

Learning to use Docker in Void Linux

Neville Jackson

26 July 2022

1 Introduction

A reasonable explanation of what docker is and how it works is given in this tutorial [2]. In one sentence docker is like a VM , but it only containerizes a selected part of user space. It was originally intended to be used for providing an isolated environment for software development. Other uses have developed.

Two key concepts are images and containers. A docker image is the software which is containerised. There is a registry of useful images called *Docker Hub* [4]. A docker container is a copy of an image, along with some organization which allows it to be moved, installed, modified and run.

It has been pointed out to me that the last sentence , above, is *Not entirely correct or at least confusing*

It has been pointed out to me that the last sentence , above, is *Not entirely correct or at least confusing*. I thank Akito Kitsune for that. I have inserted Akito's remarks on containers below, because they are very clear and will counter any confusion that I may have generated.

"The Docker Image is the unborn Docker container. It is declaratively designed through a Dockerfile. It does not have a "running" state, as it is more like a cake, which never goes bad, but also isn't baked, yet. Creating a container of an image is like taking that cake and actually baking it. You do something with it.

A container has a "running" state, because you do something in it. For example, you run software inside that container.

While images and containers are different in that regard, it's usually best to keep the container as close as possible to the initial image design by Dockerfile, because this way you keep the designing possibilities on the image, i.e. Dockerfile level.

In other words, the Dockerfile is the declarative "recipe" and the image is the result of that recipe in perfect form. Then, this perfect form is taken and used in real life, i.e. in a container."

I must say the above quote inspired me. I have moved from a dull 'shipping container' analogy, to something that makes the Docker Image come alive.

2 Install steps

Void has several docker-related packages

I chose initially to install just *docker* and see what dependencies it dragged in

```
# xbps-install docker
```

Name	Action	Version	New version	Download size
docker-cli	install	-	20.10.12_1	9979KB
runc	install	-	1.1.3_1	3024KB
containerd	install	-	1.5.7_1	46MB
moby	install	-	20.10.9_2	23MB
tini	install	-	0.19.0_1	299KB
docker	install	-	20.10.12_1	545B

```
Size to download:          83MB
```

```
Size required on disk:     253MB
```

```
Space available on disk:   263GB
```

```
Do you want to continue? [Y/n] y
```

```
.....
```

```
6 downloaded, 6 installed, 0 updated, 6 configured, 0 removed.
```

```
# exit
```

I then ran a test

```
# docker info
```

```
lient:
```

```
Context:    default
```

```
Debug Mode: false
```

```
Server:
```

```
ERROR: Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?
```

```
errors pretty printing info
```

```
#
```

So I have to start the docker daemon - there are two of them. Unlike most distros, Void does not start daemons automatically when a package is installed. In the runit init system daemons are started by making a filesystem link

```
# ln -s /etc/sv/containerd /var/service/containerd
```

```
# ln -s /etc/sv/docker /var/service/docker
```

Then test again

```
# docker info
Client:
  Context:    default
  Debug Mode: false

Server:
  Containers: 0
   Running: 0
   Paused: 0
   Stopped: 0
  Images: 0
  Server Version: 20.10.9
  Storage Driver: overlay2
   Backing Filesystem: extfs
   Supports d_type: true
   Native Overlay Diff: true
   userxattr: false
  Logging Driver: json-file
  Cgroup Driver: cgroupfs
  Cgroup Version: 1
  Plugins:
   Volume: local
   Network: bridge host ipvlan macvlan null overlay
   Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
  Swarm: inactive
  Runtimes: io.containerd.runc.v2 io.containerd.runtime.v1.linux runc
  Default Runtime: runc
  Init Binary: docker-init
  containerd version: UNSET
  runc version:
  init version:
  Security Options:
   seccomp
   Profile: default
  Kernel Version: 5.15.45_1
  Operating System: Void Linux
  OSType: linux
  Architecture: x86_64
  CPUs: 12
  Total Memory: 62.79GiB
  Name: trinity
  ID: 47TV:VTCL:OLKZ:4N3A:UPAV:MCEJ:CTRJ:HQOD:XVHC:CVJO:RUMU:YZOW
  Docker Root Dir: /var/lib/docker
  Debug Mode: false
  Registry: https://index.docker.io/v1/
  Labels:
```

```
Experimental: false
Insecure Registries:
  127.0.0.0/8
Live Restore Enabled: false
#
```

That looks OK now . There are 0 containers running , that is correct.
Now lets run a simple test container

```
# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:53f1bbee2f52c39e41682ee1d388285290c5c8a76cc92b42687eecf38e0af3f0
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:
<https://hub.docker.com/>

For more examples and ideas, visit:
<https://docs.docker.com/get-started/>

```
#
```

Well, at least it gives me some info. If I now repeat the *docker info* command

```
# docker info
Client:
 Context:    default
 Debug Mode: false
```

```
Server:
```

```
Containers: 1
  Running: 0
  Paused: 0
  Stopped: 1
Images: 1
Server Version: 20.10.9
.....
```

So there is now 1 container present, it is stopped, and there is one image. I don't see any files in my home directory, so where has docker put things? It seems docker stores files in `/var/lib/docker`

```
#ls -F /var/lib/docker
buildkit/  image/    overlay2/  runtimes/  tmp/      volumes/
containers/ network/  plugins/   swarm/     trust/
#
```

All directories. The `overlay2` directory contains the image of the hello-world container.

We don't really want to keep the hello-world container so let's delete it

```
# docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
b6ed07e4a0d03a6930092864e7201e3cce7740153d42a24e4f221427792368ab
```

```
Total reclaimed space: 0B
#
```

We use *prune* because it is a stopped container. *docker info* now reports 0 containers, but still 1 image? How do we remove the image?

