Dockerfile reference

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. This page describes the commands you can use in a Dockerfile.

[Overview](https://docs.docker.com/reference/dockerfile/#overview)

The Dockerfile supports the following instructions:

| **Instruction** | **Description** |
| --- | --- |
| [ADD](https://docs.docker.com/reference/dockerfile/#add) | Add local or remote files and directories. |
| [ARG](https://docs.docker.com/reference/dockerfile/#arg) | Use build-time variables. |
| [CMD](https://docs.docker.com/reference/dockerfile/#cmd) | Specify default commands. |
| [COPY](https://docs.docker.com/reference/dockerfile/#copy) | Copy files and directories. |
| [ENTRYPOINT](https://docs.docker.com/reference/dockerfile/#entrypoint) | Specify default executable. |
| [ENV](https://docs.docker.com/reference/dockerfile/#env) | Set environment variables. |
| [EXPOSE](https://docs.docker.com/reference/dockerfile/#expose) | Describe which ports your application is listening on. |
| [FROM](https://docs.docker.com/reference/dockerfile/#from) | Create a new build stage from a base image. |
| [HEALTHCHECK](https://docs.docker.com/reference/dockerfile/#healthcheck) | Check a container's health on startup. |
| [LABEL](https://docs.docker.com/reference/dockerfile/#label) | Add metadata to an image. |
| [MAINTAINER](https://docs.docker.com/reference/dockerfile/#maintainer-deprecated) | Specify the author of an image. |
| [ONBUILD](https://docs.docker.com/reference/dockerfile/#onbuild) | Specify instructions for when the image is used in a build. |
| [RUN](https://docs.docker.com/reference/dockerfile/#run) | Execute build commands. |
| [SHELL](https://docs.docker.com/reference/dockerfile/#shell) | Set the default shell of an image. |
| [STOPSIGNAL](https://docs.docker.com/reference/dockerfile/#stopsignal) | Specify the system call signal for exiting a container. |
| [USER](https://docs.docker.com/reference/dockerfile/#user) | Set user and group ID. |
| [VOLUME](https://docs.docker.com/reference/dockerfile/#volume) | Create volume mounts. |
| [WORKDIR](https://docs.docker.com/reference/dockerfile/#workdir) | Change working directory. |

[Format](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#parser-directives), [comments](https://docs.docker.com/reference/dockerfile/#format), and globally scoped [ARGs](https://docs.docker.com/reference/dockerfile/#arg). The FROM instruction specifies the [base image](https://docs.docker.com/glossary/#base-image) from which you are building. FROM may only be preceded by one or more ARG instructions, which declare arguments that are used in FROM lines in the Dockerfile.

BuildKit treats lines that begin with # as a comment, unless the line is a valid [parser directive](https://docs.docker.com/reference/dockerfile/#parser-directives). A # marker anywhere else in a line is treated as an argument. This allows statements like:

# Comment

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo 'we are running some # of cool things'

Comment lines are removed before the Dockerfile instructions are executed. The comment in the following example is removed before the shell executes the echo command.

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo hello \

# comment

world

The following examples is equivalent.

[RUN](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#run) echo world

# this is a comment-line

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo hello

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo world

Whitespace in instruction arguments, however, isn't ignored. The following example prints hello world with leading whitespace as specified:

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo "\

hello\

world"

[Parser directives](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#syntax)
* [escape](https://docs.docker.com/reference/dockerfile/#escape)
* [check](https://docs.docker.com/reference/dockerfile/#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=jsonargsrecommended 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:

# direc \

tive=value

Invalid due to appearing twice:

# directive=value1

# directive=value2

[FROM](https://docs.docker.com/reference/dockerfile/#from) ImageName

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

[FROM](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#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

# directive= value

# directive = value

# dIrEcTiVe=value

[syntax](https://docs.docker.com/reference/dockerfile/#syntax)

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](https://docs.docker.com/build/buildkit/dockerfile-frontend/).

[escape](https://docs.docker.com/reference/dockerfile/#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 command, except at the end of a line.

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

Consider the following example which would fail in a non-obvious way on Windows. The second \ at the end of the second line would be interpreted as an escape for the newline, instead of a target of the escape from the first \. Similarly, the \ at the end of the third line would, assuming it was actually handled as an instruction, cause it be treated as a line continuation. The result of this Dockerfile is that second and third lines are considered a single instruction:

[FROM](https://docs.docker.com/reference/dockerfile/#from) microsoft/nanoserver

[COPY](https://docs.docker.com/reference/dockerfile/#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=`

[FROM](https://docs.docker.com/reference/dockerfile/#from) microsoft/nanoserver

[COPY](https://docs.docker.com/reference/dockerfile/#copy) testfile.txt c:\

RUN dir c:\

Results in:

PS E:\myproject> docker build -t succeeds --no-cache=true .

Sending build context to Docker daemon 3.072 kB

Step 1/3 : FROM microsoft/nanoserver

---> 22738ff49c6d

Step 2/3 : COPY testfile.txt c:\

---> 96655de338de

Removing intermediate container 4db9acbb1682

Step 3/3 : RUN dir c:\

---> Running in a2c157f842f5

Volume in drive C has no label.

Volume Serial Number is 7E6D-E0F7

Directory of c:\

10/05/2016 05:04 PM 1,894 License.txt

10/05/2016 02:22 PM *DIR* Program Files

10/05/2016 02:14 PM *DIR* Program Files (x86)

10/28/2016 11:18 AM 62 testfile.txt

10/28/2016 11:20 AM *DIR* Users

10/28/2016 11:20 AM *DIR* Windows

2 File(s) 1,956 bytes

4 Dir(s) 21,259,096,064 bytes free

---> 01c7f3bef04f

Removing intermediate container a2c157f842f5

Successfully built 01c7f3bef04f

PS E:\myproject>

[check](https://docs.docker.com/reference/dockerfile/#check)

# check=skip=<checks|all>

# check=error=<boolean>

The check directive is used to configure how [build checks](https://docs.docker.com/build/checks/) are evaluated. By default, all checks are run, and failures are treated as warnings.

You can disable specific checks using #check=skip=<check-name>. To specify multiple checks to skip, separate them with a comma:

# 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](https://docs.docker.com/reference/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](https://docs.docker.com/reference/build-checks/). 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](https://docs.docker.com/reference/dockerfile/#syntax) directive to specify the Dockerfile syntax version to the latest stable version.

[Environment replacement](https://docs.docker.com/reference/dockerfile/#environment-replacement)

Environment variables (declared with [the ENV statement](https://docs.docker.com/reference/dockerfile/#env)) can also be used in certain instructions as variables to be interpreted by the Dockerfile. Escapes are also handled for including variable-like syntax into a statement literally.

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

The ${variable\_name} syntax also supports a few of the standard bash modifiers as specified below:

* ${variable:-word} indicates that if variable is set then the result will be that value. If variable is not set then word will be the result.
* ${variable:+word} indicates that if variable is set then word will be the result, otherwise the result is the empty string.

