FREE eBook

LEARNING Docker

Free unaffiliated eBook created from
Stack Overflow contributors.

#docker

Table of Contents

About

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

Chapter 1: Getting started with Docker

Remarks

Docker is an open-source project that automates the deployment of applications inside software containers. These application containers are similar to lightweight virtual machines, as they can be run in isolation to each other and the running host.

Docker requires features present in recent linux kernels to function properly, therefore on Mac OSX and Windows host a virtual machine running linux is required for docker to operate properly. Currently the main method of installing and setting up this virtual machine is via Docker Toolbox that is using VirtualBox internally, but there are plans to integrate this functionality into docker itself, using the native virtualisation features of the operating system. On Linux systems docker run natively on the host itself.

Versions

Version	Release Date
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

Installing Docker on Mac OS X

Requirements: OS X 10.8 “Mountain Lion” or newer required to run Docker.

While the docker binary can run natively on Mac OS X, to build and host containers you need to run a Linux virtual machine on the box.

1.12.0

Since version 1.12 you don't need to have a separate VM to be installed, as Docker can use the native `Hypervisor.framework` functionality of OSX to start up a small Linux machine to act as backend.

To install docker follow the following steps:

1. Go to Docker for Mac
2. Download and run the installer.
3. Continue through installer with default options and enter your account credentials when requested.

Check here for more information on the installation.

1.11.2

Until version 1.11 the best way to run this Linux VM is to install Docker Toolbox, that installs Docker, VirtualBox and the Linux guest machine.

To install docker toolbox follow the following steps:

1. Go to Docker Toolbox
2. Click the link for Mac and run the installer.
3. Continue through installer with default options and enter your account credentials when requested.

This will install the Docker binaries in `/usr/local/bin` and update any existing Virtual Box installation. Check here for more information on the installation.

To Verify Installation:

1.12.0

1. Start `Docker.app` from the Applications folder, and make sure it is running. Next open up Terminal.

1.11.2

1. Open the `Docker Quickstart Terminal`, which will open a terminal and prepare it for use for Docker commands.

2. Once the terminal is open type

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

3. If all is well then this should print a welcome message verifying that the installation was successful.

Installing Docker on Windows

Requirements: 64-bit version of Windows 7 or higher on a machine which supports Hardware Virtualization Technology, and it is enabled.

While the docker binary can run natively on Windows, to build and host containers you need to run a Linux virtual machine on the box.

1.12.0

Since version 1.12 you don't need to have a separate VM to be installed, as Docker can use the native Hyper-V functionality of Windows to start up a small Linux machine to act as backend.

To install docker follow the following steps:

1. Go to Docker for Windows
2. Download and run the installer.
3. Continue through installer with default options and enter your account credentials when requested.

Check here for more information on the installation.

1.11.2

Until version 1.11 the best way to run this Linux VM is to install Docker Toolbox, that installs Docker, VirtualBox and the Linux guest machine.

To install docker toolbox follow the following steps:

1. Go to Docker Toolbox
2. Click the link for Windows and run the installer.
3. Continue through installer with default options and enter your account credentials when requested.

This will install the Docker binaries in Program Files and update any existing Virtual Box installation. Check here for more information on the installation.

To Verify Installation:

1.12.0

1. Start `Docker` from the Start menu if it hasn't been started yet, and make sure it is running. Next upen up any terminal (either `cmd` or PowerShell)

1.11.2

1. On your Desktop, find the Docker Toolbox icon. Click the icon to launch a Docker Toolbox terminal.

2. Once the terminal is open type

```
docker run hello-world
```

3. If all is well then this should print a welcome message verifying that the installation was successful.

Installing docker on Ubuntu Linux

Docker is supported on the following *64-bit* versions of Ubuntu Linux:

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

A couple of notes:

> The following instructions involve installation using **Docker** packages only, and this ensures obtaining the latest official release of **Docker**. If you need to install only using `Ubuntu-managed` packages, consult the Ubuntu documentation (Not recommended otherwise for obvious reasons).
>
> Ubuntu Utopic 14.10 and 15.04 exist in Docker's APT repository but are no longer officially supported due to known security issues.

Prerequisites

- Docker only works on a 64-bit installation of Linux.
- Docker requires Linux kernel version 3.10 or higher (Except for `Ubuntu Precise 12.04`, which requires version 3.13 or higher). Kernels older than 3.10 lack some of the features required to run Docker containers and contain known bugs which cause data loss and frequently panic under certain conditions. Check current kernel version with the command `uname -r`. Check this post if you need to update your `Ubuntu Precise (12.04 LTS)` kernel by scrolling further down. Refer to this WikiHow post to obtain the latest version for other Ubuntu installations.

Update APT sources

This needs to be done so as to access packages from Docker repository.

1. Log into your machine as a user with `sudo` or `root` privileges.
2. Open a terminal window.
3. Update package information, ensure that APT works with the https method, and that CA certificates are installed.

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

4. Add Docker's official GPG key:

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

   Verify that the key fingerprint is **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. Find the entry in the table below which corresponds to your Ubuntu version. This determines where APT will search for Docker packages. When possible, run a long-term support (LTS) edition of Ubuntu.

Ubuntu Version	Repository
Precise 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`

   Note: Docker does not provide packages for all architectures. Binary artifacts are built nightly, and you can download them from `https://master.dockerproject.org`. To install docker on a multi-architecture system, add an `[arch=...]` clause to the entry. Refer to Debian Multiarch wiki for details.

6. Run the following command, substituting the entry for your operating system for the placeholder `<REPO>`.

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

7. Update the `APT` package index by executing `sudo apt-get update`.

8. Verify that `APT` is pulling from the right repository.

When you run the following command, an entry is returned for each version of Docker that is

available for you to install. Each entry should have the URL `https://apt.dockerproject.org/repo/`. The version currently installed is marked with `***`.See the below example's output.

```
$ 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
```

From now on when you run `apt-get upgrade`, `APT` pulls from the new repository.

Prerequisites by Ubuntu Version

For Ubuntu Trusty (14.04) , Wily (15.10) , and Xenial (16.04) , install the `linux-image-extra-*` kernel packages, which allows you use the `aufs` storage driver.

To install the `linux-image-extra-*` packages:

1. Open a terminal on your Ubuntu host.
2. Update your package manager with the command `sudo apt-get update`.
3. Install the recommended packages.

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

4. Proceed to Docker installation

For Ubuntu Precise (12.04 LTS), Docker requires the 3.13 kernel version. If your kernel version is older than 3.13, you must upgrade it. Refer to this table to see which packages are required for your environment:

Package	Description
`linux-image-generic-lts-trusty`	Generic Linux kernel image. This kernel has `AUFS` built in. This is required to run Docker.
`linux-headers-generic-lts-trusty`	Allows packages such as `ZFS` and `VirtualBox guest additions` which depend on them. If you didn't install the headers for your existing kernel, then you can skip these headers for the `trusty` kernel. If you're unsure, you should include this package for safety.
`xserver-xorg-lts-trusty`	Optional in non-graphical environments without Unity/Xorg. **Required** when running Docker on machine with a graphical environment.

Package	Description
`ligbl1-mesa-glx-lts-trusty`	To learn more about the reasons for these packages, read the installation instructions for backported kernels, specifically the LTS Enablement Stack. Refer to note 5 under each version.

To upgrade your kernel and install the additional packages, do the following:

1. Open a terminal on your Ubuntu host.
2. Update your package manager with the command `sudo apt-get update`.
3. Install both the required and optional packages.

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

4. Repeat this step for other packages you need to install.
5. Reboot your host to use the updated kernel using the command `sudo reboot`.
6. After reboot, go ahead and install Docker.

Install the latest version

Make sure you satisfy the prerequisites, only then follow the below steps.

> **Note:** For production systems, it is recommended that you install a specific version so that you do not accidentally update Docker. You should plan upgrades for production systems carefully.

1. Log into your Ubuntu installation as a user with `sudo` privileges. (Possibly running `sudo -su`).
2. Update your APT package index by running `sudo apt-get update`.
3. Install Docker Community Edition with the command `sudo apt-get install docker-ce`.
4. Start the `docker` daemon with the command `sudo service docker start`.
5. Verify that `docker` is installed correctly by running the hello-world image.

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

This command downloads a test image and runs it in a container. When the container runs, it prints an informational message and exits.

Manage Docker as a non-root user

If you don't want to use `sudo` when you use the docker command, create a Unix group called `docker` and add users to it. When the `docker` daemon starts, it makes the ownership of the Unix socket read/writable by the docker group.

To create the `docker` group and add your user:

1. Log into Ubuntu as a user with `sudo` privileges.
2. Create the `docker` group with the command `sudo groupadd docker`.
3. Add your user to the `docker` group.

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

4. Log out and log back in so that your group membership is re-evaluated.
5. Verify that you can `docker` commands without `sudo` permission.

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

If this fails, you will see an error:

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

Check whether the `DOCKER_HOST` environment variable is set for your shell.

```
$ env | grep DOCKER_HOST
```

If it is set, the above command will return a result. If so, unset it.

```
$ unset DOCKER_HOST
```

You may need to edit your environment in files such as `~/.bashrc` or `~/.profile` to prevent the `DOCKER_HOST` variable from being set erroneously.

Installing Docker on Ubuntu

Requirements: Docker can be installed on any Linux with a kernel of at least version 3.10. Docker is supported on the following 64-bit versions of Ubuntu Linux:

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

Easy Installation

Note: Installing Docker from the default Ubuntu repository will install an old version of Docker.

To install the latest version of Docker using the Docker repository, use `curl` to grab and run the installation script provided by Docker:

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

Alternatively, `wget` can be used to install Docker:

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

Docker will now be installed.

Manual Installation

If, however, running the installation script is not an option, the following instructions can be used to manually install the latest version of Docker from the official repository.

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

Add the GPG key:

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

Next, open the `/etc/apt/sources.list.d/docker.list` file in your favorite editor. If the file doesn't exist, create it. Remove any existing entries. Then, depending on your version, add the following line:

- 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
  ```

Save the file and exit, then update your package index, uninstall any installed versions of Docker, and verify `apt` is pulling from the correct repo:

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

Depending on your version of Ubuntu, some prerequisites may be required:

- 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)

  This version of Ubuntu requires kernel version 3.13. You may need to install additional packages depending on your environment:

  `linux-image-generic-lts-trusty`

  Generic Linux kernel image. This kernel has AUFS built in. This is required to run Docker.

  `linux-headers-generic-lts-trusty`

  Allows packages such as ZFS and VirtualBox guest additions which depend on them. If you didn't install the headers for your existing kernel, then you can skip these headers for the `trusty` kernel. If you're unsure, you should include this package for safety.

  `xserver-xorg-lts-trusty`

  `libgl1-mesa-glx-lts-trusty`

  These two packages are optional in non-graphical environments without Unity/Xorg. Required when running Docker on machine with a graphical environment.

  To learn more about the reasons for these packages, read the installation instructions for backported kernels, specifically the LTS Enablement Stack — refer to note 5 under each version.

  Install the required packages then reboot the host:

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

  $ sudo reboot
  ```

Finally, update the `apt` package index and install Docker:

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

Start the daemon:

```
$ sudo service docker start
```

Now verify that docker is running properly by starting up a test image:

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

This command should print a welcome message verifying that the installation was successful.

Create a docker container in Google Cloud

You can use docker, without using docker daemon (engine), by using cloud providers. In this

example, you should have a `gcloud` (Google Cloud util), that connected to your account

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

This example will create a new instance, in your Google Cloud console. Using machine time `f1-large`

Install Docker on Ubuntu

Docker is supported on the following *64-bit* versions of Ubuntu Linux:

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

A couple of notes:

> The following instructions involve installation using **Docker** packages only, and this ensures obtaining the latest official release of **Docker**. If you need to install only using `Ubuntu-managed` packages, consult the Ubuntu documentation (Not recommended otherwise for obvious reasons).
>
> Ubuntu Utopic 14.10 and 15.04 exist in Docker's APT repository but are no longer officially supported due to known security issues.

Prerequisites

- Docker only works on a 64-bit installation of Linux.
- Docker requires Linux kernel version 3.10 or higher (Except for `Ubuntu Precise 12.04`, which requires version 3.13 or higher). Kernels older than 3.10 lack some of the features required to run Docker containers and contain known bugs which cause data loss and frequently panic under certain conditions. Check current kernel version with the command `uname -r`. Check this post if you need to update your `Ubuntu Precise (12.04 LTS)` kernel by scrolling further down. Refer to this WikiHow post to obtain the latest version for other Ubuntu installations.

Update APT sources

This needs to be done so as to access packages from Docker repository.

1. Log into your machine as a user with `sudo` or `root` privileges.
2. Open a terminal window.
3. Update package information, ensure that APT works with the https method, and that CA certificates are installed.

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

4. Add the new `GPG` key. This commands downloads the key with the ID `58118E89F3A912897C070ADBF76221572C52609D` from the keyserver `hkp://ha.pool.sks-keyservers.net:80` and adds it to the `adv keychain`. For more information, see the output of `man apt-key`.

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

5. Find the entry in the table below which corresponds to your Ubuntu version. This determines where APT will search for Docker packages. When possible, run a long-term support (LTS) edition of Ubuntu.

Ubuntu Version	Repository
Precise 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`

   Note: Docker does not provide packages for all architectures. Binary artifacts are built nightly, and you can download them from `https://master.dockerproject.org`. To install docker on a multi-architecture system, add an `[arch=...]` clause to the entry. Refer to Debian Multiarch wiki for details.

6. Run the following command, substituting the entry for your operating system for the placeholder `<REPO>`.

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

7. Update the `APT` package index by executing `sudo apt-get update`.

8. Verify that `APT` is pulling from the right repository.

When you run the following command, an entry is returned for each version of Docker that is available for you to install. Each entry should have the URL `https://apt.dockerproject.org/repo/`. The version currently installed is marked with `***`.See the below example's output.

```
$ 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
```

From now on when you run `apt-get upgrade`, `APT` pulls from the new repository.

Prerequisites by Ubuntu Version

For Ubuntu Trusty (14.04) , Wily (15.10) , and Xenial (16.04) , install the `linux-image-extra-*` kernel packages, which allows you use the `aufs` storage driver.

To install the `linux-image-extra-*` packages:

1. Open a terminal on your Ubuntu host.
2. Update your package manager with the command `sudo apt-get update`.
3. Install the recommended packages.

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

4. Proceed to Docker installation

For Ubuntu Precise (12.04 LTS), Docker requires the 3.13 kernel version. If your kernel version is older than 3.13, you must upgrade it. Refer to this table to see which packages are required for your environment:

Package	Description
`linux-image-generic-lts-trusty`	Generic Linux kernel image. This kernel has `AUFS` built in. This is required to run Docker.
`linux-headers-generic-lts-trusty`	Allows packages such as `ZFS` and `VirtualBox guest additions` which depend on them. If you didn't install the headers for your existing kernel, then you can skip these headers for the `trusty` kernel. If you're unsure, you should include this package for safety.
`xserver-xorg-lts-trusty`	Optional in non-graphical environments without Unity/Xorg. **Required** when running Docker on machine with a graphical environment.
`ligbl1-mesa-glx-lts-trusty`	To learn more about the reasons for these packages, read the installation instructions for backported kernels, specifically the LTS Enablement Stack. Refer to note 5 under each version.

To upgrade your kernel and install the additional packages, do the following:

1. Open a terminal on your Ubuntu host.
2. Update your package manager with the command `sudo apt-get update`.

3. Install both the required and optional packages.

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

4. Repeat this step for other packages you need to install.

5. Reboot your host to use the updated kernel using the command `sudo reboot`.

6. After reboot, go ahead and install Docker.

Install the latest version

Make sure you satisfy the prerequisites, only then follow the below steps.

> **Note:** For production systems, it is recommended that you install a specific version so that you do not accidentally update Docker. You should plan upgrades for production systems carefully.

1. Log into your Ubuntu installation as a user with `sudo` privileges. (Possibly running `sudo -su`).

2. Update your APT package index by running `sudo apt-get update`.

3. Install Docker with the command `sudo apt-get install docker-engine`.

4. Start the `docker` daemon with the command `sudo service docker start`.

5. Verify that `docker` is installed correctly by running the hello-world image.

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

This command downloads a test image and runs it in a container. When the container runs, it prints an informational message and exits.

