Tuto Docker - Démarrer Docker (Partie 2)

Cet article fait partie d'une série de billets portants sur Docker et son environnement :

- Tuto Docker : Comprendre Docker (Partie 1)
- Tuto Docker : Démarrer Docker (Partie 2)
- Tuto Docker : Commandes Docker (partie 3)

Docker est un système permettant de gérer des containers, et est capable de faciliter l'utilisation de parties les plus obscures.

Docker repose sur une **API RESTFul**. À cela, s'ajoute un « *docker-cli* » permettant de facilement exécuter des commandes depuis notre terminal.

Depuis un terminal, si vous exécutez la commande *docker,* vous obtiendrez une liste de commandes exécutables, que voici :

```
$ docker

…

Commands:

    attach    Attach to a running container

    build     Build an image from a Dockerfile

    commit    Create a new image from a container's changes

    cp         Copy files/folders from a container's filesystem to the host
path

    create    Create a new container

    diff      Inspect changes on a container's filesystem

    events    Get real time events from the server

    exec      Run a command in an existing container

    export    Stream the contents of a container as a tar archive

    history   Show the history of an image

    images    List images

    import    Create a new filesystem image from the contents of a tarball

    info      Display system-wide information

    inspect   Return low-level information on a container

    kill      Kill a running container

    load      Load an image from a tar archive

    login     Register or log in to a Docker registry server
```

```
    logout    Log out from a Docker registry server
    logs      Fetch the logs of a container
    port      Lookup the public-facing port that is NAT-ed to PRIVATE_PORT
    pause     Pause all processes within a container
    ps        List containers
    pull      Pull an image or a repository from a Docker registry server
    push      Push an image or a repository to a Docker registry server
    restart   Restart a running container
    rm        Remove one or more containers
    rmi       Remove one or more images
    run       Run a command in a new container
    save      Save an image to a tar archive
    search    Search for an image on the Docker Hub
    start     Start a stopped container
    stop      Stop a running container
    tag       Tag an image into a repository
    top       Lookup the running processes of a container
    unpause   Unpause a paused container
    version   Show the Docker version information
    wait      Block until a container stops, then print its exit code
```

…

Docker et ses images

Jusque là, les images n'ont pas encore été évoquées. Nous avons uniquement parlé de containers. Et pourtant, les images Docker sont d'une importance cruciale. En voici un peu plus sur ces images, si essentielles dans le fonctionnement de **Docker.**

Une image est un container statique. On pourrait comparer une image à une capture d'un container à un moment donné, d'une sorte de *snapshot* d'un de vos containers. Lorsqu'on souhaite travailler avec un container, on déclare forcément un container à partir d'une image.

De plus, les images Docker fonctionnent grâce à de l'héritage d'autres images. Votre image de Tomcat hérite elle-même de l'image de Java. Cette même image de Java qui a peut-être été construite à partir d'une Debian. Les héritages peuvent ainsi aller très loin ! Le container créé à partir d'une image contient le **delta** entre l'image de base à partir de laquelle le container a été instancié et l'état actuel. Grâce à ce système, la duplication de donnée est faible.

Le point technique !

Une image peut avoir été construite à partir d'une autre image qui elle-même a pu être construite à partir d'une autre image. Ce système fonctionne parfaitement grâce à un système d'empilement de containers. Par conséquent, lorsque vous construisez une image à partir d'une autre image vous stockez en réalité tous les containers qui vous ont permis de passer de votre image de base à votre image finale. Vous pouvez visualiser ce que je vous dis en exécutant la commande « *docker history* .
Vous pouvez à tout moment voir votre « bibliothèque » d'images avec la commande "*docker images*". Vous verrez, au fil des projets, vous aurez de plus en plus d'images. Utilisables comme "moules" à votre guise pour débuter vos projets ou, à minima, vos nouveaux containers.

Pour voir votre bibliothèque d'images :

```
$ docker images
REPOSITORY      TAG     IMAGE   ID      CREATED         VIRTUAL         SIZE
```

Comment Récupérer (pull) une image Docker ?

