

 New Building AI Agents is Now Easy → [Home](#) / [Reference](#) / Dockerfile reference

# Dockerfile reference

## Table of contents

[Overview](#)[Format](#)[Parser directives](#)[syntax](#)[escape](#)[check](#)[Environment replacement](#)[.dockerignore file](#)[Shell and exec form](#)[Exec form](#)[Shell form](#)[Use a different shell](#)[FROM](#)[Understand how ARG and FROM interact](#)

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

[Cookies Settings](#)[Reject All](#)[Accept All Cookies](#)

```
RUN --mount=type=tmpfs  
RUN --mount=type=secret  
RUN --mount=type=ssh  
RUN --network  
RUN --network=default  
RUN --network=none  
RUN --network=host  
RUN --security  
CMD  
LABEL  
MAINTAINER (deprecated)  
EXPOSE  
ENV  
ADD  
Source  
Destination  
ADD --keep-git-dir  
ADD --checksum  
ADD --chown --chmod  
ADD --link  
ADD --exclude
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

COPY --parents

COPY --exclude

ENTRYPOINT

Exec form ENTRYPOINT example

Shell form ENTRYPOINT example

Understand how CMD and ENTRYPOINT interact

VOLUME

Notes about specifying volumes

USER

WORKDIR

ARG

Default values

Scope

Using ARG variables

Predefined ARGs

Automatic platform ARGs in the global scope

BuildKit built-in build args

Impact on build caching

ONBUILD

Copy or mount from stage, image, or context

ONBUILD limitations

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

## Dockerfile examples

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. This page describes the commands you can use in a Dockerfile.

## Overview

The Dockerfile supports the following instructions:

Instruction	Description
ADD	Add local or remote files and directories.
ARG	Use build-time variables.
CMD	Specify default commands.
COPY	Copy files and directories.
ENTRYPOINT	Specify default executable.
ENV	Set environment variables.
EXPOSE	Describe which ports your application is listening on.
FROM	Create a new build stage from a base image.
HEALTHCHECK	Check a container's health on startup.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

Instruction	Description
STOPSIGNAL	Specify the system call signal for exiting a container.
USER	Set user and group ID.
VOLUME	Create volume mounts.
WORKDIR	Change working directory.

## 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 begin with a FROM instruction**. This may be after **parser directives**, **comments**, and globally scoped **ARGs**. The **FROM** instruction specifies the **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.

BuildKit treats lines that begin with **#** as a comment, unless the line is a valid **parser directive**. A **#** marker anywhere else in a line is treated as an argument. This allows statements like:

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
RUN echo hello \
# comment
world
```

The following examples is equivalent.

```
RUN echo hello \
world
```

Comments don't support line continuation characters.

## Note

### Note on whitespace

For backward compatibility, leading whitespace before comments (#) and instructions (such as RUN) are ignored, but discouraged. Leading whitespace is not preserved in these cases, and the following examples are therefore equivalent:

```
# this is a comment-line
RUN echo hello
RUN echo world
-----
```

```
# this is a comment-line
RUN echo hello
-----
```

Whitespace in instruction arguments, however, isn't ignored. The following example

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

# Parser directives

Parser directives are optional, and affect the way in which subsequent lines in a Dockerfile are handled. Parser directives don't add layers to the build, and don't show up as build steps. Parser directives are written as a special type of comment in the form `# directive=value`. A single directive may only be used once.

The following parser directives are supported:

- `syntax`
- `escape`
- `check` (since Dockerfile v1.8.0)

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

Parser directive keys, such as `syntax` or `check`, aren't case-sensitive, but they're lowercase by convention. Values for a directive are case-sensitive and must be written in the appropriate case for the directive. For example, `#check=skip=jsonargs recommended` is invalid because the check name must use Pascal case, not lowercase. It's also conventional to include a blank line following any parser directives. Line continuation characters aren't supported in parser directives.

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

Invalid due to line continuation:

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
FROM ImageName
```

Treated as a comment because it appears after a builder instruction:

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

Treated as a comment because it appears after a comment that isn't a parser directive:

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

The following `unknowndirective` is treated as a comment because it isn't recognized. The known `syntax` directive is treated as a comment because it appears after a comment that isn't a parser directive.

```
# unknowndirective=value  
# syntax=value
```

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

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

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

Use the `syntax` parser directive to declare the Dockerfile syntax version to use for the build. If unspecified, BuildKit uses a bundled version of the Dockerfile frontend. Declaring a syntax version lets you automatically use the latest Dockerfile version without having to upgrade BuildKit or Docker Engine, or even use a custom Dockerfile implementation.

Most users will want to set this parser directive to `docker/dockerfile:1`, which causes BuildKit to pull the latest stable version of the Dockerfile syntax before the build.

```
# syntax=docker/dockerfile:1
```

For more information about how the parser directive works, see [Custom Dockerfile syntax](#).

## escape

```
# escape=\
```

Or

```
# escape=`
```

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`

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

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 E:\myproject> 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 E:\myproject>
```

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=`
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
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 E:\myproject>
```

## check

```
# check=skip=<checks|all>
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
# check=skip=JSONArgsRecommended,StageNameCasing
```

To disable all checks, use `#check=skip=all`.

By default, builds with failing build checks exit with a zero status code despite warnings. To make the build fail on warnings, set `#check=error=true`.

```
# check=error=true
```

### Note

When using the `check` directive, with `error=true` option, it is recommended to pin the [Dockerfile syntax](#) to a specific version. Otherwise, your build may start to fail when new checks are added in the future versions.

To combine both the `skip` and `error` options, use a semi-colon to separate them:

```
# check=skip=JSONArgsRecommended;error=true
```

To see all available checks, see the [build checks reference](#). Note that the checks available depend on the Dockerfile syntax version. To make sure you're getting the most up-to-date checks, use the `syntax` directive to specify the Dockerfile syntax version to the latest stable version.

## Environment replacement

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

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.

The following variable replacements are supported in a pre-release version of Dockerfile syntax, when using the  `# syntax=docker/dockerfile-upstream:master`  syntax directive in your Dockerfile:

- `${variable#pattern}`  removes the shortest match of  `pattern`  from  `variable` , seeking from the start of the string.

