



## Docker-1

Now, you're thinking with containers...

Franck LOURME [coton@42.fr](mailto:coton@42.fr)  
42 Staff [pedago@42.fr](mailto:pedago@42.fr)

*Résumé: Row, row, row your boat... gently down the stream...*

# Table des matières

<b>I</b>	<b>Préambule</b>	<b>2</b>
<b>II</b>	<b>Introduction</b>	<b>3</b>
<b>III</b>	<b>Prologue</b>	<b>4</b>
III.1	Before we get started . . . . .	4
III.2	Your Hitchhiker guide to the containers . . . . .	5
<b>IV</b>	<b>Consignes</b>	<b>6</b>
<b>V</b>	<b>Partie Obligatoire</b>	<b>7</b>
V.1	How to Docker . . . . .	7
V.2	Exercices . . . . .	8
<b>VI</b>	<b>Dockerfiles</b>	<b>11</b>
VI.1	Exercice 00 : vim/emacs . . . . .	12
VI.2	Exercice 01 : BYOTSS . . . . .	12
VI.3	Exercice 02 : Dockerfile in a Dockerfile... in a Dockerfile? . . . . .	13
VI.4	Exercice 03 : What does the Fox say? . . . . .	13
<b>VII</b>	<b>Partie Bonus</b>	<b>14</b>
<b>VIII</b>	<b>Rendu et peer-évaluation</b>	<b>15</b>

# Chapitre I

## Préambule

Voici ce que dit Wikipedia sur les baleines :

Sont appelées baleines les tiges, généralement métalliques, qui sont glissées dans les corsets, leur donnant forme et maintien et permettant de modeler le corps du porteur.

Des origines du corset au XVI<sup>e</sup> siècle jusqu'au XVIII<sup>e</sup> siècle, les corsetiers utilisaient dans leurs corsets de vrais fanons de baleines, étroits, solides et souples, en très grand nombre.

Au début du XIX<sup>e</sup> siècle, on inventa les baleines en acier, qui furent les seules utilisées à partir de 1850 (bien qu'on ait gardé l'ancien nom de « baleines »).

Aujourd'hui il existe trois types de baleines :

- les baleines en acier inoxydable, utilisées par les corsetiers et vendues au mètre (cerclette) ou bien déjà coupées, recouvertes de plastique blanc pour éviter encore moins de risques de rouille. Elles sont plus ou moins flexibles, selon l'épaisseur et la largeur de la baleine.
- les baleines dites « spirales » ou « ressort », pouvant servir occasionnellement, constituées également en acier mais encore plus souples, offrant moins de maintien mais plus de flexibilité.
- les baleines en plastique vendues en rouleaux, au mètre, utilisées dans les bustiers de prêt-à-porter (lingerie, robes de mariées...). Elles sont également utilisées en costume et présentent des caractéristiques proches des fanons de baleine utilisés à l'origine : avec la chaleur du corps, elles se déforment légèrement et épousent les formes. Elles sont plus ou moins souple en fonction de leur épaisseur et leur légèreté les fait parfois préférer aux baleines métal.

Les baleines désignent aussi les tiges articulées qui permettent de déployer et tendre la toile lors de l'ouverture du parapluie.

# Chapitre II

## Introduction

Vous avez peut être déjà expérimenté ce léger désagrément quand :

- Vous développez sur une machine et il vous manque les droits root pour installer une dépendance
- Vous avez passé 5h à installer la dernière version de votre BDD préféré mais que votre chef vous dit que l'appli devra être opérationnelle sur une version précédente
- Vous avez passé deux semaines à faire une application propre... mais qui ne marche que sur votre machine
- Le SysOp en charge du déploiement vous haïs car votre application est indéploable

Bref, vous passez ou passerez à un moment de votre vie, au moins autant de temps sur l'aménagement de votre environnement de développement que sur le développement de votre application.

De même, en ce moment, vous développez sûrement vos application en un seul bloc, avec des logiciels et des librairies installées en dur dans votre environnement de développement, ou peut etre dans un environnement virtuel... Imaginez que votre application soit déployée à travers le monde et que vous deviez la redévelopper pour n'importe quelle plateforme et OS...

