### Dockerfile Overview

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

**DockerFile Overview**

* 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.
* Docker 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 lines are removed before the Dockerfile instructions are executed

**The Dockerfile is basically a text file. But, unlike regular text files, you’ll see that it doesn’t have a .txt file extension. The Dockerfile is a file that you’ll save as Dockerfile, with no file extensions.**

Each Dockerfile is a script, composed of various commands (instructions) and arguments listed successively to automatically perform actions on a base image in order to create (or form) a new one. They are used for organizing things and greatly help with deployments by simplifying the process start-to-finish.

Dockerfiles begin with defining an image FROM which the build process starts. Followed by various other methods, commands and arguments (or conditions), in return, provide a new image which is to be used for creating docker containers.

***Commands in the Dockerfile are vital to building a Docker Image.***

* Every line of command in the Dockerfile creates the layers that make up the Docker image. Provided the Dockerfile remains the same, every time you build an image off it, it’s certain you’d get the same results. However, when you add a new line of command, Docker simply builds that layer and adds it to the existing layers.
* Just like the compiler or interpreter does to programming languages, Docker reads the Dockerfile from top to bottom. Hence, the placement of the commands matter a lot.
* The commands in the Dockerfile are not case sensitive.
* Comments in Dockerfiles are denoted by using the hash or pound symbol # at the beginning of the line. You should note that it only supports one-line comments, hence to write multi-line comments, you’ll use the hash symbol on each line.
* Careful though, not all hash symbols you see in a Dockerfile are comments. Hash symbols could also indicate parser directives. Parser directives are commands in the Dockerfile that indicate the way the Dockerfile should be read.

**Parser Directives**

* The "parser directives" instruct the Dockerfile parser to handle the content of Dockerfile as specified in the directives.
* For Dockerfile, Parser directives are optional. You can not define Parser directives , but once you want to define, Parser directives must be located in the first line of the Dockerfile.
* "Parser directives" direct the "way" how the content of DockerFile should be treated/interpreted. For this reason, its must be at the top of a Dockerfile.

Only two parser directives are available on Docker

escape and syntax parser directives

**Escape Directives**

# escape=\ (backslash)

Or

# escape=` (backtick)

* The escape directive sets the character used to escape characters in a Dockerfile. If not specified, the default escape character is \.
* The escape character is used both to escape characters in a line, and to escape a newline. This allows a Dockerfile instruction to span multiple lines. Note that regardless of whether the escape parser directive is included in a Dockerfile, escaping is not performed in a RUN command, except at the end of a line.
* Setting the escape character to ` is especially useful on Windows, where \ is the directory path separator. ` is consistent with Windows PowerShell.

**Example**

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

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:

**Solution**

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:\

### FROM

* 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.

***Syntax***

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

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

*--platform optional*

***Example***

*# The base image*

*FROM ubuntu:latest*

*FROM ubuntu:latest AS development*

**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 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.

RUN has 2 forms:

**Shell Form**

RUN <command> default is /bin/sh -c on Linux or cmd /S /C on Windows

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'

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

* SHELL ["/bin/bash", "-c"]
* RUN echo I am now using bash!

***SHELL FORM Example***

*RUN apt−get −y install vim*

*RUN apt−get −y update*

*RUN apt−get −y update \*

*&& apt−get −y install firefox \*

*&& apt−get −y install vim*

**EXEC FORM**

* RUN ["executable", "param1", "param2"]

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.

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: In the JSON form, it is necessary to escape backslashes

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

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

**Cache on RUN**

* 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.

### CMD

* 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.
* If you want to run a docker container by specifying a default command that gets executed for all the containers of that image by default, you can use a CMD command.
* In case you specify a command during the docker run command, it overrides the default one. Specifying more than one CMD instructions, will allow only the last one to get executed.
* These defaults can include an executable, or they can omit the executable, in which case you must specify an ENTRYPOINT instruction as well.

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)

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

**Note**:

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

*# default command*

*CMD ["/start.sh"]*

*CMD ["supervisord", "-c", "/etc/supervisor.conf"]*

### EntryPoint

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

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

*ENTRYPOINT has two forms:*

*The exec form, which is the preferred form:*

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

*The shell form:*

*ENTRYPOINT command param1 param2*

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.

**Exec form ENTRYPOINT**

**Example #1**

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

**Example #2**

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:

**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**

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>.

