Docker - Beginner Biologist 1

Jean-Yves Sgro

05 December 2019 - (last updated: 2019-12-05)

1 Learning objectives

- What is Docker
- Images and Containers
- Running a software withing a container
- Shared directory

In class these exercises will be run onto the classroom iMacs.

However, as best as I can I'll provide Windows hints and instructions when possible, but a basic understanding of line-command under Windows would be more than useful for that (*e.g.* know what is **DOS** for example.)

1.1 Requirements

- Docker will be used from a line-command terminal: **`Terminal`** on a Macintosh in the classroom. A rudimentary knowledge of `bash` command-line is necessary.
- If you are a Windows user: `PowerShell` can be used as a Terminal. However, setting Docker to run on Windows is more involved (not covered in class.)
- **Docker username**: downloads will require a (free) username, therefore registration is necessary in order to follow the tutorial. Go to https://hub.docker.com

(https://hub.docker.com) and use the button “Sign up for Docker Hub” to register.

2 Docker

In the old days a ***docker*** could be a person working on the docks:

> *“Mrs Bryden used to say despairingly at her social gatherings that if your husband worked in the docks, he was a* **docker***. Her husband, Capt. Ron Bryden, ran the ship surveying company of Edwin C. Waters and was a professional and well educated man but the stigma of working in the docks made him to the outside world, a* **docker***. At least so Mrs Bryden believed.* […] *More properly, ‘***docker***’ was a contraction of ‘***dockworker***’* […]”
>
> Martin E. Smith
> London’s Royal Docks in the 1950s: A Memory of the Docks at Work

2.1 Docker: the word

In the computer context the word “**Docker**” may mean multiple things:

1. “**Docker**” is the name of *“the company driving the container movement”* docker.com (https://www.docker.com/)
2. “**Docker**” is the name of the software provided by the Docker company. The software can run on all 3 main platforms (Linux, Windows and Macintosh.)
3. “**Docker file**” is the name of a text file describing an computing environment, with optional software.
4. “**Docker image**” is the name of a (sometimes large) binary file that can be launched into a container.
5. “**Docker container**” is launched from a “**Docker image**.” Each time the container will start in the same, initial state. Its main benefit is to package applications, allowing them to be portable on any system.
6. A “**Docker repository**” is a web site with downloadable “**Docker images**” and optionally the originating “**Docker files**”

The Docker software is based on the Linux operating system and software running within the container should be a Linux-based software, even though the container can run on all 3 platforms.

However, there is a new development that allows Windows-specific software to run in specialized containers within a Windows system running the “Windows Nano Server image”.[1]

2.2 Docker: do I need it ?

Docker is a tool that is designed to benefit both developers and system administrators, making it a part of many DevOps (developers + operations) toolchains.[2]

This definition may not mean much to non-IT computer users… Additionally, most of the tutorials are targeted towards computer and IT personel and often for large deployements.

However, biologists and other scientists can also benefit from Docker to run software that is difficult to install, or only available on the Linux platform.

The goal of this tutorials series is to introduce Docker to biologists focusing on a single software at a time.

2.3 Docker: what is it really?

Here is an "official" definition from Docker[3]:

> Docker is a platform for developers and sysadmins to develop, deploy, and run applications with containers. The use of Linux containers to deploy applications is called *containerization*. Containers are not new, but their use for easily deploying applications is.

This definition completely forgets us, biologists and other scientists as "end-users" of software! However, Docker can provide solutions that are useful to those of us trying to use a software that they have difficulty installing (on Mac/Windows) or that simply is only available for Linux.

Recently I help someone trying to install/compile a software that had other requirements, including the RNA folding algorithm of the Vienna package. After multiple frustrating days of failed software compilations, missing libraries and other incompatibilities and missing dependecies, in the end, Docker provided the only viable solution.

In simple terms, with the help of the Docker software, one creates a *container*: an isolated section of the computer that *contains* everthing needed to run a software.

> *A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.*
>
> ***Comparing Containers and Virtual Machines**: Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware. Containers are more portable and efficient.*[4]

The *container* is created from an *image* itself previously created from a list of requirements (a plain text *Docker file*.) Fortunately there are many images that are pre-made and downloadable from the Docker Hub.

