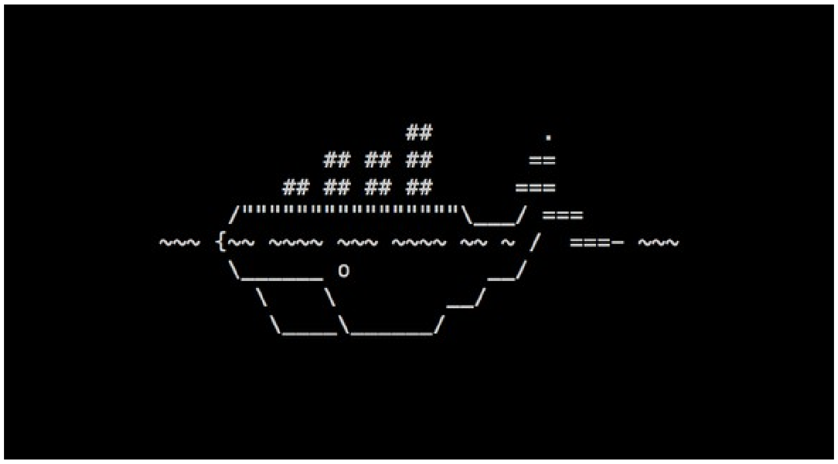
# Docker Tutorial

by

Cédric Bhihe

<cedric.bhihe@gmail.com>



*Rev. 2024.07.03*

# Table of Contents

# 1. Introduction

## 1-1. What is Docker ?

### 1-1-1. History and acceptance

Docker is a well documented container technology which started as a FOSS project, aimed at automating the deployment of applications inside software containers. It was first released in 2013 by Docker, Inc. Its conceptor was Solomon Hykes, the co-founder and CEO of dotCloud[i], a PaaS provider. Several dotCloud engineers also contributed significantly to Docker. Among them were Andrea Luzzardi and Francois-Xavier Bourlet.

Just 3 years after Docker's initial release, major contributors to Docker included Red Hat, IBM, Microsoft, Huawei, Google, and Cisco. In a short time, Docker had caught the attention of some of the largest companies in the world and established itself as the leading software container platform.

Quoting from David Moreto's own tutorial:

*Unlike virtual machines, which get virtual access to host resources through a hypervisor, Docker containers run natively on the host machine's kernel, each running as a discrete process and taking no more memory than any other executable. Docker containers don't run any guest operating system. Instead, they contain only an executable and its package dependencies. This makes containers much less resource demanding and allows containerized applications to run anywhere.*

Docker Enterprise was bought by Mirantis, Inc in 2019. Docker, Inc's fremium product Docker Desktop continues to be free for non corporate desktop-user and is known as a "Personal Plan".

### 1-1-2. High level description,

A Docker container (DC) is a set of restricted namespaces (a process namespace, a filesystem namespace, etc) where processes can run in isolation. Docker implements at its core a separation of applications (and any dependencies needed to run them) from the operating system itself. To make this possible Docker uses *containers* as running instances of *images*. A Docker image is basically a stand-alone, executable template for a filesystem, including code, RTE, libraries, and config files.

When you run a Docker image, an instance of its filesystem becomes live and runs on your system inside a DC. By default this container cannot be modified with respect to either its original image or the filesystem of the host where it currently resides and runs. In this sense it is a self-contained, immutable environment.

Another way to describe a Docker container is as a self-contained bundle of libraries and settings. Together they guarantee that a piece of software will execute correctly, regardless of where it's deployed.

### 1-1-3. What you can expect to learn from this tutorial

At the time the bulk of this tutorial was written, the Docker stable release version was v19.03.14. As of the latest quickstart update the version is 27.0.3.

i DotCloud is a Platform-as-a-Service (PaaS) supplier.

In this tutorial you will learn how to:

- install Docker and run it, pulling images to your host platform,
- make your own Docker images and push them to DockerHub, the Docker registry, or to your own repo,
- physically copy local Docker images to transfer them on to another host,
- understand and make use of Docker's networking capabilities.

Generally speaking this tutorial contains hands-on information and some background on the technology involved. By following it you will gain first-hand experience on how to put Docker to basic use. In a series of online tutorials, Docker offers a more in-depth presentation for devs to get started[i] with docker swarms and orchestrated Docker clusters.

## 1-1-4. Docker's installation pre-requisites

Before installing the Docker Community Edition (*Docker CE*), make sure you comply with prerequisites, namely that:

- your modern linux OS should be based on a 64-bit architecture[ii],

- Docker also depends on the *loop* module, which is a block device that maps its data blocks not to a physical device such as a storage device, but to the blocks of a regular file in a filesystem or to another block device.

For Windows and MacOS environments, consult Docker's official web page. Those will not be described here. Ubuntu is often quoted as the linux distribution of choice for Docker, but this hand-on tutorial is also specialized to Arch Linux. As is the case of a great many Linux distributions, both Ubuntu and Archlinux implement *cgroups* natively[iii] and are based on *systemd*.

If you are uncertain about how different the Archlinux distribution is from Ubuntu, just imaging Archlinux as you would a *lean and mean* version of Ubuntu, i.e. a version of Ubuntu where only packages of YOUR specific interest were installed and configured by you. Incidentally the Archlinux documentation can be dry, but its Wikis are widely recognize as the best in the Linux world.

## 1-4. Cgroups

*A Control Group,* or `cgroup`, constitutes a kernel feature that limits, accounts for and isolates the CPU, memory, disk I/O and network's usage of one or more processes. The cgroups framework provides the following:

- **Resource limiting:** a group can be configured not to exceed a specified memory limit or use more than the desired amount of processors or be limited to specific peripheral devices.
- **Priorization:** one or more groups may be configured to utilize fewer or more CPUs or disk I/O throughput.
- **Accounting:** a group's resource usage is monitored and measured.
- **Control:** groups of processes can be frozen or stopped and restarted.

i https://docs.docker.com/get-started/

ii x86_64 (or amd64), armhf, arm64, s390x (IBM Z), and ppc64le (IBM Power) architectures.
The 3.10.x Linux kernel is the minimum requirement for Docker.

iii Originally developed by Google, the `cgroups` technology eventually found its way to the Linux kernel mainline in version 2.6.24 (Jan-2008). A redesign of this technology—namely the addition of 'kernfs', designed to split some of the sysfs logic, was merged into both the 3.15 and 3.16 kernels. The primary design goal for cgroups was to provide a unified interface to manage processes or whole operating-system-level virtualization, including Linux Containers, or LXC.

A `cgroup` can consist of one or more processes that are all bound to the same set of limits. These groups can be hierarchical too, which means that a subgroup inherits the limits administered to its parent-group.

The Linux kernel provides access to a series of controllers or subsystems for the `cgroup` technology. The controller is responsible for distributing a specific type of system resources to a set of one or more processes. For instance, the memory controller is what limits memory usage while the `cpuacct` controller monitors CPU usage.

You can access and manage cgroups both directly and indirectly with Linux Containers (LXC), libvirt[i] or Docker.

For the curious and technically minded, this article will provide a very clear set of examples to illustrate how `cgroups` help in limiting access to resources by processes. Knowing more about `cgroups` than was was already expounded here, is not strictly necessary to run and use Docker, unless you have a special interest in kernel development and/or in Linux security aspects.

## 1-5. How safe are Docker containers ?

There are four major areas to consider when reviewing Docker security:

- the intrinsic security of the kernel and its support for namespaces and cgroups;
- the attack surface of the Docker daemon itself;
- loopholes in the container configuration profile, either by default, or when customized by users.
- the “hardening” security features of the kernel and how they interact with containers.

More information on each of them is available here.

# 2. Installation of Docker

A permanent installation in your platform is not mandatory, but it will simplify future configuration tasks and may significantly reduce the amount of scripting work, you might need to get specific applications rolling.

This section will also provide you with useful information and hands-on experience on how to put Docker to use.

