Dockerfile Reference

Docker can build images automatically by reading the instructions from a `Dockerfile`. A `Dockerfile` is a text document that contains all the commands you would normally execute manually in order to build a Docker image. By calling `docker build` from your terminal, you can have Docker build your image step by step, executing the instructions successively.

This page discusses the specifics of all the instructions you can use in your `Dockerfile`. To further help you write a clear, readable, maintainable `Dockerfile`, we've also written a `Dockerfile` Best Practices guide (/articles/dockerfile_best-practices). Lastly, you can test your Dockerfile knowledge with the Dockerfile tutorial (/userguide/level1).

Usage

To *build* (/reference/commandline/cli/#build) an image from a source repository, create a description file called `Dockerfile` at the root of your repository. This file will describe the steps to assemble the image.

Then call `docker build` with the path of your source repository as the argument (for example, `.`):

```
$ sudo docker build .
```

The path to the source repository defines where to find the *context* of the build. The build is run by the Docker daemon, not by the CLI, so 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.

> ***Warning*** *Avoid using your root directory, `/`, as the root of the source repository. The `docker build` command will use whatever directory contains the Dockerfile as the build context (including all of its subdirectories). The build context will be sent to the Docker daemon before building the image, which means if you use `/` as the source repository, the entire contents of your hard drive will get sent to the daemon (and thus to the machine running the daemon). You probably don't want that.*

In most cases, it's best to put each Dockerfile in an empty directory, and then add only the files needed for building that Dockerfile to that directory. To further speed up the build, you can exclude files and directories by adding a `.dockerignore` file to the same directory.

You can specify a repository and tag at which to save the new image if the build succeeds:

```
$ sudo docker build -t shykes/myapp .
```

The Docker daemon will run your steps one-by-one, committing the result to a new image if necessary, before finally outputting the ID of your new image. The Docker daemon will automatically clean up the context you sent.

Note that each instruction is run independently, and causes a new image to be created - so `RUN cd /tmp` will not have any effect on the next instructions.

Whenever possible, Docker will re-use the intermediate images, accelerating `docker build` significantly (indicated by `Using cache` - see the `Dockerfile` Best Practices guide (/articles/dockerfile_best-practices/#build-cache) for more information):

```
$ sudo docker build -t SvenDowideit/ambassador .
Uploading context 10.24 kB
Uploading context
Step 1 : FROM docker-ut
 ---> cbba202fe96b
Step 2 : MAINTAINER SvenDowideit@home.org.au
 ---> Using cache
 ---> 51182097be13
Step 3 : CMD env | grep _TCP= | sed 's/.*_PORT_\([0-9]*\)_TCP=tcp:\/\/\(.*\):\(.*\)/socat TCP4-LISTEN:\1,fork,reuseaddr TCP4:\2:\3 \&/'  | sh && top
 ---> Using cache
 ---> 1a5ffc17324d
Successfully built 1a5ffc17324d
```

When you're done with your build, you're ready to look into *Pushing a repository to its registry* (/userguide/dockerrepos/#contributing-to-docker-hub).

Format

Here is the format of the `Dockerfile`:

```
# Comment
INSTRUCTION arguments
```

The Instruction is not case-sensitive, however convention is for them to be UPPERCASE in order to distinguish them from arguments more easily.

Docker runs the instructions in a `Dockerfile` in order. **The first instruction must be `FROM`** in order to specify the *Base Image* (/terms/image/#base-image) from which you are building.

Docker will treat lines that *begin* with `#` as a comment. A `#` marker anywhere else in the line will be treated as an argument. This allows statements like:

```
# Comment
RUN echo 'we are running some # of cool things'
```

Here is the set of instructions you can use in a `Dockerfile` for building images.

Environment Replacement

Note: prior to 1.3, `Dockerfile` environment variables were handled similarly, in that they would be replaced as described below. However, there was no formal definition on as to which instructions handled environment replacement at the time. After 1.3 this behavior will be preserved and canonical.

Environment variables (declared with the `ENV` statement) can also be used in certain instructions as variables to be interpreted by the `Dockerfile`. Escapes are also handled for including variable-like syntax into a statement literally.

Environment variables are notated in the `Dockerfile` either with `$variable_name` or `${variable_name}`. They are treated equivalently and the brace syntax is typically used to address issues with variable names with no whitespace, like `${foo}_bar`.

Escaping is possible by adding a `\` before the variable: `\$foo` or `\${foo}`, for example, will translate to `$foo` and `${foo}` literals respectively.

