eBook Gratuit

APPRENEZ Docker

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#docker

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: docker

It is an unofficial and free Docker ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Docker.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec Docker

Remarques

Docker est un projet open-source qui automatise le déploiement d'applications dans des conteneurs logiciels. Ces conteneurs d'applications sont similaires aux machines virtuelles légères, car elles peuvent être exécutées isolément les unes des autres et avec l'hôte en cours d'exécution.

Docker exige que les fonctionnalités présentes dans les noyaux Linux récents fonctionnent correctement. Par conséquent, sur Mac OSX et Windows, une machine virtuelle exécutant Linux est requise pour que Docker fonctionne correctement. Actuellement, la principale méthode d'installation et de configuration de cette machine virtuelle est via Docker Toolbox qui utilise VirtualBox en interne, mais il est prévu d'intégrer cette fonctionnalité dans docker même, en utilisant les fonctionnalités de virtualisation natives du système d'exploitation. Sur les systèmes Linux, le docker est exécuté nativement sur l'hôte lui-même.

Versions

Version	Date de sortie
17.05.0	2017-05-04
17.04.0	2017-04-05
17.03.0	2017-03-01
1.13.1	2016-02-08
1.12.0	2016-07-28
1.11.2	2016-04-13
1.10.3	2016-02-04
1.9.1	2015-11-03
1.8.3	2015-08-11
1.7.1	2015-06-16
1.6.2	2015-04-07
1.5.0	2015-02-10

Examples

Installation de Docker sur Mac OS X

Exigences: OS X 10.8 «Mountain Lion» ou plus récent requis pour exécuter Docker.

Bien que le binaire docker puisse s'exécuter en mode natif sur Mac OS X, pour créer et héberger des conteneurs, vous devez exécuter une machine virtuelle Linux sur la boîte.

1.12.0

Depuis la version 1.12, vous n'avez pas besoin d'installer une machine virtuelle distincte, car Docker peut utiliser la fonctionnalité native d' `Hypervisor.framework` d'OSX pour démarrer une petite machine Linux agissant comme serveur principal.

Pour installer le menu fixe, procédez comme suit:

1. Aller à Docker pour Mac
2. Téléchargez et exécutez le programme d'installation.
3. Continuez à travers l'installateur avec les options par défaut et entrez vos identifiants de compte lorsque cela est demandé.

Vérifiez ici pour plus d'informations sur l'installation.

1.11.2

Jusqu'à la version 1.11, le meilleur moyen d'exécuter cette machine virtuelle Linux est d'installer Docker Toolbox, qui installe Docker, VirtualBox et l'ordinateur invité Linux.

Pour installer la boîte à outils de docker, procédez comme suit:

1. Aller à la boîte à outils Docker
2. Cliquez sur le lien pour Mac et exécutez le programme d'installation.
3. Continuez à travers l'installateur avec les options par défaut et entrez vos identifiants de compte lorsque cela est demandé.

Cela installera les fichiers binaires Docker dans `/usr/local/bin` et mettra à jour toute installation Virtual Box existante. Vérifiez ici pour plus d'informations sur l'installation.

Pour vérifier l'installation:

1.12.0

1. Démarrez `Docker.app` partir du dossier Applications et assurez-vous qu'il est en cours d'exécution. Ensuite, ouvrez Terminal.

1.11.2

1. Ouvrez le `Docker Quickstart Terminal` , qui ouvrira un terminal et le préparera pour les commandes Docker.

2. Une fois que le terminal est de type ouvert

```
$ docker run hello-world
```

3. Si tout va bien, cela devrait imprimer un message de bienvenue pour vérifier que l'installation a réussi.

Installation de Docker sous Windows

Configuration requise: Version 64 bits de Windows 7 ou supérieure sur une machine prenant en charge la technologie de virtualisation matérielle et activée.

Alors que le binaire docker peut s'exécuter en mode natif sous Windows, pour créer et héberger des conteneurs, vous devez exécuter une machine virtuelle Linux sur la boîte.

1.12.0

Depuis la version 1.12, vous n'avez pas besoin d'installer une machine virtuelle distincte, car Docker peut utiliser la fonctionnalité native Hyper-V de Windows pour démarrer une petite machine Linux en tant que serveur principal.

Pour installer le menu fixe, procédez comme suit:

1. Aller à Docker pour Windows
2. Téléchargez et exécutez le programme d'installation.
3. Continuez à travers l'installateur avec les options par défaut et entrez vos identifiants de compte lorsque cela est demandé.

Vérifiez ici pour plus d'informations sur l'installation.

1.11.2

Jusqu'à la version 1.11, le meilleur moyen d'exécuter cette machine virtuelle Linux est d'installer Docker Toolbox, qui installe Docker, VirtualBox et l'ordinateur invité Linux.

Pour installer la boîte à outils de docker, procédez comme suit:

1. Aller à la boîte à outils Docker
2. Cliquez sur le lien pour Windows et exécutez le programme d'installation.
3. Continuez à travers l'installateur avec les options par défaut et entrez vos identifiants de compte lorsque cela est demandé.

Cela installera les fichiers binaires Docker dans Program Files et mettra à jour toute installation Virtual Box existante. Vérifiez ici pour plus d'informations sur l'installation.

Pour vérifier l'installation:

1.12.0

1. Démarrez `Docker` partir du menu Démarrer s'il n'a pas encore été démarré et assurez-vous

qu'il est en cours d'exécution. Ensuite, montez n'importe quel terminal (`cmd` ou PowerShell)

1.11.2

1. Sur votre bureau, recherchez l'icône Docker Toolbox. Cliquez sur l'icône pour lancer un terminal Docker Toolbox.

2. Une fois que le terminal est de type ouvert

```
docker run hello-world
```

3. Si tout va bien, cela devrait imprimer un message de bienvenue pour vérifier que l'installation a réussi.

Installation de docker sur Ubuntu Linux

Docker est pris en charge sur les versions *64 bits* d'Ubuntu Linux suivantes:

- Ubuntu Xenial 16.04 (LTS)
- Ubuntu Wily 15.10
- Ubuntu Trusty 14.04 (LTS)
- Ubuntu Precise 12.04 (LTS)

Quelques notes:

> Les instructions suivantes impliquent l'installation à l'aide de packages **Docker** uniquement, ce qui garantit l'obtention de la dernière version officielle de **Docker** . Si vous avez besoin d'installer uniquement avec `Ubuntu-managed` packages `Ubuntu-managed` par `Ubuntu-managed` , consultez la documentation Ubuntu (non recommandé pour des raisons évidentes).
>
> Ubuntu Utopic 14.10 et 15.04 existent dans le référentiel APT de Docker mais ne sont plus officiellement pris en charge en raison de problèmes de sécurité connus.

Conditions préalables

- Docker ne fonctionne que sur une installation 64 bits de Linux.
- Docker nécessite la version 3.10 ou supérieure du noyau Linux (sauf pour `Ubuntu Precise 12.04` , qui nécessite la version 3.13 ou supérieure). Les noyaux antérieurs à 3.10 ne possèdent pas certaines des fonctionnalités requises pour exécuter les conteneurs Docker et contiennent des bogues connus qui entraînent une perte de données et paniquent souvent sous certaines conditions. Vérifiez la version actuelle du noyau avec la commande `uname -r` . Vérifiez ce post si vous avez besoin de mettre à jour votre noyau `Ubuntu Precise (12.04 LTS)` faisant défiler plus bas. Reportez-vous à ce post WikiHow pour obtenir la dernière version pour d'autres installations Ubuntu.

Mettre à jour les sources APT

Cela doit être fait pour accéder aux packages du référentiel Docker.

1. Connectez-vous à votre ordinateur en tant qu'utilisateur disposant `root` privilèges `sudo` ou `root` .
2. Ouvrez une fenêtre de terminal.
3. Mettez à jour les informations sur le package, assurez-vous que APT fonctionne avec la méthode https et que les certificats de l'autorité de certification sont installés.

```
$ sudo apt-get update
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
```

4. Ajoutez la clé GPG officielle de Docker:

   ```
   $ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
   ```

   Vérifiez que l'empreinte de la clé est **9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88** .

   ```
   $ sudo apt-key fingerprint 0EBFCD88
   ```

```
      pub   4096R/0EBFCD88 2017-02-22
            Key fingerprint = 9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
      uid                  Docker Release (CE deb) <docker@docker.com>
      sub   4096R/F273FCD8 2017-02-22
```

5. Recherchez l'entrée dans le tableau ci-dessous qui correspond à votre version d'Ubuntu. Ceci détermine où APT recherchera les paquets Docker. Si possible, exécutez une édition du support à long terme (LTS) d'Ubuntu.

Version Ubuntu	Dépôt
Précis 12.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-precise main`
Trusty 14.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-trusty main`
Wily 15.10	`deb https://apt.dockerproject.org/repo ubuntu-wily main`
Xenial 16.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-xenial main`

   Remarque: Docker ne fournit pas de packages pour toutes les architectures. Les artefacts binaires sont construits chaque nuit et vous pouvez les télécharger depuis `https://master.dockerproject.org` . Pour installer docker sur un système multi-architecture, ajoutez une clause `[arch=...]` à l'entrée. Reportez-vous au wiki Debian Multiarch pour plus de détails.

6. Exécutez la commande suivante en remplaçant l'entrée de votre système d'exploitation par l'espace réservé `<REPO>` .

$ echo "" | sudo tee /etc/apt/sources.list.d/docker.list

7. Mettez à jour l'index du package `APT` en exécutant `sudo apt-get update` .

8. Vérifiez que l' `APT` tire du bon référentiel.

Lorsque vous exécutez la commande suivante, une entrée est renvoyée pour chaque version de Docker que vous pouvez installer. Chaque entrée doit avoir l'URL `https://apt.dockerproject.org/repo/` . La version actuellement installée est marquée avec `***` Voir la sortie de l'exemple ci-dessous.

```
$ apt-cache policy docker-engine

  docker-engine:
    Installed: 1.12.2-0~trusty
    Candidate: 1.12.2-0~trusty
    Version table:
   *** 1.12.2-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
          100 /var/lib/dpkg/status
       1.12.1-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
       1.12.0-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
```

A partir de maintenant, lorsque vous exécutez la `apt-get upgrade` , `APT` extrait le nouveau référentiel.

Prérequis par la version Ubuntu

Pour Ubuntu Trusty (14.04), Wily (15.10) et Xenial (16.04), installez les paquets de noyau `linux-image-extra-*` , qui vous permettent d'utiliser le pilote de stockage d' `aufs` .

Pour installer les paquets `linux-image-extra-*` :

1. Ouvrez un terminal sur votre hôte Ubuntu.

2. Mettez à jour votre gestionnaire de paquets avec la commande `sudo apt-get update` .

3. Installez les paquets recommandés.

```
$ sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual
```

4. Procéder à l'installation de Docker

Pour Ubuntu Precise (12.04 LTS), Docker nécessite la version du noyau 3.13. Si votre version du noyau est antérieure à la version 3.13, vous devez la mettre à niveau. Reportez-vous à ce tableau pour voir quels packages sont requis pour votre environnement:

Paquet	La description
`linux-image-generic-lts-`	Image du noyau Linux générique. Ce noyau a `AUFS` intégré. Ceci est

Paquet	La description
`trusty`	nécessaire pour exécuter Docker.
`linux-headers-generic-lts-trusty`	Permet aux paquets tels que `VirtualBox guest additions ZFS` et `VirtualBox guest additions` qui en dépendent. Si vous n'avez pas installé les en-têtes pour votre noyau existant, vous pouvez ignorer ces en-têtes pour le noyau `trusty` . Si vous n'êtes pas sûr, vous devriez inclure ce paquet pour la sécurité.
`xserver-xorg-lts-trusty`	Facultatif dans les environnements non graphiques sans Unity / Xorg. **Requis** lors de l'exécution de Docker sur une machine avec un environnement graphique.
`ligbl1-mesa-glx-lts-trusty`	Pour en savoir plus sur les raisons de ces packages, lisez les instructions d'installation pour les noyaux backportés, en particulier la pile d'activation LTS . Voir la note 5 sous chaque version.

Pour mettre à niveau votre noyau et installer les packages supplémentaires, procédez comme suit:

1. Ouvrez un terminal sur votre hôte Ubuntu.
2. Mettez à jour votre gestionnaire de paquets avec la commande `sudo apt-get update` .
3. Installez les packages requis et facultatifs.

```
$ sudo apt-get install linux-image-generic-lts-trusty
```

4. Répétez cette étape pour les autres packages à installer.
5. Redémarrez votre hôte pour utiliser le noyau mis à jour à l'aide de la commande `sudo reboot` .
6. Après le redémarrage, installez Docker.

Installer la dernière version

Assurez-vous de satisfaire aux conditions préalables, puis suivez les étapes ci-dessous.

> **Remarque:** Pour les systèmes de production, il est recommandé d' installer une version spécifique afin de ne pas mettre accidentellement Docker à jour. Vous devez planifier les mises à niveau des systèmes de production avec soin.

1. Connectez-vous à votre installation Ubuntu en tant qu'utilisateur disposant des privilèges `sudo` . (Peut-être exécuter `sudo -su`).
2. Mettez à jour votre index de package APT en exécutant `sudo apt-get update` .
3. Installez Docker Community Edition avec la commande `sudo apt-get install docker-ce` .

4. Démarrez le démon `docker` avec la commande `sudo service docker start` .

5. Vérifiez que le `docker` est installé correctement en exécutant l'image hello-world.

```
$ sudo docker run hello-world
```

Cette commande télécharge une image de test et l'exécute dans un conteneur. Lorsque le conteneur est exécuté, il imprime un message d'information et quitte.

Gérer Docker en tant qu'utilisateur non root

Si vous ne souhaitez pas utiliser `sudo` lorsque vous utilisez la commande docker, créez un groupe Unix appelé `docker` et ajoutez-y des utilisateurs. Lorsque le démon `docker` démarre, le groupe `docker` devient propriétaire du socket Unix en lecture / écriture.

Pour créer le groupe de `docker` et ajouter votre utilisateur:

1. Connectez-vous à Ubuntu en tant qu'utilisateur disposant des privilèges `sudo` .

2. Créer le `docker` groupe avec la commande `sudo groupadd docker` .

3. Ajoutez votre utilisateur au groupe de `docker` .

```
$ sudo usermod -aG docker $USER
```

4. Déconnectez-vous et reconnectez-vous pour que votre appartenance à un groupe soit réévaluée.

5. Vérifiez que vous pouvez `docker` commandes sans autorisation `sudo` .

```
$ docker run hello-world
```

Si cela échoue, vous verrez une erreur:

```
Cannot connect to the Docker daemon. Is 'docker daemon' running on this host?
```

Vérifiez si la variable d'environnement `DOCKER_HOST` est définie pour votre shell.

```
$ env | grep DOCKER_HOST
```

Si elle est définie, la commande ci-dessus renverra un résultat. Si c'est le cas, désactivez-le.

```
$ unset DOCKER_HOST
```

Vous devrez peut-être modifier votre environnement dans des fichiers tels que `~/.bashrc` ou `~/.profile` pour empêcher que la variable `DOCKER_HOST` ne soit définie de manière erronée.

Installer Docker sur Ubuntu

Exigences: Docker peut être installé sur n'importe quel Linux avec un noyau d'au moins la version 3.10. Docker est pris en charge sur les versions 64 bits d'Ubuntu Linux suivantes:

- Ubuntu Xenial 16.04 (LTS)
- Ubuntu Wily 15.10
- Ubuntu Trusty 14.04 (LTS)
- Ubuntu Precise 12.04 (LTS)

Installation facile

Remarque: L'installation de Docker à partir du référentiel Ubuntu par défaut installera une ancienne version de Docker.

Pour installer la dernière version de Docker à l'aide du référentiel Docker, utilisez `curl` pour récupérer et exécuter le script d'installation fourni par Docker:

```
$ curl -sSL https://get.docker.com/ | sh
```

Vous pouvez également utiliser `wget` pour installer Docker:

```
$ wget -qO- https://get.docker.com/ | sh
```

Docker va maintenant être installé.

Installation manuelle

Si, toutefois, l'exécution du script d'installation n'est pas une option, les instructions suivantes peuvent être utilisées pour installer manuellement la dernière version de Docker à partir du référentiel officiel.

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates
```

Ajoutez la clé GPG:

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 \
  --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

Ensuite, ouvrez le fichier `/etc/apt/sources.list.d/docker.list` dans votre éditeur préféré. Si le fichier n'existe pas, créez-le. Supprimez toutes les entrées existantes. Ensuite, en fonction de votre version, ajoutez la ligne suivante:

- Ubuntu Precise 12.04 (LTS):

  ```
  deb https://apt.dockerproject.org/repo ubuntu-precise main
  ```

- Ubuntu Trusty 14.04 (LTS)

  ```
  deb https://apt.dockerproject.org/repo ubuntu-trusty main
  ```

- Ubuntu Wily 15.10

  ```
  deb https://apt.dockerproject.org/repo ubuntu-wily main
  ```

- Ubuntu Xenial 16.04 (LTS)

  ```
  deb https://apt.dockerproject.org/repo ubuntu-xenial main
  ```

Enregistrez le fichier et quittez-le, puis mettez à jour l'index de votre package, désinstallez les versions installées de Docker et vérifiez que `apt` tire du bon référentiel:

```
$ sudo apt-get update
$ sudo apt-get purge lxc-docker
$ sudo apt-cache policy docker-engine
```

Selon votre version d'Ubuntu, certaines conditions préalables peuvent être requises:

- Ubuntu Xenial 16.04 (LTS), Ubuntu Wily 15.10, Ubuntu Trusty 14.04 (LTS)

  ```
  sudo apt-get update && sudo apt-get install linux-image-extra-$(uname -r)
  ```

- Ubuntu Precise 12.04 (LTS)

  Cette version d'Ubuntu nécessite la version 3.13 du noyau. Vous devrez peut-être installer des packages supplémentaires en fonction de votre environnement:

  ```
  linux-image-generic-lts-trusty
  ```

  Image du noyau Linux générique. Ce noyau a AUFS intégré. Ceci est nécessaire pour exécuter Docker.

  ```
  linux-headers-generic-lts-trusty
  ```

  Permet aux paquets tels que les ajouts d'invités ZFS et VirtualBox qui en dépendent. Si vous n'avez pas installé les en-têtes pour votre noyau existant, vous pouvez ignorer ces en-têtes pour le noyau `trusty` . Si vous n'êtes pas sûr, vous devriez inclure ce paquet pour la sécurité.

  ```
  xserver-xorg-lts-trusty
  ```

  ```
  libgl1-mesa-glx-lts-trusty
  ```

  Ces deux packages sont facultatifs dans les environnements non graphiques sans Unity / Xorg. Requis lors de l'exécution de Docker sur une machine avec un environnement graphique.

  Pour en savoir plus sur les raisons de ces packages, lisez les instructions d'installation pour les noyaux backportés, en particulier la pile d'activation LTS - reportez-vous à la note 5 de chaque version.

  Installez les packages requis, puis redémarrez l'hôte:

  ```
  $ sudo apt-get install linux-image-generic-lts-trusty
  ```

  ```
  $ sudo reboot
  ```

Enfin, mettez à jour l'index du paquet `apt` et installez Docker:

```
$ sudo apt-get update
$ sudo apt-get install docker-engine
```

Démarrez le démon:

```
$ sudo service docker start
```

Maintenant, vérifiez que Docker fonctionne correctement en démarrant une image de test:

```
$ sudo docker run hello-world
```

Cette commande devrait imprimer un message de bienvenue vérifiant que l'installation a réussi.

Créer un conteneur de docker dans Google Cloud

Vous pouvez utiliser docker, sans utiliser le démon docker (moteur), en utilisant des fournisseurs de cloud. Dans cet exemple, vous devriez avoir un `gcloud` (Google Cloud util) connecté à votre compte

```
docker-machine create --driver google --google-project `your-project-name` google-machine-type
f1-large fm02
```

Cet exemple créera une nouvelle instance dans votre console Google Cloud. En utilisant le temps machine `f1-large`

Installer Docker sur Ubuntu

Docker est pris en charge sur les versions *64 bits* d'Ubuntu Linux suivantes:

- Ubuntu Xenial 16.04 (LTS)
- Ubuntu Wily 15.10
- Ubuntu Trusty 14.04 (LTS)
- Ubuntu Precise 12.04 (LTS)

Quelques notes:

> Les instructions suivantes impliquent l'installation à l'aide de packages **Docker** uniquement, ce qui garantit l'obtention de la dernière version officielle de **Docker** . Si vous avez besoin d'installer uniquement avec `Ubuntu-managed` packages `Ubuntu-managed` par `Ubuntu-managed` , consultez la documentation Ubuntu (non recommandé pour des raisons évidentes).
>
> Ubuntu Utopic 14.10 et 15.04 existent dans le référentiel APT de Docker mais ne sont plus officiellement pris en charge en raison de problèmes de sécurité connus.

Conditions préalables

- Docker ne fonctionne que sur une installation 64 bits de Linux.
- Docker nécessite la version 3.10 ou supérieure du noyau Linux (sauf pour `Ubuntu Precise 12.04` , qui nécessite la version 3.13 ou supérieure). Les noyaux antérieurs à 3.10 ne possèdent pas certaines des fonctionnalités requises pour exécuter les conteneurs Docker et contiennent des bogues connus qui entraînent une perte de données et paniquent souvent sous certaines conditions. Vérifiez la version actuelle du noyau avec la commande `uname -r` . Vérifiez ce post si vous avez besoin de mettre à jour votre noyau `Ubuntu Precise (12.04 LTS)` faisant défiler plus bas. Reportez-vous à ce post WikiHow pour obtenir la dernière version pour d'autres installations Ubuntu.

Mettre à jour les sources APT

Cela doit être fait pour accéder aux packages du référentiel Docker.

1. Connectez-vous à votre ordinateur en tant qu'utilisateur disposant `root` privilèges `sudo` ou `root` .
2. Ouvrez une fenêtre de terminal.
3. Mettez à jour les informations sur le package, assurez-vous que APT fonctionne avec la méthode https et que les certificats de l'autorité de certification sont installés.

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates
```

4. Ajoutez la nouvelle clé `GPG` . Cette commande télécharge la clé avec l'ID `58118E89F3A912897C070ADBF76221572C52609D` partir du serveur de `hkp://ha.pool.sks-keyservers.net:80` et l'ajoute au `adv keychain` . Pour plus d'informations, voir la sortie de `man apt-key` .

   ```
   $ sudo apt-key adv \
          --keyserver hkp://ha.pool.sks-keyservers.net:80 \
          --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
   ```

5. Recherchez l'entrée dans le tableau ci-dessous qui correspond à votre version d'Ubuntu. Ceci détermine où APT recherchera les paquets Docker. Si possible, exécutez une édition du support à long terme (LTS) d'Ubuntu.

Version Ubuntu	Dépôt
Précis 12.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-precise main`
Trusty 14.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-trusty main`
Wily 15.10	`deb https://apt.dockerproject.org/repo ubuntu-wily main`
Xenial 16.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-xenial main`

   Remarque: Docker ne fournit pas de packages pour toutes les architectures. Les artefacts binaires sont construits chaque nuit et vous pouvez les télécharger depuis

`https://master.dockerproject.org` . Pour installer docker sur un système multi-architecture, ajoutez une clause `[arch=...]` à l'entrée. Reportez-vous au wiki Debian Multiarch pour plus de détails.

6. Exécutez la commande suivante en remplaçant l'entrée de votre système d'exploitation par l'espace réservé `<REPO>` .

   $ echo "" | sudo tee /etc/apt/sources.list.d/docker.list

7. Mettez à jour l'index du package `APT` en exécutant `sudo apt-get update` .

8. Vérifiez que l' `APT` tire du bon référentiel.

Lorsque vous exécutez la commande suivante, une entrée est renvoyée pour chaque version de Docker que vous pouvez installer. Chaque entrée doit avoir l'URL `https://apt.dockerproject.org/repo/` . La version actuellement installée est marquée avec `***` Voir la sortie de l'exemple ci-dessous.

```
$ apt-cache policy docker-engine

  docker-engine:
    Installed: 1.12.2-0~trusty
    Candidate: 1.12.2-0~trusty
    Version table:
   *** 1.12.2-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
          100 /var/lib/dpkg/status
       1.12.1-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
       1.12.0-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
```

A partir de maintenant, lorsque vous exécutez la `apt-get upgrade` , `APT` extrait le nouveau référentiel.

Prérequis par la version Ubuntu

Pour Ubuntu Trusty (14.04), Wily (15.10) et Xenial (16.04), installez les paquets de noyau `linux-image-extra-*` , qui vous permettent d'utiliser le pilote de stockage d' `aufs` .

Pour installer les paquets `linux-image-extra-*` :

1. Ouvrez un terminal sur votre hôte Ubuntu.

2. Mettez à jour votre gestionnaire de paquets avec la commande `sudo apt-get update` .

3. Installez les paquets recommandés.

   ```
   $ sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual
   ```

4. Procéder à l'installation de Docker

Pour Ubuntu Precise (12.04 LTS), Docker nécessite la version du noyau 3.13. Si votre version du noyau est antérieure à la version 3.13, vous devez la mettre à niveau. Reportez-vous à ce tableau pour voir quels packages sont requis pour votre environnement:

Paquet	La description
`linux-image-generic-lts-trusty`	Image du noyau Linux générique. Ce noyau a `AUFS` intégré. Ceci est nécessaire pour exécuter Docker.
`linux-headers-generic-lts-trusty`	Permet aux paquets tels que `VirtualBox guest additions ZFS` et `VirtualBox guest additions` qui en dépendent. Si vous n'avez pas installé les en-têtes pour votre noyau existant, vous pouvez ignorer ces en-têtes pour le noyau `trusty` . Si vous n'êtes pas sûr, vous devriez inclure ce paquet pour la sécurité.
`xserver-xorg-lts-trusty`	Facultatif dans les environnements non graphiques sans Unity / Xorg. **Requis** lors de l'exécution de Docker sur une machine avec un environnement graphique.
`libgl1-mesa-glx-lts-trusty`	Pour en savoir plus sur les raisons de ces packages, lisez les instructions d'installation pour les noyaux backportés, en particulier la pile d'activation LTS . Voir la note 5 sous chaque version.

Pour mettre à niveau votre noyau et installer les packages supplémentaires, procédez comme suit:

1. Ouvrez un terminal sur votre hôte Ubuntu.
2. Mettez à jour votre gestionnaire de paquets avec la commande `sudo apt-get update` .
3. Installez les packages requis et facultatifs.

```
$ sudo apt-get install linux-image-generic-lts-trusty
```

4. Répétez cette étape pour les autres packages à installer.
5. Redémarrez votre hôte pour utiliser le noyau mis à jour à l'aide de la commande `sudo reboot` .
6. Après le redémarrage, installez Docker.

Installer la dernière version

Assurez-vous de satisfaire aux conditions préalables, puis suivez les étapes ci-dessous.

> **Remarque:** Pour les systèmes de production, il est recommandé d' installer une version spécifique afin de ne pas mettre accidentellement Docker à jour. Vous devez planifier les mises à niveau des systèmes de production avec soin.

1. Connectez-vous à votre installation Ubuntu en tant qu'utilisateur disposant des privilèges `sudo` . (Peut-être exécuter `sudo -su`).
2. Mettez à jour votre index de package APT en exécutant `sudo apt-get update` .
3. Installez Docker avec la commande `sudo apt-get install docker-engine` .
4. Démarrez le démon `docker` avec la commande `sudo service docker start` .
5. Vérifiez que le `docker` est installé correctement en exécutant l'image hello-world.

```
$ sudo docker run hello-world
```

Cette commande télécharge une image de test et l'exécute dans un conteneur. Lorsque le conteneur est exécuté, il imprime un message d'information et quitte.

Gérer Docker en tant qu'utilisateur non root

Si vous ne souhaitez pas utiliser `sudo` lorsque vous utilisez la commande docker, créez un groupe Unix appelé `docker` et ajoutez-y des utilisateurs. Lorsque le démon `docker` démarre, le groupe `docker` devient propriétaire du socket Unix en lecture / écriture.

Pour créer le groupe de `docker` et ajouter votre utilisateur:

1. Connectez-vous à Ubuntu en tant qu'utilisateur disposant des privilèges `sudo` .
2. Créer le `docker` groupe avec la commande `sudo groupadd docker` .
3. Ajoutez votre utilisateur au groupe de `docker` .

```
$ sudo usermod -aG docker $USER
```

4. Déconnectez-vous et reconnectez-vous pour que votre appartenance à un groupe soit réévaluée.
5. Vérifiez que vous pouvez `docker` commandes sans autorisation `sudo` .

```
$ docker run hello-world
```

Si cela échoue, vous verrez une erreur:

```
Cannot connect to the Docker daemon. Is 'docker daemon' running on this host?
```

Vérifiez si la variable d'environnement `DOCKER_HOST` est définie pour votre shell.

```
$ env | grep DOCKER_HOST
```

Si elle est définie, la commande ci-dessus renverra un résultat. Si c'est le cas, désactivez-le.

```
$ unset DOCKER_HOST
```

Vous devrez peut-être modifier votre environnement dans des fichiers tels que `~/.bashrc` ou `~/.profile` pour empêcher que la variable `DOCKER_HOST` ne soit définie de manière erronée.

