifying a new `COMMAND` .

If the image also specifies an `ENTRYPOINT` then the `CMD` or `COMMAND` get appended as arguments to the `ENTRYPOINT` .

ENTRYPOINT (default command to execute at runtime)

```
--entrypoint="": Overwrite the default entrypoint set by the image
```

The `ENTRYPOINT` of an image is similar to a `COMMAND` because it specifies what executable to run when the container starts, but it is (purposely) more difficult to override. The `ENTRYPOINT` gives a container its default nature or behavior, so that when you set an `ENTRYPOINT` you can run the container *as if it were that binary*, complete with default options, and you can pass in more options via the `COMMAND` . But, sometimes an operator may want to run something else inside the container, so you can override the default `ENTRYPOINT` at runtime by using a string to specify the new `ENTRYPOINT` . Here is an example of how to run a shell in a container that has been set up to automatically run something else (like `/usr/bin/redis-server`):

```
$ docker run -it --entrypoint /bin/bash example/redis
```

or two examples of how to pass more parameters to that ENTRYPOINT:

```
$ docker run -it --entrypoint /bin/bash example/redis -c ls -l
$ docker run -it --entrypoint /usr/bin/redis-cli example/redis --help
```

You can reset a containers entrypoint by passing an empty string, for example:

```
$ docker run -it --entrypoint="" mysql bash
```

> Note: Passing `--entrypoint` will clear out any default command set on the image (i.e. any `CMD` instruction in the Dockerfile used to build it).

EXPOSE (incoming ports)

The following `run` command options work with container networking:

```
--expose=[]: Expose a port or a range of ports inside the container.
             These are additional to those exposed by the ‘EXPOSE‘ instruction
-P         : Publish all exposed ports to the host interfaces
-p=[]      : Publish a container’s port or a range of ports to the host
               format: ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort |
 containerPort
               Both hostPort and containerPort can be specified as a
               range of ports. When specifying ranges for both, the
               number of container ports in the range must match the
               number of host ports in the range, for example:
                   -p 1234-1236:1234-1236/tcp

               When specifying a range for hostPort only, the
               containerPort must not be a range.  In this case the
               container port is published somewhere within the
               specified hostPort range. (e.g., ‘-p 1234-1236:1234/tcp‘)

               (use ’docker port’ to see the actual mapping)

--link=""  : Add link to another container (<name or id>:alias or <name or id>)
```

With the exception of the `EXPOSE` directive, an image developer hasn't got much control over networking. The `EXPOSE` instruction defines the initial incoming ports that provide services. These ports are available to processes inside the container. An operator can use the `--expose` option to add to the exposed ports.

To expose a container's internal port, an operator can start the container with the `-P` or `-p` flag. The exposed port is accessible on the host and the ports are available to any client that can reach the host.

The `-P` option publishes all the ports to the host interfaces. Docker binds each exposed port to a random port on the host. The range of ports are within an *ephemeral port range* defined by `/proc/sys/net/ipv4/ip_local_port_range`. Use the `-p` flag to explicitly map a single port or range of ports.

The port number inside the container (where the service listens) does not need to match the port number exposed on the outside of the container (where clients connect). For example, inside the container an HTTP service is listening on port 80 (and so the image developer specifies `EXPOSE 80` in the Dockerfile). At runtime, the port might be bound to 42800 on the host. To find the mapping between the host ports and the exposed ports, use `docker port`.

If the operator uses `--link` when starting a new client container in the default bridge network, then the client container can access the exposed port via a private networking interface. If `--link` is used when starting a container in a user-defined network as described in *Networking overview* (https://docs.docker.com/network/), it will provide a named alias for the container being linked to.

ENV (environment variables)

Docker automatically sets some environment variables when creating a Linux container. Docker does not set any environment variables when creating a Windows container.

The following environment variables are set for Linux containers:

Variable	Value
`HOME`	Set based on the value of `USER`
`HOSTNAME`	The hostname associated with the container
`PATH`	Includes popular directories, such as `/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin`
`TERM`	`xterm` if the container is allocated a pseudo-TTY

Additionally, the operator can set any environment variable in the container by using one or more `-e` flags, even overriding those mentioned above, or already defined by the developer with a Dockerfile `ENV`. If the operator names an environment variable without specifying a value, then the current value of the named variable is propagated into the container's environment:

```
$ export today=Wednesday
$ docker run -e "deep=purple" -e today --rm alpine env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=d2219b854598
deep=purple
today=Wednesday
HOME=/root
```

```
PS C:\> docker run --rm -e "foo=bar" microsoft/nanoserver cmd /s /c set
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\ContainerAdministrator\AppData\Roaming
CommonProgramFiles=C:\Program Files\Common Files
CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
CommonProgramW6432=C:\Program Files\Common Files
COMPUTERNAME=C2FAEFCC8253
ComSpec=C:\Windows\system32\cmd.exe
foo=bar
LOCALAPPDATA=C:\Users\ContainerAdministrator\AppData\Local
NUMBER_OF_PROCESSORS=8
OS=Windows_NT
Path=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerSh
ell\v1.0\;C:\Users\ContainerAdministrator\AppData\Local\Microsoft\WindowsApps
PATHEXT=.COM;.EXE;.BAT;.CMD
PROCESSOR_ARCHITECTURE=AMD64
PROCESSOR_IDENTIFIER=Intel64 Family 6 Model 62 Stepping 4, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=3e04
ProgramData=C:\ProgramData
ProgramFiles=C:\Program Files
ProgramFiles(x86)=C:\Program Files (x86)
ProgramW6432=C:\Program Files
PROMPT=$P$G
PUBLIC=C:\Users\Public
SystemDrive=C:
SystemRoot=C:\Windows
TEMP=C:\Users\ContainerAdministrator\AppData\Local\Temp
TMP=C:\Users\ContainerAdministrator\AppData\Local\Temp
USERDOMAIN=User Manager
USERNAME=ContainerAdministrator
USERPROFILE=C:\Users\ContainerAdministrator
windir=C:\Windows
```

Similarly the operator can set the HOSTNAME (Linux) or COMPUTERNAME (Windows) with `-h`.

HEALTHCHECK

```
  --health-cmd            Command to run to check health
  --health-interval       Time between running the check
  --health-retries        Consecutive failures needed to report unhealthy
  --health-timeout        Maximum time to allow one check to run
  --health-start-period   Start period for the container to initialize before starting health-r
etries countdown
  --no-healthcheck        Disable any container-specified HEALTHCHECK
```

Example:

```
$ docker run --name=test -d \
    --health-cmd='stat /etc/passwd || exit 1' \
    --health-interval=2s \
    busybox sleep 1d
$ sleep 2; docker inspect --format='{{.State.Health.Status}}' test
healthy
$ docker exec test rm /etc/passwd
$ sleep 2; docker inspect --format='{{json .State.Health}}' test
{
  "Status": "unhealthy",
  "FailingStreak": 3,
  "Log": [
    {
      "Start": "2016-05-25T17:22:04.635478668Z",
      "End": "2016-05-25T17:22:04.7272552Z",
      "ExitCode": 0,
      "Output": "  File: /etc/passwd\n  Size: 334       \tBlocks: 8          IO Block: 4096   r
egular file\nDevice: 32h/50d\tInode: 12          Links: 1\nAccess: (0664/-rw-rw-r--)  Uid: (
 0/    root)   Gid: (    0/    root)\nAccess: 2015-12-05 22:05:32.000000000\nModify: 2015..."
    },
    {
      "Start": "2016-05-25T17:22:06.732900633Z",
      "End": "2016-05-25T17:22:06.822168935Z",
      "ExitCode": 0,
      "Output": "  File: /etc/passwd\n  Size: 334       \tBlocks: 8          IO Block: 4096   r
egular file\nDevice: 32h/50d\tInode: 12          Links: 1\nAccess: (0664/-rw-rw-r--)  Uid: (
 0/    root)   Gid: (    0/    root)\nAccess: 2015-12-05 22:05:32.000000000\nModify: 2015..."
    },
    {
      "Start": "2016-05-25T17:22:08.823956535Z",
      "End": "2016-05-25T17:22:08.897359124Z",
      "ExitCode": 1,
      "Output": "stat: can't stat '/etc/passwd': No such file or directory\n"
    },
    {
      "Start": "2016-05-25T17:22:10.898802931Z",
      "End": "2016-05-25T17:22:10.969631866Z",
      "ExitCode": 1,
      "Output": "stat: can't stat '/etc/passwd': No such file or directory\n"
    },
    {
      "Start": "2016-05-25T17:22:12.971033523Z",
      "End": "2016-05-25T17:22:13.082015516Z",
      "ExitCode": 1,
      "Output": "stat: can't stat '/etc/passwd': No such file or directory\n"
    }
  ]
}
```

The health status is also displayed in the `docker ps` output.

TMPFS (mount tmpfs filesystems)

```
--tmpfs=[]: Create a tmpfs mount with: container-dir[:<options>],
            where the options are identical to the Linux
            'mount -t tmpfs -o' command.
```

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

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

VOLUME (shared filesystems)

```
-v, --volume=[host-src:]container-dest[:<options>]: Bind mount a volume.
The comma-delimited 'options' are [rw|ro], [z|Z],
[[r]shared|[r]slave|[r]private], and [nocopy].
The 'host-src' is an absolute path or a name value.

If neither 'rw' or 'ro' is specified then the volume is mounted in
read-write mode.

The 'nocopy' mode is used to disable automatically copying the requested volume
path in the container to the volume storage location.
For named volumes, 'copy' is the default mode. Copy modes are not supported
for bind-mounted volumes.

--volumes-from="": Mount all volumes from the given container(s)
```

> Note: When using systemd to manage the Docker daemon's start and stop, in the systemd unit file there is an option to control mount propagation for the Docker daemon itself, called `MountFlags`. The value of this setting may cause Docker to not see mount propagation changes made on the mount point. For example, if this value is `slave`, you may not be able to use the `shared` or `rshared` propagation on a volume.

The volumes commands are complex enough to have their own documentation in section *Use volumes* (https://docs.docker.com/storage/volumes/). A developer can define one or more `VOLUME`'s associated with an image, but only the operator can give access from one container to another (or from a container to a volume mounted on the host).

The `container-dest` must always be an absolute path such as `/src/docs`. The `host-src` can either be an absolute path or a `name` value. If you supply an absolute path for the `host-dir`, Docker bind-mounts to the path you specify. If you supply a `name`, Docker creates a named volume by that `name`.

A `name` value must start with an alphanumeric character, followed by `a-z0-9`, `_` (underscore), `.` (period) or `-` (hyphen). An absolute path starts with a `/` (forward slash).

For example, you can specify either `/foo` or `foo` for a `host-src` value. If you supply the `/foo` value, Docker creates a bind mount. If you supply the `foo` specification, Docker creates a named volume.

USER

`root` (id = 0) is the default user within a container. The image developer can create additional users. Those users are accessible by name. When passing a numeric ID, the user does not have to exist in the container.

The developer can set a default user to run the first process with the Dockerfile `USER` instruction. When starting a container, the operator can override the `USER` instruction by passing the `-u` option.

```
-u="", --user="": Sets the username or UID used and optionally the groupname or GID for the spe
cified command.

The followings examples are all valid:
--user=[ user | user:group | uid | uid:gid | user:gid | uid:group ]
```

> Note: if you pass a numeric uid, it must be in the range of 0-2147483647.

WORKDIR

The default working directory for running binaries within a container is the root directory (`/`), but the developer can set a different default with the Dockerfile `WORKDIR` command. The operator can override this with:

```
-w="": Working directory inside the container
```

docker (https://docs.docker.com/glossary/?term=docker), run (https://docs.docker.com/glossary/?term=run), configure (https://docs.docker.com/glossary/?term=configure), runtime (https://docs.docker.com/glossary/?term=runtime)

docker port

Estimated reading time: 1 minute

Description

List port mappings or a specific mapping for the container

Usage

```
docker port CONTAINER [PRIVATE_PORT[/PROTO]]
```

Parent command

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

Examples

Show all mapped ports

You can find out all the ports mapped by not specifying a `PRIVATE_PORT`, or just a specific mapping:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
            STATUS              PORTS
             NAMES
b650456536c7        busybox:latest      top                 54 minut
es ago      Up 54 minutes       0.0.0.0:1234->9876/tcp, 0.0.0.0:4321
->7890/tcp   test
$ docker port test
7890/tcp -> 0.0.0.0:4321
9876/tcp -> 0.0.0.0:1234
$ docker port test 7890/tcp
0.0.0.0:4321
$ docker port test 7890/udp
2014/06/24 11:53:36 Error: No public port '7890/udp' published for t
est
$ docker port test 7890
0.0.0.0:4321
```

UNDERSTANDING DOCKER NETWORKING DRIVERS AND THEIR USE CASES

By Mark Church [https://blog.docker.com/author/mark-church/] *December 19, 2016*

docker networking (https://blog.docker.com/tag/docker-networking/), libnetwork. network drivers (https://blog.docker.com/tag/libnetwork-network-drivers/), load balancing (https://blog.docker.com/tag/load-balancing/), service discovery (https://blog.docker.com/tag/service-discovery/), ucp (https://blog.docker.com/tag/ucp/), universal control plane (https://blog.docker.com/tag/universal-control-plane/)

Applications requirements and networking environments are diverse and sometimes opposing forces. In between applications and the network sits Docker networking, affectionately called the Container Network Model (https://github.com/docker/libnetwork/blob/master/docs/design.md) or CNM. It's CNM that brokers connectivity for your Docker containers and also what abstracts away the diversity and complexity so common in networking. The result is portability and it comes from CNM's powerful **network drivers**. These are pluggable interfaces for the Docker Engine, Swarm, and UCP that provide special capabilities like multi-host networking, network layer encryption, and service discovery.

Naturally, the next question is ***which network driver should I use***? Each driver offers tradeoffs and has different advantages depending on the use case. There are **built-in** network drivers that come included with Docker Engine and there are also **plug-in** network drivers offered by networking vendors and the community. The most commonly used built-in network drivers are **bridge**, **overlay** and **macvlan.** Together they cover a very broad list of networking use cases and environments. For a more in depth comparison and discussion of even more network drivers, check out the (https://success.docker.com/Datacenter/Apply/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docker_Container_Networks)Docker Network Reference Architecture. (https://success.docker.com/Datacenter/Apply/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docker_Container_Networks)

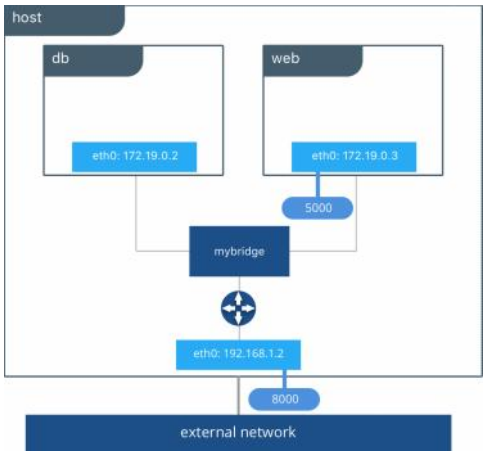
Bridge Network Driver

The `bridge` networking driver is the first driver on our list. It's simple to understand, simple to use, and simple to troubleshoot, which makes it a good networking choice for developers and those new to Docker. The `bridge` driver creates a private network internal to the host so containers on this network can communicate. External access is granted by exposing ports to containers. Docker secures the network by managing rules that block connectivity between different Docker networks.

Behind the scenes, the Docker Engine creates the necessary Linux bridges, internal interfaces, iptables rules, and host routes to make this connectivity possible. In the example highlighted below, a Docker bridge network is created and two containers are attached to it. With no extra configuration the Docker Engine does the necessary wiring, provides service discovery for the containers, and configures security rules to prevent communication to other networks. A built-in IPAM driver provides the container interfaces with private IP addresses from the subnet of the bridge network.

In the following examples, we use a fictitious app called `pets` comprised of a `web` and `db` container. Feel free to try it out on your own UCP or Swarm cluster. Your app will be accessible on `` `<host-ip>:8000`. ``

```
docker network create -d bridge mybridge
docker run -d --net mybridge --name db redis
docker run -d --net mybridge -e DB=db -p 8000:5000 --name web chrch/web
```



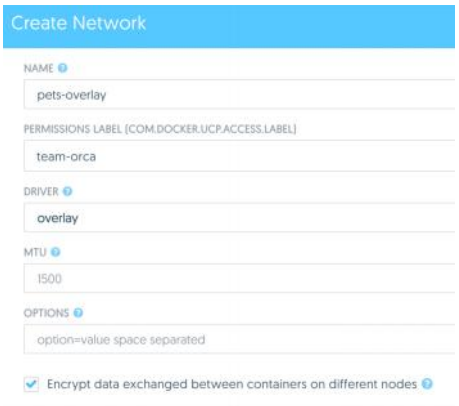
Our application is now being served on our host at port 8000. The Docker bridge is allowing `web` to communicate with `db` by its container name. The bridge driver does the service discovery for us automatically because they are on the same network. All of the port mappings, security rules, and pipework between Linux bridges is handled for us by the networking driver as containers are scheduled and rescheduled across a cluster.

The bridge driver is a **local scope** driver, which means it only provides service discovery, IPAM, and connectivity on a single host. Multi-host service discovery requires an external solution that can map containers to their host location. This is exactly what makes the `overlay` driver so great.

Overlay Network Driver

The built-in Docker `overlay` network driver radically simplifies many of the complexities in multi-host networking. It is a **swarm scope** driver, which means that it operates across an entire Swarm or UCP cluster rather than individual hosts. With the `overlay` driver, multi-host networks are first-class citizens inside Docker without external provisioning or components. IPAM, service discovery, multi-host connectivity, encryption, and load balancing are built right in. For control, the `overlay` driver uses the encrypted Swarm control plane to manage large scale clusters at low convergence times.

The `overlay` driver utilizes an industry-standard VXLAN data plane that decouples the container network from the underlying physical network (the *underlay*). This has the advantage of providing maximum portability across various cloud and on-premises networks. Network policy, visibility, and security is controlled centrally through the Docker Universal Control Plane (UCP).



In this example we create an overlay network in UCP so we can connect our `web` and `db` containers when they are living on different hosts. Native DNS-based service discovery for services & containers within an overlay network will ensure that `web` can resolve to `db` and vice-versa. We turned on encryption so that communication between our containers is secure by default. Furthermore, visibility and use of the network in UCP is restricted by the permissions label we use.

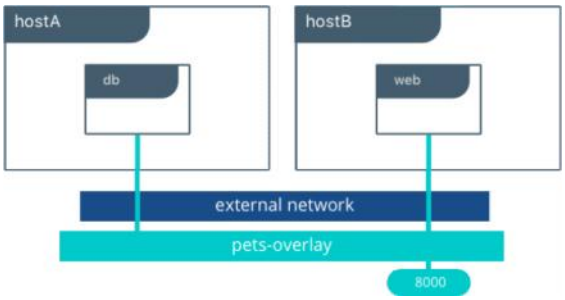
UCP will schedule services across the cluster and UCP will dynamically program the overlay network to provide connectivity to the containers wherever they are. When services are backed by multiple containers, VIP-based load balancing will distribute traffic across all of the containers.

Feel free to run this example against your UCP cluster with the following CLI commands:

```
docker network create -d overlay --opt encrypted pets-overlay

docker service create --network pets-overlay --name db redis

docker service create --network pets-overlay -p 8000:5000 -e DB=db --name web chrch/web
```



In this example we are still serving our web app on port 8000 but now we have deployed our application across different hosts. If we wanted to scale our `web` containers, Swarm & UCP networking would load balance the traffic for us automatically.

The `overlay` driver is a feature-rich driver that handles much of the complexity and integration that organizations struggle with when crafting piecemeal solutions. It provides an out-of-the-box solution for many networking challenges and does so at scale.

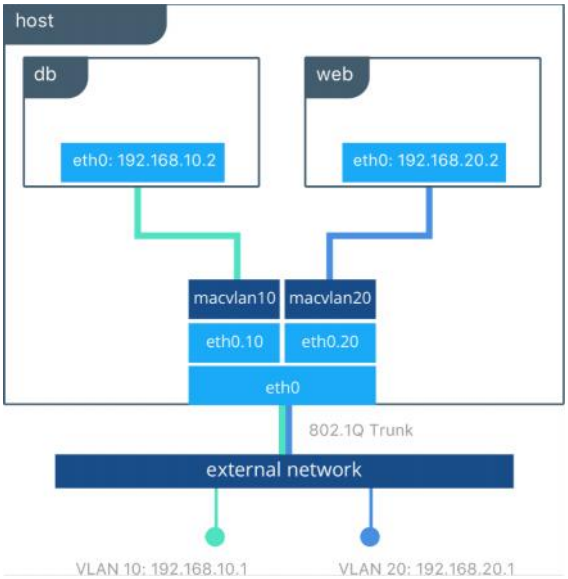
MACVLAN Driver

The `macvlan` driver is the newest built-in network driver and offers several unique characteristics. It's a very lightweight driver, because rather than using any Linux bridging or port mapping, it connects container interfaces directly to host interfaces. Containers are addressed with routable IP addresses that are on the subnet of the external network.

As a result of routable IP addresses, containers communicate directly with resources that exist outside a Swarm cluster without the use of NAT and port mapping. This can aid in network visibility and troubleshooting. Additionally, the direct traffic path between containers and the host interface helps reduce latency. `macvlan` is a local scope network driver which is configured per-host. As a result, there are stricter dependencies between MACVLAN and external networks, which is both a constraint and an advantage that is different from `overlay` or `bridge`.

The `macvlan` driver uses the concept of a parent interface. This interface can be a host interface such as `eth0`, a sub-interface, or even a bonded host adaptor which bundles Ethernet interfaces into a single logical interface. A gateway address from the external network is required during MACVLAN network configuration, as a MACVLAN network is a L2 segment from the container to the network gateway. Like all Docker networks, MACVLAN networks are segmented from each other – providing access within a network, but not between networks.

The `macvlan` driver can be configured in different ways to achieve different results. In the below example we create two MACVLAN networks joined to different subinterfaces. This type of configuration can be used to extend multiple L2 VLANs through the host interface directly to containers. The VLAN default gateway exists in the external network.



The `db` and `web` containers are connected to different MACVLAN networks in this example. Each container resides on its respective external network with an external IP provided from that network. Using this design an operator can control network policy outside of the host and segment containers at L2. The containers could have also been placed in the same VLAN by configuring them on the same MACVLAN network. This just shows the amount of flexibility offered by each network driver.

Portability and choice are important tenants in the Docker philosophy. The Docker Container Network Model provides an open interface for vendors and the community to build network drivers. The complementary evolution of Docker and SDN technologies is providing more options and capabilities every day.

Get familiar with #Docker Network drivers: bridge, overlay, macvlan (https://twitter.com/share?text=Get+familiar+with+%23Docker+Network+drivers%3A+bridge%2C+overlay%2C+macvlan&via=docker&related=docker&url=https://dockr.ly/2hNTDki)

CLICK TO TWEET (HTTPS://TWITTER.COM/SHARE?TEXT=GET+FAMILIAR+WITH+%23DOCKER+NETWORK+DRIVERS%3A+BRIDGE%2C+OVERLAY%2C+MACVLAN&VIA=DOCKER&RELATED=DOCKER&URL=HTTPS://DOCKR.LY/2H

Happy Networking!

More Resources:

- Check out (http://dockr.ly/2fIXpaU) the latest Docker Datacenter networking updates
- Read the latest RA: (https://success.docker.com/?cid=ucp112ra) Docker UCP Service Discovery and Load Balancing
- See What's New in Docker Datacenter (https://blog.docker.com/2016/11/docker-datacenter-adds-enterprise-orchestration-security-policy-refreshed-ui/)
- Sign up (http://www.docker.com/products/docker-datacenter)for a free 30 day trial

docker networking (https://blog.docker.com/tag/docker-networking/), *libnetwork. network drivers (https://blog.docker.com/tag/libnetwork-network-drivers/)*, *load balancing (https://blog.docker.com/tag/load-balancing/)*, *service discovery (https://blog.docker.com/tag/service-discovery/)*, *ucp (https://blog.docker.com/tag/ucp/)*, *universal control plane (https://blog.docker.com/tag/universal-control-plane/)*

UNDERSTANDING DOCKER NETWORKING DRIVERS AND THEIR USE CASES

By Mark Church [https://blog.docker.com/author/mark-church/]

Technical Account Manager at Docker, Inc

10 Responses to "Understanding Docker Networking Drivers and their use cases"

Renato Isidii

December 21, 2016 (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/#comment-378147)

Excellent post!!!

Reply

Alan Dixon

December 22, 2016 (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/#comment-378383)

Thanks for a great summary. Towards the end you mean important tenets instead of tenants.

Reply

Mark Church

February 27, 2017 (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/#comment-385721)

That is correct – tenets!

Reply

Tao Wang (http://blog.lab99.org/)

December 27, 2016 (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/#comment-379038)

Hi, Mark, it's a great post, very informative.

However, there is a mistake at the end of the article:
http://success.docker.com/Datacenter/Apply/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docker_Container_Networks (http://success.docker.com/Datacenter/Apply/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docker_Container_Networks)

At section "Tutorial App: MACVLAN Bridge Mode",

"'bash
#Creation of web container on host-A
host-A $ docker run -it –net macvlan –ip 192.168.0.4 -e 'DB=dog-db' -e 'ROLE=dog' –name dog-web chrch/web

#Creation of db container on host-B
host-B $ docker run -it –net macvlan –ip 192.168.0.5 –name dog-db redis
"'

'DB=dog-db' will not work, as the Service Discovery of MACVLAN only works 'locally', not 'globally'. So, although both macvlan network called 'macvlan' on both hosts, the docker engine on host-A has no idea what is the IP of 'dog-db' on host-B.

Reply

Mark Church

February 27, 2017 (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/#comment-385722)

Thanks for the catch Tao. Service discovery with local scope drivers will only work locally with a single docker engine. Only Swarm scope and global scope drivers distribute service discovery information across a cluster. We'll correct this in the next version of the reference architecture.

Reply

Jonathan

February 25, 2017 (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/#comment-385468)

Hi,

Is it possible to create network policies with the overlay driver? Is this something that will soon be supported?

Reply

Mark Church

February 27, 2017 (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/#comment-385723)

Jonathan,

Currently overlay network policies can be defined via use of multiple overlay networks to provide granular segmentation. We are looking at providing better ways of defining network policy in future releases. Feel free to drop me an email for what kind of policy management would work for your use-cases -> church at docker.com

Reply

Arthur

May 19, 2017 (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/#comment-396935)

Hi,

Thanks for this article, it's a really good job ! However I have a question for you, do you know how many network interfaces is it possible to have for 1 container ?

Thanks,
Arthur

Reply

rob

June 26, 2017 (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/#comment-402419)

in the macvlan scenario, why can't the two servers have the same IP or be in same subnet ? from the vlan perspective , the router could be in two different vrfs upstream. I feel sometimes docker network people need to think outside of the box a bit. I have tried this configuration and docker requires a unique subnet per vlan. Why does docker even need to know the subnet in this scenario if it is just bridging between container and the external network ?

Reply

Igor Podlesny (https://ru.etcinsider.com/)

January 2, 2018 (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/#comment-420703)

Still don't see how all this (any of this) helps to accomplish very simple task: "attaching" a container to network as if it was an single IP node. With –net=host it won't be bound to single IP no way; with bridge you would have to tinker with port exposing and this is not even closely an equivalent of attaching node to network. All-in-all: an over-engineered something

P. S. There're also a few flaws in current Docker approach:
• it tries to set –gateway although it's not always valid and required even; gateway should be optional not mandatory
• you can't create /32 bridge network although it's pretty legitimate

Reply

Leave a Reply

Name (required)

Email (will not be published) (required)

Website

Comment

Submit Comment

Notify me of follow-up comments by email.

Notify me of new posts by email.

Related Posts

#SwarmWeek: Advanced Orchestration with Docker Swarm (https://blog.docker.com/2016/03/swarmweek-advanced-orchestration-docker-swarm/)

By Mano Marks (https://blog.docker.com/author/mano-marks/)

March 10, 2016

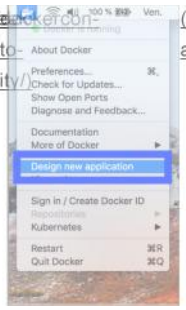
Docker orchestration (https://blog.docker.com/tag/docker-orchestration/), docker swarm (https://blog.docker.com/tag/docker-swarm/), orchestration (https://blog.docker.com/tag/orchestration/), Swarm (https://blog.docker.com/tag/swarm/), swarm week (https://blog.docker.com/tag/swarm-week/), swarmweek (https://blog.docker.com/tag/swarmweek/), video (https://blog.docker.com/tag/video/)

DockerCon video: Lessons from using Docker to improve web developer productivity (https://blog.docker.com/2014/06/dockercon-video-lessons-from-using-docker-to-improve-web-developer-productivity/)

By rogaha (https://blog.docker.com/author/rogaha/)

June 30, 2014

Auto.com (https://blog.docker.com/tag/auto-com/), aws (https://blog.docker.com/tag/aws-2/), flux7 (https://blog.docker.com/tag/flux7/), productivity (https://blog.docker.com/tag/productivity/), QA (https://blog.docker.com/tag/qa/)



Introducing an Easier Way To Design Applications in Docker Desktop (https://blog.docker.com/2018/06/des applications-in-docker-desktop/)

By Gareth Rushgrove (https://blog.docker.com/author/gareth-rushgrove/)

June 13, 2018

App in Docker Desktop (https://blog.docker.com/tag/app-in-docker-desktop/), Docker Custom App (https://blog.docker.com/tag/docker-custom-app/), Docker for Desktops (https://blog.docker.com/tag/docker-for-desktops/), docker for mac (https://blog.docker.com/tag/docker-for-mac/), docker for windows (https://blog.docker.com/tag/docker-for-windows/), Docker Template (https://blog.docker.com/tag/docker-template/), docker-desktop (https://blog.docker.com/tag/docker-desktop/)

Get the Latest Docker News by Email

Docker Weekly is a newsletter with the latest content on Docker and the agenda for the upcoming weeks.

Subscribe to our newsletter

Company Email

Select Country...

Submit

Sign up for our newsletter.

Company Email

Select Country...

Submit

What is Docker (https://www.docker.com/what-docker)

What is a Container (https://www.docker.com/what-container)

Use Cases (https://www.docker.com/use-cases)

Customers (https://www.docker.com/customers)

For Government (https://www.docker.com/industry-government)

For IT Pros (https://www.docker.com/itpro)

Find a Partner (https://www.docker.com/find-partner)

Become a Partner (https://www.docker.com/partners/partner-program)

About Docker (https://www.docker.com/company)

Management (https://www.docker.com/company/management)

Press & News (https://www.docker.com/company/news-and-press)

Careers (https://www.docker.com/careers)

Product (https://www.docker.com/get-docker)

Pricing (https://www.docker.com/pricing)

Community Edition (https://www.docker.com/community-edition)

Enterprise Edition (https://www.docker.com/enterprise-edition)

Docker Cloud (https://cloud.docker.com/)

Docker Store (https://store.docker.com/)

Documentation (https://docs.docker.com/)

Blog (/)

RSS Feed (/feed/)

Training (https://training.docker.com/)

Knowledge Base (https://success.docker.com/kbase)

Resources (https://www.docker.com/products/resources)

Community (https://www.docker.com/docker-community)

Open Source (https://www.docker.com/technologies/overview)

Forums (https://forums.docker.com/)

Docker Captains (https://www.docker.com/community/docker-captains)

Scholarships (https://www.docker.com/docker-community/scholarships)

Community News (/curated/)

Status (http://status.docker.com/) Security (https://docker.com/docker-security) Legal (https://www.docker.com/legal)

Contact (https://www.docker.com/company/contact)

Authored by: Mark Church (/author/markchurch)

Docker Reference Architecture: Designing Scalable, Portable Docker Container Networks

Article ID: KB000801

EE NETWORKING CSE-1.12.3-CS3 CSE-1.13.1-CS1 EE-17.03.0-EE-1 UCP-2.0.0

What You Will Learn

Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries – anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment. By default, containers isolate applications from one another and the underlying infrastructure, while providing an added layer of protection for the application.

What if the applications need to communicate with each other, the host, or an external network? How do you design a network to allow for proper connectivity while maintaining application portability, service discovery, load balancing, security, performance, and scalability? This document addresses these network design challenges as well as the tools available and common deployment patterns. It does not specify or recommend physical network design but provides options for how to design Docker networks while considering the constraints of the application and the physical network.

Prerequisites

Before continuing, being familiar with Docker concepts and Docker Swarm is recommended:

- Docker concepts (https://docs.docker.com/engine/understanding-docker/)
- Docker Swarm (https://docs.docker.com/engine/swarm/) and Swarm mode concepts (https://docs.docker.com/engine/swarm/key-concepts/#/services-and-tasks)

Challenges of Networking Containers and Microservices

Microservices practices have increased the scale of applications which has put even more importance on the methods of connectivity and isolation provided to applications. The Docker networking philosophy is application driven. It aims to provide options and flexibility to the network operators as well as the right level of abstraction to the application developers.

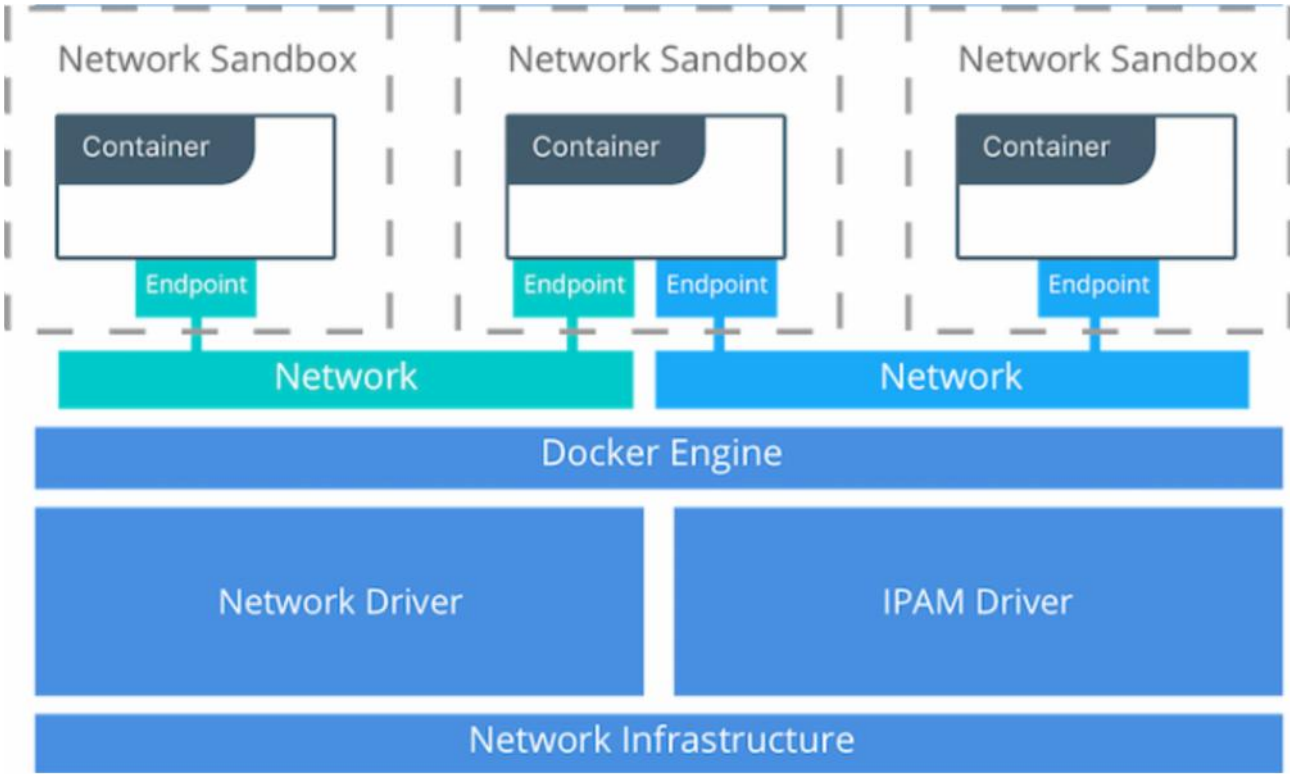
Like any design, network design is a balancing act. Docker EE and the Docker ecosystem provide multiple tools to network engineers to achieve the best balance for their applications and environments. Each option provides different benefits and tradeoffs. The remainder of this guide details each of these choices so network engineers can understand what might be best for their environments.

Docker has developed a new way of delivering applications, and with that, containers have also changed some aspects of how networking is approached. The following topics are common design themes for containerized applications:

- Portability
    - *How do I guarantee maximum portability across diverse network environments while taking advantage of unique network characteristics?*
- Service Discovery
    - *How do I know where services are living as they are scaled up and down?*
- Load Balancing
    - *How do I share load across services as services themselves are brought up and scaled?*
- Security
    - *How do I segment to prevent the wrong containers from accessing each other?*
    - *How do I guarantee that a container with application and cluster control traffic is secure?*
- Performance
    - *How do I provide advanced network services while minimizing latency and maximizing bandwidth?*
- Scalability
    - *How do I ensure that none of these characteristics are sacrificed when scaling applications across many hosts?*

The Container Networking Model

The Docker networking architecture is built on a set of interfaces called the *Container Networking Model* (CNM). The philosophy of CNM is to provide application portability across diverse infrastructures. This model strikes a balance to achieve application portability and also takes advantage of special features and capabilities of the infrastructure.



CNM Constructs

There are several high-level constructs in the CNM. They are all OS and infrastructure agnostic so that applications can have a uniform experience no matter the infrastructure stack.

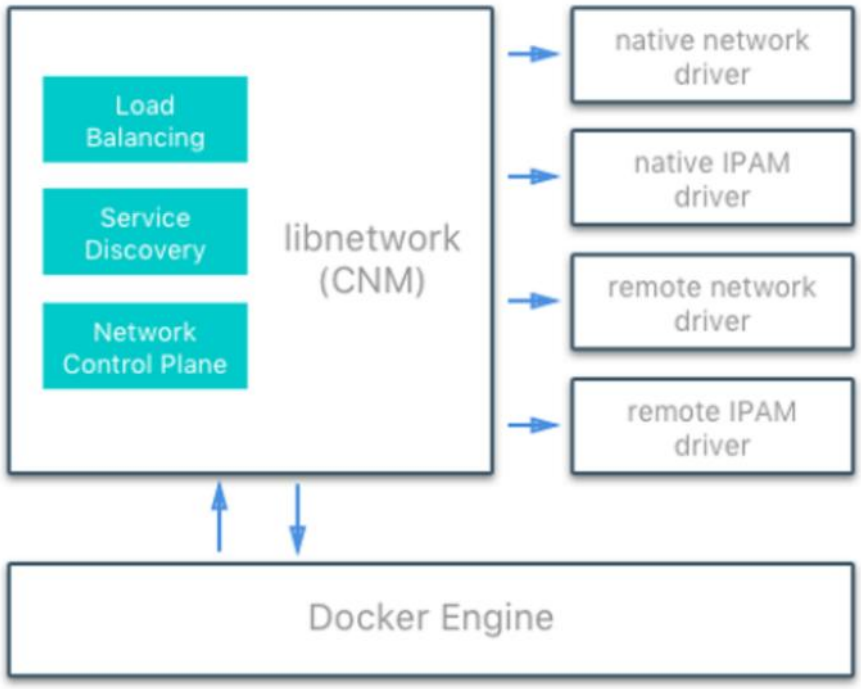
- Sandbox — A Sandbox contains the configuration of a container's network stack. This includes management of the container's interfaces, routing table, and DNS settings. An implementation of a Sandbox could be a Linux Network Namespace, a FreeBSD Jail, or other similar concept. A Sandbox may contain many endpoints from multiple networks.
- Endpoint — An Endpoint joins a Sandbox to a Network. The Endpoint construct exists so the actual connection to the network can be abstracted away from the application. This helps maintain portability so that a service can use different types of network drivers without being concerned with how it's connected to that network.
- Network — The CNM does not specify a Network in terms of the OSI model. An implementation of a Network could be a Linux bridge, a VLAN, etc. A Network is a collection of endpoints that have connectivity between them. Endpoints that are not connected to a network do not have connectivity on a network.

CNM Driver Interfaces

The Container Networking Model provides two pluggable and open interfaces that can be used by users, the community, and vendors to leverage additional functionality, visibility, or control in the network.

The following network drivers exist:

- Network Drivers — Docker Network Drivers provide the actual implementation that makes networks work. They are pluggable so that different drivers can be used and interchanged easily to support different use cases. Multiple network drivers can be used on a given Docker Engine or Cluster concurrently, but each Docker network is only instantiated through a single network driver. There are two broad types of CNM network drivers:
  - Native Network Drivers — Native Network Drivers are a native part of the Docker Engine and are provided by Docker. There are multiple drivers to choose from that support different capabilities like overlay networks or local bridges.
  - Remote Network Drivers — Remote Network Drivers are network drivers created by the community and other vendors. These drivers can be used to provide integration with incumbent software and hardware. Users can also create their own drivers in cases where they desire specific functionality that is not supported by an existing network driver.
- IPAM Drivers — Docker has a native IP Address Management Driver that provides default subnets or IP addresses for networks and endpoints if they are not specified. IP addressing can also be manually assigned through network, container, and service create commands. Remote IPAM drivers also exist and provide integration to existing IPAM tools.



Docker Native Network Drivers

The Docker native network drivers are part of Docker Engine and don't require any extra modules. They are invoked and used through standard `docker network` commands. The following native network drivers exist.

Driver	Description
Host	With the `host` driver, a container uses the networking stack of the host. There is no namespace separation, and all interfaces on the host can be used directly by the container
Bridge	The `bridge` driver creates a Linux bridge on the host that is managed by Docker. By default containers on a bridge can communicate with each other. External access to containers can also be configured through the `bridge` driver
Overlay	The `overlay` driver creates an overlay network that supports multi-host networks out of the box. It uses a combination of local Linux bridges and VXLAN to overlay container-to-container communications over physical network infrastructure
MACVLAN	The `macvlan` driver uses the MACVLAN bridge mode to establish a connection between container interfaces and a parent host interface (or sub-interfaces). It can be used to provide IP addresses to containers that are routable on the physical network. Additionally VLANs can be trunked to the `macvlan` driver to enforce Layer 2 container segmentation
None	The `none` driver gives a container its own networking stack and network namespace but does not configure interfaces inside the container. Without additional configuration, the container is completely isolated from the host networking stack

Network Scope

As seen in the `docker network ls` output, Docker network drivers have a concept of *scope*. The network scope is the domain of the driver which can be the `local` or `swarm` scope. Local scope drivers provide connectivity and network services (such as DNS or IPAM) within the scope of the host. Swarm scope drivers provide connectivity and network services across a swarm cluster. Swarm scope networks have the same network ID across the entire cluster while local scope networks have a unique network ID on each host.

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
1475f03fbecb        bridge              bridge              local
e2d8a4bd86cb        docker_gwbridge     bridge              local
407c477060e7        host                host                local
f4zr3zrswlyg        ingress             overlay             swarm
c97909a4b198        none                null                local
```

Docker Remote Network Drivers

The following community- and vendor-created remote network drivers are compatible with CNM. Each provides unique capabilities and network services for containers.

Driver	Description
contiv (http://contiv.github.io/)	An open source network plugin led by Cisco Systems to provide infrastructure and security policies for multi-tenant microservices deployments. Contiv also provides integration for non-container workloads and with physical networks, such as ACI. Contiv implements remote network and IPAM drivers.
weave (https://www.weave.works/docs/net/latest/introducing-weave/)	A network plugin that creates a virtual network that connects Docker containers across multiple hosts or clouds. Weave provides automatic discovery of applications, can operate on partially connected networks, does not require an external cluster store, and is operations friendly.

Driver	Description
calico (https://www.projectcalico.org/)	An open source solution for virtual networking in cloud datacenters. It targets datacenters where most of the workloads (VMs, containers, or bare metal servers) only require IP connectivity. Calico provides this connectivity using standard IP routing. Isolation between workloads — whether according to tenant ownership or any finer grained policy — is achieved via iptables programming on the servers hosting the source and destination workloads.
kuryr (https://github.com/openstack/kuryr)	A network plugin developed as part of the OpenStack Kuryr project. It implements the Docker networking (libnetwork) remote driver API by utilizing Neutron, the OpenStack networking service. Kuryr includes an IPAM driver as well.

Docker Remote IPAM Drivers

Community and vendor created IPAM drivers can also be used to provide integrations with existing systems or special capabilities.

Driver	Description
infoblox (https://hub.docker.com/r/infoblox/ipam-driver/)	An open source IPAM plugin that provides integration with existing Infoblox tools.

> There are many Docker plugins that exist and more are being created all the time. Docker maintains a list of the most common plugins (https://docs.docker.com/engine/extend/legacy_plugins/).

Linux Network Fundamentals

The Linux kernel features an extremely mature and performant implementation of the TCP/IP stack (in addition to other native kernel features like DNS and VXLAN). Docker networking uses the kernel's networking stack as low level primitives to create higher level network drivers. Simply put, *Docker networking is Linux networking.*

This implementation of existing Linux kernel features ensures high performance and robustness. Most importantly, it provides portability across many distributions and versions, which enhances application portability.

There are several Linux networking building blocks which Docker uses to implement its native CNM network drivers. This list includes Linux bridges, network namespaces, veth pairs, and iptables. The combination of these tools, implemented as network drivers, provides the forwarding rules, network segmentation, and management tools for complex network policy.

The Linux Bridge

A Linux bridge is a Layer 2 device that is the virtual implementation of a physical switch inside the Linux kernel. It forwards traffic based on MAC addresses which it learns dynamically by inspecting traffic. Linux bridges are used extensively in many of the Docker network drivers. A Linux bridge is not to be confused with the `bridge` Docker network driver which is a higher level implementation of the Linux bridge.

Network Namespaces

A Linux network namespace is an isolated network stack in the kernel with its own interfaces, routes, and firewall rules. It is a security aspect of containers and Linux, used to isolate containers. In networking terminology they are akin to a VRF that segments the network control and data plane inside the host. Network namespaces ensure that two containers on the same host aren't able to communicate with each other or even the host itself unless configured to do so via Docker networks. Typically, CNM network drivers implement separate namespaces for each container. However, containers can share the same network namespace or even be a part of the host's network namespace. The host network namespace containers the host interfaces and host routing table. This network namespace is called the global network namespace.

Virtual Ethernet Devices

A virtual ethernet device or veth is a Linux networking interface that acts as a connecting wire between two network namespaces. A veth is a full duplex link that has a single interface in each namespace. Traffic in one interface is directed out the other interface. Docker network drivers utilize veths to provide explicit connections between namespaces when Docker networks are

created. When a container is attached to a Docker network, one end of the veth is placed inside the container (usually seen as the `ethX` interface) while the other is attached to the Docker network.

