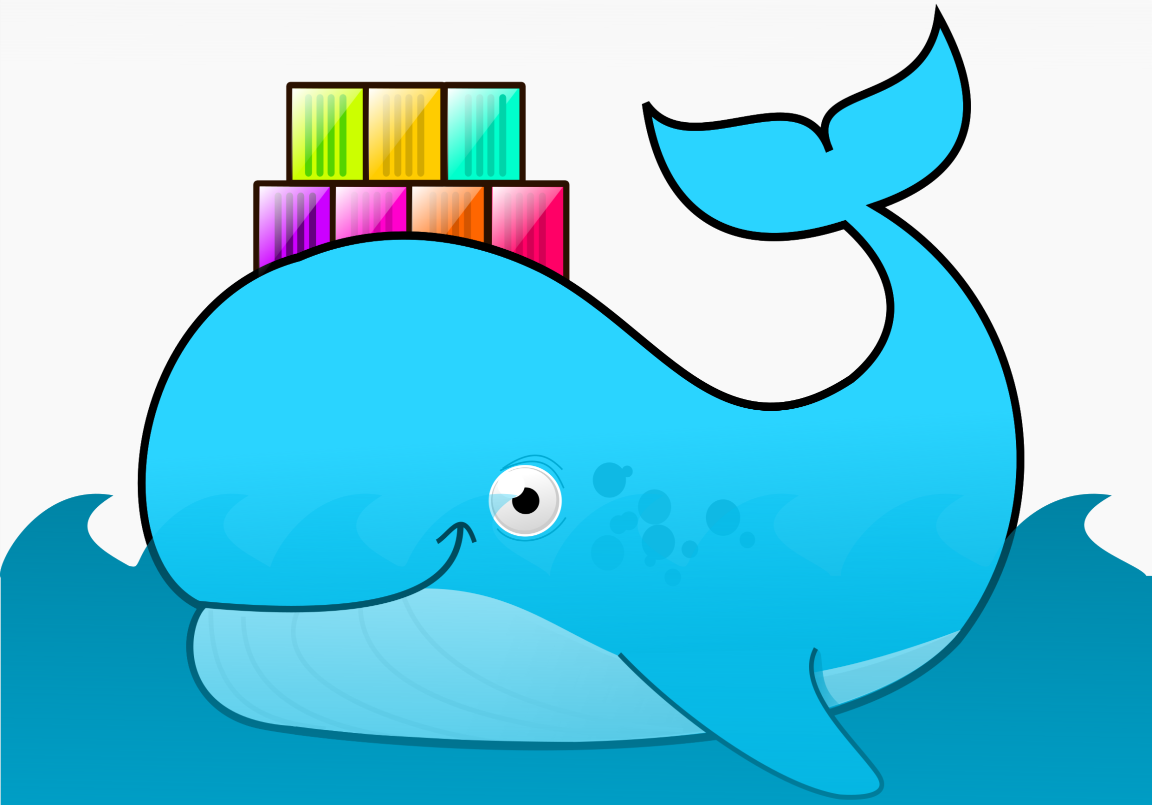
Rafael Gomes



docker for Developers

instruct

9BravoS

Docker for Developers

Rafael Gomes

This book is for sale at http://leanpub.com/docker-for-developers

This version was published on 2017-07-26

Contents

Preface

In software development it is usual to create good practices for standards. Especially for web applications, certain concepts and practices such as DevOps, cloud infrastructure, Phoenix, immutable and 12 factor apps are widely accepted theories that help on systems productivity and maintenance. While these concepts are not new, many are the tools and systems that can help to implement them. But Docker is one of the first and most commented tools and platforms that combine many of these concepts in a cohesive and simple way. As any tool, Docker is an investment that provides the best return when you understand its purpose and how to use it properly.

There are several presentations, papers and documents about Docker. However, there was the opportunity of a book connecting the theory to the practice of the tool, in which the reader could understand the motivations of Docker and also how to organize the application in order to get the best from the tool.

I am very pleased that Rafael wrote this book that I believe it's an important contribution to our field.

Rafael is extremely engaged in Docker and Devops communities in Brazil, and understands what people seek in terms of knowledge regarding this subject. In this book you will be able to understand the basics on Docker with a simple language and many practical examples.

I hope this publication turns into one more step to boost your journey. I wish you success and all the best.

Best regards,

Luís Armando Bianchin

How to read this book

This material was divided in two big parts. The first one approaches the most basic points of Docker. It is the exactly minimum necessary that a developer needs to know to use this technology properly, that is, knowing what happens exactly when executing each command.

In this first part, we will try not to approach the "low level" issues of Docker, because they are more appealing for the infrastructure team.

In case you don't know anything about Docker, we **strongly** advise you to read this first part, so you can go through the next part that focus on building a web application on Docker following the best practices, no pauses. In this book, we used the practices from 12factor[1].

The **12 factor** will be detailed in the beginning of the second part, but we can tell what we consider the "12 commandments for web applications on Docker", that is, once your application follows all the good practices presented in this document, you will be possibly using Docker at its full potential.

This second part is divided by each good practice of **12factor**. Therefore, we present a sample code in the first chapter that will evolve while the book develops. The idea is that you can practice with a real code, thus absorbing the content in a practical way. We also put together some appendices with extra-important subjects that don't fit in the following chapters.

[1] https://12factor.net/pt_br/

Acknowledgements

My first thanks go to the person who gave me the chance of being here and to be able to write this book: my mother. The famous Cigana, or Dona Arlete, a wonderful person and a role model.

I want also thank to my second mother, Dona Maria, who took so much care of me when I was a kid while Dona Arlete was taking care of her two other kids and a nephew. I feel lucky for having two moms while many don't have one.

I take this chance to thanks the person who introduced Docker to me, Robinho[2], also known as Robson Peixoto. In a conversation during the Linguágil meeting, in Salvador, Bahia, he told me: "Study Docker!" And here I am finishing a book that transformed my life. I truly thank you, Robinho!

Thanks to Luís Armando Bianchin, who started to write along with me but was not able to go on for other reasons. I'm very grateful, for your constant feedback kept me going on writing this book.

Thanks to Paulo Caroli who encouraged me to write the book and introduced me to the Leanpub platform. If it wasn't for him, this book would not be here so quick.

Thanks to the amazing Emma Pinheiro[3], for the beautiful cover. I also want to thanks a lot the incredible people from Raul Hacker Club[4], who have strongly encouraged me this whole time.

Thanks for the mother of my son, Eriane Soares, who is an amazing friend of mine and have encouraged me to write the book while we were still living together!

[2]https://twitter.com/robinhopeixoto

[3]https://twitter.com/n3k00n3

[4]http://raulhc.cc/

As every open knowledge product, this book wouldn't be possible without the help of the vibrant Docker Brazil community. I will highlight the effort of some members of reading several chapters many times and dedicating their precious time and suggesting improvements:

- Gjuniioor gjuniioor@protonmail.ch
- Marco Antonio Martins Junior[5] - Wrote the chapters "Can I run GUI applications" and "Useful commands".
- Jorge Flávio Costa[6]
- Glesio Paiva[7]
- Bruno Emanuel Silva[8]
- George Moura[9]
- Felipe de Morais[10]
- Waldemar Neto[11]
- Igor Garcia
- Diogo Fernandes[12]

Possibly, I have forgotten to mention some people here, but as I recover my logs I will updated it.

[5]https://twitter.com/somatorio
[6]https://twitter.com/JFCostta
[7]https://twitter.com/glesio
[8]https://twitter.com/jwalker_pe
[9]https://twitter.com/georgemoura
[10]https://twitter.com/felipedemorais_
[11]https://twitter.com/waldemarnt
[12]https://twitter.com/diogocfernandes

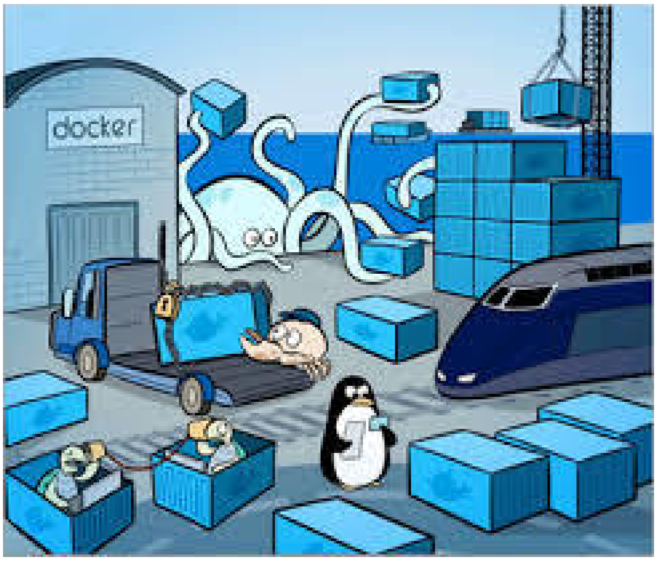
Introduction

This part of the book is for those who don’t have any basic knowledge of Docker. In case you do, don’t be shy and jump to the next part. However, even if you do know Docker, we are presenting explanations on several available resources and how they work.

Even if you are a regular Docker user, reading this part of the book at some point of your life can be important in order to know more about what happens at every executed command.

Why using Docker?

Docker has been a very commented subject lately, many articles have been written, usually talking about how to use it, auxiliary tools, integrations and the like, but many people still ask the most basic question when it's about the possibility of using any new technology: "Why should I use this?" Or would it be: "What this has to offer me that is different from what I have today?"



It is natural that people still doubt the Docker's potential, some even think that it's about some hype[13]. But in this chapter we intend to show some good reasons to use Docker.

It's important to highlight that Docker is not a "silver bullet" – it is not intended to solve all the problemas, much less being the only solution to several situations.

[13] http://techfree.com.br/2015/06/sera-que-esse-modelo-de-containers-e-um-hype/

Here are some good reasons for using Docker:

1 – Similar environments

It's worth to highlight that Docker is not a "silver bullet" – it is not intended to solve all the problemas, much less being the only solution to several situations. Once your application is turned into a Docker imagem, it can be instantiated as a container in any environment you wish. That means you can use the application at the developer's notebook as well as it would run at the production server.

The Docker image accepts parameters at the start of container, thus indicating that the same image can behave differently in distinct environments. This container can connect to its loca database for testing, using the credentials and the testing database. But when the container created from the image receives parameters from the production environment, it will access the database in a more robust infrastructure, with its respective production credentials and database, for instance.

The Docker images can be considered atomic implantations – which provides more predictability compared to other tools such as Puppet, Chef, Ansible etc. – impacting positively on errors analysis, as well as on the reliability of the continuous delivery process[14], which is strongly based on the creation of a single artefact that migrates between environments. In the case of Docker, the artefact would be the image itself with all the dependencies required to execute its code, whether compiled or dynamic.

2 – Application as a whole package

Using the Docker images makes possible to package all of your application and dependencies, making the distribution easy because

[14]https://www.thoughtworks.com/continuous-delivery

it won't be necessary to send an extent documentation explaining how to configure the required infrastructure to allow the execution, just make the image available in a repository and grant the access to the user, so the user can download the build that will be executed with no problems.

Updating is also positively affected, because the layer structure[15] of Docker allows that, in case of change, only the altered part is transferred, so the environment can be changed faster and simpler. The user only needs to execute one command to update the application image, that will be reflected on the container running on the desired moment. The Docker images can hold tags, thus making possible to store multiple versions of the same application. That means that, if there's a problem on the update, the backup plan will be basically to use the image along with the previous tag.

3 – Standardization and replication

As the Docker images are built with definition files[16], it is possible to guarantee that a given pattern will be followed, increasing the confidence on replication. You just the need the images to follow the [best practices] (https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/) of building so it is viable to escalate[17]the structure quickly.

In case a new member enters the team to work on development, he/she can get the work environment with a few commands. This process will take the time of downloading the images that are going to be used, as well as the definition files to manage them. This helps the introduction of a new member in the process of developing an application, who will be able to quickly reproduce the environment in his/her station, thus writing codes according to the team' standard.

[15]http://techfree.com.br/2015/12/entendendo-armazenamentos-de-dados-no-docker/
[16]https://docs.docker.com/engine/reference/builder/
[17]https://pt.wikipedia.org/wiki/Escalabilidade

If there's the need of testing a new version of a certain part of the solution, using Docker images, usually it's only necessary to change one or more parameters of the definition file in order to start a modified environment with the requested version to evaluation. That is: creating and modifying the infrastructure got easier and faster.

4 – Common language between infrastructure and development

The syntax used to parameterize the Docker images and environments can be considered as a common language between areas that usually don't dialogue well. Now it's possible to both sectors to make proposals and counter-proposals based on a common document. The required infrastructure is going to be present in the code of the developer and the infrastructure area will be able to analyse the document, suggesting changes to get in sync with the standards of the sector or not. All that through comments and acceptance of *merge* or *pull request* from the code version control system.

5 – Community

As it is possible to access github[18] or gitlab[19] to search code samples, using the image repository[20] of Docker makes possible to get good models of application infrastructure or services for complex integrations.

For example: nginx[21] as a reverse proxy, and mysql[22] as a database. In case the application needs these two resources, you don't have

[18]http://github.com/
[19]https://about.gitlab.com/
[20]http://hub.docker.com/
[21]https://hub.docker.com/_/nginx/
[22]https://hub.docker.com/_/mysql/

to need to waste time installing and setting up theses services. Just use the images from the repository, setting up minimum parameters for suitability to the environment. Usually the official images follow the good practices for using the services offered.

Using these images doesn't mean to "be held hostage" of their configuration, because it is possible to send you own configuration to the environments and just prevent the basic installation.

Questions

Some people sent some questions regarding the advantages we presented in this text. Thus, instead of answering them one by one, we decided to publish the questions and their answers here.

What is the difference between Docker image and definitions created by an infrastructure automation tool?

As an example of infrastructure automation tools we have Puppet[23], Ansible[24], and Chef[25]. They can guarantee similar environments, once their job is to keep a given configuration on the desired asset.

The difference between the Docker solution and the configuration management may seem very thin, for both can support the necessary configuration of every infrastructure that an application demands to be implanted, but we think that one of the most relevant differences is in the following fact: the image is a complete abstraction and doesn't require any treatment to deal with most varied GNU/Linux distributions that exit, since the Docker image comes along with a full file copy of a lean distribution.

[23] https://puppetlabs.com/
[24] https://www.ansible.com/
[25] https://www.chef.io/chef/

To carry within the copy of a GNU/Linux distribution is usually not a problem for Docker, because using the layer model saves a lot of resources by reusing the base layers. Read this article[26] to know more about Docker storage.

Another advantage of image in relation to the configuration management is that, when using the image, it is possible do make available the complete application package in a repository, and this "final product" be easily used without needing a complete configuration. Just one configuration file and one command can be enough to start an application build as a Docker image.

Still on the process of the Docker image as a product in the repository: it can also be used in the process of updating the app, as we previously explained in this chapter.

The use of the base image on Docker of a given distribution is not the same of creating a definition of a configuration management for a distribution?

No! The difference is in the host perspective. On Docker, it doesn't matter which GNU/Linux distribution is used on the host, for there is a part of the image that carries all the files from a mini-distribution that will be sufficient to support everything the app needs. In case your Docker host is Fedora and the app needs files from Debian, don't worry because this given image will bring up Debian files to support the environment. As said previously, this usually doesn't affect negatively in disk space consumption.

Does it mean that I, as a developer, have to worry about everything on Infrastructure?

No! When we say that it is possible to the developer to specify the infrastructure, we are talking about the closest layer of the

[26] http://techfree.com.br/2015/12/entendendo-armazenamentos-de-dados-no-docker/

application and not all the required architecture (Basic operational system, firewall rules, network rules etc.).

The ideia on Docker is that relevant subjects directly connected to the application can be configured by the developer. This does not obligate him/she to perform this activity. This is a possibility that pleases many developer, but in case it is not your situation you can relax, another team will deal with this part. The deploy process will get a little slower.

Many people refer to Docker for be used with microservices. Is it possible to use Docker to monolithic applications?

Yes! However, in some cases minor changes in the applications are required so it can enjoy the facilities of Docker. A common example is the log that usually the application sends to a given file, that is, in the Docker model the applications in the containers should not try to write or generate log files. Au contraire, each process in execution writes its own event flow, no buffer, to stdout[27], because Docker holds specific driver to treat the log sent this way. The subject on best practices of log manager will be approached in the next chapters.

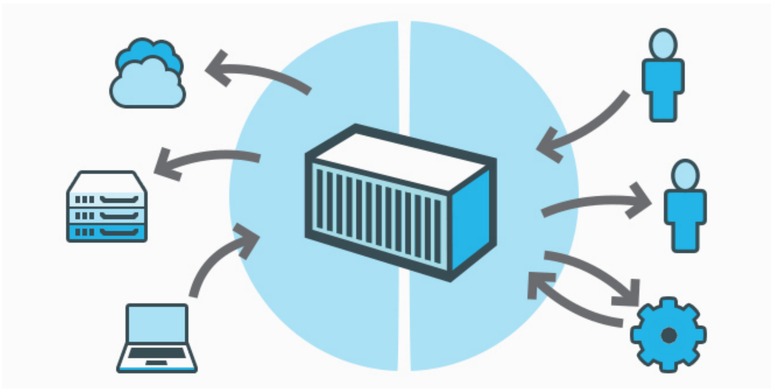
At some point you will realize that using Docker to your application demands lots of effort. In this cases, usually the problem relies in how the application works and not on the Docker configuration. Be aware of that.

Do you have more questions and/or good reasons for using Docker? Leave your comment here[28].

[27] https://pt.wikipedia.org/wiki/Fluxos_padr%C3%A3o
[28] http://techfree.com.br/2016/03/porque-usar-docker/

What is Docker?

In a very summarised way, we can say that Docker is an open platform, created with the goal of facilitating the development, deployment and execution of applications in isolated environments. It was designed especially to make an application available in the fastest way as possible.



Using Docker, you can easily manage the application infrastructure, speeding up the process of creation, maintenance and modification of your service.

The process occurs without the need of any privileged access to the corporate infrastructure. Therefore, the team responsible for the application can take part in the environment specification along with the team responsible for the servers.

Docker provided a common "language" between developers and servers administrators. This new "language" is used to build files with the definitions of the required infrastructure and to show how the application will be arranged in this environment, which port will provide the service, which data from external volumes will be requested and other possible requests.

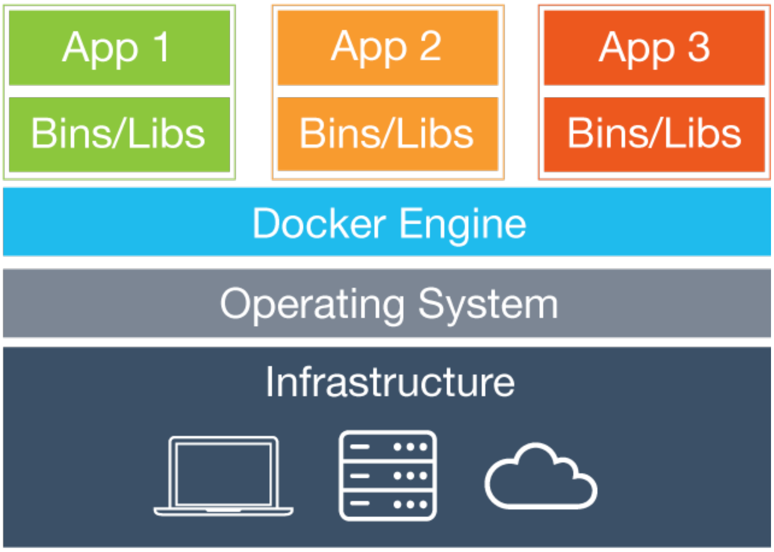
Docker also provides a public cloud to share ready environments, that can be used to enable customizations for specific environments. It is possible to get a ready image from apache and configure the specific requested modules to the applications, thus creating your own customised environment. All with a few source lines of code.

Docker uses the container model to “package” the application that, after being transformed into a Docker imagem, can be reproduced in a platform of any size; that is, in case the application runs flawlessly in your notebook, it will behave the same in the server ou mainframe. Build once, and execute wherever you want.

Containers are isolated on disk, memory, processing and network levels. This separation provides great flexibility, in which distinct environments can co-exist in the same host without any issues. It is worth to highlight that the overhead in this process is the minimum necessary, because each container usually carries only one process, that is responsible for delivering the desired service. In any case, this container also carries every file needed (configuration, library and related) for a complete isolated execution. Another interesting point on Docker is the velocity to make the desired environment feasible; as it is basically the beginning of a process and not a whole operational system, the availability time is usually counted in seconds.