C'est un peu dans cette volonté d'unification et de normalisation que Docker a été créé, en laissant aux développeurs la possibilité de séparer leur application en microservices, adaptifs, légers, universels et scalables, mais aussi aux administrateurs système une grande flexibilité de déploiement et de scalabilité.

Plusieurs projets utilisant Docker vous aideront à mieux comprendre l'outil et à mieux cerner les différentes facettes du développement d'application en microservices.

# Chapitre III

## Prologue

### III.1 Before we get started

Docker-1 a pour but de faire manipuler `docker` et `docker-machine`, la base pour comprendre le principe de la containerisation de services. Voyez ce projet comme une initiation Mais avant de se lancer dans une telle aventure, il est important de vous mettre en garde sur certains points :

- Les exercices se feront sur `docker 1.12 minimum`. Si vous n'êtes pas sûr de votre version de `docker`, faites un `docker version`. Brew fournit toujours la dernière version de Docker... au cas où.
- Il est surtout TRÈS important d'avoir le binaire `docker-machine` de disponible sur votre dump pour commencer votre excursion dans le milieu des containers et faire les bonus. Se référer à la doc de Docker disponible sur les internets pour l'installer si besoin.
- Prenez le temps de lire la documentation. De toute façon, vous n'aurez pas le choix...

## III.2 Your Hitchhicker guide to the containers

En parallèle de ce sujet, nous vous recommandons VIVEMENT d'avoir la documentation officielle de Docker à portée de main pour manipuler correctement vos futurs containers. Nous vous demandons aussi de regarder la documentation des containers officiels de chaque service que nous vous demanderons d'implémenter tout au long de ce projet.

Vous allez très vite vous rendre compte que vous n'avez pas besoin de réinventer la roue, chaque fois que vous voulez développer une application ou mettre en place un service.

Voilà donc quelques liens très utiles pour votre sujet :

- [L'excellente documentation officielle de Docker](#)
- [Le Registry public de Docker](#)
- [Le Github officiel de Docker qui contient moult projets en devenir](#)
- [Le blog de Jessie Frazelle, ex main contributor sur Docker](#)
- [Son Github avec pleins de bonnes idées](#)

# Chapitre IV

## Consignes

- Le sujet sera corrigé par des humains.
- Les containers lors de la soutenance seront hébergés sur le dump de correction et en local.
- Interdiction de faire une soutenance avec un container sur une machine distante, relire la règle juste au-dessus au cas où vous n'auriez pas compris cette règle-ci.
- Votre rendu sera composé de 2 dossiers (+1 si vous faites les bonus)
  - 00\_how\_to\_docker
  - 01\_dockerfiles
  - 02\_bonus [BONUS]

Chaque dossier attend une certaine arborescence, indiquée dans le chapitre correspondant



Pour des soucis de stockage, nous vous recommandons de déplacer vos dossiers ".docker" et "VirtualBox VMs" de votre home vers votre goinfre et de faire des liens symboliques.

# Chapitre V

## Partie Obligatoire

### V.1 How to Docker

Cette première partie est consacrée à la découverte de Docker et de ses options. Dans votre dossier de rendu, créez un dossier `00_how_to_docker` dans lequel vous allez déposer la solution de chaque exercice.

Si ce n'est pas déjà fait, installez `docker`, `docker-machine` et `virtualbox`. `docker` et `docker-machine` peuvent s'installer via Brew, alors que `virtualbox` s'installe par le Managed Software Center.

Vérifiez qu'un `docker version` vous affiche bien la version de docker et que vous êtes bien sous docker 1.12 minimum (NDR : à l'heure où j'écris ces lignes, une nouvelle notation de versionning est utilisée, et la version la plus récente est la 17.03.0-ce-mac2 ).

Vous serez tentés d'utiliser `Docker for Mac`, et je vous comprends. Je vous rassure, vous l'utiliserez pour les autres projets liés à Docker. Mais utiliser `docker-machine` vous permettra de faire du clustering en local assez facilement, ce qui vous sera demandé durant ce projet !

