**Overview of Dockerfile**

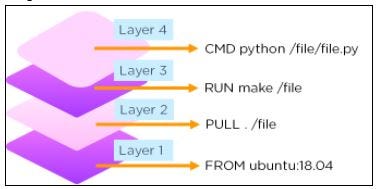
In this blog, we will look at what a dockerfile is. as well as its general guidelines for writing it. We’ll also go over the fundamental commands used in writing the dockerfile.

## **What is a Dockerfile?**

A Dockerfile is a text document that has all the command line instructions needed to assemble an image. With the help of a Dockerfile, users can create an automated build that executes several command-line instructions in succession.

Docker builds images automatically by reading instructions from a Dockerfile, which is a text file that contains all commands needed to build a given image in the correct order. A Dockerfile follows a predefined format and set of instructions.

A Docker image is made up of read-only layers, each representing a Dockerfile instruction. The layers are stacked, and each one is a delta of the previous layer’s changes. Below is the diagrammatic representation of the dockerfile.



Images source from simplilearn.com

Dockerfile consists of specific commands that guide you on how to build a specific Docker image.

The specific commands you can use in a dockerfile are:

FROM, PULL, RUN, and CMD

* FROM — Creates a layer from the ubuntu:18.04
* PULL — Adds files from your Docker repository.
* RUN — Builds your container.
* CMD — Specifies what command to run within the container.

A new writable layer known as the container layer is added on top of the underlying layers when you run an image and create a container. This writable container layer receives all writes made to the running container, including new file creation, file modification, and file deletion.

Here is the format to write the dockerfile

# Comment  
INSTRUCTION arguments

Docker runs instructions in a Dockerfile in order. A Dockerfile must begin with a FROM instruction.

The Parent Image that you are building from is specified in the FROM instruction. Only one or more ARG instructions, which declare arguments used in FROM lines in the Dockerfile, may come before a FROM instruction.

Let us see some of the common commands used in the dockerfile and their syntax.

# ****FROM****

The FROM instruction starts a new build stage and establishes the Base Image for subsequent instructions. As a result, every valid Dockerfile must begin with a FROM instruction. The image can be any valid image — starting with one from the Public Repositories is especially simple.

FROM can appear multiple times in a single Dockerfile to create multiple images or as a dependency for another build stage. Simply make a note of the last image ID output by the commit before proceeding to the next FROM instruction. Each FROM instruction clears any previous instructions’ states.

Syntax   
  
FROM [--platform=<platform>] <image> [AS <name>]  
  
FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]  
  
FROM [--platform=<platform>] <image>[@<digest>] [AS <name>]

# ****Run****

Any commands in a new layer on top of the current image will be executed by the RUN instruction, which will also commit the results. The following step in the Dockerfile will make use of the committed image that was created.

The principles of Docker, which include cheap commits and the ability to create containers from any point in an image’s history, akin to source control, are adhered to by layering RUN instructions and generating commits.

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

# ****CMD****

A Dockerfile can only contain one CMD instruction. If you enter more than one CMD, only the last one will be executed.

A CMD’s primary function is to provide defaults for an executing container. These defaults can include or exclude an executable, in which case you must also specify an ENTRYPOINT instruction.

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

The CMD instruction has three forms of syntax  
  
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)

# ****ENTRYPOINT****

You can configure an ENTRYPOINT to run as an executable container.

Docker run image> command line arguments will be appended after all elements in an exec form ENTRYPOINT and will override all elements specified using CMD. This enables arguments to be passed to the entry point; for example, docker run image> -d will send the -d argument to the entry point. The docker run — entrypoint flag can be used to override the ENTRYPOINT instruction.

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

Only the final ENTRYPOINT instruction in the Dockerfile will be executed.

The exec form, which is the preferred form:  
  
ENTRYPOINT ["executable", "param1", "param2"]  
  
The shell form:  
  
ENTRYPOINT command param1 param2

**How CMD and ENTRYPOINT interact?**

When running a container, the CMD and ENTRYPOINT instructions both define which command is executed. There are few ground rules that govern their collaboration.

1. At least one of the CMD or ENTRYPOINT commands should be specified in the Dockerfile.
2. When using the container as an executable, ENTRYPOINT must be defined.
3. CMD should be used to specify default arguments for an ENTRYPOINT command or to run an ad hoc command in a container.
4. When running the container with alternative arguments, CMD will be overridden.

# ****SHELL****

The SHELL instruction overrides the default shell used for the shell form of commands. On Linux, the default shell is [“/bin/sh”, “-c”], and on Windows, it is [“cmd”, “/S”, “/C”]. In a Dockerfile, the SHELL instruction must be written in JSON.

The SHELL instruction is especially useful on Windows, where there are two widely used and very different native shells: cmd and powershell, as well as alternative shells such as sh.