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

* ${variable%%pattern} removes the longest match of pattern from variable, seeking backwards from the end of the string.

string=foobarbaz echo ${string%%b\*} # foo

* ${variable/pattern/replacement} replace the first occurrence of pattern in variable with replacement

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](https://docs.docker.com/reference/dockerfile/#from) busybox

[ENV](https://docs.docker.com/reference/dockerfile/#env) FOO=/bar

[WORKDIR](https://docs.docker.com/reference/dockerfile/#workdir) ${FOO} # WORKDIR /bar

[ADD](https://docs.docker.com/reference/dockerfile/#add) . $FOO # ADD . /bar

[COPY](https://docs.docker.com/reference/dockerfile/#copy) \$FOO /quux # COPY $FOO /quux

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

* ADD
* COPY
* ENV
* EXPOSE
* FROM
* LABEL
* STOPSIGNAL
* USER
* VOLUME
* WORKDIR
* 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](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#env) abc=hello

[ENV](https://docs.docker.com/reference/dockerfile/#env) abc=bye def=$abc

[ENV](https://docs.docker.com/reference/dockerfile/#env) ghi=$abc

* The value of def becomes hello
* The value of ghi becomes bye

[.dockerignore file](https://docs.docker.com/reference/dockerfile/#dockerignore-file)

You can use .dockerignore file to exclude files and directories from the build context. For more information, see [.dockerignore file](https://docs.docker.com/build/building/context/#dockerignore-files).

[Shell and exec form](https://docs.docker.com/reference/dockerfile/#shell-and-exec-form)

The RUN, CMD, and ENTRYPOINT instructions all have two possible forms:

* INSTRUCTION ["executable","param1","param2"] (exec form)
* INSTRUCTION command param1 param2 (shell form)

The exec form makes it possible to avoid shell string munging, and to invoke commands using a specific command shell, or any other executable. It uses a JSON array syntax, where each element in the array is a command, flag, or argument.

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](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#entrypoint).

[Variable substitution](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#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 to not being valid JSON, and fail in an unexpected way:

[RUN](https://docs.docker.com/reference/dockerfile/#run) ["c:\windows\system32\tasklist.exe"]

The correct syntax for this example is:

[RUN](https://docs.docker.com/reference/dockerfile/#run) ["c:\\windows\\system32\\tasklist.exe"]

[Shell form](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#escape) (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](https://docs.docker.com/reference/dockerfile/#run) source $HOME/.bashrc && \

echo $HOME

They're equivalent to the following line:

[RUN](https://docs.docker.com/reference/dockerfile/#run) source $HOME/.bashrc && echo $HOME

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

[RUN](https://docs.docker.com/reference/dockerfile/#run) <<EOF

source $HOME/.bashrc && \

echo $HOME

EOF

For more information about heredocs, see [Here-documents](https://docs.docker.com/reference/dockerfile/#here-documents).

[Use a different shell](https://docs.docker.com/reference/dockerfile/#use-a-different-shell)

You can change the default shell using the SHELL command. For example:

[SHELL](https://docs.docker.com/reference/dockerfile/#shell) ["/bin/bash", "-c"]

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo hello

For more information, see [SHELL](https://docs.docker.com/reference/dockerfile/#shell).

[FROM](https://docs.docker.com/reference/dockerfile/#from)

[FROM](https://docs.docker.com/reference/dockerfile/#from) [--platform=<platform>] <image> [AS <name>]

Or

[FROM](https://docs.docker.com/reference/dockerfile/#from) [--platform=<platform>] <image>[:<tag>] [AS <name>]

Or

[FROM](https://docs.docker.com/reference/dockerfile/#from) [--platform=<platform>] <image>[@<digest>] [AS <name>]

The FROM instruction initializes a new build stage and sets the [base image](https://docs.docker.com/glossary/#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](https://docs.docker.com/reference/dockerfile/#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 <name>, [COPY --from=<name>](https://docs.docker.com/reference/dockerfile/#copy---from), and [RUN --mount=type=bind,from=<name>](https://docs.docker.com/reference/dockerfile/#run---mounttypebind) instructions to refer to the image built in this stage.
* The tag or digest values are optional. If you omit either of them, the builder assumes a latest tag by default. The builder returns an error if it can't find the tag value.

The optional --platform flag can be used to specify the platform of the image in case FROM references a multi-platform image. For example, linux/amd64, linux/arm64, or 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](https://docs.docker.com/reference/dockerfile/#automatic-platform-args-in-the-global-scope) 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](https://docs.docker.com/reference/dockerfile/#understand-how-arg-and-from-interact)

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

[ARG](https://docs.docker.com/reference/dockerfile/#arg) CODE\_VERSION=latest

[FROM](https://docs.docker.com/reference/dockerfile/#from) base:${CODE\_VERSION}

[CMD](https://docs.docker.com/reference/dockerfile/#cmd) /code/run-app

[FROM](https://docs.docker.com/reference/dockerfile/#from) extras:${CODE\_VERSION}

[CMD](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#arg) VERSION=latest

[FROM](https://docs.docker.com/reference/dockerfile/#from) busybox:$VERSION

[ARG](https://docs.docker.com/reference/dockerfile/#arg) VERSION

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo $VERSION > image\_version

[RUN](https://docs.docker.com/reference/dockerfile/#run)

The RUN instruction will execute any commands to create a new layer on top of the current image. The added layer is used in the next step in the Dockerfile. RUN has two forms:

# Shell form:

[RUN](https://docs.docker.com/reference/dockerfile/#run) [OPTIONS] <command> ...

# Exec form:

[RUN](https://docs.docker.com/reference/dockerfile/#run) [OPTIONS] [ "<command>", ... ]

For more information about the differences between these two forms, see [shell or exec forms](https://docs.docker.com/reference/dockerfile/#shell-and-exec-form).

The shell form is most commonly used, and lets you break up longer instructions into multiple lines, either using newline [escapes](https://docs.docker.com/reference/dockerfile/#escape), or with [heredocs](https://docs.docker.com/reference/dockerfile/#here-documents):

[RUN](https://docs.docker.com/reference/dockerfile/#run) <<EOF

apt-get update

apt-get install -y curl

EOF

The available [OPTIONS] for the RUN instruction are:

| **Option** | **Minimum Dockerfile version** |
| --- | --- |
| [--mount](https://docs.docker.com/reference/dockerfile/#run---mount) | 1.2 |
| [--network](https://docs.docker.com/reference/dockerfile/#run---network) | 1.3 |
| [--security](https://docs.docker.com/reference/dockerfile/#run---security) | 1.1.2-labs |

[Cache invalidation for RUN instructions](https://docs.docker.com/reference/dockerfile/#cache-invalidation-for-run-instructions)

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

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

The cache for RUN instructions can be invalidated by [ADD](https://docs.docker.com/reference/dockerfile/#add) and [COPY](https://docs.docker.com/reference/dockerfile/#copy) instructions.

