*Estimated reading time: 74 minutes*

# Dockerfile reference

Docker can build images automatically by reading the instructions from a `Dockerfile`. A `Dockerfile` is a text document that contains all the commands a user could call on the command line to assemble an image. Using `docker build` users can create an automated build that executes several command-line instructions in succession.

This page describes the commands you can use in a `Dockerfile`. When you are done reading this page, refer to the `Dockerfile` Best Practices (https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/) for a tip-oriented guide.

## Usage

The `docker build` (https://docs.docker.com/engine/reference/commandline/build/) command builds an image from a `Dockerfile` and a *context*. The build's context is the set of files at a specified location `PATH` or `URL`. The `PATH` is a directory on your local filesystem. The `URL` is a Git repository location.

A context is processed recursively. So, a `PATH` includes any subdirectories and the `URL` includes the repository and its submodules. This example shows a build command that uses the current directory as context:

```
$ docker build .
Sending build context to Docker daemon  6.51 MB
...
```

The build is run by the Docker daemon, not by the CLI. The first thing a build process does is send the entire context (recursively) to the daemon. In most cases, it's best to start with an empty directory as context and keep your Dockerfile in that directory. Add only the files needed for building the Dockerfile.

> Warning: Do not use your root directory, `/`, as the `PATH` as it causes the build to transfer the entire contents of your hard drive to the Docker daemon.

To use a file in the build context, the `Dockerfile` refers to the file specified in an instruction, for example, a `COPY` instruction. To increase the build's performance, exclude files and directories by adding a `.dockerignore` file to the context directory. For information about how to create a `.dockerignore` file (/engine/reference/builder/#dockerignore-file) see the documentation on this page.

Traditionally, the `Dockerfile` is called `Dockerfile` and located in the root of the context. You use the `-f` flag with `docker build` to point to a Dockerfile anywhere in your file system.

```
$ docker build -f /path/to/a/Dockerfile .
```

You can specify a repository and tag at which to save the new image if the build succeeds:

```
$ docker build -t shykes/myapp .
```

To tag the image into multiple repositories after the build, add multiple `-t` parameters when you run the `build` command:

```
$ docker build -t shykes/myapp:1.0.2 -t shykes/myapp:latest .
```

Before the Docker daemon runs the instructions in the `Dockerfile`, it performs a preliminary validation of the `Dockerfile` and returns an error if the syntax is incorrect:

```
$ docker build -t test/myapp .
Sending build context to Docker daemon 2.048 kB
Error response from daemon: Unknown instruction: RUNCMD
```

The Docker daemon runs the instructions in the `Dockerfile` one-by-one, committing the result of each instruction to a new image if necessary, before finally outputting the ID of your new image. The Docker daemon will automatically clean up the context you sent.

Note that each instruction is run independently, and causes a new image to be created - so `RUN cd /tmp` will not have any effect on the next instructions.

Whenever possible, Docker will re-use the intermediate images (cache), to accelerate the `docker build` process significantly. This is indicated by the `Using cache` message in the console output. (For more information, see the Build cache section (https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#build-cache) in the `Dockerfile` best practices guide):

```
$ docker build -t svendowideit/ambassador .
Sending build context to Docker daemon 15.36 kB
Step 1/4 : FROM alpine:3.2
 ---> 31f630c65071
Step 2/4 : MAINTAINER SvenDowideit@home.org.au
 ---> Using cache
 ---> 2a1c91448f5f
Step 3/4 : RUN apk update &&      apk add socat &&        rm -r /var
/cache/
 ---> Using cache
 ---> 21ed6e7fbb73
Step 4/4 : CMD env | grep _TCP= | (sed 's/.*_PORT_\([0-9]*\)_TCP=tcp
:\/\/\(.*\):\(.*\)/socat -t 100000000 TCP4-LISTEN:\1,fork,reuseaddr
TCP4:\2:\3 \&' && echo wait) | sh
 ---> Using cache
 ---> 7ea8aef582cc
Successfully built 7ea8aef582cc
```

Build cache is only used from images that have a local parent chain. This means that these images were created by previous builds or the whole chain of images was loaded with `docker load`. If you wish to use build cache of a specific image you can specify it with `--cache-from` option. Images specified with `--cache-from` do not need to have a parent chain and may be pulled from other registries.

When you're done with your build, you're ready to look into *Pushing a repository to its registry* (https://docs.docker.com/engine/tutorials/dockerrepos/#/contributing-to-docker-hub).

# BuildKit

Starting with version 18.09, Docker supports a new backend for executing your builds that is provided by the moby/buildkit (https://github.com/moby/buildkit) project. The BuildKit backend provides many benefits compared to the old implementation. For example, BuildKit can:

- Detect and skip executing unused build stages
- Parallelize building independent build stages
- Incrementally transfer only the changed files in your build context between builds
- Detect and skip transferring unused files in your build context
- Use external Dockerfile implementations with many new features
- Avoid side-effects with rest of the API (intermediate images and containers)
- Prioritize your build cache for automatic pruning

To use the BuildKit backend, you need to set an environment variable `DOCKER_BUILDKIT=1` on the CLI before invoking `docker build`.

To learn about the experimental Dockerfile syntax available to BuildKit-based builds refer to the documentation in the BuildKit repository (https://github.com/moby/buildkit/blob/master/frontend/dockerfile/docs/experimental.md).

# Format

Here is the format of the `Dockerfile`:

```
# Comment
INSTRUCTION arguments
```

The instruction is not case-sensitive. However, convention is for them to be UPPERCASE to distinguish them from arguments more easily.

Docker runs instructions in a `Dockerfile` in order. A `Dockerfile` must start with a `FROM` instruction. The `FROM` instruction specifies the *Base Image* (https://docs.docker.com/engine/reference/glossary/#base-image) from which you are building. `FROM` may only be preceded by one or more `ARG` instructions, which declare arguments that are used in `FROM` lines in the `Dockerfile`.

Docker treats lines that *begin* with `#` as a comment, unless the line is a valid parser directive (/engine/reference/builder/#parser-directives). A `#` marker anywhere else in a line is treated as an argument. This allows statements like:

```
# Comment
RUN echo 'we are running some # of cool things'
```

Line continuation characters are not supported in comments.

# Parser directives

Parser directives are optional, and affect the way in which subsequent lines in a `Dockerfile` are handled. Parser directives do not add layers to the build, and will not be shown as a build step. Parser directives are written as a special type of comment in the form `# directive=value`. A single directive may only be used once.

Once a comment, empty line or builder instruction has been processed, Docker no longer looks for parser directives. Instead it treats anything formatted as a parser directive as a comment and does not attempt to validate if it might be a parser directive. Therefore, all parser directives must be at the very top of a `Dockerfile`.

Parser directives are not case-sensitive. However, convention is for them to be lowercase. Convention is also to include a blank line following any parser directives. Line continuation characters are not supported in parser directives.

Due to these rules, the following examples are all invalid:

Invalid due to line continuation:

```
# direc \
tive=value
```

Invalid due to appearing twice:

```
# directive=value1
# directive=value2

FROM ImageName
```

Treated as a comment due to appearing after a builder instruction:

```
FROM ImageName
# directive=value
```

Treated as a comment due to appearing after a comment which is not a parser directive:

```
# About my dockerfile
# directive=value
FROM ImageName
```

The unknown directive is treated as a comment due to not being recognized. In addition, the known directive is treated as a comment due to appearing after a comment which is not a parser directive.

```
# unknowndirective=value
# knowndirective=value
```

Non line-breaking whitespace is permitted in a parser directive. Hence, the following lines are all treated identically:

```
#directive=value
# directive =value
#       directive= value
# directive = value
#         dIrEcTiVe=value
```

The following parser directives are supported:

- `syntax`
- `escape`

# syntax

```
# syntax=[remote image reference]
```

For example:

```
# syntax=docker/dockerfile
# syntax=docker/dockerfile:1.0
# syntax=docker.io/docker/dockerfile:1
# syntax=docker/dockerfile:1.0.0-experimental
# syntax=example.com/user/repo:tag@sha256:abcdef...
```

This feature is only enabled if the BuildKit (/engine/reference/builder/#buildkit) backend is used.

The syntax directive defines the location of the Dockerfile builder that is used for building the current Dockerfile. The BuildKit backend allows to seamlessly use external implementations of builders that are distributed as Docker images and execute inside a container sandbox environment.

Custom Dockerfile implementation allows you to:

- Automatically get bugfixes without updating the daemon
- Make sure all users are using the same implementation to build your Dockerfile
- Use the latest features without updating the daemon
- Try out new experimental or third-party features

## Official releases

Docker distributes official versions of the images that can be used for building Dockerfiles under `docker/dockerfile` repository on Docker Hub. There are two channels where new images are released: stable and experimental.

Stable channel follows semantic versioning. For example:

- docker/dockerfile:1.0.0 - only allow immutable version 1.0.0
- docker/dockerfile:1.0 - allow versions 1.0.*
- docker/dockerfile:1 - allow versions 1.*.*
- docker/dockerfile:latest - latest release on stable channel

The experimental channel uses incremental versioning with the major and minor

component from the stable channel on the time of the release. For example:

- docker/dockerfile:1.0.1-experimental - only allow immutable version 1.0.1-experimental
- docker/dockerfile:1.0-experimental - latest experimental releases after 1.0
- docker/dockerfile:experimental - latest release on experimental channel

You should choose a channel that best fits your needs. If you only want bugfixes, you should use `docker/dockerfile:1.0`. If you want to benefit from experimental features, you should use the experimental channel. If you are using the experimental channel, newer releases may not be backwards compatible, so it is recommended to use an immutable full version variant.

For master builds and nightly feature releases refer to the description in the source repository (https://github.com/moby/buildkit/blob/master/README.md).

## escape

```
# escape=\ (backslash)
```

Or

```
# escape=‘ (backtick)
```

The `escape` directive sets the character used to escape characters in a `Dockerfile`. If not specified, the default escape character is `\`.

The escape character is used both to escape characters in a line, and to escape a newline. This allows a `Dockerfile` instruction to span multiple lines. Note that regardless of whether the `escape` parser directive is included in a `Dockerfile`, *escaping is not performed in a `RUN` command, except at the end of a line.*

Setting the escape character to `‘` is especially useful on `Windows`, where `\` is the directory path separator. `‘` is consistent with Windows PowerShell (https://technet.microsoft.com/en-us/library/hh847755.aspx).

Consider the following example which would fail in a non-obvious way on `Windows`. The second `\` at the end of the second line would be interpreted as an escape for the newline, instead of a target of the escape from the first `\`.

Similarly, the `\` at the end of the third line would, assuming it was actually handled as an instruction, cause it be treated as a line continuation. The result of this dockerfile is that second and third lines are considered a single instruction:

```
FROM microsoft/nanoserver
COPY testfile.txt c:\\
RUN dir c:\
```

Results in:

```
PS C:\John> docker build -t cmd .
Sending build context to Docker daemon 3.072 kB
Step 1/2 : FROM microsoft/nanoserver
 ---> 22738ff49c6d
Step 2/2 : COPY testfile.txt c:\RUN dir c:
GetFileAttributesEx c:RUN: The system cannot find the file specified
.
PS C:\John>
```

One solution to the above would be to use `/` as the target of both the `COPY` instruction, and `dir`. However, this syntax is, at best, confusing as it is not natural for paths on `Windows`, and at worst, error prone as not all commands on `Windows` support `/` as the path separator.

By adding the `escape` parser directive, the following `Dockerfile` succeeds as expected with the use of natural platform semantics for file paths on `Windows`:

```
# escape=‘

FROM microsoft/nanoserver
COPY testfile.txt c:\
RUN dir c:\
```

Results in:

```
PS C:\John> docker build -t succeeds --no-cache=true .
Sending build context to Docker daemon 3.072 kB
Step 1/3 : FROM microsoft/nanoserver
 ---> 22738ff49c6d
Step 2/3 : COPY testfile.txt c:\
 ---> 96655de338de
Removing intermediate container 4db9acbb1682
Step 3/3 : RUN dir c:\
 ---> Running in a2c157f842f5
 Volume in drive C has no label.
 Volume Serial Number is 7E6D-E0F7

 Directory of c:\

10/05/2016  05:04 PM             1,894 License.txt
10/05/2016  02:22 PM    <DIR>          Program Files
10/05/2016  02:14 PM    <DIR>          Program Files (x86)
10/28/2016  11:18 AM                62 testfile.txt
10/28/2016  11:20 AM    <DIR>          Users
10/28/2016  11:20 AM    <DIR>          Windows
           2 File(s)          1,956 bytes
           4 Dir(s)  21,259,096,064 bytes free
 ---> 01c7f3bef04f
Removing intermediate container a2c157f842f5
Successfully built 01c7f3bef04f
PS C:\John>
```

# Environment replacement

Environment variables (declared with the `ENV` statement (/engine/reference/builder/#env)) can also be used in certain instructions as variables to be interpreted by the `Dockerfile`. Escapes are also handled for including variable-like syntax into a statement literally.

Environment variables are notated in the `Dockerfile` either with `$variable_name` or `${variable_name}`. They are treated equivalently and the brace syntax is typically used to address issues with variable names with no whitespace, like `${foo}_bar`.

The `${variable_name}` syntax also supports a few of the standard `bash` modifiers as specified below:

- `${variable:-word}` indicates that if `variable` is set then the result will be that value. If `variable` is not set then `word` will be the result.

- `${variable:+word}` indicates that if `variable` is set then `word` will be the result, otherwise the result is the empty string.

In all cases, `word` can be any string, including additional environment variables.

Escaping is possible by adding a `\` before the variable: `\$foo` or `\${foo}`, for example, will translate to `$foo` and `${foo}` literals respectively.

Example (parsed representation is displayed after the `#`):

```
FROM busybox
ENV foo /bar
WORKDIR ${foo}   # WORKDIR /bar
ADD . $foo       # ADD . /bar
COPY \$foo /quux # COPY $foo /quux
```

Environment variables are supported by the following list of instructions in the `Dockerfile`:

- `ADD`
- `COPY`
- `ENV`
- `EXPOSE`
- `FROM`
- `LABEL`
- `STOPSIGNAL`
- `USER`
- `VOLUME`
- `WORKDIR`

as well as:

- `ONBUILD` (when combined with one of the supported instructions above)

> Note: prior to 1.4, `ONBUILD` instructions did NOT support environment variable, even when combined with any of the instructions listed above.

Environment variable substitution will use the same value for each variable throughout the entire instruction. In other words, in this example:

```
ENV abc=hello
ENV abc=bye def=$abc
ENV ghi=$abc
```

will result in `def` having a value of `hello`, not `bye`. However, `ghi` will have a value of `bye` because it is not part of the same instruction that set `abc` to `bye`.

# .dockerignore file

Before the docker CLI sends the context to the docker daemon, it looks for a file named `.dockerignore` in the root directory of the context. If this file exists, the CLI modifies the context to exclude files and directories that match patterns in it. This helps to avoid unnecessarily sending large or sensitive files and directories to the daemon and potentially adding them to images using `ADD` or `COPY`.

The CLI interprets the `.dockerignore` file as a newline-separated list of patterns similar to the file globs of Unix shells. For the purposes of matching, the root of the context is considered to be both the working and the root directory. For example, the patterns `/foo/bar` and `foo/bar` both exclude a file or directory named `bar` in the `foo` subdirectory of `PATH` or in the root of the git repository located at `URL`. Neither excludes anything else.

If a line in `.dockerignore` file starts with `#` in column 1, then this line is considered as a comment and is ignored before interpreted by the CLI.

Here is an example `.dockerignore` file:

```
# comment
*/temp*
*/*/temp*
temp?
```

This file causes the following build behavior:

| Rule | Behavior |
| --- | --- |
| `# comment` | Ignored. |

| Rule | Behavior |
| --- | --- |
| `*/temp*` | Exclude files and directories whose names start with `temp` in any immediate subdirectory of the root. For example, the plain file `/somedir/temporary.txt` is excluded, as is the directory `/somedir/temp`. |
| `*/*/temp*` | Exclude files and directories starting with `temp` from any subdirectory that is two levels below the root. For example, `/somedir/subdir/temporary.txt` is excluded. |
| `temp?` | Exclude files and directories in the root directory whose names are a one-character extension of `temp`. For example, `/tempa` and `/tempb` are excluded. |

Matching is done using Go's filepath.Match (http://golang.org/pkg/path/filepath#Match) rules. A preprocessing step removes leading and trailing whitespace and eliminates `.` and `..` elements using Go's filepath.Clean (http://golang.org/pkg/path/filepath/#Clean). Lines that are blank after preprocessing are ignored.

Beyond Go's filepath.Match rules, Docker also supports a special wildcard string `**` that matches any number of directories (including zero). For example, `**/*.go` will exclude all files that end with `.go` that are found in all directories, including the root of the build context.

Lines starting with `!` (exclamation mark) can be used to make exceptions to exclusions. The following is an example `.dockerignore` file that uses this mechanism:

```
*.md
!README.md
```

All markdown files *except* `README.md` are excluded from the context.

The placement of `!` exception rules influences the behavior: the last line of the `.dockerignore` that matches a particular file determines whether it is included or excluded. Consider the following example:

```
*.md
!README*.md
README-secret.md
```

No markdown files are included in the context except README files other than `README-secret.md`.

Now consider this example:

```
*.md
README-secret.md
!README*.md
```

All of the README files are included. The middle line has no effect because `!README*.md` matches `README-secret.md` and comes last.

You can even use the `.dockerignore` file to exclude the `Dockerfile` and `.dockerignore` files. These files are still sent to the daemon because it needs them to do its job. But the `ADD` and `COPY` instructions do not copy them to the image.

Finally, you may want to specify which files to include in the context, rather than which to exclude. To achieve this, specify `*` as the first pattern, followed by one or more `!` exception patterns.

Note: For historical reasons, the pattern `.` is ignored.

# FROM

```
FROM <image> [AS <name>]
```

Or

```
FROM <image>[:<tag>] [AS <name>]
```

Or

```
FROM <image>[@<digest>] [AS <name>]
```

The `FROM` instruction initializes a new build stage and sets the *Base Image* (https://docs.docker.com/engine/reference/glossary/#base-image) for subsequent instructions. As such, a valid `Dockerfile` must start with a `FROM` instruction. The image can be any valid image – it is especially easy to start by pulling an image from the *Public Repositories* (https://docs.docker.com/engine/tutorials/dockerrepos/).

- `ARG` is the only instruction that may precede `FROM` in the `Dockerfile`. See Understand how ARG and FROM interact (/engine/reference/builder/#understand-how-arg-and-from-interact).
- `FROM` can appear multiple times within a single `Dockerfile` to create multiple images or use one build stage as a dependency for another. Simply make a note of the last image ID output by the commit before each new `FROM` instruction. Each `FROM` instruction clears any state created by previous instructions.
- Optionally a name can be given to a new build stage by adding `AS name` to the `FROM` instruction. The name can be used in subsequent `FROM` and `COPY --from=<name|index>` instructions to refer to the image built in this stage.
- The `tag` or `digest` values are optional. If you omit either of them, the builder assumes a `latest` tag by default. The builder returns an error if it cannot find the `tag` value.

## Understand how ARG and FROM interact

`FROM` instructions support variables that are declared by any `ARG` instructions that occur before the first `FROM`.

```
ARG  CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD  /code/run-app

FROM extras:${CODE_VERSION}
CMD  /code/run-extras
```

An `ARG` declared before a `FROM` is outside of a build stage, so it can't be used in any instruction after a `FROM`. To use the default value of an `ARG` declared before the first `FROM` use an `ARG` instruction without a value inside of a build stage:

```
ARG VERSION=latest
FROM busybox:$VERSION
ARG VERSION
RUN echo $VERSION > image_version
```

# RUN

RUN has 2 forms:

- `RUN <command>` (*shell* form, the command is run in a shell, which by default is `/bin/sh -c` on Linux or `cmd /S /C` on Windows)
- `RUN ["executable", "param1", "param2"]` (*exec* form)

The `RUN` instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the `Dockerfile`.

Layering `RUN` instructions and generating commits conforms to the core concepts of Docker where commits are cheap and containers can be created from any point in an image's history, much like source control.

The *exec* form makes it possible to avoid shell string munging, and to `RUN` commands using a base image that does not contain the specified shell executable.

The default shell for the *shell* form can be changed using the `SHELL` command.

In the *shell* form you can use a `\` (backslash) to continue a single RUN instruction onto the next line. For example, consider these two lines:

```
RUN /bin/bash -c 'source $HOME/.bashrc; \
echo $HOME'
```

Together they are equivalent to this single line:

```
RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'
```

> Note: To use a different shell, other than '/bin/sh', use the *exec* form passing in the desired shell. For example,
> `RUN ["/bin/bash", "-c", "echo hello"]`

> Note: The *exec* form is parsed as a JSON array, which means that you must use double-quotes (") around words not single-quotes (').

> Note: Unlike the *shell* form, the *exec* form does not invoke a command shell. This means that normal shell processing does not happen. For example, `RUN [ "echo", "$HOME" ]` will not do variable substitution on `$HOME`. If you want shell processing then either use the *shell* form or execute a shell directly, for example: `RUN [ "sh", "-c", "echo $HOME" ]`. When using the exec form and executing a shell directly, as in the case for the shell form, it is the shell that is doing the environment variable expansion, not docker.
>
> Note: In the *JSON* form, it is necessary to escape backslashes. This is particularly relevant on Windows where the backslash is the path separator. The following line would otherwise be treated as *shell* form due to not being valid JSON, and fail in an unexpected way: `RUN ["c:\windows\system32\tasklist.exe"]` The correct syntax for this example is: `RUN ["c:\\windows\\system32\\tasklist.exe"]`

The cache for `RUN` instructions isn't invalidated automatically during the next build. The cache for an instruction like `RUN apt-get dist-upgrade -y` will be reused during the next build. The cache for `RUN` instructions can be invalidated by using the `--no-cache` flag, for example `docker build --no-cache`.

See the `Dockerfile` Best Practices guide (https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#/build-cache) for more information.

The cache for `RUN` instructions can be invalidated by `ADD` instructions. See

below (/engine/reference/builder/#add) for details.

### Known issues (RUN)

- Issue 783 (https://github.com/docker/docker/issues/783) is about file permissions problems that can occur when using the AUFS file system. You might notice it during an attempt to `rm` a file, for example.

  For systems that have recent aufs version (i.e., `dirperm1` mount option can be set), docker will attempt to fix the issue automatically by mounting the layers with `dirperm1` option. More details on `dirperm1` option can be found at `aufs` man page (https://github.com/sfjro/aufs3-linux/tree/aufs3.18/Documentation/filesystems/aufs)

  If your system doesn't have support for `dirperm1`, the issue describes a workaround.

## CMD

The `CMD` instruction has three forms:

- `CMD ["executable","param1","param2"]` (*exec* form, this is the preferred form)
- `CMD ["param1","param2"]` (as *default parameters to ENTRYPOINT*)
- `CMD command param1 param2` (*shell* form)

There can only be one `CMD` instruction in a `Dockerfile`. If you list more than one `CMD` then only the last `CMD` will take effect.

The main purpose of a **`CMD`** is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an `ENTRYPOINT` instruction as well.

> Note: If `CMD` is used to provide default arguments for the `ENTRYPOINT` instruction, both the `CMD` and `ENTRYPOINT` instructions should be specified with the JSON array format.

> Note: The *exec* form is parsed as a JSON array, which means that you must use double-quotes (") around words not single-quotes (').

> Note: Unlike the *shell* form, the *exec* form does not invoke a command shell. This means that normal shell processing does not happen. For example, `CMD [ "echo", "$HOME" ]` will not do variable substitution on `$HOME`. If you want shell processing then either use the *shell* form or execute a shell directly, for example: `CMD [ "sh", "-c", "echo $HOME" ]`. When using the exec form and executing a shell directly, as in the case for the shell form, it is the shell that is doing the environment variable expansion, not docker.

When used in the shell or exec formats, the `CMD` instruction sets the command to be executed when running the image.

If you use the *shell* form of the `CMD`, then the `<command>` will execute in `/bin/sh -c`:

```
FROM ubuntu
CMD echo "This is a test." | wc -
```

If you want to run your `<command>` without a shell then you must express the command as a JSON array and give the full path to the executable. This array form is the preferred format of **`CMD`**. Any additional parameters must be individually expressed as strings in the array:

```
FROM ubuntu
CMD ["/usr/bin/wc","--help"]
```

If you would like your container to run the same executable every time, then you should consider using `ENTRYPOINT` in combination with `CMD`. See *ENTRYPOINT* (/engine/reference/builder/#entrypoint).

If the user specifies arguments to `docker run` then they will override the default specified in `CMD`.

> Note: Don't confuse `RUN` with `CMD`. `RUN` actually runs a command and commits the result; `CMD` does not execute anything at build time, but specifies the intended command for the image.

# LABEL

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

The `LABEL` instruction adds metadata to an image. A `LABEL` is a key-value pair. To include spaces within a `LABEL` value, use quotes and backslashes as you would in command-line parsing. A few usage examples:

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

An image can have more than one label. You can specify multiple labels on a single line. Prior to Docker 1.10, this decreased the size of the final image, but this is no longer the case. You may still choose to specify multiple labels in a single instruction, in one of the following two ways:

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

```
LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"
```

Labels included in base or parent images (images in the `FROM` line) are inherited by your image. If a label already exists but with a different value, the most-recently-applied value overrides any previously-set value.

To view an image's labels, use the `docker inspect` command.

```
"Labels": {
    "com.example.vendor": "ACME Incorporated"
    "com.example.label-with-value": "foo",
    "version": "1.0",
    "description": "This text illustrates that label-values can span
 multiple lines.",
    "multi.label1": "value1",
    "multi.label2": "value2",
    "other": "value3"
},
```

# MAINTAINER (deprecated)

```
MAINTAINER <name>
```

The `MAINTAINER` instruction sets the *Author* field of the generated images. The `LABEL` instruction is a much more flexible version of this and you should use it instead, as it enables setting any metadata you require, and can be viewed easily, for example with `docker inspect`. To set a label corresponding to the `MAINTAINER` field you could use:

```
LABEL maintainer="SvenDowideit@home.org.au"
```

This will then be visible from `docker inspect` with the other labels.

# EXPOSE

```
EXPOSE <port> [<port>/<protocol>...]
```

The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.

The `EXPOSE` instruction does not actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published. To actually publish the port when running the container, use the `-p` flag on `docker run` to publish and map one or more ports, or the `-P` flag to publish all exposed ports and map them to high-order ports.

By default, `EXPOSE` assumes TCP. You can also specify UDP:

```
EXPOSE 80/udp
```

To expose on both TCP and UDP, include two lines:

```
EXPOSE 80/tcp
EXPOSE 80/udp
```

In this case, if you use `-P` with `docker run`, the port will be exposed once for TCP and once for UDP. Remember that `-P` uses an ephemeral high-ordered host port on the host, so the port will not be the same for TCP and UDP.

Regardless of the `EXPOSE` settings, you can override them at runtime by using the `-p` flag. For example

```
docker run -p 80:80/tcp -p 80:80/udp ...
```

To set up port redirection on the host system, see using the -P flag (https://docs.docker.com/engine/reference/run/#expose-incoming-ports). The `docker network` command supports creating networks for communication among containers without the need to expose or publish specific ports, because the containers connected to the network can communicate with each other over any port. For detailed information, see the overview of this feature (https://docs.docker.com/engine/userguide/networking/)).

# ENV

```
ENV <key> <value>
ENV <key>=<value> ...
```

The `ENV` instruction sets the environment variable `<key>` to the value `<value>`. This value will be in the environment for all subsequent instructions in the build stage and can be replaced inline (/engine/reference/builder/#environment-replacement) in many as well.

The `ENV` instruction has two forms. The first form, `ENV <key> <value>`, will set a single variable to a value. The entire string after the first space will be treated as the `<value>` - including whitespace characters. The value will be interpreted for other environment variables, so quote characters will be removed if they are not escaped.

The second form, `ENV <key>=<value> ...`, allows for multiple variables to be set at one time. Notice that the second form uses the equals sign (=) in the syntax, while the first form does not. Like command line parsing, quotes and backslashes can be used to include spaces within values.

For example:

```
ENV myName="John Doe" myDog=Rex\ The\ Dog \
    myCat=fluffy
```

and

```
ENV myName John Doe
ENV myDog Rex The Dog
ENV myCat fluffy
```

will yield the same net results in the final image.

The environment variables set using `ENV` will persist when a container is run from the resulting image. You can view the values using `docker inspect`, and change them using `docker run --env <key>=<value>`.

> Note: Environment persistence can cause unexpected side effects. For example, setting `ENV DEBIAN_FRONTEND noninteractive` may confuse apt-get users on a Debian-based image. To set a value for a single command, use `RUN <key>=<value> <command>` .

# ADD

ADD has two forms:

- `ADD [--chown=<user>:<group>] <src>... <dest>`
- `ADD [--chown=<user>:<group>] ["<src>",... "<dest>"]` (this form is required for paths containing whitespace)

> Note: The `--chown` feature is only supported on Dockerfiles used to build Linux containers, and will not work on Windows containers. Since user and group ownership concepts do not translate between Linux and Windows, the use of `/etc/passwd` and `/etc/group` for translating user and group names to IDs restricts this feature to only be viable for Linux OS-based containers.

The `ADD` instruction copies new files, directories or remote file URLs from `<src>` and adds them to the filesystem of the image at the path `<dest>` .

Multiple `<src>` resources may be specified but if they are files or directories, their paths are interpreted as relative to the source of the context of the build.

