



Expert

Docker

pour les développeurs **.NET**

En téléchargement

-  scripts d'installation
-  code source
-  fichiers Dockerfile

 + QUIZ

Version en ligne
OFFERTE !
pendant 1 an

Christophe **MOMMER**



Les éléments à télécharger sont disponibles à l'adresse suivante :
<http://www.editions-eni.fr>
Saisissez la référence ENI de l'ouvrage **EIDOCNET** dans la zone de recherche
et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

Chapitre 1

Introduction

- 1. Présentation de Docker 9
- 2. Concept de conteneurs 10
 - 2.1 Avantages pour l'infrastructure 10
 - 2.1.1 Absence de modification du système 11
 - 2.1.2 Performances accrues 12
 - 2.2 Avantages pour le développement 15
 - 2.2.1 Facilité et rapidité d'évaluation 15
 - 2.2.2 Processus pilotable 15
 - 2.2.3 Ressources jetables 16
 - 2.2.4 Portabilité 17
- 3. Conclusion 18

Chapitre 2

Mise en place

- 1. Introduction 19
- 2. Installation sous Linux 19
 - 2.1 Ajout du dépôt officiel 20
 - 2.2 Installer Docker et vérifier 22
- 3. Installation sous macOS 24

4. Installation sous Windows	26
4.1 Docker Desktop (méthode classique)	26
4.2 Docker Toolbox	29
4.3 WSL 2	30
4.4 Interfaces graphiques	32
4.4.1 Kitematic	33
4.4.2 Docker Dashboard	35

Chapitre 3

Le CLI

1. Fonctionnement du CLI	39
1.1 État de l'installation	39
1.2 Récupération et exécution d'une image	42
2. État du système	44
2.1 Management des images et conteneurs	45
2.1.1 Gestion des images	45
2.1.2 Gestion des conteneurs	48
2.2 Interagir et écouter un conteneur	50
2.2.1 Cycle de vie	50
2.2.2 Monitoring	54
2.2.3 Interactions manuelles	57
3. Interactions avec le système hôte	61
3.1 Volumes	61
3.1.1 Syntaxe	62
3.2 Réseau	66
3.2.1 Syntaxe et principes de base	66
3.2.2 Gestion du réseau	68
3.3 Nettoyage	72
4. À vous de jouer !	73
4.1 Énoncé	73
4.2 Corrigé	74

Chapitre 4
Dockerfile

- 1. Principes et syntaxe 77
 - 1.1 Instructions FROM et WORKDIR 78
 - 1.2 Instruction RUN..... 80
 - 1.3 Docker build 82
 - 1.3.1 Tag des images 83
 - 1.3.2 Contexte de build..... 92
 - 1.4 Instruction COPY..... 93
 - 1.5 Instruction ENTRYPOINT et CMD..... 96
 - 1.5.1 Généralités 96
 - 1.5.2 Spécificités 98
- 2. Concepts avancés 99
 - 2.1 Cache..... 99
 - 2.2 Utilisation des couches précédentes 101
 - 2.3 Exposition réseau 104
 - 2.4 Variables d'environnement 106
 - 2.5 Volumes 108
 - 2.6 Argument de build 109
 - 2.7 Envoyer votre image..... 110
 - 2.8 Sécurité 115
 - 2.8.1 S'assurer de la mise à jour..... 115
 - 2.8.2 Gérer le niveau de droit 117
- 3. Exercice 119
 - 3.1 Énoncé..... 119
 - 3.2 Corrigé..... 120

Chapitre 5 Docker Compose

1. Introduction	121
2. Syntaxe du fichier Compose	122
2.1 Structure d'un fichier Compose	123
2.2 Bloc de version	124
2.3 Bloc services	126
2.3.1 Image	126
2.3.2 Build	127
2.3.3 Identité et dépendance	128
2.3.4 Gestion du réseau	131
2.3.5 Volumes	135
2.3.6 Pilotage avancé	138
2.4 Bloc réseau	142
2.5 Bloc volumes	146
3. CLI	147
3.1 Mise en place	147
3.2 Monitoring	151

Chapitre 6 Docker et l'usine logicielle

1. Introduction	153
1.1 Intégration continue	155
1.2 Déploiement continu	156
1.3 La place du développeur	157
2. Pipeline DevOps	158
2.1 Création manuelle	159
2.1.1 Environnement de build	159
2.1.2 Exécution des tests	163
2.1.3 Création de l'image finale	165
2.1.4 Automatisation du processus	166

