Docker Commands List

Description

**The base command for the Docker CLI**

**Child commands**

**Command Description**

**docker attach Attach local standard input, output, and error streams to a running container**

**docker build Build an image from a Dockerfile**

**docker builder Manage builds**

**docker checkpoint Manage checkpoints**

**docker commit Create a new image from a container’s changes**

**docker compose You can use compose subcommand, docker compose [-f <arg>...] [options] [COMMAND] [ARGS...], to build and manage multiple services in Docker containers.**

**Child commands**

**Command Description**

docker compose build Build or rebuild services

docker compose convert Converts the compose file to platform’s canonical format

docker compose cp Copy files/folders between a service container and the local filesystem

docker compose create Creates containers for a service.

docker compose down Stop and remove containers, networks

docker compose events Receive real time events from containers.

docker compose exec Execute a command in a running container.

docker compose images List images used by the created containers

docker compose kill Force stop service containers.

docker compose logs View output from containers

docker compose ls List running compose projects

docker compose pause pause services

docker compose port Print the public port for a port binding.

docker compose ps List containers

docker compose pull Pull service images

docker compose push Push service images

docker compose restart Restart containers

docker compose rm Removes stopped service containers

docker compose run Run a one-off command on a service.

docker compose start Start services

docker compose stop Stop services

docker compose top Display the running processes

docker compose unpause unpause services

docker compose up Create and start containers

**docker config Manage Docker configs**

**docker container Manage containers**

**Child commands**

**Command Description**

docker container attach Attach local standard input, output, and error streams to a running container

docker container commit Create a new image from a container’s changes

docker container cp Copy files/folders between a container and the local filesystem

docker container create Create a new container

docker container diff Inspect changes to files or directories on a container’s filesystem

docker container exec Run a command in a running container

docker container export Export a container’s filesystem as a tar archive

docker container inspect Display detailed information on one or more containers

docker container kill Kill one or more running containers

docker container logs Fetch the logs of a container

docker container ls List containers

docker container pause Pause all processes within one or more containers

docker container port List port mappings or a specific mapping for the container

docker container prune Remove all stopped containers

docker container rename Rename a container

docker container restart Restart one or more containers

docker container rm Remove one or more containers

docker container run Run a command in a new container

docker container start Start one or more stopped containers

docker container stats Display a live stream of container(s) resource usage statistics

docker container stop Stop one or more running containers

docker container top Display the running processes of a container

docker container unpause Unpause all processes within one or more containers

docker container update Update configuration of one or more containers

docker container wait Block until one or more containers stop, then print their exit codes

**docker context Manage contexts**

**docker cp Copy files/folders between a container and the local filesystem**

**docker create Create a new container**

**docker diff Inspect changes to files or directories on a container’s filesystem**

**docker events Get real time events from the server**

**docker exec Run a command in a running container**

**docker export Export a container’s filesystem as a tar archive**

**docker history Show the history of an image**

**docker image Manage images**

**Child commands**

**Command Description**

**docker image build Build an image from a Dockerfile**

docker image history Show the history of an image

docker image import Import the contents from a tarball to create a filesystem image

docker image inspect Display detailed information on one or more images

docker image load Load an image from a tar archive or STDIN

docker image ls List images

docker image prune Remove unused images

docker image pull Pull an image or a repository from a registry

docker image push Push an image or a repository to a registry

docker image rm Remove one or more images

docker image save Save one or more images to a tar archive (streamed to STDOUT by default)

docker image tag Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE

**docker images List images**

**docker import Import the contents from a tarball to create a filesystem image**

**docker info Display system-wide information**

**docker inspect Return low-level information on Docker objects**

**docker kill Kill one or more running containers**

**docker load Load an image from a tar archive or STDIN**

**docker login Log in to a Docker registry**

**docker logout Log out from a Docker registry**

**docker logs Fetch the logs of a container**

**docker manifest Manage Docker image manifests and manifest lists**

**docker network Manage networks. You can use subcommands to create, inspect, list, remove, prune, connect, and disconnect networks.**

