Docker Containers as Unfrozen Environment

The story with containers is as follows. When you are inside a container, your filesystem and environment variables can be isolated totally from a host and vice versa. This is the only essential thing to know about containers, period.

And this fact allows your application to be portable. Thanks to the isolation you can run your code almost on any platform. Obviously, your application must bring all the environment, all dependencies with it. And the carrier of the dependencies is an image of the container.

Physically, the filesystem of the container *is recorded* as a stack of file trees named *layers*. Each layer is overlaid on others in a historical manner, that is, each logically atomic update of the pure filesystem can be stacked into an image — a flat snapshot of the filesystem where all branches of the global file tree coexist together.

Each time you are starting a container, Docker unfreezes an image of the container, overlays a temporary layer for new files and allows you to run processes within its own environment *inside the container*. By “you” we mean the user or her application, by “inside” we mean a view on a filesystem through either a terminal (a keyboard and a display) in the case of the user or through immediate system calls of the kernel in the case of the application.

Dependencies as an Environment of an Application

We talked about logically atomic updates of the filesystem. If you’re familiar with git, you can think of it as git commits when you add something functional to the code but in a more extensive way. Applying to the entire filesystem these functional artefacts are called dependencies. It can be anything without what your code cannot function: drivers, libraries, utilities, secure keys, environment variables.

- Does your code require a GPU? You need hardware drivers, in most cases with CUDA.
- Does your code have to be built? You need a compiler utility like GCC.
- Does your code run on the fly? You need an interpreter like Bash or Python.
- Does your code depend on the other code created by a community of researchers and developers? You need some library, say OpenCV.
- Is your code hosted on a private repository? You have to generate a pair of secure keys and distribute it between the container and the code hosting.
- Will your code run on different environments, say testing/staging/production, or will it work with different datasets? You have to define and set some environment variables to make your application aware of the mode of running, the platform, paths to endpoints (input and output data), version of the code and so on. So variables are parameters of your code that change its behaviour.

All this stuff is covered by Docker containers if you know how to use them. So, let's play.

Enter a clean container

```
$> docker run -it ubuntu:20.04
#> whoami
root
```

Detach the container (this is very much like tmux or screen, but with a different escape sequence)

```
#> ^Ctrl+P+Q
$> whoami
user
```

Attach the container

```
$> docker attach <container name|digest>
```

The digest is a short hash that identifies the container. A name can do the same

```
$> docker run -it --name pure-env ubuntu:20.04
#> ^Ctrl+P+Q
$> docker container ls
CONTAINER ID   IMAGE          COMMAND   CREATED              STATUS              PORTS     NAMES
5b257e08cac4   ubuntu:20.04   "bash"    About a minute ago   Up About a minute             pure-env
```

But the user is not the only entity you have switched by entering into the container. You may say: "I can switch the user on my host by **`su`** command. What is the difference?" As we said, the difference is in filesystems (and ~~many~~ other things).

Let's compare filesystems:

```
$> ls /
bin        dev        etc        lib        mnt        proc       run        srv        tmp        var
certs      docker.log home       media      opt        root       sbin       sys        usr


$> docker attach pure-env
#> ls /
bin   boot  dev   etc   home  lib   lib32  lib64  libx32  media  mnt   opt   proc  root  run   sbin  srv   sys   tmp
usr  var
```

They look more similar than different but at the same time, they are ***completely*** separated and independent. The filesystems are isolated from each other. This is the most important feature of a container from the perspective of portability and dependency management.

Maybe you noted the **`-it`** parameter. Actually, this unfolds into two parameters: **`-i`** and **`-t`** Together they map your keyboard and your display into a container such that you intercept the control of the terminal inside the container. Do you remember "Everything is a file" in Linux? So, your keyboard from **`/dev/stdin`** and your display "output" from **`/dev/stdout`** of the host OS are mapped into the same files into the container by **`-it`** parameter. Since that moment your

host terminal doesn't listen to you, but your container terminal does. Now the only thing that the host and the container share is terminal i/o, that is, **`stdin`**+**`stdout`** (formally **`stderr`** as well). By "share some entity" we mean this is the same entity from the view of the host and from the view of the container.

