Best Practices of Dockerizing Python Apps

A detailed look on "What to do and What not to do?"

Itamar Turner-Trauring, Hynek Schlawack , Mohammad "Fazel" Fazeli

N/A, N/A, Maktabkhooneh

July 30, 2020

Who am I?

What is this talk about?

- What you already know:
  - Basic knowledge of Docker
  - Basic usage of Python
  - Basic ways of using Docker to run Python code
  - By basic I mean just had done it one time

What is this talk about?

- What you already know:
  - Basic knowledge of Docker
  - Basic usage of Python
  - Basic ways of using Docker to run Python code
  - By basic I mean just had done it one time
- What will I try to discuss in details:
  - **What** is wrong with a lot of examples of Dockerized Python apps
  - **Why** is something wrong with them?
  - **How** to make them better?

What is this talk about?

- What you already know:
  - Basic knowledge of Docker
  - Basic usage of Python
  - Basic ways of using Docker to run Python code
  - By basic I mean just had done it one time
- What will I try to discuss in details:
  - **What** is wrong with a lot of examples of Dockerized Python apps
  - **Why** is something wrong with them?
  - **How** to make them better?
  - What could go wrong when you put a Python app in a container
  - After the app is in the container, a lot of complicated things get solved

What is this talk about?

- What you already know:
  - Basic knowledge of Docker
  - Basic usage of Python
  - Basic ways of using Docker to run Python code
  - By basic I mean just had done it one time
- What will I try to discuss in details:
  - **What** is wrong with a lot of examples of Dockerized Python apps
  - **Why** is something wrong with them?
  - **How** to make them better?
  - What could go wrong when you put a Python app in a container
  - After the app is in the container, a lot of complicated things get solved
  - Then a lot of other complicated things begin :-D

What is this talk about?

- What you already know:
  - Basic knowledge of Docker
  - Basic usage of Python
  - Basic ways of using Docker to run Python code
  - By basic I mean just had done it one time
- What will I try to discuss in details:
  - **What** is wrong with a lot of examples of Dockerized Python apps
  - **Why** is something wrong with them?
  - **How** to make them better?
  - What could go wrong when you put a Python app in a container
  - After the app is in the container, a lot of complicated things get solved
  - Then a lot of other complicated things begin :-D
- What isn't included?
  - Kubernetes/Docker Swarm/Hashicorp Nomad or any other container cluster manager
  - docker-compose or anything like that

Where did these come from?

- More than 3 years of field experience deploying and managing docker containers in production.

Where did these come from?

- More than 3 years of field experience deploying and managing docker containers in production.
- More than 60 blog good blog posts, just for extracting ”Best Practices”.

Where did these come from?

- More than 3 years of field experience deploying and managing docker containers in production.
- More than 60 blog good blog posts, just for extracting ”Best Practices”.
- About 30 of them are from one blog `pythonspeed.com/docker/` by Itamar Turner-Trauring

Table of Contents

What is a Container?

- For the ones who know these stuff, It takes just one minute
- Bear with me for sake of others

Definition (Container)

Container, is a method to package an application so it can be run, with its dependencies, isolated from other processes.

What is a Container?

- For the ones who know these stuff, It takes just one minute
- Bear with me for sake of others

Definition (Container)

Container, is a method to package an application so it can be run, with its dependencies, isolated from other processes.

- OS-level virtualization, that is lightweight, secure and a standard unit of software.

What is Docker?

Definition (Docker)

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers.

- This is the industry's standard (for most of things but not all)
- There is a lot of tooling around it (though not nearly enough)

Docker Image??

Definition (Docker Image)

A Docker image is a read-only template that contains a set of instructions for creating a container that can run on the Docker platform.

Python’s Definition

Definition (Python)

Python is a high-level programming language designed to be easy to read and simple to implement.

Lets get over the boring stuff now.

What is wrong with the containerized Python examples?

- Lets dockerize a Python app:

What is wrong with the containerized Python examples?

- Lets dockerize a Python app:
- Search in google for an example of Python in docker

What is wrong with the containerized Python examples?

- Lets dockerize a Python app:
- Search in google for an example of Python in docker
- We show that it is broken how

What is wrong with the containerized Python examples?

- Lets dockerize a Python app:
- Search in google for an example of Python in docker
- We show that it is broken how
- We'll try to make it better

Dockerfile example

Taken from the first google result on ”Dockerize Python Application” https://runnable.com/docker/python/dockerize-your-python-application

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

Dockerfile example

Taken from the first google result on "Dockerize Python Application" https://runnable.com/docker/python/dockerize-your-python-application

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

What is wrong with this? pystrich is a package for generating barcodes

What is wrong with this? (I)

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

- Non-reproducible build, because of use a vague base image.

What is wrong with this? (I)

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

- Non-reproducible build, because of use a vague base image.
- One of best things about Docker is reproducible builds.

What is wrong with this? (I)

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

- Non-reproducible build, because of use a vague base image.
- One of best things about Docker is reproducible builds.
- Tracking bugs derived from these changes in language is tedious

What is wrong with this? (I)

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

- Non-reproducible build, because of use a vague base image.
- One of best things about Docker is reproducible builds.
- Tracking bugs derived from these changes in language is tedious
- **Solution**: Fix the base image, like

  ```
  FROM python:3.8.3-buster
  ```

What is wrong with this? (II)

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

- Non-reproducible builds on dependencies

What is wrong with this? (II)

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

- Non-reproducible builds on dependencies
- pystrich can change from time to time

What is wrong with this? (II)

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

- Non-reproducible builds on dependencies
- pystrich can change from time to time
- One of the plethora of dependencies the library could change

What is wrong with this? (II)

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

- Non-reproducible builds on dependencies
- pystrich can change from time to time
- One of the plethora of dependencies the library could change
- In the case of multiple libs, dependencies could get into conflict

What is wrong with this? (II)

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

- Non-reproducible builds on dependencies
- pystrich can change from time to time
- One of the plethora of dependencies the library could change
- In the case of multiple libs, dependencies could get into conflict
- **Solution**: Pin versions of libs and dependencies, preferably using one of pip-tools, poetry or pip-env(more on that later)

What is wrong with this? (III)

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

- Cache invalidates on every source code change, causes download and re-installation of dependencies

What is wrong with this? (III)

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

- Cache invalidates on every source code change, causes download and re-installation of dependencies
- **Solution**: Copy files when at the stage that are absolutely necessary, possibly just copy some files then the others

What is wrong with this? (IV)

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

- Runs as root, when there is no need for root permissions(Docker's default behavior).

What is wrong with this? (IV)

```
FROM python:3

ADD my_script.py /

RUN pip install pystrich

CMD [ "python", "./my_script.py" ]
```

- Runs as root, when there is no need for root permissions(Docker's default behavior).
- **Solution**: Run as non-root user, you don't have to listen to low ports, You can just proxy them and/or give specific permission for low ports

A somewhat better solution

```
FROM python:3.8.3-buster

COPY requirements.txt /tmp/

RUN pip install -r /tmp/requirements.txt

RUN useradd --create-home appuser
WORKDIR /home/appuser
USER appuser

COPY my_script.py .

CMD [ "python", "./my_script.py" ]
```

- Further improvement?

