Docker Documentation

0

Team Docker

2015 07 16

Contents

This documentation has the following resources:

CHAPTER 1

Docker is an open-source engine to easily create lightweight, portable, self-sufficient containers from any application. The same container that a developer builds and tests on a laptop can run at scale, in production, on VMs, bare metal, OpenStack clusters, or any major infrastructure provider.

Common use cases for Docker include:

- Automating the packaging and deployment of web applications.
- Automated testing and continuous integration/deployment.
- Deploying and scaling databases and backend services in a service-oriented environment.
- Building custom PaaS environments, either from scratch or as an extension of off-the-shelf platforms like OpenShift or Cloud Foundry.

For a high-level overview of Docker, please see the Introduction. When you're ready to start working with Docker, we have a quick start and a more in-depth guide to *Ubuntu* and other paths including prebuilt binaries, Vagrant-created VMs, Rackspace and Amazon instances.

Enough reading! *Try it out!*

There are a number of ways to install Docker, depending on where you want to run the daemon. The *Ubuntu* installation is the officially-tested version. The community adds more techniques for installing Docker all the time.

Contents:

2.1 Ubuntu

: hosts,docker.conf

```
vim /etc/hosts
54.234.135.251  get.docker.io
54.234.135.251  cdn-registry-1.docker.io

vim /etc/init/docker.conf
#respawn
evn HTTP_PROXY="http://192.241.209.203:8384"
```

: These instructions have changed for 0.6. If you are upgrading from an earlier version, you will need to follow them again.

: Docker is still under heavy development! We don't recommend using it in production yet, but we're getting closer with each release. Please see our blog post, "Getting to Docker 1.0"

Docker is supported on the following versions of Ubuntu:

- *Ubuntu Precise 12.04 (LTS) (64-bit)*
- *Ubuntu Raring 13.04 and Saucy 13.10 (64 bit)*

Please read *Docker*, if you plan to use UFW (Uncomplicated Firewall)

2.1.1 Ubuntu Precise 12.04 (LTS) (64-bit)

This installation path should work at all times.

Linux kernel 3.8

Due to a bug in LXC, Docker works best on the 3.8 kernel. Precise comes with a 3.2 kernel, so we need to upgrade it. The kernel you'll install when following these steps comes with AUFS built in. We also include the generic headers to enable packages that depend on them, like ZFS and the VirtualBox guest additions. If you didn't install the headers for your "precise" kernel, then you can skip these headers for the "raring" kernel. But it is safer to include them if you're not sure.

```
# install the backported kernel
sudo apt-get update
sudo apt-get install linux-image-generic-lts-raring linux-headers-generic-lts-raring

# reboot
sudo reboot
```

: These instructions have changed for 0.6. If you are upgrading from an earlier version, you will need to follow them again.

Docker is available as a Debian package, which makes installation easy. **See the :ref:'installmirrors' section below if you are not in the United States.** Other sources of the Debian packages may be faster for you to install.

First add the Docker repository key to your local keychain.

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 36A1D7869245C8950F966E92D8576A8BA88D21E
```

Add the Docker repository to your apt sources list, update and install the `lxc-docker` package.

You may receive a warning that the package isn't trusted. Answer yes to continue installation.

```
sudo sh -c "echo deb http://get.docker.io/ubuntu docker main\
> /etc/apt/sources.list.d/docker.list"
sudo apt-get update
sudo apt-get install lxc-docker
```

: There is also a simple `curl` script available to help with this process.

```
curl -s https://get.docker.io/ubuntu/ | sudo sh
```

Now verify that the installation has worked by downloading the `ubuntu` image and launching a container.

```
sudo docker run -i -t ubuntu /bin/bash
```

Type `exit` to exit

Done!, now continue with the *Hello World* example.

2.1.2 Ubuntu Raring 13.04 and Saucy 13.10 (64 bit)

These instructions cover both Ubuntu Raring 13.04 and Saucy 13.10.

Optional AUFS filesystem support

Ubuntu Raring already comes with the 3.8 kernel, so we don't need to install it. However, not all systems have AUFS filesystem support enabled. AUFS support is optional as of version 0.7, but it's still available as a driver and we recommend using it if you can.

To make sure AUFS is installed, run the following commands:

```
sudo apt-get update
sudo apt-get install linux-image-extra-`uname -r`
```

Docker is available as a Debian package, which makes installation easy.

: Please note that these instructions have changed for 0.6. If you are upgrading from an earlier version, you will need to follow them again.

First add the Docker repository key to your local keychain.

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 36A1D7869245C8950F966E92D8576A8BA88D21E
```

Add the Docker repository to your apt sources list, update and install the `lxc-docker` package.

```
sudo sh -c "echo deb http://get.docker.io/ubuntu docker main\
> /etc/apt/sources.list.d/docker.list"
sudo apt-get update
sudo apt-get install lxc-docker
```

Now verify that the installation has worked by downloading the `ubuntu` image and launching a container.

```
sudo docker run -i -t ubuntu /bin/bash
```

Type `exit` to exit

Done!, now continue with the *Hello World* example.

root

The `docker` daemon always runs as the root user, and since Docker version 0.5.2, the `docker` daemon binds to a Unix socket instead of a TCP port. By default that Unix socket is owned by the user *root*, and so, by default, you can access it with `sudo`.

Starting in version 0.5.3, if you (or your Docker installer) create a Unix group called *docker* and add users to it, then the `docker` daemon will make the ownership of the Unix socket read/writable by the *docker* group when the daemon starts. The `docker` daemon must always run as the root user, but if you run the `docker` client as a user in the *docker* group then you don't need to add `sudo` to all the client commands.

: The *docker* group is root-equivalent.

Example:

```
# Add the docker group if it doesn't already exist.
sudo groupadd docker

# Add the connected user "${USER}" to the docker group.
# Change the user name to match your preferred user.
# You may have to logout and log back in again for
# this to take effect.
sudo gpasswd -a ${USER} docker

# Restart the Docker daemon.
sudo service docker restart
```

To install the latest version of docker, use the standard `apt-get` method:

```
# update your sources list
sudo apt-get update

# install the latest
sudo apt-get install lxc-docker
```

2.1.3 Troubleshooting

On Linux Mint, the `cgroups-lite` package is not installed by default. Before Docker will work correctly, you will need to install this via:

```
sudo apt-get update && sudo apt-get install cgroups-lite
```

2.1.4 Docker

Docker uses a bridge to manage container networking. By default, UFW drops all *forwarding* traffic. As a result you will need to enable UFW forwarding:

```
sudo nano /etc/default/ufw
----
# Change:
# DEFAULT_FORWARD_POLICY="DROP"
# to
DEFAULT_FORWARD_POLICY="ACCEPT"
```

Then reload UFW:

```
sudo ufw reload
```

UFW's default set of rules denies all *incoming* traffic. If you want to be able to reach your containers from another host then you should allow incoming connections on the Docker port (default 4243):

```
sudo ufw allow 4243/tcp
```

2.1.5 Mirrors

You should `ping get.docker.io` and compare the latency to the following mirrors, and pick whichever one is best for you.

Yandex

Yandex in Russia is mirroring the Docker Debian packages, updating every 6 hours. Substitute `http://mirror.yandex.ru/mirrors/docker/` for `http://get.docker.io/ubuntu` in the instructions above. For example:

```
sudo sh -c "echo deb http://mirror.yandex.ru/mirrors/docker/ docker main\
> /etc/apt/sources.list.d/docker.list"
sudo apt-get update
sudo apt-get install lxc-docker
```

2.2 Red Hat Enterprise Linux

: Docker is still under heavy development! We don't recommend using it in production yet, but we're getting closer with each release. Please see our blog post, "Getting to Docker 1.0"

: This is a community contributed installation path. The only 'official' installation is using the *Ubuntu* installation path. This version may be out of date because it depends on some binaries to be updated and published

Docker is available for **RHEL** on EPEL. These instructions should work for both RHEL and CentOS. They will likely work for other binary compatible EL6 distributions as well, but they haven't been tested.

Please note that this package is part of Extra Packages for Enterprise Linux (EPEL), a community effort to create and maintain additional packages for the RHEL distribution.

Also note that due to the current Docker limitations, Docker is able to run only on the **64 bit** architecture.

2.2.1 Installation

Firstly, you need to install the EPEL repository. Please follow the EPEL installation instructions.

The `docker-io` package provides Docker on EPEL.

If you already have the (unrelated) `docker` package installed, it will conflict with `docker-io`. There's a bug report filed for it. To proceed with `docker-io` installation, please remove `docker` first.

Next, let's install the `docker-io` package which will install Docker on our host.

```
sudo yum -y install docker-io
```

To update the `docker-io` package

```
sudo yum -y update docker-io
```

Now that it's installed, let's start the Docker daemon.

```
sudo service docker start
```

If we want Docker to start at boot, we should also:

```
sudo chkconfig docker on
```

Now let's verify that Docker is working.

```
sudo docker run -i -t fedora /bin/bash
```

Done!, now continue with the *Hello World* example.

2.2.2 Issues?

If you have any issues - please report them directly in the Red Hat Bugzilla for docker-io component.

2.3 Fedora

: Docker is still under heavy development! We don't recommend using it in production yet, but we're getting closer with each release. Please see our blog post, "Getting to Docker 1.0"

: This is a community contributed installation path. The only 'official' installation is using the *Ubuntu* installation path. This version may be out of date because it depends on some binaries to be updated and published

Docker is available in **Fedora 19 and later**. Please note that due to the current Docker limitations Docker is able to run only on the **64 bit** architecture.

2.3.1 Installation

The `docker-io` package provides Docker on Fedora.

If you have the (unrelated) `docker` package installed already, it will conflict with `docker-io`. There's a bug report filed for it. To proceed with `docker-io` installation on Fedora 19, please remove `docker` first.

```
sudo yum -y remove docker
```

For Fedora 20 and later, the `wmdocker` package will provide the same functionality as `docker` and will also not conflict with `docker-io`.

```
sudo yum -y install wmdocker
sudo yum -y remove docker
```

Install the `docker-io` package which will install Docker on our host.

```
sudo yum -y install docker-io
```

To update the `docker-io` package:

```
sudo yum -y update docker-io
```

Now that it's installed, let's start the Docker daemon.

```
sudo systemctl start docker
```

If we want Docker to start at boot, we should also:

```
sudo systemctl enable docker
```

Now let's verify that Docker is working.

```
sudo docker run -i -t fedora /bin/bash
```

Done!, now continue with the *Hello World* example.