Manage Docker as a non-root user

If you don't want to use `sudo` when you use the docker command, create a Unix group called `docker` and add users to it. When the `docker` daemon starts, it makes the ownership of the Unix socket read/writable by the docker group.

To create the `docker` group and add your user:

1. Log into Ubuntu as a user with `sudo` privileges.

2. Create the `docker` group with the command `sudo groupadd docker`.

3. Add your user to the `docker` group.

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

4. Log out and log back in so that your group membership is re-evaluated.

5. Verify that you can `docker` commands without `sudo` permission.

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

If this fails, you will see an error:

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

Check whether the `DOCKER_HOST` environment variable is set for your shell.

```
$ env | grep DOCKER_HOST
```

If it is set, the above command will return a result. If so, unset it.

```
$ unset DOCKER_HOST
```

You may need to edit your environment in files such as `~/.bashrc` or `~/.profile` to prevent the `DOCKER_HOST` variable from being set erroneously.

Installating Docker-ce OR Docker-ee on CentOS

Docker has announced following editions:

-Docker-ee (Enterprise Edition) along with Docker-ce(Community Edition) and Docker (Commercial Support)

This document will help you with installation steps of Docker-ee and Docker-ce edition in CentOS

Docker-ce Installation

Following are steps to install docker-ce edition

1. Install yum-utils, which provides yum-config-manager utility:

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

2. Use the following command to set up the stable repository:

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

3. Optional: Enable the edge repository. This repository is included in the docker.repo file above but is disabled by default. You can enable it alongside the stable repository.

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

- You can disable the edge repository by running the `yum-config-manager` command with the `--disable` flag. To re-enable it, use the `--enable` flag. The following command disables the edge repository.

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

4. Update the yum package index.

```
$ sudo yum makecache fast
```

5. Install the docker-ce using following command:

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

6. Confirm the Docker-ce fingerprint

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

   If you want to install some other version of docker-ce you can use following command:

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

   Specify the `VERSION` number

7. If everything went well the docker-ce is now installed in your system, use following command to start:

```
$ sudo systemctl start docker
```

8. Test your docker installation:

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

   you should get following message:

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

-Docker-ee (Enterprise Edition) Installation

For Enterprise Edition (EE) it would be required to signup, to get your <DOCKER-EE-URL>.

1. To signup go to https://cloud.docker.com/. Enter your details and confirm your email id. After confirmation you would be given a <DOCKER-EE-URL>, which you can see in your dashboard after clicking on setup.

2. Remove any existing Docker repositories from `/etc/yum.repos.d/`

3. Store your Docker EE repository URL in a yum variable in `/etc/yum/vars/`. Replace <DOCKER-EE-URL> with the URL you noted down in the first step.

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

4. Install yum-utils, which provides the yum-config-manager utility:

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

5. Use the following command to add the stable repository:

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

6. Update the yum package index.

```
$ sudo yum makecache fast
```

7. Install docker-ee

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

8. You can start the docker-ee using following command:

```
$ sudo systemctl start docker
```

Read Getting started with Docker online: https://riptutorial.com/docker/topic/658/getting-started-with-docker

Chapter 2: Building images

Parameters

Parameter	Details
--pull	Ensures that the base image (`FROM`) is up-to-date before building the rest of the Dockerfile.

Examples

Building an image from a Dockerfile

Once you have a Dockerfile, you can build an image from it using `docker build`. The basic form of this command is:

```
docker build -t image-name path
```

If your Dockerfile isn't named `Dockerfile`, you can use the `-f` flag to give the name of the Dockerfile to build.

```
docker build -t image-name -f Dockerfile2 .
```

For example, to build an image named `dockerbuild-example:1.0.0` from a `Dockerfile` in the current working directory:

```
$ ls
Dockerfile Dockerfile2

$ docker build -t dockerbuild-example:1.0.0 .

$ docker build -t dockerbuild-example-2:1.0.0 -f Dockerfile2 .
```

See the `docker build` usage documentation for more options and settings.

A common mistake is creating a Dockerfile in the user home directory (`~`). This is a bad idea because during `docker build -t mytag .` this message will appear for a long time:

> Uploading context

The cause is the docker daemon trying to copy all the user's files (both the home directory and it's subdirectories). Avoid this by always specifying a directory for the Dockerfile.

Adding a `.dockerignore` file to the build directory is a good practice. Its syntax is similar to `.gitignore` files and will make sure only wanted files and directories are uploaded as the context of the build.

A simple Dockerfile

```
FROM node:5
```

The `FROM` directive specifies an image to start from. Any valid image reference may be used.

```
WORKDIR /usr/src/app
```

The `WORKDIR` directive sets the current working directory inside the container, equivalent to running `cd` inside the container. (Note: `RUN cd` will *not* change the current working directory.)

```
RUN npm install cowsay knock-knock-jokes
```

`RUN` executes the given command inside the container.

```
COPY cowsay-knockknock.js ./
```

`COPY` copies the file or directory specified in the first argument from the build context (the *`path`* passed to `docker build` *`path`*) to the location in the container specified by the second argument.

```
CMD node cowsay-knockknock.js
```

`CMD` specifies a command to execute when the image is run and no command is given. It can be overridden by passing a command to `docker run`.

There are many other instructions and options; see the Dockerfile reference for a complete list.

Difference between ENTRYPOINT and CMD

There are two `Dockerfile` directives to specify what command to run by default in built images. If you only specify `CMD` then docker will run that command using the default `ENTRYPOINT`, which is `/bin/sh -c`. You can override either or both the entrypoint and/or the command when you start up the built image. If you specify both, then the `ENTRYPOINT` specifies the executable of your container process, and `CMD` will be supplied as the parameters of that executable.

For example if your `Dockerfile` contains

```
FROM ubuntu:16.04
CMD ["/bin/date"]
```

Then you are using the default `ENTRYPOINT` directive of `/bin/sh -c`, and running `/bin/date` with that default entrypoint. The command of your container process will be `/bin/sh -c /bin/date`. Once you run this image then it will by default print out the current date

```
$ docker build -t test .
$ docker run test
Tue Jul 19 10:37:43 UTC 2016
```

You can override `CMD` on the command line, in which case it will run the command you have specified.

```
$ docker run test /bin/hostname
bf0274ec8820
```

If you specify an `ENTRYPOINT` directive, Docker will use that executable, and the `CMD` directive specifies the default parameter(s) of the command. So if your `Dockerfile` contains:

```
FROM ubuntu:16.04
ENTRYPOINT ["/bin/echo"]
CMD ["Hello"]
```

Then running it will produce

```
$ docker build -t test .
$ docker run test
Hello
```

You can provide different parameters if you want to, but they will all run `/bin/echo`

```
$ docker run test Hi
Hi
```

If you want to override the entrypoint listed in your Dockerfile (i.e. if you wish to run a different command than `echo` in this container), then you need to specify the `--entrypoint` parameter on the command line:

```
$ docker run --entrypoint=/bin/hostname test
b2c70e74df18
```

Generally you use the `ENTRYPOINT` directive to point to your main application you want to run, and `CMD` to the default parameters.

Exposing a Port in the Dockerfile

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

From Docker's documentation:

> The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime. `EXPOSE` does not make the ports of the container accessible to the host. To do that, you must use either the `-p` flag to publish a range of ports or the `-P` flag to publish all of the exposed ports. You can expose one port number and publish it externally under another number.

Example:

Inside your Dockerfile:

```
EXPOSE 8765
```

To access this port from the host machine, include this argument in your `docker run` command:

```
-p 8765:8765
```

ENTRYPOINT and CMD seen as verb and parameter

Suppose you have a Dockerfile ending with

```
ENTRYPOINT [ "nethogs"] CMD ["wlan0"]
```

if you build this image with a

```
docker built -t inspector .
```

launch the image built with such a Dockerfile with a command such as

```
docker run -it --net=host --rm inspector
```

,nethogs will monitor the interface named wlan0

Now if you want to monitor the interface eth0 (or wlan1, or ra1...), you will do something like

```
docker run -it --net=host --rm inspector eth0
```

or

```
docker run -it --net=host --rm inspector wlan1
```

Pushing and Pulling an Image to Docker Hub or another Registry

Locally created images can be pushed to Docker Hub or any other docker repo host, known as a registry. Use `docker login` to sign in to an existing docker hub account.

```
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
```

A different docker registry can be used by specifying a server name. This also works for private or self-hosted registries. Further, using an external credentials store for safety is possible.

```
docker login quay.io
```

You can then tag and push images to the registry that you are logged in to. Your repository must

be specified as `server/username/reponame:tag`. Omitting the server currently defaults to Docker Hub. (The default registry cannot be changed to another provider, and there are no plans to implement this feature.)

```
docker tag mynginx quay.io/cjsimon/mynginx:latest
```

Different tags can be used to represent different versions, or branches, of the same image. An image with multiple different tags will display each tag in the same repo.

Use `docker images` to see a list of installed images installed on your local machine, including your newly tagged image. Then use push to upload it to the registry and pull to download the image.

```
docker push quay.io/cjsimon/mynginx:latest
```

All tags of an images can be pulled by specifying the `-a` option

```
docker pull quay.io/cjsimon/mynginx:latest
```

Building using a proxy

Often when building a Docker image, the Dockerfile contains instructions that runs programs to fetch resources from the Internet (`wget` for example to pull a program binary build on GitHub for example).

It is possible to instruct Docker to pass set set environment variables so that such programs perform those fetches through a 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` are environment variables which are available at build time only.

Read Building images online: https://riptutorial.com/docker/topic/713/building-images

Chapter 3: Checkpoint and Restore Containers

Examples

Compile docker with checkpoint and restore enabled (ubuntu)

In order to compile docker its recommended you have at least **2 GB RAM**. Even with that it fails sometimes so its better to go for **4GB** instead.

1. make sure git and make is installed

```
sudo apt-get install make git-core -y
```

2. install a new kernel (at least 4.2)

```
sudo apt-get install linux-generic-lts-xenial
```

3. reboot machine to have the new kernel active

```
sudo reboot
```

4. compile `criu` which is needed in order to run `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. check if every requirement is fulfilled to run criu

```
sudo criu check
```

6. compile experimental docker (we need docker to compile docker)

```
cd ~
wget -qO- https://get.docker.com/ | sh
sudo usermod -aG docker $(whoami)
```

- **At this point we have to logoff and login again to have a docker daemon. After relog continue with compile step**

```
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. We now have a compiled docker. Lets move the binaries. Make sure to replace `<version>` with the version installed

```
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
```

Dont worry - we backed up the old binaries. They are still there but with an underscore added to its names (`docker_`).

Congratulation you now have an experimental docker with the ability to checkpoint a container and restore it.

Please note that experimental features are NOT ready for production

Checkpoint and Restore a Container

```
# 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
```

Read Checkpoint and Restore Containers online:
https://riptutorial.com/docker/topic/5291/checkpoint-and-restore-containers

Chapter 4: Concept of Docker Volumes

Remarks

People new to Docker often don't realize that Docker filesystems are temporary by default. If you start up a Docker image you'll get a container that on the surface behaves much like a virtual machine. You can create, modify, and delete files. However, unlike a virtual machine, if you stop the container and start it up again, all your changes will be lost -- any files you previously deleted will now be back, and any new files or edits you made won't be present.

Volumes in docker containers allow for persistent data, and for sharing host-machine data inside a container.

Examples

A) Launch a container with a 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) Now press [cont +P+Q] to move out from container without terminating the container checking for container that is running

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

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES d87bf9607836 8251da35e7a7 "/bin/bash" About a minute ago Up 31 seconds vol3 [root@localhost ~]#

C) Run 'docker inspect' to check out more info about the volume

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

"Mounts": [{ "Name": "cdf78fbf79a7c9363948e133abe4c572734cd788c95d36edea0448094ec9121c", "Source": "/var/lib/docker/volumes/cdf78fbf79a7c9363948e133abe4c572734cd788c95d36edea0448094ec9121c/_d "Destination": "/data", "Driver": "local", "Mode": "", "RW": true

D) You can attach a running containers volume to another containers

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

```
root@ef2f5cc545be:/# ls
```

bin boot data dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var

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

E) You can also mount you base directory inside container

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

Here: /etc is host machine directory and /etc1 is the target inside container

Read Concept of Docker Volumes online: https://riptutorial.com/docker/topic/5908/concept-of-docker-volumes

Chapter 5: Connecting Containers

Parameters

Parameter	Details
`tty:true`	In docker-compose.yml, the `tty: true` flag keeps the container's `sh` command running waiting for input.

Remarks

The `host` and `bridge` network drivers are able to connect containers on a single docker host. To allow containers to communicate beyond one machine, create an overlay network. Steps to create the network depend on how your docker hosts are managed.

- Swarm Mode: `docker network create --driver overlay`
- docker/swarm: requires an external key-value store

Examples

Docker network

Containers in the same docker network have access to exposed ports.

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

Consul's Dockerfile exposes `8500`, `8600`, and several more ports. To demonstrate, run another container in the same network:

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

Here the consul container is resolved from `keys`, the name given in the first command. Docker provides dns resolution on this network, to find containers by their `--name`.

Docker-compose

Networks can be specified in a compose file (v2). By default all the containers are in a shared network.

Start with this file: `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
```

Starting this stack with `docker-compose up -d` will create a network named after the parent directory, in this case `example_default`. Check with `docker network ls`

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

Connect to the alpine container to verify the containers can resolve and communicate:

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

A compose file can have a `networks:` top level section to specify the network name, driver, and other options from the docker network command.

Container Linking

The docker `--link` argument, and `link:` sections docker-compose make *aliases* to other containers.

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

With link either the original name or the mapping will resolve the redis container.

```
> 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
```

Before docker `1.10.0` container linking also setup network connectivity - behavior now provided by docker network. Links in later versions only provide `legacy` effect on the default bridge network.

Read Connecting Containers online: https://riptutorial.com/docker/topic/6528/connecting-containers

Chapter 6: Creating a service with persistence

Syntax

- docker volume create --name <volume_name> # Creates a volume called <volume_name>
- docker run -v <volume_name>:<mount_point> -d crramirez/limesurvey:latest # Mount the <volume_name> volume in <mount_point> directory in the container

Parameters

Parameter	Details
--name <volume_name>	Specify the volume name to be created
-v <volume_name>:<mount_point>	Specify where the named volume will be mounted in the container

Remarks

Persistence is created in docker containers using volumes. Docker have many ways to deal with volumes. Named volumes are very convenient by:

- They persist even when the container is removed using the -v option.
- The only way to delete a named volume is doing an explicit call to docker volume rm
- The named volumes can be shared among container without linking or --volumes-from option.
- They don't have permission issues that host mounted volumes have.
- They can be manipulated using docker volume command.

Examples

Persistence with named volumes

Persistence is created in docker containers using volumes. Let's create a Limesurvey container and persist the database, uploaded content and configuration in a named volume:

```
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
```

Backup a named volume content

We need to create a container to mount the volume. Then archive it and download the archive to our host.

Let's create first a data volume with some data:

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

Let's backup the data:

```
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
```

Let's test:

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

Read Creating a service with persistence online: https://riptutorial.com/docker/topic/7429/creating-a-service-with-persistence

Chapter 7: Data Volumes and Data Containers

Examples

Data-Only Containers

Data-only containers are obsolete and are now considered an anti-pattern!

In the days of yore, before Docker's `volume` subcommand, and before it was possible to create named volumes, Docker deleted volumes when there were no more references to them in any containers. Data-only containers are obsolete because Docker now provides the ability to create named volumes, as well as much more utility via the various `docker volume` subcommand. Data-only containers are now considered an anti-pattern for this reason.

Many resources on the web from the last couple of years mention using a pattern called a "data-only container", which is simply a Docker container that exists only to keep a reference to a data volume around.