Il existe une plateforme **maintenue par Docker** sur laquelle **tout le monde peut *pusher* et *puller* des images**. C'est un peu comme le **GitHub** des images **Docker !** Dès maintenant, nous allons tenter de récupérer une image. Il existe des images officielles pour tout et n'importe quoi ! Cette bibliothèque géante partagée, c'est le **Docker Hub Registry !**

Vous pouvez chercher une image d'une distribution ou d'un produit déjà installé. Commençons par trouver une image Docker que nous pourrions utiliser. Vous disposez de deux moyens pour en chercher une image : via l'interface web, ou via la commande "*docker search"*.

Je vais utiliser la commande "*docker search"* (car je suis un barbu) pour trouver des distributions Debian, qui possèdent suffisamment d'étoiles (confiance des utilisateurs de la plateforme). La commande est la suivante.

```
$ docker search --stars=10 debian
NAME            DESCRIPTION                                     STARS     O
FFICIAL   AUTOMATED

debian          (Semi) Official Debian base image.              248       [
OK]

google/debian                                                             2
5                   [OK]

tianon/debian   use "debian" instead - https://index.docke...   13
```

Et voilà les résultats disponibles avec mes critères. Commençons simple : ce qui m'intéresse, c'est une**Debian** officielle. Il y en a bien une, nommée debian, cotée 248

étoiles (!!) et flaguée [OK] sur l'attribut OFFICIAL. En avant, Moby Dock, pull cette image !

```
$ docker pull debian
```

…

Et voilà, après quelques secondes de téléchargement, vous disposez d'une image sur votre hôte. Vous pouvez d'ailleurs la voir en exécutant "*docker images*". Commande qui liste les images présentes sur votre machine.

Il existe un autre moyen moins utile au quotidien (mais qui peut tout de même servir), qui est d'importer une image avec la commande "**docker load"** depuis un fichier **.tar.gz,** exportée initialement par un Docker (celui d'un collègue, par exemple).

Comment créer une image ?

Désormais nous voulons apprendre à travailler de manière autonome et construire nos images à notre guise. Il existe plusieurs méthodes :

1. le **Dockerfile** : déjà vu dans le premier article, il vous permet entre autres de partir d'une image initiale, de lancer des actions et de construire une nouvelle image.
2. de lancer un container, effectuer des actions soit-même et *commiter* (avec la commande "*docker commit*") les modifications dans une nouvelle image.

Aujourd'hui je ne vais pas détailler la deuxième méthode, qui pour moi est moins utile dans un premier temps. A titre personnel, je me sers de cette méthode pour historiser mes personnalisations. Cela me permet de garder des états étape par étape de ma customisation de containers, afin de pouvoir repartir d'une étape bien précise, par exemple (d'où l'utilisation du terme *snapshot* utilisé plus haut).

Les différences entre un container et son image peuvent être vues grâce à la commande "*docker diff*".

Créer une image avec un Dockerfile

Ce fichier comporte un mélange d'instructions et de métadonnées. Grâce à ce ficher, vous donnez la recette pour que **Docker** puisse construire votre image. Le site officiel décrit bien évidemmentl'intégralité des instructions qui peuvent être réalisées.

Commençons par créer un fichier nommé *Dockerfile* (sans extension ! Le nom de ce fichier n'est pas négociable) dans le dossier de notre choix, et indiquons lui quelles sont les prérogatives pour la création d'une nouvelle image. Pour notre exemple, nous

souhaitons partir d'une Debian Wheezy (ou Debian 7), dans laquelle nous ajoutons Nginx.