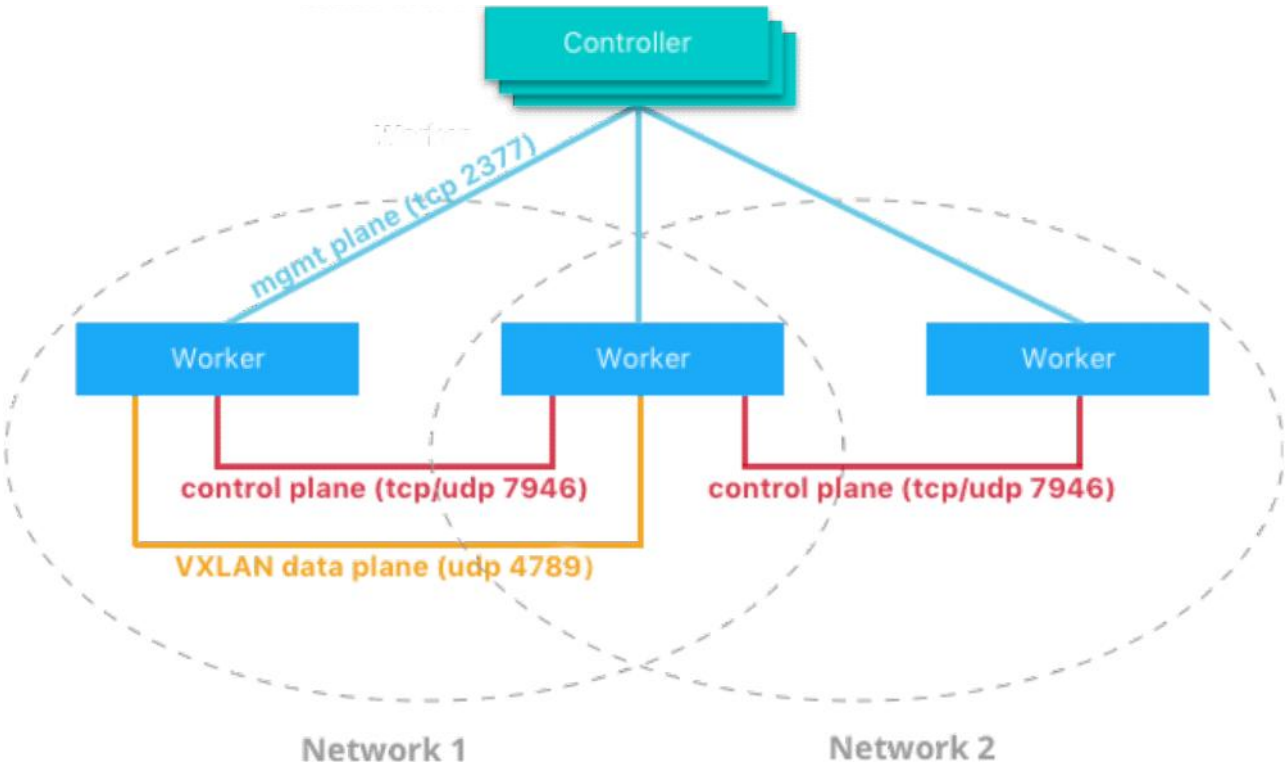
iptables

`iptables` is the native packet filtering system that has been a part of the Linux kernel since version 2.4. It's a feature rich L3/L4 firewall that provides rule chains for packet marking, masquerading, and dropping. The native Docker network drivers utilize `iptables` extensively to segment network traffic, provide host port mapping, and to mark traffic for load balancing decisions.

Docker Network Control Plane

The Docker-distributed network control plane manages the state of Swarm-scoped Docker networks in addition to propagating control plane data. It is a built-in capability of Docker Swarm clusters and does not require any extra components such as an external KV store. The control plane uses a Gossip (https://en.wikipedia.org/wiki/Gossip_protocol) protocol based on SWIM (https://www.cs.cornell.edu/~asdas/research/dsn02-swim.pdf) to propagate network state information and topology across Docker container clusters. The Gossip protocol is highly efficient at reaching eventual consistency within the cluster while maintaining constant rates of message size, failure detection times, and convergence time across very large scale clusters. This ensures that the network is able to scale across many nodes without introducing scaling issues such as slow convergence or false positive node failures.

The control plane is highly secure, providing confidentiality, integrity, and authentication through encrypted channels. It is also scoped per network which greatly reduces the updates that any given host receives.



It is composed of several components that work together to achieve fast convergence across large scale networks. The distributed nature of the control plane ensures that cluster controller failures don't affect network performance.

The Docker network control plane components are as follows:

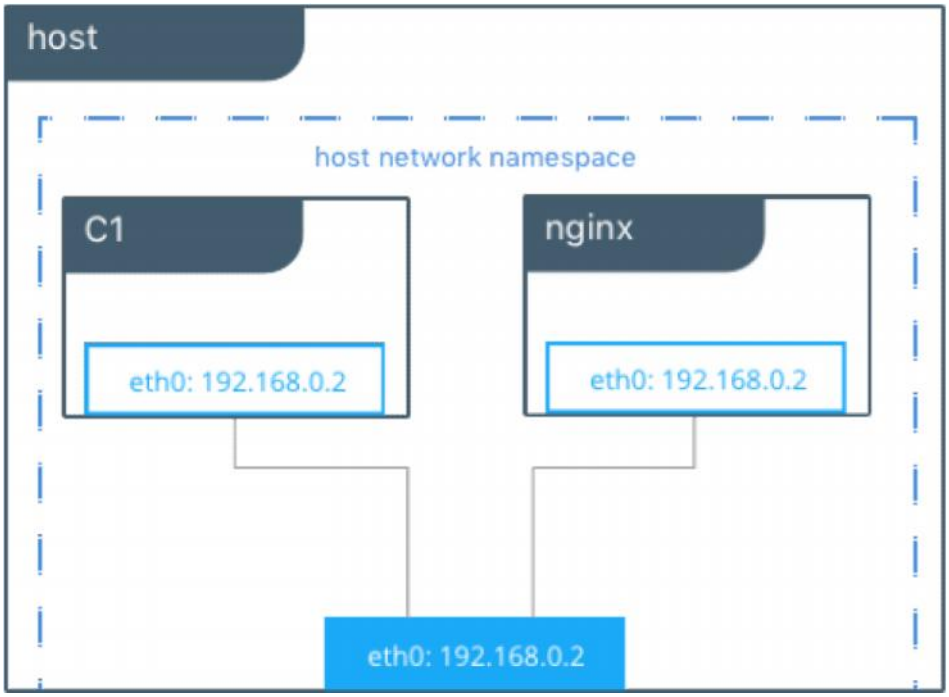
- Message Dissemination updates nodes in a peer-to-peer fashion fanning out the information in each exchange to a larger group of nodes. Fixed intervals and size of peer groups ensures that network usage is constant even as the size of the cluster scales. Exponential information propagation across peers ensures that convergence is fast and bounded across any cluster size.
- Failure Detection utilizes direct and indirect hello messages to rule out network congestion and specific paths from causing false positive node failures.
- Full State Syncs occur periodically to achieve consistency faster and resolve network partitions.
- Topology Aware algorithms understand the relative latency between themselves and other peers. This is used to optimize the peer groups which makes convergence faster and more efficient.
- Control Plane Encryption protects against man in the middle and other attacks that could compromise network security.

> The Docker Network Control Plane is a component of Swarm (https://docs.docker.com/engine/swarm/) and requires a Swarm cluster to operate.

Docker Host Network Driver

The `host` network driver is most familiar to those new to Docker because it's the same networking configuration that Linux uses without Docker. `--net=host` effectively turns Docker networking off and containers use the host (or default) networking stack of the host operating system.

Typically with other networking drivers, each container is placed in its own *network namespace* (or sandbox) to provide complete network isolation from each other. With the `host` driver containers are all in the same host network namespace and use the network interfaces and IP stack of the host. All containers in the `host` network are able to communicate with each other on the host interfaces. From a networking standpoint this is equivalent to multiple processes running on a host without containers. Because they are using the same host interfaces, no two containers are able to bind to the same TCP port. This may cause port contention if multiple containers are being scheduled on the same host.



```
#Create containers on the host network
$ docker run -itd --net host --name C1 alpine sh
$ docker run -itd --net host --name nginx

#Show host eth0
$ ip add | grep eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP group default ql
en 1000
    inet 172.31.21.213/20 brd 172.31.31.255 scope global eth0

#Show eth0 from C1
$ docker run -it --net host --name C1 alpine ip add | grep eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP qlen 1000
    inet 172.31.21.213/20 brd 172.31.31.255 scope global eth0

#Contact the nginx container through localhost on C1
$ curl localhost
!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

In this example, the host, `C1` and `nginx` all share the same interface for `eth0` when containers use the `host` network. This makes `host` ill suited for multi-tenant or highly secure applications. `host` containers have network access to every other container on the host. Communicate is possible between containers using `localhost` as shown in the example when `curl nginx` is executed from `C1`.

With the `host` driver, Docker does not manage any portion of the container networking stack such as port mapping or routing rules. This means that common networking flags like `-p` and `--icc` have no meaning for the `host` driver. They are ignored. This does make the `host` networking the simplest and lowest latency of the networking drivers. The traffic path goes directly from the container process to the host interface, offering bare-metal performance that is equivalent to a non-containerized process.

Full host access and no automated policy management may make the `host` driver a difficult fit as a general network driver. However, `host` does have some interesting properties that may be applicable for use cases such as ultra high performance applications or application troubleshooting.

Docker Bridge Network Driver

This section explains the default Docker bridge network as well as user-defined bridge networks.

Default Docker Bridge Network

On any host running Docker Engine, there is, by default, a local Docker network named `bridge`. This network is created using a `bridge` network driver which instantiates a Linux bridge called `docker0`. This may sound confusing.

- `bridge` is the name of the Docker network
- `bridge` is the network driver, or template, from which this network is created
- `docker0` is the name of the Linux bridge that is the kernel building block used to implement this network

On a standalone Docker host, `bridge` is the default network that containers connect to if no other network is specified. In the following example a container is created with no network parameters. Docker Engine connects it to the `bridge` network by default. Inside the container, notice `eth0` which is created by the `bridge` driver and given an address by the Docker native IPAM driver.

```
#Create a busybox container named "c1" and show its IP addresses
host $ docker run -it --name c1 busybox sh
c1 # ip address
4: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
...
```

> A container interface's MAC address is dynamically generated and embeds the IP address to avoid collision. Here `ac:11:00:02` corresponds to `172.17.0.2`.

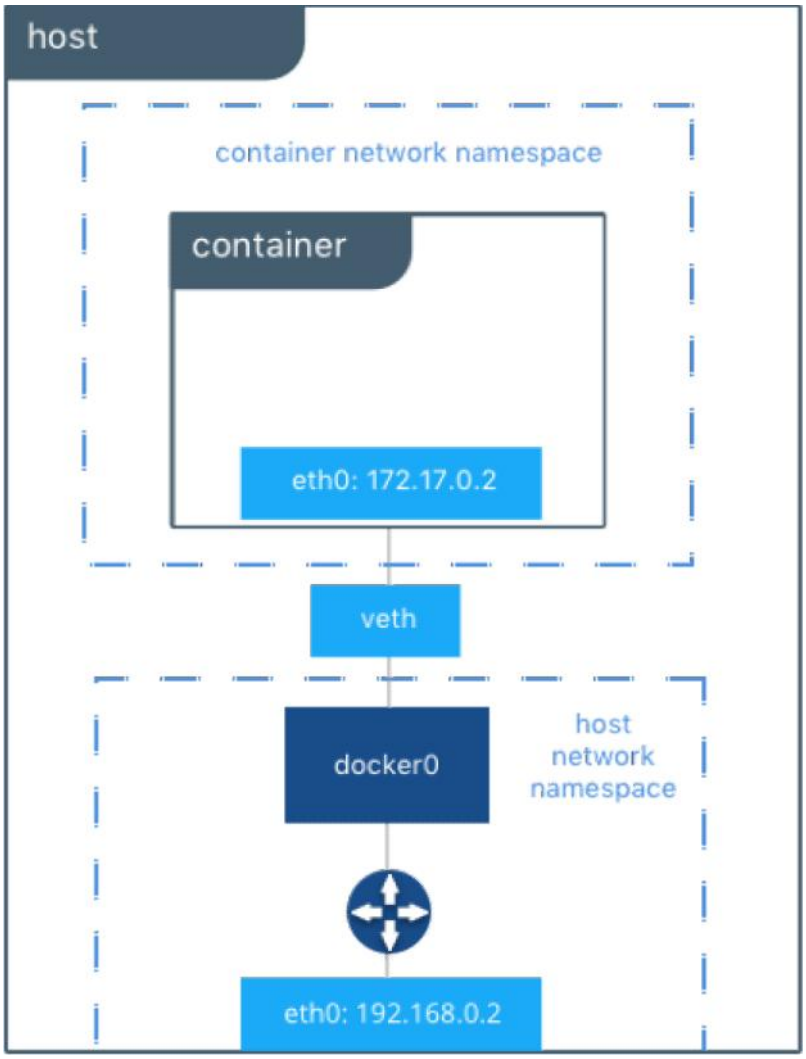
The tool `brctl` on the host shows the Linux bridges that exist in the host network namespace. It shows a single bridge called `docker0`. `docker0` has one interface, `vetha3788c4`, which provides connectivity from the bridge to the `eth0` interface inside container `c1`.

```
host $ brctl show
bridge name      bridge id            STP enabled    interfaces
docker0          8000.0242504b5200    no             vethb64e8b8
```

Inside container `c1`, the container routing table directs traffic to `eth0` of the container and thus the `docker0` bridge.

```
c1# ip route
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0  src 172.17.0.2
```

A container can have zero to many interfaces depending on how many networks it is connected to. Each Docker network can only have a single interface per container.



As shown in the host routing table, the IP interfaces in the global network namespace now include `docker0`. The host routing table provides connectivity between `docker0` and `eth0` on the external network, completing the path from inside the container to the external network.

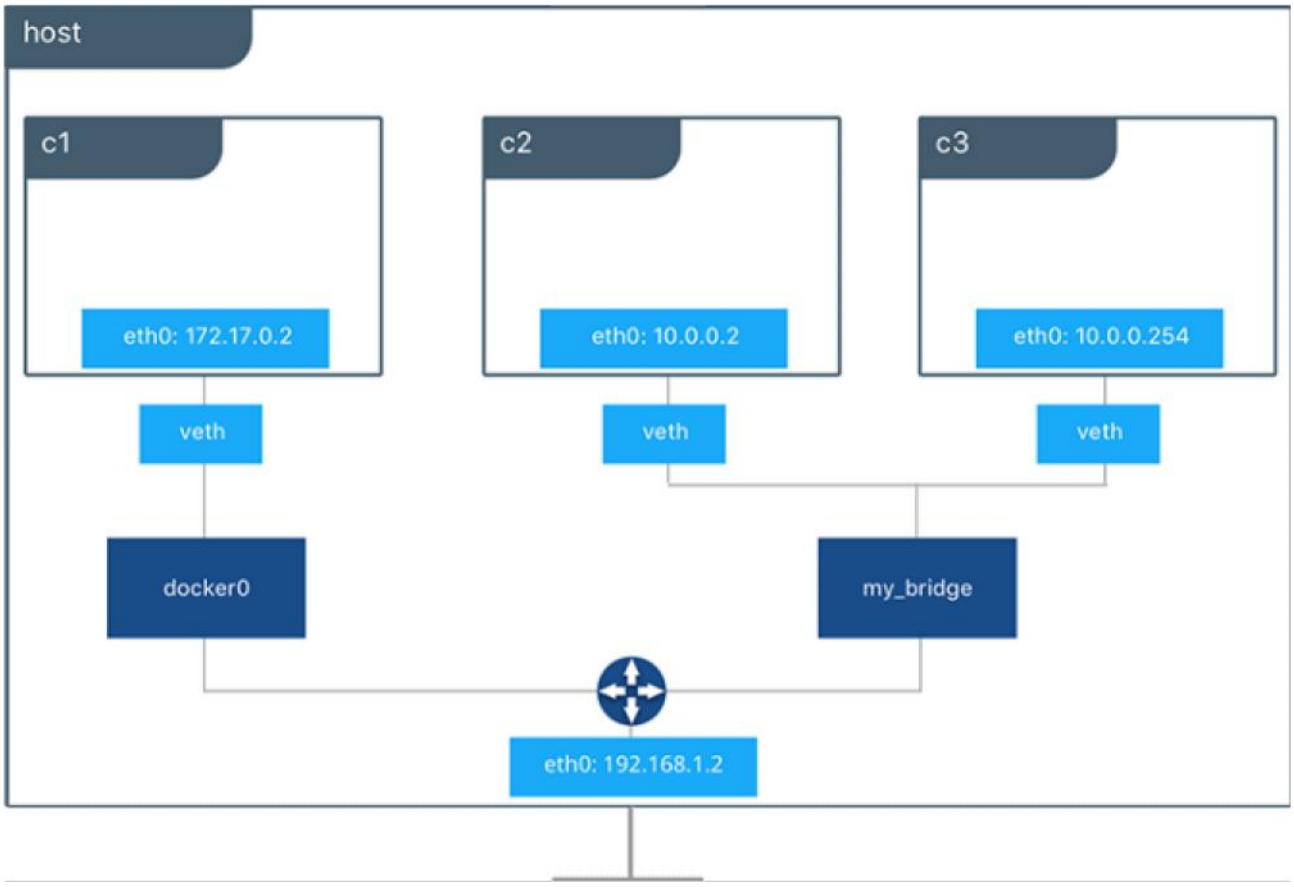
```
host $ ip route
default via 172.31.16.1 dev eth0
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.42.1
172.31.16.0/20 dev eth0  proto kernel  scope link  src 172.31.16.102
```

By default `bridge` is assigned one subnet from the ranges 172.[17-31].0.0/16 or 192.168.[0-240].0/20 which does not overlap with any existing host interface. The default `bridge` network can also be configured to use user-supplied address ranges. Also, an existing Linux bridge can be used for the `bridge` network rather than Docker creating one. Go to the Docker Engine docs (https://docs.docker.com/engine/userguide/networking/default_network/custom-docker0/) for more information about customizing `bridge` .

> The default `bridge` network is the only network that supports legacy links (https://docs.docker.com/engine/userguide/networking/default_network/dockerlinks/). Name-based service discovery and user-provided IP addresses are not supported by the default `bridge` network.

User-Defined Bridge Networks

In addition to the default networks, users can create their own networks called user-defined networks of any network driver type. In the case of user-defined `bridge` networks, a new Linux bridge is setup on the host. Unlike the default `bridge` network, user-defined networks supports manual IP address and subnet assignment. If an assignment isn't given, then Docker's default IPAM driver assigns the next subnet available in the private IP space.



Below a user-defined `bridge` network is created with two containers attached to it. A subnet is specified, and the network is named `my_bridge`. One container is not given IP parameters, so the IPAM driver assigns it the next available IP in the subnet. The other container has its IP specified.

```
$ docker network create -d bridge --subnet 10.0.0.0/24 my_bridge
$ docker run -itd --name c2 --net my_bridge busybox sh
$ docker run -itd --name c3 --net my_bridge --ip 10.0.0.254 busybox sh
```

`brctl` now shows a second Linux bridge on the host. The name of the Linux bridge, `br-4bcc22f5e5b9`, matches the Network ID of the `my_bridge` network. `my_bridge` also has two `veth` interfaces connected to containers `c2` and `c3`.

```
$ brctl show
bridge name      bridge id            STP enabled    interfaces
br-b5db4578d8c9  8000.02428d936bb1    no             vethc9b3282
                                                     vethf3ba8b5
docker0          8000.0242504b5200    no             vethb64e8b8

$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
b5db4578d8c9        my_bridge           bridge              local
e1cac9da3116        bridge              bridge              local
...
```

Listing the global network namespace interfaces shows the Linux networking circuitry that's been instantiated by Docker Engine. Each `veth` and Linux bridge interface appears as a link between one of the Linux bridges and the container network namespaces.

```
$ ip link

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
5: vethb64e8b8@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
6: br-b5db4578d8c9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
8: vethc9b3282@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
10: vethf3ba8b5@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
...
```

External Access for Standalone Containers

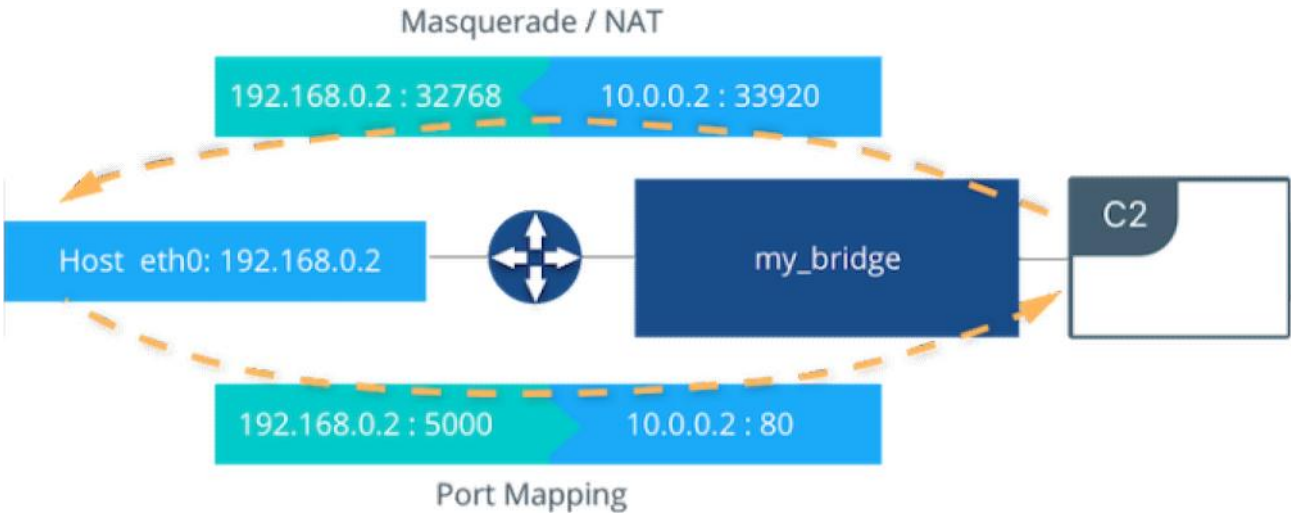
By default all containers on the same Docker network (multi-host swarm scope or local scope) have connectivity with each other on all ports. Communication between different Docker networks and and container ingress traffic that originates from outside Docker is firewalled. This is a

fundamental security aspect that protects container applications from the outside world and from each other. This is outlined in more detail in Network Security (https://success.docker.com/api/asset/.%2Frefarch%2Fnetworking%2F#security)

For most types of Docker networks (`bridge` and `overlay` included) external ingress access for applications must be explicitly granted. This is done through internal port mapping. Docker publishes ports exposed on host interfaces to internal container interfaces. The following diagram depicts ingress (bottom arrow) and egress (top arrow) traffic to container `C2` . Outbound (egress) container traffic is allowed by default. Egress connections initiated by containers are masqueraded/SNATed to an ephemeral port (*typically in the range of 32768 to 60999*). Return traffic on this connection is allowed, and thus the container uses the best routable IP address of the host on the ephemeral port.

Ingress access is provided through explicit port publishing. Port publishing is done by Docker Engine and can be controlled through UCP or the Engine CLI. A specific or randomly chosen port can be configured to expose a service or container. The port can be set to listen on a specific (or all) host interfaces, and all traffic is mapped from this port to a port and interface inside the container.

```
$ docker run -d --name C2 --net my_bridge -p 5000:80 nginx
```



External access is configured using `--publish / -p` in the Docker CLI or UCP. After running the above command, the diagram shows that container `C2` is connected to the `my_bridge` network and has an IP address of `10.0.0.2` . The container advertises its service to the outside world on port `5000` of the host interface `192.168.0.2` . All traffic going to this interface:port is port published to `10.0.0.2:80` of the container interface.

Outbound traffic initiated by the container is masqueraded so that it is sourced from ephemeral port `32768` on the host interface `192.168.0.2`. Return traffic uses the same IP address and port for its destination and is masqueraded internally back to the container address:port `10.0.0.2:33920`. When using port publishing, external traffic on the network always uses the host IP and exposed port and never the container IP and internal port.

For information about exposing containers and services in a cluster of Docker Engines read External Access for Swarm Services (https://success.docker.com/api/asset/.%2Frefarch%2Fnetworking%2F#swarm-external).

Overlay Driver Network Architecture

The native Docker `overlay` network driver radically simplifies many of the challenges in multi-host networking. With the `overlay` driver, multi-host networks are first-class citizens inside Docker without external provisioning or components. `overlay` uses the Swarm-distributed control plane to provide centralized management, stability, and security across very large scale clusters.

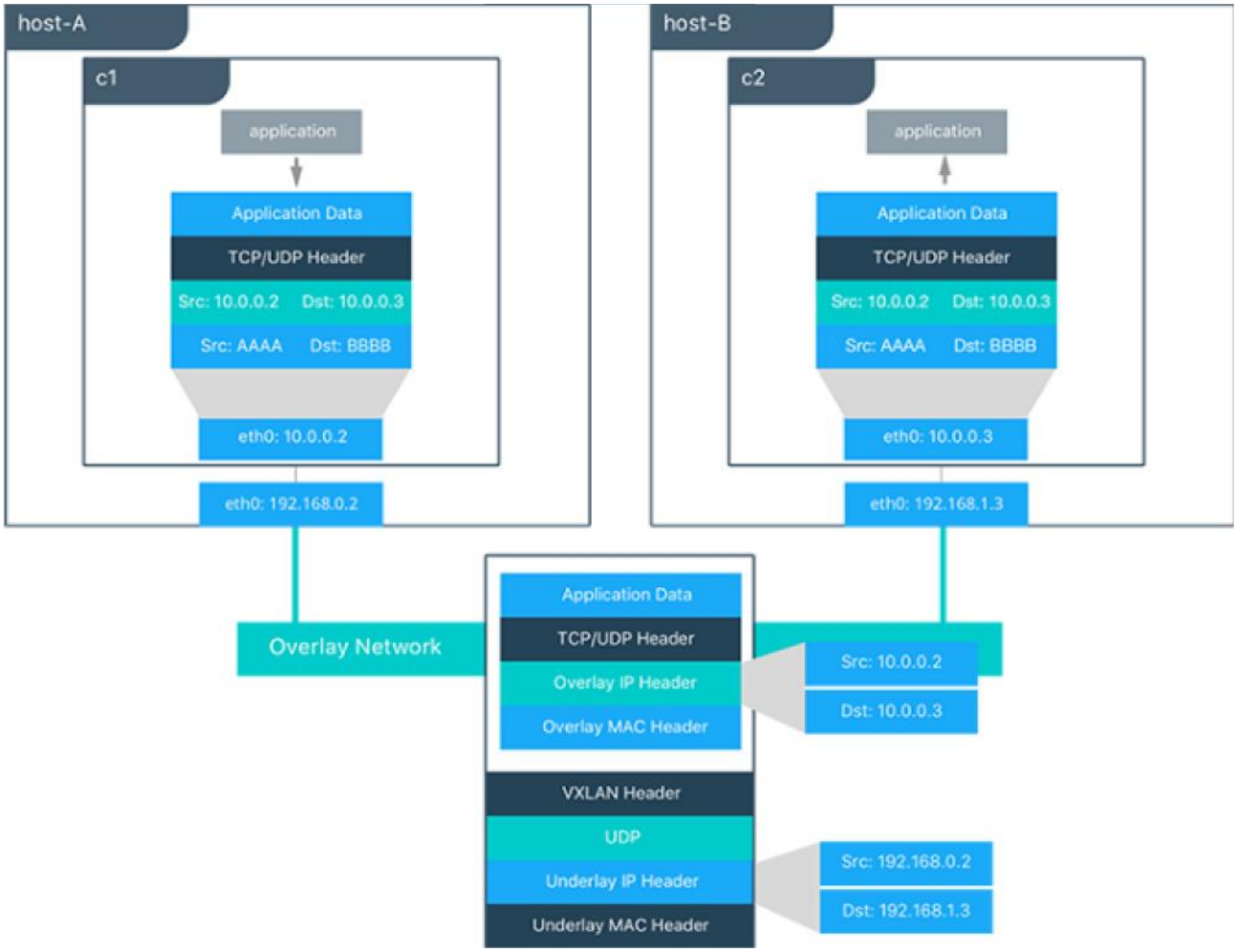
VXLAN Data Plane

The `overlay` driver utilizes an industry-standard VXLAN data plane that decouples the container network from the underlying physical network (the *underlay*). The Docker overlay network encapsulates container traffic in a VXLAN header which allows the traffic to traverse the physical Layer 2 or Layer 3 network. The overlay makes network segmentation dynamic and easy to control no matter what the underlying physical topology. Use of the standard IETF VXLAN header promotes standard tooling to inspect and analyze network traffic.

> VXLAN has been a part of the Linux kernel since version 3.7, and Docker uses the native VXLAN features of the kernel to create overlay networks. The Docker overlay datapath is entirely in kernel space. This results in fewer context switches, less CPU overhead, and a low-latency, direct traffic path between applications and the physical NIC.

IETF VXLAN (RFC 7348 (https://datatracker.ietf.org/doc/rfc7348/)) is a data-layer encapsulation format that overlays Layer 2 segments over Layer 3 networks. VXLAN is designed to be used in standard IP networks and can support large-scale, multi-tenant designs on shared physical network infrastructure. Existing on-premises and cloud-based networks can support VXLAN transparently.

VXLAN is defined as a MAC-in-UDP encapsulation that places container Layer 2 frames inside an underlay IP/UDP header. The underlay IP/UDP header provides the transport between hosts on the underlay network. The overlay is the stateless VXLAN tunnel that exists as point-to-multipoint connections between each host participating in a given overlay network. Because the overlay is independent of the underlay topology, applications become more portable. Thus, network policy and connectivity can be transported with the application whether it is on-premises, on a developer desktop, or in a public cloud.



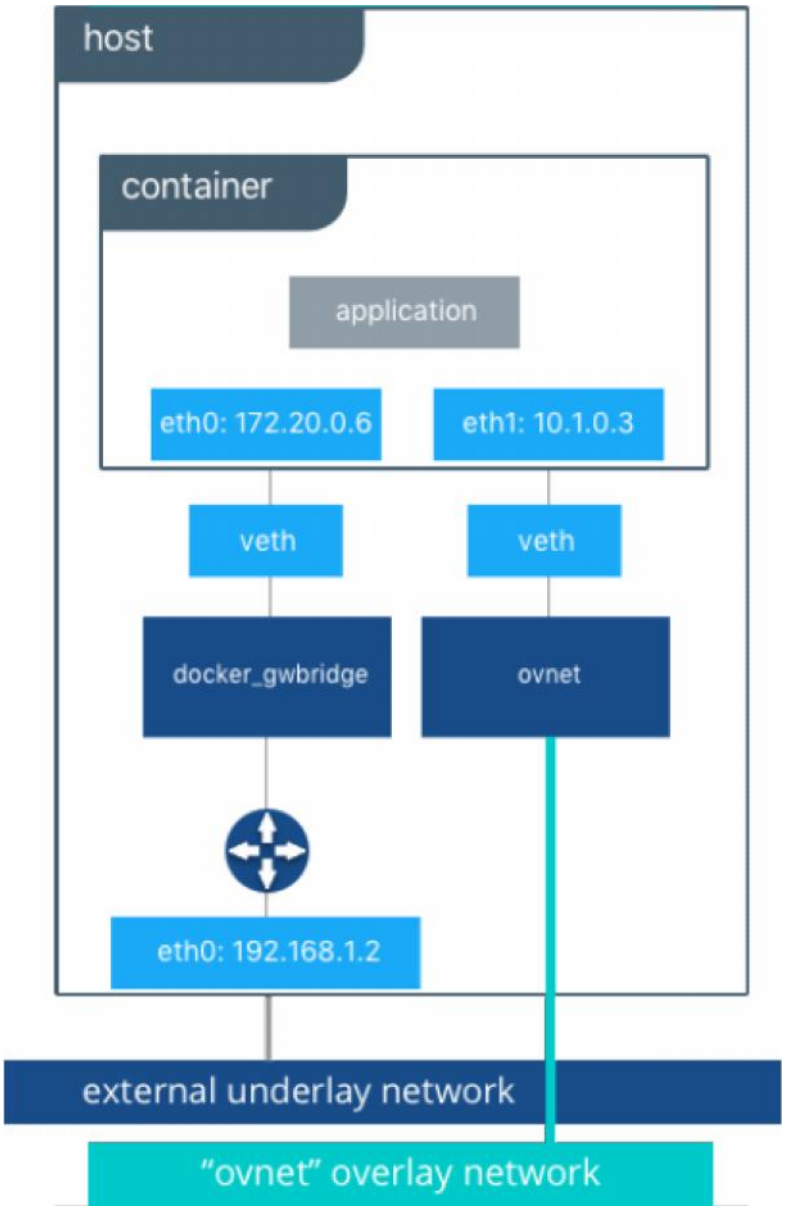
In this diagram, the packet flow on an overlay network is shown. Here are the steps that take place when `c1` sends `c2` packets across their shared overlay network:

- `c1` does a DNS lookup for `c2`. Since both containers are on the same overlay network the Docker Engine local DNS server resolves `c2` to its overlay IP address `10.0.0.3`.
- An overlay network is a L2 segment so `c1` generates an L2 frame destined for the MAC address of `c2`.
- The frame is encapsulated with a VXLAN header by the `overlay` network driver. The distributed overlay control plane manages the locations and state of each VXLAN tunnel endpoint so it knows that `c2` resides on `host-B` at the physical address of `192.168.0.3`. That address becomes the destination address of the underlay IP header.

- Once encapsulated the packet is sent. The physical network is responsible of routing or bridging the VXLAN packet to the correct host.
- The packet arrives at the `eth0` interface of `host-B` and is decapsulated by the `overlay` network driver. The original L2 frame from `c1` is passed to `c2`'s `eth0` interface and up to the listening application.

Overlay Driver Internal Architecture

The Docker Swarm control plane automates all of the provisioning for an overlay network. No VXLAN configuration or Linux networking configuration is required. Data-plane encryption, an optional feature of overlays, is also automatically configured by the overlay driver as networks are created. The user or network operator only has to define the network (`docker network create -d overlay ...`) and attach containers to that network.



During overlay network creation, Docker Engine creates the network infrastructure required for overlays on each host. A Linux bridge is created per overlay along with its associated VXLAN interfaces. The Docker Engine intelligently instantiates overlay networks on hosts only when a container attached to that network is scheduled on the host. This prevents sprawl of overlay networks where connected containers do not exist.

The following example creates an overlay network and attaches a container to that network. The Docker Swarm/UCP automatically creates the overlay network. *The following example requires Swarm or UCP to be set up beforehand.*

```
#Create an overlay named "ovnet" with the overlay driver
$ docker network create -d overlay --subnet 10.1.0.0/24 ovnet

#Create a service from an nginx image and connect it to the "ovnet" overlay network
$ docker service create --network ovnet nginx
```

When the overlay network is created, notice that several interfaces and bridges are created inside the host as well as two interfaces inside this container.

```
# Peek into the container of this service to see its internal interfaces
conatiner$ ip address

#docker_gwbridge network
52: eth0@if55: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
    link/ether 02:42:ac:14:00:06 brd ff:ff:ff:ff:ff:ff
    inet 172.20.0.6/16 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe14:6/64 scope link
       valid_lft forever preferred_lft forever

#overlay network interface
54: eth1@if53: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450
    link/ether 02:42:0a:01:00:03 brd ff:ff:ff:ff:ff:ff
    inet 10.1.0.3/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet 10.1.0.2/32 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:aff:fe01:3/64 scope link
       valid_lft forever preferred_lft forever
```

Two interfaces have been created inside the container that correspond to two bridges that now exist on the host. On overlay networks, each container has at least two interfaces that connect it to the `overlay` and the `docker_gwbridge` respectively.

Bridge	Purpose

Bridge	Purpose
overlay	The ingress and egress point to the overlay network that VXLAN encapsulates and (optionally) encrypts traffic going between containers on the same overlay network. It extends the overlay across all hosts participating in this particular overlay. One existd per overlay subnet on a host, and it has the same name that a particular overlay network is given.
docker_gwbridge	The egress bridge for traffic leaving the cluster. Only one `docker_gwbridge` exists per host. Container-to-Container traffic is blocked on this bridge allowing ingress/egress traffic flows only.

> The Docker Overlay driver has existed since Docker Engine 1.9, and an external K/V store was required to manage state for the network. Docker Engine 1.12 integrated the control plane state into Docker Engine so that an external store is no longer required. 1.12 also introduced several new features including encryption and service load balancing. Networking features that are introduced require a Docker Engine version that supports them, and using these features with older versions of Docker Engine is not supported.

External Access for Docker Services

Swarm & UCP provide access to services from outside the cluster port publishing. Ingress and egress for services do not depend on centralized gateways, but distributed ingres/egress on the host where the specific service task is running. There are two modes of port publishing for services, `host` mode and `ingress` mode.

Ingress Mode Service Publishing

`ingress` mode port publishing utilizes the Swarm Routing Mesh (https://success.docker.com/api/asset/.%2Frefarch%2Fnetworking%2F#routingmesh) to apply load balancing across the tasks in a service. Ingress mode publishes the exposed port on *every* UCP/Swarm node. Ingress traffic to the published port is load balanced by the Routing Mesh and directed via round robin load balancing to one of the *healthy* tasks of the service. Even if a given host is not running a service task, the port is published on the host and is load balanced to a host that has a task.

```
$ docker service create --replicas 2 --publish mode=ingress,target=80,published=8080 n
ginx
```

> `mode=ingress` is the default mode for services. This command can also be accomplished with the shorthand version `-p 80:8080`. Port `8080` is exposed on every host on the cluster and load balanced to the two containers in this service.

Host Mode Service Publishing

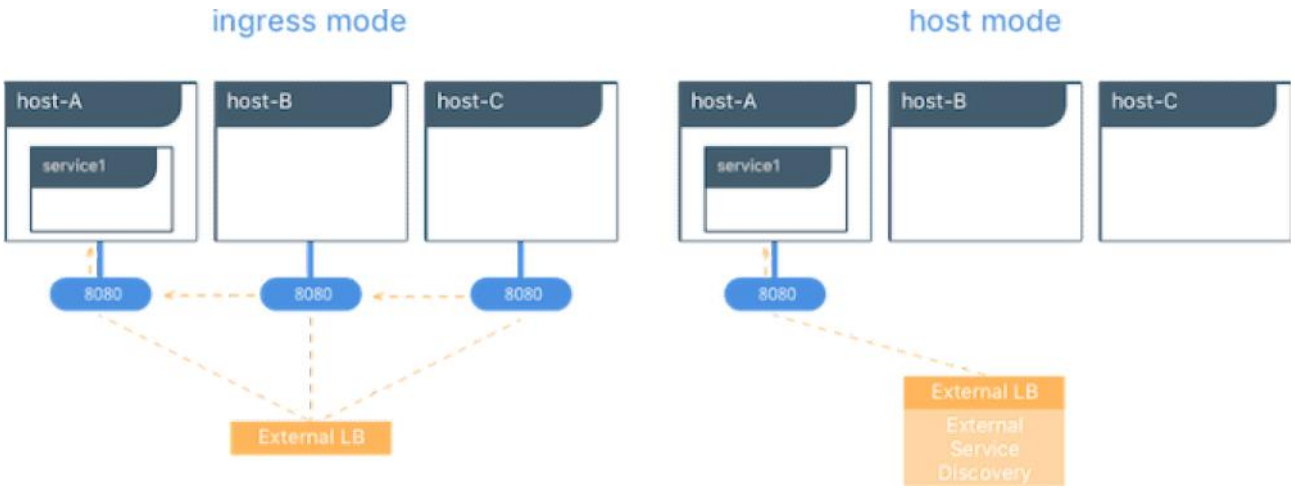
`host` mode port publishing exposes ports only on the host where specific service tasks are running. The port is mapped directly to the container on that host. Only a single task of a given service can run on each host to prevent port collision.

```
$ docker service create --replicas 2 --publish mode=host,target=80,published=8080 ngin
x
```

> `host` mode requires the `mode=host` flag. It publishes port `8080` locally on the hosts where these two containers are running. It does not apply load balancing, so traffic to those nodes are directed only to the local container. This can cause port collision if there are not enough ports available for the number of replicas.

Ingress Design

There are many good use-cases for either publishing mode. `ingress` mode works well for services that have multiple replicas and require load balancing between those replicas. `host` mode works well if external service discovery is already provided by another tool. Another good use case for `host` mode is for global containers that exist once per host. These containers may expose specific information about a the local host (such as monitoring or logging) that are only relevant for that host and so you would not want to load balance when accessing that service.



MACVLAN

The `macvlan` driver is a new implementation of the tried and true network virtualization technique. The Linux implementations are extremely lightweight because rather than using a Linux bridge for isolation, they are simply associated with a Linux Ethernet interface or sub-interface to enforce separation between networks and connectivity to the physical network.

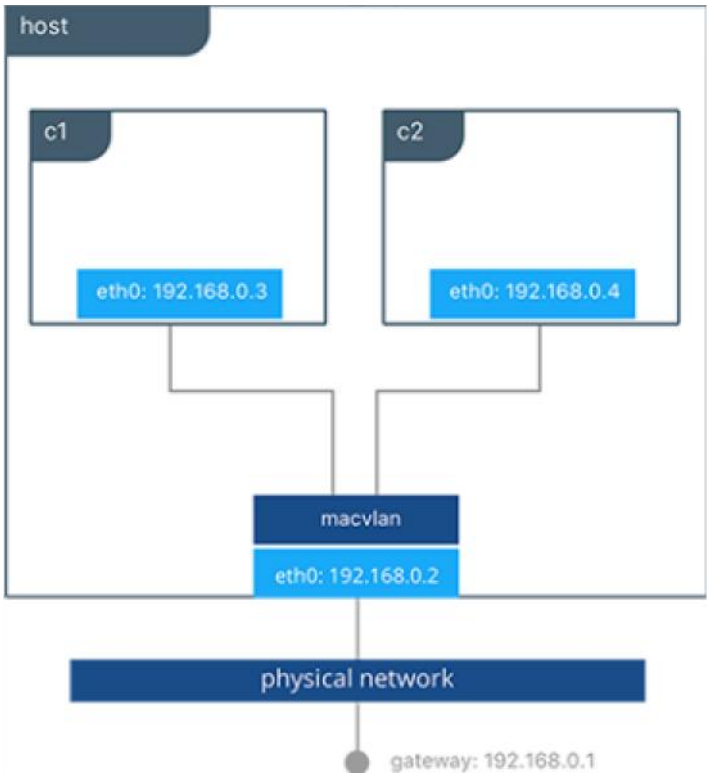
MACVLAN offers a number of unique features and capabilities. It has positive performance implications by virtue of having a very simple and lightweight architecture. Rather than port mapping, the MACVLAN driver provides direct access between containers and the physical network. It also allows containers to receive routable IP addresses that are on the subnet of the physical network.

MACVLAN use-cases may include:

- Very low-latency applications
- Network design that requires containers be on the same subnet as and using IPs as the external host network

The `macvlan` driver uses the concept of a parent interface. This interface can be a physical interface such as `eth0`, a sub-interface for 802.1q VLAN tagging like `eth0.10` (`.10` representing `VLAN 10`), or even a bonded host adaptor which bundles two Ethernet interfaces into a single logical interface.

A gateway address is required during MACVLAN network configuration. The gateway must be external to the host provided by the network infrastructure. MACVLAN networks allow access between containers on the same network. Access between different MACVLAN networks on the same host is not possible without routing outside the host.



This example binds a MACVLAN network to `eth0` on the host. It also attaches two containers to the `mvnet` MACVLAN network and shows that they can ping between themselves. Each container has an address on the `192.168.0.0/24` physical network subnet and its default gateway is an interface in the physical network.

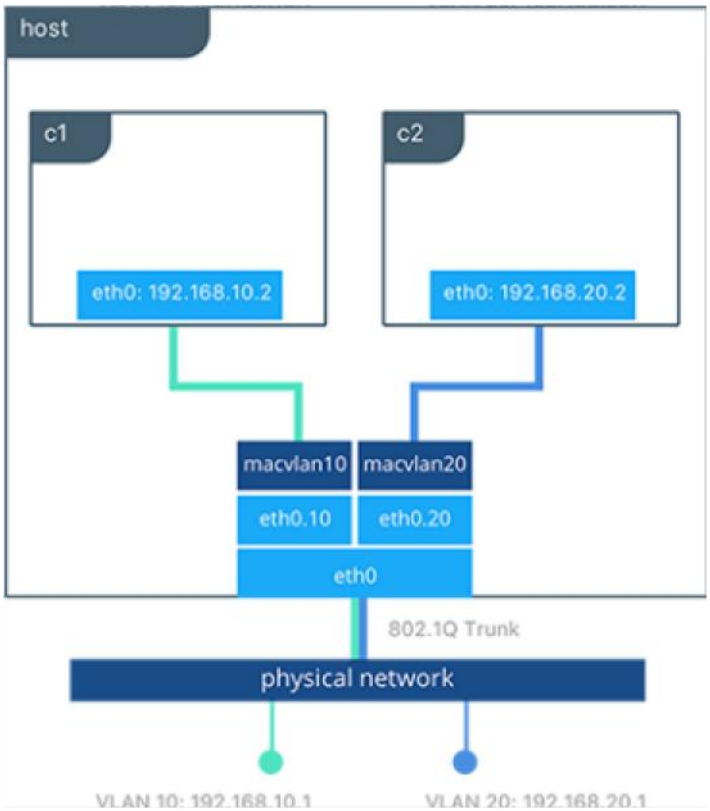
```
#Creation of MACVLAN network "mvnet" bound to eth0 on the host
$ docker network create -d macvlan --subnet 192.168.0.0/24 --gateway 192.168.0.1 -o pa
rent=eth0 mvnet

#Creation of containers on the "mvnet" network
$ docker run -itd --name c1 --net mvnet --ip 192.168.0.3 busybox sh
$ docker run -it --name c2 --net mvnet --ip 192.168.0.4 busybox sh
/ # ping 192.168.0.3
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.052 ms
```

As you can see in this diagram, `c1` and `c2` are attached via the MACVLAN network called `macvlan` attached to `eth0` on the host.

VLAN Trunking with MACVLAN

Trunking 802.1q to a Linux host is notoriously painful for many in operations. It requires configuration file changes in order to be persistent through a reboot. If a bridge is involved, a physical NIC needs to be moved into the bridge, and the bridge then gets the IP address. The `macvlan` driver completely manages sub-interfaces and other components of the MACVLAN network through creation, destruction, and host reboots.



When the `macvlan` driver is instantiated with sub-interfaces it allows VLAN trunking to the host and segments containers at L2. The `macvlan` driver automatically creates the sub-interfaces and connects them to the container interfaces. As a result each container is in a different VLAN, and communication is not possible between them unless traffic is routed in the physical network.

