ETUDES, DÉVELOPPEMENT & INTÉGRATION

DOCKER

Pratique des architectures à base de conteneurs

Pierre-Yves Cloux
Thomas Garlot
Johann Kohler

DUNOD

Illustration de couverture : © eyetronic - Fotolia.com

11 rue Paul Bert, 92240 Malakoff
www.dunod.com

ISBN 978-2-10-075350-5

Table des matières

Troisième partie – Apprendre Docker

Avant-propos

Pendant longtemps, déployer du code en production revenait à tenter de transporter de l'eau entre ses mains : c'était fonctionnel, mais pas vraiment optimal. Comme l'eau filant entre les doigts, il manquait presque nécessairement une partie des données de configuration lors du déploiement, ceci en dépit d'efforts méthodologiques, documentaires et humains conséquents.

L'apport de la virtualisation

La virtualisation a tenté de répondre à cette problématique (parmi d'autres) sans apporter une réponse complètement satisfaisante. En effet, bien qu'offrant une isolation vis-à-vis de l'architecture matérielle, qui se standardise de plus en plus, la machine virtuelle reste... une machine. Elle exécute un système d'exploitation dont les paramètres peuvent différer d'un environnement à l'autre.

Des outils comme Chef ou Puppet résolvent une partie du problème, mais n'offrent encore une fois qu'une couche d'abstraction partielle. Enfin, le déploiement d'application sur la base d'une image de VM (pour *Virtual Machine*) est lourd (plusieurs gigaoctets, y compris pour les OS les plus compacts).

Les architectures à base de conteneurs

Avec Docker, nous sommes entrés dans l'ère des architectures à base de « conteneur ». On parle aussi de « virtualisation de niveau système d'exploitation » par opposition aux technologies à base d'hyperviseurs (comme VMWare ou VirtualBox) qui cherchent à émuler un environnement matériel.

Contrairement à une VM, un conteneur n'embarque pas un système d'exploitation complet. Il repose pour l'essentiel sur les fonctionnalités offertes par l'OS sur lequel il s'exécute. L'inconvénient de cette approche est qu'elle limite la portabilité du conteneur à des OS de la même famille (Linux dans le cas de Docker). Cette approche, en revanche, a l'avantage d'être beaucoup plus légère (les conteneurs sont nettement plus petits que les VM et plus rapides au démarrage) tout en offrant une isolation satisfaisante en termes de réseau, de mémoire ou de système de fichiers.

Étonnamment, le concept de conteneur et son implémentation ne sont pas nouveaux. La version de OpenVZ pour Linux date, par exemple, de 2005, de même que Solaris Zone ou FreeBSD jail. Docker a changé la donne en simplifiant l'accès à cette technologie par l'introduction d'innovations, comme la mise à disposition d'un langage de domaine (DSL ou *Domain Specific Language*) permettant, au travers des fameux « Dockerfile », de décrire très simplement la configuration et la construction d'un conteneur.

Le propos de cet ouvrage

L'objet de cet ouvrage est d'offrir une approche à 360 degrés de l'écosystème Docker.

Docker, plus qu'une technologie, est en effet aujourd'hui un écosystème de solutions fourmillantes : Kitematic, Compose, Machine, Swarm.

Autour du *runtime* (moteur d'exécution) qui exécute les conteneurs (le Docker Engine), des outils complémentaires visent à conditionner, assembler, orchestrer et distribuer des applications à base de conteneurs. Timidement, des initiatives de standardisation voient le jour, laissant espérer une meilleure interopérabilité de celle qui prévaut aujourd'hui dans le domaine de la virtualisation.

Notre objectif dans cet ouvrage est donc multiple :

- aborder le concept de conteneur et d'architecture à base de conteneurs en décryptant les avantages proposés par cette approche ;
- apprendre à installer Docker sur un poste de travail ou dans un environnement serveur ;
- apprendre à utiliser Docker pour créer des images et manipuler des conteneurs ;
- étudier des architectures plus complexes au travers d'exemples complets : architectures multi-conteneurs, développement, intégration continue et implémentation d'un cluster multi-hôtes.

La structure du livre

Dans une première partie, nous expliquerons ce qu'est un conteneur, sur quels principes et quelles technologies il repose. Nous verrons aussi comment Docker se positionne parmi un nombre croissant d'acteurs et de logiciels importants.

Nous aborderons ensuite le concept de « CaaS », pour « *Container as a Service* », au travers de divers exemples. À cette occasion, nous utiliserons des outils tels que Swarm, Kubernetes, Mesos. Nous nous intéresserons aux liens entre les approches conteneurs et celles d'outils de gestion de configuration comme Puppet, Chef ou Ansible.

La seconde partie de cet ouvrage se focalise sur la prise en main de Docker en étudiant son installation sur un poste de travail ou dans un environnement serveur virtualisé ou physique. Nous commencerons alors à « jouer » avec des conteneurs pour comprendre par la pratique les concepts abordés dans la première partie.

La troisième partie du livre est consacrée à l'apprentissage de Docker. Nous y aborderons les commandes du client et les instructions du Dockerfile. Cette troisième partie peut être lue séquentiellement, mais aussi servir de référence.

La dernière partie du livre se compose de trois chapitres qui mettent en œuvres des cas d'usages permettant d'aborder des concepts plus avancés comme :

- les architectures multi-processus et multi-conteneurs ;
- la réalisation d'une chaîne d'intégration continue s'appuyant sur l'usage de conteneurs ;
- la mise en œuvre d'un cluster multi-hôtes en s'appuyant sur Docker Swarm.

À qui s'adresse ce livre

Cet ouvrage s'adresse à un public mixte « DevOps » :

- si vous êtes engagé dans une organisation de développement (en tant que développeur, architecte ou manager), vous trouverez les informations nécessaires pour maîtriser rapidement la conception d'images Docker, mais aussi pour la réalisation d'architectures multi-conteneurs ;
- si votre domaine est plutôt celui de l'exploitation, vous acquerrez les compétences nécessaires au déploiement de Docker sous Linux.

L'objectif de ce livre est double :

- il permet d'accélérer la prise en main de cette technologie en présentant des cas d'usages illustrés par des exemples didactiques ;
- il offre aussi une référence illustrée d'exemples du langage DSL et des commandes de Docker.

Compléments en ligne

Le code source et les exemples de ce livre sont distribués via GitHub sur le repository public :
https://github.com/dunod-docker
Un erratum, des articles et des exemples complémentaires sont aussi distribués via le site http://www.docker-patterns.org.

Remerciements

L'écriture d'un livre est une activité consommatrice de temps qu'il n'est pas possible de mener à son terme sans l'assistance de nombreuses personnes.

Nous voudrions donc adresser nos remerciements à :

- Patrick Moiroux, pour son expertise de Linux en général, des architectures web distribuées et de Docker en particulier ;
- Bernard Wittwer, pour avoir attiré notre attention, il y a plus de trois ans, sur un petit projet OpenSource sympathique nommé Docker dont il percevait déjà le potentiel ;
- Marc Schär, notre stagiaire de compétition, devenu expert de Kubernetes ;

- tous les contributeurs au projet open source BizDock (http://www.bizdock-project.org/) qui a été notre premier contact avec le monde des conteneurs et Docker ;
- nos épouses et nos enfants pour leur patience en regard du temps que nous avons passé isolés du monde extérieur à expérimenter des logiciels et à en décrire le fonctionnement.

PREMIÈRE PARTIE

Les conteneurs : principes, objectifs et solutions

Cette première partie vise à présenter :

- les origines du concept de conteneur ;
- l'apport de Docker à cette technologie déjà ancienne ;
- comment les conteneurs autorisent la réalisation de nouvelles architectures informatiques ;
- comment les conteneurs colonisent les solutions de gestion d'infrastructure.

Cette première partie comprend deux chapitres. Le premier décrit ce qu'est un conteneur d'un point de vue technique et conceptuel. Le second chapitre présente un survol complet des solutions de gestion d'infrastructure à base de conteneurs : des plus modernes (que l'on nomme également CaaS[1]) aux plus classiques pour lesquelles les conteneurs apportent les avantages décrits dans le chapitre 1.

1. CaaS est l'acronyme en anglais de *Container as a Service*, soit en français Conteneur comme un service.

1

Les conteneurs et le cas Docker

Les éléments de base de l'écosystème

L'objectif de ce chapitre est de décrire les concepts qui ont présidé à l'émergence de la notion de conteneur logiciel. Nous présenterons les briques de base sur lesquelles les conteneurs (et plus spécifiquement les conteneurs Docker) reposent. Puis nous expliquerons comment sont construits et distribués les conteneurs Docker. Enfin, nous nous intéresserons aux différents éléments des architectures à base de conteneurs : Docker et les autres.

À l'issue de ce chapitre vous comprendrez ce qu'est un conteneur, ce qu'est Docker et quels sont les concepts architecturaux et logiciels qu'il implémente. Vous saurez aussi quels sont les acteurs de cet écosystème.

> Attention : dans ce chapitre, comme dans la suite de cet ouvrage, nous allons aborder la question de la conception des architectures à base de conteneurs. Néanmoins, nous le ferons par le prisme de Docker qui en est l'implémentation la plus populaire. De ce fait, nous nous focaliserons sur les OS qui sont officiellement supportés par Docker (soit Linux, Windows et dans une moindre mesure Mac OS X). Nous tenons à indiquer que des implémentations de conteneurs existent aussi pour d'autres systèmes, comme FreeBSD ou Solaris. Néanmoins, nous ne les aborderons pas dans le cadre de ce livre.

1.1 LA CONTENEURISATION

Les conteneurs logiciels cherchent à répondre à la même problématique que les conteneurs physiques aussi appelés conteneurs intermodaux, ces grandes boîtes de

métal standardisées qui transportent de nos jours l'essentiel des biens matériels par camion, train, avion ou bateau.

Nous allons donc faire un peu d'histoire.

1.1.1 L'histoire des conteneurs intermodaux

Avant l'apparition de ces conteneurs standardisés, les biens étaient manipulés manuellement. Les Anglo-Saxons utilisent le terme de *break bulk cargo*, ce que nous pourrions traduire par « chargement en petits lots ». Plus simplement, les biens étaient chargés individuellement par des armées de travailleurs.

Les biens étaient, par exemple, produits dans une usine, chargés un par un dans un premier moyen de transport jusqu'à un hangar (généralement près d'un port ou d'une gare) où avait lieu un déchargement manuel. Un second chargement survenait alors depuis ce premier lieu de stockage jusque dans le moyen de transport longue distance (par exemple un bateau). Ce second chargement était lui aussi manuel. En fonction des destinations, des contraintes géographiques et légales, cette séquence de chargement et déchargement pouvait avoir lieu plusieurs fois. Le transport de biens était donc coûteux, lent et peu fiable (biens abîmés ou perdus).

Les premières tentatives de standardisation du transport de biens sont intervenues en Angleterre à la fin du XVII^{e} siècle, essentiellement pour le transport du charbon. L'un des exemples bien connus est celui de la ligne de chemin de fer Liverpool – Manchester qui utilisait des boîtes en bois de taille standardisée qui étaient ensuite déchargées par grues sur des charrettes tirées par des chevaux.

Dans les années trente, la Chambre internationale de commerce a mené des tests afin de standardiser les conteneurs en vue de faire baisser les coûts de transport dans le contexte de la grande dépression de 1929. En 1933, le Bureau international des conteneurs ou BIC est établi à Paris (et il s'y trouve toujours) pour gérer la standardisation de ces grosses boîtes métalliques dont le succès n'a fait que croître depuis.

Figure 1.1 — Conteneurs intermodaux

Aujourd'hui, on estime qu'il y a plus de 20 millions de conteneurs en circulation dans le monde.

1.1.2 Les avantages d'un transport par conteneur

Les avantages du transport par conteneur intermodal sont liés à ce que les Anglo-Saxons nomment *separation of concerns* (en français ségrégation des problèmes ou ségrégation des responsabilités).

Les transporteurs délèguent à leurs clients le remplissage du conteneur qui est ensuite scellé avant le transport. De ce fait, le transporteur n'a pas à s'occuper de la nature précise des biens transportés. Il doit s'assurer que le conteneur dans son ensemble arrive intact. Il assure donc la traçabilité d'un objet dont les caractéristiques physiques sont standardisées. Il peut les empiler, choisir le mode de transport adapté à la destination, optimiser les coûts et la logistique. Camions, bateaux, trains et grues vont pouvoir traiter ces biens, les arrimer correctement, qu'ils transportent des armes ou des couches-culottes.

La personne ou l'entreprise cliente a de son côté la garantie, pour peu que le conteneur ne soit pas perdu ou ouvert, que ses biens arriveront dans le même état et arrangement que lors du chargement. Le client n'a donc pas à se soucier de l'infrastructure de transport.

Cette séparation des responsabilités est à l'origine de la baisse drastique des coûts du transport de marchandises. Grâce à la standardisation et à l'automatisation qu'elle rend possible, les infrastructures de transport se sont mécanisées, automatisées et donc fiabilisées.

1.1.3 Extrapolation au monde logiciel

Comme nous l'avons évoqué en introduction, le monde informatique reste profondément organisé autour de l'aspect artisanal de la livraison du produit logiciel. Qu'il s'agisse d'un éditeur livrant un logiciel à un client ou de la mise en exploitation d'un développement, de très nombreux incidents survenant au cours du cycle de vie d'un logiciel sont liés à cette problématique.

C'est justement ce que la technologie des conteneurs logiciels cherche à résoudre.

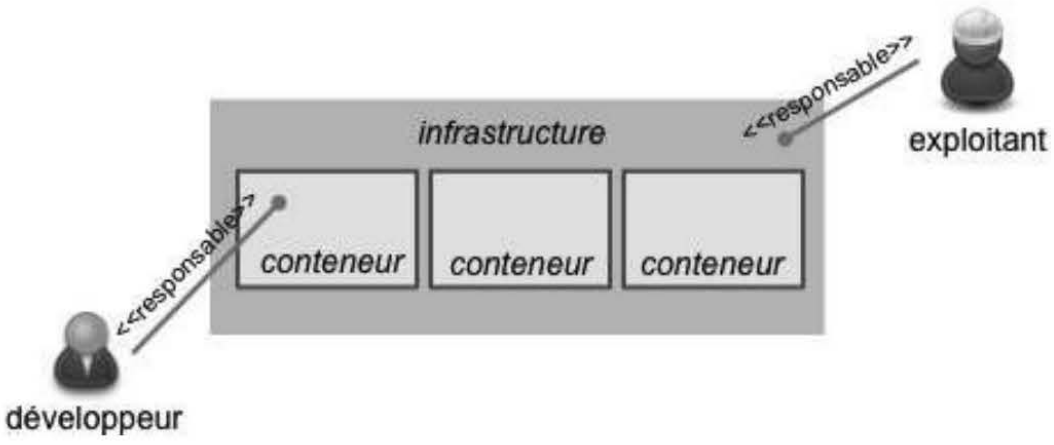

Figure 1.2 — Séparation des responsabilités dans une architecture à conteneurs

Dans une architecture à base de conteneurs :

- le contenu du conteneur, c'est-à-dire le code et ses dépendances (jusqu'au niveau de l'OS), est de la responsabilité du développeur ;
- la gestion du conteneur et les dépendances que celui-ci va entretenir avec l'infrastructure (soit le stockage, le réseau et la puissance de calcul) sont de la responsabilité de l'exploitant.

1.1.4 Les différences avec la virtualisation matérielle

Étudions l'empilement des couches dans le cas du déploiement de trois logiciels sur un même hôte (*host*) sans machine virtuelle ou conteneur.

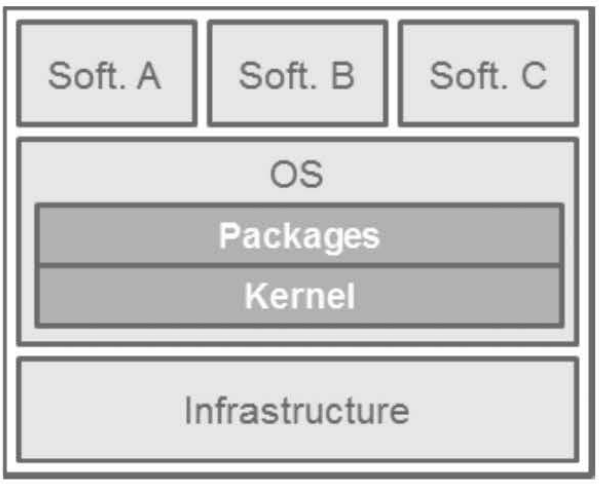

Figure 1.3 — Installation native de trois logiciels sur un même hôte

Dans ce type d'installation, on imagine que plusieurs situations problématiques peuvent survenir :

- les différents logiciels peuvent interagir entre eux s'ils n'ont pas été conçus par le même éditeur. Ils pourraient, par exemple, nécessiter des packages (bibliothèques, extensions) ou des versions de système d'exploitation différentes. Ils peuvent aussi ouvrir des ports réseaux identiques, accéder aux mêmes chemins sur le système de fichiers ou encore entrer en concurrence pour les ressources I/O ou CPU ;
- toute mise à jour de l'OS hôte va nécessairement impacter tous les logiciels qui tournent dessus ;
- chaque mise à jour d'un logiciel pourrait entraîner des impacts sur les autres.

L'expérience montre que l'exécution, sur le même système, de logiciels fournis par des éditeurs différents qui n'auraient pas testé cette cohabitation est très souvent problématique.

La virtualisation matérielle de type VMWare, VirtualBox ou HyperV n'assure-t-elle pas déjà cette séparation des responsabilités ?

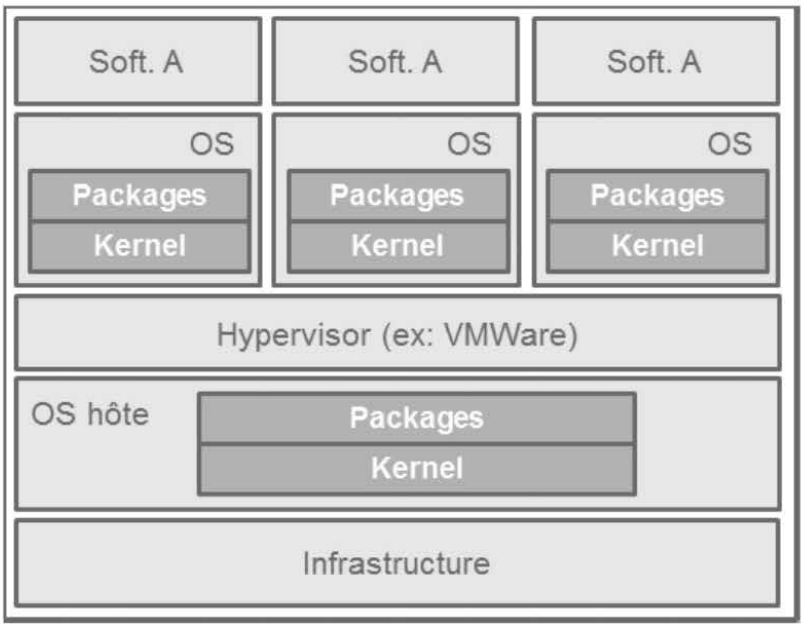

Figure 1.4 — Virtualisation matérielle : trois VM sur le même hôte

La réponse est « oui, mais... ».

Dans les faits, comme nous le voyons sur la figure précédente, cette virtualisation matérielle offre un niveau d'isolation élevé. Chaque logiciel se trouve dans son bac à sable (*sandbox* en anglais). Les problèmes évoqués précédemment sont donc résolus, mais d'autres apparaissent :

- le poids d'une machine virtuelle est tout d'abord très important. Une machine virtuelle est une machine et, même avec un système d'exploitation minimal, un OS moderne consommera difficilement moins de quelques Go de mémoire. La distribution de ce type de package demandera une bande passante réseau conséquente ;
- la machine virtuelle embarque trop d'éléments. Elle ne laisse pas le choix à l'exploitant (selon la manière dont elle aura été configurée) de choisir librement ses caractéristiques, comme le type de stockage, le nombre de CPU utilisés, la configuration réseau. Évidemment, les solutions de gestion d'environnements virtualisés (par exemple, vCenter de VMWare) offrent des solutions, mais celles-ci ont presque toujours des impacts sur le logiciel. Ce dernier ne peut pas être conçu sans savoir comment il va être exécuté.

Une architecture à base de conteneurs offre une solution de compromis. Le conteneur offre l'isolation permettant à un développeur d'embarquer l'ensemble des dépendances logicielles dont il a besoin (y compris les dépendances de niveau OS). De plus, un conteneur s'appuie sur le noyau (*kernel*) du système d'exploitation hôte[1].

1. Sur la figure 1.5, le lecteur attentif aura noté les termes « cgroups » et « namespaces ». Nous les définirons dans le paragraphe suivant. À ce stade, nous dirons qu'il s'agit d'extensions du noyau Linux qui rendent possible la réalisation de conteneurs « isolés » les uns des autres.

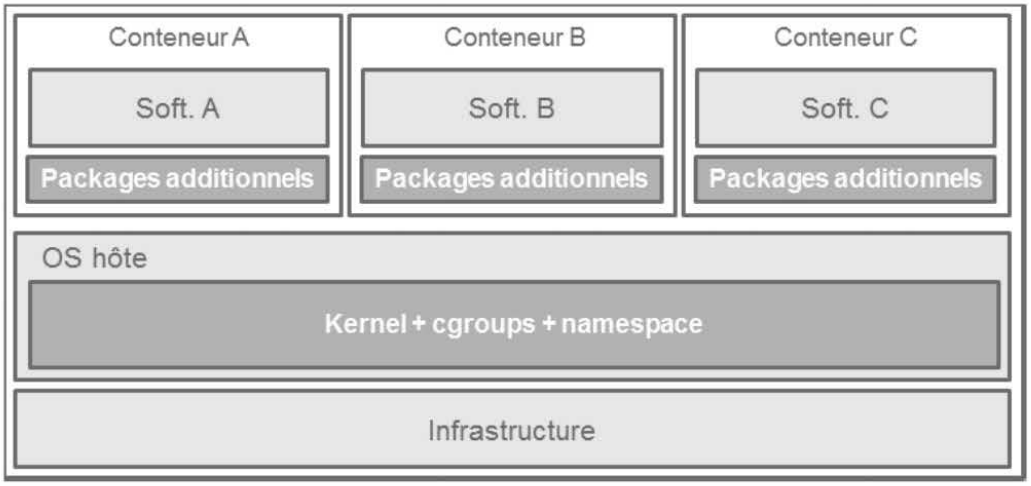

Figure 1.5 — Architecture à conteneurs

Il est donc très léger et démarre presque aussi vite que le processus qu'il encapsule. Le nombre de conteneurs qu'un même hôte peut exécuter est donc nettement plus élevé que son équivalent en machines virtuelles.

Des conteneurs dans une VM

Qu'en est-il du risque d'impact sur le logiciel en cas de mise à jour du système d'exploitation hôte ? Effectivement, dans le cas d'une architecture à conteneurs, si le noyau de l'OS est mis à jour, il peut affecter les conteneurs qu'il exécute. Rien n'empêche donc de revenir dans ce type de cas à l'option VM qui va augmenter l'isolation, tout en conservant la possibilité d'exécuter plusieurs conteneurs dans la même VM. C'est d'ailleurs ce type d'option que Microsoft met en avant avec HyperV-Containers, tout en offrant aussi la possibilité de faire tourner des conteneurs nativement sur l'OS hôte.

Il existe par ailleurs des projets, comme Fedora Atomic ou CoreOS, visant à offrir des OS hôtes minimaux dédiés à l'exécution de conteneurs et se focalisant uniquement sur les problématiques d'infrastructure. L'usage de ces OS hôtes spécialisés va limiter leur exposition aux attaques et aux bugs pouvant nécessiter des mises à jour.

Nous verrons aussi plus tard que Docker apporte deux autres avantages aux solutions à base de conteneurs : la possibilité de décrire formellement comment construire le conteneur et un format d'image standardisé. Cette capacité fondamentale est probablement la clef du succès de Docker.

Maintenant que nous sommes convaincus des avantages des architectures à base de conteneurs, nous allons en étudier les fondements techniques.

1.2 LES FONDATIONS : LINUX, CGROUPS ET NAMESPACES

Docker est une solution open source de conteneurs Linux qui s'appuie elle-même sur d'autres composants eux aussi ouverts. Ces briques de base sont en fait communes à tous les types de conteneurs Linux.

Initialement, Docker utilisait notamment LXC (*Linux Containers*) comme implémentation (on parle de driver), mais a depuis développé sa propre bibliothèque de bas niveau nommée libcontainer[1]. Ce driver encapsule les fonctionnalités fondamentales, proches du noyau du système d'exploitation, dont la combinaison permet la virtualisation.

Nous allons étudier ces différents composants de base pour :

- en comprendre le rôle fonctionnel ;
- comprendre les différentes étapes qui ont abouti au concept de conteneur.

1.2.1 Docker = Linux

Il est en premier lieu essentiel de rappeler que Docker est une technologie liée à Linux, son noyau (*Linux Kernel*) et certains de ses services orchestrés par la libcontainer de Docker.

Comment Docker marche-t-il sous Windows ou Mac OS X ?

Eh bien il ne marche pas sous ces OS, du moins pas nativement pour le moment.

Pour faire tourner un Docker Engine sous Windows, il était, encore jusqu'à récemment, nécessaire d'y installer un OS Linux. La solution usuellement pratiquée consiste à s'appuyer sur un logiciel de virtualisation matérielle : VirtualBox[2] dans le cas de Docker.

> VirtualBox est un hyperviseur open source (disponible en licence GPL) aujourd'hui géré par Oracle. Il est donc très fréquemment utilisé par des solutions open source souhaitant disposer de fonctionnalités de virtualisation sans contraintes de licences.

Comme nous le verrons dans le chapitre suivant, l'installation de la Docker Toolbox (le kit de démarrage de Docker en environnement PC) sous Windows ou Mac OS X inclut VirtualBox et une image Linux minimale qui fournit les services nécessaires à l'exécution du Docker Engine. L'installation n'est donc pas native et la complexité de gestion des terminaux, comme les performances d'accès au système de fichiers, limite très fortement l'intérêt de Docker sous ces OS Desktop.

1. Notons que ce driver d'exécution est ce qui fait le pont entre le moteur d'exécution des conteneurs et le système d'exploitation. Dans le cas de l'implémentation native sous Windows, c'est ce composant que les équipes de Microsoft ont dû réécrire pour l'adapter à Windows.
2. https://www.virtualbox.org/

Docker natif pour Windows Server 2016

À l'heure où nous écrivons ces lignes, Microsoft et Docker ont annoncé que la version 2016 de Windows Serveur inclurait la possibilité d'exécuter des conteneurs nativement. Les équipes de Microsoft, avec le support des ingénieurs de Docker, ont pour ce faire implémenté une sorte de libcontainer s'appuyant sur les capacités natives de Windows en lieu et place de Linux. Microsoft proposera aussi des images de base pour Windows (évidemment non disponibles en téléchargement libre). Il est donc possible que dans un futur indéterminé la Docker Toolbox pour Windows s'appuie sur cette implémentation native en lieu et place d'une machine virtuelle VirtualBox contenant un système Linux.

Pour comprendre comment fonctionnent les conteneurs, il est utile de décrire les services Linux de base sur lesquels s'appuie la libcontainer de Docker. Docker, tout comme d'autres implémentations de conteneurs, est basé sur deux extensions du kernel Linux :

- Cgroups ;
- Namespaces.

1.2.2 CGroups

CGroups[1] (pour *Control Groups*) permet de partitionner les ressources d'un hôte (processeur, mémoire, accès au réseau ou à d'autres terminaux). L'objectif est de contrôler la consommation de ces ressources par processus.

Prenons par exemple une machine sur laquelle serait installée une application web avec un front-end PHP et une base de données MySQL. CGroups permettrait de répartir la puissance de calcul (CPU) et la mémoire disponible entre les différents processus, afin de garantir une bonne répartition de la charge (et donc probablement des temps de réponse).

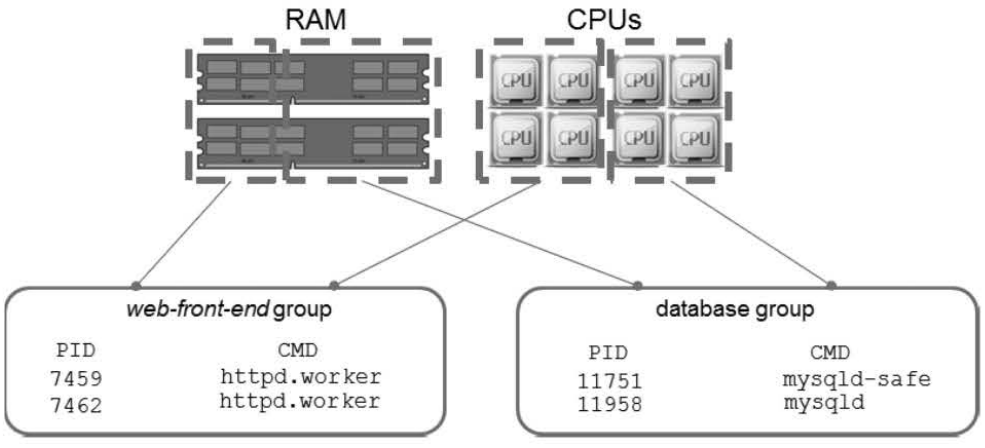

Figure 1.6 — Répartition de ressources grâce à CGroups

1. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

CGroups introduit une notion de contrôleur qui, comme son nom l'indique, a pour objectif de contrôler l'accès à une ressource pour une hiérarchie de processus. En effet, par défaut, les processus fils d'un processus associé à un group donné héritent des paramètres de leur parent.

1.2.3 Namespaces

Les Namespaces sont indépendants de CGroups, mais fonctionnent de concert. Ils permettent de faire en sorte que des processus ne voient pas les ressources utilisées par d'autres. Si CGroups gère la distribution des ressources, Namespaces apporte l'isolation nécessaire à la création de conteneurs.

L'ancêtre du mécanisme des Namespaces est la commande **chroot** qui existe au sein du système UNIX depuis 1979 !

La fonction de cette commande est de changer pour un processus donné le répertoire racine du système. Le processus en question a l'impression d'être à la racine du système. L'objectif étant, par exemple, pour des raisons de sécurité, d'empêcher l'utilisateur de se promener dans des sections du système de fichiers dont il n'aurait *a priori* pas l'usage. Il s'agit donc conceptuellement d'une sorte de virtualisation.

Namespaces étend ce concept à d'autres ressources.

Tableau 1.1 — Namespaces Linux

Namespace	Isole
IPC	Communication interprocessus
Network	Terminaux réseau, ports, etc.
Mount	Point de montage (système de fichiers)
PID	Identifiant de processus
User	Utilisateurs et groupes
UTS	Nom de domaines

1.2.4 Qu'est-ce qu'un conteneur finalement ?

Un conteneur est tout simplement un système de fichiers sur lequel s'exécutent des processus (de préférence un par conteneur) de manière :

- contrainte : grâce à CGroups qui spécifie les limites en termes de ressources ;
- isolée : grâce notamment à Namespaces qui fait en sorte que les conteneurs ne se voient pas les uns les autres.

1.3 LES APPORTS DE DOCKER : STRUCTURE EN COUCHES, IMAGES, VOLUMES ET REGISTRY

Comme nous l'avons précédemment expliqué, les conteneurs existent depuis longtemps, mais Docker a apporté des nouveautés décisives qui ont clairement favorisé leur popularisation.

L'une d'entre elle a trait à la manière dont Docker optimise la taille des conteneurs en permettant une mutualisation des données. C'est ce que nous allons expliquer dans cette section.

Nous allons aussi aborder la question de la persistance des données dans un conteneur, en expliquant la notion de volume.

Notez que nous expérimenterons ces concepts ultérieurement, à l'aide des outils Docker.

> Attention, l'apport du projet Docker à la technologie des conteneurs ne se limite pas aux concepts décrits dans ce paragraphe. Nous verrons par la suite, dans la seconde partie de cet ouvrage, que Docker a aussi changé la donne grâce à ses outils et en particulier grâce aux fameux Dockerfile.

1.3.1 La notion d'image

La notion d'image dans le monde conteneur est pratiquement synonyme de Docker. Comme nous l'avons déjà vu, un conteneur est un ensemble de fichiers sur lequel s'exécutent un ou plusieurs processus. On peut alors se poser la question suivante : mais d'où viennent ces fichiers ?

Construire un conteneur pourrait se faire à la main, en reconstruisant dans une partie du système de fichiers Linux une arborescence de fichiers : par exemple, un répertoire pourrait stocker les fichiers spécifiques au conteneur qui ne font pas partie du noyau de l'OS. Mais cette pratique serait fastidieuse et poserait le problème de la distribution et de la réutilisation.

L'un des apports essentiels de Docker est d'avoir proposé une manière de conditionner le contenu d'un conteneur en blocs réutilisables et échangeables : les images.

Ces images sont donc des archives qui peuvent être échangées entre plusieurs hôtes, mais aussi être réutilisées. Cette réutilisation est rendue possible par une seconde innovation : l'organisation en couches.

1.3.2 Organisation en couches : union file system

Les conteneurs Docker sont en fait des millefeuilles constitués de l'empilement ordonné d'images. Chaque couche surcharge la précédente en y apportant éventuellement des ajouts et des modifications.

En s'appuyant sur un type de système de fichiers Linux un peu particulier nommé *Union File System*[1], ces différentes couches s'agrègent en un tout cohérent : le conteneur ou une image elle-même réutilisable.

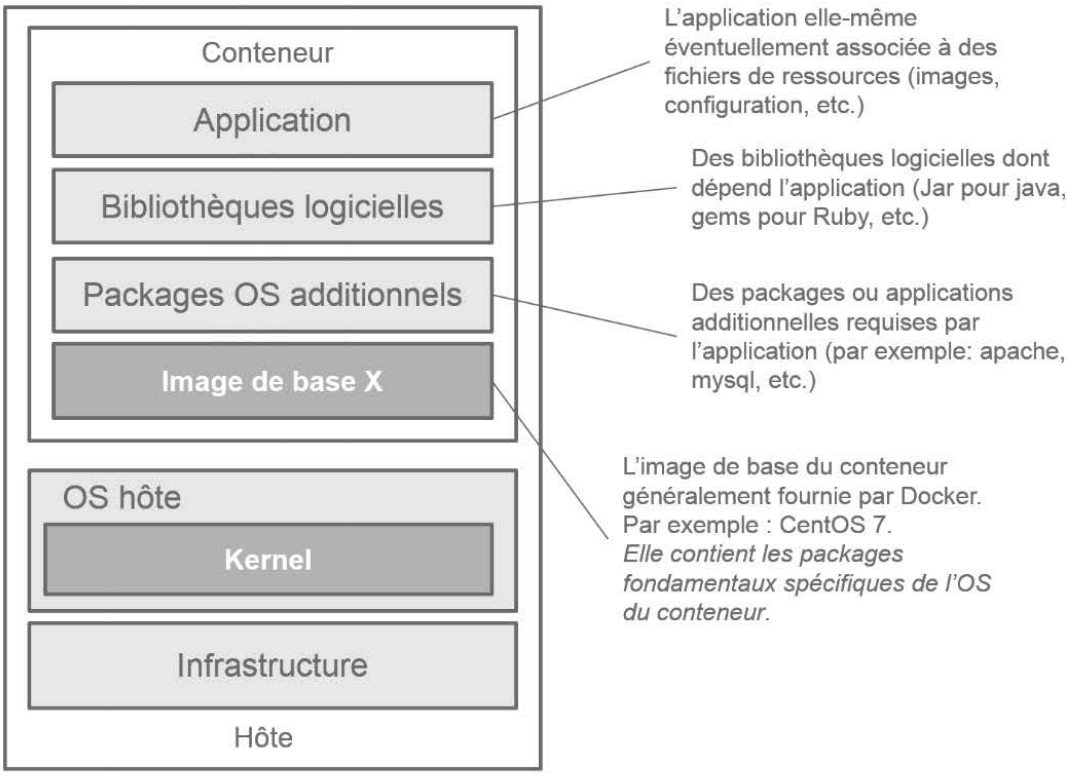

Figure 1.7 — Structure d'un conteneur au sein d'un hôte

Admettons que nous construisions une application s'appuyant sur deux conteneurs tournant sur le même hôte :

- un *front-end* basé sur une application JSP, tournant dans un serveur d'applications Tomcat[2], exécuté par une machine virtuelle Java 1.8 sur un OS CentOS 7 ;
- un *back-end* constitué d'une base de données Cassandra[3] s'exécutant elle aussi sur Java 1.8 et CentOS 7.

Lorsque nous allons installer le *front-end* (nous verrons comment par la suite), le système va charger plusieurs couches successives qui sont autant de blocs réutilisables. Lors de l'installation du conteneur de base de données, le système n'aura pas à charger l'image CentOS ou la machine virtuelle Java, mais uniquement les couches qui sont spécifiques et encore absentes du cache de l'hôte.

1. Nous verrons dans la suite de ce chapitre qu'il existe plusieurs implémentations de ce type de système de fichiers.
2. http://tomcat.apache.org/
3. http://cassandra.apache.org/

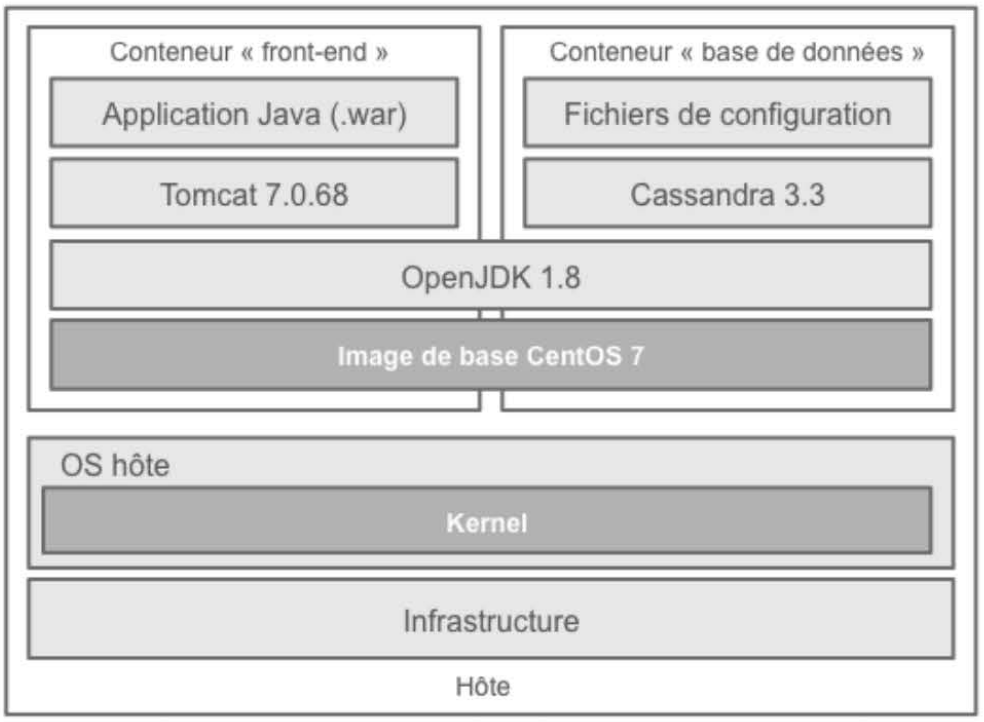

Figure 1.8 — Exemple d'application à plusieurs conteneurs

Par la suite, les modifications de l'application *front-end*, si elles se cantonnent à la modification de pages JSP, ne nécessiteront qu'un redéploiement de la couche supérieure.

Le gain de temps, de bande passante réseau et d'espace disque (par rapport à une machine virtuelle complète par exemple) est évident.

> Attention la représentation donnée des couches d'images dans la figure 1.8 est très simplifiée. En réalité, une image de base CentOS 7 est, par exemple, constituée de quatre couches. Mais le principe exposé d'empilement et d'agrégation de ces couches par l'intermédiaire d'un *union file system* reste le même.

1.3.3 Docker Hub et registry

Dans le paragraphe précédent, nous avons exposé les fondements de la structure d'un conteneur. Nous avons notamment évoqué la manière dont l'hôte pouvait charger des blocs d'images constituant les conteneurs.

Mais à partir d'où un hôte Docker va-t-il charger ces blocs ?

C'est le rôle d'un autre composant essentiel de l'architecture Docker : le registry.

Le registry est l'annuaire et le centre de stockage des images réutilisables. Sa principale instanciation est le Docker Hub public et accessible via l'URL https://hub.docker.com/.

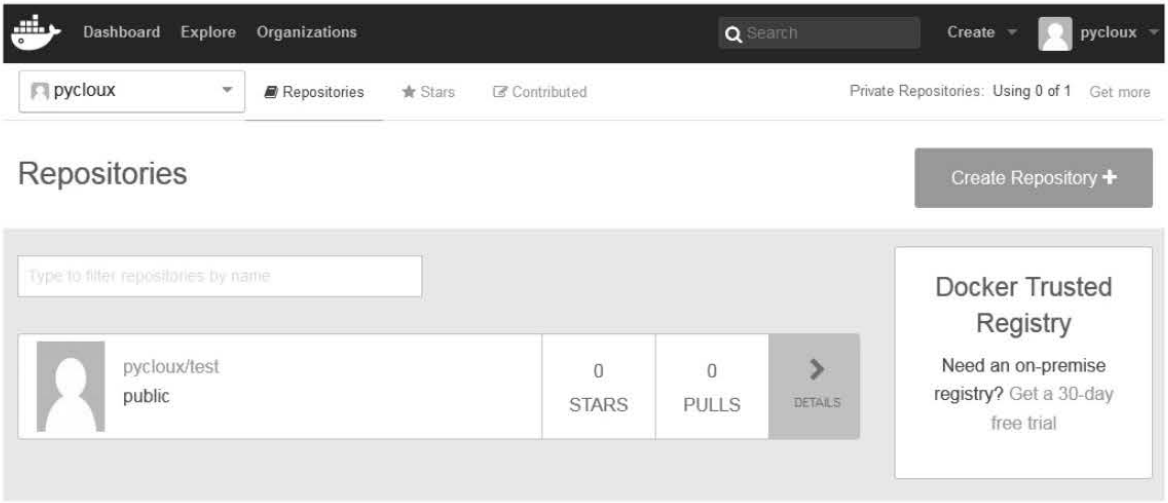

Figure 1.9 — Compte sur le Docker Hub

Il est possible, à tout à chacun, de se créer un compte gratuit sur ce registry pour publier une image. Il faudra débourser quelques dollars pour pouvoir en stocker d'autres.

Ce registry est public et héberge les images de base des principaux systèmes d'exploitation dérivés de Linux (Ubuntu, CentOS, etc.) et de nombreux logiciels courants (bases de données, serveurs d'application, bus de messages, etc.). Celles-ci, appelées images de base sont gérées par Docker Inc. et les organisations qui produisent ces logiciels.

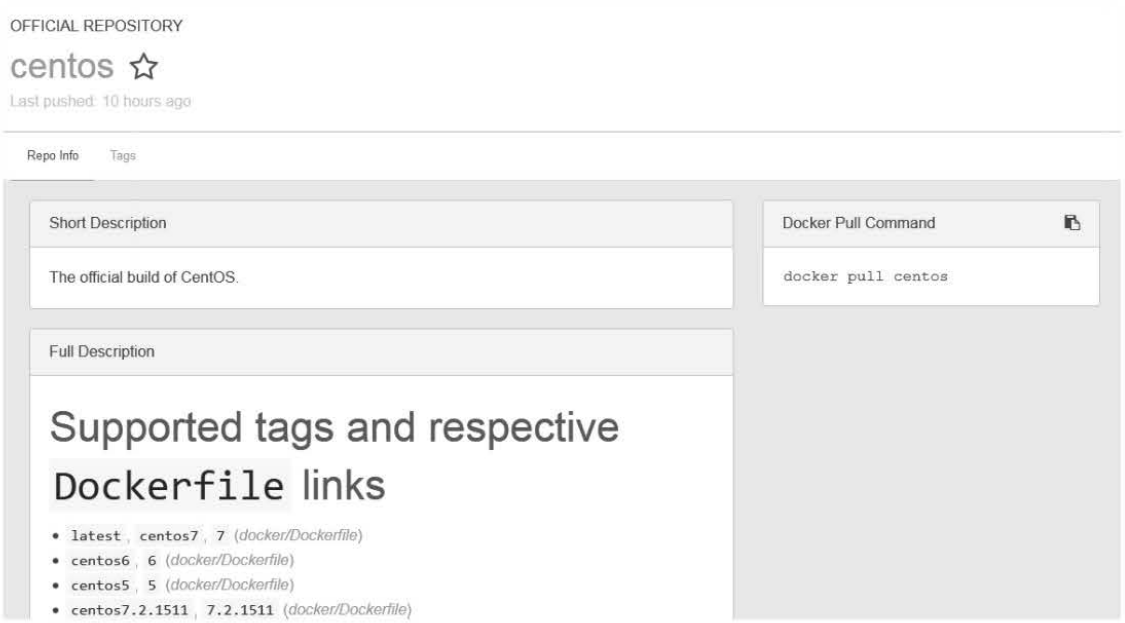

Figure 1.10 — Image de base CentOS

Mais il faut comprendre que le Docker Hub n'est qu'une instanciation publique d'un registry. Il est tout à fait possible à une entreprise d'installer un registry privé[1] pour y gérer ses propres images.

Les autres offres de registry cloud : une féroce compétition

Si le Docker Hub reste la plus connue et la plus populaire des offres de registry cloud (notamment dans le monde open source), les grandes manœuvres ont commencé. Tour à tour, Google[2] et Amazon[3] ont annoncé leurs propres implémentations de registries privés, ce qui concurrence de fait celui du Docker Hub (source principale de revenus pour Docker Inc. à l'heure actuelle).

Comment fonctionne le registry en lien avec le Docker Engine ?

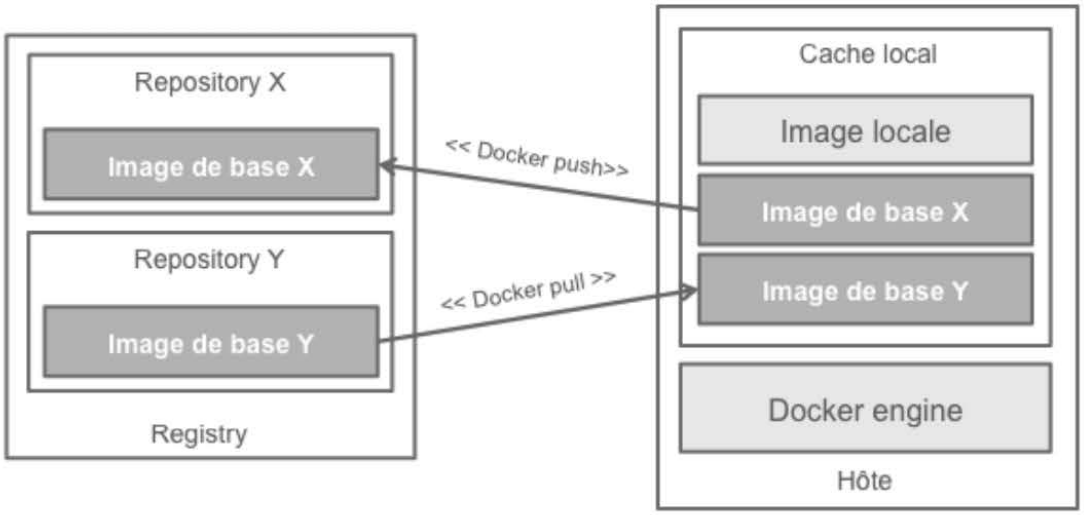

Figure 1.11 — Installation et publication d'images

Le registry va être au choix :

- le lieu de stockage d'images créées sur un hôte (par exemple par un développeur). Le registry sert alors de lieu de publication et d'annuaire. On parle alors de *push* (via la commande `docker push`) ;
- la source d'images à installer sur un ou plusieurs hôtes et notamment, dans le cas du Docker Hub, des images de base. On parle alors de *pull* (via la commande `docker pull`).

Certaines images peuvent aussi rester privées et ne pas être publiées via un registry. Dans ce cas, ces images restent au niveau du cache de l'hôte qui les a créées.

1. Docker propose d'ailleurs un logiciel nommé *Docker Trusted Registry* pour les clients qui souhaitent mettre en place leur propre instance.
2. https://cloud.google.com/container-registry/
3. https://aws.amazon.com/fr/ecr/

La réutilisation des images se produit donc de deux manières :

- à l'échelle d'un hôte, par l'intermédiaire du cache local qui permet d'économiser des téléchargements d'images déjà présentes ;
- à l'échelle du registry, en offrant un moyen de partage d'images à destination de plusieurs hôtes.

1.3.4 *Copy on write*, persistance et volumes

Nous savons maintenant qu'un conteneur est construit à partir d'une image au même titre qu'une machine virtuelle physique.

À un certain moment, cette image va être mise en route. Le conteneur va exécuter des processus qui alloueront de la mémoire, consommeront des ressources et vont évidemment écrire des données sur le système de fichiers. La question est donc maintenant de comprendre comment ces données sont produites et sont éventuellement sauvegardées entre deux exécutions d'un conteneur (et partagées entre plusieurs conteneurs).

Le container layer

Intuitivement, on comprend que ces données sont produites dans une couche au-dessus de toutes les autres, qui va être spécifique au conteneur en cours d'exécution. Cette couche se nomme la couche de conteneur (ou *container layer*).

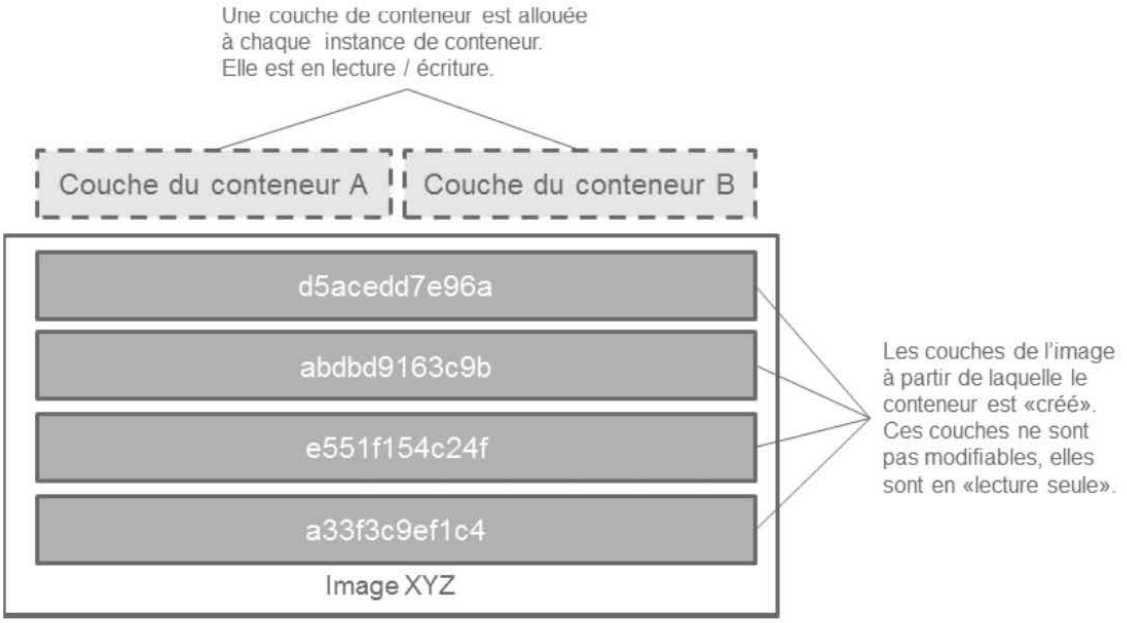

Figure 1.12 — Couches d'images et couches de conteneurs

Cette couche de conteneur est modifiable contrairement aux couches d'images qui ne le sont pas. Dans le cas contraire, faire tourner un conteneur reviendrait à altérer potentiellement l'image sur laquelle il est construit, ce qui n'aurait aucun sens,

puisque le but d'une image est justement de pouvoir créer autant de conteneurs que l'on souhaite, à partir du même moule.

Soit, mais imaginons que nous construisions un conteneur à partir d'une image de base CentOS. Imaginons encore que nous souhaitions modifier la configuration de ce système de base, par exemple /etc/bashrc, qui configure l'environnement du shell UNIX bash. Si l'image de base n'est pas modifiable, comment allons-nous pouvoir procéder ?

Le concept de copy on write

C'est là qu'entre en jeu un concept très puissant, nommé *copy on write* (souvent abrégé par COW) ou copie sur écriture.

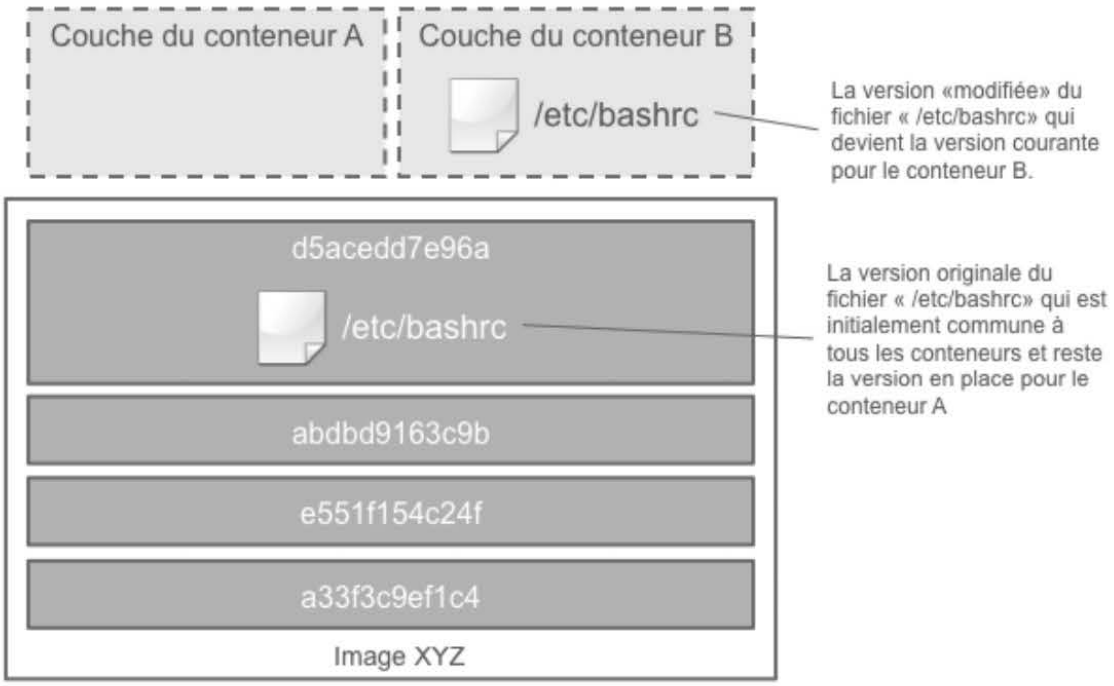

Figure 1.13 — Concept de *copy on write* (COW)

En réalité, l'image n'est pas altérée, mais lorsqu'une modification est demandée sur un élément d'une image, le fichier original est copié et la nouvelle version surcharge la version originale. C'est le principe du *copy on write*. Tant que les accès ne se font qu'en lecture, c'est la version initiale (issue de l'image) qui est employée. Le COW n'est appliqué que lors d'une modification, ce qui optimise de fait la taille de la couche du conteneur qui ne comprend que le différentiel par rapport à l'image.

Le principe est le même pour les effacements de fichiers de l'image. La couche conteneur enregistre cette modification et masque le fichier initial.

En réalité, l'explication ci-dessus est quelque peu simplifiée. La manière de gérer l'union des couches et le *copy on write* dépend du pilote de stockage (*storage driver*) associé à Docker. Certains travaillent au niveau du fichier, d'autres au niveau de blocs plus petits. Mais le principe reste identique.

Différents pilotes de stockage

Docker peut fonctionner avec différentes implémentations de gestion de stockage. Il en existe plusieurs qui sont associées à des caractéristiques différentes ; citons, par exemple, *devicemapper*, *AUFS*, *OverlayFS*, etc. En général, Docker est livré avec un driver par défaut adapté au système d'exploitation (par exemple, *devicemapper* pour les systèmes à base de Fedora, comme RedHat ou CentOS).

Persistance et volumes

Que se passe-t-il quand un conteneur s'arrête ?

La couche de conteneur est tout simplement perdue. Si cette couche comprend des données à conserver entre deux lancements de conteneurs ou bien à partager entre plusieurs conteneurs, il faut utiliser un volume (ou *data volume*).

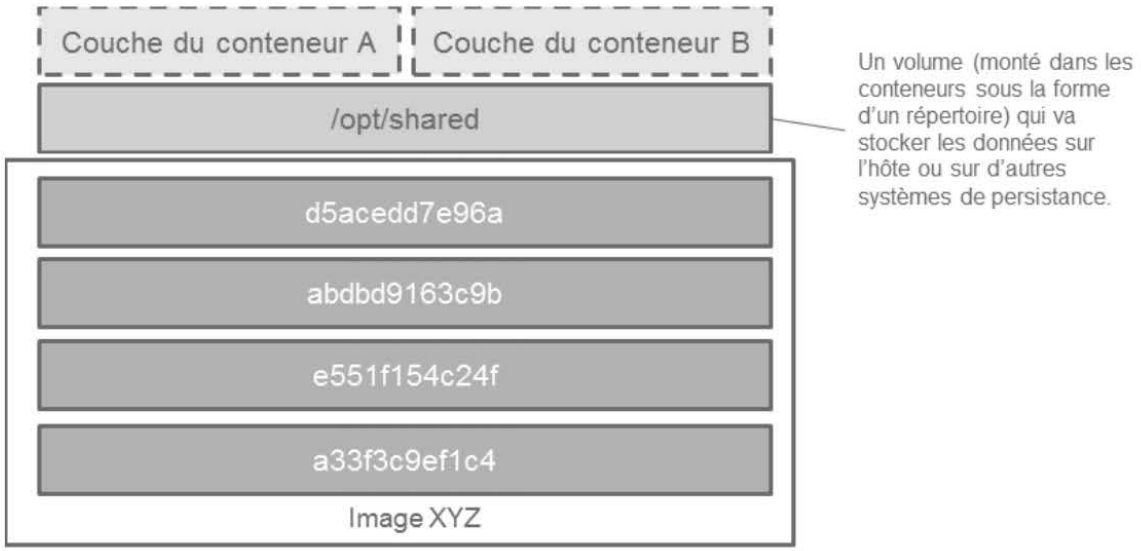

Figure 1.14 — Volumes

Les volumes sont des espaces de données qui court-circuitent le système d'*union file* que nous avons évoqué précédemment. Ils sont modifiés directement et les données qui y sont inscrites survivent à l'arrêt du conteneur. Ce sont des systèmes de fichiers montés dans le conteneur qui, selon les implémentations, reposent :

- sur un espace disque de l'hôte géré par Docker (stocké sous /var/lib/docker/volumes/ mais pas modifiables directement) ;
- sur un répertoire de l'hôte directement monté dans le conteneur et qui permet de créer un pont entre l'hôte et le conteneur ;

- sur un large panel de systèmes tiers qui imposent l'usage d'une extension (*volume plugin*) particulière[1].

Les volumes sont un concept important dans le domaine des conteneurs Docker. Nous les aborderons en détail au cours de nos différents exemples pratiques dans la suite de cet ouvrage.

1.4 LES OUTILS DE L'ÉCOSYSTEME DES CONTENEURS : DOCKER ET LES AUTRES

Nous avons vu précédemment ce qu'était un conteneur, comment il était structuré (du moins dans le cas de Docker) et comment il était distribué.

Nous allons maintenant fournir une vue générale des outils qui gravitent autour de la technologie des conteneurs. Nous souhaitons ici montrer les logiciels qui permettent d'exécuter des conteneurs sur une machine ou sur plusieurs machines en réseau.

Nous ne nous limiterons pas au cas des outils de la société Docker Inc., mais nous présenterons aussi d'autres acteurs. L'objectif pour le lecteur est d'arriver à se repérer dans ce foisonnement de produits et d'en comprendre les positionnements respectifs.

1.4.1 Les moteurs

Créer ou exécuter un conteneur implique de coordonner différents éléments du système d'exploitation (comme CGroups ou Namespaces). Comme nous l'avons vu, ces différentes briques de base sont open source et communes aux implémentations de conteneurs Linux. Ce n'est donc pas à ce niveau qu'il faut chercher la différenciation entre les différents moteurs de conteneurs.

Le moteur d'exécution de conteneur (par exemple, Docker Engine dans le cas de Docker) offre en fait d'autres fonctionnalités de plus haut niveau, au-dessus de ces briques de base comme :

- une ligne de commande ;
- des API ;
- la capacité de nommer, découvrir et gérer le cycle de vie des conteneurs ;
- la capacité à créer des conteneurs à partir d'images ou de modèles ;
- etc.

1. Citons notamment Flocker (https://clusterhq.com/flocker/introduction/) qui permet de créer des volumes en faisant abstraction de leur localisation physique.

Docker et Docker Engine

Comme nous l'avons déjà exposé dans l'avant-propos de ce livre, Docker est à la fois le nom d'une entreprise et celui d'un écosystème logiciel constitué de plusieurs outils (fournis par Docker Inc. ou aussi par d'autres entreprises). Le Docker Engine est le logiciel qui gère les conteneurs sur une machine : le moteur. Le lecteur nous pardonnera d'utiliser souvent le terme « Docker » quand nous parlerons du « Docker Engine » dans la suite de cet ouvrage. Quand nous évoquerons d'autres éléments de la suite Docker (comme la solution de clustering Swarm) nous serons plus spécifiques dans leur dénomination.

S'il existe plusieurs moteurs ou interfaces de gestion de conteneurs, il ne fait aucun doute que celle de Docker est de loin la plus conviviale et la plus aboutie (et de ce fait la plus populaire).

Citons néanmoins les principales offres alternatives :

- **LXC**[1] (Linux Containers), dont la première version date d'août 2008, est historiquement la première implémentation de la virtualisation de l'OS au niveau du noyau Linux. Docker s'appuyait initialement sur LXC avant de réécrire sa propre bibliothèque de bas niveau, la fameuse libcontainer.
- **Rkt**[2] est un logiciel open source édité par le projet/entreprise CoreOS (qui produit d'autres composants tels etcd) qui implémente la spécification App Container (appc)[3], l'une des tentatives de standardiser le monde des conteneurs. Rtk adopte une approche similaire à runC, dans le sens où il n'impose pas de recourir à un démon (à l'inverse de Docker, comme nous le verrons par la suite) pour contrôler centralement le cycle de vie du conteneur.
- **RunC**[4] est une implémentation open source supportée par un large nombre d'acteurs de l'industrie dont Docker. RunC s'appuie sur la spécification Open Container Initiative[5]. RunC doit plutôt être considéré comme un composant de bas niveau destiné à s'intégrer dans des moteurs de plus haut niveau.
- **OpenVZ**[6], qui représente le moteur open source de la solution de gestion d'infrastructure virtualisée Virtuozzo[7]. OpenVZ ne s'appuie pas sur un démon comme Rkt et positionne plutôt la virtualisation à base de conteneurs comme une alternative plus légère à la virtualisation matérielle. Cette position a changé progressivement avec le support de Docker et de ses images.

1. https://linuxcontainers.org/
2. https://coreos.com/rkt/
3. https://github.com/appc/spec/
4. https://runc.io/
5. https://www.opencontainers.org/
6. https://openvz.org/Main_Page
7. http://www.virtuozzo.com/

Nous noterons que toutes ces implémentations (et il en existe d'autres plus confidentielles) sont compatibles avec le format d'image Docker.

1.4.2 Les OS spécialisés

Les conteneurs réutilisent le noyau du système d'exploitation de l'hôte sur lequel ils tournent. En théorie, dans le monde Linux, toute distribution récente est capable de supporter l'exécution de conteneurs. Certains éditeurs ou communautés open source ont cependant commencé à travailler sur des OS spécialisés dans l'exécution de conteneurs. L'objectif est de produire des OS simplifiés, plus performants et moins exposés aux risques de sécurité ou d'instabilité.

Citons notamment :

- **Boot2Docker** est la distribution spécialisée par défaut que Docker installe en local lorsqu'on utilise Docker machine (que nous aborderons dans le chapitre 7) ou le Docker ToolBox. Il s'agit d'une distribution Linux très compacte, basée sur le Tiny Core Linux. Notons que pour les installations distantes (c'est-à-dire pour les serveurs) Docker propose d'autres distributions Linux et notamment, par défaut, une distribution Ubuntu.
- **CoreOS**[1], qui est aussi l'éditeur de Rkt précédemment cité (et d'autres outils d'infrastructure que nous allons présenter par la suite).
- **Atomic**[2], le sous-projet de Fedora (la distribution de base qui sert à RedHat ou CentOS). Atomic tente aussi d'imposer la spécification Nulecule comme une norme de composition et de conditionnement de conteneurs (c'est-à-dire une sorte de Docker compose standardisé). Nous verrons en pratique comment installer une instance *atomic* dans le chapitre 4.
- **Windows Nano Server**[3] est un système Windows réduit à sa plus simple expression et dédié aux applications cloud. Il n'est pas uniquement dédié à l'exécution de conteneurs (qui s'appuiera sur le moteur Windows Container Host), mais relève de cette même démarche de spécialisation.

1.4.3 Les outils d'orchestration : composition et clustering

Bien qu'il soit possible d'exécuter sur un même conteneur plusieurs processus, les bonnes pratiques en matière d'architecture consistent plutôt à distribuer ces processus dans plusieurs conteneurs. Une application sera donc presque nécessairement répartie sur plusieurs conteneurs. Comme ceux-ci vont dépendre les uns des autres, il est nécessaire de les associer.

Cette association que l'on nomme composition consiste :

1. https://coreos.com/using-coreos/
2. http://www.projectatomic.io/
3. https://en.wikipedia.org/wiki/Windows_Server_2016

- à définir un ordre de démarrage (directement ou indirectement) ;
- à définir les systèmes de fichiers persistants (les volumes) ;
- à définir les liens réseau que les conteneurs vont entretenir entre eux (ou du moins à assurer une communication transparente entre les processus).

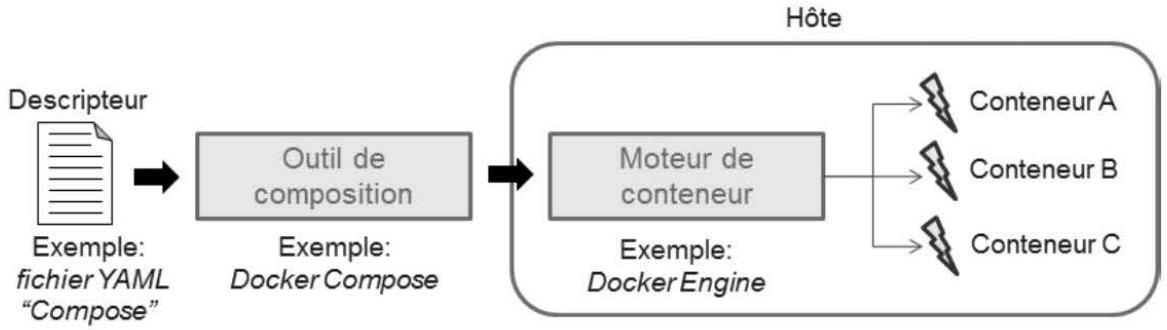

Figure 1.15 — Composition de conteneurs

Il est bien évidemment possible de démarrer les conteneurs à la main à l'aide d'un script exécutant des séries de `docker run`.

Mais cette solution peut rapidement devenir complexe. En effet, il faut aussi tenir compte de l'éventuel besoin de distribuer la charge sur plusieurs nœuds du réseau et plusieurs machines. Il faut donc que la composition puisse s'effectuer en collaboration avec des outils de clustering réseau qui vont abolir les frontières entre les différents hôtes. On parle alors d'orchestration.

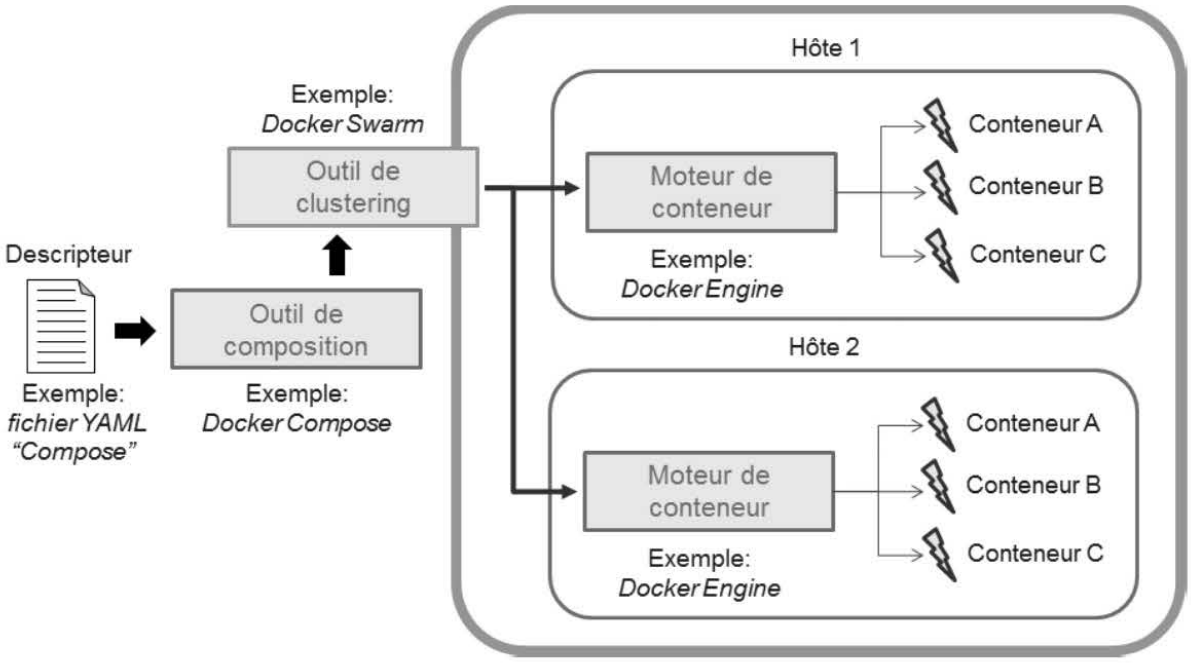

Figure 1.16 — Clustering + composition = orchestration

La composition et le clustering sont donc deux fonctions qui sont la plupart du temps liées au sein de solutions de plus haut niveau. Certains parlent de système d'exploitation de centre de calcul ou DCOS (Data Center Operating System).

Ces produits logiciels complexes sont aujourd'hui, même s'ils sont pour l'essentiel open source, l'objet d'une compétition féroce. Nous allons décrire les principaux acteurs de cette lutte pour la prééminence dans le prochain chapitre en abordant la notion de CaaS : *Container As A Service*.

En résumé

Dans ce chapitre, nous avons appris ce qu'est un conteneur, sur quelles briques fondamentales les conteneurs Linux sont basés, comment ils sont structurés dans le cas de Docker et comment ils sont distribués.

Nous avons ensuite abordé les différents outils assurant leur exécution : moteurs de conteneurs, systèmes d'exploitation spécialisés, outils de composition et d'orchestration.

Il est maintenant temps d'étudier l'apport de ces outils aux techniques d'automatisation de l'exploitation d'une infrastructure d'applications. Notre prochain chapitre va montrer comment ces briques de base s'agrègent aujourd'hui dans divers types de solutions industrielles.

Nous expliquerons aussi comment ces solutions à base de conteneurs se positionnent par rapport à des solutions de gestion de configuration comme Puppet, Ansible ou Chief.

2

Conteneurs et infrastructures

L'automatisation de l'exploitation

L'objectif de ce chapitre est de montrer comment les conteneurs se positionnent comme une nouvelle opportunité pour automatiser le déploiement et l'exécution des applications dans les entreprises. Nous allons aborder la notion de CaaS (*Container as a Service*) mais aussi tenter de positionner la technologie des conteneurs par rapport à d'autres solutions d'automatisation de la gestion de configuration, comme Puppet ou Ansible.

2.1 AUTOMATISER LA GESTION DE L'INFRASTRUCTURE : DU IAAS AU CAAS

La virtualisation de l'infrastructure n'est pas nouvelle et le *cloud computing* fait aujourd'hui partie du paysage standard de tout système d'information. Nous allons, dans un premier temps, expliquer ce que les conteneurs apportent par rapport aux solutions de virtualisations d'infrastructures actuelles ou IaaS (pour *Infrastructure as a Service*).

2.1.1 Virtualiser l'infrastructure

L'infrastructure est un service !

En pratique, cette affirmation est déjà partiellement vraie. La virtualisation matérielle a permis l'émergence d'offres *cloud* privées ou publiques qui ont abstrait la notion de centre de calcul à un tel niveau que, pour certains usages, la nature réelle de l'hôte et sa localisation géographique sont devenues secondaires.

Dans le cas des infrastructures privées, diverses solutions commerciales (vCenter de VMWare) ou open source (OpenStack) automatisent la gestion de cette puissance de calcul (mais aussi de stockage, de communication réseau, etc.) virtualisée au-dessus d'une infrastructure physique de nature variable.

> Pour ce qui est des infrastructures publiques (AmazonEC2, Google, Windows Azure, Heroku, Digital Ocean, etc.) le principe est identique, si ce n'est que les solutions sont propriétaires, tout en étant ouvertes via des services web documentés.

Concrètement, les solutions IaaS offrent aux applications conditionnées sous la forme d'images de machines virtuelles (par exemple, les AMI pour Amazon) des services d'infrastructure, comme :

- **de la puissance de calcul**, via un environnement d'exécution pour une machine virtuelle basée sur un hyperviseur : VMWare, Xen, KVM, etc. ;
- **du stockage** qui peut se présenter sous différentes formes selon les niveaux de performance et de résilience requis ;
- **des services réseau** : DHCP, pare-feu, fail-over, etc. ;
- **des services de sécurité** : authentification, gestion des mots de passe et autres secrets.

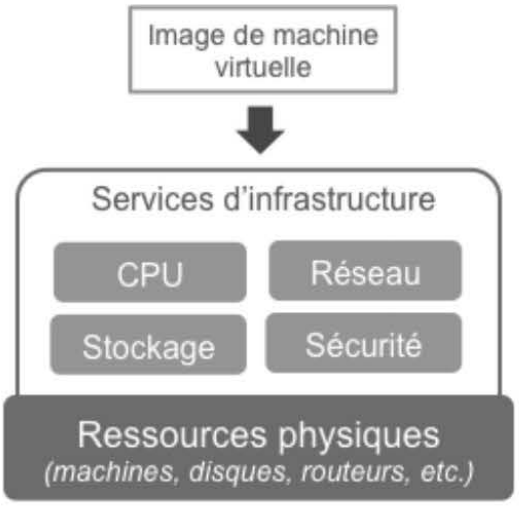

Figure 2.1 — Services d'infrastructure d'une solution IaaS

Pour peu qu'une application soit conditionnée sous la forme d'une ou plusieurs machines virtuelles, elle bénéficie des ressources dont elle a besoin sans avoir à se soucier de la gestion de la qualité de service liée à l'infrastructure.

En effet, les solutions IaaS vont détecter les failles matérielles (voire les anticiper) et s'assurer que l'environnement d'exécution est toujours actif. Si une machine physique casse, les machines virtuelles qu'elle exécute à un instant donné peuvent être transférées de manière quasi transparente sur un autre hôte.

Mais alors que vont apporter les conteneurs aux solutions IaaS ?

2.1.2 Le conteneur n'est-il qu'un nouveau niveau de virtualisation ?

Les conteneurs, et Docker en particulier, sont donc au centre d'une nouvelle gamme de solutions que certains nomment CaaS pour *Container as a Service* qui fait écho à l'IaaS que nous avons évoqué précédemment.

En quoi l'utilisation de conteneurs change-t-elle la donne ?

Pratiquement, les conteneurs ne changent pas les solutions IaaS qui conservent leurs avantages propres, mais ils peuvent s'appuyer sur elles pour les étendre et offrir de nouvelles opportunités.

Nous avons vu dans le chapitre 1 ce que pouvait apporter la virtualisation à base de conteneurs par rapport à la virtualisation matérielle. Il ne s'agit pas d'une alternative technologique, mais bien d'une autre manière de concevoir le lien entre le développeur et l'exploitant.

Pour comprendre, analysons le processus de préparation d'environnement, puis le processus de déploiement d'une application dans un environnement IaaS.

Préparation de l'environnement en mode IaaS

La phase d'installation de l'infrastructure consiste pour le développeur à spécifier ses besoins en matière d'environnement d'exécution (ou *runtime*). Il s'agit par exemple de fournir :

- la version de Java utilisée ;
- le serveur d'application, la base de données et les autres middlewares requis ;
- d'éventuels besoins de bibliothèques additionnelles de niveau OS ;
- certaines caractéristiques non fonctionnelles de l'application (besoin en stockage, niveau de performance, besoin en backup, niveau de résilience ou de sécurité, etc.).

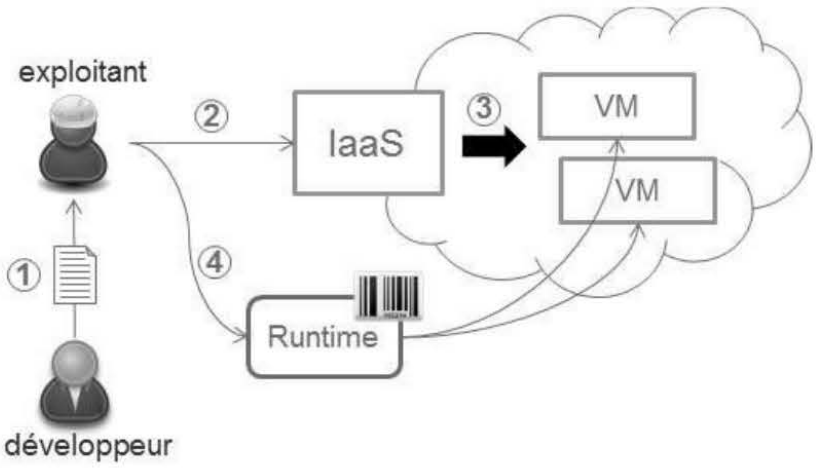

Figure 2.2 — Préparation de l'environnement

Cette phase impose un dialogue poussé entre le développeur et l'exploitant. Il faut d'une part que le développeur spécifie ses besoins, mais aussi que l'exploitant les interprète par rapport à la capacité disponible, aux règles de sécurité ou d'architectures prédéfinies, etc.

Certains types d'organisation, comme le modèle DevOps, visent à optimiser ce dialogue (qui est en fait récurrent, puisque l'architecture évolue au fil du temps), mais cette phase reste sensible et complexe à réussir. Il est clair qu'une part significative des incidents de production de toute application est souvent le résultat de problèmes d'alignement entre la vision du développeur et celle de l'exploitant.

Déploiement d'applications en mode IaaS

Entre chaque phase d'adaptation de l'environnement, on va trouver une série de déploiements, plus ou moins fréquents, de paquets logiciels correspondant à une nouvelle version de l'application.

> Techniquement, le paquet logiciel peut être un zip, un tar, un msi (dans le monde Microsoft), un rpm (dans le monde Linux Fedora), le choix étant vaste. À ce ou ces paquets sont souvent associés des scripts de déploiement plus ou moins normalisés qui sont souvent écrits par l'exploitant (en collaboration plus ou moins étroite avec l'équipe de développement).

Il s'agit pour le développeur de produire son code, de le conditionner (on parle de *build* et de *packaging*) sous une forme qui puisse être transmise à l'exploitant. En pratique, cette phase peut être évidemment automatisée (c'est le cas notamment lorsque l'équipe fonctionne en déploiement continu), mais il ne reste pas moins que les caractéristiques du paquet logiciel doivent rester cohérentes avec le processus de déploiement et l'architecture de la solution.

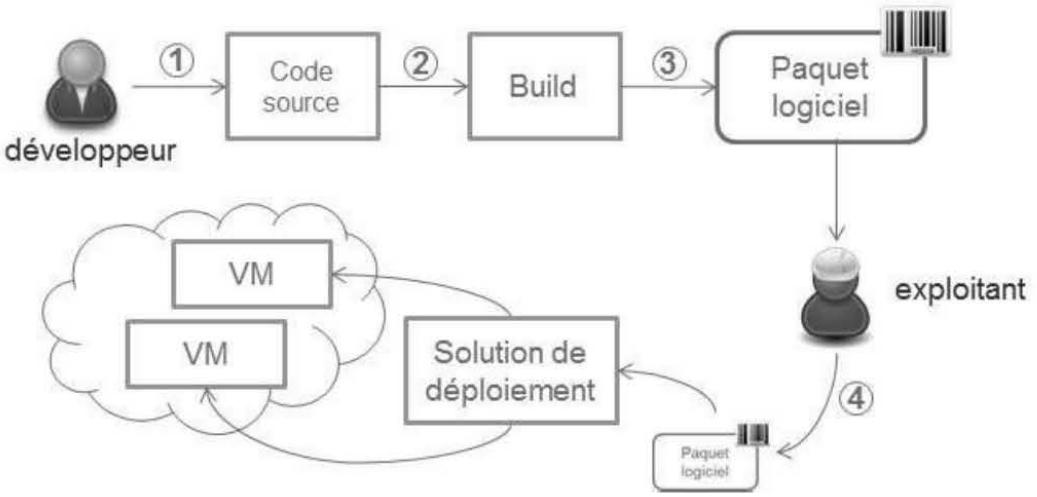

Figure 2.3 — Déploiement d'applications

L'exploitant doit se contenter de comprendre le fonctionnement de l'application (pour préparer l'environnement), mais doit aussi comprendre comment elle est conditionnée pour la déployer.

La gestion de la solution de déploiement est en effet bien souvent de la responsabilité de l'exploitant.

Certaines solutions, comme Ansible ou Puppet, permettent d'automatiser les deux phases (préparation de l'environnement et déploiement), mais pour des raisons évidentes, l'environnement n'est généralement pas reconstruit à chaque déploiement. Tout d'abord parce que cette phase peut être lourde, longue et coûteuse ; ensuite parce que l'application ne peut pas nécessairement s'accommoder d'une reconstruction complète à chaque déploiement (disponibilité).

Attention, nous ne disons pas qu'il est impossible d'adopter une approche visant à la reconstruction systématique de l'environnement d'exécution. L'usage de templates d'application et d'outils automatisés de gestion de configuration (tel que Vagrant parmi d'autres déjà cités) rend possible ce type de modèle avec des machines virtuelles classiques. Mais reconnaissons que ce type de cas est rare dans un environnement de production. La plupart du temps, il y a une différence claire entre la construction de l'environnement (ou sa modification) et le déploiement d'applications.

