Docker: The fundamentals

Gianluca Arbezzano

PUBLISHED BY THUMPFLOW
SCALEDOCKER.COM

Last release May 28, 2017v1.0

Contents

1 About the book 5

1.1 Audience 5

1.2 Mission 5

1.3 This Chapter 6

2 Containers: why we are here 7

2.1 Isolation and Virtualization 7

2.2 The Reason 9

2.3 Linux Container in practice 9

3 Docker: The Fundamental 13

3.1 Introduction 13

3.2 Install Docker on Ubuntu 16.04 14

3.3 Install Docker on Mac 15

3.4 Install Docker on Windows 16

3.5 Run your first HTTP application 16

3.6 Docker engine architecture 17

3.7 Image and Registry 18

3.8 Docker Command Line Tool 22

3.8.1 run 23

3.8.2 ps 24

3.8.3 exec 25

3.8.4 logs 25
3.8.5 push and pull 25
3.8.6 tags 25
3.8.7 inspect 25
3.8.8 start, stop, restart, kill 26
3.8.9 save, import 26
3.9 Volumes and File Systems 26
3.10 Network and Links 28
3.11 In the end 30
4 Biography 31

1. About the book

"Docker in production - Drive your boat like a Captain". You can visit scaledocker.com to keep in touch with info about the status of the book, new chapters distributed free an so on.

Docker 1.12 and Swarmkit opened the doors of your production environment and now you can manage it directly with Docker. It means that you need to care about distribution, monitoring, security, rollback and a lot of other topics that you are usually not managing in a development environment.

The general structure of the book is:

1. Containers why we are here
2. Docker the fundamentals
3. Docker over the engine
4. Distributed systems with SwarmKit
5. Road to production with Docker 1.12 and Swarm Mode
6. Monitoring, logging and high availability
7. Swarm, Mesos, Kubernetes an ocean of containers
8. Networking
9. Security
10. Extend Docker with API and events

1.1 Audience

Are you a developer or an IT Admin who has a basic knowledge of docker but is looking to understand how to manage a multi host docker infrastructure or thinking about how to manage a production environment with docker? Then this book is for you.

1.2 Mission

The mission of this book is to describe the new orchestration framework embedded in Docker 1.12 so that you can get started with Docker's clusters the right way and build to a robust production environment.

Manage a production environment is not easy microservices and distributed systems makes the ecosystem different. In this book we will how our production environment changed with there new methodologies.

1.3 This Chapter

This chapter is an extract from the entire book "Docker in Production - Driver your boat like a Captain". It's distributed free to receive opinions and ideas. Like Docker Captain I am active into the community in different ways as a speaker, writer and so on. This book is a good method to share my experience and my passion about automation and distributed system.

2. Containers: why we are here

Reviewers: Arianna Scarcella, Jenny Burcio

It is change, continuing change, inevitable change, that is the dominant factor in society today. No sensible decision can be made any longer without taking into account not only the world as it is, but the world as it will be... This, in turn, means that our statesmen, our businessmen, our every man must take on a science fictional way of thinking

Asimov, 1981

2.1 Isolation and Virtualization

I can see clearly two kind of invention: the ones that allow people to do something they couldn't do before and the ones that let them do something better. Fire, for example, gave people the chance to cook food, push away wild beasts and warm themselves up during cold nights. Many years later, electricity let people warm their houses just by pushing a button. After wheels discovery people began to travel and to trades goods, but was only with car's invention that they might do it faster and efficiently. Similarly, the web creates a huge network, able to connect people all over the world, web application gave people tools to use and customise such a complex system. Under this perspective, container is one of the main revolution of the last years, a unique tool that helps with app management and development. Let's discover something more about the real story of containers.

We have not a lot of documentation about why Bill Joy 18th March 1982 added chroot into the BSD probably to emulate him solutions and program is an isolated root. That's was amazing but not enough few years later in 1991 Bill Cheswick extended chroot with security features provided by FreeBSD and implemented the "jails" and in 2000 he introduced what we know as the proper jails command now our chroots can not be anything, anywhere out of themself. When you start a process in a chroot the PID is one and there is only that process but from outside you can see all processes that are running in a chroot. Our applications can not stay in a jail! They need to communicate with outside, exchange information and so on. To solve this problem in 2002 in the kernel version 2.4.19 a group of developers like Eric W. Biederman, Pavel Emelyanov introduced the namespace feature to manage system resources like network, process and file system.

This is just a bit of history about how the ecosystem spin up, in the end of this chapter we will try to understand how why Docker arrives on the scene, but the main goal of this book is on another layer and on another complexity, we are here to understand how manage

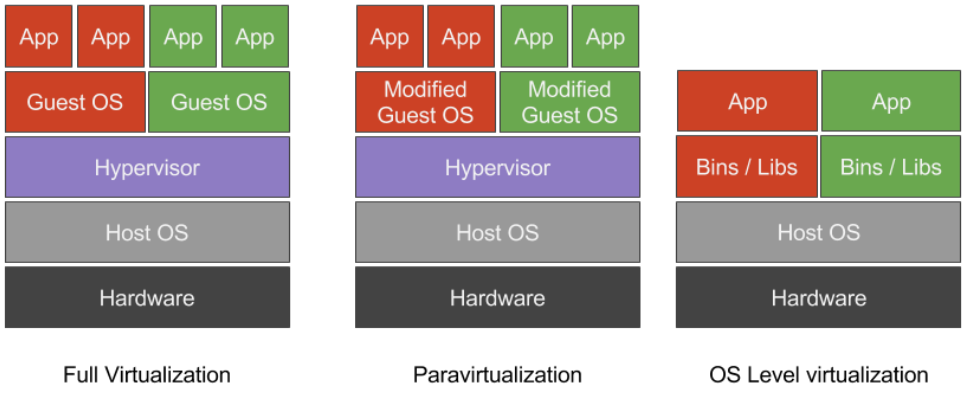
all this things in cloud and how to design a distributed system but you know the past is important to build a solid future.

All this great features are now popular under the name of container, nothing really news and this is one of the reason about why all this things are amazing! They are under the hood from a while! Solid and tested feature put together and made usable.

Nothing to say about the importance for a system to being isolated: isolation helps us to usefully manage resources, security and monitoring, in the best way, false problems creation in specific applications, often not even related to our app.