Before installing the Docker Community Edition (*Docker CE*), make sure you comply with prerequisites.

## 2-1. From an Ubuntu terminal

```
$ alias sag='sudo apt-get'
```

In case older versions of Docker are present, remove them:

```
$ sag remove docker docker-engine
```

Update the list of available packages:

```
$ sag update
$ sag upgrade -y
```

Install extra packages to allow Docker to use the `'aufs'` storage drivers:

```
$ sag install linux-image-extra-$(uname -r) linux-image-extra-virtual
```

Set up Docker’s repositories and install (here, the amd64 compatible version) from them, for ease of installation and future upgrades:

i `libvirt` is a collection of software that provides a convenient way to manage virtual machines and other virtualization functionality, such as storage and network interface management.

```
$ sag install apt-transport-https ca-certificates curl software-properties-common
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
$ sudo apt-key fingerprint 0EBFCD88  # Verify the official GPG key fingerprint
  pub   4096R/0EBFCD88 2017-02-22
  Key fingerprint = 9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
  uid                  Docker Release (CE deb) <docker@docker.com>
  sub   4096R/F273FCD8 2017-02-22
$ sudo add-apt-repository "deb [arch=amd64] \
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

Update the list of available packages:

```
$ sag update; sag upgrade -y
```

Install the latest version of Docker:

```
$ sag install docker-ce
```

Or, install a specific version of Docker, if it suits your needs better:

```
$ sag install docker-ce=<VERSION>
```

Note that Docker can only be ran as *root*.

## 2-2. From an Arch Linux terminal

### 2-2-1. Module `loop' as a Docker pre-requisite

Besides the 64-bit architecture, Docker also depends on the loop module[i], which is a block device that maps its data blocks not to a physical device such as an HD, SSD or optical disk drive, but to the blocks of a regular file in a FS or to another block device.

Docker should enable the loop module automatically during installation.  Check whether the loop module of your Arch linux OS is loaded:

```
$ if [ -z "$(sudo lsmod | grep -e ‘loop’)” ]; then
        # writes “loop” to file loop.conf
        sudo tee /etc/modules-load.d/loop.conf <<< "loop"
        sudo modprobe loop    # adds loop module to kernel
  fi
$ lsmod | grep -e “Module” -e “loop”     # show whether the `loop` module is loaded
```

### 2-2-2. Package installation

You can choose whether you want to install a stable version of Docker from the Community repository (called simply `docker`) or a development version called `docker-git` from the Archlinux User repository (AUR).

If you’re new to using Docker in general or just beginning to use Docker on Arch, install the stable packages:

```
$ sudo pacman -Syu docker docker-buildx
$ pacman -Ss docker               # check available `docker' related Arch repo package
  ……
```

Note that using `` `buildx' `` with Docker requires installation of `` `docker-buildx' `` and the Docker engine version 19.03 or newer.  `` `docker-buildx' `` is a Docker CLI plugin that is included in `Docker Desktop` for Windows, macOS, and

i http://man7.org/linux/man-pages/man4/loop.4.html when installed using the DEB or RPM packages. The `` `buildx' `` plugin replaces the Docker's legacy image builder below

```
$ docker build [OPTIONS] COMMAND
```

with the newer extended build capabilities based on BuildKit:

```
$ docker buildx [OPTIONS] COMMAND
```

If you want to set `` `buildx' `` as the default builder, run the aliasing cmd:

```
$ docker buildx install
```

This enables the legacy Docker builder invoked with `` `$ docker build […]' `` to use the current `` `buildx' `` builder normally invoked using `` `$ docker buildx build' `` command for starting a new build.

To cancel the alias, run:

```
$ docker buildx uninstall
```

The `` `$ docker buildx build' `` command supports features available for `` `$ docker build […]' ``, including features such as outputs configuration, inline build caching, and specifying target platform. In addition `` `buildx' `` also supports new features that are not yet available for regular `` `$ docker build […]' ``, such as building manifest lists, distributed caching, and exporting build results to OCI image tarballs.

`` `buildx' `` is flexible and can be run in different configurations that are exposed through various "drivers". Each driver defines how and where a build should run, and have different feature sets. Supported drivers are:

- `docker` (guide, reference)
- `docker-container` (guide, reference)
- `kubernetes` (guide, reference)
- `remote` (guide)

Instead you may also install the development version of Docker from AUR:

```
$ cd /path/to/your/git_or_aur/builds/directory
# git clone https://aur.archlinux.org/docker-git.git
$ makepkg -sric
```

## 2-2-3. Enabling the Docker daemon

In any case, before you can use Docker, start and enable the Docker daemon with `systemctl`:

```
$ sudo systemctl start docker.service    # start Docker daemon
$ sudo systemctl enable docker.service   # automatically start daemon at startup
```

The first cmd immediately starts the Docker daemon, while the second ensures that the daemon will start automatically at boot.

## 2-2-4. Installation checks

Optionally, use the following command to verify the installation and activation:

```
        # docker
        # docker version
        $ sudo docker info
```

and to run a simple “hello world” test, downloading the corresponding docker image if necessary:

```
        # sudo docker run hello-world
```

If the Docker image file ‘`hello-world`’ is not available on your platform, the last cmd above will tell you so, before proceeding to download it. After download is complete, the corresponding Docker image will be launched and `"Hello from Docker!"` should appear on your screen.

or
```
        # sudo docker run -it --rm archlinux bash -c “echo hello world”
```

The above does the same and automatically removes the container once it has exited.

## 2-3. Post-install configuration

Provided that your host machine is properly configured to begin with, there’s not much left to do after the installation before you can start using Docker on Arch or Ubuntu. Some of those things may involve:

- running Docker as a regular user,
- changing or adding a storage location for Docker images when the amount and storage requirement for images becomes large,
- removing Docker from your host,
- tweaking the storage driver configuration

To visualize and perhaps act upon Docker’s many environment variables, simply issue:

```
        $ sudo systemctl show docker                      # list all environment parameters
or
        $ sudo systemctl show docker --property NRestarts  # to list one specific variable
```

For more post-install configuration options, see Docker’s official documentation as well as the Arch wiki pages.

### 2-3-1. Running Docker as a regular user

Note that Docker can only be ran as `root`. For a regular (non-root) user to run Docker, add it to the `docker` group. Exert caution if you opt for that solution ! Anyone added to the `docker` group becomes root-equivalent. More specifically, every user in the `docker` group has a back-door to overcome any privilege escalation policy and auditing on the system. To avoid this, do not create a `docker` group at all. The upshot is that relevant commands will then require prepending with `sudo`.

***a) On Ubuntu***

```
        $ sudo usermod -aG docker $(whoami)      # to add yourself (as current user)
or      $ sudo usermod -aG docker <username>     # to add an arbitrary user
```

At this point, log out from session and log back in.

***b) On Archlinux***

```
        $ sudo groupadd docker                   # create new group called `docker`
        $ sudo gpasswd -a <username> docker      # add <username> to the `docker` group
```

At this point, log out from session and log back in.

`From now on in this document` we assume that the user is part of the `docker` group. This obviates the need to run any `docker' command by prefixing it with `sudo`.

## 2-3-2. Adapting Docker's storage location to your needs

By default Docker stores images in */var/lib/docker*. To change that location:

### *a) Stop the Docker daemon*

```
$ sudo systemctl stop docker.service
```

### *b) Move the images to the target destination*,

Your target destination is: `/path/to/new/location/docker`

```
$ sudo mkdir -p /path/to/new/location/docker
$ sudo rsync -aqxP /var/lib/docker/ /new/path/docker
$ sudo rm -fr /var/lib/docker
```

Finally, per the official docker documentation[i], you can either:

- either modify the general service configuration file `/lib/systemd/system/docker.service`
- or add directory */etc/systemd/system/docker.service.d* to post-configure the service via drop-in files

### *c-1)* Modification of */lib/systemd/system/docker.service* ← Notice path is `/lib/…`

Change the following parameter in the `[Service]` section:

```
[open text editor]
        ExecStart=/usr/bin/dockerd --data-root=/path/to/new/location/docker -H fd://
[save file]
```

```
$ sudo ps aux | grep -i docker | grep -v grep
$ systemctl daemon-reload
$ sudo systemctl start docker.service
```

### *c-2)* Add directory */etc/systemd/system/docker.service.d*

Create a new file in that directory. The file name must be suffixed with *.conf*. All files with that suffix in the new drop-in directory will be parsed **after** the original configuration file (Sect. 2.3.1) is parsed, allowing you to override the settings of the original config file without having to modify it directly. For instance choose `/etc/systemd/system/docker.service.d/docker-storage.conf`

```
$ sudo mkdir -p /etc/systemd/system/docker.service.d
$ sudo vim /etc/systemd/system/docker.service.d/docker-storage.conf
```

```
[open text editor]
        [Service]
        ExecStart=/usr/bin/dockerd --graph=/path/to/new_volume –storage-driver= devicemapper
[save file]
```

*TODO: verify that highlighted code above corresponds to official doc*

```
$ sudo systemctl daemon-reload
$ sudo systemctl start docker.service
```

i https://docs.docker.com/engine/reference/commandline/dockerd/

Don't forget to change `/path/to/new_volume` to your preferred new storage location and `devicemapper` to your current graph location and storage driver. Note that the storage driver controls how images and containers are stored and managed on your Docker host. You can find out what storage driver is currently used by Docker using the following command:

```
$ docker info | grep -i "storage driver"
Storage Driver: overlay2    # output example
```

`overlay2` is usually considered the best option for modern Docker installations. The job of a storage driver is to store layers of container images efficiently. When several images share a layer, only one layer uses disk space. You may use drivers other than `overlay2`, but a different choice will almost certainly impact performance. The compatible option, `devicemapper` offers suboptimal performance, and is ***outright terrible*** on rotating disks. For that reason alone `devicemapper` is not recommended in production particularly in cloud environment where you cannot always control storage technology details on the fly during dynamic deployment adjustment for instance. You may nevertheless choose to keep it if you are test driving your first DC install.

For users of rolling distributions (e.g. Archlinux, Gentoo, OpenSuse Tumbleweed, Solus, Denian Testing, Void, and derivatives), when host kernels are updated several times a week, there is no point using the compatibility option. A better choice is `` `overlay2` ``.

```
$ sudo docker info | grep –e 'Storage'
```

If the above does not reveal that `` `overlay2` `` is already the default driver, edit `/etc/docker/daemon.json`, before re-starting the `docker.service`:

```
$ sudo systemctl stop docker.service
$ sudo vim /etc/docker/daemon.json
{
  "storage-driver": "overlay2"
}
$ sudo systemctl restart docker.service
```

More info on configuring the Docker daemon is available in the Docker's documentation.

### 2-3-3. Removing Docker from your platform

To uninstall Docker, do:

```
$ sudo apt-get remove --purge docker-ce    # on Ubuntu
```

or

```
$ sudo pacman -Rs docker                   # on Archlinux
```

On Archlinux the Docker configuration files will be saved with a '.pacsave' suffix

If the default path for docker image files was not altered:

```
$ sudo rm -rf /var/lib/docker
```

# 3. Getting started with Docker

## 3-1. Docker's command syntax

Understanding the structure of the cmd line in Docker is paramount. You already ran:

```
$ docker           # view all available sub-cmds
$ docker version   # give running versions for client and server, storage driver, etc.
$ docker info      # view system-wide information
```

Using `docker` consists in passing options to it, then a cmd followed by args. The syntax is:

```
$ docker [opts] [cmd] [args]
```

For instance you may want to try:

```
$ docker run hello-world
  ...[output here]...
```

To view the switches available to a specific sub-cmd, type:

```
$ docker <docker-sub-cmd> --help
$ docker build --help           # to learn on how to use ”build” to build images
Usage:    docker build [OPTIONS] PATH | URL | -
Build an image from a Dockerfile
Options:
[…]
```

Note that if you have aliased the `` `docker-buildx' `` plugin invocation per Section 2.2.2 so:

```
$ docker build …
```

actually calls

```
$ docker buildx build …
```

and accordingly:

```
$ docker build –help
```

will display output for `` `docker buildx build [OPTIONS] PATH |  URL | -' ``.

## 3-2. Working with Docker

Docker containers are run from Docker images. This sub-section is a composite-rewrite of several sources:

- *Digital Ocean*'s now dated how-to[i] for Ubuntu,
- the up-to-date Arch Linux wiki and
- David Morelo's short but flawless blog post.

By default, Docker pulls images from its repo, DockerHub, a registry and storage area managed by Docker. Anybody can build and host their Docker images on DockerHub. Most applications and Linux distros you'll need to run with Docker containers already have images hosted on DockerHub. If not and your environment requirements are ultra-specific to your application, you can build your own images and create containers from it.

For instance you may download an image for the x86_64 Arch Linux *base*:

```
$ docker pull archlinux/base
[...]
```

You may also want to crawl DockerHub and return a listing of all images whose names match the search string `<image_name>`.

```
$ docker search <image_name>      # for any other image name on DockerHub
[...]
```

i https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-16-04

In the OFFICIAL column of the output, “OK” indicates an image built and supported by the company behind the project. Identify the image you need, and download it to your computer using the `pull` sub-cmd.

```
$ docker pull <image_name>
```

After downloading an image of interest, run a container using that image using the `run` sub-cmd.
If an image has not been downloaded when docker is executed with the `run` sub-cmd, the Docker client will first download the image, then run a container using it. For example:

```
$ docker run ubuntu
```

To see the images that have been downloaded to your computer, type:

```
$ docker images
```

Images that you use to run Docker containers can be modified and used to generate new images, which may then be *pushed* to DockerHub or other Docker registries.

## 3-3. Running a Docker container (DC)

You may have already tested your installation, by running the executable image “`hello-world`”:

```
$ docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.
...[more output here]...
```

In the above, you should have witnessed a DC running (very briefly) and exiting immediately after emitting a message. Leaving a stopped container behind.

Containers can be run in interactive (attached) or non-interactive (detached) mode. To exemplify this, we now run an attached DC using an image of the latest Ubuntu `operating` system.

```
$ docker run -it --rm ubuntu
  Unable to find image 'ubuntu:latest' locally
  latest: Pulling from library/ubuntu
  b6f892c0043b: Downloading  2.85 MB/46.89 MB ...
  55010f332b04: Download complete
  ...[more output here]...
root@d9b100f2f636:/# uname -a
  Linux 2a3747ea8be5 4.4.0-75-generic #96~14.04.1-Ubuntu SMP Thu Apr 20 11:06:30
  UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
root@d9b100f2f636:/# ping 8.8.4.4
…
```

Switches include:

**-i** for interactive shell access
**-t** for pseudo tty allocation; keeps **stdin** open even if the container[i] is not attached (**-a**).
--**rm** for removal of the DC automatically once it exits.

When a DC is run, the terminal prompt changes to reflect the fact that you are working inside the container. Note that the DC id (in this case ’d9b100f2f636’) is now included in the new command prompt.

i https://docs.docker.com/articles/basics/#running-an-interactive-shell

Accessing your data and/or opening a python shell at DC launch can bespecified, provided your data is actually available on disk and the python shell is available in your DC image.

