Why Docker?

Docker is essentially faster and more secure than virtual environments, as well as easier to deploy and distribute among users which enhances collaboration.

Web development example using alpine:

FROM python:3.6-alpine

WORKDIR /app

COPY requirements.txt /

RUN pip install -r /requirements.txt # flask and gunicorn

COPY . /app

Size before: 702MB, Size after: 102MB

Data Science example using alpine:

FROM python:3.6-alpine

RUN apk --no-cache add --virtual build-dependencies \

build-base \

python3-dev \

&& pip3 install \

jupyter \

pandas

WORKDIR /notebooks

Size before: 929MB, Size after: 365MB

CI example using alpine:

FROM python:3.6 as base

RUN pip wheel --no-cache-dir --no-deps --wheel-dir /wheels -r flask

COPY . /app

# What happens if the tests fail?

RUN py.test

FROM python:3.6-alpine

COPY --from=base /wheels /wheels

RUN pip install --no-cache /wheels/\*

COPY . /app

Order Dockerfile Commands:

Docker caches the steps in a Dockerfile to speed up subsequent builds. When a change is made to a step, all steps following it will be redone(I believe that this means that all the process will be redone again, and if you have a COPY step and that step was right, but it was after one that was changed, that step will run again, making an unnecessary copy). This is called invalidate the cache.

Avoid invalidating the cache by-

* Starting your Dockerfile with commands that are less likely to change
* Putting commands that are more likely to change (like COPY.) as late as possible
* Adding only the necessary files (use <a.dockerignore> file!)

Example:

Web development example using alpine:

FROM python:3.6-alpine

WORKDIR /app

# What happens when a change is made to sample.py?

COPY sample.py /app

COPY requirements.txt /

RUN pip install -r /requirements.txt # flask and gunicorn

What’s wrong?

The statement or step COPY sample.py /app should be moved to the bottom to avoid changes if a change is made to sample.py

Minimize the number of layers:

Docker uses steps to create the image. Each RUN step will create a new layer and use unnecessary disk space. To avoid that the number of layers grow, we must avoid having a lot of run steps. This can be achieved by combining as many run steps as possible.

Things to note:

* RUN, COPY, and ADD steps will create layers.
* Each layer contains the differences from the previous layer.
* Layers increase the size of the final image

Tips:

* Put related commands (apt-get update/install) in the same RUN step.
* Remove files in the same RUN step that created them.
* Avoid using apt-get upgrade since it upgrades all packages to the latest version... Why?
* Use multi-stage builds as much as possible!

Version Docker images:

If you rely on just using the latest tag, you'll have no real way of knowing which image version is actually running in a specific environment.

You could use both the git commit SHA1 hash (to associate the image back to a specific commit to help with debugging) along with and the environment name:

/$PROJECT/$ENVIRONMENT:$SHA1

$ docker build -t web/prod:a072c4e5d94b5a769225f621f08af3d4bf820a07 .

Create a non-root user:

By default, Docker runs container processes as root inside of a container. This is a bad practice since attackers can gain root access to the Docker host if they manage to break out of the container.

If you're root in the container, you'll be root on the host.

Do not store secrets in an image:

Sensitive information refers to database credentials, SSH keys, usernames and passwords, API tokens and others.

The secret data management can be done at run-time(recommended) or at build-time.

At run-time (recommended!):

You can pass secrets in via env variables, but they will be visible in all child processes - i.e., linked containers, docker inspect - and logs. It's also difficult to update them. Passing them in using a shared volume is a better solution, but they should be encrypted (via Vault or KMS) since they are saved to disc.

$ docker run -p 5000:5000 -d -e "foo=bar" -i sample \

gunicorn -b 0.0.0.0:5000 sample:app

f8650976bcb9a50257aa9c39114207bb07d42d89f9ae00f5f2ba36c68fc

$ docker inspect f8650976bcb9a50257aa9c39114207bb07d42d89f9ae00f5f2ba36c68fc

At build-time:

You should never store secrest in a Docker image that will be pushed to a public repo. You can pass secrets in at build-time using build-time args, but they will be visible to those whi have access to the image via docker history.

Dockerfile

FROM alpine

ARG foo

RUN echo "Hello, World!"

Example

$ docker build --build-arg "foo=bar" -t hi .

Successfully built f2bcff49ac09

$ docker history f2bcff49ac09 | grep foo

1796fafd3d00 About a minute ago /bin/sh -c #(nop) ARG foo

Docker Compose:

Docker Compose is an orchestration tool used for running multi-container apps. It helps streamline building, running, and connecting multiple containers together. To use Docker Compose, you'll need to define how to build your containers with YAML in a docker-compose.yml file.

It is important to ship dependences aling with the code to secure environments cut off from the internet