The most common solution is virtualization: you can use an hypervisor to create virtual server in a single machine. There are different kind of virtualization:

- Full virtualization
- Para virtualization like Virtual Machine, Xen, VMware
- Operating System virtualization like Containers
- Application virtualization like JVM



The main differences between them is how they abstract the layers, application, processing, network, storage and also about how the superior level interact with underlying level. For example into the Full virtualization the hardware is virtualized, into the para virtualization not.

Container is an operation-system-level virtualization. The main difference between Container and Virtual Machine is the layer: the first works on the operating system, the second on the hardware layer.

When we speak about container we are focused on the application virtualization and on a specific feature provided by the kernel called Linux Containers (LXC): what we do when

we build containers is create new isolated Linux systems into the same host, it means that we can not change the operation system for example because our virtualization layer doesn't allow us to run Linux containers out of Linux.

2.2 The Reason

Revolutions are not related to a single and specific event but come from multiple movements and changes: Container is just a piece of the story.

Cloud Computing allowed us to think about our infrastructure as an instable number of servers that can scale up and down, in a reasonable short amount of time, with less money and without the investment requested to manage a big infrastructure made of more than one datacenter across the world. As a consequence, applications that had been in a cellar, now are on Amazon Web Service, with a load balancer and maybe different availability zone. This allowed little teams and medium companies, without datacenter and infrastructures, to think about concept like distribution, high availability, redundancy. Evolution never stop.

Once our applications are running in few virtual machines, our business will grow up so we start to scale up and down this servers to serve all our users. We experimented few benefits but also a lot of issues related, for example, to the time requested for managing this dynamism; moreover big applications are usually more expensive to scale. Our application can only grow but the deploy can be really expensive. We discovered that the behavior of an application is not the same across all of our services and entrypoint, because few of them receive more traffic that others. So, we started to split our big applications in order to make them easy to scale and monitor. The problem was that, in order to maintain our standard, we need to find a way to keep them isolated, safe and able to communicate each others.

The Microservices Architecture arrived and companies like Netflix, Amazon, Google and others counts hundreds and hundreds of little and specific of services that together work to serve big and profitable products.

Netlix is one of first companies that started sharing the way they build Netlix.com: with more that 400 microservices, they managed feature like registration, streaming, rankins and all what the application provides.

At the moment, Containers are the best solution for managing a dense and dynamic environment with a good control, security and for moving your application between servers.

2.3 Linux Container in practice

Nothing new under the sun, Wikipedia marks 2008 as the starting date for the LXC (Linux Containers) project. Joe Beda, Senior Software Engineer in Google, speaks about more

than 10 years of success with containers in production, underling that now everything is running inside a container. What Docker did, and still does, is make container friendly and provide a big ecosystem of tools to build, ship and run containers. Before speaking about Docker, we need to know what a Linux Container is in practice.

When we think container, we are using some features provided by the kernel cgroups, namespaces are really important. Control group (cgroup) limit and isolate resources usage from processes like memory, swap, cpu, i/o, networks. It also provide other features to control, monitor and prioritize cgroup and also process what they contain. For this reason, a good practice consist in paring one process to each container, in order to have a complete control on what the process does. Namespace wraps a resource that make it available for a process or for a group of them that use that namespace. There are different kind of namespace PID, network, IPC, mount, users and others.

Docker container uses cgroup to make a container isolated and namespace as access control to make it safe.

cgroups associate a tasks with a set of resources as:

- **memory** to set a memory usage limit.
- **cpu** to use the scheduler and to give CPU access to a cgroup.
- **cpuset** to assign individual CPUs and memory nodes.
- **cpuacct** to get automatic reports about CPU usage in a cgroup.
- **devices** to allow or deny access to a specific device.
- **blkio** to set I/O limits from a block device as disks or USB.
- **freezer** to suspend or resume a tasks in a cgroup.
- **ns** to use namespace inside a cgroup.
- **net_cls** to allow Linux traffic control to identity packets from a cgroup
- **net_prio** to set the priority of the network traffic.

We know that everything in Linux is a file, a cgroup is also a bunch of files located in */sys/fs/cgroup*. Manage cgroup as files is not easy and it's in practice impossible to maintain. For this reason there are tools that allow us to manage cgroup for example:

- **libcgroup** is a libray that abstract the control group kernel feature and it contains some tools like cgcreate, cgclassify.
- **cgmnager** is a combination of one daemon and a client (cgm) to manage cgroups.

You can install libcgroup in your Ubuntu 16.10 with the command:

```
$ sudo apt-get install cgroup-bin
$ ls -lsa /sys/fs/cgroup
total 0
0 drwxr-xr-x 14 root root 360 gen 14 20:47 .
0 drwxr-xr-x 10 root root 0 gen 21 15:18 ..
0 dr-xr-xr-x 6 root root 0 gen 21 15:18 blkio
0 drwxr-xr-x 2 root root 60 gen 14 20:47 cgmanager
0 lrwxrwxrwx 1 root root 11 gen 14 20:47 cpu -> cpu,cpuacct
0 lrwxrwxrwx 1 root root 11 gen 14 20:47 cpuacct -> cpu,cpuacct
0 dr-xr-xr-x 6 root root 0 gen 21 15:18 cpu,cpuacct
0 dr-xr-xr-x 3 root root 0 gen 21 15:18 cpuset
0 dr-xr-xr-x 6 root root 0 gen 21 15:18 devices
0 dr-xr-xr-x 4 root root 0 gen 21 15:18 freezer
0 dr-xr-xr-x 3 root root 0 gen 21 15:18 hugetlb
0 dr-xr-xr-x 7 root root 0 gen 21 15:18 memory
0 lrwxrwxrwx 1 root root 16 gen 14 20:47 net_cls -> net_cls,net_prio
0 dr-xr-xr-x 3 root root 0 gen 21 15:18 net_cls,net_prio
0 lrwxrwxrwx 1 root root 16 gen 14 20:47 net_prio -> net_cls,net_prio
0 dr-xr-xr-x 3 root root 0 gen 21 15:18 perf_event
0 dr-xr-xr-x 6 root root 0 gen 21 15:18 pids
0 dr-xr-xr-x 7 root root 0 gen 21 15:18 systemd
```

Into the directory we can see all control groups listed before. At this point we can try to create our first cgroup:

```
$ sudo cgcreate -a user -g memory,cpu:groupname
```

