# Dockerfile Best Practices

## 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.

## Understand build context

When you issue a docker build command, the current working directory is called the build 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

## 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 -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.

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

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](https://tldp.org/LDP/abs/html/here-docs.html).

# create a directory to work inmkdir examplecd example

# create an example filetouch 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.txtEOF

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

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](https://github.com/docker-library/hello-world).

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

FROM busybox

COPY hello.c ./

EOF

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](https://docs.docker.com/engine/reference/builder/" \l "dockerignore-file).

## Use multi-stage builds

[Multi-stage builds](https://docs.docker.com/develop/develop-images/multistage-build/) 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](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/" \l "leverage-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

### 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.

### 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](https://docs.docker.com/develop/develop-images/multistage-build/), 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 it 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 \

&& rm -rf /var/lib/apt/lists/\*

### 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.