Virtualization at operational system level

The isolation model in Docker is the virtualization at operational system level, a virtualization method in which the kernel of the operational system allows that multiples processes are executed separately in the same host. These running isolated processes are called containers.



To create the required isolation in the process, Docker uses the kernel functionality, called namespaces[29], that creates isolated environments between containers: the processes of a running application will not have access to the resources of another one. Unless it is expressly enabled in the configuration of each environment.

To avoid the exhaustion of machine resources due to one isolated environment, Docker uses the cgroups[30] feature from kernel. This makes possible the coexistence os different containers in the same host, without one affecting the other for overusing shared resources.

[29]http://man7.org/linux/man-pages/man7/namespaces.7.html
[30]https://en.wikipedia.org/wiki/Cgroups

Set up

Docker stopped being just a software to turn into a set of softwares: an ecosystem. In this ecosystem we have the following softwares:

- **Docker Engine**: It's the base software of the solution. It is both the daemon responsible for the containers and the client used to send commands to daemon.
- **Docker Compose**: It's the tool responsible for defining and executing multiple containers based on definition files.
- **Docker Machine**: It's the tool that enables to create and keep Docker environments in virtual machines, cloud environments and even in a physical machine.

We are not mentioning Swarm[31] and other tools because they're not lined up with the goal of this book: introduction to developers.

Setting up on GNU/Linux

We will explain the set up in the most comprehensive way, thus you can install the tools in any GNU/Linux distribution you are using.

Docker engine on GNU/Linux

To set up Docker Engine is simple. Access your GNU/Linux terminal of choice and become root user:

[31] https://docs.docker.com/swarm/overview/

```
su - root
```

or, in case of using sudo

```
sudo su - root
```

Execute the following command:

```
wget -qO- https://get.docker.com/ | sh
```

We advise you to read the script that's being executed in your operational system. Access this link[32] and analyse the code when you have the time.

This procedure takes a little while. After finishing the test, execute the following command:

```
docker container run hello-world
```

Solving possible problems

If the internet access on the machine goes through traffic control (that blocks the access to certain pages), you can find some problems while in the **apt-key** step. In case you are facing this issue, execute the following command:

```
wget -qO- https://get.docker.com/gpg | sudo apt-key add\
 -
```

Setting up Docker compose with pip

[Pip](https://en.wikipedia.org/wiki/Pip_(package_manager) is a Python package manager and, as Docker Compose is written on this language, it is possible to set it up as it follows:

[32]https://get.docker.com/

```
pip install docker-compose
```

Solving possible problems

If you don't have the command **pip** installed in your computer, usually it can be set up using your package management system called **python-pip** or similar.

Docker machine on GNU/Linux

Setting up Docker Machine is simple. Access your terminal of choice on GNU/Linux and become root user:

```
su - root
```

or, in case of using sudo

```
sudo su - root
```

Execute the command below:

```
$ curl -L https://github.com/docker/machine/releases/do\
wnload/v0.10.0/docker-machine-`uname -s`-`uname -m` > /\
usr/local/bin/docker-machine && \
chmod +x /usr/local/bin/docker-machine
```

To test, execute the command below:

```
docker-machine version
```

Obs.: The previous example uses the latest version available when this book was published. Check if there's some updated version verifying the official documentation[33].

[33] https://docs.docker.com/machine/install-machine/

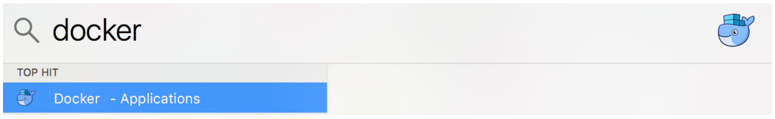
Setting up on MacOS

The installation of tools from the Docker ecosystem on MacOS is made through a single big package called **Docker for Mac**.

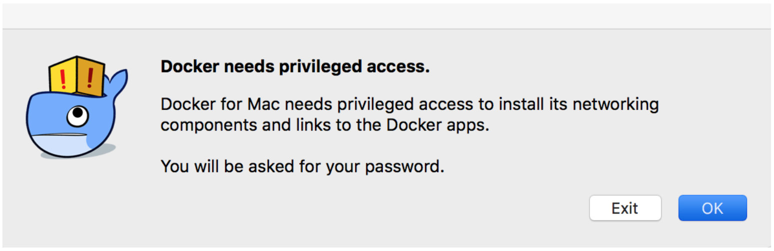
You can set it up via brew cask by following the command below:

```
brew cask install docker
```

To set up the initial configuration, you must execute the Docker application:



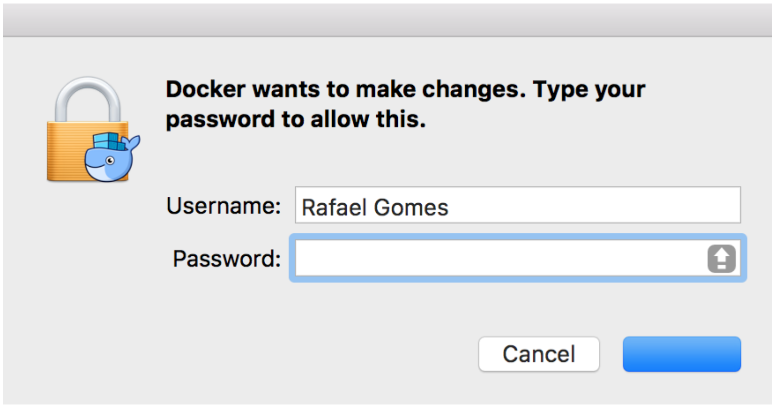
On the following screen, select the option **Ok**.


Docker needs privileged access.

Docker for Mac needs privileged access to install its networking components and links to the Docker apps.

You will be asked for your password.

Your user and password will be requested to authorize the installation of softwares. Fill them in and follow the process.


Docker wants to make changes. Type your password to allow this.

Username: Rafael Gomes

Password:

To test, open a terminal and execute the command below:

```
docker container run hello-world
```

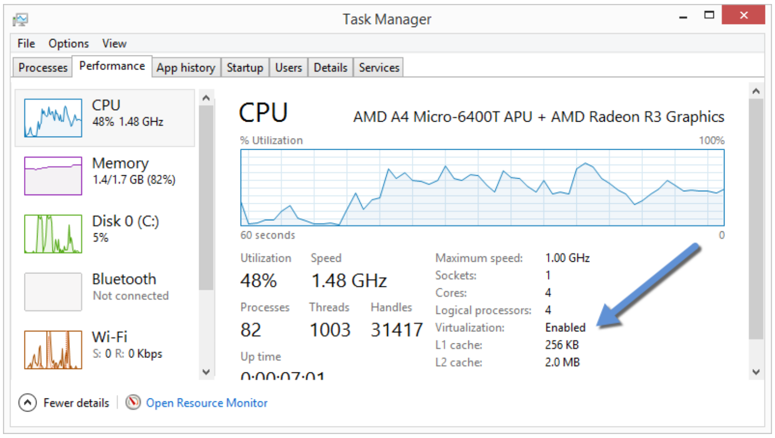
Setting up on Windows

The installation of tools of the Docker ecosystem on Windows can be made through a single big package called **Docker Toolbox**.

Docker Toolbox works only on 64 bit versions[34] of Windows and only for versions later than Windows 7.

It is important to highlight that it is necessary to enable the virtualization support. On Windows 8, it is possible to check it using the **Task Manager**. At the **Performance** tab, click on **CPU** to visualize the following window:

[34]https://support.microsoft.com/en-us/kb/827218


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
Open Resource Monitor

CPU
48% 1.48 GHz
Memory
1.4/1.7 GB (82%)
Disk 0 (C:)
5%
Bluetooth
Not connected
Wi-Fi
S: 0 R: 0 Kbps

CPU AMD A4 Micro-6400T APU + AMD Radeon R3 Graphics
% Utilization 100%
60 seconds 0

Utilization 48%
Speed 1.48 GHz
Processes 82
Threads 1003
Handles 31417
Up time 0:00:07:01

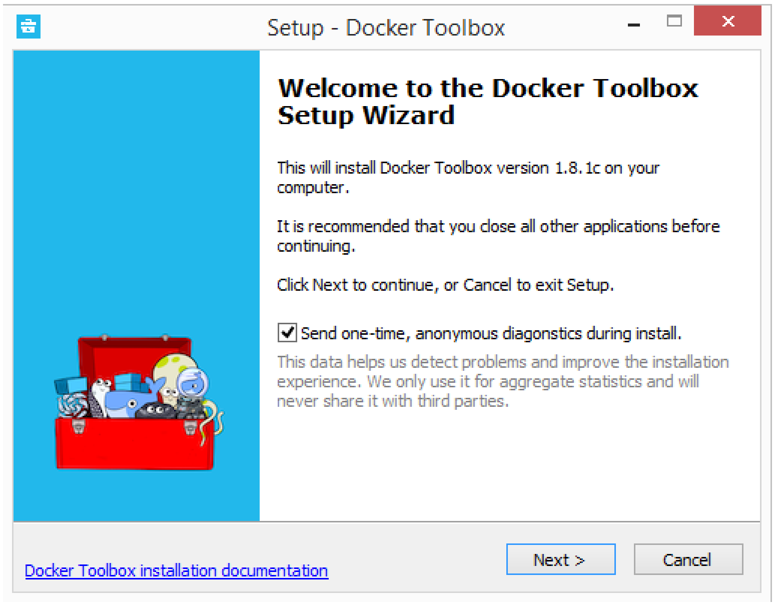
Maximum speed:	1.00 GHz
Sockets:	1
Cores:	4
Logical processors:	4
Virtualization:	Enabled
L1 cache:	256 KB
L2 cache:	2.0 MB

To check the virtualization support on Windows 7, use this link[35] to further information.

Setting up Docker Toolbox

Access the **Docker Toolbox** page[36] and download the installer for Windows. After double-clicking the installer, you will see this screen:

[35]http://www.microsoft.com/en-us/download/details.aspx?id=592
[36]https://www.docker.com/products/docker-toolbox


Setup - Docker Toolbox

Welcome to the Docker Toolbox Setup Wizard

This will install Docker Toolbox version 1.8.1c on your computer.

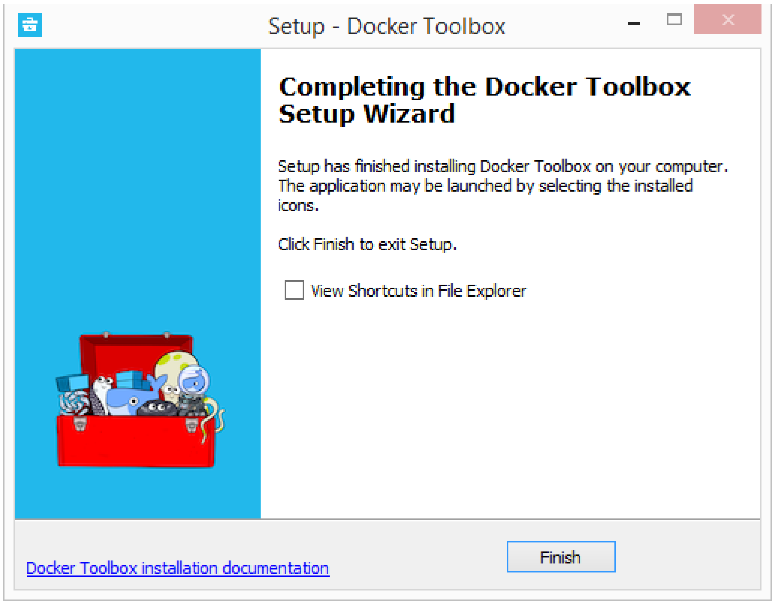
It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

[✓] Send one-time, anonymous diagonstics during install.

This data helps us detect problems and improve the installation experience. We only use it for aggregate statistics and will never share it with third parties.

Just click on **Next**.


Setup - Docker Toolbox

Completing the Docker Toolbox Setup Wizard

Setup has finished installing Docker Toolbox on your computer. The application may be launched by selecting the installed icons.

Click Finish to exit Setup.

☐ View Shortcuts in File Explorer

Lastly, click on **Finish**.

To test, search and execute the **Docker Quickstart Terminal** software; it will run all the required processes to start to use Docker.

In this new terminal, execute the following command to perform a test:

```
docker container run hello-world
```

Basic commands

For using Docker it is necessary to know a few commands and understand directly and clearly what they do, as well as some examples of use.

We are not approaching the commands for creating image and troubleshooting on Docker, because there are specific chapters on these subjects.

Running a container

To create a container is necessary to know from each image it will be executed. In order to list the images on your Docker host, execute the command below:

```
docker image list
```

The images that appear are on your **Docker host** and do not require any download from the Docker public cloud[37] unless you wish to update it. To update the image, just execute the command below:

```
docker image pull python
```

We use the image named **python** as an example, but in case you wish to update any other image, just replace **python** with your name.

In case you want to inspect the image you just updated, just use the command below:

[37]hub.docker.com

```
docker image inspect python
```

The command inspect[38] is responsible for informing every image corresponding data.

Now that the image is updated and inspected, we can start the container. But before we simply copy and paste the command, let's see how it really works.

```
docker container run <parâmetros> <imagem> <CMD> <argum\
entos>
```

The most used parameters in the container's execution are:

Parameter	Explanation
-d	Running container on background
-i	Interactive mode. Keeps the STDIN open even without console attached
-t	Allocates a pseudo TTY
–rm	Automatically removes the container after finishing (**doesn't work with -d**)
–name	Name the container
-v	Volume mapping
-p	Port mapping
-m	Limit the use of RAM memory
-c	Balance the use of CPU

Here is a simple example on the following command:

```
docker container run -it --rm --name meu_python python \
bash
```

According to the command above, a container will be created with the name **meu_python**, created from the **python** image, and the process executed in this container will be the **bash**.

[38] https://docs.docker.com/engine/reference/commandline/inspect/

It's important to remember that, in case the **CMD** is not specified in the **Docker container run** command, it's going the be used the standard value defined at the image's **Dockerfile**. In this case is **python** and its standard command executes the **python** binary, that is, if the **bash** is not specified at the end of command on the example above, instead of a shell bash of GNU/Linux it would be shown a **python** shell.

Volume mapping

To map the volume, just specify the origin of the data at the host and where it should be set inside the container.

```
docker container run -it --rm -v "<host>:<container>" p\
ython
```

The storage use is better explained in further chapters, that's why we are not detailing the use of this parameter.

Port mapping

To map the ports, you just have to know which port will be mapped on host and which one should get this connection inside the container.

```
docker container run -it --rm -p "<host>:<container>" p\
ython
```

An example with the port 80 of the host to a port 8080 inside the container holds the following command:

```
docker container run -it --rm -p 80:8080 python
```

As the command above shows, we have the port **80** accessible on **Docker host**, that passes along all the connections to the port **8080** inside the **container**. In other words, it's not possible to access the port **8080** at the IP address of **Docker host** because this port is only accessible inside the container, which is isolated at the network level, as said previously.

Managing resources

While creating the containers it is possible to specify some limits on using resources. We will discuss here the most used – RAM memory and CPU.

To limit the use of RAM memory that can be used by this container, just execute the command below:

```
docker container run -it --rm -m 512M python
```

With the command above we are limiting this container to use only 512MB of RAM.

To balance the use of CPU by the containers, we specify weights for each container; the lighter the weight, the less priority on use. The weights can oscillate from **1** to **1024**.

In case the weight of the container is not specified, it will use the heaviest weight possible, in this case, **1024**.

We will use the weight **512** as an example:

```
docker container run -it --rm -c 512 python
```

To better understand, let's imagine that three containers are running. One of them has the standard weight **1024** and the other two have **512**. In case the three processes require the whole CPU, their time of use will be divided as it follows:

- The process weighting **1024** will use 50% of the processing time.
- The two processes weighting **512** will use 25% of the processing time, each.

Checking the list of containers

To visualize the list of containers of a given **Docker host**, we use docker ps[39].

This command is responsible for displaying all containers, even those not running anymore.

```
docker container list <parâmetros>
```

The most used parameters in running a container are:

Parameter	Explanation
-a	Lists all containers, including turned offs
-l	Lists the last containers, including turned offs
-n	Lists the last N containers, including turned offs
-q	Lists only the containers' ids, great for using on scripts

[39] https://docs.docker.com/engine/reference/commandline/ps/

Managing containers

Once the container is created from an image, it is possible to manage the usage with new commands.

In case you wish to turn off the container, just use the command docker stop[40]. It gets as an argument the container **ID** or **name**. Both data can be obtained through **docker ps**, explained in the previous topic.

An example:

```
docker container stop meu_python
```

In the command above, in case there was a container named **meu_python** running, it would receive a **SIGTERM** signal and, if it was not turned off, it would receive a **SIGKILL** after 10 seconds.

In case you wish to restart the container that was turned off and not create a new one, just execute the command docker start[41]:

```
docker container start meu_python
```

> It's important to emphasize that the idea of containers is to be disposable. In case you use the same container for a long period without discarding it, you are probably using Docker wrongly. Docker is not a machine, is a running process. And, as every process, it must be discarded so another one can take its place when reseting.

[40] https://docs.docker.com/engine/reference/commandline/stop/
[41] https://docs.docker.com/engine/reference/commandline/start/

Creating your own image on Docker

Before we explain how to create your image, it's important to bring up a question that usually confuses Docker beginners: "Image or container?"

What's the difference between Image and Container?

Making a parallel with the concept of object orientation[42], **image** is the class and **container** is the object. The image is the infrastructure abstraction with reading only status, from where the container is going to be instantiated.

Every container is created from an image; thus we can conclude that we will never have an image running.

A container can only be created from a single image. In case you require a different behavior, it will be necessary to customize the image.

Anatomy of an image

Images can be official and non-official.

Official and non-official images

The official Docker images are those with no users in their names. The image "**Ubuntu:16.04**" is official; on the other hand, the image

[42] https://pt.wikipedia.org/wiki/Orienta%C3%A7%C3%A3o_a_objetos

"nuagebec/ubuntu"[43] is non-official. This second image belongs to user nuagebec[44], who keeps other non-official images.

Official images are maintained by Docker and are available[45] at the Docker cloud.

The goal of the official images is to promote a basic environment (i.e. debian, alpine, ruby, python), a starting point for users to create images, as we will explain further ahead in this chapter.

The non-official images are kept by users who created them. We will talk about sending images to the Docker cloud on another topic.

Name of the image

The name of an official image is formed by two parts. The first one is called "**repository**" according to the documentation[46], and the second one is called "**tag**". In the example of image "**ubuntu:14.04**", **ubuntu** is the repository and **14.04** is the tag.

For Docker, the "**repository**" is an abstraction of the image set. Don't mistake it by the image storage, that we will approach later. The "**tag**" is an abstraction to create unity inside the image set determined in the "**repository**".

A "**repository**" can contain more than one "**tag**" and each set repository:tag represents a different image.

Execute the command[47] below to visualize all images that are found locally in your station at this exact moment:

```
docker image list
```

[43]https://hub.docker.com/r/nuagebec/ubuntu/
[44]https://hub.docker.com/u/nuagebec/
[45]https://hub.docker.com/explore/
[46]https://docs.docker.com/engine/userguide/containers/dockerimages/
[47]https://docs.docker.com/engine/reference/commandline/images/

How to create images

There are two ways of creating customized images: using **commit** and using **Dockerfile**.

Creating images with commit

It is possible to create images executing the commit[48] command, related to a container. This command uses the current status of the chosen container and creates the image based on it.

Let's see the example. First, we create a container:

```
docker container run -it --name containercriado ubuntu:\
16.04 bash
```

Now that we are at the container bash, we install the nginx:

```
apt-get update
apt-get install nginx -y
exit
```

We stop the container using the command below:

```
docker container stop containercriado
```

Now, we **commit** this **container** into an **image**:

```
docker container commit containercriado meuubuntu:nginx
```

[48] https://docs.docker.com/engine/reference/commandline/commit/

In the previous example, **containercriado** is the name of the container created and altered in the previous steps; the name **meuubuntu:nginx** is the **commit's** resulting image; the status of **containercriado** is stored in an image called **meuubuntu:nginx** which, in this case, holds the only alteration we have in the official image of ubuntu on version 16.04 – the **nginx** package installed.

