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 a user could call on the command line to assemble an image. Using `docker build` users can create an automated build that executes several command-line instructions in succession.

This page describes the commands you can use in a `Dockerfile`. When you are done reading this page, refer to the `Dockerfile` Best Practices (../../../engine/articles/dockerfile_best-practices/) for a tip-oriented guide.

Usage

The `docker build` (../../../engine/reference/commandline/build/) command builds an image from a `Dockerfile` and a context. The build's context is the files at a specified location `PATH` or `URL`. The `PATH` is a directory on your local filesystem. The `URL` is a the location of a Git repository.

A context is processed recursively. So, a `PATH` includes any subdirectories and the `URL` includes the repository and its submodules. A simple build command that uses the current directory as context:

```
$ docker build .
Sending build context to Docker daemon  6.51 MB
...
```

The build is run by the Docker daemon, not by the CLI. The first thing a build process does is send the entire context (recursively) to the daemon. In most cases, it's best to start with an empty directory as context and keep your Dockerfile in that directory. Add only the files needed for building the Dockerfile.

> Warning: Do not use your root directory, `/`, as the `PATH` as it causes the build to transfer the entire contents of your hard drive to the Docker daemon.

To use a file in the build context, the `Dockerfile` refers to the file specified in an instruction, for example, a `COPY` instruction. To increase the build's performance, exclude files and directories by adding a `.dockerignore` file to the context directory. For information about how to create a `.dockerignore` file see the documentation on this page.

Traditionally, the `Dockerfile` is called `Dockerfile` and located in the root of the context. You use the `-f` flag with `docker build` to point to a Dockerfile anywhere in your file system.

```
$ docker build -f /path/to/a/Dockerfile .
```

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

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

The Docker daemon runs the instructions in the `Dockerfile` one-by-one, committing the result of each instruction 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 (cache), to accelerate the `docker build` process significantly. This is indicated by the `Using cache` message in the console output. (For more information, see the Build cache section (../../../engine/articles/dockerfile_best-practices/#build-cache)) in the `Dockerfile` best practices guide:

```
$ docker build -t svendowideit/ambassador .
Sending build context to Docker daemon 15.36 kB
Step 0 : FROM alpine:3.2
 ---> 31f630c65071
Step 1 : MAINTAINER SvenDowideit@home.org.au
 ---> Using cache
 ---> 2a1c91448f5f
Step 2 : RUN apk update &&      apk add socat &&        rm -r /var/cache/
 ---> Using cache
 ---> 21ed6e7fbb73
Step 3 : CMD env | grep _TCP= | sed 's/.*_PORT_\([0-9]*\)_TCP=tcp:\/\/\(.*\):\(.*\)/socat -t 100000000 TCP4-LISTEN:\1,fork,reusea
 ---> Using cache
 ---> 7ea8aef582cc
Successfully built 7ea8aef582cc
```

When you're done with your build, you're ready to look into Pushing a repository to its registry (../../../engine/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 (../../../engine/reference/glossary/#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

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`.

The `${variable_name}` syntax also supports a few of the standard `bash` modifiers as specified below:

- `${variable:-word}` indicates that if `variable` is set then the result will be that value. If `variable` is not set then `word` will be the result.
- `${variable:+word}` indicates that if `variable` is set then `word` will be the result, otherwise the result is the empty string.

In all cases, `word` can be any string, including additional environment variables.

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
```

Environment variables are supported by the following list of instructions in the `Dockerfile`:

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

as well as:

- `ONBUILD` (when combined with one of the supported instructions above)

> Note: prior to 1.4, `ONBUILD` instructions did NOT support environment variable, even when combined with any of the instructions listed above.

Environment variable substitution will use the same value for each variable throughout the entire command. In other words, in this example:

```
ENV abc=hello
ENV abc=bye def=$abc
ENV ghi=$abc
```

will result in `def` having a value of `hello`, not `bye`. However, `ghi` will have a value of `bye` because it is not part of the same command that set `abc` to `bye`.

.dockerignore file

Before the docker CLI sends the context to the docker daemon, it looks for a file named `.dockerignore` in the root directory of the context. If this file exists, the CLI modifies the context to exclude files and directories that match patterns in it. This helps to avoid unnecessarily sending large or sensitive files and directories to the daemon and potentially adding them to images using `ADD` or `COPY`.

The CLI interprets the `.dockerignore` file as a newline-separated list of patterns similar to the file globs of Unix shells. For the purposes of matching, the root of the context is considered to be both the working and the root directory. For example, the patterns `/foo/bar` and `foo/bar` both exclude a file or directory named `bar` in the `foo` subdirectory of `PATH` or in the root of the git repository located at `URL`. Neither excludes anything else.

Here is an example `.dockerignore` file:

```
    */temp*
   */*/temp*
   temp?
```

This file causes the following build behavior:

Rule	Behavior
`*/temp*`	Exclude files and directories whose names start with `temp` in any immediate subdirectory of the root. For example, the plain file `/somedir/temporary.txt` is excluded, as is the directory `/somedir/temp`.
`*/*/temp*`	Exclude files and directories starting with `temp` from any subdirectory that is two levels below the root. For example, `/somedir/subdir/temporary.txt` is excluded.
`temp?`	Exclude files and directories in the root directory whose names are a one-character extension of `temp`. For example, `/tempa` and `/tempb` are excluded.

Matching is done using Go's filepath.Match (http://golang.org/pkg/path/filepath#Match) rules. A preprocessing step removes leading and trailing whitespace and eliminates `.` and `..` elements using Go's filepath.Clean (http://golang.org/pkg/path/filepath/#Clean). Lines that are blank after preprocessing are ignored.

Lines starting with `!` (exclamation mark) can be used to make exceptions to exclusions. The following is an example `.dockerignore` file that uses this mechanism:

```
    *.md
   !README.md
```

All markdown files except `README.md` are excluded from the context.

The placement of `!` exception rules influences the behavior: the last line of the `.dockerignore` that matches a particular file determines whether it is included or excluded. Consider the following example:

```
    *.md
   !README*.md
   README-secret.md
```

No markdown files are included in the context except README files other than `README-secret.md`.

Now consider this example:

```
    *.md
   README-secret.md
   !README*.md
```

All of the README files are included. The middle line has no effect because `!README*.md` matches `README-secret.md` and comes last.

You can even use the `.dockerignore` file to exclude the `Dockerfile` and `.dockerignore` files. These files are still sent to the daemon because it needs them to do its job. But the `ADD` and `COPY` commands do not copy them to the the image.

Finally, you may want to specify which files to include in the context, rather than which to exclude. To achieve this, specify `*` as the first pattern, followed by one or more `!` exception patterns.

Note: For historical reasons, the pattern `.` is ignored.

FROM

```
FROM <image>
```

Or

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

Or

```
FROM <image>@<digest>
```

The `FROM` instruction sets the Base Image (../../../engine/reference/glossary/#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 (../../../engine/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.
- The `tag` or `digest` values are optional. If you omit either of them, the builder assumes a `latest` by default. The builder returns an error if it cannot match the `tag` value.

MAINTAINER

```
MAINTAINER <name>
```

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

RUN

RUN has 2 forms:

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

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" ]`.

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 (../../../engine/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.

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 (http://aufs.sourceforge.net/aufs3/man.html)

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.

LABEL

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

The `LABEL` instruction adds metadata to an image. A `LABEL` is a key-value pair. To include spaces within a `LABEL` value, use quotes and backslashes as you would in command-line parsing. A few usage examples:

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

An image can have more than one label. To specify multiple labels, Docker recommends combining labels into a single `LABEL` instruction where possible. Each `LABEL` instruction produces a new layer which can result in an inefficient image if you use many labels. This example results in a single image layer.

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

The above can also be written as:

```
LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"
```

Labels are additive including `LABEL`s in `FROM` images. If Docker encounters a label/key that already exists, the new value overrides any previous labels with identical keys.

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