Installation de Docker-ce OU Docker-ee sur CentOS

Docker a annoncé les éditions suivantes:

-Docker-ee (Enterprise Edition) avec Docker-ce (Community Edition) et Docker (Support commercial)

Ce document vous aidera dans les étapes d'installation de l'édition Docker-ee et Docker-ce dans CentOS

Installation de docker-ce

Voici les étapes à suivre pour installer l'édition docker-ce

1. Installez yum-utils, qui fournit l'utilitaire yum-config-manager:

```
$ sudo yum install -y yum-utils
```

2. Utilisez la commande suivante pour configurer le référentiel stable:

```
$ sudo yum-config-manager \
 --add-repo \
 https://download.docker.com/linux/centos/docker-ce.repo
```

3. Facultatif: activez le référentiel Edge. Ce référentiel est inclus dans le fichier docker.repo ci-dessus mais est désactivé par défaut. Vous pouvez l'activer avec le référentiel stable.

```
 $ sudo yum-config-manager --enable docker-ce-edge
```

- Vous pouvez désactiver le référentiel `--disable` en exécutant la commande `yum-config-manager` avec l'indicateur `--disable` . Pour le réactiver, utilisez le drapeau `--enable` . La commande suivante désactive le référentiel Edge.

```
 $ sudo yum-config-manager --disable docker-ce-edge
```

4. Mettez à jour l'index du package yum.

```
 $ sudo yum makecache fast
```

5. Installez le docker-ce en utilisant la commande suivante:

```
 $ sudo yum install docker-ce-17.03.0.ce
```

6. Confirmez l'empreinte digitale Docker-ce

   ```
   060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35
   ```

   Si vous souhaitez installer une autre version de docker-ce, vous pouvez utiliser la commande suivante:

   ```
   $ sudo yum install docker-ce-VERSION
   ```

   Spécifiez la `VERSION` numéro

7. Si tout s'est bien passé, le docker-ce est maintenant installé sur votre système, utilisez la commande suivante pour démarrer:

   ```
   $ sudo systemctl start docker
   ```

8. Testez votre installation de docker:

   ```
   $ sudo docker run hello-world
   ```

   vous devriez recevoir le message suivant:

   ```
   Hello from Docker!
   This message shows that your installation appears to be working correctly.
   ```

-Docker-ee (Enterprise Edition) Installation

Pour Enterprise Edition (EE), vous devez vous inscrire pour obtenir votre <DOCKER-EE-URL>.

1. Pour vous inscrire, rendez-vous sur https://cloud.docker.com/ . Entrez vos coordonnées et confirmez votre identifiant de messagerie. Après confirmation, vous recevrez une <DOCKER-EE-URL>, que vous pouvez voir dans votre tableau de bord après avoir cliqué sur la configuration.

2. Supprimez tous les référentiels Docker existants de `/etc/yum.repos.d/`

3. Stockez l'URL de votre référentiel Docker EE dans une variable yum dans `/etc/yum/vars/` . Remplacez <DOCKER-EE-URL> par l'URL que vous avez notée lors de la première étape.

   ```
   $ sudo sh -c 'echo "<DOCKER-EE-URL>" > /etc/yum/vars/dockerurl'
   ```

4. Installez yum-utils, qui fournit l'utilitaire yum-config-manager:

   ```
   $ sudo yum install -y yum-utils
   ```

5. Utilisez la commande suivante pour ajouter le référentiel stable:

```
$ sudo yum-config-manager \
--add-repo \
<DOCKER-EE-URL>/docker-ee.repo
```

6. Mettez à jour l'index du package yum.

```
$ sudo yum makecache fast
```

7. Installez docker-ee

```
sudo yum install docker-ee
```

8. Vous pouvez démarrer le docker-ee en utilisant la commande suivante:

```
$ sudo systemctl start docker
```

Lire Démarrer avec Docker en ligne: https://riptutorial.com/fr/docker/topic/658/demarrer-avec-docker

Chapitre 2: API Docker Engine

Introduction

Une API qui vous permet de contrôler tous les aspects de Docker depuis vos propres applications, de créer des outils pour gérer et surveiller les applications exécutées sur Docker, et même de l'utiliser pour créer des applications sur Docker.

Examples

Activer l'accès à distance à l'API Docker sous Linux

Modifiez `/etc/init/docker.conf` et mettez à jour la variable `DOCKER_OPTS` comme suit:

```
DOCKER_OPTS='-H tcp://0.0.0.0:4243 -H unix:///var/run/docker.sock'
```

Redémarrer Docker Deamon

```
service docker restart
```

Vérifier si l'API distante fonctionne

```
curl -X GET http://localhost:4243/images/json
```

Activer l'accès à distance à l'API Docker sur Linux exécutant systemd

Linux fonctionnant avec systemd, comme Ubuntu 16.04, l'ajout de `-H tcp://0.0.0.0:2375` à `/etc/default/docker` n'a pas l'effet `-H tcp://0.0.0.0:2375` .

À la place, créez un fichier appelé `/etc/systemd/system/docker-tcp.socket` pour rendre docker disponible sur un socket TCP sur le port 4243:

```
[Unit]
Description=Docker Socket for the API
[Socket]
ListenStream=4243
Service=docker.service
[Install]
WantedBy=sockets.target
```

Activez ensuite le nouveau socket:

```
systemctl enable docker-tcp.socket
systemctl enable docker.socket
systemctl stop docker
systemctl start docker-tcp.socket
systemctl start docker
```

Maintenant, vérifiez si l'API distante fonctionne:

```
curl -X GET http://localhost:4243/images/json
```

Activer l'accès à distance avec TLS sur Systemd

Copiez le fichier d'unité du programme d'installation du package vers / etc où les modifications ne seront pas écrasées lors d'une mise à niveau:

```
cp /lib/systemd/system/docker.service /etc/systemd/system/docker.service
```

Mettez à jour /etc/systemd/system/docker.service avec vos options sur ExecStart:

```
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2376 \
  --tlsverify --tlscacert=/etc/docker/certs/ca.pem \
  --tlskey=/etc/docker/certs/key.pem \
  --tlscert=/etc/docker/certs/cert.pem
```

Notez que `dockerd` est le nom du démon 1.12, avant c'était le `docker daemon` . Notez également que 2376 est le port TLS standard de dockers, 2375 est le port non crypté standard. Reportez - vous à cette page pour connaître les étapes à suivre pour créer votre propre autorité de certification, certificat et clé d'authentification TLS.

Après avoir modifié les fichiers d'unité systemd, exécutez les opérations suivantes pour recharger la configuration systemd:

```
systemctl daemon-reload
```

Et puis exécutez ce qui suit pour redémarrer docker:

```
systemctl restart docker
```

Il est déconseillé d'ignorer le chiffrement TLS lors de l'exposition du port Docker, car toute personne disposant d'un accès réseau à ce port dispose d'un accès root complet sur l'hôte.

Tirer l'image avec des barres de progression, écrit en Go

Voici un exemple de tirage d'images à l'aide de l' `Docker Engine API Go` et `Docker Engine API` et des mêmes barres de progression que celles affichées lorsque vous exécutez `docker pull your_image_name` dans la `CLI` . Aux fins des barres de progression, certains codes ANSI sont utilisés.

```
package yourpackage

import (
    "context"
    "encoding/json"
    "fmt"
    "io"
    "strings"
```

```
    "github.com/docker/docker/api/types"
    "github.com/docker/docker/client"
)

// Struct representing events returned from image pulling
type pullEvent struct {
    ID             string `json:"id"`
    Status         string `json:"status"`
    Error          string `json:"error,omitempty"`
    Progress       string `json:"progress,omitempty"`
    ProgressDetail struct {
        Current int `json:"current"`
        Total   int `json:"total"`
    } `json:"progressDetail"`
}

// Actual image pulling function
func PullImage(dockerImageName string) bool {
    client, err := client.NewEnvClient()

    if err != nil {
        panic(err)
    }

    resp, err := client.ImagePull(context.Background(), dockerImageName,
types.ImagePullOptions{})

    if err != nil {
        panic(err)
    }

    cursor := Cursor{}
    layers := make([]string, 0)
    oldIndex := len(layers)

    var event *pullEvent
    decoder := json.NewDecoder(resp)

    fmt.Printf("\n")
    cursor.hide()

    for {
        if err := decoder.Decode(&event); err != nil {
            if err == io.EOF {
                break
            }

            panic(err)
        }

        imageID := event.ID

        // Check if the line is one of the final two ones
        if strings.HasPrefix(event.Status, "Digest:") || strings.HasPrefix(event.Status,
"Status:") {
            fmt.Printf("%s\n", event.Status)
            continue
        }

        // Check if ID has already passed once
```

```
        index := 0
        for i, v := range layers {
            if v == imageID {
                index = i + 1
                break
            }
        }

        // Move the cursor
        if index > 0 {
            diff := index - oldIndex

            if diff > 1 {
                down := diff - 1
                cursor.moveDown(down)
            } else if diff < 1 {
                up := diff*(-1) + 1
                cursor.moveUp(up)
            }

            oldIndex = index
        } else {
            layers = append(layers, event.ID)
            diff := len(layers) - oldIndex

            if diff > 1 {
                cursor.moveDown(diff) // Return to the last row
            }

            oldIndex = len(layers)
        }

        cursor.clearLine()

        if event.Status == "Pull complete" {
            fmt.Printf("%s: %s\n", event.ID, event.Status)
        } else {
            fmt.Printf("%s: %s %s\n", event.ID, event.Status, event.Progress)
        }

    }

    cursor.show()

    if strings.Contains(event.Status, fmt.Sprintf("Downloaded newer image for %s",
dockerImageName)) {
        return true
    }

    return false
}
```

Pour une meilleure lisibilité, les actions du curseur avec les codes ANSI sont déplacées vers une structure distincte, qui ressemble à ceci:

```
package yourpackage

import "fmt"

// Cursor structure that implements some methods
```

```
// for manipulating command line's cursor
type Cursor struct{}

func (cursor *Cursor) hide() {
    fmt.Printf("\033[?25l")
}

func (cursor *Cursor) show() {
    fmt.Printf("\033[?25h")
}

func (cursor *Cursor) moveUp(rows int) {
    fmt.Printf("\033[%dF", rows)
}

func (cursor *Cursor) moveDown(rows int) {
    fmt.Printf("\033[%dE", rows)
}

func (cursor *Cursor) clearLine() {
    fmt.Printf("\033[2K")
}
```

Après cela, dans votre paquet principal, vous pouvez appeler la fonction `PullImage` passant le nom de l'image que vous souhaitez extraire. Bien sûr, avant de l'appeler, vous devez être connecté au registre Docker, où se trouve l'image.

Faire une requête cURL en passant une structure complexe

Lors de l'utilisation de `cURL` pour certaines requêtes sur l' `Docker API` , il peut être difficile de faire passer certaines structures complexes. Disons que l' obtention d'une liste d'images permet d'utiliser des filtres comme paramètre de requête, qui doit être une représentation `JSON` de la `map[string][]string` (à propos des cartes dans `Go` vous pouvez en trouver plus ici).
Voici comment y parvenir:

```
curl --unix-socket /var/run/docker.sock \
    -XGET "http:/v1.29/images/json" \
    -G \
    --data-urlencode 'filters={"reference":{"yourpreciousregistry.com/path/to/image": true},
"dangling":{"true": true}}'
```

Ici, l' `-G` est utilisée pour spécifier que les données du paramètre `--data-urlencode` seront utilisées dans une `HTTP GET` au lieu de la requête `POST` qui serait utilisée autrement. Les données seront ajoutées à l'URL avec un `?` séparateur.

Lire API Docker Engine en ligne: https://riptutorial.com/fr/docker/topic/3935/api-docker-engine

Chapitre 3: Classement du contenu de Dockerfile

Remarques

1. Déclaration d'image de base (`FROM`)
2. Métadonnées (par exemple `MAINTAINER` , `LABEL`)
3. Installation des dépendances du système (par exemple, `apt-get install` , `apk add`)
4. Copie du fichier de dépendances de l'application (par exemple, `bower.json` , `package.json` , `build.gradle` , `requirements.txt`)
5. Installation de dépendances d'application (par exemple, `npm install` `pip install`)
6. Copier l'intégralité du code
7. Configuration des configurations d'exécution par défaut (par exemple, `CMD` , `ENTRYPOINT` , `ENV` , `EXPOSE`)

Ces commandes sont destinées à optimiser le temps de création à l'aide du mécanisme de mise en cache intégré de Docker.

Règle de base:

> Les parties qui changent souvent (par exemple, la base de code) doivent être placées au bas du fichier Dockerfile et vice-versa. Les parties qui changent rarement (par exemple les dépendances) doivent être placées en haut.

Examples

Simple Dockerfile

```
# Base image
FROM python:2.7-alpine

# Metadata
MAINTAINER John Doe <johndoe@example.com>

# System-level dependencies
RUN apk add --update \
    ca-certificates \
    && update-ca-certificates \
    && rm -rf /var/cache/apk/*

# App dependencies
COPY requirements.txt /requirements.txt
RUN pip install -r /requirements.txt

# App codebase
WORKDIR /app
COPY . ./
```

```
# Configs
ENV DEBUG true
EXPOSE 5000
CMD ["python", "app.py"]
```

MAINTAINER sera déconseillé dans Docker 1.13 et devra être remplacé par LABEL. (Source)

Exemple: LABEL Maintainer = "John Doe johndoe@example.com"

Lire Classement du contenu de Dockerfile en ligne:
https://riptutorial.com/fr/docker/topic/6448/classement-du-contenu-de-dockerfile

Chapitre 4: Comment déboguer lorsque la construction du docker échoue

Introduction

Quand un `docker build -t mytag .` échoue avec un message tel que `---> Running in d9a42e53eb5a The command '/bin/sh -c returned a non-zero code: 127` a `The command '/bin/sh -c returned a non-zero code: 127` (127 signifie « command not found, mais 1) il n'est pas trivial pour tout le monde 2) 127 peut être remplacé par 6 ou n'importe quoi) il peut être non trivial de trouver l'erreur dans une longue ligne

Examples

exemple basique

Comme dernière couche créée par

```
docker build -t mytag .
```

montré

```
---> Running in d9a42e53eb5a
```

Vous venez de lancer la dernière image créée avec un shell et lancez la commande, et vous aurez un message d'erreur plus clair

```
docker run -it d9a42e53eb5a /bin/bash
```

(cela suppose que / bin / bash est disponible, il peut être / bin / sh ou autre chose)

et avec l'invite, vous lancez la dernière commande défaillante et voyez ce qui est affiché

Lire Comment déboguer lorsque la construction du docker échoue en ligne: https://riptutorial.com/fr/docker/topic/8078/comment-deboguer-lorsque-la-construction-du-docker-echoue

Chapitre 5: Concept de volumes Docker

Remarques

Les nouveaux venus chez Docker ne réalisent souvent pas que les systèmes de fichiers Docker sont temporaires par défaut. Si vous démarrez une image Docker, vous obtiendrez un conteneur qui, à la surface, se comporte comme une machine virtuelle. Vous pouvez créer, modifier et supprimer des fichiers. Cependant, contrairement à une machine virtuelle, si vous arrêtez le conteneur et le redémarrez, toutes vos modifications seront perdues - tous les fichiers précédemment supprimés seront désormais de retour et tous les nouveaux fichiers ou modifications que vous avez apportés ne seront plus présents.

Les volumes dans les conteneurs Docker autorisent les données persistantes et le partage des données de la machine hôte dans un conteneur.

Examples

A) Lancer un conteneur avec un volume

```
[root@localhost ~]# docker run -it -v  /data  --name=vol3   8251da35e7a7 /bin/bash
root@d87bf9607836:/# cd /data/
root@d87bf9607836:/data# touch abc{1..10}
root@d87bf9607836:/data# ls
```

abc1 abc10 abc2 abc3 abc4 abc5 abc6 abc7 abc8 abc9

B) Maintenant, appuyez sur [cont + P + Q] pour sortir du conteneur sans terminer le conteneur en vérifiant si le conteneur est en cours d'exécution.

```
[root@localhost ~]# docker ps
```

D87bf9607836 8251da35e7a7 "/ bin / bash" Il y a une minute environ 31 secondes vol3 [root @ localhost ~] #

C) Exécuter 'docker inspect' pour vérifier plus d'informations sur le volume

```
[root@localhost ~]# docker inspect  d87bf9607836
```

"Mounts": [{"Nom": "cdf78fbf79a7c9363948e133abe4c572734cd788c95d36edea0448094ec9121c", "Source": "/ var / lib / docker / volumes / cdf78fbf79a7c9363948e133abe4c572734cd788c95d36edea0448094ec9121c / _data", "Destination": "/ data", "Driver": "local" "Mode": "", "RW": vrai

D) Vous pouvez attacher un volume de conteneurs en cours d'exécution à un autre conteneur

```
[root@localhost ~]# docker run -it  --volumes-from  vol3  8251da35e7a7  /bin/bash

root@ef2f5cc545be:/# ls
```

bin boot données dev etc home lib64 media mnt opt proc root exécuter sbin srv sys tmp usr var

`root@ef2f5cc545be:/# ls` / data abc1 abc10 abc2 abc3 abc4 abc5 abc6 abc7 abc8 abc9

E) Vous pouvez également monter votre répertoire de base à l'intérieur du conteneur

```
[root@localhost ~]# docker run -it  -v  /etc:/etc1 8251da35e7a7 /bin/bash
```

Ici: / etc est le répertoire de la machine hôte et / etc1 est la cible à l'intérieur du conteneur

Lire Concept de volumes Docker en ligne: https://riptutorial.com/fr/docker/topic/5908/concept-de-volumes-docker

Chapitre 6: Connexion des conteneurs

Paramètres

Paramètre	Détails
`tty:true`	Dans docker-compose.yml, le drapeau `tty: true` maintient la commande `sh` du conteneur en attente de saisie.

Remarques

Les pilotes du réseau `host` et du `bridge` peuvent connecter des conteneurs sur un seul hôte de docker. Pour permettre aux conteneurs de communiquer au-delà d'une machine, créez un réseau superposé. Les étapes pour créer le réseau dépendent de la gestion de vos hôtes Docker.

- Mode Swarm: le `docker network create --driver overlay`
- docker / swarm : nécessite un magasin clé-valeur externe

Examples

Réseau Docker

Les conteneurs du même réseau Docker ont accès aux ports exposés.

```
docker network create sample
docker run --net sample --name keys consul agent -server -client=0.0.0.0 -bootstrap
```

Dockerfile du consul expose `8500` , `8600` et plusieurs autres ports. Pour démontrer, exécutez un autre conteneur sur le même réseau:

```
docker run --net sample -ti alpine sh
/ # wget -qO- keys:8500/v1/catalog/nodes
```

Ici, le conteneur consul est résolu à partir des `keys` , le nom donné dans la première commande. Docker fournit une résolution DNS sur ce réseau, pour rechercher des conteneurs par leur `--name` .

Docker-compose

Les réseaux peuvent être spécifiés dans un fichier de composition (v2). Par défaut, tous les conteneurs sont dans un réseau partagé.

Commencez avec ce fichier: `example/docker-compose.yml` :

```
version: '2'
```

```
services:
  keys:
    image: consul
    command: agent -server -client=0.0.0.0 -bootstrap
  test:
    image: alpine
    tty: true
    command: sh
```

Démarrer cette pile avec `docker-compose up -d` créera un réseau nommé d'après le répertoire parent, dans ce cas `example_default` . Vérifier avec le `docker network ls`

```
 > docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
719eafa8690b        example_default     bridge              local
```

Connectez-vous au conteneur alpin pour vérifier que les conteneurs peuvent se résoudre et communiquer:

```
 > docker exec -ti example_test_1 sh
/ # nslookup keys
...
/ # wget -qO- keys:8500/v1/kv/?recurse
...
```

Un fichier de composition peut avoir une section `networks:` top level pour spécifier le nom du réseau, le pilote et d'autres options à partir de la commande docker network .

Liaison de conteneur

L'argument docker `--link` et le `link:` sections docker-compose `--link` *alias* sur d'autres conteneurs.

```
docker network create sample
docker run -d --net sample --name redis redis
```

Avec link, le nom d'origine ou le mappage résoudra le conteneur redis.

```
> docker run --net sample --link redis:cache -ti python:alpine sh -c "pip install redis &&
python"
>>> import redis
>>> r = redis.StrictRedis(host='cache')
>>> r.set('key', 'value')
True
```

Avant docker `1.10.0` , la liaison de conteneur permet également de configurer la connectivité réseau - comportement désormais fourni par le réseau Docker. Les liens dans les versions ultérieures fournissent uniquement un effet `legacy` sur le réseau de pont par défaut.

Lire Connexion des conteneurs en ligne: https://riptutorial.com/fr/docker/topic/6528/connexion-des-conteneurs

Chapitre 7: Conteneurs de course

Syntaxe

- docker run [OPTIONS] IMAGE [COMMAND] [ARG ...]

Examples

Lancer un conteneur

```
docker run hello-world
```

Cela va récupérer la dernière image hello-world du Docker Hub (si vous ne l'avez pas déjà fait), créer un nouveau conteneur et l'exécuter. Vous devriez voir un message indiquant que votre installation semble fonctionner correctement.

Exécuter une commande différente dans le conteneur

```
docker run docker/whalesay cowsay 'Hello, StackExchange!'
```

Cette commande dit à Docker de créer un conteneur à partir de l'image `docker/whalesay` et d'exécuter la commande `cowsay 'Hello, StackExchange!'` dedans. Il devrait imprimer une image d'une baleine disant `Hello, StackExchange!` à votre terminal.

Si le point d'entrée de l'image est la valeur par défaut, vous pouvez exécuter toute commande disponible dans l'image:

```
docker run docker/whalesay ls /
```

S'il a été modifié lors de la création de l'image, vous devez le restaurer à la valeur par défaut

```
docker run --entrypoint=/bin/bash docker/whalesay -c ls /
```

Supprimer automatiquement un conteneur après l'avoir exécuté

Normalement, un conteneur Docker persiste après sa sortie. Cela vous permet d'exécuter à nouveau le conteneur, d'inspecter son système de fichiers, etc. Cependant, vous souhaitez parfois exécuter un conteneur et le supprimer immédiatement après sa fermeture. Par exemple, pour exécuter une commande ou afficher un fichier à partir du système de fichiers. Docker fournit l'option de ligne de commande `--rm` à cette fin:

```
docker run --rm ubuntu cat /etc/hosts
```

Cela créera un conteneur à partir de l'image "ubuntu", affichera le contenu du fichier **/ etc / hosts**

et supprimera le conteneur immédiatement après sa fermeture. Cela permet d'éviter d'avoir à nettoyer les conteneurs après avoir fait des expériences.

> **Remarque:** L'indicateur `--rm` ne fonctionne pas avec l' `--detach -d` (`--detach`) dans le docker <1.13.0.

Lorsque l' `--rm` est définie, Docker supprime également les volumes associés au conteneur lorsque le conteneur est supprimé. Ceci est similaire à l'exécution de `docker rm -v my-container` . ***Seuls les volumes spécifiés sans nom sont supprimés*** .

Par exemple, avec `docker run -it --rm -v /etc -v logs:/var/log centos /bin/produce_some_logs` , le volume de `/etc` sera supprimé, mais pas le volume de `/var/log` . Les volumes hérités via --volumes-from seront supprimés avec la même logique - si le volume d'origine a été spécifié avec un nom, il ne sera pas supprimé.

Spécifier un nom

Par défaut, les conteneurs créés avec l' `docker run` reçoivent un nom aléatoire tel que `small_roentgen` ou `modest_dubinsky` . Ces noms ne sont pas particulièrement utiles pour identifier la fonction d'un conteneur. Il est possible de fournir un nom au conteneur en passant l'option de ligne de commande `--name` :

```
docker run --name my-ubuntu ubuntu:14.04
```

Les noms doivent être uniques. Si vous transmettez un nom déjà utilisé par un autre conteneur, Docker imprimera une erreur et aucun nouveau conteneur ne sera créé.

La spécification d'un nom sera utile lors du référencement du conteneur dans un réseau Docker. Cela fonctionne pour les conteneurs Docker d'arrière-plan et de premier plan.

Les conteneurs sur le réseau de pont par défaut **doivent** être liés pour communiquer par nom.

Liaison d'un port de conteneur à l'hôte

```
docker run -p "8080:8080" myApp
docker run -p "192.168.1.12:80:80" nginx
docker run -P myApp
```

Pour utiliser des ports sur l'hôte ont été exposés dans une image (via la directive `EXPOSE` Dockerfile ou l'option de ligne de commande `--expose` pour `docker run`), ces ports doivent être liés à l'hôte à l'aide de la commande `-p` ou `-P` options de ligne. L'utilisation de `-p` nécessite que le port particulier (et l'interface hôte facultative) soit spécifié. L'utilisation de l'option de ligne de commande majuscule `-P` force Docker à lier *tous* les ports exposés de l'image d'un conteneur à l'hôte.

Politique de redémarrage du conteneur (démarrage d'un conteneur au démarrage)

```
docker run --restart=always -d <container>
```

Par défaut, Docker ne redémarre pas les conteneurs lorsque le démon Docker redémarre, par exemple après le redémarrage du système hôte. Docker fournit une stratégie de redémarrage pour vos conteneurs en fournissant l'option de ligne de commande `--restart` . La fourniture de `--restart=always` provoquera toujours le redémarrage d'un conteneur après le redémarrage du démon Docker. **Cependant,** lorsque ce conteneur est arrêté manuellement (par exemple, avec `docker stop <container>`), la stratégie de redémarrage ne sera pas appliquée au conteneur.

Plusieurs options peuvent être spécifiées pour l'option `--restart` , en fonction de l'exigence (`--restart=[policy]`). Ces options affectent également le démarrage du conteneur au démarrage.

Politique	Résultat
non	La valeur **par défaut** Ne redémarre pas automatiquement le conteneur lorsque le conteneur est arrêté.
en cas d'échec [: max-retries]	Redémarrez uniquement si le conteneur se termine par un échec (`non-zero exit status`). Pour éviter de le redémarrer indéfiniment (en cas de problème), il est possible de limiter le nombre de tentatives de redémarrage des tentatives du démon Docker.
toujours	Toujours redémarrer le conteneur quel que soit le statut de sortie. Lorsque vous spécifiez `always` , le démon Docker essaiera de redémarrer le conteneur indéfiniment. Le conteneur démarrera également toujours au démarrage du démon, quel que soit l'état actuel du conteneur.
à moins que arrêté	Redémarrez toujours le conteneur quel que soit son état de sortie, mais ne le démarrez pas au démarrage du démon si le conteneur a déjà été mis à l'arrêt.

Exécuter un conteneur en arrière-plan

Pour garder un conteneur en cours d'exécution en arrière-plan, `-d` option de ligne de commande `-d` lors du démarrage du conteneur:

```
docker run -d busybox top
```

L'option `-d` exécute le conteneur en mode détaché. C'est aussi équivalent à `-d=true` .

Un conteneur en mode détaché ne peut pas être supprimé automatiquement lorsqu'il s'arrête, cela signifie que l'on ne peut pas utiliser l'option --rm en combinaison avec l'option -d.

Attribuer un volume à un conteneur

Un volume Docker est un fichier ou un répertoire qui persiste au-delà de la durée de vie du conteneur. Il est possible de monter un fichier hôte ou un répertoire dans un conteneur en tant que

volume (en contournant UnionFS).

Ajoutez un volume avec l'option de ligne de commande `-v` :

```
docker run -d -v "/data" awesome/app bootstrap.sh
```

Cela créera un volume et le montera sur le chemin `/data` à l'intérieur du conteneur.

- Remarque: Vous pouvez utiliser le drapeau `--rm` pour supprimer automatiquement le volume lorsque le conteneur est supprimé.

Montage des répertoires hôtes

Pour monter un fichier ou un répertoire hôte dans un conteneur:

```
docker run -d -v "/home/foo/data:/data" awesome/app bootstrap.sh
```

- **Lors de la spécification d'un répertoire hôte, un chemin absolu doit être fourni.**

Cela montera le répertoire hôte `/home/foo/data` sur `/data` à l'intérieur du conteneur. Ce volume "répertoire hôte monté sur une liaison" est identique à un `mount --bind` Linux `mount --bind` et monte donc temporairement le répertoire hôte sur le chemin de conteneur spécifié pendant la durée de vie du conteneur. Les modifications du volume à partir de l'hôte ou du conteneur sont reflétées immédiatement dans l'autre, car elles sont la même destination sur le disque.