2.4 Arch Linux

: Docker is still under heavy development! We don't recommend using it in production yet, but we're getting closer with each release. Please see our blog post, "Getting to Docker 1.0"

: This is a community contributed installation path. The only 'official' installation is using the *Ubuntu* installation path. This version may be out of date because it depends on some binaries to be updated and published

Installing on Arch Linux can be handled via the package in community:

- docker

or the following AUR package:

- docker-git

The docker package will install the latest tagged version of docker. The docker-git package will build from the current master branch.

2.4.1 Dependencies

Docker depends on several packages which are specified as dependencies in the packages. The core dependencies are:

- bridge-utils
- device-mapper
- iproute2
- lxc
- sqlite

2.4.2 Installation

For the normal package a simple

```
pacman -S docker
```

is all that is needed.

For the AUR package execute:

```
yaourt -S docker-git
```

The instructions here assume **yaourt** is installed. See Arch User Repository for information on building and installing packages from the AUR if you have not done so before.

2.4.3 Starting Docker

There is a systemd service unit created for docker. To start the docker service:

```
sudo systemctl start docker
```

To start on system boot:

```
sudo systemctl enable docker
```

2.5 Gentoo

: Docker is still under heavy development! We don't recommend using it in production yet, but we're getting closer with each release. Please see our blog post, "Getting to Docker 1.0"

: This is a community contributed installation path. The only 'official' installation is using the *Ubuntu* installation path. This version may be out of date because it depends on some binaries to be updated and published

Installing Docker on Gentoo Linux can be accomplished using one of two methods. The first and best way if you're looking for a stable experience is to use the official *app-emulation/docker* package directly in the portage tree.

If you're looking for a `-bin` ebuild, a live ebuild, or bleeding edge ebuild changes/fixes, the second installation method is to use the overlay provided at https://github.com/tianon/docker-overlay which can be added using `app-portage/layman`. The most accurate and up-to-date documentation for properly installing and using the overlay can be found in the overlay README.

Note that sometimes there is a disparity between the latest version and what's in the overlay, and between the latest version in the overlay and what's in the portage tree. Please be patient, and the latest version should propagate shortly.

2.5.1 Installation

The package should properly pull in all the necessary dependencies and prompt for all necessary kernel options. The ebuilds for 0.7+ include use flags to pull in the proper dependencies of the major storage drivers, with the "device-mapper" use flag being enabled by default, since that is the simplest installation path.

```
sudo emerge -av app-emulation/docker
```

If any issues arise from this ebuild or the resulting binary, including and especially missing kernel configuration flags and/or dependencies, open an issue on the docker-overlay repository or ping tianon directly in the #docker IRC channel on the freenode network.

2.5.2 Starting Docker

Ensure that you are running a kernel that includes all the necessary modules and/or configuration for LXC (and optionally for device-mapper and/or AUFS, depending on the storage driver you've decided to use).

OpenRC

To start the docker daemon:

```
sudo /etc/init.d/docker start
```

To start on system boot:

```
sudo rc-update add docker default
```

systemd

To start the docker daemon:

```
sudo systemctl start docker.service
```

To start on system boot:

```
sudo systemctl enable docker.service
```

2.6 openSUSE

: Docker is still under heavy development! We don't recommend using it in production yet, but we're getting closer with each release. Please see our blog post, "Getting to Docker 1.0"

: This is a community contributed installation path. The only 'official' installation is using the *Ubuntu* installation path. This version may be out of date because it depends on some binaries to be updated and published

Docker is available in **openSUSE 12.3 and later**. Please note that due to the current Docker limitations Docker is able to run only on the **64 bit** architecture.

2.6.1 Installation

The `docker` package from the Virtualization project on OBS provides Docker on openSUSE.

To proceed with Docker installation please add the right Virtualization repository.

```
# openSUSE 12.3
sudo zypper ar -f http://download.opensuse.org/repositories/Virtualization/openSUSE_12.3/ Virtualizat

# openSUSE 13.1
sudo zypper ar -f http://download.opensuse.org/repositories/Virtualization/openSUSE_13.1/ Virtualizat
```

Install the Docker package.

```
sudo zypper in docker
```

It's also possible to install Docker using openSUSE's 1-click install. Just visit this page, select your openSUSE version and click on the installation link. This will add the right repository to your system and it will also install the *docker* package.

Now that it's installed, let's start the Docker daemon.

```
sudo systemctl start docker
```

If we want Docker to start at boot, we should also:

```
sudo systemctl enable docker
```

The *docker* package creates a new group named *docker*. Users, other than *root* user, need to be part of this group in order to interact with the Docker daemon.

```
sudo usermod -G docker <username>
```

Done!, now continue with the *Hello World* example.

2.7 FrugalWare

: Docker is still under heavy development! We don't recommend using it in production yet, but we're getting closer with each release. Please see our blog post, "Getting to Docker 1.0"

: This is a community contributed installation path. The only 'official' installation is using the *Ubuntu* installation path. This version may be out of date because it depends on some binaries to be updated and published

Installing on FrugalWare is handled via the official packages:

- lxc-docker i686
- lxc-docker x86_64

The *lxc-docker* package will install the latest tagged version of Docker.

2.7.1 Dependencies

Docker depends on several packages which are specified as dependencies in the packages. The core dependencies are:

- systemd
- lvm2
- sqlite3
- libguestfs
- lxc
- iproute2
- bridge-utils

2.7.2 Installation

A simple

```
pacman -S lxc-docker
```

is all that is needed.

2.7.3 Starting Docker

There is a systemd service unit created for Docker. To start Docker as service:

```
sudo systemctl start lxc-docker
```

To start on system boot:

```
sudo systemctl enable lxc-docker
```

2.8 Mac OS X

: These instructions are available with the new release of Docker (version 0.8). However, they are subject to change.

: Docker is still under heavy development! We don't recommend using it in production yet, but we're getting closer with each release. Please see our blog post, "Getting to Docker 1.0"

Docker is supported on Mac OS X 10.6 "Snow Leopard" or newer.

2.8.1 How To Install Docker On Mac OS X

VirtualBox

Docker on OS X needs VirtualBox to run. To begin with, head over to VirtualBox Download Page and get the tool for `OS X hosts x86/amd64`.

Once the download is complete, open the disk image, run the set up file (i.e. `VirtualBox.pkg`) and install VirtualBox. Do not simply copy the package without running the installer.

boot2docker

boot2docker provides a handy script to easily manage the VM running the `docker` daemon. It also takes care of the installation for the OS image that is used for the job.

Open up a new terminal window, if you have not already.

Run the following commands to get boot2docker:

```
# Enter the installation directory
cd ~/bin

# Get the file
curl https://raw.github.com/steeve/boot2docker/master/boot2docker > boot2docker

# Mark it executable
chmod +x boot2docker
```

Docker OS X Client

The `docker` daemon is accessed using the `docker` client.

Run the following commands to get it downloaded and set up:

```
# Get the file
curl -o docker http://get.docker.io/builds/Darwin/x86_64/docker-latest

# Mark it executable
chmod +x docker

# Set the environment variable for the docker daemon
export DOCKER_HOST=tcp://

# Copy the executable file
sudo cp docker /usr/local/bin/
```

And that's it! Let's check out how to use it.

2.8.2 How To Use Docker On Mac OS X

The `docker` daemon (via boot2docker)

Inside the `~/bin` directory, run the following commands:

```
# Initiate the VM
./boot2docker init

# Run the VM (the docker daemon)
./boot2docker up

# To see all available commands:
./boot2docker

# Usage ./boot2docker {init|start|up|pause|stop|restart|status|info|delete|ssh|download}
```

The `docker` client

Once the VM with the `docker` daemon is up, you can use the `docker` client just like any other application.

```
docker version
# Client version: 0.7.6
# Go version (client): go1.2
# Git commit (client): bc3b2ec
# Server version: 0.7.5
# Git commit (server): c348c04
# Go version (server): go1.2
```

SSH-ing The VM

If you feel the need to connect to the VM, you can simply run:

```
./boot2docker ssh

# User: docker
# Pwd:  tcuser
```

You can now continue with the *Hello World* example.

2.8.3 Learn More

boot2docker:

See the GitHub page for boot2docker.

If SSH complains about keys:

```
ssh-keygen -R '[localhost]:2022'
```

About the way Docker works on Mac OS X:

Docker has two key components: the `docker` daemon and the `docker` client. The tool works by client commanding the daemon. In order to work and do its magic, the daemon makes use of some Linux Kernel features (e.g. LXC, name spaces etc.), which are not supported by OS X. Therefore, the solution of getting Docker to run on OS X consists of running it inside a lightweight virtual machine. In order to simplify things, Docker comes with a bash script to make this whole process as easy as possible (i.e. boot2docker).

2.9 WindowsDocker

Docker can run on Windows using a VM like VirtualBox. You then run Linux within the VM.

2.9.1

: Docker is still under heavy development! We don't recommend using it in production yet, but we're getting closer with each release. Please see our blog post, "Getting to Docker 1.0"

: This is a community contributed installation path. The only 'official' installation is using the *Ubuntu* installation path. This version may be out of date because it depends on some binaries to be updated and published

1. Install virtualbox from https://www.virtualbox.org - or follow this tutorial
2. Install vagrant from http://www.vagrantup.com - or follow this tutorial
3. Install git with ssh from http://git-scm.com/downloads - or follow this tutorial

We recommend having at least 2Gb of free disk space and 2Gb of RAM (or more).

2.9.2

First open a cmd prompt. Press Windows key and then press "R" key. This will open the RUN dialog box for you. Type "cmd" and press Enter. Or you can click on Start, type "cmd" in the "Search programs and files" field, and click on cmd.exe.

This should open a cmd prompt window.

Alternatively, you can also use a Cygwin terminal, or Git Bash (or any other command line program you are usually using). The next steps would be the same.

2.9.3 Ubuntu

Let's download and run an Ubuntu image with docker binaries already installed.

```
git clone https://github.com/dotcloud/docker.git
cd docker
vagrant up
```

Congratulations! You are running an Ubuntu server with docker installed on it. You do not see it though, because it is running in the background.

2.9.4 Unbuntu

Let's log into your Ubuntu server now. To do so you have two choices:

- Use Vagrant on Windows command prompt OR
- Use SSH

Using Vagrant on Windows Command Prompt

Run the following command

```
vagrant ssh
```

You may see an error message starting with "*ssh* executable not found". In this case it means that you do not have SSH in your PATH. If you do not have SSH in your PATH you can set it up with the "set" command. For instance, if your ssh.exe is in the folder named "C:Program Files (x86)Gitbin", then you can run the following command:

```
set PATH=%PATH%;C:\Program Files (x86)\Git\bin
```

SSH

First step is to get the IP and port of your Ubuntu server. Simply run:

```
vagrant ssh-config
```

You should see an output with HostName and Port information. In this example, HostName is 127.0.0.1 and port is 2222. And the User is "vagrant". The password is not shown, but it is also "vagrant".