This command crateas a cgroup called foo with two control group attached memory and cpu. We can verify that now the is a foo directory into the *cpu* and *memory* directories.

```
$ ls -lsa /sys/fs/cgroup/cpu/foo/
total 0
0 drwxr-xr-x 2 gianarb root 0 gen 21 15:53 .
0 dr-xr-xr-x 7 root  root 0 gen 21 15:53 ..
0 -rw-r--r-- 1 gianarb root 0 gen 21 15:53 cgroup.clone_children
0 -rw-r--r-- 1 gianarb root 0 gen 21 15:53 cgroup.procs
0 -r--r--r-- 1 gianarb root 0 gen 21 15:53 cpuacct.stat
0 -rw-r--r-- 1 gianarb root 0 gen 21 15:53 cpuacct.usage
0 -r--r--r-- 1 gianarb root 0 gen 21 15:53 cpuacct.usage_all
0 -r--r--r-- 1 gianarb root 0 gen 21 15:53 cpuacct.usage_percpu
0 -r--r--r-- 1 gianarb root 0 gen 21 15:53 cpuacct.usage_percpu_sys
0 -r--r--r-- 1 gianarb root 0 gen 21 15:53 cpuacct.usage_percpu_user
0 -r--r--r-- 1 gianarb root 0 gen 21 15:53 cpuacct.usage_sys
0 -r--r--r-- 1 gianarb root 0 gen 21 15:53 cpuacct.usage_user
0 -rw-r--r-- 1 gianarb root 0 gen 21 15:53 cpu.cfs_period_us
0 -rw-r--r-- 1 gianarb root 0 gen 21 15:53 cpu.cfs_quota_us
0 -rw-r--r-- 1 gianarb root 0 gen 21 15:53 cpu.shares
0 -r--r--r-- 1 gianarb root 0 gen 21 15:53 cpu.stat
```

```
0 -rw-r--r-- 1 gianarb root 0 gen 21 15:53 notify_on_release
0 -rw-r--r-- 1 root  root 0 gen 21 15:53 tasks
```

We can run command into the cgroup for example:

```
$ cgexec -g memory,cpu:groupname/foo sleep 5
```

You can change the limits of your resources, our case CPU and memory in a very easy way, if you read the content of */sys/fs/cgroup/memory/foo/memory.limit_in_bytes* you get the current maximum amount of memory that a task can use into the foo cgroup. You can change this limit with

```
echo 20000000 > /sys/fs/cgroup/memory/groupname/foo/memory.limit_in_bytes
```

To remote the cgroup you can just exec:

```
sudo cgdelete -g memory,cpu:foo
```

Now you can verify yourself the *foo*'s directories expired.

This example was completly no-sense and semplicistic but I noticed that for a lot of Docker users all the lowel level mechanisms are just a magic black box and usually run in poduction something that you don't understand it's not a wise decision. Now that you have a very high level idea about what there is above, it's time to see what Docker really means.

3. Docker: The Fundamental

Reviewers: Gareth Pelly, Thomas Shaw, Jenny Burcio, Stefano Rosso

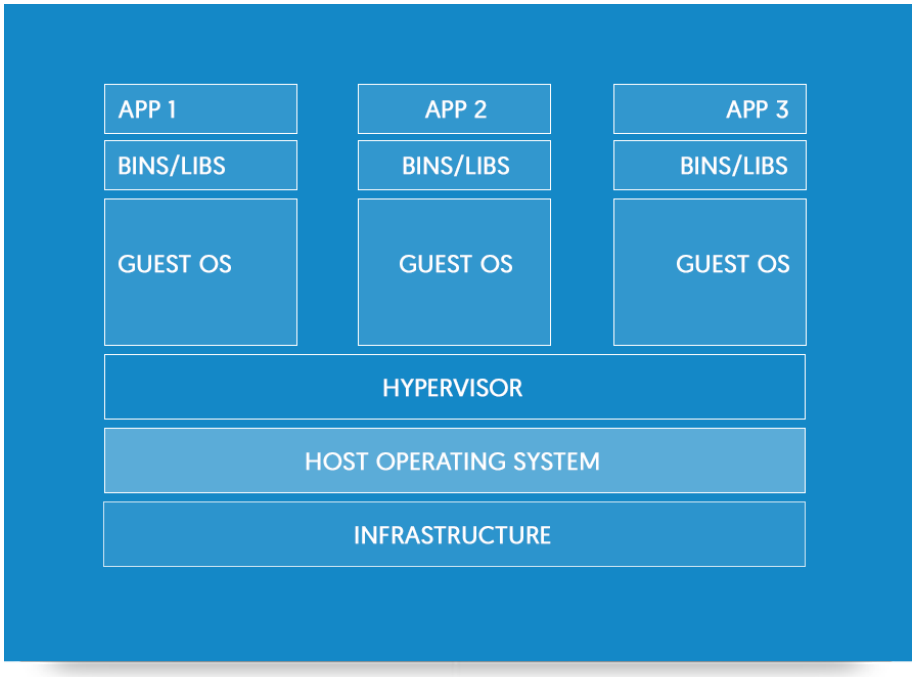
Can I jump over two or three guys like I used to? No. Am I as fast as I used to be? No, but I still have the fundamentals and smarts. That's what enables me to still be a dominant player. As a kid growing up, I never skipped steps. I always worked on fundamentals because I know athleticism is fleeting.

Kobe Bryant

3.1 Introduction

Get ready to explore the big Docker world. It's time to understand why Docker created a big revolution and containers are the current IT revolution. Docker is an open source project to build, ship and run containers. In this chapter we will focus on the Docker Engine because it is the core application that helps us to create and manage the Docker container. Docker Engine is open source and it's written in Golang. With more than 27.000 commits and 1.500 contributors, it can be considered one of the biggest open source projects worldwide.

Figure 3.1: Docker container virtualization by Docker Inc.



This diagram shows, in broad terms, where the Docker Engine works: it's located between the operating system and our applications. Docker Containers sit on top of Linux so we need a Linux kernel to work with Docker. My local environment is a MacBook Pro and a lot of examples that we will see in this book are run on my laptop with a virtual machine managed by VirtualBox.