```
#Creation of  macvlan10 network in VLAN 10
$ docker network create -d macvlan --subnet 192.168.10.0/24 --gateway 192.168.10.1 -o
parent=eth0.10 macvlan10

#Creation of  macvlan20 network in VLAN 20
$ docker network create -d macvlan --subnet 192.168.20.0/24 --gateway 192.168.20.1 -o
parent=eth0.20 macvlan20

#Creation of containers on separate MACVLAN networks
$ docker run -itd --name c1--net macvlan10 --ip 192.168.10.2 busybox sh
$ docker run -it --name c2--net macvlan20 --ip 192.168.20.2 busybox sh
```

In the preceding configuration we've created two separate networks using the `macvlan` driver that are configured to use a sub-interface as their parent interface. The `macvlan` driver creates the sub-interfaces and connects them between the host's `eth0` and the container interfaces. The host interface and upstream switch must be set to `switchport mode trunk` so that VLANs are tagged going across the interface. One or more containers can be connected to a given MACVLAN network to create complex network policies that are segmented via L2.

> Because multiple MAC addresses are living behind a single host interface you might need to enable promiscuous mode on the interface depending on the NIC's support for MAC filtering.

None (Isolated) Network Driver

Similar to the `host` network driver, the `none` network driver is essentially an unmanaged networking option. Docker Engine does not create interfaces inside the container, establish port mapping, or install routes for connectivity. A container using `--net=none` is completely isolated from other containers and the host. The networking admin or external tools must be responsible for providing this plumbing. A container using `none` only has a loopback interface and no other interfaces.

Unlike the `host` driver, the `none` driver creates a separate namespace for each container. This guarantees container network isolation between any containers and the host.

> Containers using `--net=none` or `--net=host` cannot be connected to any other Docker networks.

Physical Network Design Requirements

Docker EE and Docker networking are designed to run over common data center network infrastructure and topologies. Its centralized controller and fault-tolerant cluster guarantee compatibility across a wide range of network environments. The components that provide networking functionality (network provisioning, MAC learning, overlay encryption) are either a part of Docker Engine, UCP, or the Linux kernel itself. No extra components or special networking features are required to run any of the native Docker networking drivers.

More specifically, the Docker native network drivers have NO requirements for:

- Multicast
- External key-value stores
- Specific routing protocols
- Layer 2 adjacencies between hosts

- Specific topologies such as spine & leaf, traditional 3-tier, and PoD designs. Any of these topologies are supported.

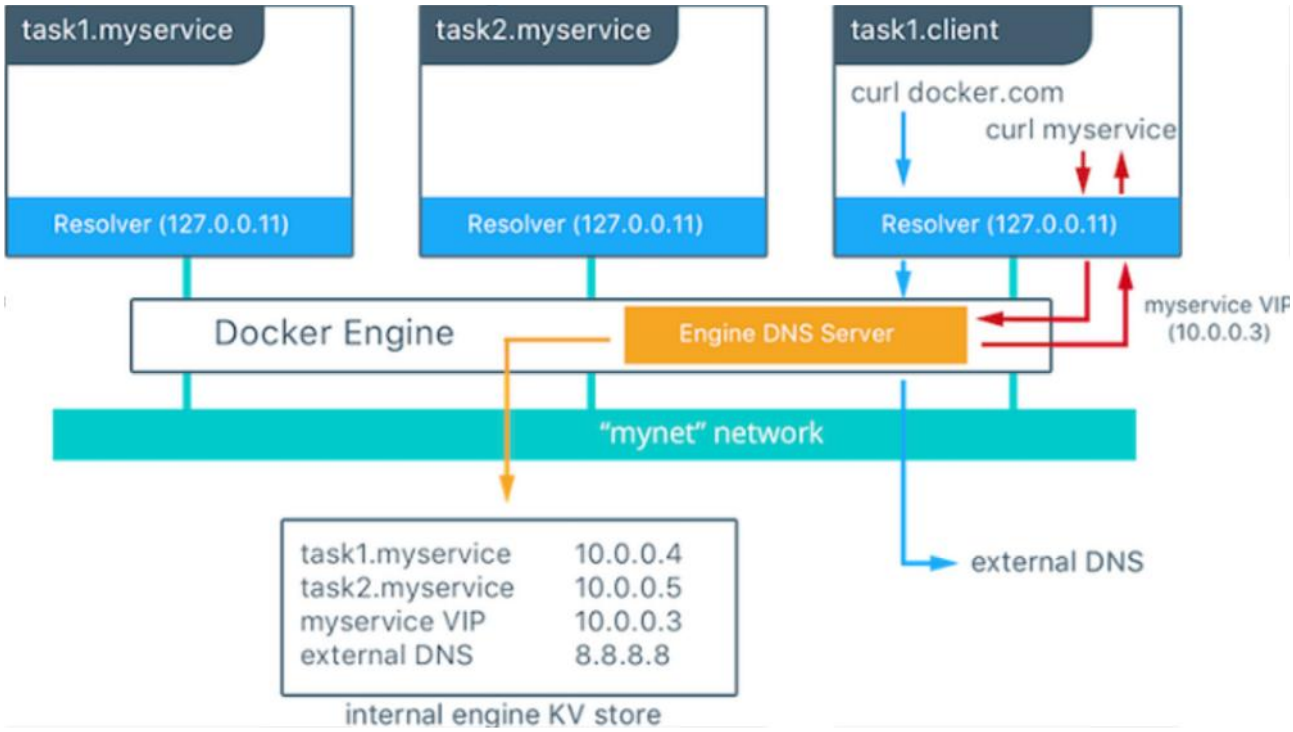
This is in line with the Container Networking Model which promotes application portability across all environments while still achieving the performance and policy required of applications.

Swarm Native Service Discovery

Docker uses embedded DNS to provide service discovery for containers running on a single Docker Engine and `tasks` running in a Docker Swarm. Docker Engine has an internal DNS server that provides name resolution to all of the containers on the host in user-defined bridge, overlay, and MACVLAN networks. Each Docker container (or `task` in Swarm mode) has a DNS resolver that forwards DNS queries to Docker Engine, which acts as a DNS server. Docker Engine then checks if the DNS query belongs to a container or `service` on network(s) that the requesting container belongs to. If it does, then Docker Engine looks up the IP address that matches a container, `task` , or `service` 's name in its key-value store and returns that IP or `service` Virtual IP (VIP) back to the requester.

Service discovery is *network-scoped*, meaning only containers or tasks that are on the same network can use the embedded DNS functionality. Containers not on the same network cannot resolve each other's addresses. Additionally, only the nodes that have containers or tasks on a particular network store that network's DNS entries. This promotes security and performance.

If the destination container or `service` does not belong on the same network(s) as the source container, then Docker Engine forwards the DNS query to the configured default DNS server.



In this example there is a service of two containers called `myservice`. A second service (`client`) exists on the same network. The `client` executes two `curl` operations for `docker.com` and `myservice`. These are the resulting actions:

- DNS queries are initiated by `client` for `docker.com` and `myservice`.
- The container's built-in resolver intercepts the DNS queries on `127.0.0.11:53` and sends them to Docker Engine's DNS server.
- `myservice` resolves to the Virtual IP (VIP) of that service which is internally load balanced to the individual task IP addresses. Container names resolve as well, albeit directly to their IP addresses.
- `docker.com` does not exist as a service name in the `mynet` network and so the request is forwarded to the configured default DNS server.

Docker Native Load Balancing

Docker Swarm clusters have built-in internal and external load balancing capabilities that are built right in to the engine. Internal load balancing provides for load balancing between containers within the same Swarm or UCP cluster. External load balancing provides for the load balancing of ingress traffic entering a cluster.

UCP Internal Load Balancing

Internal load balancing is instantiated automatically when Docker services are created. When services are created in a Docker Swarm cluster, they are automatically assigned a Virtual IP (VIP) that is part of the service's network. The VIP is returned when resolving the service's name. Traffic to that VIP is automatically sent to all healthy

tasks of that service across the overlay network. This approach avoids any client-side load balancing because only a single IP is returned to the client. Docker takes care of routing and equally distributing the traffic across the healthy service tasks.

To see the VIP, run a `docker service inspect my_service` as follows:

```
# Create an overlay network called mynet
$ docker network create -d overlay mynet
a59umzkdj2r0ua7x8jxd84dhr

# Create myservice with 2 replicas as part of that network
$ docker service create --network mynet --name myservice --replicas 2 busybox ping loc
alhost
8t5r8cr0f0h6k2c3k7ih4l6f5

# See the VIP that was created for that service
$ docker service inspect myservice
...

"VirtualIPs": [
                {
                    "NetworkID": "a59umzkdj2r0ua7x8jxd84dhr",
                    "Addr": "10.0.0.3/24"
                },
]
```

> DNS round robin (DNS RR) load balancing is another load balancing option for services (configured with `--endpoint-mode`). In DNS RR mode a VIP is not created for each service. The Docker DNS server resolves a service name to individual container IPs in round robin fashion.

UCP External L4 Load Balancing (Docker Routing Mesh)

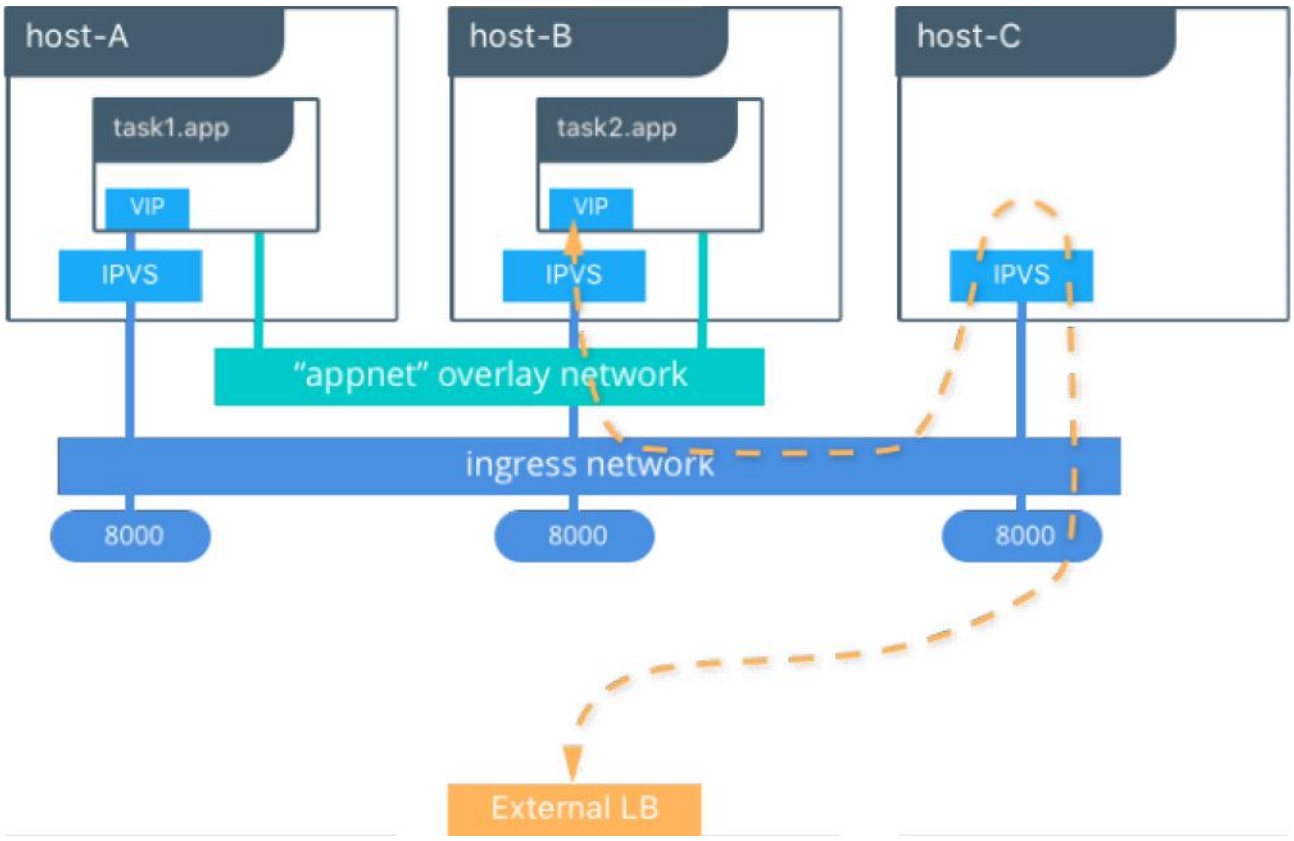
You can expose services externally by using the `--publish` flag when creating or updating the service. Publishing ports in Docker Swarm mode means that every node in your cluster is listening on that port. But what happens if the service's task isn't on the node that is listening on that port?

This is where routing mesh comes into play. Routing mesh is a new feature in Docker 1.12 that combines `ipvs` and `iptables` to create a powerful cluster-wide transport-layer (L4) load balancer. It allows all the Swarm nodes to accept connections on the services' published ports. When any Swarm node receives traffic destined to the published TCP/UDP port of a running `service`, it

forwards it to service's VIP using a pre-defined overlay network called `ingress`. The `ingress` network behaves similarly to other overlay networks but its sole purpose is to transport mesh routing traffic from external clients to cluster services. It uses the same VIP-based internal load balancing as described in the previous section.

Once you launch services, you can create an external DNS record for your applications and map it to any or all Docker Swarm nodes. You do not need to worry about where your container is running as all nodes in your cluster look as one with the routing mesh routing feature.

```
#Create a service with two replicas and export port 8000 on the cluster
$ docker service create --name app --replicas 2 --network appnet -p 8000:80 nginx
```

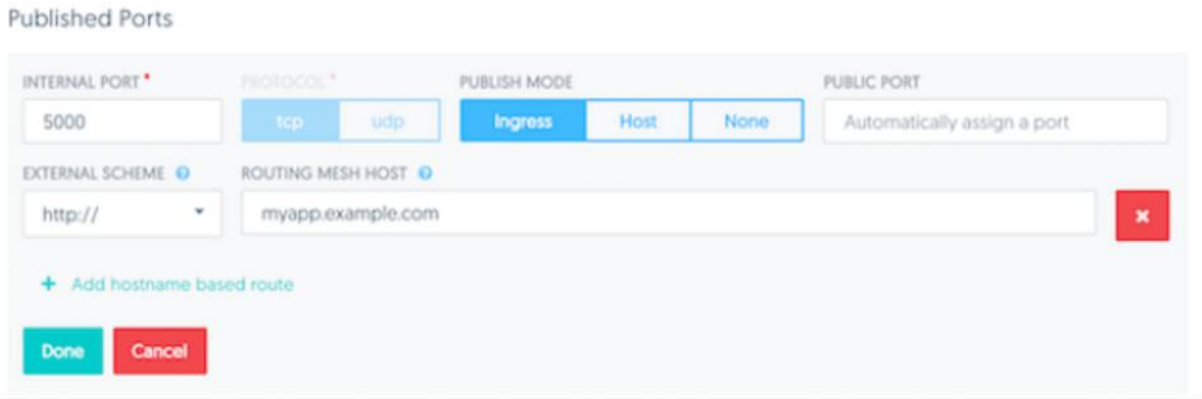


This diagram illustrates how the Routing Mesh works.

- A service is created with two replicas, and it is port mapped externally to port `8000`.
- The routing mesh exposes port `8000` on each host in the cluster.
- Traffic destined for the `app` can enter on any host. In this case the external LB sends the traffic to a host without a service replica.
- The kernel's IPVS load balancer redirects traffic on the `ingress` overlay network to a healthy service replica.

UCP External L7 Load Balancing (HTTP Routing Mesh)

UCP provides L7 HTTP/HTTPS load balancing through the HTTP Routing Mesh. URLs can be load balanced to services and load balanced across the service replicas.



Go to the UCP Load Balancing Reference Architecture (https://success.docker.com/Architecture/Docker_Reference_Architecture%3A_Universal_Control_Plane_2.0_Service_Discovery_and_Load_Balancing) understand more about the UCP L7 LB design.

Docker Network Security and Encryption

Network security is a top-of-mind consideration when designing and implementing containerized workloads with Docker. In this section, key security considerations when deploying Docker networks are covered.

Network Segmentation and Data Plane Security

Docker manages distributed firewall rules to segment Docker networks and prevent malicious access to container resources. By default, Docker networks are segmented from each other to prevent traffic between them. This approach provides true network isolation at Layer 3.

The Docker engine manages host firewall rules that prevent access between networks and manages ports for exposed containers. In a Swarm & UCP clusters this creates a distributed firewall that dynmically protects applications as they are schedued in the cluster.

This table outlines some of the access policies with Docker networks.

Path	Access
Within a Docker Network	Access is permited between all containers on all ports on the same Docker network. This applies for all network types - swarm scope, local scope, built-in, and remote drivers.

Path	Access
Between Docker Networks	Access is denied between Docker networks by distributed host firewall rules that are managed by the Docker engine. Containers can be attached to multiple networks to communicate between different Docker networks. Network connectivity between Docker networks can also be managed external to the host.
Egress From Docker Network	Traffic originating from inside a Docker nework destined for outside a Docker host is permitted. The host's local, stateful firewall tracks connections to permit responses for that connection.
Ingress to Docker Network	Ingress traffic is denied by default. Port exposure through `host` ports or `ingress` mode ports provides explicit ingress access. An exception to this is the MACVLAN driver which operates in the same IP space as the external network and is fully open within that network. Other remote drivers that operate similarly to MACVLAN may also allow ingress traffic.

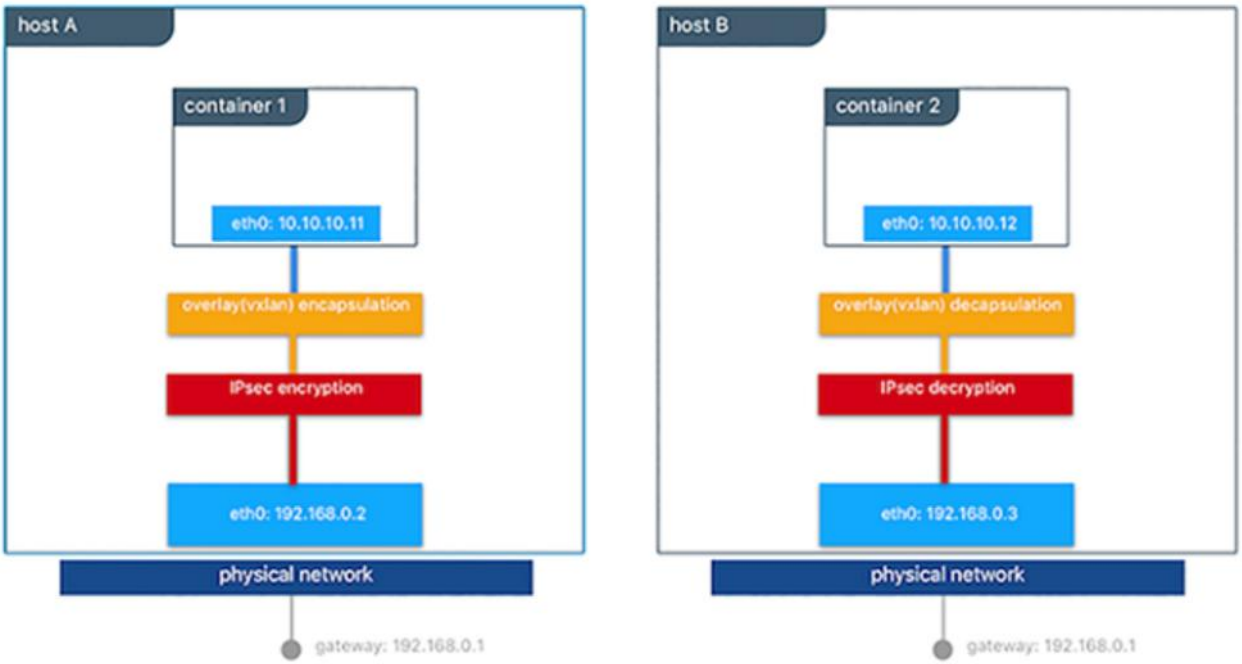
Control Plane Security

Docker Swarm comes with integrated PKI. All managers and nodes in the Swarm have a cryptographically signed identity in the form of a signed certificate. All manager-to-manager and manager-to-node control communication is secured out of the box with TLS. There is no need to generate certs externally or set up any CAs manually to get end-to-end control plane traffic secured in Docker Swarm mode. Certificates are periodically and automatically rotated.

Data Plane Network Encryption

Docker supports IPSec encryption for overlay networks out-of-the-box. The Swarm & UCP managed IPSec tunnels encrypt network traffic as it leaves the source container and decrypts it as it enters the destination container. This ensures that your application traffic is highly secure when it's in transit regardless of the underlying networks. In a hybrid, multi-tenant, or multi-cloud environment, it is crucial to ensure data is secure as it traverses networks you might not have control over.

This diagram illustrates how to secure communication between two containers running on different hosts in a Docker Swarm.

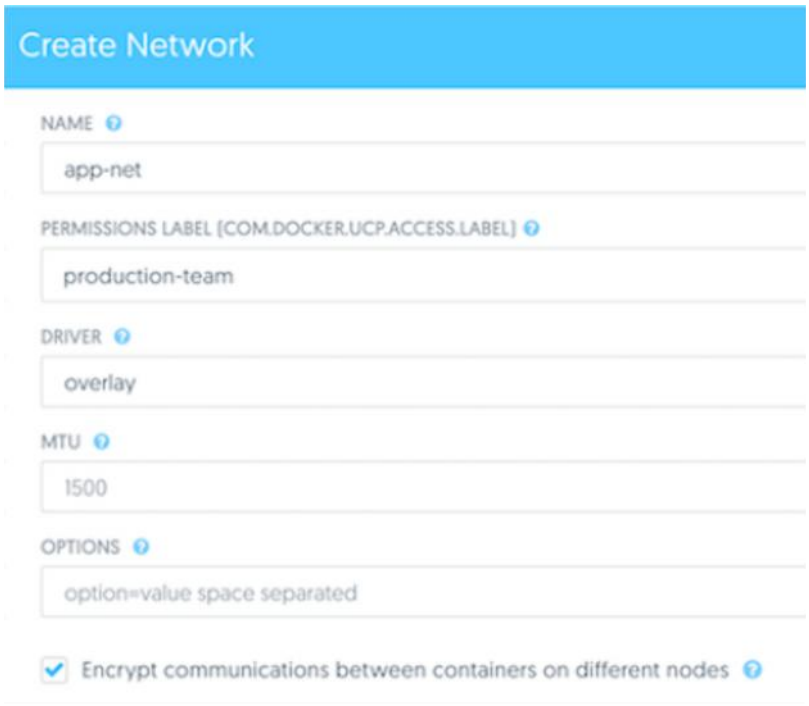


This feature works can be enabled per network at the time of creation by adding the `--opt encrypted=true` option (e.g `docker network create -d overlay --opt encrypted=true <NETWORK_NAME>`). After the network gets created, you can launch services on that network (e.g `docker service create --network <NETWORK_NAME> <IMAGE> <COMMAND>`). When two tasks of the same services are created on two different hosts, an IPsec tunnel is created between them and traffic gets encrypted as it leaves the source host and gets decrypted as it enters the destination host.

The Swarm leader periodically regenerates a symmetrical key and distributes it securely to all cluster nodes. This key is used by IPsec to encrypt and decrypt data plane traffic. The encryption is implemented via IPSec in host-to-host transport mode using AES-GCM.

Management Plane Security & RBAC with UCP

When creating networks with UCP, teams and labels define access to container resources. Resource permission labels define who can view, configure, and use certain Docker networks.



This UCP screenshot shows the use of the label `production-team` to control access to this network to only members of that team. Additionally, options like network encryption can be toggled via UCP.

IP Address Management

The Container Networking Model (CNM) provides flexibility in how IP addresses are managed. There are two methods for IP address management.

- CNM has a native IPAM driver that does simple allocation of IP addresses globally for a cluster and prevents overlapping allocations. The native IPAM driver is what is used by default if no other driver is specified.
- CNM has interfaces to use remote IPAM drivers from other vendors and the community. These drivers can provide integration into existing vendor or self-built IPAM tools.

Manual configuration of container IP addresses and network subnets can be done using UCP, the CLI, or Docker APIs. The address request goes through the chosen driver which then decides how to process the request.

Subnet size and design is largely dependent on a given application and the specific network driver. IP address space design is covered in more depth for each Network Deployment Model (https://success.docker.com/api/asset/.%2Frefarch%2Fnetworking%2F#models) in the next section. The uses of port mapping, overlays, and MACVLAN all have implications on how IP addressing is arranged. In general, container addressing falls into two buckets. Internal container networks (bridge and overlay) address containers with IP addresses that are not routable on the physical network by default. MACVLAN networks provide IP addresses to containers that are on

the subnet of the physical network. Thus, traffic from container interfaces can be routable on the physical network. It is important to note that subnets for internal networks (bridge, overlay) should not conflict with the IP space of the physical underlay network. Overlapping address space can cause traffic to not reach its destination.

Network Troubleshooting

Docker network troubleshooting can be difficult for devops and network engineers. With proper understanding of how Docker networking works and the right set of tools, you can troubleshoot and resolve these network issues. One recommended way is to use the netshoot (https://github.com/nicolaka/netshoot) container to troubleshoot network problems. The `netshoot` container has a set of powerful networking troubleshooting tools that can be used to troubleshoot Docker network issues.

The power of using a troubleshooting container like netshoot is that the network troubleshooting tools are portable. The `netshoot` container can be attached to any network, can be placed in the host network namespace, or in another container's network namespace to inspect any viewpoint of the host network.

It containers the following tools and more:

- iperf
- tcpdump
- netstat
- iftop
- drill
- util-linux(nsenter)
- curl
- nmap

Network Deployment Models

The following example uses a fictional app called Docker Pets (https://github.com/mark-church/docker-pets) to illustrate the Network Deployment Models. It serves up images of pets on a web page while counting the number of hits to the page in a backend database.

- `web` is a front-end web server based on the `chrch/docker-pets:1.0` image
- `db` is a `consul` backend

`chrch/docker-pets` expects an environment variable `DB` that tells it how to find the backend `db` service.

Bridge Driver on a Single Host

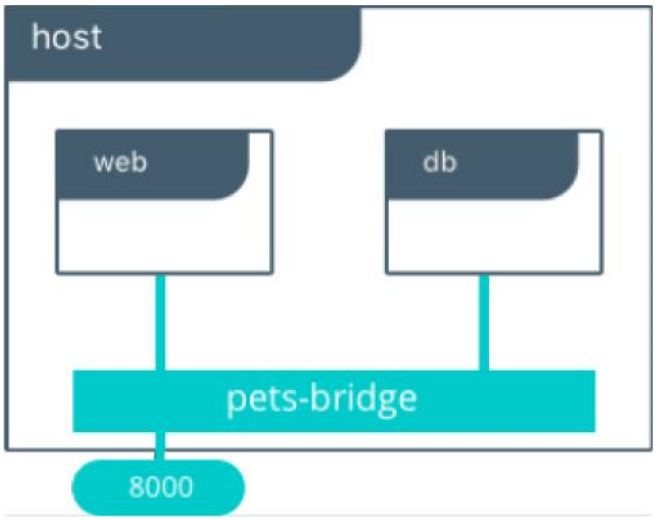
This model is the default behavior of the native Docker `bridge` network driver. The `bridge` driver creates a private network internal to the host and provides an external port mapping on a host interface for external connectivity.

```
$ docker network create -d bridge petsBridge

$ docker run -d --net petsBridge --name db consul

$ docker run -it --env "DB=db" --net petsBridge --name web -p 8000:5000 chrch/docker-p
ets:1.0
Starting web container e750c649a6b5
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

> When an IP address is not specified, port mapping is exposed on all interfaces of a host. In this case the container's application is exposed on `0.0.0.0:8000` . To provide a specific IP address to advertise on use the flag `-p IP:host_port:container_port` . More options to expose ports can be found in the Docker docs (https://docs.docker.com/engine/reference/run/#/expose-incoming-ports).



The application is exposed locally on this host on port `8000` on all of its interfaces. Also supplied is `DB=db` , providing the name of the backend container. The Docker Engine's built-in DNS resolves this container name to the IP address of `db` . Since `bridge` is a local driver, the scope of DNS resolution is only on a single host.

The output below shows us that our containers have been assigned private IPs from the `172.19.0.0/24` IP space of the `petsBridge` network. Docker uses the built-in IPAM driver to provide an IP from the appropriate subnet if no other IPAM driver is specified.

```
$ docker inspect --format {{.NetworkSettings.Networks.petsBridge.IPAddress}} web
172.19.0.3

$ docker inspect --format {{.NetworkSettings.Networks.petsBridge.IPAddress}} db
172.19.0.2
```

These IP addresses are used internally for communication internal to the `petsBridge` network. These IPs are never exposed outside of the host.

Multi-Host Bridge Driver with External Service Discovery

Because the `bridge` driver is a local scope driver, multi-host networking requires a multi-host service discovery solution. External SD registers the location and status of a container or service and then allows other services to discover that location. Because the bridge driver exposes ports for external access, external SD stores the `host-ip:port` as the location of a given container.

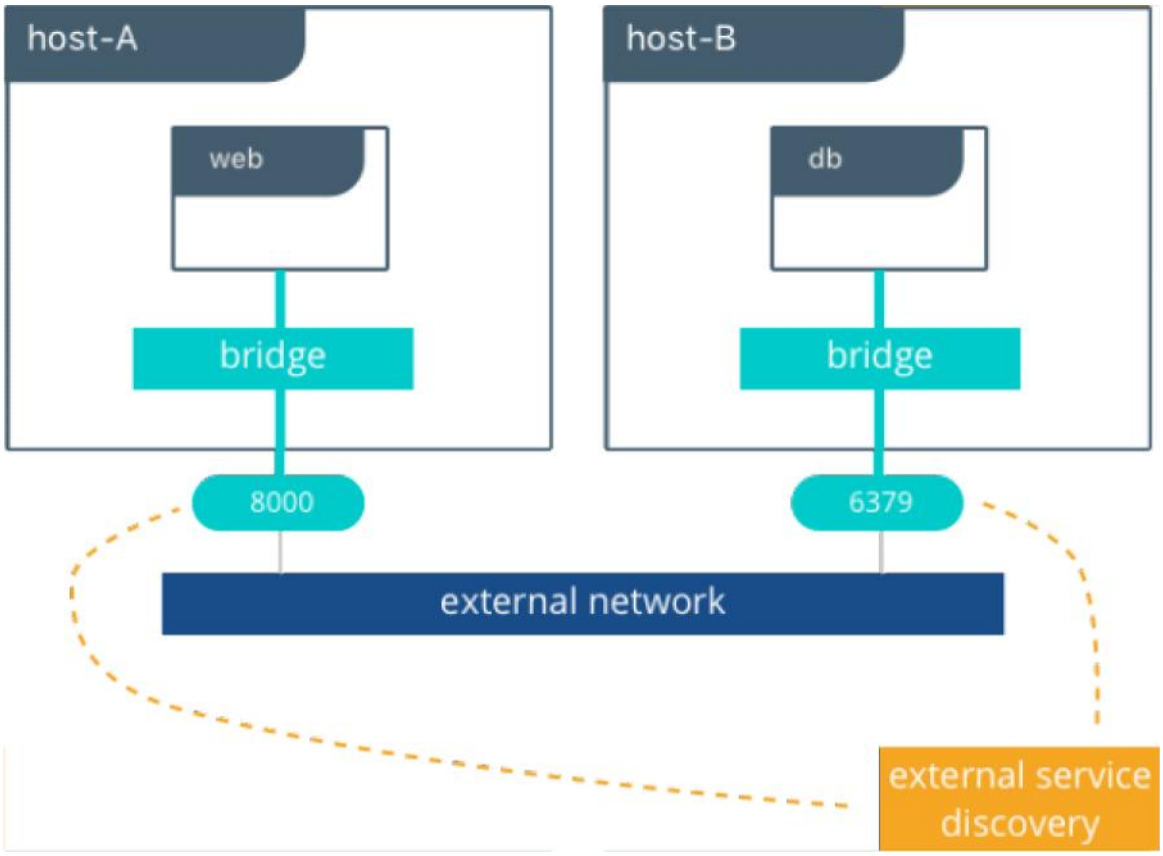
In the following example, the location of each service is manually configured, simulating external service discovery. The location of the `db` service is passed to `web` via the `DB` environment variable.

```
#Create the backend db service and expose it on port 8500
host-A $ docker run -d -p 8500:8500 --name db consul

#Display the host IP of host-A
host-A $ ip add show eth0 | grep inet
    inet 172.31.21.237/20 brd 172.31.31.255 scope global eth0
    inet6 fe80::4db:c8ff:fea0:b129/64 scope link

#Create the frontend web service and expose it on port 8000 of host-B
host-B $ docker run -d -p 8000:5000 -e 'DB=172.31.21.237:8500' --name web chrch/docker
-pets:1.0
```

The `web` service should now be serving its web page on port `8000` of `host-B` IP address.



> In this example we don't specify a network to use, so the default Docker `bridge` network is selected automatically.

When we configure the location of `db` at `172.31.21.237:8500`, we are creating a form of service discovery. We are statically configuring the location of the `db` service for the `web` service. In the single host example, this was done automatically because Docker Engine provided built-in DNS resolution for the container names. In this multi-host example we are doing the service discovery manually.

The hardcoding of application location is not recommended for production. External service discovery tools exist that provide these mappings dynamically as containers are created and destroyed in a cluster. Some examples are Consul (https://www.consul.io/) and etcd (https://coreos.com/etcd/).

The next section examines the `overlay` driver scenario, which provides global service discovery across a cluster as a built-in feature. This simplicity is a major advantage of the `overlay` driver, as opposed to using multiple external tools to provide network services.

Multi-Host with Overlay Driver

This model utilizes the native `overlay` driver to provide multi-host connectivity out of the box. The default settings of the overlay driver provide external connectivity to the outside world as well as internal connectivity and service discovery within a container application. The Overlay Driver Architecture (https://success.docker.com/api/asset/.%2Frefarch%2Fnetworking%2F#overlayarch) section reviews the internals of the Overlay driver which you should review before reading this section.

This example re-uses the previous `docker-pets` application. Set up a Docker swarm prior to following this example. For instructions on how to set up a Swarm read the Docker docs (https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/). After the Swarm is set up, use the `docker service create` command to create containers and networks to be managed by the Swarm.

The following shows how to inspect your Swarm, create an overlay network, and then provision some services on that overlay network. All of these commands are run on a UCP/swarm controller node.

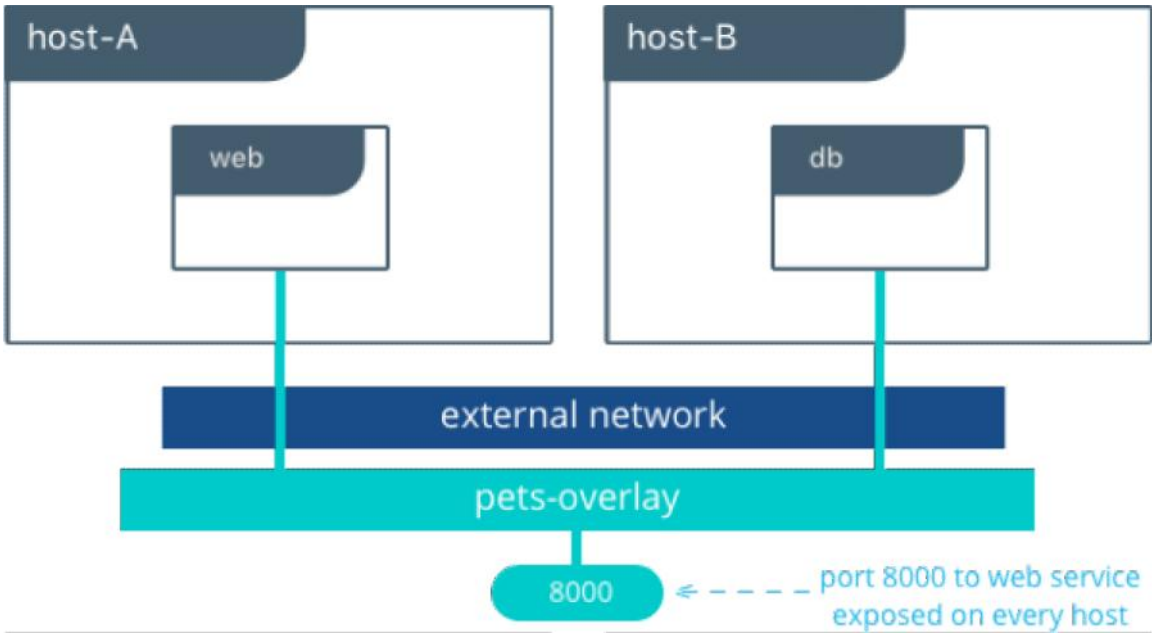
```
#Display the nodes participating in this swarm cluster that was already created
$ docker node ls
ID                           HOSTNAME          STATUS  AVAILABILITY  MANAGER STATUS
a8dwuh6gy5898z3yeuvxaetjo    host-B  Ready   Active
elgt0bfuikjrntv3c33hr0752 *  host-A  Ready   Active        Leader

#Create the dognet overlay network
host-A $ docker network create -d overlay petsOverlay

#Create the backend service and place it on the dognet network
host-A $ docker service create --network petsOverlay --name db consul

#Create the frontend service and expose it on port 8000 externally
host-A $ docker service create --network petsOverlay -p 8000:5000 -e 'DB=db' --name we
b chrch/docker-pets:1.0

host-A $ docker service ls
ID            NAME  MODE        REPLICAS  IMAGE
lxnjfo2dnjxq  db    replicated  1/1       consul:latest
t222cnez6n7h  web   replicated  0/1       chrch/docker-pets:1.0
```



As in the single-host bridge example, we pass in `DB=db` as an environment variable to the `web` service. The overlay driver resolves the service name `db` to the overlay IP address of the container. Communication between `web` and `db` occurs exclusively using the overlay IP subnet.

> Inside overlay and bridge networks, all TCP and UDP ports to containers are open and accessible to all other containers attached to the overlay network.

The `web` service is exposed on port `8000`, and the routing mesh exposes port `8000` on every host in the Swarm cluster. Test if the application is working by going to `<host-A>:8000` or `<host-B>:8000` in the browser.

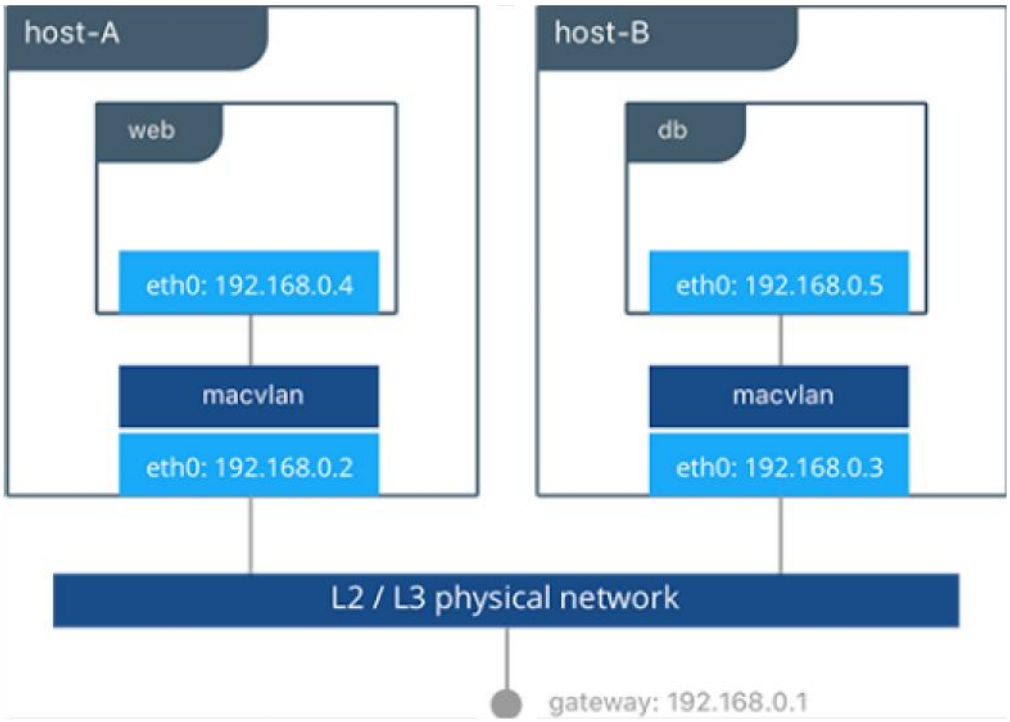
Overlay Benefits and Use Cases

- Very simple multi-host connectivity for small and large deployments
- Provides service discovery and load balancing with no extra configuration or components
- Useful for east-west micro-segmentation via encrypted overlays
- Routing mesh can be used to advertise a service across an entire cluster

Tutorial App: MACVLAN Bridge Mode

There may be cases where the application or network environment requires containers to have routable IP addresses that are a part of the underlay subnets. The MACVLAN driver provides an implementation that makes this possible. As described in the MACVLAN Architecture section (https://success.docker.com/api/asset/.%2Frefarch%2Fnetworking%2F#macvlan), a MACVLAN network binds itself to a host interface. This can be a physical interface, a logical sub-interface, or a

bonded logical interface. It acts as a virtual switch and provides communication between containers on the same MACVLAN network. Each container receives a unique MAC address and an IP address of the physical network that the node is attached to.



In this example, the Pets application is deployed on to `host-A` and `host-B`.

```
#Creation of local macvlan network on both hosts
host-A $ docker network create -d macvlan --subnet 192.168.0.0/24 --gateway 192.168.0.
1 -o parent=eth0 petsMacvlan
host-B $ docker network create -d macvlan --subnet 192.168.0.0/24 --gateway 192.168.0.
1 -o parent=eth0 petsMacvlan

#Creation of db container on host-B
host-B $ docker run -d --net petsMacvlan --ip 192.168.0.5 --name db consul

#Creation of web container on host-A
host-A $ docker run -it --net petsMacvlan --ip 192.168.0.4 -e 'DB=192.168.0.5:8500' --
name web chrch/docker-pets:1.0
```

This may look very similar to the multi-host bridge example but there are a couple notable differences:

- The reference from `web` to `db` uses the IP address of `db` itself as oppossed to the host IP. Remember that with `macvlan` container IPs are routable on the underlay network.
- We do not expose any ports for `db` or `web` because any ports opened in the container are immediately be reachable using the container IP address.

While the `macvlan` driver offers these unique advantages, one area that it sacrifices is portability. MACVLAN configuration and deployment is heavily tied to the underlay network. Container addressing must adhere to the physical location of container placement in addition to preventing overlapping address assignment. Because of this, care must be taken to manage IPAM externally to a MACVLAN network. Overlapping IP addressing or incorrect subnets can lead to loss of container connectivity.

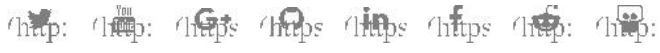
MACVLAN Benefits and Use Cases

- Very low latency applications can benefit from the `macvlan` driver because it does not utilize NAT.
- MACVLAN can provide an IP per container, which may be a requirement in some environments.
- More careful consideration for IPAM must be taken in to account.

Conclusion

Docker is a quickly evolving technology, and the networking options are growing to satisfy more and more use cases every day. Incumbent networking vendors, pure-play SDN vendors, and Docker itself are all contributors to this space. Tighter integration with the physical network, network monitoring, and encryption are all areas of much interest and innovation.

This document detailed some but not all of the possible deployments and CNM network drivers that exist. While there are many individual drivers and even more ways to configure those drivers, we hope you can see that there are only a few common models routinely deployed. Understanding the tradeoffs with each model is key to long term success.



Overview

Estimated reading time: 4 minutes

One of the reasons Docker containers and services are so powerful is that you can connect them together, or connect them to non-Docker workloads. Docker containers and services do not even need to be aware that they are deployed on Docker, or whether their peers are also Docker workloads or not. Whether your Docker hosts run Linux, Windows, or a mix of the two, you can use Docker to manage them in a platform-agnostic way.

This topic defines some basic Docker networking concepts and prepares you to design and deploy your applications to take full advantage of these capabilities.

Most of this content applies to all Docker installations. However, a few advanced features (/network/#docker-ee-networking-features) are only available to Docker EE customers.

Scope of this topic

This topic does not go into OS-specific details about how Docker networks work, so you will not find information about how Docker manipulates `iptables` rules on Linux or how it manipulates routing rules on Windows servers, and you will not find detailed information about how Docker forms and encapsulates packets or handles encryption. See Docker and iptables (https://docs.docker.com/network/iptables/) and Docker Reference Architecture: Designing Scalable, Portable Docker Container Networks (http://success.docker.com/article/networking) for a much greater depth of technical detail.

In addition, this topic does not provide any tutorials for how to create, manage, and use Docker networks. Each section includes links to relevant tutorials and command references.

Network drivers

Docker's networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

- `bridge` : The default network driver. If you don't specify a driver, this is the type of network you are creating. Bridge networks are usually used when your applications run in standalone containers that need to communicate. See bridge networks (https://docs.docker.com/network/bridge/).
- `host` : For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly. `host` is only available for swarm services on Docker 17.06 and higher. See use the host network (https://docs.docker.com/network/host/).
- `overlay` : Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers. See overlay networks (https://docs.docker.com/network/overlay/).
- `macvlan` : Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the `macvlan` driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack. See Macvlan networks (https://docs.docker.com/network/macvlan/).
- `none` : For this container, disable all networking. Usually used in conjunction with a custom network driver. `none` is not available for swarm services. See disable container networking (https://docs.docker.com/network/none/).
- Network plugins (https://docs.docker.com/engine/extend/plugins_services/): You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub (https://hub.docker.com/search?category=network&q=&type=plugin) or from third-party vendors. See the vendor's documentation for installing and using a given network plugin.

Network driver summary

- User-defined bridge networks are best when you need multiple containers to communicate on the same Docker host.
- Host networks are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.
- Overlay networks are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
- Macvlan networks are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.
- Third-party network plugins allow you to integrate Docker with specialized network stacks.

Docker EE networking features

The following two features are only possible when using Docker EE and managing your Docker services using Universal Control Plane (UCP):

- The HTTP routing mesh (https://docs.docker.com/datacenter/ucp/2.2/guides/admin/configure/use-domain-names-to-access-services/) allows you to share the same network IP address and port among multiple services. UCP routes the traffic to the appropriate service using the combination of hostname and port, as requested from the client.

- Session stickiness (https://docs.docker.com/datacenter/ucp/2.2/guides/user/services/use-domain-names-to-access-services/#sticky-sessions) allows you to specify information in the HTTP header which UCP uses to route subsequent requests to the same service task, for applications which require stateful sessions.

Networking tutorials

Now that you understand the basics about Docker networks, deepen your understanding using the following tutorials:

- Standalone networking tutorial (https://docs.docker.com/network/network-tutorial-standalone/)

- Host networking tutorial (https://docs.docker.com/network/network-tutorial-host/)
- Overlay networking tutorial (https://docs.docker.com/network/network-tutorial-overlay/)
- Macvlan networking tutorial (https://docs.docker.com/network/network-tutorial-macvlan/)

networking (https://docs.docker.com/glossary/?term=networking), bridge (https://docs.docker.com/glossary/?term=bridge), routing (https://docs.docker.com/glossary/?term=routing), routing mesh (https://docs.docker.com/glossary/?term=routing mesh), overlay (https://docs.docker.com/glossary/?term=overlay), ports (https://docs.docker.com/glossary/?term=ports)

dockerd

Estimated reading time: 54 minutes

daemon

```
Usage:  dockerd COMMAND

A self-sufficient runtime for containers.

Options:
      --add-runtime runtime                    Register an additional
 OCI compatible runtime (default [])
      --allow-nondistributable-artifacts list Push nondistributable
artifacts to specified registries (default [])
      --api-cors-header string                 Set CORS headers in th
e Engine API
      --authorization-plugin list              Authorization plugins
to load (default [])
      --bip string                             Specify network bridge
 IP
  -b, --bridge string                          Attach containers to a
 network bridge
      --cgroup-parent string                   Set parent cgroup for
all containers
      --cluster-advertise string               Address or interface n
ame to advertise
      --cluster-store string                   URL of the distributed
 storage backend
      --cluster-store-opt map                  Set cluster store opti
ons (default map[])
      --config-file string                     Daemon configuration f
ile (default "/etc/docker/daemon.json")
      --containerd string                      Path to containerd soc
ket
      --cpu-rt-period int                      Limit the CPU real-tim
e period in microseconds
      --cpu-rt-runtime int                     Limit the CPU real-tim
e runtime in microseconds
      --data-root string                       Root directory of pers
istent Docker state (default "/var/lib/docker")
  -D, --debug                                  Enable debug mode
      --default-gateway ip                     Container default gate
way IPv4 address
      --default-gateway-v6 ip                  Container default gate
way IPv6 address
      --default-address-pool                   Set the default addres
s pool for local node networks
      --default-runtime string                 Default OCI runtime fo
r containers (default "runc")
      --default-ulimit ulimit                  Default ulimits for co
ntainers (default [])
      --dns list                               DNS server to use (def
ault [])
```

```
      --dns-opt list                         DNS options to use (de
fault [])
      --dns-search list                      DNS search domains to
use (default [])
      --exec-opt list                        Runtime execution opti
ons (default [])
      --exec-root string                     Root directory for exe
cution state files (default "/var/run/docker")
      --experimental                         Enable experimental fe
atures
      --fixed-cidr string                    IPv4 subnet for fixed
IPs
      --fixed-cidr-v6 string                 IPv6 subnet for fixed
IPs
  -G, --group string                         Group for the unix soc
ket (default "docker")
      --help                                 Print usage
  -H, --host list                            Daemon socket(s) to co
nnect to (default [])
      --icc                                  Enable inter-container
 communication (default true)
      --init                                 Run an init in the con
tainer to forward signals and reap processes
      --init-path string                     Path to the docker-ini
t binary
      --insecure-registry list               Enable insecure regist
ry communication (default [])
      --ip ip                                Default IP when bindin
g container ports (default 0.0.0.0)
      --ip-forward                           Enable net.ipv4.ip_for
ward (default true)
      --ip-masq                              Enable IP masquerading
 (default true)
      --iptables                             Enable addition of ipt
ables rules (default true)
      --ipv6                                 Enable IPv6 networking
      --label list                           Set key=value labels t
o the daemon (default [])
      --live-restore                         Enable live restore of
 docker when containers are still running
      --log-driver string                    Default driver for con
tainer logs (default "json-file")
  -l, --log-level string                     Set the logging level
("debug", "info", "warn", "error", "fatal") (default "info")
      --log-opt map                          Default log driver opt
ions for containers (default map[])
      --max-concurrent-downloads int         Set the max concurrent
 downloads for each pull (default 3)
      --max-concurrent-uploads int           Set the max concurrent
```

```
 uploads for each push (default 5)
      --metrics-addr string                  Set default address an
d port to serve the metrics api on
      --mtu int                              Set the containers net
work MTU
      --node-generic-resources list          Advertise user-defined
 resource
      --no-new-privileges                    Set no-new-privileges
by default for new containers
      --oom-score-adjust int                 Set the oom_score_adj
for the daemon (default -500)
  -p, --pidfile string                       Path to use for daemon
 PID file (default "/var/run/docker.pid")
      --raw-logs                             Full timestamps withou
t ANSI coloring
      --registry-mirror list                 Preferred Docker regis
try mirror (default [])
      --seccomp-profile string               Path to seccomp profil
e
      --selinux-enabled                      Enable selinux support
      --shutdown-timeout int                 Set the default shutdo
wn timeout (default 15)
  -s, --storage-driver string                Storage driver to use
      --storage-opt list                     Storage driver options
 (default [])
      --swarm-default-advertise-addr string  Set default address or
 interface for swarm advertised address
      --tls                                  Use TLS; implied by --
tlsverify
      --tlscacert string                     Trust certs signed onl
y by this CA (default "~/.docker/ca.pem")
      --tlscert string                       Path to TLS certificat
e file (default "~/.docker/cert.pem")
      --tlskey string                        Path to TLS key file (
default ~/.docker/key.pem")
      --tlsverify                            Use TLS and verify the
 remote
      --userland-proxy                       Use userland proxy for
 loopback traffic (default true)
      --userland-proxy-path string           Path to the userland p
roxy binary
      --userns-remap string                  User/Group setting for
 user namespaces
  -v, --version                              Print version informat
ion and quit
```

Options with [] may be specified multiple times.

