Hints, tips and guidelines for writing clean, reliable Dockerfiles

This document covers recommended best practices and methods for building efficient images.

Docker builds images automatically by reading the instructions from a `Dockerfile` -- a text file that contains all commands, in order, needed to build a given image. A `Dockerfile` adheres to a specific format and set of instructions which you can find at Dockerfile reference.

A Docker image consists of read-only layers each of which represents a Dockerfile instruction. The layers are stacked and each one is a delta of the changes from the previous layer. Consider this `Dockerfile`:

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Each instruction creates one layer:

- `FROM` creates a layer from the `ubuntu:18.04` Docker image.
- `COPY` adds files from your Docker client's current directory.
- `RUN` builds your application with `make`.
- `CMD` specifies what command to run within the container.

When you run an image and generate a container, you add a new *writable layer* (the "container layer") on top of the underlying layers. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.

For more on image layers (and how Docker builds and stores images), see About storage drivers.

General guidelines and recommendations

Create ephemeral containers

The image defined by your `Dockerfile` should generate containers that are as ephemeral as possible. By "ephemeral", we mean that the container can be stopped and destroyed, then rebuilt and replaced with an absolute minimum set up and configuration.

Refer to Processes under *The Twelve-factor App* methodology to get a feel for the motivations of running containers in such a stateless fashion.

Understand build context

When you issue a `docker build` command, the current working directory is called the *build context*. By default, the Dockerfile is assumed to be located here, but you can specify a different location with the file flag (`-f`). Regardless of where the `Dockerfile` actually lives, all recursive contents of files and directories in the current directory are sent to the Docker daemon as the build context.

> Build context example
>
> Create a directory for the build context and `cd` into it. Write "hello" into a text file named `hello` and create a Dockerfile that runs `cat` on it. Build the image from within the build context (`.`):
>
> ```
> mkdir myproject && cd myproject
> echo "hello" > hello
> echo -e "FROM busybox\nCOPY /hello /\nRUN cat /hello" > Dockerfile
> docker build -t helloapp:v1 .
> ```
>
> Move `Dockerfile` and `hello` into separate directories and build a second version of the image (without relying on cache from the last build). Use `-f` to point to the Dockerfile and specify the directory of the build context:
>
> ```
> mkdir -p dockerfiles context
> mv Dockerfile dockerfiles && mv hello context
> docker build --no-cache -t helloapp:v2 -f dockerfiles/Dockerfile context
> ```

Inadvertently including files that are not necessary for building an image results in a larger build context and larger image size. This can increase the time to build the image, time to pull and push it, and the container runtime size. To see how big your build context is, look for a message like this when building your `Dockerfile`:

```
Sending build context to Docker daemon  187.8MB
```

Pipe Dockerfile through `stdin`

Docker has the ability to build images by piping `Dockerfile` through `stdin` with a *local or remote build context*. Piping a `Dockerfile` through `stdin` can be useful to perform one-off builds without writing a Dockerfile to disk, or in situations where the `Dockerfile` is generated, and should not persist afterwards.

> The examples in this section use here documents for convenience, but any method to provide the `Dockerfile` on `stdin` can be used.
>
> For example, the following commands are equivalent:
>
> ```
> echo -e 'FROM busybox\nRUN echo "hello world"' | docker build -
> ```
>
> ```
> docker build -<<EOF
> FROM busybox
> RUN echo "hello world"
> EOF
> ```

> You can substitute the examples with your preferred approach, or the approach that best fits your use-case.

Build an image using a Dockerfile from stdin, without sending build context

Use this syntax to build an image using a `Dockerfile` from `stdin`, without sending additional files as build context. The hyphen (`-`) takes the position of the `PATH`, and instructs Docker to read the build context (which only contains a `Dockerfile`) from `stdin` instead of a directory:

```
docker build [OPTIONS] -
```

The following example builds an image using a `Dockerfile` that is passed through `stdin`. No files are sent as build context to the daemon.

```
docker build -t myimage:latest -<<EOF
FROM busybox
RUN echo "hello world"
EOF
```

Omitting the build context can be useful in situations where your `Dockerfile` does not require files to be copied into the image, and improves the build-speed, as no files are sent to the daemon.