Vous allez rendre dans le dossier `00_how_to_docker`, là où les commandes correspondantes aux exercices demandés en les nommant par le numéro d'exercice.

```
$> 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 Exercices

Pour chacun de ces exercices, nous vous demandons de donner la ou les commande(s) shell pour :

1. Créer une machine virtuelle avec **docker-machine** utilisant le driver **virtualbox** et ayant pour nom **Char**
2. Récupérer l'adresse IP de la machine virtuelle **Char**
3. Assigner les variables spécifiques à la machine virtuelle **Char** dans l'env courant de votre terminal, de sorte que vous pouvez lancer la commande **docker ps** sans erreurs. Une seule commande est attendue pour fixer les 4 variables d'environnement et il vous est interdit d'utiliser le builtin de votre shell pour set à la main ces variables.
4. Récupérer depuis le Docker Hub le container **hello-world** disponible sur le Docker Hub.
5. Lancer le container **hello-world** et faire en sorte que le container affiche bien son message d'accueil, puis le quitte.
6. Lancer un container **nginx** disponible sur le Docker Hub en tâche de fond. Le container lancé doit avoir pour nom **overlord**, doit pouvoir redémarrer de lui-même et doit avoir le port 80 du container rattaché au port 5000 de **Char**. Vous pouvez vérifier le fonctionnement de votre container en allant sur un <http://<ip-de-char>:5000> comme URL sur votre navigateur internet.
7. Récupérer l'adresse IP interne du container **overlord** sans lancer son shell et en une commande.
8. Lancer un shell depuis un container **alpine**, en faisant en sorte que vous puissiez directement interagir avec le container via votre terminal et que le container se supprime à la fin de l'exécution du shell.
9. Depuis le shell d'un container **debian**, faire en sorte d'installer via le gestionnaire de paquets du container, de quoi compiler un code source en C et le pusher sur un repo git (en veillant avant de bien mettre à jour le gestionnaire de paquets et les paquets présents de base dans le container). Seules les commandes à effectuer dans le container sont demandées pour cet exercice.
10. Créer un volume **hatchery**
11. Lister les volumes Docker créés sur la machine... je dis bien VOLUMES

12. Lancer un container **mysql** en tâche de fond. Il devra aussi pouvoir redémarrer de lui-même en cas d'erreur et faire en sorte que le mot de passe root de la base de données soit **Kerrigan**. Vous ferez aussi en sorte que la base de données soit stockée dans le volume **hatchery**, que le container crée directement une base de données qui aura comme nom **zerglings** et le container s'appellera **spawning-pool**.
13. Afficher les variables d'environnement du container **spawning-pool** en une seule commande, histoire d'être sûr que vous avez bien configuré votre container.
14. Lancer un container **wordpress** en tâche de fond, pour le lulz. Le container doit avoir pour nom **lair**, le port 80 du container doit être bindé au port 8080 de la machine virtuelle et doit pouvoir utiliser le container **spawning-pool** comme service de base de données. Vous pouvez tenter d'accéder à **lair** sur votre machine via un navigateur en rentrant l'adresse IP de la machine virtuelle comme URL. Bravo, vous venez de deployer un site Wordpress fonctionnel en 2 commandes !
15. Lancer un container **phpmyadmin** en tâche de fond. Le container doit avoir pour nom **roach-warden**, le port 80 du container doit être bindé au port 8081 de la machine virtuelle et doit pouvoir faire en sorte d'aller explorer la base de données contenue dans le container **spawning-pool**.
16. Consulter les logs en temps réel du container **spawning-pool** sans exécuter son shell pour autant.
17. Afficher l'ensemble des containers actuellement actifs sur la machine virtuelle **Char**.
18. Relancer le container **overlord**
19. Démarrer un container qui se nommera **Abathur**. **Abathur** sera un container **Python** en version 2-slim, qui aura son dossier **/root** bindé à un dossier du **HOME** de votre host, ainsi que le port 3000 bindé au port 3000 de votre machine virtuelle. Vous personnaliserez ce container de telle sorte que vous puissiez utiliser le micro-framework **Flask** dans sa dernière version. Vous devrez faire en sorte qu'une page html renvoyant un "Hello World" dans des balises **<h1>**, soit servie par **Flask**. Vous testerez la bonne mise en place de votre container, en accédant via curl ou navigateur web, à l'adresse IP de votre machine virtuelle sur le port 3000. Vous listerez aussi toutes les commandes nécessaires dans votre rendu.
20. Créer un swarm local où la machine virtuelle **Char** en est le manager.
21. Créer une autre machine virtuelle avec **docker-machine** utilisant le driver **virtualbox** et ayant pour nom **Aiur**
