**Dockerfile ultimate guide**

FROM busybox

FROM scratch

*scratch image is the most minimal image in Docker. This is the base ancestor for all other images. The scratch image is actually empty. It doesn't contain any folders/files.*

## **Use multi-stage builds**

Multi-stage builds let you reduce the size of your final image, by creating a cleaner separation between the building of your image and the final output. Split your Dockerfile instructions into distinct stages to make sure that the resulting output only contains the files that's needed to run the application.

Using multiple stages can also let you build more efficiently by executing build steps in parallel.

# Multi stage example…

[FROM](https://docs.docker.com/reference/dockerfile/#from) golang:1.21

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

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

package main

import "fmt"

func main() {

fmt.Println("hello, world")

}

EOF

[RUN](https://docs.docker.com/reference/dockerfile/#run) go build -o /bin/hello ./main.go

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

[COPY](https://docs.docker.com/reference/dockerfile/#copy) --from=0 /bin/hello /bin/hello

[CMD](https://docs.docker.com/reference/dockerfile/#cmd) ["/bin/hello"]

## [Name your build stages](https://docs.docker.com/build/building/multi-stage/#name-your-build-stages)

# syntax=docker/dockerfile:1

FROM golang:1.21 AS myfirstphase

WORKDIR /src

COPY <<EOF /src/main.go

package main

import "fmt"

func main() {

fmt.Println("hello, world")

}

EOF

RUN go build -o /bin/hello ./main.go

FROM scratch

COPY --from= myfirstphase /bin/hello /bin/hello

CMD ["/bin/hello"]

## Stop at a specific build stage

*When you build your image, you don't necessarily need to build the entire Dockerfile including every stage. You can specify a target build stage.*

docker build --target myfirstphase -t hello .

Copy files from another image

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

Use previous stage as new stage

# use previous stage

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

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

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

[COPY](https://docs.docker.com/reference/dockerfile/#copy) source1.cpp source.cpp

[RUN](https://docs.docker.com/reference/dockerfile/#run) g++ -o /binary source.cpp

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

[COPY](https://docs.docker.com/reference/dockerfile/#copy) source2.cpp source.cpp

[RUN](https://docs.docker.com/reference/dockerfile/#run) g++ -o /binary source.cpp

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

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

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

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

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

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

### COPY --chown –chmod

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

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

COPY --chown=<user>:<group> <hostPath> <containerPath>

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

*When you use COPY --link, you’re telling Docker to create a symbolic link (symlink) instead of copying the files directly. This feature is not standard across all build platforms and might require you to enable experimental features in your Docker configuration.*

*The main reason is to save space when copying between stages in a multi-stage Docker build.*

# Stage 1 - Build stage

FROM node:14 AS builder

WORKDIR /app

COPY package.json package-lock.json ./

RUN npm install

COPY . .

RUN npm run build

# Stage 2 - Run stage

FROM node:14-slim

WORKDIR /app

COPY --from=builder /app/build /app

CMD ["node", "server.js"]

With the --link flag, it becomes:

# Stage 2 - Run stage

...

COPY --link --from=builder /app/build /app

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

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

*The path expression follows the same format as <src>, supporting wildcards and matching using Go's filepath.Match*

*rules. For example, to add all files starting with "hom", excluding files with a .txt extension:*

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

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

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

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

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

* [--keep-git-dir](https://docs.docker.com/reference/dockerfile/#add---keep-git-dir)
* [--checksum](https://docs.docker.com/reference/dockerfile/#add---checksum)
* [--chown](https://docs.docker.com/reference/dockerfile/#add---chown---chmod)
* [--chmod](https://docs.docker.com/reference/dockerfile/#add---chown---chmod)
* [--link](https://docs.docker.com/reference/dockerfile/#add---link)
* [--exclude](https://docs.docker.com/reference/dockerfile/#add---exclude)

Docker's official documentation notes that users should always choose **COPY** over **ADD** since it is a more transparent and straightforward command.