Each `<src>` may contain wildcards and matching will be done using Go's filepath.Match (http://golang.org/pkg/path/filepath#Match) rules. For example:

```
ADD hom* /mydir/        # adds all files starting with "hom"
ADD hom?.txt /mydir/    # ? is replaced with any single character, e
.g., "home.txt"
```

The `<dest>` is an absolute path, or a path relative to `WORKDIR` , into which the source will be copied inside the destination container.

```
ADD test relativeDir/          # adds "test" to ‘WORKDIR‘/relativeDi
r/
ADD test /absoluteDir/         # adds "test" to /absoluteDir/
```

When adding files or directories that contain special characters (such as `[` and `]`), you need to escape those paths following the Golang rules to prevent them from being treated as a matching pattern. For example, to add a file named `arr[0].txt`, use the following;

```
ADD arr[[]0].txt /mydir/    # copy a file named "arr[0].txt" to /myd
ir/
```

All new files and directories are created with a UID and GID of 0, unless the optional `--chown` flag specifies a given username, groupname, or UID/GID combination to request specific ownership of the content added. The format of the `--chown` flag allows for either username and groupname strings or direct integer UID and GID in any combination. Providing a username without groupname or a UID without GID will use the same numeric UID as the GID. If a username or groupname is provided, the container's root filesystem `/etc/passwd` and `/etc/group` files will be used to perform the translation from name to integer UID or GID respectively. The following examples show valid definitions for the `--chown` flag:

```
ADD --chown=55:mygroup files* /somedir/
ADD --chown=bin files* /somedir/
ADD --chown=1 files* /somedir/
ADD --chown=10:11 files* /somedir/
```

If the container root filesystem does not contain either `/etc/passwd` or `/etc/group` files and either user or group names are used in the `--chown` flag, the build will fail on the `ADD` operation. Using numeric IDs requires no lookup and will not depend on container root filesystem content.

In the case where `<src>` is a remote file URL, the destination will have permissions of 600. If the remote file being retrieved has an HTTP `Last-Modified` header, the timestamp from that header will be used to set the

`mtime` on the destination file. However, like any other file processed during an `ADD`, `mtime` will not be included in the determination of whether or not the file has changed and the cache should be updated.

> Note: If you build by passing a `Dockerfile` through STDIN (`docker build - < somefile`), there is no build context, so the `Dockerfile` can only contain a URL based `ADD` instruction. You can also pass a compressed archive through STDIN: (`docker build - < archive.tar.gz`), the `Dockerfile` at the root of the archive and the rest of the archive will be used as the context of the build.

> Note: If your URL files are protected using authentication, you will need to use `RUN wget`, `RUN curl` or use another tool from within the container as the `ADD` instruction does not support authentication.

> Note: The first encountered `ADD` instruction will invalidate the cache for all following instructions from the Dockerfile if the contents of `<src>` have changed. This includes invalidating the cache for `RUN` instructions. See the `Dockerfile` Best Practices guide (https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#/build-cache) for more information.

`ADD` obeys the following rules:

- The `<src>` path must be inside the *context* of the build; you cannot `ADD ../something /something`, because the first step of a `docker build` is to send the context directory (and subdirectories) to the docker daemon.
- If `<src>` is a URL and `<dest>` does not end with a trailing slash, then a file is downloaded from the URL and copied to `<dest>`.
- If `<src>` is a URL and `<dest>` does end with a trailing slash, then the filename is inferred from the URL and the file is downloaded to `<dest>/<filename>`. For instance, `ADD http://example.com/foobar /` would create the file `/foobar`. The URL must have a nontrivial path so that an appropriate filename can be discovered in this case (`http://example.com` will not work).

- If `<src>` is a directory, the entire contents of the directory are copied, including filesystem metadata.

> Note: The directory itself is not copied, just its contents.

- If `<src>` is a *local* tar archive in a recognized compression format (identity, gzip, bzip2 or xz) then it is unpacked as a directory. Resources from *remote* URLs are not decompressed. When a directory is copied or unpacked, it has the same behavior as `tar -¡`, the result is the union of:

  1. Whatever existed at the destination path and
  2. The contents of the source tree, with conflicts resolved in favor of "2." on a file-by-file basis.

  > Note: Whether a file is identified as a recognized compression format or not is done solely based on the contents of the file, not the name of the file. For example, if an empty file happens to end with `.tar.gz` this will not be recognized as a compressed file and will not generate any kind of decompression error message, rather the file will simply be copied to the destination.

- If `<src>` is any other kind of file, it is copied individually along with its metadata. In this case, if `<dest>` ends with a trailing slash `/`, it will be considered a directory and the contents of `<src>` will be written at `<dest>/base(<src>)`.
- If multiple `<src>` resources are specified, either directly or due to the use of a wildcard, then `<dest>` must be a directory, and it must end with a slash `/`.
- If `<dest>` does not end with a trailing slash, it will be considered a regular file and the contents of `<src>` will be written at `<dest>`.
- If `<dest>` doesn't exist, it is created along with all missing directories in its path.

# COPY

COPY has two forms:

- `COPY [--chown=<user>:<group>] <src>... <dest>`

- `COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]` (this form is required for paths containing whitespace)

> Note: The `--chown` feature is only supported on Dockerfiles used to build Linux containers, and will not work on Windows containers. Since user and group ownership concepts do not translate between Linux and Windows, the use of `/etc/passwd` and `/etc/group` for translating user and group names to IDs restricts this feature to only be viable for Linux OS-based containers.

The `COPY` instruction copies new files or directories from `<src>` and adds them to the filesystem of the container at the path `<dest>` .

Multiple `<src>` resources may be specified but the paths of files and directories will be interpreted as relative to the source of the context of the build.

Each `<src>` may contain wildcards and matching will be done using Go's filepath.Match (http://golang.org/pkg/path/filepath#Match) rules. For example:

```
COPY hom* /mydir/        # adds all files starting with "hom"
COPY hom?.txt /mydir/    # ? is replaced with any single character,
e.g., "home.txt"
```

The `<dest>` is an absolute path, or a path relative to `WORKDIR` , into which the source will be copied inside the destination container.

```
COPY test relativeDir/   # adds "test" to ‘WORKDIR‘/relativeDir/
COPY test /absoluteDir/  # adds "test" to /absoluteDir/
```

When copying files or directories that contain special characters (such as `[` and `]` ), you need to escape those paths following the Golang rules to prevent them from being treated as a matching pattern. For example, to copy a file named `arr[0].txt` , use the following;

```
COPY arr[[]0].txt /mydir/    # copy a file named "arr[0].txt" to /my
dir/
```

All new files and directories are created with a UID and GID of 0, unless the optional `--chown` flag specifies a given username, groupname, or UID/GID combination to request specific ownership of the copied content. The format of the `--chown` flag allows for either username and groupname strings or direct integer UID and GID in any combination. Providing a username without groupname or a UID without GID will use the same numeric UID as the GID. If a username or groupname is provided, the container's root filesystem `/etc/passwd` and `/etc/group` files will be used to perform the translation from name to integer UID or GID respectively. The following examples show valid definitions for the `--chown` flag:

```
COPY --chown=55:mygroup files* /somedir/
COPY --chown=bin files* /somedir/
COPY --chown=1 files* /somedir/
COPY --chown=10:11 files* /somedir/
```

If the container root filesystem does not contain either `/etc/passwd` or `/etc/group` files and either user or group names are used in the `--chown` flag, the build will fail on the `COPY` operation. Using numeric IDs requires no lookup and will not depend on container root filesystem content.

> Note: If you build using STDIN ( `docker build - < somefile` ), there is no build context, so `COPY` can't be used.

Optionally `COPY` accepts a flag `--from=<name|index>` that can be used to set the source location to a previous build stage (created with `FROM .. AS <name>` ) that will be used instead of a build context sent by the user. The flag also accepts a numeric index assigned for all previous build stages started with `FROM` instruction. In case a build stage with a specified name can't be found an image with the same name is attempted to be used instead.

`COPY` obeys the following rules:

- The `<src>` path must be inside the *context* of the build; you cannot `COPY ../something /something` , because the first step of a `docker build` is to send the context directory (and subdirectories) to the docker daemon.
- If `<src>` is a directory, the entire contents of the directory are copied,

including filesystem metadata.

> Note: The directory itself is not copied, just its contents.

- If `<src>` is any other kind of file, it is copied individually along with its metadata. In this case, if `<dest>` ends with a trailing slash `/`, it will be considered a directory and the contents of `<src>` will be written at `<dest>/base(<src>)`.
- If multiple `<src>` resources are specified, either directly or due to the use of a wildcard, then `<dest>` must be a directory, and it must end with a slash `/`.
- If `<dest>` does not end with a trailing slash, it will be considered a regular file and the contents of `<src>` will be written at `<dest>`.
- If `<dest>` doesn't exist, it is created along with all missing directories in its path.

# ENTRYPOINT

ENTRYPOINT has two forms:

- `ENTRYPOINT ["executable", "param1", "param2"]` (*exec* form, preferred)
- `ENTRYPOINT command param1 param2` (*shell* form)

An `ENTRYPOINT` allows you to configure a container that will run as an executable.

For example, the following will start nginx with its default content, listening on port 80:

```
docker run -i -t --rm -p 80:80 nginx
```

Command line arguments to `docker run <image>` will be appended after all elements in an *exec* form `ENTRYPOINT`, and will override all elements specified using `CMD`. This allows arguments to be passed to the entry point, i.e., `docker run <image> -d` will pass the `-d` argument to the entry point. You can override the `ENTRYPOINT` instruction using the `docker run --entrypoint` flag.

The *shell* form prevents any `CMD` or `run` command line arguments from being used, but has the disadvantage that your `ENTRYPOINT` will be started as a subcommand of `/bin/sh -c`, which does not pass signals. This means that the executable will not be the container's `PID 1` - and will *not* receive Unix signals - so your executable will not receive a `SIGTERM` from `docker stop <container>`.

Only the last `ENTRYPOINT` instruction in the `Dockerfile` will have an effect.

## Exec form ENTRYPOINT example

You can use the *exec* form of `ENTRYPOINT` to set fairly stable default commands and arguments and then use either form of `CMD` to set additional defaults that are more likely to be changed.

```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

When you run the container, you can see that `top` is the only process:

```
$ docker run -it --rm --name test  top -H
top - 08:25:00 up  7:27,  0 users,  load average: 0.00, 0.01, 0.05
Threads:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zomb
ie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0
si,  0.0 st
KiB Mem:   2056668 total,  1616832 used,   439836 free,    99352 buf
fers
KiB Swap:  1441840 total,        0 used,  1441840 free.  1324440 cac
hed Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ C
OMMAND
    1 root      20   0   19744   2336   2080 R  0.0  0.1   0:00.04 t
op
```

To examine the result further, you can use `docker exec`:

```
$ docker exec -it test ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COM
MAND
root         1  2.6  0.1  19752  2352 ?        Ss+  08:24   0:00 top
 -b -H
root         7  0.0  0.1  15572  2164 ?        R+   08:25   0:00 ps
aux
```

And you can gracefully request `top` to shut down using `docker stop test`.

The following `Dockerfile` shows using the `ENTRYPOINT` to run Apache in the foreground (i.e., as `PID 1`):

```
FROM debian:stable
RUN apt-get update && apt-get install -y --force-yes apache2
EXPOSE 80 443
VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

If you need to write a starter script for a single executable, you can ensure that the final executable receives the Unix signals by using `exec` and `gosu` commands:

```
#!/usr/bin/env bash
set -e

if [ "$1" = 'postgres' ]; then
    chown -R postgres "$PGDATA"

    if [ -z "$(ls -A "$PGDATA")" ]; then
        gosu postgres initdb
    fi

    exec gosu postgres "$@"
fi

exec "$@"
```

Lastly, if you need to do some extra cleanup (or communicate with other containers) on shutdown, or are co-ordinating more than one executable, you may need to ensure that the `ENTRYPOINT` script receives the Unix signals, passes them on, and then does some more work:

```
#!/bin/sh
# Note: I've written this using sh so it works in the busybox contai
ner too

# USE the trap if you need to also do manual cleanup after the servi
ce is stopped,
#     or need to start multiple services in the one container
trap "echo TRAPed signal" HUP INT QUIT TERM

# start service in background here
/usr/sbin/apachectl start

echo "[hit enter key to exit] or run 'docker stop <container>'"
read

# stop service and clean up here
echo "stopping apache"
/usr/sbin/apachectl stop

echo "exited $0"
```

If you run this image with `docker run -it --rm -p 80:80 --name test apache`, you can then examine the container's processes with `docker exec`, or `docker top`, and then ask the script to stop Apache:

```
$ docker exec -it test ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COM
MAND
root         1  0.1  0.0   4448   692 ?        Ss+  00:42   0:00 /bi
n/sh /run.sh 123 cmd cmd2
root        19  0.0  0.2  71304  4440 ?        Ss   00:42   0:00 /us
r/sbin/apache2 -k start
www-data    20  0.2  0.2 360468  6004 ?        Sl   00:42   0:00 /us
r/sbin/apache2 -k start
www-data    21  0.2  0.2 360468  6000 ?        Sl   00:42   0:00 /us
r/sbin/apache2 -k start
root        81  0.0  0.1  15572  2140 ?        R+   00:44   0:00 ps
aux
$ docker top test
PID                 USER                COMMAND
10035               root                {run.sh} /bin/sh /run.sh 123
 cmd cmd2
10054               root                /usr/sbin/apache2 -k start
10055               33                  /usr/sbin/apache2 -k start
10056               33                  /usr/sbin/apache2 -k start
$ /usr/bin/time docker stop test
test
real    0m 0.27s
user    0m 0.03s
sys     0m 0.03s
```

> Note: you can override the `ENTRYPOINT` setting using `--entrypoint`, but this can only set the binary to *exec* (no `sh -c` will be used).

> Note: The *exec* form is parsed as a JSON array, which means that you must use double-quotes (“) around words not single-quotes (‘).

> Note: Unlike the *shell* form, the *exec* form does not invoke a command shell. This means that normal shell processing does not happen. For example, `ENTRYPOINT [ "echo", "$HOME" ]` will not do variable substitution on `$HOME`. If you want shell processing then either use the *shell* form or execute a shell directly, for example: `ENTRYPOINT [ "sh", "-c", "echo $HOME" ]`. When using the exec form and executing a shell directly, as in the case for the shell form, it is the shell that is doing the environment variable expansion, not docker.

## Shell form ENTRYPOINT example

You can specify a plain string for the `ENTRYPOINT` and it will execute in `/bin/sh -c`. This form will use shell processing to substitute shell environment variables, and will ignore any `CMD` or `docker run` command line arguments. To ensure that `docker stop` will signal any long running `ENTRYPOINT` executable correctly, you need to remember to start it with `exec`:

```
FROM ubuntu
ENTRYPOINT exec top -b
```

When you run this image, you'll see the single `PID 1` process:

```
$ docker run -it --rm --name test top
Mem: 1704520K used, 352148K free, 0K shrd, 0K buff, 140368121167873K
 cached
CPU:   5% usr   0% sys   0% nic  94% idle   0% io   0% irq   0% sirq
Load average: 0.08 0.03 0.05 2/98 6
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
    1     0 root     R     3164   0%   0% top -b
```

Which will exit cleanly on `docker stop`:

```
$ /usr/bin/time docker stop test
test
real    0m 0.20s
user    0m 0.02s
sys     0m 0.04s
```

If you forget to add `exec` to the beginning of your `ENTRYPOINT` :

```
FROM ubuntu
ENTRYPOINT top -b
CMD --ignored-param1
```

You can then run it (giving it a name for the next step):

```
$ docker run -it --name test top --ignored-param2
Mem: 1704184K used, 352484K free, 0K shrd, 0K buff, 140621524238337K
 cached
CPU:   9% usr   2% sys   0% nic  88% idle   0% io   0% irq   0% sirq
Load average: 0.01 0.02 0.05 2/101 7
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
    1     0 root     S     3168   0%   0% /bin/sh -c top -b cmd cmd2
    7     1 root     R     3164   0%   0% top -b
```

You can see from the output of `top` that the specified `ENTRYPOINT` is not `PID 1` .

If you then run `docker stop test` , the container will not exit cleanly - the `stop` command will be forced to send a `SIGKILL` after the timeout:

```
$ docker exec -it test ps aux
PID   USER     COMMAND
    1 root     /bin/sh -c top -b cmd cmd2
    7 root     top -b
    8 root     ps aux
$ /usr/bin/time docker stop test
test
real    0m 10.19s
user    0m 0.04s
sys     0m 0.03s
```

## Understand how CMD and ENTRYPOINT interact

Both `CMD` and `ENTRYPOINT` instructions define what command gets executed when running a container. There are few rules that describe their co-operation.

1. Dockerfile should specify at least one of `CMD` or `ENTRYPOINT` commands.

2. `ENTRYPOINT` should be defined when using the container as an executable.
3. `CMD` should be used as a way of defining default arguments for an `ENTRYPOINT` command or for executing an ad-hoc command in a container.
4. `CMD` will be overridden when running the container with alternative arguments.

The table below shows what command is executed for different `ENTRYPOINT` / `CMD` combinations:

| | No ENTRYPOINT | ENTRYPOINT exec_entry p1_entry | ENTRYPOINT [“exec_entry”, “p1_entry”] |
|---|---|---|---|
| No CMD | *error, not allowed* | /bin/sh -c exec_entry p1_entry | exec_entry p1_entry |
| CMD [“exec_cmd”, “p1_cmd”] | exec_cmd p1_cmd | /bin/sh -c exec_entry p1_entry | exec_entry p1_entry exec_cmd p1_cmd |
| CMD [“p1_cmd”, “p2_cmd”] | p1_cmd p2_cmd | /bin/sh -c exec_entry p1_entry | exec_entry p1_entry p1_cmd p2_cmd |
| CMD exec_cmd p1_cmd | /bin/sh -c exec_cmd p1_cmd | /bin/sh -c exec_entry p1_entry | exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd |

> Note: If `CMD` is defined from the base image, setting `ENTRYPOINT` will reset `CMD` to an empty value. In this scenario, `CMD` must be defined in the current image to have a value.

# VOLUME

```
VOLUME ["/data"]
```

The `VOLUME` instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers. The value can be a JSON array, `VOLUME ["/var/log/"]`, or a plain string with multiple arguments, such as `VOLUME /var/log` or `VOLUME /var/log /var/db`. For more information/examples and mounting instructions via the Docker client, refer to *Share Directories via Volumes* (https://docs.docker.com/engine/tutorials/dockervolumes/#/mount-a-host-directory-as-a-data-volume) documentation.

The `docker run` command initializes the newly created volume with any data that exists at the specified location within the base image. For example, consider the following Dockerfile snippet:

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

This Dockerfile results in an image that causes `docker run` to create a new mount point at `/myvol` and copy the `greeting` file into the newly created volume.

## Notes about specifying volumes

Keep the following things in mind about volumes in the `Dockerfile`.

- Volumes on Windows-based containers: When using Windows-based containers, the destination of a volume inside the container must be one of:
    - a non-existing or empty directory
    - a drive other than `C:`
- Changing the volume from within the Dockerfile: If any build steps change the data within the volume after it has been declared, those changes will be discarded.
- JSON formatting: The list is parsed as a JSON array. You must enclose words with double quotes ( `"` ) rather than single quotes ( `'` ).

- The host directory is declared at container run-time: The host directory (the mountpoint) is, by its nature, host-dependent. This is to preserve image portability, since a given host directory can't be guaranteed to be available on all hosts. For this reason, you can't mount a host directory from within the Dockerfile. The `VOLUME` instruction does not support specifying a `host-dir` parameter. You must specify the mountpoint when you create or run the container.

# USER

```
USER <user>[:<group>] or
USER <UID>[:<GID>]
```

The `USER` instruction sets the user name (or UID) and optionally the user group (or GID) to use when running the image and for any `RUN`, `CMD` and `ENTRYPOINT` instructions that follow it in the `Dockerfile`.

> Warning: When the user doesn't have a primary group then the image (or the next instructions) will be run with the `root` group.

> On Windows, the user must be created first if it's not a built-in account. This can be done with the `net user` command called as part of a Dockerfile.

```
FROM microsoft/windowsservercore
# Create Windows user in the container
RUN net user /add patrick
# Set it for subsequent commands
USER patrick
```

# WORKDIR

```
WORKDIR /path/to/workdir
```

The `WORKDIR` instruction sets the working directory for any `RUN`, `CMD`, `ENTRYPOINT`, `COPY` and `ADD` instructions that follow it in the `Dockerfile`. If the `WORKDIR` doesn't exist, it will be created even if it's not used in any subsequent `Dockerfile` instruction.

The `WORKDIR` instruction can be used multiple times in a `Dockerfile`. If a relative path is provided, it will be relative to the path of the previous `WORKDIR` instruction. For example:

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

The output of the final `pwd` command in this `Dockerfile` would be `/a/b/c`.

The `WORKDIR` instruction can resolve environment variables previously set using `ENV`. You can only use environment variables explicitly set in the `Dockerfile`. For example:

```
ENV DIRPATH /path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
```

The output of the final `pwd` command in this `Dockerfile` would be `/path/$DIRNAME`

# ARG

```
ARG <name>[=<default value>]
```

The `ARG` instruction defines a variable that users can pass at build-time to the builder with the `docker build` command using the `--build-arg <varname>=<value>` flag. If a user specifies a build argument that was not defined in the Dockerfile, the build outputs a warning.

```
[Warning] One or more build-args [foo] were not consumed.
```

A Dockerfile may include one or more `ARG` instructions. For example, the following is a valid Dockerfile:

```
FROM busybox
ARG user1
ARG buildno
...
```

> Warning: It is not recommended to use build-time variables for passing secrets like github keys, user credentials etc. Build-time variable values are visible to any user of the image with the `docker history` command.

## Default values

An `ARG` instruction can optionally include a default value:

```
FROM busybox
ARG user1=someuser
ARG buildno=1
...
```

If an `ARG` instruction has a default value and if there is no value passed at build-time, the builder uses the default.

## Scope

An `ARG` variable definition comes into effect from the line on which it is defined in the `Dockerfile` not from the argument's use on the command-line or elsewhere. For example, consider this Dockerfile:

```
1 FROM busybox
2 USER ${user:-some_user}
3 ARG user
4 USER $user
...
```

A user builds this file by calling:

```
$ docker build --build-arg user=what_user .
```

The `USER` at line 2 evaluates to `some_user` as the `user` variable is defined on the subsequent line 3. The `USER` at line 4 evaluates to `what_user` as `user` is defined and the `what_user` value was passed on the command line. Prior to its definition by an `ARG` instruction, any use of a variable results in an empty string.

An `ARG` instruction goes out of scope at the end of the build stage where it was defined. To use an arg in multiple stages, each stage must include the `ARG` instruction.

```
FROM busybox
ARG SETTINGS
RUN ./run/setup $SETTINGS

FROM busybox
ARG SETTINGS
RUN ./run/other $SETTINGS
```

## Using ARG variables

You can use an `ARG` or an `ENV` instruction to specify variables that are available to the `RUN` instruction. Environment variables defined using the `ENV` instruction always override an `ARG` instruction of the same name. Consider this Dockerfile with an `ENV` and `ARG` instruction.

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER v1.0.0
4 RUN echo $CONT_IMG_VER
```

Then, assume this image is built with this command:

```
$ docker build --build-arg CONT_IMG_VER=v2.0.1 .
```

In this case, the `RUN` instruction uses `v1.0.0` instead of the `ARG` setting passed by the user: `v2.0.1` This behavior is similar to a shell script where a locally scoped variable overrides the variables passed as arguments or inherited from environment, from its point of definition.

Using the example above but a different `ENV` specification you can create more useful interactions between `ARG` and `ENV` instructions:

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER ${CONT_IMG_VER:-v1.0.0}
4 RUN echo $CONT_IMG_VER
```

Unlike an `ARG` instruction, `ENV` values are always persisted in the built image. Consider a docker build without the `--build-arg` flag:

```
$ docker build .
```

Using this Dockerfile example, `CONT_IMG_VER` is still persisted in the image but its value would be `v1.0.0` as it is the default set in line 3 by the `ENV` instruction.

The variable expansion technique in this example allows you to pass arguments from the command line and persist them in the final image by leveraging the `ENV` instruction. Variable expansion is only supported for a limited set of Dockerfile instructions. (/engine/reference/builder/#environment-replacement)

## Predefined ARGs

Docker has a set of predefined `ARG` variables that you can use without a corresponding `ARG` instruction in the Dockerfile.

- `HTTP_PROXY`
- `http_proxy`
- `HTTPS_PROXY`
- `https_proxy`
- `FTP_PROXY`
- `ftp_proxy`
- `NO_PROXY`
- `no_proxy`

To use these, simply pass them on the command line using the flag:

```
--build-arg <varname>=<value>
```

By default, these pre-defined variables are excluded from the output of `docker history`. Excluding them reduces the risk of accidentally leaking sensitive authentication information in an `HTTP_PROXY` variable.

For example, consider building the following Dockerfile using `--build-arg HTTP_PROXY=http://user:pass@proxy.lon.example.com`

```
FROM ubuntu
RUN echo "Hello World"
```

In this case, the value of the `HTTP_PROXY` variable is not available in the `docker history` and is not cached. If you were to change location, and your proxy server changed to `http://user:pass@proxy.sfo.example.com`, a subsequent build does not result in a cache miss.

If you need to override this behaviour then you may do so by adding an `ARG` statement in the Dockerfile as follows:

```
FROM ubuntu
ARG HTTP_PROXY
RUN echo "Hello World"
```

When building this Dockerfile, the `HTTP_PROXY` is preserved in the `docker history`, and changing its value invalidates the build cache.

## Automatic platform ARGs in the global scope

This feature is only available when using the BuildKit (/engine/reference/builder/#buildkit) backend.

Docker predefines a set of `ARG` variables with information on the platform of the node performing the build (build platform) and on the platform of the resulting image (target platform). The target platform can be specified with the `--platform` flag on `docker build` .

The following `ARG` variables are set automatically:

- `TARGETPLATFORM` - platform of the build result. Eg `linux/amd64` , `linux/arm/v7` , `windows/amd64` .
- `TARGETOS` - OS component of TARGETPLATFORM
- `TARGETARCH` - architecture component of TARGETPLATFORM
- `TARGETVARIANT` - variant component of TARGETPLATFORM
- `BUILDPLATFORM` - platform of the node performing the build.
- `BUILDOS` - OS component of BUILDPLATFORM
- `BUILDARCH` - OS component of BUILDPLATFORM
- `BUILDVARIANT` - OS component of BUILDPLATFORM

These arguments are defined in the global scope so are not automatically available inside build stages or for your `RUN` commands. To expose one of these arguments inside the build stage redefine it without value.

For example:

```
FROM alpine
ARG TARGETPLATFORM
RUN echo "I'm building for $TARGETPLATFORM"
```

## Impact on build caching

`ARG` variables are not persisted into the built image as `ENV` variables are. However, `ARG` variables do impact the build cache in similar ways. If a Dockerfile defines an `ARG` variable whose value is different from a previous build, then a "cache miss" occurs upon its first usage, not its definition. In particular, all `RUN` instructions following an `ARG` instruction use the `ARG` variable implicitly (as an environment variable), thus can cause a cache miss. All predefined `ARG` variables are exempt from caching unless there is a matching `ARG` statement in the `Dockerfile` .

For example, consider these two Dockerfile:

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 RUN echo $CONT_IMG_VER
```

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 RUN echo hello
```

If you specify `--build-arg CONT_IMG_VER=<value>` on the command line, in both cases, the specification on line 2 does not cause a cache miss; line 3 does cause a cache miss. `ARG CONT_IMG_VER` causes the RUN line to be identified as the same as running `CONT_IMG_VER=<value>` echo hello, so if the `<value>` changes, we get a cache miss.

Consider another example under the same command line:

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER $CONT_IMG_VER
4 RUN echo $CONT_IMG_VER
```

In this example, the cache miss occurs on line 3. The miss happens because the variable's value in the `ENV` references the `ARG` variable and that variable is changed through the command line. In this example, the `ENV` command causes the image to include the value.

If an `ENV` instruction overrides an `ARG` instruction of the same name, like this Dockerfile:

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER hello
4 RUN echo $CONT_IMG_VER
```

Line 3 does not cause a cache miss because the value of `CONT_IMG_VER` is a constant ( `hello` ). As a result, the environment variables and values used on the `RUN` (line 4) doesn't change between builds.

# ONBUILD

```
ONBUILD [INSTRUCTION]
```

The `ONBUILD` instruction adds to the image a *trigger* instruction to be executed at a later time, when the image is used as the base for another build. The trigger will be executed in the context of the downstream build, as if it had been inserted immediately after the `FROM` instruction in the downstream `Dockerfile` .

Any build instruction can be registered as a trigger.

This is useful if you are building an image which will be used as a base to build other images, for example an application build environment or a daemon which may be customized with user-specific configuration.

For example, if your image is a reusable Python application builder, it will require application source code to be added in a particular directory, and it might require a build script to be called *after* that. You can't just call `ADD` and `RUN` now, because you don't yet have access to the application source code, and it will be different for each application build. You could simply provide application developers with a boilerplate `Dockerfile` to copy-paste into their application, but that is inefficient, error-prone and difficult to update because it mixes with application-specific code.

The solution is to use `ONBUILD` to register advance instructions to run later, during the next build stage.

Here's how it works:

1. When it encounters an `ONBUILD` instruction, the builder adds a trigger to the metadata of the image being built. The instruction does not otherwise affect the current build.
2. At the end of the build, a list of all triggers is stored in the image manifest, under the key `OnBuild` . They can be inspected with the `docker inspect` command.
3. Later the image may be used as a base for a new build, using the `FROM` instruction. As part of processing the `FROM` instruction, the downstream

builder looks for `ONBUILD` triggers, and executes them in the same order they were registered. If any of the triggers fail, the `FROM` instruction is aborted which in turn causes the build to fail. If all triggers succeed, the `FROM` instruction completes and the build continues as usual.

4. Triggers are cleared from the final image after being executed. In other words they are not inherited by "grand-children" builds.