[RUN --mount](https://docs.docker.com/reference/dockerfile/#run---mount)

[RUN](https://docs.docker.com/reference/dockerfile/#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** |
| --- | --- |
| [bind](https://docs.docker.com/reference/dockerfile/#run---mounttypebind) (default) | Bind-mount context directories (read-only). |
| [cache](https://docs.docker.com/reference/dockerfile/#run---mounttypecache) | Mount a temporary directory to cache directories for compilers and package managers. |
| [tmpfs](https://docs.docker.com/reference/dockerfile/#run---mounttypetmpfs) | Mount a tmpfs in the build container. |
| [secret](https://docs.docker.com/reference/dockerfile/#run---mounttypesecret) | Allow the build container to access secure files such as private keys without baking them into the image or build cache. |
| [ssh](https://docs.docker.com/reference/dockerfile/#run---mounttypessh) | Allow the build container to access SSH keys via SSH agents, with support for passphrases. |

[RUN --mount=type=bind](https://docs.docker.com/reference/dockerfile/#run---mounttypebind)

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

| **Option** | **Description** |
| --- | --- |
| target, dst, destination[1](https://docs.docker.com/reference/dockerfile/#fn:1) | Mount path. |
| source | Source path in the from. Defaults to the root of the from. |
| from | Build stage, context, or image name for the root of the source. Defaults to the build context. |
| rw,readwrite | Allow writes on the mount. Written data will be discarded. |

[RUN --mount=type=cache](https://docs.docker.com/reference/dockerfile/#run---mounttypecache)

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

| **Option** | **Description** |
| --- | --- |
| id | Optional ID to identify separate/different caches. Defaults to value of target. |
| target, dst, destination[1](https://docs.docker.com/reference/dockerfile/#fn:1) | Mount path. |
| ro,readonly | Read-only if set. |
| sharing | One of shared, private, or locked. Defaults to shared. A shared cache mount can be used concurrently by multiple writers. private creates a new mount if there are multiple writers. locked pauses the second writer until the first one releases the mount. |
| from | Build stage, context, or image name to use as a base of the cache mount. Defaults to empty directory. |
| source | Subpath in the from to mount. Defaults to the root of the from. |
| mode | File mode for new cache directory in octal. Default 0755. |
| uid | User ID for new cache directory. Default 0. |
| gid | Group ID for new cache directory. Default 0. |

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](https://docs.docker.com/reference/dockerfile/#example-cache-go-packages)

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/#from) golang

[RUN](https://docs.docker.com/reference/dockerfile/#run) --mount=type=cache,target=/root/.cache/go-build \

go build ...

[Example: cache apt packages](https://docs.docker.com/reference/dockerfile/#example-cache-apt-packages)

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[RUN](https://docs.docker.com/reference/dockerfile/#run) rm -f /etc/apt/apt.conf.d/docker-clean; echo 'Binary::apt::APT::Keep-Downloaded-Packages "true";' > /etc/apt/apt.conf.d/keep-cache

[RUN](https://docs.docker.com/reference/dockerfile/#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

Apt needs exclusive access to its data, so the caches use the option sharing=locked, which will make sure multiple parallel builds using the same cache mount will wait for each other and not access the same cache files at the same time. You could also use sharing=private if you prefer to have each build create another cache directory in this case.

[RUN --mount=type=tmpfs](https://docs.docker.com/reference/dockerfile/#run---mounttypetmpfs)

This mount type allows mounting tmpfs in the build container.

| **Option** | **Description** |
| --- | --- |
| target, dst, destination[1](https://docs.docker.com/reference/dockerfile/#fn:1) | Mount path. |
| size | Specify an upper limit on the size of the filesystem. |

[RUN --mount=type=secret](https://docs.docker.com/reference/dockerfile/#run---mounttypesecret)

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** |
| --- | --- |
| id | ID of the secret. Defaults to basename of the target path. |
| target, dst, destination | Mount the secret to the specified path. Defaults to /run/secrets/ + id if unset and if env is also unset. |
| env | Mount the secret to an environment variable instead of a file, or both. (since Dockerfile v1.10.0) |
| required | If set to true, the instruction errors out when the secret is unavailable. Defaults to false. |
| mode | File mode for secret file in octal. Default 0400. |
| uid | User ID for secret file. Default 0. |
| gid | Group ID for secret file. Default 0. |

[Example: access to S3](https://docs.docker.com/reference/dockerfile/#example-access-to-s3)

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/#from) python:3

[RUN](https://docs.docker.com/reference/dockerfile/#run) pip install awscli

[RUN](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#from) alpine

[RUN](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#run---mounttypessh)

This mount type allows the build container to access SSH keys via SSH agents, with support for passphrases.

| **Option** | **Description** |
| --- | --- |
| id | ID of SSH agent socket or key. Defaults to "default". |
| target, dst, destination | SSH agent socket path. Defaults to /run/buildkit/ssh\_agent.${N}. |
| required | If set to true, the instruction errors out when the key is unavailable. Defaults to false. |
| mode | File mode for socket in octal. Default 0600. |
| uid | User ID for socket. Default 0. |
| gid | Group ID for socket. Default 0. |

[Example: access to GitLab](https://docs.docker.com/reference/dockerfile/#example-access-to-gitlab)

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/#from) alpine

[RUN](https://docs.docker.com/reference/dockerfile/#run) apk add --no-cache openssh-client

[RUN](https://docs.docker.com/reference/dockerfile/#run) mkdir -p -m 0700 ~/.ssh && ssh-keyscan gitlab.com >> ~/.ssh/known\_hosts

[RUN](https://docs.docker.com/reference/dockerfile/#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 here

# 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.

[RUN --network](https://docs.docker.com/reference/dockerfile/#run---network)

[RUN](https://docs.docker.com/reference/dockerfile/#run) --network=*TYPE*

RUN --network allows control over which networking environment the command is run in.

The supported network types are:

| **Type** | **Description** |
| --- | --- |
| [default](https://docs.docker.com/reference/dockerfile/#run---networkdefault) (default) | Run in the default network. |
| [none](https://docs.docker.com/reference/dockerfile/#run---networknone) | Run with no network access. |
| [host](https://docs.docker.com/reference/dockerfile/#run---networkhost) | Run in the host's network environment. |

[RUN --network=default](https://docs.docker.com/reference/dockerfile/#run---networkdefault)

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

[RUN --network=none](https://docs.docker.com/reference/dockerfile/#run---networknone)

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

[Example: isolating external effects](https://docs.docker.com/reference/dockerfile/#example-isolating-external-effects)

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/#from) python:3.6

[ADD](https://docs.docker.com/reference/dockerfile/#add) mypackage.tgz wheels/

[RUN](https://docs.docker.com/reference/dockerfile/#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.

[RUN --network=host](https://docs.docker.com/reference/dockerfile/#run---networkhost)

The command is run in the host's network environment (similar to docker build --network=host, but on a per-instruction basis)

**Warning**

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](https://github.com/moby/buildkit/blob/master/docs/buildkitd.toml.md), and for a build request with [--allow network.host flag](https://docs.docker.com/engine/reference/commandline/buildx_build/#allow).

