



# Docker

Formateur : Ludovic  
Quenec'Hdu



# Bienvenue !

# Formation mixte présentiel et distanciel



# L'OFFRE DE FORMATION PLB

- Près de 25 ans d'existence et plus de 2 000 formations au catalogue !
- PLB dispose d'un catalogue de formation qui répond à la quasi-totalité **des besoins de formation IT des entreprises.**
- Nous proposons des formations 100% développées par PLB

**PLB est Organisme ATO (Accredited Training Organisation) et Centre d'examen**



**PLB est partenaires de centres ATO**



- Notre offre est classique et digitale dans ses modalités : présentiel, distanciel, blended, @learning, MOOC...

# Rechercher une autre formation ?

■ [www.plb.fr](http://www.plb.fr)

The screenshot shows a navigation bar with 'Formations', 'Infos pratiques', and 'Financement'. Below it is a section titled 'Domaines de formation' with two columns of links:

Domaine	Exemples
Bases de données	IBM
Big Data - BI	Multimédia
Microsoft Système	Gestion de projet
Microsoft Application	Management des SI
.NET	Sécurité
Java - Java EE	Télécom - Mobile
Développement	Réseaux
Virtualisation - Cloud - DevOps	Développement Personnel
Cisco	Relation client

A blue button labeled 'Voir toutes les formations' is at the bottom.

The homepage features a large banner with a man in glasses and the text '+ de 2000 formations Informatique et Management EN PRÉSENTIEL, À DISTANCE, BLENDED, E-LEARNING Qu'allez-vous choisir ?' Below it is a search bar with 'Intitulé, mot-clé, référence...' and a magnifying glass icon. A blue callout box points to the search bar with the text 'Barre de recherche par mot clef, ref ...'.

The 'Top formations' section displays four cards:

- Angular : Développer une application Web** (3 hrs, Niveau : Intermédiaire)
- Préparer la certification DevOps Foundation** (2 hrs, Niveau : Intermédiaire, Certification : DevOps Foundation, Cours officiel : DevOps Institute)
- Scrum Master - Niveau 1 (PSM1)** (2 hrs, Niveau : Fondamental, Certification : Professional Scrum Master, Niveau 1, Éligible CPF : Oui)
- Power BI - Initiation** (3 hrs, Niveau : Fondamental, Éligible CPF : Oui)



# Quelques chiffres



**2000**

FORMATIONS INFORMATIQUE  
ET MANAGEMENT DISPONIBLES  
À CE JOUR



**98 %**

DE NOS COURS SONT  
RÉALISABLES À DISTANCE



**8000**

STAGIAIRES DANS NOS SALLES  
DE COURS PAR AN



**94 %**

DE PARTICIPANTS SATISFAITS  
OU TRÈS SATISFAITS  
DE NOS FORMATIONS



**90 %**

DE RÉUSSITE AUX EXAMENS  
DE CERTIFICATION



**800**

FORMATEURS EXPERTS  
VALIDÉS PLB



**500**

FORMATIONS ÉLIGIBLES  
AUX DIFFÉRENTS LEVIERS  
DE FINANCEMENT (CPF, AIF...)

# Déroulement de votre formation

## ▪ Horaires

- 9h /12h30 ○ Pause déjeuner 1h30 ○ 14h / 17h30
- Pause : matin ( $\approx$ 11h) + après midi ( $\approx$ 15h30)

## ▪ Présentation

- Votre formateur
- Tour de table : votre profil, vos attentes, vos besoins

## ▪ Détail du programme

## ▪ Les objectifs de votre formation

## ▪ Plateforme formation PLB

- Signatures matin + après midi
- Avant de commencer : auto - évaluez vos connaissances
- Evaluation en fin de formation



## VOS CONTACTS CHEZ PLB

**Equipe IT** [assistance@plb.fr](mailto:assistance@plb.fr) 01 43 34 34 10  
pour toute question technique (VM, Teams, login, etc.)

**Référent stagiaire** [rs@plb.fr](mailto:rs@plb.fr)  
pour tout autre sujet ... et vous accompagner durant votre formation

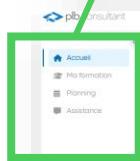
**Service qualité** [qualite@plb.fr](mailto:qualite@plb.fr)  
Pour toute question ou remarque liée à votre satisfaction

# Votre espace formation PLB

- Vous avez reçu le lien pour créer votre identifiant et accéder à votre espace en ligne

## Le menu de votre espace

PLB



Accès à votre espace formation

Ici les tâches à effectuer  
Signature, auto-évaluation, évaluation

A screenshot of the PLB consultant website showing the 'Tâches à effectuer' (Tasks to perform) section. It includes options for 'Signer', 'Auto-évaluer', and 'Évaluer la formation'. A green box highlights the 'Signer' button.

A screenshot of the PLB consultant website's footer area. It features a map of Paris with various locations marked, followed by several blue cards with text and icons. At the bottom, there is a navigation bar with links for 'CGV', 'Règlement intérieur', and 'Assistant'. A green box highlights the 'Assistant' link.

Vous trouverez également le lien vers le règlement intérieur et toutes les infos pratiques<sup>7</sup> si votre formation est en présentiel

# Auto-évaluez vos connaissances avant / après la formation sur le LMS

## ■ Objectif de ces auto-évaluation individuelles

- Permettre à votre formateur de mieux connaître votre niveau de connaissance avant de commencer
- Valider la qualité de votre progression en fin de formation

Pour chaque objectif pédagogique de votre formation, **choisissez votre niveau entre 0 et 9 avant la formation**

Vous pourrez remplir votre auto-évaluation à la fin de la formation

Ressources Formation à distance Travaux pratiques Auto-évaluation Enseignement Évaluation

L'auto-évaluation vous permet de mesurer votre progression tout au long de la formation. Avant de commencer, vous attribuez une note de 0 à 9 à chaque objectif de la formation. Une fois la formation terminée, vous réévaluez les mêmes objectifs.

Les modifications sont automatiquement enregistrées.

**Avant la formation**

- Automatiser des tests fonctionnels dans de multiples environnements techniques  
0 1 2 3 4 **5** 6 7 8 9
- Perfectionner le script de test en plaçant des points de synchronisation et de contrôle  
0 1 2 3 4 **5** 6 7 8 9
- Paramétrier le script de test avec des jeux de données  
0 1 2 3 4 **5** 6 7 8 9

**Après la formation**

- Automatiser des tests fonctionnels dans de multiples environnements techniques  
0 1 2 3 4 5 6 7 **8** 9
- Perfectionner le script de test en plaçant des points de synchronisation et de contrôle  
0 1 2 3 4 5 6 7 **8** 9
- Paramétrier le script de test avec des jeux de données  
0 1 2 3 4 5 6 7 **8** 9

**Signer**

\* L'auto-évaluation ne sera plus modifiable une fois signée.

# Signatures, documents, évaluations sur le LMS

- Vous avez reçu le lien pour créer votre identifiant et accéder à votre espace en ligne

*Les signatures sont obligatoires pour le suivi de votre session de formation*

Vous pouvez signer chaque matin + après midi

The screenshot shows a LMS interface for a course titled "BMBI - Maîtriser Power BI". It displays a daily sign-in section for February 12, 2024. The interface includes a sidebar with links like Accueil, Ma formation, Planning, and Assistance. The main content area shows course details: "Du 12/02/2024 au 17/05/2024 (5 jours)" and "animée par Céline ARMANDES dans nos locaux". Below this are tabs for Ressources, Formation à distance, Travaux pratiques, Auto-évaluation, Émargement (which is highlighted with a green box), and Évaluation. A note below the tabs states: "Pour attester de votre présence quotidienne, il est nécessaire de signer chaque demi-journée. Les créneaux de signature seront ouverts au tout moment, tant que la formation n'est pas terminée." Below the note, a message says: "Vous pouvez signer une journée à partir du moment où celle-ci a débuté". The daily schedule shows two slots: "Lundi 12/02/2024" with "Matin" and "Après-midi" and "Mardi 13/02/2024" with "Matin" and "Après-midi", each accompanied by a handwritten signature icon.

Vous pourrez remplir votre évaluation le dernier jour

The screenshot shows a LMS interface for a course titled "HPUFT - Micro Focus UFT 12 prise en main". It displays an evaluation section for the last day of the course. The interface includes a sidebar with links like Accueil, Ma formation, Planning, and Assistance. The main content area shows course details: "Du 14/02/2024 au 15/02/2024 (2 jours)" and "animée par Daniel HALWANI dans nos locaux". Below this are tabs for Ressources, Formation à distance, Travaux pratiques, Auto-évaluation, Émargement, and Évaluation (which is highlighted with a green box). A note below the tabs states: "Votre évaluation est essentielle pour recueillir votre avis sur le contenu de la formation, le formateur, l'environnement de formation, etc. Nous vous remercions de bien vouloir nous faire part de vos impressions." Below the note, a message says: "Vous êtes globalement". A row of radio buttons follows: "Très satisfait", "Satisfait", "Moyen", "Peu satisfait", and "Insatisfait". The evaluation section is divided into sections: "1. Le contenu du stage", "2. La formation", "3. Le formateur", and "4. L'environnement de formation". Each section contains questions and a series of radio buttons for responses ranging from "Oui" to "Excellent" or "Mauvais".

## Table des matières

### Avant-propos

### PREMIÈRE PARTIE Les conteneurs : principes, objectifs et solutions

#### 1 Les conteneurs et le cas Docker

1.1 La conteneurisation

1.2 Les fondations : Linux, cgroups et namespaces

1.3 Les apports de Docker : structure en couches, images, volumes et registry

1.4 Les outils de l'écosystème des conteneurs : Docker et les autres

#### 2 Orchestration de conteneurs

2.1 Automatiser la gestion de l'infrastructure : du IaaS au CaaS

2.2 Les solutions CaaS

2.3 Ansible, chef et puppet : objet et lien avec Docker et CaaS

### DEUXIÈME PARTIE Docker en pratique : les outils de base

#### 3 Prise en main

3.1 Installation des exemples du livre

3.2 Installation de Docker

3.3 Votre premier conteneur

#### 4 Conteneurs et images

4.1 Le cycle de vie du conteneur

4.2 Accéder au conteneur et modifier ses données

4.3 Construire une image Docker originale

4.4 Le Dockerfile

### TROISIÈME PARTIE Apprendre Docker

#### 5 Prise en main du client Docker

5.1 Introduction à la CLI Docker

5.2 Les commandes système

5.3 Cycle de vie des conteneurs

5.4 Interactions avec un conteneur démarré

[5.5 Commandes relatives aux images](#)

[5.6 Interactions avec le registry](#)

[5.7 Réseau et volumes](#)

## **[6 Les instructions Dockerfile](#)**

[6.1 Les modèles d'instruction](#)

[6.2 Les instructions d'un Dockerfile](#)

[6.3 Bonnes pratiques](#)

## **[7 Docker avancé](#)**

[7.1 Variables d'environnement et conteneurs : ENV](#)

[7.2 Méta-information et images : LABEL](#)

[7.3 Paramétriser le \*build\* d'une image](#)

[7.4 Modifier le contexte système au cours du \*build\*](#)

[7.5 Auto-guérison \(\*self healing\*\)](#)

## **QUATRIÈME PARTIE Développer, déployer et opérer avec Docker**

### **[8 « Real-life » Docker : Mettre en place une application complète](#)**

[8.1 Notre application exemple](#)

[8.2 Le réseau avec Docker](#)

[8.3 Persistance : \*bind mounts\* et volumes](#)

[8.4 Configuration d'application](#)

[8.5 Monitoring](#)

### **[9 Conditionnement et déploiement](#)**

[9.1 Build / run : principes](#)

[9.2 Option 1 : un seul conteneur, plusieurs processus](#)

[9.3 Option 2 : application multi-conteneurs](#)

[9.4 Option 3 : orchestration avec Compose](#)

### **[10 Intégration continue avec Docker](#)**

[10.1 Avant de commencer](#)

[10.2 Un environnement de \*build\* lui-même dockerisé](#)

[10.3 Installation des outils et chargement du code source](#)

[10.4 Image et job de \*build\*](#)

[10.5 Lancement automatique](#)

[10.6 Extensions et améliorations](#)

## **CINQUIÈME PARTIE Orchestration de conteneurs**

### **11 Docker Swarm : clustering avec Docker**

[11.1 Docker Swarm](#)

[11.2 Premier service et stack](#)

[11.3 Gestions des configurations et des secrets](#)

[11.4 L'avenir de Docker Swarm](#)

### **12 Kubernetes : clustering avancé**

[12.1 Environnement](#)

[12.2 Prise en main](#)

[12.3 Découverte des fonctionnalités](#)

[12.4 Déploiement de l'application exemple](#)

### **Conclusion : un potentiel en devenir**

[Les domaines d'applications existants](#)

[De nouvelles applications pour les conteneurs](#)

[Les défauts de jeunesse de Docker](#)

[Index](#)

## 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.

### ***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'applications sur la base d'une image de VM (pour *Virtual Machine*) est lourd (plusieurs gigaoctets, y compris pour les OS les plus compacts).

### ***o 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).



Nous verrons dans le chapitre 1 que l'implémentation Windows de Docker autorise l'exécution de conteneurs Windows et Linux, ce qui contredit l'affirmation précédente. Néanmoins Microsoft fait usage d'une solution un peu particulière qui, quoique fonctionnelle, s'éloigne du concept usuel de conteneur tel qu'il a été imaginé à l'origine.

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 du fameux « Dockerfile », de décrire très simplement la configuration et la construction d'un conteneur.

### ***o 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 : Docker Compose, Kubernetes (et ses nombreuses implémentations chez les leaders du cloud public), ou encore Docker 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 que celle qui prévaut aujourd'hui dans le domaine de la virtualisation matérielle.

Notre objectif dans cet ouvrage est donc multiple :

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

#### *o 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 parlerons d'outils tels que Kubernetes, Swarm, Mesos ou Azure Service Fabric. 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 virtualisé. Nous commencerons alors à « 0 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 quatrième partie du livre se consacre à la mise en œuvre des concepts appris précédemment autour d'exemples pratiques et réalistes. Nous étudierons ainsi le développement, le conditionnement et le déploiement d'une architecture à base de conteneurs en nous limitant au cas mono-hôte.

La cinquième et dernière partie est consacrée aux solutions d'orchestration de conteneurs que sont Swarm et Kubernetes. Nous montrerons comment l'application étudiée dans la quatrième partie peut être déployée sur plusieurs hôtes sans modification majeure des conteneurs.

#### **5888 À qui s'adresse ce livre**

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

- 23 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 ;
- 24 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 :

- 5888 il permet d'accélérer la prise en main de cette technologie en présentant des cas d'usages illustrés par des exemples didactiques ;
- 5889 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 dépôt public :  
<https://github.com/dunod-docker/docker-examples-edition2/>

Le dépôt contient, outre les exemples de code, toutes les commandes chapitre par chapitre (permettant un copier/coller), l'ensemble des liens vers les sites externes ainsi qu'un errata et des articles complémentaires.

**Veuillez vous reporter à la procédure d'installation du chapitre 3.**

## **PREMIÈRE PARTIE**

### **Les conteneurs : principes, objectifs et solutions**

Cette première partie vise à présenter :

- 23 les origines du concept de conteneur ;
- 24 l'apport de Docker à cette technologie déjà ancienne ;
- 25 comment les conteneurs autorisent la réalisation de nouvelles architectures informatiques ;
- 26 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<sup>1</sup>) aux plus classiques pour lesquelles les conteneurs apportent les avantages décrits dans le chapitre 1.

---

<sup>1</sup>. 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

### Objectif

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.

5888 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.

### 5889 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.



#### Docker et les conteneurs

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, Mac OS X et Windows). 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.

#### 1.1.1 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<sup>e</sup> 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 1930, 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.

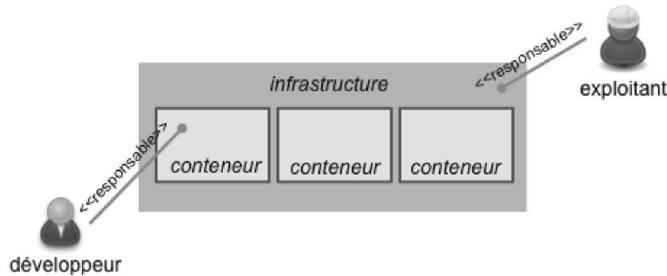
Cette séparation des responsabilités est à l'origine de la baisse drastique des coûts du transport de marchandises. Grâce à la standardisation et à l'automatisation qu'elle rend possible, les infrastructures de transport se sont mécanisées, automatisées et donc fiabilisées.

### 1.1.3 Extraposition 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 base de conteneurs



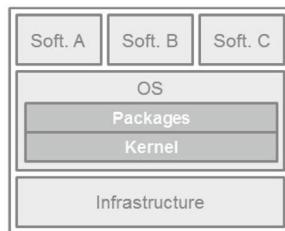
Dans une architecture à base de conteneurs :

- 23 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 ;
- 24 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 :

5888 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 ;

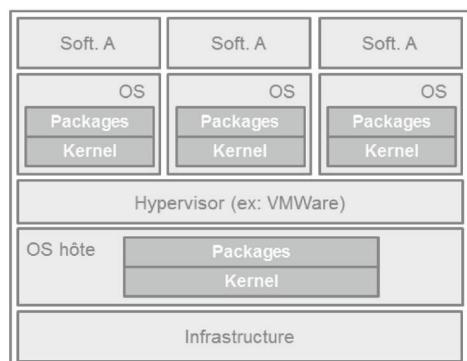
5889 toute mise à jour de l'OS hôte va nécessairement impacter tous les logiciels qui tournent dessus ;

5890 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

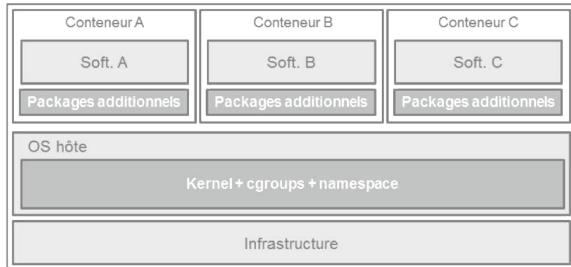


La réponse est « oui, mais... ».

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 :

- 0 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 ;
- 1 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.

Figure 1.5 – Architecture à base de conteneurs



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<sup>1</sup>. 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 base de 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 (*Virtual Machine*), 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 les conteneurs HyperV, tout en conservant aussi la possibilité de faire tourner des conteneurs nativement sur l'OS hôte. C'est également l'approche du projet Kata Containers<sup>2</sup> ou du projet HyperKit<sup>3</sup>.

Il existe par ailleurs des projets 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 en verrons quelques exemples plus loin dans ce chapitre.



Cette idée de simplifier au maximum le système d'exploitation en éliminant les services non utilisés est aussi appliquée dans une technologie technique différente des conteneurs mais cherchant à atteindre les mêmes objectifs : Unikernel. Le lecteur intéressé pourra notamment regarder le projet <https://github.com/solo-io/unik>, initialement développé par EMC avant d'être confié à la communauté.

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 clé 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 ensuite développé sa propre bibliothèque de bas niveau nommée libcontainer<sup>4</sup> pour enfin migrer vers runC, le standard de l'OCI (Open Container Initiative), dont nous parlerons plus tard. Ce composant encapsule les fonctionnalités fondamentales, proches du noyau du système d'exploitation, dont la combinaison permet la virtualisation de niveau OS. Aujourd'hui le moteur Docker est construit au-dessus de containerd<sup>5</sup> qui lui-même intègre runC. Pas d'inquiétude néanmoins, la maîtrise de cet empilement de projets et de composants n'est pas fondamentalement utile pour tirer parti de Docker.

Nous allons étudier les différents composants historiques et fondamentaux sur lesquels s'est construite la technologie des conteneurs pour :

- 0 en comprendre le rôle fonctionnel ;
- 1 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 initialement une technologie liée à Linux, son noyau (*Linux Kernel*) et certains de ses services orchestrés par containerd.

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

Eh bien, initialement il ne marchait pas réellement sous ces OS, du moins pas nativement.

Pour faire tourner Docker sous Windows, il était, encore jusqu'à récemment, nécessaire d'y installer un OS Linux. La solution usuellement pratiquée consistait à s'appuyer sur un logiciel de virtualisation matérielle : [VirtualBox<sup>6</sup>](#) dans le cas de Docker (Docker Toolbox).



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.

L'installation n'était donc pas native et la complexité de gestion des terminaux, comme les performances d'accès au système de fichiers, limitait très fortement l'intérêt de Docker sous ces OS Desktop.

Mais les choses ont changé en 2016.

### 1.2.2 Docker = Linux + Windows + Mac

En 2016, Microsoft et Docker ont annoncé que la version éponyme de Windows Server inclurait la possibilité d'exécuter des conteneurs nativement. Les équipes de Microsoft, avec le support des ingénieurs de Docker, ont pour ce faire porté runC en s'appuyant sur les capacités natives de Windows en lieu et place de Linux. Cette fonctionnalité a ensuite été étendue à Windows 10 desktop pro 64 bits avec pour objectif de cibler les développeurs.

Microsoft propose aussi des images de base pour ces conteneurs « natifs » Windows (Windows Server Core et la très compacte Windows Nano Server).

Question brûlante : est-il possible d'exécuter un conteneur Linux sous Windows ?

Oui mais via un « truc » déjà évoqué dans ce chapitre : en encapsulant le conteneur Linux dans une VM Hyper-V exécutant un Linux sous Windows. Le Linux utilisé par Microsoft est nommé LinuxKit LCOW (pour Linux Containers On Windows).

Il est donc possible, sous Windows, d'exécuter deux types de conteneurs différents :

- 0 des conteneurs Windows « natifs » (utilisant des images Windows) et tournant directement sur l'OS Windows (et apparaissant donc comme des processus Windows) ;
- 1 des conteneurs Windows ou Linux s'exécutant dans des « mini-VM » Hyper-V contenant des versions « simplifiées » de Linux et Windows.

Sous Mac, l'approche a été similaire à cette dernière option. Docker pour Mac utilise une VM HyperKit très légère qui joue le rôle d'Hyper-V pour Windows.

Bien que tout ceci puisse sembler un peu compliqué, nous verrons, dans le chapitre 3, que ces différences restent pour l'essentielle transparentes pour l'utilisateur de Docker.

Il faut néanmoins noter que Linux reste le citoyen de première classe du monde Docker. Que ce soit sous Windows ou MacOS, le moteur Docker sert souvent à exécuter des conteneurs Linux.

Pour comprendre comment fonctionnent les conteneurs, il est d'ailleurs utile de décrire les services Linux historiques sur lesquels se sont appuyées les premières implémentations (LXC par exemple). Docker (plus exactement runC) est basé sur deux extensions originales du kernel Linux :

- 0 CGroups ;
- 1 Namespaces.

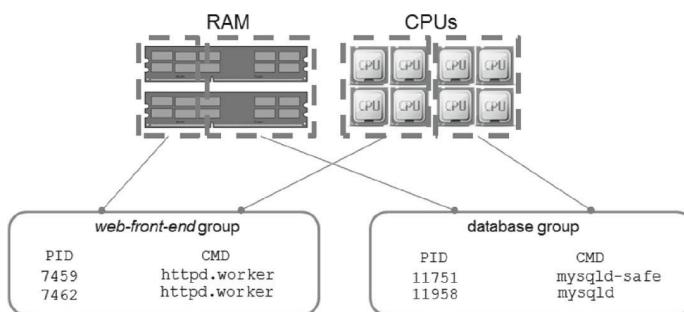
### 1.2.3 CGroups

CGroups<sup>7</sup> (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.

On doit cette bibliothèque aux ingénieurs de Google, qui a été probablement la première entreprise commerciale à utiliser des conteneurs, bien avant l'apparition de Docker.

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



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 groupe donné héritent des paramètres de leur parent.

### 1.2.4 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

### 1.2.5 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 :

- 0 contrainte : grâce à CGroups qui spécifie les limites en termes de ressources ;
- 1 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 Dockerfiles.

### 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.

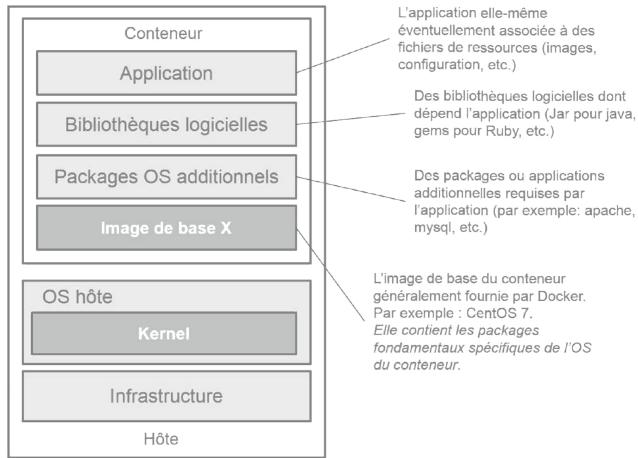
Ces images sont donc des archives qui peuvent être échangées entre plusieurs hôtes, mais aussi être réutilisées. Cette réutilisation est rendue possible par une seconde innovation : l'organisation en couches.

### 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*<sup>8</sup>, ces différentes couches s'agrègent en un tout cohérent : le conteneur ou une image elle-même réutilisable.

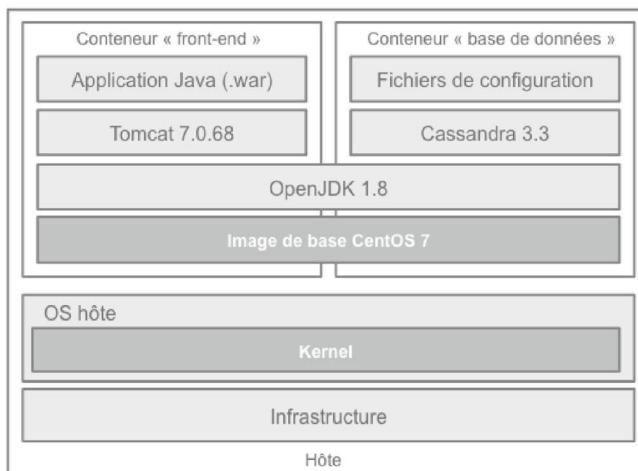
Figure 1.7 – Structure d'un conteneur au sein d'un hôte



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

- 0 un front-end basé sur une application JSP, tournant dans un serveur d'applications Tomcat<sup>9</sup>, exécuté par une machine virtuelle Java 1.8 sur un OS CentOS 7 ;
- 1 un back-end constitué d'une base de données Cassandra<sup>10</sup> s'exécutant elle aussi sur Java 1.8 et CentOS 7.

**Figure 1.8 – Exemple d'application à plusieurs conteneurs**



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.

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

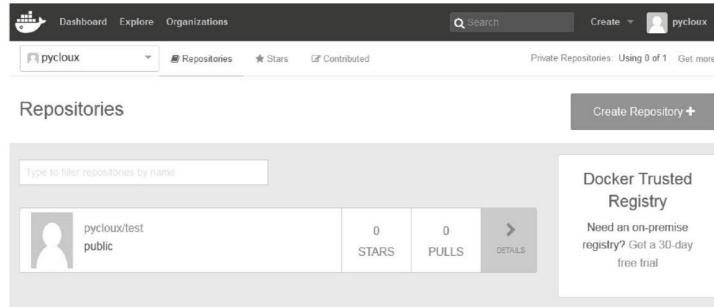
les conteneurs.

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 instantiation est le Docker Hub public et accessible via l'URL <https://hub.docker.com/>.

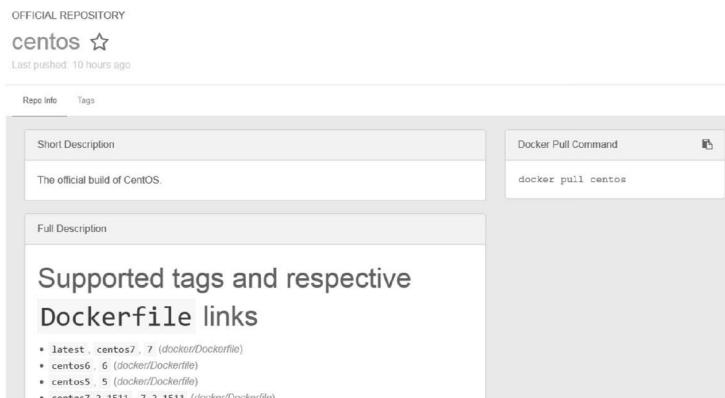
Figure 1.9 – Compte sur le Docker Hub



Il est possible, à tout un chacun, de se créer un compte gratuit sur ce registry pour publier des images. Il est même possible de les fabriquer « à distance » grâce à un Dockerfile disponible par exemple sur GitHub<sup>11</sup>. Un peu à la manière de GitHub, la publication d'images « publiques » est gratuite. Il faudra débourser quelques dollars pour pouvoir stocker plus d'une image privée.

Le registry Docker héberge les images de base des principaux systèmes d'exploitation dérivés de Linux (Ubuntu, CentOS, Windows, 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 instantiation publique d'un registry. Il est tout à fait possible à une entreprise d'installer un registry privé<sup>12</sup> pour y gérer ses propres images.



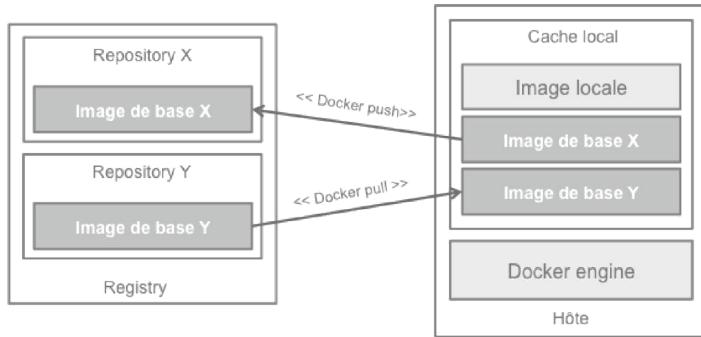
#### Les autres offres de registry cloud :

#### 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<sup>13</sup>, Amazon<sup>14</sup> et Microsoft<sup>15</sup> ont annoncé leurs propres implémentations de registries privés, ce qui concurrence de fait celui du Docker Hub.

Comment fonctionne le registry en lien avec le Docker Engine ?

Figure 1.11 – Installation et publication d'images



Le registry va être au choix :

0 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`) ;

1 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.

La réutilisation des images se produit donc de deux manières :

23 à l'échelle d'un hôte, par l'intermédiaire du cache local qui permet d'économiser des téléchargements d'images déjà présentes ;

24 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

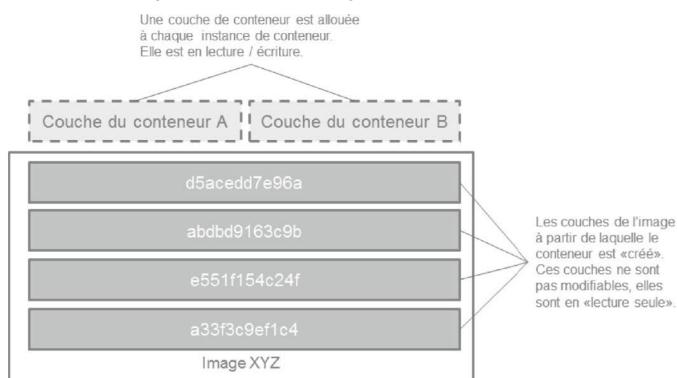
Nous savons maintenant qu'un conteneur est construit à partir d'une image au même titre qu'une machine virtuelle physique.

5888 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).

#### 23 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.

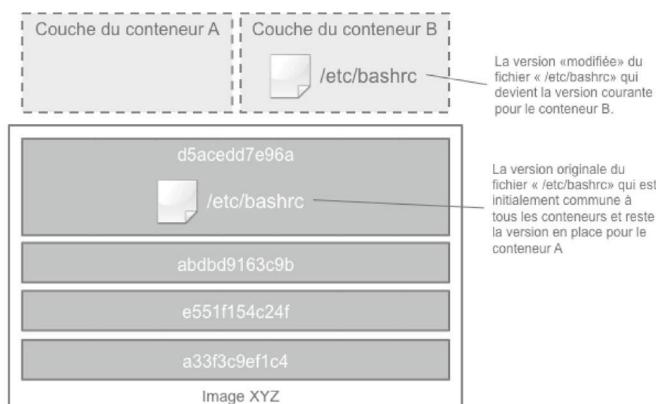
Dans le cas contraire, faire tourner un conteneur reviendrait à altérer potentiellement l'image sur laquelle il est construit, ce qui n'aurait aucun sens, puisque le but d'une image est justement de pouvoir créer autant de conteneurs que l'on souhaite, à partir du même moule.

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 ?

### 5888 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

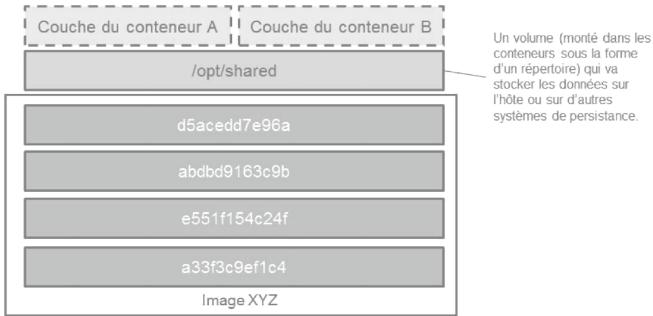
Docker peut fonctionner avec différentes implémentations de gestion de stockage. Il en existe plusieurs qui sont associées à des caractéristiques différentes ; citons, par exemple, `devicemapper`, `AUFS`, `OverlayFS`, etc. En général, Docker est livré avec un driver par défaut adapté au système d'exploitation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

### 23 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 :

5888 sur un espace disque de l'hôte géré par Docker (stocké sous `/var/lib/docker/volumes/` mais pas modifiables directement) ;

5889 sur un répertoire de l'hôte directement monté dans le conteneur et qui permet de créer un pont entre l'hôte et le conteneur ;

5890 sur un large panel de systèmes tiers qui imposent l'usage d'extensions particulières.

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.

## 23 4 LES OUTILS DE L'ÉCOSYSTÈME 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 :

5888 une ligne de commande ;

5889 des API ;

5890 la capacité de nommer, découvrir et gérer le cycle de vie des conteneurs ;

5891 la capacité à créer des conteneurs à partir d'images ou de modèles ;

5892 etc.



#### 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 :

**23 LXC**<sup>16</sup> (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.

**24 Rkt**<sup>17</sup> 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)<sup>18</sup>, l'une des tentatives de standardiser le monde des conteneurs. Rkt adopte une approche similaire à runC, dans le sens où il n'impose pas de recourir à un démon (à l'inverse de Docker) pour contrôler centralement le cycle de vie du conteneur. Pour résumer, CoreOS (qui fait partie de Red Hat Software depuis janvier 2018) a fabriqué un moteur de conteneur compatible avec les images Docker mais qui ne s'appuie pas sur les standards containerd ou runC.

**25 runC**<sup>19</sup> 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<sup>20</sup>. runC doit plutôt être considéré comme un composant de bas niveau destiné à s'intégrer dans des moteurs de plus haut niveau.

**26 OpenVZ**<sup>21</sup>, qui représente le moteur open source de la solution de gestion d'infrastructure virtualisée Virtuozzo<sup>22</sup>. 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.

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 :

**5888 Boot2Docker** est la distribution spécialisée par défaut que Docker installait dans VirtualBox pour le défunt Docker ToolBox, le kit Docker pour Windows et MacOS quand il n'existait pas encore de solution native. Il s'agit d'une distribution Linux très compacte, basée sur le Tiny Core Linux.

**5889 CoreOS**<sup>23</sup> aujourd'hui renommé **Container Linux**, qui est aussi l'éditeur de Rkt précédemment cité (et d'autres outils d'infrastructure que nous allons présenter par la suite).

**5890 Atomic**<sup>24</sup>, le sous-projet de Fedora (la distribution de base qui sert à RedHat ou CentOS) destiné à fournir un système d'exploitation pour les hôtes de conteneurs.

**5891 Windows Server Core** est la version compactée de Windows Server (6 GB d'installation contre 10 GB, tout est relatif). C'est une version de Windows que Microsoft recommande pour des hôtes qui n'exécutent que des machines virtuelles ou des conteneurs (un peu à la manière d'Atomic ou Container Linux). Il est aussi possible d'utiliser cet OS à l'intérieur d'un conteneur Windows.

**5892 Windows Nano Server**<sup>25</sup> est un système Windows réduit à sa plus simple expression. C'est aujourd'hui le système recommandé par Microsoft comme image de base pour un conteneur Windows. Notons néanmoins que cette version est si compacte qu'il est bien difficile d'exécuter une application avec. Beaucoup préfèrent encore utiliser Windows Server Core.

**5893 LinuxKit**<sup>26</sup> n'est pas, à proprement parler, un système d'exploitation. C'est en fait un outil pour

fabriquer des distributions Linux compactes destinées à des hôtes spécialisés dans l'exécution de conteneurs. L'un des OS créés avec LinuxKit est LinuxKit LCOW, le Linux qui est utilisé dans la version native de Docker pour Windows pour exécuter des conteneurs Linux au sein d'une machine virtuelle HyperV.

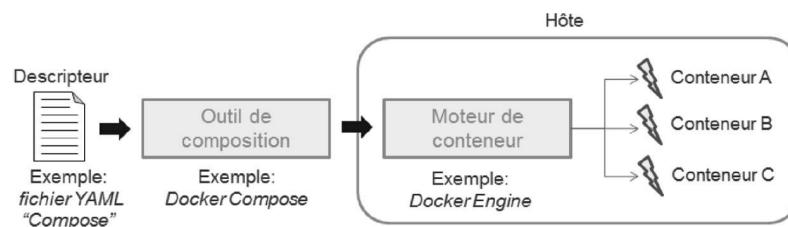
### 1.4.3 Les outils d'orchestration : composition et clustering

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 :

- 23 à définir un ordre de démarrage (directement ou indirectement) ;
- 24 à définir les systèmes de fichiers persistants (les volumes) ;
- 25 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



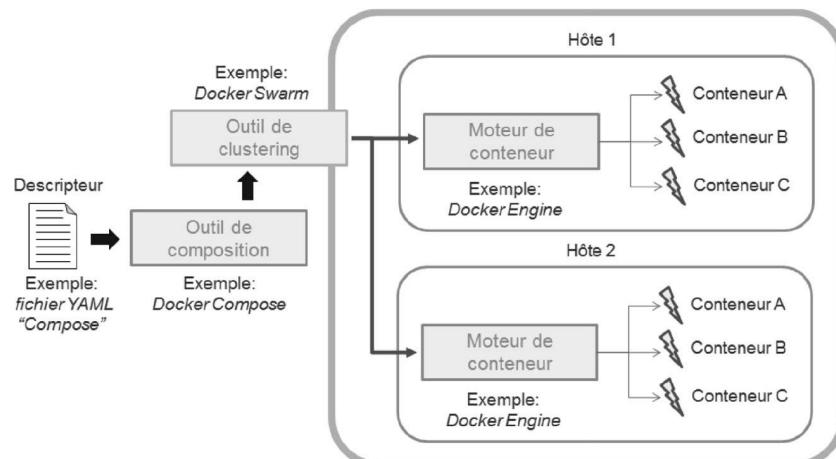
Il est bien évidemment possible de démarrer les conteneurs à la main à l'aide d'un script exécutant des séries de docker run.

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).

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*) et plus spécifiquement le standard de fait en la matière : **Kubernetes**.

Figure 1.16 – Clustering + composition = orchestration





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 Chef.

- 
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.
  2. <https://katacontainers.io>
  3. <https://github.com/moby/hyperkit>
  4. 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.
  5. <https://containerd.io/>
  6. <https://www.virtualbox.org/>
  7. <https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt>
  8. Nous verrons dans la suite de ce chapitre qu'il existe plusieurs implémentations de ce type de système de fichiers.
  9. <http://tomcat.apache.org/>
  10. <http://cassandra.apache.org/>
  11. <https://docs.docker.com/docker-hub/builds/>
  12. Docker propose d'ailleurs un logiciel nommé *Docker Trusted Registry* pour les clients qui souhaitent mettre en place leur propre instance.
  13. <https://cloud.google.com/container-registry/>
  14. <https://aws.amazon.com/fr/ecr/>
  15. <https://azure.microsoft.com/en-us/services/container-registry/>
  16. <https://linuxcontainers.org/>
  17. <https://coreos.com/rkt/>
  18. <https://github.com/appc/spec/>
  19. <https://runc.io/>
  20. <https://www.opencontainers.org/>
  21. <https://openvz.org/>
  22. <http://www.virtuozzo.com/>
  23. <https://coreos.com/os/docs/latest/>
  24. <https://getfedora.org/atomic/>
  25. [https://en.wikipedia.org/wiki/Server\\_Core](https://en.wikipedia.org/wiki/Server_Core)
  26. <https://github.com/linuxkit/linuxkit>

# 2

## Orchestration de conteneurs

### Objectif

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.

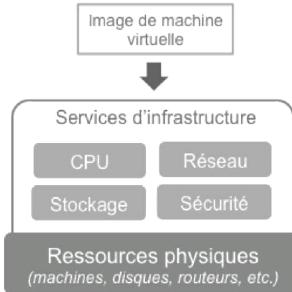


Pour ce qui est des infrastructures publiques (AmazonEC2, Google, Windows Azure, Heroku, Digital Ocean, etc.), le principe est identique, si ce n'est que les solutions sont propriétaires, tout en étant ouvertes via des services web documentés.

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 :

- 0 **de la puissance de calcul**, via un environnement d'exécution pour une machine virtuelle basée sur un hyperviseur : VMWare, Xen, KVM, etc. ;
- 1 **du stockage** qui peut se présenter sous différentes formes selon les niveaux de performance et de résilience requis ;
- 2 **des services réseau** : DHCP, pare-feu, fail-over, etc. ;
- 3 **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.

#### o 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 :

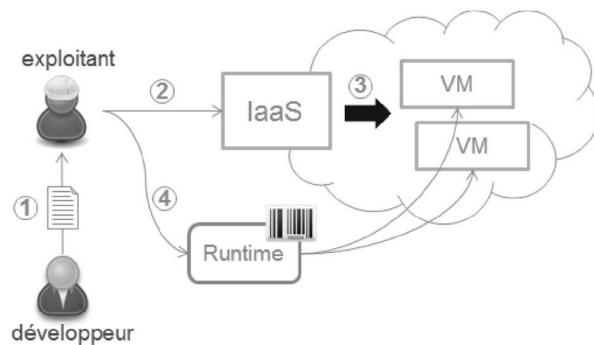
5888 la version de Java utilisée ;

5889 le serveur d'application, la base de données et les autres middlewares requis ;

5890 d'éventuels besoins de bibliothèques additionnelles de niveau OS ;

5891 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.

### 23 Déploiement d'applications en mode IaaS

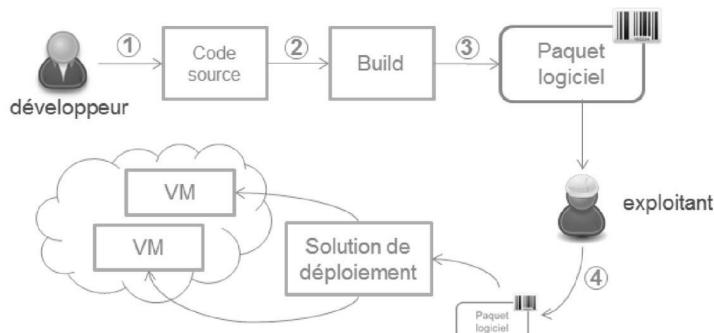
Entre chaque phase d'adaptation de l'environnement, on va trouver une série de déploiements, plus ou moins fréquents, de paquets logiciels correspondant à une nouvelle version de l'application.



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 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.

### 5888 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 :

- 23 en l'automatisant : tout se passe par configuration, pour peu que la capacité disponible soit suffisante ;
  - 24 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).
- 24 l'inverse, l'IaaS n'apporte aucune solution pour traiter deux questions fondamentales :
- 23 le déploiement des applications ;
  - 24 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

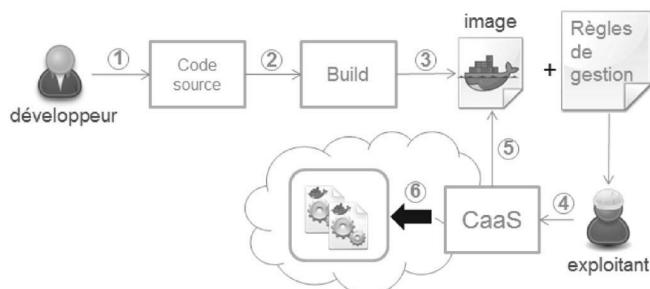
Les approches CaaS visent à aborder en premier lieu la problématique du déploiement d'applications. La problématique de la virtualisation des ressources matérielles est en fait secondaire.