Les limites du modèle IaaS

La virtualisation de l'infrastructure proposée par le modèle IaaS constitue un progrès énorme dans le commissionnement et la préparation d'un environnement applicatif.

L'IaaS va faciliter le provisionnement de l'infrastructure et sa gestion :

- en l'automatisant : tout se passe par configuration, pour peu que la capacité disponible soit suffisante ;
- en offrant une manière simple de gérer la montée en charge, pour peu que l'application le permette (par la possibilité de redimensionner l'environnement en ajoutant de la CPU, de la RAM ou du stockage par exemple).

À l'inverse, l'IaaS n'apporte aucune solution pour traiter deux questions fondamentales :

- le déploiement des applications ;
- la description formelle des caractéristiques d'évolutivité et de résilience de l'application par le développeur, leur transmission (sans perte d'information) à l'exploitant et leur exécution en production.

2.1.3 L'apport du modèle de déploiement CaaS

Les approches CaaS visent à aborder en premier lieu la problématique du déploiement d'applications. La problématique de la virtualisation des ressources matérielles est en fait secondaire.

Déployer une infrastructure à chaque version d'application

Dans un modèle CaaS, les phases de préparation de l'environnement et de déploiement semblent indistinctes. La raison tient tout d'abord au fait qu'une image de conteneur Docker inclut l'environnement d'exécution ou *runtime* en même temps que le code de l'application. Le flux 4 de la figure 2.4 est donc inexistant. L'infrastructure est totalement banalisée et se limite à un moteur de conteneurs.

Ensuite, dans le cas d'un développement sur la base de Docker, le paquet logiciel, grâce au principe d'image évoqué dans le précédent chapitre, correspond très exactement à ce qui va être déployé par l'exploitant.

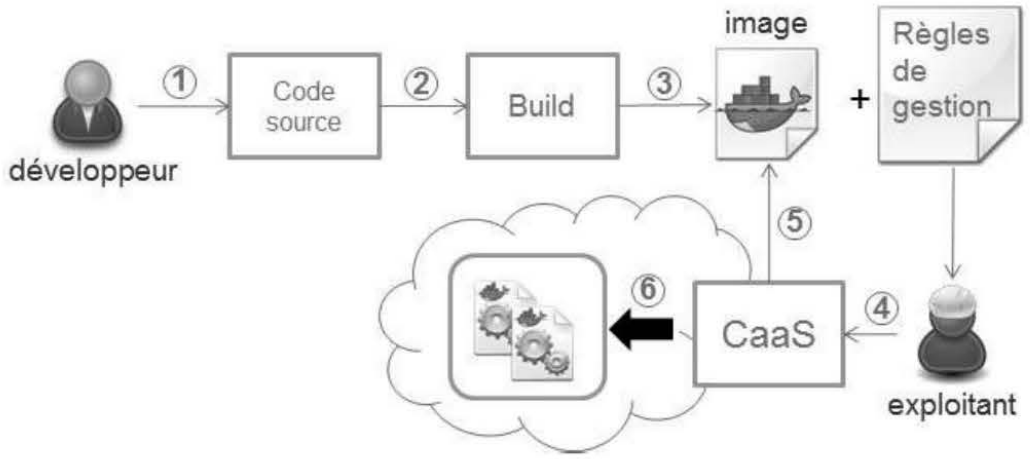

Figure 2.4 — Construction et déploiement d'une application en mode CaaS

Il n'y a pas de retraitement de l'image par une solution tierce.

Comme un conteneur multimodal physique, le conteneur logiciel est scellé. L'exploitant joue le rôle de transporteur. Il choisit la manière dont le conteneur va être exécuté sans l'altérer.

Enfin, les solutions CaaS associent au déploiement des règles de gestion. Celles-ci peuvent prendre la forme d'un fichier YAML ou JSON, selon les solutions. Celui-ci peut être produit par l'équipe de développement ou être élaboré en collaboration avec l'exploitant. Cette configuration va spécifier d'une manière totalement formelle les caractéristiques non-fonctionnelles de l'architecture :

- nombre d'instances de chaque type de conteneur ;
- liens entre les conteneurs ;
- règles de montée en charge ;
- règles de répartition de charge ;
- volumes persistants ;
- etc.

Il ne s'agit ni plus ni moins que de programmer l'infrastructure. Vous trouverez ci-dessous un exemple de ce type de fichier de configuration (de la solution Kubernetes) :

```
apiVersion: extensions/v1
kind: Deployment
(...)
spec:
  replicas: 4
  selector:
    matchLabels:
      run: hello-node
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: hello-node
    spec:
      containers:
      - image: gcr.io/PROJECT_ID/hello-node:v1
        imagePullPolicy: IfNotPresent
        name: hello-node
        ports:
        - containerPort: 8080
          protocol: TCP
        resources: {}
        terminationMessagePath: /dev/termination-log
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      securityContext: {}
      terminationGracePeriodSeconds: 30
```

Sans entrer dans le détail, on note que le fichier va spécifier des informations concernant :

- les règles de déploiement (par exemple `rollingUpdate` qui explique que les différents nœuds de l'application doivent être mis à jour les uns après les autres) ;
- les caractéristiques de l'architecture en termes de nombre d'instances (`replicas`) et de politique de redémarrage (`restartPolicy`).

Nous allons expliquer le mode de fonctionnement dynamique des solutions CaaS sur la base de ce type de règles de gestion dans la section suivante.

À première vue, cette programmabilité complète de l'infrastructure semble séduisante. Elle a cependant un coût : l'application doit être conçue dès son origine pour fonctionner dans un mode conteneur ou micro-services. Ce sera l'objet des chapitres suivants d'expliquer la manière de concevoir une image Docker et comment celle-ci peut être assemblée à d'autres images pour produire une application *container-ready*.

Conclusion sur le lien entre CaaS et IaaS

Le modèle CaaS présuppose que des machines sont mises à disposition avec la puissance de calcul appropriée et une solution CaaS spécialisée dont nous verrons plusieurs exemples dans la suite de ce chapitre.

L'apport de l'IaaS dans la mise en place d'une abstraction des caractéristiques physiques de l'infrastructure offre un intérêt certain. Néanmoins, ce lien entre CaaS et IaaS n'est pas obligatoire. Il est parfaitement possible d'adopter une approche CaaS sur une infrastructure physique (sans couche de virtualisation additionnelle). Le choix de l'une ou de l'autre option dépend avant tout de choix stratégiques d'entreprise plutôt que d'une nécessité technique.

2.1.4 L'architecture générique d'une solution CaaS

Les solutions CaaS ont un certain nombre de caractéristiques communes qu'il est bon de connaître :

- elles incluent presque toutes Docker (en dehors de celle de CoreOS qui lui préfère Rkt) en tant que moteur de conteneurs ;
- elles fournissent des fonctions de clustering et d'orchestration (décrites dans le chapitre précédent et qui seront abordées en pratique dans la suite de cet ouvrage) ;
- elles s'appuient sur un principe de gestion automatisée de l'infrastructure que nous qualifierons d'homéostatique.

Une régulation homéostatique

L'hémostasie est un concept décrit par le grand biologiste français Claude Bernard au XIX^{e} siècle et qui se définit de la manière suivante :

> L'homéostasie est un phénomène par lequel un facteur clé (par exemple, la température) est maintenu autour d'une valeur bénéfique pour le système considéré, grâce à un processus de régulation.

Les exemples courants sont :

- la régulation de la température corporelle ;
- la régulation du taux de glucose dans le sang ;
- etc.

Par processus de régulation, on entend la possibilité de capter l'état du système et de conditionner la réaction, l'intervention correctrice suite à un déséquilibre, à ce *feedback*. Certains auront reconnu des notions qui sont aussi à la base de la cybernétique.

Les outils CaaS s'appuient sur ce principe dans le sens où leur objectif n'est pas de simplement fournir une interface de commande pilotée par un administrateur, mais plutôt de maintenir un état stable par l'intermédiaire de processus automatiques.

Toutes les solutions CaaS ont ainsi des composants en commun. Ils peuvent se nommer de manière diverse, mais ont en réalité le même rôle :

- surveiller l'état du système ;
- détecter les écarts par rapport aux règles de gestion programmées ;
- prendre des mesures pour ramener le système à l'équilibre ;
- prévenir l'exploitant en cas d'échec.

La gestion automatisée de l'infrastructure

Le système de régulation s'appuie généralement sur :

- un contrôleur qui va analyser l'état de l'infrastructure par l'intermédiaire des données remontées par des agents installés sur les différents nœuds de l'architecture et, en fonction des règles préprogrammées, donner des ordres aux agents ;
- des agents qui collectent des informations sur l'état des nœuds sur lesquels ils sont installés (offrant ainsi le *feedback* nécessaire au contrôleur) et, la plupart du temps, servant de relais au contrôleur pour exécuter des actions correctrices (le plus souvent des commandes docker) ;
- une interface d'entrée de règles de gestion qui se présente généralement sous la forme d'une ligne de commande (CLI) pouvant accepter des commandes individuelles ou des fichiers de description. Cette interface est en général basée sur des services web REST dont la ligne de commande constitue un client particulier.

La figure ci-dessous présente une architecture générique de type CaaS.

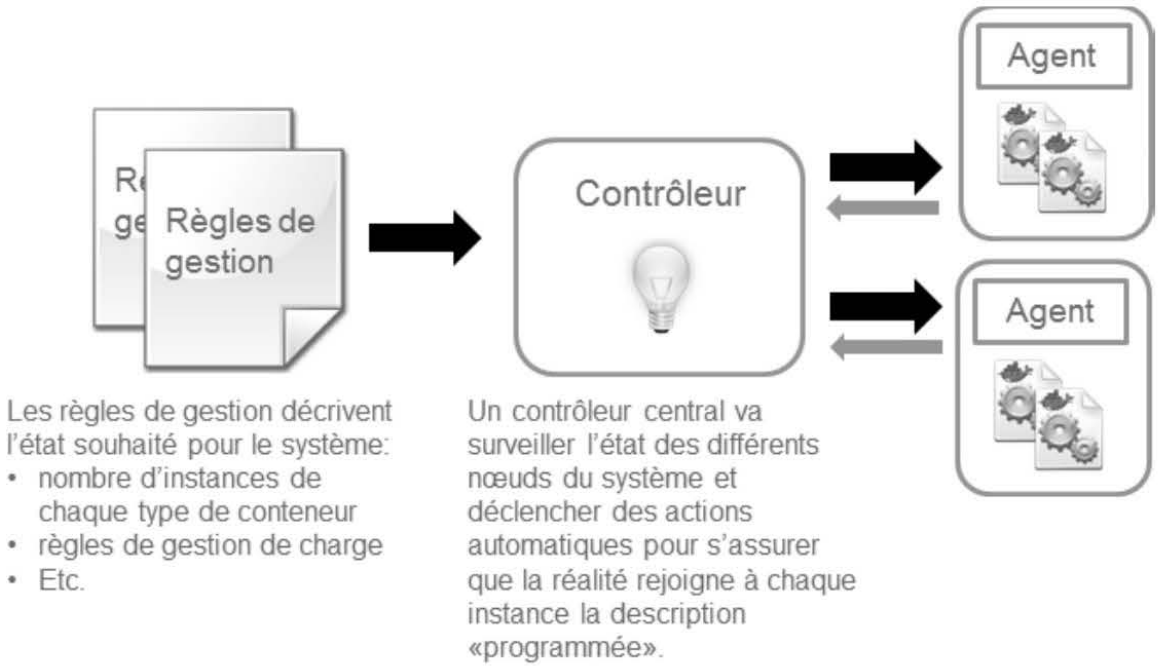

Figure 2.5 — Principe de fonctionnement des CaaS

Évidemment, aux composants ci-dessous, s'ajoutent le moteur de conteneurs (Docker la plupart du temps) et, éventuellement, un registry d'image privé (cf. chapitre 1). La plupart des solutions offrent aussi une interface graphique qui permet une visualisation synthétique de l'état de l'infrastructure.

L'exploitant (souvent sur la base de propositions du développeur) programme les règles de gestion et le système prendra des décisions en fonction de la puissance des ressources disponibles : puissance de calcul, caractéristiques de l'architecture ou de bien d'autres critères formulés sur la base de divers types de métadonnées (par exemple, le mécanisme des labels et filtres chez Kubernetes).

Il est aussi important de comprendre que ces règles de gestion vont persister pendant tout le cycle de l'application. En clair, le système va s'assurer que ces règles restent toujours vraies.

Par exemple, si le configurateur a déclaré vouloir cinq instances d'un certain type de conteneur mais qu'un agent, à un instant donné, ne détecte que quatre instances, le contrôleur, ainsi informé, va automatiquement déclencher une nouvelle création, sans aucune intervention humaine.

> Les règles expriment des exigences, l'état cible souhaité et le système s'assure au travers d'actions déclenchées automatiquement que cet état soit atteint et maintenu autant que possible. D'autres solutions que nous aborderons dans la dernière section, comme Ansible, fonctionnent sur le même principe.

Solutions CaaS et DCOS

On commence à voir émerger le terme de système d'exploitation d'infrastructure (ou DCOS pour *Data Center Operating System*).

Le système d'exploitation (Windows, Linux, Mac OS) offre des services de haut niveau qui permettent aux applications de disposer des ressources d'une machine sans en connaître les caractéristiques techniques intimes. L'objectif de ces solutions est identique, mais à l'échelle d'un centre de calcul. Ils offrent des services de haut niveau pour résoudre la plupart des problèmes de gestion de déploiement et de charge sur la base de modèles standardisés (*patterns*) et éprouvés.

Les conteneurs et les solutions CaaS sont au cœur de ces DCOS mais n'en sont probablement que l'une des composantes. En effet, toutes les applications ne sont pas encore *container-ready*, c'est-à-dire architecturées pour être compatibles avec une approche à base de micro-services. Les DCOS se doivent donc d'être capables de gérer différents types de ressources (et pas uniquement des conteneurs).

Nous présenterons dans ce chapitre la solution Apache Mesos[1], qui constitue un exemple de DCOS, pour expliquer comment les moteurs CaaS s'intègrent dans ce type de produit.

1. http://mesos.apache.org/

2.2 LES SOLUTIONS CAAS

Nous allons maintenant aborder quelques-unes des solutions emblématiques de ce phénomène CaaS. Attention, il est important de bien intégrer le fait que ce nouveau marché est en pleine construction. **Aucune des solutions décrites ne recouvre complètement les autres d'un point de vue des fonctionnalités offertes**. Certaines affichent même la possibilité de collaborer ou de s'intégrer avec d'autres. Mais, ne nous y trompons pas, la compétition est féroce et c'est à une course de vitesse que se livrent les principaux acteurs de ce marché.

2.2.1 Docker Datacenter : vers une suite intégrée

Outre le Docker Engine, la société Docker Inc. édite d'autres logiciels. Certains ont été développés en interne, tandis que d'autres ont été rachetés à d'autres sociétés (comme Compose ou Tutum, par exemple).

Docker travaille activement à la consolidation de ces différents outils en une suite cohérente : Docker Datacenter.

> **Docker Datacenter vs. Tutum**
>
> Actuellement, Docker offre en fait deux suites différentes : une offre *public cloud* construite autour de Tutum et le Docker Hub (registry public), et une offre derrière le firewall que nous allons décrire ci-dessous, et qui s'appuie sur UCP (*Universal Control Plane*). Cette situation est sans aucun doute transitoire. Les fonctionnalités de la version cloud (historiquement apportées par Tutum) vont sans aucun doute être fusionnées avec UCP pour ne faire qu'une seule suite avec deux instanciations : une offre cloud et une offre *on premise* pour les clients souhaitant installer ces logiciels au sein de leur infrastructure privée.

Les briques de base

La suite de Docker est constituée de plusieurs produits qui aujourd'hui sont toujours disponibles indépendamment :

- **Docker Compose**[1] est certainement l'outil de composition de conteneurs le plus connu et la norme *de facto*. Il s'appuie sur un descripteur YAML[2] qui spécifie les conteneurs, leurs dépendances et un certain nombre d'autres paramètres utiles à la description de l'architecture. Docker Compose travaille de concert avec Docker Swarm pour former une solution d'orchestration complète.

1. https://docs.docker.com/compose/overview/
2. http://yaml.org/

En réalité, Docker Compose ne fait aujourd'hui pas partie du produit Docker Datacenter. Néanmoins, son intégration native avec Swarm en fait le choix quasi automatique pour la composition et le déploiement d'architectures multi-conteneurs. Nul ne doute que dans les versions ultérieures de la suite de Docker, Docker Compose (au même titre que Notary ou le Docker Trusted Registry) sera de plus en plus intégré pour progressivement constituer une solution homogène.

- **Docker Swarm**[1] est l'outil de clustering de Docker. En pratique, Docker Swarm transforme plusieurs machines en un seul hôte Docker. L'API proposée est exactement la même que pour un seul et unique hôte. Swarm y ajoute des fonctions de plus haut niveau qui virtualisent le réseau et permettent d'exécuter en ligne de commande des opérations de déploiement ou de gestion de charge.
- **Docker Trusted Registry**[2] permet de stocker les images de conteneurs de manière privée, derrière le firewall du client. La communication avec ce registry est sécurisée par **Docker Notary**[3] qui gère la signature des images et leur vérification.
- **Docker UCP (*Universal Control Plane*)**[4] est le centre de contrôle du DataCenter. Cette interface graphique permet de visualiser ses conteneurs répartis sur plusieurs hôtes et d'effectuer diverses opérations de supervision, d'administration ou de configuration.

L'objectif de Docker est de proposer une suite supportant d'un bout à l'autre le cycle de développement, déploiement et opération.

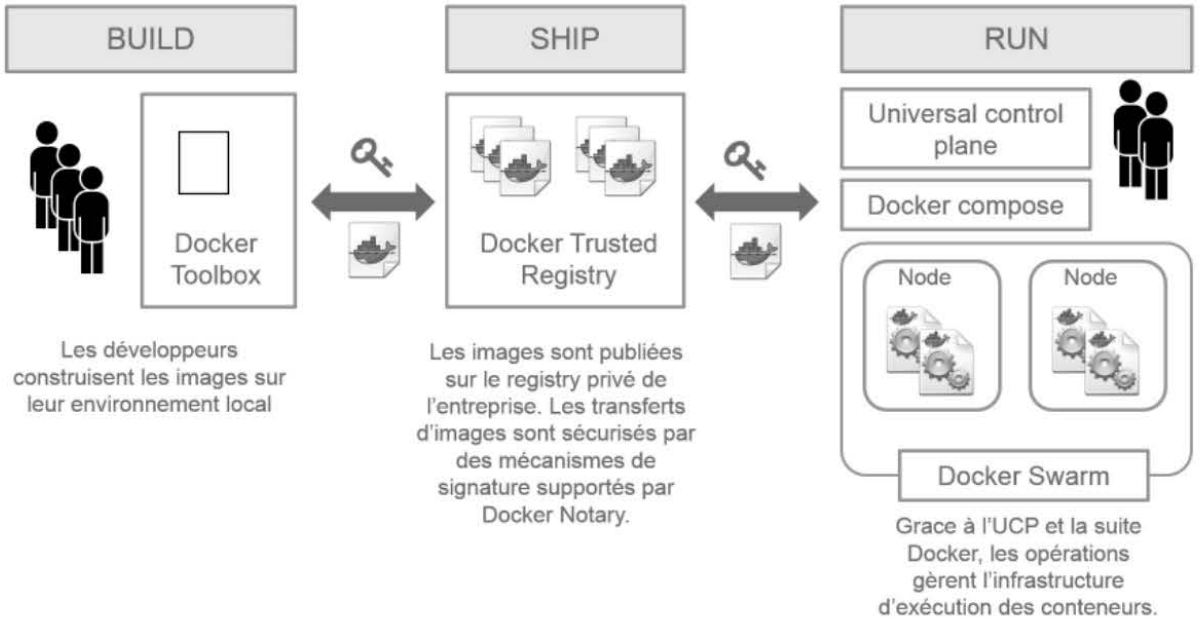

Figure 2.6 — Chaîne de livraison de Docker

1. https://www.docker.com/products/docker-swarm
2. https://docs.docker.com/docker-trusted-registry/
3. https://docs.docker.com/notary/
4. https://docs.docker.com/ucp/

Comment ça marche ?

Chaque hôte de l'architecture exécute un agent Swarm. Cet hôte constitue alors un *Swarm node*. L'agent offre une interface de contrôle (et de surveillance) au point central de pilotage du cluster swarm : le manager.

Le manager est le service qui expose l'API de contrôle du cluster. Il peut être télécommandé en ligne de commande (Docker CLI) ou bien via le Docker UCP (une interface web).

> Notez qu'en production on aura généralement plusieurs *master nodes* (pour des raisons de haute disponibilité). Dans ce cas, plusieurs managers sont installés et la résilience de l'ensemble est assurée par la réplication de l'état de ces managers, par exemple au travers d'etcd.

Une fois l'ensemble installé, Swarm va offrir différents algorithmes de gestion automatisée de conteneurs. Dans la terminologie Swarm, on parle de filtres et de stratégies. Nous aborderons en détail et par la pratique ces concepts dans le chapitre 11, mais nous pouvons ici illustrer notre propos par quelques exemples.

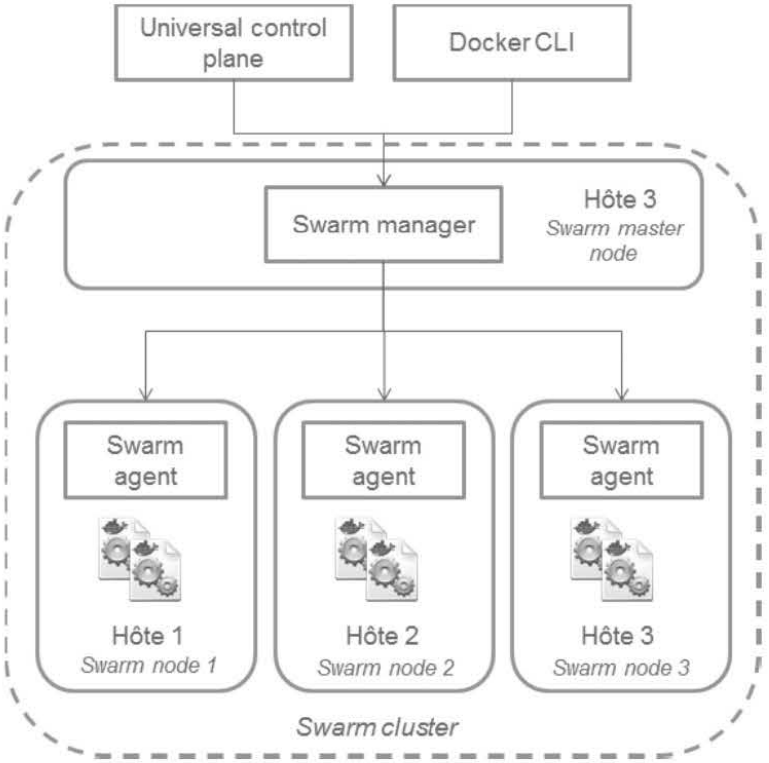

Figure 2.7 — Architecture Swarm

L'une des stratégies de Swarm se nomme *spread*. Celle-ci, lorsqu'un administrateur (via l'UCP ou en ligne de commande) va demander la création d'un certain nombre de conteneurs, va automatiquement s'assurer que ces créations sont uniformément distribuées sur chaque nœud du cluster. D'autres algorithmes tenteront d'optimiser le déploiement en fonction de la charge des machines (CPU et RAM, par exemple).

Il est aussi possible de déclarer des relations d'affinité plus ou moins complexes. Par exemple, vous pouvez vous assurer que les conteneurs base de données ne sont créés que sur les nœuds qui possèdent des disques SSD haute performance.

Grâce à cette bibliothèque de règles (qui va probablement s'enrichir au fil du temps) Swarm offre la possibilité de configurer un large panel d'architectures applicatives et de stratégie de gestion de charge.

UCP, le centre de pilotage central

Le pilotage d'un cluster Swarm se fait par défaut par l'intermédiaire de la ligne de commande Docker. Pour compléter son offre, Docker a néanmoins mis à la disposition de ses clients une interface de gestion graphique : l'UCP (pour *Universal Control Plane*).

L'UCP propose une visualisation synthétique de l'état du cluster de conteneurs Docker. Il permet aussi de définir différents niveaux de profils d'administration associés à des niveaux de droits différents.

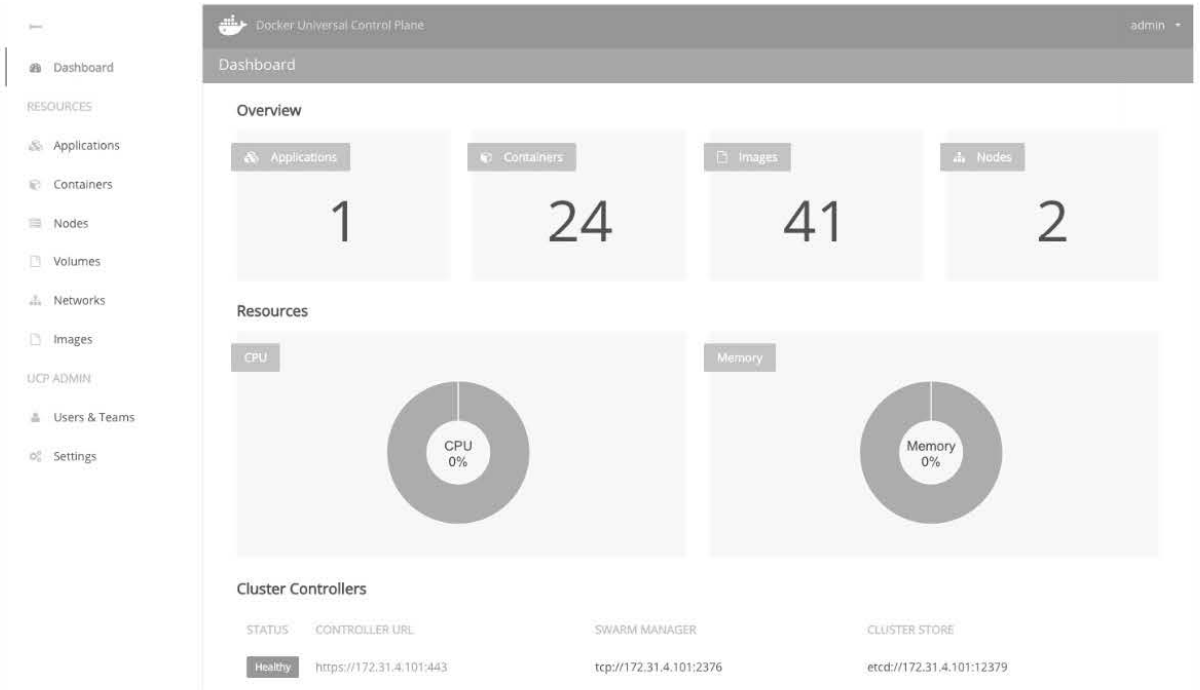

Figure 2.8 — UCP, le cockpit graphique de l'écosystème Docker

Certains auront peut-être entendu parler du projet open source *Shipyard*[1] qui propose aussi une interface d'administration graphique pour les outils Docker. Aujourd'hui, Shipyard offre une alternative vraiment intéressante à UCP pour les petits budgets. Néanmoins, au fil des nouvelles versions, il est probable que Docker enrichisse son offre de fonctionnalités plus avancées et moins ouvertes (UCP n'est en effet pas un projet open source).

La suite CaaS de Docker n'est pas la seule sur le marché. D'autres solutions offrent des fonctionnalités similaires, tout en s'appuyant néanmoins presque systématiquement sur le moteur de Docker.

2.2.2 Kubernetes : l'expérience de Google offerte à la communauté

Kubernetes[2] est un projet open source initié par Google. Contrairement à ce qui est souvent écrit, Google n'utilise pas Kubernetes en interne, mais un autre système nommé Borg. Néanmoins, Kubernetes a clairement bénéficié de l'expérience de Google dans la gestion de centres de calculs et d'applications distribuées hautement disponibles. Notons que Google utilise des conteneurs depuis près de dix ans, ce qui explique pourquoi la plupart des technologies de base des conteneurs (notamment CGroups et Namespaces) ont été élaborées par des ingénieurs de Google.

Kubernetes s'appuie sur le moteur de Docker pour la gestion des conteneurs[3]. Par contre, il se positionne comme une alternative à Docker Swarm et Docker Compose pour la gestion du clustering et de l'orchestration.

Dans les faits, Google ne souhaite probablement pas concurrencer Docker sur le terrain de l'édition de logiciels, mais utilise Kubernetes pour promouvoir son offre Cloud CaaS, nommée *Google container engine*[4]. Ce produit, qui s'intègre dans son offre de cloud public (certainement la plus riche du marché après celle d'Amazon), est justement basé sur Kubernetes, sa ligne de commande et son format de descripteur.

L'architecture de Kubernetes

La figure ci-dessous présente une architecture synthétique de Kubernetes. On y distingue notamment les deux composantes habituelles des CaaS :

- un contrôleur central, chef d'orchestre du système, ici nommé *kubernetes control plane* qui, comme pour Swarm, peut stocker sa configuration dans un cluster etcd ;

1. https://shipyard-project.com/
2. http://kubernetes.io/
3. Kubernetes a aussi pour objectif de s'ouvrir à d'autres moteurs, notamment RKT, le moteur de conteneurs de CoreOS.
4. https://cloud.google.com/container-engine/

- un ou plusieurs nœuds (ou *node*) qui hébergent les agents Kubernetes, que l'on nomme également *kubelets*, et évidemment, les conteneurs organisés en *pods*[1].

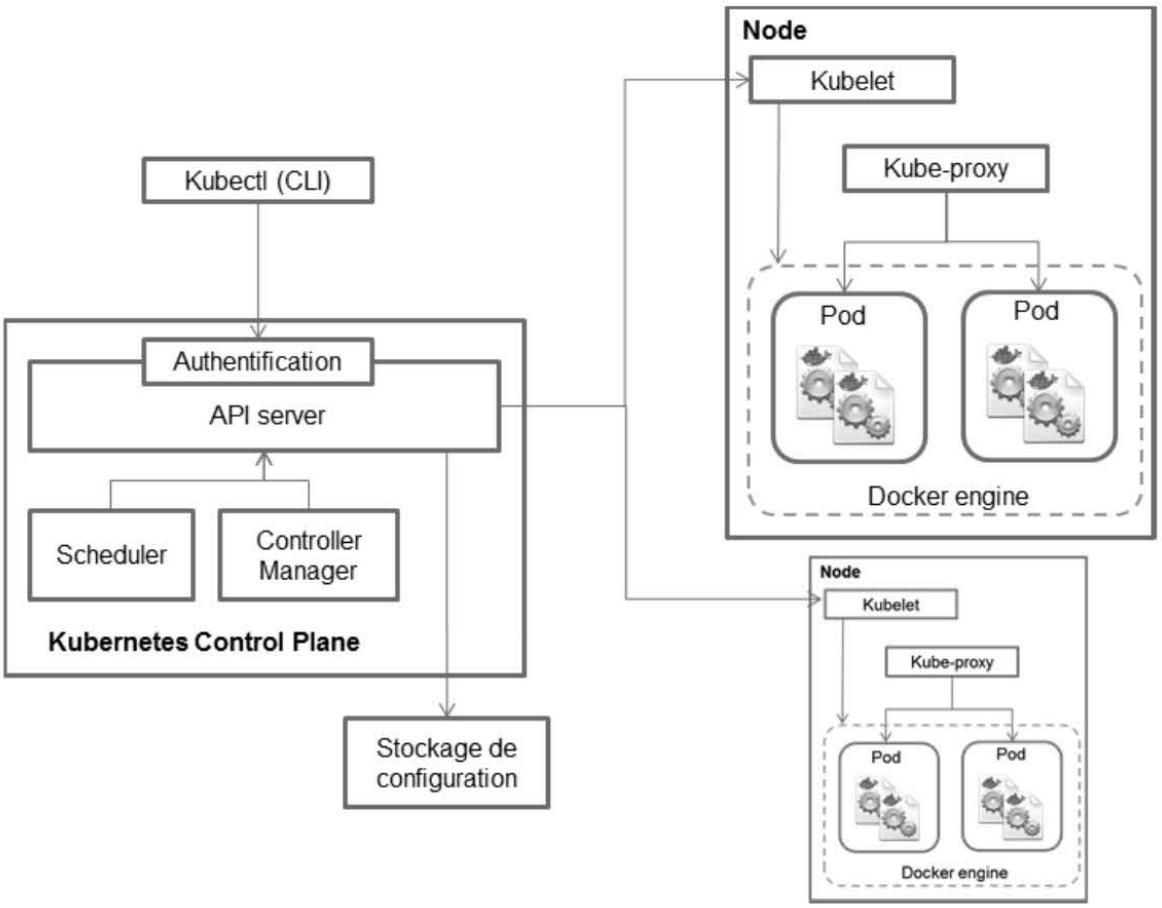

Figure 2.9 — Architecture de Kubernetes

Les *kubelets* sont les agents qui contrôlent les conteneurs qui sont regroupés en *pod* dont nous allons évoquer les caractéristiques un peu plus bas. Ces *kubelets* contrôlent le cycle de vie des conteneurs et surveillent leur état (en utilisant notamment cAdvisor[2] en lien avec le moteur Docker). On trouve aussi sur chaque nœud un autre composant important : le *kube-proxy*.

Le *kube-proxy* est un proxy répartiteur de charge (*load balancer*) qui gère les règles d'accès aux services (la plupart du temps des interfaces http/https) offerts par les *pods*, mais **à l'intérieur du cluster Kubernetes uniquement**. Ce dernier point est important et nous y reviendrons dans la prochaine section.

Les *kube-proxies* sont mis à jour chaque fois qu'un service Kubernetes est mis à jour (qu'il soit créé ou que son implémentation ait changé de place) et ils permettent

1. *Pod*, que l'on pourrait traduire en français par réceptacle ou enveloppe, désigne un groupe de conteneurs qui collaborent étroitement pour fournir un service donné. Ils vont donc, dans l'architecture Kubernetes, pouvoir communiquer entre eux sans intermédiaires.
2. https://github.com/google/cadvisor

aux conteneurs de s'abstraire de la localisation réelle des autres conteneurs dont ils pourraient avoir besoin.

On notera que le *Kubernetes Control Plane* est en fait constitué de plusieurs sous-composants :

- un serveur d'API REST « API server », qui est notamment utilisé par Kubectl, la ligne de commande Kubernetes qui permet de piloter tous les composants de l'architecture ;
- le scheduler, qui est utilisé pour provisionner sur différents nœuds (*node*) de nouveaux pools de conteneurs en fonction de la topologie, de l'usage des ressources ou de règles d'affinité plus ou moins complexes (sur le modèle des stratégies Swarm que nous avons évoquées précédemment). Dans le futur, Kubernetes affirme vouloir s'ouvrir à différentes implémentations, via un mécanisme de plugins, probablement pour prendre en compte des règles de plus en plus sophistiquées ;
- le *controller manager*, qui exécute les boucles de contrôle du système ou *controllers* (à une fréquence déterminée) pour vérifier que les règles de gestion programmées sont respectées (nombre d'instances de certains conteneurs, par exemple).

Node, pod et conteneurs

Kubernetes organise les applications à base de conteneurs selon une structure et un modèle réseau propre auxquels il faut se soumettre.

Nous avons déjà vu que les nœuds ou *node* (correspondant généralement à un hôte physique ou virtuel) hébergent un ou plusieurs *pods*. Un *pod* est un groupe de conteneurs qui seront toujours co-localisés et provisionnés ensemble. Ce sont des conteneurs qui sont liés et qui offrent une fonction (une instance de micro-service). À ce titre, tous les conteneurs d'un *pod* vont se trouver dans une sorte de même machine logique :

- ils partagent les mêmes volumes (stockage partagé) ;
- ils peuvent interagir en utilisant le nom de domaine localhost, ce qui simplifie les interactions entre conteneurs à l'intérieur d'un même *pod*.

Modèle réseau Kubernetes

En réalité, le modèle réseau de Kubernetes va encore plus loin. Chaque *pod* est associé à une adresse IP unique. De plus, l'espace d'adressage à l'intérieur d'un cluster Kubernetes est plat, c'est-à-dire que tous les *pods* peuvent se parler en utilisant l'adresse IP qui leur a été allouée sans avoir recours à de la translation d'adresse (NAT ou *Network Adress Translation*).

Cette pratique facilite l'implémentation de clusters multi-hôtes, mais diffère notablement du modèle réseau par défaut de Docker qu'il est néanmoins possible de changer. C'est d'ailleurs ce qu'offre Swarm en créant un moteur Docker au travers de plusieurs hôtes.

Si un *pod* est une instance de micro-service, alors qu'est-ce qu'un micro-service ?

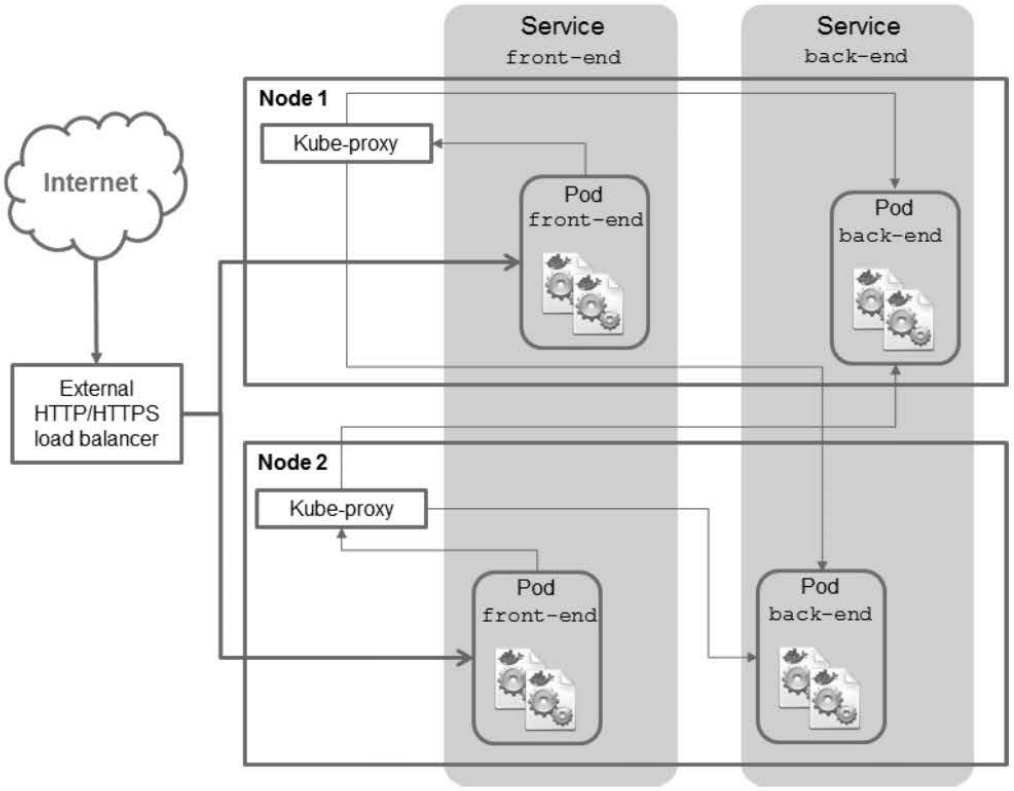

Figure 2.10 — Topologie, services et load balancing

Kubernetes définit un service (ou micro-service) comme un ensemble de *pods* associés à des règles d'accès, de réplication et de répartition de charge spécifiques. Le service est une abstraction (grâce notamment aux *kube-proxies*) qui permet à un consommateur (à l'intérieur du cluster) de ne pas avoir à connaître la localisation ou même le nombre des *pods* qui produisent la fonctionnalité consommée.

Attention néanmoins, la notion de service est interne au cluster Kubernetes. En effet, le *kube-proxy* (figure 2.10) permet, par exemple, à un *pod front-end* de trouver un *pod back-end*. Pour ce faire, il demande à accéder au service back-end (automatiquement déclaré sur tous les *kube-proxy* du cluster) et le *kube-proxy* de son *node* va le router automatiquement vers une instance de *pod back-end* en utilisant une règle simple de type *round robin*[1]. Grâce à ce mécanisme, le front-end n'a pas besoin de savoir où se trouve le back-end et à combien d'exemplaires il existe.

External load balancer

Par contre si un utilisateur extérieur souhaite accéder au front-end, il est nécessaire d'utiliser un *external load balancer*, c'est-à-dire un répartiteur de charge externe au cluster qui va être associé à une IP publique et offrir les fonctions habituelles :

- chiffrement SSL ;

1. *Round robin* est un algorithme de répartition de traitement dans lequel les processus participants sont sollicités à tour de rôle.

- *load balancing* de niveau 7 ;
- affinité de session ;
- etc.

La configuration de ce type de service (car il s'agit bien d'un service) dépend du fournisseur de l'implémentation de Kubernetes. Dans le cas de Google Container Engine, l'offre cloud de Google, l'implémentation de cet *external load balancer* est basée sur le *Google Compute Engine network load balancer*[1].

Depuis sa version 1.2, Kubernetes a même introduit un nouveau type de contrôleur, *Ingress controller*, pour provisionner dynamiquement ce type de composant externe (nommé de ce fait *Ingress load balancer*, qui pourrait se traduire par « load balancer entrant »). L'objectif est de s'intégrer à différentes solutions (reverse proxy, load balancer, firewall), mais, à cette date, la seule implémentation disponible (à l'état de beta) est basée sur HAProxy[2].

Avec Kubernetes, Google tente de s'imposer comme l'implémentation de référence du CaaS. Kubernetes, par rapport à l'offre de Docker, est plus intégré, si bien que sa présence n'est pas étonnante dans d'autres solutions, notamment :

- Mesos (ou DCOS[3] sa déclinaison commerciale) dont Kubernetes constitue un framework, au même titre que Marathon (que nous présentons dans la suite de ce chapitre) ;
- OpenShift[4], la solution PaaS de RedHat ;
- OpenStack, la solution open source de référence pour la gestion de IaaS publiques et privées.

Notons enfin un projet comme Ubernetes[5] qui étudie la possibilité de proposer un modèle CaaS multi-cluster pour gérer plusieurs data centers et donc éventuellement plusieurs fournisseurs de cloud privés ou publics. Ainsi, une entreprise pourrait choisir de répartir ses applications de manière transparente entre son infrastructure interne et Google Container Engine.

2.2.3 DCOS : le cas Apache Mesos et Marathon

Apache Mesos est à l'origine même du concept de système d'exploitation de centre de calcul (ou DCOS pour *Data Center Operating System* qui est d'ailleurs le nom de la version commerciale de Mesos) que nous avons déjà évoqué précédemment.

1. https://cloud.google.com/compute/docs/load-balancing/network/
2. http://www.haproxy.org/ site de HA Proxy et implémentation au sein de Kubernetes https://github.com/kubernetes/contrib/tree/master/service-loadbalancer
3. https://www.mesosphere.com/
4. https://www.openshift.com/
5. tinyurl.com/ubernetes

Mesos a été développé dans le cas d'un projet de recherche de l'université américaine de Berkeley. Il est notamment utilisé pour la gestion de l'infrastructure de Twitter ou d'AirBnb.

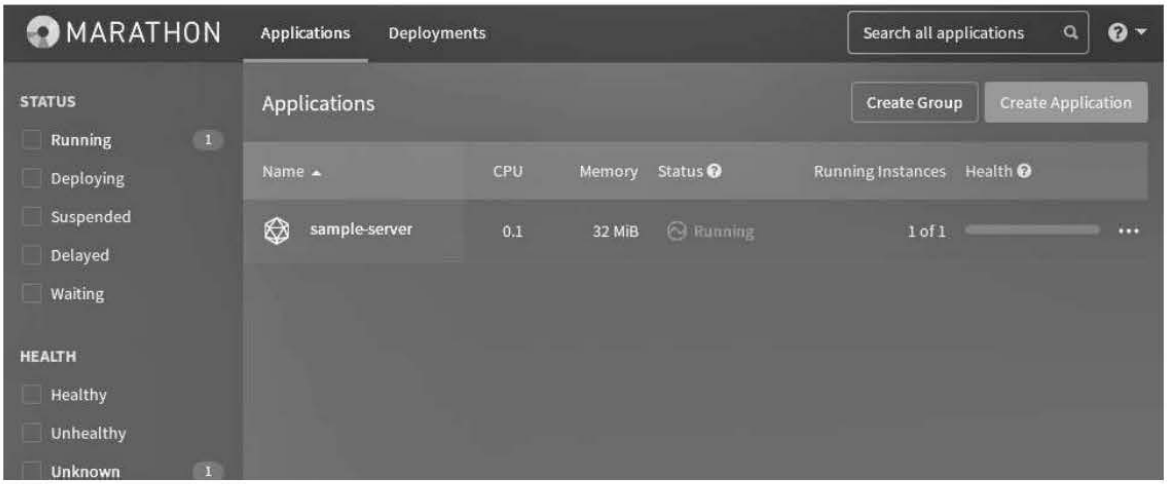

Figure 2.11 — Interface graphique de contrôle de Marathon

Mesos a une vocation plus large qu'un CaaS classique, dans la mesure où il est apparu avant la vague Docker. Mesos offre notamment un concept de framework qui lui permet de se brancher sur diverses solutions de contrôle, notamment Kubernetes, mais aussi Marathon, la solution maison de gestion de conteneurs.

En pratique, on retrouve exactement la même architecture conceptuelle :

- un mesos *master* (éventuellement répliqué dans le cadre d'une architecture à haute disponibilité) ;
- des mesos *slave* qui hébergent les agents qui vont exécuter les ordres émis depuis le master et fournir des informations en temps réel sur l'état des hôtes.

Tout comme les solutions précédemment étudiées, Mesos/Marathon va utiliser divers algorithmes pour mettre en correspondance les ressources (CPU, RAM ou autre) disponibles dans l'infrastructure et les demandes d'exécutions d'applications (synchrone ou batch).

À l'inverse des solutions pure CaaS, comme Kubernetes ou Swarm, Mesos peut aussi utiliser d'autres technologies de déploiement et d'exécution. En plus des framework Marathon ou Kubernetes, Mesos peut utiliser:

- des *schedulers* classiques, comme Jenkins ou Chronos (qui peuvent donc exécuter n'importe quel type de commande) ;
- des solutions Bigdata, comme Hadoop ;
- des solutions de stockage de données, comme Cassendra.

2.2.4 Fleet + Rkt + Etcd : la solution de CoreOS

La solution de CoreOS est présentée très sommairement au travers de la figure 2.12. Celle-ci, quoique complète et open source, est probablement plus confidentielle que celles que nous avons citées précédemment.

Figure 2.12 — Offre de gestion CaaS de CoreOS

La suite de CoreOS ressemble à celle de Docker, dans la mesure où elle est constituée de produits indépendants organisés en une suite plus ou moins intégrée.

Notons qu'etcd, qui est un projet de la société CoreOS, est utilisé pour assurer la haute disponibilité d'autres solutions dont Docker Swarm et Kubernetes.

2.2.5 Les offres cloud de *container service*

Les trois grands acteurs du monde cloud offrent aujourd'hui des solutions CaaS à base de conteneurs. Si ces solutions sont encore peu utilisées, on constate qu'il s'agit pour tous d'un axe de développement majeur.

Google Container Engine

Il y a peu de chose à ajouter sur cette offre lancée par Google en novembre 2014, si ce n'est qu'elle s'appuie évidemment sur Kubernetes, projet open source financé et promu par le géant californien.

Amazon ECS, l'autre offre cloud

Amazon a lancé son offre *Amazon EC2 Container Service* (ou ECS) en novembre 2014. Cette offre ne s'appuie ni sur Kubernetes, ni sur la suite de Docker. Amazon a tout simplement fabriqué sa propre solution.

Celle-ci s'appuie sur un agent open source[1], mais sur un contrôleur qui, lui, ne l'est évidemment pas, puisqu'il est uniquement implémenté par Amazon EC2 (dans

1. https://github.com/aws/amazon-ecs-agent

le cloud). Le choix de mettre le code de l'agent ECS en open source est peut-être motivé par l'idée de proposer des offres hybrides public/privé à des sociétés qui y trouveraient leur intérêt. On pourrait en effet envisager une société qui s'appuierait sur un front-end hébergé dans le cloud Amazon et un back-end privé, hébergé dans un cloud d'entreprise, l'ensemble étant géré par un contrôleur fourni... par Amazon.

Azure Container Service

Microsoft est sans aucun doute le dernier arrivé dans la course, tant sur la technologie Docker que pour la fourniture d'un CaaS complet.

Comme nous l'avons évoqué en début de chapitre, le lien étroit entre conteneur et Linux (ou du moins Unix) est probablement la raison de ce retard.

Aujourd'hui, Azure Container Service s'appuie sur la virtualisation matérielle maison HyperV et propose deux déclinaisons de CaaS basées sur :

- Mesos et Marathon ;
- Docker Swarm.

Gageons que la sortie de Windows 2016 et de son implémentation native de Docker (évoquée en début de chapitre) apportera son lot de nouveautés, la proximité entre la firme de Redmond et la startup de San Francisco ne pouvant qu'engendrer des synergies intéressantes.

2.3 ANSIBLE, CHEF ET PUPPET : OBJET ET LIEN AVEC DOCKER ET CAAS

Les solutions CaaS sont clairement des nouveautés sur le marché de la gestion d'infrastructures (configuration et déploiement). La plupart des départements informatiques doivent composer avec un existant constitué d'un large panel de technologies. Il est donc probable que ces systèmes ne soient mis en œuvre que pour des applications nouvelles ou, plus certainement, par des startups ayant la chance de pouvoir s'offrir un *greenfield*[1].

La technologie des conteneurs (et plus spécifiquement Docker) a néanmoins rapidement percolé au sein de solutions d'administration de systèmes existantes. Ces solutions ont simplement ajouté à leur portefeuille la gestion des conteneurs, en raison des avantages qu'ils offrent pour le conditionnement d'applications.

2.3.1 Objectif de ces solutions

À partir de 2005, plusieurs solutions sont apparues visant à unifier la gestion de configuration d'infrastructures informatiques. Citons par exemple Puppet (sortie en 2005), Chef (sortie en 2009) ou Ansible (qui date de 2012).

1. Littéralement un « champ vert » qui symbolise l'absence d'existant, aussi appelé *legacy*.

Ces solutions se basent sur des DSL[1] (*Domain Specific Language*) qui permettent de déclarer la configuration cible à atteindre pour des systèmes éventuellement distribués.

Cette configuration peut recouvrir :

- la configuration de l'OS ;
- l'installation et la configuration de logiciels standards ou personnalisés ;
- la gestion de services (démarrage, redémarrage) ;
- l'exécution de commandes distantes.

Le but de ces outils est de remplacer les nombreux scripts (shell, python, perl, etc.) couramment utilisés pour les tâches courantes de configuration et de déploiement par une bibliothèque d'actions standardisée utilisant un seul langage, une seule solution.

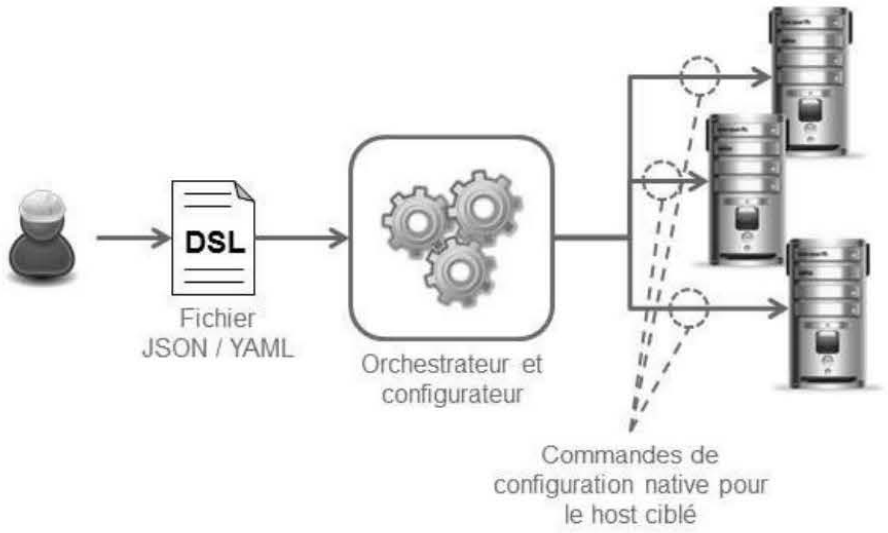

Figure 2.13 — Principe des DSL de configuration

Outre l'uniformisation de l'outillage, ces solutions facilitent les transferts de connaissance et permettent de s'abstraire (dans une certaine mesure) des évolutions technologiques. En effet, les personnels n'ont plus besoin de connaître tous les détails et spécificités des systèmes opérés. Ils peuvent aussi communiquer plus simplement les uns avec les autres.

Enfin, ces outils permettent de simplifier l'élaboration de procédures de configuration indépendantes des environnements. Elles sont donc particulièrement adaptées aux approches de type intégration continue ou déploiement continu. Un seul ensemble de scripts peut ainsi être utilisé pour déployer sur les différents environnements applicatifs : développement, qualification, pré-production et production.

1. Un DSL est un langage « spécialisé », c'est-à-dire qu'il vise à un usage particulier (comme le langage Dockerfile qui vise à construire des images de conteneurs). Le DSL se définit par opposition aux langages universels, comme Java, C, Python, etc.

2.3.2 Un exemple : Ansible

Nous allons prendre, à titre d'exemple, le cas d'Ansible (http://www.ansible.com/). Nous verrons par la suite que les principes de fonctionnement sont relativement similaires pour les différentes solutions.

Ansible est une solution appartenant à Red Hat Software (l'éditeur de la distribution Linux éponyme) et disponible en version open source depuis 2012. Ansible s'appuie sur une bibliothèque de modules permettant de définir des descripteurs de configuration appelés *playbooks*. Ansible s'appuie sur une très large bibliothèque de modules couvrant la grande majorité des actions les plus courantes : copie de fichier, installation de logiciels, gestion de services, etc.

> Contrairement à d'autres solutions, Ansible est une solution dite *agentless*. Cela signifie qu'il n'impose pas l'installation d'un agent sur l'hôte qui doit être configuré. La seule exigence est de disposer d'un accès SSH et que Python (2.x) soit installé sur cette machine distante.

La notion d'inventaire

Une fois Ansible installé sur la machine de contrôle (celle qui va configurer les autres), une ligne de commande va permettre de prendre le contrôle des hôtes ciblés. Ces hôtes cibles sont organisés dans ce qu'Ansible nomme un *inventory*. Celui-ci, par défaut, se trouve dans le fichier /etc/ansible/hosts dont voici un exemple :

```
[frontend]
fe01.example.com
fe02.example.com
fe03.example.com

[backend]
be01.example.com
be02.example.com
```

Ces machines (qui peuvent être virtuelles grâce à des connecteurs pour diverses solutions cloud comme AWS) sont organisées en groupes. Ces groupes vont permettre de lancer des actions de masse. On peut vouloir, par exemple, mettre à jour tous les serveurs web ou bien faire un ping sur toutes les machines de l'inventaire, comme ci-dessous :

```
$ ansible all -m ping
fe01.example.com | SUCCESS => {
"changed" : false,
"ping" : "pong"
}
fe02.example.com | SUCCESS => {
"changed" : false,
"ping" : "pong"
}
...
```

Dans le cas ci-dessus le `all` signifie « toutes les machines de l'inventaire » qui, dans ce cas, n'est constitué que d'un seul et unique hôte. Si l'inventaire correspondait à l'exemple de fichier /etc/ansible/hosts listé plus haut, alors la commande suivante serait aussi valide :

```
$ ansible backend -m ping
```

Cette commande appliquerait la commande ping aux hôtes be01.example.com et be02.example.com.

Un exemple de playbook

Un *playbook* Ansible va décrire l'état cible souhaité pour un ou plusieurs hôtes. L'exemple ci-dessous (stocké dans un fichier install_apache.yml) vise à installer le serveur web apache :

```
$ cat install_apache.yml
---
- hosts: all
user: maf
become: yes
become_user: root
tasks:
- name: installer la dernière version d'apache
yum: name=httpd state=latest
- name: configurer apache pour se lancer au démarrage du système
service: name=httpd state=started enabled=yes
```

Il s'agit d'un exemple de descripteur Ansible au format YAML.

> Nous n'entrerons pas dans le détail de l'usage d'Ansible, mais sachez que le fichier ci-dessous suppose un certain nombre de choses concernant l'hôte cible. Nous voyons notamment qu'un utilisateur « maf » doit exister sur cet hôte et que celui-ci doit être pourvu des droits sudo, afin de pouvoir fonctionner en mode root. Il est aussi nécessaire d'avoir fait un échange de clefs SSH entre le contrôleur et l'hôte à configurer.

Ansible va se connecter à l'hôte (via SSH) et :

- vérifier qu'Apache est bien installé et qu'il s'agit de la dernière version disponible ;
- que celui-ci est bien installé pour démarrer au boot de l'hôte.

Une description, pas un script impératif

Il est important de comprendre que le descripteur ci-dessus n'est pas une séquence de commandes impératives. Il s'agit de la description d'un état souhaité pour l'hôte.

Lançons la commande une première fois sur un hôte sur lequel Apache n'est pas installé :

```
$ ansible-playbook install_apache.yml
PLAY
***************************************************************************

TASK [setup]
*******************************************************************
ok: [10.0.2.4]

TASK [ensure apache is at the latest version]
**********************************
changed: [10.0.2.4]

TASK [ensure apache is running (and enable it at boot)]
************************
changed: [10.0.2.4]

PLAY RECAP
*********************************************************************
10.0.2.4                   : ok=3    changed=2    unreachable=0    failed=0
```

La séquence se conclut par le message `changed = 2` qui fait référence aux deux actions opérées par Ansible :

- installer Apache avec sa dernière version ;
- faire en sorte qu'il démarre au boot de la machine.

Si nous lançons la commande une seconde fois :

```
$ ansible-playbook install_apache.yml

PLAY
***************************************************************************

TASK [setup]
*******************************************************************
ok: [10.0.2.4]

TASK [ensure apache is at the latest version]
**********************************
ok: [10.0.2.4]

TASK [ensure apache is running (and enable it at boot)]
************************
ok: [10.0.2.4]

PLAY RECAP
*********************************************************************
10.0.2.4            : ok=3   changed=0   unreachable=0   failed=0
```

Le résultat est différent : `changed=0`

En effet, Ansible collecte, avant d'effectuer des actions, des informations relatives à l'environnement de la machine (on parle de *facts*). Ensuite Ansible analyse la

différence entre cette configuration et la configuration souhaitée. Les actions requises (pas nécessairement toutes) sont ensuite lancées pour atteindre la configuration cible.

Lorsque nous avons lancé la commande une seconde fois, comme le système était déjà dans l'état cible, aucune action n'a été lancée.

La configuration actuelle collectée par Ansible est visualisable par la commande suivante (dont seul un court extrait du résultat est affiché) :

```
$ ansible all -m setup
10.0.2.4 | SUCCESS => {
"ansible_facts": {
"ansible_all_ipv4_addresses": [
"192.168.122.1",
"10.0.2.4"
],
"ansible_all_ipv6_addresses": [
"fe80::a00:27ff:fe78:602d"
],
"ansible_architecture": "x86_64",
"ansible_bios_date": "12/01/2006",
"ansible_bios_version": "VirtualBox",
"ansible_cmdline": {
"BOOT_IMAGE": "/vmlinuz-3.10.0-327.el7.x86_64",
"LANG": "en_US.UTF-8",
"crashkernel": "auto",
"quiet": true,
(...)
```

2.3.3 Le lien avec Docker et les solutions CaaS

Comme nous l'avons vu, Ansible présente une similarité conceptuelle avec les CaaS. Le système fonctionne en comparant un état actuel avec l'état cible.

> Chef ou Puppet fonctionnent selon le même principe, si ce n'est qu'à la différence d'Ansible, ceux-ci requièrent l'installation d'un agent sur l'hôte.

La différence avec les solutions Kubernetes ou Swarm évoquées précédemment est que cette comparaison d'état n'est pas permanente. Ansible vise à configurer un système, et non pas à le gérer en temps réel en analysant à chaque instant si un processus est ou n'est plus actif.

Les solutions CaaS et celles de gestion de configuration de type Ansible ne sont donc pas, en pratique, des solutions concurrentes, mais complémentaires. En premier lieu, ces solutions peuvent prendre en charge l'installation et la configuration des agents CaaS (kubelets ou agent Swarm) sur les hôtes. Ensuite, certaines de ces solutions de gestion de configuration disposent de modules pour agir sur des environnements à base de conteneurs.

Ansible dispose notamment d'un *playbook* Docker.

Ainsi le *playbook* suivant va s'assurer qu'un conteneur nommé myserver, construit à partir de l'image nginx (un serveur web que nous utiliserons fréquemment dans la suite de cet ouvrage), est actif :

```
---
- hosts: all
  user: maf
  tasks:
    - name: fait en sorte qu'un conteneur web tourne sur la machine cible
      docker:
        name: myserver
        image: nginx
        state: started
        ports: 8080:80
```

Ansible va ainsi gérer tout le cycle de vie du conteneur, depuis le chargement de l'image jusqu'à sa création et son démarrage (ou redémarrage au besoin). Autant de commandes qu'il n'est plus nécessaire de lancer manuellement une par une.

2.3.4 Controller Kubernetes avec Puppet ou Chef

Pour conclure sur ce sujet de la complémentarité des solutions, notons que Puppet ou Chef disposent aujourd'hui de modules (on parle de *cookbook* pour Chef, mais la notion est similaire) pour gérer les déploiements sur Kubernetes ou Swarm.

Dans ce type de cas, Puppet va directement (ou en s'appuyant sur le package manager Helm[1]) déployer une application sur un cluster Kubernetes.

On note dans ce cas que le cluster Kubernetes est considéré comme un hôte, ce qui correspond exactement au niveau d'abstraction qu'un CaaS cherche à atteindre.

En résumé

Dans ce chapitre, nous avons vu la manière dont les conteneurs, et plus spécifiquement Docker, étaient en train de coloniser les solutions de gestion d'infrastructures. Nous avons étudié les solutions CaaS dédiées aux architectures applicatives exclusivement à base de conteneur. Nous nous sommes aussi attardés sur les solutions de gestion de configuration, comme Ansible, Puppet ou Chef qui s'intègrent aussi à Docker ou des solutions CaaS, comme Kubernetes.

1. Helm (https://helm.sh) est un package manager dédié à Kubernetes. Il se pose en équivalent de « yum » ou « apt » non plus pour un hôte mais à l'échelle d'un cluster Kubernetes.

DEUXIÈME PARTIE

Docker en pratique : les outils de base

Cette seconde partie constitue une prise en main de Docker et des outils de base de son écosystème.

Le premier chapitre décrit l'installation des outils Docker et de l'environnement qui sera ensuite utilisé pour les différents exemples pratiques de cet ouvrage. Ce chapitre nous permet aussi d'aborder les bases du fonctionnement du démon et du client Docker.

Le deuxième chapitre est consacré spécifiquement à deux exemples significatifs d'installation et de configuration d'hôtes Docker. Nous y abordons aussi des questions relatives à la configuration du stockage Docker pour les OS de type Fedora.

Le dernier chapitre aborde la création et le cycle de vie des conteneurs au travers d'un exemple pratique. À ce titre, nous verrons pour la première fois quelques exemples de commandes Docker.

3

Prise en main

Installer et configurer Docker

L'objectif de ce chapitre est d'installer Docker sur votre ordinateur et de démarrer votre premier conteneur. Sous Microsoft Windows (abrégé par la suite en Windows) ou Mac OS X, nous verrons comment, au moyen d'une machine virtuelle Linux, nous pouvons jouer avec Docker.

À l'issue de ce chapitre vous aurez un système prêt pour la suite des exercices et cas pratiques présentés dans cet ouvrage et vous aurez compris comment interagir avec le démon Docker.

3.1 INSTALLATION DE DOCKER

L'installation de Docker est en général simple et rapide à effectuer sur une distribution Linux. On peut bien sûr aussi l'utiliser à travers un environnement virtualisé sous Windows et Mac OS X. Prochainement, grâce aux implémentations natives en préparation (s'appuyant sur une VM xhyve sous Mac OS X et Hyper-V sous Windows), il sera aussi possible d'installer Docker directement sous Windows ou Mac OS X.

3.1.1 Docker Toolbox : la solution rapide pour Windows et Mac OS X

Le plus simple pour démarrer rapidement et efficacement Docker sous Windows ou Mac OS X est d'utiliser Docker ToolBox. Ce dernier est un programme d'installation élaboré par Docker Inc., qui contient tous les logiciels nécessaires pour l'utilisation de Docker sur un environnement autre que Linux :

- Oracle VirtualBox, logiciel de virtualisation open source[1] ;
- Docker Machine, qui permet de créer un hôte Docker (nous l'aborderons en détail dans un prochain chapitre) ;
- Docker Engine, le démon et le client Docker ;
- Docker Kitematic, une interface graphique pour Docker ;
- Docker Compose, outil simplifiant l'interconnexion de conteneurs Docker, disponible uniquement sous Mac OS X.

Docker ToolBox utilise Docker Machine pour créer une machine virtuelle (VM) Linux minimale tournant sous Windows ou Mac OS X, grâce à VirtualBox. Cette VM fait tourner le démon Docker qui peut ensuite être contrôlé depuis votre système d'exploitation de différentes façons :

- via Kitematic ;
- via le client Docker ;
- en direct grâce à des requêtes HTTP (n'oublions pas que le démon Docker expose son API en REST[2]).

Pour pouvoir installer Docker Toolbox, votre machine doit disposer de Windows 7 ou supérieur, ou Mac OS X 10.8 ou supérieur.

Le processus d'installation est relativement simple :

- allez sur la page de Docker ToolBox[3] et téléchargez l'installeur correspondant à votre système d'exploitation ;
- lancez l'installeur en double-cliquant dessus.

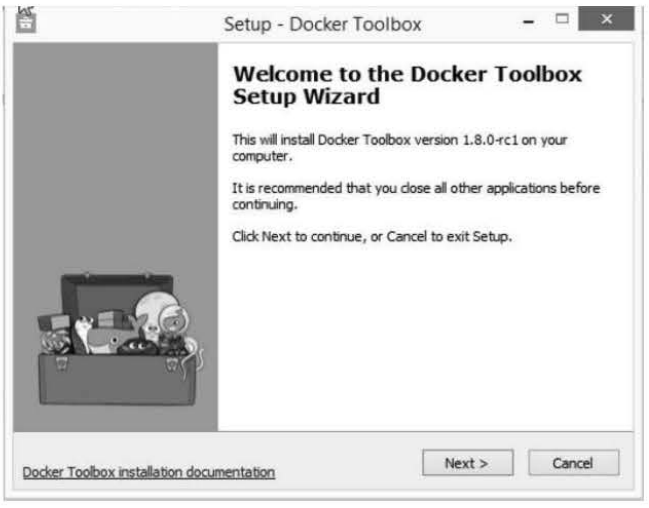

Figure 3.1 — Installation de Docker Toolbox sous Windows

1. Remplacé prochainement par xhyve sous Mac OS X et Hyper-V sous Windows.
2. https://fr.wikipedia.org/wiki/Representational_State_Transfer
3. https://www.docker.com/docker-toolbox

Une fois l'installation terminée, lancez le Docker CLI (sous Windows) ou le Docker Quickstart Terminal (sous Mac OS X), puis :

- ouvrez un terminal ;
- créez une machine virtuelle, qui sera nommée *default* si elle n'existe pas, et démarrez-la. C'est une VM légère (environ 24 Mo) qui tourne uniquement en RAM ;
- faites pointer le client Docker vers le démon Docker tournant sur la machine virtuelle et sécurisez-le avec des certificats TLS.

Bien comprendre la différence avec une installation native sous Linux

Lors d'une installation de Docker sous Linux, tous les composants Docker (client, démon, conteneur) tournent directement sur votre localhost. Les conteneurs sont donc directement atteignables via un adressage local, par exemple localhost:8000.

Dans le cas de Windows ou Mac OS X, vos conteneurs tournent à l'intérieur d'une VM Linux. Il faut donc utiliser l'adresse IP de la VM pour les atteindre. L'adresse IP s'obtient facilement avec la commande :

```
$ docker-machine ip default
192.168.99.100
```

Il suffit ensuite d'utiliser cette adresse au lieu de localhost : 192.168.99.200:8000

L'utilisation de Docker Toolbox, bien que pratique, a néanmoins ses limites dès que l'on commence à utiliser les *data volumes* car la machine virtuelle et la machine hôte ne partagent pas le même système de fichiers. Heureusement, il existe une autre méthode, utilisant les capacités de partage de fichiers entre hôte et invité de VirtualBox (nommée *shared folders*), qui permet de s'affranchir de ces problèmes, mais au prix de performances relativement médiocres.

Soyons clairs, la Docker Toolbox est essentiellement un démonstrateur, un outil de prise en main, ou éventuellement un outil de développement monoposte. Il n'est évidemment pas question de l'utiliser sur un serveur et encore moins en production.

3.1.2 Linux sous VirtualBox

L'idée est d'installer une distribution Linux, de type CentOS ou Ubuntu, sur une VM dans VirtualBox et d'installer ensuite Docker de manière standard.

Pour simplifier cette étape, nous vous proposons d'utiliser Vagrant[1].

Vagrant est « un outil pour fabriquer des environnements de développement ». Il permet de créer une box (un package Vagrant) et de la rendre ensuite disponible à d'autres utilisateurs pour créer simplement une machine virtuelle VirtualBox ou

1. http://www.vagrantup.com

VMWare. L'environnement est décrit via un fichier de configuration qui en spécifie toutes les caractéristiques.

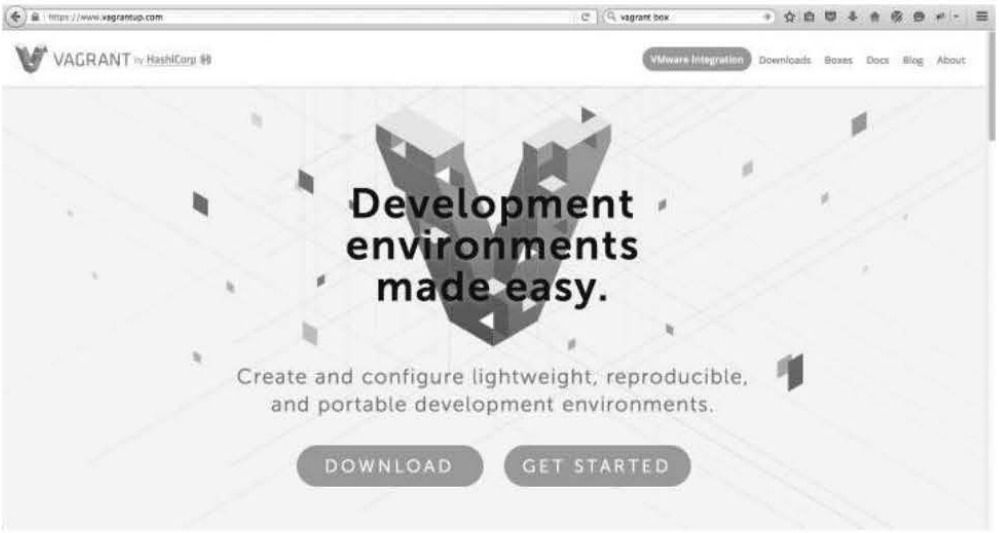

Figure 3.2 — Vagrant, le configurateur de machine virtuelle

Une VM Linux CentOS

Nous avons préparé une box Vagrant, basée sur un CentOS 7[1], avec un environnement graphique qui vous permettra de mettre en œuvre les différents exemples et cas pratiques de ce livre.

Il vous faut pour cela :

- installer VirtualBox[2] ;
- installer Vagrant[3] ;
- exécuter dans un terminal les commandes suivantes :

```
$ vagrant init dunod-docker/centos7
A 'Vagrantfile' has been placed in this directory. You are now
ready to 'vagrant up' your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
'vagrantup.com' for more information on using Vagrant.
```

Vagrant va télécharger la box et ensuite créer une machine virtuelle automatiquement.

1. https://atlas.hashicorp.com/dunod-docker/boxes/centos7
2. http://virtualbox.org (version utilisée dans ce livre 5.1.14)
3. https://docs.vagrantup.com/v2/installation/index.html (version utilisée dans ce livre 1.7.4)

```
$ vagrant up --provider virtualbox
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box ' dunod-docker/centos7'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box ' dunod-docker/centos7' is up
to date...
==> default: Setting the name of the VM: VM_default_1462554217727_87155
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 22 => 2222 (adapter 1)
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default: Warning: Connection timeout. Retrying...
    default: Warning: Remote connection disconnect. Retrying...
    default:
    default: Vagrant insecure key detected. Vagrant will automatically replace
    default: this with a newly generated keypair for better security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's present...
    default: Key inserted! Disconnecting and reconnecting using new SSH key...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Mounting shared folders...
    default: /vagrant => C:/Users/dunod/Documents /VM
```

Une fois l'image créée, ouvrez VirtualBox et sélectionnez la machine virtuelle nouvellement créée pour en afficher l'interface graphique.

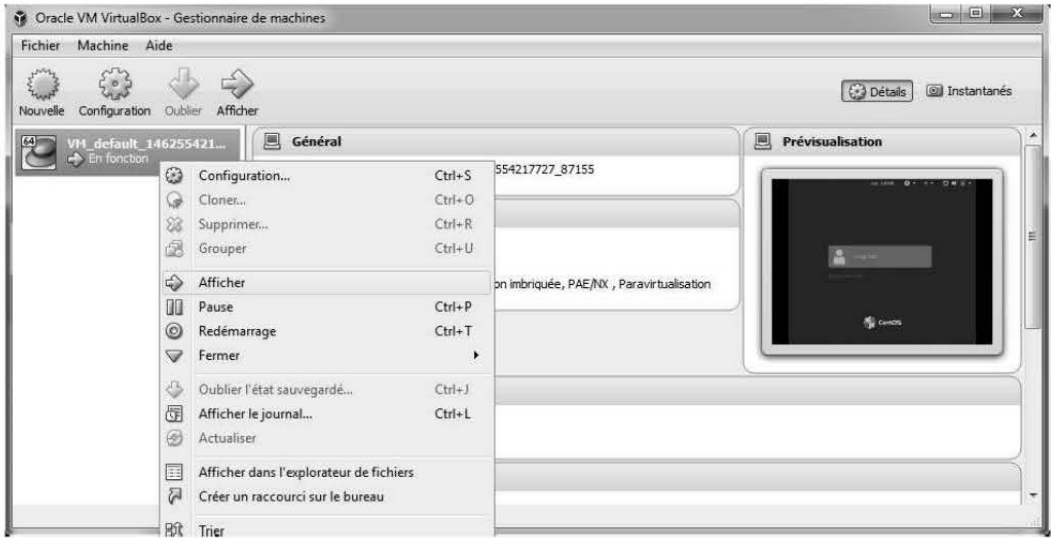

Figure 3.3 — Accès à l'image Vagrant pour se connecter

Vous pouvez ensuite vous connecter sur cette machine virtuelle avec l'utilisateur *vagrant* (mot de passe *vagrant*).

Figure 3.4 — Page de login CentOS7 de notre VM

Mise à jour des guest additions VirtualBox

Par défaut, la VM ainsi créée est isolée de l'hôte sur lequel vous avez installé VirtualBox. Il n'y a pas de possibilité de copier/coller, ni de transfert de fichiers. La chose peut être relativement pénible.

Heureusement, VirtualBox offre une fonctionnalité très utile, nommée *guest additions*, qui facilite la communication entre l'hôte et l'invité (la machine virtuelle que vous venez de créer).

L'image Vagrant que nous livrons est installée avec les *guest additions* pour la version 5.10.14 de VirtualBox. Si vous disposez d'une version plus récente de VirtualBox, il vous faudra les mettre à jour (VirtualBox vous y incitera sans doute par des messages répétés ou le fera même automatiquement lors de la création de la VM).

Voici comment procéder :

- arrêtez votre VM ;
- ajoutez un CDROM à votre VM (allez dans Configuration, Stockage, puis ajouter un lecteur optique que vous laisserez vide) ;
- redémarrez votre VM ;
- demandez l'installation de la dernière version des *guest additions* (ceci se fait en cliquant sur le menu *Périphériques*, puis *Insérer l'image CD des Additions Invité…* dans l'interface de VirtualBox, comme cela est illustré à la figure 3.5).

Le programme d'installation des *guest additions* vous demande de redémarrer votre VM. **Arrêtez cette dernière et lisez le paragraphe suivant avant de la lancer à nouveau.**

Activation du partage de fichiers avec l'hôte

Maintenant que vous disposez de guest additions à jour, il est possible d'activer le partage de fichiers entre l'hôte et l'invité. Ceci sera utile à plusieurs reprises pour échanger des fichiers entre la VM et le système sur lequel vous avez installé VirtualBox.

Pour ce faire, rendez-vous dans le panneau de configuration de votre VM (désormais arrêtée) puis sélectionnez *Dossiers partagés*. Un lien existe déjà. Double-cliquez sur ce lien et sélectionnez (comme indiqué sur la figure 3.6) *Montage automatique*.

Figure 3.5 — Mise à jour des guest additions

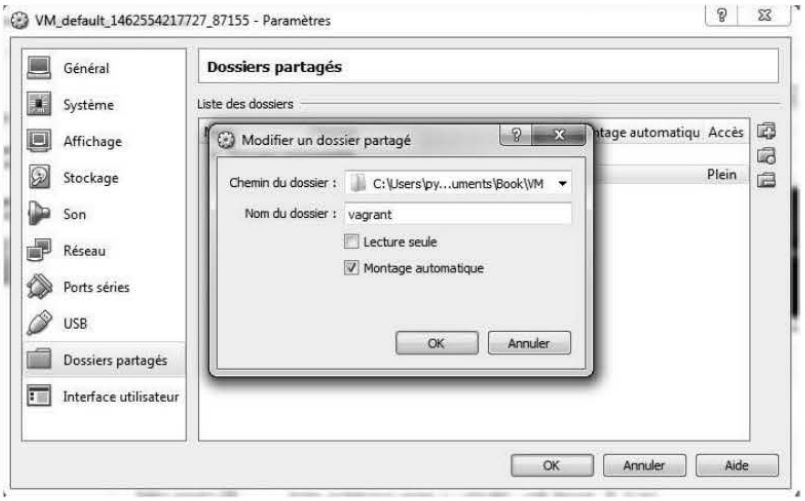

Figure 3.6 — Auto-montage du partage de fichiers

Au démarrage de votre VM, un lien sera créé entre le répertoire dans lequel vous avez lancé la commande `vagrant up` et le chemin /media/sf_vagrant dans votre VM.

> Attention cependant, depuis l'intérieur de la VM, le chemin /media/sf_vagrant ne sera accessible qu'à partir de l'utilisateur root ou d'un utilisateur disposant des droits *sudo*, ce qui est le cas de notre utilisateur vagrant.

Maintenant que nous disposons d'une machine virtuelle Linux, il ne nous reste plus qu'à installer le Docker Engine.

Installation de Docker dans notre VM

Docker est disponible via les différents gestionnaires de packages des distributions Linux (aptitude pour Ubuntu/Debian, yum pour RedHat/CentOS, dnf pour Fedora)

Pour obtenir la dernière version de Docker, il est généralement conseillé d'utiliser le dépôt de packages proposé par Docker.

Vous souhaitez une installation particulière de Docker ou tout simplement plus d'information ? N'hésitez pas à consulter la documentation d'installation fournie par Docker :
https://docs.docker.com/installation

La distribution Linux sur laquelle vous voulez installer Docker doit remplir les prérequis suivants pour être prise en charge :

- une architecture 64 bits, Docker n'étant pas supporté en 32 bits ;
- un kernel Linux récent (au moins en version 3.10), même s'il est possible de trouver des versions depuis la version 2.6 sur lesquelles Docker fonctionne.

C'est évidemment le cas de notre VM CentOS 7.

L'installation de Docker se fait simplement via le gestionnaire de package yum :

- connectez-vous sur votre machine avec un user qui dispose des droits sudo ou directement avec root. Dans notre VM, l'utilisateur *vagrant* disposant des droits sudo, il suffit donc de saisir la commande suivante pour disposer des droits root :

```
$ sudo su
```

- rajoutez le dépôt docker pour yum (cela permet d'obtenir la dernière version de docker, étant donné que le dépôt central CentOS met du temps à être mis à jour) :

```
$ tee /etc/yum.repos.d/docker.repo <<-'EOF'
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
EOF
```

- installez le package Docker :

```
$ yum install docker-engine-1.11.2
```

Si vous omettez le numéro de version, la version la plus récente de Docker sera installée, mais il est possible que certains exemples ne fonctionnent plus. Nous indiquerons sur le site du livre pour chaque version ultérieure de Docker les différences éventuelles dont il faudra tenir compte.