[RUN --security](https://docs.docker.com/reference/dockerfile/#run---security)

**Note**

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

[RUN](https://docs.docker.com/reference/dockerfile/#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](https://github.com/moby/buildkit/blob/master/docs/buildkitd.toml.md), and for a build request with [--allow security.insecure flag](https://docs.docker.com/engine/reference/commandline/buildx_build/#allow).

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

[Example: check entitlements](https://docs.docker.com/reference/dockerfile/#example-check-entitlements)

# syntax=docker/dockerfile:1-labs

[FROM](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[RUN](https://docs.docker.com/reference/dockerfile/#run) --security=insecure cat /proc/self/status | grep CapEff

#84 0.093 CapEff: 0000003fffffffff

[CMD](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#shell-and-exec-form):

* 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](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#exec-form).

**Note**

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

[LABEL](https://docs.docker.com/reference/dockerfile/#label)

[LABEL](https://docs.docker.com/reference/dockerfile/#label) <key>=<value> [<key>=<value>...]

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

[LABEL](https://docs.docker.com/reference/dockerfile/#label) "com.example.vendor"="ACME Incorporated"

[LABEL](https://docs.docker.com/reference/dockerfile/#label) com.example.label-with-value="foo"

[LABEL](https://docs.docker.com/reference/dockerfile/#label) version="1.0"

[LABEL](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#label) multi.label1="value1" multi.label2="value2" other="value3"

[LABEL](https://docs.docker.com/reference/dockerfile/#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.

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

To view an image's labels, use the docker image inspect command. You can use the --format option to show just the labels;

$ docker image inspect --format='{{json .Config.Labels}}' myimage

{

"com.example.vendor": "ACME Incorporated",

"com.example.label-with-value": "foo",

"version": "1.0",

"description": "This text illustrates that label-values can span multiple lines.",

"multi.label1": "value1",

"multi.label2": "value2",

"other": "value3"

}

[MAINTAINER (deprecated)](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#label) org.opencontainers.image.authors="SvenDowideit@home.org.au"

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

[EXPOSE](https://docs.docker.com/reference/dockerfile/#expose)

[EXPOSE](https://docs.docker.com/reference/dockerfile/#expose) <port> [<port>/<protocol>...]

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

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](https://docs.docker.com/reference/dockerfile/#expose) 80/udp

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

[EXPOSE](https://docs.docker.com/reference/dockerfile/#expose) 80/tcp

[EXPOSE](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/cli/docker/container/run/#publish). The docker network command supports creating networks for communication among containers without the need to expose or publish specific ports, because the containers connected to the network can communicate with each other over any port. For detailed information, see the [overview of this feature](https://docs.docker.com/engine/userguide/networking/).

[ENV](https://docs.docker.com/reference/dockerfile/#env)

[ENV](https://docs.docker.com/reference/dockerfile/#env) <key>=<value> [<key>=<value>...]

The ENV instruction sets the environment variable <key> to the value <value>. This value will be in the environment for all subsequent instructions in the build stage and can be [replaced inline](https://docs.docker.com/reference/dockerfile/#environment-replacement) 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](https://docs.docker.com/reference/dockerfile/#env) MY\_NAME="John Doe"

[ENV](https://docs.docker.com/reference/dockerfile/#env) MY\_DOG=Rex\ The\ Dog

[ENV](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#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 --env <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](https://docs.docker.com/build/building/multi-stage/) 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 of your image.

If an environment variable is only needed during build, and not in the final image, consider setting a value for a single command instead:

[RUN](https://docs.docker.com/reference/dockerfile/#run) DEBIAN\_FRONTEND=noninteractive apt-get update && apt-get install -y ...

Or using [ARG](https://docs.docker.com/reference/dockerfile/#arg), which is not persisted in the final image:

[ARG](https://docs.docker.com/reference/dockerfile/#arg) DEBIAN\_FRONTEND=noninteractive

[RUN](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#add)

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

[ADD](https://docs.docker.com/reference/dockerfile/#add) [OPTIONS] <src> ... <dest>

[ADD](https://docs.docker.com/reference/dockerfile/#add) [OPTIONS] ["<src>", ... "<dest>"]

The available [OPTIONS] are:

| **Option** | **Minimum Dockerfile version** |
| --- | --- |
| [--keep-git-dir](https://docs.docker.com/reference/dockerfile/#add---keep-git-dir) | 1.1 |
| [--checksum](https://docs.docker.com/reference/dockerfile/#add---checksum) | 1.6 |
| [--chown](https://docs.docker.com/reference/dockerfile/#add---chown---chmod) |  |
| [--chmod](https://docs.docker.com/reference/dockerfile/#add---chown---chmod) | 1.2 |
| [--link](https://docs.docker.com/reference/dockerfile/#add---link) | 1.4 |
| [--exclude](https://docs.docker.com/reference/dockerfile/#add---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](https://docs.docker.com/build/building/best-practices/#add-or-copy).

[Source](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#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 as the source. For example:

[ADD](https://docs.docker.com/reference/dockerfile/#add) https://example.com/archive.zip /usr/src/things/

[ADD](https://docs.docker.com/reference/dockerfile/#add) git@github.com:user/repo.git /usr/src/things/

BuildKit detects the type of <src> and processes it accordingly.

* 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](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#adding-files-from-a-git-repository).

[Adding files from the build context](https://docs.docker.com/reference/dockerfile/#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 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 ADD instruction to copy remote files. You can also pass a tar archive through stdin: (docker build - < archive.tar), the Dockerfile at the root of the archive and the rest of the archive will be used as the context of the build.

[Pattern matching](https://docs.docker.com/reference/dockerfile/#pattern-matching)

For local files, each <src> may contain wildcards and matching will be done using Go's [filepath.Match](https://golang.org/pkg/path/filepath#Match) rules.

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

[ADD](https://docs.docker.com/reference/dockerfile/#add) \*.png /dest/

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

[ADD](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#add) arr[[]0].txt /dest/

[Adding local tar archives](https://docs.docker.com/reference/dockerfile/#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.

When a directory is extracted, it has the same behavior as tar -x. The result is the union of:

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

**Note**

Whether a file is identified as a recognized compression format or not is done solely based on the contents of the file, not the name of the file. For example, if an empty file happens to end with .tar.gz this 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](https://docs.docker.com/reference/dockerfile/#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.

[Adding files from a Git repository](https://docs.docker.com/reference/dockerfile/#adding-files-from-a-git-repository)

To use a Git repository as the source for ADD, you can reference the repository's HTTP or SSH address as the source. The repository is cloned to the specified destination in the image.

[ADD](https://docs.docker.com/reference/dockerfile/#add) https://github.com/user/repo.git /mydir/

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](https://docs.docker.com/reference/dockerfile/#add) git@github.com:moby/buildkit.git#v0.14.1:docs /buildkit-docs

For more information about Git URL fragments, see [URL fragments](https://docs.docker.com/build/building/context/#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](https://docs.docker.com/reference/dockerfile/#from) alpine

[ADD](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/build/building/secrets/).

