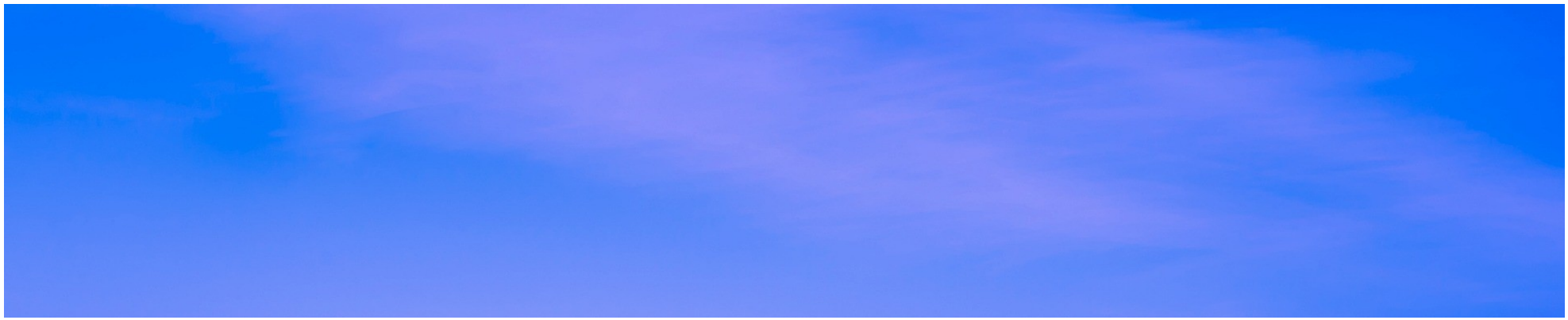
Sign in Get started

Follow 540K Followers · Editors' Picks Features Explore Contribute About

You have **1** free member-only story left this month. Sign up for Medium and get an extra one

Docker Best Practices for Data Scientists

Thushan Ganegedara Feb 6, 2020 · 9 min read ★



Docker ... whale ... you get it.

As a data scientist, I grapple with Docker on a daily basis. Creating images, spinning up contains have become as common as writing Python scripts for me. And this journey has its achievements as well as moments, “I wish I knew that before”.

This article discusses some of the best practices while using Docker for your data science projects. By no means this is an exhaustive checklist. But this covers most things I've come across as a data scientist.

This article assumes basic-to-moderate knowledge of Docker. For example, you should know what Docker is used for and should be able to comfortably write a Dockerfile and understand Docker commands like `RUN` , `CMD` , etc. If not, have a read-through this article from official Docker site. You can also explore through the collection of articles found there.

Why Docker?

Since Docker has been released it has taken the world by a storm. Before the era of Docker, virtual machines used to fill that void. But Docker offers so much than virtual machines.

Advantages of docker

- Isolation — isolated environment regardless of the changes in the underlying OS/infrastructure, installed software, updates
- light-weight — shares the OS kernel avoiding having OS kernel for each container

- performance — being lightweight allows many containers to be run simultaneously on the same OS

Primer on Docker

Docker has three important concepts.

Images — This is a set of runnable libraries and binaries that represents a development/production/testing environment. You can download/create an image in the following ways.

- Pulling from an image registry: e.g. `docker pull alpine` . What happens here is that Docker will look locally in your computer for an image named `alpine` , if it's not found, it looks in Dockerhub
- Building an image locally using a Dockerfile: e.g. `docker build . -t <image_name>:<image_version>` . Here you're not trying to download/pull images, rather, you are building your own image. But this is not entirely true, as a `Dockerfile` contains a line that starts with `FROM <base-image>` which looks for a base image to start with, which might be pulled from Dockerhub.

Containers- This is a running instance of an image. You can stand up a container using the syntax `` `docker container run <arguments> <image> <command> `` , for example to create a container from the `alpine` image use, `docker container run -it alpine /bin/bash` command.

Volumes — Volumes are used to permanently/temporarily store data (e.g. logs, downloaded data) for containers to use. Additionally, volumes can be shared among multiple containers. You can use volumes in couple of ways.