```
$ docker run -it --rm -v /mnt/vol/data:/data:ro ubuntu /usr/bin/python
Python 3.6.9 (default, Dec  8 2021, 21:08:43)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyeddl, os
>>> pyeddl.VERSION
'1.2.0'
>>> os.listdir('/')
…
```

Note that using the `-v` switch above, creates an (optionally) read-only bind mount, in this case of `/mnt/vol/data` (HOST-DIR) onto `/data` (CONTAINER-DIR). Note that the manual (`` `man docker-run` ``) specifies that:

*If you supply a HOST-DIR that is an absolute path, Docker bind-mounts to the path you specify. If you supply a name, Docker creates a named volume by that name. For example, you can specify either /foo or foo for a HOST-DIR value. If you supply the /foo value, Docker creates a bind mount. If you supply the foo specification, Docker creates a named volume.*

You may exit with `CTRL-d` or run any cmd inside the DC provided it is recognized by the resources included in the image. For example, let's update the package database inside the container and install `nodeJS`. There is no need to prefix any cmd with sudo, because you are operating inside the container with root privileges:

```
root@d9b100f2f636:/# apt-get update
root@d9b100f2f636:/# apt-get install -y nodejs
```

To stop an active container from within its environment (i.e. from inside the DC), just exit it:

```
$ root@d9b100f2f636:/# exit          # to stop an active container in interactive mode
```

To re-start a stopped (exited) container identified by its id, attach tty and stdin to it, in interactive mode:

```
        $ docker start -a -i container-id
or      $ docker start -a -i "$(docker ps -q -l)"    # to restart the last exited DC RTE
or      $ docker start -a -i "$(docker ps -aq --filter "status=exited")" # to re-start all exited DCs
```

As already mentioned, one may also run container images in detached (non-interactive) mode, by including the **-d** switch. This allows you to return to your underlying session shell, while the container keeps running:

```
$ docker run -d -it ubuntu
  746fd4aceacffe3a60cc835ce95c97377884d2a96f23ad519c2e40de6fdc395f
$
```

To (re-)attach the DC, do:

```
$ docker ps [-a|l]    # list containers: -a = all (active and inactive), -l = latest created
CONTAINER ID   IMAGE   COMMAND     CREATED     STATUS    PORTS   NAMES
746fd4aceacf   ubuntu  "/bin/bash" 19sec ago   Up 18sec          heuristic_archimedes

$ docker attach --sig-proxy=false 746fd4aceacf # see Docker docs[i] for more info
root@746fd4aceacf:/#
```

i https://docs.docker.com/engine/reference/commandline/attach/#attach-to-and-detach-from-a-running-container

To detach a DC from within its own RTE, simply issue:

```
CTRL-p CTRL-q
```

to recover your usual host shell-prompt, neither killing the container nor any process running inside it.

To stop an active DC running non-interactively (detached mode) from the normal shell prompt, simply issue:

```
         $ docker stop container-id         # container-id = 746fd4aceacf
or       $ docker stop container-name       # container-name = heuristic_archimedes
         $ docker kill $(docker ps -q)      # stop all running containers at once
         $ docker rm [-f] $(docker ps -a -q)# remove all containers (running and stopped) at once,
                                              while the optional flag -f conducts a forceful removal.
```

## 3-4. Monitoring DCs

There are several available options how to collect useful metrics from DCs, whether stopped or running.

```
$ docker ps --format={{.Names}} --all    # list names of all containers
...[output here]...

$ docker system df    # list a number of disk usage metrics for stopped and active DCs
TYPE           TOTAL          ACTIVE         SIZE           RECLAIMABLE
Images         1              1              1.84kB         0B (0%)
Containers     1              0              0B             0B
Local Volumes  0              0              0B             0B
Build Cache    0              0              0B             0B
```

which in the case of very small images is not very useful. Some users are under the misconception (because they were told so) that containers are executable processes, they weigh nothing in themsselves. In reality[i] the `*docker run*` command first creates a writeable container layer over the specified image. The ideal of containers' weightlessness is disproved when issuing:

```
$ docker ps -a --no-trunc --size   # displays all DCs with complete ID hash, approx size
$ sudo du -d 2 -h /var/lib/docker  # displays actual disk usage to match ID hash
```

For either one or all the containers running on the host, one option is the `*docker stats*` command, which gives access to CPU, memory, network and a rough measure of disk utilization:

```
$ docker stats [container ID [container ID [container ID [...]]]]
```

Without any container ID, the above cmd will stat all active containers. Add the **--no-stream** option to get a one-time snapshot of current container resource usage, and **--all** to also include stopped containers:

```
$ docker stats --no-stream --all
```

Apart from `*docker stats*`, you can also use cAdvisor (a container monitoring tool from Google), Prometheus (an open source monitoring system and time series database), or Agentless System Crawler (ASC) (a cloud monitoring tool from IBM with support for containers), among other services.

i docs.docker.com/v1.1/reference/commandline/cli

# 4. Interacting with DCs

## 4-1. How to `ssh` into a DC

To run a command in a container that runs locally, you can use the docker exec command to run a command in a running container. For example:

```
# docker exec -it <my-container> bash
```

However whenever the container runs on a remote host, you might want to resort to `ssh'ing` into that DC.

A working option is to install an SSH server in the images, whose DCs you wish to `` `ssh' `` into. Then run each container from their image while mapping the SSH port to one of the host's ports. This is the solution propose in subsection 4-1-1 below.

Another option, perhaps more observant of good practices, attempts to reduce complexity, dependencies, file sizes, and build times. It requires containerized SSH server used to extend any running container, `my_container,` as shown in subsection 4-1-2 below. The only requirement is that the container comes equipped with *`` `bash' ``*.

### *4-1-1. Build a SSH server into a Docker image*

This solution is available but can and should be avoided in most cases. If you want to know why (there are excellent reasons for that), read this. If you still insist on going down that path, call the corresponding Dockerfile, for instance `"sshd-ubuntu.dockerfile"` and proceed as follows:

```
        $ cat Dockerfiles/sshd-ubuntu.dockerfile

        FROM ubuntu:23.04 as basebuild
        RUN apt-get update && apt-get install -y openssh-server
        RUN mkdir -p -m 755 /var/run/sshd
        USER root
        RUN echo -n 'root:passwd' | chpasswd
        RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin yes/' /etc/ssh/sshd_config
        # SSH login fix. Otherwise user is kicked off after login
        RUN sed -i 's@session\s*required\s*pam_loginuid.so@session optional pam_loginuid.so@g' 
/etc/pam.d/sshd
        RUN grep -qxe "session\s*optional\s*pam_loginuid.so" /etc/pam.d/sshd || sed -i "$ a session 
optional pam_loginuid.so" /etc/pam.d/sshd
        ENV NOTVISIBLE "in users profile"
        RUN echo "export VISIBLE=now" >> /etc/profile
        EXPOSE 22
        CMD ["/usr/sbin/sshd", "-D"]

        $ docker build -t example_ubuntu-w-sshd -f Dockerfiles/sshd-ubuntu.dockerfile .

        $ docker run -d -P --name test_sshd example_ubuntu-w-sshd
```

will:

- `-d`, spawn a detached container from the image (runs container in background and prints its ID).
- `--name`, name it "test_sshd",
- `-P`, publish all exposed container's ports (22 in this case) to random host's ports

```
$ docker port test_sshd 22    # discovers the host's port mapped to the container's port 22
0.0.0.0:32768
[::]:32768

$ ssh root@127.0.0.1 -p 32768
```

### *4-1-2. Extend a running container with another one based on sshd*

An example is provided in https://stackoverflow.com/questions/28134239 and supplemented by this GitHub repo.