3 Installation

Tutorials will be held in the Biochemistry classroom 201, and Docker has already be installed.

Instruction for installation can be found on the install link.[5] Scroll down the page to find the paragraph “Supported platforms” to find specific MacOS and Windows installers. It may be necessary to create a free username (see Reqirements section above.)

4 Getting started

To get started we need to open a text terminal as detailed below. In class we’ll use a Macintosh.

TASK:

Do one of the following:.

If you are on a Macintosh:

1. Find the `Terminal` icon in the `/Applications/Utilities` directory. Then double-click on the icon and `Terminal` will open.
2. **OR** use the top-right icon that looks like a magnifying glass (*Spotlight Search*,) start typing the word `Terminal` and press return. `Terminal` will open.

If you are on a PC:

1. Find `Power Shell` *e.g.* using Windows search or Cortana. This will open a suitable text-based terminal.

(*Note*: Windows `cmd` does not offer the appropriate commands.)

> *Note* HTML Version only:

If you are following this document in HTML format the code is shown with a colored background:

```
Green background: commands from local computer bash terminal
```

```
Blue background: commands and output when WITHIN a bash container
```

```
Yellow background: commands or output for information. Do not run!
```

4.1 Version check

At the `$` or `>` prompt within the window of `Terminal`, `cmd` or `PowerShell` type `docker --version` to check the version currently installed.

```
docker --version
```

```
Docker version 19.03.5, build 633a0ea
```

> *Note*: If you type the command without `--` you will obtain a longer output.

Below only the first 8 lines are shown:

```
docker version
```

```
Client: Docker Engine - Community
 Version:           19.03.5
 API version:       1.40
 Go version:        go1.12.12
 Git commit:        633a0ea
 Built:             Wed Nov 13 07:22:34 2019
 OS/Arch:           darwin/amd64
 Experimental:      false
```

In the same way the command `docker info` will provide a long list, here are the first 10 lines:

```
docker info
```

```
Client:
 Debug Mode: false

Server:
 Containers: 2
  Running: 0
  Paused: 0
  Stopped: 2
 Images: 10
 Server Version: 19.03.5
```

4.2 Docker login: Required!

Before going further, it is necessary now to login with your Docker Hub ID (See above to obtain one.)

TASK:

Docker login:.

```
docker login
```

```
Login with your Docker ID to push and pull images from Docker Hub.
If you don't have a Docker ID, head over to https://hub.docker.com
to create one.
Username: YOUR_DOCKER_ID_HERE
Password:
Login Succeeded
$
```

Without this step an error message like this will appear:

```
docker:
Got permission denied while trying to connect to the Docker daemon
socket
at unix:///var/run/docker.sock:
Post http://%2Fvar%2Frun%2Fdocker.sock/v1.40/containers/create:
dial unix /var/run/docker.sock: connect: permission denied.
See 'docker run --help'.
```

4.3 Test installation: Hello World

“ `Hello World` ” is usually the first example when testing programming language.[6]

However, since we are going to use Docker, we’ll “pull” (*i.e.* copy automatically) a pre-made Docker image that will run “ `Hello World` ” within a container and display this on our screen.

The command will have 3 components:

- `docker` : invoques the Docker software. All commands will start with `docker` .
- `run` : Docker will run the following named item: a Docker image.
- `hello-world` : name of a Docker image with informations to run a version of the “ `Hello World` ” program.

Since this image is not (yet) on the local computer it will be “pulled” (copied) directly from the docker hub web site: hence the need to register and login as a user.

“Running”" the image will create a temporary container, that in turn will run the “ `Hello World` ” program. When the “ `Hello World` ” program has finished, the container will stop. However, depending on the command structure, the container can remain active for further use as we’ll see later.

TASK:
run commands.

```
docker run hello-world
```

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:451ce787d12369c5df2a32c85e5a03d52cbcef6eb3586dd0307
5f3034f10adcd
Status: Downloaded newer image for hello-world:latest

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

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 Dock
er Hub.
    (amd64)
 3. The Docker daemon created a new container from that image whic
h runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, w
hich sent it
    to your terminal.
[ . . . ]
```

If you now run the command again:

```
docker run hello-world
```

```
Hello from Docker!
This message shows that your installation appears to be working co
rrectly.
...
```

You will note that this runs immediately without the need to “pull” or download anything.

This also means that the Docker Image is now installed (copied) onto the local computer disk. In order to list the images that are present we can use `docker image ls` where `ls` is a listing sub-command.