[Destination](https://docs.docker.com/reference/dockerfile/#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

[ADD](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#workdir) /usr/src/app

# create /usr/src/app/rel/test.txt

[ADD](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#add---keep-git-dir)

[ADD](https://docs.docker.com/reference/dockerfile/#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.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/#from) alpine

[ADD](https://docs.docker.com/reference/dockerfile/#add) --keep-git-dir=true https://github.com/moby/buildkit.git#v0.10.1 /buildkit

[ADD --checksum](https://docs.docker.com/reference/dockerfile/#add---checksum)

[ADD](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#add) --checksum=sha256:24454f830cdb571e2c4ad15481119c43b3cafd48dd869a9b2945d1036d1dc68d https://mirrors.edge.kernel.org/pub/linux/kernel/Historic/linux-0.01.tar.gz /

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

[ADD --chown --chmod](https://docs.docker.com/reference/dockerfile/#add---chown---chmod)

See [COPY --chown --chmod](https://docs.docker.com/reference/dockerfile/#copy---chown---chmod).

[ADD --link](https://docs.docker.com/reference/dockerfile/#add---link)

See [COPY --link](https://docs.docker.com/reference/dockerfile/#copy---link).

[ADD --exclude](https://docs.docker.com/reference/dockerfile/#add---exclude)

See [COPY --exclude](https://docs.docker.com/reference/dockerfile/#copy---exclude).

[COPY](https://docs.docker.com/reference/dockerfile/#copy)

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

[COPY](https://docs.docker.com/reference/dockerfile/#copy) [OPTIONS] <src> ... <dest>

[COPY](https://docs.docker.com/reference/dockerfile/#copy) [OPTIONS] ["<src>", ... "<dest>"]

The available [OPTIONS] are:

| **Option** | **Minimum Dockerfile version** |
| --- | --- |
| [--from](https://docs.docker.com/reference/dockerfile/#copy---from) |  |
| [--chown](https://docs.docker.com/reference/dockerfile/#copy---chown---chmod) |  |
| [--chmod](https://docs.docker.com/reference/dockerfile/#copy---chown---chmod) | 1.2 |
| [--link](https://docs.docker.com/reference/dockerfile/#copy---link) | 1.4 |
| [--parents](https://docs.docker.com/reference/dockerfile/#copy---parents) | 1.7-labs |
| [--exclude](https://docs.docker.com/reference/dockerfile/#copy---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](https://docs.docker.com/build/building/best-practices/#add-or-copy).

[Source](https://docs.docker.com/reference/dockerfile/#source-1)

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](https://docs.docker.com/reference/dockerfile/#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 /).

COPY accepts a flag --from=<name> that lets you specify the source location to be a build stage, context, or image. The following example copies files from a stage named build:

[FROM](https://docs.docker.com/reference/dockerfile/#from) golang AS build

[WORKDIR](https://docs.docker.com/reference/dockerfile/#workdir) /app

[RUN](https://docs.docker.com/reference/dockerfile/#run) --mount=type=bind,target=. go build -o /myapp ./cmd

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --from=build /myapp /usr/bin/

For more information about copying from named sources, see the [--from flag](https://docs.docker.com/reference/dockerfile/#copy---from).

[Copying from the build context](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#copy---from). You can also pass a tar archive through stdin: (docker build - < archive.tar), the Dockerfile at the root of the archive and the rest of the archive will be used as the context of the build.

When using a Git repository as the build context, the permissions bits for copied 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.

[Pattern matching](https://docs.docker.com/reference/dockerfile/#pattern-matching-1)

For local files, each <src> may contain wildcards and matching will be done using Go's [filepath.Match](https://golang.org/pkg/path/filepath#Match) rules.

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

[COPY](https://docs.docker.com/reference/dockerfile/#copy) \*.png /dest/

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

[COPY](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#copy) arr[[]0].txt /dest/

[Destination](https://docs.docker.com/reference/dockerfile/#destination-1)

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](https://docs.docker.com/reference/dockerfile/#copy) test.txt /abs/

Trailing slashes are significant. For example, COPY test.txt /abs creates a file at /abs, whereas COPY 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](https://docs.docker.com/reference/dockerfile/#workdir) /usr/src/app

# create /usr/src/app/rel/test.txt

[COPY](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#copy) [--from=<image|stage|context>] <src> ... <dest>

To copy from a build stage in a [multi-stage build](https://docs.docker.com/build/building/multi-stage/), 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](https://docs.docker.com/reference/dockerfile/#from) alpine AS build

[COPY](https://docs.docker.com/reference/dockerfile/#copy) . .

[RUN](https://docs.docker.com/reference/dockerfile/#run) apk add clang

[RUN](https://docs.docker.com/reference/dockerfile/#run) clang -o /hello hello.c

[FROM](https://docs.docker.com/reference/dockerfile/#from) scratch

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --from=build /hello /

You can also copy files directly from named contexts (specified with --build-context <name>=<source>) or images. The following example copies an nginx.conf file from the official Nginx image.

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --from=nginx:latest /etc/nginx/nginx.conf /nginx.conf

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

[COPY --chown --chmod](https://docs.docker.com/reference/dockerfile/#copy---chown---chmod)

**Note**

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

[COPY](https://docs.docker.com/reference/dockerfile/#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:

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --chown=55:mygroup files\* /somedir/

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --chown=bin files\* /somedir/

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --chown=1 files\* /somedir/

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --chown=10:11 files\* /somedir/

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --chown=myuser:mygroup --chmod=644 files\* /somedir/

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](https://docs.docker.com/reference/dockerfile/#from) alpine

[WORKDIR](https://docs.docker.com/reference/dockerfile/#workdir) /src

[ARG](https://docs.docker.com/reference/dockerfile/#arg) MODE=440

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --chmod=$MODE . .

[COPY --link](https://docs.docker.com/reference/dockerfile/#copy---link)

[COPY](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#from) alpine

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --link /foo /bar

Is equivalent of doing two builds:

[FROM](https://docs.docker.com/reference/dockerfile/#from) alpine

and

[FROM](https://docs.docker.com/reference/dockerfile/#from) scratch

[COPY](https://docs.docker.com/reference/dockerfile/#copy) /foo /bar

and merging all the layers of both images together.

[Benefits of using --link](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#incompatibilities-with---linkfalse)

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 contained a symlink, COPY/ADD can not follow it. In the final image the destination path created with --link will always be a path containing only directories.

If you don't rely on the behavior of following symlinks in the destination path, using --link is always recommended. The performance of --link is equivalent or better than the default behavior and, it creates much better conditions for cache reuse.

[COPY --parents](https://docs.docker.com/reference/dockerfile/#copy---parents)

**Note**

Not yet available in stable syntax, use [docker/dockerfile:1.7-labs](https://docs.docker.com/reference/dockerfile/#syntax) version.