22. Basculer **Aiur** comme esclave du swarm local où **Char** est leader (La commande de prise de contrôle de **Aiur** n'est pas demandée).
23. Créer un réseau interne de type overlay que vous nommerez **overmind**.
24. Lancer un SERVICE **rabbitmq** qui aura pour nom **orbital-command**. Vous devrez définir un user et un mot de passe spécifiques à l'utilisation du service RabbitMQ, et ceux-ci seront à votre libre convenance. Ce service sera sur le réseau **overmind**

25. Lister l'ensemble des services du swarm local.
26. Lancer un service `42school/engineering-bay` en 2 répliques et faire en sorte que le service fonctionne (se référer à la doc fournie dans [hub.docker.com](https://hub.docker.com)). Ce service s'appellera `engineering-bay` et sera sur le réseau `overmind`
27. Récupérer les logs en continu d'une des tasks du service `engineering-bay`
28. ... Damn, des zergs sont en train d'attaquer `orbital-command` et couper le service `engineering-bay` ne servira à rien... Vous devez envoyer des Marines pour les éliminer... Lancer un service `42school/marine-squad` en 2 répliques, et faites en sorte que le service fonctionne (se référer à la doc fournie dans [hub.docker.com](https://hub.docker.com)). Ce service s'appellera ... `marines` et sera sur le réseau `overmind`
29. Afficher l'ensemble des tâches du service `marines`.
30. Mettre à jour le nombre de répliques du service `marines` à 20, car on n'a jamais assez de Marines pour annihiler du Zerg. (Pensez aussi à regarder les tâches et les logs du service, vous allez voir, c'est marrant).
31. Forcer l'arrêt et supprimer l'ensemble des services sur le swarm local, en une seule commande
32. Forcer l'arrêt et supprimer l'ensemble des containers (tous états confondus), en une seule commande.
33. Supprimer l'ensemble des images de containers stocké sur la machine virtuelle `Char`, en une seule commande aussi.
34. Supprimer la machine virtuelle `Aiur` autrement qu'avec un `rm -rf`

# Chapitre VI

## Dockerfiles

Alors, c'était bien hein ?

Maintenant, il est temps de passer la seconde. Docker vous permet aussi de créer vos PROPRES containers pour vos PROPRES applications ! Et c'est aussi l'objet de ce chapitre. Encore heureux, Docker pense à tout en vous laissant programmer des Dockerfiles (un Makefile pour Docker... ouais bon, je pense que vous avez compris la nuance). Les Dockerfiles sont des fichiers utilisant une syntaxe spécifique qui réutilise une image de base ou un container existant pour y ajouter vos propres dépendances et vos propres fichiers.


Vous allez constater aussi que chaque commande construite par vos Dockerfiles génère un layer, réutilisable dans d'autres Dockerfiles ou d'autres versions du même Dockerfile, pour éviter la redondance de données. Génial non ?

Mais avant de commencer à faire vos propres containers, assurez-vous que votre machine virtuelle soit bien vide de toute image et container actif résiduel. On va partir de la base et faire des applications de plus en plus complexes.

Créez un dossier `01_dockerfiles` dans lequel vous rangerez tous les Dockerfiles demandés ensuite, dans des dossiers séparés (ex00 / ex01 ...).


Un lien pour bien vous aider à concevoir des Dockerfiles de qualité : [Dockerfile Best Practices](#)

## VI.1 Exercice 00 : vim/emacs

	Exercice : 00
Exercice 00 : vim/emacs	
Dossier de rendu : <i>ex00/</i>	
Fichiers à rendre : <b>Dockerfile</b>	
Fonctions Autorisées : -	


Depuis une image **alpine**, vous ajouterez à votre container l'éditeur de texte de votre choix entre **vim** ou **emacs**, qui se lancera au démarrage de votre container.

## VI.2 Exercice 01 : BYOTSS

	Exercice : 01
Exercice 01 : BYOTSS	
Dossier de rendu : <i>ex01/</i>	
Fichiers à rendre : <b>Dockerfile + Scripts éventuels</b>	
Fonctions Autorisées : -	

Depuis une image **debian**, vous ajouterez les sources adéquates pour créer un serveur TeamSpeak, serveur qui se lancera au démarrage de votre container. Celui-ci est considéré comme valide si au moins un utilisateur peut se connecter dessus et discuter normalement (pas de configuration hasardeuse), donc créez votre Dockerfile avec les options adéquates. Vous devez faire en sorte que les sources soient récupérées au build, elles ne doivent pas être présentes dans le dépôt.

## VI.3 Exercice 02 : Dockerfile in a Dockerfile... in a Dockerfile ?


	Exercice : 02
Exercice 02 : Dockerfile in a Dockerfile... in a Dockerfile ?	
Dossier de rendu : <i>ex02/</i>	
Fichiers à rendre : <b>Dockerfile</b>	
Fonctions Autorisées : -	

Vous allez créer votre premier Dockerfile containerisateur d'application Rails. Ce Dockerfile sera un peu spécial car il sera générique et devra être appelé dans un autre Dockerfile, qui devrait ressembler un peu à ça :