To see the image list and find the one you just created, execute the command below again:

```
docker image list
```

To run a test in you new image, let's creat a container from it and check if the nginx is installed:

```
docker container run -it --rm meuubuntu:nginx dpkg -l n\
ginx
```

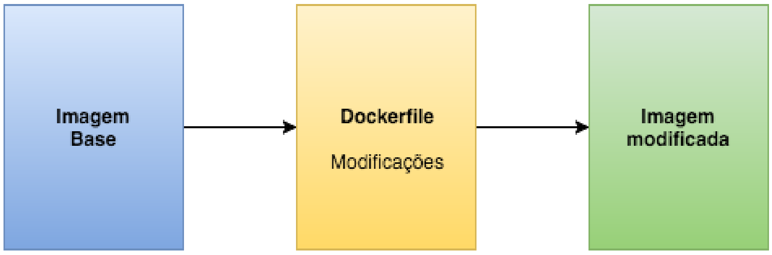
If you want validation, run the same command on the official image of ubuntu:

```
docker container run -it --rm ubuntu:16.04 dpkg -l nginx
```

> It's worth to emphasize that the **commit** method is not the best option to create images, because, as we saw, the process of altering the image is completely manual and presents a little difficulty to track the alterations made, once the manual changes are not registered automatically in the Docker structure.

Creating images with Dockerfile

When you use Dockerfile to create an image, basically you are presented a list of instructions that will be applied in a given image to the other one is created based on alterations.



Dockerfile

We can summarize this by saying that the Dockerfile file, in fact, represents the exact difference between a given image, that we call **base**, and the image you want to create. In this model, we hold total traceability on what is going to be modified in the new image.

Let's go back to the example on the nginx installation on ubuntu 16.04.

First, create a file for a future test:

```
touch arquivo_teste
```

Create a file called **Dockerfile** and insert the following content on it:

```
FROM ubuntu:16.04
RUN apt-get update && apt-get install nginx -y
COPY arquivo_teste /tmp/arquivo_teste
CMD bash
```

In the file above, we used four instructions[49]:

FROM to inform which image we will use as a base; in this case, **ubuntu:16.04**.

[49] https://docs.docker.com/engine/reference/builder/

RUN to inform which commands are going to be executed in this environment to make the necessary changes in the system's infrastructure. They're like commands executed at the environment' shell, just like the commit model, but in this case done automatically and it is completely traceable, since this Dockerfile will be store in the system at the version control.

COPY is used to copy files from the station where the building is being executed into the image. We use a test file just to exemplify this possibility, but this instruction is much used to send environment configuration files and codes to be executed in application services.

CMD to inform which command will be executed as a standard, in case none is informed by the container initialization from this image. In the example, we put the command bash; if this image is being used to initiate a container and we don't inform the command, it will execute the bash.

After building your Dockerfile, just execute the command[50] below:

```
docker image build -t meuubuntu:nginx_auto .
```

Such command has the option "**-t**", that also works to inform the name of the image that is going to be created. In this case, is going to be **meuubuntu:nginx_auto** and the "." at the end, informing which context must be used in this image building. All files from the current folder will be send to the Docker service and only they can be used to manipulations on Dockerfile (example of using COPY).

The sequence matters

It is important to notice that the **Dockerfile** file is a sequence of instructions read from the top to the base, and each line is executed at a time. If any instructions depend on another one, this dependency must be described earlier in the document.

[50]https://docs.docker.com/engine/reference/commandline/build/

The result of each file instruction is stored in the local cache. If the **Dockerfile** is not modified in the next image creation (**build**), the process won't take long, for everything will be in the cache. If there's any alterations, just the modified instruction and the next will be executed again.

The suggestion to better use the **Dockerfile** cache is to keep instructions often altered next to the base of the document. It's important to remember to attend the dependencies between instructions as well.

An example to make it clear:

```
FROM ubuntu:16.04
RUN apt-get update
RUN apt-get install nginx
RUN apt-get install php5
COPY arquivo_teste /tmp/arquivo_teste
CMD bash
```

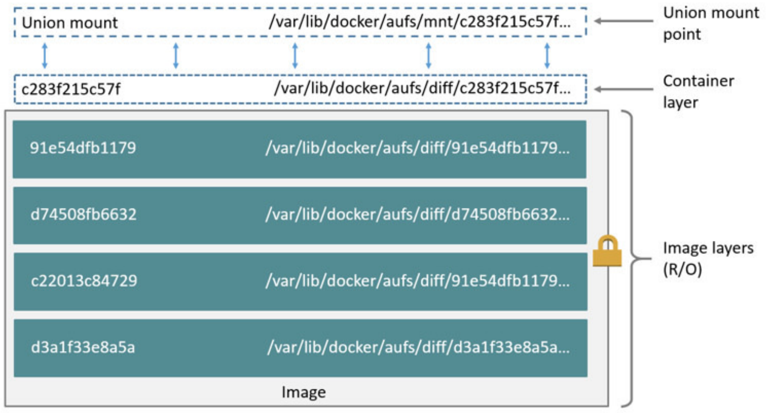
If we modify the third line of the file and, instead of installing nginx, we change it to apache2, the instruction that updates on apt will not be executed again, but rather the installation of apache2, because it just entered the file, as well as the php5 and the file copy, because all of them are subsequent to the modified line.

As we noticed, holding the **Dockerfile** file enables us to have the exact notion of which changes were made in the image, thus recording the modifications in the version control system.

Sending your image to the cloud

Understanding storage on Docker

To understand how Docker manages its volumes, first we need to explain how does it work at least a Docker storage backend[51]. We will do this here with the AUFS, that was the first one and still is the standard in a good part of Docker installations.


Union mount
/var/lib/docker/aufs/mnt/c283f215c57f...
Union mount point
c283f215c57f
/var/lib/docker/aufs/diff/c283f215c57f...
Container layer
91e54dfb1179
/var/lib/docker/aufs/diff/91e54dfb1179...
d74508fb6632
/var/lib/docker/aufs/diff/d74508fb6632...
c22013c84729
/var/lib/docker/aufs/diff/91e54dfb1179...
d3a1f33e8a5a
/var/lib/docker/aufs/diff/d3a1f33e8a5a...
Image
Image layers (R/O)

How does it work a Docker backend (ex.: AUFS)

Storage backend is the part of Docker that takes care of data management. On Docker there are several possibilities of storage backend, but here we'll only talk about the one that deploys the AUFS.

AUFS[52] is a unification file system. It is responsible for managing multiple directories, stacking them up, and provide a single and

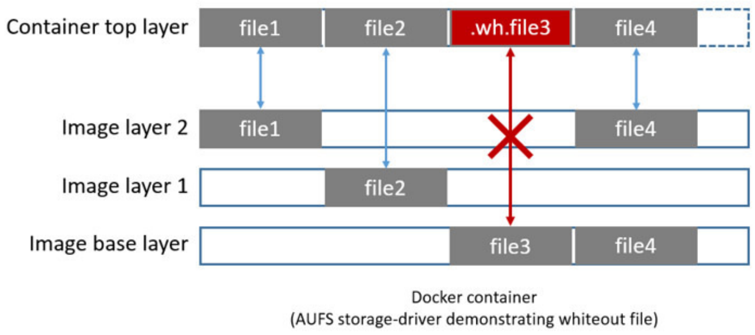
[51] http://searchdatacenter.techtarget.com/definition/back-end
[52] https://en.wikipedia.org/wiki/Aufs

unified view, as if they all together were only one directory.

This single directory is used to present the container and works as if it was a single common file system. Each directory used in the stack corresponds to one layer. And that's how Docker unifies them and provides reutilization amongst containers. Because the same directory that corresponds to the image can be set up in several stacks of several containers.

Aside the folder (layer) that corresponds to the container, every other one is set up with read only permission; otherwise, the changes in a container could interfere on another one. And this really goes totally against the principles of Linux Container.

In case it is necessary to modify a file on the layers (folders) referring to the images, the technology Copy-on-write[53] (CoW) is used, and it's responsible for copying the file to the folder (layer) of the container and to do all the modifications on this level. Thus, the original file of the inferior layer is superimposed in this stack, that is, the given container will always see only the files of the highest layers.



Removing a file

In case of removal, the file of the superior layer is marked as whiteout file, enabling the visualization of the file of lower layers.

[53] https://en.wikipedia.org/wiki/Copy-on-write

Performance issues

Docker takes advantage of the AUFS's Copy-on-write (CoW) technology to allow image sharing and the use of disk space. AUFS works at file level. This means that all AUFS CoW operations will copy whole files even if only a little part of the file is being altered. This behavior can significantly impact the container's performance, especially if the copied files are big and are located below several image layers. In this case, the procedure copy-on-write will spend a lot of time to make an internal copy.

Volume as a performance solution

By using volumes, Docker sets up this folder (layers) in the level immediately below the container, allowing fast access of all stored data in this layer (folder), solving the performance problem.

The volume also solves matters of data endurance, for the information stored in the container layers (folder) are lost while removing the container, that is, by using volumes we have a bigger guarantee in storing these data.

Using volumes

Mapping of a specific host folder

In this model the user choses a specific folder of the host (ex.: /var/lib/container1) and maps it into a container's inside folder (ex.: /var). What is written in the folder /var of the container is also written in the folder /var/lib/container1 of the host.

Here's the sample of the command used to this mapping model:

```
docker container run -v /var/lib/container1:/var ubuntu
```

This model is not portable. It needs the host to have a specific folder so the container works properly.

Mapping via data container

In this model, a container is created and, inside it, a volume to be consumed by other containers is named. Therefore, it's not necessary to create a specific folder on the host to persist data. This folder is automatically created inside the root folder of Docker daemon. However, you don't need to worry with it, since all the reference is going to made for the container that holds the volume and not for the folder.

Here's an example of using the mapping model::

```
docker create -v /dbdata --name dbdata postgres /bin/tr\
ue
```

In the command above, we created a data container, in which the folder /dbdata can be used by other containers, that is, the content of the folder /dbdata can be visualized and/or edited by other containers.

To consume this container volume, just use the command:

```
docker container run -d --volumes-from dbdata --name db\
2 postgres
```

Now the container db2 has a folder /dbdata that is the same as the one from the container dbdata, making this model completely portable.

A disadvantage is the need of keeping a container just to this, because in some environment containers are removed with some frequency, making it necessary to take special care with special containers. In a certain way, it's an additional management problem.

Mapping volumes

In Docker's 1.9 version, the possibility of creating containers' isolated volumes was added. Now it's possible to create a portable volume, with no need of associating it to a special container.

Here's an example of using the mapping model:

```
docker volume create --name dbdata
```

In the command above, Docker created a volume that can be consumed by any container.

The association of the volume to the container happens in similar way to the one practiced on mapping the host folder, because in this case you need to associate the volume to a folder inside the container, as we can see below:

```
docker container run -d -v dbdata:/var/lib/data postgres
```

This model is the most indicated since the release, because it gives you portability. It is not removed easily when this container is deleted and, still, is very easy to manage.

Understanding the network on Docker

What Docker calls network, in fact, is an abstraction created to ease the data communication management between containers and untie the external knots of the Docker environment.

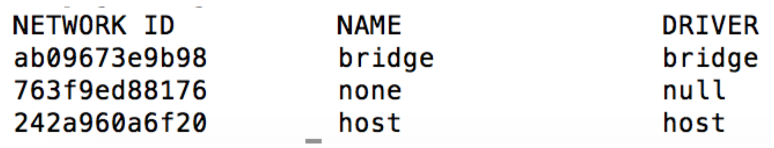
Don't mistake the Docker network with the already known network used to group the IP addresses (ex.: 192.168.10.0/24). Therefore, every time we mention this second type of network, we'll use "IP network".

Standard networks on Docker

Docker has three standard networks. These networks offer specific configurations to manage data usage. To visualize these interfaces, just use the command below:

docker network ls

The return is going to be:



```
NETWORK ID          NAME                DRIVER
ab09673e9b98        bridge              bridge
763f9ed88176        none                null
242a960a6f20        host                host
```

Bridge

Each container created on Docker is associated to an specific network. This is the standard network to any container, unless we associate, explicitly, another network to it. The network gives to the container an interface that makes a bridge with the docker0 interface of the Docker host. This interface receives, automatically, the next address available in the IP network 172.17.0.0/16.

All containers in this network can communicate via TCP/IP protocol. If you know which is the IP address of the container you wish to connect, it is possible to send data to it. After all, they are all in the same IP network (172.17.0.0/16).

A detail worth noticing: as the IPs are assigned automatically, it'n not a trivial task to discover which is the IP of the destination container. To help on this location, Docker provides, at the moment of creating a container, the option "-link".

> It's important to emphasize that "-link" is an outdated option and its use is discouraged. We'll explain this feature only for understanding the legacy. This function was replace with a built-in DNS on Docker and it doesn't work for Docker standard networks, only for networks created by the user.

The option "-link" is responsible for associating the destination container IP to its name. In case you create a container from the Docker image of the mysql with the name "bd", then create another with the name "app" from the image tutum/apache-php, you wish that the later container can connect on mysql using the name of the container "bd", just create both containers the same way:

```
docker container run -d --name bd -e MYSQL_ROOT_PASSWOR\
D=minhasenha mysql

docker container run -d -p 80:80 --name app --link db t\
utum/apache-php
```

After executing the commands, the container named "app" will be able to connect to the mysql container using the name "bd", that is, every time it try to access the name "bd", it will be automatically resolved to the IP of the IP network 172.17.0.0/16 that the mysql container got on its creation.

To test it, we'll use the function exec to run the command inside an existent container. For this, we will use the name of the container as a parameter of the command below:

```
docker container exec -it app ping db
```

The action will be responsible for executing the command "ping db" inside the container "app", that is, the container "app" will send icmp packages, usually used to test the connectivity between two hosts, to the "db" address. The name "db" is translated to the IP the container - created from the image mysql - got when created.

Example: The "db" container created firstly and got the IP 172.17.0.2. The "app" container started next and got the IP 172.17.0.3. When the "app" container executes the command "ping db", in fact, it will send icmp packages to the 172.17.0.2 address.

> Caution: The name of the option "-link" causes some confusion because it doesn't create any IP network link between containers, once the communication between them is already possible, even if the option link is not configured. As we cleared up in the previous paragraph, it just facilitates the translation of names to the dynamic IP received in initialization.

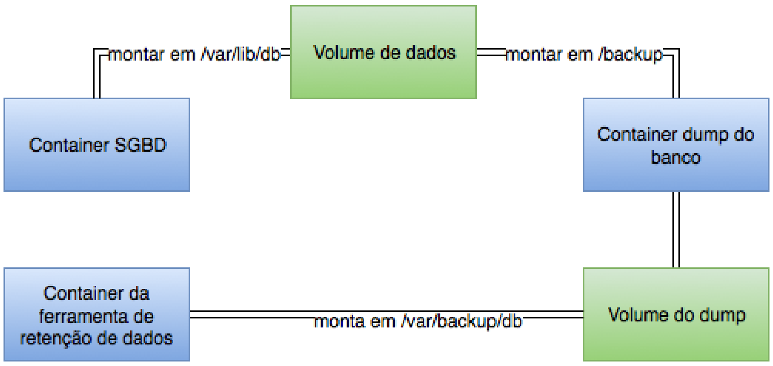
The containers configures to this network will have the opportunity of external traffic using the routes of the IP networks defined on Docker host. In case Docker host has internet access, automatically, the containers will also have.

In this network is possible to display containers ports to all the actives that have access to the Docker host.

None

This network aims to isolate the container regarding external communications. The network doesn't get any interface to external communication. The only interface of the IP network will be the localhost.

This network, usually, is used for containers that only manipulate files, with no need of sending them to another place using the network (Ex.: backup container uses the volumes of the database container to do the dump and will be used in the data retention process.



Exemplo de uso da rede none

If you have any questions on using volumes on Docker, check out this article[54] and see more on Docker storage.

Host

This network has the objective of delivering into the container all the interfaces existent on Docker host. In a way, it can speed up the package delivery, once there's no bridge on the way of the messages. But usually this overhead is minimum and the use of a bridge can be important to security and management of traffic.

[54]http://techfree.com.br/2015/12/entendendo-armazenamentos-de-dados-no-docker/

Networks set by user

Docker allows the user to create networks. These networks are associated to the element the Docker calls network driver.

Each network created per user must be linked to a given driver. And in case you didn't create your own driver, you must choose amongst the drivers provided by Docker:

Bridge

This is the network driver more simple to use, for requires little configuration. The network created by the user using the bridge driver is similar to the Docker standard network named "bridge".

> One more point that deserves attention: Docker has a standard network called "bridge" that uses a driver also called "bridge". Maybe, because of this, the confusion only gets bigger. But it is important to make clear that they are distinct.

The networks created by the user with the bridge driver have all features described in the standard network, called bridge. However, it has additional features.

Amongst one of the features: the network created by the user doesn't need to user the old "-link" option. Because every network created by the user with the bridge driver will be able to user the Docker internal DNS that automatically associates every container names of this network to its respective IPs from the corresponding IP network.

To make it clearer: the containers using the standard bridge network will not be able to enjoy the Docker internal DNS feature. In case you are using this network, it is necessary to specify the "-link" legacy for translating the names in IP addresses dynamically allocated on Docker.

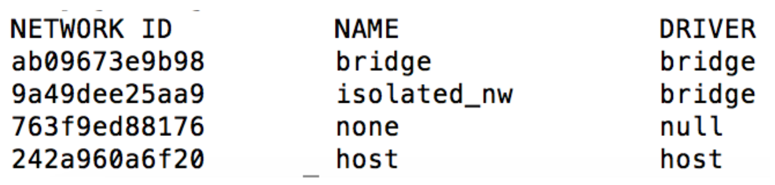
To exemplify the usage of the network created by user, let's create the network called isolated_nw with the bridge driver:

```
docker network create --driver bridge isolated_nw
```

Now, we verify the network:

```
docker network list
```

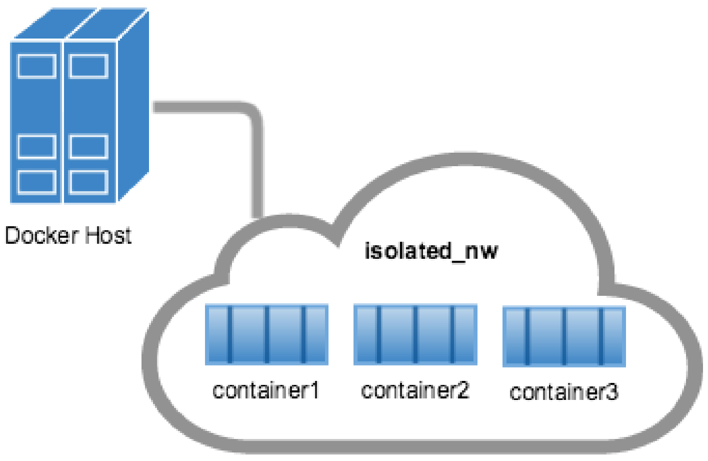
The result must be:



```
NETWORK ID          NAME                DRIVER
ab09673e9b98        bridge              bridge
9a49dee25aa9        isolated_nw         bridge
763f9ed88176        none                null
242a960a6f20        host                host
```

Now, we create a container at the isolated_nw network:

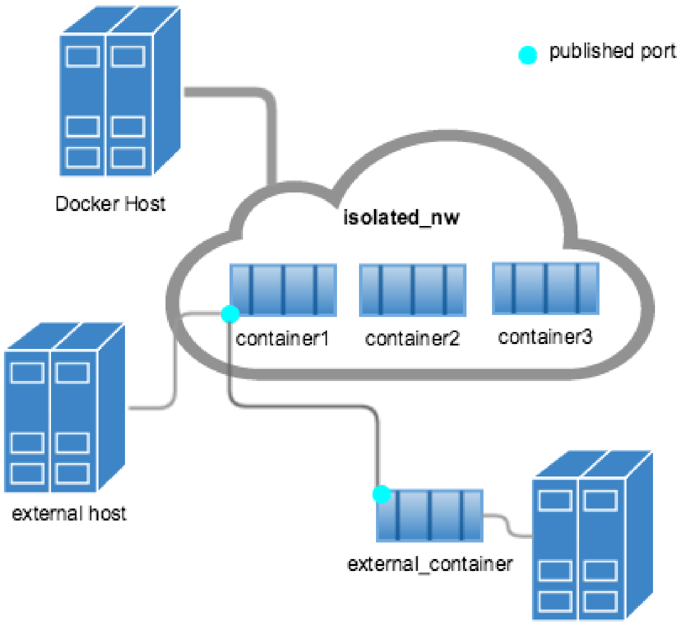
```
docker container run -itd --net isolated_nw alpine sh
```



Rede isolada

It's important to emphasize: a container that is in a given network doesn't access another container that is in another network. Even if you know the destination IP. For one container to access another in another network, it's necessary that the origin is present in both network that you wish to reach.