```
# docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
hello-world   latest    feb5d9fea6a5   9 months ago   13.3kB
# docker image rm hello-world
Untagged: hello-world:latest
Untagged: hello-world@sha256:53f1bbee2f52c39e41682ee1d388285290c5c8a76cc92b42687eecf38e0af31
Deleted: sha256:feb5d9fea6a5e9606aa995e879d862b825965ba48de054caab5ef356dc6b3412
Deleted: sha256:e07ee1baac5fae6a26f30cabfe54a36d3402f96afda318fe0a96cec4ca393359
#
```

docker info now reports 0 images.

3 Docker Desktop

We have been using the command line interface (CLI) to docker, which is the `docker-cli` package. There is a Docker Desktop available from the docker

website [3] as a .deb.or .rpm file. Void cannot use .deb.or .rpm files, and there does not seem to be a Void Docker Desktop package, so we are stuck with the CLI in Void.

The Docker Desktop install webpage [3] says that in addition to the .deb or .rpm package, Docker Desktop requires KVM support, and QEMO, and Gnome or KDE DTE. My Void installation has none of those installed. The KVM and QEMU packages exist

```
xbps-query -Rs KVM
[-] aqemu-0.9.4_1          GUI to QEMU and KVM emulators, writte...
[-] barrier-2.4.0_1       Open-source KVM software based on Syn...
[-] barrier-gui-2.4.0_1   Open-source KVM software based on Syn...
[-] docker-machine-driver-kvm-0.10.1_1 KVM driver for docker-machine
[-] docker-machine-driver-kvm2-1.24.0_1 Minikube-maintained KVM driver for do...
[-] virtme-0.1.1_4       Easy way to test your kernel changes .
```

```
xbps-query -Rs qemu
[-] aqemu-0.9.4_1          GUI to QEMU and KVM emulators, written in Qt4
[-] novaboot-20191023_2   Tool that automates booting of operating systems...
[-] qemu-7.0.0_1          Open Source Processor Emulator
[-] qemu-ga-7.0.0_1       QEMU Guest Agent
[-] qemu-user-static-7.0.0_1 QEMU User-mode emulators (statically compiled)
[-] qemuconf-0.2.1_3      Simple qemu launcher with config file support
[-] virtme-0.1.1_4       Easy way to test your kernel changes in qemu/kvm
```

but none of these are starred, which would indicate installed. So *docker* is able to run without these requirements, but not *docker desktop*.

4 An interactive container

Lets try and run a container which we can interact with. The docker run command has options which setup an interactive shell. To see them use the -help option as follows

```
$ docker run --help
```

```
Usage:  docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Run a command in a new container

Options:

```
.....
  -i, --interactive          Keep STDIN open even if not attached
.....
  -t, --tty                  Allocate a pseudo-TTY
.....
```

I have only shown the needed options. The *-i* keeps STDIN open, the *-t* assigns a pseudo-tty device to the container.

There is a docker image on Docker Hub called *ubuntu*. We can get the ubuntu image and run it in a container

```
# docker run --name my_ubuntu_container -it ubuntu /bin/bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
405f018f9d1d: Downloading 3.095MB/30.42MB
405f018f9d1d: Pull complete
Digest: sha256:b6b83d3c331794420340093eb706a6f152d9c1fa51b262d9bf34594887c2c7ac
Status: Downloaded newer image for ubuntu:latest
root@59112813a303:/#
```

Well I get a prompt, and it looks like I am somewhere other than Void Linux. We can check with

```
root@59112813a303:/# cat /usr/lib/os-release
PRETTY_NAME="Ubuntu 22.04 LTS"
NAME="Ubuntu"
VERSION_ID="22.04"
VERSION="22.04 LTS (Jammy Jellyfish)"
VERSION_CODENAME=jammy
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
UBUNTU_CODENAME=jammy
root@59112813a303:/#
```

Yes it looks like I am inside a container running Ubuntu. I am still root. There are no users. There is no DTE, just a command line. That is different from a VM. So what can we see from the Void Linux host system?

```
# docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED          STATUS          PORTS          NAMES
59112813a303   ubuntu     "/bin/bash"             About an hour ago Up 26 seconds   my_ubuntu

# ps -ax
 928 ?          Ss        0:00 runsv docker
 925 ?          Ss        0:00 runsv containerd
 956 ?          Sl        0:00 containerd
 958 ?          Sl        0:00 dockerd
3632 ?          Sl        0:00 /usr/bin/containerd-shim-runc-v2 -namespace moby -id 5
```

All the daemons plus one process.

Now lets stop the container

```
# docker stop my_ubuntu_container
my_ubuntu_container
# docker info
Client:
  Context:    default
  Debug Mode: false

Server:
  Containers: 1
    Running: 0
    Paused: 0
    Stopped: 1
  Images: 1

# docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
ubuntu        latest    27941809078c   6 weeks ago    77.8MB
```

Nothing is running, but we still have the ubuntu image.

So copying someone's image and running it in a container is easy. Lets see if we can setup our own container and build some software in it.

5 Build an application image

Assume we have the source code for an application in some folder on our local machine. This may have been written from scratch, or we may have cloned an existing source repo. Just for a trial, I chose to use a simple C program which does some calculations related to development of wool follicles in sheep. I made a new clean directory called Folli.docker, and copied the programs work environment into that directory

```
nevj@trinity Folli.docker$ ls
Makefile  folli.c  folli.h  folli.scr  junk
```

It contains just 2 C program files (folli.c and folli.h), a Makefile, and a script (folli.scr) to do a test run. There is some other irrelevant material hidden away in a subdirectory called junk.