Ligne par ligne, détaillons notre futur Dockerfile :

1 - J'indique la distribution de départ avec la ligne

```
FROM debian:wheezy
```

2 - Je nomme la personne qui à écrit ce **Dockerfile**, ici c'est moi. Grâce à la ligne

```
MAINTAINER Baptiste Donaux <bdonaux@wanadev.fr>
```

3 - Je recherche les paquets disponibles et j'installe **Nginx**.

```
RUN apt-get update \
    && apt-get install -y \
        nginx
```

4 - Je copie successivement les configurations et scripts de mon système hôte vers mon image

```
COPY nginx.conf /etc/nginx/nginx.conf
COPY service_start.sh /home/docker/script/service_start.sh
```

5 - J'applique les droits pour exécuter mon script

```
RUN chmod 744 /home/docker/script/service_start.sh
```

6 - Je définie un point d'entrée : le premier script qui va se lancer au démarrage du container

```
ENTRYPOINT /home/docker/script/service_start.sh
```

7 - Le dossier dans lequel je serai quand j'exécuterai un nouveau container sera **WORKDIR**

```
WORKDIR /home/docker
```

Nous avons toutes les lignes pour faire notre Dockerfile. Le voilà en version complete :

```
FROM debian:wheezy


MAINTAINER Baptiste Donaux <bdonaux@wanadev.fr>


RUN apt-get update \
```

```
    && apt-get install -y \
        nginx

COPY nginx.conf /etc/nginx/nginx.conf

COPY service_start.sh /home/docker/script/service_start.sh
RUN chmod 744 /home/docker/script/service_start.sh

ENTRYPOINT /home/docker/script/service_start.sh
WORKDIR /home/docker
```

Construire une image depuis un Dockerfile

Maintenant que nous avons un **Dockerfile**, nous voulons créer notre image. Et la commande magique est... « *docker build »* !

```
$ docker build .
```

Lorsque vous exécutez « *docker build »,* vous devez spécifier le chemin du **Dockerfile** (d'où le point à la fin de la commande si vous lancez la commande depuis le même endroit).

Durant la construction vous allez voir défiler toutes les étapes s'exécuter les une après les autres. Le premier lancement est long, car aucune des étapes n'a été déjà appelée. Lors d'un second *build*, vous verrez que les étapes sont expédiées à grande vitesse ! (Essayez, pour voir !). Tout ça à condition que les étapes précédentes n'aient pas été modifiées dans votre Dockerfile, bien sûr !

Évidemment, Docker historise toutes les actions qu'il réalise (chaque instruction réalisée dans un Dockerfile est stockée dans un container intermédiaire). C'est également ce principe qui est utilisé pour réaliser des héritages entre images. Si les instructions sont inchangées et que leur ordre est identique à la précédente fois, elles ne sont PAS régénérées car **Docker** sait utiliser son cache et ré-utiliser les actions déjà réalisées. Fortiche, la baleine !

Explication : comment fonctionne le cache Docker ?

Le cache que « *docker build »* utilise n'est pas magique (même si ça en a l'air), mais il est facile à comprendre.

Plus haut, je vous ai vaguement parlé de la commande *docker-commit* qui vous permet de gérer des états de vos images en sauvegardant les changements d'un container dans une nouvelle image. Par défaut, quand vous faites un *docker-build*,

chaque étape (chaque instruction de votre **Dockerfile**) est stocké dans un container intermédiaire (vous pouvez voir toutes les étapes de la construction d'une image en utilisant la commande *docker-history*). Docker possède une toutouille interne permettant de manager ses containers internes de manière à proposer les meilleurs performances lors de la construction d'une image. Héritage, historisation, vous vous souvenez ;) ?

Tagger une image ou sortir vos images de l'anonymat

Toute image ou tout container possède un identifiant unique. Nous pourrions utiliser ces identifiants unique, mais **Docker Inc** a prévu une fonctionnalité pour *tagger* nos images ainsi que nos containers afin de les réutiliser plus facilement.

Voici comment nommer vos images :

1. Donner un nom à la construction de l'image avec l'option **--tag**
2. Utiliser la commande *docker-tag*
3. Nommer votre image au moment de l'utilisation de *docker-commit*

Exemple plus concret : création d'une image grâce à la commande docker-build et de son option **--tag**

```
$ docker build ––tag="myImage[:myTag]"
```

Et voilà, faites désormais un *docker-images* et vous votre image correctement taggée apparaîtra dans votre liste !