Il ne reste plus qu'à démarrer le démon Docker :

```
$ systemctl start docker[1]
```

Pour faire en sorte que Docker démarre automatiquement au démarrage de la VM, il suffit de lancer la commande suivante :

```
$ systemctl enable docker
```

Vous pouvez maintenant vérifier que tout fonctionne correctement grâce au conteneur de test hello-world :

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world b901d36b6f2f:
Pull complete  0a6ba66e537a:
Pull complete
Digest:sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
Status: Downloaded newer image for hello-world:latest

Hello from Docker. This message shows that your installation appears to be
working correctly.
```

Finalement, pour éviter de devoir préfixer toutes les commandes Docker par un sudo, il est possible de créer un groupe docker et d'y rajouter votre utilisateur (dans notre cas, vagrant) :

```
$ usermod -aG docker vagrant
```

Tous les utilisateurs du groupe docker bénéficient du droit de se connecter au démon Docker tournant sur la machine.

Attention, le fait d'appartenir au groupe *docker* est hautement sensible. Docker dispose des droits root sur l'hôte. Exploité malicieusement, le droit de lancer des conteneurs Docker peut faire de très gros dégâts.

Un redémarrage de la machine virtuelle est nécessaire pour que la modification soit prise en compte (ou du moins un logout de l'utilisateur courant).

Installation automatisée

Docker propose aussi un script pour l'installation. Ce script détecte votre distribution Linux et réalise les opérations nécessaires à l'installation de la dernière version de Docker :

```
curl -sSL https://get.docker.com/ | sh
```

Il ne reste ensuite plus qu'à démarrer le service Docker comme cela est expliqué ci-dessus.

1. CentOS 7 utilise maintenant systemd comme système d'initialisation. https://wiki.freedesktop.org/www/Software/systemd/ pour en savoir plus.

3.2 VOTRE PREMIER CONTENEUR

Maintenant que Docker est installé et fonctionnel, il est temps de démarrer nos premiers conteneurs.

Il existe plusieurs façons d'interagir avec le démon Docker :

- via Kinematic (uniquement sous Windows et Mac OS X) ;
- via le client Docker, le cas le plus commun, que nous privilégierons dans la suite de cet ouvrage ;
- directement avec des appels HTTP en utilisant l'API Docker Remote.

3.2.1 Kitematic

Kitematic est un logiciel fourni avec Docker ToolBox (donc disponible uniquement sous Windows et Mac OS X) qui permet de gérer les conteneurs d'un démon Docker local. Il permet d'appréhender Docker visuellement sans utiliser la ligne de commande.

> Pour utiliser Kitematic, vous devez avoir un compte sur le Docker Hub[1]. Il n'y a pas à proprement parler de raison technique à cette obligation. Il s'agit pour la société Docker Inc. de promouvoir son Hub qui joue un rôle essentiel dans l'écosystème Docker.

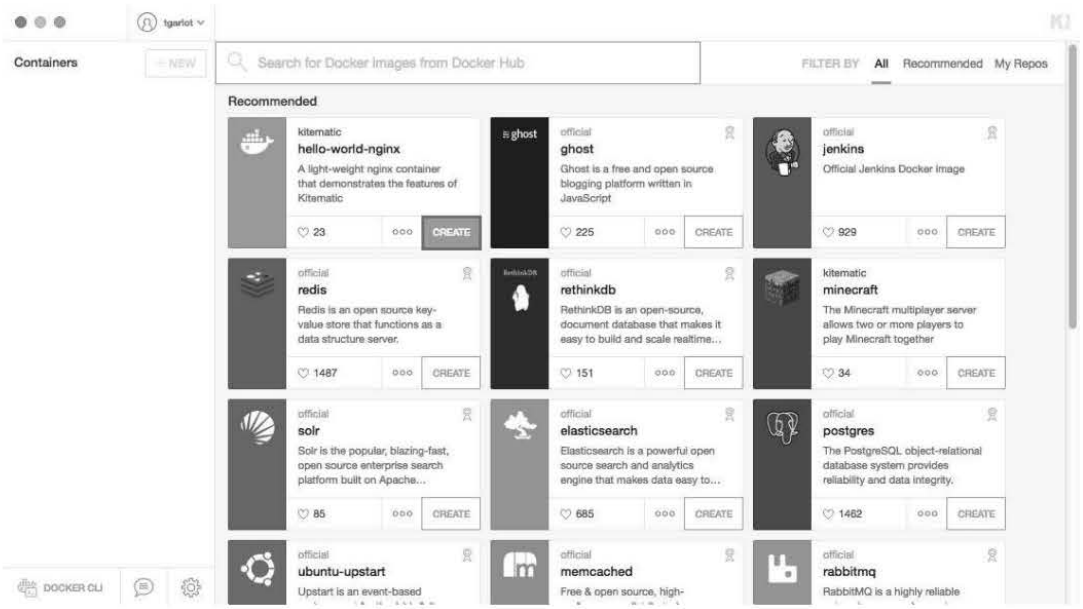

Figure 3.7 — Kitematic : Docker en mode graphique

Pour prendre en main Kitematic, cliquez sur le bouton *Create* de l'image *hello-world-nginx*. Kitematic va alors télécharger l'image correspondante depuis le Docker Hub, puis créer et démarrer un conteneur à partir de cette image.

1. https://hub.docker.com/

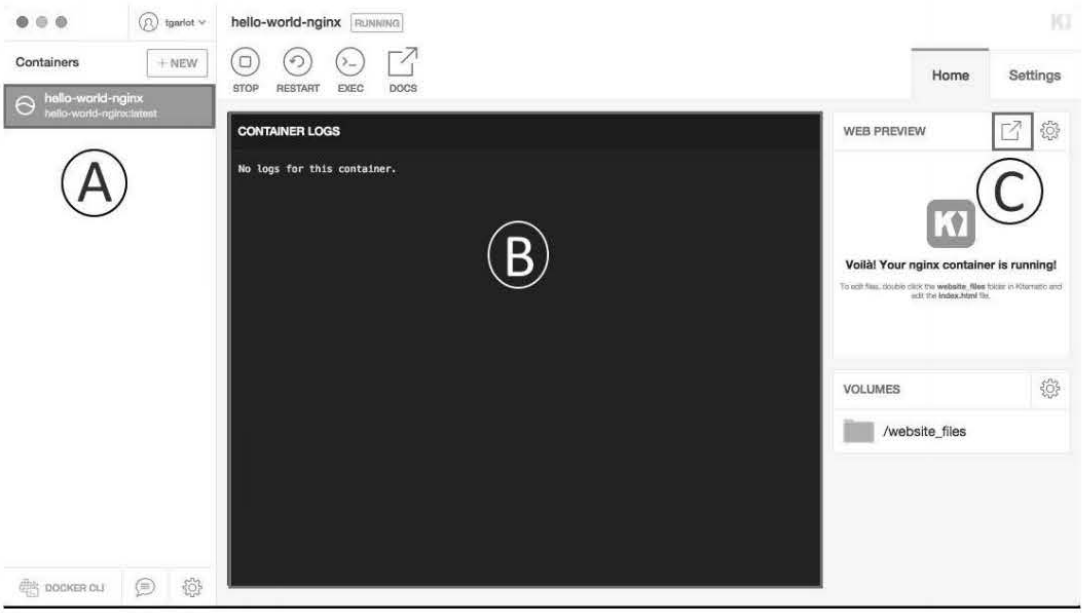

Figure 3.8 — Démarrage d'un conteneur Nginx dans Kitematic

Kitematic liste tous les conteneurs existants arrêtés ou actifs (ce qui est indiqué par la marque « A » sur la figure 3.8). On voit dans notre cas le conteneur que nous venons de créer.

> Kitematic assigne comme nom par défaut le nom de l'image. S'il y en a plusieurs, elles sont suffixées par _1, _2...

Kitematic permet de visualiser directement les logs du conteneur (marque « B » sur la figure 3.8). Nous verrons qu'il s'agit là d'une fonction standard d'un conteneur Docker.

Si le conteneur expose un port web, tel que 80 ou 8000 (nous verrons plus tard ce que cela signifie), Kitematic le rend disponible directement sur l'hôte. N'oubliez pas que le conteneur s'exécute dans une machine virtuelle Linux gérée par VirtualBox. En cliquant sur le lien *web preview* (marque « C » sur la figure 3.8), un navigateur s'ouvre et vous pouvez voir la page d'accueil de Nginx.

Figure 3.9 — Page d'accueil de Nginx du conteneur hello-world-nginx

Nous en avons terminé avec cette introduction rapide à Kitematic et vous trouverez des informations complémentaires dans le guide d'utilisateur de Docker[1].

Nous allons maintenant étudier l'utilisation du client Docker en ligne de commande, qui reste la méthode principale d'interaction avec le démon Docker, du moins dans un cadre professionnel.

3.2.2 Le client Docker

Vous avez déjà utilisé le client Docker, peut-être sans le savoir, durant la phase d'installation, lorsque vous avez saisi la commande `docker run hello-world`.

Avant d'aller plus loin dans l'explication de son utilisation, il est temps de regarder un peu plus en détail comment le démon Docker interagit avec le reste du monde.

Le démon Docker écoute sur un socket Unix[2] à l'adresse /var/run/docker.sock. Le client Docker utilise HTTP pour envoyer des commandes à ce socket qui sont ensuite transmises au démon, aussi via HTTP.

Figure 3.10 — Communication client/serveur Docker

Il est possible de matérialiser cette communication simplement en sniffant le trafic HTTP entre le client et le démon. Pour cela, nous allons utiliser Socat[3].

> Socat est une boîte à outils qui permet de spécifier deux flux de données (des fichiers, des connexions...) et de transférer les données de l'un vers l'autre.

La figure ci-contre permet de comprendre ce que nous allons faire. Nous insérons un *socket proxy*, qui va d'une part transférer les commandes du client au démon Docker, et d'autre part, les afficher dans notre fenêtre de terminal.

Installez *socat* à l'aide du gestionnaire de paquet yum :

```
$ sudo yum install -y socat
```

1. https://docs.docker.com/kitematic/userguide/
2. https://fr.wikipedia.org/wiki/Berkeley_sockets#Socket_unix
3. http://www.dest-unreach.org/socat/

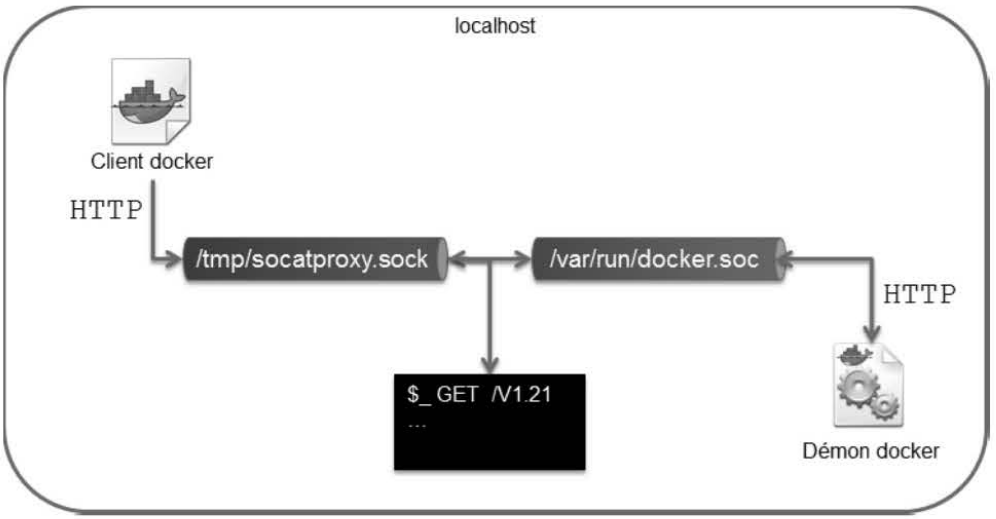

Figure 3.11 — Interception de communication avec socat

Lancez ensuite simplement la commande suivante :

```
$ socat -v UNIX-LISTEN:/tmp/socatproxy.sock,fork,reuseaddr
UNIX-CONNECT:/var/run/docker.sock &
```

Et maintenant, utilisons le client Docker pour lister tous les conteneurs de notre système :

```
$ docker -H unix:///tmp/socatproxy.sock ps -a
> 2016/01/10 10:41:53.862867  length=114 from=0 to=113
GET /v1.21/containers/json?all=1 HTTP/1.1\r
Host: /tmp/socatproxy.sock\r
User-Agent: Docker-Client/1.9.1 (linux)\r
\r
< 2016/01/10 10:41:53.864414  length=477 from=0 to=476
HTTP/1.1 200 OK\r
Content-Type: application/json\r
Server: Docker/1.9.1 (linux)\r
Date: Sun, 10 Jan 2016 09:41:53 GMT\r
Content-Length: 338\r
\r
[{"Id":"885d2f6583979e20b5b8a8f285ba0a459fda4dacf59a1c42790e53cd57c18f7b","Names":["/evil_
world","ImageID":"0a6ba66e537a53a5ea94f7c6a99c534c6adb12e3ed09326d4bf3b38f7c3ba4e7","Commar
(0) 4 minutes ago","HostConfig":{"NetworkMode":"default"}}]
CONTAINER ID   IMAGE          COMMAND      CREATED  -- Répo
5d2f658397     hello-world    "/hello"     4 minutes ago
STATUS                     PORTS       NAMES
Exited (0) 4 minutes ago               evil_fermat
```

Regardons plus en détail :

Paramètres	Description
docker -H unix:///tmp/socatproxy.sock ps -a	Commande pour laquelle nous voulons voir la requête et la réponse HTTP. Notez le paramètre `-H` qui permet de préciser où envoyer la requête, dans notre cas le socket proxy socat.
GET /v1.21/containers/json?all=1 HTTP/1.1 ...	La première partie du log socat. Notre commande est convertie en requête HTTP par le client Docker (dans notre cas un appel GET pour obtenir la liste des conteneurs).
HTTP/1.1 200 OK ... [{"Id":"885d2f6583979e20b5b8a8f285ba0a459fda4dacf59a1c42790e53cd57c18f7b",...	La seconde partie du log socat. Il s'agit de la réponse HTTP retournée par le démon au format JSON.
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES 885d2f658397 hello-world "/hello" 4 minutes ago Exited (0) 4 minutes ago evil_fermat	La réponse interprétée en mode texte par le client Docker. Sans le *socket socat*, seul ce texte aurait été affiché.

Si le client Docker ne fait que des appels HTTP, pouvons-nous nous substituer à lui et appeler le démon directement ?

Bien sûr ! Et c'est exactement ce que nous allons faire dans le prochain chapitre.

Le groupe docker

Souvenez-vous, lors de l'installation de Docker, nous avions rajouté notre utilisateur *vagrant* dans un groupe *docker*.
Le client et le démon Docker ont besoin de tourner avec des droits de type root (cela est nécessaire pour pouvoir, par exemple, monter des volumes). Si un groupe *docker* existe, Docker va donner les permissions sur le socket /var/run/docker.sock à ce groupe. Tout utilisateur en faisant partie pourra donc ensuite utiliser Docker sans la commande sudo.

3.2.3 API Docker Remote

Le démon Docker expose l'intégralité de ses méthodes à travers l'API Docker Remote[1]. C'est une API de type REST qu'il est possible d'appeler avec des outils tels que *curl* ou *wget*, à partir du moment où la configuration le permet.

Docker info avec curl

Rappelons que Docker n'écoute que sur un socket Unix à l'adresse /var/run/docker.sock, comme nous l'avons vu précédemment. Pour pouvoir appeler l'API directement avec un client HTTP, il faut que nous définissions un autre socket de connexion de type TCP.

Pour cela, arrêtons le démon :

1. https://docs.docker.com/engine/reference/api/docker_remote_api/

```
$ sudo systemctl stop docker
```

Et redémarrons-le à la main (c'est-à-dire sans utiliser systemd) en précisant l'IP et le port sur lequel celui-ci doit désormais écouter :

```
$ sudo docker daemon -H tcp://0.0.0.0:2375 &
[1] 15360
[vagrant@localhost docker.service.d]$ WARN[0000] /!\ DON'T BIND ON ANY IP
ADDRESS WITHOUT setting -tlsverify IF YOU DON'T KNOW WHAT YOU'RE DOING /!\
INFO[0000] API listen on [::]:2375
WARN[0000] Usage of loopback devices is strongly discouraged for production
use. Please use `--storage-opt dm.thinpooldev` or use `man docker` to refer to
dm.thinpooldev section.
INFO[0000] [graphdriver] using prior storage driver "devicemapper"
INFO[0000] Firewalld running: false
INFO[0000] Default bridge (docker0) is assigned with an IP address
172.17.0.1/16. Daemon option --bip can be used to set a preferred IP address
INFO[0000] Loading containers: start.
...
INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon     commit=a34a1d5 execdriver=native-0.2
graphdriver=devicemapper version=1.9.1
```

Regardons maintenant en détail le résultat :

Paramètre	Description
sudo docker daemon -H tcp://0.0.0.0:2375 &	Démarre le démon Docker et le lie à toutes les adresses IP de notre machine hôte (0.0.0.0) en utilisant le port 2375.
WARN[0000] /!\ DON'T BIND ON ANY IP ADDRESS WITHOUT setting -tlsverify IF YOU DON'T KNOW WHAT YOU'RE DOING /!\	Docker nous prévient que cette connexion n'est pas sécurisée. N'importe qui peut donc appeler Docker depuis une machine distante. Dans le cas d'un déploiement réel, il serait nécessaire de sécuriser l'accès à cette interface à l'aide de SSL et d'un moyen d'authentification.
WARN[0000] Usage of loopback devices is strongly discouraged for production use. Please use '–storage-opt dm.thinpooldev' or use 'man docker' to refer to dm.thinpooldev section.	Il s'agit d'un message d'avertissement de Docker qui nous signale que le démon est notamment lié à l'interface réseau loopback du serveur et qu'il s'agit, en général, d'une très mauvaise idée pour un serveur de production. Nous nous en accommoderons à ce stade de l'ouvrage.
INFO[0000] [graphdriver] using prior storage driver "devicemapper"	Nous utilisons le device storage *devicemapper*. Il s'agit là, comme nous l'avons expliqué dans le chapitre 1, du gestionnaire de persistance utilisé par Docker (dans le cas de CentOS).

Nous pouvons vérifier maintenant que Docker n'est plus accessible sur le socket Unix /var/run/docker.sock (qui n'est d'ailleurs pas créé) :

```
$ grep -f /var/run/docker.sock
grep: /var/run/docker.sock: Aucun fichier ou dossier de ce type
$ docker ps
Cannot connect to the Docker daemon. Is the docker daemon running on this
host?
```

Pour se connecter à notre démon, il faut maintenant passer l'adresse que nous venons de définir avec le paramètre `-H` :

```
$ docker -H 0.0.0.0:2375 info
INFO[4403] GET /v1.21/info
Containers: 1
Images: 2 Server
Version: 1.9.1
Storage Driver: devicemapper
  Pool Name: docker-253:0-69821582-pool
  Pool Blocksize: 65.54 kB
  Base Device Size: 107.4 GB
...
```

127.0.0.1 vs 0.0.0.0

Nous sommes tous plus ou moins familiers avec l'IP 127.0.0.1 : c'est l'adresse IP de loopback, aussi connue sous le nom de localhost. Cette adresse est associée à une interface réseau virtuelle qui ne permet de communiquer qu'avec l'hôte lui-même (elle n'a donc de sens que pour les processus tournant sur cet hôte et n'est pas accessible depuis l'extérieur).
L'adresse 0.0.0.0 est une autre adresse standard. Elle signifie « toutes les adresses IPv4 de la machine » (incluant donc 127.0.0.1, d'où le message d'alerte faisant référence au loopback qui s'est affiché lorsque nous avons démarré le démon). Le démon ainsi associé à l'adresse 0.0.0.0 est donc accessible depuis n'importe quelle adresse (ce qui n'est généralement pas le cas dans un environnement sécurisé de production).

Si nous ne voulons pas avoir à spécifier à chaque appel le paramètre `-H`, il suffit de définir la variable d'environnement DOCKER_HOST et ainsi le client Docker l'utilisera automatiquement :

```
$ export DOCKER_HOST="tcp://0.0.0.0:2375"
```

Utilisons maintenant *curl* pour appeler directement le démon Docker. Pour rendre la réponse JSON plus lisible, nous allons employer *jq*[1] , et il faut donc installer le paquet nécessaire :

```
sudo yum install -y jq
```

1. jq est un processeur de JSON en ligne de commande. Plus d'info à https://stedolan.github.io/jq/

Maintenant que l'utilitaire est installé, lançons la commande `docker info` à l'aide de *curl* :

```
$ curl http://localhost:2375/info |jq
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
Dload  Upload   Total   Spent    Left  Speed   0     0    0     0    0     0
0      0 --:--:-- --:--:-- --:--:--     0INFO[1260] GET /info
100  1859  100  1859    0     0  11776      0 --:--:-- --:--:-- --:--:-- 11840
{   "ID": "MDLV:VVHJ:GMIP:VRXK:QOVB:7FQS:27NJ:YRR3:5B4V:WOAV:U6IX:KSFN",
"Containers": 1,   "Images": 2,   "Driver": "devicemapper",   "DriverStatus":
[     [       "Pool Name",       "docker-253:0-69821582-pool"      ], ...
```

Nous vous conseillons de redémarrer votre VM si vous avez réalisé l'exercice précédent pour ne pas créer de conflit avec les exercices qui suivent. Sinon, vous pouvez aussi appliquer la commande unix `kill` sur un processus créé par `sudo docker daemon -H tcp://0.0.0.0:2375 &`.

Modifier la configuration du démon Docker

Au prochain redémarrage de votre machine, Docker va reprendre son ancienne configuration. Il n'écoutera donc plus sur le socket TCP et le port que nous avons défini. Pour rendre ce paramétrage permanent, il est possible de modifier la configuration du démon au moyen du fichier /etc/docker/daemon.json.

Ce fichier de configuration permet de modifier les paramètres de démarrage du démon sans être contraint de le lancer manuellement. Nous pouvons, par exemple, créer un fichier /etc/docker/daemon.json avec pour contenu :

```
$ sudo tee /etc/docker/daemon.json <<-'EOF'
{
  "hosts" : ["fd://","tcp://0.0.0.0:2375"]
}
EOF
```

Notre démon écoutera maintenant sur deux sockets :

- le socket standard (/var/run/docker.sock) ;
- le socket tcp que nous avons configuré ci-dessus.

Malheureusement, sous CentOS (ou plus exactement sous un système utilisant systemd), une petite manipulation additionnelle est nécessaire. À l'heure où nous écrivons ces lignes, sous CentOS, la configuration de démarrage de Docker (présente dans le fichier /usr/lib/systemd/system/docker.service) s'appuie sur l'instruction :

```
ExecStart=/usr/bin/docker daemon -H fd://
```

Or il n'est pas possible de configurer le même paramètre à la fois via le fichier de configuration *daemon.json* et par l'intermédiaire d'un flag (`-H` dans ce cas). Il est donc

nécessaire de modifier l'instruction de démarrage par défaut du démon Docker via les instructions suivantes :

```
$ sudo mkdir /etc/systemd/system/docker.service.d
$ sudo tee /etc/systemd/system/docker.service.d/startup.conf <<-'EOF'
[Service]
ExecStart=
ExecStart=/usr/bin/docker daemon
EOF
```

Le lecteur curieux pourra se reporter à la documentation Docker[1] relative à *systemd* qui aborde cette problématique en détail.

On recharge la configuration du service Docker et on démarre le démon :

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

Nous disposons maintenant d'une d'installation du Docker Engine totalement fonctionnelle et prête à être expérimentée.

En résumé

Dans ce chapitre nous avons appris à installer Docker sur une machine de bureau. Nous avons appris à interagir avec le démon Docker au travers de divers outils pour lancer des commandes et créer des conteneurs. Dans notre prochain chapitre, nous allons nous intéresser à l'installation de Docker sur un hôte serveur de manière plus professionnelle.

1. https://docs.docker.com/engine/admin/systemd

4

Docker sur un serveur

Installer Docker sur un hôte

Avant d'entamer la mise en pratique de Docker, nous avons trouvé utile d'aborder la question de la création des hôtes Docker. Dans le chapitre 3, nous avons vu comment installer Docker sur une machine de bureau (afin d'apprendre et d'expérimenter), et l'objectif de ce chapitre est de voir comment installer Docker sur une machine de type serveur. Nous utiliserons pour ce faire différents types d'outils et modes d'installation.

L'installation de Docker peut se faire de plusieurs manières :

- utiliser Docker Machine proposé par Docker Inc. que nous avons déjà expérimenté (sans vraiment le savoir) dans le précédent chapitre avec Docker ToolBox ;
- utiliser des distributions Linux spécialisées (déjà évoquées dans le chapitre 1) qui permettent de créer simplement des images Docker « ready » ;
- finalement, on peut s'appuyer sur des offres cloud de type CaaS (présentées dans le chapitre 2) qui rendent transparentes la configuration et même la nature physique de l'hôte.

Dans ce chapitre, nous allons aborder deux exemples d'installation :

- Docker Machine, qui permet de construire des hôtes Docker sur un très large panel d'environnements et de configurations ;
- Atomic Host, une distribution spécialisée (au même titre que CoreOS ou Boot2Docker) élaborée par le projet Fedora.

4.1 DOCKER MACHINE

Docker Machine permet d'installer et de configurer le Docker Engine sur des machines virtuelles (existantes ou non) aussi bien sur votre ordinateur (Windows ou Mac) que sur des fournisseurs cloud, comme Amazon AWS ou Microsoft Azure. Vous pourrez ensuite piloter ces hôtes de manière très simple, directement avec la commande **docker-machine**.

4.1.1 Installation de Docker Machine

Avant tout, Docker Machine requiert que le Docker Engine soit installé sur votre ordinateur. Docker Machine est l'un des composants de la Docker Toolbox. Vous pouvez vous reporter au chapitre précédent pour plus de détails concernant l'installation de cette boîte à outils.

Il est aussi possible de télécharger le binaire de Docker Machine manuellement et de l'extraire en adaptant les permissions pour le rendre exécutable :

```
# Sous Mac OS X ou Linux - nécessite des droits root
$ curl -L https://github.com/docker/machine/releases/download/v0.6.0/docker-
machine-‘uname -s‘-‘uname -m‘ > /usr/local/bin/docker-machine &&
\
chmod +x /usr/local/bin/docker-machine

# Sous Windows (nécessite git bash[1])
if [[ ! -d "$HOME/bin" ]]; then mkdir -p "$HOME/bin"; fi && \
curl -L https://github.com/docker/machine/releases/download/v0.6.0/docker-
machine-Windows-x86_64.exe > "$HOME/bin/docker-machine.exe" &&
\
chmod +x "$HOME/bin/docker-machine.exe"
```

Nous avons installé ici la dernière version disponible de Docker Machine (0.6.0) au moment de l'écriture de ce livre. Vérifiez et utilisez toujours la dernière version disponible en mettant à jour l'URL d'installation dans les commandes ci-dessus.

Vérifions finalement que l'installation s'est bien déroulée en affichant la version de Docker Machine :

```
$ docker-machine version
 docker-machine version 0.6.0, build e27fb87
```

Explorons maintenant les capacités de Docker Machine pour bien comprendre son fonctionnement.

1. https://git-for-windows.github.io/

Script Bash completion de Docker machine

Les commandes de Docker Machine requièrent en général de nombreux paramètres et sont globalement longues à saisir. Pour simplifier leur utilisation, il existe un ensemble de scripts *bash* qui permettent, à la façon d'une ligne de commande, de compléter les commandes Docker Machine. Leur installation est simple et si vous avez une utilisation intensive, nous ne pouvons que vous encourager à les utiliser[1].

4.1.2 Prise en main de Docker Machine

Il faut vraiment voir Docker Machine comme un outil de pilotage distant d'hôtes avec Docker, que ce soit en local, dans votre cloud d'entreprise ou sur un fournisseur cloud (comme Amazon AWS ou Microsoft Azure).

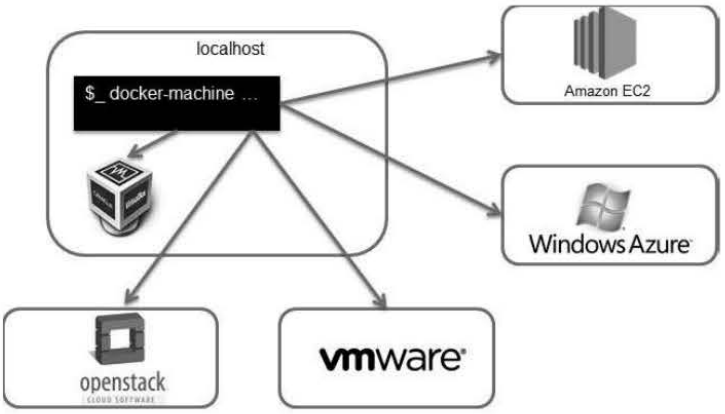

Figure 4.1 — Les différents drivers de Docker Machine

Docker Machine repose pour cela sur un mécanisme de drivers. Chaque driver est spécifique d'un type d'installation et vous permet de créer une machine Docker pour un environnement spécifique.

> Dans la suite de ce chapitre, nous utiliserons de manière interchangeable le terme machine et hôte, une machine n'étant finalement qu'un hôte sur lequel tourne un démon Docker.

On peut catégoriser ces drivers de la manière suivante :

- ceux qui permettent de créer une machine localement sur votre poste de travail. Dans ce cas, Docker Machine interagit directement avec le programme de virtualisation, tel qu'Oracle VirtualBox, Microsoft Hyper-V ou VMware Fusion ;
- ceux qui gèrent des machines sur une infrastructure privée d'entreprise : OpenStack ou VMware vSphere par exemple ;
- et ceux qui utilisent des infrastructures cloud, telles que Digital Ocean, Google Compute Engine ou Microsoft Azure.

1. https://github.com/docker/machine/tree/master/contrib/completion/bash

La liste des drivers officiellement supportés par Docker Machine évoluant régulièrement, il est plus simple de se référer au site de Docker pour disposer de la liste exhaustive. L'architecture modulaire de Docker Machine a aussi permis à des tierces parties de créer des drivers supplémentaires (tel KVM[1]) dont la liste se trouve sur le dépôt GitHub de Docker Machine[2].

Regardons maintenant plus en détail la commande `docker-machine create`.

4.1.3 La commande docker-machine create

Comme son nom l'indique, la commande `docker-machine create` permet de créer des machines avec un Docker Engine et de configurer directement ce dernier. Elle nécessite pour cela au moins deux paramètres :

- `--driver` : comme nous l'avons vu précédemment, c'est le nom du driver qui sera utilisé, par exemple virtualbox ou digitalocean ;
- le nom de la machine, à donner généralement en fin de commande.

Il y a bien sûr de nombreux autres paramètres que l'on peut lister avec la commande suivante :

```
$ docker-machine create -help
```

De plus, pour obtenir de l'aide sur un driver particulier, il suffit de rajouter son nom, par exemple, pour le driver Virtualbox :

```
$ docker-machine create --driver virtualbox --help
```

Ces paramètres peuvent être regroupés en trois catégories :

- Ceux commençant par `--engine` : ils permettent de configurer directement le Docker Engine lors de la création de la machine. Par exemple, le paramètre `engine-insecure-registry` configure un registre Docker privé que le Docker Engine pourra utiliser.
- Ceux commençant par `--swarm` : comme nous le verrons dans la troisième partie de cet ouvrage, il est possible de créer et de configurer directement un cluster Swarm avec Docker Machine. Par exemple, le paramètre `--swarm-discovery` configure le service de découverte du cluster Swarm (typiquement l'URL d'un registre Consul ou etcd).
- Ceux qui sont spécifiques à chaque driver : ils sont préfixés par le nom du driver (*virtualbox* ou *amazonec2*, par exemple). Nous verrons dans le chapitre suivant comment les utiliser dans le cas d'une machine Amazon AWS.

Docker Machine utilise pour certains de ces paramètres des valeurs par défaut. Par exemple, si vous créez une machine sur le cloud Microsoft Azure, la valeur par défaut du paramètre `azure-size`, qui définit la taille de la machine, est Small.

1. http://www.linux-kvm.org/page/Main_Page
2. https://github.com/docker/machine/blob/master/docs/AVAILABLE_DRIVER_PLUGINS.md

Nous en savons maintenant assez pour créer nos premières machines. Commençons par une machine sur le cloud d'Amazon AWS.

4.1.4 Installation de Docker avec Docker Machine sur Amazon AWS

Avant tout, il faut évidemment que vous disposiez d'un compte Amazon AWS.

Si ce n'est pas le cas, rendez-vous sur https://aws.amazon.com/ pour en créer un. Vous disposez automatiquement de l'offre AWS Free Tier qui inclut une machine virtuelle avec un abonnement t2.micro pour une durée d'un an qui vous permettra d'expérimenter Docker Machine.

Attention, l'offre AWS Free Tiers n'inclut pas une adresse IP statique pour la machine virtuelle, ce qui signifie que l'adresse IP va changer et que les certificats TLS générés par Docker Machine pour sécuriser le Docker Engine ne seront plus valables. La solution est d'activer une Elastic IP qui permet ainsi d'obtenir une adresse IP statique, mais il vous en coûtera un peu d'argent.

Le driver Docker Machine Amazon attend deux paramètres obligatoires pour la création de machines virtuelles EC2 :

- un AWS Access Key ID ;
- un AWS Secret Access Key.

Pour obtenir ces deux valeurs, il faut se connecter à la console Amazon AWS.

Les prérequis

Nous allons tout d'abord créer un groupe avec des droits d'administration, puis un utilisateur de ce groupe. Finalement, nous créerons pour cet utilisateur les clés d'accès requises par Docker Machine.

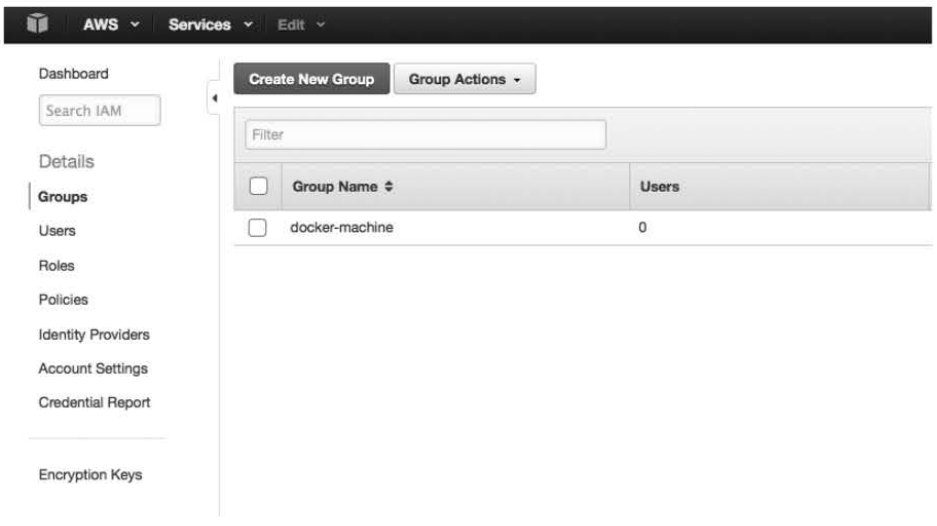

Figure 4.2 — Création d'un groupe de sécurité

Commençons par créer le group *docker-machine* :

- depuis le menu principal, sélectionnez *Services* puis *IAM* (Identity and Access Management) ;
- dans le menu de gauche, sélectionnez *Groups* puis *Create New Group* ;
- choisissez *docker-machine* comme nom de groupe puis rajoutez-lui la permission *AdministratorAccess* ;
- confirmez finalement la création du groupe.

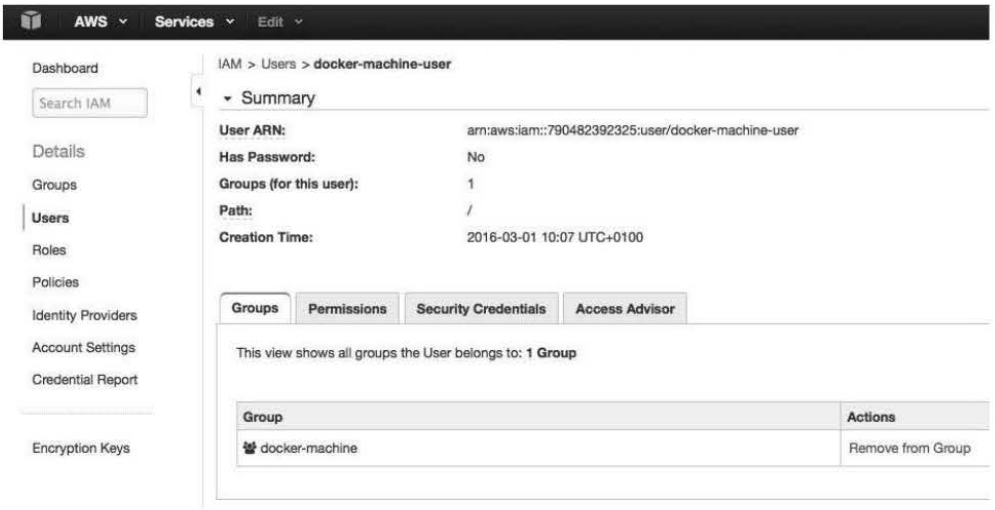

Figure 4.3 — Association d'un utilisateur au groupe docker-machine

Ajoutons maintenant un utilisateur :

- dans le menu de gauche, sélectionnez *Users*, puis *Create New Users* ;
- dans le champ 1, saisissez *docker-machine-user* et confirmez ;
- vous pouvez maintenant télécharger, via le bouton *Download credentials*, un fichier qui contient l'ID et la clé dont nous aurons besoin avec Docker Machine.

Gardez-le précieusement car c'est la seule fois où vous pouvez obtenir l'AWS Secret Access Key ! Si vous la perdez, vous ne pourrez plus vous servir de cette clé et vous devrez en créer une autre.

Maintenant que nous disposons de notre utilisateur, ajoutons-le dans le groupe *docker-machine* que nous avons précédemment créé et attribuons-lui un mot de passe :

- dans le menu de gauche, sélectionnez *Users*, puis l'utilisateur *docker-machine-user* ;
- allez dans l'onglet *Groups* et rajoutez-le au groupe *docker-machine*.

Il ne nous reste plus qu'à définir un mot de passe pour notre utilisateur :

- allez dans l'onglet *Security Credentials* et cliquez sur *Manage Password* ;
- choisissez *Assign a custom password*, saisissez un mot de passe deux fois et confirmez.

Création de la machine EC2

Nous disposons maintenant de tout ce dont nous avons besoin pour créer notre première Docker Machine *aws-machine* avec le driver *amazonec2*.

Ouvrez un terminal et saisissez la commande suivante en remplaçant les paramètres `amazonec2-access-key` et `amazonec2-secret-key` par ceux que vous avez obtenus lors de la création de l'utilisateur docker-machine-user :

```
$ docker-machine create --driver amazonec2 --amazonec2-access-key AKIXXXX
--amazonec2-secret-key YYYYYY  aws-machine
Running pre-create checks...
Creating machine...
(aws-machine) Launching instance...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with ubuntu(systemd)...
Installing Docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this
virtual machine, run: docker-machine env aws-machine
```

Via la console AWS, nous pouvons voir qu'une nouvelle machine virtuelle a bien été créée :

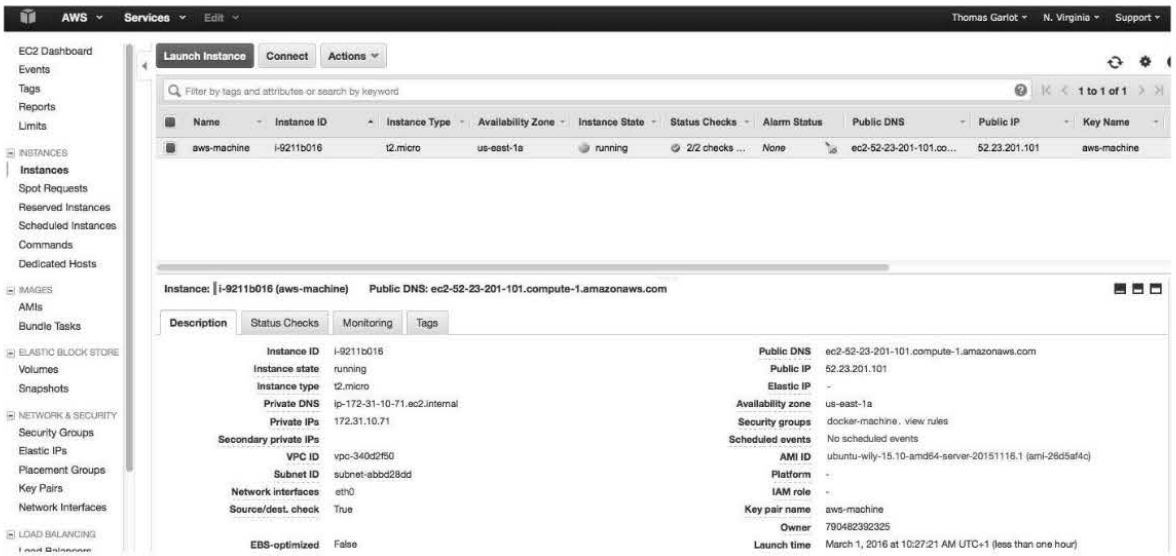

Figure 4.4 — Machine virtuelle EC2 provisionnée par Docker Machine

Détaillons un peu ce qui s'est passé :

- Docker Machine s'est connecté à l'interface d'Amazon EC2 et a lancé la création d'une machine virtuelle avec le nom *aws-machine* ;

- cette machine est localisée par défaut dans la région géographique us-east-1 que nous aurions pu modifier, via le paramètre `amazonec2-region` ;
- cette machine est basée sur l'AMI (modèle de machine virtuelle Amazon) ami-26d5af4c qui repose sur une Ubuntu 15.10. De nouveau, il s'agit d'une valeur par défaut que nous aurions pu surcharger avec le paramètre `amazonec2-ami` en utilisant, par exemple, une AMI personnalisée.

Si vous désirez en savoir plus, la liste des valeurs par défaut est disponible sur la page de documentation du driver Amazon de Docker Machine[1].

En plus de créer la machine virtuelle EC2, Docker Machine a aussi configuré un groupe de sécurité (*security group*). Il s'agit tout simplement des règles de pare-feu (*firewall*) qui ouvre les connexions suivantes :

- port 22/TCP pour les connexions SSH ;
- port 2376/TCP pour exposer le démon Docker ;
- en option, le port 3376/TCP peut être ouvert si nous passons le paramètre `swarm-master`.

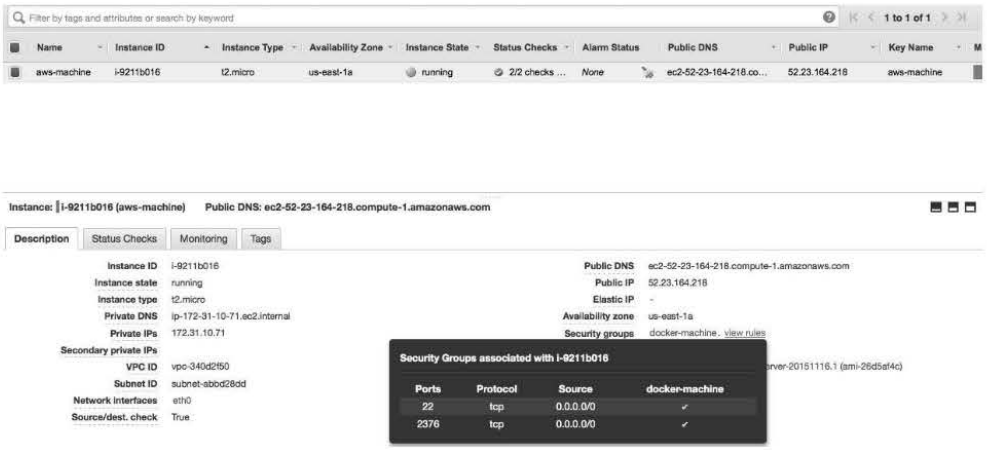

Figure 4.5 — Ports ouverts par Docker Machine

Pour sécuriser la connexion SSH, Docker Machine génère une clé RSA. La partie publique de la clé est ensuite copiée sur notre machine virtuelle (on la retrouve d'ailleurs via la console AWS sous le menu Network & Security, puis Key Pairs).

Données de configuration locales

Lorsque vous utilisez Docker Machine depuis un hôte, un certain nombre de fichiers de configuration sont créés et conservés.

Par exemple, dans le cas de notre machine EC2 :

1. https://docs.docker.com/machine/drivers/aws/

```
$ cd ~/.docker/machine/machines/aws-machine/
[aws-machine]$ ls -l
ca.pem
cert.pem
config.json
id_rsa
id_rsa.pub
key.pem
server-key.pem
server.pem
```

On retrouve :

- notre clé SSH (id_rsa et id_rsa.pub) ;
- un fichier config.json qui contient les détails de la configuration de notre machine ;
- et un ensemble de certificats.

Ces certificats sont utilisés pour sécuriser l'accès à l'interface HTTP du démon Docker à l'aide de TLS, ce qui sera expliqué en détails dans le chapitre 8 sur la sécurisation du démon Docker.

Partage de la configuration Docker Machine

Pour partager la gestion d'une machine entre plusieurs utilisateurs, il suffit théoriquement de transférer le contenu du répertoire Docker Machine d'un poste sur un autre. Le souci est que le contenu du fichier *config.json* dépend de la configuration locale du poste initial ; par exemple, les chemins vers les différents certificats sont absolus, si bien que ce fichier est difficilement portable. Des discussions sont en cours sur le dépôt GitHub de Docker Machine pour résoudre ce problème.

L'approche de création de machines est relativement identique pour tous les providers cloud, chaque fournisseur ayant bien sûr des paramètres propres.

Étudions maintenant la création d'une machine locale à notre poste de travail à l'aide d'Oracle VirtualBox.

4.1.5 Installation de Docker avec Docker Machine et Oracle VirtualBox

La création d'une machine sur VirtualBox nécessite évidemment que VirtualBox soit installé sur votre poste de travail. C'est le cas si vous utilisez Docker ToolBox présenté dans le chapitre précédent.

Le processus de création est très similaire à celui que nous venons de suivre pour Amazon AWS : c'est d'ailleurs là un point fort de Docker Machine. Le principe des drivers permet de masquer simplement l'hétérogénéité des processus d'installation.

Créons donc notre machine *local-machine* avec le driver virtualbox :

```
$ docker-machine create --driver virtualbox local-machine
(local-machine) No default Boot2Docker ISO found locally, downloading the
latest release...
(local-machine) Latest release for github.com/boot2docker/boot2docker is
v1.10.2
(local-machine) Downloading /.docker/machine/cache/boot2docker.iso from
https://github.com/boot2docker/boot2docker/releases/download/v1.10.2/boot2docker.iso...
(local-machine)
0%....10%....20%....30%....40%....50%....60%....70%....80%....90%....100%
Creating machine...
(local-machine) Copying /.docker/machine/cache/boot2docker.iso to
/.docker/machine/machines/local-machine/boot2docker.iso...
(local-machine) Creating VirtualBox VM...
(local-machine) Creating SSH key...
(local-machine) Starting the VM...
(local-machine) Check network to re-create if needed...
(local-machine) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this
virtual machine, run: docker-machine env local-machine
```

Docker Machine se base ici sur une image boot2docker.iso, et non pas sur une image Ubuntu, comme pour Amazon AWS. Cette image est copiée dans le répertoire ~/.docker/cache pour accélérer la création ultérieure d'autres machines locales. Elle est ensuite copiée dans le répertoire de notre machine ~/.docker/machine/machines/local-machine.

À nouveau, Docker Machine utilise un ensemble de valeurs par défaut pour créer notre machine, chacune pouvant être surchargée[1]. Le reste du processus de création est identique à celui de notre machine Amazon EC2 :

- génération d'une clé RSA pour nous connecter en SSH ;
- copie des certificats nécessaires pour sécuriser avec TLS le démon Docker ;
- configuration du démon Docker avec des valeurs par défaut.

4.1.6 Quelques autres commandes utiles

Nous pouvons lister toutes les machines administrables depuis le poste de travail avec la commande `docker-machine ls` (l'utilisation du paramètre optionnel `--format` permet de personnaliser le détail des informations que nous voulons visualiser, dans notre cas le nom de la machine, son driver, son URL et la version de Docker[2]) :

1. https://docs.docker.com/machine/drivers/virtualbox/
2. https://docs.docker.com/machine/reference/ls/

```
$ docker-machine ls --format "{{.Name}}: {{.DriverName}} - {{.URL}} -
{{.DockerVersion}}"
aws-machine: amazonec2 - tcp://52.23.164.218:2376 - v1.10.2
local-machine: virtualbox - tcp://192.168.99.101:2376 - v1.10.2
```

Nous retrouvons bien nos deux machines.

Pour obtenir plus d'information sur notre machine, nous disposons de la commande `docker-machine inspect` (qui ne fait qu'imprimer le ficher *config.json* que l'on retrouve dans ~/.docker/machine/machines/local-machine) :

```
$ docker-machine inspect local-machine
{
    "ConfigVersion": 3,
    "Driver": {
        "IPAddress": "192.168.99.101",
        "MachineName": "local-machine",
        "SSHUser": "docker",
        "SSHPort": 49793,
...
```

Nous disposons aussi d'autres commandes permettant d'obtenir rapidement des informations sur notre machine :

```
$ docker-machine ip local-machine# IP de notre machine
192.168.99.101
$ docker-machine url local-machine# URL de l'hôte
tcp://192.168.99.101:2376
$ docker-machine status local-machine # État de la machine
Running
```

Utilisons maintenant notre machine pour démarrer un conteneur. Pour cela, nous devons avant tout définir un certain nombre de variables d'environnement pour indiquer à notre client Docker sur quelle machine il doit se connecter. Nous disposons pour cela de la commande suivante :

```
$ docker-machine env local-machine
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.101:2376"
export DOCKER_CERT_PATH="/Users/thomas/.docker/machine/machines/local-machine"
export DOCKER_MACHINE_NAME="local-machine"
# Run this command to configure your shell:
# eval $(docker-machine env local-machine)
```

Pour configurer votre environnement avec ces variables, utilisez la commande proposée et vérifiez ensuite avec la commande `docker-machine active` que notre machine *local-machine* est bien celle qui sera utilisée par la suite.

```
$ eval $(docker-machine env local-machine)
$ docker-machine active
local-machine
```

Par défaut, la commande `docker-machine env` génère un résultat pour un terminal Unix de type *bash*. Si vous utilisez Windows (Powershell ou cmd), il vous suffit de rajouter le paramètre `--shell` pour obtenir un résultat compatible avec votre environnement. Par exemple, pour Powershell :

```
$ docker-machine.exe env --shell powershell local-machine
$Env:DOCKER_TLS_VERIFY = "1"
$Env:DOCKER_HOST = "tcp://192.168.99.101:2376" $Env:DOCKER_CERT_PATH =
"C:\Users\thomas\.docker\machine\machines\local-machine"
$Env:DOCKER_MACHINE_NAME = " local-machine "
# Run this command to configure your shell:
# docker-machine.exe env --shell=powershell local-machine | Invoke-Expression
```

Toutes les commandes de notre client Docker utiliseront désormais le démon Docker tournant sur notre machine *local-machine*.

Démarrons notre premier conteneur :

```
$ docker run busybox echo Je tourne sur la machine local-machine
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
f810322bba2c: Pull complete
a3ed95caeb02: Pull complete
Digest:
sha256:97473e34e311e6c1b3f61f2a721d038d1e5eef17d98d1353a513007cf46ca6bd
Status: Downloaded newer image for busybox:latest
Je tourne sur la machine local-machine
$ docker ps -a
CONTAINER ID   IMAGE    COMMAND                  CREATED             STATUS
PORTS               NAMES
bdf68af495ea   busybox  "echo Je tourne sur l"   35 seconds ago      Exited
(0) 34 seconds ago  boring_mahavira
```

Nous pouvons même nous connecter en SSH pour vérifier que notre conteneur a bien été créé sur cette machine :

```
$ docker-machine ssh local-machine
     ##         .
               ## ## ##        ==
            ## ## ## ## ##    ===
        /"""""""""""""""""\___/ ===
   ~~~ {~~ ~~~~ ~~~ ~~~~ ~~~ ~ /  ===- ~~~
        \_____ o           __/
          \    \         __/
           \____\_______/

 _                 _   ____     _            _
| |__   ___   ___ | |_|___ \ __| | ___   ___| | _____ _ __
| '_ \ / _ \ / _ \| __| __) / _` |/ _ \ / __| |/ / _ \ '__|
| |_) | (_) | (_) | |_ / __/ (_| | (_) | (__|   <  __/ |
|_.__/ \___/ \___/ \__|_____\__,_|\___/ \___|_|\_\___|_|
Boot2Docker version 1.10.2, build master : 611be10 - Mon Feb 22 22:47:06 UTC
2016
Docker version 1.10.2, build c3959b1
```

```
docker@local-machine:~$ docker ps -a
CONTAINER ID   IMAGE    COMMAND                  CREATED             STATUS
PORTS              NAMES
bdf68af495ea   busybox  "echo Je tourne sur l"   2 minutes ago       Exited
(0) 2 minutes ago   boring_mahavira
```

Pour être complet, listons rapidement les commandes de Docker Machine qui permettent de gérer le démarrage, l'arrêt et la suppression d'une machine :

Commande	Description
`docker-machine start` *machine*	Démarre une machine.
`docker-machine stop` *machine*	Arrête une machine « gentiment ».
`docker-machine kill` *machine*	Arrête une machine de manière « brutale » (selon les drivers, cette commande est parfois identique à la commande stop, par exemple pour OpenStack).
`docker-machine restart` *machine*	Redémarre une machine (équivalent à la commande stop suivie de la commande start).
`docker-machine provision` *machine*	Relance la création d'une machine existante. Cette commande est principalement utilisée dans le cas où la commande docker-machine create a partiellement échoué.
`docker-machine regenerate-certs` *machine*	Régénère et recopie les certificats TLS utilisés pour sécuriser le démon Docker.

Nous savons maintenant tout ce qu'il faut pour créer des machines Docker à l'aide de l'outil Docker Machine.

4.2 PROJECT ATOMIC : LA DISTRIBUTION DOCKER DE FEDORA

Nous l'avons évoqué dans le chapitre 1, il existe aujourd'hui des versions d'OS spécialisées pour l'exécution de Docker. Parmi celles-ci, nous allons nous intéresser à Project Atomic[1].

Cet OS est issu d'un groupe de travail du projet Fedora qui fournit la base Open Source à plusieurs distributions bien connues, comme RedHat ou CentOS. L'intérêt d'utiliser ce type de distribution est qu'il est associé à une configuration optimisée pour Docker.

Dans cette section, nous allons installer un hôte Atomic Fedora (il existe aussi une version Atomic CentOS) dans une machine virtuelle VirtualBox. Notez qu'il existe une box Vagrant préconfigurée, mais nous avons choisi ici une installation manuelle.

4.2.1 Préparer l'image

Le projet Atomic met à disposition des images toutes prêtes pour divers environnements d'exécution (dont Vagrant ou Amazon). Pour notre part, nous opterons pour

1. http://www.projectatomic.io/

une image host Fedora conditionnée pour OpenStack sous le format qcow2. Nous devrons ensuite la convertir en fichier VDI, le format d'image VirtualBox.

Attention : processus de préparation

La préparation de l'image nécessite plutôt une machine Linux. Il est en effet nécessaire de recourir à des utilitaires (comme qemu-img ou cloud-init) qui existent sous Windows, mais dont l'installation est relativement pénible.

Dans cette section, nous procéderons à la préparation de l'image sur la machine virtuelle CentOS que nous avons préparée dans le chapitre précédent.

Afin que les étapes soient claires pour le lecteur, nous les avons résumées dans la figure ci-dessous :

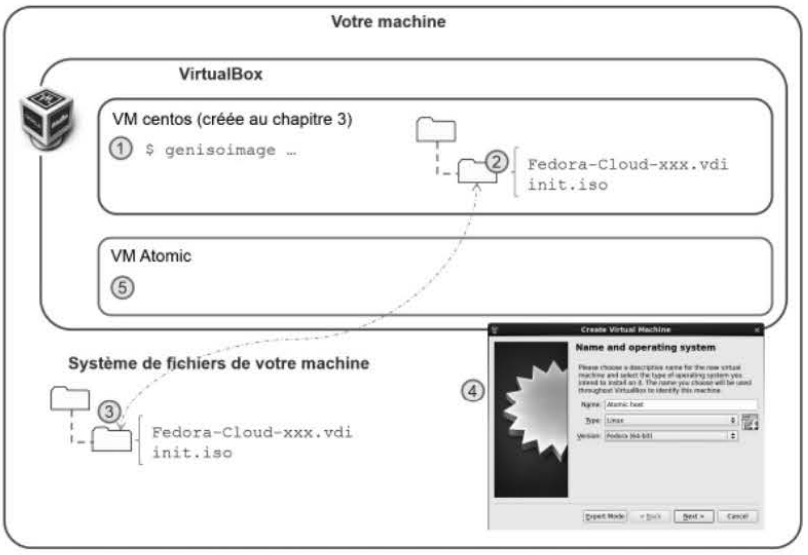

Figure 4.6 — Étapes de création de notre VM Atomic

Les étapes décrites dans la figure 4.6 sont les suivantes :

1. Dans la VM de travail, nous allons créer l'image de notre VM Atomic à l'aide de divers utilitaires.
2. Ensuite, nous transférerons les images créées (un fichier VDI et un fichier ISO) vers notre machine hôte (celle que vous utilisez et sur laquelle est installé VirtualBox).
3. Ce transfert est très simple car il est géré par VirtualBox et Vagrant (il s'agit du partage de fichiers que nous avons activé dans le chapitre 3).
4. Ensuite, nous utiliserons VirtualBox pour créer une VM Atomic à partir des fichiers VDI et ISO.
5. Enfin, nous pourrons nous connecter sur cette VM Atomic pour quelques opérations additionnelles.

Téléchargeons l'image

Connectez-vous à notre VM de travail et téléchargez (à l'aide de Firefox qui est installé) l'image depuis le site du projet Fedora :

https://getfedora.org/cloud/download/index.html

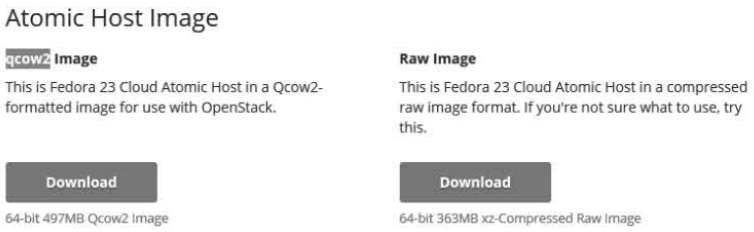

Figure 4.7 — Téléchargement de l'image host sur le site https://getfedora.org/cloud/download/index.htm

Dans notre cas, nous allons utiliser l'image QCOW2.

Convertissons l'image au format VDI

La première étape du processus consiste à convertir l'image QCOW2 au format VDI qui est le format de disque supporté par VirtualBox.

Pour ce faire, nous avons besoin de l'utilitaire qemu-img que nous allons installer :

```
$ sudo yum install -y qemu-img
```

Créez ensuite dans votre répertoire courant un répertoire nommé *atomic*, dans lequel nous allons placer les différents éléments dont nous aurons besoin :

```
$ mkdir atomic
$ cd atomic
$ pwd
/home/vargrant/atomic
```

Placez dans ce répertoire le fichier QCOW2 que vous venez de télécharger. Si vous avez gardé le chemin par défaut de Firefox, la commande suivante devrait convenir :

```
$ mv ~/Téléchargements/*.qcow2 .
```

Lançons maintenant la commande de conversion :

```
$ qemu-img convert -f qcow2 Fedora-Cloud-Atomic-23-20160420.x86_64.qcow2 -O
vdi Fedora-Cloud-Atomic-23-20160420.x86_64.vdi
```

Attention, la commande ci-dessus utilise une version donnée de l'image Fedora. Celle-ci peut être différente dans votre cas.

Créons le fichier d'initialisation de l'image

Le projet Atomic utilise l'utilitaire cloud-init[1] pour préparer l'image au démarrage de la VM. Cet utilitaire fonctionne un peu à la manière de Vagrant. Il va, au démarrage, configurer la machine virtuelle en s'appuyant sur un fichier ISO qui sera inséré dans le lecteur CD virtuel de la machine VirtualBox.

Installons d'abord l'utilitaire nécessaire à cette opération :

```
$ sudo yum install genisoimage
```

Nous devons ensuite préparer les fichiers de configuration (ce sont les équivalents du fichier VagrantFile de Vagrant). Ceux-ci permettent de nommer la machine qui sera créée, mais aussi de définir le mot de passe de l'utilisateur par défaut.

Vous devez donc éditer deux fichiers : meta-data et user-data.

```
$ vi meta-data
instance-id: atomic-host-sample
local-hostname: atomic.sample.org

$ vi user-data
#cloud-config
password: atomic
ssh_pwauth: True
chpasswd: { expire: False }
```

L'utilisateur qui sera créé aura :

- pour login *fedora* ;
- pour mot de passe *atomic*.

Lançons maintenant la commande de génération du fichier init.iso :

```
$ genisoimage -output init.iso -volid cidata -joliet -rock user-data meta-data
I: -input-charset not specified, using utf-8 (detected in locale settings)
Total translation table size: 0
Total rockridge attributes bytes: 331
Total directory bytes: 0
Path table size(bytes): 10
Max brk space used 0
183 extents written (0 MB)
```

Et maintenant transférons ces fichiers sur notre hôte pour passer à l'étape de création de la VM avec VirtualBox :

```
$ sudo mv *.vdi *.iso /media/sf_vagrant
```

Voilà ce que devrait contenir votre répertoire de travail Vagrant sur votre machine :

1. http://cloudinit.readthedocs.io

Figure 4.8 — Fichiers images prêts à l'emploi

4.2.2 Création de la VM Atomic

À l'aide des fichiers VDI et ISO, la création de la machine virtuelle se fait en mode graphique avec VirtualBox.

Une nouvelle machine virtuelle

Pour commencer, créez une nouvelle machine virtuelle en entrant les paramètres affichés sur la figure 4.9 :

Figure 4.9 — Création d'une machine virtuelle

Nous allons maintenant définir la mémoire disponible pour cette machine virtuelle. Nous allouons ici 4 Go de RAM, mais vous êtes libre de laisser la valeur par défaut proposée par VirtualBox, car elle sera suffisante pour cet exercice.

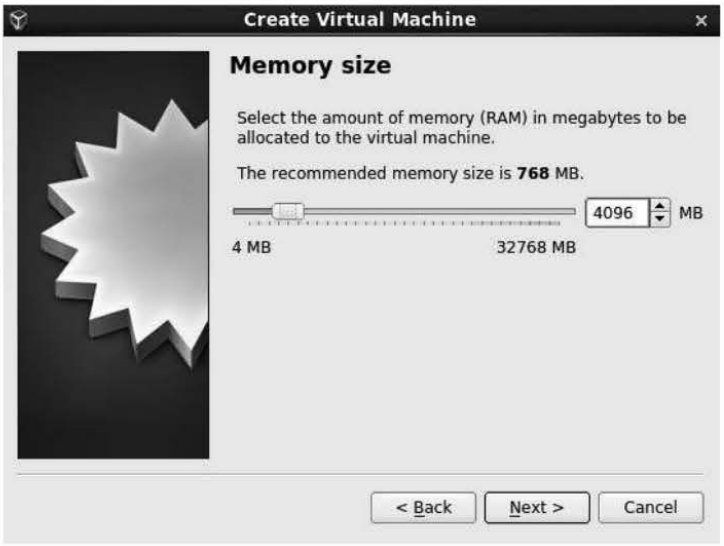

Figure 4.10 — Configuration de la mémoire

Nous arrivons maintenant à la création du disque de la machine (figure 4.11). Choisissez un disque existant et sélectionnez le fichier VDI que nous avons précédemment créé :

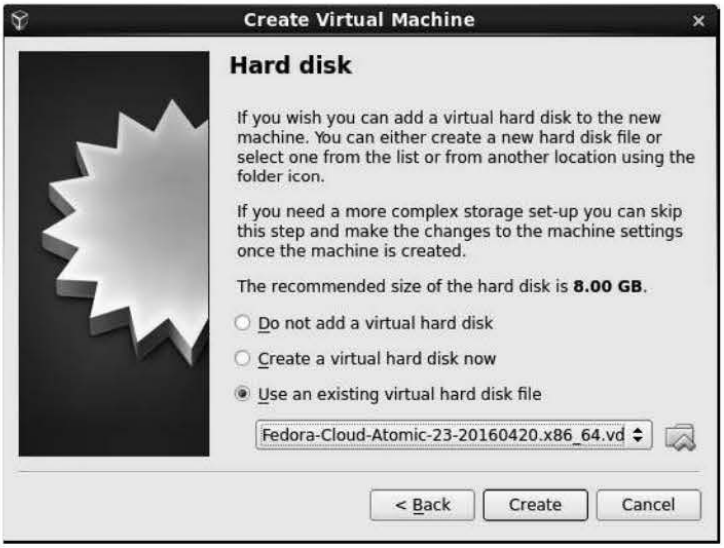

Figure 4.11 — Ajout du fichier image Fedora Atomic (fichier VDI)

Pour finir, il faut insérer le fichier init.iso dans le lecteur CD virtuel de la machine (figure 4.12).

Nous pourrions démarrer la machine à ce stade, mais nous allons encore effectuer deux opérations additionnelles :

- créer un disque additionnel de grande capacité (nous verrons qu'il nous permettra d'aborder la problématique de la configuration de *devicemapper*, le pilote de stockage de Docker déjà évoqué dans le chapitre 1) ;
- configurer l'accès réseau de la machine afin de pouvoir s'y connecter en SSH (et ainsi éviter les problèmes de clavier QWERTY).

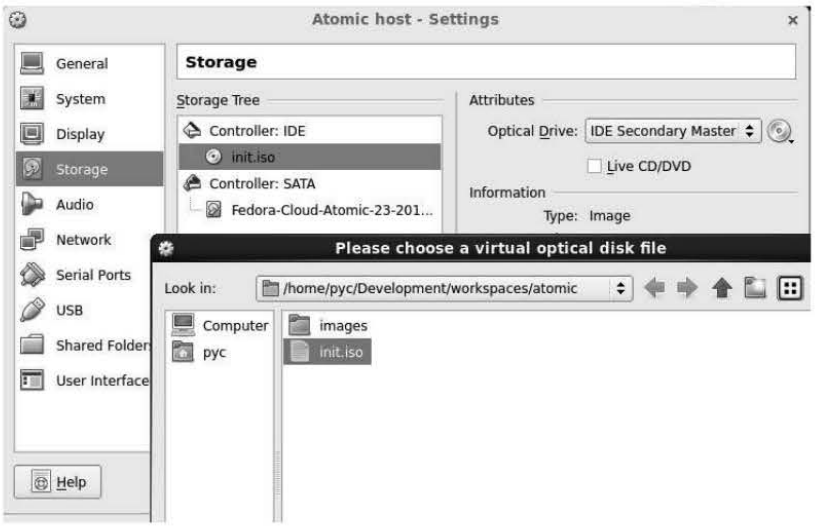

Figure 4.12 — Ajout du fichier init.iso dans le lecteur CD de la machine virtuelle

Ajout d'un disque dur

Toujours dans le menu *Storage*, sélectionnez le contrôleur SATA et ajoutez un disque (en cliquant sur l'icône correspondante).

Une première étape (figure 4.13) vous demande de choisir entre un disque existant et un nouveau disque. C'est cette seconde option que nous allons choisir.

Dans la suite des écrans, sélectionnez les options suivantes :

- disque VDI ;
- disque de taille fixe ;
- taille de 20 Go.

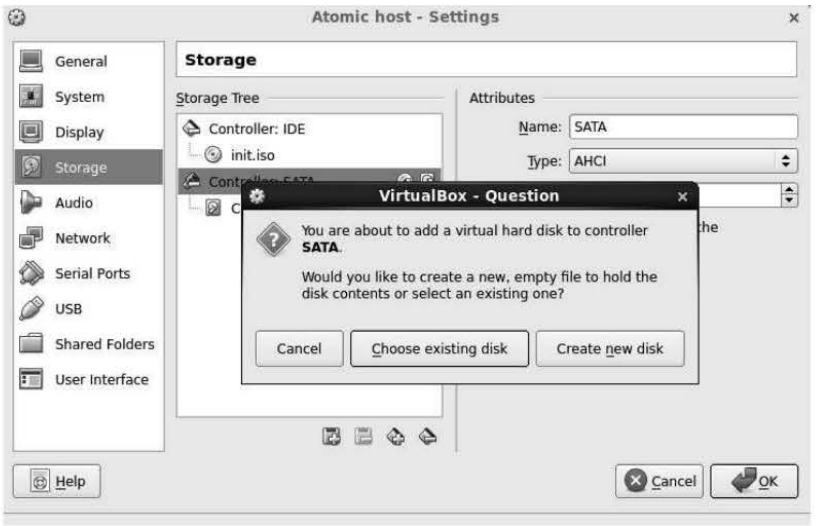

Figure 4.13 — Création d'un disque additionnel

Paramétrage du réseau

Nous vous proposons de configurer la VM pour qu'elle expose son port 22 sur l'hôte via le port 9022. Ceci nous permettra de nous connecter à cette VM à l'aide d'un client SSH (et ainsi de ne pas souffrir des problèmes de clavier américain).

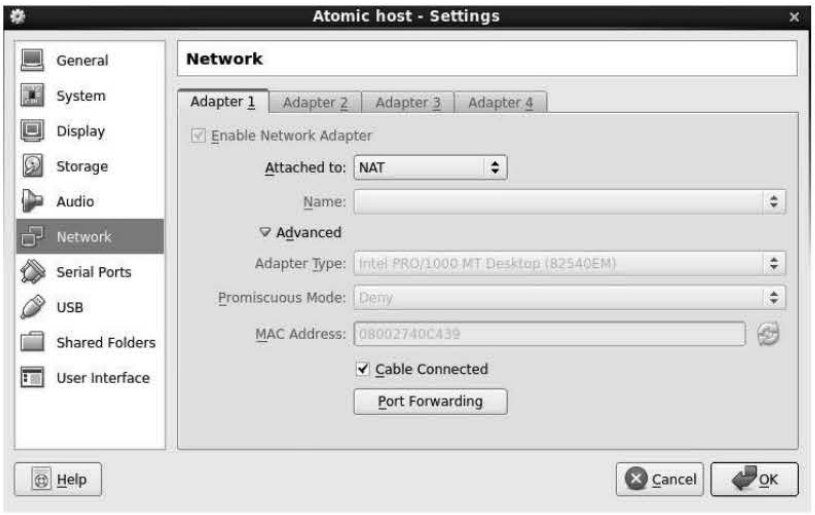

Figure 4.14 — Configuration du réseau

Ouvrez le panneau de configuration du réseau (figure 4.14), puis sélectionnez *Port forwarding*.

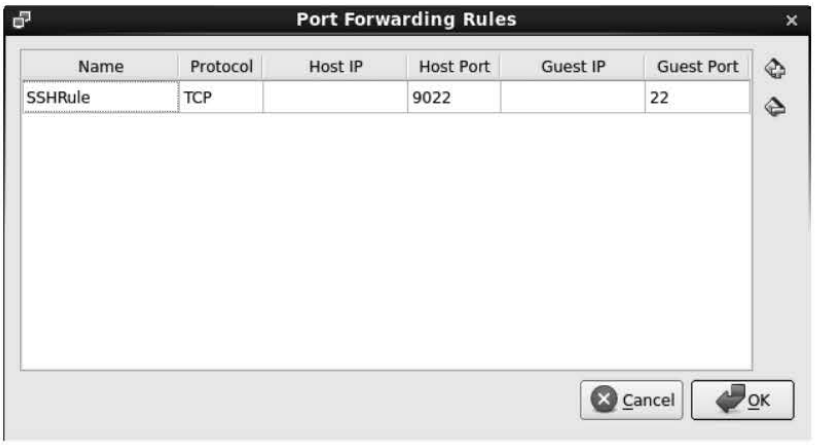

Figure 4.15 — Association des ports réseau de l'hôte et de la VM

Configurez les redirections de ports, comme cela est indiqué à la figure 4.15.

Nous sommes maintenant prêts à nous connecter à notre VM Atomic.

4.2.3 Lancement et configuration de devicemapper

Lancez la machine virtuelle et connectez-vous à l'aide de SSH :

```
$ ssh -p 9022 fedora@localhost
```

Sous Windows, vous pouvez utiliser un outil comme putty si vous ne disposez pas d'une ligne de commande.

Une fois la machine lancée, vous pouvez constater que Docker est déjà installé en lançant la commande `docker info` dont nous verrons qu'elle donne des informations sur l'installation du moteur de conteneurs :

```
$ sudo docker info
Containers: 0
Images: 0
Server Version: 1.9.1
Storage Driver: devicemapper
 Pool Name: atomicos-docker--pool
 Pool Blocksize: 524.3 kB
 Base Device Size: 107.4 GB
 Backing Filesystem: xfs
 Data file:
 Metadata file:
 Data Space Used: 62.39 MB
 Data Space Total: 2.147 GB
 Data Space Available: 2.085 GB
 Metadata Space Used: 40.96 kB
 Metadata Space Total: 8.389 MB
 Metadata Space Available: 8.348 MB
```

Vous pouvez noter dans le listing ci-dessus que l'espace alloué pour les données Docker n'est que de 2 Go. Dans un environnement réel, cette taille est très rapidement atteinte. C'est la raison pour laquelle nous avons créé un nouveau disque de grande capacité.

Ceci va nous permettre d'aborder un sujet important : la configuration du stockage de données Docker.

Petite digression sur devicemapper

Nous l'avons déjà évoqué dans le chapitre 1, l'un des *storage drivers* (pilote de gestion des images, conteneurs et volumes) de Docker les plus utilisés (spécifiquement dans les environnements Fedora, mais aussi sous Ubuntu ou Debian) est *devicemapper*.

Celui-ci est différent des autres storage drivers dans la mesure où il fonctionne avec une granularité inférieure au fichier. Il s'appuie sur un mécanisme de stockage basé sur des blocs de données.

Sans entrer dans le détail de ce mécanisme[1], il est important de savoir que l'installation par défaut de Docker pour ces environnements **ne doit pas être utilisée telle quelle en production**. En effet, par défaut, le mécanisme de stockage s'appuie sur des fichiers (aussi nommés sparse files ou fichiers dynamiques). Ceci simplifie la configuration (l'utilisateur peut utiliser Docker immédiatement), mais ce n'est pas

1. Le lecteur intéressé pourra se reporter à l'article suivant :
https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/

approprié pour un système réel car c'est très consommateur de ressources, ce qui dégrade les performances.

Heureusement, grâce à sa conception, Atomic est correctement préconfiguré avec le mode de production (direct-lvm), bien que le disque image par défaut soit trop petit (à peine 2 Go pour le stockage des blocs).

C'est la raison pour laquelle nous avons créé un disque additionnel. Nous allons maintenant étudier comment installer ce nouveau disque en créant un nouveau *volume group*.

Étendre le pool de stockage Docker

La commande suivante va nous permettre d'identifier notre disque de 20 Go :

```
$ sudo fdisk -l
```

Le voici :

```
Disk /dev/sdb: 20 GiB, 21474836480 bytes, 41943040 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

À ce stade, le disque est accessible depuis l'OS, mais pas alloué à Docker qui s'appuie sur un *volume group* par défaut nommé atomicos et de taille restreinte :

```
$ sudo vgdisplay
  --- Volume group ---
  VG Name               atomicos
  System ID
  Format                lvm2
...
  VG Size               5.70 GiB
  PE Size               4.00 MiB
```

Nous allons donc en créer un nouveau en utilisant le disque que nous avons ajouté à notre machine virtuelle lors de sa création.

Pour ce faire, nous allons en premier lieu acquérir les droits root :

```
$ sudo su
```

Ensuite, nous allons modifier le fichier de configuration de l'utilitaire docker-storage-setup :

```
$ bash-4.2# vi /etc/sysconfig/docker-storage-setup
GROWPART=true
VG="myextendedpool"
STORAGE_DRIVER="devicemapper"
DEVS="/dev/sdb"
```

Ce fichier vous indique que :

- nous utilisons le gestionnaire de stockage *devicemapper* ;
- nous allouons notre disque physique de 20 Go (précédemment identifié comme /dev/sdb) ;
- c'est le nouveau volume group[1] qui assurera le stockage des données pour Docker (via le driver *devicemapper*).

Lançons maintenant le script de reconfiguration docker-storage-setup :

```
bash-4.3# docker-storage-setup
Checking that no-one is using this disk right now ... OK

Disk /dev/sdb: 20 GiB, 21474836480 bytes, 41943040 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

>>> Script header accepted.
>>> Created a new DOS disklabel with disk identifier 0xf3d12669.
Created a new partition 1 of type 'Linux LVM' and of size 20 GiB.
/dev/sdb2:
New situation:

Device     Boot Start      End  Sectors Size Id Type
/dev/sdb1        2048 41943039 41940992  20G 8e Linux LVM

The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
  Physical volume "/dev/sdb1" successfully created
  Volume group "myextendedpool" successfully created
NOCHANGE: partition 2 is size 11966464. it cannot be grown
  Physical volume "/dev/sda2" changed
  1 physical volume(s) resized / 0 physical volume(s) not resized
  Rounding up size to full physical extent 24.00 MiB
  Logical volume "docker-poolmeta" created.
  Logical volume "docker-pool" created.
  WARNING: Converting logical volume myextendedpool/docker-pool and
myextendedpool/docker-poolmeta to pool's data and metadata volumes.
  THIS WILL DESTROY CONTENT OF LOGICAL VOLUME (filesystem etc.)
  Converted myextendedpool/docker-pool to thin pool.
  Logical volume "docker-pool" changed.
```

Nous pouvons maintenant redémarrer le démon Docker et regarder si le changement de configuration a porté ses fruits :

1. Pour plus d'explications relatives à la gestion de volumes logiques sous Linux, reportez-vous à l'article Wikipédia suivant :
https://en.wikipedia.org/wiki/Logical_Volume_Manager_%28Linux%29

```
bash-4.3# service docker restart
Redirecting to /bin/systemctl restart  docker.service
bash-4.3# docker info
Containers: 0
Images: 0
Server Version: 1.9.1
Storage Driver: devicemapper
 Pool Name: myextendedpool-docker--pool
 Pool Blocksize: 524.3 kB
 Base Device Size: 107.4 GB
 Backing Filesystem: xfs
 Data file:
 Metadata file:
 Data Space Used: 62.39 MB
 Data Space Total: 8.577 GB
 Data Space Available: 8.515 GB
 Metadata Space Used: 45.06 kB
 Metadata Space Total: 25.17 MB
 Metadata Space Available: 25.12 MB
```

Vous pouvez constater que le volume de stockage alloué aux données est maintenant de 8,577 Go contre 2,085 Go. Sans entrer dans le détail du fonctionnement du script docker-storage-setup, il s'agit par défaut de 40 % de l'espace disque physique qui a été ajouté (nos 20 Go). Il est évidemment possible de modifier la configuration du script pour obtenir d'autres résultats[1].

En résumé

Dans ce chapitre, nous avons appris comment installer Docker sur un hôte en utilisant deux types de solutions différentes. Ceci nous a permis d'aborder l'usage de l'un des outils de l'écosystème Docker (Docker Machine). Nous nous sommes aussi intéressés à la distribution Linux Atomic et, par l'intermédiaire de cet exemple, à la configuration de *devicemapper*, un pilote de stockage (*storage driver*) majeur pour Docker.

Il est temps à présent de créer nos premiers conteneurs et de jouer avec la ligne de commande Docker.

1. https://github.com/projectatomic/docker-storage-setup

5

Conteneurs et images

Fabriquer une image et exécuter un conteneur

L'objectif de ce chapitre est de débuter la prise en main de Docker à l'aide de l'environnement décrit dans le chapitre 3. Pour ce faire, nous allons étudier le cycle de vie d'un conteneur pas à pas, à partir d'une image officielle du Docker Hub. Ensuite nous montrerons comment construire une image originale pour fabrique nos propres conteneurs.

L'objet de ce chapitre n'est pas d'approfondir chaque concept, mais d'effectuer un survol relativement complet des fonctionnalités du Docker Engine.

À l'issue de ce chapitre, vous saurez créer une image, un conteneur et vous maîtriserez quelques commandes de base du client Docker.

5.1 LE CYCLE DE VIE DU CONTENEUR

Comme nous l'avons évoqué dans les deux précédents chapitres, un conteneur Docker est fabriqué à partir d'une image. Il peut ensuite adopter différents états que nous allons maintenant présenter.

> Nous supposons que le lecteur a installé l'environnement recommandé dans le chapitre 3 et s'est assuré que le service Docker est effectivement démarré.

Le diagramme ci-dessous montre l'ensemble des états possibles d'un conteneur. Vous noterez qu'un conteneur peut être créé et démarré immédiatement (commande `run`) ou créé dans l'attente d'être démarré (commande `create`). D'autres commandes,

comme `stop` ou `kill`, peuvent aussi sembler redondantes de prime abord. Nous en expliquerons en temps voulu les subtilités.

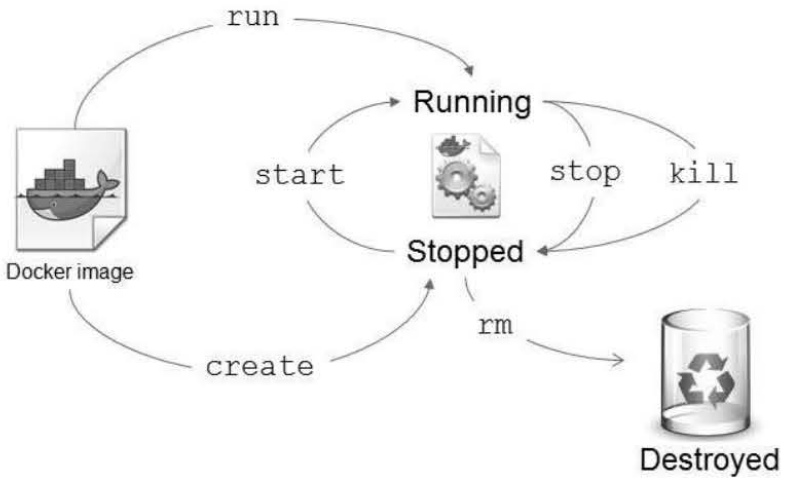

Figure 5.1 — Cycle de vie du conteneur Docker

5.1.1 Créer et démarrer un conteneur (commande run)

Créons un conteneur nginx[1]. Il s'agit d'un conteneur exécutant le serveur web open source Nginx. Notez que l'image nginx est un *official repository* du Docker Hub, c'est-à-dire une image certifiée par l'entreprise Docker Inc.

```
$ docker run -p 8000:80 -d nginx
```

Sans entrer dans le détail à ce stade, vous noterez que nous passons trois paramètres à cette commande `run` :

- `-p 8000:80` spécifie comment le port 80 du serveur web, ouvert à l'intérieur du conteneur, doit être exposé à l'extérieur (sur notre hôte). Nous reviendrons par la suite sur ces problématiques de gestion du réseau. À ce stade, nous retiendrons que le port 80 du serveur web sera exposé sur le port 8000 de l'hôte.
- `-d` indique que le conteneur doit s'exécuter en mode démon, c'est-à-dire non bloquant. Sans ce paramètre, la ligne de commande serait bloquée et afficherait les logs du serveur web.
- `nginx` correspond au nom de l'image qui va être utilisée pour instancier le conteneur.

Si cette commande est exécutée pour la première fois, elle va entraîner le téléchargement de l'image Docker officielle nginx qui va servir de moule pour la création de notre conteneur.

1. http://nginx.org/

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
9ee13ca3b908: Pull complete
23cb15b0fcec: Pull complete
62df5e17dafa: Pull complete
d65968c1aa44: Pull complete
f5bb1dddc876: Pull complete
1526247f349d: Pull complete
2e518e3d3fad: Pull complete
0e07123e6531: Pull complete
21656a3c1256: Pull complete
f608475c6c65: Pull complete
1b6c0a20b353: Pull complete
5328fdfe9b8e: Pull complete
Digest:
sha256:a79db4b83c0dbad9542d5442002ea294aa77014a3dfa67160d8a55874a5520cc
Status: Downloaded newer image for nginx:latest
050e5c557d8a7a7ed37508c4cf35d260844ef75bfae52f5bdd2302dac7b78806
```

Comme vous pourrez le constater par la suite, cette phase de téléchargement ne sera plus nécessaire par la suite. L'image est en effet désormais stockée dans le registry local de votre ordinateur.