The task is to get this simple work environment into a docker container, with the necessary support software, such as gcc, make, C libraries, editor,... I assume that some form of cutdown Linux will have to be present in the container, to be able to use the above software.

There are two ways to proceed

- Use the Ubuntu container that I already have, and interactively add my little program directory and install necessary support software

- Make a new image and container from scratch using a file called *Dockerfile* which specifies what to put in the new image.

Using a *Dockerfile* seems to be the recommended approach. The *Dockerfile* is placed in the top directory of the work environment which is to be made into an image - ie in my case in `/Folli.docker`. It is just a text file and its name is *Dockerfile*.

The official docker guide to writing Dockerfiles is here [5]. A better explanatory document is here [6] or here [7] or here [8]. After reading all those guides, I am still not clear on some issues. The only way is to start making a Dockerfile and learn from mistakes.

The first thing is to specify the parent image - that is the cutdown Linux that is to support my work environment. In Dockerfile that is done with a `FROM` command. For example

```
FROM ubuntu:18.04
```

so you can specify the version, as well as the distro name.

I would rather use Alpine than Ubuntu, so what I want to tell my Dockerfile to do is to fetch from docker-hub a cut down Alpine image, and add to it my local working directory to make a custom image. That is easy. We write a simple Dockerfile

```
FROM alpine
COPY . .
```

The `COPY` statement says copy everything from current directory (first `.'`) in host system (ie where the Dockerfile is) to `.'` in the custom built image. I also added a *.dockerignore* file to the `/Folli.docker` directory, so that the *junk* directory will not be included in the image.

Then we use these Dockerfile instructions to build a custom image, working from within the `/Folli.docker` directory

```
# docker build .
Sending build context to Docker daemon 15.36kB
Step 1/2 : FROM alpine
latest: Pulling from library/alpine
530afca65e2e: Pull complete
Digest: sha256:7580ece7963bfa863801466c0a488f11c86f85d9988051a9f9c68cb27f6b7872
Status: Downloaded newer image for alpine:latest
--> d7d3d98c851f
Step 2/2 : COPY . .
--> 9fbddcffedb9
Successfully built 9fbddcffedb9
#
```

We can see the custom image we built with

```
# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none>	<none>	9fbddcuffedb9	12 minutes ago	5.54MB
alpine	latest	d7d3d98c851f	5 days ago	5.53MB
ubuntu	latest	27941809078c	6 weeks ago	77.8MB

So it called it *< none >* which is not ideal, it needs a custom name, we will fix that later. The other two images (called 'ubuntu' and 'alpine') are the parent images of those distros, as downloaded from DockerHub.

Now we can run the image *< none >* as an interactive container with

```
# docker run --name my_custom_container -it 9fbddcuffedb9 /bin/ash
/ #
```

Well that is fairly brief. Lets see what is running, from a host system window do

```
# docker stats
```

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O
db6329f387a8	my_custom_container	0.00%	676KiB / 62.79GiB	0.00%	3.01kB / 522B

So a container is running , and we succeeded in giving it a name.. Lets go to the interactive prompt 9ie the container window) and see what it contains

```
/ # pwd
/
/ # ls -aF
./          bin/          home/        root/        usr/
../         dev/          lib/         run/         var/
.dockerenv* etc/          media/       sbin/
.dockerignore folli.c      mnt/         srv/
Dockerfile  folli.h      opt/         sys/
Makefile    folli.scr    proc/        tmp/
/ #
```

So my workfiles are there, but it put it all in the root directory. That is not very nice either.. more things to fix later.

Now can I build my little software collection with make?

```
/ # make
/bin/ash: make: not found
/ #
```

No. There is no *make* command. Not surprising , the Dockerfile did not add make and other build requirements to the alpine image.

Lets see if we can add it interactively (just to learn what is needed)

```
/ # apk add make
(1/1) Installing make (4.3-r0)
```

```

Executing busybox-1.35.0-r15.trigger
OK: 6 MiB in 15 packages
/ # which make
/usr/bin/make
/ #

```

And we had better have a few other build requirements

```

/ # apk add gcc
(1/10) Installing libgcc (11.2.1_git20220219-r2)
(2/10) Installing libstdc++ (11.2.1_git20220219-r2)
(3/10) Installing binutils (2.38-r3)
(4/10) Installing libgomp (11.2.1_git20220219-r2)
(5/10) Installing libatomic (11.2.1_git20220219-r2)
(6/10) Installing gmp (6.2.1-r2)
(7/10) Installing isl22 (0.22-r0)
(8/10) Installing mpfr4 (4.1.0-r0)
(9/10) Installing mpc1 (1.2.1-r0)
(10/10) Installing gcc (11.2.1_git20220219-r2)
Executing busybox-1.35.0-r15.trigger
OK: 109 MiB in 25 packages
/ #

```

That seems to have dragged in all the necessary tools. It has *vi* and *more* , so lets try a compile again.

