NAME

Dockerfile - automate the steps of creating a Docker image

INTRODUCTION

The **Dockerfile** is a configuration file that automates the steps of creating a Docker image. It is similar to a Makefile. Docker reads instructions from the **Dockerfile** to automate the steps otherwise performed manually to create an image. To build an image, create a file called **Dockerfile**.

The **Dockerfile** describes the steps taken to assemble the image. When the **Dockerfile** has been created, call the `docker build` command, using the path of directory that contains **Dockerfile** as the argument.

SYNOPSIS

INSTRUCTION arguments

For example:

FROM image

DESCRIPTION

A Dockerfile is a file that automates the steps of creating a Docker image. A Dockerfile is similar to a Makefile.

USAGE

docker build .

-- Runs the steps and commits them, building a final image.
The path to the source repository defines where to find the context of the build. The build is run by the Docker daemon, not the CLI. The whole context must be transferred to the daemon. The Docker CLI reports `"Sending build context to Docker daemon"` when the context is sent to the daemon.

docker build -t repository/tag .

-- specifies a repository and tag at which to save the new image if the build succeeds. The Docker daemon runs the steps one-by-one, committing the result to a new image if necessary, before finally outputting the ID of the new image. The Docker daemon automatically cleans up the context it is given.

Docker re-uses intermediate images whenever possible. This significantly accelerates the *docker build* process.

FORMAT

```
FROM image

FROM image:tag

FROM image@digest
```

-- The **FROM** instruction sets the base image for subsequent instructions. A valid Dockerfile must have **FROM** as its first instruction. The image can be any valid image. It is easy to start by pulling an image from the public repositories.

-- **FROM** must be the first non-comment instruction in Dockerfile.

-- **FROM** may appear multiple times within a single Dockerfile in order to create multiple images. Make a note of the last image ID output by the commit before each new **FROM** command.

-- If no tag is given to the **FROM** instruction, Docker applies the `latest` tag. If the used tag does not exist, an error is returned.

-- If no digest is given to the **FROM** instruction, Docker applies the `latest` tag. If the used tag does not exist, an error is returned.

MAINTAINER

-- **MAINTAINER** sets the Author field for the generated images. Useful for providing users with an email or url for support.

RUN

-- **RUN** has two forms:

```
# the command is run in a shell - /bin/sh -c
RUN <command>

# Executable form
RUN ["executable", "param1", "param2"]
```

-- The **RUN** instruction executes any commands in a new layer on top of the current image and commits the results. The committed image is used for the next step in 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 the history of an image. This is similar to source control. The exec form makes it possible to avoid shell string munging. The exec form makes it possible to **RUN** commands using a base image that does not contain `/bin/sh`.

Note that the exec form is parsed as a JSON array, which means that you must

use double-quotes (") around words not single-quotes (’).

CMD
-- **CMD** has three forms:

```
# Executable form
CMD ["executable", "param1", "param2"]‘

# Provide default arguments to ENTRYPOINT
CMD ["param1", "param2"]‘

# the command is run in a shell - /bin/sh -c
CMD command param1 param2
```

-- There should be only one **CMD** in a Dockerfile. If more than one **CMD** is listed, only the last **CMD** takes effect.
The main purpose of a **CMD** is to provide defaults for an executing container. These defaults may include an executable, or they can omit the executable. If they omit the executable, an **ENTRYPOINT** must be specified.
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**, the `<command>` executes in `/bin/sh -c`:

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

```
FROM ubuntu
CMD echo "This is a test." | wc -
```

-- If you run **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 form of **CMD**. All additional parameters must be individually expressed as strings in the array:

```
FROM ubuntu
CMD ["/usr/bin/wc","--help"]
```

-- To make the container run the same executable every time, use **ENTRYPOINT** in combination with **CMD**.
If the user specifies arguments to `docker run`, the specified commands override the default in **CMD**.
Do not confuse **RUN** with **CMD**. **RUN** runs a command and commits the result. **CMD** executes nothing at build time, but specifies the intended command for the image.