3.2 Install Docker on Ubuntu 16.04

This book is interactive, you can read it but also try yourself and for this reason has send start from how install Docker on your laptop. It is distributed on different Linux distribution, we can start from Ubuntu with few simple steps:

1. Update your repositories

Listing 3.1: setup apt

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80
    --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

2. If the file doesn't exist, create it with your favourite editor

```
$ /etc/apt/sources.list.d/docker.list
```

3. Copy this line into the file

```
$ deb https://apt.dockerproject.org/repo ubuntu-xenial main
```

4. Update APT index

```
$ sudo apt-get update
```

5. Install Docker

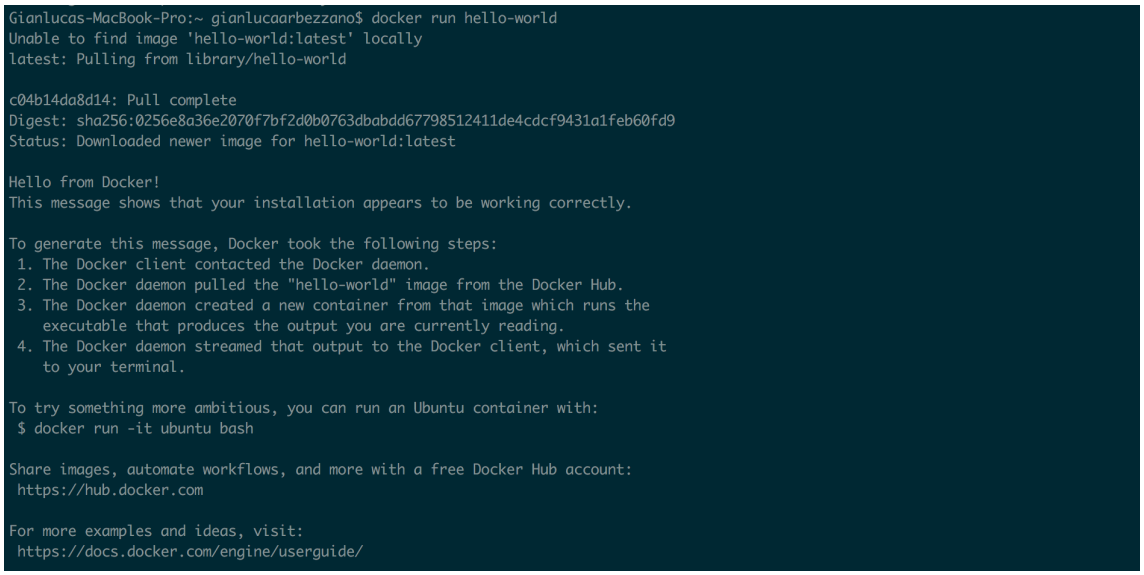
```
$ sudo apt-get purge lxc-docker
$ sudo apt-get install linux-image-extra-$(uname -r)
    linux-image-extra-virtual docker-engine
```

6. Start the docker daemon

```
$ sudo service docker start
```

7. Start your first container

```
$ sudo docker run hello-world
```

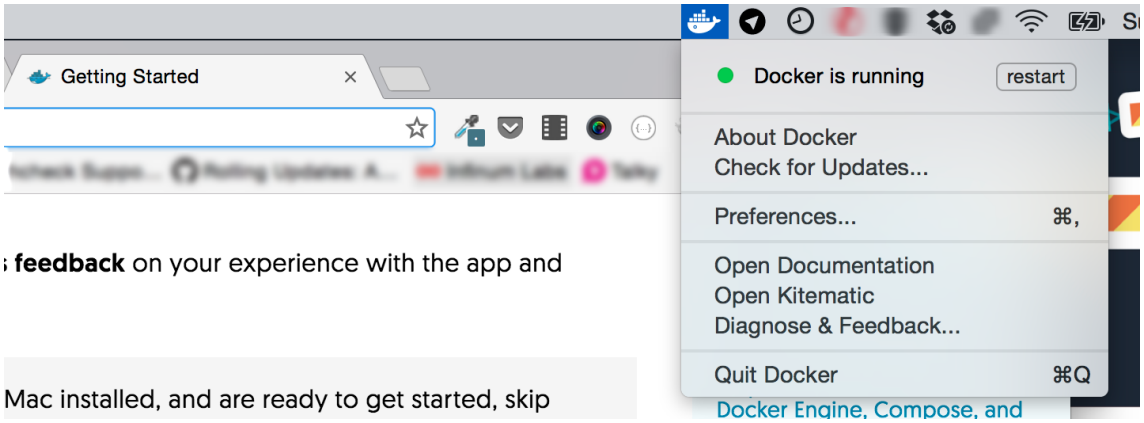


Docker usually provides good documentation to install on Linux but if you have a MacBook or use Windows, the next two paragraphs will show you how to install[1] it on your computer.

3.3 Install Docker on Mac

Docker for Mac[2] is a tool which provides a better experience with the Docker Engine in OSX. It's a simple dmg, you can download directly from the site and install in your MacBook.

[1] https://docs.docker.com/engine/installation/
[2] https://docs.docker.com/docker-for-mac/



When it's up and running you can enjoy Docker in your terminal. Just open it and run

```
$ sudo docker run hello-world
```

3.4 Install Docker on Windows

The same concept applies for Windows in order to have a great experience in your development environment. You download Docker for Windows[3], install it as a classic application and you can start to play with Docker.

Remember these two solutions don't change the requirement that Docker has to run with Linux. Both Docker for Mac and Docker for Windows use a virtual machine and a common layer to make your experience with Docker exactly as with Linux.