Vérifions que le serveur web fonctionne correctement en faisant un simple appel à sa *home page*, http://localhost:8000, en utilisant le navigateur Firefox installé dans notre image Linux de référence.

Figure 5.2 — Premier conteneur Nginx

Que faire de ce conteneur ?

Le cycle de vie de la figure 5.1 nous indique que deux options sont possibles :

- stopper le conteneur avec la commande `stop` ;
- tuer le conteneur avec la commande `kill`.

5.1.2 Stopper un conteneur (commande stop ou kill)

Il n'y a pas de grande différence entre les deux commandes, si ce n'est la violence de l'injonction de s'arrêter. `kill` va envoyer un signal *SIGKILL* au processus principal qui s'exécute dans le conteneur (Nginx dans notre cas). La commande `stop` va tout

d'abord envoyer un message SIGTERM puis, après un délai donné et si le processus n'est pas encore arrêté, un message *SIGKILL* (comme le ferait la commande Unix `kill -9`). En général, on utilise donc plutôt la méthode stop.

Pour stopper un conteneur, il est nécessaire de connaître son identifiant.

Identifier un conteneur

Pour déterminer cet identifiant, nous utiliserons la commande `ps`.

```
$ docker ps
```

Cette commande liste tous les conteneurs en cours d'exécution de même que d'autres informations utiles :

```
CONTAINER ID    IMAGE         COMMAND                  CREATED
855bbd5e9823    nginx         "nginx -g 'daemon off"   17 minutes ago
STATUS              PORTS                           NAMES
Up 17 minutes       443/tcp, 0.0.0.0:8000->80/tcp   stupefied_fermat
```

Voici un descriptif des informations retournées par cette commande `ps` :

Champ	Description
CONTAINER ID	Chaque conteneur Docker est associé à un identifiant unique généré par le démon Docker. Celui qui est affiché ci-dessus est l'identifiant court. Il existe aussi un identifiant long globalement unique (selon la norme RFC4122*) et qui est visualisable en utilisant la commande : docker ps –no-trunc
IMAGE	L'image à partir de laquelle a été créé ce conteneur
COMMAND	La commande qui est exécutée dans le conteneur. Dans notre cas, on constate que Nginx est lancé en mode *deamon off*. Cela signifie que cette commande est bloquante : elle ne rend pas la main après avoir été exécutée. Nous verrons que c'est une condition nécessaire, dans le cas contraire le conteneur s'arrêterait immédiatement après l'exécution de la commande.
CREATED	Combien de temps s'est écoulé depuis la création du conteneur
STATUS	L'état actuel du conteneur, dans notre cas Up. S'il venait d'être stoppé, le conteneur serait dans un état Exited.
PORTS	Les ports utilisés par le conteneur et éventuellement exposés sur l'hôte. Ici nous pouvons noter que Nginx occupe deux ports (80 et 443) à l'intérieur du conteneur. Par contre, seul le port 80 est réexposé sur l'hôte et associé au port 8000.
NAMES	C'est le nom intelligible du conteneur. Il est possible lors de l'exécution de la commande `run` (ou `create`) de spécifier un autre nom de son choix. Si tel n'est pas le cas, Docker en génère un automatiquement composé d'un adjectif et d'un nom de famille de personnage célèbre**.

* https://www.ietf.org/rfc/rfc4122.txt

** https://github.com/docker/docker/blob/master/pkg/namesgenerator/names-generator.go

Résultat de la commande stop

Maintenant que nous sommes en possession de l'UUID (l'identifiant unique) de notre conteneur, nous allons le stopper. Notez que l'UUID court est suffisant lorsque l'on travaille sur un hôte donné.

```
$ docker stop 855bbd5e9823
```

Le rafraîchissement du navigateur sur l'adresse http://localhost:8000/ montre que le serveur Nginx est désormais éteint.

Notre conteneur est-il pour autant définitivement détruit ?

Non, celui-ci existe toujours ou, plus exactement, l'image complémentée de la couche persistante read/write propre aux conteneurs (cf. chapitre 1, *union file systems*) est toujours stockée sur le système.

Sur notre hôte (pour peu que l'on dispose des droits root), nous pouvons même lister chaque répertoire contenant l'état des conteneurs.

```
$ ls /var/lib/docker/containers/
855bbd5e98239bd156253b370dae0384e628bc1f3300dea60fde20b3d952b2a1
```

Nous constatons dans notre cas que le répertoire /var/lib/docker/containers/ ne contient qu'un seul sous-répertoire. Vous aurez peut-être remarqué que les douze premiers caractères correspondent à l'UUID court de notre conteneur (855bbd5e9823). Le nom du répertoire est en effet constitué de l'UUID long de ce dernier.

Redémarrer un conteneur stoppé

Si nous exécutons la commande `docker ps` comme précédemment, vous noterez que notre conteneur stoppé n'apparait plus. En effet, par défaut, la commande `ps` n'affiche que les conteneurs Up. Il faut ajouter le paramètre `-a` (pour « All ») afin d'afficher les conteneurs qui ne sont pas actifs.

```
$ docker ps -a

CONTAINER ID        IMAGE               COMMAND                  CREATED
855bbd5e9823        nginx               "nginx -g 'daemon off"   2 hours ago

STATUS                      PORTS               NAMES
Exited (0) 26 minutes ago                       stupefied_fermat
```

Nous allons maintenant redémarrer ce conteneur.

```
$ docker start 855bbd5e9823
```

Le test habituel via notre navigateur nous confirme que Nginx est actif à nouveau.

Notez que nous aurions pu aussi lancer la commande suivante qui, à la place de l'identifiant du conteneur, utilise son nom (généré automatiquement par Docker).

```
$ docker start stupefied_fermat
```

5.1.3 Détruire un conteneur (commande rm)

C'est la dernière étape du cycle de vie de notre conteneur. Tentons de stopper notre conteneur.

```
$ docker rm 855bbd5e9823
Error response from daemon: Cannot destroy container 855bbd5e9823: Conflict,
You cannot remove a running container. Stop the container before attempting
removal or use -f
Error: failed to remove containers: [855bbd5e9823]
```

Docker nous indique que nous ne pouvons pas détruire un conteneur qui n'est pas stoppé. Comme cela est indiqué sur la figure 5.1, nous devons d'abord stopper le conteneur avant de le détruire.

```
$ docker stop 855bbd5e9823
$ docker rm 855bbd5e9823
```

Cette fois, `docker ps -a` montre que notre conteneur n'existe plus au niveau du système, ce que confirme un *ls /var/lib/docker/containers/*.

> Notez qu'il est possible de forcer la destruction d'un conteneur Up en utilisant, comme le suggère le message d'erreur plus haut, le paramètre `-f` (ou `--force`). Ce paramètre provoque l'envoie un *SIGKILL* (comme pour la commande `kill`), ce qui a pour effet de stopper le conteneur avant de le détruire.

5.1.4 La commande create

Certains auront probablement remarqué que nous n'avons pas abordé le cas de la commande `create`. Celle-ci est en fait très similaire à la commande `run`, si ce n'est qu'elle crée un conteneur inactif et prêt à être démarré.

```
$ docker create -p 8000:80 nginx
```

La commande ci-dessus crée un conteneur dans un statut nouveau (ni Up, ni Exited) : Created.

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
2ab2ba1543ce        nginx               "nginx -g 'daemon off"   8 seconds ago
STATUS              PORTS               NAMES
Created                                 romantic_nobel
```

Il ne reste alors qu'à démarrer notre conteneur en utilisant l'habituelle commande `start`.

```
$ docker start 2ab2ba1543ce
```

Nous avons parcouru dans cette section l'ensemble du cycle de vie d'un conteneur. Néanmoins, le conteneur que nous avons utilisé était relativement inintéressant. Nous devons donc maintenant aborder deux questions :

- Comment configurer le comportement d'un conteneur et le faire, par exemple, interagir avec son système hôte ?
- Comment créer une image originale qui fournisse un service correspondant à nos besoins ?

5.2 ACCÉDER AU CONTENEUR ET MODIFIER SES DONNÉES

Nous allons aborder dans cette section différents points d'intégration entre le conteneur et son système hôte :

- la connexion en mode terminal avec le conteneur ;
- la gestion des volumes et la manière d'assurer la persistance de données sur l'ensemble du cycle de vie d'un conteneur (y compris après sa destruction) ;
- la configuration des ports IP.

Nous ne traiterons pas des liens réseaux entre les conteneurs. Cet aspect sera traité dans la quatrième partie de cet ouvrage sur la base d'exemples significatifs.

5.2.1 Connexion en mode terminal

Dans le paragraphe précédent, nous avons utilisé une image générique prise depuis le Docker Hub sans la modifier. En pratique, lancer un conteneur sans le modifier n'a aucun intérêt.

Nous allons reprendre notre image nginx en modifiant les pages retournées par le serveur web.

Attention, pensez à supprimer le conteneur créé au paragraphe précédent avant de vous lancer dans celui-ci. Pour ce faire utilisez la commande `docker stop` puis `docker rm` comme nous l'avons fait un peu plus haut. Dans le cas contraire il est probablement qu'un message d'erreur « *Bind for 0.0.0.0:8000 failed: port is already allocated* » ne s'affiche indiquant que le port 8000 est déjà occupé !

La page que nous avons vue jusqu'à maintenant se trouve (au sein du conteneur) dans le répertoire /usr/share/nginx/html.

Comment la modifier ?

Comme nous allons le voir plus bas, Docker offre la possibilité de lancer des commandes dans un conteneur actif tout en obtenant un lien de type pseudo-terminal.

En premier lieu, démarrons un nouveau conteneur.

```
$ docker run -d -p 8000:80 --name webserver nginx
```

Vous noterez que nous venons d'utiliser un nouveau paramètre `--name` pour la commande `run`. Celui-ci permet de changer le nom d'un conteneur en utilisant un alias généralement plus simple à mémoriser que l'UUID ou le nom donné par défaut par Docker.

```
CONTAINER ID        IMAGE               COMMAND                  CREATED
28314fdbd549        nginx               "nginx -g 'daemon off"   2 minutes ago
STATUS              PORTS                                NAMES
Up 2 minutes        443/tcp, 0.0.0.0:8000->80/tcp   webserver
```

Pour se connecter à l'intérieur de ce conteneur actif, nous allons utiliser la commande `exec`.

```
$ docker exec -t -i webserver /bin/bash
```

La commande `exec` permet de démarrer un processus dans un conteneur actif. Dans notre cas, nous lançons un nouveau terminal (*bash*) en l'associant (grâce aux paramètres `-i` et `-t`) au flux d'entrée (*STDIN*) et à un pseudo-terminal. L'exécution de la commande ouvre un *prompt* Unix qui permet alors de naviguer à l'intérieur du conteneur.

```
root@28314fdbd549:/# hostname
28314fdbd549
root@28314fdbd549:/# ls
bin  boot  devetc  home  liblib64  media  mnt  optproc  root  run  sbin  srv
sys  tmp  usr  var
```

Comme vous pouvez le constater, le nom d'hôte est 28314fdbd549, ce qui correspond à l'UUID court du conteneur. Ce sera systématiquement le cas pour tout conteneur.

Pour confirmer que nous sommes bien à l'intérieur du conteneur, nous allons modifier la *home page* du serveur web. La commande suivante écrase le contenu du fichier HTML par défaut avec une simple chaîne de caractères « Je suis dedans ».

```
$ echo "Je suis dedans" > /usr/share/nginx/html/index.html
$ exit
```

Ouvrons notre navigateur pour constater que la modification s'est déroulée correctement.

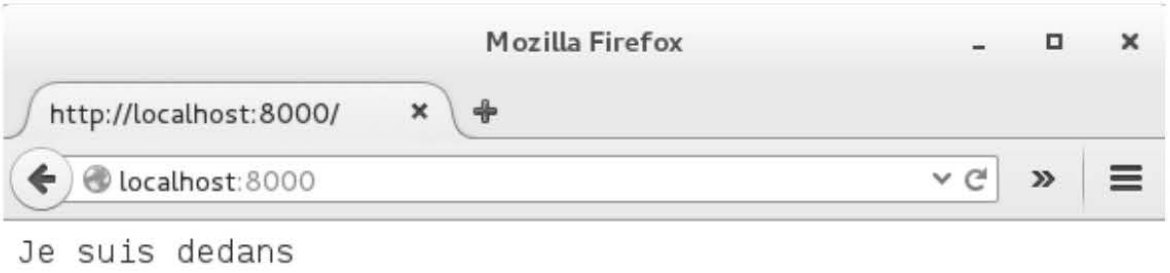

Figure 5.3 — *Home page* de Nginx modifiée depuis l'intérieur du contrôleur

Bien que fonctionnelle, cette approche n'est pas franchement efficace car il n'est pas réellement possible de modifier des quantités importantes de données. Par ailleurs, la destruction du conteneur entraînera automatiquement la perte des modifications opérées.

5.2.2 Créer un volume

Les volumes[1] sont le moyen qu'offre Docker pour gérer la persistance des données au sein des conteneurs (et en relation avec leur machine hôte).

Comme nous l'avons vu précédemment, la page d'accueil du serveur web Nginx se trouve dans le répertoire /usr/share/nginx/html/index.html. Définissons un volume correspondant à ce chemin à l'aide du paramètre `-v` de la commande `run`.

```
$ docker run -p 8000:80 --name webserver -d -v /usr/share/nginx/html  nginx
```

Attention, si vous n'avez pas supprimé le conteneur webserver créé dans le paragraphe précédent, vous noterez que la tentative d'en créer un nouveau (avec le même nom) se solde par une erreur « *Error response from daemon: Conflict. The name "webserver" is already in use by container … You have to remove (or rename) that container to be able to reuse that name.* »

1. Veuillez-vous reporter au chapitre 1 pour une explication du concept de volume de conteneur.

Pour comprendre le résultat de cette action, nous allons inspecter le conteneur ainsi créé. La commande `inspect` permet d'obtenir une structure JSON[1] décrivant le conteneur et sa configuration.

```
$ docker inspect webserver
```

Au sein de la structure retournée par la commande `inspect`, nous pouvons notamment identifier la section suivante :

```
"Mounts": [
{
    "Name":
"c29b645f024f59988c24dbb0564a6f893da51b17ccd687dfbf794ef4143c48f5",
    "Source": "/var/lib/docker/volumes
/c29b645f024f59988c24dbb0564a6f893da51b17ccd687dfbf794ef4143c48f5/_data",
    "Destination": "/usr/share/nginx/html",
    "Driver": "local",
    "Mode": "",
    "RW": true
}
```

Celle-ci nous donne plusieurs informations résumées dans le tableau ci-dessous.

Paramètre	Description
Name	Un identifiant unique pour cette structure (UUID).
Source	Le chemin sur le système de fichiers de la machine hôte contenant les données correspondant au volume.
Destination	Le chemin à l'intérieur du conteneur.
Driver	Comme nous le verrons par la suite, il est possible d'utiliser plusieurs drivers pour la gestion des volumes en offrant un stockage des données via différents systèmes (pas uniquement sur le système hôte). Dans notre cas, le driver utilisé est celui par défaut nommé « local ». Les données sont stockées sur l'hôte dans le répertoire identifié par le paramètre Source.
RW	Indique que le volume (à l'intérieur du conteneur) est *read-write* (il peut être lu et modifié).

Si nous nous plaçons dans le répertoire obtenu via la commande `inspect`, nous constatons effectivement qu'il contient le fichier index.html que nous avions trouvé précédemment dans le répertoire */usr/share/nginx/html* à l'intérieur du conteneur.

```
$ sudo ls /var/lib/docker/volumes
/c29b645f024f59988c24dbb0564a6f893da51b17ccd687dfbf794ef4143c48f5/_data
50x.html  index.html
```

1. http://www.json.org/

Attention, notez que nous sommes contraints d'utiliser la commande `sudo` pour disposer des droits root. L'accès au système de fichiers Docker n'est évidemment pas accordé sans ce niveau de privilège.

Un accès à l'URL http://localhost:8000 nous confirme que le serveur web tourne bien correctement et affiche l'habituelle page d'accueil standard de Nginx.

Nous allons maintenant l'éditer pour vérifier si cette modification est prise en compte par le conteneur. Encore une fois, nous devons recourir à des privilèges étendus. N'oubliez pas d'exécuter la commande `exit` pour revenir à l'utilisateur *vagrant*.

```
$ sudo su
$ echo "Je modifie un volume" > /var/lib/docker/volumes
/61d5b52ba3385203910dd8237700de02056a40a23af31e3b5b62d95f63117d06/_data/index.html
$ exit
```

Le résultat peut une fois de plus être visualisé à l'aide de notre navigateur web.

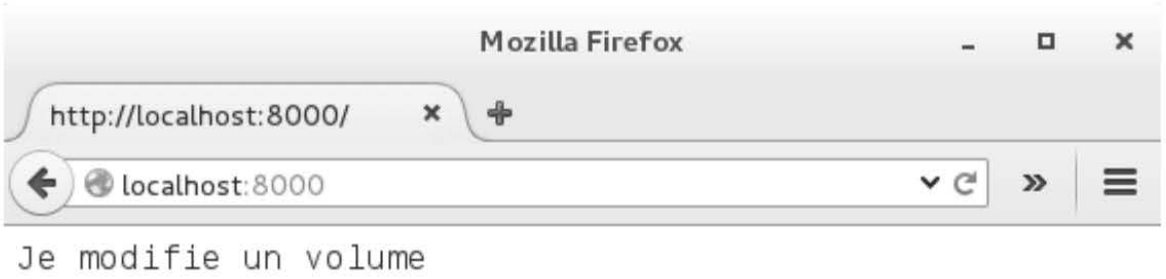

Figure 5.4 — *Home page* de Nginx modifiée sur l'hôte

Cette méthode présente néanmoins des inconvénients majeurs :

- il est nécessaire de disposer de privilèges étendus pour accéder aux données et pour les modifier ;
- la destruction/recréation du conteneur entraînera l'allocation d'un nouveau volume différent et les modifications opérées seront perdues.

En réalité Docker n'efface **jamais** le contenu du répertoire /var/lib/docker/volumes/, même lorsque les conteneurs sont détruits. Par conséquent, la création d'un nouveau conteneur à partir de la même image entraînera la création d'un autre volume. Le volume précédent restera orphelin jusqu'à ce qu'il soit effacé explicitement (manuellement ou par un script écrit à cet effet).

Si on considère que dans de nombreuses architectures les conteneurs ont pour vocation d'être éphémères et, autant que possible, sans état propre, on pourrait penser que l'intérêt de ce type de volumes locaux est limité. Nous verrons dans la troisième partie de cet ouvrage qu'il est possible à plusieurs conteneurs de référencer les mêmes volumes, ce qui permet dans une large mesure de s'abstraire des contraintes d'implémentation de ceux-ci.

5.2.3 Monter un répertoire de notre hôte dans un conteneur

Il est possible, à la place du driver local de Docker, de monter un répertoire de l'hôte dans le conteneur. Nous allons en premier lieu créer ce répertoire dans l'espace de travail de l'utilisateur *vagrant*.

```
$ mkdir -p /home/vagrant/workdir/chapitre5
```

La syntaxe de création du volume est la suivante :

```
$ docker run -p 8000:80 --name webserver -d -v
/home/vagrant/workdir/chapitre5:/usr/share/nginx/html nginx
```

> Encore une fois, pensez à vérifier à l'aide de la commande `docker ps -a` que vous n'avez pas de conteneur existant qui porte déjà le nom « webserver » ou qui écoute sur le même port 8000. Si c'est le cas (ça devrait l'être si vous avez suivi les exercices précédents pas à pas), pensez à supprimer ce conteneur à l'aide des commandes `stop` puis `rm`.

La syntaxe est relativement intuitive :

`-v` <Répertoire de l'hôte>:<Répertoire à l'intérieur du conteneur>

Il ne s'agit ni plus ni moins que d'un `mount` Unix.

Regardons notre navigateur pour voir ce que la page d'accueil affiche.

Figure 5.5 — Montage d'un répertoire vide

Le résultat est inattendu, si on le compare avec le comportement des volumes locaux, et pourtant il est parfaitement logique. Ce que nous avons créé à l'aide de la commande précédente, c'est un *mount* d'un répertoire de l'hôte dans le conteneur. Malheureusement, ce répertoire ne contenait aucun fichier index.html à afficher.

```
$ cd /home/vagrant/workdir/chapitre5
$ echo "Un fichier sur le host" > index.html
```

La commande précédente crée un fichier index.html sommaire dans notre répertoire et le résultat est immédiat.

Figure 5.6 — Montage d'un répertoire contenant un fichier index.html

Notez qu'il est possible de faire un montage en lecture-seule du point de vue du conteneur (c'est-à-dire que le conteneur ne pourra pas modifier ces données). Pour ce faire, il faut ajouter à la syntaxe de définition d'un volume le suffixe `:ro`.

5.2.4 La configuration des ports IP

Depuis le début de ce chapitre, nous créons des conteneurs qui offrent un service sur le port 8000 de l'hôte à l'aide du paramètre `-p 8000:80`. Comment la gestion des ports fonctionne-t-elle avec Docker ?

```
$ docker run -d -p 8000:80 --name webserver nginx
```

Faisons une inspection de la configuration du conteneur.

```
$ docker inspect webserver
```

La structure JSON retournée contient la section suivante :

```
"Ports": {
            "443/tcp": null,
            "80/tcp": [
                {
                    "HostIp": "0.0.0.0",
                    "HostPort": "8000"
                }
            ]
        }
```

Cette structure nous indique que le conteneur que nous avons créé déclare en fait deux ports TCP :

- le port 80 qui correspond à HTTP ;
- le port 443 qui correspond à la version sécurisée HTTPS.

Pourtant seul le port 80 est accessible au niveau de l'hôte et, comme nous l'avons indiqué, il est mappé sur le port 8000.

> Nous reviendrons plus en détail sur les problématiques de réseau et sur l'adresse IP associée ici au port exposé : 0.0.0.0[1]. À ce stade, notez seulement qu'il s'agit de l'adresse par défaut à laquelle Docker attache le port ainsi ouvert sur l'hôte.

Notez qu'il est possible de laisser Docker choisir le port de l'hôte.

```
$ docker rm -f webserver
$ docker run -P --name webserver -d nginx
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
97066be7b321        nginx               "nginx -g 'daemon off"   4 minutes ago
STATUS              PORTS                                           NAMES
Up 3 minutes        0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp   webserver
```

Lors que le paramètre `-P` est utilisé, Docker va associer les ports du conteneur avec les ports disponibles sur l'hôte (choisis au hasard).

Dans le cas ci-dessus, le port 80 est associé au port 32769 de l'hôte.

5.3 CONSTRUIRE UNE IMAGE DOCKER ORIGINALE

Nous avons jusqu'à maintenant abordé rapidement les différents aspects de la gestion des conteneurs. Au cours de cette présentation, nous avons volontairement passé sous silence un certain nombre d'éléments. Le lecteur curieux aura, par exemple, noté en lançant une commande `inspect` sur l'un de nos conteneurs Nginx que certaines configurations semblaient être prédéfinies :

- Qui a défini que le conteneur pouvait gérer les ports 80 et 443 ?
- Qui a défini le volume /var/cache/nginx qui est par défaut présent ?

Ces paramètres sont apportés par l'image nginx.

Nous allons maintenant étudier la création d'images. Dans cette première section consacrée aux images, nous n'allons pas aborder la manière canonique de procéder, mais plutôt étudier ce qu'est une image et comment la créer manuellement. Une fois ces connaissances acquises, nous étudierons l'un des apports majeurs de Docker, qui représente la manière aujourd'hui usuelle de création d'images : le Dockerfile.

1. La notion d'adresse "0.0.0.0" par rapport à "127.0.0.1" est expliquée dans le chapitre 3 (§ 3.2.3).

5.3.1 Lister les images

Avant de nous intéresser à la création de nouvelles images, étudions quelques instructions qui permettent de les manipuler.

En premier lieu, nous allons lister les images présentes sur notre machine hôte à l'aide de l'instruction `docker images` :

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
nginx               latest              5328fdfe9b8e        3 weeks ago
VIRTUAL SIZE
133.8 MB
```

Comme vous pouvez le constater, notre image nginx est bien là.

Si vous ajoutez le paramètre `-a` (pour *all*) à cette commande, vous constatez que la liste est nettement plus longue :

```
$ docker images -a
REPOSITORY          TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
nginx               latest              5328fdfe9b8e        3 weeks ago
133.8 MB
<none>              <none>              f608475c6c65        3 weeks ago
133.8 MB
<none>              <none>              1b6c0a20b353        3 weeks ago
133.8 MB
<none>              <none>              21656a3c1256        3 weeks ago
133.8 MB
<none>              <none>              0e07123e6531        3 weeks ago
133.8 MB
<none>              <none>              2e518e3d3fad        3 weeks ago
133.8 MB
<none>              <none>              1526247f349d        3 weeks ago
125.1 MB
<none>              <none>              f5bb1dddc876        5 weeks ago
125.1 MB
<none>              <none>              d65968c1aa44        5 weeks ago
125.1 MB
<none>              <none>              62df5e17dafa        5 weeks ago
125.1 MB
<none>              <none>              23cb15b0fcec        5 weeks ago
125.1 MB
<none>              <none>              9ee13ca3b908        5 weeks ago
125.1 MB
```

Mais d'où viennent ces images qui n'ont pas de nom, mais uniquement un UUID ?

Si vous lisez attentivement la section « Créer et démarrer un conteneur », vous remarquez que les mêmes identifiants étaient présents lors du chargement de l'image (lors de la création de notre premier conteneur).

Ces images sans nom sont ce que l'on appelle des images intermédiaires. Elles correspondent aux couches dont une image est constituée (voir le chapitre 1). Nous

verrons plus tard comment ces images intermédiaires sont utilisées lors de la création d'une nouvelle image.

5.3.2 Charger et effacer des images

Effaçons notre image nginx avec la commande `docker rmi` (à ne pas confondre avec `docker rm` qui, comme nous l'avons déjà vu, est réservé aux conteneurs) :

```
$ docker rmi nginx
```

Comme pour les conteneurs, nous pouvons indifféremment utiliser le nom de l'image ou son UUID (5328fdfe9b8e).

Si vous avez encore un conteneur nginx (actif ou inactif), vous risquez de voir apparaître le message d'erreur suivant :

```
Error response from daemon: conflict: unable to remove repository reference
"nginx" (must force) - container 050e5c557d8a is using its referenced image
5328fdfe9b8e
Error: failed to remove images: [nginx]
```

Il n'est en effet pas possible de supprimer une image pour laquelle des conteneurs sont encore présents. Si l'on se souvient de la structure en couches des conteneurs, on comprend que cela n'aurait effectivement pas de sens. Il vous faut donc détruire tous les conteneurs.

> Le client du Docker Engine ne propose pas de commande pour détruire plusieurs conteneurs en une seule fois. Inutile d'essayer les « * » ou autres jokers. L'astuce habituelle consiste à utiliser la commande `docker rm -f $(docker ps -a -q)` qui va arrêter tous les conteneurs et les effacer. À utiliser évidemment avec prudence.

Une fois l'image effacée, nous pouvons la recharger en utilisant le Docker Hub. Notez que le téléchargement de l'image sera automatique lors de la création d'un conteneur (si l'image n'est pas présente localement). Il est parfois cependant utile de pouvoir manipuler directement les images.

Jusqu'à maintenant nous avons utilisé l'image Nginx. Certains auront probablement remarqué que nous n'avons pas précisé de numéro de version. Par défaut, lorsque rien n'est précisé, c'est la version la plus récente de l'image qui est chargée, d'où le statut en fin de chargement :

```
Status: Downloaded newer image for nginx:latest
```

Nous allons volontairement charger l'image de version (ou plus communément de *tag*) 1.7 de Nginx.

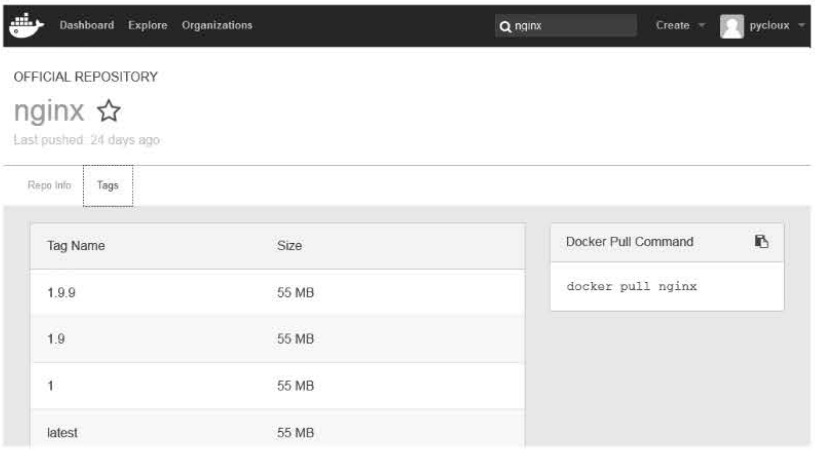

Figure 5.7 — *Tags* de l'image officielle nginx sur le Docker Hub

Reportez-vous au chapitre 2 pour savoir comment naviguer au sein du Docker Hub et lister les tags disponibles pour une image. Notez, comme nous le verrons par la suite, qu'il est aussi possible d'interroger le Docker Hub par l'intermédiaire d'une API REST.

```
$ docker pull nginx:1.7
1.7: Pulling from library/nginx
91408b29417e: Pull complete
1d57bb32d3b4: Pull complete
e6c41bf19dd4: Pull complete
16b4b2a3620a: Pull complete
b643082c0754: Pull complete
57557712fa86: Pull complete
249130fac1e4: Pull complete
23299622ed29: Pull complete
7abe60a22c0d: Pull complete
fea4feb4d44b: Pull complete
1b03d3f2a77e: Pull complete
5109569c0fed: Pull complete
Digest:
sha256:02537b932a849103ab21c75fac25c0de622ca12fe2c5ba8af2c7cb23339ee6d4
Status: Downloaded newer image for nginx:1.7
```

Cela n'est pas aussi compliqué qu'il y paraît : il suffit de suffixer le nom de l'image avec le tag que vous souhaitez télécharger grâce à la commande `pull`, soit « nginx:1.7 » pour le tag 1.7.

Créons maintenant un conteneur à partir de cette image :

```
$ docker run -p 8000:80 --name webserver17 -d nginx:1.7
```

Ouvrez maintenant un navigateur et entrez l'URL suivante : http://localhost:8000/j. En effet, Nginx, par défaut, affiche son numéro de version sur les pages d'erreur 404.

Figure 5.8 — Conteneur avec Nginx en version 1.7

Maintenant, créons une image avec la dernière version (*latest*) de Nginx, comme nous l'avons déjà fait précédemment :

```
$ docker run -p 8001:80 --name webserver19 -d nginx
```

Attention à ne pas utiliser le même port TCP que pour notre conteneur webserver17, afin d'éviter un message d'erreur déplaisant.

Ouvrez maintenant un navigateur et entrez l'URL suivante, http://localhost:8001/j, pour constater que la version de Nginx est bien la dernière (dans notre cas la 1.9). Il n'y a pas d'obstacle particulier au fait d'avoir deux versions de la même image à un même instant sur le même système.

5.3.3 Créer une image à partir d'un conteneur

Maintenant que nous avons vu comment obtenir une image depuis le Hub, voyons comment en créer une qui nous convienne et intègre des modifications par rapport à une image de base.

Créons donc un conteneur nginx :

```
$ docker run -p 8000:80 --name webserver -d nginx
```

Comme nous l'avons fait dans la section 5.2, ouvrons un terminal sur le conteneur pour modifier la page d'accueil de Nginx :

```
$ docker exec -i -t  webserver /bin/bash
root@7d4f688992b0:/# echo "Hello world" > /usr/share/nginx/html/index.html
root@7d4f688992b0:/# exit
```

À partir de ce conteneur, nous pouvons créer une image en utilisant la commande `commit`.

```
$ docker commit webserver nginxhello
908a3cb565656bca615367b5009ba8d17ddaadbd24fa77cfbb4bbfcf4c336227
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
nginxhello          latest              908a3cb56565        3 seconds ago       133.8 MB
nginx               latest              5328fdfe9b8e        3 weeks ago         133.8 MB
nginx               1.7                 5109569c0fed        8 months ago        93.4 MB
```

Nous venons de créer une nouvelle image « nginxhello », à partir de laquelle nous pouvons maintenant créer de nouveaux conteneurs, par exemple :

```
$ docker run -p 8001:80 --name webserverhello -d nginxhello
```

Un accès à l'URL http://localhost:8001 montrera que la page d'accueil de Nginx dans cette image est bien modifiée. Nous avons donc créé une variante de l'image nginx à partir d'un conteneur existant (qui lui-même s'appuie sur l'image nginx).

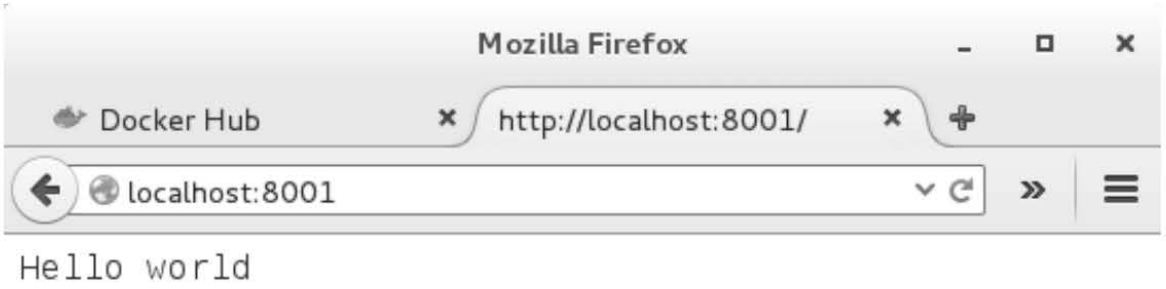

Figure 5.9 — Conteneur créé à partir de l'image « nginxhello »

Cette technique de création d'image, bien que fonctionnelle, n'est utilisée que pour des besoins de *debugging*, comme nous le verrons dans la prochaine section. On lui préfère la création d'image à partir de Dockerfile.

5.4 LE DOCKERFILE

Le Dockerfile est probablement l'une des raisons du succès de Docker dans le domaine des conteneurs. Voyons en pratique ce dont il s'agit.

5.4.1 Les risques d'une création d'image par commit

Dans ce chapitre, nous avons créé un conteneur Nginx dont nous avons changé la page par défaut (index.html). Nous avons vu une technique pour créer un conteneur à l'aide de la commande commit, c'est-à-dire en sauvegardant l'état d'un conteneur précédemment créé à partir d'une image de base, puis modifié manuellement.

Le défaut de cette technique réside dans la documentation du processus de création de l'image. Il est possible de distribuer cette image de conteneur web, mais la recette

pour le créer n'est pas normalisée. On pourrait certes la documenter, mais rien ne permettrait d'avoir l'assurance que l'image ainsi produite soit bien la résultante de la séquence d'instructions documentées. En clair, quelqu'un pourrait avoir exécuté d'autres commandes (non documentées) pour créer cette image et il ne serait pas possible de prévoir les conséquences de son usage.

C'est justement le propos des Dockerfile : décrire la création d'images de manière formelle.

5.4.2 Programmer la création des images

Dans cette section nous allons programmer un Dockerfile réalisant la création d'un conteneur ayant les mêmes caractéristiques que notre exemple Nginx.

En premier lieu, créez un répertoire vide.

Dans ce répertoire, nous allons mettre deux fichiers :

- un fichier Dockerfile ;
- un fichier index.html.

Dans ce dernier fichier, nous allons écrire le texte suivant (par exemple, à l'aide de l'éditeur vi) :

```
Un fichier qui dit "Hello"
```

Éditons maintenant le contenu du fichier Dockerfile :

```
FROM nginx:1.7
COPY index.html /usr/share/nginx/html/index.html
```

Nous en expliquerons le contenu par la suite.

Nous allons maintenant créer notre image « nginxhello » à l'aide de ce fichier Dockerfile et de la commande `build` :

```
$ docker build -t nginxhello .
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM nginx:1.7
 ---> d5acedd7e96a
Step 2 : COPY index.html /usr/share/nginx/html/index.html
 ---> a8f7c2454f7f
Removing intermediate container 4954cb1b74a7
Successfully built a8f7c2454f7f
```

Si nous lançons la commande `docker images`, nous voyons qu'une image a bien été créée il y a quelques instants :

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
nginxhello          latest              a8f7c2454f7f        About a minute ago
93.4 MB
<none>              <none>              cc0b89245d0d        15 minutes ago
93.4 MB
<none>              <none>              e3d6892f8073        3 months ago
133.8 MB
nginx               latest              4045d5284714        4 months ago
133.8 MB
nginx               1.7                 d5acedd7e96a        12 months ago
93.4 MB
```

Créons maintenant un conteneur à partir de cette image :

```
$ docker run -p 8000:80 --name webserver -d nginxhello
```

Un accès à l'URL http://localhost:8000 nous confirme que notre serveur web est actif et que sa page par défaut correspond bien au texte que nous avions mis dans le fichier index.html.

Nous venons de créer notre premier Dockerfile et, grâce à celui-ci, nous pouvons :

- créer une image permettant de produire des conteneurs Nginx modifiés à loisir ;
- distribuer la recette pour fabriquer cette image.

5.4.3 Quelques explications

Penchons-nous maintenant sur les quelques instructions que nous avons utilisées dans ce fichier Docker.

```
FROM nginx:1.7
```

Cette première commande indique à Docker que nous souhaitons créer une image à partir de l'image de base nginx en version 1.7. Il ne s'agit ni plus ni moins que d'une fonction d'héritage. Elle permet de ne pas avoir à recréer la recette qui a permis l'installation de Nginx.

Cette recette est d'ailleurs publiquement disponible puisque les Dockerfile des images sont publiés sur la partie publique du Docker Hub :

Figure 5.10 — Lien vers le Dockerfile de l'image de base nginx sur le Docker Hub

En voici d'ailleurs le contenu pour sa version 1.9.15-1 :

```
FROM debian:jessie
MAINTAINER Nginx Docker Maintainers "docker-maint@nginx.com"
ENV Nginx_VERSION 1.9.15-1~jessie
RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys
573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62 \
&& echo "deb http://nginx.org/packages/mainline/debian/ jessie nginx" >>
/etc/apt/sources.list \
&& apt-get update \
&& apt-get install --no-install-recommends --no-install-suggests -y \
ca-certificates \
nginx=${Nginx_VERSION} \
nginx-module-xslt \
nginx-module-geoip \
nginx-module-image-filter \
nginx-module-perl \
nginx-module-njs \
gettext-base \
&& rm -rf /var/lib/apt/lists/*
# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log \
&& ln -sf /dev/stderr /var/log/nginx/error.log
EXPOSE 80 443
CMD ["nginx", "-g", "daemon off;"]
```

À nouveau, nous notons que ce Dockerfile hérite lui-même de « debian:jessie » qui n'est autre qu'une version de l'image de base de la distribution Linux Debian.

La seconde instruction indique à Docker qu'il faut, lors de la création de l'image, remplacer le fichier index.html par défaut de Nginx (localisé comme nous l'avons vu auparavant dans /usr/share/nginx/html/index.html) par le fichier que nous avions posé dans notre répertoire.

```
COPY index.html /usr/share/nginx/html/index.html
```

Cette dernière instruction est l'application de la notion d'images en couches que nous avons abordée dans le chapitre 1. Elle ne modifie pas l'image de base nginx mais ajoute notre fichier par-dessus, et via le concept d'union file system (UFS), le système de fichiers résultant prend en compte notre fichier au lieu du fichier original.

La preuve est d'ailleurs visible lors de la création de notre image. Chaque étape de création (c'est-à-dire chaque ligne d'instruction) correspond à une couche.

Rappelons-nous de la première étape (*Step* 1) :

```
$ docker build -t nginxhello .
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM nginx:1.7
 ---> d5acedd7e96a
...
```

Notez l'identifiant de cette couche : `d5acedd7e96a`. Nous l'avons déjà vu auparavant car il s'agit de celui de l'image de base nginx en version 1.7 :

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
nginx               1.7                 d5acedd7e96a        12 months ago
93.4 MB
```

Notre image finale est donc constituée d'une série d'images qui peuvent être soit complètement originales, soit héritées d'une image existante par l'intermédiaire de l'instruction `FROM`.

Il est d'ailleurs possible de visualiser l'ensemble des couches d'une image donnée grâce la commande `docker history` :

```
$ docker history nginxhello
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
a8f7c2454f7f        2 days ago          /bin/sh -c #(nop) COPY
file:1378a5a0c1fa30fe5   27 B
d5acedd7e96a        12 months ago       /bin/sh -c #(nop) CMD ["nginx" "-g"
"daemon o   0 B
abdbd9163c9b        12 months ago       /bin/sh -c #(nop) EXPOSE 443/tcp
80/tcp          0 B
e551f154c24f        12 months ago       /bin/sh -c #(nop) VOLUME
[/var/cache/nginx]      0 B
a33f3c9ef1c4        12 months ago       /bin/sh -c ln -sf /dev/stderr
/var/log/nginx/   0 B
b1ce25f2cd9c        12 months ago       /bin/sh -c ln -sf /dev/stdout
/var/log/nginx/   0 B
3c5fee6e68b3        12 months ago       /bin/sh -c apt-get update &&
apt-get inst   8.38 MB
4446b7f2ac03        12 months ago       /bin/sh -c #(nop) ENV
Nginx_VERSION=1.7.12-1~   0 B
e97439968948        12 months ago       /bin/sh -c echo "deb
http://nginx.org/package   221 B
```

```
acca21866ee7        12 months ago       /bin/sh -c apt-key adv --keyserver
pgp.mit.ed   58.45 kB
6dfc7b45c331        12 months ago       /bin/sh -c #(nop) MAINTAINER Nginx
Docker Mai   0 B
325520a5dbfd        12 months ago       /bin/sh -c #(nop) CMD ["/bin/bash"]
0 B
358523180737        12 months ago       /bin/sh -c #(nop) ADD
file:20cd6318f68d34ca8e   84.96 MB
```

Là encore, vous remarquerez l'image `a8f7c2454f7f` qui correspond à notre instruction `copy`. Si vous vous reportez au journal (*log*) de l'instruction `docker build`, un peu plus haut, vous verrez que l'identifiant correspond.

En résumé

Nous avons vu dans ce chapitre ce qu'était un conteneur en pratique : comment le créer, le supprimer, le démarrer et l'arrêter. Nous avons vu que les conteneurs étaient produits à partir de moules que représentent les images. Nous avons enfin étudié comment créer de nouvelles images qui intègrent des spécificités par rapport à des images de base téléchargées d'habitude depuis le Docker Hub. Nous nous sommes aussi penchés sommairement sur la notion de Dockerfile.

À l'issue de ce chapitre, vous maîtrisez les concepts et commandes fondamentaux du moteur Docker. Il est temps de monter en puissance et de mettre en pratique ces connaissances à travers des exemples de plus en plus complexes.

TROISIÈME PARTIE

Apprendre Docker

Cette troisième partie, qui peut aussi servir de référence, vise à expliquer en détail, à travers des exemples pratiques :

- les commandes du client Docker ;
- les instructions des Dockerfile que nous avons commencé d'étudier dans le dernier chapitre de la précédente partie.

Nous avons choisi de découper cette partie en trois chapitres qui peuvent être lus de manière non séquentielle :

- le chapitre 6 montre l'usage des commandes principales du client Docker. L'objectif ici n'est pas d'être exhaustif, mais de montrer et d'expliquer les commandes essentielles (celles dont on se sert tous les jours) à travers des exemples utiles ;
- le chapitre 7 est consacré aux principales instructions des Dockerfile, c'est-à-dire celles qui sont présentes dans 95 % des fichiers Dockerfile que vous aurez à lire ;
- le chapitre 8 se penche, quant à lui, sur des commandes et instructions Dockerfile plus avancées.

Chacun de ces chapitres peut être lu individuellement. Des références croisées permettent de passer d'un chapitre à l'autre pour acquérir progressivement la maîtrise de Docker.

Nous conseillons cependant de lire les chapitres dans leur ordre naturel en commençant par le chapitre 6.

6

Prise en main du client Docker

Dans ce chapitre, nous vous présenterons les commandes usuelles du client Docker et leurs options les plus importantes.

À travers les différents chapitres précédents, nous avons déjà abordé certaines de ces commandes. Notre objectif ici est d'en présenter quelques autres par l'intermédiaire d'exemples simples et ciblés.

Le but de ce chapitre est de faire atteindre au lecteur un bon niveau de maîtrise de cette ligne de commande que nous utiliserons intensivement dans la suite de ce livre.

6.1 INTRODUCTION À LA CLI[1] DOCKER

Dans ce premier paragraphe, nous allons donner quelques informations complémentaires sur le client Docker, à savoir :

- les variables d'environnement qu'il utilise (et qui permettent d'en modifier la configuration) ;
- la structure générique des options des différents outils Docker.

1. Command Line Interface = la ligne de commande

6.1.1 Les variables d'environnement Docker

La ligne de commande Docker, qui permet l'exécution des commandes Docker, utilise un certain nombre de variables d'environnement. En modifiant ces dernières, le comportement des commandes Docker est affecté. Le tableau 6.1 décrit de manière exhaustive les variables d'environnement Docker. Nous verrons ensuite un exemple illustrant l'impact fonctionnel d'une telle variable.

Tableau 6.1 — Variables d'environnement Docker

Nom	Définition
DOCKER_API_VERSION	La version de l'API Docker à utiliser.
DOCKER_CONFIG	L'emplacement des fichiers de configuration Docker.
DOCKER_CERT_PATH	L'emplacement des certificats liés à l'authentification.
DOCKER_DRIVER	Le pilote graphique à utiliser.
DOCKER_HOST	Le démon Docker à utiliser.
DOCKER_NOWARN_KERNEL_VERSION	Si le paramètre est activé, cela empêche l'affichage d'avertissements liés au fait que la version du noyau Linux n'est pas compatible avec Docker.
DOCKER_RAMDISK	Si le paramètre est activé, alors Docker fonctionne avec un utilisateur en mémoire RAM (l'utilisateur est sauvé en RAM et non sur le disque dur).
DOCKER_TLS_VERIFY	Si le paramètre est activé, alors Docker ne permet des connexions distantes qu'avec le protocole de sécurisation TLS.
DOCKER_CONTENT_TRUST	Si le paramètre est activé, alors Docker utilise Docker Content Trust pour signer et vérifier les images. Ce comportement est équivalent à l'option `--disable-content-trust=false` lors de l'utilisation des commandes `build`, `create`, `pull`, `push` et `run`. Nous le verrons en détail dans le chapitre 8.
DOCKER_CONTENT_TRUST_SERVER	L'URL du serveur Notary à utiliser lorsque la variable d'environnement DOCKER_CONTENT_TRUST est activée.
DOCKER_TMPDIR	L'emplacement des fichiers temporaires Docker.

Pour illustrer l'impact fonctionnel d'une variable d'environnement, prenons un exemple simple : nous utiliserons la variable `DOCKER_CONFIG` qui décrit l'emplacement des fichiers de configuration du client Docker (par défaut stockés dans le répertoire *~/.docker*). Il est ainsi possible de définir un fichier de configuration particulier *config.json* qui permet de spécifier diverses options pour la ligne de commande.

Pour commencer créons un répertoire de configuration alternatif :

```
mkdir -p /home/vagrant/alt_docker_config/
```

Éditons rapidement un fichier de configuration :

```
tee /home/vagrant/alt_docker_config/config.json <<-'EOF'
{
  "psFormat":"table
{{.ID}}\\t{{.Image}}\\t{{.Command}}\\t{{.Ports}}\\t{{.Status}}"
}
EOF
```

Modifions maintenant la variable d'environnement DOCKER_CONFIG pour que cette configuration alternative s'applique automatiquement par la suite :

```
export DOCKER_CONFIG=/home/vagrant/alt_docker_config/
```

Nous spécifions ici un format alternatif pour l'affichage de la commande `ps`. Par défaut, celle-ci montre un tableau dont les colonnes sont *CONTAINER ID*, *IMAGE*, *COMMAND*, *CREATED*, *STATUS*, *PORTS* et *NAMES*. La modification que nous avons configurée changera l'affichage par défaut de la manière suivante :

```
$ docker ps
CONTAINER ID   IMAGE    COMMAND                  PORTS             STATUS
c7cc1a31d8f0   nginx    "nginx -g 'daemon off"   80/tcp, 443/tcp   Up 12
minutes
```

Il est bien sûr possible de spécifier d'autres paramètres de configuration. Pour plus de détails reportez-vous à la documentation en ligne de Docker.

6.1.2 Les options des commandes Docker

Les commandes Docker peuvent être paramétrées grâce à des options. Chaque option a une valeur par défaut, si bien que lorsque l'option n'est pas spécifiée, c'est sa valeur par défaut qui sera appliquée.

En termes d'écriture, les options sont soit représentées par une lettre (dans ce cas elles sont spécifiées par un simple tiret « - »), soit par plusieurs lettres (dans ce cas nous utilisons un double tiret « -- »). Par exemple :

```
$ docker run -i
$ docker run --detach
```

Généralement, une option définie par un simple tiret est un raccourci du nom d'une option avec un double tiret. Par exemple, les deux options suivantes sont équivalentes :

```
$ docker run -m 5M
$ docker run --memory 5M
```

Comme nous le constatons dans les exemples ci-dessus, certaines options ont une valeur (par exemple, `docker run -m 5M`) et d'autres non (par exemple, `docker run -i`). En réalité, toutes les options ont une valeur, mais dans certains cas, il n'est pas nécessaire de la spécifier ; tout dépend du type de l'option qui peut être Booléen, Valeur simple ou Valeur multiple.

Les types d'option

Booléen

Une option de type Booléen décrit si un comportement spécifique de la commande est activé ou non ; ainsi les valeurs possibles sont vrai (*true* en anglais) et faux (*false* en anglais).

Si une option de type Booléen est utilisée sans valeur, alors la valeur de l'option est vrai. Ainsi les appels suivants sont équivalents :

```
$ docker run -i
$ docker run -i=true
```

Dans la documentation des commandes Docker, les options de type Booléen sont décrites par le format suivant :

```
{nomOption}
```

{nomOption} définit le nom de l'option, par exemple `-i` ou `--interactive`. La valeur par défaut est faux.

Son modèle d'utilisation est :

```
(-|--){nomOption}[={valeur}]
```

Par exemple :

```
$ docker run -i
$ docker run -i=true
$ docker run --interactive
```

Valeur simple

Une option de type Valeur simple définit une option qui ne peut être utilisée qu'une seule fois.

Dans la documentation des commandes Docker, les options de type Valeur simple sont décrites par le format suivant :

```
{nomOption}="{valeurParDefaut}"
```

{nomOption} définit le nom de l'option, par exemple `--net`, et {valeurParDefaut} sa valeur par défaut, par exemple `bridge`. À noter que la valeur par défaut peut être vide.

Son modèle d'utilisation est :

```
(-|--){nomOption}(=| )("{valeur}"|{valeur})
```

Par exemple :

```
$ docker run -m 5M
$ docker run --memory 5M
```

Valeur multiple

Une option de type Valeur multiple définit une option qui peut être utilisée plusieurs fois ; on dit que l'option est cumulative.

Dans la documentation des commandes Docker, les options de type Valeur multiple sont décrites par le format suivant :

```
{nomOption}=[]
```

{nomOption} définit le nom de l'option, par exemple `-p`.

Son modèle d'utilisation est identique à celui du type Valeur simple, à l'exception du fait qu'il peut être répété autant que nécessaire. Par exemple :

```
docker run -p 35022:22 -p 35080:80
```

La bonne pratique

Comme cela est indiqué dans les modèles ci-dessus (=|), la valeur d'une option, quand elle est donnée, peut soit être placée après un signe égal, soit après un espace ; ainsi les deux écritures suivantes sont équivalentes :

```
$ docker run -m=5M
$ docker run -m 5M
```

Toutefois, pour des raisons de lisibilité et de conformité avec le monde Unix, il convient de n'utiliser le sigle égal que pour les options de type Booléen. Ainsi nous aurions :

```
$ docker run -i=true
$ docker run -m 5M
```

Toujours selon les modèles, la valeur d'une option peut être encapsulée par des guillemets. Cependant, nous éviterons cette écriture, sauf si la valeur en question contient des espaces.

6.2 LES COMMANDES SYSTÈME

Dans cette section, nous allons présenter ce que nous avons regroupé sous le terme de « commandes système ». Il s'agit des commandes qui sont relatives au Docker Engine dans son ensemble, soit pour piloter le démon, soit pour obtenir des informations sur son fonctionnement.

6.2.1 docker daemon

docker daemon [OPTIONS]

C'est la commande permettant de contrôler le démon Docker. Nous l'avons déjà abordée dans le chapitre 3.

Évidemment, dans la plupart des cas, le démon est démarré automatiquement par un gestionnaire de système (comme systemd pour CentOS ou REHL). Par exemple, sous CentOS, la définition du service est stockée sous /usr/lib/systemd/system/docker.service :

```
$ cat /usr/lib/systemd/system/docker.service
[Unit]
Description=Docker Application Container Engine
Documentation=https://docs.docker.com
After=network.target docker.socket
Requires=docker.socket

[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues
still
# exists and systemd currently does not support the cgroup feature set
required
# for containers run by docker
ExecStart=/usr/bin/docker daemon -H fd://
...
```

Nous voyons que celle-ci fait appel à la commande `docker daemon`. Cette dernière prend en paramètre de très nombreuses options relatives à différents sujets :

- authentification (plugins spécifiques et headers HTTP à ajouter aux requêtes REST du client Docker) ;
- paramétrage réseau (certificats, serveur DNS à utiliser, règles de forwarding IP, etc.) que nous aborderons dans les chapitres 9 et 11 ;
- labels (dont nous parlerons dans le chapitre 11 avec Swarm) qui permettent de qualifier un hôte pour permettre l'application automatique de règles de déploiement dans le cadre d'un cluster ;
- etc.

6.2.2 docker info

docker info [OPTIONS]

Cette commande permet de visualiser la configuration du démon. Nous l'avons notamment utilisée dans le chapitre 4 pour visualiser la configuration du stockage Docker. Elle fournit aussi des informations sur l'état du moteur, comme le nombre de conteneurs, le nombre d'images, les plugins installés, etc.

6.2.3 docker version

docker version [OPTIONS]

Affiche la version du client et du serveur (démon) Docker (pour peu que ce dernier soit installé) :

```
$ docker version
Client:
 Version:      1.11.1
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   5604cbe
 Built:        Wed Apr 27 00:34:42 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.11.1
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   5604cbe
 Built:        Wed Apr 27 00:34:42 2016
 OS/Arch:      linux/amd64
```

6.2.4 docker stats

docker stats [OPTIONS]

Une instruction qui donne (un peu à la manière d'un `top` sous Unix) des informations sur les conteneurs exécutés par le moteur :

```
$ docker stats
CONTAINER           CPU %               MEM USAGE / LIMIT     MEM %
NET I/O               BLOCK I/O           PIDS
b7f691e522b8        0.00%               7.815 MB / 1.929 GB   0.41%
5.213 kB / 4.956 kB   6.791 MB / 0 B      0
```

Cette commande accepte notamment une option `-a` qui permet de visualiser tous les conteneurs (sous-entendu, même ceux qui sont à l'arrêt) et une option `--no-stream` qui permet de n'afficher qu'un instantané (un *snapshot* en anglais) des statistiques qui ne sont donc pas rafraîchies en temps réel.

6.2.5 docker ps

docker ps [OPTIONS]

Cette commande permet de lister les conteneurs actifs (par défaut) ou l'ensemble des conteneurs avec l'option `-a`.

La commande est aussi à la base de l'astuce permettant d'effacer tous les conteneurs et que nous avons vue dans le chapitre 5 :

```
$ docker rm $(docker ps -a -q)
```

Le modificateur `-q` limite l'affichage de la commande à l'identifiant du conteneur, ce qui permet à la commande `rm` de détruire tous les conteneurs (actifs ou inactifs du fait du modificateur `-a`).

Il est aussi possible de modifier le format d'affichage de la commande à l'aide du modificateur `--format` (comme nous l'avons montré en début de chapitre) ou de filtrer les conteneurs à l'aide de l'option `--filter`.

6.2.6 docker events

docker events [OPTIONS]

Une commande qui permet d'afficher les événements qui se produisent sur le bus d'événements Docker. Une fois activée, la commande (un peu à la manière d'un `tail -f` sur un fichier) affiche les événements du moteur Docker concernant les conteneurs, les images, les volumes et le réseau.

Par exemple, si nous lançons un conteneur (ici nommé « tiny_bassi ») et que nous lui faisons subir une séquence d'actions (ici `pause`, `unpause`, `stop` puis `start`) :

```
$ docker ps
CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS  NAMES
b7f691e522b8  nginx  "nginx -g 'daemon off"  3 minutes ago  Up 3 minutes
0.0.0.0:80->80/tcp, 443/tcp  tiny_bassi
$ docker pause b7f691e522b8
b7f691e522b8
$ docker unpause b7f691e522b8
b7f691e522b8
```

```
$ docker stop b7f691e522b8
b7f691e522b8
$ docker start b7f691e522b8
```

Nous obtiendrons l'affichage suivant (à la condition de lancer la commande sur un autre terminal avant d'effectuer les actions) :

```
$ docker events
2016-05-09T13:16:24.768480481+02:00 container pause
b7f691e522b80b69ca4a01d9f267eb31717f95b4e029c9068620322c0e81ac12 (image=nginx,
name=tiny_bassi)
2016-05-09T13:16:29.567098215+02:00 container unpause
b7f691e522b80b69ca4a01d9f267eb31717f95b4e029c9068620322c0e81ac12 (image=nginx,
name=tiny_bassi)
2016-05-09T13:16:34.573241379+02:00 container kill
b7f691e522b80b69ca4a01d9f267eb31717f95b4e029c9068620322c0e81ac12 (image=nginx,
name=tiny_bassi, signal=15)
2016-05-09T13:16:34.619735796+02:00 container die
b7f691e522b80b69ca4a01d9f267eb31717f95b4e029c9068620322c0e81ac12 (exitCode=0,
image=nginx, name=tiny_bassi)
2016-05-09T13:16:34.767099031+02:00 network disconnect
cdb6ff80d4f9ff518d907de596fbd6ed437a9d247c1edb8a6b0b4d24ef4d583d
(container=b7f691e522b80b69ca4a01d9f267eb31717f95b4e029c9068620322c0e81ac12,
name=bridge, type=bridge)
2016-05-09T13:16:34.813257612+02:00 container stop
b7f691e522b80b69ca4a01d9f267eb31717f95b4e029c9068620322c0e81ac12 (image=nginx,
name=tiny_bassi)
2016-05-09T13:16:38.957730296+02:00 network connect
cdb6ff80d4f9ff518d907de596fbd6ed437a9d247c1edb8a6b0b4d24ef4d583d
(container=b7f691e522b80b69ca4a01d9f267eb31717f95b4e029c9068620322c0e81ac12,
name=bridge, type=bridge)
2016-05-09T13:16:39.166071968+02:00 container start
b7f691e522b80b69ca4a01d9f267eb31717f95b4e029c9068620322c0e81ac12 (image=nginx,
name=tiny_bassi)
```

On remarquera les événements réseau qui sont implicitement déclenchés par l'arrêt et le démarrage du conteneur.

À noter que, comme nous l'avons déjà expliqué, toutes les commandes du client Docker correspondent à des API REST. Il est ainsi possible à des applications tierces de s'enregistrer sur l'API suivante qui va streamer les événements au format JSON :

```
GET /events
```

6.2.7 docker inspect

docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE...]

Cette commande permet de récupérer sous un format JSON des métadonnées relatives à une image ou un conteneur. Les données ainsi rendues sont très variées. Par exemple, pour un conteneur : configuration des volumes, paramètres de démarrage du conteneur, image, configuration réseau, etc.

6.3 CYCLE DE VIE DES CONTENEURS

Cette section étudie les commandes permettant d'influer sur le cycle de vie d'un conteneur. Nous avons déjà vu un nombre important de ces commandes dans le chapitre 5. Le lecteur pourra donc s'y reporter pour voir leur effet sur divers exemples pratiques.

6.3.1 docker start

docker start [OPTIONS] CONTAINER

Cette commande permet de démarrer un conteneur qui aura été préalablement créé (mais pas encore démarré), à l'aide de la commande `create`, ou arrêté.

6.3.2 docker stop

docker stop [OPTIONS] CONTAINER

Cette commande, qui permet de stopper un conteneur, a aussi déjà été étudiée dans le chapitre 5. Elle accepte notamment une option `-t` qui permet de spécifier un nombre de secondes à attendre avant de tenter un `kill`.

6.3.3 docker kill

docker kill [OPTIONS] CONTAINER

Cette commande, déjà évoquée au chapitre 5, permet de forcer l'arrêt d'un conteneur (un peu à la manière du célèbre `kill -9` pour un processus Unix).

6.3.4 docker restart

docker restart [OPTIONS] CONTAINER

Une commande qui combine un `stop` et un `start` en séquence. Elle accepte comme option le même paramètre `-t` que la commande `stop`.

6.3.5 docker pause et docker unpause

docker pause [OPTIONS] CONTAINER

docker unpause [OPTIONS] CONTAINER

La commande `pause` permet de *freezer* un conteneur. Attention, le conteneur n'est pas arrêté ; il est suspendu, c'est-à-dire qu'il ne fait plus rien. Il peut être réactivé en utilisant la commande `unpause`.

Cette fonctionnalité s'appuie sur celles de CGroups dans la gestion des processus.

6.3.6 docker rm

docker rm [OPTIONS] CONTAINER

La commande permet de détruire un conteneur (qui doit au préalable avoir été arrêté). Si le conteneur n'est pas arrêté, une erreur est affichée à moins d'utiliser l'option `-f` (pour forcer) qui va d'abord déclencher une commande `kill` avant d'exécuter la commande `rm`.

6.3.7 docker wait

docker wait [OPTIONS] CONTAINER

Cette commande intéressante permet de bloquer l'invite de commande tant que le conteneur passé en paramètre n'est pas arrêté. En effet, par défaut, la commande `stop` rend la main immédiatement, le conteneur pouvant mettre plusieurs minutes à s'arrêter ensuite.

Grâce à cette commande, il est par exemple possible de s'assurer qu'un conteneur est effectivement arrêté avant d'effectuer une autre action. Par exemple :

```
$ docker stop mon-conteneur
$ docker wait mon-conteneur
```

Cette dernière commande rend la main dès que le conteneur mon-conteneur est correctement arrêté.

6.3.8 docker update

docker update [OPTIONS] CONTAINER

La commande `update` a un lien avec les commandes `create` et `run`. Elle permet en effet de modifier les paramètres d'un conteneur alors que celui-ci est démarré. Attention, il ne s'agit pas de modifier des paramètres de volume ou réseau, mais uniquement la configuration des ressources allouées au conteneur, par exemple :

- `-cpuset-cpus` pour le nombre de CPU ;
- `-m` pour la mémoire maximale disponible.

Notons tout de même le modificateur `--restart` qui permet de modifier la politique de redémarrage automatique du conteneur (dont les valeurs peuvent être `no`, `on-failure[nombre de tentatives]`, `always`, `unless-stopped`).

6.3.9 docker create et docker run

docker create [OPTIONS] IMAGE [COMMAND] [ARG...]

docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

`create` et `run` sont des commandes sœurs dans la mesure où un `run` correspond à un `create` suivi d'un `start`. Rien d'étonnant à ce qu'elles partagent la plupart de leurs options.

Ces deux commandes prennent pour paramètre l'identifiant ou le nom d'une image et, optionnellement, une commande à exécuter ou des paramètres qui compléteront l'ENTRYPOINT défini dans le Dockerfile qui a présidé à la création de l'image (voir le chapitre 7) :

```
$ docker create -name=webserver nginx
#Crée un conteneur nginx prêt à être démarré
$ docker start webserver
#Démarre le conteneur précédemment créé
```

La liste des options est longue, au point que `run` a sa propre entrée dans la documentation Docker. On peut catégoriser les options de cette commande comme suit :

- les paramètres relatifs au contrôle des ressources (mémoire, CPU, IO) allouées au conteneur (ce sont les mêmes paramètres que pour la commande `update`) ;
- les paramètres relatifs aux volumes (que nous avons déjà vus sommairement dans le chapitre 5, et que nous reverrons plus en détail dans le chapitre 7 et les suivants) :

- `--volumes-from` qui permet à un conteneur d'hériter des volumes d'un autre conteneur (en général un *data container*, c'est-à-dire un conteneur qui n'a pour autre objectif que de référencer des volumes),
- `--volume-driver` qui permet de changer le gestionnaire de volume (cf. chapitre 1) qui permet à Docker de déléguer la gestion des volumes à un système tiers externe,
- `-v` qui permet de spécifier les montages de volumes selon divers arguments que nous verrons en détail dans le chapitre 7,
- les paramètres relatifs à la gestion du réseau (qui seront étudiés dans les chapitres 9, 10 et 11) et au mappage des ports IP (que nous verrons dans le prochain chapitre en lien avec l'instruction EXPOSE du Dockerfile).

Il existe d'autres paramètres dont les plus importants sont détaillés ci-dessous :

- `--entrypoint=""` qui permet de surcharger l'instruction ENTRYPOINT du conteneur spécifiée dans le Dockerfile. Nous verrons le but de cette commande dans le chapitre 7 ;
- `-t` et `-i`, la plupart du temps combinés, qui permettent d'ouvrir un pseudo terminal sur le conteneur. Ceci permet donc de travailler avec le conteneur en mode interactif. Par exemple :

```
$ docker run -t -i centos:7 /bin/bash
#ouvre une ligne de commande dans un conteneur de type centos 7
```

- `-d` (évidemment incompatible avec `-i`) qui lance le conteneur en mode démon. La commande `run` rend la main et le conteneur tourne en tâche de fond ;
- `-w` qui permet de spécifier un répertoire de travail différent de celui spécifié avec l'instruction Dockerfile WORKDIR (nous aborderons cette instruction dans le chapitre 8) ;
- `--privileged` permet de lancer un conteneur en mode privilégié. Cela signifie que ce dernier est alloué avec toutes les permissions du kernel (root) et peut faire tout ce que l'hôte peut faire. Ce type d'allocation de droits est à utiliser avec de grandes précautions, mais est nécessaire notamment dans le cas de Docker dans Docker que nous étudions dans le chapitre 10 ;
- `--name`, vu dans le chapitre 5, permet de nommer un conteneur (en contournant le système d'allocation de nom par défaut de Docker) ;
- `--rm` va garantir que si le conteneur est stoppé, il est automatiquement détruit avec la commande `rm`. Attention, cette option n'est pas compatible avec le mode démon ;
- `--log-driver` permet de changer le gestionnaire de logs pour ce conteneur (voir la commande `logs` ci-après).

6.4 INTERACTIONS AVEC UN CONTENEUR DÉMARRÉ

Les diverses commandes de ce paragraphe permettent d'interagir avec un conteneur démarré.

6.4.1 docker logs

docker logs [OPTIONS] CONTAINER

Cette commande permet d'afficher les logs du conteneur.

Il s'agit d'une propriété intéressante des conteneurs Docker. Par défaut, Docker va mettre à disposition de la commande `logs` ce qui est écrit dans le STDOUT pour le processus racine (celui qui est à la base de l'arborescence des processus du conteneur).

Prenons la ligne suivante :

```
$ docker run -d --name loop php php -r "while(true){echo \"Log something every
2 sec\n\";sleep(2);}"
```

Elle n'est certes pas très élégante, mais compréhensible :

- la commande `run` va créer puis démarrer un conteneur ;
- ce conteneur est créé à partir de l'image de base php[1] ;
- le paramètre `-d` indique que le conteneur doit être lancé en tâche de fond ;
- le nom de ce conteneur sera loop ;
- il va exécuter le mini script PHP `while(true){echo \"Log something every 2 sec\n\";sleep(2);}` qui va afficher une chaîne de caractères toutes les deux secondes.

Une fois ce conteneur démarré, nous constatons qu'il s'exécute :

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
6d54638a92a5        php                 "php -r 'while(true){"   3 seconds ago
Up 2 seconds                            loop
```

La commande `logs` va permettre de visualiser les logs du conteneur :

```
$ docker logs --tail 4 loop
Log something every 2 sec
Log something every 2 sec
Log something every 2 sec
Log something every 2 sec
```

1. https://hub.docker.com/_/php/

La commande `logs` admet deux options utiles :

- `--follow` ou `-f` permet de fonctionner en mode `tail -f` en affichant les logs en continu (au fur et à mesure qu'ils sont produits) ;
- `--tail X` permet d'afficher X lignes de logs en partant de la fin.

Notez qu'il est possible de changer le pilote de logs (*logs driver*) d'un conteneur. Docker en supporte nativement plusieurs et notamment des pilotes comme fluentd[1] qui permettent de mettre en place des systèmes de collecte, de centralisation et d'indexation des logs.

Voyons ici un exemple dans lequel nous remplaçons le pilote par défaut par *syslog* (le log système de Linux) :

```
$ docker run -d --log-driver syslog --name loop php php -r "while(true){echo
\"Log something every 2 sec\n\";sleep(2);}"
```

Maintenant, nous pouvons (sous CentOS) visualiser le conteneur du syslog :

```
$ sudo tail -f /var/log/messages
May 10 00:21:00 localhost docker/94250ba05053[23043]: Log something every 2
sec
May 10 00:21:02 localhost docker/94250ba05053[23043]: Log something every 2
sec
May 10 00:21:04 localhost docker/94250ba05053[23043]: Log something every 2
sec
May 10 00:21:06 localhost docker/94250ba05053[23043]: Log something every 2
sec
```

6.4.2 docker exec

docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

Cette commande permet d'exécuter une commande à l'intérieur d'un conteneur démarré. L'un des cas d'usage très commun consiste à lancer un terminal *bash* dans un conteneur Linux (pour aller fouiller dedans).

Lançons par exemple un conteneur Nginx (que nous avons déjà vu dans le chapitre 5) que nous nommerons « webserver » :

```
$ docker run -d --name webserver nginx
```

Il est possible de se connecter à l'intérieur du conteneur démarré à l'aide de la commande :

1. http://www.fluentd.org/

```
$docker exec -t -i webserver /bin/bash
root@e80fd51f2119:/# ls
bin  boot  devetc  home  liblib64  media  mnt  optproc  root  run  sbin  srv
sys  tmp  usr  var
root@e80fd51f2119:/# exit
```

La commande `exit` met fin à l'exécution du terminal et rend la main sans pour autant affecter le conteneur qui continue son exécution.

Il est évidemment possible de lancer des commandes sans les modificateurs `-i` et `-t` qui seront alors exécutées mode non interactif.

6.4.3 docker attach

docker attach [OPTIONS] CONTAINER

Cette commande permet de s'attacher à un conteneur démarré pour visualiser et éventuellement interagir avec le processus racine du conteneur (si ce dernier le permet).

Lançons un conteneur de type serveur web :

```
$ docker run -d -p 8000:80 --name webserver nginx
```

Il est ensuite possible de se connecter au conteneur pour en visualiser le contenu. Dans notre cas, ouvrez un navigateur et faites quelques appels à l'URL http://localhost:8000 pour générer des logs :

```
$ docker attach --sig-proxy=false  webserver
172.17.0.1 - - [09/May/2016:22:39:30 +0000] "GET / HTTP/1.1" 304 0 "-"
"Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0" "-"
172.17.0.1 - - [09/May/2016:22:39:31 +0000] "GET / HTTP/1.1" 304 0 "-"
"Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0" "-"
```

L'option `--sig-proxy=false` permet d'éviter que la commande *Ctrl-C* que vous utiliserez pour quitter l'attachement ne mette fin au processus et, de ce fait, arrête le conteneur.

6.4.4 docker rename

docker rename OLD_NAME NEW_NAME

Cette commande, comme son nom l'indique, permet de renommer un conteneur.

6.4.5 docker cp

docker cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH

docker cp [OPTIONS] SRC_PATH|- CONTAINER:DEST_PATH

Cette commande permet de copier des fichiers entre un conteneur démarré et le système de fichiers de l'hôte.

Reprenons notre exemple de serveur web Nginx :

```
$ docker run -d -p 8000:80 --name webserver nginx
```

Ouvrez un navigateur sur http://localhost:8000 et constatez que la page par défaut du serveur s'affiche. Nous allons maintenant la remplacer par le texte « Hello World » :

```
$ docker cp webserver:/usr/share/nginx/html/index.html .
$ echo "Hello World" > index.html
$ docker cp index.html webserver:/usr/share/nginx/html/index.html
```

La séquence de commandes ci-dessus effectue les opérations suivantes :

- copie le fichier index.html du serveur web (dans le conteneur) sur la machine hôte ;
- remplace le contenu de ce fichier ;
- copie le fichier modifié depuis l'hôte vers le conteneur.

6.4.6 docker diff

docker diff [OPTIONS] CONTAINER

Cette commande permet de visualiser les changements effectués sur les fichiers d'un conteneur, qu'il s'agisse :

- d'ajouts (A) ;
- de modifications (C) ;
- d'effacements (D).

L'exemple suivant nécessite deux terminaux.

Dans le premier, nous allons ouvrir un conteneur CentOS en mode interactif :

```
$ docker run -t -i --name exemple centos:7 /bin/bash
[root@57538f45c5d2 /]#
```

Une fois le conteneur ouvert, l'invite du terminal bash est affichée. Ouvrez un nouveau terminal et entrez la commande suivante :

```
$ docker diff exemple
```

Même si vous attendez longtemps, rien ne s'affiche car le conteneur n'a subi aucune modification par rapport à son image de base.

Revenons à notre premier terminal et entrez la ligne suivante :

```
[root@57538f45c5d2 /]# echo "Hello" > test.txt
[root@57538f45c5d2 /]#
```

Lançons alors à nouveau la même commande `diff` dans notre second terminal :

```
$ docker diff exemple
A /test.txt
```

Nous pouvons constater que l'ajout du fichier test.txt a bien été détecté.

6.4.7 docker top

docker top [OPTIONS] CONTAINER

Cette commande affiche le résultat de la commande `top` effectuée à l'intérieur d'un conteneur actif.

```
$ docker run -d --name webserver nginx
94aaaa9db232626baaf8dc6179899528bfb9e9bdd5e40b19e3e976292875eb2d
$ docker top webserver
UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
root                32259               32243               0
01:14               ?                   00:00:00            nginx: master
process nginx -g daemon off;
104                 32271               32259               0
01:14               ?                   00:00:00            nginx: worker
process
```

6.4.8 docker export

docker export [OPTIONS] CONTAINER

Cette commande permet d'exporter l'ensemble du système de fichiers d'un conteneur dans un fichier tar.

```
$docker run -d --name webserver nginx
$docker export webserver > test.tar
```

Rarement utilisée en pratique, cette commande ne prend vraiment de sens qu'avec `docker import` qui permet de créer une image à partir de ce type d'export.

Elle peut aussi être utilisée pour faire tourner un conteneur avec un autre moteur qui ne serait pas directement compatible avec le format d'image Docker, par exemple RunC[1].

6.4.9 docker port

docker port [OPTIONS] CONTAINER

Cette commande permet de visualiser les ports exposés par un conteneur :

```
$ docker run -d --name webserver -p 8000:80 nginx
84515d2c56807e6f428d0a044c2f2f277c42eac0bed5d65f929a1abc80289c37
$ docker port webserver
80/tcp -> 0.0.0.0:8000
```

6.5 COMMANDES RELATIVES AUX IMAGES

Nous allons maintenant effectuer un recensement des commandes qui permettent de manipuler des images Docker.

6.5.1 docker build

docker build [OPTIONS] PATH | URL

Il s'agit d'une commande que nous avons abordée sommairement à la fin du chapitre 5 et que nous allons utiliser intensivement dans le chapitre 7.

Cette commande permet de construire une image à partir d'un Dockerfile.

Voici quelques-unes de ses options les plus courantes :

- `-t` permet de définir le *tag* d'une image, c'est-à-dire son nom. Il est de convention de nommer les images en combinant le nom de leur auteur (ou organisation) à un nom unique (un peu à la manière des repository GitHub)[2] ;

1. https://runc.io/
2. Par exemple, l'image https://hub.docker.com/r/ingensi/play-framework/ de la société Ingensi qui fournit une image Play framework prête à l'emploi.

- `-f` permet de spécifier un Dockerfile possédant un nom différent du standard usuel ;
- `--rm` et `--force-rm` permettent d'effacer les images intermédiaires produites lors du processus de build (uniquement en cas de succès pour `--rm` ou systématiquement pour `--force-rm`).

À noter que la commande accepte aussi les mêmes options de configuration des ressources que pour les commandes `run` ou `create`.

6.5.2 docker commit

docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

Nous avons déjà utilisé cette commande dans le chapitre 5. Celle-ci permet de créer une image à partir d'un conteneur démarré.

6.5.3 docker history

docker history [OPTIONS] IMAGE

Nous avons utilisé cette commande dans le chapitre 1 pour expliquer la structure en couches des images Docker. Elle permet en effet de visualiser les différentes couches d'une image et, en regard, les instructions du Dockerfile qui ont été utilisées pour les produire.

6.5.4 docker images

docker images [OPTIONS] [REPOSITORY[:TAG]]

Cette commande permet de lister les images du cache local. Le paramètre `-a` permet de visualiser toutes les images intermédiaires (les couches) qui, par défaut, ne sont pas visibles. Il est aussi possible d'appliquer des filtres via le paramètre `--filter`.

6.5.5 docker rmi

docker rmi [OPTIONS] IMAGE [IMAGE...]

Cette commande permet d'effacer une image du registre local.

Elle est souvent combinée avec la commande `docker images`, comme dans l'exemple suivant :

```
$ docker rmi $(docker images --quiet --filter "dangling=true")
```

Il s'agit ici d'effacer les images *dangling* (littéralement « pendantes » c'est-à-dire dépourvues de *tag* donc de nom). Ce cas arrive lorsque vous construisez plusieurs fois la même image (en phase de développement). Comme il n'est possible dans le cache local Docker (comme dans un registry) de n'avoir qu'une seule image pour un tag donné, l'image précédente est dépossédée de son nom. Il est donc utile de lancer cette commande pour faire le ménage.

6.5.6 docker save et docker load

docker save [OPTIONS] IMAGE [IMAGE...]

docker load [OPTIONS]

Ces deux commandes permettent d'importer des images depuis le cache local Docker ou de les exporter vers le cache local. Elles permettent donc de transférer des images entre hôtes sans faire appel à un registry :

```
$ docker save centos:7 > centos.tar
$ docker load -i=centos.tar
```

6.5.7 docker import

docker import [OPTIONS] file|URL|- [REPOSITORY[:TAG]]

Cette commande fonctionne avec la commande `export`. Elle permet d'importer le système de fichiers d'un conteneur (préalablement exporté) en tant qu'image. À ce titre, son but est assez proche de celui de la commande `commit`.

6.6 INTERACTIONS AVEC LE REGISTRY

Les commandes de cette section sont consacrées aux interactions entre le Docker Engine d'un hôte et les registries qu'il s'agisse du Docker Hub ou d'un autre registry public ou privé.

Notons que Google (https://cloud.google.com/container-registry/) et Amazon (https://aws.amazon.com/ecr/) ont annoncé la création de leurs propres offres de registry de conteneur.

6.6.1 docker login

docker login [OPTIONS] [SERVER]

Cette commande permet de s'authentifier auprès d'un registry en vue d'effectuer des opérations de pull et de push. Elle accepte deux options :

- `--username, -u` : le nom d'utilisateur d'un compte du registry ;
- `--password, -p` : le mot de passe de ce compte.

6.6.2 docker logout

docker logout [OPTIONS] [SERVER]

Sans surprise, cette commande déconnecte le démon du registry auquel il est connecté (suite à l'usage de la commande `login`).

6.6.3 docker push

docker push [OPTIONS] NAME[:TAG]

Cette commande permet d'importer une image du cache local dans un registry.

6.6.4 docker pull

docker pull [OPTIONS] NAME[:TAG|@DIGEST]

La commande `pull` permet de télécharger une image dans le cache local de l'hôte.

Notons que cette commande est par défaut implicitement lancée par `run` ou `create` lorsque l'image utilisée n'est pas présente dans le cache local.

Il est possible d'utiliser le nom de l'image ou le *digest*, un identifiant globalement unique et non mutable.

6.6.5 docker search

docker search [OPTIONS] NAME[:TAG|@DIGEST]

Cette commande permet de rechercher dans un registry. La recherche se fait sur le nom de l'image. Il est possible d'appliquer un filtre sur le nombre d'étoiles attribuées à l'image à l'aide du paramètre `-s`.