#### 5888 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 :

- 23 nombre d'instances de chaque type de conteneur ;
- 24 liens entre les conteneurs ;
- 25 règles de montée en charge ;
- 26 règles de répartition de charge ;
- 27 volumes persistants ;
- 28 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 (que nous aborderons plus en détail dans un chapitre ultérieur) :

```

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 :

- 23 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) ;
- 24 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*.

### **5888 Conclusion sur le lien entre CaaS et IaaS**

Le modèle CaaS presuppose 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 :

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

##### 5888 Une régulation homéostatique

L'hémostasie est un concept décrit par le grand biologiste français Claude Bernard au XIX<sup>e</sup> siècle et qui se définit de la manière suivante :

23 *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 :

- 5888 la régulation de la température corporelle ;
- 5889 la régulation du taux de glucose dans le sang ;
- 5890 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 :

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

##### 5888 La gestion automatisée de l'infrastructure

Le système de régulation s'appuie généralement sur :

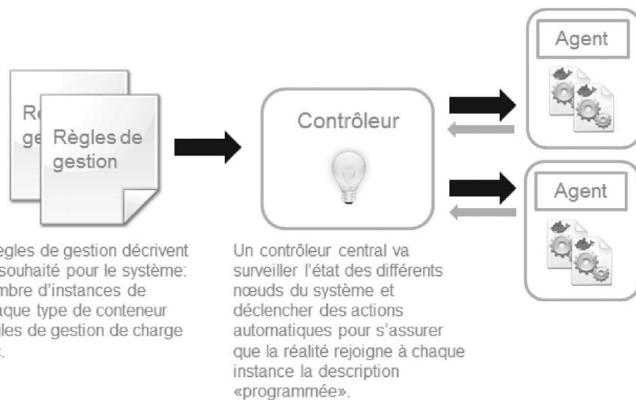
- 23 un contrôleur qui va analyser l'état de l'infrastructure par l'intermédiaire des données remontées par des agents installés sur les différents nœuds de l'architecture et, en fonction des règles préprogrammées, donner des ordres aux agents ;
- 24 des agents qui collectent des informations sur l'état 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) ;
- 25 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 2.5 présente une architecture générique de type 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.

Figure 2.5 – Principe de fonctionnement des CaaS



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.

## 5888 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<sup>1</sup>, qui constitue un exemple de DCOS ouvert comme OpenShift de RedHat mais aussi Service Fabric de Microsoft.

## 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. 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. Néanmoins un standard technologique émerge

nettement : Kubernetes.

### 2.2.1 Kubernetes : l'expérience de Google offerte à la communauté

Kubernetes<sup>2</sup> 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<sup>3</sup>. Par contre, il se positionne comme une alternative ouverte à Docker Swarm et Docker Compose pour la gestion du clustering et de l'orchestration.

Dans les faits, Kubernetes est aujourd'hui l'espéranto de l'orchestration de conteneurs. Même Docker, dont l'offre est historiquement basée sur Swarm, ou encore celle d'Amazon (ECS), s'appuyant aussi sur une solution maison, intègrent le support natif de Kubernetes.

Kubernetes est en outre aujourd'hui géré par la Linux Foundation<sup>4</sup> et, à ce titre, doit être considéré comme un standard à part entière.

#### 2.3 L'architecture de Kubernetes

La figure 2.6 présente une architecture synthétique de Kubernetes. On y distingue notamment les deux composantes habituelles des CaaS :

5888 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 ;

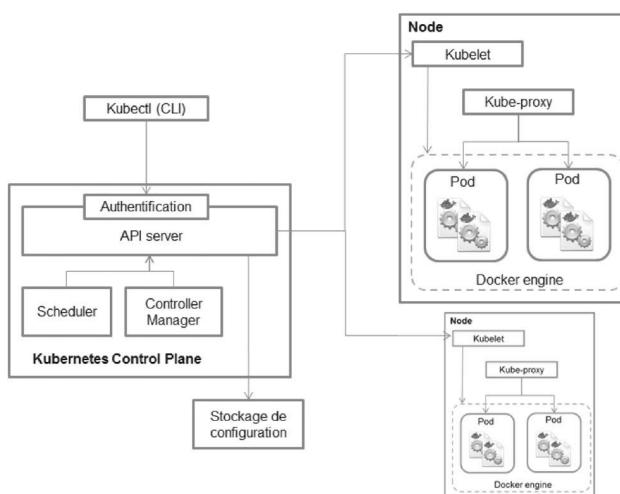
5889 un ou plusieurs nœuds (ou *node*) qui hébergent les agents Kubernetes, que l'on nomme également *kubelets*, et évidemment, les conteneurs organisés en *pods*<sup>5</sup>.

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<sup>6</sup> 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 aux conteneurs de s'abstraire de la localisation réelle des autres conteneurs dont ils pourraient avoir besoin.

Figure 2.6 – Architecture de Kubernetes



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

- 23 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 ;
- 24 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 ;
- 25 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).

#### 5888 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 *nodes* (correspondant généralement à un hôte physique ou virtuel) hébergent un ou plusieurs *pods*. Un *pod* est un groupe de conteneurs qui seront toujours colocalisés 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 machine logique commune :

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

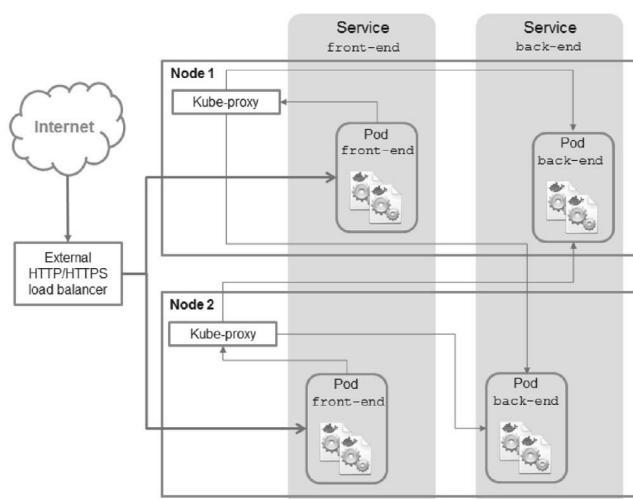


#### Modèle réseau Kubernetes

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*).

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

Figure 2.7 – 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 2.7) 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*<sup>7</sup>. 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.

#### **5888 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 :

- 5888 chiffrement SSL ;
- 5889 *load balancing* de niveau 7 ;
- 5890 affinité de session ;
- 5891 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*<sup>8</sup>.

Depuis sa version 1.2, Kubernetes a 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 »). Chaque implémentation commerciale (notamment cloud) de Kubernetes implémente cette fonction avec différents produits (ex. : Google Load Balancer). L'implémentation open source de référence est construite sur NGINX.

Avec Kubernetes, Google a tenté de s'imposer comme l'implémentation de référence du CaaS et a été dépassé par son succès. Aujourd'hui toute solution commerciale d'orchestration de conteneur se doit de supporter Kubernetes :

- 23 OpenShift<sup>9</sup>, la solution PaaS de RedHat ;
- 24 Google Container Engine (renommé Google Kubernetes Engine GKE) ;
- 25 Docker Entreprise ;
- 26 Microsoft Azure Kubernetes Service (AKS) ;
- 27 Amazon Elastic Container Service for Kubernetes (EKS) ;
- 28 Alibaba Cloud Kubernetes ;
- 29 Mesos (ou DCOS<sup>10</sup> sa déclinaison commerciale) dont Kubernetes constitue un framework, au même titre que Marathon (que nous présentons dans la suite de ce chapitre).

### **2.2.2 Docker Enterprise : 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 a ces dernières années beaucoup travaillé à la consolidation de ces différents outils en une suite cohérente : Docker Enterprise (ex- Docker Datacenter). La société a, dans un premier temps, proposé une solution cloud avant de l'abandonner au profit d'une approche d'éditeur logiciel plus classique. Il existe toujours des offres « cloud » de la suite Docker mais celles-ci sont désormais portées par les grands acteurs du cloud (Google, Amazon, Microsoft, Alibaba, etc.).

#### **5888 Les briques de base**

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

- 23 Docker Compose<sup>11</sup> est certainement le premier outil de composition de conteneurs qui soit apparu sur le marché. Il s'appuie sur un descripteur YAML<sup>12</sup> qui spécifie les conteneurs, leurs dépendances et un certain nombre d'autres paramètres utiles à la description de l'architecture.

Compose travaille de concert avec Swarm et même aujourd'hui Kubernetes pour former une solution d'orchestration complète. Docker essaie donc de positionner Compose comme une abstraction commune au-dessus de Swarm ou Kubernetes sans pour autant en imposer l'usage.

**5888 Docker Swarm<sup>13</sup>** est l'outil de clustering maison de Docker concurrent historique direct de Kubernetes qui offre exactement les mêmes fonctions. En pratique, 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.

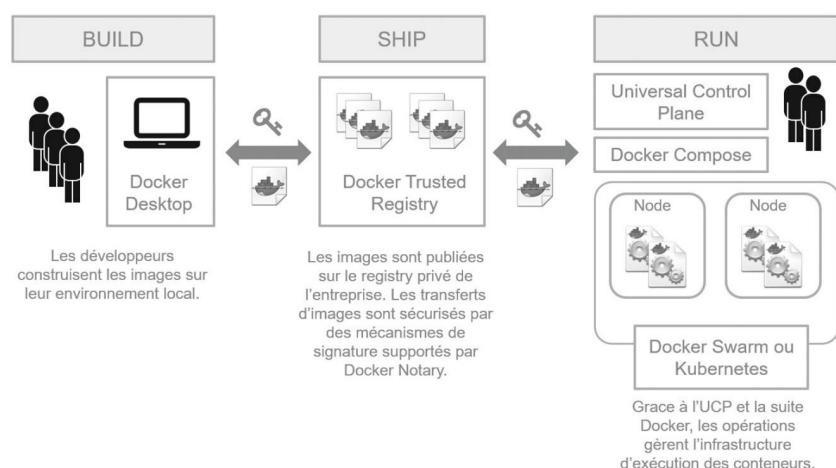
**5889 Kubernetes** est aujourd'hui nativement supporté par Docker Enterprise comme une alternative (équivalente) à la solution d'orchestration Swarm. Après avoir tenté d'imposer Swarm sur le marché, l'annonce de Docker en octobre 2017 a été reçue comme l'aveux d'un demi-échec. Il est aujourd'hui clair que Kubernetes est LE standard de fait de l'orchestration de conteneur.

**5890 Docker Trusted Registry<sup>14</sup>** 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<sup>15</sup>** qui gère la signature des images et leur vérification.

**5891 Docker UCP (Universal Control Plane)<sup>16</sup>** 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 2.8 – Chaîne de livraison de Docker



### 23 Comment ça marche ?

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

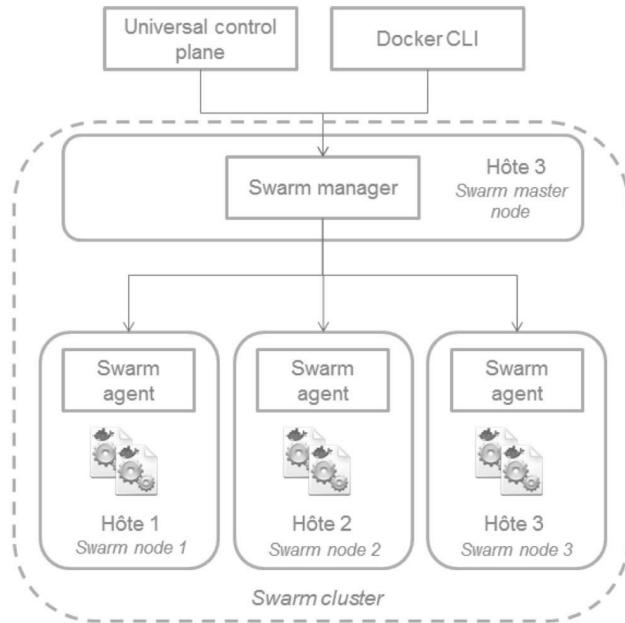


Notez qu'en production on aura généralement plusieurs *master nodes* (pour des raisons de haute disponibilité). Dans ce cas, plusieurs managers sont installés et la résilience de l'ensemble est assurée par la réplication de l'état de ces managers, par exemple au travers d'etcd.

Une fois l'ensemble installé, le moteur d'orchestration 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 les derniers chapitres, mais nous pouvons ici illustrer notre propos par quelques exemples.

Figure 2.9 – 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.

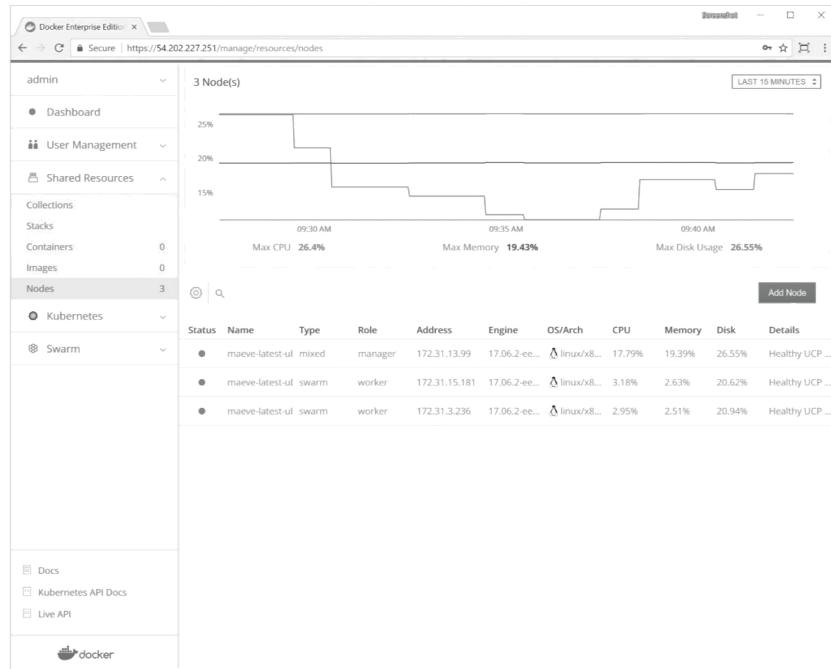
Grâce à cette bibliothèque de règles (qui va probablement s'enrichir progressivement), Swarm offre la possibilité de configurer un large panel d'architectures applicatives et de stratégie de gestion de charge.

#### **5888 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 2.10 – UCP, le cockpit graphique de l'écosystème Docker



Certains auront peut-être entendu parler du défunt projet open source *Shipyard*<sup>17</sup>, qui proposait aussi une interface d'administration graphique pour les outils Docker. Aujourd'hui, d'autres produits comme *Portainer*<sup>18</sup> (lui aussi open source) ou *Rancher*<sup>19</sup> offrent des alternatives vraiment intéressantes à UCP sans imposer d'intégration avec une suite logicielle particulière.

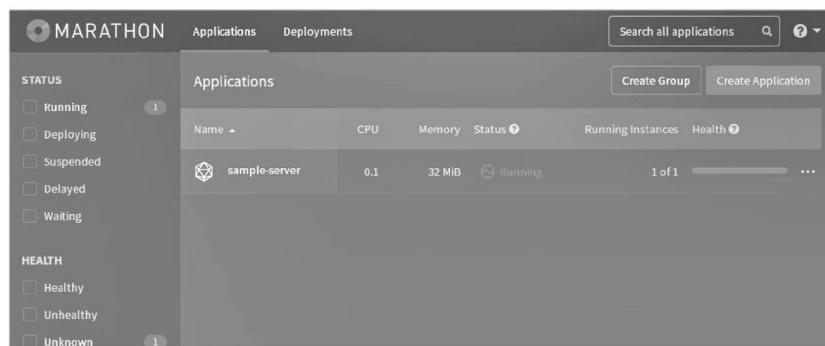
La suite CaaS de Docker n'est pas la seule sur le marché. D'autres solutions offrent des fonctionnalités similaires, tout en s'appuyant néanmoins presque systématiquement sur le moteur de Docker.

### 2.2.3 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.

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 :

23 un mesos master (éventuellement répliqué dans le cadre d'une architecture à haute disponibilité) ;

5888 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 (*synchronise ou batch*).

En inverse des solutions pures 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 :

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

## 2.2.4 OpenShift la solution de Red Hat

Comme Mesos, OpenShift a démarré avant la vague Docker mais il s'est rapidement imposé comme la solution commerciale de référence de Kubernetes pour les entreprises souhaitant une installation privée.

RedHat est depuis longtemps un contributeur majeur du projet open source Kubernetes avec Google ou aujourd'hui Microsoft.

En janvier 2018, RedHat a par ailleurs acquis l'un des pionniers du monde des conteneurs : CoreOS. La suite de CoreOS, constituée de produits indépendants organisés en une suite plus ou moins intégrée, s'efforçait de ressembler à celle de Docker. Aujourd'hui il est clair que l'approche de RedHat va être de s'approprier les composants de CoreOS pour les intégrer à OpenShift.

<sup>20</sup> Fleet, le moteur d'orchestration de composant de CoreOS (concurrent de Swarm ou Kubernetes) a été purement et simplement abandonné et le site GitHub de ce composant annonce tout simplement : « CoreOS (...) recommends Kubernetes for all clustering needs. »

Outre Rkt, le moteur de conteneur alternatif à Docker, dont nous avons déjà parlé, les produits survivants de la suite CoreOS sont :

- 0 Container Linux, un Linux ultra-compact pour les hôtes d'un cluster de conteneur ;
- 1 etcd, qui est utilisé pour assurer la haute disponibilité de Kubernetes.

Notons qu'OpenShift est toujours proposé dans une version cloud. Dans ce cas, l'infrastructure s'appuie sur l'offre IaaS d'Amazon ou de Google et le support est fourni par RedHat.

## 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 toutes compatibles avec Kubernetes. Si ces solutions sont encore peu utilisées commercialement, on constate qu'il s'agit pour tous d'un axe de développement majeur.

### o Google Kubernetes Engine

Il y a peu de choses 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.

### o Azure Kubernetes 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) était probablement la raison de ce retard.

Mais depuis 2016, et grâce à une collaboration très étroite avec Docker, Microsoft a plus que rattrapé son retard. Les conteneurs sont aujourd'hui des citoyens de première classe sur Windows, et

Microsoft dispose d'offres plus que convaincantes sur Azure.

L'offre AKS de Microsoft est probablement l'une des meilleures du marché. Le recrutement par la firme de Redmond de Bredan Burns (l'un des co-créateur de Kubernetes et ex-Google) n'est certainement pas étranger à cette situation.

#### ***o Amazon ECS, l'autre offre cloud***

Amazon a lancé son offre *Amazon EC2 Container Service* (ou ECS) en novembre 2014. Cette offre ne s'appuyait initialement 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<sup>21</sup>, mais aussi sur un contrôleur qui ne l'est évidemment pas, puisqu'il est uniquement implémenté par Amazon EC2 (dans le cloud). Le choix de mettre le code de l'agent ECS en open source était peut-être motivé par l'idée de proposer des offres hybrides public/privé à des sociétés qui y trouveraient leur intérêt. On aurait pu 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.

Mais en novembre 2017 Amazon, en lançant *Amazon Elastic Container Service for Kubernetes* (ou EKS), a néanmoins suivi la tendance du marché et s'est mis en conformité comme les autres. ECS reste néanmoins offert et supporté (pour le moment).

## **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 start-up ayant la chance de pouvoir s'offrir un *greenfield*<sup>22</sup>.

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**

Dès 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).

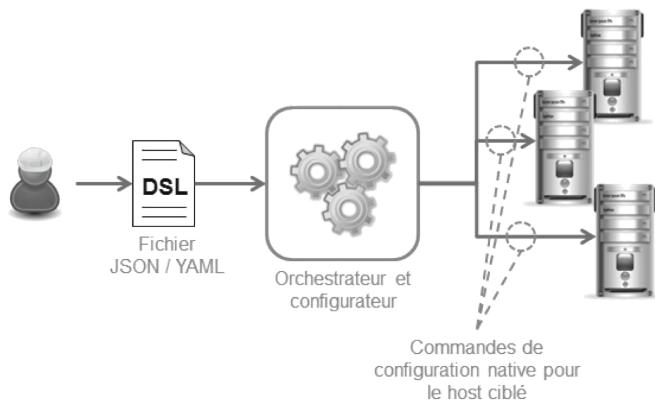
Ces solutions se basent sur des DSL<sup>23</sup> (*Domain Specific Language*) qui permettent de déclarer la configuration cible à atteindre pour des systèmes éventuellement distribués.

Cette configuration peut recouvrir :

- 23 la configuration de l'OS ;
- 24 l'installation et la configuration de logiciels standards ou personnalisés ;
- 25 la gestion de services (démarrage, redémarrage) ;
- 26 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 2.12 – 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.

### 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.

#### 5888 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 :

```
23    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`.

### **5888 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 :

```

0 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-dessus 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 clés SSH entre le contrôleur et l'hôte à configurer.

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

- 0 est bien installé et qu'il s'agit de la dernière version disponible ;
- 1 est bien installé pour démarrer au démarrage de l'hôte.

#### **o 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 :

- 0 installer Apache avec sa dernière version ;
- 1 faire en sorte qu'il démarre au démarrage 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) :

```
0 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<sup>24</sup>) 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.



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. Nous avons vu enfin que le standard d'orchestration de conteneur est aujourd'hui sans conteste Kubernetes.

- 
1. <http://mesos.apache.org/>
  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://www.linuxfoundation.org/>
  5. *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.
  6. <https://github.com/google/cadvisor>
  7. *Round robin* est un algorithme de répartition de traitement dans lequel les processus participants sont sollicités à tour de rôle.
  8. <https://cloud.google.com/compute/docs/load-balancing/network/>
  9. <https://www.openshift.com/>
  10. <https://www.mesosphere.com/>
  11. <https://docs.docker.com/compose/overview/>
  12. <http://yaml.org/>
  13. <https://www.docker.com/products/docker-swarm>
  14. <https://docs.docker.com/docker-trusted-registry/>
  15. <https://docs.docker.com/notary/>
  16. <https://docs.docker.com/ucp/>
  17. [https://github.com\(shipyard/shipyard](https://github.com(shipyard/shipyard)
  18. <https://portainer.io/>
  19. <https://rancher.com/>
  20. <https://github.com/coreos/fleet>
  21. <https://github.com/aws/amazon-ecs-agent>
  22. Littéralement un « champ vert » qui symbolise l'absence d'existant, aussi appelé *legacy*.
  23. 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.
  24. 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 chapitre suivant 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

### Objectif

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, nous pouvons jouer avec Docker.

O 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.

### 1 1 INSTALLATION DES EXEMPLES DU LIVRE

Les exemples de code du livre sont disponibles via notre dépôt GitHub. Une fois que votre environnement Docker aura été installé, vous pouvez récupérer les fichiers exemples avec la commande suivante :

```
| $ git clone https://github.com/dunod-docker/docker-examples-edition2.git
```

Les exemples de cet ouvrage ont été conçus pour un environnement Linux CentOS 7, ce qui correspond, pour les utilisateurs sous Windows ou Mac, à l'installation présentée ci-après dans le paragraphe « Linux sous VirtualBox ».

### 3. 2 INSTALLATION DE DOCKER

L'installation de Docker est en général simple et rapide à effectuer sur une distribution Linux. Sous Windows et Mac OS X, il est nécessaire de passer à travers un environnement virtualisé (s'appuyant sur une VM xhyve sous Mac OS X et Hyper-V ou VirtualBox selon la version de Windows).

Dans l'optique d'avoir une configuration aussi proche que possible d'un environnement Linux si vous êtes sous Mac OS X ou Windows, nous vous proposons d'utiliser une autre méthode d'installation.

#### 3.2.1 Linux sous VirtualBox

Il s'agit d'installer une distribution Linux de type CentOS sur une VM dans VirtualBox et d'installer ensuite Docker de manière standard.

Pour simplifier cette étape, nous vous proposons d'utiliser Vagrant<sup>1</sup>.



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

Figure 3.1 – Vagrant, le configurateur de machine virtuelle



Development Environments Made Easy

[GET STARTED](#) [DOWNLOAD 2.1.2](#) [FIND BOXES](#)

### o Une VM Linux CentOS

Sous Windows, il faut s'assurer que l'option de virtualisation (Intel VT ou AMD-V) soit bien activée dans le BIOS. Cela dépend de votre ordinateur, aussi nous vous recommandons une brève recherche sur Internet pour trouver la démarche à suivre.

Nous avons préparé une box Vagrant, basée sur un CentOS 7<sup>2</sup>, 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 :

23 installer VirtualBox<sup>3</sup>;

24 installer Vagrant<sup>4</sup>;

25 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
'verbs.vagrantup.com' for more information on using Vagrant.
```

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

```
$ 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.
```

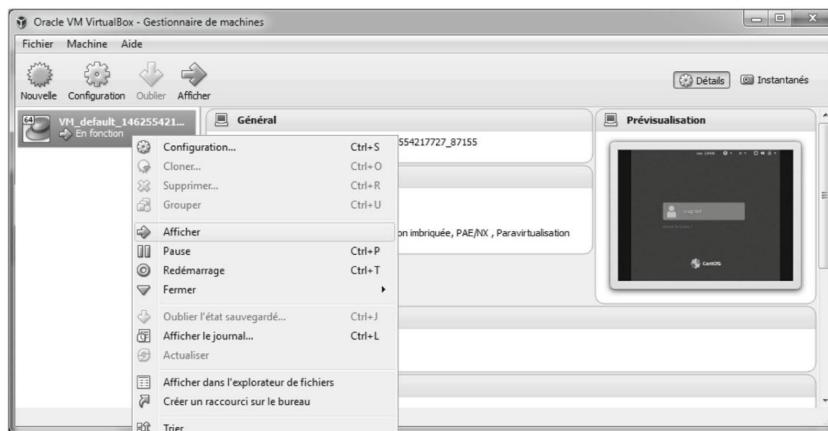
```

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 3.2 – Accès à l'image Vagrant pour se connecter



Cliquez sur « Finaliser la configuration » en bas à droite.

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

Figure 3.3 – Page de login CentOS7 de notre VM



### 23 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 5.2.10 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 :

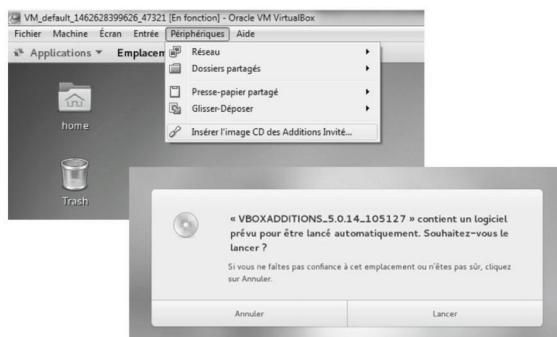
5888 arrêtez votre VM ;

5889 ajoutez un CDROM à votre VM (allez dans Configuration, Stockage, puis ajouter un lecteur optique que vous laisserez vide) ;

5890 redémarrez votre VM ;

5891 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 3.4).

Figure 3.4 – Mise à jour des guest additions

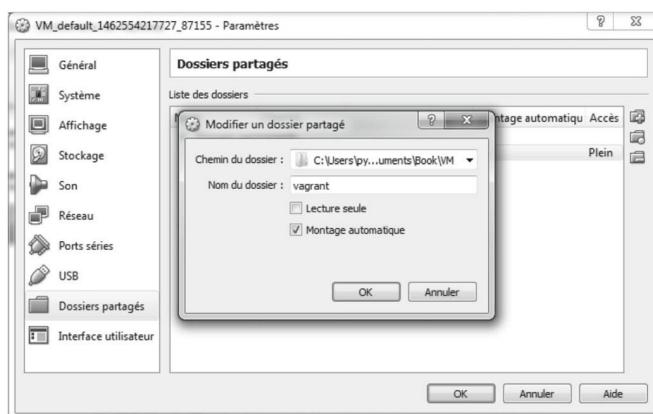


Le programme d'installation des *guest additions* vous demande de redémarrer votre VM. **Arrêtez cette dernière et lisez le paragraphe suivant avant de la lancer à nouveau.**

### 23 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.

Figure 3.5 – Auto-montage du partage de fichiers



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 3.5) *Montage automatique*.

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.

### 5888 Installation de Docker dans notre VM

Docker est disponible via les différents gestionnaires de paquets 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/install>

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

- 23 une architecture 64 bits, Docker n'étant pas supporté en 32 bits ;
- 24 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 :

5888 Connectez-vous sur votre machine avec un utilisateur 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 :

```
| 5888 sudo su
```

5888 Ajoutez le dépôt Docker pour yum (cela permet d'obtenir la dernière version, étant donné que le dépôt central CentOS met du temps à être mis à jour) :

```
| $ yum-config-manager \
```

```
|   --add-repo \
```

```
|   https://download.docker.com/linux/centos/docker-ce.repo
```

23 Installez le package Docker :

```
| $ yum install -y docker-ce-18.06.0.ce-3.el7
```

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.

Il ne reste plus qu'à démarrer le démon Docker<sup>5</sup>:

```
| $ systemctl start docker
```

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 :

```
$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world b901d36b6f2f:
9db2ca6ccae0: Pull complete
Digest:sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afacdc
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 une déconnexion 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.

### 3.2.2 Docker Desktop : 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 Desktop. 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 :

- 23 Docker Engine, le démon et le client Docker ;
- 24 Compose, outil simplifiant l'interconnexion de conteneurs Docker, disponible uniquement sous Docker Desktop pour Mac OS X ;
- 25 Docker Machine, qui permet de créer un hôte Docker.

Docker Desktop remplace Docker ToolBox qui permettait précédemment d'utiliser Docker sous Windows ou Mac OS X. Les différences principales sont une intégration plus native avec le système d'exploitation ainsi que la suppression d'Oracle VirtualBox comme couche de virtualisation.

Cependant, Docker Desktop nécessite Windows 10 Pro, Enterprise ou Education ou une version plus récente que Mac OS Yosemite pour fonctionner. Vous devrez continuer d'utiliser Docker ToolBox si vous disposez d'une version différente.

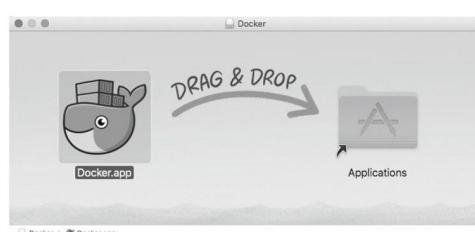
Docker Desktop crée une machine virtuelle (VM) Linux minimale tournant sous Windows via Hyper-V ou Mac OS X via HyperKit. Cette VM fait tourner le démon Docker qui peut ensuite être contrôlé depuis votre système d'exploitation de différentes façons :

- 5888 via Kitematic ;
- 5889 via le client Docker ;
- 5890 en direct grâce à des requêtes HTTP (n'oublions pas que le démon Docker expose son API en REST<sup>6</sup>).

Le processus d'installation est relativement simple :

- 23 allez sur la page de Docker for Mac<sup>7</sup> ou Docker for Windows<sup>8</sup> et téléchargez l'installateur correspondant à votre système d'exploitation depuis le Docker store;
- 24 lancez l'installateur en double-cliquant dessus et suivez les instructions.

Figure 3.6 – Installation de Docker Desktop sous Mac OS X



Une fois l'installation terminée, lancez un terminal pour vérifier que l'installation s'est bien déroulée :

```
5888 docker
version Client:
  Version: 18.06.0-ce
  API version: 1.38
  Go version: go1.10.3
  Git commit: 0ffa825
  Built: Wed Jul 18 19:05:26 2018
```

```

OS/Arch: darwin/amd64
Experimental: false

Server:
  Engine:
    Version: 18.06.0-ce
    API version: 1.38 (minimum version 1.12)
    Go version: go1.10.3
    Git commit: 0ffa825
    Built: Wed Jul 18 19:13:46 2018
  OS/Arch: linux/amd64
  Experimental: true

Kubernetes:
  Version: v1.10.3
  StackAPI: v1beta2

```

Docker Desktop est un moyen simple et rapide pour découvrir Docker. Il permet d'avoir accès au client avec la ligne de commande mais aussi d'avoir accès aux options de configuration simplement via une interface graphique.

Figure 3.7 – Docker Desktop : panneaux de configuration



Soyons clairs, Docker Desktop 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. 3 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 :

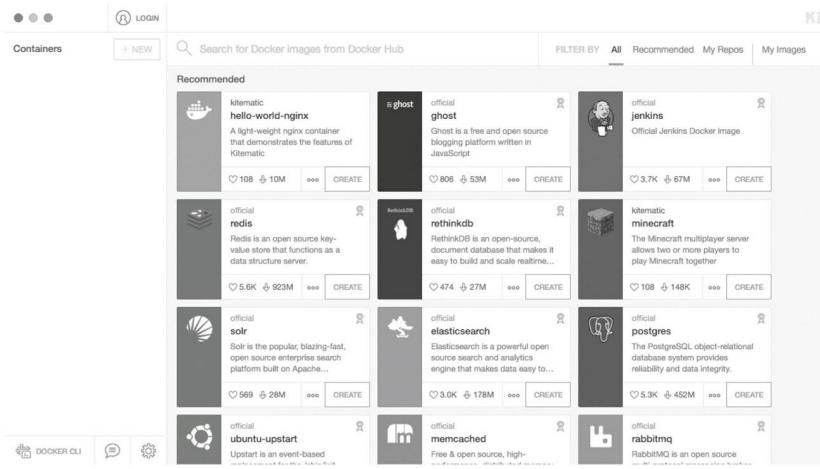
- 23 via Kitematic ;
- 24 via le client Docker, le cas le plus commun, que nous privilégierons dans la suite de cet ouvrage ;
- 25 directement avec des appels HTTP en utilisant l'API Docker Remote.

#### 3.3.1 Kitematic

Kitematic est un logiciel fourni avec Docker ToolBox ou téléchargeable depuis Docker Desktop (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.

Figure 3.8 – 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.

Figure 3.9 – Démarrage d'un conteneur NGINX dans Kitematic



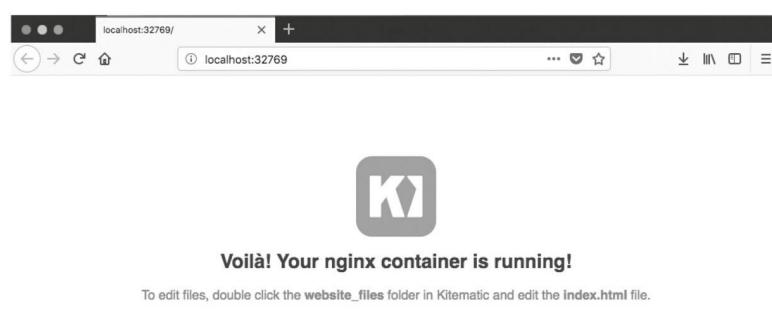
Kitematic liste tous les conteneurs existants arrêtés ou actifs (ce qui est indiqué par la marque « A » sur la figure 3.9). On voit dans notre cas le conteneur que nous venons de créer.

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 3.9). 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. En cliquant sur le lien *web preview* (marque « C » sur la figure 3.9), un navigateur s'ouvre et vous pouvez voir la page d'accueil de NGINX.

Figure 3.10 – 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<sup>9</sup>.

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.

### 3.3.2 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<sup>10</sup> 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 3.11 – Communication client/serveur Docker



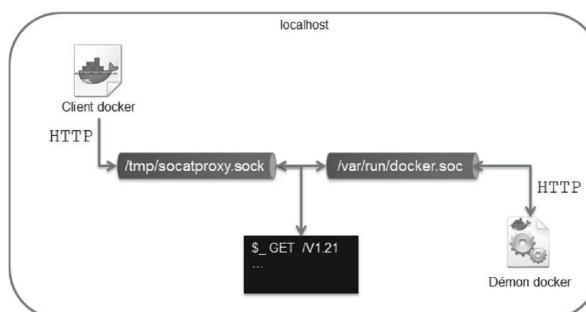
Il est possible de matérialiser cette communication simplement en sniffant le trafic HTTP entre le client et le démon. Pour cela, nous allons utiliser Socat<sup>11</sup>.



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.

Figure 3.12 – Interception de communication avec socat



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

```
| $ sudo yum install -y 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
5888 2018/08/18 14:54:31.603773 length=83 from=0
to=82 GET /_ping HTTP/1.1\r
Host: docker\r
User-Agent: Docker-Client/18.06.0-ce (linux)\r
\r
< 2018/08/18 14:54:31.604633 length=215 from=0 to=214
```

```

HTTP/1.1 200 OK\r
Api-Version: 1.38\r
Docker-Experimental: false\r
Ostype: linux\r
Server: Docker/18.06.0-ce (linux)\r
Date: Sat, 18 Aug 2018 12:54:31 GMT\r
Content-Length: 2\r
Content-Type: text/plain; charset=utf-8\r
\r
OK> 2018/08/18 14:54:31.605472 length=105 from=83 to=187
GET /v1.38/containers/json?all=1 HTTP/1.1\r
Host: docker\r
User-Agent: Docker-Client/18.06.0-ce (linux)\r
\r
23 2018/08/18 14:54:31.606601 length=908 from=215
to=1122 HTTP/1.1 200 OK\r
Api-Version: 1.38\r
Content-Type: application/json\r
Docker-Experimental: false\r
Ostype: linux\r
Server: Docker/18.06.0-ce (linux)\r
Date: Sat, 18 Aug 2018 12:54:31 GMT\r
Content-Length:
702\r \r
[{"Id": "9a9510f8095ac5cb557f9871bc8da099ba2e211a1e9dd5fb1ff7ca42777a0854", "Names": [
"/upbeat_hermann"], "Image": "hello-world", "ImageID": "sha256:2cb0d9787c4dd17ef9eb03e512923bc4db10add190d3f84af63b744e353a9b34", "Command": "/hello", "Created": "1534589264", "Ports": [], "Labels": {}, "State": "exited", "Status": "Exited (0) 2 hours ago", "HostConfig": {"NetworkMode": "default"}, "NetworkSettings": {"Networks": {"bridge": {"IPAMConfig": null, "Links": null, "Aliases": null, "NetworkID": "87cacd51e70523481c9321ed58325f2c07c2eb3c18367bf871ca64780c273f99", "EndpointID": "", "Gateway": "", "IPAddress": "", "IPPrefixLen": 0, "IPv6Gateway": "", "GlobalIPv6Address": ""}, "GlobalIPv6PrefixLen": 0, "MacAddress": "", "DriverOpts": null}}}, "Mounts": []}
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS NAMES
9a9510f8095a hello-world "/hello" 2 hours ago Exited (0) 2 hours ago
upbeat_hermann

```

Regardons plus en détail :

Paramètre	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 /v1.38/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": "9a9510f8095ac5cb557f9871bc8da099ba2e211a1e9dd5fb1ff7ca42777a0854", ...]	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 9a9510f8095a hello-world "/hello" 2 hours ago Exited (0) 2 hours ago upbeat_hermann	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.



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.

### 3.3.3 API Docker Remote

Le démon Docker expose l'intégralité de ses méthodes à travers l'API Docker Remote [12](#). 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.

#### 5888 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 :

```
| $ 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 :

```
23    sudo dockerd -H
tcp://0.0.0.0:2375 & [2] 3012
[vagrant@localhost ~]$ WARN[2018-08-18T15:14:21.735748706+02:00] [!] DON'T BIND ON ANY IP
ADDRESS WITHOUT setting --tlsverify IF YOU DON'T KNOW WHAT YOU'RE DOING [!]
INFO[2018-08-18T15:14:21.738046001+02:00] libcontainerd: started new docker-containerd process
pid=3026
INFO[2018-08-18T15:14:21.738118595+02:00] parsed scheme: "unix"
module=grpc
INFO[2018-08-18T15:14:21.738129267+02:00] scheme "unix" not registered, fallback to default
scheme module=grpc
INFO[2018-08-18T15:14:21.738203148+02:00] ccResolverWrapper: sending new addresses to cc:
[{unix:///var/run/docker/containerd/docker-containerd.sock 0 <nil>}] module=grpc
...
INFO[2018-08-18T15:14:22.067824300+02:00] Loading containers: done.
INFO[2018-08-18T15:14:22.101551277+02:00] Docker daemon
commit=0ffa825 graphdriver(s)=overlay2 version=18.06.0-ce
INFO[2018-08-18T15:14:22.101638279+02:00] Daemon has completed initialization
INFO[2018-08-18T15:14:22.114086443+02:00] API listen on [::]:2375
```

Regardons maintenant en détail le résultat :

Paramètre	Description
sudo dockerd -H tcp://0.0.0.0:2375 &	Démarre le démon Docker dockerd 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.
API listen on [::]:2375	Ce message nous confirme que notre démon docker est bien démarré sur le port 2375

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 nous connecter à notre démon, il faut maintenant passer l'adresse que nous venons de définir

avec le paramètre -H :

```
5888 docker -H 0.0.0.0:2375
  info Containers: 1
    Running: 0
    Paused: 0
    Stopped: 1
    Images: 1
  Server Version: 18.06.0-ce
  Storage Driver: overlay2
    Backing Filesystem: xfs
    Supports d_type: true
    Native Overlay Diff: true
...
```



### 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* [13](#), et il faut donc installer le paquet nécessaire :

```
| sudo yum install -y 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  Current
          Dload  Upload Total   Spent    Left Speed
100 2195    0  2195    0  0 97309      0 --:--:-- --:--:-- --:-- 99772
{
  "ID": "AEEB:2VRP:W6CM:MHCT:USVJ:ME76:BTUL:XIDO:46IR:NEKK:S3JT:H4PI",
  "Containers": 1,
  "ContainersRunning": 0,
  "ContainersPaused": 0,
  "ContainersStopped": 1,
  "Images": 1,
  "Driver": "overlay2",
  "DriverStatus": [
    [
      "Backing Filesystem",
      "xfs"
    ],
    [
      "Supports d_type",
      "true"
    ],
    [
      "Native Overlay Diff",
      "true"
    ]
  ],
}
```

| ...

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 fils dockerd -H tcp://0.0.0.0:2375.

### **23 Modifier la configuration du démon Docker**

Au prochain redémarrage de votre machine, Docker va reprendre son ancienne configuration. Il n'écouterait 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 :

```
5888 sudo tee /etc/docker/daemon.json <<- 'EOF'  
{  
  "hosts": ["unix:///var/run/docker.sock", "tcp://0.0.0.0:2375"]  
}  
EOF
```

Notre démon écoute maintenant sur deux sockets :

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

Le lecteur curieux pourra se reporter à la documentation Docker<sup>14</sup> relative à *systemd* qui aborde cette problématique en détail.

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

```
0 sudo systemctl daemon-reload  
1 sudo systemctl restart docker
```

Nous pouvons facilement le confirmer avec la commande suivante :

```
$ sudo netstat -lntp | grep dockerd  
tcp6 0 0 ::::2375 :*: LISTEN 2459/dockerd
```

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



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. <http://www.vagrantup.com>
  2. <https://app.vagrantup.com/dunod-docker/boxes/centos7>
  3. <http://virtualbox.org> (version utilisée dans ce livre 5.2.10)
  4. <https://docs.vagrantup.com/v2/installation/index.html> (version utilisée dans ce livre 2.0.3)
  5. CentOS 7 utilise systemd comme système d'initialisation. Pour en savoir plus :  
<https://wiki.freedesktop.org/www/Software/systemd/>
  6. [https://fr.wikipedia.org/wiki/Representational\\_State\\_Transfer](https://fr.wikipedia.org/wiki/Representational_State_Transfer)
  7. <https://docs.docker.com/docker-for-mac/install/>
  8. <https://docs.docker.com/docker-for-windows/install/>
  9. <https://docs.docker.com/kitematic/userguide/>
  10. [https://fr.wikipedia.org/wiki/Berkeley\\_sockets#Socket\\_unix](https://fr.wikipedia.org/wiki/Berkeley_sockets#Socket_unix)
  11. <http://www.dest-unreach.org/socat/>
  12. <https://docs.docker.com/engine/api/version-history/>
  13. jq est un processeur de JSON en ligne de commande. Plus d'information à <https://stedolan.github.io/jq/>
  14. <https://docs.docker.com/engine/admin/systemd>

# 4

## Conteneurs et images

### Objectif

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

### 1 1 LE CYCLE DE VIE DU CONTENEUR

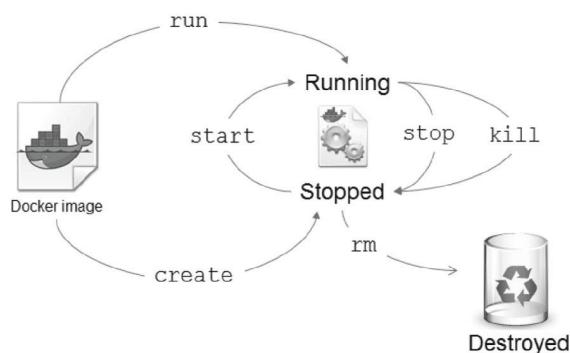
Comme nous l'avons évoqué dans les précédents chapitres, un conteneur Docker est fabriqué à partir d'une image. Il peut ensuite adopter différents états que nous allons maintenant présenter.



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 4.1 – Cycle de vie du conteneur Docker



#### 4.1.1 Créer et démarrer un conteneur (commande `run`)

Créons un conteneur NGINX<sup>1</sup>. 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` :

0 `-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 ;

1 `-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 ;

2 `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.

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
9ee13ca3b908: Pull complete
23cb15b0fcce: 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 page d'accueil, `http://localhost:8000`, en utilisant le navigateur Firefox installé dans notre image Linux de référence.

Figure 4.2 – Premier conteneur NGINX



Que faire de ce conteneur ?

Le cycle de vie de la figure 4.1 nous indique que deux options sont possibles :

23 stopper le conteneur avec la commande `stop` ;

24 tuer le conteneur avec la commande `kill`.

#### 4.1.2 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.

### 5888 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
STATUS	PORTS	NAMES	
855bbd5e9823	nginx	"nginx -g 'daemon off..."	17 minutes ago
Up 17 minutes	443/tcp, 0.0.0.0:8000->80/tcp	stupefied_fermat	

Voici un descriptif des informations rentrées 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) <sup>2</sup> 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 à 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 <sup>3</sup> .

### 23 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 system*) 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'apparaît 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		COMMAND	CREATED
CONTAINER ID	IMAGE		
STATUS	POR	TS	NAMES
855bbd5e9823	nginx	"nginx -g 'daemon of..."'	2 hours ago
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
```

#### 4.1.3 Détruire un conteneur (commande rm)

C'est la dernière étape du cycle de vie de notre conteneur. Tentons de détruire 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 4.1, 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.

#### 4.1.4 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		COMMAND	CREATED
CONTAINER ID	IMAGE		
STATUS	POR	TS	NAMES
2ab2ba1543ce	nginx	"nginx -g 'daemon of..."'	8 seconds ago
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 fournit un service correspondant à nos besoins ?

## ← 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.

### 4.2.1 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	POR	T	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 dev etc home lib lib64 media mnt opt proc 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

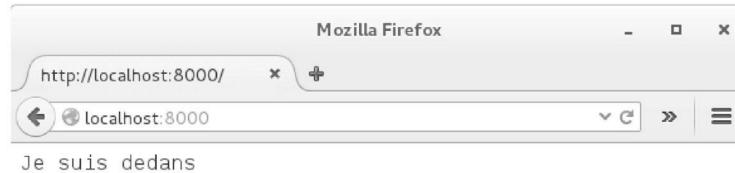
du conteneur. Ce sera systématiquement le cas pour tout conteneur.

Pour confirmer que nous sommes bien à l'intérieur du conteneur, nous allons modifier la page d'accueil 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 4.3 – Page d'accueil 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.

#### 4.2.2 Créer un volume

Les volumes<sup>4</sup> 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.* »

Pour comprendre le résultat de cette action, nous allons inspecter le conteneur ainsi créé. La commande `inspect` permet d'obtenir une structure JSON<sup>5</sup> décrivant le conteneur et sa configuration.