```
"Labels": {
    "com.example.vendor": "ACME Incorporated"
    "com.example.label-with-value": "foo",
    "version": "1.0",
    "description": "This text illustrates that label-values can span multiple lines.",
    "multi.label1": "value1",
    "multi.label2": "value2",
    "other": "value3"
},
```

EXPOSE

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

The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime. `EXPOSE` does not make the ports of the container accessible to the host. To do that, you must use either the `-p` flag to publish a range of ports or the `-P` flag to publish all of the exposed ports. You can expose one port number and publish it externally under another number.

To set up port redirection on the host system, see using the -P flag (../../../engine/reference/run/#expose-incoming-ports). The Docker network feature supports creating networks without the need to expose ports within the network, for detailed information see the overview of this feature (../../../engine/userguide/networking/)).

ENV

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

The `ENV` instruction sets the environment variable `<key>` to the value `<value>`. This value will be in the environment of all "descendent" `Dockerfile` commands and can be replaced inline in many as well.

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 is preferred because it produces a single cache 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: Environment persistence can cause unexpected side effects. For example, setting `ENV DEBIAN_FRONTEND noninteractive` may confuse apt-get users on a Debian-based image. To set a value for a single command, use `RUN <key>=<value> <command>`.

ADD

ADD has two forms:

- `ADD <src>... <dest>`
- `ADD ["<src>",... "<dest>"]` (this form is required for paths containing whitespace)

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 example:

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

The `<dest>` is an absolute path, or a path relative to `WORKDIR`, into which the source will be copied inside the destination container.

```
ADD test relativeDir/          # adds "test" to `WORKDIR`/relativeDir/
ADD test /absoluteDir          # adds "test" to /absoluteDir
```

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. However, like any other file processed during an `ADD`, `mtime` will not 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 (../../../engine/articles/dockerfile_best-practices/#build-cache) for more information.

`ADD` 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 has two forms:

- `COPY <src>... <dest>`
- `COPY ["<src>",... "<dest>"]` (this form is required for paths containing whitespace)

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 example:

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

The `<dest>` is an absolute path, or a path relative to `WORKDIR`, into which the source will be copied inside the destination container.

```
COPY test relativeDir/   # adds "test" to `WORKDIR`/relativeDir/
COPY test /absoluteDir   # adds "test" to /absoluteDir
```

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.

`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"]` (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 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 creates a mount point with the specified name and marks 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 (../../../engine/userguide/dockervolumes/#mount-a-host-directory-as-a-data-volume) documentation.

The `docker run` command initializes the newly created volume with any data that exists at the specified location within the base image. For example, consider the following Dockerfile snippet:

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

This Dockerfile results in an image that causes `docker run`, to create a new mount point at `/myvol` and copy the `greeting` file into the newly created volume.

> Note: If any build steps change the data within the volume after it has been declared, those changes will be discarded.
>
> 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`, `ENTRYPOINT`, `COPY` and `ADD` 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
RUN pwd
```

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

ARG

```
ARG <name>[=<default value>]
```

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 an error.

```
One or more build-args were not consumed, failing build.
```

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.

> Note: It is not recommended to use build-time variables for passing secrets like github keys, user credentials etc.

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.

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, simply pass them on the command line using the `--build-arg <varname>=<value>` flag.

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.

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 valid unsigned number that matches a position in the kernel’s syscall table, for instance 9, or a signal name in the format SIGNAME, for instance SIGKILL.

Dockerfile examples

Below you can see some examples of Dockerfile syntax. If you’re interested in something more realistic, take a look at the list of Dockerization examples (../../../engine/examples/).

```
# Nginx
#
# VERSION               0.0.1

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

LABEL Description="This image is used to start the foobar executable" Vendor="ACME Products" Version="1.0"
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.
```

Best practices for writing Dockerfiles

Overview

Docker can build images automatically by reading the instructions from a `Dockerfile`, a text file that contains all the commands, in order, needed to build a given image. `Dockerfile`s adhere to a specific format and use a specific set of instructions. You can learn the basics on the Dockerfile Reference (../../../engine/reference/builder/) page. If you're new to writing `Dockerfile`s, you should start there.

This document covers the best practices and methods recommended by Docker, Inc. and the Docker community for creating easy-to-use, effective `Dockerfile`s. We strongly suggest you follow these recommendations (in fact, if you're creating an Official Image, you must adhere to these practices).