Another entity we can share between the host and the container is a directory, a part of the host filesystem.

```
$> for sec in {1..2}; do echo "[$(date)] host" >> dates.log; sleep 1; done

$> cat dates.log
[Mon Oct 11 20:13:23 UTC 2021] host
[Mon Oct 11 20:13:24 UTC 2021] host

$> mkdir dates && mv dates.log dates/
$> docker run -it --name dates -v $PWD/dates:/data ubuntu:20.04
#> cd /data
#> for sec in {1..2}; do echo "[$(date)] container" >> dates.log; sleep 1; done
#> cat dates.log
[Mon Oct 11 20:13:23 UTC 2021] host
[Mon Oct 11 20:13:24 UTC 2021] host
[Mon Oct 11 20:15:13 UTC 2021] container
[Mon Oct 11 20:15:14 UTC 2021] container

#> exit
$> cat /dates/dates.log
[Mon Oct 11 20:13:23 UTC 2021] host
[Mon Oct 11 20:13:24 UTC 2021] host
[Mon Oct 11 20:15:13 UTC 2021] container
[Mon Oct 11 20:15:14 UTC 2021] container
```

We have created a file on the host, then we mapped into the container the directory containing the file by the parameter **`-v`** and its argument **`$PWD/dates:/data`**, where **`$PWD/dates`** is an arbitrary but *absolute* path to the source directory on the host and **`/data`** is the target directory in the container both separated by **`:`**. Such mapping is called mounting. As you can see the file **`dates.log`** within the host directory **`dates/`** is shared with the container during the entire lifetime of the container.

When is mounting useful? You may want to process some dataset and the only way to do it within a container is to mount it. Or you may want to edit your code on the host again and again and run it inside the container. You may ask: "Why do I need to run my processing within a container? Why don't I run my code on the host purely?" You can, but how do you know your code will run on the other host? On the server with an unfamiliar environment?

Building a Docker Image from Dockerfile

The only guarantee to be sure the application will work is to ship the environment alongside your code. And Docker containers are the best known way to run the code everywhere by everyone. More precisely, a Docker image of the container is the very artefact you should ship and distribute to others to make them able to reproduce the behaviour of your code. And the Docker images themself have their own source code, a **`Dockerfile`** consisting of instructions on how to build and run the image. These instructions are short and simple, almost all of them create a separate layer within the image with a fragment of filesystem, and most of them are **`RUN`** instructions that pass shell commands to the temporary container that runs during the construction of each layer of the image.

Any Dockerfile consists of a half dozen instructions: FROM, ENV, WORKDIR, RUN, COPY, ENTRYPOINT.

FROM tells Docker which base image to take as a basis for its filesystem. As you remember

```
$> docker run -it ubuntu:20.04
```

had magic argument **`ubuntu:20.04`** — for your container and your timestamp generator that was a base image containing a standard set of configs and utils for Ubuntu distribution of the Linux. Hence, to give birth to a filesystem the first line of Dockerfile is **`FROM`** almost always. Namely, for example

```
Dockerfile:
FROM ubuntu:20.04
```

where 20.04 is the concrete version of the distribution. It is a best practice to define a version of all you are able to versionize.
`ubuntu:20.04` is not the only image you can use of course. There are plenty of images on hub.docker.com with pre-installed almost all you want. GCC, Python, Anaconda, PyTorch, OpenCV, JupyterLab, PostgreSQL, Node.js — a lot of software that just works. Docker Hub is the hosting for Docker images like GitHub is the hosting for repositories.

If we set a goal to distribute our toy application, the timestamp generator, then we have to wrap its functionality into a script file.

```
timestamper.sh:
for second in {1..60}
do
  date >> /data/dates.log
done
```

Obviously, it will not work on Windows from within Powershell. It will not work even on Linux if you do not have enough permissions to write to the `/` of the filesystem. And even with such simplest examples, we have faced portability problems already. However, Docker solves it.