Example (parsed representation is displayed after the `#`):

```
FROM busybox
ENV foo /bar
WORKDIR ${foo}   # WORKDIR /bar
ADD . $foo       # ADD . /bar
COPY \$foo /quux # COPY $foo /quux
```

The instructions that handle environment variables in the `Dockerfile` are:

- `ENV`
- `ADD`
- `COPY`
- `WORKDIR`
- `EXPOSE`
- `VOLUME`
- `USER`

`ONBUILD` instructions are **NOT** supported for environment replacement, even the instructions above.

The `.dockerignore` file

If a file named `.dockerignore` exists in the source repository, then it is interpreted as a newline-separated list of exclusion patterns. Exclusion patterns match files or directories relative to the source repository that will be excluded from the context. Globbing is done using Go's filepath.Match (http://golang.org/pkg/path/filepath#Match) rules.

The following example shows the use of the `.dockerignore` file to exclude the `.git` directory from the context. Its effect can be seen in the changed size of the uploaded context.

```
$ sudo docker build .
Uploading context 18.829 MB
Uploading context
Step 0 : FROM busybox
 ---> 769b9341d937
Step 1 : CMD echo Hello World
 ---> Using cache
 ---> 99cc1ad10469
Successfully built 99cc1ad10469
$ echo ".git" > .dockerignore
$ sudo docker build .
Uploading context  6.76 MB
Uploading context
Step 0 : FROM busybox
 ---> 769b9341d937
Step 1 : CMD echo Hello World
 ---> Using cache
 ---> 99cc1ad10469
Successfully built 99cc1ad10469
```

FROM

```
FROM <image>
```

Or

```
FROM <image>:<tag>
```

The `FROM` instruction sets the *Base Image* (/terms/image/#base-image) for subsequent instructions. As such, a valid `Dockerfile` must have `FROM` as its first instruction. The image can be any valid image – it is especially easy to start by **pulling an image** from the *Public Repositories* (/userguide/dockerrepos).

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

`FROM` can appear multiple times within a single `Dockerfile` in order to create multiple images. Simply 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, `latest` is assumed. If the used tag does not exist, an error will be returned.

MAINTAINER

```
MAINTAINER <name>
```

The `MAINTAINER` instruction allows you to set the *Author* field of the generated images.

RUN

RUN has 2 forms:

- `RUN <command>` (the command is run in a shell - `/bin/sh -c` - *shell* form)
- `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 committed 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 `/bin/sh`.

> ***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" ]`.*

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 (/articles/dockerfile_best-practices/#build-cache) for more information.

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

Known Issues (RUN)

- Issue 783 (https://github.com/docker/docker/issues/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. 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.*

EXPOSE

```
EXPOSE <port> [<port>...]
```

The `EXPOSE` instructions informs Docker that the container will listen on the specified network ports at runtime. Docker uses this information to interconnect containers using links (see the Docker User Guide (/userguide/dockerlinks)) and to determine which ports to expose to the host when using the -P flag (/reference/run/#expose-incoming-ports). **Note:** `EXPOSE` doesn't define which ports can be exposed to the host or make ports accessible from the host by default. To expose ports to the host, at runtime, use the `-p` flag (/userguide/dockerlinks) or the -P flag (/reference/run/#expose-incoming-ports).

ENV

```
ENV <key> <value>
ENV <key>=<value> ...
```

The `ENV` instruction sets the environment variable `<key>` to the value `<value>`. This value will be passed to all future `RUN` instructions. This is functionally equivalent to prefixing the command with `<key>=<value>`

The `ENV` instruction has two forms. The first form, `ENV <key> <value>`, will set a single variable to a value. The entire string after the first space will be treated as the `<value>` - including characters such as spaces and quotes.

The second form, `ENV <key>=<value> ...`, allows for multiple variables to be set at one time. Notice that the second form uses the equals sign (=) in the syntax, while the first form does not. Like command line parsing, quotes and backslashes can be used to include spaces within values.

For example:

```
ENV myName="John Doe" myDog=Rex\ The\ Dog \
    myCat=fluffy
```

and

```
ENV myName John Doe
ENV myDog Rex The Dog
ENV myCat fluffy
```

will yield the same net results in the final container, but the first form does it all in one layer.

The environment variables set using `ENV` will persist when a container is run from the resulting image. You can view the values using `docker inspect`, and change them using `docker run --env <key>=<value>`.

> ***Note**: One example where this can cause unexpected consequences, is setting `ENV DEBIAN_FRONTEND noninteractive`. Which will persist when the container is run interactively; for example: `docker run -t -i image bash`*

ADD

```
ADD <src>... <dest>
```

The `ADD` instruction copies new files, directories or remote file URLs from `<src>` and adds them to the filesystem of the container at the path `<dest>`.

Multiple `<src>` resource 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).