For example you might add something like this:

```
[...]
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

> Warning: Chaining `ONBUILD` instructions using `ONBUILD ONBUILD` isn't allowed.

> Warning: The `ONBUILD` instruction may not trigger `FROM` or `MAINTAINER` instructions.

# STOPSIGNAL

```
STOPSIGNAL signal
```

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

# HEALTHCHECK

The `HEALTHCHECK` instruction has two forms:

- `HEALTHCHECK [OPTIONS] CMD command` (check container health by running a command inside the container)
- `HEALTHCHECK NONE` (disable any healthcheck inherited from the base image)

The `HEALTHCHECK` instruction tells Docker how to test a container to check that it is still working. This can detect cases such as a web server that is stuck in an infinite loop and unable to handle new connections, even though the server process is still running.

When a container has a healthcheck specified, it has a *health status* in addition to its normal status. This status is initially `starting`. Whenever a health check passes, it becomes `healthy` (whatever state it was previously in). After a certain number of consecutive failures, it becomes `unhealthy`.

The options that can appear before `CMD` are:

- `--interval=DURATION` (default: `30s`)
- `--timeout=DURATION` (default: `30s`)
- `--start-period=DURATION` (default: `0s`)
- `--retries=N` (default: `3`)

The health check will first run interval seconds after the container is started, and then again interval seconds after each previous check completes.

If a single run of the check takes longer than timeout seconds then the check is considered to have failed.

It takes retries consecutive failures of the health check for the container to be considered `unhealthy`.

start period provides initialization time for containers that need time to bootstrap. Probe failure during that period will not be counted towards the maximum number of retries. However, if a health check succeeds during the start period, the container is considered started and all consecutive failures will be counted towards the maximum number of retries.

There can only be one `HEALTHCHECK` instruction in a Dockerfile. If you list more than one then only the last `HEALTHCHECK` will take effect.

The command after the `CMD` keyword can be either a shell command (e.g. `HEALTHCHECK CMD /bin/check-running`) or an *exec* array (as with other Dockerfile commands; see e.g. `ENTRYPOINT` for details).

The command's exit status indicates the health status of the container. The possible values are:

- 0: success - the container is healthy and ready for use
- 1: unhealthy - the container is not working correctly

- 2: reserved - do not use this exit code

For example, to check every five minutes or so that a web-server is able to serve the site's main page within three seconds:

```
HEALTHCHECK --interval=5m --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

To help debug failing probes, any output text (UTF-8 encoded) that the command writes on stdout or stderr will be stored in the health status and can be queried with `docker inspect`. Such output should be kept short (only the first 4096 bytes are stored currently).

When the health status of a container changes, a `health_status` event is generated with the new status.

The `HEALTHCHECK` feature was added in Docker 1.12.

# SHELL

```
SHELL ["executable", "parameters"]
```

The `SHELL` instruction allows the default shell used for the *shell* form of commands to be overridden. The default shell on Linux is `["/bin/sh", "-c"]`, and on Windows is `["cmd", "/S", "/C"]`. The `SHELL` instruction *must* be written in JSON form in a Dockerfile.

The `SHELL` instruction is particularly useful on Windows where there are two commonly used and quite different native shells: `cmd` and `powershell`, as well as alternate shells available including `sh`.

The `SHELL` instruction can appear multiple times. Each `SHELL` instruction overrides all previous `SHELL` instructions, and affects all subsequent instructions. For example:

```
FROM microsoft/windowsservercore

# Executed as cmd /S /C echo default
RUN echo default

# Executed as cmd /S /C powershell -command Write-Host default
RUN powershell -command Write-Host default

# Executed as powershell -command Write-Host hello
SHELL ["powershell", "-command"]
RUN Write-Host hello

# Executed as cmd /S /C echo hello
SHELL ["cmd", "/S", "/C"]
RUN echo hello
```

The following instructions can be affected by the `SHELL` instruction when the *shell* form of them is used in a Dockerfile: `RUN`, `CMD` and `ENTRYPOINT`.

The following example is a common pattern found on Windows which can be streamlined by using the `SHELL` instruction:

```
...
RUN powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
...
```

The command invoked by docker will be:

```
cmd /S /C powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
```

This is inefficient for two reasons. First, there is an un-necessary cmd.exe command processor (aka shell) being invoked. Second, each `RUN` instruction in the *shell* form requires an extra `powershell -command` prefixing the command.

To make this more efficient, one of two mechanisms can be employed. One is to use the JSON form of the RUN command such as:

```
...
RUN ["powershell", "-command", "Execute-MyCmdlet", "-param1 \"c:\\fo
o.txt\""]
...
```

While the JSON form is unambiguous and does not use the un-necessary cmd.exe, it does require more verbosity through double-quoting and escaping. The alternate mechanism is to use the `SHELL` instruction and the *shell* form, making a more natural syntax for Windows users, especially when combined with the `escape` parser directive:

```
# escape=‘

FROM microsoft/nanoserver
SHELL ["powershell","-command"]
RUN New-Item -ItemType Directory C:\Example
ADD Execute-MyCmdlet.ps1 c:\example\
RUN c:\example\Execute-MyCmdlet -sample ’hello world’
```

Resulting in:

```
PS E:\docker\build\shell> docker build -t shell .
Sending build context to Docker daemon 4.096 kB
Step 1/5 : FROM microsoft/nanoserver
 ---> 22738ff49c6d
Step 2/5 : SHELL powershell -command
 ---> Running in 6fcdb6855ae2
 ---> 6331462d4300
Removing intermediate container 6fcdb6855ae2
Step 3/5 : RUN New-Item -ItemType Directory C:\Example
 ---> Running in d0eef8386e97


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----       10/28/2016  11:26 AM                Example


 ---> 3f2fbf1395d9
Removing intermediate container d0eef8386e97
Step 4/5 : ADD Execute-MyCmdlet.ps1 c:\example\
 ---> a955b2621c31
Removing intermediate container b825593d39fc
Step 5/5 : RUN c:\example\Execute-MyCmdlet 'hello world'
 ---> Running in be6d8e63fe75
hello world
 ---> 8e559e9bf424
Removing intermediate container be6d8e63fe75
Successfully built 8e559e9bf424
PS E:\docker\build\shell>
```

The `SHELL` instruction could also be used to modify the way in which a shell operates. For example, using `SHELL cmd /S /C /V:ON|OFF` on Windows, delayed environment variable expansion semantics could be modified.

The `SHELL` instruction can also be used on Linux should an alternate shell be required such as `zsh`, `csh`, `tcsh` and others.

The `SHELL` feature was added in Docker 1.12.

# External implementation features

This feature is only available when using the BuildKit (/engine/reference/builder/#buildkit) backend.

Docker build supports experimental features like cache mounts, build secrets and ssh forwarding that are enabled by using an external implementation of the builder with a syntax directive. To learn about these features, refer to the documentation in BuildKit repository (https://github.com/moby/buildkit/blob/master/frontend/dockerfile/docs/experimental.md).

# Dockerfile examples

Below you can see some examples of Dockerfile syntax. If you're interested in something more realistic, take a look at the list of Dockerization examples (https://docs.docker.com/engine/examples/).

```
# Nginx
#
# VERSION               0.0.1

FROM      ubuntu
LABEL Description="This image is used to start the foobar executable
" Vendor="ACME Products" Version="1.0"
RUN apt-get update && apt-get install -y inotify-tools nginx apache2
 openssh-server
```

```
# Firefox over VNC
#
# VERSION               0.3

FROM ubuntu

# Install vnc, xvfb in order to create a 'fake' display and firefox
RUN apt-get update && apt-get install -y x11vnc xvfb firefox
RUN mkdir ~/.vnc
# Setup a password
RUN x11vnc -storepasswd 1234 ~/.vnc/passwd
# Autostart firefox (might not be the best way, but it does the tric
k)
RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900
CMD    ["x11vnc", "-forever", "-usepw", "-create"]



# Multiple images example
#
# VERSION               0.1

FROM ubuntu
RUN echo foo > bar
# Will output something like ===> 907ad6c2736f

FROM ubuntu
RUN echo moo > oink
# Will output something like ===> 695d7793cbe4

# You'll now have two images, 907ad6c2736f with /bar, and 695d7793cb
e4 with
# /oink.
```

builder (https://docs.docker.com/glossary/?term=builder), docker (https://docs.docker.com/glossary/?term=docker), Dockerfile (https://docs.docker.com/glossary/?term=Dockerfile), automation (https://docs.docker.com/glossary/?term=automation), image creation (https://docs.docker.com/glossary/?term=image creation)

# Best practices for writing Dockerfiles

*Estimated reading time: 26 minutes*

This document covers recommended best practices and methods for building efficient images.

Docker builds images automatically by reading the instructions from a `Dockerfile` -- a text file that contains all commands, in order, needed to build a given image. A `Dockerfile` adheres to a specific format and set of instructions which you can find at Dockerfile reference (https://docs.docker.com/engine/reference/builder/).

A Docker image consists of read-only layers each of which represents a Dockerfile instruction. The layers are stacked and each one is a delta of the changes from the previous layer. Consider this `Dockerfile` :

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Each instruction creates one layer:

- `FROM` creates a layer from the `ubuntu:15.04` Docker image.
- `COPY` adds files from your Docker client's current directory.
- `RUN` builds your application with `make` .
- `CMD` specifies what command to run within the container.

When you run an image and generate a container, you add a new *writable layer* (the "container layer") on top of the underlying layers. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.

For more on image layers (and how Docker builds and stores images), see About storage drivers (https://docs.docker.com/storage/storagedriver/).

## General guidelines and recommendations

### Create ephemeral containers

The image defined by your `Dockerfile` should generate containers that are as ephemeral as possible. By "ephemeral", we mean that the container can be stopped and destroyed, then rebuilt and replaced with an absolute minimum set up and configuration.

Refer to Processes (https://12factor.net/processes) under *The Twelve-factor App* methodology to get a feel for the motivations of running containers in such a stateless fashion.

### Understand build context

When you issue a `docker build` command, the current working directory is called the *build context*. By default, the Dockerfile is assumed to be located here, but you can specify a different location with the file flag ( `-f` ). Regardless of where the `Dockerfile` actually lives, all recursive contents of files and directories in the current directory are sent to the Docker daemon as the build context.

Build context example

Create a directory for the build context and `cd` into it. Write "hello" into a text file named `hello` and create a Dockerfile that runs `cat` on it. Build the image from within the build context ( `.` ):

```
mkdir myproject && cd myproject
echo "hello" > hello
echo -e "FROM busybox\nCOPY /hello /\nRUN cat /hello" > Dockerfile
docker build -t helloapp:v1 .
```

Move `Dockerfile` and `hello` into separate directories and build a second version of the image (without relying on cache from the last build). Use `-f` to point to the Dockerfile and specify the directory of the build context:

```
mkdir -p dockerfiles context
mv Dockerfile dockerfiles && mv hello context
docker build --no-cache -t helloapp:v2 -f dockerfiles/Dockerfile context
```

Inadvertently including files that are not necessary for building an image results in a larger build context and larger image size. This can increase the time to build the image, time to pull and push it, and the container runtime size. To see how big your build context is, look for a message like this when building your `Dockerfile` :

```
Sending build context to Docker daemon  187.8MB
```

## Pipe Dockerfile through `stdin`

Docker 17.05 added the ability to build images by piping `Dockerfile` through `stdin` with a *local or remote build-context*. In earlier versions, building an image with a `Dockerfile` from `stdin` did not send the build-context.

Docker 17.04 and lower

```
docker build -t foo -<<EOF
FROM busybox
RUN echo "hello world"
EOF
```

Docker 17.05 and higher (local build-context)

```
docker build -t foo . -f-<<EOF
FROM busybox
RUN echo "hello world"
COPY . /my-copied-files
EOF
```

Docker 17.05 and higher (remote build-context)

```
docker build -t foo https://github.com/thajeztah/pgadmin4-docker.git -f-<<EOF
FROM busybox
COPY LICENSE config_distro.py /usr/local/lib/python2.7/site-packages/pgadmin4/
EOF
```

## Exclude with .dockerignore

To exclude files not relevant to the build (without restructuring your source repository) use a `.dockerignore` file. This file supports exclusion patterns similar to `.gitignore` files. For information on creating one, see the .dockerignore file (https://docs.docker.com/engine/reference/builder/#dockerignore-file).

## Use multi-stage builds

Multi-stage builds (https://docs.docker.com/develop/develop-images/multistage-build/) (in Docker 17.05 (https://docs.docker.com/release-notes/docker-ce/#17050-ce-2017-05-04) or higher) allow you to drastically reduce the size of your final image, without struggling to reduce the number of intermediate layers and files.

Because an image is built during the final stage of the build process, you can minimize image layers by leveraging build cache (/develop/develop-images/dockerfile_best-practices/#leverage-build-cache).

For example, if your build contains several layers, you can order them from the less frequently changed (to ensure the build cache is reusable) to the more frequently changed:

- Install tools you need to build your application
- Install or update library dependencies
- Generate your application

A Dockerfile for a Go application could look like:

```
FROM golang:1.9.2-alpine3.6 AS build

# Install tools required for project
# Run ‘docker build --no-cache .‘ to update dependencies
RUN apk add --no-cache git
RUN go get github.com/golang/dep/cmd/dep

# List project dependencies with Gopkg.toml and Gopkg.lock
# These layers are only re-built when Gopkg files are updated
COPY Gopkg.lock Gopkg.toml /go/src/project/
WORKDIR /go/src/project/
# Install library dependencies
RUN dep ensure -vendor-only

# Copy the entire project and build it
# This layer is rebuilt when a file changes in the project directory
COPY . /go/src/project/
RUN go build -o /bin/project

# This results in a single layer image
FROM scratch
COPY --from=build /bin/project /bin/project
ENTRYPOINT ["/bin/project"]
CMD ["--help"]
```

## Don’t install unnecessary packages

To reduce complexity, dependencies, file sizes, and build times, avoid installing extra or unnecessary packages just because they might be “nice to have.” For example, you don’t need to include a text editor in a database image.

## Decouple applications

Each container should have only one concern. Decoupling applications into multiple containers makes it easier to scale horizontally and reuse containers. For instance, a web application stack might consist of three separate containers, each with its own unique image, to manage the web application, database, and an in-memory cache in a decoupled manner.

Limiting each container to one process is a good rule of thumb, but it is not a hard and fast rule. For example, not only can containers be spawned with an init process (https://docs.docker.com/engine/reference/run/#specify-an-init-process), some programs might spawn additional processes of their own accord. For instance, Celery (http://www.celeryproject.org/) can spawn multiple worker processes, and Apache (https://httpd.apache.org/) can create one process per request.

Use your best judgment to keep containers as clean and modular as possible. If containers depend on each other, you can use Docker container networks (https://docs.docker.com/engine/userguide/networking/) to ensure that these containers can communicate.

## Minimize the number of layers

In older versions of Docker, it was important that you minimized the number of layers in your images to ensure they were performant. The following features were added to reduce this limitation:

- In Docker 1.10 and higher, only the instructions `RUN`, `COPY`, `ADD` create layers. Other instructions create temporary intermediate images, and do not directly increase the size of the build.
- In Docker 17.05 and higher, you can do multi-stage builds (https://docs.docker.com/develop/develop-images/multistage-build/) and only copy the artifacts you need into the final image. This allows you to include tools and debug information in your intermediate build stages without increasing the size of the final image.

## Sort multi-line arguments

Whenever possible, ease later changes by sorting multi-line arguments alphanumerically. This helps to avoid duplication of packages and make the list much easier to update. This also makes PRs a lot easier to read and review. Adding a space before a backslash ( `\` ) helps as well.

Here's an example from the `buildpack-deps` image (https://github.com/docker-library/buildpack-deps):

```
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
  git \
  mercurial \
  subversion
```

## Leverage build cache

When building an image, Docker steps through the instructions in your `Dockerfile`, executing each in the order specified. As each instruction is examined, Docker looks for an existing image in its cache that it can reuse, rather than creating a new (duplicate) image.

If you do not want to use the cache at all, you can use the `--no-cache=true` option on the `docker build` command. However, if you do let Docker use its cache, it is important to understand when it can, and cannot, find a matching image. The basic rules that Docker follows are outlined below:

- Starting with a parent image that is already in the cache, the next instruction is compared against all child images derived from that base image to see if one of them was built using the exact same instruction. If not, the cache is invalidated.

- In most cases, simply comparing the instruction in the `Dockerfile` with one of the child images is sufficient. However, certain instructions require more examination and explanation.
- For the `ADD` and `COPY` instructions, the contents of the file(s) in the image are examined and a checksum is calculated for each file. The last-modified and last-accessed times of the file(s) are not considered in these checksums. During the cache lookup, the checksum is compared against the checksum in the existing images. If anything has changed in the file(s), such as the contents and metadata, then the cache is invalidated.
- Aside from the `ADD` and `COPY` commands, cache checking does not look at the files in the container to determine a cache match. For example, when processing a `RUN apt-get -y update` command the files updated in the container are not examined to determine if a cache hit exists. In that case just the command string itself is used to find a match.

Once the cache is invalidated, all subsequent `Dockerfile` commands generate new images and the cache is not used.

# Dockerfile instructions

These recommendations are designed to help you create an efficient and maintainable `Dockerfile` .

## FROM

Dockerfile reference for the FROM instruction (https://docs.docker.com/engine/reference/builder/#from)

Whenever possible, use current official images as the basis for your images. We recommend the Alpine image (https://hub.docker.com/_/alpine/) as it is tightly controlled and small in size (currently under 5 MB), while still being a full Linux distribution.

## LABEL

Understanding object labels (https://docs.docker.com/config/labels-custom-metadata/)

You can add labels to your image to help organize images by project, record licensing information, to aid in automation, or for other reasons. For each label, add a line beginning with `LABEL` and with one or more key-value pairs. The following examples show the different acceptable formats. Explanatory comments are included inline.

> Strings with spaces must be quoted or the spaces must be escaped. Inner quote characters ( `"` ), must also be escaped.

```
# Set one or more individual labels
LABEL com.example.version="0.0.1-beta"
LABEL vendor1="ACME Incorporated"
LABEL vendor2=ZENITH\ Incorporated
LABEL com.example.release-date="2015-02-12"
LABEL com.example.version.is-production=""
```

An image can have more than one label. Prior to Docker 1.10, it was recommended to combine all labels into a single `LABEL` instruction, to prevent extra layers from being created. This is no longer necessary, but combining labels is still supported.

```
# Set multiple labels on one line
LABEL com.example.version="0.0.1-beta" com.example.release-date="2015-02-12"
```

The above can also be written as:

```
# Set multiple labels at once, using line-continuation characters to break long lines
LABEL vendor=ACME\ Incorporated \
      com.example.is-beta= \
      com.example.is-production="" \
      com.example.version="0.0.1-beta" \
      com.example.release-date="2015-02-12"
```

See Understanding object labels (https://docs.docker.com/config/labels-custom-metadata/) for guidelines about acceptable label keys and values. For information about querying labels, refer to the items related to filtering in Managing labels on objects (https://docs.docker.com/config/labels-custom-metadata/#managing-labels-on-objects). See also LABEL (https://docs.docker.com/engine/reference/builder/#label) in the Dockerfile reference.

## RUN

Dockerfile reference for the RUN instruction (https://docs.docker.com/engine/reference/builder/#run)

Split long or complex `RUN` statements on multiple lines separated with backslashes to make your `Dockerfile` more readable, understandable, and maintainable.

### APT-GET

Probably the most common use-case for `RUN` is an application of `apt-get`. Because it installs packages, the `RUN apt-get` command has several gotchas to look out for.

Avoid `RUN apt-get upgrade` and `dist-upgrade`, as many of the "essential" packages from the parent images cannot upgrade inside an unprivileged container (https://docs.docker.com/engine/reference/run/#security-configuration). If a package contained in the parent image is out-of-date, contact its maintainers. If you know there is a particular package, `foo`, that needs to be updated, use `apt-get install -y foo` to update automatically.

Always combine `RUN apt-get update` with `apt-get install` in the same `RUN` statement. For example:

```
RUN apt-get update && apt-get install -y \
    package-bar \
    package-baz \
    package-foo
```

Using `apt-get update` alone in a `RUN` statement causes caching issues and subsequent `apt-get install` instructions fail. For example, say you have a Dockerfile:

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y curl
```

After building the image, all layers are in the Docker cache. Suppose you later modify `apt-get install` by adding extra package:

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y curl nginx
```

Docker sees the initial and modified instructions as identical and reuses the cache from previous steps. As a result the `apt-get update` is *not* executed because the build uses the cached version. Because the `apt-get update` is not run, your build can potentially get an outdated version of the `curl` and `nginx` packages.

Using `RUN apt-get update && apt-get install -y` ensures your Dockerfile installs the latest package versions with no further coding or manual intervention. This technique is known as "cache busting". You can also achieve cache-busting by specifying a package version. This is known as version pinning, for example:

```
RUN apt-get update && apt-get install -y \
    package-bar \
    package-baz \
    package-foo=1.3.*
```

Version pinning forces the build to retrieve a particular version regardless of what's in the cache. This technique can also reduce failures due to unanticipated changes in required packages.

Below is a well-formed `RUN` instruction that demonstrates all the `apt-get` recommendations.

```
RUN apt-get update && apt-get install -y \
    aufs-tools \
    automake \
    build-essential \
    curl \
    dpkg-sig \
    libcap-dev \
    libsqlite3-dev \
    mercurial \
    reprepro \
    ruby1.9.1 \
    ruby1.9.1-dev \
    s3cmd=1.1.* \
 && rm -rf /var/lib/apt/lists/*
```

The `s3cmd` argument specifies a version `1.1.*` . If the image previously used an older version, specifying the new one causes a cache bust of `apt-get update` and ensures the installation of the new version. Listing packages on each line can also prevent mistakes in package duplication.

In addition, when you clean up the apt cache by removing `/var/lib/apt/lists` it reduces the image size, since the apt cache is not stored in a layer. Since the `RUN` statement starts with `apt-get update` , the package cache is always refreshed prior to `apt-get install` .

> Official Debian and Ubuntu images automatically run `apt-get clean` (https://github.com/moby/moby/blob/03e2923e42446dbb830c654d0eec323a0b4ef02a/contrib/mkimage/debootstrap#L82-L105), so explicit invocation is not required.

### USING PIPES

Some `RUN` commands depend on the ability to pipe the output of one command into another, using the pipe character ( `|` ), as in the following example:

```
RUN wget -O - https://some.site | wc -l > /number
```

Docker executes these commands using the `/bin/sh -c` interpreter, which only evaluates the exit code of the last operation in the pipe to determine success. In the example above this build step succeeds and produces a new image so long as the `wc -l` command succeeds, even if the `wget` command fails.

If you want the command to fail due to an error at any stage in the pipe, prepend `set -o pipefail &&` to ensure that an unexpected error prevents the build from inadvertently succeeding. For example:

```
RUN set -o pipefail && wget -O - https://some.site | wc -l > /number
```

> ✔ Not all shells support the `-o pipefail` option.
>
> In cases such as the `dash` shell on Debian-based images, consider using the *exec* form of `RUN` to explicitly choose a shell that does support the `pipefail` option. For example:
>
> ```
> RUN ["/bin/bash", "-c", "set -o pipefail && wget -O - https://some.site | wc -l > /number"]
> ```

## CMD

Dockerfile reference for the CMD instruction (https://docs.docker.com/engine/reference/builder/#cmd)

The `CMD` instruction should be used to run the software contained by your image, along with any arguments. `CMD` should almost always be used in the form of `CMD ["executable", "param1", "param2"…]` . Thus, if the image is for a service, such as Apache and Rails, you would run something like `CMD ["apache2","-DFOREGROUND"]` . Indeed, this form of the instruction is recommended for any service-based image.

In most other cases, `CMD` should be given an interactive shell, such as bash, python and perl. For example, `CMD ["perl", "-de0"]` , `CMD ["python"]` , or `CMD ["php", "-a"]` . Using this form means that when you execute something like `docker run -it python` , you'll get dropped into a usable shell, ready to go. `CMD` should rarely be used in the manner of `CMD ["param", "param"]` in conjunction with `ENTRYPOINT` (https://docs.docker.com/engine/reference/builder/#entrypoint), unless you and your expected users are already quite familiar with how `ENTRYPOINT` works.

## EXPOSE

Dockerfile reference for the EXPOSE instruction (https://docs.docker.com/engine/reference/builder/#expose)

The `EXPOSE` instruction indicates the ports on which a container listens for connections. Consequently, you should use the common, traditional port for your application. For example, an image containing the Apache web server would use `EXPOSE 80` , while an image containing MongoDB would use `EXPOSE 27017` and so on.

For external access, your users can execute `docker run` with a flag indicating how to map the specified port to the port of their choice. For container linking, Docker provides environment variables for the path from the recipient container back to the source (ie, `MYSQL_PORT_3306_TCP` ).

## ENV

Dockerfile reference for the ENV instruction (https://docs.docker.com/engine/reference/builder/#env)

To make new software easier to run, you can use `ENV` to update the `PATH` environment variable for the software your container installs. For example, `ENV PATH /usr/local/nginx/bin:$PATH` ensures that `CMD ["nginx"]` just works.

The `ENV` instruction is also useful for providing required environment variables specific to services you wish to containerize, such as Postgres's `PGDATA` .

Lastly, `ENV` can also be used to set commonly used version numbers so that version bumps are easier to maintain, as seen in the following example:

```
ENV PG_MAJOR 9.3
ENV PG_VERSION 9.3.4
RUN curl -SL http://example.com/postgres-$PG_VERSION.tar.xz | tar -xJC /usr/src/postgress && …
ENV PATH /usr/local/postgres-$PG_MAJOR/bin:$PATH
```

Similar to having constant variables in a program (as opposed to hard-coding values), this approach lets you change a single `ENV` instruction to auto-magically bump the version of the software in your container.

Each `ENV` line creates a new intermediate layer, just like `RUN` commands. This means that even if you unset the environment variable in a future layer, it still persists in this layer and its value can be dumped. You can test this by creating a Dockerfile like the following, and then building it.

```
FROM alpine
ENV ADMIN_USER="mark"
RUN echo $ADMIN_USER > ./mark
RUN unset ADMIN_USER
CMD sh
```

```
$ docker run --rm -it test sh echo $ADMIN_USER

mark
```

To prevent this, and really unset the environment variable, use a `RUN` command with shell commands, to set, use, and unset the variable all in a single layer. You can separate your commands with `;` or `&&`. If you use the second method, and one of the commands fails, the `docker build` also fails. This is usually a good idea. Using `\` as a line continuation character for Linux Dockerfiles improves readability. You could also put all of the commands into a shell script and have the `RUN` command just run that shell script.

```
FROM alpine
RUN export ADMIN_USER="mark" \
    && echo $ADMIN_USER > ./mark \
    && unset ADMIN_USER
CMD sh
```

```
$ docker run --rm -it test sh echo $ADMIN_USER
```

## ADD or COPY

- Dockerfile reference for the ADD instruction (https://docs.docker.com/engine/reference/builder/#add)
- Dockerfile reference for the COPY instruction (https://docs.docker.com/engine/reference/builder/#copy)

Although `ADD` and `COPY` are functionally similar, generally speaking, `COPY` is preferred. That's because it's more transparent than `ADD`. `COPY` only supports the basic copying of local files into the container, while `ADD` has some features (like local-only tar extraction and remote URL support) that are not immediately obvious. Consequently, the best use for `ADD` is local tar file auto-extraction into the image, as in `ADD rootfs.tar.xz /`.

If you have multiple `Dockerfile` steps that use different files from your context, `COPY` them individually, rather than all at once. This ensures that each step's build cache is only invalidated (forcing the step to be re-run) if the specifically required files change.

For example:

```
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY . /tmp/
```

Results in fewer cache invalidations for the `RUN` step, than if you put the `COPY . /tmp/` before it.

Because image size matters, using `ADD` to fetch packages from remote URLs is strongly discouraged; you should use `curl` or `wget` instead. That way you can delete the files you no longer need after they've been extracted and you don't have to add another layer in your image. For example, you should avoid doing things like:

```
ADD http://example.com/big.tar.xz /usr/src/things/
RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things
RUN make -C /usr/src/things all
```

And instead, do something like:

```
RUN mkdir -p /usr/src/things \
    && curl -SL http://example.com/big.tar.xz \
    | tar -xJC /usr/src/things \
    && make -C /usr/src/things all
```

For other items (files, directories) that do not require `ADD` 's tar auto-extraction capability, you should always use `COPY` .

## ENTRYPOINT

Dockerfile reference for the ENTRYPOINT instruction (https://docs.docker.com/engine/reference/builder/#entrypoint)

The best use for `ENTRYPOINT` is to set the image's main command, allowing that image to be run as though it was that command (and then use `CMD` as the default flags).

Let's start with an example of an image for the command line tool `s3cmd` :

```
ENTRYPOINT ["s3cmd"]
CMD ["--help"]
```

Now the image can be run like this to show the command's help:

```
$ docker run s3cmd
```

Or using the right parameters to execute a command:

```
$ docker run s3cmd ls s3://mybucket
```

This is useful because the image name can double as a reference to the binary as shown in the command above.

The `ENTRYPOINT` instruction can also be used in combination with a helper script, allowing it to function in a similar way to the command above, even when starting the tool may require more than one step.

For example, the Postgres Official Image (https://hub.docker.com/_/postgres/) uses the following script as its `ENTRYPOINT` :

```
#!/bin/bash
set -e

if [ "$1" = 'postgres' ]; then
    chown -R postgres "$PGDATA"

    if [ -z "$(ls -A "$PGDATA")" ]; then
        gosu postgres initdb
    fi

    exec gosu postgres "$@"
fi

exec "$@"
```

> **Configure app as PID 1**
>
> This script uses the `exec` Bash command (http://wiki.bash-hackers.org/commands/builtin/exec) so that the final running application becomes the container's PID 1. This allows the application to receive any Unix signals sent to the container. For more, see the `ENTRYPOINT` reference (https://docs.docker.com/engine/reference/builder/#entrypoint).

The helper script is copied into the container and run via `ENTRYPOINT` on container start:

```
COPY ./docker-entrypoint.sh /
ENTRYPOINT ["/docker-entrypoint.sh"]
CMD ["postgres"]
```

This script allows the user to interact with Postgres in several ways.

It can simply start Postgres:

```
$ docker run postgres
```

Or, it can be used to run Postgres and pass parameters to the server:

```
$ docker run postgres postgres --help
```

Lastly, it could also be used to start a totally different tool, such as Bash:

```
$ docker run --rm -it postgres bash
```

## VOLUME

Dockerfile reference for the VOLUME instruction (https://docs.docker.com/engine/reference/builder/#volume)

The `VOLUME` instruction should be used to expose any database storage area, configuration storage, or files/folders created by your docker container. You are strongly encouraged to use `VOLUME` for any mutable and/or user-serviceable parts of your image.