- Creating a volume: You can create a volume using `docker volume create <volume_name>` command. *Note that, information/changes stored here will be lost if that volume is deleted.*
- Bind mount a volume: You can also bind mount an existing volume from the host to your container using `-v <source>:<target>` syntax. For example, if you need to mount the `/my_data` volume to the container as the `/data` volume, you can do, `docker container run -it -v /my_data:/data alpine /bin/bash` command. *The changes you do at the mount point will be reflected on the host.*

1. Creating images

1. Keep the image small, avoid caching

Two common things you'd have to do when building images is,

- Install Linux packages
- Install Python libraries

When installing these packages and libraries the package mangers will cached data so local data will be used if you want to install them again. But this increases the image size unnecessarily. And docker images are supposed to be light-weight as possible.

When installing Linux packages remember to remove any cached data by adding the last line to your `apt-get install` command.

```
RUN apt-get update && apt-get install tini && \
 rm -rf /var/lib/apt/lists/*
```

When installing Python packages, to avoid caching, do the following.

```
RUN pip3 install <library-1> <library-2> --no-cache-dir`
```

2. Separate out Python libraries to a requirements.txt

The last command you saw brings us to the next point. It is better to separate Python libraries to a `requirements.txt` file and install libraries using that file using the following syntax.

```
RUN pip3 install -r requirements.txt --no-cache-dir
```

This gives a nice separation of Dockerfile doing "Docker stuff" and not (explicitly) worrying about "Python stuff". Additionally, if you have multiple Dockerfiles (e.g. for production / development / testing) and they all want the same libraries installed, you can reuse this command easily. The `requirements.txt` file is just a bunch of library names.

```
numpy==1.18.0
scikit-learn==0.20.2
pandas==0.25.0
```

3. Fixing library versions

Note how in the `requirements.txt` I am freezing the version I want to install. This is very important. Because otherwise, every time you build your

Docker image, you might be installing different versions of different things. "Dependency Hell" is real.

Running containers

1. Embrace the non-root user

When you run the containers, if you don't specify an user to run as, it is going to assume `root` user. I'm not going to lie. my naive self used to love having the ability to use `sudo` or being `root` to get things my way (especially to get around permission). But if I've learnt one thing, it's that having unnecessary privileges than needed is an exacerbation catalyst, leading to even more problems.

To run a container as a non-root user, simply do

- `docker run -it -u <user-id>:<group-id> <image-name> <command>`

Or, if you want to jump into an existing container do,

- `docker exec -it -u <user-id>:<group-id> <container-id> <command>`

For example, you can match the user id and group id of the host by assigning `<user-id>` as `$(id -u)` and `<group-id>` as `$(id -g)` .

> *Beware of how different operating systems assign user IDs and group IDs. For example your user ID/group ID on a MacOS might be a* ***pre-assigned/reserved*** *user ID / group ID inside an Ubuntu container.*

2. Creating a non-priviledged user

It is great that we can log in as a non-root user to our host-away from host. But if you login like this, you're a user without a username. Because, obviously the container has no-clue where that user id came from. And you need to remember and type these user id and group id everytime you want

to spin-up a container or `exec` into one. So for that, you can include this user/group creation as a part of the `Dockerfile` .

```
ARG UID=1000
ARG GID=1000
```

- First add `ARG UID=1000` and `ARG GID=1000` to the `Dockerfile` . `UID` and `GID` are environment variables in the container to which you'll pass the value at `docker build` stage (defaults to 1000).
- Then add a Linux group in the image with the group ID `GID` using, `RUN groupadd -g $GID john-group` .
- Next add a Linux user in the image with some user ID `UID` using, `useradd -N -l -u $UID -g john-group -G sudo john` . You can see that here we are adding `john` to the `sudo` group. But this is an optional thing. You can leave it out if you are 100% sure you don't need `sudo` permission.

Then during image build, you can pass values for these arguments like,

- `docker build <build_dir> -t <image>:<image_tag> --build-arg UID=<uid-value> --build-arg GID=<gid-value>`

For example,

- `docker build . -t docker-tut:latest --build-arg UID=$(id -u) --build-arg GID=$(id -g)`

Having a non-privileged user helps you to run processes that should not have root permission. For example, why run your Python script as root when all it does is reading from a dir (e.g. data) and writing to one (e.g. model). And as an added benefit, if you match the user ID and group ID of the host, within the container, all the files you created will have your host user's ownership. So if you bind-mount these files (or create new files) they will still look like you created them on the host.