Remember that in this context, a "data volume" is a Docker volume which is not mounted from the host. To clarify, a "data volume" is a volume which is created either with the `VOLUME` Dockerfile directive, or using the `-v` switch on the command line in a `docker run` command, specifically with the format `-v /path/on/container`. Therefore a "data-only container" is a container whose only purpose is to have a data volume attached, which is used by the `--volumes-from` flag in a `docker run` command. For example:

```
docker run -d --name "mysql-data" -v "/var/lib/mysql" alpine /bin/true
```

When the above command is run, a "data-only container" is created. It is simply an empty container which has a data volume attached. It was then possible to use this volume in another container like so:

```
docker run -d --name="mysql" --volumes-from="mysql-data" mysql
```

The `mysql` container now has the same volume in it that is also in `mysql-data`.

Because Docker now provides the `volume` subcommand and named volumes, this pattern is now obsolete and not recommended.

To get started with the `volume` subcommand and named volumes see Creating a named volume

Creating a data volume

```
docker run -d --name "mysql-1" -v "/var/lib/mysql" mysql
```

This command creates a new container from the `mysql` image. It also creates a new data volume, which it then mounts in the container at `/var/lib/mysql`. This volume helps any data inside of it persist beyond the lifetime of the container. That is to say, when a container is removed, its filesystem changes are also removed. If a database was storing data in the container, and the container is removed, all of that data is also removed. Volumes will persist a particular location even beyond when its container is removed.

It is possible to use the same volume in multiple containers with the `--volumes-from` command line option:

```
docker run -d --name="mysql-2" --volumes-from="mysql-1" mysql
```

The `mysql-2` container now has the data volume from `mysql-1` attached to it, also using the path `/var/lib/mysql`.

Read Data Volumes and Data Containers online: https://riptutorial.com/docker/topic/3224/data-volumes-and-data-containers

Chapter 8: Debugging a container

Syntax

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

Examples

Entering in a running container

To execute operations in a container, use the `docker exec` command. Sometimes this is called "entering the container" as all commands are executed inside the container.

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

or

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

And now you have a shell in your running container. For example, list files in a directory and then leave the container:

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

You can use the `-u flag` to enter the container with a specific user, e.g. `uid=1013`, `gid=1023`.

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

The uid and gid does not have to exist in the container but the command can result in errors.If you want to launch a container and immediately enter inside in order to check something, you can do

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

the command `docker ps -lq` outputs only the id of the last (the l in `-lq`) container started. (this supposes you have bash as interpreter available in your container, you may have sh or zsh or any other)

Monitoring resource usage

Inspecting system resource usage is an efficient way to find misbehaving applications. This example is an equivalent of the traditional `top` command for containers:

```
docker stats
```

To follow the stats of specific containers, list them on the command line:

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

Docker stats displays the following information:

```
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
```

By default `docker stats` displays the id of the containers, and this is not very helpful, if your prefer to display the names of the container, just do

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

Monitoring processes in a container

Inspecting system resource usage is an efficient way to narrow down a problem on a live running application. This example is an equivalent of the traditional `ps` command for containers.

```
docker top 7786807d8084
```

To filter of format the output, add `ps` options on the command line:

```
docker top 7786807d8084 faux
```

Or, to get the list of processes running as root, which is a potentially harmful practice:

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

The `docker top` command proves especially useful when troubleshooting minimalistic containers without a shell or the `ps` command.

Attach to a running container

'Attaching to a container' is the act of starting a terminal session within the context that the container (and any programs therein) is running. This is primarily used for debugging purposes, but may also be needed if specific data needs to be passed to programs running within the container.

The `attach` command is utilized to do this. It has this syntax:

```
docker attach <container>
```

`<container>` can be either the container id or the container name. For instance:

```
docker attach c8a9cf1a1fa8
```

Or:

```
docker attach graceful_hopper
```

You may need to `sudo` the above commands, depending on your user and how docker is set up.

> Note: Attach only allows a single shell session to be attached to a container at a time.
>
> Warning: *all* keyboard input will be forwarded to the container. Hitting `Ctrl-c` will *kill* your container.

To detach from an attached container, successively hit `Ctrl-p` then `Ctrl-q`

To attach multiple shell sessions to a container, or simply as an alternative, you can use `exec`. Using the container id:

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

Using the container's name:

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

`exec` will run a program within a container, in this case `/bin/bash` (a shell, presumably one the container has). `-i` indicates an interactive session, while `-t` allocates a pseudo-TTY.

> Note: Unlike *attach*, hitting `Ctrl-c` will only terminate the *exec*'d command when running interactively.

Printing the logs

Following the logs is the less intrusive way to debug a live running application. This example reproduces the behavior of the traditional `tail -f some-application.log` on container `7786807d8084`.

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

This command basically shows the standard output of the container process (the process with pid 1).

If your logs do not natively include timestamping, you may add the `--timestamps` flag.

It is possible to look at the logs of a stopped container, either

- start the failing container with `docker run ... ; docker logs $(docker ps -lq)`
- find the container id or name with

`docker ps -a`

and then

`docker logs container-id` or

`docker logs containername`

as it is possible to look at the logs of a stopped container

Docker container process debugging

Docker is just a fancy way to run a process, not a virtual machine. Therefore, debugging a process "in a container" is also possible "on the host" by simply examining the running container process as a user with the appropriate permissions to inspect those processes on the host (e.g. root). For example, it's possible to list every "container process" on the host by running a simple `ps` as root:

```
sudo ps aux
```

Any currently running Docker containers will be listed in the output.

This can be useful during application development for debugging a process running in a container. As a user with appropriate permissions, typical debugging utilities can be used on the container process, such as strace, ltrace, gdb, etc.

Read Debugging a container online: https://riptutorial.com/docker/topic/1333/debugging-a-container

Chapter 9: Docker Data Volumes

Introduction

Docker data volumes provide a way to persist data independent of a container's life cycle. Volumes present a number of helpful features such as:

Mounting a host directory within the container, sharing data in-between containers using the filesystem and preserving data if a container gets deleted

Syntax

- docker volume [OPTIONS] [COMMAND]

Examples

Mounting a directory from the local host into a container

It is possible to mount a host directory to a specific path in your container using the `-v` or `--volume` command line option. The following example will mount `/etc` on the host to `/mnt/etc` in the container:

```
(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
```

The default access to the volume inside the container is read-write. To mount a volume read-only inside of a container, use the suffix `:ro`:

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

Creating a named volume

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

Using a named volume makes managing volumes much more human-readable. It is possible to create a named volume using the command specified above, but it's also possible to create a named volume inside of a `docker run` command using the `-v` or `--volume` command line option:

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

Note that creating a named volume in this form is similar to mounting a host file/directory as a volume, except that instead of a valid path, the volume name is specified. Once created, named volumes can be shared with other containers:

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

After running the above command, a new container has been created with the name `myApp-2` from the `myApp:1.5.3` image, which is sharing the `myAwesomeApp` named volume with `myApp-1`. The `myAwesomeApp` named volume is mounted at `/data/app` in the `myApp-2` container, just as it is mounted at `/data/app` in the `myApp-1` container.

Read Docker Data Volumes online: https://riptutorial.com/docker/topic/1318/docker-data-volumes

Chapter 10: Docker Engine API

Introduction

An API that allows you to control every aspect of Docker from within your own applications, build tools to manage and monitor applications running on Docker, and even use it to build apps on Docker itself.

Examples

Enable Remote access to Docker API on Linux

Edit `/etc/init/docker.conf` and update the `DOCKER_OPTS` variable to the following:

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

Restart Docker deamon

```
service docker restart
```

Verify if Remote API is working

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

Enable Remote access to Docker API on Linux running systemd

Linux running systemd, like Ubuntu 16.04, adding `-H tcp://0.0.0.0:2375` to `/etc/default/docker` does not have the effect it used to.

Instead, create a file called `/etc/systemd/system/docker-tcp.socket` to make docker available on a TCP socket on port 4243:

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

Then enable the new socket:

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

Now, verify if Remote API is working:

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

Enable Remote Access with TLS on Systemd

Copy the package installer unit file to /etc where changes will not be overwritten on an upgrade:

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

Update /etc/systemd/system/docker.service with your options on 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
```

Note that `dockerd` is the 1.12 daemon name, prior it was `docker daemon`. Also note that 2376 is dockers standard TLS port, 2375 is the standard unencrypted port. See this page for steps to create your own TLS self signed CA, cert, and key.

After making changes to the systemd unit files, run the following to reload the systemd config:

```
systemctl daemon-reload
```

And then run the following to restart docker:

```
systemctl restart docker
```

It's a bad idea to skip TLS encryption when exposing the Docker port since anyone with network access to this port effectively has full root access on the host.

Image pulling with progress bars, written in Go

Here is an example of image pulling using `Go` and `Docker Engine API` and the same progress bars as the ones shown when you run `docker pull your_image_name` in the `CLI`. For the purposes of the progress bars are used some ANSI codes.

```
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
}
```

For better readability, cursor actions with the ANSI codes are moved to a separate structure, which looks like this:

```
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")
}
```

After that in your main package you can call the `PullImage` function passing the image name you want to pull. Of course, before calling it, you have to be logged into the Docker registry, where the image is.

Making a cURL request with passing some complex structure

When using `cURL` for some queries to the `Docker API`, it might be a bit tricky to pass some complex structures. Let's say, getting a list of images allows using filters as a query parameter, which have to be a `JSON` representation of `map[string][]string` (about the maps in `Go` you can find more here). Here is how to achieve this:

```
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}}'
```

Here the `-G` flag is used to specify that the data in the `--data-urlencode` parameter will be used in an `HTTP GET` request instead of the `POST` request that otherwise would be used. The data will be appended to the URL with a `?` separator.

Read Docker Engine API online: https://riptutorial.com/docker/topic/3935/docker-engine-api

Chapter 11: Docker events

Examples

Launch a container and be notified of related events

The documentation for `docker events` provides details, but when debugging it may be useful to launch a container and be notified immediately of any related event:

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

In `docker ps -lq`, the `l` stands for `last`, and the `q` for `quiet`. This removes the `id` of the last container launched, and creates a notification immediately if the container dies or has another event occur.

Read Docker events online: https://riptutorial.com/docker/topic/6200/docker-events

Chapter 12: Docker in Docker

Examples

Jenkins CI Container using Docker

This chapter describes how to set up a Docker Container with Jenkins inside, which is capable of sending Docker commands to the Docker installation (the Docker Daemon) of the Host. Effectively using Docker in Docker. To achieve this, we have to build a custom Docker Image which is based on an arbitrary version of the official Jenkins Docker Image. The Dockerfile (The Instruction how to build the Image) looks like this:

```
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
```

This Dockerfile builds an Image which contains the Docker client binaries this client is used to communicate with a Docker Daemon. In this case the Docker Daemon of the Host. The `RUN` statement in this file also creates an UNIX usergroup with the UID 999 and adds the user Jenkins to it. Why exactly this is necessary is described in the further chapter. With this Image we can run a Jenkins server which can use Docker commands, but if we just run this Image the Docker client we installed inside the image cannot communicate with the Docker Daemon of the Host. These two components do communicate via a UNIX Socket `/var/run/docker.sock`. On Unix this is a file like everything else, so we can easily mount it inside the Jenkins Container. This is done with the command `docker run -v /var/run/docker.sock:/var/run/docker.sock --name jenkins MY_CUSTOM_IMAGE_NAME`. But this mounted file is owned by `docker:root`and because of this does the Dockerfile create this group with a well know UID and adds the Jenkins user to it. Now is the Jenkins Container really capable of running and using Docker. In production the run command should also contain `-v jenkins_home:/var/jenkins_home` to backup the Jenkins_home directory and of course a port-mapping to access the server over network.

Read Docker in Docker online: https://riptutorial.com/docker/topic/8012/docker-in-docker

Chapter 13: docker inspect getting various fields for key:value and elements of list

Examples

various docker inspect examples

I find that the examples in the `docker inspect` documentation seem magic, but do not explain much.

Docker inspect is important because it is the clean way to extract information from a running container `docker inspect -f ... container_id`

(or all running container)

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

avoiding some unreliable

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

When you launch a `docker inspect` you can get the values from the "top-level" easily, with a basic syntax like, for a container running htop (from https://hub.docker.com/r/jess/htop/) with a pid ae1

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

can show

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

or

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

can show

```
htop
```

Now if I extract a part of my `docker inspect`

I see

```
"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"
```

So I get a dictionary, as it has `{...}` and a lot of key:values

So the command

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

will return a list, such as

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

I can get the value of State.Pid easily

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

I get

```
4525
```

Sometimes docker inspect gives a list as it begins with `[` and ends with `]`

another example, with another container

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

gives

```
[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]
```

In order to get the first element of the list, we add index before the required field and 0 (as first element) after, so

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

gives

```
DISPLAY=:0
```

We get the next element with 1 instead of 0 using the same syntax

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

gives

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

We can get the number of elements of this list

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

gives

```
9
```

and we can get the last element of the list, the syntax is not easy

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

Read docker inspect getting various fields for key:value and elements of list online: https://riptutorial.com/docker/topic/6470/docker-inspect-getting-various-fields-for-key-value-and-

elements-of-list

Chapter 14: Docker Machine

Introduction

Remote management of multiple docker engine hosts.

Remarks

`docker-machine` manages remote hosts running Docker.

The `docker-machine` command line tool manages the full machine's life cycle using provider specific drivers. It can be used to select an "active" machine. Once selected, an active machine can be used as if it was the local Docker Engine.

Examples

Get current Docker Machine environment info

All these are shell commands.

`docker-machine env` to get the current default docker-machine configuration

`eval $(docker-machine env)` to get the current docker-machine configuration and set the current shell environment up to use this docker-machine with .

If your shell is set up to use a proxy, you can specify the --no-proxy option in order to bypass the proxy when connecting to your docker-machine: `eval $(docker-machine env --no-proxy)`

If you have multiple docker-machines, you can specify the machine name as argument: `eval $(docker-machine env --no-proxy machinename)`

SSH into a docker machine

All these are shell commands

- If you need to log onto a running docker-machine directly, you can do that:

`docker-machine ssh` to ssh into the default docker-machine

`docker-machine ssh machinename` to ssh into a non-default docker-machine

- If you just want to run a single command, you can do so. To run `uptime` on the default docker-machine to see how long it's been running for, run `docker-machine ssh default uptime`

Create a Docker machine

Using `docker-machine` is the best method to install Docker on a machine. It will automatically apply the best security settings available, including generating a unique pair of SSL certificates for mutual authentication and SSH keys.

To create a local machine using Virtualbox:

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

To install Docker on an existing machine, use the `generic` driver:

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

The `--driver` option tells docker how to create the machine. For a list of supported drivers, see:

- officially supported
- third party

List docker machines

Listing docker-machines will return the state, address and version of Docker of each docker machines.

```
 docker-machine ls
```

Will print something like:

```
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
```

To list running machines:

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

To list error machines:

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

To list machines who's name starts with 'side-project-', use Golang filter:

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

To get only the list of machine's URLs:

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

See https://docs.docker.com/machine/reference/ls/ for the full command reference.

Upgrade a Docker Machine

Upgrading a docker machine implies a downtime and may require planing. To upgrade a docker machine, run:

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

This command does not have options

Get the IP address of a docker machine

To get the IP address of a docker machine, you can do that with this command :

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

Read Docker Machine online: https://riptutorial.com/docker/topic/1349/docker-machine

Chapter 15: Docker --net modes (bridge, hots, mapped container and none).

Introduction

Getting Started

Bridge Mode It's a default and attached to docker0 bridge. Put container on a completely separate network namespace.

Host Mode When container is just a process running in a host, we'll attach the container to the host NIC.

Mapped Container Mode This mode essentially maps a new container into an existing containers network stack. It's also called 'container in container mode'.

None It tells docker put the container in its own network stack without configuration

Examples

Bridge Mode, Host Mode and Mapped Container Mode

Bridge Mode

```
$ docker run -d --name my_app -p 10000:80 image_name
```

Note that we did not have to specify **--net=bridge** because this is the default working mode for docker. This allows to run multiple containers to run on same host without any assignment of dynamic port. So **BRIDGE** mode avoids the port clashing and it's safe as each container is running its own private network namespace.

Host Mode

```
$ docker run -d --name my_app -net=host image_name
```

As it uses the host network namespace, no need of special configuraion but may leads to security issue.

Mapped Container Mode

This mode essentially maps a new container into an existing containers network stack. This implies that network resources such as IP address and port mappings of the first container will be shared by the second container. This is also called as 'container in container' mode. Suppose you have two contaienrs as web_container_1 and web_container_2 and we'll run web_container_2 in mapped container mode. Let's first download web_container_1 and runs it into detached mode

with following command,

```
$ docker run -d --name web1 -p 80:80 USERNAME/web_container_1
```

Once it’s downloaded let’s take a look and make sure its running. Here we just mapped a port into a container that's running in the default bridge mode. Now, let’s run a second container in mapped container mode. We’ll do that with this command.

