Best practices for writing Dockerfiles

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 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 building efficient images. To see many of these practices and recommendations in action, check out the Dockerfile for buildpack-deps.

> **Note**: for more detailed explanations of any of the Dockerfile commands mentioned here, visit the Dockerfile Reference 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. You may want to take a look at the Processes section of the 12 Factor app methodology to get a feel for the motivations of running containers in such a stateless fashion.

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 of the 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 .
> ```
>
> Now move `Dockerfile` and `hello` into separate directories and build a second version of the image (without relying on cache from the last build). Use the `-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 build time, time to pull and push the image, and the runtime size of containers. 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
```

Use a .dockerignore file

To exclude files which are 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. In addition to using a `.dockerignore` file, check out the information below on multi-stage builds.

Use multi-stage builds

If you use Docker 17.05 or higher, you can use multi-stage builds to drastically reduce the size of your final image, without the need to jump through hoops to reduce the number of intermediate layers or remove intermediate files during the build.

Images being built by the final stage only, you can most of the time benefit both the build cache and minimize images layers.

Your build stage may contain several layers, ordered from the less frequently changed to the more frequently changed for example:

- 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:1.9.2-alpine3.6 AS build

# Install tools required to build the project
# We need to run `docker build --no-cache .` to update those dependencies
RUN apk add --no-cache git
RUN go get github.com/golang/dep/cmd/dep

# Gopkg.toml and Gopkg.lock lists project dependencies
# 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 all project and build it
# This layer is rebuilt when ever a file has changed 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"]
```

Avoid installing unnecessary packages

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.

Each container should have only one concern

Decoupling applications into multiple containers makes it much 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.

You may have heard that there should be "one process per container". While this mantra has good intentions, it is not necessarily true that there should be only one operating system process per container. In addition to the fact that containers can now be spawned with an init process, some programs might spawn additional processes of their own accord. For instance, Celery can spawn multiple worker processes, or Apache might create a process per request. While "one process per container" is frequently a good rule of thumb, it is not a hard and fast rule. 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

Prior to Docker 17.05, and even more, prior to Docker 1.10, it was important to minimize the number of layers in your image. The following improvements have mitigated this need:

- In Docker 1.10 and higher, only `RUN`, `COPY`, and `ADD` instructions create layers. Other instructions create temporary intermediate images, and no longer directly increase the size of the build.
- Docker 17.05 and higher add support for multi-stage builds, which allow you to copy only 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 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:

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

Build cache

During the process of 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 then it is very 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 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 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.

The Dockerfile instructions

These recommendations help you to write an efficient and maintainable `Dockerfile`.

FROM

Dockerfile reference for the FROM instruction

Whenever possible, use current Official Repositories as the basis for your image. We recommend the Alpine image since it's very tightly controlled and kept minimal (currently under 5 mb), while still being a full 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.

> **Note**: If your string contains spaces, it must be quoted **or** the spaces must be escaped. If your string contains inner quote characters (`"`), escape them as well.

```
# Set one or more individual labels
LABEL com.example.version="0.0.1-beta"
LABEL vendor="ACME 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

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 parent images can't upgrade inside an unprivileged container. If a package contained in the parent 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 \
    mercurial \
    reprepro \
    ruby1.9.1 \
    ruby1.9.1-dev \
    s3cmd=1.1.* \
 && rm -rf /var/lib/apt/lists/*
```

The `s3cmd` instructions 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 ensure 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` 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`.

> **Note**: The 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
```

> **Note**: Not all shells support the `-o pipefail` option. In such cases (such as the `dash` shell, which is the default 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 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, 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 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
CMD sh
```

```
$ docker run --rm -it test sh 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 -it test sh 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 "$@"
```

> **Note**: 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. See the `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"]
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`.

> **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.

> **Note**: Due to an unresolved bug in the Go archive/tar package's handling of sparse files, attempting to create a user with a sufficiently large UID inside a Docker container can lead to disk exhaustion as `/var/log/faillog` in the container layer is filled with NUL (\0) characters. Passing the `--no-log-init` flag to useradd works around this issue. The Debian/Ubuntu `adduser` wrapper does not support the `--no-log-init` flag and should be avoided.

Avoid installing or using `sudo` since 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 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 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 Repositories

These Official Repositories have exemplary `Dockerfile`s:

- Go
- Perl
- Hy
- Ruby

Additional resources:

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