**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.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **No ENTRYPOINT** | **ENTRYPOINT exec\_entry p1\_entry** | **ENTRYPOINT [“exec\_entry”, “p1\_entry”]** |
| **No CMD** | *error, not allowed* | /bin/sh -c exec\_entry p1\_entry | exec\_entry p1\_entry |
| **CMD [“exec\_cmd”, “p1\_cmd”]** | exec\_cmd p1\_cmd | /bin/sh -c exec\_entry p1\_entry | exec\_entry p1\_entry exec\_cmd p1\_cmd |
| **CMD [“p1\_cmd”, “p2\_cmd”]** | p1\_cmd p2\_cmd | /bin/sh -c exec\_entry p1\_entry | exec\_entry p1\_entry p1\_cmd p2\_cmd |
| **CMD exec\_cmd p1\_cmd** | /bin/sh -c exec\_cmd p1\_cmd | /bin/sh -c exec\_entry p1\_entry | exec\_entry p1\_entry /bin/sh -c exec\_cmd p1\_cmd |

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

### LABEL

*Maintianer deprciated*

*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."*

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 --format='' myimage*

### EXPOSE

*EXPOSE <port> [<port>/protocol>…]*

*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*

* 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.
* When you put EXPOSE 80 (or any port you want) in your Dockerfile that’s going to tell Docker that your container’s service can be connected to on port 80.
* Containers on the same network can talk to each other regardless of whether or not you exposed or published your ports back to the Docker host with -p.

You can expose a port through your Dockerfile or use --expose and then publish it with the -P flag. This will bind the exposed port to your Docker host on a random port (verified by running docker container ls).

You can expose a port through your Dockerfile or use --expose and then publish it with the -p 80:80 flag. This will bind the exposed port to your Docker host on port 80, and it expects the exposed port is 80 too (adjust as necessary with HOST:CONTAINER).

You can ignore exposing anything and just use -p 80:80 in which case this doubles as both exposing AND publishing the port.

### ENV

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.

*ENV MY\_NAME="John Doe"*

*ENV MY\_DOG=Rex\ The\ Dog*

*ENV 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>.

* 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 ...

### ADD

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>.

*Syntax*

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

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

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

* To add all files starting with “hom”:
  + ADD hom\* /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/

* For example, to add a file named arr[0].txt

ADD arr[[]0].txt /mydir/ # need to escape using golang rules

* All new files and directories are created with a UID and GID of 0, unless the optional --chown flag specifies a given username, groupname, or UID/GID combination to request specific ownership of the content added.
* The format of the --chown flag allows for either username and groupname strings or direct integer UID and GID in any combination
* Providing a username without groupname or a UID without GID will use the same numeric UID as the GID

ADD --chown=55:mygroup files\* /somedir/

ADD --chown=bin files\* /somedir/

ADD --chown=1 files\* /somedir/

ADD --chown=10:11 files\* /somedir/

* If the container root filesystem does not contain either /etc/passwd or /etc/group files and either user or group names are used in the --chown flag, the build will fail on the ADD operation. Using numeric IDs requires no lookup and will not depend on container root filesystem content.
* In the case where <src> is a remote file URL, the destination will have permissions of 600.
* If 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 does not support authentication.

**ADD obeys the following rules:**

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

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

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

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

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

If <dest> does not end with a trailing slash, it will be considered a regular file and the contents of <src> will be written at <dest>

If <dest> doesn’t exist, it is created along with all missing directories in its path

### COPY

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

*Syntax*

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

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

* To add all files starting with “hom”:
  + COPY hom\* /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/
* For example, to copy a file named arr[0].txt, use the following;
  + COPY arr[[]0].txt /mydir/

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

*COPY --chown=55:mygroup files\* /somedir/*

*COPY --chown=bin files\* /somedir/*

*COPY --chown=1 files\* /somedir/*

*COPY --chown=10:11 files\* /somedir/*

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

COPY obeys the following rules:

* The <src> path must be inside the context of the build; you cannot COPY ../something /something, because the first step of a docker build is to send the context directory (and subdirectories) to the docker daemon.
* If <src> is a directory, the entire contents of the directory are copied, including filesystem metadata.
* If <src> is any other kind of file, it is copied individually along with its metadata. In this case, if <dest> ends with a trailing slash /, it will be considered a directory and the contents of <src> will be written at <dest>/base(<src>).
* If multiple <src> resources are specified, either directly or due to the use of a wildcard, then <dest> must be a directory, and it must end with a slash /.
* If <dest> does not end with a trailing slash, it will be considered a regular file and the contents of <src> will be written at <dest>.
* If <dest> doesn’t exist, it is created along with all missing directories in its path.