You can now use this information for connecting via SSH to your server. To do so you can:

- Use putty.exe OR
- Use SSH from a terminal

putty.exe

You can download putty.exe from this page http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html Launch putty.exe and simply enter the information you got from last step.

Open, and enter user = vagrant and password = vagrant.

SSH

You can also run this command on your favorite terminal (windows prompt, cygwin, git-bash, . . .). Make sure to adapt the IP and port from what you got from the vagrant ssh-config command.

```
ssh vagrant@127.0.0.1 -p 2222
```

Enter user = vagrant and password = vagrant.

Congratulations, you are now logged onto your Ubuntu Server, running on top of your Windows machine !

2.9.5 Docker

First you have to be root in order to run docker. Simply run the following command:

```
sudo su
```

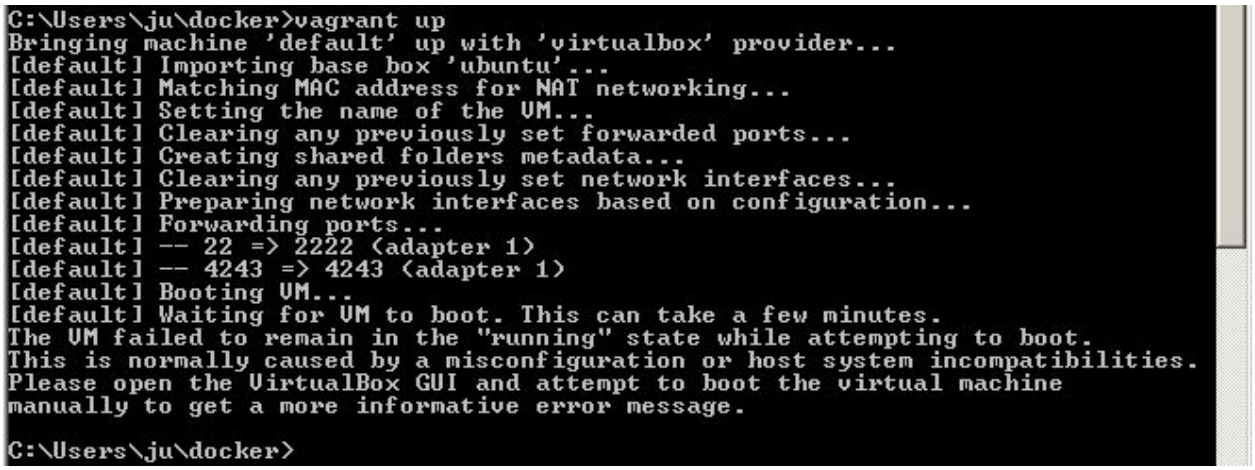
You are now ready for the docker's "hello world" example. Run

```
docker run busybox echo hello world
```

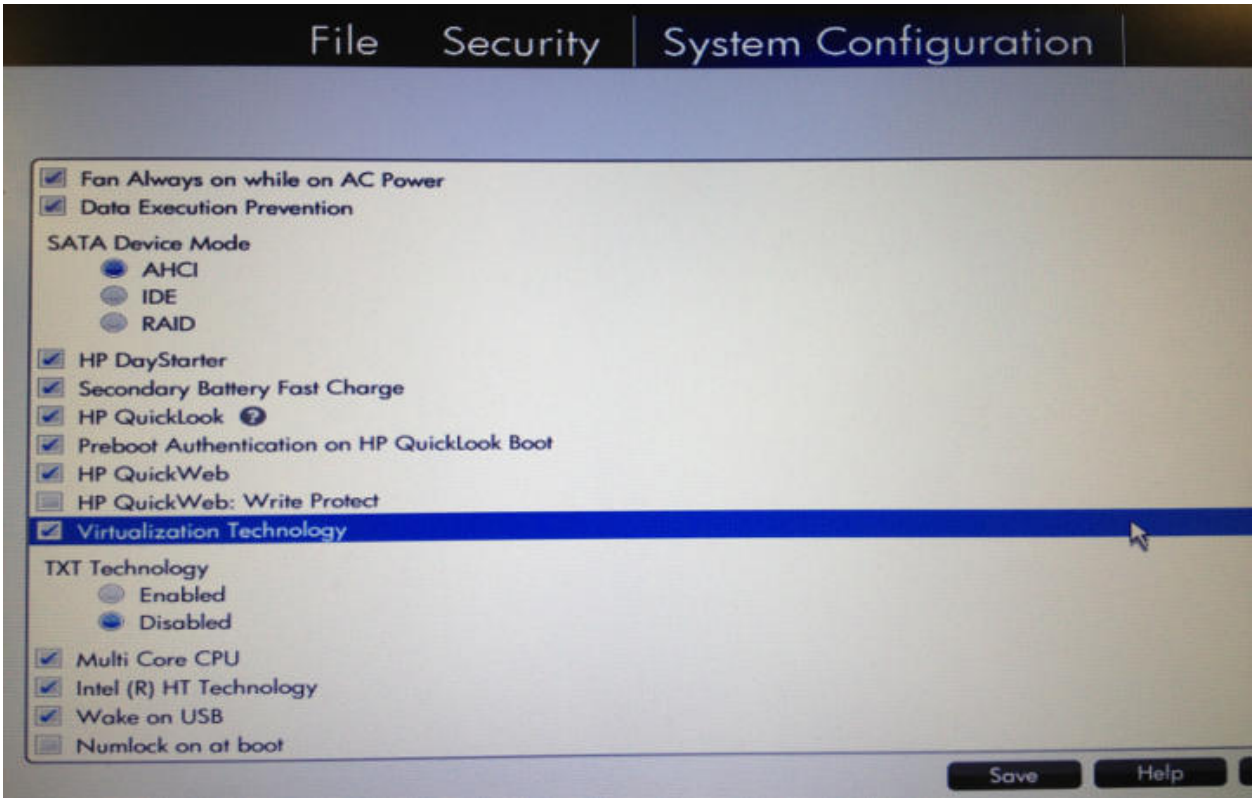
All done!

Now you can continue with the *Hello World* example.

2.9.6

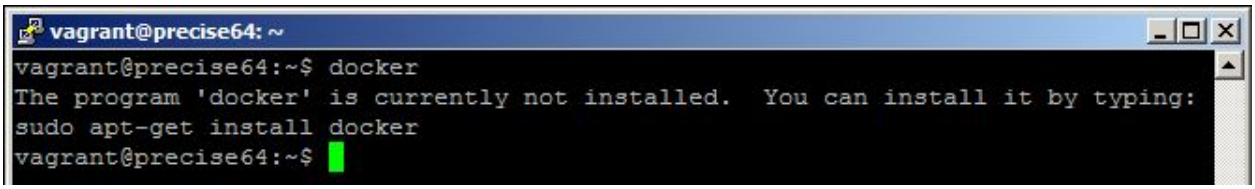


If you run into this error message "The VM failed to remain in the 'running' state while attempting to boot", please check that your computer has virtualization technology available and activated by going to the BIOS. Here's an example for an HP computer (System configuration / Device configuration)



On some machines the BIOS menu can only be accessed before startup. To access BIOS in this scenario you should restart your computer and press ESC/Enter when prompted to access the boot and BIOS controls. Typically the option to allow virtualization is contained within the BIOS/Security menu.

Docker



If you run into this error message "The program 'docker' is currently not installed", try deleting the docker folder and restart from *Ubuntu*

2.10 Amazon EC2

: Docker is still under heavy development! We don't recommend using it in production yet, but we're getting closer with each release. Please see our blog post, "Getting to Docker 1.0"

There are several ways to install Docker on AWS EC2:

- *Amazon QuickStart* or
- *Standard Ubuntu Installation* or
- *Use Vagrant*

You'll need an AWS account first, of course.

2.10.1 Amazon QuickStart

1. **Choose an image:**
   - Launch the Create Instance Wizard menu on your AWS Console.
   - Click the `Select` button for a 64Bit Ubuntu image. For example: Ubuntu Server 12.04.3 LTS
   - For testing you can use the default (possibly free) `t1.micro` instance (more info on pricing).
   - Click the `Next: Configure Instance Details` button at the bottom right.
2. **Tell CloudInit to install Docker:**
   - When you're on the "Configure Instance Details" step, expand the "Advanced Details" section.
   - Under "User data", select "As text".
   - Enter `#include https://get.docker.io` into the instance *User Data*. CloudInit is part of the Ubuntu image you chose; it will bootstrap Docker by running the shell script located at this URL.
3. After a few more standard choices where defaults are probably ok, your AWS Ubuntu instance with Docker should be running!

If this is your first AWS instance, you may need to set up your Security Group to allow SSH. By default all incoming ports to your new instance will be blocked by the AWS Security Group, so you might just get timeouts when you try to connect.

Installing with `get.docker.io` (as above) will create a service named `lxc-docker`. It will also set up a *docker group* and you may want to add the *ubuntu* user to it so that you don't have to use `sudo` for every Docker command.

Once you've got Docker installed, you're ready to try it out – head on over to the *Docker* or section.

2.10.2 Standard Ubuntu Installation

If you want a more hands-on installation, then you can follow the *Ubuntu* instructions installing Docker on any EC2 instance running Ubuntu. Just follow Step 1 from *Amazon QuickStart* to pick an image (or use one of your own) and skip the step with the *User Data*. Then continue with the *Ubuntu* instructions.

2.10.3 Use Vagrant

: This is a community contributed installation path. The only 'official' installation is using the *Ubuntu* installation path. This version may be out of date because it depends on some binaries to be updated and published

And finally, if you prefer to work through Vagrant, you can install Docker that way too. Vagrant 1.1 or higher is required.

1. Install vagrant from http://www.vagrantup.com/ (or use your package manager)
2. Install the vagrant aws plugin

   ```
   vagrant plugin install vagrant-aws
   ```

3. Get the docker sources, this will give you the latest Vagrantfile.

```
git clone https://github.com/dotcloud/docker.git
```

4. Check your AWS environment.

   Create a keypair specifically for EC2, give it a name and save it to your disk. *I usually store these in my ~/.ssh/ folder*.

   Check that your default security group has an inbound rule to accept SSH (port 22) connections.

5. Inform Vagrant of your settings

   Vagrant will read your access credentials from your environment, so we need to set them there first. Make sure you have everything on amazon aws setup so you can (manually) deploy a new image to EC2.

   Note that where possible these variables are the same as those honored by the ec2 api tools.