Official Docker Best practices review(I)

- Containers should be as ephemeral as possible

Official Docker Best practices review(I)

- Containers should be as ephemeral as possible
- Exclude files using ‘.dockerignore‘

Official Docker Best practices review(I)

- Containers should be as ephemeral as possible
- Exclude files using '.dockerignore'
- Use multi stage builds(discussed in detail here)

Official Docker Best practices review(I)

- Containers should be as ephemeral as possible
- Exclude files using ‘.dockerignore‘
- Use multi stage builds(discussed in detail here)
- Don’t install unnecessary packages

Official Docker Best practices review(I)

- Containers should be as ephemeral as possible
- Exclude files using '.dockerignore'
- Use multi stage builds(discussed in detail here)
- Don't install unnecessary packages
- Decouple applications

Official Docker Best practices review(I)

- Containers should be as ephemeral as possible
- Exclude files using '.dockerignore'
- Use multi stage builds(discussed in detail here)
- Don't install unnecessary packages
- Decouple applications
- Minimize number of layers

Official Docker Best practices review(I)

- Containers should be as ephemeral as possible
- Exclude files using ‘.dockerignore‘
- Use multi stage builds(discussed in detail here)
- Don’t install unnecessary packages
- Decouple applications
- Minimize number of layers
- Sort multi-line arguments

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

Official Docker Best practices review(I)

- Containers should be as ephemeral as possible
- Exclude files using ‘.dockerignore‘
- Use multi stage builds(discussed in detail here)
- Don’t install unnecessary packages
- Decouple applications
- Minimize number of layers
- Sort multi-line arguments

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

- Leverage build caches

Official Docker Best practices review(I)

- Containers should be as ephemeral as possible
- Exclude files using ‘.dockerignore‘
- Use multi stage builds(discussed in detail here)
- Don’t install unnecessary packages
- Decouple applications
- Minimize number of layers
- Sort multi-line arguments

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

- Leverage build caches
- and more

Docker networking and its consequences

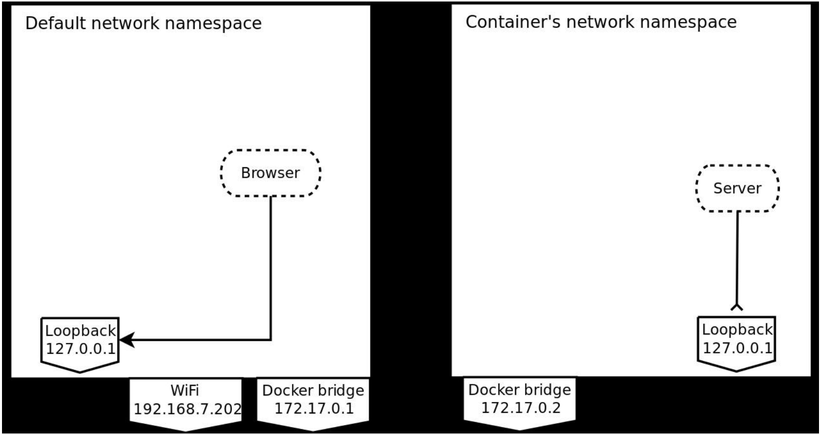
- Networks in Docker are isolated by default and have a clean relationship with host.
- Their interface, IP, firewall rules and etc are all different from host and each other

```
$ python3 -m http.server --bind 127.0.0.1
Serving HTTP on 127.0.0.1 port 8000 (http
    ↪ ://127.0.0.1:8000/) ...
```

an other is this:

```
$ docker run -p 8000:8000 -it --entrypoint=bash
    ↪ python:3.7-slim
root@85bdb39291b3:/# python3 -m http.server --
    ↪ bind 127.0.0.1
Serving HTTP on 127.0.0.1 port 8000 (http
    ↪ ://127.0.0.1:8000/) ...
```

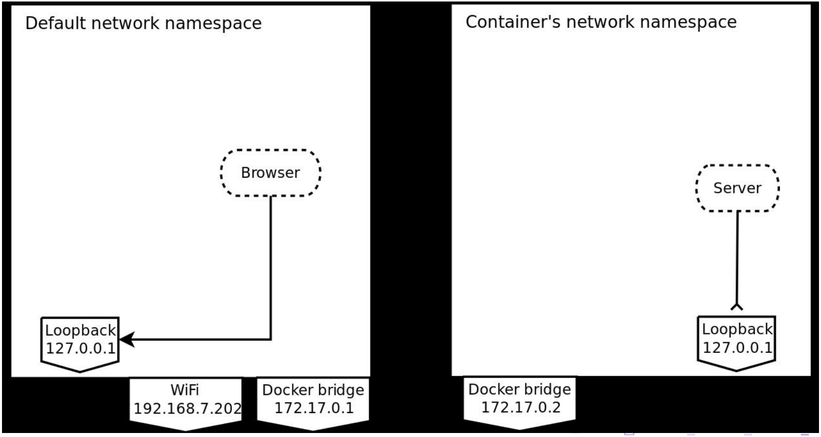
Docker networking and its consequences II



Docker networking and its consequences III

- Docker forwards on *all interfaces* to the container's port

```
$ docker run -p 8000:8000 -it python:3.7-slim python3
  ↪  -m http.server --bind 0.0.0.0
```



Why builds are slow?

- It takes time to download and install packages, building C extensions and etc

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc
- Good news is most of these stuff doesn't change frequently

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc
- Good news is most of these stuff doesn't change frequently
- The Basic caching algorithm:

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc
- Good news is most of these stuff doesn't change frequently
- The Basic caching algorithm:
  - For most commands if the text of command hasn't changed, it would be cached

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc
- Good news is most of these stuff doesn't change frequently
- The Basic caching algorithm:
  - For most commands if the text of command hasn't changed, it would be cached
  - For COPY, it checks if hash of the file has been changed or not

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc
- Good news is most of these stuff doesn't change frequently
- The Basic caching algorithm:
  - For most commands if the text of command hasn't changed, it would be cached
  - For COPY, it checks if hash of the file has been changed or not
- For example if code is copied into to image build process before installing dependencies, it would invalidate dependencies and every time you change the code, you would reinstall packages needlessly.

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc
- Good news is most of these stuff doesn't change frequently
- The Basic caching algorithm:
  - For most commands if the text of command hasn't changed, it would be cached
  - For COPY, it checks if hash of the file has been changed or not
- For example if code is copied into to image build process before installing dependencies, it would invalidate dependencies and every time you change the code, you would reinstall packages needlessly.
- Make sure you copy the files when exactly needed so changing them would invalidate what is necessary

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc
- Good news is most of these stuff doesn't change frequently

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc
- Good news is most of these stuff doesn't change frequently
- The Basic caching algorithm:

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc
- Good news is most of these stuff doesn't change frequently
- The Basic caching algorithm:
  - For most commands if the text of command hasn't changed, it would be cached

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc
- Good news is most of these stuff doesn't change frequently
- The Basic caching algorithm:
  - For most commands if the text of command hasn't changed, it would be cached
  - For COPY, it checks if hash of the file has been changed or not

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc
- Good news is most of these stuff doesn't change frequently
- The Basic caching algorithm:
    - For most commands if the text of command hasn't changed, it would be cached
    - For COPY, it checks if hash of the file has been changed or not
    - If cache in one layer is invalidated, all subsequent layers caches would be invalid and they are going to built again

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc
- Good news is most of these stuff doesn't change frequently
- The Basic caching algorithm:
  - For most commands if the text of command hasn't changed, it would be cached
  - For COPY, it checks if hash of the file has been changed or not
  - If cache in one layer is invalidated, all subsequent layers caches would be invalid and they are going to built again
- For example if code is copied into to image build process before installing dependencies, it would invalidate dependencies and every time you change the code, you would reinstall packages needlessly.