*The* ***ADD*** *command use is discouraged for all cases except when the user wants to extract a local compressed file. For copying remote files, the* [*run command*](https://phoenixnap.com/kb/docker-run-command-with-examples) *combined with* [*wget*](https://phoenixnap.com/kb/wget-command-with-examples) *or* [*curl*](https://phoenixnap.com/kb/curl-command) *is safer and more efficient. This method avoids creating an additional image layer and saves space.*

## USER

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

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

# Test building docker image from Dockerfile

ARG myimage

FROM $myimage

LABEL maintainer="moh.imam@protonmail.com"

docker build --build-arg myimage=alpine .

***To set default argument value:***  
 ARG myimage=alpine

**VAR***Environment variables are the variables which can be used in both build and run time. It is possible to set the variables in the Dockerfile before running the container and use them in your project without any problem. Unlike ARG, ENVs can be used in the CMD command and in the project code.*

# Test building docker image from Dockerfile

FROM alpine

LABEL maintainer="moh.imam@protonmail.com"

# Install Node and NPM

RUN apk add --update nodejs npm curl

ENV foo=123

COPY . /src

WORKDIR /src

# Install dependencies

RUN npm install

EXPOSE 9090

ENTRYPOINT ["node", "./app.js"]

docker run -itd -e foo=987 --name varimg testenv:beta

docker run --rm -d --name foocontainer --env-file=./.env fooimage

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

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

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

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

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

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

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

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

**Docker cache**

*Using the Docker build cache speeds up builds by reusing layers from previous builds.*

*Using layer caching during docker build is why the order of your steps is essential. If you change a step, all the steps below get built again.*

sometimes, you want to build an image without the cache. For debugging a build, you may need to start from clean slate. Or to force a dependency to get upgraded.

docker build --no-cache .

## Exclude with .dockerignore

*To exclude files not relevant to the build, without restructuring your source repository, use a .dockerignore file. This file supports exclusion patterns similar to .gitignore files.*

.dockerignore

\*.md

# BuildKit

[BuildKit](https://github.com/moby/buildkit)

is an improved backend to replace the legacy builder. BuildKit is the default builder for users on Docker Desktop, and Docker Engine as of version 23.0.

BuildKit provides new functionality and improves your builds' performance. It also introduces support for handling more complex scenarios:

* Detect and skip executing unused build stages
* Parallelize building independent build stages
* Incrementally transfer only the changed files in your [build context](https://docs.docker.com/build/building/context/) between builds
* Detect and skip transferring unused files in your [build context](https://docs.docker.com/build/building/context/)
* Use [Dockerfile frontend](https://docs.docker.com/build/dockerfile/frontend/) implementations with many new features
* Avoid side effects with rest of the API (intermediate images and containers)
* Prioritize your build cache for automatic pruning

## Differences between legacy builder and BuildKit

The legacy Docker Engine builder processes all stages of a Dockerfile leading up to the selected --target. It will build a stage even if the selected target doesn't depend on that stage.

[BuildKit](https://docs.docker.com/build/buildkit/) only builds the stages that the target stage depends on.

For example, given the following Dockerfile:

# syntax=docker/dockerfile:1

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

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

[FROM](https://docs.docker.com/reference/dockerfile/#from) base AS stage1

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo "stage1"

[FROM](https://docs.docker.com/reference/dockerfile/#from) base AS stage2

[RUN](https://docs.docker.com/reference/dockerfile/#run) echo "stage2"

With [BuildKit enabled](https://docs.docker.com/build/buildkit/#getting-started), building the stage2 target in this Dockerfile means only base and stage2 are processed. There is no dependency on stage1, so it's skipped.

DOCKER\_BUILDKIT=1 docker build --no-cache -f Dockerfile --target stage2 .

[+] Building 0.4s (7/7) FINISHED

=> [internal] load build definition from Dockerfile 0.0s

=> => transferring dockerfile: 36B 0.0s

=> [internal] load .dockerignore 0.0s

=> => transferring context: 2B 0.0s

=> [internal] load metadata for docker.io/library/ubuntu:latest 0.0s

=> CACHED [base 1/2] FROM docker.io/library/ubuntu 0.0s

=> [base 2/2] RUN echo "base" 0.1s

=> [stage2 1/1] RUN echo "stage2" 0.2s

=> exporting to image 0.0s

=> => exporting layers 0.0s

=> => writing image sha256:f55003b607cef37614f607f0728e6fd4d113a4bf7ef12210da338c716f2cfd15 0.0s

On the other hand, building the same target without BuildKit results in all stages being processed:

DOCKER\_BUILDKIT=0 docker build --no-cache -f Dockerfile --target stage2 .