Child commands

Command Description

docker network connect Connect a container to a network

docker network create Create a network

docker network disconnect Disconnect a container from a network

docker network inspect Display detailed information on one or more networks

docker network ls List networks

docker network prune Remove all unused networks

docker network rm Remove one or more networks

**docker node Manage Swarm nodes**

**Child commands**

**Command Description**

docker node demote Demote one or more nodes from manager in the swarm

docker node inspect Display detailed information on one or more nodes

docker node ls List nodes in the swarm

docker node promote Promote one or more nodes to manager in the swarm

docker node ps List tasks running on one or more nodes, defaults to current node

docker node rm Remove one or more nodes from the swarm

docker node update Update a node

**docker pause Pause all processes within one or more containers**

**docker plugin Manage plugins**

**docker port List port mappings or a specific mapping for the container**

**docker ps List containers**

**docker pull Pull an image or a repository from a registry**

**docker push Push an image or a repository to a registry**

**docker rename Rename a container**

**docker restart Restart one or more containers**

**docker rm Remove one or more containers**

**docker rmi Remove one or more images**

**docker run Run a command in a new container**

**docker save Save one or more images to a tar archive (streamed to STDOUT by default)**

**docker search Search the Docker Hub for images**

**docker secret Manage Docker secrets**

**docker service Manage services**

**docker stack Manage Docker stacks**

**docker start Start one or more stopped containers**

**docker stats Display a live stream of container(s) resource usage statistics**

**docker stop Stop one or more running containers**

**docker swarm Manage Swarm**

**Child commands**

**Command Description**

docker swarm ca Display and rotate the root CA

docker swarm init Initialize a swarm

docker swarm join Join a swarm as a node and/or manager

docker swarm join-token Manage join tokens

docker swarm leave Leave the swarm

docker swarm unlock Unlock swarm

docker swarm unlock-key Manage the unlock key

docker swarm update Update the swarm

**docker system Manage Docker**

**docker tag Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE**

**docker top Display the running processes of a container**

**docker trust Manage trust on Docker images**

**docker unpause Unpause all processes within one or more containers**

**docker update Update configuration of one or more containers**

**docker version Show the Docker version information**

**docker volume Manage volumes**

**Related commands**

**Command Description**

docker volume create Create a volume

docker volume inspect Display detailed information on one or more volumes

docker volume ls List volumes

docker volume prune Remove all unused local volumes

docker volume rm Remove one or more volumes

**docker wait Block until one or more containers stop, then print their exit codes**

**-----------------------------------------------------------------------------**

**Docker Compose Commands List**

**docker-compose** Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

**Child Commands:**

**build Build or rebuild services**

**bundle Generate a Docker bundle from the Compose file**

**config Validate and view the Compose file**

**create Create services**

**down Stop and remove containers, networks, images, and volumes**

**events Receive real time events from containers**

**exec Execute a command in a running container**

**help Get help on a command**

**images List images**

**kill Kill containers**

**logs View output from containers**

**pause Pause services**

**port Print the public port for a port binding**

**ps List containers**

**pull Pull service images**

**push Push service images**

**restart Restart services**

**rm Remove stopped containers**

**run Run a one-off command**

**scale Set number of containers for a service**

**start Start services**

**stop Stop services**

**top Display the running processes**

**unpause Unpause services**

**up Create and start containers**

**version Show the Docker-Compose version information**

**-----------------------------------------------------------------------------**

**Docker Environment variables**

The following list of environment variables are supported by the docker command line:

**DOCKER\_API\_VERSION** Override the negotiated API version to use for debugging (e.g. 1.19)

**DOCKER\_CERT\_PATH** Location of your authentication keys. This variable is used both by the docker CLI and the dockerd daemon.Configures the path to the ca.pem, cert.pem, and key.pem files used for TLS verification. Defaults to ~/.docker.

**DOCKER\_CONFIG** The location of your client configuration files.

**DOCKER\_CONTENT\_TRUST\_SERVER** The URL of the Notary server to use. Defaults to the same URL as the registry.

**DOCKER\_CONTENT\_TRUST** When set Docker uses notary to sign and verify images. Equates to --disable-content-trust=false for build, create, pull, push, run.