UNIX exemple de montage d'un dossier relatif

```
docker run -d -v $(pwd)/data:/data awesome/app bootstrap.sh
```

Nommer les volumes

Un volume peut être nommé en fournissant une chaîne au lieu d'un chemin de répertoire hôte, docker créera un volume utilisant ce nom.

```
docker run -d -v "my-volume:/data" awesome/app bootstrap.sh
```

Après avoir créé un volume nommé, le volume peut être partagé avec d'autres conteneurs utilisant ce nom.

Définition des variables d'environnement

```
$ docker run -e "ENV_VAR=foo" ubuntu /bin/bash
```

Les deux `--env -e` et `--env` peuvent être utilisées pour définir des variables d'environnement à l'intérieur d'un conteneur. Il est possible de fournir de nombreuses variables d'environnement en utilisant un fichier texte:

```
$ docker run --env-file ./env.list ubuntu /bin/bash
```

Exemple de fichier de variable d'environnement:

```
# This is a comment
TEST_HOST=10.10.0.127
```

L' `--env-file` prend un nom de fichier comme argument et s'attend à ce que chaque ligne soit au format `VARIABLE=VALUE` , imitant l'argument passé à `--env` . Les lignes de commentaires ne doivent être précédées que du préfixe `#` .

Indépendamment de l'ordre de ces trois indicateurs, les `--env-file` sont traités en premier, puis les indicateurs `-e` / `--env` . De cette façon, toutes les variables d'environnement fournies individuellement avec `-e` ou `--env` remplaceront les variables fournies dans le fichier texte `--env-var` .

Spécifier un nom d'hôte

Par défaut, les conteneurs créés avec l'exécution de docker reçoivent un nom d'hôte aléatoire. Vous pouvez donner au conteneur un nom d'hôte différent en passant le drapeau --hostname:

```
docker run --hostname redbox -d ubuntu:14.04
```

Exécuter un conteneur de manière interactive

Pour exécuter un conteneur de manière interactive, `-it` options `-it` :

```
$ docker run -it ubuntu:14.04 bash
root@8ef2356d919a:/# echo hi
hi
root@8ef2356d919a:/#
```

`-i` garde STDIN ouvert, alors que `-t` alloue un pseudo-TTY.

Conteneur en cours d'exécution avec limites de mémoire / swap

Définir la limite de mémoire et désactiver la limite de swap

```
docker run -it -m 300M --memory-swap -1 ubuntu:14.04 /bin/bash
```

Définissez à la fois la mémoire et la limite de swap. Dans ce cas, le conteneur peut utiliser 300M de mémoire et 700M de swap.

```
docker run -it -m 300M --memory-swap 1G ubuntu:14.04 /bin/bash
```

Mettre un shell dans un conteneur en cours d'exécution

Connectez-vous à un conteneur en cours

d'exécution

Un utilisateur peut entrer un conteneur en cours d'exécution dans un nouveau shell interactif bash avec la commande **`exec`** .

Supposons qu'un conteneur s'appelle `jovial_morse` vous pouvez alors obtenir un shell pseudo-TTY interactif en exécutant:

```
docker exec -it jovial_morse bash
```

Se connecter à un conteneur en cours d'exécution avec un utilisateur spécifique

Si vous souhaitez entrer un conteneur en tant qu'utilisateur spécifique, vous pouvez le définir avec le paramètre `-u` ou `--user` . Le nom d'utilisateur doit exister dans le conteneur.

> `-u, --user` Nom d'utilisateur ou UID (format: `<name|uid>[:<group|gid>]`)

Cette commande se connecte à `jovial_morse` avec l'utilisateur `dockeruser`

```
docker exec -it -u dockeruser jovial_morse bash
```

Connectez-vous à un conteneur en cours d'exécution en tant que root

Si vous voulez vous connecter en tant que root, utilisez simplement le paramètre **`-u root`** .
L'utilisateur racine existe toujours.

```
docker exec -it -u root jovial_morse bash
```

Connectez-vous à une image

Vous pouvez également vous connecter à une image avec la commande `run` , mais cela nécessite un nom d'image au lieu d'un nom de conteneur.

```
docker run -it dockerimage bash
```

Connectez-vous à une image intermédiaire

(debug)

Vous pouvez également vous connecter à une image intermédiaire créée lors de la création d'un fichier Dockerfile.

Sortie de `docker build .`

```
$ docker build .
Uploading context 10240 bytes
Step 1 : FROM busybox
Pulling repository busybox
 ---> e9aa60c60128MB/2.284 MB (100%) endpoint: https://cdn-registry-1.docker.io/v1/
Step 2 : RUN ls -lh /
 ---> Running in 9c9e81692ae9
total 24
drwxr-xr-x    2 root     root        4.0K Mar 12  2013 bin
drwxr-xr-x    5 root     root        4.0K Oct 19 00:19 dev
drwxr-xr-x    2 root     root        4.0K Oct 19 00:19 etc
drwxr-xr-x    2 root     root        4.0K Nov 15 23:34 lib
lrwxrwxrwx    1 root     root           3 Mar 12  2013 lib64 -> lib
dr-xr-xr-x  116 root     root           0 Nov 15 23:34 proc
lrwxrwxrwx    1 root     root           3 Mar 12  2013 sbin -> bin
dr-xr-xr-x   13 root     root           0 Nov 15 23:34 sys
drwxr-xr-x    2 root     root        4.0K Mar 12  2013 tmp
drwxr-xr-x    2 root     root        4.0K Nov 15 23:34 usr
 ---> b35f4035db3f
Step 3 : CMD echo Hello world
 ---> Running in 02071fceb21b
 ---> f52f38b7823e
```

Notez le `---> Running in 02071fceb21b` , vous pouvez vous connecter à ces images:

```
docker run -it 02071fceb21b bash
```

Passer stdin au conteneur

Dans des cas tels que la restauration d'une sauvegarde de base de données ou si vous souhaitez transmettre certaines informations via un canal depuis l'hôte, vous pouvez utiliser l' `-i` en tant qu'argument pour `docker run` ou `docker exec` .

Par exemple, en supposant que vous souhaitiez mettre à la disposition d'un client mariadb conteneurisé un vidage de base de données sur l'hôte, dans un fichier `dump.sql` local, vous pouvez exécuter la commande suivante:

```
docker exec -i mariadb bash -c 'mariadb "-p$MARIADB_PASSWORD" ' < dump.sql
```

En général,

```
docker exec -i container command < file.stdin
```

Ou

```
docker exec -i container command <<EOF
inline-document-from-host-shell-HEREDOC-syntax
EOF
```

Détachement d'un conteneur

`docker run -it ...` vous êtes attaché à un conteneur en cours d'exécution avec un pty assigné (`docker run -it ...`), vous pouvez appuyer sur `Control P` - `Control Q` pour le détacher.

Remplacement de la directive sur les points d'entrée d'image

```
docker run --name="test-app" --entrypoint="/bin/bash" example-app
```

Cette commande remplace la directive `ENTRYPOINT` de l'image `example-app` lorsque l' `test-app` conteneur est créée. La directive `CMD` de l'image restera inchangée sauf indication contraire:

```
docker run --name="test-app" --entrypoint="/bin/bash" example-app /app/test.sh
```

Dans l'exemple ci-dessus, les `ENTRYPOINT` et `CMD` de l'image ont été remplacés. Ce processus de conteneur devient `/bin/bash /app/test.sh` .

Ajouter une entrée hôte au conteneur

```
docker run --add-host="app-backend:10.15.1.24" awesome-app
```

Cette commande ajoute une entrée au fichier `/etc/hosts` du conteneur, qui suit le format `--add-host <name>:<address>` . Dans cet exemple, le nom `app-backend` sera résolu en `10.15.1.24` . Ceci est particulièrement utile pour lier ensemble des composants d'applications disparates par programmation.

Empêcher le conteneur de s'arrêter quand aucune commande n'est en cours d'exécution

Un conteneur s'arrêtera si aucune commande n'est exécutée au premier plan. L'utilisation de l'option `-t` empêchera le conteneur de s'arrêter, même s'il est détaché avec l'option `-d` .

```
docker run -t -d debian bash
```

Arrêter un conteneur

```
docker stop mynginx
```

De plus, l'ID du conteneur peut également être utilisé pour arrêter le conteneur au lieu de son nom.

Cela arrêtera un conteneur en cours d'exécution en envoyant le signal SIGTERM, puis le signal

SIGKILL si nécessaire.

De plus, la commande kill peut être utilisée pour envoyer immédiatement un SIGKILL ou tout autre signal spécifié en utilisant l'option `-s` .

```
docker kill mynginx
```

Signal spécifié:

```
docker kill -s SIGINT mynginx
```

L'arrêt d'un conteneur ne le supprime pas. Utilisez le `docker ps -a` pour voir votre conteneur arrêté.

Exécuter une autre commande sur un conteneur en cours d'exécution

Si nécessaire, vous pouvez demander à Docker d'exécuter des commandes supplémentaires sur un conteneur déjà en cours à l'aide de la commande `exec` . Vous avez besoin de l'ID du conteneur que vous pouvez obtenir avec `docker ps` .

```
docker exec 294fbc4c24b3 echo "Hello World"
```

Vous pouvez attacher un shell interactif si vous utilisez l'option `-it` .

```
docker exec -it 294fbc4c24b3 bash
```

Exécution d'applications GUI dans un conteneur Linux

Par défaut, un conteneur Docker ne pourra pas *exécuter* une application graphique.

Avant cela, le socket X11 doit être transmis en premier au conteneur, de sorte qu'il puisse être utilisé directement. La variable d'environnement *DISPLAY* doit également être transmise:

```
docker run -v /tmp/.X11-unix:/tmp/.X11-unix -e DISPLAY=unix$DISPLAY <image-name>
```

Cela échouera au début, car nous n'avons pas défini les autorisations sur l'hôte du serveur X:

```
cannot connect to X server unix:0
```

Le moyen le plus rapide (mais pas le plus sûr) est d’autoriser l’accès directement avec:

```
xhost +local:root
```

Après avoir fini avec le conteneur, nous pouvons revenir à l’état initial avec:

```
xhost -local:root
```

Une autre méthode (plus sûre) consiste à préparer un fichier Dockerfile qui générera une nouvelle image qui utilisera les informations d'identification de notre utilisateur pour accéder au serveur X:

```
FROM <iamge-name>
MAINTAINER <you>

# Arguments picked from the command line!
ARG user
ARG uid
ARG gid

#Add new user with our credentials
ENV USERNAME ${user}
RUN useradd -m $USERNAME && \
        echo "$USERNAME:$USERNAME" | chpasswd && \
        usermod --shell /bin/bash $USERNAME && \
        usermod  --uid ${uid} $USERNAME && \
        groupmod --gid ${gid} $USERNAME

USER ${user}

WORKDIR /home/${user}
```

Lorsque vous appelez `docker build` partir de la ligne de commande, vous devez transmettre les variables *ARG* qui apparaissent dans le fichier Docker:

```
docker build --build-arg user=$USER --build-arg uid=$(id -u) --build-arg gid=$(id -g) -t <new-image-with-X11-enabled-name> -f <Dockerfile-for-X11> .
```

Maintenant, avant de créer un nouveau conteneur, nous devons créer un fichier xauth avec une autorisation d'accès:

```
xauth nlist $DISPLAY | sed -e 's/^..../ffff/' | xauth -f /tmp/.docker.xauth nmerge -
```

Ce fichier doit être monté dans le conteneur lors de sa création / exécution:

```
docker run -e DISPLAY=unix$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix -v
/tmp/.docker.xauth:/tmp/.docker.xauth:rw -e XAUTHORITY=/tmp/.docker.xauth
```

Lire Conteneurs de course en ligne: https://riptutorial.com/fr/docker/topic/679/conteneurs-de-course

Chapitre 8: Créer un service avec persistance

Syntaxe

- docker volume create --name <nom_volume> # Crée un volume appelé <nom_volume>
- docker run -v <nom_volume>: <point_montage> -d crramirez / limesurvey: latest # Montez le volume <nom_volume> dans le répertoire <point_montage> du conteneur

Paramètres

Paramètre	Détails
--name <nom_volume>	Indiquez le nom du volume à créer
-v <nom_volume>: <point_montage>	Indiquez où le volume nommé sera monté dans le conteneur

Remarques

La persistance est créée dans des conteneurs de docker à l'aide de volumes. Docker a plusieurs façons de gérer les volumes. Les volumes nommés sont très pratiques par:

- Ils persistent même lorsque le conteneur est supprimé à l'aide de l'option -v.
- La seule façon de supprimer un volume nommé consiste à effectuer un appel explicite à docker volume rm
- Les volumes nommés peuvent être partagés entre des conteneurs sans liaison ou option --volumes-from.
- Ils n'ont pas de problèmes de permission que les volumes montés par l'hôte ont.
- Ils peuvent être manipulés à l'aide de la commande docker volume.

Examples

Persistance avec des volumes nommés

La persistance est créée dans des conteneurs de docker à l'aide de volumes. Créons un conteneur Limesurvey et conservons la base de données, le contenu téléchargé et la configuration dans un volume nommé:

```
docker volume create --name mysql
docker volume create --name upload

docker run -d --name limesurvey -v mysql:/var/lib/mysql -v upload:/app/upload -p 80:80
crramirez/limesurvey:latest
```

Sauvegarder un contenu de volume nommé

Nous devons créer un conteneur pour monter le volume. Ensuite, archivez-le et téléchargez l'archive sur notre hôte.

Créons d'abord un volume de données avec quelques données:

```
docker volume create --name=data
echo "Hello World" |  docker run -i --rm=true -v data:/data ubuntu:trusty tee /data/hello.txt
```

Sauvegardons les données:

```
docker run -d --name backup -v data:/data ubuntu:trusty tar -czvf /tmp/data.tgz /data
docker cp backup:/tmp/data.tgz data.tgz
docker rm -fv backup
```

Testons:

```
tar -xzvf data.tgz
cat data/hello.txt
```

Lire Créer un service avec persistance en ligne: https://riptutorial.com/fr/docker/topic/7429/creer-un-service-avec-persistance

Chapitre 9: Déboguer un conteneur

Syntaxe

- docker stats [OPTIONS] [CONTAINER ...]
- docker logs [OPTIONS] CONTAINER
- docker top [OPTIONS] CONTAINER [ps OPTIONS]

Examples

Entrer dans un conteneur en cours d'exécution

Pour exécuter des opérations dans un conteneur, utilisez la commande `docker exec` . Parfois, cela s'appelle "entrer dans le conteneur" car toutes les commandes sont exécutées à l'intérieur du conteneur.

```
docker exec -it container_id bash
```

ou

```
docker exec -it container_id /bin/sh
```

Et maintenant, vous avez un shell dans votre conteneur en cours d'exécution. Par exemple, répertoriez les fichiers dans un répertoire, puis quittez le conteneur:

```
docker exec container_id ls -la
```

Vous pouvez utiliser le `-u flag` pour entrer le conteneur avec un utilisateur spécifique, par exemple `uid=1013` , `gid=1023` .

```
docker exec -it -u 1013:1023 container_id ls -la
```

L'ID utilisateur et le gid ne doivent pas nécessairement exister dans le conteneur, mais la commande peut générer des erreurs. Si vous souhaitez lancer un conteneur et entrer immédiatement à l'intérieur pour vérifier quelque chose, vous pouvez le faire.

```
docker run...; docker exec -it $(docker ps -lq) bash
```

la commande `docker ps -lq` ne `docker ps -lq` que l'id du dernier conteneur (le l in `-lq`) démarré. (cela suppose que vous ayez bash comme interpréteur disponible dans votre conteneur, vous pouvez avoir sh ou zsh ou tout autre)

Surveillance de l'utilisation des ressources

L'inspection de l'utilisation des ressources du système est un moyen efficace de trouver des

applications qui fonctionnent mal. Cet exemple est équivalent à la commande `top` traditionnelle pour les conteneurs:

```
docker stats
```

Pour suivre les statistiques de conteneurs spécifiques, listez-les sur la ligne de commande:

```
docker stats 7786807d8084 7786807d8085
```

Docker stats affiche les informations suivantes:

```
CONTAINER     CPU %   MEM USAGE / LIMIT   MEM %    NET I/O               BLOCK I/O
7786807d8084  0.65%   1.33 GB / 3.95 GB   33.67%   142.2 MB / 57.79 MB   46.32 MB / 0 B
```

Par défaut, les `docker stats` affichent l'ID des conteneurs, ce qui n'est pas très utile, si vous préférez afficher les noms du conteneur, faites simplement

```
docker stats $(docker ps --format '{{.Names}}')
```

Surveillance des processus dans un conteneur

L'inspection de l'utilisation des ressources du système est un moyen efficace de réduire un problème sur une application en cours d'exécution. Cet exemple est un équivalent de la commande `ps` traditionnelle pour les conteneurs.

```
docker top 7786807d8084
```

Pour filtrer le format de la sortie, ajoutez des options `ps` sur la ligne de commande:

```
docker top 7786807d8084 faux
```

Ou, pour obtenir la liste des processus exécutés en tant que root, ce qui est une pratique potentiellement dangereuse:

```
docker top 7786807d8084 -u root
```

La commande `docker top` s'avère particulièrement utile lors du dépannage de conteneurs minimalistes sans shell ni commande `ps` .

Attacher à un conteneur en cours d'exécution

"Attachement à un conteneur" est l'acte de démarrer une session de terminal dans le contexte dans lequel le conteneur (et les programmes qu'il contient) est en cours d'exécution. Ceci est principalement utilisé à des fins de débogage, mais peut également être nécessaire si des données spécifiques doivent être transmises aux programmes exécutés dans le conteneur.

La commande `attach` est utilisée pour cela. Il a cette syntaxe:

```
docker attach <container>
```

`<container>` peut être soit l'identifiant du conteneur, soit le nom du conteneur. Par exemple:

```
docker attach c8a9cf1a1fa8
```

Ou:

```
docker attach graceful_hopper
```

Vous pouvez avoir besoin de `sudo` les commandes ci-dessus, selon votre utilisateur et comment docker est configuré.

> Remarque: Attach ne permet qu'une seule session shell à joindre à un conteneur à la fois.
>
> Attention: *toutes* les entrées au clavier seront transmises au conteneur. Frapper `Ctrl-c` va *tuer* votre conteneur.

Pour détacher d'un conteneur attaché, appuyez successivement sur `Ctrl-p` puis `Ctrl-q`

Pour attacher plusieurs sessions shell à un conteneur, ou simplement comme alternative, vous pouvez utiliser `exec` . En utilisant l'ID du conteneur:

```
docker exec -i -t c8a9cf1a1fa8 /bin/bash
```

En utilisant le nom du conteneur:

```
docker exec -i -t graceful_hopper /bin/bash
```

`exec` exécute un programme dans un conteneur, dans ce cas `/bin/bash` (un shell, probablement un conteneur). `-i` indique une session interactive, tandis que `-t` alloue un pseudo-TTY.

> Remarque: contrairement à l' *attachement* , si vous `appuyez` sur `Ctrl-c` , la commande *exec* 'd ne se terminera que lorsque vous exécutez de manière interactive.

Impression des journaux

Suivre les journaux est le moyen le moins intrusif de déboguer une application en cours d'exécution. Cet exemple reproduit le comportement du `tail -f some-application.log` traditionnel sur le conteneur `7786807d8084` .

```
docker logs --follow --tail 10 7786807d8084
```

Cette commande affiche essentiellement la sortie standard du processus conteneur (le processus avec pid 1).

Si vos journaux n'incluent pas nativement l'horodatage, vous pouvez ajouter l'indicateur `--`

`timestamps` .

Il est possible de regarder les journaux d'un conteneur arrêté, soit

- démarrer le conteneur défaillant avec le `docker run ... ; docker logs $(docker ps -lq)`
- trouver l'identifiant ou le nom du conteneur avec

```
docker ps -a
```

et alors

`docker logs container-id` ou

`docker logs containername`

comme il est possible de regarder les journaux d'un conteneur arrêté

Débogage du processus du conteneur Docker

Docker est juste un moyen sophistiqué d'exécuter un processus, pas une machine virtuelle. Par conséquent, le débogage d'un processus "dans un conteneur" est également possible "sur l'hôte" en examinant simplement le processus du conteneur en cours d'exécution en tant qu'utilisateur disposant des autorisations appropriées pour inspecter ces processus sur l'hôte (par exemple root). Par exemple, il est possible de répertorier tous les "processus de conteneur" sur l'hôte en exécutant un simple `ps` tant que root:

```
sudo ps aux
```

Tous les conteneurs Docker en cours d'exécution seront répertoriés dans la sortie.

Cela peut être utile lors du développement d'applications pour déboguer un processus exécuté dans un conteneur. En tant qu'utilisateur disposant des autorisations appropriées, des utilitaires de débogage classiques peuvent être utilisés sur le processus de conteneur, tels que strace, ltrace, gdb, etc.

Lire Déboguer un conteneur en ligne: https://riptutorial.com/fr/docker/topic/1333/deboguer-un-conteneur

Chapitre 10: Docker dans Docker

Examples

Jenkins CI Container utilisant Docker

Ce chapitre décrit comment configurer un conteneur Docker avec Jenkins à l'intérieur, capable d'envoyer des commandes Docker à l'installation Docker (le démon Docker) de l'hôte. Utiliser efficacement Docker dans Docker. Pour ce faire, nous devons créer une image Docker personnalisée basée sur une version arbitraire de l'image officielle Jenkins Docker. Le fichier Dockerfile (l'instruction de génération de l'image) ressemble à ceci:

```
FROM jenkins

USER root

RUN cd /usr/local/bin && \
curl https://master.dockerproject.org/linux/amd64/docker > docker  && \
chmod +x docker  && \
groupadd -g 999 docker && \
usermod -a -G docker jenkins

USER Jenkins
```

Ce fichier Dockerfile crée une image qui contient les fichiers binaires du client Docker que ce client est utilisé pour communiquer avec un démon Docker. Dans ce cas, le démon Docker de l'hôte. L'instruction `RUN` dans ce fichier crée également un groupe d'utilisateurs UNIX avec l'UID 999 et y ajoute l'utilisateur Jenkins. Pourquoi exactement cela est nécessaire est décrit dans le chapitre suivant. Avec cette image, nous pouvons exécuter un serveur Jenkins qui peut utiliser les commandes Docker, mais si nous exécutons simplement cette image, le client Docker installé dans l'image ne peut pas communiquer avec le démon Docker de l'hôte. Ces deux composants communiquent via un socket UNIX `/var/run/docker.sock` . Sous Unix, c'est un fichier comme tout le reste, donc nous pouvons facilement le monter dans le conteneur Jenkins. Ceci est fait avec la commande `docker run -v /var/run/docker.sock:/var/run/docker.sock --name jenkins MY_CUSTOM_IMAGE_NAME` . Mais ce fichier monté appartient à `docker:root` et à cause de cela, le fichier Dockerfile crée ce groupe avec un UID bien connu et y ajoute l'utilisateur Jenkins. Maintenant, le conteneur Jenkins est vraiment capable d'exécuter et d'utiliser Docker. En production, la commande run doit également contenir `-v jenkins_home:/var/jenkins_home` pour sauvegarder le répertoire Jenkins_home et bien sûr un mappage de port pour accéder au serveur via le réseau.

Lire Docker dans Docker en ligne: https://riptutorial.com/fr/docker/topic/8012/docker-dans-docker

Chapitre 11: Docker Data Volumes

Introduction

Les volumes de données Docker permettent de conserver les données indépendamment du cycle de vie du conteneur. Les volumes présentent un certain nombre de fonctionnalités utiles telles que:

Montage d'un répertoire hôte dans le conteneur, partage des données entre les conteneurs à l'aide du système de fichiers et conservation des données si un conteneur est supprimé

Syntaxe

- volume du docker [OPTIONS] [COMMANDE]

Examples

Montage d'un répertoire de l'hôte local dans un conteneur

Il est possible de monter un répertoire hôte sur un chemin spécifique de votre conteneur à l'aide de l'option de ligne de commande `-v` ou `--volume` . L'exemple suivant montera `/etc` sur l'hôte vers `/mnt/etc` dans le conteneur:

```
(on linux) docker run -v "/etc:/mnt/etc" alpine cat /mnt/etc/passwd
(on windows)  docker run -v "/c/etc:/mnt/etc" alpine cat /mnt/etc/passwd
```

L'accès par défaut au volume à l'intérieur du conteneur est en lecture-écriture. Pour monter un volume en lecture seule à l'intérieur d'un conteneur, utilisez le suffixe `:ro` :

```
docker run -v "/etc:/mnt/etc:ro" alpine touch /mnt/etc/passwd
```

Créer un volume nommé

```
docker volume create --name="myAwesomeApp"
```

L'utilisation d'un volume nommé rend la gestion des volumes beaucoup plus lisible par l'homme. Il est possible de créer un volume nommé en utilisant la commande spécifiée ci-dessus, mais il est également possible de créer un volume nommé à l'intérieur d'une commande d' `docker run` utilisant l'option de ligne de commande `-v` ou `--volume` :

```
docker run -d --name="myApp-1" -v="myAwesomeApp:/data/app" myApp:1.5.3
```

Notez que la création d'un volume nommé sous cette forme est similaire au montage d'un fichier / répertoire hôte en tant que volume, sauf qu'au lieu d'un chemin valide, le nom du volume est

spécifié. Une fois créés, les volumes nommés peuvent être partagés avec d'autres conteneurs:

```
docker run -d --name="myApp-2" --volumes-from "myApp-1" myApp:1.5.3
```

Après avoir exécuté la commande ci - dessus, un nouveau conteneur a été créé avec le nom `myApp-2` de la `myApp:1.5.3` image, qui partage le `myAwesomeApp` volume nommé avec `myApp-1` . Le volume nommé `myAwesomeApp` est monté dans `/data/app` dans le `myApp-2` , tout comme il est monté dans `/data/app` dans le `myApp-1` .

Lire Docker Data Volumes en ligne: https://riptutorial.com/fr/docker/topic/1318/docker-data-volumes

Chapitre 12: docker inspecte l'obtention de différents champs pour la clé: valeur et éléments de la liste

Examples

divers docker inspecter des exemples

Je trouve que les exemples dans le `docker inspect` documentation semblent magiques, mais n'expliquent pas beaucoup.

Docker inspect est important car il s'agit de la manière propre d'extraire des informations d'un conteneur `docker inspect -f ... container_id` cours d'exécution `docker inspect -f ... container_id`

(ou tout conteneur en cours d'exécution)

```
docker inspect -f ... $(docker ps -q)
```

éviter certains peu fiables

```
docker command | grep or awk | tr or cut
```

Lorsque vous lancez un `docker inspect` vous pouvez facilement obtenir les valeurs du "niveau supérieur", avec une syntaxe de base comme pour un conteneur exécutant htop (à partir de https://hub.docker.com/r/jess/htop/). avec un pid ae1

```
docker inspect -f '{{.Created}}' ae1
```

peut montrer

```
2016-07-14T17:44:14.159094456Z
```

ou

```
docker inspect -f '{{.Path}}' ae1
```

peut montrer

```
htop
```

Maintenant, si j'extrais une partie de mon `docker inspect`

je vois

```
"State": { "Status": "running", "Running": true, "Paused": false, "Restarting": false, "OOMKilled": false, "Dead": false, "Pid": 4525, "ExitCode": 0, "Error": "", "StartedAt": "2016-07-14T17:44:14.406286293Z", "FinishedAt": "0001-01-01T00:00:00Z"
```

Je reçois donc un dictionnaire, comme il a `{ ...}` et beaucoup de clé: valeurs

Donc la commande

```
docker inspect -f '{{.State}}' ae1
```

renverra une liste, telle que

```
{running true false false false false 4525 0 2016-07-14T17:44:14.406286293Z 0001-01-
01T00:00:00Z}
```

Je peux obtenir la valeur de State.Pid facilement

```
docker inspect -f '{{ .State.Pid }}' ae1
```

Je reçois

```
4525
```

Parfois, docker inspecter donne une liste car elle commence par `[` et se termine par `]`

un autre exemple, avec un autre conteneur

```
docker inspect -f '{{ .Config.Env }}' 7a7
```

donne

```
[DISPLAY=:0 PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin LANG=fr_FR.UTF-8
LANGUAGE=fr_FR:en LC_ALL=fr_FR.UTF-8 DEBIAN_FRONTEND=noninteractive HOME=/home/gg WINEARCH=win32
WINEPREFIX=/home/gg/.wine_captvty]
```

Afin d'obtenir le premier élément de la liste, nous ajoutons index avant le champ requis et 0 (comme premier élément) après, donc

```
docker inspect -f '{{ index ( .Config.Env) 0 }}' 7a7
```

donne

```
DISPLAY=:0
```

Nous obtenons l'élément suivant avec 1 au lieu de 0 en utilisant la même syntaxe

```
docker inspect -f '{{ index ( .Config.Env) 1 }}' 7a7
```

donne

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Nous pouvons obtenir le nombre d'éléments de cette liste

```
docker inspect -f '{{ len .Config.Env }}' 7a7
```

donne

```
9
```

et nous pouvons obtenir le dernier élément de la liste, la syntaxe n'est pas facile

```
docker inspect -f "{{ index .Config.Cmd $[$(docker inspect -format '{{ len .Config.Cmd }}'
$CID)-1]}}" 7a7
```

Lire docker inspecte l'obtention de différents champs pour la clé: valeur et éléments de la liste en ligne: https://riptutorial.com/fr/docker/topic/6470/docker-inspecte-l-obtention-de-differents-champs-pour-la-cle--valeur-et-elements-de-la-liste

Chapitre 13: Docker Machine

Introduction

Gestion à distance de plusieurs hôtes de moteur docker.