3.5 Run your first HTTP application

Micro[4] is a service written in Golang, a very small application which provides two HTTP entrypoints. The index show the container's ip and there is a /health call to check if your service works as expected. It runs on port 8000.

```
$ docker run -p 8000:8000 gianarb/micro:1.0.0
```

[3]https://docs.docker.com/docker-for-windows/
[4]https://github.com/gianarb/micro/releases/tag/1.0.0

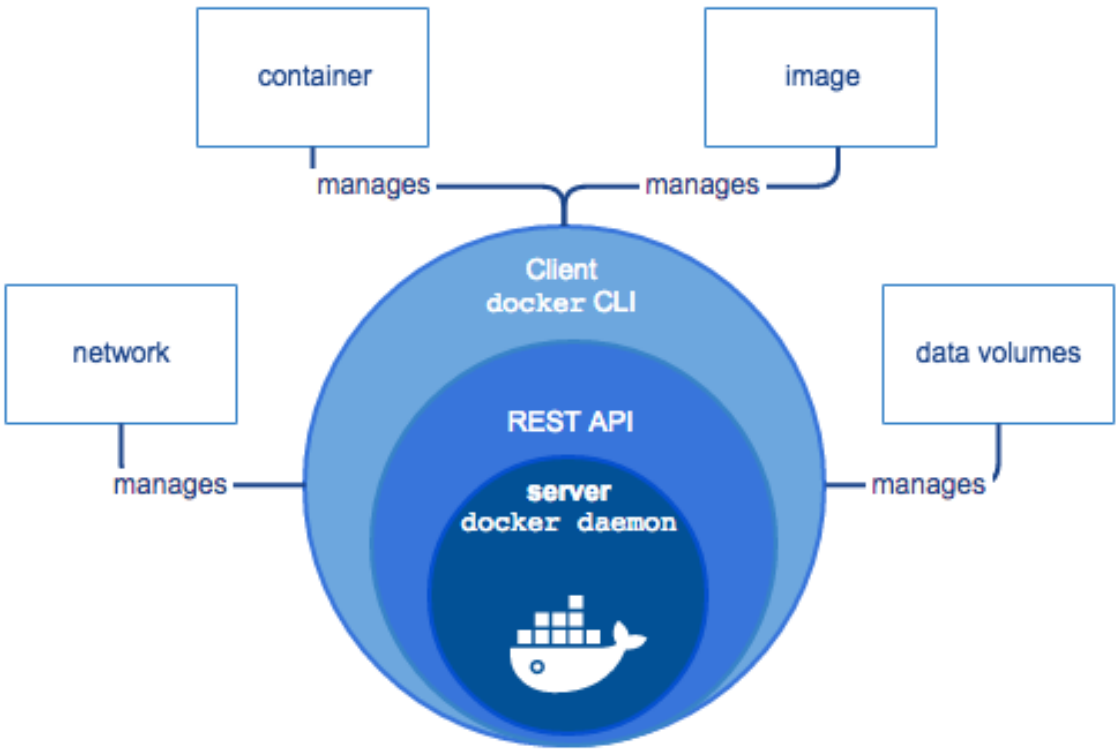
```
Gianlucas-MacBook-Pro:~ gianlucaarbezzano$ docker run -p 8000:8000 gianarb/micro:1.0.0
Unable to find image 'gianarb/micro:1.0.0' locally
1.0.0: Pulling from gianarb/micro

e110a4a17941: Already exists
8ceaae8440c5: Pull complete
0d2896b707d7: Pull complete
Digest: sha256:2a61eebc376fcdeac1de0e78a0baabb39c610ebbc411d1281940a82338380782
Status: Downloaded newer image for gianarb/micro:1.0.0
2016/09/11 23:19:27 Micro started
2016/09/11 23:19:33 %s called / localhost:8000
2016/09/11 23:19:34 %s called / localhost:8000
2016/09/11 23:19:58 %s called /health localhost:8000
```

'run' is one of the most important commands that Docker Engine provides. It starts a new container from an image, in this case gianarb/micro version 1.0.0. '-p' is an option to forward a port from the container to the host, in this case it allows us to contact our HTTP server. Now you can open your browser and see the site on localhost:8000.

3.6 Docker engine architecture

Figure 3.2: Docker archteture by Docker Inc.



We saw something about images, containers, command line and daemon but before continuing we need to understand how Docker Engine is designed in order to use the same terminology and also to have an idea about the architecture. Docker is made of two main parts:

- An HTTP server, in practice the backend that provides a REST API that you can use to integrate your applications with the engine.
- Docker CLI, a command line tool that uses the API to make all features easy to use.

From Docker 1.11, the CLI and the daemon are two different binaries you can decide to install or just the engine if you don't need the command line. This solution is also good for production, where you can install only the daemon.

You can use the Docker daemon directly with the command 'dockerd'. It supports different communication protocols, by default it exposes a Unix Socket in unix:///var/run/docker.sock. There is a long list of options that you can append after the command, for example:

- -D to allow debug mode.
- -H to configure the tcp host tcp://192.168.1.3:2376 for example.
- –tls to enable or disable TLS, by default it's false but it's highly recommended in production

```
$ dockerd -D -H tcp://192.168.1.3:2376 --tls=false
```

Remember that the best way to manage the process and the service is with a process manager. Each Linux distribution has their own application Upstart or systemd for Ubuntu, Fedora and CentOS.

During the installation process for the most common Linux distributions like Ubuntu, Fedora and CentOS, Docker sets up a common and standard init script that you can always modify to fit your use case. In Ubuntu 15.04 you can find it in /etc/init.d/docker. It contains start, restart, stop and all functions that you need to manage the process.

3.7 Image and Registry

To understand the Docker internals, we need to cover two other concepts: image and registry. Every container starts from a base image, a read-only template that contains your application or a base Ubuntu for example.You can create an image from a container or from another image with a Dockerfile. Let me proceed with an example, pull Nginx from the official repository with

```
\$ docker pull nginx
```

```
Gianlucas-MacBook-Pro:github.com gianlucaarbezzano$ docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx

8ad8b3f87b37: Already exists
c6b290308f88: Pull complete
f8f1e94eb9a9: Pull complete
Digest: sha256:aa5ac743d65e434c06fff5ceaab6f35cc8519d80a5b6767ed3bdb330f47e4c31
Status: Downloaded newer image for nginx:latest
Gianlucas-MacBook-Pro:github.com gianlucaarbezzano$
```

We are downloading an official image from the hub and there are a lot of these kind of images on the hub. 'Official' means that some organizations or some companies created and maintains Nginx, Ubuntu, Debian, Redis, Apache, Php plus others and they allow us to start our container from these images. Each layer composes the filesystem of our container and Docker manages parallel downloads in order to make our pull efficient and fast.

```
\$ docker run -d --name demo_nginx -p 80:80 nginx
\$ docker ps
```

```
Gianlucas-MacBook-Pro:github.com gianlucaarbezzano$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                           NAMES
1c50c34d399e        nginx               "nginx -g 'daemon off"   32 seconds ago      Up 30 seconds       0.0.0.0:80->80/tcp, 443/tcp     demo_nginx
Gianlucas-MacBook-Pro:github.com gianlucaarbezzano$
```

At this point, we have pulled an image that someone made, we created a new container and now we can open our browser on localhost:80 and see our nginx up and running. We can go directly into the container with:

```
$ docker exec -it demo_nginx /bin/bash
```

If we change the index.html and we want to save this change so we can use it in the future, we would need to create a new layer on top of the previous ones. Every change made in a running container will be lost if we don't create this new saved point. We can run this command outside of the previous container:

```
$ docker commit demo_nginx my/nginx
```

At this point we created a new layer and we have a new image called my/nginx to use as a starting point for other images or to deploy in our servers. We need to read this change from the top to the bottom, if we modify and commit index.html twice we will see just the last version.

