We work with Dockerfiles on a daily basis; all the code we run for ourselves and for our customers, we run from a set of Dockerfiles. In this article, we’ll talk about what mistakes people commonly make, and how to write them better. For those of you who are Docker experts, a lot of the tips in this article will probably be pretty obvious and will just provoke a lot of head-nodding. But for beginner to intermediate developers, this will be a useful guide that will hopefully help clean and speed up your workflow.

**1. Running apt-get**

Running apt-get install is one of those things virtually every Dockerfile will have. You will probably need to install some external package in order to run your code. But using apt-get comes with its fair share of gotchas.

The first is running apt-get upgrade. This will update all your packages to their latests versions — which is bad because it prevents your Dockerfile from creating consistent, immutable builds.

Another issue is with running apt-get update in a different line than running your apt-get install command. The reason why this is bad is because a line with only apt-get update will get cached by the build and won't actually run every time you need to run apt-get install. Instead, make sure you run apt-get update in the same line with all the packages to ensure all are updated correctly.

The apt-install in the [Golang Dockerfile](https://github.com/docker-library/golang/blob/master/1.7/Dockerfile#L4) is a good example of how this should be done:

# From https://github.com/docker-library/golang

RUN apt-get update && \

apt-get install -y --no-install-recommends \

g++ \

gcc \

libc6-dev \

make \

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

**2. Using ADD instead of COPY**

While similar, ADD and COPY are actually different commands. COPY is the simplest of the two, since it just copies a file or a directory from your host to your image. ADD does this too, but also has some more magical features like extracting TAR files or fetching files from remote URLs. In order to reduce the complexity of your Dockerfile and prevent some unexpected behavior, it's usually best to always use COPY to copy your files.

FROM busybox:1.24

ADD example.tar.gz /add # Will untar the file into the ADD directory

COPY example.tar.gz /copy # Will copy the file directly

**3. Adding your entire application directory in one line**

Being explicit about what part of your code should be included in your build, and at what time, might be the most important thing you can do to significantly speed up your builds.

Often times, when looking at a Dockerfile, you'll see this:

# !!! ANTIPATTERN !!!

COPY ./my-app/ /home/app/

RUN npm install # or RUN pip install or RUN bundle install

# !!! ANTIPATTERN !!!

This means that every time we make a change to any of our files, we’ll have to rebuild everything below that line. In most cases (including the example above), this means having to re-install our application dependencies. In order to use Docker’s cache as smartly as possible, copy over the files that are needed to install all your dependencies first, and then execute the commands that install those dependencies. Doing those two steps before copying over the rest of your application files (which should be done at the latest possible line) will enable your changes to be quickly re-built.

COPY ./my-app/package.json /home/app/package.json # Node/npm packages

WORKDIR /home/app/

RUN npm install

# Maybe you have to install python packages too?

COPY ./my-app/requirements.txt /home/app/requirements.txt

RUN pip install -r requirements.txt

COPY ./my-app/ /home/app/

This will ensure that your builds run as fast as possible.

**4. Using :latest**

Many Dockerfiles use the FROM node:latest pattern at the top of their Dockerfiles to pull the latest image from a Docker registry. While simple, using the latest tag for an image means that your build can suddenly break if that image gets updated. Figuring this out might prove to be very difficult, since the maintainer of the Dockerfile didn’t actually make any changes. To prevent this, just make sure you use a specific tag of an image (example: node:6.2.1). This will ensure your Dockerfile remains immutable.

**5. Using external services during the build**

Many people forget the difference between building a Docker image and running a Docker container. When building an image, Docker reads the commands in your Dockerfile and creates an image from it. Your image should be immutable and reusable until any of your dependencies or your code changes. This process should be completely independent of any other container. Anything that requires interaction with other containers or other services (like a database) should happen when you run the container.

An example of this is running a database migration. Most people attempt to run these when they are building their image. This has a couple of problems. First, the database might not be available during build time, since it might not be built on the same server that it will be running on. Second, you might want to use this same image to connect to different databases (development vs production), at which point the migration would not run if it’s in the build.