## USER

Dockerfile reference for the USER instruction (https://docs.docker.com/engine/reference/builder/#user)

If a service can run without privileges, use `USER` to change to a non-root user. Start by creating the user and group in the `Dockerfile` with something like `RUN groupadd -r postgres && useradd --no-log-init -r -g postgres postgres`.

> Consider an explicit UID/GID
>
> Users and groups in an image are assigned a non-deterministic UID/GID in that the "next" UID/GID is assigned regardless of image rebuilds. So, if it's critical, you should assign an explicit UID/GID.

> Due to an unresolved bug (https://github.com/golang/go/issues/13548) in the Go archive/tar package's handling of sparse files, attempting to create a user with a significantly large UID inside a Docker container can lead to disk exhaustion because `/var/log/faillog` in the container layer is filled with NULL (\0) characters. A workaround is to pass the `--no-log-init` flag to useradd. The Debian/Ubuntu `adduser` wrapper does not support this flag.

Avoid installing or using `sudo` as it has unpredictable TTY and signal-forwarding behavior that can cause problems. If you absolutely need functionality similar to `sudo`, such as initializing the daemon as `root` but running it as non-`root`), consider using "gosu" (https://github.com/tianon/gosu).

Lastly, to reduce layers and complexity, avoid switching `USER` back and forth frequently.

### WORKDIR

Dockerfile reference for the WORKDIR instruction (https://docs.docker.com/engine/reference/builder/#workdir)

For clarity and reliability, you should always use absolute paths for your `WORKDIR`. Also, you should use `WORKDIR` instead of proliferating instructions like `RUN cd … && do-something`, which are hard to read, troubleshoot, and maintain.

### ONBUILD

Dockerfile reference for the ONBUILD instruction (https://docs.docker.com/engine/reference/builder/#onbuild)

An `ONBUILD` command executes after the current `Dockerfile` build completes. `ONBUILD` executes in any child image derived `FROM` the current image. Think of the `ONBUILD` command as an instruction the parent `Dockerfile` gives to the child `Dockerfile`.

A Docker build executes `ONBUILD` commands before any command in a child `Dockerfile`.

`ONBUILD` is useful for images that are going to be built `FROM` a given image. For example, you would use `ONBUILD` for a language stack image that builds arbitrary user software written in that language within the `Dockerfile`, as you can see in Ruby's `ONBUILD` variants (https://github.com/docker-library/ruby/blob/master/2.4/jessie/onbuild/Dockerfile).

Images built from `ONBUILD` should get a separate tag, for example: `ruby:1.9-onbuild` or `ruby:2.0-onbuild`.

Be careful when putting `ADD` or `COPY` in `ONBUILD`. The "onbuild" image fails catastrophically if the new build's context is missing the resource being added. Adding a separate tag, as recommended above, helps mitigate this by allowing the `Dockerfile` author to make a choice.

## Examples for Official Images

These Official Images have exemplary `Dockerfile`s:

- Go (https://hub.docker.com/_/golang/)
- Perl (https://hub.docker.com/_/perl/)

- Hy (https://hub.docker.com/_/hylang/)
- Ruby (https://hub.docker.com/_/ruby/)

## Additional resources:

- Dockerfile Reference (https://docs.docker.com/engine/reference/builder/)
- More about Base Images (https://docs.docker.com/develop/develop-images/baseimages/)
- More about Automated Builds (https://docs.docker.com/docker-hub/builds/)
- Guidelines for Creating Official Images (https://docs.docker.com/docker-hub/official_images/)

parent image (https://docs.docker.com/glossary/?term=parent image), images (https://docs.docker.com/glossary/?term=images), dockerfile (https://docs.docker.com/glossary/?term=dockerfile), best practices (https://docs.docker.com/glossary/?term=best practices), hub (https://docs.docker.com/glossary/?term=hub), official image (https://docs.docker.com/glossary/?term=official image)

# docker image ls

*Estimated reading time: 1 minute*

## Description

List images

## Usage

```
docker image ls [OPTIONS] [REPOSITORY[:TAG]]
```

## Options

| Name, shorthand | Default | Description |
|---|---|---|
| `--all , -a` | | Show all images (default hides intermediate images) |
| `--digests` | | Show digests |
| `--filter , -f` | | Filter output based on conditions provided |
| `--format` | | Pretty-print images using a Go template |
| `--no-trunc` | | Don’t truncate output |
| `--quiet , -q` | | Only show numeric IDs |

## Parent command

| Command | Description |
|---|---|

| Command | Description |
| --- | --- |
| docker image (https://docs.docker.com/engine/reference/commandline/image) | Manage images |

## Related commands

| Command | Description |
| --- | --- |
| docker image build (https://docs.docker.com/engine/reference/commandline/image_build/) | Build an image from a Dockerfile |
| docker image history (https://docs.docker.com/engine/reference/commandline/image_history/) | Show the history of an image |
| docker image import (https://docs.docker.com/engine/reference/commandline/image_import/) | Import the contents from a tarball to create a filesystem image |
| docker image inspect (https://docs.docker.com/engine/reference/commandline/image_inspect/) | Display detailed information on one or more images |
| docker image load (https://docs.docker.com/engine/reference/commandline/image_load/) | Load an image from a tar archive or STDIN |
| docker image ls (https://docs.docker.com/engine/reference/commandline/image_ls/) | List images |
| docker image prune (https://docs.docker.com/engine/reference/commandline/image_prune/) | Remove unused images |
| docker image pull (https://docs.docker.com/engine/reference/commandline/image_pull/) | Pull an image or a repository from a registry |

| Command | Description |
| --- | --- |
| docker image push (https://docs.docker.com/engine/reference/commandline/image_push/) | Push an image or a repository to a registry |
| docker image rm (https://docs.docker.com/engine/reference/commandline/image_rm/) | Remove one or more images |
| docker image save (https://docs.docker.com/engine/reference/commandline/image_save/) | Save one or more images to a tar archive (streamed to STDOUT by default) |
| docker image tag (https://docs.docker.com/engine/reference/commandline/image_tag/) | Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE |

# docker image rm

*Estimated reading time: 1 minute*

## Description

Remove one or more images

## Usage

```
docker image rm [OPTIONS] IMAGE [IMAGE...]
```

## Options

| Name, shorthand | Default | Description |
|---|---|---|
| `--force , -f` | | Force removal of the image |
| `--no-prune` | | Do not delete untagged parents |

## Parent command

| Command | Description |
|---|---|
| docker image (https://docs.docker.com/engine/reference/commandline/image) | Manage images |

## Related commands

| Command | Description |
|---|---|

| Command | Description |
| --- | --- |
| docker image build (https://docs.docker.com/engine/reference/commandline/image_build/) | Build an image from a Dockerfile |
| docker image history (https://docs.docker.com/engine/reference/commandline/image_history/) | Show the history of an image |
| docker image import (https://docs.docker.com/engine/reference/commandline/image_import/) | Import the contents from a tarball to create a filesystem image |
| docker image inspect (https://docs.docker.com/engine/reference/commandline/image_inspect/) | Display detailed information on one or more images |
| docker image load (https://docs.docker.com/engine/reference/commandline/image_load/) | Load an image from a tar archive or STDIN |
| docker image ls (https://docs.docker.com/engine/reference/commandline/image_ls/) | List images |
| docker image prune (https://docs.docker.com/engine/reference/commandline/image_prune/) | Remove unused images |
| docker image pull (https://docs.docker.com/engine/reference/commandline/image_pull/) | Pull an image or a repository from a registry |
| docker image push (https://docs.docker.com/engine/reference/commandline/image_push/) | Push an image or a repository to a registry |
| docker image rm (https://docs.docker.com/engine/reference/commandline/image_rm/) | Remove one or more images |

| Command | Description |
| --- | --- |
| docker image save (https://docs.docker.com/engine/reference/commandline/image_save/) | Save one or more images to a tar archive (streamed to STDOUT by default) |
| docker image tag (https://docs.docker.com/engine/reference/commandline/image_tag/) | Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE |

# docker image prune

*Estimated reading time: 7 minutes*

## Description

Remove unused images

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

## Usage

```
docker image prune [OPTIONS]
```

## Options

| Name, shorthand | Default | Description |
|---|---|---|
| `--all , -a` | | Remove all unused images, not just dangling ones |
| `--filter` | | Provide filter values (e.g. 'until=') |
| `--force , -f` | | Do not prompt for confirmation |

## Parent command

| Command | Description |
|---|---|
| docker image (https://docs.docker.com/engine/reference/commandline/image) | Manage images |

# Related commands

| Command | Description |
| --- | --- |
| docker image build (https://docs.docker.com/engine/reference/commandline/image_build/) | Build an image from a Dockerfile |
| docker image history (https://docs.docker.com/engine/reference/commandline/image_history/) | Show the history of an image |
| docker image import (https://docs.docker.com/engine/reference/commandline/image_import/) | Import the contents from a tarball to create a filesystem image |
| docker image inspect (https://docs.docker.com/engine/reference/commandline/image_inspect/) | Display detailed information on one or more images |
| docker image load (https://docs.docker.com/engine/reference/commandline/image_load/) | Load an image from a tar archive or STDIN |
| docker image ls (https://docs.docker.com/engine/reference/commandline/image_ls/) | List images |
| docker image prune (https://docs.docker.com/engine/reference/commandline/image_prune/) | Remove unused images |
| docker image pull (https://docs.docker.com/engine/reference/commandline/image_pull/) | Pull an image or a repository from a registry |
| docker image push (https://docs.docker.com/engine/reference/commandline/image_push/) | Push an image or a repository to a registry |
| docker image rm (https://docs.docker.com/engine/reference/commandline/image_rm/) | Remove one or more images |

| Command | Description |
| --- | --- |
| docker image save (https://docs.docker.com/engine/reference/commandline/image_save/) | Save one or more images to a tar archive (streamed to STDOUT by default) |
| docker image tag (https://docs.docker.com/engine/reference/commandline/image_tag/) | Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE |

# Extended description

Remove all dangling images. If `-r` is specified, will also remove all images not referenced by any container.

# Examples

Example output:

```
$ docker image prune -f

WARNING! This will remove all images without at least one container assoc
iated to them.
Are you sure you want to continue? [y/N] y
Deleted Images:
untagged: alpine:latest
untagged: alpine@sha256:3dcdb92d7432d56604d4545cbd324b14e647b313626d99b88
9d0626de158f73a
deleted: sha256:4e38e38c8ce0b8d9041a9c4fefe786631d1416225e13b0bfe8cfa2321
aec4bba
deleted: sha256:4fe15f8d0ae69e169824f25f1d4da3015a48feeeeebb265cd2e328e15
c6a869f
untagged: alpine:3.3
untagged: alpine@sha256:4fa633f4feff6a8f02acfc7424efd5cb3e76686ed3218abf4
ca0fa4a2a358423
untagged: my-jq:latest
deleted: sha256:ae67841be6d008a374eff7c2a974cde3934ffe9536a7dc7ce589585ed
dd83aff
deleted: sha256:34f6f1261650bc341eb122313372adc4512b4fceddc2a7ecbb84f0958
ce5ad65
deleted: sha256:cf4194e8d8db1cb2d117df33f2c75c0369c3a26d96725efb978cc69e0
46b87e7
untagged: my-curl:latest
deleted: sha256:b2789dd875bf427de7f9f6ae001940073b3201409b14aba7e5db71f40
8b8569e
deleted: sha256:96daac0cb203226438989926fc34dd024f365a9a8616b93e168d303cf
e4cb5e9
deleted: sha256:5cbd97a14241c9cd83250d6b6fc0649833c4a3e84099b968dd4ba403e
609945e
deleted: sha256:a0971c4015c1e898c60bf95781c6730a05b5d8a2ae6827f53837e6c9d
38efdec
deleted: sha256:d8359ca3b681cc5396a4e790088441673ed3ce90ebc04de388bfcd31a
0716b06
deleted: sha256:83fc9ba8fb70e1da31dfcc3c88d093831dbd4be38b34af998df37e8ac
538260c
deleted: sha256:ae7041a4cc625a9c8e6955452f7afe602b401f662671cea3613f08f3d
9343b35
deleted: sha256:35e0f43a37755b832f0bbea91a2360b025ee351d7309dae0d9737bc96
b6d0809
deleted: sha256:0af941dd29f00e4510195dd00b19671bc591e29d1495630e7e0f7c44c
1e6a8c0
deleted: sha256:9fc896fc2013da84f84e45b3096053eb084417b42e6b35ea0cce5a352
9705eac
deleted: sha256:47cf20d8c26c46fff71be614d9f54997edacfe8d46d51769706e5aba9
4b16f2b
deleted: sha256:2c675ee9ed53425e31a13e3390bf3f539bf8637000e4bcfbb85ee03ef
4d910a1

Total reclaimed space: 16.43 MB
```

## Filtering

The filtering flag ( `--filter` ) format is of "key=value". If there is more than one filter, then pass multiple flags (e.g., `--filter "foo=bar" --filter "bif=baz"` )

The currently supported filters are:

- until ( `<timestamp>` ) - only remove images created before given timestamp
- label ( `label=<key>` , `label=<key>=<value>` , `label!=<key>` , or `label!=<key>=<value>` ) - only remove images with (or without, in case `label!=...` is used) the specified labels.

The `until` filter can be Unix timestamps, date formatted timestamps, or Go duration strings (e.g. `10m` , `1h30m` ) computed relative to the daemon machine's time. Supported formats for date formatted time stamps include RFC3339Nano, RFC3339, `2006-01-02T15:04:05` , `2006-01-02T15:04:05.999999999` , `2006-01-02Z07:00` , and `2006-01-02` . The local timezone on the daemon will be used if you do not provide either a `Z` or a `+-00:00` timezone offset at the end of the timestamp. When providing Unix timestamps enter seconds[.nanoseconds], where seconds is the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT), not counting leap seconds (aka Unix epoch or Unix time), and the optional .nanoseconds field is a fraction of a second no more than nine digits long.

The `label` filter accepts two formats. One is the `label=...` ( `label=<key>` or `label=<key>=<value>` ), which removes images with the specified labels. The other format is the `label!=...` ( `label!=<key>` or `label!=<key>=<value>` ), which removes images without the specified labels.

> **Predicting what will be removed**
>
> If you are using positive filtering (testing for the existence of a label or that a label has a specific value), you can use `docker image ls` with the same filtering syntax to see which images match your filter.
>
> However, if you are using negative filtering (testing for the absence of a label or that a label does *not* have a specific value), this type of filter does not work with `docker image ls` so you cannot easily predict which images will be removed. In addition, the confirmation prompt for `docker image prune` always warns that *all* dangling images will be removed, even if you are using `--filter` .

The following removes images created before `2017-01-04T00:00:00` :

```
$ docker images --format 'table {{.Repository}}\t{{.Tag}}\t{{.ID}}\t{{.Cr
eatedAt}}\t{{.Size}}'
REPOSITORY          TAG                 IMAGE ID            CREATED AT
                   SIZE
foo                 latest              2f287ac753da        2017-01-04 13
:42:23 -0800 PST   3.98 MB
alpine              latest              88e169ea8f46        2016-12-27 10
:17:25 -0800 PST   3.98 MB
busybox             latest              e02e811dd08f        2016-10-07 14
:03:58 -0700 PDT   1.09 MB

$ docker image prune -f --force --filter "until=2017-01-04T00:00:00"

Deleted Images:
untagged: alpine:latest
untagged: alpine@sha256:dfbd4a3a8ebca874ebd2474f044a0b33600d4523d03b0df76
e5c5986cb02d7e8
untagged: busybox:latest
untagged: busybox@sha256:29f5d56d12684887bdfa50dcd29fc31eea4aaf4ad3bec43d
af19026a7ce69912
deleted: sha256:e02e811dd08fd49e7f6032625495118e63f597eb150403d02e3238af1
df240ba
deleted: sha256:e88b3f82283bc59d5e0df427c824e9f95557e661fcb0ea15fb0fb6f97
760f9d9

Total reclaimed space: 1.093 MB

$ docker images --format 'table {{.Repository}}\t{{.Tag}}\t{{.ID}}\t{{.Cr
eatedAt}}\t{{.Size}}'

REPOSITORY          TAG                 IMAGE ID            CREATED AT
                   SIZE
foo                 latest              2f287ac753da        2017-01-04 13
:42:23 -0800 PST   3.98 MB
```

The following removes images created more than 10 days ( `240h` ) ago:

```
$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED
       SIZE
foo                 latest              2f287ac753da        14 seconds ag
o      3.98 MB
alpine              latest              88e169ea8f46        8 days ago
       3.98 MB
debian              jessie              7b0a06c805e8        2 months ago
       123 MB
busybox             latest              e02e811dd08f        2 months ago
       1.09 MB
golang              1.7.0               138c2e655421        4 months ago
       670 MB

$ docker image prune -f --force --filter "until=240h"

Deleted Images:
untagged: golang:1.7.0
untagged: golang@sha256:6765038c2b8f407fd6e3ecea043b44580c229ccfa2a13f6d8
5866cf2b4a9628e
deleted: sha256:138c2e6554219de65614d88c15521bfb2da674cbb0bf840de161f89ff
4264b96
deleted: sha256:ec353c2e1a673f456c4b78906d0d77f9d9456cfb5229b78c6a960bfb7
496b76a
deleted: sha256:fe22765feaf3907526b4921c73ea6643ff9e334497c9b7e177972cf22
f68ee93
deleted: sha256:ff845959c80148421a5c3ae11cc0e6c115f950c89bc949646be55ed18
d6a2912
deleted: sha256:a4320831346648c03db64149eafc83092e2b34ab50ca6e8c13112388f
25899a7
deleted: sha256:4c76020202ee1d9709e703b7c6de367b325139e74eebd6b55b30a63c1
96abaf3
deleted: sha256:d7afd92fb07236c8a2045715a86b7d5f0066cef025018cd3ca9a45498
c51d1d6
deleted: sha256:9e63c5bce4585dd7038d830a1f1f4e44cb1a1515b00e620ac718e934b
484c938
untagged: debian:jessie
untagged: debian@sha256:c1af755d300d0c65bb1194d24bce561d70c98a54fb5ce5b16
93beb4f7988272f
deleted: sha256:7b0a06c805e8f23807fb8856621c60851727e85c7bcb751012c813f12
2734c8d
deleted: sha256:f96222d75c5563900bc4dd852179b720a0885de8f7a0619ba0ac76e92
542bbc8

Total reclaimed space: 792.6 MB

$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED
```

```
        SIZE
foo                 latest              2f287ac753da        About a minut
e ago   3.98 MB
alpine              latest              88e169ea8f46        8 days ago
        3.98 MB
busybox             latest              e02e811dd08f        2 months ago
        1.09 MB
```

The following example removes images with the label `deprecated`:

```
$ docker image prune --filter="label=deprecated"
```

The following example removes images with the label `maintainer` set to `john`:

```
$ docker image prune --filter="label=maintainer=john"
```

This example removes images which have no `maintainer` label:

```
$ docker image prune --filter="label!=maintainer"
```

This example removes images which have a maintainer label not set to `john`:

```
$ docker image prune --filter="label!=maintainer=john"
```

> Note: You are prompted for confirmation before the `prune` removes anything, but you are not shown a list of what will potentially be removed. In addition, `docker image ls` does not support negative filtering, so it difficult to predict what images will actually be removed.

# docker image inspect

*Estimated reading time: 1 minute*

## Description

Display detailed information on one or more images

## Usage

```
docker image inspect [OPTIONS] IMAGE [IMAGE...]
```

## Options

| Name, shorthand | Default | Description |
|---|---|---|
| `--format , -f` | | Format the output using the given Go template |

## Parent command

| Command | Description |
|---|---|
| docker image (https://docs.docker.com/engine/reference/commandline/image) | Manage images |

## Related commands

| Command | Description |
|---|---|
| docker image build (https://docs.docker.com/engine/reference/commandline/image_build/) | Build an image from a Dockerfile |

| Command | Description |
| --- | --- |
| docker image history (https://docs.docker.com/engine/reference/commandline/image_history/) | Show the history of an image |
| docker image import (https://docs.docker.com/engine/reference/commandline/image_import/) | Import the contents from a tarball to create a filesystem image |
| docker image inspect (https://docs.docker.com/engine/reference/commandline/image_inspect/) | Display detailed information on one or more images |
| docker image load (https://docs.docker.com/engine/reference/commandline/image_load/) | Load an image from a tar archive or STDIN |
| docker image ls (https://docs.docker.com/engine/reference/commandline/image_ls/) | List images |
| docker image prune (https://docs.docker.com/engine/reference/commandline/image_prune/) | Remove unused images |
| docker image pull (https://docs.docker.com/engine/reference/commandline/image_pull/) | Pull an image or a repository from a registry |
| docker image push (https://docs.docker.com/engine/reference/commandline/image_push/) | Push an image or a repository to a registry |
| docker image rm (https://docs.docker.com/engine/reference/commandline/image_rm/) | Remove one or more images |
| docker image save (https://docs.docker.com/engine/reference/commandline/image_save/) | Save one or more images to a tar archive (streamed to STDOUT by default) |

| Command | Description |
| --- | --- |
| docker image tag (https://docs.docker.com/engine/reference/commandline/image_tag/) | Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE |

# docker images

*Estimated reading time: 9 minutes*

## Description

List images

## Usage

```
docker images [OPTIONS] [REPOSITORY[:TAG]]
```

## Options

| Name, shorthand | Default | Description |
|---|---|---|
| `--all , -a` | | Show all images (default hides intermediate images) |
| `--digests` | | Show digests |
| `--filter , -f` | | Filter output based on conditions provided |
| `--format` | | Pretty-print images using a Go template |
| `--no-trunc` | | Don’t truncate output |
| `--quiet , -q` | | Only show numeric IDs |

## Parent command

| Command | Description |
|---|---|

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

# Extended description

The default `docker images` will show all top level images, their repository and tags, and their size.

Docker images have intermediate layers that increase reusability, decrease disk usage, and speed up `docker build` by allowing each step to be cached. These intermediate layers are not shown by default.

The `SIZE` is the cumulative space taken up by the image and all its parent images. This is also the disk space used by the contents of the Tar file created when you `docker save` an image.

An image will be listed more than once if it has multiple repository names or tags. This single image (identifiable by its matching `IMAGE ID`) uses up the `SIZE` listed only once.

# Examples

## List the most recently created images

```
$ docker images

REPOSITORY                TAG                 IMAGE ID            CR
EATED             SIZE
<none>                    <none>              77af4d6b9913        19
 hours ago        1.089 GB
committ                   latest              b6fa739cedf5        19
 hours ago        1.089 GB
<none>                    <none>              78a85c484f71        19
 hours ago        1.089 GB
docker                    latest              30557a29d5ab        20
 hours ago        1.089 GB
<none>                    <none>              5ed6274db6ce        24
 hours ago        1.089 GB
postgres                  9                   746b819f315e        4
days ago          213.4 MB
postgres                  9.3                 746b819f315e        4
days ago          213.4 MB
postgres                  9.3.5               746b819f315e        4
days ago          213.4 MB
postgres                  latest              746b819f315e        4
days ago          213.4 MB
```

## List images by name and tag

The `docker images` command takes an optional `[REPOSITORY[:TAG]]` argument that restricts the list to images that match the argument. If you specify `REPOSITORY` but no `TAG`, the `docker images` command lists all images in the given repository.

For example, to list all images in the "java" repository, run this command :

```
$ docker images java

REPOSITORY          TAG                 IMAGE ID            CREATED
            SIZE
java                8                   308e519aac60        6 days a
go          824.5 MB
java                7                   493d82594c15        3 months
 ago        656.3 MB
java                latest              2711b1d6f3aa        5 months
 ago        603.9 MB
```

The `[REPOSITORY[:TAG]]` value must be an “exact match”. This means that, for example, `docker images jav` does not match the image `java` .

If both `REPOSITORY` and `TAG` are provided, only images matching that repository and tag are listed. To find all local images in the “java” repository with tag “8” you can use:

```
$ docker images java:8

REPOSITORY          TAG                 IMAGE ID            CREATED
            SIZE
java                8                   308e519aac60        6 days a
go          824.5 MB
```

If nothing matches `REPOSITORY[:TAG]` , the list is empty.

```
$ docker images java:0

REPOSITORY          TAG                 IMAGE ID            CREATED
            SIZE
```

## List the full length image IDs

```
$ docker images --no-trunc

REPOSITORY                    TAG                 IMAGE ID
                                                        CREATED
        SIZE
<none>                        <none>              sha256:77af4d6b991
3e693e8d0b4b294fa62ade6054e6b2f1ffb617ac955dd63fb0182   19 hours ago
        1.089 GB
committest                    latest              sha256:b6fa739cedf
5ea12a620a439402b6004d057da800f91c7524b5086a5e4749c9f   19 hours ago
        1.089 GB
<none>                        <none>              sha256:78a85c484f7
1509adeaace20e72e941f6bdd2b25b4c75da8693efd9f61a37921   19 hours ago
        1.089 GB
docker                        latest              sha256:30557a29d5a
bc51e5f1d5b472e79b7e296f595abcf19fe6b9199dbbc809c6ff4   20 hours ago
        1.089 GB
<none>                        <none>              sha256:0124422dd9f
9cf7ef15c0617cda3931ee68346455441d66ab8bdc5b05e9fdce5   20 hours ago
        1.089 GB
<none>                        <none>              sha256:18ad6fad340
262ac2a636efd98a6d1f0ea775ae3d45240d3418466495a19a81b   22 hours ago
        1.082 GB
<none>                        <none>              sha256:f9f1e26352f
0a3ba6a0ff68167559f64f3e21ff7ada60366e2d44a04befd1d3a   23 hours ago
        1.089 GB
tryout                        latest              sha256:2629d1fa0b8
1b222fca63371ca16cbf6a0772d07759ff80e8d1369b926940074   23 hours ago
        131.5 MB
<none>                        <none>              sha256:5ed6274db6c
eb2397844896966ea239290555e74ef307030ebb01ff91b1914df   24 hours ago
        1.089 GB
```

## List image digests

Images that use the v2 or later 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. To list image digest values, use the `--digests` flag:

```
$ docker images --digests
REPOSITORY                          TAG                 DIGEST
                                                                IMAGE I
D            CREATED             SIZE
localhost:5000/test/busybox         <none>              sha256:cbbf2f
9a99b47fc460d422812b6a5adff7dfee951d8fa2e4a98caa0382cfbdbf   4986bf8
c1536        9 weeks ago         2.43 MB
```

When pushing or pulling to a 2.0 registry, the `push` or `pull` command output includes the image digest. You can `pull` using a digest value. You can also reference by digest in `create`, `run`, and `rmi` commands, as well as the `FROM` image reference in a Dockerfile.

## Filtering

The filtering flag (`-f` or `--filter`) format is of "key=value". If there is more than one filter, then pass multiple flags (e.g., `--filter "foo=bar" --filter "bif=baz"`)

The currently supported filters are:

- dangling (boolean - true or false)
- label (`label=<key>` or `label=<key>=<value>`)
- before (`<image-name>[:<tag>]`, `<image id>` or `<image@digest>`) - filter images created before given id or references
- since (`<image-name>[:<tag>]`, `<image id>` or `<image@digest>`) - filter images created since given id or references
- reference (pattern of an image reference) - filter images whose reference matches the specified pattern

### SHOW UNTAGGED IMAGES (DANGLING)

```
$ docker images --filter "dangling=true"

REPOSITORY          TAG                 IMAGE ID            CREATED
            SIZE
<none>              <none>              8abc22fbb042        4 weeks
ago         0 B
<none>              <none>              48e5f45168b9        4 weeks
ago         2.489 MB
<none>              <none>              bf747efa0e2f        4 weeks
ago         0 B
<none>              <none>              980fe10e5736        12 weeks
 ago        101.4 MB
<none>              <none>              dea752e4e117        12 weeks
 ago        101.4 MB
<none>              <none>              511136ea3c5a        8 months
 ago        0 B
```

This will display untagged images that are the leaves of the images tree (not intermediary layers). These images occur when a new build of an image takes the `repo:tag` away from the image ID, leaving it as `<none>:<none>` or untagged. A warning will be issued if trying to remove an image when a container is presently using it. By having this flag it allows for batch cleanup.

You can use this in conjunction with `docker rmi ...` :

```
$ docker rmi $(docker images -f "dangling=true" -q)

8abc22fbb042
48e5f45168b9
bf747efa0e2f
980fe10e5736
dea752e4e117
511136ea3c5a
```

> Note: Docker warns you if any containers exist that are using these untagged images.

### SHOW IMAGES WITH A GIVEN LABEL

The `label` filter matches images based on the presence of a `label` alone or a `label` and a value.