**DOCKER\_CONTEXT** Name of the docker context to use (overrides DOCKER\_HOST env var and default context set with docker context use)

**DOCKER\_DEFAULT\_PLATFORM** Default platform for commands that take the --platform flag.

**DOCKER\_HIDE\_LEGACY\_COMMANDS** When set, Docker hides “legacy” top-level commands (such as docker rm, and docker pull) in docker help output, and only Management commands per object-type (e.g., docker container) are printed. This may become the default in a future release, at which point this environment-variable is removed.

**DOCKER\_HOST** Sets the URL of the docker daemon. As with the Docker client, defaults to unix:///var/run/docker.sock.

**DOCKER\_STACK\_ORCHESTRATOR** Configure the default orchestrator to use when using docker stack management commands.

**DOCKER\_TLS\_VERIFY** When set Docker uses TLS and verifies the remote. This variable is used both by the docker CLI and the dockerd daemon

**BUILDKIT\_PROGRESS** Set type of progress output (auto, plain, tty) when building with BuildKit backend. Use plain to show container output (default auto).

Because Docker is developed using Go, you can also use any environment variables used by the Go runtime. In particular, you may find these useful:

**HTTP\_PROXY**

**HTTPS\_PROXY**

**NO\_PROXY**

---------------------------------------------------------------------------

**Compose CLI environment variables**

**COMPOSE\_PROJECT\_NAME**

Sets the project name. This value is prepended along with the service name to the container on start up. For example, if your project name is myapp and it includes two services db and web, then Compose starts containers named myapp\_db\_1 and myapp\_web\_1 respectively.

**COMPOSE\_FILE**

Specify the path to a Compose file. If not provided, Compose looks for a file named docker-compose.yml in the current directory and then each parent directory in succession until a file by that name is found.

This variable supports multiple Compose files separated by a path separator (on Linux and macOS the path separator is :, on Windows it is ;). For example: COMPOSE\_FILE=docker-compose.yml:docker-compose.prod.yml. The path separator can also be customized using COMPOSE\_PATH\_SEPARATOR.

**COMPOSE\_PROFILES**

Specify one or multiple active profiles to enable. Calling docker-compose up with COMPOSE\_PROFILES=frontend will start the services with the profile frontend and services without specified profiles.

You can specify a list of profiles separated with a comma: COMPOSE\_PROFILES=frontend,debug will enable the profiles frontend and debug.

**COMPOSE\_API\_VERSION**

The Docker API only supports requests from clients which report a specific version. If you receive a client and server don't have same version error using docker-compose, you can workaround this error by setting this environment variable. Set the version value to match the server version.

Setting this variable is intended as a workaround for situations where you need to run temporarily with a mismatch between the client and server version. For example, if you can upgrade the client but need to wait to upgrade the server.

**COMPOSE\_HTTP\_TIMEOUT**

Configures the time (in seconds) a request to the Docker daemon is allowed to hang before Compose considers it failed. Defaults to 60 seconds.

**COMPOSE\_TLS\_VERSION**

Configure which TLS version is used for TLS communication with the docker daemon. Defaults to TLSv1. Supported values are: TLSv1, TLSv1\_1, TLSv1\_2.

**COMPOSE\_CONVERT\_WINDOWS\_PATHS**

Enable path conversion from Windows-style to Unix-style in volume definitions. Users of Docker Machine on Windows should always set this. Defaults to 0. Supported values: true or 1 to enable, false or 0 to disable.

**COMPOSE\_PATH\_SEPARATOR**

If set, the value of the COMPOSE\_FILE environment variable is separated using this character as path separator.

**COMPOSE\_FORCE\_WINDOWS\_HOST**

If set, volume declarations using the short syntax are parsed assuming the host path is a Windows path, even if Compose is running on a UNIX-based system. Supported values: true or 1 to enable, false or 0 to disable.

**COMPOSE\_IGNORE\_ORPHANS**

If set, Compose doesn’t try to detect orphaned containers for the project. Supported values: true or 1 to enable, false or 0 to disable.

**COMPOSE\_PARALLEL\_LIMIT**

Sets a limit for the number of operations Compose can execute in parallel. The default value is 64, and may not be set lower than 2.