Each `<src>` may contain wildcards and matching will be done using Go's filepath.Match (http://golang.org/pkg/path/filepath#Match) rules. For most command line uses this should act as expected, for example:

```
ADD hom* /mydir/        # adds all files starting with "hom"
ADD hom?.txt /mydir/    # ? is replaced with any single character
```

The `<dest>` is the absolute path to which the source will be copied inside the destination container.

All new files and directories are created with a UID and GID of 0.

In the case where `<src>` is a remote file URL, the destination will have permissions of 600. If the remote file being retrieved has an HTTP `Last-Modified` header, the timestamp from that header will be used to set the `mtime` on the destination file. Then, like any other file processed during an `ADD`, `mtime` will be included in the determination of whether or not the file has changed and the cache should be updated.

> ***Note**: If you build by passing a `Dockerfile` through STDIN (`docker build - < somefile`), there is no build context, so the `Dockerfile` can only contain a URL based `ADD` instruction. You can also pass a compressed archive through STDIN: (`docker build - < archive.tar.gz`), the `Dockerfile` at the root of the archive and the rest of the archive will get used at the context of the build.*
> ***Note**: If your URL files are protected using authentication, you will need to use `RUN wget`, `RUN curl` or use another tool from within the container as the `ADD` instruction does not support authentication.*
> ***Note**: The first encountered `ADD` instruction will invalidate the cache for all following instructions from the Dockerfile if the contents of `<src>` have changed. This includes invalidating the cache for `RUN` instructions. See the `Dockerfile` Best Practices guide (/articles/dockerfile_best-practices/#build-cache) for more information.*

The copy obeys the following rules:

- The `<src>` path must be inside the *context* of the build; you cannot `ADD ../something /something`, because the first step of a `docker build` is to send the context directory (and subdirectories) to the docker daemon.
- If `<src>` is a URL and `<dest>` does not end with a trailing slash, then a file is downloaded from the URL and copied to `<dest>`.
- If `<src>` is a URL and `<dest>` does end with a trailing slash, then the filename is inferred from the URL and the file is downloaded to `<dest>/<filename>`. For instance, `ADD http://example.com/foobar /` would create the file `/foobar`. The URL must have a nontrivial path so that an appropriate filename can be discovered in this case (`http://example.com` will not work).
- If `<src>` is a directory, the entire contents of the directory are copied, including filesystem metadata.

  > ***Note**: The directory itself is not copied, just its contents.*

- If `<src>` is a *local* tar archive in a recognized compression format (identity, gzip, bzip2 or xz) then it is unpacked as a directory. Resources from *remote* URLs are **not** decompressed. When a directory is copied or unpacked, it has the same behavior as `tar -x`: the result is the union of:

  1. Whatever existed at the destination path and
  2. The contents of the source tree, with conflicts resolved in favor of "2." on a file-by-file basis.

- If `<src>` is any other kind of file, it is copied individually along with its metadata. In this case, if `<dest>` ends with a trailing slash `/`, it will be considered a directory and the contents of `<src>` will be written at `<dest>/base(<src>)`.
- If multiple `<src>` resources are specified, either directly or due to the use of a wildcard, then `<dest>` must be a directory, and it must end with a slash `/`.
- If `<dest>` does not end with a trailing slash, it will be considered a regular file and the contents of `<src>` will be written at `<dest>`.
- If `<dest>` doesn't exist, it is created along with all missing directories in its path.

COPY

```
COPY <src>... <dest>
```

The `COPY` instruction copies new files or directories from `<src>` and adds them to the filesystem of the container at the path `<dest>`.

Multiple `<src>` resource may be specified but they must be relative to the source directory that is being built (the context of the build).

Each `<src>` may contain wildcards and matching will be done using Go's filepath.Match (http://golang.org/pkg/path/filepath#Match) rules. For most command line uses this should act as expected, for example:

```
COPY hom* /mydir/        # adds all files starting with "hom"
COPY hom?.txt /mydir/    # ? is replaced with any single character
```

The `<dest>` is the absolute path to which the source will be copied inside the destination container.

All new files and directories are created with a UID and GID of 0.

> ***Note**: If you build using STDIN (`docker build - < somefile`), there is no build context, so `COPY` can't be used.*

The copy obeys the following rules:

- The `<src>` path must be inside the *context* of the build; you cannot `COPY ../something /something`, because the first step of a `docker build` is to send the context directory (and subdirectories) to the docker daemon.
- If `<src>` is a directory, the entire contents of the directory are copied, including filesystem metadata.