Why builds are slow?

- It takes time to download and install packages, building C extensions and etc
- Good news is most of these stuff doesn’t change frequently
- The Basic caching algorithm:
  - For most commands if the text of command hasn’t changed, it would be cached
  - For COPY, it checks if hash of the file has been changed or not
  - If cache in one layer is invalidated, all subsequent layers caches would be invalid and they are going to built again
- For example if code is copied into to image build process before installing dependencies, it would invalidate dependencies and every time you change the code, you would reinstall packages needlessly.
- Make sure you copy the files when exactly needed so changing them would invalidate what is necessary

Cache Example(skippable)

The following is not cool

```
FROM python:3.7-slim-buster

COPY requirements.txt .
COPY server.py .

RUN pip install --quiet -r requirements.txt

ENTRYPOINT ["python", "server.py"]
```

This is the cool one:

```
FROM python:3.7-slim-buster

COPY requirements.txt .

RUN pip install --quiet -r requirements.txt
```

Choosing a base image

Desiderata

- Stability
- Security updates
- Up to date dependencies
- Extensive dependencies
- Up-to-date Python
- Small images

Choosing a base image II

Generally Alpine is not cool, since:

- Have obscure bugs

Choosing a base image II

Generally Alpine is not cool, since:

- Have obscure bugs
- In some cases makes builds slower

Choosing a base image II

Generally Alpine is not cool, since:

- Have obscure bugs
- In some cases makes builds slower
- In some cases makes images bigger !!

Choosing a base image II

Generally Alpine is not cool, since:

- Have obscure bugs
- In some cases makes builds slower
- In some cases makes images bigger !!
- People use because base image is 5MB, in contrast ubuntu:18.04 is 65MB

Alpine problems with Python

- A lot of useful packages use C dependencies(matplotlib, numpy, pandas, psycopg2, MySQLdb and ...)

Alpine problems with Python

- A lot of useful packages use C dependencies(matplotlib, numpy, pandas, psycopg2, MySQLdb and ...)
- When installed from PyPI, almost always you install a pre-compiled package called wheel.

Alpine problems with Python

- A lot of useful packages use C dependencies(matplotlib, numpy, pandas, psycopg2, MySQLdb and ...)
- When installed from PyPI, almost always you install a pre-compiled package called wheel.
- These are compiled for linux distros using glibc as C standard lib

Alpine problems with Python

- A lot of useful packages use C dependencies(matplotlib, numpy, pandas, psycopg2, MySQLdb and ...)
- When installed from PyPI, almost always you install a pre-compiled package called wheel.
- These are compiled for linux distros using glibc as C standard lib
- One of secret sauces of Alpine is it doesn't use glibc but a smaller yet compatible(or trying to be compatible) version named ‘musl‘

Alpine Problems with Python II

- Because of ‘musl‘,most of pre-compiled wheels doesn’t work for them,they and you have to compile each of these libraries.

Alpine Problems with Python II

- Because of ‘musl‘,most of pre-compiled wheels doesn’t work for them,they and you have to compile each of these libraries.
- There is a way of building wheels called ‘manylinux‘ which is a very simple way of distributing your package for many linux systems and most packages use this. These aren’t MUSL compatible.

Alpine Problems with Python II

- Because of ‘musl‘,most of pre-compiled wheels doesn’t work for them,they and you have to compile each of these libraries.
- There is a way of building wheels called ‘manylinux‘ which is a very simple way of distributing your package for many linux systems and most packages use this. These aren’t MUSL compatible.
- For example installing the famous pandas packages using a wheel takes less than 1 minute but building it takes 20 minutes on my machine.

Alpine Problems with Python III

- With some workarounds you can build most of these packages in alpine but image size would become huge.(bc workarounds)

Alpine Problems with Python IV

Obscure Bugs of Alpine+Python:

- MUSL has occasional incompatibilies with glibc(apps assume glibc behavior), these are considered bugs but they happen fairly often

Alpine Problems with Python IV

Obscure Bugs of Alpine+Python:

- MUSL has occasional incompatibilies with glibc(apps assume glibc behavior), these are considered bugs but they happen fairly often
- Alpine by default has smaller stack-size for threads that could lead to Python crashes, here

Alpine Problems with Python IV

Obscure Bugs of Alpine+Python:

- MUSL has occasional incompatibilies with glibc(apps assume glibc behavior), these are considered bugs but they happen fairly often
- Alpine by default has smaller stack-size for threads that could lead to Python crashes, here
- Memory allocations differ from glibc and can cause slow runs. here

Good base images

Long-term supported mainstream distros are good

- Ubuntu 18.04 and 20.04 are in good shape
- CentOS 8 and Debian 10 Buster are very good too
- They have most recent packages(18.04 lacks behind slightly) and all receive security updates for at least 4 years(CentOS receives until 2029)
- 20.04 has an edge because of using the latest stuff

Good base images

Python Official Docker images are a great choice

- There is alpine, buster/buster-slim(Debian 10), stretch/stretch-slim(Debian 9)

Good base images

Python Official Docker images are a great choice

- There is alpine, buster/buster-slim(Debian 10), stretch/stretch-slim(Debian 9)
- Alpine isn't recommended, Buster is very good

Good base images

Python Official Docker images are a great choice

- There is alpine, buster/buster-slim(Debian 10), stretch/stretch-slim(Debian 9)
- Alpine isn't recommended, Buster is very good
- slim version are a lot smaller

Good base images

Python Official Docker images are a great choice

- There is alpine, buster/buster-slim(Debian 10), stretch/stretch-slim(Debian 9)
- Alpine isn't recommended, Buster is very good
- slim version are a lot smaller
- ‘python:3.8-slim-buster‘ is 60MB

Good base images

Python Official Docker images are a great choice

- There is alpine, buster/buster-slim(Debian 10), stretch/stretch-slim(Debian 9)
- Alpine isn't recommended, Buster is very good
- slim version are a lot smaller
- ‘python:3.8-slim-buster‘ is 60MB
- ‘python:3.8-alpine‘ is 35MB

Good base images

Python Official Docker images are a great choice

- There is alpine, buster/buster-slim(Debian 10), stretch/stretch-slim(Debian 9)
- Alpine isn't recommended, Buster is very good
- slim version are a lot smaller
- ‘python:3.8-slim-buster‘ is 60MB
- ‘python:3.8-alpine‘ is 35MB
- By not installing some packages and having less layers

Good base images

Python Official Docker images are a great choice

- There is alpine, buster/buster-slim(Debian 10), stretch/stretch-slim(Debian 9)
- Alpine isn't recommended, Buster is very good
- slim version are a lot smaller
- ‘python:3.8-slim-buster‘ is 60MB
- ‘python:3.8-alpine‘ is 35MB
- By not installing some packages and having less layers
- But if you install those lacking packages it becomes bigger than the regular ones.

