Docker RUN vs CMD vs ENTRYPOINT

Some Docker instructions look similar and cause confusion among developers who just started using Docker or do it irregularly. In this post I will explain the difference between CMD, RUN, and ENTRYPOINT on examples.

In a nutshell

- 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.

If it doesn't make much sense or you after details, then read on.

Docker images and layers

When Docker runs a container, it runs an *image* inside it. This image is usually built by executing Docker instructions, which add *layers* on top of existing image or *OS distribution*. *OS distribution* is the initial image and every added layer creates a new image.

Final Docker *image* reminds an onion with OS distribution inside and a number of layers on top of it. For example, your image can be built by installing a number of deb packages and your application on top of Ubuntu 14.04 distribution.

Shell and Exec forms

All three instructions (RUN, CMD and ENTRYPOINT) can be specified in *shell* form or *exec* form. Let's get familiar with these forms first, because the forms usually cause more confusion than instructions themselves.

Shell form

`<instruction> <command>`

Examples:

```
RUN apt-get install python3
CMD echo "Hello world"
ENTRYPOINT echo "Hello world"
```

When instruction is executed in *shell* form it calls `/bin/sh -c <command>` under the hood and normal shell processing happens. For example, the following snippet in Dockerfile

```
ENV name John Dow
ENTRYPOINT echo "Hello, $name"
```

when container runs as `docker run -it <image>` will produce output

```
Hello, John Dow
```

Note that variable *name* is replaced with its value.

Exec form

This is the preferred form for CMD and ENTRYPOINT instructions.

`<instruction> ["executable", "param1", "param2", ...]`

Examples:

```
RUN ["apt-get", "install", "python3"]
CMD ["/bin/echo", "Hello world"]
ENTRYPOINT ["/bin/echo", "Hello world"]
```

When instruction is executed in *exec* form it calls executable directly, and shell processing does not happen. For example, the following snippet in Dockerfile

```
ENV name John Dow
ENTRYPOINT ["/bin/echo", "Hello, $name"]
```

when container runs as `docker run -it <image>` will produce output

```
Hello, $name
```

Note that variable *name* is not substituted.

How to run bash?

If you need to run *bash* (or any other interpreter but sh), use *exec* form with `/bin/bash` as executable. In this case, normal shell processing will take place. For example, the following snippet in Dockerfile

```
ENV name John Dow
ENTRYPOINT ["/bin/bash", "-c", "echo Hello, $name"]
```

when container runs as `docker run -it <image>` will produce output

```
Hello, John Dow
```

RUN

RUN instruction allows you to install your application and packages requited for it. It executes any commands on top of the current image and creates a new layer by committing the results. Often you will find multiple RUN instructions in a Dockerfile.

RUN has two forms:

- `RUN <command>` (shell form)
- `RUN ["executable", "param1", "param2"]` (exec form)

(The forms are described in detail in *Shell and Exec forms* section above.)

A good illustration of RUN instruction would be to install multiple version control systems packages:

```
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
  git \
  mercurial \
  subversion
```

Note that `apt-get update` and `apt-get install` are executed in a single RUN instruction. This is done to make sure that the latest packages will be installed. If `apt-get install` were in a separate RUN instruction, then it would reuse a layer added by `apt-get update`, which could had been created a long time ago.

CMD

CMD instruction allows you to set a *default* command, which will be executed only when you run container without specifying a command. If Docker container runs with a command, the default command will be ignored. If Dockerfile has more than one CMD instruction, all but last CMD instructions are ignored.

CMD has three forms:

- `CMD ["executable","param1","param2"]` (exec form, preferred)
- `CMD ["param1","param2"]` (sets additional default parameters for ENTRYPOINT in *exec* form)
- `CMD command param1 param2` (shell form)

Again, the first and third forms were explained in *Shell and Exec forms* section. The second one is used together with ENTRYPOINT instruction in *exec* form. It sets default parameters that will be added after ENTRYPOINT parameters if container runs without command line arguments. See ENTRYPOINT for example.