   ```
   export AWS_ACCESS_KEY=xxx
   export AWS_SECRET_KEY=xxx
   export AWS_KEYPAIR_NAME=xxx
   export SSH_PRIVKEY_PATH=xxx

   export BOX_NAME=xxx
   export AWS_REGION=xxx
   export AWS_AMI=xxx
   export AWS_INSTANCE_TYPE=xxx
   ```

   The required environment variables are:

   - `AWS_ACCESS_KEY` - The API key used to make requests to AWS
   - `AWS_SECRET_KEY` - The secret key to make AWS API requests
   - `AWS_KEYPAIR_NAME` - The name of the keypair used for this EC2 instance
   - `SSH_PRIVKEY_PATH` - The path to the private key for the named keypair, for example `~/.ssh/docker.pem`

   There are a number of optional environment variables:

   - `BOX_NAME` - The name of the vagrant box to use. Defaults to `ubuntu`.
   - `AWS_REGION` - The aws region to spawn the vm in. Defaults to `us-east-1`.
   - `AWS_AMI` - The aws AMI to start with as a base. This must be be an ubuntu 12.04 precise image. You must change this value if `AWS_REGION` is set to a value other than `us-east-1`. This is because AMIs are region specific. Defaults to `ami-69f5a900`.
   - `AWS_INSTANCE_TYPE` - The aws instance type. Defaults to `t1.micro`.

   You can check if they are set correctly by doing something like

   ```
   echo $AWS_ACCESS_KEY
   ```

6. Do the magic!

   ```
   vagrant up --provider=aws
   ```

   If it stalls indefinitely on `[default] Waiting for SSH to become available...`, Double check your default security zone on AWS includes rights to SSH (port 22) to your container.

   If you have an advanced AWS setup, you might want to have a look at vagrant-aws.

7. Connect to your machine

```
vagrant ssh
```

8. Your first command

   Now you are in the VM, run docker

   ```
   sudo docker
   ```

Continue with the *Hello World* example.

2.11 Rackspace Cloud

: This is a community contributed installation path. The only 'official' installation is using the *Ubuntu* installation path. This version may be out of date because it depends on some binaries to be updated and published

Installing Docker on Ubuntu provided by Rackspace is pretty straightforward, and you should mostly be able to follow the *Ubuntu* installation guide.

However, there is one caveat:

If you are using any Linux not already shipping with the 3.8 kernel you will need to install it. And this is a little more difficult on Rackspace.

Rackspace boots their servers using grub's `menu.lst` and does not like non 'virtual' packages (e.g. Xen compatible) kernels there, although they do work. This results in `update-grub` not having the expected result, and you will need to set the kernel manually.

Do not attempt this on a production machine!

```
# update apt
apt-get update

# install the new kernel
apt-get install linux-generic-lts-raring
```

Great, now you have the kernel installed in `/boot/`, next you need to make it boot next time.

```
# find the exact names
find /boot/ -name '*3.8*'

# this should return some results
```

Now you need to manually edit `/boot/grub/menu.lst`, you will find a section at the bottom with the existing options. Copy the top one and substitute the new kernel into that. Make sure the new kernel is on top, and double check the kernel and initrd lines point to the right files.

Take special care to double check the kernel and initrd entries.

```
# now edit /boot/grub/menu.lst
vi /boot/grub/menu.lst
```

It will probably look something like this:

```
## ## End Default Options ##

title              Ubuntu 12.04.2 LTS, kernel 3.8.x generic
root               (hd0)
kernel             /boot/vmlinuz-3.8.0-19-generic root=/dev/xvda1 ro quiet splash console=hvc0
```

```
initrd          /boot/initrd.img-3.8.0-19-generic

title           Ubuntu 12.04.2 LTS, kernel 3.2.0-38-virtual
root            (hd0)
kernel          /boot/vmlinuz-3.2.0-38-virtual root=/dev/xvda1 ro quiet splash console=hvc0
initrd          /boot/initrd.img-3.2.0-38-virtual

title           Ubuntu 12.04.2 LTS, kernel 3.2.0-38-virtual (recovery mode)
root            (hd0)
kernel          /boot/vmlinuz-3.2.0-38-virtual root=/dev/xvda1 ro quiet splash  single
initrd          /boot/initrd.img-3.2.0-38-virtual
```

Reboot the server (either via command line or console)

```
# reboot
```

Verify the kernel was updated

```
uname -a
# Linux docker-12-04 3.8.0-19-generic #30~precise1-Ubuntu SMP Wed May 1 22:26:36 UTC 2013 x86_64 x86_

# nice! 3.8.
```

Now you can finish with the *Ubuntu* instructions.

2.12 Google Cloud Platform

: Docker is still under heavy development! We don't recommend using it in production yet, but we're getting closer with each release. Please see our blog post, "Getting to Docker 1.0"

2.12.1 Compute Engine QuickStart for Debian

1. Go to Google Cloud Console and create a new Cloud Project with Compute Engine enabled.
2. Download and configure the Google Cloud SDK to use your project with the following commands:

```
$ curl https://dl.google.com/dl/cloudsdk/release/install_google_cloud_sdk.bash | bash
$ gcloud auth login
Enter a cloud project id (or leave blank to not set): <google-cloud-project-id>
```

3. Start a new instance, select a zone close to you and the desired instance size:

```
$ gcutil addinstance docker-playground --image=backports-debian-7
1: europe-west1-a
...
4: us-central1-b
>>> <zone-index>
1: machineTypes/n1-standard-1
...
12: machineTypes/g1-small
>>> <machine-type-index>
```

4. Connect to the instance using SSH:

```
$ gcutil ssh docker-playground
docker-playground:~$
```

5. Install the latest Docker release and configure it to start when the instance boots:

```
docker-playground:~$ curl get.docker.io | bash
docker-playground:~$ sudo update-rc.d docker defaults
```

6. Start a new container:

```
docker-playground:~$ sudo docker run busybox echo 'docker on GCE \o/'
docker on GCE \o/
```

2.13 Binaries

: Docker is still under heavy development! We don't recommend using it in production yet, but we're getting closer with each release. Please see our blog post, "Getting to Docker 1.0"

This instruction set is meant for hackers who want to try out Docker on a variety of environments.

Before following these directions, you should really check if a packaged version of Docker is already available for your distribution. We have packages for many distributions, and more keep showing up all the time!

2.13.1 Check runtime dependencies

To run properly, docker needs the following software to be installed at runtime:

- iproute2 version 3.5 or later (build after 2012-05-21), and specifically the "ip" utility
- iptables version 1.4 or later
- The LXC utility scripts (http://lxc.sourceforge.net) version 0.8 or later
- Git version 1.7 or later
- XZ Utils 4.9 or later

2.13.2 Check kernel dependencies

Docker in daemon mode has specific kernel requirements. For details, check your distribution in .

Note that Docker also has a client mode, which can run on virtually any linux kernel (it even builds on OSX!).

2.13.3 Get the docker binary:

```
wget https://get.docker.io/builds/Linux/x86_64/docker-latest -O docker
chmod +x docker
```

2.13.4 Run the docker daemon

```
# start the docker in daemon mode from the directory you unpacked
sudo ./docker -d &
```

2.13.5 Giving non-root access

The `docker` daemon always runs as the root user, and since Docker version 0.5.2, the `docker` daemon binds to a Unix socket instead of a TCP port. By default that Unix socket is owned by the user *root*, and so, by default, you can access it with `sudo`.

Starting in version 0.5.3, if you (or your Docker installer) create a Unix group called *docker* and add users to it, then the `docker` daemon will make the ownership of the Unix socket read/writable by the *docker* group when the daemon starts. The `docker` daemon must always run as the root user, but if you run the `docker` client as a user in the *docker* group then you don't need to add `sudo` to all the client commands.

: The *docker* group is root-equivalent.

2.13.6 Upgrades

To upgrade your manual installation of Docker, first kill the docker daemon:

```
killall docker
```

Then follow the regular installation steps.

2.13.7 Run your first container!

```
# check your docker version
sudo ./docker version

# run a container and open an interactive shell in the container
sudo ./docker run -i -t ubuntu /bin/bash
```

Continue with the *Hello World* example.

CHAPTER 3

Contents:

3.1 Docker

3.1.1 Docker

This guide assumes you have a working installation of Docker. To check your Docker install, run the following command:

```
# Check that you have a working install
docker info
```

If you get `docker: command not found` or something like `/var/lib/docker/repositories: permission denied` you may have an incomplete docker installation or insufficient privileges to access Docker on your machine.

Please refer to for installation instructions.

3.1.2

```
# Download an ubuntu image
sudo docker pull ubuntu
```

This will find the `ubuntu` image by name in the *Central Index* and download it from the top-level Central Repository to a local image cache.

: When the image has successfully downloaded, you will see a 12 character hash `539c0211cd76: Download complete` which is the short form of the image ID. These short image IDs are the first 12 characters of the full image ID - which can be found using `docker inspect` or `docker images -notrunc=true`

3.1.3 shell

```
# Run an interactive shell in the ubuntu image,
# allocate a tty, attach stdin and stdout
# To detach the tty without exiting the shell,
# use the escape sequence Ctrl-p + Ctrl-q
sudo docker run -i -t ubuntu /bin/bash
```

3.1.4 Docker/Unix

: Changing the default `docker` daemon binding to a TCP port or Unix *docker* user group will increase your security risks by allowing non-root users to potentially gain *root* access on the host (e.g. #1369). Make sure you control access to `docker`.

With `-H` it is possible to make the Docker daemon to listen on a specific IP and port. By default, it will listen on `unix:///var/run/docker.sock` to allow only local connections by the *root* user. You *could* set it to `0.0.0.0:4243` or a specific host IP to give access to everybody, but that is **not recommended** because then it is trivial for someone to gain root access to the host where the daemon is running.

Similarly, the Docker client can use `-H` to connect to a custom port.

`-H` accepts host and port assignment in the following format: `tcp://[host][:port]` or `unix://path`

For example:

- `tcp://host:4243` -> tcp connection on host:4243
- `unix://path/to/socket` -> unix socket located at `path/to/socket`

`-H`, when empty, will default to the same value as when no `-H` was passed in.

`-H` also accepts short form for TCP bindings: `host[:port]` or `:port`

```
# Run docker in daemon mode
sudo <path to>/docker -H 0.0.0.0:5555 -d &
# Download an ubuntu image
sudo docker -H :5555 pull ubuntu
```

You can use multiple `-H`, for example, if you want to listen on both TCP and a Unix socket

```
# Run docker in daemon mode
sudo <path to>/docker -H tcp://127.0.0.1:4243 -H unix:///var/run/docker.sock -d &
# Download an ubuntu image, use default Unix socket
sudo docker pull ubuntu
# OR use the TCP port
sudo docker -H tcp://127.0.0.1:4243 pull ubuntu
```

3.1.5

```
# Start a very useful long-running process
JOB=$(sudo docker run -d ubuntu /bin/sh -c "while true; do echo Hello world; sleep 1; done")

# Collect the output of the job so far
sudo docker logs $JOB

# Kill the job
sudo docker kill $JOB
```

3.1.6

```
sudo docker ps
```

3.1.7 Bind a service on a TCP port

```
# Bind port 4444 of this container, and tell netcat to listen on it
JOB=$(sudo docker run -d -p 4444 ubuntu:12.10 /bin/nc -l 4444)

# Which public port is NATed to my container?
PORT=$(sudo docker port $JOB 4444 | awk -F: '{ print $2 }')

# Connect to the public port
echo hello world | nc 127.0.0.1 $PORT

# Verify that the network connection worked
echo "Daemon received: $(sudo docker logs $JOB)"
```

3.1.8

Save your containers state to a container image, so the state can be re-used.

