



Docker-1 (macOS flavored)

All aboard the xhy.ve train!

Franck LOURME coton@42.fr
42 Staff pedago@42.fr

Summary: Row, row, row your boat... gently down the stream...

Contents

I	Foreword	2
II	Introduction	3
III	Goals	4
III.1	Before we get started	4
III.2	Your Hitchhiker's guide to the containers	5
IV	General instructions	6
V	Mandatory part	7
V.1	How to Docker	7
V.2	Exercises	8
VI	Dockerfiles	11
VI.1	Exercise 00: vim/emacs	12
VI.2	Exercise 01: BYOTSS	12
VI.3	Exercise 02: Dockerfile in a Dockerfile... in a Dockerfile ?	13
VI.4	Exercise 03: ... and bacon strips ... and bacon strips	13
VII	Bonus part	14
VIII	Turn-in and peer-evaluation	15

Chapter I

Foreword

This is what Wikipedia says about Baleen :

Baleen is a filter-feeder system inside the mouths of baleen whales. The baleen system works by whale opening its mouth underwater and taking in water. The whale then pushes the water out, and animals such as krill are filtered by the baleen and remain as food source for the whale. Baleen is similar to bristles and is made of keratin, the same substance found in human fingernails and hair. Baleen is a skin derivative. Some whales, such as the bowhead whale, have longer baleen than others. Other whales, such as the gray whale, only use one side of their baleen. These baleen bristles are arranged in plates across the upper jaw of the whale. Baleen is often called whalebone, but that name also can refer to the normal bones of whales, which have often been used as a material, especially as a cheaper substitute for ivory in carving.

The word baleen derives from the Latin balaena, related to the Greek phalaina – both of which mean "whale".

People formerly used baleen (usually referred to as "whalebone") for making numerous items where flexibility and strength were required, including backscratchers, collar stiffeners, buggy whips, parasol ribs, crinoline petticoats and corset stays. It was commonly used to crease paper; its flexibility kept it from damaging the paper. It was also occasionally used in cable-backed bows. Synthetic materials are now usually used for similar purposes, especially plastic and fibre glass. It is not to be confused with whale's bone meaning the bones of whales, used in carving, for cutlery handles and other uses for the bones of various large species.

Chapter II

Introduction

You might have experienced one of these unpleasant situations:

- You're developing on a computer and you don't have the admin rights to install a dependency.
- It took you five hours to install the latest version of you favorite DB but your boss tells you the app will have to run on a previous version.
- It took you two weeks to build a perfectly clean application... that only works on your machine.
- The SysOp in charge of all launches hates you because your app is undeployable.

Long story short, you already have or soon will spend, at some point in your life, as much time setting up your work environment as actually developing your app.

Right now, you are probably developing a one-block app, with softwares and libraries installed directly in your development environment, or maybe in a virtual environment... Imagine if your application has to be deployed all over the world and you have to re-develop it for all existing platforms and OSs...

Docker was created to satisfy this need for unification and normalisation: it makes it possible to split an application into several microservices, light, adaptable, universal and scalable, and it also gives the system administrators a great flexibility to deploy and scale up the app.

This suite of projects on Docker will help you better understand this specific tool, but also the various aspects of applications development using microservices.

Chapter III

Goals

III.1 Before we get started

The aim of the Docker-1 project is to make you handle `docker` and `docker-machine`, the bases to understand the idea of containerization of services. You can see this project as an initiation. But before you go on this adventure, it's important to warn you about a few things:

- You will do the exercises on `docker 1.12 at least`. If you're not sure what version of `docker` you have, you can run `docker version`. Brew always provides the latest version of `docker`... just saying.
- It is EXTREMELY important to have the `docker-machine` binary available on your dump before you start exploring the wonderful world of containers, and for the bonuses. Read the `docker` documentation available online to install it if necessary.
- Take the time to read the documentation thoroughly. You will have to sooner or later anyway.

III.2 Your Hitchhicker's guide to the containers

As you work through this project, we STRONGLY advise you to have the official docker documentation at hand in order to handle properly your future containers. You should also take a look at the documentation for the official containers we will ask you to implement throughout this project.

You will very soon realize that you don't need to reinvent the wheel all over again every time you want to develop an application or implement a specific service.

In this perspective, here are some very useful links for your project:

- [The excellent Docker official documentation](#)
- [Docker's public Registry](#)
- [Docker's official Github with many upcoming projects](#)
- [Jessie Frazelle's blog, former main contributor on Docker](#)
- [Her Github with many good ideas](#)

Chapter IV

General instructions

- The project will be corrected by humans.
- For the evaluation, the containers will be hosted on the correction dump and locally.
- You cannot correct your project with a container hosted on a remote machine, please read the rule above if you don't understand this one.
- There will be 2 folders on your repository (+1 if you did the bonuses):
 - 00_how_to_docker
 - 01_dockerfiles
 - 02_bonus [BONUS]

There should be a specific tree structure in each file, defined in the corresponding chapters.



For storage reasons, we advise you to move your ".docker" and "VirtualBox VMs" folders from your home to your goinfre and to use symbolic links.

