Running containers

By Docker Inc

Published: 2026-03-23 · Archived: 2026-04-05 14:20:29 UTC

Docker runs processes in isolated containers. A container is a process which runs on a host. The host may be local or remote. When you execute `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 run containers.

A `docker run` command takes the following form:

The `docker run` command must specify an image reference to create the container from.

Image references

The image reference is the name and version of the image. You can use the image reference to create or run a container based on an image.

- `docker run IMAGE[:TAG][@DIGEST]`
- `docker create IMAGE[:TAG][@DIGEST]`

An image tag is the image version, which defaults to `latest` when omitted. Use the tag to run a container from specific version of an image. For example, to run version `24.04` of the `ubuntu` image: `docker run ubuntu:24.04`.

Image digests

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.

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

Options

`[OPTIONS]` let you configure options for the container. For example, you can give the container a name (`--name`), or run it as a background process (`-d`). You can also set options to control things like resource constraints and networking.

Commands and arguments

You can use the `[COMMAND]` and `[ARG...]` positional arguments to specify commands and arguments for the container to run when it starts up. For example, you can specify `sh` as the `[COMMAND]`, combined with the `-i` and `-t` flags, to start an interactive shell in the container (if the image you select has an `sh` executable on `PATH`).

> 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.

When you start a container, the container runs in the foreground by default. If you want to run the container in the background instead, you can use the `--detach` (or `-d`) flag. This starts the container without occupying your terminal window.

While the container runs in the background, you can interact with the container using other CLI commands. For example, `docker logs` lets you view the logs for the container, and `docker attach` brings it to the foreground.

For more information about `docker run` flags related to foreground and background modes, see:

- `docker run --detach` : run container in background
- `docker run --attach` : attach to `stdin`, `stdout`, and `stderr`
- `docker run --tty` : allocate a pseudo-tty
- `docker run --interactive` : keep `stdin` open even if not attached

For more information about re-attaching to a background container, see `docker attach`.

You can identify a container in three ways:

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

The UUID identifier is a random ID assigned to the container by the daemon.

The daemon generates a random string name for containers automatically. You can also define a custom name using the `--name` flag. Defining a `name` can be a handy way to add meaning to a container. If you specify a `name`, you can use it when referring to the container in a user-defined network. This works for both background and foreground Docker containers.

A container identifier is not the same thing as an image reference. The image reference specifies which image to use when you run a container. You can't run `docker exec nginx:alpine sh` to open a shell in a container based on the `nginx:alpine` image, because `docker exec` expects a container identifier (name or ID), not an image.

While the image used by a container is not an identifier for the container, you find out the IDs of containers using an image by using the `--filter` flag. For example, the following `docker ps` command gets the IDs of all running containers based on the `nginx:alpine` image:

For more information about using filters, see Filtering.

Containers have networking enabled by default, and they can make outgoing connections. If you're running multiple containers that need to communicate with each other, you can create a custom network and attach the containers to the network.

When multiple containers are attached to the same custom network, they can communicate with each other using the container names as a DNS hostname. The following example creates a custom network named `my-net`, and runs two containers that attach to the network.

For more information about container networking, see Networking overview

By default, the data in a container is stored in an ephemeral, writable container layer. Removing the container also removes its data. If you want to use persistent data with containers, you can use filesystem mounts to store the data persistently on the host system. Filesystem mounts can also let you share data between containers and the host.

Docker supports two main categories of mounts:

- Volume mounts
- Bind mounts

Volume mounts are great for persistently storing data for containers, and for sharing data between containers. Bind mounts, on the other hand, are for sharing data between a container and the host.

You can add a filesystem mount to a container using the `--mount` flag for the `docker run` command.

The following sections show basic examples of how to create volumes and bind mounts. For more in-depth examples and descriptions, refer to the section of the storage section in the documentation.

Volume mounts

To create a volume mount:

The `--mount` flag takes two parameters in this case: `source` and `target`. The value for the `source` parameter is the name of the volume. The value of `target` is the mount location of the volume inside the container. Once you've created the volume, any data you write to the volume is persisted, even if you stop or remove the container:

The `target` must always be an absolute path, such as `/src/docs`. An absolute path starts with a `/` (forward slash). Volume names must start with an alphanumeric character, followed by `a-z0-9`, `_` (underscore), `.` (period) or `-` (hyphen).