```
$ docker run -d --name web2 --net=container:web1 USERNAME/web_container_2
```

Now, if you simply get the interface information on both the contaienrs, you will get the same network config. This actually include the HOST mode that maps with exact info of the host. The first contaienr ran in default bridge mode and second container is running in mapped container mode. We can obtain very similar results by starting the first container in host mode and the second container in mapped container mode.

Read Docker --net modes (bridge, hots, mapped container and none). online: https://riptutorial.com/docker/topic/9643/docker---net-modes--bridge--hots--mapped-container-and-none--

Chapter 16: Docker network

Examples

How to find the Container's host ip

You need to find out the IP address of the container running in the host so you can, for example, connect to the web server running in it.

`docker-machine` is what is used on MacOSX and Windows.

Firstly, list your machines:

```
$ docker-machine ls

NAME      ACTIVE   DRIVER       STATE     URL                         SWARM
default   *        virtualbox   Running   tcp://192.168.99.100:2376
```

Then select one of the machines (the default one is called default) and:

```
$ docker-machine ip default

192.168.99.100
```

Creating a Docker network

```
docker network create app-backend
```

This command will create a simple bridged network called `appBackend`. No containers are attached to this network by default.

Listing Networks

```
docker network ls
```

This command lists all networks that have been created on the local Docker host. It includes the default bridge `bridge` network, the host `host` network, and the null `null` network. All containers by default are attached to the default bridge `bridge` network.

Add container to network

```
docker network connect app-backend myAwesomeApp-1
```

This command attaches the `myAwesomeApp-1` container to the `app-backend` network. When you add a container to a user-defined network, the embedded DNS resolver (which is not a full-featured DNS

server, and is not exportable) allows each container on the network to resolve each other container on the same network. This simple DNS resolver is not available on the default bridge `bridge` network.

Detach container from network

```
docker network disconnect app-backend myAwesomeApp-1
```

This command detaches the `myAwesomeApp-1` container from the `app-backend` network. The container will no longer be able to communicate with other containers on the network it has been disconnected from, nor use the embedded DNS resolver to look up other containers on the network it has been detached from.

Remove a Docker network

```
docker network rm app-backend
```

This command removes the user-defined `app-backend` network from the Docker host. All containers on the network not otherwise connected via another network will lose communication with other containers. It is not possible to remove the default bridge `bridge` network, the `host` host network, or the `null` null network.

Inspect a Docker network

```
docker network inspect app-backend
```

This command will output details about the `app-backend` network.

The of the output of this command should look similar to:

```
[
    {
        "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": {}
```

```
        }
]
```

Read Docker network online: https://riptutorial.com/docker/topic/3221/docker-network

Chapter 17: Docker private/secure registry with API v2

Introduction

A private and secure docker registry instead of a Docker Hub. Basic docker skills are required.

Parameters

Command	Explanation
sudo docker run -p 5000:5000	Start a docker container and bind the port 5000 from container to the port 5000 of the physical machine.
--name registry	Container name (use to make “docker ps” readability better).
-v 'pwd'/certs:/certs	Bind CURRENT_DIR/certs of the physical machine on /certs of the container (like a “shared folder”).
-e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/server.crt	We specify that the registry should use /certs/server.crt file to start. (env variable)
-e REGISTRY_HTTP_TLS_KEY=/certs/server.key	Same for the RSA key (server.key).
-v /root/images:/var/lib/registry/	If you want to save all your registry images you should do this on the physical machine. Here we save all images on /root/images on the physical machine. If you do this then you can stop and restart the registry without losing any images.
registry:2	We specify that we would like to pull the registry image from docker hub (or locally), and we add « 2 » because we want install the version 2 of registry.

Remarks

How to install a docker-engine (called client on this tutorial)

How to generate SSL self-signed certificate

Examples

Generating certificates

Generate a RSA private key: `openssl genrsa -des3 -out server.key 4096`

Openssl should ask for a pass phrase at this step. Notice that we'll use only certificate for communication and authentication, without pass phrase. Just use 123456 for example.

Generate the Certificate Signing Request: `openssl req -new -key server.key -out server.csr`

This step is important because you'll be asked for some information about certificates. The most important information is "Common Name" that is the domain name, which be used for communication between private docker registry and all other machine. Example : mydomain.com

Remove pass phrase from RSA private key: `cp server.key server.key.org && openssl rsa -in server.key.org -out server.key`

Like I said we'll focus on certificate without pass phrase. So be careful with all your key's files (.key,.csr,.crt) and keep them on a secure place.

Generate the self-signed certificate: `openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt`

You have now two essential files, *server.key* and *server.crt*, that are necessary for the private registry authentication.

Run the registry with self-signed certificate

To run the private registry (securely) you have to generate a self-signed certificate, you can refer to previous example to generate it.

For my example I put *server.key* and *server.crt* into /root/certs

Before run docker command you should be placed (use `cd`) into the directory that contains *certs* folder. If you're not and you try to run the command you'll receive an error like

> level=fatal msg="open /certs/server.crt: no such file or directory"

When you are (`cd /root` in my example), you can basically start the secure/private registry using :
`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`
Explanations about the command is available on Parameters part.

Pull or push from a docker client

When you get a working registry running you can pull or push images on it. For that you need the *server.crt* file into a special folder on your docker client. The certificate allows you to authenticate with the registry, and then encrypt communication.

Copy *server.crt* from registry machine into /etc/docker/certs.d/mydomain.com:5000/ on your client machine. And then rename it to *ca-certificates.crt* : `mv /etc/docker/certs.d/mydomain.com:5000/server.crt /etc/docker/certs.d/mydomain.com:5000/ca-certificates.crt`

At this point you can pull or push images from your private registry :
PULL : `docker pull mydomain.com:5000/nginx` or
PUSH :

1. Get an official image from hub.docker.com : `docker pull nginx`
2. Tag this image before pushing into private registry : `docker tag IMAGE_ID mydomain.com:5000/nginx` (use `docker images` to get the IMAGE_ID)
3. Push the image to the registry : `docker push mydomain.com:5000/nginx`

Read Docker private/secure registry with API v2 online:
https://riptutorial.com/docker/topic/8707/docker-private-secure-registry-with-api-v2

Chapter 18: Docker Registry

Examples

Running the registry

Do not use `registry:latest`! This image points to the old v1 registry. That Python project is no longer being developed. The new v2 registry is written in Go and is actively maintained. When people refer to a "private registry" they are referring to the v2 registry, *not* the v1 registry!

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

The above command runs the newest version of the registry, which can be found in the Docker Distribution project.

For more examples of image management features, such as tagging, pulling, or pushing, see the section on managing images.

Configure the registry with AWS S3 storage backend

Configuring a private registry to use an AWS S3 backend is easy. The registry can do this automatically with the right configuration. Here is an example of what should be in your `config.yml` file:

```
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
```

The `accesskey` and `secretkey` fields are IAM credentials with specific S3 permissions (see the documentation for more information). It can just as easily use credentials with the `AmazonS3FullAccess` policy attached. The `region` is the region of your S3 bucket. The `bucket` is the bucket name. You may elect to store your images encrypted with `encrypt`. The `secure` field is to indicate the use of HTTPS. You should generally set `v4auth` to true, even though its default value is false. The `chunksize` field allows you to abide by the S3 API requirement that chunked uploads are at least five megabytes in size. Finally, `rootdirectory` specifies a directory underneath your S3 bucket to use.

There are other storage backends that can be configured just as easily.

Read Docker Registry online: https://riptutorial.com/docker/topic/4173/docker-registry

Chapter 19: Docker stats all running containers

Examples

Docker stats all running containers

```
sudo docker stats $(sudo docker inspect -f "{{ .Name }}" $(sudo docker ps -q))
```

Shows live CPU usage of all running containers.

Read Docker stats all running containers online: https://riptutorial.com/docker/topic/5863/docker-stats-all-running-containers

Chapter 20: Docker swarm mode

Introduction

A swarm is a number of Docker Engines (or *nodes*) that deploy *services* collectively. Swarm is used to distribute processing across many physical, virtual or cloud machines.

Syntax

- Initialize a swarm: docker swarm init [OPTIONS]
- Join a swarm as a node and/or manager: docker swarm join [OPTIONS] HOST:PORT
- Create a new service: docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]
- Display detailed information on one or more services: docker service inspect [OPTIONS] SERVICE [SERVICE...]
- List services: docker service ls [OPTIONS]
- Remove one or more services: docker service rm SERVICE [SERVICE...]
- Scale one or multiple replicated services: docker service scale SERVICE=REPLICAS [SERVICE=REPLICAS...]
- List the tasks of one or more services: docker service ps [OPTIONS] SERVICE [SERVICE...]
- Update a service: docker service update [OPTIONS] SERVICE

Remarks

Swarm mode implements the following features:

- Cluster management integrated with Docker Engine
- Decentralized design
- Declarative service model
- Scaling
- Desired state reconciliation
- Multi-host networking
- Service discovery
- Load balancing
- Secure design by default
- Rolling updates

For more official Docker documentation regarding Swarm visit: Swarm mode overview

Swarm Mode CLI Commands

Click on commands description for documentation

Initialize a swarm

```
docker swarm init [OPTIONS]
```

Join a swarm as a node and/or manager

```
docker swarm join [OPTIONS] HOST:PORT
```

Create a new service

```
docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Display detailed information on one or more services

```
docker service inspect [OPTIONS] SERVICE [SERVICE...]
```

List services

```
docker service ls [OPTIONS]
```

Remove one or more services

```
docker service rm SERVICE [SERVICE...]
```

Scale one or multiple replicated services

```
docker service scale SERVICE=REPLICAS [SERVICE=REPLICAS...]
```

List the tasks of one or more services

```
docker service ps [OPTIONS] SERVICE [SERVICE...]
```

Update a service

```
docker service update [OPTIONS] SERVICE
```

Examples

Create a swarm on Linux using docker-machine and 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
```

Find out worker and manager join token

When automating the provisioning of new nodes to a swarm, you need to know what the right join token is for the swarm as well as the advertised address of the manager. You can find this out by running the following commands on any of the existing manager nodes:

```
# 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)
```

The -q option outputs only the token. Without this option you get the full command for registering to a swarm.

Then on newly provisioned nodes, you can join the swarm using.

```
docker swarm join --token $WORKER_TOKEN $MANAGER_ADDRESS
```

Hello world application

Usually you'd want to create a stack of services to form a replicated and orchestrated application.

A typical modern web application consists of a database, api, frontend and reverse proxy.

Persistence

Database needs persistence, so we need some filesystem which is shared across all the nodes in a swarm. It can be NAS, NFS server, GFS2 or anything else. Setting it up is out of scope here. Currently Docker doesn't contain and doesn't manage persistence in a swarm. This example assumes that there's `/nfs/` shared location mounted across all nodes.

Network

To be able to communicate with each other, services in a swarm need to be on the same network.

Choose an IP range (here `10.0.9.0/24`) and network name (`hello-network`) and run a command:

```
docker network create \
  --driver overlay \
  --subnet 10.0.9.0/24 \
  --opt encrypted \
  hello-network
```

Database

The first service we need is a database. Let's use postgresql as an example. Create a folder for a database in `nfs/postgres` and run this:

```
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
```

Notice that we've used `--network hello-network` and `--mount` options.

API

Creating API is out of scope of this example, so let's pretend you have an API image under `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
```

Notice that we passed a name of our database service. Docker swarm has an embedded round-robin DNS server, so API will be able to connect to database by using its DNS name.

Reverse proxy

Let's create nginx service to serve our API to an outer world. Create nginx config files in a shared location and run this:

```
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
```

Notice that we've used `-p` option to publish a port. This port would be available to any node in a swarm.

Node Availablility

Swarm Mode Node Availability:

- Active means that the scheduler can assign tasks to a node.
- Pause means the scheduler doesn't assign new tasks to the node, but existing tasks remain running.
- Drain means the scheduler doesn't assign new tasks to the node. The scheduler shuts down any existing tasks and schedules them on an available node.

To change Mode Availability:

```
#Following commands can be used on swarm manager(s)
docker node update --availability drain node-1
#to verify:
docker node ls
```

Promote or Demote Swarm Nodes

To promote a node or set of nodes, run `docker node promote` from a manager node:

```
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.
```

To demote a node or set of nodes, run `docker node demote` from a manager node:

```
docker node demote node-3 node-2

Manager node-3 demoted in the swarm.
Manager node-2 demoted in the swarm.
```

Leaving the Swarm

Worker Node:

```
#Run the following on the worker node to leave the swarm.

docker swarm leave
Node left the swarm.
```

If the node has the *Manager* role, you will get a warning about maintaining the quorum of

Managers. You can use --force to leave on the manager node:

```
#Manager Node

docker swarm leave --force
Node left the swarm.
```

Nodes that left the Swarm will still show up in `docker node ls` output.

To remove nodes from the list:

```
docker node rm node-2

node-2
```

Read Docker swarm mode online: https://riptutorial.com/docker/topic/749/docker-swarm-mode

Chapter 21: Dockerfile contents ordering

Remarks

1. Base image declaration (`FROM`)
2. Metadata (e.g. `MAINTAINER`, `LABEL`)
3. Installing system dependencies (e.g. `apt-get install`, `apk add`)
4. Copying app dependencies file (e.g. `bower.json`, `package.json`, `build.gradle`, `requirements.txt`)
5. Installing app dependencies (e.g. `npm install`, `pip install`)
6. Copying entire code base
7. Setting up default runtime configs (e.g. `CMD`, `ENTRYPOINT`, `ENV`, `EXPOSE`)

These orderings are made for optimizing build time using Docker's built-in caching mechanism.

Rule of thumbs:

> Parts that change often (e.g. codebase) should be placed near bottom of Dockerfile, and vice-versa. Parts that rarely change (e.g. dependencies) should be placed at top.

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 will be deprecated in Docker 1.13, and should be replaced by using LABEL. (Source)

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

Read Dockerfile contents ordering online: https://riptutorial.com/docker/topic/6448/dockerfile-contents-ordering

Chapter 22: Dockerfiles

Introduction

Dockerfiles are files used to programatically build Docker images. They allow you to quickly and reproducibly create a Docker image, and so are useful for collaborating. Dockerfiles contain instructions for building a Docker image. Each instruction is written on one row, and is given in the form `<INSTRUCTION><argument(s)>`. Dockerfiles are used to build Docker images using the `docker build` command.

Remarks

Dockerfiles are of the form:

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

- Comments starts with a `#`
- Instructions are upper case only
- The first instruction of a Dockerfile must be `FROM` to specify the base image

---

When building a Dockerfile, the Docker client will send a "build context" to the Docker daemon. The build context includes all files and folder in the same directory as the Dockerfile. `COPY` and `ADD` operations can only use files from this context.

Some Docker file may start with:

```
# escape=`
```

This is used to instruct the Docker parser to use `` ` `` as an escape character instead of `\`. This is mostly useful for Windows Docker files.

Examples

HelloWorld Dockerfile

A minimal Dockerfile looks like this:

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

This will instruct Docker to build an image based on Alpine (`FROM`), a minimal distribution for containers, and to run a specific command (`CMD`) when executing the resulting image.

Build and run it:

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

This will output:

```
Hello StackOverflow!
```

Copying files

To copy files from the build context in a Docker image, use the `COPY` instruction:

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

If the filename contains spaces, use the alternate syntax:

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

The `COPY` command supports wildcards. It can be used for example to copy all images to the `images/` directory:

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

Note: in this example, `images/` may not exist. In this case, Docker will create it automatically.

Exposing a port

To declare exposed ports from a Dockerfile use the `EXPOSE` instruction:

```
EXPOSE 8080 8082
```

Exposed ports setting can be overridden from the Docker commandline but it is a good practice to explicitly set them in the Dockerfile as it helps understand what an application does.