```
| $ docker inspect webserver
```

Au sein de la structure renournée 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).

Driver	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 <i>read-write</i> (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
```

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.

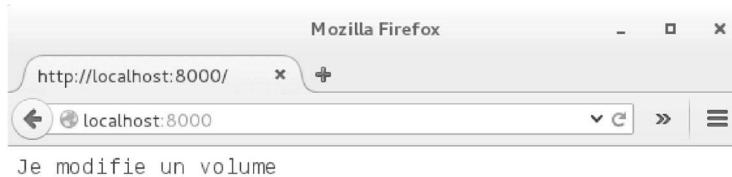
Un accès à l'URL `http://localhost:8000` nous confirme que le serveur web tourne bien correctement et affiche l'habituelle page d'accueil standard de NGINX.

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 priviléges 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 4.4 – Page d'accueil de NGINX modifiée sur l'hôte



Cette méthode présente néanmoins des inconvénients majeurs :

- ← il est nécessaire de disposer de priviléges 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 via l'instruction `docker system prune` par exemple).

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 quatrième 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.

#### 4.2.3 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.

La syntaxe de création du volume est la suivante :

```
$ docker run -p 8000:80 --name webserver -d -v  
/home/vagrant/workdir/chapitre4:/usr/share/nginx/html nginx
```

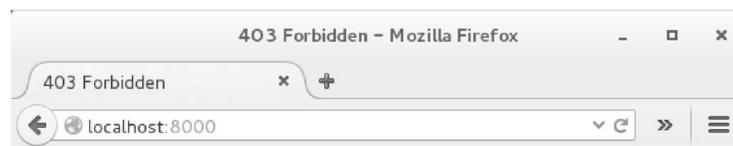
Encore une fois, pensez à vérifier à l'aide de la commande `docker ps -a` que vous n'avez pas de conteneur existant qui porte déjà le nom « `webserver` » ou qui écoute sur le même port 8000. Si c'est le cas (ça devrait l'être si vous avez suivi les exercices précédents pas à pas), pensez à supprimer 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 4.5 – Montage d'un répertoire vide



## 403 Forbidden

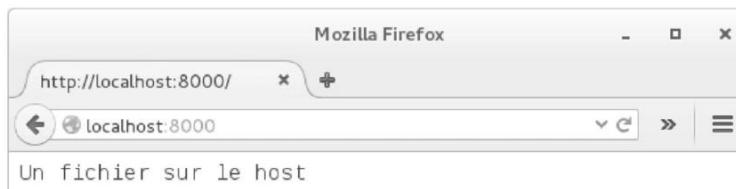
nginx/1.9.9

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/chapitre4  
$ 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 4.6 – 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`.

### 4.2.4 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": {
```

```

"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<sup>6</sup>: 0.0.0.0. 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            STATUS              NAMES
97066be7b321        nginx              "nginx -g 'daemon of..."   4 minutes ago      Up 3 minutes       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.

## 4. 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.

### 4.3.1 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` :

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>	23cb15b0fcce	5 weeks ago	125.1 MB
	<none>				
	<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.

### 4.3.2 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 050e5c5557d8a 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

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 4.7 – Tags de l'image officielle nginx sur le Docker Hub

Tag Name	Size	Docker Pull Command
1.9.9	55 MB	docker pull nginx
1.9	55 MB	
1	55 MB	
latest	55 MB	

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
```

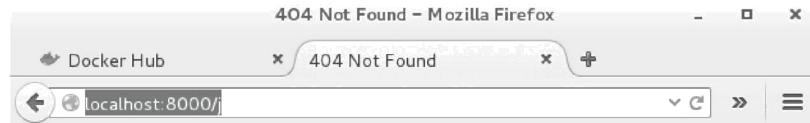
Cela n'est pas aussi compliqué qu'il y paraît : il suffit de suffixer le nom de l'image avec le *tag* que vous souhaitez télécharger grâce à la commande `pull`, soit `nginx:1.7` pour le *tag* 1.7.

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/>. En effet, NGINX, par défaut, affiche son numéro de version sur les pages d'erreur 404.

Figure 4.8 – Conteneur avec NGINX en version 1.7



## 404 Not Found

nginx/1.7.12

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/>, 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.

### 4.3.3 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`.

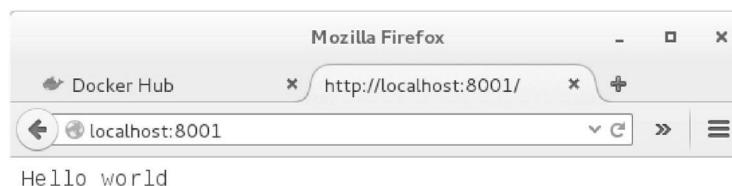
```
← docker commit webserver nginxhello
908a3cb565656bca615367b5009ba8d17ddadbd24fa77cfbb4bbfcf4c336227
← docker images
REPOSITORY          TAG      IMAGE ID      CREATED       VIRTUAL SIZE
nginxhello          latest   908a3cb56565  3 seconds ago  133.8 MB
nginx               latest   5328fdfe9b8e  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 4.9 – Conteneur créé à partir de l'image « `nginxhello` »



Hello world

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.

## 4. 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.

### 4.4.1 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 Dockerfiles : décrire la création d'images de manière formelle.

### 4.4.2 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
--> d5aceedd7e96a
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 :

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.

#### 4.4.3 Quelques explications

Penchons-nous maintenant sur les quelques instructions que nous avons utilisées dans ce fichier Dockerfile.

```
| 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 Dockerfiles des images sont publiés sur la partie publique du Docker Hub :

Figure 4.10 – 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;"]
```

← 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 --format "table {{. ID}} {{. CreatedBy}} {{. Size}}"
IMAGE          CREATED BY      SIZE
a8f7c2454f7f    /bin/sh -c #(nop) COPY
                  file :1378a5a0c1fa30f...
d5acedd7e96a    /bin/sh -c #(nop) CMD ["nginx"  0 B
                  "-g" "daemon...
abdbd9163c9b    /bin/sh -c #(nop) EXPOSE
                  443/tcp 80/tcp
e551f154c24f    /bin/sh -c #(nop) VOLUME
                  [/var/cache/nginx]
a33f3c9ef1c4    /bin/sh -c ln -sf /dev/stderr
                  /var/log/nginx...
b1ce25f2cd9c    /bin/sh -c ln -sf /dev/stdout
                  /var/log/nginx...
3c5fee6e68b3    /bin/sh -c apt-get update &&
                  apt-get install
4446b7f2ac03    /bin/sh -c #(nop) ENV
                  Nginx_VERSION=1.7.12-...
e97439968948    /bin/sh -c echo "deb
                  http://nginx.org/packa...
acca21866ee7    /bin/sh -c apt-key adv --
                  keyserver pgp.mit...
6dfc7b45c331    /bin/sh -c #(nop) MAINTAINER
                  Nginx Docker M...
325520a5dbfd    /bin/sh -c #(nop) CMD
                  ["/bin/bash"]
358523180737    /bin/sh -c #(nop) ADD
                                         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.



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.

→ 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.

- 
1. <http://nginx.org/>
  2. <https://www.ietf.org/rfc/rfc4122.txt>
  3. <https://github.com/docker/docker/blob/master/pkg/namesgenerator/names-generator.go>
  4. Veuillez vous reporter au chapitre 1 pour une explication du concept de volume de conteneur.
  5. <http://www.json.org/>
  6. La notion d'adresse "0.0.0.0" par rapport à "127.0.0.1" est expliquée dans le chapitre 3 (§ [3.3.3](#)).

## 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 Dockerfiles que nous avons commencé à étudier dans le dernier chapitre de la précédente partie.

Nous avons choisi de découper cette partie en trois chapitres qui peuvent être éventuellement lus de manière non séquentielle :

- ← le chapitre 5 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 6 est consacré aux principales instructions du Dockerfile, c'est-à-dire celles qui sont présentes dans 95 % des fichiers Dockerfile que vous aurez à lire ;
- ← le chapitre 7 se penche, quant à lui, sur les instructions Dockerfile et autres fonctionnalités plus avancées de Docker.

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 5.

# 5

## Prise en main du client Docker

### Objectif

Dans ce chapitre, nous vous présenterons les commandes usuelles du client Docker et leurs options les plus importantes.

1. 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.

### ← 1 INTRODUCTION À LA CLI<sup>1</sup> DOCKER

Dans ce premier paragraphe, nous allons donner quelques informations complémentaires sur le client Docker, à savoir :

- ← le format des commandes ;
- ← les variables d'environnement qu'il utilise (et qui permettent d'en modifier la configuration) ;
- ← la structure générique des options des différents outils Docker.

#### 5.1.1 Le format des commandes

Initialement, les commandes étaient toutes sous le format `docker command options`. Le nombre de fonctionnalités ne faisant qu'augmenter avec les versions successives de Docker, il devint nécessaire de « mettre de l'ordre ». De plus, certaines commandes n'étaient pas forcément intuitives (comme `docker ps` pour lister les conteneurs alors qu'on utilise `docker images` pour lister les images...).

← la release 1.13, le format a été aligné sur celui des volumes et réseaux : toutes les commandes concernant un conteneur sont préfixées par `docker container` et celles pour les images par `docker image`.

Ce qui donne des commandes plus lisibles dont voici quelques exemples :

- ← `docker container ls` au lieu de `docker ps` pour lister les conteneurs ;
- ← `docker image ls` au lieu de `docker images` ;
- ← `docker container rm` au lieu de `docker rm` pour supprimer un conteneur ;
- ← `docker image rm` au lieu de `docker rmi` pour supprimer une image.

Les anciennes commandes étant toujours disponibles et plus rapides à saisir, nous continuerons d'en utiliser certaines à travers ce livre.

#### 5.1.2 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 5.1 décrit de manière exhaustive les variables d'environnement Docker. Nous verrons ensuite un exemple illustrant l'impact fonctionnel d'une telle variable.

**Tableau 5.1 – 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 de stockage 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 <code>--disable-content-trust=false</code> lors de l'utilisation des commandes <code>build</code> , <code>create</code> , <code>pull</code> et <code>run</code> . Nous le verrons en détail dans le chapitre 7.
DOCKER_CONTENT_TRUST_SERVER	L'URL du serveur Notary à utiliser lorsque la variable d'environnement <code>DOCKER_CONTENT_TRUST</code> 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
c7cc1a31d8f0        nginx              "nginx -g 'daemon off..."
```

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.

### 5.1.3 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.

## 5. 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.

### 5.2.1 dockerd

```
dockerd [OPTIONS]
```

C'est la commande permettant de contrôler le démon Docker. Nous l'avons déjà abordée dans le chapitre 3.

Docker utilise des binaires différents pour le démon et le client. Dans les premières versions de docker, il n'y avait qu'un binaire docker et on utilisait alors la commande docker daemon pour contrôler le processus serveur.

Évidemment, dans la plupart des cas, le démon est démarré automatiquement par un gestionnaire de système (comme systemd pour CentOS ou RHEL). Par exemple, sous CentOS, la définition du service est stockée sous /usr/lib/systemd/system/docker.service comme nous l'avons vu au chapitre 3.

La commande dockerd peut prendre en paramètre de très nombreuses options<sup>2</sup> 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 le chapitre 9 ;
- ← 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.

### 5.2.2 docker info / docker system info

```
docker info [OPTIONS]
```

Cette commande permet de visualiser des informations sur la configuration de Docker pour notre système. Elle fournit aussi des informations sur l'état du moteur, comme le nombre de conteneurs, le nombre d'images, les plugins installés, etc.

### 5.2.3 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:          18.06.0-ce
  API version:      1.38
  Go version:       go1.10.3
  Git commit:       0ffa825
  Built:            Wed Jul 18 19:08:18 2018
  OS/Arch:          linux/amd64
  Experimental:     false

Server:
  Engine
    Server:
      Engine:
        Version:          18.06.0-ce
        API version:      1.38 (minimum version 1.12)
        Go version:       go1.10.3
        Git commit:       0ffa825
        Built:            Wed Jul 18 19:10:42 2018
        OS/Arch:          linux/amd64
        Experimental:     false
```

### 5.2.4 docker stats /docker container 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 ID	NAME	CPU %	MEM USAGE / LIMIT
MEM %	NET I/O	BLOCK I/O	PIDS
d48d421909a6	nginx	0.00%	1.359MiB / 1.796GiB
0.07%	648B / 0B	0B / 0B	2

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.

## 5.2.5 docker ps / docker container ls

docker ps [OPTIONS]
---------------------

Cette commande permet de lister les conteneurs actifs (par défaut) ou l'ensemble des conteneurs avec l'option `-a`.

Le modificateur `-q` limite l'affichage de la commande à l'identifiant du conteneur.

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`.

## 5.2.6 docker events / docker system 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
STATUS              PORTS              NAMES
d48d421909a6        nginx              "nginx -g 'daemon of..."   6 hours ago
Up About a minute   0.0.0.0:80->80/tcp    nginx
← docker pause nginx
nginx
← docker unpause nginx
nginx
← docker stop nginx
nginx
← docker start nginx
```

Nous obtiendrons l'affichage suivant (à la condition de lancer la commande sur un autre terminal avant d'effectuer les actions) :

```
$ docker events
2018-08-19T14:46:08.094535218+02:00           container          pause
d48d421909a6a56a17470139c2be2271c289438f554dc0cf89654d8d0a1fdfe1  (image=nginx,  maintainer=NGINX
Docker Maintainers <docker-maint@nginx.
com>,  name=nginx)
2018-08-19T14:46:11.890902106+02:00           container          unpause
d48d421909a6a56a17470139c2be2271c289438f554dc0cf89654d8d0a1fdfe1  (image=nginx,  maintainer=NGINX
Docker Maintainers <docker-maint@nginx.
com>,  name=nginx)
```

```

2018-08-19T14:46:15.506755541+02:00           container          kill
d48d421909a6a56a17470139c2be2271c289438f554dc0cf89654d8d0a1fdfe1 (image=nginx, maintainer=NGINX
Docker Maintainers <docker-maint@nginx.com>, name=nginx, signal=15)
2018-08-19T14:46:15.699377464+02:00           container          die
d48d421909a6a56a17470139c2be2271c289438f554dc0cf89654d8d0a1fdfe1 (exitCode=0, image=nginx,
maintainer=NGINX Docker Maintainers <docker-maint@nginx.com>, name=nginx)
2018-08-19T14:46:15.780775529+02:00           network          disconnect
8c5d627eefb634c6d95c7df901b0dd2091cbedff8db69df623752642de284f67
  (container=d48d421909a6a56a17470139c2be2271c289438f554dc0cf89654d8d0a1fdfe1, name=bridge,
type=bridge)
2018-08-19T14:46:15.815675320+02:00           container          stop
d48d421909a6a56a17470139c2be2271c289438f554dc0cf89654d8d0a1fdfe1 (image=nginx, maintainer=NGINX
Docker Maintainers <docker-maint@nginx.com>, name=nginx)
2018-08-19T14:46:18.801495635+02:00           network          connect
8c5d627eefb634c6d95c7df901b0dd2091cbedff8db69df623752642de284f67
  (container=d48d421909a6a56a17470139c2be2271c289438f554dc0cf89654d8d0a1fdfe1, name=bridge,
type=bridge)
2018-08-19T14:46:19.080406567+02:00           container          start
d48d421909a6a56a17470139c2be2271c289438f554dc0cf89654d8d0a1fdfe1 (image=nginx, maintainer=NGINX
Docker Maintainers <docker-maint@nginx.com>, name=nginx)

```

On remarquera les événements réseau qui sont implicitement déclenchés par l'arrêt et le démarrage du conteneur.

← 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
```

## 5.2.7 docker inspect

```
docker inspect [OPTIONS] NAME|ID [NAME|ID...]
```

Cette commande permet de récupérer sous un format JSON des métadonnées relatives à un objet Docker (image, conteneur, réseau, volume...). Les données ainsi rendues sont très variées et dépendent de l'objet inspecté. Par exemple, pour un conteneur : configuration des volumes, paramètres de démarrage du conteneur, image, configuration réseau, etc.

## 5.2.8 docker system df

```
docker system df [OPTIONS]
```

Cette commande permet d'obtenir des informations sur l'espace disque utilisé par le démon docker. La valeur de la colonne RECLAIMABLE correspond à l'espace qui sera récupéré suite à une commande docker system prune.

\$ docker system df	TYPE	TOTAL	ACTIVE	SIZE	RECLAIMABLE
	Images	3	2	420.5MB	366.8MB (87%)
	Containers	5	1	2B	0B (0%)
	Local Volumes	1	0	0B	0B
	Build Cache	0	0	0B	0B

## 5.2.9 docker system prune / docker image prune

```
docker system prune [OPTIONS]
docker image prune [OPTIONS]
```

Sans option, la commande `docker system prune` permet de supprimer en une fois tous les conteneurs et réseaux non utilisés ainsi que 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.

On peut y ajouter les options suivantes :

- ← `-a`, qui supprime en plus les images non utilisées ;
- ← `--volumes`, qui supprime les volumes non attachés à un conteneur.

Comme son nom l'indique, `docker image prune` ne supprime que les images avec une logique identique.

## 5. 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 4. Le lecteur pourra donc s'y reporter pour voir leur effet sur divers exemples pratiques.

### 5.3.1 docker start / docker container 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.

### 5.3.2 docker stop / docker container stop

```
docker stop [OPTIONS] CONTAINER
```

Cette commande, qui permet de stopper un conteneur, a aussi déjà été étudiée dans le chapitre 4. Elle accepte notamment une option `-t` qui permet de spécifier un nombre de secondes à attendre avant de tenter un `kill`.

### 5.3.3 docker kill / docker container kill

```
docker kill [OPTIONS] CONTAINER
```

Cette commande, déjà évoquée au chapitre 4, permet de forcer l'arrêt d'un conteneur (un peu à la manière du célèbre `kill -9` pour un processus Unix).

### 5.3.4 docker restart / docker container 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.

### 5.3.5 docker pause et docker unpause / docker container pause et docker container 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.

### 5.3.6 docker rm / docker container 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.

### 5.3.7 docker wait / docker container 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.

### 5.3.8 docker update / docker container 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.
```

### 5.3.9 docker create et docker run / docker container create et docker container 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 6](#)) :

```
$ 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 4, et que nous reverrons plus en détail dans le chapitre 6 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 6,
- ✓ les paramètres relatifs à la gestion du réseau (qui seront étudiés dans le chapitre 8) 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 spécifié dans le Dockerfile. Nous verrons le but de cette commande dans le chapitre 6 ;
- ← `-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 7) ;
- --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 étudierons dans le chapitre 10 ;
- --name, vu dans le chapitre 4, 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).

```

## ← 4 INTERACTIONS AVEC UN CONTENEUR DÉMARRÉ

Les diverses commandes de ce paragraphe permettent d'interagir avec un conteneur démarré.

### 5.4.1 docker logs / docker container 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`<sup>3</sup> ;
- ← 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 miniscript 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              NAMES
STATUS              PORTS              NAMES
cdb30aee1ff6        php                "docker-php-entrypoi..."   About a minute ago
Up About a minute          loop
```

La commande `logs` va permettre de visualiser les logs du conteneur :

```
- docker logs --tail 4 loop
Log something every 2 sec
```

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, notamment des pilotes comme `fluentd`<sup>4</sup> 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
Aug 19 15:04:59 localhost bee47dd5c607[2459]: Log something every 2 sec
Aug 19 15:05:01 localhost bee47dd5c607[2459]: Log something every 2 sec
Aug 19 15:05:03 localhost bee47dd5c607[2459]: Log something every 2 sec
Aug 19 15:05:05 localhost bee47dd5c607[2459]: Log something every 2 sec
Aug 19 15:05:07 localhost bee47dd5c607[2459]: Log something every 2 sec
Aug 19 15:05:09 localhost bee47dd5c607[2459]: Log something every 2 sec
```

## 5.4.2 docker exec / docker container 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).

Launchons par exemple un conteneur NGINX (que nous avons déjà vu dans le chapitre 4) 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 :

```
| ~ docker exec -t -i webserver  
/bin/bash root@e80fd51f2119:/# ls  
bin boot dev etc home lib lib64 media mnt opt proc 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.

#### 5.4.3 docker attach / docker container 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).

Launchons 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 webserver172.17.0.1 - - [19/Aug/2018:13:07:52 +0000] "GET /  
HTTP/1.1" 200 612 "-"  
"Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0" "-"  
2018/08/19 13:07:52 [error] 6#6: *1 open() "/usr/share/nginx/html/favicon.ico"  
failed (2: No such file or directory), client: 172.17.0.1, server: localhost,  
request: "GET /favicon.ico HTTP/1.1", host: "localhost:8000"
```

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.

#### 5.4.4 docker rename / docker container rename

```
docker rename OLD_NAME NEW_NAME
```

Cette commande, comme son nom l'indique, permet de renommer un conteneur.

#### 5.4.5 docker cp / docker container 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.

#### 5.4.6 docker diff / docker container 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.

#### 5.4.7 docker top / docker container 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
```

#### 5.4.8 docker export / docker container 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<sup>5</sup>.

#### 5.4.9 docker port / docker container 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
```

### ← 5 COMMANDES RELATIVES AUX IMAGES

Nous allons maintenant effectuer un recensement des commandes qui permettent de manipuler des images Docker.

#### 5.5.1 docker build / docker image build

```
docker build [OPTIONS] PATH | URL
```

Il s'agit d'une commande que nous avons abordée sommairement à la fin du chapitre 4 et que nous allons utiliser intensivement dans le [chapitre 6](#).

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 dépôts GitHub)<sup>6</sup> ;  
\$ -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).

← noter que la commande accepte aussi les mêmes options de configuration des ressources que pour les commandes run ou create.

#### 5.5.2 docker commit / docker container 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.

### 5.5.3 docker history / docker image 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.

### 5.5.4 docker images / docker image ls

```
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.

### 5.5.5 docker rmi / docker image rm

```
docker rmi [OPTIONS] IMAGE [IMAGE...]
```

Cette commande permet d'effacer une image du registre local.

### 5.5.6 docker save et docker load / docker image save et docker image 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
```

### 5.5.7 docker import / docker image 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.

## 5. 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/>), Amazon (<https://aws.amazon.com/ecr/>) et Azure (<https://azure.microsoft.com/en-us/services/container-registry/>) ont leurs propres offres de registry de conteneur.

### 5.6.1 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.

### 5.6.2 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).

### 5.6.3 docker push / docker image push

```
docker push [OPTIONS] NAME[:TAG]
```

Cette commande permet d'importer une image du cache local dans un registry.

### 5.6.4 docker pull / docker image 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.

### 5.6.5 docker search

```
docker search [OPTIONS] NAME[:TAG]@DIGEST
```

Cette commande permet de rechercher une image dans le Docker Hub. La recherche se fait sur le nom de l'image. Il est possible d'appliquer un filtre comme par exemple "is-official=true" si on ne recherche que les images officielles maintenues par Docker Inc.

```
$ docker search --filter "is-official=true" php
NAME          DESCRIPTION           STARS          OFFICIAL          AUTOMATED
php           While designed for 3711          [OK]
```

php-zendserver	...	Zend Server - the i...	151	[OK]
adminer	Database managemen...		134	[OK]

## 5.6.6 docker tag / docker image 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.

## 5.7 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 le chapitre 8 ;
- ← les volumes nommés qui permettent de s'affranchir de la technique des *data containers* dont nous verrons des exemples dans les chapitres 8, 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.

### 5.7.1 Les commandes docker network

Tableau 5.2 – Commandes docker network

Commande	Objet
<code>docker network create</code>	Permet de créer un réseau Docker. Cette commande prend notamment en paramètre <code>--driver</code> qui permet de spécifier le type de réseau souhaité (par défaut <code>bridge</code> ).
<code>docker network connect</code>	Cette commande permet de connecter un conteneur à un réseau.
<code>docker network disconnect</code>	Cette commande permet de déconnecter un conteneur d'un réseau.
<code>docker network inspect</code>	Une fonction d'inspection du réseau dont nous verrons l'utilité dans le chapitre 8.
<code>docker network ls</code>	Une commande qui liste les réseaux disponibles. Par défaut, trois réseaux sont systématiquement définis : <code>none</code> , <code>host</code> et <code>bridge</code> .
<code>docker network rm</code>	La commande qui permet de détruire des réseaux existants (sauf évidemment nos trois réseaux prédéfinis).
<code>docker network prune</code>	Supprime les réseaux qui ne sont utilisés par aucun conteneur.

### 5.7.2 Les commandes docker volume

Tableau 5.3 – Commandes docker volume

Commande	Objet
<code>docker volume create</code>	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 <code>--driver</code> 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).
<code>docker volume inspect</code>	Une commande pour visualiser des métadonnées relatives à un volume.
<code>docker volume ls</code>	La commande qui permet de lister les volumes disponibles.
<code>docker volume rm</code>	La commande qui permet d'effacer des volumes. Attention, une fois un volume détruit, les données qui lui sont associées sont perdues définitivement. Il n'est cependant pas possible d'effacer un volume utilisé par un conteneur.
<code>docker volume prune</code>	Supprime les volumes qui ne sont référencés par aucun conteneur.



Ce chapitre 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 4.

- 
1. CLI (*command line interface*) : la ligne de commande.
  2. <https://docs.docker.com/engine/reference/commandline/dockerd/>
  3. [https://hub.docker.com/\\_/php/](https://hub.docker.com/_/php/)
  4. <http://www.fluentd.org/>
  5. <https://runc.io/>
  6. Par exemple, la société Ingensi fournit une image Play framework prête à l'emploi :  
<https://hub.docker.com/r/ingensi/play-framework/>

# 6

## Les instructions Dockerfile

### Objectif

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 7.

### 6. 1 LES MODÈLES D'INSTRUCTION

#### 6.1.1 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 6.1 – 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".

#### 6.1.2 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 :

- ← Le PID du processus démarré par le point d'entrée d'un conteneur (instructions CMD et/ou `ENTRYPOINT`) est le 1.
- ← Lorsqu'un conteneur actif est arrêté par la commande `docker stop`, 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).
- ← Lorsque le binaire `/bin/sh` reçoit un signal lui demandant de s'arrêter, il s'arrête proprement, mais ne retransmet pas 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 :

- ← `/bin/sh` avec le PID 1 (car représentant le point d'entrée du conteneur).
- ← `ping` avec un autre PID.

Si on souhaite arrêter le conteneur (`docker stop`), le résultat sera le suivant :

- ← `/bin/sh` s'arrêtera proprement car il a le PID 1.
- ← `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 est disponible via le lien suivant : <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 6.2 décrit les règles d'utilisation des formats.

**Tableau 6.2 – Règles d'utilisation des formats**

Contexte d'utilisation	Format
Cas de tests	terminal*
Commande se terminant directement après son exécution (par exemple : ls ou	terminal*

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.

### 6.1.3 Les commentaires

Un fichier Dockerfile peut contenir des commentaires, c'est-à-dire des lignes qui ne seront pas 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
```

## 6. 2 LES INSTRUCTIONS D'UN DOCKERFILE

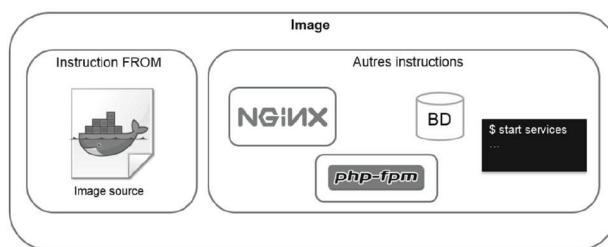
Cette section décrit les instructions principales disponibles pour un fichier Dockerfile. L'objectif ici est de fournir les connaissances nécessaires à l'élaboration d'un Dockerfile afin d'éviter les erreurs usuelles. Pour cela, de nombreux principes sont illustrés par des exemples.

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.

### 6.2.1 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 6.1 – Décomposition d'une image Docker



La figure 6.1 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>] [AS <name>]
FROM <image>[@<digest>] [AS <name>]
```

<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.

```
image ls - digests. 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  
--> cfeef3a2fa477  
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 cfeef3a2fa477 et la deuxième 5b7f2d5028bd.

Démarrons un conteneur avec la première image :

```
| ~ docker run  
| cfeef3a2fa477 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.

## 6.2.2 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"]
```

Il est possible de changer le shell par défaut avec l'instruction SHELL, que nous verrons dans le prochain chapitre.

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 aisés 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
```

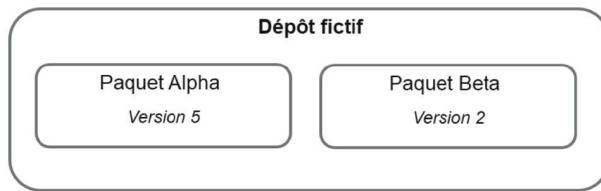
Il est important de ne pas utiliser l'instruction seule RUN yum update -y, car un problème de cache Docker peut survenir, entraînant une incohérence d'images. La prochaine section illustre ce problème

### ← **Quand le cache s'en mêle**

Comme expliqué précédemment, l'utilisation de la commande `RUN yum update -y` peut entraîner un problème d'incohérence.

Pour l'illustrer, nous utiliserons un dépôt fictif de paquets, illustré par l'image suivante :

Figure 6.2 – 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 6.3 – Dépôt fictif plus ancien



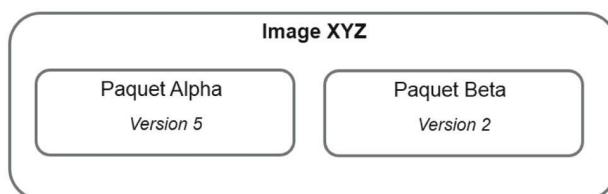
← noter que seuls les paquets utiles à l'exemple sont représentés, tout autre élément de l'image étant volontairement ignoré afin de faciliter la compréhension.

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 :

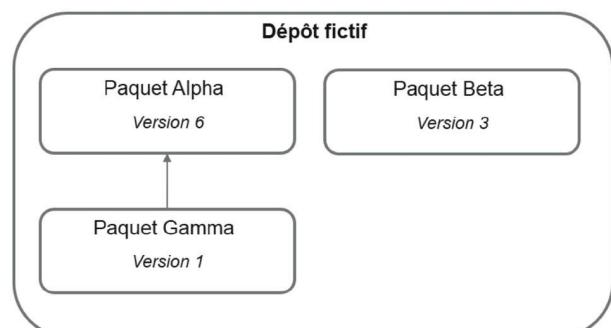
Figure 6.4 – Image basée sur le dépôt le plus récent

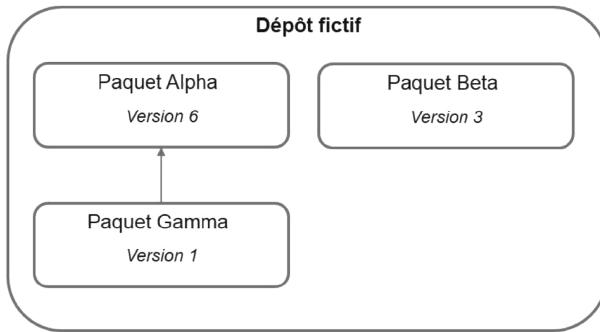


L'image XYZ contient donc logiquement le paquet Alpha en version 5, car il a été mis à jour suite à la commande `RUN yum update -y`.

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 6.5).

Figure 6.5 – Mise à jour du dépôt



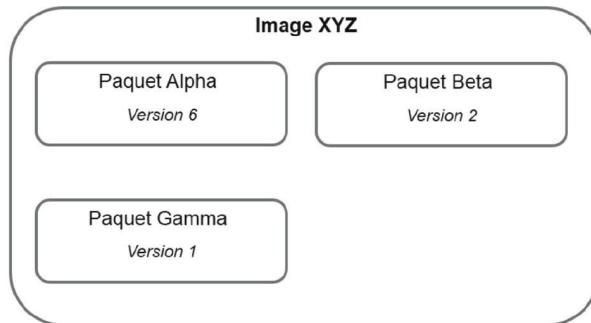


Nous souhaitons maintenant ajouter le paquet Gamma à l'image XYZ, et nous modifions donc le Dockerfile ainsi :

```
FROM AB
RUN yum update -y
RUN yum install -y gamma
```

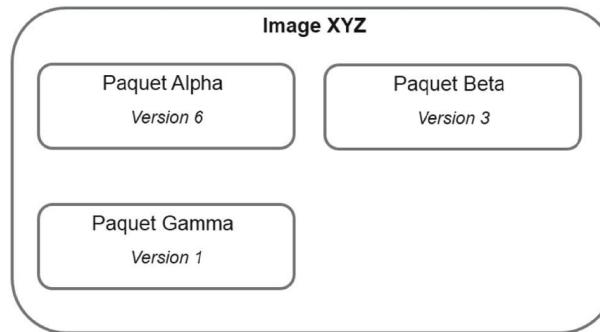
En construisant à nouveau le Dockerfile sur la même machine, la commande `RUN yum update -y` n'est pas exécutée, mais lire depuis le cache (car elle n'a pas été modifiée) ; ainsi, à ce stade, ni le paquet Alpha ni le paquet Beta ne sont mis à jour. Alpha sera mis en version 6 grâce à l'instruction en figure 6.6 (où seuls les paquets nécessaires à Gamma sont mis à jour), Beta restant lui en version 2.

Figure 6.6 – 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 version 3 après construction ; ainsi, le résultat devient incohérent d'une machine à l'autre :

Figure 6.7 – Seconde image : même Dockerfile, résultat différent



### ← Conclusion

Lorsque nous souhaitons installer un paquet, il faut toujours s'assurer que les paquets existants sont 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 \
```

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 sont orientés 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).

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 et imaginons que nous souhaitons surcharger localhost par "[google.com](http://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
```

## 6.2.4 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"]
```

```
| 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            TEST
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      COMMAND
                  START TIME
root              1 0.2 0.0          17176    1088 ? Ss ping google.com
                           09:26 0:00
root              6 0.0 0.0          35888     1452 ? RS ps aux
                           09:26 0:00
```

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
```

#### 4. Format terminal

Reprendons 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            TEST
STATUS              PORTS              NAMES
```

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              NAMES
STATUS              PORTS
c844052a05e9        ping-google-shell   "/bin/sh -c 'ping goo'"   15 seconds ago
Up 14 seconds          test
```

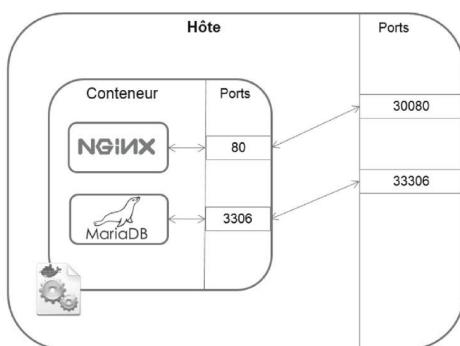
La commande en cours est cette fois-ci différente : `/bin/sh -c 'ping google.com'`.

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.

### 6.2.5 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 6.8 – Mappage de ports entre un conteneur et un hôte



La figure 6.8 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 au processus MariaDB du conteneur.

Le modèle de l'instruction `EXPOSE` est :

```
EXPOSE <port> [<port>/<protocol>...]
```

Par exemple :

```
EXPOSE 80 22
EXPOSE 80/TCP
```

```

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 priviléges ;
- ✓ 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	PORTE	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 :

- ← D'une part, spécifier manuellement quels ports doivent être mappés parmi ceux exposés dans le Dockerfile.
- ← D'autre part, spécifier manuellement les ports hôtes à utiliser afin de les mapper avec les ports

```
| -> 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 :

- ← D'une part, spécifier manuellement quels ports doivent être mappés parmi ceux exposés dans le Dockerfile.
- ← 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`.

Reprendons 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 ~]$
```

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 [--chown=<user>:<group>] <src>... <dest>
ADD [-chown=<user>:<group>] [<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 l'adresse suivante : <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.

L'option --chown=<user>:<group>, disponible uniquement pour la fabrication d'images Linux, permet de changer les permissions sur les fichiers qui seront copiés. En effet, tous les fichiers sont créés par défaut avec le UID et GID 0 si cela n'est pas spécifié explicitement.

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\* (ex. : test1 ou test1lambda) 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 7](#)).

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
```

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 7](#)).

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 sauvégardé 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 paramétriser 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.

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](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.

← première vue, il s'agit d'une fonctionnalité utile, et pourtant elle constitue deux désavantages majeurs :

- \$ 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).
- \$ 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` (déttaillée dans la prochaine section) au lieu de l'instruction `ADD`. En effet, `COPY` ne décomprime 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.

## 6.2.7 COPY

| ...

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.

### 6.2.7 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 décomprime pas automatiquement une archive tar compatible.

Les modèles possibles sont :

```
COPY [--chown=<user>:<group>] <src>... <dest>
COPY [--chown=<user>:<group>] [<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 Dockerfiles 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
```

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
--> 5dcbb5162ec7c
Removing intermediate container 32e19bb0e2e0
Step 4 : RUN tar xzf /tmp/test2.tar.gz
--> Running in a06fcfa73ed6c
--> d33c13e5bbc0
Removing intermediate container a06fcfa73ed6c
Successfully built d33c13e5bbc0
```

Là encore, le cache n'est pas utilisé : dans le premier Dockerfile, la copie de `test1.tar.gz` étant placée en début de fichier, toutes les instructions suivantes seront rejouées ; dans le deuxième Dockerfile, la constatation est la même (rappelons-nous que dès qu'un 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
```

```

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.

## 6.2.8 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.

La figure 6.9 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.

Figure 6.9 – Point de montage dans un conteneur

```
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 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
[{"Type": "volume", "Name": "1d48770c47565b2052f783f7cc41cf44aed2805fb787033598c78903c38dc", "Source": "/var/lib/docker/volumes/1d48770c47565b2052f783f7cc41cf44aed2805fb787033598c78903c38dc/_data", "Destination": "/tmp/data", "Driver": "local", "Mode": "", "RW": true, "Propagation": ""}]
```

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/1d48770c ...903c38dc/_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/1d48770c ...903c38dc/_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
```

noter que le volume dans l'hôte est aussi supprimé. Ensuite, démarrons le conteneur avec l'option -v :

Listons une nouvelle fois le volume :

```
| docker exec volume-conteneur ls /tmp/data  
helloTest  
sudo ls /var/lib/docker/volumes/1d48770c ...903c38dc/_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/1d48770c ...903c38dc/_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
```

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  
[{"Type": "bind", "Source": "/var/home/vagrant/data", "Destination": "/tmp/data", "Mode": "", "RW": true,  
"Propagation": "rprivate"}]
```

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.

### 6.3.1 Généralités

Pour les instructions CMD et ENTRYPPOINT, 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 :

```
FROM  
ARG  
ENV, LABEL  
VOLUME  
RUN, COPY, WORKDIR  
EXPOSE  
USER  
ONBUILD  
CMD, ENTRYPPOINT
```

Dans toute instruction, nous n'utiliserons pas la commande sudo.

### 6.3.2 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.

### 6.3.3 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 7](#)).

### 6.3.4 CMD et ENTRYPPOINT

Nous utiliserons en priorité l'instruction ENTRYPPOINT (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 ENTRYPPOINT) ou complètement (dans ce cas l'instruction ENTRYPPOINT ne sera pas utilisée).

Un fichier Dockerfile ne contiendra au maximum qu'une seule instruction CMD et ENTRYPPOINT.

### 6.3.5 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.

### 6.3.6 ADD et COPY

Nous utiliserons systématiquement l'instruction COPY au lieu de ADD, ceci afin d'éviter des

l'instruction WORKDIR (voir le [chapitre 7](#)).

#### 6.3.4 CMD et ENTRYPPOINT

Nous utiliserons en priorité l'instruction ENTRYPPOINT (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 ENTRYPPOINT) ou complètement (dans ce cas l'instruction ENTRYPPOINT ne sera pas utilisée).

Un fichier Dockerfile ne contiendra au maximum qu'une seule instruction CMD et ENTRYPPOINT.

#### 6.3.5 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.

#### 6.3.6 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.

#### 6.3.7 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 \
```

#### 6.3.8 ENV, LABEL et ARG

Ces commandes seront expliquées en détail dans le chapitre 7.

Afin de limiter le nombre de couches, on regroupera les variables d'environnement (ENV) définies 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'exéutable 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:-valeurParDefaut}
```



## Docker avancé

### Objectif

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 instructions et des commandes du CLI que nous n'avons pas encore étudiées. Certaines sont d'un usage courant (comme ENV ou LABEL), d'autres sont relativement moins usitées mais peuvent rendre d'étonnantes services.

### 7. 1 VARIABLES D'ENVIRONNEMENT ET CONTENEURS : 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 « = » doit 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.

## 7. 2 MÉTA-INFORMATION ET IMAGES : LABEL

Les premières versions de Docker ne comprenaient qu'une instruction permettant d'ajouter de la métा-information à une image : MAINTAINER. Celle-ci permettait tout simplement de spécifier l'auteur d'une image.

Aujourd'hui cette instruction a été dépréciée au profit d'une solution beaucoup plus générique : les *labels*.

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.

On pourra noter qu'il est aussi possible d'ajouter de la métा-information au moment de la construction d'une image :

```
| docker build -t exempleimage --label license=GPL .
```

Cette option est très souvent utilisée pour enregistrer dans l'image des informations liées au contexte de build. Nous aborderons ce sujet dans le chapitre 10 de la prochaine partie consacrée au déploiement d'application.

## 7. 3 PARAMÉTRER LE BUILD D'UNE IMAGE

Nous avons appris l'instruction `build` dans le chapitre 5. Les instructions Dockerfile que nous allons étudier dans ce paragraphe visent à influencer la manière dont l'image va être construite.

### 7.3.1 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 renvoyée 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.

### 7.3.2 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 sauvegardées 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 renvoyée 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, ou « 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.

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
```

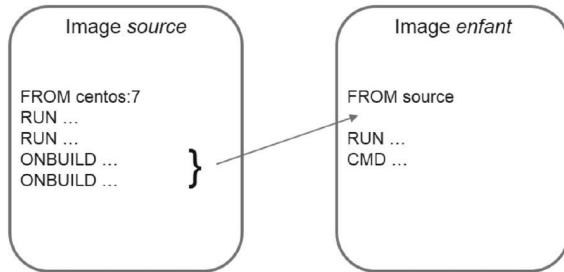
### 7.3.3 ONBUILD

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 7.1 – Exécution d'instructions ONBUILD dans une image enfant



La figure 7.1 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.

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 :

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"}]
```

Nous constatons bien que les trois instructions ONBUILD sont présentes dans OnBuild.

Après la création de la métadonnée, le constructeur Docker a terminé son travail pour l'instruction concernée.

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'elles échoue, alors c'est l'instruction FROM qui échouera.

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.

## 4 MODIFIER LE CONTEXTE SYSTÈME AU COURS DU BUILD

Une image hérite des caractéristiques système de son image de base. Celle-ci impose généralement un certain nombre de contraintes contextuelles (comme le shell courant), qu'il est néanmoins possible d'influencer par quelques instructions simples.

### 7.4.1 SHELL

```
SHELL ["executable", "parameters"]
```

L'instruction SHELL permet de changer le terminal utilisé par défaut. Comme nous l'avons vu précédemment, sous Linux, il s'agit de `["/bin/sh", "-c"]` et, sous Windows, de `["cmd", "/S", "/C"]`.

Elle peut être présente plusieurs fois dans le Dockerfile, chaque instruction SHELL annulant l'effet des utilisations précédentes et affectant les commandes suivantes.

L'intérêt principal de cette instruction est visible sous Windows, où il y a deux terminaux disponibles : cmd et powershell.

Quelques exemples pour mieux comprendre son fonctionnement :

```
FROM microsoft/windowsservercore

Exécuté comme cmd /S /C echo default
RUN echo default

Exécuté comme cmd /S /C powershell -command Write-Host default
RUN powershell -command Write-Host default

Exécuté comme powershell -command Write-Host hello
SHELL ["powershell", "-command"]
RUN Write-Host hello

Exécuté comme cmd /S /C echo hello
SHELL ["cmd", "/S", "/C"]
RUN echo hello
```

### 7.4.2 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 Dockerfiles 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 groupe 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"
```

Reprendons 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 superutilisateur (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 devrait pas contenir d'exécution de commandes avec un superutilisateur, autrement dit il ne contiendra pas de sudo.

#### 7.4.3 STOP SIGNAL

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 7.1 – 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 interprétera 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.

## 7. 5 AUTO-GUÉRISON (SELF HEALING)

Sans entrer à ce stade dans les détails de l'opération d'une application à base de conteneurs, il nous semble intéressant de présenter deux fonctions utiles pour augmenter la résilience d'un conteneur. Il existe en effet deux cas dans lesquels il est possible (et même nécessaire) de prévoir des actions automatiques pour agir sur un conteneur ne fonctionnant pas correctement :

lorsque le processus principal du conteneur meurt, ce qui entraîne aussi la mort involontaire du conteneur ;

lorsque le processus ne fonctionne plus correctement mais ne s'éteint pas de lui-même.

### 7.5.1 --restart

Le modificateur `--restart` est disponible pour les commandes CLI Docker `run`, `start` et `update`. Il permet de modifier la politique de redémarrage automatique du conteneur dans le cas où le processus racine se termine.

Le conteneur ci-dessous est programmé pour mourir automatiquement après 5 secondes (grâce à l'instruction Linux `sleep`) :

```
| docker run -d --restart=always --name=sepuku centos:7 /bin/sh -c "sleep 5"
```

Effectivement, après 5 secondes :

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	POR	TS	NAMES
d933421eb68e	centos:7	"/bin/sh -c 'sleep 12 seconds ago 5'"	
Exited (0) 7 seconds ago			sepuku

Si maintenant nous ajoutons le paramètre `restart` avec l'option `always`, le comportement n'est plus du tout le même. Le conteneur meurt toujours après 5 secondes mais est automatiquement immédiatement redémarré. Son `STATUS` ne dépasse donc jamais 5 secondes.

Généralement on utilise plutôt, et plus prudemment, l'option `on-failure` avec un nombre de redémarrages prédéfini, par exemple trois. Cette option ne fonctionnerait pas avec notre exemple précédent parce que `sleep` se termine « normalement », c'est-à-dire avec un code de retour 0. L'option `on-failure` est donc dédiée à redémarrer une application qui s'est arrêtée involontairement.

Mais que faire lorsque l'application fonctionne en apparence normalement (processus principal vivant) mais ne fournit plus le résultat attendu ?

## 7.5.2 HEALTHCHECK

L'instruction `HEALTHCHECK` offre la possibilité de laisser le moteur docker moniter l'état de santé d'un conteneur. Il est par exemple possible d'imaginer une application qui continuerait à fonctionner (du point de vue du processus principal du conteneur) alors que celle-ci n'aurait plus un comportement normal.

Imaginons un simple conteneur Python utilisant le framework Flask<sup>1</sup>, donc le code serait :

```
from flask import Flask
from flask import request
from datetime import datetime

app = Flask(__name__)
calls=[]

@app.route('/')
def index():
    return 'Call count : <ul><li>{}</li></ul>'.format('</li><li>'.join(calls))

@app.route('/health')
def health():
    global calls
    dt = datetime.now()
    calls.append("from {} at {}".format(request.remote_addr, dt.strftime("%H:%M:%S")))
    return 'OK'

if __name__ == '__main__':
    app.run(host="0.0.0.0")
```

Cette petite application web expose deux API HTTP :

`/health` : qui incrémente une liste en mémoire des appels à cette API (en stockant l'heure et l'IP appelante) ;  
`/` : qui affiche une page listant les appels à l'API `/health`.

Voici maintenant une image très simple qui va exécuter cette application web :

```
FROM python:3.7.0
RUN pip install flask==1.0.2
COPY app.py app.py
HEALTHCHECK --interval=10s \
```

```
CMD curl -f http://localhost:5000/health || exit 1  
ENTRYPOINT python app.py
```

Nous utilisons une image de base Python dans laquelle nous installons le framework Python Flask et nous copions le code source de notre application (placé dans le fichier `app.py`).

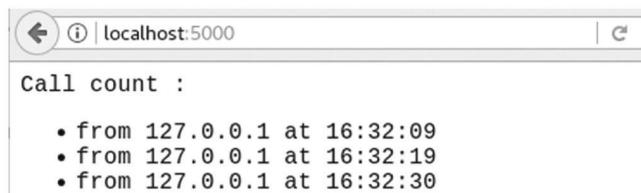
Vous notez l'usage de l'instruction `HEALTHCHECK`. Celle-ci prend dans ce cas un paramètre `interval`, qui indique la fréquence de vérification de l'état de santé. Pour faire ce contrôle de santé, nous utilisons une instruction `CMD` effectuant un appel à l'URL `/health`. Dans cette première implémentation, pas de problème, le code de retour sera systématiquement 0 (succès).

Lançons notre application pour vérifier son fonctionnement :

```
$ docker build -t health-flask .  
$ docker run -d --rm -p 5000:5000 --name=health-flask health-flask
```

Un appel à `http://localhost:5000` nous montre (après quelques secondes de fonctionnement) que Docker appelle automatiquement l'application toutes les 10 secondes comme prévu :

Figure 7.2 – Contrôle automatique de l'état de santé d'un conteneur



Nous allons maintenant simuler un dysfonctionnement de l'application. Pour ce faire il suffit d'ajouter « `,500` » à la ligne suivante de notre code Python Flask :

```
| return 'OK', 500
```

Cette légère modification va provoquer (via `curl`) un code de retour « `1` », soit « `unhealthy` ». Voyons comment Docker va réagir à cette situation.

