Dockerfile reference

Table of contents

- Overview
- Format
- Parser directives
  - syntax
  - escape
- Environment replacement
- .dockerignore file
- Shell and exec form
  - Exec form
  - Shell form
  - Use a different shell
- FROM
  - Understand how ARG and FROM interact
- RUN
  - Cache invalidation for RUN instructions
  - RUN --mount
  - RUN --mount=type=bind
  - RUN --mount=type=cache
  - RUN --mount=type=tmpfs
  - RUN --mount=type=secret
  - RUN --mount=type=ssh
  - RUN --network
  - RUN --network=default
  - RUN --network=none
  - RUN --network=host
  - RUN --security
- CMD
- LABEL
- MAINTAINER (deprecated)
- EXPOSE
- ENV
- ADD
  - Adding private Git repositories
  - ADD --keep-git-dir
  - ADD --checksum
  - ADD --chown --chmod
  - ADD --link
  - ADD --exclude
- COPY
  - COPY --from
  - COPY --chown --chmod
  - COPY --link
  - COPY --parents
  - COPY --exclude
- ENTRYPOINT
  - Exec form ENTRYPOINT example
  - Shell form ENTRYPOINT example
  - Understand how CMD and ENTRYPOINT interact
- VOLUME
  - Notes about specifying volumes
- USER
- WORKDIR
- ARG
  - Default values
  - Scope
  - Using ARG variables

- Predefined ARGs
- Automatic platform ARGs in the global scope
- BuildKit built-in build args
- Impact on build caching
- ONBUILD
  - ONBUILD limitations
- STOPSIGNAL
- HEALTHCHECK
- SHELL
- Here-Documents
  - Example: Running a multi-line script
  - Example: Creating inline files
- Dockerfile examples

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

Overview

The Dockerfile supports the following instructions:

Instruction	Description
`ADD`	Add local or remote files and directories.
`ARG`	Use build-time variables.
`CMD`	Specify default commands.
`COPY`	Copy files and directories.
`ENTRYPOINT`	Specify default executable.
`ENV`	Set environment variables.
`EXPOSE`	Describe which ports your application is listening on.
`FROM`	Create a new build stage from a base image.
`HEALTHCHECK`	Check a container's health on startup.
`LABEL`	Add metadata to an image.
`MAINTAINER`	Specify the author of an image.
`ONBUILD`	Specify instructions for when the image is used in a build.
`RUN`	Execute build commands.
`SHELL`	Set the default shell of an image.
`STOPSIGNAL`	Specify the system call signal for exiting a container.
`USER`	Set user and group ID.
`VOLUME`	Create volume mounts.
`WORKDIR`	Change working directory.

Format

Here is the format of the Dockerfile:

```
# Comment
INSTRUCTION arguments
```

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

Docker runs instructions in a Dockerfile in order. A Dockerfile **must begin with a `FROM` instruction**. This may be after parser directives, comments, and globally scoped ARGs. The `FROM` instruction specifies the parent image from which you are building. `FROM` may only be preceded by one or more `ARG` instructions, which declare arguments that are used in `FROM` lines in the Dockerfile.

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

```
# Comment
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 echo hello \
# comment
world
```

The following examples is equivalent.

```
RUN echo hello \
world
```

Comments don't support line continuation characters.

> **Note on whitespace**
>
> For backward compatibility, leading whitespace before comments (#) and instructions (such as `RUN`) are ignored, but discouraged. Leading whitespace is not preserved in these cases, and the following examples are therefore equivalent:
>
> ```
>         # this is a comment-line
>    RUN echo hello
> RUN echo world
> ```
>
> ```
> # this is a comment-line
> RUN echo hello
> RUN echo world
> ```
>
> Whitespace in instruction arguments, however, isn't ignored. The following example prints `hello world` with leading whitespace as specified:
>
> ```
> RUN echo "\
>     hello\
>     world"
> ```

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.