Dockerfiles best pratices

Group common operations

Docker builds images as a collection of layers. Each layer can only add data, even if this data says that a file has been deleted. Every instruction creates a new layer. For example:

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

Has a couple of downsides:

- It will create two layers, producing a larger image.
- Using `apt-get update` alone in a `RUN` statement causes caching issues and subsequently `apt-get install` instructions may **fail**. Suppose you later modify `apt-get install` by adding extra packages, then docker interprets the initial and modified instructions as identical and reuses the cache from previous steps. As a result the `apt-get update` command is **not** executed because its cached version is used during the build.

Instead, use:

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

as this only produce one layer.

Mention the maintainer

This is usually the second line of the Dockerfile. It tells who is in charge and will be able to help.

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

If you skip it, it will not break your image. But it will not help your users either.

Be concise

Keep your Dockerfile short. If a complex setup is necessary, consider using a dedicated script or setting up base images.

USER Instruction

```
USER daemon
```

The `USER` instruction sets the user name or UID to use when running the image and for any `RUN`, `CMD` and `ENTRYPOINT` instructions that follow it in the `Dockerfile`.

WORKDIR Instruction

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

The `WORKDIR` instruction sets the working directory for any `RUN`, `CMD`, `ENTRYPOINT`, `COPY` and `ADD` instructions that follow it in the Dockerfile. If the `WORKDIR` doesn't exist, it will be created even if it's not used in any subsequent `Dockerfile` instruction.

It can be used multiple times in the one `Dockerfile`. If a relative path is provided, it will be relative to the path of the previous `WORKDIR` instruction. For example:

```
WORKDIR /a
WORKDIR b
WORKDIR c
```

```
RUN pwd
```

The output of the final `pwd` command in this `Dockerfile` would be `/a/b/c`.

The `WORKDIR` instruction can resolve environment variables previously set using `ENV`. You can only use environment variables explicitly set in the `Dockerfile`. For example:

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

The output of the final `pwd` command in this Dockerfile would be `/path/$DIRNAME`

VOLUME Instruction

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

The `VOLUME` instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers. The value can be a JSON array, `VOLUME ["/var/log/"]`, or a plain string with multiple arguments, such as `VOLUME /var/log` or `VOLUME /var/log /var/db`. For more information/examples and mounting instructions via the Docker client, refer to Share Directories via Volumes documentation.

The `docker run` command initializes the newly created volume with any data that exists at the specified location within the base image. For example, consider the following Dockerfile snippet:

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

This Dockerfile results in an image that causes docker run, to create a new mount point at /myvol and copy the greeting file into the newly created volume.

Note: If any build steps change the data within the volume after it has been declared, those changes will be discarded.

Note: The list is parsed as a JSON array, which means that you must use double-quotes (“) around words not single-quotes (‘).

COPY Instruction

`COPY` has two forms:

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

The `COPY` instruction copies new files or directories from `<src>` and adds them to the filesystem of the container at the path `<dest>`.

Multiple `<src>` resource may be specified but they must be relative to the source directory that is being built (the context of the build).

Each `<src>` may contain wildcards and matching will be done using Go's `filepath.Match` rules. For example:

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

The `<dest>` is an absolute path, or a path relative to `WORKDIR`, into which the source will be copied inside the destination container.

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

All new files and directories are created with a UID and GID of 0.

Note: If you build using stdin (`docker build - < somefile`), there is no build context, so `COPY` can't be used.

`COPY` obeys the following rules:

- The `<src>` path must be inside the context of the build; you cannot `COPY` ../something /something, because the first step of a docker build is to send the context directory (and subdirectories) to the docker daemon.
- If `<src>` is a directory, the entire contents of the directory are copied, including filesystem metadata. Note: The directory itself is not copied, just its contents.
- If `<src>` is any other kind of file, it is copied individually along with its metadata. In this case, if `<dest>` ends with a trailing slash /, it will be considered a directory and the contents of `<src>` will be written at `<dest>/base(<src>)`.
- If multiple `<src>` resources are specified, either directly or due to the use of a wildcard, then `<dest>` must be a directory, and it must end with a slash `/`.
- If `<dest>` does not end with a trailing slash, it will be considered a regular file and the contents of `<src>` will be written at `<dest>`.
- If `<dest>` doesn't exist, it is created along with all missing directories in its path.

The ENV and ARG Instruction

ENV

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

The `ENV` instruction sets the environment variable `<key>` to the value . This value will be in the

environment of all “descendant” Dockerfile commands and can be replaced inline in many as well.

The `ENV` instruction has two forms. The first form, `ENV <key> <value>`, will set a single variable to a value. The entire string after the first space will be treated as the `<value>` - including characters such as spaces and quotes.

The second form, `ENV <key>=<value> ...`, allows for multiple variables to be set at one time. Notice that the second form uses the equals sign (=) in the syntax, while the first form does not. Like command line parsing, quotes and backslashes can be used to include spaces within values.

For example:

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

and

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

will yield the same net results in the final container, but the first form is preferred because it produces a single cache layer.

The environment variables set using `ENV` will persist when a container is run from the resulting image. You can view the values using docker inspect, and change them using `docker run --env <key>=<value>`.

ARG

If you don't wish to persist the setting, use `ARG` instead. `ARG` will set environments only during the build. For example, setting

```
ENV DEBIAN_FRONTEND noninteractive
```

may confuse `apt-get` users on a Debian-based image when they enter the container in an interactive context via `docker exec -it the-container bash`.

Instead, use:

```
ARG DEBIAN_FRONTEND noninteractive
```

You might alternativly also set a value for a single command only by using:

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

EXPOSE Instruction

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

The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime. `EXPOSE` does NOT make the ports of the container accessible to the host. To do that, you must use either the `-p` flag to publish a range of ports or the `-P` flag to publish all of the exposed ports. These flags are used in the `docker run [OPTIONS] IMAGE [COMMAND][ARG...]` to expose the port to the host. You can expose one port number and publish it externally under another number.

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

This command will create a container with the name <image> and bind the container's port 80 to the host machine's port 2500.

To set up port redirection on the host system, see using the `-P` flag. The Docker network feature supports creating networks without the need to expose ports within the network, for detailed information see the overview of this feature).

LABEL Instruction

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

The `LABEL` instruction adds metadata to an image. A `LABEL` is a key-value pair. To include spaces within a `LABEL` value, use quotes and backslashes as you would in command-line parsing. A few usage examples:

```
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."
```

An image can have more than one label. To specify multiple labels, Docker recommends combining labels into a single `LABEL` instruction where possible. Each `LABEL` instruction produces a new layer which can result in an inefficient image if you use many labels. This example results in a single image layer.

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

The above can also be written as:

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

Labels are additive including `LABEL`s in `FROM` images. If Docker encounters a label/key that already exists, the new value overrides any previous labels with identical keys.

To view an image's labels, use the docker inspect command.

```
"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"
},
```

CMD Instruction

The `CMD` instruction has three forms:

```
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)
```

There can only be one `CMD` instruction in a `Dockerfile`. If you list more than one `CMD` then only the last `CMD` will take effect.

The main purpose of a `CMD` is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an `ENTRYPOINT` instruction as well.

Note: If `CMD` is used to provide default arguments for the `ENTRYPOINT` instruction, both the `CMD` and `ENTRYPOINT` instructions should be specified with the JSON array format.

Note: The exec form is parsed as a JSON array, which means that you must use double-quotes (“) around words not single-quotes (‘).

Note: Unlike the shell form, the exec form does not invoke a command shell. This means that normal shell processing does not happen. For example, `CMD [ "echo", "$HOME" ]` will not do variable substitution on `$HOME`. If you want shell processing then either use the shell form or execute a shell directly, for example: `CMD [ "sh", "-c", "echo $HOME" ]`.

When used in the shell or exec formats, the `CMD` instruction sets the command to be executed when running the image.

If you use the shell form of the `CMD`, then the command will execute in `/bin/sh -c`:

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

If you want to run your command without a shell then you must express the command as a JSON array and give the full path to the executable. This array form is the preferred format of `CMD`. Any additional parameters must be individually expressed as strings in the array:

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

If you would like your container to run the same executable every time, then you should consider using `ENTRYPOINT` in combination with `CMD`. See `ENTRYPOINT`.

If the user specifies arguments to docker run then they will override the default specified in `CMD`.

Note: don't confuse `RUN` with `CMD`. `RUN` actually runs a command at image building time and commits the result; `CMD` does not execute anything at build time, but specifies the intended command for the image.

MAINTAINER Instruction

```
MAINTAINER <name>
```

The `MAINTAINER` instruction allows you to set the Author field of the generated images.

DO NOT USE THE MAINTAINER DIRECTIVE

According to Official Docker Documentation the `MAINTAINER` instruction is deprecated. Instead, one should use the `LABEL` instruction to define the author of the generated images. The `LABEL` instruction is more flexible, enables setting metadata, and can be easily viewed with the command `docker inspect`.

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

FROM Instruction

```
FROM <image>
```

Or

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

Or

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

The `FROM` instruction sets the Base Image for subsequent instructions. As such, a valid Dockerfile must have `FROM` as its first instruction. The image can be any valid image – it is especially easy to start by pulling an image from the Public Repositories.

`FROM` must be the first non-comment instruction in the Dockerfile.

`FROM` can appear multiple times within a single Dockerfile in order to create multiple images. Simply make a note of the last image ID output by the commit before each new `FROM` command.

The tag or digest values are optional. If you omit either of them, the builder assumes a latest by default. The builder returns an error if it cannot match the tag value.

RUN Instruction

`RUN` has 2 forms:

```
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)
```

The `RUN` instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the `Dockerfile`.

Layering `RUN` instructions and generating commits conforms to the core concepts of Docker where commits are cheap and containers can be created from any point in an image's history, much like source control.

The exec form makes it possible to avoid shell string munging, and to `RUN` commands using a base image that does not contain the specified shell executable.

The default shell for the shell form can be changed using the `SHELL` command.

In the shell form you can use a `\` (backslash) to continue a single `RUN` instruction onto the next line. For example, consider these two lines:

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

Together they are equivalent to this single line:

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

Note: To use a different shell, other than '/bin/sh', use the exec form passing in the desired shell. For example, `RUN ["/bin/bash", "-c", "echo hello"]`

Note: The exec form is parsed as a JSON array, which means that you must use double-quotes (") around words not single-quotes (').

Note: Unlike the shell form, the exec form does not invoke a command shell. This means that normal shell processing does not happen. For example, `RUN [ "echo", "$HOME" ]` will not do variable substitution on `$HOME`. If you want shell processing then either use the shell form or execute a shell directly, for example: `RUN [ "sh", "-c", "echo $HOME" ]`.

Note: In the JSON form, it is necessary to escape backslashes. This is particularly relevant on Windows where the backslash is the path separator. The following line would otherwise be treated as shell form due to not being valid JSON, and fail in an unexpected way: `RUN ["c:\windows\system32\tasklist.exe"]`

The correct syntax for this example is: `RUN ["c:\\windows\\system32\\tasklist.exe"]`

The cache for `RUN` instructions isn't invalidated automatically during the next build. The cache for

an instruction like `RUN apt-get dist-upgrade -y` will be reused during the next build. The cache for `RUN` instructions can be invalidated by using the --no-cache flag, for example docker build --no-cache.

See the Dockerfile Best Practices guide for more information.

The cache for `RUN` instructions can be invalidated by `ADD` instructions. See below for details.

ONBUILD Instruction

```
ONBUILD [INSTRUCTION]
```

The `ONBUILD` instruction adds to the image a trigger instruction to be executed at a later time, when the image is used as the base for another build. The trigger will be executed in the context of the downstream build, as if it had been inserted immediately after the `FROM` instruction in the downstream Dockerfile.

Any build instruction can be registered as a trigger.

This is useful if you are building an image which will be used as a base to build other images, for example an application build environment or a daemon which may be customized with user-specific configuration.

For example, if your image is a reusable Python application builder, it will require application source code to be added in a particular directory, and it might require a build script to be called after that. You can't just call `ADD` and `RUN` now, because you don't yet have access to the application source code, and it will be different for each application build. You could simply provide application developers with a boilerplate Dockerfile to copy-paste into their application, but that is inefficient, error-prone and difficult to update because it mixes with application-specific code.

The solution is to use `ONBUILD` to register advance instructions to run later, during the next build stage.

Here's how it works:

When it encounters an `ONBUILD` instruction, the builder adds a trigger to the metadata of the image being built. The instruction does not otherwise affect the current build.

At the end of the build, a list of all triggers is stored in the image manifest, under the key OnBuild. They can be inspected with the `docker inspect` command. Later the image may be used as a base for a new build, using the `FROM` instruction. As part of processing the `FROM` instruction, the downstream builder looks for `ONBUILD` triggers, and executes them in the same order they were registered. If any of the triggers fail, the `FROM` instruction is aborted which in turn causes the build to fail. If all triggers succeed, the `FROM` instruction completes and the build continues as usual.

Triggers are cleared from the final image after being executed. In other words they are not inherited by "grand-children" builds.

For example you might add something like this:

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

Warning: Chaining `ONBUILD` instructions using `ONBUILD ONBUILD` isn't allowed.

Warning: The `ONBUILD` instruction may not trigger `FROM` or `MAINTAINER` instructions.

STOPSIGNAL Instruction

```
STOPSIGNAL signal
```

The `STOPSIGNAL` instruction sets the system call signal that will be sent to the container to exit. This signal can be a valid unsigned number that matches a position in the kernel's syscall table, for instance 9, or a signal name in the format SIGNAME, for instance SIGKILL.

HEALTHCHECK Instruction

The `HEALTHCHECK` instruction has two forms:

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

The `HEALTHCHECK` instruction tells Docker how to test a container to check that it is still working. This can detect cases such as a web server that is stuck in an infinite loop and unable to handle new connections, even though the server process is still running.

When a container has a healthcheck specified, it has a health status in addition to its normal status. This status is initially starting. Whenever a health check passes, it becomes healthy (whatever state it was previously in). After a certain number of consecutive failures, it becomes unhealthy.

The options that can appear before `CMD` are:

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

The health check will first run interval seconds after the container is started, and then again interval seconds after each previous check completes.

If a single run of the check takes longer than timeout seconds then the check is considered to have failed.

It takes retries consecutive failures of the health check for the container to be considered unhealthy.

There can only be one `HEALTHCHECK` instruction in a `Dockerfile`. If you list more than one then only the last `HEALTHCHECK` will take effect.

The command after the `CMD` keyword can be either a shell command (e.g. `HEALTHCHECK CMD /bin/check-running`) or an exec array (as with other Dockerfile commands; see e.g. `ENTRYPOINT` for details).

The command's exit status indicates the health status of the container. The possible values are:

- `0: success` - the container is healthy and ready for use
- `1: unhealthy` - the container is not working correctly
- `2: starting` - the container is not ready for use yet, but is working correctly

If the probe returns 2 ("starting") when the container has already moved out of the "starting" state then it is treated as "unhealthy" instead.

For example, to check every five minutes or so that a web-server is able to serve the site's main page within three seconds:

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

To help debug failing probes, any output text (UTF-8 encoded) that the command writes on stdout or stderr will be stored in the health status and can be queried with `docker inspect`. Such output should be kept short (only the first 4096 bytes are stored currently).

When the health status of a container changes, a `health_status` event is generated with the new status.

The `HEALTHCHECK` feature was added in Docker 1.12.

SHELL Instruction

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

The `SHELL` instruction allows the default shell used for the shell form of commands to be overridden. The default shell on Linux is `["/bin/sh", "-c"]`, and on Windows is `["cmd", "/S", "/C"]`. The `SHELL` instruction must be written in JSON form in a Dockerfile.

The `SHELL` instruction is particularly useful on Windows where there are two commonly used and quite different native shells: cmd and powershell, as well as alternate shells available including sh.

The `SHELL` instruction can appear multiple times. Each `SHELL` instruction overrides all previous `SHELL` instructions, and affects all subsequent instructions. For example:

```
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
```

The following instructions can be affected by the `SHELL` instruction when the shell form of them is used in a Dockerfile: `RUN`, `CMD` and `ENTRYPOINT`.