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

**COPY with –from flag**

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.

<https://medium.com/@tonistiigi/advanced-multi-stage-build-patterns-6f741b852fae>

### 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.
* 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

* 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.
* When the user doesn’t have a primary group then the image (or the next instructions) will be run with the root group.

USER <user>[:<group>]

or

USER <UID>[:<GID>]

### WORKDIR

* WORKDIR /path/to/workdir
* The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile. If the WORKDIR doesn’t exist, it will be created even if it’s not used in any subsequent Dockerfile instruction.
* The WORKDIR instruction can be used multiple times in a Dockerfile. If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction. For example:

WORKDIR /a

WORKDIR b

WORKDIR c

RUN pwd

* The output of the final pwd command in this Dockerfile would be /a/b/c.
* The WORKDIR instruction can resolve environment variables previously set using ENV. You can only use environment variables explicitly set in the Dockerfile. For example:

ENV DIRPATH=/path

WORKDIR $DIRPATH/$DIRNAME

RUN pwd

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

### ARG

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.

User can pass the input

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

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

* FROM busybox
* ARG user1
* ARG buildno

**WARNING**

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

Refer to the “build images with BuildKit” section to learn about secure ways to use secrets when building images.

**#1 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.

#2 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

**#3 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.

**#4 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 .
* 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.
* 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.

### ONBUILD

### STOPSIGNAL

STOPSIGNAL signal

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

### HEALTHCHECK

The HEALTHCHECK instruction has two forms:

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

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

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

The options that can appear before CMD are:

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

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

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

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

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

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

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

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

* 0: success - the container is healthy and ready for use
* 1: unhealthy - the container is not working correctly
* 2: reserved - do not use this exit code

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

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

CMD curl -f http://localhost/ || exit 1

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

When the health status of a container changes, a health\_status event is generated with the new status.

### SHELL

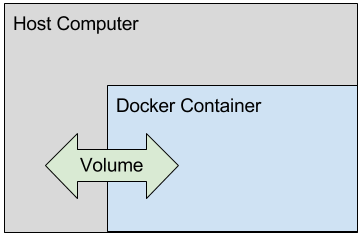
* SHELL ["executable", "parameters"]
* The SHELL instruction allows the default shell used for the shell form of commands to be overridden. The default shell on Linux is ["/bin/sh", "-c"], and on Windows is ["cmd", "/S", "/C"]. The SHELL instruction must be written in JSON form in a Dockerfile.
* The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

### COPY vs ADD vs VOLUME

* To sum up – use **COPY**. As Docker itself suggests, avoid the ADD command unless you need to extract a local tar file.
* COPY takes in a src and destination. It only lets you copy in a local file or directory from your host (the machine building the Docker image) into the Docker image itself.
* ADD lets you do that too, but it also supports 2 other sources. First, you can use a URL instead of a local file / directory. Secondly, you can extract a tar file from the source directly into the destination.
* COPY only supports the basic copying of local files into the container, while ADD has some features (like local-only tar extraction and remote URL support) that are not immediately obvious. Consequently, the best use for ADD is local tar file auto-extraction into the image, as in ADD rootfs.tar.xz /.
* Because image size matters, using ADD to fetch packages from remote URLs is strongly discouraged; you should use curl or wgetinstead. That way you can delete the files you no longer need after they’ve been extracted and you don’t have to add another layer in your image.
* Really, the only reason to use ADD is when you have an archive file that you definitely want to have auto-extracted into the image.
* ADD unpacks archives from the host machine, it does not unpack files from URLs. To unpack an archive you just use it in its default form; ADD <src>... <dest>

**VOLUME**

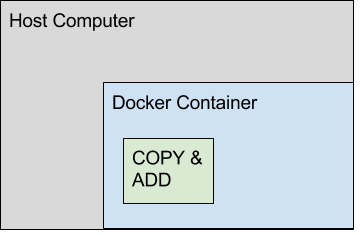


VOLUME is different from COPY and ADD because it creates a mount point that the host operating system can interact with.

