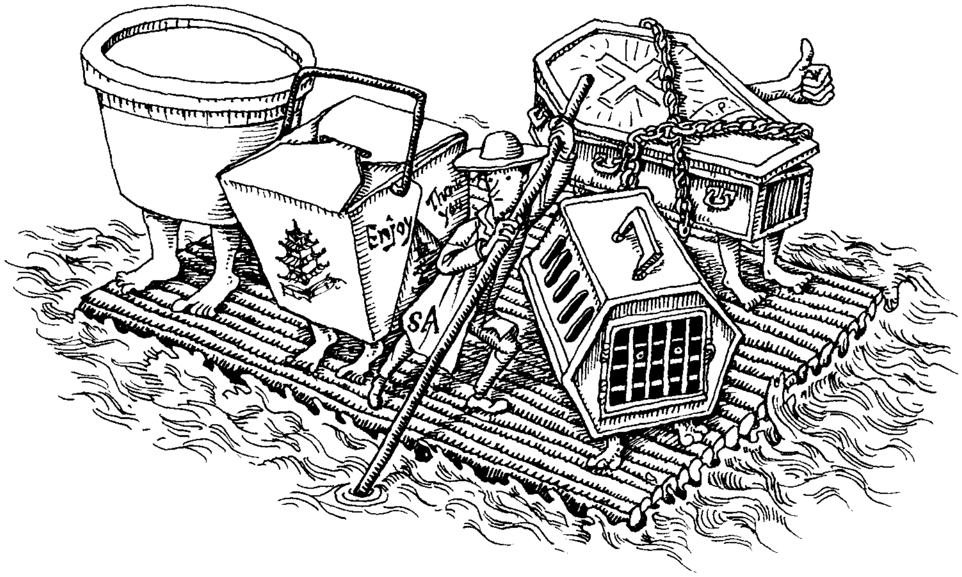
25 Containers



Few technologies have generated as much excitement and hype in recent years as the humble container, whose explosion in popularity coincided with the release of the open source Docker project in 2013. Containers are of particular interest to system administrators because they standardize software packaging, an ambition that has long been tantalizingly out of reach.

To illustrate the utility of containers, consider a typical web application developed in any modern language or framework. At a minimum, the following ingredients are needed to install and run the app:

- The code for the application and its correct configuration
- Libraries and other dependencies, potentially numbered in the dozens, each pinned to a specific version that is known to be compatible
- An interpreter (e.g., Python or Ruby) or run time (JRE) to execute the code, also version pinned
- Localizations such as user accounts, environment settings, and services provided by the operating system

A typical site runs dozens or hundreds of such applications. Maintaining uniformity in each of these areas across multiple application deployments is a constant challenge, even with the assistance of the tools discussed in Chapter 23, *Configuration Management*, and Chapter 26, *Continuous Integration and Delivery*. Incompatible dependencies required by separate applications lead to systems that are underutilized because they cannot be shared. In addition, at sites where software developers and system administrators are functionally separated, careful coordination is needed because it's not always straightforward to identify who's responsible for what parts of the operating environment.

A container image simplifies matters by packaging an application and its prerequisites into a standard, portable file. Any host with a compatible container run-time engine can create a container by using the image as a template. Tens or hundreds of containers can run simultaneously without conflicts. With images typically being a few hundred megabytes in size or less, it's practical to copy them among systems. This easy application portability is perhaps the primary reason for the popularity of containers.

This chapter focuses on Docker. The eponymous business behind Docker has played a central role in bringing containers into mainstream use, and the Docker ecosystem is the one you're most likely to encounter as a system administrator. Docker, Inc., offers several products related to containers, but we limit our discussion to the main container engine and the Swarm cluster manager.

Several viable alternative container engines are available. rkt, from CoreOS, is the most complete. It has a cleaner process model than Docker and a more secure default configuration. rkt integrates well with the Kubernetes orchestration system. **systemd-nspawn**, from the **systemd** project, is another option for lightweight containers. It has fewer features than Docker or rkt, but in some cases that can be a good thing. rkt cooperates with **systemd-nspawn** to configure container namespaces.

25.1 Background and core concepts

The container's rapid rise to grace can be attributed more to timing than to the emergence of any single technology. Containers are a fusion of numerous existing kernel features, filesystem tricks, and networking hacks. A container engine is the management software that pulls it all together.

In essence, a container is an isolated group of processes that are restricted to a private root filesystem and process namespace. The contained processes share the kernel and other services of the host OS, but by default they cannot access files or system resources outside their container. Applications that run within a container are not aware of their containerized state and do not require modification.

After you read the following sections, it should be clear that containers contain no magic. In fact, they rely on some features of UNIX and Linux that have been around for many years. See Chapter 24, *Virtualization*, for a description of how containers differ from virtual machines.

Kernel support

The container engine uses several kernel features that are essential for isolating processes. In particular:

- *Namespaces* isolate containerized processes from the perspective of several operating system facilities, including filesystem mounts, process management, and networking. The mount namespace, for example, shows processes a customized view of the filesystem hierarchy.[1] Containers can run with varying levels of integration with the host operating system, depending on how these namespaces have been configured.
- *Control groups* (contextually abbreviated to cgroups) limit the use of system resources and prioritize certain processes over others. Cgroups prevent runaway containers from consuming all available CPU and memory.
- *Capabilities* allow processes to execute certain sensitive kernel operations and system calls. For example, a process might have a capability that permits it to change the ownership of a file or to set the system time.
- *Secure computing mode* (usually shortened to seccomp) restricts access to system calls. It allows more fine-grained control than do capabilities.

Development of these features was driven in part by the Linux Containers project, LXC, which began at Google in 2006. LXC was the basis of Borg, Google's internal virtualization platform. LXC supplies the raw functions and tools needed to create and run Linux containers, but with more than 30 command-line tools and configuration files, it's quite complicated. The first few releases of Docker were essentially user-friendly wrappers that made LXC easier to use.

Docker now relies on an improved, standards-based container run time dubbed **containerd**. It too relies on Linux namespaces, cgroups, and capabilities to isolate containers. Learn more at containerd.io.

Images

A container image is akin to a template for a container. Images rely on union filesystem mounts for performance and portability. Unions overlay multiple filesystems to create a single, consistent hierarchy.[2] Container images are union filesystems that are organized to resemble the root filesystem of a typical Linux distribution. The directory layout and the locations of binaries, libraries, and supporting files all

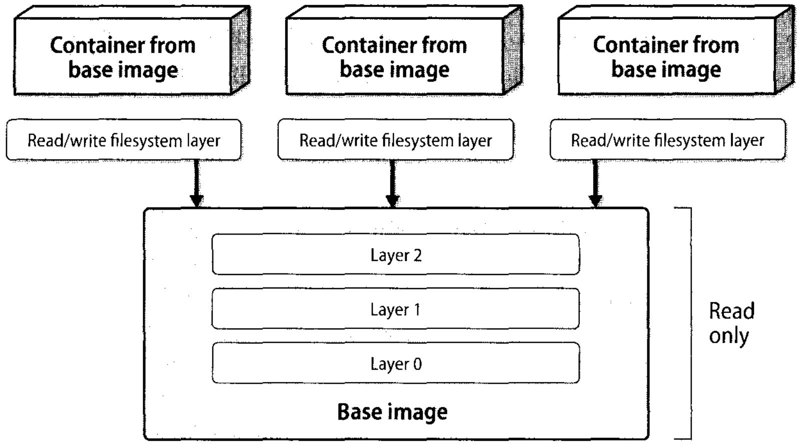
1. This is similar in principle to the **chroot** system call, which irreversibly sets a process's apparent root directory and thereby disables access to files and directories above the level of the **chroot**.
2. The LWN.net article "A brief history of union mounts" describes the relevant background. The related articles are interesting reading, too. See lwn.net/Articles/396020.

conform to standard Linux filesystem hierarchy specifications. Specialized Linux distributions have been developed for use as the basis of container images.

To create a container, Docker points to the read-only union filesystem of an image and adds a read/write layer that the container can update. When containerized processes modify the filesystem, their changes are transparently saved within the read/write layer. The base remains unmodified. This is known as a copy-on-write strategy.

Many containers can share the same immutable base layers, thus improving storage efficiency and reducing startup times. Exhibit A depicts the scheme.