Good base images

Python Official Docker images are a great choice

- There is alpine, buster/buster-slim(Debian 10), stretch/stretch-slim(Debian 9)
- Alpine isn't recommended, Buster is very good
- slim version are a lot smaller
- ‘python:3.8-slim-buster‘ is 60MB
- ‘python:3.8-alpine‘ is 35MB
- By not installing some packages and having less layers
- But if you install those lacking packages it becomes bigger than the regular ones.
- even ‘python:3.8.3-slim-buster‘ isn't a fix image, it have changes with system packages, pip version.

Good base images

Python Official Docker images are a great choice

- There is alpine, buster/buster-slim(Debian 10), stretch/stretch-slim(Debian 9)
- Alpine isn't recommended, Buster is very good
- slim version are a lot smaller
- ‘python:3.8-slim-buster‘ is 60MB
- ‘python:3.8-alpine‘ is 35MB
- By not installing some packages and having less layers
- But if you install those lacking packages it becomes bigger than the regular ones.
- even ‘python:3.8.3-slim-buster‘ isn't a fix image, it have changes with system packages, pip version.
- if you want complete reproducibility pin base image using hash for example: python@sha256:89d719142

Good base images

Python Official Docker images are a great choice

- There is alpine, buster/buster-slim(Debian 10), stretch/stretch-slim(Debian 9)
- Alpine isn't recommended, Buster is very good
- slim version are a lot smaller
- 'python:3.8-slim-buster' is 60MB
- 'python:3.8-alpine' is 35MB
- By not installing some packages and having less layers
- But if you install those lacking packages it becomes bigger than the regular ones.
- even 'python:3.8.3-slim-buster' isn't a fix image, it have changes with system packages, pip version.
- if you want complete reproducibility pin base image using hash for example: python@sha256:89d719142
- Docker won't keep images indefinitely though, you can move them to your own registry

Good base images

Python Official Docker images are a great choice

- There is alpine, buster/buster-slim(Debian 10), stretch/stretch-slim(Debian 9)
- Alpine isn’t recommended, Buster is very good
- slim version are a lot smaller
- ‘python:3.8-slim-buster‘ is 60MB
- ‘python:3.8-alpine‘ is 35MB
- By not installing some packages and having less layers
- But if you install those lacking packages it becomes bigger than the regular ones.
- even ‘python:3.8.3-slim-buster‘ isn’t a fix image, it have changes with system packages, pip version.
- if you want complete reproducibility pin base image using hash for example: python@sha256:89d719142
- Docker won’t keep images indefinitely though, you can move them to your own registry
- ‘bitnami‘ is an image provider that has tags which are

When to upgrade your Python I

- It is not about 2vs3, though there is a fork of Python2 that is community maintained, it is here: tauthon

When to upgrade your Python I

- It is not about 2vs3, though there is a fork of Python2 that is community maintained, it is here: tauthon
- Why upgrade:

When to upgrade your Python I

- It is not about 2vs3, though there is a fork of Python2 that is community maintained, it is here: tauthon
- Why upgrade:
    - There are gonna be bugs that won't be fixed ever, some of them are security ones

When to upgrade your Python I

- It is not about 2vs3, though there is a fork of Python2 that is community maintained, it is here: tauthon
- Why upgrade:
  - There are gonna be bugs that won't be fixed ever, some of them are security ones
  - 3.5 receives security updates until 2020-09-13, 3.8 until 2024-10, but putting it back increase the amount of work needed by much, only 3.8 is in full support

When to upgrade your Python I

- It is not about 2vs3, though there is a fork of Python2 that is community maintained, it is here: tauthon
- Why upgrade:
    - There are gonna be bugs that won't be fixed ever, some of them are security ones
    - 3.5 receives security updates until 2020-09-13, 3.8 until 2024-10, but putting it back increase the amount of work needed by much, only 3.8 is in full support
    - *

When to upgrade your Python I

- It is not about 2vs3, though there is a fork of Python2 that is community maintained, it is here: tauthon
- Why upgrade:
  - There are gonna be bugs that won't be fixed ever, some of them are security ones
  - 3.5 receives security updates until 2020-09-13, 3.8 until 2024-10, but putting it back increase the amount of work needed by much, only 3.8 is in full support
  - *
  - Performance has improved in recent version 3.7 and 3.8

When to upgrade your Python I

- It is not about 2vs3, though there is a fork of Python2 that is community maintained, it is here: tauthon
- Why upgrade:
  - There are gonna be bugs that won't be fixed ever, some of them are security ones
  - 3.5 receives security updates until 2020-09-13, 3.8 until 2024-10, but putting it back increase the amount of work needed by much, only 3.8 is in full support
  - *
  - Performance has improved in recent version 3.7 and 3.8
  - Good stuff (walrus operator, f-string, dataclass, secrets module, breakpoint, @ for matrix multiplication), since changing parser in 3.9 a lot of new syntactic sugar could be added

When to upgrade your Python I

- It is not about 2vs3, though there is a fork of Python2 that is community maintained, it is here: tauthon
- Why upgrade:
  - There are gonna be bugs that won't be fixed ever, some of them are security ones
  - 3.5 receives security updates until 2020-09-13, 3.8 until 2024-10, but putting it back increase the amount of work needed by much, only 3.8 is in full support
  - *
  - Performance has improved in recent version 3.7 and 3.8
  - Good stuff (walrus operator, f-string, dataclass, secrets module, breakpoint, @ for matrix multiplication), since changing parser in 3.9 a lot of new syntactic sugar could be added
  - Movements towards a Go like concurrency control system by sub-interpreter in PEP 554,(check here)

When to upgrade your Python II

- Why not to Upgrade:
  - Missing packages
  - Bugs in Python(3.7 had a lot of them solved in 3.7.1)
  - You can't use syntax because of lacking tools(linters, autoformatters)

When to upgrade your Python III

- All your libraries support the new version
- All your tooling support the new version
- Maybe postpone it to a bug fix version like 3.9.1

Reproducible Builds

- You build a container, a few month later, you fix a minor bug, then the same container doesn't build or it just don't work
- One cause is you have different versions of Python, packages, libraries and even OS
- If your builds aren't reproducible, a minor change can spiral out of control
- I end up changing uWSGI with Gunicorn in production and figure out configs on the fly
- There are many layers of reproducible build e.g. Python, libs, binary libs, OS and etc.

Reproducible Builds(system packages)

- System packages seems stable, if the OS is stable
- In practice if you have enough containers, having different versions of nginx or other libs could become problems
- If you wanna avoid that maybe instead of

  ```
  RUN apt-get install -y nginx
  ```

  do it like this

  ```
  RUN apt-get install -y nginx=1.14.0-0ubuntu1.7
  ```

- There something for even more paranoid that is Nix which generate Docker Images

Reproducible Builds(Python Packages)

- If you had ran

  ```
  pip install django
  ```

  at October 5, 2017 you got django 1.11.6, if you had ran it at December 2, 2017 you got 2.0.0
- This two version handle URL routing differently and any projects written with them are incompatible
- In three month you get not-working-code, ‘url‘ changed to ‘re_path‘
- Possible solution:

  ```
  pip install django=1.11
  ```

- it means ‘1.11.0‘ if you want the latest minor version which is not reproducible:

  ```
  pip install 'django<=1.12'
  ```

Reproducible Builds(Python Packages)

- There is a problem, dependencies of django aren’t pinned.
- You can to ‘pip freeze > requirements.txt‘

```
$ cat requirements.txt
requests[security]
flask
gunicorn==19.4.5
```

- Is it good enough?