So, having the **`timestamper.sh`** script in your working directory on the host and the Dockerfile you can inject the script into the next layer of the Docker image with **`COPY`** instruction.

```
Dockerfile:
FROM ubuntu:20.04

COPY timestamper.sh /app/timestamper.sh
```

This instruction works for the image like **`-v`** works for the container.

Now your code is rooted inside the Docker image. But how to run it? Up and run our image, enter the container and execute the script manually? You can, but it is not the purpose we are creating Docker image. In a routine situation, your application has to be running standalone within a container and do its work well without your intervention.

To tell the Docker to run something inside a container there is **`ENTRYPOINT`** instruction.

```
Dockerfile:
FROM ubuntu:20.04

COPY timestamper.sh /app/timestamper.sh

ENTRYPOINT [“/app/timestamper.sh”]
```

Now Docker knows where to call your script from.

But there is a design bug in the application. Timestamper writes its output into the directory that does not exist within the Docker image, into **`/data/`**. And we have to create the directory for Timestamper before since it can’t do it by design.

To make any internal changes to the image filesystem there is instruction **`RUN`**. It is followed by shell commands like in a regular terminal.

```
Dockerfile:
FROM ubuntu:20.04

RUN mkdir /data/

COPY timestamper.sh /app/timestamper.sh
```

```
ENTRYPOINT [“/app/timestamper.sh”]
```

or

```
Dockerfile:
FROM ubuntu:20.04

WORKDIR /
RUN mkdir data/

COPY timestamper.sh /app/timestamper.sh

ENTRYPOINT [“/app/timestamper.sh”]
```

Instructions WORKDIR, RUN, COPY can be repeated multiple times, FROM and ENTRYPOINT can not.

Now we have the working Dockerfile. Let’s build it into a Docker image.

```
$> docker build -t timestamper:v0.1.0 -f Dockerfile .
```

`-t` names the image, **`-f`** points to the file with build instructions (Dockerfile by default), **`.`** points where the files needed for the image are placed (current direrctory by convension, but not by default).

Check that the image has been built.

```
$> docker images
```

And run our application.

```
$> docker run -it timestamper:v0.1.0
```

Debugging Dockerfiles

It will not output anything, but intercept control and finish in 1 minute. Looks like a black hole. How to make sure we did alright and the application works? We need to debug our container and replace an entry point with an interactive shell, run the application manually and perform a sanity check.

```
$> docker run -it --entrypoint bash timestamper:v0.1.0
#> /app/timestamper.sh &
#> tail -f /data/dates.log
```

Here should be time stamps.

Another way to debug the container and get the correct output is to read the logs of the container. In this case you don't need interactive mode of the container and it is recommended to run the container in detached mode

But of course your application have to output something.

```
$> docker run -d -it --name timestamper timestamper:v0.1.0
$> docker logs -f timestamper
```

Okay, all work but useless. Let's remember we work with data all the time and the main product of the application is data. And according to our design, the data generated into **`/data/`** directory. So let's do some data mining of the container and copy it to the host. You can do it two ways. Manually and not. Manual way.

```
$> docker run -it --name timestamper --entrypoint bash timestamper:v0.1.0
#> /app/timestamper.sh &
#> ^Ctrl+P+Q
$> sleep 60 && docker cp timestamper:/data/dates.log .
```

We copied our data from the container. But it is not parameterizable, not scalable way.

Remembering the script of **`timestamper.sh`** one may note that 60 seconds is a constant that a good software engineer has to get rid of and move to the environment parameters.

Besides environment variables there is another way to parameterize your application: command line arguments of the entry point to the container. To pass the arguments into the entry point there is **`CMD`** instruction.

```
Dockerfile:
FROM ubuntu:20.04

WORKDIR /
RUN mkdir data/

COPY timestamper.sh /app/timestamper.sh

ENTRYPOINT ["/app/timestamper.sh"]
CMD ["60"]
```