# !!! ANTIPATTERN !!!

COPY /YOUR-PROJECT /YOUR-PROJECT

RUN python manage.py migrate

# runserver would actually try to the migration, but imagine it doesn’t

CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]

# !!! ANTIPATTERN !!!

**6. Adding EXPOSE and ENV at the top of your Dockerfile**

EXPOSE and ENV are cheap commands to run. If you bust the cache for them, rebuilding them is almost instantaneous. Therefore, it’s best to declare these commands as late as possible. You should only ever declare ENVs whenever you need them in your build process. If they’re not needed during build time, then they should be at the end of your Dockerfile, along with EXPOSE.

If you look at the Go Dockerfile again, you'll see that they declare all the ENVs they need before using them and declare all other at the end:

ENV GOLANG\_VERSION 1.7beta1

ENV GOLANG\_DOWNLOAD\_URL https://golang.org/dl/go$GOLANG\_VERSION.linux-amd64.tar.gz

ENV GOLANG\_DOWNLOAD\_SHA256 a55e718935e2be1d5b920ed262fd06885d2d7fc4eab7722aa02c205d80532e3b

RUN curl -fsSL "$GOLANG\_DOWNLOAD\_URL" -o golang.tar.gz \

&& echo "$GOLANG\_DOWNLOAD\_SHA256 golang.tar.gz" | sha256sum -c - \

&& tar -C /usr/local -xzf golang.tar.gz \

&& rm golang.tar.gz

ENV GOPATH /go

ENV PATH $GOPATH/bin:/usr/local/go/bin:$PATH

If they need to change the ENV GOPATH or ENV PATH, their image will get rebuilt almost immediately.

**7. Multiple FROM statements**

It might be tempting to try to combine different images together by using multiple FROM statements; this won’t work. Instead, Docker will just use the last FROM specified and ignore everything before that.

So if you have this Dockerfile:

# !!! ANTIPATTERN !!!

FROM node:6.2.1

FROM python:3.5

CMD ["sleep", "infinity"]

# !!! ANTIPATTERN !!!

And then docker exec into that running container, you'll notice the following:

$ docker exec -it d86fcf0775d3 bash

root@d86fcf0775d3:/# which python

/usr/local/bin/python

root@d86fcf0775d3:/# which node

root@d86fcf0775d3:/#

There is actually a GitHub issue for combining different images together, but this does not look like a feature that will be added any time soon.

**8. Multiple services running in the same container**

This is probably the biggest head-nodder for people who already know Docker. It's a well established best-practice that every different service which composes your application should run in its own container. It's tempting to add multiple services to one docker image, but this practice has some downsides.

First, you'll make it more difficult to horizontally scale your application. Second, the additional dependencies and layers will make your build slower. Finally, it'll make your Dockerfile harder to write, maintain, and debug.

Of course, as with all technical advice, you’ll need to use your best judgement. If you want to quickly setup a Django+Nginx application for development, it might make sense to just run them in the same container and have a different Dockerfile in production where they run separately.

**9. Using VOLUME in your build process**

Volumes in your image are added when you run your container, not when you build it. In a similar way to #5, you should never interact with your declared volume in your build process. Rather, you should only use it when you run the container.

For example, if I create a file in my build process and use that file when I run that image, everything works fine:

FROM busybox:1.24

RUN echo "hello-world!!!!" > /myfile.txt

CMD ["cat", "/myfile.txt"]

$ docker run volume-in-build

hello-world!!!!

On the other hand, if I do the same thing for a file stored in a volume, it won't work.

FROM busybox:1.24

VOLUME /data

RUN echo "hello-world!!!!" > /data/myfile.txt

CMD ["cat", "/data/myfile.txt"]

$ docker run volume-in-build

cat: can't open '/data/myfile.txt': No such file or directory

An interesting gotcha for this is that if any of your previous layers has a VOLUME declaration (which might be several FROMs away) you will still run into the same issue. For that reason, it's a good idea to be aware of what volumes your parent images declare. Use docker inspect if you run into problems.

**Conclusion**

Understanding how to write a good Dockerfile will take you a long way into understanding how Docker works and will also help you in abstracting your infrastructure. Understanding the Docker cache alone will save you hours and hours of waiting for a build to finish over time!