VOLUME /Users/sjobs/Documents/cool-project /var/www

* This command syncs the Docker container’s /var/www directory with the host OS’s cool-project directory. When any change is made to cool-project within the host OS it is immediately made available to the docker container mounting that directory and vice versa.
* Using VOLUME is especially useful for development environments where frequent modifications to the code are being made. For example, running nodemon on a Docker container with a VOLUME allows you to make changes to your code that are immediately reflected as if Docker were invisible.

**COPY**



* COPY accomplishes everything that VOLUME accomplishes, but does it during build time. This is necessary for configuring containers with modified config files with services such as httpd, Nginx, MongoDB, etc… For example when configuring an Nginx container modifying the nginx.conf file is an operation that must be done with COPY.
* One setback to COPY is that it makes only the container that was copied to aware of the contents. Meaning we cannot access the contents on our host computer, or on other running docker containers, something VOLUME does.

**ADD**

* ADD is exactly the same as COPY except for one distinct difference. ADD can fetch content from a URL and it will not extract the content, only downloading to the specific location. Check the bellow example.
* ADD https://www-us.apache.org/dist//httpd/httpd-2.4.41.tar.gz /linuxteacher/from\_url
* If the local content is recognized as compressed, will uncompress the contents. Check the bellow example.
* ADD abc.tar.gz /linuxteacher/from\_local

### RUN vs CMD vs ENTRYPOINT

RUN executes command(s) in a new layer and creates a new image. E.g., it is often used for installing software packages.

CMD sets default command and/or parameters, which can be overwritten from command line when docker container runs.

ENTRYPOINT configures a container that will run as an executable. An ENTRYPOINT command, unlike CMD, does not ignore additional parameters that you specify in your Docker run command. Default parameters that cannot be overridden when Docker Containers run with CLI parameters

**Bottom Line**

* Use RUN instructions to build your image by adding layers on top of initial image.
* Prefer ENTRYPOINT to CMD when building executable Docker image and you need a command always to be executed. Additionally use CMD if you need to provide extra default arguments that could be overwritten from command line when docker container runs.
* Choose CMD if you need to provide a default command and/or arguments that can be overwritten from command line when docker container runs

### Dockerignore File

The docker build command builds Docker images from a Dockerfile and a “context”. A build’s context is the set of files located in the specified PATH or URL. The build process can refer to any of the files in the context. For example, your build can use a COPY instruction to reference a file in the context.

The URL parameter can refer to three kinds of resources: Git repositories, pre-packaged tarball contexts and plain text files.

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

Matching is done using Go’s filepath.Match rules.

# comment Ignored.

\*/temp\* Exclude files and directories whose names start with temp in any immediate subdirectory of the root. For example, the plain file /somedir/temporary.txt is excluded, as is the directory /somedir/temp.

\*/\*/temp\* Exclude files and directories starting with temp from any subdirectory that is two levels below the root. For example, /somedir/subdir/temporary.txt is excluded.

temp? Exclude files and directories in the root directory whose names are a one-character extension of temp. For example, /tempa and /tempb are excluded.

**Reason**

**Security Issues** – Some important files such as passwords, secret keys, .git folders, etc contain a lot of information about your project and you might now want to expose those details to the outside world to prevent intrusion.

**Cache Invalidation** – When you write the Dockerfile, it’s a general practice to use the COPY instruction to copy the files and folders inside the Docker build context. Each statement inside the Dockerfile results in building a new intermediate image layer. Hence, when you make changes in your dockerfile again and again, this might lead to multiple Cache Invalidation and leads to wastage of resources.

Also, excluding unnecessary large files from your Docker Build Context will lead to lower Docker Image size.

It speeds up the process of building the Docker Image.

### Docker Build Cache

<https://pythonspeed.com/articles/docker-caching-model/>

Packaging can often be slow, and Docker builds are no exception. Downloading and installing system and Python packages, compiling C extensions, building assets—it all adds up.

In order to speed up your builds, Docker implements caching: if your Dockerfile and related files haven’t changed, a rebuild can reuse some of the existing layers in your local image cache.

But in order to take advantage of this cache, you need to understand how it works, and that’s what we’ll cover in this article.

**The Basic Algorithm**

When you build a Dockerfile, Docker will see if it can use the cached results of previous builds:

For most commands, if the text of the command hasn’t changed, the version from the cache will be used.

For COPY, it also checks that the files you’re copying haven’t changed.

For the first time I will run all the commands and download all form scratch.