You can see many of these practices and recommendations in action in the buildpack-deps `Dockerfile` (https://github.com/docker-library/buildpack-deps/blob/master/jessie/Dockerfile).

> Note: for more detailed explanations of any of the Dockerfile commands mentioned here, visit the Dockerfile Reference (../../../engine/reference/builder/) page.

General guidelines and recommendations

Containers should be ephemeral

The container produced by the image your `Dockerfile` defines should be as ephemeral as possible. By "ephemeral," we mean that it can be stopped and destroyed and a new one built and put in place with an absolute minimum of set-up and configuration.

Use a .dockerignore file

In most cases, it's best to put each Dockerfile in an empty directory. Then, add to that directory only the files needed for building the Dockerfile. To increase the build's performance, you can exclude files and directories by adding a `.dockerignore` file to that directory as well. This file supports exclusion patterns similar to `.gitignore` files. For information on creating one, see the .dockerignore file (../../../engine/reference/builder/#dockerignore-file).

Avoid installing unnecessary packages

In order to reduce complexity, dependencies, file sizes, and build times, you should avoid installing extra or unnecessary packages just because they might be "nice to have." For example, you don't need to include a text editor in a database image.

Run only one process per container

In almost all cases, you should only run a single process in a single container. Decoupling applications into multiple containers makes it much easier to scale horizontally and reuse containers. If that service depends on another service, make use of container linking (../../../engine/userguide/networking/default_network/dockerlinks/).

Minimize the number of layers

You need to find the balance between readability (and thus long-term maintainability) of the `Dockerfile` and minimizing the number of layers it uses. Be strategic and cautious about the number of layers you use.

Sort multi-line arguments

Whenever possible, ease later changes by sorting multi-line arguments alphanumerically. This will help you avoid duplication of packages and make the list much easier to update. This also makes PRs a lot easier to read and review. Adding a space before a backslash (`\`) helps as well.

Here's an example from the `buildpack-deps` image (https://github.com/docker-library/buildpack-deps):

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

Build cache

During the process of building an image Docker will step through the instructions in your `Dockerfile` executing each in the order specified. As each instruction is examined Docker will look for an existing image in its cache that it can reuse, rather than creating a new (duplicate) image. If you do not want to use the cache at all you can use the `--no-cache=true` option on the `docker build` command.

However, if you do let Docker use its cache then it is very important to understand when it will, and will not, find a matching image. The basic rules that Docker will follow are outlined below:

- Starting with a base image that is already in the cache, the next instruction is compared against all child images derived from that base image to see if one of them was built using the exact same instruction. If not, the cache is invalidated.
- In most cases simply comparing the instruction in the `Dockerfile` with one of the child images is sufficient. However, certain instructions require a little more examination and explanation.
- For the `ADD` and `COPY` instructions, the contents of the file(s) in the image are examined and a checksum is calculated for each file. The last-modified and last-accessed times of the file(s) are not considered in these checksums. During the cache lookup, the checksum is compared against the checksum in the existing images. If anything has changed in the file(s), such as the contents and metadata, then the cache is invalidated.
- Aside from the `ADD` and `COPY` commands, cache checking will not look at the files in the container to determine a cache match. For example, when processing a `RUN apt-get -y update` command the files updated in the container will not be examined to determine if a cache hit exists. In that case just the command string itself will be used to find a match.

Once the cache is invalidated, all subsequent `Dockerfile` commands will generate new images and the cache will not be used.

The Dockerfile instructions

Below you'll find recommendations for the best way to write the various instructions available for use in a `Dockerfile`.

FROM

Dockerfile reference for the FROM instruction (../../../engine/reference/builder/#from)

Whenever possible, use current Official Repositories as the basis for your image. We recommend the Debian image (https://registry.hub.docker.com/_/debian/) since it's very tightly controlled and kept extremely minimal (currently under 100 mb), while still being a full distribution.