```
$ docker run      -d
                  -p 2222:22 \
                  -v /var/run/docker.sock:/var/run/docker.sock \
                  -e CONTAINER=my_container \
                  -e AUTH_MECHANISM=noAuth \
                  jeroenpeeters/docker-ssh
$ ssh -p 2222 localhost
```

## 4-2. Sharing data between DCs and localhost (`$HOST`)

You can use Docker volumes to share files between a host system and a DC.  This can prove handy, for example, when you want to create a permanent copy of a log file to analyze it later.

First, create a directory on the host in a location that a Docker user will have access to:

```
$ mkdir ~/container-share
```

Then, attach the host directory to the container volume located in the /data directory within the container:

```
$ docker run -d -P --name test-container -v /home/user/container-share:/data archlinux
```

where used flags are:

| | |
|---|---|
| ***-P*** (`--publish-all`) | all exposed ports are published to random ports |
| **-v** (`--volume dir1:dir2`) | bind-mount a volume |

You will see the ID of the newly created container. Gain shell access to the container:

```
$ docker attach [container ID]
```

Once you've entered the command above, you will be at the data directory we added at container run-time. Any file you add to this directory will be available from the host folder.

# 5 - Creating Docker images

## 5-1. Modifying an existing image

It consists in committing changes in a DC to a file.  It is recommended, if you mean to preserve for future re-use any configuration work made in the DC's environment.

After installing nodeJS inside the previous DC, you now have a running container, whose composition is different from that of the image you used to create it. To save the state of the container as a new image, first exit from it:

```
$ exit
$ docker container ls
CONTAINER ID   IMAGE     COMMAND    CREATED      STATUS              PORTS   NAMES
d9b100f2f636   ubuntu    "bash"      2 days ago   Exited (0) 1 min ag         agitated_turing
……              ……        ……
```

Then commit the changes to a new Docker image with its new `<image_name>` and optional `<tag>`:

```
$ docker commit [-m <commit_message>] [-a <author>] <container_id> <my_repo/image_name>[:<tag>]
```

Specify: **-m** `<commit_message>` it is optional, but helps to remember what changes were made,
**-a** `<author>` it is optional, but helps to remember who made the changes,
`<`**`container_id>`** is the container's identification from which the new image commit will derive.

as noted earlier when starting the interactive docker session. Unless you created additional repositories on Docker Hub, the repository is usually your DockerHub username:

```
$ docker commit -m "added node.js" -a "ckb" d9b100f2f636 ckb/ubuntu-nodejs
sha256:a6f9c14d6ce51033b7798011b5e442848ed287b51d7e93332cb2716f7f58710d
```

When you *commit* an image, the new image is saved ***locally***. In absence of a `<tag>`, the image name tag defaults to ":latest". Later, we will push an image to a Docker registry, such as DockerHub, so it becomes accessible to other authorized users.

After the commit operation has completed, listing the Docker images now on your computer should show the new image, as well as the old one that it was derived from:

```
$ docker images
  REPOSITORY         TAG        IMAGE ID        CREATED          SIZE
  ckb/ubuntu-nodejs  latest     a6f9c14d6ce5    2 minutes ago    206 MB
  ubuntu             latest     ebcd9d4fca80    24 hours ago     118 MB
  hello-world        latest     48b5124b2768    4 months ago     1.84 kB
```

In the above example, *`ubuntu-nodejs`* is a new local image, built from a container instantiated from the derived from the existing *`ubuntu`* image. The size difference reflects the change as *`nodejs`* got installed. To run a DC using Ubuntu with *`nodejs`* pre-installed, use the new image:

```
$ docker run -it -[q-name <dc_name>] ckb/ubuntu-nodejs
root@7d6db03013ba:/#
```

Note that the `container-id` above is different from before, as you now run the container image under a new parent-PID. Note too that we added the new option ***`'--name'`*** which allows you to choose a name for each DC instance you may want to create (here “dc_name”). That name may be used at your convenience to detach and re-attach DCs, commit changes, stop a DC, as you would for these and more operations, when using the DC id.

To rename a DC, issue:

```
$ docker rename old_name new_name
```

To remove an image:

```
$ docker rmi image_id
```

To remove all images at once:

```
$ docker rmi $(docker images -q)
```

## 5-2. Building an image from a Dockerfile

A Dockerfile is similar in concept to the recipes and manifests found in infrastructure automation (IA) tools like Chef or Puppet, or to the makefiles used at compilation time in various programming languages.

Building images from Dockerfiles is a convenient way to ***automate*** the building of an ***image*** and to include a ***context*** in it. The context is the set of files at the build location in case no Docker file is provided. It is typically specified by `.` to signify "present working directory and recursively parsed sub-tree".

### *5-2-1. Using `docker-build'*

To build an image for platforms identical to that of your host computer just issue:

```
$ docker build . [-t image-name[:image_tag]]  # build Docker image from current context
```

where ***-t*** or --***tag*** is followed by an image name and tag in *`image-name[:image_tag]`* format.

It can however be any directory and sub-tree specified by its path or a URL, e.g. to specify a github repo. To build the DC image by pulling a Dockerfile, issue cmd:

```
$ docker build -f path/to/Dockerfile [-t image-name[:image_tag]] .
```

In the above Docker will read instructions from the Dockerfile, a text document which contains any number of cmds a user could call on cli to manually assemble the same image. Users can also read cli from a Dockerfile located anywhere on the host. When no fully qualified path for Dockerfile is provided, Docker will revert to the default Dockerfile location, which is the root of the context.

The content of the Dockerfile is parsed and run by the Docker daemon, not by the host's shell. It conforms to a set of specific syntactic rules and uses case-insensitive reserved words. The complete documentation on this subject is available at https://docs.docker.com. Section 7 of this Document covers how to write a simple Dockerfile in some details, in order to get you started.

### *5-2-2. Using the `docker-buildx' CLI plugin to extend `docker-build' capabilities*

`docker-buildx` is a Docker CLI plugin for extended build capabilities with BuildKit. It is installed from your OS package repo and supports multiple builder instances that extend sthe more traditional "`docker build…`" instruction with support for multi-platform builds, parallel builds, and more. It can be installed following instructions available on its GitHub project page. To use it, first:

- create an image `moby/buildkit:buildx-stable-1` and raise the corresponding container running a builder, with:

```
$ docker buildx create –user
```

- optionally login your Git Docker registry, with:

```
$ docker login <docker-git-registry>/<repo>
```

- create images like so:

```
$ docker buildx build --no-cache \
        -f <path/to/Dockerfile> \
        -t registry.<git_instance>/<repo>/<image_name>:<image_tag> \
        --load \
        --push \
        .
```

where:

`--no-cache` is optional and prevents the use of the cache when building, effectively building from scratch, meaning that every instruction in the Dockerfile will be executed regardless of whether the command was executed before.

`--load` is optional and instructs `docker-buildx` to load the built image into the local Docker daemon automatically, immediately after the build process completes.

`--push` pushes the built image to a container registry after the build process completes, making the image readily available for deployment and distribution across different environments. To use that option a proper login to the Docker Git registry is required with: `` `$ docker login <docker-git-registry>'. ``

`.` includes the context in $PWD as filetree root and its subtree during the build process.

After this a container raised from the image `moby/buildkit:buildx-stable-1` still runs and should be killed and optionally removed:

```
$ docker container kill <container_id>
$ docker container rm <container_id>
```

## 5-3. Pushing Docker images to a repo

After creating a new container image, you will want to push it to DockerHub or to any other Docker image repo. You must have an account on that repo. To learn how to create your own private Docker repo[i], see the blog post How to set up a private docker registry on Ubuntu.

First create an account on DockerHub by registering on the Docker registry's website. In order to push a container image, you must have previously logged into DockerHub from your local terminal session. You'll be prompted to authenticate:

First authenticate:

```
$ docker login -u <docker-git-registry-username> <docker-git-registry>
Enter password: ____
Login suceeded.
```

***WARNING!*** With this method, your password will be stored unencrypted in `/home/ckb/.docker/config.json`. Configure a credential helper to remove this warning. See `https://docs.docker.com/engine/reference/commandline/login/#credentials-store`. If that facility is available, then login only requires:

```
$ docker login <docker-git-registry>
```

Only then can you push your own image using:

```
$ docker push <docker-git-registry>/<image-name>[:<image_tag>]
[...]
```

i https://www.digitalocean.com/community/tutorials/how-to-set-up-a-private-docker-registry-on-ubuntu-14-04

Pushing refers to a repository, `e.g. docker.io/ckb/ubuntu-nodejs` or to a registry `https://registry.gitlab.<mydomain>/<repo>`. After pushing an image to your GitHub repo's registry, it should be listed on your account's dashboard.

To logout from the Git repo and its Docker registry, simply issue:

```
$ docker logout
```

## 6. Docker networking

Networking with DCs is highly configurable. It involves NAT IPTables, port mapping between the containerized apps and the host network space, virtual interfaces and default networks created upon installation of the Docker engine in the host environement.

### 6-1. Network configuration

By default, Docker creates three networks automatically. Even in the absence of a running container, querying the default network configuration built into Docker yields:

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE

e67410aa7b91        bridge              bridge              local

463c03780b9d        host                host                local

138a9b152b62        none                null                local
```

The ***bridge*** network corresponds to the ***docker0*** network, which is present in all Docker installations.