```

/ # make
cc -v -g -static -c -o folli.o folli.c
Using built-in specs.
COLLECT_GCC=cc
Target: x86_64-alpine-linux-musl
Configured with: /home/buildozer/aports/main/gcc/src/gcc-11.2.1_git20220219/configure --pre
Thread model: posix
Supported LTO compression algorithms: zlib
gcc version 11.2.1 20220219 (Alpine 11.2.1_git20220219)
COLLECT_GCC_OPTIONS='-v' '-g' '-static' '-c' '-o' 'folli.o' '-mtune=generic' '-march=x86-64'
/usr/libexec/gcc/x86_64-alpine-linux-musl/11.2.1/cc1 -quiet -v folli.c -quiet -dumpbase folli.o -o
GNU C17 (Alpine 11.2.1_git20220219) version 11.2.1 20220219 (x86_64-alpine-linux-musl)
compiled by GNU C version 11.2.1 20220219, GMP version 6.2.1, MPFR version 4.1.0, MPC version
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
ignoring nonexistent directory "/usr/local/include"
ignoring nonexistent directory "/usr/lib/gcc/x86_64-alpine-linux-musl/11.2.1/../../../../x86_64-alpine-linux-musl/11.2.1/include"
ignoring nonexistent directory "/usr/include/fortify"
#include "... " search starts here:
#include <...> search starts here:
/usr/include

```

```

/usr/lib/gcc/x86_64-alpine-linux-musl/11.2.1/include
End of search list.
GNU C17 (Alpine 11.2.1_git20220219) version 11.2.1 20220219 (x86_64-alpine-linux-musl)
compiled by GNU C version 11.2.1 20220219, GMP version 6.2.1, MPFR version 4.1.0, MPC version

GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
Compiler executable checksum: 032e78b3e0ace96e0ed58573fd512cc9
folli.c:12:17: fatal error: stdio.h: No such file or directory
    12 | #include      <stdio.h>
        |             ~~~~~~
compilation terminated.
make: *** [<built-in>: folli.o] Error 1
/ #

```

Not quite, we forgot the include files for standard C.

```

/ # apk add libc-dev
(1/2) Installing musl-dev (1.2.3-r0)
(2/2) Installing libc-dev (0.7.2-r3)
OK: 119 MiB in 27 packages
/ #

```

It seem libc-dev is a metapackage that pulls in musl-dev. So Alpine uses musl not glibc. OK, it should still work. Try again

```

/ # make
cc -v -g -static -c -o folli.o folli.c
Using built-in specs.
COLLECT_GCC=cc
Target: x86_64-alpine-linux-musl
Configured with: /home/buildozer/aports/main/gcc/src/gcc-11.2.1_git20220219/configure --pre
Thread model: posix
Supported LTO compression algorithms: zlib
gcc version 11.2.1 20220219 (Alpine 11.2.1_git20220219)
COLLECT_GCC_OPTIONS='-v' '-g' '-static' '-c' '-o' 'folli.o' '-mtune=generic' '-march=x86-64'
/usr/libexec/gcc/x86_64-alpine-linux-musl/11.2.1/cc1 -quiet -v folli.c -quiet -dumpbase fol
GNU C17 (Alpine 11.2.1_git20220219) version 11.2.1 20220219 (x86_64-alpine-linux-musl)
compiled by GNU C version 11.2.1 20220219, GMP version 6.2.1, MPFR version 4.1.0, MPC version

GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
ignoring nonexistent directory "/usr/local/include"
ignoring nonexistent directory "/usr/lib/gcc/x86_64-alpine-linux-musl/11.2.1/../../../../x86_64-alpine-linux-musl/11.2.1/include"
ignoring nonexistent directory "/usr/include/fortify"
#include "...": search starts here:
#include <...>: search starts here:
/usr/include
/usr/lib/gcc/x86_64-alpine-linux-musl/11.2.1/include

```

End of search list.

GNU C17 (Alpine 11.2.1_git20220219) version 11.2.1 20220219 (x86_64-alpine-linux-musl)
compiled by GNU C version 11.2.1 20220219, GMP version 6.2.1, MPFR version 4.1.0, MPC version

GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072

Compiler executable checksum: 032e78b3e0ace96e0ed58573fd512cc9

folli.c:23:1: warning: return type defaults to 'int' [-Wimplicit-int]

```
23 | main(int argc, char *argv[])
    | ~~~~
```

COLLECT_GCC_OPTIONS='-v' '-g' '-static' '-c' '-o' 'folli.o' '-mtune=generic' '-march=x86-64'

/usr/lib/gcc/x86_64-alpine-linux-musl/11.2.1/../../../../x86_64-alpine-linux-musl/bin/as -v

GNU assembler version 2.38 (x86_64-alpine-linux-musl) using BFD version (GNU Binutils) 2.38

COMPILER_PATH=/usr/libexec/gcc/x86_64-alpine-linux-musl/11.2.1/:/usr/libexec/gcc/x86_64-alp

LIBRARY_PATH=/usr/lib/gcc/x86_64-alpine-linux-musl/11.2.1/:/usr/lib/gcc/x86_64-alpine-linux-

COLLECT_GCC_OPTIONS='-v' '-g' '-static' '-c' '-o' 'folli.o' '-mtune=generic' '-march=x86-64'

gcc -o folli folli.o -lm

/ #

/ # ls -aF

./	bin/	folli.o	opt/	sys/
../	dev/	folli.scr	proc/	tmp/
.dockerenv*	etc/	home/	root/	usr/
.dockerignore	folli*	lib/	run/	var/
Dockerfile	folli.c	media/	sbin/	
Makefile	folli.h	mnt/	srv/	

/ #

So it worked. Now lets try and run the binary, which is called folli. We will use
the script folli.scr. Alpine has *sh* shell, so it should work

/ # cat folli.scr

./folli <<eoi

2000000

3.0

4.303449

0.000002017

450.

64

86

1.0e8

35. 33. 30.

eoi

Now run the script

/ # sh -ex folli.scr

+ ./folli

number of primary sites 2000000

```

So/P ratio    3.00
growthrate - slope of log_wt/log_age line  4.30345
growth intercept - of log_wt/log_age line 0.0000020170
follicle initiation rate - increase per timeincrement per cm sq 450.0000
time of start of primary follicle initiation period      64
time of start of secondary original follicle initiation period      86
number of founder cells at time zero 100000000.0
average number of cells per p,so,sd follicle  35.000   33.000   30.000

```