The following example is a common pattern found on Windows which can be streamlined by using the `SHELL` instruction:

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

The command invoked by docker will be:

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

This is inefficient for two reasons. First, there is an un-necessary cmd.exe command processor (aka shell) being invoked. Second, each `RUN` instruction in the shell form requires an extra powershell -command prefixing the command.

To make this more efficient, one of two mechanisms can be employed. One is to use the JSON form of the `RUN` command such as:

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

While the JSON form is unambiguous and does not use the un-necessary cmd.exe, it does require more verbosity through double-quoting and escaping. The alternate mechanism is to use the `SHELL` instruction and the shell form, making a more natural syntax for Windows users, especially when combined with the escape parser directive:

```
# 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'
```

Resulting in:

```
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>
```

The `SHELL` instruction could also be used to modify the way in which a shell operates. For example, using `SHELL cmd /S /C /V:ON|OFF` on Windows, delayed environment variable expansion semantics could be modified.

The `SHELL` instruction can also be used on Linux should an alternate shell be required such zsh, csh, tcsh and others.

The `SHELL` feature was added in Docker 1.12.

Installing Debian/Ubuntu packages

Run the install on a single run command to merge the update and install. If you add more packages later, this will run the update again and install all the packages needed. If the update is run separately, it will be cached and package installs may fail. Setting the frontend to noninteractive and passing the -y to install is needed for scripted installs. Cleaning and purging at the end of the install minimizes the size of the layer.

```
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/*
```

Read Dockerfiles online: https://riptutorial.com/docker/topic/3161/dockerfiles

Chapter 23: How to debug when docker build fails

Introduction

When a `docker build -t mytag .` fails with a message such as `---> Running in d9a42e53eb5a The command '/bin/sh -c returned a non-zero code: 127` (127 means "command not found, but 1) it is not trivial for everybody 2) 127 may be replaced by 6 or anything) it may be non trivial to find the error in a long line

Examples

basic example

As the last layer created by

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

showed

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

You just launch the last created image with a shell and launch the command, and you will have a more clear error message

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

(this assumes /bin/bash is available, it may be /bin/sh or anything else)

and with the prompt, you launch the last failing command, and see what is displayed

Read How to debug when docker build fails online: https://riptutorial.com/docker/topic/8078/how-to-debug-when-docker-build-fails

Chapter 24: How to Setup Three Node Mongo Replica using Docker Image and Provisioned using Chef

Introduction

This Documentation describe how to build a three node Mongo replica set using Docker Image and auto provisioned using Chef.

Examples

Build Step

Steps:

1. Generate a Base 64 keyfile for Mongo node authentication. Place this file in chef data_bags
2. Go to chef suppermarket and download docker cookbook. Generate a custom cookbook (e.g custom_mongo) and add depends 'docker', '~> 2.0' to your cookbook's metadata.rb
3. Create an attributes and recipe in your custom cookbook
4. Initialise Mongo to form Rep Set cluster

Step 1: Create Key file

create data_bag called mongo-keyfile and item called keyfile. This will be in the data_bags directory in chef. Item content will be as below

```
openssl rand -base64 756 > <path-to-keyfile>
```

keyfile item content

```
{
  "id": "keyfile",
  "comment": "Mongo Repset keyfile",
  "key-file": "generated base 64 key above"
}
```

Step 2: Download docker cookbook from chef supper market and then create custom_mongo cookbook

```
knife cookbook site download docker
knife cookbook create custom_mongo
```

in metadat.rb of custom_mongo add

```
depends          'docker', '~> 2.0'
```

Step 3: create attribute and recipe

Attributes

```
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'
```

Recipe

```
#
# 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
```

Create Role called mongo-role in role directory

```
{
  "name": "mongo-role",
  "description": "mongo DB Role",
  "run_list": [
    "recipe[custom_mongo]"
  ]
}
```

Add role above to the three mongo nodes run list

```
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]'
```

Step 4: Initialise the three node Mongo to form repset

I'm assuming that the above role has already been applied on all three Mongo nodes. On node 01 only, Start Mongo with --auth to enable authentication

```
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
```

Access the interactive shell of running docker container on node 01 and Create admin user

```
docker exec -it mongo /bin/sh
    mongo
    use admin
    db.createUser( {
         user: "admin-user",
         pwd: "password",
         roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
       });
```

Create root user

```
db.createUser( {
         user: "RootAdmin",
         pwd: "password",
         roles: [ { role: "root", db: "admin" } ]
       });
```

Stop and Delete the Docker container created above on node 01. This will not affect the data and keyfile in the host DIR. After deleting start Mongo again on node 01 but this time with with 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"
```

now start mongo on Node 02 and 03 with the rep set flag

```
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"
```

Authenticate with the root user on Node 01 and initiate the replica set

```
use admin
db.auth("RootAdmin", "password");
rs.initiate()
```

On node 01 add Node 2 and 3 to the Replica Set to form repset0 cluster

```
rs.add("mongo-02.example.com")
rs.add("mongo-03.example.com")
```

Testing

On the primary run db.printSlaveReplicationInfo() and observe the SyncedTo and Behind the primary time. The later should be 0 sec as below

Output

```
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
```

I hope this helps someone

Read How to Setup Three Node Mongo Replica using Docker Image and Provisioned using Chef online: https://riptutorial.com/docker/topic/10014/how-to-setup-three-node-mongo-replica-using-docker-image-and-provisioned-using-chef

Chapter 25: Inspecting a running container

Syntax

- docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE...]

Examples

Get container information

To get all the information for a container you can run:

```
docker inspect <container>
```

Get specific information from a container

You can get an specific information from a container by running:

```
docker inspect -f '<format>' <container>
```

For instance, you can get the Network Settings by running:

```
docker inspect -f '{{ .NetworkSettings }}' <container>
```

You can also get just the IP address:

```
docker inspect -f '{{ .NetworkSettings.IPAddress }}' <container>
```

The parameter -f means format and will receive a Go Template as input to format what is expected, but this won't bring a beautiful return, so try:

```
docker inspect -f '{{ json .NetworkSettings }}' {{containerIdOrName}}
```

the json keyword will bring the return as a JSON.

So to finish, a little tip is to use python in there to format the output JSON:

```
docker inspect -f '{{ json .NetworkSettings }}' <container> | python -mjson.tool
```

And voila, you can query anything on the docker inspect and make it look pretty in your terminal.

It's also possible to use a utility called "jq" in order to help process `docker inspect` command output.

```
docker inspect -f '{{ json .NetworkSettings }}' aa1 | jq [.Gateway]
```

The above command will return the following output:

```
[
  "172.17.0.1"
]
```

This output is actually a list containing one element. Sometimes, `docker inspect` displays a list of several elements, and you may want to refer to a specific element. For example, if `Config.Env` contains several elements, you can refer to the first element of this list using `index`:

```
docker inspect --format '{{ index (index .Config.Env) 0 }}' <container>
```

The first element is indexed at zero, which means that the second element of this list is at index `1`:

```
docker inspect --format '{{ index (index .Config.Env) 1 }}' <container>
```

Using `len` it is possible to get the number of elements of the list:

```
docker inspect --format `{{ len .Config.Env }}` <container>
```

And using negative numbers, it's possible to refer to the last element of the list:

```
docker inspect -format "{{ index .Config.Cmd $[$(docker inspect -format `{{ len .Config.Cmd
}}` <container>)-1]}}" <container>
```

Some `docker inspect` information comes as a dictionary of key:value, here is an extract of a `docker inspect`of a jess/spotify running container

```
"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": {}
},
```

so I an get the values of the whole Config section

```
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[] }
```

but also a single field, like the value of Config.Image

```
docker inspect -f '{{index (.Config) "Image" }}' 825

jess/spotify
```

or Config.Cmd

```
docker inspect -f '{{.Config.Cmd}}' 825
```

```
[-stylesheet=/home/spotify/spotify-override.css]
```

Inspect an image

In order to inspect an image, you can use the image ID or the image name, consisting of repository and tag. Say, you have the CentOS 6 base image:

```
➜ ~ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
centos              centos6             cf2c3ece5e41        2 weeks ago         194.6 MB
```

In this case you can run either of the following:

- ➜ ~ docker inspect cf2c3ece5e41
- ➜ ~ docker inspect centos:centos6

Both of these command will give you all information available in a JSON array:

```
[
    {
        "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"
            ]
        }
    }
]
```


Printing specific informations

docker `inspect` supports Go Templates via the `--format` option. This allows for better integration in scripts, without resorting to pipes/sed/grep traditional tools.

Print a container internal IP:

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' 7786807d8084
```

This is useful for direct network access of load-balancers auto-configuration.

Print a container *init* PID:

```
docker inspect --format '{{ .State.Pid }}' 7786807d8084
```

This is useful for deeper inspection via `/proc` or tools like `strace`.

Advanced formating:

```
docker inspect --format 'Container {{ .Name }} listens on {{ .NetworkSettings.IPAddress }}:{{
range $index, $elem := .Config.ExposedPorts }}{{ $index }}{{ end }}' 5765847de886 7786807d8084
```

Will output:

```
Container /redis listens on 172.17.0.3:6379/tcp
Container /api listens on 172.17.0.2:4000/tcp
```

Debugging the container logs using docker inspect

`docker inspect` command can be used to debug the container logs.

The stdout and stderr of container can be checked to debug the container, whose location can be obtained using `docker inspect`.

Command : `docker inspect <container-id> | grep Source`

It gives the location of containers stdout and stderr.

Examining stdout/stderr of a running container

```
docker logs --follow <containerid>
```

This tails the output of the running container. This is useful if you did not set up a logging driver on the docker daemon.

Read Inspecting a running container online: https://riptutorial.com/docker/topic/1336/inspecting-a-running-container

Chapter 26: Iptables with Docker

Introduction

This topic is about how to limit access to your docker containers from outside world using iptables.

For impatient people, you can check the examples. For the others, please read the remark section to understand how to build new rules.

Syntax

- iptables -I DOCKER [RULE ...] [ACCEPT|DROP] // To add a rule a the top of the DOCKER table
- iptables -D DOCKER [RULE ...] [ACCEPT|DROP] // To remove a rule from the DOCKER table
- ipset restore < /etc/ipfriends.conf // To reconfigure your ipset *ipfriends*

Parameters

Parameters	Details
ext_if	Your external interface on Docker host.
XXX.XXX.XXX.XXX	A particular IP on which Docker containers access should be given.
YYY.YYY.YYY.YYY	Another IP on which Docker containers access should be given.
ipfriends	The ipset name defining the IPs allowed to access your Docker containers.

Remarks

The problem

Configuring iptables rules for Docker containers is a bit tricky. At first, you would think that "classic" firewall rules should do the trick.

For example, let's assume that you have configured a nginx-proxy container + several service containers to expose via HTTPS some personal web services. Then a rule like this should give access to your web services only for IP XXX.XXX.XXX.XXX.

```
$ iptables -A INPUT -i eth0 -p tcp -s XXX.XXX.XXX.XXX -j ACCEPT
$ iptables -P INPUT DROP
```

It won't work, your containers are still accessible for everyone.

Indeed, Docker containers are not host services. They rely on a virtual network in your host, and the host acts as a gateway for this network. And concerning gateways, routed traffic is not handled by the INPUT table, but by the FORWARD table, which makes the rule posted before uneffective.

But it's not all. In fact, Docker daemon creates a lot of iptables rules when it starts to do its magic concerning containers network connectivity. In particular, a DOCKER table is created to handle rules concerning containers by forwarding traffic from the FORWARD table to this new 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
```

The solution

If you check the official documentation (https://docs.docker.com/v1.5/articles/networking/), a first solution is given to limit Docker container access to one particular IP.

```
$ iptables -I DOCKER -i ext_if ! -s 8.8.8.8 -j DROP
```

Indeed, adding a rule at the top of the DOCKER table is a good idea. It does not interfere with the rules automatically configured by Docker, and it is simple. But two major lacks :

- First, what if you need to access from two IP instead of one ? Here only one src IP can be accepted, other will be dropped without any way to prevent that.
- Second, what if your docker need access to Internet ? Pratically no request will succeed, as

only the server 8.8.8.8 could respond to them.
- Finally, what if you want to add other logics ? For example, give access to any user to your webserver serving on HTTP protocol, but limit everything else to particular IP.

For the first observation, we can use *ipset*. Instead of allowing one IP in the rule above, we allow all IPs from the predefined ipset. As a bonus, the ipset can be updated without the necessity to redefine the iptable rule.

```
$ iptables -I DOCKER -i ext_if -m set ! --match-set my-ipset src -j DROP
```

For the second observation, this is a canonical problem for firewalls : if you are allowed to contact a server through a firewall, then the firewall should authorize the server to respond to your request. This can be done by authorizing packets which are related to an established connection. For the docker logic, it gives :

```
$ iptables -I DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
```

The last observation focuses on one point : iptables rules is essential. Indeed, additional logic to ACCEPT some connections (including the one concerning ESTABLISHED connections) must be put at the top of the DOCKER table, before the DROP rule which deny all remaining connections not matching the ipset.

As we use the -I option of iptable, which inserts rules at the top of the table, previous iptables rules must be inserted by reverse order :

```
// 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
```

With all of this in mind, you can now check the examples which illustrate this configuration.

Examples

Limit access on Docker containers to a set of IPs

First, install *ipset* if needed. Please refer to your distribution to know how to do it. As an example, here is the command for Debian-like distributions.

```
$ apt-get update
$ apt-get install ipset
```

Then create a configuration file to define an ipset containing the IPs for which you want to open access to your Docker containers.

```
$ 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
```

Load this ipset.

```
$ ipset restore < /etc/ipfriends.conf
```

Be sure that your Docker daemon is running : no error should be shown after entering the following command.

```
$ docker ps
```

You are ready to insert your iptables rules. You **must** respect the order.

```
// 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
```

If you want to create new rules, you will need to remove all custom rules you've added before inserting the new ones.

```
$ 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
```

Configure restriction access when Docker daemon starts

Work in progress

Some custom iptables rules

Work in progress

Read Iptables with Docker online: https://riptutorial.com/docker/topic/9201/iptables-with-docker

Chapter 27: Logging

Examples

Configuring a log driver in systemd service

```
[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
```

This enables syslog logging for the docker daemon. The file should be created in the appropriate directory with owner root, which typically would be `/etc/systemd/system/docker.service.d` on e.g. Ubuntu 16.04.

Overview

Docker's approach to logging is that you construct your containers in such a way, so that logs are written to standard output (console/terminal).

If you already have a container which writes logs to a file, you can redirect it by creating a symbolic link:

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

After you've done that you can use various log drivers to put your logs where you need them.

Read Logging online: https://riptutorial.com/docker/topic/7378/logging

Chapter 28: Managing containers

Syntax

- docker rm [OPTIONS] CONTAINER [CONTAINER...]
- docker attach [OPTIONS] CONTAINER
- docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
- docker ps [OPTIONS]
- docker logs [OPTIONS] CONTAINER
- docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE...]

Remarks

- In the examples above, whenever container is a parameter of the docker command, it is mentioned as `<container>` or `container id` or `<CONTAINER_NAME>`. In all these places you can either pass a container name or container id to specify a container.

Examples

Listing containers

```
$ 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` on its own only prints currently running containers. To view all containers (including stopped ones), use the `-a` flag:

```
$ 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
```

To list containers with a specific status, use the `-f` command line option to filter the results. Here is an example of listing all containers which have exited:

```
$ 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
```

It is also possible to list only the Container IDs with the `-q` switch. This makes it very easy to operate on the result with other Unix utilities (such as `grep` and `awk`):

```
$ docker ps -aq
9cc69f11a0f7
2bc9b1988080
817879be2230
```

When launching a container with `docker run --name mycontainer1` you give a specific name and not a random name (in the form mood_famous, such as nostalgic_stallman), and it can be easy to find them with such a command

```
docker ps -f name=mycontainer1
```

Referencing containers

Docker commands which take the name of a container accept three different forms:

Type	Example
Full UUID	`9cc69f11a0f76073e87f25cb6eaf0e079fbfbd1bc47c063bcd25ed3722a8cc4a`
Short UUID	`9cc69f11a0f7`
Name	`berserk_wozniak`

Use `docker ps` to view these values for the containers on your system.