**COMPOSE\_INTERACTIVE\_NO\_CLI**

If set, Compose doesn’t attempt to use the Docker CLI for interactive run and exec operations. This option is not available on Windows where the CLI is required for the aforementioned operations. Supported: true or 1 to enable, false or 0 to disable.

**COMPOSE\_DOCKER\_CLI\_BUILD**

Configure whether to use the Compose python client for building images or the native docker cli. By default, Compose uses the docker CLI to perform builds, which allows you to use BuildKit to perform builds.

Set COMPOSE\_DOCKER\_CLI\_BUILD=0 to disable native builds, and to use the built-in python client.

--------------------------------------------------------------------

Environment variables are supported by the following list of instructions in the Dockerfile or Instructions used in Docker file

**ADD : ADD has two forms:**

**ADD [--chown=<user>:<group>] <src>... <dest>**

**ADD [--chown=<user>:<group>] ["<src>",... "<dest>"]**

The latter form is required for paths containing whitespace.

Note

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

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

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

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

**To add all files starting with “hom”:**

**ADD hom\* /mydir/**

**ARG** :

ARG <name>[=<default value>]

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

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

*FROM busybox*

*ARG user1*

*ARG buildno*

# ...

**Warning:**

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

Default values

An ARG instruction can optionally include a default value:

*FROM busybox*

*ARG user1=someuser*

*ARG buildno=1*

# ...

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

Scope

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

*FROM busybox*

*USER ${user:-some\_user}*

*ARG user*

*USER $user*

# ...

A user builds this file by calling:

**docker build --build-arg user=what\_user .**

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

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

**FROM busybox**

**ARG SETTINGS**

**RUN ./run/setup $SETTINGS**

**FROM busybox**

**ARG SETTINGS**

**RUN ./run/other $SETTINGS**

Using ARG variables

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

*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**

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 .**

**CMD** : The CMD command​ specifies the instruction that is to be executed when a Docker container starts. ... For example, the user may need to run an executable .exe file or a Bash terminal as soon as the container starts – t​he CMD command can be used to handle such requests.

The CMD instruction has three forms:

CMD ["executable","param1","param2"] (exec form, this is the preferred form)

CMD ["param1","param2"] (as default parameters to ENTRYPOINT)

CMD command param1 param2 (shell form)

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

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

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

**Note**

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

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

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

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

***FROM ubuntu***

***CMD echo "This is a test." | wc -***

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

***FROM ubuntu***

***CMD ["/usr/bin/wc","--help"]***

If you would like your container to run the same executable every time, then you should consider using ENTRYPOINT in combination with CMD. See ENTRYPOINT.

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

**Note**

**Do not confuse RUN with CMD. RUN actually runs a command and commits the result; CMD does not execute anything at build time, but specifies the intended command for the image.**

**COPY**

**COPY has two forms:**

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

**COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]**

**This latter form is required for paths containing whitespace**

**Note**

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

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

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

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

To add all files starting with “hom”:

***COPY hom\* /mydir/***

**ENV**

**Syntax 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>.

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 ...

**EXPOSE**

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

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

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

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

**EXPOSE 80/udp**

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

**EXPOSE 80/tcp**

**EXPOSE 80/udp**

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

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

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

To set up port redirection on the host system, see using the -P flag. 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

**ENTRYPOINT**

ENTRYPOINT has two forms:

The exec form, which is the preferred form:

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

The shell form:

ENTRYPOINT command param1 param2

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

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

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

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

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

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

**Exec form ENTRYPOINT example**

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

***FROM ubuntu***

***ENTRYPOINT ["top", "-b"]***

***CMD ["-c"]***

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

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

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

docker exec -it test 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 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

docker top test

/usr/bin/time docker stop test

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

Which exits cleanly on docker stop:

/usr/bin/time docker stop test

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

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

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

docker exec -it test ps aux

/usr/bin/time docker stop test

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.

**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 – it is especially easy to start by pulling an image from the Public Repositories.

ARG is the only instruction that may precede FROM in the Dockerfile. .