Exhibit A Docker images and the union filesystem


Container from base image
Container from base image
Container from base image
Read/write filesystem layer
Read/write filesystem layer
Read/write filesystem layer
Layer 2
Layer 1
Layer 0
Base image
Read only

Networking

The default way to connect containers to the network is to use a network namespace and a bridge within the host. In this configuration, containers have private IP addresses that aren't reachable from outside the host. The host acts as a poor man's IP router and proxies traffic between the outside world and the containers. This architecture gives administrators control over which container ports are exposed to the outside world.

It's also possible to forgo the private container addressing scheme and expose entire containers directly to the network. This is called host mode networking, and it means that the container has unfettered access to the host's network stack. This might be desirable in some situations, but it also presents a security risk because the container is not fully isolated.

See *Docker networks* on page 927 for more details.

25.2 Docker: the open source container engine

Docker, Inc.'s primary product is a client/server application that builds and manages containers. The Docker container engine, written in Go, is highly modular. Separate, individual projects manage pluggable storage, networking, and other features.

Docker, Inc., is not without controversy. Its tools tend to evolve rapidly, and new versions have sometimes been incompatible with existing deployments. Some sites worry that relying on Docker's ecosystem will result in vendor lock-in. And as with any new technology, containers introduce complexity and require some study to understand.

To counter these sources of resistance, Docker, Inc., became one of the founding members of the Open Container Initiative, a consortium whose mission is to guide the growth of container technology in a healthily competitive direction that fosters standards and collaboration. You can learn more at opencontainers.org. In 2017, Docker founded the Moby project and contributed the primary Docker Git repository to it to facilitate easier community development of the Docker execution engine. Refer to mobyproject.org for details.

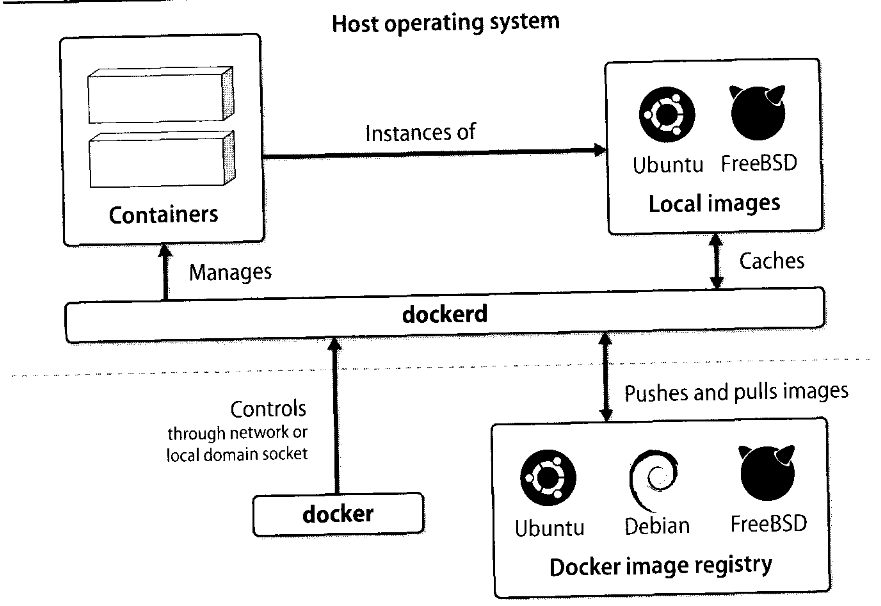
Our discussion of Docker is based on version 1.13.1. Docker maintains an exceptionally rapid pace of development, and the current features are a moving target. We focus on the nuts and bolts here, but be sure to supplement our tutorial with the reference material at docs.docker.com. You might also dip your toes into the Moby sandbox at play-with-moby.com and the Docker lab environment at labs.play-with-docker.com.

Basic architecture

docker is an executable command that handles all management tasks for the Docker system. **dockerd** is the persistent daemon process that implements container and image operations. **docker** can run on the same system as **dockerd** and can communicate with it through UNIX domain sockets, or it can contact **dockerd** from a remote host over TCP. The architecture is depicted in Exhibit B on the next page.

dockerd owns all the scaffolding needed to run containers. It creates the virtual network plumbing and maintains the data directory in which containers and images are stored (**/var/lib/docker** by default). It's responsible for creating containers by invoking the appropriate system calls, setting up union filesystems, and executing processes. In short, it is the container management software.

Exhibit B Docker architecture



Administrators interface with **dockerd** from the command line by running subcommands of the **docker** client. You can create a container with **docker run**, for example, or view information about the server with **docker info**. Table 25.1 summarizes some frequently used subcommands.

An image is the *template* for a container. It includes the files that processes running within the container instance depend on, such as libraries, operating system binaries, and applications. Linux distributions can function as convenient base images because they define a complete operating environment. However, an image is not necessarily based on a Linux distribution. The "scratch" image is an explicitly empty image intended as a basis for creation of other, more practical images.

A container relies on the image template as a basis for execution. When **dockerd** runs a container, it creates a writable filesystem layer that is separate from the source image. The container can read any of the files and other metadata stored within the image, but any writes are confined to the container's own read/write layer.

An image registry is a centralized collection of images. **dockerd** communicates with registries when you **docker pull** an image that isn't already present or when you **docker push** one of your own images. The default registry is Docker Hub, which stockpiles images for many popular applications. Most standard Linux distributions also publish Docker images.

Table 25.1 Frequently used docker subcommands

Subcommand	What it does
docker info	Displays summary information about the daemon
docker ps	Displays running containers
docker version	Displays extensive version info about the server and client
docker rm	Removes a container
docker rmi	Removes an image
docker images	Displays local images
docker inspect	Displays the configuration of a container (JSON output)
docker logs	Displays the standard output from a container
docker exec	Executes a command in an existing container
docker run	Runs a new container
docker pull/push	Downloads images from or uploads images to a remote registry
docker start/stop	Starts or stops an existing container
docker top	Displays containerized process status

You can run your own registry, or you can add your custom images to private registries that are hosted on Docker Hub. Any system with Docker can pull images from a registry as long as the registry server is accessible over the network.

Installation

Docker runs on Linux, macOS, Windows, and FreeBSD, but Linux is the flagship platform. FreeBSD support is considered experimental. Visit docker.com to choose the installation method that best suits your environment.

Users in the docker group can control the Docker daemon through its socket, which effectively gives those users root privileges. This is a significant security risk, so we suggest that you use **sudo** to control access to **docker** rather than adding users to the docker group. In the examples below, we run **docker** commands as the root user.

The installation process may not immediately start the daemon. If it isn't running, start it through the system's normal **init** system. On CentOS, for example, run **sudo systemctl start docker**.

Client setup

If you're connecting to a local **dockerd** and you're in the docker group or have **sudo** privileges, no client configuration is necessary. The **docker** client connects to **dockerd** through a local socket by default. You can modify the default client behavior by setting environment variables.

To connect to a remote **dockerd**, set the DOCKER_HOST environment variable. The usual HTTP port for the daemon is 2375, and the TLS version is 2376.

For example:

```
$ export DOCKER_HOST=tcp://10.0.0.10:2376
```

Always use TLS to communicate with remote daemons. If you use plain HTTP, you may as well hand out root privileges freely to anyone on your network. You can find additional details on Docker TLS configuration in *Use TLS* starting on page 940.

We also suggest enabling the content trust:

```
$ export DOCKER_CONTENT_TRUST=1
```

This feature validates the integrity and publisher of Docker images. Enabling the content trust prevents the client from pulling images that are not trusted.

If you run **docker** through **sudo**, you can prevent **sudo** from purging your environment variables with the **-E** flag. You can also whitelist specific environment variables by setting the value of the env_keep variable in **/etc/sudoers**. For example,

```
Defaults env_keep += "DOCKER_CONTENT_TRUST"
```

The container experience

To create a container, you need an image to use as a template. The image has all the filesystem bits needed to run programs. A new installation of Docker has no images. To download images from the Docker Hub, use **docker pull**.[3]

```
# docker pull debian
Using default tag: latest
latest: Pulling from library/debian
f50f9524513f: Download complete
d8bd0657b25f: Download complete
Digest: sha256:e7d38b3517548a1c71e41bffe9c8ae6d6...
Status: Downloaded newer image for debian:latest
```

The hex strings are the layers of the union filesystem. If the same layer is used by more than one image, Docker needs only a single copy. We didn't request a specific tag, or version, of the Debian image, so Docker downloaded the "latest" tag by default.