Creating volumes

1. Separate artifacts using volumes

As a data scientist, obviously you'll be working with various artifacts (e.g. data, models and code). You can have the code in one volume (e.g. `/app`)

and data in another (e.g. `/data`). This will provide a nice structure for your Docker image as well as get rid of any host-level artifact dependencies.

What did I mean by artifact dependencies? Say you have the code at `/home/<user>/code/src` and the data at `/home/<user>/code/data`. If you copy/mount `/home/<user>/code/src` to the volume `/app` and `/home/<user>/code/data` to the volume `/data`. It doesn't matter if the location of the code and data changes on the host. They will always be available at the same location inside the Docker container as long as you mount those artifacts. So you could fix those paths nicely in your Python script as follows.

```
data_dir = "/data"
model_dir = "/models"
src_dir = "/app"
```

You can `COPY` the necessary code and data into the image using

```
COPY test-data /data
COPY test-code /app
```

Note that `test-data` and `test-code` are directories on the host.

2. Bind-mount directories during development

Great thing about bind-mounting is that, whatever you do in the container is reflected on the host itself. This is great when you're doing developments and you want to debug your project. Let's see this through an example.

Say you created your docker image by running:

```
docker build <build-dir> <image-name>:<image-version>
```

Now you can stand up a container from this image using:

```
docker run -it <image-name>:<image-version> -v
/home/<user>/my_code:/code
```

Now you can run the code within the container and debug at the same time and the changes to the code will be reflected on the host. And this loops back to the benefit of using the same host user ID and group ID in your container. All changes you do, looks like came from the user on the host.

3. NEVER bind-mount critical directories of the host

Funny story! I once mounted the home directory of my machine to a Docker container and managed to change the permission of the home directory. No need to say that I was unable to log into the system afterwards and spent a good couple of hours fixing this. Therefore, mount only what is needed.

For example, say you have three directories that you want to mount during developments:

- `/home/<user>/my_data`
- `/home/<user>/my_code`
- `/home/<user>/my_model`

You might be very tempted to mount `/home/<user>` with a single line of code. But it is definitely worth writing three lines to mount these individual sub directories separately, as it will save you several painstaking hours (if not days) of your life.

Additional tips

1. Know the difference between ADD and COPY

You probably know that there are two Docker commands called `ADD` and `COPY` . What's the difference?

- `ADD` can be used to download files from URLs when used like, `ADD <url>`
- `ADD` when given a compressed file (e.g. `tar.gz`) will extract the file to the provided location.
- `COPY` copies a given file/folder to the specified location in the container.

2. Difference between ENTRYPOINT and CMD

A great analogy that comes to my mind is, think of `ENTRYPOINT` as a vehicle and `CMD` as the controls in that vehicle (e.g. accelerator, brakes, steering wheel). `ENTRYPOINT` it self does nothing, it's just a vessel for what you want to do within that container. It just stays stand-by for any incoming commands you push to the container.

A command `CMD` is what actually gets executed within a container. For example `bash` would create a shell in your container so you could work withing the container like you work on a normal terminal on Ubuntu.

3. Copying files to existing containers

Not again! I've created this container and forgot to add this file to the image. It takes so long to build the image. Is there any way I could cheat and add this to the existing container?

Yes there is, you could use `docker cp` command for this. Simply do,

```
docker cp <src> <container>:<dest>
```

Next time you jump into the container you will see the copied file at `<dest>`. But remember to actually change the `Dockerfile` to copy the necessary files at build time.

3. Conclusion

Great! That's all folks. We discussed,

- What Docker images / containers / volumes are?
- How to write a good Dockerfile
- How to spin-up containers as a non-root user
- How to do proper volume mounting in Docker
- And bonus tips such as using `docker cp` to save the day

Now you should have grown your confidence to look at Docker in the eyes and say "*You can't scare me*". Jokes aside, it always pays off to know what you're doing with Docker. Because if you're not careful you can bring down a whole server and disrupt the work of everyone else whose working on that same machine.

Sign up for The Daily Pick

By Towards Data Science

Hands-on real-world examples, research, tutorials, and cutting-edge techniques delivered Monday to Thursday. Make learning your daily ritual. Take a look

Your email

Get this newsletter

By signing up, you will create a Medium account if you don't already have one. Review our Privacy Policy for more information about our privacy practices.

Docker Python Data Science Best Practices Light On Math

About Help Legal