The ***none*** network doesn't have any access to the ***external*** network, but it can be used for running batch jobs.

The ***host*** network adds a ***container on the host's network stack*** without any isolation between the host machine and the container.

Use the following command to see information about the default bridge network:

```
$ docker network inspect bridge
```

Docker recommends using user-defined bridge networks to control which containers can communicate with each other. It doesn't limit how many new networks users can create using the default networks as templates, and containers can be connected to multiple networks at the same time. Try to create a new bridge network and to inspect it:

```
$ docker network create --driver bridge bridge_new
$ docker network inspect bridge_new
```

To illustrate that, launch a *busybox* (or any other) container connected to the newly created network:

```
$ docker run --network= bridge_new -itd --name=[container ID] busybox
```

Docker comes with 4 built-in network drivers: (i) **bridge**, (ii) **host**, (iii) **macvlan** and (iv) **overlay**.

- "**bridge**" is the default network in which all containers are launched.
- "**host**" lets containers access the host's network stack (i.e. using the exact same url:port network space)
- "**macvlan**" gives DCs a direct access the host interface or sub-interface and permits trunking.
- "**overlay**" allows networks to be built across multiple hosts running Docker (usually in the case of a *Docker swarm cluster*). In this configuration DCs have their own subnet and network addresses and can communicate directly with one another, even when running on physically different hosts.

For this work, the 2 networks of interest are *bridge* and *overlay*. To keep our configuration as simple as possible, we will focus on the bridge-network identified "`bridge`" created automatically when installing Docker. As described on https://securitynik.blogspot.co.uk/, "the Linux bridge interfaces are similar to switches in their function in that they connect different interfaces to the same subnet, and forward traffic based on MAC addresses. As we shall see below, each container connected to a bridge network will have its own virtual interface created on the docker host, and the docker engine will connect all containers in the same network to the same bridge interface, which will allow them to communicate with each other."

Recall that issuing:

```
$ docker network ls
  NETWORK ID          NAME                DRIVER              SCOPE
  a2e715a00b4b        bridge              bridge              local
  939bff610adb        host                host                local
  2874386a03ef        none                null                local
```

The "`none`" network permits adding a container to a container-specific network stack, so that container lacks a network interface.

Beside its default network configuration, Docker allows you to create user-defined networks in addition to the default bridge network. In any case when running docker, use the "-- `network`" flag to specify which network you would like your DC to connect to. By default DCs connect to the bridge network, whose interface is known as "`docker0`".

From the host's environment, issue:

```
$ ifconfig
docker0  Link encap:Ethernet  HWaddr 02:42:95:f7:81:b6
         inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
         inet6 addr: fe80::42:95ff:fef7:81b6/64 Scope:Link
         UP BROADCAST MULTICAST  MTU:1500  Metric:1
         RX packets:14 errors:0 dropped:0 overruns:0 frame:0
         TX packets:37 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:0
         RX bytes:992 (992.0 B)  TX bytes:17411 (17.4 KB)
         […]
```

The "`docker network inspect network-name`" cmd returns information about the named network:

```
$ docker network inspect bridge
```

will reveal the bridge network gateway IP address, its name (default is "`docker0`"), its subnet (e.g. 172.17.0.0/16), its gateway (e.g. 172.17.0.1) and various option settings pertaining to network traffic, such as ICMP[i], masquerading for NAT and more.

```
"com.docker.network.bridge.default_bridge": "true",
"com.docker.network.bridge.enable_icc": "true",
"com.docker.network.bridge.enable_ip_masquerade": "true",
"com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
"com.docker.network.bridge.name": "docker0",
"com.docker.network.driver.mtu": "1500"
```

We provide two examples to illustrate how to access specific networking information:

**Tip 1**: To grab the IP address of a running container:

```
$ docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' \
                          <container-name OR id>
172.17.0.37
```

**Tip 2**: For each running container, the name and corresponding IP address can be listed for use in `/etc/hosts`:

```
#!/usr/bin/env sh
for ID in "$(docker ps -q)"; do
    IP=$(docker inspect \
        --format="{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}" "$ID")
    NAME=$(docker ps | grep "$ID" | awk '{print $NF}')
    printf "%s %s\n" "$IP" "$NAME"
done
```

## 6-2. Docker IPTables[ii]

As a rule you will only need to configure IPTables in *production* environments or when the bridge network for containers is manually set up at container deployment time. Otherwise you will mostly rely on Docker defaults. In such a case, this sub-section will only serve informative purposes. In any case this sub-section should not be considered an attempt to review IPTables. Other online resources[iii] are better suited for this purpose.

Docker uses IPTables to control communication to and from the interfaces and networks it creates. IPTables consist of different tables, two of which are of particular interest to us: `filter` and `nat`.

- `filter` is the security rules table used to allow or deny traffic to IP addresses, networks or interfaces,
- `nat` contains the rules responsible for masking IP addresses or ports. Docker uses Network Address Translation (NAT) to allow containers on bridge networks to communicate with destinations outside the docker host (otherwise routes pointing to the container networks would have to be added in the docker host's network, a potential source of vulnerabilities, in particular when running several containers on the same host).

i ICMP or Internet Control Message Protocol is a transport layer control protocol of the internet protocol suite, used by network devices, including routers, to send error messages and operational information indicating, for example, that a requested service is not available or that a host or router cannot be reached. For instance it is involved in verifying whether ping queries are correctly answered by the queried host. The utility `traceroute' also relies on it. (http://www.networksorcery.com/enp/protocol/icmp.htm)

ii This section is a shortened version of a highly recommended blog post by Nik Alleyne. Its conclusion is quoted in full.

iii https://wiki.archlinux.org/index.php/Iptables#Tables

Note that `iptables` settings are lost across system reboots.

## IPTables: `filter`

Tables in IPTables consist of different *chains* that correspond to different conditions or stages in processing a packet on the docker host. The `filter` table has 3 chains by default:

- chain: INPUT

for packets arriving at the host and destined for the same host,

- chain: OUTPUT

for packets originating on the host destined towards an outside destination,

- chain: FORWARD

for packets entering the host but with a destination outside the host.

Each chain consists of a list of rules, the which dictate some action to be taken on the packet (for example drop or accept the packet) as well as conditions for matching the rule. Rules are processed in sequence up until a match is found, otherwise the default policy of the chain is applied.

It is possible to define custom chains in a table. This is outside the scope of this short tutorial.