Reproducible Builds(Python Packages)

```
$ cat requirements.txt
cffi==1.5.2
cryptography==1.2.2
enum34==1.1.2
Flask==0.10.1
gunicorn==19.4.5
idna==2.0
ipaddress==1.0.16
itsdangerous==0.24
Jinja2==2.8
MarkupSafe==0.23
ndg-httpsclient==0.4.0
pyasn1==0.1.9
pycparser==2.14
pyOpenSSL==0.15.1
requests==2.9.1
six==1.10.0
```

Reproducible Builds

It is hard to upgrade, You have know upper dependencies

```
pip install --upgrade flask gunicorn requests[
    ↪ security]
```

Is there a more convenient way?

- ‘pipenv‘, ‘poetry‘ and ‘pip-tools‘

Reproducible Builds

It is hard to upgrade, You have know upper dependencies

```
pip install --upgrade flask gunicorn requests[
    ↪ security]
```

Is there a more convenient way?

- ‘pipenv‘, ‘poetry‘ and ‘pip-tools‘
- I use ‘pip-tools‘ since it is simpler

Reproducible Builds

It is hard to upgrade, You have know upper dependencies

```
pip install --upgrade flask gunicorn requests[
    ↪ security]
```

Is there a more convenient way?

- ‘pipenv‘, ‘poetry‘ and ‘pip-tools‘
- I use ‘pip-tools‘ since it is simpler
- First

```
$ pip install pip-tools
```

Reproducible Builds

It is hard to upgrade, You have know upper dependencies

```
pip install --upgrade flask gunicorn requests[
    ↪ security]
```

Is there a more convenient way?

- ‘pipenv‘, ‘poetry‘ and ‘pip-tools‘
- I use ‘pip-tools‘ since it is simpler
- First

  ```
  $ pip install pip-tools
  ```

- You make a ‘requirements.in‘ that is top level dependencies, something like:

  ```
  django==3.0.3
  ```

New Package Management Tools

- Then do a

```
$ pip-compile > requirements.txt
$ cat requirements.txt
#
# This file is autogenerated by pip-compile
# To update, run:
#
#    pip-compile
#
asgiref==3.2.3            # via django
django==3.0.3             # via requirements.in
pytz==2019.3              # via django
sqlparse==0.3.0           # via django
```

New Package Management Tools

- Use ‘pip-sync‘, it syncs your Python packages with ‘requirements.txt‘
- For upgrading top-level deps just change ‘requirements.in‘, then do ‘pip-compile‘
- For reproducibility install

  ```
  $ pip install requirements.txt
  or
  $ pip-sync
  ```
- Putting deps in setup.py would need a full code copy then install process, which in turn invalidates cache of packages, just copy it in another file

Problems with Pinning

- You need upgrade
- Bc Security updates and critical bug fixes
- Bc of new features, APIs and fixes for less serious bugs and etc
- There is value in not updating but putting it behind for long has a price in hardship of updates
- Security updates are a must but at a time you have to upgrade and if put on hold too much it becomes harder and harder

How to update

- requires.io and PyUP, Check your dependencies for security updates automatically and do alerting(Github does it too)
- Once in while, like every month or two or even six, do a general upgrade of your dependencies
- This help them not get too much behind the releases
- This requires you to have good enough tests in place, with tests it is much less scary

Installing System Packages

- Needs:
  - Upgrading system packages for security
  - Installing new packages needed for app to run
- Example of a problem:

```
FROM python:3.8-slim-buster
RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get -y install syslog-ng
```

Lets check size:

```
$ docker build −t python−with−syslog .
 ...
$ docker image ls −−format "{{ .Size }}" python:3.8−slim−
    ↪ buster
193MB
$ docker image ls −−format "{{ .Size }}" python−with−syslog
327MB
```

Installing System Packages

- There is two causes
    - Extra Packages installed wihtout notice
    - Cache files and indecies downloaded behind the scenes
- Install just the need packages
- Clean up after it, there is some caveats to it
- Each line in Dockerfile is commit into an extra layer in image
- A file create doesn't really get remove, deleted ones just get overlayed, but are actually there
- You need to do installing and cleaning in one line

```dockerfile
FROM python:3.8-slim-buster

COPY install-packages.sh .
RUN ./install-packages.sh
```

Installing System Packages

```
#!/bin/bash
set -euo pipefail
export DEBIAN_FRONTEND=noninteractive
apt-get update
apt-get -y upgrade
apt-get -y install --no-install-recommends syslog-ng
rm -rf /var/lib/apt/lists/*
```

This is the size after that change

```
$ docker build −t python−with−syslog−2 .
...
$ docker image ls −−format "{{ .Size }}" python−with−syslog−2
238MB
```

Do something like that for your distro of your choice

Capabilities Problems

- Capabilities in Linux gives you some root powers without being root(That is bad because of CVEs)
- Docker gives you a lot these capabilites
- This could be security issues
- Just keep the ones you need

Capabilities Example

Seeing some capabilites, example Dockerfile:

```
FROM ubuntu:18.04
RUN apt−get update && apt−get install −y libcap2−bin inetutils−ping
CMD [”/sbin/getpcaps”, ”1”]
```

Here is capabilities of a Docker Container:

```
$ docker run --rm getpcaps
Capabilities for '1': = cap_chown,cap_dac_override,
    ↪ cap_fowner,cap_fsetid,cap_kill,cap_setgid,
    ↪ cap_setuid,cap_setpcap,cap_net_bind_service,
    ↪ cap_net_raw,cap_sys_chroot,cap_mknod,
    ↪ cap_audit_write,cap_setfcap+eip
```

Capabilites Example

```
$ docker run --rm --user 1000 getpcaps
Capabilities for '1': = cap_chown,cap_dac_override,
    ↪ cap_fowner,cap_fsetid,cap_kill,cap_setgid,
    ↪ cap_setuid,cap_setpcap,cap_net_bind_service,
    ↪ cap_net_raw,cap_sys_chroot,cap_mknod,
    ↪ cap_audit_write,cap_setfcap+i
```

The only difference is '+i' vs '+eip'
'+i' means inherited, '+eip' means effective, inherited, permitted
The user itself doesn't have Caps but its child process can get it back by running an executable that can escalate permissions e.g. via setuid

How to solve Caps

```
$ docker run --rm --user 1000 -it --cap-drop ALL
    ↪ getpcaps /bin/bash
I have no name!@aacfefe4cc3a:/$ ping 192.167.7.1
ping: Lacking privilege for raw socket.
```

ping needs CAP_NET_RAW or SETUID
By dropping Caps, OS prevents the container from access to it

How to solve Caps

With out dropping:

```
$ docker run --rm --user 1000 -it getpcaps /bin/bash
I have no name!@9ce0e2e21c21:/$ ping 192.168.7.1
PING 192.168.7.1 (192.168.7.1): 56 data bytes
64 bytes from 192.168.7.1: icmp_seq=0 ttl=63 time
    ↪ =0.675 ms
64 bytes from 192.168.7.1: icmp_seq=1 ttl=63 time
    ↪ =25.509 ms
```