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 directives aren't case-sensitive, but they're lowercase by convention. 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 ImageName
```

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

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

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

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

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

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

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

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

```
#          dIrEcTiVe=value
```

The following parser directives are supported:

- `syntax`
- `escape`

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.

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.

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

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

Results in:

```
PS 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 microsoft/nanoserver
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>
```

Environment replacement

Environment variables (declared with the `ENV` statement) 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 busybox
ENV FOO=/bar
WORKDIR ${FOO}   # WORKDIR /bar
ADD . $FOO       # ADD . /bar
COPY \$FOO /quux # COPY $FOO /quux
```

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

- `ADD`
- `COPY`
- `ENV`
- `EXPOSE`
- `FROM`
- `LABEL`
- `STOPSIGNAL`
- `USER`
- `VOLUME`
- `WORKDIR`
- `ONBUILD` (when combined with one of the supported instructions above)

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

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

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

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

.dockerignore file

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

Shell and exec form

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

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

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

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

Variable substitution

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

If you want shell processing then either use the shell form or execute a shell directly with the exec form, for example: `RUN [ "sh", "-c", "echo $HOME" ]`. When using the exec form and executing a shell directly, as in the case for the shell form, it's the shell that's doing the environment variable substitution, not the builder.

Backslashes

In exec form, you must escape backslashes. This is particularly relevant on Windows where the backslash is the path separator. The following line would otherwise be treated as shell form due to not being valid JSON, and fail in an unexpected way:

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

The correct syntax for this example is:

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

Shell form

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

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

They're equivalent to the following line:

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

You can also use heredocs with the shell form to break up a command:

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

For more information about heredocs, see Here-documents.

Use a different shell

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

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

For more information, see SHELL.

FROM

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

Or

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

Or

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

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

- `ARG` is the only instruction that may precede `FROM` in the Dockerfile. See Understand how ARG and FROM interact.
- `FROM` can appear multiple times within a single Dockerfile to create multiple images or use one build stage as a dependency for another. Simply make a note of the last image ID output by the commit before each new `FROM` instruction. Each `FROM` instruction clears any state created by previous instructions.
- Optionally a name can be given to a new build stage by adding `AS name` to the `FROM` instruction. The name can be used in subsequent `FROM <name>`, `COPY --from=<name>`, and `RUN --mount=type=bind,from=<name>` 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 allow you to force a stage to native build platform (`--platform=$BUILDPLATFORM`), and use it to cross-compile to the target platform inside the stage.

Understand how ARG and FROM interact

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

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

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

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

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

RUN

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

For more information about the differences between these two forms, see shell or exec forms.

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

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

The available [OPTIONS] for the RUN instruction are:

- --mount
- --network
- --security

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 for more information.

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

RUN --mount

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

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

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

The supported mount types are:

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

`ssh`	Allow the build container to access SSH keys via SSH agents, with support for passphrases.

RUN --mount=type=bind

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

Option	Description
`target`[1]	Mount path.
`source`	Source path in the `from`. Defaults to the root of the `from`.
`from`	Build stage 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

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`[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 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

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

Example: cache apt packages

```
# syntax=docker/dockerfile:1
FROM ubuntu
RUN rm -f /etc/apt/apt.conf.d/docker-clean; echo 'Binary::apt::APT::Keep-Downloaded-Packages "true";' > /etc/apt/apt.conf.d/ke
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

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

Option	Description
`target`[1]	Mount path.
`size`	Specify an upper limit on the size of the filesystem.

RUN --mount=type=secret

This mount type allows the build container to access secure files such as private keys without baking them into the image.

Option	Description
`id`	ID of the secret. Defaults to basename of the target path.
`target`	Mount path. Defaults to `/run/secrets/` + `id`.

`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

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

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

RUN --mount=type=ssh

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`	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

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

```
RUN --network=<TYPE>
```

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

The supported network types are:

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

RUN --network=default

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

RUN --network=none

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

Example: isolating external effects