Remarques

`docker-machine` gère les hôtes distants exécutant Docker.

L'outil de ligne de commande `docker-machine` gère le cycle de vie complet de la machine à l'aide de pilotes spécifiques au fournisseur. Il peut être utilisé pour sélectionner une machine "active". Une fois sélectionnée, une machine active peut être utilisée comme s'il s'agissait du moteur Docker local.

Examples

Obtenir les informations actuelles sur l'environnement Docker Machine

Toutes ces commandes sont des commandes shell.

`docker-machine env` pour obtenir la configuration par défaut du docker-machine par défaut

`eval $(docker-machine env)` pour obtenir la configuration actuelle de docker-machine et définir l'environnement shell actuel pour utiliser cette machine-docker.

Si votre shell est configuré pour utiliser un proxy, vous pouvez spécifier l'option --no-proxy afin de contourner le proxy lors de la connexion à votre docker-machine: `eval $(docker-machine env --no-proxy)`

Si vous avez plusieurs machines docker, vous pouvez spécifier le nom de la machine en argument: `eval $(docker-machine env --no-proxy machinename)`

SSH dans une machine à docker

Toutes ces commandes sont des commandes shell

- Si vous devez vous connecter directement à un docker-machine en cours d'exécution, vous pouvez le faire:

`docker-machine ssh` à ssh dans le docker-machine par défaut

`docker-machine ssh machinename` to ssh dans un docker-machine autre que celui par défaut

- Si vous souhaitez simplement exécuter une seule commande, vous pouvez le faire. Pour exécuter la `uptime` sur le docker-machine par défaut pour voir depuis combien de temps il est

exécuté, lancez le `docker-machine ssh default uptime`

Créer une machine Docker

L'utilisation de `docker-machine` est la meilleure méthode pour installer Docker sur une machine. Il appliquera automatiquement les meilleurs paramètres de sécurité disponibles, notamment la génération d'une paire unique de certificats SSL pour l'authentification mutuelle et les clés SSH.

Pour créer un ordinateur local à l'aide de Virtualbox:

```
docker-machine create --driver virtualbox docker-host-1
```

Pour installer Docker sur une machine existante, utilisez le pilote `generic` :

```
docker-machine -D create -d generic --generic-ip-address 1.2.3.4 docker-host-2
```

L'option `--driver` indique à docker comment créer la machine. Pour obtenir la liste des pilotes pris en charge, voir:

- officiellement pris en charge
- tierce personne

Liste des machines docker

La liste des machines Docker renverra l'état, l'adresse et la version de Docker de chaque machine docker.

```
 docker-machine ls
```

Imprimera quelque chose comme:

```
NAME             ACTIVE   DRIVER    STATE     URL                         SWARM   DOCKER
ERRORS
docker-machine-1 -        ovh       Running   tcp://1.2.3.4:2376                  v1.11.2
docker-machine-2 -        generic   Running   tcp://1.2.3.5:2376                  v1.11.2
```

Pour répertorier les machines en cours d'exécution:

```
docker-machine ls --filter state=running
```

Pour lister les machines d'erreur:

```
docker-machine ls --filter state=
```

Pour lister les machines dont le nom commence par "side-project-", utilisez le filtre Golang:

```
docker-machine ls --filter name="^side-project-"
```

Pour obtenir uniquement la liste des URL de la machine:

```
docker-machine ls --format '{{ .URL }}'
```

Voir https://docs.docker.com/machine/reference/ls/ pour la référence complète de la commande.

Mettre à niveau une machine Docker

La mise à niveau d'un docker implique un temps d'arrêt et peut nécessiter une planification. Pour mettre à niveau un ordinateur fixe, exécutez:

```
docker-machine upgrade docker-machine-name
```

Cette commande n'a pas d'options

Obtenir l'adresse IP d'une machine de docker

Pour obtenir l'adresse IP d'un ordinateur fixe, vous pouvez le faire avec cette commande:

```
docker-machine ip machine-name
```

Lire Docker Machine en ligne: https://riptutorial.com/fr/docker/topic/1349/docker-machine

Chapitre 14: Docker Registry

Examples

Lancer le registre

Ne pas utiliser le `registry:latest` ! Cette image pointe vers l'ancien registre v1. Ce projet Python n'est plus en cours de développement. Le nouveau registre v2 est écrit en Go et est activement maintenu. Lorsque les gens se réfèrent à un "registre privé", ils font référence au registre v2, *pas* au registre v1!

```
docker run -d -p 5000:5000 --name="registry" registry:2
```

La commande ci-dessus exécute la version la plus récente du registre, qui peut être trouvée dans le projet Docker Distribution .

Pour plus d'exemples de fonctionnalités de gestion des images, telles que le marquage, l'extraction ou la poussée, consultez la section sur la gestion des images.

Configurez le registre avec le backend de stockage AWS S3

La configuration d'un registre privé pour utiliser un backend AWS S3 est simple. Le registre peut le faire automatiquement avec la bonne configuration. Voici un exemple de ce que devrait `config.yml` votre fichier `config.yml` :

```
storage:
    s3:
        accesskey: AKAAAAAACCCCCCCBBBDA
        secretkey: rn9rjnNuX44iK+26qpM4cDEoOnonbBW98FYaiDtS
        region: us-east-1
        bucket: registry.example.com
        encrypt: false
        secure: true
        v4auth: true
        chunksize: 5242880
        rootdirectory: /registry
```

Les champs `accesskey` et `secretkey` sont des identifiants IAM avec des autorisations S3 spécifiques (voir la documentation pour plus d'informations). Il peut tout aussi facilement utiliser des informations d'identification avec la politique `AmazonS3FullAccess` associée. La `region` est la région de votre seau S3. Le `bucket` est le nom du seau. Vous pouvez choisir de stocker vos images cryptées avec `encrypt` . Le champ `secure` indique l'utilisation de HTTPS. Vous devez généralement définir `v4auth` sur true, même si sa valeur par défaut est false. Le champ `chunksize` vous permet de respecter les exigences de l'API S3 selon lesquelles les téléchargements fragmentés doivent avoir une taille d'au moins cinq mégaoctets. Enfin, `rootdirectory` spécifie un répertoire à utiliser sous votre `rootdirectory` S3.

D' autres solutions de stockage peuvent être configurées tout aussi facilement.

Lire Docker Registry en ligne: https://riptutorial.com/fr/docker/topic/4173/docker-registry

Chapitre 15: Dockerfiles

Introduction

Dockerfiles sont des fichiers utilisés pour créer des images Docker par programmation. Ils vous permettent de créer rapidement et de manière reproductible une image Docker, ce qui vous permet de collaborer. Les fichiers Docker contiennent des instructions pour créer une image Docker. Chaque instruction est écrite sur une ligne et est donnée sous la forme `<INSTRUCTION><argument(s)>` . Les fichiers Docker sont utilisés pour créer des images Docker à l'aide de la commande `docker build` .

Remarques

Dockerfiles sont de la forme:

```
# This is a comment
INSTRUCTION arguments
```

- Les commentaires commencent par un `#`
- Les instructions sont uniquement en majuscules
- La première instruction d'un fichier Docker doit être `FROM` pour spécifier l'image de base

---

Lors de la création d'un fichier Docker, le client Docker enverra un "contexte de construction" au démon Docker. Le contexte de génération inclut tous les fichiers et dossiers du même répertoire que le fichier Docker. `COPY` et `ADD` ne peuvent utiliser que des fichiers de ce contexte.

Certains fichiers Docker peuvent commencer par:

```
# escape=`
```

Ceci est utilisé pour demander à l'analyseur Docker d'utiliser `` ` `` comme caractère d'échappement au lieu de `\` . Ceci est surtout utile pour les fichiers Windows Docker.

Examples

HelloWorld Dockerfile

Un fichier Dockerfile minimal ressemble à ceci:

```
FROM alpine
CMD ["echo", "Hello StackOverflow!"]
```

Cela indiquera à Docker de créer une image basée sur Alpine (`FROM`), une distribution minimale

pour les conteneurs et d'exécuter une commande spécifique (`CMD`) lors de l'exécution de l'image résultante.

Construisez et exécutez-le:

```
docker build -t hello .
docker run --rm hello
```

Cela va sortir:

```
Hello StackOverflow!
```

Copier des fichiers

Pour copier des fichiers à partir du contexte de génération dans une image Docker, utilisez l'instruction `COPY` :

```
COPY localfile.txt containerfile.txt
```

Si le nom du fichier contient des espaces, utilisez la syntaxe alternative:

```
COPY ["local file", "container file"]
```

La commande `COPY` prend en charge les caractères génériques. Il peut être utilisé par exemple pour copier toutes les images dans le répertoire `images/` :

```
COPY *.jpg images/
```

Remarque: dans cet exemple, les `images/` peuvent ne pas exister. Dans ce cas, Docker le créera automatiquement.

Exposer un port

Pour déclarer des ports exposés à partir d'un fichier Docker, utilisez l'instruction `EXPOSE` :

```
EXPOSE 8080 8082
```

Les paramètres des ports exposés peuvent être remplacés à partir de la ligne de commande Docker, mais il est recommandé de les définir explicitement dans le fichier Docker car cela aide à comprendre ce que fait une application.

Dockerfiles meilleures pratiques

Opérations communes de groupe

Docker construit des images en tant que collection de calques. Chaque couche ne peut ajouter que des données, même si ces données indiquent qu'un fichier a été supprimé. Chaque

instruction crée un nouveau calque. Par exemple:

```
RUN apt-get -qq update
RUN apt-get -qq install some-package
```

A quelques inconvénients:

- Il va créer deux couches, produisant une image plus grande.
- Utiliser `apt-get update` seul dans une instruction `RUN` entraîne des problèmes de mise en cache et, par la suite, les instructions d' `apt-get install` peuvent **échouer** . Supposons que vous modifiez ultérieurement `apt-get install` en ajoutant des paquets supplémentaires, puis que docker interprète les instructions initiales et modifiées comme étant identiques et réutilise le cache des étapes précédentes. En conséquence, la commande `apt-get update` n'est **pas** exécutée car sa version en cache est utilisée lors de la génération.

Au lieu de cela, utilisez:

```
RUN apt-get -qq update && \
    apt-get -qq install some-package
```

comme cela ne produit qu'une couche.

Mentionnez le mainteneur

C'est généralement la deuxième ligne du fichier Docker. Il indique qui est responsable et sera en mesure d'aider.

```
LABEL maintainer John Doe <john.doe@example.com>
```

Si vous le sautez, cela ne cassera pas votre image. Mais cela n’aidera pas non plus vos utilisateurs.

Être concis

Gardez votre fichier Dockerfile court. Si une configuration complexe est nécessaire, envisagez d'utiliser un script dédié ou de configurer des images de base.

Instruction utilisateur

```
USER daemon
```

L'instruction `USER` définit le nom d'utilisateur ou l'UID à utiliser lors de l'exécution de l'image et pour toutes les instructions `RUN` , `CMD` et `ENTRYPOINT` qui suivent dans le `Dockerfile` .

Instruction WORKDIR

```
WORKDIR /path/to/workdir
```

L'instruction `WORKDIR` définit le répertoire de travail pour toutes les instructions `RUN` , `CMD` , `ENTRYPOINT` , `COPY` et `ADD` qui le suivent dans le fichier Dockerfile. Si le `WORKDIR` n'existe pas, il sera créé même s'il n'est utilisé dans aucune instruction `Dockerfile` ultérieure.

Il peut être utilisé plusieurs fois dans le `Dockerfile` . Si un chemin relatif est fourni, il sera relatif au chemin de l'instruction `WORKDIR` précédente. Par exemple:

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

La sortie de la commande `pwd` finale dans ce `Dockerfile` serait `/a/b/c` .

L'instruction `WORKDIR` peut résoudre les variables d'environnement précédemment définies à l'aide d' `ENV` . Vous ne pouvez utiliser que des variables d'environnement explicitement définies dans le `Dockerfile` . Par exemple:

```
ENV DIRPATH /path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
```

La sortie de la commande `pwd` finale dans ce fichier Docker serait `/path/$DIRNAME`

Instruction VOLUME

```
VOLUME ["/data"]
```

L'instruction `VOLUME` crée un point de montage avec le nom spécifié et le marque comme contenant des volumes montés en externe à partir d'un hôte natif ou d'autres conteneurs. La valeur peut être un tableau JSON, `VOLUME ["/var/log/"]` ou une chaîne simple avec plusieurs arguments, tels que `VOLUME /var/log` ou `VOLUME /var/log /var/db` . Pour plus d'informations / exemples et instructions de montage via le client Docker, reportez-vous à la documentation sur les répertoires partagés via des volumes.

La commande `docker run` initialise le volume nouvellement créé avec toutes les données existant à l'emplacement spécifié dans l'image de base. Par exemple, considérez l'extrait de fichier Dockerfile suivant:

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

Ce fichier Dockerfile génère une image qui provoque l'exécution du menu fixe, pour créer un nouveau point de montage sur / myvol et copier le fichier de message d'accueil dans le volume nouvellement créé.

Remarque: Si des étapes de génération modifient les données dans le volume après sa

déclaration, ces modifications seront ignorées.

Remarque: La liste est analysée en tant que tableau JSON, ce qui signifie que vous devez utiliser des guillemets (“) autour des mots et non des guillemets simples (').

Instruction COPY

`COPY` a deux formes:

```
COPY <src>... <dest>
COPY ["<src>",... "<dest>"] (this form is required for paths containing whitespace)
```

L'instruction `COPY` copie les nouveaux fichiers ou répertoires de `<src>` et les ajoute au système de fichiers du conteneur sur le chemin `<dest>` .

Plusieurs ressources `<src>` peuvent être spécifiées, mais elles doivent être relatives au répertoire source en cours de construction (contexte de la génération).

Chaque `<src>` peut contenir des caractères génériques et la correspondance sera effectuée à l'aide des règles du `filepath.Match` de `filepath.Match` de Go. Par exemple:

```
COPY hom* /mydir/        # adds all files starting with "hom"
COPY hom?.txt /mydir/    # ? is replaced with any single character, e.g., "home.txt"
```

Le `<dest>` est un chemin absolu ou un chemin relatif à `WORKDIR` dans lequel la source sera copiée dans le conteneur de destination.

```
COPY test relativeDir/   # adds "test" to `WORKDIR`/relativeDir/
COPY test /absoluteDir/  # adds "test" to /absoluteDir/
```

Tous les nouveaux fichiers et répertoires sont créés avec un UID et un GID de 0.

Remarque: Si vous générez à l'aide de stdin (`docker build - < somefile`), il n'y a pas de contexte de génération, `COPY` ne peut donc pas être utilisé.

`COPY` obéit aux règles suivantes:

- Le chemin `<src>` doit se trouver dans le contexte de la génération; vous ne pouvez pas `COPY` ../squelque chose / quelque chose, car la première étape de la construction d'un menu fixe consiste à envoyer le répertoire de contexte (et ses sous-répertoires) au démon docker.
- Si `<src>` est un répertoire, tout le contenu du répertoire est copié, y compris les métadonnées du système de fichiers. Remarque: le répertoire lui-même n'est pas copié, mais uniquement son contenu.
- Si `<src>` est un autre type de fichier, il est copié individuellement avec ses métadonnées. Dans ce cas, si `<dest>` se termine par une barre oblique /, il sera considéré comme un répertoire et le contenu de `<src>` sera écrit dans `<dest>/base(<src>)` .

- Si plusieurs ressources `<src>` sont spécifiées, que ce soit directement ou en raison de l'utilisation d'un caractère générique, alors `<dest>` doit être un répertoire et doit se terminer par une barre oblique `/` .
- Si `<dest>` ne se termine pas par une barre oblique, elle sera considérée comme un fichier normal et le contenu de `<src>` sera écrit sur `<dest>` .
- Si `<dest>` n'existe pas, il est créé avec tous les répertoires manquants sur son chemin.

Instruction ENV et ARG

ENV

```
ENV <key> <value>
ENV <key>=<value> ...
```

L'instruction `ENV` définit la variable d'environnement `<key>` sur la valeur. Cette valeur sera dans l'environnement de toutes les commandes Dockerfile «descendantes» et peut également être remplacée en ligne.

L'instruction `ENV` a deux formes. Le premier formulaire, `ENV <key> <value>` , définira une variable unique sur une valeur. La chaîne entière après le premier espace sera traitée comme la `<value>` - y compris les caractères tels que les espaces et les guillemets.

La deuxième forme, `ENV <key>=<value> ...` , permet de définir plusieurs variables à la fois. Notez que la seconde forme utilise le signe égal (=) dans la syntaxe, alors que la première forme ne le fait pas. Comme pour l'analyse de ligne de commande, les guillemets et les barres obliques inverses peuvent être utilisés pour inclure des espaces dans les valeurs.

Par exemple:

```
ENV myName="John Doe" myDog=Rex\ The\ Dog \
    myCat=fluffy
```

et

```
ENV myName John Doe
ENV myDog Rex The Dog
ENV myCat fluffy
```

produira les mêmes résultats nets dans le conteneur final, mais la première forme est préférable car elle produit une seule couche de cache.

Les variables d'environnement définies à l'aide de `ENV` persisteront lorsqu'un conteneur est exécuté à partir de l'image résultante. Vous pouvez afficher les valeurs à l'aide de `docker run --env <key>=<value>` et les modifier à l'aide de `docker run --env <key>=<value>` .

ARG

Si vous ne souhaitez pas conserver le paramètre, utilisez plutôt `ARG` . `ARG` définira les environnements uniquement pendant la construction. Par exemple, paramètre

```
ENV DEBIAN_FRONTEND noninteractive
```

peut confondre les utilisateurs d' `apt-get` sur une image basée sur Debian quand ils entrent dans le conteneur dans un contexte interactif via `docker exec -it the-container bash` .

Au lieu de cela, utilisez:

```
ARG DEBIAN_FRONTEND noninteractive
```

Vous pouvez également définir une valeur pour une seule commande uniquement en utilisant:

```
RUN <key>=<value> <command>
```

Instruction EXPOSE

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

L'instruction `EXPOSE` informe Docker que le conteneur écoute les ports réseau spécifiés au moment de l'exécution. `EXPOSE` ne rend pas les ports du conteneur accessibles à l'hôte. Pour ce faire, vous devez utiliser l' `-p` pour publier une plage de ports ou l' `-P` pour publier tous les ports exposés. Ces indicateurs sont utilisés dans le `docker run [OPTIONS] IMAGE [COMMAND][ARG...]` pour exposer le port à l'hôte. Vous pouvez exposer un numéro de port et le publier en externe sous un autre numéro.

```
docker run -p 2500:80 <image name>
```

Cette commande crée un conteneur portant le nom <image> et lie le port 80 du conteneur au port 2500 de la machine hôte.

Pour configurer la redirection de port sur le système hôte, consultez l' `-P` . La fonctionnalité réseau Docker prend en charge la création de réseaux sans avoir à exposer les ports du réseau. Pour plus d'informations, consultez la présentation de cette fonctionnalité.

Instruction LABEL

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

L'instruction `LABEL` ajoute des métadonnées à une image. Un `LABEL` est une paire clé-valeur. Pour inclure des espaces dans une valeur `LABEL` , utilisez des guillemets et des barres obliques inverses comme vous le feriez dans l'analyse de ligne de commande. Quelques exemples d'utilisation:

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

Une image peut avoir plusieurs étiquettes. Pour spécifier plusieurs étiquettes, Docker recommande de combiner les étiquettes dans une seule instruction `LABEL` si possible. Chaque instruction `LABEL` produit un nouveau calque qui peut entraîner une image inefficace si vous utilisez plusieurs étiquettes. Cet exemple se traduit par une couche d'image unique.

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

Ce qui précède peut aussi être écrit comme suit:

```
LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"
```

Les étiquettes sont additives, y compris les `LABEL` dans les images `FROM` . Si Docker rencontre une étiquette / clé qui existe déjà, la nouvelle valeur remplace toutes les étiquettes précédentes avec des clés identiques.

Pour afficher les étiquettes d'une image, utilisez la commande docker inspect.

```
"Labels": {
    "com.example.vendor": "ACME Incorporated"
    "com.example.label-with-value": "foo",
    "version": "1.0",
    "description": "This text illustrates that label-values can span multiple lines.",
    "multi.label1": "value1",
    "multi.label2": "value2",
    "other": "value3"
},
```

Instruction CMD

L'instruction `CMD` a trois formes:

```
CMD ["executable","param1","param2"] (exec form, this is the preferred form)
CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
CMD command param1 param2 (shell form)
```

Il ne peut y avoir qu'une seule instruction `CMD` dans un `Dockerfile` . Si vous listez plusieurs `CMD` seul le dernier `CMD` prendra effet.

Le but principal d'un `CMD` est de fournir des valeurs par défaut pour un conteneur en cours d'exécution. Ces valeurs par défaut peuvent inclure un exécutable ou omettre l'exécutable. Dans ce cas, vous devez également spécifier une instruction `ENTRYPOINT` .

Remarque: Si `CMD` est utilisé pour fournir des arguments par défaut pour l'instruction `ENTRYPOINT`

instructions `CMD` et `ENTRYPOINT` doivent être spécifiées avec le format de tableau JSON.

Remarque: Le formulaire d'exécution est analysé en tant que tableau JSON, ce qui signifie que vous devez utiliser des guillemets (“) autour des mots et non des guillemets simples (').

Remarque: Contrairement au formulaire shell, le formulaire exec n'invoque pas un shell de commandes. Cela signifie que le traitement normal du shell ne se produit pas. Par exemple, `CMD [ "echo", "$HOME" ]` ne fera pas de substitution de variable sur `$HOME` . Si vous voulez un traitement du shell, utilisez soit la forme du shell, soit exécutez un shell directement, par exemple: `CMD [ "sh", "-c", "echo $HOME" ]` .

Lorsqu'elle est utilisée dans les formats shell ou exec, l'instruction `CMD` définit la commande à exécuter lors de l'exécution de l'image.

Si vous utilisez la forme shell du `CMD` , la commande s'exécutera dans `/bin/sh -c` :

```
FROM ubuntu
CMD echo "This is a test." | wc -
```

Si vous voulez exécuter votre commande sans shell, vous devez exprimer la commande en tant que tableau JSON et donner le chemin d'accès complet à l'exécutable. Cette forme de tableau est le format préféré de `CMD` . Tout paramètre supplémentaire doit être exprimé individuellement sous la forme de chaînes dans le tableau:

```
FROM ubuntu
CMD ["/usr/bin/wc","--help"]
```

Si vous souhaitez que votre conteneur exécute le même exécutable à chaque fois, vous devriez envisager d'utiliser `ENTRYPOINT` en combinaison avec `CMD` . Voir le point d' `ENTRYPOINT` .

Si l'utilisateur spécifie des arguments pour exécuter le menu fixe, il remplacera la valeur par défaut spécifiée dans `CMD` .

Remarque: ne confondez pas `RUN` avec `CMD` . `RUN` exécute une commande au moment de la création de l'image et valide le résultat. `CMD` n'exécute rien au moment de la construction, mais spécifie la commande prévue pour l'image.

Instruction MAINTAINER

```
MAINTAINER <name>
```

L'instruction `MAINTAINER` vous permet de définir le champ Auteur des images générées.

N'UTILISEZ PAS LA DIRECTIVE MAINTAINER

Selon la documentation officielle de Docker, l'instruction `MAINTAINER` est obsolète. Au lieu de cela, on devrait utiliser l'instruction `LABEL` pour définir l'auteur des images générées. L'instruction `LABEL` est plus flexible, permet de définir des métadonnées et peut être facilement visualisée avec le `docker inspect` commandes `docker inspect` .

```
LABEL maintainer="someone@something.com"
```

À partir de l'instruction

```
FROM <image>
```

Ou

```
FROM <image>:<tag>
```

Ou

```
FROM <image>@<digest>
```

L'instruction `FROM` définit l'image de base pour les instructions suivantes. En tant que tel, un Dockerfile valide doit avoir `FROM` comme première instruction. L'image peut être n'importe quelle image valide - il est particulièrement facile de commencer en tirant une image des référentiels publics.

`FROM` doit être la première instruction de non-commentaire du fichier Dockerfile.

`FROM` peut apparaître plusieurs fois dans un seul fichier Dockerfile afin de créer plusieurs images. Notez simplement le dernier identifiant d'image généré par le commit avant chaque nouvelle commande `FROM` .

La balise ou les valeurs de résumé sont facultatives. Si vous omettez l'un ou l'autre, le générateur prend la dernière valeur par défaut. Le générateur renvoie une erreur s'il ne peut pas correspondre à la valeur de la balise.

Instruction RUN

`RUN` a 2 formes:

```
RUN <command> (shell form, the command is run in a shell, which by default is /bin/sh -c on
Linux or cmd /S /C on Windows)
RUN ["executable", "param1", "param2"] (exec form)
```

L'instruction `RUN` exécute toutes les commandes d'un nouveau calque au-dessus de l'image en cours et valide les résultats. L'image `Dockerfile` résultante sera utilisée pour l'étape suivante du `Dockerfile` .

La superposition des instructions `RUN` et la génération de commits sont conformes aux concepts de base de Docker où les commits sont peu coûteux et les conteneurs peuvent être créés à partir de n'importe quel point de l'historique d'une image, un peu comme le contrôle de source.

La forme exec permet d'éviter munging de chaîne de coquille, et à `RUN` commandes à l' aide d' une image de base qui ne contient pas le fichier exécutable d'enveloppe spécifiée.

Le shell par défaut du formulaire shell peut être modifié à l'aide de la commande `SHELL` .

Dans la forme du shell, vous pouvez utiliser un `\` (backslash) pour continuer une seule instruction `RUN` sur la ligne suivante. Par exemple, considérons ces deux lignes:

```
RUN /bin/bash -c 'source $HOME/.bashrc ;\
echo $HOME'
```

Ensemble, ils sont équivalents à cette seule ligne:

```
RUN /bin/bash -c 'source $HOME/.bashrc ; echo $HOME'
```

Remarque: Pour utiliser un shell différent de «/ bin / sh», utilisez le formulaire exec en passant dans le shell souhaité. Par exemple, `RUN ["/bin/bash", "-c", "echo hello"]`

Remarque: Le formulaire d'exécution est analysé en tant que tableau JSON, ce qui signifie que vous devez utiliser des guillemets (`"`) autour des mots et non des guillemets simples (`'`).

Remarque: Contrairement au formulaire shell, le formulaire exec n'invoque pas un shell de commandes. Cela signifie que le traitement normal du shell ne se produit pas. Par exemple, `RUN [ "echo", "$HOME" ]` ne fera pas de substitution de variable sur `$HOME` . Si vous voulez un traitement du shell, utilisez soit le shell, soit exécutez directement un shell, par exemple: `RUN [ "sh", "-c", "echo $HOME" ]` .

Remarque: Dans le formulaire JSON, il est nécessaire d'échapper les barres obliques inverses. Ceci est particulièrement pertinent sous Windows où la barre oblique inverse est le séparateur de chemin. La ligne suivante serait sinon traitée en tant que formulaire shell en raison de ne pas être JSON valide, et échouer de manière inattendue: `RUN ["c:\windows\system32\tasklist.exe"]`

La syntaxe correcte pour cet exemple est la suivante: `RUN ["c:\\windows\\system32\\tasklist.exe"]`

Le cache pour les instructions `RUN` n'est pas automatiquement invalidé lors de la prochaine génération. Le cache pour une instruction comme `RUN apt-get dist-upgrade -y` sera réutilisé lors de la prochaine génération. Le cache pour les instructions `RUN` peut être invalidé à l'aide de l'indicateur --no-cache, par exemple docker build --no-cache.

Reportez-vous au guide Dockerfile Best Practices pour plus d'informations.