Let's have a look how CMD instruction works. The following snippet in Dockerfile

```
CMD echo "Hello world"
```

when container runs as `docker run -it <image>` will produce output

```
Hello world
```

but when container runs with a command, e.g., `docker run -it <image> /bin/bash`, CMD is ignored and bash interpreter runs instead:

```
root@7de4bed89922:/#
```

ENTRYPOINT

ENTRYPOINT instruction allows you to configure a container that will run as an executable. It looks similar to CMD, because it also allows you to specify a command with parameters. The difference is ENTRYPOINT command and parameters are not ignored when Docker container runs with command line parameters. (There is a way to ignore ENTTRYPOINT, but it is unlikely that you will do it.)

ENTRYPOINT has two forms:

- `ENTRYPOINT ["executable", "param1", "param2"]` (exec form, preferred)
- `ENTRYPOINT command param1 param2` (shell form)

Be very careful when choosing ENTRYPOINT form, because forms behaviour differs significantly.

Exec form

Exec form of ENTRYPOINT allows you to set commands and parameters and then use either form of CMD to set additional parameters that are more likely to be changed. ENTRYPOINT arguments are always used, while CMD ones can be overwritten by command line arguments provided when Docker container runs. For example, the following snippet in Dockerfile

```
ENTRYPOINT ["/bin/echo", "Hello"]
CMD ["world"]
```

when container runs as `docker run -it <image>` will produce output

```
Hello world
```

but when container runs as `docker run -it <image> John` will result in

```
Hello John
```

Shell form

Shell form of ENTRYPOINT ignores any CMD or docker run command line arguments.

The 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.

.....
.....

RUN

RUN has 2 forms:

- `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 `RUN` instruction will execute any commands in a new layer on top of the current image and

commit the results. The resulting comitted image will be used for the next step in the `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 an image's history, much like source control.

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.

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

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'
```

> **Note**: 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**: The *exec* form is parsed as a JSON array, which means that you must use double-quotes (") around words not single-quotes (').
>
> **Note**: Unlike the *shell* form, the *exec* form does not invoke a command shell. This means that normal shell processing does not happen. For example, `RUN [ "echo", "$HOME" ]` will not do variable substitution on `$HOME`. If you want shell processing then either use the *shell* form or execute a shell directly, for example: `RUN [ "sh", "-c", "echo $HOME" ]`.
>
> **Note**: In the *JSON* form, it is necessary to escape backslashes. This is particularly relevant on Windows where the backslash is the path separator. The following line would otherwise be treated as *shell* form due to not being valid JSON, and fail in an unexpected way: `RUN ["c:\windows\system32\tasklist.exe"]` The correct syntax for this example is: `RUN ["c:\\windows\\system32\\tasklist.exe"]`

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`.

See the `Dockerfile` Best Practices guide for more information.

The cache for `RUN` instructions can be invalidated by `ADD` instructions. See below for details.

Known issues (RUN)

- Issue 783 is about file permissions problems that can occur when using the AUFS file system. You might notice it during an attempt to `rm` a file, for example.

For systems that have recent aufs version (i.e., `dirperm1` mount option can be set), docker will attempt to fix the issue automatically by mounting the layers with `dirperm1` option. More details on `dirperm1` option can be found at `aufs` man page

If your system doesn't have support for `dirperm1`, the issue describes a workaround.

CMD

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)

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. These defaults can include an executable, or they can omit the executable, in which case you must specify an `ENTRYPOINT` instruction as well.

> **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.
>
> **Note**: The *exec* form is parsed as a JSON array, which means that you must use double-quotes (“) around words not single-quotes (‘).
>
> **Note**: Unlike the *shell* form, the *exec* form does not invoke a command shell. This means that normal shell processing does not happen. For example, `CMD [ "echo", "$HOME" ]` will not do variable substitution on `$HOME`. If you want shell processing then either use the *shell* form or execute a shell directly, for example: `CMD [ "sh", "-c", "echo $HOME" ]`.

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`. See *ENTRYPOINT*.

If the user specifies arguments to `docker run` then they will override the default specified in `CMD`.

> **Note**: don't confuse `RUN` with `CMD`. `RUN` actually runs a command and commits the result; `CMD` does not execute anything at build time, but specifies the intended command for the image.

ENTRYPOINT

ENTRYPOINT has two forms:

- `ENTRYPOINT ["executable", "param1", "param2"]` (*exec* form, preferred)
- `ENTRYPOINT command param1 param2` (*shell* form)

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

For example, the following will start 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.

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>`.

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

Exec form ENTRYPOINT example

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
```

```
top - 08:25:00 up  7:27,  0 users,  load average: 0.00, 0.01, 0.05
Threads:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   2056668 total,  1616832 used,   439836 free,    99352 buffers
KiB Swap:  1441840 total,        0 used,  1441840 free.  1324440 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
    1 root      20   0   19744   2336   2080 R  0.0  0.1   0:00.04 top
```