```
# syntax=docker/dockerfile:1
FROM python:3.6
```

```
ADD mypackage.tgz wheels/
RUN --network=none pip install --find-links wheels mypackage
```

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

RUN --network=host

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, and for a build request with `--allow network.host` flag.

RUN --security

> **Note**
>
> Not yet available in stable syntax, use `docker/dockerfile:1-labs` version.

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

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

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

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

Example: check entitlements

```
# syntax=docker/dockerfile:1-labs
FROM ubuntu
RUN --security=insecure cat /proc/self/status | grep CapEff

#84 0.093 CapEff:        0000003fffffffff
```

CMD

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

You can specify `CMD` instructions using shell or exec forms:

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

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

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

If you would like your container to run the same executable every time, then you should consider using `ENTRYPOINT` in combination with `CMD`. See `ENTRYPOINT`. If the user specifies arguments to `docker run` then they will override the default specified in `CMD`, but still use the default `ENTRYPOINT`.

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

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

LABEL

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

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

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

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

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

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

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

To view an image's labels, use the `docker 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)

```
MAINTAINER <name>
```

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

```
LABEL org.opencontainers.image.authors="SvenDowideit@home.org.au"
```

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

EXPOSE

```
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 80/udp
```

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

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

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

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

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

To set up port redirection on the host system, see using the -P flag. The `docker network` command supports creating networks for communication among containers without the need 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.

ENV

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

The ENV instruction sets the environment variable `<key>` to the value `<value>`. This value will be in the environment for all subsequent instructions in the build stage and can be replaced inline in many as well. The value will be interpreted for other environment variables, so quote characters will be removed if they are not escaped. Like command line parsing, quotes and backslashes can be used to include spaces within values.

Example:

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

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

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

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

A stage inherits any environment variables that were set using ENV by its parent stage or any ancestor. Refer here for more on multi-staged builds.

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

Or using ARG, which is not persisted in the final image:

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

> **Alternative syntax**
>
> The ENV instruction also allows an alternative syntax `ENV <key> <value>`, omitting the =. For example:
>
> ```
> ENV MY_VAR my-value
> ```
>
> This syntax does not allow for multiple environment-variables to be set in a single ENV instruction, and can be confusing. For example, the following sets a single environment variable (`ONE`) with value `"TWO= THREE=world"`:
>
> ```
> ENV ONE TWO= THREE=world
> ```
>
> The alternative syntax is supported for backward compatibility, but discouraged for the reasons outlined above, and may be removed in a future release.

ADD

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

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

The available `[OPTIONS]` are:

- `--keep-git-dir`
- `--checksum`
- `--chown`
- `--chmod`
- `--link`

- `--exclude`

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

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

Each `<src>` may contain wildcards and matching will be done using Go's filepath.Match rules. For example:

To add all files in the root of the build context starting with "hom":

```
ADD hom* /mydir/
```

In the following example, ? is a single-character wildcard, matching e.g. "home.txt".

```
ADD hom?.txt /mydir/
```

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

The example below uses a relative path, and adds "test.txt" to `<WORKDIR>/relativeDir/`:

```
ADD test.txt relativeDir/
```

Whereas this example uses an absolute path, and adds "test.txt" to `/absoluteDir/`

```
ADD test.txt /absoluteDir/
```

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