```
$ docker stop health-flask  
$ docker build -t health-flask .  
$ docker run -d --rm -p 5000:5000 --name=health-flask health-flask
```

Après 30 secondes, soit trois tentatives d'appels à l'API `/health`, l'effet est visible :

```
$ docker ps  
CONTAINER ID        IMAGE           COMMAND       CREATED          STATUS          PORTS          NAMES  
61ec3d743fd4        health-flask     "/bin/sh -c 'python ...'" 34 seconds ago  
Up 33 seconds (unhealthy)   0.0.0.0:5000->5000/tcp health-flask
```

Il est nécessaire d'attendre 30 secondes parce que le nombre de tentatives infructueuses est fixé par défaut à trois (soit 3 fois les 10 secondes que nous avons configurées). Il est évidemment possible de changer cette configuration via le modificateur `--retries` de l'instruction `HEALTHCHECK`. On dispose aussi des paramètres :

- `--timeout` : le délai accordé à la commande `CMD` pour se terminer ;
- `--start-period` : le délai, une fois le conteneur actif, d'attente avant la première vérification (dans le cas où l'application aurait besoin de temps pour s'initialiser).

Que faire, maintenant que nous avons un moyen de détecter le problème, pour y remédier ?

L'option la plus simple est de déclencher un redémarrage du conteneur fautif en utilisant l'information produite par `docker events` (voir [chapitre 5](#)). L'instruction suivante permet par exemple de lister les identifiants des conteneurs ayant déclaré un état `unhealthy` au cours des 10 dernières minutes :

```
docker events --since 10m --until 10s --filter "event=health_status: unhealthy" --format  
'{{.ID}}'
```

On pourrait donc, par exemple, envisager un script exécuté toutes les 10 minutes (à l'aide d'un CRON Linux par exemple) qui redémarrerait automatiquement les conteneurs ainsi détectés :

```
#!/bin/bash  
set -e  
CONTAINERS_TO_RESTART=$(docker events --since 10m --until 10s --filter "event=health_status:  
unhealthy" --format '{{.ID}}')  
docker restart $CONTAINERS_TO_RESTART
```

Nous verrons dans les deux derniers chapitres comment un orchestrateur tel que Swarm ou

Kubernetes nous permet de ne pas avoir recours à ce script de « surveillance ».



Nous arrivons au terme de notre partie consacrée à l'apprentissage de Docker. Nous avons, sans être complètement exhaustif, abordé la grande majorité des instructions et commandes Docker, à l'exception des réseaux que nous étudierons par l'exemple dans la prochaine partie.

Dans la prochaine partie, nous allons mettre en pratique Docker pour développer, conditionner et opérer une application à base de conteneurs.

---

1. <http://flask.pocoo.org/>

## QUATRIÈME PARTIE

### Développer, déployer et opérer avec Docker

La partie précédente vous a offert une présentation académique complète de Docker : sa ligne de commande et ses instructions Dockerfile.

Il est maintenant temps d'aborder, par l'exemple, le développement et le déploiement d'une application complète et réaliste dans les problèmes qu'elle soulève.

Nous avons choisi de proposer une application Java simple mais qui nous permettra d'étudier, au prix de quelques digressions pédagogiques, l'ensemble des points à considérer pour l'élaboration d'une solution « conteneurisée ».

Le chapitre 8 présente les différents concepts et outils qu'il est nécessaire de maîtriser pour élaborer notre application exemple dans un cadre professionnel : réseau, sécurité, gestion des volumes, monitoring, etc.

Le chapitre 9 expose les différents types d'architectures qui peuvent être envisagés pour notre application : de l'application mono-conteneur à un déploiement multi-conteneur automatisé et *scalable*.

Le chapitre 10 traite spécifiquement de la mise en place d'un environnement de développement de type CI (Intégration continue).

Dans cette partie, nous nous limiterons volontairement aux architectures n'utilisant pas de solution d'orchestration CaaS (telles qu'abordées au chapitre 2). L'objectif est de comprendre tout ce qu'il est possible de faire avec Docker et seulement Docker. Nous verrons dans la prochaine partie comment notre application exemple peut bénéficier des fonctionnalités de Kubernetes et Swarm pour passer à un stade d'automatisation supérieur.

## « Real-life » Docker : Mettre en place une application complète

### Objectif

Jusqu'à maintenant nous nous sommes limités, pour nos exemples, à des conteneurs simples et autonomes. Il semble évident qu'à de rares exceptions près les applications informatiques modernes sont d'un niveau de complexité notablement supérieur.

L'objectif de ce chapitre est de présenter les concepts et outils qu'il faut nécessairement maîtriser pour utiliser Docker dans un cadre professionnel.

Pour ce faire, nous nous servirons d'une application exemple qui sera utilisée dans les chapitres suivants, y compris dans la dernière partie du livre, consacrée à l'orchestration de conteneurs.



### Compléments en ligne

Le code source et les exemples de ce chapitre sont disponibles sur GitHub.

Veuillez vous reporter à la procédure d'installation du chapitre 3.

## 8. 1 NOTRE APPLICATION EXEMPLE

L'application (présente dans le répertoire `/application-exemple/` du dépôt GitHub) a été conçue pour démontrer le maximum de concepts nécessaires dans un cadre réel. Elle n'est sans aucun doute pas complètement réaliste afin de rester suffisamment simple pour être utilisée dans un ouvrage tel que celui-ci. Néanmoins, sa structure est conforme à ce qu'on rencontre usuellement pour des architectures web classiques.

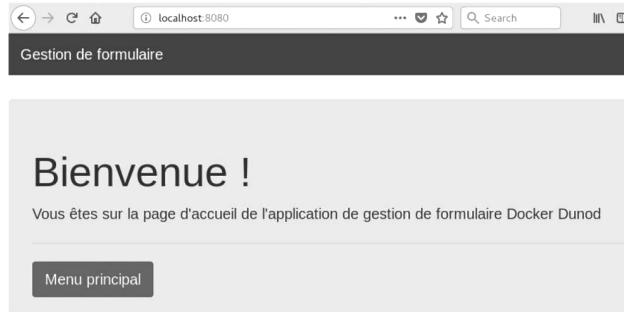
### 8.1.1 Objectif : gestion de formulaires

Nous allons imaginer une application utilisée pour collecter des informations relatives aux communes auprès des préfectures françaises. Fonctionnellement l'application consiste, du point de vue de l'utilisateur :

- à charger (*upload*) des formulaires au format CSV contenant les données collectées ;
- à attendre le traitement de ces fichiers ;
- à visualiser les données qui auront été extraites des formulaires.

L'application se présente sous la forme d'une interface web (figure 8.1).

Figure 8.1 – Page d'accueil



L'utilisateur presse le bouton « Menu principal » et est dirigé, s'il n'est pas encore authentifié, vers une page de login (figure 8.2).

Figure 8.2 – Page de login



L'utilisateur entre ses paramètres d'identification et est redirigé vers le menu principal (figure 8.3) de l'application, qui offre deux fonctions :

- chargement de nouveaux formulaires sous la forme de fichiers ;
- visualisation des données traitées.

Figure 8.3 – Menu principal



En pressant sur « Charger » l'utilisateur est redirigé vers une interface graphique (figure 8.4) proposant de charger un fichier CSV dans ce cas, les données du Vaucluse).

Figure 8.4 – Interface de chargement de fichier



En cas de succès, l'utilisateur est renvoyé au menu principal qui affiche un message de succès (figure 8.5).

Figure 8.5 – Interface de chargement de fichier



L'utilisateur peut ensuite visualiser les fichiers en attente de traitement (figure 8.6).

**Figure 8.6 – Visualisation : fichier en attente de traitement**

This screenshot shows the 'En attente de traitement' section. It contains a message stating 'Le fichiers suivants ont été chargés mais n'ont pas encore été traités.' Below this, a table lists one file: 'vaucluse.csv' with the timestamp '16-09-2018 17:07'.

Après quelques minutes, le fichier disparaît de la liste d'attente et les données peuvent être visualisées (figure 8.7).

**Figure 8.7 – Visualisation : données après traitement**

This screenshot shows the 'Données en base' section. It contains a message stating 'Les données suivants sont issues des fichiers chargés qui ont été traités.' Below this, a table displays four rows of data:

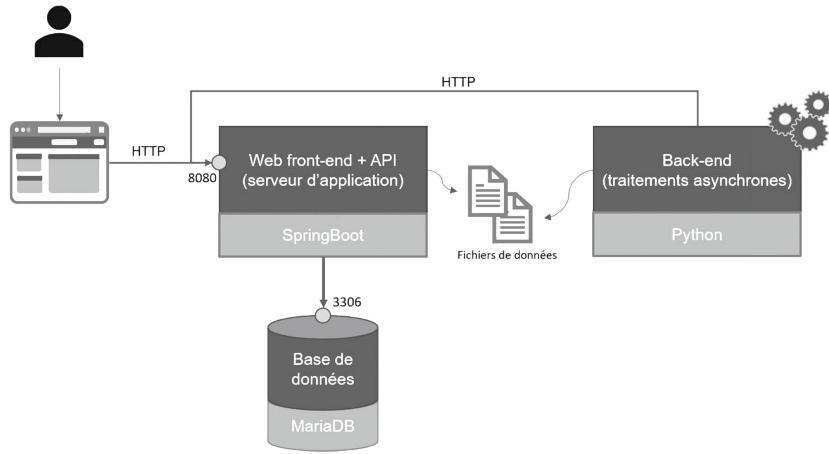
Département	Population totale	Commune la plus peuplée	Commune la moins peuplée	Nombre d'hôtels	Mise à jour
32	172511	Auch	Sérempuy	64	2018-09-09 22:57:11.0
51	565153	Reims	Rouvroy-Ripont	116	2018-09-09 22:51:26.0
68	707709	Mulhouse	Lucelle	262	2018-09-09 22:51:26.0
84	499665	Avignon	Saint-Léger-du-Ventoux	212	2018-09-09 22:51:26.0

### 8.1.2 Architecture logique et technologies

La solution consiste en plusieurs modules (figure 8.8) :

- un serveur d'application qui sert à la fois à afficher l'interface graphique à l'utilisateur (front-end) et à exposer une API REST permettant au back-end de poster les données après traitement des fichiers ;
- un module back-end qui, à une fréquence déterminée, va lire les fichiers, effectuer des calculs et en poster le résultat au front-end ;
- une base de données relationnelle qui sert à stocker les données une fois traitées par le back-end.

**Figure 8.8 – Architecture logique de l'application**



Comme indiqué sur le diagramme, le serveur d'application et le back-end partageront un système de fichiers pour le transfert des formulaires CSV.

### ***Choix technologique pour le serveur d'application***

Pour le serveur d'application, nous utiliserons une application Java développée sur la base du moteur [SpringBoot](#)<sup>1</sup>. Ce choix est motivé par deux facteurs :

- ✓ SpringBoot est un choix très courant pour l'implémentation
- d'application Java

« conteneurisées » ;

Java étant un langage compilé, nous pourrons aborder les problématiques de compilation dans le cadre du chapitre 10.

### ***Choix technologique pour le back-end***

Nous implémenterons le module back-end en Python, qui est un langage de choix pour ce type de traitement. Ceci nous permettra d'aborder le *scheduling* (planification de tâches asynchrones) avec des conteneurs. Le partage de fichier entre le serveur d'application et le module *back-end* nous donnera l'occasion d'étudier la mise place d'un volume associé à plusieurs conteneurs.

### ***Choix technologique pour la base de données***

Nous utiliserons *MariaDB* comme base de données car il existe une image de base officielle. Les outils *MariaDB* sont aussi disponibles dans les dépôts standards Linux.

#### **8.1.3 Hypothèses et simplifications**

Pour les besoins de l'exercice, nous admettrons (même si ce n'est pas le cas) que le traitement des fichiers est un processus long qui justifie la mise en place d'un back-end.

On pourrait discuter de la logique d'employer le même composant pour exposer une interface graphique et une API back-end. Dans une architecture de grande ampleur, ces deux fonctions pourraient ou devraient effectivement être implémentées par deux modules indépendants.

Enfin nous verrons que cette application n'a qu'un seul compte utilisateur. Dans la réalité, il en serait évidemment autrement.

#### **8.1.4 Les connaissances dont nous avons besoin**

Quoique simple en apparence cette architecture va cependant nous permettre, dans le reste de ce chapitre, d'aborder plusieurs problématiques essentielles :

les réseaux ;

la persistance et le partage de fichier : *bind mounts* et volumes ;

la configuration d'application (gestion de paramètres et de mots de passe) ;  
le *monitoring*.

Commençons immédiatement avec les réseaux et Docker : un sujet incontournable !

## 8. 2 LE RÉSEAU AVEC 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*<sup>2</sup>, 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.



### Compléments en ligne

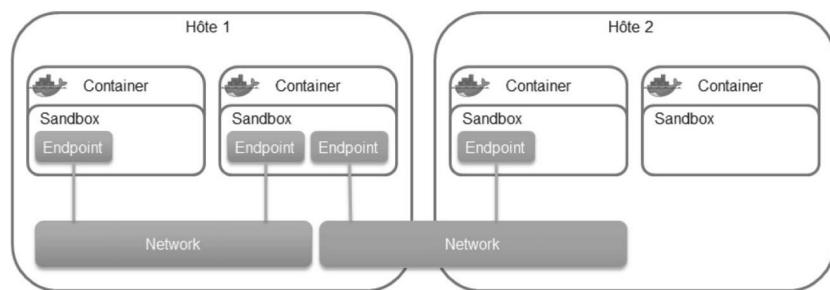
Le code source de ce paragraphe se trouve dans le répertoire /chapitre8/network.

Veuillez vous reporter à la procédure d'installation du chapitre 3.

### 8.2.1 CNM (*Container Network Model*)

Le modèle CNM est une abstraction pour la communication inter-conteneurs (potentiellement déployés sur des hôtes différents).

Figure 8.9 – Le modèle CNM



Elle repose sur trois composants principaux qui permettent de couvrir toutes les topologies de réseaux.

Nous avons :

- le *sandbox* : il contient la configuration réseau du conteneur. Il inclut les interfaces réseau (eth0...), les tables de routage et les configurations DNS. Dans le cas de Docker, il y en a un par conteneur et son implémentation est un *namespace*<sup>3</sup> 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 *nets* ;
- le *network* : c'est un ensemble de *endpoints* qui peuvent communiquer ensemble. Il n'est au final qu'une instanciation d'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 :

- none* : c'est un driver un peu particulier qui signifie « pas de réseau ». Il est là par souci de rétrocompatibilité au niveau de Docker ;
- bridge* : ce driver se base sur un bridge Linux<sup>4</sup>. Il n'est disponible qu'à l'intérieur d'un même hôte ;
- 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 ;
- overlay* : ce driver est pour l'instant le seul qui permette une communication entre plusieurs

hôtes. Nous l'utiliserons dans la dernière partie dans le cadre du chapitre consacré à Swarm ; *macvlan* : un driver récent qui permet d'associer une adresse physique (MAC) à un conteneur pour simuler un attachement physique au réseau. Nous ne nous attarderons pas sur ce cas dont l'usage est moins courant.

Il existe par ailleurs des drivers spécifiques<sup>5</sup> qu'il est possible d'installer via des plugins Docker. Certains offrent des fonctionnalités d'administration, de cryptage, de supervision, etc.

Par volonté de simplicité, nous utiliserons de manière égale les termes *réseau* et *driver*.

Le driver *bridge* est le driver historique de Docker, c'est encore celui qui est utilisé par défaut et c'est donc le premier que nous aborderons.

### 8.2.2 Les réseaux standards de Docker

Pour lister les réseaux disponibles pour nos conteneurs, nous utilisons la commande suivante :

```
$ docker network ls
NETWORK ID          NAME            DRIVER
b2d747bcd553        host            host
8998ee49ca35        bridge          bridge
758827c4904b        none            null
```

Docker est installé par défaut avec trois 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 réseau avec des outils tiers comme *pipework*<sup>6</sup> ou un conteneur temporaire qui fait une action et s'arrête) et nous ne les couvrirons pas dans ce livre ;

*host* : ce 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 réseau historique de Docker, qui permet de connecter plusieurs conteneurs.

### 8.2.3 Le réseau *bridge*

Lançons le script suivant :

```
| $ ./bridge_network/run_bridge_network.sh
```

Celui-ci crée deux conteneurs (*centos-test-1* et *centos-test-2*) utilisant l'image *centos:7* sur le réseau *bridge*.

Inspectons le réseau *bridge* :

```
docker network inspect bridge
[
  {
    "Name": "bridge",
    ...
    "Driver": "bridge",
    "IPAM": {
      ...
      "Config": [
        {
          "Subnet": "172.17.0.0/16",
          "Gateway": "172.17.0.1"
        }
      ]
    }
  }
]
```

```

        ],
        ...
        "ConfigOnly": false,
        "Containers": {
      "5ebcbb36c42e0d33ca7bd5b3820699dc7bbb87d1a1da713388e409bfc6ed14ed": {
        "Name": "centos-test-1",
        ...
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
      },
      "d47d616164aba0d7b2b0487982db8cdf9193302b684188cd945de5c79f848824": {
        "Name": "centos-test-2",
        ...
        "IPv4Address": "172.17.0.3/16",
        "IPv6Address": ""
      }
    },
    ...
  }
]

```

Les adresses assignées pourraient être différentes sur votre ordinateur si vous avez déjà des conteneurs attachés au réseau *bridge*.

Nous voyons que ce réseau est bien de type *bridge* et qu'il dispose des adresses IP 172.17.0.0/16. L'adresse IP 172.17.0.1 est allouée à la *gateway*.

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 et de vérifier que l'adresse IP qui lui est allouée correspond à la *gateway* du réseau *bridge* :

```
$ ifconfig docker0
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
  inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
...
```

Cette interface *docker0* est en effet connectée à l'interface réseau de l'hôte et permet aux conteneurs de communiquer entre eux ainsi qu'avec l'extérieur.

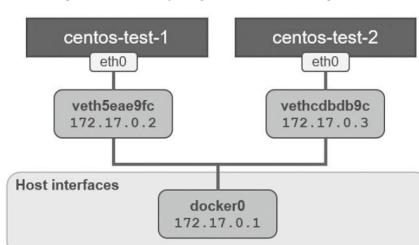
Vous pouvez lister toutes les interfaces virtuelles connectées sur notre interface *docker0* avec la commande suivante :

```
$ sudo brctl show docker0
bridge name          bridge id           STP enabled     interfaces
docker0              8000.02420c8206df   no            veth5eaefc
                                         vethcdbdb9c
```

Au démarrage d'un conteneur, Docker crée une interface virtuelle sur l'hôte avec un nom du type *vethxxxxx* et assigne une adresse IP libre. Cette interface est ensuite connectée à l'interface *eth0* de notre conteneur.

Voici la topologie ainsi créée :

Figure 8.10 – Topologie du réseau *bridge* standard



Nos conteneurs sont sur le même réseau *bridge*. Peuvent-ils néanmoins communiquer entre eux ou avec l'hôte ?

Connectons-nous dans l'un de nos conteneurs et tentons de contacter notre autre conteneur (dont nous connaissons l'adresse IP : 172.17.0.3) :

```
| docker exec -ti centos-test-1 /bin/bash  
# ping 172.17.0.3  
PING 172.17.0.3 (172.17.0.3): 56 data bytes  
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.113 ms
```

Les conteneurs peuvent se voir en effet mais peuvent-ils communiquer librement entre eux ?

Installons quelques utilitaires pour le démontrer :

```
| yum install -y nmap  
| nc -l 5000
```

La commande ci-dessus crée un processus écoutant sur le port TCP 5000. Voyons maintenant si nous pouvons y accéder depuis notre autre conteneur :

```
| docker exec -ti centos-test-2 /bin/bash  
# yum install -y telnet  
# telnet 172.17.0.2 5000  
Trying 172.17.0.2...  
Connected to 172.17.0.2.  
Escape character is  
'^]'. Et hop un message
```

Une fois la connexion telnet ouverte ci-dessous vous pouvez entrer une chaîne de caractères puis la touche « entrée » (ici « Et hop un message »). Comme vous pouvez le constater la chaîne de caractères apparaît sous la commande nc précédente. La communication entre les conteneurs attachés au réseau *bridge* est donc ouverte !

Le script suivant va supprimer les conteneurs précédemment créés :

```
| $ ./bridge_network/clean_bridge_network.sh
```

#### 8.2.4 Créer un réseau personnalisé de type *bridge*

Nous devons ici bien séparer deux types de réseaux *bridge* Docker :

le réseau prédéfini qui se nomme *bridge* (présenté précédemment) 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*.

Quelles sont donc les différences ?

**Tableau 8.1** – Différences entre les deux réseaux *bridge*

Réseau <i>bridge</i> via docker0	Réseau de type <i>bridge</i>
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.
Ne peut être supprimé.	Peut être supprimé avec la commande docker network rm.
Les conteneurs associés au réseau peuvent communiquer entre eux mais doivent utiliser leur adresse IP (qui peut potentiellement changer après un redémarrage du conteneur).	Les conteneurs associés au réseau peuvent s'adresser les uns aux autres par leur nom grâce au DNS intégré au démon docker.

Avant Docker 1.9, seul le réseau *bridge* était disponible. Par défaut, tous les conteneurs d'un même hôte pouvaient communiquer entre eux. Pour que les conteneurs puissent s'adresser entre eux par leur nom, il fallait utiliser le modificateur -- link de la commande run. Cette manière de procéder est aujourd'hui clairement déconseillée.

Lançons le script suivant :

```
| $ ./decicated_network/run_dedicated_network.sh
```

Ce script crée un réseau *test* et l'associe à deux conteneurs busybox (utilisant l'image Docker *busybox*). Un troisième conteneur busybox est aussi créé sans être associé à notre réseau *test*.

```
| $ docker network create test  
$ docker run -itd --name busybox-1 --net test busybox  
$ docker run -itd --name busybox-2 --net test busybox  
$ docker run -itd --name busybox-3 busybox
```

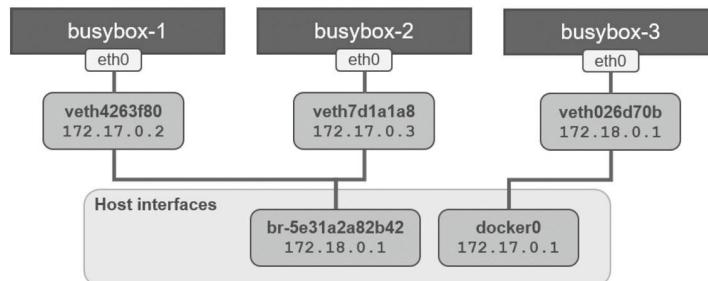
Le réseau ainsi créé est maintenant visible :

```
$ docker network ls  
NETWORK ID      NAME      DRIVER      SCOPE  
79063141fcad    bridge    bridge      local  
9f9a3da3b637    host      host       local  
0295a8a47114    none      null       local  
bdf2d8c764fd  test      bridge      local
```

L'inspection du réseau *test* nous indique que deux conteneurs lui sont associés :

```
$ docker network inspect test --format '{{.Containers}}'  
map[3615cfa6dec5569f68b5ff4707fc763055da3822f89a9928bed6e76f549add1:  
{busybox-1 0a2fe85468edaf0ee810f01e85fb9f8e55f6a4d43966ab227eed618d0dd6fc7a 02:42:ac:12:00:02  
172.18.0.2/16 } fc8adb4f90831ffc01df98f51021e6486590093bdd8c8e015827ae587c59a78d:  
{busybox-2 5006444153a265b278c81c4d5663bb1f21b1195c09117541a6cd4e3135ae9b07 02:42:ac:12:00:03  
172.18.0.3/16 }]
```

Figure 8.11 – Topologie d'un réseau *bridge* personnalisé



L'inspection du réseau *bridge* n'a effectivement qu'un seul conteneur associé (*busybox-3* comme prévu) :

```
$ docker network inspect bridge --format '{{.Containers}}'  
map[02f51e771af54934375f8451c935dc59238cd150788b44e5d50d99ac88c1c58:{ busybox-3  
3d0038b7e93978b3de41e0114ee09386918ecdd81c18bc34a9693a8d5e7a6e29 02:42:ac:11:00:02 172.17.0.2/16  
}]
```

Connectons-nous à l'intérieur du conteneur *busybox-1* :

```
$ docker exec -ti busybox-1 /bin/sh  
# ping 172.22.0.3  
PING 172.22.0.3 (172.22.0.3): 56 data bytes  
64 bytes from 172.22.0.3: seq=0 ttl=64 time=0.178 ms  
...  
# ping busybox-2  
PING busybox-2 (172.22.0.3): 56 data bytes  
64 bytes from 172.22.0.3: seq=0 ttl=64 time=0.064 ms
```

On constate, comme prévu, que le conteneur est capable de voir *busybox-2* et de l'adresser par son adresse IP comme par son nom. En revanche, impossible d'accéder au conteneur *busybox-3* :

```
# ping busybox-3  
ping: bad address 'busybox-3'  
# ping 172.17.0.2  
PING 172.17.0.2 (172.17.0.2): 56 data bytes  
^C  
- - - 172.17.0.2 ping statistics ---  
3 packets transmitted, 0 packets received, 100% packet loss  
# exit
```

Par contre, si on attache depuis l'hôte le conteneur busybox-1 au réseau *test* (grâce à la commande *connect*), le résultat est différent :

```
docker network connect bridge busybox-1
docker exec -ti busybox-1 /bin/sh
# ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.179 ms
# ping busybox-3
ping: bad address 'busybox-3'
```

On constate que le conteneur busybox-3 peut désormais être atteint via son adresse IP mais pas par son nom puisque le réseau *bridge* ne le permet pas.

Vous pouvez lancer la commande suivante pour détruire les conteneurs et le réseau que nous venons de créer :

```
| $ ./dedicated_network/clean_dedicated_network.sh
```

Les réseaux dédiés offrent donc un moyen simple de regrouper les conteneurs ayant besoin de communiquer entre eux, tout en offrant un système d'adressage simplifié grâce au DNS de Docker. Mais qu'en est-il de la communication entre les conteneurs, leur hôte et le monde extérieur ?

### 8.2.5 Lien entre les conteneurs, l'hôte et les réseaux auxquels il est connecté

La communication de nos conteneurs entre eux, avec l'hôte et avec le monde extérieur est contrôlée 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 entre les interfaces réseau de l'hôte.

#### 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*<sup>7</sup>, qui réalise effectivement le filtre des paquets réseau au niveau du kernel Linux.

Le réseau Docker *bridge* (ainsi que tous les autres réseaux Docker) se base sur *iptables* pour autoriser/refuser les connexions entre conteneurs.



Il est possible de désactiver la gestion automatique d'*iptables* par Docker. Notons néanmoins qu'une gestion 100 % manuelle est dans la plupart des cas relativement

pénible. Il est souvent préférable, et possible, de combiner des règles spécifiques à celles qui sont automatiquement générées par Docker. Nous aborderons un exemple de ce type de pratique dans quelques instants.

Regardons maintenant comment Docker utilise *iptables* pour certains cas courants.

### Accès aux réseaux extérieurs

Pour sortir d'un 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* pour assurer la translation d'adresse (NAT).

Nous pouvons le visualiser 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
...
```

Dans ce cas, notre réseau *bridge* (172.17.0.0/16) pourra accéder à toute IP en dehors de ce réseau.

### Accès à l'hôte depuis un conteneur : l'option « host »

La première solution pour permettre à un conteneur d'accéder à des ressources de l'hôte est de l'associer au réseau *host* :

```
$ ./host_network/run_host_network.sh
docker exec -ti centos-test /bin/bash
# yum install -y nmap
# nc -l 5000
```

La séquence de commande précédente lance (à l'intérieur du conteneur) un processus écoutant sur le port 5000. Immédiatement, l'effet est visible depuis l'hôte :

```
$ netstat -vatn | grep 5000
tcp            0      0.0.0.0.0:5000          0.0.0.0:*      LISTEN
tcp6           0      :::::5000              ::::*          LISTEN
```

En réalité, il ne s'agit pas de communication entre l'hôte et le conteneur puisque ceux-ci partagent exactement le même réseau. Toutes les ressources exposées par l'hôte sont accessibles du conteneur et vice versa.

Ce type d'option doit être réservé aux conteneurs provenant d'une source fiable. Elle peut aussi être utile pour des conteneurs autonomes exécutant des scripts ayant besoin d'accéder aux ressources de l'hôte.

Vous pouvez maintenant détruire le conteneur précédemment créé :

```
| ./host_network/clean_host_network.sh
```

### Accès à l'hôte depuis un conteneur : l'option « gateway »

Comme nous l'avons vu un peu plus haut, chaque réseau de type *bridge* dispose d'une *gateway* qui se révèle être l'hôte. Celle-ci est pourvue d'une adresse IP. Serait-il donc possible d'accéder à l'hôte de cette manière ?

Créons un conteneur associé au réseau *bridge* (notez que la démonstration serait la même pour un réseau dédié de type *bridge*), connectons-nous dans celui-ci et installons quelques packages dont nous aurons besoin :

```
$ ./access_host/run_access_host.sh
```

```
$ docker exec -ti centos-host /bin/bash
# yum install -y net-tools nmap openssh-clients
```

L'adresse IP de l'hôte est, comme nous l'avons expliqué précédemment, par défaut celle de la gateway :

```
# echo $(route -n | awk '/UG[ \t]/{print $2}')
172.17.0.1
```

Vérifions si cette adresse est atteignable en utilisant l'utilitaire nmap qui va scanner les différents ports pour identifier des services disponibles :

```
# nmap 172.17.0.1
Starting Nmap 6.40 ( http://nmap.org ) at 2018-09-16 11:26 UTC
Nmap scan report for 172.17.0.1
Host is up (0.000065s latency).
Not shown: 999 filtered ports
PORT STATE SERVICE
22/tcp open ssh
MAC Address: 02:42:8B:A3:95:AC (Unknown)
```

La commande nous indique que le port 22 (SSH) semble ouvert et accessible. Essayons de nous y connecter :

```
# ssh vagrant@172.17.0.1
The authenticity of host '172.17.0.1 (172.17.0.1)' can't be established. ECDSA key
fingerprint is SHA256:HmOsSM00xdalDntOvoKiDFYJilyv//3Mk9ya0+d330k. ECDSA key
fingerprint is MD5:84:6b:86:8b:ac:09:d8:f2:01:37:85:12:84:47:f0:f8. Are you sure
you want to continue connecting (yes/no)? yes
Warning: Permanently added '172.17.0.1' (ECDSA) to the list of known hosts.
vagrant@172.17.0.1's password:
```

Last login: Fri Sep 14 14:00:33 2018

La connexion est donc possible mais par quelle route ?

En réalité le trafic vers l'hôte est protégé par le firewall de la machine (*iptables*). Le fait que le port 22 soit accessible n'est ni plus ni moins la conséquence du fait que ce port est ouvert pour qui souhaite se connecter à l'hôte.

La chose est facilement vérifiable en ouvrant un autre port sur ce firewall. Sur l'hôte, entrez les commandes suivantes :

```
sudo service firewalld start
sudo service docker restart
sudo firewall-cmd --list-all
sudo firewall-cmd --add-service=http --permanent
sudo firewall-cmd --reload
```

Nous utilisons ici l'utilitaire *firewalld*, qui propose une interface simple pour modifier la configuration *iptables*. Nous ajoutons ici le port 80.



Vous noterez que nous redémarrons le service Docker après avoir activé le firewall (s'il n'était pas déjà activé sur votre hôte). La raison est liée à ce que nous avons exposé plus haut. Docker génère automatiquement des règles *iptables* en fonction de la création des conteneurs et des réseaux. Il existe des risques de conflit lorsque vous modifiez ces mêmes règles vous-même. Il est donc généralement recommandé de redémarrer Docker après des modifications de firewall sur la machine.

Revenons à l'intérieur de notre conteneur :

```
docker exec -ti centos-host /bin/bash
# nmap 172.17.0.1
Starting Nmap 6.40 ( http://nmap.org ) at 2018-09-16 13:02 UTC
Nmap scan report for 172.17.0.1
Host is up (0.000075s latency).
Not shown: 998 filtered ports
PORT STATE SERVICE
22/tcp open ssh
80/tcp open http
```

Cette fois la commande nmap nous indique que le port HTTP (80) est bien accessible du conteneur : encore une fois tout simplement parce que ce port est aussi ouvert pour la machine dans son ensemble. L'accès à l'hôte n'est donc, heureusement, pas complètement transparent pour un conteneur non connecté au réseau *host*.

Nous en avons fini avec les réseaux. Abordons maintenant le sujet du partage de fichier : les volumes Docker.

## 8. 3 PERSISTANCE : *BIND MOUNTS ET VOLUMES*

Toute application a besoin d'un état. Dans notre application exemple, nous avons trois besoins de persistance :

- le partage de fichier entre le serveur d'application et le processeur asynchrone ;
- la conservation des fichiers de notre base de données (si nous choisissons de l'encapsuler dans un conteneur) ;
- la conservation des logs des différents composants entre deux redémarrages.

Nous allons analyser les solutions applicables pour chacun de ces cas. Mais auparavant regardons de plus près les fonctionnalités Docker que nous pourrions utiliser : qu'est-ce qu'un *bind mount*, qu'est-ce qu'un volume Docker, comment fonctionnent-ils et quelle solution est la plus adaptée pour chacun des cas précédemment cités ?

### 8.3.1 Pas de persistance par défaut

Comme nous l'avons expliqué précédemment, l'architecture même d'un conteneur entraîne la perte de toutes les données ajoutées à l'image (*copy on write*) lorsque celui-ci est détruit. Il ne faut pas nécessairement considérer cette situation comme un désavantage. La possibilité d'effacer complètement (et proprement) l'état d'un composant :

- est diablement utile dans un environnement d'intégration continue (comme nous le verrons dans le chapitre suivant) ;
- est un facteur de résilience et de scalabilité dans la plupart des architectures modernes.

Mais il reste des situations dans lesquelles nous avons besoin de conserver un état.

### 8.3.2 Les deux types de systèmes de fichiers persistants

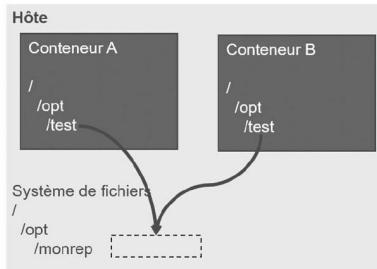
Il existe deux manières de procéder (que nous avons évoquées dès le chapitre 1) :

- les *bind mounts* ;
- les volumes, qui sont plus versatiles et clairement indispensables dans les architectures multi-hôtes.

#### ***Bind mount : rappel***

C'est la méthode la plus simple pour rendre persistant un répertoire du conteneur. En pratique il s'agit de « monter » un répertoire de l'hôte dans le conteneur. Toute modification à l'intérieur du conteneur se répercute sur l'hôte (et vice versa si on le souhaite).

Figure 8.12 – Un *bind mount* de deux conteneurs sur le même répertoire de l'hôte



```
$ docker system prune -f
docker run --rm -ti --name=centos-test -v ${PWD}:/opt/test centos:7
# touch /opt/test/mylocalfile
# exit
ls -la
- rwxrwx--- 1 root vboxsf 0 17 sept. 14:47 mylocalfile
```

Dans l'exemple ci-dessus, nous avons créé un conteneur avec un montage dans le répertoire courant de l'hôte (représenté par \${PWD}) vers le répertoire /opt/test du conteneur. Nous avons créé un fichier que nous sommes en mesure de retrouver sur l'hôte.

Attention : vous aurez peut-être remarqué que l'utilisateur propriétaire de ce fichier est root. Nous reviendrons sur ce sujet mais c'est la conséquence directe du fait que le conteneur a démarré sous cet utilisateur. Nous aurions pu demander au conteneur de démarrer sous un autre utilisateur mais celui-ci n'aurait peut-être pas eu les droits d'accès sur le montage.

```
$ docker volume ls
DRIVER      VOLUME NAME
```

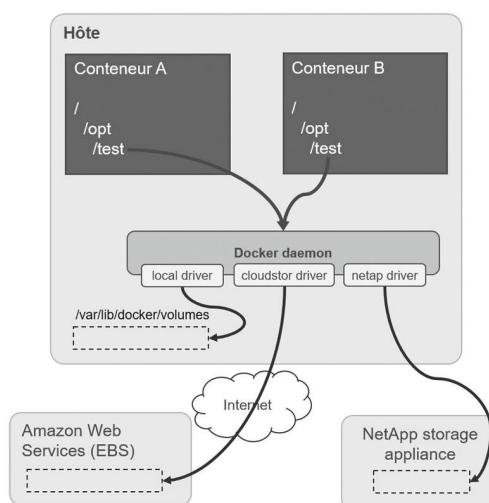
Nous pouvons aussi noter que ce montage n'est pas un volume au sens Docker du terme. La commande ci-dessus nous indique bien qu'aucun volume n'a été créé.

Attention : le montage d'un répertoire depuis l'hôte se substitue complètement au contenu originel du répertoire dans le conteneur. Si vous montez un répertoire sur le chemin /var du conteneur (ce que nous vous déconseillons fortement), tout le contenu originel de ce répertoire pourtant bien fourni dans une distribution Linux sera écrasé.

## Volume

Les volumes sont une autre solution plus flexible de gestion de la persistance. Il s'agit dans ce cas de déléguer à Docker la gestion de blocs de système de fichiers via des drivers à la manière des réseaux. Par défaut Docker, utilisant le driver « local », stocke ses volumes dans une zone dédiée de l'hôte (/var/lib/docker/volumes). Nous verrons qu'il existe d'autres drivers qui peuvent virtualiser la localisation de ces données et ainsi, par exemple, en assurer la conservation en cas de crash de l'hôte.

Figure 8.13 – Gestion de la persistance avec les volumes Docker



Il y a quelques autres avantages dans l'usage de volumes. Commençons par créer notre volume :

```
docker volume create volume-test
docker volume ls
DRIVER      VOLUME NAME
local      volume-test
```

Nous pouvons ensuite le monter dans un conteneur pour en inspecter le contenu :

```
$ docker run --rm -ti --name=centos-test -v volume-test:/opt/test centos:7
```

```

ls -la /opt/test/
total 0
drwxr-xr-x 2 root root 6 Sep 17 13:26 .
drwxr-xr-x 1 root root 18 Sep 17 13:28 ..
exit

```

Comme prévu, le volume est vide. Essayons maintenant de monter ce même volume (vide, c'est important) sur le chemin /var :

```

docker run --rm -ti --name=centos-test -v volume-test:/var centos:7
# ls /var/
adm cache db empty games gopher kerberos lib local lock log mail nis opt preserve run spool tmp
yp
# exit

```

Premier étonnement, le volume est maintenant plein. Les données originellement présentent dans le répertoire /var du conteneur ont été copiées dans le volume. Nous pouvons le vérifier en l'inspectant à nouveau :

```

docker run --rm -ti --name=centos-test -v volume-test:/opt/test centos:7
# ls /opt/test/
adm cache db empty games gopher kerberos lib local lock log mail
nis opt preserve run spool tmp yp
# exit

```

C'est l'une des propriétés utiles des volumes : pour peu que le volume soit vide, le contenu du répertoire originel du conteneur est automatiquement copié dans le volume au démarrage. Nous en avons incidemment vérifié une autre : nous avons pu sans problème monter le même volume dans plusieurs conteneurs en utilisant son nom comme identifiant (sans référence à un chemin particulier).

Ouvrez deux terminaux et dans chacun créez les deux conteneurs avec la commande suivante :

```
| $ docker run --rm -ti -v volume-test:/opt/test centos:7
```

Dans le premier, lancez la commande suivante qui va tenter, s'il existe, d'afficher, toutes les secondes, le contenu d'un fichier présent dans le volume :

```
| # watch -n 1 cat /opt/test/fichier.txt
```

Dans le second, lancez la commande suivante :

```
| # echo -e "Heure du moment $(date)" >> /opt/test/fichier.txt
```

Retournez dans le premier terminal et constatez que la communication fonctionne. Il est donc possible et très simple d'associer plusieurs conteneurs au même volume.

Revenons maintenant aux trois cas d'usage que nous avons évoqués précédemment.

### 8.3.3 Les solutions pour notre application

#### **Partage de fichier**

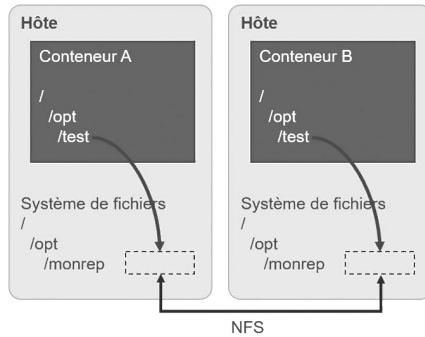
Pour ce cas d'usage, les deux méthodes évoquées ci-dessus sont relativement équivalentes lorsque les deux conteneurs se trouvent sur le même hôte.

##### ***Pour le bind mount : simplicité opérationnelle***

L'usage d'un montage sur le système de fichiers peut sembler plus simple pour l'opération de la solution (comme le backup par exemple). On constate souvent que les équipes de support peu familières de Docker préfèrent cette solution qui ne remet pas en cause les pratiques existantes (backup, usage de la ligne de commande pour accéder aux données depuis l'hôte, gestion des autorisations, etc.).

Dans le cas d'une architecture multi-hôte, la solution est aussi applicable mais il est alors nécessaire de trouver un moyen de synchroniser les systèmes de fichiers des différents hôtes (par NFS ou en montant un volume commun depuis un NAS par exemple).

Figure 8.14 – Gestion de volume *bind mount* sur plusieurs hôtes



#### **Pour les volumes : 100 % Docker et sécurité**

Notons tout d'abord que le montage de répertoires présente parfois quelques surprises notamment avec des systèmes de sécurité comme [selinux](#)<sup>8</sup> mais rien qui n'arrête un administrateur système compétent.

Les volumes, hors des cas d'orchestration de conteneurs qui imposent une abstraction complète de l'hôte, n'ont que peu d'avantages opérationnels pour les applications simples. Évidemment la chose est différente si on dispose d'un driver et d'une solution standardisée de gestion du stockage compatible avec Docker. Mais reconnaissons que la chose est encore relativement rare.

La valeur des volumes s'exprime plutôt pour les environnements de développement et de test. En effet lorsque l'on doit créer un nombre important de conteneurs du même type, utiliser un montage sur l'hôte nécessite de créer une arborescence de répertoire « manuellement » (un sous-répertoire par conteneur par exemple). On doit donc, lors de l'instanciation d'un conteneur, intervenir préalablement sur l'hôte. Les volumes, eux, permettent de rester dans un monde 100 % Docker.

Les volumes ont aussi des avantages en termes de sécurité. Le montage d'un répertoire de l'hôte dans un conteneur doit toujours se faire avec une certaine prudence. N'oublions pas que l'utilisateur du conteneur courant peut être root. Les volumes locaux (driver « local ») étant placés dans un chemin « neutre » (/var/lib/docker/volumes), le risque d'exposer l'hôte est nécessairement plus faible quoi qu'il advienne.

D'une manière générale, il est toujours utile de limiter l'exposition de l'hôte au minimum. Si vous n'avez besoin que de lire des données depuis l'hôte, il est possible et préférable de monter le répertoire en « lecture seule » avec le modificateur `ro` :

```
docker run --rm -ti -v volume-test:/opt/test:ro centos:7
```

#### **Gestion des logs**

Il est bien évidemment possible d'utiliser l'une des techniques précédentes pour conserver les logs d'un conteneur. Les avantages et inconvénients sont les mêmes.