LABEL
-- `LABEL <key>=<value> [<key>=<value> ...]`or

```
LABEL <key>[ <value>]
LABEL <key>[ <value>]
...
```

The **LABEL** instruction adds metadata to an image. A **LABEL** is a key-value pair. To specify a **LABEL** without a value, simply use an empty string. To include spaces within a **LABEL** value, use quotes and backslashes as you would in command-line parsing.

```
LABEL com.example.vendor="ACME Incorporated"
LABEL com.example.vendor "ACME Incorporated"
LABEL com.example.vendor.is-beta ""
LABEL com.example.vendor.is-beta=
LABEL com.example.vendor.is-beta=""
```

An image can have more than one label. To specify multiple labels, separate each key-value pair by a space.

Labels are additive including `LABEL`s in `FROM` images. As the system encounters and then applies a new label, new `keys` override any previous labels with identical keys.

To display an image's labels, use the `docker inspect` command.

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**, 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(1)** and **docker-create(1)**.

EXPOSE
-- `EXPOSE <port> [<port>...]`
The **EXPOSE** instruction informs Docker that the container listens on the specified network ports at runtime. Docker uses this information to interconnect containers using links and to set up port redirection on the host system.

ENV
-- `ENV <key> <value>`
The **ENV** instruction sets the environment variable to the value `<value>`. This value is passed to all future **RUN**, **ENTRYPOINT**, and **CMD** instructions. This is

functionally equivalent to prefixing the command with `<key>=<value>`. The environment variables that are set with **ENV** persist when a container is run from the resulting image. Use `docker inspect` to inspect these values, and change them using `docker run --env <key>=<value>`.

Note that setting "`ENV DEBIAN_FRONTEND=noninteractive`" may cause unintended consequences, because it will persist when the container is run interactively, as with the following command: `docker run -t -i image bash`

ADD

-- **ADD** has two forms:

```
ADD <src> <dest>

# Required for paths with whitespace
ADD ["<src>",... "<dest>"]
```

The **ADD** instruction copies new files, directories or remote file URLs to the filesystem of the container at path `<dest>`. Multiple `<src>` resources may be specified but if they are files or directories then they must be relative to the source directory that is being built (the context of the build). The `<dest>` is the absolute path, or path relative to **WORKDIR**, into which the source is copied inside the target container. If the `<src>` argument is a local file in a recognized compression format (tar, gzip, bzip2, etc) then it is unpacked at the specified `<dest>` in the container's filesystem. Note that only local compressed files will be unpacked, i.e., the URL download and archive unpacking features cannot be used together. All new directories are created with mode 0755 and with the uid and gid of **0**.

COPY

-- **COPY** has two forms:

```
COPY <src> <dest>

# Required for paths with whitespace
COPY ["<src>",... "<dest>"]
```

The **COPY** instruction copies new files from `<src>` and adds them to the filesystem of the container at path . The `<src>` must be the path to a file or directory relative to the source directory that is being built (the context of the build) or a remote file URL. The `<dest>` is an absolute path, or a path relative to **WORKDIR**, into which the source will be copied inside the target container. If you **COPY** an archive file it will land in the container exactly as it appears in the build context without any attempt to unpack it. All new files and directories are created with mode **0755** and with the uid and gid of **0**.

ENTRYPOINT

-- **ENTRYPOINT** has two forms:

```
# executable form
ENTRYPOINT ["executable", "param1", "param2"]‘

# run command in a shell - /bin/sh -c
ENTRYPOINT command param1 param2
```