- 2.2 Azure DevOps 169
 - 2.2.1 Initialiser l'environnement 169
 - 2.2.2 Création du pipeline 171
 - 2.2.3 Création du dépôt sur Azure 177
 - 2.2.4 Mise à jour du pipeline
pour mettre à disposition l'image 181
 - 2.2.5 Déploiement automatique sur Azure 189
- 3. Outils pour le développement 197
 - 3.1 Dépôt privé 197
 - 3.1.1 Dépôt officiel 198
 - 3.1.2 Sonatype Nexus 199
 - 3.2 Outil d'analyse de code 204
 - 3.2.1 Déploiement du serveur Sonarqube 204
 - 3.2.2 Analyse avec l'outil global dotnet 207
 - 3.2.3 Couplage à Visual Studio sous Windows 209
 - 3.3 Outil de monitoring 212
 - 3.3.1 Surveillance globale 212
 - 3.3.2 Test de montée en charge 215

Chapitre 7
Aller plus loin avec les outils de développement

- 1. Introduction 217
- 2. Visual Studio pour Windows 217
 - 2.1 Assistant d'intégration de Docker 218
 - 2.2 Fenêtre de gestion des conteneurs 219
 - 2.2.1 Détails d'un conteneur 220
 - 2.2.2 Interaction avec le conteneur 221
 - 2.2.3 Interaction avec les images 223

3.	Visual Studio Code	224
3.1	Vue du système	225
3.1.1	Conteneurs	226
3.1.2	Images	227
3.1.3	Registries	228
3.1.4	Networks	228
3.1.5	Volumes	229
3.2	Amélioration de l'éditeur	229
4.	Conseils généraux aux développeurs	232
4.1	Proche de la production	232
4.2	Accélérer votre workflow	233
4.3	Prendre en compte la sécurité	234

Chapitre 8

Conteneurs Windows

1.	Introduction	237
1.1	Fonctionnement de la licence	238
1.2	Changement sous Windows 10	239
1.3	Activation sous Windows Server	240
1.3.1	Installation	240
1.3.2	Mise à jour	242
1.4	Version de l'image	243
1.5	Différences des images de base	245
2.	Spécificités Windows	247
2.1	Volumes	247
2.2	Spécificités du Dockerfile	249
2.3	Couches protégées	250
3.	Outils spécifiques	251
3.1	Dépôt local	251
3.2	Outils de monitoring	254

Table des matières _____ 7

4. Déployer une application .NET Framework	255
4.1 Étape de build	256
4.2 Étape d'exploitation	257
4.3 Installation des outils de management	259
4.4 Finalisation du Dockerfile	263
Index	265

Chapitre 4 Dockerfile

1. Principes et syntaxe

Vous êtes maintenant armés pour pouvoir manipuler Docker grâce au CLI. Cependant, vous avez jusqu'à présent utilisé uniquement des images officielles publiées sur le Docker Hub.

Pour pouvoir créer ces images, ceux qui les ont publiées ont dû les construire au préalable. Ainsi, pour avoir la possibilité d'exploiter vos créations à l'aide de Docker vous allez devoir, à votre tour, créer vos propres images Docker. Ceci est possible grâce à un fichier appelé **Dockerfile**.

Dans la logique, il faut voir le Dockerfile comme étant une succession d'instructions ordonnées, analogue à une recette de cuisine, nécessaire pour créer l'image. Ce fichier, sans extension, se comporte comme un fichier texte (il peut être édité simplement avec un bloc-notes). Pour suivre l'exercice, il est conseillé de se placer dans un dossier vide individuel et d'y créer un fichier nommé "Dockerfile" sans extension.

■ Remarque

Appeler le fichier "Dockerfile" est une convention et n'est pas forcément nécessaire. On peut très bien l'appeler autrement et renseigner ce nom lors des instructions de génération de l'image. L'avantage de suivre cette convention est que l'on va éviter l'obligation d'ajouter un argument supplémentaire à notre ligne de commande. Il est recommandé de la suivre s'il n'y a qu'un seul fichier Dockerfile dans le dossier.

Explorons les différentes instructions nécessaires pour construire un Dockerfile.

1.1 Instructions FROM et WORKDIR

■ Remarque

Par convention, les instructions du Dockerfile s'écrivent en majuscule. Il s'agit uniquement d'une convention et ce n'est pas obligatoire. Cependant, adopter cette pratique améliorera à la lisibilité de votre fichier.

Pour commencer, il est souhaitable de partir d'une image existante pour pouvoir construire le Dockerfile. Il faut savoir qu'il est possible de partir d'un très bas niveau, auquel cas il faudra choisir une image correspondant à un système Linux, comme **Debian** ou **Alpine**. Cependant, en tant que développeur, ce choix sera rarement effectué, car il existe des images préconfigurées de vos environnements favoris (que ce soient les SDK, mais aussi les *runtimes*).