```
$ docker search -s=3 redmine
NAME                      DESCRIPTION
STARS     OFFICIAL   AUTOMATED
sameersbn/redmine                                                    194
[OK]
redmine                   Redmine is a flexible project management w...   176
[OK]
bitnami/redmine           Bitnami Docker Image for Redmine               7
[OK]
74th/redmine-all-in-one   Redmine includes hosting SVN & Git , backl...   3
[OK]
```

6.6.6 docker tag

docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/] [USERNAME/] NAME[:TAG]

Cette commande permet de créer un nom alternatif pour une image. Par exemple si vous disposez de l'image « monimage » en version 1.7, il est possible de créer un alias alternatif pour cette image avec la commande suivante :

```
$ docker tag monimage:1.7 monimage:latest
```

Si d'aventure quelqu'un lançait la commande `rmi` sur ce nouveau nom, l'image ne serait pas effacée pour autant, seul l'alias serait supprimé :

```
$ docker rmi monimage:latest
Untagged: monimage:latest
```

Ceci permet notamment de s'assurer que l'alias *latest* (couramment utilisé pour des images Docker) pointe toujours vers la version la plus récente d'une image.

6.7 RÉSEAU ET VOLUMES

Ces commandes sont apparues avec Docker 1.9 et apportent deux avancées majeures :

- le nouveau modèle réseau Docker dont nous allons parler en détail dans les chapitres 9 et 11 ;
- les volumes nommés qui permettent de s'affranchir de la technique des *data containers* dont nous verrons des exemples dans les chapitres 9 et 10.

Comme ces commandes seront abordées en détail dans la suite de cet ouvrage, nous n'en ferons ici qu'une présentation sommaire.

6.7.1 Les commandes docker network

Tableau 6.2 — Commandes docker network

Commande	Objet
docker network create	Permet de créer un réseau Docker. Cette commande prend notamment en paramètre `--driver` qui permet de spécifier le type de réseau souhaité (par défaut bridge).
docker network connect	Cette commande permet de connecter un conteneur à un réseau.
docker network disconnect	Cette commande permet de déconnecter un conteneur d'un réseau.
docker network inspect	Une fonction d'inspection du réseau dont nous verrons l'utilité dans le chapitre 9.
docker network ls	Une commande qui liste les réseaux disponibles. Par défaut, trois réseaux sont systématiquement définis : none, host et bridge.
docker network rm	La commande qui permet de détruire des réseaux existants (sauf évidemment nos trois réseaux prédéfinis).

6.7.2 Les commandes docker volume

Tableau 6.3 — Commandes docker volume

Commande	Objet
docker volume create	Une commande qui permet de créer un volume qu'il sera ensuite possible d'associer à un ou plusieurs conteneurs. Cette commande prend en paramètre `--driver` qui permet de spécifier le driver utilisé pour ce volume. Il existe aujourd'hui des plugins Docker permettant de s'appuyer sur des systèmes de stockage tiers (en lieu et place d'une simple persistance sur l'hôte).
docker volume inspect	Une commande pour visualiser des métadonnées relatives à un volume.
docker volume ls	La commande qui permet de lister les volumes disponibles.
docker volume rm	La commande qui permet d'effacer des volumes. Attention, une fois détruit, les données associées à un volume sont perdues définitivement. Il n'est cependant pas possible d'effacer un volume utilisé par un conteneur.

En résumé

Ce chapitre, qui peut être utilisé comme référence, nous a permis d'aborder la grande majorité des commandes Docker disponibles. En pratique, il est assez rare d'avoir recours à l'ensemble de ces commandes tous les jours, mais les connaître rend parfois de précieux services.

Il est maintenant temps d'aborder plus en détail la conception de Dockerfile que nous n'avons fait qu'effleurer dans le chapitre 5.

7

Les instructions Dockerfile

Ce chapitre a pour but de décrire les instructions d'un fichier Dockerfile, leurs paramètres et leurs particularités, et comment elles doivent être utilisées en pratique.

La première section introduira la façon d'écrire une instruction, appelée modèle, puis la deuxième section détaillera les instructions principales. Les instructions plus techniques seront étudiées dans le chapitre 8.

7.1 LES MODÈLES D'INSTRUCTION

7.1.1 Introduction

Un modèle d'instruction Dockerfile décrit simplement comment cette dernière doit être utilisée. On peut le comparer à la signature d'une méthode dans un langage de programmation. Ainsi, pour chaque instruction, il existe au moins un modèle, mais plusieurs sont souvent disponibles, chaque modèle couvrant un cas d'utilisation spécifique.

De manière générale, les modèles d'instruction sont simples à comprendre. Il existe pourtant une particularité : certaines instructions (par exemple CMD ou ENTRYPOINT) contiennent plusieurs modèles dont le résultat peut sembler similaire, mais qui comportent quelques subtiles différences. Ces modèles sont décrits par deux formats particuliers : terminal et exécution. Nous nous attarderons sur ces deux formats afin de déterminer dans quels cas l'un ou l'autre doit être utilisé.

Tableau 7.1 — Formats de modèles

Format	Exemple de modèle	Exemple d'application	Commande résultante
terminal	CMD <command>	CMD ping localhost	/bin/sh -c "ping localhost"
exécution	CMD ["executable", "param1", "param2"]	CMD ["ping", "localhost"]	ping localhost

L'option `-c` du binaire `/bin/sh` (dernière colonne de la première ligne) définit que la commande qui suit est décrite par une chaîne de caractères, dans notre cas `"ping localhost"`.

7.1.2 Terminal ou exécution ?

Le format terminal (*shell* en anglais) permet d'exécuter un binaire (par exemple, la commande `ping`) au travers d'un terminal applicatif, autrement dit la commande sera préfixée par `"/bin/sh -c"`. Par exemple, l'instruction `CMD ping localhost` sera exécutée par :

```
/bin/sh -c "ping localhost"
```

Premier point important et obligatoire dans le cas terminal : le binaire `/bin/sh` doit être disponible dans l'image pour que la commande puisse être exécutée. Une image minimale pourrait ne pas le contenir si bien que cette commande échouerait.

Le deuxième problème est plus technique : comme cela est mentionné plus haut, l'exécution du `ping` est encapsulée dans l'exécution du `/bin/sh`, ce qui en soit n'est pas un problème, mais le devient dans un contexte Docker, notamment lors de l'arrêt d'un conteneur. Pour comprendre le problème, il faut tout d'abord expliquer les principes suivants :

1. Le PID du processus démarré par le point d'entrée d'un conteneur (instructions CMD et/ou ENTRYPOINT) est le 1.
2. Lorsqu'un conteneur actif est arrêté par la commande `docker stop`, alors cette dernière n'essaie d'arrêter proprement (SIGTERM) que le processus dont le PID est 1. S'il existe d'autres processus, alors ils seront arrêtés brutalement (SIGKILL) après le délai accepté pour l'arrêt d'un conteneur (par défaut 10 secondes).
3. Lorsque le binaire /bin/sh reçoit un signal lui demandant de s'arrêter, alors il s'arrête proprement, mais ne retransmet par le signal à ses enfants, c'est-à-dire les autres binaires qu'il encapsule (dans notre exemple `ping`).

Reprenons maintenant l'exemple du `ping`. Nous aurons donc deux processus démarrés dans le conteneur :

1. `/bin/sh` avec le PID 1 (car représentant le point d'entrée du conteneur).
2. `ping` avec un autre PID.

Si on souhaite arrêter le conteneur (`docker stop`), le résultat sera le suivant :

1. `/bin/sh` s'arrêtera proprement car il a le PID 1.
2. `ping` ne s'arrêtera pas proprement car il n'a pas le PID 1, et `/bin/sh` ne retransmet pas le signal d'arrêt à ses enfants. Il sera ainsi arrêté brutalement à la fin du délai d'arrêt du conteneur.

Notre exemple utilise le binaire `ping`, ce qui, au final, n'est pas un réel problème s'il s'arrête brutalement ; par contre, si le processus en question est plus critique, par exemple un serveur HTTP (Nginx, Apache...), la situation serait différente : les requêtes du serveur HTTP en attente de traitement ne seraient pas exécutées, ce qui pourrait entraîner des comportements étranges au niveau des clients, ou pire, une perte de données.

Vous souhaitez en savoir plus sur l'arrêt des conteneurs ? Un article décrivant comment arrêter proprement un conteneur Docker et disponible via le lien ci-dessous :
https://www.ctl.io/developers/blog/post/gracefully-stopping-docker-containers/

Le format exécutable (*exec* en anglais) permet d'exécuter un binaire sans intermédiaire. Par exemple, l'instruction CMD ["ping", "localhost"] sera simplement exécutée par `ping localhost`.

Ainsi on remarque que le terminal applicatif `/bin/sh` n'est plus utilisé, et que le binaire à exécuter (`ping` dans notre exemple) est directement appelé, ce qui règle les deux problèmes du format terminal :

- une image minimale qui ne contiendrait pas le binaire `/bin/sh` ;
- un arrêt brutal d'un processus supplémentaire à celui défini par le point d'entrée du conteneur.

Toutefois, le format terminal a l'avantage d'être plus lisible et confortable à utiliser, si bien qu'on l'utilisera dans les cas suivants :

- l'image contient le binaire `/bin/sh` ;
- il n'existe pas de processus supplémentaire demandant un arrêt propre.

Dans les autres cas, on utilisera le format exécution. Plus concrètement, le tableau 7.2 décrit les règles d'utilisation des formats.

Tableau 7.2 — Règles d'utilisation des formats

Contexte d'utilisation	Format
Cas de tests	terminal*
Commande se terminant directement après son exécution (par exemple : ls ou mkdir)	terminal*
Commande lançant un processus (par exemple : ping ou httpd)	exécution
Autres cas	exécution

* pour autant que l'image contienne le binaire /bin/sh, sinon exécution.

7.1.3 Les commentaires

Un fichier Dockerfile peut contenir des commentaires, c'est-à-dire des lignes qui ne seront par interprétées. Pour cela, il suffit d'utiliser le caractère « # » :

```
#L'image source est centos 7
FROM centos:7
#On ajoute fichier test1 dans tmp
COPY test1 /tmp/
#On liste le dossier tmp du conteneur
CMD ls /tmp
```

7.2 LES INSTRUCTIONS D'UN DOCKERFILE

Cette section décrit les instructions principales disponibles pour un fichier Dockerfile. L'objectif ici est de fournir les connaissances nécessaires à l'élaboration d'un Dockerfile afin d'éviter les erreurs usuelles. Pour cela, de nombreux principes sont illustrés par des exemples.

Pour rappel, un fichier Dockerfile représente simplement le descripteur d'une image Docker : après l'élaboration d'un Dockerfile, on construit l'image avec `docker build` puis on démarre un conteneur basé sur cette image avec `docker run`.

7.2.1 FROM

L'instruction FROM permet de spécifier l'image Docker parente à utiliser. Ainsi, l'image résultante du Dockerfile sera basée sur cette dernière à laquelle seront ajoutés les blocs d'images produits par les instructions suivantes.

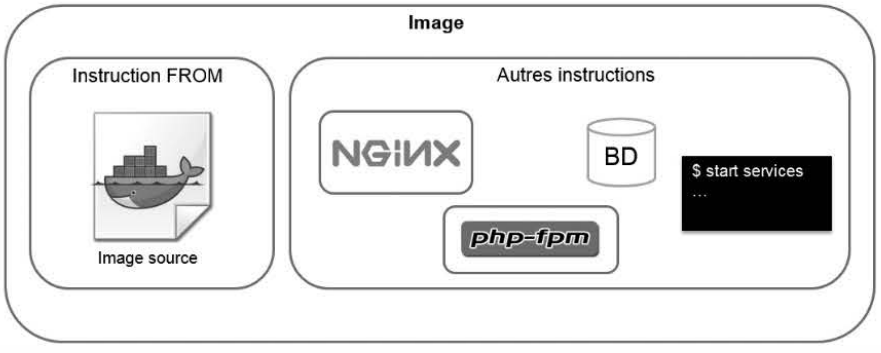

Figure 7.1 — Décomposition d'une image Docker

La figure 7.1 représente l'image résultante d'un Dockerfile. Nous pouvons constater que l'instruction FROM définit l'image source, et que les autres instructions sont exécutées à partir de cette image (installation d'une application, exécution d'un script...).

Le modèle de l'instruction FROM est :

FROM <image>[:<tag>]

<image> représente l'image Docker source. Cette image doit être disponible soit localement, soit dans le Docker Hub ou dans votre registry privé.

<tag> représente la version de l'image Docker source. S'il n'est pas spécifié, alors la dernière version (*latest*) est utilisée.

Par exemple :

```
FROM centos
FROM centos:7
```

FROM est la seule instruction obligatoire d'un Dockerfile ; de plus, elle doit être placée en début de fichier (précédée d'éventuels commentaires).

Un même Dockerfile peut contenir plusieurs fois l'instruction FROM afin de créer plusieurs images. Prenons un exemple simple dont le but est de créer deux images : la première affiche un message « Hello world » à son exécution et la deuxième « Bonjour à tous ».

```
FROM centos:7
CMD echo "Hello world"
FROM centos:7
CMD echo "Bonjour à tous"
```

Construisons ensuite les images depuis le répertoire contenant notre Dockerfile :

```
$ docker build .
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM centos:7
---> ce20c473cd8a
Step 2 : CMD echo "Hello world"
---> Running in 1e094f7b1a01
---> cfef3a2fa477
Removing intermediate container 1e094f7b1a01
Step 3 : FROM centos:7
---> ce20c473cd8a
Step 4 : CMD echo "Bonjour à tous"
---> Running in 07dc136e24b6
---> 5b7f2d5028bd
Removing intermediate container 07dc136e24b6
Successfully built 5b7f2d5028bd
```

Dans notre cas, la première image a pour identifiant `cfef3a2fa477` et la deuxième `5b7f2d5028bd`.

Démarrons un conteneur avec la première image :

```
$ docker run cfef3a2fa477
Hello world
```

Et maintenant un conteneur avec la deuxième image :

```
$ docker run 5b7f2d5028bd
Bonjour à tous
```

Cet exemple démontre que l'utilisation de plusieurs instructions FROM dans un même Dockerfile est plus ou moins équivalente à l'utilisation de plusieurs fichiers avec les mêmes instructions. Cela nous empêche cependant de spécifier des options différentes lors du `docker build` pour chaque image (par exemple, spécifier le nom de l'image). L'utilisation d'un même Dockerfile pour construire plusieurs images s'utilise dans des cas très particuliers, par exemple lors de la construction de plusieurs images fonctionnellement très proches, c'est-à-dire contenant très peu d'instructions, et héritant d'une même image source : on se rend bien compte que ces cas sont anecdotiques, et on appliquera la règle générale suivante :

> Un fichier Dockerfile doit contenir exactement une et une seule instruction FROM.

Pour conclure, revenons sur l'utilisation du modèle sans spécifier la version de l'image de base (le *tag*), par exemple FROM centos. La probabilité que l'image ne puisse pas être construite augmente avec le temps : pour rappel, en omettant la version, c'est la dernière qui est utilisée. Ainsi, entre la création initiale du Dockerfile et la date actuelle, la version de l'image source va très certainement évoluer. En reconstruisant un Dockerfile dont la version de l'image source aurait changé, notamment si une nouvelle version majeure a été publiée, alors les instructions suivantes peuvent ne plus fonctionner ; ça serait le cas, par exemple, lors de l'installation d'un paquet existant sur une ancienne distribution de Linux, mais plus disponible sur la version courante. On peut donc retenir la règle suivante :

> Dans l'instruction FROM d'un fichier Dockerfile, il faut toujours spécifier la version de l'image source.

7.2.2 MAINTAINER

L'instruction MAINTAINER permet de spécifier la personne en charge de maintenir le Dockerfile, généralement son auteur.

Le modèle est :

> **MAINTAINER <name>**

`<name>` est une chaîne de caractères représentant l'auteur, par exemple son prénom et son nom.

Par exemple :

```
MAINTAINER John Doe
```

L'instruction MAINTAINER va renseigner le champ *Author* de l'image. Cette information est visible lorsqu'on exécute un `docker inspect` sur l'image. Prenons un simple exemple pour constater cela :

```
FROM centos:7
MAINTAINER John Doe
```

Construisons et inspectons l'image :

```
$ docker build -t maintainer .
$ docker inspect --format='{{json .Author}}' maintainer
"John Doe"
```

L'option `--format` utilisée dans le `docker inspect` ci-dessus permet de focaliser le résultat : dans notre cas, nous souhaitons visualiser l'auteur de l'image au format JSON (car simple à lire), et nous utilisons ainsi le format {{json .Author}}. « maintainer » représente le nom de l'image dont nous souhaitons visualiser des informations. À noter qu'il est possible d'exécuter un `docker inspect` sur une image ou sur un conteneur.

7.2.3 RUN

L'instruction RUN (à ne pas confondre avec la commande `docker run`) permet d'exécuter des commandes utilisées généralement pour construire l'image. On peut représenter l'ensemble des instructions RUN comme l'écart entre l'image source et l'image résultante.

Il existe deux modèles pour l'instruction RUN :

RUN <command>

RUN ["executable", "param1", "param2"]

Le premier est orienté terminal et le second exécution.

Par exemple :

```
RUN mkdir /tmp/test
RUN ["/bin/sh", "-c", "mkdir /tmp/test"]
```

Les deux exemples ci-dessus produiront le même résultat, c'est-à-dire créer un dossier test sous /tmp. Pour rappel, l'option `-c` dans le second exemple permet de spécifier que la commande (dans notre cas `mkdir /tmp/test`) sera décrite par une chaîne de caractères.

> Les instructions RUN servent à construire l'image ; ainsi, les commandes à exécuter seront finies, c'est-à-dire inactives après leur exécution. Il est donc préférable d'utiliser le format terminal qui est plus aisé à écrire et à comprendre.

Il est possible de combiner plusieurs commandes pour une même instruction RUN, simplement en les séparant par des points-virgules (;), par exemple :

```
RUN mkdir /tmp/test1; mkdir /tmp/test2
```

Cet exemple crée deux dossiers (test1 et test2) sous /tmp.

Pour gagner en lisibilité, il est parfois préférable d'écrire les commandes sur plusieurs lignes. Pour cela, il suffit de terminer une ligne par un antislash (\) ; en reprenant le dernier exemple, nous aurions :

```
RUN mkdir /tmp/test1;\
mkdir /tmp/test2
```

L'utilité majeure de l'instruction RUN est l'installation des fonctionnalités particulières à l'image : par exemple, si un conteneur a pour but l'exploitation d'une application web PHP, alors l'image Docker devra probablement contenir un serveur web (par exemple Apache) ainsi que PHP et éventuellement une base de données. Le Dockerfile doit donc décrire l'installation des services et applications nécessaires. Dans la mesure où ce livre utilise comme image source un linux CentOS 7, le gestionnaire de paquets natif utilisé est *Yellowdog Updater Modified* (yum). L'installation de tout paquet se fera alors grâce à la commande `yum`. Lors de son exécution, une confirmation d'action est demandée à l'utilisateur : étant donné que l'utilisateur ne peut agir lors de la construction d'une image, il faut forcer les actions grâce à l'option `-y`. Par exemple, pour l'installation d'Apache nous aurions :

```
RUN yum install -y httpd
```

Pour pouvoir installer un paquet, on s'attend à un certain état de l'image, c'est-à-dire une version à jour et surtout cohérente avec d'autres paquets. Par cohérente, nous entendons un état connu et qui ne diffère pas d'une exécution à l'autre. Ainsi, il faut s'assurer de l'état avant l'installation du paquet souhaité ; pour cela, nous incluons généralement la commande `yum update` à celle d'installation, par exemple :

```
RUN yum update -y && yum install -y httpd
```

Il est important de ne pas utiliser l'instruction seule `RUN yum update -y`, car un problème de cache Docker peut survenir, entraînant une incohérence d'images. La prochaine section illustre ce problème grâce à un exemple.

Quand le cache s'en mêle

Comme expliqué précédemment, l'utilisation de la commande `RUN yum update -y` peut entrainer un problème d'incohérence.

Pour l'illustrer, nous utiliserons un dépôt fictif de paquets, illustré par l'image suivante :

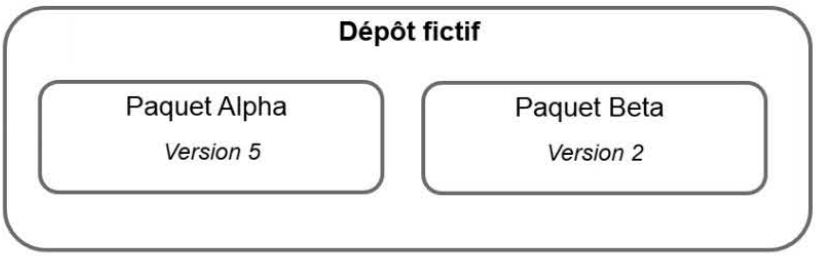

Figure 7.2 — Dépôt fictif

Imaginons à présent une image, appelée AB, avec un paquet Alpha en version 3 et un paquet Beta en version 2 ; cela signifie que lors de la construction de l'image ces deux paquets étaient disponibles dans ces versions.

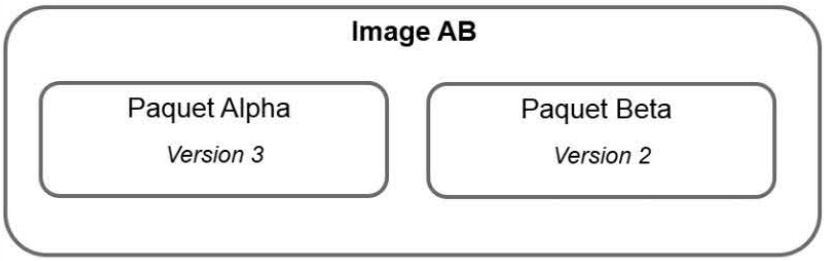

Figure 7.3 — Dépôt fictif plus ancien

À noter que seuls les paquets utiles à l'exemple sont représentés, tout autre élément de l'image étant volontairement ignoré afin de faciliter la compréhension.

On souhaite construire une image XYZ à jour dont l'image source est AB. On écrit alors naïvement le Dockerfile suivant :

```
FROM AB
RUN yum update -y
```

En le construisant nous obtenons :

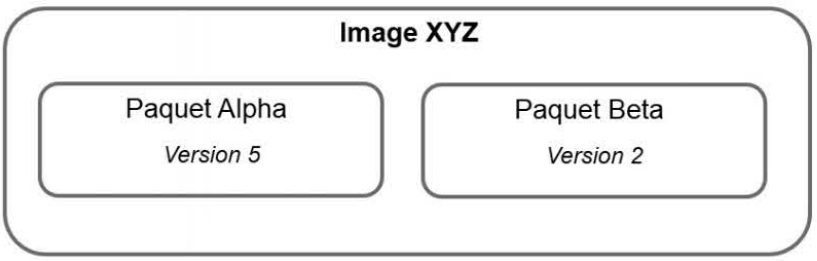

Figure 7.4 — Image basée sur le dépôt le plus récent

L'image XYZ contient donc logiquement le paquet Alpha en version 5, car il a été mis à jour suite à la commande `RUN yum update -y`.

Quelque temps plus tard, les paquets du dépôt fictif sont mis à jour ; de plus, un nouveau paquet Gamma est disponible, qui est dépendant du paquet Alpha dans sa dernière version :

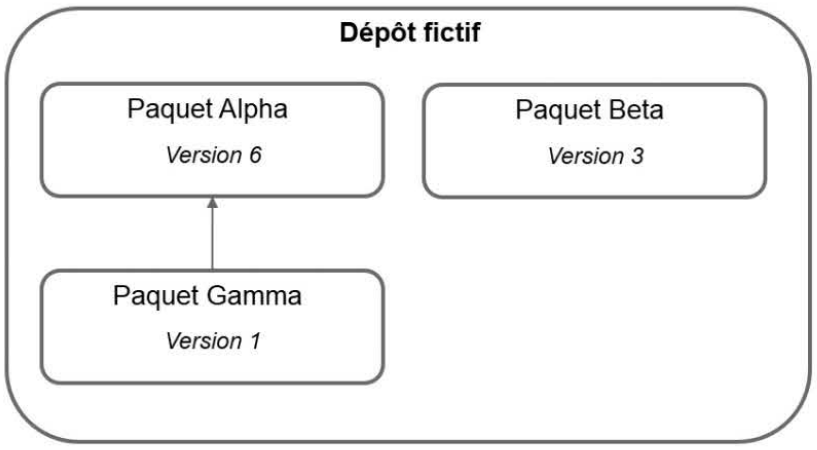

Figure 7.5 — Mise à jour du dépôt

Nous souhaitons maintenant ajouter le paquet Gamma à l'image XYZ, et nous modifions donc le Dockerfile ainsi :

```
FROM AB
RUN yum update -y
RUN yum install -y gamma
```

En construisant à nouveau le Dockerfile sur la même machine, la commande `RUN yum update -y` n'est pas exécutée, mais lue depuis le cache (car elle n'a pas été modifiée) ; ainsi, à ce stade, ni le paquet Alpha ni le paquet Beta ne sont mis à jour. Alpha sera mis en version 6 grâce à l'instruction suivante (où seuls les paquets nécessaires à Gamma sont mis à jour), Beta restant lui en version 2 :

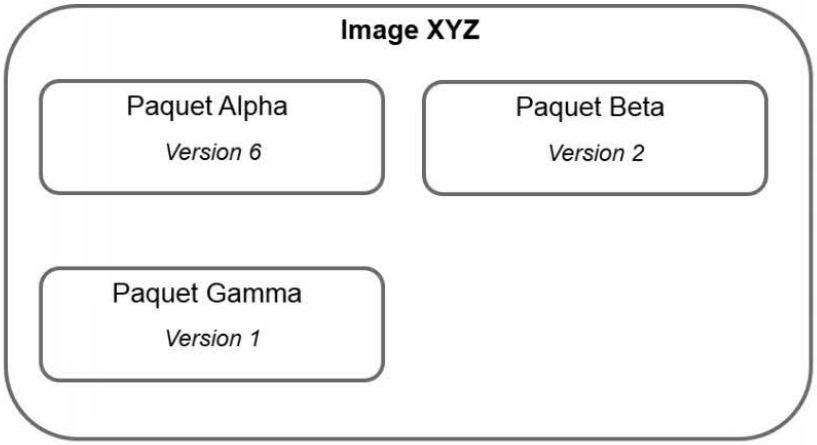

Figure 7.6 — Première image

En construisant le Dockerfile depuis une autre machine, on se rend compte aisément qu'Alpha et Beta seraient respectivement en version 6 et 3 après construction ; ainsi, le résultat devient incohérent d'une machine à l'autre :

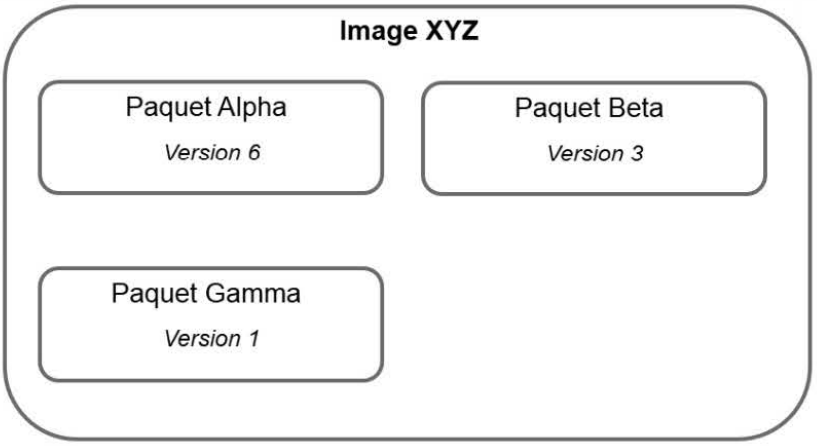

Figure 7.7 — Seconde image : même Dockerfile, résultat différent

Conclusion

Lorsque nous souhaitons installer un paquet, il faut toujours s'assurer que les paquets existants sont à jour : nous utilisons pour cela la commande `yum update -y`. Toutefois nous devons garantir que cette dernière est exécutée lors de chaque construction impliquant l'installation d'un paquet supplémentaire. Ainsi, nous devons combiner la commande `yum update -y` avec les `yum install -y` nécessaires, si bien que nous obtenons par exemple :

```
RUN yum update -y && yum install -y \
httpd \
mariadb-server \
php php-mysql
```

7.2.4 CMD

L'instruction CMD permet d'exécuter une commande (par exemple, une commande Unix) au démarrage du conteneur résultant.

Il est important de comprendre que cette instruction n'est pas jouée lors de la construction de l'image, mais bien lors de l'exécution d'un conteneur associé. Ainsi, son rôle est de fournir le contexte d'utilisation du conteneur, par exemple le démarrage d'un service.

> Un Dockerfile peut contenir plusieurs instructions CMD, cependant seule la dernière instruction du fichier sera exécutée au démarrage du conteneur. Ainsi il convient de n'avoir, au maximum, qu'une seule instruction CMD dans un Dockerfile.

Il existe trois modèles pour l'instruction CMD :

CMD <command>

CMD ["executable", "param1", "param2"]

CMD ["param1", "param2"]

Le premier est orienté terminal et les autres exécution (sur le même principe que l'instruction RUN).

Prenons un peu de temps pour décrire la troisième forme : nous constatons qu'aucun exécutable n'est spécifié, cela signifiant que l'instruction seule est incomplète ; elle doit donc être combinée avec l'instruction ENTRYPOINT (dont le principe de fonctionnement sera donné dans la prochaine section) pour qu'elle devienne fonctionnelle. Illustrons la combinaison d'une instruction CMD avec une instruction ENTRYPOINT par un exemple :

Soit le Dockerfile suivant :

```
FROM centos:7
ENTRYPOINT ["/bin/ping","-c","5"]
CMD ["localhost"]
```

L'instruction ENTRYPOINT représente l'exécution de la commande `ping` cinq fois. Par contre, elle ne décrit pas la destination du `ping` ; autrement dit, en exécutant

la commande `ping -c 5` depuis un terminal, nous obtiendrons une erreur du type « mauvaise utilisation de la commande ping ».

| L'option `-c` de la commande `ping` signifie *count*, soit le nombre souhaité d'exécutions.

L'instruction CMD représente uniquement le paramètre décrivant la destination de la commande `ping`. Ainsi, mis bout à bout, nous obtiendrons la commande suivante `ping -c 5 localhost`.

Construisons maintenant notre image et démarrons un conteneur :

```
$ docker build -t my-ping .
$ docker run my-ping
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.083 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.117 ms
64 bytes from localhost (127.0.0.1): icmp_seq=3 ttl=64 time=0.082 ms
64 bytes from localhost (127.0.0.1): icmp_seq=4 ttl=64 time=0.071 ms
64 bytes from localhost (127.0.0.1): icmp_seq=5 ttl=64 time=0.058 ms
--- localhost ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 0.058/0.082/0.117/0.020 ms
```

Le résultat représente l'équivalent de l'exécution `ping -c 5 localhost` depuis un terminal.

Mais alors pourquoi utiliser deux instructions (ENTRYPOINT et CMD) alors qu'une seule donnerait le même résultat ? La réponse à cette question réside dans la surcharge du point d'entrée d'un conteneur.

En effet, ce dernier décrit l'effet des instructions CMD et/ou ENTRYPOINT au démarrage du conteneur. La surcharge représente quant à elle la modification de cet effet. Illustrons ces principes par deux exemples, le premier pour l'instruction CMD et le deuxième pour ENTRYPOINT.

Soit le Dockerfile suivant :

```
FROM centos:7
CMD ping localhost
```

Pour surcharger l'instruction CMD, nous spécifions simplement la commande comme paramètre d'exécution (un build préalable est bien sûr nécessaire) :

```
$ docker run my-ping-cmd ls
```

Ainsi `ping localhost` sera remplacé par la commande `ls` (lister le contenu du répertoire courant).

Continuons notre exemple avec le Dockerfile suivant :

```
FROM centos:7
ENTRYPOINT ping localhost
```

Pour surcharger l'instruction ENTRYPOINT, nous devons utiliser l'option `--entrypoint`, par exemple :

```
$ docker run --entrypoint ls my-ping-entrypoint
```

La première constatation est qu'il est plus simple de surcharger une instruction CMD que ENTRYPOINT ; ainsi, nous privilégierons généralement cette façon de faire. Ensuite, nous remarquons qu'il n'est pas toujours souhaité de surcharger la totalité d'une commande. Dans ces derniers exemples, nous pourrions ne vouloir surcharger que le nom de l'hôte (ici localhost) du `ping` et non toute la commande : c'est dans ce sens que le cumul des instructions CMD et ENTRYPOINT devient utile. Reprenons l'exemple my-ping : imaginons que nous souhaitons surcharger localhost par "google.com" ; l'exécution serait alors :

```
$ docker run my-ping google.com
PING google.com (77.153.128.182) 56(84) bytes of data.
64 bytes from 182.128.153.77.rev.sfr.net (77.153.128.182): icmp_seq=1 ttl=47
time=22.0 ms
64 bytes from 182.128.153.77.rev.sfr.net (77.153.128.182): icmp_seq=2 ttl=47
time=21.7 ms
64 bytes from 182.128.153.77.rev.sfr.net (77.153.128.182): icmp_seq=3 ttl=47
time=21.6 ms
64 bytes from 182.128.153.77.rev.sfr.net (77.153.128.182): icmp_seq=4 ttl=47
time=22.7 ms
64 bytes from 77.153.128.182: icmp_seq=5 ttl=47 time=21.5 ms
--- google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 21374ms
rtt min/avg/max/mdev = 21.515/21.927/22.701/0.468 ms
```

7.2.5 ENTRYPOINT

L'instruction ENTRYPOINT permet, tout comme l'instruction CMD, d'exécuter une commande au démarrage du conteneur résultant.

Elle possède les caractéristiques suivantes qui sont équivalentes à celles de l'instruction CMD :

- ENTRYPOINT n'est pas jouée lors de la construction de l'image, mais lors du démarrage du conteneur ;
- un Dockerfile peut contenir plusieurs instructions ENTRYPOINT, cependant seule la dernière instruction du fichier sera exécutée.

> Si un Dockerfile contient l'instruction ENTRYPOINT au format exécution et l'instruction CMD (quel que soit son format), alors le contenu de l'instruction CMD (représentant dans ce cas des paramètres) sera ajouté la fin de l'instruction ENTRYPOINT.
> Si, par contre, un Dockerfile contient l'instruction ENTRYPOINT au format terminal et l'instruction CMD (quel que soit son format), alors l'instruction CMD sera ignorée.

Il existe deux modèles pour l'instruction ENTRYPOINT :

ENTRYPOINT <command>

ENTRYPOINT ["executable", "param1", "param2"]

Le premier est orienté terminal et le deuxième exécution (sur le même principe que l'instruction RUN).

> Lors du démarrage du conteneur (`docker run`), il est possible de surcharger l'instruction ENTRYPOINT grâce à l'option `--entrypoint`, par exemple, `docker run --entrypoint ls my-app`.

Format exécution

Prenons un exemple d'utilisation de l'instruction ENTRYPOINT au format exécution. Soit le Dockerfile suivant :

```
FROM centos:7
ENTRYPOINT ["ping","google.com"]
```

Nous construisons l'image :

```
$ docker build -t ping-google-exec .
```

Puis nous démarrons un conteneur :

```
$ docker run --rm --name test ping-google-exec
```

L'option `--rm` permet de supprimer automatiquement le conteneur dès qu'il se termine. Elle est généralement utilisée dans des cas de tests afin d'éviter la multiplication de création de conteneurs. Dans notre exemple, cette option est particulièrement intéressante pour deux raisons :

- D'abord, en supprimant le conteneur, son nom sera libéré et ainsi réutilisable par un autre conteneur (c'est-à-dire une autre instance de l'image) ; nous pourrons ainsi exécuter la commande `docker run` à plusieurs reprises en préservant le nom du conteneur.
- Ensuite, notre conteneur représente un simple test sans état : il ne démarre aucun service et n'offre aucune utilité particulière après son démarrage ; ainsi, sa suppression automatique évite une multiplication de conteneurs inutiles et procure donc un gain d'espace évident.

L'option `--name` permet de nommer le conteneur résultant ; dans notre cas, son nom sera « test ». On peut par la suite utiliser ce nom dans d'autres commandes (par exemple `docker exec`).

Nous souhaitons maintenant afficher les conteneurs actifs afin de voir les commandes en cours (colonne COMMAND du résultat).

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
409dfc57a7a6        ping-google-exec    "ping google.com"   About a minute ago
Up About a minute                       test
```

Comme attendu, la commande en cours du conteneur est `ping google.com`.

Il est également possible d'exécuter une commande directement dans un conteneur actif ; pour cela, on utilise la commande `docker exec`. Si, par exemple, nous souhaitons lister les processus actifs (`ps aux`) dans le conteneur « test », nous utilisons :

```
$ docker exec test ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.2  0.0  17176  1088 ?        Ss   09:26   0:00 ping
google.com
root         6  0.0  0.0  35888  1452 ?        Rs   09:26   0:00 ps aux
```

Le premier processus correspond au `ping` et le second au `ps` lui-même.

Nous pouvons maintenant arrêter le conteneur grâce à la commande `docker stop` :

```
$ docker stop test
```

Format terminal

Reprenons le même exemple, mais en utilisant le format terminal. Soit le Dockerfile suivant :

```
FROM centos:7
ENTRYPOINT ping google.com
```

Nous construisons ensuite l'image, puis nous démarrons un conteneur :

```
$ docker build -t ping-google-shell .
$ docker run --rm --name test ping-google-shell
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
c844052a05e9        ping-google-shell   "/bin/sh -c 'ping goo"   15 seconds ago
Up 14 seconds                           test
```

La commande en cours est cette fois-ci différente : `/bin/sh -c 'ping google.com'`

Il s'agit du `ping google.com`, précédé par `/bin/sh -c`. C'est le fameux problème d'encapsulation de l'exécution d'un binaire (`ping` dans notre cas) par un terminal applicatif (`/bin/sh -c`) lors de l'utilisation du format terminal. Vous trouverez plus

d'information à ce sujet dans la section « Terminal ou exécution ? », au début de ce chapitre.

7.2.6 EXPOSE

L'instruction EXPOSE décrit les ports du conteneur que ce dernier écoute. Un port exposé n'est pas directement accessible, et il devra ensuite être mappé (soit automatiquement, soit manuellement) avec un port de l'hôte exécutant le conteneur.

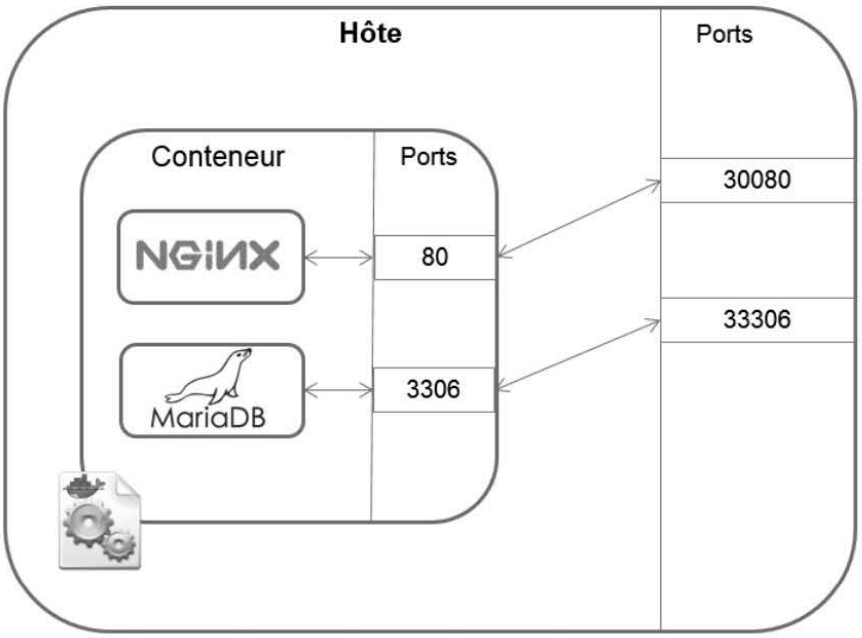

Figure 7.8 — Mappage de ports entre un conteneur et un hôte

La figure 7.8 illustre sommairement le mappage de ports entre un conteneur et un hôte :

- le conteneur contient les processus Nginx et MariaDB fonctionnant respectivement avec les ports 80 et 3306 ;
- le conteneur expose les ports 80 et 3306 afin de les rendre potentiellement accessibles avec l'hôte (autrement dit le Dockerfile utilisé pour la construction de l'image contient l'instruction EXPOSE 80 3306) ;
- le port 80 du conteneur est mappé avec le port 30080 de l'hôte ;
- le port 3306 du conteneur est mappé avec le port 33306 de l'hôte ;
- l'hôte expose les ports 30080 et 33306 ;
- un client ayant accès à l'hôte pourra ainsi accéder au processus Nginx et MariaDB du conteneur.

Le modèle de l'instruction EXPOSE est :

```
EXPOSE <port> [<port>...]
```

Par exemple :

```
EXPOSE 80 22
```

Cet exemple indique simplement que les ports 80 et 22 du conteneur sont exposés ou, autrement dit, écoutés.

Le mappage d'un port exposé du conteneur avec un port de l'hôte peut se faire automatiquement (dans ce cas le port hôte est choisi par Docker dans une plage prédéfinie) ou manuellement (dans ce cas le port hôte est simplement spécifié). Illustrons ces deux méthodes par des exemples. Soit le Dockerfile suivant dont l'image résultante est un conteneur accessible via SSH :

```
FROM centos:7
RUN yum update -y && yum install -y \
openssh-server \
passwd
RUN mkdir /var/run/sshd
RUN ssh-keygen -t rsa -f /etc/ssh/ssh_host_rsa_key -N ''
RUN useradd user
RUN echo -e "pass\npass" | (passwd --stdin user)
EXPOSE 22
CMD ["/usr/sbin/sshd", "-D"]
```

`openssh-server` est un ensemble d'outils permettant de contrôler à distance une machine et de lui transférer des fichiers grâce au protocole OpenSSH. `passwd` est un outil permettant d'assigner ou de modifier un mot de passe à un utilisateur.

Parcourons rapidement les différentes instructions auxquelles vous êtes à présent habitué :

- le dossier `/var/run/sshd` est requis par `sshd` (démon SSH) pour la gestion de privilèges ;
- la commande `ssh-keygen -t rsa -f /etc/ssh/ssh_host_rsa_key -N ''` permet la génération d'une clé RSA pour SSH ;
- les commandes `useradd user` et `echo -e "pass\npass" | (passwd --stdin user)` permettent de créer un nouvel utilisateur "user" dont le mot de passe est "pass" ;
- finalement, l'instruction CMD démarre le démon SSH. L'option `-D` spécifie que le démon SSH ne doit pas être exécuté en tâche de fond, mais au premier plan afin qu'il soit toujours accessible.

Pour finir, nous construisons l'image (cette dernière sera utilisée ci-après) :

```
$ docker build -t ssh .
```

Mappage automatique

Pour mapper automatiquement tous les ports exposés dans le Dockerfile, il suffit de spécifier l'option `-P` au démarrage du conteneur :

```
$ docker run -P -d ssh
```

L'option `-d`, quant à elle, permet d'exécuter le conteneur en tâche de fond (ainsi l'invite de commande reste disponible). L'identifiant du conteneur est affiché.

> Docker choisit un port aléatoire dans une plage. Cette dernière est définie à partir du fichier /proc/sys/net/ipv4/ip_local_port_range.

Pour découvrir les ports alloués, on peut utiliser la commande `docker ps` et se référer à la colonne PORTS :

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                   NAMES
1d8e61685fe8        ssh                 "/usr/sbin/sshd -D"   26 seconds ago
Up 26 seconds       0.0.0.0:32772->22/tcp   condescending_booth
```

Dans le résultat ci-dessus, nous constatons que le port 32772 de l'hôte est mappé avec le port 22 du conteneur (ce dernier étant le port exposé).

Il est également possible d'utiliser la commande `docker port <Identifiant du conteneur>` ; dans ce cas, nous obtenons le résultat suivant :

```
$ docker port 5b7f2d5028bd
22/tcp -> 0.0.0.0:32772
```

Mappage manuel

Le mappage manuel d'un port exposé peut se comprendre sous deux angles différents :

1. d'une part, spécifier manuellement quels ports doivent être mappés parmi ceux exposés dans le Dockerfile ;
2. d'autre part, spécifier manuellement les ports hôtes à utiliser afin de les mapper avec les ports exposés.

Ces deux cas sont couverts par l'option `-p` de la commande `docker run`. Voici son principe de fonctionnement :

```
-p [<port hôte>:]<port exposé>
```

La spécification du port exposé est obligatoire (dans notre exemple, 22), ce qui n'est pas le cas du port hôte. Il est donc possible grâce à l'option `-p` de spécifier quels sont les ports exposés sans préciser un port hôte à utiliser. Par contre, dès que l'on

souhaite spécifier les ports hôtes à mapper, alors on est contraint de préciser également quels sont les ports exposés correspondant, ce qui au final est logique.

> L'option `-p` est cumulative, dans le sens où il est possible d'avoir plusieurs fois l'option `-p` pour la commande `docker run`.

Reprenons l'exemple ssh. Imaginons que nous souhaitons spécifier manuellement le port exposé (soit le 22) en le mappant automatiquement avec un port hôte ; la commande serait :

```
$ docker run -d -p 22 ssh
```

Et maintenant, si nous souhaitons mapper manuellement le port 22, et cela en spécifiant le port hôte, par exemple 35022, nous aurions :

```
$ docker run -d -p 35022:22 ssh
```

Comme mentionné plus haut, l'option `-p` requiert le port exposé ; nous le constatons aisément dans les deux exemples ci-dessus avec le port 22. Ainsi, il y a une redondance avec l'instruction EXPOSE du Dockerfile. En réalité, dans le cadre de l'utilisation de l'option `-p`, l'instruction EXPOSE n'est plus utilisée formellement ; elle devient uniquement informelle : elle permet une lecture rapide et aisée des ports que le conteneur expose, et donne ainsi des informations sur la nature des services correspondants. Grâce à un `docker inspect` sur une image, nous pouvons rapidement nous rendre compte des ports exposés de l'image :

```
$ docker inspect --format='{{json .ContainerConfig.ExposedPorts}}' ssh
{"22/tcp":{}}
```

En résumé

Soit nous utilisons l'option `-P` et, dans ce cas, tous les ports exposés par l'instruction EXPOSE seront mappés automatiquement avec des ports disponibles de la machine hôte.

Soit nous utilisons l'option `-p` autant que nécessaire et, dans ce cas, les ports exposés sont spécifiés manuellement ; l'instruction EXPOSE devient alors uniquement informelle. Le mappage se fait soit automatiquement (on ne spécifie pas le port hôte, par exemple `-p 22`) soit manuellement (on spécifie le port hôte, par exemple `-p 35022:22`).

Pour conclure cette section, reprenons une nouvelle fois l'exemple ssh : l'image a été construite, le conteneur correspondant démarré et le port hôte est connu (dans notre cas 35022), ce dernier étant mappé avec le port 22 du conteneur qui fait référence au processus SSH. On souhaite maintenant se connecter au conteneur par SSH avec l'utilisateur « user » et le mot de passe « pass » :

```
$ ssh user@localhost -p 35022
The authenticity of host '[localhost]:35022 ([::1]:35022)' can't be
established.
RSA key fingerprint is 09:7f:b6:6b:d5:83:3d:db:c5:1a:29:f8:7c:f6:48:1b.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[localhost]:35022' (RSA) to the list of known
hosts.
user@localhost's password:
[user@1d8e61685fe8 ~]$
```

Pour arrêter le conteneur, nous utilisons `docker stop` :

```
docker stop 1d8e61685fe8
```

7.2.7 ADD

L'instruction ADD permet d'ajouter un fichier dans l'image. La source du fichier (d'où il provient) doit être disponible à travers la machine qui construira l'image (par exemple, un fichier local ou distant si une connexion distante est possible). Le fichier sera définitivement ajouté dans l'image, dans le sens où il ne s'agira pas d'un raccourci ou d'un alias.

L'instruction ADD est très proche de l'instruction COPY, néanmoins il existe quelques différences qui sont détaillées dans cette section et la suivante.

Les modèles possibles sont :

ADD <src>... <dest>

ADD ["<src>",... "<dest>"]

<src> désigne le chemin local ou distant (une URL, par exemple) du fichier à ajouter à l'image. <src> peut être répété autant de fois que souhaité et peut contenir des caractères génériques (par exemple « * »).

L'utilisation de caractères génériques dans un chemin source répond aux mêmes règles utilisées pour la fonction Match du langage Go dont la documentation est accessible à :
https://golang.org/pkg/path/filepath/#Match

<dst> désigne le chemin de destination du ou des fichiers à ajouter dans l'image.

Si <src> désigne un fichier local, alors ce dernier doit être disponible sur la machine construisant l'image (celle qui exécute le `docker build`) et non sur la machine hôte

(celle qui exécute `docker run`), simplement car l'ajout se fait à la construction de l'image.

Par exemple :

```
ADD http://code.jquery.com/jquery-2.2.0.min.js /tmp/jquery.js
ADD test* dossierRelatif/
ADD ["/home/vagrant/add/test1", "/home/vagrant/add/test2", "/dossierAbsolu/"]
```

Le premier exemple téléchargera le fichier jquery-2.2.0.min.js à l'URL indiquée, puis le déposera dans le dossier /tmp de l'image en le renommant par jquery.js.

En réalité, nous éviterons d'utiliser une source au format URL. En effet, la fonctionnalité de téléchargement de l'instruction ADD est limitée : elle ne supporte pas les URL protégées par un mécanisme d'authentification. Nous préférerons donc utiliser une instruction RUN (par exemple, RUN wget... ou RUN curl...) à la place.

Le deuxième exemple prendra tout fichier du dossier courant (de la machine qui construit l'image) dont le modèle de nom correspond à « test* » (par exemple « test1 » ou « testlambda ») et les déposera dans le dossier « dossierRelatif/ » dont le chemin est donné relativement (car il n'y a pas de slash en début de chemin) par rapport au chemin courant dans le conteneur.

Pour finir, le troisième exemple prendra les fichiers test1 et test2 (dans la machine qui construit l'image) dont les chemins sont donnés spécifiquement (c'est-à-dire en absolu) ; ils seront déposés dans le dossier « /dossierAbsolu/ » qui lui aussi est donné spécifiquement.

Le dossier courant, ou plutôt le chemin courant de la machine qui construit l'image, correspond simplement au chemin spécifié qui est donné en paramètre de la commande docker build (le fameux « . » en fin de commande).

Lors de la construction d'une image, le chemin courant dans le conteneur est « / », à moins qu'il n'ait été changé par une instruction WORKDIR (voir chapitre 8).

Afin de bien comprendre le principe du chemin courant dans le conteneur, prenons un exemple complet.

Soit le Dockerfile suivant :

```
FROM centos:7
RUN pwd > /tmp/initialPath
RUN mkdir output
RUN cd output
RUN pwd > /tmp/pathAfterOutput
ADD test1 ./
ADD ["test2" , "/output/"]
CMD ls /output
```

Nous considérons que les fichiers test1 et test2 existent et sont dans le même dossier que le Dockerfile.

Les instructions de type RUN pwd > file nous permettent de sauvegarder le chemin courant dans des fichiers que nous déposons dans le dossier /tmp du conteneur.

L'instruction CMD ls /output nous permet, quant à elle, de lister les fichiers et dossiers dans le dossier /output au démarrage du conteneur.

Nous construisons l'image :

```
$ docker build -t add .
```

Puis nous démarrons un conteneur :

```
$ docker run --rm add
```

Le résultat est :

```
test2
```

Nous voyons logiquement le fichier test2 qui a été déposé par l'instruction ADD ["test2" , "/output/"]. Étant donné les instructions RUN cd output puis ADD test1 ./, nous pourrions cependant nous attendre à ce que le fichier test1 soit aussi dans le dossier output, car l'instruction ADD utilise le chemin relatif « ./ » et, selon l'instruction RUN, nous pourrions penser être dans le dossier output... Mais ce n'est pas le cas : rappelons-nous que chaque instruction dans un Dockerfile est exécutée de manière indépendante dans le but de créer des images intermédiaires pour le système de cache. Ainsi, à chaque instruction le contexte courant redevient celui par défaut (dans notre cas « / »), et notre fichier test1 se trouve donc à la racine. En démarrant à nouveau le conteneur, mais en surchargeant le point d'entrée par un ls / :

```
$ docker run --rm add ls /
```

nous obtenons la liste des fichiers et dossiers se trouvant à la racine, notamment test1.

Pour confirmer les explications ci-dessus, regardons les fichiers qui ont sauvegardé l'état du chemin courant :

```
$ docker run --rm add cat /tmp/initialPath
/
$ docker run --rm add cat /tmp/pathAfterOutput
/
```

Dans les deux cas, nous constatons bien que le chemin courant est « / ».

Mais alors, si le dossier courant dans le conteneur est toujours « / », quelle est la différence entre un chemin absolu et un chemin relatif ? En réalité, le dossier courant du conteneur n'est pas toujours « / » ; il peut se paramétrer grâce à l'instruction WORKDIR, et c'est dans ce cas qu'il existe une différence entre un chemin absolu et un chemin relatif.

Maintenant que nous avons bien compris comment fonctionnent les chemins à spécifier dans l'instruction ADD, regardons la différence entre les deux modèles : le premier (ADD <src>... <dest>) est plus facile à lire, mais le deuxième (ADD ["<src>",... "<dest>"]) permet d'ajouter des espaces dans les chemins sans devoir les échapper. Dans la mesure où la bonne pratique d'écriture des noms de fichiers et de dossiers consiste à éviter les espaces, nous préférerons utiliser la première forme.

Quand une facilité de l'instruction ADD devient un inconvénient

L'instruction ADD offre également une facilité supplémentaire : lorsque la source est un fichier local de type archive tar avec un format de compression reconnu (identity, gzip, bzip2 or xz), alors le fichier est automatiquement décompressé en un dossier.

> Lorsque la source est un fichier défini par une URL, il ne sera pas décompressé, et ce même s'il correspond à une archive de format reconnu.

Pour illustrer le principe de décompression automatique, prenons un exemple.

Soit le Dockerfile suivant :

```
FROM centos:7
ADD wordpress-4.4.1-fr_FR.tar.gz /tmp/
CMD ls /tmp
```

Nous considérons que le fichier wordpress-4.4.1-fr_FR.tar.gz existe et réside dans le même dossier que le Dockerfile ; le cas échéant, il peut être téléchargé sur le site de WordPress (https://fr.wordpress.org/wordpress-4.4.1-fr_FR.tar.gz).

Nous construisons l'image et démarrons le conteneur :

```
$ docker build -t untar .
$ docker run --rm untar
ks-script-06FeP3
wordpress
```

Nous remarquons que le fichier wordpress-4.4.1-fr_FR.tar.gz n'est pas présent, mais nous apercevons une entrée « wordpress ». Essayons de voir ce qu'il y a à l'intérieur :

```
$ docker run --rm untar ls /tmp/wordpress
index.php
license.txt
readme.html
wp-activate.php
wp-admin
...
```

Nous constatons aisément qu'il s'agit bien de l'archive qui a été décompressée, car elle contient des fichiers et des dossiers du CMS WordPress.

À première vue, il s'agit d'une fonctionnalité utile, et pourtant elle constitue deux désavantages majeurs :

1. Tout fichier tar compatible sera décompressé. Si nous souhaitons simplement ajouter une archive tar sans la décompresser, cela n'est pas possible (sauf en utilisant un format source de type URL).
2. Il n'est pas toujours aisé de savoir si le format de compression est compatible avec la décompression automatique. Nous pouvons donc penser qu'un fichier sera décompressé, alors que cela ne sera pas le cas.

Ces deux désavantages vont nous inciter à utiliser l'instruction COPY (détaillée dans la prochaine section) au lieu de l'instruction ADD. En effet, COPY ne décompresse pas les archives. Ainsi, nous serons sûrs que le fichier à ajouter ne sera pas modifié, et si nous souhaitons le décompresser, nous le ferons alors spécifiquement, grâce à une instruction RUN.

Le slash à la fin de la destination... ?

Lorsque la destination se termine par un slash (/), cela signifie simplement qu'elle représente un dossier et que la source y sera ajoutée telle quelle, c'est-à-dire que le nom du fichier ne sera pas modifié. Si la destination ne contient pas de slash à la fin, alors le dernier élément du chemin (c'est-à-dire tout ce qui suit le dernier slash) deviendra le nouveau nom de la source, qui sera ainsi renommée.

Imaginons un Dockerfile qui contient entre autres les instructions suivantes :

```
...
ADD test1 /tmp/output1
ADD test2 /tmp/output2/
...
```

Dans le premier ADD, le fichier test1 sera déposé dans le dossier /tmp du conteneur avec pour nouveau nom output1.

Dans le second ADD, le fichier test2 sera déposé dans le dossier /tmp/output2 et gardera son nom.

7.2.8 COPY

L'instruction COPY permet d'ajouter un fichier dans l'image. La source du fichier (d'où il provient) doit être un fichier local à la machine qui construira l'image. Le fichier sera définitivement ajouté dans l'image, dans le sens où il ne s'agira pas d'un raccourci ou d'un alias.

L'instruction COPY est très proche de l'instruction ADD, car elle compose avec les mêmes propriétés suivantes (voir la section précédente pour plus de détails) :

- le principe des chemins courants dans la machine construisant l'image et dans le conteneur ;
- l'utilisation de la fonction Match du langage Go pour l'interprétation des caractères génériques ;
- l'interprétation du slash optionnel à la fin de la destination.

Toutefois, il existe des différences dont les principales sont les suivantes :

- la source doit être un fichier local, et il n'est pas possible de spécifier une URL ;
- l'instruction COPY ne décompresse pas automatiquement une archive tar compatible.

Les modèles possibles sont :

COPY <src>... <dest>

COPY ["<src>",... "<dest>"]

<src> désigne le chemin local du fichier à ajouter à l'image. <src> peut être répété autant de fois que souhaité et peut contenir des caractères génériques (par exemple « * »).

<dst> désigne le chemin de destination du ou des fichiers à ajouter dans l'image.

Par exemple :

```
COPY test* dossierRelatif/
COPY ["/home/vagrant/add/test1", "/home/vagrant/add/test2", "/dossierAbsolu/"]
```

Le principe de fonctionnement de ces deux exemples est le même que celui expliqué pour l'instruction ADD. En résumé, le premier exemple prendra tout fichier du dossier courant dont le modèle de nom correspond à « test* » et les déposera dans le dossier « dossierRelatif/ » du conteneur. Le deuxième exemple prendra les fichiers test1 et test2 dans /home/vagrant/add/ et les déposera dans le dossier « /dossierAbsolu/ » du conteneur.

Comme cela est mentionné dans la précédente section, nous utiliserons systématiquement l'instruction COPY (et non ADD), notamment pour éviter les problèmes de décompression automatique.

L'instruction COPY, tout comme l'instruction ADD, permet d'ajouter plusieurs fichiers dans l'image, d'un seul coup. Pourtant, cette pratique n'est pas recommandée, afin de pouvoir exploiter au mieux le cache Docker.

Avant d'illustrer cela par un exemple, regardons tout d'abord comment fonctionne le cache avec les instructions ADD et COPY ; le principe est simple : si au moins un fichier source est modifié (c'est-à-dire que son contenu est modifié), alors l'instruction sera considérée comme changée et sera donc rejouée (pas de cache utilisé), ainsi que toute instruction suivante (principe général du cache Docker).

Voyons maintenant un exemple. Soit les deux Dockerfile suivants :

```
#Dockerfile multicopy1
FROM centos:7
COPY test1.tar.gz /tmp/
RUN tar xzf /tmp/test1.tar.gz
COPY test2.tar.gz /tmp/
RUN tar xzf /tmp/test2.tar.gz

#Dockerfile multicopy2
FROM centos:7
COPY test1.tar.gz test2.tar.gz /tmp/
RUN tar xzf /tmp/test1.tar.gz
RUN tar xzf /tmp/test2.tar.gz
```

Nous construisons les images :

```
$ docker build -t multicopy1 .
$ docker build -t multicopy2 .
```

Dans les deux cas, la construction n'utilise pas le cache Docker car il s'agit de la première tentative.

Modifions les fichiers test1.tar.gz (pour chaque Dockerfile) et reconstruisons les images :

```
$ docker build -t multicopy1 .
Sending build context to Docker daemon 7.545 MB
Step 1 : FROM centos:7
 ---> 60e65a8e4030
Step 2 : COPY test1.tar.gz /tmp/
 ---> d0e43db03e6c
Removing intermediate container 4d3868594a50
Step 3 : RUN tar xzf /tmp/test1.tar.gz
 ---> Running in 9202535ddaed
 ---> 39a9decd83fb
Removing intermediate container 9202535ddaed
Step 4 : COPY test2.tar.gz /tmp/
 ---> e9536947ac28
Removing intermediate container eace3434fea3
Step 5 : RUN tar xzf /tmp/test2.tar.gz
 ---> Running in 8f58bc549131
 ---> 6d0ee121a384
Removing intermediate container 8f58bc549131
Successfully built 6d0ee121a384
...
$ docker build -t multicopy2 .
Sending build context to Docker daemon 7.545 MB
Step 1 : FROM centos:7
 ---> 60e65a8e4030
Step 2 : COPY test1.tar.gz test2.tar.gz /tmp/
 ---> 9a1f8af948f0
Removing intermediate container 95ce5740585c
Step 3 : RUN tar xzf /tmp/test1.tar.gz
 ---> Running in 32e19bb0e2e0
 ---> 5dcb5162ec7c
```

```
Removing intermediate container 32e19bb0e2e0
Step 4 : RUN tar xzf /tmp/test2.tar.gz
 ---> Running in a06fca73ed6c
 ---> d33c13e5bbc0
Removing intermediate container a06fca73ed6c
Successfully built d33c13e5bbc0
```

Là encore, le cache n'est pas utilisé : dans le premier Dockerfile, la copie de test1.tar.gz étant placé en début de fichier, toutes les instructions suivantes seront rejouées ; dans le deuxième Dockerfile, la constatation est la même (rappelons-nous que dès qu'au moins un fichier d'une instruction COPY est modifié, alors cette dernière est considérée comme changée).

Modifions les fichiers test2 (pour chaque Dockerfile) et reconstruisons encore une fois les images :

```
$ docker build -t multicopy1 .
Sending build context to Docker daemon 4.096 kB
Step 1 : FROM centos:7
 ---> 60e65a8e4030
Step 2 : COPY test1.tar.gz /tmp/
 ---> Using cache
 ---> d0e43db03e6c
Step 3 : RUN tar xzf /tmp/test1.tar.gz
 ---> Using cache
 ---> 39a9decd83fb
Step 4 : COPY test2.tar.gz /tmp/
 ---> e27d6421f696
Removing intermediate container 64173406dc32
Step 5 : RUN tar xzf /tmp/test2.tar.gz
 ---> Running in 10f9ded2eb05
 ---> f8c775d61595
Removing intermediate container 10f9ded2eb05
Successfully built f8c775d61595
...
$ docker build -t multicopy2 .
Sending build context to Docker daemon 4.096 kB
Step 1 : FROM centos:7
 ---> 60e65a8e4030
Step 2 : COPY test1.tar.gz test2.tar.gz /tmp/
 ---> c3f989c037a2
Removing intermediate container d7facd75577e
Step 3 : RUN tar xzf /tmp/test1.tar.gz
 ---> Running in a93769d926ff
 ---> d67aa62ef046
Removing intermediate container a93769d926ff
Step 4 : RUN tar xzf /tmp/test2.tar.gz
 ---> Running in ee02c75ad812
 ---> b8ec10c61515
Removing intermediate container ee02c75ad812
Successfully built b8ec10c61515
```

Nous constatons cette fois-ci que la copie et la décompression de test1.tar.gz dans le premier Dockerfile utilisent le cache (instruction Using cache), ce qui n'est toujours

pas le cas dans le deuxième Dockerfile ; cela démontre bien l'utilité de séparer l'ajout de plusieurs fichiers en autant d'instructions.

7.2.9 VOLUME

L'instruction VOLUME permet de créer un point de montage dans l'image. Ce dernier se référera à un emplacement, appelé volume, dans l'hôte ou dans un autre conteneur.

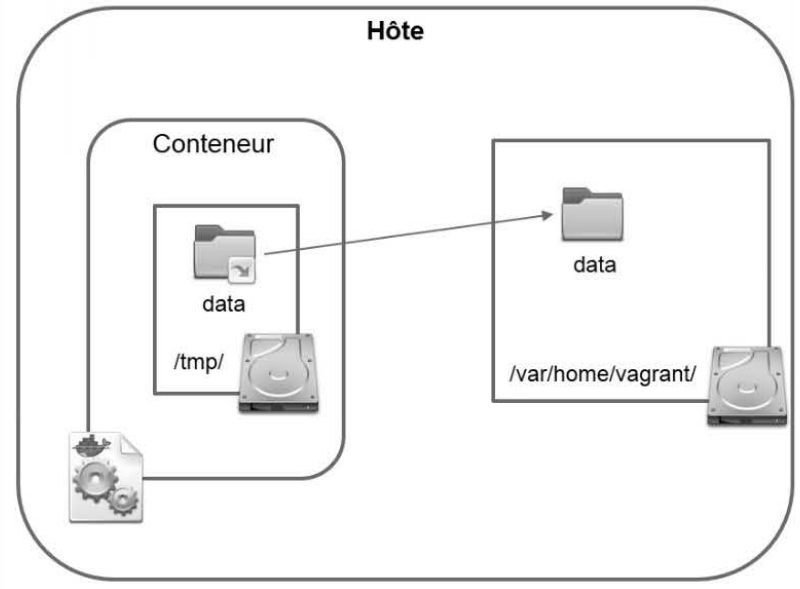

Figure 7.9 — Point de montage dans un conteneur

La figure 7.9 illustre un point de montage /tmp/data dans un conteneur qui référence l'emplacement /var/home/vagrant/data ; autrement dit, le dossier data du conteneur peut être vu comme un lien vers le dossier data de l'hôte. À noter que dans cet exemple les deux dossiers (dans le conteneur et dans l'hôte) sont nommés de manière identique (data), cependant leurs noms pourraient être différents.

> Le montage et l'exploitation de volumes dans un conteneur sont un élément clé et complexe de Docker. Cette section se focalisera sur l'instruction VOLUME, et sur son utilisation correcte en pratique. Des exemples complets ont été donnés dans les autres chapitres de cet ouvrage.

Les modèles possibles sont :

```
VOLUME <path>...
```

```
VOLUME ["<path>",...]
```

Par exemple :

```
VOLUME /tmp/data
VOLUME ["/tmp/data1", "/tmp/data2"]
```

Nous préférerons utiliser le modèle VOLUME <path>..., ce dernier étant plus lisible ; et pour limiter le nombre d'images intermédiaires, nous n'utiliserons qu'une seule instruction VOLUME, avec au besoin plusieurs chemins, par exemple :

```
VOLUME /chemin1/dossier1 \
/chemin2/dossier2 \
```

Prenons maintenant un exemple simple pour bien comprendre le fonctionnement de de l'instruction VOLUME :

```
FROM centos:7
VOLUME /tmp/data
CMD ping localhost
```

L'instruction CMD ping localhost permet de laisser le conteneur actif une fois qu'il a démarré.

Nous construisons l'image, puis démarrons le conteneur :

```
$ docker build -t volume .
$ docker run -d --name volume-conteneur volume
```

Nous n'avons volontairement pas mis l'option `--rm` afin de conserver le volume monté lorsqu'on arrête le conteneur. Pour rappel, l'option `-d` permet d'exécuter le conteneur en arrière-plan (ainsi l'invite de commande reste disponible).

Grâce à un `docker inspect`, nous pouvons voir les volumes montés dans un conteneur :

```
$ docker inspect --format='{{json .Mounts}}' volume-conteneur
[{"Name":"4b1a5d7749dded44d569f15ce55c1f7250bbadebb4b7747190e5656ecb711c55",
"Source":"/var/lib/docker/volumes/4b1a5... 11c55/_data",
"Destination":"/tmp/data","Driver":"local","Mode":"","RW":true}]
```

Dans notre exemple, on voit que le volume du conteneur /tmp/data est basé sur le dossier /var/lib/docker/volumes/4b1a5...11c55/_data dans l'hôte. Listons ce volume depuis le conteneur :

```
$ docker exec volume-conteneur ls /tmp/data
```

Le dossier est vide. Faisons la même chose depuis l'hôte (à faire avec un utilisateur root) :

```
$ sudo ls /var/lib/docker/volumes/4b1a5...11c55/_data
```

Là encore, le dossier est vide. Ajoutons maintenant un contenu :

```
$ docker exec volume-conteneur /bin/sh -c 'echo "Hello" > /tmp/data/helloTest'
```

Afin de s'assurer que la commande `echo "Hello" > /tmp/data/helloTest` soit bien exécutée entièrement dans le conteneur, on l'encapsule par terminal applicatif avec /bin/sh -c. En omettant ce procédé, la partie `> /tmp/data/helloTest` serait exécutée dans l'hôte, ce qui n'est évidemment pas souhaité.

Listons une nouvelle fois le volume :

```
$ docker exec volume-conteneur ls /tmp/data
helloTest
$ sudo ls /var/lib/docker/volumes/4b1a5...11c55/_data
helloTest
```

Nous y voyons bien le fichier helloTest fraîchement créé, quelle que soit la méthode employée pour lister le volume. Arrêtons maintenant le conteneur :

```
$ docker stop volume-conteneur
```

Si nous listons à nouveau le volume (depuis l'hôte uniquement car le conteneur est arrêté) :

```
$ sudo ls /var/lib/docker/volumes/4b1a5...11c55/_data
helloTest
```

Nous y voyons toujours le fichier helloTest : le comportement attendu est bien respecté. Toutefois, il serait utile que le chemin dans l'hôte soit plus explicite ; pour cela, nous utilisons l'option `-v` de `docker run`.

Tout d'abord, supprimons le conteneur afin de libérer l'espace utilisé par ce dernier et pour pouvoir réutiliser son nom :

```
$ docker rm volume-conteneur
```

À noter que le volume dans l'hôte est aussi supprimé. Ensuite, démarrons le conteneur avec l'option `-v` :

```
$ docker run -d -v /var/home/vagrant/data:/tmp/data --name volume-conteneur
volume
```

/var/home/vagrant/data représente le chemin dans l'hôte et /tmp/data le chemin dans le conteneur. Créons à nouveau un fichier dans le volume (rappelons-nous que la suppression du conteneur a également supprimé le volume) :

```
$ docker exec volume-conteneur /bin/sh -c 'echo "Hello" > /tmp/data/helloTest'
```

Maintenant, listons le contenu du volume depuis l'hôte :

```
$ sudo ls /var/home/vagrant/data
helloTest
```

Revenons sur la valeur de l'option **-v**, notamment la partie qui décrit le point de montage dans le conteneur, c'est-à-dire /tmp/data : nous remarquons que cette information fait double emploi avec l'instruction VOLUME du Dockerfile. En réalité, si nous spécifions l'option -v sur un même volume défini par l'instruction VOLUME, alors l'effet fonctionnel de cette dernière devient nul. Reprenons l'exemple :

- D'abord, nous arrêtons et supprimons le conteneur :

```
$ docker stop volume-conteneur
$ docker rm volume-conteneur
```

- Nous modifions le Dockerfile en commentant l'instruction VOLUME :

```
FROM centos:7
# VOLUME /tmp/data
CMD ping localhost
```

- Nous construisons et démarrons l'image, puis nous ajoutons un fichier au volume :

```
$ docker build -t volume .
$ docker run -d -v /var/home/vagrant/data:/tmp/data --name volume-conteneur
volume
$ docker exec volume-conteneur /bin/sh -c 'echo "Hello" > /tmp/data/helloTest'
```

- Nous effectuons un `docker inspect` afin de constater l'état du volume et son point de montage :

```
$ docker inspect --format='{{json .Mounts}}' volume-conteneur
[{"Source":"/var/home/vagrant/data","Destination":"/tmp/data","Mode":"","RW":true}]
```

- Pour finir, nous listons le volume :

```
$ ls /var/home/vagrant/data/
helloTest
```

Ainsi, le résultat avec ou sans l'instruction VOLUME dans le Dockerfile reste le même dès le moment où l'on utilise l'option `-v` lors du `docker run`. Cependant, il existe un cas où l'instruction VOLUME reste malgré tout utile : elle apporte une métadonnée qui décrit le point de montage dans le conteneur du volume dans l'hôte. Nous nous efforcerons donc d'inclure l'instruction VOLUME au Dockerfile, et cela même si son effet est surchargé lors du démarrage du conteneur.

Pour terminer cette section, arrêtons-nous sur le format du chemin : faut-il le spécifier en relatif ou en absolu ? Par sa conception, l'instruction VOLUME est toujours exécutée à partir du chemin « / », et ce même si un précédent WORKDIR avait changé le chemin courant ; en conséquence, la présence d'un « / » initial dans le chemin de l'instruction VOLUME ne change rien. Cependant, pour qu'il ne puisse y avoir d'ambiguïté à la lecture du chemin, nous le spécifierons toujours.

Les volumes nommés

Depuis la version 1.9 de Docker, il est possible de créer à l'aide de la commande `docker volume create` des volumes nommés. En pratique, cette nouveauté permet de ne plus recourir aux *data containers* c'est-à-dire aux conteneurs dont la seule vocation est de référencer des volumes. Il est ainsi possible de créer un volume avant même d'avoir créé un conteneur, puis de l'associer à un ou plusieurs conteneurs. Nous verrons un exemple de ces volumes nommés dans le chapitre suivant (dans la section consacrée à l'installation d'un registry privé).

7.3 BONNES PRATIQUES

Pour conclure cette section, nous allons récapituler les bonnes pratiques de l'écriture d'un fichier Dockerfile.

Généralités

Pour les instructions CMD et ENTRYPOINT, nous préférerons utiliser le format exécution (par exemple CMD ["ping", "localhost"]) afin d'assurer un arrêt propre d'un conteneur.

Pour l'instruction RUN, nous préférerons utiliser le format terminal pour des raisons de lisibilité.

Afin d'optimiser la gestion du cache Docker, nous trierons les instructions en fonction de la possibilité qu'elles soient modifiées dans le temps.

L'ordre ci-dessous apporte un bon niveau fonctionnel et une bonne lisibilité. Nous tâcherons de l'appliquer autant que possible :

1. FROM
2. MAINTAINER
3. ARG
4. ENV, LABEL
5. VOLUME
6. RUN, COPY, WORKDIR
7. EXPOSE
8. USER
9. ONBUILD
10. CMD, ENTRYPOINT

Dans toute instruction, nous n'utiliserons pas la commande `sudo`.

FROM

Un fichier Dockerfile contiendra une et une seule instruction FROM, et nous spécifierons aussi formellement la version (le *tag*) de l'image source à utiliser, par exemple FROM centos:7.

RUN

Lors de l'installation de paquets avec l'instruction RUN, nous mettrons systématiquement à jour l'image courante et nous n'utiliserons qu'une seule instruction, c'est-à-dire :

```
RUN yum update -y && yum install -y \
httpd \
mariadb-server \
php php-mysql
```

Nous éviterons de changer le chemin courant avec l'instruction RUN et nous utiliserons plutôt l'instruction WORKDIR (voir le chapitre 8).

CMD et ENTRYPOINT

Nous utiliserons en priorité l'instruction ENTRYPOINT (au format exécution, comme mentionné ci-dessus). L'instruction CMD sera utilisée si le point d'entrée doit être surchargé partiellement (dans ce cas, l'instruction sera combinée avec ENTRYPOINT) ou complètement (dans ce cas l'instruction ENTRYPOINT ne sera pas utilisée).

Un fichier Dockerfile ne contiendra au maximum qu'une seule instruction CMD et ENTRYPOINT.

EXPOSE

Bien que l'instruction EXPOSE ne soit utile que si nous spécifions l'option **-P** au démarrage du conteneur, nous nous efforcerons de l'employer afin de renseigner clairement sur les ports (la nature des services) qu'utilise le conteneur.

ADD et COPY

Nous utiliserons systématiquement l'instruction COPY au lieu de ADD, ceci afin d'éviter des comportements imprévisibles liés à la décompression automatique des archives tar de l'instruction ADD.

Nous préférerons utiliser le modèle COPY <src>... <dest> (au lieu de COPY ["<src>",... "<dest>"]) car il est plus agréable à lire et à maintenir. Dans des cas exceptionnels, où la bonne pratique de l'écriture des noms de fichiers et de dossiers n'est pas respectée, nous utiliserons l'autre forme.

Nous utiliserons une instruction COPY par fichier à ajouter, ceci afin d'optimiser l'utilisation du cache Docker.

VOLUME

Même si l'instruction VOLUME devait être surchargée lors du `docker run`, nous nous efforcerons de la mettre dans le Dockerfile afin de renseigner sur les points de montage du conteneur.

Le chemin commencera toujours par un « / » afin d'éviter toute confusion, notamment si le Dockerfile contient des instructions WORKDIR.

Nous utiliserons une seule instruction VOLUME avec le modèle VOLUME <path>... et au besoin plusieurs chemins :

```
VOLUME /chemin1/dossier1 \
/chemin2/dossier2 \
```

ENV, LABEL et ARG

Ces commandes seront expliquées en détail dans le chapitre 8.

Afin de limiter le nombre de couches, on regroupera les variables d'environnement (ENV) définie dans un Dockerfile en une instruction. On procédera de la même manière pour les labels (LABEL).

Si cela est nécessaire, on préférera surcharger la variable d'environnement PATH avec les chemins d'éventuels nouveaux exécutables installés dans l'image, au lieu de devoir spécifier le chemin absolu de l'exécutable à chaque utilisation.

Les instructions ARG seront toujours placées avant les instructions ENV et LABEL.

Lorsqu'une variable d'environnement peut être surchargée lors de la construction d'une image, alors on appliquera une utilisation conjointe des instructions ENV et ARG, c'est-à-dire :

```
ARG variable
ENV variable ${variable:-valeurParDéfaut}
```

En résumé

Nous avons décrit dans ce chapitre les principales instructions de Dockerfile. À travers différents exemples, nous avons abordé les principaux cas d'usage de ces commandes.

Il est temps à présent d'aborder les commandes moins fréquentes et d'étudier quelques fonctionnalités plus avancées de Docker.

8

Usage avancé de Docker

Nous avons vu dans les précédents chapitres comment installer Docker sur différents environnements, comment créer des conteneurs, comment utiliser le client Docker et comment créer des images à partir de Dockerfile.

L'objectif de ce chapitre est d'aborder des usages plus avancés de Docker.

Nous y décrirons quelques instructions Dockerfile un peu moins courantes que celles qui ont été présentées dans les chapitres précédents. Nous découvrirons ensuite Notary, le système de signature d'images proposé par Docker. Enfin, nous verrons comment installer un Docker registry privé.

8.1 DOCKERFILE : QUELQUES INSTRUCTIONS SUPPLÉMENTAIRES

Dans le chapitre 7, nous nous sommes attachés à décrire les commandes Dockerfile les plus courantes. Dans cette nouvelle section, nous allons décrire celles que nous n'avons pas encore vues.

8.1.1 ENV

L'instruction ENV permet de créer ou de mettre à jour une ou plusieurs variables d'environnement, afin de les utiliser lors de la construction de l'image et dans les conteneurs associés.

> Les variables d'environnement créées (ou modifiées) sont disponibles dans toute la descendance de l'image. Autrement dit, un conteneur Docker a accès aux variables

d'environnement décrites par son Dockerfile et celles décrites par le Dockerfile de l'image source.
Si une variable d'environnement existe déjà, alors l'instruction ENV remplacera simplement la valeur existante.

Les modèles sont :

ENV <name> <value>

ENV <name>=<value> ...

<name> représente le nom de la variable d'environnement.

<value> représente la (nouvelle) valeur souhaitée pour la variable d'environnement.

Le premier modèle permet de renseigner une variable et sa valeur, tandis que le second permet de renseigner autant de variables (avec valeurs) que souhaité.

Par exemple :

```
ENV myName James Bond
ENV myName="James Bond" myJob=Agent\ secret
```

Dans le premier exemple, nous constatons qu'il n'y a pas de signe « = », et que la valeur n'est pas encapsulée par des guillemets ni échappée (par exemple, un antislash avant un espace). Dans le second exemple, les valeurs doivent être encapsulées par des guillemets (par exemple, "James Bond") ou échappées (par exemple, Agent\ secret), un signe « = » devant également être présent entre le nom et la valeur.

Si plusieurs variables d'environnement doivent être créées, alors nous préférerons utiliser la deuxième forme car elle ne produira qu'une seule couche de cache d'image, alors que la première forme en produira autant que d'instructions ENV.

Prenons un exemple pour illustrer le fonctionnement de l'instruction ENV. Soit le Dockerfile suivant :

```
FROM centos:7
ENV myName="James Bond" myJob=Agent\ secret
CMD echo $myName
```

Construisons et démarrons un conteneur :

```
$ docker build -t env .
$ docker run --rm env
James Bond
```

Il est possible de surcharger une variable d'environnement au démarrage d'un conteneur, nous utilisons pour cela l'option `--env <name>=<value>`, par exemple :

```
$ docker run --rm --env myName="Jason Bourne" env
```

L'option `--env` est cumulative, par exemple :

```
$ docker run --rm --env myName="Jason Bourne" --env myJob="CIA" env
```

Pour finir, si nous souhaitons afficher la variable myJob au lieu de myName au démarrage du conteneur, il suffit de surcharger l'instruction CMD :

```
$ docker run --rm  env /bin/sh -c 'echo "$myJob"'
```

Cette dernière commande est un peu particulière : en effet, nous n'utilisons pas `echo $myJob`, mais `/bin/sh -c 'echo "$myJob"'`. La variable `$myJob` de la commande `echo $myJob` sera substituée au niveau de l'hôte, ce qui correspondrait à `echo ""` (en considérant que la variable n'existe pas dans l'hôte) ; à l'exécution, le résultat serait donc vide au lieu d'afficher la valeur de la variable `$myJob` du conteneur. La version `/bin/sh -c 'echo "$myJob"'`, quant à elle, permet d'effectuer la substitution au niveau du conteneur (car `echo` est encapsulé par un terminal applicatif), de telle sorte que nous obtenons le résultat souhaité.

Un autre cas typique d'utilisation de l'instruction ENV est la modification de la variable d'environnement PATH. En effet, si un Dockerfile contient, par exemple, l'installation d'une application (`RUN yum install...`), il devient intéressant de pouvoir y accéder facilement (c'est-à-dire en saisissant uniquement son nom et pas son chemin absolu). Par exemple :

```
...
ENV PATH /usr/local/nginx/bin:$PATH
...
```

L'application nginx (supposée installée par une autre instruction) est ajoutée à la variable d'environnement PATH. Nous pouvons donc l'utiliser directement dans une autre instruction :

```
...
ENTRYPOINT ["nginx"]
...
```

Pour terminer cette section, voyons comment il est possible de lister les variables d'environnement d'un conteneur actif. Nous utilisons pour cela un `docker inspect` :

```
$ docker inspect --format='{{json .Config.Env}}' test
```

Pour exécuter un `docker inspect` sur un conteneur, ce dernier doit être actif. Dans les exemples ci-dessus, le conteneur n'est actif qu'un très court instant car il se termine directement après l'instruction CMD qui est un simple `echo`. Pour prolonger l'activité du conteneur, on pourrait surcharger la commande `echo` par un `ping`, ainsi le `docker run` serait `docker run -rm -name test env ping localhost`.

Le résultat retourné est :

```
["PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin","myName=James
Bond","myJob=Agent secret"]
```

La variable d'environnement PATH n'est pas décrite directement par le Dockerfile, mais appartient à l'image source. On confirme ainsi l'accès aux variables parentes.

8.1.2 LABEL

L'instruction LABEL permet d'ajouter des métadonnées à une image.

Le modèle est :

LABEL <name>=<value> ...

`<name>` représente le nom du label.

`<value>` représente la (nouvelle) valeur souhaitée pour le label.

Par exemple :

```
LABEL name="Mon application" version="1.0"
LABEL app.name="Mon application" "app.version"="1.0"
```

Les noms peuvent contenir lettres, chiffres, points (.), tirets (-) et soulignés (_). Ils peuvent être encapsulés par les guillemets (") ou non.

Les valeurs des labels doivent être encapsulées par des guillemets ("). Si elles contiennent un guillemet ou un retour à la ligne alors ces derniers doivent être préfixés par un antislash, par exemple :