```
ADD arr[[]0].txt /mydir/
```

In the case where `<src>` is a remote file URL, the destination will have permissions of 600. If the remote file being retrieved has an HTTP `Last-Modified` header, the timestamp from that header will be used to set the `mtime` on the destination file. However, like any other file processed during an ADD, `mtime` isn't included in the determination of whether or not the file has changed and the cache should be updated.

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

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.

> **Note**
>
> The first encountered ADD instruction will invalidate the cache for all following instructions from the Dockerfile if the contents of `<src>` have changed. This includes invalidating the cache for RUN instructions. See the Dockerfile Best Practices guide â■■ Leverage build cache for more information.

ADD obeys the following rules:

> The `<src>` path must be inside the build context; you can't use `ADD ../something /something`, because the builder can only access files from the context, and `../something` specifies a parent file or directory of the build context root.
>
> If `<src>` is a URL and `<dest>` does end with a trailing slash, then the filename is inferred from the URL and the file is downloaded to `<dest>/<filename>`. For instance, `ADD http://example.com/foobar /` would create the file `/foobar`. The URL must have a nontrivial path so that an appropriate filename can be discovered in this case (`http://example.com` doesn't work).
>
> If `<src>` is a directory, the entire contents of the directory are copied, including filesystem metadata.
>
> > **Note**
> >
> > The directory itself isn't copied, only its contents.
>
> If `<src>` is a local `tar` archive in a recognized compression format (`identity`, `gzip`, `bzip2` or `xz`) then it's unpacked as a directory. Resources from remote URLs aren't decompressed. When a directory is copied or unpacked, 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 "2." on a file-by-file basis.
>
> > **Note**
> >
> > Whether a file is identified as a recognized compression format or not is done solely based on the contents of the file, not the name of the file. For example, if an empty file happens to end with `.tar.gz` this 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.

If `<src>` is any other kind of file, it's copied individually along with its metadata. In this case, if `<dest>` ends with a trailing slash `/`, it will be considered a directory and the contents of `<src>` will be written at `<dest>/base(<src>)`.

If multiple `<src>` resources are specified, either directly or due to the use of a wildcard, then `<dest>` must be a directory, and it must end with a slash `/`.

If `<src>` is a file, and `<dest>` doesn't end with a trailing slash, the contents of `<src>` will be written as filename `<dest>`.

If `<dest>` doesn't exist, it's created, along with all missing directories in its path.

Adding private Git repositories

To add a private repository via SSH, create a Dockerfile with the following form:

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

This Dockerfile can be built with `docker build --ssh` or `buildctl build --ssh`, e.g.,

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

```
$ buildctl build --frontend=dockerfile.v0 --local context=. --local dockerfile=. --ssh default
```

ADD --keep-git-dir

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

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

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

```
# syntax=docker/dockerfile:1
FROM alpine
ADD --keep-git-dir=true https://github.com/moby/buildkit.git#v0.10.1 /buildkit
```

ADD --checksum

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

The `--checksum` flag lets you verify the checksum of a remote resource:

```
ADD --checksum=sha256:24454f830cdb571e2c4ad15481119c43b3cafd48dd869a9b2945d1036d1dc68d https://mirrors.edge.kernel.org/pub/lin
```

The `--checksum` flag only supports HTTP sources currently.

ADD --chown --chmod

See `COPY --chown --chmod`.

ADD --link

See `COPY --link`.

ADD --exclude

See `COPY --exclude`.

COPY

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

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

The available `[OPTIONS]` are:

- `--from`
- `--chown`

- `--chmod`
- `--link`
- `--parents`
- `--exclude`

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

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

Each `<src>` may contain wildcards and matching will be done using Go's filepath.Match rules. For example:

To add all files in the root of the build context starting with "hom":

```
COPY hom* /mydir/
```

In the following example, `?` is a single-character wildcard, matching e.g. "home.txt".

```
COPY hom?.txt /mydir/
```

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

The example below uses a relative path, and adds "test.txt" to `<WORKDIR>/relativeDir/`:

```
COPY test.txt relativeDir/
```

Whereas this example uses an absolute path, and adds "test.txt" to `/absoluteDir/`