Description

`dockerd` is the persistent process that manages containers. Docker uses different binaries for the daemon and client. To run the daemon you type `dockerd` .

To run the daemon with debug output, use `dockerd -D` or add `"debug": true` to the `daemon.json` file.

> Note: In Docker 1.13 and higher, enable experimental features by starting `dockerd` with the `--experimental` flag or adding `"experimental": true` to the `daemon.json` file. In earlier Docker versions, a different build was required to enable experimental features.

Examples

Daemon socket option

The Docker daemon can listen for Docker Engine API (https://docs.docker.com/engine/reference/api/) requests via three different types of Socket: `unix` , `tcp` , and `fd` .

By default, a `unix` domain socket (or IPC socket) is created at `/var/run/docker.sock` , requiring either `root` permission, or `docker` group membership.

If you need to access the Docker daemon remotely, you need to enable the `tcp` Socket. Beware that the default setup provides un-encrypted and un-authenticated direct access to the Docker daemon - and should be secured either using the built in HTTPS encrypted socket (https://docs.docker.com/engine/security/https/), or by putting a secure web proxy in front of it. You can listen on port `2375` on all network interfaces with `-H tcp://0.0.0.0:2375` , or on a particular network interface using its IP address: `-H tcp://192.168.59.103:2375` . It is conventional to use port `2375` for un-encrypted, and port `2376` for encrypted communication with the daemon.

> Note: If you're using an HTTPS encrypted socket, keep in mind that only TLS1.0 and greater are supported. Protocols SSLv3 and under are not supported anymore for security reasons.

On Systemd based systems, you can communicate with the daemon via Systemd socket activation (http://0pointer.de/blog/projects/socket-activation.html), use `dockerd -H fd://`. Using `fd://` will work perfectly for most setups but you can also specify individual sockets: `dockerd -H fd://3`. If the specified socket activated files aren't found, then Docker will exit. You can find examples of using Systemd socket activation with Docker and Systemd in the Docker source tree (https://github.com/docker/docker/tree/master/contrib/init/systemd/).

You can configure the Docker daemon to listen to multiple sockets at the same time using multiple `-H` options:

```
# listen using the default unix socket, and on 2 specific IP address
es on this host.

$ sudo dockerd -H unix:///var/run/docker.sock -H tcp://192.168.59.10
6 -H tcp://10.10.10.2
```

The Docker client will honor the `DOCKER_HOST` environment variable to set the `-H` flag for the client. Use one of the following commands:

```
$ docker -H tcp://0.0.0.0:2375 ps
```

```
$ export DOCKER_HOST="tcp://0.0.0.0:2375"

$ docker ps
```

Setting the `DOCKER_TLS_VERIFY` environment variable to any value other than the empty string is equivalent to setting the `--tlsverify` flag. The following are equivalent:

```
$ docker --tlsverify ps
# or
$ export DOCKER_TLS_VERIFY=1
$ docker ps
```

The Docker client will honor the `HTTP_PROXY`, `HTTPS_PROXY`, and `NO_PROXY` environment variables (or the lowercase versions thereof). `HTTPS_PROXY` takes precedence over `HTTP_PROXY`.

Starting with Docker 18.09, the Docker client supports connecting to a remote daemon via SSH:

```
$ docker -H ssh://me@example.com:22 ps
$ docker -H ssh://me@example.com ps
$ docker -H ssh://example.com ps
```

To use SSH connection, you need to set up `ssh` so that it can reach the remote host with public key authentication. Password authentication is not supported. If your key is protected with passphrase, you need to set up `ssh-agent`.

Also, you need to have `docker` binary 18.09 or later on the daemon host.

BIND DOCKER TO ANOTHER HOST/PORT OR A UNIX SOCKET

> Warning: Changing the default `docker` daemon binding to a TCP port or Unix *docker* user group will increase your security risks by allowing non-root users to gain *root* access on the host. Make sure you control access to `docker`. If you are binding to a TCP port, anyone with access to that port has full Docker access; so it is not advisable on an open network.

With `-H` it is possible to make the Docker daemon to listen on a specific IP and port. By default, it will listen on `unix:///var/run/docker.sock` to allow only local connections by the *root* user. You *could* set it to `0.0.0.0:2375` or a specific host IP to give access to everybody, but that is not recommended because then it is trivial for someone to gain root access to the host where the daemon is running.

Similarly, the Docker client can use `-H` to connect to a custom port. The Docker client will default to connecting to `unix:///var/run/docker.sock` on Linux, and `tcp://127.0.0.1:2376` on Windows.

`-H` accepts host and port assignment in the following format:

```
tcp://[host]:[port][path] or unix://path
```

For example:

- `tcp://` -> TCP connection to `127.0.0.1` on either port `2376` when TLS encryption is on, or port `2375` when communication is in plain text.
- `tcp://host:2375` -> TCP connection on host:2375
- `tcp://host:2375/path` -> TCP connection on host:2375 and prepend path to all requests
- `unix://path/to/socket` -> Unix socket located at `path/to/socket`

`-H` , when empty, will default to the same value as when no `-H` was passed in.

`-H` also accepts short form for TCP bindings: `host:` or `host:port` or `:port`

Run Docker in daemon mode:

```
$ sudo <path to>/dockerd -H 0.0.0.0:5555 &
```

Download an `ubuntu` image:

```
$ docker -H :5555 pull ubuntu
```

You can use multiple `-H` , for example, if you want to listen on both TCP and a Unix socket

```
# Run docker in daemon mode
$ sudo <path to>/dockerd -H tcp://127.0.0.1:2375 -H unix:///var/run/
docker.sock &
# Download an ubuntu image, use default Unix socket
$ docker pull ubuntu
# OR use the TCP port
$ docker -H tcp://127.0.0.1:2375 pull ubuntu
```

Daemon storage-driver

On Linux, the Docker daemon has support for several different image layer storage drivers: `aufs`, `devicemapper`, `btrfs`, `zfs`, `overlay` and `overlay2`.

The `aufs` driver is the oldest, but is based on a Linux kernel patch-set that is unlikely to be merged into the main kernel. These are also known to cause some serious kernel crashes. However `aufs` allows containers to share executable and shared library memory, so is a useful choice when running thousands of containers with the same program or libraries.

The `devicemapper` driver uses thin provisioning and Copy on Write (CoW) snapshots. For each devicemapper graph location – typically `/var/lib/docker/devicemapper` – a thin pool is created based on two block devices, one for data and one for metadata. By default, these block devices are created automatically by using loopback mounts of automatically created sparse files. Refer to Devicemapper options (/engine/reference/commandline/dockerd/#devicemapper-options) below for a way how to customize this setup. ~jpetazzo/Resizing Docker containers with the Device Mapper plugin (http://jpetazzo.github.io/2014/01/29/docker-device-mapper-resize/) article explains how to tune your existing setup without the use of options.

The `btrfs` driver is very fast for `docker build` - but like `devicemapper` does not share executable memory between devices. Use `dockerd -s btrfs -g /mnt/btrfs_partition`.

The `zfs` driver is probably not as fast as `btrfs` but has a longer track record on stability. Thanks to `Single Copy ARC` shared blocks between clones will be cached only once. Use `dockerd -s zfs`. To select a different zfs filesystem set `zfs.fsname` option as described in ZFS options (/engine/reference/commandline/dockerd/#zfs-options).

The `overlay` is a very fast union filesystem. It is now merged in the main Linux kernel as of 3.18.0 (https://lkml.org/lkml/2014/10/26/137). `overlay` also supports page cache sharing, this means multiple containers accessing the same file can share a single page cache entry (or entries), it makes `overlay` as efficient with memory as `aufs` driver. Call `dockerd -s overlay` to use it.

> Note: As promising as `overlay` is, the feature is still quite young and should not be used in production. Most notably, using `overlay` can cause excessive inode consumption (especially as the number of images grows), as well as > being incompatible with the use of RPMs.

The `overlay2` uses the same fast union filesystem but takes advantage of additional features (https://lkml.org/lkml/2015/2/11/106) added in Linux kernel 4.0 to avoid excessive inode consumption. Call `dockerd -s overlay2` to use it.

> Note: Both `overlay` and `overlay2` are currently unsupported on `btrfs` or any Copy on Write filesystem and should only be used over `ext4` partitions.

On Windows, the Docker daemon supports a single image layer storage driver depending on the image platform: `windowsfilter` for Windows images, and `lcow` for Linux containers on Windows.

Options per storage driver

Particular storage-driver can be configured with options specified with `--storage-opt` flags. Options for `devicemapper` are prefixed with `dm` , options for `zfs` start with `zfs` , options for `btrfs` start with `btrfs` and options for `lcow` start with `lcow` .

DEVICEMAPPER OPTIONS

This is an example of the configuration file for devicemapper on Linux:

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

`dm.thinpooldev`

Specifies a custom block storage device to use for the thin pool.

If using a block device for device mapper storage, it is best to use `lvm` to create and manage the thin-pool volume. This volume is then handed to Docker to exclusively create snapshot volumes needed for images and containers.

Managing the thin-pool outside of Engine makes for the most feature-rich method of having Docker utilize device mapper thin provisioning as the backing storage for Docker containers. The highlights of the lvm-based thin-pool management feature include: automatic or interactive thin-pool resize support, dynamically changing thin-pool features, automatic thinp metadata checking when lvm activates the thin-pool, etc.

As a fallback if no thin pool is provided, loopback files are created. Loopback is very slow, but can be used without any pre-configuration of storage. It is strongly recommended that you do not use loopback in production. Ensure your Engine daemon has a `--storage-opt dm.thinpooldev` argument provided.

Example:

```
$ sudo dockerd --storage-opt dm.thinpooldev=/dev/mapper/thin-pool
```

`dm.directlvm_device`

As an alternative to providing a thin pool as above, Docker can setup a block device for you.

Example:

```
$ sudo dockerd --storage-opt dm.directlvm_device=/dev/xvdf
```

`dm.thinp_percent`

Sets the percentage of passed in block device to use for storage.

Example:

```
$ sudo dockerd --storage-opt dm.thinp_percent=95
```

`dm.thinp_metapercent`

Sets the percentage of the passed in block device to use for metadata storage.

Example:

```
$ sudo dockerd --storage-opt dm.thinp_metapercent=1
```

`dm.thinp_autoextend_threshold`

Sets the value of the percentage of space used before `lvm` attempts to autoextend the available space [100 = disabled]

Example:

```
$ sudo dockerd --storage-opt dm.thinp_autoextend_threshold=80
```

`dm.thinp_autoextend_percent`

Sets the value percentage value to increase the thin pool by when `lvm` attempts to autoextend the available space [100 = disabled]

Example:

```
$ sudo dockerd --storage-opt dm.thinp_autoextend_percent=20
```

`dm.basesize`

Specifies the size to use when creating the base device, which limits the size of images and containers. The default value is 10G. Note, thin devices are inherently “sparse”, so a 10G device which is mostly empty doesn’t use 10 GB of space on the pool. However, the filesystem will use more space for the empty case the larger the device is.

The base device size can be increased at daemon restart which will allow all future images and containers (based on those new images) to be of the new base device size.

Examples

```
$ sudo dockerd --storage-opt dm.basesize=50G
```

This will increase the base device size to 50G. The Docker daemon will throw an error if existing base device size is larger than 50G. A user can use this option to expand the base device size however shrinking is not permitted.

This value affects the system-wide "base" empty filesystem that may already be initialized and inherited by pulled images. Typically, a change to this value requires additional steps to take effect:

```
$ sudo service docker stop
$ sudo rm -rf /var/lib/docker
$ sudo service docker start
```

dm.loopdatasize

> Note: This option configures devicemapper loopback, which should not be used in production.

Specifies the size to use when creating the loopback file for the "data" device which is used for the thin pool. The default size is 100G. The file is sparse, so it will not initially take up this much space.

Example

```
$ sudo dockerd --storage-opt dm.loopdatasize=200G
```

dm.loopmetadatasize

> Note: This option configures devicemapper loopback, which should not be used in production.

Specifies the size to use when creating the loopback file for the “metadata” device which is used for the thin pool. The default size is 2G. The file is sparse, so it will not initially take up this much space.

Example

```
$ sudo dockerd --storage-opt dm.loopmetadatasize=4G
```

`dm.fs`

Specifies the filesystem type to use for the base device. The supported options are “ext4” and “xfs”. The default is “xfs”

Example

```
$ sudo dockerd --storage-opt dm.fs=ext4
```

`dm.mkfsarg`

Specifies extra mkfs arguments to be used when creating the base device.

Example

```
$ sudo dockerd --storage-opt "dm.mkfsarg=-O ^has_journal"
```

`dm.mountopt`

Specifies extra mount options used when mounting the thin devices.

Example

```
$ sudo dockerd --storage-opt dm.mountopt=nodiscard
```

`dm.datadev`

(Deprecated, use `dm.thinpooldev`)

Specifies a custom blockdevice to use for data for the thin pool.

If using a block device for device mapper storage, ideally both `datadev` and `metadatadev` should be specified to completely avoid using the loopback device.

Example

```
$ sudo dockerd \
      --storage-opt dm.datadev=/dev/sdb1 \
      --storage-opt dm.metadatadev=/dev/sdc1
```

`dm.metadatadev`

(Deprecated, use `dm.thinpooldev`)

Specifies a custom blockdevice to use for metadata for the thin pool.

For best performance the metadata should be on a different spindle than the data, or even better on an SSD.

If setting up a new metadata pool it is required to be valid. This can be achieved by zeroing the first 4k to indicate empty metadata, like this:

```
$ dd if=/dev/zero of=$metadata_dev bs=4096 count=1
```

Example

```
$ sudo dockerd \
      --storage-opt dm.datadev=/dev/sdb1 \
      --storage-opt dm.metadatadev=/dev/sdc1
```

`dm.blocksize`

Specifies a custom blocksize to use for the thin pool. The default blocksize is 64K.

Example

```
$ sudo dockerd --storage-opt dm.blocksize=512K
```

`dm.blkdiscard`

Enables or disables the use of `blkdiscard` when removing devicemapper devices. This is enabled by default (only) if using loopback devices and is required to resparsify the loopback file on image/container removal.

Disabling this on loopback can lead to *much* faster container removal times, but will make the space used in `/var/lib/docker` directory not be returned to the system for other use when containers are removed.

Examples

```
$ sudo dockerd --storage-opt dm.blkdiscard=false
```

`dm.override_udev_sync_check`

Overrides the `udev` synchronization checks between `devicemapper` and `udev`. `udev` is the device manager for the Linux kernel.

To view the `udev` sync support of a Docker daemon that is using the `devicemapper` driver, run:

```
$ docker info
[...]
Udev Sync Supported: true
[...]
```

When `udev` sync support is `true`, then `devicemapper` and udev can coordinate the activation and deactivation of devices for containers.

When `udev` sync support is `false`, a race condition occurs between the `devicemapper` and `udev` during create and cleanup. The race condition results in errors and failures. (For information on these failures, see docker#4036 (https://github.com/docker/docker/issues/4036))

To allow the `docker` daemon to start, regardless of `udev` sync not being supported, set `dm.override_udev_sync_check` to true:

```
$ sudo dockerd --storage-opt dm.override_udev_sync_check=true
```

When this value is `true`, the `devicemapper` continues and simply warns you the

errors are happening.

> Note: The ideal is to pursue a `docker` daemon and environment that does support synchronizing with `udev`. For further discussion on this topic, see docker#4036 (https://github.com/docker/docker/issues/4036). Otherwise, set this flag for migrating existing Docker daemons to a daemon with a supported environment.

`dm.use_deferred_removal`

Enables use of deferred device removal if `libdm` and the kernel driver support the mechanism.

Deferred device removal means that if device is busy when devices are being removed/deactivated, then a deferred removal is scheduled on device. And devices automatically go away when last user of the device exits.

For example, when a container exits, its associated thin device is removed. If that device has leaked into some other mount namespace and can't be removed, the container exit still succeeds and this option causes the system to schedule the device for deferred removal. It does not wait in a loop trying to remove a busy device.

Example

```
$ sudo dockerd --storage-opt dm.use_deferred_removal=true
```

`dm.use_deferred_deletion`

Enables use of deferred device deletion for thin pool devices. By default, thin pool device deletion is synchronous. Before a container is deleted, the Docker daemon removes any associated devices. If the storage driver can not remove a device, the container deletion fails and daemon returns.

```
Error deleting container: Error response from daemon: Cannot destroy
 container
```

To avoid this failure, enable both deferred device deletion and deferred device removal on the daemon.

```
$ sudo dockerd \
      --storage-opt dm.use_deferred_deletion=true \
      --storage-opt dm.use_deferred_removal=true
```

With these two options enabled, if a device is busy when the driver is deleting a container, the driver marks the device as deleted. Later, when the device isn't in use, the driver deletes it.

In general it should be safe to enable this option by default. It will help when unintentional leaking of mount point happens across multiple mount namespaces.

`dm.min_free_space`

Specifies the min free space percent in a thin pool require for new device creation to succeed. This check applies to both free data space as well as free metadata space. Valid values are from 0% - 99%. Value 0% disables free space checking logic. If user does not specify a value for this option, the Engine uses a default value of 10%.

Whenever a new a thin pool device is created (during `docker pull` or during container creation), the Engine checks if the minimum free space is available. If sufficient space is unavailable, then device creation fails and any relevant `docker` operation fails.

To recover from this error, you must create more free space in the thin pool to recover from the error. You can create free space by deleting some images and containers from the thin pool. You can also add more storage to the thin pool.

To add more space to a LVM (logical volume management) thin pool, just add more storage to the volume group container thin pool; this should automatically resolve any errors. If your configuration uses loop devices, then stop the Engine daemon, grow the size of loop files and restart the daemon to resolve the issue.

Example

```
$ sudo dockerd --storage-opt dm.min_free_space=10%
```

`dm.xfs_nospace_max_retries`

Specifies the maximum number of retries XFS should attempt to complete IO when ENOSPC (no space) error is returned by underlying storage device.

By default XFS retries infinitely for IO to finish and this can result in unkillable process. To change this behavior one can set xfs_nospace_max_retries to say 0 and XFS will not retry IO after getting ENOSPC and will shutdown filesystem.

Example

```
$ sudo dockerd --storage-opt dm.xfs_nospace_max_retries=0
```

`dm.libdm_log_level`

Specifies the maxmimum `libdm` log level that will be forwarded to the `dockerd` log (as specified by `--log-level`). This option is primarily intended for debugging problems involving `libdm`. Using values other than the defaults may cause false-positive warnings to be logged.

Values specified must fall within the range of valid `libdm` log levels. At the time of writing, the following is the list of `libdm` log levels as well as their corresponding levels when output by `dockerd`.

`libdm` Level	Value	**`--log-level`**
`_LOG_FATAL`	2	error
`_LOG_ERR`	3	error
`_LOG_WARN`	4	warn
`_LOG_NOTICE`	5	info
`_LOG_INFO`	6	info
`_LOG_DEBUG`	7	debug

Example

```
$ sudo dockerd \
      --log-level debug \
      --storage-opt dm.libdm_log_level=7
```

ZFS OPTIONS

`zfs.fsname`

Set zfs filesystem under which docker will create its own datasets. By default docker will pick up the zfs filesystem where docker graph (`/var/lib/docker`) is located.

Example

```
$ sudo dockerd -s zfs --storage-opt zfs.fsname=zroot/docker
```

BTRFS OPTIONS

`btrfs.min_space`

Specifies the minimum size to use when creating the subvolume which is used for containers. If user uses disk quota for btrfs when creating or running a container with --storage-opt size option, docker should ensure the size cannot be smaller than btrfs.min_space.

Example

```
$ sudo dockerd -s btrfs --storage-opt btrfs.min_space=10G
```

OVERLAY2 OPTIONS

`overlay2.override_kernel_check`

Overrides the Linux kernel version check allowing overlay2. Support for specifying multiple lower directories needed by overlay2 was added to the Linux kernel in 4.0.0. However, some older kernel versions may be patched to add multiple lower directory support for OverlayFS. This option should only be used after verifying this support exists in the kernel. Applying this option on a kernel without this support will cause failures on mount.

`overlay2.size`

Sets the default max size of the container. It is supported only when the backing fs is `xfs` and mounted with `pquota` mount option. Under these conditions the user can pass any size less then the backing fs size.

Example

```
$ sudo dockerd -s overlay2 --storage-opt overlay2.size=1G
```

WINDOWSFILTER OPTIONS

`size`

Specifies the size to use when creating the sandbox which is used for containers. Defaults to 20G.

Example

```
C:\> dockerd --storage-opt size=40G
```

LCOW (LINUX CONTAINERS ON WINDOWS) OPTIONS

`lcow.globalmode`

Specifies whether the daemon instantiates utility VM instances as required (recommended and default if omitted), or uses single global utility VM (better performance, but has security implications and not recommended for production deployments).

Example

```
C:\> dockerd --storage-opt lcow.globalmode=false
```

`lcow.kirdpath`

Specifies the folder path to the location of a pair of kernel and initrd files used for booting a utility VM. Defaults to `%ProgramFiles%\Linux Containers` .

Example

```
C:\> dockerd --storage-opt lcow.kirdpath=c:\path\to\files
```

`lcow.kernel`

Specifies the filename of a kernel file located in the `lcow.kirdpath` path. Defaults to `bootx64.efi` .

Example

```
C:\> dockerd --storage-opt lcow.kernel=kernel.efi
```

`lcow.initrd`

Specifies the filename of an initrd file located in the `lcow.kirdpath` path. Defaults to `initrd.img` .

Example

```
C:\> dockerd --storage-opt lcow.initrd=myinitrd.img
```

`lcow.bootparameters`

Specifies additional boot parameters for booting utility VMs when in kernel/ initrd mode. Ignored if the utility VM is booting from VHD. These settings are kernel specific.

Example

```
C:\> dockerd --storage-opt "lcow.bootparameters='option=value'"
```

`lcow.vhdx`

Specifies a custom VHDX to boot a utility VM, as an alternate to kernel and initrd booting. Defaults to `uvm.vhdx` under `lcow.kirdpath` .

Example

```
C:\> dockerd --storage-opt lcow.vhdx=custom.vhdx
```

`lcow.timeout`

Specifies the timeout for utility VM operations in seconds. Defaults to 300.

Example

```
C:\> dockerd --storage-opt lcow.timeout=240
```

`lcow.sandboxsize`

Specifies the size in GB to use when creating the sandbox which is used for containers. Defaults to 20. Cannot be less than 20.

Example

```
C:\> dockerd --storage-opt lcow.sandboxsize=40
```

Docker runtime execution options

The Docker daemon relies on a OCI (https://github.com/opencontainers/runtime-spec) compliant runtime (invoked via the `containerd` daemon) as its interface to the Linux kernel `namespaces`, `cgroups`, and `SELinux`.

By default, the Docker daemon automatically starts `containerd`. If you want to control `containerd` startup, manually start `containerd` and pass the path to the `containerd` socket using the `--containerd` flag. For example:

```
$ sudo dockerd --containerd /var/run/dev/docker-containerd.sock
```

Runtimes can be registered with the daemon either via the configuration file or using the `--add-runtime` command line argument.

The following is an example adding 2 runtimes via the configuration:

```
{
        "default-runtime": "runc",
        "runtimes": {
                "runc": {
                        "path": "runc"
                },
                "custom": {
                        "path": "/usr/local/bin/my-runc-replacement"
,
                        "runtimeArgs": [
                                "--debug"
                        ]
                }
        }
}
```

This is the same example via the command line:

```
$ sudo dockerd --add-runtime runc=runc --add-runtime custom=/usr/loc
al/bin/my-runc-replacement
```

> Note: Defining runtime arguments via the command line is not supported.

OPTIONS FOR THE RUNTIME

You can configure the runtime using options specified with the `--exec-opt` flag. All the flag's options have the `native` prefix. A single `native.cgroupdriver` option is available.

The `native.cgroupdriver` option specifies the management of the container's cgroups. You can only specify `cgroupfs` or `systemd`. If you specify `systemd` and it is not available, the system errors out. If you omit the `native.cgroupdriver` option, `cgroupfs` is used.

This example sets the `cgroupdriver` to `systemd`:

```
$ sudo dockerd --exec-opt native.cgroupdriver=systemd
```

Setting this option applies to all containers the daemon launches.

Also Windows Container makes use of `--exec-opt` for special purpose. Docker user can specify default container isolation technology with this, for example:

```
> dockerd --exec-opt isolation=hyperv
```

Will make `hyperv` the default isolation technology on Windows. If no isolation value is specified on daemon start, on Windows client, the default is `hyperv`, and on Windows server, the default is `process`.

Daemon DNS options

To set the DNS server for all Docker containers, use:

```
$ sudo dockerd --dns 8.8.8.8
```

To set the DNS search domain for all Docker containers, use:

```
$ sudo dockerd --dns-search example.com
```

Allow push of nondistributable artifacts

Some images (e.g., Windows base images) contain artifacts whose distribution is restricted by license. When these images are pushed to a registry, restricted artifacts are not included.

To override this behavior for specific registries, use the `--allow-nondistributable-artifacts` option in one of the following forms:

- `--allow-nondistributable-artifacts myregistry:5000` tells the Docker daemon to push nondistributable artifacts to myregistry:5000.
- `--allow-nondistributable-artifacts 10.1.0.0/16` tells the Docker daemon to push nondistributable artifacts to all registries whose resolved IP address is within the subnet described by the CIDR syntax.

This option can be used multiple times.

This option is useful when pushing images containing nondistributable artifacts to a registry on an air-gapped network so hosts on that network can pull the images without connecting to another server.

> Warning: Nondistributable artifacts typically have restrictions on how and where they can be distributed and shared. Only use this feature to push artifacts to private registries and ensure that you are in compliance with any terms that cover redistributing nondistributable artifacts.

Insecure registries

Docker considers a private registry either secure or insecure. In the rest of this section, *registry* is used for *private registry*, and `myregistry:5000` is a placeholder example for a private registry.

A secure registry uses TLS and a copy of its CA certificate is placed on the Docker host at `/etc/docker/certs.d/myregistry:5000/ca.crt`. An insecure registry is either not using TLS (i.e., listening on plain text HTTP), or is using TLS with a CA certificate not known by the Docker daemon. The latter can happen when the certificate was not found under `/etc/docker/certs.d/myregistry:5000/`, or if the certificate verification failed (i.e., wrong CA).

By default, Docker assumes all, but local (see local registries below), registries are secure. Communicating with an insecure registry is not possible if Docker assumes that registry is secure. In order to communicate with an insecure registry, the Docker daemon requires `--insecure-registry` in one of the following two forms:

- `--insecure-registry myregistry:5000` tells the Docker daemon that myregistry:5000 should be considered insecure.
- `--insecure-registry 10.1.0.0/16` tells the Docker daemon that all registries whose domain resolve to an IP address is part of the subnet described by the CIDR syntax, should be considered insecure.

The flag can be used multiple times to allow multiple registries to be marked as insecure.

If an insecure registry is not marked as insecure, `docker pull`, `docker push`, and `docker search` will result in an error message prompting the user to either secure or pass the `--insecure-registry` flag to the Docker daemon as described above.

Local registries, whose IP address falls in the 127.0.0.0/8 range, are automatically marked as insecure as of Docker 1.3.2. It is not recommended to rely on this, as it may change in the future.

Enabling `--insecure-registry`, i.e., allowing un-encrypted and/or untrusted communication, can be useful when running a local registry. However, because its use creates security vulnerabilities it should ONLY be enabled for testing purposes. For increased security, users should add their CA to their system's list of trusted CAs instead of enabling `--insecure-registry`.

LEGACY REGISTRIES

Starting with Docker 17.12, operations against registries supporting only the legacy v1 protocol are no longer supported. Specifically, the daemon will not attempt `push`, `pull` and `login` to v1 registries. The exception to this is `search` which can still be performed on v1 registries.

The `disable-legacy-registry` configuration option has been removed and, when used, will produce an error on daemon startup.

Running a Docker daemon behind an HTTPS_PROXY

When running inside a LAN that uses an `HTTPS` proxy, the Docker Hub certificates will be replaced by the proxy's certificates. These certificates need to be added to your Docker host's configuration:

1. Install the `ca-certificates` package for your distribution
2. Ask your network admin for the proxy's CA certificate and append them to `/etc/pki/tls/certs/ca-bundle.crt`
3. Then start your Docker daemon with `HTTPS_PROXY=http://username:password@proxy:port/ dockerd`. The `username:` and `password@` are optional - and are only needed if your proxy is set up to require authentication.

This will only add the proxy and authentication to the Docker daemon's requests - your `docker build`s and running containers will need extra configuration to use the proxy

Default `ulimit` settings

`--default-ulimit` allows you to set the default `ulimit` options to use for all containers. It takes the same options as `--ulimit` for `docker run`. If these defaults are not set, `ulimit` settings will be inherited, if not set on `docker run`, from the Docker daemon. Any `--ulimit` options passed to `docker run` will overwrite these defaults.

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 details please check the run (https://docs.docker.com/engine/reference/commandline/run/) reference.

Node discovery

The `--cluster-advertise` option specifies the `host:port` or `interface:port` combination that this particular daemon instance should use when advertising itself to the cluster. The daemon is reached by remote hosts through this value. If you specify an interface, make sure it includes the IP address of the actual Docker host. For Engine installation created through `docker-machine`, the interface is typically `eth1`.

The daemon uses libkv (https://github.com/docker/libkv/) to advertise the node within the cluster. Some key-value backends support mutual TLS. To configure the client TLS settings used by the daemon can be configured using the `--cluster-store-opt` flag, specifying the paths to PEM encoded files. For example:

```
$ sudo dockerd \
    --cluster-advertise 192.168.1.2:2376 \
    --cluster-store etcd://192.168.1.2:2379 \
    --cluster-store-opt kv.cacertfile=/path/to/ca.pem \
    --cluster-store-opt kv.certfile=/path/to/cert.pem \
    --cluster-store-opt kv.keyfile=/path/to/key.pem
```

The currently supported cluster store options are:

Option	Description
`discovery.heartbeat`	Specifies the heartbeat timer in seconds which is used by the daemon as a `keepalive` mechanism to make sure discovery module treats the node as alive in the cluster. If not configured, the default value is 20 seconds.

Option	Description
`discovery.ttl`	Specifies the TTL (time-to-live) in seconds which is used by the discovery module to timeout a node if a valid heartbeat is not received within the configured ttl value. If not configured, the default value is 60 seconds.
`kv.cacertfile`	Specifies the path to a local file with PEM encoded CA certificates to trust.
`kv.certfile`	Specifies the path to a local file with a PEM encoded certificate. This certificate is used as the client cert for communication with the Key/Value store.
`kv.keyfile`	Specifies the path to a local file with a PEM encoded private key. This private key is used as the client key for communication with the Key/Value store.
`kv.path`	Specifies the path in the Key/Value store. If not configured, the default value is 'docker/nodes'.

Access authorization

Docker's access authorization can be extended by authorization plugins that your organization can purchase or build themselves. You can install one or more authorization plugins when you start the Docker `daemon` using the `--authorization-plugin=PLUGIN_ID` option.

```
$ sudo dockerd --authorization-plugin=plugin1 --authorization-plugin
=plugin2,...
```

The `PLUGIN_ID` value is either the plugin's name or a path to its specification file. The plugin's implementation determines whether you can specify a name or path. Consult with your Docker administrator to get information about the plugins available to you.

Once a plugin is installed, requests made to the `daemon` through the command line or Docker's Engine API are allowed or denied by the plugin. If you have multiple plugins installed, each plugin, in order, must allow the request for it to complete.

For information about how to create an authorization plugin, see authorization plugin (https://docs.docker.com/engine/extend/plugins_authorization/) section in the Docker extend section of this documentation.

Daemon user namespace options

The Linux kernel user namespace support (http://man7.org/linux/man-pages/man7/user_namespaces.7.html) provides additional security by enabling a process, and therefore a container, to have a unique range of user and group IDs which are outside the traditional user and group range utilized by the host system. Potentially the most important security improvement is that, by default, container processes running as the `root` user will have expected administrative privilege (with some restrictions) inside the container but will effectively be mapped to an unprivileged `uid` on the host.

For details about how to use this feature, as well as limitations, see Isolate containers with a user namespace (https://docs.docker.com/engine/security/userns-remap/).

Miscellaneous options

IP masquerading uses address translation to allow containers without a public IP to talk to other machines on the Internet. This may interfere with some network topologies and can be disabled with `--ip-masq=false` .

Docker supports softlinks for the Docker data directory (`/var/lib/docker`) and for `/var/lib/docker/tmp` . The `DOCKER_TMPDIR` and the data directory can be set like this:

```
DOCKER_TMPDIR=/mnt/disk2/tmp /usr/local/bin/dockerd -D -g /var/lib/d
ocker -H unix:// > /var/lib/docker-machine/docker.log 2>&1
# or
export DOCKER_TMPDIR=/mnt/disk2/tmp
/usr/local/bin/dockerd -D -g /var/lib/docker -H unix:// > /var/lib/d
ocker-machine/docker.log 2>&1
```

DEFAULT CGROUP PARENT

The `--cgroup-parent` option allows you to set the default cgroup parent to use for containers. If this option is not set, it defaults to `/docker` for fs cgroup driver and `system.slice` for systemd cgroup driver.

If the cgroup has a leading forward slash (`/`), the cgroup is created under the root cgroup, otherwise the cgroup is created under the daemon cgroup.

Assuming the daemon is running in cgroup `daemoncgroup` , `--cgroup-parent=/foobar` creates a cgroup in `/sys/fs/cgroup/memory/foobar` , whereas using `--cgroup-parent=foobar` creates the cgroup in `/sys/fs/cgroup/memory/daemoncgroup/foobar`

The systemd cgroup driver has different rules for `--cgroup-parent` . Systemd represents hierarchy by slice and the name of the slice encodes the location in the tree. So `--cgroup-parent` for systemd cgroups should be a slice name. A name can consist of a dash-separated series of names, which describes the path to the slice from the root slice. For example, `--cgroup-parent=user-Γ-b.slice` means the memory cgroup for the container is created in `/sys/fs/cgroup/memory/user.slice/user-a.slice/user-Γ-b.slice/docker-<id>.scope` .

This setting can also be set per container, using the `--cgroup-parent` option on `docker create` and `docker run` , and takes precedence over the `--cgroup-parent` option on the daemon.

DAEMON METRICS

The `--metrics-addr` option takes a tcp address to serve the metrics API. This feature is still experimental, therefore, the daemon must be running in experimental mode for this feature to work.

To serve the metrics API on localhost:1337 you would specify `--metrics-addr 127.0.0.1:1337` allowing you to make requests on the API at `127.0.0.1:1337/metrics` to receive metrics in the prometheus (https://prometheus.io/docs/instrumenting/exposition_formats/) format.

If you are running a prometheus server you can add this address to your scrape configs to have prometheus collect metrics on Docker. For more information on prometheus you can view the website here (https://prometheus.io/).

```
scrape_configs:
  - job_name: 'docker'
    static_configs:
      - targets: ['127.0.0.1:1337']
```

Please note that this feature is still marked as experimental as metrics and metric names could change while this feature is still in experimental. Please provide feedback on what you would like to see collected in the API.

NODE GENERIC RESOURCES

The `--node-generic-resources` option takes a list of key-value pair (`key=value`) that allows you to advertise user defined resources in a swarm cluster.

The current expected use case is to advertise NVIDIA GPUs so that services requesting `NVIDIA-GPU=[0-16]` can land on a node that has enough GPUs for the task to run.

Example of usage:

```
{
        "node-generic-resources": ["NVIDIA-GPU=UUID1", "NVIDIA-GPU=U
UID2"]
}
```

Daemon configuration file

The `--config-file` option allows you to set any configuration option for the daemon in a JSON format. This file uses the same flag names as keys, except for flags that allow several entries, where it uses the plural of the flag name, e.g., `labels` for the `label` flag.

The options set in the configuration file must not conflict with options set via flags. The docker daemon fails to start if an option is duplicated between the file and the flags, regardless their value. We do this to avoid silently ignore changes introduced in configuration reloads. For example, the daemon fails to start if you set daemon labels in the configuration file and also set daemon labels via the `--label` flag. Options that are not present in the file are ignored when the daemon starts.

On Linux

The default location of the configuration file on Linux is `/etc/docker/daemon.json`. The `--config-file` flag can be used to specify a non-default location.

This is a full example of the allowed configuration options on Linux:

```
{
        "authorization-plugins": [],
        "data-root": "",
        "dns": [],
        "dns-opts": [],
        "dns-search": [],
        "exec-opts": [],
        "exec-root": "",
        "experimental": false,
        "features": {},
        "storage-driver": "",
        "storage-opts": [],
        "labels": [],
        "live-restore": true,
        "log-driver": "json-file",
        "log-opts": {
                "max-size": "10m",
                "max-files":"5",
                "labels": "somelabel",
                "env": "os,customer"
        },
        "mtu": 0,
        "pidfile": "",
        "cluster-store": "",
        "cluster-store-opts": {},
        "cluster-advertise": "",
        "max-concurrent-downloads": 3,
        "max-concurrent-uploads": 5,
        "default-shm-size": "64M",
        "shutdown-timeout": 15,
        "debug": true,
        "hosts": [],
        "log-level": "",
        "tls": true,
        "tlsverify": true,
        "tlscacert": "",
        "tlscert": "",
        "tlskey": "",
        "swarm-default-advertise-addr": "",
        "api-cors-header": "",
        "selinux-enabled": false,
        "userns-remap": "",
        "group": "",
        "cgroup-parent": "",
        "default-ulimits": {
                "nofile": {
                        "Name": "nofile",
                        "Hard": 64000,
```

```
                        "Soft": 64000
                }
        },
        "init": false,
        "init-path": "/usr/libexec/docker-init",
        "ipv6": false,
        "iptables": false,
        "ip-forward": false,
        "ip-masq": false,
        "userland-proxy": false,
        "userland-proxy-path": "/usr/libexec/docker-proxy",
        "ip": "0.0.0.0",
        "bridge": "",
        "bip": "",
        "fixed-cidr": "",
        "fixed-cidr-v6": "",
        "default-gateway": "",
        "default-gateway-v6": "",
        "icc": false,
        "raw-logs": false,
        "allow-nondistributable-artifacts": [],
        "registry-mirrors": [],
        "seccomp-profile": "",
        "insecure-registries": [],
        "no-new-privileges": false,
        "default-runtime": "runc",
        "oom-score-adjust": -500,
        "node-generic-resources": ["NVIDIA-GPU=UUID1", "NVIDIA-GPU=U
UID2"],
        "runtimes": {
                "cc-runtime": {
                        "path": "/usr/bin/cc-runtime"
                },
                "custom": {
                        "path": "/usr/local/bin/my-runc-replacement"
,
                        "runtimeArgs": [
                                "--debug"
                        ]
                }
        },
        "default-address-pools":[{"base":"172.80.0.0/16","size":24},
        {"base":"172.90.0.0/16","size":24}]
}
```

> Note: You cannot set options in `daemon.json` that have already been set on daemon startup as a flag. On systems that use `systemd` to start the Docker daemon, `-H` is already set, so you cannot use the `hosts` key in `daemon.json` to add listening addresses. See https://docs.docker.com/engine/admin/systemd/#custom-docker-daemon-options for how to accomplish this task with a systemd drop-in file.

On Windows

The default location of the configuration file on Windows is `%programdata%\docker\config\daemon.json`. The `--config-file` flag can be used to specify a non-default location.

This is a full example of the allowed configuration options on Windows:

```
{
    "authorization-plugins": [],
    "data-root": "",
    "dns": [],
    "dns-opts": [],
    "dns-search": [],
    "exec-opts": [],
    "experimental": false,
    "features":{},
    "storage-driver": "",
    "storage-opts": [],
    "labels": [],
    "log-driver": "",
    "mtu": 0,
    "pidfile": "",
    "cluster-store": "",
    "cluster-advertise": "",
    "max-concurrent-downloads": 3,
    "max-concurrent-uploads": 5,
    "shutdown-timeout": 15,
    "debug": true,
    "hosts": [],
    "log-level": "",
    "tlsverify": true,
    "tlscacert": "",
    "tlscert": "",
    "tlskey": "",
    "swarm-default-advertise-addr": "",
    "group": "",
    "default-ulimits": {},
    "bridge": "",
    "fixed-cidr": "",
    "raw-logs": false,
    "allow-nondistributable-artifacts": [],
    "registry-mirrors": [],
    "insecure-registries": []
}
```

FEATURE OPTIONS

The optional field `features` in `daemon.json` allows users to enable or disable specific daemon features. For example, `{"features":{"buildkit": true}}` enables `buildkit` as the default docker image builder.

The list of currently supported feature options:

- `buildkit` : It enables `buildkit` as default builder when set to `true` or disables it by `false` . Note that if this option is not explicitly set in the daemon config file, then it is up to the cli to determine which builder to invoke.

CONFIGURATION RELOAD BEHAVIOR

Some options can be reconfigured when the daemon is running without requiring to restart the process. We use the `SIGHUP` signal in Linux to reload, and a global event in Windows with the key `Global\docker-daemon-config-$PID` . The options can be modified in the configuration file but still will check for conflicts with the provided flags. The daemon fails to reconfigure itself if there are conflicts, but it won't stop execution.

The list of currently supported options that can be reconfigured is this:

- `debug` : it changes the daemon to debug mode when set to true.
- `cluster-store` : it reloads the discovery store with the new address.
- `cluster-store-opts` : it uses the new options to reload the discovery store.
- `cluster-advertise` : it modifies the address advertised after reloading.
- `labels` : it replaces the daemon labels with a new set of labels.
- `live-restore` : Enables keeping containers alive during daemon downtime (https://docs.docker.com/config/containers/live-restore/).
- `max-concurrent-downloads` : it updates the max concurrent downloads for each pull.
- `max-concurrent-uploads` : it updates the max concurrent uploads for each push.
- `default-runtime` : it updates the runtime to be used if not is specified at container creation. It defaults to "default" which is the runtime shipped with the official docker packages.
- `runtimes` : it updates the list of available OCI runtimes that can be used to run containers.
- `authorization-plugin` : it specifies the authorization plugins to use.
- `allow-nondistributable-artifacts` : Replaces the set of registries to which the daemon will push nondistributable artifacts with a new set of registries.
- `insecure-registries` : it replaces the daemon insecure registries with a new set of insecure registries. If some existing insecure registries in daemon's configuration are not in newly reloaded insecure resgitries, these existing ones will be removed from daemon's config.
- `registry-mirrors` : it replaces the daemon registry mirrors with a new set of registry mirrors. If some existing registry mirrors in daemon's configuration are not in newly reloaded registry mirrors, these existing ones will be removed from daemon's config.
- `shutdown-timeout` : it replaces the daemon's existing configuration timeout with a new timeout for shutting down all containers.

- `features` : it explicitly enables or disables specific features.

Updating and reloading the cluster configurations such as `--cluster-store` , `--cluster-advertise` and `--cluster-store-opts` will take effect only if these configurations were not previously configured. If `--cluster-store` has been provided in flags and `cluster-advertise` not, `cluster-advertise` can be added in the configuration file without accompanied by `--cluster-store` . Configuration reload will log a warning message if it detects a change in previously configured cluster configurations.

Run multiple daemons

> Note: Running multiple daemons on a single host is considered as "experimental". The user should be aware of unsolved problems. This solution may not work properly in some cases. Solutions are currently under development and will be delivered in the near future.

This section describes how to run multiple Docker daemons on a single host. To run multiple daemons, you must configure each daemon so that it does not conflict with other daemons on the same host. You can set these options either by providing them as flags, or by using a daemon configuration file (/engine/reference/commandline/dockerd/#daemon-configuration-file).

The following daemon options must be configured for each daemon:

```
-b, --bridge=                           Attach containers to a networ
k bridge
--exec-root=/var/run/docker             Root of the Docker execdriver
--data-root=/var/lib/docker             Root of persisted Docker data
-p, --pidfile=/var/run/docker.pid       Path to use for daemon PID fi
le
-H, --host=[]                           Daemon socket(s) to connect t
o
--iptables=true                         Enable addition of iptables r
ules
--config-file=/etc/docker/daemon.json   Daemon configuration file
--tlscacert="~/.docker/ca.pem"          Trust certs signed only by th
is CA
--tlscert="~/.docker/cert.pem"          Path to TLS certificate file
--tlskey="~/.docker/key.pem"            Path to TLS key file
```

When your daemons use different values for these flags, you can run them on the same host without any problems. It is very important to properly understand the meaning of those options and to use them correctly.

- The `-b, --bridge=` flag is set to `docker0` as default bridge network. It is created automatically when you install Docker. If you are not using the default, you must create and configure the bridge manually or just set it to 'none': `--bridge=none`
- `--exec-root` is the path where the container state is stored. The default value is `/var/run/docker`. Specify the path for your running daemon here.
- `--data-root` is the path where persisted data such as images, volumes, and cluster state are stored. The default value is `/var/lib/docker`. To avoid any conflict with other daemons, set this parameter separately for each daemon.
- `-p, --pidfile=/var/run/docker.pid` is the path where the process ID of the daemon is stored. Specify the path for your pid file here.
- `--host=[]` specifies where the Docker daemon will listen for client connections. If unspecified, it defaults to `/var/run/docker.sock`.
- `--iptables=false` prevents the Docker daemon from adding iptables rules. If multiple daemons manage iptables rules, they may overwrite rules set by another daemon. Be aware that disabling this option requires you to manually add iptables rules to expose container ports. If you prevent Docker from adding iptables rules, Docker will also not add IP masquerading rules, even if you set `--ip-masq` to `true`. Without IP masquerading rules, Docker containers will not be able to connect to external hosts or the internet when using network other than default bridge.
- `--config-file=/etc/docker/daemon.json` is the path where configuration file is stored. You can use it instead of daemon flags. Specify the path for each daemon.
- `--tls*` Docker daemon supports `--tlsverify` mode that enforces encrypted and authenticated remote connections. The `--tls*` options enable use of specific certificates for individual daemons.