```

time      foll  pfoll  sofoll  sdfoll diffoundcel  foundcel  wght surfarea folirate celbra
between P and So periods
61         0       0       0       0       0       107054834   97.2   190.3   8563
.....
  out  of founder cells
300 27069651 2171768  6543389 18354494  842578546       0 92229.7 18372.0 82674
  adult S+P density per cm sq =    2572
  adult P density per cm sq =    206
  adult So density per cm sq =    622
  adult Sd density per cm sq =   1744
  adult S density per cm sq =   2365
  adult S/P ratio =    11.5
  p interval = 14
  so interval = 18
/ #

```

So the script runs. Everything is OK, just terribly untidy. So lets quit and cleanup

```

/ # exit
#

```

So we exit back to the Void host system

5.1 Saving containers

Before we get rid of the *< none >* image , I need to see whether the stuff I added interactively while I had the container running was preserved when I killed the running container with

```
# docker container prune
```

So lets run *< none >* again in a new container

```
#docker run --name my_second_container -it 9fbddcffedb9 /bin/ash
```

```

/ # ls -aF
./          bin/          home/          root/          usr/

```

```

../          dev/          lib/          run/          var/
.dockerenv*  etc/          media/        sbin/
.dockerignore folli.c      mnt/          srv/
Dockerfile   folli.h      opt/          sys/
Makefile     folli.scr    proc/         tmp/
/ # which make
/ #

```

So , no , the stuff I added is not saved in the *< none >* image. It contains only the stuff put there by Dockerfile. Lesson learnt! Dont prune a container without saving it. How to save a container? This seems likely

```

# docker --help
.....
  commit      Create a new image from a container's changes
.....
# docker commit --help

```

Usage: docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

Create a new image from a container's changes

```

Options:
  -a, --author string      Author (e.g., "John Hannibal Smith
                           <hannibal@a-team.com>")
  -c, --change list        Apply Dockerfile instruction to the created image
  -m, --message string     Commit message
  -p, --pause              Pause container during commit (default true)
#

```

Well, thats all very well, but how do I give it a name? Lets get an example

```
docker commit c3f279d17e0a svendowideit/testimage:version3
```

That came from the docker documentation. It looks like convention is to use a repository name (like nevj/dockerfiletest) and a tag indicating version (like :version1). So lets reinstall the software (make,gcc,libc-dev) and run the make, so I can tell it from the *< none >* image. Then try to save the container

```

# docker ps
CONTAINER ID   IMAGE                COMMAND              CREATED          STATUS              PORTS              NAMES
c8a4532843d8   9fbddcuffedb9       "/bin/ash"          32 minutes ago   Up 32 minutes      my_se

# docker commit c8a4532843d8 nevj/dockerfiletest:version1
sha256:d15df250f7ac209b987c1daf264d773f4954411324989b470119920374589216
#
# docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nevj/dockerfiletest	version1	d15df250f7ac	9 seconds ago	124MB
alpine	latest	d7d3d98c851f	6 days ago	5.53MB
ubuntu	latest	27941809078c	6 weeks ago	77.8MB

#

Yes looks like we have it saved as an image. Now what happens if I exit from the running container?

In the container window
 / # exit
 #

In the host system window

```
# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
# docker images						
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE		
nevj/dockerfiletest	version1	d15df250f7ac	12 minutes ago	124MB		
alpine	latest	d7d3d98c851f	6 days ago	5.53MB		
ubuntu	latest	27941809078c	6 weeks ago	77.8MB		

There is no running container now, but the image is still saved.

```
# docker info
.....
Server:
Containers: 2
Running: 0
Paused: 0
Stopped: 2
Images: 4
```

I still have 2 containers, both are stopped, and there are 4 images although *docker images* only lists 3 images? So I guess we still have an unsaved container, and its image, still hanging around? Will it survive a reboot?

After reboot

```
# docker info
Server:
Containers: 2
Running: 0
Paused: 0
Stopped: 2
Images: 4
```

Nothing has changed. Even stopped containers and their images survive a reboot.

5.2 Big cleanup

How do I get rid of the stopped container without interfering with the saved image nevj/dockerfiletest?

```
# docker rm --help
```

```
Usage:  docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

Remove one or more containers

But I can't remember its name? So I am forced to use prune again

```
# docker container prune --help
```

```
Usage:  docker container prune [OPTIONS]
```

Remove all stopped containers

That should do it

```
# docker container prune
```

WARNING! This will remove all stopped containers.

Are you sure you want to continue? [y/N] y

Deleted Containers:

c8a4532843d8a911c2e43249f8c24ee87a0562bd3114598413707c243bc3ecac
db6329f387a885c012827a0ff80403fd2279a228530f317e375d3b7b2c6bab24

Total reclaimed space: 236.5MB

```
#
```

```
# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nevj/dockerfiletest	version1	d15df250f7ac	37 minutes ago	124MB
alpine	latest	d7d3d98c851f	6 days ago	5.53MB
ubuntu	latest	27941809078c	6 weeks ago	77.8MB

```
#
```

That worked, and the 3 images are still saved, but

```
# docker info
```

```
....
```

Server:

Containers: 0

Running: 0

Paused: 0

Stopped: 0

Images: 4

. The container is gone, but why 4 images, when *docker images* only lists 3 images? Tried *docker image prune* but *docker info* still lists 4 images. I give up. I think docker has lost count!

Now, if I run that saved nevj/dockerfiletest image in another new container, does it contain all the interactive mods?