[COPY](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#from) scratch

[COPY](https://docs.docker.com/reference/dockerfile/#copy) ./x/a.txt ./y/a.txt /no\_parents/

[COPY](https://docs.docker.com/reference/dockerfile/#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](https://www.man7.org/linux/man-pages/man1/cp.1.html) --parents or [rsync](https://man7.org/linux/man-pages/man1/rsync.1.html) --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.

# syntax=docker/dockerfile:1-labs

[FROM](https://docs.docker.com/reference/dockerfile/#from) scratch

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --parents ./x/./y/\*.txt /parents/

# Build context:

# ./x/y/a.txt

# ./x/y/b.txt

#

# 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](https://docs.docker.com/reference/dockerfile/#copy---exclude)

**Note**

Not yet available in stable syntax, use [docker/dockerfile:1.7-labs](https://docs.docker.com/reference/dockerfile/#syntax) version.

[COPY](https://docs.docker.com/reference/dockerfile/#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](https://golang.org/pkg/path/filepath#Match) rules. For example, to add all files starting with "hom", excluding files with a .txt extension:

# syntax=docker/dockerfile:1-labs

[FROM](https://docs.docker.com/reference/dockerfile/#from) scratch

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --exclude=\*.txt hom\* /mydir/

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](https://docs.docker.com/reference/dockerfile/#from) scratch

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --exclude=\*.txt --exclude=\*.md hom\* /mydir/

[ENTRYPOINT](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#entrypoint) ["executable", "param1", "param2"]

* The shell form:

[ENTRYPOINT](https://docs.docker.com/reference/dockerfile/#entrypoint) command param1 param2

For more information about the different forms, see [Shell and exec form](https://docs.docker.com/reference/dockerfile/#shell-and-exec-form).

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

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

Command line arguments to docker run <image> will be appended after all elements in an exec form ENTRYPOINT, and will override all elements specified using CMD.

This allows arguments to be passed to the entry point, i.e., docker run <image> -d will pass the -d argument to the entry point. You can override the ENTRYPOINT instruction using the docker run --entrypoint flag.

The shell form 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](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[ENTRYPOINT](https://docs.docker.com/reference/dockerfile/#entrypoint) ["top", "-b"]

[CMD](https://docs.docker.com/reference/dockerfile/#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 st

KiB Mem: 2056668 total, 1616832 used, 439836 free, 99352 buffers

KiB Swap: 1441840 total, 0 used, 1441840 free. 1324440 cached Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

1 root 20 0 19744 2336 2080 R 0.0 0.1 0:00.04 top

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

$ docker exec -it test ps aux

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME 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](https://docs.docker.com/reference/dockerfile/#from) debian:stable

[RUN](https://docs.docker.com/reference/dockerfile/#run) apt-get update && apt-get install -y --force-yes apache2

[EXPOSE](https://docs.docker.com/reference/dockerfile/#expose) 80 443

[VOLUME](https://docs.docker.com/reference/dockerfile/#volume) ["/var/www", "/var/log/apache2", "/etc/apache2"]

[ENTRYPOINT](https://docs.docker.com/reference/dockerfile/#entrypoint) ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]

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

#!/usr/bin/env bash

set -e

if [ "$1" = 'postgres' ]; then

chown -R postgres "$PGDATA"

if [ -z "$(ls -A "$PGDATA")" ]; then

gosu postgres initdb

fi

exec gosu postgres "$@"

fi

exec "$@"

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

#!/bin/sh

# Note: I've written this using sh so it works in the busybox 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

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

root 1 0.1 0.0 4448 692 ? Ss+ 00:42 0:00 /bin/sh /run.sh 123 cmd cmd2

root 19 0.0 0.2 71304 4440 ? Ss 00:42 0:00 /usr/sbin/apache2 -k start

www-data 20 0.2 0.2 360468 6004 ? Sl 00:42 0:00 /usr/sbin/apache2 -k start

www-data 21 0.2 0.2 360468 6000 ? Sl 00:42 0:00 /usr/sbin/apache2 -k start

root 81 0.0 0.1 15572 2140 ? R+ 00:44 0:00 ps aux

$ docker top test

PID USER COMMAND

10035 root {run.sh} /bin/sh /run.sh 123 cmd cmd2

10054 root /usr/sbin/apache2 -k start

10055 33 /usr/sbin/apache2 -k start

10056 33 /usr/sbin/apache2 -k start

$ /usr/bin/time docker stop test

test

real 0m 0.27s

user 0m 0.03s

sys 0m 0.03s

**Note**

You can override the ENTRYPOINT setting using --entrypoint, but this can only set the binary to exec (no sh -c will be used).

[Shell form ENTRYPOINT example](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[ENTRYPOINT](https://docs.docker.com/reference/dockerfile/#entrypoint) exec top -b

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

$ docker run -it --rm --name test top

Mem: 1704520K used, 352148K free, 0K shrd, 0K buff, 140368121167873K cached

CPU: 5% usr 0% sys 0% nic 94% idle 0% io 0% irq 0% sirq

Load average: 0.08 0.03 0.05 2/98 6

PID PPID USER STAT VSZ %VSZ %CPU COMMAND

1 0 root R 3164 0% 0% top -b

Which 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](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[ENTRYPOINT](https://docs.docker.com/reference/dockerfile/#entrypoint) top -b

[CMD](https://docs.docker.com/reference/dockerfile/#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 st

MiB Mem : 1990.8 total, 1354.6 free, 231.4 used, 404.7 buff/cache

MiB Swap: 1024.0 total, 1024.0 free, 0.0 used. 1639.8 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

1 root 20 0 2612 604 536 S 0.0 0.0 0:00.02 sh

6 root 20 0 5956 3188 2768 R 0.0 0.2 0:00.00 top

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

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

$ docker exec -it test ps 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 top -b --ignored-param2

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](https://docs.docker.com/reference/dockerfile/#understand-how-cmd-and-entrypoint-interact)

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

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

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

|  | **No ENTRYPOINT** | **ENTRYPOINT exec\_entry p1\_entry** | **ENTRYPOINT ["exec\_entry", "p1\_entry"]** |
| --- | --- | --- | --- |
| **No CMD** | error, not allowed | /bin/sh -c exec\_entry p1\_entry | exec\_entry p1\_entry |
| **CMD ["exec\_cmd", "p1\_cmd"]** | exec\_cmd p1\_cmd | /bin/sh -c exec\_entry p1\_entry | exec\_entry p1\_entry exec\_cmd p1\_cmd |
| **CMD 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](https://docs.docker.com/reference/dockerfile/#volume)

[VOLUME](https://docs.docker.com/reference/dockerfile/#volume) ["/data"]

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

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

[FROM](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[RUN](https://docs.docker.com/reference/dockerfile/#run) mkdir /myvol

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo "hello world" > /myvol/greeting

[VOLUME](https://docs.docker.com/reference/dockerfile/#volume) /myvol

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

[Notes about specifying volumes](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#user)

[USER](https://docs.docker.com/reference/dockerfile/#user) <user>[:<group>]

or

[USER](https://docs.docker.com/reference/dockerfile/#user) *UID*[:*GID*]