The UUID is generated by Docker and cannot be modified. You can provide a name to the container when you start it `docker run --name <given name> <image>`. Docker will generate a random name to the container if you don't specify one at the time of starting the container.

***NOTE**: The value of the UUID (or a 'short' UUID) can be any length as long as the given value is unique to one container*

Starting and stopping containers

To stop a running container:

```
docker stop <container> [<container>...]
```

This will send the main process in the container a SIGTERM, followed by a SIGKILL if it doesn't stop within the grace period. The name of each container is printed as it stops.

To start a container which is stopped:

```
docker start <container> [<container>...]
```

This will start each container passed in the background; the name of each container is printed as it starts. To start the container in the foreground, pass the `-a` (`--attach`) flag.

List containers with custom format

```
docker ps --format 'table {{.ID}}\t{{.Names}}\t{{.Status}}'
```

Finding a specific container

```
docker ps --filter name=myapp_1
```

Find container IP

To find out the IP address of your container, use:

```
docker inspect <container id> | grep IPAddress
```

or use docker inspect

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' ${CID}
```

Restarting docker container

```
docker restart <container> [<container>...]
```

Option **--time** : Seconds to wait for stop before killing the container (default 10)

```
docker restart <container> --time 10
```

Remove, delete and cleanup containers

`docker rm` can be used to remove a specific containers like this:

```
docker rm <container name or id>
```

To remove all containers you can use this expression:

```
docker rm $(docker ps -qa)
```

By default docker will not delete a container that is running. Any container that is running will produce a warning message and not be deleted. All other containers will be deleted.

Alternatively you can use `xargs`:

```
docker ps -aq -f status=exited | xargs -r docker rm
```

Where `docker ps -aq -f status=exited` will return a list of container IDs of containers that have a status of "Exited".

> Warning: All the above examples will only remove 'stopped' containers.

To remove a container, regardless of whether or not it is stopped, you can use the force flag `-f`:

```
docker rm -f <container name or id>
```

To remove all containers, regardless of state:

```
docker rm -f $(docker ps -qa)
```

If you want to remove only containers with a `dead` status:

```
docker rm $(docker ps --all -q -f status=dead)
```

If you want to remove only containers with an `exited` status:

```
docker rm $(docker ps --all -q -f status=exited)
```

These are all permutations of filters used when listing containers.

To remove both unwanted containers and dangling images that use space after version 1.3, use the following (similar to the Unix tool `df`):

```
$ docker system df
```

To remove all unused data:

```
$ docker system prune
```

Run command on an already existing docker container

```
docker exec -it <container id> /bin/bash
```

It is common to log in an already running container to make some quick tests or see what the application is doing. Often it denotes bad container use practices due to logs and changed files should be placed in volumes. This example allows us log in the container. This supposes that /bin/bash is available in the container, it can be /bin/sh or something else.

```
docker exec <container id> tar -czvf /tmp/backup.tgz /data
docker cp <container id>:/tmp/backup.tgz .
```

This example archives the content of data directory in a tar. Then with `docker cp` you can retrieve it.

Container logs

```
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
```

For example:

```
$ 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"
```

Connect to an instance running as daemon

There are two ways to achieve that, the first and most known is the following:

```
docker attach --sig-proxy=false <container>
```

This one literally attaches your bash to the container bash, meaning that if you have a running script, you will see the result.

To detach, just type: `Ctl-P Ctl-Q`

But if you need a more friendly way and to be able to create new bash instances, just run the following command:

```
docker exec -it <container> bash
```

Copying file from/to containers

from container to host

```
docker cp CONTAINER_NAME:PATH_IN_CONTAINER PATH_IN_HOST
```

from host to container

```
docker cp PATH_IN_HOST CONTAINER_NAME:PATH_IN_CONTAINER
```

If I use jess/transmission from

https://hub.docker.com/r/jess/transmission/builds/bsn7eqxrkzrhxazcuytbmzp/

, the files in the container are in /transmission/download

and my current directory on the host is /home/$USER/abc, after

```
docker cp transmission_id_or_name:/transmission/download .
```

I will have the files copied to

```
/home/$USER/abc/transmission/download
```

you can not, using `docker cp` copy only one file, you copy the directory tree and the files

Remove, delete and cleanup docker volumes

Docker volumes are not automatically removed when a container is stopped. To remove associated volumes when you stop a container:

```
docker rm -v <container id or name>
```

If the `-v` flag is not specified, the volume remains on-disk as a 'dangling volume'. To delete all dangling volumes:

```
docker volume rm $(docker volume ls -qf dangling=true)
```

The `docker volume ls -qf dangling=true` filter will return a list of docker volumes names, including untagged ones, that are not attached to a container.

Alternatively, you can use `xargs`:

```
docker volume ls -f dangling=true -q | xargs --no-run-if-empty docker volume rm
```

Export and import Docker container filesystems

It is possible to save a Docker container's filesystem contents to a tarball archive file. This is useful in a pinch for moving container filesystems to different hosts, for example if a database container has important changes and it isn't otherwise possible to replicate those changes elsewhere. **Please note** that it is preferable to create an entirely new container from an updated image using a `docker run` command or `docker-compose.yml` file, instead of exporting and moving a container's filesystem. Part of Docker's power is the auditability and accountability of its declarative style of creating images and containers. By using `docker export` and `docker import`, this power is subdued because of the obfuscation of changes made inside of a container's filesystem from its original state.

```
docker export -o redis.tar redis
```

The above command will create an empty image and then export the filesystem of the `redis` container into this empty image. To import from a tarball archive, use:

```
docker import ./redis.tar redis-imported:3.0.7
```

This command will create the `redis-imported:3.0.7` image, from which containers can be created. It is also possible to create changes on import, as well as set a commit message:

```
docker import -c="ENV DEBUG true" -m="enable debug mode" ./redis.tar redis-changed
```

The Dockerfile directives available for use with the `-c` command line option are `CMD`, `ENTRYPOINT`, `ENV`, `EXPOSE`, `ONBUILD`, `USER`, `VOLUME`, `WORKDIR`.

Read Managing containers online: https://riptutorial.com/docker/topic/689/managing-containers

Chapter 29: Managing images

Syntax

- docker images [OPTIONS] [REPOSITORY[:TAG]]
- docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE...]
- docker pull [OPTIONS] NAME[:TAG|@DIGEST]
- docker rmi [OPTIONS] IMAGE [IMAGE...]
- docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]

Examples

Fetching an image from Docker Hub

Ordinarily, images are pulled automatically from Docker Hub. Docker will attempt to pull any image from Docker Hub that doesn't already exist on the Docker host. For example, using `docker run ubuntu` when the `ubuntu` image is not already on the Docker host will cause Docker to initiate a pull of the latest `ubuntu` image. It is possible to pull an image separately by using `docker pull` to manually fetch or update an image from Docker Hub.

```
docker pull ubuntu
docker pull ubuntu:14.04
```

Additional options for pulling from a different image registry or pulling a specific version of an image exist. Indicating an alternate registry is done using the full image name and optional version. For example, the following command will attempt to pull the `ubuntu:14.04` image from the `registry.example.com` registry:

```
docker pull registry.example.com/username/ubuntu:14.04
```

Listing locally downloaded images

```
$ 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
```

Referencing images

Docker commands which take the name of an image accept four different forms:

Type	Example
Short ID	`693bce725149`

Type	Example
Name	`hello-world` *(defaults to `:latest` tag)*
Name+tag	`hello-world:latest`
Digest	`hello-world@sha256:e52be8ffeeb1f374f440893189cd32f44cb166650e7ab185fa7735b7dc48d619`

Note: You can only refer to an image by its digest if that image was originally pulled using that digest. To see the digest for an image (if one is available) run `docker images --digests`.

Removing Images

The `docker rmi` command is used to remove images:

```
docker rmi <image name>
```

The full image name must be used to remove an image. Unless the image has been tagged to remove the registry name, it needs to be specified. For example:

```
docker rmi registry.example.com/username/myAppImage:1.3.5
```

It is also possible to remove images by their ID instead:

```
docker rmi 693bce725149
```

As a convenience, it is possible to remove images by their image ID by specifying only the first few characters of the image ID, as long as the substring specified is unambiguous:

```
docker rmi 693
```

> **Note:** Images can be removed even if there are existing containers that use that image; docker rmi simply "untags" the image.

If no containers are using an image it is garbage-collected. If a container uses an image, the image will be garbage-collected once all the containers using it are removed. For example:

```
$ 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
```

Remove All Images With No Started Containers

To remove all local images that have no started containers, you can provide a listing of the images as a parameter:

```
docker rmi $(docker images -qa)
```

Remove All Images

If you want to remove images regardless of whether or not they have a started container use the force flag (`-f`):

```
docker rmi -f $(docker images -qa)
```

Remove Dangling Images

If an image is not tagged and not being used by any container, it is 'dangling' and may be removed like this:

```
docker images -q --no-trunc -f dangling=true | xargs -r docker rmi
```

Search the Docker Hub for images

You can search Docker Hub for images by using the search command:

```
docker search <term>
```

For example:

```
$ 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]
...
```

Inspecting images

```
docker inspect <image>
```

The output is in JSON format. You can use `jq` command line utility to parse and print only the desired keys.

```
docker inspect <image> | jq -r '.[0].Author'
```

The above command will shows author name of the images.

Tagging images

Tagging an image is useful for keeping track of different image versions:

```
docker tag ubuntu:latest registry.example.com/username/ubuntu:latest
```

Another example of tagging:

```
docker tag myApp:1.4.2 myApp:latest
docker tag myApp:1.4.2 registry.example.com/company/myApp:1.4.2
```

Saving and loading Docker images

```
docker save -o ubuntu.latest.tar ubuntu:latest
```

This command will save the `ubuntu:latest` image as a tarball archive in the current directory with the name `ubuntu.latest.tar`. This tarball archive can then be moved to another host, for example using `rsync`, or archived in storage.

Once the tarball has been moved, the following command will create an image from the file:

```
docker load -i /tmp/ubuntu.latest.tar
```

Now it is possible to create containers from the `ubuntu:latest` image as usual.

Read Managing images online: https://riptutorial.com/docker/topic/690/managing-images

Chapter 30: Multiple processes in one container instance

Remarks

Usually each container should hosts one process. In case you need multiple processes in one container (e.g. an SSH server to login to your running container instance) you could get the idea to write you own shell script that starts those processes. In that case you had to take care about the `SIGNAL` handling yourself (e.g. redirecting a caught `SIGINT` to the child processes of your script). That's not really what you want. A simple solution is to use `supervisord` as the containers root process which takes care about `SIGNAL` handling and its child processes lifetime.

But keep in mind, that this ist not the "docker way". To achive this example in the docker way you would log into the `docker host` (the machine the container runs on) and run `docker exec -it container_name /bin/bahs`. This command opens you a shell inside the container as ssh would do.

Examples

Dockerfile + supervisord.conf

To run multiple processes e.g. an Apache web server together with an SSH daemon inside the same container you can use `supervisord`.

Create your `supervisord.conf` configuration file like:

```
[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"
```

Then create a `Dockerfile` like:

```
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"]
```

Then you can build your image:

```
docker build -t supervisord-test .
```

Afterwards you can run it:

```
$ 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
...
```

Read Multiple processes in one container instance online:
https://riptutorial.com/docker/topic/4053/multiple-processes-in-one-container-instance

Chapter 31: passing secret data to a running container

Examples

ways to pass secrets in a container

The not very secure way (because `docker inspect` will show it) is to pass an environment variable to

```
docker run
```

such as

```
docker run -e password=abc
```

or in a file

```
docker run --env-file myfile
```

where myfile can contain

```
password1=abc password2=def
```

it is also possible to put them in a volume

```
docker run -v $(pwd)/my-secret-file:/secret-file
```

some better ways, use

keywhiz https://square.github.io/keywhiz/

vault https://www.hashicorp.com/blog/vault.html

etcd with crypt https://xordataexchange.github.io/crypt/

Read passing secret data to a running container online: https://riptutorial.com/docker/topic/6481/passing-secret-data-to-a-running-container

Chapter 32: Restricting container network access

Remarks

Example docker networks that blocks traffic. Use as the network when starting the container with `--net` or `docker network connect`.

Examples

Block access to LAN and out

```
docker network create -o "com.docker.network.bridge.enable_ip_masquerade"="false" lan-restricted
```

- Blocks
  - Local LAN
  - Internet
- Does not block
  - Host running docker daemon (example access to `10.0.1.10:22`)

Block access to other containers

```
docker network create -o "com.docker.network.bridge.enable_icc"="false" icc-restricted
```

- Blocks
  - Containers accessing other containers on the same `icc-restricted` network.
- Does not block
  - Access to host running docker daemon
  - Local LAN
  - Internet

Block access from containers to the local host running docker daemon

```
iptables -I INPUT -i docker0 -m addrtype --dst-type LOCAL -j DROP
```

- Blocks
  - Access to host running docker daemon
- Does not block
  - Container to container traffic
  - Local LAN
  - Internet
  - Custom docker networks that doesn't use `docker0`

Block access from containers to the local host running docker daemon (custom network)

```
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
```

Creates a network called `local-host-restricted` which which:

- Blocks
  - Access to host running docker daemon
- Does not block
  - Container to container traffic
  - Local LAN
  - Internet
  - Access originating from other docker networks

Custom networks have bridge names like `br-15bbe9bb5bf5`, so we uses it's subnet instead.

Read Restricting container network access online:
https://riptutorial.com/docker/topic/6331/restricting-container-network-access

Chapter 33: run consul in docker 1.12 swarm

Examples

Run consul in a docker 1.12 swarm

This relies on the official consul docker image to run consul in clustered mode in a docker swarm with new swarm mode in Docker 1.12. This example is based on http://qnib.org/2016/08/11/consul-service/. Briefly the idea is to use two docker swarm services that talk to each other. This solves the problem that you cannot know the ips of individual consul containers up front and allows you to rely on docker swarm's dns.

This assumes you already have a running docker 1.12 swarm cluster with at least three nodes.

You may want to configure a log driver on your docker daemons so that you can see what is happening. I used the syslog driver for this: set the `--log-driver=syslog` option on dockerd.

First create an overlay network for consul:

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

Now bootstrap the cluster with just 1 node (default --replicas is 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
```

You should now have a 1 node cluster. Now bring up the second 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
```

You should now have a four node consul cluster. You can verify this by running on any of the docker containers:

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

Read run consul in docker 1.12 swarm online: https://riptutorial.com/docker/topic/6437/run-consul-in-docker-1-12-swarm

Chapter 34: Running containers

Syntax

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

Examples

Running a container