Le cache pour les instructions `RUN` peut être invalidé par les instructions `ADD` . Voir ci-dessous pour plus de détails.

Instruction ONBUILD

```
ONBUILD [INSTRUCTION]
```

L'instruction `ONBUILD` ajoute à l'image une instruction de déclenchement à exécuter ultérieurement, lorsque l'image est utilisée comme base pour une autre version. Le déclencheur sera exécuté dans le contexte de la construction en aval, comme s'il avait été inséré immédiatement après

l'instruction `FROM` dans le fichier Dockerfile en aval.

Toute instruction de construction peut être enregistrée en tant que déclencheur.

Ceci est utile si vous créez une image qui sera utilisée comme base pour construire d'autres images, par exemple un environnement de génération d'application ou un démon qui peut être personnalisé avec une configuration spécifique à l'utilisateur.

Par exemple, si votre image est un générateur d'application Python réutilisable, le code source de l'application doit être ajouté dans un répertoire particulier et un script de génération peut ensuite être appelé. Vous ne pouvez pas simplement appeler `ADD` et `RUN` maintenant, car vous n'avez pas encore accès au code source de l'application, et il sera différent pour chaque génération d'application. Vous pouvez simplement fournir aux développeurs d'applications avec un Dockerfile standard pour copier-coller dans leur application, mais cela est inefficace, sujet aux erreurs et difficile à mettre à jour car il se mélange au code spécifique à l'application.

La solution consiste à utiliser `ONBUILD` pour enregistrer les instructions à exécuter ultérieurement, lors de la prochaine étape de la génération.

Voici comment cela fonctionne:

Lorsqu'il rencontre une instruction `ONBUILD` , le générateur ajoute un déclencheur aux métadonnées de l'image en cours de création. L'instruction n'affecte pas la construction en cours.

À la fin de la construction, une liste de tous les déclencheurs est stockée dans le manifeste de l'image, sous la clé OnBuild. Ils peuvent être inspectés avec la commande `docker inspect` . Plus tard, l'image peut être utilisée comme base pour une nouvelle construction, en utilisant l'instruction `FROM` . Dans le cadre du traitement de l'instruction `FROM` , le générateur en aval recherche les déclencheurs `ONBUILD` et les exécute dans l'ordre dans `ONBUILD` ils ont été enregistrés. Si l'un des déclencheurs échoue, l'instruction `FROM` est abandonnée, ce qui entraîne l'échec de la génération. Si tous les déclencheurs réussissent, l'instruction `FROM` termine et la construction se poursuit normalement.

Les déclencheurs sont effacés de l'image finale après leur exécution. En d'autres termes, ils ne sont pas hérités par les constructions de «petits-enfants».

Par exemple, vous pourriez ajouter quelque chose comme ceci:

```
[...]
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

Avertissement: le chaînage des instructions `ONBUILD` utilisant `ONBUILD ONBUILD` n'est pas autorisé.

Avertissement: l'instruction `ONBUILD` ne peut pas déclencher d'instructions `FROM` ou `MAINTAINER` .

Instruction STOPSIGNAL

```
STOPSIGNAL signal
```

L'instruction `STOPSIGNAL` définit le signal d'appel système qui sera envoyé au conteneur pour quitter. Ce signal peut être un nombre non signé valide correspondant à une position dans la table syscall du noyau, par exemple 9, ou un nom de signal au format SIGNAME, par exemple SIGKILL.

HEALTHCHECK Instruction

L'instruction `HEALTHCHECK` a deux formes:

```
HEALTHCHECK [OPTIONS] CMD command (check container health by running a command inside the
container)
HEALTHCHECK NONE (disable any healthcheck inherited from the base image)
```

L'instruction `HEALTHCHECK` indique à Docker comment tester un conteneur pour vérifier qu'il fonctionne toujours. Cela peut détecter des cas tels qu'un serveur Web bloqué dans une boucle infinie et incapable de gérer de nouvelles connexions, même si le processus du serveur est toujours en cours d'exécution.

Lorsqu'un contrôle d'intégrité est spécifié pour un conteneur, son état d'intégrité s'ajoute à son statut normal. Ce statut commence initialement. Chaque fois qu'un bilan de santé passe, il devient sain (quel que soit l'état dans lequel il se trouvait auparavant). Après un certain nombre d'échecs consécutifs, cela devient malsain.

Les options pouvant apparaître avant `CMD` sont les suivantes:

```
--interval=DURATION (default: 30s)
--timeout=DURATION (default: 30s)
--retries=N (default: 3)
```

Le bilan de santé commence par un intervalle de secondes après le démarrage du conteneur, puis à nouveau quelques secondes après la fin de chaque vérification précédente.

Si une seule exécution de la vérification dure plus longtemps que le délai d'attente, la vérification est considérée comme ayant échoué.

Il faut réessayer les échecs consécutifs du bilan de santé pour que le conteneur soit considéré comme malsain.

Il ne peut y avoir qu'une `HEALTHCHECK` instruction `HEALTHCHECK` dans un `Dockerfile` . Si vous en `HEALTHCHECK` plus d'un, seul le dernier `HEALTHCHECK` prendra effet.

La commande après le mot-clé `CMD` peut être une commande shell (par exemple, `HEALTHCHECK CMD /bin/check-running`) ou un tableau exec (comme pour les autres commandes Dockerfile; voir par exemple `ENTRYPOINT` pour plus de détails).

Le statut de sortie de la commande indique l'état d'intégrité du conteneur. Les valeurs possibles sont:

- `0: success`

- le conteneur est sain et prêt à l'emploi
- `1: unhealthy` - le conteneur ne fonctionne pas correctement
- `2: starting` - le conteneur n'est pas encore prêt à être utilisé, mais fonctionne correctement

Si la sonde renvoie 2 («commençant») lorsque le conteneur est déjà sorti de l'état «de démarrage», il est traité comme «non sain».

Par exemple, pour vérifier toutes les cinq minutes ou pour qu'un serveur Web puisse diffuser la page principale du site en trois secondes:

```
HEALTHCHECK --interval=5m --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

Pour aider à déboguer les sondes défaillantes, tout texte de sortie (encodé en UTF-8) que la commande écrit sur stdout ou stderr sera stocké dans l'état de santé et peut être interrogé avec `docker inspect` . Une telle sortie doit rester courte (seuls les premiers 4096 octets sont stockés actuellement).

Lorsque l'état de santé d'un conteneur change, un événement `health_status` est généré avec le nouveau statut.

La fonctionnalité `HEALTHCHECK` été ajoutée dans Docker 1.12.

Instruction SHELL

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

L'instruction `SHELL` permet de `SHELL` le shell par défaut utilisé pour les commandes de type shell. Le shell par défaut sous Linux est `["/bin/sh", "-c"]` et sous Windows, `["cmd", "/S", "/C"]` .
L'instruction `SHELL` doit être écrite sous forme JSON dans un fichier Docker.

L'instruction `SHELL` est particulièrement utile sous Windows où il y a deux shells natifs couramment utilisés et très différents: cmd et powershell, ainsi que des shells alternatifs disponibles, y compris sh.

L'instruction `SHELL` peut apparaître plusieurs fois. Chaque instruction `SHELL` remplace toutes les instructions `SHELL` précédentes et affecte toutes les instructions suivantes. Par exemple:

```
FROM windowsservercore

# Executed as cmd /S /C echo default
RUN echo default

# Executed as cmd /S /C powershell -command Write-Host default
RUN powershell -command Write-Host default

# Executed as powershell -command Write-Host hello
SHELL ["powershell", "-command"]
RUN Write-Host hello

# Executed as cmd /S /C echo hello
```

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

Les instructions suivantes peuvent être affectées par l'instruction `SHELL` lorsque leur forme de shell est utilisée dans un fichier Dockerfile: `RUN` , `CMD` et `ENTRYPOINT` .

L'exemple suivant est un modèle commun trouvé sous Windows, qui peut être rationalisé à l'aide de l'instruction `SHELL` :

```
...
RUN powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
...
```

La commande appelée par docker sera:

```
cmd /S /C powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
```

Ceci est inefficace pour deux raisons. Tout d'abord, un processeur de commandes cmd.exe (aka shell) non nécessaire est appelé. Deuxièmement, chaque instruction `RUN` de la forme shell nécessite une commande powershell supplémentaire préfixant la commande.

Pour rendre cela plus efficace, l'un des deux mécanismes peut être utilisé. L'une consiste à utiliser la forme JSON de la commande `RUN` telle que:

```
...
RUN ["powershell", "-command", "Execute-MyCmdlet", "-param1 \"c:\\foo.txt\""]
...
```

Bien que le formulaire JSON ne soit pas ambigu et n'utilise pas le fichier cmd.exe inutile, il nécessite plus de verbosité en citant et en échappant. L'autre mécanisme consiste à utiliser l'instruction `SHELL` et la forme du shell, ce qui rend la syntaxe plus naturelle pour les utilisateurs Windows, en particulier lorsqu'elle est associée à la directive d'échappement parser:

```
# escape=`

FROM windowsservercore
SHELL ["powershell","-command"]
RUN New-Item -ItemType Directory C:\Example
ADD Execute-MyCmdlet.ps1 c:\example\
RUN c:\example\Execute-MyCmdlet -sample 'hello world'
```

Résultant en:

```
PS E:\docker\build\shell> docker build -t shell .
Sending build context to Docker daemon 3.584 kB
Step 1 : FROM windowsservercore
 ---> 5bc36a335344
Step 2 : SHELL powershell -command
 ---> Running in 87d7a64c9751
 ---> 4327358436c1
Removing intermediate container 87d7a64c9751
```

```
Step 3 : RUN New-Item -ItemType Directory C:\Example
 ---> Running in 3e6ba16b8df9


Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         6/2/2016   2:59 PM                Example


 ---> 1f1dfdcec085
Removing intermediate container 3e6ba16b8df9
Step 4 : ADD Execute-MyCmdlet.ps1 c:\example\
 ---> 6770b4c17f29
Removing intermediate container b139e34291dc
Step 5 : RUN c:\example\Execute-MyCmdlet -sample 'hello world'
 ---> Running in abdcf50dfd1f
Hello from Execute-MyCmdlet.ps1 - passed hello world
 ---> ba0e25255fda
Removing intermediate container abdcf50dfd1f
Successfully built ba0e25255fda
PS E:\docker\build\shell>
```

L'instruction `SHELL` pourrait également être utilisée pour modifier la manière dont fonctionne un shell. Par exemple, en utilisant `SHELL cmd /S /C /V:ON|OFF` sous Windows, la sémantique d'extension de variable d'environnement retardée pourrait être modifiée.

L'instruction `SHELL` peut également être utilisée sous Linux si un autre shell est requis, tel que zsh, csh, tcsh et autres.

La fonctionnalité `SHELL` été ajoutée dans Docker 1.12.

Installer des paquets Debian / Ubuntu

Exécutez l'installation sur une seule commande d'exécution pour fusionner la mise à jour et installer. Si vous ajoutez plus de paquetages ultérieurement, cela lancera à nouveau la mise à jour et installera tous les paquetages nécessaires. Si la mise à jour est exécutée séparément, elle sera mise en cache et les installations du package risquent d'échouer. La définition de l'interface sur non-interactive et la transmission de l'option -y à l'installation est nécessaire pour les installations par script. Le nettoyage et la purge à la fin de l'installation minimisent la taille de la couche.

```
FROM debian

RUN apt-get update \
 && DEBIAN_FRONTEND=noninteractive apt-get install -y \
    git \
    openssh-client \
    sudo \
    vim \
    wget \
 && apt-get clean \
 && rm -rf /var/lib/apt/lists/*
```

Lire Dockerfiles en ligne: https://riptutorial.com/fr/docker/topic/3161/dockerfiles

Chapitre 16: Enregistrement

Examples

Configuration d'un pilote de journal dans le service systemd

```
[Service]

# empty exec prevents error "docker.service has more than one ExecStart= setting, which is
only allowed for Type=oneshot services. Refusing."
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// --log-driver=syslog
```

Cela permet la journalisation syslog pour le démon docker. Le fichier doit être créé dans le répertoire approprié avec la racine propriétaire, qui serait généralement `/etc/systemd/system/docker.service.d` , par exemple sur Ubuntu 16.04.

Vue d'ensemble

L'approche de Docker en matière de journalisation consiste à construire vos conteneurs de manière à ce que les journaux soient écrits sur la sortie standard (console / terminal).

Si vous avez déjà un conteneur qui écrit des journaux dans un fichier, vous pouvez le rediriger en créant un lien symbolique:

```
ln -sf /dev/stdout /var/log/nginx/access.log
ln -sf /dev/stderr /var/log/nginx/error.log
```

Après cela, vous pouvez utiliser différents pilotes de journal pour placer vos journaux là où vous en avez besoin.

Lire Enregistrement en ligne: https://riptutorial.com/fr/docker/topic/7378/enregistrement

Chapitre 17: Événements Docker

Examples

Lancer un conteneur et être informé des événements associés

La documentation des `docker events` fournit des détails, mais lors du débogage, il peut être utile de lancer un conteneur et d'être immédiatement informé de tout événement lié:

```
docker run... & docker events --filter 'container=$(docker ps -lq)'
```

Dans `docker ps -lq` , le `l` correspond au `last` et le `q` au `quiet` . Cela supprime l' `id` du dernier conteneur lancé et crée une notification immédiatement si le conteneur meurt ou si un autre événement se produit.

Lire Événements Docker en ligne: https://riptutorial.com/fr/docker/topic/6200/evenements-docker

Chapitre 18: exécuter consul dans docker 1.12 essaim

Examples

Courir consul dans un docker 1.12 essaim

Cela repose sur l'image officielle du docker consul à exécuter consul en mode cluster dans un essaim de docker avec un nouveau mode essaim dans Docker 1.12. Cet exemple est basé sur http://qnib.org/2016/08/11/consul-service/ . En bref, l'idée est d'utiliser deux services d'essaims de dockers qui se parlent. Cela résout le problème que vous ne pouvez pas connaître les ips des conteneurs de consul individuels à l'avant et que vous pouvez vous fier aux dns de Docker Swarm.

Cela suppose que vous avez déjà un cluster docker 1.12 avec au moins trois nœuds.

Vous souhaiterez peut-être configurer un pilote de journal sur vos démons de docker pour pouvoir voir ce qui se passe. J'ai utilisé le pilote syslog pour cela: définissez l' `--log-driver=syslog` sur dockerd.

Créez d'abord un réseau de recouvrement pour le consul:

```
docker network create consul-net -d overlay
```

Maintenant, démarrez le cluster avec un seul nœud (par défaut --replicas est 1):

```
docker service create --name consul-seed \
  -p 8301:8300 \
  --network consul-net \
  -e 'CONSUL_BIND_INTERFACE=eth0' \
  consul agent -server -bootstrap-expect=3  -retry-join=consul-seed:8301 -retry-join=consul-
cluster:8300
```

Vous devriez maintenant avoir un cluster à 1 nœud. Maintenant, ouvrez le deuxième service:

```
docker service create --name consul-cluster \
  -p 8300:8300 \
  --network consul-net \
  --replicas 3 \
  -e 'CONSUL_BIND_INTERFACE=eth0' \
  consul agent -server -retry-join=consul-seed:8301 -retry-join=consul-cluster:8300
```

Vous devriez maintenant avoir un cluster de consul à quatre nœuds. Vous pouvez le vérifier en exécutant l'un des conteneurs Docker:

```
docker exec <containerid> consul members
```

Lire exécuter consul dans docker 1.12 essaim en ligne:
https://riptutorial.com/fr/docker/topic/6437/executer-consul-dans-docker-1-12-essaim

Chapitre 19: Exécution de l'application Simple Node.js

Examples

Exécution d'une application Basic Node.js dans un conteneur

L'exemple que je vais aborder suppose que vous avez une installation Docker qui fonctionne dans votre système et une compréhension de base de la façon de travailler avec Node.js. Si vous savez comment travailler avec Docker, il est évident que le framework Node.js ne doit pas nécessairement être installé sur votre système. Nous utiliserons plutôt la `latest` version de l'image de `node` disponible auprès de Docker. Par conséquent, si nécessaire, vous pouvez télécharger l'image au préalable avec le `docker pull node` commande `docker pull node` . (La commande `pulls` automatiquement la dernière version de l'image de `node` depuis le menu fixe.)

1. Procédez à la création d'un répertoire où tous vos fichiers d'application de travail résideraient. Créez un fichier `package.json` dans ce répertoire qui décrit votre application ainsi que les dépendances. Votre fichier `package.json` devrait ressembler à ceci:

```
{
  "name": "docker_web_app",
  "version": "1.0.0",
  "description": "Node.js on Docker",
  "author": "First Last <first.last@example.com>",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.13.3"
  }
}
```

2. Si nous devons travailler avec Node.js, nous créons généralement un fichier de `server` qui définit une application Web. Dans ce cas, nous utilisons le framework `Express.js` (version `4.13.3`). Un fichier `server.js` base ressemblerait à ceci:

```
var express = require('express');
var PORT = 8080;
var app = express();
app.get('/', function (req, res) {
  res.send('Hello world\n');
});

app.listen(PORT);
console.log('Running on http://localhost:' + PORT);
```

3. Pour ceux qui connaissent Docker, vous auriez rencontré un `Dockerfile` . Un `Dockerfile` est un fichier texte contenant toutes les commandes requises pour créer une image

personnalisée adaptée à votre application.

Créez un fichier texte vide nommé `Dockerfile` dans le répertoire en cours. La méthode pour en créer un est simple dans Windows. Sous Linux, vous souhaiterez peut-être exécuter `touch Dockerfile` dans le répertoire contenant tous les fichiers requis pour votre application. Ouvrez le fichier Dockerfile avec n'importe quel éditeur de texte et ajoutez les lignes suivantes:

```
FROM node:latest
RUN mkdir -p /usr/src/my_first_app
WORKDIR /usr/src/my_first_app
COPY package.json /usr/src/my_first_app/
RUN npm install
COPY . /usr/src/my_first_app
EXPOSE 8080
```

- `FROM node:latest` indique au démon Docker quelle image nous voulons générer. Dans ce cas, nous utilisons la `latest` version du `node` image Docker officiel disponible sur le Docker Hub .
- Dans cette image, nous procédons à la création d'un répertoire de travail contenant tous les fichiers requis et nous demandons au démon de définir ce répertoire comme répertoire de travail souhaité pour notre application. Pour cela nous ajoutons

  ```
  RUN mkdir -p /usr/src/my_first_app
  WORKDIR /usr/src/my_first_app
  ```

- Nous procédons ensuite à l'installation des dépendances de l'application en déplaçant d'abord le fichier `package.json` (qui spécifie les informations d'application, y compris les dépendances) dans le `/usr/src/my_first_app` de l'image. Nous faisons cela par

  ```
  COPY package.json /usr/src/my_first_app/
  RUN npm install
  ```

- Nous tapez ensuite `COPY . /usr/src/my_first_app` pour ajouter tous les fichiers d'application et le code source au répertoire de travail de l'image.
- Nous utilisons ensuite la directive `EXPOSE` pour demander au démon de rendre visible le port `8080` du conteneur résultant (via un mappage de conteneur à hôte) puisque l'application est liée au port `8080` .
- Au cours de la dernière étape, nous demandons au démon d'exécuter la commande `node server.js` dans l'image en exécutant la commande de base `npm start` . Nous utilisons la directive `CMD` pour cela, qui prend les commandes comme arguments.

  ```
  CMD [ "npm", "start" ]
  ```

4. Nous créons ensuite un fichier `.dockerignore` dans le même répertoire que le `Dockerfile` pour empêcher que notre copie de `node_modules` et de journaux utilisés par notre installation système Node.js ne soit copiée sur l'image Docker. Le fichier `.dockerignore` doit avoir le contenu suivant:

```
node_modules
npm-debug.log
```

5. # Construisez votre image

Naviguez jusqu'au répertoire contenant le `Dockerfile` et exécutez la commande suivante pour créer l'image Docker. L' `-t` vous permet de baliser votre image pour la retrouver plus facilement à l'aide de la commande docker images:

```
    $ docker build -t <your username>/node-web-app .
```

Votre image sera maintenant répertoriée par Docker. Visualisez les images en utilisant la commande ci-dessous:

```
$ docker images

REPOSITORY                         TAG        ID              CREATED
node                               latest     539c0211cd76    10 minutes ago
<your username>/node-web-app       latest     d64d3505b0d2    1 minute ago
```

6. # Lancer l'image

Nous pouvons maintenant exécuter l'image que nous venons de créer en utilisant le contenu de l'application, l'image de base du `node` et le `Dockerfile` . Nous allons maintenant exécuter notre image nouvellement créée `<your username>/node-web-app` . Fournir l'option `-d` à la commande `docker run` exécute le conteneur en mode détaché, de sorte que le conteneur s'exécute en arrière-plan. L' `-p` redirige un port public vers un port privé à l'intérieur du conteneur. Exécutez l'image que vous avez précédemment créée à l'aide de cette commande:

```
$ docker run -p 49160:8080 -d <your username>/node-web-app
```

7. Imprimez la sortie de votre application en exécutant `docker ps` sur votre terminal. La sortie devrait ressembler à ceci.

```
 CONTAINER ID        IMAGE                          COMMAND             CREATED
STATUS              PORTS                      NAMES
 7b701693b294      <your username>/node-web-app   "npm start"         20 minutes ago
Up 48 seconds       0.0.0.0:49160->8080/tcp    loving_goldstine
```

Obtenez la sortie de l'application en saisissant les `docker logs <CONTAINER ID>` . Dans ce cas, il s'agit des `docker logs 7b701693b294` .

Sortie: en `Running on http://localhost:8080`

8. À partir de la sortie du `docker ps` , le mappage de port obtenu est `0.0.0.0:49160->8080/tcp` . Docker a donc mappé le port `8080` intérieur du conteneur sur le port 49160 de la machine

hôte. Dans le navigateur, nous pouvons maintenant entrer `localhost:49160` .

Nous pouvons également appeler notre application en utilisant `curl` :

```
$ curl -i localhost:49160

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 12
Date: Sun, 08 Jan 2017 14:00:12 GMT
Connection: keep-alive

Hello world
```

Lire Exécution de l'application Simple Node.js en ligne:
https://riptutorial.com/fr/docker/topic/8754/execution-de-l-application-simple-node-js

Chapitre 20: Gérer des images

Syntaxe

- images de docker [OPTIONS] [REPOSITORY [: TAG]]
- docker inspect [OPTIONS] CONTENEUR | IMAGE [CONTENEUR | IMAGE ...]
- docker pull [OPTIONS] NAME [: TAG | @DIGEST]
- docker rmi [OPTIONS] IMAGE [IMAGE ...]
- tag docker [OPTIONS] IMAGE [: TAG] [REGISTRYHOST /] [NOM D'UTILISATEUR /] NOM [: TAG]

Examples

Récupérer une image depuis Docker Hub

Normalement, les images sont extraites automatiquement de Docker Hub . Docker tentera d'extraire n'importe quelle image de Docker Hub qui n'existe pas déjà sur l'hôte Docker. Par exemple, si vous utilisez `docker run ubuntu` lorsque l'image d' `ubuntu` n'est pas déjà sur l'hôte Docker, Docker déclenchera la dernière image `ubuntu` . Il est possible de tirer une image séparément en utilisant Docker `docker pull` pour `docker pull` ou mettre à jour manuellement une image depuis Docker Hub.

```
docker pull ubuntu
docker pull ubuntu:14.04
```

Des options supplémentaires pour extraire un registre d'images différent ou extraire une version spécifique d'une image existent. Indiquant qu'un autre registre est effectué à l'aide du nom complet de l'image et de la version facultative. Par exemple, la commande suivante tente d'extraire l'image `ubuntu:14.04` du `registry.example.com` registry.example.com:

```
docker pull registry.example.com/username/ubuntu:14.04
```

Liste des images téléchargées localement

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              693bce725149        6 days ago          967 B
postgres            9.5                 0f3af79d8673        10 weeks ago        265.7 MB
postgres            latest              0f3af79d8673        10 weeks ago        265.7 MB
```

Référencement d'images

Les commandes Docker qui prennent le nom d'une image acceptent quatre formes différentes:

Type	Exemple
ID court	`693bce725149`
prénom	`hello-world` *(par défaut à `:latest` balise)*
Nom + tag	`hello-world:latest`
Digérer	`hello-world@sha256:e52be8ffeeb1f374f440893189cd32f44cb166650e7ab185fa7735b7dc48d619`

Remarque: vous ne pouvez vous référer à une image que par son condensé si cette image a été à l'origine extraite à l'aide de ce résumé. Pour voir le résumé d'une image (s'il en existe une), exécutez les `docker images --digests` .

Supprimer des images

La commande `docker rmi` permet de supprimer des images:

```
docker rmi <image name>
```

Le nom complet de l'image doit être utilisé pour supprimer une image. Si l'image n'a pas été marquée pour supprimer le nom du registre, elle doit être spécifiée. Par exemple:

```
docker rmi registry.example.com/username/myAppImage:1.3.5
```

Il est également possible de supprimer les images par leur ID à la place:

```
docker rmi 693bce725149
```

Pour plus de commodité, il est possible de supprimer des images par leur identifiant d'image en spécifiant uniquement les premiers caractères de l'ID d'image, tant que la sous-chaîne spécifiée n'est pas ambiguë:

```
docker rmi 693
```

> **Remarque: les** images peuvent être supprimées même si des conteneurs existants utilisent cette image. docker rmi se contente de "débloquer" l'image.

Si aucun conteneur n'utilise une image, celle-ci est récupérée. Si un conteneur utilise une image, l'image sera collectée une fois que tous les conteneurs qui l'utilisent sont supprimés. Par exemple:

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
PORTS               NAMES
5483657ee07b        hello-world         "/hello"            Less than a second ago   Exited
(0) 2 seconds ago                       small_elion

$ docker rmi hello-world
```

```
Untagged: hello-world:latest

$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED                   STATUS
PORTS               NAMES
5483657ee07b        693bce725149        "/hello"            Less than a second ago    Exited
(0) 12 seconds ago                       small_elion
```

Supprimer toutes les images sans conteneur démarré

Pour supprimer toutes les images locales sans conteneur démarré, vous pouvez fournir une liste des images en tant que paramètre:

```
docker rmi $(docker images -qa)
```

Supprimer toutes les images

Si vous souhaitez supprimer des images, qu'elles aient ou non un conteneur démarré, utilisez l'indicateur de force (`-f`):

```
docker rmi -f $(docker images -qa)
```

Supprimer les images pendantes

Si une image n'est pas étiquetée et n'est pas utilisée par un conteneur, elle est "en suspens" et peut être supprimée comme ceci:

```
docker images -q --no-trunc -f dangling=true | xargs -r docker rmi
```

Rechercher des images dans le Docker Hub

Vous pouvez rechercher des images dans Docker Hub en utilisant la commande de recherche :

```
docker search <term>
```

Par exemple:

```
$ docker search nginx
NAME                     DESCRIPTION                                     STARS     OFFICIAL
AUTOMATED
nginx                    Official build of Nginx.                        3565      [OK]
jwilder/nginx-proxy      Automated Nginx reverse proxy for docker c...   717
[OK]
richarvey/nginx-php-fpm  Container running Nginx + PHP-FPM capable ...   232
[OK]
...
```

Inspection d'images

```
docker inspect <image>
```

La sortie est au format JSON. Vous pouvez utiliser l'utilitaire de ligne de commande `jq` pour analyser et imprimer uniquement les clés souhaitées.

```
docker inspect <image> | jq -r '.[0].Author'
```

La commande ci-dessus affiche le nom de l'auteur des images.

Marquage des images

Le marquage d'une image est utile pour suivre les différentes versions d'image:

```
docker tag ubuntu:latest registry.example.com/username/ubuntu:latest
```

Un autre exemple de marquage:

```
docker tag myApp:1.4.2 myApp:latest
docker tag myApp:1.4.2 registry.example.com/company/myApp:1.4.2
```

Enregistrement et chargement des images Docker

```
docker save -o ubuntu.latest.tar ubuntu:latest
```

Cette commande enregistre l'image `ubuntu:latest` sous forme d'archive tarball dans le répertoire actuel sous le nom de `ubuntu.latest.tar` . Cette archive tarball peut ensuite être déplacée vers un autre hôte, par exemple en utilisant `rsync` , ou archivée dans le stockage.

Une fois l'archive a été déplacée, la commande suivante crée une image à partir du fichier:

```
docker load -i /tmp/ubuntu.latest.tar
```

Maintenant, il est possible de créer des conteneurs à partir de l’ `ubuntu:latest` image comme d’habitude.