When you commit your container only the differences between the image the container was created from and the current state of the container will be stored (as a diff). See which images you already have using the `docker images` command.

```
# Commit your container to a new named image
sudo docker commit <container_id> <some_name>

# List your containers
sudo docker images
```

You now have a image state from which you can create new instances.

Read more about or continue to the complete *Command Line Help*

3.2

A *repository* is a hosted collection of tagged *images* that together create the file system for a container. The repository's name is a tag that indicates the provenance of the repository, i.e. who created it and where the original copy is located.

You can find one or more repositories hosted on a *registry*. There can be an implicit or explicit host name as part of the repository tag. The implicit registry is located at `index.docker.io`, the home of "top-level" repositories and the Central Index. This registry may also include public "user" repositories.

So Docker is not only a tool for creating and managing your own *containers* – **Docker is also a tool for sharing**. The Docker project provides a Central Registry to host public repositories, namespaced by user, and a Central Index which provides user authentication and search over all the public repositories. You can host your own Registry too! Docker acts as a client for these services via `docker search, pull, login` and `push`.

3.2.1

There are two types of public repositories: *top-level* repositories which are controlled by the Docker team, and *user* repositories created by individual contributors. Anyone can read from these repositories – they really help people get started quickly! You could also use if you need to keep control of who accesses your images, but we will only refer to public repositories in these examples.

- Top-level repositories can easily be recognized by **not** having a `/` (slash) in their name. These repositories can generally be trusted.

- User repositories always come in the form of <username>/<repo_name>. This is what your published images will look like if you push to the public Central Registry.
- Only the authenticated user can push to their *username* namespace on the Central Registry.
- User images are not checked, it is therefore up to you whether or not you trust the creator of this image.

3.2.2

You can search the Central Index online or by the CLI. Searching can find images by name, user name or description:

```
$ sudo docker help search
Usage: docker search NAME

Search the docker index for images

  -notrunc=false: Don't truncate output
$ sudo docker search centos
Found 25 results matching your query ("centos")
NAME                            DESCRIPTION
centos
slantview/centos-chef-solo      CentOS 6.4 with chef-solo.
...
```

There you can see two example results: `centos` and `slantview/centos-chef-solo`. The second result shows that it comes from the public repository of a user, `slantview/`, while the first result (`centos`) doesn't explicitly list a repository so it comes from the trusted Central Repository. The `/` character separates a user's repository and the image name.

Once you have found the image name, you can download it:

```
# sudo docker pull <value>
$ sudo docker pull centos
Pulling repository centos
539c0211cd76: Download complete
```

What can you do with that image? Check out the and, when you're ready with your own image, come back here to learn how to share it.

3.2.3

Anyone can pull public images from the Central Registry, but if you would like to share one of your own images, then you must register a unique user name first. You can create your username and login on the central Docker Index online, or by running

```
sudo docker login
```

This will prompt you for a username, which will become a public namespace for your public repositories.

If your username is available then `docker` will also prompt you to enter a password and your e-mail address. It will then automatically log you in. Now you're ready to commit and push your own images!

3.2.4

When you make changes to an existing image, those changes get saved to a container's file system. You can then promote that container to become an image by making a `commit`. In addition to converting the container to an image,

this is also your opportunity to name the image, specifically a name that includes your user name from the Central Docker Index (as you did a `login` above) and a meaningful name for the image.

```
# format is "sudo docker commit <container_id> <username>/<imagename>"
$ sudo docker commit $CONTAINER_ID myname/kickassapp
```

3.2.5

In order to push an image to its repository you need to have committed your container to a named image (see above)

Now you can commit this image to the repository designated by its name or tag.

```
# format is "docker push <username>/<repo_name>"
$ sudo docker push myname/kickassapp
```

3.2.6

Trusted Builds automate the building and updating of images from GitHub, directly on docker.io servers. It works by adding a commit hook to your selected repository, triggering a build and update when you push a commit.

1. Create a Docker Index account and login.
2. Link your GitHub account through the `Link Accounts` menu.
3. Configure a Trusted build.
4. Pick a GitHub project that has a `Dockerfile` that you want to build.
5. Pick the branch you want to build (the default is the `master` branch).
6. Give the Trusted Build a name.
7. Assign an optional Docker tag to the Build.
8. Specify where the `Dockerfile` is located. The default is `/`.

Once the Trusted Build is configured it will automatically trigger a build, and in a few minutes, if there are no errors, you will see your new trusted build on the Docker Index. It will will stay in sync with your GitHub repo until you deactivate the Trusted Build.

If you want to see the status of your Trusted Builds you can go to your Trusted Builds page on the Docker index, and it will show you the status of your builds, and the build history.

Once you've created a Trusted Build you can deactive or delete it. You cannot however push to a Trusted Build with the `docker push` command. You can only manage it by committing code to your GitHub repository.

You can create multiple Trusted Builds per repository and configure them to point to specific `Dockerfile`'s or Git branches.

3.2.7

Right now (version 0.6), private repositories are only possible by hosting your own registry. To push or pull to a repository on your own registry, you must prefix the tag with the address of the registry's host, like this:

```
# Tag to create a repository with the full registry location.
# The location (e.g. localhost.localdomain:5000) becomes
# a permanent part of the repository name
sudo docker tag 0u812deadbeef localhost.localdomain:5000/repo_name

# Push the new repository to its home location on localhost
sudo docker push localhost.localdomain:5000/repo_name
```

Once a repository has your registry's host name as part of the tag, you can push and pull it like any other repository, but it will **not** be searchable (or indexed at all) in the Central Index, and there will be no user name checking performed. Your registry will function completely independently from the Central Index.

:

Docker Blog: How to use your own registry

3.2.8

The authentication is stored in a json file, `.dockercfg` located in your home directory. It supports multiple registry urls.

`docker login` will create the "https://index.docker.io/v1/" key.

`docker login https://my-registry.com` will create the "https://my-registry.com" key.

For example:

```
{
      "https://index.docker.io/v1/": {
              "auth": "xXxXxXxXxXx=",
              "email": "email@example.com"
      },
      "https://my-registry.com": {
              "auth": "XxXxXxXxXxX=",
              "email": "email@my-registry.com"
      }
}
```

The `auth` field represents `base64(<username>:<password>)`

3.3

Interacting with a service is commonly done through a connection to a port. When this service runs inside a container, one can connect to the port after finding the IP address of the container as follows:

```
# Find IP address of container with ID <container_id>
docker inspect <container_id> | grep IPAddress | cut -d '"' -f 4
```

However, this IP address is local to the host system and the container port is not reachable by the outside world. Furthermore, even if the port is used locally, e.g. by another container, this method is tedious as the IP address of the container changes every time it starts.

Docker addresses these two problems and give a simple and robust way to access services running inside containers.

To allow non-local clients to reach the service running inside the container, Docker provide ways to bind the container port to an interface of the host system. To simplify communication between containers, Docker provides the linking mechanism.

3.3.1

To bind a port of the container to a specific interface of the host system, use the `-p` parameter of the `docker run` command:

```
# General syntax
docker run -p [([<host_interface>:[host_port]])|(<host_port>):]<container_port>[/udp] <image> <cmd>
```

When no host interface is provided, the port is bound to all available interfaces of the host machine (aka INADDR_ANY, or 0.0.0.0).When no host port is provided, one is dynamically allocated. The possible combinations of options for TCP port are the following:

```
# Bind TCP port 8080 of the container to TCP port 80 on 127.0.0.1 of the host machine.
docker run -p 127.0.0.1:80:8080 <image> <cmd>

# Bind TCP port 8080 of the container to a dynamically allocated TCP port on 127.0.0.1 of the host ma
docker run -p 127.0.0.1::8080 <image> <cmd>

# Bind TCP port 8080 of the container to TCP port 80 on all available interfaces of the host machine
docker run -p 80:8080 <image> <cmd>

# Bind TCP port 8080 of the container to a dynamically allocated TCP port on all available interfaces
docker run -p 8080 <image> <cmd>
```

UDP ports can also be bound by adding a trailing `/udp`. All the combinations described for TCP work. Here is only one example:

```
# Bind UDP port 5353 of the container to UDP port 53 on 127.0.0.1 of the host machine.
docker run -p 127.0.0.1:53:5353/udp <image> <cmd>
```

The command `docker port` lists the interface and port on the host machine bound to a given container port. It is useful when using dynamically allocated ports:

```
# Bind to a dynamically allocated port
docker run -p 127.0.0.1::8080 -name dyn-bound <image> <cmd>

# Lookup the actual port
docker port dyn-bound 8080
127.0.0.1:49160
```

3.3.2

Communication between two containers can also be established in a docker-specific way called linking.

To briefly present the concept of linking, let us consider two containers: `server`, containing the service, and `client`, accessing the service. Once `server` is running, `client` is started and links to server. Linking sets environment variables in `client` giving it some information about `server`. In this sense, linking is a method of service discovery.

Let us now get back to our topic of interest; communication between the two containers. We mentioned that the tricky part about this communication was that the IP address of `server` was not fixed. Therefore, some of the environment variables are going to be used to inform `client` about this IP address. This process called exposure, is possible because `client` is started after `server` has been started.

Here is a full example. On `server`, the port of interest is exposed. The exposure is done either through the `-expose` parameter to the `docker run` command, or the `EXPOSE` build command in a Dockerfile:

```
# Expose port 80
docker run -expose 80 -name server <image> <cmd>
```

The client then links to the server:

```
# Link
docker run -name client -link server:linked-server <image> <cmd>
```

client locally refers to server as linked-server. The following environment variables, among others, are available on client:

```
# The default protocol, ip, and port of the service running in the container
LINKED-SERVER_PORT=tcp://172.17.0.8:80

# A specific protocol, ip, and port of various services
LINKED-SERVER_PORT_80_TCP=tcp://172.17.0.8:80
LINKED-SERVER_PORT_80_TCP_PROTO=tcp
LINKED-SERVER_PORT_80_TCP_ADDR=172.17.0.8
LINKED-SERVER_PORT_80_TCP_PORT=80
```

This tells client that a service is running on port 80 of server and that server is accessible at the IP address 172.17.0.8

Note: Using the -p parameter also exposes the port..