```
LABEL app.description="Voici la description de \"Mon application\" \
sur plusieurs lignes."
```

Tout comme pour l'instruction ENV, les labels créés (ou modifiés) sont disponibles dans toute la descendance de l'image. Autrement dit, un conteneur Docker a accès aux labels décrits par son Dockerfile et ceux décrits par le Dockerfile de l'image source (et ce de façon récursive).
Si le label existe déjà, alors l'instruction LABEL remplacera simplement la valeur existante.

Soit le Dockerfile suivant :

```
FROM centos:7
LABEL app.name="Mon application" \
"app.version"="1.0" \
app.description="Voici la description de \"Mon application\" \
sur plusieurs lignes."
CMD echo "fin"
```

Construisons l'image et démarrons un conteneur :

```
$ docker build -t label .
$ docker run -rm -name test label
fin
```

Tout s'est bien déroulé, toutefois les labels étant des métadonnées, on ne peut rien constater à travers l'exécution d'un conteneur. On peut cependant utiliser la commande `docker inspect` sur l'image afin de lister les labels existants :

```
$ docker inspect --format='{{json .Config.Labels}}' test
```

json .Config.Labels signifie que nous affichons uniquement les labels, et ce au format JSON. « test » représente le nom du conteneur actif.

Le résultat retourné est :

```
{"app.description":"Voici la description de \"Mon application\" sur plusieurs
lignes.","app.name":"Mon application","app.version":"1.0","build-date":"2015-
12-23","license":"GPLv2","name":"CentOS Base Image","vendor":"CentOS"}
```

Les labels build-date, license, name et vendor sont fournis par l'image source.

8.1.3 WORKDIR

L'instruction WORKDIR permet de changer le chemin courant (appelé dossier de travail) pour les instructions RUN, CMD, ENTRYPOINT, COPY et ADD. Elle peut être utilisée plusieurs fois dans un fichier Dockerfile. Son effet s'applique à toute instruction qui suit.

Son modèle est :

WORKDIR <chemin>

Par exemple :

```
WORKDIR /tmp
```

Dans cet exemple, le chemin courant sera changé en /tmp, cela signifiant que pour toute instruction RUN, CMD, ENTRYPOINT, COPY et ADD impliquant un chemin relatif dans l'image, ce dernier sera pris à partir de /tmp.

> Il est important de comprendre que l'instruction WORKDIR change le chemin courant dans l'image. Ainsi, lorsqu'un conteneur est démarré pour une image, le chemin courant du conteneur sera également le chemin spécifié par le dernier WORKDIR.

En spécifiant un chemin relatif, alors ce dernier sera appliqué au contexte courant :

```
FROM centos:7
WORKDIR /tmp
WORKDIR test
CMD pwd
```

Cet exemple produira /tmp/test comme résultat du `pwd` (au démarrage du conteneur). De ce fait, on constate que pour écrire WORKDIR test nous devons connaître le contexte courant (dans notre cas /tmp). Pour cet exemple, il n'y a pas d'ambiguïté (le chemin courant est /tmp), par contre pour un Dockerfile conséquent, ou si l'image source a modifié le WORKDIR, le contexte n'est plus si évident. Nous appliquerons donc la règle suivante :

> Le chemin de l'instruction WORKDIR doit toujours être spécifié en absolu.

L'effet d'un WORKDIR est appliqué en cascade aux images enfants. Voyons cela par un exemple. Soit le Dockerfile suivant :

```
#Dockerfile workdirsource
FROM centos:7
WORKDIR /var
RUN pwd > /tmp/cheminCourant
CMD cat /tmp/cheminCourant
```

Dans cet exemple, nous changeons le chemin courant en /var, puis on sauvegarde immédiatement la valeur retournée par la commande `pwd` dans un fichier cheminCourant déposé dans /tmp. Si nous construisons et démarrons le conteneur, nous affichons le contenu de ce fichier, c'est-à-dire :

```
/var
```

Si maintenant nous surchargeons le point d'entrée afin d'exécuter directement la commande `pwd` dans le conteneur démarré, soit `docker run --rm workdirsource pwd`, nous obtenons exactement le même résultat :

```
/var
```

Ce qui démontre bien que l'effet de l'instruction WORKDIR s'applique à l'image et donc aux conteneurs qui en découlent.

Continuons l'exemple avec un autre Dockerfile :

```
FROM workdirsource
CMD pwd
```

Ce Dockerfile est très simple : nous utilisons l'image que nous venons de créer comme source et nous affichons le chemin courant au démarrage du conteneur. Là encore, le résultat est le même :

```
/var
```

Ce qui démontre que l'effet d'un WORKDIR s'applique bien aux enfants d'une image.

De plus, l'effet de WORKDIR /unChemin est équivalent à `RUN cd /unChemin && faireQuelqueChose`, mais l'utilisation de l'instruction WORKDIR offre deux avantages non négligeables :

- tout d'abord, un gain de lisibilité ; en effet, cela permet de voir directement la véritable fonction d'un RUN ;
- ensuite, un gain en maintenabilité ; il suffit de changer une et une seule instruction WORKDIR si nous décidons de modifier un chemin (un nom de dossier, par exemple) impliqué dans plusieurs instructions RUN.

Il y a un dernier point au sujet de l'instruction WORKDIR : elle permet l'utilisation de variables d'environnement créées par l'instruction ENV au sein du même Dockerfile. Cela signifie qu'une variable d'environnement provenant d'une image source ou native à un système ne pourra pas être utilisée. Par exemple :

```
FROM cento:7
ENV cheminCourant /tmp
WORKDIR $cheminCourant/$unFichier
CMD pwd
```

Cet exemple produira /tmp/$unFichier comme résultat du `pwd` (au démarrage du conteneur). La variable $unFichier n'existant pas dans le Dockerfile, elle ne sera pas substituée, et ce même si le Dockerfile de l'image source contient une instruction du genre ENV unFichier monFichier.

8.1.4 USER

L'instruction USER permet de définir l'utilisateur qui exécute les commandes issues des instructions RUN, CMD et ENTRYPOINT. Son effet s'applique à toute instruction qui suit et à toute image enfant.

Le modèle est :

USER <user>

<user> définit le nom d'utilisateur ou l'UID à utiliser.

Par exemple :

```
USER httpd
```

Un fichier Dockerfile peut contenir plusieurs fois l'instruction USER, cependant il convient de ne l'utiliser qu'une seule fois, sauf en cas de besoins particuliers, afin de simplifier la compréhension de fonctionnement des permissions dans l'image.

La spécification d'un utilisateur avec l'instruction USER est surtout utile, et même indispensable du point de vue de la sécurité, dans le cycle de vie d'un conteneur. Par contre, lors de la construction de l'image elle-même, c'est-à-dire avec les instructions RUN, nous garderons l'utilisateur root. Pour résumer :

- une image non-finale, c'est-à-dire une image qui sera principalement utilisée par d'autres Dockerfile comme source, ne contiendra en général pas d'instruction USER ;
- pour une image finale, les instructions RUN seront effectuées en root (c'est-à-dire avant toute instruction USER), puis on spécifiera un utilisateur afin d'assurer que le processus actif du conteneur n'est pas exécuté en root.

Voyons le fonctionnement de l'instruction USER dans un exemple :

```
FROM centos:7
RUN chown root:root /bin/top && \
    chmod 774 /bin/top
RUN groupadd -r mygroup && \
    useradd -r -g mygroup myuser
USER myuser
CMD /bin/top -b
```

Avec le premier RUN, nous nous assurons que le binaire /bin/top (affichage des processus en cours) appartient à l'utilisateur et groupe root et que seuls ces derniers peuvent l'exécuter. Le deuxième RUN crée un utilisateur myuser, un group mygroup et assigne l'utilisateur créé au groupe créé. L'option -b du binaire *top* définit que ce dernier est exécuté en mode batch ; cela signifie que son exécution est toujours active avec un rafraîchissement régulier des processus en cours, jusqu'à ce qu'il soit manuellement arrêté (dans notre cas, lors de l'arrêt du conteneur).

En construisant l'image et en démarrant le conteneur, nous obtenons :

```
$ docker build -t user .
Sending build context to Docker daemon 2.048 kB
...
```

```
Successfully built af7b2fa53d95
$ docker run --rm --name user-conteneur user
/bin/sh: /bin/top: Permission denied
```

Étant donné que l'utilisateur courant est myuser (instruction USER), nous constatons logiquement que le binaire `/bin/top` ne peut être exécuté car seul root peut le faire. Modifions maintenant le Dockerfile :

```
FROM centos:7
RUN chown root:root /bin/top && \
 chmod 774 /bin/top
RUN groupadd -r mygroup && \
 useradd -r -g mygroup myuser
RUN chown myuser:mygroup /bin/top
USER myuser
CMD /bin/top -b
```

En reconstruisant l'image et en redémarrant le conteneur, le binaire `/bin/top` s'exécute correctement car l'instruction RUN, qui a été ajoutée, définit l'utilisateur du binaire `/bin/top` par myuser et son groupe par mygroup.

Il est également possible de visualiser l'utilisateur courant du conteneur grâce à un `docker inspect` :

```
$ docker inspect --format='{{json .Config.User}}' user-conteneur
"myuser"
```

Reprenons maintenant l'exemple initial, et essayons une autre approche : nous allons autoriser les utilisateurs du groupe mygroup (soit l'utilisateur myuser) à exécuter le binaire `/bin/top` comme super utilisateur (`sudo`). Nous modifions le Dockerfile ainsi :

```
FROM centos:7
RUN yum update -y && yum install -y sudo
RUN chown root:root /bin/top && \
    chmod 774 /bin/top
RUN groupadd -r mygroup && \
    useradd -r -g mygroup myuser
RUN echo '%mygroup ALL=NOPASSWD: /bin/top' >> /etc/sudoers
USER myuser
CMD sudo /bin/top -b
```

Le premier RUN va installer le binaire `sudo` (permettant la gestion de superutilisateurs). Le dernier RUN va définir le groupe mygroup comme superutilisateur du binaire `/bin/top`, c'est-à-dire que tout membre pourra l'exécuter sans condition. Pour finir, nous ajoutons `sudo` à l'instruction CMD pour dire que la commande sera exécutée via le binaire `sudo`.

Nous construisons l'image :

```
$ docker build -t user .
```

Puis nous démarrons le conteneur :

```
$ docker run --rm --name user-conteneur user
sudo: sorry, you must have a tty to run sudo
```

Alors que la configuration semble correcte, cela ne fonctionne pas : le système nous dit qu'on doit avoir un `tty` pour faire fonctionner `sudo` (car CentOS 7 est configuré ainsi par défaut pour `sudo`). `tty` est une commande permettant d'afficher le nom du terminal associé à l'entrée standard : dans notre cas, il n'y en a pas. Pour résoudre notre problème, nous avons deux solutions : soit modifier la configuration de CentOS 7 afin qu'elle n'ait pas besoin de `tty` pour `sudo`, soit configurer un terminal pour l'entrée standard. La deuxième solution étant complexe, nous utiliserons la première :

```
FROM centos:7
RUN yum update -y && yum install -y \
 sudo
RUN sed -i -e 's/requiretty/!requiretty/g' /etc/sudoers
RUN chown root:root /bin/top && \
 chmod 774 /bin/top
RUN groupadd -r mygroup && \
 useradd -r -g mygroup myuser
RUN echo '%mygroup ALL=NOPASSWD: /bin/top' >> /etc/sudoers
USER myuser
CMD sudo /bin/top -b
```

Nous avons modifié la configuration de `sudo` (dans le fichier /etc/sudoers) grâce à sed en remplaçant `requiretty` par `!requiretty`, signifiant simplement qu'il n'y a pas besoin de terminal associé à l'entrée standard.

En construisant puis en démarrant le conteneur, nous constatons que cette fois-ci cela fonctionne. Toutefois, cette dernière manipulation que nous avons dû effectuer n'est pas intuitive ; on conclura donc cette section par cette bonne pratique :

> Un Dockerfile ne doit pas contenir d'exécution de commandes avec un superutilisateur, autrement dit il ne contiendra pas de sudo.

8.1.5 ARG

L'instruction ARG permet de définir des variables (appelées arguments) qui sont passées comme paramètres lors de la construction de l'image. Pour cela, l'option `--build-arg` de `docker build` devra être utilisée. Les arguments définis par ARG ne peuvent être utilisés que par des instructions de construction (RUN, ADD, COPY, USER) de l'image elle-même, c'est-à-dire qu'elles ne sont pas disponibles dans les images enfants ainsi que dans le conteneur.

Le modèle est :

> **ARG <name>[=<defaultValue>]**

<name> représente le nom de l'argument.

<defaultValue> représente la valeur par défaut. Ce paramètre est optionnel, s'il n'est pas spécifié et que lors de la construction de l'image l'argument correspondant n'est pas donné en paramètre, alors sa valeur sera simplement vide.

Par exemple :

```
ARG var1
ARG var2="ma valeur"
ARG var3=4
```

Si la valeur contient des espaces, alors elle doit être encapsulée par des guillemets.

L'utilisation d'un argument défini par l'instruction ARG dans un Dockerfile se fait de manière similaire à l'utilisation d'une variable d'environnement, c'est-à-dire :

```
${<name>[:-<defaultValue>]}
```

Les accolades « { » et « } » sont facultatives si defaultValue n'est pas donné.

Par exemple :

```
RUN echo $var1
RUN echo ${var2:-"une autre valeur"]}
```

Dans le deuxième exemple, nous assignons une autre valeur à var2 si cette dernière est vide. Si sa création provenait de l'instruction `ARG var2="ma valeur"`, alors `var2` ne serait jamais vide, et donc une autre valeur ne serait jamais assignée.

Il n'est possible d'utiliser un argument que s'il a été défini (par l'instruction ARG) avant. Ainsi, nous veillerons à définir les arguments en début de Dockerfile.

Comme cela est mentionné plus haut, l'assignation des arguments définie par ARG se fait lors de la construction de l'image, grâce à l'option `--build arg` qui fonctionne ainsi :

```
--build-arg <name>=<value>
```

Il est fortement déconseillé d'utiliser l'option `--build-arg` pour passer des informations sensibles, comme des clés secrètes ou des mots de passe, car les valeurs seront très probablement sauvées dans le cache des commandes exécutées dans le terminal.

Si nous souhaitons spécifier plusieurs arguments, alors nous utiliserons plusieurs fois l'option, par exemple :

```
$ docker build --build-arg var1=valeur1 --build-arg var2="encore une valeur" .
```

Les arguments donnés dans l'option `--build arg` doivent obligatoirement être définis dans le Dockerfile. Si tel n'était pas le cas, une erreur serait retournée lors de la construction de l'image.

Prenons un exemple, soit le Dockerfile suivant :

```
FROM centos:7
ARG var1
ARG var2="ma valeur"
RUN echo $var1 > /tmp/var
RUN echo $var2 >> /tmp/var
CMD echo $var1
```

Ce Dockerfile va simplement mettre le contenu des arguments var1 et var2 dans un fichier var dans tmp. Puis nous afficherons le contenu de var1 lors du démarrage du conteneur.

Nous construisons l'image en spécifiant valeur1 comme valeur pour var1 et nous laissons la valeur par défaut de var2 :

```
$ docker build --build-arg var1=valeur1 -t arg .
```

Nous démarrons le conteneur :

```
$ docker run --rm arg
```

Nous constatons que la valeur de var1 est vide dans le conteneur, ce qui est normal car la portée des variables définies par ARG ne s'étend pas au conteneur. Modifions l'instruction CMD par :

```
CMD cat /tmp/var
```

Construisons et démarrons le conteneur une nouvelle fois :

```
$ docker build --build-arg var1=valeur1 -t arg .
...
$ docker run --rm arg
valeur1
ma valeur
```

Cette fois-ci nous voyons bien les valeurs attendues pour var1 et var2 car leur ajout dans /tmp/var a été fait lors de la construction.

Essayons maintenant de surcharger la valeur de var2 lors de la construction de l'image :

```
$ docker build --build-arg var1=valeur1 --build-arg var2="une autre valeur" -t
arg .
...
$ docker run --rm arg
valeur1
une autre valeur
```

var2 a bien pris la nouvelle valeur.

Comme cela est mentionné plus haut et démontré dans les exemples, l'utilisation des arguments définis par ARG se fait de façon similaire à l'utilisation des variables d'environnement ; mais alors qu'en est-il d'une utilisation conjointe d'un argument et d'une variable d'environnement avec un même nom ?

Soit le Dockerfile suivant :

```
FROM centos:7
ARG var=argument
ENV var variable
RUN echo $var > /tmp/var
CMD cat /tmp/var
```

Nous construisons l'image et démarrons un conteneur :

```
$ docker build -t arg .
...
$ docker run --rm arg
variable
```

Nous constatons que la valeur de $var est « variable », ce qui peut sembler logique car l'assignement de la variable d'environnement est placé après celui de l'argument. Recommençons le même exemple en inversant les assignations :

```
FROM centos:7
ENV var variable
ARG var=argument
RUN echo $var > /tmp/var
CMD cat /tmp/var
```

De nouveau, construisons l'image et démarrons le conteneur :

```
$ docker build -t arg .
...
$ docker run --rm arg
variable
```

Le résultat est le même : $var vaut toujours « variable ». Ce comportement est dû au fait que ENV prime sur ARG, quel que soit l'ordre des instructions ; plus précisément, le résultat sera le suivant :

- si ENV est placé avant ARG : ARG sera ignoré ;

- si ENV est placé après ARG : ARG sera valable jusqu'à ENV, puis, dès l'instruction ENV, ARG sera remplacé.

Nous savons que l'exploitation d'un argument et d'une variable d'environnement dans un Dockerfile se fait de façon équivalente, et que ENV est prioritaire sur ARG. Voyons maintenant dans quel cas on utilise l'une ou l'autre instruction, voire les deux.

Le critère de choix de l'instruction à utiliser (ENV ou ARG) est défini par la portée souhaitée :

- si l'on souhaite agir sur l'image et les conteneurs associés, alors il s'agit d'une variable d'environnement (ENV) ;
- et si l'on souhaite agir uniquement sur les instructions de construction de l'image, alors il s'agit d'un argument (ARG).

Il faut voir une variable d'environnement comme une configuration nécessaire durant tout le cycle vie d'une application, c'est-à-dire de sa construction à son exécution, typiquement, un chemin ou un port. Par contre, un argument est un simple paramètre qui, selon sa valeur, fera varier la façon de construire une application, et qui pourra potentiellement avoir un impact sur son fonctionnel, par exemple un nom d'utilisateur.

> Il n'est pas toujours évident d'identifier quels services utilisent telle ou telle variable d'environnement ; ainsi, l'utilisation de l'instruction ENV doit se faire avec précaution : il convient de toujours vérifier que le nom d'une variable d'environnement que nous souhaitons ajouter n'est pas déjà utilisé par un service dont l'application dépend.

Nous savons maintenant quelle instruction utiliser selon la portée souhaitée. Nous avons également vu qu'il est possible de surcharger une variable d'environnement et un argument. Toutefois, la surcharge se fait différemment :

- pour l'instruction ARG, la surcharge se fait logiquement lors de la construction de l'image (`docker build`) ; en effet, cela n'aurait pas de sens de la faire lors du démarrage d'un conteneur dans la mesure où un argument n'y est pas disponible ;
- pour l'instruction ENV, la surcharge se fait lors du démarrage du conteneur (`docker run`), ce qui, dans certains cas, peut poser un problème de consistance : imaginons une variable d'environnement utile lors de la construction d'une image, puis lors de l'exécution des conteneurs associés ; si l'on surcharge la variable seulement lors du démarrage, alors nous aurons une valeur différente à la construction et à l'exécution.

La solution au problème de consistance de l'instruction ENV est représentée par le cumul des instructions ARG et ENV. Voyons un exemple :

```
FROM centos:7
ARG version
ENV version ${version:-1.0}
RUN echo $version > /tmp/version
CMD cat /tmp/version
```

La valeur de la variable d'environnement est « ${version:-1.0} », ce qui représente la valeur de l'argument « version » s'il n'est pas vide, et « 1.0 » dans le cas contraire (rappelons-nous qu'un argument placé avant une variable d'environnement est valable jusqu'à cette dernière).

Si l'argument « version » n'est pas surchargé lors de la construction de l'image, alors ce dernier est vide et la variable d'environnement correspondante vaudra « 1.0 ». Et si l'argument est surchargé, alors ce dernier n'est pas vide et la variable d'environnement vaudra sa valeur.

Essayons le cas où l'argument « version » n'est pas surchargé :

```
$ docker build -t arg .
...
$ run --rm arg
1.0
```

La variable d'environnement vaut bien « 1.0 ». Surchargeons maintenant l'argument avec « 1.2 » :

```
$ docker build --build-arg version=1.2 -t arg .
...
$ docker run --rm arg
1.2
```

Cela correspond bien au résultat souhaité.

À noter que l'utilisation de cette technique n'empêche pas explicitement la surcharge d'une variable d'environnement lors du démarrage d'un conteneur (le problème d'inconsistance pourrait ainsi avoir lieu), cependant il convient de toujours documenter une image, notamment les variables d'environnement qui peuvent être surchargées au démarrage d'un conteneur associé.

> Une variable d'environnement, qui ne peut être surchargée lors de la construction de l'image, ne doit en aucun cas être utilisée par des instructions de construction de l'image.

Pour terminer cette section, mentionnons que Docker dispose d'un ensemble prédéfini d'arguments qui peuvent être utilisés sans être explicitement définis par des instructions ARG dans un Dockerfile. En voici la liste :

- HTTP_PROXY
- http_proxy
- HTTPS_PROXY
- https_proxy
- FTP_PROXY
- ftp_proxy
- NO_PROXY
- no_proxy

8.1.6 ONBUILD

L'instruction ONBUILD permet de définir des instructions qui seront exécutées uniquement lors de la construction d'images enfants, ce qui signifie qu'elles ne seront pas exécutées lors de la construction de l'image les contenant.

> L'exécution des instructions ONBUILD n'est effectuée que dans les enfants directs. Les éventuels enfants d'images enfants n'incluront pas ces instructions.

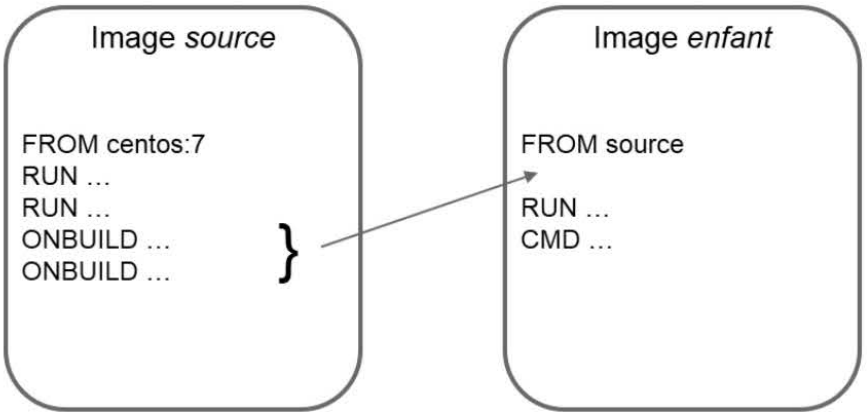

Figure 8.1 — Exécution d'instructions ONBUILD dans une image enfant

La figure 8.1 illustre l'emplacement où sont exécutées dans une image enfant les commandes définies par des instructions ONBUILD dans une image source : directement après l'instruction FROM.

Le modèle de l'instruction ONBUILD est :

ONBUILD <instruction>

`<instruction>` représente une instruction de construction (RUN, ADD, COPY, USER et ARG).

Par exemple :

```
ONBUILD RUN mkdir /tmp/test
ONBUILD COPY test* dossierRelatif/
ONBUILD USER httpd
```

L'utilité de l'instruction ONBUILD n'est pas forcément évidente à comprendre, et c'est pour cela que nous emploierons un exemple pratique pour illustrer un cas d'utilisation réelle : il s'agira de construire une image permettant de compiler et d'exécuter une application Python quelconque.

> À titre d'information, Python est un langage généralement utilisé pour de la programmation fonctionnelle, c'est-à-dire qui considère le calcul en tant qu'évaluation

de fonctions mathématiques. Il peut être interprété ou compilé, et dans notre exemple il sera compilé.

Pour commencer l'exemple, nous avons besoin d'une image source (appelée python-app) capable de compiler et d'exécuter une application Python (dans notre cas, un simple fichier .py) sans connaître cette dernière. Nous aurons le Dockerfile suivant :

```
#python-app
FROM centos:7
ONBUILD ARG fichier="app.py"
ONBUILD COPY $fichier /app/app.py
ONBUILD RUN python -m py_compile /app/app.py
ENTRYPOINT ["python", "/app/app.pyc"]
```

Les instructions ONBUILD vont copier un fichier Python (dont le nom par défaut est app.py) dans l'image et le compiler : le fichier Python dans l'image se nommera dans tous les cas app.py.

Python est déjà présent dans l'image source CentOS 7, ainsi nous n'avons pas besoin de le faire.

Comme cela est expliqué plus haut, les instructions ONBUILD ne sont jouées que dans les images enfants ; ainsi, l'argument fichier (défini par ARG) ne pourra, si nécessaire, être surchargé que lors de la construction d'images enfants, ce qui représente bien notre souhait dans la mesure où l'image source ne connaît pas l'application Python.

L'instruction ONBUILD COPY $fichier /app/app.py fera échouer la construction de l'image enfant si le fichier décrit par l'argument « fichier » n'existe pas. Il n'existe pas de solution directe à ce problème, toutefois il convient de taguer avec le mot-clé « onbuild » toute image incluant une ou plusieurs instructions ONBUILD. L'utilisateur sera ainsi averti et devra prendre la peine de lire la documentation ou au moins le Dockerfile de l'image source.

Construisons maintenant l'image source en n'oubliant pas le tag « onbuild » dans la version :

```
$ docker build -t python-app:1.0-onbuild .
```

Nous avons maintenant notre image source python-app (en version 1.0-onbuild) permettant de compiler et d'exécuter un fichier Python dont le nom par défaut est app.py.

Continuons l'exemple en exploitant l'image source python-app. Pour cela nous avons besoin d'une application Python (qui sera appelée x2) : il s'agira d'un simple programme dont le but est de multiplier une valeur (donnée en argument) par deux et d'afficher le résultat. Voici son code :

```
# x2.py
import sys
x=int(sys.argv[1])
resultat=x*2
print(repr(x)+" * 2 = " +repr(resultat))
```

Commentons sommairement ce programme :

- import sys permet d'importer une librairie permettant d'exploiter les arguments ;
- x=int(sys.argv[1]) signifie qu'on assigne à x le premier argument donné lors de l'exécution de l'application ;
- les autres lignes de codes sont relativement simples à comprendre.

Si nous exécutons l'application (sans la compiler pour simplifier l'explication), nous obtenons :

```
$ python x2.py 5
5 * 2 = 10
```

Créons maintenant le Dockerfile permettant d'exécuter notre application x2 grâce à notre image source générique python-app :

```
FROM python-app:1.0-onbuild
CMD ["0"]
```

Nous avons défini une instruction CMD ["0"] car l'application x2 a besoin d'un argument lors de son exécution (nous avons choisi arbitrairement 0). Pour rappel, il est possible de combiner des instructions ENTRYPOINT et CMD, dans notre cas :

```
ENTRYPOINT ["python", "/app/app.pyc"]
CMD ["0"]
```

produira `python /app/app.pyc 0` où 0 pourra être surchargé lors du démarrage du conteneur.

L'image source introduit un argument « fichier » dans les images enfants dont la valeur par défaut vaut app.py. Dans notre cas, l'application est définie par le fichier x2.py (placé au même endroit que le Dockerfile), et on devra surcharger l'argument lors de la construction de l'image :

```
$ docker build --build-arg fichier=x2.py -t python-x2 .
```

Essayons maintenant de démarrer un conteneur :

```
$ docker run --rm python-x2
0 * 2 = 0
$ docker run --rm python-x2 17
17 * 2 = 34
```

Tout fonctionne correctement : dans le premier cas, python-x2 utilise la valeur d'exécution par défaut, soit 0, et dans le deuxième cas 17 qui représente bien la surcharge de l'instruction CMD.

Revenons un peu en arrière, au niveau du résultat retourné lors de la construction de l'image python-x2. La ligne suivante en faisait partie :

```
# Executing 3 build triggers...
```

Cette ligne particulière est retournée uniquement lorsque l'image source contient des instructions ONBUILD : elle sert à attirer l'attention de l'utilisateur sur le fait que des instructions qui ne sont pas dans le Dockerfile de l'image seront exécutées.

Pour terminer cette section, regardons ce qui se passe techniquement au niveau de la construction d'une image contenant une instruction ONBUILD :

1. À la lecture d'une instruction ONBUILD, le constructeur Docker va créer une métadonnée qu'il placera dans le manifeste de l'image au niveau de la clé « OnBuild ». Cette métadonnée décrira l'instruction qui devra être exécutée dans les images enfants. Il est possible de voir son contenu grâce à un docker inspect sur l'image concernée. Par exemple, pour python-app, nous obtenons :

```
$ docker inspect --format='{{json .ContainerConfig.OnBuild}}' python-app
["ARG fichier=\"app.py\"","COPY $fichier /app/app.py","RUN python -m
py_compile /app/app.py"]
```

2. Nous constatons bien que les trois instructions ONBUILD sont présentes dans « OnBuild ».
3. Après la création de la métadonnée, le constructeur Docker a terminé son travail pour l'instruction concernée.
4. Plus tard, lorsque l'image contenant l'instruction ONBUILD est utilisée comme source, le démon Docker lira le manifeste et constatera que des instructions provenant de l'image source doivent être exécutées : il exécutera ces instructions au niveau de l'instruction FROM, et ceci dans le même ordre où elles ont été ajoutées au manifeste, c'est-à-dire l'ordre dans le Dockerfile de l'image source. Si l'une d'elle échoue, alors c'est l'instruction FROM qui échouera.
5. Si l'exécution des instructions provenant du manifeste est un succès, alors le constructeur supprime les métadonnées concernées dans le manifeste et continue la construction de l'image à partir des instructions suivant le FROM.

8.1.7 STOPSIGNAL

L'instruction STOPSIGNAL permet de définir le signal à exécuter lors de l'arrêt d'un conteneur. Un signal représente une information impactant un processus, par exemple son arrêt immédiat.

Le modèle de l'instruction est :

STOPSIGNAL <signal>

`<signal>` est le signal à utiliser. Il est représenté soit par un nombre définissant sa position dans la table syscall, soit par son nom au format SIGNAME.

Tableau 8.1 — Extrait de la table syscall

Position	Nom au format SIGNAME	Description
1	SIGHUP	Termine la session du processus.
2	SIGINT	Interrompt le processus.
3	SIGQUIT	Quitte le processus.
6	SIGABRT	Annule le processus.
9	SIGKILL	Termine le processus immédiatement.
10	SIGUSR1	Signal utilisateur 1.
12	SIGUSR2	Signal utilisateur 2.
15	SIGTERM	Termine le processus.

Tout signal, sauf SIGKILL et SIGTERM, peut être intercepté par un processus, ce qui signifie que ce dernier le gérera comme il le souhaite.

Par exemple :

```
STOPSIGNAL 9
STOPSIGNAL SIGKILL
```

Pour rappel, lors de l'arrêt d'un conteneur (docker stop), seul le processus de PID 1 est arrêté proprement par Docker ; c'est ce processus qui interprètera le signal décrit par l'instruction STOPSIGNAL (par défaut il s'agira de SIGTERM). L'utilité de STOPSIGNAL est donc de changer ce signal et par conséquent le travail à accomplir par le processus lors de l'arrêt du conteneur. À noter que le signal SIGKILL est également reçu par le processus après une certaine période (de 10 secondes par défaut) si ce dernier ne s'est pas terminé correctement.

8.2 SÉCURISONS DOCKER

La sécurisation d'une architecture à base de conteneurs est un vaste sujet. Il existe cependant deux cas d'usage relativement courants qu'il est utile de connaître :

- la sécurisation de la Docker Remote API avec SSL/TLS ;
- Docker Content Trust, le logiciel de la suite Docker dédié à la signature des images.

8.2.1 SSL/TLS pour la Remote API

L'API distante de Docker (en anglais *Docker Remote API*) a été présentée dans le chapitre 3 de cet ouvrage. Nous allons rappeler ici brièvement comment elle fonctionne et comment la configurer.

Le client Docker permet l'exécution des commandes Docker depuis un terminal. Lorsqu'une commande est exécutée depuis le client, par exemple `docker ps`, une requête HTTP REST correspondante est créée, envoyée et exécutée par le démon Docker à travers un socket. Le démon Docker expose donc toutes les commandes Docker par le biais d'un socket. L'API Docker permet d'envoyer et d'exécuter des requêtes au démon (via le socket) grâce à une interface REST/JSON.

Dans la configuration initiale du démon Docker, le socket par défaut est un socket Unix localisé par /var/run/docker.sock. Ainsi, en exposant ce socket Unix comme un socket HTTP, on peut accéder à l'API. Regardons maintenant comment sécuriser cette API. Nous allons pour cela devoir :

- créer une Autorité de certification (abrégé en CA en anglais) ; bien évidemment, sur une véritable installation, nous utiliserions un certificat délivré par une agence de certification ;
- créer et configurer un *Certificate Signing Request* (abrégé en CSR et traduit par demande de signature de certificat) et une clé pour l'API de notre démon Docker ;
- faire de même pour le client pour pouvoir ensuite interagir avec le démon Docker.

> Nous utiliserons pour cela l'utilitaire `openssl` qui est disponible sur la plupart des distributions Linux.

Création de notre Autorité de certification

Créons avant tout un répertoire où nous stockerons nos clés et certificats.

```
$ mkdir ~/docker-cert
$ cd docker-cert
```

Commençons par créer la clé privée (private key) `ca-key.pem` de notre CA. Ce sera une clé de 4096 bits et nous utiliserons l'algorithme de chiffrement symétrique AES256 :

```
$ echo 01 | tee ca.srl
$ openssl genrsa -aes256 -out ca-key.pem 4096
Generating RSA private key, 4096 bit long modulus .....................++
.....................................................................
.....................................................................++
e is 65537 (0x10001)
Enter pass phrase for ca-key.pem:
Verifying - Enter pass phrase for ca-key.pem:
```

Notez bien la *pass phrase* (phrase secrète) que vous utilisez. C'est le mot de passe de votre clé et nous en aurons prochainement besoin.

Vous avez sûrement noté une commande étrange : `echo 01 | tee ca.srl`. Elle contient un numéro de série qui est incrémenté chaque fois que nous utilisons notre clé pour signer un certificat.

Nous pouvons maintenant générer notre certificat `ca.pem` :

```
$ openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -out ca.pem
Enter pass phrase for ca-key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [XX]:FR
State or Province Name (full name) []:France
Locality Name (eg, city) [Default City]:
Organization Name (eg, company) [Default Company Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (eg, your name or your server's hostname) []:dunod-docker
Email Address []:
```

Nous avons maintenant notre certificat CA valable une année. Utilisons le maintenant pour créer le certificat de notre démon Docker et le sécuriser.

Sécurisation de l'API du démon Docker

Commençons par une clé `server-key.pem` pour notre serveur (c'est-à-dire notre démon) :

```
$ openssl genrsa -out server-key.pem 4096
..++
.++
e is 65537 (0x10001)
```

Créons le CSR `server.csr` pour notre serveur. Cette demande de signature de certificat est propre à notre machine, si bien que vous ne devez pas oublier de remplacer $HOST (sous Linux, vous pouvez simplement l'obtenir avec la commande `hostname`) par le nom de la machine sur laquelle le démon Docker tourne.

```
openssl req -subj "/CN=$HOST" -sha256 -new -key server-key.pem -out server.csr
```

Il ne nous reste plus qu'à signer notre CSR pour générer le certificat de notre serveur :

```
$ openssl x509 -req -days 365 -sha256 -in server.csr -CA ca.pem -CAkey
ca-key.pem -out server-cert.pem
Signature ok
subject=/CN=localhost.localdomain
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

Nous avons maintenant tout ce qu'il nous faut pour sécuriser notre démon. La procédure est identique à ce que nous avons déjà vu au chapitre 3 : il nous faut éditer le fichier de configuration de Docker (disponible sous CentOS dans le fichier /usr/lib/systemd/system/docker.service) et adapter la ligne avec l'instruction ExecStart :

```
$ sudo -i
$ vi /usr/lib/systemd/system/docker.service
...
ExecStart=/usr/bin/docker daemon -H tcp://0.0.0.0:2376 '--tlsverify'
'--tlscacert=/home/vagrant/docker-cert/ca.pem'
'--tlscert=/home/vagrant/docker-cert/server-cert.pem'
'--tlskey=/home/vagrant/docker-cert/server-key.pem'...
$ exit
```

Redémarrons notre démon Docker :

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

Il ne nous reste plus qu'à créer un certificat et une clé pour notre client afin de nous connecter sur notre démon Docker maintenant sécurisé.

Configuration SSL du client Docker

Le processus est presque identique à ce que nous venons de réaliser pour notre démon : création de la clé et d'un CSR client, puis signature de notre CSR avec notre CA pour générer le certificat :

```
$ openssl genrsa -out key.pem 4096
...
$ openssl req -subj '/CN=client' -new -key key.pem -out client.csr
```

Il existe une petite différence avec la partie serveur : nous devons activer l'authentification client pour notre clé en rajoutant l'attribut SSL extendedKeyUsage :

```
$ echo extendedKeyUsage = clientAuth > extfile.cnf
```

et nous générons finalement notre certificat client :

```
$ openssl x509 -req -days 365 -sha256 -in client.csr -CA ca.pem -CAkey
ca-key.pem -out cert.pem -extfile extfile.cnf
Signature ok
subject=/CN=client
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

Il ne nous reste plus qu'à passer en paramètre à notre client Docker les différents clés/certificats que nous venons de créer :

```
$ docker -H=tcp://$HOST:2376 --tlsverify
--tlscacert=/home/vagrant/docker-cert/ca.pem
--tlscert=/home/vagrant/docker-cert/cert.pem
--tlskey=/home/vagrant/docker-cert/key.pem info
```

N'oubliez pas de remplacer $HOST par le nom de l'hôte sur lequel tourne le démon Docker !

Et si nous essayons sans les paramètres SSL :

```
$ docker -H=tcp://$HOST:2376 info
Get http://localhost:2376/v1.23/info: malformed HTTP response
"\x15\x03\x01\x00\x02\x02". * Are you trying to connect to a TLS-enabled
daemon without TLS?
```

Docker nous signale effectivement que nous ne pouvons pas accéder au démon. Nous disposons maintenant d'un démon totalement sécurisé. Heureusement, comme nous l'avons vu au chapitre 4, des outils comme Docker Machine permettent d'automatiser cette configuration standard qui est fastidieuse si on doit la réaliser manuellement.

8.2.2 Docker Content Trust

En début de chapitre 7, nous avons listé les variables d'environnement utilisées par le Docker Engine. DOCKER_CONTENT_TRUST et DOCKER_CONTENT_TRUST_SERVER sont utilisées dans le cadre de Docker Content Trust.

Docker Content Trust est un mécanisme qui permet de signer puis vérifier une image qui se trouve dans un registry. Il se base pour cela sur le projet open source Notary[1] incluant un client et un serveur et permettant d'interagir avec des collections de confiance.

Que cela peut-il bien signifier en pratique et quelles sont les conséquences sur la création d'images ? Le plus simple est, comme toujours, de prendre un exemple.

Activons tout d'abord temporairement Docker Content Trust :

1. https://docs.docker.com/notary/

```
$ export DOCKER_CONTENT_TRUST=1
```

Cette variable n'est configurée que pour la session bash active. Si vous désirez l'avoir active en permanence, il vous faudra la définir de manière permanente pour l'utilisateur, typiquement en la rajoutant dans votre `~/.bashrc`

Créons un Dockerfile minimal :

```
FROM alpine:3.1
CMD ping localhost
```

Construisons notre image en la taguant de telle façon que nous puissions ensuite la pousser sur le Docker Hub. Nous devons pour cela spécifier le nom du dépôt (dunoddocker) :

```
$ docker build -t dunoddocker/content_trust:latest .
Sending build context to Docker daemon 3.786 MB
Step 1 : FROM alpine:3.1
3.1: Pulling from library/alpine
40f9ed72912e: Pull complete
Digest:
sha256:f3d4f10120752f738efeee4e639d4767110a2eb10c9632aa861b5d5eb5af7e35
Status: Downloaded newer image for alpine:3.1
 ---> 885194f4c89a
Step 2 : CMD ping localhost
 ---> Running in 51bf540dad6e
 ---> 6282600887dc
Removing intermediate container 51bf540dad6e
Successfully built 6282600887dc
```

Nous avons maintenant une image prête à être poussée sur le Docker Hub. C'est à partir d'ici que les choses diffèrent d'un push normal. Poussons notre image avec la commande standard docker push

```
$ docker push dunoddocker/content_trust:latest
The push refers to a repository [docker.io/dunoddocker/content_trust]
8fe0a3ef03d9: Preparing
unauthorized: authentication required
```

Nous avons effectivement oublié de nous identifier sur le Docker Hub. Utilisons la commande docker login avant de pousser notre image :

```
$ docker login -u dunoddocker
Password :
Login Succeeded
$ docker push dunoddocker/content_trust:latest
```

Le push commence normalement et vous obtenez ensuite un message vous demandant de créer une phrase secrète et cela deux fois de suite.

```
Signing and pushing trust metadata You are about to create a new root signing
key passphrase. This passphrase will be used to protect the most sensitive key
in your signing system. Please choose a long, complex passphrase and be
careful to keep the password and the key file itself secure and backed up. It
is highly recommended that you use a password manager to generate the
passphrase and keep it safe. There will be no way to recover this key. You can
find the key in your config directory.
Enter passphrase for new root key with ID b059adb:
Repeat passphrase for new root key with ID b059adb:
Enter passphrase for new repository key with ID 0b06cba
(docker.io/dunoddocker/content_trust):
Repeat passphrase for new repository key with ID 0b06cba
(docker.io/dunoddocker/content_trust):
Finished initializing "docker.io/dunoddocker/content_trust"
Successfully signed "docker.io/dunoddocker/content_trust":latest
```

En effet, Docker Content Trust repose sur l'utilisation de deux clés distinctes :

- la clé de *tagging* : elle est générée pour chaque dépôt. Elle peut facilement être partagée avec quiconque voudrait pouvoir signer du contenu pour ce dépôt. Elle est stockée dans le répertoire ~/.docker/trust/private/tuf_keys/docker.io/<reponame> ;
- la clé offline : c'est la clé la plus importante car c'est la clé root. Il est donc critique de bien la protéger, car au contraire des clés de *tagging*, il n'est pas possible de la récupérer. On peut d'ailleurs la trouver dans le dossier ~/.docker/trust/private/root_keys.

En vérité, il y a même une troisième clé, dite de *timestamp*. Elle est stockée côté serveur et permet de s'assurer de la fraîcheur de l'image.

Maintenant que nous avons une image sur le Docker Hub, tout utilisateur voulant utiliser notre image aura la garantie que ce sera bien celle du publicateur (car « buildé » par lui). De plus, grâce au mécanisme de clé de *timestamp*, il aura aussi la garantie que ce sera la dernière image pour le tag.

Il y a cependant une contrainte à bien comprendre : si vous activez Docker Content Trust et que vous désirez « puller » une image qui n'est pas signée, vous aurez un message d'erreur du type :

```
Using default tag: latest
Error: remote trust data does not exist for docker.io/xxx/yyy:
notary.docker.io does not have trust data for docker.io/xxx/yyy
```

L'astuce est de rajouter le paramètre `--disable-content-trust` à la commande `docker pull` pour ignorer cette contrainte de manière ponctuelle.

Docker Content Trust est relativement récent (Docker 1.8), mais il répond à une vraie attente : garantir l'intégrité des images, particulièrement lorsque celles-ci sont distribuées via des réseaux non sécurisés tels qu'Internet. Toutes les images officielles disponibles sur le Docker Hub sont d'ailleurs maintenant signées par Docker Inc.

8.3 INSTALLER UN REGISTRY PRIVÉ

Nous avons expliqué dans le chapitre 1 le principe de registry Docker. Nous avons ensuite jusqu'ici (et nous le ferons encore dans les chapitres suivants) utilisé essentiellement le registry public de Docker : le Docker Hub.

Cependant, nous avons aussi évoqué le fait qu'il était possible d'installer un registry Docker privé. Dans cette section qui traite de cette fonctionnalité, nous utiliserons l'image officielle Docker qui est disponible à l'adresse suivante : https://hub.docker.com/_/registry/

8.3.1 Mot de passe et certificats

Nous avons besoin de créer pour notre registry :

- un compte (login/password) ;
- des certificats SSL auto-signés pour sécuriser l'API du registry avec HTTPS.

Pour ce faire, nous allons utiliser un conteneur qui va générer ces fichiers dans des volumes qu'il nous suffira ensuite de brancher sur notre registry.

Pour créer le login/password, nous aurons besoin de l'utilitaire htpasswd qui permet de générer une entrée cryptée dans un fichier, qui sera ensuite utilisé pour l'authentification (en mode http basic). Cet utilitaire est disponible dans le repository EPEL https-tools.

Pour la création des certificats, nous emploierons de nouveau l'utilitaire `openssl`.

Voici le Dockerfile de notre conteneur de configuration :

```
FROM centos:7

#Installe le repository EPEL
RUN yum install -y --setopt=tsflags=nodocs epel-release && yum clean all

#Installe les deux utilitaires dont nous avons besoin
RUN yum install -y --setopt=tsflags=nodocs httpd-tools openssl && yum clean
all

#Identifie deux volumes destinés à recevoir les fichiers
RUN mkdir -p /opt/certs
RUN mkdir -p /opt/auth
VOLUME /opt/certs
VOLUME /opt/auth
```

```
COPY entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh

ENTRYPOINT /entrypoint.sh
```

Il fait usage d'un script ENTRYPOINT nommé entrypoint.sh que vous prendrez soin de disposer dans le même répertoire que le Dockerfile. Voici son contenu :

```
#!/bin/bash
#Création du premier compte
echo "Creation d'un compte pour l'utilisateur $REG_LOGIN"
htpasswd -Bbn $REGLOGIN $REGPASSWD > /opt/auth/htpasswd
#Création des certificats
echo "Creation des certificats SSL"
openssl req -newkey rsa:4096 -nodes -sha256 -keyout /opt/certs/domain.key
-x509 -days 365 -out /opt/certs/domain.crt -subj '/CN=localhost'
```

Notez l'usage de deux variables d'environnement REGLOGIN et REGPASSWD. Celles-ci devront être passées en paramètre du `docker run`.

Nous pouvons maintenant construire notre image :

```
$ docker build -t regconfig .
```

8.3.2 Exécution de notre conteneur de configuration

Nous allons maintenant créer deux volumes nommés qui vont recevoir les fichiers à créer :

- un volume certs_volume qui va recevoir les clefs publique et privée SSL ;
- un volume auth_volume qui recevra le fichier htpasswd contenant le login/password encrypté.

Voici les instructions requises :

```
$ docker volume create --name=certs_volume
$ docker volume create --name=auth_volume
```

Nous pouvons maintenant lancer notre conteneur avec la commande suivante :

```
$ docker run --rm --env REGLOGIN=admin --env REGPASSWD=admin1234 -v
certs_volume:/opt/certs -v auth_volume:/opt/auth regconfig
```

Voici quelques explications relatives à la ligne de commande ci-dessus :

- `--rm` va faire en sorte que ce conteneur s'autodétruise après usage. Il s'agit clairement d'un conteneur de type script ;

- `--env` spécifie les valeurs pour les deux variables d'environnement (REGLOGIN et REGPASSWD) contenant respectivement le login et le mot de passe de notre compte sur le registry ;
- nous associons les volumes nommés à des chemins du conteneur :
  - certs_volume va être associé au chemin /opt/certs et il recevra les certificats SSL générés par openssl,
  - auth_volume va être associé au chemin /opt/auth et il contiendra le fichier htpasswd.

Si vous regardez le fichier entrypoint.sh plus haut, vous verrez que les différents fichiers sont bien créés dans les chemins qui pointent vers nos deux volumes.

Voici le résultat de l'exécution de notre conteneur :

```
$ docker run --rm --env REG_LOGIN=admin --env REG_PASSWD=admin1234 -v
certs_volume:/opt/certs -v auth_volume:/opt/auth regconfig
Creation d'un compte pour l'utilisateur admin
Creation des certificats SSL
Generating a 4096 bit RSA private key
.....................................++
...................++
writing new private key to '/opt/certs/domain.key'
-----
```

Nous pouvons maintenant créer un conteneur pour aller vérifier que nos deux volumes contiennent bien les fichiers dont nous avons besoin :

```
$ docker run --rm -t -i -v certs_volume:/opt/certs -v auth_volume:/opt/auth
centos:7 /bin/bash
[root@42bda61680f6 /]# ls /opt/certs/
domain.crt  domain.key
[root@42bda61680f6 /]# ls /opt/auth/
htpasswd
```

C'est effectivement le cas !

8.3.3 Lancement du registry

La création de notre registry repose sur la commande suivante :

```
docker run -d -p 5000:5000 --restart=always --name registry \
  -v auth_volume:/auth \
  -v certs_volume:/certs \
  -e "REGISTRY_AUTH=htpasswd" \
  -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
  -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
```

```
  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
  -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
  registry:2
```

Vous noterez :

- que nous associons le volume auth_volume (qui contient le fichier htpasswd) au chemin /auth ;
- que nous associons le volume certs_volume (qui contient les deux clefs SSL domain.crt et domain.key) au chemin /certs.

Le registry va aller puiser dans ces deux chemins pour y trouver le login/password et les certificats SSL.

Une fois le registry lancé, vérifions que nous pouvons nous y connecter :

```
$ docker login localhost:5000
Username (admin): admin
Password:
Login Succeeded
```

Rappel : le mot de passe défini précédemment est « admin1234 »

8.3.4 Pousser une image

Poussons, par exemple, notre image de script de configuration regconfig définie plus haut dans notre nouveau registry.

Nous devons d'abord taguer notre image à l'aide de l'instruction suivante :

```
$ docker tag regconfig localhost:5000/regconfig
```

Celle-ci crée une nouvelle entrée associant l'image que nous avions précédemment construite à un tag dont le nom est compatible avec notre registry :

```
 $ docker images
REPOSITORY                TAG                 IMAGE ID            CREATED
SIZE
localhost:5000/regconfig  latest              eb33ef7afd99        21 minutes
ago      235.1 MB
regconfig                 latest              eb33ef7afd99        21 minutes
ago      235.1 MB
```

Nous voyons clairement que la même image eb33ef7afd99 a maintenant deux noms différents dans notre cache local.

Nous sommes à présent prêts à pousser notre image dans le registry :

```
$ docker push localhost:5000/regconfig
The push refers to a repository [localhost:5000/regconfig]
fc2ea32faa0a: Pushed
9a04fce8334f: Pushed
2318cd41fc71: Pushed
267a1753b916: Pushed
bd7029ca7d8e: Pushed
21d4ec20adbe: Pushed
5f70bf18a086: Pushed
6a6c96337be1: Pushed
latest: digest:
sha256:445da39665fe0315fa3ff76e4ee4466e9e12bad01d6d7fa370a852dd5c3a1f27 size:
2397
```

Nous pouvons vérifier que l'image est bien présente en utilisant curl pour interroger l'API REST du registry (sans oublier le login/password) :

```
$ curl -u admin:admin1234 -k https://localhost:5000/v2/_catalog
{"repositories":["regconfig"]}
```

Notre image est bien chargée !

Le flag `-k` de curl est nécessaire car nos certificats SSL sont *self-signed*. Ils seraient donc rejetés par curl sans ce paramètre.

En résumé

Les trois chapitres de cette partie vous ont permis d'apprendre à fabriquer des images, mais aussi à piloter le Docker Engine au travers de sa ligne de commande.

Il est maintenant temps d'aborder des exemples plus complexes mettant en jeu des concepts plus pointus.

QUATRIÈME PARTIE

Exemples d'architectures et concepts avancés

Cette dernière partie comprend trois chapitres qui sont autant d'exemples de mise en œuvre d'architectures à base de conteneurs.

Outre leur sujet propre, ces chapitres nous permettent aussi d'expliquer des concepts avancés que nous n'avons pas encore pu complètement aborder.

Ainsi cette partie comprend les chapitres suivants :

- le chapitre 9 traite des architectures multi-conteneurs et nous permet d'aborder l'usage d'outils comme Supervisor, mais aussi de comprendre en détail la gestion du réseau avec Docker (et plus spécifiquement du réseau de type bridge) ;
- le chapitre 10 montre comment utiliser Docker dans le cas d'un environnement de développement « intégration continue ». Il nous permet d'aborder l'usage des conteneurs comme outils de déploiement, mais aussi la notion de Docker API, conteneur nécessaire pour utiliser Docker depuis un conteneur ;
- le chapitre 11 expose un exemple de mise en œuvre de la solution Docker Swarm (que nous avons décrite dans le chapitre 2). Il permet d'aborder le dernier modèle de réseau Docker : le modèle *overlay*.

Chaque chapitre s'appuie sur des cas pratiques mettant en œuvre l'environnement que vous avez créé dans le chapitre 3.

9

Application multi-conteneurs

Rendre modulaire son infrastructure

L'objectif de ce chapitre est d'expliquer comment utiliser Docker pour le développement d'application. Nous lèverons au passage le voile sur les communications intra-conteneurs.

À l'issue de ce chapitre, vous aurez compris les bases du réseau Docker, ainsi que différentes techniques pour tirer tous les bénéfices d'une infrastructure « dockerisée ».

Tout au long de ce chapitre, nous allons travailler avec une application web basée sur le framework PHP Symfony2[1]. Nous aurons besoin pour cela :

- d'un serveur web. Nous utiliserons Nginx que vous avez déjà rencontré dans les chapitres 3 et 5 ;
- de PHP-FPM[2] pour exécuter notre application. PHP Fast Process Manager est une implémentation FastCGI PHP spécialement adaptée aux sites à forte charge ;
- et d'une base de données MariaDB[3] pour la persistance.

Nous allons suivre plusieurs approches pour « dockeriser » notre application :

- tout d'abord, nous n'utiliserons qu'un conteneur et verrons rapidement les limitations de ce modèle pour un environnement de production ;
- nous répartirons ensuite nos différents moteurs d'exécution sur plusieurs conteneurs, afin de mettre en place une architecture de micro-services ;

1. https://symfony.com/
2. http://php.net/manual/en/install.fpm.php
3. http://mariadb.org/

- nous terminerons ensuite avec le meilleur des deux mondes en découvrant Docker Compose[1].

À l'occasion, nous découvrirons aussi quelques astuces qui nous permettront de mieux appréhender tous les avantages à utiliser un tel environnement, aussi bien pour le développement que pour la mise en production d'une application.

9.1 UN SEUL CONTENEUR, PLUSIEURS PROCESSUS

La première idée que nous allons mettre en œuvre est la suivante : fabriquer une image Docker fournissant tous les services nécessaires à notre application (Nginx, PHP-FPM et MariaDB).

Schématiquement, le conteneur résultant aura cet aspect :

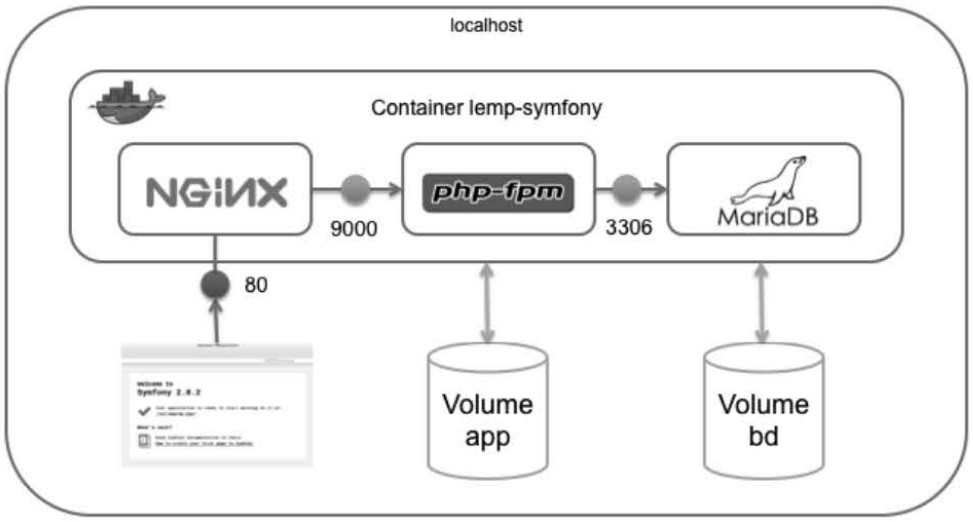

Figure 9.1 — Application mono-conteneur

Quelques explications s'imposent :

- nous aurons donc un seul conteneur LEMP[2] qui exposera le port 80 de Nginx. Les autres ports nécessaires à notre application ne seront pas exposés sur notre machine hôte ;
- nous utiliserons deux volumes : un qui contiendra le code source de notre application et un autre pour les fichiers de notre base de données.

Avec cette configuration, nous pourrons détruire/reconstruire/modifier notre conteneur sans risque de perdre des informations. En effet, notre code source et notre base de

1. https://www.docker.com/products/docker-compose
2. LEMP : Linux, Nginx, MySQL/MariaDB, PHP

données seront accessibles pour notre machine hôte et faciliteront le développement de notre application.

Si ce n'est déjà fait, récupérez les fichiers nécessaires avec la commande suivante :

```
$ git clone https://github.com/dunod-docker/application-multi-conteneurs.git
```

Tous les fichiers dont nous aurons besoin sont disponibles dans le répertoire supervisor. Nous allons décomposer petit à petit le Dockerfile qui va nous permettre ensuite de créer l'image de notre futur conteneur.

9.1.1 Installation des moteurs d'exécution

Le Dockerfile que nous allons utiliser se trouve directement à la racine du répertoire supervisor. Notre image sera basée sur une distribution CentOS 7. Nous utiliserons pour cela l'image officielle fournie par le Docker Hub :

```
FROM centos:7
```

Nous utilisons ensuite `yum` pour :

- installer le dépôt EPEL. Celui-ci nous permet d'obtenir des versions plus récentes des différents programmes qu'en utilisant le dépôt standard de CentOS ;
- mettre à jour les paquets déjà présents sur notre image ;
- installer Nginx, les différents paquets PHP nécessaires à Symfony dont PHP-FPM, MariaDB (serveur et client) et finalement Supervisor. Nous verrons un peu plus loin ce qu'est Supervisor.

```
# Dockerfile
RUN yum install -y epel-release && \
    yum update -y && \
    yum install -y mariadb-server mariadb-client\
                   nginx \
                   php-common php-cli php-fpm php-mysql php-apcu php-curl
php-intl php-mbstring \
                   supervisor && \
    yum clean all
```

Cette façon d'écrire notre Dockerfile suit les bonnes pratiques de Docker. Elle offre un juste milieu entre lisibilité et nombre de couches pour notre image finale.

Maintenant que nous avons installé nos moteurs d'exécution, il ne nous reste plus qu'à les configurer. Commençons tout d'abord par Nginx.

9.1.2 Nginx

La façon dont Nginx et ses différents processus fonctionnent est déterminée via des fichiers de configuration. Nous aurons besoin de deux fichiers :

- le fichier principal de configuration de Nginx. Par défaut, ce fichier qui se nomme *nginx.conf*, est présent dans le répertoire /etc/nginx/ sous CentOS 7 ;
- un fichier de configuration par site web, soit, dans notre cas, un fichier *symfony.conf* qui contiendra la configuration Nginx propre à notre application.

Nous allons tout d'abord créer notre propre fichier *nginx.conf* qui remplacera ensuite celui créé lors de l'installation de Nginx :

```
worker_processes 2;
pid /var/run/nginx.pid;

events { }

http {
    sendfile on ;
    default_type application/octet-stream;
    include /etc/nginx/mime.types;
    include /etc/nginx/sites-enabled/*;
    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;
}
```

L'explication exhaustive du contenu de ce fichier dépassant nettement le périmètre de cet ouvrage, nous n'en expliquerons que les parties pertinentes :

- notre serveur Nginx utilisera deux processus pour répondre aux requêtes (`worker_processes 2;`) ;
- la configuration de notre serveur Symfony (le fichier *symfony.conf*) sera disponible dans le répertoire /etc/nginx/sites-enabled/. Nous le déclarons donc via un *include* pour que Nginx tienne compte de ce répertoire lors du démarrage.

Symfony propose une configuration minimale[1] pour Nginx que nous allons utiliser dans notre fichier *symfony.conf*. Elle se base sur la capacité de Nginx à définir des *server blocks* (l'équivalent des hôtes virtuels sous Apache) :

```
server {
    listen 80;
    server_name symfony;
    root /var/www/my_app/web;

    location / { try_files $uri /app.php$is_args$args; }

    location ~ ^/(app_dev|config)\.php(/|$) {
        fastcgi_pass 127.0.0.1:9000;
```

1. http://symfony.com/doc/2.8/cookbook/configuration/web_server_configuration.html#nginx

```
        fastcgi_split_path_info ^(.+\.php)(/.*)$;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME  $realpath_root$fastcgi_script_name;
        fastcgi_param DOCUMENT_ROOT $realpath_root;
      }

    location ~ ^/app\.php(/|$) {
        fastcgi_pass 127.0.0.1:9000;
        fastcgi_split_path_info ^(.+\.php)(/.*)$;
        include fastcgi_params;
        fastcgi_param  SCRIPT_FILENAME  $realpath_root$fastcgi_script_name;
        fastcgi_param DOCUMENT_ROOT $realpath_root;
        internal;
      }

    error_log /var/log/nginx/symfony_error.log;
    access_log /var/log/nginx/symfony_access.log;
}
```

Nous voyons que :

- notre application sera disponible sur le port 80 de notre conteneur ;
- le dossier qui servira de répertoire racine sera /var/www/my_app/web, notre application Symfony devant donc être disponible dans ce dossier ;
- dès qu'une requête HTTP sera émise à destination de l'application, elle sera déléguée à PHP-FPM à l'adresse 127.0.0.1 (ou localhost) sur le port 9000.

Nos deux fichiers sont ensuite copiés dans notre image, dans les répertoires standards de configuration de Nginx, via les instructions COPY du Dockerfile. Il ne nous reste plus qu'à rendre notre site disponible (en créant un lien symbolique de notre fichier *symfony.conf* du répertoire sites-enabled vers sites-available) grâce à une commande RUN et créer notre répertoire racine :

```
COPY nginx/nginx.conf /etc/nginx/nginx.conf
COPY nginx/symfony.conf /etc/nginx/sites-available/
RUN mkdir -p /etc/nginx/sites-enabled/ && \
    ln -s /etc/nginx/sites-available/symfony.conf
/etc/nginx/sites-enabled/symfony && \
    mkdir -p /var/www/
```

9.1.3 PHP-FPM

La configuration de PHP-FPM suit la même logique que pour Nginx. Nous allons tout d'abord créer, puis copier les fichiers de configuration dans le conteneur, à savoir :

- un fichier de configuration propre à Symfony : *symfony.ini* ;
- un fichier de configuration pour le processus PHP-FPM, *symfony.pool.conf*.

Symfony impose d'avoir le paramètre `date.timezone` défini au niveau de PHP. L'installation de PHP ne le fournissant pas par défaut, il nous faut donc le rajouter via notre fichier *symfony.ini*.

```
date.timezone=Europe/Paris
```

Ensuite, nous devons créer un autre fichier *symfony.pool.conf*, propre à PHP-FPM, qui contient la configuration du processus qui traitera les demandes transmises par Nginx. À nouveau, nous ne décrivons ci-dessous que les parties intéressantes :

```
[symfony]
user = apache
group = apache
...
listen = 0.0.0.0:9000
...
catch_workers_output = yes

php_admin_value[error_log] = /var/log/php-fpm/symfony_error.log
php_admin_flag[log_errors] = on

php_value[session.save_handler] = files
php_value[session.save_path] = /var/lib/php/session
```

Nous remarquons que :

- notre processus utilisera l'utilisateur « apache »[1] ;
- toute requête arrivant sur le port 9000 sera acceptée et traitée par ce pool de connexion ;
- nous définissons un fichier de log propre à notre application (symfony_error.log) ;
- nous plaçons les fichiers de sessions qui seront créées par notre application dans un répertoire dédié.

Occupons-nous maintenant de notre Dockerfile :

- nous allons copier nos deux fichiers de configuration dans notre image avec la commande COPY ;
- nous supprimons le fichier de configuration par défaut de PHP-FPM *www.conf* puis créons le répertoire pour stocker les données des sessions Symfony (/var/lib/php/session), et donner les droits à l'utilisateur apache (de sorte que le processus PHP-FPM puisse y accéder) ;
- nous terminons en récupérant l'installateur Symfony (il nous sera utile pour initialiser notre application) et en le rendant exécutable.

Voici la section du Dockerfile résultante :

1. Cet utilisateur, qui est créé par l'installation de PHP-FPM, est utilisé par défaut. Nous avons donc choisi de le garder.

```
#Adaptation de l'UID du user apache
RUN usermod -u 1000 apache

#Copie de la configuration php pour Symfony
COPY php-fpm/symfony.ini /etc/php.d/
COPY php-fpm/symfony.pool.conf /etc/php-fpm.d/
RUN rm -rf /etc/php-fpm.d/www.conf && \
    mkdir -p /var/lib/php/session && \
    chown -R apache:apache /var/lib/php
RUN curl -LsS https://symfony.com/installer -o /usr/local/bin/symfony && \
    chmod a+x /usr/local/bin/symfony
```

Certains auront remarqué que nous n'avons pas expliqué la commande `usermod -u 1000 apache`. Il s'agit d'une modification indirectement imposée pour assurer une communication correcte entre le conteneur et son hôte au travers des volumes. Nous y reviendrons en temps voulu.

9.1.4 MariaDB

La configuration de MariaDB est relativement rapide : nous lançons le script standard d'installation (mysql_install_db) qui crée les schémas ainsi que l'utilisateur de base et rajoutons les droits à l'utilisateur mysql (l'utilisateur par défaut de MariaDB) sur le répertoire où sont stockées les données.

```
RUN /usr/bin/mysql_install_db > /dev/null && \
    chown -R mysql:mysql /var/lib/mysql/
```

9.1.5 Ports et volumes

Il ne nous reste plus qu'à exposer le port 80 de Nginx et à définir deux volumes :

- un pour le code de notre application (/var/www/) ;
- un pour nos données MariaDB (/var/lib/mysql).

```
EXPOSE 80
VOLUME ["/var/lib/mysql/", "/var/www/"]
```

Voilà ! Nous avons nos trois programmes installés et configurés. Il ne nous reste plus qu'à ajouter une commande CMD (ou ENTRYPOINT). Mais laquelle ? Nous allons nous heurter à une limitation de Docker : un seul processus peut tourner en mode *foreground* (bloquant) par conteneur.

Comment donc faire tourner nos serveurs Nginx, PHP-FPM et MariaDB ?

Pour ce faire, il nous faut utiliser un gestionnaire de services : nous allons employer Supervisor[1].

1. http://supervisord.org/

9.1.6 Supervisor

Supervisor (ou plus précisément son démon supervisord) est, comme son nom l'indique, un superviseur de processus. C'est lui qui sera le processus bloquant unique du conteneur (celui avec le PID 1). Il va devoir démarrer, puis contrôler tous les autres processus à savoir Nginx, PHP-FPM et MariaDB.

Sa configuration est stockée dans le fichier *supervisord.conf* :

```
[supervisord]
nodaemon=true
logfile=/var/log/supervisor/supervisord.log

[program:php-fpm]
command=/usr/sbin/php-fpm -c /etc/php-fpm.d
autorestart=true

[program:nginx]
command=/usr/sbin/nginx -g "daemon off;"
autorestart=true

[program:mysql]
command=/usr/bin/mysqld_safe
```

Nous définissons dans les blocs `[program]` les processus que Supervisor devra gérer. Sans entrer dans les détails, le paramètre `command` correspond à la commande qui va lancer le processus fils. Notez qu'il est possible de demander à Supervisor de redémarrer automatiquement un processus qui se serait arrêté avec la commande `autorestart=true`.

Il ne nous reste plus qu'à copier ce fichier et à configurer notre conteneur pour ne lancer que Supervisor au démarrage, le laissant ensuite démarrer et gérer les autres processus :

```
ADD supervisor/supervisord.conf /etc/supervisor/conf.d/supervisord.conf

CMD ["/usr/bin/supervisord", "-n", "-c",
"/etc/supervisor/conf.d/supervisord.conf"]
```

9.1.7 Notre code source

Nous avons maintenant une image totalement fonctionnelle. Avant de la fabriquer à l'aide de la commande `docker build`, il nous reste une dernière chose à faire : créer sur notre machine de développement le répertoire qui sera utilisé comme volume.

Nous les créons simplement avec la commande :

```
$ mkdir -p ~/symfony-app/code
```

Il est maintenant temps d'expliquer pourquoi nous avons changé l'UID de l'utilisateur apache un peu plus haut.

La gestion des droits entre hôte et conteneur

Dans notre exemple, le processus qui va lire notre code source pour le présenter à travers l'interface HTTP de notre application est PHP-FPM. Nous avons vu précédemment que celui-ci tournait avec l'utilisateur apache. Il est donc nécessaire que cet utilisateur ait des droits en lecture sur les fichiers PHP (le code source).

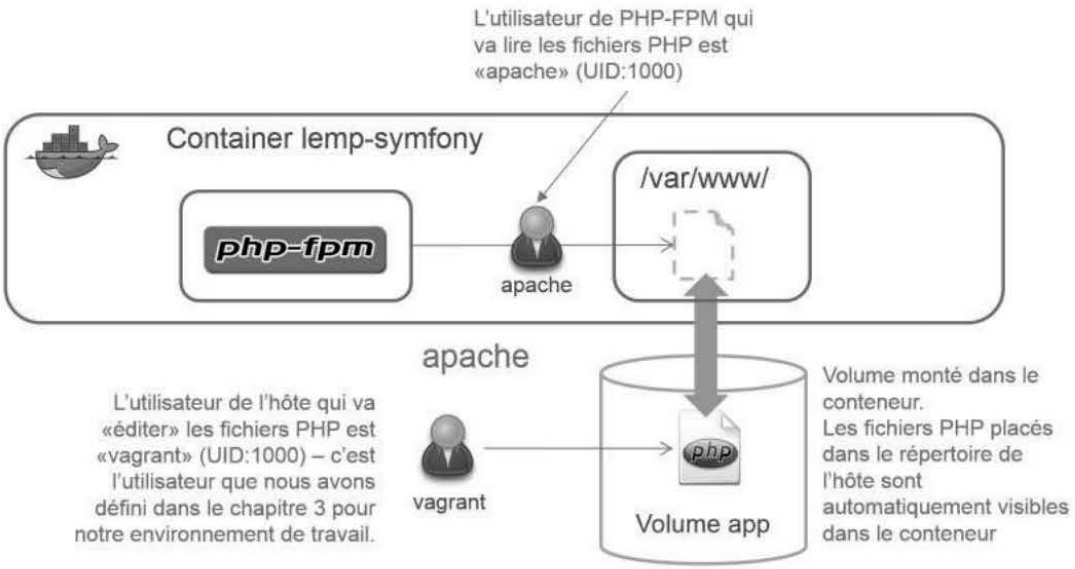

Figure 9.2 — Gestion des droits des fichiers dans les volumes montés

Nous avons décidé de créer un volume entre notre hôte et notre invité afin de pouvoir modifier le code source depuis l'extérieur du conteneur.

C'est là que la question de l'UID entre en jeu. Le nom de l'utilisateur importe peu, mais il est essentiel que les deux utilisateurs, *vagrant* à l'extérieur et *apache* à l'intérieur aient le même UID pour disposer des mêmes droits d'accès. Dans le cas contraire, les fichiers édités par notre utilisateur *vagrant* seraient illisibles pour notre utilisateur *apache*.

Or sur notre hôte, la commande suivante nous apprend que l'utilisateur *vagrant* est associé à l'UID 1000 :

```
$ cat /etc/passwd|grep vagrant
vagrant:x:1000:1000::/home/vagrant:/bin/bash
```

Par défaut, l'utilisateur apache créé par le processus PHP-FPM dans le conteneur est associé à l'UID 48 :

```
[root@9311ce834a53 /]
# cat /etc/passwd | grep apache
apache:x:48:48:Apache:/usr/share/httpd:/sbin/nologin
```

Ceci explique la raison pour laquelle nous lançons la commande suivante dans notre Dockerfile :

```
RUN usermod -u 1000 apache
```

Celle-ci fait simplement en sorte que l'utilisateur *apache* puisse lire les fichiers produits par notre utilisateur *vagrant*.

Création de l'image et lancement du conteneur

Construisons maintenant notre image :

```
$ docker build --tag=lemp-symfony .
Sending build context to Docker daemon 14.34 kB
Step 1 : FROM centos:7
 ---> c8a648134623
Step 2 : MAINTAINER docker-ecosysteme@dunod.com
 ---> Running in d67413cdcb4f
 ---> 5bc8b8e0c35f
Removing intermediate container d67413cdcb4f
Step 3 : RUN yum install -y epel-release &&     yum update -y &&     yum
install -y mariadb-server mariadb-client                    nginx
php-common php-cli php-fpm php-mysql php-apcu php-curl php-intl php-mbstring
supervisor &&     yum clean all
 ---> Running in 7b5592f42f3d
...
```

Nous pouvons vérifier que cette nouvelle image est effectivement disponible grâce à la commande `docker images`, avec comme paramètre le nom de notre image (lemp-symfony) :

```
$ docker images lemp-symfony

REPOSITORY    TAG      IMAGE ID      CREATED         VIRTUAL SIZE
lemp-symfony  latest   e7d3ca4528df  57 seconds ago  435.4 MB
```

Lançons notre conteneur multiprocessus :

```
$  docker run -d -p 8000:80 -v $HOME/symfony-app/code/:/var/www/ --name=lemp
lemp-symfony
```

Nous exposons le port 80 de Nginx sur le port 8000 de notre hôte. Nous montons aussi le volume pour notre code source $HOME/symfony-app/code sous /var/www.

> Attention, le nom du chemin vers le répertoire de l'hôte dépend de l'endroit où vous avez exécuté la commande `mkdir -p symfony-app/code`. Ici nous supposons que vous avez suivi nos instructions et ce répertoire devrait se trouver dans le home de l'utilisateur *vagrant*.

Les données de MariaDB seront stockées dans un volume Docker non exposé à l'hôte. Nous aurions aussi pu choisir de définir un volume nommé afin de ne pas perdre les données en cas d'arrêt ou de destruction du conteneur. Nous nous contenterons de cette approche simplifiée pour les besoins de cet exercice.

Voyons si tout est en ordre en visualisant l'arbre des processus à l'intérieur du conteneur grâce à la commande suivante :

```
$ docker exec -it lemp ps axf
PID TTY      STAT   TIME COMMAND
  182 ?      Rs+    0:00 ps axf
    1 ?      Ss     0:00 /usr/bin/python /usr/bin/supervisord -n -c /etc/super
    8 ?      S      0:00 nginx: master process /usr/sbin/nginx -g daemon off;
   71 ?      S      0:00  \_ nginx: worker process
   72 ?      S      0:00  \_ nginx: worker process
    9 ?      S      0:00 php-fpm: master process (/etc/php-fpm.conf)
  116 ?      S      0:00  \_ php-fpm: pool symfony
  117 ?      S      0:00  \_ php-fpm: pool symfony
   10 ?      S      0:00 /bin/sh /usr/bin/mysqld_safe --port=3306
  158 ?      Sl     0:00  \_ /usr/libexec/mysqld --basedir=/usr --datadir=/var
```

Nous avons :

- un processus *supervisord* avec le PID 1, ce qui signifie qu'il officie comme processus racine (celui qui est géré par Docker) ;
- nous avons ensuite un processus Nginx avec deux processus fils (les deux workers de notre configuration) ;
- un processus PHP-FPM avec deux workers (valeur du paramètre pm.start_servers de *symfony.pool.conf*) ;
- un processus *mysqld_safe* (MariaDB) qui n'a qu'un unique worker.

Supervisor peut exposer un socket TCP sur laquelle un serveur HTTP écoute et permet de contrôler le processus supervisord depuis une interface graphique.
Pour l'activer, il faut rajouter une section au fichier de configuration supervisord.conf :

```
[inet_http_server]
port=9001
```

et ensuite exposer ce port via une commande `EXPOSE 9001` dans notre Dockerfile. Le serveur supervisord est ensuite disponible via http://localhost:9001 (pour peu que vous ayez ajouté `-p 9001:9001` à la commande `docker run` au lancement du conteneur).

Figure 9.3 — Interface de Supervisor

Ajout du code applicatif

Notre conteneur est prêt. Nous devons maintenant initialiser notre application. Pour ce faire, Symfony dispose d'un installateur (que nous avons copié dans notre conteneur). Lançons la commande suivante :

```
$ docker exec -i --user=apache lemp /bin/bash -c 'cd /var/www/; symfony new
my_app 2.8'
Downloading Symfony...

   0.00 B/5.17 MB ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ 0%
 15.40 KB/5.17 MB ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ 0%
 31.40 KB/5.17 MB ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ 0%
 47.40 KB/5.17 MB ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ 0%
 63.40 KB/5.17 MB ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ 1%
 79.40 KB/5.17 MB ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ 1%
 95.40 KB/5.17 MB ▓▓▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ 1%
111.40 KB/5.17 MB ▓▓▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ 2%
127.40 KB/5.17 MB ▓▓▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ 2%
```

Que fait-elle ?

Elle se connecte à notre conteneur et lance la commande *symfony* **à l'intérieur du conteneur**. Cela a pour conséquence qu'il n'est pas nécessaire d'avoir l'installeur Symfony ni même PHP sur notre machine hôte !

Elle va ensuite télécharger le code de Symfony et initialiser notre application dans le répertoire /var/www/. Ce répertoire étant monté sur notre hôte, notre code source est directement éditable.

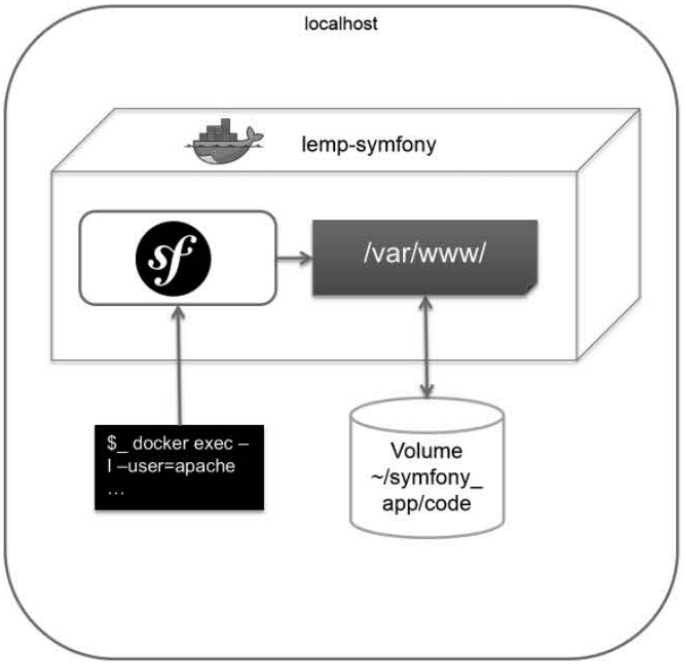

Figure 9.4 — docker exec symfony

Nous pouvons le vérifier aisément :

```
$ cd ~/symfony-app/code/my_app/
ls -l
total 96
drwxr-xr-x.  6 vagrant 48  4096 21 janv. 15:53 app
drwxr-xr-x.  2 vagrant 48    83 21 janv. 15:53 bin
-rw-r--r--.  1 vagrant 48  2299 21 janv. 15:53 composer.json
-rw-r--r--.  1 vagrant 48 75496 21 janv. 15:53 composer.lock
-rw-rw-rw-.  1 vagrant 48    71 21 janv. 15:53 README.md
drwxr-xr-x.  3 vagrant 48    38 21 janv. 15:53 src
drwxr-xr-x. 15 vagrant 48  4096 21 janv. 15:53 vendor
drwxr-xr-x.  3 vagrant 48  4096 14 janv. 13:25 web
```

Dans notre commande `docker exec`, nous précisons que nous voulons que la commande soit lancée en tant qu'utilisateur apache (`--user=apache`). Si nous ne le faisons pas, Docker utilisera l'utilisateur *root* pour créer les répertoires de notre application. Nous aurons alors un problème de permission sur notre volume (notre utilisateur *vagrant* ne pourra pas accéder aux fichiers et donc les modifier).

Vous pouvez maintenant voir votre application tourner depuis votre hôte sur l'URL http://localhost:8000 :

Figure 9.5 — Notre application totalement fonctionnelle

Pour finaliser la mise en place de notre environnement, créons le schéma dans notre base de données avec la commande suivante :

```
$ docker exec -i --user=apache lemp php /var/www/my_app/app/console
doctrine:database:create
Created database `symfony` for connection named default
```

Vous vous demandez sûrement comment notre application Symfony a pu créer un schéma sans que nous ne configurions les paramètres de connexion. Cela est tout simplement dû au fait que les paramètres par défaut de Symfony (my_app/app/config/parameters.yml) correspondent à ceux de notre configuration de MariaDB (user *root* sans mot de passe tournant sur le même hôte).

Ajout du code applicatif

Pour éviter d'avoir à taper chaque fois la longue commande `docker exec` nécessaire à l'exécution des instructions Symfony, il suffit de créer un alias dans le *~/.bashrc* de l'utilisateur courant (*vagrant* dans notre cas) avec la commande suivante :

```
$ echo -e "\nalias symfony='docker exec -i --user=apache lemp php
/var/www/my_app/app/console'" >> ~/.bashrc
```

Maintenant, quand vous voudrez utiliser la ligne de commande de Symfony, il suffira de taper, par exemple, pour vider le cache dans l'environnement de production :

```
$ symfony cache:clear --env=prod --no-debug
```

comme si Symfony était installé en local sur votre machine (n'oubliez pas de relancer votre terminal pour prendre en compte l'alias *symfony)*.

9.1.8 En conclusion

Nous avons donc notre environnement de développement prêt et fonctionnel sans avoir installé un seul moteur d'exécution sur notre machine. Si vous avez besoin de mettre à jour PHP ou de changer de base de données, une simple modification du Dockerfile et un build suffiront pour mettre à niveau votre environnement.

Cependant, cette approche a quelques inconvénients :

- nous ne pouvons pas profiter des images du Docker Hub (si ce n'est l'image de base de notre OS) ;
- notre configuration n'est pas en ligne avec une architecture micro-services comme nous l'avons expliqué au chapitre 1 : elle n'est pas facilement *scalable*, elle crée des dépendances fortes entre les processus et elle ne nous permet donc pas de changer facilement les caractéristiques de déploiement.

Voyons comment résoudre ces inconvénients avec une approche orientée micro-services.

9.2 APPLICATION MULTI-CONTENEURS

Nous allons maintenant rendre modulaire notre application.

Pour cela, nous utiliserons quatre conteneurs comme l'illustre la figure ci-dessous :

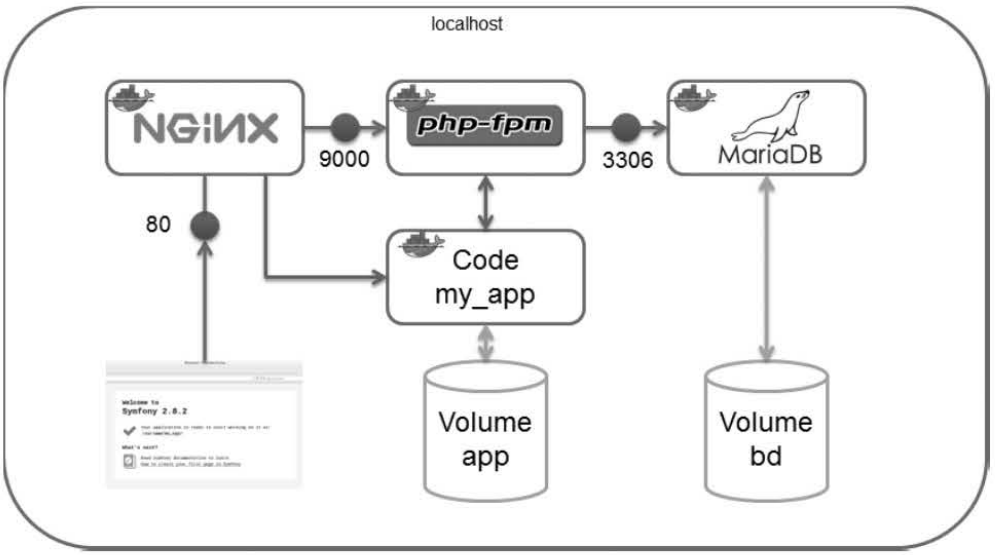

Figure 9.6 — Application multi-conteneurs

Chaque conteneur aura donc son propre Dockerfile (à l'exception de MariaDB car nous utiliserons l'image officielle disponible sur le Hub Docker).

Tous les fichiers que nous allons utiliser dans le cadre de ce paragraphe sont disponibles dans le répertoire micro-service. Par convention, nous avons utilisé un sous-répertoire par conteneur qui contient le Dockerfile et les fichiers nécessaires à la fabrication de l'image correspondante.

9.2.1 Nginx

Nous utiliserons maintenant l'image de base de Nginx que nous étendons en intégrant nos spécificités :

```
FROM nginx

COPY nginx.conf /etc/nginx/nginx.conf
COPY symfony.conf /etc/nginx/sites-available/
RUN mkdir -p /etc/nginx/sites-enabled/ && \
    mkdir -p /var/www/ && \
    ln -s /etc/nginx/sites-available/symfony.conf
/etc/nginx/sites-enabled/symfony
```

Comme vous pouvez le constater, notre Dockerfile est beaucoup plus simple que dans notre version initiale :

- il contient uniquement les informations nécessaires à Nginx ;
- nous n'avons plus besoin de nous occuper d'installation, d'exposition du port, etc. Tout est déjà inclus et optimisé dans l'image de base.

> Dès que vous le pouvez, utilisez les images du Hub Docker (https://hub.docker.com/), en particulier les images officielles. Elles vous permettent de bénéficier des meilleures pratiques et d'accélérer la mise en œuvre de solutions courantes.

9.2.2 PHP-FPM

Notre Dockerfile pour le conteneur PHP-FPM est relativement simple car il reprend les mêmes commandes que dans notre exemple précédent. La différence principale réside dans la ligne CMD qui lance PHP-FPM et non plus Supervisor.