> ***Note**: The directory itself is not copied, just its contents.*

- If `<src>` is any other kind of file, it is copied individually along with its metadata. In this case, if `<dest>` ends with a trailing slash `/`, it will be considered a directory and the contents of `<src>` will be written at `<dest>/base(<src>)`.
- If multiple `<src>` resources are specified, either directly or due to the use of a wildcard, then `<dest>` must be a directory, and it must end with a slash `/`.
- If `<dest>` does not end with a trailing slash, it will be considered a regular file and the contents of `<src>` will be written at `<dest>`.
- If `<dest>` doesn't exist, it is created along with all missing directories in its path.

ENTRYPOINT

ENTRYPOINT has two forms:

- `ENTRYPOINT ["executable", "param1", "param2"]` (the preferred *exec* form)
- `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` (see the Dockerfile best practices (/articles/dockerfile_best-practices/#entrypoint) for more details):

```
#!/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 KILL 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
```

VOLUME

```
VOLUME ["/data"]
```

The `VOLUME` instruction will create a mount point with the specified name and mark 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`. For more information/examples and mounting instructions via the Docker client, refer to *Share Directories via Volumes* (/userguide/dockervolumes/#volume) documentation.

> ***Note**: The list is parsed as a JSON array, which means that you must use double-quotes (") around words not single-quotes (').*

USER

```
USER daemon
```

The `USER` instruction sets the user name or UID to use when running the image and for any `RUN`, `CMD` and `ENTRYPOINT` instructions that follow it in the `Dockerfile`.

WORKDIR

```
WORKDIR /path/to/workdir
```

The `WORKDIR` instruction sets the working directory for any `RUN`, `CMD` and `ENTRYPOINT` instructions that follow it in the `Dockerfile`.

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

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

The output of the final `pwd` command in this `Dockerfile` would be `/a/b/c`.

The `WORKDIR` instruction can resolve environment variables previously set using `ENV`. You can only use environment variables explicitly set in the `Dockerfile`. For example:

```
ENV DIRPATH /path
WORKDIR $DIRPATH/$DIRNAME
```

The output of the final `pwd` command in this `Dockerfile` would be `/path/$DIRNAME`

ONBUILD

```
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.

This is useful if you are building an image which will be used as a base to build other images, for example an application build environment or a daemon which may be customized with user-specific configuration.

For example, if your image is a reusable Python application builder, it will require application source code to be added in a particular directory, and it might require a build script to be 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 will be different for each application build. You could simply provide application developers with a boilerplate `Dockerfile` to copy-paste into their application, but that is inefficient, error-prone and difficult to update because it mixes with application-specific code.

The solution is to use `ONBUILD` to register advance instructions to run later, during the next build stage.

Here's how it works:

1. When it encounters an `ONBUILD` instruction, the builder adds a trigger to the metadata of the image being built. The instruction does not otherwise affect the current build.
2. At the end of the build, a list of all triggers is stored in the image manifest, under the key `OnBuild`. They can be inspected with the `docker inspect` command.
3. Later the image may be used as a base for a new build, using the `FROM` instruction. As part of processing the `FROM` instruction, the downstream builder looks for `ONBUILD` triggers, and executes them in the same order they were registered. If any of the triggers fail, the `FROM` instruction is aborted which in turn causes the build to fail. If all triggers succeed, the `FROM` instruction completes and the build continues as usual.
4. Triggers are cleared from the final image after being executed. In other words they are not inherited by "grand-children" builds.

For example you might add something like this:

```
[...]
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

> ***Warning**: Chaining `ONBUILD` instructions using `ONBUILD ONBUILD` isn't allowed.*
> ***Warning**: The `ONBUILD` instruction may not trigger `FROM` or `MAINTAINER` instructions.*

Dockerfile Examples

```
# Nginx
#
# VERSION               0.0.1

FROM      ubuntu
MAINTAINER Victor Vieux <victor@docker.com>

RUN apt-get update && apt-get install -y inotify-tools nginx apache2 openssh-server

# Firefox over VNC
#
# VERSION               0.3

FROM ubuntu

# Install vnc, xvfb in order to create a 'fake' display and firefox
RUN apt-get update && apt-get install -y x11vnc xvfb firefox
RUN mkdir ~/.vnc
# Setup a password
RUN x11vnc -storepasswd 1234 ~/.vnc/passwd
# Autostart firefox (might not be the best way, but it does the trick)
RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900
CMD    ["x11vnc", "-forever", "-usepw", "-create"]

# Multiple images example
#
# VERSION               0.1

FROM ubuntu
RUN echo foo > bar
# Will output something like ===> 907ad6c2736f

FROM ubuntu
RUN echo moo > oink
# Will output something like ===> 695d7793cbe4

# You'll now have two images, 907ad6c2736f with /bar, and 695d7793cbe4 with
# /oink.
```