```
str=foobarbaz echo ${str#f*b}      # arbaz
```

- `${variable##pattern}`  removes the longest match of  `pattern`  from  `variable` , seeking from the start of the string.

```
str=foobarbaz echo ${str##f*b}      # az
```

- `${variable%pattern}`  removes the shortest match of  `pattern`  from  `variable` , seeking backwards from the end of the string.

```
string=foobarbaz echo ${string%b*}    # foobar
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
string=foobarbaz echo ${string/ba/fo} # fooforbaz
```

- `/${variable//pattern/replacement}` replaces all occurrences of `pattern` in `variable` with `replacement`

```
string=foobarbaz echo ${string//ba/fo} # fooforfoz
```

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

`pattern` is a glob pattern where `?` matches any single character and `*` any number of characters (including zero). To match literal `?` and `*`, use a backslash escape: `\?` and `\*`.

You can escape whole variable names 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`

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

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

You can also use environment variables with `RUN`, `CMD`, and `ENTRYPOINT` instructions, but in those cases the variable substitution is handled by the command shell, not the builder. Note that instructions using the exec form don't invoke a command shell automatically. See [Variable substitution](#).

Environment variable substitution use the same value for each variable throughout the entire instruction. Changing the value of a variable only takes effect in subsequent instructions. Consider the following example:

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

- The value of `def` becomes `hello`
- The value of `ghi` becomes `bye`

## .dockerignore file

You can use `.dockerignore` file to exclude files and directories from the build context. For more information, see [.dockerignore file](#).

## Shell and exec form

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

The shell form is more relaxed, and emphasizes ease of use, flexibility, and readability. The shell form automatically uses a command shell, whereas the exec form does not.

## Exec form

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

```
ENTRYPOINT ["/bin/bash", "-c", "echo hello"]
```

The exec form is best used to specify an `ENTRYPOINT` instruction, combined with `CMD` for setting default arguments that can be overridden at runtime. For more information, see [ENTRYPOINT](#).

### Variable substitution

Using the exec form doesn't automatically invoke a command shell. This means that normal shell processing, such as variable substitution, doesn't happen. For example, `RUN [ "echo", "$HOME" ]` won't handle variable substitution for `$HOME`.

If you want shell processing then either use the shell form or execute a shell directly with the exec form, 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's the shell that's doing the environment variable substitution, not the builder.

### Backslashes

In exec form, you must 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

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
RUN ["c:\\windows\\system32\\tasklist.exe"]
```

## Shell form

Unlike the exec form, instructions using the shell form always use a command shell. The shell form doesn't use the JSON array format, instead it's a regular string. The shell form string lets you escape newlines using the [escape character](#) (backslash by default) to continue a single instruction onto the next line. This makes it easier to use with longer commands, because it lets you split them up into multiple lines. For example, consider these two lines:

```
RUN source $HOME/.bashrc && \
echo $HOME
```

They're equivalent to the following line:

```
RUN source $HOME/.bashrc && echo $HOME
```

You can also use heredocs with the shell form to break up supported commands.

```
RUN <<EOF
source $HOME/.bashrc && \
echo $HOME
EOF
```

For more information about heredocs, see [Here-documents](#).

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

For more information, see [SHELL](#).

## FROM

```
FROM [--platform=<platform>] <image> [AS <name>]
```

Or

```
FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]
```

Or

```
FROM [--platform=<platform>] <image>[@<digest>] [AS <name>]
```

The `FROM` instruction initializes a new build stage and sets the [base image](#) for subsequent instructions. As such, a valid Dockerfile must start with a `FROM` instruction. The image can be any valid image.

- `ARG` is the only instruction that may precede `FROM` in the Dockerfile. See [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 <names>`, `COPY from-<names>`,

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

`windows/amd64`. By default, the target platform of the build request is used. Global build arguments can be used in the value of this flag, for example `automatic platform ARGs` allow you to force a stage to native build platform (`--platform=$BUILDPLATFORM`), and use it to cross-compile to the target platform inside the stage.

## 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

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
# Shell form:  
-----  
RUN [OPTIONS] <command> ...  
  
# Exec form:  
-----  
RUN [OPTIONS] [ "<command>", ... ]
```

For more information about the differences between these two forms, see [shell or exec forms](#).

The shell form is most commonly used, and lets you break up longer instructions into multiple lines, either using newline [escapes](#), or with [heredocs](#):

```
RUN <<EOF  
-----  
apt-get update  
apt-get install -y curl  
EOF
```

The available `[OPTIONS]` for the `RUN` instruction are:

Option	Minimum Dockerfile version
--device	1.14-labs
--mount	1.2
--network	1.3
--security	1.1.2-labs

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

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

## RUN --device



### Note

Not yet available in stable syntax, use `docker/dockerfile:1-labs` version. It also needs BuildKit 0.20.0 or later.

```
RUN --device=name, [required]
```

`RUN --device` allows build to request CDI devices to be available to the build step.

The device `name` is provided by the CDI specification registered in BuildKit.

In the following example, multiple devices are registered in the CDI specification for the `vendor1.com/device` vendor.