```
docker image ls
```

```
REPOSITORY        TAG         IMAGE ID        CREATED          SIZE
hello-world       latest      fce289e99eb9    8 months ago     1.84
kB
```

In the same way we can list the (running) containers with the command `docker container ls`. However, since the `hello-world` container has finished its job and stopped, we need to add the `--all` option:

```
docker container ls --all
```

```
CONTAINER ID    IMAGE        COMMAND    CREATED           STATUS
3a3fd809ce14    hello-world  "/hello"   20 minutes ago    Exited
(0) 20 seconds ago

NAME
trusting_fermi
```

Note that there are 2 more columns: `PORTS` is empty and pertains to more complex situations (for example a web server running inside the Docker container.) The last column `NAMES` contains an alternate name (here `trusting_fermi`) that is made-up each time from a dictionary of words, therefore you will have a different name. This can be useful later to *e.g.* delete the container permanently. This is simply easier for humans, as the name of the container is that more difficult to remember or type under the column `CONTAINER ID`.

5 Review

So far we have tested if `docker` is functional and ran a quick first test with `hello-world`. During this process we have already encountered some key elements:

- `docker` is the software invoked for each command
- a Docker `image` is automatically **`pulled`** from the *Docker Hub* if it is not present on the local computer.
- We can list images and container.

Here is a summary of commands already seen and additional commands (from[7])

```
## List Docker CLI commands
docker
docker container --help

## Display Docker version and info
docker --version
docker version
docker info

## Execute Docker image
docker run hello-world

## List Docker images
docker image ls

## List Docker containers (running, all, all in quiet mode)
docker container ls
docker container ls --all
docker container ls -aq
```

6 Docker for (biologist) *end-user*

For now we simply want to use Docker as an *end-user*. In other words we just want to use software that has been implemented as a Docker image by someone else and is available most likely at the *Docker Hub* web site.

Also, for now we will only deal at first with text-only software that does not require graphical interface.

Later, in a more advanced tutorial, we'll learn how to create our own images.

6.1 Container: 3 access modes

The Docker experience can be very confusing at first as there are many concepts and ideas to think about.

Docker will create a container to run software that is first listed, like a cooking receipe, on a *Docker File*, a plain text file. This file is later converted into an *Image*, that can be stored locally or uploaded (and later downloaded again on another computer) on the *Docker Hub*. The image can be as small as a few Kb, but can be as big as multiple Gb.

As end-user we will at first only use existing images that will be pulled from the *Docker Hub*. These will be stored on the local computer once they are "*pulled*."

The image will run and spawn a container once the `docker run some-image` command is applied to the *`some-image`* image (stored or pulled.)

There are then three main ways that we can **use the software inside the container**, (and that in turn might also depend on how the image was created in the first place.)

> 1. A software inside the container is called to some action, and then the container stops.
> 2. A command is given so that we now see “inside” the container. Anything created in the container will be lost when the container terminates.
> 3. The `docker run` command is modified to *share a folder* with a drive on the local computer. All files created or modified in that directory will be saved on the drive on the local computer when the container ends.

To illustrate this we’ll use a very small Linux distribution called *Alpine*.

6.2 Alpine: a small Linux distribution

Since most of the use and need of Docker will be to run software within a Linux environment (even if you are on a MacOS or Windows computer) we’ll try a few examples within Alpine[8] which provides a Linux environment in a small 8Mb container

6.2.1 Pull Alpine

First let’s get Alpine locally. The official Docker Hub page is https://hub.docker.com/_/alpine (https://hub.docker.com/_/alpine)

All hub pages show a `pull` command on the right hand side.

```
docker pull alpine
```

You can then verify that the image exists:

TASK:

get Alpine on your computer.