The SHELL instruction may appear more than once. Each SHELL instruction overrides all preceding SHELL instructions and has an impact on all subsequent instructions.

Syntax  
  
SHELL ["executable", "parameters"]  
  
Example  
  
SHELL ["powershell", "-command"]  
RUN echo Hello

# ****HEALTHCHECK****

The HEALTHCHECK instruction instructs Docker on how to test a container to ensure that it is still operational. This can detect cases such as a web server that is stuck in an infinite loop and unable to handle new connections, despite the server process still being active.

When a container is marked with a healthcheck, it has a health status in addition to its normal status. This status is just getting started. When a health check passes, it becomes healthy (regardless of its previous state). It becomes unhealthy after a certain number of consecutive failures.

Syntax  
  
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 options that can appear before CMD are:

* — interval=DURATION (default: 30s)
* — timeout=DURATION (default: 30s)
* — start-period=DURATION (default: 0s)
* — start-interval=DURATION (default: 5s)
* — retries=N (default: 3)

The health check will be performed at intervals of seconds after the container is started, and then at intervals of seconds after each previous check has been completed.

If a single check run takes more than timeout seconds, the check is considered to have failed.

The container is considered unhealthy after three consecutive failures of the health check.

Containers that require time to bootstrap benefit from the start period. Probe failure during that period will not be counted towards the maximum number of retries. If a health check succeeds during the start period, the container is considered started, and all subsequent failures are counted towards the maximum number of retries.

The time between health checks during the start period is referred to as the start interval. Docker Engine version 25.0 or later is required for this option.

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

0: success — the container is in good condition and ready for use.  
1: unhealthy — the container is not functioning properly  
2: reserved — this exit code should not be used.

# ****VOLUME****

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

The docker run command populates the newly created volume with any data that exists in the base image at the specified location.

Syntax  
  
VOLUME ["/data"]  
  
Example  
  
FROM ubuntu  
RUN mkdir /myvol  
RUN echo "hello world" > /myvol/greeting  
VOLUME /myvol  
  
The image produced by this Dockerfile instructs Docker run to create a   
new mount point at /myvol and copy the greeting file into the volume that   
has just been created.

This Dockerfile generates an image that instructs Docker to create a new mount point at /myvol and copy the greeting file into the newly created volume.

# ****ENV****

The ENV instruction changes the value of the environment variable key> to value>. Because the value will be interpreted for other environment variables, quote characters that are not escaped will be removed. Quotes and backslashes, like command line parsing, can be used to include spaces within values.

Syntax  
ENV <key>=<value> ...  
  
Example  
ENV MY\_NAME="John Doe"

Multiple <key>=<value>… variables can be set at the same time using the ENV instruction.

# ****EXPOSE****

Docker is informed by the EXPOSE instruction that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, with TCP being the default if no protocol is specified.

The port is not actually published by the EXPOSE instruction. It serves as a kind of documentation between the person who creates the image and the person who runs the container, indicating which ports are intended to be published. 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, to actually publish the port when running the container.

EXPOSE assumes TCP by default. You can also use UDP.

Syntax  
  
EXPOSE <port> [<port>/<protocol>...]  
  
Example  
  
EXPOSE 80/udp

# ****ADD****

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

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

Syntax  
  
ADD [--chown=<user>:<group>] [--chmod=<perms>] [--checksum=<checksum>] <src>... <dest>  
ADD [--chown=<user>:<group>] [--chmod=<perms>] ["<src>",... "<dest>"]  
  
Example  
  
ADD hom\* /mydir/  
  
ADD --chown=myuser:mygroup --chmod=655 files\* /somedir/

**Adding a git repository**

This form allows you to add a git repository to an image without using the git command within the image.

Syntax  
  
ADD [--keep-git-dir=<boolean>] <git ref> <dir>  
  
Example  
  
FROM alpine  
ADD --keep-git-dir=true https://github.com/moby/buildkit.git#v0.10.1 /buildkit

**Adding a private repository**

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

FROM alpine  
ADD git@git.example.com:foo/bar.git /bar

**Advantages of using the dockerfile**

* Automatically get bugfixes without updating the Docker daemon.
* Make sure all users are using the same implementation to build your Dockerfile.
* Use the latest features without updating the Docker daemon.
* Try out new features or third-party features before they are integrated in the Docker daemon.

I have explained a few commands that are commonly used in the dockerfile; however, other commands may be used in the dockerfile depending on the situation.

**Conclusion**

With that, we have completed the article on what is a dockerfile. We discussed what docker is, what a dockerfile is, the syntax of a dockerfile, important commands to create a docker image using a dockerfile, and how to build a docker image using a dockerfile in this article.

Thanks for Reading.

I’m filled with joy to present this article, eager to ignite your curiosity with fresh insights. Let your applause resonate by clicking that heartfelt 👏 button below, a gesture that means the world to us.