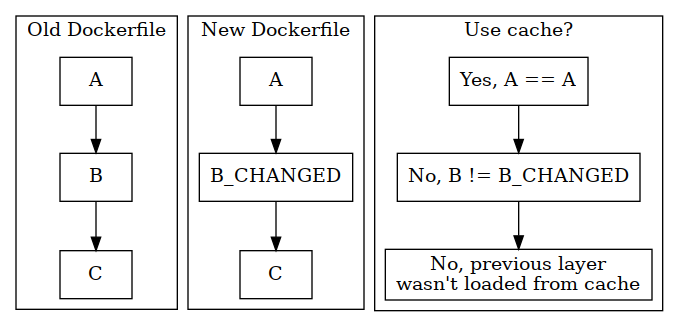
The second time, however, because nothing has changed docker build will use the image cache:

**Taking advantage of caching**

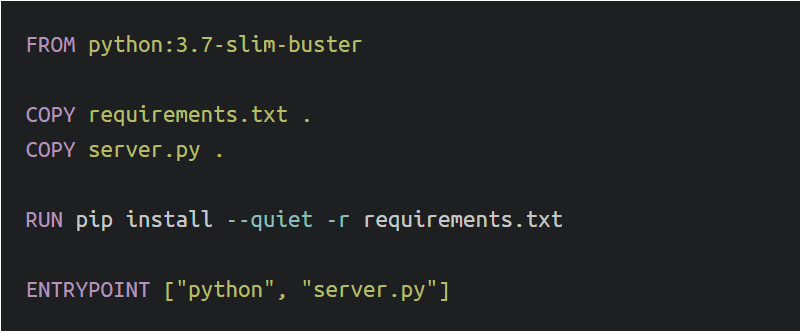
here’s one more important rule to the caching algorithm:

If the cache can’t be used for a particular layer, all subsequent layers won’t be loaded from the cache.

In the following example the C layer hasn’t changed between new and old Dockerfiles. Nonetheless, it still can’t be loaded from the cache since the previous layer (B\_CHANGED) couldn’t be loaded from the cache:



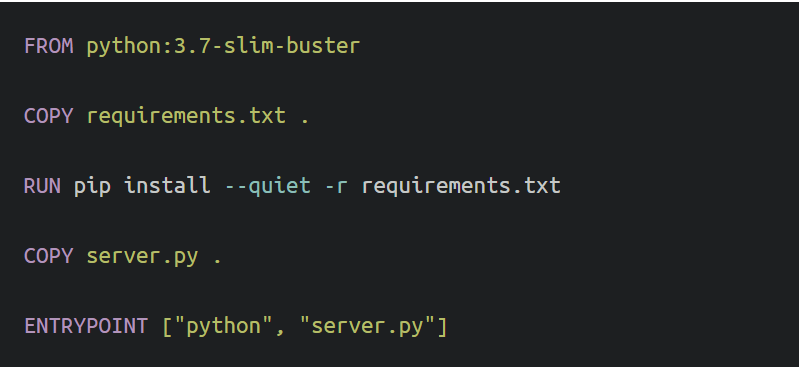
Let’s consider what that means for the following Dockerfile:



If any of the files we COPY in change, that invalidates all later layers: we’ll need to rerun pip install, for example.

But if server.py has changed but requirements.txt hasn’t, why should we have to redo the pip install? After all, the pip install only uses requirements.txt.

What you want to do therefore is to copy only those files that you actually need to run the next step, so as to minimize the opportunity for cache invalidation. For example:



Because server.py is only copied in after the pip install, the layer created by pip install can still be loaded from the cache so long as requirements.txt hasn’t changed.

**Designing your Dockerfile for caching**

If you want fast builds by reusing your previously cached builds, you’ll need to write your Dockerfile appropriately:

Only copy in the files you need for the next step, to minimize cache invalidation in the build process.

Make sure not to invalidate the cache accidentally by having an command early in the Dockerfile that always changes, e.g. a LABEL that contains the build timestamp.

### Docker Layering / Build Cache

<https://docs.semaphoreci.com/ci-cd-environment/docker-layer-caching/>

<https://thenewstack.io/understanding-the-docker-cache-for-faster-builds/>

<https://www.ctl.io/developers/blog/post/more-docker-image-cache-tips/>

<https://betterprogramming.pub/how-to-improve-docker-image-size-with-layers-3ad62be0da9b>

### DockerFile Best Practices

<https://docs.docker.com/develop/dev-best-practices/>

<https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#create-ephemeral-containers>

<https://sysdig.com/blog/dockerfile-best-practices/>