The containers in the isolated_nw network can expose their ports on Docker host and these ports can be accessed both by containers outside the network - the isolated_nw - and external machines with access to Docker host.



Rede isolada publicando portas

Para descobrir quais containers estão associados a uma determinada rede, execute o comando abaixo:

```
docker network inspect isolated_nw
```

The result must be:

```
[
    {
        "Name": "isolated_nw",
        "Id": "9a49dee25aa984beb923d41aab887459f059b47e71c558a8ccc38edeb4e12c7f",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1/16"
                }
            ]
        },
        "Containers": {
            "3b6b476a77c14249bed8344f5f75b47543c2d65f0ab399235d8b5a887ac33a5f": {
                "Name": "amazing_noyce",
                "EndpointID": "b92bf296fb368b686320470a8feb0d7398a4a33358646983831160dfcc06b9db",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {}
    }
]
```

In the section "Containers" it is possible to check which containers are part of this network. All containers that are in the same network will be able to communicate using only their respective names. As we can see in the example above, if a new container access the isolated_nw network, it will be able to access the amazing_noyce container using only its name.

Overlay

The overlay driver allows the communication between Dockers hosts; by using it, the containers of a given Docker host will be able to access, natively, containers from another Docker environment.

This driver require a more complex configuration, therefore, we'll approach the details in some other opportunity.

Using networks on Docker Compose

The subject deserves a whole paper for itself. So, we'll just show an interesting link[55] for furthers references on the subject.

Concluding

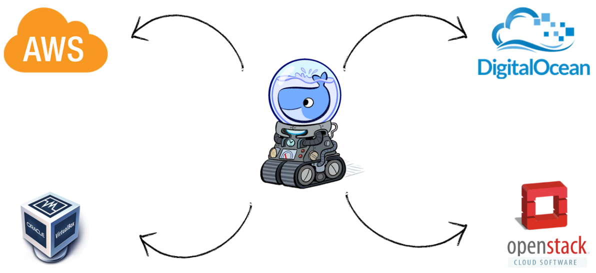
We realize that the use of networks defined by the user make the option "-link" obsolete, as well as provides a new internal Docker DNS service, making it easy for those who want to keep a big and complex Docker infrastructure, as well as provide the network isolation of its services.

To know and to use well the new technologies is a good practice that avoids future problems and facilitates building and maintaining big and complex projects.

[55]https://docs.docker.com/compose/networking/

Using Docker in multiple environments

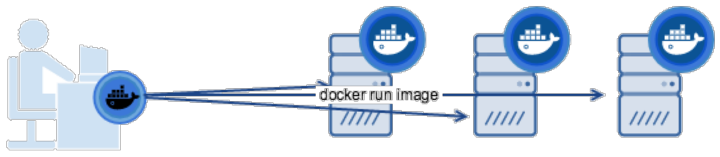
Docker host is the name of the active responsible for managing Docker environments; in this chapter we will demonstrate how is it possible to create them and manage them in distinct infrastructures, such as virtual machines, cloud, and physical machine.



Docker machine[56] is the tool used for this distributed management, and allow the docker hosts installation and management in an easy and direct way.

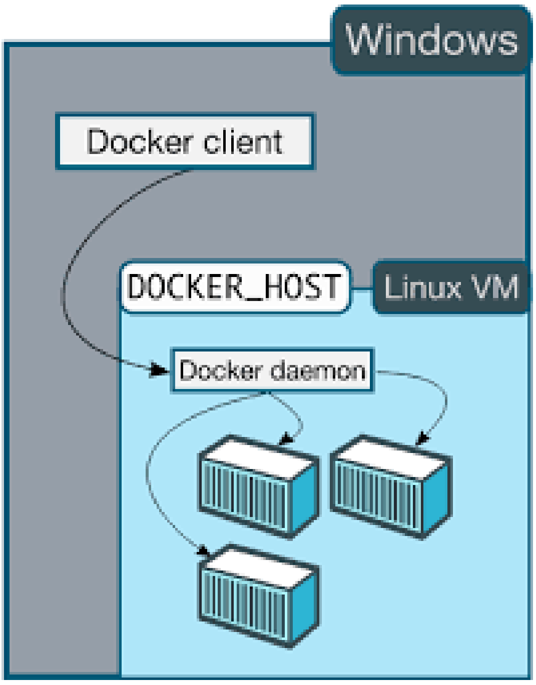
This tool is very much used by users of the "non Linux" operational systems, as we will show, but its function is not limited to that, for is also much used to provide and manage the Docker infrastructure on cloud, such as AWS, Digital Ocean and Openstack.

[56]https://docs.docker.com/machine/



How it works

Before we explain how to use the Docker machine, we need to reinforce the knowledge on Docker architecture.



As the picture above shows, the usage of Docker is divided in two services: the one that runs in daemon mode, in background, called **Docker Host**, responsible for viabilization of containers on the kernel Linux; and the client, that we'll call **Docker client**, responsible for getting commands from the user and translating them into management of **Docker Host**.

Each **Docker client** is configured to connect itself to a given **Docker host** and, at this moment, **Docker machine** takes the action, for it enables the automatization of access configuration choice of Docker client to distinct **Docker hosts**.

The **Docker machine** enables the use of several different environments just changing the client configuration to the desired **Docker host**: basically, modify some environment variables. Here's the example:

```
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/gomex/.docker/machine/m\
achines/default"
export DOCKER_MACHINE_NAME="default"
```

Modifying these four variables, the Docker client will be able to use a different environments rapidly and with no need of restart any service.

Creating environment

The Docker machine is good mainly for creating environments that in the future will be managed by it in the automatized exchange of configuration context, through the modification of environment, as we explained previously.

To create the environment, it is necessary to verify if the infrastructure you wish to create has some driver that supports this process. Here is the list of available drivers[57].

[57] https://docs.docker.com/machine/drivers/

Virtual machine

For this example, we will use the most common driver, the virtualbox[58], that is, we need a virtualbox[59] installed in our station so this driver works properly.

Before creating the environment, let's understand how does it work the creation command on Docker machine:

docker-machine create –driver=<nome do driver> <nome do ambiente>

Regarding the driver **virtualbox**, we have a few parameters that can be used:

Parameter	Explanation
–virtualbox-memory	Specifies the amount of RAM memory that the environment can use. The standard value is 1024MB (always in MB).
–virtualbox-cpu-count	Specifies the amount of CPU cores that this environment can use. The standard value is 1.
–virtualbox-disk-size	Specifies the size of the disk that this environment can use. The standard value is 20000MB (always in MB).

As a test we run the following command:

```
docker-machine create --driver=virtualbox --virtualbox-\
disk-size 30000 teste-virtualbox
```

The result of this command is the creation of a virtual machine at

[58] https://docs.docker.com/machine/drivers/virtualbox/
[59] https://www.virtualbox.org/

the virtual box. The machine will have 30GB of disk space, 1 core and 1GB RAM memory.

To make sure the process happened as expected, just use the following command:

```
docker-machine ls
```

The command above is responsible for listing all the environments that can be used from the client's station.

To change clients, just run the command:

```
eval $(docker-machine env teste-virtualbox)
```

Executing the command **ls** makes possible to verify which environment is active:

```
docker-machine ls
```

Initiate a test container to run a test on the new environment:

```
docker container run hello-world
```

If you want to change environments, just type the command below, using the name of the desired environment::

```
eval $(docker-machine env <ambiente>)
```

If you want to turn off the environment, use the command:

```
docker-machine stop teste-virtualbox
```

If you want to initiate the environment, use the command:

```
docker-machine start teste-virtualbox
```

If you want to remove the environment, use the command::

```
docker-machine rm teste-virtualbox
```

Known troubleshooting: in case you're using Docker machine on MacOS and for some reason the station sleeps when the virtual box environment initiates, it is possible that, when it comes back, the Docker host presents some internet communication problems. We advise to, every time you go through connectivity problems on Docker host with virtual box driver, turn off the environment and reinitiate as a solution.

Cloud

For this example we will use the most common cloud driver, AWS[60]. For such, we need an AWS account to this driver[61] works properly.

It is required that your credentials are in the file ~/.aws/credentials as it follows:

```
[default]
aws_access_key_id = AKID1234567890
aws_secret_access_key = MY-SECRET-KEY
```

In case you don't want to put these information on file, you can specify them via environment variables:

[60]http://aws.amazon.com/

[61]https://docs.docker.com/machine/drivers/aws/

```
export AWS_ACCESS_KEY_ID=AKID1234567890
export AWS_SECRET_ACCESS_KEY=MY-SECRET-KEY
```

You can find more information on AWS credentials in this article[62].

When we create an environment using the command **docker-machine create**, is is translated into AWS in the creation of a EC2 instance[63] and then it is automatically installed in every required software in the new environment.

The most used parameters in the creation of this environment are:

Parameter	Explanation
–amazonec2-region	Says what AWS region is used to host your environment. The standard value is us-east-1.
–amazonec2-zone	It's the letter that represents the region used. The standard value is "a".
–amazonec2-subnet-id	Says which sub-network is used in this EC2 instance. Needs to be created previously.
–amazonec2-security-group	Says which security group is used in this EC2 instance. Needs to be created previously.
–amazonec2-use-private-address	It will be created an interface with private IP, because as default it only specifies an interface with public IP.

[62]http://blogs.aws.amazon.com/security/post/Tx3D6U6WSFGOK2H/A-New-and-Standardized-Way-to-Manage-Credentials-in-the-AWS-SDKs

[63]https://aws.amazon.com/ec2/

Parameter	Explanation
–amazonec2-vpc-id	Says which VPC ID is desired for this EC2 instance. Needs to be created previously.

As an example, we will use the following command while creating the environment:

```
docker-machine create --driver amazonec2 --amazonec2-zo\
ne a --amazonec2-subnet-id subnet-5d3dc191 --amazonec2-\
security-group docker-host --amazonec2-use-private-addr\
ess --amazonec2-vpc-id vpc-c1d33dc7 teste-aws
```

After running the command, just wait it to finish up; it is common to take a while..

To test the success of the action, execute the command below:

```
docker-machine ls
```

Check if the environment called teste-aws exists in the list; if so, use the command below to change the environment:

```
eval $(docker-machine env teste-aws)
```

Create a test container to verify the new environment:

```
docker container run hello-world
```

If you want to turn off the environment, use the command:

```
docker-machine stop teste-aws
```

If you want to start the environment, use the command:

```
docker-machine start teste-aws
```

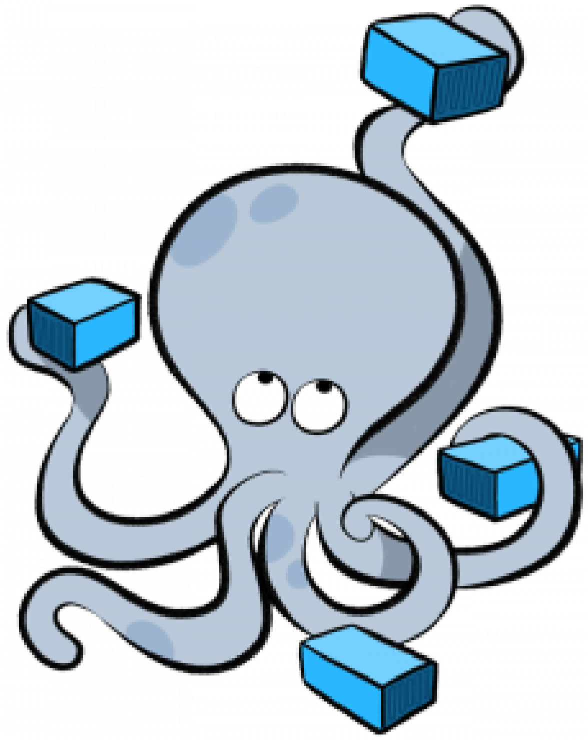
If you want to remove the environment, use the command:

```
docker-machine rm teste-aws
```

After being removed, the instance EC2, provided on AWS, will be automatically removed.

Managing multiple Docker containers with Docker Compose

This article aims to explain in details, and with examples, how does it work the process of managing multiple Docker containers, because as your confidence in using Docker grows in you, your need of using a bigger number of containers increases in the same proportion, and following the good practice of keeping only one service per container commonly results in some extra-request.



Usually, with the increase of the number of running containers, it becomes evident the need of a better communication management, for it is ideal that the services can exchange data between containers when required, that is, you need to deal with the **network** of this new environment.

Think about the work you would have executing dozens of containers manually in the command line, one by one and every required parameter, the network configurations between containers, volumes, etc. Well, you don't have to think about it anymore, for that won't be necessary. To meet this demand of management of multiple containers, the solution is Docker Compose[64].

[64]https://docs.docker.com/compose/overview/

Docker compose is a tool for define and execute multiple Docker containers. It makes possible to configure all required parameters to execute each container from a **definition file**. Inside this file we define each container as a **service**, that is, from now on, every time this text mentions **service**, imagine that that is the definition that is going to be used to initiate a **container**, such as exposed ports, environment variables, etc.

With Docker Compose we can also specify which **volumes** and **network** will be created to be used in the **services** parameters; in other words, that means that you don't have to create them manually so the **services** use additional **network** and **volume** resources.

Docker Compose's **definition file** is the place where all the environment is specified (**network**, **volume** and **services**); it's written according to the YAML[65] format. As a standard, this file is named docker-compose.yml[66].

Anatomy of docker-compose.yml

The YAML standard uses the indentation to separate the code block from the definitions; because of this the use of indentation is very important, that is, if you don't use it correctly, the docker-compose will fail to execute.

Each line of this file can be defined as a value key or a list. Let's see the examples to make the explanation clearer:

[65] https://en.wikipedia.org/wiki/YAML
[66] https://docs.docker.com/compose/compose-file/

```
version: '2'
services:
  web:
    build: .
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        versao: 1
    ports:
      - "5000:5000"
  redis:
    image: redis
```

In the file above we have the first line that define the version of **docker-compose.yml**; in this case we will use the latest version. If you want to compare the difference amongst versions, check out this link[67].

```
version: '2'
```

On the same indentation level we have **services**, that define the beginning of the **services** block that will be define right below.

```
version: '2'
services:
```

On the second indentation level (here it's done with two spaces) we have the name of the first **service** of this file, that gets the name **web**. It opens the **service** definitions block, that is, from the next level of indentation everything that is defined is going to be part of this service.

[67] https://docs.docker.com/compose/compose-file/#versioning

```
version: '2'
services:
  web:
```

On the next level of indentation (done again with two more spaces) we have the first definition of the **web service** that, in this case, is the build[68] that informs that this service will be created not from an existent image, but it will be necessary to build your image before executing it. It also opens a new block of code to parameterize the operation of this image build.

```
version: '2'
services:
  web:
    build: .
```

On the next level of indentation (done again with two more spaces) we have a **build** parameter that, in this case, is the **context**. It is responsible for informing which file context will be used to build the given image; in other words, only files that exist inside this folder will be able to be used in the image building. The context chosen was the **"./dir"**, that is, this indicates that a folder named **dir**, that is in the same file system level of **docker-compose.vml** or of the place where this command will be executed, will be used as a context in the creation of this image. When, soon after the key, a value is provided, this indicates that no block of code will be opened.

```
    build: .
      context: ./dir
```

On the same level of indentation of the **context** definition, that is, still inside the **build** definition block, we have the **dockerfile**, that

[68] https://docs.docker.com/compose/reference/build/

indicates the name of the file that will be used to build the given image. It would be the equivalent to the parameter "-f"[69] of the **docker build** command. If this definition didn't exist, the **docker-compose**, as a standard, would look for a file called **Dockerfile** inside the folder informed in the **context**.

```
    build: .
      context: ./dir
      dockerfile: Dockerfile-alternate
```

On the same level of indentation of the **dockerfile** definition, that is, still inside the **build** definition block, we have the **args**, that defines the arguments that will be used by **Dockerfile**; it's similar to the parameter "–build-args"[70] of the **docker build** command. As its value is not informed in the same line, it's evident that it opens a new block of code.

On the next level of indentation (done again with two more spaces) we have the key "**version**" and the value "**1**", that is, as this definition is part of the code block **args**, this value key is the only argument sent to **Dockerfile**; in other words, the given **Dockerfile** file must be prepared to receive this argument or it will be lost in the image building.

```
    build: .
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        versao: 1
```

Going back two indentation levels (four spaces less in relation to the previous line), we have the **ports** definition, that would be similar

[69] https://docs.docker.com/engine/reference/commandline/build/#specify-dockerfile-f
[70] https://docs.docker.com/engine/reference/commandline/build/#set-build-time-variables-build-arg

to the parameter “-p”[71] of the Docker container run command. It defines which container port will be exposed at the **Docker host**. In our example, is going to be the container port *5000, with the **Docker host** port 5000.

```
  web:
    build: .
    ...
    ports:
      - "5000:5000"
```

Going back one indentation level (two spaces less in relation to the previous line), we leave the block of code of the **web** service; this indicates that no definition informed on this line will be applied to this service, that is, we need to start a block of code of a new service, that in our example will be named **redis**.

```
  redis:
    image: redis
```

On the next indentation level (done again with two more spaces), we have the first definition of the **redis** service, that in this case is the **image**, that is responsible for showing which image will be used to initiate this container. This image will be found in the repository configured on **Docker host**, that is hub.docker.com[72] by default.

Executing Docker Compose

After understanding and creation your own **definition file**, we need to manage it by using the docker-compose binary; the most common usage options are the following:

[71]https://docs.docker.com/engine/reference/commandline/run/#publish-or-expose-port-p-expose

[72]https://hub.docker.com/

- **build** : Used to build all the **services** images that are described with the definition **build** in their block of code.
- **up** : Initiates all the **services** are in the **docker-compose.yml** file.
- **stop** : Stops all the **services** that are in the **docker-compose.yml** file.
- **ps** : Lists all the **services** that were initiated from the **docker-compose.yml** file.

To check other options, see the documentation[73].

[73]https://docs.docker.com/compose/reference/

How to use Docker without GNU/Linux

This article aims to explain, with details and examples, the use of Docker in MacOS and Windows stations.

Docker Toolbox

This text is for people who already know Docker, but still don't know how Docker can be used in a "non Linux" station.

As we said previously, Docker uses specific resources of the host operational system. Today, we have support for Windows and GNU/Linux systems, This means that is not possible to initiate Docker containers in a MacOS station, for instance.

But don't worry if you don't use GNU/Linux, or Windows as an operational system, it is still possible to use this technology without necessarily, executing it in your computer.

It's worth to emphasize that Docker images and containers created on Windows won't work in GNU/Linux because of the dependency of the operations system mentioned earlier.

It's possible to user Docker on MacOS and Windows in two ways:

- Toolbox
- Docker For Mac/Windows

Because it's more complex, therefore demanding a greater context, we will approach in this chapter only the installation and configuration of Docker Toolbox[74]. This solution is, in fact, an abstraction for installation of every environment that requires to use Docker from a MacOS or Windows station.

The installation is simple: both on Windows and on MacOS, just download the corresponding installer in this site[75]; and execute it following the steps in the screen.

The softwares installed on the station - MacOS or Windows - from the Docker Toolbox package are:

- Virtualbox[76]
- Docker machine[77]
- Docker client[78]
- Docker compose[79]

[74]https://www.docker.com/products/docker-toolbox
[75]https://www.docker.com/products/docker-toolbox
[76]https://www.virtualbox.org/
[77]https://docs.docker.com/machine/overview/
[78]https://docs.docker.com/
[79]https://docs.docker.com/compose/overview/

- Kitematic[80]

Docker Machine is the tool that allows to create and maintain Docker environments in virtual machines, cloud environments and even in physical machines. But in this topic we'll only talk about the virtual machine with virtual box.

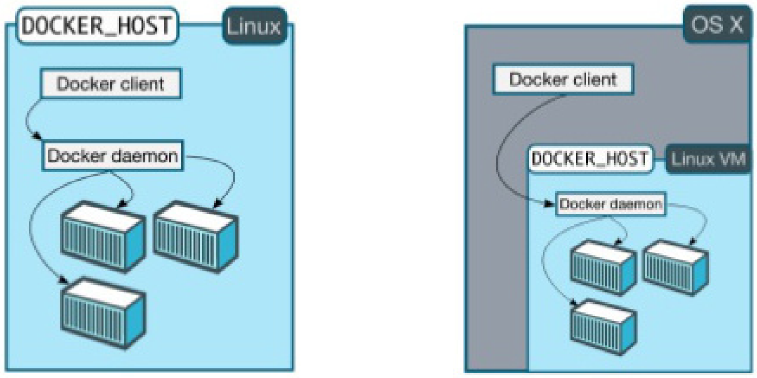
After installing Docker Toolbox, it's simple to create a Docker environment with a virtual machine using Docker Machine.