```
COPY test.txt /absoluteDir/
```

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

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

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

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

`COPY` obeys the following rules:

> The `<src>` path is resolved relative to the build context. If you specify a relative path leading outside of the build context, such as `COPY ../something /something`, parent directory paths are stripped out automatically. The effective source path in this example becomes `COPY something /something`
>
> If `<src>` is a directory, the entire contents of the directory are copied, including filesystem metadata.
>
> > **Note**
> >
> > The directory itself isn't copied, only its contents.
>
> If `<src>` is any other kind of file, it's copied individually along with its metadata. In this case, if `<dest>` ends with a trailing slash `/`, it will be considered a directory and the contents of `<src>` will be written at `<dest>/base(<src>)`.
>
> If multiple `<src>` resources are specified, either directly or due to the use of a wildcard, then `<dest>` must be a directory, and it must end with a slash `/`.
>
> If `<src>` is a file, and `<dest>` doesn't end with a trailing slash, the contents of `<src>` will be written as filename `<dest>`.
>
> If `<dest>` doesn't exist, it's created, along with all missing directories in its path.

> **Note**
>
> The first encountered `COPY` instruction will invalidate the cache for all following instructions from the Dockerfile if the contents of `<src>` have changed. This includes invalidating the cache for `RUN` instructions. See the Dockerfile Best Practices guide â■■ Leverage build cache for more information.

COPY --from

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

```
COPY [--from=&lt;image|stage|context&gt;] &lt;src&gt; ... &lt;dest&gt;
```

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

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

FROM scratch
COPY --from=build /hello /
```

You can also copy files directly from other images. The following example copies an `nginx.conf` file from the official Nginx image.

```
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

> **Note**
>
> Only octal notation is currently supported. Non-octal support is tracked in moby/buildkit#1951.

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

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

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

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

COPY --link

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

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

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

```
# syntax=docker/dockerfile:1
FROM alpine
COPY --link /foo /bar
```

Is equivalent of doing two builds:

```
FROM alpine
```

and

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

and merging all the layers of both images together.

Benefits of using `--link`

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

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

Incompatibilities with `--link=false`

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

Note

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

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

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

```
# syntax=docker/dockerfile:1.7-labs
FROM scratch

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

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

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

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

```
# syntax=docker/dockerfile:1.7-labs
FROM scratch

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

Note

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

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

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

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

```
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:

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

ENTRYPOINT

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

`ENTRYPOINT` has two possible forms:

The exec form, which is the preferred form:

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

The shell form:

```
ENTRYPOINT command param1 param2
```

For more information about the different forms, see Shell and exec form.

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

```
$ 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

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

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

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

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

top - 08:25:00 up  7:27,  0 users,  load average: 0.00, 0.01, 0.05
Threads:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 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 debian:stable
RUN apt-get update && apt-get install -y --force-yes apache2
EXPOSE 80 443
VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

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

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

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

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

   exec gosu postgres "$@"
fi

exec "$@"
```

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

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

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

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

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

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

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

Which exits cleanly on `docker stop`:

```
$ /usr/bin/time docker stop test

test
real        0m 0.20s
user        0m 0.02s
sys        0m 0.04s
```

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

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

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

```
$ docker run -it --name test top --ignored-param2

top - 13:58:24 up 17 min,  0 users,  load average: 0.00, 0.00, 0.00
Tasks:   2 total,   1 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s): 16.7 us, 33.3 sy,  0.0 ni, 50.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 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

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

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

`ENTRYPOINT` should be defined when using the container as an executable.

`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.

`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

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

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

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

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

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

Notes about specifying volumes

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

Volumes on Windows-based containers: When using Windows-based containers, the destination of a volume inside the container must be one of:

- a non-existing or empty directory
- a drive other than `C:`

Changing the volume from within the Dockerfile: If any build steps change the data within the volume after it has been declared, those changes will be discarded.

JSON formatting: The list is parsed as a JSON array. You must enclose words with double quotes (`"`) rather than single quotes (`'`).

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

USER

```
USER <user>[:<group>]
```

or

```
USER <UID>[:<GID>]
```

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

> Note that when specifying a group for the user, the user will have *only* the specified group membership. Any other configured group memberships will be ignored.
>
> **Warning**
>
> When the user doesn't have a primary group then the image (or the next instructions) will be run with the `root` group.
>
> On Windows, the user must be created first if it's not a built-in account. This can be done with the `net user` command called as part of a Dockerfile.

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

WORKDIR

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

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

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

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

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

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

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

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

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

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

ARG

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

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` section to learn about secure ways to use secrets when building images.

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

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

Default values

An ARG instruction can optionally include a default value:

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

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

Scope

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