```
docker run hello-world
```

This will fetch the latest hello-world image from the Docker Hub (if you don't already have it), create a new container, and run it. You should see a message stating that your installation appears to be working correctly.

Running a different command in the container

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

This command tells Docker to create a container from the `docker/whalesay` image and run the command `cowsay 'Hello, StackExchange!'` in it. It should print a picture of a whale saying `Hello, StackExchange!` to your terminal.

If the entrypoint in the image is the default you can run any command that's available in the image:

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

If it has been changed during image build you need to reverse it back to the default

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

Automatically delete a container after running it

Normally, a Docker container persists after it has exited. This allows you to run the container again, inspect its filesystem, and so on. However, sometimes you want to run a container and delete it immediately after it exits. For example to execute a command or show a file from the filesystem. Docker provides the `--rm` command line option for this purpose:

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

This will create a container from the "ubuntu" image, show the content of **/etc/hosts** file and then delete the container immediately after it exits. This helps to prevent having to clean up containers after you're done experimenting.

> **Note:** The `--rm` flag doesn't work in conjunction with the `-d` (`--detach`) flag in docker < 1.13.0.

When `--rm` flag is set, Docker also removes the volumes associated with the container when the container is removed. This is similar to running `docker rm -v my-container`. ***Only volumes that are specified without a name are removed***.

For example, with `docker run -it --rm -v /etc -v logs:/var/log centos /bin/produce_some_logs`, the volume of `/etc` will be removed, but the volume of `/var/log` will not. Volumes inherited via --volumes-from will be removed with the same logic -- if the original volume was specified with a name it will not be removed.

Specifying a name

By default, containers created with `docker run` are given a random name like `small_roentgen` or `modest_dubinsky`. These names aren't particularly helpful in identifying the purpose of a container. It is possible to supply a name for the container by passing the `--name` command line option:

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

Names must be unique; if you pass a name that another container is already using, Docker will print an error and no new container will be created.

Specifying a name will be useful when referencing the container within a Docker network. This works for both background and foreground Docker containers.

Containers on the default bridge network **must** be linked to communicate by name.

Binding a container port to the host

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

In order to use ports on the host have been exposed in an image (via the `EXPOSE` Dockerfile directive, or `--expose` command line option for `docker run`), those ports need to be bound to the host using the `-p` or `-P` command line options. Using `-p` requires that the particular port (and optional host interface) to be specified. Using the uppercase `-P` command line option will force Docker to bind *all* exposed ports in a container's image to the host.

Container restart policy (starting a container at boot)

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

By default, Docker will not restart containers when the Docker daemon restarts, for example after a host system reboot. Docker provides a restart policy for your containers by supplying the `--restart` command line option. Supplying `--restart=always` will always cause a container to be

restarted after the Docker daemon is restarted. **However** when that container is manually stopped (e.g. with `docker stop <container>`), the restart policy will not be applied to the container.

Multiple options can be specified for `--restart` option, based on the requirement (`--restart=[policy]`). These options effect how the container starts at boot as well.

Policy	Result
no	The **default** value. Will not restart container automatically, when container is stopped.
on-failure[:max-retries]	Restart only if the container exits with a failure (`non-zero exit status`). To avoid restarting it indefinitely (in case of some problem), one can limit the number of restart retries the Docker daemon attempts.
always	Always restart the container regardless of the exit status. When you specify `always`, the Docker daemon will try to restart the container indefinitely. The container will also always start on daemon startup, regardless of the current state of the container.
unless-stopped	Always restart the container regardless of its exit status, but do not start it on daemon startup if the container has been put to a stopped state before.

Run a container in background

To keep a container running in the background, supply the `-d` command line option during container startup:

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

The option `-d` runs the container in detached mode. It is also equivalent to `-d=true`.

A container in detached mode cannot be removed automatically when it stops, this means one cannot use the --rm option in combination with -d option.

Assign a volume to a container

A Docker volume is a file or directory which persists beyond the lifetime of the container. It is possible to mount a host file or directory into a container as a volume (bypassing the UnionFS).

Add a volume with the `-v` command line option:

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

This will create a volume and mount it to the path `/data` inside the container.

- Note: You can use the flag `--rm` to automatically remove the volume when the container is removed.

Mounting host directories

To mount a host file or directory into a container:

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

- **When specifying a host directory, an absolute path must be supplied.**

This will mount the host directory `/home/foo/data` onto `/data` inside the container. This "bind-mounted host directory" volume is the same thing as a Linux `mount --bind` and therefore temporarily mounts the host directory over the specified container path for the duration of the container's lifetime. Changes in the volume from either the host or the container are reflected immediately in the other, because they are the same destination on disk.

UNIX example mounting a relative folder

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

Naming volumes

A volume can be named by supplying a string instead of a host directory path, docker will create a volume using that name.

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

After creating a named volume, the volume can then be shared with other containers using that name.

Setting environment variables

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

Both `-e` and `--env` can be used to define environment variables inside of a container. It is possible to supply many environment variables using a text file:

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

Example environment variable file:

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

The `--env-file` flag takes a filename as an argument and expects each line to be in the `VARIABLE=VALUE` format, mimicking the argument passed to `--env`. Comment lines need only be prefixed with `#`.

Regardless of the order of these three flags, the `--env-file` are processed first, and then `-e`/`--env` flags. This way, any environment variables supplied individually with `-e` or `--env` will override

variables supplied in the `--env-var` text file.

Specifying a hostname

By default, containers created with docker run are given a random hostname. You can give the container a different hostname by passing the --hostname flag:

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

Run a container interactively

To run a container interactively, pass in the `-it` options:

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

`-i` keeps STDIN open, while `-t` allocates a pseudo-TTY.

Running container with memory/swap limits

Set memory limit and disable swap limit

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

Set both memory and swap limit. In this case, container can use 300M memory and 700M swap.

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

Getting a shell into a running (detached) container

Log into a running container

A user can enter a running container in a new interactive bash shell with `exec` command.

Say a container is called `jovial_morse` then you can get an interactive, pseudo-TTY bash shell by running:

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

Log into a running container with a specific user

If you want to enter a container as a specific user, you can set it with `-u` or `--user` parameter. The username must exists in the container.

`-u, --user` Username or UID (format: `<name|uid>[:<group|gid>]`)

This command will log into `jovial_morse` with the `dockeruser` user

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

Log into a running container as root

If you want to log in as root, just simply use the **`-u root`** parameter. Root user always exists.

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

Log into a image

You can also log into a image with the `run` command, but this requires an image name instead of a container name.

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

Log into a intermediate image (debug)

You can log into an intermediate image as well, which is created during a Dockerfile build.

Output of `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
```

Notice the `---> Running in 02071fceb21b` output, you can log into these images:

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

Passing stdin to the container

In cases such as restoring a database dump, or otherwise wishing to push some information through a pipe from the host, you can use the `-i` flag as an argument to `docker run` or `docker exec`.

E.g., assuming you want to put to a containerized mariadb client a database dump that you have on the host, in a local `dump.sql` file, you can perform the following command:

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

In general,

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

Or

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

Detaching from a container

While attached to a running container with a pty assigned (`docker run -it ...`), you can press `ControlP` - `ControlQ` to detach.

Overriding image entrypoint directive

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

This command will override the `ENTRYPOINT` directive of the `example-app` image when the container `test-app` is created. The `CMD` directive of the image will remain unchanged unless otherwise specified:

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

In the above example, both the `ENTRYPOINT` and the `CMD` of the image have been overridden. This container process becomes `/bin/bash /app/test.sh`.

Add host entry to container

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

This command adds an entry to the container's `/etc/hosts` file, which follows the format `--add-host <name>:<address>`. In this example, the name `app-backend` will resolve to `10.15.1.24`. This is particularly useful for tying disparate app components together programmatically.

Prevent container from stopping when no commands are running

A container will stop if no command is running on the foreground. Using the `-t` option will keep the container from stopping, even when detached with the `-d` option.

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

Stopping a container

```
docker stop mynginx
```

Additionally, the container id can also be used to stop the container instead of its name.

This will stop a running container by sending the SIGTERM signal and then the SIGKILL signal if necessary.

Further, the kill command can be used to immediately send a SIGKILL or any other specified signal using the `-s` option.

```
docker kill mynginx
```

Specified signal:

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

Stopping a container doesn't delete it. Use `docker ps -a` to see your stopped container.

Execute another command on a running container

When required you can tell Docker to execute additional commands on an already running container using the `exec` command. You need the container's ID which you can get with `docker ps`.

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

You can attach an interactive shell if you use the `-it` option.

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

Running GUI apps in a Linux container

By default, a Docker container won't be able to *run* a GUI application.

Before that, the X11 socket must be forwarded first to the container, so it can be used directly. The *DISPLAY* environment variable must be forwarded as well:

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

This will fail at first, since we didn't set the permissions to the X server host:

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

The quickest (but not safest) way is to allow access directly with:

```
xhost +local:root
```

After finishing with the container, we can go back to the original state with:

```
xhost -local:root
```

Another (safer) way is to prepare a Dockerfile that will build a new image that will use the our user credentials to access the X server:

```
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}
```

When invoking `docker build` from the command line, we have to pass the *ARG* variables that appear in the Dockerfile:

```
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> .
```

Now, before spawning a new container, we have to create a xauth file with access permission:

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

This file has to be mounted into the container when creating/running it:

```
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
```

Read Running containers online: https://riptutorial.com/docker/topic/679/running-containers

Chapter 35: Running services

Examples

Creating a more advanced service

In the following example we will create a service with the name *visualizer*. We will specify a custom label and remap the internal port of the service from 8080 to 9090. In addition we will bind mount an external directory of the host into the 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
```

Creating a simple service

This simple exampe will create a hello world web service that will listen on the port 80.

```
docker service create \
    --publish 80:80 \
    tutum/hello-world
```

Removing a service

This simple example will remove the service with name "visualizer":

```
docker service rm visualizer
```

Scaling a service

This example will scale the service to 4 instances:

```
docker service scale visualizer=4
```

In Docker Swarm Mode we do not stop a service. We scale it down to zero:

```
docker service scale visualizer=0
```

Read Running services online: https://riptutorial.com/docker/topic/8802/running-services

Chapter 36: Running Simple Node.js Application

Examples

Running a Basic Node.js application inside a Container

The example I'm going to discuss assumes you have a Docker installation that works in your system and a basic understanding of how to work with Node.js . If you are aware of how you must work with Docker , it should be evident that Node.js framework need not be installed on your system, rather we would be using the `latest` version of the `node` image available from Docker. Hence if needed you may download the image beforehand with the command `docker pull node`. (The command automatically `pulls` the latest version of the `node` image from docker.)

1. Proceed to make a directory where all your working application files would reside. Create a `package.json` file in this directory that describes your application as well as the dependencies. Your `package.json` file should look something like this:

```
{
  "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. If we need to work with Node.js we usually create a `server` file that defines a web application. In this case we use the `Express.js` framework (version `4.13.3` onwards). A basic `server.js` file would look something like this:

```
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. For those familiar with Docker, you would have come across a `Dockerfile`. A `Dockerfile` is a text file that contains all the commands required to build a custom image that is tailored for your application.

Create an empty text file named `Dockerfile` in the current directory. The method to create one is straightforward in Windows. In Linux, you may want to execute `touch Dockerfile` in the directory containing all the files required for your application. Open the Dockerfile with any text editor and add the following lines:

```
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` instructs the Docker daemon what image we want to build from. In this case we use the `latest` version of the official Docker image `node` available from the Docker Hub.
- Inside this image we proceed to create a working directory that contains all the required files and we instruct the daemon to set this directory as the desired working directory for our application. For this we add

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

- We then proceed to install application dependencies by first moving the `package.json` file (which specifies app info including dependencies) to the `/usr/src/my_first_app` working directory in the image. We do this by

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

- We then type `COPY . /usr/src/my_first_app` to add all the application files and source code to the working directory in the image.
- We then use the `EXPOSE` directive to instruct the daemon to make port `8080` of the resulting container visible (via a container-to-host mapping) since the application binds to port `8080`.
- In the last step, we instruct the daemon to run the command `node server.js` inside the image by executing the basic `npm start` command. We use the `CMD` directive for this, which takes the commands as arguments.

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

4. We then create a `.dockerignore` file in the same directory as the `Dockerfile` to prevent our copy of `node_modules` and logs used by our Node.js system installation from being copied on to the Docker image. The `.dockerignore` file must have the following content:

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

5. ---

Build your image

Navigate to the directory that contains the `Dockerfile` and run the following command to build the Docker image. The `-t` flag lets you tag your image so it's easier to find later using the docker images command:

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

Your image will now be listed by Docker. View images using the below command:

```
$ docker images

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

6. ## Running the image

We can now run the image we just created using the application contents, the `node` base image and the `Dockerfile`. We now proceed to run our newly created `<your username>/node-web-app` image. Providing `-d` switch to the `docker run` command runs the container in detached mode,so that the container runs in the background. The `-p` flag redirects a public port to a private port inside the container. Run the image you previously built using this command:

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

7. Print the output of your app by running `docker ps` on your terminal. The output should look something like this.

```
 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
```

Get application output by entering `docker logs <CONTAINER ID>`. In this case it is `docker logs 7b701693b294`.

Output: `Running on http://localhost:8080`

8. From the `docker ps` output, the port mapping obtained is `0.0.0.0:49160->8080/tcp`. Hence Docker mapped the `8080` port inside of the container to the port 49160 on the host machine. In the browser we can now enter `localhost:49160`.

We can also call our app using `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
```

Read Running Simple Node.js Application online: https://riptutorial.com/docker/topic/8754/running-simple-node-js-application

Chapter 37: security

Introduction

In order to keep our images up to date for the security patches, we need to know from which base image we depend

Examples

How to find from which image our image comes from

As an example, lets us look at a Wordpress container

The Dockerfile begins with FROM php:5.6-apache

so we go to the Dockerfile abovementioned https://github.com/docker-library/php/blob/master/5.6/apache/Dockerfile

and we find FROM debian:jessie So this means that we a security patch appears for Debian jessie, we need to build again our image.

Read security online: https://riptutorial.com/docker/topic/8077/security

Credits

S. No	Chapters	Contributors
1	Getting started with Docker	abaracedo, Aminadav, Braiam, Carlos Rafael Ramirez, Community, ganesshkumar, HankCa, Josha Inglis, L0j1k, mohan08p, Nathaniel Ford, schumacherj, Siddharth Srinivasan, SztupY, Vishrant
2	Building images	cjsimon, ETL, Ken Cochrane, L0j1k, Nathan Arthur, Nathaniel Ford, Nour Chawich, SztupY, user2915097, Wolfgang
3	Checkpoint and Restore Containers	Bastian, Fuzzyma
4	Concept of Docker Volumes	Amit Poonia, Rob Bednark, serieznyi
5	Connecting Containers	Jett Jones
6	Creating a service with persistence	Carlos Rafael Ramirez, Vanuan
7	Data Volumes and Data Containers	GameScripting, L0j1k, melihovv
8	Debugging a container	allprog, Binary Nerd, foraidt, L0j1k, Nathaniel Ford, user2915097, yadutaf
9	Docker Data Volumes	James Hewitt, L0j1k, NRKirby, Nuno Curado, Scott Coates, t3h2mas
10	Docker Engine API	Ashish Bista, atv, BMitch, L0j1k, Radoslav Stoyanov, SztupY
11	Docker events	Nathaniel Ford, user2915097
12	Docker in Docker	Ohmen
13	docker inspect getting various fields for key:value and elements of list	user2915097
14	Docker Machine	Amine24h, kubanczyk, Nik Rahmel, user2915097, yadutaf
15	Docker --net modes	mohan08p

	(bridge, hots, mapped container and none).	
16	Docker network	HankCa, L0j1k, Nathaniel Ford
17	Docker private/secure registry with API v2	bastien enjalbert, kubanczyk
18	Docker Registry	Ashish Bista, L0j1k
19	Docker stats all running containers	Kostiantyn Rybnikov
20	Docker swarm mode	abronan, Christian, Farhad Farahi, Jilles van Gurp, kstromeiraos , kubanczyk, ob1, Philip, Vanuan
21	Dockerfile contents ordering	akhyar, Philip
22	Dockerfiles	BMitch, foraidt, k0pernikus, kubanczyk, L0j1k, ob1, Ohmen, rosysnake, satsumas, Stephen Leppik, Thiago Almeida, Wassim Dhif, yadutaf
23	How to debug when docker build fails	user2915097
24	How to Setup Three Node Mongo Replica using Docker Image and Provisioned using Chef	Innocent Anigbo
25	Inspecting a running container	AlcaDotS, devopskata, Felipe Plets, h3nrik, Jilles van Gurp, L0j1k, Milind Chawre, Nik Rahmel, Stephen Leppik, user2915097, yadutaf
26	Iptables with Docker	Adrien Ferrand
27	Logging	Jilles van Gurp, Vanuan
28	Managing containers	akhyar, atv, Binary Nerd, BrunoLM, Carlos Rafael Ramirez, Emil Burzo, Felipe Plets, ganesshkumar, L0j1k, Matt, Nathaniel Ford, Rafal Wiliński, Sachin Malhotra, serieznyi, sk8terboi87 ヾ(, tommyyards, user2915097, Victor Oliveira Antonino, Wolfgang, Xavier Nicollet, zygimantus
29	Managing images	akhyar, Björn Enochsson, dsw88, L0j1k, Nathan Arthur, Nathaniel Ford, Szymon Biliński, user2915097, Wolfgang,

		zygimantus
30	Multiple processes in one container instance	h3nrik, Ohmen, Xavier Nicollet
31	passing secret data to a running container	user2915097
32	Restricting container network access	xeor
33	run consul in docker 1.12 swarm	Jilles van Gurp
34	Running containers	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
35	Running services	Mateusz Mrozewski, Philip
36	Running Simple Node.js Application	Siddharth Srinivasan
37	security	user2915097