To view the currently configured rules and default policies for chains in the `filter` table:

```
$ sudo iptables -t filter -L  # same as 'iptables -L' as the filter table is used by default
```

The output shows that Docker adds two custom chains: DOCKER and DOCKER-ISOLATION, and inserts rules in the FORWARD chain with those 2 new chains as *targets*. The DOCKER-ISOLATION chains contains rules that restrict access between the different container networks. To illustrate that in details, consider the following example where we ran the previous command in verbose mode:

```
        $ sudo iptables -t filter -L -v | grep -e "Chain DOCKER-ISOLATION" -A10
          […]
          Chain DOCKER-ISOLATION (1 references)
  pkts bytes target prot opt in      out              source          destination
    0   0   DROP    all  --  br-e6bc7d6b75f3 docker0  anywhere        anywhere
    0   0   DROP    all  --  docker0 br-e6bc7d6b75f3  anywhere        anywhere
    0   0   DROP    all  --  docker_gwbridge docker0  anywhere        anywhere
    0   0   DROP    all  --  docker0 docker_gwbridge  anywhere        anywhere
    0   0   DROP    all  --  docker_gwbridge br-e6bc7d6b75f3  anywhere       anywhere
    0   0   DROP    all  --  br-e6bc7d6b75f3 docker_gwbridge  anywhere       anywhere
36991 3107K RETURN  all  --  any     any              anywhere        anywhere
```

In the above a number of DROP rules block traffic between various bridge interfaces created by Docker, thus making sure that container networks cannot communicate. At an early development stage (NOT at production or pre-production stages), it is convenient to relax those rules so that all containers in a cluster may talk to one another. In that case the output would simply be:

```
$ sudo iptables -t filter -L -v | grep -e "Chain DOCKER-ISOLATION" -A3
Chain DOCKER-ISOLATION (1 references)
pkts  bytes target  prot opt in      out              source          destination
36991 3107K RETURN  all  --  any     any              anywhere        anywhere
```

***icc=false***

The cmd `docker network create` admits several options. One of them is instrumental in denying or permiting inter-container communication on one single network:

```
com.docker.network.bridge.enable_icc.
```

"icc" stands for inter-container communication. Setting this option to *false* blocks containers on the same network from communicating with each other. In that case Docker adds a drop rule in the FORWARD chain that matches packets which come from the bridge interface associated with the network and are destined for the same interface.

For example, create a new network called "no-icc-network" with:

```
$ docker network create --driver bridge --subnet 192.168.200.0/24 –ip-range   \
192.168.200.0/24 -o "com.docker.network.bridge.enable_icc"="false" no-icc-network
```

That newly created bridge network uses an interface whose name (`br-8e3f0d353353`) you can discover with:

```
$ ifconfig | grep 192.168.200 -B1
  br-8e3f0d353353 Link encap:Ethernet  HWaddr 02:42:c4:6b:f1:40
         inet addr:192.168.200.1  Bcast:0.0.0.0  Mask:255.255.255.0
```

Then check that a new rule was appended to the `FORWARD` chain:

```
$ sudo iptables -t filter -S FORWARD | grep "br-8e3f0d353353"
  -A FORWARD -i br-8e3f0d353353 -o br-8e3f0d353353 -j DROP
```

thereby dropping all packets simultaneously originating at and destined to the network bridge interface: `br-8e3f0d353353`.

Again for early development purposes, we make sure that, when appropriate:

```
"com.docker.network.bridge.enable_icc"="true"
```

In addition, although this may not always be necessary, depending on the particulars of a multi-host cluster configuration, inter-container communication is set to ACCEPT when issuing:

```
$ sudo iptables -P FORWARD ACCEPT
```

This however is not a secure configuration as any compromised Docker container will then gain the ability to also compromise other containers across hosts in the same cluster.

### ***iptables:nat***

NAT allows the host to change the IP address or port of a packet. In this instance, it is used to mask the source IP address of packets coming from docker bridge networks (for example hosts in the 172.17.0.0/24 subnet) destined to the outside world, behind the IP address of the docker host.

The `com.docker.network.bridge.enable_ip_masquerade` option controls this. As in the above it can be passed to `'docker network create'` and defaults to "true" when not specified.

You can see the effect of this command in the nat table of iptables:

```
         $ sudo iptables -t nat -L | grep -e POSTROUTING -A3
           Chain POSTROUTING (policy ACCEPT
target      prot  opt source               destination
MASQUERADE  all   --  172.17.0.0/16        anywhere
MASQUERADE  all   --  10.0.3.0/24          !10.0.3.0/24
```

In the above masquerading is effective everywhere from subnet 172.17.0.0/16, but not for local traffic within subnet 10.0.3.0/24.

## 6-3. DCs' virtual interfaces

Containers connected to the same bridge network (e.g. to the default bridge network with bridge interface "`docker0`") can communicate with each other by their IP address, as revealed by the above cmd's full ouput. Docker does not support automatic service discovery on the default bridge network, so for containers to be able to resolve IP addresses by container name, a user-defined fully specified network should be set up instead of the default one.

Each running container belonging to the same bridge network with bridge network interface "`docker0`" has a virtual interface named "`veth`xxxxxxx", where xxxxxxx is an 7-character alphanumerical chain unique to each container. Virtual interfaces are revealed, e.g. in the presence of 2 running Ubuntu-based containers ("dc1" and "dc2"), by:

```
$ docker pull ubuntu
$ docker run -it --name "dc1" ubuntu
root@a754719db594:/# apt-get install -y ethtool iputils-ping bridge-utils
^p ^q
$ docker run -it --name "dc2" ubuntu
root@976041ec420f :/# apt-get install -y ethtool iputils-ping bridge-utils
^p ^q

$ docker ps -q    # query running containers' ID
  a754719db594
  976041ec420f

$ sudo brctl show docker0
  bridge name     bridge id               STP enabled     interfaces
  docker0         8000.02424488bd75       no              veth2177159
                                                          vethd8e05dd
```

Listening on either one of the two virtual interfaces or on both (e.g. with `tcpdump`) is the same as listening on the bridge network interface. By way of example, imagine pinging the RIPE Network Coordination Center from inside container id `a754719db594`.

```
$ docker exec a754719db594 ping -c 40 80.58.61.254 >& /dev/null &
  [1] 13599

$ sudo tcpdump -i veth2177159 icmp          # listen in from the host, outside the container
  listening on vethc955099, link-type EN10MB (Ethernet), capture size 262144 bytes
  22:55:56.027337 IP 172.17.0.2 > 254.red-80-58-61.staticip.rima-tde.net: ICMP echo request,
id 436, seq 261, length 64
  22:55:56.030896 IP 254.red-80-58-61.staticip.rima-tde.net > 172.17.0.2: ICMP echo reply, id
436, seq 261, length 64
  22:55:57.029147 IP 172.17.0.2 > 254.red-80-58-61.staticip.rima-tde.net: ICMP echo request,
id 436, seq 262, length 64
  22:55:57.030912 IP 254.red-80-58-61.staticip.rima-tde.net > 172.17.0.2: ICMP echo reply, id
436, seq 262, length 64
  …
```

To kill the ping process above, re-attach your tty and stdin to the container, identify the process, isolate its PID and kill it:

```
$ docker attach a754719db594
root@a754719db594:/# kill -9 $(awk '/ping/ {print $1}' <<<"$(ps -e)")
^p ^q
```

In the above example, we magically "guessed" what the virtual interface name was (we had the choice between 2 such interfaces, since there are only two containers running). In the `` `tcpdump' `` output, the IP address of the container where the ping process runs also appears; it is `172.17.0.2` .

In production situations however, there may be tens or hundreds of running containers in a cluster. In this case, we can automate the following cmds in a script to build a table of correspondence between container ID, container IP and container virtual interface, veth__. Starting from a container ID obtained from `` `$ docker ps -q' `` :

```
        $ docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' a754719db594
          172.17.0.2
        $ sudo ip link | grep $(docker exec 14dd5e0b3063 ethtool -S eth0 | awk '/peer_ifindex/ {print
$2;}') | cut -d " " -f 2 | cut -d "@" -f 1
          veth2177159
```

The resulting bash script is listed in Appendix D. <<<< TODO content CHECK

## 6-4. Listening to inter-container communication

Let us now simulate message-passing between 2 arbitrary containers among many in a cluster. For instance one may decide to ping a container from another container.