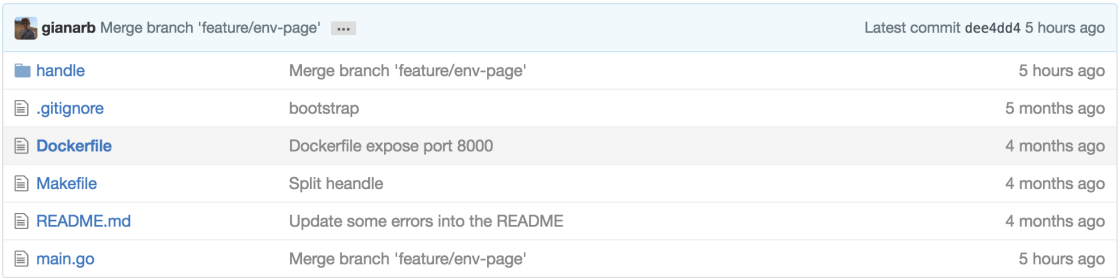
UnionFS[5] is a filesystem service for Linux that implements a union mount for other file systems in order to merge files and directories as branches and create a virtual filesystem.

[5] https://en.wikipedia.org/wiki/UnionFS

Docker uses this technology to split and mount images, an image is built on top of something else and these layers are not copied but reused when we need them. The top layer is a writable layer, it is added when a container starts and it's the main difference between an image and a container. An image doesn't have this layer and for this reason if you log inside a container and you make 'su' changes and you delete it without a commit you just lose your changes.

This is just one way that we can create images. The best way to create an image is a Dockerfile, because this file contains all the commands and information that we need to know about your image. If your container is created from interactive mode without the Dockerfile, you typed all commands directly into the container and you committed it after a short time and you are not able to recreate or update it anymore. Usually this file is located in your VCS close your application like into the micro application[6] or you can have a repository with a set of Dockerfiles to build your common images[7].

Figure 3.3: Dockerfile located in the application repository.



Putting a Dockerfile close to your application helps developers to build a development environment. Having one in your repository could be a smart way to enable new people to start to work on your application without losing time to configure their laptop with all services that your application requires. And also your application and your Dockerfile change together and you can track these changes in Git. This last aspect is interesting because you can rollback or apply QA not only to your code but also to your Dockerfile.

It practice a Dockerfile is a text file that contains commands and instructions that Docker needs to build an image.

```
FROM ubuntu:15.04

RUN apt-get update RUN apt-get install -y php5 RUN echo "<?php echo 'Hello
folks!';" > /var/index.php

WORKDIR /var

CMD ["php", "-S", "0.0.0.0:9090", "-t", "."]
```

[6] https://github.com/gianarb/micro
[7] https://github.com/gianarb/dockerfile

This is an example of Dockerfile, it contains a php script that prints “Hello folks!”. What we are doing with this file is specifying the base image.

```
FROM ubuntu:15.04
```

We are executing a command to install Apache2 and to create our index. CMD describes the command which starts the process in the container when it runs, in this case we are starting the PHP built-in web server.

```
$ docker build -t php-example .
```

‘build’ is a command to create an image from a Dockerfile, ‘-t’ is the image’s name and ‘.’ is the directory’s path that contains our Dockerfile. We can create a container from this image and enjoy our application with the ‘run’ command

```
$ docker run -it -p 9090:9090 php-example
```

Now open your favourite browser and open localhost:9090

In the real world, the index.php for your application is not created with the container but it’s added into it with the ADD instruction

```
ADD ./directory/to/add/ /path/into/the/container
```

In our specific example if the index.php is in the same folder as the Dockerfile, you can replace

```
RUN echo "<?php echo 'Hello folks!';" > /var/index.php
```

With

```
ADD ./index.php /var/index.php
```

An ./index.php is a php script that contains

```
<?php echo 'Hello folks!';
```

Now build your image once more, when you have the new image remove index.php from your filesystem and try to run a container

```
$ docker build -t your-username/php-example .
$ rm -rf index.php
$ docker run -it -p 9090:9090 your-username/php-example
```

You’ll notice that without the file you can see the correct page in the browser because you built a new container that contains your application, it means that you can share and deploy it like an artifact.

For this reason another important tool that Docker provides is the registry, it helps us to manage the distribution of our images. There are public and private registries and the most well known is hub.docker.com. It contains a lot of public images but once registered you can also manage your private images. We will see in Chapter 3 how to install and manage a private registry for your team or company. Go ahead and Register on the Hub if you haven't already because now it's time to push 'your-username/php-example' into the Hub.

```
$ docker login
```

This command allows Docker daemon to communicate with your Hub account. The name of the image defines a specific behavior, in my case my-username becomes 'gianarb' because it's my username in Docker Hub.

```
    $ docker push gianarb/php-example
```

Now our image is public in the hub and we can see all our images in the profile page.

```
$ docker rmi -f gianarb/php-example
$ docker pull gianarb/php-example
```

The first command deletes the image from our laptop and the second command pulls the image from the Hub. This is a very simple example, but it explains what a registry is and how it manages your images. Docker can also manage tags and has utilities to know the difference between your image and your container in order to understand what you have before creating a new image from a container. In the next chapters we will see all these features in practice.