```
FROM      ft-rails:on-build
EXPOSE    3000
CMD       ["rails", "s", "-b", "0.0.0.0", "-p", "3000"]
```

Votre container générique devra, depuis un container **ruby**, installer toutes les dépendances nécessaires, puis **copier votre application** rails dans le dossier **/opt/app** de votre container. Au build, Docker doit faire l'installation des gems spécifiques, ainsi que les migrations et la population de la db de votre application. Le Dockerfile-fils devra exposer les bons ports et lancer le serveur de rails (voir exemple ci-dessus). Si vous ne connaissez pas les commandes, il est temps de faire un tour sur la doc de [Ruby on Rails](#).

## VI.4 Exercice 03 : What does the Fox say ?

	Exercice : 03
Exercice 03 : What does the Fox say ?	
Dossier de rendu : <i>ex03/</i>	
Fichiers à rendre : <b>Dockerfile</b>	
Fonctions Autorisées : -	

Docker peut être pratique pour pouvoir tester une application encore en développement sans créer de la pollution dans vos fichiers. Vous allez par ailleurs, devoir concevoir un Dockerfile qui, au build, récupère la version actuelle de [Gitlab - Community Edition](#) l'installe avec toutes les dépendances et les configurations nécessaires et lance l'application (HTTP et SSH). Le container est jugé valide, si vous pouvez accéder au client web, et si vous êtes capables d'utiliser correctement avec Gitlab et d'interagir via GIT avec ce container. Bien sûr, vous ne devrez pas utiliser le container officiel de Gitlab, ce serait un peu tricher...

# Chapitre VII

## Partie Bonus

Pour les bonus, maintenant que vous avez saisi toute la subtilité de Docker, vous pouvez allègrement vous amuser !

À vous de créer vos futurs environnements de travail, tels que :

- Un environnement de dev pour faire du nodejs, mais en utilisant **yarn** plutôt que **npm**
- PAMP (!) avec docker-compose, mais en utilisant **MariaDB** plutôt que **MySQL**
- Recréer une MEAN Stack complète
- Des Dockerfiles pour vos projets de C, Ruby, Go, Ocaml, Rust...
- etc.

Pensez aussi que vous n'avez encore rien vu de la puissance de Docker. Vous pouvez :

- Tenter de containeriser une configuration VPN pour vos containers
- Essayer de clusteriser plusieurs machines sur différents services cloud avec Docker Swarm
- Vous lancer dans la construction d'une image de base, comme faire votre propre container debian ou archlinux
- Containeriser le serveur de votre jeu préféré, comme Minecraft (?)
- etc.

Dans tous les cas, nous attendons que vous mettiez tout à disposition dans le dossier `02_bonus` de votre rendu.

Have Fun!!

# Chapitre VIII

## Rendu et peer-évaluation

Rendez votre travail sur votre dépôt GiT comme d'habitude. Seul le travail présent dessus sera évalué en soutenance.