FROM can appear multiple times within a single Dockerfile to create multiple images or use one build stage as a dependency for another. Simply make a note of the last image ID output by the commit before each new FROM instruction. Each FROM instruction clears any state created by previous instructions.

Optionally a name can be given to a new build stage by adding AS name to the FROM instruction. The name can be used in subsequent FROM and COPY --from=<name> instructions to refer to the image built in this stage.

The tag or digest values are optional. If you omit either of them, the builder assumes a latest tag by default. The builder returns an error if it cannot find the tag value.

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***

**HEALTHCHECK**

**The HEALTHCHECK instruction has two forms:**

**HEALTHCHECK [OPTIONS] CMD command (check container health by running a command inside the container)**

**HEALTHCHECK NONE (disable any healthcheck inherited from the base image)**

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

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

The options that can appear before CMD are:

--interval=DURATION (default: 30s)

--timeout=DURATION (default: 30s)

--start-period=DURATION (default: 0s)

--retries=N (default: 3)

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

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

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

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

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

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

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

0: success - the container is healthy and ready for use

1: unhealthy - the container is not working correctly

2: reserved - do not use this exit code

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

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

***CMD curl -f*** [***http://localhost/***](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.

**LABEL**

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

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

LABEL "com.example.vendor"="ACME Incorporated"

LABEL com.example.label-with-value="foo"

LABEL version="1.0"

LABEL description="This text illustrates \

that label-values can span multiple lines."

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

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

LABEL multi.label1="value1" \

multi.label2="value2" \

other="value3"

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

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

docker image inspect --format='' 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"

}

**RUN**

RUN has 2 forms:

RUN <command> (shell form, the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows)

RUN ["executable", "param1", "param2"] (exec form)

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

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

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

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

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

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

echo $HOME'

Together they are equivalent to this single line:

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

To use a different shell, other than ‘/bin/sh’, use the exec form passing in the desired shell. For example:

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

Note

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

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

Note

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

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

The correct syntax for this example is:

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

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

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

**STOPSIGNAL**

STOPSIGNAL signal

The STOPSIGNAL instruction sets the system call signal that will be sent to the container to exit. This signal can be a 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.

**USER**

USER <user>[:<group>]

or

USER <UID>[:<GID>]

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

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***

**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.

**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

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

ONBUILD <INSTRUCTION>

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

Any build instruction can be registered as a trigger.

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

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

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

Here’s how it works:

When it encounters an ONBUILD instruction, the builder adds a trigger to the metadata of the image being built. The instruction does not otherwise affect the current build.

At the end of the build, a list of all triggers is stored in the image manifest, under the key OnBuild. They can be inspected with the docker inspect command.

Later the image may be used as a base for a new build, using the FROM instruction. As part of processing the FROM instruction, the downstream builder looks for ONBUILD triggers, and executes them in the same order they were registered. If any of the triggers fail, the FROM instruction is aborted which in turn causes the build to fail. If all triggers succeed, the FROM instruction completes and the build continues as usual.

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

For example you might add something like this:

***ONBUILD ADD . /app/src***

***ONBUILD RUN /usr/local/bin/python-build --dir /app/src***

Warning

Chaining ONBUILD instructions using ONBUILD ONBUILD isn’t allowed.

Warning

The ONBUILD instruction may not trigger FROM or MAINTAINER instructions.

SHELL

SHELL ["executable", "parameters"]

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

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

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

***FROM microsoft/windowsservercore***

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

***RUN echo default***

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

***RUN powershell -command Write-Host default***

***# Executed as powershell -command Write-Host hello***

***SHELL ["powershell", "-command"]***

***RUN Write-Host hello***

***# Executed as cmd /S /C echo hello***

***SHELL ["cmd", "/S", "/C"]***

***RUN echo hello***

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

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

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

The command invoked by docker will be:

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

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

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

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

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

# escape=`

***FROM microsoft/nanoserver***

***SHELL ["powershell","-command"]***

***RUN New-Item -ItemType Directory C:\Example***

***ADD Execute-MyCmdlet.ps1 c:\example\***

***RUN c:\example\Execute-MyCmdlet -sample 'hello world'***