

Docker

Déploiement de microservices
sous **Linux** ou **Windows**

(Docker Swarm,
Docker Compose,
Docker Machine)

Jean-Philippe
GOUGOUX



→ Informatique technique

Fichiers complémentaires
à télécharger




Collection

epsilon

Les éléments à télécharger sont disponibles à l'adresse suivante :

<http://www.editions-eni.fr>

Saisissez la référence de l'ouvrage **EPDOCDE** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Chapitre 1

Introduction

1. Contenu du livre	9
1.1 Objectifs	9
1.2 Prérequis de lecture	9
1.3 Avertissement sur les versions de Docker	10
1.4 Approche simplificatrice	12
2. Principes de séparation de l'application, du déploiement et du support	13
2.1 Industrialisation du déploiement	13
2.2 Principes de structuration du système d'information	13
2.3 Découplage obtenu par Docker	15

Chapitre 2

Docker en réseau

1. Mise en réseau de Docker	17
1.1 Approche théorique	17
1.1.1 Problématique de montée en charge	17
1.1.2 Solution découplée	21
1.1.3 Conséquences sur l'approche initiale	22
1.2 Fonctionnement pratique	23
1.2.1 Notion de réseau	23
1.2.2 Docker Swarm ou Swarm mode ?	24
1.2.3 Les différents types de nœuds	27
1.2.4 Fonctionnalités du cluster	28

2.	Préparation de machines avec Vagrant	28
2.1	Principe de Vagrant	28
2.2	Création d'une machine	29
2.3	Provisionnement de Docker sur la machine	35
2.4	Accès à la machine	36
2.4.1	Gestion du certificat	36
2.4.2	Connexion par PuTTY	39
2.5	Mise en place des autres machines	44
3.	Docker Swarm	44
3.1	Initialisation du cluster Swarm	45
3.2	Liaison des agents au cluster Swarm	47
3.3	Ajout d'un manager	50
3.4	Limites à l'ajout de managers	52
3.5	Promotion d'un nœud	54
3.6	Suppression d'un nœud du cluster Swarm	56
3.7	Utilisation simple du cluster Swarm	58
4.	Docker Machine	65
4.1	Concepts	65
4.2	Installation	66
4.3	Mise en œuvre pour une machine seule	67
4.4	Commandes supplémentaires	72
4.5	Accès aux machines	77
4.6	Cycle de vie des machines	78
4.7	Mise en œuvre pour un ensemble de machines	80
4.7.1	Choix du driver	80
4.7.2	Pilotage Azure par la ligne de commande	82
4.7.3	Analyse des options pour le driver Azure	88
4.7.4	Dernières recommandations avant la création d'une machine	95
4.7.5	Déploiement d'une machine dans Azure par Docker Machine	96
4.7.6	Mise en place du cluster Swarm	98
4.7.7	Options Swarm dans la ligne de commande	102

- 4.7.8 Suppression des ressources. 103
- 5. Docker for Azure 105
 - 5.1 Principe 105
 - 5.2 Préparation du Service Principal 108
 - 5.3 Remplissage des autres paramètres 111
 - 5.4 Accès au cluster. 118
 - 5.5 Quelques remarques complémentaires 121
- 6. Azure Container Service. 124
 - 6.1 Concepts 124
 - 6.2 Création d'un cluster ACS 125
 - 6.3 Accès au cluster. 130
 - 6.4 Quelques remarques complémentaires 136
 - 6.4.1 Fonctionnement interne du cluster ACS 136
 - 6.4.2 Autres orchestrateurs disponibles 137
 - 6.4.3 Arrêter le cluster ACS. 138
 - 6.4.4 Autres solutions de CaaS 138

Chapitre 3
Distribution logicielle

- 1. Une première application en mode cluster 139
 - 1.1 Présentation de l'application exemple 140
 - 1.1.1 Fichier docker-compose.yml 140
 - 1.1.2 Fichier Dockerfile 142
 - 1.1.3 Fichier package.json 143
 - 1.1.4 Fichier index.js 144
 - 1.2 Déploiement manuel sur le cluster Swarm. 147
 - 1.2.1 Accès au cluster. 147
 - 1.2.2 Préparation des images. 148
 - 1.2.3 Lancement des services. 149
 - 1.2.4 Premier test 154
 - 1.2.5 Mise en place d'un réseau overlay dédié 155
 - 1.2.6 Validation du fonctionnement 157