```
docker image ls alpine
```

```
REPOSITORY    TAG          IMAGE ID       CREATED          SIZE
alpine        latest       11cd0b38bc3c   14 months ago    4.41MB
```

We’ll now explore the 3 modes described above, one at a time.

6.2.2 Run a command from container and stop

In a Linux system the command `echo` will type the words following the command back on the screen, just like a mountain echo will repeat what you have just said or sung!

While the `echo` command actually also exists on a Mac Terminal, the one that will actually be running will be the one ***inside*** the Alpine container.

We’ll have the words “hi there” typed on the screen. Note that in this case the quotes are actually not mandatory, but here for better clarity.

TASK:
run commands.

```
docker run alpine echo 'hi there!'
```

```
hi there!
```

(*Note*: Single quotes above work. Double quote would give an error.)

To better convince ourselves that it is the container that is creating the output, we’ll as a more “personal” question to Alpine in 2 forms: namely to tell us it’s name and versions with commands `cat /etc/os-release` which will type on the screen the content of the requested file, and `uname -a` which will provide a more generic answer. These will be ***specific*** to the Alpine version that is installed and run *within* the container.

```
docker run alpine cat /etc/os-release
```

```
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.10.3
PRETTY_NAME="Alpine Linux v3.10"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
```

```
docker run alpine cat /etc/os-release
```

```
Linux bb7428667939 4.9.184-linuxkit #1 SMP Tue Jul 2 22:58:16 UTC
2019 x86_64 Linux
```

We can also see a list of the now terminated container. Again the `--all` makes it show containers that are no longer running.

```
docker container ls -all
```

```
CONTAINER ID  IMAGE   COMMAND     CREATED        STATUS
NAMES
39ce8e7bf126  alpine  "uname -a"  4 seconds ago  Exited (0) 2 seco
nds ago   stoic_snyder
```

(Note: Column *PORTS* not shown.)

6.2.3 Run command inside container

One easy command to request is to stay within the container and explore it's "**inside**."

To accomplish this we can simply ask to run the shell command from *within* the container by requesting `/bin/sh` .

However, this time we need to be "*interactive*" and we need to add in either form either `-it` , `-ti` or `-t -i` to specify that we want an interactive (`-i`) terminal (`-t` .)

TASK:
Run the following commands:

This will provide us with a Terminal looking within the container.

We'll get a `#` prompt meaning that we have root (admin) privileges.

```
docker run -it alpine /bin/sh
```

```
/ #
```

We are now executing command within the Linux Alpine terminal shell session!

We can try a few commands. This one will tell us what is our username:

```
/ # whoami
```

```
root
```

This one lists the directories seen from `/` .

```
/ # ls
```

```
bin    etc    lib    mnt    root   sbin   sys    usr
dev    home   media  proc   run    srv    tmp    var
```

As an exercise we’ll go inside the directory called `home` and create a simple text file.

Note: to exit the typing mode, it is necessary to issue a keyboard control `CTRL + D`

```
/ # cd home
/home # cat > Sample.txt
This text will exist inside the container.
But will not be saved when we exit.
[NEED CONTROL + D HERE  TO EXIT TYPING]
/home #
```

Now we can verify that the file exists:

```
/home # ls
```

```
Sample.txt
```

```
/home # cat Sample.txt
```

```
This text will exist inside the container.
But will not be saved when we exit.
/home #
```

However, when we exit the container the file will be lost and not saved.

Let’s exit now:

```
exit
```

Restarting the image will only create a new, blank container in the same state as we did the first time: *i.e.* there was no `Sample.txt` file present.

```
docker run -it alpine /bin/sh
```

You can try `ls home` and verify that the file is no longer there.

This is one way that Docker allows to "start fresh" each time the container is created from the image.

6.2.4 ADVANCED: idle containers

> In fact this is not completely true!

This is an advanced feature right here in a Beginner's tutorial, but I think it will help clarify some things and indeed could prove very useful for recovery.

When the image is activated again with the `docker run` command, it will create a ***new*** container each time. When that container ends its job, it is in fact not destroyed but still "lives" inside the computer in an idle form. We can list them with the command. We are interested in the top 2: the one we just restarted, and the one *before* onto which we created the file.

To make it simpler, it is best to note the names in the `NAMES` column. These will be different for each one of you, but in any case is easier for a human brain than the `CONTAINER ID` name.