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](https://docs.docker.com/reference/dockerfile/#from) microsoft/windowsservercore

# Create Windows user in the container

[RUN](https://docs.docker.com/reference/dockerfile/#run) net user /add patrick

# Set it for subsequent commands

[USER](https://docs.docker.com/reference/dockerfile/#user) patrick

[WORKDIR](https://docs.docker.com/reference/dockerfile/#workdir)

[WORKDIR](https://docs.docker.com/reference/dockerfile/#workdir) /path/to/workdir

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

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

[WORKDIR](https://docs.docker.com/reference/dockerfile/#workdir) /a

[WORKDIR](https://docs.docker.com/reference/dockerfile/#workdir) b

[WORKDIR](https://docs.docker.com/reference/dockerfile/#workdir) c

[RUN](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#env) DIRPATH=/path

[WORKDIR](https://docs.docker.com/reference/dockerfile/#workdir) $DIRPATH/$DIRNAME

[RUN](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#arg)

[ARG](https://docs.docker.com/reference/dockerfile/#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.

**Warning**

It isn't recommended to use build arguments for passing secrets such as user credentials, API tokens, etc. Build arguments are visible in the docker history command and in max mode provenance attestations, which are attached to the image by default if you use the Buildx GitHub Actions and your GitHub repository is public.

Refer to the [RUN --mount=type=secret](https://docs.docker.com/reference/dockerfile/#run---mounttypesecret) 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](https://docs.docker.com/reference/dockerfile/#from) busybox

[ARG](https://docs.docker.com/reference/dockerfile/#arg) user1

[ARG](https://docs.docker.com/reference/dockerfile/#arg) buildno

# ...

[Default values](https://docs.docker.com/reference/dockerfile/#default-values)

An ARG instruction can optionally include a default value:

[FROM](https://docs.docker.com/reference/dockerfile/#from) busybox

[ARG](https://docs.docker.com/reference/dockerfile/#arg) user1=someuser

[ARG](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#scope)

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

[FROM](https://docs.docker.com/reference/dockerfile/#from) busybox

[USER](https://docs.docker.com/reference/dockerfile/#user) ${username:-some\_user}

[ARG](https://docs.docker.com/reference/dockerfile/#arg) username

[USER](https://docs.docker.com/reference/dockerfile/#user) $username

# ...

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](https://docs.docker.com/build/building/variables/#scoping).

[Using ARG variables](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[ARG](https://docs.docker.com/reference/dockerfile/#arg) CONT\_IMG\_VER

[ENV](https://docs.docker.com/reference/dockerfile/#env) CONT\_IMG\_VER=v1.0.0

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo $CONT\_IMG\_VER

Then, assume this image is built with this command:

$ docker build --build-arg CONT\_IMG\_VER=v2.0.1 .

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

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

[FROM](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[ARG](https://docs.docker.com/reference/dockerfile/#arg) CONT\_IMG\_VER

[ENV](https://docs.docker.com/reference/dockerfile/#env) CONT\_IMG\_VER=${CONT\_IMG\_VER:-v1.0.0}

[RUN](https://docs.docker.com/reference/dockerfile/#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.](https://docs.docker.com/reference/dockerfile/#environment-replacement)

[Predefined ARGs](https://docs.docker.com/reference/dockerfile/#predefined-args)

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

* HTTP\_PROXY
* http\_proxy
* HTTPS\_PROXY
* https\_proxy
* FTP\_PROXY
* ftp\_proxy
* NO\_PROXY
* no\_proxy
* 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](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo "Hello World"

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

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

[FROM](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[ARG](https://docs.docker.com/reference/dockerfile/#arg) HTTP\_PROXY

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo "Hello World"

When building this Dockerfile, the HTTP\_PROXY is preserved in the docker history, and changing its value invalidates the build cache.

[Automatic platform ARGs in the global scope](https://docs.docker.com/reference/dockerfile/#automatic-platform-args-in-the-global-scope)

This feature is only available when using the [BuildKit](https://docs.docker.com/build/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.
* BUILDOS - 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.

For example:

[FROM](https://docs.docker.com/reference/dockerfile/#from) alpine

[ARG](https://docs.docker.com/reference/dockerfile/#arg) TARGETPLATFORM

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo "I'm building for $TARGETPLATFORM"

[BuildKit built-in build args](https://docs.docker.com/reference/dockerfile/#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 .git directory. |
| BUILDKIT\_INLINE\_CACHE[2](https://docs.docker.com/reference/dockerfile/#fn:2) | 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 buildkitsandbox) |
| BUILDKIT\_SYNTAX | String | Set frontend image |
| SOURCE\_DATE\_EPOCH | Int | Set the Unix timestamp for created image and layers. More info from [reproducible builds](https://reproducible-builds.org/docs/source-date-epoch/). Supported since Dockerfile 1.5, BuildKit 0.11 |

[Example: keep .git dir](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#from) alpine

[WORKDIR](https://docs.docker.com/reference/dockerfile/#workdir) /src

[RUN](https://docs.docker.com/reference/dockerfile/#run) --mount=target=. \

make REVISION=$(git rev-parse HEAD) build

$ docker build --build-arg BUILDKIT\_CONTEXT\_KEEP\_GIT\_DIR=1 https://github.com/user/repo.git#main

[Impact on build caching](https://docs.docker.com/reference/dockerfile/#impact-on-build-caching)

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

For example, consider these two Dockerfile:

[FROM](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[ARG](https://docs.docker.com/reference/dockerfile/#arg) CONT\_IMG\_VER

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo $CONT\_IMG\_VER

[FROM](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[ARG](https://docs.docker.com/reference/dockerfile/#arg) CONT\_IMG\_VER

[RUN](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[ARG](https://docs.docker.com/reference/dockerfile/#arg) CONT\_IMG\_VER

[ENV](https://docs.docker.com/reference/dockerfile/#env) CONT\_IMG\_VER=$CONT\_IMG\_VER

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo $CONT\_IMG\_VER

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

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

[FROM](https://docs.docker.com/reference/dockerfile/#from) ubuntu

[ARG](https://docs.docker.com/reference/dockerfile/#arg) CONT\_IMG\_VER

[ENV](https://docs.docker.com/reference/dockerfile/#env) CONT\_IMG\_VER=hello

[RUN](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#onbuild)

[ONBUILD](https://docs.docker.com/reference/dockerfile/#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, but that's inefficient, error-prone and difficult to update because it mixes with application-specific code.

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

Here's how it works:

1. When it encounters an ONBUILD instruction, the builder adds a trigger to the metadata of the image being built. The instruction doesn't otherwise affect the current build.
2. At the end of the build, a list of all triggers is stored in the image manifest, under the key OnBuild. They can be inspected with the docker inspect command.
3. Later the image may be used as a base for a new build, using the FROM instruction. As part of processing the FROM instruction, the downstream builder looks for ONBUILD triggers, and executes them in the same order they were registered. If any of the triggers fail, the FROM instruction is aborted which in turn causes the build to fail. If all triggers succeed, the FROM instruction completes and the build continues as usual.
4. Triggers are cleared from the final image after being executed. In other words they aren't inherited by "grand-children" builds.