Pour pouvoir travailler efficacement, partir d'une image déjà composée est recommandé pour gagner du temps. Dans le cas des images .NET Core par exemple, Microsoft en propose deux types de base : le runtime ou le SDK. Le **runtime** sera préféré pour construire une image prête à l'exploitation, ne comprenant que les fichiers binaires de l'application finale, alors que le SDK sera utilisé pour toutes les opérations entrant dans le cadre du développement (phases de *build*, de tests, etc.).

L'instruction FROM du Dockerfile est systématiquement la première ligne dans nos fichiers, et elle détermine à partir de quelle image vous souhaitez commencer.

Par exemple, pour partir du SDK de .NET Core 3.1, on utilisera l'instruction FROM suivante :

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1
```

■ Remarque

On constate ici que la version de l'outil est précisée. À l'instar de ce que l'on a vu dans le chapitre précédent avec la version latest, il est encore plus important dans le cas d'un Dockerfile de préciser la version, par mesure de sécurité et pour les performances. Ce chapitre expliquera plus en détail comment le cache de Docker se comporte avec les différentes lignes d'instructions d'un Dockerfile.

En utilisant cette première instruction, vous allez définir le contexte d'exécution qui statuera que cette image est la vôtre, tout en partant d'une base existante. Cette instruction FROM nous permet de nous assurer que nous avons le SDK de .NET Core 3.1, avec la dernière version de patch et avec tous les outils nécessaires.

■ Remarque

Si vous avez besoin de partir sur une création totalement vierge, utilisez l'instruction FROM scratch.

Si on se contente d'enchaîner les instructions, elles vont dorénavant être exécutées dans le contexte d'exécution issu du lancement de cette image. Dans le cas du SDK .NET Core par exemple, c'est à la racine du dossier home de l'utilisateur root courant que seront exécutées la totalité des instructions. Un développeur va plutôt souhaiter se placer dans un répertoire de travail vierge, afin de ne pas être parasité.

Cette opération peut être réalisée grâce à l'instruction WORKDIR, qui permet de spécifier le nom du répertoire de travail.

```
WORKDIR [NOM_DOSSIER]
```

À la suite de cela, on se retrouve dorénavant dans un répertoire (créé automatiquement s'il n'existe pas) où nous avons une totale liberté pour notre travail. Cette commande, bien que facultative, est souvent recommandée pour éviter de rencontrer des conflits, souvent dus à la présence de fichiers indésirables propres à l'image qui se mélangent à nos propres fichiers, ou encore à des problèmes de droits d'écriture/exécution.

■ Remarque

WORKDIR définit le répertoire courant. Il est dès lors possible de l'utiliser plusieurs fois pour naviguer de répertoire en répertoire au fil des instructions.

1.2 Instruction RUN

Une fois que le contexte de travail est défini (image de base et répertoire de travail), nous pouvons exécuter des commandes au sein de l'image. Pour faire un parallèle, il faut voir l'instruction RUN comme l'équivalent de l'exécution d'une ligne de commande sur un terminal. Ainsi, cette instruction va permettre d'exécuter des commandes pour la bonne création de l'image. Par exemple, il serait possible de préinstaller certains outils sur une image Debian grâce à cette commande en lançant l'outil `apt-get`.

Lors de chaque instruction RUN, Docker crée un conteneur temporaire, permettant l'exécution de cette commande, qui sera utilisé comme base pour l'instruction RUN suivante. Ainsi, il est généralement préférable de chaîner les commandes de console dans une seule instruction RUN plutôt que de multiplier ces dernières, afin de profiter du mécanisme de cache pour gagner en vitesse. En effet, les reliquats de chaque conteneur intermédiaire vont se retrouver dans le conteneur final de façon cumulative, ce qui pourra avoir un impact sur la taille de l'image.

Chapitre 4

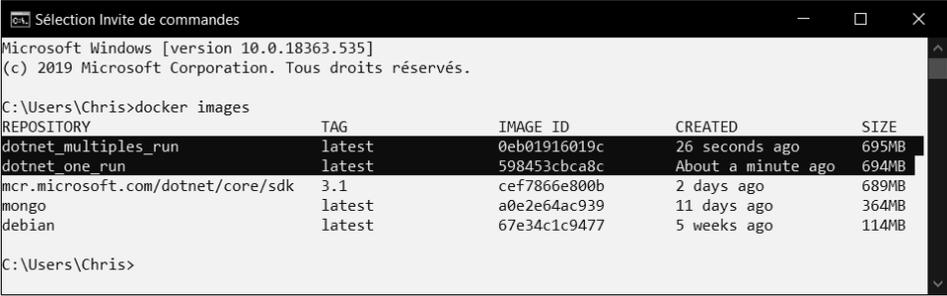
Dans notre exemple, étant donné que la base du Dockerfile est l'image du SDK .NET Core, il est possible de demander la création d'un nouveau projet, puis d'exécuter les commandes `restore`, `build` et `publish` à l'aide du CLI .NET Core :