Example script for a separate "bootstrap" instance of the Docker daemon without network:

```
$ sudo dockerd \
        -H unix:///var/run/docker-bootstrap.sock \
        -p /var/run/docker-bootstrap.pid \
        --iptables=false \
        --ip-masq=false \
        --bridge=none \
        --data-root=/var/lib/docker-bootstrap \
        --exec-root=/var/run/docker-bootstrap
```

container (https://docs.docker.com/glossary/?term=container), daemon (https://docs.docker.com/glossary/?term=daemon), runtime (https://docs.docker.com/glossary/?term=runtime)

Use swarm mode routing mesh

Estimated reading time: 8 minutes

Docker Engine swarm mode makes it easy to publish ports for services to make them available to resources outside the swarm. All nodes participate in an ingress routing mesh. The routing mesh enables each node in the swarm to accept connections on published ports for any service running in the swarm, even if there's no task running on the node. The routing mesh routes all incoming requests to published ports on available nodes to an active container.

To use the ingress network in the swarm, you need to have the following ports open between the swarm nodes before you enable swarm mode:

- Port `7946` TCP/UDP for container network discovery.
- Port `4789` UDP for the container ingress network.

You must also open the published port between the swarm nodes and any external resources, such as an external load balancer, that require access to the port.

You can also bypass the routing mesh (/engine/swarm/ingress/#bypass-the-routing-mesh) for a given service.

Publish a port for a service

Use the `--publish` flag to publish a port when you create a service. `target` is used to specify the port inside the container, and `published` is used to specify the port to bind on the routing mesh. If you leave off the `published` port, a random high-numbered port is bound for each service task. You need to inspect the task to determine the port.

```
$ docker service create \
  --name <SERVICE-NAME> \
  --publish published=<PUBLISHED-PORT>,target=<CONTAINER-PORT> \
  <IMAGE>
```

> Note: The older form of this syntax is a colon-separated string, where the published port is first and the target port is second, such as `-p 8080:80`. The new syntax is preferred because it is easier to read and allows more flexibility.

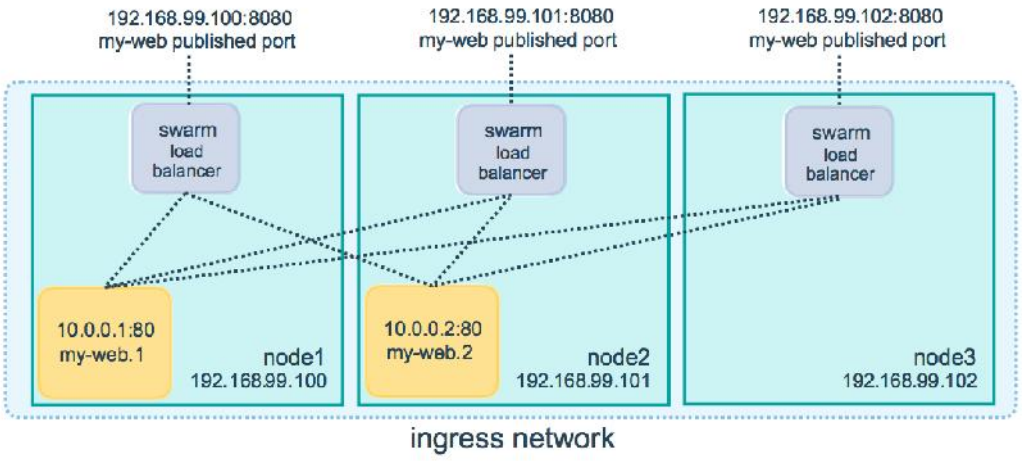
The `<PUBLISHED-PORT>` is the port where the swarm makes the service available. If you omit it, a random high-numbered port is bound. The `<CONTAINER-PORT>` is the port where the container listens. This parameter is required.

For example, the following command publishes port 80 in the nginx container to port 8080 for any node in the swarm:

```
$ docker service create \
  --name my-web \
  --publish published=8080,target=80 \
  --replicas 2 \
  nginx
```

When you access port 8080 on any node, Docker routes your request to an active container. On the swarm nodes themselves, port 8080 may not actually be bound, but the routing mesh knows how to route the traffic and prevents any port conflicts from happening.

The routing mesh listens on the published port for any IP address assigned to the node. For externally routable IP addresses, the port is available from outside the host. For all other IP addresses the access is only available from within the host.



You can publish a port for an existing service using the following command:

```
$ docker service update \
  --publish-add published=<PUBLISHED-PORT>,target=<CONTAINER-PORT> \
  <SERVICE>
```

You can use `docker service inspect` to view the service's published port. For instance:

```
$ docker service inspect --format="{{json .Endpoint.Spec.Ports}}" my
-web

[{"Protocol":"tcp","TargetPort":80,"PublishedPort":8080}]
```

The output shows the `<CONTAINER-PORT>` (labeled `TargetPort`) from the containers and the `<PUBLISHED-PORT>` (labeled `PublishedPort`) where nodes listen for requests for the service.

Publish a port for TCP only or UDP only

By default, when you publish a port, it is a TCP port. You can specifically publish a UDP port instead of or in addition to a TCP port. When you publish both TCP and UDP ports, If you omit the protocol specifier, the port is published as a TCP port. If you use the longer syntax (recommended for Docker 1.13 and higher), set the `protocol` key to either `tcp` or `udp`.

TCP ONLY

Long syntax:

```
$ docker service create --name dns-cache \
  --publish published=53,target=53 \
  dns-cache
```

Short syntax:

```
$ docker service create --name dns-cache \
  -p 53:53 \
  dns-cache
```

TCP AND UDP

Long syntax:

```
$ docker service create --name dns-cache \
  --publish published=53,target=53 \
  --publish published=53,target=53,protocol=udp \
  dns-cache
```

Short syntax:

```
$ docker service create --name dns-cache \
  -p 53:53 \
  -p 53:53/udp \
  dns-cache
```

UDP ONLY

Long syntax:

```
$ docker service create --name dns-cache \
  --publish published=53,target=53,protocol=udp \
  dns-cache
```

Short syntax:

```
$ docker service create --name dns-cache \
  -p 53:53/udp \
  dns-cache
```

Bypass the routing mesh

You can bypass the routing mesh, so that when you access the bound port on a given node, you are always accessing the instance of the service running on that node. This is referred to as `host` mode. There are a few things to keep in mind.

- If you access a node which is not running a service task, the service does not listen on that port. It is possible that nothing is listening, or that a completely different application is listening.
- If you expect to run multiple service tasks on each node (such as when you have 5 nodes but run 10 replicas), you cannot specify a static target port. Either allow Docker to assign a random high-numbered port (by leaving off the `published`), or ensure that only a single instance of the service runs on a given node, by using a global service rather than a replicated one, or by using placement constraints.

To bypass the routing mesh, you must use the long `--publish` service and set `mode` to `host`. If you omit the `mode` key or set it to `ingress`, the routing mesh is used. The following command creates a global service using `host` mode and bypassing the routing mesh.

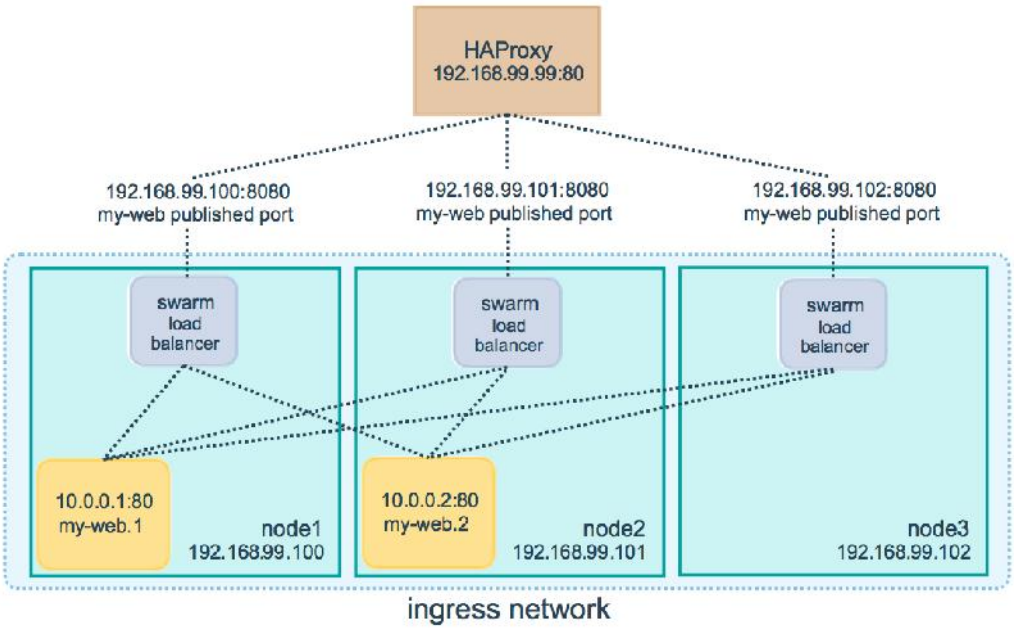
```
$ docker service create --name dns-cache \
  --publish published=53,target=53,protocol=udp,mode=host \
  --mode global \
  dns-cache
```

Configure an external load balancer

You can configure an external load balancer for swarm services, either in combination with the routing mesh or without using the routing mesh at all.

Using the routing mesh

You can configure an external load balancer to route requests to a swarm service. For example, you could configure HAProxy (http://www.haproxy.org) to balance requests to an nginx service published to port 8080.



In this case, port 8080 must be open between the load balancer and the nodes in the swarm. The swarm nodes can reside on a private network that is accessible to the proxy server, but that is not publicly accessible.

You can configure the load balancer to balance requests between every node in the swarm even if there are no tasks scheduled on the node. For example, you could have the following HAProxy configuration in `/etc/haproxy/haproxy.cfg` :

```
global
        log /dev/log    local0
        log /dev/log    local1 notice
...snip...

# Configure HAProxy to listen on port 80
frontend http_front
   bind *:80
   stats uri /haproxy?stats
   default_backend http_back

# Configure HAProxy to route requests to swarm nodes on port 8080
backend http_back
   balance roundrobin
   server node1 192.168.99.100:8080 check
   server node2 192.168.99.101:8080 check
   server node3 192.168.99.102:8080 check
```

When you access the HAProxy load balancer on port 80, it forwards requests to nodes in the swarm. The swarm routing mesh routes the request to an active task. If, for any reason the swarm scheduler dispatches tasks to different nodes, you don't need to reconfigure the load balancer.

You can configure any type of load balancer to route requests to swarm nodes. To learn more about HAProxy, see the HAProxy documentation (https://cbonte.github.io/haproxy-dconv/).

Without the routing mesh

To use an external load balancer without the routing mesh, set `--endpoint-mode` to `dnsrr` instead of the default value of `vip`. In this case, there is not a single virtual IP. Instead, 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. You are responsible for providing the list of IP addresses and ports to your load balancer. See Configure service discovery (https://docs.docker.com/engine/swarm/networking/#configure-service-discovery).

Learn more

- Deploy services to a swarm (https://docs.docker.com/engine/swarm/services/)

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), network (https://docs.docker.com/glossary/?term=network), ingress (https://docs.docker.com/glossary/?term=ingress), routing mesh (https://docs.docker.com/glossary/?term=routing mesh)

Use a load balancer

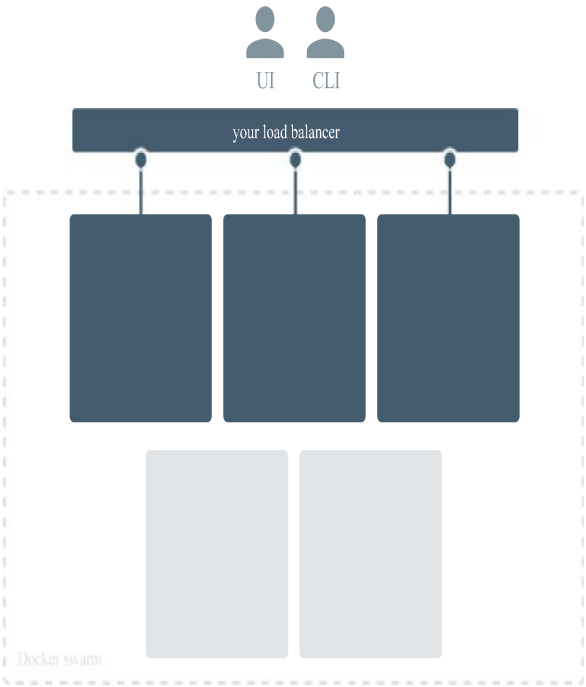
Estimated reading time: 6 minutes

These are the docs for UCP version 2.2.14

To select a different version, use the selector below.

2.2.14

Once you've joined multiple manager nodes for high-availability, you can configure your own load balancer to balance user requests across all manager nodes.



This allows users to access UCP using a centralized domain name. If a manager node goes down, the load balancer can detect that and stop forwarding requests to that node, so that the failure goes unnoticed by users.

Load-balancing on UCP

Since Docker UCP uses mutual TLS, make sure you configure your load balancer to:

- Load-balance TCP traffic on port 443,
- Not terminate HTTPS connections,
- Use the `/_ping` endpoint on each manager node, to check if the node is healthy and if it should remain on the load balancing pool or not.

Load balancing UCP and DTR

By default, both UCP and DTR use port 443. If you plan on deploying UCP and DTR, your load balancer needs to distinguish traffic between the two by IP address or port number.

- If you want to configure your load balancer to listen on port 443:
  - Use one load balancer for UCP, and another for DTR,
  - Use the same load balancer with multiple virtual IPs.
- Configure your load balancer to expose UCP or DTR on a port other than 443.

Configuration examples

Use the following examples to configure your load balancer for UCP.

NGINX HAProxy AWS LB

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}

stream {
    upstream ucp_443 {
        server <UCP_MANAGER_1_IP>:443 max_fails=2 fail_timeout=30s;
        server <UCP_MANAGER_2_IP>:443 max_fails=2 fail_timeout=30s;
        server <UCP_MANAGER_N_IP>:443  max_fails=2 fail_timeout=30s;
    }
    server {
        listen 443;
        proxy_pass ucp_443;
    }
}
```

You can deploy your load balancer using:

NGINX | HAProxy

```
# Create the nginx.conf file, then
# deploy the load balancer

docker run --detach \
  --name ucp-lb \
  --restart=unless-stopped \
  --publish 443:443 \
  --volume ${PWD}/nginx.conf:/etc/nginx/nginx.conf:ro \
  nginx:stable-alpine
```

Where to go next

- Add labels to cluster nodes (https://docs.docker.com/datacenter/ucp/2.2/guides/admin/configure/add-labels-to-cluster-nodes/)

UCP (https://docs.docker.com/glossary/?term=UCP), high-availability (https://docs.docker.com/glossary/?term=high-availability), load balancer (https://docs.docker.com/glossary/?term=load balancer)

Authored by: Andy Clemenko (/author/clemenko)

Docker Reference Architecture: Securing Docker EE and Security Best Practices

Article ID: KB000686

EE LINUX WINDOWS SERVER SECURITY KUBERNETES EE-17.06.2-EE-8 UCP-3.0.0 DTR-2.5.0 EE 2.0

Introduction

Docker lives by “Secure by Default.” With Docker Enterprise Edition (Docker EE), the default configuration and policies provide a solid foundation for a secure environment. However, they can easily be changed to meet the specific needs of any organization.

Docker focuses on three key areas of container security: *secure access*, *secure content*, and *secure platform*. This results in having isolation and containment features not only built into Docker EE but also enabled out of the box. The attack surface area of the Linux kernel is reduced, the containment capabilities of the Docker daemon are improved, and admins build, ship, and run safer applications.

What You Will Learn

This document outlines the default security of Docker EE as well as best practices for further securing Universal Control Plane and Docker Trusted Registry. New features introduced in Docker EE 2.0 such as Image Mirroring and Kubernetes are also explored.

Prerequisites

- Docker EE 2.0 (UCP 3.0, DTR 2.5, Engine 17.06-2) and higher on a Linux host OS with kernel 3.10-0.693 or greater
- Become familiar with Docker Concepts from the Docker docs (https://docs.docker.com/engine/docker-overview/)

Abbreviations

The following abbreviations are used in this document:

- UCP = Universal Control Plane
- DTR = Docker Trusted Registry
- RBAC = Role Based Access Control
- CA = Certificate Authority
- EE = Docker Enterprise Edition
- HA = High Availability
- BOM = Bill of Materials

- CLI = Command Line Interface
- CI = Continuous Integration

Engine and Node Security

There are already several resources that cover the basics of Docker Engine security.

- Docker Security Documentation (https://docs.docker.com/engine/security/security/) covers the fundamentals, such as namespaces and control groups, the attack surface of the Docker daemon, and other kernel security features.
- CIS Docker Community Edition Benchmark (https://www.cisecurity.org/benchmark/docker/) covers the various security-related options in Docker Engine. Useful with Docker EE.
- Docker Bench Security (https://github.com/docker/docker-bench-security) is a script that audits your configuration of Docker Engine against the CIS Benchmark.

Choice of Operating Systems

Docker EE Engine 17.06 (a required prerequisite for installing UCP and included with Docker EE) is supported on the following host operating systems:

- RHEL/CentOS/Oracle Linux 7.4/7.5 (YUM-based systems)
- Ubuntu 16.04 LTS
- SUSE Linux Enterprise 12

For other versions, check out the official Docker support matrix (https://success.docker.com/article/compatibility-matrix).

To take advantage of the built-in security configurations and policies, run the latest version of Docker EE Engine. Also ensure that you update the OS with all of the latest patches. In all cases it is highly recommended to remove as much unnecessary software as possible.

Limit Root Access to Node

Docker EE uses a completely separate authentication backend from the host, providing a clear separation of duties. Docker EE can leverage an existing LDAP/AD infrastructure for authentication. It even utilizes RBAC Labels (https://success.docker.com/api/asset/.%2Frefarch%2Fsecurity-best-practices%2F#clusterrole-basedaccesscontrol) to control access to objects like images and running containers, meaning teams of users can be given full access to running containers. With this access, the users can watch the logs and execute a shell inside the running container. The user never needs to log into the host. Limiting the number of users that have access to the host reduces the attack surface.

Remote Access to Daemon

Don't enable the remote daemon socket. If you must open it for Engine, then ALWAYS secure it with certs. When using Universal Control Plane, you should not open the daemon socket. If you must, be sure to review the instructions for securing the daemon socket (https://docs.docker.com/engine/security/https/).

Privileged Containers

Avoid running privileged containers if at all possible. Running a container privileged gives the container access to ALL the host namespaces (i.e. net, pid, and others). This gives full control of the host to the container. Keep your infrastructure secure by keeping the container and host authentication separate.

Container UID Management

By default the user inside the container is root. Using a defense in depth model, it is recommended that not all containers run as root. An easy way to mitigate this is to use the `--user` declaration at run time. The container runs as the specified user, essentially removing root access.

Also keep in mind that the UID/GID combination for a file inside a container is the same outside of the container. In the following example, a container is running with a UID of 10000 and GID of 10000. If the user touches a file such as `/tmp/secret_file`, on a BIND-mounted directory, the UID/GID of the file is the same both inside and outside of the container as shown:

```
root @ ~  docker run --rm -it -v /tmp:/tmp --user 10000:10000 alpine sh
/ $ whoami
whoami: unknown uid 10000
/ $ touch /tmp/secret_file
/ $ ls -asl /tmp/secret_file
     0 -rw-r--r--    1 10000    10000            0 Jan 26 13:48 /tmp/secret_file
/ $ exit
root @ ~  ls -asl /tmp/secret_file
0 -rw-r--r-- 1 10000 10000 0 Jan 26 08:48 /tmp/secret_file
```

Developers should `root` as little as possible inside the container. Developers should create their app containers with the `USER` declaration in their Dockerfiles.

Seccomp

> Note: Seccomp for Docker EE Engine is available starting with RHEL/CentOS 7 and SLES 12.

Seccomp (short for Secure Computing Mode) is a security feature of the Linux kernel, used to restrict the syscalls available to a given process. This facility has been in the kernel in various forms since 2.6.12 and has been available in Docker Engine since 1.10. The current implementation in Docker Engine provides a default set of restricted syscalls and also allows syscalls to be filtered via either a whitelist or a blacklist on a per-container basis (i.e. different filters can be applied to different containers running in the same Engine). Seccomp profiles are applied at container creation time and cannot be altered for running containers.

Out of the box, Docker comes with a default Seccomp profile that works extremely well for the vast majority of use cases. In general, applying custom profiles is not recommended unless absolutely necessary. More information about building custom profiles and applying them can be found in the Docker Seccomp docs (https://docs.docker.com/engine/security/seccomp/).

To check if your kernel supports seccomp:

```
cat /boot/config-`uname -r` | grep CONFIG_SECCOMP=
```

Look for the following in the output:

```
CONFIG_SECCOMP=y
```

AppArmor / SELinux

AppArmor and SELinux are similar to Seccomp in regards to use profiles. They differ in their execution though. The profile languages used by AppArmor and SELinux are different. AppArmor is only on Debian-based distributions such as Debian and Ubuntu. SELinux is available on Fedora/RHEL/CentOS/Oracle Linux. Rather than a simple list of system calls and arguments, both allow for defining actors (generally processes), actions (reading files, network operations), and targets (files, IPs, protocols, etc.). Both are Linux kernel security modules, and both support mandatory access controls (MAC).

They need to be enabled on the host, while SELinux can be enabled at the daemon level.

To enable SELinux in the Docker daemon modify `/etc/docker/daemon.json` and add the following:

```
"selinux-enabled": true
```

> Note Remember that the file `daemon.json` is JSON-based and needs opening and closing braces — `{ }`.

To check if SELinux is enabled:

```
docker info |grep -A 3 "Security Options"
```

`selinux` should be in the output if it is enabled:

```
Security Options:
 seccomp
  Profile: default
 selinux
```

AppArmor is not applied to the Docker daemon. Apparmor profiles need to be applied at container run time:

```
docker run --rm -it --security-opt apparmor=docker-default hello-world
```

There are some good resources for installing and setting up AppArmor/SELinux such as:

- Techmint - Implementing Mandatory Access Control with SELinux or AppArmor in Linux (http://www.tecmint.com/mandatory-access-control-with-selinux-or-apparmor-linux/)
- nixCraft - Linux Kernel Security (SELinux vs AppArmor vs Grsecurity) (https://www.cyberciti.biz/tips/selinux-vs-apparmor-vs-grsecurity.html)

Bottom line is that you should always use AppArmor or SELinux for their supported operating systems.

Runtime Privilege and Linux Capabilities — Advanced Tooling

> *Starting with kernel 2.2, Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities, which can be independently enabled and disabled. —* Capabilities man page (http://man7.org/linux/man-pages/man7/capabilities.7.html)

Linux capabilities are an even more granular way of reducing surface area. Docker Engine has a default list of capabilities that are kept for newly-created containers, and by using the `--cap-drop` option for `docker run`, users can exclude additional capabilities from being used by processes inside the container on a capability-by-capability basis. All privileges can be dropped with the `--user` option.

Likewise, capabilities that are, by default, not granted to new containers can be added with the `--cap-add` option, though this is discouraged unless absolutely necessary. Using `--cap-add=ALL` is highly discouraged.

More details can be found in the Docker Run Reference (https://docs.docker.com/engine/reference/run/#/runtime-privilege-and-linux-capabilities).

Controls from the CIS Benchmark

There are many good practices that should be applied from the CIS Docker Community Edition Benchmark v1.1.0 (https://success.docker.com/api/asset/.%2Frefarch%2Fsecurity-best-practices%2FCIS_Docker_Community_Edition_Benchmark_v1.1.0.pdf). Some of the controls may not be applicable to your environment. To apply these controls, edit the Engine settings. Editing the Engine setting in `/etc/docker/daemon.json` is the best choice for most of these controls. Refer to the daemon.json guide (https://docs.docker.com/engine/reference/commandline/dockerd/#/daemon-configuration-file) for details.

Apply Centralized Logging — CIS CE Benchmark v1.1.0 : Section 2.12

Having a central location for all Engine and container logs is recommended. This provides "off-node" access to all the logs, empowering developers without having to grant them SSH access.

To enable centralized logging, modify `/etc/docker/daemon.json` and add the following:

```
"log-level": "syslog",
"log-opts": {syslog-address=tcp://192.x.x.x},
```

Then restart the daemon:

```
sudo systemctl restart docker
```

Disable Legacy Registries — CIS CE Benchmark v1.1.0 : Section 2.13

With Docker registry v2, there are some new security features. More specifically, v2 disables the use of HTTP requests in favor of HTTPS. This is why it is advised to disable legacy support.

To disable support for legacy registries, modify `/etc/docker/daemon.json` and add the following:

```
"disable-legacy-registry": true,
```

Then restart the daemon:

```
sudo systemctl restart docker
```

Enable Content Trust — CIS CE Benchmark v1.1.0 : Section 4.5

Content Trust is the cryptographic guarantee that the image pulled is the correct image. Content Trust is enabled by Notary. [Signing images with Notary is discussed](https://success.docker.com/api/asset/.%2Frefarch%2Fsecurity-best-practices%2FContent Trust and Image Signing with Notary) later in this document.

When transferring data among networked systems, trust is a central concern. In particular, when communicating over an untrusted medium such as the Internet, it is critical to ensure the integrity and the publisher of all the data a system operates on. Docker engine is used to push and pull images (data) to a public or private registry. Content Trust provides the ability to verify both the integrity and the publisher of all the data received from a registry over any channel. Content Trust is available on Docker Hub or DTR 2.1.0 and higher. To enable it, add the following shell variable:

```
export DOCKER_CONTENT_TRUST=1
```

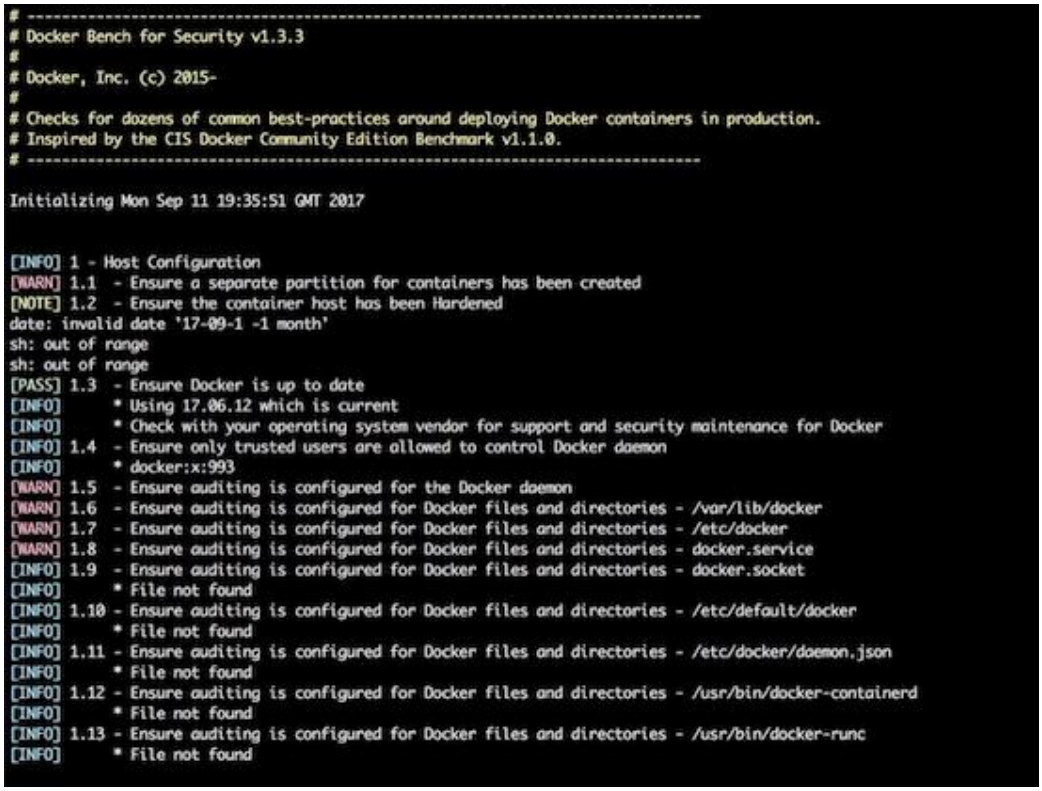
Audit with Docker Bench

Docker Bench Security (https://store.docker.com/community/./images/docker/docker-bench-security) is a script that checks for dozens of common best practices around deploying Docker containers in production. The tests are all automated and are inspired by the CIS Docker Community Edition Benchmark v1.1.0 (https://success.docker.com/api/asset/.%2Frefarch%2Fsecurity-best-practices%2FCIS_Docker_Community_Edition_Benchmark_v1.1.0.pdf).

Here is how to run it :

```
docker run -it --net host --pid host --cap-add audit_control \
    -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
    -v /var/lib:/var/lib \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -v /usr/lib/systemd:/usr/lib/systemd \
    -v /etc:/etc --label docker_bench_security \
    docker/docker-bench-security
```

Here is example output:



The output is straightforward. There is a status message, CIS Benchmark Control number, and description fields. Look for the `[WARN]` messages. The biggest section to pay attention to is `1 - Host Configuration`. Keep in mind that this tool is designed to audit Docker Engine and is a good starting point. Docker Bench Security is NOT intended for auditing the setup of UCP/DTR. There are a few controls that, when enabled, break UCP and DTR.

The following controls are not needed because they affect the operation of UCP/DTR:

- 2.1 Restrict network traffic between containers — Needed for container communication
- 2.6 Configure TLS authentication for Docker daemon — Should not be enabled as it is not needed
- 2.8 Enable user namespace support — Currently not supported with UCP/DTR
- 2.15 Disable Userland Proxy — Disabling the proxy affects how the routing mesh works

Windows Engine and Node Security

As 17.06, Docker EE includes native Windows Server 2016 support. This means you can install Docker EE on Windows natively and attach it to a UCP Swarm. Currently, only Windows worker nodes are supposed. There are some really nice advantages to this. Linux and Windows workloads can now be managed from the same orchestration framework. Windows actually has an added advantage. Windows Server 2016 can encapsulate Docker containers with a Hyper-V virtual machine. This means the Windows workers can actually run the Linux workloads independently.

Some of the advantages of Windows worker nodes include:

- Eliminates conflicts between different versions of IIS/.NET to coexist on a single system with container isolation
- Works with Hyper-V virtualization
- Takes advantage of new base images like Windows Server Core and Nano Server
- Provides a consistent Docker user experience — use the same commands as Docker for Linux environments
- Adds isolation properties with Hyper V containers selected at runtime

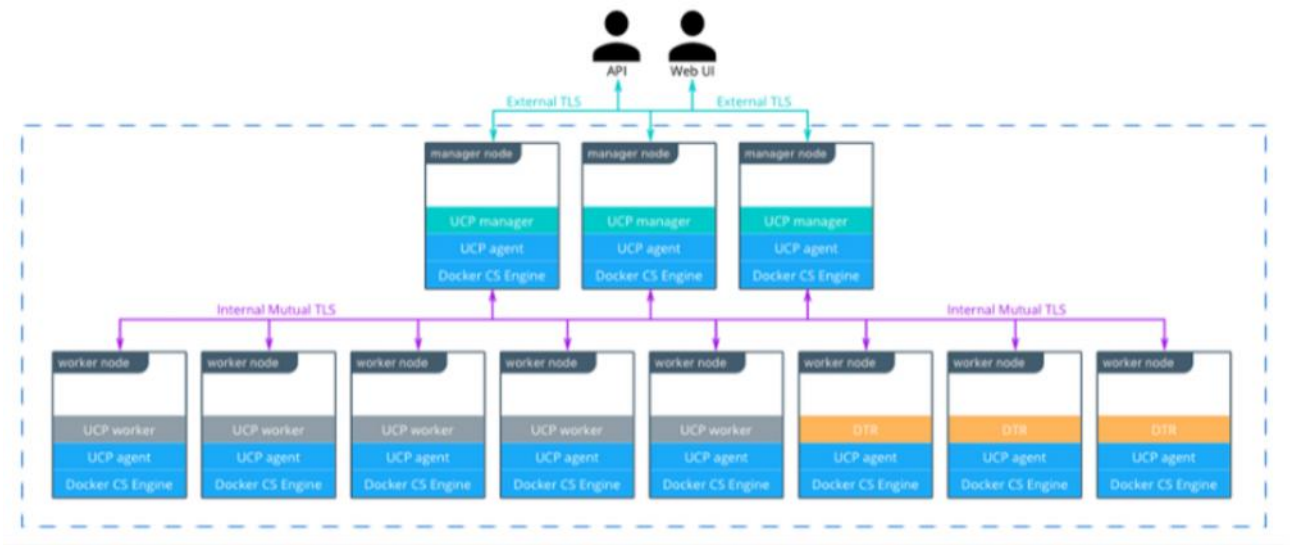
Using the Hyper-V isolation provides another layer of security.

For more information about installing Docker EE on Windows Server 2016 follow the documentation (https://docs.docker.com/engine/installation/windows/docker-ee/). To learn more about Docker on Windows 2016 read the Docker for Windows Server features (https://www.docker.com/docker-windows-server).

UCP Security

Universal Control Plane is configured to be "Secure by default." It uses two Certificate Authorities with Mutual TLS. UCP sets up two CAs. One CA is used for ALL internal communication between managers and workers. The second CA is for the end user communication. Two communication paths are vital to keeping the traffic segregated. The use of Mutual TLS is automatic between the manager and worker nodes. Mutual TLS is where both the client and the server verify the identity of each other.

Worker nodes are unprivileged, meaning they do not have access to the cluster state or secrets. When adding nodes to the UCP cluster, a join token must be used. The token itself incorporates the checksum of the CA cert so the new node can verify that it is communicating with the right swarm.



NEW with Docker EE 2.0 the same "Secure by default" approach is being applied to Kubernetes.

Kubernetes

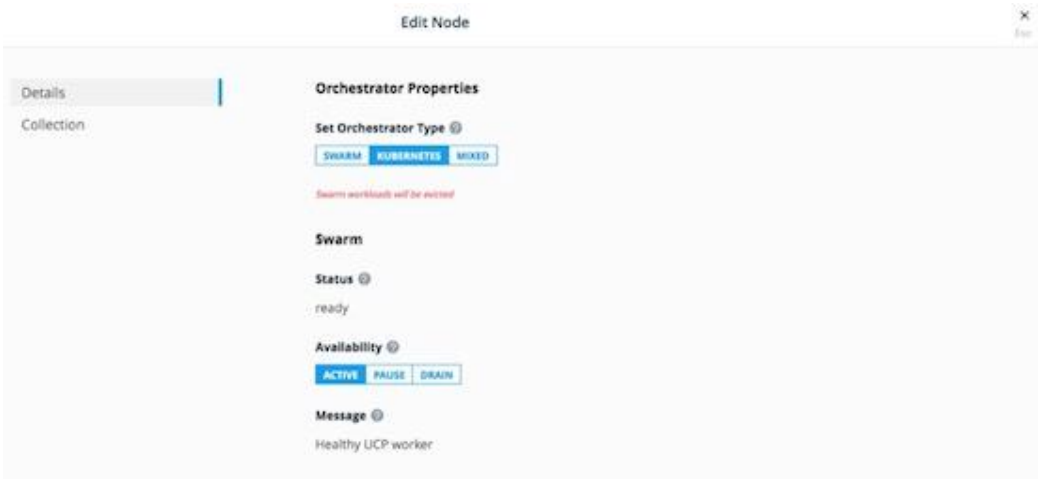
With Docker EE 2.0, UCP now includes an upstream distribution of Kubernetes. From a security point of view this is the best of both worlds. Out of the box Docker EE 2.0 provides user authentication and RBAC on top of Kubernetes. To ensure the Kubernetes orchestrator follows all the security best practices UCP utilizes TLS for the Kubernetes API port. When combined with UCP's auth model, this allows for the same client bundle to talk to the Swarm or Kubernetes API.

For the configuration of Kubernetes, it is recommended that you follow the CIS Kubernetes Benchmark (https://www.cisecurity.org/benchmark/kubernetes/).

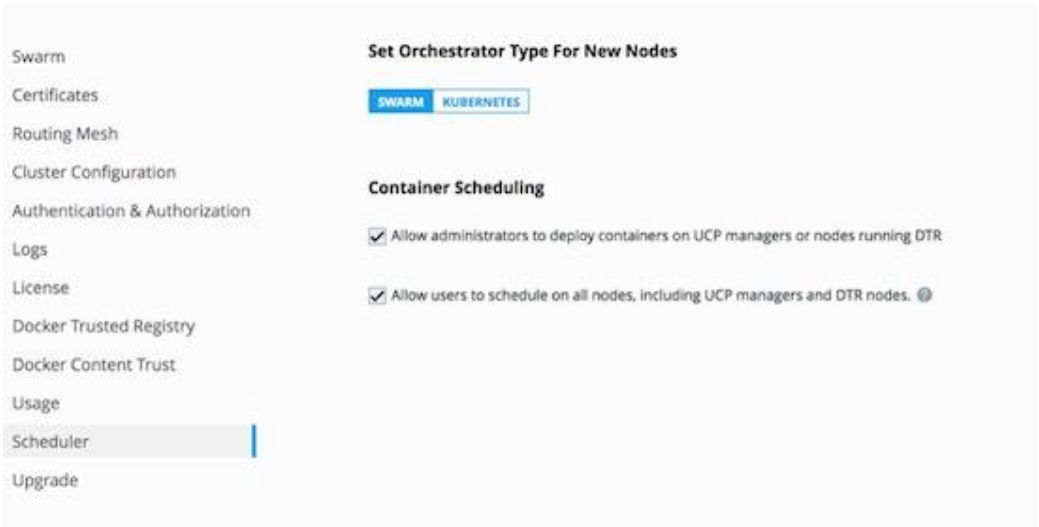
In order to deploy Kubernetes within EE 2.0, the nodes need to be setup. It is not advised to have nodes configure in "Mixed Mode" — do not have a node configured to respond to both Swarm and Kubernetes. This causes an issue where each orchestrator tries to control the containers on that node. There is an exception, Manager nodes. Manager nodes need to be in Mixed Mode. UCP deploys controllers in Mixed mode to ensure all the components are highly available.

Status	Name	Type	Role	Address	Engine	OS/Arch	CPU	Memory	Disk	Details
●	beta-9815	mixed	worker	167.99.228.128	17.06.2-ee...	linux/x86_64	3.74%	66.55%	14.2%	Healthy UCP worker
●	beta-7856	swarm	worker	167.99.228.165	17.06.2-ee...	linux/x86_64	4.26%	5.96%	4.31%	Healthy UCP worker
●	beta-4074	mixed	manager	167.99.228.124	17.06.2-ee...	linux/x86_64	20.1%	33.09%	4.77%	Healthy UCP mana...

To set a node's orchestrator, navigate to Shared Resources -> Nodes -> select the node you want to change. Next select the Configure -> Details. From there select KUBERNETES and save. Notice the warning that all the Swarm workloads will be evicted.



In addition to setting individual nodes for Kubernetes. UCP allows for all new nodes to be set to a specific orchestrator. To set the default orchestrator for new nodes navigate to Admin Settings -> Scheduler, select Kubernetes, and save.



One caveat. Since Docker EE has its own implementation of role-based access control, you can't use Kubernetes RBAC objects directly. Instead, you create UCP roles and grants that correspond with the role objects and bindings in your Kubernetes app. There is a Migrate Kubernetes roles to Docker EE authorization (https://beta.docs.docker.com/ee/ucp/authorization/migrate-kubernetes-roles/) guide for this.

Networking

Networking can be an important part of a Docker EE deployment. The basic rule of thumb is not to have firewalls between the manager and worker nodes. When deploying to a cloud infrastructure, low latency is a must between nodes. Low latency ensures the databases are able to keep quorum. When a software, or hardware, firewall is deployed between the nodes, the following ports need to be opened. Definitions for Scope:

- Internal - Inside the cluster
- External - External to the cluster, vlan, vpc, or subnet
- Self - Within the single node

Hosts	Port	Scope	Purpose
managers, workers	TCP 179	Internal	Port for BGP peers, used for kubernetes networking
managers	TCP 443 (configurable)	External, Internal	Port for the UCP web UI and API
managers	TCP 2376 (configurable)	Internal	Port for the Docker Swarm manager. Used for backwards compatibility
managers	TCP 2377 (configurable)	Internal	Port for control communication between swarm nodes
managers, workers	UDP 4789	Internal	Port for overlay networking
managers	TCP 6443 (configurable)	External, Internal	Port for Kubernetes API server
managers, workers	TCP 6444	Self	Port for Kubernetes API reverse proxy
managers, workers	TCP, UDP 7946	Internal	Port for gossip-based clustering
managers, workers	TCP 10250	Internal	Port for Kubelet
managers, workers	TCP 12376	Internal	Port for a TLS authentication proxy that provides access to the Docker Engine
managers, workers	TCP 12378	Self	Port for Etcd reverse proxy
managers	TCP 12379	Internal	Port for Etcd Control API

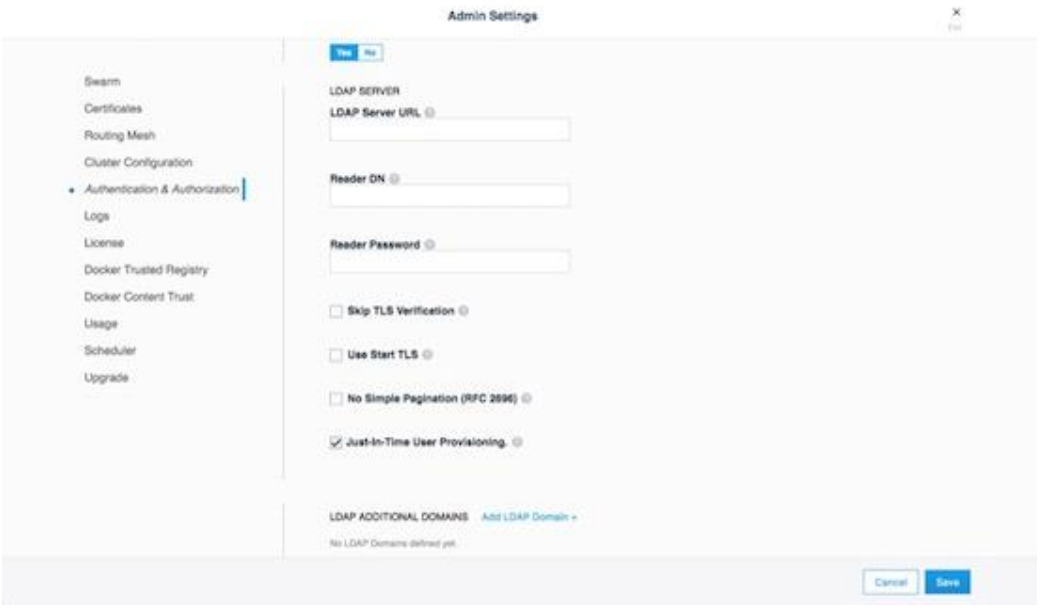
Hosts	Port	Scope	Purpose
managers	TCP 12380	Internal	Port for Etcd Peer API
managers	TCP 12381	Internal	Port for the UCP cluster certificate authority
managers	TCP 12382	Internal	Port for the UCP client certificate authority
managers	TCP 12383	Internal	Port for the authentication storage backend
managers	TCP 12384	Internal	Port for the authentication storage backend for replication across managers
managers	TCP 12385	Internal	Port for the authentication service API
managers	TCP 12386	Internal	Port for the authentication worker
managers	TCP 12387	Internal	Port for the metrics service

Authentication

Docker EE features a single sign-on for the entire cluster, which is accomplished via shared authentication service for UCP and DTR. The single sign-on is provided out of the box with AuthN or via an externally-managed LDAP/AD authentication service. Both authentication backends provide the same level of control. When available, a corporate LDAP service can provide a smoother account experience for users. Refer to the LDAP/AD configuration docs (https://docs.docker.com/datacenter/ucp/2.2/guides/admin/configure/external-auth/) and Docker EE Best Practices and Design Considerations (https://success.docker.com/Architecture/Docker_Reference_Architecture%3A_Docker_EE_Best_Practices_and_Design_Considerations) for instructions and best practices while configuring LDAP authentication.

To change the authentication to LDAP, go to Admin -> Admin Settings -> Certificates in the UCP web

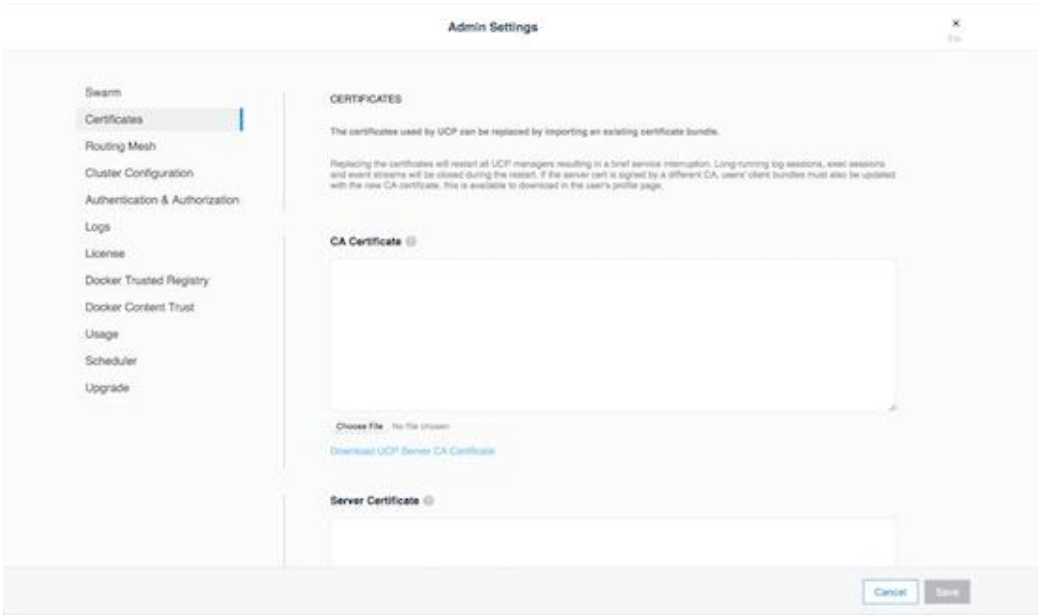
interface.



External Certificates

Using external certificates is a good option when integrating with a corporate environment. Using external, officially-signed certificates simplifies having to distribute Certificate Authority certificates. One best practice is to use the Certificate Authority (CA) for your organization. Reduce the number of certificates by adding multiple Subject Alternative Names (SANs) to a single certificate. This allows the certificate to be valid for multiple URLs. For example, you can set up a certificate for `ucp.example.com`, `dtr.example.com`, and all the underlying hostnames and IP addresses. One certificate/key pair makes deploying certs easier.

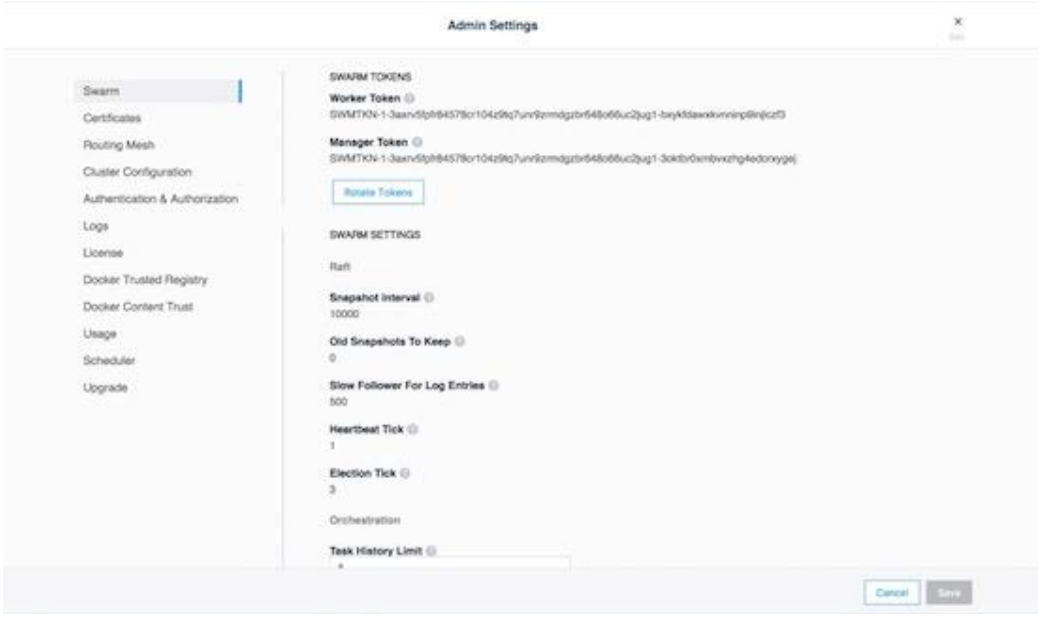
To add an external certificate, go to Admin -> Admin Settings -> Certificates in the UCP web interface and add the CA, Cert, and Key.



More detailed instructions for adding external certificates (https://docs.docker.com/datacenter/ucp/2.2/guides/admin/configure/use-your-own-tls-certificates/) are available in the Docker docs.

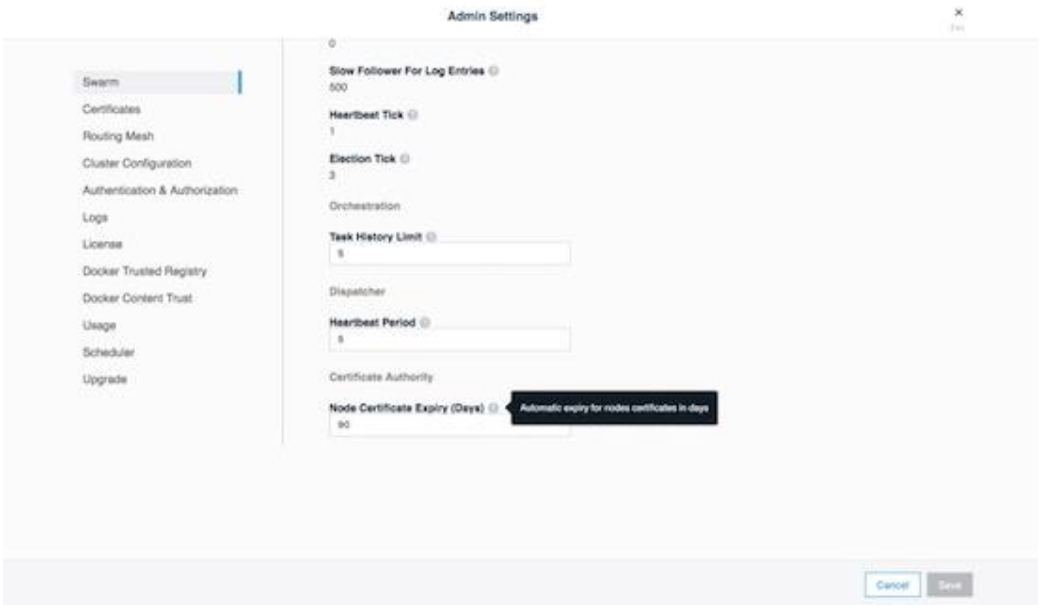
Join Token Rotation

Depending on how the swarm cluster is built, it is possible to have the join token stored in an insecure location. To alleviate any concerns, join tokens can be rotated once the cluster is built. To rotate the keys, go to the Admin -> Admin Settings -> Swarm page, and click the Rotate button.



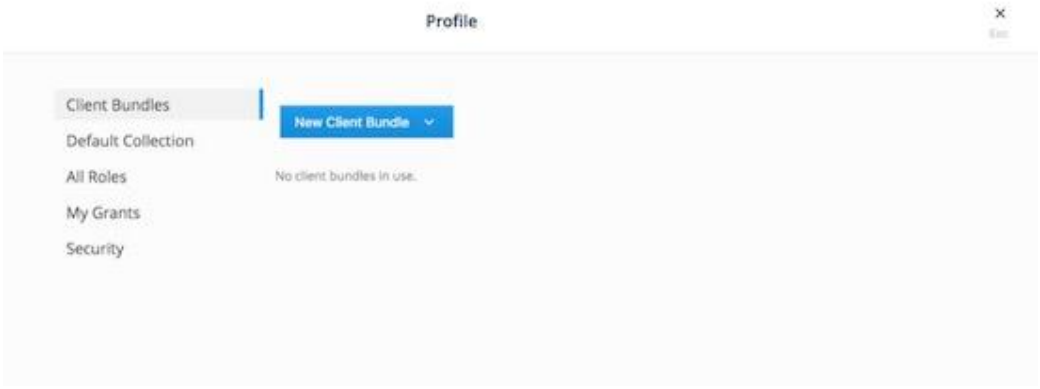
Node Certificate Expiration

UCP's management plane uses a private CA and certificates for all internal communication. The client certificates are automatically rotated on a schedule. Key rotation is a strong method for reducing the effect of a compromised node. There is an option to reduce the time interval, with the default being 90 days. Shorter intervals add stress to the UCP cluster. Similar to rotating the join tokens, go to Admin -> Admin Settings -> Swarm and scroll down.



Client Certificate Bundles

Universal Control Plane makes it easy to create a client certificate bundle for use with the Docker client. The client bundle allows end users to create objects and deploy services from a local Docker client. To create a client bundle, log into UCP, and click the login name in the upper left. Then select My Profile -> Client Bundles.



From here a client bundle can be created and downloaded. Inside the bundle are the files necessary for talking to the UCP cluster directly.

Navigate to the directory where you downloaded the user bundle, and unzip it.