3.4

Docker uses Linux bridge capabilities to provide network connectivity to containers. The docker0 bridge interface is managed by Docker for this purpose. When the Docker daemon starts it :

- creates the docker0 bridge if not present
- searches for an IP address range which doesn't overlap with an existing route
- picks an IP in the selected range
- assigns this IP to the docker0 bridge

```
# List host bridges
$ sudo brctl show
bridge      name    bridge id               STP enabled     interfaces
docker0             8000.000000000000       no

# Show docker0 IP address
$ sudo ifconfig docker0
docker0   Link encap:Ethernet  HWaddr xx:xx:xx:xx:xx:xx
     inet addr:172.17.42.1  Bcast:0.0.0.0  Mask:255.255.0.0
```

At runtime, a *specific kind of virtual interface* is given to each container which is then bonded to the docker0 bridge. Each container also receives a dedicated IP address from the same range as docker0. The docker0 IP address is used as the default gateway for the container.

```
# Run a container
$ sudo docker run -t -i -d base /bin/bash
52f811c5d3d69edddefc75aff5a4525fc8ba8bcfa1818132f9dc7d4f7c7e78b4

$ sudo brctl show
bridge      name    bridge id               STP enabled     interfaces
docker0             8000.fef213db5a66       no              vethQCDY1N
```

Above, docker0 acts as a bridge for the vethQCDY1N interface which is dedicated to the 52f811c5d3d6 container.

3.4.1 IP

Docker will try hard to find an IP range that is not used by the host. Even though it works for most cases, it's not bullet-proof and sometimes you need to have more control over the IP addressing scheme.

For this purpose, Docker allows you to manage the `docker0` bridge or your own one using the `-b=<bridgename>` parameter.

In this scenario:

- ensure Docker is stopped
- create your own bridge (`bridge0` for example)
- assign a specific IP to this bridge
- start Docker with the `-b=bridge0` parameter

```
# Stop Docker
$ sudo service docker stop

# Clean docker0 bridge and
# add your very own bridge0
$ sudo ifconfig docker0 down
$ sudo brctl addbr bridge0
$ sudo ifconfig bridge0 192.168.227.1 netmask 255.255.255.0

# Edit your Docker startup file
$ echo "DOCKER_OPTS=\"-b=bridge0\"" >> /etc/default/docker

# Start Docker
$ sudo service docker start

# Ensure bridge0 IP is not changed by Docker
$ sudo ifconfig bridge0
bridge0   Link encap:Ethernet  HWaddr xx:xx:xx:xx:xx:xx
          inet addr:192.168.227.1  Bcast:192.168.227.255  Mask:255.255.255.0

# Run a container
$ docker run -i -t base /bin/bash

# Container IP in the 192.168.227/24 range
root@261c272cd7d5:/# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr xx:xx:xx:xx:xx:xx
          inet addr:192.168.227.5  Bcast:192.168.227.255  Mask:255.255.255.0

# bridge0 IP as the default gateway
root@261c272cd7d5:/# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.227.1   0.0.0.0         UG    0      0        0 eth0
192.168.227.0   0.0.0.0         255.255.255.0   U     0      0        0 eth0

# hits CTRL+P then CTRL+Q to detach

# Display bridge info
$ sudo brctl show
bridge      name    bridge id               STP enabled     interfaces
bridge0             8000.fe7c2e0faebd       no              vethAQI2QT
```

3.4.2

The value of the Docker daemon's `icc` parameter determines whether containers can communicate with each other over the bridge network.

- The default, `-icc=true` allows containers to communicate with each other.
- `-icc=false` means containers are isolated from each other.

Docker uses `iptables` under the hood to either accept or drop communication between containers.

3.4.3 vethXXXX

Well. Things get complicated here.

The `vethXXXX` interface is the host side of a point-to-point link between the host and the corresponding container; the other side of the link is the container's `eth0` interface. This pair (host `vethXXX` and container `eth0`) are connected like a tube. Everything that comes in one side will come out the other side.

All the plumbing is delegated to Linux network capabilities (check the ip link command) and the namespaces infrastructure.

3.4.4

Jérôme Petazzoni has create `pipework` to connect together containers in arbitrarily complex scenarios : https://github.com/jpetazzo/pipework

3.5

You can use your Docker containers with process managers like `upstart`, `systemd` and `supervisor`.

3.5.1

If you want a process manager to manage your containers you will need to run the docker daemon with the `-r=false` so that docker will not automatically restart your containers when the host is restarted.

When you have finished setting up your image and are happy with your running container, you may want to use a process manager to manage it. When your run `docker start -a` docker will automatically attach to the process and forward all signals so that the process manager can detect when a container stops and correctly restart it.

Here are a few sample scripts for systemd and upstart to integrate with docker.

3.5.2

In this example we've already created a container to run Redis with an id of 0a7e070b698b. To create an upstart script for our container, we create a file named `/etc/init/redis.conf` and place the following into it:

```
description "Redis container"
author "Me"
start on filesystem and started docker
stop on runlevel [!2345]
respawn
script
```

```
  # Wait for docker to finish starting up first.
  FILE=/var/run/docker.sock
  while [ ! -e $FILE ] ; do
    inotifywait -t 2 -e create $(dirname $FILE)
  done
  /usr/bin/docker start -a 0a7e070b698b
end script
```

Next, we have to configure docker so that it's run with the option `-r=false`. Run the following command:

```
$ sudo sh -c "echo 'DOCKER_OPTS=\"-r=false\"' > /etc/default/docker"
```

3.5.3

```
[Unit]
Description=Redis container
Author=Me
After=docker.service

[Service]
Restart=always
ExecStart=/usr/bin/docker start -a 0a7e070b698b
ExecStop=/usr/bin/docker stop -t 2 0a7e070b698b

[Install]
WantedBy=local.target
```

3.6

v0.3.0 : Data volumes have been available since version 1 of the *Docker Remote API*

A *data volume* is a specially-designated directory within one or more containers that bypasses the *Union File System* to provide several useful features for persistent or shared data:

- **Data volumes can be shared and reused between containers.** This is the feature that makes data volumes so powerful. You can use it for anything from hot database upgrades to custom backup or replication tools. See the example below.
- **Changes to a data volume are made directly**, without the overhead of a copy-on-write mechanism. This is good for very large files.
- **Changes to a data volume will not be included at the next commit** because they are not recorded as regular filesystem changes in the top layer of the *Union File System*

Each container can have zero or more data volumes.

3.6.1

Using data volumes is as simple as adding a `-v` parameter to the `docker run` command. The `-v` parameter can be used more than once in order to create more volumes within the new container. To create a new container with two new volumes:

```
$ docker run -v /var/volume1 -v /var/volume2 busybox true
```

This command will create the new container with two new volumes that exits instantly (`true` is pretty much the smallest, simplest program that you can run). Once created you can mount its volumes in any other container using the `-volumes-from` option; irrespecive of whether the container is running or not.

Or, you can use the VOLUME instruction in a Dockerfile to add one or more new volumes to any container created from that image:

```
# BUILD-USING:        docker build -t data .
# RUN-USING:          docker run -name DATA data
FROM          busybox
VOLUME        ["/var/volume1", "/var/volume2"]
CMD           ["/usr/bin/true"]
```

If you have some persistent data that you want to share between containers, or want to use from non-persistent containers, its best to create a named Data Volume Container, and then to mount the data from it.

Create a named container with volumes to share (`/var/volume1` and `/var/volume2`):

```
$ docker run -v /var/volume1 -v /var/volume2 -name DATA busybox true
```

Then mount those data volumes into your application containers:

```
$ docker run -t -i -rm -volumes-from DATA -name client1 ubuntu bash
```

You can use multiple `-volumes-from` parameters to bring together multiple data volumes from multiple containers.

Interestingly, you can mount the volumes that came from the `DATA` container in yet another container via the `client1` middleman container:

```
$ docker run -t -i -rm -volumes-from client1 -name client2 ubuntu bash
```

This allows you to abstract the actual data source from users of that data, similar to *ambassador_pattern_linking*.

If you remove containers that mount volumes, including the initial DATA container, or the middleman, the volumes will not be deleted until there are no containers still referencing those volumes. This allows you to upgrade, or effectivly migrate data volumes between containers.

:

```
-v=[]: Create a bind mount with: [host-dir]:[container-dir]:[rw|ro].
```

If `host-dir` is missing from the command, then docker creates a new volume. If `host-dir` is present but points to a non-existent directory on the host, Docker will automatically create this directory and use it as the source of the bind-mount.

Note that this is not available from a Dockerfile due the portability and sharing purpose of it. The `host-dir` volumes are entirely host-dependent and might not work on any other machine.

For example:

```
sudo docker run -v /var/logs:/var/host_logs:ro ubuntu bash
```

The command above mounts the host directory `/var/logs` into the container with read only permissions as `/var/host_logs`.

v0.5.0 .

3.6.2

- Issue 2702: "lxc-start: Permission denied - failed to mount" could indicate a permissions problem with AppArmor. Please see the issue for a workaround.
- Issue 2528: the busybox container is used to make the resulting container as small and simple as possible - whenever you need to interact with the data in the volume you mount it into another container.

3.7

From version 0.6.5 you are now able to `name` a container and `link` it to another container by referring to its name. This will create a parent -> child relationship where the parent container can see selected information about its child.

3.7.1

v0.6.5 .

You can now name your container by using the `-name` flag. If no name is provided, Docker will automatically generate a name. You can see this name using the `docker ps` command.

```
# format is "sudo docker run -name <container_name> <image_name> <command>"
$ sudo docker run -name test ubuntu /bin/bash

# the flag "-a" Show all containers. Only running containers are shown by default.
$ sudo docker ps -a
CONTAINER ID        IMAGE                              COMMAND             CREATED             STATUS
2522602a0d99        ubuntu:12.04                       /bin/bash           14 seconds ago      Exit 0
```

3.7.2 docker

v0.6.5 .

Links allow containers to discover and securely communicate with each other by using the flag `-link name:alias`. Inter-container communication can be disabled with the daemon flag `-icc=false`. With this flag set to `false`, Container A cannot access Container B unless explicitly allowed via a link. This is a huge win for securing your containers. When two containers are linked together Docker creates a parent child relationship between the containers. The parent container will be able to access information via environment variables of the child such as name, exposed ports, IP and other selected environment variables.

When linking two containers Docker will use the exposed ports of the container to create a secure tunnel for the parent to access. If a database container only exposes port 8080 then the linked container will only be allowed to access port 8080 and nothing else if inter-container communication is set to false.