```
cdiVersion: "0.6.0"
kind: "vendor1.com/device"
devices:
  - name: foo
    containerEdits:
      env:
        - FOO=injected
  - name: bar
    annotations:
      org.mobvproject.buildkit.device.class: class1
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
- BAZ=injected
- name: qux
  annotations:
    org.mobyproject.buildkit.device.class: class2
  containerEdits:
    env:
      - QUX=injected
```

The device name format is flexible and accepts various patterns to support multiple device configurations:

- `vendor1.com/device`: request the first device found for this vendor
- `vendor1.com/device=foo`: request a specific device
- `vendor1.com/device==*`: request all devices for this vendor
- `class1`: request devices by `org.mobyproject.buildkit.device.class` annotation

### Example: CUDA-Powered LLaMA Inference

In this example we use the `--device` flag to run `llama.cpp` inference using an NVIDIA GPU device through CDI:

```
# syntax=docker/dockerfile:1-labs

FROM scratch AS model
ADD https://huggingface.co/bartowski/Llama-3.2-1B-Instruct-GGUF/resolve/main/llama.cpp /llama.cpp

FROM scratch AS prompt
COPY <<EOF prompt.txt
On Generate a list of 10 unique biggest countries by population in JSON with
EOF
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
--mount=from=model,target=/models \
--mount=from=prompt,target=/tmp \
./llama-cli -m /models/model.gguf -no-cnv -nlg 99 -f /tmp/prompt.txt
```

## RUN --mount

```
RUN --mount=[type=TYPE] [,option=<value>[,option=<value>]...]
```

`RUN --mount` allows you to create filesystem mounts that the build can access. This can be used to:

- Create bind mount to the host filesystem or other build stages
- Access build secrets or ssh-agent sockets
- Use a persistent package management cache to speed up your build

The supported mount types are:

Type	Description
<code>bind</code> (default)	Bind-mount context directories (read-only).
<code>cache</code>	Mount a temporary directory to cache directories for compilers and package managers.
<code>tmpfs</code>	Mount a <code>tmpfs</code> in the build container.
<code>secret</code>	Allow the build container to access secure files such as private keys without baking them into the image or build cache.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

This mount type allows binding files or directories to the build container. A bind mount is read-only by default.

Option	Description
<code>target</code> , <code>dst</code> , <code>destination</code>	Mount path.
<code>source</code>	Source path in the <code>from</code> . Defaults to the root of the <code>from</code> .
<code>from</code>	Build stage, context, or image name for the root of the source. Defaults to the build context.
<code>rw</code> , <code>readwrite</code>	Allow writes on the mount. Written data will be discarded.

## RUN --mount=type=cache

This mount type allows the build container to cache directories for compilers and package managers.

Option	Description
<code>id</code>	Optional ID to identify separate/different caches. Defaults to value of <code>target</code> .
<code>target</code> , <code>dst</code> , <code>destination</code>	Mount path.
<code>ro</code> , <code>readonly</code>	Read-only if set.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

Option	Description
<code>mode</code>	File mode for new cache directory in octal. Default <code>0755</code> .
<code>uid</code>	User ID for new cache directory. Default <code>0</code> .
<code>gid</code>	Group ID for new cache directory. Default <code>0</code> .

Contents of the cache directories persists between builder invocations without invalidating the instruction cache. Cache mounts should only be used for better performance. Your build should work with any contents of the cache directory as another build may overwrite the files or GC may clean it if more storage space is needed.

### Example: cache Go packages

```
# syntax=docker/dockerfile:1
FROM golang
RUN --mount=type=cache,target=/root/.cache/go-build \
-----
go build ...
```

### Example: cache apt packages

```
# syntax=docker/dockerfile:1
FROM ubuntu
RUN rm -f /etc/apt/apt.conf.d/docker-clean; echo 'Binary::apt::APT::Keep-Dov
-----
RUN --mount=type=cache,target=/var/cache/apt,sharing=locked \
--mount=type=cache,target=/var/lib/apt,sharing=locked \
apt update && apt-get --no-install-recommends install -y gcc
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

This mount type allows mounting `tmpfs` in the build container.

Option	Description
<code>target</code> , <code>dst</code> , <code>destination</code> <sup>1</sup>	Mount path.
<code>size</code>	Specify an upper limit on the size of the filesystem.

## RUN --mount=type=secret

This mount type allows the build container to access secret values, such as tokens or private keys, without baking them into the image.

By default, the secret is mounted as a file. You can also mount the secret as an environment variable by setting the `env` option.

Option	Description
<code>id</code>	ID of the secret. Defaults to basename of the target path.
<code>target</code> , <code>dst</code> , <code>destination</code>	Mount the secret to the specified path. Defaults to <code>/run/secrets/</code> + <code>id</code> if unset and if <code>env</code> is also unset.
<code>env</code>	Mount the secret to an environment variable instead of a file, or both. (since Dockerfile v1.10.0)
<code>required</code>	If set to <code>true</code> , the instruction errors out when the secret is unavailable. Defaults to <code>false</code> .

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
# syntax=docker/dockerfile:1
FROM python:3
RUN pip install awscli
RUN --mount=type=secret,id=aws,target=/root/.aws/credentials \
aws s3 cp s3://... ...
```

```
$ docker buildx build --secret id=aws,src=$HOME/.aws/credentials .
```

### Example: Mount as environment variable

The following example takes the secret `API_KEY` and mounts it as an environment variable with the same name.

```
# syntax=docker/dockerfile:1
FROM alpine
RUN --mount=type=secret,id=API_KEY,env=API_KEY \
some-command --token-from-env $API_KEY
```

Assuming that the `API_KEY` environment variable is set in the build environment, you can build this with the following command:

```
$ docker buildx build --secret id=API_KEY .
```

## RUN --mount=type=ssh

This mount type allows the build container to access SSH keys via SSH agents, with support for [passwordless logins](#).

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

Option	Description
<code>required</code>	If set to <code>true</code> , the instruction errors out when the key is unavailable. Defaults to <code>false</code> .
<code>mode</code>	File mode for socket in octal. Default <code>0600</code> .
<code>uid</code>	User ID for socket. Default <code>0</code> .
<code>gid</code>	Group ID for socket. Default <code>0</code> .

## Example: access to GitLab

```
# syntax=docker/dockerfile:1
FROM alpine
RUN apk add --no-cache openssh-client
RUN mkdir -p -m 0700 ~/.ssh && ssh-keyscan gitlab.com >> ~/.ssh/known_hosts
RUN --mount=type=ssh \
    ssh -q -T git@gitlab.com 2>&1 | tee /hello
# "Welcome to GitLab, @GITLAB_USERNAME_ASSOCIATED_WITH_SSHKEY" should be printed
# with the type of build progress is defined as `plain`.
```