RUN

Dockerfile reference for the RUN instruction (../../../engine/reference/builder/#run)

As always, to make your `Dockerfile` more readable, understandable, and maintainable, split long or complex `RUN` statements on multiple lines separated with backslashes.

apt-get

Probably the most common use-case for `RUN` is an application of `apt-get`. The `RUN apt-get` command, because it installs packages, has several gotchas to look out for.

You should avoid `RUN apt-get upgrade` or `dist-upgrade`, as many of the "essential" packages from the base images won't upgrade inside an unprivileged container. If a package contained in the base image is out-of-date, you should contact its maintainers. If you know there's a particular package, `foo`, that needs to be updated, use `apt-get install -y foo` to update automatically.

Always combine `RUN apt-get update` with `apt-get install` in the same `RUN` statement, for example:

```
    RUN apt-get update && apt-get install -y \
        package-bar \
        package-baz \
        package-foo
```

Using `apt-get update` alone in a `RUN` statement causes caching issues and subsequent `apt-get install` instructions fail. For example, say you have a Dockerfile:

```
    FROM ubuntu:14.04
    RUN apt-get update
    RUN apt-get install -y curl
```

After building the image, all layers are in the Docker cache. Suppose you later modify `apt-get install` by adding extra package:

```
    FROM ubuntu:14.04
    RUN apt-get update
    RUN apt-get install -y curl nginx
```

Docker sees the initial and modified instructions as identical and reuses the cache from previous steps. As a result the `apt-get update` is NOT executed because the build uses the cached version. Because the `apt-get update` is not run, your build can potentially get an outdated version of the `curl` and `nginx` packages.

Using `RUN apt-get update && apt-get install -y` ensures your Dockerfile installs the latest package versions with no further coding or manual intervention. This technique is known as "cache busting". You can also achieve cache-busting by specifying a package version. This is known as version pinning, for example:

```
    RUN apt-get update && apt-get install -y \
        package-bar \
        package-baz \
        package-foo=1.3.*
```

Version pinning forces the build to retrieve a particular version regardless of what's in the cache. This technique can also reduce failures due to unanticipated changes in required packages.

Below is a well-formed `RUN` instruction that demonstrates all the `apt-get` recommendations.

```
RUN apt-get update && apt-get install -y \
    aufs-tools \
    automake \
    build-essential \
    curl \
    dpkg-sig \
    libcap-dev \
    libsqlite3-dev \
    lxc=1.0* \
    mercurial \
    reprepro \
    ruby1.9.1 \
    ruby1.9.1-dev \
    s3cmd=1.1.* \
&& apt-get clean \
&& rm -rf /var/lib/apt/lists/*
```

The `s3cmd` instructions specifies a version `1.1.0*`. If the image previously used an older version, specifying the new one causes a cache bust of `apt-get update` and ensure the installation of the new version. Listing packages on each line can also prevent mistakes in package duplication.

In addition, cleaning up the apt cache and removing `/var/lib/apt/lists` helps keep the image size down. Since the `RUN` statement starts with `apt-get update`, the package cache will always be refreshed prior to `apt-get install`.

CMD

Dockerfile reference for the CMD instruction (../../../engine/reference/builder/#cmd)

The `CMD` instruction should be used to run the software contained by your image, along with any arguments. `CMD` should almost always be used in the form of `CMD ["executable", "param1", "param2"…]`. Thus, if the image is for a service (Apache, Rails, etc.), you would run something like `CMD ["apache2","-DFOREGROUND"]`. Indeed, this form of the instruction is recommended for any service-based image.

In most other cases, `CMD` should be given an interactive shell (bash, python, perl, etc), for example, `CMD ["perl", "-de0"]`, `CMD ["python"]`, or `CMD ["php", "-a"]`. Using this form means that when you execute something like `docker run -it python`, you'll get dropped into a usable shell, ready to go. `CMD` should rarely be used in the manner of `CMD ["param", "param"]` in conjunction with

`ENTRYPOINT` (../../../engine/reference/builder/#entrypoint), unless you and your expected users are already quite familiar with how `ENTRYPOINT` works.

EXPOSE

Dockerfile reference for the EXPOSE instruction (../../../engine/reference/builder/#expose)

The `EXPOSE` instruction indicates the ports on which a container will listen for connections. Consequently, you should use the common, traditional port for your application. For example, an image containing the Apache web server would use `EXPOSE 80`, while an image containing MongoDB would use `EXPOSE 27017` and so on.