Lire Gérer des images en ligne: https://riptutorial.com/fr/docker/topic/690/gerer-des-images

Chapitre 21: Gestion des conteneurs

Syntaxe

- docker rm [OPTIONS] CONTENEUR [CONTENEUR ...]
- docker attach [OPTIONS] CONTENEUR
- docker exec [OPTIONS] COMMANDER COMMAND [ARG ...]
- docker ps [OPTIONS]
- docker logs [OPTIONS] CONTAINER
- docker inspect [OPTIONS] CONTENEUR | IMAGE [CONTENEUR | IMAGE ...]

Remarques

- Dans les exemples ci-dessus, chaque fois que le conteneur est un paramètre de la commande docker, il est mentionné sous la forme `<container>` ou `container id` ou `<CONTAINER_NAME>` . Dans tous ces endroits, vous pouvez soit passer un nom de conteneur ou un identifiant de conteneur pour spécifier un conteneur.

Examples

Liste des conteneurs

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                     NAMES
2bc9b1988080        redis               "docker-entrypoint.sh"   2 weeks ago         Up 2
hours          0.0.0.0:6379->6379/tcp    elephant-redis
817879be2230        postgres            "/docker-entrypoint.s"   2 weeks ago         Up 2
hours          0.0.0.0:65432->5432/tcp   pt-postgres
```

`docker ps` imprime seul les conteneurs en cours d'exécution. Pour afficher tous les conteneurs (y compris ceux arrêtés), utilisez l'indicateur `-a` :

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                     NAMES
9cc69f11a0f7        docker/whalesay     "ls /"                   26 hours ago        Exited
(0) 26 hours ago                            berserk_wozniak
2bc9b1988080        redis               "docker-entrypoint.sh"   2 weeks ago         Up 2
hours                0.0.0.0:6379->6379/tcp    elephant-redis
817879be2230        postgres            "/docker-entrypoint.s"   2 weeks ago         Up 2
hours                0.0.0.0:65432->5432/tcp   pt-postgres
```

Pour répertorier les conteneurs avec un statut spécifique, utilisez l'option de ligne de commande `-f` pour filtrer les résultats. Voici un exemple de liste de tous les conteneurs sortis:

```
$ docker ps -a -f status=exited
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                     NAMES
9cc69f11a0f7        docker/whalesay     "ls /"                   26 hours ago        Exited
(0) 26 hours ago
```

Il est également possible de répertorier uniquement les ID de conteneur avec le commutateur `-q` . Cela facilite le fonctionnement du résultat avec d'autres utilitaires Unix (tels que `grep` et `awk`):

```
$ docker ps -aq
9cc69f11a0f7
2bc9b1988080
817879be2230
```

Lorsque vous `docker run --name mycontainer1` un conteneur avec `docker run --name mycontainer1` vous donnez un nom spécifique et non un nom aléatoire (sous la forme mood_famous, tel que nostalgic_stallman), et il peut être facile de les trouver avec une telle commande

```
docker ps -f name=mycontainer1
```

Référencement de conteneurs

Les commandes Docker qui prennent le nom d'un conteneur acceptent trois formes différentes:

Type	Exemple
UUID complet	`9cc69f11a0f76073e87f25cb6eaf0e079fbfbd1bc47c063bcd25ed3722a8cc4a`
UUID court	`9cc69f11a0f7`
prénom	`berserk_wozniak`

Utilisez `docker ps` pour afficher ces valeurs pour les conteneurs sur votre système.

L'UUID est généré par Docker et ne peut pas être modifié. Vous pouvez donner un nom au conteneur lorsque vous le lancez. `docker run --name <given name> <image>` . Docker générera un nom aléatoire pour le conteneur si vous n'en spécifiez pas un au moment du démarrage du conteneur.

***REMARQUE** : la valeur de l'UUID (ou un UUID court) peut avoir n'importe quelle longueur tant que la valeur donnée est unique pour un conteneur.*

Démarrage et arrêt des conteneurs

Pour arrêter un conteneur en cours d'exécution:

```
docker stop <container> [<container>...]
```

Cela enverra au processus principal du conteneur un SIGTERM, suivi d'un SIGKILL s'il ne s'arrête pas pendant la période de grâce. Le nom de chaque conteneur est imprimé à mesure qu'il

s'arrête.

Pour démarrer un conteneur qui est arrêté:

```
docker start <container> [<container>...]
```

Cela va démarrer chaque conteneur passé en arrière-plan; le nom de chaque conteneur est imprimé au démarrage. Pour démarrer le conteneur au premier plan, passez le drapeau `-a` (`--attach`).

Liste des conteneurs au format personnalisé

```
docker ps --format 'table {{.ID}}\t{{.Names}}\t{{.Status}}'
```

Recherche d'un conteneur spécifique

```
docker ps --filter name=myapp_1
```

Rechercher un conteneur IP

Pour connaître l'adresse IP de votre conteneur, utilisez:

```
docker inspect <container id> | grep IPAddress
```

ou utiliser docker inspecter

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' ${CID}
```

Redémarrage du conteneur Docker

```
docker restart <container> [<container>...]
```

Option - **time** : secondes pour attendre l'arrêt avant de tuer le conteneur (par défaut 10)

```
docker restart <container> --time 10
```

Supprimer, supprimer et nettoyer des conteneurs

`docker rm` peut être utilisé pour supprimer un conteneur spécifique comme celui-ci:

```
docker rm <container name or id>
```

Pour supprimer tous les conteneurs, vous pouvez utiliser cette expression:

```
docker rm $(docker ps -qa)
```

Par défaut, docker ne supprimera pas un conteneur en cours d'exécution. Tout conteneur en cours d'exécution produira un message d'avertissement et ne sera pas supprimé. Tous les autres conteneurs seront supprimés.

Sinon, vous pouvez utiliser `xargs` :

```
docker ps -aq -f status=exited | xargs -r docker rm
```

Où `docker ps -aq -f status=exited` retournera une liste des ID de conteneur des conteneurs ayant le statut "Exit".

> Avertissement: tous les exemples ci-dessus ne supprimeront que les conteneurs "arrêtés".

Pour supprimer un conteneur, qu'il soit ou non arrêté, vous pouvez utiliser l'indicateur de force `-f` :

```
docker rm -f <container name or id>
```

Pour supprimer tous les conteneurs, quel que soit leur état:

```
docker rm -f $(docker ps -qa)
```

Si vous souhaitez supprimer uniquement les conteneurs avec un statut `dead` :

```
docker rm $(docker ps --all -q -f status=dead)
```

Si vous souhaitez supprimer uniquement les conteneurs avec un statut `exited` :

```
docker rm $(docker ps --all -q -f status=exited)
```

Ce sont toutes les permutations de filtres utilisées lors de la liste des conteneurs .

Pour supprimer à la fois les conteneurs indésirables et les images en attente qui utilisent de l'espace après la version 1.3 , utilisez ce qui suit (similaire à l'outil Unix `df`):

```
$ docker system df
```

Pour supprimer toutes les données inutilisées:

```
$ docker system prune
```

Exécuter la commande sur un conteneur de docker existant

```
docker exec -it <container id> /bin/bash
```

Il est courant de se connecter à un conteneur déjà en cours pour effectuer des tests rapides ou voir ce que fait l'application. Souvent, cela dénote des mauvaises pratiques d'utilisation des conteneurs dues aux journaux et les fichiers modifiés doivent être placés dans des volumes. Cet

exemple nous permet de connecter le conteneur. Cela suppose que / bin / bash est disponible dans le conteneur, il peut être / bin / sh ou autre chose.

```
docker exec <container id> tar -czvf /tmp/backup.tgz /data
docker cp <container id>:/tmp/backup.tgz .
```

Cet exemple archive le contenu du répertoire de données dans un tar. Ensuite, avec `docker cp` vous pouvez le récupérer.

Journaux de conteneurs

```
Usage:  docker logs [OPTIONS] CONTAINER

Fetch the logs of a container

  -f, --follow=false        Follow log output
  --help=false              Print usage
  --since=                  Show logs since timestamp
  -t, --timestamps=false    Show timestamps
  --tail=all                Number of lines to show from the end of the logs
```

Par exemple:

```
$ docker ps
CONTAINER ID    IMAGE    COMMAND                  CREATED     STATUS       PORTS
ff9716dda6cb    nginx    "nginx -g 'daemon off"  8 days ago  Up 22 hours  443/tcp,
0.0.0.0:8080->80/tcp

$ docker logs ff9716dda6cb
xx.xx.xx.xx - - [15/Jul/2016:14:03:44 +0000] "GET /index.html HTTP/1.1" 200 511
"https://google.com" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/50.0.2661.75 Safari/537.36"
xx.xx.xx.xx - - [15/Jul/2016:14:03:44 +0000] "GET /index.html HTTP/1.1" 200 511
"https://google.com" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/50.0.2661.75 Safari/537.36"
```

Se connecter à une instance exécutée en tant que démon

Il y a deux manières d'y parvenir, la première et la plus connue est la suivante:

```
docker attach --sig-proxy=false <container>
```

Celui-ci attache littéralement votre bash au conteneur bash, ce qui signifie que si vous avez un script en cours d'exécution, vous verrez le résultat.

Pour détacher, tapez simplement: `Ctl-P Ctl-Q`

Mais si vous avez besoin d'une méthode plus conviviale et pour pouvoir créer de nouvelles instances de bash, lancez simplement la commande suivante:

```
docker exec -it <container> bash
```

Copier un fichier depuis / vers des conteneurs

de conteneur à hôte

```
docker cp CONTAINER_NAME:PATH_IN_CONTAINER PATH_IN_HOST
```

de l'hôte au conteneur

```
docker cp PATH_IN_HOST CONTAINER_NAME:PATH_IN_CONTAINER
```

Si j'utilise jess / transmission de

https://hub.docker.com/r/jess/transmission/builds/bsn7eqxrkzrhxazcuytbmzp/

, les fichiers dans le conteneur sont dans / transmission / téléchargement

et mon répertoire actuel sur l'hôte est / home / $ USER / abc, après

`docker cp transmission_id_or_name:/transmission/download .`

Je vais avoir les fichiers copiés dans

`/home/$USER/abc/transmission/download`

vous ne pouvez pas, en utilisant `docker cp` copier un seul fichier, vous copiez l'arborescence et les fichiers

Supprimer, supprimer et nettoyer les volumes du menu fixe

Les volumes Docker ne sont pas automatiquement supprimés lorsqu'un conteneur est arrêté. Pour supprimer des volumes associés lorsque vous arrêtez un conteneur:

```
docker rm -v <container id or name>
```

Si l' `-v` n'est pas spécifiée, le volume reste sur le disque en tant que «volume en suspens». Pour supprimer tous les volumes en suspens:

```
docker volume rm $(docker volume ls -qf dangling=true)
```

Le filtre `docker volume ls -qf dangling=true` renvoie une liste de noms de volumes de `docker volume ls -qf dangling=true` , y compris ceux qui ne sont pas `docker volume ls -qf dangling=true` , qui ne sont pas attachés à un conteneur.

Vous pouvez également utiliser `xargs` :

```
docker volume ls -f dangling=true -q | xargs --no-run-if-empty docker volume rm
```

Exporter et importer des systèmes de fichiers de conteneur Docker

Il est possible de sauvegarder le contenu du système de fichiers d'un conteneur Docker dans un fichier d'archive tarball. Ceci est utile pour déplacer des systèmes de fichiers de conteneur vers différents hôtes, par exemple si un conteneur de base de données a des modifications importantes et qu'il n'est pas possible de répliquer ces modifications ailleurs. **Veuillez noter** qu'il est préférable de créer un conteneur entièrement nouveau à partir d'une image mise à jour à l'aide d'une commande `docker run` ou d' `docker-compose.yml` fichier `docker-compose.yml` , au lieu d'exporter et de déplacer le système de fichiers d'un conteneur. Le pouvoir de Docker repose en partie sur l'auditabilité et la responsabilisation de son style déclaratif de création d'images et de conteneurs. En utilisant l' `docker export` et l' `docker import` , cette puissance est modérée en raison de l'obscurcissement des modifications apportées à l'intérieur du système de fichiers d'un conteneur par rapport à son état d'origine.

```
docker export -o redis.tar redis
```

La commande ci-dessus créera une image vide, puis exportera le système de fichiers du conteneur `redis` dans cette image vide. Pour importer depuis une archive tarball, utilisez:

```
docker import ./redis.tar redis-imported:3.0.7
```

Cette commande crée l'image `redis-imported:3.0.7` partir de laquelle des conteneurs peuvent être créés. Il est également possible de créer des modifications lors de l'importation, ainsi que de définir un message de validation:

```
docker import -c="ENV DEBUG true" -m="enable debug mode" ./redis.tar redis-changed
```

Les directives Dockerfile disponibles pour être utilisées avec l'option de ligne de commande `-c` sont `CMD` , `ENTRYPOINT` , `ENV` , `EXPOSE` , `ONBUILD` , `USER` , `VOLUME` , `WORKDIR` .

Lire Gestion des conteneurs en ligne: https://riptutorial.com/fr/docker/topic/689/gestion-des-conteneurs

Chapitre 22: Images de construction

Paramètres

Paramètre	Détails
--tirer	S'assure que l'image de base (`FROM`) est à jour avant de construire le reste du fichier Dockerfile.

Examples

Construire une image à partir d'un fichier Dockerfile

Une fois que vous avez un fichier Docker, vous pouvez créer une image à l'aide de `docker build` . La forme de base de cette commande est la suivante:

```
docker build -t image-name path
```

Si votre fichier Dockerfile n'est pas nommé `Dockerfile` , vous pouvez utiliser l'indicateur `-f` pour donner le nom du fichier Dockerfile à générer.

```
docker build -t image-name -f Dockerfile2 .
```

Par exemple, pour créer une image nommée `dockerbuild-example:1.0.0` partir d'un `Dockerfile` dans le répertoire de travail en cours:

```
$ ls
Dockerfile Dockerfile2

$ docker build -t dockerbuild-example:1.0.0 .

$ docker build -t dockerbuild-example-2:1.0.0 -f Dockerfile2 .
```

Reportez-vous à la documentation relative à l' utilisation de la `docker build` pour plus d'options et de paramètres.

Une erreur courante est de créer un fichier Dockerfile dans le répertoire personnel de l'utilisateur (`~`). C'est une mauvaise idée car lors de la `docker build -t mytag .` ce message apparaîtra longtemps:

> Contexte de téléchargement

La cause est le démon docker qui tente de copier tous les fichiers de l'utilisateur (à la fois le répertoire de base et ses sous-répertoires). Évitez cela en spécifiant toujours un répertoire pour le fichier Docker.

L'ajout d'un fichier `.dockerignore` répertoire de construction est une bonne pratique . Sa syntaxe est similaire à `.gitignore` fichiers `.gitignore` et s'assurera que seuls les fichiers et répertoires `.gitignore` sont téléchargés en tant que contexte de la génération.

Un simple Dockerfile

```
FROM node:5
```

La directive `FROM` spécifie une image à partir de laquelle Toute référence d'image valide peut être utilisée.

```
WORKDIR /usr/src/app
```

La directive `WORKDIR` définit le répertoire de travail actuel dans le conteneur, ce qui équivaut à exécuter `cd` dans le conteneur. (Remarque: `RUN cd` *ne* changera *pas* le répertoire de travail en cours.)

```
RUN npm install cowsay knock-knock-jokes
```

`RUN` exécute la commande donnée à l'intérieur du conteneur.

```
COPY cowsay-knockknock.js ./
```

`COPY` copie le fichier ou le répertoire spécifié dans le premier argument à partir du contexte de génération (le *`path`* transmis au `docker build` *`path`*) vers l'emplacement du conteneur spécifié par le deuxième argument.

```
CMD node cowsay-knockknock.js
```

`CMD` spécifie une commande à exécuter lorsque l'image est exécutée et qu'aucune commande n'est donnée. Il peut être remplacé en transmettant une commande à `docker run` .

Il existe de nombreuses autres instructions et options. voir la référence Dockerfile pour une liste complète.

Différence entre ENTRYPOINT et CMD

Il existe deux directives `Dockerfile` pour spécifier quelle commande exécuter par défaut dans les images construites. Si vous spécifiez uniquement `CMD` alors docker exécutera cette commande en utilisant le `ENTRYPOINT` par défaut, à savoir `/bin/sh -c` . Vous pouvez remplacer soit le point d'entrée, soit la commande lorsque vous démarrez l'image construite. Si vous spécifiez les deux, alors `ENTRYPOINT` spécifie l'exécutable de votre processus de conteneur et `CMD` sera fourni comme paramètre de cet exécutable.

Par exemple, si votre `Dockerfile` contient

```
FROM ubuntu:16.04
```

```
CMD ["/bin/date"]
```

Vous utilisez ensuite la directive `ENTRYPOINT` par défaut de `/bin/sh -c` et exécutez `/bin/date` avec `/bin/date` d' `/bin/date` par défaut. La commande de votre processus de conteneur sera `/bin/sh -c /bin/date` . Une fois que vous exécutez cette image, elle affichera par défaut la date actuelle

```
$ docker build -t test .
$ docker run test
Tue Jul 19 10:37:43 UTC 2016
```

Vous pouvez remplacer `CMD` sur la ligne de commande, auquel cas il exécutera la commande que vous avez spécifiée.

```
$ docker run test /bin/hostname
bf0274ec8820
```

Si vous spécifiez une directive `ENTRYPOINT` , Docker utilisera cet exécutable et la directive `CMD` spécifie le ou les paramètres par défaut de la commande. Donc, si votre `Dockerfile` contient:

```
FROM ubuntu:16.04
ENTRYPOINT ["/bin/echo"]
CMD ["Hello"]
```

Alors en courant ça va produire

```
$ docker build -t test .
$ docker run test
Hello
```

Vous pouvez fournir différents paramètres si vous le souhaitez, mais ils seront tous exécutés `/bin/echo`

```
$ docker run test Hi
Hi
```

Si vous souhaitez remplacer le point d'entrée répertorié dans votre fichier Docker (par exemple, si vous souhaitez exécuter une commande différente de `echo` dans ce conteneur), vous devez spécifier le paramètre `--entrypoint` sur la ligne de commande:

```
$ docker run --entrypoint=/bin/hostname test
b2c70e74df18
```

Généralement, vous utilisez la directive `ENTRYPOINT` pour pointer vers votre application principale que vous souhaitez exécuter et `CMD` vers les paramètres par défaut.

Exposer un port dans le fichier Dockerfile

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

De la documentation de Docker:

> L'instruction `EXPOSE` informe Docker que le conteneur écoute les ports réseau spécifiés au moment de l'exécution. `EXPOSE` ne rend pas les ports du conteneur accessibles à l'hôte. Pour ce faire, vous devez utiliser l' `-p` pour publier une plage de ports ou l' `-P` pour publier tous les ports exposés. Vous pouvez exposer un numéro de port et le publier en externe sous un autre numéro.

Exemple:

À l'intérieur de votre fichier Docker:

```
EXPOSE 8765
```

Pour accéder à ce port depuis la machine hôte, incluez cet argument dans votre commande d' `docker run` :

```
-p 8765:8765
```

ENTRYPOINT et CMD vus sous forme de verbe et de paramètre

Supposons que vous avez un fichier Dockerfile se terminant par

```
ENTRYPOINT [ "nethogs"] CMD ["wlan0"]
```

si vous construisez cette image avec un

```
docker built -t inspector .
```

lancer l'image construite avec un tel Dockerfile avec une commande telle que

```
docker run -it --net=host --rm inspector
```

, nethogs surveillera l'interface nommée wlan0

Maintenant, si vous voulez surveiller l'interface eth0 (ou wlan1, ou ra1 ...), vous ferez quelque chose comme

```
docker run -it --net=host --rm inspector eth0
```

ou

```
docker run -it --net=host --rm inspector wlan1
```

Pousser et tirer une image vers Docker Hub ou un autre registre

Les images créées localement peuvent être transférées vers Docker Hub ou tout autre hôte repo docker, appelé registre. Utilisez la `docker login` au `docker login` pour vous connecter à un compte docker hub existant.

```
docker login

Login with your Docker ID to push and pull images from Docker Hub.
If you don't have a Docker ID, head over to https://hub.docker.com to create one.

Username: cjsimon
Password:
Login Succeeded
```

Un registre de docker différent peut être utilisé en spécifiant un nom de serveur. Cela fonctionne également pour les registres privés ou auto-hébergés. De plus, l'utilisation d'un magasin de données d'identification externe pour la sécurité est possible.

```
docker login quay.io
```

Vous pouvez ensuite baliser et transférer des images vers le registre auquel vous êtes connecté. Votre référentiel doit être spécifié en tant que `server/username/reponame:tag` . Omettre le serveur par défaut est actuellement Docker Hub. (Le registre par défaut ne peut pas être remplacé par un autre fournisseur et il n'est pas prévu de l'implémenter.)

```
docker tag mynginx quay.io/cjsimon/mynginx:latest
```

Différentes balises peuvent être utilisées pour représenter différentes versions, ou branches, de la même image. Une image avec plusieurs balises différentes affichera chaque balise dans le même repo.

Utilisez les `docker images` pour afficher la liste des images installées sur votre ordinateur local, y compris votre nouvelle image. Puis, appuyez sur pour le télécharger dans le registre et tirez pour télécharger l'image.

```
docker push quay.io/cjsimon/mynginx:latest
```

Toutes les balises d'une image peuvent être extraites en spécifiant l'option `-a`

```
docker pull quay.io/cjsimon/mynginx:latest
```

Construire en utilisant un proxy

Souvent , lors de la construction d' une image Docker, le Dockerfile contient des instructions qui exécute des programmes pour aller chercher des ressources de l'Internet (`wget` par exemple pour tirer une version binaire de programme sur GitHub exemple).

Il est possible d'indiquer à Docker de transmettre les variables d'environnement set set afin que ces programmes effectuent ces récupérations via un proxy:

```
$ docker build --build-arg http_proxy=http://myproxy.example.com:3128 \
               --build-arg https_proxy=http://myproxy.example.com:3128 \
               --build-arg no_proxy=internal.example.com \
               -t test .
```

`build-arg` sont des variables d'environnement disponibles uniquement au moment de la construction.

Lire Images de construction en ligne: https://riptutorial.com/fr/docker/topic/713/images-de-construction

Chapitre 23: Inspection d'un conteneur en cours d'exécution

Syntaxe

- docker inspect [OPTIONS] CONTENEUR | IMAGE [CONTENEUR | IMAGE ...]

Examples

Obtenir des informations sur le conteneur

Pour obtenir toutes les informations relatives à un conteneur, vous pouvez exécuter:

```
docker inspect <container>
```

Obtenir des informations spécifiques à partir d'un conteneur

Vous pouvez obtenir des informations spécifiques à partir d'un conteneur en exécutant:

```
docker inspect -f '<format>' <container>
```

Par exemple, vous pouvez obtenir les paramètres réseau en exécutant:

```
docker inspect -f '{{ .NetworkSettings }}' <container>
```

Vous pouvez également obtenir uniquement l'adresse IP:

```
docker inspect -f '{{ .NetworkSettings.IPAddress }}' <container>
```

Le paramètre -f signifie le format et recevra un gabarit comme entrée pour formater ce qui est attendu, mais cela n'apportera pas un beau retour, alors essayez:

```
docker inspect -f '{{ json .NetworkSettings }}' {{containerIdOrName}}
```

le mot-clé json apportera le retour en tant que JSON.

Donc pour finir, un petit truc est d'utiliser Python pour formater la sortie JSON:

```
docker inspect -f '{{ json .NetworkSettings }}' <container> | python -mjson.tool
```

Et voilà, vous pouvez interroger tout ce qui se trouve sur le docker et le rendre joli dans votre terminal.

Il est également possible d'utiliser un utilitaire appelé " jq " pour aider à traiter `docker inspect` sortie de la commande.

```
docker inspect -f '{{ json .NetworkSettings }}' aa1 | jq [.Gateway]
```

La commande ci-dessus renvoie la sortie suivante:

```
[
  "172.17.0.1"
]
```

Cette sortie est en fait une liste contenant un élément. Parfois, `docker inspect` affiche une liste de plusieurs éléments et vous souhaiterez peut-être faire référence à un élément spécifique. Par exemple, si `Config.Env` contient plusieurs éléments, vous pouvez vous référer au premier élément de cette liste à l'aide de l' `index` :

```
docker inspect --format '{{ index (index .Config.Env) 0 }}' <container>
```

Le premier élément est indexé à zéro, ce qui signifie que le deuxième élément de cette liste est à l'index `1` :

```
docker inspect --format '{{ index (index .Config.Env) 1 }}' <container>
```

En utilisant `len` il est possible d'obtenir le nombre d'éléments de la liste:

```
docker inspect --format `{{ len .Config.Env }}' <container>
```

Et en utilisant des nombres négatifs, il est possible de se référer au dernier élément de la liste:

```
docker inspect -format "{{ index .Config.Cmd $[$(docker inspect -format `{{ len .Config.Cmd
}}' <container>)-1]}}" <container>
```

Certains `docker inspect` l' information se présente comme un dictionnaire de clé: valeur, voici un extrait d'un `docker inspect` d'un jess / spotify conteneur en cours d' exécution

```
"Config": { "Hostname": "8255f4804dde", "Domainname": "", "User": "spotify", "AttachStdin":
false, "AttachStdout": false, "AttachStderr": false, "Tty": false, "OpenStdin": false,
"StdinOnce": false, "Env": [ "DISPLAY=unix:0",
"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin", "HOME=/home/spotify" ],
"Cmd": [ "-stylesheet=/home/spotify/spotify-override.css" ], "Image": "jess/spotify", "Volumes":
null, "WorkingDir": "/home/spotify", "Entrypoint": [ "spotify" ], "OnBuild": null, "Labels": {}
},
```

donc je reçois les valeurs de toute la section Config

```
docker inspect -f '{{.Config}}' 825

{8255f4804dde spotify false false false map[] false false false [DISPLAY=unix:0
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin HOME=/home/spotify] [-
stylesheet=/home/spotify/spotify-override.css] false jess/spotify map[] /home/spotify [spotify]
false [] map[] }
```

mais aussi un seul champ, comme la valeur de Config.Image

```
docker inspect -f '{{index (.Config) "Image" }}' 825

jess/spotify
```

ou Config.Cmd

```
docker inspect -f '{{.Config.Cmd}}' 825

[-stylesheet=/home/spotify/spotify-override.css]
```

Inspecter une image

Pour inspecter une image, vous pouvez utiliser l'ID de l'image ou le nom de l'image, composé du référentiel et de la balise. Dites, vous avez l'image de base CentOS 6:

```
➜ ~ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
centos              centos6             cf2c3ece5e41        2 weeks ago         194.6 MB
```

Dans ce cas, vous pouvez exécuter l'une des opérations suivantes:

- ➜ ~ docker inspect cf2c3ece5e41
- ➜ ~ docker inspect centos:centos6

Ces deux commandes vous donneront toutes les informations disponibles dans un tableau JSON:

```
[
    {
        "Id": "sha256:cf2c3ece5e418fd063bfad5e7e8d083182195152f90aac3a5ca4dbfbf6a1fc2a",
        "RepoTags": [
            "centos:centos6"
        ],
        "RepoDigests": [],
        "Parent": "",
        "Comment": "",
        "Created": "2016-07-01T22:34:39.970264448Z",
        "Container": "b355fe9a01a8f95072e4406763138c5ad9ca0a50dbb0ce07387ba905817d6702",
        "ContainerConfig": {
            "Hostname": "68a1f3cfce80",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) CMD [\"/bin/bash\"]"
            ],
```

```
            "Image":
"sha256:cdbcc7980b002dc19b4d5b6ac450993c478927f673339b4e6893647fe2158fa7",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "build-date": "20160701",
                "license": "GPLv2",
                "name": "CentOS Base Image",
                "vendor": "CentOS"
            }
        },
        "DockerVersion": "1.10.3",
        "Author": "https://github.com/CentOS/sig-cloud-instance-images",
        "Config": {
            "Hostname": "68a1f3cfce80",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/bash"
            ],
            "Image":
"sha256:cdbcc7980b002dc19b4d5b6ac450993c478927f673339b4e6893647fe2158fa7",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "build-date": "20160701",
                "license": "GPLv2",
                "name": "CentOS Base Image",
                "vendor": "CentOS"
            }
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 194606575,
        "VirtualSize": 194606575,
        "GraphDriver": {
            "Name": "aufs",
            "Data": null
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:2714f4a6cdee9d4c987fef019608a4f61f1cda7ccf423aeb8d7d89f745c58b18"
            ]
        }
    }
]
```

Impression des informations spécifiques

docker `inspect` les `--format` Go via l'option `--format` . Cela permet une meilleure intégration dans les scripts, sans recourir aux outils traditionnels pipes / sed / grep.

Imprimer un IP interne du conteneur :

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' 7786807d8084
```

Ceci est utile pour l'accès direct au réseau de la configuration automatique des équilibreurs de charge.