Examine the locally available images with **docker images**:

```
# docker images
REPOSITORY    TAG        IMAGE ID        CREATED        SIZE
ubuntu        latest     07c86167cdc4    2 weeks ago    187.9 MB
ubuntu        wily       b5e09e0cd052    5 days ago     136.1 MB
ubuntu        trusty     97434d46f197    5 days ago     187.9 MB
ubuntu        15.04      d1b55fd07600    8 weeks ago    131.3 MB
centos        7          d0e7f81ca65c    2 weeks ago    196.6 MB
centos        latest     d0e7f81ca65c    2 weeks ago    196.6 MB
debian        jessie     f50f9524513f    3 weeks ago    125.1 MB
debian        latest     f50f9524513f    3 weeks ago    125.1 MB
```

3. You can review the available images by browsing hub.docker.com.

This machine has the images for several Linux distributions, including the just-downloaded Debian image. The same image can be tagged more than once. Notice that debian:jessie and debian:latest share an image ID; they are two different names for the same image.

Armed with an image, it's remarkably simple to run a basic container:

```
# docker run debian /bin/echo "Hello World"
Hello World
```

What just happened? Docker created a container from the Debian base image and ran the command **/bin/echo "Hello World"** inside it.[4] The container stops running when the command exits: in this case, immediately after **echo** completes. If the "debian" image didn't already exist locally, the daemon would attempt to automatically download it before running the command. We didn't specify a tag, so the "latest" image was used by default.

We start an interactive shell with the **-i** and **-t** flags to **docker run.** The command below starts a **bash** shell within the container and connects the "outer" shell's I/O channels to it. We also assign the container a hostname, which is helpful for identifying it in logs. (Otherwise, we'd see the container's random ID in log messages.)

```
ben@host$ sudo docker run --hostname debian -it debian /bin/bash
root@debian:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@debian:/# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY    STAT START  TIME COMMAND
root         1  0.5  0.4  20236  1884 ?       Ss  19:02  0:00 /bin/bash
root         7  0.0  0.2  17492  1144 ?       R+  19:02  0:00 ps aux
root@debian:/# uname -r
3.10.0-327.10.1.el7.x86_64
root@debian:/# exit
exit
ben@host$ uname -r
3.10.0-327.10.1.el7.x86_64
```

The experience is oddly similar to accessing a virtual machine. There is a complete root filesystem, but the process tree appears nearly empty. **/bin/bash** is PID 1 because it's the command that Docker started in the container.

The result of **uname -r** is the same both inside and outside the container. That will always be the case; we show it as a reminder that the kernel is shared.

Processes in containers cannot see other processes running on the system because of PID namespacing. However, processes on the host can see the containerized processes. The PID of a process as seen from within a container differs from the PID that is visible from the host.

4. This is GNU **echo**, not to be confused with the **echo** command built into most shells. They do exactly the same thing.

For real work, you need long-lived containers that run in the background and accept connections over the network. The following command runs in the background (**-d**) a container named "nginx" that's generated from the official NGINX image. We tunnel port 80 from the host into the same port within the container:

```
# docker run -p 80:80 --hostname nginx --name nginx -d nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
fdd5d7827f33: Already exists
a3ed95caeb02: Pull complete
e04488adab39: Pull complete
2af76486f8b8: Pull complete
Digest: sha256:a234ab64f6893b9a13811f2c81b46cfac885cb141dcf4e275ed3
    ca18492ab4e4
Status: Downloaded newer image for nginx:latest
0cc36b0e61b5a8211432acf198c39f7b1df864a8132a2e696df55ed927d42c1d
```

We didn't have the "nginx" image locally, so Docker had to pull it from the registry. Once the image was downloaded, Docker started the container and printed its ID, a unique 65-character hexadecimal string.

docker ps shows a brief summary of running containers:

```
# docker ps
IMAGE  COMMAND                   STATUS        PORTS
nginx  "nginx -g 'daemon off"   Up 2 minutes  0.0.0.0:80->80/tcp
```

We didn't tell **docker** what to run in the container, so it used the default command that was specified when the image was created. The output shows this command to be **nginx -g 'daemon off'** which runs **nginx** as a foreground process rather than as a background daemon. The container has no **init** to manage processes, and if the **nginx** server were started as a daemon, the container would run but immediately exit when the **nginx** process forked and exited to enter the background.

Most server daemons offer a command-line flag that forces them to run in the foreground. If your software doesn't run in the foreground or if you need to run several processes in a container, you can assign a process control system such as **supervisord** to act as a lightweight **init** for the container.

With NGINX running in the container and port 80 mapped from the host, we can make HTTP requests to the container with **curl**. NGINX serves a generic HTML landing page by default.

```
host$ curl localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

We can use **docker logs** to view the STDOUT from the container, which in this case is the NGINX access log. The only traffic is our **curl** request:

```
# docker logs nginx
172.17.0.1 - - [24/Feb/2017:19:12:24 +0000] "GET / HTTP/1.1" 200 612
    "-" "curl/7.29.0" "-"
```

We can also use **docker logs -f** to get a real-time stream of a container's output, just like running **tail -f** on a growing log file.

docker exec creates a new process in an existing container. For example, to debug or troubleshoot, we could start an interactive shell in a container:

```
# docker exec -ti nginx bash
root@nginx:/# apt-get update && apt-get -y install procps
root@nginx:/# ps ax
  PID TTY     STAT   TIME  COMMAND
    1 ?       Ss     0:00  nginx: master process nginx -g daemon off;
    7 ?       S      0:00  nginx: worker process
    8 ?       Ss     0:00  bash
   21 ?       R+     0:00  ps ax
```

Container images are as slim as possible and are often missing common administrative utilities. In this sequence, we first updated the package index and then installed **ps**, which is part of the procps package.

The process list reveals the **nginx** master daemon, an **nginx** worker, and our **bash** shell. When we exit the shell created with **docker exec**, the container continues to run. If PID 1 exited while our shell was active, the container would terminate and our shell would also exit.

We can stop and start the container:

```
# docker stop nginx
nginx
# docker ps
IMAGE  COMMAND                  STATUS        PORTS
# docker start nginx
# docker ps
IMAGE  COMMAND                  STATUS        PORTS
nginx  "nginx -g 'daemon off"  Up 2 minutes  0.0.0.0:80->80/tcp
```

docker start starts the container with the same arguments that were passed when the container was created with **docker run**.

When containers exit, they remain on the system in a dormant state. You can list all containers, including those that are stopped, with **docker ps -a**. It's not particularly harmful to keep unneeded old containers lying around, but it's considered poor hygiene and might cause name collisions if you reuse container names.

When we finish with the container, we can stop and remove it:

```
# docker stop nginx && docker rm nginx
```

docker run --**rm** runs a container and removes it automatically when it exits, but this works only for containers that are not daemonized with **-d**.

Volumes

The filesystem layers for most containers consist of static application code, libraries, and other supporting or OS files. The read/write filesystem layer allows containers to make local modifications to these layers. However, heavy reliance on the overlay filesystem isn't the best storage solution for data-intensive applications such as databases. For those kinds of apps, Docker has the notion of volumes.

A volume is an independent, writable directory within a container that's maintained separately from the union filesystem. If the container is removed, the data in the volume persists and can be accessed from the host. Volumes can also be shared among multiple containers.

We add a volume to a container with **docker**'s **-v** argument:

```
# docker run -v /data --rm --hostname web --name web -d nginx
```

If **/data** already exists within the container, any files found there are copied to the volume. We can find the volume on the host by running **docker inspect**:

```
# docker inspect -f '{{ json .Mounts }}' web
...
 "Mounts": [
    {
      "Name": "8f026ebb9c0cda27441fb7fd275c8e767685f260...f5fd1939823558",
      "Source": "/var/lib/docker/volumes/8f026ebb9c0cda...93823558/_data",
      "Destination": "/data",
      "Driver": "local",
      "Mode": "",
      "RW": true,
      "Propagation": ""
    }
  ]
```

The **inspect** subcommand returns verbose output; we applied a filter so that only the mounted volumes would be printed. If the container terminates or needs to be removed, we can find the data volume at the Source directory on the host. The Name looks more like an ID, but it's useful if we need to identify the volume later.

For a higher-level overview of volumes on the system, we run **docker volume ls**.

Docker also supports "bind mounts," which mount volumes on the host and in containers simultaneously. For example, we can bind-mount **/mnt/data** from the host to **/data** in the container with the following command:

```
# docker run -v /mnt/data:/data --rm --name web -d nginx
```

When the container writes to **/data**, the changes are also visible in **/mnt/data** on the host.

For bind-mounted volumes, Docker does not copy existing files from the container's mount directory to the volume. As with a traditional filesystem mount, the volume's contents supersede the original contents of the container's mount directory.

When running containers in the cloud, we suggest combining bind mounts with the block storage options offered by cloud providers. For example, AWS's Elastic Block Storage volumes make great backing stores for Docker bind mounts. They have built-in snapshot facilities and can move among EC2 instances. They can also be copied between nodes, which makes it straightforward for other systems to retrieve a container's data. You can leverage EBS's native snapshotting facilities to create a simple backup system.

Data volume containers

One helpful pattern that has emerged from real-world experience is the data-only container. Its purpose is to hold a volume configuration on behalf of other containers so that those containers can be easily restarted and replaced.

Create a data container by using either a normal volume or a bind-mounted volume from the host. The data container never actually runs.

