INFORMATIQUE
TECHNIQUE

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

eni

Table des matières

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

Chapitre 3
Le CLI

Chapitre 4
Dockerfile

Chapitre 5
Docker Compose

Chapitre 6
Docker et l'usine logicielle

Chapitre 7

Aller plus loin avec les outils de développement

Chapitre 8
Conteneurs Windows

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.

© Editions ENI - All rights reserved

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.

© Editions ENI - All rights reserved

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 :

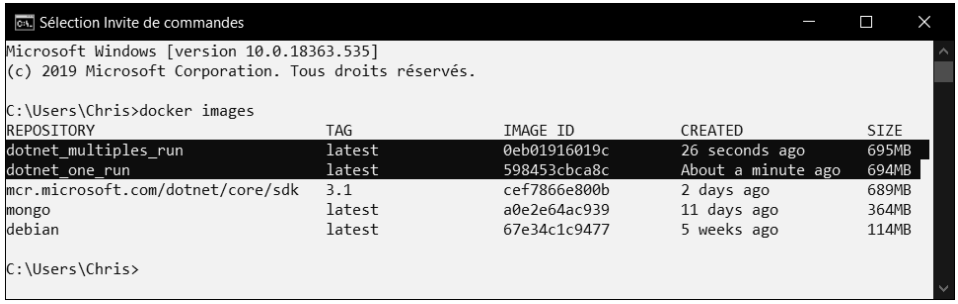

*Figure 1 : Différence de build avec instruction **RUN** unique et multiple*

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.

© Editions ENI - All rights reserved

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.