```
$ eval $(ssh-agent)
$ ssh-add ~/.ssh/id_rsa
(Input your passphrase here)
$ docker buildx build --ssh default=$SSH_AUTH_SOCK .
```

You can also specify a path to `*.pem` file on the host directly instead of `$SSH_AUTH_SOCK`.

However, `pem` files with passphrases are not supported.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

The supported network types are:

Type	Description
default (default)	Run in the default network.
none	Run with no network access.
host	Run in the host's network environment.

## RUN --network=default

Equivalent to not supplying a flag at all, the command is run in the default network for the build.

## RUN --network=none

The command is run with no network access (`lo` is still available, but is isolated to this process)

### Example: isolating external effects

```
# syntax=docker/dockerfile:1
FROM python:3.6
ADD mypackage.tgz wheels/
RUN --network=none pip install --find-links wheels mypackage
```

`pip` will only be able to install the packages provided in the tarfile, which can be controlled by an earlier build stage.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

The use of `--network=host` is protected by the `network.host` entitlement, which needs to be enabled when starting the buildkitd daemon with `--allow-insecure-entitlement network.host` flag or in [buildkitd config](#), and for a build request with `--allow network.host` flag.

## RUN --security

### Note

Not yet available in stable syntax, use [docker/dockerfile:1-labs](#) version.

```
RUN --security=<sandbox|insecure>
```

The default security mode is `sandbox`. With `--security=insecure`, the builder runs the command without sandbox in insecure mode, which allows to run flows requiring elevated privileges (e.g. containerd). This is equivalent to running `docker run --privileged`.

### Warning

In order to access this feature, entitlement `security.insecure` should be enabled when starting the buildkitd daemon with `--allow-insecure-entitlement security.insecure` flag or in [buildkitd config](#), and for a build request with `--allow security.insecure` flag.

Default sandbox mode can be activated via `--security=sandbox`, but that is no-op.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

# CMD

The `CMD` instruction sets the command to be executed when running a container from an image.

You can specify `CMD` instructions using [shell](#) or [exec](#) forms:

- `CMD ["executable","param1","param2"]` (exec form)
- `CMD ["param1","param2"]` (exec form, 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`, only the last one takes effect.

The 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.

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](#). If the user specifies arguments to `docker run` then they will override the default specified in `CMD`, but still use the default `ENTRYPOINT`.

If `CMD` is used to provide default arguments for the `ENTRYPOINT` instruction, both the `CMD` and `ENTRYPOINT` instructions should be specified in the [exec](#) form.

## Note

Don't confuse `RUN` with `CMD`. `RUN` actually runs a command and commits the result; `CMD`

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

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"
```

### Note

Be sure to use double quotes and not single quotes. Particularly when you are using string interpolation (e.g. `LABEL example="foo-$ENV_VAR"`), single quotes will take the string as is without unpacking the variable's value.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
{  
  "com.example.vendor": "ACME Incorporated",  
  "com.example.label-with-value": "foo",  
  "version": "1.0",  
  "description": "This text illustrates that label-values can span multiple  
  "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 org.opencontainers.image.authors="SvenDowideit@home.org.au"
```

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

## EXPOSE

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

The `EXPOSE` instruction doesn't 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 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 TCP and UDP doesn't use the same port.

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](#). The `docker network` command supports creating networks for communication among containers without the need

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

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](#) in many as well. The value will be interpreted for other environment variables, so quote characters will be removed if they are not escaped. Like command line parsing, quotes and backslashes can be used to include spaces within values.

Example:

```
ENV MY_NAME="John Doe"
ENV MY_DOG=Rex\ The\ Dog
ENV MY_CAT=fluffy
```

The `ENV` instruction allows for multiple `<key>=<value> ...` variables to be set at one time, and the example below will yield the same net results in the final image:

```
ENV MY_NAME="John Doe" MY_DOG=Rex\ The\ Dog \
MY_CAT=fluffy
```

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 -e <key>=<value>`.

A stage inherits any environment variables that were set using `ENV` by its parent stage or any ancestor. Refer to the [multi-stage builds section](#) in the manual for more information.

Environment variable persistence can cause unexpected side effects. For example, setting `ENV DEBIAN_FRONTEND=noninteractive` changes the behavior of `apt-get`, and may confuse users.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
ARG DEBIAN_FRONTEND=noninteractive
-----
RUN apt-get update && apt-get install -y ...
```

## Note

### Alternative syntax

The ENV instruction also allows an alternative syntax ENV <key> <value>, omitting the =. For example:

```
ENV MY_VAR my-value
-----
```

This syntax does not allow for multiple environment-variables to be set in a single ENV instruction, and can be confusing. For example, the following sets a single environment variable (ONE) with value "TWO= THREE=world":

```
ENV ONE TWO= THREE=world
-----
```

The alternative syntax is supported for backward compatibility, but discouraged for the reasons outlined above, and may be removed in a future release.

## ADD

ADD has two forms. The latter form is required for paths containing whitespace.

```
ADD [OPTIONS] <src> ... <dest>
-----
ADD [OPTIONS] ["<src>", ... "<dest>"]
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

Option	Minimum Dockerfile version
--chown	
--chmod	1.2
--link	1.4
--exclude	1.7-labs

The `ADD` instruction copies new files or directories from `<src>` and adds them to the filesystem of the image at the path `<dest>`. Files and directories can be copied from the build context, a remote URL, or a Git repository.

The `ADD` and `COPY` instructions are functionally similar, but serve slightly different purposes. Learn more about the differences between `ADD` and `COPY`.

## Source

You can specify multiple source files or directories with `ADD`. The last argument must always be the destination. For example, to add two files, `file1.txt` and `file2.txt`, from the build context to `/usr/src/things/` in the build container:

```
ADD file1.txt file2.txt /usr/src/things/
```

If you specify multiple source files, either directly or using a wildcard, then the destination must be a directory (must end with a slash `/`).

To add files from a remote location, you can specify a URL or the address of a Git repository.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

- If `<src>` is a local file or directory, the contents of the directory are copied to the specified destination. See [Adding files from the build context](#).
- If `<src>` is a local tar archive, it is decompressed and extracted to the specified destination. See [Adding local tar archives](#).
- If `<src>` is a URL, the contents of the URL are downloaded and placed at the specified destination. See [Adding files from a URL](#).
- If `<src>` is a Git repository, the repository is cloned to the specified destination. See [Adding files from a Git repository](#).

### Adding files from the build context

Any relative or local path that doesn't begin with a `http://`, `https://`, or `git@` protocol prefix is considered a local file path. The local file path is relative to the build context. For example, if the build context is the current directory, `ADD file.txt /` adds the file at `./file.txt` to the root of the filesystem in the build container.

Specifying a source path with a leading slash or one that navigates outside the build context, such as `ADD ../something /something`, automatically removes any parent directory navigation (`..`). Trailing slashes in the source path are also disregarded, making `ADD something/ /something` equivalent to `ADD something /something`.

If the source is a directory, the contents of the directory are copied, including filesystem metadata. The directory itself isn't copied, only its contents. If it contains subdirectories, these are also copied, and merged with any existing directories at the destination. Any conflicts are resolved in favor of the content being added, on a file-by-file basis, except if you're trying to copy a directory onto an existing file, in which case an error is raised.

If the source is a file, the file and its metadata are copied to the destination. File permissions are

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

## Pattern matching

For local files, each `<src>` may contain wildcards and matching will be done using Go's [filepath.Match](#) rules.

For example, to add all files and directories in the root of the build context ending with `.png`:

```
ADD *.png /dest/
```

In the following example, `?` is a single-character wildcard, matching e.g. `index.js` and `index.ts`.

```
ADD index.?s /dest/
```

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 /dest/
```

## Adding local tar archives

When using a local tar archive as the source for `ADD`, and the archive is in a recognized compression format (`gzip`, `bzip2` or `xz`, or uncompressed), the archive is decompressed and extracted into the specified destination. Only local tar archives are extracted. If the tar archive is a remote URL, the archive is not extracted, but downloaded and placed at the destination.

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

## 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 isn't recognized as a compressed file and doesn't generate any kind of decompression error message, rather the file will simply be copied to the destination.

### Adding files from a URL

In the case where source is a remote file URL, the destination will have permissions of 600. If the HTTP response contains a `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` isn't included in the determination of whether or not the file has changed and the cache should be updated.

If the destination ends with a trailing slash, then the filename is inferred from the URL path. For example, `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 (`http://example.com` doesn't work).

If the destination doesn't end with a trailing slash, the destination path becomes the filename of the file downloaded from the URL. For example, `ADD http://example.com/foo /bar` creates the file `/bar`.

If your URL files are protected using authentication, you need to use `RUN wget`, `RUN curl` or use another tool from within the container as the `ADD` instruction doesn't support authentication.

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

You can use URL fragments to specify a specific branch, tag, commit, or subdirectory. For example, to add the `docs` directory of the `v0.14.1` tag of the `buildkit` repository:

```
ADD git@github.com:moby/buildkit.git#v0.14.1:docs /buildkit-docs
```

For more information about Git URL fragments, see [URL fragments](#).

When adding from a Git repository, the permissions bits for files are 644. If a file in the repository has the executable bit set, it will have permissions set to 755. Directories have permissions set to 755.

When using a Git repository as the source, the repository must be accessible from the build context. To add a repository via SSH, whether public or private, you must pass an SSH key for authentication. For example, given the following Dockerfile:

```
# syntax=docker/dockerfile:1
FROM alpine
-----
ADD git@git.example.com:foo/bar.git /bar
```

To build this Dockerfile, pass the `--ssh` flag to the `docker build` to mount the SSH agent socket to the build. For example:

```
$ docker build --ssh default .
```

For more information about building with secrets, see [Build secrets](#).

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
# create /abs/test.txt  
ADD test.txt /abs/
```

Trailing slashes are significant. For example, `ADD test.txt /abs` creates a file at `/abs`, whereas `ADD test.txt /abs/` creates `/abs/test.txt`.

If the destination path doesn't begin with a leading slash, it's interpreted as relative to the working directory of the build container.

```
WORKDIR /usr/src/app  
-----  
# create /usr/src/app/rel/test.txt  
ADD test.txt rel/
```

If destination doesn't exist, it's created, along with all missing directories in its path.

If the source is a file, and the destination doesn't end with a trailing slash, the source file will be written to the destination path as a file.

## ADD --keep-git-dir

```
ADD [--keep-git-dir=<boolean>] <src> ... <dir>
```

When `<src>` is the HTTP or SSH address of a remote Git repository, BuildKit adds the contents of the Git repository to the image excluding the `.git` directory by default.

The `--keep-git-dir=true` flag lets you preserve the `.git` directory.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
ADD [--checksum=<hash>] <src> ... <dir>
```

The `--checksum` flag lets you verify the checksum of a remote resource. The checksum is formatted as `sha256:<hash>`. SHA-256 is the only supported hash algorithm.

```
ADD --checksum=sha256:24454f830cdb571e2c4ad15481119c43b3caf48dd869a9b2945d:
```

The `--checksum` flag only supports HTTP(S) sources.

## ADD --chown --chmod

See [COPY --chown --chmod](#).

## ADD --link

See [COPY --link](#).

## ADD --exclude

See [COPY --exclude](#).

# COPY

COPY has two forms. The latter form is required for paths containing whitespace.

```
COPY [OPTIONS] <src> ... <dest>
COPY [OPTIONS] ["<src>", ... "<dest>"]
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

Option	Minimum Dockerfile version
--chown	
--chmod	1.2
--link	1.4
--parents	1.7-labs
--exclude	1.7-labs

The `COPY` instruction copies new files or directories from `<src>` and adds them to the filesystem of the image at the path `<dest>`. Files and directories can be copied from the build context, build stage, named context, or an image.

The `ADD` and `COPY` instructions are functionally similar, but serve slightly different purposes. Learn more about the differences between `ADD` and `COPY`.

## Source

You can specify multiple source files or directories with `COPY`. The last argument must always be the destination. For example, to copy two files, `file1.txt` and `file2.txt`, from the build context to `/usr/src/things/` in the build container:

```
COPY file1.txt file2.txt /usr/src/things/
```

If you specify multiple source files, either directly or using a wildcard, then the destination must be a directory (must end with a slash `/`).

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
COPY --from=build /myapp /usr/bin/
```

For more information about copying from named sources, see the `--from` flag.