Chapter V

Mandatory part

V.1 How to Docker

In this first part you will be introduced to Docker and its main options. In your repository, create a `00_how_to_docker` folder in which you will push the solutions for all exercises.

If you don't have it already, install `docker`, `docker-machine` and `virtualbox`. Docker and `docker-machine` can be installed via Brew, and `virtualbox` via the Managed Software Center.

Check that `docker version` prints out your current docker version, and that you have at least the 1.12 docker (Editor's Note: as I write, a new versioning notation is being used, and the most recent version is 17.03.0-ce-mac2).

You might be tempted to use `Docker for Mac`, and I sympathize. Don't worry, you'll use it for the next Docker projects. But for this project, you will need to implement clustering locally, and that can be done quite easily with `docker-machine`!

You will push your work in the `00_how_to_docker` folder: one file per exercise, named after the exercise number, with the command(s) requested for this exercise.

```
$> ls 00_how_to_docker
01      02      03      04      05      06
[...]
31      32      33      34
$> cat 00_how_to_docker/01
docker-machine [...]
```


V.2 Exercises

For each exercise, we will ask you to give the shell command(s) to:

1. **Create** a virtual machine with **docker-machine** using the **virtualbox** driver, and named **Char**.
2. Get the **IP** address of the **Char** virtual machine.
3. Define the variables needed by your virtual machine **Char** in the general **env** of your terminal, so that you can run the **docker ps** command without errors. You have to fix all four environment variables with one command, and you are not allowed to use your shell's builtin to set these variables by hand.
4. Get the **hello-world** container from the Docker Hub, where it's available.
5. **Launch** the **hello-world** container, and make sure that it prints its welcome message, then leaves it.
6. Launch an **nginx** container, available on Docker Hub, as a background task. It should be named **overlord**, be able to restart on its own, and have its 80 port attached to the 5000 port of **Char**. You can check that your container functions properly by visiting `http://<ip-de-char>:5000` on your web browser.
7. Get the internal IP address of the **overlord** container without starting its shell and in one command.
8. Launch a shell from an **alpine** container, and make sure that you can interact directly with the container via your terminal, and that the container deletes itself once the shell's execution is done.
9. From the shell of a **debian** container, install via the container's package manager everything you need to compile C source code and push it onto a git repo (of course, make sure before that the package manager and the packages already in the container are updated). For this exercise, you should only specify the commands to be run directly in the container.
10. Create a volume named **hatchery**.
11. List all the Docker volumes created on the machine. Remember. **VOLUMES**.

12. Launch a **mysql** container as a background task. It should be able to restart on its own in case of error, and the root password of the database should be **Kerrigan**. You will also make sure that the database is stored in the **hatchery** volume, that the container directly creates a database named **zerglings**, and that the container itself is named **spawning-pool**.
13. Print the environment variables of the **spawning-pool** container in one command, to be sure that you have configured your container properly.
14. Launch a **wordpress** container as a background task, just for fun. The container should be named **lair**, its 80 port should be bound to the 8080 port of the virtual machine, and it should be able to use the **spawning-pool** container as a database service. You can try to access **lair** on your machine via a web browser, with the IP address of the virtual machine as a URL.
Congratulations, you just deployed a functional Wordpress website in two commands!
15. Launch a **phpmyadmin** container as a background task. It should be named **roach-warden**, its 80 port should be bound to the 8081 port of the virtual machine and it should be able to explore the database stored in the **spawning-pool** container.
16. Look up the **spawning-pool** container's logs in real time without running its shell.
17. Display all the currently active containers on the **Char** virtual machine.
18. Relaunch the **overlord** container.
19. Launch a container name **Abathur**. It will be a **Python** container, 2-slim version, its **/root** folder will be bound to a **HOME** folder on your host, and its 3000 port will be bound to the 3000 port of your virtual machine.
You will personalize this container so that you can use the **Flask** micro-framework in its latest version. You will make sure that an html page displaying "Hello World" with **<h1>** tags can be served by **Flask**. You will test that your container is properly set up by accessing, via curl or a web browser, the IP address of your virtual machine on the 3000 port.
You will also list all the necessary commands in your repository.
20. Create a local swarm, the **Char** virtual machine should be its manager.
21. Create another virtual machine with **docker-machine** using the **virtualbox** driver, and name it **Aiur**.
22. Add **Aiur** as a worker of the local swarm in which **Char** is leader (the command to take control of **Aiur** is not requested).
23. Create an overlay-type internal network that you will name **overmind**.

24. Launch a **rabbitmq** SERVICE that will be named **orbital-command**. You should define a specific user and password for the RabbitMQ service, they can be whatever you want. This service will be on the **overmind** network.
25. List all the services of the local swarm.
26. Launch a **42school/engineering-bay** service in two replicas and make sure that the service works properly (see the documentation provided at hub.docker.com). This service will be named **engineering-bay** and will be on the **overmind** network.
27. Get the real-time logs of one the tasks of the **engineering-bay** service.
28. ... Damn it, a group of zergs is attacking **orbital-command**, and shutting down the **engineering-bay** service won't help at all... You must send a troupe of Marines to eliminate the intruders. Launch a **42school/marine-squad** in two replicas, and make sure that the service works properly (see the documentation provided at hub.docker.com). This service will be named... **marines** and will be on the **overmind** network.
29. Display all the tasks of the **marines** service.
30. Increase the number of copies of the **marines** service up to twenty, because there's never enough Marines to eliminate Zergs. (Remember to take a look at the tasks and logs of the service, you'll see, it's fun.)
31. Force quit and delete all the services on the local swarm, in one command.
32. Force quit and delete all the containers (whatever their status), in one command.
33. Delete all the container images stored on the **Char** virtual machine, in one command as well.
34. Delete the **Aiur** virtual machine without using `rm -rf`.