```
# docker run --name my_third_container -it d15df250f7ac /bin/ash
/ #
```

```
/ # ls -aF
./          bin/          folli.o      opt/          sys/
../         dev/          folli.scr    proc/         tmp/
.dockerenv* etc/          home/        root/         usr/
.dockerignore folli*        lib/         run/          var/
Dockerfile  folli.c       media/       sbin/
Makefile    folli.h       mnt/         srv/
/ # which make
/usr/bin/make
/ #
```

It is all there. So the save was successful. This time lets see if we can kiil the container while it is running

```
# docker kill my_third_container
my_third_container
# docker ps
CONTAINER ID   IMAGE          COMMAND      CREATED      STATUS      PORTS      NAMES
# docker images
REPOSITORY          TAG          IMAGE ID      CREATED      SIZE
nejv/dockerfiletest version1     d15df250f7ac  54 minutes ago  124MB
alpine              latest      d7d3d98c851f  6 days ago    5.53MB
ubuntu              latest      27941809078c  6 weeks ago    77.8MB
#
```

It works, and the prompt returns in the container window. But

```
# docker info
.....
Server:
Containers: 1
  Running: 0
  Paused: 0
  Stopped: 1
Images: 4
```

So kill only stopped the container, it did not remove it. So

```
# docker rm my_third_container
```

```

my_third_container
# docker info
....
Server:
  Containers: 0
    Running: 0
    Paused: 0
    Stopped: 0
  Images: 4

```

Yes *docker rm* works like *docker container prune* but *docker kill* only stops the container.

At last, I think I understand container management.

5.3 An improved Dockerfile

The next step is to do all the work in the Dockerfile, instead of interactively. So lets attempt to rewrite the Dockerfile as follows

```

# Use the latest Alpine parent image
FROM alpine
# set the working directory inside the container
WORKDIR /home/Folli
# copy current project directory to workdir
COPY . /home/Folli
# install support packages
RUN apk add make && \
    apk add gcc && \
    apk add libc-dev && \
    make /home/Folli
# run the test script
CMD ["sh -ex folli.scr"]

```

There is more to it this time than fetching alpine and copying the Folli project directory into the container filesystem. We use RUN to execute the *apk* commands in alpine to install needed packages. We use WORKDIR to define where to put the Folli directory in the alpine filesystem. We use CMD to run the rttest script.

I am not sure about the WORKDIR command. The official documentation [9] on constructing Dockerfiles is hard to follow. The third party doc [7] I found told me this

WORKDIR sets the path where the command, defined with CMD, is to be executed.

but the best overall guide [8] actually explained things better, while this [10] has the best example Dockerfile.

I am now confident about my Dockerfile; the only way to proceed is to use it to build a Docker Image, and learn from mistakes. So here is the build, this time we might try to give it a name, so it doesn't end up as *< none >*

```

# docker build -t nevj/dockerfiletest:version2 .
Sending build context to Docker daemon 15.36kB
Step 1/5 : FROM alpine
---> d7d3d98c851f
Step 2/5 : WORKDIR /home/Folli
---> Running in 7d84b0d79343
Removing intermediate container 7d84b0d79343
---> 4ce5883562a1
Step 3/5 : COPY . /home/Folli
---> 44fe56c74e9f
Step 4/5 : RUN apk add make &&      apk add gcc &&      apk add libc-dev &&      make /home/Folli
---> Running in 238dee4d1bdc
fetch https://dl-cdn.alpinelinux.org/alpine/v3.16/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.16/community/x86_64/APKINDEX.tar.gz
(1/1) Installing make (4.3-r0)
Executing busybox-1.35.0-r15.trigger
OK: 6 MiB in 15 packages
(1/10) Installing libgcc (11.2.1_git20220219-r2)
(2/10) Installing libstdc++ (11.2.1_git20220219-r2)
(3/10) Installing binutils (2.38-r3)
(4/10) Installing libgomp (11.2.1_git20220219-r2)
(5/10) Installing libatomic (11.2.1_git20220219-r2)
(6/10) Installing gmp (6.2.1-r2)
(7/10) Installing isl22 (0.22-r0)
(8/10) Installing mpfr4 (4.1.0-r0)
(9/10) Installing mpc1 (1.2.1-r0)
(10/10) Installing gcc (11.2.1_git20220219-r2)
Executing busybox-1.35.0-r15.trigger
OK: 109 MiB in 25 packages
(1/2) Installing musl-dev (1.2.3-r0)
(2/2) Installing libc-dev (0.7.2-r3)
OK: 119 MiB in 27 packages
make: Nothing to be done for '/home/Folli'.
Removing intermediate container 238dee4d1bdc
---> a022eab277ad
Step 5/5 : CMD ["sh -ex folli.scr"]
---> Running in 3a033ceb3035
Removing intermediate container 3a033ceb3035
---> d90895ba8dc3
Successfully built d90895ba8dc3
Successfully tagged nevj/dockerfiletest:version2
#

```

Seems to have done something, but what is that message about *make: Nothing to be done for '/home/Folli'.* ?

The new image is present

```
# docker images
REPOSITORY          TAG          IMAGE ID          CREATED           SIZE
nevj/dockerfiletest version2     d90895ba8dc3     About a minute ago 124MB
nevj/dockerfiletest version1     d15df250f7ac     25 hours ago      124MB
alpine              latest      d7d3d98c851f     7 days ago        5.53MB
ubuntu              latest      27941809078c     7 weeks ago        77.8MB
#
```

and *docker info* now says there are 8 images? Clearly it is counting something else apart from what we see with *docker images*?

We need to run that new (version2) image and see if things are setup as intended.