To examine the result further, you can use `docker exec`:

```
$ docker exec -it test ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  2.6  0.1  19752  2352 ?        Ss+  08:24   0:00 top -b -H
root         7  0.0  0.1  15572  2164 ?        R+   08:25   0:00 ps aux
```

And you can gracefully request `top` to shut down using `docker stop test`.

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:

```
#!/bin/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:

```
#!/bin/sh
# Note: I've written this using sh so it works in the busybox container too

# USE the trap if you need to also do manual cleanup after the service is
stopped,
#     or need to start multiple services in the one container
trap "echo TRAPed signal" HUP INT QUIT TERM

# start service in background here
/usr/sbin/apachectl start
```

```
echo "[hit enter key to exit] or run 'docker stop <container>'"
read

# stop service and clean up here
echo "stopping apache"
/usr/sbin/apachectl stop

echo "exited $0"
```

If you run this image with `docker run -it --rm -p 80:80 --name test apache`, you can then examine the container's processes with `docker exec`, or `docker top`, and then ask the script to stop Apache:

```
$ docker exec -it test ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.0   4448   692 ?        Ss+  00:42   0:00 /bin/sh /run.sh
123 cmd cmd2
root        19  0.0  0.2  71304  4440 ?        Ss   00:42   0:00
/usr/sbin/apache2 -k start
www-data    20  0.2  0.2 360468  6004 ?        Sl   00:42   0:00
/usr/sbin/apache2 -k start
www-data    21  0.2  0.2 360468  6000 ?        Sl   00:42   0:00
/usr/sbin/apache2 -k start
root        81  0.0  0.1  15572  2140 ?        R+   00:44   0:00 ps aux
$ docker top test
PID                 USER                COMMAND
10035               root                {run.sh} /bin/sh /run.sh 123 cmd cmd2
10054               root                /usr/sbin/apache2 -k start
10055               33                  /usr/sbin/apache2 -k start
10056               33                  /usr/sbin/apache2 -k start
$ /usr/bin/time docker stop test
test
real    0m 0.27s
user    0m 0.03s
sys     0m 0.03s
```

> **Note:** you can over ride the `ENTRYPOINT` setting using `--entrypoint`, but this can only set the binary to *exec* (no `sh -c` will be used).
>
> **Note**: The *exec* form is parsed as a JSON array, which means that you must use double-quotes (“) around words not single-quotes (‘).
>
> **Note**: Unlike the *shell* form, the *exec* form does not invoke a command shell. This means that normal shell processing does not happen. For example, `ENTRYPOINT [ "echo", "$HOME" ]` will not do variable substitution on `$HOME`. If you want shell processing then either use the *shell* form or execute a shell directly, for example: `ENTRYPOINT [ "sh", "-c", "echo $HOME" ]`. Variables that are defined in the `Dockerfile`using `ENV`, will be substituted by the `Dockerfile` parser.

Shell form ENTRYPOINT example

You can specify a plain string for the `ENTRYPOINT` and it will execute in `/bin/sh -c`. This form will use shell processing to substitute shell environment variables, and will ignore any `CMD` or `docker run` command line arguments. To ensure that `docker stop` will signal any long running `ENTRYPOINT` executable correctly, you need to remember to start it with `exec`:

```
FROM ubuntu
ENTRYPOINT exec top -b
```

When you run this image, you'll see the single `PID 1` process:

```
$ docker run -it --rm --name test top
Mem: 1704520K used, 352148K free, 0K shrd, 0K buff, 140368121167873K cached
CPU:   5% usr   0% sys   0% nic  94% idle   0% io   0% irq   0% sirq
Load average: 0.08 0.03 0.05 2/98 6
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
    1     0 root     R     3164   0%   0% top -b
```

Which will exit cleanly on `docker stop`:

```
$ /usr/bin/time docker stop test
test
real    0m 0.20s
user    0m 0.02s
sys 0m 0.04s
```

If you forget to add `exec` to the beginning of your `ENTRYPOINT`:

```
FROM ubuntu
ENTRYPOINT top -b
CMD --ignored-param1
```

You can then run it (giving it a name for the next step):

```
$ docker run -it --name test top --ignored-param2
Mem: 1704184K used, 352484K free, 0K shrd, 0K buff, 140621524238337K cached
CPU:   9% usr   2% sys   0% nic  88% idle   0% io   0% irq   0% sirq
Load average: 0.01 0.02 0.05 2/101 7
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
    1     0 root     S     3168   0%   0% /bin/sh -c top -b cmd cmd2
    7     1 root     R     3164   0%   0% top -b
```

You can see from the output of `top` that the specified `ENTRYPOINT` is not `PID 1`.

If you then run `docker stop test`, the container will not exit cleanly - the `stop` command will be forced to send a `SIGKILL` after the timeout:

```
$ docker exec -it test ps aux
PID   USER     COMMAND
    1 root     /bin/sh -c top -b cmd cmd2
    7 root     top -b
    8 root     ps aux
$ /usr/bin/time docker stop test
test
real    0m 10.19s
user    0m 0.04s
sys 0m 0.03s
```

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.

1. Dockerfile should specify at least one of `CMD` or `ENTRYPOINT` commands.

2. `ENTRYPOINT` should be defined when using the container as an executable.
3. `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.
4. `CMD` will be overridden when running the container with alternative arguments.

The table below shows what command is executed for different `ENTRYPOINT` / `CMD` combinations:

	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_cmd p1_cmd	exec_entry p1_entry exec_cmd p1_cmd
CMD [“p1_cmd”, “p2_cmd”]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry p1_cmd p2_cmd	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 /bin/sh -c exec_cmd p1_cmd	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

.....
.....

Docker CMD vs ENTRYPOINT

on tips, docker

Inside the Dockerfile:

CMD

This is the default command to be run by the entrypoint

ENTRYPOINT

This is the program to run the given command.

Docker uses a default entrypoint of `/bin/sh -c` without a default command.
When you call `docker run [options] ubuntu bash`, bash is the command you're passing in, so your actually running: `/bin/sh -c bash`.
You could then set a command in the dockerfile with CMD and this will be used by default unless you call a specific command instead when calling docker run.

You can also specify an ENTRYPOINT in your dockerfile which can be usefull if you want to use

your container as more of an execuatable and don't need to run any other executable commands inside.

Which to use?

My advice is to use CMD in most places where you're running a command, but you may want to run a different command instead at some other stage, as if you used entrypoint you wouldn't be able to do this so easily.

But use entrypoint when you want to make the container act more like an executable, as a way of prefixing commands you pass to it with another command.