Chapter VI

Dockerfiles

So, that was fun wasn't it?

Now, it's time get down to business. Docker makes it possible to create yourself OWN containers for your OWN applications! That's what this chapter is about. Luckily, Docker lets you program Dockerfiles (aka a Makefile for Docker... I think you get the nuance). Dockerfiles use a specific syntax that reuses a base image or an existing container to add your own dependencies and your own files.


You will also notice that every command built by your Dockerfiles generates a layer that's reusable in other Dockerfiles or other versions of the same Dockerfile, in order to avoid data duplication. Awesome, isn't it?

But before you start making your own containers, make sure that your virtual machine is cleared of all images and residual active containers. We'll start from the beginning and build up more and more complex applications.

Create a `01_dockerfiles` folder in which you'll store all the Dockerfiles that you'll need later, in separate folders (ex00 / ex01 ...).


Check out this link to design high quality Dockerfiles: [Dockerfile Best Practices](#)

VI.1 Exercise 00: vim/emacs

	Exercise 00
Exercise 00: vim/emacs	
Turn-in directory : <i>ex00/</i>	
Files to turn in : Dockerfile	
Allowed functions : -	
Notes : n/a	


From an **alpine** image you'll add to your container your favorite text editor, **vim** or **emacs**, that will launch along with your container.

VI.2 Exercise 01: BYOTSS

	Exercise 01
Exercise 01: BYOTSS	
Turn-in directory : <i>ex01/</i>	
Files to turn in : Dockerfile + Scripts (if applicable)	
Allowed functions : -	
Notes : n/a	

From a **debian** image you will add the appropriate sources to create a TeamSpeak server, that will launch along with your container. It will be deemed valid if at least one user can connect to it and engage in a normal discussion (no far-fetched setup), so be sure to create your Dockerfile with the right options. Your program should get the sources when it builds, they cannot be in your repository.

VI.3 Exercise 02: Dockerfile in a Dockerfile... in a Dockerfile ?


	Exercise 02
Exercise 02: Dockerfile in a Dockerfile... in a Dockerfile ?	
Turn-in directory : <i>ex02/</i>	
Files to turn in : Dockerfile	
Allowed functions : -	
Notes : n/a	

You are going to create your first Dockerfile to containerize Rails applications. That's a special configuration: this particular Dockerfile will be generic, and called in another Dockerfile, that will look something like this:

```
FROM          ft-rails:5.0.2-on-build
CMD           ["rails", "s", "-b", "0.0.0.0"]
```

Your generic container should install, via a **ruby** container, all the necessary dependencies and gems, then **copy your rails application** in the `/opt/app` folder of your container. Docker has to install the appropriate gems when it builds, but also launch the migrations and the db population for your application. The child Dockerfile should launch the rails server (see example below). If you don't know what commands to use, it's high time to look at the [Ruby on Rails](#) documentation.

VI.4 Exercise 03: ... and bacon strips ... and bacon strips ...

	Exercise 03
Exercise 03: ... and bacon strips ... and bacon strips ...	
Turn-in directory : <i>ex03/</i>	
Files to turn in : Dockerfile	
Allowed functions : -	
Notes : n/a	

Docker can be useful to test an application that's still being developed without polluting your libraries. You will have to design a Dockerfile that gets the development version of [Gogs](#), installs it with all the dependencies and the necessary configurations, and launches the application, all as it builds. The container will be deemed valid if you can access the web client, and if you can interact via GIT with this container.

Chapter VII

Bonus part

Now that you understand all the subtleties of Docker, you can enjoy yourself for the bonuses!

You can create your future work environments, such as:

- A developing environment to code in nodejs, but using `yarn` rather than `npm`
- PAMP (!) with docker-compose, but using `MariaDB` rather than `MySQL`
- Recreate a full MEAN Stack
- Specific Dockerfiles for your C, Ruby, Go, Ocaml, Rust... projects
- etc.

Keep in mind that you have seen nothing yet of the full power of Docker. You could for instance:

- Try to containerize a VPN configuration for your containers
- Try to cluster several machines on different cloud services with Docker Swarm
- Start the construction of a base image, make your own debian or archlinux container.
- Containerize the server of your favorite game, like Minecraft (?)
- etc.

Whatever you decide to do, we expect you to store everything that's relevant in the `02_bonus` folder of your repository.

Have Fun!!

Chapter VIII

Turn-in and peer-evaluation

Turn in your work using your `Git` repository, as usual. Only the work that's in your repository will be graded during the evaluation.