```
FROM busybox
USER ${username:-some_user}
ARG username
USER $username
# ...
```

A user builds this file by calling:

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

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

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

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

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

Using ARG variables

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

```
FROM ubuntu
ARG CONT_IMG_VER
ENV CONT_IMG_VER=v1.0.0
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 ubuntu
ARG CONT_IMG_VER
ENV CONT_IMG_VER=${CONT_IMG_VER:-v1.0.0}
RUN echo $CONT_IMG_VER
```

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

```
$ docker build .
```

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

The variable expansion technique in this example allows you to pass arguments from the command line and persist them in the final image by leveraging the ENV instruction. Variable expansion is only supported for a limited set of Dockerfile instructions.

Predefined ARGs

Docker has a set of predefined ARG variables that you can use without a corresponding ARG 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 ubuntu
RUN echo "Hello World"
```

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

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

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

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

Automatic platform ARGs in the global scope

This feature is only available when using the BuildKit 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 alpine
ARG TARGETPLATFORM
RUN echo "I'm building for $TARGETPLATFORM"
```

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]	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. Supported since Dockerfile 1.5, BuildKit 0.11

Example: keep `.git` dir

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

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

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

For example, consider these two Dockerfile:

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

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

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

Consider another example under the same command line:

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

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

ONBUILD

```
ONBUILD <INSTRUCTION>
```

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

Any build instruction can be registered as a trigger.

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

For example, if your image is a reusable Python application builder, it will require application source code to be added in a particular directory, and it might require a build script to be called after that. You can't just call `ADD` and `RUN` now, because you don't yet have access to the application source code, and it will be different for each application build. You could simply provide application developers with a boilerplate Dockerfile to copy-paste into their application, but that'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 ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
```

ONBUILD limitations

- Chaining `ONBUILD` instructions using `ONBUILD ONBUILD` isn't allowed.
- The `ONBUILD` instruction may not trigger `FROM` or `MAINTAINER` instructions.
- `ONBUILD COPY --from` is not supported.

STOPSIGNAL

```
STOPSIGNAL signal
```

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

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

HEALTHCHECK

The `HEALTHCHECK` instruction has two forms:

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

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

When a container has a healthcheck specified, it has a health status in addition to its normal status. This status is initially `starting`. Whenever a health check passes, it becomes `healthy` (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 curl -f http://localhost/ || exit 1
```

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

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

SHELL

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

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

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

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

```
FROM microsoft/windowsservercore

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

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

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

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

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

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

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

The command invoked by the builder will be:

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

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

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

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

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

Resulting in:

```
PS E:\myproject> docker build -t shell .

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


    Directory: C:\


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


---> 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

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

Example: Running a multi-line script

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

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

```
# syntax=docker/dockerfile:1
FROM debian
RUN <<EOT
 mkdir -p foo/bar
EOT
```

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

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

More complex examples may use multiple here-documents.

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

Example: Creating inline files

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

```
# syntax=docker/dockerfile:1
FROM alpine
COPY <<EOF greeting.txt
hello world
EOF
```

Regular here-doc variable expansion and tab stripping rules 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 alpine
ARG FOO=bar
COPY <<-EOT /script.sh
 echo "hello ${FOO}"
EOT
ENTRYPOINT ash /script.sh
```

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

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

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

```
# syntax=docker/dockerfile:1
FROM alpine
ARG FOO=bar
COPY <<-"EOT" /script.sh
```

```
 echo "hello ${FOO}"
EOT
ENTRYPOINT ash /script.sh
```

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

For examples of Dockerfiles, refer to:

- The "build images" section
- The "get started" tutorial
- The language-specific getting started guides
- The build guide

Value required ■■ ■■ ■■

For Docker-integrated BuildKit and `docker buildx build` ■■