For external access, your users can execute `docker run` with a flag indicating how to map the specified port to the port of their choice. For container linking, Docker provides environment variables for the path from the recipient container back to the source (ie, `MYSQL_PORT_3306_TCP`).

ENV

Dockerfile reference for the ENV instruction (../../../engine/reference/builder/#env)

In order to make new software easier to run, you can use `ENV` to update the `PATH` environment variable for the software your container installs. For example, `ENV PATH /usr/local/nginx/bin:$PATH` will ensure that `CMD [“nginx”]` just works.

The `ENV` instruction is also useful for providing required environment variables specific to services you wish to containerize, such as Postgres's `PGDATA`.

Lastly, `ENV` can also be used to set commonly used version numbers so that version bumps are easier to maintain, as seen in the following example:

```
ENV PG_MAJOR 9.3
ENV PG_VERSION 9.3.4
RUN curl -SL http://example.com/postgres-$PG_VERSION.tar.xz | tar -xJC /usr/src/postgress && …
ENV PATH /usr/local/postgres-$PG_MAJOR/bin:$PATH
```

Similar to having constant variables in a program (as opposed to hard-coding values), this approach lets you change a single `ENV` instruction to auto-magically bump the version of the software in your container.

ADD or COPY

Dockerfile reference for the ADD instruction (../../../engine/reference/builder/#add)
Dockerfile reference for the COPY instruction (../../../engine/reference/builder/#copy)

Although `ADD` and `COPY` are functionally similar, generally speaking, `COPY` is preferred. That's because it's more transparent than `ADD`. `COPY` only supports the basic copying of local files into the container, while `ADD` has some features (like local-only tar extraction and remote URL support) that are not immediately obvious. Consequently, the best use for `ADD` is local tar file auto-extraction into the image, as in `ADD rootfs.tar.xz /`.

If you have multiple `Dockerfile` steps that use different files from your context, `COPY` them individually, rather than all at once. This will ensure that each step's build cache is only invalidated (forcing the step to be re-run) if the specifically required files change.

For example:

```
COPY requirements.txt /tmp/
RUN pip install /tmp/requirements.txt
COPY . /tmp/
```

Results in fewer cache invalidations for the `RUN` step, than if you put the `COPY . /tmp/` before it.

Because image size matters, using `ADD` to fetch packages from remote URLs is strongly discouraged; you should use `curl` or `wget` instead. That way you can delete the files you no longer need after they've been extracted and you won't have to add another layer in your image. For example, you should avoid doing things like:

```
ADD http://example.com/big.tar.xz /usr/src/things/
RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things
RUN make -C /usr/src/things all
```

And instead, do something like:

```
RUN mkdir -p /usr/src/things \
    && curl -SL http://example.com/big.tar.xz \
    | tar -xJC /usr/src/things \
    && make -C /usr/src/things all
```

For other items (files, directories) that do not require `ADD`'s tar auto-extraction capability, you should always use `COPY`.

ENTRYPOINT

Dockerfile reference for the ENTRYPOINT instruction (../../../engine/reference/builder/#entrypoint)

The best use for `ENTRYPOINT` is to set the image's main command, allowing that image to be run as though it was that command (and then use `CMD` as the default flags).

Let's start with an example of an image for the command line tool `s3cmd`:

```
ENTRYPOINT ["s3cmd"]
CMD ["--help"]
```

Now the image can be run like this to show the command's help:

```
$ docker run s3cmd
```

Or using the right parameters to execute a command:

```
$ docker run s3cmd ls s3://mybucket
```

This is useful because the image name can double as a reference to the binary as shown in the command above.

The `ENTRYPOINT` instruction can also be used in combination with a helper script, allowing it to function in a similar way to the command above, even when starting the tool may require more than one step.