-- An **ENTRYPOINT** helps you configure a container that can be run as an executable. When you specify an **ENTRYPOINT**, the whole container runs as if it was only that executable. The **ENTRYPOINT** instruction adds an entry command that is not overwritten when arguments are passed to docker run. This is different from the behavior of **CMD**. This allows arguments to be passed to the entrypoint, for instance `docker run <image> -d` passes the -d argument to the **ENTRYPOINT**. Specify parameters either in the **ENTRYPOINT** JSON array (as in the preferred exec form above), or by using a **CMD** statement. Parameters in the **ENTRYPOINT** are not overwritten by the docker run arguments. Parameters specified via **CMD** are overwritten by docker run arguments. Specify a plain string for the **ENTRYPOINT**, and it will execute in `/bin/sh -c`, like a **CMD** instruction:

```
FROM ubuntu
ENTRYPOINT wc -l -
```

This means that the Dockerfile’s image always takes stdin as input (that’s what "-" means), and prints the number of lines (that’s what "-l" means). To make this optional but default, use a **CMD**:

```
FROM ubuntu
CMD ["-l", "-"]
ENTRYPOINT ["/usr/bin/wc"]
```

VOLUME

-- `VOLUME ["/data"]`

The **VOLUME** instruction creates a mount point with the specified name and marks it as holding externally-mounted volumes from the native host or from other containers.

USER

-- `USER daemon`

Sets the username or UID used for running subsequent commands.

The **USER** instruction can optionally be used to set the group or GID. The followings examples are all valid:

USER [user | user:group | uid | uid:gid | user:gid | uid:group]

Until the **USER** instruction is set, instructions will be run as root. The USER instruction can be used any number of times in a Dockerfile, and will only affect subsequent commands.

WORKDIR
-- `WORKDIR /path/to/workdir`
The **WORKDIR** instruction sets the working directory for the **RUN**, **CMD**, **ENTRYPOINT**, **COPY** and **ADD** Dockerfile commands that follow it. It can be used multiple times in a single Dockerfile. Relative paths are defined relative to the path of the previous **WORKDIR** instruction. For example:

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

In the above example, the output of the **pwd** command is **a/b/c**.

ARG
-- 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.

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

The Dockerfile author can define a single variable by specifying `ARG` once or many variables by specifying `ARG` more than once. For example, a valid Dockerfile:

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

A Dockerfile author may optionally specify a default value for an `ARG` instruction:

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

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

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:

```
1 FROM busybox
2 USER ${user:-some_user}
3 ARG user
4 USER $user
...
```

A user builds this file by calling:

```
$ docker build --build-arg user=what_user Dockerfile
```

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.

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.

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.

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER=v1.0.0
4 RUN echo $CONT_IMG_VER
```

Then, assume this image is built with this command:

```
$ docker build --build-arg CONT_IMG_VER=v2.0.1 Dockerfile
```

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:

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER=${CONT_IMG_VER:-v1.0.0}
4 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 Dockerfile
```

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. ⟨#environment-replacement⟩

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 `--build-arg` flag, for example:

```
$ docker build --build-arg HTTPS_PROXY=https://my-proxy.example.com .
```

ONBUILD

-- `ONBUILD [INSTRUCTION]`

The **ONBUILD** instruction adds a trigger instruction to an image. The trigger is executed at a later time, when the image is used as the base for another build. Docker executes the trigger in the context of the downstream build, as if the trigger existed immediately after the **FROM** instruction in the downstream Dockerfile.

You can register any build instruction as a trigger. A trigger is useful if you are defining an image to use as a base for building other images. For example, if you are defining an application build environment or a daemon that is customized with a user-specific configuration.

Consider an image intended as a reusable python application builder. It must add application source code to a particular directory, and might need a build

script 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 is different for each application build.

-- Providing application developers with a boilerplate Dockerfile to copy-paste into their application is inefficient, error-prone, and difficult to update because it mixes with application-specific code. The solution is to use **ONBUILD** to register instructions in advance, to run later, during the next build stage.

HISTORY

*May 2014, Compiled by Zac Dover (zdover at redhat dot com) based on docker.com Dockerfile documentation. *Feb 2015, updated by Brian Goff (cpuguy83@gmail.com) for readability *Sept 2015, updated by Sally O'Malley (somalley@redhat.com) *Oct 2016, updated by Addam Hardy (addam.hardy@gmail.com)