For example you might add something like this:

[ONBUILD](https://docs.docker.com/reference/dockerfile/#onbuild) [ADD](https://docs.docker.com/reference/dockerfile/#add) . /app/src

[ONBUILD](https://docs.docker.com/reference/dockerfile/#onbuild) [RUN](https://docs.docker.com/reference/dockerfile/#run) /usr/local/bin/python-build --dir /app/src

[Copy or mount from stage, image, or context](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#from) alpine AS baseimage

[ONBUILD](https://docs.docker.com/reference/dockerfile/#onbuild) [COPY](https://docs.docker.com/reference/dockerfile/#copy) --from=build /usr/bin/app /app

[ONBUILD](https://docs.docker.com/reference/dockerfile/#onbuild) [RUN](https://docs.docker.com/reference/dockerfile/#run) --mount=from=config,target=/opt/appconfig ...

If the source of from is a build stage, the stage must be defined in the Dockerfile where ONBUILD gets triggered. If it's a named context, that context must be passed to the downstream build.

[ONBUILD limitations](https://docs.docker.com/reference/dockerfile/#onbuild-limitations)

* Chaining ONBUILD instructions using ONBUILD ONBUILD isn't allowed.
* The ONBUILD instruction may not trigger FROM or MAINTAINER instructions.

[STOPSIGNAL](https://docs.docker.com/reference/dockerfile/#stopsignal)

[STOPSIGNAL](https://docs.docker.com/reference/dockerfile/#stopsignal) signal

The STOPSIGNAL 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](https://docs.docker.com/reference/dockerfile/#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 (whatever state it was previously in). After a certain number of consecutive failures, it becomes unhealthy.

The options that can appear before CMD are:

* --interval=DURATION (default: 30s)
* --timeout=DURATION (default: 30s)
* --start-period=DURATION (default: 0s)
* --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:

* 0: success - the container is healthy and ready for use
* 1: unhealthy - the container isn't working correctly
* 2: reserved - don't use this exit code

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

HEALTHCHECK --interval=5m --timeout=3s \

[CMD](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#shell)

[SHELL](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#from) microsoft/windowsservercore

# Executed as cmd /S /C echo default

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo default

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

[RUN](https://docs.docker.com/reference/dockerfile/#run) powershell -command Write-Host default

# Executed as powershell -command Write-Host hello

[SHELL](https://docs.docker.com/reference/dockerfile/#shell) ["powershell", "-command"]

[RUN](https://docs.docker.com/reference/dockerfile/#run) Write-Host hello

# Executed as cmd /S /C echo hello

[SHELL](https://docs.docker.com/reference/dockerfile/#shell) ["cmd", "/S", "/C"]

[RUN](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#run) ["powershell", "-command", "Execute-MyCmdlet", "-param1 \"c:\\foo.txt\""]

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

# escape=`

[FROM](https://docs.docker.com/reference/dockerfile/#from) microsoft/nanoserver

[SHELL](https://docs.docker.com/reference/dockerfile/#shell) ["powershell","-command"]

[RUN](https://docs.docker.com/reference/dockerfile/#run) New-Item -ItemType Directory C:\Example

[ADD](https://docs.docker.com/reference/dockerfile/#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

---> 3f2fbf1395d9

Removing intermediate container d0eef8386e97

Step 4/5 : ADD Execute-MyCmdlet.ps1 c:\example\

---> a955b2621c31

Removing intermediate container b825593d39fc

Step 5/5 : RUN c:\example\Execute-MyCmdlet 'hello world'

---> Running in be6d8e63fe75

hello world

---> 8e559e9bf424

Removing intermediate container be6d8e63fe75

Successfully built 8e559e9bf424

PS E:\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](https://docs.docker.com/reference/dockerfile/#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](https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#tag_18_07_04) 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](https://docs.docker.com/reference/dockerfile/#example-running-a-multi-line-script)

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/#from) debian

[RUN](https://docs.docker.com/reference/dockerfile/#run) <<EOT bash

set -ex

apt-get update

apt-get install -y vim

EOT

If the command only contains a here-document, its contents is evaluated with the default shell.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/#from) debian

[RUN](https://docs.docker.com/reference/dockerfile/#run) <<EOT

mkdir -p foo/bar

EOT

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

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/#from) python:3.6

[RUN](https://docs.docker.com/reference/dockerfile/#run) <<EOT

#!/usr/bin/env python

print("hello world")

EOT

More complex examples may use multiple here-documents.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/#from) alpine

[RUN](https://docs.docker.com/reference/dockerfile/#run) <<FILE1 cat > file1 && <<FILE2 cat > file2

I am

first

FILE1

I am

second

FILE2

[Example: Creating inline files](https://docs.docker.com/reference/dockerfile/#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.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/#from) alpine

[COPY](https://docs.docker.com/reference/dockerfile/#copy) <<EOF greeting.txt

hello world

EOF

Regular here-doc [variable expansion and tab stripping rules](https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#tag_18_07_04) apply. The following example shows a small Dockerfile that creates a hello.sh script file using a COPY instruction with a here-document.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/#from) alpine

[ARG](https://docs.docker.com/reference/dockerfile/#arg) FOO=bar

[COPY](https://docs.docker.com/reference/dockerfile/#copy) <<-EOT /script.sh

echo "hello ${FOO}"

EOT

[ENTRYPOINT](https://docs.docker.com/reference/dockerfile/#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](https://docs.docker.com/reference/dockerfile/#from) alpine

[ARG](https://docs.docker.com/reference/dockerfile/#arg) FOO=bar

[COPY](https://docs.docker.com/reference/dockerfile/#copy) <<-"EOT" /script.sh

echo "hello ${FOO}"

EOT

[ENTRYPOINT](https://docs.docker.com/reference/dockerfile/#entrypoint) ash /script.sh

Note that ARG FOO=bar is excessive here, and can be removed. The variable gets interpreted at runtime, when the script is invoked:

$ docker build -t heredoc .

$ docker run -e FOO=world heredoc

hello world

[Dockerfile examples](https://docs.docker.com/reference/dockerfile/#dockerfile-examples)

For examples of Dockerfiles, refer to:

* The [building best practices page](https://docs.docker.com/build/building/best-practices/)
* The ["get started" tutorials](https://docs.docker.com/get-started/)
* The [language-specific getting started guides](https://docs.docker.com/guides/language/)

1. Value required [↩︎](https://docs.docker.com/reference/dockerfile/#fnref:1) [↩︎](https://docs.docker.com/reference/dockerfile/#fnref1:1) [↩︎](https://docs.docker.com/reference/dockerfile/#fnref2:1)
2. For Docker-integrated [BuildKit](https://docs.docker.com/build/buildkit/#getting-started) and docker buildx build [↩︎](https://docs.docker.com/reference/dockerfile/#fnref:2)