Imprimer un conteneur PID *init* :

```
docker inspect --format '{{ .State.Pid }}' 7786807d8084
```

Ceci est utile pour une inspection plus approfondie via `/proc` ou des outils tels que `strace` .

Formations avancées :

```
docker inspect --format 'Container {{ .Name }} listens on {{ .NetworkSettings.IPAddress }}:{{
range $index, $elem := .Config.ExposedPorts }}{{ $index }}{{ end }}' 5765847de886 7786807d8084
```

Va sortir:

```
Container /redis listens on 172.17.0.3:6379/tcp
Container /api listens on 172.17.0.2:4000/tcp
```

Déboguer les journaux de conteneur à l'aide de docker inspect

`docker inspect` commande `docker inspect` peut être utilisée pour déboguer les journaux de conteneur.

La stdout et le stderr du conteneur peuvent être vérifiés pour déboguer le conteneur, dont l'emplacement peut être obtenu à l'aide de `docker inspect` .

Commande: `docker inspect <container-id> | grep Source`

Il donne l'emplacement des conteneurs stdout et stderr.

Examen de stdout / stderr d'un conteneur en cours d'exécution

```
docker logs --follow <containerid>
```

Cela limite la sortie du conteneur en cours d'exécution. Ceci est utile si vous n'avez pas configuré de pilote de journalisation sur le démon docker.

Lire Inspection d'un conteneur en cours d'exécution en ligne:

https://riptutorial.com/fr/docker/topic/1336/inspection-d-un-conteneur-en-cours-d-execution

Chapitre 24: Iptables avec Docker

Introduction

Cette rubrique explique comment limiter l'accès à vos conteneurs de docker du monde extérieur à l'aide d'iptables.

Pour les personnes impatientes, vous pouvez consulter les exemples. Pour les autres, veuillez lire la section des remarques pour comprendre comment construire de nouvelles règles.

Syntaxe

- iptables -I DOCKER [RULE ...] [ACCEPT | DROP] // Pour ajouter une règle en haut de la table DOCKER
- iptables -D DOCKER [RULE ...] [ACCEPT | DROP] // Pour supprimer une règle de la table DOCKER
- ipset restore </etc/ipfriends.conf // Pour reconfigurer vos ipfriends *ipset*

Paramètres

Paramètres	Détails
ext_if	Votre interface externe sur l'hôte Docker.
XXX.XXX.XXX.XXX	Une adresse IP particulière sur laquelle les conteneurs Docker doivent être accessibles.
AAAA.AAAA.AAAA.AAAA	Une autre adresse IP sur laquelle les conteneurs Docker doivent être accessibles doit être fournie.
ipfriends	Le nom de l'ipset définissant les adresses IP autorisées à accéder à vos conteneurs Docker.

Remarques

Le problème

La configuration des règles iptables pour les conteneurs Docker est un peu délicate. Au début, vous penseriez que les règles de pare-feu "classiques" devraient faire l'affaire.

Par exemple, supposons que vous avez configuré un conteneur nginx-proxy + plusieurs conteneurs de services pour exposer via HTTPS certains services Web personnels. Ensuite, une règle comme celle-ci devrait donner accès à vos services Web uniquement pour IP

XXX.XXX.XXX.XXX.

```
$ iptables -A INPUT -i eth0 -p tcp -s XXX.XXX.XXX.XXX -j ACCEPT
$ iptables -P INPUT DROP
```

Cela ne marchera pas, vos conteneurs sont toujours accessibles pour tout le monde.

En effet, les conteneurs Docker ne sont pas des services hôtes. Ils s'appuient sur un réseau virtuel dans votre hôte et l'hôte agit comme une passerelle pour ce réseau. Et en ce qui concerne les passerelles, le trafic routé n'est pas géré par la table INPUT, mais par la table FORWARD, ce qui rend la règle affichée avant qu'elle soit inefficace.

Mais ce n'est pas tout. En fait, le démon Docker crée beaucoup de règles iptables quand il commence à faire sa magie concernant la connectivité réseau des conteneurs. En particulier, une table DOCKER est créée pour gérer les règles concernant les conteneurs en transférant le trafic de la table FORWARD vers cette nouvelle table.

```
$ iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy DROP)
target     prot opt source               destination
DOCKER-ISOLATION  all  --  anywhere             anywhere
DOCKER     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere             ctstate RELATED,ESTABLISHED
ACCEPT     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere
DOCKER     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere             ctstate RELATED,ESTABLISHED
ACCEPT     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain DOCKER (2 references)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             172.18.0.4           tcp dpt:https
ACCEPT     tcp  --  anywhere             172.18.0.4           tcp dpt:http

Chain DOCKER-ISOLATION (1 references)
target     prot opt source               destination
DROP       all  --  anywhere             anywhere
DROP       all  --  anywhere             anywhere
RETURN     all  --  anywhere             anywhere
```

La solution

Si vous consultez la documentation officielle (https://docs.docker.com/v1.5/articles/networking/) , une première solution est proposée pour limiter l'accès du conteneur Docker à une adresse IP particulière.

```
$ iptables -I DOCKER -i ext_if ! -s 8.8.8.8 -j DROP
```

En effet, ajouter une règle en haut de la table DOCKER est une bonne idée. Cela n'interfère pas avec les règles configurées automatiquement par Docker, et c'est simple. Mais deux grands manques:

- Tout d'abord, que faire si vous avez besoin d'accéder à deux adresses IP au lieu d'une seule? Ici, une seule IP src peut être acceptée, les autres seront supprimées sans aucun moyen de prévention.
- Deuxièmement, que se passe-t-il si votre docker doit avoir accès à Internet? Dans la pratique, aucune requête ne réussira car seul le serveur 8.8.8.8 pourra y répondre.
- Enfin, que faire si vous souhaitez ajouter d'autres logiques? Par exemple, donnez accès à n'importe quel utilisateur à votre serveur Web utilisant le protocole HTTP, mais limitez tout le reste à une adresse IP particulière.

Pour la première observation, nous pouvons utiliser *ipset* . Au lieu d'autoriser une IP dans la règle ci-dessus, nous autorisons toutes les IP de l'ipset prédéfini. En prime, l'ipset peut être mis à jour sans qu'il soit nécessaire de redéfinir la règle iptable.

```
$ iptables -I DOCKER -i ext_if -m set ! --match-set my-ipset src -j DROP
```

Pour la seconde observation, il s'agit d'un problème canonique pour les pare-feu: si vous êtes autorisé à contacter un serveur via un pare-feu, le pare-feu doit autoriser le serveur à répondre à votre demande. Cela peut être fait en autorisant des paquets liés à une connexion établie. Pour la logique du docker, cela donne:

```
$ iptables -I DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
```

La dernière observation porte sur un point: les règles iptables sont essentielles. En effet, une logique supplémentaire pour ACCEPTER certaines connexions (y compris celle concernant les connexions ESTABLISHED) doit être placée en haut de la table DOCKER, avant la règle DROP qui refuse toutes les connexions restantes ne correspondant pas à l'ipset.

Comme nous utilisons l'option -I d'iptable, qui insère des règles en haut de la table, les règles iptables précédentes doivent être insérées par ordre inverse:

```
// Drop rule for non matching IPs
$ iptables -I DOCKER -i ext_if -m set ! --match-set my-ipset src -j DROP
// Then Accept rules for established connections
$ iptables -I DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
$ iptables -I DOCKER -i ext_if ... ACCEPT // Then 3rd custom accept rule
$ iptables -I DOCKER -i ext_if ... ACCEPT // Then 2nd custom accept rule
$ iptables -I DOCKER -i ext_if ... ACCEPT // Then 1st custom accept rule
```

En gardant cela à l'esprit, vous pouvez maintenant vérifier les exemples qui illustrent cette configuration.

Examples

Limiter l'accès aux conteneurs Docker à un ensemble d'IP

D'abord, installez *ipset* si nécessaire. Veuillez vous référer à votre distribution pour savoir comment le faire. À titre d'exemple, voici la commande pour les distributions de type Debian.

```
$ apt-get update
$ apt-get install ipset
```

Créez ensuite un fichier de configuration pour définir un ipset contenant les adresses IP pour lesquelles vous souhaitez ouvrir l'accès à vos conteneurs Docker.

```
$ vi /etc/ipfriends.conf
# Recreate the ipset if needed, and flush all entries
create -exist ipfriends hash:ip family inet hashsize 1024 maxelem 65536
flush
# Give access to specific ips
add ipfriends XXX.XXX.XXX.XXX
add ipfriends YYY.YYY.YYY.YYY
```

Chargez cet ipset.

```
$ ipset restore < /etc/ipfriends.conf
```

Assurez-vous que votre démon Docker est en cours d'exécution: aucune erreur ne doit apparaître après la saisie de la commande suivante.

```
$ docker ps
```

Vous êtes prêt à insérer vos règles iptables. Vous **devez** respecter la commande.

```
// All requests of src ips not matching the ones from ipset ipfriends will be dropped.
$ iptables -I DOCKER -i ext_if -m set ! --match-set ipfriends src -j DROP
// Except for requests coming from a connection already established.
$ iptables -I DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
```

Si vous souhaitez créer de nouvelles règles, vous devrez supprimer toutes les règles personnalisées que vous avez ajoutées avant d'insérer les nouvelles.

```
$ iptables -D DOCKER -i ext_if -m set ! --match-set ipfriends src -j DROP
$ iptables -D DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
```

Configurez l'accès aux restrictions au démarrage du démon Docker

Travaux en cours

Quelques règles iptables personnalisées

Travaux en cours

Lire Iptables avec Docker en ligne: https://riptutorial.com/fr/docker/topic/9201/iptables-avec-docker

Chapitre 25: Les services en cours d'exécution

Examples

Créer un service plus avancé

Dans l'exemple suivant, nous allons créer un service avec le nom *visualizer* . Nous allons spécifier une étiquette personnalisée et remapper le port interne du service de 8080 à 9090. En outre, nous allons associer un répertoire externe de l'hôte au service.

```
docker service create \
        --name=visualizer \
        --label com.my.custom.label=visualizer \
        --publish=9090:8080 \
        --mount type=bind,source=/var/run/docker.sock,target=/var/run/docker.sock \
        manomarks/visualizer:latest
```

Créer un service simple

Ce simple exemple créera un service Web de salut dans le monde qui écoutera sur le port 80.

```
docker service create \
    --publish 80:80 \
    tutum/hello-world
```

Supprimer un service

Cet exemple simple supprimera le service avec le nom "visualizer":

```
docker service rm visualizer
```

Mise à l'échelle d'un service

Cet exemple mettra à l'échelle le service à 4 instances:

```
docker service scale visualizer=4
```

En mode Essaim Docker, nous n'arrêtons pas un service. Nous réduisons à zéro:

```
docker service scale visualizer=0
```

Lire Les services en cours d'exécution en ligne: https://riptutorial.com/fr/docker/topic/8802/les-services-en-cours-d-execution

Chapitre 26: Mode essaim Docker

Introduction

Un essaim est un nombre de moteurs Docker (ou *nœuds*) qui déploient des *services* collectivement. Swarm est utilisé pour distribuer le traitement sur de nombreuses machines physiques, virtuelles ou cloud.

Syntaxe

- Initialiser un essaim : init docker swarm [OPTIONS]
- Joindre un essaim en tant que nœud et / ou gestionnaire : Docker Swarm join [OPTIONS] HOST: PORT
- Créer un nouveau service : service docker créer [OPTIONS] IMAGE [COMMANDE] [ARG ...]
- Afficher des informations détaillées sur un ou plusieurs services : service docker inspecter [OPTIONS] SERVICE [SERVICE ...]
- Liste des services : service docker ls [OPTIONS]
- Supprimer un ou plusieurs services : service docker rm SERVICE [SERVICE ...]
- Mettre à l'échelle un ou plusieurs services répliqués : échelle du service docker SERVICE = REPLICAS [SERVICE = REPLICAS ...]
- Liste les tâches d'un ou plusieurs services : service docker ps [OPTIONS] SERVICE [SERVICE ...]
- Mettre à jour un service : mise à jour du service docker [OPTIONS] SERVICE

Remarques

Le mode Swarm implémente les fonctionnalités suivantes:

- Gestion de cluster intégrée à Docker Engine
- Conception décentralisée
- Modèle de service déclaratif
- Mise à l'échelle
- Réconciliation de l'Etat souhaitée
- Réseau multi-hôte
- Découverte de service
- L'équilibrage de charge
- Conception sécurisée par défaut
- Mises à jour roulantes

Pour plus de documentation officielle sur Docker concernant la visite de Swarm : Présentation du mode Swarm

Commandes CLI du mode Swarm

Cliquez sur la description des commandes pour la documentation

Initialiser un essaim

```
docker swarm init [OPTIONS]
```

Rejoindre un essaim en tant que nœud et / ou gestionnaire

```
docker swarm join [OPTIONS] HOST:PORT
```

Créer un nouveau service

```
docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Afficher des informations détaillées sur un ou plusieurs services

```
docker service inspect [OPTIONS] SERVICE [SERVICE...]
```

Liste des services

```
docker service ls [OPTIONS]
```

Supprimer un ou plusieurs services

```
docker service rm SERVICE [SERVICE...]
```

Mettre à l'échelle un ou plusieurs services répliqués

```
docker service scale SERVICE=REPLICAS [SERVICE=REPLICAS...]
```

Liste les tâches d'un ou plusieurs services

```
docker service ps [OPTIONS] SERVICE [SERVICE...]
```

Mettre à jour un service

```
docker service update [OPTIONS] SERVICE
```

Examples

Créer un essaim sous Linux en utilisant docker-machine et VirtualBox

```
# Create the nodes
# In a real world scenario we would use at least 3 managers to cover the fail of one manager.
docker-machine create -d virtualbox manager
docker-machine create -d virtualbox worker1

# Create the swarm
# It is possible to define a port for the *advertise-addr* and *listen-addr*, if none is
defined the default port 2377 will be used.
docker-machine ssh manager \
    docker swarm init \
    --advertise-addr $(docker-machine ip manager)
    --listen-addr $(docker-machine ip manager)

# Extract the Tokens for joining the Swarm
# There are 2 different Tokens for joining the swarm.
MANAGER_TOKEN=$(docker-machine ssh manager docker swarm join-token manager --quiet)
WORKER_TOKEN=$(docker-machine ssh manager docker swarm join-token worker --quiet)


# Join a worker node with the worker token
docker-machine ssh worker1 \
    docker swarm join \
    --token $WORKER_TOKEN \
    --listen-addr $(docker-machine ip worker1) \
    $(docker-machine ip manager):2377
```

Découvrez que le travailleur et le responsable se joignent au jeton

Lors de l'automatisation de la mise à disposition de nouveaux nœuds vers un essaim, vous devez connaître le jeton de jointure correct pour l'essaim ainsi que l'adresse publiée du gestionnaire. Vous pouvez le découvrir en exécutant les commandes suivantes sur l'un des nœuds de gestionnaire existants:

```
# grab the ipaddress:port of the manager (second last line minus the whitespace)
export MANAGER_ADDRESS=$(docker swarm join-token worker | tail -n 2 |  tr -d '[[:space:]]')

# grab the manager and worker token
export MANAGER_TOKEN=$(docker swarm join-token manager -q)
export WORKER_TOKEN=$(docker swarm join-token worker -q)
```

L'option -q génère uniquement le jeton. Sans cette option, vous obtenez la commande complète pour vous inscrire à un essaim.

Ensuite, sur les nouveaux nœuds provisionnés, vous pouvez rejoindre l'essaim en utilisant.

```
docker swarm join --token $WORKER_TOKEN $MANAGER_ADDRESS
```

Bonjour application mondiale

Généralement, vous souhaitez créer une pile de services pour former une application répliquée et orchestrée.

Une application Web moderne typique consiste en une base de données, une API, une interface frontale et un proxy inverse.

Persistance

La base de données a besoin de persistance. Nous avons donc besoin d'un système de fichiers partagé entre tous les nœuds d'un essaim. Cela peut être NAS, serveur NFS, GFS2 ou autre chose. Sa configuration est hors de propos ici. Actuellement, Docker ne contient pas et ne gère pas la persistance dans un essaim. Cet exemple suppose que `/nfs/` emplacement partagé est monté sur tous les nœuds.

Réseau

Pour pouvoir communiquer entre eux, les services d'un essaim doivent être sur le même réseau.

Choisissez une plage IP (ici `10.0.9.0/24`) et un nom de réseau (`hello-network`) et exécutez une commande:

```
docker network create \
  --driver overlay \
  --subnet 10.0.9.0/24 \
  --opt encrypted \
  hello-network
```

Base de données

Le premier service dont nous avons besoin est une base de données. Utilisons postgresql comme exemple. Créez un dossier pour une base de données dans `nfs/postgres` et exécutez ceci:

```
docker service create --replicas 1 --name hello-db \
       --network hello-network -e PGDATA=/var/lib/postgresql/data \
       --mount type=bind,src=/nfs/postgres,dst=/var/lib/postgresql/data \
       kiasaki/alpine-postgres:9.5
```

Notez que nous avons utilisé les `--network hello-network` et `--mount` .

API

La création de l'API est hors de portée de cet exemple, alors supposons que vous avez une image API sous le `username/hello-api` .

```
docker service create --replicas 1 --name hello-api \
       --network hello-network \
       -e NODE_ENV=production -e PORT=80 -e POSTGRESQL_HOST=hello-db \
       username/hello-api
```

Notez que nous avons passé un nom de notre service de base de données. Docker swarm a un serveur DNS intégré, ce qui permet à l'API de se connecter à la base de données en utilisant son nom DNS.

Proxy inverse

Créons le service nginx pour servir notre API à un monde extérieur. Créez des fichiers de configuration nginx dans un emplacement partagé et exécutez-le:

```
docker service create --replicas 1 --name hello-load-balancer \
       --network hello-network \
       --mount type=bind,src=/nfs/nginx/nginx.conf,dst=/etc/nginx/nginx.conf \
       -p 80:80 \
       nginx:1.10-alpine
```

Notez que nous avons utilisé l'option `-p` pour publier un port. Ce port serait disponible pour n'importe quel nœud d'un essaim.

Disponibilité des nœuds

Disponibilité du nœud du mode essaim:

- Active signifie que le planificateur peut affecter des tâches à un nœud.
- Pause signifie que le planificateur n'attribue pas de nouvelles tâches au nœud, mais que les tâches existantes restent en cours d'exécution.
- Drain signifie que le planificateur n'attribue pas de nouvelles tâches au nœud. Le planificateur arrête toutes les tâches existantes et les planifie sur un nœud disponible.

Pour modifier la disponibilité du mode:

```
#Following commands can be used on swarm manager(s)
docker node update --availability drain node-1
#to verify:
docker node ls
```

Promouvoir ou rétrograder les nœuds de l'essaim

Pour promouvoir un noeud ou un ensemble de noeuds, exécutez le `docker node promote` de gestionnaire:

```
docker node promote node-3 node-2

Node node-3 promoted to a manager in the swarm.
Node node-2 promoted to a manager in the swarm.
```

Pour rétrograder un nœud ou un ensemble de nœuds, exécutez la `docker node demote` du `docker node demote` depuis un nœud de gestionnaire:

```
docker node demote node-3 node-2
```

```
Manager node-3 demoted in the swarm.
Manager node-2 demoted in the swarm.
```

Quitter l'essaim

Noeud du travailleur:

```
#Run the following on the worker node to leave the swarm.

docker swarm leave
Node left the swarm.
```

Si le noeud a le rôle de *gestionnaire* , vous recevrez un avertissement sur la gestion du quorum des gestionnaires. Vous pouvez utiliser --force pour laisser sur le noeud du gestionnaire:

```
#Manager Node

docker swarm leave --force
Node left the swarm.
```

Les nœuds qui ont quitté le Swarm apparaîtront toujours dans la sortie du `docker node ls` .

Pour supprimer des nœuds de la liste:

```
docker node rm node-2

node-2
```

Lire Mode essaim Docker en ligne: https://riptutorial.com/fr/docker/topic/749/mode-essaim-docker

Chapitre 27: Modes Docker --net (pont, hots, conteneur mappé et aucun).

Introduction

Commencer

Mode pont C'est un mode par défaut et associé au pont docker0. Placez le conteneur sur un espace de noms réseau complètement distinct.

Mode hôte Lorsque le conteneur est juste un processus exécuté sur un hôte, nous attacherons le conteneur à la carte réseau hôte.

Mode conteneur conteneurisé Ce mode mappe essentiellement un nouveau conteneur dans une pile réseau de conteneurs existante. Il est également appelé «conteneur en mode conteneur».

Aucun Il dit à docker de mettre le conteneur dans sa propre pile réseau sans configuration

Examples

Mode pont, mode hôte et mode conteneur mappé

Mode pont

```
$ docker run -d --name my_app -p 10000:80 image_name
```

Notez que nous n'avons pas eu à spécifier **--net = bridge** car il s'agit du mode de travail par défaut pour docker. Cela permet d'exécuter plusieurs conteneurs sur le même hôte sans attribution de port dynamique. Ainsi, le mode **BRIDGE** évite la collision entre les ports et est sûr car chaque conteneur exécute son propre espace de noms de réseau privé.

Mode hôte

```
$ docker run -d --name my_app -net=host image_name
```

Comme il utilise l'espace de noms du réseau hôte, aucune configuration particulière n'est requise, mais peut entraîner des problèmes de sécurité.

Mode conteneur mappé

Ce mode mappe essentiellement un nouveau conteneur dans une pile de réseau de conteneurs existante. Cela implique que les ressources réseau telles que l'adresse IP et les mappages de ports du premier conteneur seront partagées par le deuxième conteneur. Ceci est également appelé en tant que mode "conteneur en conteneur". Supposons que vous ayez deux composants

comme web_container_1 et web_container_2 et que vous exécutiez web_container_2 en mode conteneur mappé. Commençons par télécharger web_container_1 et l'exécuter en mode détaché avec la commande suivante,

```
$ docker run -d --name web1 -p 80:80 USERNAME/web_container_1
```

Une fois téléchargé, jetons un coup d'oeil et assurons-le. Ici, nous avons simplement mappé un port dans un conteneur qui s'exécute dans le mode de pont par défaut. Maintenant, exécutons un deuxième conteneur en mode conteneur mappé. Nous allons le faire avec cette commande.

```
$ docker run -d --name web2 --net=container:web1 USERNAME/web_container_2
```

Maintenant, si vous obtenez simplement les informations d'interface sur les deux composants, vous obtiendrez la même configuration réseau. Cela inclut en fait le mode HOST qui correspond aux informations exactes de l'hôte. Le premier contaienr a fonctionné en mode pont par défaut et le deuxième conteneur est exécuté en mode conteneur mappé. Nous pouvons obtenir des résultats très similaires en démarrant le premier conteneur en mode hôte et le deuxième conteneur en mode conteneur mappé.

Lire Modes Docker --net (pont, hots, conteneur mappé et aucun). en ligne: https://riptutorial.com/fr/docker/topic/9643/modes-docker---net--pont--hots--conteneur-mappe-et-aucun--

Chapitre 28: Plusieurs processus dans une instance de conteneur

Remarques

Chaque conteneur devrait généralement héberger un processus. Si vous avez besoin de plusieurs processus dans un conteneur (par exemple, un serveur SSH pour vous connecter à votre instance de conteneur en cours d'exécution), vous pouvez avoir l'idée d'écrire votre propre script shell qui lance ces processus. Dans ce cas, vous devez faire attention à la gestion de `SIGNAL` par vous-même (par exemple, rediriger un `SIGINT` capturé vers les processus enfants de votre script). Ce n'est pas vraiment ce que vous voulez. Une solution simple consiste à utiliser `supervisord` tant que processus racine de conteneur qui prend en charge la gestion de `SIGNAL` et la durée de vie de ses processus enfants.

Mais gardez à l'esprit que ce n'est pas la "voie des dockers". Pour obtenir cet exemple dans le mode docker, connectez-vous à l' `docker host` (la machine sur laquelle le conteneur s'exécute) et exécutez `docker exec -it container_name /bin/bahs` . Cette commande vous ouvre un shell dans le conteneur comme le ferait ssh.

Examples

Dockerfile + supervisord.conf

Pour exécuter plusieurs processus, par exemple un serveur Web Apache avec un démon SSH dans le même conteneur, vous pouvez utiliser `supervisord` .

Créez votre fichier de configuration `supervisord.conf` comme:

```
[supervisord]
nodaemon=true

[program:sshd]
command=/usr/sbin/sshd -D

[program:apache2]
command=/bin/bash -c "source /etc/apache2/envvars && exec /usr/sbin/apache2 -DFOREGROUND"
```

Puis créez un `Dockerfile` comme:

```
FROM ubuntu:16.04
RUN apt-get install -y openssh-server apache2 supervisor
RUN mkdir -p /var/lock/apache2 /var/run/apache2 /var/run/sshd /var/log/supervisor
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
CMD ["/usr/bin/supervisord"]
```

Ensuite, vous pouvez construire votre image:

```
docker build -t supervisord-test .
```

Ensuite, vous pouvez l'exécuter:

```
$ docker run -p 22 -p 80 -t -i supervisord-test
2016-07-26 13:15:21,101 CRIT Supervisor running as root (no user in config file)
2016-07-26 13:15:21,101 WARN Included extra file     "/etc/supervisor/conf.d/supervisord.conf"
during parsing
2016-07-26 13:15:21,112 INFO supervisord started with pid 1
2016-07-26 13:15:21,113 INFO spawned: 'sshd' with pid 6
2016-07-26 13:15:21,115 INFO spawned: 'apache2' with pid 7
...
```

Lire Plusieurs processus dans une instance de conteneur en ligne:
https://riptutorial.com/fr/docker/topic/4053/plusieurs-processus-dans-une-instance-de-conteneur

Chapitre 29: Points de contrôle et de restauration

Examples

Compiler le menu fixe avec le point de contrôle et la restauration activés (Ubuntu)

Pour compiler docker, il est recommandé d'avoir au moins **2 Go de RAM** . Même si cela échoue parfois, il vaut mieux opter pour **4 Go à la** place.

1. assurez-vous que git et make sont installés

```
sudo apt-get install make git-core -y
```

2. installer un nouveau noyau (au moins 4.2)

```
sudo apt-get install linux-generic-lts-xenial
```

3. redémarrer la machine pour que le nouveau noyau soit actif

```
sudo reboot
```

4. compiler `criu` qui est nécessaire pour exécuter le `docker checkpoint`

```
sudo apt-get install libprotobuf-dev libprotobuf-c0-dev protobuf-c-compiler protobuf-
compiler python-protobuf libnl-3-dev libcap-dev -y
wget http://download.openvz.org/criu/criu-2.4.tar.bz2 -O - | tar -xj
cd criu-2.4
make
make install-lib
make install-criu
```

5. vérifier si chaque exigence est remplie pour exécuter criu

```
sudo criu check
```

6. compiler le docker expérimental (il faut docker pour compiler docker)

```
cd ~
wget -qO- https://get.docker.com/ | sh
sudo usermod -aG docker $(whoami)
```

- **À ce stade, nous devons nous déconnecter et nous reconnecter pour avoir un démon docker. Après relog continuer avec l'étape de compilation**

```
git clone https://github.com/boucher/docker
cd docker
git checkout docker-checkpoint-restore
make #that will take some time - drink a coffee
DOCKER_EXPERIMENTAL=1 make binary
```

7. Nous avons maintenant un menu fixe compilé. Permet de déplacer les binaires. Assurez-vous de remplacer `<version>` par la version installée

```
sudo service docker stop
sudo cp $(which docker) $(which docker)_ ; sudo cp ./bundles/latest/binary-client/docker-
<version>-dev $(which docker)
sudo cp $(which docker-containerd) $(which docker-containerd)_ ; sudo cp
./bundles/latest/binary-daemon/docker-containerd $(which docker-containerd)
sudo cp $(which docker-containerd-ctr) $(which docker-containerd-ctr)_ ; sudo cp
./bundles/latest/binary-daemon/docker-containerd-ctr $(which docker-containerd-ctr)
sudo cp $(which docker-containerd-shim) $(which docker-containerd-shim)_ ; sudo cp
./bundles/latest/binary-daemon/docker-containerd-shim $(which docker-containerd-shim)
sudo cp $(which dockerd) $(which dockerd)_ ; sudo cp ./bundles/latest/binary-
daemon/dockerd $(which dockerd)
sudo cp $(which docker-runc) $(which docker-runc)_ ; sudo cp ./bundles/latest/binary-
daemon/docker-runc $(which docker-runc)
sudo service docker start
```

Ne vous inquiétez pas - nous avons sauvegardé les anciens fichiers binaires. Ils sont toujours là mais avec un trait de soulignement ajouté à ses noms (`docker_`).

Félicitations, vous avez maintenant un docker expérimental avec la possibilité de contrôler un conteneur et de le restaurer.

S'il vous plaît noter que les fonctionnalités expérimentales ne sont pas prêts pour la production

Point de contrôle et restauration d'un conteneur