1.3	Passage à l'échelle	160
1.4	Alternative d'installation automatisée	162
1.4.1	Le retour de Docker Compose	162
1.4.2	Détail du vocabulaire	163
1.4.3	Déploiement d'une stack	165
1.4.4	Gestion par le registre.	167
1.4.5	Alternative pour l'envoi sur Docker Hub	169
1.4.6	Diagnostic et validation du fonctionnement	179
1.4.7	Utilisation des réseaux Docker	184
1.4.8	Passage à l'échelle	193
1.4.9	Utilisation du DNS.	195
1.4.10	Arrêt de la stack	196
1.5	Le futur avec les bundles	198
1.5.1	Principe des bundles	198
1.5.2	Passage de Docker en mode expérimental	199
1.5.3	Création du bundle.	202
1.5.4	Déploiement du bundle	205
1.5.5	Dernière remarque sur les DAB	205
2.	Informations complémentaires	207
2.1	Évolutions supplémentaires dans la grammaire Dockerfile.	207
2.1.1	Obsolescence du mot-clé MAINTAINER.	207
2.1.2	Passage d'argument à la compilation d'image.	207
2.1.3	Gestion de l'état de santé des conteneurs	208
2.1.4	Approche complètement découplée	210
2.2	Évolutions supplémentaires dans la grammaire Docker Compose.	212
2.2.1	Coopération entre image et build	212
2.2.2	Composition de fichiers.	213
2.2.3	Détails sur les versions.	213
3.	Mise en pratique sur un exemple plus complexe	214
3.1	Contexte	214
3.2	Modifications	214
3.2.1	Récupération de la release	214

- 3.2.2 Modifications du fichier Docker Compose 215
- 3.2.3 Modifications annexes 220
- 3.3 Mise en œuvre 221
 - 3.3.1 Préparation du Swarm 221
 - 3.3.2 Ouverture de la sécurité 223
 - 3.3.3 Récupération du code de l'application 225
 - 3.3.4 Compilation éventuelle des images 226
 - 3.3.5 Lancement de l'application 226
 - 3.3.6 Test de l'application 228
 - 3.3.7 Fermeture pour sécurité 231

Chapitre 4

Maintien en condition opérationnelle d'un cluster

- 1. Utilisation d'un registre 233
 - 1.1 Docker Hub ou registre privé 233
 - 1.2 Registre privé as a Service avec Azure 234
- 2. Gestion de la mise à jour des services 239
 - 2.1 Lien entre évolution logicielle et Docker 239
 - 2.2 Problématique associée 240
 - 2.3 Principe du rolling update 241
 - 2.4 Bonnes pratiques sur les versions applicatives 242
 - 2.5 Mise en œuvre sur un service dans Swarm 243
 - 2.6 Autres ajustements possibles du service 249
- 3. Métadonnées et aiguillage 254
 - 3.1 Problématique 254
 - 3.2 Contraintes 255
 - 3.3 Mise en application 256
 - 3.4 Couplage lâche 258
- 4. Répartition de charge dynamique 259
 - 4.1 Problématique 259
 - 4.2 Solution apportée 261

4.3	Fonctionnement	261
4.4	Préparation du fichier Docker Compose	262
4.5	Ajout des instructions Traefik	265
4.6	Test de fonctionnement	267
4.7	Montage dans une stack Docker	272

Chapitre 5

Docker pour Windows

1.	Docker et Windows	273
1.1	Généralités	273
1.2	Plusieurs fonctionnements	274
1.2.1	Docker Toolbox	274
1.2.2	Docker for Windows	274
2.	Docker dans Windows	276
2.1	Windows 10	276
2.1.1	Installation	276
2.1.2	Concurrence sur VT-X	280
2.1.3	Modes de conteneurs disponibles	282
2.1.4	Premier essai	285
2.2	Windows Server 2016	288
2.2.1	Installation	288
2.2.2	Niveau d'isolation	290
2.2.3	Utilisation depuis un client	290
3.	Spécificités sous Windows	296
3.1	Images de base	296
3.1.1	Nécessité	296
3.1.2	Windows Server Core	296
3.1.3	Nano Server	297
3.1.4	Compatibilité et disponibilité	298
3.2	Différences avec Linux	302
3.2.1	Généralités	302
3.2.2	Dockerfile	302

- 3.2.3 Gestion du réseau 304
- 3.2.4 Autres différences 305
- 3.3 Versions 306
- 4. Paramétrages 307
 - 4.1 Configuration standard 307
 - 4.2 Mode conteneurs Linux 310
 - 4.3 Cas particulier des lecteurs de disque 311
- 5. Premiers conteneurs sous Windows 312
 - 5.1 Remarque préliminaire 312
 - 5.2 Remplacement de Nginx par IIS 313
 - 5.2.1 Création du Dockerfile 313
 - 5.2.2 Compilation du Dockerfile en une image 314
 - 5.2.3 Lancement du conteneur 315
 - 5.2.4 Test du conteneur 316
 - 5.2.5 Suppression du conteneur 317
 - 5.2.6 Mise en place de volumes 318