```
unzip ucp-bundle-admin.zip
```

Then run the `env.sh` script:

```
eval $(<env.sh)
```

Verify the changes:

```
docker info
```

The `env.sh` script updates the `DOCKER_HOST` environment variable to make your local Docker CLI communicate with UCP. It also updates the `DOCKER_CERT_PATH` environment variables to use the client certificates that are included in the client bundle you downloaded.

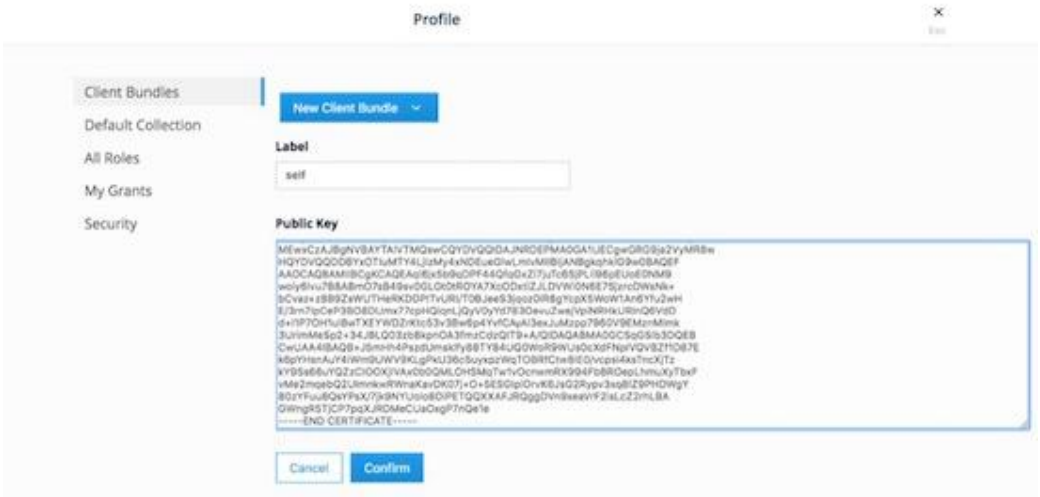
From now on, the Docker CLI client will include the client certificates as part of the request to the Docker engine. The Docker CLI can now be used to create services, networks, volumes, and other resources on a swarm managed by UCP.

To stop talking to the UCP cluster run the following command:

```
unset DOCKER_HOST DOCKER_TLS_VERIFY DOCKER_CERT_PATH
```

Run `docker info` to verify that the Docker CLI is communicating with the local daemon.

NEW with Docker EE 2.0 you can now import your own existing certificate. Because the public certificate is hashed, the CA is needed.



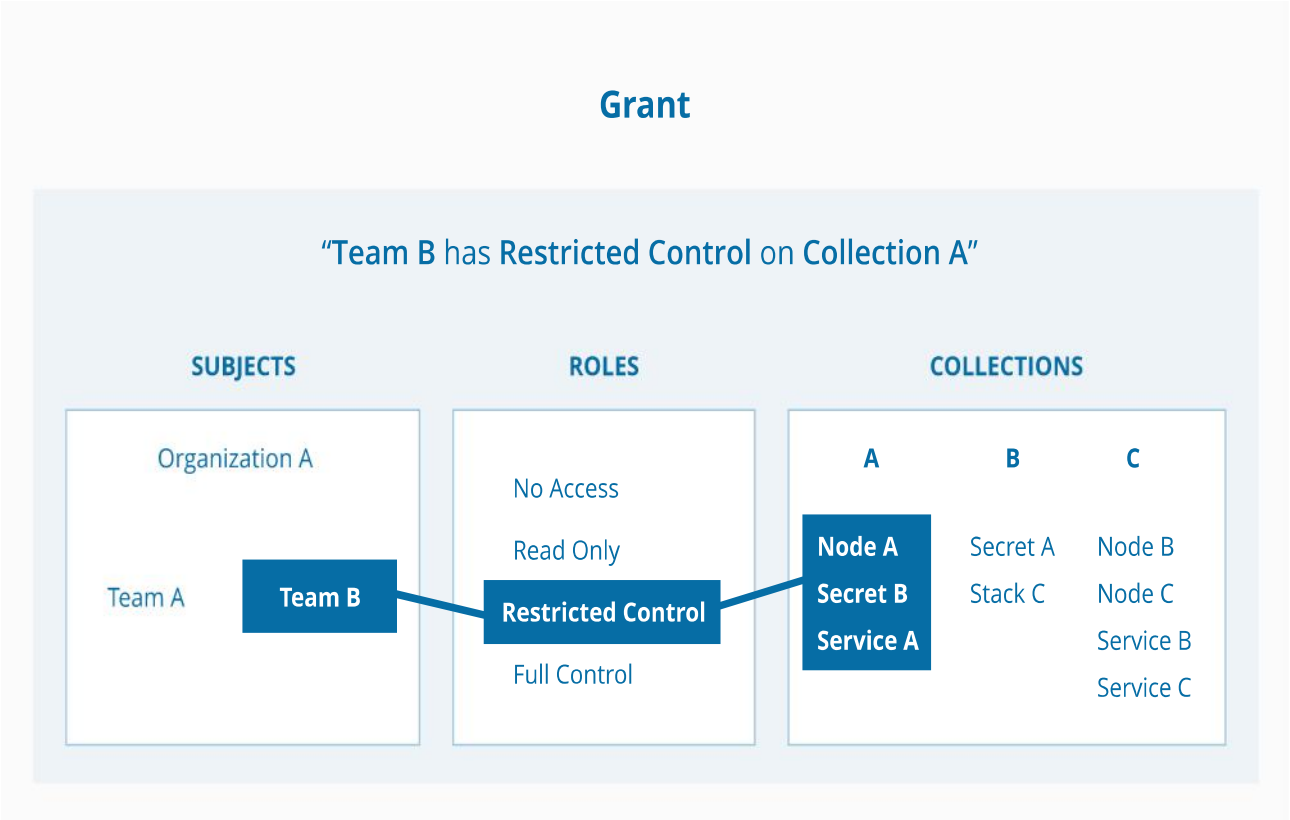
Cluster Role-Based Access Control

Docker EE 17.06 introduced a greatly enhanced Access Control system (https://docs.docker.com/datacenter/ucp/2.2/guides/access-control/) for UCP 2.2. (https://docs.docker.com/datacenter/ucp/2.2/guides/release-notes/). The new Access Control model provides an extremely fine-grained control of what resources users can access within a cluster. Use of RBAC is highly recommended for a secure cluster. Security principles of *least privilege* dictate the use of access control to limit access to resources whenever possible.

Access Control Policy

Docker EE Access Control is a policy-based model that uses access control lists (ACLs) called grants to dictate access between users and cluster resources. A grant ties together *who*, has permission for *which actions*, against *what resource*. They are a flexible way of implementing access control for complex scenarios without incurring high management overhead for the system administrators.

As shown below, a grant is made up of a *subject* (who), *role* (which permissions), and a *collection* (what resources).



Note: It is the UCP administrators' responsibility to create and manage the grants, subjects, roles, and collections.

Subjects

A subject represents a user, team, or organization. A subject is granted a role for a collection of resources. These groups of users are the same across UCP and DTR making RBAC management across the entire software pipeline uniform.

- User: A single user or system account that an authentication backend (AD/LDAP) has validated.
- Team: A group of users that share a set of permissions defined in the team itself. A team exists only as part of an organization, and all team members are members of the organization. A team can exist in one organization only. Assign users to one or more teams and one or more organizations.
- Organization: The largest organizational unit in Docker EE. Organizations group together teams to provide broader scope to apply access policy against.

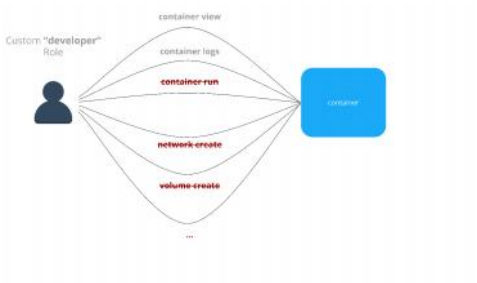
Roles and Permissions

A role is a set of permitted API operations that you can assign to a specific subject and collection by using a grant. Roles define what operations can be done against cluster resources. An organization will likely use several different kinds of roles to give the right kind of access. A given team or user may have different roles provided to them depending on what resource they are accessing. There are default roles provided by UCP, and there is also the ability to build custom roles.

Custom Roles

Docker EE defines very granular roles down to the Docker API level to match unique requirements that an organization may have. Roles and Permission Levels (https://docs.docker.com/datacenter/ucp/2.2/guides/access-control/permission-levels/) has a full list of the operations that can be used to build new roles.

For example, a custom role called *developer* could be created to allow developers to view and retrieve logs from their own containers that are deployed in production. A developer cannot affect the container lifecycle in any way but can gather enough information about the state of the application to troubleshoot application issues.



Built-In Roles

UCP also provides default roles that are pre-created. These are common role types that can be used to ease the burden of creating custom roles.

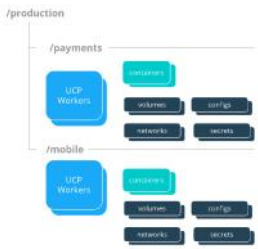
Built-In Role	Description
`None`	The user has no access to swarm resources. This maps to the `No Access` role in UCP 2.1.x.
`View Only`	The user can view resources like services, volumes, and networks but can't create them.
`Restricted Control`	The user can view and edit volumes, networks, and images but can't run a service or container in a way that might affect the node where it's running. The user can't mount a node directory and can't `exec` into containers. Also, the user can't run containers in privileged mode or with additional kernel capabilities.

Built-In Role	Description
`Scheduler`	The user can view nodes and schedule workloads on them. Worker nodes and manager nodes are affected by `Scheduler` grants. Having `Scheduler` access doesn't allow the user to view workloads on these nodes. They need the appropriate resource permissions, like `Container View`. By default, all users get a grant with the `Scheduler` role against the `/Shared` collection.
`Full Control`	The user can view and edit volumes, networks, and images. They can create containers without any restriction but can't see other users' containers.

Collections

Docker EE enables controlling access to swarm resources by using *collections*. A collection is a grouping of swarm cluster resources that you access by specifying a directory-like path. Before grants can be implemented, collections need to be designed to group resources in a way that makes sense for an organization.

The following example shows the potential access policy of an organization. Consider an organization with two application teams, Mobile and Payments, that share cluster hardware resources, but still need to segregate access to the applications. Collections should be designed to map to the organizational structure desired, in this case the two application teams.



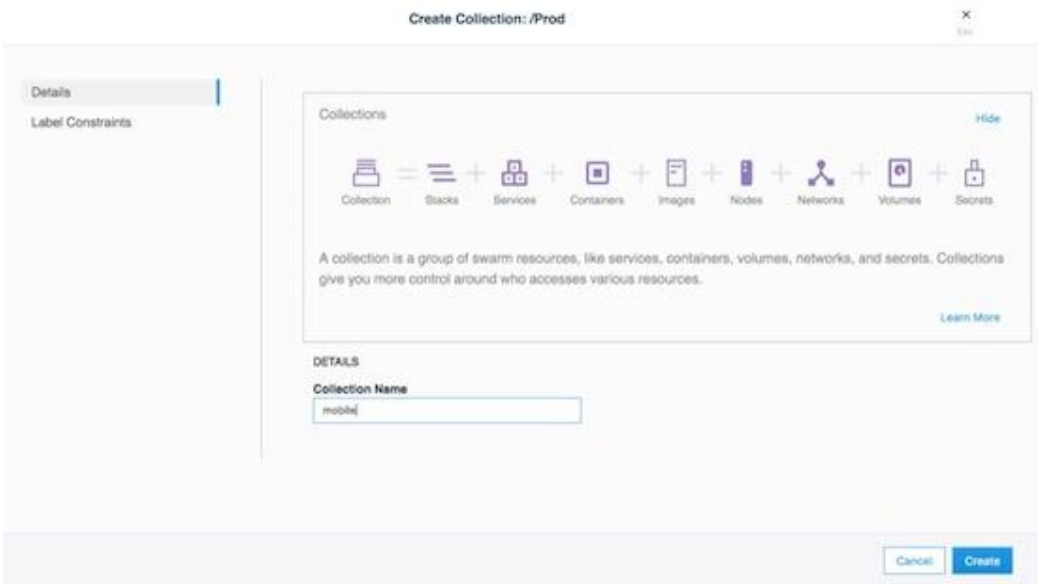
> Note: Permissions to a given collection are inherited by all children of that collection.

Collections are implemented in UCP through the use of Docker labels. All resources within a given collection are labeled with the collection, `/production/mobile` for instance.

Collections are flexible security tools because they are hierarchical. For instance, an organization may have multiple levels of access. This might neccessitate a collection architecture like the following:

```
├── production
│&amp;nbsp;&amp;nbsp; ├── database
│&amp;nbsp;&amp;nbsp; ├── mobile
│&amp;nbsp;&amp;nbsp; └── payments
│&amp;nbsp;&amp;nbsp;     ├── restricted
│&amp;nbsp;&amp;nbsp;     └── front-end
└── staging
    ├── database
    ├── mobile
    └── payments
        ├── restricted
        └── front-end
```

To create a child collection, navigate into the parent collection. Then create the child.



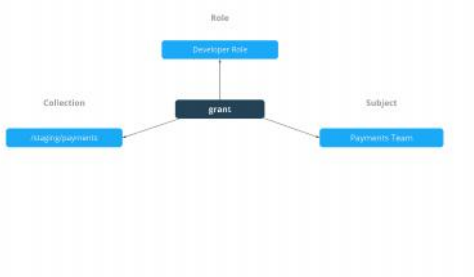
To add objects to collections, leverage labels. When deploying a stack make sure all objects are "labeled." Here is a good example of a few labels :

- Add an object to the `/production` collection: `com.docker.ucp.access.label: "/production"`
- Add an object to the `/production/mobile` collection: `com.docker.ucp.access.label: "/production/mobile"`

Adding nodes to a collection takes a little more care. Please follow the documentation (https://docs.docker.com/datacenter/ucp/2.2/guides/access-control/isolate-nodes-between-teams/#create-a-team) for isolating nodes to specific teams. Isolation nodes is a great way to provide more separation for multi-tenant clusters.

Grant Composition

When subjects, collections, and roles are set up, grants are created to map all of these objects together into a full access control policy. The following grant is one of many that might be created:



Together the grants clearly define which users have access to which resources. This is a list of some of the default grants in UCP that exist to provide an admin the appropriate access to UCP and DTR infrastructure.

Subject	Role	Collection
admin	Restricted Control	/Shared/Private/admin
admin	Scheduler	/Shared
admin	Full Control	/
Org - docker-datacenter	Scheduler	/

Secrets

Secrets were introduced with Docker EE Engine 1.13 as well as Docker EE 17.03. A *secret* is a blob of data such as a password, SSH private key, SSL certificate, or another piece of data that should not be transmitted over a network. Secrets are stored unencrypted in a Dockerfile or stored in your application's source code. Use Docker secrets to centrally manage this data and securely transmit it only to those containers that need access to it. Secrets follow a Least Privileged Distribution model and are encrypted at rest and in transit in a Docker swarm. A given secret is only accessible to those services which have been granted explicit access to it and only while those service tasks are running.

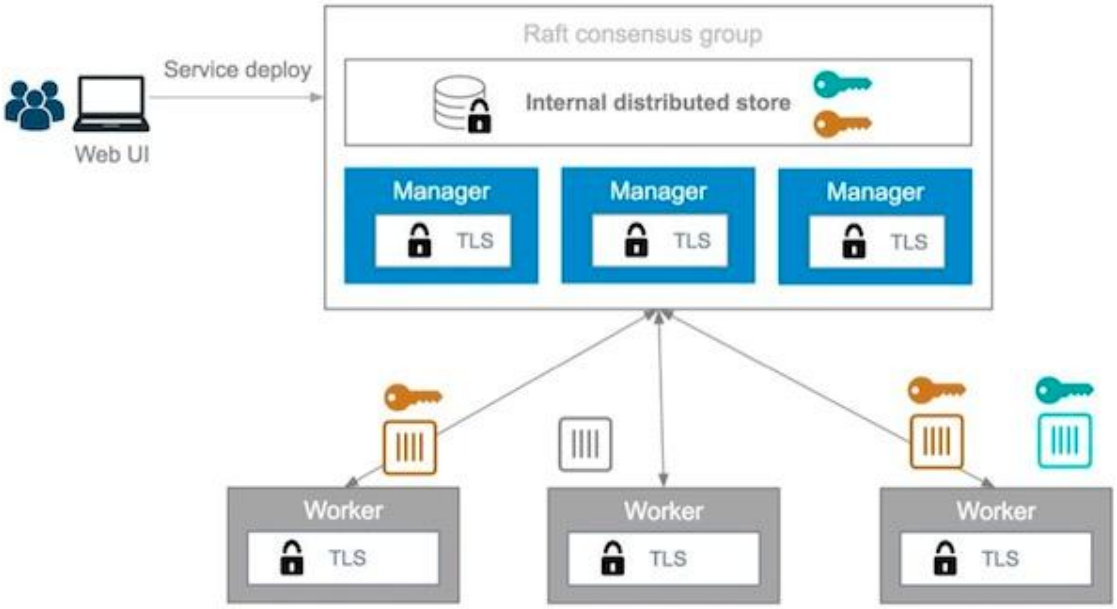
Secrets requires a swarm mode cluster. Use secrets to manage any sensitive data which a container needs at runtime but shouldn't be stored in the image or in source control such as:

- Usernames and passwords
- TLS certificates and keys
- SSH keys
- Other important data such as the name of a database or internal server
- Generic strings or binary content (up to 500 kb in size)

> Note: Docker secrets are only available to swarm services, not to standalone containers. To use this feature, consider adapting the container to run as a service with a scale of 1.

Another use case for using secrets is to provide a layer of abstraction between the container and a set of credentials. Consider a scenario where there have separate development, test, and production environments for an application. Each of these environments can have different credentials, stored in the development, test, and production swarms with the same secret name. The containers only need to know the name of the secret to function in all three environments.

When a secret is added to the swarm, Docker sends the secret to the swarm manager over a mutual TLS connection. The secret is stored in the Raft log, which is encrypted. The entire Raft log is replicated across the other managers, ensuring the same high availability guarantees for secrets as for the rest of the swarm management data.



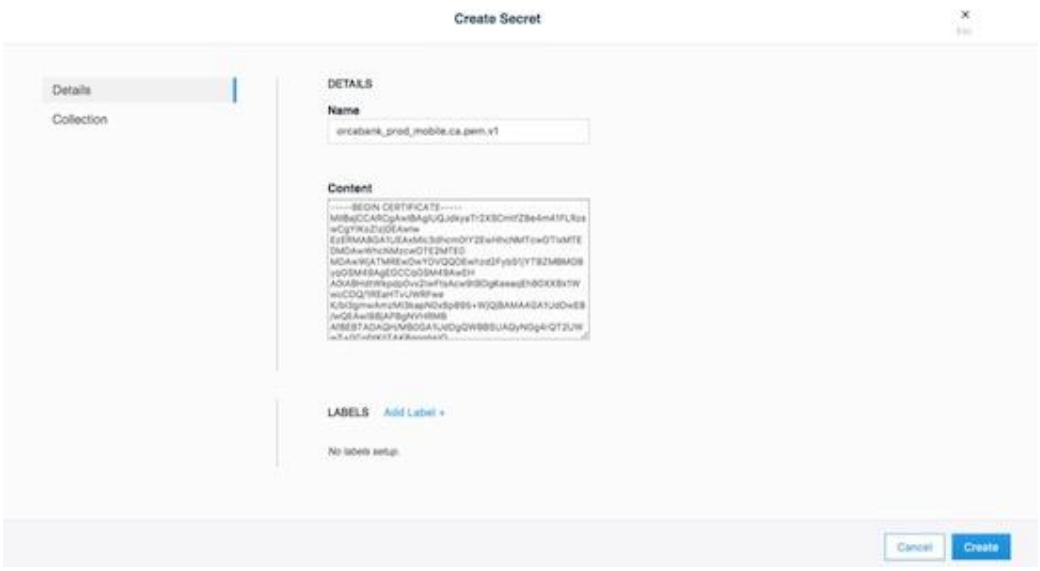
When a newly-created or running service is granted access to a secret, the decrypted secret is mounted into the container in an in-memory filesystem at `/run/secrets/<secret_name>` . It is possible to update a service to grant it access to additional secrets or revoke its access to a given secret at any time.

A node only has access to (encrypted) secrets if the node is a swarm manager or if it is running service tasks which have been granted access to the secret. When a container task stops running, the decrypted secrets shared to it are unmounted from the in-memory filesystem for that container and flushed from the node's memory.

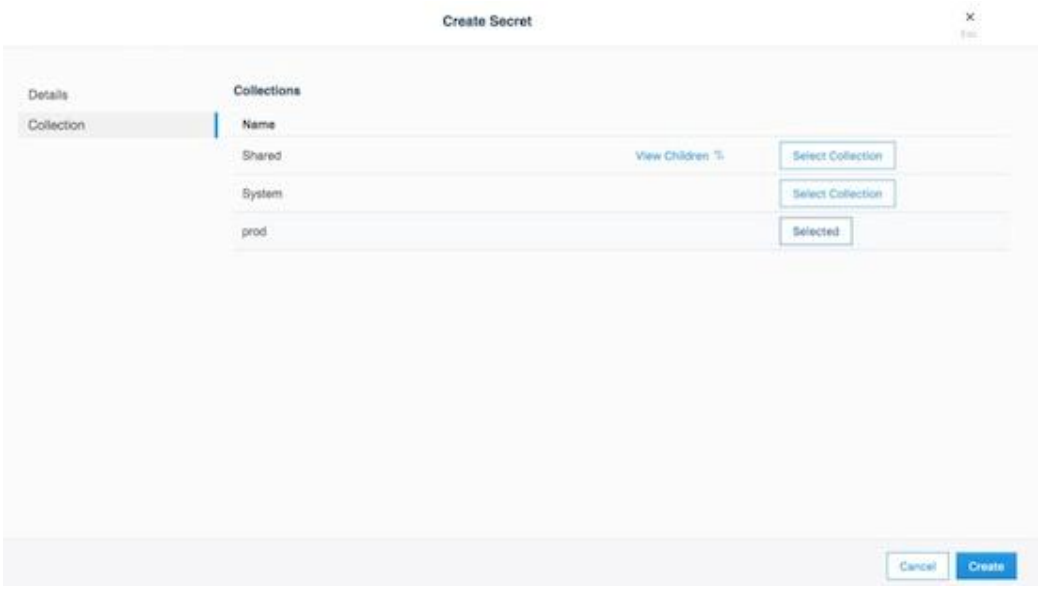
If a node loses connectivity to the swarm while it is running a task container with access to a secret, the task container still has access to its secrets but cannot receive updates until the node reconnects to the swarm.

Docker EE's strong RBAC system can tie *secrets* into it with the exact same labels demonstrated before, meaning you should always limit the scope of each secret to a specific team. If there are NO labels applied, the default label is the owner.

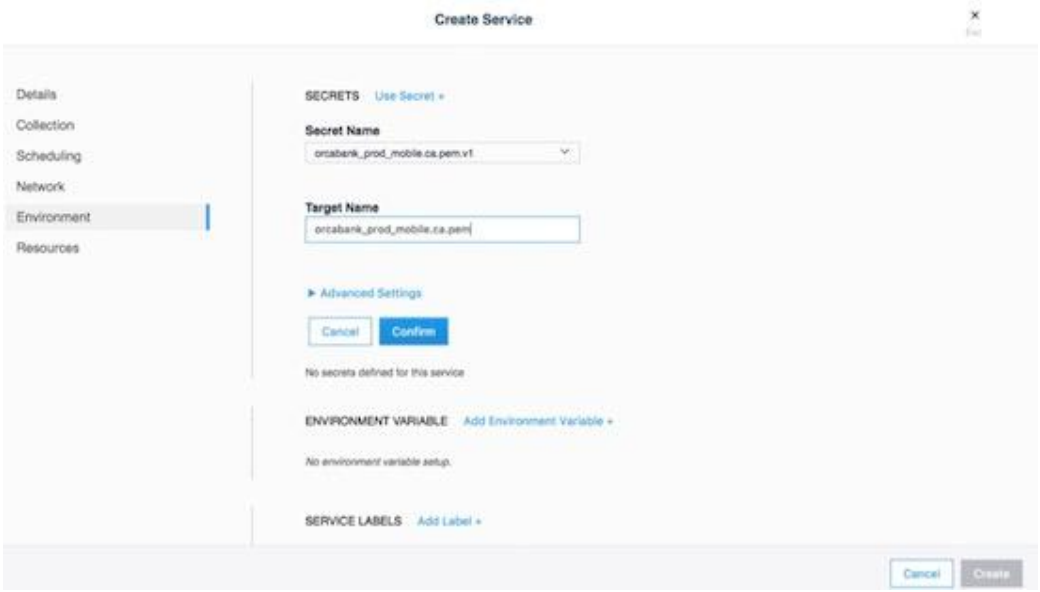
For example, TLS certificates can be added as secrets. Using the same RBAC example teams as previously mentioned, the following example adds `ca.pem`, `cert.pub`, and `cert.pem` to the secrets vault. Notice the use of the label `com.docker.ucp.access.label=/prod`. This is important for enforcing the RBAC rules. Also note the use of the team name in the naming of the secret. For another idea for updating or rolling back secrets, consider adding a version number or date to the secret name. This is made easier by the ability to control the mount point of the secret within a given container. This also prevents teams from trying to use the same secret name. Secrets can be found under the Swarm menu. The following adds the CA's public certificate in pem format as a secret named `orcabank_prod_mobile.ca.pem.v1`.



Next, set the collection the secret is in. Using the same example from above, select the `/prod` collection.



Secrets are only available to services. The following creates an `nginx` service. The service and the secret MUST be in the same collection. Again, apply the collection through the use of labels. If they don't match, UCP won't allow you to deploy. The next example deploys a service that can be used as a secret:

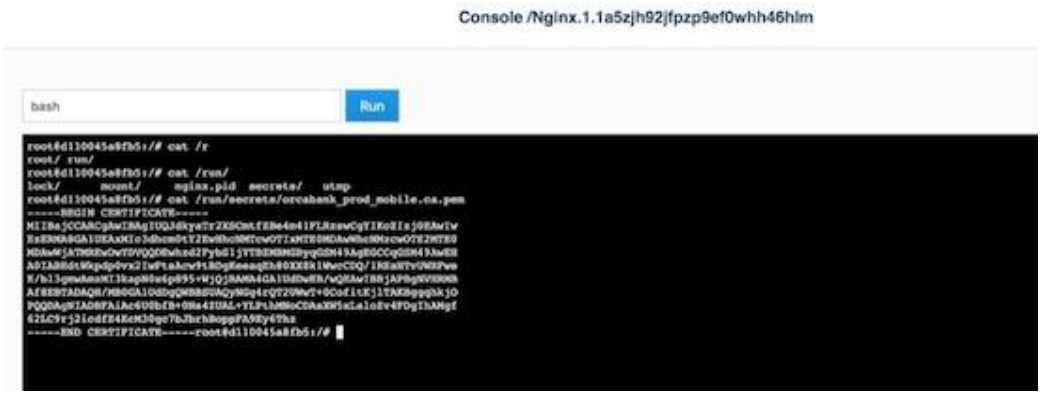


The important part is on the Environment tab. Click the + Use a secret. Use the advanced settings to configure the UID/GID and file mode for the secret when it is mounted. Binaries and tarballs can be added as secrets, with a file size up to 500KB. Be sure to click Confirm to add the secret.

When using the CLI, the option, `--secret source=,target=,mode=` needs to be added to the `docker service create` command as follows:

```
$ docker service create \
--secret source=orcabank_prod_mobile.ca.pem.v1,target=ca.pem \
--secret source=orcabank_prod_mobile.cert.pub.v1,target=cert.pub \
--secret source=orcabank_prod_mobile.cert.pem.v1,target=cert.pem \
-l com.docker.ucp.access.label=/prod -p 443 --name nginx nginx
```

Notice that the secrets are mounted to `/run/secrets/`. Because of labels in this example, only administrators and the crm team have access to this container and its secrets.



Changing secrets is as easy as removing the current version and creating it again. *Be sure the labels on the new secret are correct.*

Logging

Since UCP is deployed as a containerized application, it uses the Engine logging configuration automatically. However, there is an easier way to configure logging across the cluster if the syslog format is being used. In Admin Settings -> Logs set the logging level and destination.

This is a great way to point all the logs to Splunk or an ELK (Elasticsearch Logstash Kibana) stack, as well as change the logging level.

DTR Security

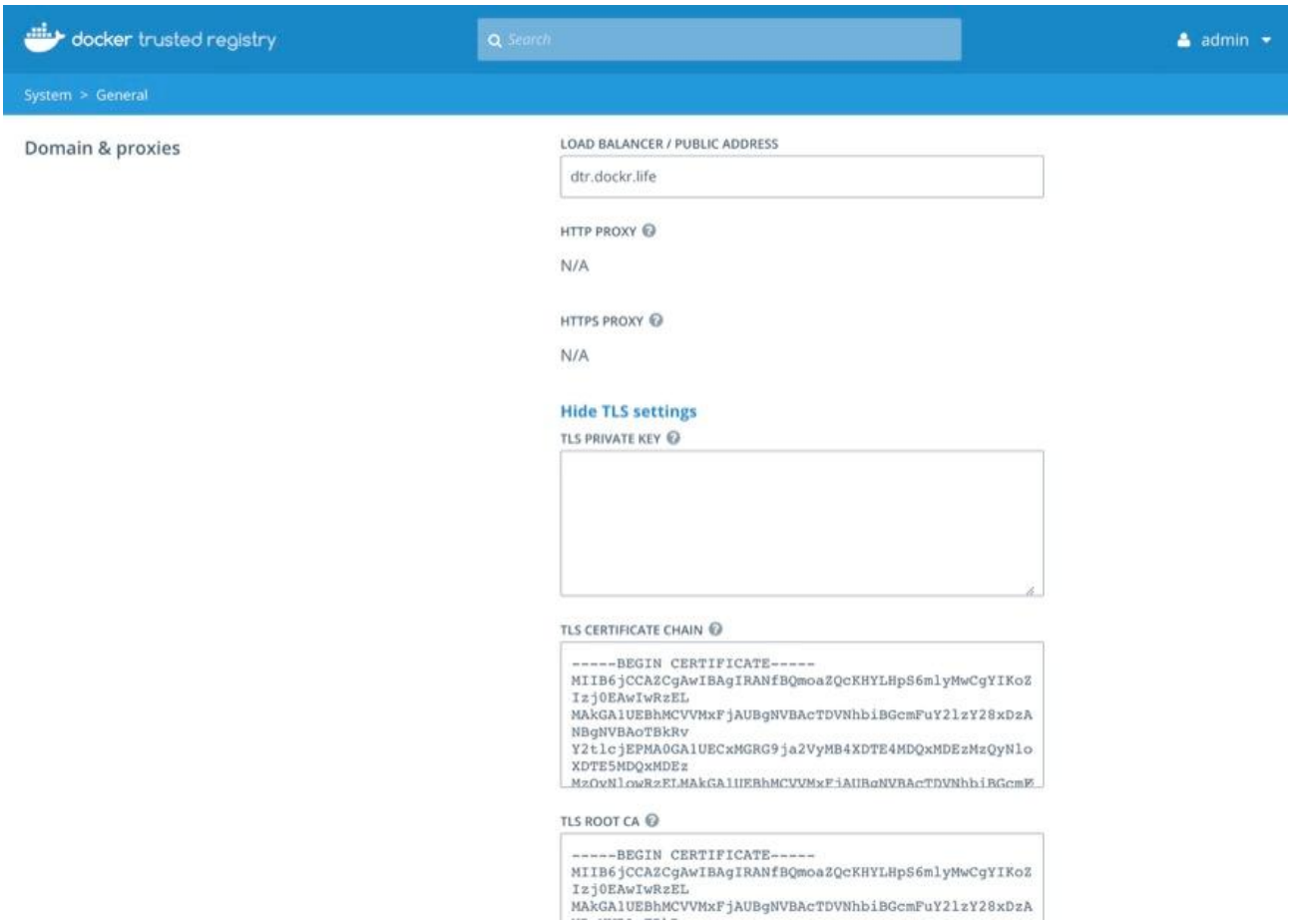
Docker Trusted Registry continues the "Secure by Default" theme with two new strong features: *Image Signing* (via the Notary project) and *Image Scanning*. Additionally, DTR shares authentication with UCP, which simplifies setup and provides strong RBAC without any effort.

DTR stores metadata and layer data in two separate locations. The metadata is stored locally in a database that is shared between replicas. The layer data is stored in a configurable location.

External Certificates

Just as with UCP, DTR can use fully-signed company certificates or self-signed certs. The Certificate Authority (CA) for the organization can be used. To reduce the number of certificates, add multiple Subject Alternative Names (SANs) to a single certificate. This allows the certificate to be valid for multiple URLs. For example, when setting up a certificate for `ucp.example.com`, add SANs of `dtr.example.com` and all the underlying hostnames and IP addresses. Using this technique allows the same certificate to be used for both UCP and DTR.

External certificates are added to DTR by going to Settings -> General -> Domain & proxies -> Show TLS Settings.



For more instructions on adding external certificates, refer to the Docker docs (https://docs.docker.com/datacenter/dtr/2.2/guides/admin/configure/use-your-own-tls-certificates/).

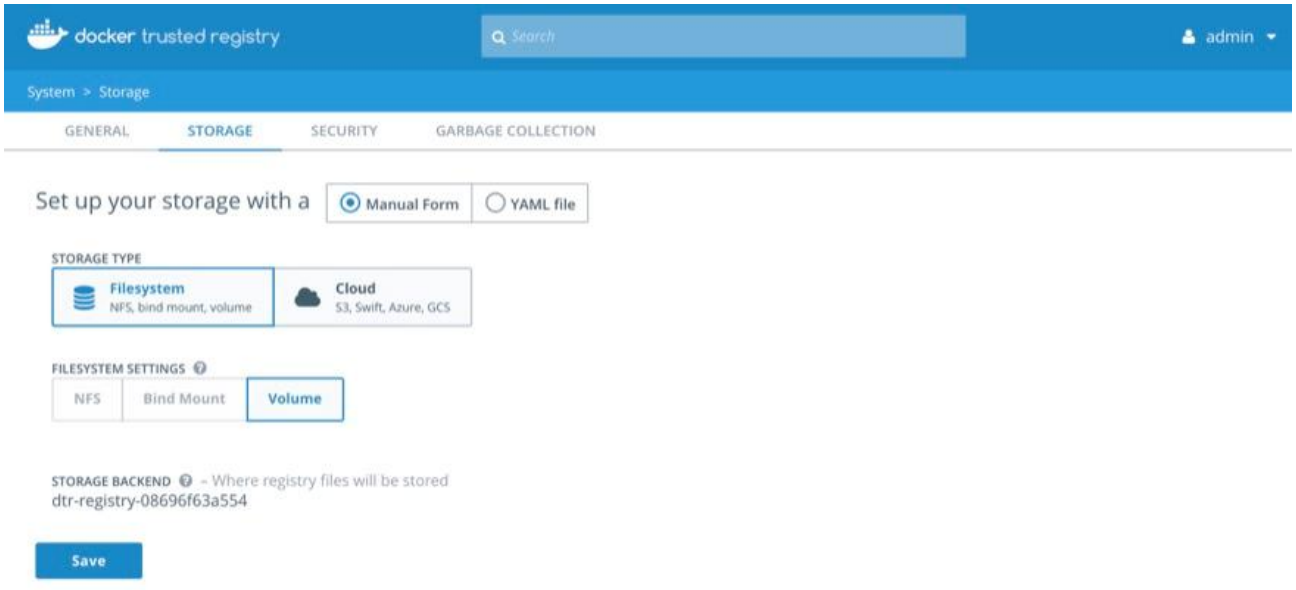
Storage Backend — S3 or NFS

The choice of the storage backend for DTR has effects on both performance and security. The choices are as follows:

Type	Pros	Cons
Local Filesystem	Fast and Local. Pairs great with local block storage.	Requires bare metal or ephemeral volumes. NOT good for HA.

Type	Pros	Cons
S3	Great for HA and HTTPS communications. Several third party servers available. Can be encrypted at rest. (http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingEncryption.html)	Requires maintaining or paying for an external S3 compliant service.
Azure Blob Storage	Can be configured to act as local but have redundancy within Azure Storage. Can be encrypted at rest. (https://docs.microsoft.com/en-us/azure/storage/common/storage-service-encryption)	Requires Azure cloud account.
Swift	Similar to S3 being an object store.	Requires OpenStack infrastructure for service.
Google Cloud Storage	Similar to S3 being an object store. Can be encrypted at rest.	Requires a Google Cloud account.
NFS	Easy to setup/integrate with existing infrastructure.	Slower due to network calls.

To change the settings, go to Settings -> Storage in UCP.



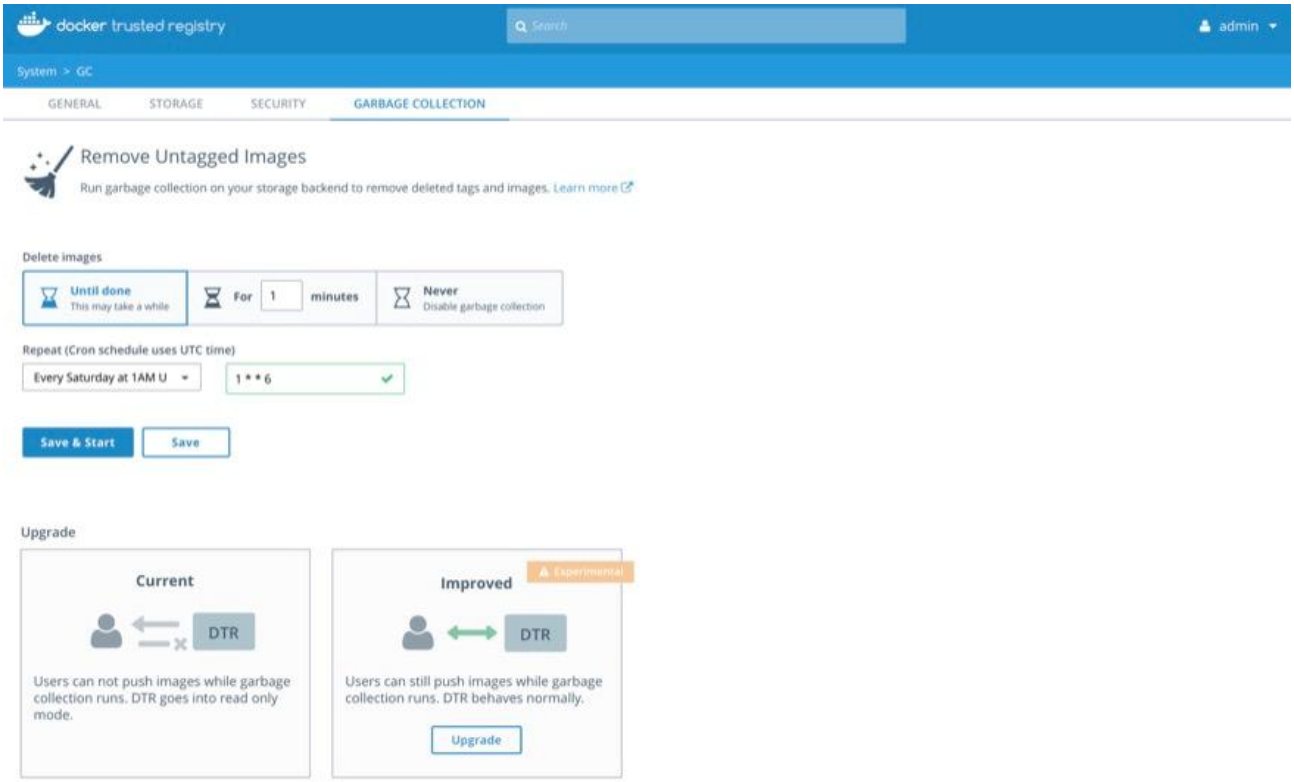
Storage choice is highly influenced by where Docker EE is deployed because it is important to place DTR's backend storage as close as possible to DTR itself. Always ensure that HTTPS (TLS) is being used. Also consider how to backup DTR's images. When in doubt, use a secure object store, such as S3 or similar. Object stores provide the best balance between security and ease of use and also make it easy for HA DTR setups.

Garbage Collection

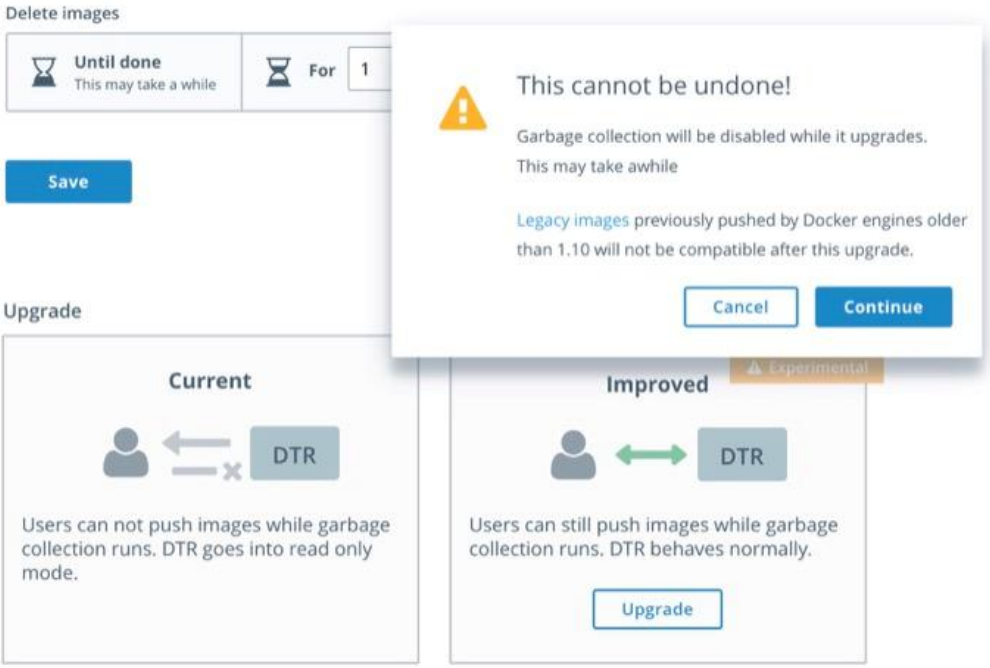
Garbage collection is an often-overlooked area from a security standpoint. Old, out-of-date images may contain security flaws or exploitable vulnerabilities, so removing unnecessary images is important. Garbage collection is a feature that ensures that unreferenced images (and layers) are removed.

With Docker EE 2.0 there is a new experimental online Garbage Collection. The current Garbage Collection is a blocking process, so it's best to run at times when the system is least utilized.

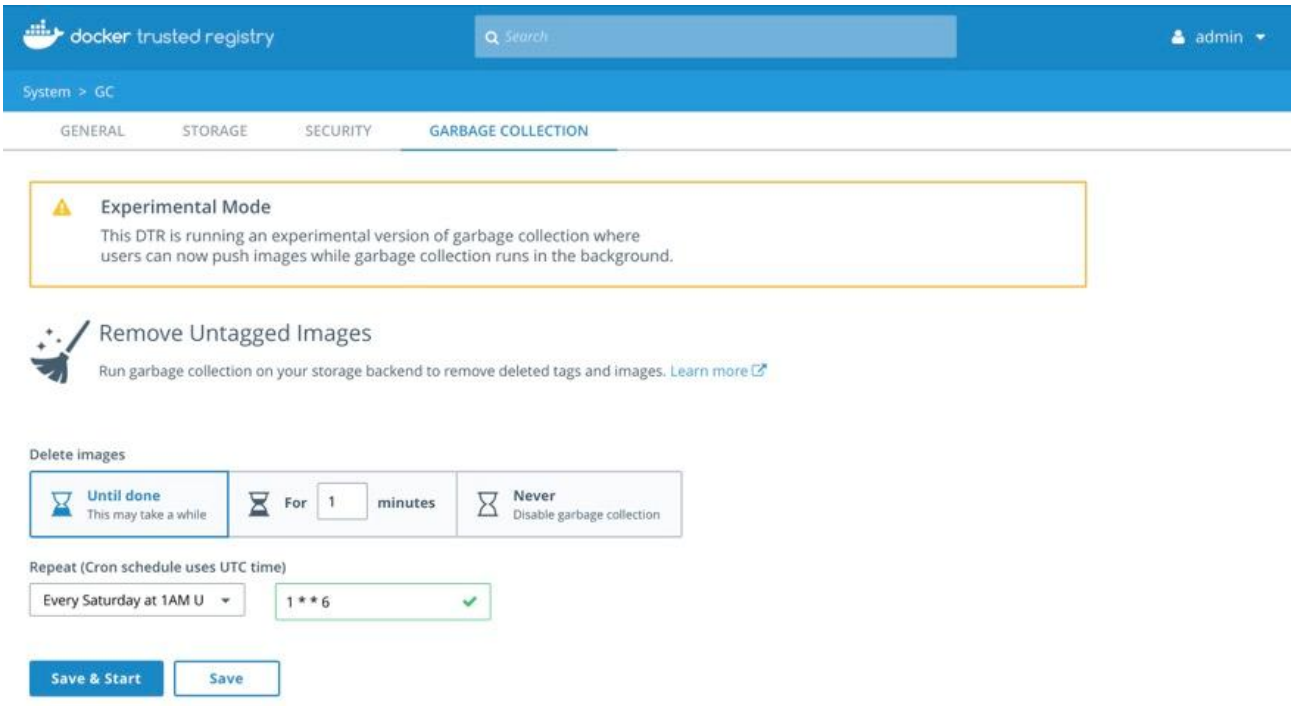
To scheduling the current Garbage Collection navigate to Settings -> Garbage Collection. The current best practices is to create a schedule for every Saturday or Sunday and Until Done. Click Save & Start.



The Improved Garbage Collection is no longer a blocking process. Meaning that it can be run online without a scheduled slow period. Upgrading to the experimental Improved Garbage Collection can not be reversed.



With the experimental Improved Garbage Collection enabled the process still needs to be scheduled. Schedule it to run identically as before. The current best practices is to create a schedule for every Saturday or Sunday and Until Done. Click Save & Start.



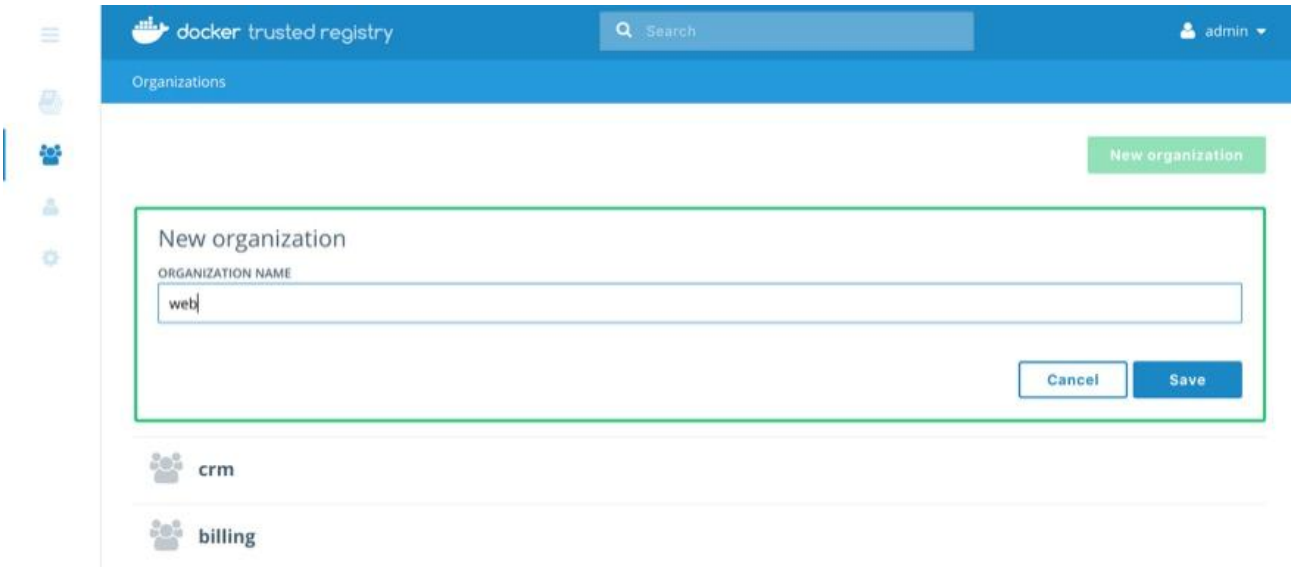
Organizations and Teams — RBAC

Since Universal Control Plane and Docker Trusted Registry utilize the same authentication backend, users are shared between the two. This simplifies user management since UCP 2.2 and DTR 2.3 organizations are now shared. That means DTR and UCP can manage the organizations and teams. Consider the differences between organizations and teams. Teams are nested underneath organizations. Teams allow for a finer grain control of access.

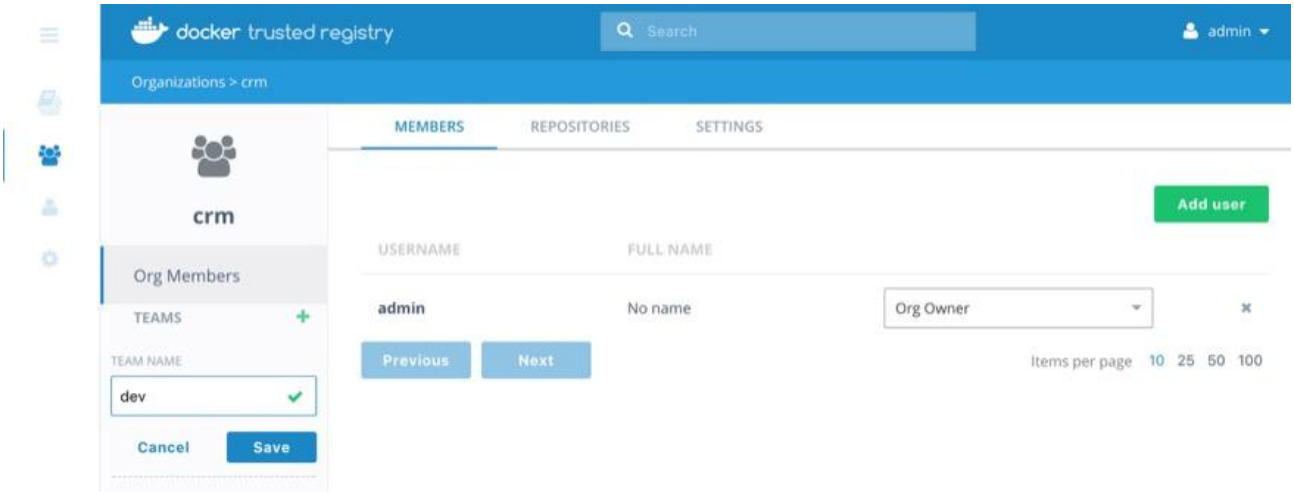
Here's an overview of the permission levels available for organizations and users:

- Anonymous users: Search and pull public repositories.
- Users: Search and pull public repos. Create and manage their own repositories.
- Team member: Can do everything a user can do plus the permissions granted by the teams the user is a member of.
- Team admin: Can do everything a team member can do, and can also add members to the team.
- Organization admin: Can do everything a team admin can do, can create new teams, and add members to the organization.
- Admin: Can manage anything across UCP and DTR.

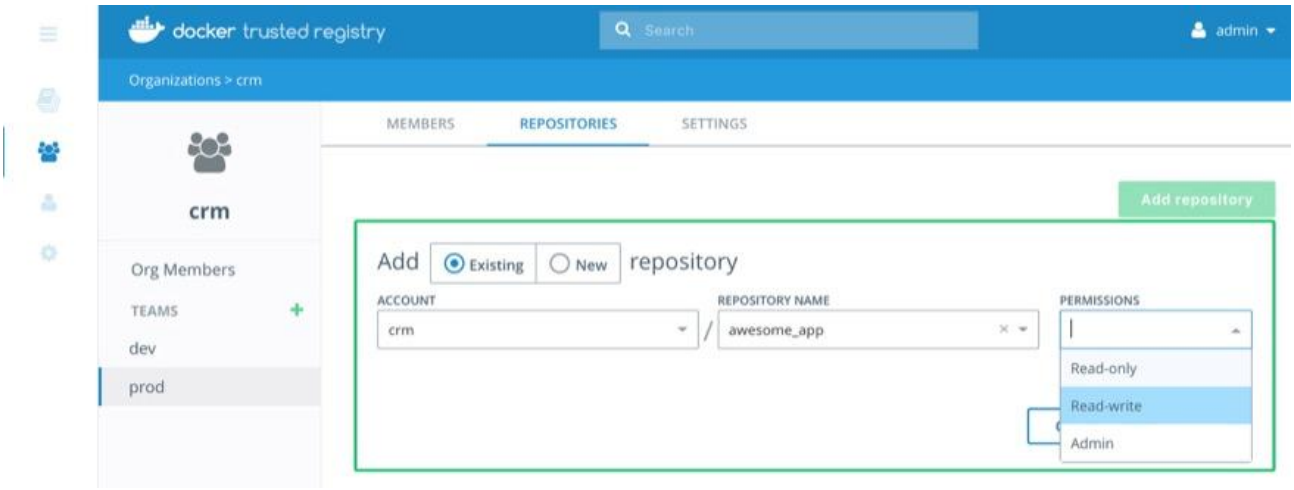
The following example creates an organization called `crm`:



Once the organizations are created, add teams to the organization.



For example, an organization named `crm`, a team named `prod`, and a repository named `crm/awesome_app` were created. Permissions can now be applied to the images themselves.



This chart shows the different permission levels for a team against a repository:

Repository Operation	read	read-write	admin
View / browse	x	x	x
Pull	x	x	x
Push		x	x
Delete tags		x	x
Edit description			x
Set public or private			x
Manage user access			x
Delete repository			

It is important to limit the number of users that have access to images. Applying the permission levels correctly is important. This helps in creating a Secure Supply Chain.

Content Trust and Image Signing with Notary

Notary is a tool for publishing and managing trusted collections of content. Publishers can digitally sign collections and consumers can verify integrity and origin of content. This ability is built on a straightforward key management and signing interface to create signed collections and configure trusted publishers.

Docker Content Trust/Notary provides a cryptographic signature for each image. The signature provides security so that the image requested is the image you get. Read Notary's Architecture (https://docs.docker.com/notary/service_architecture/) to learn more about how Notary is secure. Since Docker EE is "Secure by Default," Docker Trusted Registry comes with the Notary server out of the box.

In addition, Docker Content Trust allows for threshold signing and gating for the releases. Under this model, software is not released until all necessary parties (or a quorum) sign off. This can be enforced by requiring (and verifying) the needed signatures for an image. This policy ensures that the image has made it through the whole process: if someone tries to make it skip a step, the image will lack a necessary signature, thus preventing deployment of that image.

The following examples shows the basic usage of Notary. To use image signing, create a repository in DTR and enable it on the local Docker engine. First, enable the client, and sign an image:

```
root @ ~  export DOCKER_CONTENT_TRUST=1
root @ ~  docker tag alpine dtr.example.com/admin/alpine:signed
root @ ~  docker push dtr.example.com/admin/alpine:signed
The push refers to a repository [dtr.example.com/admin/alpine]
865e1c468a35: Layer already exists
e0cfcaccf697: Layer already exists
e2d4ee32e967: Layer already exists
60ab55d3379d: Layer already exists
signed: digest: sha256:131d77d4ccf5916a94d026e2f5865a2e2acefd56fc6debceb83e50cf24eb4e99 size
: 1156
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system. Please
choose a long, complex passphrase and be careful to keep the password and the
key file itself secure and backed up. It is highly recommended that you use a
password manager to generate the passphrase and keep it safe. There will be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with ID 44d193b:
Repeat passphrase for new root key with ID 44d193b:
Enter passphrase for new repository key with ID 2a0738c (dtr.example.com/admin/alpine):
Repeat passphrase for new repository key with ID 2a0738c (dtr.example.com/admin/alpine):
Finished initializing "dtr.example.com/admin/alpine"
Successfully signed "dtr.example.com/admin/alpine":signed
```

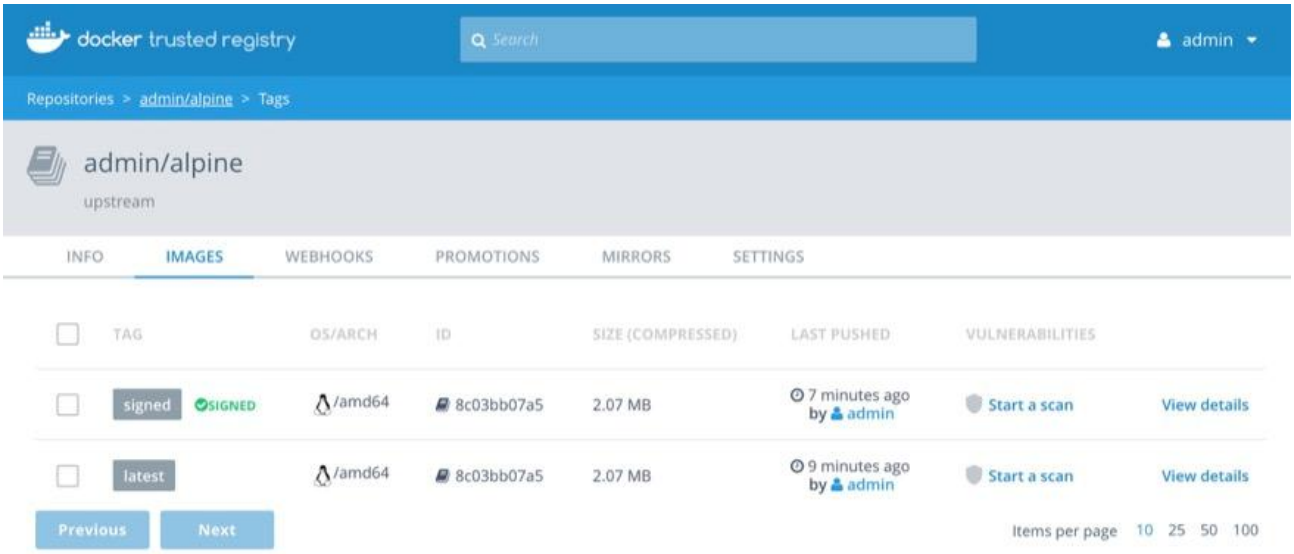
The above does the following:

- Enables Content Trust with `export DOCKER_CONTENT_TRUST=1`
- Tags an image destined for DTR with `docker tag alpine dtr.example.com/admin/alpine:signed`
- Pushes the image with the tag "signed" with `docker push dtr.example.com/admin/alpine:signed`
- The Docker client starts the push to DTR. Since there is no local root key (certificate), it needs to be created with a passphrase. `Enter passphrase for new root key with ID 44d193b:`
- A repository key is created with a passphrase. `Enter passphrase for new repository key with ID f93c4a5 (dtr.example.com/admin/alpine):`

When re-pushing signed images to DTR, the keys do not need to be created again. It will prompt for the image passphrase.

```
root @ ~  docker push dtr.example.com/admin/alpine:signed
The push refers to a repository [dtr.example.com/admin/alpine]
60ab55d3379d: Layer already exists
signed: digest: sha256:3952dc48dcc4136ccdde37fbef7e250346538a55a0366e3fccc683336377e372 size
: 528
Signing and pushing trust metadata
Enter passphrase for repository key with ID 2a0738c:
Successfully signed "dtr.example.com/admin/alpine":signed
```

A successfully signed image has a green check mark in the DTR GUI.



Key Management

Docker and Notary clients store state in its `trust_dir` directory, which is `~/.docker/trust` when enabling Docker Content Trust. This directory is where all the keys are stored. All the keys are encrypted at rest. It is VERY important to protect that directory with permissions.

The `root_keys` subdirectory within `private` stores root private keys, while `tuf_keys` stores targets, snapshots, and delegations private keys.

Interacting with the local keys requires the installation of the Notary client. Binaries can be found at https://github.com/docker/notary/releases (https://github.com/docker/notary/releases). Here is a quick installation script:

```
$ wget -O /usr/local/bin/notary https://github.com/theupdateframework/notary/releases/downlo
ad/v0.6.0/notary-Linux-amd64
$ chmod 755 /usr/local/bin/notary
```

At the same time, getting the notary client DTR's CA public key is also needed. Assuming Centos/Rhel :

```
$ sudo curl -sk https://dtr.example.com/ca -o /usr/local/share/ca-certificates/dtr.example.c
om.crt
```

It is easy to simplify the notary command with an alias.

```
$ alias notary="notary -s https://dtr.example.com -d ~/.docker/trust --tlscacert /usr/local/
share/ca-certificates/example.com.crt"
```

With the alias in place, run `notary key list` to show the local keys and where they are stored.

```
ROLE       GUN                          KEY ID
                LOCATION
----       ---                          ------
                --------
root                                    44d193b5954facdb5f21584537774b9732cfea91e5d753107582
2c58f979cc93    /root/.docker/trust/private
targets    ...ullet.com/admin/alpine    2a0738c4f75e97d3a5bbd48d3e166da5f624ccb86899479ce238
1d4e268834ee    /root/.docker/trust/private
```

To make the keys more secure it is recommended to always store the `root_keys` offline, meaning, not on the machine used to sign the images. If that machine were to get compromised, then an unauthorized person would have everything needed to sign "bad" images. Yubikey is a really good method for storing keys offline.

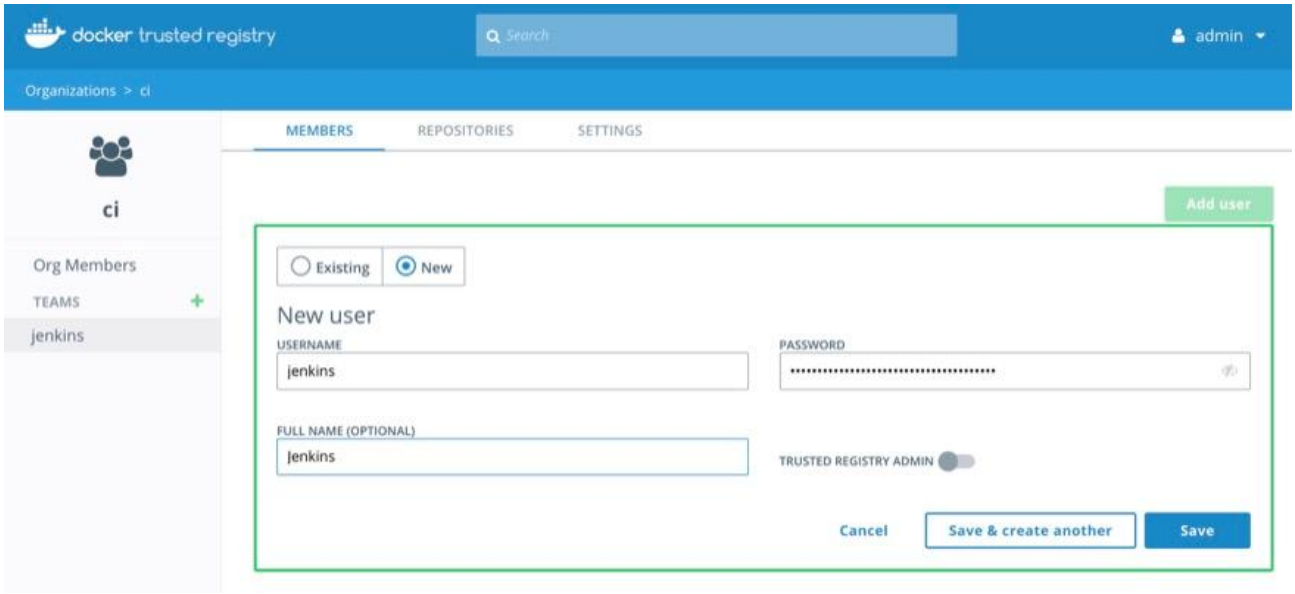
Use a Yubikey

Notary can be used with a hardware token storage device called a Yubikey (https://www.yubico.com/products/yubikey-hardware/). The Yubikey must be prioritized to store root keys and requires user touch-input for signing. This creates a two-factor authentication for signing images. Note that Yubikey support is included with the Docker Engine 1.11 client for use with Docker Content Trust. The specific use is to have all of your developers use Yubikeys with their workstations. Get more information about Yubikeys from the Docker docs (https://docs.docker.com/notary/advanced_usage/#/use-a-yubikey).

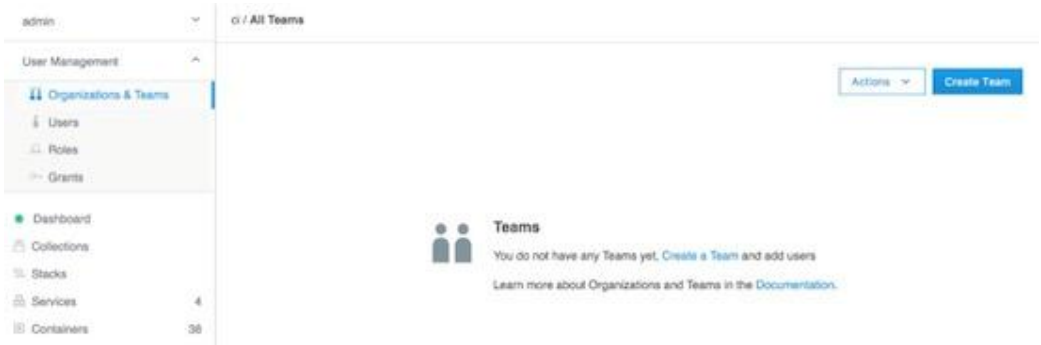
Signing with Jenkins

When teams get large, it becomes harder to manage all the developer keys. One method for reducing the management load is to not let developers sign images. Using Jenkins to sign all the images that are destined for production eliminates most of the key management. The keys on the Jenkins server still need to be protected and backed up.

The first step is to create a user account for your CI system. For example, assume Jenkins is the CI system. As an admin user, navigate to Organizations and select New organization. Assume it is called "ci". Next, add a Jenkins user by navigating into the organization and selecting Add User. Create a user with the name `jenkins` and set a strong password. This will create a new user and add the user to the "ci" organization. Next, give the Jenkins user "Org Admin" status so the user is able to manage the repositories under the "ci" organization.

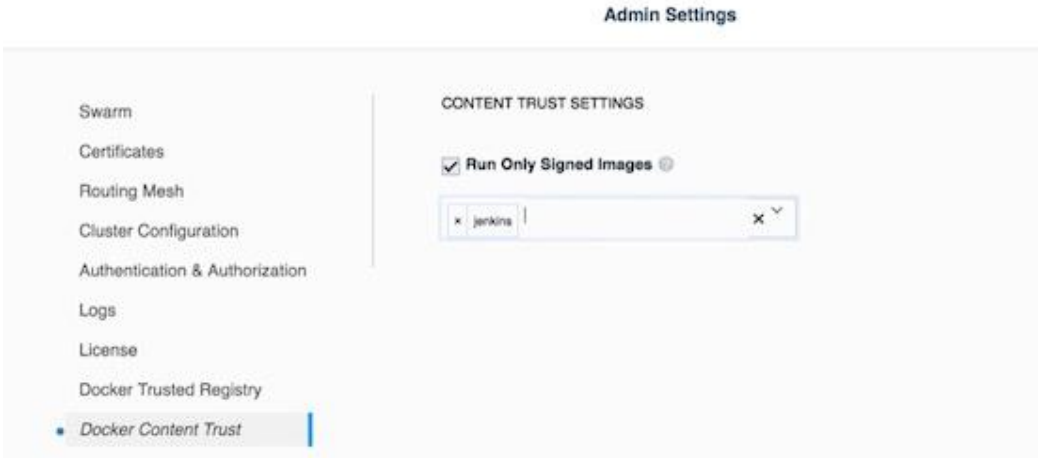


Navigate to UCP's User Management and create a team under the "ci" organization. Assume this team is named "jenkins".



Now that the team is setup, turn on the policy enforcement. Navigate to Admin Settings and then the Docker Content Trust subsection. Select the "Run Only Signed Images" checkbox to enable Docker Content Trust. In the select box that appears, select the "jenkins" team that was just created. Save the settings.

This policy requires every image that is referenced in a `docker pull`, `docker run`, or `docker service create` be signed by a key corresponding to a member of the "jenkins" team. In this case, the only member is the `jenkins` user.



The signing policy implementation uses the certificates issued in user client bundles to connect a signature to a user. Using an incognito browser window (or otherwise), log into the `jenkins` user account created earlier. Download a client bundle for this user. It is also recommended to change the description associated with the public key stored in UCP such that it can be identify in the future as the key being used for signing.

Please note each time a user retrieves a new client bundle, a new keypair is generated. It is therefore necessary to keep track of a specific bundle that a user chooses to designate as the user's signing bundle.

Once the client bundle has been decompressed, the only two files needed for the purpose of signing are `cert.pem` and `key.pem`. These represent the public and private parts of the user's signing identity respectively. Load the `key.pem` file onto the Jenkins servers, and use `cert.pem` to create delegations for the `jenkins` user in the Trusted Collection.

On the Jenkins server, use the notary client to load keys. Simply run `notary -d /path/to/.docker/trust key import /path/to/key.pem`. When prompted, set a password to encrypt the key on disk. For automated signing, this password can be configured into the environment under the variable name `DOCKER_CONTENT_TRUST_REPOSITORY_PASSPHRASE`. The `-d` flag to the command specifies the path to the trust subdirectory within the server's Docker configuration directory. Typically this is found at `~/.docker/trust`.

There are two ways to enable Content Trust: globally and per operation. To enabled Content Trust globally, set the environment variable `DOCKER_CONTENT_TRUST=1`. To enable on a per operation basis, wherever `docker push` is run in the Jenkins scripts, add the flag `--disable-content-trust=false`. To sign only certain images, use the second option.

The Jenkins server is now prepared to sign images, but delegations are needed to reference the key to

give it the necessary permissions.

Any commands displayed in this section should not be run from the Jenkins server. They can be run them from the local system.

If this is a new repository, create it in Docker Trusted Registry (DTR).

Next, initialize the trust data and create the delegation that provides the Jenkins key with permissions to sign content. The following commands initialize the trust data and rotate snapshotting responsibilities to the server. This is necessary to ensure human involvement it not required to publish new content.

Create an alias to streamline all of the following commands. The alias sets the server and the default trust store location. Adding the CA for DTR's TLS certificate is needed if the certificate is signed by a root server.

```
$ alias notary="notary -s https://dtr.example.com -d ~/.docker/trust --tlscacert ~/.docker/t
ls/dtr.example.com/ca.crt"
```

Initialize the repository if the signed image hasn't been pushed:

```
$ notary init dtr.example.com/admin/alpine
$ notary key rotate dtr.example.com/admin/alpine snapshot -r
$ notary publish dtr.example.com/admin/alpine
```

Now that the repository is initialized, create the delegations for Jenkins. Docker Content Trust treats a delegation role called `targets/releases` in a special way. It considers this delegation to contain the canonical list of published images for the repository. It is therefore generally desirable to add all users to this delegation with the following command:

```
$ notary delegation add dtr.example.com/admin/alpine targets/releases --all-paths /path/to/c
ert.pem
```

This solves a number of prioritization problems that would result from needing to determine which delegation should ultimately be trusted for a specific image. However, because it is anticipated that any user will be able to sign the targets/releases role, it is not trusted in determining if a signing policy has been met. Therefore it is also necessary to create a delegation specifically for Jenkins:

```
$ notary delegation add dtr.example.com/admin/alpine targets/jenkins --all-paths /path/to/ce
rt.pem
```

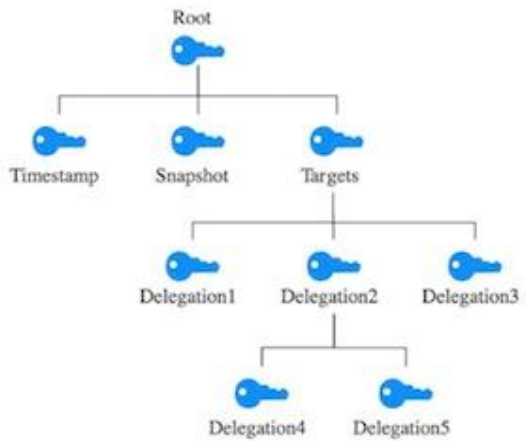
Next publish both these updates (remember to add the correct `-s` and `-d` flags):

```
$ notary publish dtr.example.com/admin/alpine
```

> Informational (Advanced): When including the targets/releases role in determining if a signing policy had been met, there is the potential of images being opportunistically deployed when an appropriate user signs it. In the scenario described so far, only images signed by the `CI` team (containing only the `jenkins` user) should be deployable. If a user `Moby` could also sign images but was not part of the `CI` team, he might sign and publish a new target/release that contained his image. UCP would refuse to deploy this image because it was not signed by the `CI` team. However, the next time Jenkins published an image, it would update and sign the targets/releases role as whole, enabling `Moby` to deploy his image.

Key Delegation

Similar to delegating a key for Jenkins, multiple keys can be delegate for teams. There are a few tricks for this. When adding a delegation, it is recommended to add the delegation for the targets/releases role as well as a role to indicate the team.



For example, with three teams (`developer` , `qa` , and `devops`) in the targets/releases role, it would be best to add each to the targets/releases role and also create a role for each key:

```
# Keep in mind the '-p' is to automatically publish.
# create delegation for the targets/releases role for each of the keys
$ notary delegation add -p dtr.exampleexample.com/admin/alpine targets/releases --all-paths
~/developer.pem ~/qa.pem ~/devops.pem

# create delegation to the targets/developer role
$ notary delegation add -p dtr.example.com/admin/alpine targets/developer --all-paths ~/deve
loper.pem

# create delegation to the targets/qa role
$ notary delegation add -p dtr.example.com/admin/alpine targets/qa --all-paths ~/qa.pem

# create delegation to the targets/devops role
$ notary delegation add -p dtr.example.com/admin/alpine targets/devops --all-paths ~/devops.
pem
```

This is ideal so in the case of everyone pushing to the same tag, it results in the same hash in targets/developer, targets/qa, and targets/devops, and then whoever signed last also signed the same hash into targets/releases. Without the signature on targets/releases, the image can't be pulled. With individual roles for each, additional data is given about which signatures are actually in place based off of which key.

Lost key? Rotate it?

If the root or signing key is lost, all hope is not lost. The keys can simply be rotated. Then re-pushing the image will trigger a resigning.

To rotate the "targets" (signing) key:

```
root @ ~  notary key rotate dtr.example.com/admin/alpine targets
Enter passphrase for new targets key with ID 00aeaf3 (dtr.example.com/admin/alpine):
Repeat passphrase for new targets key with ID 00aeaf3 (dtr.example.com/admin/alpine):
Enter username: admin
Enter password:
Enter passphrase for root key with ID 2a0738c:
Successfully rotated targets key for repository dtr.example.com/admin/alpine
```

Notice that a new passphrase is entered for the target key. Also note that Notary removes the old targets key and replaces it with the new one. The behavior is a little different for the root key. Notary keeps the old root key to ensure the downstream clients can transition. Next, rotate the root key:

```
root @ ~  notary key rotate dtr.example.com/admin/alpine root
Warning: you are about to rotate your root key.

You must use your old key to sign this root rotation. We recommend that
you sign all your future root changes with this key as well, so that
clients can have a smoother update process. Please do not delete
this key after rotating.

Are you sure you want to proceed?  (yes/no)  yes
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system. Please
choose a long, complex passphrase and be careful to keep the password and the
key file itself secure and backed up. It is highly recommended that you use a
password manager to generate the passphrase and keep it safe. There will be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with ID 75cb534:
Repeat passphrase for new root key with ID 75cb534:
Enter username: admin
Enter password:
Successfully rotated root key for repository dtr.example.com/admin/alpine
```

Remember to keep the keys private, and when possible, use a hardware token like a Yubikey. Currently only the Yubikey version 4 is compatible.

Key Verification

There are some more useful notary commands to list and even unsign images:

```
### verify image is signed
$ notary list dtr.example.com/admin/alpine -r targets/releases
$ notary list dtr.example.com/admin/alpine -r targets/admin

### unsign image
$ notary remove -p dtr.example.com/admin/alpine latest -r targets/releases
$ notary remove -p dtr.example.com/admin/alpine latest -r targets/admin