3.8 Docker Command Line Tool

The command line is a complete and powerful client to interact with the daemon and knowing how it works will help you to create and manage containers.

docker –help shows the list of commands, we saw already some of them but you can use this section as a summary of the most used commands.

```
Commands:
    attach    Attach to a running container
    build     Build an image from a Dockerfile
    commit    Create a new image from a container's changes
    cp        Copy files/folders between a container and the local filesystem
    create    Create a new container
    diff      Inspect changes on a container's filesystem
    events    Get real time events from the server
    exec      Run a command in a running container
    export    Export a container's filesystem as a tar archive
    history   Show the history of an image
    images    List images
    import    Import the contents from a tarball to create a filesystem image
    info      Display system-wide information
    inspect   Return low-level information on a container, image or task
    kill      Kill one or more running containers
    load      Load an image from a tar archive or STDIN
    login     Log in to a Docker registry.
    logout    Log out from a Docker registry.
    logs      Fetch the logs of a container
    network   Manage Docker networks
    node      Manage Docker Swarm nodes
    pause     Pause all processes within one or more containers
    port      List port mappings or a specific mapping for the container
    ps        List containers
    pull      Pull an image or a repository from a registry
    push      Push an image or a repository to a registry
    rename    Rename a container
    restart   Restart a container
    rm        Remove one or more containers
    rmi       Remove one or more images
    run       Run a command in a new container
    save      Save one or more images to a tar archive (streamed to STDOUT by default)
    search    Search the Docker Hub for images
    service   Manage Docker services
    start     Start one or more stopped containers
    stats     Display a live stream of container(s) resource usage statistics
    stop      Stop one or more running containers
    swarm     Manage Docker Swarm
    tag       Tag an image into a repository
    top       Display the running processes of a container
    unpause   Unpause all processes within one or more containers
    update    Update configuration of one or more containers
    version   Show the Docker version information
    volume    Manage Docker volumes
    wait      Block until a container stops, then print its exit code
```

3.8.1 run

```
$ docker run hello-world
```

This command uses the hello-world image to run a new container. It has a lot of options that you can use to share ports, mount volumes, run the container in a specific network and also limit resources for a specific container. Let me show you a long composition in order

to understand in practice what means.

```
$ docker run -i -t -p 8000:8000 \
    -v /home/gianarb/.ssh/id_rsa:/root/.ssh/id_rsa \
    -v $PWD:/opt \
    --network front-tier \
    --memory 10M \
    --name site \
    gianarb/micro
```

- -i keeps the STDIN open
- -t allocates a tty
- -p shares a port from the container to the host in this case port 8000 to 8000, this value is an array, you can share more ports. You just need to add more -p options.
- -v shares files and directories from the host to the container like volumes. This value is an array and in this example we are sharing two volumes, the first one is my ssh key that it's stored in the host into the path /home/gianarb/.ssh/id_rsa in the container /root/.ssh/id_rsa
- –network insert this container into the 'front-tier' network, we will see later what it means
- –memory limits the RAM usable from the container to 10M
- –name is the name of the container
- gianarb/micro is the name of the image

3.8.2 ps

```
$ docker ps
```

```
Gianlucas-MacBook-Pro:github.com gianlucaarbezzano$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                           NAMES
1c50c34d399e        nginx               "nginx -g 'daemon off"   32 seconds ago      Up 30 seconds       0.0.0.0:80->80/tcp, 443/tcp     demo_nginx
Gianlucas-MacBook-Pro:github.com gianlucaarbezzano$
```

This command shows you the number of containers that are running with a high level summary of your containers. If you use the option -a you can also list the stopped containers. Every container has an id or a name, you can use the complete name or a partial id (only the first letter if it's unique) to identity a specific container when you run a command that requires a container id.

3.8.3 exec

```
$ docker exec container_name ls -lsa
```

This executes a command (ls -lsa) in this case in a running container (container-name).

3.8.4 logs

```
$ docker logs container_name
```

This shows the logs written by a container, you can follow them with the -f option.

3.8.5 push and pull

```
$ docker pull gianarb/micro:1.0.0
$ docker push gianarb/micro
```

These two commands are used to manage an image and push or pull it from or to a remote registry.

3.8.6 tags

```
$ docker tag 5wgs46h gianarb/micro:1.0.1
```

Every image can have a tag, it is a label assigned to an image. The default one is 'latest' but you can follow best practice and use SemVer[8] for example like a normal codebase. In this example we are tagging the image with ID 5wgs46h like example:1.0.1. You can also use the tag command to prepare an image to be pushed to your own registry, the default one is registry-1.docker.io but you can specify another and push your image to a private one.

```
$ docker tag 5wgs46h registry.gianarb.it:gianarb/micro:1.0.1
```

3.8.7 inspect

```
$ docker inspect 4645shgre
```

Inspect returns information about an image or a running container. One of the very useful options is –format, it converts the common cli output to raw json with information about volumes, ports, ip and everything the daemon knows about the container. This is very useful to create basic automation with tool like jq[9] that allow you to cross a JSON directly into the bash.

[8] http://semver.org/
[9] https://stedolan.github.io/jq

3.8.8 start, stop, restart, kill

```
$ docker start 464gts4
```

A Docker container exposes and manages processes therefore you can start, stop, kill and restart a container like a normal process.

3.8.9 save, import

```
$ docker save gianarb/micro > micro.tar.gz
$ cat micro.tar.gz | docker import - my/micro:new
$ docker import http://example.exampleapp.com/micro.tar.gz
```

We already know that the registry is the best way to ship and distribute images but Docker also supports the 'save' command to create a tarball of the image and the 'import' command to create the Docker image from the tarball. This practice is good in development mode if you need to share something with someone close to you. But to manage deployment and production releases, the registry is the best way to ship and distribute images.

3.9 Volumes and File Systems

We know that a container has its own isolated filesystem and we also saw that it's built of different layers; but we haven't yet gotten into volumes. A data volume allows your container to bypass the Union Filesystem. Volumes are initialized when a container is created and they provide features to persist and share your data. Every volume has a mount point and if there is existing data it will be copied into the container's volume to be available and usable:

```
$ docker run -it -v /tmp ubuntu /bin/bash
```

-v is the option to create and attach new volumes.

```
$ docker run -it -v /tmp -v /opt ubuntu /bin/bash
```

You can also concatenate more than one in order to attach more volumes to the same container.