The following filter matches images with the `com.example.version` label regardless of its value.

```
$ docker images --filter "label=com.example.version"

REPOSITORY          TAG                 IMAGE ID            CREATED
             SIZE
match-me-1          latest              eeae25ada2aa        About a
minute ago   188.3 MB
match-me-2          latest              dea752e4e117        About a
minute ago   188.3 MB
```

The following filter matches images with the `com.example.version` label with the `1.0` value.

```
$ docker images --filter "label=com.example.version=1.0"

REPOSITORY          TAG                 IMAGE ID            CREATED
             SIZE
match-me            latest              511136ea3c5a        About a
minute ago   188.3 MB
```

In this example, with the `0.1` value, it returns an empty set because no matches were found.

```
$ docker images --filter "label=com.example.version=0.1"
REPOSITORY          TAG                 IMAGE ID            CREATED
             SIZE
```

### FILTER IMAGES BY TIME

The `before` filter shows only images created before the image with given id or reference. For example, having these images:

```
$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED
             SIZE
image1              latest              eeae25ada2aa        4 minute
s ago        188.3 MB
image2              latest              dea752e4e117        9 minute
s ago        188.3 MB
image3              latest              511136ea3c5a        25 minut
es ago       188.3 MB
```

Filtering with `before` would give:

```
$ docker images --filter "before=image1"

REPOSITORY          TAG                 IMAGE ID            CREATED
             SIZE
image2              latest              dea752e4e117        9 minute
s ago        188.3 MB
image3              latest              511136ea3c5a        25 minut
es ago       188.3 MB
```

Filtering with `since` would give:

```
$ docker images --filter "since=image3"
REPOSITORY          TAG                 IMAGE ID            CREATED
             SIZE
image1              latest              eeae25ada2aa        4 minute
s ago        188.3 MB
image2              latest              dea752e4e117        9 minute
s ago        188.3 MB
```

### FILTER IMAGES BY REFERENCE

The `reference` filter shows only images whose reference matches the specified pattern.

```
$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED
            SIZE
busybox             latest              e02e811dd08f        5 weeks
ago         1.09 MB
busybox             uclibc              e02e811dd08f        5 weeks
ago         1.09 MB
busybox             musl                733eb3059dce        5 weeks
ago         1.21 MB
busybox             glibc               21c16b6787c6        5 weeks
ago         4.19 MB
```

Filtering with `reference` would give:

```
$ docker images --filter=reference='busy*:*libc'

REPOSITORY          TAG                 IMAGE ID            CREATED
            SIZE
busybox             uclibc              e02e811dd08f        5 weeks
ago         1.09 MB
busybox             glibc               21c16b6787c6        5 weeks
ago         4.19 MB
```

## Format the output

The formatting option ( `--format` ) will pretty print container output using a Go template.

Valid placeholders for the Go template are listed below:

| Placeholder | Description |
|---|---|
| `.ID` | Image ID |
| `.Repository` | Image repository |
| `.Tag` | Image tag |
| `.Digest` | Image digest |
| `.CreatedSince` | Elapsed time since the image was created |

| Placeholder | Description |
| --- | --- |
| `.CreatedAt` | Time when the image was created |
| `.Size` | Image disk size |

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

The following example uses a template without headers and outputs the `ID` and `Repository` entries separated by a colon for all images:

```
$ docker images --format "{{.ID}}: {{.Repository}}"

77af4d6b9913: <none>
b6fa739cedf5: committ
78a85c484f71: <none>
30557a29d5ab: docker
5ed6274db6ce: <none>
746b819f315e: postgres
746b819f315e: postgres
746b819f315e: postgres
746b819f315e: postgres
```

To list all images with their repository and tag in a table format you can use:

```
$ docker images --format "table {{.ID}}\t{{.Repository}}\t{{.Tag}}"

IMAGE ID            REPOSITORY                TAG
77af4d6b9913        <none>                    <none>
b6fa739cedf5        committ                   latest
78a85c484f71        <none>                    <none>
30557a29d5ab        docker                    latest
5ed6274db6ce        <none>                    <none>
746b819f315e        postgres                  9
746b819f315e        postgres                  9.3
746b819f315e        postgres                  9.3.5
746b819f315e        postgres                  latest
```

# docker image build

*Estimated reading time: 4 minutes*

## Description

Build an image from a Dockerfile

## Usage

```
docker image build [OPTIONS] PATH | URL | -
```

## Options

| Name, shorthand | Default | Description |
|---|---|---|
| `--add-host` | | Add a custom host-to-IP mapping (host:ip) |
| `--build-arg` | | Set build-time variables |
| `--cache-from` | | Images to consider as cache sources |
| `--cgroup-parent` | | Optional parent cgroup for the container |
| `--compress` | | Compress the build context using gzip |
| `--cpu-period` | | Limit the CPU CFS (Completely Fair Scheduler) period |
| `--cpu-quota` | | Limit the CPU CFS (Completely Fair Scheduler) quota |
| `--cpu-shares , -c` | | CPU shares (relative weight) |
| `--cpuset-cpus` | | CPUs in which to allow execution (0-3, 0,1) |
| `--cpuset-mems` | | MEMs in which to allow execution (0-3, 0,1) |
| `--disable-content-trust` | `true` | Skip image verification |
| `--file , -f` | | Name of the Dockerfile (Default is 'PATH/Dockerfile') |
| `--force-rm` | | Always remove intermediate containers |
| `--iidfile` | | Write the image ID to the file |
| `--isolation` | | Container isolation technology |
| `--label` | | Set metadata for an image |

| Name, shorthand | Default | Description |
|---|---|---|
| `--memory , -m` | | Memory limit |
| `--memory-swap` | | Swap limit equal to memory plus swap: '-1' to enable unlimited swap |
| `--network` | | API 1.25+ (https://docs.docker.com/engine/api/v1.25/) Set the networking mode for the RUN instructions during build |
| `--no-cache` | | Do not use cache when building the image |
| `--platform` | | experimental (daemon) (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file) API 1.32+ (https://docs.docker.com/engine/api/v1.32/) Set platform if server is multi-platform capable |
| `--progress` | `auto` | Set type of progress output (auto, plain, tty). Use plain to show container output |
| `--pull` | | Always attempt to pull a newer version of the image |
| `--quiet , -q` | | Suppress the build output and print image ID on success |
| `--rm` | `true` | Remove intermediate containers after a successful build |
| `--secret` | | API 1.39+ (https://docs.docker.com/engine/api/v1.39/) Secret file to expose to the build (only if BuildKit enabled): id=mysecret,src=/local/secret |
| `--security-opt` | | Security options |
| `--shm-size` | | Size of /dev/shm |
| `--squash` | | experimental (daemon) (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file) API 1.25+ (https://docs.docker.com/engine/api/v1.25/) Squash newly built layers into a single new layer |
| `--ssh` | | API 1.39+ (https://docs.docker.com/engine/api/v1.39/) SSH agent socket or keys to expose to the build (only if BuildKit enabled) (format: default\|[=\|[,]]) |
| `--stream` | | experimental (daemon) (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file) API 1.31+ (https://docs.docker.com/engine/api/v1.31/) Stream attaches to server to negotiate build context |
| `--tag , -t` | | Name and optionally a tag in the 'name:tag' format |
| `--target` | | Set the target build stage to build. |
| `--ulimit` | | Ulimit options |

## Parent command

| Command | Description |
| --- | --- |
| docker image (https://docs.docker.com/engine/reference/commandline/image) | Manage images |

## Related commands

| Command | Description |
| --- | --- |
| docker image build (https://docs.docker.com/engine/reference/commandline/image_build/) | Build an image from a Dockerfile |
| docker image history (https://docs.docker.com/engine/reference/commandline/image_history/) | Show the history of an image |
| docker image import (https://docs.docker.com/engine/reference/commandline/image_import/) | Import the contents from a tarball to create a filesystem image |
| docker image inspect (https://docs.docker.com/engine/reference/commandline/image_inspect/) | Display detailed information on one or more images |
| docker image load (https://docs.docker.com/engine/reference/commandline/image_load/) | Load an image from a tar archive or STDIN |
| docker image ls (https://docs.docker.com/engine/reference/commandline/image_ls/) | List images |
| docker image prune (https://docs.docker.com/engine/reference/commandline/image_prune/) | Remove unused images |
| docker image pull (https://docs.docker.com/engine/reference/commandline/image_pull/) | Pull an image or a repository from a registry |
| docker image push (https://docs.docker.com/engine/reference/commandline/image_push/) | Push an image or a repository to a registry |
| docker image rm (https://docs.docker.com/engine/reference/commandline/image_rm/) | Remove one or more images |
| docker image save (https://docs.docker.com/engine/reference/commandline/image_save/) | Save one or more images to a tar archive (streamed to STDOUT by default) |
| docker image tag (https://docs.docker.com/engine/reference/commandline/image_tag/) | Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE |

# docker image history

*Estimated reading time: 1 minute*

## Description

Show the history of an image

## Usage

```
docker image history [OPTIONS] IMAGE
```

## Options

| Name, shorthand | Default | Description |
|---|---|---|
| `--format` | | Pretty-print images using a Go template |
| `--human , -H` | `true` | Print sizes and dates in human readable format |
| `--no-trunc` | | Don't truncate output |
| `--quiet , -q` | | Only show numeric IDs |

## Parent command

| Command | Description |
|---|---|
| docker image (https://docs.docker.com/engine/reference/commandline/image) | Manage images |

## Related commands

| Command | Description |
| --- | --- |
| docker image build (https://docs.docker.com/engine/reference/commandline/image_build/) | Build an image from a Dockerfile |
| docker image history (https://docs.docker.com/engine/reference/commandline/image_history/) | Show the history of an image |
| docker image import (https://docs.docker.com/engine/reference/commandline/image_import/) | Import the contents from a tarball to create a filesystem image |
| docker image inspect (https://docs.docker.com/engine/reference/commandline/image_inspect/) | Display detailed information on one or more images |
| docker image load (https://docs.docker.com/engine/reference/commandline/image_load/) | Load an image from a tar archive or STDIN |
| docker image ls (https://docs.docker.com/engine/reference/commandline/image_ls/) | List images |
| docker image prune (https://docs.docker.com/engine/reference/commandline/image_prune/) | Remove unused images |
| docker image pull (https://docs.docker.com/engine/reference/commandline/image_pull/) | Pull an image or a repository from a registry |
| docker image push (https://docs.docker.com/engine/reference/commandline/image_push/) | Push an image or a repository to a registry |
| docker image rm (https://docs.docker.com/engine/reference/commandline/image_rm/) | Remove one or more images |

| Command | Description |
| --- | --- |
| docker image save (https://docs.docker.com/engine/reference/commandline/image_save/) | Save one or more images to a tar archive (streamed to STDOUT by default) |
| docker image tag (https://docs.docker.com/engine/reference/commandline/image_tag/) | Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE |

# docker image push

*Estimated reading time: 1 minute*

## Description

Push an image or a repository to a registry

## Usage

```
docker image push [OPTIONS] NAME[:TAG]
```

## Options

| Name, shorthand | Default | Description |
|---|---|---|
| `--disable-content-trust` | `true` | Skip image signing |

## Parent command

| Command | Description |
|---|---|
| docker image (https://docs.docker.com/engine/reference/commandline/image) | Manage images |

## Related commands

| Command | Description |
|---|---|
| docker image build (https://docs.docker.com/engine/reference/commandline/image_build/) | Build an image from a Dockerfile |

| Command | Description |
| --- | --- |
| docker image history (https://docs.docker.com/engine/reference/commandline/image_history/) | Show the history of an image |
| docker image import (https://docs.docker.com/engine/reference/commandline/image_import/) | Import the contents from a tarball to create a filesystem image |
| docker image inspect (https://docs.docker.com/engine/reference/commandline/image_inspect/) | Display detailed information on one or more images |
| docker image load (https://docs.docker.com/engine/reference/commandline/image_load/) | Load an image from a tar archive or STDIN |
| docker image ls (https://docs.docker.com/engine/reference/commandline/image_ls/) | List images |
| docker image prune (https://docs.docker.com/engine/reference/commandline/image_prune/) | Remove unused images |
| docker image pull (https://docs.docker.com/engine/reference/commandline/image_pull/) | Pull an image or a repository from a registry |
| docker image push (https://docs.docker.com/engine/reference/commandline/image_push/) | Push an image or a repository to a registry |
| docker image rm (https://docs.docker.com/engine/reference/commandline/image_rm/) | Remove one or more images |
| docker image save (https://docs.docker.com/engine/reference/commandline/image_save/) | Save one or more images to a tar archive (streamed to STDOUT by default) |

| Command | Description |
| --- | --- |
| docker image tag (https://docs.docker.com/engine/reference/commandline/image_tag/) | Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE |

# docker create

*Estimated reading time: 11 minutes*

## Description

Create a new container

## Usage

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

## Options

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

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

| Name, shorthand | Default | Description |
|---|---|---|
| `--init` | | API 1.25+ (https://docs.docker.com/engine/api/v1.25/) Run an init inside the container that forwards signals and reaps processes |
| `--interactive , -i` | | Keep STDIN open even if not attached |
| `--io-maxbandwidth` | | Maximum IO bandwidth limit for the system drive (Windows only) |
| `--io-maxiops` | | Maximum IOps limit for the system drive (Windows only) |
| `--ip` | | IPv4 address (e.g., 172.30.100.104) |
| `--ip6` | | IPv6 address (e.g., 2001:db8::33) |
| `--ipc` | | IPC mode to use |
| `--isolation` | | Container isolation technology |
| `--kernel-memory` | | Kernel memory limit |
| `--label , -l` | | Set meta data on a container |
| `--label-file` | | Read in a line delimited file of labels |
| `--link` | | Add link to another container |
| `--link-local-ip` | | Container IPv4/IPv6 link-local addresses |
| `--log-driver` | | Logging driver for the container |
| `--log-opt` | | Log driver options |
| `--mac-address` | | Container MAC address (e.g., 92:d0:c6:0a:29:33) |
| `--memory , -m` | | Memory limit |
| `--memory-reservation` | | Memory soft limit |
| `--memory-swap` | | Swap limit equal to memory plus swap: '-1' to enable unlimited swap |
| `--memory-swappiness` | `-1` | Tune container memory swappiness (0 to 100) |
| `--mount` | | Attach a filesystem mount to the container |
| `--name` | | Assign a name to the container |
| `--net` | | Connect a container to a network |
| `--net-alias` | | Add network-scoped alias for the container |
| `--network` | | Connect a container to a network |
| `--network-alias` | | Add network-scoped alias for the container |
| `--no-healthcheck` | | Disable any container-specified HEALTHCHECK |

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

| Name, shorthand | Default | Description |
|---|---|---|
| `--volumes-from` | | Mount volumes from the specified container(s) |
| `--workdir , -w` | | Working directory inside the container |

# Parent command

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

# Extended description

The `docker create` command creates a writeable container layer over the specified image and prepares it for running the specified command. The container ID is then printed to `STDOUT`. This is similar to `docker run -d` except the container is never started. You can then use the `docker start <container_id>` command to start the container at any point.

This is useful when you want to set up a container configuration ahead of time so that it is ready to start when you need it. The initial status of the new container is `created`.

Please see the run command (https://docs.docker.com/engine/reference/commandline/run/) section and the Docker run reference (https://docs.docker.com/engine/reference/run/) for more details.

# Examples

## Create and start a container

```
$ docker create -t -i fedora bash

6d8af538ec541dd581ebc2a24153a28329acb5268abe5ef868c1f1a261221752

$ docker start -Γ -i 6d8af538ec5

bash-4.2#
```

## Initialize volumes

As of v1.4.0 container volumes are initialized during the `docker create` phase (i.e., `docker run` too). For example, this allows you to `create` the `data` volume container, and then use it from another container:

```
$ docker create -Ấ /data --name data ubuntu

240633dfbb98128fa77473d3d9018f6123b99c454b3251427ae190a7d951ad57

$ docker run --rm --volumes-from data ubuntu ls -la /data

total 8
drwxr-xr-x  2 root root 4096 Dec  5 04:10 .
drwxr-xr-x 48 root root 4096 Dec  5 04:11 ..
```

Similarly, `create` a host directory bind mounted volume container, which can then be used from the subsequent container:

```
$ docker create -Ấ /home/docker:/docker --name docker ubuntu

9aa88c08f319cd1e4515c3c46b0de7cc9aa75e878357b1e96f91e2c773029f03

$ docker run --rm --volumes-from docker ubuntu ls -la /docker

total 20
drwxr-sr-x  5 1000 staff  180 Dec  5 04:00 .
drwxr-xr-x 48 root root  4096 Dec  5 04:13 ..
-rw-rw-r--  1 1000 staff 3833 Dec  5 04:01 .ash_history
-rw-r--r--  1 1000 staff  446 Nov 28 11:51 .ashrc
-rw-r--r--  1 1000 staff   25 Dec  5 04:00 .gitconfig
drwxr-sr-x  3 1000 staff   60 Dec  1 03:28 .local
-rw-r--r--  1 1000 staff  920 Nov 28 11:51 .profile
drwx--S---  2 1000 staff  460 Dec  5 00:51 .ssh
drwxr-xr-x 32 1000 staff 1140 Dec  5 04:01 docker
```

Set storage driver options per container.

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

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

## Specify isolation technology for container (--isolation)

This option is useful in situations where you are running Docker containers on Windows. The `--isolation=<value>` option sets a container's isolation technology. On Linux, the only supported is the `default` option which uses Linux namespaces. On Microsoft Windows, you can specify these values:

| Value | Description |
|---|---|
| `default` | Use the value specified by the Docker daemon's `--exec-opt`. If the `daemon` does not specify an isolation technology, Microsoft Windows uses `process` as its default value if the |

| Value | Description |
| --- | --- |
| daemon is running on Windows server, or `hyperv` if running on Windows client. | |
| `process` | Namespace isolation only. |
| `hyperv` | Hyper-V hypervisor partition-based isolation. |

Specifying the `--isolation` flag without a value is the same as setting `--isolation="default"` .

## Dealing with dynamically created devices (--device-cgroup-rule)

Devices available to a container are assigned at creation time. The assigned devices will both be added to the cgroup.allow file and created into the container once it is run. This poses a problem when a new device needs to be added to running container.

One of the solution is to add a more permissive rule to a container allowing it access to a wider range of devices. For example, supposing our container needs access to a character device with major `42` and any number of minor number (added as new devices appear), the following rule would be added:

```
docker create --device-cgroup-rule='c 42:* rmw' -name my-container my-image
```

Then, a user could ask `udev` to execute a script that would `docker exec my-container mknod newDevX c 42 <minor>` the required device when it is added.

NOTE: initially present devices still need to be explicitly added to the create/run command

# docker export

*Estimated reading time: 1 minute*

## Description

Export a container's filesystem as a tar archive

## Usage

```
docker export [OPTIONS] CONTAINER
```

## Options

| Name, shorthand | Default | Description |
|---|---|---|
| `--output , -o` | | Write to a file, instead of STDOUT |

## Parent command

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

## Extended description

The `docker export` command does not export the contents of volumes associated with the container. If a volume is mounted on top of an existing directory in the container, `docker export` will export the contents of the *underlying* directory, not the contents of the volume.

Refer to Backup, restore, or migrate data volumes (https://docs.docker.com/v17.03/engine/tutorials/dockervolumes/#backup-restore-or-migrate-data-volumes) in the user guide for examples on exporting data in a volume.

# Examples

Each of these commands has the same result.

```
$ docker export red_panda > latest.tar
```

```
$ docker export --output="latest.tar" red_panda
```

# docker image import

*Estimated reading time: 2 minutes*

## Description

Import the contents from a tarball to create a filesystem image

## Usage

```
docker image import [OPTIONS] file|URL|- [REPOSITORY[:TAG]]
```

## Options

| Name, shorthand | Default | Description |
|---|---|---|
| `--change , -c` | | Apply Dockerfile instruction to the created image |
| `--message , -m` | | Set commit message for imported image |
| `--platform` | | experimental (daemon) (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file) API 1.32+ (https://docs.docker.com/engine/api/v1.32/) Set platform if server is multi-platform capable |

## Parent command

| Command | Description |
|---|---|
| docker image (https://docs.docker.com/engine/reference/commandline/image) | Manage images |

## Related commands

| Command | Description |
|---|---|
| docker image build (https://docs.docker.com/engine/reference/commandline/image_build/) | Build an image from a Dockerfile |

| Command | Description |
| --- | --- |
| docker image history (https://docs.docker.com/engine/reference/commandline/image_history/) | Show the history of an image |
| docker image import (https://docs.docker.com/engine/reference/commandline/image_import/) | Import the contents from a tarball to create a filesystem image |
| docker image inspect (https://docs.docker.com/engine/reference/commandline/image_inspect/) | Display detailed information on one or more images |
| docker image load (https://docs.docker.com/engine/reference/commandline/image_load/) | Load an image from a tar archive or STDIN |
| docker image ls (https://docs.docker.com/engine/reference/commandline/image_ls/) | List images |
| docker image prune (https://docs.docker.com/engine/reference/commandline/image_prune/) | Remove unused images |
| docker image pull (https://docs.docker.com/engine/reference/commandline/image_pull/) | Pull an image or a repository from a registry |
| docker image push (https://docs.docker.com/engine/reference/commandline/image_push/) | Push an image or a repository to a registry |
| docker image rm (https://docs.docker.com/engine/reference/commandline/image_rm/) | Remove one or more images |
| docker image save (https://docs.docker.com/engine/reference/commandline/image_save/) | Save one or more images to a tar archive (streamed to STDOUT by default) |
| docker image tag (https://docs.docker.com/engine/reference/commandline/image_tag/) | Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE |

# About storage drivers

*Estimated reading time: 14 minutes*

To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way.

Storage drivers allow you to create data in the writable layer of your container. The files won't be persisted after the container is deleted, and both read and write speeds are low.

Learn how to use volumes (https://docs.docker.com/storage/volumes/) to persist data and improve performance.

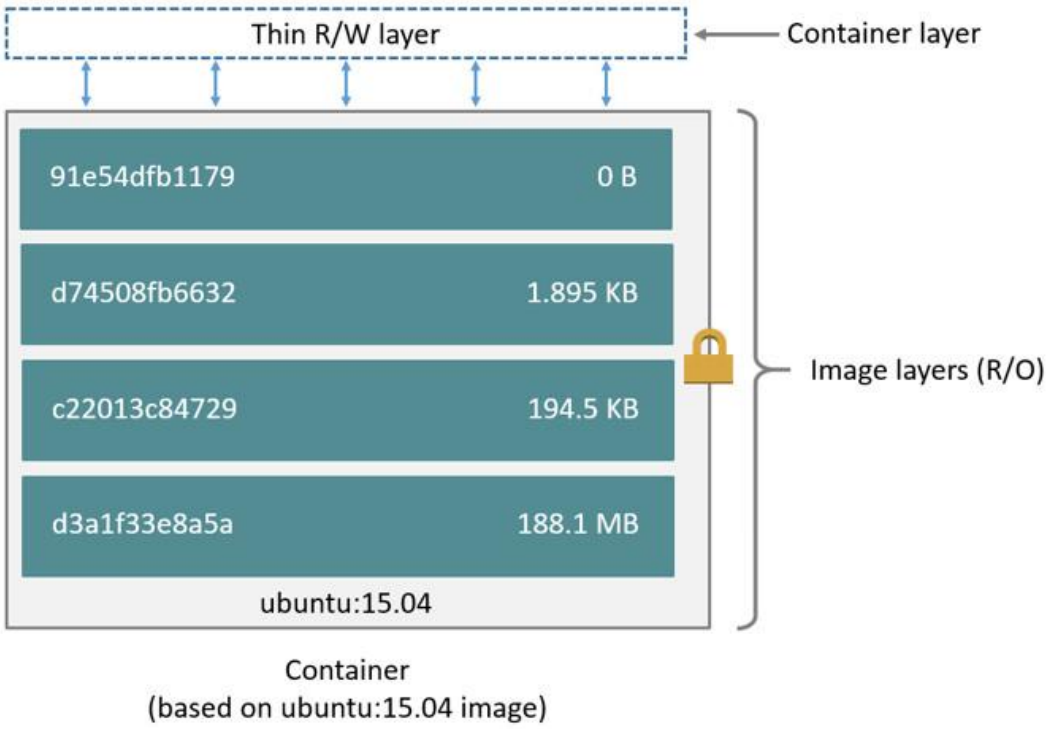
## Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

This Dockerfile contains four commands, each of which creates a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:15.04` image. The `COPY` command adds some files from your Docker client's current directory. The `RUN` command builds your application using the `make` command. Finally, the last layer specifies what command to run within the container.

Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on the Ubuntu 15.04 image.

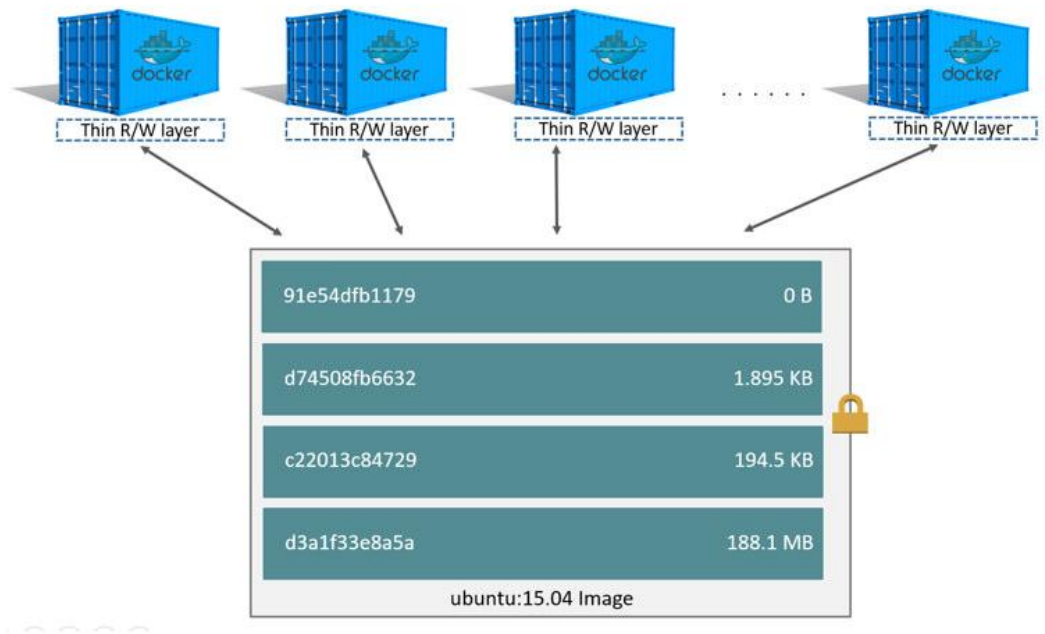



A *storage driver* handles the details about the way these layers interact with each other. Different storage drivers are available, which have advantages and disadvantages in different situations.

# Container and layers

The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.

Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.




> Note: If you need multiple images to have shared access to the exact same data, store this data in a Docker volume and mount it into your containers.

Docker uses storage drivers to manage the contents of the image layers and the writable container layer. Each storage driver handles the implementation differently, but all drivers use stackable image layers and the copy-on-write (CoW) strategy.

# Container size on disk

To view the approximate size of a running container, you can use the `docker ps -s` command. Two different columns relate to size.

- `size` : the amount of data (on disk) that is used for the writable layer of each container.
- `virtual size` : the amount of data used for the read-only image data used by the container plus the container's writable layer `size` . Multiple containers may share some or all read-only image data. Two containers started from the same image share 100% of the read-only data, while two containers with different images which have layers in common share those common layers. Therefore, you can't just total the virtual sizes. This over-estimates the total disk usage by a potentially non-trivial amount.

The total disk space used by all of the running containers on disk is some combination of each container's `size` and the `virtual size` values. If multiple containers started from the same exact image, the total size on disk for these containers would be SUM ( `size` of containers) plus one image size ( `virtual size - size` ).

This also does not count the following additional ways a container can take up disk space:

- Disk space used for log files if you use the `json-file` logging driver. This can be non-trivial if your container generates a large amount of logging data and log rotation is not configured.
- Volumes and bind mounts used by the container.
- Disk space used for the container's configuration files, which are typically small.
- Memory written to disk (if swapping is enabled).
- Checkpoints, if you're using the experimental checkpoint/restore feature.

# The copy-on-write (CoW) strategy

Copy-on-write is a strategy of sharing and copying files for maximum efficiency. If a file or directory exists in a lower layer within the image, and another layer (including the writable layer) needs read access to it, it just uses the existing file. The first time another layer needs to modify the file (when building the image or running the container), the file is copied into that layer and modified. This minimizes I/O and the size of each of the subsequent layers. These advantages are explained in more depth below.

## Sharing promotes smaller images

When you use `docker pull` to pull down an image from a repository, or when you create a container from an image that does not yet exist locally, each layer is pulled down separately, and stored in Docker's local storage area, which is usually `/var/lib/docker/` on Linux hosts. You can see these layers being pulled in this example:

```
$ docker pull ubuntu:15.04

15.04: Pulling from library/ubuntu
1ba8ac955b97: Pull complete
f157c4e5ede7: Pull complete
0b7e98f84c4c: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:5e279a9df07990286cce22e1b0f5b0490629ca6d187698746ae5e
28e604a640e
Status: Downloaded newer image for ubuntu:15.04
```

Each of these layers is stored in its own directory inside the Docker host's local storage area. To examine the layers on the filesystem, list the contents of `/var/lib/docker/<storage-driver>/layers/`. This example uses the `aufs` storage driver:

```
$ ls /var/lib/docker/aufs/layers
1d6674ff835b10f76e354806e16b950f91a191d3b471236609ab13a930275e24
5dbb0cbe0148cf447b9464a358c1587be586058d9a4c9ce079320265e2bb94e7
bef7199f2ed8e86fa4ada1309cfad3089e0542fec8894690529e4c04a7ca2d73
ebf814eccfe98f2704660ca1d844e4348db3b5ccc637eb905d4818fbfb00a06a
```

The directory names do not correspond to the layer IDs (this has been true since Docker 1.10).

Now imagine that you have two different Dockerfiles. You use the first one to create an image called `acme/my-base-image:1.0`.

```
FROM ubuntu:16.10
COPY . /app
```

The second one is based on `acme/my-base-image:1.0`, but has some additional layers:

```
FROM acme/my-base-image:1.0
CMD /app/hello.sh
```

The second image contains all the layers from the first image, plus a new layer with the `CMD` instruction, and a read-write container layer. Docker already has all the layers from the first image, so it does not need to pull them again. The two images share any layers they have in common.

If you build images from the two Dockerfiles, you can use `docker image ls` and `docker history` commands to verify that the cryptographic IDs of the shared layers are the same.

1. Make a new directory `cow-test/` and change into it.
2. Within `cow-test/`, create a new file with the following contents:

   ```
   #!/bin/sh
   echo "Hello world"
   ```

   Save the file, and make it executable:

   ```
   chmod +x hello.sh
   ```

3. Copy the contents of the first Dockerfile above into a new file called `Dockerfile.base`.
4. Copy the contents of the second Dockerfile above into a new file called `Dockerfile`.
5. Within the `cow-test/` directory, build the first image. Don't forget to include the final `.` in the command. That sets the `PATH`, which tells Docker where to look for any files that need to be added to the image.