Caching, the good and the bad

- How caching works?(seen before)
- If you rebuild your image regularly, it actually doesn't build it from scratch, just the deemed changed ones
- It can make your regular upgrade ineffective and give you false sense of security
- You force rebuild from scratch using the build option '–no-cache'
- You can force redownload of image from your image registry by '–pull'
- Do regular rebuilds(e.g. weekly, biweekly, monthly ...) using both '–no-cache' and '–pull'
- Putting rebuilding in CI/CD is a good idea

Handling Secrets

- Not about code secrets, that should be handled separately
- Build secrets, like authentication info of your private pip repository
- Secrets that get into image and remain there for more than one line, are get committed in layers of image
- They can't be removed from image but deleting the file in later commands(just get overlaid)
- A lot of famous opensource images contain such secrets

Handling Build Secrets

- You can use BuildKit, It should be turn on to be usable
- For secrets

```
# syntax = docker/dockerfile:1.0-experimental
FROM alpine
# shows secret from default secret location:
RUN --mount=type=secret,id=mysecret cat /run/
    ↪ secrets/mysecret
```

Handling Build Secrets

- You can do ssh auth access too

```
# syntax=docker/dockerfile:experimental
FROM alpine
# Install ssh client and git
RUN apk add --no-cache openssh-client git
# Download public key for github.com
RUN mkdir -p -m 0600 ~/.ssh && ssh-keyscan github
    ↪ .com >> ~/.ssh/known_hosts
# Clone private repository
RUN --mount=type=ssh git clone git@github.com:
    ↪ myorg/myproject.git myproject
```

- You can use these in tandem to get ssh key using secrets and authenticate using it to your private git repos

Bad methods of Handling Build Secrets

- Enviromental variable and volumes, they aren't available by default(Only through BuildKit which is experimental).
- COPY secrets in a file, they end up in image layers even when explicitly deleted after use
- Pass using '–build-arg', It can be accessed by

  ```
  docker history --no-trunc <yourimage>
  ```

- Passing secret in a former stages in multi-stage builds, if you don't push the first image it is secure
- It can be passed using a network service that runs while the image is building

Security Scanners

Use Bandit for scanning Python code

```
import pickle
import sys
from urllib.request import urlopen

obj = pickle.loads(urlopen(sys.argv[1]).read())
print(obj)
```

Security Scanners

Use Bandit for scanning Python code

Scan result:

```
$ bandit example.py
...
>> Issue: [B403:blacklist] Consider possible security implications
   ↪ associated with pickle module.
...
>> Issue: [B301:blacklist] Pickle and modules that wrap it can be
   ↪ unsafe when used to deseria$ize untrusted data, possible
   ↪ security issue.
...
>> Issue: [B310:blacklist] Audit url open for permitted schemes.
   ↪ Allowing use of file:/ or custom schemes is often
   ↪ unexpected.
...
```

Another Scanner is Pysa

Security Scanners

safety is scanner for your dependencies

```
$ pip install django==1.8
$ pip install safety
$ safety check
...
| package    | installed | affected       | ID        |
+===========+===========+==============+==========+
| django     | 1.8       | <1.11.27       | 37771     |
| django     | 1.8       | <1.8.10        | 33074     |
| django     | 1.8       | <1.8.10        | 33073     |
...
$ pip install --upgrade django
$ safety check
+============================================+
| No known security vulnerabilities found.   |
+============================================+
```

Security Scanners

```
$ trivy −−light python:3.7−slim−buster

python:3.7−slim−buster (debian 10.3)
=====================================    Total: 102 (UNKNOWN: 0, LOW: 71,
   ↪ MEDIUM: 31, HIGH: 0, CRITICAL: 0)

+−−−−−−−−−−−−−−−−|−−−−−−−−−−−−−−−−−−−−−|−−−−−−−−−−−−−−−+
   ↪
|    LIBRARY     | VULNERABILITY ID  | FIXED VERSION |
+−−−−−−−−−−−−−−−−|−−−−−−−−−−−−−−−−−−−−−|−−−−−−−−−−−−−−−+
   ↪
| apt           | CVE−2020−3810     | 1.8.2.1       |
+               +−−−−−−−−−−−−−−−−−−−−−|−−−−−−−−−−−−−−−+
|               | CVE−2011−3374     |               |
+−−−−−−−−−−−−−−−−|−−−−−−−−−−−−−−−−−−−−−|−−−−−−−−−−−−−−−+
   ↪
| bash          | CVE−2019−18276    |               |
+               +−−−−−−−−−−−−−−−−−−−−−|−−−−−−−−−−−−−−−+
|               | TEMP−0841856−B18BAF |             |
+−−−−−−−−−−−−−−−−|−−−−−−−−−−−−−−−−−−−−−|−−−−−−−−−−−−−−−+
   ↪
| coreutils     | CVE−2016−2781     |               |
+               +−−−−−−−−−−−−−−−−−−−−−|−−−−−−−−−−−−−−−+
|               | CVE−2017−18018    |               |

... many more vulnerabilities here ...
```

Cost of slow Image builds

- Slow image builds are annoying but how important are they in the grant scheme of things
- The time to solve it could be use to develop *Features*
- Using back of envelope calculations we can estimate it:
  - Assume each build takes 6
  - You have 10 devs each build 2 times per day
  - You have 2 hours per day wasted on builds
  - .25 of a dev per day
  - .25*salary is the price of builds
  - Switching tasks lower your performance, American Psychological Association got a good article on it here

VirtualEnvs in Docker

This doesn't work

```
FROM python:3.8-slim-buster

RUN python3 -m venv /opt/venv

RUN . /opt/venv/bin/activate
COPY requirements.txt .
RUN pip install -r requirements.txt

COPY myapp.py .
CMD ["python", "myapp.py"]
```

Why?

VirtualEnvs in Docker

This mostly works

```dockerfile
FROM python:3.8-slim-buster

RUN python3 -m venv /opt/venv

COPY requirements.txt .
RUN /opt/venv/bin/pip install -r requirements.txt

COPY myapp.py .
CMD ["/opt/venv/bin/python", "myapp.py"]
```

VirtualEnvs in Docker

This always works

```
FROM python:3.8-slim-buster

RUN python3 -m venv /opt/venv

COPY requirements.txt .
RUN . /opt/venv/bin/activate && pip install -r
    ↪ requirements.txt

COPY myapp.py .
CMD . /opt/venv/bin/activate && exec python myapp.py
^^I
```

VirtualEnvs in Docker

What does activate do?