### verify image is no longer signed
$ notary list dtr.example.com/admin/alpine -r targets/releases
$ notary list dtr.example.com/admin/alpine -r targets/admin
```

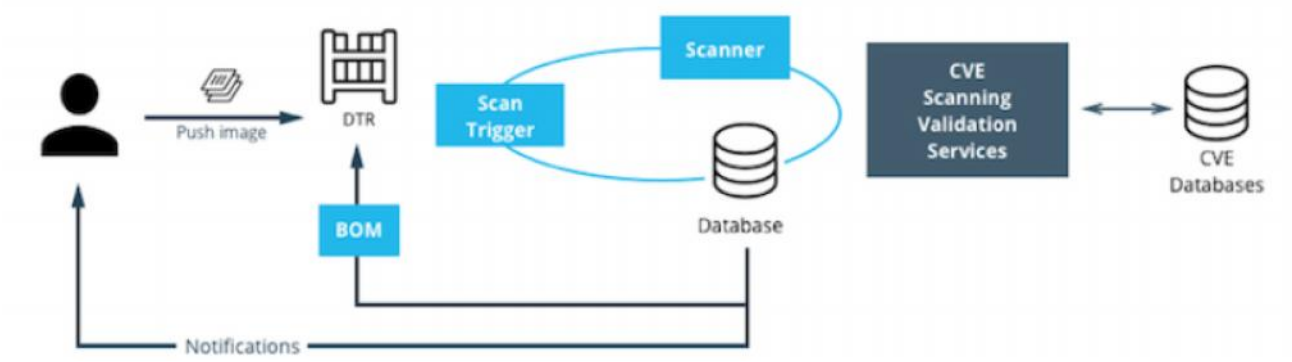
Image Scanning

Starting with version 2.2.0, DTR includes on-premises image scanning. The on-prem scanning engine within DTR scans images against the CVE Database (https://cve.mitre.org/). First, the scanner performs a binary scan on each layer of the image, identifies the software components in each layer, and indexes the SHA of each component. This binary scan evaluates the components on a bit-by-bit basis, so

vulnerable components are discovered regardless of filename, whether or not they're included on a distribution manifest or in a package manager, whether they are statically or dynamically linked, or even if they are from the base image OS distribution.

The scan then compares the SHA of each component against the CVE database (a "dictionary" of known information security vulnerabilities). When the CVE database is updated, the scanning service reviews the indexed components for any that match newly discovered vulnerabilities. Most scans complete within a few minutes, however larger repositories may take longer to scan depending on available system resources. The scanning engine provides a central point to scan all the images and delivers a Bill of Materials (BOM), which can be coupled with Notary (https://success.docker.com/api/asset/.%2Frefarch%2Fsecurity-best-practices%2F#contenttrustandimagesigningwithnotary) to ensure an extremely secure supply chain for the images.

As of DTR 2.3.0, the Scanning Engine now can scan Windows binaries.

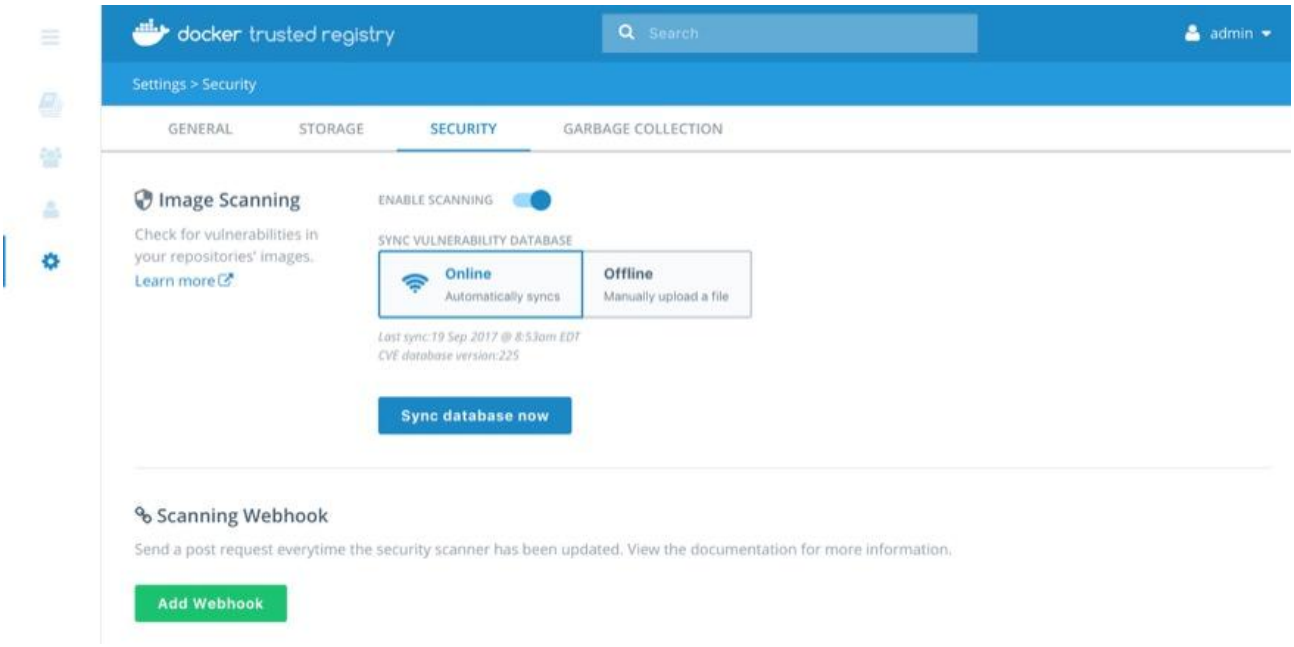


Setup Image Scanning

Before beginning, make sure the DTR license includes Docker Security Scanning and that the Docker ID being used can access and download this license from the Docker Store.

To enable Image Scanning, go to Settings -> Security, and select Enable Scanning. Then select whether to use the Docker-supplied CVE database (Online — the default option) or use a locally-uploaded file (Offline — this option is only recommended for environments that are isolated from the Internet or otherwise can't connect to Docker for consistent updates). Once enabled in online mode, DTR downloads the CVE database from Docker, which may take a while for the initial sync. If the installation cannot access `https://dss-cve-updates.docker.com/` manually upload a `.tar` file containing the security database.

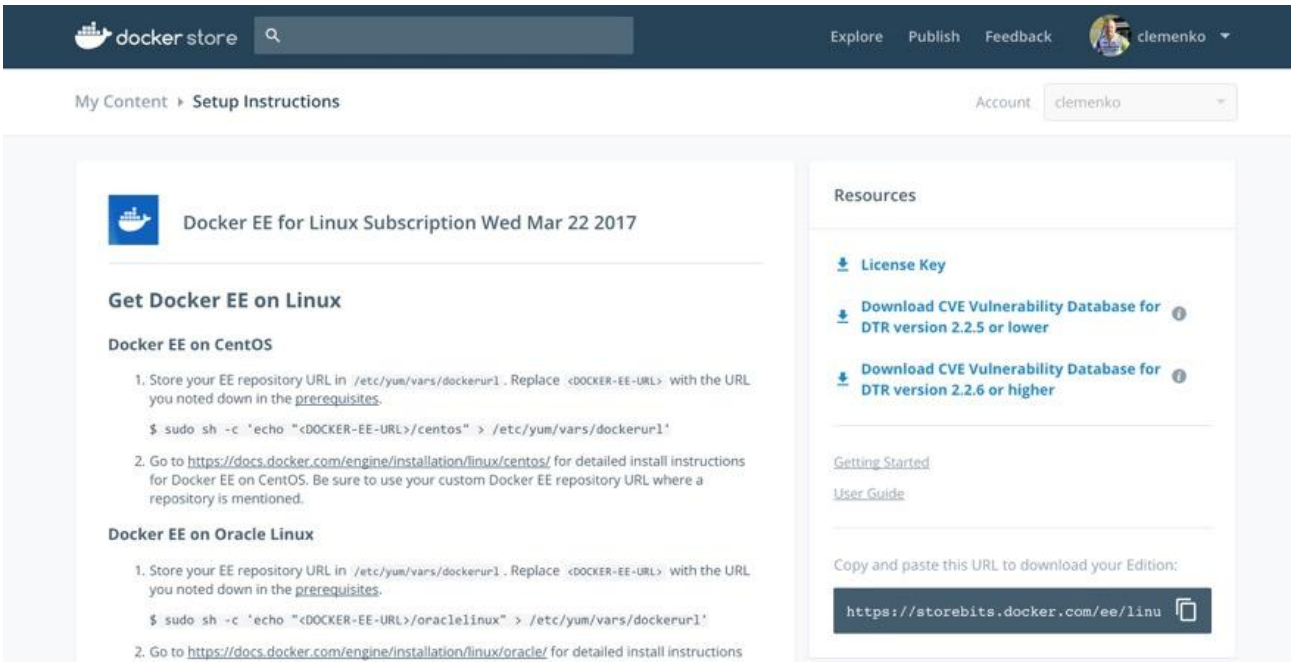
- If using Online mode, the DTR instance contacts a Docker server, download the latest vulnerability database, and install it. Scanning can begin once this process completes.
- If using Offline mode, use the instructions in Update scanning database - offline mode to upload an initial security database.



By default, when Security Scanning is enabled, new repositories automatically scan on `docker push`, but any repositories that existed before scanning was enabled are set to "scan manually" mode by default. If these repositories are still in use, this setting can be changed from each repository's Settings page.

CVE Offline Database

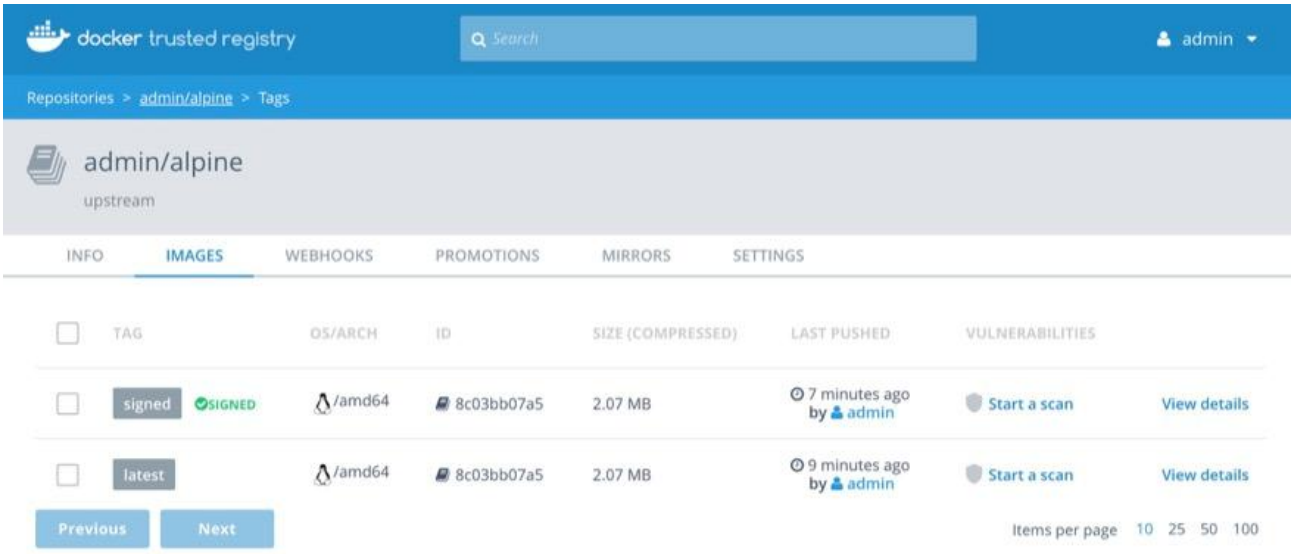
If the DTR instance cannot contact the update server, download and install a `.tar` file that contains the database updates. These offline CVE database files can be retrieved from Store.docker.com (https://store.docker.com) under My Content License Setup.



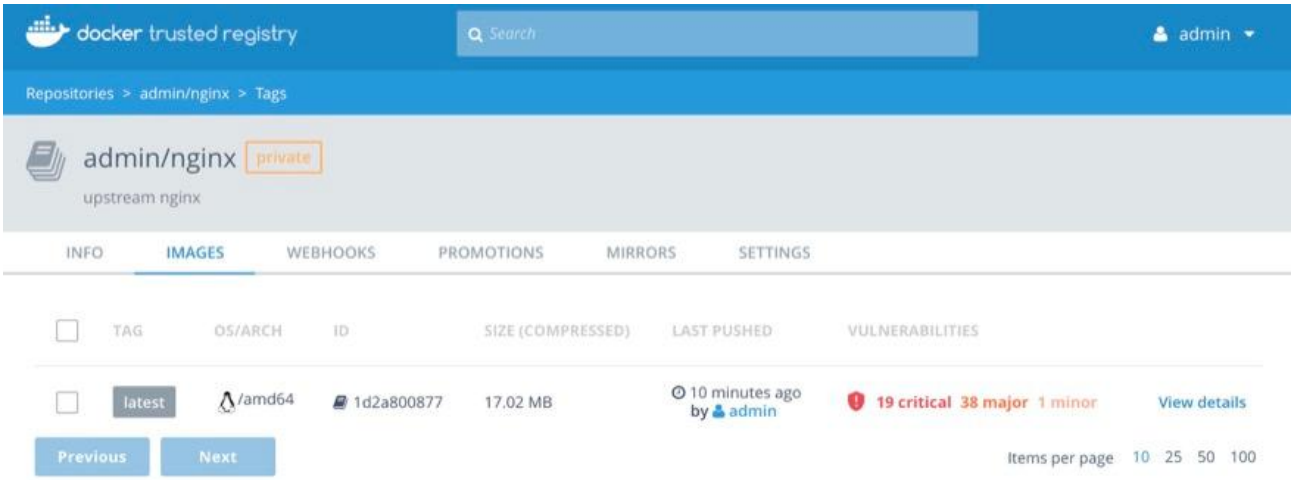
Scanning Results

To see the results of the scans, navigate to the repository itself, then click Images. A clean image scan

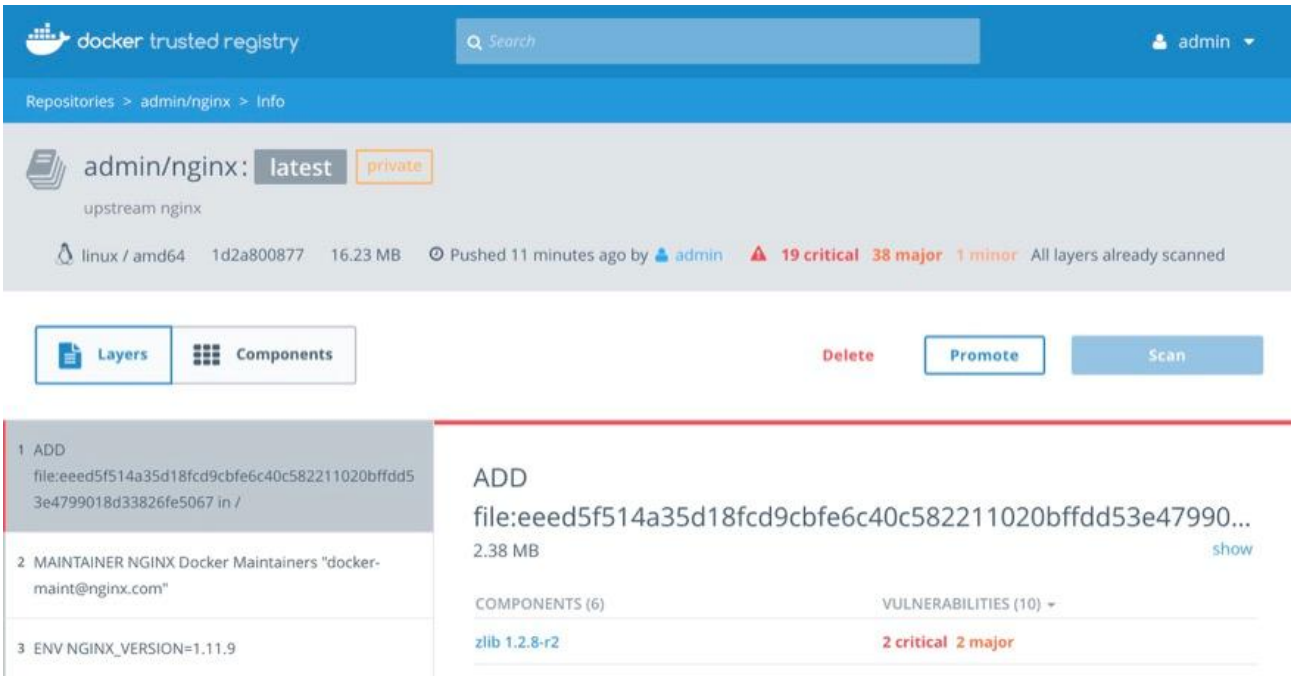
has a green checkmark shield icon:



A vulnerable image scan has a red warning shield:

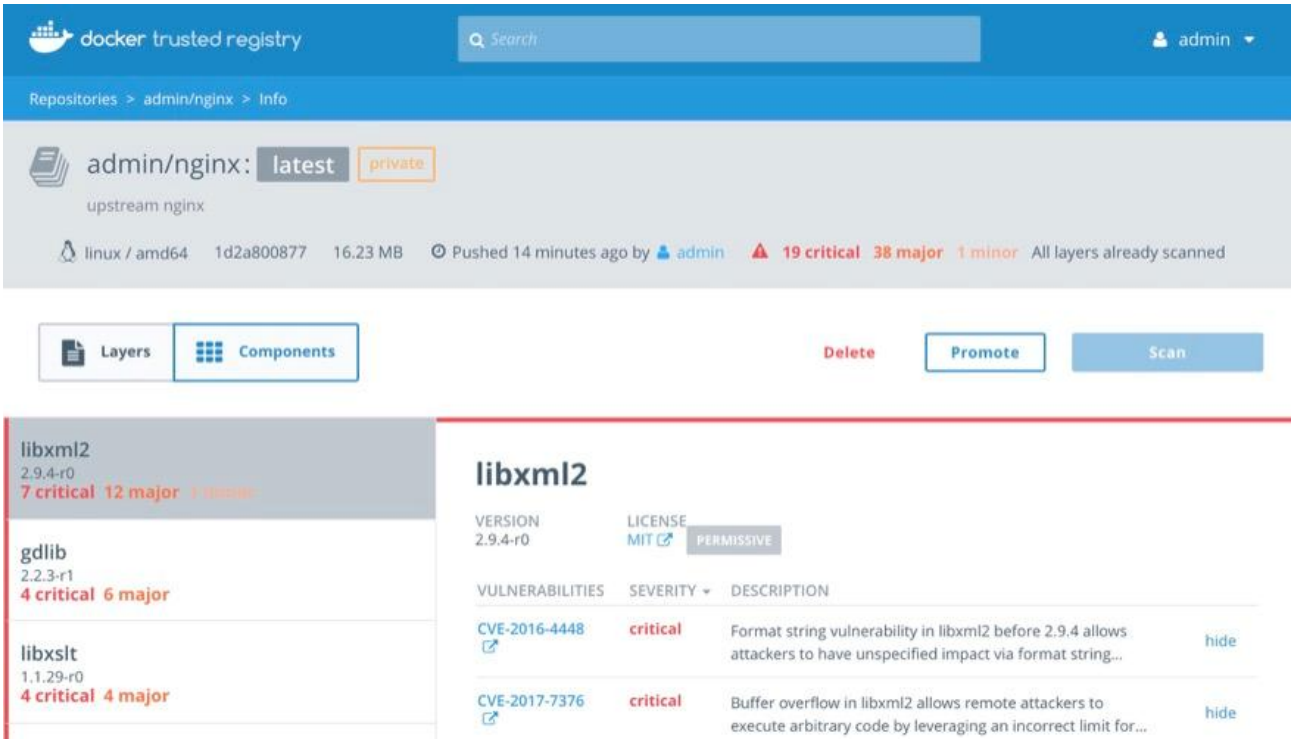


There are two views for the scanning resultsL Layers and Components. The Layers view shows which layer of the image had the vulnerable binary. This is extremely useful when diagnosing where the vulnerability is in the Dockerfile:



The vulnerable binary is displayed, along with all the other contents of the layer, when the layer itself is clicked on.

From the Components view, the CVE number, a link to CVE database, file path, layers affected, severity, and description of severity are available:



Now it is possible to take action against and vulnerable binary/layer/image.

If vulnerable components are discovered, check if there is an updated version available where the security vulnerability has been addressed. If necessary, contact the component's maintainers to ensure that the vulnerability is being addressed in a future version or patch update.

If the vulnerability is in a `base layer` (such as an operating system) it might not be possible to correct the issue in the image. In this case, switching to a different version of the base layer or finding an equivalent, less vulnerable base layer might help. Deciding that the vulnerability or exposure is acceptable is also an option.

Address vulnerabilities in the repositories by updating the images to use updated and corrected versions of vulnerable components, or by using different components that provide the same functionality. After updating the source code, run a build to create a new image, tag the image, and push the updated image to the DTR instance. Then re-scan the image to confirm that the vulnerabilities have been addressed.

What happens when there are new vulnerabilities released? There are actually two phases. The first phase is to fingerprint the image's binaries and layers into hashes. The second phase is to compare the hashes with the CVE database. The fingerprinting phase takes the longest amount of time to complete. Comparing the hashes is very quick. When there is a new CVE database, DTR simply compares the existing hashes with the new database. This process is also very quick. The scan results are always updated.

Webhooks

As of DTR 2.3.0 webhooks can be managed through the GUI. DTR includes webhooks for common events, such as pushing a new tag or deleting an image. This allows you to build complex CI and CD pipelines from your own DTR cluster.

The webhook events you can subscribe to are as follows (specific to a repository):

- Tag push
- Tag delete
- Manifest push
- Manifest delete
- Security scan completed

To subscribe to an event requires admin access to the particular repository. A global administrator can subscribe to any event. For example, a user must be an admin of repository to subscribe to its tag push events.

More information about webhooks can be found in the Docker docs (https://docs.docker.com/datacenter/dtr/2.3/guides/user/create-and-manage-webhooks/). DTR also presents the API by going to the menu under the login in the upper right, and then clicking API docs.

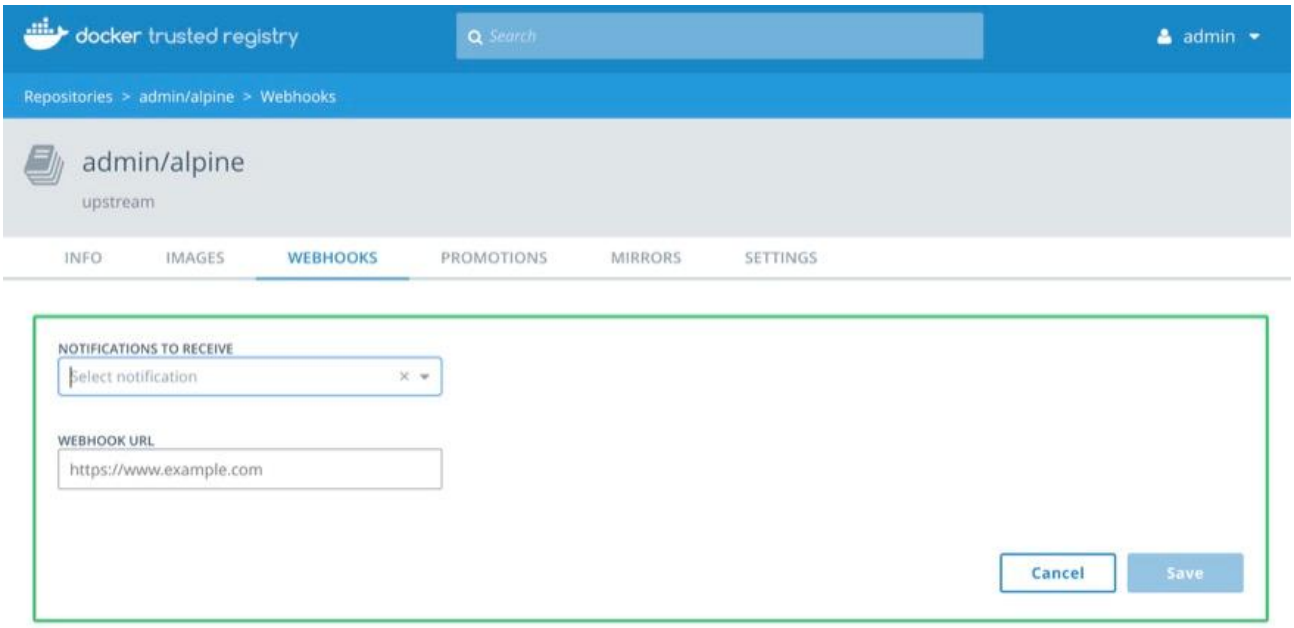


Image Immutability

As of DTR 2.3.0, there is an option to set a repository to Immutable. Setting a repository to Immutable means the tags can not be overwritten. This is a great feature for ensure the base images do not change over time. This next example is of the Alpine base image. Ideally CI would update the base image and push to DTR with a specific tag. Being Immutable simply guarantees that an authorized user can always go back to the specific tag and trust it has not changed. An Image Promotion Policy can extend on this.

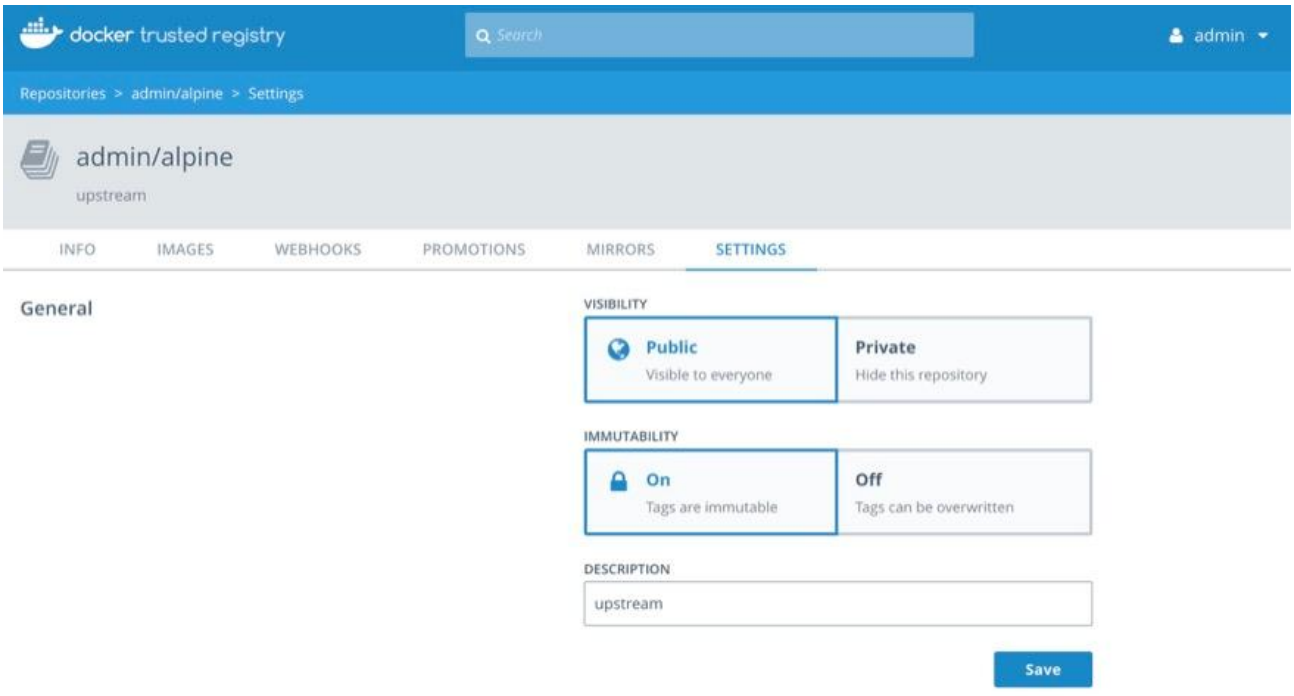
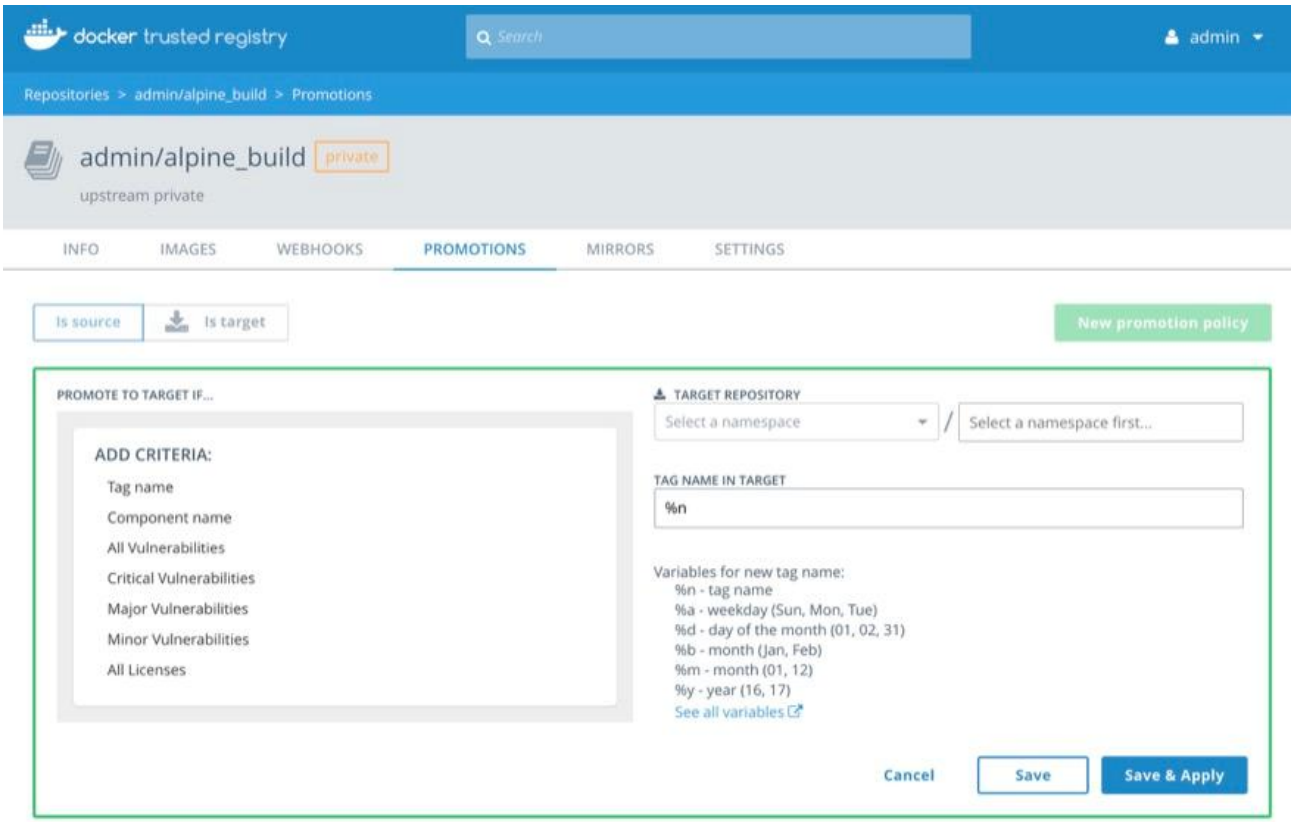


Image Promotion Policy

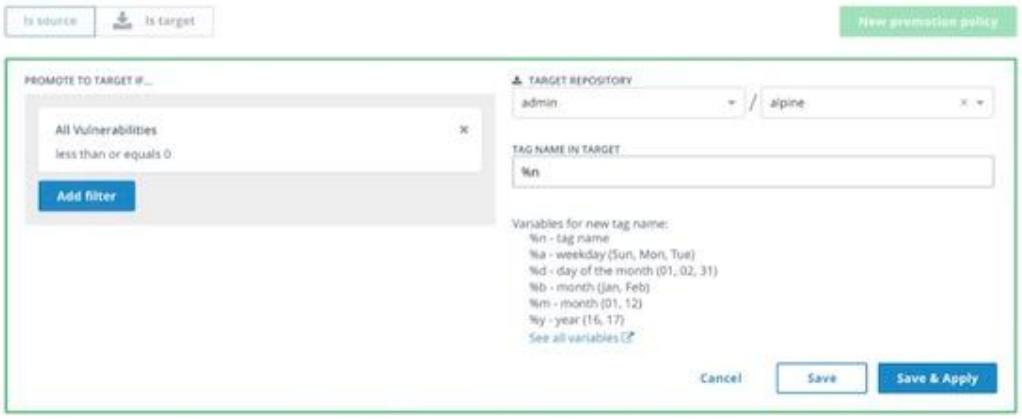
The release of Docker Trusted Registry 2.3.0 added a new way to promote images. Policies can be created for promotion based upon thresholds for vulnerabilities, tag matching, and package names, and even the license. This gives great powers in automating the flow of images. It also ensures that images that don't match the policy don't make it to production. The criteria are as follows:

- Tag Name
- Package Name
- All Vulnerabilities
- Critical Vulnerabilities
- Major Vulnerabilities
- Minor Vulnerabilities
- All Licenses

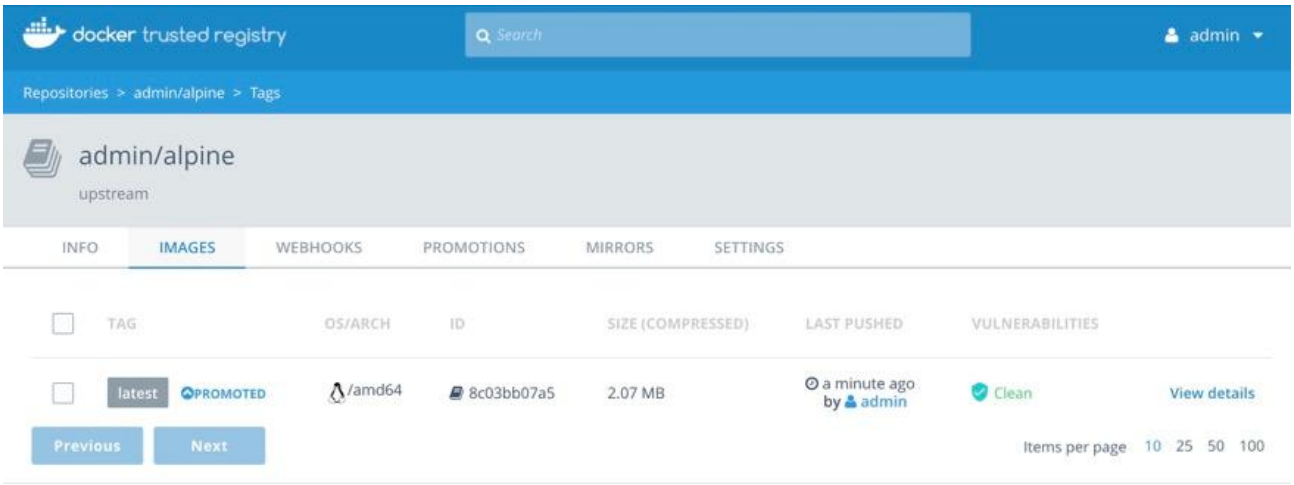


Policies can be created and viewed from either the source or the target. Consider the example of All Vulnerabilities to setup a promotion policy for the `admin/alpine_build` repo to "promote" to `admin/alpine` if there are zero vulnerabilities. Navigate to the source repository and go to the Policies tab. From there select New Promotion Policy. Select the All Vulnerabilities on the left. Then click less than or equals and enter `0` (zero) into the textbox,and click Add. Select a target for the promotion. On the right hand side select the namespace and image to be the target. Now click Save & Apply. Applying the policy will execute against the source repository. Save will apply the policy to future pushes.

Notice the Tag Name In Target that allows changes to the tag according to some variables. It is recommended to start with leaving the tag name the same. For more information please check out the Image Promotion Policy docs (https://docs.docker.com/datacenter/dtr/2.3/guides/user/create-promotion-policies/).



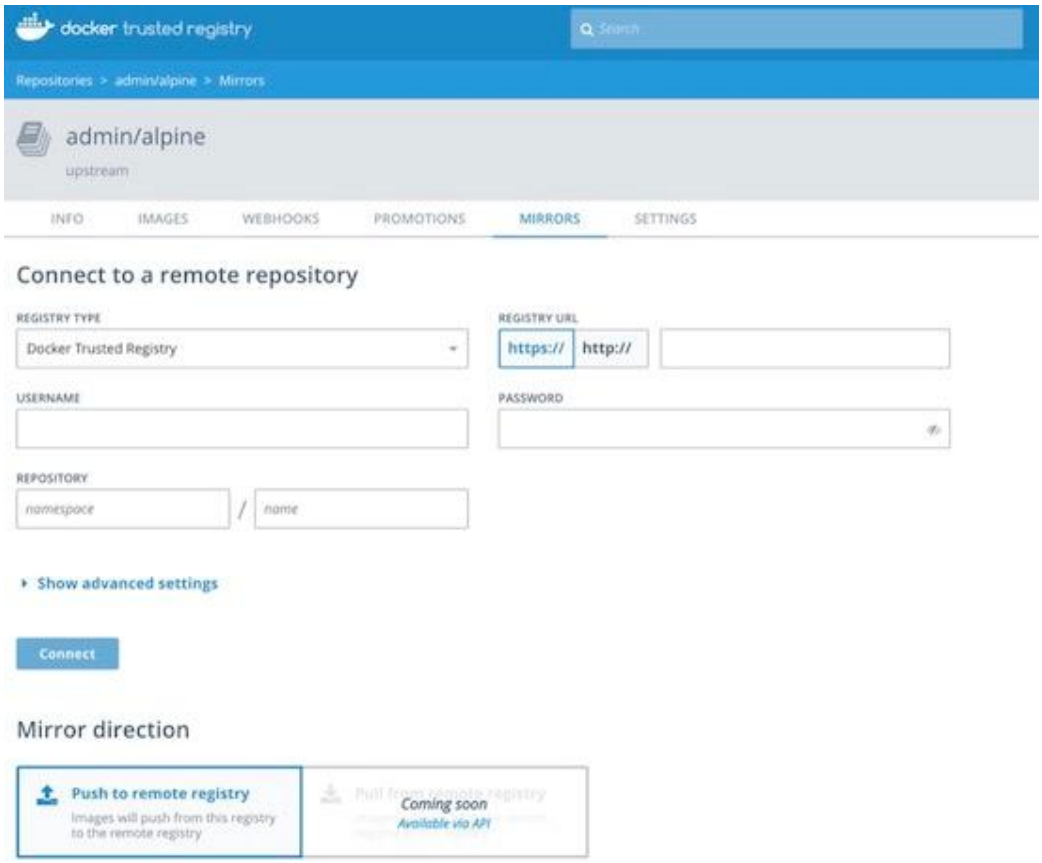
Notice the PROMOTED badge. One thing to note is that the Notary signature is not promoted with the image. This means a CI system will be needed to sign the promoted images. This can be achieved with the use of webhooks and promotion policy.



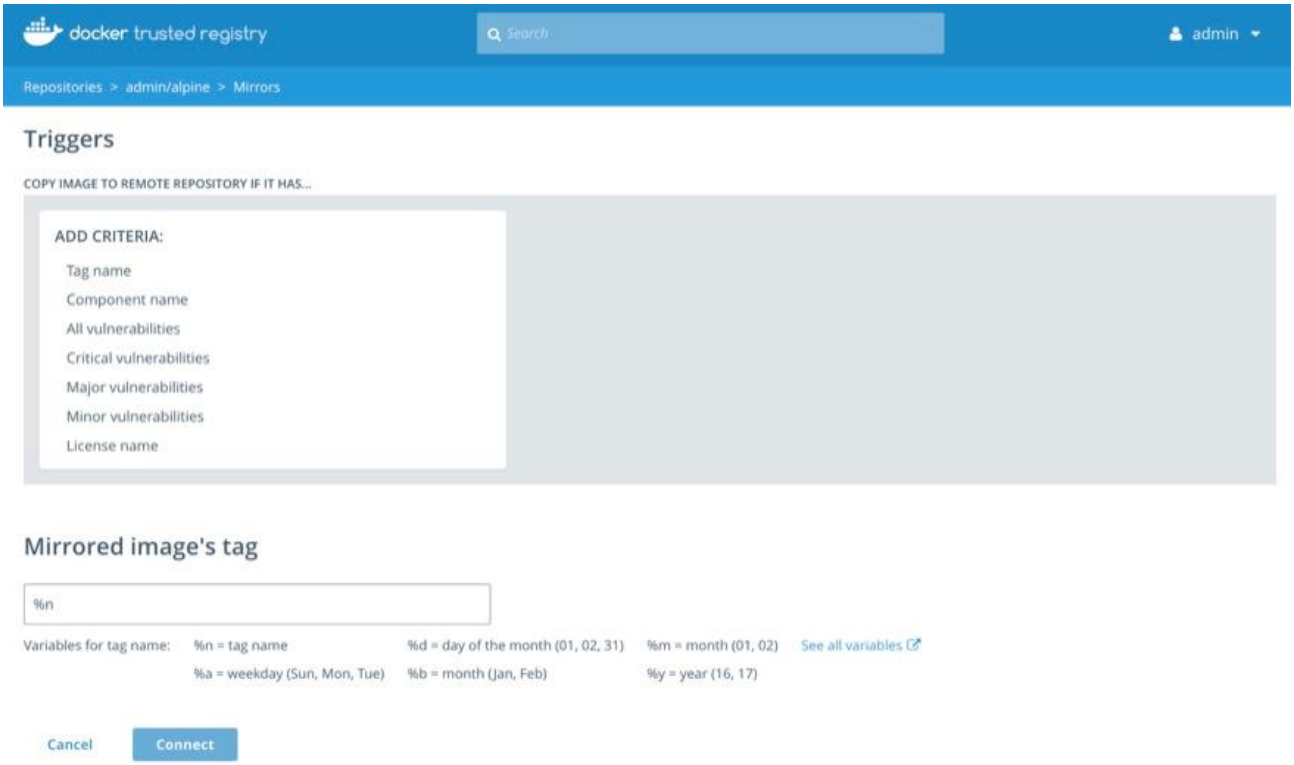
Imagine a DTR setup where the base images get pushed from Jenkins to DTR. Then the images get scanned and promoted if they have zero vulnerabilities. Sounds like a good part of a Secure Supply Chain.

Image Mirroring

NEW with Docker EE 2.0 DTR now adds Image Mirroring. Image Mirroring allows for images to be mirrored between DTR and another DTR. It also allows for mirroring between DTR and hub.docker.com (https://success.docker.com/api/asset/.%2Frefarch%2Fsecurity-best-practices%2Fhub.docker.com). Image Mirroring allows from increased control of your image pipeline.



One of the new features of Image Mirroring is the ability to PULL images from hub.docker.com (https://success.docker.com/api/asset/.%2Frefarch%2Fsecurity-best-practices%2Fhub.docker.com). Another great feature is the ability to trigger the PUSH mirroring to another DTR based on security scans or other criteria. Image Mirroring even has the capability to change the tag name. This is a good way to tag the image that it was pushed.



Summary

From limiting root access to nodes to using RBAC for UCP and DTR to storing secrets securely, this document provides all the security information needed to create a secure, customized, containerized infrastructure.

What is Docker (https://www.docker.com/what-docker)

What is a Container (https://www.docker.com/what-container)

Use Cases (https://www.docker.com/use-cases)

Customers (https://www.docker.com/customers)

For Government (https://www.docker.com/industry-government)

For IT Pros (https://www.docker.com/itpro)

Find a Partner (https://www.docker.com/find-partner)

Become a Partner (https://www.docker.com/partners/partner-program)

About Docker (https://www.docker.com/company)

Management (https://www.docker.com/company/management)

Press & News (https://www.docker.com/company/news-and-press)

Careers (https://www.docker.com/careers)

Product (https://www.docker.com/get-docker)

Pricing (https://www.docker.com/pricing)

Community Edition (https://www.docker.com/community-edition)

Enterprise Edition (https://www.docker.com/enterprise-edition)

Docker Datacenter (https://www.docker.com/enterprise-edition#container_management)

Docker Cloud (https://cloud.docker.com/)

Docker Store (https://store.docker.com/)

Get Docker (https://www.docker.com/get-docker)

Docker for Mac (https://www.docker.com/docker-mac)

Docker for Windows(PC) (https://www.docker.com/docker-windows)

Docker for AWS (https://www.docker.com/docker-aws)

Docker for Azure (https://www.docker.com/docker-azure)

Docker for Windows Server (https://www.docker.com/docker-windows-server)

Docker for Debian (https://www.docker.com/docker-debian)

Docker for Fedora® (https://www.docker.com/docker-fedora)

Docker for Oracle Linux (https://www.docker.com/docker-oracle-linux)

Docker for RHEL (https://www.docker.com/docker-red-hat-enterprise-linux-rhel)

Docker for SLES (https://www.docker.com/docker-suse-linux-enterprise-server-sles)

Docker for Ubuntu (https://www.docker.com/docker-ubuntu)

Documentation (https://docs.docker.com/)

Blog (https://blog.docker.com/)

RSS Feed (https://blog.docker.com/feed/)

Training (https://training.docker.com/)

Knowledge Base (https://success.docker.com/kbase)

Resources (https://www.docker.com/products/resources)

Community (https://www.docker.com/docker-community)

Open Source (https://www.docker.com/technologies/overview)

Events (https://www.docker.com/community/events)

Forums (https://forums.docker.com/)

An unexpected error has occurred.

Use overlay networks

Estimated reading time: 11 minutes

The `overlay` network driver creates a distributed network among multiple Docker daemon hosts. This network sits on top of (overlays) the host-specific networks, allowing containers connected to it (including swarm service containers) to communicate securely. Docker transparently handles routing of each packet to and from the correct Docker daemon host and the correct destination container.

When you initialize a swarm or join a Docker host to an existing swarm, two new networks are created on that Docker host:

- an overlay network called `ingress`, which handles control and data traffic related to swarm services. When you create a swarm service and do not connect it to a user-defined overlay network, it connects to the `ingress` network by default.
- a bridge network called `docker_gwbridge`, which connects the individual Docker daemon to the other daemons participating in the swarm.

You can create user-defined `overlay` networks using `docker network create`, in the same way that you can create user-defined `bridge` networks. Services or containers can be connected to more than one network at a time. Services or containers can only communicate across networks they are each connected to.

Although you can connect both swarm services and standalone containers to an overlay network, the default behaviors and configuration concerns are different. For that reason, the rest of this topic is divided into operations that apply to all overlay networks, those that apply to swarm service networks, and those that apply to overlay networks used by standalone containers.

Operations for all overlay networks

Create an overlay network

> Prerequisites:
>
> - Firewall rules for Docker daemons using overlay networks
>
> You need the following ports open to traffic to and from each Docker host participating on an overlay network:
>
> - TCP port 2377 for cluster management communications
> - TCP and UDP port 7946 for communication among nodes
> - UDP port 4789 for overlay network traffic
>
> - Before you can create an overlay network, you need to either initialize your Docker daemon as a swarm manager using `docker swarm init` or join it to an existing swarm using `docker swarm join`. Either of these creates the default `ingress` overlay network which is used by swarm services by default. You need to do this even if you never plan to use swarm services. Afterward, you can create additional user-defined overlay networks.

To create an overlay network for use with swarm services, use a command like the following:

```
$ docker network create -d overlay my-overlay
```

To create an overlay network which can be used by swarm services or standalone containers to communicate with other standalone containers running on other Docker daemons, add the `--attachable` flag:

```
$ docker network create -d overlay --attachable my-attachable-overla
y
```

You can specify the IP address range, subnet, gateway, and other options. See `docker network create --help` for details.

Encrypt traffic on an overlay network

All swarm service management traffic is encrypted by default, using the AES algorithm (https://en.wikipedia.org/wiki/Galois/Counter_Mode) in GCM mode. Manager nodes in the swarm rotate the key used to encrypt gossip data every 12 hours.

To encrypt application data as well, add `--opt encrypted` when creating the overlay network. This enables IPSEC encryption at the level of the vxlan. This encryption imposes a non-negligible performance penalty, so you should test this option before using it in production.

When you enable overlay encryption, Docker creates IPSEC tunnels between all the nodes where tasks are scheduled for services attached to the overlay network. These tunnels also use the AES algorithm in GCM mode and manager nodes automatically rotate the keys every 12 hours.

> Do not attach Windows nodes to encrypted overlay networks.
>
> Overlay network encryption is not supported on Windows. If a Windows node attempts to connect to an encrypted overlay network, no error is detected but the node cannot communicate.

SWARM MODE OVERLAY NETWORKS AND STANDALONE CONTAINERS

You can use the overlay network feature with both `--opt encrypted --attachable` and attach unmanaged containers to that network:

```
$ docker network create --opt encrypted --driver overlay --attachabl
e my-attachable-multi-host-network
```

Customize the default ingress network

Most users never need to configure the `ingress` network, but Docker 17.05 and higher allow you to do so. This can be useful if the automatically-chosen subnet conflicts with one that already exists on your network, or you need to customize other low-level network settings such as the MTU.

Customizing the `ingress` network involves removing and recreating it. This is usually done before you create any services in the swarm. If you have existing services which publish ports, those services need to be removed before you can remove the `ingress` network.

During the time that no `ingress` network exists, existing services which do not publish ports continue to function but are not load-balanced. This affects services which publish ports, such as a WordPress service which publishes port 80.

1. Inspect the `ingress` network using `docker network inspect ingress`, and remove any services whose containers are connected to it. These are services that publish ports, such as a WordPress service which publishes port 80. If all such services are not stopped, the next step fails.
2. Remove the existing `ingress` network:

```
$ docker network rm ingress

WARNING! Before removing the routing-mesh network, make sure al
l the nodes
in your swarm run the same docker engine version. Otherwise, re
moval may not
be effective and functionality of newly created ingress network
s will be
impaired.
Are you sure you want to continue? [y/N]
```

3. Create a new overlay network using the `--ingress` flag, along with the custom options you want to set. This example sets the MTU to 1200, sets the subnet to `10.11.0.0/16`, and sets the gateway to `10.11.0.2`.

```
$ docker network create \
  --driver overlay \
  --ingress \
  --subnet=10.11.0.0/16 \
  --gateway=10.11.0.2 \
  --opt com.docker.network.driver.mtu=1200 \
  my-ingress
```

> Note: You can name your `ingress` network something other than `ingress`, but you can only have one. An attempt to create a second one fails.

4. Restart the services that you stopped in the first step.

Customize the docker_gwbridge interface

The `docker_gwbridge` is a virtual bridge that connects the overlay networks (including the `ingress` network) to an individual Docker daemon's physical network. Docker creates it automatically when you initialize a swarm or join a Docker host to a swarm, but it is not a Docker device. It exists in the kernel of the Docker host. If you need to customize its settings, you must do so before joining the Docker host to the swarm, or after temporarily removing the host from the swarm.

1. Stop Docker.
2. Delete the existing `docker_gwbridge` interface.

```
$ sudo ip link set docker_gwbridge down

$ sudo ip link del dev docker_gwbridge
```

3. Start Docker. Do not join or initialize the swarm.
4. Create or re-create the `docker_gwbridge` bridge manually with your custom settings, using the `docker network create` command. This example uses the subnet `10.11.0.0/16`. For a full list of customizable options, see Bridge driver options (https://docs.docker.com/engine/reference/commandline/network_create/#bridge-driver-options).

```
$ docker network create \
--subnet 10.11.0.0/16 \
--opt com.docker.network.bridge.name=docker_gwbridge \
--opt com.docker.network.bridge.enable_icc=false \
--opt com.docker.network.bridge.enable_ip_masquerade=true \
docker_gwbridge
```

5. Initialize or join the swarm. Since the bridge already exists, Docker does not create it with automatic settings.

Operations for swarm services

Publish ports on an overlay network

Swarm services connected to the same overlay network effectively expose all ports to each other. For a port to be accessible outside of the service, that port must be *published* using the `-p` or `--publish` flag on `docker service create` or `docker service update`. Both the legacy colon-separated syntax and the newer comma-separated value syntax are supported. The longer syntax is preferred because it is somewhat self-documenting.

Flag value	Description
`-p 8080:80` or `-p published=8080,target=80`	Map TCP port 80 on the service to port 8080 on the routing mesh.
`-p 8080:80/udp` or `-p published=8080,target=80,protocol=udp`	Map UDP port 80 on the service to port 8080 on the routing mesh.

`-p 8080:80/tcp -p 8080:80/udp` or
`-p published=8080,target=80,protocol=tcp`
`-p published=8080,target=80,protocol=udp`
	Map TCP port 80 on the service to TCP port 8080 on the routing mesh, and map UDP port 80 on the service to UDP port 8080 on the routine mesh.

Bypass the routing mesh for a swarm service

By default, swarm services which publish ports do so using the routing mesh. When you connect to a published port on any swarm node (whether it is running a given service or not), you are redirected to a worker which is running that service, transparently. Effectively, Docker acts as a load balancer for your swarm services. Services using the routing mesh are running in *virtual IP (VIP) mode*.

Even a service running on each node (by means of the `--mode global` flag) uses the routing mesh. When using the routing mesh, there is no guarantee about which Docker node services client requests.

To bypass the routing mesh, you can start a service using *DNS Round Robin (DNSRR) mode*, by setting the `--endpoint-mode` flag to `dnsrr`. You must run your own load balancer in front of the service. A DNS query for the service name on the Docker host returns a list of IP addresses for the nodes running the service. Configure your load balancer to consume this list and balance the traffic across the nodes.

Separate control and data traffic

By default, control traffic relating to swarm management and traffic to and from your applications runs over the same network, though the swarm control traffic is encrypted. You can configure Docker to use separate network interfaces for handling the two different types of traffic. When you initialize or join the swarm, specify `--advertise-addr` and `--datapath-addr` separately. You must do this for each node joining the swarm.

Operations for standalone containers on overlay networks

Attach a standalone container to an overlay network

The `ingress` network is create without the `--attachable` flag, which means that only swarm services can use it, and not standalone containers. You can connect standalone containers to user-defined overlay networks which are created with the `--attachable` flag. This gives standalone containers running on different Docker daemons the ability to communicate without the need to set up routing on the individual Docker daemon hosts.

Publish ports

Flag value	Description
`-p 8080:80`	Map TCP port 80 in the container to port 8080 on the overlay network.

Flag value	Description
`-p 8080:80/udp`	Map UDP port 80 in the container to port 8080 on the overlay network.
`-p 8080:80/sctp`	Map SCTP port 80 in the container to port 8080 on the overlay network.
`-p 8080:80/tcp -p 8080:80/udp`	Map TCP port 80 in the container to TCP port 8080 on the overlay network, and map UDP port 80 in the container to UDP port 8080 on the overlay network.

Container discovery

For most situations, you should connect to the service name, which is load-balanced and handled by all containers ("tasks") backing the service. To get a list of all tasks backing the service, do a DNS lookup for `tasks.<service-name>.`

Next steps

- Go through the overlay networking tutorial (https://docs.docker.com/network/network-tutorial-overlay/)
- Learn about networking from the container's point of view (https://docs.docker.com/config/containers/container-networking/)
- Learn about standalone bridge networks (https://docs.docker.com/network/bridge/)
- Learn about Macvlan networks (https://docs.docker.com/network/macvlan/)

network (https://docs.docker.com/glossary/?term=network), overlay (https://docs.docker.com/glossary/?term=overlay), user-defined (https://docs.docker.com/glossary/?term=user-defined), swarm (https://docs.docker.com/glossary/?term=swarm), service (https://docs.docker.com/glossary/?term=service)

2 SEPTEMBER 2016 / DOCKER SWARM

Docker Swarm Firewall Ports

:s) ports. Below that, I also include the "Classic" Swarm ports from 1.11 and older. In each, the

Docker Swarm Mode Ports

Starting with 1.12 in July 2016, Docker *Swarm Mode* is a built-in solution with built-in key/value store. Easier to get started, and fewer ports to configure.

Inbound Traffic for Swarm Management

- TCP port 2377 for cluster management & raft sync communications
- TCP and UDP port 7946 for "control plane" gossip discovery communication between all nodes
- UDP port 4789 for "data plane" VXLAN overlay network traffic
- IP Protocol 50 (ESP) if you plan on using overlay network with the encryption option

AWS Security Group Example

AWS Tip: You should use Security Groups in AWS's "source" field rather then subnets, so SG's will all dynamically update when new nodes are added.

Inbound to Swarm Managers (superset of worker ports)

TYPE	PROTOCOL	PORTS	SOURCE
Custom TCP Rule	TCP	2377	swarm + remote mgmt
Custom TCP Rule	TCP	7946	swarm
Custom UDP Rule	UDP	7946	swarm
Custom UDP Rule	UDP	4789	swarm
Custom Protocol	50	all	swarm

Inbound to Swarm Workers

TYPE	PROTOCOL	PORTS	SOURCE
Custom TCP Rule	TCP	7946	swarm
Custom UDP Rule	UDP	7946	swarm
Custom UDP Rule	UDP	4789	swarm
Custom Protocol	50	all	swarm

Docker Swarm "Classic" Ports, with Consul

For Docker 1.11 and older. I Used this list from Docker Docs on Swarm Classic, then tested on multiple swarms.

Inbound to Swarm Nodes

- 2375 TCP for swarm manger -> nodes (LOCK PORT DOWN, no auth)

- 7946 TCP/UDP for container network discovery from other swarm nodes
- 4789 UDP container overlay network from other swarm nodes

Inbound to Swarm Managers

- 3375 TCP for spawner -> swarm manager (LOCK PORT DOWN, no auth)

Inbound to Consul

- 8500 TCP for swarm manager/nodes -> consul server (LOCK PORT DOWN, no auth)
- 8300 TCP for consul agent -> consul server
- 8301 TCP/UDP for consul agent -> consul agent
- 8302 TCP/UDP for consul server -> consul server

Swarm Classic Inbound Ports In AWS Security Group Format, with Consul

AWS Tip: You should use Security Groups in AWS's "source" field rather then subnets, so SG's will all dynamically update when new nodes are added.

This is another way to look at the above lists, in a format that makes sense for AWS SG's

- assume AWS inbound from:
  - Internet ELB -> Swarm Managers
  - Swarm Managers -> Swarm Nodes
  - Swarm Managers -> Consul Internal ELB
  - Swarm Nodes -> Consul Internal ELB
  - Consul Internal ELB -> Consul Nodes

ELB Swarm Manager

TYPE	PROTOCOL	PORTS	SOURCE
Custom TCP Rule	TCP	3375	spawners

Swarm Managers

TYPE	PROTOCOL	PORTS	SOURCE
Custom TCP Rule	TCP	3375	elb-swarm-manager

Swarm Nodes

TYPE	PROTOCOL	PORTS	SOURCE
Custom TCP Rule	TCP	2375	swarm-managers
Custom TCP Rule	TCP	7946	swarm-nodes
Custom UDP Rule	UDP	7946	swarm-nodes
Custom UDP Rule	UDP	4789	swarm-nodes

ELB Consul

TYPE	PROTOCOL	PORTS	SOURCE
Custom TCP Rule	TCP	8500	swarm-nodes
Custom TCP Rule	TCP	8500	swarm-managers

Consul Nodes

TYPE	PROTOCOL	PORTS	SOURCE
Custom TCP Rule	TCP	8500	elb-consul

TYPE	PROTOCOL	PORTS	SOURCE
Custom TCP Rule	TCP	8300-8302	consul-nodes
Custom UDP Rule	UDP	8301-8302	consul-nodes

Bret Fisher
Author of Docker Mastery courses on Udemy, freelance DevOps and Docker consultant, trainer, speaker, and Open Source volunteer working from Virginia Beach, Virginia.

Read More

TALKS AND WORKSHOPS

Strange Loop 2016: Docker Orchestration Workshop

Resources from Workshop at Strange Loop 2016 Slides - Current Master Slide Deck from Jérôme Petazzoni - (Docker OG) Optional Workshop Repo (for feedback/PR's) Twitter List of Docker Captains

CHEAT SHEETS

CLI Tips and Tricks (From Docker Online Meetup)

Along with other fantastic Docker Captains Ajeet Singh Raina and Viktor Farcic, I talked for a few minutes on a Docker Online Meetup (hosted by Docker Inc.). You can watch my 10 minutes

1 MIN READ

1 MIN READ

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
```

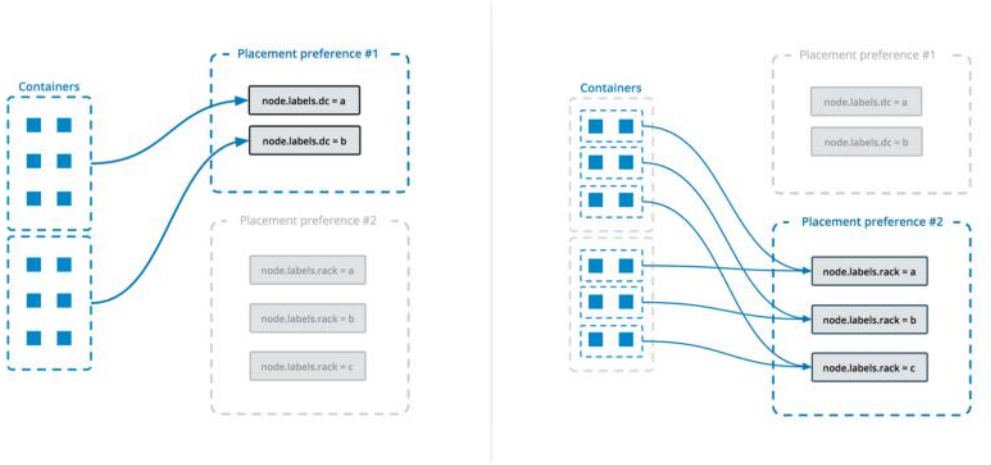
> **Missing or null labels**
>
> Nodes which are missing the label used to spread still receive task assignments. As a group, these nodes receive tasks in equal proportion to any of the other groups identified by a specific label value. In a sense, a missing label is the same as having the label with 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)