```
# docker run --name my_version2_container -it d90895ba8dc3 /bin/ash
/home/Folli # pwd
/home/Folli
/home/Folli # ls -aF
./          .dockerignore Makefile    folli.h
../         Dockerfile   folli.c     folli.scr
/home/Folli #
```

Well, it has setup in /home/folli instead of / , and the base files are present, but where is the stuff made by make? It is not in /home/Folli, and not in / , so I have to presume it did not do the make. That message about make meant something.

Looking at the Dockerfile again, it would seem I have a mistake on the *make* line. The target /home/Folli is wrong, *make* does not have a file argument, it has a TARGET which is defined in the Makefile. The target name is *folli*, so it should read *make folli* or just *make* because that is the default target. Here is the patched up Dockerfile

```
# Use the latest Alpine parent image
FROM alpine
# set the working directory inside the container
WORKDIR /home/Folli
# copy current project directory to workdir
COPY . /home/Folli
# install support packages
RUN apk add make && \
    apk add gcc && \
    apk add libc-dev && \
    make
# run the test script
CMD ["sh -ex folli.scr >&out"]
```

I have also added something to the CMD line, to save the output of the test script.

Now we redo the build using the above Dockerfile

```

# docker build -t nevj/dockerfiletest:version2 .^[D^[[D^C
#
# docker build -t nevj/dockerfiletest:version3 .
Sending build context to Docker daemon 15.36kB
Step 1/5 : FROM alpine
---> d7d3d98c851f
Step 2/5 : WORKDIR /home/Folli
---> Using cache
---> 4ce5883562a1
Step 3/5 : COPY . /home/Folli
---> 53a6850eb717
Step 4/5 : RUN apk add make &&      apk add gcc &&      apk add libc-dev &&      make
---> Running in b6937b87177b
fetch https://dl-cdn.alpinelinux.org/alpine/v3.16/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.16/community/x86_64/APKINDEX.tar.gz
(1/1) Installing make (4.3-r0)
Executing busybox-1.35.0-r15.trigger
OK: 6 MiB in 15 packages
(1/10) Installing libgcc (11.2.1_git20220219-r2)
(2/10) Installing libstdc++ (11.2.1_git20220219-r2)
(3/10) Installing binutils (2.38-r3)
(4/10) Installing libgomp (11.2.1_git20220219-r2)
(5/10) Installing libatomic (11.2.1_git20220219-r2)
(6/10) Installing gmp (6.2.1-r2)
(7/10) Installing isl22 (0.22-r0)
(8/10) Installing mpfr4 (4.1.0-r0)
(9/10) Installing mpc1 (1.2.1-r0)
(10/10) Installing gcc (11.2.1_git20220219-r2)
Executing busybox-1.35.0-r15.trigger
OK: 109 MiB in 25 packages
(1/2) Installing musl-dev (1.2.3-r0)
(2/2) Installing libc-dev (0.7.2-r3)
OK: 119 MiB in 27 packages
cc -v -g -static -c -o folli.o folli.c
Using built-in specs.
COLLECT_GCC=cc
Target: x86_64-alpine-linux-musl
Configured with: /home/buildozer/aports/main/gcc/src/gcc-11.2.1_git20220219/configure --pre
Thread model: posix
Supported LTO compression algorithms: zlib
gcc version 11.2.1 20220219 (Alpine 11.2.1_git20220219)
COLLECT_GCC_OPTIONS='-v' '-g' '-static' '-c' '-o' 'folli.o' '-mtune=generic' '-march=x86-64'
/usr/libexec/gcc/x86_64-alpine-linux-musl/11.2.1/cc1 -quiet -v folli.c -quiet -dumpbase folli.o -
GNU C17 (Alpine 11.2.1_git20220219) version 11.2.1 20220219 (x86_64-alpine-linux-musl)
compiled by GNU C version 11.2.1 20220219, GMP version 6.2.1, MPFR version 4.1.0, MPC version

```

```

GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
ignoring nonexistent directory "/usr/local/include"
ignoring nonexistent directory "/usr/lib/gcc/x86_64-alpine-linux-musl/11.2.1/../../../../x86_64-alpine-linux-musl/include"
ignoring nonexistent directory "/usr/include/fortify"
#include "... " search starts here:
#include <...> search starts here:
/usr/include
/usr/lib/gcc/x86_64-alpine-linux-musl/11.2.1/include
End of search list.
GNU C17 (Alpine 11.2.1_git20220219) version 11.2.1 20220219 (x86_64-alpine-linux-musl)
compiled by GNU C version 11.2.1 20220219, GMP version 6.2.1, MPFR version 4.1.0, MPC version 1.1.0
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
Compiler executable checksum: 032e78b3e0ace96e0ed58573fd512cc9
folli.c:23:1: warning: return type defaults to 'int' [-Wimplicit-int]
    23 | main(int argc, char *argv[])
        | ~~~~
COLLECT_GCC_OPTIONS='-v' '-g' '-static' '-c' '-o' 'folli.o' '-mtune=generic' '-march=x86-64'
/usr/lib/gcc/x86_64-alpine-linux-musl/11.2.1/../../../../x86_64-alpine-linux-musl/bin/as -v
GNU assembler version 2.38 (x86_64-alpine-linux-musl) using BFD version (GNU Binutils) 2.38
COMPILER_PATH=/usr/libexec/gcc/x86_64-alpine-linux-musl/11.2.1/:/usr/libexec/gcc/x86_64-alpine-linux-musl/11.2.1/libexec/:/usr/libexec/gcc/x86_64-alpine-linux-musl/11.2.1/
LIBRARY_PATH=/usr/lib/gcc/x86_64-alpine-linux-musl/11.2.1/:/usr/lib/gcc/x86_64-alpine-linux-musl/11.2.1/lib64/:/usr/lib/gcc/x86_64-alpine-linux-musl/11.2.1/libexec/
COLLECT_GCC_OPTIONS='-v' '-g' '-static' '-c' '-o' 'folli.o' '-mtune=generic' '-march=x86-64'
gcc -o folli folli.o -lm
Removing intermediate container b6937b87177b
---> ba1b247845a7
Step 5/5 : CMD ["sh -ex folli.scr >&out"]
---> Running in bfe533ca7e81
Removing intermediate container bfe533ca7e81
---> 3780c79c60ea
Successfully built 3780c79c60ea
Successfully tagged nevj/dockerfiletest:version3
#

```

Now it looks like *make* has run successfully this time. So we run this Version3 image and check