Néanmoins, en pratique il est préférable d'avoir recours aux fonctions natives de Docker pour la gestion des logs. Comme nous l'avons vu dans le chapitre 5, Docker offre un large panel de drivers permettant de collecter les logs issus de plusieurs conteneurs. Couplée à des solutions de concentration de logs comme [logstash](#)<sup>9</sup>, cette option offre des avantages tout à fait significatifs.

#### **Une base de données dans Docker**

De nombreux articles ont circulé mettant en garde contre la conteneurisation des bases de données. En pratique, dans la plupart des cas, la chose est faisable et même largement pratiquée (au moins en développement).

La contrainte est, encore une fois, le niveau de maîtrise des équipes opérationnelles. Si celles-ci disposent d'une expérience pluriannuelle de gestion et d'optimisation de vos bases de données (en cluster qui plus est) dans un environnement natif, les avantages de la conteneurisation sont difficiles à démontrer. L'argument est un peu similaire si vous déployez votre solution dans un cloud public qui offre des solutions de gestion de base de données en PaaS (RDS d'Amazon ou Azure Database).

En revanche, si vous débutez la mise en place d'une toute nouvelle infrastructure (notamment sur la base d'une solution d'orchestration comme Kubernetes), la conteneurisation peut être considérée.

Elle nécessite cependant un certain goût du risque.

Voici quelques liens pour ceux qui souhaiteraient se lancer dans l'aventure :

<https://kubedb.com/>

<http://galeracluster.com/documentation-webpages/docker.html>

<https://blog.docker.com/2017/09/microsoft-sql-on-docker-ee/>

## 4 CONFIGURATION D'APPLICATION

L'un des challenges communs pour une application est de gérer d'une manière sécurisée les secrets et configurations de plusieurs environnements (production, qualification...) :

mots de passe ;

clé SSH ;

certificats d'authentification ;

chaîne de connexion à une base de données ;

etc.

Outre le besoin de s'assurer que seuls les utilisateurs autorisés aient accès à certaines de ces informations, il faut aussi s'assurer de la possibilité de les modifier, si possible sans avoir à reconstruire l'application.

Voici un tableau résumant quelques solutions usuelles et les problèmes qui leur sont associés :

**Tableau 8.2 – Solutions possibles pour la gestion des configurations**

Solution possible	Problème(s)
Ajouter un fichier de configuration au moment du <i>build</i> (via une instruction Dockerfile COPY ou ADD).	Cette solution a deux défauts : <ul style="list-style-type: none"><li>Premièrement elle ne permet pas de changer les secrets ou la configuration sans reconstruire l'image de l'application. Lors d'un incident de production la chose peut être ennuyeuse.</li><li>Si on dispose de plusieurs environnements, on est conduit à embarquer dans l'image les paramètres de tous les environnements ou bien à construire plusieurs variantes de la même image. La première solution est plutôt mauvaise en termes de sécurité, puisque les développeurs pourraient avoir accès aux paramètres de production. La seconde est probablement la meilleure mais nécessite une certaine dose d'automatisation. <i>Reportez-vous par exemple au chapitre 7 et à la fonction ARG, qui permet de rendre un Dockerfile paramétrable.</i> Comme les paramètres d'authentification sont inclus dans l'image, il faut s'assurer que le registre qui référence ces images (ou même le cache de l'hôte) soit inaccessible au personnel non autorisé.</li></ul>
Fournir le fichier de configuration au démarrage via un volume ou un <i>bind mount</i> .	L'avantage de cette solution est qu'une seule image peut être utilisée pour plusieurs environnements. L'inconvénient est que cette solution requiert la mise en place d'une solution pour « amener » les fichiers de configuration jusqu'à l'hôte.
Passer un paramètre au démarrage du conteneur via une variable d'environnement.	Cette solution utilise la commande ENV de Dockerfile. Elle a l'avantage de permettre un changement de paramètre sans nécessiter de nouveau <i>build</i> de l'image. Le problème est déporté en amont. En effet l'usage de variables d'environnement impose que ces secrets ou configuration soient transmis en clair, via la ligne de commande ou un script. Il y a donc une rupture entre l'espace de stockage de ces paramètres (qui sera généralement crypté) et le conteneur.

Il existe aussi des solutions visant à extraire la gestion de configuration de l'application elle-même pour la déporter sur des serveurs sécurisés et dédiés. Citons par exemple Spring Cloud Config<sup>10</sup>, qui peut extraire ses données d'un dépôt GIT mais aussi d'un coffre-fort numérique (*vault*). Avec ce type de solution, le passage d'information requise est limité aux paramètres d'authentification du serveur de configuration.

Nous verrons, dans la dernière partie du livre, que les orchestrateurs (Kubernetes et Swarm) disposent de fonctionnalités simplifiant le transfert de secrets et de configurations.

## 8. 5 MONITORING

La plupart des outils de monitoring professionnels disposent de plugins pour Docker.

Les outils open source ne sont pas en reste. Nagios<sup>11</sup> ou Prometheus<sup>12</sup>, pour ne citer qu'eux, disposent de plugins pour le démon Docker. Il en est de même pour les outils de mesure de la performance comme NewRelic<sup>13</sup>.

Le monitoring des applications Docker n'est pas une problématique complexe.

Pour les besoins simples, en environnement de test ou de développement, on peut évidemment utiliser la commande docker stats, que nous avons vue dans le chapitre 5.

Notons aussi l'outil CAdvisor de Google :

```
docker run \ --volume=/:/rootfs:ro \
  volume=/var/run:/var/run:rw \
  volume=/sys:/sys:ro \
  volume=/var/lib/docker/:/var/lib/docker:ro \
  --publish=8080:8080 \
  --detach=true \
  name=cadvisor \
  privileged=true \
  google/cadvisor:latest
```

Celui-ci offre une interface graphique très complète.

Figure 8.15 – CAdvisor, l'outil de monitoring de conteneur de Google



Dans ce chapitre, nous avons présenté notre application exemple. Celle-ci nous a permis d'aborder des concepts clés qui nous seront utiles dans les chapitres suivants. Nous sommes maintenant prêts à passer à la phase d'implémentation.

- 
1. <http://spring.io/projects/spring-boot>
  2. <https://github.com/docker/libnetwork>
  3. Pour plus d'information sur la notion de *namespace*, vous pouvez vous reporter au chapitre 1.
  4. <http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge>
  5. <https://store.docker.com/search?category=network&q=&type=plugin>
  6. <https://github.com/jpetazzo/pipework>
  7. <http://www.netfilter.org/>
  8. [https://www.projectatomic.io/blog/2016/03/dwalsh\\_selinux\\_containers/](https://www.projectatomic.io/blog/2016/03/dwalsh_selinux_containers/)
  9. <https://www.elastic.co/products/logstash>
  10. <https://cloud.spring.io/spring-cloud-config/>
  11. <https://www.nagios.org/>
  12. <https://prometheus.io/>
  13. <https://newrelic.com/>

# 9

## Conditionnement et déploiement

### Objectif

L'objectif de ce chapitre est d'expliquer comment utiliser Docker pour conditionner une application.

Nous aborderons pour cela différentes architectures avec leurs avantages et leurs inconvénients.

L'issue de ce chapitre, vous aurez compris les différentes techniques permettant de « dockeriser » une application.

Nous nous limitons ici à un cas n'ayant pas recours aux fonctions d'orchestration (comme Kubernetes ou Swarm, décrites dans la cinquième partie du livre), plus complexes et parfois surdimensionnées pour les besoins d'applications simples.



### Application exemple et code source

L'application que nous allons utiliser dans ce chapitre est celle qui a été présentée dans le chapitre 8.

Le code source et les exemples de ce chapitre sont disponibles sur GitHub.

Veuillez vous reporter à la procédure d'installation du chapitre 3.

### 9. 1 BUILD / RUN : PRINCIPES

#### 9.1.1 Code source de l'application

Le code source des différents modules se trouve dans le répertoire `application-exemple`. Le tableau 9.1 donne une description du contenu de chaque sous-répertoire.

**Tableau 9.1 – Usage des paquets installés**

Sous-répertoire	Description
back-end	Le code source du module back-end et son fichier de configuration.
build	Un répertoire contenant des outils pour compiler le front-end et gérer la configuration des applications.
data	Des exemples de fichier CSV qu'il est possible de charger via le web front-end (ils serviront à vérifier que l'application fonctionne).
database	Le schéma SQL nécessaire pour initialiser la base de données.
front-end	Le code source Java du web front-end. Nous utilisons un projet Gradle <sup>1</sup> pour organiser la compilation et les dépendances du module.

Les fichiers compris dans ces différents sous-répertoires seront utilisés par les scripts de `build` et de `run`. La manière de les utiliser variera selon les architectures que nous étudierons.

#### 9.1.2 Scripting

chaque mode d'installation que nous allons étudier correspond un sous-répertoire du répertoire chapitre9.

Dans chaque cas vous trouverez deux scripts :

`build.sh`, qui construit les images Docker nécessaires au déploiement ;

`run.sh`, qui lance l'application.

La durée d'exécution du `build` peut être relativement longue selon les cas.

Nous appliquons quelques principes essentiels que nous retrouverons dans le prochain chapitre, consacré à l'intégration continue :

le `build` comme le `run` sont « idempotents », c'est-à-dire qu'ils peuvent être exécutés autant de fois que nécessaire puisqu'ils aboutissent au même état final ;

le `build` écrase la version précédente de l'image à chaque fois qu'il est lancé ;

le répertoire de travail du `build` est remis en état à chaque lancement (les dépendances, fichiers sources ou autre, sont pris depuis le répertoire `application-exemple` ;

le `run` détruit les conteneurs précédemment lancés avant d'en recréer de nouveaux.

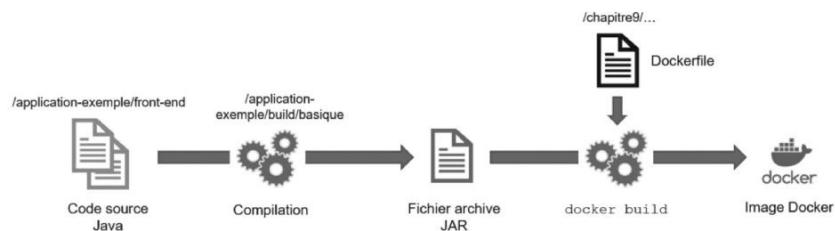
Seuls les volumes sont préservés entre deux lancements du `run` afin de conserver les données précédemment acquises.

### 9.1.3 Compilation du front-end

Notre interface graphique est une application web Java. Il est donc nécessaire de la compiler avant de pouvoir l'utiliser.

Nous aborderons la question du développement dans le chapitre suivant. Dans celui-ci nous utiliserons un script présent dans le répertoire `application-exemple/build/basique`, qui créera le fichier d'application JAR. Nous n'aurons plus ensuite qu'à le copier dans notre image lors de la phase de `build`.

Figure 9.1 – Compilation du front-end



Les applications SpringBoot sont conditionnées sous la forme d'un simple JAR (fichier archive Java) qui inclut le serveur d'application (Tomcat par défaut) et l'ensemble des dépendances nécessaires à l'application. Ce mode de conditionnement explique pourquoi SpringBoot est très utilisé pour l'implémentation de micro-services en Java. Une fois l'application compilée, l'installation de celle-ci ne nécessite qu'un seul fichier et l'exécutable Java.

Le script de compilation est appelé du début de chaque appel à `build.sh` pour chaque architecture. Le temps de compilation est long car il n'est pas optimisé pour de multiples compilations successives. Il doit donc télécharger l'ensemble des dépendances nécessaires depuis Internet à chaque exécution. Nous verrons dans le chapitre suivant qu'il est possible de fabriquer une version rapide de ce processus de compilation en conservant un cache des données téléchargées.

### 9.1.4 Petit truc

Vous trouverez au début de la plupart des scripts shell les instructions suivantes :

```
ORIGINDIR=$PWD
```

```
BASEDIR=$(dirname $0)
```

```
| cd $BASEDIR
```

Ces instructions n'ont pas de rapport avec notre application. Elles permettent seulement de s'assurer que le répertoire courant du script est toujours celui où ce dernier est localisé. Il s'agit de simplifier l'usage des scripts dans le cadre de cet ouvrage (dans le cas où le script serait lancé depuis un autre répertoire).

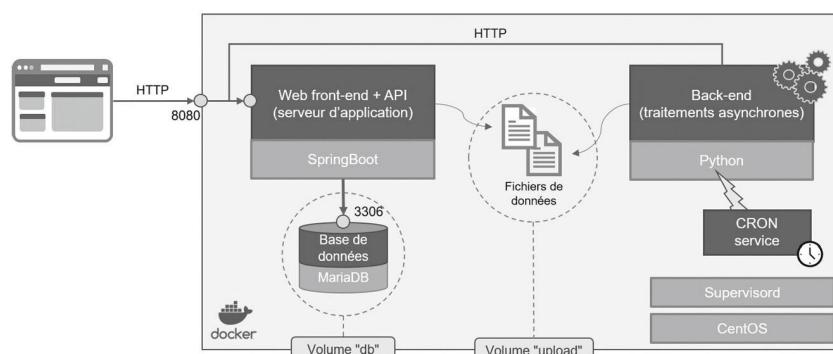
## 9. 2 OPTION 1 : UN SEUL CONTENEUR, PLUSIEURS PROCESSUS

Tous les fichiers dont nous aurons besoin sont disponibles, pour ce paragraphe, dans le répertoire supervisor du code que vous avez téléchargé de GitHub pour le chapitre 9.

La première idée que nous allons mettre en œuvre est la suivante : fabriquer une image Docker fournissant tous les services nécessaires à notre application (front-end, back-end et base de données).

Architecturalement cette option revient à utiliser Docker comme une sorte de machine virtuelle. Pour coordonner les différents processus de notre application, nous aurons recours à supervisord<sup>2</sup>.

Figure 9.2 – Application mono-conteneur



Notre hôte et le conteneur qu'il hébergera interagiront comme suit :

nous créerons donc un seul conteneur qui exposera le port 8080 de notre front-end web ;  
nous utiliserons deux volumes : le premier contiendra les fichiers partagés de notre application et le second les fichiers de notre base de données.

Avec cette configuration, nous pourrons détruire/reconstruire/modifier notre conteneur. L'état de l'application sera conservé grâce aux volumes.

### 9.2.1 Installation des différentes dépendances

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<sup>3</sup>. 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 (par l'intermédiaire de yum update -y) ;  
installer les différents paquets dont nous avons besoin.

```
RUN yum install -y epel-release && \  
yum update -y && \  
yum install -y mariadb-server \  
    java \  
    supervisor \  
    python36 \  
    curl
```

```

| cronie \
|   && yum clean all

```

Le tableau 9.2 explique l'usage de chaque paquet.

**Tableau 9.2 – Usage des paquets installés**

Paquet	Description
mariadb-server	Le paquet apportant le serveur de base de données (MariaDB) qui est utilisé par notre application exemple.
java	Apporte le programme Java qui permet d'exécuter le front-end web développé dans ce langage.
supervisor	Le gestionnaire de service qui va coordonner l'exécution des trois composants de notre application.
python36	Le langage qui est utilisé pour exécuter le script d'ingestion des fichiers.
cronie	Le paquet apportant l'utilitaire CRON <sup>4</sup> qui permet de déclencher à intervalle régulier notre script d'ingestion de fichier.

Maintenant que nous avons installé nos différentes dépendances, il ne nous reste plus qu'à les configurer.

### 9.2.2 Création de l'image

Chaque sous-paragraphe suivant explique une partie du Dockerfile suivant : chapitre9/supervisor/Dockerfile.

#### **Étape Dockerfile : 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/

```

Nous devons ensuite préparer l'exécution de l'initialisation de la base de données avec un schéma SQL prédéfini (celui de notre application) :

```
| COPY schema.sql /schema.sql
```

Pour ce faire, nous chargeons dans l'image le schéma SQL correspondant. Nous verrons par la suite comment il sera exécuté.

#### **Étape Dockerfile : web front-end**

Comme indiqué dans le paragraphe précédent, la phase de compilation du front-end précède la création de l'image. Nous disposons donc du fichier archive Java JAR au moment du docker build de notre image.

La première instruction consiste alors à copier le fichier JAR dans l'image :

```
| COPY *.jar /app.jar
```

Les suivantes nécessitent quelques explications :

```

| RUN echo -e "/usr/bin/mysqladmin --no-defaults --port=3306 --user=root password '$
| {MYSQL_ROOT_PASSWORD}'\n" > /start-front-end.sh
| RUN echo -e "/usr/bin/mysql -v -u root -p${MYSQL_ROOT_PASSWORD} < /schema.sql\n" >> /start-
| front-end.sh
| RUN echo -e "/usr/bin/java -jar /app.jar\n" >> /start-front-end.sh
| RUN chmod u+x /start-front-end.sh

```

Comme nous l'avons précédemment expliqué, il est nécessaire d'initialiser la base de données avant d'y accéder et donc avant de démarrer le front-end. Dans le cas contraire, la connexion du front-end échouerait car le schéma serait absent.

Il n'est pas possible d'effectuer cette initialisation au moment de la construction de l'image. En effet, les fichiers de base de données doivent être placés dans des volumes pour être conservés entre deux exécutions. Réinitialiser la base après chaque redémarrage de l'application serait quelque peu... contre-productif.

Ces fichiers de base de données ne peuvent donc être produits qu'au moment où le conteneur est créé.

Le code précédent crée un fichier `start-front-end.sh` qui va initialiser la base de données avant de démarrer le front-end :

```
en définissant un mot de passe root pour MariaDB (qui est le compte que nous utiliserons pour accéder à la base de données) ;  
en exécutant le code SQL du schéma pour le créer si celui-ci n'existe pas.
```

Si vous ouvrez le fichier `application-exemple/database/schema.sql`, vous noterez que nous utilisons pour nos instructions SQL la commande `IF NOT EXISTS`. Celle-ci nous assure que le script de démarrage fonctionne à chaque lancement même quand le schéma et les tables existent déjà.

### Étape Dockerfile : back-end

Cette partie du Dockerfile commence par la copie des ressources relatives au back-end dans l'image :

```
COPY main.py /main.py  
COPY config.json /config.json
```

Ces ressources consistent en deux fichiers :

```
l'application elle-même : un script Python ;  
un fichier de configuration.
```

ce stade, nous admettrons que la configuration du back-end comme celle du front-end sont correctes. Nous verrons par la suite comment améliorer la configuration d'une application distribuée.

Nous configurons ensuite Python. Par défaut, sous CentOS, la version de Python installée est la version 2. Nous devons donc indiquer au système que nous souhaitons utiliser la version 3 de Python, qui a été installée en début de Dockerfile :

```
RUN ln -sf /usr/bin/python3.6 /usr/local/bin/python  
ENV PYTHONIOENCODING=utf-8
```

La dernière instruction crée un fichier CRON dans le répertoire standard `/etc/cron.d`. Ce fichier a un format particulier qui indique que notre script back-end doit être lancé toutes les minutes :

```
RUN echo -e "* * * * * root /usr/local/bin/python /main.py\n" > /etc/cron.d/backend
```

Ce fichier CRON est lu par le processus CRON, dont nous verrons plus tard comment il est démarré.

### Étape Dockerfile : ports et volumes

Il ne nous reste plus qu'à exposer le port 8080 du front-end et un autre port (9001), dont nous verrons l'utilité dans quelques instants :

```
EXPOSE 8080  
EXPOSE 9001
```

Nous définissons ensuite deux volumes :

```
un pour nos données MariaDB (/var/lib/mysql) ;  
un pour recevoir les fichiers chargés depuis le front-end et en attente de traitement par le back-end (/chargement).
```

```
VOLUME ["/var/lib/mysql/"]  
VOLUME ["/chargement"]
```

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 composants ?

Pour ce faire, il nous faut utiliser un gestionnaire de services : nous allons employer Supervisor.

### **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 le front-end, le back-end et MariaDB.

Sa configuration est stockée dans le fichier `supervisord.conf` :

```
[supervisord]
nodaemon=true
logfile=/var/log/supervisor/supervisord.log
[program:mysql]
command=/usr/bin/mysqld_safe --port=3306
autorestart=true
[program:crond]
command=/usr/sbin/crond -n
autorestart=true
autostart=true
[program:front-end]
command=/bin/bash -c /start-front-end.sh
autorestart=true
autostart=false
[inet_http_server]
port=9001
```

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 :

```
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
CMD ["/usr/bin/supervisord", "-n", "-e", "debug", "-c", "/etc/supervisor/conf.d/supervisord.conf"]
```

Nous démarrons ici Supervisor en mode `debug` afin de disposer de plus de logs. Il est évidemment possible d'enlever le modificateur `-e debug`.

### **9.2.3 Lancement du build**

La construction de l'image peut être lancée en utilisant l'instruction `build.sh` présente dans le répertoire `chapitre9/supervisor`. Le processus est long, probablement plusieurs minutes.

#### **Préparation**

La première partie du script n'est pas consacrée à la création de l'image. Elle prépare les différentes ressources qui doivent être copiées dans l'image et notamment, longuement, compile l'application front-end.

```
*****
Copie du code source du projet
*****
*****
Build de l'image de build basique
*****
Sending build context to Docker daemon 117.8kB
Step 1/4 : FROM openjdk:8-alpine
--> 54ae553cb104
```

```
...
Successfully built d6ef82e38bf1
Successfully tagged build-basique:latest
*****
Lancement du build
*****
Downloading https://services.gradle.org/distributions/gradle-4.8.1-bin.zip
.....
```

Vous aurez noté que nous utilisons une image et un conteneur pour compiler l'application. C'est une solution que nous étudierons dans le chapitre suivant. Pour le moment, nous nous satisferons du fait que le processus fonctionne.

### **Création de l'image**

La création de l'image s'effectue avec la désormais classique instruction docker build :

```
| docker build --build-arg MYSQL_ROOT_PASSWORD=root -t appex-supervisor .
```

Vous noterez l'usage du modificateur `--build-arg`. Celui-ci correspond à l'instruction suivante du Dockerfile :

```
| ARG MYSQL_ROOT_PASSWORD
```

Il s'agit de l'instruction ARG, qui permet de définir des arguments de *build*. Ceci nous permet de modifier le mot de passe `root` de la base de données au moment du *build*.



En réalité, si vous tentez de modifier le mot de passe, vous constaterez que l'application ne fonctionne plus. La cause n'est pas liée à cette instruction ARG, qui est parfaitement fonctionnelle, mais au fait que les fichiers de configuration du front-end et du back-end contiennent le mot de passe « hardcodé ». Nous verrons dans la prochaine architecture qu'il est possible de remédier à cette situation au prix d'un peu plus d'efforts au moment du *build*.

### **Nettoyage**

la fin du processus de *build*, le script nettoie les fichiers intermédiaires (copiés depuis l'application exemple) afin de s'assurer que tout lancement ultérieur donne exactement le même résultat.

Une fois le *build* terminé, nous pouvons vérifier que cette nouvelle image est effectivement disponible grâce à la commande docker images, avec comme paramètre le nom de notre image :

```
$ docker images appex-supervisor
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
appex-supervisor latest  8c7723b5f204  4 hours ago  622MB
```

## **9.2.4 Lancement de l'application**

### **Démarrage via le script run.sh**

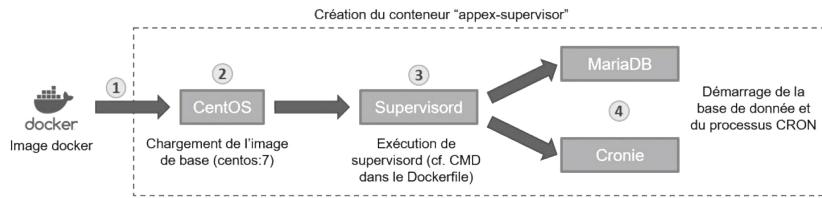
Le lancement de l'application s'effectue au moyen du script `run.sh`. Celui-ci contient essentiellement l'instruction suivante :

```
| $ docker run --rm -d -p 8080:8080 -p 9001:9001 --name=appex-supervisor appex-supervisor
```

L'instruction lance le conteneur en mode auto-destruction (`--rm`). Si vous stoppez le conteneur, celui-ci sera automatiquement détruit. Pour le moment, nous exposons les ports mais nous n'avons pas défini de volume. Nous y reviendrons plus tard.

La figure 9.3 indique la séquence de démarrage du conteneur et des processus qu'il encapsule.

Figure 9.3 – Séquence de démarrage de l'application



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 appex-supervisor ps axf
PID TTY STAT TIME COMMAND
 177 pts/0 R+ 0:00 ps axf
 1 ? Ss 0:00 /usr/bin/python /usr/bin/supervisord -n -e debug -c /
 8 ? S 0:00 /usr/sbin/crond -n
 9 ? S 0:00 /bin/sh /usr/bin/mysqld_safe --port=3306
155 ? S1 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) ;
- un processus Crond, qui gère l'exécuteur de tâches pour notre back-end ;
- un processus mysqld\_safe (MariaDB) et son worker mysqld.

ce stade, le processus front-end n'est pas démarré, ce qui est parfaitement attendu vu que le processus dans le fichier supervisor.conf est associé au paramètre suivant :

| autostart=false

### Démarrage du front-end

Nous allons, pour le lancer manuellement, utiliser l'interface graphique de Supervisor, qui est accessible via le port 9001.



Supervisor peut exposer un socket TCP sur lequel 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).

Pour ce faire, ouvrons un navigateur <http://localhost:9001> :

Figure 9.4 – Console d'administration de supervisor

State	Description	Name	Action
Running	pid 8, uptime 0:07:33	crond	Restart Stop Clear Log Tail-f
Stopped	Not started	front-end	Start Clear Log Tail-f
Running	pid 9, uptime 0:07:33	mysql	Restart Stop Clear Log Tail-f

Un clic sur le lien « Start » du service front-end va lancer ce dernier et, après quelques secondes, celui-ci est accessible via le lien <http://localhost:8080>.

### 9.2.5 Quelques explications complémentaires

Pourquoi ne pas avoir démarré le front-end en même temps que les autres processus ?

La première raison est que cela nous permet de vous indiquer qu'une interface d'administration existe.

Dans un contexte de production réelle, cette interface ne serait évidemment jamais ouverte sans protection par mot de passe et HTTPS. Dans la plupart des cas, on préfère d'ailleurs à l'interface graphique la commande supervisorctl, qui permet d'effectuer le même travail en ligne de commande.

Ensuite, pour nous assurer que la base de données soit démarrée « avant » son initialisation (qui se produit au moment du démarrage du front-end). L'un des problèmes de Supervisor est qu'il ne séquence pas les démarrages de ses différents services. Il est possible d'influencer l'ordre des lancements en utilisant des paramètres de priorité mais rien ne garantirait que la base de données soit effectivement démarrée et prête à recevoir des instructions.

Il est bien sûr possible d'utiliser des attentes (comme l'instruction sleep par exemple) mais elle n'offre pas plus de garanties.

Nous verrons par la suite qu'il existe des moyens plus fiables d'assurer le séquencement des démarrages.

### 9.2.6 Gestion des volumes

Dans la version du run.sh qui est disponible par défaut, aucun volume n'est explicitement associé à l'image du conteneur. Chaque redémarrage du conteneur réinitialise complètement la base de données !

En réalité, deux volumes nommés sont créés à chaque démarrage. L'instruction docker volume ls vous permet de le constater. Ces volumes sont néanmoins anonymes et ne sont donc pas réassociés à chaque nouveau conteneur, ce qui entraîne de ce fait la perte des données.

Il existe alors deux options pour assurer la persistance :

- monter des volumes nommés ;
- monter des volumes de l'hôte.

#### *Les volumes nommés*

La première variante fonctionne très simplement en modifiant l'instruction docker run :

```
docker run --rm -d -p 8080:8080 -p 9001:9001  
-v volume_chargement:/chargement -v volume_database:/var/lib/mysql --name=appex-supervisor  
appex-supervisor
```

Un appel à la commande docker volume ls indique que nos volumes nommés ont été automatiquement créés et seront réutilisés à chaque démarrage. Pour réinitialiser l'application, il suffira de détruire ces volumes :

```
$ docker volume rm volume_database  
$ docker volume rm volume_chargement
```

La seconde option nécessite la création de deux répertoires sur l'hôte pour contenir les données :

```
mkdir -p mysql  
mkdir -p chargement  
docker run --rm -d -p 8080:8080 -p 9001:9001  
-v ${PWD}/chargement:/chargement -v ${PWD}/mysql:/var/lib/mysql --name=appex-supervisor appex-supervisor
```

Mais tout ne se passe pas toujours comme prévu...

#### *Volume « bind mount » et droits*

Après le démarrage du conteneur, il semble que quelque chose n'ait pas fonctionné :

Figure 9.5 – Erreur au démarrage de la base de données

State	Description	Name	Action
running	pid 8, uptime 0:00:53	crond	<a href="#">Restart</a> <a href="#">Stop</a> <a href="#">Clear Log</a> <a href="#">Tail -f</a>
stopped	Not started	front-end	<a href="#">Start</a> <a href="#">Clear Log</a> <a href="#">Tail -f</a>
fatal	Exited too quickly (process log may have details)	mysql	<a href="#">Start</a> <a href="#">Clear Log</a> <a href="#">Tail -f</a>

Un accès aux logs du conteneur nous apprend que la base de données n'a pas pu démarrer après trois tentatives :

```
$
...
2018-09-29 15:01:03,488 INFO exited: mysql (exit status 0; not expected)
2018-09-29 15:01:03,488 DEBG received SIGCLD indicating a child quit
2018-09-29 15:01:04,490 INFO gave up: mysql entered FATAL state, too many start retries too quickly
```

Une autre partie des logs nous incite à nous référer au fichier de log de MariaDB :

```
docker exec -ti appex-supervisor cat /var/log/mariadb/mariadb.log
...
180929 15:01:03 InnoDB: Operating system error number 13 in a file operation.
InnoDB: The error means mysqld does not have the access rights to
InnoDB: the directory.
...
```

L'explication est simple : MariaDB n'a pas pu écrire dans le répertoire qui a été monté depuis l'hôte ! MariaDB utilise l'utilisateur mysql et non root. Par défaut, le répertoire contenant les fichiers est la propriété de cet utilisateur :

```
ls -la /var/lib/mysql
total 36892
drwxr-xr-x 5 mysql mysql 177 Sep 29 15:26 .
```

Mais si l'on monte le même répertoire depuis l'hôte :

```
ls -la /var/lib/mysql/
total 0
drwxrwxr-x 2 1000 1000 6 Sep 29 15:37 .
```

1000 est l'ID de notre utilisateur vagrant qui a créé le répertoire sur l'hôte.

```
$ id vagrant
uid=1000(vagrant) gid=1000(vagrant) groupes=1000(vagrant),10(wheel),989(docker)
```

Mais il y a un second problème. Comme nous utilisons un répertoire monté depuis l'hôte, les fichiers créés lors de l'installation de MariaDB et présents dans l'image, dans le répertoire /var/lib/mysql, ont été tout simplement écrasés : le répertoire au démarrage du conteneur est vide !

**Tableau 9.3 – Contenu du répertoire /var/lib/mysql (fichiers de base de données)**

Étape	Contenu du répertoire	Propriétaire
Création de l'image, commande : yum install mariadb-server	Répertoire vide	mysql id=27
Création de l'image, commande : /usr/bin/mysql_install_db ... && chown mysql:mysql /var/lib/mysql	aria_log.00000001 aria_log_control mysql performance_schema test	mysql id=27
Démarrage du conteneur	<>vide>>	vagrant id=1000

Comment corriger cette situation ?

La première partie de la solution consiste à modifier l'utilisateur mysql pour qu'il dispose du même identifiant que l'utilisateur vagrant de l'hôte (1000). Pour ce faire, nous pouvons ajouter l'instruction suivante à notre Dockerfile avant l'installation des paquets MariaDB :

```
| RUN useradd -u 1000 mysql
```

Celle-ci va faire en sorte que l'utilisateur mysql dispose de l'ID dont nous avions besoin.

Ensuite, nous devons « initialiser » le contenu du répertoire de l'hôte avec les fichiers de base de

données créés lors de l'installation de la base.

Pour ce faire, nous allons procéder en trois étapes :

premièrement, reconstruire l'image (après la modification précédente) en utilisant le script

`build.sh` ;

ensuite lancer le script original `run.sh` sans les volumes ;

enfin copier les fichiers présents dans le conteneur dans le répertoire `mysql` que nous avons créé sur l'hôte :

```
| $ docker cp appex-supervisor:/var/lib/mysql mysql
```

Une fois la commande exécutée, vous pouvez relancer le `run.sh` modifié avec le montage des volumes sur le système de fichiers.

Cette fois la base démarre normalement.

D'une manière générale, la gestion des droits est un sujet parfois pénible lorsque les volumes sont des *bind mounts* (depuis l'hôte). De plus, le fait que le contenu des répertoires de l'image soit écrasé oblige à une certaine gymnastique. Les avantages et les inconvénients que nous avons évoqués dans le chapitre précédent entre volumes nommés et hôtes sont donc à revoir en fonction de vos besoins et des compétences des administrateurs système avec lesquels vous travaillez.

### 9.2.7 En conclusion

Nous disposons certes d'une architecture fonctionnelle mais 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), ce qui impose une phase d'installation de paquets longue et complexe à mettre en œuvre car tous les services partagent le même environnement.

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.

Le temps de construction est long et la mise au point fastidieuse. Chaque erreur nécessite de reconstruire l'image complète même si un seul des composants est fautif.

Voyons comment résoudre ces inconvénients avec une approche orientée micro-services.

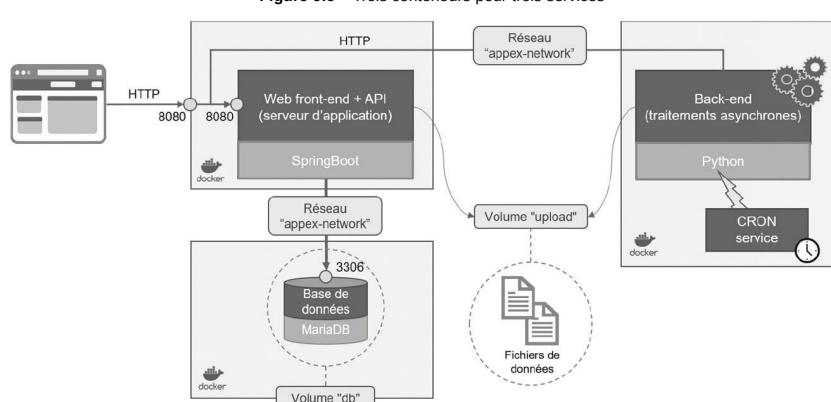
## 9.3 OPTION 2 : APPLICATION MULTI-CONTENEURS

Tous les fichiers dont nous aurons besoin sont disponibles, pour ce paragraphe, dans le répertoire multi-conteneurs du code que vous avez téléchargé de GitHub pour le chapitre 9.

Nous allons maintenant rendre modulaire notre application.

Pour cela, nous utiliserons trois conteneurs comme l'illustre la figure 9.6.

Figure 9.6 – Trois conteneurs pour trois services



Comme nous l'avons appris au chapitre précédent, nos conteneurs seront attachés au même réseau

Docker pour pouvoir communiquer entre eux par nom et ainsi nous affranchir de la configuration de la machine hôte.

Chaque conteneur aura son propre Dockerfile et nous emploierons les images de base disponibles sur le Docker Hub pour nous épargner beaucoup de mises au point (et économiser en temps de *build*).

### 9.3.1 Création de l'image

#### Image MariaDB

Le Dockerfile se trouve dans le répertoire `chapitre9/multi-conteneurs/database/Dockerfile` et est extrêmement simple :

```
FROM mariadb:10.1
COPY schema.sql /schema.sql
```

Cette fois nous utilisons une image officielle MariaDB. Outre l'économie du processus d'installation, celle-ci nous permet d'accéder à une version plus récente du moteur de base de données :

5.5 pour la version standard fournie avec CentOS ;

10.1 pour la version apportée par l'image standard Docker Hub.

Notez que nous chargeons toujours dans l'image le fichier schéma qui servira lors de l'initialisation de la 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.

#### Image front-end

Notre Dockerfile pour le conteneur front-end est relativement simple car il reprend les mêmes commandes que dans notre exemple précédent. La différence principale réside dans la ligne CMD qui lance Java et non plus Supervisor.

```
FROM openjdk:8-alpine
COPY *.jar /app.jar
COPY application.properties /config/application.properties
EXPOSE 8080
ENTRYPOINT java -jar app.jar -Dspring.config.location=/config
```

Cette fois nous utilisons une image de base s'appuyant sur l'OS Alpine Linux qui présente l'avantage d'être extrêmement compact et d'embarquer Java.

Contrairement à ce que nous avions fait dans notre image Supervisor la configuration du composant jar n'utilise pas la configuration par défaut. Elle est externalisée grâce au modificateur SpringBoot :  
- `Dspring.config.location`  
Nous reviendrons sur ce point dans un prochain paragraphe.

#### Image back-end

Là encore, nous nous servons d'une image de base Python compacte :

```
FROM python:3-slim-stretch
RUN apt-get update && apt-get install -y cron
ENV PYTHONIOENCODING=utf-8
RUN echo "* * * * * root /usr/local/bin/python /main.py" >
/etc/cron.d/backend
COPY config.json /config.json
COPY main.py /main.py
ENTRYPOINT /usr/sbin/cron -f
```

## Quelques constats

Il semble évident que la création des images est plus simple. Un lancement du script `build.sh` nous permet aussi de constater que le temps de construction des trois images est plus court (spécialement après la première exécution, une fois que les images de base sont téléchargées).

On pourrait penser que l'usage de trois images résulte en une consommation d'espace disque nettement supérieure. Ce n'est pourtant pas le cas.

Notre image Supervisor occupait 626 MB.

Les trois images construites occupent à peine plus de place (710 MB) :

\$ docker images	REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
	appex-multi/back-end	latest	a288bb52c7c9	7 minutes ago	213MB
	appex-multi/front-end	latest	6bd2ea15f5b3	7 minutes ago	124MB
	appex-multi/database	latest	280ffcabf2e4	5 days ago	373MB

La raison est que nous pouvons utiliser des images de base qui sont plus compactes et réalisées par des experts.

## Gestion de la configuration

L'une des problématiques de la mise en œuvre d'une application multi-conteneurs est qu'il faut s'assurer de la cohérence de la configuration qui se trouve, de fait, distribuée (par exemple, même clé d'API utilisé par le front-end pour l'authentification de l'API appelée par le back-end).

Dans notre fichier `build.sh`, nous utilisons une technique courante : le remplacement. Il s'agit de remplacer des variables présentes dans les fichiers de configuration (`config.json` pour le back-end et `application-properties` pour le front-end).

Par exemple, le fichier `config.json` (présent dans le répertoire `application-exemple/back-end/config-a-replacer/`) est formaté comme suit :

```
{  
    "formulaire.app.chemin.changement.fichiers" : "/chargement",  
    "formulaire.app.clef.api" : "${api.key}",  
    "formulaire.app.url.api" : "http://${front-end.host}:8080/insere_departement"  
}
```

Certains paramètres ne contiennent pas de valeurs mais des variables (exemple : `${api.key}`).

Dans le script `build.sh`, ces valeurs sont remplacées à l'aide d'un petit utilitaire (`replacer.py`) écrit en Python :

```
python ../../application-exemple/build/replacer/replacer.py global.properties front-end/application.properties
```

et

```
python ../../application-exemple/build/replacer/replacer.py global.properties back-end/config.json
```

Celui-ci utilise les données d'un fichier de configuration commun nommé `global.properties`.

À titre d'exemple, le fichier `config.json` résultant du remplacement sera le suivant :

```
{  
    "formulaire.app.chemin.changement.fichiers" : "/chargement",  
    "formulaire.app.clef.api" : "oReiA18NBYITXoxOk7NdoReiA18NBYITXoxOk7Nd",  
    "formulaire.app.url.api" : "http://appex-front-end:8080/insere_departement"  
}
```

S'il existe d'autres techniques (que nous avons évoquées dans le chapitre 8), il est important d'élaborer une stratégie pour extraire les paramètres d'environnement des images. En effet, ceci permet de conserver le même processus de construction d'image pour les environnements de développement, de test ou de production.

### 9.3.2 Lancement de l'application

N'oubliez pas d'arrêter le conteneur créé lors de l'exercice précédent avec `docker stop appex-supervisor`, sinon vous obtiendrez un message d'erreur indiquant que le port 8080 est déjà en cours d'utilisation.

Le script `run.sh` est plus compliqué que la version Supervisor. Il est en effet nécessaire de créer le réseau et les deux volumes nommés avant de lancer les conteneurs.

```
docker network create appex-network  
docker volume create --name=appex-db-volume  
docker volume create --name=appex-upload-files
```

Ces derniers sont ensuite associés à chaque conteneur.

Prenons l'exemple de la base de données :

```
docker run --rm -d --name appex-db -p 3306:3306 --net=appex-network -v appex-db-volume:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD} appex-multi/database
```

Vous noterez qu'ici l'initialisation de la base de données est réalisée par une commande, depuis l'hôte, lancée 20 secondes après le démarrage du conteneur `database` (afin de s'assurer que cette dernière est effectivement disponible).

```
sleep 20  
docker exec appex-db /bin/bash -c "/usr/bin/mysql -v -u root -p${MYSQL_ROOT_PASSWORD} < /schema.sql"
```

Voici enfin les deux autres conteneurs :

```
docker run --rm -d --name appex-front-end -p 8080:8080 --net=appex-network -v appex-upload-files:/chargement appex-multi/front-end  
docker run --rm -d --name appex-back-end --net=appex-network -v appex-upload-files:/chargement appex-multi/back-end
```

Notez que, comme prévu, le front-end et le back-end partagent bien le même réseau et le même volume (utilisé pour partager des fichiers).

### 9.3.3 En conclusion

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 : Compose.

## 9.4 OPTION 3 : ORCHESTRATION AVEC COMPOSE

Tous les fichiers dont nous aurons besoin sont disponibles, pour ce paragraphe, dans le répertoire `compose` du code que vous avez téléchargé de GitHub pour le chapitre 9.

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 manière coordonnée.

Compose est le module d'orchestration de la suite Docker. Nous l'avons déjà évoqué dans le chapitre 2. À noter que d'autres solutions CaaS, comme Kubernetes, disposent de leur propre module d'orchestration. Compose reste néanmoins très utilisé par la communauté en raison de sa simplicité.

Regardons comment mettre cela en œuvre pour notre application.

### 9.4.1 Introduction et premiers pas avec 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 Desktop), sous Linux et 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/1.22.0/docker-compose-$(uname -s)-$(
uname -m) -o /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose
docker-compose --version
docker-compose version 1.22.0, build f46880fe
```

Compose recherche par défaut un fichier de configuration dans le répertoire courant, nommé [docker-compose.yml](#), au format YAML<sup>5</sup>. 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 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 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 leurs contenus.

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 Construction des images

Aucun changement par rapport à la précédente option. L'usage de Compose change uniquement le démarrage. Nous réutilisons ici les trois images construites précédemment.

#### 9.4.3 Le fichier de déploiement Compose

Nous allons utiliser la version 3 du format du fichier 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.

```
version: "3"
services:
  database:
    image: "appex-multi/database"
    container_name: appex-db
    environment:
      - MYSQL_ROOT_PASSWORD
    networks:
      - appex-network
    volumes:
      - appex-db-volume:/var/lib/mysql

  initdb:
    build: "initdb"
    environment:
      - MYSQL_ROOT_PASSWORD
    command: /bin/sh -c '/wait-for-it.sh appex-db:3306 -t 20 -- /usr/bin/mysql -v
-h appex-db -u root -p${MYSQL_ROOT_PASSWORD} < /schema.sql'
    depends_on:
      - database
    networks:
      - appex-network

  front-end:
```

```

image: "appex-multi/front-end"
container_name: appex-front-end
depends_on:
- initdb
ports:
- "8080:8080"
networks:
- appex-
network volumes:
- appex-upload-files:/chargement

back-end:
image: "appex-multi/back-end"
container_name: appex-back-end
depends_on:
- initdb
networks:
- appex-network
volumes:
- appex-upload-files:/chargement

networks:
appex-network:

volumes:
appex-upload-files:
appex-db-volume:

```

Nous retrouvons les faits suivants :

- nous utilisons la version 3 du format Compose ;
- un bloc services : contient la définition de nos trois conteneurs ;
- pour chaque conteneur, nous avons un paramètre `image` qui permet de spécifier l'image du Docker Hub à utiliser ;
- le paramètre `depends_on` permet de nous assurer que le conteneur de base de données sera démarré avant le conteneur front-end ;
- le fichier se termine par la déclaration des volumes et du réseau ;
- le reste des paramètres est relativement simple à comprendre et correspond dans la syntaxe aux paramètres de la ligne de commande Docker.

Vous noterez l'usage de la variable d'environnement : `MYSQL_ROOT_PASSWORD`. La valeur de celle-ci est fournie dans le fichier `.env` placé dans le même répertoire que `docker-compose.yml`. Il s'agit du moyen proposé par Compose pour paramétriser un fichier de déploiement.

#### 9.4.4 Synchronisation des conteneurs

L'instruction `depends_on` permet de s'assurer que les conteneurs sont démarrés dans un certain ordre. Malheureusement, cette seule garantie n'est pas suffisante. Il est en effet possible que le conteneur de base de données soit démarré alors même que la base n'est pas encore en mesure de recevoir des commandes via son port 3306.

Ici nous utilisons un « conteneur » intermédiaire qui disparaît immédiatement après usage :

```

initdb:
build: "initdb"
environment:
- MYSQL_ROOT_PASSWORD
command: /bin/sh -c '/wait-for-it.sh appex-db:3306 -t 20 -- /usr/bin/mysql -v -h appex-db -uroot -p${MYSQL_ROOT_PASSWORD} < /schema.sql'

```

```

depends_on:
  - database
networks:
  - appex-network

```

Celui-ci utilise un petit script<sup>6</sup> (`chapitre9/compose/initdb/wait-for-it.sh`) qui permet de bloquer l'initialisation de base jusqu'à ce que le port 3306 de MariaDB réponde correctement. Intercalé entre le conteneur de la base de données et ceux des autres conteneurs, il garantit la disponibilité de la base de données.

Notez que l'image « `initdb` » est construite automatiquement par Compose juste avant le démarrage des conteneurs. Le fichier Dockerfile hérite de l'image de base MariaDB afin de disposer des outils nécessaires pour réaliser l'initialisation.

