Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'Containerfile.5' command

$ man Containerfile.5

CONTAINERFILE(5) Container User Manuals CONTAINERFILE(5)

NAME

Containerfile(Dockerfile) - automate the steps of creating a container image

INTRODUCTION

The Containerfile is a configuration file that automates the steps of creating a container image. It is similar to a Makefile. Container en? gines (Podman, Buildah, Docker) read instructions from the Container? file to automate the steps otherwise performed manually to create an image. To build an image, create a file called Containerfile.

The Containerfile describes the steps taken to assemble the image. When the Containerfile has been created, call the buildah bud, podman build, docker build command, using the path of context directory that contains Containerfile as the argument. Podman and Buildah default to Container? file and will fall back to Dockerfile. Docker only will search for Dockerfile in the context directory.

Dockerfile is an alternate name for the same object. Containerfile and Dockerfile support the same syntax.

SYNOPSIS

INSTRUCTION arguments

For example:

FROM image

DESCRIPTION

A Containerfile is a file that automates the steps of creating a con?

tainer image. A Containerfile is similar to a Makefile.

USAGE

buildah bud .

podman 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.

buildah bud -t repository/tag .

podman build -t repository/tag .

-- specifies a repository and tag at which to save the new image if the

build

succeeds. The container engine runs the steps one-by-one, committing

the result

to a new image if necessary, before finally outputting the ID of the

new

image.

Container engines re-use intermediate images whenever possible. This

significantly

accelerates the build process.

FORMAT

FROM image

FROM image:tag

FROM image@digest

-- The FROM instruction sets the base image for subsequent instruc?

tions. A

valid Containerfile must have either ARG or *FROM** as its first in?

struction.

If FROM is not the first instruction in the file, it may only be pre?

ceded by

one or more ARG instructions, which declare arguments that are used

in the next FROM line in the Containerfile.

The image can be any valid image. It is easy to start by pulling an image from the public repositories.

-- FROM must appear at least once in the Containerfile.

-- FROM The first FROM command must come before all other instructions in the Containerfile except ARG

-- FROM may appear multiple times within a single Containerfile 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, container engines apply the latest tag. If the used tag does not exist, an error is returned.

-- If no digest is given to the FROM instruction, container engines apply 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"]
```

RUN mounts

--mount=type=TYPE,TYPE-SPECIFIC-OPTION[,...]

Attach a filesystem mount to the container

Current supported mount TYPES are bind, cache, secret and tmpfs.

```
e.g.
mount=type=bind,source=/path/on/host,destination=/path/in/container
```

mount=type=tmpfs,tmpfs-size=512M,destination=/path/in/container

mount=type=secret,id=mysecret cat /run/secrets/mysecret

Common Options:

? src, source: mount source spec for bind and volume. Mandatory for bind. If `from` is specified, `src` is the subpath in the `from` field.

? dst, destination, target: mount destination spec.

? ro, read-only: true (default) or false.

Options specific to bind:

? bind-propagation: shared, slave, private, rshared, rslave, or rprivate(default). See also mount(2).

. bind-nonrecursive: do not setup a recursive bind mount. By default it is recursive.

? from: stage or image name for the root of the source. Defaults to the build context.

? rw, read-write: allows writes on the mount.

Options specific to tmpfs:

? tmpfs-size: Size of the tmpfs mount in bytes. Unlimited by default in Linux.

? tmpfs-mode: File mode of the tmpfs in octal. (e.g. 700 or 0700.) Defaults to 1777 in Linux.

? tmpcopyup: Path that is shadowed by the tmpfs mount is recursively copied up to the tmpfs itself.

Options specific to cache:

? id: Create a separate cache directory for a particular id.

? mode: File mode for new cache directory in octal. Default 0755.

? ro, readonly: read only cache if set.

? uid: uid for cache directory.

? gid: gid for cache directory.

? from: stage name for the root of the source. Defaults to host cache directory.

? rw, read-write: allows writes on the mount.

RUN --network

RUN --network allows control over which networking environment the com?

mand is run in.

Syntax: --network=<TYPE>

Network types

??

?Type ? Description ?

??

?default ? ?

??

??#run---networkdefault? (default) ? Run in the default network. ?

??

?none ? ?

??

??#run---networknone? ? Run with no network access. ?

??

?host ? ?

??

??#run---networkhost? ? Run in the host's network ?

? ? environment. ?

??

RUN --network=default

Equivalent to not supplying a flag at all, the command is run in the default network for the build.

RUN --network=none

The command is run with no network access (lo is still available, but is isolated to this process).

Example: isolating external effects