First, let's verify if there's no virtual machines with Docker installed in their environment:

```
docker-machine ls
```

The command above shows only environments created and maintained by their Docker Machine. It's possible that, after installing Docker Toolbox, you won't find any machine. In this situation, we use the command below in order to create the machine:

```
docker-machine create --driver virtualbox default
```



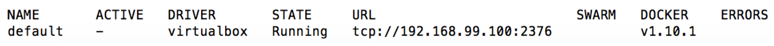
Arquitetura do Docker Toolbox

[80] https://docs.docker.com/kitematic/userguide/

The command creates an environment called "default". In fact, is a virtual machine ("Linux VM" that appears on the image) created on virtual box. With the command is possible to see the machine created:

```
docker-machine ls
```

It should retorne something like this:



```
NAME      ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER    ERRORS
default   -        virtualbox   Running   tcp://192.168.99.100:2376           v1.10.1
```

A virtual machine was created, inside it we have a GNU/Linux operational system with Docker Host installed. This Docker service is listening on the TCP 2376 port from the 192.168.99.100 address. This interface uses a specific network between your computer and the virtual box machines.

To turn off the virtual machine, just execute the command below:

```
docker-machine stop default
```

To start the machine again, just execute the command:

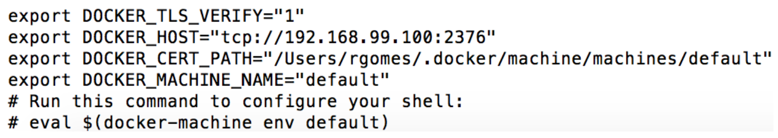
```
docker-machine start default
```

The command "start" is responsible only for starting the machine. It's necessary that the Docker control applications, installed on the station, will be able to connect to the virtual machine created on virtual box with the command "docker-machine create".

The control applications (Docker and Docker-compose) use environment variables to configure which Docker Host will be used. The command below facilitates the work of using all the variables correctly:

```
docker-machine env default
```

The result of this command on MacOS is:



```
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/rgomes/.docker/machine/machines/default"
export DOCKER_MACHINE_NAME="default"
# Run this command to configure your shell:
# eval $(docker-machine env default)
```

As we can see, it shows what can be done to configure all the variables. You can copy the first four lines, that start with "export", and paste on the terminar, or take just the last line without the "#" in the beginning and execute on the command line:

```
eval $(docker-machine env default)
```

Now, the control applications (Docker and Docker-Compose) are ready to use Docker Host form the connection made in the 192.168.99.100 IP service - machine created with the previously mentioned "docker-machine create" command.

To test, we list the running containers in this Docker Host using the command:

```
docker ps
```

While executed in the MacOS or Windows' command line, this Docker client connects itself to the virtual machine that here we call "Linux VM", and requires the list of running containers at the remote Docker Host.

We initiate a container with the command below:

```
docker container run -itd alpine sh
```

Now we verify again the list of running containers:

```
docker ps
```

We can see that the container created from the image “alpine” is running. It’s important to emphasize that this process is executed on Docker Host, in the machine created inside the virtual box that, in this example, holds the IP 192.168.99.100.

To verify the machine’s IP address, just execute the command below:

```
docker-machine ip
```

If the container exposes any port to the Docker Host, whether via “-p” parameter of the “docker container run -p porta_host:porta_-container” command or via “ports” parameter of the docker-compose.yml, it’s good to remember that the IP to access the exposed service is the IP address of the Docker Host; in the example, is “192.168.99.100”.

At this moment, you must be asking yourself: how is it possible to map a folder from the “non Linux” station into a container? Here enters a new Docker artifice to overcome this problem.

Every machine created with the “virtual box” driver automatically creates a mapping of typ “virtual box shared folders” from the user folder to the Docker Host root.

To visualize this mapping, we access the virtual machine we’ve just created in the previous steps:

```
docker-machine ssh default
```

In the GNU/Linux machine console, type the following commands:

```
sudo su
mount | grep vboxsf
```

The vboxsf[81] is a file system used by virtual box to put together shared volumes of the station used to instal the virtual box. In other words, using the shared folder resource makes possible to set up the MacOS folder /Users in the folder /Users from the virtual machine of the Docker Host.

All the content in the MacOS folder /Users/SeuUsuario will be accessible in the folder /Users/SeuUsuario of the GNU/Linux machine that works as a Docker Host in the example. In case you set up the folder /Users/SeuUsuario/MeuCogidog into the container, the data to be set up is the same of the station and nothing needs to be done to replicate this code into the Docher Host.

Let's test it. Create a new file inside the user folder:

```
touch teste
```

We start a container and map the current folder into it:

```
docker container run -itd -v "$PWD:/tmp" --name teste a\
lpine sh
```

In the command above, we initiated a container that will be named "test" and will map the current folder (the variable PWD indicates the current address on MacOS) in the folder /tmp, into the container.

We verify if the file we've just created is inside the container:

```
docker container exec teste ls /tmp/teste
```

[81]https://help.ubuntu.com/community/VirtualBox/SharedFolders

The line above executed the command "ls /tmp/teste" inside the container named "test", created in the previous step.

Now, access Docker Host with the command below, and verify if the test file is in the user folder:

```
docker-machine ssh default
```

Can everything be done automatically? Yes, of course!

Now that you know how to do it manually, if you need to install Docker Toolbox in a new machine and doesn't remember the commands to create the new machine or simply how to set the environment for use, just execute the "Docker Quickstart Terminal" software. It will do the job automatically. In case this machine doesn't exist, it creates one named "default". In case the machine is already created, it automatically configures its environment variables and sets up for using the remote Docker Host from the control applications (Docker and Docker-Compose).

Turning your application into a container

We are continually evolving to deliver ever better applications, in less time, replicable and scalable. However, the efforts and learnings to reach this level of maturity, many times, are not so simple to achieve.

Currently, we observe the rising of several platforms to facilitate the deployment, configuration and scalability of the applications we develop. However, to increase our maturity level we can not just depend on the platform, we need to build our application following the best practices.

Aiming to define a series of best practices common to modern web applications, some developers from Heroku[82] wrote the 12Factor app[83] manifesto, counting on a wide experience in developing web applications.

THE TWELVE-FACTOR APP

"The Twelve-Factor app" (12factor) is a manifesto with a series of best practices for building software using automation declarative formats, maximizing the portability and minimizing divergencies amongst execution environments, allowing the deployment in modern cloud platforms and facilitating scalability. Thus, applications

[82]https://www.heroku.com/

[83]http://12factor.net/pt_br/

are build stateless and connected to any infrastructure services combination to data retention (database, queue, cache memory and similar).

In this chapter we'll talk about creating applications with Docker images based on 12factor app. The idea is to show the best practices to create an infrastructure to support, pack and make your application available with a high level of maturity and agility.

The use of 12factor practices fits Docker very well, because many of its resources are better used if your application is thought within this purpose. Therefore, we will give you an idea of how to take advantage of your application's whole potential.

We will show a HTTP service as an example. It's written in Python, that displays how many times it was accessed, and this information is stored in a conter in a Redis instance.

Now, let's hit these best practices!

Codebase

Aiming to facilitate controlling the code changes, by enabling the traceability of alterations, this best practice indicates that each application must have only one code base and, from it, must be deployed in different environments. It's important to emphasize that this practice is also part of the Continuous Integration (CI[84]) practices. Traditionally, most part of continuous integration systems have, as a starting point, a code base that is built and, later, deployed to development, test and production.

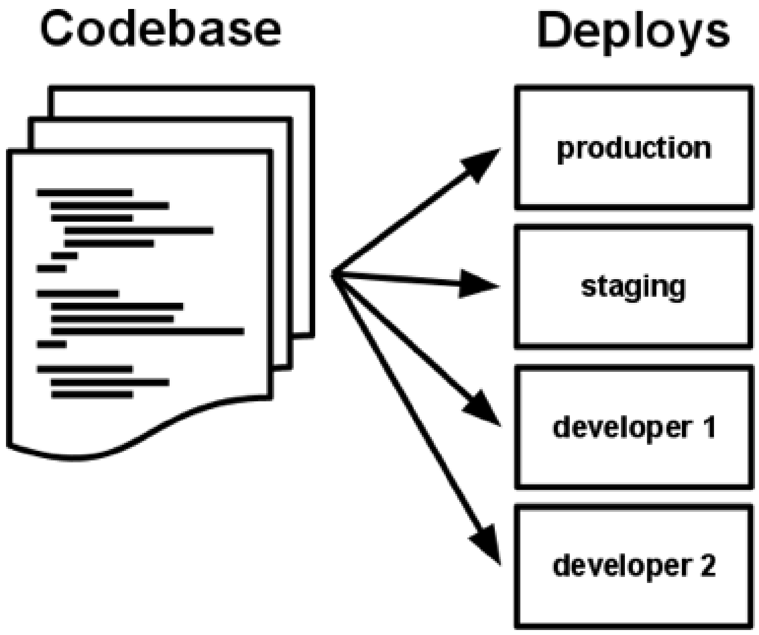
For this explanation, we use the version control system Git and the hosting service Github. We create and provide an example repository[85].

See, every code is inside the repository, arranged by practice in each folder, to facilitate the reproduction. Remember of entering the corresponding folder at each best practice presented.

Docker holds the possibility of using the environment variable to parameterize the infrastructure. Therefore, the same application will behave differently based on the value of environment variables.

Here we use Docker Compose to compose different relevant services for the application in time to execute. Thus, we must define the configuration of these distinct services and the way they communicate.

[84]https://www.thoughtworks.com/continuous-integration
[85]https://github.com/gomex/exemplo-12factor-docker.git



Later, more precisely in the third best practice (called **Config**) of this suggestion compendium we will approach application parameterization in details. For now, we just use options via environment variable in architecture, instead of using it internally in the application code.

To configure the development environment for the example presented, we created the file docker-compose.yml:

```
version: '2'
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
    labels:
     - 'app.environment=${ENV_APP}'
  redis:
    image: redis
    volumes:
     - dados_${ENV_APP}:/data
    labels:
     - 'app.environment=${ENV_APP}'
```

We can notice that the "redis" service is used from the official "redis" image, with no modification. And the web service is generated from the building of a Docker image.

In order to build the Docker image of the web service, we create the following Dockerfile, using the official Python 2.7 image as a base:

```
FROM python:2.7
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
ADD . /code
WORKDIR /code
CMD python app.py
```

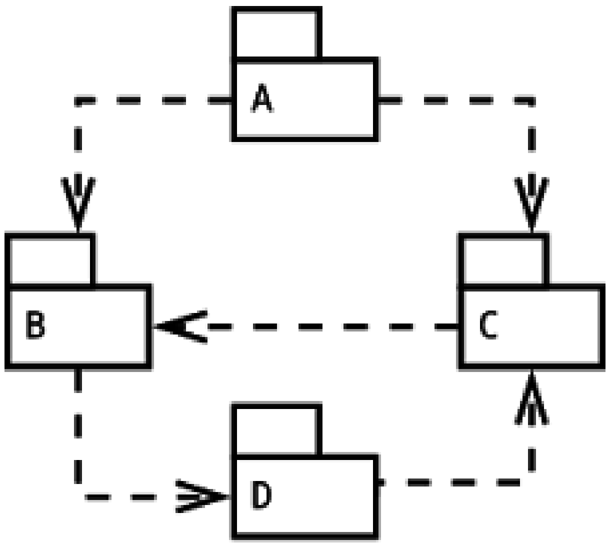
After putting all files in the same folder, we start the environment with the following command:

```
export ENV_APP=devel ; docker-compose -p $ENV_APP up -d
```

As we can notice in the example of this chapter, the environment variable 'ENV_APP' defines which volume is use for persisting the data that is going to be send by the web application. In other words, based on the change of this option, we have the service running with a different behavior, but always from the same code. There you have the concept of the first best practice.

Dependencies

Moving in the list of the 12factor[86] model, right after we approached the code base in this article[87], we have "**Dependency**" as the second best practice.



This best practice suggests the declaration of all required dependencies to execute the code. You must not assume that some component is previously installed in the active responsible for hosting the application.

To make feasible the portability "dream", we need to manage correctly the given application dependencies; this indicates that we

[86]http://12factor.net/
[87]http://techfree.com.br/2016/06/dockerizando-aplicacoes-base-de-codigo/

should also avoid the need of manual work while preparing the infrastructure that supports the application.

Automating the dependency installation process is the big secret of success to attend this best practice. In case the infrastructure is not automated enough to provide initialization without errors, the attendance to this best practice is compromised.

These automated procedures help maintaining the integrity of the process, for the name of dependency packages and their respective versions are specified in the file located in the same repository of the code that, in turn, is traced in a control version system. Thus, we can conclude that nothing is modified without the due record.

Docker fits perfectly in the best practice. It's possible to deliver a minimum infrastructure profile for the application. In turn, it's necessary the explicit declaration of dependencies, so the application runs in the environment.

The example application, written in Python, as we saw a little in the code below, needs two libraries in order to work correctly:

```python
from flask import Flask
from redis import Redis
```

These two dependencies are specified in the file requirements.txt and this file is used a PIP parameter.

"PIP is a package management system used to install and manage software packages written in Python". (Wikipedia)

The PIP command is used with the file requirements.txt in the creation of image, as shown at the Dockerfile of previous best practice (codebase):

```
FROM python:2.7
ADD requirements.txt requirements.txt
RUN pip install -r requirements.txt
ADD . /code
WORKDIR /code
CMD python app.py
```

Notice that one of the steps of Dockerfile is to install the dependencies written in the file requirements.txt with the Python's PIP package manager. Check out the content of requirements.txt file:

```
flask==0.11.1
redis==2.10.5
```

It's important to emphasize the need of specifying the versions of each dependency, because, as in the container model, the images can be built at any time. It's important to know which specific version the application requires. Otherwise, we can find some compatibility problems if one of the dependencies updates and doesn't stay compatible with the complete set of other dependencies and the application that uses it.

To access the code written here, download the repository[88] and go to the "**factor2**" folder.

Another positive outcome of using the best practices is the simplification if another developer uses the code. A new developer can verify in the dependencies file which are prerequisites for the application to run, as well as executing the environment without the need of following the extensive documentation that is rarely updated.

By using Docker it is possible to configure automatically the necessary to run the application code, following perfectly the best practice.

[88] https://github.com/gomex/exemplo-12factor-docker

Config

Moving on on the list of 12factor[89] model, "**Config**" is the third best practice.

When we are creating a software, we apply a given behavior inside the code and usually it's not parameterizable. For the application behave differently, it will be necessary to change part of the code.

The need of modifying the code to change the application's behavior makes unfeasable the execution in the machine (development) in the same way it is used to attend users (production). And, with that, we kill the possibility of portability. And with no portability, what is the advantage of using containers, right?

The goal of the best practice is to make feasible the application configuration without the need of changing the code. Since the application behavior varies according to the environment where is executed, the configurations must consider the environment.

Here are some examples:

- Database configuration that usually are different between environments
- Credentials for accessing remote services (ex.: Digital Ocean or Twitter)
- Which DNS name will be used by the application

As we mentioned, when the configuration is statistically explicit in the code, it's necessary to change manually and do a new binary build at each system reconfiguration.

[89] http://12factor.net/

As showed in the codebase best practice, we use a environments variable to modify the volume we are going to use in redis. In a way, we are already complying with the best practice, but we can go further and change not only the infrastructure behavior, but also something inherent in the code itself.

Here it is the modified application:

```python
from flask import Flask
from redis import Redis
import os
host_run=os.environ.get('HOST_RUN', '0.0.0.0')
debug=os.environ.get('DEBUG', 'True')
app = Flask(__name__)
redis = Redis(host='redis', port=6379)
@app.route('/')
def hello():
   redis.incr('hits')
   return 'Hello World! %s times.' % redis.get('hits')
if __name__ == "__main__":
   app.run(host=host_run, debug=debug)
```

Remember! To access the practice code, just clone this repository[90] and go to the “**factor3**“ folder.

As we can see, we added some parameters to the configuration of the address used to start the web application that will be parameters based on the ”**HOST_RUN**” environment variable value. And the possibility of performing or not the application debug with the ”**DEBUG**” environment variable.

It’s worth to say: in this case the environment variable needs to go into the container, it’s not enough to hold the variable on Docker Host. It is necessary to send it into the container using the parameter

[90] https://github.com/gomex/exemplo-12factor-docker

"-e", in case you use the command "docker container run" or the instruction "environment" on docker-compose.yml:

```
version: "2"
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
    labels:
     - 'app.environment=${ENV_APP}'
    environment:
     - HOST_RUN=${HOST_RUN}
     - DEBUG=${DEBUG}
  redis:
    image: redis:3.2.1
    volumes:
     - dados:/data
    labels:
     - 'app.environment=${ENV_APP}'
volumes:
  dados:
    external: false
```

To execute Docker-Compose, we should do this way:

```
export HOST_RUN="0.0.0.0"; export DEBUG=True ; docker-c\
ompose up -d
```

In the command above, we use the environment variables "**HOST_-RUN**" and "**DEBUG**" from Docker Host to send the environment variables with the same names into the container that, in turn, is consumed by the Python code. In case there aren't any parameters, the container assumes the standard values set in the code.

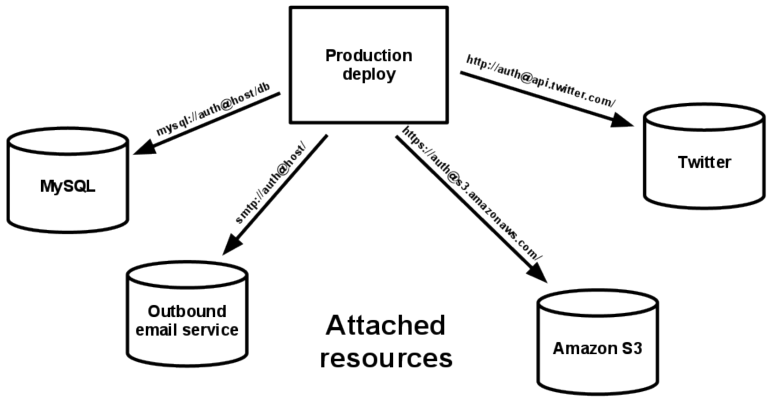
This best practice is followed with the help of Docker, for the code is the same and the configuration is an attachment of the solution that can be parameterized in different ways based on what is configured in the environment variables.

If the application grows, the variables can be carried out in files and paremeterized on docker-compose-yml with the option "env_file".

Backing services

Moving on on the list of 12factor[91] model, we find ”**Backing services**” as the fourth best practice.

To bring some context, “support service” is any application your code consumes in order to function correctly (ex.: database, message service etc.).



Aiming to prevent that the code is overly dependent of a given infrastructure, the best practice says that you, while writing the software, don’t differentiate the internal and external service. That is, the application must be ready to get parameters that configure the service correctly, thus making possible the consume of applications necessary to the solution.

The application in the example was modified to support the best practice:

[91] http://12factor.net/pt_br

```python
from flask import Flask
from redis import Redis
import os
host_run=os.environ.get('HOST_RUN', '0.0.0.0')
debug=os.environ.get('DEBUG', 'True')
host_redis=os.environ.get('HOST_REDIS', 'redis')
port_redis=os.environ.get('PORT_REDIS', '6379')
app = Flask(__name__)
redis = Redis(host=host_redis, port=port_redis)
@app.route('/')
def hello():
   redis.incr('hits')
   return 'Hello World! %s times.' % redis.get('hits')
if __name__ == "__main__":
   app.run(host=host_run, debug=True)
```

As you can see in the code above, the application now gets environment variables to configure the host name and Redis service port. In this case, it's possible to configure the host and Redis port you wish to connect. And this can and must be specified in the docker-compose.yml that has also being through a change to suit this new best practice:

```yaml
version: "2"
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
    labels:
     - 'app.environment=${ENV_APP}'
    environment:
      - HOST_RUN=${HOST_RUN}
      - DEBUG=${DEBUG}
      - PORT_REDIS=6379
      - HOST_REDIS=redis
  redis:
    image: redis:3.2.1
    volumes:
     - dados:/data
    labels:
     - 'app.environment=${ENV_APP}'
volumes:
  dados:
    external: false
```

As we could see in the mentioned codes, the advantage of the best practice goes through the possibility of changing behavior without changing the code. Once more it's possible to enable that the same code build in a certain moment can be reused in a similar way, both in the developer's notebook and in the production server.