If you want to improve the build-speed by excluding *some* files from the build- context, refer to exclude with .dockerignore.

> **Note**: Attempting to build a Dockerfile that uses `COPY` or `ADD` will fail if this syntax is used. The following example illustrates this:

```
# create a directory to work in
mkdir example
cd example

# create an example file
touch somefile.txt

docker build -t myimage:latest -<<EOF
FROM busybox
COPY somefile.txt .
RUN cat /somefile.txt
EOF

# observe that the build fails
...
Step 2/3 : COPY somefile.txt .
COPY failed: stat /var/lib/docker/tmp/docker-builder249218248/somefile.txt:
no such file or directory
```

Build from a local build context, using a Dockerfile from stdin

Use this syntax to build an image using files on your local filesystem, but using a `Dockerfile` from `stdin`. The syntax uses the `-f` (or `--file`) option to specify the `Dockerfile` to use, using a hyphen (`-`) as filename to instruct Docker to read the `Dockerfile` from `stdin`:

```
docker build [OPTIONS] -f- PATH
```

The example below uses the current directory (`.`) as the build context, and builds an image using a `Dockerfile` that is passed through `stdin` using a here document.

```
# create a directory to work in
mkdir example
cd example

# create an example file
touch somefile.txt

# build an image using the current directory as context, and a Dockerfile passed
through stdin
docker build -t myimage:latest -f- . <<EOF
FROM busybox
COPY somefile.txt .
RUN cat /somefile.txt
EOF
```

Build from a remote build context, using a Dockerfile from stdin

Use this syntax to build an image using files from a remote `git` repository, using a `Dockerfile` from `stdin`. The syntax uses the `-f` (or `--file`) option to specify the `Dockerfile` to use, using a hyphen (`-`) as filename to instruct Docker to read the `Dockerfile` from `stdin`:

```
docker build [OPTIONS] -f- PATH
```

This syntax can be useful in situations where you want to build an image from a repository that does not contain a `Dockerfile`, or if you want to build with a custom `Dockerfile`, without maintaining your own fork of the repository.

The example below builds an image using a `Dockerfile` from `stdin`, and adds the `hello.c` file from the "hello-world" Git repository on GitHub.

```
docker build -t myimage:latest -f- https://github.com/docker-library/hello-
world.git <<EOF
```

```
FROM busybox
COPY hello.c .
EOF
```

> **Under the hood**
>
> When building an image using a remote Git repository as build context, Docker performs a `git clone` of the repository on the local machine, and sends those files as build context to the daemon. This feature requires `git` to be installed on the host where you run the `docker build` command.

Exclude with .dockerignore

To exclude files not relevant to the build (without restructuring your source repository) use a `.dockerignore` file. This file supports exclusion patterns similar to `.gitignore` files. For information on creating one, see the .dockerignore file.

Use multi-stage builds

Multi-stage builds allow you to drastically reduce the size of your final image, without struggling to reduce the number of intermediate layers and files.

Because an image is built during the final stage of the build process, you can minimize image layers by leveraging build cache.

For example, if your build contains several layers, you can order them from the less frequently changed (to ensure the build cache is reusable) to the more frequently changed:

- Install tools you need to build your application
- Install or update library dependencies
- Generate your application

A Dockerfile for a Go application could look like:

```
FROM golang:1.11-alpine AS build

# Install tools required for project
# Run `docker build --no-cache .` to update dependencies
RUN apk add --no-cache git
RUN go get github.com/golang/dep/cmd/dep

# List project dependencies with Gopkg.toml and Gopkg.lock
# These layers are only re-built when Gopkg files are updated
COPY Gopkg.lock Gopkg.toml /go/src/project/
WORKDIR /go/src/project/
# Install library dependencies
RUN dep ensure -vendor-only

# Copy the entire project and build it
# This layer is rebuilt when a file changes in the project directory
```

```
COPY . /go/src/project/
RUN go build -o /bin/project

# This results in a single layer image
FROM scratch
COPY --from=build /bin/project /bin/project
ENTRYPOINT ["/bin/project"]
CMD ["--help"]
```

Don't install unnecessary packages

To reduce complexity, dependencies, file sizes, and build times, 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.