Create a volume is useful if your application stores files that you need to mount in other containers for example because you can mount the same volume on different containers. If the current container dies or if you delete it all files store into the volume will be available and you can attach them in the next container. Volumes are usually used to manage logs and cache for example. You can also describe it in the Dockerfile with the keyword VOLUME:

```
VOLUME /tmp
```

You can also map a specific directory from your host into the container -v /tmp:/temporary where the first path is the directory into the host and the second is the mount point into the container. Because the changes persist in a volume are shared inside and outside of the container, this feature is perfect in development as you can mount your project in a container and simulate your production environment It's also important to remember that data inside a volume is also persisted when your container expires.

If you are not mapping a volume but you need to know where it is located on your host, you can use the 'docker inspect' command and look at the Mounts field

```
{
  ...
     "Mounts": [
           {
               "Name": "0f83dc",
               "Source": "/var/lib/docker/volumes/0f83dc/_data",
               "Destination": "/tmp",
               "Driver": "local",
               "Mode": "",
               "RW": true,
               "Propagation": ""
           }
       ],
  ...
}
```

You can also use a cli command like

```
$ docker volume ls
$ docker volume create
$ docker volume rm
```

to create, delete and persist other actions to a volume.

This feature is a core feature which makes a container usable in development and production. For this reason, Docker also provides a good abstraction layer to create custom plugins to extend the system in order to mount different filesystems like NFS or a custom one. There are many plugins that allow you to mount distributed storage, for example, flocker[10]. Infinit[11] is another great project that provides a Docker integration to manage decentralized volumes on different platform like S3, Azure. I met the folks behind this project during Docker Summit 2016 in Berlin, and it is a great service to manage persistent and scalable storage on different Cloud providers. When you work in a cluster of Docker Engines, your volume is created in the same host where your container starts. If your system spins up containers in different servers this means that the other machines do not have this volume, Infinit helps you to avoid this problem and takes care of the migration of volumes in a cluster.

[10] https://github.com/ClusterHQ/flocker
[11] https://infinit.io/

Another way to use 'volume' is to make a backup. You can attach a volume created in another container with –volumes-from <name-volume> like in this example:

```
$ docker run --rm --volumes-from img_avatar -v \
    $PWD:/backup ubuntu tar cvf /backup/backup.tar
        /var/www/front/public/avatar
```

As you have learned, it is possible to the same volume in more than one container at the same time, but you need to remember that two applications that can write to the same files can corrupt data, useful but you need to understand when and how to use it.

3.10 Network and Links

If volumes help us manage our data, network and links are a great feature to connect our containers and allow communication between our applications. There are really important features because usually it's one of the first requirements for every application to be free to communicate with other services like: MySQL, Redis, Vault or with other containers in our cluster.

The simplest way to connect two containers is with a link:

```
$ docker run --name mysql-service -e MYSQL_ROOT_PASSWORD=root -d mysql
$ docker run --name my-wordpress \
    -e WORDPRESS_DB_PASSWORD=root \
    -e WORDPRESS_DB_HOST=mysql.my-wordpress \
    --link mysql-service:mysql.my-wordpress \
    -d wordpress
```

We create one MySQL container with a root password 'root' and we connect it with a wordpress container. The hostname that we can then use to reach MySQL from the wordpress container is 'mysql.my-wordpress'.

–link <container-name>:alias this is the link's format, you are creating bidirectional communication between two containers and the alias describe the hostname that you can use. You can create more than one link, you just need to add more –link options when you execute the run command.

Links are a legacy way to easily connect containers, it's very easy to understand but in Docker 1.9 we have a new stable concept of network ready to be used. But before proceeding, how does docker manage communication between containers?

```
docker@sw1:~$ ifconfig

docker0   Link encap:Ethernet  HWaddr 02:42:04:FF:32:1C
          inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

By default docker create a new network interface called docker0 and a bridge network that you can inspect with the command

```
$ docker network inspect bridge
```

It creates a default iptables NAT rule to allow outside communication between all containers connected to docker0 and the outside world.

```
$ sudo iptables -L -n
```

```
docker@d:~$ sudo iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DOCKER-ISOLATION  all  --  0.0.0.0/0            0.0.0.0/0
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0           ctstate RELATED,ESTABLISHED
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
```

The best way to manage connections and traffic between our containers is with networks. We already know that docker creates a bridge by default, which means that every container that runs in this network is free to reach each others.

We can convert our last example with wordpress

```
$ docker network create wp-test
$ docker run --name mysql-service -e MYSQL_ROOT_PASSWORD=root --network
    wp-test -d mysql
$ docker run --name my-wordpress \
    -e WORDPRESS_DB_PASSWORD=root \
    --network wp-test \
    -d wordpress
```

What changed is that we no longer have an alias but instead we can use the name of the server. This is because docker has an embedded DNS server that provides this feature. Remember that this particular feature doesn't work with the default network bridge, instead you'll need to create your network, in my case 'wp-test'.

3.11 In the end

What we saw is the Docker Engine but it is not alone, there is a big ecosystem around docker containers. Now you know what Docker is and the basics about how to interact with the daemon, we can now learn about what is available to manage our containers and our infrastructure.

4. Biography

I am Gianluca Arbezzano (@gianarb) and I work as Software Engineer in different languages: PHP, Golang, JS and so on.

I am passionate about automation and all the DevOps philosophy during my work experience I used different cloud providers like Amazon Web Service, OpenStack, Digitalocean.

I have a good experience about different web layers I worked on AngularJs to build mobile and web application but also developed backend and scalable infrastructure but I am focused right know on the last one.

Figure 4.1: Thanks Ivan Frantar for this photo.

I started my experience as developer with open source product and framework like Linux, PHP, Zend Framework, Vagrant and for this reason I am actively involved on different open source community Zend Framework, Doctrine, PHP, Golang and Docker.

I am a Docker Captain, it's a group of Docker experts and leaders in their communities who demonstrate a commitment to sharing their Docker knowledge with others (by. Docker Inc).[1] This book is a way to share my experience with the big and great community around Docker, containers and distributed systems.

You can find me on Twitter[2] and GitHub[3], they are my mainly social network, I am also a blogger[4] and an active and passionate speakers[5].

Out of the web I am a volunteer involved on different ONG and ONLUS, I like the idea to leave the world better than you found it. I love ski, play football with friend, have a chat in some nice pub with a pint, eat and cook and son on.

[1] https://www.docker.com/community/docker-captains
[2] https://twitter.com/gianarb
[3] https://github.com/gianarb
[4] http://gianarb.it/blog
[5] http://gianarb.it/conferences