Start a ping from one container (ID `a754719db594`, IP 172.17.0.2) to another (ID `a754719db594`, IP 172.17.0.3):

```
        $ docker exec a754719db594 ping -w60 172.17.0.3
          PING 172.18.0.3 (172.17.0.3) 56(84) bytes of data.
           bytes from 172.17.0.3: icmp_seq=1 ttl=64 time=0.070 ms
          64 bytes from 172.17.0.3: icmp_seq=2 ttl=64 time=0.053 ms
```

To see this traffic from the DC host, we can capture packets on either one of the known virtual interfaces corresponding to the containers, or we can capture traffic on the bridge network interface (`'docker0'`). The bridge network interface shows all inter-container communication on the corresponding subnet 172.17.0.0/16:

```
        $ sudo tcpdump -ni docker0 host 172.17.0.2 and host 172.17.0.3
          listening on docker0, link-type EN10MB (Ethernet), capture size 262144 bytes
          20:55:37.990831 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 14, seq 200, length 64
          20:55:37.990865 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 14, seq 200, length 64
          20:55:38.990828 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 14, seq 201, length 64
          20:55:38.990866 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 14, seq 201, length 64
```

Equivalent data would be obtained by querying either containers' virtual interface, using:

```
        $ sudo tcpdump -i veth2177159 icmp  # for container ID a754719db594 with IP address 172.17.0.2
or      $ sudo tcpdump -i vethd8e05dd icmp  # for container ID e88238f68668 with IP address 172.17.0.3
```

## 6-5. Section summary

• A bridge network has a corresponding linux bridge interface on the docker host that acts as a layer2 switch, and which connects different containers on the same subnet.

• Each network interface in a container has a corresponding virtual interface on the docker host that is created while the container is running.

• A traffic capture from the docker host on the bridge interface is equivalent to configuring a SPAN port on a switch in that you can see all inter-container communication on that network.

• A traffic capture from the docker host on the virtual interface (veth-*) will show all traffic the container is sending on a particular subnet.

• Linux IPTables rules are used to block different networks (and sometimes hosts within the network) from communicating using the filter table. These rules are usually added in the DOCKER-ISOLATION chain.

• Containers communicating with the outside world through a bridge interface have their IP hidden behind the docker host's IP address. This is done by adding rules to the NAT table in IPTables.

# 7. Dockerizing an appplication

Running the cmd below should normally give you a python prompt as part of an interactive tty session:

```
$ docker run -it python:3.6
```

after which you could write normal Python code. Upon exiting the Python prompt `` `>>> quit()` ``, you'd both quit the Python process and leave the interactive container mode, the which shuts down the Docker container without deleting it.

Instead, in section 7-1, we're going to run a Jupyter image, that is to say an **application specific** image. In the following sections, we go into more details in the blueprint of the image, called a **Dockerfile**, which can be found in this Github repo. The Docker file can be thought of as the *source code* for the created image. A Docker image's Dockerfile is commonly hosted on Github, while the built image is hosted on DockerHub.

## 7-1. Jupyter notebook with Python3 on Docker

Let's run the minimal-notebook that only has Python and Jupyter installed. Issue:

```
$ docker run jupyter/minimal-notebook
```

As you expect this will pull the latest image of the *`minimal-notebook`* from the jupyter DockerHub account. At startup, the output will include the URL to paste in your browser to connect for the first time. That URL includes a connection token.

A more sophisticated way to work with a prepared Jupyter NB 4Gb image is exemplified below. Firt navigate to the directory where you want to work and possibly save result files generated from the Jupyter NB.

```
$ cd path/to/work/directory/
$ docker run -it -p 10000:8888 -v "$PWD":/home/jovyan/work jupyter/scipy-notebook
[...]
The Jupyter Notebook is running at:
http://127.0.0.1:8888/?token=a6ab2f2c183ba86ab3d60ae41bddfc61a91ba180b3b26622
```

The scipy-notebook's contents are listed in Appendix .

That cmd does three things:

(i) it does not remove the container when the Jupyter kernel is shut down. This permits the re-use of the container when the Jupyter NB must be completed or modified in ulterior work sessions. Adding the flags ***--it*** secures an interactive tty throughout the use of your DC. This may be dispensed with in this particular example.

(ii) it exposes DC's internal port 8888 on host port 10000

(iii) it bind-mounts host's volume *$PWD* on */home/jovyan/work* in DC's namespace's work directory, with the ***-v* | *--volume dir1:dir2*** flag. Any result file created and saved on NB namespace-disk, in *~/work*, by the Jupyter NB will be accessible during DC execution and after DC shutdown in *$PWD*.

To connect to the Jupyter NB in your browser, the correct URL to paste is:

http://127.0.0.1:**10000**/?token=a6ab2f2c183ba86ab3d60ae41bddfc61a91ba180b3b26622

Note that instead of exposing DC's port 8888 on a specific host's port (here 10000), you could have used:

```
$ docker run -it -P -v "$PWD":/home/jovyan/work jupyter/scipy-notebook
```

which exposes all DC's namespace ports onto so many host's ports. To discover the host's port randomly assigned to DC's port 8888 , run:

```
$ docker port $(docker ps -l --format={{.Names}}) 8888
0.0.0.0:32769
```

You can also get the NB's token from the logs, with:

```
$ docker logs --tail 3 $(docker ps -l --format={{.Names}})
Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
http://localhost:8888/?token=a6ab2f2c183ba86ab3d60ae41bddfc61a91ba180b3b26622
```

where as before the actual URL's port should be 32769 in this example.

As seen before, CTRL-P CTRL-Q (or ^p^q) allow users to recover their usual host shell-prompt. In doing so the DC keeps running but in *detached* mode. No process is interrupted, nor is the DC stopped.

Per Section 3-3, stopping a running DC is as simple as:

```
        $ docker stop container-id
or      $ docker stop container-name
```

and restarting a stopped DC as:

```
        $ docker start [-a] [-i] container-id
```

where flags:

| | |
|---|---|
| ***-a, --attach*** | attach *STDOUT/STDERR* and forward signals |
| ***-i, --interactive*** | attach container's *STDIN* |

## 7-2. Build a Dockerfile

…. lookup https://docs.docker.com/engine/reference/builder/ ….

## 8. Some other section

## 9. Miscellaneous

Congratulation if you've reached the end without skipping anything. Docker is an immensely powerful technology, and this tutorial was merely an introduction for those who have never used it before and want to get started quickly. You can **learn much more about Docker** from the official documentation, which is always kept up to date.

For a **refresher on basic Docker cmds**, take a look at the Docker Cheat Sheet on GitHub[i], as well as at the Arch Linux wiki.

For **security minded users**, references on how to get started in that area can be found here, in relation with Docker Compose, an orchestration tool for DCs.

If you would like to learn how to **define and deploy applications with Docker**, check the Get started with Docker guide.

Last, if you need to **troubleshoot any problem pertaining to Docker**, Docker site's official documentation is often the best place to start looking for a solution. StackExchange and StackOverflow are also good sources for troubleshooting tips.

i https://github.com/wsargent/docker-cheat-sheet

# Appendix A

## Contents of Docker Container

***jupyter/scipy-notebook***

- Miniconda Python 3.x in `/opt/conda`
- Jupyter Notebook server
- Pandoc and TeX Live for notebook document conversion
- *git*, *emacs*, *jed*, *nano*, *tzdata*, and *unzip*
- Unprivileged user jovyan (`uid=1000`, configurable, see options) in group users (`gid=100`) with ownership over the `/home/jovyan and /opt/conda` paths
- *tini* as the container entrypoint and a `start-notebook.sh` script as the default command
- A `start-singleuser.sh` script useful for launching containers in JupyterHub
- A `start.sh` script useful for running alternative commands in the container (e.g. ipython, jupyter kernelgateway, jupyter lab)
- Options for a self-signed HTTPS certificate and passwordless sudo
- pandas, numexpr, matplotlib, scipy, seaborn, scikit-learn, scikit-image, sympy, cython, patsy, statsmodel, cloudpickle, dill, numba, bokeh, sqlalchemy, hdf5, vincent, beautifulsoup, protobuf, and xlrd packages
- ipywidgets for interactive visualizations in Python notebooks
- Facets for visualizing machine learning datasets

Python packages can be updated or installed manually from within the Jupyter notebook.