#### 9.4.5 Démarrage de l'application

Démarrons notre service avec notre script `run.sh`.

Celui-ci comprend pour l'essentiel la commande suivante :

```
| $ docker-compose up -d
```

Cette commande va :

- créer les images de nos services à partir de nos différents Dockerfiles ;
- démarrer, pour chaque image, 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 les réseaux (de type *bridge*) et volumes nécessaires à l'application.

Nous pouvons lister nos conteneurs :

```
$ docker-compose ps
Name Command State Ports
-----
appex-back-end /bin/sh -c /usr/sbin/cron -f Up
appex-db docker-entrypoint.sh mysqld Up 3306/tcp appex-front-end /bin/sh
-c java -jar app.j ... Up 0.0.0.0:8080->8080/tcp compose_initdb_1
docker-entrypoint.sh /bin/ ... Exit 0
```

Le conteneur `compose_initdb_1` est créé à partir de l'image `initdb` par Compose. Comme vous pouvez le noter, il est arrêté puisque son rôle se limite à l'initialisation de la base.

Un des avantages majeurs de 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, 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.

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 9.4 – Commandes Docker Compose**

Commande	Résultat
<code>docker-compose logs</code>	Affiche une vue agrégée de tous les logs de notre application.
<code>docker-compose start</code> / <code>docker-compose stop</code>	Démarre/arrête l'ensemble des conteneurs de notre application.
<code>docker-compose pause</code> / <code>docker-compose</code>	Met en pause/relance les processus qui tournent dans les conteneurs de notre

unpause	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.



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, notamment pour les applications existantes qu'on souhaite reconditionner sous la forme d'un petit conteneur pour en automatiser le déploiement. Nous avons ensuite vu comment implémenter une application à base de micro-services. Enfin, nous nous sommes penchés sur Compose, l'outil d'orchestration de la suite Docker pour centraliser et automatiser la création de nos services.

- 
1. <https://gradle.org/>
  2. <http://supervisord.org/>
  3. Extra Packages for Enterprise Linux, <https://fedoraproject.org/wiki/EPEL>
  4. <https://fr.wikipedia.org/wiki/Cron>
  5. <http://www.yaml.org/>
  6. Le code original, open source, de ce script se trouve dans le dépôt GitHub suivant :  
<https://github.com/vishnubob/wait-for-it>

# 10

## Intégration continue avec Docker

### Objectif

Dans le chapitre précédent nous nous sommes attachés à montrer différentes options de déploiement pour notre application exemple. Nous allons maintenant nous intéresser au développement d'application à base de conteneurs. Pour ce faire nous allons mettre en place un système d'intégration continue (dont l'acronyme anglais est CI, pour *Continuous Integration*) reposant sur des conteneurs Docker. Ce chapitre comprend :

un exemple de mise en place d'une chaîne de CI simplifiée étape par étape ;  
une courte présentation de variantes et extensions à ce cas simplifié relative aux différentes problématiques de l'intégration continue.



### Application exemple et code source

L'application que nous allons utiliser dans ce chapitre est celle qui a été présentée dans le chapitre 8. Nous vous demanderons d'ailleurs de lancer l'application multi-conteneur à l'aide de l'implémentation Compose du chapitre 9 car c'est elle que nous allons mettre à jour au cours de nos exemples.

Le code source et les exemples de ce chapitre sont disponibles sur GitHub.

Veuillez vous reporter à la procédure d'installation du chapitre 3.

## 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 continu (*continuous deployment*, 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 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<sup>1</sup>;

un outil de **gestion de code source** permettant de versionner le code d'une application. Pour notre exemple, nous utiliserons GitLab<sup>2</sup>, 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. Typiquement un registry docker. Dans ce chapitre, nous nous limiterons au cache local de l'hôte.

### 10.1.2 Un exemple simplifié

Nous ne pourrions pas en l'espace d'un chapitre aborder l'ensemble des sujets relatifs à l'intégration continue. Tel n'est d'ailleurs pas le propos de cet ouvrage.

Nous allons plus spécifiquement nous intéresser à la manière dont Docker peut intervenir pour simplifier certaines tâches. On pense immédiatement au déploiement mais Docker peut aussi être utile à la compilation.

ce titre, notre chaîne sera limitée au module front-end de l'architecture multi-conteneurs du chapitre 9. Comme ce composant est codé en Java, il requiert une phase de compilation qui nous permet d'aborder plus de problématiques.

La figure 10.1 représente la chaîne de CI que nous allons décrire :

Le code source est modifié par le développeur.

Les modifications sont poussées (*git push*) dans un dépôt géré par Gitlab.

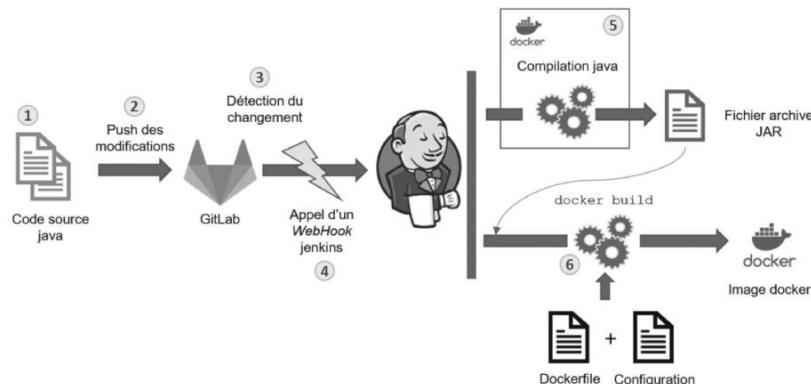
Ce dernier détecte les changements.

Gitlab notifie Jenkins (notre moteur de *build*) en utilisant un *webhook* qui déclenche un processus de *build* en plusieurs étapes.

Dans un premier temps, le code source modifié est compilé en utilisant une image Docker dédiée à cet usage et contrôlée depuis Jenkins.

Dans un second temps, le produit de la compilation (une archive Java : JAR) est utilisé pour produire une image Docker. Cette phase est aussi associée à la sélection de la configuration correspondant à l'environnement (DEV, QA, PROD).

Figure 10.1 – Une chaîne de CI simplifiée



#### 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*.

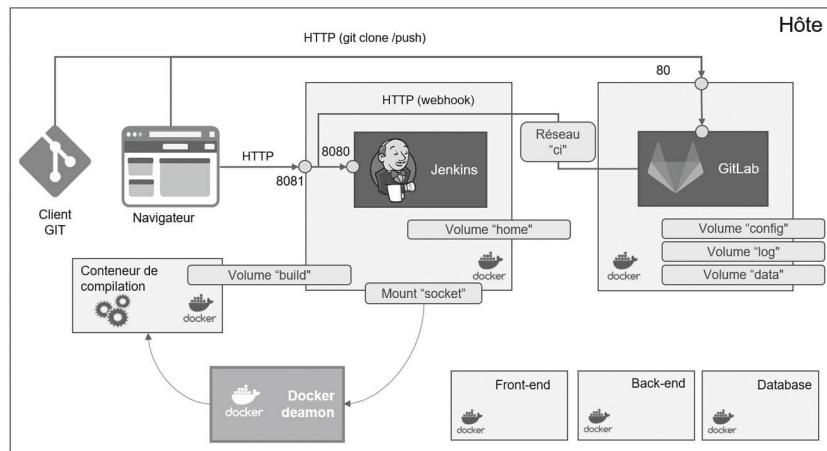
Par souci de simplification, nous ne respectons pas dans notre exemple le Git flow « classique ». Dans notre projet exemple,

l'ensemble des modifications seront poussées dans la branche master. Nous traiterons néanmoins des techniques de *release management* dans la seconde partie de ce chapitre en lien avec la production d'images Docker.

## 10. 2 UN ENVIRONNEMENT DE *BUILD* LUI-MÊME DOCKERISÉ

Notre environnement de *build* est lui-même réalisé sous la forme de conteneurs (figure 10.2).

Figure 10.2 – Un environnement de CI complet



### 10.2.1 Les principaux conteneurs

Les tableaux 10.1 et 10.2 décrivent l'ensemble des conteneurs et pour chacun d'eux les volumes et réseaux qui sont utilisés.

#### *Les conteneurs applicatifs*

Le propos d'une CI est de maintenir en état de marche constant une application à chaque modification du code source. Nous aurons donc besoin de l'application élaborée au chapitre 9 (que vous lancerez en utilisant le `run.sh` de l'option 3 basée sur Compose).

Tableau 10.1 – Les conteneurs applicatifs

Conteneur	Description
Front-end	Conteneur front-end de l'architecture multi-conteneur présentée dans le chapitre 9. C'est ce composant dont nous allons gérer les modifications avec notre CI. <b>Port</b> : 8080 (interface graphique web)
Back-end	Conteneur back-end présenté dans le chapitre 9.
Database	Conteneur de base de données présenté dans le chapitre 9.

#### *L'environnement de build*

Cet environnement est constitué de trois conteneurs.

Tableau 10.2 – Les conteneurs utilisés pour le *build*

Conteneur	Description
GitLab	Un conteneur exécutant une instance de l'outil GitLab assurant la gestion du code source et offrant une interface graphique d'administration et de configuration. <b>Port</b> : 80 (interface graphique web) <b>Volumes</b> : trois volumes (ci-gitlab-config-volume, ci-gitlab-log-volume, ci-gitlab-data-volume)

	<p><b>data-volume</b>) dédiés au fonctionnement de l'outil assurant la persistance du référentiel de code et de la configuration.</p> <p><b>Réseau</b> : le réseau <code>ci-network</code> partagé avec Jenkins</p>
Jenkins	<p>Un conteneur exécutant l'outil d'intégration continue Jenkins. Il exécute les <i>jobs</i> de <i>build</i> sur demande de GitLab (ou de l'utilisateur via l'interface graphique).</p> <p><b>Port</b> : 8081 (interface graphique web)</p> <p><b>Volumes</b> :</p> <ul style="list-style-type: none"> <li>– un volume dédié à la persistance de la configuration et des logs de l'outil (<code>ci-jenkins-home</code>) ;</li> <li>– un volume partagé avec le conteneur de <i>build</i> Java (<code>ci-jenkins-build</code>) qui permet d'échanger le code source et le résultat de la compilation avec Jenkins ;</li> <li>– un <i>bind mount</i> (<code>/var/run/docker.sock</code>) permettant à Jenkins d'accéder au moteur Docker de l'hôte.</li> </ul> <p><b>Réseau</b> : le réseau <code>ci-network</code> partagé avec Gitlab</p>
Build java	<p>Un conteneur éphémère assurant la compilation Java.</p> <p><b>Volumes</b> : un volume partagé Jenkins (<code>ci-jenkins-build</code>) qui permet d'échanger le code source et le résultat de la compilation.</p>

Voyons maintenant comment construire cet environnement de *build*.

### 10.2.2 Préparation des images de l'environnement de *build*

Tous les fichiers dont nous aurons besoin sont présents dans le répertoire `chapitre9` du code que nous avons téléchargé depuis GitHub.

Pour construire et lancer cet environnement, nous nous appuierons sur Compose. Comme nous l'avons vu dans le chapitre précédent, Compose permet de simplifier le déploiement de solutions multi-conteneurs et même de prendre en charge la création des images.

#### Image GitLab

Le Dockerfile permettant de créer l'image GitLab se trouve dans le répertoire `gitlab`. Nous nous appuyons sur une image standard de ce produit que nous modifions juste pour nous assurer que le nom de domaine fonctionne sur l'hôte.

```
FROM gitlab/gitlab-ce:8.5.8-ce.0
COPY gitlab.rb /etc/gitlab/gitlab.rb
```

Le fichier de configuration `gitlab.rb` n'est modifié que d'une ligne par rapport à l'installation par défaut :

```
| external_url 'http://localhost:80'
```

Cette modification est nécessaire pour permettre à Gitlab d'afficher les liens des dépôts correctement.

#### Image Jenkins

Le Dockerfile utilisé pour Jenkins hérite aussi d'une image standard mais nécessite un peu plus de modifications. Nous allons en effet installer dans cette image le client Docker. Celui-ci nous permettra d'accéder au moteur Docker de l'hôte pour déclencher des commandes `docker build` et `docker run`.



#### Docker in Docker (DIND)

Historiquement certains environnements de *build* utilisaient une technique nommée « Docker in Docker »<sup>3</sup>. Celle-ci permet d'installer un démon docker dans un conteneur et de pouvoir créer des conteneurs fils à partir d'un conteneur parent. À l'origine, ce « hack » a été élaboré par l'équipe Docker elle-même pour simplifier le développement du démon Docker en le... virtualisant.

Aujourd'hui, cette pratique est largement abandonnée pour diverses raisons fort bien décrites dans un article de Jérôme Petazzoni<sup>4</sup>.

Ce n'est pas la technique que nous utilisons. Nous ne faisons qu'installer un client Docker dans notre conteneur Jenkins pour lui permettre d'accéder au démon de l'hôte. Notre conteneur Jenkins ne créera donc pas de conteneurs fils mais des conteneurs « frères » qui s'exécutent au même niveau que lui sur l'hôte.

Le Dockerfile Jenkins se trouve dans le répertoire `jenkins` du chapitre 10. Celui-ci est constitué des trois parties distinctes :

Une première section installe le client Docker sous l'utilisateur root :

```
USER root
RUN apt-get update && \
apt-get -y install docker-ce
RUN apt-get update \
apt-get install -y sudo \
rm -rf /var/lib/apt/lists/*
RUN echo "jenkins ALL=NOPASSWD: ALL" >> /etc/sudoers
```

Certains auront noté que nous allouons les droits sudo à l'utilisateur jenkins. Cette allocation de droit est nécessaire pour permettre au conteneur de se connecter au socket de l'hôte et donc au démon Docker. Nous convenons du fait que la chose présente des faiblesses en termes de sécurité. Jenkins dispose aussi d'un plugin pouvant utiliser l'API HTTPS de Docker dont l'usage n'est pas non plus exempt de risques. D'une manière générale, un environnement de CI est un environnement qui doit être correctement isolé de l'environnement de production. Même dans le cas de déploiement continu (*continuous deployment*), la mise en place de sas (type jump box) avec l'environnement de production est une bonne pratique à respecter.

Une seconde partie prépare le contenu du volume `ci-jenkins-build` qui servira d'espace de compilation en s'appuyant sur le conteneur de *build Java* (que nous allons décrire dans quelques instants) :

```
RUN mkdir -p /opt/build/jobs
COPY scripts /opt/build/scripts
RUN chown -R jenkins:jenkins /opt/build
```

Notez que nous installons des scripts qui seront utilisés par les jobs Jenkins. Nous les décrirons plus tard au moment où nous aborderons l'écriture des jobs.

Une troisième et dernière partie, sous l'utilisateur jenkins pour permettre l'exécution desdits scripts par Jenkins justement :

```
USER jenkins
RUN chmod u+x /opt/build/scripts/*.sh
```

### ***Le descripteur Compose***

Nous sommes maintenant en mesure d'écrire le descripteur Compose de notre environnement.

La première partie est consacrée à GitLab :

```
version: "3"
services:
gitlab:
  build: "gitlab"
  container_name: gitlab
  ports:
    - "80:80"
    - "8122:22"
  networks:
    - ci-network
  volumes:
    - ci-gitlab-config-volume:/etc/gitlab
    - ci-gitlab-log-volume:/var/log/gitlab
    - ci-gitlab-data-volume:/var/opt/gitlab
```

Notez que nous délégons (via l'instruction `build`) à Compose la gestion de la création des images au démarrage.

Attention : par défaut, Compose ne recompile pas les images à chaque démarrage, même si les Dockerfiles ont changé. Pour pallier cette contrainte, notre script de `run.sh` (dans le répertoire `chapitre10`) utilise l'instruction `docker-compose up -d --build`. Celle-ci, grâce au modificateur `--build` force la recompilation des images à chaque lancement.

Vient ensuite le tour du conteneur Jenkins :

```
jenkins:
```

```

build: "jenkins"
container_name: jenkins
hostname: jenkins
networks:
- ci-network
ports:
- "8081:8080"
- "50000:50000"
volumes:
- /var/run/docker.sock:/var/run/docker.sock
- ci-jenkins-home:/var/jenkins_home
- ci-jenkins-build:/opt/build

```

Nous voyons ici le montage de la socket docker.sock de l'hôte. C'est par son intermédiaire que le client Docker installé dans le conteneur Jenkins pourra accéder au démon Docker de l'hôte.

Et pour finir les réseaux et volumes :

```

networks:
ci-network:

volumes:
ci-gitlab-config-volume:
ci-gitlab-log-volume:
ci-gitlab-data-volume:
ci-jenkins-home:
ci-jenkins-build:

```

Nous pouvons maintenant lancer notre application à l'aide du script run.sh. Vous disposez aussi d'un script stop.sh pour stopper les conteneurs et d'un script clean.sh qui efface les volumes pour redémarrer un environnement (au prix de la perte de l'ensemble des données).

## 10.3 INSTALLATION DES OUTILS ET CHARGEMENT DU CODE SOURCE

Cette section présente la configuration de GitLab et Jenkins (au travers de leurs interfaces graphiques respectives) et le chargement initial du code source.

### 10.3.1 Configuration de GitLab

Connectez-vous à l'adresse <http://localhost/>. La page d'accueil de GitLab s'ouvre et vous invite à vous connecter (figure 10.3).

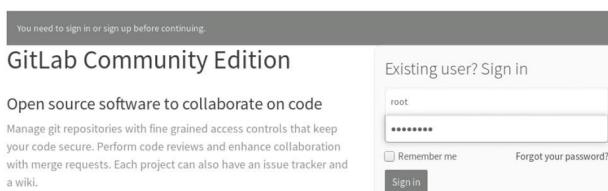
Le démarrage de GitLab peut être relativement long la première fois. Soyez patient !

Connectez-vous avec l'utilisateur suivant :

Nom d'utilisateur : root

Mot de passe : 5iveL!fe

Figure 10.3 – Écran d'accueil GitLab



Après la première connexion, GitLab nous propose de changer de mot de passe root, et nous utiliserons arbitrairement un mot de passe simple : passw0rd. Après la modification, nous devons nous reconnecter.

Créons maintenant le projet `appex-front-end` (figure 10.4), qui contiendra le code source de notre front-end :

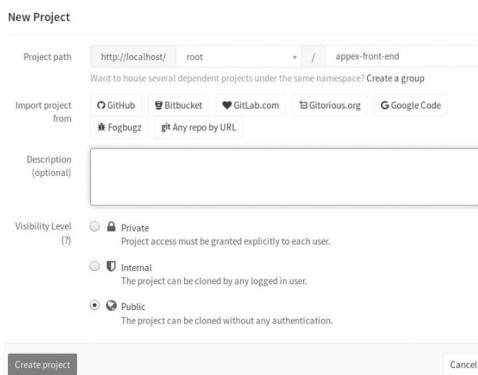
Cliquez sur *New project*.

Sous *Project path*, saisissez « `appex-front-end` ».

Pour *Visibility Level*, choisissez *Public*.

Cliquez sur *Create Project*.

Figure 10.4 – Création d'un projet GitLab



Réitérez la même opération pour le projet `appex-front-end-configuration`, qui contiendra les fichiers de configuration pour les différents environnements.

Le code est prêt à être chargé !

### 10.3.2 Chargement initial du code source

Nous fournissons deux scripts pour le chargement du code source et de la configuration :

- ✓ code source :  
`chapitre10/environnement_de_developpement/code_source/push_code.sh`
- ✓ configuration :  
`chapitre10/environnement_de_developpement/fichiers_configuration/push_fi`

Dans les deux cas, on vous demandera d'entrer les paramètres d'authentification suivants (ceux de l'utilisateur `root` de GitLab) :

Username : `root`

Password : `passw0rd`

Ces scripts font appel à des commandes Git classiques pour charger les fichiers Java dans le projet `appex-front-end` et des fichiers de configuration dans le projet `appex-front-end-configuration`.

Ces derniers sont nommés comme suit :

```
| <>nom de l'environnement (DEV, QA, PROD)>_application.properties
```

Dans une situation réelle, l'accès à ce projet serait restreint. Une telle pratique d'isolation des configurations dépendantes de l'environnement de la construction de l'image est générique. Nous verrons qu'il est ainsi possible de créer rapidement des variantes de notre job de `build` pour chaque type d'environnement.

### 10.3.3 Configuration de Jenkins

Connectez-vous à l'adresse `http://localhost:8081/`. La page d'accueil de Jenkins s'ouvre et vous invite à entrer un code (figure 10.5).

Figure 10.5 – Déblocage de Jenkins

## Débloquer Jenkins

Pour être sûr que que Jenkins soit configuré de façon sécurisée par un administrateur, un mot de passe a été généré dans le fichier de logs ([où le trouver](#)) ainsi que dans ce fichier sur le serveur :

```
/var/jenkins_home/secrets/initialAdminPassword
```

Veuillez copier le mot de passe depuis un des 2 endroits et le coller ci-dessous.

Mot de passe administrateur

```
*****
```

Pour récupérer le mot de passe qui se trouve dans le conteneur, nous pouvons une fois de plus faire appel à Docker (cf. script `chapitre10/jenkins/get_key.sh`) :

```
| docker exec -ti jenkins cat /var/jenkins_home/secrets/initialAdminPassword
```

Vous êtes aussi invité à choisir les plugins à installer. Choisissez la liste par défaut avec « Installez les plugins suggérés » :

**Figure 10.6 – Installation des plugins**

## Personnaliser Jenkins

Les plugins étendent Jenkins avec des fonctionnalités additionnelles pour satisfaire différents besoins.



La procédure d'installation est relativement longue mais ne requiert pas d'intervention particulière.

Les écrans qui suivent vous invitent à créer un utilisateur. Pour ce faire utilisez les valeurs suivantes :

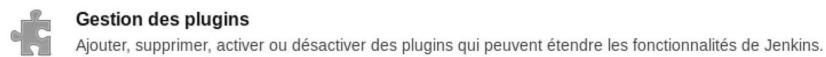
Nom d'utilisateur : admin

Mot de passe : passw0rd

Et confirmez ensuite l'URL à utiliser `http://localhost:8081/`.

Ensuite allez dans le menu « Administrer Jenkins » et choisissez l'option « Gestion des plugins » (figure 10.7).

**Figure 10.7 – Installation d'un plugin spécifique**



Installez ensuite le plugin « *PostBuildScript Plugin* » qui se trouve sous la liste des plugins installables (figure 10.8) :

**Figure 10.8 – Plugin de post-exécution**



Ce plugin nous permet d'exécuter un script à la fin d'un job, qu'il soit un succès ou un échec. Nous l'utiliserons pour vider notre espace de *build* à la fin de chaque job.

## 10. 4 IMAGE ET JOB DE BUILD

Il est maintenant temps de construire notre job de CI. Celui-ci, comme nous l'avons vu dans la figure 10.1, est constitué de plusieurs étapes distinctes. Ces étapes vont utiliser un espace de travail sur le volume `ci-jenkins-build`.

#### 10.4.1 Le volume de travail

Le volume `ci-jenkins-build` contient l'arborescence décrite dans le tableau 10.3.

**Tableau 10.3 – Arborescence de l'espace de build dans le volume ci-jenkins-build**

/opt/build	scripts	Répertoire contenant les scripts intervenant dans le job de <i>build</i> . Nous les étudierons un par un.		
	jobs	Répertoire contenant les espaces des différents jobs de <i>build</i> . Chaque job crée un sous-répertoire nommé selon le nom (unique) du job généré par Jenkins.		
		<>job id>> (ex. : CI_BUILD_DEV-12)	Cet espace de travail contient le code source du composant front-end mais aussi le résultat de la compilation Java.	
			appex-front-end	Le projet qui contient le code source du composant (pris depuis GitLab).
			output	Le répertoire qui contient le résultat de la compilation Java.

Ce volume sert d'espace de travail pour Jenkins lors du docker `build` mais aussi d'échange avec le conteneur de compilation Java.

#### 10.4.2 Le conteneur de build Java

L'une des problématiques courantes des systèmes d'intégration continue est la stabilité des environnements dans le temps et leur gestion de configuration. Si vous utilisez une machine pour compiler puis tester un composant, vous devez vous assurer d'une parfaite traçabilité de sa configuration :

- version d'OS et packages installés ;
- configuration réseau ;
- arborescence de fichiers ;
- cache de dépendances ;
- etc.

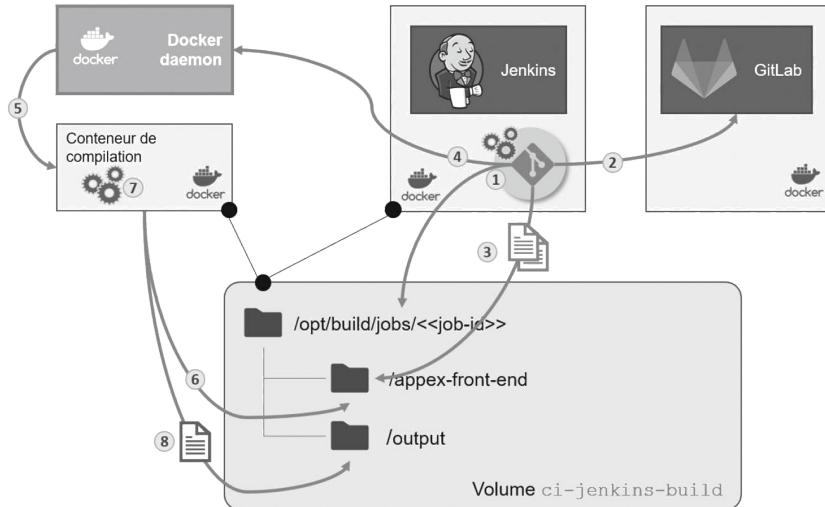
Le risque est de passer à côté d'un bug en raison d'une différence de configuration. Les habitués de Java ont tous vécu au moins une fois des soucis de cache de bibliothèques logicielles ou de version de JDK.

Docker offre plusieurs fonctionnalités intéressantes à ce titre :

- la description formelle de la configuration des conteneurs grâce au Dockerfile ;
- la possibilité de détruire et reconstruire les conteneurs à loisir pour garantir la même configuration de départ.

Pour compiler notre front-end SpringBoot, nous allons donc utiliser une image `appex-build-java` dont le Dockerfile se trouve dans le répertoire `chapitre10/build-image`.

Figure 10.9 – Séquence d'actions du *build Java*



La figure 10.9 expose le workflow qui implique différents conteneurs pour aboutir à la version compilée de l'application front-end (un JAR, c'est-à-dire une archive Java SpringBoot) :

Jenkins lance le job de *build*, qui commence par créer un répertoire de travail dans `/opt/build/jobs`. Comme nous le verrons plus tard, ce répertoire est effacé en fin de processus.

Celui-ci effectue un `git clone` pour aller chercher le code source stocké par GitLab.

Le job place les fichiers sources dans le répertoire `/opt/build/jobs/<>job-id>>/appex-front-end`.

Le job utilise le client Docker de l'image Jenkins pour communiquer avec le démon Docker de l'hôte et exécuter un `docker run`.

Le démon lance une instance de l'image `appex-build-java`.

Le conteneur ainsi créé accède au code source présent dans `/opt/build/jobs/<>job-id>>/appex-front-end`. Il le peut car il partage le volume `ci-jenkins-build` avec Jenkins.

La compilation java est exécutée. Elle nécessite le téléchargement des dépendances Java (usuuellement via Maven Central<sup>5</sup>). Elle peut aussi inclure des tests unitaires et fonctionnels.

En cas de succès, l'archive JAR résultante est placée dans le répertoire `/opt/build/jobs/<>job-id>>/` et est donc accessible à Jenkins pour la suite des opérations.

Notre conteneur `appex-build-java` utilise l'image de base `openjdk:8-alpine`, que nous avons déjà utilisé dans le chapitre 9 pour le front-end.

```
FROM openjdk:8-alpine
VOLUME /opt/build
RUN adduser -D -u 1000 jenkins
ENTRYPOINT cd /opt/build/jobs/${BUILD_ID} \
    mkdir -p output \
    cd ${PROJECT_NAME} \
    chmod u+x gradlew \
    ./gradlew clean build \
    cp build/libs/*.jar /opt/build/jobs/${BUILD_ID}/output/ \
    chown -R jenkins:jenkins /opt/build/jobs/${BUILD_ID}
```

Notez que nous créons un utilisateur `jenkins` avec le même ID que celui de l'utilisateur `jenkins` du conteneur Jenkins. L'objectif est de s'assurer que les échanges entre les deux conteneurs soient possibles en termes de droits. Jenkins pourra donc effacer les fichiers produits par le conteneur de *build*.

Il est possible d'optimiser ce job de *build* en cachant les dépendances à l'aide de volumes Docker (pour les chemins `~/m2` et `~/gradle`). Celles-ci ne seraient pas re-téléchargées à chaque *build*.

### 10.4.3 Configuration du job dans Jenkins

Pour configurer le job, il suffit de se connecter à l'interface graphique de Jenkins et d'initier la création d'un nouveau job via le menu « Nouveau Item » ou « Créer un nouveau job » (si vous n'en avez jamais créé).

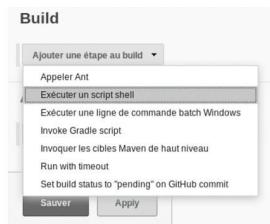
Choisissez un projet de type « free style » (figure 10.10).

Figure 10.10 – Crédit de projet/job de build



Vous pouvez ensuite ajouter un script *shell* en tant que tâche du job (figure 10.11).

Figure 10.11 – Ajout d'une étape de build



Puis entrer la commande suivante dans la zone prévue à cet effet (figure 10.12).

```
/opt/build/scripts/start-build.sh ${BUILD_TAG}  
/opt/build/scripts/build-java.sh ${BUILD_TAG}  
/opt/build/scripts/build-configure.sh ${BUILD_TAG} "DEV"  
/opt/build/scripts/build-front-end-image.sh ${BUILD_TAG}
```

Nous allons expliquer le contenu de ces scripts dans quelques instants.

Figure 10.12 – Édition du script shell



Quel que soit le résultat du *build*, même si celui-ci échoue, nous aimerais nous assurer que le répertoire de travail (/opt/build/jobs/<>job-id>/) est bien effacé. Pour ce faire, nous ajoutons une étape de *post-build* qui va effectuer cette opération (figure 10.13).

```
| /opt/build/scripts/end-build.sh ${BUILD_TAG}
```

Si vous ne voyez pas l'option « script » pour les « Actions à la suite du build », il est possible que vous ayez manqué l'installation du plugin requis. Reportez-vous au paragraphe d'installation plus haut.

Figure 10.13 – Script de post-build

Actions à la suite du build



Nous sommes maintenant prêts à lancer notre premier job en pressant sur l'icône prévue à cet effet « Lancer un build ». Si tout se passe correctement après quelques secondes (peut-être un peu plus la première fois), le job se termine par un succès (figure 10.14).

Figure 10.14 – Job réussi

Tous	+	S	M	Nom du projet	Dernier succès	Dernier échec	Dernière durée
		CI_BUILD_DEV			10 h - #13	13 h - #11	1 mn 6 s

Pendant le *build*, il est possible de cliquer sur le job en cours et de visualiser en temps réel les logs (figure 10.15). La même fonction est disponible en cas d'échec.

Figure 10.15 – Visualisation des logs du job

The screenshot shows the Jenkins interface for the CI\_BUILD\_DEV job. The left sidebar has links like Retour au projet, État, Modifications, Console Output (selected), View as plain text, Informations de la construction, Supprimer le build, and Build précédent. The main area is titled "Sortie de la console" and shows the command-line output of build #13. It includes commands like "Started by user Administrateur", "Building in workspace /var/jenkins\_home/workspace/CI\_BUILD\_DEV", and "Cloning into 'appex-front-end'...". The output ends with "Downloading https://services.gradle.org/distributions/gradle-4.8.1-bin.zip".

#### 10.4.4 Les scripts /opt/build/scripts

Comme vous l'avez noté, le job que nous avons précédemment configuré utilise des scripts qui ont été préinstallés dans l'image Jenkins (tableau 10.4).

Tableau 10.4 – Scripts de build

Script	Description
start-build.sh	Crée le répertoire de travail (/opt/build/jobs/<>job-id>/) en utilisant le nom unique du job qui lui est passé en paramètre (\${BUILD_TAG} dans Jenkins).
build-java.sh	Effectue la séquence d'opération décrite dans la figure 10.9. Vous noterez dans le code que nous utilisons l'instruction sudo docker run. L'accès au socket du démon Docker nécessitant les droits root, l'utilisateur jenkins se doit d'utiliser sudo.
build-configure.sh	Ce script télécharge via un git clone la version du fichier de configuration correspondant à l'environnement cible présente dans le projet GitLab appex-front-end-configuration que nous avons créé précédemment. Dans le cas du job créé dans la figure 10.12, on voit que nous utilisons l'environnement « DEV ». Créer un autre job pour l'environnement « QA » ou « PROD » serait donc simple et ne changerait en rien le reste du processus.
build-front-end-image.sh	Ce script crée l'image de déploiement du front-end. Nous allons détailler ce processus dans le paragraphe suivant.
end-build.sh	Cette dernière étape du script (post-build) efface le répertoire /opt/build/jobs/<>job-id>/ à l'issue du job, que celui-ci ait réussi ou pas.

#### 10.4.5 Crédit de l'image de déploiement

Cette étape est le fait du script build-front-end-image.sh. Elle s'appuie aussi sur Docker, pas pour exécuter un docker run, comme lors de la compilation Java, mais cette fois pour lancer un docker build.

```
cp Dockerfile /opt/build/jobs/${BUILD_ID}/output
sudo docker build -t appex-multi/front-end --label build=${BUILD_ID} /opt/build/jobs/${BUILD_ID}/output
```

La séquence d'action est simple.

Dans un premier temps, le script copie le Dockerfile. Il s'agit du même Dockerfile qui a été utilisé dans

le chapitre 9. Ensuite l'image appex-multi/front-end est créée avec l'habituelle commande docker build.

Notez cependant l'usage de `--label`, permettant d'enregistrer dans l'image le nom du *build* qui a créé cette image. Ajouter ce type d'information permet ultérieurement (via un docker inspect sur l'image) de disposer de méta-information sur le contexte de création de l'image.

## 10. 5 LANCEMENT AUTOMATIQUE

La dernière étape consiste par exemple à programmer le déclenchement du *build*. Dans le cas d'un système CI, le facteur déclenchant est la modification du code (ou plus exactement le *merge* d'une modification avec la branche *develop*. Dans notre cas, nous nous contenteront de déclencher le *build* sur un simple commit.

### 10.5.1 Configuration de Jenkins et GitLab

#### Désactivation de la sécurité sur les API dans Jenkins

En préalable à l'exposition d'une API par Jenkins, il est nécessaire de désactiver la sécurité CSRF. Celle-ci n'est pas compatible avec de simples appels d'URL comme celui que GitLab est en mesure de générer.

Pour ce faire, cliquez sur « Administrer Jenkins » puis « Configurer la sécurité globale » et désélectionnez la case à cocher comme sur la figure 10.16.

Figure 10.16 – Désactivation de la protection CSRF



#### Activation du WebHook pour notre job

Allez dans la configuration du job que vous avez créé précédemment et cochez la case comme indiqué sur la figure 10.17. Vous devez aussi fournir un jeton d'authentification (une chaîne de caractères prise au hasard).

Figure 10.17 – Activation de l'API pour le déclenchement à distance du build



#### Création du WebHook dans GitLab

Connectez-vous à l'interface d'administration de GitLab puis au projet appex-front-end. Allez dans le menu « Settings » puis « Web Hooks » et ajoutez une configuration comme indiquée sur la figure 10.18.

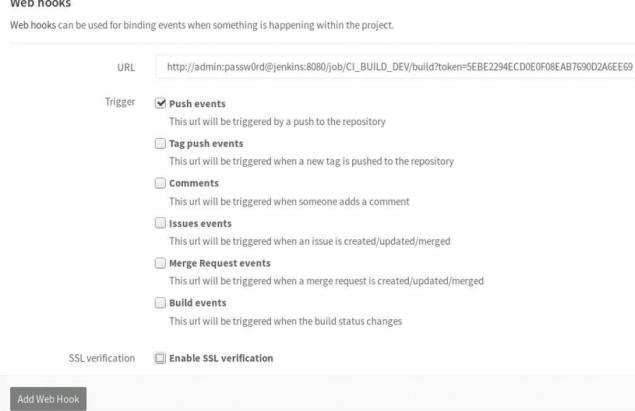
Notez dans l'URL quatre points importants :

- la présence du jeton d'authentification défini précédemment dans la configuration du job ;
- l'URL de l'hôte qui n'est pas localhost mais jenkins. En effet cet appel est initié à l'intérieur du conteneur GitLab et, grâce au réseau ci-network, Jenkins peut être atteint en utilisant le nom de son conteneur (utiliser localhost ne fonctionnerait pas) ;
- le port qui est 8080 et non 8081 car, encore une fois, l'appel s'effectue à l'intérieur du conteneur et non à l'extérieur (le port 8080 de Jenkins est exposé au niveau de l'hôte via le

port 8081) ;  
la présence du login et du mot de passe de l'utilisateur `admin`. Dans une implémentation réelle, nous créerions évidemment un utilisateur dédié associé à des droits restreints.  
L'URL sera donc de la forme :

```
| http://admin:password@jenkins:8080/job/CI_BUILD_DEV/build?token=123456789
```

**Figure 10.18 – Création du WebHook**



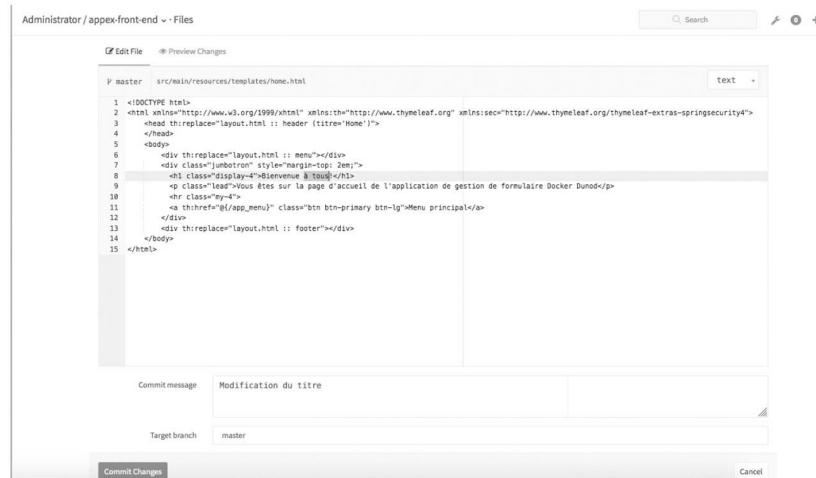
### 10.5.2 Exemple de modification et déploiement

Nous allons maintenant effectuer un test complet.

#### **Modification du code**

Pour nous simplifier la tâche, Gitlab nous propose d'éditer directement un fichier « en ligne ». Allez dans le projet, placez-vous dans le répertoire `appex-front-end` et éditez le fichier `src/main/resources/templates/home.html` comme indiqué sur la figure 10.19.

**Figure 10.19 – Modification et commit**



N'oubliez pas bien sûr de *commiter* votre changement.

Le code est désormais modifié et, quasi instantanément, le job Jenkins est déclenché.

#### **Déploiement**

L'image `appex-multi/front-end` a été mise à jour. Vous pouvez le constater à l'aide de la commande suivante :

```
| $ docker inspect --format '{{ json .ContainerConfig.Labels}}' appex-multi/front-end
```

```
| {"build":"jenkins-CI_BUILD_DEV-14"}
```

La valeur renvoyée doit correspondre à l'identifiant du *build* dans Jenkins.

Il faut maintenant redémarrer l'application avec Compose pour que celle-ci utilise la nouvelle image disponible. Utiliser le script `run.sh` se trouvant dans le répertoire `chapitre9/compose`.

```
./run.sh
Building initdb
...
appex-db is up-to-date
Creating compose_initdb_1 ... done
Recreating appex-back-end ... done
Recreating appex-front-end ... done
```

Connectez-vous à l'application front-end (`http://localhost:8080`) et constatez que votre modification a été déployée.

Dans un système de CI complet, le redémarrage de l'application pourrait être piloté automatiquement et être associé à une batterie de tests fonctionnels pour valider la non-régression de l'application.

## 10. 6 EXTENSIONS ET AMÉLIORATIONS

Le cas qui a été exposé ci-dessus est simple mais parfaitement fonctionnel pour de petites applications. Abordons quelques sujets complémentaires.

### 10.6.1 Gestion de version et images Docker

Notre application n'utilise pas de version. Évidemment, nous utilisons Git qui va garder la trace des modifications mais, dans la réalité, sur la base d'un *GitFlow* courant, chaque composant doit être associé à une *release version*. Celle-ci permet de tracer avec précision l'état du composant avant un déploiement.

En CI, il est possible d'utiliser des versions SNAPSHOTS qui gardent le même numéro de version mais dont le contenu évolue à chaque *build*. À l'inverse, en QA ou en production, il est nécessaire d'être plus précis.

L'une des questions courantes est celle de la politique de *tagging* des images selon le format Docker. Le nom des images est influencé par deux dimensions :

la version du code du composant que l'image encapsule (et du Dockerfile associé) gérée avec Git ;

le type d'environnement et la configuration qui lui sont associés (stockée sous la forme de fichiers dans le projet `appex-front-end-configuration` dans le cadre de notre exemple).

Notons de plus que la notion d'environnement peut être large. Si votre application cible plusieurs clients et comprend une configuration spécifique pour chacun d'eux, alors la configuration est encore plus complexe.

Deux grandes options sont possibles<sup>6</sup> :

On peut choisir de faire apparaître les deux dimensions dans le nommage de l'image, par exemple : `appex-multi/front-end:10.1_DEV` et `appex-multi/front-end:10.1_PROD`. Dans ce cas l'image est spécifique d'un environnement donné et la configuration est embarquée au moment du docker build (comme c'est le cas dans notre exemple).

On peut aussi choisir de ne produire qu'une seule et unique image pour tous les environnements et gérer la configuration au moment de la création du conteneur : docker run.

La première option aboutit à la création d'un plus grand nombre d'images mais est plus simple à gérer au moment du déploiement. L'image est complète et aucune dépendance tierce n'est susceptible d'introduire des problèmes de configuration. Par contre, chaque modification de configuration impose de recréer une nouvelle image.

La seconde option permet de configurer dynamiquement le conteneur au démarrage mais impose

l'usage d'une source de donnée externe (variable d'environnement, serveur de configuration, etc.). Le choix de l'une ou de l'autre dépend des contraintes d'environnement et du nombre de clients que vous devez gérer. S'il est important, la seconde option est inévitable.

### 10.6.2 Exécution de tests

Nous aurions pu exécuter des tests unitaires lors de la construction de notre application front-end. Si ces tests avaient été présents, Gradle les aurait déclenchés automatiquement. C'est d'ailleurs une bonne pratique.

Outre les tests unitaires, Docker apporte beaucoup à la gestion des tests. Outre les avantages évoqués plus haut avec l'exemple de notre image de *build Java*, il faut aussi noter la possibilité de modifier le contexte d'un conteneur en créant des *mocks*<sup>7</sup> à l'aide de volumes ou de réseaux. On peut par exemple disposer de diverses bases de tests et connecter un même conteneur à chacune de ces bases successivement. L'alias d'un conteneur (partageant le même réseau *bridge*) étant son nom, aucune reconfiguration du code n'est nécessaire.

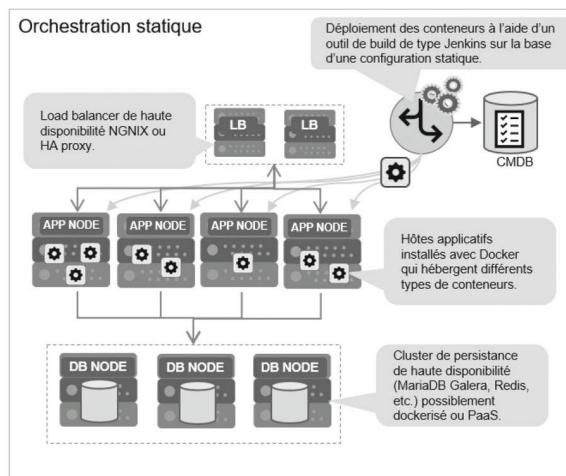
Docker permet donc de lancer simultanément des phases de test parallèles avec différents contextes d'exécution.

### 10.6.3 Déploiement multi-hôte

Dans notre exemple, nous ne travaillons qu'avec un seul hôte. Dans la réalité, la plupart des applications ont besoin de plusieurs machines.

La distribution des images est donc nécessaire via un registry Docker. La répartition de l'instanciation des conteneurs est la question suivante. La figure 10.20 montre un exemple classique d'architecture pour une application web conteneurisée.

Figure 10.20 – Architecture de déploiement statique



Une couche de répartiteurs de charge (*load balancer*) distribue la charge entre plusieurs hôtes hébergeant des conteneurs (certains front-end et d'autres back-end). Le *load balancer*, associé à un DNS privé, sert à mettre en relation les différents modules d'une application. Un cluster de base de données de haute disponibilité pour la persistance complète l'ensemble.

Reste la problématique du déploiement.

Si nous nous limitons à l'usage de Docker sans orchestrateur de type CaaS, alors un outil de *build* associé à une configuration statique (CMDB)<sup>8</sup> peut faire l'affaire.

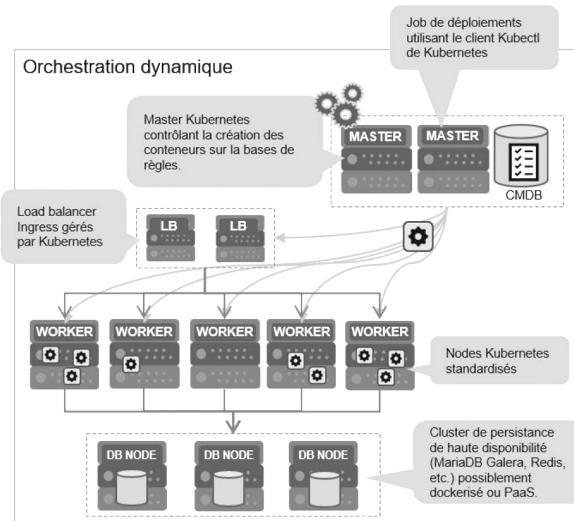
Cette CMDB peut être outillée de différentes manières, par exemple :

- un fichier de configuration lu par un job se connectant sur chaque machine pour « lancer » les conteneurs attendus ;

- un outil de gestion de configuration comme Chef ou Puppet.