```
RUN dotnet new console && \  
dotnet restore && \  
dotnet build && \  
dotnet publish -o publish  
WORKDIR publish  
RUN dotnet testproject.dll # affichera "Hello World!" Sur la console
```

Il est également possible de décomposer cette instruction en plusieurs instructions `RUN` indépendantes :

```
RUN dotnet new console  
RUN dotnet restore  
RUN dotnet build  
RUN dotnet publish -o publish  
WORKDIR publish  
RUN dotnet testproject.dll # affichera "Hello World!" Sur la console
```

Cependant, sur le résultat de l'image finale, on peut constater une différence en termes de place sur le disque local :



REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
dotnet_multiples_run	latest	0eb01916019c	26 seconds ago	695MB
dotnet_one_run	latest	598453cbca8c	About a minute ago	694MB
mcr.microsoft.com/dotnet/core/sdk	3.1	cef7866e800b	2 days ago	689MB
mongo	latest	a0e2e64ac939	11 days ago	364MB
debian	latest	67e34c1c9477	5 weeks ago	114MB

Figure 1 : Différence de build avec instruction `RUN` unique et multiple

Remarque

La différence n'est pas ici extrêmement significative (1 Mo), car les couches intermédiaires créées par chaque instruction `RUN` ne sont pas lourdes. Dans le cas d'opérations plus coûteuses, cela peut vraiment faire la différence.

1.3 Docker build

Si vous avez suivi les instructions jusqu'ici, vous devriez avoir un fichier appelé `Dockerfile` dans votre répertoire. Cependant, vous ne pouvez encore l'utiliser avant de l'avoir transformé en image. La commande qui permet de réaliser ceci est l'instruction `build` du CLI.

Cette commande n'a pas été vue dans le chapitre précédent, car il n'y avait pas matière à l'exploiter sous forme d'exercice. Elle est donc détaillée maintenant.

La syntaxe de cette commande ressemble à toute commande du CLI :

```
docker build [ARGUMENTS] [PATH]|[URL]|-
```

Comme on peut le voir, elle présente cependant une spécificité : elle prend en paramètre final un **chemin** (PATH), une **URL** ou un **flux** (caractère -). Si l'on souhaite réaliser un *build* de l'image du répertoire actuel, on pourrait utiliser au choix :

- `docker build .`
- (PowerShell) `Get-Content Dockerfile | docker build -`
- (CMD) `more Dockerfile | docker build -`

Le résultat sera totalement identique, le CLI analysera les instructions de votre `Dockerfile` pour les traiter de façon séquentielle.

Une option très intéressante lorsque l'on réalise un *build*, pour s'assurer qu'on n'utilise pas d'anciens reliquats de cache, est l'option `--no-cache`. En toute logique, cette option indiquera au CLI d'effectuer la totalité des opérations sans tenir compte du cache local, pour s'assurer de construire l'image sur une base neuve.

À la suite de cette opération, si vous n'avez précisé aucun argument, vous obtiendrez une image "anonyme" sur votre système (vous pourrez le constater grâce à la commande qui liste les images présentes, une ou plusieurs images taguées <none> seront listées). Bien évidemment, on va vouloir différencier nos images avec un nom facile à utiliser pour n'importe quelle opération en ayant besoin (comme le lancement d'un conteneur par exemple). Cela nous permettra de travailler avec ce nom plutôt qu'avec un ID. Cette opération correspond à "taguer" son image.

1.3.1 Tag des images

Afin de pouvoir aborder cette section, il est nécessaire de comprendre le fonctionnement de l'organisation du Docker Hub ainsi que les informations remontées par un listing des images.

À l'exécution de la commande `docker images`, ce sont les deux premières colonnes qui nous intéressent dans le cas présent. Il s'agit des colonnes **repository** et **tag**. Jusqu'ici, nous nous sommes contentés de lancer des commandes `docker run` avec un nom d'image simple, comme par exemple `docker run debian`. Implicitement, cela signifie qu'on veut récupérer l'image Debian depuis la racine du Docker Hub et non dans une organisation particulière.

En effet, Docker Hub est une place d'échange publique. Ainsi, chacun peut librement pousser ses images pour que d'autres puissent les télécharger et les utiliser. Cela signifie également qu'il faut pouvoir distinguer les images produites par chacun. Pour répondre à cette problématique, Docker Hub introduit une notion **d'organisation**. Dès lors que vous avez un compte sur Docker Hub avec un nom d'utilisateur, il correspond à votre organisation personnelle.