Le point technique !

Les images peuvent avoir autant de noms et autant de tags que vous voulez. D'ailleurs l'image officielle de Debian Wheezy peut être trouvée sous le nom *debian:7*, *debian:latest* ou encore*debian:wheezy*. Dans ce cas, nous avons appliqué plusieurs tags sur un même nom d'image, mais peut-être voulez-vous appeler cette image avec un autre nom.

```
default                         debian              c90d655b99b2
debian                          7.8                 c90d655b99b2
debian                          latest              c90d655b99b2
debian                          wheezy              c90d655b99b2
debian                          7                   c90d655b99b2
```

Ici vous voyez que l'image identifié par c90d655b99b2 est également identifié par *default:debian*.

Gestion des containers

Maintenant que la gestion des images n'est plus un problème pour vous, parlons un peu des containers.

Définir le fonctionnement d'un container

Nous en avons déjà parlé au-dessus, un container incarne la notion d'*environnement*. Dans **Docker**, même les images sont des containers, mais la différence entre une image et un container est dûe aux comportements.

Les images sont statiques ; on n'écrit pas directement dans une image. On déclare un container À PARTIR d'une image. Ce container va vivre et contenir les différences réalisés par rapport à l'image de base. Vous suivez ?

Un container contient également d'autres éléments, il stocke des paramètres d'exécutions, comme par exemple le mappage de port, les dossiers à partager.

Le point technique !

Actuellement on applique des paramètres à un container comme le *forward* de port, le partage de dossier… Mais il n'est pas possible de changer à chaud ses paramètres. **Docker Inc.** travaille également sur la possibilité de changer à chaud ses variables. Actuellement, le seul moyen pour nous de changer les paramètres est de :

1. Enregistrer son container comme image (docker commit)
2. Lancer un container à partir de la nouvelle image avec l'intégralité des paramètres que vous souhaitez utiliser.

Faisons ensemble un exercice.

Déclarer un container

Nous avons normalement une image **debian** avec plusieurs *tags* sur notre machine grâce à l'image créée plus haut. Déclarons un container.

```
$ docker run debian:wheezy
```

Et voilà, vous avez exécuté un container depuis une image Docker ! Avec cela, ne pensez pas faire des choses dingues, mais c'est un début. Si vous avez l'impression que cette commande n'a rien fait, vous pouvez voir avec la commande "docker ps -l" que des choses se sont passées :

```
$ docker ps -l
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS                      PORTS               NAMES
9b4841d73b6e        debian:7            "/bin/bash"         3 seconds ago
Exited (0) 3 seconds ago                        furious_hopper
```

Ce retour indique qu'un container nommé "*furious_hopper*" identifié par l'ID 9b4841d73b6e, a lancé la commande */bin/bash* depuis une image *debian:7* et s'est terminé avec le code retour 0. (Notez que Docker sait nommer ses containers de façon originale ! Ici *furious_hopper* !)

Même si votre container est éteint (statut *Exited*), celui-ci **n'est pas supprimé, il est stocké** (c'est important pour la suite). Vous pouvez d'ailleurs afficher tous les containers (ceux en cours et ceux stoppés) en utilisant la commande "*docker ps -a"*.

Désormais, nous voulons interagir avec notre container. Pour cela, nous allons devoir utiliser quelques options de la commande "*docker run"*, que voici :

```
$ docker run --tty --interactive debian:7
```

Vous devriez être dans un magnifique container avec un *bash* prêt à répondre. Vous remarquerez le temps de lancement est assez faible. Mais qu'avons-nous fait ? Détaillons ce qu'elle veut dire :

- L'option **--tty** permet d'attacher la console à notre console actuelle et de ne pas perdre le focus. C'est grâce à cette option que votre *container* ne va pas se terminer.
- L'option **--interactive** vous permet de dialoguer avec votre container. Sans cette option, tout ce que vous taperez dans votre console ne sera pas transmis au **bash** du *container*.