Quoi qu'il en soit, la configuration est statique.

Figure 10.21 – Architecture à orchestration dynamique



Si une machine s'éteint, le *load balancer* pourra router l'ensemble du trafic vers d'autres instances mais il n'est pas possible de réorganiser dynamiquement la distribution.

Nous touchons là du doigt la limite de ce type d'architecture Docker. C'est là qu'entrent en jeu les solutions d'orchestration dynamiques (figure 10.21). Elles seront l'objet de la suite de cet ouvrage.



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.

- 
1. <https://jenkins.io/>
  2. <https://about.gitlab.com/>
  3. <https://github.com/jpetazzo/dind>
  4. <https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/>
  5. <https://search.maven.org/>
  6. Nous avons déjà évoqué ces questions dans le chapitre 8 (§ [8.4](#)).
  7. Un *mocks* est un simulacre d'une dépendance. Ce peut être une base de données, une API, etc. Le but est de faciliter les tests en créant un environnement que l'on va modifier à loisir pour étudier un maximum de cas de tests. Cet environnement est aussi systématiquement reproductible.
  8. CMDB (*Configuration Management Database*) : elle contient la métadescription de l'environnement. Initialement, le focus était sur les aspects physiques (machines, réseau et stockage), aujourd'hui l'aspect logiciel devient essentiel.

## CINQUIÈME PARTIE

### Orchestration de conteneurs

Dans les parties précédentes, nous nous sommes concentrés sur la mise en œuvre d'architectures mono-hôte. Nous avons étudié dans le détail toutes les fonctionnalités du démon Docker et des outils de création d'images.

Dans cette dernière partie, nous allons aborder le sujet de l'orchestration des conteneurs au travers de deux exemples :  
le chapitre 11 présente la solution Swarm (que nous avons décrite dans le chapitre 2). Il permet d'aborder le dernier modèle de réseau Docker : le modèle *overlay* ;  
le chapitre 12 reprend notre application exemple du chapitre 9 et montre comment Kubernetes permet, tout en s'appuyant sur Docker, d'apporter un lot de fonctionnalités innovantes et très puissantes.



# 11

## Docker Swarm : clustering avec Docker

### Objectif

L'objectif de ce chapitre est de découvrir la solution de clustering implémentée nativement dans l'écosystème de Docker : Swarm. Nous étudierons d'abord les différents composants, puis créerons localement sur notre poste de travail un petit cluster. Finalement, nous déployerons nos premiers services en externalisant leurs configurations et secrets avec Docker Compose.

L'issue de ce chapitre, vous saurez déployer et utiliser un cluster Docker et comprendrez les concepts sous-jacents à tout orchestrateur.



### Application exemple et code source

Le code source et les exemples de ce chapitre sont disponibles sur GitHub.

Veuillez vous reporter à la procédure d'installation du chapitre 3.

## 11. 1 DOCKER SWARM

### 11.1.1 Vue générale

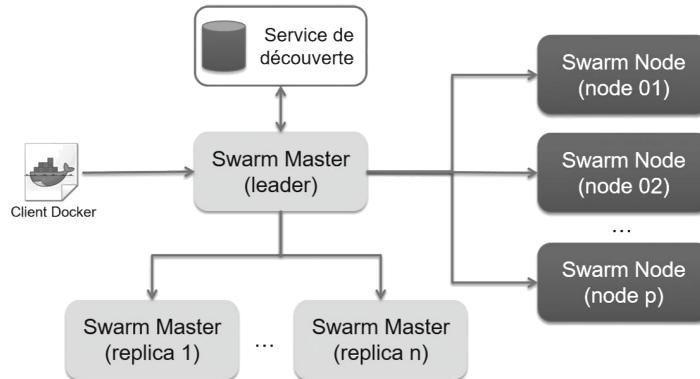
Nous n'avons pour l'instant utilisé Docker que sur un seul hôte, principalement notre poste de travail. Cependant, pour une utilisation productive de Docker, il est plus que probable que vous voudrez disposer de plusieurs noeuds, typiquement pour assurer une haute disponibilité de votre application. C'est là que Swarm entre en œuvre.

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 :

le client Docker : il sera utilisé aussi bien pour gérer nos conteneurs que notre cluster Swarm ;  
un service de découverte (*discovery service*) intégré qui permet de centraliser les informations de tous les hôtes (principalement la configuration réseau et l'état de chaque hôte) ;  
un noeud maître Swarm (*Swarm Master*) et optionnellement, dans le cas d'une architecture hautement disponible, un ou plusieurs réplicas de ce noeud maître qui, en tant que contrôleur, gère les différents noeuds du cluster et orchestre le déploiement des conteneurs. C'est celui-ci qui reçoit les commandes de déploiement et qui distribue ensuite les tâches sur les noeuds esclaves. Par défaut, les noeuds maîtres sont aussi des noeuds esclaves et peuvent donc faire tourner des conteneurs comme nous le verrons prochainement ;  
plusieurs noeuds Swarm (*Swarm Node*), qui sont simplement des hôtes sur lesquels nos conteneurs vont s'exécuter.

Figure 11.1 – Architecture générique d'un cluster Swarm



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.

### 11.1.2 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 l'utilisation de Vagrant, nous pouvons créer localement, sur notre poste de travail, un cluster minimal.

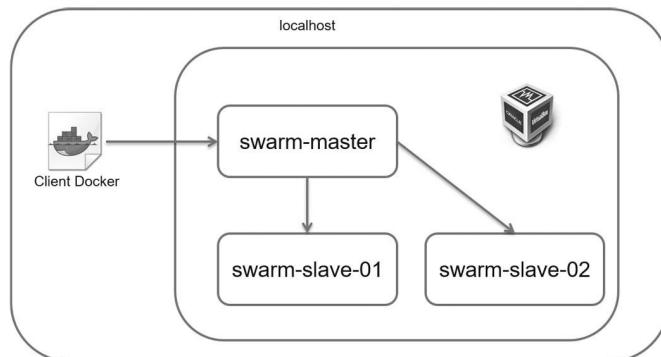
Ce cluster reposera sur trois hôtes (soit trois machines virtuelles VirtualBox) :

**swarm-manager** : notre nœud maître sera notre point d'entrée pour commander notre cluster (dans notre cas, il permettra également d'exécuter des conteneurs de la même manière que les nœuds esclaves) ;

**swarm-slave-01** et **swarm-slave-02** : nous aurons deux nœuds dans notre cluster qui feront tourner nos conteneurs et seront pilotés depuis notre *manager*.

Au final, nous aurons la configuration suivante :

Figure 11.2– Architecture de notre cluster Swarm



Le répertoire `chapitre11` contient un fichier `Vagrantfile` qui installe automatiquement trois machines virtuelles avec Docker. Pour les installer, utilisez la commande suivante, à lancer depuis le répertoire `chapitre11` :

```
vagrant up
...
$ vagrant global-status
  id          name      provider   state       directory
  -----
  2bc138e     swarm-master   virtualbox running   ...
  7aac3e6     swarm-slave-01  virtualbox running   ...
  7aac3e6     swarm-slave-02  virtualbox running   .../docker-swarm-vagrant
  7aac3e6     swarm-slave-03  virtualbox running   .../docker-swarm-
```

```
vagrant  
swarm-slave-02 virtualbox running .../docker-swarm-vagrant
```

Il ne nous reste plus qu'à connecter nos différents démons docker. Commençons par notre nœud maître et initialisons notre Swarm :

```
$ vagrant ssh swarm-master  
$ docker swarm init --advertise-addr=10.100.192.200  
Swarm initialized: current node (9y4w8bd48tjvgfgpymzhz7wk7) is now a manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join --token SWMTKN-1-5rqmrfsk7k7rmjw8z0gvxomzdm8orcjhnh4fkc0nof2780p4w7s-  
7yfhnh1ee5kko9atmyfjbrui3 10.100.192.200:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

La suite des instructions nous est fournie directement par Docker : nous devons simplement nous connecter sur nos nœuds esclaves et lancer la commande `docker swarm join` à l'aide du token fourni ci-dessus (celui-ci sera nécessairement différent pour vous).

```
$ vagrant ssh swarm-slave-01  
docker swarm join --token SWMTKN-1-5rqmrfsk7k7rmjw8z0gvxomzdm8orcjhnh4fkc0nof2780p4w7s-  
7yfhnh1ee5kko9atmyfjbrui3 10.100.192.200:2377  
This node joined a swarm as a worker.
```

Procédez de la même façon avec le nœud `swarm-slave-02`.

Nos nœuds communiquent via le port 2377 par défaut, qui doit donc être ouvert ainsi que le port 2376 sur lequel écoute le démon docker.

Si nous nous reconnectons sur notre nœud maître, nous pouvons facilement voir l'état de notre cluster Swarm :

```
$ docker node ls  
ID          HOSTNAME   STATUS    AVAILABILITY  MANAGER STATUS  ENGINE VERSION  
9y4w...     * swarm-    Ready     Active        Leader       18.06.0-ce  
           manager  
stqh...     swarm-slave-01  Ready     Active        18.06.0-ce  
2odn...     swarm-slave-02  Ready     Active        18.06.0-ce
```

Nous sommes donc prêts pour saisir tout l'intérêt d'un orchestrateur tel que Swarm.

## 11. 2 PREMIER SERVICE ET STACK

### 11.2.1 Service et montée en charge

Abordons cela étape par étape pour bien comprendre comment Swarm orchestre le déploiement de nos conteneurs.

Commençons par déployer trois conteneurs NGINX comme nous l'avons vu de nombreuses fois depuis le premier chapitre :

```
#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 ps --format "table {{.ID}} {{.Status}} {{.Ports}} {{.Names}}"  
CONTAINER ID      STATUS      PORTS      NAMES  
ad2c6d9e31bd     Up 7 seconds  0.0.0.0:32770->80/tcp  nginx-3
```

0a9a9210ff94	Up 10 seconds	0.0.0.0:32769->80/tcp	nginx-2
c1a8399d70c0	Up 25 seconds	0.0.0.0:32768->80/tcp	nginx-1

Nous constatons que nos trois conteneurs sont déployés sur l'hôte sur lequel nous sommes connectés. La question que nous pouvons nous poser est pourquoi certains n'ont-ils pas été déployés sur nos nœuds esclaves ? Supprimons maintenant notre conteneur nginx-1.

\$ docker rm -f nginx-1	\$ docker ps --format "table {{.ID}} {{.Status}} {{.Ports}} {{.Names}}"	CONTAINER ID	STATUS	PORTS	NAMES
		ad2c6d9e31bd	Up 45 seconds	0.0.0.0:32770->80/tcp	nginx-3
		0a9a9210ff94	Up 48 seconds	0.0.0.0:32769->80/tcp	nginx-2

Si nous effectuons cette commande plusieurs fois, nous voyons que notre conteneur nginx-1 n'est pas automatiquement redémarré. Pourtant, c'est le principe même d'un orchestrateur qui se veut déclaratif (nous « déclarons » l'état que nous voulons) plutôt qu'impératif (nous donnons les commandes nécessaires pour arriver à un état donné).

C'est ici que les services Swarm entrent en jeu. Un service peut être défini tout simplement comme la configuration du déploiement d'une même image (port à exposer, volume, nombre de réplicas requis...). Regardons la différence en installant maintenant nos conteneurs NGINX sous forme d'un service.

```
docker rm -f $(docker ps -a | grep nginx | awk '{print $1}')
docker service create --replicas=3 --name nginx-service nginx
fepe2qe4g0l51blc4ggzowbgf
overall progress: 3 out of 3 tasks
1/3: running [=====>]
2/3: running [=====>]
3/3: running [=====>]
verify: Service converged
docker service ls
ID          NAME      MODE      REPLICAS      IMAGE      PORTS
fepe2qe4g015  nginx-service  replicated  3/3        nginx:latest
```

La commande `docker service create` « soumet » un ensemble de tâches à notre maître Swarm, qui les répartit sur les différents nœuds. Vous aurez peut-être remarqué que le déploiement est plus long que d'habitude. C'est que Docker doit, pour chaque nœud sur lequel un 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 service ps nginx-service --format "table {{.ID}} {{.Name}} {{.Node}} {{.CurrentState}}"
ID          NAME      NODE      CURRENT      STATE
44c8jj4og0zr  nginx-service.1  swarm-slave-01  Running  33 minutes ago
22atzmmawm81  nginx-service.2  swarm-slave-02  Running  33 minutes ago
slckmkf1rx7u  nginx-service.3  swarm-manager  Running  35 minutes ago
```

Nous avons bien nos trois conteneurs, un sur chacun de nos nœuds. 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, Swarm essaiera de regrouper un maximum de conteneurs sur un minimum de nœuds. Pour cela, Swarm calcule un poids<sup>1</sup> pour chaque nœud en fonction d'un certain nombre de paramètres (CPU, RAM, nombre de conteneurs déjà présents, santé du démon Docker...) ;

**random** : comme son nom l'indique, la répartition sera effectuée de manière aléatoire sur tous les nœuds du cluster.

Que se passe-t-il si nous supprimons un de nos conteneurs (simulant un « crash ») ? Pour cela, un simple docker rm sera suffisant.

```
$ docker ps --filter "ancestor=nginx" --format "table {{.ID}} {{.Names}}"
CONTAINER ID          NAMES
1d5fc9017b77        nginx-service.3.slckmkf1rx7ugtyj9hhq2mh6n
docker rm -f 1d5fc9017b77 && while sleep 2; do docker service ls --format "{{.ID}} {{.Name}} {{.Replicas}}"; done
13cc5b9bbc05
fepe2qe4g015  nginx-service 2/3
fepe2qe4g015  nginx-service 2/3
fepe2qe4g015  nginx-service 3/3
fepe2qe4g015  nginx-service 3/3
```

Cette fois-ci, Swarm a détecté la disparition du conteneur et a automatiquement créé une nouvelle instance pour atteindre le nombre attendu de réplicas (qui est de trois).

Il est par ailleurs très facile de monter en charge en adaptant le nombre de réplicas.

```
docker service scale nginx-service=4
nginx-service scaled to 4
overall progress: 4 out of 4 tasks
1/4: running [=====>]
2/4: running [=====>]
3/4: running [=====>]
4/4: running [=====>]
verify: Service converged
docker service ps nginx-service --format "table {{.ID}} {{.Name}} {{.Node}} {{.DesiredState}}"
ID          NAME          NODE          DESIRED STATE
44c8jj4og0zr    nginx-service.1    swarm-slave-01    Running
22atzmmawm81    nginx-service.2    swarm-slave-02    Running
ss5n97k2qac9    nginx-service.3    swarm-manager    Running
slckmkf1rx7u \_  nginx-service.3    swarm-manager    Shutdown
jn0yhcvuj9im    nginx-service.4    swarm-manager    Running
```

Swarm a bien créé une nouvelle instance de notre image. Nous remarquons au passage que nous retrouvons l'instance que nous avons supprimée avec un statut *Shutdown* plus que significatif. Nous avons donc quatre conteneurs membres d'un même service. La prochaine étape est maintenant d'y accéder et, pour cela, découvrons tout d'abord comment fonctionne le réseau *overlay* de Swarm.

### 11.2.2 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 avons utilisé 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 les réseaux disponibles sur notre cluster :

```
$ docker network ls
NETWORK ID          NAME          DRIVER          SCOPE
b7e3d2c1aa0c        bridge        bridge        local
6ec44c096629        docker_gwbridge  bridge        local
f5c6964ed6e7        host          host          local
utqo3e3x0hvxa       ingress       overlay       swarm
b74d4464c614        none          null         local
```

Nous retrouvons les trois réseaux locaux *bridge*, *host* et *none* que nous avons déjà rencontrés. Nous

en listons cependant deux supplémentaires :

**docker\_gwbridge** : ce réseau de type *bridge* connecte les différents démons Docker de notre Swarm ;

**ingress** : ce réseau de type *overlay* gère le trafic de nos services. Par défaut, un service est automatiquement attaché à ce réseau si un autre réseau *overlay* n'est pas spécifié.

Il est conseillé, pour chaque service, de définir un réseau dédié afin de ségrégner et contrôler les connexions entre services. Créons donc un nouveau réseau et un service associé :

```
docker network create -d overlay nginx-overlay  
my2lixvxyy7hoaunn077vj6vy  
docker service create --name nginx-overlay --replicas=2 --network=nginx-overlay --publish  
published=8080,target=80 nginx
```

Nous voyons de nouvelles options dans notre commande :

--network=nginx-overlay : attache notre service à notre réseau *overlay* ;  
--publish published=8080,target=80: expose le port 80 de notre service (c.à.d. tous les ports 80 de tous nos conteneurs faisant partie de notre service) sur le port 8080 de notre cluster. Cette dernière option demande quelques explications.

Tous les nœuds de notre cluster participent à un *routing mesh*<sup>2</sup>. Ce réseau de routage permet à un service d'être accessible sur tous les nœuds de notre Swarm même si aucun conteneur de ce service ne tourne localement. Il est facile de le vérifier car nous n'avons déployé que deux répliques de notre service donc nous avons un nœud qui ne dispose pas d'un conteneur NGINX.

```
$ docker service ps nginx-overlay --format "table {{.ID}} {{.Name}} {{.Node}} {{.DesiredState}}"  
ID NAME NODE DESIRED STATE  
s46lavhf9ezt nginx-overlay.1 swarm-slave-01 Running  
x3hunnp3ycv5 nginx-overlay.2 swarm-manager Running
```

Nous n'avons donc pas de conteneur pour notre service sur le noeud *swarm-slave-02*. Essayons de joindre le service via l'IP de ce nœud.

```
curl http://$(docker node inspect swarm-slave-02 --format  
"{{.Status.Addr}}"):8080  
<!DOCTYPE html>  
<html>  
<head>  
<title>Welcome to nginx!</title>  
...
```

Notre service est cependant bien disponible sur le port 8080 via le réseau de maillage de Swarm.

Bien sûr, il n'est pas pratique de devoir accéder à un service à l'aide d'une adresse IP et d'un port. Dans la pratique, un répartiteur de charge type HAProxy<sup>3</sup> serait utilisé pour router le trafic sur nos différents hôtes et gérerait le domaine virtuel associé à l'IP publique d'entrée de notre site.

Une application n'est cependant que très rarement composée d'un unique service. Pour profiter de tous les bénéfices de Swarm, il est nécessaire d'utiliser un *stack* Docker.

### 11.2.3 Le *stack* Docker

Un *stack* docker n'est autre qu'un ensemble de services interdépendants. Ces derniers sont déployés et orchestrés ensemble par Swarm.

Le moyen le plus simple pour définir un *stack* est d'utiliser un fichier au format Compose.

La version du fichier Compose dépend de la version du Docker Engine<sup>4</sup>. Il est donc important de renseigner correctement cette valeur pour assurer une parfaite compatibilité. Dans notre cas, il nous faudra utiliser la version « 3.7 » qui est compatible avec le Docker 18.06.

Par exemple, nous allons créer une application qui nécessite un service NGINX et un service PHP-FPM<sup>5</sup>.

Les fichiers nécessaires sont disponibles dans le répertoire `simple-stack` du dépôt de ce chapitre. Il faudra les copier sur le nœud maître de notre Swarm avant de continuer.

```
version: "3.7"
services:
  web:
    image: nginx
    ports:
      - "8081:80"
    volumes:
      - "$PWD/nginx/nginx.conf:/etc/nginx/nginx.conf"
  deploy:
    restart_policy:
      condition: on-failure
    placement:
      constraints: [node.role == manager]
  depends_on:
    - php-fpm
  networks:
    - stack-net

  php-fpm:
    image: bitnami/php-fpm
    deploy:
      replicas: 2
    restart_policy:
      condition: on-failure
    networks:
      - stack-net

networks:
  stack-net:
```

Quelques points intéressants à noter :

Le service web sera exposé sur notre cluster sur le port 8001 et utilisera l'image officielle `nginx`.

Il montera sa configuration depuis un volume local à l'hôte. Autant le dire tout de suite, cette approche souffre de nombreux défauts :

nous devons contraindre notre service à tourner sur le nœud maître (`constraints: [node.role == manager]`) car, sinon, il n'aurait pas accès à son fichier de configuration. Nous devrions sinon le copier sur tous nos nœuds « au cas où » Swarm déciderait de l'instancier sur un autre nœud ;

une conséquence directe est que nous ne pouvons avoir qu'un réplica car, sinon, nous ne pourrions exposer le port 8001. En effet, il n'est pas possible de configurer plusieurs processus sur le même port de notre machine hôte.

Heureusement, il existe des solutions pour contourner ces défauts comme les `config` de Swarm, ce que nous aborderons dans la suite de ce chapitre.

L'instruction `depends_on` permet de définir un ordre de démarrage de nos services. En effet, dans ce cas précis, si nous omettions cette commande, le service web ne démarrerait pas car il ne pourrait joindre le processus `php-fpm`.

`networks` : nous créons notre réseau `overlay` auquel nous attacherons nos services.

Nous pouvons maintenant démarrer notre `stack` :

```
docker stack deploy -c docker-compose.yml stack-dunod
Creating network stack-dunod_stack-net
Creating service stack-dunod_php-fpm
Creating service stack-dunod_web
docker stack ps stack-dunod --format "table {{.ID}} {{.Name}} {{.Node}} {{.DesiredState}}"
```

ID	NAME	NODE	DESIRED STATE
w9g1y44qf84j	stack-dunod_web.1	swarm-manager	Running
bdwafqe2064u	stack-dunod_php-fpm.1	swarm-slave-01	Running
3dvacf7j8ejj	stack-dunod_php-fpm.2	swarm-manager	Running

Puisque notre *stack* est opérationnel, il est temps de voir comment Swarm gère la configuration des services.

## 11. 3 GESTIONS DES CONFIGURATIONS ET DES SECRETS

Toute application a des paramètres de configuration, ne serait-ce par exemple que le chemin d'accès une base de données. Ces paramètres varient en fonction de l'environnement (Production, Recette ou Développement par exemple).

Nous nous sommes attachés, dans le chapitre 8, à décrire les solutions possibles avec le moteur Docker. Nous avons vu dans les chapitres 9 et 10 quelles solutions pratiques pouvaient être mises en œuvre avec une architecture Docker statique.

Voyons maintenant comment Swarm offre une solution distribuée et sécurisée de bout en bout pour la gestion de ses secrets et des configurations.

### 11.3.1 Gestion de configuration

Les *config* de Swarm permettent de stocker des informations de configuration, typiquement des fichiers.

Les fichiers nécessaires sont disponibles dans le répertoire *stack-avec-config* du dépôt de ce chapitre.

Créons une configuration avec le fichier *http.conf* de notre service web :

```
#Depuis le repertoire nginx de notre Swarm Master
$ docker config create nginx-config http.conf
```

Une *config* est facilement lisible avec la commande suivante :

```
$ docker config inspect --pretty nginx-config
ID: mld76o42u2ggw0cfes72zxhi0
Name: nginx-config
Created at: 2018-09-15 15:56:01.431957611
+0000 utc
Updated at: 2018-09-15 15:56:01.431957611
+0000 utc
Data:
server {
  listen 0.0.0.0:80;
  server_name myapp.com;
...
}
```

Remplaçons ensuite le volume que nous utilisions par la configuration. La définition de notre service devient alors la suivante :

```
web:
  image: nginx
  ports:
    - "8081:80"
  configs:
    - nginx_config
    target: /etc/nginx/conf.d/http.conf
    mode: 0440
  deploy:
```

```

replicas: 3
restart_policy:
  condition: on-failure
depends_on:
  - php-fpm
networks:
  - stack-net
...
configs:
  nginx_config:
    external: true

```

Nous devons rajouter un bloc `configs` pour déclarer notre configuration. Nous en profitons pour enlever les contraintes de déploiement et rajouter une directive `replicas`. Nous pouvons maintenant mettre à jour notre service en utilisant une nouvelle version de notre fichier Compose :

```

docker stack deploy -c docker-compose-config.yml stack-dunod Updating
service stack-dunod_web (id: jqe3ljkq1x4qwzlx56x4wa6t) Updating service
stack-dunod_php-fpm (id: s8dyw3s4edx4p7u5i6hf7ezsa)

```

La configuration est accessible depuis le conteneur par l'intermédiaire d'un fichier placé sur le système de fichiers du conteneur :

```

$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
STATUS              PORTS              NAMES
15458b57c02c        nginx:latest       "nginx -g 'daemon of..."   7 minutes ago
Up 7 minutes        80/tcp             stack-dunod_web.3.71690d9hfeghywxkaeqgdc1yf
docker exec -it stack-dunod_web.3.71690d9hfeghywxkaeqgdc1yf /bin/bash
root@15458b57c02c:/# ls
bin boot dev etc home lib lib64 media mnt nginx_config opt proc root run sbin srv sys tmp usr var
root@15458b57c02c:/# cat /etc/nginx/conf.d/http.conf
server {
  listen 0.0.0.0:80;
  server_name myapp.com;
...

```

Nous retrouvons, ci-dessus, notre configuration sous la forme d'un fichier. L'avantage évident des `config` est de ne plus avoir à utiliser des volumes montés qui ne sont évidemment pas partagés entre nos hôtes. Mais comment faire si les données que nous voulons rendre disponibles à nos applications sont sensibles ? Voyons cela avec les `secrets` de Swarm.

### 11.3.2 Gestion de secrets

L'un des challenges communs pour une application est de gérer d'une manière sécurisée les credentials » :

- mots de passe ;
- clé SSH ;
- certificats d'authentification ;
- etc.

Outre le besoin de s'assurer que seuls les utilisateurs autorisés aient accès à ces informations, il faut aussi s'assurer de la possibilité de les modifier, si possible sans avoir à reconstruire l'application.

La différence principale entre les secrets et les `config` Swarm réside dans le fait que l'accès au secret n'est possible que depuis un service pour lequel il a été explicitement autorisé. Il n'est pas possible d'inspecter le contenu d'un secret depuis par exemple la ligne de commande.

Nous allons créer deux secrets qui contiendront la clé et le certificat SSL pour notre application, ce qui nous permettra de l'exposer en HTTPS.

Les fichiers nécessaires sont disponibles dans le répertoire `stack-avec-config-et-secret` du dépôt de ce chapitre

Nous avons déjà généré les fichiers nécessaires. Il ne nous reste qu'à nous en servir pour créer nos secrets :

```
| docker secret create site.key nginx/site.key  
| docker secret create site.crt nginx/site.crt  
| docker config create nginx-config-ssl nginx/ssl.conf
```

Notre fichier Compose est adapté en conséquence :

```
web:  
  image: nginx  
  ports:  
    - "8081:443"  
  configs:  
    - source: nginx-config-ssl  
      target: /etc/nginx/conf.d/ssl.conf  
      mode: 0440  
  secrets:  
    - site.crt  
    - site.key  
  ...  
  
secrets:  
  site.crt:  
    external: true  
  site.key:  
    external: true
```

Nous exposons maintenant notre application sur le port 443 et avons adapté notre configuration NGINX.

Redéployons notre *stack* pour le mettre à jour avec cette nouvelle configuration :

```
| $ docker stack deploy -c docker-compose-config-secret.yml stack-dunod
```

De manière identique aux *config*, les secrets sont visibles en se connectant dans un des conteneurs de notre service. Les secrets sont décryptés et insérés dans les conteneurs sous la forme de fichiers localisés dans le répertoire /run/secrets.

## 11. 4 L'AVENIR DE DOCKER SWARM

Nous terminons ce chapitre consacré à Swarm en évoquant la question du positionnement de cette solution par rapport à Kubernetes (que nous allons présenter dans le chapitre 12).

Aujourd'hui, pour ce qui est de la question de la standardisation, il ne fait aucun doute que Kubernetes est la solution de référence pour l'orchestration de conteneurs. À tel point que Docker a maintenant adopté Kubernetes aussi bien dans sa solution Docker Entreprise que dans les clients Windows et Mac.

Doit-on pour autant en conclure que Swarm n'a plus d'avenir ?

La réponse n'est pas aussi simple qu'il pourrait sembler.

Kubernetes a l'avantage d'être une solution extrêmement riche, ce qui en fait aussi une solution complexe. Installer un cluster Kubernetes n'est pas une tâche aisée. Le mettre à jour est très complexe. C'est la raison pour laquelle la plupart des cas d'utilisation de Kubernetes sont liés à l'usage de solutions *cloud* (Google, Microsoft, Amazon) CaaS (*Container as a Service*).

Swarm est sans aucun doute moins riche en termes de fonctionnalités mais aussi beaucoup plus simple à déployer (vous avez pu le constater ci-dessus) car il est complètement intégré au client Docker.

La réponse à la question que nous avons posée plus haut est probablement entre les mains des dirigeants de Docker Inc. S'ils essaient de positionner Swarm frontalement en face de Kubernetes, il est peu probable que la solution résiste longtemps. Par contre, une entreprise qui souhaiterait mettre en place, à moindre coût, une solution d'orchestration de conteneurs de moyenne complexité saurait certainement bénéficier de Swarm. L'inconnue de l'équation est alors la tarification des licences Swarm Entreprise ou l'appétence de cette entreprise à utiliser la version open source de la solution.



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 et déployer des services à l'aide des stacks. Nous nous sommes enfin attachés à montrer comment gérer les configurations de nos applications à l'aide des secrets et *config*.

Il est temps d'étudier la principale solution d'orchestration de conteneurs du marché : Kubernetes.

- 
1. [https://github.com/docker/swarm/blob/master/scheduler/strategy/weighted\\_node.go](https://github.com/docker/swarm/blob/master/scheduler/strategy/weighted_node.go)
  2. <https://docs.docker.com/engine/swarm/ingress/>
  3. <http://www.haproxy.org/>
  4. <https://docs.docker.com/compose/compose-file>
  5. PHP Fast Process Manager est une implémentation FastCGI PHP spécialement adaptée aux sites à forte charge. PHP-FPM se présente sous la forme d'un processus auquel se connecte le serveur web (en l'occurrence NGINX).

## 12

# Kubernetes : clustering avancé

### Objectif

L'objectif de ce chapitre est de présenter la solution Kubernetes. Dans un premier temps, nous utiliserons une application PHP très simple pour en démontrer les principales fonctionnalités. Ensuite nous configurerons un déploiement de notre application exemple déjà utilisée dans les chapitres 8, 9 et 10. Nous avons présenté Kubernetes dans le chapitre 2 mais sous un angle architectural. Ce chapitre vise à l'aborder sous un angle pratique. Le propos n'est pas de décrire exhaustivement Kubernetes, un ouvrage complet serait nécessaire, mais d'en présenter les fonctionnalités majeures. Il s'agit aussi de montrer comment Kubernetes complète Docker avec des fonctionnalités permettant de produire des architectures dynamiques.



### Application exemple et code source

Le code source et les exemples de ce chapitre sont disponibles sur GitHub.

Veuillez vous reporter à la procédure d'installation du chapitre 3.

**Attention, veillez à bien exécuter la procédure d'installation complémentaire décrite dans le [paragraphe 12.2](#).**

## 12. 1 ENVIRONNEMENT

Nous aurions pu pour ce chapitre utiliser un cluster local. Il existe des outils comme `minikube`<sup>1</sup> ou le client Docker pour Windows qui permettent de monter un cluster à un seul nœud. Le souci de ces solutions est qu'elles ne sont pas représentatives pour démontrer les fonctionnalités de Kubernetes. Expliquer ici la procédure pour mettre en œuvre une instance complète de Kubernetes (avec l'outil `kubeadm`<sup>2</sup> par exemple) nécessiterait un chapitre complet. Comme nous l'avons évoqué dans le chapitre précédent, monter un cluster Kubernetes est une tâche complexe. Le rendre opérationnel pour de la production l'est encore plus.

Nous avons donc choisi, comme la plupart des utilisateurs de Kubernetes actuels, de recourir à une solution « *as a service* », en l'occurrence la solution de Microsoft : Azure Kubernetes Services<sup>3</sup>.

Vous pouvez évidemment utiliser un autre produit (Google, Amazon ou OpenShift). À l'exception de la configuration des volumes (dont nous verrons qu'elle est spécifique de l'infrastructure technique), l'ensemble des commandes présentées devraient fonctionner de manière identique si la version de Kubernetes utilisée est supérieure à la version 1.10 (pour notre part, nous utilisons une version 1.11.2).

Voici la configuration installée pour nos exemples :

- un cluster managé AKS (nous n'avons pas accès aux nœuds master) ;
- quatre *nodes* Kubernetes de 2vCPU et 7GB RAM chacun.



### Une instance Kubernetes dans le cloud

À titre indicatif, une instance AKS de la taille de celle que nous utilisons coûtera pour 48 heures environ 30 \$. En l'éteignant

quand vous ne vous en servez pas, pour le même prix, elle pourra durer une semaine. Les offres de Google ou d'Amazon se situent dans la même gamme de prix.

Vous aurez aussi besoin d'un compte Docker Hub, dont le nom est à configurer dans le fichier `chapitre12/configuration.env`. La création du compte est gratuite pour les images publiques.

Kubernetes peut bien sûr utiliser d'autres registries que le Docker Hub mais celui-ci présente l'intérêt d'être gratuit et d'une configuration simple.

## 12. 2 PRISE EN MAIN

Avant de débuter le déploiement de notre application dans le cluster, apprenons à connaître Kubernetes.

### 12.2.1 IMPORTANT : installation des exemples

Les exemples pour ce chapitre se trouvent dans le répertoire `chapitre12`.



#### Configuration des scripts du chapitre

Avant de les utiliser, vous devez lancer le script `setup.sh`, qui se trouve à la racine du répertoire, après avoir rempli le fichier `configuration.env`. Si le script n'est pas exécutable, vous pouvez, au préalable, lancer la commande `chmod u+x setup.sh`.

Le script `setup.sh` utilise une variante de notre script « replacer » utilisé dans les chapitres 9 et 10. Il va configurer les scripts et les fichiers YAML avec les paramètres qui vous sont spécifiques (par exemple le nom de votre compte Docker Hub).

Dans certaines explications ci-dessous, vous trouverez des variables comme  `${DOCKER_HUB_ACCOUNT}` par exemple. Dans vos scripts, sur votre ordinateur, ces variables seront remplacées par les valeurs que vous aurez spécifiées dans `configuration.env`.

### 12.2.2 Kubectl : la CLI de Kubernetes

`kubectl4` est la ligne de commande de Kubernetes. C'est l'interface de prédilection pour interagir avec le (ou les) nœud(s) master(s) du cluster. Ce client dialogue directement avec les API REST exposées par le cluster.

Commençons par une prise en main rapide :

```
$ kubectl version
Client      Version:        version.Info{Major:"1",      Minor:"11",      GitVersion:"v1.11.2",
GitCommit:"bb9ffb1654d4a729bb4cec18ff088eacc153c239",  GitTreeState:"clean",  BuildDate:"2018-08-
07T23:17:28Z", GoVersion:"go1.10.3", Compiler:"gc", Platform:"windows/amd64"}
Server      Version:        version.Info{Major:"1",      Minor:"11",      GitVersion:"v1.11.2",
GitCommit:"bb9ffb1654d4a729bb4cec18ff088eacc153c239",  GitTreeState:"clean",  BuildDate:"2018-08-
07T23:08:19Z", GoVersion:"go1.10.3", Compiler:"gc", Platform:"linux/amd64"}
```

La commande précédente affiche la version du client et du serveur. Comme pour Docker, il est possible d'utiliser un client et un serveur avec des versions différentes au risque de voir certaines fonctionnalités indisponibles.

Voyons maintenant les principaux composants impliqués dans la gestion du cluster :

```
$ kubectl get componentstatuses
NAME            STATUS   MESSAGE           ERROR
controller-manager  Healthy
scheduler        Healthy
etcd-0          Healthy
                  {"health": "true"}
```

Notez que dans le cas d'AKS un bug dans certaines versions aboutit, pour le contrôleur et le *scheduler*, à l'affichage *Unhealthy*. Et pourtant, il n'en est rien : c'est un bug !

Quelques explications :

le **contrôleur** : est en charge de scruter l'état de santé des services déployés dans le cluster ;

le **scheduler** : est en charge du démarrage des conteneurs, qui sont groupés en *pods* (comme nous l'avons vu dans le chapitre 2) ;

**etcd** : est un stockage répliqué de haute disponibilité qui assure la persistance de la configuration du cluster.

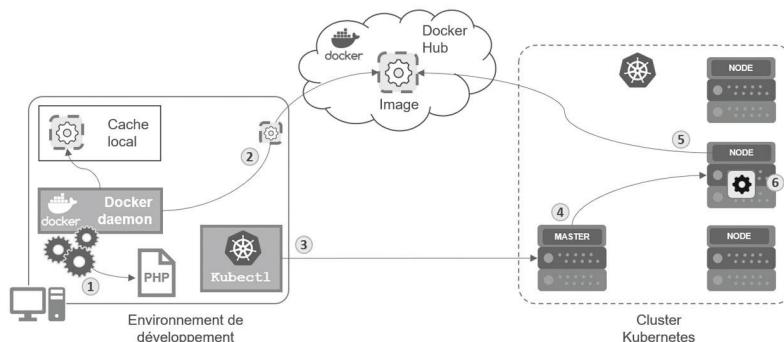
La prochaine commande `kubectl get nodes` nous permet de connaître le nombre de nœuds (*nodes*) du cluster. Il s'agit généralement de machines virtuelles Linux sur lesquelles sont installés Docker et les composants de Kubernetes (l'agent Kubelet). Comme indiqué dans notre introduction, ils sont ici au nombre de 4.

\$ kubectl get nodes				
NAME	STATUS	ROLES	AGE	VERSION
aks-agentpool-42554519-0	Ready	agent	3h	v1.11.2
aks-agentpool-42554519-1	Ready	agent	3h	v1.11.2
aks-agentpool-42554519-2	Ready	agent	3h	v1.11.2
aks-agentpool-42554519-3	Ready	agent	3h	v1.11.2

### 12.2.3 Un exemple simple

La figure 12.1 décrit la séquence d'opérations qui aboutira au déploiement d'un *pod* qui, dans ce cas, ne comprend qu'un seul conteneur encapsulant une application très simple que nous allons réutiliser dans les paragraphes suivants.

Figure 12.1 – Déploiement d'un *pod* Kubernetes



Voici une description des premières étapes et des éléments de code associés (instructions présentes dans le fichier `build.sh`) :

La première étape est un *build* Docker utilisant le fichier source `index.php` et le `Dockerfile` présent dans le répertoire `chapitre12/quelleheure/prise-en-main` :

```
| docker build -t ${DOCKER_HUB_ACCOUNT}/quelleheure .
```

La seconde étape est un *push* de l'image précédemment créée vers le Docker Hub :

```
$ docker login -u ${DOCKER_HUB_ACCOUNT}
docker push ${DOCKER_HUB_ACCOUNT}/quelleheure
docker logout
```

Exécutez ces étapes tout simplement en vous plaçant dans le répertoire `quelleheure/prise-en-main` et lancez le script `build.sh`.

Maintenant, nous allons déployer notre image dans un *pod* Kubernetes en utilisant `kubectl` et un

descripteur YAML :

```
| $ kubectl apply -f quelleheure-prise-en-main.yaml
```

Ceci correspond à l'étape 3 sur la figure 12.1. Regardons maintenant ce que cette instruction déclenche :

`kubectl` ordonne au master Kubernetes de déployer une instance de notre *pod* basée sur l'image qui est indiquée dans notre descripteur.

Le master sélectionne un *node* et lui commande (en interagissant avec l'agent Kubelet) de démarrer le *pod*.

L'agent Kubelet télécharge l'image (`docker pull`) depuis le Docker Hub et démarre le *pod* (et son unique conteneur).

Voyons maintenant si notre *pod* est démarré :

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
quelleheure	0/1	ContainerCreating	0	9s

Au premier lancement, le *pod* peut mettre un peu de temps à démarrer en raison du temps de chargement de l'image depuis le Docker Hub. Après quelques secondes, le *pod* est en statut ContainerCreating.

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
quelleheure	1/1	Running	0	59s

Après quelque temps, le conteneur est démarré et le statut devient Running.

Il est important de se souvenir que toutes les opérations initiées par `kubectl` sont asynchrones. Ce n'est pas parce que le client « rend la main » que l'opération demandée est terminée.

Aucun port n'étant directement ouvert à l'extérieur du cluster, nous ne pouvons pas encore accéder à notre *pod*. Nous verrons que la chose nécessite un peu de travail supplémentaire.

Cependant, nous pouvons demander à `kubectl` de nous ouvrir un tunnel entre le cluster et notre ordinateur pour visualiser ce que le *pod* expose sur le port 80.

```
kubectl port-forward quelleheure 8090:80
```

```
Forwarding from 127.0.0.1:8090 -> 80
```

```
Forwarding from [::1]:8090 -> 80
```

Nous pouvons alors ouvrir un navigateur sur le port 8090 (voir figure 12.2).

Le conteneur est actif et son hôte porte le nom « quelleheure » (ce qui avec Docker signifie qu'il s'agit du nom du conteneur).

Figure 12.2 – Notre premier *pod*



### Quelle heure est-il ?

Hello il est : 01:17:41pm  
Il semble que je sois installé sur un hôte nommé : quelleheure

On peut aussi visualiser ses logs :

```
$ kubectl logs -f quelleheure
```

```
[Sun Aug 19 13:14:36.192078 2018] [mpm_prefork:notice] [pid 1] AH00163: Apache/2.4.25 (Debian)  
PHP/7.2.8 configured -- resuming normal operations  
[Sun Aug 19 13:14:36.192143 2018] [core:notice] [pid 1] AH00094: Command line: 'apache2 -D  
FOREGROUND'  
...
```

## 12.2.4 Un descripteur Kubernetes

Kubernetes s'appuie sur des descripteurs au format YAML ou JSON. Néanmoins, les bonnes pratiques<sup>5</sup> recommandent l'usage de YAML. L'édition de fichiers YAML, pour qui ne l'a jamais pratiqué, étant un exercice parfois étonnant, nous recommandons l'usage d'un éditeur<sup>6</sup>.

Voici le descripteur que nous avons utilisé dans l'exemple précédent :

```
apiVersion: v1
kind: Pod
metadata:
  name: quelleheure
spec:
  containers:
    - image: ${DOCKER_HUB_ACCOUNT}/quelleheure:v1
      name: quelleheure
      imagePullPolicy:
        Always
      ports:
        - containerPort: 80
          name: http
          protocol: TCP
```

Il s'agit d'un *pod* dont nous voyons que les spécifications (*spec*) ne comprennent qu'un unique conteneur écoutant sur le port 80.

Notez l'instruction *imagePullPolicy*: *Always*. Celle-ci est importante. Par défaut, Kubernetes ne va pas chercher à re-télécharger une image si elle est déjà dans le cache du *node*. Le principe, plutôt bon, est que si une image change, son identifiant doit aussi avoir changé.

Ici nous « forçons » le re-téléchargement pour tenir compte d'éventuels changements et ainsi simplifier les exemples.

Voici la commande à utiliser pour détruire un *pod* :

```
kubectl delete pod quelleheure pod
"quelleheure" deleted
```

## 12. 3 DÉCOUVERTE DES FONCTIONNALITÉS

ce stade, notre petite application n'a pas d'état, une simple page affichant l'heure courante du système. Nous allons maintenant progressivement étudier l'ajout de nouvelles fonctionnalités. Nous nous servirons ensuite de ces connaissances pour déployer l'application « application-exemple » du chapitre 8 dans notre cluster.

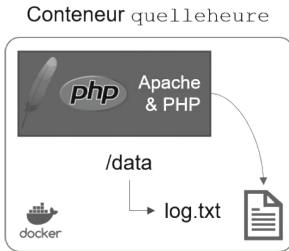
### 12.3.1 Ajout d'un état : écriture sur un volume host

Nous allons ajouter à notre application l'écriture d'un log d'accès. Celui-ci enregistrera chaque accès à l'application sous la forme d'une ligne dans un fichier texte lui-même affiché à l'utilisateur.

#### *Lancement du pod et description*

Le code de cette version modifiée de notre application se trouve dans le fichier chapitre12/quelleheure/log-sans-volume. Placez-vous dans le répertoire précédent et lancez le script build.sh. Celui-ci pousse une nouvelle version de l'image cette fois nommée : \${DOCKER\_HUB\_ACCOUNT}/quelleheure:v2.

Figure 12.3 – Écrire dans un log



La modification consiste dans l'ajout de code pour écrire dans un fichier `log.txt` placé dans un répertoire `/data` à l'intérieur du conteneur. Déployons notre nouveau *pod*.

```
| $ kubectl apply -f quelleheure-log-sans-volume.yaml
```

Voyons maintenant si notre *pod* s'est déployé correctement à l'aide de l'instruction `describe` :

```
$ kubectl describe pods quelleheure
Name:           quelleheure
...
Events:
Type    Reason     Age   From            Message
----    -----     ---   ----            -----
Normal  Scheduled  39s   default-scheduler  Successfully assigned default/quelleheure to aks-agentpool-42554519-2
Normal  Pulling    38s   kubelet, aks-agentpool-42554519-2  "dunoddocker/quelleheure:v2"
Normal  Pulled     3s    kubelet, aks-agentpool-42554519-2  Successfully pulled image "dunoddocker /quelleheure:v2"
```

Nous utiliserons fréquemment cette instruction qui nous donne des informations sur l'état du *pod* et plus spécifiquement pour la dernière section `Events`. Celle-ci montre chaque étape du déploiement : on peut y voir ici le téléchargement de l'image dont nous avons parlé précédemment.

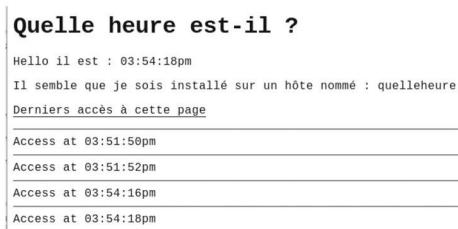
### **Test et... problème**

Comme précédemment, nous ouvrons un tunnel pour visualiser l'application.

```
| $ kubectl port-forward quelleheure 8090:80
```

Tout semble fonctionner normalement (figure 12.4). Les logs s'accumulent au fur et à mesure des accès (rafraîchissez simplement la page pour créer quelques logs).