Bind mounts

To create a bind mount:

In this case, the `--mount` flag takes three parameters. A type (`bind`), and two paths. The `source` path is the location on the host that you want to bind mount into the container. The `target` path is the mount destination inside the container.

By default, bind mounts require the source path to exist on the daemon host. If the source path doesn't exist, an error is returned. To create the source path on the daemon host if it doesn't exist, use the `bind-create-src` option:

Bind mounts are read-write by default, meaning that you can both read and write files to and from the mounted location from the container. Changes that you make, such as adding or editing files, are reflected on the host filesystem:

The exit code from `docker run` gives information about why the container failed to run or why it exited. The following sections describe the meanings of different container exit codes values.

125

Exit code `125` indicates that the error is with Docker daemon itself.

126

Exit code `126` indicates that the specified contained command can't be invoked. The container command in the following example is: `/etc` .

127

Exit code `127` indicates that the contained command can't be found.

Other exit codes

Any exit code other than `125` , `126` , and `127` represent the exit code of the provided container command.

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 6M.
`--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` .

Option	Description
`--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.
`-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.

Option	Description
`--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
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:

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

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).

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.

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.

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.

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:

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

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:

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.

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:

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

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:

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.

Cpuset constraint

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

Examples:

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

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:

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

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.

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.

> 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:

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

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`:

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`:

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`:

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`:

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`:

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`:

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

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:

Option	Description
`--cap-add`	Add Linux capabilities
`--cap-drop`	Drop Linux capabilities
`--privileged`	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).

The `--privileged` flag gives all capabilities to the container. When the operator executes `docker run --privileged`, Docker enables access to all devices on the host, and reconfigures AppArmor or SELinux to allow the container nearly all the same access to the host as processes running outside containers on the host. Use this flag with caution. For more information about the `--privileged` flag, see the `docker run` reference.

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.

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:

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
AUDIT_WRITE	Write records to kernel auditing log.
CHOWN	Make arbitrary changes to file UIDs and GIDs (see chown(2)).
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.
MKNOD	Create special files using mknod(2).
NET_BIND_SERVICE	Bind a socket to internet domain privileged ports (port numbers less than 1024).
NET_RAW	Use RAW and PACKET sockets.
SETFCAP	Set file capabilities.
SETGID	Make arbitrary manipulations of process GIDs and supplementary GID list.
SETPCAP	Modify process capabilities.
SETUID	Make arbitrary manipulations of process UIDs.
SYS_CHROOT	Use chroot(2), change root directory.

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

Capability Key	Capability Description
AUDIT_CONTROL	Enable and disable kernel auditing; change auditing filter rules; retrieve auditing status and filtering rules.
AUDIT_READ	Allow reading the audit log via multicast netlink socket.
BLOCK_SUSPEND	Allow preventing system suspends.
BPF	Allow creating BPF maps, loading BPF Type Format (BTF) data, retrieve JITed code of BPF programs, and more.
CHECKPOINT_RESTORE	Allow checkpoint/restore related operations. Introduced in kernel 5.9.
DAC_READ_SEARCH	Bypass file read permission checks and directory read and execute permission checks.
IPC_LOCK	Lock memory (mlock(2), mlockall(2), mmap(2), shmctl(2)).

Capability Key	Capability Description
IPC_OWNER	Bypass permission checks for operations on System V IPC objects.
LEASE	Establish leases on arbitrary files (see fcntl(2)).
LINUX_IMMUTABLE	Set the FS_APPEND_FL and FS_IMMUTABLE_FL i-node flags.
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.
NET_BROADCAST	Make socket broadcasts, and listen to multicasts.
PERFMON	Allow system performance and observability privileged operations using perf_events, i915_perf and other kernel subsystems
SYS_ADMIN	Perform a range of system administration operations.
SYS_BOOT	Use reboot(2) and kexec_load(2), reboot and load a new kernel for later execution.
SYS_MODULE	Load and unload kernel modules.
SYS_NICE	Raise process nice value (nice(2), setpriority(2)) and change the nice value for arbitrary processes.
SYS_PACCT	Use acct(2), switch process accounting on or off.
SYS_PTRACE	Trace arbitrary processes using ptrace(2).
SYS_RAWIO	Perform I/O port operations (iopl(2) and ioperm(2)).
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.
SYSLOG	Perform privileged syslog(2) operations.
WAKE_ALARM	Trigger something that will wake up the system.