```
FROM centos:7

MAINTAINER docker-ecosysteme@dunod.com

RUN yum install -y epel-release && \
    yum update -y && \
    yum install -y php-common php-cli php-fpm php-mysql php-apcu php-curl
php-intl php-mbstring && \
    yum clean all

COPY php-fpm/symfony.ini /etc/php.d/
COPY php-fpm/symfony.pool.conf /etc/php-fpm.d/
RUN rm -rf /etc/php-fpm.d/www.conf
    mkdir -p /var/www/ && \
    mkdir -p /var/lib/php/session && \
    chown -R apache:apache /var/lib/php

RUN curl -LsS https://symfony.com/installer -o /usr/local/bin/symfony && \
    chmod a+x /usr/local/bin/symfony

EXPOSE 9000

CMD ["/usr/sbin/php-fpm", "-c /etc/php-fpm.d"]
```

9.2.3 Notre code source

Pour suivre les bonnes pratiques de gestion des volumes, notre code sera disponible via un *data container*, c'est-à-dire un conteneur qui n'exposera qu'un volume. En effet, ce conteneur n'a pas de CMD ou d'ENTRYPOINT. Ce n'est pas un conteneur destiné à exécuter un processus, mais simplement le conteneur qui va gérer les données via le volume qu'il spécifie.

Nous le lierons ensuite aux conteneurs Nginx et PHP-FPM.

```
FROM busybox:glibc

VOLUME /var/www
```

Ce conteneur sera basé sur l'image busybox[1]. C'est une image de taille minimale, mais qui est largement suffisante pour ce conteneur.

Depuis la version 1.9 de Docker, il est possible d'utiliser des volumes nommés, c'est-à-dire de créer des volumes (avec la commande docker volume) puis de les associer à divers conteneurs. Cette alternative sera probablement à privilégier à l'avenir, mais de nombreuses architectures utilisent encore le principe des *data containers*. Il est donc utile de connaître ce pattern Docker.

9.2.4 Assemblons tout cela

Il ne nous reste plus maintenant qu'à fabriquer nos images :

```
$ docker build -t symfony-nginx nginx
...
$ docker build -t symfony-php php-fpm
...
$ docker build -t symfony-code symfony-code
...
```

Ces commandes doivent être lancées depuis le répertoire ~/application-multi-conteneurs/micro-services. Chacune indique en paramètre le répertoire workdir pour la construction de l'image correspondante.

Nos images étant prêtes, nous pouvons maintenant les utiliser pour nos conteneurs. Démarrons déjà notre *data container* qui va contenir le code source de notre application (et être par la suite lié aux conteneurs Nginx et PHP-FPM) :

```
$ docker create -v $HOME/symfony-app/code/:/var/www/ --name code symfony-code
```

Nous ne faisons que créer ce conteneur, sans le démarrer, avec la commande `docker create`. Nous montons ensuite le répertoire où se trouve le code source de notre application.

Démarrons ensuite Nginx et PHP-FPM en les liant à notre *data container* car ils vont tous les deux avoir besoin d'y accéder.

N'oubliez pas d'arrêter le conteneur lemp créé lors de l'exercice précédent avec `docker stop lemp`, sinon vous obtiendrez un message d'erreur indiquant que le port 8000 est déjà en cours d'utilisation.

1. https://hub.docker.com/_/busybox/

```
$ docker run -d -p 8000:80 --volumes-from code --name nginx symfony-nginx
ea53c81fd485357732b05c040034676928224 0d9940ec93f53c92af46648a158
$ docker run -d --volumes-from code --name php symfony-php
0bc7a35166afb91a01d5c0433d183e2bf51eb1a3a6f8b460a8cce0e4978c9f18
```

Il ne nous reste plus qu'à démarrer notre conteneur MariaDB. Cette fois-ci, nous utilisons le conteneur officiel MariaDB. Ce dernier nécessite une variable d'environnement contenant le mot de passe de l'utilisateur *root* pour démarrer.

```
$ docker run -d -e MYSQL_ROOT_PASSWORD=root --name mariadb mariadb:10.1
```

Ouvrons notre navigateur et saisissons l'adresse http://localhost:8000 pour nous apercevoir que nous obtenons un message d'erreur !

Si nous regardons les logs de notre conteneur Nginx, la raison est claire :

```
docker exec -i nginx cat /var/log/nginx/symfony_error.log
2016/01/28 21:28:25 [error] 6#6: *1 connect() failed (111: Connection refused)
while connecting to upstream, client: 172.17.0.1, server: symfony, request:
"GET / HTTP/1.1", upstream: "fastcgi://127.0.0.1:9000", host: "localhost:8000"
```

Nous avons maintenant plusieurs conteneurs et nos moteurs d'exécution ne peuvent donc plus communiquer librement comme s'ils étaient installés sur le même hôte. Chaque processus est isolé des autres.

Nous devons donc utiliser les fonctionnalités du réseau Docker.

9.3 LE RÉSEAU DOCKER

La communication entre conteneurs est un élément clé du succès de l'architecture micro-services et par conséquent de Docker. Elle repose sur la librairie *libnetwork*[1] qui est née de la volonté d'extraire le système de gestion du réseau du moteur. Elle implémente le modèle CNM (Container Network Model) que nous allons brièvement présenter pour bien comprendre ensuite les différents types de réseaux que fournit Docker.

Nous lèverons finalement le secret sur la gestion d'iptables par Docker et la gestion du DNS.

9.3.1 Libnetwork et le modèle CNM

Le modèle CNM est véritablement une abstraction pour la communication inter-conteneurs (potentiellement déployés sur des hôtes différents).

Elle repose sur trois composants principaux qui permettent de couvrir toutes les topologies de réseaux.

1. https://github.com/docker/libnetwork

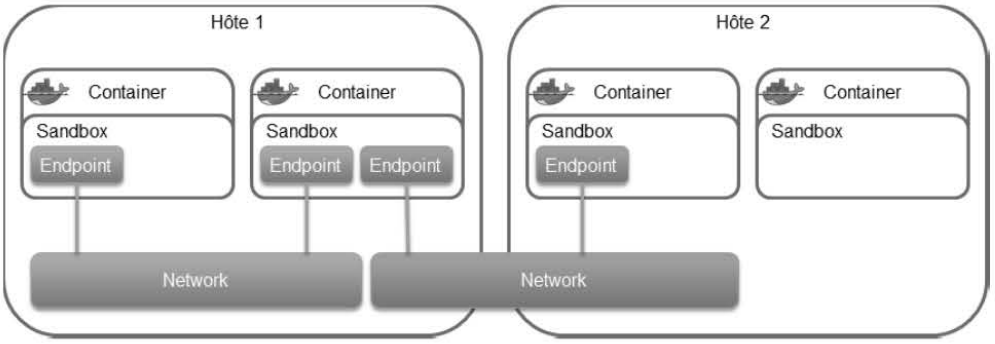

Figure 9.7 — Modèle CNM

Nous avons :

- le *sandbox* : il contient la configuration réseau du conteneur. Elle inclut les interfaces réseau (eth0...), les tables de routage et les configurations DNS. Dans le cas de Docker, il y en a une par conteneur et son implémentation est un *namespace*[1] network de notre machine hôte ;
- le *endpoint* : il permet de relier un *sandbox* à un network. Dans notre cas, ce sera typiquement une paire veth. Un *endpoint* n'est lié qu'à un *sandbox* et un *network*. Un conteneur aura donc autant de *endpoints* que de *network* auquel il est connecté ;
- le *network* : c'est un ensemble de *endpoints* qui peuvent communiquer ensemble. Il n'est au final qu'une abstraction sur une implémentation d'un pilote (*driver*) qui fournit les fonctionnalités de connectivité.

Libnetwork fournit plusieurs drivers que nous retrouvons plus loin dans ce chapitre car certains sont utilisés par Docker :

- *null* : c'est un driver un peu particulier qui signifie « pas de réseau ». Il est là par souci de rétrocompatibilité au niveau de Docker ;
- *overlay* : ce driver est pour l'instant le seul qui permette une communication entre plusieurs hôtes. Nous l'utiliserons dans le chapitre suivant qui est consacré à Docker Swarm ;
- *host* : permet de rendre disponible la configuration de la machine hôte à notre conteneur ; le conteneur peut donc directement accéder à toutes les ressources de l'hôte ;
- *remote* : ce n'est pas à proprement parler un driver, mais un proxy pour un driver distant. Il permet d'intégrer des drivers externes tels que Weave[2] ou Openstack Kuryr[3] ;

1. Pour plus d'information sur la notion de *namespace*, vous pouvez vous reporter au chapitre 1.
2. http://www.weave.works/products/weave-net/
3. https://github.com/openstack/kuryr

- *bridge* : ce driver se base sur un bridge Linux[1]. Il n'est disponible qu'à l'intérieur d'un même hôte.

Le driver bridge, qui est le driver historique de Docker, est encore celui qui est utilisé par défaut. Nous allons voir en détail comment il fonctionne, maintenant que nous avons une représentation claire du CNM.

Par volonté de simplicité, nous utiliserons de manière égale les termes réseau et driver.

Notez que, tout comme pour les volumes, Docker propose à des tiers de fabriquer des plugins pour des systèmes externes de gestion de réseau. Nous verrons donc prochainement des spécialistes des équipements de routage et de sécurité proposer des plugins pour gérer les réseaux de conteneurs.

9.3.2 L'interface docker0 et le réseau bridge

Pour lister les réseaux disponibles pour nos conteneurs au niveau de notre démon Docker sur notre hôte, nous utilisons la commande suivante :

```
$ docker network ls
NETWORK ID          NAME                DRIVER
b2d747bcd553        host                host
8998ee49ca35        bridge              bridge
758827c4904b        none                null
```

En fait, Docker est installé par défaut avec trois sous-réseaux :

- *none* : ce réseau utilise le driver *null* de *libnetwork*, et il n'a donc pas d'interface réseau. Tout conteneur en faisant partie n'aura donc pas d'accès réseau et ne pourra pas être connecté à un autre conteneur. Les cas d'utilisations sont rares (définition du sous-réseau avec des outils tiers comme pipework[2] ou un conteneur temporaire qui fait une action et s'arrête) et nous ne les couvrirons pas dans ce livre ;
- *host* : ce sous-réseau permet de rendre disponible la configuration de notre machine hôte à notre conteneur. La commande `ifconfig` dans un conteneur et sur notre hôte donnerait les mêmes résultats. Ce type de réseau simplifie la communication du conteneur avec l'hôte, mais au détriment de la lisibilité des liens réseau. On l'utilisera essentiellement pour la mise au point des conteneurs ou lorsqu'on se trouve dans une architecture mixte conteneur et hôte ;
- *bridge* : c'est le sous-réseau historique de Docker qui permet de connecter plusieurs conteneurs.

Regardons-le plus en détail avec la commande suivante :

1. http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge
2. https://github.com/jpetazzo/pipework

```
$ docker network inspect bridge
[
    {
      "Name": "bridge",
      "Id":
"9df329fa52975a8af8cf33042a48561087c7e7d60a2b694eaf7a7b4b6d969d4e",
      "Scope": "local",
      "Driver": "bridge",
      "EnableIPv6": false,
      "IPAM": {
          "Driver": "default",
          "Options": null,
          "Config": [
              {
                  "Subnet": "172.17.0.0/16"
              }
          ]
      }, ...
"Options": {
      "com.docker.network.bridge.default_bridge": "true",
      "com.docker.network.bridge.enable_icc": "true",
      "com.docker.network.bridge.enable_ip_masquerade": "true",
      "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
      "com.docker.network.bridge.name": "docker0",
      "com.docker.network.driver.mtu": "1500"
        }
    }
]
```

Nous voyons que ce réseau est bien de type *bridge* et qu'il dispose des adresses IP 172.17.0.1/16. Lors de l'installation de Docker, ce dernier a créé un pont virtuel Ethernet, nommé *docker0* sur notre machine hôte. La commande `ifconfig` nous permet de voir les détails de cette interface :

```
$ ifconfig docker0
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:4ff:fe06:b1dd prefixlen 64 scopeid 0x20<link>
        ether 02:42:04:06:b1:dd  txqueuelen 0  (Ethernet)
        RX packets 2489  bytes 116961 (114.2 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2549  bytes 29591068 (28.2 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Cette interface est elle-même connectée ensuite à l'interface réseau de l'hôte et permet aux conteneurs de communiquer entre eux ainsi qu'avec l'extérieur.

Au démarrage d'un conteneur, Docker crée une interface virtuelle sur l'hôte avec un nom du type veth4bdf012 et assigne une adresse IP libre. Cette interface est ensuite connectée à l'interface eth0 de notre conteneur.

Dans le cas de notre application, nous aurons une configuration similaire à la figure ci-dessous.

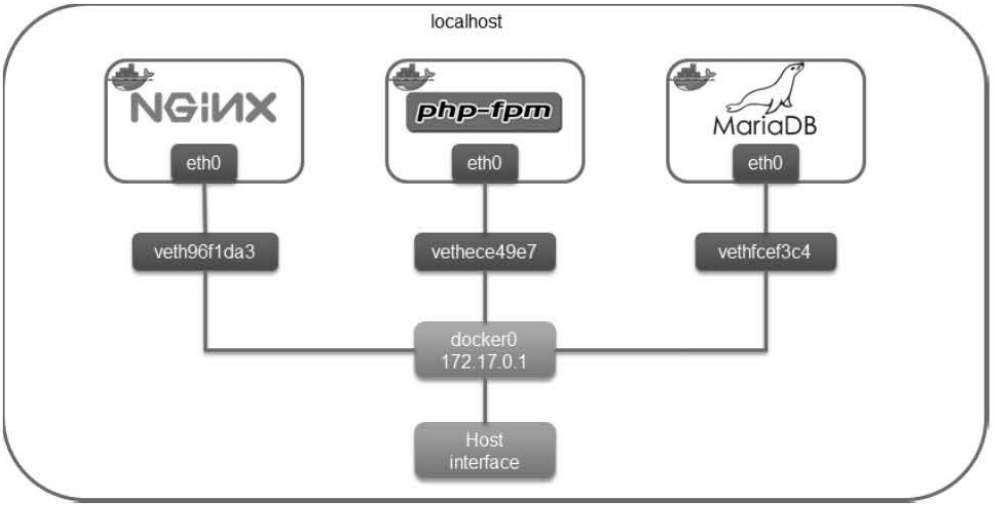

Figure 9.8 — Réseau bridge

Vous pouvez lister toutes les interfaces virtuelles connectées sur notre interface docker0 avec la commande suivante :

```
$ sudo brctl show docker0
bridge name     bridge id               STP enabled     interfaces
docker0         8000.02422b1d4dda       no              veth96f1da3
                                                        vethece49e7
                                                        vethfcef3c4
```

Pour connaître la configuration réseau d'un de notre conteneur php, il suffit d'utiliser la commande suivante :

```
$ docker inspect  php | jq .[0].NetworkSettings
{
  "Bridge": "",
  "SandboxID":
"820b96e0b4481dfefdd4d8d2fc0910cd45dca5969b668d896df2541baa33d5a2",
  "HairpinMode": false,
  "LinkLocalIPv6Address": "",
  "LinkLocalIPv6PrefixLen": 0,
  "Ports": {
    "9000/tcp": null
  },
  "SandboxKey": "/var/run/docker/netns/820b96e0b448",
  "SecondaryIPAddresses": null,
  "SecondaryIPv6Addresses": null,
  "EndpointID":
"91adcfec7ea5cac6cfde91c7b2971b6feddc2c35e669b2edf90d49ee733f8aad",
  "Gateway": "172.17.0.1",
  "GlobalIPv6Address": "",
  "GlobalIPv6PrefixLen": 0,
```

```
    "IPAddress": "172.17.0.3",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "MacAddress": "02:42:ac:11:00:03",
    "Networks": {
      "bridge": {
        "IPAMConfig": null,
        "Links": null,
        "Aliases": null,
        "NetworkID":
"9df329fa52975a8af8cf33042a48561087c7e7d60a2b694eaf7a7b4b6d969d4e",
        "EndpointID":
"91adcfec7ea5cac6cfde91c7b2971b6feddc2c35e669b2edf90d49ee733f8aad",
        "Gateway": "172.17.0.1",
        "IPAddress": "172.17.0.3",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:11:00:03"
      }
    }
}
```

Nous retrouvons bien l'adresse de la *gateway* (172.17.0.1) assignée automatiquement par Docker et l'adresse IP de notre conteneur (172.17.0.3) est aussi présente dans le sous-réseau de notre réseau bridge.

Le NetworkId, SandboxID et le EndpointID correspondent à ceux présentés dans le modèle CNM au chapitre précédent.

Nos conteneurs sont donc tous sur le même réseau. Cela ne signifie pourtant pas qu'ils se connaissent, qu'ils puissent être joints depuis l'extérieur du réseau ou qu'ils puissent accéder à Internet.

9.3.3 Communication entre conteneurs

La communication de nos conteneurs entre eux et avec le monde extérieur est contrôlée à deux niveaux :

- En premier lieu, au niveau de la machine hôte : permet-elle le transfert des paquets IP ?
- Ensuite par la configuration d'iptables qui gère le dialogue avec le monde extérieur.

Transfert des paquets IP

Le fait que les paquets IP soient transmis (*forwarded*) par la machine hôte est gouverné par la variable système (kernel) *net.ipv4.conf.all.forwarding*. Sa valeur courante est simple à obtenir avec la commande suivante :

```
$ sysctl net.ipv4.conf.all.forwarding
net.ipv4.conf.all.forwarding = 1
#Pour changer sa valeur si cette dernière est égale à 0 c.à.d. non activée
$ sudo sysctl net.ipv4.conf.all.forwarding=1
```

Si le *forwarding* n'est pas activé, les conteneurs ne peuvent ni communiquer entre eux, ni atteindre un réseau externe comme Internet.

Il est possible de surcharger la valeur système définie au démarrage du démon Docker par le paramètre `--ip-forward` (par défaut à true).

Attention, dans le cas improbable où vous voudriez le désactiver, le paramètre `--ip-forward=false` n'aurait aucun effet si la valeur de *net.ipv4.conf.all.forwarding* est à 1, c'est-à-dire activée au niveau système.

Docker et Iptables

Iptables est un pare-feu logiciel présent par défaut dans la grande majorité des distributions Linux. Il serait plus juste de parler d'iptables/netfilter car *iptables* n'est finalement qu'un moyen de configuration de *netfilter*[1] qui réalise effectivement le filtre des paquets réseau au niveau du kernel Linux.

Le réseau Docker bridge (ainsi que tous les autres types de réseaux Docker) se base sur iptables pour autoriser/refuser les connexions entre conteneurs.

> Si vous désirez ne pas laisser Docker modifier votre configuration iptables ou si vous disposez, par exemple, d'autres outils pour le faire, vous pouvez l'empêcher en passant le paramètre `--iptables=false` au démarrage du démon Docker.

Regardons maintenant comment Docker utilise iptables pour certains cas courants.

Accès à d'autres réseaux

Pour sortir d'un sous-réseau géré par Docker, il est nécessaire que notre gateway implémente du masquage d'adresses IP (nos IP privées n'étant pas routables sur Internet). Par défaut, le démon Docker démarre avec le paramètre `--ip-masq=true` qui a pour conséquence de rajouter des règles dans notre configuration iptables. Nous pouvons le voir facilement avec la commande suivante :

```
$ sudo iptables -t nat -L -n
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16        0.0.0.0/0
...
```

1. http://www.netfilter.org/

Dans ce cas, notre réseau bridge (172.17.0.0/16) pourra accéder à toute IP en dehors de ce réseau. Il est bien sûr possible de bloquer l'accès des conteneurs à l'extérieur de ce réseau en passant le paramètre `--ip-masq=false` au démarrage du démon Docker.

Communications inter-conteneurs

Nous devons ici bien séparer deux types de réseau bridge Docker :

- nous avons le réseau prédéfini qui se nomme « bridge » et qui utilise le driver *libnetwork* de type bridge. C'est le réseau historique de Docker ;
- nous pouvons aussi créer un nouveau réseau privé de type bridge. Celui-ci résout les limitations du réseau historique.

Quelles sont donc les différences ?

Tableau 9.1 — Différences entre les deux réseaux bridge

Réseau bridge via docker0	Réseau de type bridge
Créé par défaut au démarrage du démon Docker.	Doit être créé manuellement avec la commande `docker network create`.
Utilisation du pont internet prédéfini docker0.	Création d'un nouveau pont ethernet propre au réseau br-xxx.
Tout nouveau conteneur est automatiquement connecté à ce réseau si aucun réseau n'est spécifié.	Connexion manuelle du conteneur sur ce réseau avec la commande `docker network connect` ou en passant le paramètre `--net=` à la commande `docker run`.
Si `--icc=false`, nécessite la déclaration explicite via des links pour connecter les conteneurs entre eux via le paramètre `--link` de la commande `docker run`.	Le paramètre `--icc` est ignoré et tous les conteneurs du même réseau peuvent communiquer ensemble par défaut.
Si un conteneur lié via un link est supprimé, il faut recréer tous les conteneurs qui avaient un lien vers ce conteneur. En effet, sans DNS, l'adresse IP à laquelle est lié ce conteneur peut changer (et aura probablement changée) au démarrage du nouveau conteneur. C'est la plus grosse limitation des links Docker.	Si un conteneur est supprimé et recréé avec le même nom, les conteneurs qui en ont besoin le trouveront toujours.
Si un lien est créé entre deux conteneurs, Docker crée automatiquement des variables d'environnement sur le conteneur cible[a] concernant son nom.	Plus nécessaire.
Ne peut être supprimé.	Peut être supprimé avec la commande `docker network rm`.

a. https://docs.docker.com/engine/userguide/networking/default_network/dockerlinks/

Vous l'aurez compris, le nouveau système de réseau est plus flexible et plus configurable que le réseau bridge historique avec son système de liens. Docker conseille d'ailleurs fortement de passer sur le nouveau modèle de réseau car les liens seront dépréciés dans une future version de Docker.

Nous allons donc regarder la communication inter-conteneurs avec un nouveau réseau. Tout d'abord, déconnectons nos conteneurs du réseau bridge par défaut.

```
$ docker network disconnect bridge nginx
$ docker network disconnect bridge php
$ docker network disconnect bridge mariadb
```

Créons un nouveau réseau my_net et connectons-y nos conteneurs :

```
$ docker network create -d bridge my_net
a34441233b8ded3e9fdd56a9e33e51c3115e56e623832a9689c2f241d6f7a8b0
$ docker network connect my_net nginx
$ docker network connect my_net php
$ docker network connect my_net mariadb
```

Nous précisons que nous voulons un type de réseau bridge avec l'option `-d` (optionnelle car c'est la valeur par défaut). Docker crée automatiquement un pont Ethernet et un sous-réseau.

NB : La commande `docker network create` accepte des paramètres qui permettent de définir l'IP de la gateway (`--gateway`), le sous-réseau (`--subnet` au format CIDR) et la plage d'IP (`--ip-range`).

Nos conteneurs sont maintenant sur le même réseau et peuvent communiquer entre eux soit par IP, soit en utilisant le nom du conteneur comme nom d'hôte. Les ports exposés par le conteneur sont eux aussi disponibles. Pour nous en convaincre, connectons-nous dans le conteneur php (nous installerons le logiciel *nmap* pour obtenir un peu plus d'information) :

```
$ docker exec -it php /bin/bash
[root@1844d7f5d5e6 /]# yum install -y nmap
[root@1844d7f5d5e6 /]# nmap -T4 mariadb

Starting Nmap 6.40 ( http://nmap.org ) at 2016-01-27 16:26 UTC
Nmap scan report for mariadb (172.18.0.4)
Host is up (0.000014s latency).
Not shown: 999 closed ports
PORT     STATE SERVICE
3306/tcp open  mysql
MAC Address: 02:42:AC:12:00:04 (Unknown)

Nmap done: 1 IP address (1 host up) scanned in 0.35 s
[root@1844d7f5d5e6 /]# nmap -T4 nginx

Starting Nmap 6.40 ( http://nmap.org ) at 2016-01-27 16:27 UTC
Nmap scan report for nginx (172.18.0.3)
Host is up (0.000014s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
80/tcp open  http
MAC Address: 02:42:AC:12:00:03 (Unknown)

Nmap done: 1 IP address (1 host up) scanned in 0.14 seconds
```

Nous voyons que notre conteneur nginx a l'IP 172.18.0.3 et expose son port 80, mariadb ayant l'IP 172.18.0.4 et exposant le port 3306.

Reste un mystère pour que tout soit totalement transparent : comment notre conteneur réussit-il à résoudre les noms mariadb ou nginx ?

Depuis la version 1.10 de Docker, un serveur DNS a été intégré au démon. La résolution des adresses IP peut donc aussi s'appuyer sur ce serveur DNS, qui en principe est automatiquement ajouté à la liste des serveurs DNS du conteneur. Néanmoins, Docker, dans sa documentation la plus récente indique que le mode de résolution de nom de domaine peut varier en fonction des implémentations de Docker. À ce titre, Docker Inc. recommande de ne faire aucune hypothèse sur le conteneur des fichiers /etc/hosts ou /etc/resolv.conf. Regardons cela après avoir installé les utilitaires *bind-utils* :

```
root@bca409b26400:/#  yum install -y bind-utils
root@bca409b26400:/#  dig nginx
; <<>> DiG 9.9.4-RedHat-9.9.4-29.el7_2.3 <<>> nginx
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 22562
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;nginx.INA

;; ANSWER SECTION:
nginx.600INA172.19.0.4

;; Query time: 0 msec
;; SERVER: 127.0.0.11#53(127.0.0.11)
;; WHEN: Sun Jun 26 09:09:09 UTC 2016
;; MSG SIZE  rcvd: 44
```

Nous voyons que le serveur DNS a l'adresse 127.0.0.11, ce que nous confirme le fichier /etc/resolv.conf :

```
root@bca409b26400:/#  cat /etc/resolv.conf
search home
nameserver 127.0.0.11
options ndots:0
```

Utilisons ce que nous venons d'apprendre pour faire fonctionner notre application Symfony.

9.3.4 Mise en œuvre sur notre application Symfony

Ayant déjà créé notre réseau au chapitre précédent, il ne nous reste plus qu'à adapter nos fichiers de configuration pour qu'ils l'utilisent.

Nginx

Pour notre serveur Nginx, il faut que nous configurions la localisation de PHP-FPM : au lieu de nous connecter à *localhost*, nous devons utiliser le conteneur php.

Éditons notre fichier *symfony.conf* et remplaçons simplement l'IP 127.0.0.1 par php (le nom de notre conteneur) :

```
$ vi $HOME/application-multi-conteneurs/micro-services/nginx/symfony.conf
# DEV
      location ~ ^/(app_dev|config)\.php(/|$) {
          fastcgi_pass php:9000;
          fastcgi_split_path_info ^(.+\.php)(/.*)$;
          include fastcgi_params;
          fastcgi_param  SCRIPT_FILENAME  $realpath_root$fastcgi_script_name;
          fastcgi_param DOCUMENT_ROOT $realpath_root;
      }

  # PROD
      location ~ ^/app\.php(/|$) {
   fastcgi_pass php:9000;
```

Il suffit ensuite de relancer la création de notre image, de supprimer l'ancien conteneur nginx et de démarrer un nouveau conteneur en le connectant bien à notre nouveau réseau my_net :

```
$ docker build -t symfony-nginx nginx
$ docker rm -f nginx
$ docker run -d -p 8000:80 --net my_net --volumes-from code --name nginx
symfony-nginx
```

PHP-FPM

Ici, nous n'avons que le fichier de connexion de base de données à modifier :

```
$ vi $HOME/symfony-app/code/my_app/app/config/parameters.yml

# This file is auto-generated during the composer install
parameters:
    database_host: mariadb
    database_port: 3306
    database_name: symfony
    database_user: root
    database_password: root
    mailer_transport: smtp
    ...
```

Redémarrons notre conteneur avec `docker restart php` pour que notre configuration soit prise en compte ; nous avons maintenant une application utilisant tous les bénéfices du réseau Docker :

- nous pouvons remplacer n'importe lequel de nos conteneurs, et il ne sera pas nécessaire de reconfigurer les autres à partir du moment, où, bien sûr, son nom ne change pas ;
- nous n'avons plus à lier nos conteneurs manuellement lorsque nous les démarrons.

Nous avons beaucoup progressé depuis notre première tentative avec un seul conteneur. Nous pouvons cependant faire encore mieux. En effet, pour construire et démarrer tous nos conteneurs, il faut un nombre important de commandes Docker et nous devons nous rappeler à chaque fois tous les paramètres à utiliser (port à exposer, volumes, réseaux...).

Heureusement, Docker propose une solution pour documenter tout cela, Docker Compose.

9.4 ORCHESTRATION AVEC DOCKER COMPOSE

Docker Compose vous permet de définir dans un fichier de configuration toutes les dépendances de votre application multi-conteneurs. Ensuite, avec une seule commande, il vous permet de démarrer tous les conteneurs de votre application.

Docker Compose est le module d'orchestration de la suite Docker. Nous l'avons déjà évoqué dans notre chapitre 2. À noter que d'autres solutions CaaS, comme Kubernetes, disposent de leur propre module d'orchestration. Docker Compose reste néanmoins très utilisé par la communauté en raison de sa simplicité.

Regardons comment mettre cela en œuvre pour notre application Symfony.

9.4.1 Introduction et premiers pas avec Docker Compose

Compose nécessite bien sûr que Docker soit installé sur votre hôte. Il peut être utilisé sous Mac OS X (inclus par défaut dans Docker ToolBox) et sous Linux, mais n'est pas encore pris en charge sous Windows.

L'installation se fait en récupérant l'exécutable depuis le dépôt GitHub avec *curl* :

```
$ sudo su
$ curl -L https://github.com/docker/compose/releases/download/1.7.1/docker-
compose-‘uname -s‘-‘uname -m‘ >
/usr/local/bin/docker-compose
$ chmod +x /usr/local/bin/docker-compose
$ /usr/local/bin/docker-compose --version
docker-compose version 1.7.1, build 0a9ab35
```

Docker Compose recherche par défaut un fichier de configuration dans le répertoire courant, nommé docker-compose.yml, au format YAML[1]. Ce fichier est le descripteur de notre application, il permet de définir :

- les conteneurs avec leurs relations, les volumes qu'ils utilisent, les ports qu'ils exposent ;
- les réseaux nécessaires à votre application (même si Docker Compose en crée un par défaut si vous n'en spécifiez aucun) ;
- les volumes éventuellement partagés entre les différents conteneurs.

Il est tout à fait possible de fournir à Docker Compose un fichier avec un autre nom à l'aide du paramètre `-f`. Il peut même accepter plusieurs fichiers de configuration : dans ce cas, il fusionnera leur contenu.

Docker Compose utilise aussi la notion de projet. Il faut voir un projet comme une instance de votre application. Par défaut, le nom du projet sera celui du répertoire courant. On peut bien sûr en passer un autre avec le paramètre `-p`.

Mettons tout cela en œuvre.

9.4.2 Notre application

Nous allons utiliser la version 2 du format du fichier Docker Compose. En effet, Le format a changé avec la version 1.6 et nécessite au moins le Docker Engine 1.10. Les raisons principales de ce changement sont liées au support des volumes et des réseaux.

Vous trouvez l'intégralité du fichier docker-compose.yml de notre application dans le répertoire ~/application-multi-conteneurs/docker-compose.

```
version: '2'

services:
  code:
    build:
      context: symfony-code
    container_name: code
    volumes:
      - $HOME/symfony-app/code/:/var/www/

  nginx:
    build:
      context: nginx
    container_name: nginx
    depends_on:
      - php
```

1. http://www.yaml.org/

```
    ports:
      - 8000:80
    volumes_from:
      - code

  php:
    build:
      context: php-fpm
    container_name: php
    volumes_from:
      - code

  db:
    image: mariadb:10.1
    container_name: db
    environment:
      MYSQL_ROOT_PASSWORD: root
```

Nous retrouvons :

- le fait que nous utilisons la version 2 du format Compose ;
- un bloc `services:` qui contient la définition de nos quatre conteneurs ;
- pour chaque conteneur, nous avons un paramètre `build` qui permet de spécifier le chemin du répertoire à utiliser comme contexte de création. Docker Compose fabriquera une nouvelle version de notre image si nécessaire ;
- nous spécifions aussi le nom de nos conteneurs avec le paramètre `container_name`. Si nous ne le spécifions pas, Docker Compose générera automatiquement un nom à partir du nom de projet et du nom du bloc (par exemple, pour notre conteneur nginx, ce sera dockercompose_nginx_1) ;
- le paramètre `depends_on` permet de nous assurer que le conteneur php sera disponible lors du démarrage de nginx. Si nous n'avons pas ce paramètre, le conteneur nginx s'arrête s'il démarre avant le conteneur php ;
- le reste des paramètres est relativement simple à comprendre et correspond dans leur syntaxe aux paramètres de la ligne de commande Docker.

Démarrons notre service (il faut lancer la commande depuis le répertoire qui contient notre fichier docker-compose.yml) :

```
$ /usr/local/bin/docker-compose up -d
```

Cette commande va :

- créer les images de nos services à partir de nos différents Dockerfile ;
- pour chaque image, démarrer un conteneur basé sur la configuration (ports, volumes...) de notre fichier docker-compose.yml. Le paramètre `-d` est identique à celui de Docker : nos conteneurs seront démarrés en mode démon ;
- créer un réseau (de type bridge) avec le nom de notre projet (ce que nous pouvons vérifier, après lancement, grâce à la commande `docker network ls`).

Un des avantages majeurs de Docker Compose est qu'il est possible d'utiliser la commande `docker-compose up` de manière répétée. Si la configuration de notre application (c'est-à-dire notre fichier docker-compose.yml) ou une de nos images a changé, Docker Compose va arrêter et redémarrer de nouveaux conteneurs. Et si nos anciens conteneurs avaient des volumes, ces derniers seront attachés aux nouveaux, garantissant que notre application soit toujours fonctionnelle sans perte de données.

Vous pouvez passer à la commande `docker-compose up` les paramètres `--no-recreate` ou (à l'opposé) `--force-recreate` pour respectivement ne pas recréer/forcer la recréation systématique de tous les conteneurs.

Docker Compose fournit un ensemble de commandes similaires aux commandes du Docker Engine, ce qui simplifie d'autant plus son utilisation. Ces commandes affectent tous les conteneurs de notre projet.

Tableau 9.2 — Commandes Docker Compose

Commande	Résultat
docker-compose ps	Liste les conteneurs de notre application.
docker-compose logs	Affiche une vue agrégée de tous les logs de notre application.
docker-compose start / docker-compose stop	Démarre/arrête l'ensemble des conteneurs de notre application.
docker-compose pause / docker-compose unpause	Met en pause/relance les processus qui tournent dans les conteneurs de notre application.
docker-compose rm	Supprime tous les conteneurs de notre application. Le paramètre **-v** permet de forcer la suppression même si les conteneurs sont démarrés.

En résumé

Ce chapitre nous a permis de suivre pas à pas la création d'une application multi-processus. Nous avons vu comment combiner plusieurs processus au sein d'un même conteneur avec Supervisor. Même si la chose n'est pas recommandée, elle est toujours couramment pratiquée. Nous avons ensuite vu comment implémenter une application à base de micro-services et, notamment, comment en configurer le réseau. Enfin, nous nous sommes penchés sur Docker Compose, l'outil d'orchestration de la suite Docker pour centraliser et automatiser la création de nos services.

10

Intégration continue avec Docker

Dans ce chapitre, nous allons mettre en place un système d'intégration continue (dont l'acronyme anglais est CI, pour *Continuous Integration*) reposant sur des conteneurs Docker. D'une part, nous utiliserons ces derniers pour déployer les outils nécessaires à notre CI entièrement « dockerisée », puis nous construirons et packagerons une application dans un conteneur afin de la déployer ensuite dans plusieurs environnements (développement, test et production dans notre cas).

10.1 AVANT DE COMMENCER

10.1.1 Quelques mots sur l'intégration continue

Un système d'intégration continue permet de supporter le cycle de vie complet d'une application, de sa conception à son déploiement. Il a pour objectif de mettre à disposition dans un environnement de CI une version fonctionnelle de l'application à chaque instant (c'est-à-dire généralement après chaque modification du code source ou de la configuration de l'application).

Pour aller encore plus loin, certaines entreprises mettent aujourd'hui en œuvre des approches de type déploiement continue (*continuous delivery* ou CD) où des versions de l'application sont déployées en production plusieurs fois par semaine, voire par jour.

Dans ce chapitre, nous allons mettre en œuvre une version simplifiée, mais fonctionnelle de cette approche qui nous permettra de comprendre les apports des conteneurs à ces pratiques de plus en plus courantes.

Pour cela, nous aurons besoin des outils suivants :

- un outil d'**intégration continue** permettant la construction applicative (par exemple, la compilation d'une application Java) afin de produire des artefacts logiciels (dans notre cas, une image Docker). Nous l'utiliserons aussi pour les déploiements dans nos divers environnements. Pour notre exemple, nous utiliserons Jenkins[1] ;
- un outil de **gestion de code source** permettant de versionner le code d'une application. Pour notre exemple, nous utiliserons GiLab[2] qui repose sur Git et offre une interface graphique proche de celle de GitHub ;
- un outil de **suivi des exigences** (*issues tracking* en anglais) permettant de créer des tickets décrivant les besoins et anomalies d'une application. Nous ne présenterons pas cet aspect dans notre exemple, toutefois GitLab ferait très bien l'affaire ;
- un outil de **dépôt des artefacts logiciels** permettant de centraliser les composants issus de la phase de construction applicative (compilation et conditionnement). Comme pour l'outil de suivi des exigences, nous n'aborderons pas les détails de ce composant. Cependant, un registry privé Docker, ou encore le Docker Hub lui-même, pourrait convenir à cet usage.

Regardons maintenant concrètement les conteneurs outils nécessaires à notre environnement d'intégration continue.

10.1.2 Les conteneurs outils

Rappelons tout d'abord les objectifs de notre système d'intégration continue :

- construire automatiquement une image Docker générique après toute modification (*commit*) du code. Par générique, nous entendons déployable dans tout environnement ;
- mettre en place trois environnements : le premier pour le développement, le deuxième, qui sera déployé automatiquement, pour le test, et pour finir, le troisième pour la production. Ce dernier permettra de déployer des versions *release* du code (c'est-à-dire considérées comme testées et stables) ;
- gérer un cycle de versions, c'est-à-dire être capable de taguer un code afin de le mettre en production.

Le schéma ci-contre illustre l'architecture globale qui sera mise en place, aussi bien nos conteneurs outils que ceux embarquant l'application :

1. https://jenkins.io/
2. https://about.gitlab.com/

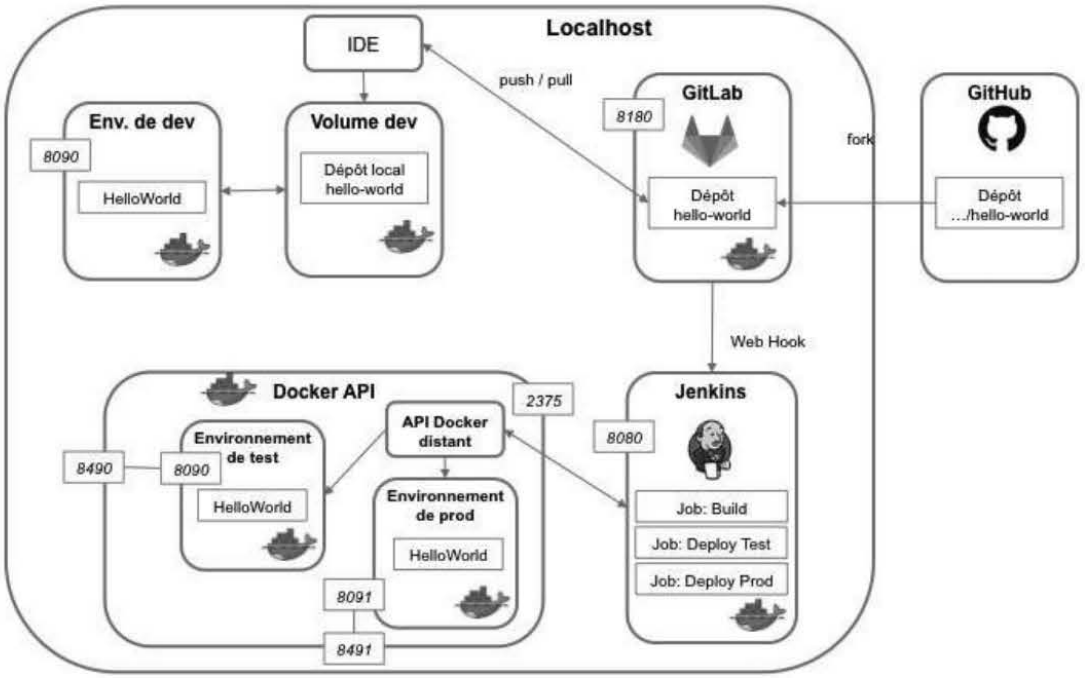

Figure 10.1 — Notre architecture cible

GitLab

GitLab est un conteneur qui intègre notamment l'outil de gestion de version Git. Nous commencerons par *forker* le dépôt hello-world présent dans le dépôt GitHub de cet ouvrage. Ensuite, GitLab sera utilisé comme source pour les builds Jenkins et permettra de versionner l'application en définissant des *tags* correspondant aux versions stables de l'application.

> Le *fork* de l'application hello-world depuis GitHub est réalisé afin d'obtenir directement une application fonctionnelle et correctement configurée. Cette opération n'est faite qu'une seule fois et aucun *push* vers GitHub ne sera effectué.

Le code de l'application sera toujours « pushé » dans la branche *master*, ainsi cette dernière contiendra en tout temps la dernière version de l'application. Comme cela est mentionné plus haut, nous utiliserons les *tags* pour marquer l'application à livrer en production.

> **Le Git flow**
>
> L'utilisation de Git se fait généralement en respectant certaines conventions, notamment au niveau des branches. Une convention communément utilisée est le Git flow.
>
> Le Git flow est représenté par deux branches principales, *master* et *develop*, ainsi que par des branches secondaires pour la réalisation des fonctionnalités et la correction des anomalies (*feature branches*).

La branche master représente la *code base* qui est actuellement en production, aucun push n'y est directement effectué. La branche develop représente la *code base* incluant les dernières fonctionnalités considérées comme développées. Lors d'une release, la branche develop est ainsi fusionnée dans la branche master (afin que cette dernière contienne la *code base* de production).

Pour chaque nouvelle fonctionnalité ou anomalie, un développeur crée une branche temporaire dans laquelle il « pushera » ses modifications. Une fois le développement terminé, il demandera la fusion de sa branche dans la branche develop (et supprimera généralement sa branche). C'est ce qu'on nomme une *pull request*.

Par souci de simplification, nous ne respectons pas dans notre exemple le Git flow, mais nous ne pouvons que vous encourager à le faire dans le cadre de vos projets.

Nous configurerons un *webhook* dans GitLab afin de déclencher automatiquement un *build* dans Jenkins après chaque *push*, c'est-à-dire après toute modification de l'application. Ce *webhook* utilisera la branche *master*.

Jenkins

Jenkins est utilisé pour construire (*build* en anglais) l'application sous la forme d'une image Docker (en s'appuyant sur Gradle[1], comme nous le verrons un peu plus loin), puis pour la déployer dans nos deux environnements, un de test et un autre de production. Pour cela, nous aurons besoin de trois *jobs* :

- Build : ce job clonera la *code base* de l'application selon la version souhaitée (c'est-à-dire soit la branche *master*, soit un *tag* représentant une version) et construira une image Docker contenant notre application. Il déléguera cette tâche au conteneur Docker API ;
- Deploy test : ce job invoquera aussi l'API Docker distant (du conteneur Docker API) afin d'instancier un conteneur basé sur l'image créée par le job Build. Il utilisera toujours l'image basée sur la branche *master* et sera déclenché automatiquement après tout *build* ;
- Deploy prod : ce job est assez similaire au précédent, à la différence qu'il faudra spécifier la version de l'image à utiliser (un *tag* dans notre cas).

Docker API

Docker API est un conteneur qui contiendra une API Docker distant (autrement dit un « Docker dans Docker »). Il construira les images à la demande du job Jenkins Build, et démarrera des conteneurs à la demande des jobs Deploy test et Deploy prod.

Ces derniers seront ensuite embarqués directement dans le conteneur Docker API.

1. http://gradle.org/

Environnement de développement

L'environnement de développement est un conteneur, associé à un volume dev qui contient une copie (ou plutôt un clone) du code, permettant de tester rapidement un nouveau développement conçu localement. Le développement, à proprement parler, sera quant à lui directement effectué depuis la machine hôte.

10.1.3 Notre application HelloWorld

Notre application sera un simple « Hello World » développé grâce au framework SpringMVC[1] (Java) et qui utilisera le serveur d'application SpringBoot[2]. En termes du code de l'application, il n'y a pas grand-chose à dire : il s'agit d'un contrôleur (au sens MVC du terme pour *Model View Controller*) définissant une unique action et produisant une vue qui affiche un « Hello, {name} », {name} étant un paramètre que l'on peut donner à la route de notre action.

> Le code de l'application HelloWorld est fourni grâce à un *fork* dans GitHub que nous ferons un peu plus tard

L'application n'aura aucun état si bien qu'aucune base de données ni volume ne seront nécessaires.

Nous utiliserons Gradle pour « builder » notre application. Il faut voir ce dernier comme un interpréteur de fichiers décrivant comment construire notre application (au même titre que make ou Maven). Le fichier principal se nomme par convention *build.gradle* et indiquera :

- que notre application repose sur le framework SpringMVC ;
- qu'elle doit être compilée en Java ;
- que le résultat de la compilation doit être embarqué dans un serveur d'application SpringBoot ;
- qu'une image Docker doit être construite afin de pouvoir démarrer notre serveur d'application qui, lui-même, exécute l'application Java.

Ainsi, en interprétant ce fameux fichier Gradle, nous aurons en entrée le code source Java et des fichiers de configuration, et en sortie une image Docker. Cette dernière repose sur un fichier Dockerfile.

Regardons un peu plus en détail ces deux fichiers.

1. https://spring.io/guides/gs/serving-web-content/
2. http://projects.spring.io/spring-boot/

Le fichier build.gradle

Ce fichier contient, tout d'abord, les librairies à utiliser : SpringBoot (qui inclut également le framework SpingMVC) et GradleDockerPlugin qui permet d'appeler une API Docker distant :

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:1.3.3.RELEASE")
        classpath("com.bmuschko:gradle-docker-plugin:2.6.7")
    }
}
```

Ensuite, nous listons les actions à effectuer : compiler le code Java, mettre en place le serveur d'application SpringBoot et utiliser l'API Docker distant :

```
apply plugin: 'java'
apply plugin: 'spring-boot'
apply plugin: 'com.bmuschko.docker-remote-api'
```

Afin de simplifier la lecture, nous ajoutons des « import » utiles aux tâches Gradle ci-dessous :

```
import com.bmuschko.gradle.docker.tasks.image.Dockerfile
import com.bmuschko.gradle.docker.tasks.image.DockerBuildImage
```

Nous créons un argument qui devra être donné lorsque le script Gradle sera invoqué. Cet argument décrit la version de l'application (par exemple, *master* ou alors le numéro d'une version) :

```
def appVersion = project.getProperty("appVersion")
```

Nous indiquons le nom du jar souhaité pour la compilation Java. Nous omettons volontairement la version, car celle-ci sera traitée au niveau de l'image Docker :

```
jar {
    baseName = "helloworld"
}
```

Nous spécifions la version de Java à utiliser :

```
sourceCompatibility = 1.8
targetCompatibility = 1.8
```

Nous ajoutons les librairies utiles au démarrage du serveur d'application SpringBoot (pour l'environnement de développement) :

```
repositories {
    mavenCentral()
}
dependencies {
    compile("org.springframework.boot:spring-boot-starter-thymeleaf")
    compile("org.springframework.boot:spring-boot-devtools")
}
```

Nous donnons l'adresse de l'API Docker distant (nous verrons plus tard comment démarrer le conteneur exposant cette API) :

```
docker {
    url = 'http://docker-api:2375'
}
```

Nous créons une tâche qui va simplement copier le fichier Dockerfile au niveau du répertoire contenant le résultat de la compilation Java. Il s'agit ici de simplifier les chemins dans le Dockerfile :

```
task copyTask(type: Copy) {
    from 'src/main/resources/docker/Dockerfile'
    into 'build/libs'
}
```

Pour finir, nous créons une tâche qui va construire l'application et l'image Docker à partir du Dockerfile :

```
task buildImage(type: DockerBuildImage) {
    dependsOn build, copyTask
    inputDir = file('build/libs')
    tag = "helloworld:" + appVersion
}
```

Le fichier Dockerfile

Le fichier Dockerfile décrit comment construire l'image de notre application. Ce fichier sera utilisé par le fichier *build.gradle* ;

L'image sera basée sur un CentOS 7 :

```
FROM centos:7
```

Nous installons Java 8 afin de pouvoir démarrer l'application :

```
RUN yum -y install wget
RUN wget --no-cookies --no-check-certificate --header "Cookie:
oraclelicense=accept-securebackup-cookie"
```

```
"http://download.oracle.com/otn-pub/java/jdk/8u77-b03/jre-8u77-linux-x64.rpm"
-O /tmp/jdk-8-linux-x64.rpm
RUN yum -y install /tmp/jdk-8-linux-x64.rpm
ENV JAVA_HOME /usr/java/latest
```

Pour finir, nous copions et démarrons l'application :

```
RUN mkdir /opt/helloworld
COPY helloworld.jar /opt/helloworld/
ENTRYPOINT ["java", "-jar", "/opt/helloworld/helloworld.jar"]
```

Notons que l'application est conditionnée sous la forme d'un JAR unique qui inclut le serveur d'application SpringBoot.

10.2 LA PRÉPARATION DES CONTENEURS

Dans cette section nous allons mettre en place les conteneurs nécessaires à notre système d'intégration continue. Comme nous l'avons vu précédemment, nous aurons besoin de quatre conteneurs :

1. GitLab ;
2. Jenkins ;
3. Docker API ;
4. environnement de développement.

10.2.1 Prérequis

Avant tout, nous avons besoin de configurer un réseau afin que nos conteneurs puissent communiquer les uns avec les autres. Pour cela, nous créons un réseau de type bridge dont le nom est my_ci :

```
$ docker network create -d bridge my_ci
```

10.2.2 Le conteneur dépôt de code Gitlab

Nous utiliserons Git avec GitLab comme outil de gestion de code source. Ce dernier étant très courant, une image Docker existe déjà sur le Docker Hub, et il nous suffit de la démarrer :

```
$ docker run --detach \
    --net my_ci \
    --hostname gitlab \
    -p 8180:80 -p 8122:22 \
    --name gitlab \
```

```
    --restart always \
    --volume /home/vagrant/volumes/gitlab/config:/etc/gitlab \
    --volume /home/vagrant/volumes/gitlab/logs:/var/log/gitlab \
    --volume /home/vagrant/volumes/gitlab/data:/var/opt/gitlab \
    gitlab/gitlab-ce:8.5.8-ce.0
```

Il est également possible de mapper le port 443 afin d'accéder à GitLab via HTTPS ; nous avons volontairement omis ce mappage car nous ne l'utiliserons pas dans notre exemple et nous souhaitons éviter les tracas liés à la gestion des certificats SSL.

Nous remarquons que trois volumes sont nécessaires :

- config : pour stocker la configuration de GitLab, par exemple les utilisateurs ;
- logs : pour stocker les logs ;
- data : pour stocker les données applicatives, c'est-à-dire les dépôts Git.

La création de volumes permet de mettre à jour GitLab avec une nouvelle image sans perdre les données.

Après le démarrage du conteneur, nous devons configurer l'URL de GitLab ainsi que le *timeout* pour l'exécution des *webhooks* (car la valeur par défaut est trop faible pour notre CI). Pour cela, nous ouvrons un terminal sur le conteneur ainsi lancé :

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

Puis nous éditons le fichier */etc/gitlab/gitlab.rb* à l'aide, par exemple, de l'utilitaire *vi* :

```
root@gitlab:/# vi /etc/gitlab/gitlab.rb
```

Nous pouvons alors ajouter les lignes suivantes :

```
## Latest options listed at https://gitlab.com/gitlab-org/omnibus-
gitlab/blob/master/files/gitlab-config-template/gitlab.rb.template

## Url on which GitLab will be reachable.
## For more details on configuring external_url see:
## https://gitlab.com/gitlab-org/omnibus-
gitlab/blob/629def0a7a26e7c2326566f0758d4a27857b52a3/README.md#configuring-the-
external-url-for-gitlab
# external_url 'GENERATED_EXTERNAL_URL' # default: http://hostname
external_url 'http://gitlab:8180'
gitlab_rails['webhook_timeout'] = 30

## Note: configuration settings below are optional.
## Uncomment and change the value.
###############################
# gitlab.yml configuration #
###############################
...
```

Nous pouvons terminer la connexion avec le conteneur en entrant simplement la commande *exit* :

```
root@gitlab:/# exit
```

Notons que puisque le dossier /etc/gitlab du conteneur est un volume monté sur un répertoire de notre hôte, nous aurions aussi pu faire cette modification directement depuis l'hôte. Néanmoins nous aurions dû prendre garde aux problèmes de droits que nous avons déjà évoqués dans le chapitre 9. Le résultat est, quoi qu'il en soit, identique.

Finalement, nous devons redémarrer le conteneur pour qu'il prenne en compte ces modifications :

```
$ docker restart gitlab
```

10.2.3 Le conteneur d'intégration continue Jenkins

Nous utiliserons Jenkins comme outil d'intégration continue. Tout comme pour GitLab, une image standard du Docker Hub, prête à l'emploi, peut être utilisée.

Démarrons un conteneur :

```
$ docker run --detach \
--net my_ci \
--hostname jenkins \
-u root \
-p 8080:8080 -p 50000:50000 \
--name jenkins \
--volume /home/vagrant/volumes/jenkins:/var/jenkins_home \
jenkinsci/jenkins:2.0-beta-1
```

Comme nous pouvons le voir dans la dernière ligne ci-dessus, nous utilisons une version *beta* de Jenkins. Nous avons fait ce choix afin de présenter la dernière monture de Jenkins qui sera tout prochainement disponible. Il est évident que pour une vraie application ce choix ne serait pas judicieux, néanmoins cela ne porte pas à conséquence pour notre exemple.

10.2.4 Le conteneur d'API Docker distant Docker API

L'API Docker représente un Docker dans Docker. Ici aussi, une image est déjà existante ; démarrons-la :

```
docker run --detach \
--net my_ci \
--hostname docker-api \
```

```
--privileged \
--name docker-api \
docker:1.10-dind
```

`--privileged` est nécessaire pour qu'un Docker dans Docker fonctionne correctement, cependant il est important de comprendre que cette option donne un accès complet à l'hôte, si bien qu'elle doit être utilisée avec une grande précaution.

10.2.5 Les conteneurs d'environnement de développement

Pour notre environnement de développement nous avons besoin de deux conteneurs : le premier pour tester les développements réalisés, et le second pour le volume (où sera le code source).

Commençons par l'environnement qui permettra de tester les développements. Nous avons besoin de Java pour compiler le code et de Gradle pour déployer localement l'application. Ainsi, nous utilisons le Dockerfile (Dockerfile_devenv) suivant :

```
FROM centos:7

RUN yum -y install wget unzip

RUN wget --no-cookies --no-check-certificate --header "Cookie:
oraclelicense=accept-securebackup-cookie"
"http://download.oracle.com/otn-pub/java/jdk/8u//-b03/jdk-8u//-linux-x64.rpm"
-O /tmp/jdk-8-linux-x64.rpm
RUN yum -y install /tmp/jdk-8-linux-x64.rpm
ENV JAVA_HOME /usr/java/latest

RUN wget -N https://services.gradle.org/distributions/gradle-2.12-all.zip
RUN mkdir /opt/gradle
RUN unzip gradle-2.12-all.zip -d /opt/gradle
RUN ln -sfn /opt/gradle/gradle-2.12 /opt/gradle/latest
ENV GRADLE_HOME /opt/gradle/latest

WORKDIR /var/hello-world

ENTRYPOINT ["/opt/gradle/latest/bin/gradle", "clean", "bootRun",
"-PappVersion=master"]
```

Le dernier paramètre du ENTRYPOINT (`-PappVersion=master`) ne sera, dans notre cas, pas employé, car nous utiliserons le code présent dans le volume. Néanmoins, ce paramètre est obligatoire, aussi l'avons-nous ajouté avec la valeur arbitraire master.

Construisons maintenant l'image ainsi spécifiée :

```
$ docker build -t devenv .
```

Continuons avec le volume : nous utilisons pour cela le Dockerfile (*Dockerfile_devenv*-volume) suivant qui permet de spécifier un *data container*, c'est-à-dire un conteneur dont la seule raison d'être est de garder un lien sur un volume :

```
FROM busybox:glibc
VOLUME /var/hello-world
```

Nous pouvons construire l'image :

```
$ docker build -t devenv-volume .
```

Créons maintenant un conteneur pour notre volume :

```
$ docker create -v /home/vagrant/volumes/devenv/hello-world/:/var/hello-world/
--name=devenv-volume devenv-volume
```

Nous lierons notre volume à notre environnement de développement lors du démarrage de ce dernier, cette étape sera faite ultérieurement. Ainsi, notre environnement est prêt. Nous verrons par la suite comment l'utiliser, et nous commencerons notamment par cloner le *code base* depuis GitLab (qui lui-même contiendra un *fork* d'un projet GitHub) dans /home/vagrant/volumes/devenv/hello-world.

10.2.6 Mise à jour du fichier /etc/hosts sur notre machine hôte

Notre réseau Docker est déjà en place. Regardons sa composition :

```
$ docker network inspect my_ci
[
    {
        "Name": "my_ci",
        "Id":
"b29c0ed96b3e1867eacb4f7d2e9f2771e4193de967f53a2b8cd236429ec39868",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1/16"
                }
            ]
        },
        "Containers": {
            "652792c0a0d4af5baf99f92cd28cb55c63ada4a9cf683d62a86c9eec1fa5fd1d":
{
                "Name": "jenkins",
                "EndpointID":
"0b710a84dceefbbc64d1f9a2f9f92975e693ecf1903d33597dac3a4237cab51c",
```

```
                "MacAddress": "02:42:ac:12:00:03",
                "IPv4Address": "172.18.0.3/16",
                "IPv6Address": ""
            },
            "6e2c6048defa781a97ab55f3a9878b3fdc8a00438854652de51f400af6b10450":
{
                "Name": "docker-api",
                "EndpointID":
"22ac569a88b7189f04d26e7d8f575b5c59daa46d0f61ca70e3fb0bacf3db44d8",
                "MacAddress": "02:42:ac:12:00:04",
                "IPv4Address": "172.18.0.4/16",
                "IPv6Address": ""
            },
            "b222b70940d6476a9da5732f1df7aa6fd0224886597c5926090ee5d56529111f":
{
                "Name": "gitlab",
                "EndpointID":
"8b83b7afa1db47c1912bfa23adce76e46f1da565c145d1f0f4aaeabbb6eeec66",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {}
    }
]
```

Docker a automatiquement configuré tous les conteneurs afin qu'ils se connaissent les uns les autres. Toutefois, ce n'est pas le cas de notre machine hôte. Celui-ci peut accéder aux conteneurs par l'intermédiaire de leur adresse IP, mais nous préférons permettre un appel via un nom de domaine.

Nous allons donc modifier la configuration de notre hôte à cet effet :

```
$ sudo vi /etc/hosts
```

Nous ajoutons les lignes suivantes (en fin de fichier) :

```
172.18.0.3 jenkins
172.18.0.4 docker-api
172.18.0.2 gitlab
```

Attention, il convient de vérifier les IP en fonction des conteneurs (celles-ci étant allouées dynamiquement par Docker) selon le résultat de la commande `docker network inspect my_ci`.

Nous pouvons maintenant tester que l'API Docker fonctionne correctement :

```
$ curl http://docker-api:2375/images/json
```

10.3 CONFIGURATION DE NOTRE CI

10.3.1 Initialiser GitLab pour l'application HelloWorld

La première étape est de récupérer le code de l'application depuis GitHub. Pour cela, nous commençons par initialiser GitLab.

Depuis un navigateur, allez à l'adresse suivante :

```
http://gitlab:8180
```

Et connectez-vous avec l'utilisateur suivant :

- Nom d'utilisateur : root
- Mot de passe : 5iveL!fe

Après la première connexion, GitLab nous propose de changer de mot de passe root, et nous utiliserons arbitrairement un mot de passe simple : root1234. Après la modification, nous devons nous reconnecter.

Créons maintenant notre projet hello-world (figure 10.2) :

1. Cliquez sur *New project*.
2. Sous *Project path*, saisissez « hello-world ».
3. Pour *Import project from*, choisissez *Any repo by URL*.
4. Sous *Git repository URL*, saisissez https://github.com/dunod-docker/hello-world.git.
5. Pour *Visibility Level*, choisissez *Public*.
6. Cliquez sur *Create Project*.

10.3.2 Testons l'application

Nous pouvons maintenant tester l'application localement. Pour cela, nous allons utiliser notre environnement de développement.

Commençons par cloner le code depuis GitLab :

```
$ sudo mkdir -p /home/vagrant/volumes/devenv
$ sudo chown vagrant:vagrant /home/vagrant/volumes/devenv
$ cd /home/vagrant/volumes/devenv
$ git clone http://gitlab:8180/root/hello-world.git
```

Nous pouvons maintenant démarrer notre conteneur d'environnement de développement en l'attachant au volume devenv-volume :

```
$ docker run --rm -p 8090:8090 --volumes-from=devenv-volume --name=devenv
devenv
```

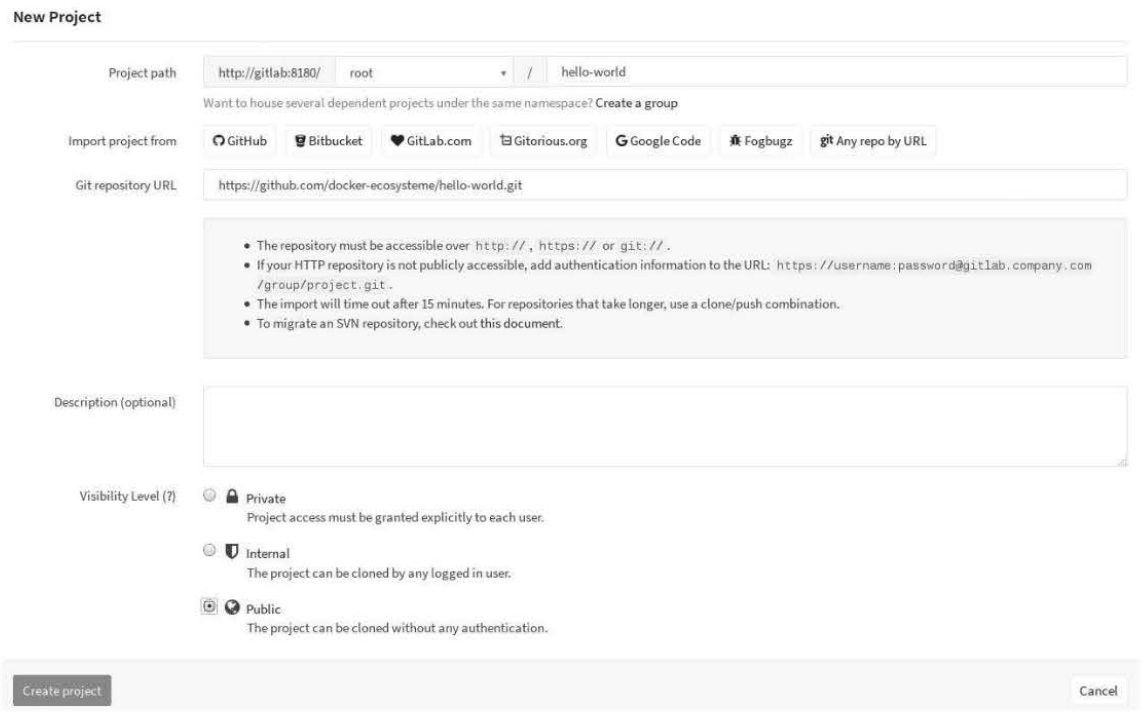

Figure 10.2 — Configuration de GitLab

Notre application est maintenant directement disponible à l'adresse http://localhost:8090/hello?name=John.

Pour finir, arrêtons le conteneur (étant donné l'option `--rm` passée au démarrage du conteneur, ce dernier sera également supprimé) :

```
$ docker stop devenv
```

L'avantage d'une telle approche est évident : il n'est pas nécessaire d'avoir à installer sur notre poste de travail Gradle ou Java, ces derniers étant fournis par notre conteneur de développement.

10.3.3 Configuration de Jenkins

Nous allons maintenant configurer Jenkins, notamment les plugins à utiliser et la gestion des permissions.

Installation des plugins et création de l'utilisateur admin

Depuis un navigateur, allons à l'adresse suivante :

```
http://jenkins:8080/
```

L'écran suivant est affiché :

Figure 10.3 — Premier démarrage de Jenkins

Nous devons saisir le mot de passe d'administration qui a été créé par défaut lors du démarrage du conteneur. Ce mot de passe ayant été sauvegardé dans un fichier, affichons ce dernier :

```
$ sudo cat /home/vagrant/volumes/jenkins/secrets/initialAdminPassword
e5f5c2489a454e229f71f019494fbc49
```

Dans notre cas, le mot de passe est « e5f5c2489a454e229f71f019494fbc49 », et il suffit de le saisir dans l'écran, puis de cliquer sur Continue.

Le système nous demande ensuite le type d'installation que nous souhaitons. Nous choisissons *Install suggested plugins*, en cliquant simplement sur le bouton correspondant :

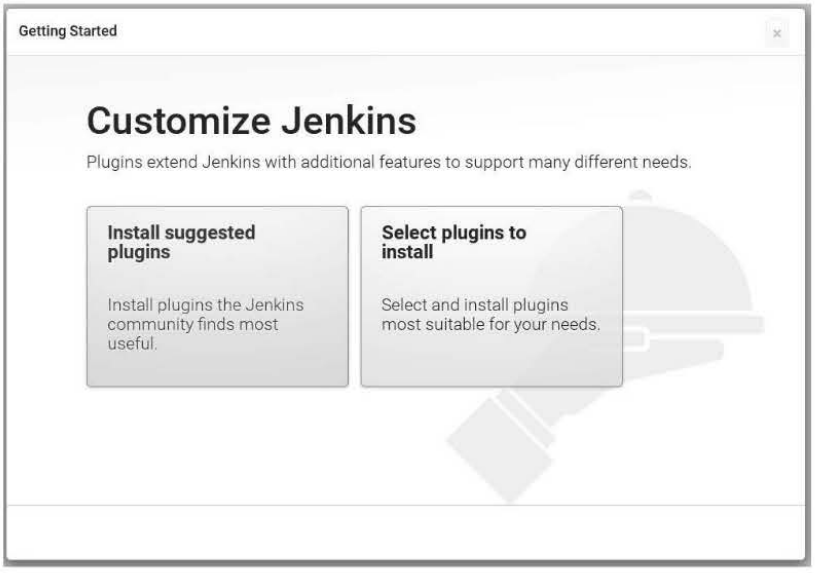

Figure 10.4 — Installation des plugins Jenkins

L'installation des plugins est alors effectuée, cette opération pouvant durer quelques minutes.

À la fin de l'installation, le système nous demande de créer un premier utilisateur ; nous utiliserons :

- Nom d'utilisateur : admin
- Mot de passe : admin1234
- Nom : Administrateur
- Email : votre adresse email

Connectons-nous à Jenkins avec les informations saisies ci-dessus.

Installation de Gradle

Afin de pouvoir construire notre application, nous devons configurer Gradle :

1. Cliquez sur *Administrer Jenkins* (dans le menu à gauche).
2. Cliquez sur *Global Tool Configuration*.
3. Descendez dans la page au niveau du titre *Gradle*.
4. Cliquez sur *Add Gradle*.
5. Sous *Name*, saisissez « Gradle 2.12 ».
6. Cochez *Install automatically*.
7. Sous *Version*, saisissez « 2.12 » (il se peut également qu'il s'agisse d'une liste déroulante, dans ce cas choisissez *Gradle 2.12*).

Figure 10.5 — Configuration des jobs Jenkins

8. Cliquez sur *Save* (en bas de page).
9. Retournez dans l'administration Jenkins et choisissez *Gestion des plugins*.
10. Cliquez sur l'onglet *Avancé*.
11. Tout en bas, cliquez sur *Vérifier maintenant*.

Création de l'utilisateur gitlab

Nous allons maintenant ajouter un utilisateur gitlab afin que ce dernier puisse exécuter des builds automatiques (grâce à un *webhook* que nous configurerons dans le prochain chapitre) :

1. Cliquez sur *Administrer Jenkins*.
2. Cliquez sur *Gérer les utilisateurs*.
3. Cliquez sur *Créer un utilisateur*, et saisissez les informations suivantes :
   - Nom d'utilisateur : gitlab
   - Mot de passe : gitlab1234
   - Nom complet : GitLab
   - Adresse courriel : une adresse email de votre choix

Figure 10.6 — Création d'un utilisateur GitLab

4. Cliquez sur *Créer un utilisateur*.

Assignation des permissions

Pour finir, nous configurons la gestion des permissions Jenkins :

1. Cliquez sur *Administrer Jenkins*.
2. Cliquez sur *Configurer la sécurité globale*.
3. Sous *Contrôle de l'accès/Autorisations*, sélectionnez *Stratégie d'authorisation matricielle basée sur les projets*.
   Sous *Utilisateur/group à ajouter*, saisissez « admin », cliquez sur *Ajouter*, puis cochez toutes les cases.
   Sous *Utilisateur/group à ajouter*, saisissez « gitlab », cliquez sur *Ajouter*, puis cochez *Read* sous *Global*.
4. Plus bas, décochez l'option *Se protéger contre les exploits de type Cross Site Request Forgery*.
5. Cliquez finalement sur *Enregistrer*.
6. Nous avons maintenant tout ce qu'il nous faut pour passer à la création des différents jobs nécessaire à la CI.

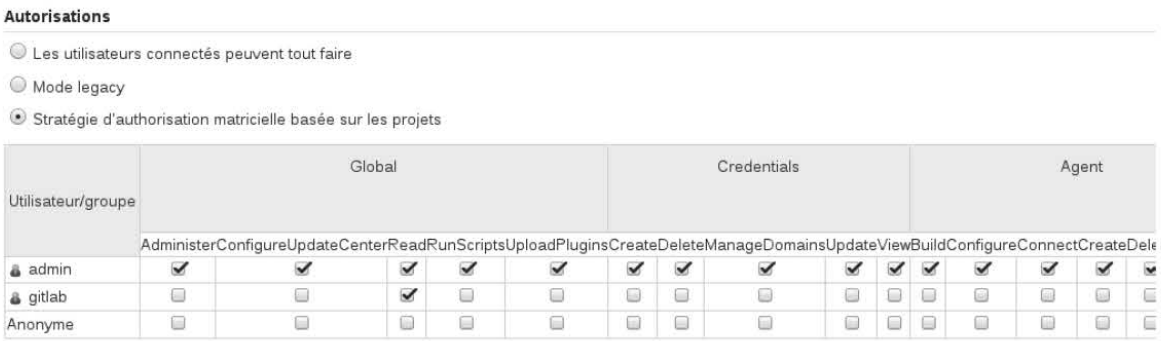

Figure 10.7 — Configuration des droits sur les jobs Jenkins

10.3.4 Création des jobs Jenkins et Web Hook

Nous allons maintenant préparer les trois jobs décrits précédemment : celui pour la construction de l'image et ceux traitant du déploiement de l'application (sous la forme de conteneurs) pour l'environnement de test et celui de production. Nous créerons également le *webhook* dans GitLab qui déclenchera le build et le déploiement automatiques sur l'environnement de test.

Le job Build

Nous créons dans Jenkins le job qui construira notre application :

1. Cliquez dans le menu à gauche sur *Nouveau Item* (sic).
2. Sous *New item name…*, saisissez « hello-world ».
3. Choisissez *Construire un projet free-style*.
4. Cliquez sur *OK*.
5. Pour l'onglet *General* :
   Cochez *Activer la sécurité basée projet*.
   Sous *Utilisateur/groupe à ajouter*, saisissez « gitlab » et cliquez sur *Ajouter*, puis cochez *Build* et *Read* sous *Job*.

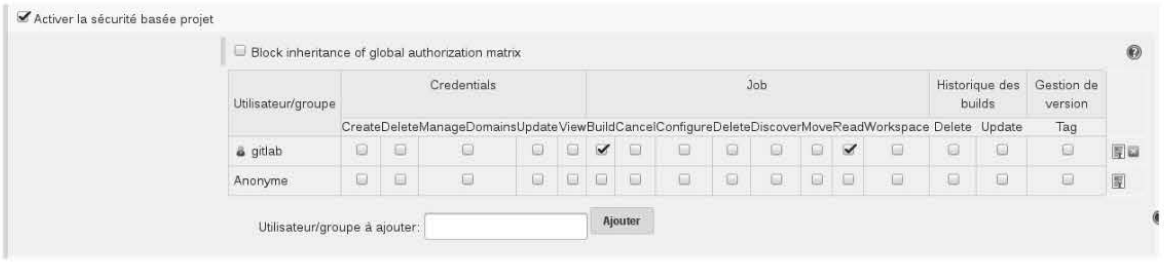

Figure 10.8 — Configuration du job de build

Plus bas, cochez *Ce build a des paramètres*, et créez deux paramètres de type Paramètre String :

- Nom : appVersion ; Valeur par défaut : master
- Nom : gitPath ; Valeur par défaut : *

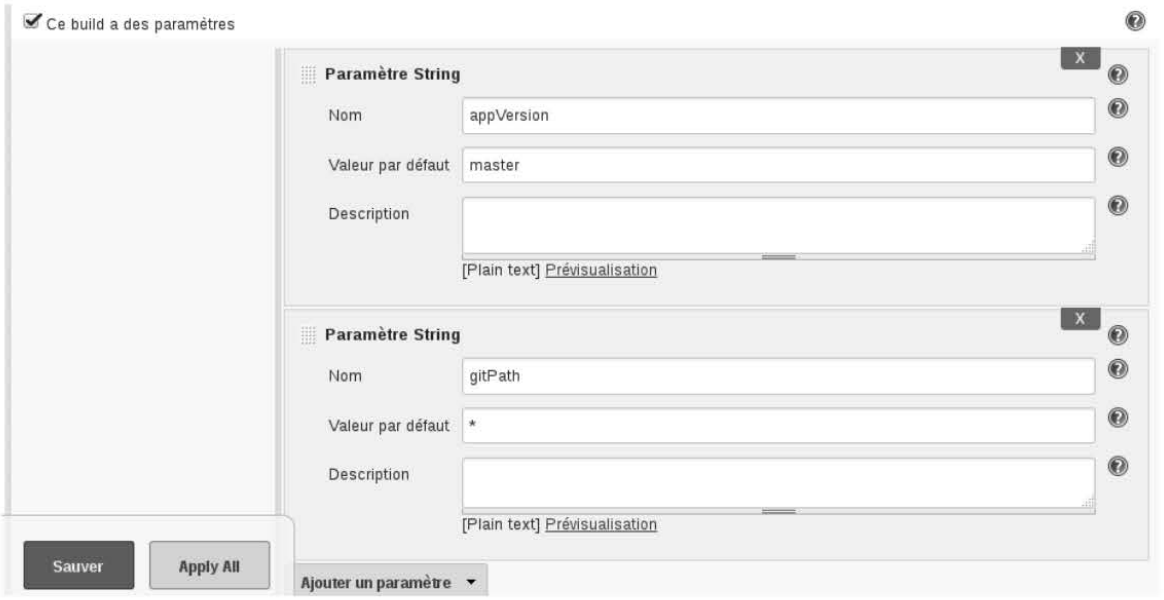

Figure 10.9 — Configuration du job de build

Ces deux paramètres nous permettront de définir la branche ou le *tag* dans GitLab à utiliser comme *base code*. Par défaut, il s'agit de la branche *master*.

6. Pour l'onglet *Gestion de code source*, choisissez *Git* et remplissez avec les informations suivantes :
   - Repository URL : http://gitlab:8180/root/hello-world.git
   - Branch Specifier (blank for 'any') : $gitPath/$appVersion

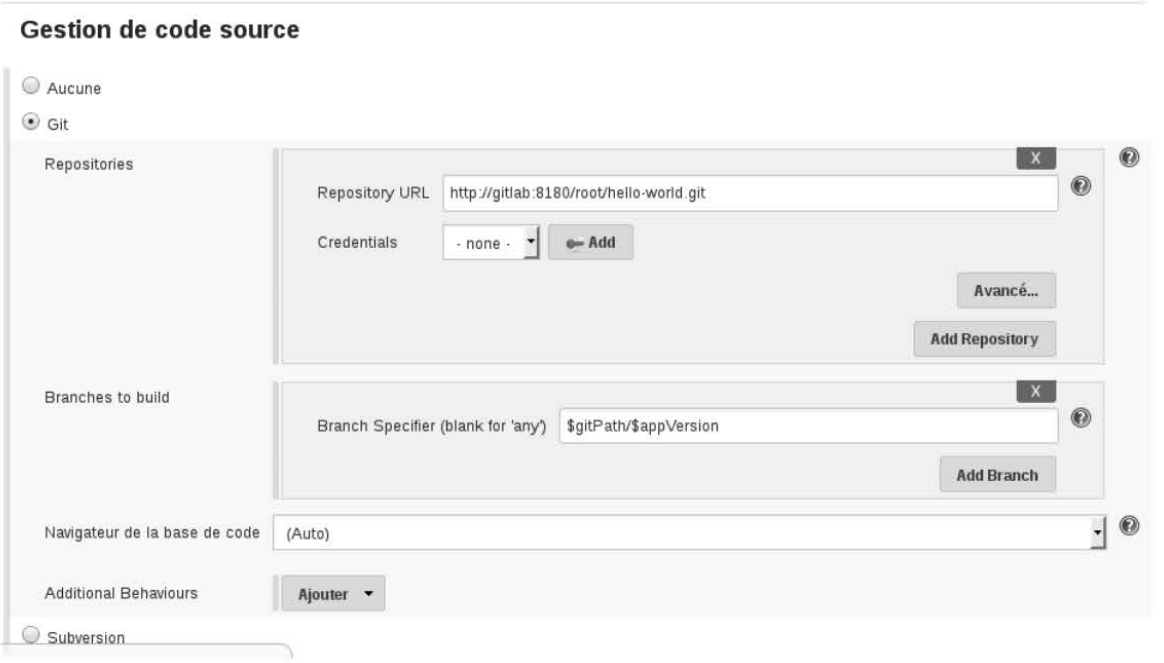

Figure 10.10 — Paramétrage du lien avec le repository Git

7. Pour l'onglet *Ce qui déclenche le build*, cochez *Déclencher les builds à distance (Par exemple, à partir de scripts)*.
   Sous *Jeton d'authentification*, saisir « abcdef ».

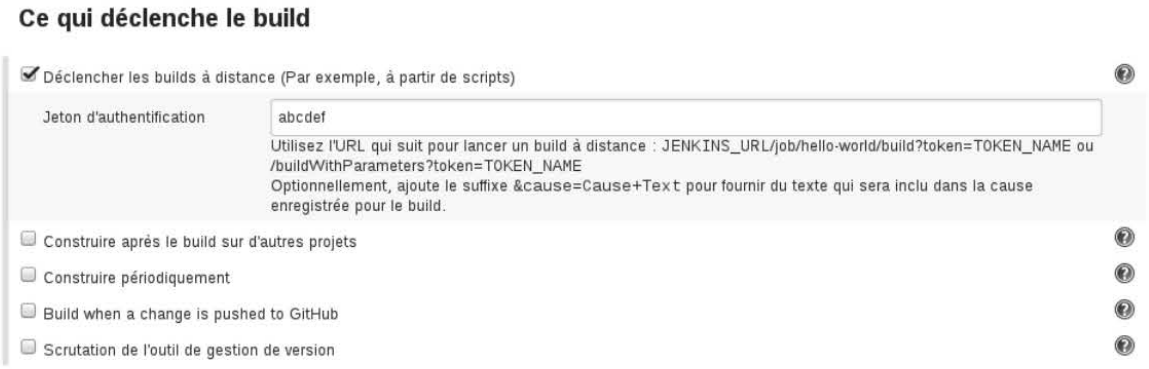

Figure 10.11 — Configuration du WebHook

Nous avons choisi volontairement un jeton très simple, ce dernier devant être donné comme paramètre lors de la création du webhook. Il convient logiquement de choisir une valeur complexe dans un vrai environnement.

8. Pour l'onglet *Build* (figure 10.12) :
   Cliquez sur *Ajouter une étape au build* et choisissez *Invoke Gradle script*, puis *Invoke Gradle*.
   Pour *Gradle Version*, choisissez *Gradle 2.12*.
   Sous *Tasks*, saisissez :