## Copying from the build context

When copying source files from the build context, paths are interpreted as relative to the root of the context.

Specifying a source path with a leading slash or one that navigates outside the build context, such as `COPY ../something /something`, automatically removes any parent directory navigation (`..`). Trailing slashes in the source path are also disregarded, making `COPY something/ /something` equivalent to `COPY something /something`.

If the source is a directory, the contents of the directory are copied, including filesystem metadata. The directory itself isn't copied, only its contents. If it contains subdirectories, these are also copied, and merged with any existing directories at the destination. Any conflicts are resolved in favor of the content being added, on a file-by-file basis, except if you're trying to copy a directory onto an existing file, in which case an error is raised.

If the source is a file, the file and its metadata are copied to the destination. File permissions are preserved. If the source is a file and a directory with the same name exists at the destination, an error is raised.

If you pass a Dockerfile through stdin to the build (`docker build - < Dockerfile`), there is no build context. In this case, you can only use the `COPY` instruction to copy files from other stages, named contexts, or images, using the `--from` flag. You can also pass a tar archive through stdin: (`docker build - < archive.tar`), the Dockerfile at the root of the archive and

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

For local files, each `<src>` may contain wildcards and matching will be done using Go's [filepath.Match](#) rules.

For example, to add all files and directories in the root of the build context ending with `.png`:

```
COPY *.png /dest/
```

In the following example, `?` is a single-character wildcard, matching e.g. `index.js` and `index.ts`.

```
COPY index.?s /dest/
```

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;

```
COPY arr[[0]].txt /dest/
```

## Destination

If the destination path begins with a forward slash, it's interpreted as an absolute path, and the source files are copied into the specified destination relative to the root of the current build stage.

```
# create /abs/test.txt
COPY test.txt /abs/
```

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
WORKDIR /usr/src/app  
-----  
# create /usr/src/app/rel/test.txt  
COPY test.txt rel/  
-----
```

If destination doesn't exist, it's created, along with all missing directories in its path.

If the source is a file, and the destination doesn't end with a trailing slash, the source file will be written to the destination path as a file.

## COPY --from

By default, the `COPY` instruction copies files from the build context. The `COPY --from` flag lets you copy files from an image, a build stage, or a named context instead.

```
COPY [--from=<image|stage|context>] <src> ... <dest>  
-----
```

To copy from a build stage in a [multi-stage build](#), specify the name of the stage you want to copy from. You specify stage names using the `AS` keyword with the `FROM` instruction.

```
# syntax=docker/dockerfile:1  
FROM alpine AS build  
COPY . .  
RUN apk add clang  
RUN clang -o /hello hello.c  
  
FROM scratch  
COPY --from=build /hello /
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

The source path of `COPY --from` is always resolved from filesystem root of the image or stage that you specify.

## COPY --chown --chmod

### Note

Only octal notation is currently supported. Non-octal support is tracked in [moby/buildkit#1951](#).

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

The `--chown` and `--chmod` features are only supported on Dockerfiles used to build Linux containers, and doesn't 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.

All files and directories copied from the build context 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:

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

If the container root filesystem doesn't 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 does not depend on container root filesystem content.

With the Dockerfile syntax version 1.10.0 and later, the `--chmod` flag supports variable interpolation, which lets you define the permission bits using build arguments:

```
# syntax=docker/dockerfile:1.10
FROM alpine
WORKDIR /src
ARG MODE=440
COPY --chmod=$MODE . .
```

## COPY --link

```
-----  
COPY [--link[=<boolean>]] <src> ... <dest>
```

Enabling this flag in `COPY` or `ADD` commands allows you to copy files with enhanced semantics where your files remain independent on their own layer and don't get invalidated when commands on previous layers are changed.

When `--link` is used your source files are copied into an empty destination directory. That directory is turned into a layer that is linked on top of your previous state.

```
# syntax=docker/dockerfile:1
FROM alpine
```

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

and

```
FROM scratch  
-----  
COPY /foo /bar
```

and merging all the layers of both images together.

### Benefits of using `--link`

Use `--link` to reuse already built layers in subsequent builds with `--cache-from` even if the previous layers have changed. This is especially important for multi-stage builds where a `COPY --from` statement would previously get invalidated if any previous commands in the same stage changed, causing the need to rebuild the intermediate stages again. With `--link` the layer the previous build generated is reused and merged on top of the new layers. This also means you can easily rebase your images when the base images receive updates, without having to execute the whole build again. In backends that support it, BuildKit can do this rebase action without the need to push or pull any layers between the client and the registry. BuildKit will detect this case and only create new image manifest that contains the new layers and old layers in correct order.

The same behavior where BuildKit can avoid pulling down the base image can also happen when using `--link` and no other commands that would require access to the files in the base image. In that case BuildKit will only build the layers for the `COPY` commands and push them to the registry directly on top of the layers of the base image.

### Incompatibilities with `--link=false`

When using `--link` the `COPY/ADD` commands are not allowed to read any files from the previous state. This means that if in previous state the destination directory was a path that

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

## COPY --parents

### Note

Not yet available in stable syntax, use `docker/dockerfile:1.7-labs` version.

```
COPY [--parents[=<boolean>]] <src> ... <dest>
```

The `--parents` flag preserves parent directories for `src` entries. This flag defaults to `false`.

```
# syntax=docker/dockerfile:1-labs
FROM scratch

COPY ./x/a.txt ./y/a.txt /no_parents/
COPY --parents ./x/a.txt ./y/a.txt /parents/

# /no_parents/a.txt
# /parents/x/a.txt
# /parents/y/a.txt
```

This behavior is similar to the Linux `cp` utility's `--parents` or `rsync` `--relative` flag.