```
# create docker container
export cid=$(docker run -d --security-opt seccomp:unconfined busybox /bin/sh -c 'i=0; while
true; do echo $i; i=$(expr $i + 1); sleep 1; done')

# container is started and prints a number every second
# display the output with
docker logs $cid

# checkpoint the container
docker checkpoint create $cid checkpointname

# container is not running anymore
docker np

# lets pass some time to make sure

# resume container
docker start $cid --checkpoint=checkpointname

# print logs again
```

```
docker logs $cid
```

Lire Points de contrôle et de restauration en ligne:
https://riptutorial.com/fr/docker/topic/5291/points-de-controle-et-de-restauration

Chapitre 30: Procédure de configuration d'un réplica Mongo à trois nœuds à l'aide de l'image Docker et de Provisioned à l'aide de Chef

Introduction

Cette documentation explique comment créer un jeu de répliques Mongo à trois nœuds à l'aide de Docker Image et approvisionnement automatique à l'aide de Chef.

Examples

Étape de construction

Pas:

1. Générez un fichier de clés Base 64 pour l'authentification du noeud Mongo. Placez ce fichier dans chef data_bags
2. Accédez au chef de supermarché et téléchargez le livre de recettes docker. Générez un livre de recettes personnalisé (par exemple, custom_mongo) et ajoutez depend 'docker', '~> 2.0' au metadata.rb de votre livre de recettes
3. Créer un attribut et une recette dans votre livre de recettes personnalisé
4. Initialise Mongo pour former le cluster Rep Set

Étape 1: Créer un fichier de clé

créer data_bag appelé mongo-keyfile et élément appelé keyfile. Ce sera dans le répertoire data_bags en chef. Le contenu de l'article sera comme ci-dessous

```
openssl rand -base64 756 > <path-to-keyfile>
```

contenu de l'élément de fichier clé

```
{
  "id": "keyfile",
  "comment": "Mongo Repset keyfile",
  "key-file": "generated base 64 key above"
}
```

Étape 2: Téléchargez le livre de recettes de docker sur le marché du chef et créez un livre de recettes personnalisé_mongo

```
knife cookbook site download docker
knife cookbook create custom_mongo
```

dans metadat.rb de custom_mongo add

```
depends          'docker', '~> 2.0'
```

Étape 3: créer un attribut et une recette

Les attributs

```
default['custom_mongo']['mongo_keyfile'] = '/data/keyfile'
default['custom_mongo']['mongo_datadir'] = '/data/db'
default['custom_mongo']['mongo_datapath'] = '/data'
default['custom_mongo']['keyfilename'] = 'mongodb-keyfile'
```

Recette

```
#
# Cookbook Name:: custom_mongo
# Recipe:: default
#
# Copyright 2017, Innocent Anigbo
#
# All rights reserved - Do Not Redistribute
#

data_path = "#{node['custom_mongo']['mongo_datapath']}"
data_dir = "#{node['custom_mongo']['mongo_datadir']}"
key_dir = "#{node['custom_mongo']['mongo_keyfile']}"
keyfile_content = data_bag_item('mongo-keyfile', 'keyfile')
keyfile_name = "#{node['custom_mongo']['keyfilename']}"

#chown of keyfile to docker user
execute 'assign-user' do
 command "chown 999 #{key_dir}/#{keyfile_name}"
 action :nothing
end

#Declaration to create Mongo data DIR and Keyfile DIR
%W[ #{data_path} #{data_dir} #{key_dir} ].each do |path|
directory path do
  mode '0755'
  end
end

#declaration to copy keyfile from data_bag to keyfile DIR on your mongo server
file "#{key_dir}/#{keyfile_name}" do
  content keyfile_content['key-file']
  group 'root'
  mode '0400'
  notifies :run, 'execute[assign-user]', :immediately
end

#Install docker
docker_service 'default' do
  action [:create, :start]
```

```
end

#Install mongo 3.4.2
docker_image 'mongo' do
  tag '3.4.2'
  action :pull
end
```

Créer un rôle appelé mongo-role dans le répertoire des rôles

```
{
  "name": "mongo-role",
  "description": "mongo DB Role",
  "run_list": [
    "recipe[custom_mongo]"
  ]
}
```

Ajouter le rôle ci-dessus à la liste d'exécution des trois nœuds mongo

```
knife node run_list add FQDN_of_node_01 'role[mongo-role]'
knife node run_list add FQDN_of_node_02 'role[mongo-role]'
knife node run_list add FQDN_of_node_03 'role[mongo-role]'
```

Étape 4: Initialiser le Mongo à trois nœuds pour former un repset

Je suppose que le rôle ci-dessus a déjà été appliqué aux trois nœuds Mongo. Sur le noeud 01 uniquement, démarrez Mongo avec --auth pour activer l'authentification

```
docker run --name mongo -v /data/db:/data/db -v /data/keyfile:/opt/keyfile --hostname="mongo-
01.example.com" -p 27017:27017 -d mongo:3.4.2 --keyFile /opt/keyfile/mongodb-keyfile --auth
```

Accéder au shell interactif du conteneur docker en cours d'exécution sur le noeud 01 et Créer un utilisateur admin

```
docker exec -it mongo /bin/sh
    mongo
    use admin
    db.createUser( {
         user: "admin-user",
         pwd: "password",
         roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
       });
```

Créer un utilisateur root

```
db.createUser( {
         user: "RootAdmin",
         pwd: "password",
         roles: [ { role: "root", db: "admin" } ]
       });
```

Arrêtez et supprimez le conteneur Docker créé ci-dessus sur le noeud 01. Cela n'affectera pas les

données et le fichier de clés dans le DIR de l'hôte. Après avoir supprimé Mongo, redémarrez le nœud 01, mais cette fois avec repset flag

```
docker rm -fv mongo
docker run --name mongo-uat -v /data/db:/data/db -v /data/keyfile:/opt/keyfile --
hostname="mongo-01.example.com" -p 27017:27017 -d mongo:3.4.2 --keyFile /opt/keyfile/mongodb-
keyfile --replSet "rs0"
```

lancez maintenant mongo sur les nœuds 02 et 03 avec le drapeau rep set

```
docker run --name mongo -v /data/db:/data/db -v /data/keyfile:/opt/keyfile --hostname="mongo-
02.example.com" -p 27017:27017 -d mongo:3.4.2 --keyFile /opt/keyfile/mongodb-keyfile --replSet
"rs0"
docker run --name mongo -v /data/db:/data/db -v /data/keyfile:/opt/keyfile --hostname="mongo-
03.example.com" -p 27017:27017 -d mongo:3.4.2 --keyFile /opt/keyfile/mongodb-keyfile --replSet
"rs0"
```

Authentifiez-vous avec l'utilisateur root sur le nœud 01 et lancez le jeu de réplicas

```
use admin
db.auth("RootAdmin", "password");
rs.initiate()
```

Sur le noeud 01, ajoutez les noeuds 2 et 3 au jeu de réplicas pour former le cluster repset0

```
rs.add("mongo-02.example.com")
rs.add("mongo-03.example.com")
```

Essai

Sur le primaire, exécutez db.printSlaveReplicationInfo () et observez SyncedTo et Behind the main time. Le plus tard devrait être 0 sec comme ci-dessous

Sortie

```
rs0:PRIMARY> db.printSlaveReplicationInfo()
   source: mongo-02.example.com:27017
           syncedTo: Mon Mar 27 2017 15:01:04 GMT+0000 (UTC)
           0 secs (0 hrs) behind the primary
   source: mongo-03.example.com:27017
           syncedTo: Mon Mar 27 2017 15:01:04 GMT+0000 (UTC)
           0 secs (0 hrs) behind the primary
```

J'espère que ça aidera quelqu'un

Lire Procédure de configuration d'un réplica Mongo à trois nœuds à l'aide de l'image Docker et de Provisioned à l'aide de Chef en ligne: https://riptutorial.com/fr/docker/topic/10014/procedure-de-configuration-d-un-replica-mongo-a-trois-nouds-a-l-aide-de-l-image-docker-et-de-provisioned-a-l-aide-de-chef

Chapitre 31: Registre privé / sécurisé Docker avec API v2

Introduction

Un registre de docker privé et sécurisé au lieu d'un Docker Hub. Les compétences de base de docker sont requises.

Paramètres

Commander	Explication
sudo docker run -p 5000: 5000	Démarrez un conteneur Docker et liez le port 5000 du conteneur au port 5000 de la machine physique.
--nom de registre	Nom du conteneur (utilisé pour améliorer la lisibilité de «docker ps»).
-v 'pwd' / certs: / certs	Liez CURRENT_DIR / certs de la machine physique sur / certs du conteneur (comme un «dossier partagé»).
-e REGISTRY_HTTP_TLS_CERTIFICATE = / certs / server.crt	Nous spécifions que le registre doit utiliser le fichier /certs/server.crt pour démarrer. (variable env)
-e REGISTRY_HTTP_TLS_KEY = / certs / server.key	Identique pour la clé RSA (server.key).
-v / root / images: / var / lib / registry /	Si vous souhaitez enregistrer toutes vos images de registre, vous devez le faire sur la machine physique. Ici, nous sauvegardons toutes les images sur / root / images sur la machine physique. Si vous faites cela, vous pouvez arrêter et redémarrer le registre sans perdre aucune image.
registre: 2	Nous spécifions que nous aimerions extraire l'image du registre de docker hub (ou localement), et nous ajoutons «2» car nous voulons installer la version 2 du registre.

Remarques

Comment installer un moteur de docker (appelé client sur ce tutoriel)

Comment générer un certificat SSL auto-signé

Examples

Générer des certificats

Génère une clé privée RSA: `openssl genrsa -des3 -out server.key 4096`

Openssl devrait demander une phrase de passe à cette étape. Notez que nous n'utiliserons que des certificats pour la communication et l'authentification, sans phrase secrète. Utilisez simplement 123456 par exemple.

Générer la demande de signature de certificat: `openssl req -new -key server.key -out server.csr`

Cette étape est importante car vous devrez fournir des informations sur les certificats. Les informations les plus importantes sont «Nom commun», c'est-à-dire le nom de domaine, utilisé pour la communication entre le registre de docker privé et tous les autres ordinateurs. Exemple: mydomain.com

Supprimer le mot de passe de la clé privée RSA: `cp server.key server.key.org && openssl rsa -in server.key.org -out server.key`

Comme je l'ai dit, nous allons nous concentrer sur le certificat sans phrase secrète. Soyez donc prudent avec tous les fichiers de votre clé (.key, .csr, .crt) et conservez-les dans un endroit sûr.

Générez le certificat auto-signé: `openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt`

Vous avez maintenant deux fichiers essentiels, *server.key* et *server.crt* , nécessaires à l'authentification du registre privé.

Exécutez le registre avec un certificat auto-signé

Pour exécuter le registre privé (en toute sécurité), vous devez générer un certificat auto-signé, vous pouvez vous référer à l'exemple précédent pour le générer.

Pour mon exemple, je mets *server.key* et *server.crt* dans / root / certs

Avant d'exécuter la commande docker, vous devriez être placé (utilisez `cd`) dans le répertoire contenant le dossier *certs* . Si vous n'êtes pas et que vous essayez d'exécuter la commande, vous recevrez une erreur comme

> level = fatal msg = "open /certs/server.crt: pas de tel fichier ou répertoire"

Quand vous êtes (`cd /root` dans mon exemple), vous pouvez essentiellement démarrer le registre sécurisé / privé en utilisant: `sudo docker run -p 5000:5000 --restart=always --name registry -v `pwd`/certs:/certs -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/server.crt -e REGISTRY_HTTP_TLS_KEY=/certs/server.key -v /root/Documents:/var/lib/registry/ registry:2`
Des explications sur la commande sont disponibles dans la partie Paramètres.

Tirez ou poussez depuis un client Docker

Lorsque vous obtenez un registre en cours d'exécution, vous pouvez extraire ou diffuser des images. Pour cela, vous avez besoin du fichier *server.crt* dans un dossier spécial sur votre client docker. Le certificat vous permet de vous authentifier avec le registre, puis de chiffrer la communication.

Copiez *server.crt* de la machine de registre dans /etc/docker/certs.d/mydomain.com:5000/ sur votre ordinateur client. Et puis renommez-le en *ca-certificates.crt* : `mv /etc/docker/certs.d/mydomain.com:5000/server.crt /etc/docker/certs.d/mydomain.com:5000/ca-certificates.crt`

À ce stade, vous pouvez extraire ou pousser des images à partir de votre registre privé:
PULL: `docker pull mydomain.com:5000/nginx` ou
POUSSER :

1. Obtenez une image officielle de hub.docker.com: `docker pull nginx`
2. Marquez cette image avant de la placer dans un registre privé: `docker tag IMAGE_ID mydomain.com:5000/nginx` (utilisez les `docker images` pour obtenir le IMAGE_ID)
3. Poussez l'image dans le registre: `docker push mydomain.com:5000/nginx`

Lire Registre privé / sécurisé Docker avec API v2 en ligne:
https://riptutorial.com/fr/docker/topic/8707/registre-prive---securise-docker-avec-api-v2

Chapitre 32: Réseau Docker

Examples

Comment trouver l'ip hôte du conteneur

Vous devez connaître l'adresse IP du conteneur s'exécutant sur l'hôte pour pouvoir, par exemple, vous connecter au serveur Web qui y est exécuté.

`docker-machine` est ce qui est utilisé sur MacOSX et Windows.

Tout d'abord, listez vos machines:

```
$ docker-machine ls

NAME      ACTIVE   DRIVER       STATE     URL                         SWARM
default   *        virtualbox   Running   tcp://192.168.99.100:2376
```

Ensuite, sélectionnez l'une des machines (celle par défaut s'appelle default) et:

```
$ docker-machine ip default

192.168.99.100
```

Créer un réseau Docker

```
docker network create app-backend
```

Cette commande créera un réseau ponté simple appelé `appBackend` . Aucun conteneur n'est attaché à ce réseau par défaut.

Liste de réseaux

```
docker network ls
```

Cette commande répertorie tous les réseaux créés sur l'hôte Docker local. Il inclut le réseau de `bridge` pont par défaut, le réseau hôte `host` et le réseau null `null` . Tous les conteneurs par défaut sont attachés au réseau de `bridge` pont par défaut.

Ajouter un conteneur au réseau

```
docker network connect app-backend myAwesomeApp-1
```

Cette commande associe le `myAwesomeApp-1` au réseau d' `app-backend` . Lorsque vous ajoutez un conteneur à un réseau défini par l'utilisateur, le résolveur DNS intégré (qui n'est pas un serveur

DNS complet et n'est pas exportable) permet à chaque conteneur du réseau de résoudre chaque conteneur sur le même réseau. Ce résolveur DNS simple n'est pas disponible sur le réseau de `bridge` pont par défaut.

Détachez le conteneur du réseau

```
docker network disconnect app-backend myAwesomeApp-1
```

Cette commande détache le `myAwesomeApp-1` du réseau d' `app-backend` . Le conteneur ne sera plus en mesure de communiquer avec d'autres conteneurs sur le réseau dont il a été déconnecté, ni d'utiliser le résolveur DNS intégré pour rechercher d'autres conteneurs sur le réseau dont il a été déconnecté.

Supprimer un réseau Docker

```
docker network rm app-backend
```

Cette commande supprime le réseau d' `app-backend` défini par l'utilisateur de l'hôte Docker. Tous les conteneurs sur le réseau non connectés via un autre réseau perdront la communication avec les autres conteneurs. Il est impossible de supprimer le réseau de `bridge` pont par défaut, le réseau `host` hôte ou le réseau `null` null.

Inspecter un réseau Docker

```
docker network inspect app-backend
```

Cette commande les détails de sortie sur l' `app-backend` - `app-backend` réseau.

La sortie de cette commande doit ressembler à:

```
[
    {
        "Name": "foo",
        "Id": "a0349d78c8fd7c16f5940bdbaf1adec8d8399b8309b2e8a969bd4e3226a6fc58",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1/16"
                }
            ]
        },
        "Internal": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
```

```
    }
]
```

Lire Réseau Docker en ligne: https://riptutorial.com/fr/docker/topic/3221/reseau-docker

Chapitre 33: Restreindre l'accès au réseau de conteneurs

Remarques

Exemple de réseau fixe qui bloque le trafic. Utilisez-le comme réseau lors du démarrage du conteneur avec `--net` ou `docker network connect` .

Examples

Bloquer l'accès au LAN et à la sortie

```
docker network create -o "com.docker.network.bridge.enable_ip_masquerade"="false" lan-restricted
```

- Blocs
  - LAN local
  - l'Internet
- Ne bloque pas
  - Hôte exécutant le démon `10.0.1.10:22` (exemple d'accès au `10.0.1.10:22`)

Bloquer l'accès à d'autres conteneurs

```
docker network create -o "com.docker.network.bridge.enable_icc"="false" icc-restricted
```

- Blocs
  - Conteneurs accédant à d'autres conteneurs sur le même réseau `icc-restricted` .
- Ne bloque pas
  - Accès à l'hôte exécutant le démon docker
  - LAN local
  - l'Internet

Bloquer l'accès des conteneurs à l'hôte local exécutant le démon docker

```
iptables -I INPUT -i docker0 -m addrtype --dst-type LOCAL -j DROP
```

- Blocs
  - Accès à l'hôte exécutant le démon docker
- Ne bloque pas
  - Trafic conteneur à conteneur
  - LAN local
  - l'Internet
  - Réseaux de dockers personnalisés n'utilisant pas `docker0`

Bloquer l'accès des conteneurs à l'hôte local exécutant le démon docker (réseau personnalisé)

```
docker network create --subnet=192.168.0.0/24 --gateway=192.168.0.1 --ip-range=192.168.0.0/25
local-host-restricted
iptables -I INPUT -s 192.168.0.0/24 -m addrtype --dst-type LOCAL -j DROP
```

Crée un réseau appelé `local-host-restricted` auquel:

- Blocs
  - Accès à l'hôte exécutant le démon docker
- Ne bloque pas
  - Trafic conteneur à conteneur
  - LAN local
  - l'Internet
  - Accès provenant d'autres réseaux de docker

Les réseaux personnalisés ont des noms de pont comme `br-15bbe9bb5bf5` , nous utilisons donc son sous-réseau à la place.

Lire Restreindre l'accès au réseau de conteneurs en ligne:
https://riptutorial.com/fr/docker/topic/6331/restreindre-l-acces-au-reseau-de-conteneurs

Chapitre 34: Sécurité

Introduction

Afin de garder nos images à jour pour les correctifs de sécurité, nous devons savoir à partir de quelle image de base nous dépendons

Examples

Comment trouver à partir de quelle image notre image provient

A titre d'exemple, permet de regarder un conteneur Wordpress

Le fichier Docker commence par FROM php: 5.6-apache

nous allons donc au fichier Dockerfile mentionné ci-dessus https://github.com/docker-library/php/blob/master/5.6/apache/Dockerfile

et nous trouvons FROM debian: jessie Cela signifie donc que nous avons un correctif de sécurité pour Debian Jessie, nous devons reconstruire notre image.

Lire Sécurité en ligne: https://riptutorial.com/fr/docker/topic/8077/securite

Chapitre 35: Statistiques Docker tous les conteneurs en cours d'exécution

Examples

Statistiques Docker tous les conteneurs en cours d'exécution

```
sudo docker stats $(sudo docker inspect -f "{{ .Name }}" $(sudo docker ps -q))
```

Affiche l'utilisation du processeur en direct de tous les conteneurs en cours d'exécution.

Lire Statistiques Docker tous les conteneurs en cours d'exécution en ligne: https://riptutorial.com/fr/docker/topic/5863/statistiques-docker-tous-les-conteneurs-en-cours-d-execution

Chapitre 36: transmettre des données secrètes à un conteneur en cours d'exécution

Examples

façons de transmettre des secrets dans un conteneur

Le moyen peu sécurisé (parce que `docker inspect` le montrera) est de passer une variable d'environnement à

```
docker run
```

tel que

```
docker run -e password=abc
```

ou dans un fichier

```
docker run --env-file myfile
```

où mon fichier peut contenir

```
password1=abc password2=def
```

il est également possible de les mettre dans un volume

```
docker run -v $(pwd)/my-secret-file:/secret-file
```

de meilleurs moyens, utiliser

keywhiz https://square.github.io/keywhiz/

vault https://www.hashicorp.com/blog/vault.html

etcd avec la crypte https://xordataexchange.github.io/crypt/

Lire transmettre des données secrètes à un conteneur en cours d'exécution en ligne: https://riptutorial.com/fr/docker/topic/6481/transmettre-des-donnees-secretes-a-un-conteneur-en-cours-d-execution

Chapitre 37: Volumes de données et conteneurs de données

Examples

Conteneurs de données uniquement

Les conteneurs de données uniquement sont obsolètes et sont désormais considérés comme un anti-modèle!

Auparavant, avant la sous-commande de `volume` de Docker et avant qu'il soit possible de créer des volumes nommés, Docker supprimait les volumes lorsqu'il n'y avait plus de références à ceux-ci dans aucun conteneur. Les conteneurs de données uniquement sont obsolètes car Docker offre désormais la possibilité de créer des volumes nommés, ainsi que beaucoup plus d'utilitaire via les différentes sous-commandes `docker volume` . Les conteneurs de données uniquement sont désormais considérés comme un anti-pattern pour cette raison.

De nombreuses ressources sur le Web datant des deux dernières années mentionnent l'utilisation d'un modèle appelé «conteneur de données uniquement», qui est simplement un conteneur Docker qui existe uniquement pour conserver une référence à un volume de données.

Rappelez-vous que dans ce contexte, un "volume de données" est un volume Docker qui n'est pas monté depuis l'hôte. Pour clarifier, un "volume de données" est un volume créé avec la directive `VOLUME` Dockerfile ou en utilisant le commutateur `-v` sur la ligne de commande dans une commande `docker run` , en particulier au format `-v /path/on/container` . Par conséquent, un "conteneur de données uniquement" est un conteneur dont le seul but est de `--volumes-from` un volume de données, qui est utilisé par l' `--volumes-from` dans une commande d' `docker run` . Par exemple:

```
docker run -d --name "mysql-data" -v "/var/lib/mysql" alpine /bin/true
```

Lorsque la commande ci-dessus est exécutée, un "conteneur de données uniquement" est créé. C'est simplement un conteneur vide auquel est attaché un volume de données. Il était alors possible d'utiliser ce volume dans un autre conteneur comme ceci:

```
docker run -d --name="mysql" --volumes-from="mysql-data" mysql
```

Le conteneur `mysql` contient maintenant le même volume qui se trouve également dans `mysql-data` .

Docker fournissant désormais la sous-commande `volume` et les volumes nommés, ce modèle est désormais obsolète et non recommandé.

Pour commencer avec la sous-commande `volume` et les volumes nommés, voir Création d'un volume nommé

Créer un volume de données

```
docker run -d --name "mysql-1" -v "/var/lib/mysql" mysql
```

Cette commande crée un nouveau conteneur à partir de l'image `mysql` . Il crée également un nouveau volume de données, qu'il monte ensuite dans le conteneur dans `/var/lib/mysql` . Ce volume aide les données à l'intérieur de celui-ci à persister au-delà de la durée de vie du conteneur. C'est-à-dire que lorsqu'un conteneur est supprimé, ses modifications du système de fichiers sont également supprimées. Si une base de données stockait des données dans le conteneur et que le conteneur était supprimé, toutes ces données sont également supprimées. Les volumes persisteront dans un emplacement particulier au-delà de la suppression de son conteneur.

Il est possible d'utiliser le même volume dans plusieurs conteneurs avec l' `--volumes-from` ligne `--volumes-from` commande `--volumes-from` :

```
docker run -d --name="mysql-2" --volumes-from="mysql-1" mysql
```

Le conteneur `mysql-2` est désormais associé au volume de données de `mysql-1` , utilisant également le chemin `/var/lib/mysql` .

Lire Volumes de données et conteneurs de données en ligne:
https://riptutorial.com/fr/docker/topic/3224/volumes-de-donnees-et-conteneurs-de-donnees

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Docker	abaracedo, Aminadav, Braiam, Carlos Rafael Ramirez, Community, ganesshkumar, HankCa, Josha Inglis, L0j1k, mohan08p, Nathaniel Ford, schumacherj, Siddharth Srinivasan, SztupY, Vishrant
2	API Docker Engine	Ashish Bista, atv, BMitch, L0j1k, Radoslav Stoyanov, SztupY
3	Classement du contenu de Dockerfile	akhyar, Philip
4	Comment déboguer lorsque la construction du docker échoue	user2915097
5	Concept de volumes Docker	Amit Poonia, Rob Bednark, serieznyi
6	Connexion des conteneurs	Jett Jones
7	Conteneurs de course	abaracedo, Adri C.S., AlcaDotS, atv, Binary Nerd, BMitch, Camilo Silva, Carlos Rafael Ramirez, cizixs, cjsimon, Claudiu, ElMesa, Emil Burzo, enderland, Felipe Plets, ganesshkumar, Gergely Fehérvári, ISanych, L0j1k, Nathan Arthur, Patrick Auld, RoyB, ssice, SztupY, Thomasleveil, tommyyards, VanagaS, Wolfgang, zinking
8	Créer un service avec persistance	Carlos Rafael Ramirez, Vanuan
9	Déboguer un conteneur	allprog, Binary Nerd, foraidt, L0j1k, Nathaniel Ford, user2915097, yadutaf
10	Docker dans Docker	Ohmen
11	Docker Data Volumes	James Hewitt, L0j1k, NRKirby, Nuno Curado, Scott Coates, t3h2mas
12	docker inspecte l'obtention de	user2915097

	différents champs pour la clé: valeur et éléments de la liste	
13	Docker Machine	Amine24h, kubanczyk, Nik Rahmel, user2915097, yadutaf
14	Docker Registry	Ashish Bista, L0j1k
15	Dockerfiles	BMitch, foraidt, k0pernikus, kubanczyk, L0j1k, ob1, Ohmen, rosysnake, satsumas, Stephen Leppik, Thiago Almeida, Wassim Dhif, yadutaf
16	Enregistrement	Jilles van Gurp, Vanuan
17	Événements Docker	Nathaniel Ford, user2915097
18	exécuter consul dans docker 1.12 essaim	Jilles van Gurp
19	Exécution de l'application Simple Node.js	Siddharth Srinivasan
20	Gérer des images	akhyar, Björn Enochsson, dsw88, L0j1k, Nathan Arthur, Nathaniel Ford, Szymon Biliński, user2915097, Wolfgang, zygimantus
21	Gestion des conteneurs	akhyar, atv, Binary Nerd, BrunoLM, Carlos Rafael Ramirez, Emil Burzo, Felipe Plets, ganesshkumar, L0j1k, Matt, Nathaniel Ford, Rafal Wiliński, Sachin Malhotra, serieznyi, sk8terboi87 ツ, tommyyards, user2915097, Victor Oliveira Antonino, Wolfgang, Xavier Nicollet, zygimantus
22	Images de construction	cjsimon, ETL, Ken Cochrane, L0j1k, Nathan Arthur, Nathaniel Ford, Nour Chawich, SztupY, user2915097, Wolfgang
23	Inspection d'un conteneur en cours d'exécution	AlcaDotS, devopskata, Felipe Plets, h3nrik, Jilles van Gurp, L0j1k, Milind Chawre, Nik Rahmel, Stephen Leppik, user2915097, yadutaf
24	Iptables avec Docker	Adrien Ferrand
25	Les services en cours d'exécution	Mateusz Mrozewski, Philip
26	Mode essaim Docker	abronan, Christian, Farhad Farahi, Jilles van Gurp, kstromeiraos , kubanczyk, ob1, Philip, Vanuan
27	Modes Docker --net (pont, hots,	mohan08p

	conteneur mappé et aucun).	
28	Plusieurs processus dans une instance de conteneur	h3nrik, Ohmen, Xavier Nicollet
29	Points de contrôle et de restauration	Bastian, Fuzzyma
30	Procédure de configuration d'un réplica Mongo à trois nœuds à l'aide de l'image Docker et de Provisioned à l'aide de Chef	Innocent Anigbo
31	Registre privé / sécurisé Docker avec API v2	bastien enjalbert, kubanczyk
32	Réseau Docker	HankCa, L0j1k, Nathaniel Ford
33	Restreindre l'accès au réseau de conteneurs	xeor
34	Sécurité	user2915097
35	Statistiques Docker tous les conteneurs en cours d'exécution	Kostiantyn Rybnikov
36	transmettre des données secrètes à un conteneur en cours d'exécution	user2915097
37	Volumes de données et conteneurs de données	GameScripting, L0j1k, melihovv