Decouple applications

Each container should have only one concern. Decoupling applications into multiple containers makes it easier to scale horizontally and reuse containers. For instance, a web application stack might consist of three separate containers, each with its own unique image, to manage the web application, database, and an in-memory cache in a decoupled manner.

Limiting each container to one process is a good rule of thumb, but it is not a hard and fast rule. For example, not only can containers be spawned with an init process, some programs might spawn additional processes of their own accord. For instance, Celery can spawn multiple worker processes, and Apache can create one process per request.

Use your best judgment to keep containers as clean and modular as possible. If containers depend on each other, you can use Docker container networks to ensure that these containers can communicate.

Minimize the number of layers

In older versions of Docker, it was important that you minimized the number of layers in your images to ensure they were performant. The following features were added to reduce this limitation:

- Only the instructions `RUN`, `COPY`, `ADD` create layers. Other instructions create temporary intermediate images, and do not increase the size of the build.
- Where possible, use multi-stage builds, and only copy the artifacts you need into the final image. This allows you to include tools and debug information in your intermediate build stages without increasing the size of the final image.

Sort multi-line arguments

Whenever possible, ease later changes by sorting multi-line arguments alphanumerically. This helps to 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:

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

Leverage build cache

When building an image, Docker steps through the instructions in your `Dockerfile`, executing each in the order specified. As each instruction is examined, Docker looks 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, it is important to understand when it can, and cannot, find a matching image. The basic rules that Docker follows are outlined below:

- Starting with a parent 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 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 does 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 are not examined to determine if a cache hit exists. In that case just the command string itself is used to find a match.

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

Dockerfile instructions

These recommendations are designed to help you create an efficient and maintainable `Dockerfile`.

FROM

Dockerfile reference for the FROM instruction

Whenever possible, use current official images as the basis for your images. We recommend the Alpine image as it is tightly controlled and small in size (currently under 5 MB), while still being a full Linux distribution.

LABEL

Understanding object labels

You can add labels to your image to help organize images by project, record licensing information, to aid in automation, or for other reasons. For each label, add a line beginning with `LABEL` and with one or more key-value pairs. The following examples show the different acceptable formats. Explanatory comments are included inline.

> Strings with spaces must be quoted **or** the spaces must be escaped. Inner quote characters (`"`), must also be escaped.

```
# Set one or more individual labels
LABEL com.example.version="0.0.1-beta"
LABEL vendor1="ACME Incorporated"
LABEL vendor2=ZENITH\ Incorporated
LABEL com.example.release-date="2015-02-12"
LABEL com.example.version.is-production=""
```

An image can have more than one label. Prior to Docker 1.10, it was recommended to combine all labels into a single `LABEL` instruction, to prevent extra layers from being created. This is no longer necessary, but combining labels is still supported.

```
# Set multiple labels on one line
LABEL com.example.version="0.0.1-beta" com.example.release-date="2015-02-12"
```

The above can also be written as:

```
# Set multiple labels at once, using line-continuation characters to break long
lines
LABEL vendor=ACME\ Incorporated \
      com.example.is-beta= \
      com.example.is-production="" \
      com.example.version="0.0.1-beta" \
      com.example.release-date="2015-02-12"
```

See Understanding object labels for guidelines about acceptable label keys and values. For information about querying labels, refer to the items related to filtering in Managing labels on objects. See also LABEL in the Dockerfile reference.

RUN

Dockerfile reference for the RUN instruction

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

apt-get

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

Avoid `RUN apt-get upgrade` and `dist-upgrade`, as many of the "essential" packages from the parent images cannot upgrade inside an unprivileged container. If a package contained in the parent image is out-of-date, contact its maintainers. If you know there is 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:18.04
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:18.04
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 \
    mercurial \
    reprepro \
    ruby1.9.1 \
    ruby1.9.1-dev \
    s3cmd=1.1.* \
 && rm -rf /var/lib/apt/lists/*
```

The `s3cmd` argument specifies a version `1.1.*`. If the image previously used an older version, specifying the new one causes a cache bust of `apt-get update` and ensures the installation of the new version. Listing packages on each line can also prevent mistakes in package duplication.

In addition, when you clean up the apt cache by removing `/var/lib/apt/lists` it reduces the image size, since the apt cache is not stored in a layer. Since the `RUN` statement starts with `apt-get update`, the package cache is always refreshed prior to `apt-get install`.

> Official Debian and Ubuntu images automatically run `apt-get clean`, so explicit invocation is not required.

Using pipes

Some `RUN` commands depend on the ability to pipe the output of one command into another, using the pipe character (`|`), as in the following example:

```
RUN wget -O - https://some.site | wc -l > /number
```

Docker executes these commands using the `/bin/sh -c` interpreter, which only evaluates the exit code of the last operation in the pipe to determine success. In the example above this build step succeeds and produces a new image so long as the `wc -l` command succeeds, even if the `wget` command fails.

If you want the command to fail due to an error at any stage in the pipe, prepend `set -o pipefail &&` to ensure that an unexpected error prevents the build from inadvertently succeeding. For example:

```
RUN set -o pipefail && wget -O - https://some.site | wc -l > /number
```

> Not all shells support the `-o pipefail` option.
>
> In cases such as the `dash` shell on Debian-based images, consider using the *exec* form of `RUN` to explicitly choose a shell that does support the `pipefail` option. For example:
>
> ```
> RUN ["/bin/bash", "-c", "set -o pipefail && wget -O - https://some.site | wc -l > /number"]
> ```

CMD

Dockerfile reference for the CMD instruction

The `CMD` instruction should be used to run the software contained in 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, such as Apache and Rails, 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, such as bash, python and perl. 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`, unless you and your expected users are already quite familiar with how `ENTRYPOINT` works.

EXPOSE

Dockerfile reference for the EXPOSE instruction

The `EXPOSE` instruction indicates the ports on which a container listens 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

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` ensures 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.

Each `ENV` line creates a new intermediate layer, just like `RUN` commands. This means that even if you unset the environment variable in a future layer, it still persists in this layer and its value can't be dumped. You can test this by creating a Dockerfile like the following, and then building it.

```
FROM alpine
ENV ADMIN_USER="mark"
RUN echo $ADMIN_USER > ./mark
RUN unset ADMIN_USER
```

```
$ docker run --rm test sh -c 'echo $ADMIN_USER'

mark
```

To prevent this, and really unset the environment variable, use a `RUN` command with shell commands, to set, use, and unset the variable all in a single layer. You can separate your commands with `;` or `&&`. If you use the second method, and one of the commands fails, the `docker build` also fails. This is usually a good idea. Using `\` as a line continuation character for Linux Dockerfiles improves readability. You could also put all of the commands into a shell script and have the `RUN` command just run that shell script.

```
FROM alpine
RUN export ADMIN_USER="mark" \
    && echo $ADMIN_USER > ./mark \
    && unset ADMIN_USER
CMD sh
```

```
$ docker run --rm test sh -c 'echo $ADMIN_USER'
```

ADD or COPY

- Dockerfile reference for the ADD instruction
- Dockerfile reference for the COPY instruction

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 ensures 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 --requirement /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 don'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

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 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 "$@"
```

> Configure app as PID 1
>
> This script uses the `exec` Bash command 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. For more, see the `ENTRYPOINT` reference.

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

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

```
CMD ["postgres"]
```

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

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

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 --no-log-init -r -g postgres postgres`.

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

> Due to an unresolved bug in the Go archive/tar package's handling of sparse files, attempting to create a user with a significantly large UID inside a Docker container can lead to disk exhaustion because `/var/log/faillog` in the container layer is filled with NULL (\0) characters. A workaround is to pass the `--no-log-init` flag to useradd. The Debian/Ubuntu `adduser` wrapper does not support this flag.

Avoid installing or using `sudo` as it has unpredictable TTY and signal-forwarding behavior that can cause problems. If you absolutely need functionality similar to `sudo`, such as initializing the daemon as `root` but running it as non-`root`, consider using "gosu".

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

WORKDIR

Dockerfile reference for the WORKDIR instruction

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

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.

Images built with `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 fails catastrophically if the new build's context is missing the resource being added. Adding a separate tag, as recommended above, helps mitigate this by allowing the `Dockerfile` author to make a choice.

Examples for Official Images

These Official Images have exemplary `Dockerfile`s:

- Go
- Perl
- Hy
- Ruby

Additional resources:

- Dockerfile Reference
- More about Base Images
- More about Automated Builds
- Guidelines for Creating Official Images