```
docker container ls -a
```

```
CONTAINER ID  IMAGE   COMMAND   CREATED          STATUS
NAMES
c43b479b62de  alpine  "/bin/sh" 16 minutes ago  Exited (0) 14 minu
tes ago   nostalgic_chatterjee
2b33dfa50f02  alpine  "/bin/sh" 31 minutes ago  Up 12 minutes
kind_rhodes
```

Indeed if you made multiple attempts at the `docker run` commands you may have more than 2. All you need to remember is which one was used when you created the `Sample.txt` file. In my case it was `kind_rhodes` (or `2b33dfa50f02` .)

This is how to "revive" the dormant docker:

- Step 1: start the container by name:

```
docker start kind_rhodes
```

```
kind_rhodes
```

- Step 2: run it as interactive terminal (`-it`) and attach a process to execute (`exec`) within it, here a shell (`sh`):

```
docker exec -it kind_rhodes sh
```

```
/ #
```

This will have “revived” the original container.

- Step 3: verify that the `Sample.txt` file is there:

```
/home # ls
```

```
Sample.txt
```

```
/home # cat Sample.txt
```

```
This text will exist inside the container.
But will not be saved when we exit.
/home #
```

In the next section we’ll see how we can share a folder between the container and the local computer to avoid loosing files and for an overall better access to data to and from within the container.

6.2.5 Preventing idle containers

> **`--rm`**

There is a simple way to avoid having a heap pile of stopped containers by adding **`--rm`** to remove (delete) the container once it stops running. For example:

```
docker run -it --rm alpine /bin/sh
```

Note that this command uses a double dash `--` .

Therefore, the container started above would be deleted when exiting (command: `exit`) and this container would not appear in the list with command `docker container ls -a` (see paragraph above.)

6.2.6 Removing idle containers

If `--rm` is not given when starting the container and it becomes *idle* this can be addressed in 2 steps:

- **stop** the container
- **delete** the container

The container can be named with either the `CONTAINER ID` or its name under `NAMES` found in the list given by `docker container  ls -a` .

Continuing with the `kind_rhodes` example: (change to the name(s) of your idle container(s).)

```
docker stop kind_rhodes
docker rm kind_rhodes
```

6.2.7 Share a directory

This is a third method to using a container.

The best way to provide two-way exchange of files and data between the local computer and the container is to share a directory.

This is accomplished by adding the qualifier `-v` and the path (location) of the directory to be shared. The added command will have the form **`-v /host/directory:/container/directory`** . Therefore we need to know two things:

> - Directory on local machine
> - Directory within container

On the local machine we can use an existing directory, or simply create a new one. For this exercise we'll create a new directory that we could call `dockershare` and for ease of use we can place it onto the Desktop.

On the container it may be useful to explore what is available before makign a decision, as each container will have a different organization within. For Alpine we already know that `/home` is an empty directory when started fresh as we have explored above.

Time for action!

TASK:

Run the following commands:

- **Directory on local machine**:

Within the terminal we'll nagigate to the default users directory a new directory on the local machine:

```
cd
mkdir dockershare
```

The first command (`cd`) takes us back to our user area. Then we move to the Desktop and create a new directory with the `mkdir` ("make directory") command. Of course, you could create that directory also with the Macintosh graphical interface ahead of time.

```
cd dockershare
pwd
```

```
/Users/jsgro/dockershare
```

Of course the path to your directory will reflect YOUR username!

- **Directory within container**:

We'll use the `/home` directory. For any other container it would be useful to explore its content first.

Note: It is important to note that the content of the `/home` directory would be "masked" *i.e.* not available while the folder inside the container is being shared.

We are now ready to open a new container and share the directory with the following command:

```
 docker run -it --rm -v /Users/jsgro/dockershare:/home  alpine /bi
 n/sh
```

Note that this command has `-it` to make it interactive, `--rm` to remove the container when we exit, and `-v` to engage the shared folders. Then we specify the Alpine image and ask for the shell to run within the container.

We are now sharing a directory with the local computer, that we can access via the `/home` directory within the docker session. We can create a simple test file to verify that it will remain after we are done with the docker container:

```
/ # pwd
cd home
cat > Test.txt
This file is created within the container
and should remain within the shared folder
once we are finished.
[NEED CONTROL + D HERE  TO EXIT TYPING]
/home #
```

Now let's exit the container:

```
/home # exit
```

If you now navigate to the the directory on the local machine you should see the file `Test.txt` that was created within the container.

6.2.8 Making the command mode general

In the command above, the fact that we specify a hard-coded (full absolute path) for the local machine directory makes the command ultra-specific in a way that prevents it to be "portable" or general to apply on another computer. In this case it uses our "username" and that means that the command would have to be manually modified to be useful.

Variables to the rescue: rather than specifying the full path, we can use a variable created "on the fly" that would contain the name of the current directory. Under `bash` this is accomplished with `$(pwd)`. In this formula, `pwd` is executed first within the parentheses and provides the name of the current directory. Then the `$` implicitly makes this a substitutable variable in a formula. Therefore the command could be given as follows to use the current directory as the shared directory on the local machine:

```
 docker run -it -v $(pwd):/home  alpine /bin/sh
```

Note: This variable is created "on the fly" but could be created in advance with the `export` command.

Note for *Windows* users (in `PowerShell`): In Windows the variable has to be created first with the Windows command `Get-Location` (that works like `pwd`, which is also available at least in PowerShell.) The other difference is the change from parenthesis `()` to curly brackets `{}`. The Windows cascade of command would therefore be:

```
$loc = Get-Location
docker run -it -v ${loc}:/home alpine:latest /bin/sh
```

7 Background on Docker

An early paper by Boettiger (2014) describes using Docker for reproducible research.

An early article in *Linux Journal* describes the beginnings of Docker: Merkel (2014)

See Reference section below.

8 Summary of commands learned

Command	Comment
`docker --version`	Short output of version
`docker version` or `docker info`	long output info
`docker version | head -8`	only see first 8 lines of long output
`docker login`	Required. Register at docker.com
`docker run hello-world`	First run. Automatically download (pull) `hello-world` image
`docker image ls`	list stored docker images
`docker container --help`	list all commands pertaining to containers
`docker container ls`	list all active containers
`docker container ls --all`	list all containers, same as command below
`docker container ls -a`	list all containers, same as command above
`docker container ls -aq`	list all containers as only CONTAINER_ID
`docker pull alpine`	download `alpine` from docker hub

Command	Comment
`docker image ls alpine`	list only the `alpine` image(s)
`docker run ubuntu echo "hi there!"`	run `echo` command within `ubuntu` container
`docker run alpine cat /etc/os-release`	run `cat` command within `alpine` container
`docker run -it alpine /bin/sh`	run shell within container, until `exit`
`whoami` , `ls` , `cd` , `exit` , etc.	shell commands.
`docker run -it --rm -v $pwd:/home  alpine /bin/sh`	run shell in container, share current directory

REFERENCES

Boettiger, Carl. 2014. "An Introduction to Docker for Reproducible Research, with Examples from the R Environment." *Arxiv*. https://arxiv.org/abs/1410.0846v1 (https://arxiv.org/abs/1410.0846v1).

Merkel, Dirk. 2014. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." *Linux Journal*, March. https://dl.acm.org/citation.cfm?id=2600241 (https://dl.acm.org/citation.cfm?id=2600241).

---

1. https://en.wikipedia.org/wiki/Docker_(software) (https://en.wikipedia.org/wiki/Docker_(software))↩
2. https://opensource.com/resources/what-docker (https://opensource.com/resources/what-docker)↩
3. https://docs.docker.com/get-started/ (https://docs.docker.com/get-started/)↩
4. https://www.docker.com/resources/what-container (https://www.docker.com/resources/what-container)↩
5. https://docs.docker.com/install/ (https://docs.docker.com/install/)↩
6. https://stackoverflow.com/questions/602237/where-does-hello-world-come-from (https://stackoverflow.com/questions/602237/where-does-hello-world-come-from)↩

7. https://docs.docker.com/get-started/ (https://docs.docker.com/get-started/)↩

8. https://alpinelinux.org/about/ (https://alpinelinux.org/about/)↩