For example, there is an image called `crosbymichael/redis` that exposes the port 6379 and starts the Redis server. Let's name the container as `redis` based on that image and run it as daemon.

```
$ sudo docker run -d -name redis crosbymichael/redis
```

We can issue all the commands that you would expect using the name `redis`; start, stop, attach, using the name for our container. The name also allows us to link other containers into this one.

Next, we can start a new web application that has a dependency on Redis and apply a link to connect both containers. If you noticed when running our Redis server we did not use the `-p` flag to publish the Redis port to the host system. Redis exposed port 6379 and this is all we need to establish a link.

```
$ sudo docker run -t -i -link redis:db -name webapp ubuntu bash
```

When you specified `-link redis:db` you are telling Docker to link the container named `redis` into this new container with the alias `db`. Environment variables are prefixed with the alias so that the parent container can access network and environment information from the containers that are linked into it.

If we inspect the environment variables of the second container, we would see all the information about the child container.

```
$ root@4c01db0b339c:/# env

HOSTNAME=4c01db0b339c
DB_NAME=/webapp/db
TERM=xterm
DB_PORT=tcp://172.17.0.8:6379
DB_PORT_6379_TCP=tcp://172.17.0.8:6379
DB_PORT_6379_TCP_PROTO=tcp
DB_PORT_6379_TCP_ADDR=172.17.0.8
DB_PORT_6379_TCP_PORT=6379
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
SHLVL=1
HOME=/
container=lxc
_=/usr/bin/env
root@4c01db0b339c:/#
```

Accessing the network information along with the environment of the child container allows us to easily connect to the Redis service on the specific IP and port in the environment.

Running `docker ps` shows the 2 containers, and the `webapp/db` alias name for the redis container.

```
$ docker ps
CONTAINER ID        IMAGE                          COMMAND                CREATED             STATUS
4c01db0b339c        ubuntu:12.04                   bash                   17 seconds ago      Up 16 se
d7886598dbe2        crosbymichael/redis:latest     /redis-server --dir    33 minutes ago      Up 33 mi
```

3.8

Rather than hardcoding network links between a service consumer and provider, Docker encourages service portability.

eg, instead of

```
(consumer) --> (redis)
```

requiring you to restart the `consumer` to attach it to a different `redis` service, you can add ambassadors

```
(consumer) --> (redis-ambassador) --> (redis)
```

or

```
(consumer) --> (redis-ambassador) ---network---> (redis-ambassador) --> (redis)
```

When you need to rewire your consumer to talk to a different redis server, you can just restart the `redis-ambassador` container that the consumer is connected to.

This pattern also allows you to transparently move the redis server to a different docker host from the consumer.

Using the svendowideit/ambassador container, the link wiring is controlled entirely from the docker run parameters.

3.8.1

Start actual redis server on one Docker host

```
big-server $ docker run -d -name redis crosbymichael/redis
```

Then add an ambassador linked to the redis server, mapping a port to the outside world

```
big-server $ docker run -d -link redis:redis -name redis_ambassador -p 6379:6379 svendowideit/ambassa
```

On the other host, you can set up another ambassador setting environment variables for each remote port we want to proxy to the big-server

```
client-server $ docker run -d -name redis_ambassador -expose 6379 -e REDIS_PORT_6379_TCP=tcp://192.1
```

Then on the client-server host, you can use a redis client container to talk to the remote redis server, just by linking to the local redis ambassador.

```
client-server $ docker run -i -t -rm -link redis_ambassador:redis relateiq/redis-cli
redis 172.17.0.160:6379> ping
PONG
```

3.8.2

The following example shows what the svendowideit/ambassador container does automatically (with a tiny amount of sed)

On the docker host (192.168.1.52) that redis will run on:

```
# start actual redis server
$ docker run -d -name redis crosbymichael/redis

# get a redis-cli container for connection testing
$ docker pull relateiq/redis-cli

# test the redis server by talking to it directly
$ docker run -t -i -rm -link redis:redis relateiq/redis-cli
redis 172.17.0.136:6379> ping
PONG
^D

# add redis ambassador
$ docker run -t -i -link redis:redis -name redis_ambassador -p 6379:6379 busybox sh
```

in the redis_ambassador container, you can see the linked redis containers's env

```
$ env
REDIS_PORT=tcp://172.17.0.136:6379
REDIS_PORT_6379_TCP_ADDR=172.17.0.136
REDIS_NAME=/redis_ambassador/redis
HOSTNAME=19d7adf4705e
REDIS_PORT_6379_TCP_PORT=6379
HOME=/
REDIS_PORT_6379_TCP_PROTO=tcp
```

```
container=lxc
REDIS_PORT_6379_TCP=tcp://172.17.0.136:6379
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
```

This environment is used by the ambassador socat script to expose redis to the world (via the -p 6379:6379 port mapping)

```
$ docker rm redis_ambassador
$ sudo ./contrib/mkimage-unittest.sh
$ docker run -t -i -link redis:redis -name redis_ambassador -p 6379:6379 docker-ut sh

$ socat TCP4-LISTEN:6379,fork,reuseaddr TCP4:172.17.0.136:6379
```

then ping the redis server via the ambassador

Now goto a different server

```
$ sudo ./contrib/mkimage-unittest.sh
$ docker run -t -i  -expose 6379 -name redis_ambassador docker-ut sh

$ socat TCP4-LISTEN:6379,fork,reuseaddr TCP4:192.168.1.52:6379
```

and get the redis-cli image so we can talk over the ambassador bridge

```
$ docker pull relateiq/redis-cli
$ docker run -i -t -rm -link redis_ambassador:redis relateiq/redis-cli
redis 172.17.0.160:6379> ping
PONG
```

3.8.3 The svendowideit/ambassador Dockerfile

The `svendowideit/ambassador` image is a small busybox image with `socat` built in. When you start the container, it uses a small `sed` script to parse out the (possibly multiple) link environment variables to set up the port forwarding. On the remote host, you need to set the variable using the `-e` command line option.

`-expose 1234 -e REDIS_PORT_1234_TCP=tcp://192.168.1.52:6379` will forward the local `1234` port to the remote IP and port - in this case `192.168.1.52:6379`.

```
#
#
# first you need to build the docker-ut image
# using ./contrib/mkimage-unittest.sh
# then
#   docker build -t SvenDowideit/ambassador .
#   docker tag SvenDowideit/ambassador ambassador
# then to run it (on the host that has the real backend on it)
#   docker run -t -i -link redis:redis -name redis_ambassador -p 6379:6379 ambassador
# on the remote host, you can set up another ambassador
#    docker run -t -i -name redis_ambassador -expose 6379 sh

FROM    docker-ut
MAINTAINER      SvenDowideit@home.org.au


CMD     env | grep _TCP= | sed 's/.*_PORT_\([0-9]*\)_TCP=tcp:\/\/\(.*\):\(.*\)/socat TCP4-LISTEN:\1,
```

3.9 Puppet

: Please note this is a community contributed installation path. The only 'official' installation is using the *Ubuntu* installation path. This version may sometimes be out of date.

3.9.1

To use this guide you'll need a working installation of Puppet from Puppetlabs .

The module also currently uses the official PPA so only works with Ubuntu.

3.9.2

The module is available on the Puppet Forge and can be installed using the built-in module tool.

```
puppet module install garethr/docker
```

It can also be found on GitHub if you would rather download the source.

3.9.3

The module provides a puppet class for installing Docker and two defined types for managing images and containers.

```
include 'docker'
```

The next step is probably to install a Docker image. For this, we have a defined type which can be used like so:

```
docker::image { 'ubuntu': }
```

This is equivalent to running:

```
docker pull ubuntu
```

Note that it will only be downloaded if an image of that name does not already exist. This is downloading a large binary so on first run can take a while. For that reason this define turns off the default 5 minute timeout for the exec type. Note that you can also remove images you no longer need with:

```
docker::image { 'ubuntu':
  ensure => 'absent',
}
```

Now you have an image where you can run commands within a container managed by Docker.

```
docker::run { 'helloworld':
  image   => 'ubuntu',
  command => '/bin/sh -c "while true; do echo hello world; sleep 1; done"',
}
```

This is equivalent to running the following command, but under upstart:

```
docker run -d ubuntu /bin/sh -c "while true; do echo hello world; sleep 1; done"
```

Run also contains a number of optional parameters:

```
docker::run { 'helloworld':
  image        => 'ubuntu',
  command      => '/bin/sh -c "while true; do echo hello world; sleep 1; done"',
  ports        => ['4444', '4555'],
  volumes      => ['/var/lib/couchdb', '/var/log'],
  volumes_from => '6446ea52fbc9',
  memory_limit => 10485760, # bytes
  username     => 'example',
  hostname     => 'example.com',
  env          => ['FOO=BAR', 'FOO2=BAR2'],
  dns          => ['8.8.8.8', '8.8.4.4'],
}
```

Note that ports, env, dns and volumes can be set with either a single string or as above with an array of values.

Here are some examples of how to use Docker to create running processes, starting from a very simple *Hello World* and progressing to more substantial services like those which you might find in production.

4.1 Hello World

4.2 Docker

This guide assumes you have a working installation of Docker. To check your Docker install, run the following command:

```
# Check that you have a working install
docker info
```

If you get `docker: command not found` or something like `/var/lib/docker/repositories: permission denied` you may have an incomplete Docker installation or insufficient privileges to access docker on your machine.

Please refer to for installation instructions.

4.2.1 Hello World

:

- This example assumes you have Docker running in daemon mode. For more information please see *Docker*.
- **If you don't like sudo** then see *Giving non-root access*

This is the most basic example available for using Docker.

Download the base image which is named `ubuntu`:

```
# Download an ubuntu image
sudo docker pull ubuntu
```

Alternatively to the `ubuntu` image, you can select `busybox`, a bare minimal Linux system. The images are retrieved from the Docker repository.

```
sudo docker run ubuntu /bin/echo hello world
```

This command will run a simple `echo` command, that will echo `hello world` back to the console over standard out.

Explanation:

- **"sudo"** execute the following commands as user *root*
- **"docker run"** run a command in a new container
- **"ubuntu"** is the image we want to run the command inside of.
- **"/bin/echo"** is the command we want to run in the container
- **"hello world"** is the input for the echo command

Video:

See the example in action

4.2.2 Hello World Daemon

:

- This example assumes you have Docker running in daemon mode. For more information please see *Docker*.
- **If you don't like sudo** then see *Giving non-root access*

And now for the most boring daemon ever written!

We will use the Ubuntu image to run a simple hello world daemon that will just print hello world to standard out every second. It will continue to do this until we stop it.