As with Rsync, it is possible to limit which parent directories are preserved by inserting a dot and a slash (`./`) into the source path. If such point exists, only parent directories after it will be preserved. This may be especially useful copies between stages with `--from` where the source paths need to be absolute.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
#  
# Output:  
# /parents/y/a.txt  
# /parents/y/b.txt
```

Note that, without the `--parents` flag specified, any filename collision will fail the Linux `cp` operation with an explicit error message (`cp: will not overwrite just-created './x/a.txt' with './y/a.txt'`), where the Buildkit will silently overwrite the target file at the destination.

While it is possible to preserve the directory structure for `COPY` instructions consisting of only one `src` entry, usually it is more beneficial to keep the layer count in the resulting image as low as possible. Therefore, with the `--parents` flag, the Buildkit is capable of packing multiple `COPY` instructions together, keeping the directory structure intact.

## COPY --exclude



### Note

Not yet available in stable syntax, use `docker/dockerfile:1.7-labs` version.

```
COPY [--exclude=<path> ...] <src> ... <dest>
```

The `--exclude` flag lets you specify a path expression for files to be excluded.

The path expression follows the same format as `<src>`, supporting wildcards and matching using Go's `filepath.Match` rules. For example, to add all files starting with "hom", excluding files

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

You can specify the `--exclude` option multiple times for a `COPY` instruction. Multiple `--excludes` are files matching its patterns not to be copied, even if the files paths match the pattern specified in `<src>`. To add all files starting with "hom", excluding files with either `.txt` or `.md` extensions:

```
# syntax=docker/dockerfile:1-labs
FROM scratch

COPY --exclude=*.txt --exclude=*.md hom* /mydir/
```

## ENTRYPOINT

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

`ENTRYPOINT` has two possible forms:

- The exec form, which is the preferred form:

```
ENTRYPOINT ["executable", "param1", "param2"]
```

- The shell form:

```
ENTRYPOINT command param1 param2
```

For more information about the different forms, see [Shell and exec form](#).

The following command starts a container from the `nginx` with its default content, listening on port 80:

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

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 of `ENTRYPOINT` prevents any `CMD` command line arguments from being used. It also starts your `ENTRYPOINT` 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. In this case, your executable doesn't 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 zombie
%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
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
$ docker exec -it test ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
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
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

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 container too

# USE the trap if you need to also do manual cleanup after the service 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
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

PID	USER	COMMAND
10035	root	{run.sh} /bin/sh /run.sh 123 cmd cmd
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).

## 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
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
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 exits 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

top - 13:58:24 up 17 min,  0 users,  load average: 0.00, 0.00, 0.00
Tasks:  2 total,   1 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s): 16.7 us, 33.3 sy,  0.0 ni, 50.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0
MiB Mem : 1990.8 total, 1354.6 free,   231.4 used,   404.7 buff/cache
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

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 waux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.4	0.0	2612	604	pts/0	Ss+	13:58	0:00	/bin/sh -c
root	6	0.0	0.1	5956	3188	pts/0	S+	13:58	0:00	top -b
root	7	0.0	0.1	5884	2816	pts/1	Rs+	13:58	0:00	ps waux

```
$ /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.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

		ENTRYPOINT	ENTRYPOINT ["exec_entry", "p1_entry"]
	No ENTRYPOINT	exec_entry p1_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 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](#) documentation.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

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 when using the legacy builder. When using Buildkit, the changes will instead be kept.
- **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

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

The `USER` instruction sets the user name (or UID) and optionally the user group (or GID) to use as the default user and group for the remainder of the current stage. The specified user is used for `RUN` instructions and at runtime, runs the relevant `ENTRYPOINT` and `CMD` commands.

Note that when specifying a group for the user, the user will have *only* the specified group membership. Any other configured group memberships will be ignored.

### 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.

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
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`

If not specified, the default working directory is `/`. In practice, if you aren't building a Dockerfile from scratch (`FROM scratch`), the `WORKDIR` may likely be set by the base image you're using.

Therefore, to avoid unintended operations in unknown directories, it's best practice to set your `WORKDIR` explicitly.

## ARG

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

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

Refer to the `RUN --mount=type=secret` section to learn about secure ways to use secrets when building images.

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

```
FROM busybox
-----
ARG user1
ARG buildno
-----
# ...
```

## 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 comes into effect from the line on which it is declared in the Dockerfile. For example, consider this Dockerfile:

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

A user builds this file by calling:

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

- The `USER` instruction on line 2 evaluates to the `some_user` fallback, because the `username` variable is not yet declared.
- The `username` variable is declared on line 3, and available for reference in Dockerfile instruction from that point onwards.
- The `USER` instruction on line 4 evaluates to `what_user`, since at that point the `username` argument has a value of `what_user` which 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` variable declared within a build stage is automatically inherited by other stages based on that stage. Unrelated build stages do not have access to the variable. To use an argument in multiple distinct stages, each stage must include the `ARG` instruction, or they must both be based on a shared base stage in the same Dockerfile where the variable is declared.

For more information, refer to [variable scoping](#).

## 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.

```
FROM ubuntu
-----
ARG CONT_IMG_VER
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

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:

```
FROM ubuntu
-----
ARG CONT_IMG_VER
-----
ENV CONT_IMG_VER=${CONT_IMG_VER:-v1.0.0}
-----
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](#).

## Predefined ARGs

Docker has a set of predefined `ARG` variables that you can use without a corresponding `ARG`

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

- `ftp_proxy`
- `NO_PROXY`
- `no_proxy`
- `ALL_PROXY`
- `all_proxy`

To use these, pass them on the command line using the `--build-arg` flag, for example:

```
$ docker build --build-arg HTTPS_PROXY=https://my-proxy.example.com .
```

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

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

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](#) backend.