```
FROM python:3.6
ADD mypackage.tgz wheels/
RUN --network=none pip install --find-links wheels mypackage
```

pip will only be able to install the packages provided in the tarfile, which can be controlled by an earlier build stage.

RUN --network=host

The command is run in the host's network environment (similar to buil? dah build --network=host, but on a per-instruction basis)

RUN Secrets

The RUN command has a feature to allow the passing of secret informa? tion into the image build. These secrets files can be used during the RUN command but are not committed to the final image. The RUN command supports the --mount option to identify the secret file. A secret file from the host is mounted into the container while the image is being

built.

Container engines pass secret the secret file into the build using the --secret flag.

--mount=type=secret,TYPE-SPECIFIC-OPTION[,...]

? id is the identifier for the secret passed into the buildah bud --secret or podman build --secret. This identifier is associated with the RUN --mount identifier to use in the Containerfile.

? dst|target|destination rename the secret file to a specific file in the Containerfile RUN command to use.

? type=secret tells the --mount command that it is mounting in a secret file

```
# shows secret from default secret location:
RUN --mount=type=secret,id=mysecret cat /run/secrets/mysecret
# shows secret from custom secret location:
RUN --mount=type=secret,id=mysecret,dst=/foobar cat /foobar
```

The secret needs to be passed to the build using the --secret flag. The final image built does not container the secret file:

```
buildah bud --no-cache --secret id=mysecret,src=mysecret.txt .
```

-- 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 Containerfile.

-- Layering RUN instructions and generating commits conforms to the core concepts of container engines 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 Containerfile. 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 exe? cutable. 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 com? mand 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 ENTRY?

POINT in

combination with CMD.

If the user specifies arguments to podman run or docker run, the

specified commands

override the default in CMD.

Do not confuse RUN with CMD. RUN runs a command and commits the re?

sult.

CMD executes nothing at build time, but specifies the intended com?

mand 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, sep?

arate
each key-value pair by a space.
Labels are additive including LABELs in FROM images. As the system
encounters and then applies a new label, new keys override any previ?
ous
labels with identical keys.
To display an image's labels, use the buildah inspect command.
EXPOSE
-- EXPOSE <port> [<port>...]
The EXPOSE instruction informs the container engine that the con?
tainer listens on the
specified network ports at runtime. The container engine 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 podman inspect to inspect these values,
and
change them using podman 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: podman 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 con?

tainer.

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 EN?
TRYPOINT,
the whole container runs as if it was only that executable. The EN?
TRYPOINT
instruction adds an entry command that is not overwritten when argu?
ments are
passed to podman run. This is different from the behavior of CMD.
This allows
arguments to be passed to the entrypoint, for instance podman 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 podman run arguments. Parameters specified via CMD are overwritten by podman 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 Containerfile'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

following 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 Containerfile, 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 Containerfile commands that follow it. It can be used multiple times in a single Containerfile. Relative paths are defined relative to the path of the previous WORKDIR instruction. For exam? ple:

```
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 podman build and buildah build commands using the --build-arg <varname>=<value> flag. If a user specifies a build argu? ment that was not defined in the Containerfile, the build outputs a warning.

Note that a second FROM in a Containerfile sets the values associated with an Arg variable to nil and they must be reset if they are to be used later in the Containerfile

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

The Containerfile author can define a single variable by specifying ARG

once or many

variables by specifying ARG more than once. For example, a valid Con?

tainerfile:

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

A Containerfile 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 Containerfile not from the argument's use on the com?

mand-line or

elsewhere. For example, consider this Containerfile:

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

A user builds this file by calling:

```
$ podman build --build-arg user=what_user Containerfile
```

The USER at line 2 evaluates to some_user as the user variable is de?

fined 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 podman 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 Containerfile 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:

```
$ podman build --build-arg CONT_IMG_VER=v2.0.1 Containerfile
```

In this case, the RUN instruction uses v1.0.0 instead of the ARG set?ting

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 defini?tion.

Using the example above but a different ENV specification you can cre?ate 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 podman build without the --build-arg flag:

$ podman build Containerfile

Using this Containerfile 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 Containerfile instructions. ?#environment-replacement?

Container engines have a set of predefined ARG variables that you can use without a corresponding ARG instruction in the Containerfile.

? HTTP_PROXY

? http_proxy

? HTTPS_PROXY

? https_proxy

? FTP_PROXY

? ftp_proxy

? NO_PROXY

? no_proxy

? ALL_PROXY

? all_proxy

To use these, pass them on the command line using --build-arg flag, for example:

$ podman 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. Container engines execute the trigger in the context of the downstream build, as if the trigger existed immediately after the FROM instruction in the downstream Containerfile.

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 Containerfile 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.

SEE ALSO

buildah(1), podman(1), docker(1)

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)

Aug 2021, converted Dockerfile man page to Containerfile by Dan Walsh (dwalsh@redhat.com)

Aug 2021 CONTAINERFILE(5)