   ```
   $ docker build -t acme/my-base-image:1.0 -f Dockerfile.base .

   Sending build context to Docker daemon  4.096kB
   Step 1/2 : FROM ubuntu:16.10
    ---> 31005225a745
   Step 2/2 : COPY . /app
    ---> Using cache
    ---> bd09118bcef6
   Successfully built bd09118bcef6
   Successfully tagged acme/my-base-image:1.0
   ```

6. Build the second image.

```
$ docker build -t acme/my-final-image:1.0 -f Dockerfile .

Sending build context to Docker daemon  4.096kB
Step 1/2 : FROM acme/my-base-image:1.0
 ---> bd09118bcef6
Step 2/2 : CMD /app/hello.sh
 ---> Running in a07b694759ba
 ---> dbf995fc07ff
Removing intermediate container a07b694759ba
Successfully built dbf995fc07ff
Successfully tagged acme/my-final-image:1.0
```

7. Check out the sizes of the images:

```
$ docker image ls

REPOSITORY                          TAG                 IMAG
E ID            CREATED             SIZE
acme/my-final-image                 1.0                 dbf9
95fc07ff        58 seconds ago      103MB
acme/my-base-image                  1.0                 bd09
118bcef6        3 minutes ago       103MB
```

8. Check out the layers that comprise each image:

```
$ docker history bd09118bcef6
IMAGE               CREATED             CREATED BY
                         SIZE                COMMENT
bd09118bcef6        4 minutes ago       /bin/sh -c #(nop) COPY
dir:35a7eb158c1504e...   100B
31005225a745        3 months ago        /bin/sh -c #(nop)  CMD
["/bin/bash"]            0B
<missing>           3 months ago        /bin/sh -c mkdir -p /ru
n/systemd && echo '...   7B
<missing>           3 months ago        /bin/sh -c sed -i 's/^#
\s*\(deb.*universe\...   2.78kB
<missing>           3 months ago        /bin/sh -c rm -rf /var/
lib/apt/lists/*          0B
<missing>           3 months ago        /bin/sh -c set -xe   &&
 echo '#!/bin/sh' >...   745B
<missing>           3 months ago        /bin/sh -c #(nop) ADD f
ile:eef57983bd66e3a...   103MB
```

```
$ docker history dbf995fc07ff

IMAGE               CREATED             CREATED BY
                         SIZE                COMMENT
dbf995fc07ff        3 minutes ago       /bin/sh -c #(nop)  CMD
["/bin/sh" "-c" "/a...   0B
bd09118bcef6        5 minutes ago       /bin/sh -c #(nop) COPY
dir:35a7eb158c1504e...   100B
31005225a745        3 months ago        /bin/sh -c #(nop)  CMD
["/bin/bash"]            0B
<missing>           3 months ago        /bin/sh -c mkdir -p /ru
n/systemd && echo '...   7B
<missing>           3 months ago        /bin/sh -c sed -i 's/^#
\s*\(deb.*universe\...   2.78kB
<missing>           3 months ago        /bin/sh -c rm -rf /var/
lib/apt/lists/*          0B
<missing>           3 months ago        /bin/sh -c set -xe   &&
 echo '#!/bin/sh' >...   745B
<missing>           3 months ago        /bin/sh -c #(nop) ADD f
ile:eef57983bd66e3a...   103MB
```

Notice that all the layers are identical except the top layer of the second image. All the other layers are shared between the two images, and are only stored once in `/var/lib/docker/`. The new layer actually doesn't take any room at all, because it is not changing any files, but only running a command.

> Note: The `<missing>` lines in the `docker history` output indicate that those layers were built on another system and are not available locally. This can be ignored.

## Copying makes containers efficient

When you start a container, a thin writable container layer is added on top of the other layers. Any changes the container makes to the filesystem are stored here. Any files the container does not change do not get copied to this writable layer. This means that the writable layer is as small as possible.

When an existing file in a container is modified, the storage driver performs a copy-on-write operation. The specifics steps involved depend on the specific storage driver. For the `aufs`, `overlay`, and `overlay2` drivers, the copy-on-write operation follows this rough sequence:

- Search through the image layers for the file to update. The process starts at the newest layer and works down to the base layer one layer at a time. When results are found, they are added to a cache to speed future operations.
- Perform a `copy_up` operation on the first copy of the file that is found, to copy the file to the container's writable layer.
- Any modifications are made to this copy of the file, and the container cannot see the read-only copy of the file that exists in the lower layer.

Btrfs, ZFS, and other drivers handle the copy-on-write differently. You can read more about the methods of these drivers later in their detailed descriptions.

Containers that write a lot of data consume more space than containers that do not. This is because most write operations consume new space in the container's thin writable top layer.

> Note: for write-heavy applications, you should not store the data in the container. Instead, use Docker volumes, which are independent of the running container and are designed to be efficient for I/O. In addition, volumes can be shared among containers and do not increase the size of your container's writable layer.

A `copy_up` operation can incur a noticeable performance overhead. This overhead is different depending on which storage driver is in use. Large files, lots of layers, and deep directory trees can make the impact more noticeable. This is mitigated by the fact that each `copy_up` operation only occurs the first time a given file is modified.

To verify the way that copy-on-write works, the following procedures spins up 5 containers based on the `acme/my-final-image:1.0` image we built earlier and examines how much room they take up.

> Note: This procedure doesn't work on Docker for Mac or Docker for Windows.

1. From a terminal on your Docker host, run the following `docker run` commands. The strings at the end are the IDs of each container.

```
$ docker run -dit --name my_container_1 acme/my-final-image:1.0
 bash \
  && docker run -dit --name my_container_2 acme/my-final-image:
1.0 bash \
  && docker run -dit --name my_container_3 acme/my-final-image:
1.0 bash \
  && docker run -dit --name my_container_4 acme/my-final-image:
1.0 bash \
  && docker run -dit --name my_container_5 acme/my-final-image:
1.0 bash

  c36785c423ec7e0422b2af7364a7ba4da6146cbba7981a0951fcc3fa0430c
409
  dcad7101795e4206e637d9358a818e5c32e13b349e62b00bf05cd5a4343ea
513
  1e7264576d78a3134fbaf7829bc24b1d96017cf2bc046b7cd8b08b5775c33
d0c
  38fa94212a419a082e6a6b87a8e2ec4a44dd327d7069b85892a707e3fc818
544
  1a174fc216cccf18ec7d4fe14e008e30130b11ede0f0f94a87982e310cf2e
765
```

2. Run the `docker ps` command to verify the 5 containers are running.

```
CONTAINER ID      IMAGE                     COMMAND     CREATED
              STATUS              PORTS      NAMES
1a174fc216cc      acme/my-final-image:1.0   "bash"      About a
 minute ago   Up About a minute              my_container_5
38fa94212a41      acme/my-final-image:1.0   "bash"      About a
 minute ago   Up About a minute              my_container_4
1e7264576d78      acme/my-final-image:1.0   "bash"      About a
 minute ago   Up About a minute              my_container_3
dcad7101795e      acme/my-final-image:1.0   "bash"      About a
 minute ago   Up About a minute              my_container_2
c36785c423ec      acme/my-final-image:1.0   "bash"      About a
 minute ago   Up About a minute              my_container_1
```

3. List the contents of the local storage area.

```
$ sudo ls /var/lib/docker/containers

1a174fc216cccf18ec7d4fe14e008e30130b11ede0f0f94a87982e310cf2e76
5
1e7264576d78a3134fbaf7829bc24b1d96017cf2bc046b7cd8b08b5775c33d0
c
38fa94212a419a082e6a6b87a8e2ec4a44dd327d7069b85892a707e3fc81854
4
c36785c423ec7e0422b2af7364a7ba4da6146cbba7981a0951fcc3fa0430c40
9
dcad7101795e4206e637d9358a818e5c32e13b349e62b00bf05cd5a4343ea51
3
```

4. Now check out their sizes:

```
$ sudo du -sh /var/lib/docker/containers/*

32K  /var/lib/docker/containers/1a174fc216cccf18ec7d4fe14e008e3
0130b11ede0f0f94a87982e310cf2e765
32K  /var/lib/docker/containers/1e7264576d78a3134fbaf7829bc24b1
d96017cf2bc046b7cd8b08b5775c33d0c
32K  /var/lib/docker/containers/38fa94212a419a082e6a6b87a8e2ec4
a44dd327d7069b85892a707e3fc818544
32K  /var/lib/docker/containers/c36785c423ec7e0422b2af7364a7ba4
da6146cbba7981a0951fcc3fa0430c409
32K  /var/lib/docker/containers/dcad7101795e4206e637d9358a818e5
c32e13b349e62b00bf05cd5a4343ea513
```

   Each of these containers only takes up 32k of space on the filesystem.

Not only does copy-on-write save space, but it also reduces start-up time. When you start a container (or multiple containers from the same image), Docker only needs to create the thin writable container layer.

If Docker had to make an entire copy of the underlying image stack each time it started a new container, container start times and disk space used would be significantly increased. This would be similar to the way that virtual machines work, with one or more virtual disks per virtual machine.

# Related information

- Volumes (https://docs.docker.com/storage/volumes/)

- Select a storage driver (https://docs.docker.com/storage/storagedriver/select-storage-driver/)

container (https://docs.docker.com/glossary/?term=container), storage (https://docs.docker.com/glossary/?term=storage), driver (https://docs.docker.com/glossary/?term=driver), AUFS (https://docs.docker.com/glossary/?term=AUFS), btfs (https://docs.docker.com/glossary/?term=btfs), devicemapper (https://docs.docker.com/glossary/?term=devicemapper), zvfs (https://docs.docker.com/glossary/?term=zvfs)

# Docker Registry

*Estimated reading time: 1 minute*

> Looking for Docker Trusted Registry?
>
> Docker Trusted Registry (DTR) is a commercial product that enables complete image management workflow, featuring LDAP integration, image signing, security scanning, and integration with Universal Control Plane. DTR is offered as an add-on to Docker Enterprise subscriptions of Standard or higher.
>
> Go to Docker Trusted Registry (https://docs.docker.com/ee/dtr/)

## What it is

The Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images. The Registry is open-source, under the permissive Apache license (http://en.wikipedia.org/wiki/Apache_License).

## Why use it

You should use the Registry if you want to:

- tightly control where your images are being stored
- fully own your images distribution pipeline
- integrate image storage and distribution tightly into your in-house development workflow

## Alternatives

Users looking for a zero maintenance, ready-to-go solution are encouraged to head-over to the Docker Hub (https://hub.docker.com), which provides a free-to-use, hosted Registry, plus additional features (organization accounts, automated builds, and more).

Users looking for a commercially supported version of the Registry should look into Docker Trusted Registry (https://docs.docker.com/datacenter/dtr/2.1/guides/).

# Requirements

The Registry is compatible with Docker engine version 1.6.0 or higher.

# Basic commands

Start your registry

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

Pull (or build) some image from the hub

```
docker pull ubuntu
```

Tag the image so that it points to your registry

```
docker image tag ubuntu localhost:5000/myfirstimage
```

Push it

```
docker push localhost:5000/myfirstimage
```

Pull it back

```
docker pull localhost:5000/myfirstimage
```

Now stop your registry and remove all data

```
docker container stop registry && docker container rm -v registry
```

# Next

You should now read the detailed introduction about the registry (https://docs.docker.com/registry/introduction/), or jump directly to deployment instructions (https://docs.docker.com/registry/deploying/).

registry (https://docs.docker.com/glossary/?term=registry), on-prem (https://docs.docker.com/glossary/?term=on-prem), images (https://docs.docker.com/glossary/?term=images), tags (https://docs.docker.com/glossary/?term=tags), repository (https://docs.docker.com/glossary/?term=repository), distribution (https://docs.docker.com/glossary/?term=distribution)

# Configuring a registry

*Estimated reading time: 32 minutes*

The Registry configuration is based on a YAML file, detailed below. While it comes with sane default values out of the box, you should review it exhaustively before moving your systems to production.

## Override specific configuration options

In a typical setup where you run your Registry from the official image, you can specify a configuration variable from the environment by passing `-e` arguments to your `docker run` stanza or from within a Dockerfile using the `ENV` instruction.

To override a configuration option, create an environment variable named `REGISTRY_variable` where `variable` is the name of the configuration option and the `_` (underscore) represents indention levels. For example, you can configure the `rootdirectory` of the `filesystem` storage backend:

```
storage:
  filesystem:
    rootdirectory: /var/lib/registry
```

To override this value, set an environment variable like this:

```
REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY=/somewhere
```

This variable overrides the `/var/lib/registry` value to the `/somewhere` directory.

> Note: Create a base configuration file with environment variables that can be configured to tweak individual values. Overriding configuration sections with environment variables is not recommended.

# Overriding the entire configuration file

If the default configuration is not a sound basis for your usage, or if you are having issues overriding keys from the environment, you can specify an alternate YAML configuration file by mounting it as a volume in the container.

Typically, create a new configuration file from scratch,named `config.yml` , then specify it in the `docker run` command:

```
$ docker run -d -p 5000:5000 --restart=always --name registry \
             -Â ‘pwd‘/config.yml:/etc/docker/registry/config.yml \
             registry:2
```

Use this example YAML file (https://github.com/docker/distribution/blob/master/cmd/registry/config-example.yml) as a starting point.

# List of configuration options

These are all configuration options for the registry. Some options in the list are mutually exclusive. Read the detailed reference information about each option before finalizing your configuration.

```
version: 0.1
log:
  accesslog:
    disabled: true
  level: debug
  formatter: text
  fields:
    service: registry
    environment: staging
  hooks:
    - type: mail
      disabled: true
      levels:
        - panic
      options:
        smtp:
          addr: mail.example.com:25
          username: mailuser
          password: password
          insecure: true
        from: sender@example.com
        to:
          - errors@example.com
loglevel: debug # deprecated: use "log"
storage:
  filesystem:
    rootdirectory: /var/lib/registry
    maxthreads: 100
  azure:
    accountname: accountname
    accountkey: base64encodedaccountkey
    container: containername
  gcs:
    bucket: bucketname
    keyfile: /path/to/keyfile
    rootdirectory: /gcs/object/name/prefix
    chunksize: 5242880
  s3:
    accesskey: awsaccesskey
    secretkey: awssecretkey
    region: us-west-1
    regionendpoint: http://myobjects.local
    bucket: bucketname
    encrypt: true
    keyid: mykeyid
    secure: true
    v4auth: true
    chunksize: 5242880
```

```
    multipartcopychunksize: 33554432
    multipartcopymaxconcurrency: 100
    multipartcopythresholdsize: 33554432
    rootdirectory: /s3/object/name/prefix
  swift:
    username: username
    password: password
    authurl: https://storage.myprovider.com/auth/v1.0 or https://stor
age.myprovider.com/v2.0 or https://storage.myprovider.com/v3/auth
    tenant: tenantname
    tenantid: tenantid
    domain: domain name for Openstack Identity v3 API
    domainid: domain id for Openstack Identity v3 API
    insecureskipverify: true
    region: fr
    container: containername
    rootdirectory: /swift/object/name/prefix
  oss:
    accesskeyid: accesskeyid
    accesskeysecret: accesskeysecret
    region: OSS region name
    endpoint: optional endpoints
    internal: optional internal endpoint
    bucket: OSS bucket
    encrypt: optional data encryption setting
    secure: optional ssl setting
    chunksize: optional size valye
    rootdirectory: optional root directory
  inmemory:  # This driver takes no parameters
  delete:
    enabled: false
  redirect:
    disable: false
  cache:
    blobdescriptor: redis
  maintenance:
    uploadpurging:
      enabled: true
      age: 168h
      interval: 24h
      dryrun: false
    readonly:
      enabled: false
auth:
  silly:
    realm: silly-realm
    service: silly-service
  token:
    realm: token-realm
```

```
    service: token-service
    issuer: registry-token-issuer
    rootcertbundle: /root/certs/bundle
  htpasswd:
    realm: basic-realm
    path: /path/to/htpasswd
middleware:
  registry:
    - name: ARegistryMiddleware
      options:
        foo: bar
  repository:
    - name: ARepositoryMiddleware
      options:
        foo: bar
  storage:
    - name: cloudfront
      options:
        baseurl: https://my.cloudfronted.domain.com/
        privatekey: /path/to/pem
        keypairid: cloudfrontkeypairid
        duration: 3000s
  storage:
    - name: redirect
      options:
        baseurl: https://example.com/
reporting:
  bugsnag:
    apikey: bugsnagapikey
    releasestage: bugsnagreleasestage
    endpoint: bugsnagendpoint
  newrelic:
    licensekey: newreliclicensekey
    name: newrelicname
    verbose: true
http:
  addr: localhost:5000
  prefix: /my/nested/registry/
  host: https://myregistryaddress.org:5000
  secret: asecretforlocaldevelopment
  relativeurls: false
  tls:
    certificate: /path/to/x509/public
    key: /path/to/x509/private
    clientcas:
      - /path/to/ca.pem
      - /path/to/another/ca.pem
    letsencrypt:
      cachefile: /path/to/cache-file
```

```
      email: emailused@letsencrypt.com
  debug:
    addr: localhost:5001
  headers:
    X-Content-Type-Options: [nosniff]
  http2:
    disabled: false
notifications:
  endpoints:
    - name: alistener
      disabled: false
      url: https://my.listener.com/event
      headers: <http.Header>
      timeout: 500
      threshold: 5
      backoff: 1000
      ignoredmediatypes:
        - application/octet-stream
redis:
  addr: localhost:6379
  password: asecret
  db: 0
  dialtimeout: 10ms
  readtimeout: 10ms
  writetimeout: 10ms
  pool:
    maxidle: 16
    maxactive: 64
    idletimeout: 300s
health:
  storagedriver:
    enabled: true
    interval: 10s
    threshold: 3
  file:
    - file: /path/to/checked/file
      interval: 10s
  http:
    - uri: http://server.to.check/must/return/200
      headers:
        Authorization: [Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==]
      statuscode: 200
      timeout: 3s
      interval: 10s
      threshold: 3
  tcp:
    - addr: redis-server.domain.com:6379
      timeout: 3s
      interval: 10s
```

```
      threshold: 3
proxy:
  remoteurl: https://registry-1.docker.io
  username: [username]
  password: [password]
compatibility:
  schema1:
    signingkeyfile: /etc/registry/key.json
validation:
  enabled: true
  manifests:
    urls:
      allow:
        - ^https?://([^/]+\.)*example\.com/
      deny:
        - ^https?://www\.example\.com/
```

In some instances a configuration option is optional but it contains child options marked as required. In these cases, you can omit the parent with all its children. However, if the parent is included, you must also include all the children marked required.

## version

```
version: 0.1
```

The `version` option is required. It specifies the configuration's version. It is expected to remain a top-level field, to allow for a consistent version check before parsing the remainder of the configuration file.

## log

The `log` subsection configures the behavior of the logging system. The logging system outputs everything to stdout. You can adjust the granularity and format with this configuration section.

```
log:
  accesslog:
    disabled: true
  level: debug
  formatter: text
  fields:
    service: registry
    environment: staging
```

| Parameter | Required | Description |
|---|---|---|
| `level` | no | Sets the sensitivity of logging output. Permitted values are `error`, `warn`, `info`, and `debug`. The default is `info`. |
| `formatter` | no | This selects the format of logging output. The format primarily affects how keyed attributes for a log line are encoded. Options are `text`, `json`, and `logstash`. The default is `text`. |
| `fields` | no | A map of field names to values. These are added to every log line for the context. This is useful for identifying log messages source after being mixed in other systems. |

## accesslog

```
accesslog:
  disabled: true
```

Within `log`, `accesslog` configures the behavior of the access logging system. By default, the access logging system outputs to stdout in Combined Log Format (https://httpd.apache.org/docs/2.4/logs.html#combined). Access logging can be disabled by setting the boolean flag `disabled` to `true`.

## hooks

```
hooks:
  - type: mail
    levels:
      - panic
    options:
      smtp:
        addr: smtp.sendhost.com:25
        username: sendername
        password: password
        insecure: true
      from: name@sendhost.com
      to:
        - name@receivehost.com
```

The `hooks` subsection configures the logging hooks' behavior. This subsection includes a sequence handler which you can use for sending mail, for example. Refer to `loglevel` to configure the level of messages printed.

## loglevel

> DEPRECATED: Please use log (/registry/configuration/#log) instead.

```
loglevel: debug
```

Permitted values are `error`, `warn`, `info` and `debug`. The default is `info`.

### `storage`

```
storage:
  filesystem:
    rootdirectory: /var/lib/registry
  azure:
    accountname: accountname
    accountkey: base64encodedaccountkey
    container: containername
  gcs:
    bucket: bucketname
    keyfile: /path/to/keyfile
    rootdirectory: /gcs/object/name/prefix
  s3:
    accesskey: awsaccesskey
    secretkey: awssecretkey
    region: us-west-1
    regionendpoint: http://myobjects.local
    bucket: bucketname
    encrypt: true
    keyid: mykeyid
    secure: true
    v4auth: true
    chunksize: 5242880
    multipartcopychunksize: 33554432
    multipartcopymaxconcurrency: 100
    multipartcopythresholdsize: 33554432
    rootdirectory: /s3/object/name/prefix
  swift:
    username: username
    password: password
    authurl: https://storage.myprovider.com/auth/v1.0 or https://stor
age.myprovider.com/v2.0 or https://storage.myprovider.com/v3/auth
    tenant: tenantname
    tenantid: tenantid
    domain: domain name for Openstack Identity v3 API
    domainid: domain id for Openstack Identity v3 API
    insecureskipverify: true
    region: fr
    container: containername
    rootdirectory: /swift/object/name/prefix
  oss:
    accesskeyid: accesskeyid
    accesskeysecret: accesskeysecret
    region: OSS region name
    endpoint: optional endpoints
    internal: optional internal endpoint
    bucket: OSS bucket
    encrypt: optional data encryption setting
    secure: optional ssl setting
```

```
    chunksize: optional size valye
    rootdirectory: optional root directory
  inmemory:
  delete:
    enabled: false
  cache:
    blobdescriptor: inmemory
  maintenance:
    uploadpurging:
      enabled: true
      age: 168h
      interval: 24h
      dryrun: false
    readonly:
      enabled: false
  redirect:
    disable: false
```

The `storage` option is required and defines which storage backend is in use. You must configure exactly one backend. If you configure more, the registry returns an error. You can choose any of these backend storage drivers:

| Storage driver | Description |
|---|---|
| `filesystem` | Uses the local disk to store registry files. It is ideal for development and may be appropriate for some small-scale production applications. See the driver's reference documentation (https://github.com/docker/docker.github.io/tree/master/registry/storage-drivers/filesystem.md). |
| `azure` | Uses Microsoft Azure Blob Storage. See the driver's reference documentation (https://github.com/docker/docker.github.io/tree/master/registry/storage-drivers/azure.md). |
| `gcs` | Uses Google Cloud Storage. See the driver's reference documentation (https://github.com/docker/docker.github.io/tree/master/registry/storage-drivers/gcs.md). |
| `s3` | Uses Amazon Simple Storage Service (S3) and compatible Storage Services. See the driver's reference documentation (https://github.com/docker/docker.github.io/tree/master/registry/storage-drivers/s3.md). |

| Storage driver | Description |
| --- | --- |
| `swift` | Uses Openstack Swift object storage. See the driver's reference documentation (https://github.com/docker/docker.github.io/tree/master/registry/storage-drivers/swift.md). |
| `oss` | Uses Aliyun OSS for object storage. See the driver's reference documentation (https://github.com/docker/docker.github.io/tree/master/registry/storage-drivers/oss.md). |

For testing only, you can use the `inmemory` storage driver (https://github.com/docker/docker.github.io/tree/master/registry/storage-drivers/inmemory.md). If you would like to run a registry from volatile memory, use the `filesystem` driver (https://github.com/docker/docker.github.io/tree/master/registry/storage-drivers/filesystem.md) on a ramdisk.

If you are deploying a registry on Windows, a Windows volume mounted from the host is not recommended. Instead, you can use a S3 or Azure backing data-store. If you do use a Windows volume, the length of the `PATH` to the mount point must be within the `MAX_PATH` limits (typically 255 characters), or this error will occur:

```
mkdir /XXX protocol error and your registry will not function properl
y.
```

## maintenance

Currently, upload purging and read-only mode are the only `maintenance` functions available.

## uploadpurging

Upload purging is a background process that periodically removes orphaned files from the upload directories of the registry. Upload purging is enabled by default. To configure upload directory purging, the following parameters must be set.

| Parameter | Required | Description |
| --- | --- | --- |

| Parameter | Required | Description |
|---|---|---|
| `enabled` | yes | Set to `true` to enable upload purging. Defaults to `true` . |
| `age` | yes | Upload directories which are older than this age will be deleted.Defaults to `168h` (1 week). |
| `interval` | yes | The interval between upload directory purging. Defaults to `24h` . |
| `dryrun` | yes | Set `dryrun` to `true` to obtain a summary of what directories will be deleted. Defaults to `false` . |

> Note: `age` and `interval` are strings containing a number with optional fraction and a unit suffix. Some examples: `45m` , `2h10m` , `168h` .

## `readonly`

If the `readonly` section under `maintenance` has `enabled` set to `true` , clients will not be allowed to write to the registry. This mode is useful to temporarily prevent writes to the backend storage so a garbage collection pass can be run. Before running garbage collection, the registry should be restarted with readonly's `enabled` set to true. After the garbage collection pass finishes, the registry may be restarted again, this time with `readonly` removed from the configuration (or set to false).

## `delete`

Use the `delete` structure to enable the deletion of image blobs and manifests by digest. It defaults to false, but it can be enabled by writing the following on the configuration file:

```
delete:
  enabled: true
```

## `cache`

Use the `cache` structure to enable caching of data accessed in the storage backend. Currently, the only available cache provides fast access to layer metadata, which uses the `blobdescriptor` field if configured.

You can set `blobdescriptor` field to `redis` or `inmemory` . If set to `redis` ,a Redis pool caches layer metadata. If set to `inmemory` , an in-memory map caches layer metadata.

> NOTE: Formerly, `blobdescriptor` was known as `layerinfo` . While these are equivalent, `layerinfo` has been deprecated.

## redirect

The `redirect` subsection provides configuration for managing redirects from content backends. For backends that support it, redirecting is enabled by default. In certain deployment scenarios, you may decide to route all data through the Registry, rather than redirecting to the backend. This may be more efficient when using a backend that is not co-located or when a registry instance is aggressively caching.

To disable redirects, add a single flag `disable` , set to `true` under the `redirect` section:

```
redirect:
  disable: true
```

## auth

```
auth:
  silly:
    realm: silly-realm
    service: silly-service
  token:
    realm: token-realm
    service: token-service
    issuer: registry-token-issuer
    rootcertbundle: /root/certs/bundle
  htpasswd:
    realm: basic-realm
    path: /path/to/htpasswd
```

The `auth` option is optional. Possible auth providers include:

- `silly` (/registry/configuration/#silly)
- `token` (/registry/configuration/#token)
- `htpasswd` (/registry/configuration/#token)

You can configure only one authentication provider.

### silly

The `silly` authentication provider is only appropriate for development. It simply checks for the existence of the `Authorization` header in the HTTP request. It does not check the header's value. If the header does not exist, the `silly` auth responds with a challenge response, echoing back the realm, service, and scope for which access was denied.

The following values are used to configure the response:

| Parameter | Required | Description |
|---|---|---|
| `realm` | yes | The realm in which the registry server authenticates. |
| `service` | yes | The service being authenticated. |

### token

Token-based authentication allows you to decouple the authentication system from the registry. It is an established authentication paradigm with a high degree of security.

| Parameter | Required | Description |
|---|---|---|
| `realm` | yes | The realm in which the registry server authenticates. |
| `service` | yes | The service being authenticated. |
| `issuer` | yes | The name of the token issuer. The issuer inserts this into the token so it must match the value configured for the issuer. |
| `rootcertbundle` | yes | The absolute path to the root certificate bundle. This bundle contains the public part of the certificates used to sign authentication tokens. |

For more information about Token based authentication configuration, see the specification (https://docs.docker.com/registry/spec/auth/token/).

### `htpasswd`

The *htpasswd* authentication backed allows you to configure basic authentication using an Apache htpasswd file (https://httpd.apache.org/docs/2.4/programs/htpasswd.html). The only supported password format is `bcrypt` (http://en.wikipedia.org/wiki/Bcrypt). Entries with other hash types are ignored. The `htpasswd` file is loaded once, at startup. If the file is invalid, the registry will display an error and will not start.

> Warning: Only use the `htpasswd` authentication scheme with TLS configured, since basic authentication sends passwords as part of the HTTP header.

| Parameter | Required | Description |
|---|---|---|
| `realm` | yes | The realm in which the registry server authenticates. |

| Parameter | Required | Description |
|---|---|---|
| `path` | yes | The path to the `htpasswd` file to load at startup. |

## middleware

The `middleware` structure is optional. Use this option to inject middleware at named hook points. Each middleware must implement the same interface as the object it is wrapping. For instance, a registry middleware must implement the `distribution.Namespace` interface, while a repository middleware must implement `distribution.Repository`, and a storage middleware must implement `driver.StorageDriver`.

This is an example configuration of the `cloudfront` middleware, a storage middleware:

```
middleware:
  registry:
    - name: ARegistryMiddleware
      options:
        foo: bar
  repository:
    - name: ARepositoryMiddleware
      options:
        foo: bar
  storage:
    - name: cloudfront
      options:
        baseurl: https://my.cloudfronted.domain.com/
        privatekey: /path/to/pem
        keypairid: cloudfrontkeypairid
        duration: 3000s
```

Each middleware entry has `name` and `options` entries. The `name` must correspond to the name under which the middleware registers itself. The `options` field is a map that details custom configuration required to initialize the middleware. It is treated as a `map[string]interface{}`. As such, it supports any interesting structures desired, leaving it up to the middleware initialization function to best determine how to handle the specific interpretation of the options.