- It is just a convenience
  1. It checks for your shell
  2. Adds a deactivate function to shell and fixes pydoc
  3. Adds name of venv in your shell prompt
  4. Unset PYTHONHOME environmental variable
  5. Set two variables VIRTUAL_ENV and PATH
- The first four are just irrelevant in Docker

VirtualEnvs in Docker

Here is a simple way to do the same

```
FROM python:3.8-slim-buster

ENV VIRTUAL_ENV=/opt/venv
RUN python3 -m venv $VIRTUAL_ENV
ENV PATH="$VIRTUAL_ENV/bin:$PATH"

# Install dependencies:
COPY requirements.txt .
RUN pip install -r requirements.txt

COPY myapp.py .
CMD ["python", "myapp.py"]
```

Multi-Stage Builds

Building an image with compiled binary(C/C++/Rust and etc) dependencies naively makes images big

```
FROM python:3.7-slim
RUN apt-get update
RUN apt-get install -y --no-install-recommends gcc
COPY myapp/ .
COPY setup.py .
RUN python setup.py install
```

243MB in size

Multi-Stage Builds

```
FROM python:3.7-slim
RUN apt-get update
RUN apt-get install -y --no-install-recommends gcc
COPY myapp/ .
COPY setup.py .
RUN python setup.py install

RUN apt-get remove -y gcc
RUN apt-get -y autoremove
```

This is 245MB because of how caching works

Multi-Stage Builds

```
FROM python:3.7-slim

COPY myapp/ .
COPY setup.py .

RUN apt-get update && \
apt-get install -y --no-install-recommends gcc && \
python setup.py install && \
apt-get remove -y gcc && apt-get -y autoremove
```

This is 161MB but doing it causes slow build on code change bc gcc is installed. You can do ‘docker build –squash‘ but it destroys cache layers

Multi-Stage Builds, C example

```
FROM ubuntu:18.04 AS compile-image
RUN apt-get update
RUN apt-get install -y --no-install-recommends gcc
    ↪ build-essential

WORKDIR /root
COPY hello.c .
RUN gcc -o helloworld hello.c

FROM ubuntu:18.04 AS runtime-image
COPY --from=compile-image /root/helloworld .
CMD ["./helloworld"]
```

88.9MB basically the same size as ‘ubuntu:18.04‘

Multi-Stage Builds, Python

- Python differs bc some packages need compiler
- Most packages come in Wheel, a package format that contains compiled and compatible binaries
- ‘python:3.8-slim-buster‘ official image contains gcc compiler
- ‘python:3.8-buster‘ contains extra g++ too(and python2 :-|)
- You don’t really need them
- You can’t do as you did with C, because Python package contains several types of file like:
    - Codes and other files(e.g. .pth files)
    - Data files, mostly put in /usr
    - Scripts that are put in /usr/bin
- It is hard to get all files, and it could face bugs

Multi-Stage Builds pip –user

- ‘pip –user‘ installs all package files in ‘ /.local‘
- If it is pip installable, you just have to copy the whole folder
- Downside is you share your packages with system packages that could result in conflict

Multi-Stage Builds pip –user

```
FROM python:3.7−slim AS compile−image
RUN apt−get update
RUN apt−get install −y −−no−install−recommends build−essential gcc

COPY requirements.txt .
RUN pip  install  −−user −r requirements.txt

COPY setup.py .
COPY myapp/ .
RUN pip  install  −−user .

FROM python:3.7−slim AS build−image
COPY −−from=compile−image /root/.local /root/.local

# Make sure scripts  in  . local  are  usable:
ENV PATH=/root/.local/bin:$PATH
CMD ['myapp']
```

Multi-Stage Builds virtualenv

```
FROM python:3.7-slim AS compile-image
RUN apt-get update
RUN apt-get install -y --no-install-recommends build-essential gcc
RUN python -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY setup.py .
COPY myapp/ .
RUN pip install .

FROM python:3.7-slim AS build-image
COPY --from=compile-image /opt/venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"
CMD ['myapp']
```

Multi-Stage Builds other Methods

- You can build your own wheel in another container, then move it, but not much point in it
- ‘pex‘ and ‘shiv‘ create one executable file containing the Python and its dependencies that can be copied
- ‘platter‘ untar into a virtualenv
- For ‘pex‘, ‘shiv‘ and ‘platter‘ There are corner cases, so not that reliable
- For more complicated cases of moving virtualenvs, first run ‘virtualenv –relocatable my-venv‘ in builder image, it relativized .pth files and fixes scripts(not activate script)

Multi-Stage Builds: Build Cache Oddities

- You want a CI/CD system like Gitlab CI or Circle CI(for github)
- Builds on these are slower
- Bc they always do from-scratch builds
- Bc there no where store cache layers
- If your builds are reproducible, only problem is speed
- In single stage, one can speed it up by pulling the latest image, then build, so there be potential previous layers

Multi-Stage Builds: Build Cache Oddities

```
#!/bin/bash
set -euo pipefail
# Pull the latest version of the image, in order to
# populate the build cache:
docker pull itamarst/helloworld || true
# Build the new version:
docker build -t itamarst/helloworld .
# Push the new version:
docker push itamarst/helloworld
```

Multi-Stage Builds: Build Cache Oddities

```
#!/bin/bash
set −euo pipefail
# Pull the latest version of the image, in order to
# populate the build cache:
docker pull itamarst/helloworld:compile−stage || true
docker pull itamarst/helloworld:latest        || true
# Build the compile stage:
docker build −−target compile−image \
−−cache−from=itamarst/helloworld:compile−stage \
−−tag itamarst/helloworld:compile−stage .
# Build the runtime stage, using cached compile stage:
docker build −−target runtime−image \
−−cache−from=itamarst/helloworld:compile−stage \
−−cache−from=itamarst/helloworld:latest \
−−tag itamarst/helloworld:latest .
# Push the new versions:
docker push itamarst/helloworld:compile−stage
docker push itamarst/helloworld:latest
```

Gunicorn

- There are subtle differences between a container and a regular desktop or VM or etc.
- Gunicorn Hangs for half a minutes:
    - Gunicorn has a heartbeat that checks workers are still alive
    - By default, it is in /tmp
    - In most regular Linux computer or VM, /tmp is stored in RAM
    - In Docker it isn't default behavior
    - That could cause performance problems
    - Gunicorn FAQ: "in AWS an EBS root instance volume may sometimes hang for half a minute and during this time Gunicorn workers may completely block"
    - /dev/shm in docker is by default in RAM, you just need to config for that

```
$ gunicorn --worker-tmp-dir /dev/shm ...
```

Gunicorn

- In using Gunicorn you can have multiple workers, and you should
- Gunicorn doesn't do the best job of distributing load between the worker
- A good advice is to run multiple Gunicorn on different ports and use Nginx or Traefik
- In container orchestration system like Kubernetes, Docker Swarm and etc. has a heartbeat system to replace dead containers
- It is tempting use autoscaling and put a gunicorn worker in each container

Gunicorn

- If you just do one worker, it could be stuck on a slow query and be mistaken for a dead container
- So you should just add a little bit more worker or thread to handle more than one req
- It just reduces the chance especially if it uses CPU

```
$ gunicorn --workers=2 --threads=4 --worker-class=
    ↪ gthread ...
```

Gunicorn

- Docker handles logs for you
- Most Containers don't have a thing like systemd or sysv-init
- Mostly you have only processes that you explicitly run and manage
- You don't have crontab, syslog and etc
- Just log to stdout, and run gunicorn in front

```
$ gunicorn --log-file=- ...
```

- Containers could be different from other systems

Gunicorn

- Docker handles logs for you
- Most Containers don't have a thing like systemd or sysv-init
- Mostly you have only processes that you explicitly run and manage
- You don't have crontab, syslog and etc
- Just log to stdout, and run gunicorn in front

```
$ gunicorn --log-file=- ...
```

Migration on Startup: Not cool

In development it is okay and even preferable. In production, not so much

- In dev, you have one container, in prod, almost always you wanna more than one container, then you have more than one migration

Migration on Startup: Not cool

In development it is okay and even preferable. In production, not so much

- In dev, you have one container, in prod, almost always you wanna more than one container, then you have more than one migration
- Depending on DB, migration type and other factors it might break your DB