Dans notre cas, la commande lancée a été */bin/bash* mais il ne s'agit pas d'un hasard. Lors de la déclaration d'une image (dans son **Dockerfile**), vous pouvez spécifier certaines **metadatas** commeCMD qui vous permettent de donner une commande par défaut si aucune n'est spécifiée.

Si vous aviez voulu lancer une autre commande que */bin/bash*, vous auriez pu faire comme ceci au moment de la déclaration de votre container :

```
$ docker run debian:7 echo Docker is fun
```

Et voilà, votre container a été créé et a lancé la commande que vous avez donné ("*echo Docker is fun"*), puis s'est terminé car aucune commande exécutée n'a pris le focus de la console.

Utiliser un container

Maintenant que vous savez lancer des commandes dans un container, allons un peu plus loin avec un exemple simple.

Un container est l'équivalent d'un **Filesystem**. Aussi, mon projet **Symfony** utilise une base de données. Puisque ma base de données va changer, nous ne voulons pas perdre nos données (enfin, j'imagine ;) !). Ce serait le cas si vous faisiez un "*docker run"* à chaque fois que vouliez lancer votre service de base de données. Pourquoi ? Car comme vu plus haut, le "*docker run"* lance un nouveau *container*. Un nouveau ! Il n'utilise pas celui qui avait déjà été créé auparavant ! Voyons comment faire pour ne pas avoir ce problème.

*Détaillons le fonctionnement de "*docker run*"*

Au début, c'était LA (!) commande que j'utilisais tout le temps. Maintenant plus vraiment. Que fait vraiment le "*docker run" ?* Cette commande exécute successivement deux autres commandes auxquelles vous avez accès : "*docker create"* et "*docker start"*. Détaillons les ensemble.

docker-create

"docker create" est une commande **indispensable** ! Si vous faites un *man*, vous remarquerez que celle-ci contient les mêmes options que "*docker run"*. Avec cette commande, vous allez créer un container à partir d'une image en lui donnant des paramètres d'exécution.

- mappage des ports
- partage des dossiers
- …

Parmi les données paramétrables, vous allez pouvoir donner un nom à votre container. Docker donnera toujours un nom à votre container (*furious_hopper*, vous vous souvenez ?), mais il vous sera plus simple de retrouver le *container* de votre projet avec un nom adéquat plutôt que "condescending_wozniak" ! XD

Voici la commande complète pour créer un container, celle que vous utiliserez désormais !

```
$ docker create --tty --interactive --name="wonderful_mobidock" debian:7
```

Le retour de cette commande devrait être une suite indigeste de caractères (indiquant l'ID donné à votre container).

docker-start

Notre container est désormais créé grâce à la commande "*docker create*". D'ailleurs en faisant un "*docker ps -a"* vous devriez le trouver.

Maintenant que vous avez créé votre container et que vous l'avez configuré, vous pouvez le lancer avec la commande "*docker start*".

```
$ docker start wonderful_mobidock
```

Par défaut, "*docker start"* ne vous attache pas la console, mais vous pouvez le spécifier avec l'option **--attach**.

```
$ docker start --attach wonderful_mobido
```

Voilà qui est simple non ?

Ré-utiliser un container

Notre container est lancé et nommé. Notre base de données vit dans notre container, et d'une fois sur l'autre nous ne voulons pas perdre nos données (ce qui serait dommageable avouons-le). La commande "*docker stop"* permet d'éteindre proprement un container (l'utilisation de "*docker kill"* doit être limitée).

Si plus tard, vous voulez relancer ce *wonderful_mobydock* et récupérer vos données, exécutez "*docker start" !*

Conclusion

Désormais, vous êtes capable de créer une image et de gérer vos containers simplement en les nommant, les créant et en les arrêtant. L'intégralité des commandes et des options n'ont pas été détaillées ici mais vous devriez désormais disposer des bonnes pratiques pour commencer à utiliser**Docker**. Le manuel est votre ami, et **Docker** grandit vite, tenez vous au courant !