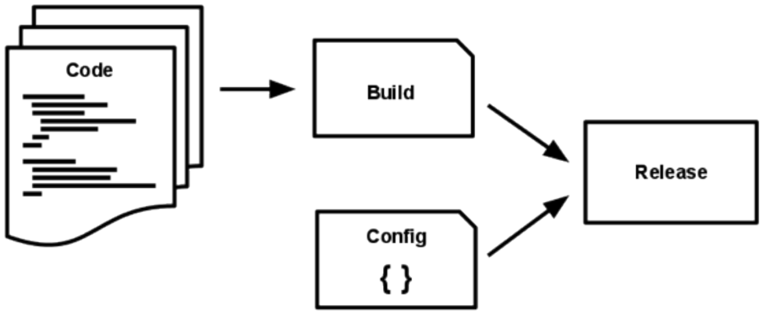
Pay attention to storing secrets inside the docker-compose.yml, because this file is sent to the version control repository and it's important to think about another strategy for keeping secrets.

A possible strategy is to maintain environment variables in Docker Host. Therefore, you need to use variables like **${variable}** inside docker-compose.yml to repass the configuration or use another more advanced secret management resource.

Build, release, run

The next item of the list of 12factor[92] model, "Build, launch, run", is the fifth best practice.

In the process of automating the software deployment infrastructure, we need to be careful so the process behavior is within the expectations and so human errors have low impact in the whole development process, from release to production.



Aiming to organize, divide duties and make the process clearer, 12factor indicates that the base code, to be put into production, needs to go through three phases:

- **Build** - convert the repository code into executable package. In this process we obtain the dependencies, compile the code's binaries and actives.
- **Release** - the package produced in the **build** phase is combined with the configuration. The result is the whole environment, configured and ready to **run**.

[92]http://12factor.net

- **Run** (also known as "runtime") - begins **running** the **release** (application + configuration of that environment), based on the specific configurations of the required environment.

The best practice points out that the application has explicit separations at the
Build, **Release** and **Run** stages. Thus, every change in the application code is build only once in the **Build** stage. Changes in configuration don't need a new **build**, so it's only necessary the **release** and **run** stages.

In such a way, it's possible to create clear controls and processes in each stage. In case something happens in the code **build**, a measure can be taken or even the release can be canceled, so the code in production wouldn't be compromised due to a possible error.

The separation of duties makes possible to know in which stage the problem happened, and fix it manually, if needed.

The artefacts produced must have a single **release** ID. It can be the timestamp (like 2011-04-06-20:32:17) or an incremental number (like v100). With the single artefact, it's possible to guarantee the use of the old version, whether for a rollback or even to compare behaviors after changing the code.

In order to follow the best practice, we need to build the Docker image with the application inside of it. It will be our artifact.

We will have a new script, here called build.sh, with the following content:

```bash
#!/bin/bash

USER="gomex"
TIMESTAMP=$(date "+%Y.%m.%d-%H.%M")

echo "Construindo a imagem ${USER}/app:${TIMESTAMP}"
docker build -t ${USER}/app:${TIMESTAMP} .

echo "Marcando a tag latest também"
docker tag ${USER}/app:${TIMESTAMP} ${USER}/app:latest

echo "Enviando a imagem para nuvem docker"
docker push ${USER}/app:${TIMESTAMP}
docker push ${USER}/app:latest
```

Aside of building the image, it sends it to the Docker's image repository[93].

Remember: this code and others from the best practice are in the repository[94], in the folder "**factor5**".

Sending the image to the repository is an important part of this best practice, for it isolates the process. In case the image is not sent to the repository, it remains only in the server that executed the **build** process; therefore, the next stage needs to be executed in the same server, because such stage needs the image to be available.

In the proposed model, the image in the central repository is available to be downloaded in the server. In case you are using a pipeline tool, it's important to use the product variables (instead of using the date) to uniquely identify the artifact, and guarantee that the image that is going to be used in the Run stage is the same built in the Release stage. Exemple on GoCD: the variables

[93]http://hub.docker.com/
[94]https://github.com/gomex/exemplo-12factor-docker

GO_PIPELINE_NAME and **GO_PIPELINE_COUNTER** can be used together to identify the artifact.

With the image creation, we can guarantee that the **Build** stage was fulfilled perfectly, because now we have an artifact built and ready to be put together with the configuration.

Com a geração da imagem podemos garantir que a etapa **Construir** foi atendida perfeitamente, pois, agora temos um artefato construído e pronto para ser reunido à configuração.

The **Release** stage is the file docker-compose.yml itself, because it gets the due configurations for the environments in which you wish to put the application. Therefore, the file docker-compose.yml changes a little and stops making the image **build**, since now it will be used only for **Release** and **Run** (later):

```
version: "2"
services:
  web:
    image: gomex/app:latest
    ports:
     - "5000:5000"
    volumes:
     - .:/code
    labels:
     - 'app.environment=${ENV_APP}'
    environment:
     - HOST_RUN=${HOST_RUN}
     - DEBUG=${DEBUG}
     - PORT_REDIS=6379
     - HOST_REDIS=redis
  redis:
    image: redis:3.2.1
    volumes:
     - dados:/data
    labels:
     - 'app.environment=${ENV_APP}'
volumes:
  dados:
    external: false
```

In the **docker-compose**.**yml** example above, we used the tag latest to guarantee it always searches for the last **built** image in the process. But as we have already mentioned, in case you are using some continuous delivery tool (such as GoCD, for instance), use the variables to guarantee the image created in the specific pipeline execution.

Therefore, **release** and **run** will use the same artifact: the Docker image, built in the build stage.

The **run** stage, basically, executes the Docker-Compose with the command below:

```
docker-compose up -d
```

Processes

Next in the list of 12factor[95] model, we present "**Processes**" as the sixth best practice.

Nowadays, with the automated processes and the due intelligence in maintaining applications, it is expected that the application can respond to demand peaks with automatic initialization of new processes without affecting its behavior.

The best practice says that 12factor application processes are stateless (don't store state) and share-nothing. Any data that need to persist must be stored in stateful support service, usually used in a database.

The final goal of this practice does not differentiates if the application is executed in the developer's machine or in production, because in this case what changes is the amount of initialized processes to respond the demands. In the developer's machine is only one process; in production this number can be higher.

12factor points out that the memory space or file system of the server can be used briefly as a single transaction cache. For instance,

[95] http://12factor.net

the download of a big file, working over it and storing the results in the database.

We highlight that a state should never be store between requirements, it doesn't matter the processing status of the next requirement.

It's important to emphasize: by following a practice, a application doesn't assume that any item stored in memory cache or in disk will be available for a future requirement or job - with many different processes running, higher are the chances of a future requirement to be served by a different process, even by a different server. Even when running in a single process, a restart (initiated by a code's deployment, changes in configuration, or the running environment reallocating the process to a different physical location) usually will end up with the local state (memory and file system, for instance).

Some applications require persistent sessions to store information of the user session and so. Such sessions are used in future requirements from the same visitor. That is, if it's stored with the processe, it's clearly violating the best practice. In this case, the advice is to use a support service, such as redis, memcached or similar to this type of job that is external to the process. With that, it's possible that the next process, no matter where it is, is able to get the update information.

The application we are working on does not keep local data and everything it need is stored on Redis. We don't need to adequate anything in this code to comply with the best practice, as we can see:

```python
from flask import Flask
from redis import Redis
import os
host_redis=os.environ.get('HOST_REDIS', 'redis')
port_redis=os.environ.get('PORT_REDIS', '6379')
app = Flask(__name__)
 redis = Redis(host=host_redis, port=port_redis)
@app.route('/')
def hello():
 redis.incr('hits')
 return 'Hello World! %s times.' % redis.get('hits')
if __name__ == "__main__":
 app.run(host="0.0.0.0", debug=True)
```

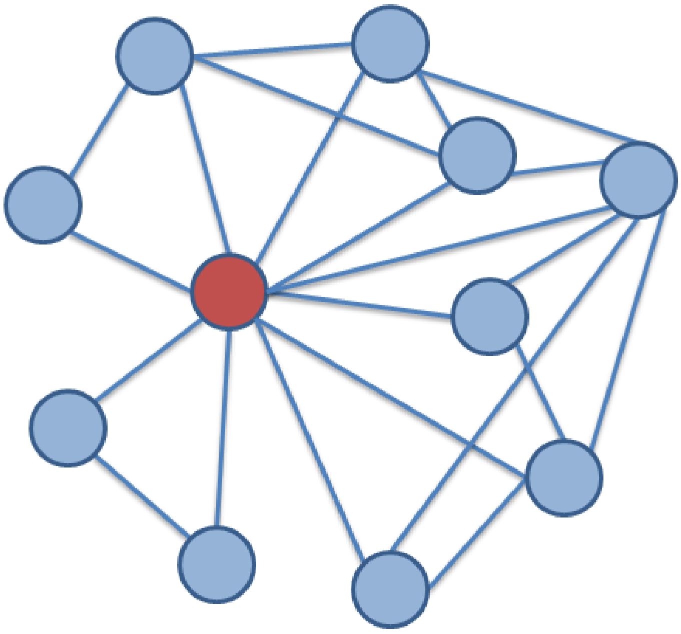
To access the code of the practice, go to the repository[96], in the folder "**factor6**".

[96]https://github.com/gomex/exemplo-12factor-docker

Port binding

According to the list of the 12factor[97] model, the seventh best practice is **port binding**.

It's usual to find applications executed inside containers of web servers, such as Tomcat or Jboss, for instance. Usually, these applications are deployed into the services so they can be access by user externally.



[97]http://12factor.net

The best practice suggests that the given application would be self-contained and depend on a application server, such as Jboss, Tomcat and similar. The software must export a HTTP services and deal with the requirements that come through it. This means that any additional application is unnecessary for the code to be available to the external communication.

Traditionally, the artifact deployments in an application server, such as Tomcat and Jboss, requires the generation of an artifact, that is sent to the given web service. But in the Docker's container model the ideia is that the artifact of the deployment process would be the container itself.

The old artifact deployment process in an application server usually didn't have a fast return, overly increasing the process of deploying a service, because each alteration required to send the artifact to the web application service; the later was responsible for importing, reading and executing the new artifact.

By using Docker, the application become self-contained easily. We built a Dockerfile that describes what the application needs:

```
FROM python:2.7
ADD requirements.txt requirements.txt
RUN pip install -r requirements.txt
ADD . /code
WORKDIR /code
CMD python app.py
EXPOSE 5000
```

The dependencies are described in the file requirements.txt and the data that must be persisted are managed by a service (support services) external to the application.

Another point of the best practice: the application must export the service by binding to a single port. As we seed in the example code, the standard Python port (5000) is initiated, but you can choose another one if you think it's necessary. Here's the part of the code that approaches the subject:

```
if __name__ == "__main__":
  app.run(host="0.0.0.0", debug=True)
```

The port 5000 can be used to serve data locally in a development environment or through a reverse proxy, when migrating to production with the proper domain name to the given application.

Using the binding port model makes the application update process more fluid, once using an intelligent reverse proxy makes possible to add new knots gradually, with the new version, and remove the old ones as the updated versions are executed in parallel.

Important: even that Docker allows using more than one port per container, the best practice emphasizes that you should only use one binding port per application.

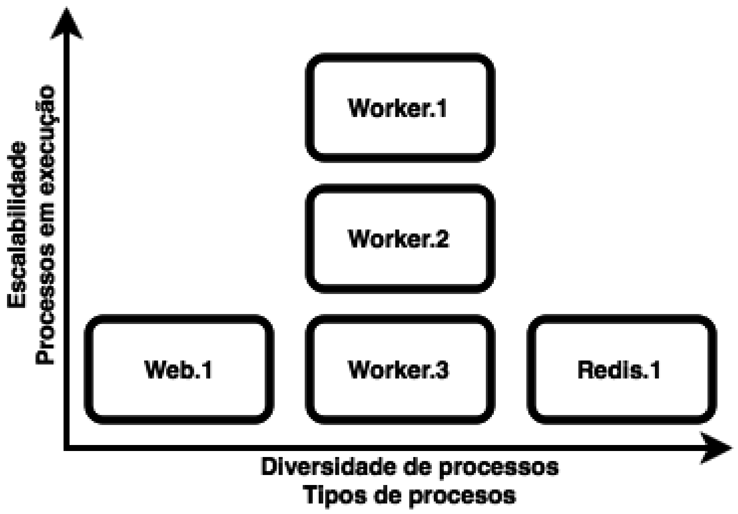
Concurrency

The eighth best practice from the list of 12factor[98] model is "**Concurrency**".

During the process of developing an application it's hard to imagine the amount of requirements it will have when in production. On the other hand, a service that bears great usage volumes is expected in modern solutions. Nothing is more frustrating than requiring access to an application and it is not available. It suggests lack of care and professionalism in most cases.

When the application is put into production, it's usually dimensioned to a given load; however, it's important that the service is ready to escalate. The solution must be able to initiate new processes of the same application if necessary, without affecting the product. The picture below shows the service scalability graphic.

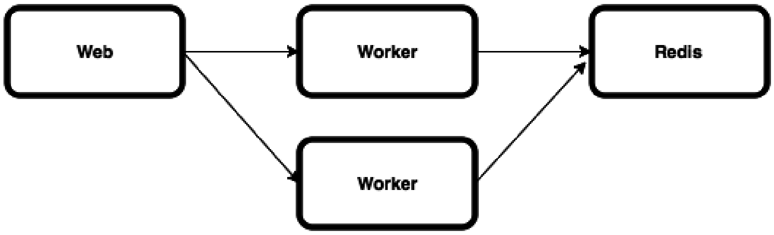
[98]http://12factor.net



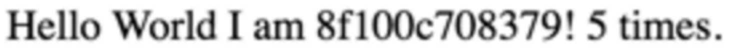
Aiming to avoid any kind of problem in service scalability, the best practice says that the applications must support concurrent executions, and when a process is in execution, instantiating another one in parallel and attend the service, without any loss.

For such, it's important to distribute tasks correctly. It's interesting the process of attaining to the goals, in case it's necessary to execute some activity in backend and later return a page for the browser, it's salutary that two services respond to two activities, separately. Docker makes this task simpler, because in this model it's only necessary to specify a container for each function and properly configure the network between them.

To illustrate this best practice, we'll use the architecture shown in the picture below:



The web service is responsible for receiving the requirements anda balance the workers, which are responsible for processing the requirements, connecting to Redis and return the “Hello World” screen, informing how many times it appeared and which worker name is responding to the requirement (to be sure it’s balancing the load), as we can see in the picture below:



```
Hello World I am 8f100c708379! 5 times.
```

The file **docker-compose.yml** exemplifies the best practice:

```
version: "2"
services:
  web:
    container_name: web
    build: web
    networks:
      - backend
    ports:
      - "80:80"

  worker:
    build: worker
    networks:
      backend:
        aliases:
          - apps
    expose:
      - 80
    depends_on:
      - web

  redis:
    image: redis
    networks:
      - backend

networks:
  backend:
      driver: bridge
```

To do the build of the load balancer, we have the web directory containing Dockerfile files (responsible for creating the image used) and nginx.conf files (configuration file from the load balancer used).

Here's the web Dockerfile content:

```
FROM nginx:1.9

COPY nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

And the content of nginx.conf file:

```
user nginx;
worker_processes 2;

events {
  worker_connections 1024;
}

http {
  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;

  resolver 127.0.0.11 valid=1s;

  server {
    listen 80;
    set $alias "apps";

    location / {
      proxy_pass http://$alias;
    }
  }
}
```

In the configuration file above, some innovations were introduced. The first one, "**resolv 127.0.0.11**", is the Docker's internal DNS service. By using this approach, it's possible to balance the load via name, using Docker's internal resource. For more details on Docker's internal DNS, check out this documentation (https://docs.docker.com/engin dns/[99]) (only in English).

The second innovation, the set **$alias** "**apps**" function, is responsible for specifying the name "**apps**" used to configure the reverse proxy, then "**proxy_pass http://$alias;**". It's important to emphasize that

[99]https://docs.docker.com/engine/userguide/networking/configure-dns/

“**apps**” is the name of the network specified inside the file **docker-compose.yml**. In this case, the balancing is made for the network, and every new container that enters this network is automatically added to the load balancing.

To build the **worker** we have the directory **worker** containing the **Dockerfile** files (responsible for creating the image used), **app.py** (application used in all chapters) and **requirements.txt** (describes the dependencies of app.py).

Below is the content of the file **app.py** that was modified for the practice:

```
from flask import Flask
from redis import Redis
import os
import socket
print(socket.gethostname())
host_redis=os.environ.get('HOST_REDIS', 'redis')
port_redis=os.environ.get('PORT_REDIS', '6379')

app = Flask(__name__)
redis = Redis(host=host_redis, port=port_redis)

@app.route('/')
def hello():
   redis.incr('hits')
   return 'Hello World I am %s! %s times.' % (socket.ge\
thostname(), redis.get('hits'))
if __name__ == "__main__":
   app.run(host="0.0.0.0", debug=True)
```

The content of **requirements.txt**:

```
flask==0.11.1
redis==2.10.5
```

And lastly the **worker Dockerfile** holds the following content:

```
FROM python:2.7
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
COPY . /code
WORKDIR /code
CMD python app.py
```

In the **redis** services there's not building the image, we'll use the official image to exemplify.

To test what was presented so far, clone the repository (https://github.com/gomex/exe 12factor-docker[100]) and access the folder **factor8**, executing the command below in order to initiate the containers:

```
docker-compose up -d
```

Access the containers through the browser at the port 80 from the localhost address. Refresh the page and see that only one name appears.

As a standard, Docker-Compose executes only one instance of each service explicit on **docker-compose.yml**. To increase the amount of **worker** containers from one to two, execute the command below:

```
docker-compose scale worker=2
```

Refresh the page and see that the name of the host alternates between two possibilities, that is, the requirements are being balanced to both containers.

[100] https://github.com/gomex/exemplo-12factor-docker

In this new environment proposal, the **web** service is in charge of receiving the HTTP requirements and balance the load. Then, the **worker** is responsible for processing the requirements - basically getting the host name, access **redis** and count how many time the service was required - and then rollback to return it to the **web** service that, in turn, responds to the user. As we can notice, each environment instance has a defined function, thus making easier to escalate it.

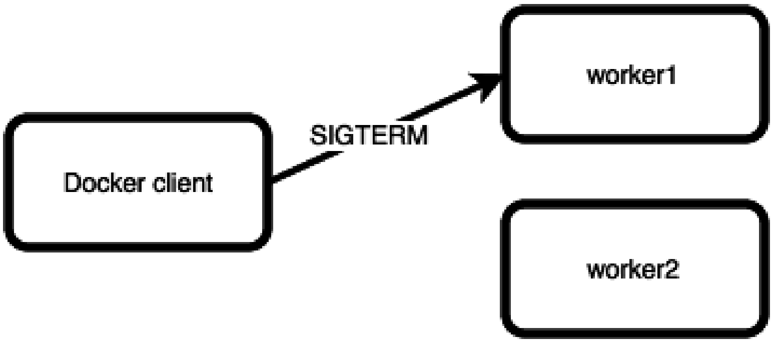
We take the opportunity to give the credits to captain Marcosnils[101], who showed us that is possible to balance the load by the Docker network name.

[101]https://twitter.com/marcosnils

Disposability

In the ninth position of the list of the 12factor[102] model, we have "**Disposability**".

When we talk about web applications, the expectancy is that more than one process attends to the whole traffic required to the service. However, as important as the capacity of starting new processes is the ability of a defective process ending up in the same velocity that started, for a process that takes too long to finish can compromise the whole solution, once it can still be responding to requirements defectively.



Summing up, we can say that we applications should be able to quickly remove defective processes.

Aiming to prevent that the service is dependent on instances that serve it, the best practice says that the applications must be disposable; in other words, shutting down one of its instances must not affect the solution as a whole.

Docker gives the option to automatically dispose a container after using it - on **docker container run** use the option -**rm**. It's impor-

[102]http://12factor.net

tant to highlight that this option doesn't work while in **daemon (-d)** mode, therefore, it only makes sense to use it on **interactive (-i)** mode.

Another important detail of the best practice is to enable the code to shut down "graciously" and restart with no errors. Thus, when hearing a **SIGTERM** the code must finish any requirement in progress and then shut down the process with no problems and quickly, allowing that another process is quickly attended as well.

We consider a "gracious" shut down when an application is capable of self-finishing with no damages to the solution; as it receives the signal to shut down, it immediately refuses new requirements and just finishes up the pendent tasks that are running at that moment. It is implicit in this model: the HTTP requirements are short (no more than a few seconds), and in the case of long connections the client can automatically reconnect if the connection is lost.

The application went through the following alteration to attend the specification:

```python
from flask import Flask
from redis import Redis
from multiprocessing import Process
import signal, os

host_redis=os.environ.get('HOST_REDIS', 'redis')
port_redis=os.environ.get('PORT_REDIS', '6379')

app = Flask(__name__)
redis = Redis(host=host_redis, port=port_redis)

@app.route('/')
def hello():
    redis.incr('hits')
    return 'Hello World! %s times.' % redis.get('hits')
```

```

if __name__ == "__main__":
    def server_handler(signum, frame):
        print 'Signal handler called with signal', sign\
um
        server.terminate()
        server.join()

    signal.signal(signal.SIGTERM, server_handler)

    def run_server():
        app.run(host="0.0.0.0", debug=True)

    server = Process(target=run_server)
    server.start()
```

In the code above, we added up handling for, when it gets a SIGTERM signal, finishing up the process quickly. Without the handling, the code takes longer to be shut down. Therefore, we conclude that the solution is disposable enough. We can shut down and restart the container in another Docker Host and this change won't impact the data integrity.

For understanding purposes on what we work here, it's important to explain: according to Wikipedia a signal is "(...) an asynchronous notification sent to processes aiming to notify the occurrence of an event". And SIGTERM is "(...) the name of a signal known as a computer process in POSIX operative systems. This is the standard signal sent by kill and killall commands. It causes the process to finish, as in SIGKILL, however it can be interpreted or ignored by the process. Thereby, SIGTERM performs a friendlier shut down, allowing to clear memory and closing files".

To perform the test of what was presented so far, clone the repository (https://github.com/gomex/exemplo-12factor-docker[103]) and

[103]https://github.com/gomex/exemplo-12factor-docker

access the folder factor8 (that's right, number 8; let's show the difference relating to factor9), executing the command below to initiate the containers:

```
docker-compose up -d
```

Then, execute the command below to finish up the containers:

```
time docker-compose stop
```

You'll see that the worker finishing up takes about 11 seconds, because of the behavior of Docker-Compose that, in order to finish up first performs a **SIGTERM** and wait 10 seconds so the application shuts down by itself; otherwise, it sends a **SIGKILL** that shuts down the process abruptly. This time out is configurable. In case you wish to change it, just use the parameter "**-t**" or "**–timeout**". Check an example:

```
docker-compose stop -t 5
```

Obs.: The value informed after the parameter is measured in seconds.

Now, to test the modified code, go to the folder **factor9** and execute the following command:

```
docker-compose up -d
```

Later, request the conclusion:

```
time docker-compose stop
```

Notice that the worker process finished up faster, for it got the SIGTERM signal. The application shut down by itself and didn't need to receive a SIGKILL signal to be effectively shut down.

Development/production parity

Next on the 12factor[104] model list, we have "**Development/production parity**" as the tenth best practice.

Desenvolvimento
Mudar estado
Produção

Unfortunately, in most software work environments there is a deep abyss between development and production. It's not just fortuity or lack of luck, it's because of the differences between the development and infrastructure teams. According to 12factor, these differences appear within the following:

- **Time**: developers can work in a code for days, weeks or even months to go into production.
- **Personnel**: developers write code, operation engineers deploy the code.
- **Tools**: devedores can use sets such as Nginx, SQLite and OS X, while the app in production uses Apache, MySQL and Linux.

12factor intends to collaborate in order to reduce this abyss between teams and equalize environments. Regarding the differences presente, here are the respective proposals:

- **Time**: developers can write code and see the deployment finished hours or even minutes later.

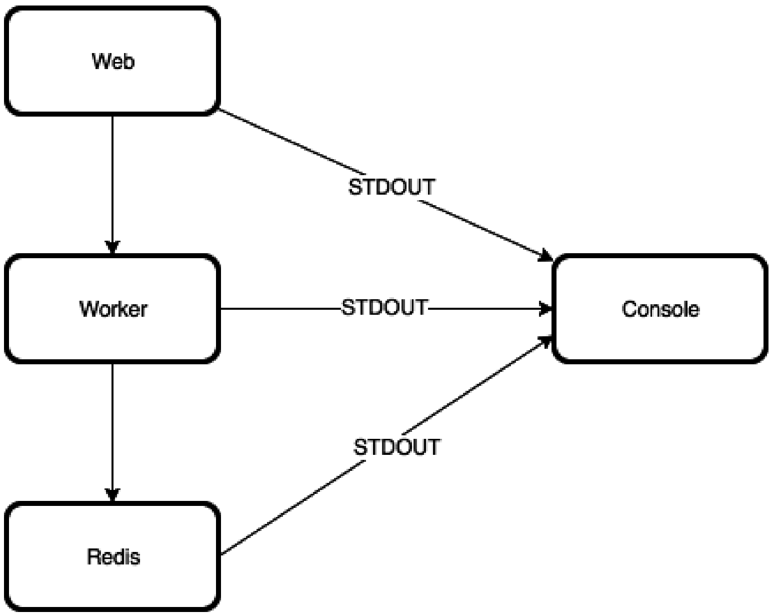
[104] http://12factor.net

- **Personnel**: developers who write code are closely involved in deployment and following its behavior in production.
- **Tools**: keep development and production as similar as possible.

One of container's making goals is to collaborate with portability between development and production environments. The ideia is that the image is built and only its status will be modified to be put into production. The current code is ready to this behavior, thus there's not much to be modified to guarantee the best practice. It's like a bonus for adopting Docker and following the other 12factor best practices.

Logs

The eleventh best practice in the 12factor[105] model list is "**Logs**".



While developing codes, generating data for logs is something very much consolidated. We don't believe that there's softwares in development without this concern. However, the correct use of log goes beyond of just generating data.

For context effect, according to 12factor, log is: "(...) the stream of aggregated, time-ordered events collected from the output streams of all running processes and backing services".

Usually, logs are stores in files, with events per line (backtraces from exception may span multiple lines). But this practice is not

[105] http://12factor.net

recommended, at least not from an application's perspective. This means that the application should not worry in which file it will store the logs.

To specify files implies on informing the correct directory of this file, that, in turn, results in previous environment configuration. This impacts negatively in application's portability, because is necessary that the environment that will get the solution follows a series of technical requirements to back the application, burying the possibility of "Build it once, run it anywhere".

The best practice says that the applications should not manage or route log files, but should be deposited without any buffer to the standard output (STDOUT). Thus, an infrastructure external to the application - platform - must manage, collect and format the logs output to future reading. This is really important when the application is running in several instances.

With Docker, such task becomes easy for Docker already collects standard output logs and send them to some of the several log drivers. The driver can be configured in the container initialization in order to group the logs in the log remote service, such as syslog.

The example code in the repository(https://github.com/gomex/exemplo-12factor-docker[106]), in the folder factor11, is ready to test the best practice, for it sends all data outputs to STDOUT and you can check it by initiating the service with the command below:

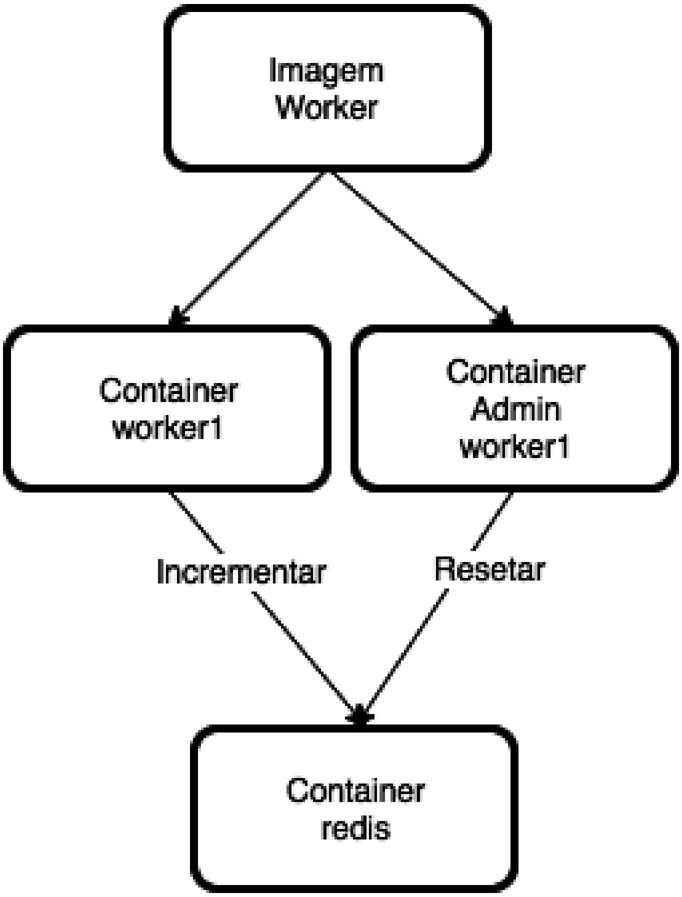
```
docker-compose up
```

After initiating, access the browser and verify the application requirements that appear on the Docker-Compose console.

[106]https://github.com/gomex/exemplo-12factor-docker

Admin processes

The twelfth and last best practice from the 12factor[107] model list: “**Admin processes**”.



[107] http://12factor.net

Every application requires administration. That means that, once deployed, it's possible that the application need to receive some commands to correct possible issues or simply change behavior. As examples we have database migrations, running several scripts as backup and also running a console to inspect the service.

The best practice recommends admin processes executed in environments similar to the ones used in the running code, following all of the practices presented so far.

Using Docker makes possible to run the processes using the same base image in the running environment you wish. Thereby, with can benefit from the communication between containers and the use of volumes required and similar.

To exemplify the best practice we create the file **reset.py**:

```python
from redis import Redis
import signal, os

host_redis=os.environ.get('HOST_REDIS', 'redis')
port_redis=os.environ.get('PORT_REDIS', '6379')

redis = Redis(host=host_redis, port=port_redis)

redis.set('hits', 0)
```

The command is given by using a different container from the same Docker image, and it's responsible for reinitiating the Redis' visit counter. First, we start the environment, download the repository, and access the folder factor12 and execute the command:

```
docker-compose up
```

Access the application in the browser. In case you are using GNU/Linux or Docker For Mac and Windows, access the address 127.0.0.1. You'll see the following sentence:

```
“Hello World! 1 times.”
```

Access the application a couple more times so the counter goes up.

Then, execute the admin command from the worker service:

```
docker-compose exec worker python reset.py
```

The command “**python reset.py**” will be executed inside a new container, but using the same image of a regular worker.

Access the application again and check if the counter started from 1 again.

Tips for using Docker

If you read the first part of the book, you already know the basics on Docker; but now that you intend to start using it more frequently some issues may rise, because as in any tool, Docker has its own set of best practices and tips.

The goal of this article is to present some tips for better use Docker. That doesn't mean that your way of using is necessarily wrong.

Every tool requires some best practices to make its use more effective and less likely to show future problems.

This chapter is divided in two sections: tips for running ('docker container run') e best practices in image building ('docker build'/'Dockerfile').

Tips for running

Remember: each 'docker container run' command creates a new container based on a specific image and starts a process inside it, after a command ('CMD' specified on Dockerfile).

Disposable containers

It's expected that executed containers could be disposed without any problems. Therefore, it's important to use truly ephemera containers.

For such, use the arguments '–rm', that makes all containers, and their data, to bem removed after finishing the execution, preventing from taking unnecessary space in disk.

In general, the command 'run' can be used as in the example:

```
docker container run --rm -it debian /bin/bash
```

Notice that '-it' means '–interactive —tty'. It's used to fix the command line to the container, thus, after this 'docker container run', every command are executed by the 'bash' inside the container. To exit, use 'exit' or press 'Control-d'. These parameters are very useful to execute a container in the foreground.

Check environment variables

Some times, it's necessary to check which meta-data are defined as environment variables in an image. Use the command 'env' to get this information:

```
docker container run --rm -it debian env
```

To check the old environment variables of a container:

```
docker inspect --format '{{.Config.Env}}' <container>
```

For other meta-data, use variations of command 'docker inspect'.

Logs

Docker captures standard output logs ('STDOUT') and errors output ('STDERR').

These records can be routed to different systems ('syslog', 'fluentd', ...) that can be specified in the driver configuration[108] '–log-driver=VALUE' in the command 'docker container run'.

When using the standard driver 'json-file' (also, 'journald'), you can use the following command to recover the logs:

[108] https://docs.docker.com/engine/admin/logging/overview/

```
docker logs -f <container_name>
```

Notice yet the argument '-f' to follow up the next log messages interactively. If you want to stop, press 'Ctrl-c'.

Backup

Docker container data are exposed and shared via volume arguments used while creating and starting the container. These volumes don't follow the rules from Union File System[109], because the data persist even when the container is removed.

To create a volume in a given container, execute as it follows:

```
docker container run --rm -v /usr/share/nginx/html --na\
me nginx_teste nginx
```

By executing this container, we'll have the Nginx service that uses the volume created to persist its data; the data will persist even after the container is removed.

It a system admin best practice to do periodic backups; to execute this activity (extract data), use the command:

```
docker container run --rm -v /tmp:/backup --volumes-fro\
m nginx-teste busybox tar -cvf /backup/backup_nginx.tar\
 /usr/share/nginx/html
```

After executing the command, we have a 'backup_nginx.tar' file inside the folder /tmp of **Docker host**.

In order to restores this backup, use:

[109]https://docs.docker.com/engine/reference/glossary/#union-file-system

```
docker container run --rm -v /tmp:/backup --volumes-fro\
m nginx-teste busybox tar -xvf /backup/backup.tar /usr/\
share/nginx/html
```

More information can be found in the answer[110], where is possible to find some *aliases* for these two commands. These *aliases* are also available below, in the *Aliases* section.

Some other sources:

- Docker official documentation about Data backup, restoration or migration[111]
- A backup tool (currently deprecated): docker-infra/docker-backup[112]

Use docker container exec to "enter a container"

Eventually, it's necessary to enter a running container so you can check any problem, running tests or simply debug.

Never install daemon SSH in a Docker container. Use 'docker container exec' to enter a container and run commands:

```
docker container exec -it <nome do container em execuçã\
o> bash
```

The feature is useful in local development and experiments. But avoid using container in production or automating tools around it.

Check the documentation[113].

[110] http://stackoverflow.com/a/34776997/1046584
[111] https://docs.docker.com/engine/userguide/containers/dockervolumes/#backup-restore-or-migrate-data-volumes
[112] https://github.com/docker-infra/docker-backup
[113] https://docs.docker.com/engine/reference/commandline/exec/

No space in Docker Host disk

By executing containers and building images several times, the space in disk can become scarce. When it happens, it’s necessary to clean some containers, images and logs.

A fast way of cleaning containers and images is by using the following command:

```
docker system prune
```

With this command you will remove:

- All the containers not in use at the moment
- All the volumes not in use by at least one container
- All the networks not in use by at least one container
- Every *dangling* images

Obs.: Let’s not get too deep in Docker’s low level concept and say that *dangling* images are simply images with no tags, therefore unnecessary for conventional use.

Depending on the type of application, logs can occupy some volume too. The management depends a lot on which driver[114] is used. In the standard driver (‘json-file’), the cleaning can be done by executing the following command inside **Docker Host**:

```
echo "" > $(docker inspect --format='{{.LogPath}}' <con\
tainer_name_or_id>)
```

- The functionality proposal of cleaning the logs history was, actually, rejected. More information on: https://github.com/docker/compose/is

[114] https://docs.docker.com/engine/admin/logging/overview/
[115] https://github.com/docker/compose/issues/1083

- Consider specifying the ‘max-size’ option to the log *driver* while executing ‘docker container run’: https://docs.docker.com/engine/referen file-options[116]

Aliases

Alias makes possible to transform big commands into smaller ones. We have some new options to execute more complex tasks.

Use these *aliases* in your ‘.zshrc’ or ‘.bashrc’ to clean imagens and containers, to backup and restore etc.

```
# runs docker container exec in the latest container
function docker-exec-last {
  docker container exec -ti $( docker ps -a -q -l) /bin\
/bash
}

function docker-get-ip {
  # Usage: docker-get-ip (name or sha)
  [ -n "$1" ] && docker inspect --format "{{ .NetworkSe\
ttings.IPAddress }}" $1
}

function docker-get-id {
  # Usage: docker-get-id (friendly-name)
  [ -n "$1" ] && docker inspect --format "{{ .ID }}" "$\
1"
}

function docker-get-image {
  # Usage: docker-get-image (friendly-name)
  [ -n "$1" ] && docker inspect --format "{{ .Image }}"\
```

[116] https://docs.docker.com/engine/reference/logging/overview/#json-file-options

```
 "$1"
}

function docker-get-state {
  # Usage: docker-get-state (friendly-name)
  [ -n "$1" ] && docker inspect --format "{{ .State.Run\
ning }}" "$1"
}

function docker-memory {
  for line in `docker ps | awk '{print $1}' | grep -v C\
ONTAINER`; do docker ps | grep $line | awk '{printf $NF\
" "}' && echo $(( `cat /sys/fs/cgroup/memory/docker/$li\
ne*/memory.usage_in_bytes` / 1024 / 1024 ))MB ; done
}
# keeps the commmand history when running a container
function basher() {
    if [[ $1 = 'run' ]]
    then
        shift
        docker container run -e HIST_FILE=/root/.bash_h\
istory -v $HOME/.bash_history:/root/.bash_history "$@"
    else
        docker "$@"
    fi
}
# backup files from a docker volume into /tmp/backup.ta\
r.gz
function docker-volume-backup-compressed() {
  docker container run --rm -v /tmp:/backup --volumes-f\
rom "$1" debian:jessie tar -czvf /backup/backup.tar.gz \
"${@:2}"
}
# restore files from /tmp/backup.tar.gz into a docker v\
olume
```

```
function docker-volume-restore-compressed() {
  docker container run --rm -v /tmp:/backup --volumes-f\
rom "$1" debian:jessie tar -xzvf /backup/backup.tar.gz \
"${@:2}"
  echo "Double checking files..."
  docker container run --rm -v /tmp:/backup --volumes-f\
rom "$1" debian:jessie ls -lh "${@:2}"
}
# backup files from a docker volume into /tmp/backup.tar
function docker-volume-backup() {
  docker container run --rm -v /tmp:/backup --volumes-f\
rom "$1" busybox tar -cvf /backup/backup.tar "${@:2}"
}
# restore files from /tmp/backup.tar into a docker volu\
me
function docker-volume-restore() {
  docker container run --rm -v /tmp:/backup --volumes-f\
rom "$1" busybox tar -xvf /backup/backup.tar "${@:2}"
  echo "Double checking files..."
  docker container run --rm -v /tmp:/backup --volumes-f\
rom "$1" busybox ls -lh "${@:2}"
}
```

Sources:

- https://zwischenzugs.wordpress.com/2015/06/14/my-favourite-docker-tip/[117]
- https://website-humblec.rhcloud.com/docker-tips-and-tricks/[118]

[117] https://zwischenzugs.wordpress.com/2015/06/14/my-favourite-docker-tip/
[118] https://website-humblec.rhcloud.com/docker-tips-and-tricks/

Best practices to build images

On Docker, images are traditionally built using a 'Dockerfile'. There are some good guides on the best practices to build Docker images. Take a look at our recommendations:

- Documentação oficial[119]
- Guia do projeto Atomic[120]
- Melhores práticas do Michael Crosby Parte 1[121]
- Melhores práticas do Michael Crosby Parte 2[122]

Use a "linter"

"Linter" is a tool that provides tips and warning on some source code. There are some simple options for 'Dockerfile', but it is, still, a new evolving space.

Many options were discussed here[123].

Since January 2016, the most complete "linter" seems to be hadolint[124], available in two versions: on-line and terminal. The interest thing abou this tool is that it uses the mature Shell Check[125] to validate the shell commands.

The basics

The container produced by the 'Dockerfile' image must be as ephemeral as possible. This means that it should be possible to

[119] https://docs.docker.com/engine/articles/dockerfile_best-practices/
[120] http://www.projectatomic.io/docs/docker-image-author-guidance/
[121] http://crosbymichael.com/dockerfile-best-practices.html
[122] http://crosbymichael.com/dockerfile-best-practices.html
[123] https://stackoverflow.com/questions/28182047/is-there-a-way-to-lint-the-dockerfile
[124] http://hadolint.lukasmartinelli.ch/
[125] http://www.shellcheck.net/about.html

stop it, destroy it and replace it for a new container built with the minimum effort.

It's usual to put other files, such as documentation, in the same directory of 'Dockerfile'; to improve the building performance, delete files and directories creating a dockerignore[126] file in the same directory. This file works similarly to '.gitignore'. Using it helps to minimize the building context `docker build`.

Avoid adding packages and unnecessary extra dependencies to the application and minimize complexity, image size, building time and attack surface.

Also minimize the layer amount: whenever possible, group up various commands. However, take in consideration the volatility and maintenance of these layers.

In most cases, run only one process per container. Decoupling applications in several container eases up horizontal scalability, reuse and monitoring of containers.

Choose COPY over ADD

The 'ADD' command exists since the beginning of Docker. It's versatile, and provides some tricks aside of simply copying files from the building context, and that's what makes it magical and hard to understand. It allows to download url files and automatically extract files of known formats (tar, gzip, bzip2, etc.).

On the other hand 'COPY' is a simpler command to put files and folders of the building path inside the Docker image. Thus, choose 'COPY' unless you are absolutely sure that 'ADD' is necessary. For more details, check here[127].

[126]https://docs.docker.com/engine/reference/builder/
[127]https://labs.ctl.io/dockerfile-add-vs-copy/

Run a “checksum” after downloading and before using the file

Instead of using ‘ADD’ to download and add files to the image, prefer using curl[128] and verify through a ‘checksum’ after the download. This guarantees that the file is the expected one and will not vary over time. If the file that the URL indicates changes, the ‘checksum’ will change and the image building will fail. This is important, because it favor reproducibility and safety in image building.

A good inspiration is Jenkins’ official Dockerfile[129]:

```
ENV JENKINS_VERSION 1.625.3
ENV JENKINS_SHA 537d910f541c25a23499b222ccd37ca25e074a0c

RUN curl -fL http://mirrors.jenkins-ci.org/war-stable/$\
JENKINS_VERSION/jenkins.war -o /usr/share/jenkins/jenki\
ns.war \
  && echo "$JENKINS_SHA /usr/share/jenkins/jenkins.war"\
 | sha1sum -c -
```

Use a image with smallest base

Whenever possible, use official images as a base for your image. You can use the ‘debian’[130] image, for instance, that is very well controlled and kept small (around 150mb). Remember of also using specific *tags*, such as ‘debian:jessie’.

If more tools and dependencies are required, look for images like ‘buildpack-deps’[131].

[128] https://curl.haxx.se/
[129] https://github.com/jenkinsci/docker/blob/83ce6f6070f1670563a00d0f61d04edd62b78f4f/Dockerfile#L36
[130] https://hub.docker.com/_/debian/
[131] https://hub.docker.com/_/buildpack-deps/

However, in case ‘debian’ is still too big, there are minimalist images such as ‘alpine’[132] or even ‘busybox’[133]. Avoid ‘alpine’ if DNS is required, for there are a few issues to be solved[134]. In addition, avoid it for languages that use GCC, such as Ruby, Node, Python, etc.; because ‘alpine’ uses libc MUSL that can produce different binaries.

Avoid gigantic images such as ‘phusion/baseimage’[135]. This image is too big, it defeats the philosophy of process per container and much of what makes it up is not essential for Docker containers; read more here[136] .

Other sources[137]

Use the layer building cache

Another useful feature provided by ‘Dockerfile’ is the fast rebuild using the layer cache. In order to take advantage from this resource, add tools and dependencies that change less frequently in the top of ‘Dockerfile’.

For instance, consider to install the code dependencies before adding the code. In the case of NodeJS:

```
COPY package.json /app/
RUN npm install
COPY . /app
```

To read more on this subject, check out this link[138].

132 https://hub.docker.com/r/gliderlabs/alpine/
133 https://hub.docker.com/r/gliderlabs/alpine/
134 https://github.com/gliderlabs/docker-alpine/blob/master/docs/caveats.md
135 https://hub.docker.com/r/phusion/baseimage/
136 https://blog.docker.com/2014/06/why-you-dont-need-to-run-sshd-in-docker/
137 http://www.iron.io/microcontainers-tiny-portable-containers/
138 http://bitjudo.com/blog/2014/03/13/building-efficient-dockerfiles-node-dot-js/

Clean on the same layer

While using a package manager to install any software, the best practice is to clean the cache generated by the package manager soon after installing the dependencies. For instance, using 'apt-get':

```
RUN apt-get update && \
    apt-get install -y curl python-pip && \
    pip install requests && \
    apt-get remove -y python-pip curl && \
    rm -rf /var/lib/apt/lists/*
```

In general, the apt cache (generated by 'apt-get update') must be cleaned up by removing '/var/lib/apt/lists'. This helps to keep the image size small. In addition, note that 'pip' and 'curl' also are removed once they're unnecessary to the production application. Remember that the cleaning must be made in the same layer (command 'RUN'). Otherwise, data will be persisted on this layer and removing them later will not have the same effect in the final image size.

Note that, according to the documentation[139], the official Debian and Ubuntu images run 'apt-get clean' automatically. Ergo, the explicit invocation is not necessary.

Avoid to run 'apt-get upgrade' or 'dist-upgrade', badales several packages of the base image are not going to update inside a container with no privileges. If there's a specific package to update, just use 'apt-get install -y foo' to automatically update it.

To read more on this subject, check out the link[140] and this one[141].

[139] https://github.com/docker/docker/blob/03e2923e42446dbb830c654d0eec323a0b4ef02a/contrib/mkimage/debootstrap#L82-L105

[140] http://blog.replicated.com/2016/02/05/refactoring-a-dockerfile-for-image-size/

[141] https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#apt-get

Use a “wrapper” script as ENTRYPOINT, sometimes

A *wrapper script* can help to configure the environment and define the application configuration. It can event define the standard configurations when they’re not available.

A great example is provided in Kelsey Hightower: 12 Fracturated Apps’ article[142]:

```
#!/bin/sh
set -e
datadir=${APP_DATADIR:="/var/lib/data"}
host=${APP_HOST:="127.0.0.1"}
port=${APP_PORT:="3306"}
username=${APP_USERNAME:=""}
password=${APP_PASSWORD:=""}
database=${APP_DATABASE:=""}
cat <<EOF > /etc/config.json
{
  "datadir": "${datadir}",
  "host": "${host}",
  "port": "${port}",
  "username": "${username}",
  "password": "${password}",
  "database": "${database}"
}
EOF
mkdir -p ${APP_DATADIR}
exec "/app"
```

Note: **always** use ‘exec’ in shell scripts regarding the application. Therefore, the application can get Unix signals.

[142]https://medium.com/@kelseyhightower/12-fractured-apps-1080c73d481c#.xn2cylwnk

Also, consider using a simple initialization system (e.g. dumb init[143]), such as 'CMD' base, so the Unix signal can be duly treated. Read more here[144].

Log for stdout

Applications inside Docker should emit logs for 'stdout'. However, some applications write logs in files. In these cases, the solution is to create a file *symlink* for 'stdout'.

Example: Dockerfile of nginx[145]:

```
# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log
RUN ln -sf /dev/stderr /var/log/nginx/error.log
```

To read more, check out this link[146].

Be careful while adding data to a volume on Dockerfile

Remember of using the instruction 'VOLUME' to expose data from database, configuration or files and folders created by the container. Use for any mutable data and parts served to the user of the service to which the image was created.

Avoid adding a lot of data in a folder and then turn it into a 'VOLUME' only when starting the container, because you can slow down the loading. By creating the container, the data will be copied from the image to the set volume. As said before, use 'VOLUME' when creating the imagem.

[143]https://github.com/Yelp/dumb-init
[144]http://engineeringblog.yelp.com/2016/01/dumb-init-an-init-for-docker.html
[145]https://github.com/nginxinc/docker-nginx/blob/master/Dockerfile
[146]https://serverfault.com/questions/599103/make-a-docker-application-write-to-stdout

Besides, while still creating the image (‘build’), don’t add data to paths previously declared as ‘VOLUME’. This doesn’t work, the data won’t be persisted, for datas in volumes are not *committed* into images.

Read more at Jérôme Petazzoni’s explanation[147].

Ports EXPOSE

Docker favors reproducibility and portability. Images should be capable of run in any server, how many times necessary. Therefore, never expose public ports. However, expose the application’ standard ports privately.

```
# public and private mapping, avoid
EXPOSE 80:8080

# only private
EXPOSE 80
```

[147] https://jpetazzo.github.io/2015/01/19/dockerfile-and-data-in-volumes/

Apêndice

Container or virtual machine?

After Docker's sudden success - virtualization based on containers - many people are asking about a possible migration from the virtual machine model to containers.

⚠

We answer this calmly: **Both!**

Both are virtualization methods, but work in different "layers". It's worth to detail each solution to make clear that they are not necessarily competition.

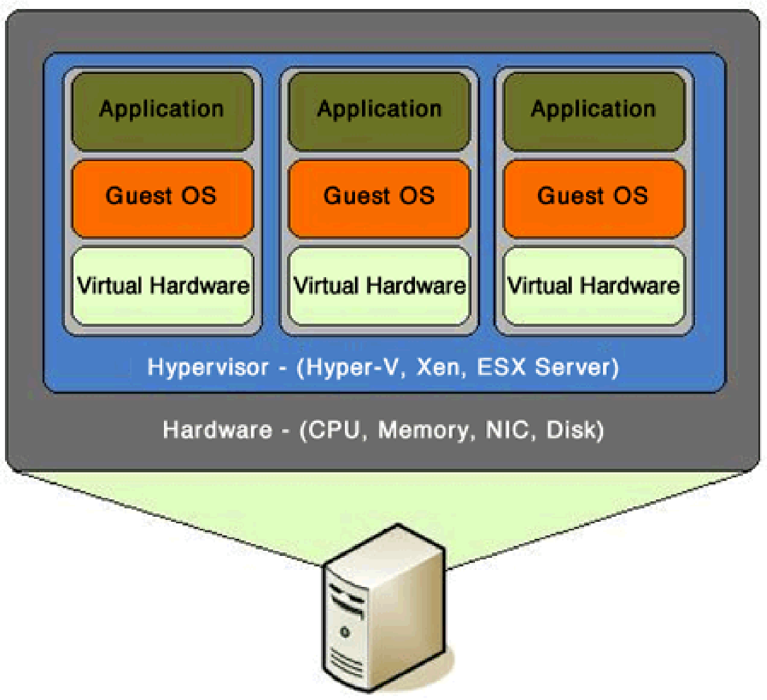
Virtual machine

An old concept, coming from the Mainframes on mid 1960s. Each operator had the vision of accessing a dedicated machine but, in fact, every Mainframe resource was shared to all operators.

The goal of the model is to share physical resources amongst several isolated environments, and each one of them is guarding a whole machine: with memory, disk, processor, network and other peripherals, all delivered via virtualization abstraction.

It's like if inside the physical machine, smaller and independent machines were created. Each one has its own operational system that, in turn, interacts with the virtual hardwares that were delivered by the virtualization model in the machine level.

It's important to say: the operational system installed in the virtual machine will interact with the virtual hardware, not the real ones.



While this model evolved, the softwares that implement the solution could offer more features, such as better interface to manage virtual environments and high availability using several physical hosts.

With the new features for managing environments on virtual machines, it's possible to specify the amount of physical resource each virtual environment uses and even to gradually increase it if necessary.

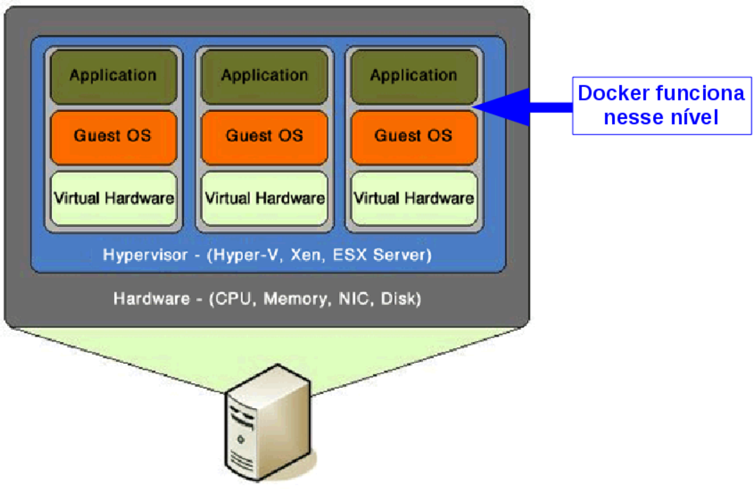
Currently, virtual machines are a reality for any organization that requires TI environments, for it facilitates the management of physical machines and sharing amongst several environments necessary for the basic infrastructure.

Container

This virtualization model is on the operational system level; that is, different from a virtual machine, a container does not view the whole machine, it's just a running process in a kernel shared amongst all other containers.

It uses the namespace to provide the due isolation of RAM memory, processing, disk and network access. Even when shared in the same kernel, this running process views the use of a dedicated operational system.

It's a relatively old virtualization model. In mid 1982, chroot was already doing something we can consider virtualization on operational system level, and in 2008, LXC was doing something relatively similiar to the current Docker. Yet, in the beginning, Docker was using LXC but today it has its own interface in order to access namespace, cgroup and others.



Being an innovative solution, Docker brings up several services and new facilities that make the model much more attractive.

The LXC environment configuration was not a simple task; it was necessary some technical knowledge to create and maintain an environment with it. As Docker arrived, this process became much simpler. Just install the binary, download the images and execute them.

Other Docker novelty was the creation of the "images" concept. Roughly, we can describe images as the static definitions of what containers must be in initialization. They are like ambience photographs. Once they're instantiated and put into execution, they assume the role of containers; they leave the definition abstraction and turn into running processes, within an isolated context. They see an operational system dedicated to themselves, but in fact they share the same kernel.

Along with the easiness of using containers, Docker put together the concept of cloud, that offers the service of loading and "downloading" Docker images. It's a web application that provides repository of ready environments, making available an enormous level of environment sharing.

By using Docker's cloud service, we can realize that adopting the container model surpasses the technical issue and introduces subjects such as environment management and update; now it's possible to easily share changes and provide a centralized management of environment definitions.

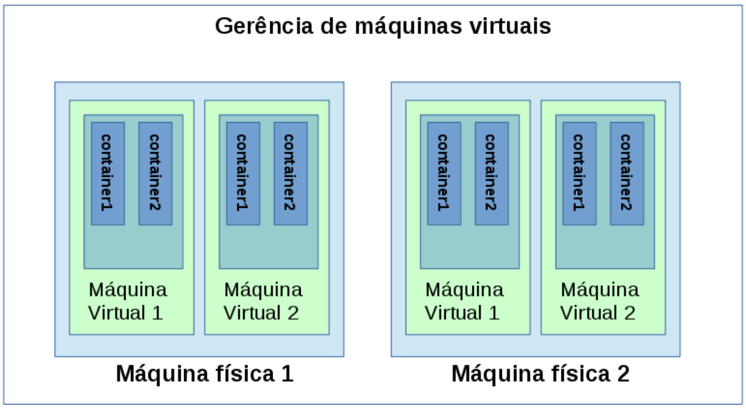
Using Docker cloud now makes possible to provide lighter testing environments, allowing you to, in a meeting with your boss, for instance, download the solution of a problem he/she explains and show it to him/her before he/she leaves the room. It allows also that you make available a pattern of best practices to a given service and share it with everybody in your company, where you can get feedbacks and perform modifications through time.

Conclusion

With the data presented we realized that the conflict point between solutions is low. They can and usually will be adopted together. You can provide a physical machine with a virtual machine server, in which host virtual machines will be created that, in turn, will have Docker installed. In this Docker, they will make available environments and their respective services, each one in a container.

See that we'll have several isolation levels. In the first one, the physical machine, that was separated in various virtual machines, that is, we already have our layer of operational systems interacting with distinct virtual hardware, such as virtual network card, disks, processor and memory. In this environment, we would only have installed the basic operational system and Docker.

In the second isolation level, we have Docker downloading ready images and providing running containers that, in turn, create new isolated environments, at the level of processing, memory, disk and network. In this case, we can have in the same virtual machine a web application environments and a database. But in different containers (and that wouldn't be a best practice issue for service management), a lot less security.



If these containers are replicated between virtual machines, it would be possible to provide high availability with no huge costs, that is, using an external balancer and making available the cluster of persisted data via database.

With all this facility using a few commands, resources and knowledge, we just need a little bit of time to change the paradigm of managing actives and patience to face the new issues within the model.

Useful commands

Here are some useful, simple commands:

- Remove all inactive containers
  `docker container prune`
- Stop all containers
  `docker stop $(docker ps -q)`
- Remove all local images
  `docker image prune`
- Remove "orphan" volumes
  `docker volume prune`
- Shows use of containers resources running
  `docker stats $(docker ps --format {{.Names}})`

Can I run GUI applications?

Absolutely, it's perfectly possible to run gUI (or X11) applications in containers; that means that all advantages of using Docker apply also to graphic applications.

In addition, it's possible to make the application run in multiple systems (Linux, Windows and macOS) just by building it for Linux.

How?

First of all… in 99% of the cases it's necessary to grant access to X: 'xhost local' (this access will be available until you shut down/restart the host).

This is the simplest command, it mount the X11 socket of the host in the container and defines the display (note that we are "evolving" the commands little by little, but you can use only the flags that you find necessary - the only ones that are mandatory are the mounting of the volume '/tmp/.X11-unix' and the environment variable 'DISPLAY'):

```
docker container run [--rm [-it]|-d] \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-e DISPLAY \
imagem [comando]
```

In some cases, the 'DISPLAY/ variable has to be'DISPLAY=unix$DISPLAY' (but, to be honest, I don't know why, only know that that was the recommended by the person who built the image).

To user the hardware 3D acceleration support:

```
docker container run [--rm [-it]|-d] \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-e DISPLAY \
--device /dev/dri \
imagem [comando]
```

Adding audio:

```
docker container run [--rm [-it]|-d] \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-e DISPLAY \
--device /dev/dri \
--device /dev/snd \
imagem [comando]
```

Adding webcam:

```
docker container run [--rm [-it]|-d] \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-e DISPLAY \
--device /dev/dri \
--device /dev/snd \
--device /dev/video0 \
imagem [comando]
```

Using the same date/hour from the host:

```
docker container run [--rm [-it]|-d] \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-e DISPLAY \
--device /dev/dri \
--device /dev/snd \
--device /dev/video0 \
-v /etc/localtime:/etc/localtime:ro \
imagem [comando]
```

Attention: depending on the distribution, there's not a /etc/localtime; you have to check how it defines the timezone and "replicate" it into the container.

Keeping the application's configurations:

```
docker container run [--rm [-it]|-d] \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-e DISPLAY \
--device /dev/dri \
--device /dev/snd \
--device /dev/video0 \
-v /etc/localtime:/etc/localtime:ro \
-v $HOME/.config/app:/root/.config/app \
imagem [comando]
```

Obs.: the path is just an example.

Bonus: Videogame joystick (actually, any input device):

```
docker container run [--rm [-it]|-d] \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-e DISPLAY \
--device /dev/dri \
--device /dev/snd \
--device /dev/video0 \
-v /etc/localtime:/etc/localtime:ro \
-v $HOME/.config/app:/root/.config/app \
--device /dev/input \
imagem [comando]
```

What about docker-compose?

It works normally… Just mount the X11 socket and define the environment variable on docker-compose.yml and will be possible to start multiple application with only one command.

On Windows and macOS

Mac OS X

Install Docker for Mac

```
brew install socat
brew cask install xquartz
open -a XQuartz

socat TCP-LISTEN:6000,reuseaddr,fork UNIX-CLIENT:\"$DIS\
PLAY\"
docker container run -e DISPLAY=hostip:0 [...] image OU\
 DISPLAY=hostip:0 docker-compose up [-d]
```

Windows[148]
Install xming
Install o Docker for Windows

```
xming :0 -ac -clipboard -multiwindow
docker container run -e DISPLAY=hostip:0 [...] image OU\
 DISPLAY=hostip:0 docker-compose up [-d]
```

Obs.: In case you're using Docker Toolbox, insert VM's IP ('docker-machine ip default' will inform).

[148] https://github.com/docker/docker/issues/8710#issuecomment-135109677