For example, the Postgres Official Image (https://registry.hub.docker.com/_/postgres/) uses the following script as its `ENTRYPOINT`:

```
#!/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 "$@"
```

> Note: This script uses the `exec` Bash command (http://wiki.bash-hackers.org/commands/builtin/exec) so that the final running application becomes the container's PID 1. This allows the application to receive any Unix signals sent to the container. See the `ENTRYPOINT` (../../../engine/reference/builder/#entrypoint) help for more details.

The helper script is copied into the container and run via `ENTRYPOINT` on container start:

```
COPY ./docker-entrypoint.sh /
ENTRYPOINT ["/docker-entrypoint.sh"]
```

This script allows the user to interact with Postgres in several ways.

It can simply start Postgres:

```
$ docker run postgres
```

Or, it can be used to run Postgres and pass parameters to the server:

```
$ docker run postgres postgres --help
```

Lastly, it could also be used to start a totally different tool, such as Bash:

```
$ docker run --rm -it postgres bash
```

VOLUME

Dockerfile reference for the VOLUME instruction (../../../engine/reference/builder/#volume)

The `VOLUME` instruction should be used to expose any database storage area, configuration storage, or files/folders created by your docker container. You are strongly encouraged to use `VOLUME` for any mutable and/or user-serviceable parts of your image.

USER

Dockerfile reference for the USER instruction (../../../engine/reference/builder/#user)

If a service can run without privileges, use `USER` to change to a non-root user. Start by creating the user and group in the `Dockerfile` with something like `RUN groupadd -r postgres && useradd -r -g postgres postgres`.

> Note: Users and groups in an image get a non-deterministic UID/GID in that the "next" UID/GID gets assigned regardless of image rebuilds. So, if it's critical, you should assign an explicit UID/GID.

You should avoid installing or using `sudo` since it has unpredictable TTY and signal-forwarding behavior that can cause more problems than it solves. If you absolutely need functionality similar to `sudo` (e.g., initializing the daemon as root but running it as non-root), you may be able to use "gosu" (https://github.com/tianon/gosu).

Lastly, to reduce layers and complexity, avoid switching `USER` back and forth frequently.

WORKDIR

Dockerfile reference for the WORKDIR instruction (../../../engine/reference/builder/#workdir)

For clarity and reliability, you should always use absolute paths for your `WORKDIR`. Also, you should use `WORKDIR` instead of proliferating instructions like `RUN cd … && do-something`, which are hard to read, troubleshoot, and maintain.

ONBUILD

Dockerfile reference for the ONBUILD instruction (../../../engine/reference/builder/#onbuild)

An `ONBUILD` command executes after the current `Dockerfile` build completes. `ONBUILD` executes in any child image derived `FROM` the current image. Think of the `ONBUILD` command as an instruction the parent `Dockerfile` gives to the child `Dockerfile`.

A Docker build executes `ONBUILD` commands before any command in a child `Dockerfile`.

`ONBUILD` is useful for images that are going to be built `FROM` a given image. For example, you would use `ONBUILD` for a language stack image that builds arbitrary user software written in that language within the `Dockerfile`, as you can see in Ruby's `ONBUILD` variants (https://github.com/docker-library/ruby/blob/master/2.1/onbuild/Dockerfile).

Images built from `ONBUILD` should get a separate tag, for example: `ruby:1.9-onbuild` or `ruby:2.0-onbuild`.

Be careful when putting `ADD` or `COPY` in `ONBUILD`. The "onbuild" image will fail catastrophically if the new build's context is missing the resource being added. Adding a separate tag, as recommended above, will help mitigate this by allowing the `Dockerfile` author to make a choice.

Examples for Official Repositories

These Official Repositories have exemplary `Dockerfile`s:

- Go (https://registry.hub.docker.com/_/golang/)
- Perl (https://registry.hub.docker.com/_/perl/)
- Hy (https://registry.hub.docker.com/_/hylang/)
- Rails (https://registry.hub.docker.com/_/rails)

Additional resources:

- Dockerfile Reference (../../../engine/reference/builder/)
- More about Base Images (../../../engine/articles/baseimages/)
- More about Automated Builds (https://docs.docker.com/docker-hub/builds/)

- Guidelines for Creating Official Repositories (https://docs.docker.com/docker-hub/official_repos/)