```
# docker create -v /mnt/data:/data --name nginx-data nginx
```

Now you can use the data container's volume in the nginx container:

```
# docker run --volumes-from nginx-data -p 80:80 --name web -d nginx
```

The "web" container has read and write access to the **/data** volume of the data-only "nginx-data" container. "web" can be restarted, removed, or replaced, but so long as it is started with --**volumes-from**, the files in **/data** will remain persistent.

In truth, combining data persistence with containers is a bit of an impedance mismatch. Containers are meant to be created and removed at a moment's notice in response to external events. The ideal is to have a fleet of identical servers that run **dockerd**, with containers being deployable to any of the servers. Once you add persistent data volumes, however, the container becomes coupled to a particular server. As much as we'd like to be living in the ideal world, many applications do need persistent data.

Docker networks

As discussed in *Networking* on page 918, there is more than one way to connect containers to the network. During installation, Docker creates three default networking options. List them with **docker network ls**:

```
# docker network ls
NETWORK ID          NAME                DRIVER
6514e7108508        bridge              bridge
1a72c1e4b230        none                null
e0f4e608c92c        host                host
```

See page 440 for more information about ***iptables****.*

In the default bridge mode, containers reside on a private namespaced network within the host. The bridge connects the host's network to the container namespace. When you create a container and map a port from the host with **docker run -p**, Docker creates **iptables** rules that route traffic from the host's public interface to the container's interface on the bridge network.

With "host" networking, no separate network namespace is used. Instead, the container shares the network stack with the host, including all its interfaces. Ports exposed by the container are also exposed on the interfaces of the host. Some software behaves better when running with host networking, but this configuration can also lead to port conflicts and other problems.

"None" networking indicates that Docker shouldn't take any steps whatsoever to configure networking. It is intended for advanced use cases that have custom networking requirements.

Pass the --**net** argument to **docker run** to select a container's network.

Namespaces and the bridge network

A bridge is a Linux kernel feature that connects two network segments. During installation, Docker quietly creates a bridge called docker0 on the host. Docker chooses an IP address space for the far side of the bridge that it calculates as unlikely to collide with any networks reachable by the host. Each container is given a namespaced virtual network interface that has an IP address within the bridged network range.

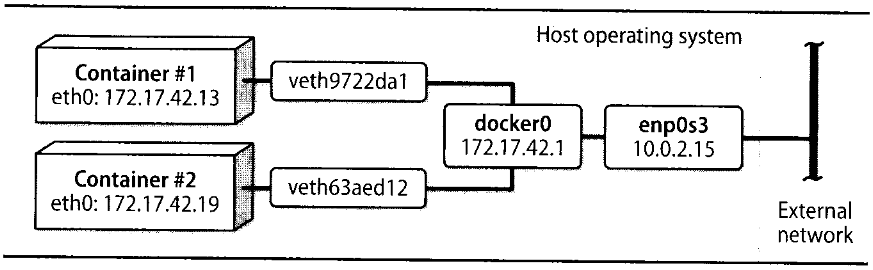
The address selection algorithm is practical but not perfect. Your network may have routes that aren't visible from the host. If a collision occurs, the host will no longer be able to access the remote network that has the overlapping address space, but it will be able to reach local containers. If you find yourself in this situation or if you need to customize the bridge's address space for some other reason, use the --**fixed-cidr** argument to **dockerd**.

Network namespaces rely on virtual interfaces, strange constructs that are created in pairs, where one side is in the host's namespace and the other is in the container's. Data flows in one end of the pair and out the other end, thus connecting the container to the host. In most cases a container has only one such pair. Exhibit C illustrates the concept.

One half of each pair is visible from the host's networking stack. For example, here are the visible interfaces on a CentOS host with just one container running:

```
centos$ ip addr show
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
    state UP qlen 1000
    link/ether 08:00:27:c3:36:f0 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic enp0s3
       valid_lft 71368sec preferred_lft 71368sec
    inet6 fe80::a00:27ff:fec3:36f0/64 scope link
       valid_lft forever preferred_lft forever
```

Exhibit C A docker bridge network



```
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    state UP
    link/ether 02:42:d4:30:59:24 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:d4ff:fe30:5924/64 scope link
       valid_lft forever preferred_lft forever
53: veth584a021@if52: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
    noqueue master docker0 state UP
    link/ether d6:39:a7:bd:bf:eb brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::d439:a7ff:febd:bfeb/64 scope link
       valid_lft forever preferred_lft forever
```

The output shows enp0s3, the primary interface on the host, and docker0, the virtual Ethernet bridge, which uses the 172.17.42.0/16 range. The veth interface is the host side of the virtual interface pair that connects the container to the bridged network.

The container's side of the bridged pair is not visible from the host without low-level inspection of the networking stack. This invisibility is just a side effect of the way network namespaces work. However, we can find the interface by inspecting the container itself:

```
# docker inspect -f '{{ json .NetworkSettings.Networks.bridge }}' nginx
    "bridge": {
      "Gateway": "172.17.42.1",
      "IPAddress": "172.17.42.13",
      "IPPrefixLen": 16,
      "MacAddress": "02:42:ac:11:00:03"
    }
```

The container's IP address is 172.17.42.13, and the default gateway is the docker0 bridge interface. (This is the bridge network depicted in Exhibit C.)

In the default bridge configuration, all containers can communicate with one another because they are all on the same virtual network. However, you can create additional network namespaces to isolate containers from one another. That way, you can serve multiple, isolated environments from the same set of container instances.

Network overlays

Docker has lots of additional networking flexibility available to help with advanced use cases. For example, you can create user-defined private networks that automatically have container linking. With network overlay software, containers running on separate hosts can route traffic to each other through a private network address space. Virtual eXtensible LAN (VXLAN) technology, described in RFC7348, is one system that can be combined with containers to implement advanced networking capabilities. See the Docker networking documentation for more details.

Storage drivers

UNIX and Linux systems offer multiple ways to implement a union filesystem. Docker is technology-agnostic in this regard and filters all filesystem operations through a storage driver that you select.

The storage driver is configured as part of the **docker daemon** launch options. Your choice of storage engine has important consequences for performance and stability, especially in production environments that support many containers. Table 25.2 shows the current menu of drivers.

The VFS driver effectively disables the use of a union filesystem. Docker creates a complete copy of an image for each container, resulting in higher disk usage and longer container start times. However, this implementation is simple and robust. If your use case involves long-lived containers, VFS is a reliable choice. We've never encountered a site that uses VFS in production, however.

Btrfs and ZFS are also not true union filesystems. However, they implement overlays efficiently and reliably because they natively support copy-on-write filesystem clones. Docker support for Btrfs and ZFS is currently limited to a few specific Linux distributions (and FreeBSD, for ZFS), but these are good options to keep an eye on for the future. The less filesystem magic specific to the container system, the better.

Storage driver selection is a nuanced topic. Unless you or somebody on your team has comprehensive knowledge of one of these filesystems, we recommend that you stick with the default for your distribution. The Docker storage driver documentation has further information.

dockerd option editing

You'll inevitably need to modify some of **dockerd**'s settings. Tuning options include the storage engine, DNS options, and the base directory in which images and metadata are stored. Run **dockerd -h** to see a complete list of arguments.

Table 25.2 Docker storage drivers

Driver	Description and comments
`aufs`	A reimplementation of the original UnionFS The original Docker storage engine Default for Debian and Ubuntu Now deprecated because it's not part of the mainline Linux kernel
`btrfs`	Uses the Btrfs copy-on-write filesystem (see page 783) Btrfs is stable and is included in the mainline Linux kernel Docker's use is distribution-limited and somewhat experimental
`devicemapper`	Default for RHEL/CentOS 6 Direct LVM mode strongly recommended but needs configuration Has a history of bugs Study Docker's devicemapper documentation
`overlay`	Based on OverlayFS Considered the replacement for AuFS The default in CentOS 7 if the overlay kernel module is loaded
`vfs`	Not a real union filesystem Slow but stable, suitable for some production environments Good as a proof of concept or as a testbed
`zfs`	Uses the ZFS copy-on-write filesystem (see page 773) Default for FreeBSD Considered experimental on Linux

You can examine a running daemon's configuration with **docker info**:

```
centos# docker info
Containers: 6
  Running: 0
  Paused: 0
  Stopped: 6
Images: 9
Server Version: 1.10.3
Storage Driver: overlay
  Backing Filesystem: xfs
Logging Driver: json-file
Plugins:
  Volume: local
  Network: bridge null host
Kernel Version: 3.10.0-327.10.1.el7.x86_64
Operating System: CentOS Linux 7 (Core)
OSType: linux
Architecture: x86_64
```

This is a good place to check that any customizations you made have taken effect.

Docker conforms to the operating system's native **init** system for managing daemon processes, including settings for startup options. For example, on a distribution that uses **systemd**, the following command edits the Docker service unit to set a nondefault storage driver, a set of DNS servers, and a custom address space for the bridge network:

*See Chapter 2 for more information about editing unit files with **systemctl**.*

```
$ systemctl edit docker
[Service]
ExecStart=
ExecStart=/usr/bin/docker daemon -D --storage-driver overlay \
   --dns 8.8.8.8 --dns 8.8.4.4 --bip 172.18.0.0/19
```

The redundant ExecStart= is not a mistake. It's a **systemd**-ism that clears the default setting, ensuring that the new definition is used exactly as shown. Once the edits are complete, we restart the daemon with **systemctl** and review the changes:

```
centos$ sudo systemctl restart docker
centos$ sudo systemctl status docker
  docker.service
     Loaded: loaded (/etc/systemd/system/docker.service; static;
       vendor preset: disabled)
  Drop-In: /etc/systemd/system/docker.service.d
           └─override.conf
   Active: active (running) since Wed 2016-03-09 23:14:56 UTC; 12s ago
 Main PID: 4328 (docker)
   CGroup: /system.slice/docker.service
           └─4328 /usr/bin/docker daemon -D --storage-driver overlay
     --dns 8.8.8.8 --dns 8.8.4.4 --bip 172.18.0.0/19
...
```

On systems running **upstart**, configure daemon options in **/etc/default/docker**. For older systems with SysV-style **init**, use **/etc/sysconfig/docker**.

By default, **dockerd** listens for connections from **docker** on the UNIX domain socket at **/var/run/docker.sock**. To set the daemon to listen on a TLS socket instead, use the daemon option **-H tcp://0.0.0.0:2376**. See page 940 for more details about how to set up TLS.

Image building

You can containerize your own applications by building images that include your application code. The build process begins with a base image. You add your application by committing any changes as new layers and saving the image to the local image database. You can then create containers from the image. You can also push your image to a registry to make it accessible to other systems running Docker.

Each layer of an image is identified by a cryptographic hash of its contents. The hash serves as a validation system that lets Docker confirm that no corruption or malicious intervention has modified the contents of the image.

Choosing a base image

Before creating a custom image, choose a suitable base. The rule of thumb for base images is that the smaller footprint, the better. The base should have what you need to run your software and nothing more.

Many of the official images are based on a distribution called Alpine Linux, which weighs in at a lean 5MB but may have library incompatibilities with some applications. The Ubuntu image is larger at 188MB, but still small in comparison to a typical server installation. You might be able to find a base image that has your application run-time components already configured. Default base images exist for the most common languages, run-times, and application platforms.

Thoroughly vet your base image before you make a final decision. Examine the base's **Dockerfile** (see the next section) and any nonobvious dependencies to avoid surprises. Base images may have unexpected requirements or include vulnerable versions of software. In some circumstances, you may need to copy the **Dockerfile** of a base image and rebuild it to suit your needs.

When **dockerd** downloads an image, it downloads only the layers that it doesn't already have. If all your applications use the same base, there is less data for the daemon to download and containers start faster when first run.

Building from a ***Dockerfile***

A **Dockerfile** is a recipe for building an image. It contains a series of instructions and shell commands. The **docker build** command reads the **Dockerfile**, runs its instructions in sequence, and commits the result as an image. Software projects that have a **Dockerfile** usually keep it in the root directory of the Git repository to facilitate building new images that contain that software.

The first instruction in a **Dockerfile** specifies an image to use as the base. Each subsequent instruction commits a change to a new layer, which is used in turn as the base for the next instruction. Each layer includes only the changes from the previous layer. The union filesystem merges the layers to create a container's root filesystem.

Here is a **Dockerfile** that builds the official NGINX image for Debian:[5]

```
FROM debian:jessie
MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"
ENV NGINX_VERSION 1.10.3-1~jessie
RUN apt-get update \
    && apt-get install -y ca-certificates nginx=${NGINX_VERSION} \
    && rm -rf /var/lib/apt/lists/*
# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log \
    && ln -sf /dev/stderr /var/log/nginx/error.log
EXPOSE 80 443
CMD ["nginx", "-g", "daemon off;"]
```

5. From github.com/nginxinc/docker-nginx. Slightly simplified.

NGINX uses the debian:jessie image as a base. After declaring a maintainer, the file sets an environment variable (`NGINX_VERSION`) that's then available to every subsequent instruction in the **Dockerfile** and also to any process that runs inside the container once the image has been built and instantiated. The first `RUN` instruction does the heavy lifting by installing NGINX from a package repository.

By default, NGINX sends log data to **/var/log/nginx/access.log**, but the convention for containers is to log messages to STDOUT. In the final `RUN` command, the maintainers use a symbolic link to redirect the access log to the STDOUT device file. Similarly, errors are redirected to the container's STDERR.

The `EXPOSE` command tells **dockerd** which ports the container listens on. The exposed ports can be overridden at container run time with the **-p** option to **docker run**.

The final instruction in the NGINX **Dockerfile** is the command that **dockerd** should execute when it starts the container. In this case, the container runs the **nginx** binary as a foreground process.

See Table 25.3 for a rundown of common **Dockerfile** instructions. The reference manual at docs.docker.com is the authoritative documentation.

Composing a derived ***Dockerfile***

We can use a very simple **Dockerfile** to build a derived NGINX image that adds a custom **index.html**, replacing the default from the official image:

```
$ cat index.html
<!DOCTYPE html>
<title>ULSAH index.html file</title>
<p>A simple Docker image, brought to you by ULSAH.</p>
$ cat Dockerfile
FROM nginx
# Add a new index.html to the document root
ADD index.html /usr/share/nginx/html/
```

Other than having a custom **index.html**, our new image will be identical to the base image. Here's how we build the customized image:

```
# docker build -t nginx:ulsah .
Step 1 : FROM nginx
  ---> fd19524415dc
Step 2 : ADD index.html /usr/share/nginx/html/
  ---> c0c25eaf7415
Removing intermediate container 04cc3278fdb4
Successfully built c0c25eaf7415
```

We use **docker build** with **-t nginx:ulsah** to create an image with the name nginx and the tag ulsah to distinguish it from the official NGINX image. The trailing dot tells **docker build** where to search for the **Dockerfile** (in this case, the current directory).

Table 25.3 Abbreviated list of Dockerfile instructions

Instruction	What it does
ADD	Copies files from the build host to the image[a]
ARG	Sets variables that can be referenced during the build but not from the final image; not intended for secrets
CMD	Sets the default commands to execute in a container
COPY	Like ADD, but only for files and directories
ENV	Sets environment variables available to all subsequent build instructions and containers spawned from this image
EXPOSE	Informs **dockerd** of the network ports exposed by the container
FROM	Sets the base image; must be the first instruction
LABEL	Sets image tags (visible with **docker inspect**)
RUN	Runs commands and saves the result in the image
STOPSIGNAL	Specifies a signal to send to the process when told to quit with **docker stop**; defaults to SIGKILL
USER	Sets the account name to use when running the container and any subsequent build instructions
VOLUME	Designates a volume for storing persistent data
WORKDIR	Sets the default working directory for subsequent instructions

a. The source can be a file, directory, tarball, or remote URL.

Now we can run the image and see our customized **index.html**:

```
# docker run -p 80:80 --name nginx-ulsah -d nginx:ulsah
$ curl localhost
<!DOCTYPE html>
<title>ULSAH index.html file</title>
<p>A simple Docker image, brought to you by ULSAH.</p>
```

We can check that our image is listed among the local images by running the command **docker images**:

```
# docker images | grep ulsah
REPOSITORY     TAG        IMAGE ID       CREATED            SIZE
nginx          ulsah      c0c25eaf7415   3 minutes ago      134.6 MB
```

To remove images, run **docker rmi**. You can't remove an image until you've stopped and removed any containers that are using it:

```
# docker ps | grep nginx:ulsah
IMAGE        COMMAND                  STATUS          PORTS
nginx:ulsah  "nginx -g 'daemon off"   Up 37 seconds   0.0.0.0:80->80/tcp
# docker stop nginx-ulsah && docker rm nginx-ulsah
nginx-ulsah
nginx-ulsah
# docker rmi nginx:ulsah
```

Both **docker stop** and **docker rm** echo the name of the container they affect, resulting in "nginx-ulsah" being printed twice.

Registries

A registry is an index of Docker images that **dockerd** can access through HTTP. When an image is requested that doesn't exist on the local disk, **dockerd** pulls it from the registry. Images are uploaded to a registry with **docker push**. Although image operations are initated by the **docker** command, only **dockerd** actually interacts with registries.

Docker Hub is a hosted registry service run by Docker, Inc. It hosts images for many distributions and open source projects, including all our example Linux systems. The integrity of these official images is verified through a content trust system, thus ensuring that the image you download is provided by the vendor whose name is on the label. You can also publish your own images to Docker Hub for others to use.

Anyone can download public images from Docker Hub, but with a subscription you can also create private repositories. Once you have a paid account at hub.docker.com, log in from the command line with **docker login** to access the private registry so that you can push and pull your own custom images. You can also trigger an image build whenever a commit is detected on a GitHub repository.

Docker Hub is not the only subscription-based registry. Others include quay.io, Artifactory, Google Container Registry, and the Amazon EC2 Container Registry.

Docker Hub is the generous benefactor of the greater image ecosystem, and it also benefits from being the default registry when nothing more specific is requested. For example, the command

```
# docker pull debian:jessie
```

first looks for a local copy of the image. If the image isn't available locally, the next stop is Docker Hub. You can tell **docker** to use a different registry by including a hostname or URL in the image specification:

```
# docker pull registry.admin.com/debian:jessie
```

Similarly, when building an image to push to a custom registry, you must tag it with the registry's URL, and you must authenticate before you push:

```
# docker tag debian:jessie registry.admin.com/debian:jessie .
...
# docker login https://registry.admin.com
Username: ben
Password: <password>
# docker push registry.admin.com/debian:jessie
```

Docker saves the login details to a file in your home directory called **.dockercfg** so that you need not log in again the next time you interact with the private registry.

For performance or security reasons, you might prefer to run your own image registry. The registry project is open source (github.com/docker/distribution), and a simple registry is easy to run as a container:[6]

```
# docker run -d -p 5000:5000 --name registry registry:2
```

The registry service is now running on port 5000. You can pull an image from it by qualifying the name of the image you're seeking:

```
# docker pull localhost:5000/debian:jessie
```

The registry implements two authentication methods: `token` and `htpasswd`. `token` delegates authentication to an external provider, which is likely to require custom development effort. `htpasswd` is simpler and allows HTTP basic authentication for registry access. Alternatively, you can set up a proxy (e.g., NGINX) to handle authentication. Always run the registry with TLS.

The default private registry configuration is not appropriate for a large-scale deployment. Considerations for production use include storage space, authentication and authorization requirements, image cleanup, and other maintenance tasks.

As your containerized environment expands, your registry will be inundated with new images. For users working in the cloud, an object store such as Amazon S3 or Google Cloud Storage is one possible way to store all this data. The registry natively supports both services.

Better yet, you can outsource your registry functions to the registries built into your cloud platform of choice and have one less thing to worry about. Both Google and Amazon run managed container registry services. You pay for storage and for the network traffic to upload and download images.

25.3 Containers in practice

Once you're comfortable with the general way that containers work, you'll find that certain administrative chores need to be approached differently in a containerized world. For example, how do you manage log files for containerized applications? What are some security considerations? How do you troubleshoot errors?

The list below offers a few rules of thumb to help you adjust to life inside a container:

- When your application needs to run a scheduled job, don't run **cron** in a container. Use the **cron** daemon from the host (or a **systemd** timer) to schedule a short-lived container that runs the job and exits. Containers are meant to be disposable.
- Need to log in and check out what a process is doing? Don't run **sshd** in your container. Log in to the host with **ssh**, then use **docker exec** to open an interactive shell.

6. The **registry:2** tag differentiates the latest-generation registry from the previous version, which implements an API that is incompatible with current versions of Docker.

- If possible, set up your software to accept its configuration information from environment variables. You can pass environment variables to containers with the **-e** *KEY=value* argument to **docker run**. Or set up many variables at once from a separate file with **--env-file** *filename*.
- Ignore the commonly dispensed advice "one process per container." That's nonsense. Split processes into separate containers only when it makes sense to do so. For example, it's usually a good idea to run an application and its database server in separate containers because they are separated by clear architectural boundaries. But it's perfectly OK to have more than one process in a container when that's appropriate. Use common sense.
- Focus on the automatic creation of containers for your environment. Write scripts to build images and upload them to registries. Make sure that software deployment procedures involve replacing containers, not updating them in place.
- On that note, avoid maintaining containers. If you're accessing a container manually to fix something, figure out what the problem is, resolve it in the image, then replace the container. Immediately update your automation tooling if necessary.
- Stuck? Ask questions on the Docker User mailing list, on the Docker Community Slack, or in the #docker IRC channel on freenode.

Everything an application needs to function should be available within its container: the filesystem, network access, and kernel facilities. The only processes that run in a container are the ones that you start. It is atypical of containers to run normal OS services such as **cron**, **rsyslogd**, and **sshd**, although it is certainly possible to do so. Those duties are best left to the host OS. If you find yourself needing these services within a container, reconsider your problem and see if you can solve it in a more container-friendly way.

Logging

UNIX and Linux applications traditionally use syslog (now the **rsyslogd** daemon) to process log messages. Syslog handles log filtering, sorting, and routing to remote systems. Some applications don't use syslog and instead write directly to log files.

Containers do not run syslog. Instead, Docker collects the logs for you through logging drivers. Container processes need only write logs to STDOUT and errors to STDERR. Docker collects those messages and sends them to a configurable destination.

If your software supports logging only to files, apply the same technique as the NGINX example on page 933: create symbolic links from log files to **/dev/stdout** and **/dev/stderr** when you build the image.

Docker forwards the log entries it receives to a selectable logging driver. Table 25.4 lists some of the more common and useful logging drivers.

Table 25.4 Docker logging drivers

Driver	What it does
`json-file`	Writes JSON logs in the daemon's data directory (default)[a]
`syslog`	Writes logs to a configurable syslog destination[b]
`journald`	Writes logs to the **systemd** journal[a]
`gelf`	Writes logs in the Graylog Extended Log Format
`awslogs`	Writes logs to the AWS CloudWatch service
`gcplogs`	Writes logs to Google Cloud Logging.
`none`	Does not collect logs

a. Log entries stored this way are accessible through the **docker logs** command.
b. Supports UDP, TCP, and TCP+TLS.

When using `json-file` or `journald`, you can access log data from the command line through **docker logs** *container-id.*

You set the default logging driver for **dockerd** with the **--log-driver** option. You can also assign a logging driver at container run time with **docker run --logging-driver**. Some drivers accept additional options. For example, the **--log-opt max-size** option configures log file rotation for the `json-file` driver. Use this option to avoid filling up the disk with log files. Refer to the Docker logging documentation for complete details.

Security advice

Container security relies on processes within containers being unable to access files, processes, and other resources outside their sandbox. Vulnerabilities that allow attackers to escape containers—known as breakout attacks—are serious but rare. The code that underlies container isolation has been in the Linux kernel since at least 2008; it's mature and stable. As with bare-metal or virtualized systems, insecure configurations are a far more likely source of compromises than are vulnerabilities in the isolation layer.

Docker maintains an interesting list of known software vulnerabilities that are and are not mitigated by containerization. See docs.docker.com/engine/security/non-events.

Restrict access to the daemon

Above all, protect the docker daemon. Because **dockerd** necessarily runs with elevated privileges, it's trivial for any user with access to the daemon to gain full root access to the host.

The following sequence of commands demonstrates the risk:

```
$ id
uid=1001(ben) gid=1001(ben) groups=1001(ben),992(docker)
# docker run --rm -v /:/host -t -i debian bash
root@e51ae86c5f7b:/# cd /host
root@e51ae86c5f7b:/host# ls
bin   dev  home  lib64  mnt  proc  run   srv  test  usr
boot  etc  lib   media  opt  root  sbin  sys  tmp   var
```

This transcript shows that any user in the docker group can mount the host's root filesystem to a container and gain full control of its contents. This is just one of many possible ways to elevate privileges through Docker.

If you use the default UNIX domain socket to communicate with the daemon, add only trusted users to the docker group, which has access to the socket. Better yet, control access through **sudo**.

Use TLS

See page 1008 for information about TLS.

We said it before, and we'll say it again: if the docker daemon must be remotely accessible (**dockerd -H**), require the use of TLS to encrypt network communications and to mutually authenticate the client and server.

Setting up TLS involves having a certificate authority issue certificates to the docker daemon and clients. Once the key pairs and certificate authority are in place, actually enabling TLS for **docker** and **dockerd** is a simple matter of supplying the right command-line arguments. Table 25.5 lists the essential settings.

Table 25.5 TLS arguments common to docker and dockerd

Argument	Meaning or argument
--tlsverify	Require authentication
--tlscert[a]	Path to a signed certificate
--tlskey[a]	Path to a private key
--tlscacert[a]	Path to the certificate of a trusted authority

a. Optional. The default locations are **~/.docker/{cert,key,ca}.pem**.

Successful use of TLS relies on a mature certificate management processes. Certificate issuance, revocation, and expiration are a few of the issues that need attention. Heavy is the burden of a security-conscious administrator.

Run processes as unprivileged users

Processes in containers should run as nonroot users, just as they should on a full-fledged operating system. This practice limits an attacker's ability to launch breakout attacks. When you are writing a **Dockerfile**, use the USER instruction to run future commands in the image under the named user account.

Use a read-only root filesystem

To further restrict containers, you can specify **docker run --read-only**, thereby limiting the container to a read-only root filesystem. This works well for stateless services that never need to write. You can also mount a read/write volume that your process can modify, but leave the root filesystem read-only.

Limit capabilities

See page 82 for more information about Linux capabilities.

The Linux kernel defines 40 separate capabilities that can be assigned to processes. By default, Docker containers are granted a large subset of these. You can enable an even greater subset by starting a container with the **--privileged** flag. However, this option disables many of the isolation benefits of using Docker. You can tune the specific capabilities that are available to containerized processes with the **--cap-add** and **--cap-drop** arguments:

```
# docker run --cap-drop SETUID --cap-drop SETGID debian:jessie
```

You can also drop all privileges and add back just the ones you need:

```
# docker run --cap-drop ALL --cap-add NET_RAW debian:jessie
```

Secure images

The Docker content trust feature validates the authenticity and integrity of images in a registry. The publisher of the image signs it with a secret key, and the registry validates it with the corresponding public key. This process ensures that the image was produced by the expected creator. You can use content trust to sign your own images or to validate the images in a remote registry. The feature is available on Docker Hub and on some third party registries, such as Artifactory.

Unfortunately, most of the content on Docker Hub is unsigned and should be considered untrustworthy. Indeed, most images on the Hub are never patched, updated, or audited in any way.

This lack of a proper chain of trust associated with many Docker images is representative of the miserable state of security on the Internet in general. It's quite common for software packages to depend on third party libraries with little or no concern being given to the trustworthiness of the content that's pulled in. Some software repositories have no cryptographic signatures whatsoever. It's also common to find articles that actively encourage disabling validation. Responsible system administrators are highly suspicious of unknown and untrusted software repositories.

Debugging and troubleshooting

Containers bring with them a particularly heinous complement of obscure debugging techniques. When an application is containerized, its symptoms become more difficult to characterize and their root causes more puzzling. Many applications can run without modification inside a container, but in some scenarios they may behave differently. You might also encounter bugs within Docker itself. This section helps navigate these treacherous waters.

Errors usually manifest themselves in log files, so that's the first place to look for information. Use the advice in *Logging* on page 938 to configure logging for containers, and always review the logs when you encounter issues.

If you experience problems with a running container, try

```
docker exec -ti containername bash
```

to open an interactive shell. From there you can attempt to reproduce the problem, examine the filesystem for evidence, and search for configuration errors.

If you see errors related to the docker daemon or if you have trouble starting it, search the issues list at github.com/moby/moby. You may find others that have the same problem, and one of them may have identified a potential fix or workaround.

Docker doesn't automatically clean up images or containers. When neglected, these remnants can consume an inordinate amount of disk space. If your container workload is predictable, configure a **cron** job to clean up by running **docker system prune** and **docker image prune.**

A related annoyance are "dangling" volumes, volumes that were once attached to a container but for which the container has since been removed. Volumes are independent of containers, so any files within them will continue to consume disk space until the volumes are destroyed. You can use the following incantation to clean out orphaned volumes:

```
# docker volume ls -f dangling=true  # List dangling volumes
# docker volume rm $(docker volume ls -qf dangling=true)  # Remove 'em
```

Base images you depend on may have a VOLUME instruction in their **Dockerfile**. If you don't notice this case, you might end up with a full disk after running a few containers from that image. You can show the volumes associated with a container by running **docker inspect**:

```
# docker inspect -f '{{ .Volumes }}' container-name
```

25.4 Container clustering and management

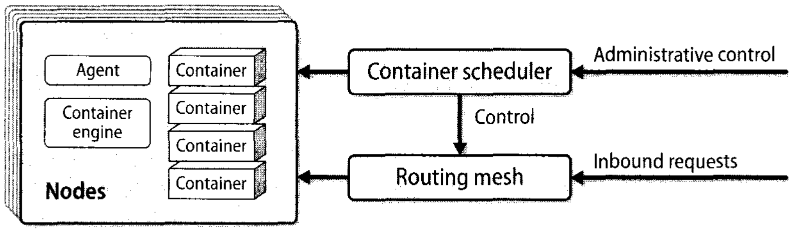
One of the great promises of containerization is the prospect of co-locating many applications on the same host while avoiding interdependencies and conflicts, thereby making more efficient use of servers. This is an appealing vision, but the Docker engine is responsible only for individual containers, not for answering the broader question of how to run many containers on distributed hosts in a highly available configuration.

Configuration management tools such as Chef, Puppet, Ansible, and Salt all support Docker. They can ensure that hosts run a certain set of containers with declared configurations. They also support image building, registry interfaces, network and volume management, and other container-related chores. These tools centralize and standardize container configuration, but they do not solve the problem of conducting the deployment of many containers across a network of servers. (Note that although configuration management systems are useful for a variety of container-related tasks, you will rarely need to use configuration management *inside* of containers.)

For network-wide container deployments, you need container orchestration software, also known as container scheduling or container management software. An entire symphony of open source and commercial tooling is available to handle large numbers of containers. Such tools are crucial for running containers at scale in a production context.

To understand how these systems work, think of the servers on the network as a farm of compute capacity. Each node in the farm offers CPU, memory, disk, and network resources to the scheduler. When the scheduler receives a request to run a container (or set of containers), it places the container on a node that has sufficient spare resources to meet the container's needs. Because the scheduler knows where containers have been placed, it can also assist in routing network requests to the correct nodes within the cluster. Administrators interact with the container management system rather than dealing with any individual container engine. Exhibit D illustrates this architecture.

Exhibit D Basic container scheduler architecture



Container management systems supply several helpful features:

- Scheduling algorithms select the best node in light of a job's requested resources and the utilization of the cluster. For example, a job with high bandwidth requirements might be slotted onto a node with a 10 Gb/s network interface.

- Formal APIs allow programs to submit jobs to the cluster, opening the door to integration with external tools. It's easy to use container management systems in conjunction with CI/CD systems to simplify software deployments.
- Container placement can accommodate the needs of high-availability configurations. For example, an application may need to run on host nodes in several distinct geographical regions.
- Health monitoring is built in. The system can terminate and reschedule unhealthy jobs and can route jobs away from unhealthy nodes.
- It's easy to add or remove capacity. If your compute farm doesn't have enough resources available to satisfy demand, you can simply add another node. This facility is especially well suited to cloud environments.
- The container management system can interface with a load balancer to route network traffic from external clients. This facility obviates the complex administrative process of manually configuring network access to containerized applications.

One of the most challenging tasks in a distributed container system is mapping service names to containers. Remember that containers are typically ephemeral in nature and may have dynamic ports assigned. How do you map a friendly, persistent service name to multiple containers, especially when the nodes and ports change frequently? This problem is known as service discovery, and container management systems have various solutions.

It helps to be familiar with the underlying container execution engine before diving into orchestration tooling. All the container management systems we're aware of rely on Docker as the default container execution engine, although some systems also support other engines.

A synopsis of container management software

Despite their relative youth, the container management tools we discuss below are mature beyond their years and can be considered production grade. In fact, many are already used in production at high-profile, large-scale technology companies. Most are open source and have sizable user communities. Based on recent trends, we anticipate substantial development in this area in the coming years.

In the upcoming sections, we highlight the functionality and features of the most widely used systems. We also mention their integration points and common use cases.

Kubernetes

Kubernetes—sometimes shortened to "k8s" because there are eight letters between the leading "k" and the trailing "s"—has emerged as a leader in the container management space. It originated within Google and was launched by some of the same developers that worked on Borg, Google's internal cluster manager. Kubernetes was released as an open source project in 2014 and now has more than a thousand active contributors. It has the most features and the fastest development cycle of any system we're aware of.

Kubernetes consists of a few separate services that integrate to form a cluster. The basic building blocks include

- The API server, for operator requests
- A scheduler, for placing tasks
- A controller manager, for tracking the state of the cluster
- The Kubelet, an agent that runs on all cluster nodes
- cAdvisor, for monitoring container metrics
- A proxy, for routing incoming requests to the appropriate container

The first three items on this list run on a set of masters (which can optionally serve dual duty as nodes) for high availability. The Kubelet and cAdvisor processes run on each node, handling requests from the controller manager and reporting statistics about the health of their tasks.

In Kubernetes, containers are deployed as a "pod" which contains one or more containers. All containers in a pod are guaranteed to be co-located on the same node. Pods are assigned a cluster-wide unique IP address, and they are labeled for identification and placement purposes.

Pods are not meant to be long-lived. If a node dies, the controller schedules a replacement pod on a different node with a new IP address. Therefore, you cannot use the address of a pod as a durable name.

Services are collections of related pods with an address that is guaranteed not to change. If a pod within a service dies or fails a health check, the service removes that pod from its rotation. You can also use the built-in DNS server to assign resolvable names to services.

Kubernetes has integrated support for service discovery, secret management, deployment, and pod autoscaling. It has pluggable networking options to enable container network overlays. It can support stateful applications by migrating volumes among nodes as needed. Its CLI tool, **kubectl**, is one of the most intuitive that we've ever worked with. In short, it has more advanced features than we can possibly cover in this short section.

Although Kubernetes has the most active and engaged community and the most advanced features, those assets are accompanied by a steep learning curve. Recent versions have improved the experience for first-time users, but a full-fledged, customized Kubernetes deployment is not for the timid. Production k8s deployments impose a substantial administrative and operational burden.

The Google Container Engine service is implemented with Kubernetes, and it offers one of the best experiences for teams that want to run containerized workloads without the operational overhead of cluster management.

Mesos and Marathon

Mesos is an entirely different breed. It was conceived at the University of California at Berkeley around 2009 as a generic cluster manager. It quickly made its way to Twitter, where it now runs on thousands of nodes. Today, Mesos is a top-level project from the Apache Foundation and boasts a large number of enterprise users.

The major conceptual entities in Mesos are masters, agents, and frameworks. A master is a proxy between agents and frameworks. Masters relay offers of system resources from agents to frameworks. If a framework has a task to run, it chooses an offer and instructs the master to run the task. The master sends along the task details to the agent.

Marathon is a Mesos framework that deploys and manages containers. It includes a handsome user interface for managing applications and a simple, RESTful API. To run an application, you write a request definition in JSON format and submit it to Marathon through the API or the UI. Because it's an external framework, the deployment of Marathon is flexible. Marathon can run on the same node as the master for convenience, or it can run externally.

Support for multiple, coexisting frameworks is Mesos's biggest differentiator. Apache Spark, the big-data processing tool, and Apache Cassandra, a NoSQL database, both offer Mesos frameworks, thus allowing you to use Mesos agents as nodes in a Spark or Cassandra cluster. Chronos is a framework for scheduled jobs, rather like a version of **cron** that runs on a cluster instead of an individual machine. The ability to run so many frameworks on the same set of nodes is a nice feature and helps create a unified and centralized experience for administrators.

Unlike Kubernetes, Mesos does not come with batteries included. For example, load balancing and traffic routing are pluggable options that depend on your preferred solution. Marathon includes a tool, the Marathon-lb, that implements this service, or you can choose your own. We've had success using HashiCorp's Consul and HAProxy. Designing and implementing an exact solution is left as an exercise for the administrator.

Like Kubernetes, Mesos requires some contemplation to understand and use. Mesos and most of its frameworks rely on Apache Zookeeper for cluster coordination. Zookeeper is somewhat difficult to administer and is known for complex failure cases. In addition, a high-availability Mesos cluster requires a minimum of three nodes, which may be an onerous burden at some sites.

Docker Swarm

Not to be left behind, Docker offers Swarm, a container cluster manager built directly into Docker. The current incarnation of Swarm emerged in 2016 as an answer to the growing popularity of Mesos, Kubernetes, and other cluster managers that used Docker containers under the hood. Container orchestration is now a major focus for Docker, Inc.

Swarm is easier to get started with than is Mesos or Kubernetes. Any node that runs Docker can join the swarm as a worker node, and any worker node can also be a manager. There is no need to run separate nodes as masters.[7] Starting a swarm is as simple as running **docker swarm init**. There are no additional processes to manage and configure, and there is no state to track. It works out of the box.

You can use familiar **docker** commands to run services (as in Kubernetes, collections of containers) on the swarm. You declare the state you want to achieve ("three containers running my web application") and Swarm schedules the tasks on the cluster. It automatically handles failure states and zero-downtime updates.

Swarm has a built-in load balancer that adjusts automatically as containers are added or removed. The Swarm load balancer is not as full-featured as tools such as NGINX or HAProxy, but on the other hand, it doesn't require any administrative attention.

Swarm supplies a secure Docker experience by default. All connections between nodes in a swarm are TLS-encrypted, and no configuration is required on the part of the administrator. This is a major differentiator for Swarm when compared to its competitors.

AWS EC2 Container Service

AWS offers ECS, a container management service designed for EC2 instances (AWS's native virtual servers). In a manner reminiscent of many Amazon services, AWS launched ECS with minimal functionality but has steadily enhanced it over time. ECS has matured into a fine choice for sites that are already invested in AWS and want to stick to E-Z mode.

ECS is a "mostly managed" service. The cluster manager components are operated by AWS. Users run EC2 instances that have Docker and the ECS agent installed. The agent connects to the central ECS API and registers its resource availability. To run a task on your ECS cluster, you submit a task definition in JSON format through the API. ECS then schedules the task on one of your nodes.

Because the service is mostly managed, the barrier to entry is low. You can get started with ECS in just a few minutes. The service scales well to at least hundreds of nodes and thousands of concurrent tasks.

7. Strictly speaking this is true for Kubernetes and Mesos as well, but we've found it to be common practice to separate masters from agents in high-availability configurations.

ECS integrates with other AWS services. For example, load balancing among multiple tasks, along with the requisite service discovery, are handled by the Application Load Balancer service. You can add resource capacity to your ECS cluster by taking advantage of EC2 autoscaling. ECS also integrates with AWS's Identity and Access Manager service to grant permissions for your container tasks to interact with other services.

One of the most polished parts of ECS is the included Docker image registry. You can upload Docker images to the EC2 Container Registry, where they're stored and made available to any Docker client, whether it's running on ECS or not. If you're running containers on AWS, use the container registry in the same region as your instances. You'll achieve far better reliability and performance than with any other registry.

The ECS user interface, although functional, shares the limitations of other AWS interfaces. The AWS CLI tool has complete support for the ECS API. For management of applications on ECS, we recommend turning to third party, open source tools such as Empire (github.com/remind101/empire) or Convox (convox.com) for a more streamlined experience.

25.5 Recommended reading

DOCKER, INC. *Official Docker Documentation.* docs.docker.com. Docker has good documentation. It's comprehensive and usually up to date.

MATTHIAS, KARL, AND SEAN KANE. *Docker Up & Running.* Sebastopol, CA: O'Reilly Media, 2015. This book focuses on running Docker containers in production environments.

MOUAT, ADRIAN. *Using Docker: Developing and Deploying software with Containers.* Sebastopol, CA: O'Reilly Media, 2016. This book covers topics from basic to advanced and includes plenty of examples.

TURNBULL, JAMES. *The Docker Book.* www.dockerbook.com.

The Container Solutions blog at container-solutions.com/blog includes technical HOWTOs, best practices, and interviews with experts in the container space.