Migration on Startup: Not cool

In development it is okay and even preferable. In production, not so much

- In dev, you have one container, in prod, almost always you wanna more than one container, then you have more than one migration
- Depending on DB, migration type and other factors it might break your DB
- Mental coupling of the two ideas is bad in general

Migration on Startup: Not cool

In development it is okay and even preferable. In production, not so much

- In dev, you have one container, in prod, almost always you wanna more than one container, then you have more than one migration
- Depending on DB, migration type and other factors it might break your DB
- Mental coupling of the two ideas is bad in general
- You may need to roll back(Bad if done mental coupling)

Migration on Startup: Not cool

In development it is okay and even preferable. In production, not so much

- In dev, you have one container, in prod, almost always you wanna more than one container, then you have more than one migration
- Depending on DB, migration type and other factors it might break your DB
- Mental coupling of the two ideas is bad in general
- You may need to roll back(Bad if done mental coupling)
- Rolling updates of containers could lead to problem(Mental Coupling, Bad)

Migration on Startup: Not cool

In development it is okay and even preferable. In production, not so much

- In dev, you have one container, in prod, almost always you wanna more than one container, then you have more than one migration
- Depending on DB, migration type and other factors it might break your DB
- Mental coupling of the two ideas is bad in general
- You may need to roll back(Bad if done mental coupling)
- Rolling updates of containers could lead to problem(Mental Coupling, Bad)
- You may wanna do canary deployment

Migration on Startup: Not cool

In development it is okay and even preferable. In production, not so much

- In dev, you have one container, in prod, almost always you wanna more than one container, then you have more than one migration
- Depending on DB, migration type and other factors it might break your DB
- Mental coupling of the two ideas is bad in general
- You may need to roll back(Bad if done mental coupling)
- Rolling updates of containers could lead to problem(Mental Coupling, Bad)
- You may wanna do canary deployment
- Migrate using an explicit command

Zero Down-time Migrations

- If you don't need it, don't bother

Zero Down-time Migrations

- If you don’t need it, don’t bother
- Structure your migration to be additive in short-term

Zero Down-time Migrations

- If you don't need it, don't bother
- Structure your migration to be additive in short-term
- You can keep in sync new columns using db triggers

Zero Down-time Migrations

- If you don't need it, don't bother
- Structure your migration to be additive in short-term
- You can keep in sync new columns using db triggers
    1. Migrate from schema S to S+1 using only additive changes

Zero Down-time Migrations

- If you don't need it, don't bother
- Structure your migration to be additive in short-term
- You can keep in sync new columns using db triggers
    1. Migrate from schema S to S+1 using only additive changes
    2. Upgrade one or more processes to use S+1, name it V+1

Zero Down-time Migrations

- If you don't need it, don't bother
- Structure your migration to be additive in short-term
- You can keep in sync new columns using db triggers
  1. Migrate from schema S to S+1 using only additive changes
  2. Upgrade one or more processes to use S+1, name it V+1
  3. Let it be for some time

Zero Down-time Migrations

- If you don’t need it, don’t bother
- Structure your migration to be additive in short-term
- You can keep in sync new columns using db triggers
    1. Migrate from schema S to S+1 using only additive changes
    2. Upgrade one or more processes to use S+1, name it V+1
    3. Let it be for some time
    4. Upgrade all to V+1

Zero Down-time Migrations

- If you don't need it, don't bother
- Structure your migration to be additive in short-term
- You can keep in sync new columns using db triggers
  1. Migrate from schema S to S+1 using only additive changes
  2. Upgrade one or more processes to use S+1, name it V+1
  3. Let it be for some time
  4. Upgrade all to V+1
  5. Wait until you are sure that you don't need S

Zero Down-time Migrations

- If you don't need it, don't bother
- Structure your migration to be additive in short-term
- You can keep in sync new columns using db triggers
    1. Migrate from schema S to S+1 using only additive changes
    2. Upgrade one or more processes to use S+1, name it V+1
    3. Let it be for some time
    4. Upgrade all to V+1
    5. Wait until you are sure that you don't need S
    6. Run a migration S+2 that destroys anything V+1 doesn't use

Zero Down-time Migrations

- If you don't need it, don't bother
- Structure your migration to be additive in short-term
- You can keep in sync new columns using db triggers
  1. Migrate from schema S to S+1 using only additive changes
  2. Upgrade one or more processes to use S+1, name it V+1
  3. Let it be for some time
  4. Upgrade all to V+1
  5. Wait until you are sure that you don't need S
  6. Run a migration S+2 that destroys anything V+1 doesn't use
- It isn't that simple though, for postgres read here

Identifiable Images

- Tags are mutable(python:3.7 pointed to python:3.7.1 then python:3.7.2)
- Tag in a way that can be recovered by content
- Multiple tags can point to one image
- Add a tag that indicates your git commit hash

```
$ GIT_COMMIT=$(git rev-parse --short HEAD)
$ GIT_BRANCH=$(git rev-parse --abbrev-ref HEAD)
$ echo $GIT_COMMIT $GIT_BRANCH
45eefe3 master
$ docker build -t myimage:commit-$GIT_COMMIT -t
    ↪ myimage:branch-$GIT_BRANCH .
$ docker image ls myimage
REPOSITORY          TAG                 IMAGE ID
    ↪        CREATED             SIZE
myimage             branch-master       f3119f0b1743
    ↪        About an hour ago   202MB
```

Identifiable Images

We can use Docker labels too

```
docker build -t myimage:latest --label git-commit=
    ↪ $GIT_COMMIT .
$ docker inspect myimage | grep "git-commit"
"LABEL git-commit=45eefe3"
"git-commit": "45eefe3",
"git-commit": "45eefe3",
```

Identifiable Images

We could also have access to it in

```
docker build -t myimage:latest --label git-commit=
    ↪ $GIT_COMMIT .
$ docker inspect myimage | grep "git-commit"
"LABEL git-commit=45eefe3"
"git-commit": "45eefe3",
"git-commit": "45eefe3",
```

Identifiable Images

What about logs and APIs? They could have access git commit info

```
FROM centos
ARG git_commit
RUN echo $git_commit > /git-commit.txt
```

In build time:

```
$ docker build -t myimage --build-arg git_commit=
    ↪ $GIT_COMMIT .
$ docker run -ti myimage
[root@6d9d99b56d9b /]# cat /git-commit.txt
45eefe3
```

Debugging the Building of Images

```
$ docker build -t mynewimage .
Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM python:3.8-slim-buster
---> 3d8f801fc3db
Step 2/3 : COPY build.sh .
---> 541b65a7b417
Step 3/3 : RUN ./build.sh
---> Running in 9917e3865f96
Building...
Building some more...
Build failed, see /tmp/builderr024321.log for details
The command '/bin/sh -c ./build.sh' returned a non-
    ↪ zero code: 1
```

Debugging the Building Process

The images are built in layers and each layer is a working containers, that can be accessed.
What about disk storage? CoW
How to see containers that aren't running?

```
$ docker container ls -a
CONTAINER ID   IMAGE          COMMAND
    ↪    CREATED         STATUS
9917e3865f96   541b65a7b417   "/bin/sh -c ./build…."
    ↪    3 minutes ago   Exited (1) 3 minutes ago
```

Debugging the Building Process

```
$ docker container cp 9917:/tmp/builderr024321.log .
$ cat builderr024321.log
Error, missing flux capacitor!
```