Sending build context to Docker daemon 219.1kB

Step 1/6 : FROM ubuntu AS base

---> a7870fd478f4

Step 2/6 : RUN echo "base"

---> Running in e850d0e42eca

base

Removing intermediate container e850d0e42eca

---> d9f69f23cac8

Step 3/6 : FROM base AS stage1

---> d9f69f23cac8

Step 4/6 : RUN echo "stage1"

---> Running in 758ba6c1a9a3

stage1

Removing intermediate container 758ba6c1a9a3

---> 396baa55b8c3

Step 5/6 : FROM base AS stage2

---> d9f69f23cac8

Step 6/6 : RUN echo "stage2"

---> Running in bbc025b93175

stage2

Removing intermediate container bbc025b93175

---> 09fc3770a9c4

Successfully built 09fc3770a9c4

The legacy builder processes stage1, even if stage2 doesn't depend on it.

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

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

ENTRYPOINT has two possible forms:

## *Shell and exec form*

Shell:  
ENTRYPOINT python app.py

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

ENTRYPOINT ["python", "app.py"]

*Like ENTRYPOINT, you can write CMD in either exec or shell form. The key difference is that CMD sets default commands or parameters you can override from the command line. Meanwhile, ENTRYPOINT configures containers to run as executables, meaning you can’t override the command from the command line.*

*You can use CMD to extend ENTRYPOINT’s functionality to give your image greater flexibility. Combining the two allows you to customize your image’s behavior, with the CMD values acting as default arguments for the ENTRYPOINT instruction. This method lets you set a default command via ENTRYPOINT and default arguments via CMD.*

ENTRYPOINT ["python", "myapp.py"]

CMD ["--help"]

*In this example, starting a* [*Docker container*](https://kinsta.com/blog/docker-security/) *without providing any command* line arguments means **python myapp.py --help** will execute by default. However, providing arguments when starting the container (such as **docker run mypyimage --version**) will replace the default CMD arguments, resulting in python app.py --version. This approach gives you greater flexibility when running your containers.

## Override ENTRYPOINT

docker run --entrypoint mypyimage “bin/sh”

COPY myentrypoint.sh /myentrypoint.sh

RUN chmod +x /myentrypoint.sh

ENTRYPOINT ["/myentrypoint.sh"]

Myentrypoint.sh:

#!/bin/bash

echo "Starting my application..."

./my-app -f "${username}"

FROM ubuntu

ENTRYPOINT ["top", "-b"]

CMD ["-c"]

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

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

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

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

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

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

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

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

Any build instruction can be registered as a trigger.

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

ONBUILD ADD . /app/src

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

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

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

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

The image's default stopsignal can be overridden per container, using the --stop-signal flag on

docker stop 09466f183c19 --signal 9

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

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

The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers. The value can be a JSON array, VOLUME ["/var/log/"], or a plain string with multiple arguments, such as VOLUME /var/log or VOLUME /var/log /var/db

FROM ubuntu

RUN mkdir /myvol

RUN echo "hello world" > /myvol/greeting

VOLUME /myvol

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

The HEALTHCHECK instruction has two forms:

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

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

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

HEALTHCHECK CMD curl --fail http://localhost:8080 || exit 1

HEALTHCHECK --interval=1m CMD <command>

FROM nginx:latest  
  
# Define a healthcheck command  
HEALTHCHECK --interval=30s --timeout=10s --retries=3 CMD curl -f http://localhost/ || exit 1

### [Creating inline files](https://docs.docker.com/reference/dockerfile/#example-creating-inline-files)

# syntax=docker/dockerfile:1

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

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

hello world

EOF

Building best practices:  
https://docs.docker.com/build/building/best-practices/