### cloudfront

| Parameter | Required | Description |
|---|---|---|
| `baseurl` | yes | The `SCHEME://HOST[/PATH]` at which Cloudfront is served. |
| `privatekey` | yes | The private key for Cloudfront, provided by AWS. |
| `keypairid` | yes | The key pair ID provided by AWS. |
| `duration` | no | An integer and unit for the duration of the Cloudfront session. Valid time units are `ns`, `us` (or `µs`), `ms`, `s`, `m`, or `!`. For example, `3000s` is valid, but `3000 s` is not. If you do not specify a `duration` or you specify an integer without a time unit, the duration defaults to `20m` (20 minutes). |

### redirect

You can use the `redirect` storage middleware to specify a custom URL to a location of a proxy for the layer stored by the S3 storage driver.

| Parameter | Required | Description |
|---|---|---|
| `baseurl` | yes | `SCHEME://HOST` at which layers are served. Can also contain port. For example, `https://example.com:5443`. |

## reporting

```
reporting:
  bugsnag:
    apikey: bugsnagapikey
    releasestage: bugsnagreleasestage
    endpoint: bugsnagendpoint
  newrelic:
    licensekey: newreliclicensekey
    name: newrelicname
    verbose: true
```

The `reporting` option is optional and configures error and metrics reporting tools. At the moment only two services are supported:

- Bugsnag (/registry/configuration/#bugsnag)
- New Relic (/registry/configuration/#new-relic)

A valid configuration may contain both.

### bugsnag

| Parameter | Required | Description |
|---|---|---|
| `apikey` | yes | The API Key provided by Bugsnag. |
| `releasestage` | no | Tracks where the registry is deployed, using a string like `production` , `staging` , or `development` . |
| `endpoint` | no | The enterprise Bugsnag endpoint. |

### newrelic

| Parameter | Required | Description |
|---|---|---|
| `licensekey` | yes | License key provided by New Relic. |
| `name` | no | New Relic application name. |
| `verbose` | no | Set to `true` to enable New Relic debugging output on `stdout` . |

## http

```
http:
  addr: localhost:5000
  net: tcp
  prefix: /my/nested/registry/
  host: https://myregistryaddress.org:5000
  secret: asecretforlocaldevelopment
  relativeurls: false
  tls:
    certificate: /path/to/x509/public
    key: /path/to/x509/private
    clientcas:
      - /path/to/ca.pem
      - /path/to/another/ca.pem
    letsencrypt:
      cachefile: /path/to/cache-file
      email: emailused@letsencrypt.com
  debug:
    addr: localhost:5001
  headers:
    X-Content-Type-Options: [nosniff]
  http2:
    disabled: false
```

The `http` option details the configuration for the HTTP server that hosts the registry.

| Parameter | Required | Description |
|---|---|---|
| `addr` | yes | The address for which the server should accept connections. The form depends on a network type (see the `net` option). Use `HOST:PORT` for TCP and `FILE` for a UNIX socket. |
| `net` | no | The network used to create a listening socket. Known networks are `unix` and `tcp` . |
| `prefix` | no | If the server does not run at the root path, set this to the value of the prefix. The root path is the section before `v2` . It requires both preceding and trailing slashes, such as in the example `/path/` . |

| Parameter | Required | Description |
|---|---|---|
| `host` | no | A fully-qualified URL for an externally-reachable address for the registry. If present, it is used when creating generated URLs. Otherwise, these URLs are derived from client requests. |
| `secret` | no | A random piece of data used to sign state that may be stored with the client to protect against tampering. For production environments you should generate a random piece of data using a cryptographically secure random generator. If you omit the secret, the registry will automatically generate a secret when it starts. If you are building a cluster of registries behind a load balancer, you MUST ensure the secret is the same for all registries. |
| `relativeurls` | no | If `true`, the registry returns relative URLs in Location headers. The client is responsible for resolving the correct URL. This option is not compatible with Docker 1.7 and earlier. |

### `tls`

The `tls` structure within `http` is optional. Use this to configure TLS for the server. If you already have a web server running on the same host as the registry, you may prefer to configure TLS on that web server and proxy connections to the registry server.

| Parameter | Required | Description |
|---|---|---|
| `certificate` | yes | Absolute path to the x509 certificate file. |
| `key` | yes | Absolute path to the x509 private key file. |
| `clientcas` | no | An array of absolute paths to x509 CA files. |

### `letsencrypt`

The `letsencrypt` structure within `tls` is optional. Use this to configure TLS certificates provided by Let's Encrypt (https://letsencrypt.org/how-it-works/).

> NOTE: When using Let's Encrypt, ensure that the outward-facing address is accessible on port `443`. The registry defaults to listening on port `5000`. If you run the registry as a container, consider adding the flag `-p 443:5000` to the `docker run` command or using a similar setting in a cloud configuration.

| Parameter | Required | Description |
|---|---|---|
| `cachefile` | yes | Absolute path to a file where the Let's Encrypt agent can cache data. |
| `email` | yes | The email address used to register with Let's Encrypt. |

### `debug`

The `debug` option is optional. Use it to configure a debug server that can be helpful in diagnosing problems. The debug endpoint can be used for monitoring registry metrics and health, as well as profiling. Sensitive information may be available via the debug endpoint. Please be certain that access to the debug endpoint is locked down in a production environment.

The `debug` section takes a single required `addr` parameter, which specifies the `HOST:PORT` on which the debug server should accept connections.

### `headers`

The `headers` option is optional. Use it to specify headers that the HTTP server should include in responses. This can be used for security headers such as `Strict-Transport-Security`.

The `headers` option should contain an option for each header to include, where the parameter name is the header's name, and the parameter value a list of the header's payload values.

Including `X-Content-Type-Options: [nosniff]` is recommended, so that browsers will not interpret content as HTML if they are directed to load a page from the registry. This header is included in the example configuration file.

### `http2`

The `http2` structure within `http` is optional. Use this to control http2 settings for the registry.

| Parameter | Required | Description |
|---|---|---|
| `disabled` | no | If `true`, then `http2` support is disabled. |

## notifications

```
notifications:
  endpoints:
    - name: alistener
      disabled: false
      url: https://my.listener.com/event
      headers: <http.Header>
      timeout: 500
      threshold: 5
      backoff: 1000
      ignoredmediatypes:
        - application/octet-stream
```

The notifications option is optional and currently may contain a single option, `endpoints`.

### endpoints

The `endpoints` structure contains a list of named services (URLs) that can accept event notifications.

| Parameter | Required | Description |
|---|---|---|
| `name` | yes | A human-readable name for the service. |
| `disabled` | no | If `true`, notifications are disabled for the service. |
| `url` | yes | The URL to which events should be published. |

| Parameter | Required | Description |
|---|---|---|
| `headers` | yes | A list of static headers to add to each request. Each header's name is a key beneath `headers`, and each value is a list of payloads for that header name. Values must always be lists. |
| `timeout` | yes | A value for the HTTP timeout. A positive integer and an optional suffix indicating the unit of time, which may be `ns`, `us`, `ms`, `s`, `m`, or `!`. If you omit the unit of time, `ns` is used. |
| `threshold` | yes | An integer specifying how long to wait before backing off a failure. |
| `backoff` | yes | How long the system backs off before retrying after a failure. A positive integer and an optional suffix indicating the unit of time, which may be `ns`, `us`, `ms`, `s`, `m`, or `!`. If you omit the unit of time, `ns` is used. |
| `ignoredmediatypes` | no | A list of target media types to ignore. Events with these target media types are not published to the endpoint. |

## redis

```
redis:
  addr: localhost:6379
  password: asecret
  db: 0
  dialtimeout: 10ms
  readtimeout: 10ms
  writetimeout: 10ms
  pool:
    maxidle: 16
    maxactive: 64
    idletimeout: 300s
```

Declare parameters for constructing the `redis` connections. Registry instances may use the Redis instance for several applications. Currently, it caches information about immutable blobs. Most of the `redis` options control how the registry connects to the `redis` instance. You can control the pool's behavior with the pool (/registry/configuration/#pool) subsection.

You should configure Redis with the allkeys-lru eviction policy, because the registry does not set an expiration value on keys.

| Parameter | Required | Description |
|---|---|---|
| `addr` | yes | The address (host and port) of the Redis instance. |
| `password` | no | A password used to authenticate to the Redis instance. |
| `db` | no | The name of the database to use for each connection. |
| `dialtimeout` | no | The timeout for connecting to the Redis instance. |
| `readtimeout` | no | The timeout for reading from the Redis instance. |
| `writetimeout` | no | The timeout for writing to the Redis instance. |

## `pool`

```
pool:
  maxidle: 16
  maxactive: 64
  idletimeout: 300s
```

Use these settings to configure the behavior of the Redis connection pool.

| Parameter | Required | Description |
|---|---|---|
| `maxidle` | no | The maximum number of idle connections in the pool. |

| Parameter | Required | Description |
|---|---|---|
| `maxactive` | no | The maximum number of connections which can be open before blocking a connection request. |
| `idletimeout` | no | How long to wait before closing inactive connections. |

## health

```
health:
  storagedriver:
    enabled: true
    interval: 10s
    threshold: 3
  file:
    - file: /path/to/checked/file
      interval: 10s
  http:
    - uri: http://server.to.check/must/return/200
      headers:
        Authorization: [Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==]
      statuscode: 200
      timeout: 3s
      interval: 10s
      threshold: 3
  tcp:
    - addr: redis-server.domain.com:6379
      timeout: 3s
      interval: 10s
      threshold: 3
```

The health option is optional, and contains preferences for a periodic health check on the storage driver's backend storage, as well as optional periodic checks on local files, HTTP URIs, and/or TCP servers. The results of the health checks are available at the `/debug/health` endpoint on the debug HTTP server if the debug HTTP server is enabled (see http section).

### storagedriver

The `storagedriver` structure contains options for a health check on the configured storage driver's backend storage. The health check is only active when `enabled` is set to `true` .

| Parameter | Required | Description |
|---|---|---|
| `enabled` | yes | Set to `true` to enable storage driver health checks or `false` to disable them. |
| `interval` | no | How long to wait between repetitions of the storage driver health check. A positive integer and an optional suffix indicating the unit of time. The suffix is one of `ns`, `us`, `ms`, `s`, `m`, or `!`. Defaults to `10s` if the value is omitted. If you specify a value but omit the suffix, the value is interpreted as a number of nanoseconds. |
| `threshold` | no | A positive integer which represents the number of times the check must fail before the state is marked as unhealthy. If not specified, a single failure marks the state as unhealthy. |

### `file`

The `file` structure includes a list of paths to be periodically checked for the\ existence of a file. If a file exists at the given path, the health check will fail. You can use this mechanism to bring a registry out of rotation by creating a file.

| Parameter | Required | Description |
|---|---|---|
| `file` | yes | The path to check for existence of a file. |
| `interval` | no | How long to wait before repeating the check. A positive integer and an optional suffix indicating the unit of time. The suffix is one of `ns`, `us`, `ms`, `s`, `m`, or `!`. Defaults to `10s` if the value is omitted. |

### `http`

The `http` structure includes a list of HTTP URIs to periodically check with `HEAD` requests. If a `HEAD` request does not complete or returns an unexpected status code, the health check will fail.

| Parameter | Required | Description |
|---|---|---|
| `uri` | yes | The URI to check. |

| Parameter | Required | Description |
|---|---|---|
| `headers` | no | Static headers to add to each request. Each header's name is a key beneath `headers`, and each value is a list of payloads for that header name. Values must always be lists. |
| `statuscode` | no | The expected status code from the HTTP URI. Defaults to `200`. |
| `timeout` | no | How long to wait before timing out the HTTP request. A positive integer and an optional suffix indicating the unit of time. The suffix is one of `ns`, `us`, `ms`, `s`, `m`, or `!`. If you specify a value but omit the suffix, the value is interpreted as a number of nanoseconds. |
| `interval` | no | How long to wait before repeating the check. A positive integer and an optional suffix indicating the unit of time. The suffix is one of `ns`, `us`, `ms`, `s`, `m`, or `!`. Defaults to `10s` if the value is omitted. If you specify a value but omit the suffix, the value is interpreted as a number of nanoseconds. |
| `threshold` | no | The number of times the check must fail before the state is marked as unhealthy. If this field is not specified, a single failure marks the state as unhealthy. |

### `tcp`

The `tcp` structure includes a list of TCP addresses to periodically check using TCP connection attempts. Addresses must include port numbers. If a connection attempt fails, the health check will fail.

| Parameter | Required | Description |
|---|---|---|
| `addr` | yes | The TCP address and port to connect to. |

| Parameter | Required | Description |
|---|---|---|
| `timeout` | no | How long to wait before timing out the TCP connection. A positive integer and an optional suffix indicating the unit of time. The suffix is one of `ns`, `us`, `ms`, `s`, `m`, or `!`. If you specify a value but omit the suffix, the value is interpreted as a number of nanoseconds. |
| `interval` | no | How long to wait between repetitions of the check. A positive integer and an optional suffix indicating the unit of time. The suffix is one of `ns`, `us`, `ms`, `s`, `m`, or `!`. Defaults to `10s` if the value is omitted. If you specify a value but omit the suffix, the value is interpreted as a number of nanoseconds. |
| `threshold` | no | The number of times the check must fail before the state is marked as unhealthy. If this field is not specified, a single failure marks the state as unhealthy. |

## proxy

```
proxy:
  remoteurl: https://registry-1.docker.io
  username: [username]
  password: [password]
```

The `proxy` structure allows a registry to be configured as a pull-through cache to Docker Hub. See mirror (https://github.com/docker/docker.github.io/tree/master/registry/recipes/mirror.md) for more information. Pushing to a registry configured as a pull-through cache is unsupported.

| Parameter | Required | Description |
|---|---|---|
| `remoteurl` | yes | The URL for the repository on Docker Hub. |
| `username` | no | The username registered with Docker Hub which has access to the repository. |

| Parameter | Required | Description |
|---|---|---|
| `password` | no | The password used to authenticate to Docker Hub using the username specified in `username` . |

To enable pulling private repositories (e.g. `batman/robin` ) specify the username (such as `batman` ) and the password for that username.

> Note: These private repositories are stored in the proxy cache's storage. Take appropriate measures to protect access to the proxy cache.

## compatibility

```
compatibility:
  schema1:
    signingkeyfile: /etc/registry/key.json
```

Use the `compatibility` structure to configure handling of older and deprecated features. Each subsection defines such a feature with configurable behavior.

### schema1

| Parameter | Required | Description |
|---|---|---|
| `signingkeyfile` | no | The signing private key used to add signatures to `schema1` manifests. If no signing key is provided, a new ECDSA key is generated when the registry starts. |

### validation

```
validation:
  enabled: true
  manifests:
    urls:
      allow:
        - ^https?://([^/]+\.)*example\.com/
      deny:
        - ^https?://www\.example\.com/
```

#### enabled

Use the `enabled` flag to enable the other options in the `validation` section. They are disabled by default.

#### manifests

Use the `manifest` subsection to configure manifest validation.

##### URLS

The `allow` and `deny` options are each a list of regular expressions (https://godoc.org/regexp/syntax) that restrict the URLs in pushed manifests.

If `allow` is unset, pushing a manifest containing URLs fails.

If `allow` is set, pushing a manifest succeeds only if all URLs match one of the `allow` regular expressions and one of the following holds:

1. `deny` is unset.
2. `deny` is set but no URLs within the manifest match any of the `deny` regular expressions.

## Example: Development configuration

You can use this simple example for local development:

```
version: 0.1
log:
  level: debug
storage:
    filesystem:
        rootdirectory: /var/lib/registry
http:
    addr: localhost:5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
```

This example configures the registry instance to run on port `5000`, binding to `localhost`, with the `debug` server enabled. Registry data is stored in the `/var/lib/registry` directory. Logging is set to `debug` mode, which is the most verbose.

See config-example.yml (https://github.com/docker/distribution/blob/master/cmd/registry/config-example.yml) for another simple configuration. Both examples are generally useful for local development.

## Example: Middleware configuration

This example configures Amazon Cloudfront (http://aws.amazon.com/cloudfront/) as the storage middleware in a registry. Middleware allows the registry to serve layers via a content delivery network (CDN). This reduces requests to the storage layer.

Cloudfront requires the S3 storage driver.

This is the configuration expressed in YAML:

```
middleware:
  storage:
  - name: cloudfront
    disabled: false
    options:
      baseurl: http://d111111abcdef8.cloudfront.net
      privatekey: /path/to/asecret.pem
      keypairid: asecret
      duration: 60
```

See the configuration reference for Cloudfront (/registry/configuration/#cloudfront) for more information about configuration options.

> Note: Cloudfront keys exist separately from other AWS keys. See the documentation on AWS credentials (http://docs.aws.amazon.com/general/latest/gr/aws-security-credentials.html) for more information.

registry (https://docs.docker.com/glossary/?term=registry), on-prem (https://docs.docker.com/glossary/?term=on-prem), images (https://docs.docker.com/glossary/?term=images), tags (https://docs.docker.com/glossary/?term=tags), repository (https://docs.docker.com/glossary/?term=repository), distribution (https://docs.docker.com/glossary/?term=distribution), configuration (https://docs.docker.com/glossary/?term=configuration)

# docker login

*Estimated reading time: 6 minutes*

## Description

Log in to a Docker registry

## Usage

```
docker login [OPTIONS] [SERVER]
```

## Options

| Name, shorthand | Default | Description |
|---|---|---|
| `--password , -p` | | Password |
| `--password-stdin` | | Take the password from stdin |
| `--username , -u` | | Username |

## Parent command

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

# Extended description

Login to a registry.

## Login to a self-hosted registry

If you want to login to a self-hosted registry you can specify this by adding the server name.

```
$ docker login localhost:8080
```

## Provide a password using STDIN

To run the `docker login` command non-interactively, you can set the `--password-stdin` flag to provide a password through `STDIN`. Using `STDIN` prevents the password from ending up in the shell's history, or log-files.

The following example reads a password from a file, and passes it to the `docker login` command using `STDIN`:

```
$ cat ~/my_password.txt | docker login --username foo --password-std
in
```

## Privileged user requirement

`docker login` requires user to use `sudo` or be `root`, except when:

1. connecting to a remote daemon, such as a `docker-machine` provisioned `docker engine`.
2. user is added to the `docker` group. This will impact the security of your system; the `docker` group is `root` equivalent. See Docker Daemon Attack Surface (https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface) for details.

You can log into any public or private repository for which you have credentials. When you log in, the command stores credentials in `$HOME/.docker/config.json` on Linux or `%USERPROFILE%/.docker/config.json` on Windows, via the procedure described below.

## Credentials store

The Docker Engine can keep user credentials in an external credentials store, such as the native keychain of the operating system. Using an external store is more secure than storing credentials in the Docker configuration file.

To use a credentials store, you need an external helper program to interact with a specific keychain or external store. Docker requires the helper program to be in the client's host `$PATH` .

This is the list of currently available credentials helpers and where you can download them from:

- D-Bus Secret Service: https://github.com/docker/docker-credential-helpers/releases
- Apple macOS keychain: https://github.com/docker/docker-credential-helpers/releases
- Microsoft Windows Credential Manager: https://github.com/docker/docker-credential-helpers/releases
- pass (https://www.passwordstore.org/): https://github.com/docker/docker-credential-helpers/releases

## CONFIGURE THE CREDENTIALS STORE

You need to specify the credentials store in `$HOME/.docker/config.json` to tell the docker engine to use it. The value of the config property should be the suffix of the program to use (i.e. everything after `docker-credential-` ). For example, to use `docker-credential-osxkeychain` :

```
{
        "credsStore": "osxkeychain"
}
```

If you are currently logged in, run `docker logout` to remove the credentials from the file and run `docker login` again.

## DEFAULT BEHAVIOR

By default, Docker looks for the native binary on each of the platforms, i.e. "osxkeychain" on macOS, "wincred" on windows, and "pass" on Linux. A special case is that on Linux, Docker will fall back to the "secretservice" binary if it cannot find the "pass" binary. If none of these binaries are present, it stores the credentials (i.e. password) in base64 encoding in the config files described above.

## CREDENTIAL HELPER PROTOCOL

Credential helpers can be any program or script that follows a very simple protocol. This protocol is heavily inspired by Git, but it differs in the information shared.

The helpers always use the first argument in the command to identify the action. There are only three possible values for that argument: `store` , `get` , and `erase` .

The `store` command takes a JSON payload from the standard input. That payload carries the server address, to identify the credential, the user name, and either a password or an identity token.

```
{
        "ServerURL": "https://index.docker.io/v1",
        "Username": "david",
        "Secret": "passw0rd1"
}
```

If the secret being stored is an identity token, the Username should be set to `<token>` .

The `store` command can write error messages to `STDOUT` that the docker engine will show if there was an issue.

The `get` command takes a string payload from the standard input. That payload carries the server address that the docker engine needs credentials for. This is an example of that payload: `https://index.docker.io/v1` .

The `get` command writes a JSON payload to `STDOUT` . Docker reads the user name and password from this payload:

```
{
        "Username": "david",
        "Secret": "passw0rd1"
}
```

The `erase` command takes a string payload from `STDIN` . That payload carries the server address that the docker engine wants to remove credentials for. This is an example of that payload: `https://index.docker.io/v1` .

The `erase` command can write error messages to `STDOUT` that the docker engine will show if there was an issue.

## Credential helpers

Credential helpers are similar to the credential store above, but act as the designated programs to handle credentials for *specific registries*. The default credential store ( `credsStore` or the config file itself) will not be used for operations concerning credentials of the specified registries.

### CONFIGURE CREDENTIAL HELPERS

If you are currently logged in, run `docker logout` to remove the credentials from the default store.

Credential helpers are specified in a similar way to `credsStore` , but allow for multiple helpers to be configured at a time. Keys specify the registry domain, and values specify the suffix of the program to use (i.e. everything after `docker-credential-` ). For example:

```
{
  "credHelpers": {
    "registry.example.com": "registryhelper",
    "awesomereg.example.org": "hip-star",
    "unicorn.example.io": "vcbait"
  }
}
```

# docker search

*Estimated reading time: 5 minutes*

## Description

Search the Docker Hub for images

## Usage

```
docker search [OPTIONS] TERM
```

## Options

| Name, shorthand | Default | Description |
|---|---|---|
| `--automated` | | deprecated (https://docs.docker.com/engine/deprecated/) Only show automated builds |
| `--filter , -f` | | Filter output based on conditions provided |
| `--format` | | Pretty-print search using a Go template |
| `--limit` | `25` | Max number of search results |
| `--no-trunc` | | Don't truncate output |
| `--stars , -s` | | deprecated (https://docs.docker.com/engine/deprecated/) Only displays with at least x stars |

# Parent command

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

# Extended description

Search Docker Hub (https://hub.docker.com) for images

See *Find Public Images on Docker Hub* (https://docs.docker.com/engine/tutorials/dockerrepos/#searching-for-images) for more details on finding shared images from the command line.

> Note: Search queries return a maximum of 25 results.

# Examples

## Search images by name

This example displays images with a name containing 'busybox':

```
$ docker search busybox

NAME                                DESCRIPTION
                STARS     OFFICIAL   AUTOMATED
busybox                             Busybox base image.
                316       [OK]
progrium/busybox
                50                   [OK]
radial/busyboxplus                  Full-chain, Internet enabled, busyb
ox made...      8                    [OK]
odise/busybox-python
                2                    [OK]
azukiapp/busybox                    This image is meant to be used as t
he base...      2                    [OK]
ofayau/busybox-jvm                  Prepare busybox to install a 32 bit
s JVM.          1                    [OK]
shingonoide/archlinux-busybox       Arch Linux, a lightweight and flexi
ble Lin...      1                    [OK]
odise/busybox-curl
                1                    [OK]
ofayau/busybox-libc32               Busybox with 32 bits (and 64 bits)
libs            1                    [OK]
peelsky/zulu-openjdk-busybox
                1                    [OK]
skomma/busybox-data                 Docker image suitable for data volu
me cont...      1                    [OK]
elektritter/busybox-teamspeak       Lightweight teamspeak3 container ba
sed on...       1                    [OK]
socketplane/busybox
                1                    [OK]
oveits/docker-nginx-busybox         This is a tiny NginX docker image b
ased on...      0                    [OK]
ggtools/busybox-ubuntu              Busybox ubuntu version with extra g
oodies          0                    [OK]
nikfoundas/busybox-confd            Minimal busybox based distribution
of confd        0                    [OK]
openshift/busybox-http-app
                0                    [OK]
jllopis/busybox
                0                    [OK]
swyckoff/busybox
                0                    [OK]
powellquiring/busybox
                0                    [OK]
williamyeh/busybox-sh               Docker image for BusyBox's sh
                0                    [OK]
simplexsys/busybox-cli-powered      Docker busybox images, with a few o
ften us...      0                    [OK]
```

```
fhisamoto/busybox-java          Busybox java
             0                   [OK]
scottabernethy/busybox
             0                   [OK]
marclop/busybox-solr
```

## Display non-truncated description (--no-trunc)

This example displays images with a name containing 'busybox', at least 3 stars and the description isn't truncated in the output:

```
$ docker search --stars=3 --no-trunc busybox
NAME                DESCRIPTION
                                        STARS     OFFICIAL   AUTO
MATED
busybox             Busybox base image.
                                        325       [OK]
progrium/busybox
                                        50                   [OK]
radial/busyboxplus   Full-chain, Internet enabled, busybox made from
 scratch. Comes in git and cURL flavors.   8                   [OK]
```

## Limit search results (--limit)

The flag `--limit` is the maximum number of results returned by a search. This value could be in the range between 1 and 100. The default value of `--limit` is 25.

## Filtering

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

The currently supported filters are:

- stars (int - number of stars the image has)
- is-automated (boolean - true or false) - is the image automated or not
- is-official (boolean - true or false) - is the image official or not

#### STARS

This example displays images with a name containing 'busybox' and at least 3

stars:

```
$ docker search --filter stars=3 busybox

NAME                DESCRIPTION
 STARS     OFFICIAL   AUTOMATED
busybox             Busybox base image.
 325       [OK]
progrium/busybox
 50                  [OK]
radial/busyboxplus  Full-chain, Internet enabled, busybox made...
 8                   [OK]
```

### IS-AUTOMATED

This example displays images with a name containing 'busybox' and are automated builds:

```
$ docker search --filter is-automated busybox

NAME                DESCRIPTION
 STARS     OFFICIAL   AUTOMATED
progrium/busybox
 50                  [OK]
radial/busyboxplus  Full-chain, Internet enabled, busybox made...
 8                   [OK]
```

### IS-OFFICIAL

This example displays images with a name containing 'busybox', at least 3 stars and are official builds:

```
$ docker search --filter "is-official=true" --filter "stars=3" busyb
ox

NAME                DESCRIPTION
 STARS     OFFICIAL   AUTOMATED
progrium/busybox
 50                  [OK]
radial/busyboxplus  Full-chain, Internet enabled, busybox made...
 8                   [OK]
```

## Format the output

The formatting option ( `--format` ) pretty-prints search output using a Go template.

Valid placeholders for the Go template are:

| Placeholder | Description |
|---|---|
| `.Name` | Image Name |
| `.Description` | Image description |
| `.StarCount` | Number of stars for the image |
| `.IsOfficial` | “OK” if image is official |
| `.IsAutomated` | “OK” if image build was automated |

When you use the `--format` option, the `search` command will output the data exactly as the template declares. If you use the `table` directive, column headers are included as well.

The following example uses a template without headers and outputs the `Name` and `StarCount` entries separated by a colon for all images:

```
{% raw %}
$ docker search --format "{{.Name}}: {{.StarCount}}" nginx

nginx: 5441
jwilder/nginx-proxy: 953
richarvey/nginx-php-fpm: 353
million12/nginx-php: 75
webdevops/php-nginx: 70
h3nrik/nginx-ldap: 35
bitnami/nginx: 23
evild/alpine-nginx: 14
million12/nginx: 9
maxexcloo/nginx: 7
{% endraw %}
```

This example outputs a table format:

```
{% raw %}
$ docker search --format "table {{.Name}}\t{{.IsAutomated}}\t{{.IsOf
ficial}}" nginx

NAME                                      AUTOMATED           OFFICIA
L
nginx                                                         [OK]
jwilder/nginx-proxy                       [OK]
richarvey/nginx-php-fpm                   [OK]
jrcs/letsencrypt-nginx-proxy-companion    [OK]
million12/nginx-php                       [OK]
webdevops/php-nginx                       [OK]
{% endraw %}
```

# docker image tag

*Estimated reading time: 1 minute*

## Description

Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE

## Usage

```
docker image tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]
```

## Parent command

| Command | Description |
|---|---|
| docker image (https://docs.docker.com/engine/reference/commandline/image) | Manage images |

## Related commands

| Command | Description |
|---|---|
| docker image build (https://docs.docker.com/engine/reference/commandline/image_build/) | Build an image from a Dockerfile |
| docker image history (https://docs.docker.com/engine/reference/commandline/image_history/) | Show the history of an image |

| Command | Description |
| --- | --- |
| docker image import (https://docs.docker.com/engine/reference/commandline/image_import/) | Import the contents from a tarball to create a filesystem image |
| docker image inspect (https://docs.docker.com/engine/reference/commandline/image_inspect/) | Display detailed information on one or more images |
| docker image load (https://docs.docker.com/engine/reference/commandline/image_load/) | Load an image from a tar archive or STDIN |
| docker image ls (https://docs.docker.com/engine/reference/commandline/image_ls/) | List images |
| docker image prune (https://docs.docker.com/engine/reference/commandline/image_prune/) | Remove unused images |
| docker image pull (https://docs.docker.com/engine/reference/commandline/image_pull/) | Pull an image or a repository from a registry |
| docker image push (https://docs.docker.com/engine/reference/commandline/image_push/) | Push an image or a repository to a registry |
| docker image rm (https://docs.docker.com/engine/reference/commandline/image_rm/) | Remove one or more images |
| docker image save (https://docs.docker.com/engine/reference/commandline/image_save/) | Save one or more images to a tar archive (streamed to STDOUT by default) |

| Command | Description |
| --- | --- |
| docker image tag (https://docs.docker.com/engine/reference/commandline/image_tag/) | Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE |

# Content trust in Docker

*Estimated reading time: 13 minutes*

When transferring data among networked systems, *trust* is a central concern. In particular, when communicating over an untrusted medium such as the internet, it is critical to ensure the integrity and the publisher of all the data a system operates on. You use Docker Engine to push and pull images (data) to a public or private registry. Content trust gives you the ability to verify both the integrity and the publisher of all the data received from a registry over any channel.