Figure 12.4 – Affichage des logs



Néanmoins un redémarrage montre que les logs ne sont pas conservés :

```
| kubectl delete pods quelleheure pod
"quelleheure" deleted
kubectl apply -f quelleheure-log-sans-volume.yaml
pod/quelleheure created
```

Le descripteur nous montre que nous avons instancié le *pod* mais qu'aucun moyen de persistance (volume) n'a été défini. Le log est écrit dans le conteneur et est détruit quand le *pod* est redémarré.

### Ajoutons un volume

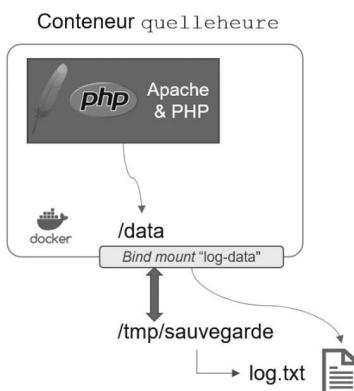
Placez-vous dans le répertoire chapitre12/quelleheure/log-volume-host et lancez le *build* d'une nouvelle image avec le script build.sh.

Nous allons créer un *bind mount* sur l'hôte pour persister le contenu du répertoire /data. Notez que nous devons modifier notre Dockerfile pour tenir compte des problématiques de droits :

```
RUN echo "chown -R 'www-data':'www-data' /data/ /var/www/html/ && apache2-foreground" >>
/usr/local/bin/start_server.sh
RUN chmod u+x /usr/local/bin/start_server.sh
ENTRYPOINT start_server.sh
```

Le répertoire monté étant la propriété de l'utilisateur root, le serveur Apache exécutant PHP ne serait pas en mesure d'y écrire. Nous changeons donc l'ENTRYPOINT du Dockerfile pour nous assurer que les droits sur le répertoire seront modifiés avant le démarrage du serveur web.

Figure 12.5 – Écrire dans un log avec *bind mount*



En regardant le fichier descripteur, outre le changement d'image, nous notons une nouvelle section :

```
volumes:
  - name: "log-data"
    hostPath:
      path: "/tmp/sauvegarde"
containers:
  - image: dunoddocker/quelleheure:v3
    name: quelleheure
    volumeMounts:
      - mountPath: "/data"
        name: "log-data"
```

Il s'agit de la déclaration du volume log-data (qui se trouve être associé au chemin /tmp/sauvegarde sur l'hôte) et son usage au sein du conteneur associé au chemin /data que nous voulons rendre persistant.

Lançons notre pod « nouvelle formule » :

```
$ kubectl apply -f quelleheure-log-volume-host.yaml
The Pod "quelleheure" is invalid: spec: Forbidden: pod updates may not change fields other than
'spec.containers[*].image',
'spec.initContainers[*].image', 'spec.activeDeadlineSeconds'
or 'spec.tolerations' (only additions to existing tolerations)
```

L'erreur provient du fait que nous tentons de lancer un *pod* du même nom qu'un *pod* actif (quelleheure) mais utilisant une image différente : quelleheure:v3. Kubernetes ne l'autorise pas. Pour n'utiliser qu'une seule instruction (à la place d'une séquence delete / apply), on peut utiliser la commande replace :

```
kubectl replace --force -f quelleheure-log-volume-host.yaml
pod "quelleheure" deleted
pod/quelleheure replaced
```

```
$ kubectl get pods -o wide
NAME        READY   STATUS    RESTARTS   AGE      IP          NODE
quelleheure  1/1     Running   0          33s     172.20.3.46  aks-
agentpool-37386073-1
```

Comme précédemment, nous ouvrons un tunnel pour visualiser l'application :

```
| $ kubectl port-forward quelleheure 8090:80
```

En apparence le fonctionnement est identique à notre précédente version. Et pourtant si nous relançons le déploiement plusieurs fois nous notons des comportements différents :

si le *pod* est lancé sur le même hôte (aks-agentpool-37386073-1) alors le log continue de s'accumuler ;

si le *pod* est lancé sur un autre hôte, le log repart à zéro.

Rien d'étonnant, nous avons utilisé un *bind mount*. Nous avons donc des fichiers sur chaque hôte.

Selon l'hôte sur lequel le *pod* est activé, nous ne voyons pas le même fichier.

### 12.3.2 Influencer le *scheduler* Kubernetes : les labels

La première solution à notre problème est de s'assurer que notre *pod* soit créé systématiquement sur le même *node*, autrement dit sur le même hôte.

S'il n'est pas possible de spécifier explicitement le *node* sur lequel nous souhaitons déployer un *pod*, nous pouvons néanmoins inciter le contrôleur à choisir celui que nous voulons.

#### **Définition d'un label**

Pour commencer, nous allons marquer un de nos *nodes* à l'aide d'un *label*. Le principe est exactement le même que les *labels* Swarm :

```
kubectl label nodes aks-agentpool-42554519-0 log=ok
node/aks-agentpool-42554519-0 labeled
kubectl get nodes --show-labels
NAME        STATUS   ROLES      AGE      VERSION   LABELS
aks-agentpool-  Ready    agent      2d       v1.11.2   agentpool=
42554519-0
agentpool,beta.kubernetes.io/arch=amd64,beta.kubernetes.
io/instance-type=Standard_DS2_v2,beta.kubernetes.io/os=linux,failure-
domain.beta.kubernetes.io/region=westeurope,failure-
domain.beta.kubernetes.io/zone=0,kubernetes.azure.com/cluster=MC_AKSTest_akstest_westeurope,kubern
agentpool-42554519-
0,kubernetes.io/role=agent,log=ok,storageprofile=managed,storagetier=Premium_LRS
aks-agentpool-  Ready    agent      15h      v1.11.2   agentpool=
42554519-1
agentpool,beta.kubernetes.io/arch=amd64,beta.kubernetes.io/instance-
type=Standard_DS2_v2,beta.kubernetes.io/os=linux,failure-
domain.beta.kubernetes.io/region=westeurope,failure-
domain.beta.kubernetes.io/zone=1,kubernetes.azure.com/cluster=MC_AKSTest_akstest_westeurope,kubern
agentpool-42554519-1,kubernetes.io/role=agent,storageprofile=managed,storagetier=Premium_LRS ...
```

Notez que le modificateur `--show-labels` nous montre que les *nodes* sont associés à un nombre conséquent de *labels*. Par exemple, Microsoft indique la région (westeurope) et le type de VM (Standard\_DS2\_v2).

#### **Définition des exigences pour un pod**

Placez-vous maintenant dans le répertoire chapitre12/quelleheure/labels et ouvrez le fichier `quelleheure-labels.yaml`. L'image utilisée est la même (pas de raison de la changer) mais une nouvelle instruction a fait son apparition :

```
| nodeSelector:
```

```
| log: ok
```

Celle-ci indique à Kubernetes que nous souhaitons que ce *pod* soit instancié sur un *node* marqué par le *label* `log=ok`. Nous pouvons en vérifier l'effet :

```
kubectl replace --force -f quelleheure-labels.yaml
kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE      IP
NODE          NOMINATED NODE
quelleheure   1/1     Running   0          1m       10.244.2.26
aks-agentpool-42554519-0 <none>
```

Quel que soit le nombre de fois que vous lancerez la commande `replace`, le *pod* sera systématiquement instancié sur le même *node*. Si en revanche vous remplacez, dans le fichier, le `nodeSelector` par une chaîne de caractères quelconque, après redéploiement nous obtiendrons la chose suivante :

```
kubectl describe pod quelleheure
Events:
  Type        Reason     Age           From            Message
  ----        -----     --           --             -----
Warning FailedScheduling 1s (x16 over 35s) default-scheduler 0/4 nodes are available: 4 node(s)
didn't match node selector.
```

Kubernetes nous indique qu'il a cherché un *node* compatible avec nos exigences mais qu'il n'en a pas trouvé. Notre *pod* est donc dans un état d'attente. Dès qu'un *node* compatible fera son apparition, il sera instancié.



Les *labels* sont généralement utilisés pour marquer des *nodes* ayant des caractéristiques techniques particulières. Admettons que vous disposiez de certains *nodes* avec des disques haute performance et d'autres avec des disques plus classiques. Vous pourriez vouloir réserver ces *nodes* « haute performance » à des *pods* qui en auraient besoin (nœud de base de données par exemple).

Nous avons résolu notre problème initial mais d'une manière peu satisfaisante. Disposer d'un cluster haute disponibilité et s'assurer en même temps qu'un *pod* ne puisse être instancié qu'à un seul exemplaire et toujours sur le même nœud n'est pas franchement très utile.

#### ***Si nous voulions effacer ou visualiser le fichier log.txt***

L'instruction `exec` fonctionne exactement comme son équivalent Docker. Elle ouvre un accès vers le conteneur et permet de naviguer à l'intérieur :

```
kubectl exec -ti quelleheure /bin/bash
# ls -la /data/
total 12
drwxr-xr-x 2      www-data      www-data        4096 Oct  3 22:48 .
drwxr-xr-x 1      root         root          4096 Oct  3 22:47 ..
-rw-r--r-- 1      www-data      www-data        42 Oct  3 22:48 log.txt
```

### **12.3.3 Persistent volumes : la persistance gérée par Kubernetes**

Pour disposer d'un système de persistance distribué avec Kubernetes, il faut utiliser les volumes persistants (*persistent volumes* ou *pv*). Le support des *providers*<sup>7</sup> de volume<sup>8</sup> varie en fonction des implémentations. Il est souvent dépendant de l'infrastructure. Ainsi AKS (l'offre de Microsoft) propose deux plugins principaux : Azure Files et Azure Disk. Ces deux implémentations ont des caractéristiques différentes que nous n'allons pas aborder à ce stade. Sachez seulement que celle qui

correspond à notre cas d'usage est Azure files (figure 12.6).

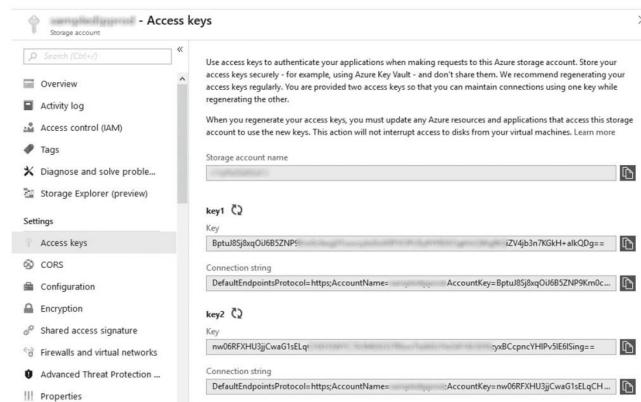
Attention : Azure Files présente des caractéristiques de performance en lecture/écriture qui ne sont pas compatibles avec des processus entrée/sortie intensifs. Il convient pour notre exemple mais doit être utilisé avec prudence en production.

### Création du stockage Azure

En pratique les *persistent volumes* Azure Files s'appuient sur des Azure Storage Accounts<sup>9</sup>. Il s'agit d'un disque virtuel qui est accessible via divers protocoles (REST et SMB), qui est identifié par un nom et dont l'accès est protégé par une clé.

La première opération consiste donc à créer un Azure Storage Account. Nous ne décrirons pas le processus mais la chose est simple et peut être réalisée en ligne de commande ou à l'aide du portail Azure (figure 12.6).

Figure 12.6 – Crédit d'un Azure Storage account



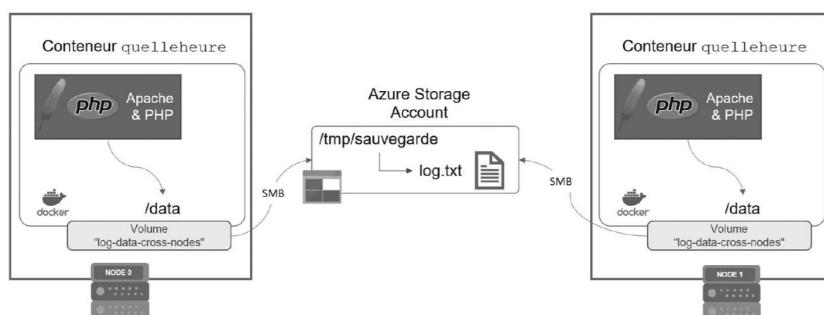
La figure 12.7 fournit une représentation de l'architecture. Les différents nœuds (si notre *pod* est instancié sur différents *nodes*) communiquent avec l'Azure Storage Account. Les volumes montés sont donc, en quelque sorte, des *bind mounts* accédant un disque distant via le protocole SMB. C'est exactement la même technologie que les disques réseau d'un réseau local d'entreprise.

### Création du secret Kubernetes pour l'authentification

Une fois le Storage Account créé, vous devez stocker les paramètres d'accès (soit : `azurestorageaccountname` et `azurestorageaccountkey`) sous la forme d'un secret pour que le provider de *persistent volume* puisse établir la connexion.

```
$ kubectl create secret generic azure-secret --from-literal=azurestorageaccountname=monstorageaccount --from-literal=azurestorageaccountkey="BptuJ8S(...)+alkQDg=="  
secret/azure-secret created
```

Figure 12.7 – Persistent volume avec Azure Files



Ceci est l'occasion d'aborder la fonction des secrets Kubernetes, qui est tout à fait

similaire à celle des secrets Swarm qui a été décrite dans le chapitre précédent. Il s'agit tout simplement d'un contenu chiffré qui est ensuite accessible des *pods* (possiblement monté comme un système de fichiers) ou du système Kubernetes lui-même. En l'occurrence, ici, le secret est utilisé par le *provider* de volume Azure Files pour se connecter à l'Azure Storage Account.

### **Création des objets Kubernetes et du Pod**

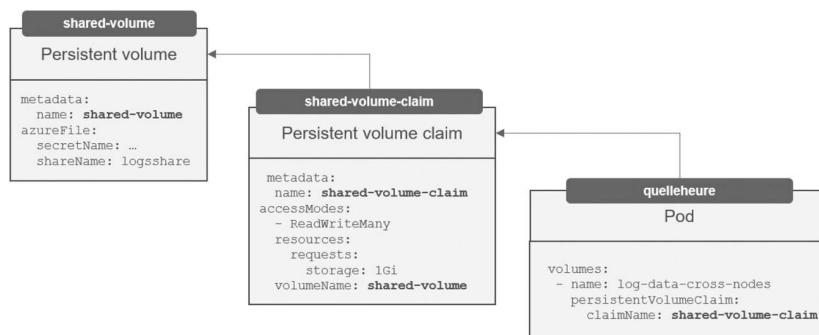
Nous devons maintenant créer les objets Kubernetes. Le modèle objet est décrit dans la figure 12.8 et comprend trois entités :

Le volume persistant lui-même (*pv* ou *persistent volume*), qui contient des paramètres spécifiques du *provider* de volume. C'est donc la partie de la configuration qui dépend de l'instanciation de Kubernetes. Elle serait différente avec Google Compute Engine Persistent Disk ou Amazon Web Services EBS Volume.

Le *persistent volume claim* (ou *pvc*), qui référence le *persistent volume* et constitue une requête » d'accès au volume spécifiant la taille de l'espace demandé et éventuellement d'autres caractéristiques. À l'inverse de l'objet précédent, celui-ci est agnostique de l'implémentation réelle du volume.

Le *pod* qui va référencer le *persistent volume claim* pour son besoin de persistance.

Figure 12.8 – Persistent volume, persistent volume claim et pod



Le code de description se répartit dans trois fichiers YAML différents pour chacun de ces objets. Pour exécuter le déploiement, lancez les fichiers `deploy-*.sh` depuis le répertoire `chapitre12/quelleheure/persistent-volume`. Ils comprennent trois instructions `kubectl` présentées ci-après.

```
| $ kubectl apply -f quelleheure-pv.yaml
```

Vérifiez que le volume a été correctement créé. La création des ressources peut prendre un peu de temps côté Azure. Soyez patient !

```
$ kubectl get pv
NAME          CAPACITY   ACCESS MODES  RECLAIM POLICY  STATUS
CLAIM        STORAGECLASS      REASON AGE
shared-volume  1Gi           RWX        Retain     Available
```

Maintenant nous pouvons lancer la création du *claim*.

```
| $ kubectl apply -f quelleheure-pvc.yaml
```

Et enfin, nous créons le *pod* qui utilise le *claim* pour accéder au volume.

```
| $ kubectl replace --force -f quelleheure-persistent-volume.yaml
```

L'inspection de la description du *pod* et de sa section *Events* nous montre la succession d'opérations réalisées du *mount* du volume à l'instanciation du *pod* lui-même :

```
kubectl describe pods quelleheure
Events:
Type          Reason          Age           From

```

```

Message
Normal          Scheduled           3m               default-scheduler
Successfully assigned quelleheure to aks-agentpool-37386073-0
Normal          SuccessfulMountVolume 3m               kubelet, aks-agentpool-
                                                               37386073-0
MountVolume.SetUp succeeded for volume "default-token-88c7s"
Normal          SuccessfulMountVolume 3m               kubelet, aks-agentpool-
                                                               37386073-0
MountVolume.SetUp succeeded for volume "shared-volume"
Normal          Pulling              3m               kubelet, aks-agentpool-
                                                               37386073-0
pulling image "dunoddocker/quelleheure:v4"
Normal          Pulled               3m               kubelet, aks-agentpool-
                                                               37386073-0
Successfully pulled image "dunoddocker/quelleheure:v4"
Normal          Created              3m               kubelet, aks-agentpool-
                                                               37386073-0
Created          container
Normal          Started              3m               kubelet, aks-agentpool-
                                                               37386073-0
Started container

```

Vous pouvez constater que désormais plusieurs redémarrages du *pod* (via la commande ci-dessous) n'ont pas d'impact sur le log. Quel que soit le *node* sur lequel celui-ci est déployé, le même fichier est lu ou mis à jour grâce à notre volume persistant.

```
| $ kubectl replace --force -f quelleheure-persistent-volume.yaml
```

#### 12.3.4 Hautes disponibilités basique et avancée

Nous avons résolu nos problèmes de persistance. Voyons maintenant comment nous pouvons bénéficier de Kubernetes pour augmenter la résilience et la scalabilité de notre application.

##### Résilience automatique

Regardons ce qu'il advient de notre *pod* quand nous tuons le processus Apache à l'intérieur de notre conteneur :

```

kubectl exec -ti quelleheure /bin/bash
# ps -aux
USER PID      %CPU %MEM    VSZ   RSS      TTY      STAT      START  TIME  COMMAND
...
root 10      0.0  0.0     240432 27548 ?          S          21:02 0:00  apache2 -
DFOREGROUND
...
# kill -9 10

```

Si vous êtes suffisamment rapide, vous pourrez lancer la commande suivante avant que Kubernetes ait réactivé le *pod* (possiblement sur un autre *node*) :

```

$ kubectl get pods -o wide
NAME        READY     STATUS    RESTARTS   AGE       IP
NODE
NOMINATED  NODE
quelleheure  0/1      Error     1          1h       10.244.6.3

```

```
| aks-agentpool-42554519-2 <none>
```

Le contrôleur Kubernetes a détecté que le *pod* était en erreur. Il informe donc le *scheduler* qu'il faut remédier à cette situation, et quelques secondes plus tard :

```
| $ kubectl get pods -o wide
NAME        READY     STATUS    RESTARTS   AGE      IP
NODE        NOMINATED NODE
quelleheure   1/1      Running   2          1h      10.244.6.3
aks-agentpool-42554519-2 <none>
```

Le service est de nouveau disponible !

Regardons maintenant comment Kubernetes est capable de gérer plusieurs instances du même *pod*.

Attention, pensez à détruire le *pod* créé dans les exemples précédents :

```
| kubectl delete pod quelleheure
```

### Répliques : gestion de plusieurs pods au sein d'un même déploiement

Dans la pratique personne ne crée directement de *pod*. Le *pod* est le résultat d'un déploiement (*deployment*). Ce dernier utilise un modèle ou *template* (une sorte de moule à *pod*) pour créer un certain nombre de *pods* et en assurer la distribution sur le cluster à l'aide de règles plus ou moins complexes.

Placez-vous dans le répertoire chapitre12/quelleheure/deployment.

Le descripteur quelleheure-deployment.yaml ressemble beaucoup à celui d'un *pod* mais contient un nouvel attribut :

```
| replicas: 3
```

Celui-ci indique que nous souhaitons que le *pod* soit instancié trois fois.

Laissons de côté les autres changements pour le moment et testons ce déploiement :

```
| kubectl apply -f quelleheure-deployment.yaml
deployment.apps/quelleheure-deployment created
kubectl get pods -o wide
NAME        READY     STATUS    RESTARTS   AGE
IP          NODE
quelleheure-deployment-
6949bf485f-nvc76   1/1      Running   0          5m
172.20.3.69      aks-agentpool-37386073-3
quelleheure-deployment-
6949bf485f-vmfnb   1/1      Running   0          2m
172.20.3.53      aks-agentpool-37386073-1
quelleheure-deployment-
6949bf485f-w84gc   1/1      Running   0          5m
172.20.3.12      aks-agentpool-37386073-0
```

Kubernetes a créé trois *pods* nommés à partir du nom du déploiement et les a répartis automatiquement sur nos quatre *nodes*.

Maintenant, réduisons le nombre de *nodes* à deux (par exemple à l'aide du portal Azure) en éteignant deux des quatre machines virtuelles.

```
| kubectl get pods -o wide
NAME        READY     STATUS    RESTARTS   AGE
IP          NODE
quelleheure-deployment-
6949bf485f-nvc76   1/1      Running   0          5m
172.20.3.53      aks-agentpool-37386073-1
```

quelleheure-deployment-6949bf485f-vmfnb	1/1	Running	0	2m
172.20.3.77	aks-agentpool-37386073-1			
quelleheure-deployment-6949bf485f-w84gc	1/1	Running	0	5m
172.20.3.12	aks-agentpool-37386073-0			

La magie a opéré et nos trois *pods* ont été automatiquement répartis sur les deux *nodes* restants. Le *node* 1 (aks-agentpool-37386073-1) héberge dans ce cas deux instances.

### Services : répartition de charge

Nous pouvons à ce stade nous connecter à l'aide de tunnels à chacun des *pods* mais l'intérêt est limité. Nous devons introduire un nouvel objet pour virtualiser l'accès à notre pool de *pods* : le service.

Regardons à nouveau le fichier quelleheure-deployment.yaml et plus exactement le code suivant :

```
metadata:
labels:
app: qh
```

Il s'agit d'un *label* dont le principe est équivalent à celui que nous avons vu un peu plus haut pour les *nodes*. Ici le *label* marque un type de *pod*. Il crée un moyen de les sélectionner.

Regardons maintenant le descripteur du service quelleheure-service.yaml :

```
apiVersion: v1
kind: Service
metadata:
name: quelleheure-service
spec:
type: LoadBalancer
ports:
- port: 80
targetPort: 80
protocol: TCP
selector:
app: qh
```

Celui-ci instancie un *load balancer* (proxy répartiteur de charge) qui va exposer le port 80 et l'associer au port 80 des *pods* sélectionnés (*selector*) grâce à notre *label* qh. Lançons les instructions suivantes :

```
kubectl apply -f quelleheure-service.yaml
service/quelleheure-service configured
kubectl get services quelleheure-service --watch
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
quelleheure-   LoadBalancer  10.0.212.229  <pending>       80:30236/TCP
service
12s
```

La seconde instruction est bloquante jusqu'à ce que l'IP publique (ouverte sur Internet) soit allouée et assignée à un répartiteur de charge. La chose peut prendre plus ou moins de temps mais généralement, au bout des quelques secondes, une ligne s'ajoute :

```
| quelleheure-service LoadBalancer 10.0.212.229 13.93.65.156 80:30236/TCP    63s
```

L'adresse publique (13.93.65.156) est désormais allouée et accessible depuis Internet. Chaque rafraîchissement de la page nous montre que le répartiteur de charge effectue une distribution séquentielle sur chaque *pod*.

Figure 12.9 – Un service exposé sur Internet



## Quelle heure est-il ?

Hello il est : 11:33:16am

Il semble que je suis installé sur un hôte nommé : quelleheure-deployment-5cb778d686-pxfjg

[Derniers accès à cette page](#)

Access at 07:47:25am

Access at 07:47:26am

Access at 07:47:27am

Access at 11:32:22am

Access at 11:33:15am

Access at 11:33:16am

Évidemment, chaque fournisseur de cloud peut fournir des services additionnels. Il est possible de choisir une adresse IP fixe, de disposer de HTTPS pour un domaine donné, de configurer une affinité de session, etc.

### ***Auto-scaling : montée et répartition de charge automatique***

Dans l'exemple précédent, notre nombre de réplicas était fixe. Il est également possible de programmer Kubernetes pour monter en charge automatiquement et choisir le nombre de réplicas en fonction de la charge.

```
| $ kubectl autoscale deployment quelleheure-deployment --cpu-percent=50 --min=1 --max=5
```

Cette instruction crée un HPA (*Horizontal Pod Autoscaler*) qui va déclencher l'instanciation de pods additionnels (au maximum 5) dès que la charge sur un pod excède 50 % de la limite que celui-ci aura spécifiée dans son descripteur, par exemple :

```
...
containers:
- image:
  dunoddocker/quelleheure:v5 name:
  quelleheure imagePullPolicy:
  Always resources:
  limits:
  cpu: 250m
...
...
```

## **12. 4 DÉPLOIEMENT DE L'APPLICATION EXEMPLE**

Grâce à nos exemples précédents nous avons appris à :

créer des déploiements de *pods* ;

mettre en place une répartition de charge grâce aux services ;

créer des volumes assurant une persistance distribuée.

Nous sommes maintenant prêts à déployer notre application exemple sur Kubernetes.

Reportez-vous au chapitre 8 pour connaître le fonctionnement de l'application. Le chapitre 9 propose un déploiement mono-hôte de cette application.

Placez-vous dans le répertoire chapitre12/application-exemple.

### **12.4.1 Nettoyage**

Faisons un peu de nettoyage de notre application exemple précédente :

```

kubectl delete service quelleheure-service
kubectl delete deployments --all
kubectl delete pvc shared-volume-claim
kubectl delete pyv shared-volume

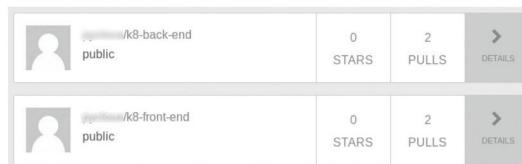
```

### 12.4.2 Build et publication des images

Nous allons fabriquer et publier les trois images nécessaires sur notre compte Docker Hub (database, front-end et back-end). Nous devons faire un nouveau *build* pour que ces images intègrent des paramètres compatibles avec notre nouvelle architecture. Si vous ouvrez le fichier `chapitre12/application-exemple/global.properties`, vous pourrez constater que les noms de domaine de la base de données et du front-end sont différents de ceux que nous avions configurés dans le chapitre 9. Ils correspondent aux noms des services que nous allons définir qui sont automatiquement ajoutés au DNS de Kubernetes.

Lancez le script `build.sh` puis ensuite le script `push-images.sh` qui, comme son nom l'indique, va pousser les images dans le registry Docker (figure 12.10).

Figure 12.10 – Images publiées dans le Docker Hub



### 12.4.3 La base de données

Pour la base de données, nous ne pouvons pas utiliser le même type de volume que lors de nos exemples précédents. Les performances IO ne seraient clairement pas suffisantes. Nous allons donc utiliser un autre *provider* de volume Kubernetes : Azure Disk<sup>10</sup>.

Celui-ci consiste en fait en un disque virtuel monté sur l'hôte (vu comme un disque physique par l'OS). La contrepartie de cette solution est qu'un seul *node* à la fois peut être connecté au disque (le volume est en mode `ReadWriteOnce` contrairement à Azure Files, qui supporte un mode `ReadWriteMany` comme vous pourrez le constater dans le descripteur YAML).

Nous n'aurons donc qu'un seul *pod* de base de données. Nous bénéficierons néanmoins de la résilience offerte par Kubernetes car si le *node* venait à ne plus fonctionner, le *pod* serait réinstancié automatiquement sur un autre *node*.

Il existe des moyens d'installer une base de données en cluster (multi-nœuds) dans un cluster Kubernetes<sup>11</sup>, il est néanmoins plus fréquent d'utiliser une base de données installée sur des machines dédiées ou d'utiliser une solution PaaS (exemple : Azure MySQL sur Azure), essentiellement pour des raisons de simplicité de gestion opérationnelle (backup et autres opérations courantes).

### 12.4.4 De quels objets avons-nous besoin ?

Pour construire notre déploiement, nous allons avoir besoin de spécifier les objets décrits dans le tableau 12.1.

Tableau 12.1 – Les objets Kubernetes requis pour notre application

Objet	Description
chargement-volume	Le volume qui va servir de zone d'échange entre le front-end et le back-end. On utilisera un <i>persistent volume</i> Azure Files qui permettra à plusieurs instances du front-end et au back-end d'accéder aux mêmes fichiers.
database-volume	Le volume qui va servir à la base de données utilisant Azure Disk comme discuté précédemment.
database-service	Le service offrant un accès à la base de données (non ouvert sur une adresse IP publique) exposant le port 3306. Les données de la base de données seront stockées sur le volume database-

	volume.
front-end-service	Le service offrant accès au front-end (ouvert sur Internet et exposant le port 80) et accédant au service de base de données de même qu'au volume chargement-volume.
back-end-deployment	Le déploiement du service back-end accédant au service front-end (appel d'API) de même qu'au volume chargement-volume.

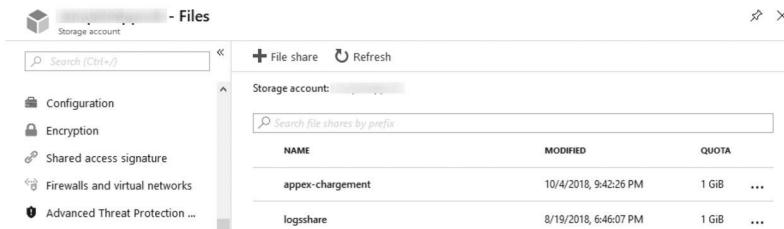
Les fichiers de descriptions YAML se trouvent dans le répertoire chapitre12/application-exemple/deploy.

#### 12.4.5 Création des volumes

##### **Création du partage de fichier pour « chargement-volume »**

Nous allons utiliser le même Storage Account Azure. Nous pouvons donc conserver notre secret qui contient la clé d'accès au compte. Nous devons en revanche créer un nouveau partage de fichiers comme indiqué sur la figure 12.11.

Figure 12.11 – Un nouveau partage de fichier appex-chargement



##### **Création des volumes « chargement-volume » et « database-volume »**

Nous exécutons le descripteur préparé à cet effet :

```
| $ kubectl apply -f creation-volume.yaml
```

Et nous vérifions que tout s'est bien passé et que notre volume est bien disponible :

```
$ kubectl get pv
NAME                                     CAPACITY      ACCESS      MODES      RECLAIM POLICY
STATUS        CLAIM        STORAGECLASS  REASON     AGE
chargement-volume          1Gi          RWX           Retain
Bound        default/chargement-
volume-claim
pvc-eb46efd6-c8e8-11e8-9e94-5e8b1cdd9095   1Gi          RWO          Delete
default/database-volume-claim    default
$ kubectl get pvc
NAME           STATUS      VOLUME
CAPACITY      ACCESS MODES  STORAGECLASS      AGE
chargement-volume-claim  Bound    chargement-volume
1Gi           RWX
database-volume-claim   Bound    pvc-eb46efd6-c8e8-11e8-9e94-5e8b1cdd9095
1Gi           RWO
default
```

En regardant le descripteur `creation-volume.yaml`, vous noterez que nous n'avons pas eu besoin de créer explicitement le `pv` Azure Disk. Le `pvc` (claim) suffit car Kubernetes sait, grâce au paramètre `storageClassName: default`, comment créer la ressource correspondante. Il s'agit d'une propriété propre à l'implémentation spécifique AKS (`dynamic-pv`). Pour GCE (Google), ou

d'autres, des options similaires sont aussi disponibles.

#### 12.4.6 Configuration et déploiement des composants

Nous allons maintenant créer nos différents *pods* en utilisant des déploiements et des services comme indiqué dans le tableau 12.1.

##### Pre-initialisation de la base de données

Comme nous l'avons vu dans le chapitre 9, lorsque nous utilisons un *bind mount* (ce qu'est le *pv* AzureDisk pour le conteneur qui contient la base de données), le contenu du répertoire */var/lib/mysql*, qui contient les fichiers d'installation de MariaDB, est écrasé !

Nous devons donc, avant de lancer notre base de données, copier ces fichiers initiaux dans Azure Disk. C'est une opération que nous n'avons besoin d'effectuer qu'une seule fois.

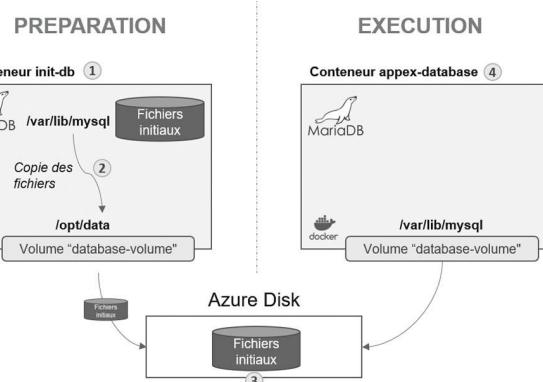
Pour ce faire, nous créons un *pod* qui va nous servir uniquement à copier les fichiers sur l'Azure Disk comme indiqué sur la figure 12.12. Dans celui-ci, le disque sera monté sur le chemin */opt/data*.

```
| $ kubectl apply -f init-db.yaml
```

Attention : l'attachement du conteneur au disque peut prendre du temps. Vérifiez que le *pod* est Running avant de tenter de vous y connecter.

```
$ kubectl exec -ti init-db /bin/bash  
/bin/bash -c "/usr/bin/mysql -u root -proot < /schema.sql"  
cp -r /var/lib/mysql/* /opt/data  
chown -R mysql.mysql /opt/data
```

Figure 12.12 – Initialisation des fichiers de la base de données



Le disque est prêt et nous n'avons plus besoin de notre *pod* d'initialisation. Il faut aussi libérer le *volume-database* qui ne peut être utilisé que par un unique *pod*.

```
| kubectl delete pod init-db
```

##### Lancement des autres composants et test

Il ne nous reste plus qu'à lancer nos différents composants en cascade.

```
kubectl apply -f creation-database-service.yaml  
kubectl apply -f creation-front-end.yaml  
kubectl apply -f creation-back-end.yaml
```

Notre application est désormais déployée avec trois instances de front-end et une instance de back-end. Après quelques minutes, l'adresse IP est allouée au load-balancer du front-end.

kubectl get service				
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE				
appex-front-end	LoadBalancer	10.0.6.254	137.117.204.66	80:30629/TCP

1h

Il ne reste plus qu'à tester l'application via <http://137.117.204.66>.



Dans ce chapitre, nous nous sommes attachés à présenter les principales caractéristiques de Kubernetes.

Il faudra un livre complet, d'ailleurs disponible chez Dunod, pour étudier l'ensemble des fonctionnalités de distribution dynamique de charge. Nous n'avons ici utilisé qu'une petite partie des fonctionnalités disponibles. Mais probablement suffisamment pour vous permettre d'aller plus avant dans vos expérimentations.

Il ne fait pas beaucoup de doute que l'orchestration de conteneurs va se généraliser, ajoutant une nouvelle couche de virtualisation à celle de l'infrastructure (machines virtuelles) et celle de l'OS (Docker et les conteneurs). Essayons maintenant en conclusion d'entrevoir quelques tendances fortes de ce marché.

- 
1. <https://kubernetes.io/docs/setup/minikube/>
  2. <https://kubernetes.io/docs/setup/independent/install-kubeadm/>
  3. <https://azure.microsoft.com/fr-fr/services/kubernetes-service/>
  4. Pour une référence rapide ou complète : <https://kubernetes.io/docs/reference/kubectl/>
  5. <https://kubernetes.io/docs/concepts/configuration/overview/>
  6. Visual Studio code avec son plugin Docker est un très bon éditeur disponible sous tous les OS courants : <https://code.visualstudio.com/>
  7. Le concept de *provider* est similaire à celui des *plugins* Docker.
  8. <https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes>
  9. Azure Storage est le pendant de Amazon S3 pour Microsoft Azure.
  10. <https://docs.microsoft.com/en-us/azure/aks/azure-disks-dynamic-pv>
  11. <https://severalnines.com/blog/mysql-docker-running-galera-cluster-kubernetes>

## Conclusion :

### un potentiel en devenir

travers les différents chapitres de ce livre, nous nous sommes attachés à donner une vue aussi complète que possible de ce que sont 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 prémisses 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 :

**Domaines d’application actuels de Docker**

Domaine d’application	Description / chapitre du livre
Distribution d’application	Nous l’avons abordé à diverses reprises dans ce livre (dès le <b>chapitre 1</b> ). Le Docker Hub 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 considèrent le Docker et son registry public comme le centre de téléchargement officiel de services logiciels.
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 <b>chapitre 2</b> ), permet de conditionner des environnements de développement ou d’intégration pouvant être détruits et reconstruits à loisir. Le <b>chapitre 10</b> 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 <b>chapitre 1</b> ).
Construction d’architectures à micro-services	C’est le propos de solutions CaaS (pour <i>Container as a Service</i> ) qui sont décrites dans le <b>chapitre 2</b> puis, à travers divers exemples, dans les <b>chapitres 8, 9, 10, 11 et 12</b> . 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 start-up Internet, l’architecture à micro-service devrait être une option privilégiée.

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.

Nous ne sommes toutefois qu’au début des changements initiés par la technologie des conteneurs sur les pratiques et l’organisation des départements informatiques.

## 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.

### Autres applications potentielles 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 <i>message queues</i> (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 difficiles à répliquer (et nécessitent parfois de dédier une machine pour un connecteur).
	Des initiatives comme Dray ( <a href="http://dray.it/">http://dray.it/</a> ) 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'applications sur le poste client	Docker est déjà un moyen de distribution d'applications sur le serveur. Docker pourrait-il devenir un moyen alternatif aux <i>package managers</i> et autres <i>Windows installers</i> pour déployer des applications sur le poste client ?
Documentation automatique d'architecture	Comme nous l'avons vu dans le <b>chapitre 7</b> , il est possible d'ajouter de la métainformation 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 métainformation est à la base des interfaces graphiques (pour le moment sommaires) des solutions CaaS. Pour peu qu'un standard de métainformation pour les images Docker émerge et que le pattern « micro-services » soit implémenté, on pourrait envisager des architectures autodescriptives. La documentation (les dépendances, les services exposés) serait alors contenue, non plus dans des documents, mais dans l'environnement d'exécution lui-même. Les dernières versions de Kubernetes sont proches de réaliser cet objectif.

## LES DÉFAUTS DE JEUNESSE DE DOCKER

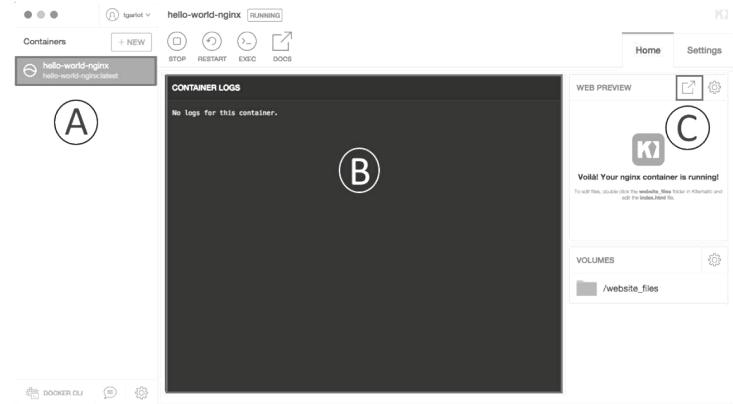
Aussi intéressant que puisse être Docker, il faut reconnaître qu'il s'agit encore d'une solution présentant des défauts de maturité. Ceux-ci se corrigeant peu à peu, mais certains sont encore difficilement acceptables pour un exploitant prudent.

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 start-up française devenue l'une des valeurs montantes de la Silicon Valley saura-t-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 !



# Index

ADD [1](#)  
agent [1](#)  
Amazon ECS [1](#)  
Ansible [1, 2](#)  
Apache Marathon [1](#)  
Apache Mesos [1](#)  
API Docker Remote [1](#)  
application multi-conteneurs [1, 2](#)  
ARG [1](#)  
Atomic [1](#)  
attach [1](#)  
Boot2Docker [1](#)  
bridge [1](#)  
build [1, 2](#)  
CaaS [1, 2, 3](#)  
    architecture [1](#)  
    déploiement d'applications [1](#)  
    solutions [1, 2](#)  
cache [1](#)  
CentOS [1, 2](#)  
CGroups [1, 2](#)  
Chef [1](#)  
CLI [1, 2](#)  
cloud [1](#)  
clustering [1, 2](#)  
    déploiement d'application [1](#)  
CMD [1](#)  
CNM [1](#)  
commit [1](#)  
Compose [1](#)  
composition [1](#)  
Container as a Service *Voir* CaaS [1](#)  
Container ID [1](#)  
container layer [1](#)  
conteneur  
    accès [1](#)

arrêter [1](#)  
clustering [1](#)  
commandes de gestion [1](#)  
communication [1](#)  
composition [1](#)  
connexion [1](#)  
couche [1](#)  
création [1](#)  
création d'une image [1](#)  
cycle de vie [1, 2](#)  
dans une machine virtuelle [1](#)  
définition [1](#)  
démarrage [1](#)  
écosystème [1](#)  
fondations [1](#)  
image [1, 2](#)  
infrastructure [1](#)  
intermodal [1](#)  
montage d'un répertoire [1](#)  
moteur d'exécution [1](#)  
multiple [1, 2](#)  
orchestration [1](#)  
OS spécialisé [1](#)  
suppression [1](#)  
conteneurisation [1](#)  
COPY [1](#)  
copy on write [1](#)  
CoreOS [1](#)  
COW *Voir* copy on write [1](#)  
cp [1](#)  
create [1, 2](#)  
daemon [1](#)  
Data Center Operating System [1, 2, 3](#)  
data volume [1](#)  
DCOS *Voir* Data Center Operating System [1](#)  
démon [1, 2](#)  
devicemapper [1](#)  
diff [1](#)  
Docker  
    alternatives [1](#)  
    avantages [1](#)  
    client [1, 2](#)  
    commandes [1](#)  
    commandes système [1](#)  
    construction image originale [1](#)  
    démon [1](#)  
    groupe [1](#)  
    installation [1](#)

- installation automatisée [1](#)
- intégration continue [1](#)
- manipulation des images [1](#)
- options des commandes [1](#)
- réseau [1](#)
- réseau bridge [1](#)
- variables d'environnement [1](#)
- Docker Compose [1](#)
  - Commandes [1](#)
- dockerd [1](#)
- Docker daemon [1, 2](#)
- Docker Datacenter [1](#)
- Docker Engine [1, 2](#)
- Dockerfile [1, 2, 3](#)
  - commentaires [1](#)
  - instruction format exécution [1](#)
  - instruction format terminal [1](#)
  - instructions [1](#)
  - modèles d'instruction [1](#)
  - programmation [1](#)
- Docker Hub [1, 2](#)
- Docker Machine [1](#)
- Docker Remote API [1](#)
- Docker Swarm [1, 2](#)
- Docker ToolBox [1](#)
- Docker Trusted Registry [1](#)
- Docker UCP [1, 2](#)
- ENTRYPOINT [1](#)
- ENV [1](#)
- etcd [1, 2](#)
- events [1](#)
- exec [1, 2](#)
- export [1](#)
- EXPOSE [1](#)
- forward [1](#)
- FROM [1](#)
- Google Container Engine [1](#)
- guest additions [1](#)
- history [1, 2](#)
- HyperV [1](#)
- IaaS [1, 2, 3](#)
  - déploiement d'applications [1](#)
  - limites [1](#)
- image [1](#)
  - de base [1](#)
- images [1, 2, 3](#)
- import [1](#)
- info [1](#)

Infrastructure as a Service *Voir* IaaS [1](#)

inspect [1](#), [2](#)

intégration continue [1](#)

iptables [1](#)

kernel [1](#)

kill [1](#), [2](#)

Kitematic [1](#)

kubelet [1](#)

kube-proxy [1](#)

Kubernetes [1](#)

    architecture [1](#)

    modèle réseau [1](#)

Kubernetes Control Plane [1](#)

LABEL [1](#)

libcontainer [1](#)

libnetwork [1](#)

ligne de commande *Voir* CLI [1](#)

load [1](#)

load balancer [1](#)

login [1](#)

logout [1](#)

logs [1](#)

LXC [1](#)

mappage des ports [1](#)

MariaDB [1](#)

micro-service [1](#), [2](#)

Namespaces [1](#), [2](#)

network [1](#)

NGINX [1](#)

node [1](#)

ONBUILD [1](#)

OpenSSH [1](#)

OpenVZ [1](#)

orchestration [1](#)

overlay [1](#)

pause [1](#)

persistance [1](#)

pod [1](#)

port [1](#)

port IP [1](#)

ps [1](#), [2](#)

pull [1](#)

Puppet [1](#), [2](#), [3](#)

push [1](#)

registry [1](#), [2](#)

registry cloud [1](#)

rename [1](#)

réseau [1](#)

restart [1](#)  
Rkt [1](#)  
rm [1](#), [2](#)  
rmi [1](#), [2](#)  
run [1](#), [2](#), [3](#), [4](#)  
runC [1](#)  
save [1](#)  
scheduler [1](#), [2](#)  
search [1](#)  
start [1](#), [2](#)  
stats [1](#)  
stop [1](#), [2](#)  
STOPSIG<sup>NAL</sup> [1](#)  
storage driver [1](#)  
Supervisor [1](#)  
Swarm [1](#)  
    cluster [1](#)  
    mode esclave [1](#)  
system df [1](#)  
tag [1](#)  
top [1](#)  
UCP *Voir* Universal Control Plane [1](#)  
union file system [1](#)  
Universal Control Plane [1](#), [2](#)  
unpause [1](#)  
update [1](#)  
USER [1](#)  
Vagrant [1](#)  
variables d'environnement [1](#)  
version [1](#)  
VirtualBox [1](#), [2](#), [3](#)  
virtualisation matérielle [1](#), [2](#)  
VMWare [1](#)  
volume [1](#), [2](#), [3](#), [4](#)  
    création [1](#)  
wait [1](#)  
Windows Nano Server [1](#)  
Windows Server 2016 [1](#)  
WORKDIR [1](#)