Chapitre 6
Industrialisation

- 1. Objectifs d'industrialisation du cluster 319
- 2. Vérification du fonctionnement 319
 - 2.1 Supervision matérielle du cluster 321
 - 2.1.1 Définition des tâches 321
 - 2.1.2 Outillage standard 323
 - 2.2 Supervision des conteneurs 324
 - 2.2.1 Définition des tâches 324
 - 2.2.2 Approche bétail plutôt qu'animal domestique 326
 - 2.2.3 Outillage possible 328
 - 2.3 Supervision applicative 330
 - 2.3.1 Définition des tâches 330
 - 2.3.2 Approches d'outillage 331

2.3.3	Point de vue de l'utilisateur	332
3.	Pilotage du cluster	333
4.	Bonnes pratiques logicielles	333
4.1	Importance de la normalisation des logs.	334
4.2	API de statut	334
4.3	Retry politiques et circuit breakers	335
4.4	Répartition de la charge	337
4.4.1	Load-balancing	337
4.4.2	Inversion de consommation	338
4.5	Gestion du multitenant	341
5.	Gestion de la performance	343
5.1	Passage à l'échelle automatique	343
5.2	Cas particulier du cache	344
5.3	Alignement sur le CPU et les threads	345
5.4	Approche "production only"	346
6.	Gestion de la persistance	348
7.	Sécurité	349
7.1	Évolution de la prise en compte.	349
7.2	Sécurisation du cluster	351
7.3	Pratique de sécurisation des ports	351
7.4	Sécurité sur l'utilisation des images.	352
7.4.1	Choix des versions	353
7.4.2	Distribution	353
7.4.3	Docker Security Benchmark	354
8.	Pour aller plus loin	354
8.1	Autres technologies	354
8.2	Cluster hybride	356
8.3	Déploiement et intégration continu	356
	Index	357



Chapitre 2 Docker en réseau △

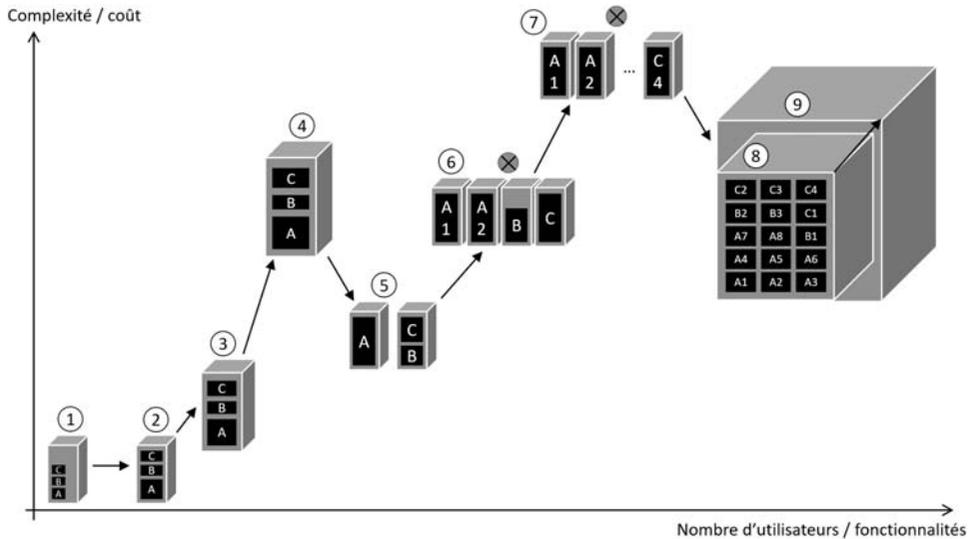
1. Mise en réseau de Docker

1.1 Approche théorique

1.1.1 Problématique de montée en charge

Un problème auquel est confronté presque tout administrateur d'une application web au cours de sa vie est l'augmentation forte des ressources consommées par une application. Des optimisations logicielles ou d'organisation peuvent parfois être réalisées, mais globalement, la ressource utilisée croît avec le nombre d'utilisateurs et les fonctionnalités exposées.