## About trust in Docker

Docker Content Trust (DCT) allows operations with a remote Docker registry to enforce client-side signing and verification of image tags. DCT provides the ability to use digital signatures for data sent to and received from remote Docker registries. These signatures allow client-side verification of the integrity and publisher of specific image tags.

Once DCT is enabled, image publishers can sign their images. Image consumers can ensure that the images they use are signed. Publishers and consumers can either be individuals or organizations. DCT supports users and automated processes such as builds.

When you enable DCT, signing occurs on the client after push and verification happens on the client after pull if you use Docker CE. If you use UCP, and you have configured UCP to require images to be signed before deploying, signing is verified by UCP.

### Image tags and DCT

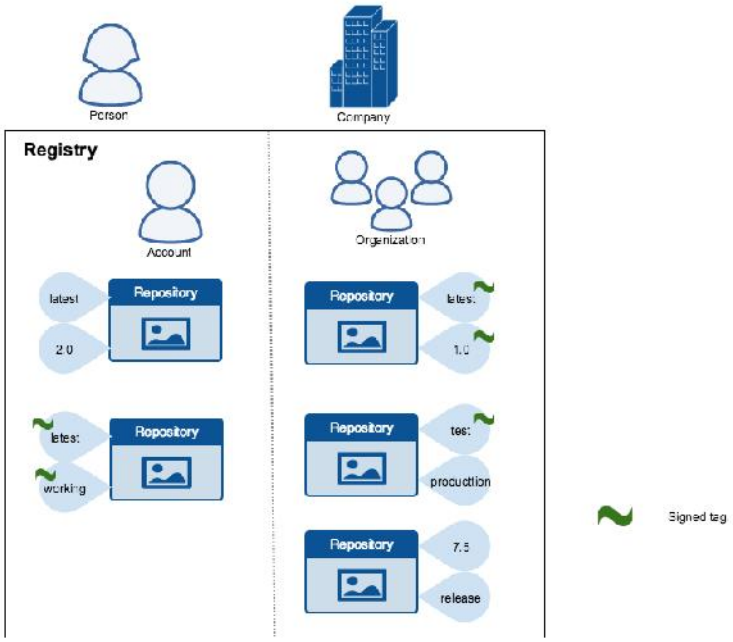
An individual image record has the following identifier:

```
[REGISTRY_HOST[:REGISTRY_PORT]/]REPOSITORY[:TAG]
```

A particular image `REPOSITORY` can have multiple tags. For example, `latest` and `3.1.2` are both tags on the `mongo` image. An image publisher can build an image and tag combination many times changing the image with each build.

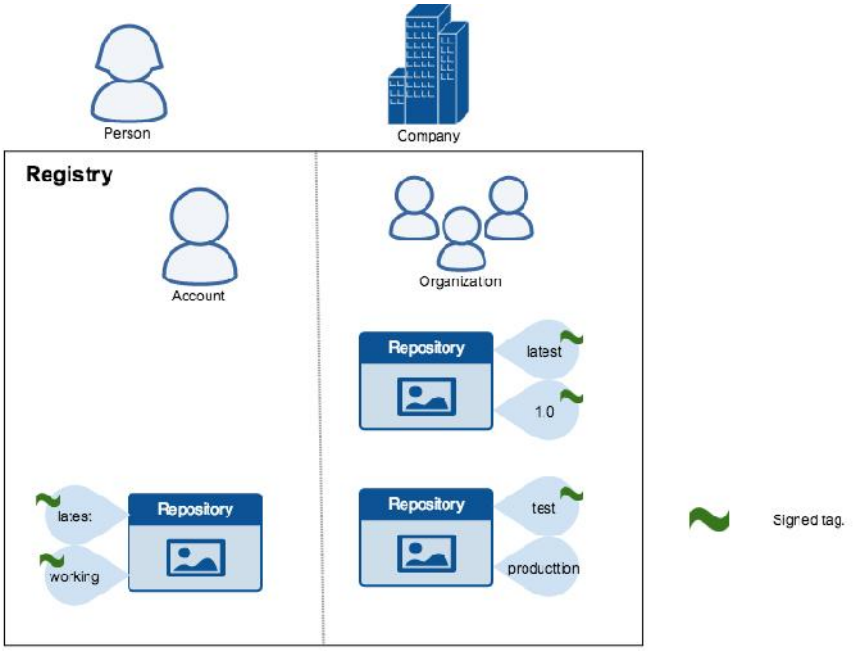
DCT is associated with the `TAG` portion of an image. Each image repository has a set of keys that image publishers use to sign an image tag. Image publishers have discretion on which tags they sign.

An image repository can contain an image with one tag that is signed and another tag that is not. For example, consider the Mongo image repository (https://hub.docker.com/r/library/mongo/tags/). The `latest` tag could be unsigned while the `3.1.6` tag could be signed. It is the responsibility of the image publisher to decide if an image tag is signed or not. In this representation, some image tags are signed, others are not:




Publishers can choose to sign a specific tag or not. As a result, the content of an unsigned tag and that of a signed tag with the same name may not match. For example, a publisher can push a tagged image `someimage:latest` and sign it. Later, the same publisher can push an unsigned `someimage:latest` image. This second push replaces the last unsigned tag `latest` but does not affect the signed `latest` version. The ability to choose which tags they can sign, allows publishers to iterate over the unsigned version of an image before officially signing it.

Image consumers can enable DCT to ensure that images they use were signed. If a consumer enables DCT, they can only pull, run, or build with trusted images. Enabling DCT is like wearing a pair of rose-colored glasses. Consumers "see" only signed image tags and the less desirable, unsigned image tags are "invisible" to them.




To the consumer who has not enabled DCT, nothing about how they work with Docker images changes. Every image is visible regardless of whether it is signed or not.

## DCT operations and keys

When DCT is enabled, `docker` CLI commands that operate on tagged images must either have content signatures or explicit content hashes. The commands that operate with DCT are:

- `push`
- `build`
- `create`
- `pull`
- `run`

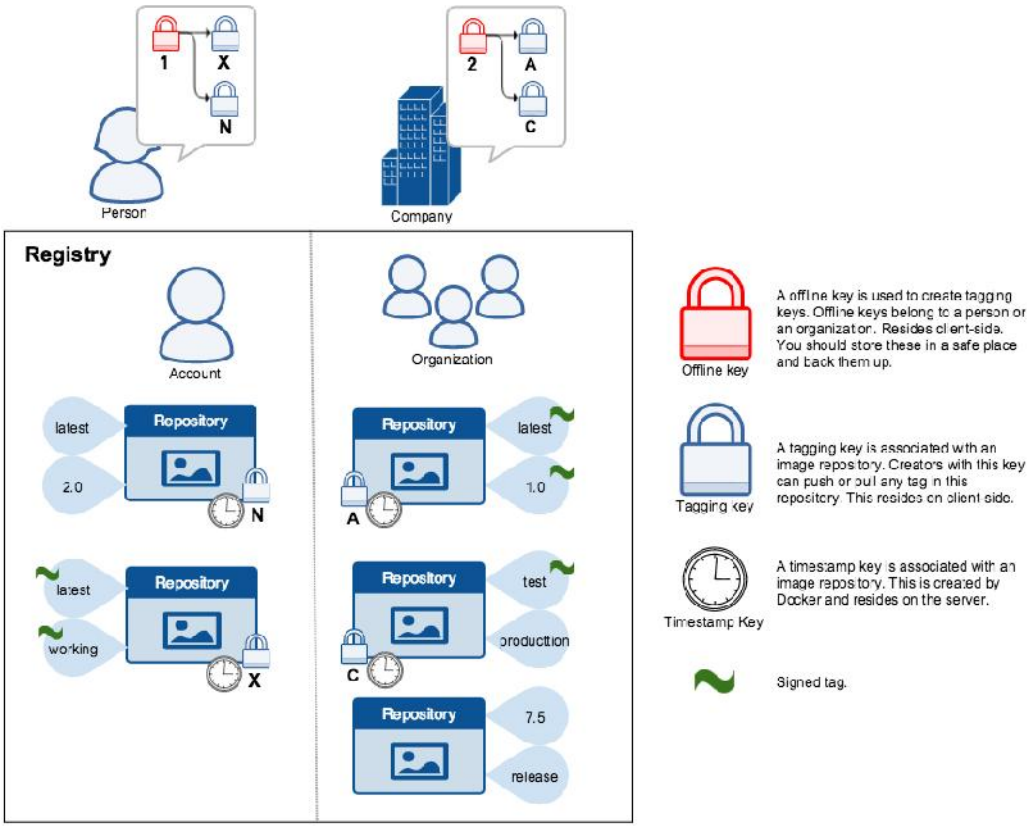
For example, with DCT enabled a `docker pull someimage:latest` only succeeds if `someimage:latest` is signed. However, an operation with an explicit content hash always succeeds as long as the hash exists:

```
$ docker pull someimage@sha256:d149ab53f8718e987c3a3024bb8aa0e2caadf
6c0328f1d9d850b2a2a67f2819a
```

Trust for an image tag is managed through the use of signing keys. A key set is created when an operation using DCT is first invoked. A key set consists of the following classes of keys:

- an offline key that is the root of DCT for an image tag
- repository or tagging keys that sign tags
- server-managed keys such as the timestamp key, which provides freshness security guarantees for your repository

The following image depicts the various signing keys and their relationships:




> ✖ WARNING: Loss of the root key is very difficult to recover from. Correcting this loss requires intervention from Docker Support (https://support.docker.com) to reset the repository state. This loss also requires manual intervention from every consumer that used a signed tag from this repository prior to the loss.

You should backup the root key somewhere safe. Given that it is only required to create new repositories, it is a good idea to store it offline in hardware. For details on securing, and backing up your keys, make sure you read how to manage keys for DCT (https://docs.docker.com/engine/security/trust/trust_key_mng/).

# Survey of typical DCT operations

This section surveys the typical trusted operations users perform with Docker images. Specifically, we go through the following steps to help us exercise these various trusted operations:

- Build and push an unsigned image
- Pull an unsigned image
- Build and push a signed image
- Pull the signed image pushed above
- Pull unsigned image pushed above

## Enabling DCT in Docker Engine Configuration

Engine Signature Verification prevents the following behaviors on an image:

- Running a container to build an image (the base image must be signed, or must be scratch)
- Creating a container from an image that is not signed

DCT does not verify that a running container's filesystem has not been altered from what was in the image. For example, it does not prevent a container from writing to the filesystem, nor the container's filesystem from being altered on disk.

It will also pull and run signed images from registries, but will not prevent unsigned images from being imported, loaded, or created.

The image name, digest, or tag must be verified if DCT is enabled. The latest DCT metadata for an image must be downloaded from the trust server associated with the registry:

- If an image tag does not have a digest, the DCT metadata translates the name to an image digest
- If an image tag has an image digest, the DCT metadata verifies that the name matches the provided digest
- If an image digest does not have an image tag, the DCT metadata does a reverse lookup and provides the image tag as well as the digest.

The signature verification feature is configured in the Docker daemon

configuration file `daemon.json` .

```
{
    ...
    “content-trust”: {
        “trust-pinning”: {
            “root-keys”: {
                “myregistry.com/myorg/*”: [“keyID1”, “keyID2”],
                “myregistry.com/otherorg/repo”: [“keyID3”]
            },
            “official-images”: true,
        },
        “mode”: “disabled” | “permissive” | “enforced”,
        “allow-expired-trust-cache”: true,
    }
}
```

| Stanza | Description |
| --- | --- |
| `trust-pinning:root-keys` | Root key IDs are canonical IDs that sign the root metadata of the image trust data. In Docker Certified Trust (DCT), the root keys are unique certificates tying the name of the image to the repo metadata. The private key ID (the canonical key ID) corresponding to the certificate does not depend on the image name. If an image’s name matches more than one glob, then the most specific (longest) one is chosen. |
| `trust-pinning:library-images` | This option pins the official libraries ( `docker.io/library/*` ) to the hard-coded Docker official images root key. DCT trusts the official images by default. This is in addition to whatever images are specified by `trust-pinning:root-keys` . If `trustpinning:root-keys` specifies a key mapping for `docker.io/library/*` , those keys will be preferred for trust pinning. Otherwise, if a more general `docker.io/*` or `*` are specified, the official images key will be preferred. |

| Stanza | Description |
| --- | --- |
| `allow-expired-trust-cache` | Specifies whether cached locally expired metadata validates images if an external server is unreachable or does not have image trust metadata. This is necessary for machines which may be often offline, as may be the case for edge. This does not provide mitigations against freeze attacks, which is a necessary to provide availability in low-connectivity environments. |
| `mode` | Specifies whether DCT is enabled and enforced. Valid modes are: `disabled` : Verification is not active and the remainder of the content-trust related metadata will be ignored. *NOTE* that this is the default configuration if `mode` is not specified. `permissive` : Verification will be performed, but only failures will only be logged and remain unenforced. This configuration is intended for testing of changes related to content-trust. `enforced` : DCT will be enforced and an image that cannot be verified successfully will not be pulled or run. |

*Note:* The DCT configuration defined here is agnostic of any policy defined in UCP (https://docs.docker.com/v17.09/datacenter/ucp/2.0/guides/content-trust/#configure-ucp). Images that can be deployed by the UCP trust policy but are disallowed by the Docker Engine configuration will not successfully be deployed or run on that engine.

## Enable and disable DCT per-shell or per-invocation

Instead of enabling DCT through the system-wide configuration, DCT can be enabled or disabled on a per-shell or per-invocation basis.

To enable on a per-shell basis, enable the `DOCKER_CONTENT_TRUST` environment variable. Enabling per-shell is useful because you can have one shell configured for trusted operations and another terminal shell for untrusted operations. You can also add this declaration to your shell profile to have it enabled by default.

To enable DCT in a `bash` shell enter the following command:

```
export DOCKER_CONTENT_TRUST=1
```

Once set, each of the "tag" operations requires a key for a trusted tag.

In an environment where `DOCKER_CONTENT_TRUST` is set, you can use the `--disable-content-trust` flag to run individual operations on tagged images without DCT on an as-needed basis.

Consider the following Dockerfile that uses an untrusted parent image:

```
$ cat Dockerfile
FROM docker/trusttest:latest
RUN echo
```

To build a container successfully using this Dockerfile, one can do:

```
$ docker build --disable-content-trust -t <username>/nottrusttest:l
atest .
Sending build context to Docker daemon 42.84 MB
...
Successfully built f21b872447dc
```

The same is true for all the other commands, such as `pull` and `push` :

```
$ docker pull --disable-content-trust docker/trusttest:latest
...
$ docker push --disable-content-trust <username>/nottrusttest:lates
t
...
```

To invoke a command with DCT enabled regardless of whether or how the `DOCKER_CONTENT_TRUST` variable is set:

```
$ docker build --disable-content-trust=false -t <username>/trusttes
t:testing .
```

All of the trusted operations support the `--disable-content-trust` flag.

## Push trusted content

To create signed content for a specific image tag, simply enable DCT and push a tagged image. If this is the first time you have pushed an image using DCT on your system, the session looks like this:

```
$ docker push <username>/trusttest:testing
The push refers to a repository [docker.io/<username>/trusttest] (le
n: 1)
9a61b6b1315e: Image already exists
902b87aaaec9: Image already exists
latest: digest: sha256:d02adacee0ac7a5be140adb94fa1dae64f4e71a68696e
7f8e7cbf9db8dd49418 size: 3220
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This pass
phrase
will be used to protect the most sensitive key in your signing syste
m. Please
choose a long, complex passphrase and be careful to keep the passwor
d and the
key file itself secure and backed up. It is highly recommended that
you use a
password manager to generate the passphrase and keep it safe. There
will be no
way to recover this key. You can find the key in your config directo
ry.
Enter passphrase for new root key with id a1d96fb:
Repeat passphrase for new root key with id a1d96fb:
Enter passphrase for new repository key with id docker.io/<username>
/trusttest (3a932f1):
Repeat passphrase for new repository key with id docker.io/<username
>/trusttest (3a932f1):
Finished initializing "docker.io/<username>/trusttest"
```

When you push your first tagged image with DCT enabled, the `docker` client recognizes this is your first push and:

- alerts you that it is creating a new root key
- requests a passphrase for the root key
- generates a root key in the `~/.docker/trust` directory
- requests a passphrase for the repository key
- generates a repository key in the `~/.docker/trust` directory

The passphrase you chose for both the root key and your repository key-pair should be randomly generated and stored in a *password manager*.

> NOTE: If you omit the `testing` tag, DCT is skipped. This is true even if DCT is enabled and even if this is your first push.

```
$ docker push <username>/trusttest
The push refers to a repository [docker.io/<username>/trusttest] (le
n: 1)
9a61b6b1315e: Image successfully pushed
902b87aaaec9: Image successfully pushed
latest: digest: sha256:a9a9c4402604b703bed1c847f6d85faac97686e48c579
bd9c3b0fa6694a398fc size: 3220
No tag specified, skipping trust metadata push
```

It is skipped because as the message states, you did not supply an image `TAG` value. In DCT, signatures are associated with tags.

Once you have a root key on your system, subsequent images repositories you create can use that same root key:

```
$ docker push docker.io/<username>/otherimage:latest
The push refers to a repository [docker.io/<username>/otherimage] (·
en: 1)
a9539b34a6ab: Image successfully pushed
b3dbab3810fc: Image successfully pushed
latest: digest: sha256:d2ba1e603661a59940bfad7072eba698b79a8b20ccbb4
e3bfb6f9e367ea43939 size: 3346
Signing and pushing trust metadata
Enter key passphrase for root key with id a1d96fb:
Enter passphrase for new repository key with id docker.io/<username>
/otherimage (bb045e3):
Repeat passphrase for new repository key with id docker.io/<username
>/otherimage (bb045e3):
Finished initializing "docker.io/<username>/otherimage"
```

The new image has its own repository key and timestamp key. The `latest` tag is signed with both of these.

## Pull image content

A common way to consume an image is to `pull` it. With DCT enabled, the Docker client only allows `docker pull` to retrieve signed images. Let's try to pull the image you signed and pushed earlier:

```
$  docker pull <username>/trusttest:testing
Pull (1 of 1): <username>/trusttest:testing@sha256:d149ab53f871
...
Tagging <username>/trusttest@sha256:d149ab53f871 as docker/trusttest
:testing
```

In the following example, the command does not specify a tag, so the system uses the `latest` tag by default again and the `docker/trusttest:latest` tag is not signed.

```
$ docker pull docker/trusttest
Using default tag: latest
no trust data available
```

Because the tag `docker/trusttest:latest` is not trusted, the `pull` fails.

# Related information

- Manage keys for content trust (https://docs.docker.com/engine/security/trust/trust_key_mng/)
- Automation with content trust (https://docs.docker.com/engine/security/trust/trust_automation/)
- Delegations for content trust (https://docs.docker.com/engine/security/trust/trust_delegation/)
- Play in a content trust sandbox (https://docs.docker.com/engine/security/trust/trust_sandbox/)

content (https://docs.docker.com/glossary/?term=content), trust (https://docs.docker.com/glossary/?term=trust), security (https://docs.docker.com/glossary/?term=security), docker (https://docs.docker.com/glossary/?term=docker), documentation (https://docs.docker.com/glossary/?term=documentation)

# docker pull

*Estimated reading time: 8 minutes*

## Description

Pull an image or a repository from a registry

## Usage

```
docker pull [OPTIONS] NAME[:TAG|@DIGEST]
```

## Options

| Name, shorthand | Default | Description |
|---|---|---|
| `--all-tags , -a` | | Download all tagged images in the repository |
| `--disable-content-trust` | `true` | Skip image verification |
| `--platform` | | experimental (daemon) (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file) API 1.32+ (https://docs.docker.com/engine/api/v1.32/) Set platform if server is multi-platform capable |

## Parent command

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

## Extended description

Most of your images will be created on top of a base image from the Docker Hub (https://hub.docker.com) registry.

Docker Hub (https://hub.docker.com) contains many pre-built images that you can `pull` and try without needing to define and configure your own.

To download a particular image, or set of images (i.e., a repository), use `docker pull` .

### Proxy configuration

If you are behind an HTTP proxy server, for example in corporate settings, before open a connect to registry, you may need to configure the Docker daemon's proxy settings, using the `HTTP_PROXY`, `HTTPS_PROXY`, and `NO_PROXY` environment variables. To set these environment variables on a host using `systemd`, refer to the control and configure Docker with systemd (https://docs.docker.com/engine/admin/systemd/#http-proxy) for variables configuration.

### Concurrent downloads

By default the Docker daemon will pull three layers of an image at a time. If you are on a low bandwidth connection this may cause timeout issues and you may want to lower this via the `--max-concurrent-downloads` daemon option. See the daemon documentation (https://docs.docker.com/engine/reference/commandline/dockerd/) for more details.

## Examples

### Pull an image from Docker Hub

To download a particular image, or set of images (i.e., a repository), use `docker pull`. If no tag is provided, Docker Engine uses the `:latest` tag as a default. This command pulls the `debian:latest` image:

```
$ docker pull debian

Using default tag: latest
latest: Pulling from library/debian
fdd5d7827f33: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:e7d38b3517548a1c71e41bffe9c8ae6d6d29546ce46bf62159837aad072c90aa
Status: Downloaded newer image for debian:latest
```

Docker images can consist of multiple layers. In the example above, the image consists of two layers; `fdd5d7827f33` and `a3ed95caeb02`.

Layers can be reused by images. For example, the `debian:jessie` image shares both layers with `debian:latest`. Pulling the `debian:jessie` image therefore only pulls its metadata, but not its layers, because all layers are already present locally:

```
$ docker pull debian:jessie

jessie: Pulling from library/debian
fdd5d7827f33: Already exists
a3ed95caeb02: Already exists
Digest: sha256:a9c958be96d7d40df920e7041608f2f017af81800ca5ad23e327bc402626b58e
Status: Downloaded newer image for debian:jessie
```

To see which images are present locally, use the `docker images` (https://docs.docker.com/engine/reference/commandline/images/) command:

```
$ docker images

REPOSITORY   TAG      IMAGE ID       CREATED      SIZE
debian       jessie   f50f9524513f   5 days ago   125.1 MB
debian       latest   f50f9524513f   5 days ago   125.1 MB
```

Docker uses a content-addressable image store, and the image ID is a SHA256 digest covering the image's configuration and layers. In the example above, `debian:jessie` and `debian:latest` have the same image ID because they are actually the *same* image tagged with different names. Because they are the same image, their layers are stored only once and do not consume extra disk space.

For more information about images, layers, and the content-addressable store, refer to understand images, containers, and storage drivers (https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/).

## Pull an image by digest (immutable identifier)

So far, you've pulled images by their name (and "tag"). Using names and tags is a convenient way to work with images. When using tags, you can `docker pull` an image again to make sure you have the most up-to-date version of that image. For example, `docker pull ubuntu:14.04` pulls the latest version of the Ubuntu 14.04 image.

In some cases you don't want images to be updated to newer versions, but prefer to use a fixed version of an image. Docker enables you to pull an image by its *digest*. When pulling an image by digest, you specify *exactly* which version of an image to pull. Doing so, allows you to "pin" an image to that version, and guarantee that the image you're using is always the same.

To know the digest of an image, pull the image first. Let's pull the latest `ubuntu:14.04` image from Docker Hub:

```
$ docker pull ubuntu:14.04

14.04: Pulling from library/ubuntu
5a132a7e7af1: Pull complete
fd2731e4c50c: Pull complete
28a2f68d1120: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2
Status: Downloaded newer image for ubuntu:14.04
```

Docker prints the digest of the image after the pull has finished. In the example above, the digest of the image is:

```
sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2
```

Docker also prints the digest of an image when *pushing* to a registry. This may be useful if you want to pin to a version of the image you just pushed.

A digest takes the place of the tag when pulling an image, for example, to pull the above image by digest, run the following command:

```
$ docker pull ubuntu@sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2

sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2: Pulling from library/u
buntu
5a132a7e7af1: Already exists
fd2731e4c50c: Already exists
28a2f68d1120: Already exists
a3ed95caeb02: Already exists
Digest: sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2
Status: Downloaded newer image for ubuntu@sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c1
9722f87876677c5cb2
```

Digest can also be used in the `FROM` of a Dockerfile, for example:

```
FROM ubuntu@sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2
MAINTAINER some maintainer <maintainer@example.com>
```

> Note: Using this feature "pins" an image to a specific version in time. Docker will therefore not pull updated versions of an image, which may include security updates. If you want to pull an updated image, you need to change the digest accordingly.

## Pull from a different registry

By default, `docker pull` pulls images from Docker Hub (https://hub.docker.com). It is also possible to manually specify the path of a registry to pull from. For example, if you have set up a local registry, you can specify its path to pull from it. A registry path is similar to a URL, but does not contain a protocol specifier ( `https://` ).

The following command pulls the `testing/test-image` image from a local registry listening on port 5000 ( `myregistry.local:5000` ):

```
$ docker pull myregistry.local:5000/testing/test-image
```

Registry credentials are managed by docker login (https://docs.docker.com/engine/reference/commandline/login/).

Docker uses the `https://` protocol to communicate with a registry, unless the registry is allowed to be accessed over an insecure connection. Refer to the insecure registries (https://docs.docker.com/engine/reference/commandline/dockerd/#insecure-registries) section for more information.

## Pull a repository with multiple images

By default, `docker pull` pulls a *single* image from the registry. A repository can contain multiple images. To pull all images from a repository, provide the `-a` (or `--all-tags` ) option when using `docker pull` .

This command pulls all images from the `fedora` repository:

```
$ docker pull --all-tags fedora

Pulling repository fedora
ad57ef8d78d7: Download complete
105182bb5e8b: Download complete
511136ea3c5a: Download complete
73bd853d2ea5: Download complete
....

Status: Downloaded newer image for fedora
```

After the pull has completed use the `docker images` command to see the images that were pulled. The example below shows all the `fedora` images that are present locally:

```
$ docker images fedora

REPOSITORY   TAG         IMAGE ID        CREATED      SIZE
fedora       rawhide     ad57ef8d78d7    5 days ago   359.3 MB
fedora       20          105182bb5e8b    5 days ago   372.7 MB
fedora       heisenbug   105182bb5e8b    5 days ago   372.7 MB
fedora       latest      105182bb5e8b    5 days ago   372.7 MB
```

## Cancel a pull

Killing the `docker pull` process, for example by pressing `CTRL-C` while it is running in a terminal, will terminate the pull operation.

```
$ docker pull fedora

Using default tag: latest
latest: Pulling from library/fedora
a3ed95caeb02: Pulling fs layer
236608c7b546: Pulling fs layer
^C
```

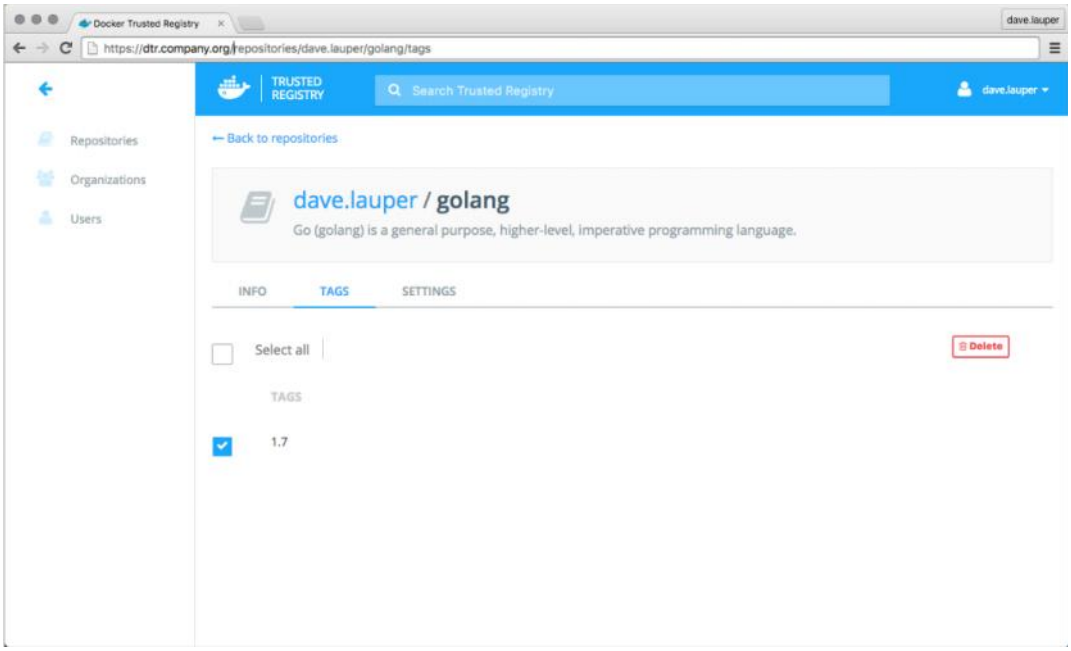
> Note: Technically, the Engine terminates a pull operation when the connection between the Docker Engine daemon and the Docker Engine client initiating the pull is lost. If the connection with the Engine daemon is lost for other reasons than a manual interaction, the pull is also aborted.

# Delete an image

*Estimated reading time: 1 minute*

> These are the docs for DTR version 2.0
>
> To select a different version, use the selector below.
>
> 2.0

To delete an image, go to the DTR web UI, and navigate to the image repository you want to delete. In the Tags tab, select all the image tags you want to delete, and click the Delete button.




You can also delete all image versions, by deleting the repository. For that, in the image repository, navigate to the Settings tab, and click the Delete button.

docker (https://docs.docker.com/glossary/?term=docker), registry (https://docs.docker.com/glossary/?term=registry), repository (https://docs.docker.com/glossary/?term=repository), delete (https://docs.docker.com/glossary/?term=delete), image (https://docs.docker.com/glossary/?term=image)