```
clean
buildImage
-PappVersion=$appVersion
```

`clean` permet de nettoyer le résultat d'une précédente exécution Gradle, `buildImage` exécute la tâche du même nom présente dans le fichier de configuration Gradle, et `-PappVersion=$appVersion` permet de spécifier la version de l'image Docker qui sera créée ; logiquement, cette dernière est alignée avec la branche ou le tag (pour une release) utilisés dans GitLab.

9. Cliquez sur *Sauver* (en bas de page).

Le webhook dans GitLab

Le webhook dans GitLab permet le déclenchement automatique d'un build Jenkins après tout *push* dans Git. Sa création se fait ainsi :

1. Ouvrez le projet hello-world dans GitLab.
2. Cliquez sur *Edit project* (figure 10.13).
3. Dans le menu de gauche, cliquez sur *Web Hooks*.

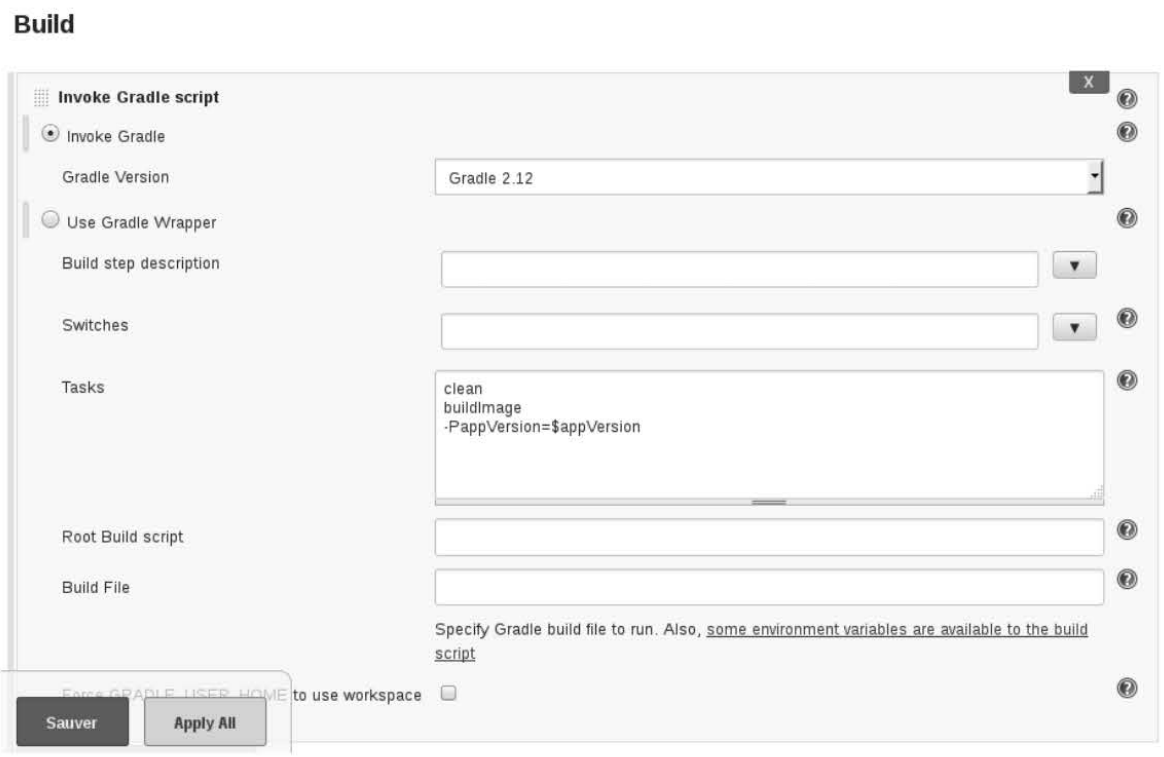

Figure 10.12 — Définition du script de build (appel Gradle)

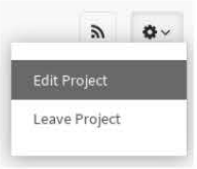

Figure 10.13 — Édition du projet GitLab

4. Sous URL, saisissez http://gitlab:gitlab1234@jenkins:8080/job/hello-world/buildWithParameters?token=abcdef&appVersion=master&gitPath=*.

   Nous devons spécifier les paramètres du build (dans notre cas appVersion et gitPath) ; dans notre exemple, il s'agit simplement de la branche master de GitLab car le webhook concerne l'environnement de test qui doit toujours contenir la dernière version du *code base*.

   Vous pouvez également remarquer que nous devons spécifier un utilisateur et un mot de passe dans l'URL. Nous avons logiquement choisi l'utilisateur gitlab qui dispose de l'unique droit de build pour le job hello-world.

5. Pour Trigger, cochez uniquement *Push events*.
6. Pour SSL Verification, décochez *Enable SSL verification*.
7. Finalement, sauvez avec *Add Web Hook*.

Nous allons tester que le Web Hook fonctionne correctement, et, pour cela, il suffit de cliquer sur le bouton Test Hook et de vérifier qu'un nouveau Build a bien eu lieu dans Jenkins pour notre application hello-world et qu'il a fonctionné.

Le résultat devrait être similaire à celui-ci :

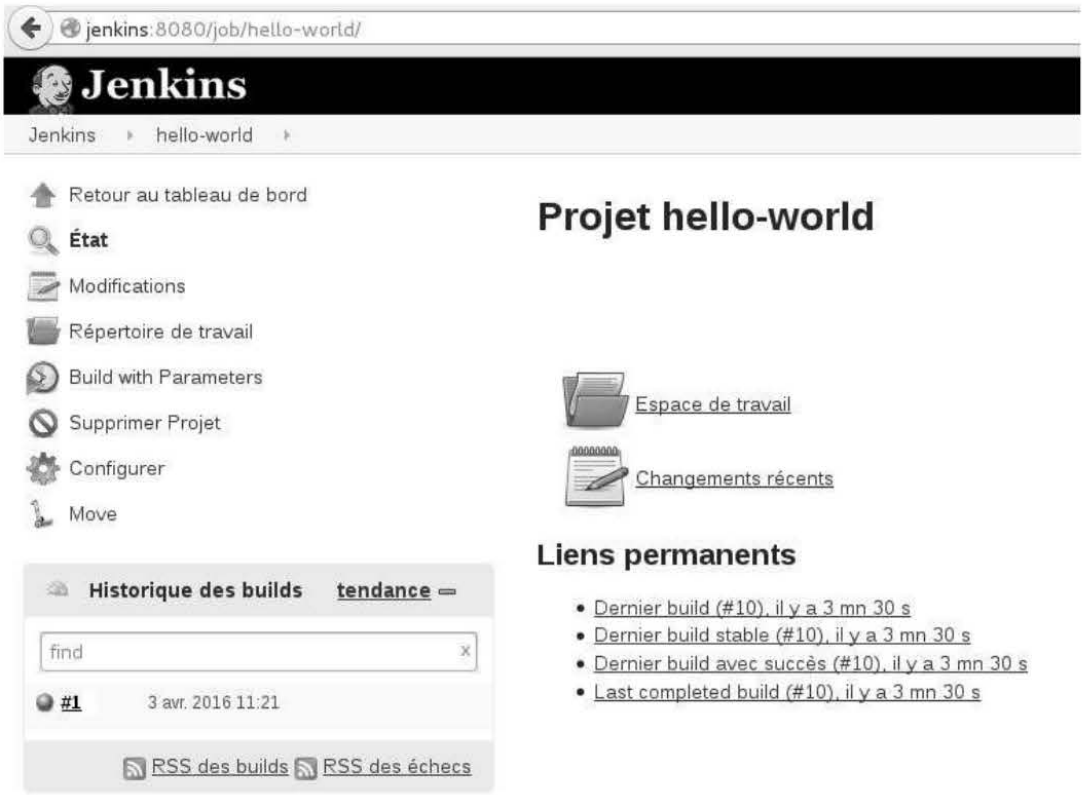

Figure 10.14 — Résultat d'un build dans Jenkins

Le job de déploiement de l'environnement de test Deploy test

Nous créons ensuite dans Jenkins le job qui démarrera un conteneur pour notre environnement de test. Pour rappel, ce conteneur sera localisé dans le conteneur docker-api, et sera créé grâce à l'API Docker distant présente dans ce dernier :

1. Cliquez dans le menu à gauche sur *Nouveau Item* (sic).
2. Sous *New item name…*, saisissez « deploy-test-hello-world ».
3. Choisissez *Construire un projet free-style*.
4. Cliquez sur *OK*.
5. Pour l'onglet *Ce qui déclenche le build* :
   Cliquez sur *Construire après le build sur d'autres projets* (afin de déployer automatiquement l'environnement de test après tout *push* dans GitLab).
   Sous *Projet à surveiller*, saisissez « hello-world ».

Figure 10.15 — Paramétrage du job de déploiement en test

6. Pour l'onglet *Build* :
   Pour *Ajouter une étape au build*, choisissez *Exécuter un script shell.*
   Sous *Commande*, saisissez :

```
# suppression du conteneur existant
curl -v -X POST -H "Content-Type: application/json"
http://docker-api:2375/containers/test-helloworld/stop
curl -v -X DELETE http://docker-api:2375/containers/test-helloworld

# création et démarrage d'un nouveau conteneur
curl -v -X POST -H "Content-Type: application/json" -d
'{"Image":"helloworld:master", "ExposedPorts": { "8090/tcp": {} },
"PortBindings": { "8090/tcp": [{ "HostPort": "8490" }] } }'
http://docker-api:2375/containers/create?name=test-helloworld
curl -v -X POST -H "Content-Type: application/json"
http://docker-api:2375/containers/test-helloworld/start
```

   Notons que nous commençons par arrêter et supprimer le conteneur existant. Lors du premier déploiement, ce conteneur n'existe pas, et l'API Docker retournera des erreurs http 404, ce qui n'est pas gênant pour notre job.
   Nous constatons également que nous utilisons la branche *master* (ou plutôt l'image de version master) lors de la création du conteneur, car nous voulons toujours déployer en test la dernière version de l'application.

7. Cliquez sur *Sauver* (en bas de page).

Il faut noter que ce conteneur utilise le fichier de propriétés par défaut de l'application (c'est-à-dire src/main/resources/application.properties) qui utilise le port 8090. Cette façon de faire est acceptable pour un seul environnement ; cependant, pour tout autre environnement il faudra utiliser un autre port (autrement dit un autre fichier de propriétés), ce qui sera le cas pour l'environnement de production expliqué ci-après.

Le job de déploiement de l'environnement de production Deploy prod

Notre environnement de production est très proche de celui de test, à l'exception des points suivants :

- il n'est pas déployé automatiquement, autrement dit il n'y aura pas de configuration au niveau de ce qui déclenche le build ;
- nous ne devons pas utiliser la branche *master*, mais une release spécifique. Pour cela nous ajouterons un paramètre au build afin de spécifier manuellement le *tag* à utiliser ou, plus précisément, la version de l'image Docker issue du *tag* ;
- le fichier de propriétés que nous avons accepté d'utiliser pour l'environnement de test ne devra pas être utilisé pour l'environnement de production, car il y aurait un conflit de ports si bien que nous devrons le surcharger.

Les fichiers de propriétés

Nous créons tout d'abord un nouveau projet dans GitLab pour gérer les fichiers de propriétés relatifs aux environnements (dans notre cas la production uniquement). Pour rappel, on utilise le fichier de propriétés par défaut pour l'environnement de test.

> Ce genre de projet doit être protégé afin de limiter son accès aux personnes autorisées car il pourrait contenir des données sensibles, telles que des mots de passe. Dans notre cas, nous le laissons « Public » pour simplifier le processus.

Créons donc ce nouveau projet :

1. Dans GitLab, cliquez sur *New project*.
2. Sous *Project path*, saisissez « environnements ».
3. Pour *Visibility Level*, choisissez *Public*.
4. Cliquez sur *Create Project*.

Ajoutons un fichier *prod.properties* directement à la racine de ce nouveau projet :

1. Cliquez sur l'icône « + » et choisissez *New files*.
2. Sous *File name*, saisissez « prod.properties ».
3. Saisissez le contenu suivant :

```
server.port = 8091
```

4. Commitez le fichier dans la branche *master* en n'oubliant pas de spécifier un message.

Le job Jenkins

1. Dans Jenkins, cliquez dans le menu à gauche sur *Nouveau Item* (sic).
2. Sous *New item name…*, saisissez « deploy-prod-hello-world ».
3. Choisissez *Construire un projet free-style*.
4. Cliquez sur *OK*.

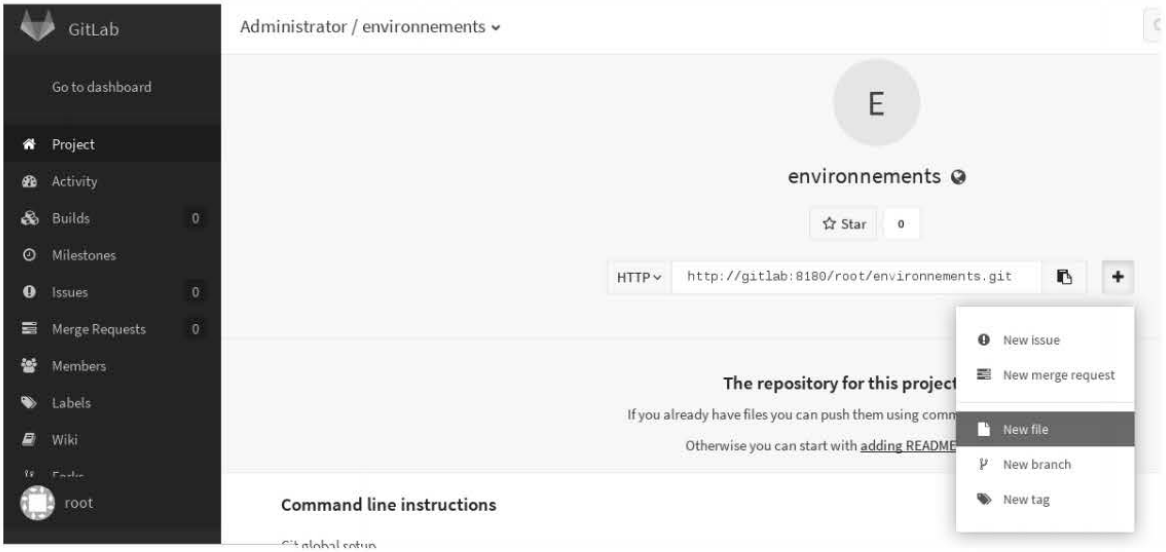

Figure 10.16 — Création d'un repo Git pour contenir les propriétés d'environnement

5. Pour l'onglet *General* :
   Cochez *Ce build a des paramètres*, et créez un paramètre de type Paramètre String :
   - Nom : appVersion ; Pas de valeur par défaut

   Ce paramètre correspond à la version de l'image Docker à utiliser.
6. Pour l'onglet *Build* :
   Pour *Ajouter une étape au build*, choisissez *Exécuter un script shell.*
   Sous *Commande*, saisissez :

```
# suppression du conteneur existant
curl -v -X POST -H "Content-Type: application/json"
http://docker-api:2375/containers/prod-helloworld/stop
curl -v -X DELETE http://docker-api:2375/containers/prod-helloworld

# récupération des fichiers de propriétés depuis git
rm -rf environnements
rm -f environnements.tar
git clone http://gitlab:8180/root/environnements.git
tar -cvzf environnements.tar environnements

# création d'un nouveau conteneur
JSON_CONTENT="{\"Image\":\"helloworld:${appVersion}\", \"Entrypoint\":
[\"java\", \"-jar\", \"/opt/helloworld/helloworld.jar\",
\"--spring.config.location=/opt/helloworld/environnements/prod.properties\"],
\"ExposedPorts\": { \"8091/tcp\": {} }, \"PortBindings\": { \"8091/tcp\": [{
\"HostPort\": \"8491\" }] } }"
curl -v -X POST -H "Content-Type: application/json" -d "$JSON_CONTENT"
http://docker-api:2375/containers/create?name=prod-helloworld

# ajout du fichier de propriétés dans le conteneur
```

```
curl -v -X PUT
http://docker-api:2375/containers/prod-helloworld/archive?path=/opt/helloworld
--upload-file environnements.tar

# démarrage du conteneur
curl -v -X POST -H "Content-Type: application/json"
http://docker-api:2375/containers/prod-helloworld/start
```

Ce script va, en plus de créer et démarrer le conteneur pour l'environnement de production, injecter les propriétés : pour cela, nous copions le fichier prod.properties dans le conteneur avant de le démarrer. Notons que nous faisons cela grâce à une archive tar car l'API Docker ne permet une copie que par ce biais-là.
Lors du démarrage du conteneur, nous spécifions le fichier de propriétés à utiliser. Nous constatons également que nous utilisons la version de l'image spécifiée par le paramètre appVersion.

7. Cliquez sur *Sauver* (en bas de page).

Cette fois-ci notre CI est prête ! Nous allons dans la prochaine section illustrer son exploitation par un exemple.

10.4 L'EXPLOITATION DE NOTRE CI

Dans les précédentes sections, nous avons mis en place la totalité des composants utiles à notre CI, et nous pouvons enfin l'exploiter. Cette dernière section est un cas d'utilisation réaliste du système mis en place.

10.4.1 Développement d'une nouvelle fonctionnalité

Un besoin métier a été exprimé : la couleur du « HelloWorld » ne doit plus être noire, mais rouge. Un développeur est assigné et doit donc modifier le code de l'application, le tester localement et « pusher » ses modifications.

La modification demandée concerne la vue de l'application, soit le fichier suivant (c'est-à-dire un fichier du volume devenv-volume) :

```
/home/vagrant/volumes/devenv/hello-
world/src/main/resources/templates/hello.html
```

Ouvrons le fichier avec un éditeur de texte et modifions-le ainsi :

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Getting Started: Serving Web Content</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
```

```
<body>
    <p style="color:red;" th:text="'Hello, ' + ${name} + '!'" />
</body>
</html>
```

Nous pouvons simplement tester notre modification en démarrant notre conteneur de développement :

```
$ docker run --rm -p 8090:8090 --volumes-from devenv-volume --name devenv
devenv
```

Ouvrons un navigateur à l'adresse suivante :

```
http://localhost:8090/hello?name=John
```

Nous constatons que le texte est désormais en rouge. Nous pouvons arrêter notre environnement de développement :

```
$ docker stop devenv
```

Publions la modification dans le dépôt git :

```
$ cd /home/vagrant/volumes/devenv/hello-world/
$ git add src/main/resources/templates/hello.html
$ git commit -m"Texte en rouge"
$ git push origin master:master
```

Utilisez l'utilisateur suivant lorsque le système le demande :

- Nom d'utilisateur : root
- Mot de passe : root1234

Si lors d'une commande *git* l'erreur suivante s'affiche :

```
*** Please tell me who you are.
Run
  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"
...
```

il faut alors spécifier votre nom et adresse email grâce aux commandes suivantes :

```
$ git config --global user.email "you@example.com"
$ git config --global user.name "Your Name"
```

Nous vous rappelons que nous avons installé un webhook dans GitLab afin de builder et déployer en test automatiquement notre application après tout *push*.

Vérifions que tout s'est bien déroulé dans Jenkins :

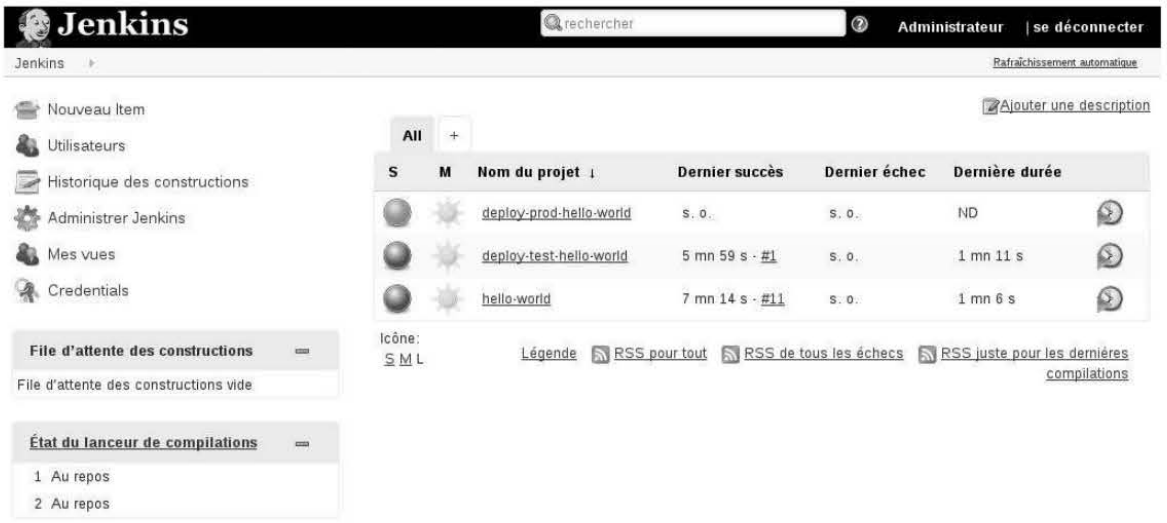

Figure 10.17 — Résultat d'un build et d'un déploiement en test dans Jenkins

Les jobs hello-world (pour la construction) et deploy-test-hello-world (pour le déploiement) contiennent une pastille bleue, ce qui signifie que leur dernière exécution s'est bien déroulée. Un testeur peut désormais constater le résultat simplement en atteignant l'URL suivante depuis un navigateur :

```
http://docker-api:8490/hello?name=John
```

10.4.2 Mise en production

Les tests applicatifs (dans notre cas, la vérification de la couleur rouge) sont positifs. Une mise en production de l'application est désormais souhaitée. Pour cela, nous devons « releaser » le code, et nous utiliserons un *tag git* pour y parvenir.

> Lors d'une mise en production, dans un mode d'intégration continue, il convient d'avertir tous les développeurs de ne plus « pusher » de code dans la branche courante (master dans notre cas) afin d'éviter toute altération du code avant la release.

Ouvrons le projet hello-world dans GitLab et cliquons sur « 0 tags » dans le panneau principal :

Créons un nouveau tag :

1. Cliquez sur *New tag*.
2. Pour *Tag name*, saisir « v1.0.0 ».
   Par convention, les noms de tags git suivent le format vX.Y.Z, où v signifie version, X le numéro de version majeure, Y le numéro de version mineure et Z le numéro utilisé pour des corrections d'anomalie.
3. Sous *Create from*, saisissez « master ».
4. Cliquez sur *Create tag*.

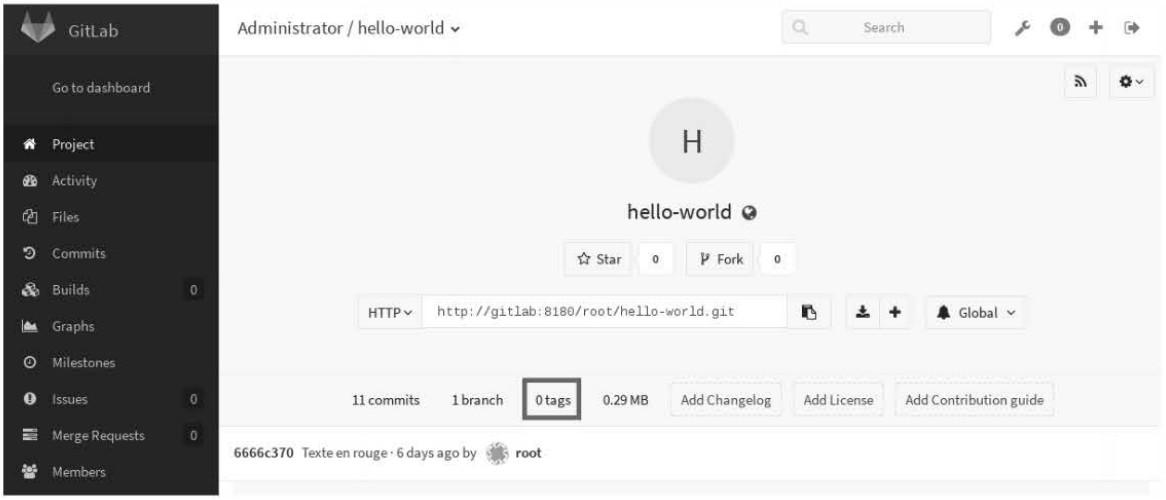

Figure 10.18 — Tagging dans GitLab

Le tag est maintenant créé. Nous pouvons ainsi construire le conteneur qui lui est associé (car par défaut notre job de build construit un conteneur pour la branche master). Pour cela nous utilisons Jenkins.

1. Cliquez sur le projet *hello-world*.
2. Cliquez sur *Build with Parameters*.
3. Pour *appVersion*, saisissez « v1.0.0 ».
4. Pour *gitPath*, saisissez « tags ».
   Le caractère « * » décrit tout chemin possible représentant des branches ; dans notre cas, il s'agit d'un tag et nous devons le spécifier particulièrement.
5. Cliquez sur *Build*.

> Vous vous souvenez que le build hello-world déclenche, après un succès, le job deploy-test-hello-world. Ce déclenchement est également valable lorsque le build est exécuté à la main. Ainsi, dans notre cas, nous pourrons constater qu'un nouveau déploiement sur l'environnement de test sera effectué (basé sur la branche master). Ce dernier n'aura toutefois aucune utilité, mais il n'est pas dérangeant.

Dès la fin de la construction du conteneur, nous pouvons l'utiliser grâce au job de déploiement de l'environnement de production. Ainsi, toujours dans Jenkins :

1. Cliquez sur le projet *deploy-prod-hello-world*.
2. Cliquez sur *Build with Parameters*.
3. Pour *appVersion*, saisissez « v1.0.0 ».
4. Cliquez sur *Build*.

Testons maintenant notre environnement de production :

```
http://docker-api:8491/hello?name=John
```

10.4.3 Correction d'une anomalie

À ce stade, tous les environnements sont alignés, dans le sens où ils ont tous exactement la même version du code. Afin de constater qu'il s'agit bien d'environnements différents, notamment au niveau du test et de la production, nous allons utiliser le prétexte d'une anomalie qui doit être corrigée : dans notre cas, il s'agira de la virgule entre le mot « hello » et le nom fourni en paramètre.

La vue doit ainsi être corrigée. Pour cela, éditons le fichier et supprimons la virgule :

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Getting Started: Serving Web Content</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
    <p style="color:red;" th:text="'Hello ' + ${name} + '!'" />
</body>
</html>
```

Testons la modification dans notre conteneur d'environnement de développement :

```
$ docker run --rm -p 8090:8090 --volumes-from devenv-volume --name devenv
devenv
```

Si nous accédons avec un navigateur à nos trois environnements, nous obtiendrons le résultat décrit dans le tableau ci-dessous (la couleur rouge n'est ici pas représentée) :

Environnement	URL	Résultat
Développement	http://localhost:8090/hello?name=John	Hello John!
Test	http://docker-api:8490/hello?name=John	Hello, John!
Production	http://docker-api:8491/hello?name=John	Hello, John!

Logiquement, seul l'environnement de développement est impacté.

Publions la modification :

```
$ cd /home/vagrant/volumes/devenv/hello-world/
$ git add src/main/resources/templates/hello.html
$ git commit -m"Suppression virgule"
$ git push origin master:master
```

Comme précédemment, l'environnement de test est automatiquement construit et déployé.

En accédant (après le déploiement) aux trois environnements avec un navigateur, cette fois-ci, les environnements de développement et de test sont impactés, et l'environnement de production reste inchangé :

Environnement	URL	Résultat
Développement	http://localhost:8090/hello?name=John	Hello John!
Test	http://docker-api:8490/hello?name=John	Hello John!
Production	http://docker-api:8491/hello?name=John	Hello, John!

Essayons maintenant de redéployer l'environnement de production avec la version précédemment créée (la v1.0.0) afin de constater que le résultat sera toujours le même. Dans Jenkins :

1. Cliquez sur le projet *deploy-prod-hello-world*.
2. Cliquez sur *Build with Parameters*.
3. Pour *appVersion*, saisissez « v1.0.0 ».
4. Cliquez sur *Build*.

En ouvrant un navigateur à l'adresse de production, la virgule est bien toujours présente. Il faudrait « releaser » le code (dans une version V1.0.1 pour suivre la convention de nommage) et déployer cette nouvelle version pour voir la correction appliquée.

En résumé

Dans ce chapitre, nous avons pu voir comment un système d'intégration continue peut être facilement mis en place grâce à Docker. Nous avons pour cela utilisé deux aspects :

– d'une part des images existantes (conçues par des tiers) contenant des outils directement fonctionnels (dans notre exemple, il s'agissait de GitLab et Jenkins) ;

– d'autre part une architecture de déploiement basée sur des conteneurs afin de pouvoir tester très rapidement une application et la mettre en production le cas échéant.

11

Docker Swarm

Clustering avec Docker

L'objectif de ce chapitre est de découvrir la solution de clustering implémentée dans l'écosystème de Docker Inc. : Docker Swarm. Nous découvrirons d'abord les différents composants nécessaires, puis créerons localement sur notre poste de travail un petit cluster. Finalement, nous déploierons quelques conteneurs, tout d'abord manuellement, puis avec Docker Compose.

À l'issue de ce chapitre, vous saurez déployer et utiliser un cluster Docker et comprendrez comment les différents conteneurs communiquent entre eux via le réseau overlay.

Nous n'avons pour l'instant utilisé Docker que sur un seul hôte, principalement notre poste de travail. Cependant, pour une utilisation productive de Docker, il est plus que probable que vous voudrez disposer de plusieurs nœuds, typiquement pour assurer une haute disponibilité de votre application. Ces différents nœuds devront de plus pouvoir héberger des conteneurs devant communiquer entre eux, ce que le réseau bridge ne peut réaliser ; c'est là que nous aurons besoin d'un nouveau type de réseau Docker : le réseau overlay.

Commençons par comprendre les différents éléments constituant un cluster Docker Swarm.

Swarm est un exemple de solution pour implémenter une architecture CaaS telle que nous l'avons décrite dans le chapitre 2. Le lecteur pourra s'y reporter pour découvrir d'autres solutions comme Kubernetes ou Mesos.

11.1 DOCKER SWARM

Docker Swarm permet de gérer centralement un ensemble d'hôtes disposant d'une instance de Docker Engine, le tout comme un hôte unique.

Il s'appuie pour cela sur :

- notre client Docker : il sera utilisé aussi bien pour gérer nos conteneurs que notre cluster Swarm ;
- un service de découverte (discovery service) qui permet de centraliser les informations de tous les hôtes (principalement la configuration réseau et l'état de chaque hôte) ;
- d'un maître Swarm (Swarm Master) et optionnellement de plusieurs réplicas dans le cas d'une architecture hautement disponible ;
- de plusieurs nœuds Swarm (Swarm Node).

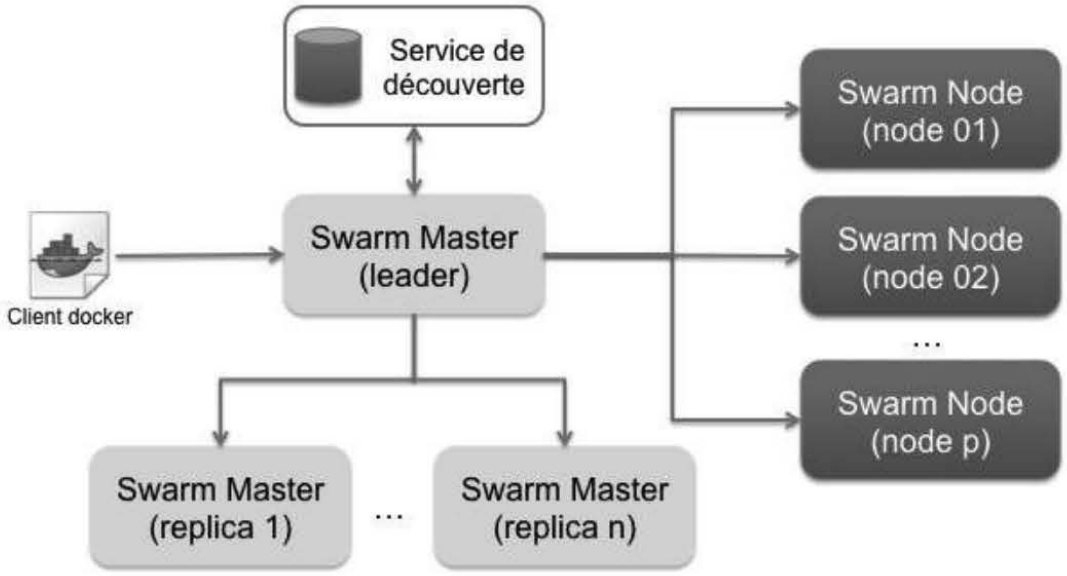

Figure 11.1 — Architecture générique d'un cluster Swarm

Docker Swarm permet même de gérer de manière transparente des hôtes localisés dans des centres de calculs différents, voire chez des fournisseurs cloud différents. Une réflexion poussée sur les aspects opérationnels de gestion et de performance est, dans ce cas, obligatoire, et nous ne pouvons que vous conseiller l'article sur le site de Docker qui couvre ce sujet plus en détail[1].

Détaillons un peu plus chacun de ces nouveaux composants.

1. https://docs.docker.com/swarm/plan-for-production/

11.1.1 Le service de découverte

Le service de découverte peut être considéré comme la CMDB[1] du cluster. Il fournit principalement :

- un registre des services disponibles ;
- un mécanisme pour pouvoir enregistrer de nouveaux services et surveiller leur fonctionnement ;
- un mécanisme pour pouvoir découvrir les services disponibles et, bien sûr, s'y connecter en fournissant tous les paramètres nécessaires.

Le registre des services est généralement implémenté à l'aide d'un système de persistance de type clé/valeur qui stocke aussi bien les informations sur les différents hôtes que les paramètres de configuration des applications.

Docker propose actuellement trois types de service de découverte[2] :

- le premier est basé sur un fichier listant uniquement les IP des différents hôtes. Comme vous l'imaginez, ce mécanisme est vite limitant et n'est conseillé que pour les configurations simples ;
- le Docker Hub. L'utilisation de ce service, qui nécessite que tous vos hôtes aient accès à Internet, est déconseillée par Docker pour une utilisation en production. Il repose sur un token (qui représente l'identifiant de votre cluster) qui est ensuite utilisé pour requêter le service de découverte du Docker Hub ;
- le dernier repose sur l'utilisation d'un système clé/valeur dédié. Docker utilise cela pour la bibliothèque libkv[3] qui permet d'abstraire l'utilisation d'un tel service et même d'en implémenter de nouveaux. Actuellement, libkv supporte Consul[4] (>= 0.5.1), etcd[5] (>= 2.0), Zookeeper[6] (>= 3.4.5) et BoltDB[7] (uniquement disponible localement).

Chacun dispose en général d'une interface de gestion REST qui permet aussi bien l'enregistrement, le retrait des services et l'envoi de requêtes de découverte.

11.1.2 Maître et nœuds Swarm

Quand on parle de nœud du cluster, cela signifie simplement un hôte sur lequel tourne le programme (binaire) Swarm. Ce programme dispose, comme nous l'avons vu dans le chapitre 2, de deux modes de fonctionnement :

1. *Configuration and Management Database*
2. https://docs.docker.com/swarm/discovery/
3. https://github.com/docker/libkv
4. https://www.consul.io/
5. https://coreos.com/etcd/
6. https://zookeeper.apache.org/
7. https://github.com/boltdb/bolt

- **mode maître** (*Swarm master*). Nous devons évidemment avoir au moins un master par cluster, qui, en tant que contrôleur, gère les différents nœuds du cluster et orchestre le déploiement des conteneurs sur les différents hôtes. Étant un point individuel de défaillance, il est recommandé de définir plusieurs réplicas, dont un deviendra le nouveau maître en cas de défaillance du premier[1] ;

Dans le cas d'une architecture multi-master, vous pouvez directement utiliser les réplicas pour faire parvenir des commandes à votre cluster ; celles-ci seront tout simplement transmises au maître pour leur exécution (les réplicas n'agissent finalement qu'en tant que proxies).

- **mode esclave**. Les nœuds Swarm sont simplement des hôtes sur lesquels nos conteneurs vont tourner.

Mettons maintenant cela en œuvre.

11.2 MISE EN ŒUVRE D'UN CLUSTER SWARM

Mettre en œuvre un cluster Swarm pour un environnement haute disponibilité, potentiellement réparti sur plusieurs réseaux privés différents, voire plusieurs centres d'hébergement, est clairement en dehors du périmètre de ce livre. Par contre, grâce à ce que nous avons appris sur l'utilisation de Docker Machine au chapitre 4, nous pouvons créer localement, sur notre poste de travail, un cluster minimal.

Ce cluster reposera sur quatre hôtes (soit quatre machines virtuelles VirtualBox) :

- **consul** : cet hôte hébergera notre registre de service qui sera déployé comme un conteneur Docker. La documentation de Consul recommande d'avoir entre trois et cinq serveurs Consul[2] par datacenter pour éviter toute perte de données et fournir un système de haute disponibilité. Dans notre cas, nous n'utiliserons qu'un seul serveur pour simplifier la mise en œuvre ;
- **swarm-master** : notre *Swarm Master* sera installé sur cette machine virtuelle. Il sera notre point d'entrée pour commander notre cluster (dans notre cas, il permettra également de faire tourner des conteneurs de la même manière que les nœuds esclaves) ;
- **swarm-node-01** et **swarm-node-02** : nous aurons deux nœuds dans notre cluster qui feront tourner nos conteneurs.

Nous allons créer ces quatre hôtes localement à l'aide de Docker Machine et de son driver Virtualbox.

Au final, nous aurons la configuration suivante :

1. https://docs.docker.com/swarm/multi-manager-setup/
2. https://www.consul.io/intro/getting-started/join.html

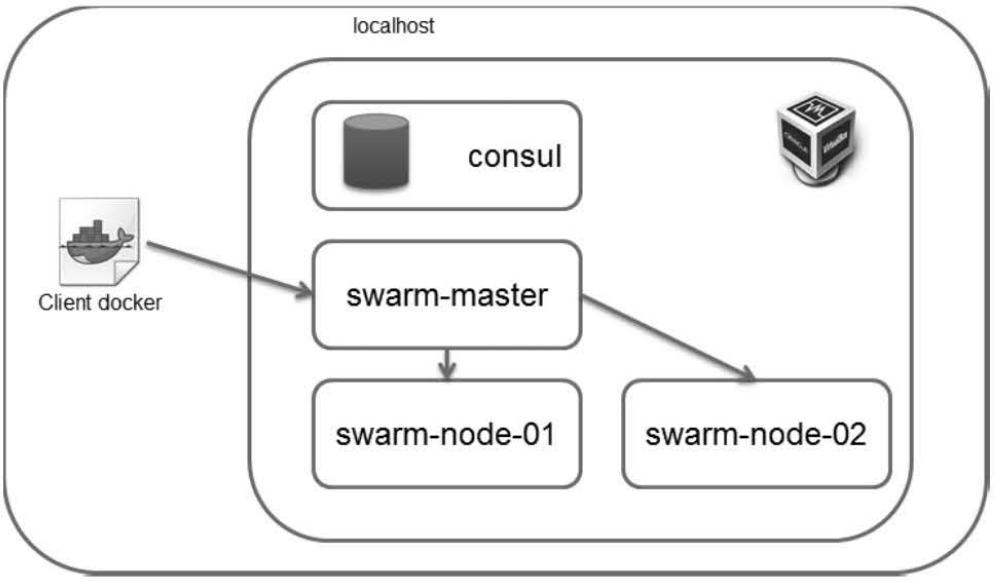

Figure 11.2 — Architecture de notre cluster Swarm

Commençons directement par notre machine virtuelle « consul ».

11.2.1 Service de registre : Consul

Consul est un système de découverte et de configuration des services. Il fournit en plus un système de *health check*, inclut une base de données clé/valeur et peut être utilisé dans une configuration avec plusieurs centres de calculs.

Il est basé sur un système distribué d'agents, chacun pouvant fonctionner :

- **soit en mode serveur** : dans ce cas, il stockera localement les informations dans sa base de données qui seront ensuite répliquées entre les différents agents serveurs. Ces différents serveurs formeront un cluster Consul (idéalement entre trois et cinq serveurs pour avoir une haute disponibilité). Dans le cas de plusieurs centres de données, il est conseillé que chacun dispose de son cluster Consul pour éviter une perte de données ;
- **soit en mode agent simple** : chaque hôte peut contenir un agent Consul, mais ce dernier ne servira qu'à surveiller les services de son hôte. Il n'est pas obligatoire d'en avoir un.

Dans notre configuration, nous n'aurons qu'un serveur Consul qui tournera sur un hôte dédié pour simplifier la mise en œuvre. Nous avons déjà vu au chapitre précédent la commande `docker-machine create`. Utilisons-la pour créer notre machine « consul » :

```
$ docker-machine create --driver=virtualbox consul
Running pre-create checks...
Creating machine...
(consul) Copying /Users/thomas/.docker/machine/cache/boot2docker.iso to
/Users/thomas/.docker/machine/machines/consul/boot2docker.iso...
```

```
(consul) Creating VirtualBox VM...
(consul) Creating SSH key...
(consul) Starting the VM...
(consul) Check network to re-create if needed...
(consul) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this
virtual machine, run: docker-machine env consul
```

Pointons notre environnement vers notre nouvelle machine et démarrons un conteneur Consul : nous utiliserons pour cela l'image progrium/consul[1] disponible sur le Docker Hub.

```
$ eval $(docker-machine env consul)
$ docker run -d -p 8500:8500 -p 8600:53/udp -h consul --name=consul
progrium/consul -server -bootstrap -ui-dir /ui
```

Quelques explications sur les différents paramètres de la commande précédente :

- nous exposons les deux ports qui permettent d'interagir avec notre registre, HTTP (port 8500) et DNS (8600). Nous verrons bientôt plus en détail comment ils sont utilisés ;
- le paramètre `server` permet d'indiquer à l'agent Consul de démarrer en mode serveur ;
- le paramètre `bootstrap` est requis par Consul pour indiquer à notre agent qu'il peut s'auto-élire comme leader Raft. Raft est le protocole de consensus utilisé en interne par Consul. Dans le cas d'une configuration à plusieurs agents Consul, il permet d'élire un leader et ensuite de s'assurer que tous les agents répondent de manière consistante aux requêtes qu'ils reçoivent[2] ;
- le paramètre `ui-dir /ui` : Consul embarque une interface web qui permet d'explorer le contenu du magasin clé/valeur. Ce paramètre est optionnel et nous permettra de comprendre plus simplement le fonctionnement de notre cluster. Il nous suffit pour cela de récupérer l'IP de notre machine et, avec un navigateur, d'aller à l'adresse http://<ip_machine>:8500/ui.

Pour récupérer l'adresse IP de la machine Docker hébergeant le conteneur « consul », il suffit d'exécuter la commande suivante : `docker-machine ip consul`.

1. https://hub.docker.com/r/progrium/consul/
2. https://www.consul.io/docs/internals/consensus.html

Nous pouvons d'ailleurs directement nous en servir pour voir que nous avons maintenant un hôte Consul qui fournit le service Consul sur le port 8300 (qui est le port RPC[1] utilisé pour la communication entre serveurs Consul).

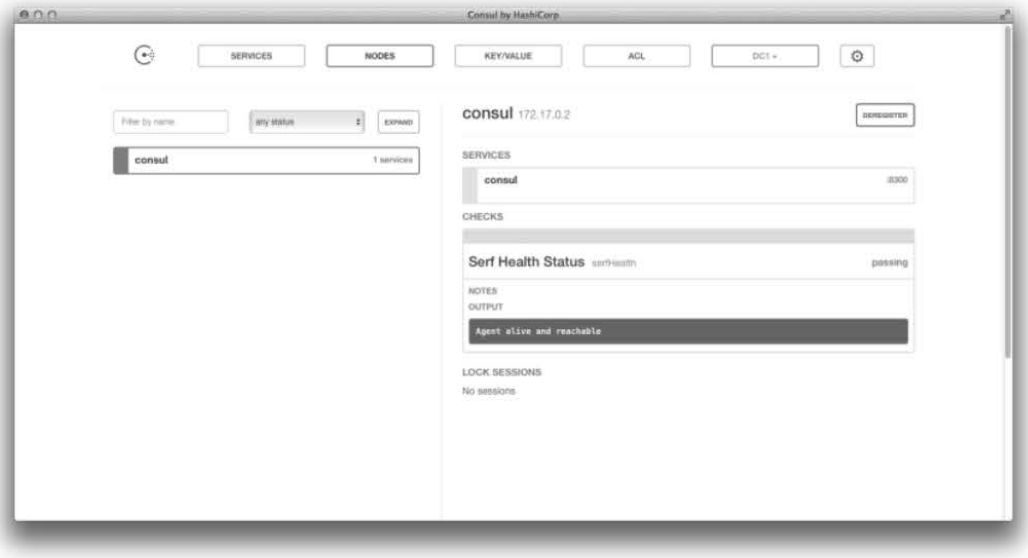

Figure 11.3 — Interface graphique de Consul

Consul dispose d'une documentation d'excellente qualité. Nous ne pouvons que vous encourager à la parcourir pour comprendre toutes les capacités et les rouages internes[2].

Prochaine étape : créons notre Swarm Master ainsi que les deux nœuds esclaves.

11.2.2 Maître et nœuds Swarm

Nous allons de nouveau utiliser Docker Machine pour créer nos différents hôtes. Commençons par notre Swarm Master :

```
$ docker-machine create --driver=virtualbox --swarm --swarm-master
--swarm-discovery="consul://$(docker-machine ip consul):8500"
--engine-opt="cluster-store=consul://$(docker-machine ip consul):8500"
--engine-opt="cluster-advertise=eth1:2376" swarm-master
```

Regardons un peu les différents paramètres :

- `--driver=virtualbox` : sans surprise, nous utilisons le driver VirtualBox ;
- `--swarm` : installe automatiquement Docker Swarm sur notre machine ;
- `--swarm-master` : configure Swarm comme étant un Swarm Master ;

1. Remote Procedure Call
2. https://www.consul.io/docs/index.html

- `--swarm-discovery="consul://$(docker-machine ip consul):8500"` : spécifie la localisation de notre service de découverte. Nous obtenons l'IP de notre machine Consul avec la commande `docker-machine ip consul` et précisons le port exposé HTTP (8500) ;
- `--engine-opt="cluster-store=consul://$(docker-machine ip consul):8500"` : ce paramètre permet de configurer le démon Docker qui sera installé sur notre machine. Cela permettra à notre hôte de s'enregistrer comme faisant partie de notre cluster ;
- `--engine-opt="cluster-advertise=eth1:2376"` : adresse à laquelle notre démon sera exposé et joignable sur notre cluster.

Si nous nous connectons sur notre machine, nous pouvons vérifier que la configuration s'est effectivement bien passée :

```
$ docker-machine ssh swarm-master
...
docker@swarm-master:~$ docker ps -a
CONTAINER ID    IMAGE            COMMAND                  CREATED
STATUS          PORTS            NAMES
e46d3a19e743    swarm:latest     "/swarm join --advert"   6 minutes ago
Up 6 minutes                            swarm-agent
5d13ed6ca0b4    swarm:latest     "/swarm manage --tlsv"   6 minutes ago
Up 6 minutes                            swarm-agent-master
```

Docker Machine a automatiquement instancié deux conteneurs Swarm (basés sur la même image qui contient principalement le binaire de Swarm[1]) :

- swarm-master-agent : comme vous l'avez deviné, c'est notre Swarm Master ;
- swarm-agent : c'est l'agent Swarm qui gérera les conteneurs de notre hôte et qui recevra les commandes du Swarm Master.

Nos deux conteneurs ne contiennent finalement que le binaire Swarm (et quelques certificats CA nécessaire au mode TLS). C'est une technique efficace pour packager un exécutable et profiter ainsi du Docker Hub comme moyen de distribution. Il est aussi possible d'installer manuellement le binaire de Docker Swarm, même si l'intérêt est limité, comme le reconnaît d'ailleurs Docker Inc[2].

Regardons plus en détail les processus docker/swarm qui tournent dans notre machine :

```
docker@swarm-master:~$ ps -ax
2636 ?        Sl     0:12 /usr/local/bin/docker daemon -D -g /var/lib/docker
-H unix:// -H tcp://0.0.0.0:2376 --label provider=virtualbox
--cluster-store=consul://192.168.99.102:8500 --cluster-advertise=eth1:2376
```

1. https://github.com/docker/swarm-library-image
2. https://docs.docker.com/swarm/get-swarm/

```
--tlsverify --tlscacert=/var/lib/boot2docker/ca.pem
--tlscert=/var/lib/boot2docker/server.pem
--tlskey=/var/lib/boot2docker/server-key.pem -s aufs
...
 2777 ?        Ssl    0:00 /swarm manage --tlsverify
--tlscacert=/var/lib/boot2docker/ca.pem
--tlscert=/var/lib/boot2docker/server.pem
--tlskey=/var/lib/boot2docker/server-key.pem -H tcp://0.0.0.0:3376 --strategy
spread --advertise 192.168.99.103:3376 consul://192.168.99.102:8500
...
 2790 ?        Ssl    0:00 /swarm join --advertise 192.168.99.103:2376
consul://192.168.99.102:8500
...
```

Nous retrouvons bien :

- notre démon Docker disponible sur le port 2376 (port TLS par défaut) et avec les options que nous avions passées à Docker Machine (`--cluster-store` et `--cluster-advertise`) ;
- notre Swarm Master (/swarm manage) qui sera disponible sur le port 3376. C'est par ce port que nous piloterons notre cluster depuis notre client Docker. Au passage, nous remarquons que notre Swarm Master utilise bien la stratégie par défaut « spread » ;
- notre agent Swarm (/swarm join) qui enregistre le démon Docker de notre hôte dans Consul et le rend disponible pour notre cluster. Nous pouvons d'ailleurs nous en rendre compte via l'interface de Consul sous KEY/VALUE, puis DOCKER/NODES qui liste les démons Docker disponibles.

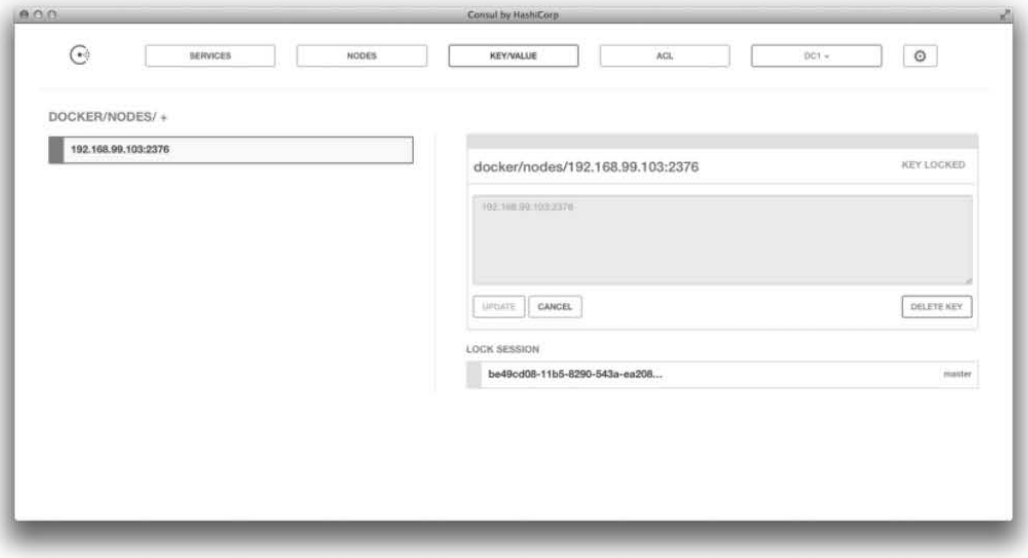

Figure 11.4 — Nœuds Dockers enregistrés par Consul

Il est conseillé sur un environnement de production de ne pas avoir d'agent Swarm non-master sur l'hôte qui héberge le Swarm Master, car cela évite d'avoir

des conteneurs qui se déploient sur ce serveur. Par défaut, Docker Machine en installe un, ce qui peut être considéré comme un bug.

Sur le même principe, il ne nous reste plus qu'à créer nos deux nœuds esclaves. L'unique différence est l'absence du paramètre `--swarm-master`, si bien que nous n'aurons qu'un agent Swarm sur ces hôtes (ce sont donc des nœuds esclaves).

```
$ docker-machine create --driver=virtualbox --swarm
--swarm-discovery="consul://$(docker-machine ip consul):8500"
--engine-opt="cluster-store=consul://$(docker-machine ip consul):8500"
--engine-opt="cluster-advertise=eth1:2376" swarm-node-01
...
$ docker-machine create --driver=virtualbox --swarm
--swarm-discovery="consul://$(docker-machine ip consul):8500"
--engine-opt="cluster-store=consul://$(docker-machine ip consul):8500"
--engine-opt="cluster-advertise=eth1:2376" swarm-node-02
...
```

Connectons maintenant notre client Docker sur notre Swarm Master. Pour cela, nous utilisons une légère variante de la commande que nous avons vue précédemment : nous rajoutons le paramètre `--swarm` qui nous permet d'interagir avec le Swarm Master (sur le port 3376). Si nous oublions ce paramètre, par défaut, notre client serait configuré pour utiliser le port 2376 qui est notre démon Docker. Dans ce cas, nous court-circuiterions Swarm et nous ne ferions que piloter un unique hôte.

On peut le voir facilement :

```
$ docker-machine env swarm-master
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.107:2376"
export DOCKER_CERT_PATH="/Users/thomas/.docker/machine/machines/swarm-master"
export DOCKER_MACHINE_NAME="swarm-master"
# Run this command to configure your shell:
# eval $(docker-machine env swarm-master)

$ docker-machine env --swarm swarm-master
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.107:3376"
export DOCKER_CERT_PATH="/Users/thomas/.docker/machine/machines/swarm-master"
export DOCKER_MACHINE_NAME="swarm-master"
# Run this command to configure your shell:
# eval $(docker-machine env --swarm swarm-master)
```

Listons maintenant tous les nœuds et conteneurs de notre cluster. Nous pouvons pour cela utiliser la commande `docker info` sur notre Swarm Master après avoir configuré les variables d'environnement pour notre client Docker.

```
$ eval $(docker-machine env --swarm swarm-master)
$ docker info
Containers: 4
 Running: 4
 Paused: 0
```

```
 Stopped: 0
Images: 3
Server Version: swarm/1.1.3
Role: primary
Strategy: spread
Filters: health, port, dependency, affinity, constraint
Nodes: 3
 swarm-master: 192.168.99.107:2376
  └ Status: Healthy
  └ Containers: 2
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.1.19-boot2docker,
operatingsystem=Boot2Docker 1.10.3 (TCL 6.4.1); master : 625117e - Thu Mar 10
22:09:02 UTC 2016, provider=virtualbox, storagedriver=aufs
  └ Error: (none)
  └ UpdatedAt: 2016-03-13T07:38:12Z
 swarm-node-01: 192.168.99.108:2376
  └ Status: Healthy
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.1.19-boot2docker,
operatingsystem=Boot2Docker 1.10.3 (TCL 6.4.1); master : 625117e - Thu Mar 10
22:09:02 UTC 2016, provider=virtualbox, storagedriver=aufs
  └ Error: (none)
  └ UpdatedAt: 2016-03-13T07:39:04Z
 swarm-node-02: 192.168.99.109:2376
  └ Status: Healthy
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.1.19-boot2docker,
operatingsystem=Boot2Docker 1.10.3 (TCL 6.4.1); master : 625117e - Thu Mar 10
22:09:02 UTC 2016, provider=virtualbox, storagedriver=aufs
  └ Error: (none)
  └ UpdatedAt: 2016-03-13T07:38:38Z
Plugins:
 Volume:
 Network:
Kernel Version: 4.1.19-boot2docker
Operating System: linux
Architecture: amd64
CPUs: 3
Total Memory: 3.064 GiB
Name: swarm-master
```

Nous retrouvons bien nos trois nœuds avec le nombre de conteneurs actifs sur chaque nœud : deux pour swarm-master (le Swarm Master et l'agent Swarm), un pour chacun de nos nœuds esclaves (uniquement l'agent Swarm).

Notre registre de service est totalement fonctionnel, et nos machines sont maintenant prêtes. Nous pouvons donc déployer nos premiers conteneurs.

11.3 DÉPLOYER UNE APPLICATION SUR UN CLUSTER SWARM

11.3.1 Prise en main rapide

Abordons cela étape par étape pour bien comprendre comment Swarm orchestre le déploiement de nos conteneurs. Commençons, par exemple, par déployer quatre conteneurs Nginx.

```
$ eval $(docker-machine env --swarm swarm-master)
#Nous utilisons notre Swarm Master
$ docker run -d -p 80 --name=nginx-1 nginx
...
$ docker run -d -p 80 --name=nginx-2 nginx
...
$ docker run -d -p 80 --name=nginx-3 nginx
...
$ docker run -d -p 80 --name=nginx-4 nginx
...
```

Vous remarquerez que les commandes `run` prennent plus de temps qu'à la normale. C'est que Docker doit, pour chaque nœud sur lequel le conteneur Nginx doit être instancié, télécharger l'image depuis le Docker Hub. En effet, les images ne sont pas partagées par les différents nœuds du cluster et doivent être disponibles dans le cache local de chaque hôte. Ainsi la première fois qu'un conteneur est démarré sur un nœud, l'image devra être téléchargée ; dans notre cas cela se produit pour les trois premiers conteneurs car nous avons en tout trois nœuds disponibles (deux nœuds esclaves et le nœud Swarm Master). Listons nos conteneurs :

```
$ docker ps --format "table {{.ID}} {{.Status}} {{.Ports}} {{.Names}}"
CONTAINER ID        STATUS              PORTS               NAMES
555b54781876Up About an hour 443/tcp, 192.168.99.109:32769->80/tcp
swarm-node-02/nginx-4
b144c4e25694Up About an hour 443/tcp, 192.168.99.108:32769->80/tcp
swarm-master/nginx-3
11bbade4a679Up About an hour 443/tcp, 192.168.99.109:32768->80/tcp
swarm-node-02/nginx-2
7d7f17a74c33Up About an hour 443/tcp, 192.168.99.101:32768->80/tcp
swarm-node-01/nginx-1
```

Nous avons bien nos quatre conteneurs, deux sur swarm-node-01, un sur swarm-node-02, et un sur swarm-master. Lorsqu'un Swarm Master reçoit une commande pour instancier un nouveau conteneur, il utilise deux mécanismes complémentaires pour sélectionner le nœud sur lequel le conteneur va être démarré : les filtres et la stratégie de déploiement.

Les filtres

Les filtres vous permettent de définir finement les règles de déploiement et de dire à Swarm où instancier le conteneur.

Il existe cinq filtres répartis en deux types :

- les premiers se basent sur les attributs du nœud, plus particulièrement sur les attributs du démon Docker directement. Ce sera soit des attributs standards (la version du noyau Linux, le système d'exploitation...), soit des étiquettes personnalisées définies au niveau du démon. Regardons quelques exemples utiles :

Commande à utiliser sur le Swarm Master	Commentaire
$ docker run -d -e constraint:node==swarm-node-01 –name=nginx-5 nginx	Déploie un conteneur nginx-5 sur le nœud avec le nom « swarm-node-01 ».
$ docker run -d -e constraint: operatingsystem==Boot2Docker* –name=nginx-6 nginx	Déploie le conteneur nginx-6 sur un nœud qui est basé sur une image Boot2Docker.
$ docker run -d -e constraint: storagedriver==devicemapper –name=nginx-7 nginx	Déploie le conteneur nginx-7 sur un nœud dont le démon Docker utilise le pilote de stockage devicemapper (référez-vous au chapitre 4 pour plus de détails sur ce sujet).
$ docker run -d -e constraint: type==front –name=nginx-8 nginx	Déploie le conteneur nginx-8 sur un nœud dont le démon Docker a été démarré avec l'étiquette « –label type="front" » comme paramètre.
$ docker run -d -p 80 :80 –name= nginx-9 nginx	Déploie le conteneur nginx-9 sur un nœud dont le port 80 est disponible. La création de ce conteneur échouera si aucun nœud n'a de port 80 disponible.

La liste des attributs utilisables dans les filtres s'obtient facilement avec la commande `docker info` et en regardant le champ Labels.

- les seconds types de filtres s'appliquent aux conteneurs eux-mêmes. Il est alors possible de regrouper des conteneurs ayant certaines affinités sur le même hôte ou partageant certaines dépendances. De nouveau, voici quelques exemples intéressants à garder en tête :

Commande à utiliser sur le Swarm Master	Commentaire
$ docker run -d -e affinity:container==nginx-4 –name=nginx-10 nginx	Déploie le conteneur nginx-10 sur le même nœud qu'un conteneur portant le nom « nginx-4 ».
$ docker run -d -e -e affinity:image==nginx –name= nginx-11 nginx	Déploie le conteneur nginx-11 sur un nœud qui dispose déjà de l'image Nginx localement.
$ docker run -d –volumes-from=volume_1 –name= nginx-12 nginx	Déploie le conteneur nginx-12 sur le même nœud que le conteneur volume_1 pour permettre l'accès aux volumes de ce dernier.

Attention, si Docker Swarm ne peut pas honorer l'affinité demandée, le conteneur ne sera pas instancié !

Les stratégies de déploiement

Si aucun filtre n'est défini, Docker Swarm se base sur une stratégie de distribution des conteneurs. Il existe trois stratégies possibles :

- **spread** : les conteneurs seront répartis de manière homogène sur les différents hôtes. Swarm sélectionne pour cela le nœud qui dispose du plus petit nombre de conteneurs, indépendamment du fait que les conteneurs présents soient arrêtés ou non ;
- **binpack** : au contraire de la stratégie spread, Docker Swarm essayera de regrouper un maximum de conteneurs sur un minimum de nœuds. Pour cela, Docker Swarm calcule un poids[1] pour chaque nœud en fonction d'un certain nombre de paramètres (CPU, RAM, du nombre de conteneurs déjà présents, santé du démon Docker...) et déploie le conteneur sur celui qui est le moins utilisé ;
- **random** : comme son nom l'indique, la répartition sera effectuée de manière aléatoire sur tous les nœuds du cluster.

Par défaut, Docker Swarm utilise la stratégie spread. Si nous reprenons nos commandes dans l'ordre et regardons après chaque commande combien de conteneurs nous avons sur chaque nœud, voici le résultat illustré dans un tableau :

Étape/commande	Nombre de conteneurs par nœud
Initialement	swarm-master : 2 swarm-node -01 : 1 swarm-node-02 : 1
$ docker run -d -p 80 –name=nginx-1 nginx	swarm-master : 2 swarm-node -01 : 2 (1 + nginx-1) swarm-node-02 : 1
$ docker run -d -p 80 –name=nginx-2 nginx	swarm-master : 2 swarm-node -01 : 2 swarm-node-02 : 2 (1 + nginx-2)
$ docker run -d -p 80 –name=nginx-3 nginx	swarm-master : 3 (2 + nginx-3) – N'oublions pas que nous avons un nœud esclave sur cet hôte ! swarm-node -01 : 2 swarm-node-02 : 2
$ docker run -d -p 80 –name=nginx-4 nginx	swarm-master : 3 swarm-node -01 : 3 (3 + nginx-4) swarm-node-02 : 2

Intéressons-nous maintenant à un cas où nos conteneurs doivent communiquer les uns avec les autres, tout en étant localisés sur des nœuds différents.

1. https://github.com/docker/swarm/blob/master/scheduler/strategy/weighted_node.go

11.3.2 Le réseau overlay

Nous avons déjà couvert en détail le fonctionnement du réseau Docker dans le cas d'un hôte (réseau de type bridge) au chapitre 9. Pour notre cluster, nous allons devoir utiliser un nouveau type de réseau : le réseau overlay. Ce réseau permet de faire communiquer de manière transparente des conteneurs qui sont localisés sur des nœuds différents.

Regardons avant tout les réseaux disponibles sur notre cluster :

```
$ eval $(docker-machine env --swarm swarm-master)
$ docker network ls
NETWORK ID          NAME                    DRIVER
55d03caed3e0        swarm-node-02/host      host
96b2d6913fbd        swarm-node-02/bridge    bridge
2c220eb04dc7        swarm-node-02/none      null
4422eb632120        swarm-master/none       null
66a56314efb0        swarm-master/bridge     bridge
53e82e1b3384        swarm-node-01/none      null
6cf9cb621ff6        swarm-master/host       host
78286be3ab1e        swarm-node-01/bridge    bridge
64a36a3c4672        swarm-node-01/host      host
```

Nous retrouvons pour chacun de nos nœuds, les trois réseaux standards de Docker : bridge, null et host. Pour que nos conteneurs sur des hôtes différents puissent communiquer entre eux, nous allons créer un nouveau réseau :

```
$ docker network create --driver overlay swarm-net
5e9bb5166be1dc85fb903234c8de74bb9a50c9162aef99d7a4479d6b1b2d2fb1
```

Nous pouvons effectivement vérifier que ce réseau est maintenant disponible sur tous les nœuds de notre cluster :

```
$ eval $(docker-machine env swarm-master)
$ docker network ls
NETWORK ID          NAME                DRIVER
5e9bb5166be1        swarm-net           overlay
4422eb632120        none                null
6cf9cb621ff6        host                host
66a56314efb0        bridge              bridge
$ eval $(docker-machine env swarm-node-01)
$ docker network ls
NETWORK ID          NAME                DRIVER
5e9bb5166be1        swarm-net           overlay
78286be3ab1e        bridge              bridge
53e82e1b3384        none                null
64a36a3c4672        host                host
```

Un réseau de type *overlay* nécessite pour fonctionner quelques prérequis qui ont été satisfaits automatiquement en construisant notre cluster :

- un registre de service : en effet, Docker l'utilise pour stocker des informations sur le réseau. Nous pouvons d'ailleurs nous en rendre compte via l'interface web

de Consul en allant dans Key/value, puis docker/network/v1.0. Nous retrouvons, par exemple, la liste de nos endpoints et de notre réseau overlay ;

- les différents hôtes doivent bien sûr avoir accès au service de registre ;
- les démons Docker doivent être configurés avec les options suivantes : `--cluster-store` et `--cluster-advertise`. Rappelez-vous que nous les avons passées en paramètre à Docker Machine pour configurer notre démon.

Notre réseau étant prêt, nous allons pouvoir maintenant déployer quelques conteneurs devant fonctionner de concert. Nous allons pour cela déployer un serveur Gitlab.

Au contraire de notre serveur Gitlab du chapitre consacré à l'intégration continue, celui que nous allons créer se basera sur l'image *sameersbn/gitlab* qui nécessite deux conteneurs additionnels : un pour la base de données (postgreSQL) et un pour le registre clé/valeur (redis). Nous utilisons ici cette image à des fins didactiques pour disposer facilement d'une application multi-conteneurs simple à déployer.

Commençons tout d'abord par notre conteneur de base de données :

```
$ eval $(docker-machine env --swarm swarm-master) #Nous utilisons notre Swarm
Master
$ docker run --name gitlab-postgresql -d --net=swarm-net --env
'DB_NAME=gitlabhq_production' --env 'DB_USER=gitlab' --env 'DB_PASS=password'
--env 'DB_EXTENSION=pg_trgm' sameersbn/postgresql:9.4-17
```

Quelques points importants à noter :

- nous attachons notre conteneur à notre réseau overlay swarm-net. Il sera ainsi atteignable par tous les autres conteneurs du réseau simplement par son nom d'hôte « gitlab-postgresql » ;
- nous passons un certain nombre de variables d'environnement nécessaires à cette image (sameersbn/postgresql:9.4-17[1]) pour son script entrypoint.sh, telles que le nom de l'utilisateur et le mot de passe pour la base de données.

Démarrons ensuite notre conteneur redis de la même manière, en n'oubliant pas de l'attacher aussi au réseau *swarm-net* :

```
$ docker run --name gitlab-redis -d --net=swarm-net sameersbn/redis:latest
```

Il ne nous reste plus qu'à instancier le conteneur contenant Gitlab lui-même :

```
$ docker run --name gitlab -d --net=swarm-net -p 10022:22 -p 10080:80 --env
'GITLAB_PORT=10080' --env 'GITLAB_SSH_PORT=10022' --env
'GITLAB_SECRETS_DB_KEY_BASE=cdsp5439850' --env 'DB_HOST=gitlab-postgresql' --env
```

1. https://hub.docker.com/r/sameersbn/postgresql/

```
'DB_PASS=password' --env 'DB_USER=gitlab' --env 'REDIS_HOST=gitlab-redis'
sameersbn/gitlab:8.6.1
```

De nouveau :

- nous avons attaché ce conteneur au réseau swarm-net ;
- nous avons défini les variables d'environnement nécessaires : DB_HOST et REDIS_HOST sont particulièrement intéressantes car ce sont les noms d'hôtes des deux conteneurs que nous avons créés précédemment. Ces variables sont remplacées dans les différents fichiers de configuration de Gitlab pour les noms d'hôtes de la base de données et du registre clé/valeur.

En listant nos conteneurs, nous voyons qu'ils ont été répartis par Docker Swarm sur les différents nœuds de notre cluster :

```
$ docker ps --format "table {{.ID}} {{.Ports}} {{.Names}}" -f name=gitlab*
CONTAINER ID        PORTS                NAMES
8520b0fefc96 6379/tcp swarm-node-01/gitlab-redis
01b465148718 5432/tcp swarm-node-02/gitlab-postgresql
333596179af1 443/tcp, 192.168.99.112:10022->22/tcp,
192.168.99.112:10080->80/tcp swarm-master/gitlab
```

Notre serveur Gitlab est disponible à l'adresse http://<ip_noeud_du_conteneur_gitlab>:10080, soit dans notre cas http://192.168.99.112:10080.

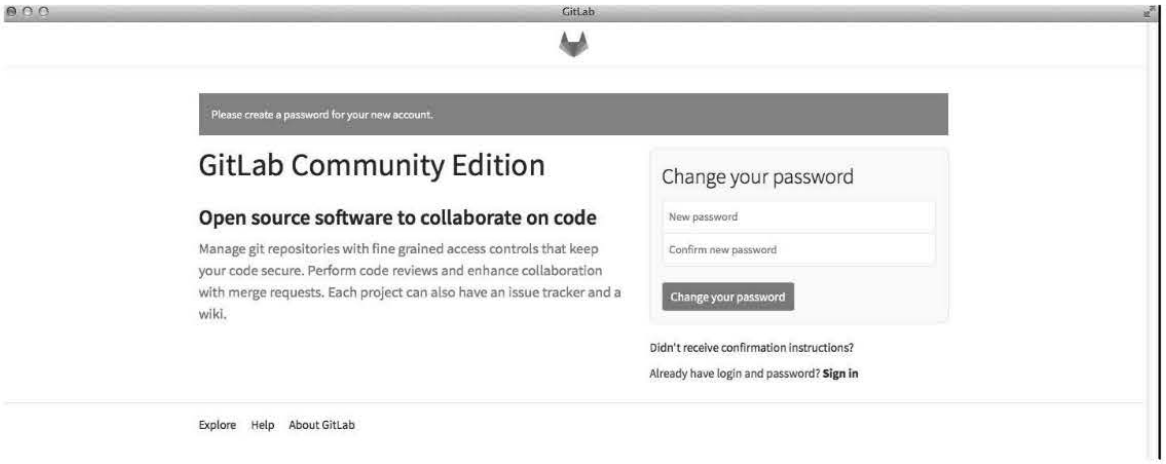

Figure 11.5 — Application GitLab

Nous pouvons aussi, bien sûr, utiliser Docker Compose pour définir une application multi-conteneurs répartie sur plusieurs hôtes. Cette approche est même, avec quelques limitations[1], totalement supportée par Docker Swarm.

1. https://docs.docker.com/compose/swarm/

Reprenons directement notre exemple ci-dessus et créons un fichier docker-compose.yml pour représenter notre application[1]. Reportez-vous au chapitre consacré aux applications multi-conteneurs pour plus de détails sur Docker Compose.

> Vous trouverez le fichier complet dans le dépôt Git :
> https://github.com/dunod-docker/swarm-compose.git

Plaçons-nous dans le répertoire où se trouve ce fichier et utilisons simplement la commande suivante :

```
$ docker-compose --project-name=gitlab up -d
Creating network "gitlab_default" with the default driver
Creating gitlab_redis_1
...
Creating gitlab_gitlab_1
...
Creating gitlab_postgresql_1
```

Que fait Docker Compose ?

- il crée tout d'abord un réseau dédié pour notre application de type overlay avec le nom « gitlab_default ». Nous n'avons plus besoin de le créer manuellement car Docker Compose s'en charge directement ;
- il crée ensuite nos trois conteneurs, les répartit ensuite sur les différents nœuds de notre cluster et les attache au réseau créé précédemment.

Nous venons donc de déployer notre applicatif sur plusieurs hôtes et de manière transparente.

En résumé

Dans ce chapitre, nous avons pu mettre en œuvre Swarm, le logiciel de clustering Docker à la base de l'offre CaaS de Docker. Nous avons vu comment instancier une architecture à plusieurs nœuds s'appuyant sur un service de découverte comme Consul. Nous nous sommes enfin attachés à montrer le potentiel des filtres et des stratégies de déploiement pour influencer la distribution des conteneurs sur les différents hôtes.

1. Ce fichier est basé sur celui proposé à https://hub.docker.com/r/sameersbn/gitlab/#quick-start. Nous l'avons juste légèrement adapté pour fonctionner avec la nouvelle version 2 du format de Docker Compose.

Conclusion : un potentiel en devenir

À travers les différents chapitres de ce livre, nous nous sommes attachés à donner une vue aussi complète que possible de ce qu'est aujourd'hui Docker et la technologie des conteneurs. En plus d'une présentation purement fonctionnelle, nous avons étudié des exemples plus complexes et plus opérationnels.

Le lecteur aura sans doute compris la portée de cette petite révolution. Nous n'en sommes qu'aux prémices de changements qui mettront probablement plus de dix ans à se faire sentir dans le quotidien des départements informatiques des entreprises.

Commençons par un bilan des domaines d'application que nous avons abordés précédemment.

LES DOMAINES D'APPLICATIONS EXISTANTS

Le tableau ci-dessous liste les domaines d'application de la technologie des conteneurs présentés dans ce livre :

Domaine d'application	Description / chapitre du livre
Distribution d'application	Nous l'avons abordé à diverses reprises dans ce livre (dès le **chapitre 1**). Le DockerHub est aujourd'hui le lieu de publication de nombreuses images de base par des organisations open source (citons Apache, Nginx, MySQL, etc.) mais aussi, plus marginalement, par des sociétés commerciales. De nombreux éditeurs commencent à considérer Docker et son registry public comme le centre de téléchargement officiel de services logiciels. Il reste encore à travailler le modèle de sécurisation de cette distribution, mais Docker Content Trust (présenté dans le **chapitre 8**) apporte les bases d'un mécanisme de signature fiable.
Conditionnement de modules applicatifs pour le support du cycle de développement	C'est probablement l'usage le plus commun de Docker aujourd'hui. Docker, au même titre que Vagrant (que nous avons présenté dans le **chapitre 2**), permet de conditionner des environnements de développement ou d'intégration pouvant être détruits et reconstruits à loisir. Le **chapitre 10** présente d'ailleurs un exemple complet de chaîne d'intégration continue. Docker offre enfin un mode de conditionnement des logiciels facilitant le transfert en exploitation (comme nous l'avons expliqué dans le **chapitre 1**).
Construction d'architectures à micro-services	C'est le propos de solutions CaaS (pour *Container As A Service*) qui sont décrites dans le **chapitre 2** puis, à travers un exemple complet, dans le **chapitre 9** et le **chapitre 11**. Il s'agit sans aucun doute d'une nouvelle manière de considérer le développement d'applications. Néanmoins, celle-ci mettra du temps à s'installer dans les entreprises. La rupture que cette approche impose est difficilement compatible avec les existants plus classiques. À l'inverse, pour toute entreprise d'édition de logiciel SaaS ou encore pour une startup Internet, l'architecture à micro-service devrait être une option privilégiée. Notons enfin que la solution PaaS OpenShift (https://www.openshift.com/) de RedHat s'appuie solidement sur Docker et Kubernetes.

Domaines d'application actuels de Docker

Les applications décrites ci-dessus sont déjà une réalité. Les offres CaaS d'Amazon, Google ou Microsoft, ou encore l'architecture distribuée de Spotify (l'un des utilisateurs emblématiques de Docker) ne sont plus des prototypes.

Mais nous pensons que d'autres applications de Docker peuvent encore être développées.

DE NOUVELLES APPLICATIONS POUR LES CONTENEURS

Bien que ces usages ne soient encore que balbutiants (ou encore en maturation), nous pouvons déjà entrevoir d'autres usages possibles pour Docker.

Domaine d'application	Description
Bus d'intégration	La plupart des ESB sont aujourd'hui construits sur la base d'une logique de middleware relativement centralisée. Dans la pratique, il s'agit de moteurs de workflow plus ou moins complexes associés à des *message queues* (pour l'asynchronisme) et à des connecteurs. Ce type d'architecture se prend parfois un peu les pieds dans le tapis en imposant des modèles complexes de distribution de charge. Par ailleurs certaines intégrations nécessitent des configurations OS spécifiques qui sont souvent difficile à répliquer (et nécessitent parfois de dédier une machine pour un connecteur). Des initiatives comme Dray (http://dray.it/) proposent d'implémenter des workflows sur la base de processus encapsulés dans des conteneurs indépendants. L'avantage de ce type d'approche est d'offrir une isolation forte de chaque étape du workflow (y compris pour ce qui concerne les ressources système) tout en permettant d'implémenter des configurations OS très spécifiques pour des connecteurs exotiques.
Déploiement d'application sur le poste client	Docker est déjà un moyen de distribution d'application sur le serveur. Kitematic (que nous avons abordé dans le **chapitre 3**) offre une interface graphique pour lancer des applications Docker sur son poste client Windows ou Mac OSX. Docker pourrait-il devenir un moyen alternatif aux *package managers* et autres *Windows installers* pour déployer des applications sur le poste client ? Les deux prérequis sont de disposer d'une version native de Docker pour ces deux OS desktop, ce qui est pratiquement le cas. Le second prérequis sera de disposer d'images natives pour ces OS capables d'exposer une interface graphique sur l'hôte.
Documentation automatique d'architecture	Comme nous l'avons vu dans le **chapitre 8**, il est possible d'ajouter de la méta-information aux images Docker par l'intermédiaire de l'instruction LABEL. Par ailleurs, la commande `docker inspect` permet aussi de visualiser sous une forme JSON de nombreuses informations sur une image ou un conteneur. L'exploitation de cette méta-information est à la base des interfaces graphiques (pour le moment sommaires) des solutions CaaS. Pour peu qu'un standard de méta-information pour les images Docker émerge et que le pattern « micro-services » soit implémenté, on pourrait envisager des architectures auto-descriptives. La documentation (les dépendances, les services exposés) serait alors contenue, non plus dans des documents, mais dans l'environnement d'exécution lui-même.

Autres applications potentielles pour Docker

LES DÉFAUTS DE JEUNESSE DE DOCKER

Aussi intéressant que puisse être Docker, il faut reconnaître qu'il s'agit encore d'une solution présentant des défauts de maturité. Ceux-ci se corrigent peu à peu, mais certains sont encore difficilement acceptables pour un exploitant prudent.

Sous CentOS, par exemple, un `yum update` de Docker provoque l'arrêt du Docker engine et de tous ses conteneurs[1]. Dans certaines conditions, il arrive que les montages de répertoires du host donnent, à l'intérieur du conteneur, une image incorrecte de l'état des fichiers.

En dépit de ces défauts (bien référencés), l'usage de Docker en production s'accroît chaque jour. Les éditeurs qui adoptent les conteneurs (et Docker en particulier) sont nombreux et motivés. On ne peut donc douter que les avantages surpassent déjà les inconvénients qui, par ailleurs, sont souvent surmontables pour peu que l'on s'appuie sur une architecture adaptée.

Nous sommes convaincus que Docker Inc. a semé en quelques années les germes d'un changement profond. Cette petite startup française devenue l'une des valeurs montantes de la Silicon Valley saura-elle en retirer les fruits ?

Difficile d'en être certain. Le monde IT est très cruel avec les nouveaux entrants dans le domaine des logiciels d'infrastructure, mais il ne fait aucun doute que des géants comme Google, Microsoft ou Amazon ont déjà su tirer parti des avantages technologiques des conteneurs.

Que Docker soit connu dans quelques années comme le nom d'un projet open source ou comme une société profitable du NASDAQ est difficile à prédire, mais il est certain que nous utiliserons (peut-être sans le savoir) de plus en plus de conteneurs !

1. https://github.com/docker/docker/issues/2658

Index

Les commandes apparaissent dans l'index en police à espacement fixe. Les commandes Docker sont en `minuscules` et les commandes Dockerfile sont en `MAJUSCULES`.

D

E

F

R

S

T

U

V

W