BuildKit supports a predefined 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.
- `BUILDOOS` - OS component of `BUILDPLATFORM`
- `BUILDARCH` - architecture component of `BUILDPLATFORM`
- `BUILDVARIANT` - variant 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.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

## BuildKit built-in build args

Arg	Type	Description
BUILDKIT_CACHE_MOUNT_NS	String	Set optional cache ID namespace.
BUILDKIT_CONTEXT_KEEP_GIT_DIR	Bool	Trigger Git context to keep the <code>.git</code> directory.
BUILDKIT_HISTORY_PROVENANCE_V1	Bool	Enable <a href="#">SLSA Provenance v1</a> for build history record.
BUILDKIT_INLINE_CACHE <sup>2</sup>	Bool	Inline cache metadata to image config or not.
BUILDKIT_MULTI_PLATFORM	Bool	Opt into deterministic output regardless of multi-platform output or not.
BUILDKIT_SANDBOX_HOSTNAME	String	Set the hostname (default <code>buildkitsandbox</code> )
BUILDKIT_SYNTAX	String	Set frontend image
SOURCE_DATE_EPOCH	Int	Set the Unix timestamp for created image and layers. More info from <a href="#">reproducible builds</a> . Supported since Dockerfile 1.5, BuildKit 0.11

### Example: keep `.git` dir

When using a Git context, `.git` dir is not kept on checkouts. It can be useful to keep it around if you want to retrieve git information during your build:

```
# syntax=docker/dockerfile:1
FROM alpine
WORKDIR /src
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

`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:

```
FROM ubuntu
-----
ARG CONT_IMG_VER
-----
RUN echo $CONT_IMG_VER
```

```
FROM ubuntu
-----
ARG CONT_IMG_VER
-----
RUN echo hello
```

If you specify `--build-arg CONT_IMG_VER=<value>` on the command line, in both cases, the specification on line 2 doesn't 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, you get a cache miss.

Consider another example under the same command line:

```
FROM ubuntu
-----
ARG CONT_IMG_VER
```

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
FROM ubuntu
-----
ARG CONT_IMG_VER
-----
ENV CONT_IMG_VER=hello
-----
RUN echo $CONT_IMG_VER
```

Line 3 doesn't 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.

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,

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

- When it encounters an `ONBUILD` instruction, the builder adds a trigger to the metadata of the image being built. The instruction doesn't otherwise affect the current build.
- 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.
- 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.
- Triggers are cleared from the final image after being executed. In other words they aren't 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
```

## Copy or mount from stage, image, or context

As of Dockerfile syntax 1.11, you can use `ONBUILD` with instructions that copy or mount files from other stages, images, or build contexts. For example:

```
# syntax=docker/dockerfile:1.11
FROM alpine AS baseimage
ONBUILD COPY --from=build /usr/bin/app /app
ONBUILD RUN --mount=from=config,target=/opt/appconfig ...
```

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

- The `ONBUILD` instruction may not trigger `FROM` or `MAINTAINER` instructions.

## STOP SIGNAL

### `STOP SIGNAL` signal

The `STOP SIGNAL` instruction sets the system call signal that will be sent to the container to exit. This signal can be a signal name in the format `SIG<NAME>`, for instance `SIGKILL`, or an unsigned number that matches a position in the kernel's syscall table, for instance `9`. The default is `SIGTERM` if not defined.

The image's default stopsignal can be overridden per container, using the `--stop-signal` flag on `docker run` and `docker create`.

## 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's still working. This can detect cases such as a web server 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`.

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

- `--start-period=DURATION` (default: `0s`)
- `--start-interval=DURATION` (default: `5s`)
- `--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.

**start interval** is the time between health checks during the start period. This option requires Docker Engine version 25.0 or later.

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:

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
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.

## 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
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
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 the builder will be:

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

This is inefficient for two reasons. First, there is an unnecessary `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:\\\\foo.txt\""]
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
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:\myproject> 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

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

```
----> 8e559e9bf424
Removing intermediate container be6d8e63fe75
Successfully built 8e559e9bf424
PS E:\myproject>
```

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.

## Here-Documents

Here-documents allow redirection of subsequent Dockerfile lines to the input of `RUN` or `COPY` commands. If such command contains a `here-document` the Dockerfile considers the next lines until the line only containing a here-doc delimiter as part of the same command.

### Example: Running a multi-line script

```
# syntax=docker/dockerfile:1
FROM debian
RUN <<EOT bash
-----
  set -ex
  apt-get update
  apt-get install -y vim
EOT
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

Alternatively, shebang header can be used to define an interpreter.

```
# syntax=docker/dockerfile:1
FROM python:3.6
RUN <<EOT
#!/usr/bin/env python
print("hello world")
EOT
```

More complex examples may use multiple here-documents.

```
# syntax=docker/dockerfile:1
FROM alpine
RUN <<FILE1 cat > file1 && <<FILE2 cat > file2
I am
first
FILE1
I am
second
FILE2
```

## Example: Creating inline files

With `COPY` instructions, you can replace the source parameter with a here-doc indicator to write the contents of the here-document directly to a file. The following example creates a `greeting.txt` file containing `hello world` using a `COPY` instruction.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

here-document.

```
# syntax=docker/dockerfile:1
FROM alpine
ARG FOO=bar
COPY <<-EOT /script.sh
    echo "hello ${FOO}"
EOT
ENTRYPOINT ash /script.sh
```

In this case, file script prints "hello bar", because the variable is expanded when the `COPY` instruction gets executed.

```
$ docker build -t heredoc .
$ docker run heredoc
hello bar
```

If instead you were to quote any part of the here-document word `EOT`, the variable would not be expanded at build-time.

```
# syntax=docker/dockerfile:1
FROM alpine
ARG FOO=bar
COPY <<-"EOT" /script.sh
    echo "hello ${FOO}"
EOT
ENTRYPOINT ash /script.sh
```

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

# Dockerfile examples

For examples of Dockerfiles, refer to:

- The [building best practices page](#)
- The "get started" tutorials
- The [language-specific getting started guides](#)

- 
1. Value required ↵ ↵ ↵
  2. For Docker-integrated [BuildKit](#) and `docker buildx build` ↵

Copyright © 2013-2025 Docker Inc. All rights reserved.

By clicking “Accept All Cookies”, you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.