```

# docker run --name my_version3_container -it nevj/dockerfiletest:version3 /bin/ash
/home/Folli # pwd
/home/Folli
/home/Folli # ls -aF
./          .dockerignore  Makefile      folli.c      folli.o
../         Dockerfile     folli*        folli.h      folli.scr
/home/Folli #

```

OK, the *make* has worked, because folli, and folli.o are present, but there is no file *out* from running the folli.scr script?

So run it interactively to check

```
/home/Folli # sh -ex folli.scr >&out
/home/Folli # ls
Dockerfile  folli      folli.h    folli.scr
Makefile    folli.c    folli.o    out

/home/Folli # head out
+ ./folli
number of primary sites 2000000
So/P ratio 3.00
growthrate - slope of log_wt/log_age line 4.30345
growth intercept - of log_wt/log_age line 0.0000020170
follicle initiation rate - increase per timeincrement per cm sq 450.0000
time of start of primary follicle initiation period 64
time of start of secondary original follicle initiation period 86
number of founder cells at time zero 100000000.0
average number of cells per p,so,sd follicle 35.000 33.000 30.000
```

Yes, the script , and the command to run it are OK in that environment. Must be something wrong with the CMD statement in Dockerfile. It looks like it might need a full path to the executable, so change it to

```
.....
# run the test script
CMD ["sh -ex /home/Folli/folli.scr >&/home/Folli/out"]
```

Then build yet another version

```
# docker build -t nevj/dockerfiletest:version4 .
Sending build context to Docker daemon 15.36kB
Step 1/5 : FROM alpine
.....
Step 5/5 : CMD ["sh -ex /home/Folli/folli.scr >&/home/Folli/out"]
---> Running in bd6df7418d48
Removing intermediate container bd6df7418d48
---> f7acbf180ee
Successfully built f7acbf180ee
Successfully tagged nevj/dockerfiletest:version4
#
```

Then run version 4

```
# docker run --name my_version_4_container -it nevj/dockerfiletest:version4 /bin/ash

/home/Folli # ls -aF
./          .dockerignore Makefile     folli.c      folli.o
../         Dockerfile    folli*      folli.h      folli.scr
/home/Folli #
```


So still no output from running the shellsript. Otherwise OK. I am giving up on the CMD line. The documentation is no help and I have run out of guesses. I don't understand why one uses CMD for this and RUN for all the apt and make commands? One of the tutorial documents seems to indicate you use RUN for building the application, and CMD for running the application. Well I followed that and arrived here. Clearly there is a gap in understanding.

6 Discussion

There is more to learn. Doing it all with the command line is not the problem. The Docker Desktop GUI would not help with understanding how to build a Dockerfile. It might help with managing all the images and containers that one accumulates.

The lessons so far about learning to use docker are

- make sure docker is installed
- start by running a container built from some simple DockerHub parent image such as the hello-world image
- learn how to use docker --help
- learn how to manage images and containers
- run a container you can interact with, and practice doing things inside it.
- start to build your own application, and learn to use a Dockerfile to do this
- read the online documentation... you will be disappointed.

One might get the impression that docker is only for people who build software. While it is certainly used by software builders, like Github, that is not the only possible use.

Containers are a general method of packaging any segment of user space, even something interactive and GUI driven, like a browser. They are not just another package system, docker containers are very portable, they are guaranteed to run in any system with the same architecture and be unaffected by other things going on outside the container. So they can be used to share pieces of one's work environment.

One of the things that is under discussion in the itsFOSS community is to try and improve the way people share things about their linux systems. There has to be something better than text and static pictures. Containers, whether they be docker or something else, may have a role here.

References

- [1] Docker tutorial. URL <https://www.guru99.com/docker-tutorial.html>
- [2] Docker get started URL <https://docs.docker.com/get-started/>
- [3] Docker Desktop URL <https://docs.docker.com/desktop/install/linux-install/>
- [4] Docker Hub URL <https://hub.docker.com/>
- [5] Official Dockerfile document URL https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
- [6] Dockerfile Guide URL <https://medium.com/@BeNitinAgarwal/best-practices-for-working-with-dockerfiles-fb2d22b78186>
- [7] Docker Basics: How to use Dockerfiles URL <https://thenewstack.io/docker-basics-how-to-use-dockerfiles/>
- [8] A Beginners Guide to Understanding and Building Docker Images URL <https://jfrog.com/knowledge-base/a-beginners-guide-to-understanding-and-building-docker-images/>
- [9] Best practices for writing Dockerfiles URL https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
- [10] Creating a Docker Image for your Application URL <https://www.stereolabs.com/docs/docker/creating-your-image/>
- [11] Void Linux Docker Images URL <https://github.com/void-linux/void-docker>
- [12] LibreWolf source code website. URL <https://gitlab.com/librewolf-community/browser/source>
- [13] Waterfox website. URL <https://www.waterfox.net>