Further reference information is available on the capabilities(7) - Linux man page, and in the Linux kernel source code.

Both flags support the value `ALL`, so to allow a container to use all capabilities except for `MKNOD`:

The `--cap-add` and `--cap-drop` flags accept capabilities to be specified with a `CAP_` prefix. The following examples are therefore equivalent:

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

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

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.

When you build an image from a Dockerfile, or when committing it, you can set a number of default parameters that take effect when the image starts up as a container. When you run an image, you can override those defaults using flags for the `docker run` command.

- Default entrypoint
- Default command and options
- Expose ports
- Environment variables
- Healthcheck
- User
- Working directory

Default command and options

The command syntax for `docker run` supports optionally specifying commands and arguments to the container's entrypoint, represented as `[COMMAND]` and `[ARG...]` in the following synopsis example:

This command is optional because whoever created the `IMAGE` may have already provided a default `COMMAND`, using the Dockerfile `CMD` instruction. When you run a container, 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`.

Default entrypoint

The entrypoint refers to the default executable that's invoked when you run a container. A container's entrypoint is defined using the Dockerfile `ENTRYPOINT` instruction. It's similar to specifying a default command because it specifies, but the difference is that you need to pass an explicit flag to override the entrypoint, whereas you can override default commands with positional arguments. The defines a container's default behavior, with the idea 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 as commands. But there are cases where you may want to run something else

inside the container. This is when overriding the default entrypoint at runtime comes in handy, using the `--entrypoint` flag for the `docker run` command.

The `--entrypoint` flag expects a string value, representing the name or path of the binary that you want to invoke when the container starts. The following example shows you how to run a Bash shell in a container that has been set up to automatically run some other binary (like `/usr/bin/redis-server`):

The following examples show how to pass additional parameters to the custom entrypoint, using the positional command arguments:

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

> Passing `--entrypoint` clears out any default command set on the image. That is, any `CMD` instruction in the Dockerfile used to build it.

Exposed ports

By default, when you run a container, none of the container's ports are exposed to the host. This means you won't be able to access any ports that the container might be listening on. To make a container's ports accessible from the host, you need to publish the ports.

You can start the container with the `-P` or `-p` flags to expose its ports:

- The `-P` (or `--publish-all`) flag publishes all the exposed ports to the host. Docker binds each exposed port to a random port on the host.

  The `-P` flag only publishes port numbers that are explicitly flagged as exposed, either using the Dockerfile `EXPOSE` instruction or the `--expose` flag for the `docker run` command.

- The `-p` (or `--publish`) flag lets you explicitly map a single port or range of ports in the container to the host.

The port number inside the container (where the service listens) doesn't need to match the port number published on the outside of the container (where clients connect). For example, inside the container an HTTP service might be listening on port 80. 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 the `docker port` command.

Environment variables

Docker automatically sets some environment variables when creating a Linux container. Docker doesn't 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`

Variable	Value
`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, you can set any environment variable in the container by using one or more `-e` flags. You can even override the variables mentioned above, or variables defined using a Dockerfile `ENV` instruction when building the image.

If you name an environment variable without specifying a value, the current value of the named variable on the host is propagated into the container's environment:

Healthchecks

The following flags for the `docker run` command let you control the parameters for container healthchecks:

Option	Description
`--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-retries countdown
`--health-start-interval`	Time between running the check during the start period
`--no-healthcheck`	Disable any container-specified `HEALTHCHECK`

Example:

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

User

The default user within a container is `root` (uid = 0). You can set a default user to run the first process with the Dockerfile `USER` instruction. When starting a container, you can override the `USER` instruction by passing the `-u` option.

The followings examples are all valid:

> If you pass a numeric user ID, it must be in the range of 0-2147483647. If you pass a username, the user must exist in the container.

Working directory

The default working directory for running binaries within a container is the root directory (`/`). The default working directory of an image is set using the Dockerfile `WORKDIR` command. You can override the default working directory for an image using the `-w` (or `--workdir`) flag for the `docker run` command:

If the directory doesn't already exist in the container, it's created.

Source: https://docs.docker.com/engine/reference/run/#entrypoint-default-command-to-execute-at-runtime