Le schéma ci-dessous montre les différentes étapes par lesquelles un administrateur "traditionnel" peut théoriquement passer :



- L'étape 1 est la mise en production pour un ensemble d'utilisateurs restreints, à savoir généralement les seules personnes connaissant l'application avant qu'elle ne soit globalement répertoriée sur Internet : typiquement, les testeurs et personnes intéressées de l'entreprise publiant le site ou l'application. À ce niveau de charge, tout se passe généralement bien. Chaque module de l'application (des conteneurs Docker, par exemple) est à l'aise avec la part de ressources qu'il peut consommer.
- Imaginons que l'application fonctionne correctement et que les utilisateurs sont au rendez-vous. La croissance aboutira à l'étape 2, à savoir que les conteneurs augmentant leur besoin en ressource, ils occuperont au final toute la puissance de la machine hôte.

- À ce moment-là, un facteur limitant apparaîtra sur la mémoire vive, le stockage, la bande passante réseau ou la puissance de calcul (voire sur plusieurs de ces caractéristiques). Une des approches les plus simples pour éviter des pertes de performance – ou même des réponses en erreur dans le pire de cas – est de réaliser l'opération nommée "scale up" et qui consiste à augmenter la taille de la machine utilisée. C'est ce que nous représentons à l'étape 3. Si le système n'est pas virtualisé, ceci nécessite de réinstaller la totalité des briques logicielles, et si l'application n'a pas été structurée pour gérer cette éventualité, le coût peut être élevé.
- En règle générale, l'administrateur prévoyant qui a été confronté à ce besoin de "scale up" prend ses précautions et, si l'application est prévue pour augmenter encore son empreinte, il choisira une machine suffisamment bien dotée pour voir venir les prochains pics d'exploitation. Pourtant, quelles que soient les ressources financières à disposition, il arrive un moment où il n'est plus possible d'acheter une machine plus grosse (en pratique, la limite où l'application n'est plus en mesure d'exploiter correctement des ressources plus abondantes peut parfois être atteinte avant, typiquement lorsqu'une application ne peut tirer parti des processeurs multiples). C'est l'étape 4 qui représente ce moment dans la croissance théorique d'une application.
- Le coût a augmenté de manière forte sur ces dernières étapes, mais la complexité est toutefois contenue. Le passage à l'étape 5, obligatoire car le "scale up" ne peut plus fonctionner, va permettre de réduire drastiquement les coûts, mais au prix d'une complexité en hausse : il consiste à passer au "scale out", c'est-à-dire à l'augmentation du nombre de machines. Dans un premier temps, comme précédemment, l'approche est relativement simple : il suffit de mettre en place deux machines et de répartir intelligemment les conteneurs entre celles-ci, en fonction de la connaissance que l'administrateur a de leur fonctionnement (connaissance qui nécessite une communication avec les développeurs ayant créé ladite application, ce qui est parfois en soi un problème).

- Ensuite, le "scale out" va lui aussi connaître sa phase de complexification, lorsqu'il va falloir gérer plusieurs instances d'un même service pour continuer à augmenter en performance. Pour cela, il faudra introduire dans le système un mécanisme de répartition de charge (*load balancing* en anglais), comme HAProxy. L'étape 6 montre ce cas de figure, avec deux instances du conteneur A, et le symbole du load balancer au-dessus des machines logiques.
- Si la croissance se poursuit, de plus en plus de machines seront nécessaires pour gérer les multiples instances de chacun des services (comme montré à l'étape 7). De plus, si l'habitude a été prise de gérer des gros conteneurs occupant toute la ressource d'un serveur logique, le nombre de ces derniers va commencer à poser problème.
- La solution proposée à ce problème serait de disposer d'un système qui, bien que composé de multiples machines logiques (physiques ou virtuelles), apparaîtrait comme un seul hôte Docker, ce qui permettrait une gestion plus simple, tout en rendant possible la présence de nombreux conteneurs. C'est cette solution, schématisée par l'étape 8, que nous allons étudier plus avant dans ce chapitre.
- Le principal avantage de cette solution est que la mise à l'échelle étant complètement disjointe entre le nombre de machines ou leur taille et le nombre de conteneurs que l'ensemble va pouvoir porter, il est très simple de faire évoluer indépendamment ces deux caractéristiques. À l'augmentation des performances attendues correspondra l'augmentation du nombre de conteneurs Docker. Ceci se traduira par la nécessité d'augmenter la taille de l'hôte Docker "virtuel", ce qui se fera par ajout de machines dans ce qu'on appelle communément le cluster (grappe en français, mais ce vocabulaire est très peu utilisé). C'est l'étape 9, d'agrandissement du parallélépipède représentant un cluster, qui illustre ceci.

1.1.2 Solution découplée

Revenons un peu plus en détail sur la solution proposée à l'étape 8 ci-dessus. Elle consiste à séparer (on parle parfois de "découplage") la taille et le nombre des machines physiques et virtuelles supportant le démon Docker du dimensionnement de l'application utilisant cette infrastructure. Pour arriver à cette séparation des deux notions, il est nécessaire qu'un client Docker soit capable d'adresser un ensemble d'hôtes Docker de la même manière qu'il appellerait le démon positionné sur une machine hôte seule. Pour cela, il convient que le mode d'appel du démon ne soit pas lié strictement à des caractéristiques d'une machine. L'utilisation du protocole TCP pour échanger avec les démons permet précisément ceci, car seul un identifiant de type IP est nécessaire pour indiquer au client Docker quel démon ou quel ensemble de démons (en passant par une des machines) il doit contacter. C'est ce qui est réalisable en modifiant la valeur de la variable d'environnement `DOCKER_HOST` :

```
■ set DOCKER_HOST = tcp://172.99.79.150:2376
```

Le fait que le contrat entre un client Docker et un serveur Docker soit aussi simple est une première partie de la solution. La seconde tient dans le fait qu'un ensemble de démons Docker en réseau peut être adressé par un seul couple adresse IP + port réseau, correspondant à n'importe lequel des nœuds de gestion du cluster (le vocable anglais de "manager node" est souvent utilisé). Ce nœud de gestion se charge ensuite de "pousser" les bonnes instructions Docker sur les nœuds de travail (on parle de "worker nodes"). Enfin, la troisième partie de la solution tient dans le fait que l'API Docker est contractuelle, c'est-à-dire qu'elle reste la même, qu'on accède à un démon ou à un réseau de démons. Ces trois caractéristiques aboutissent ensemble au découplage précité entre la taille du cluster et le nombre de démons Docker vus par le client, à savoir systématiquement un.

■ Remarque

L'API Docker est désormais standardisée par le biais de deux spécifications de l'Open Container Initiative, un consortium web ouvert de normalisation de la gestion des conteneurs auquel adhèrent Docker ainsi que de nombreuses autres grandes sociétés d'informatique. La première de ces spécifications concerne la gestion de l'exécution des conteneurs (et donc l'API pour les démarrer, les arrêter, etc.) et la seconde, la définition des images. Plus d'informations sur <https://www.opencontainers.org/>.

1.1.3 Conséquences sur l'approche initiale

Nous avons montré l'existence d'un mode que l'on qualifie couramment d'élastique, dans le sens où les conteneurs n'ont pas à se poser la question de la structure des machines virtuelles ou physiques sous-jacentes au démon Docker qui les accueille. Dès lors, la question se pose de directement créer les infrastructures de déploiement dans ce mode, et ce sans attendre que la montée en charge ne nous y contraigne.

Plusieurs arguments vont dans ce sens. Tout d'abord, le fait que le cluster puisse n'être composé que d'une machine permet de limiter au maximum les coûts de fonctionnement en première étape. Ce n'est certes pas le cas pour la part d'investissement de ces coûts, mais nous verrons un peu plus loin qu'elle est très limitée depuis les dernières versions de Docker, l'installation d'un cluster étant à peine plus complexe que l'installation d'un démon Docker sur une seule machine. Elle peut même être limitée encore plus fortement en faisant appel à un fournisseur dédié, qui proposera ce service en ligne (on parle alors de CaaS, pour *Container as a Service*).

De plus, le coût des changements de machines et d'infrastructures tels que montrés en théorie dans les étapes décrites plus haut est en général très élevé, non seulement pour le matériel mais aussi à cause de la mise en œuvre, potentiellement complexe, et ceci, sans même compter les coûts en termes d'image à cause des possibles ruptures de services dans ces moments à risque pour la production.

À l'inverse, il ne faut pas négliger le coût de formation à cette nouvelle façon de gérer les serveurs, qui va à l'encontre de ce que bon nombre d'administrateurs ont mis en pratique pendant des années. Dans le mode traditionnel, l'administrateur connaît chacun des serveurs par son nom, il sait ce qui se trouve dessus, a l'habitude de gérer telle ou telle spécificité. Lorsque le serveur montre des signes de fatigue, sa connaissance de ce serveur particulier lui permet de rapidement déterminer la criticité du problème et l'urgence qu'il y a à le régler par rapport à d'autres. Dans ce nouveau mode où les machines ne sont que des supports d'un cluster, l'administrateur les gère de manière indifférenciée. Il en va d'ailleurs de même pour les conteneurs, puisqu'il ne peut pas savoir, dans un cluster, sur quelle machine quels conteneurs vont s'exécuter. Le système peut même arrêter dynamiquement un conteneur sur un serveur en même temps qu'il le redémarre sur un autre.