Steps:

```
CONTAINER_ID=$(sudo docker run -d ubuntu /bin/sh -c "while true; do echo hello world; sleep 1; done")
```

We are going to run a simple hello world daemon in a new container made from the `ubuntu` image.

- **"sudo docker run -d "** run a command in a new container. We pass "-d" so it runs as a daemon.
- **"ubuntu"** is the image we want to run the command inside of.
- **"/bin/sh -c"** is the command we want to run in the container
- **"while true; do echo hello world; sleep 1; done"** is the mini script we want to run, that will just print hello world once a second until we stop it.
- **$CONTAINER_ID** the output of the run command will return a container id, we can use in future commands to see what is going on with this process.

```
sudo docker logs $CONTAINER_ID
```

Check the logs make sure it is working correctly.

- **"docker logs**" This will return the logs for a container
- **$CONTAINER_ID** The Id of the container we want the logs for.

```
sudo docker attach -sig-proxy=false $CONTAINER_ID
```

Attach to the container to see the results in real-time.

- **“docker attach**” This will allow us to attach to a background process to see what is going on.
- **“-sig-proxy=false”** Do not forward signals to the container; allows us to exit the attachment using Control-C without stopping the container.
- **$CONTAINER_ID** The Id of the container we want to attach too.

Exit from the container attachment by pressing Control-C.

```
sudo docker ps
```

Check the process list to make sure it is running.

- **“docker ps”** this shows all running process managed by docker

```
sudo docker stop $CONTAINER_ID
```

Stop the container, since we don’t need it anymore.

- **“docker stop”** This stops a container
- **$CONTAINER_ID** The Id of the container we want to stop.

```
sudo docker ps
```

Make sure it is really stopped.

Video:

See the example in action

The next example in the series is a *Python Web* example, or you could skip to any of the other examples:

- *Python Web*
- *Node.js Web*
- *Redis*
- *SSH*
- *CouchDB*
- *PostgreSQL*
- *MongoDB*

4.3 Python Web

:

- This example assumes you have Docker running in daemon mode. For more information please see *Docker*.
- **If you don’t like sudo** then see *Giving non-root access*

The goal of this example is to show you how you can author your own Docker images using a parent image, making changes to it, and then saving the results as a new image. We will do that by making a simple hello Flask web application image.

Steps:

```
sudo docker pull shykes/pybuilder
```

We are downloading the shykes/pybuilder Docker image

```
URL=http://github.com/shykes/helloflask/archive/master.tar.gz
```

We set a URL variable that points to a tarball of a simple helloflask web app

```
BUILD_JOB=$(sudo docker run -d -t shykes/pybuilder:latest /usr/local/bin/buildapp $URL)
```

Inside of the shykes/pybuilder image there is a command called buildapp, we are running that command and passing the $URL variable from step 2 to it, and running the whole thing inside of a new container. The BUILD_JOB environment variable will be set with the new container ID.

```
sudo docker attach -sig-proxy=false $BUILD_JOB
[...]
```

While this container is running, we can attach to the new container to see what is going on. The flag --sig-proxy set as false allows you to connect and disconnect (Ctrl-C) to it without stopping the container.

```
sudo docker ps -a
```

List all Docker containers. If this container has already finished running, it will still be listed here.

```
BUILD_IMG=$(sudo docker commit $BUILD_JOB _/builds/github.com/shykes/helloflask/master)
```

Save the changes we just made in the container to a new image called _/builds/github.com/hykes/helloflask/master and save the image ID in the BUILD_IMG variable name.

```
WEB_WORKER=$(sudo docker run -d -p 5000 $BUILD_IMG /usr/local/bin/runapp)
```

- **"docker run -d "** run a command in a new container. We pass "-d" so it runs as a daemon.
- **"-p 5000"** the web app is going to listen on this port, so it must be mapped from the container to the host system.
- **"$BUILD_IMG"** is the image we want to run the command inside of.
- **/usr/local/bin/runapp** is the command which starts the web app.

Use the new image we just created and create a new container with network port 5000, and return the container ID and store in the WEB_WORKER variable.

```
sudo docker logs $WEB_WORKER
 * Running on http://0.0.0.0:5000/
```

View the logs for the new container using the WEB_WORKER variable, and if everything worked as planned you should see the line Running on http://0.0.0.0:5000/ in the log output.

```
WEB_PORT=$(sudo docker port $WEB_WORKER 5000 | awk -F: '{ print $2 }')
```

Look up the public-facing port which is NAT-ed. Find the private port used by the container and store it inside of the WEB_PORT variable.

```
# install curl if necessary, then ...
curl http://127.0.0.1:$WEB_PORT
  Hello world!
```

Access the web app using the curl binary. If everything worked as planned you should see the line Hello world! inside of your console.

Video:

See the example in action

Continue to *SSH*.

4.4 Node.js Web

:

- This example assumes you have Docker running in daemon mode. For more information please see *Docker*.
- **If you don't like sudo** then see *Giving non-root access*

The goal of this example is to show you how you can build your own Docker images from a parent image using a `Dockerfile` . We will do that by making a simple Node.js hello world web application running on CentOS. You can get the full source code at https://github.com/gasi/docker-node-hello.

4.4.1 Node.js

First, create a `package.json` file that describes your app and its dependencies:

```
{
  "name": "docker-centos-hello",
  "private": true,
  "version": "0.0.1",
  "description": "Node.js Hello World app on CentOS using docker",
  "author": "Daniel Gasienica <daniel@gasienica.ch>",
  "dependencies": {
    "express": "3.2.4"
  }
}
```

Then, create an `index.js` file that defines a web app using the Express.js framework:

```
var express = require('express');

// Constants
var PORT = 8080;

// App
var app = express();
app.get('/', function (req, res) {
  res.send('Hello World\n');
});

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

In the next steps, we'll look at how you can run this app inside a CentOS container using Docker. First, you'll need to build a Docker image of your app.

4.4.2 `Dockerfile`

Create an empty file called `Dockerfile`:

```
touch Dockerfile
```

Open the `Dockerfile` in your favorite text editor and add the following line that defines the version of Docker the image requires to build (this example uses Docker 0.3.4):

```
# DOCKER-VERSION 0.3.4
```

Next, define the parent image you want to use to build your own image on top of. Here, we'll use CentOS (tag: `6.4`) available on the Docker index:

```
FROM    centos:6.4
```

Since we're building a Node.js app, you'll have to install Node.js as well as npm on your CentOS image. Node.js is required to run your app and npm to install your app's dependencies defined in `package.json`. To install the right package for CentOS, we'll use the instructions from the Node.js wiki:

```
# Enable EPEL for Node.js
RUN     rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm
# Install Node.js and npm
RUN     yum install -y npm
```

To bundle your app's source code inside the Docker image, use the `ADD` instruction:

```
# Bundle app source
ADD . /src
```

Install your app dependencies using the `npm` binary:

```
# Install app dependencies
RUN cd /src; npm install
```

Your app binds to port `8080` so you'll use the `EXPOSE` instruction to have it mapped by the `docker` daemon:

```
EXPOSE  8080
```

Last but not least, define the command to run your app using `CMD` which defines your runtime, i.e. `node`, and the path to our app, i.e. `src/index.js` (see the step where we added the source to the container):

```
CMD ["node", "/src/index.js"]
```

Your `Dockerfile` should now look like this:

```
# DOCKER-VERSION 0.3.4
FROM    centos:6.4

# Enable EPEL for Node.js
RUN     rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm
# Install Node.js and npm
RUN     yum install -y npm

# Bundle app source
ADD . /src
# Install app dependencies
RUN cd /src; npm install

EXPOSE  8080
CMD ["node", "/src/index.js"]
```

4.4.3

Go to the directory that has your `Dockerfile` and run the following command to build a Docker image. The `-t` flag let's you tag your image so it's easier to find later using the `docker images` command:

```
sudo docker build -t <your username>/centos-node-hello .
```

Your image will now be listed by Docker:

```
sudo docker images
```

```
> # Example
> REPOSITORY                TAG       ID              CREATED
> centos                    6.4       539c0211cd76    8 weeks ago
> gasi/centos-node-hello    latest    d64d3505b0d2    2 hours ago
```

4.4.4

Running your image with `-d` runs the container in detached mode, leaving the container running in the background. The `-p` flag redirects a public port to a private port in the container. Run the image you previously built:

```
sudo docker run -p 49160:8080 -d <your username>/centos-node-hello
```

Print the output of your app:

```
# Get container ID
sudo docker ps

# Print app output
sudo docker logs <container id>

> # Example
> Running on http://localhost:8080
```

4.4.5

To test your app, get the the port of your app that Docker mapped:

```
sudo docker ps
```

```
> # Example
> ID            IMAGE                          COMMAND               ...   PORTS
> ecce33b30ebf  gasi/centos-node-hello:latest  node /src/index.js          49160->8080
```

In the example above, Docker mapped the `8080` port of the container to `49160`.

Now you can call your app using `curl` (install if needed via: `sudo apt-get install 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, 02 Jun 2013 03:53:22 GMT
> Connection: keep-alive
>
> Hello World
```

We hope this tutorial helped you get up and running with Node.js and CentOS on Docker. You can get the full source code at https://github.com/gasi/docker-node-hello.

Continue to *Redis*.

4.5 Redis

:

- This example assumes you have Docker running in daemon mode. For more information please see *Docker*.
- **If you don't like sudo** then see *Giving non-root access*

Very simple, no frills, Redis service attached to a web application using a link.

4.5.1 Create a docker container for Redis

Firstly, we create a `Dockerfile` for our new Redis image.

```
FROM        ubuntu:12.10
RUN         apt-get update
RUN         apt-get -y install redis-server
EXPOSE      6379
ENTRYPOINT  ["/usr/bin/redis-server"]
```

Next we build an image from our `Dockerfile`. Replace `<your username>` with your own user name.

```
sudo docker build -t <your username>/redis .
```

4.5.2 Run the service

Use the image we've just created and name your container `redis`.

Running the service with `-d` runs the container in detached mode, leaving the container running in the background.

Importantly, we're not exposing any ports on our container. Instead we're going to use a container link to provide access to our Redis database.

```
sudo docker run --name redis -d <your username>/redis
```

4.5.3 Create your web application container

Next we can create a container for our application. We're going to use the `-link` flag to create a link to the `redis` container we've just created with an alias of `db`. This will create a secure tunnel to the `redis` container and expose the Redis instance running inside that container to only this container.

```
sudo docker run --link redis:db -i -t ubuntu:12.10 /bin/bash
```

Once inside our freshly created container we need to install Redis to get the `redis-cli` binary to test our connection.

```
apt-get update
apt-get -y install redis-server
service redis-server stop
```

Now we can test the connection. Firstly, let's look at the available environmental variables in our web application container. We can use these to get the IP and port of our `redis` container.

```
env
. . .
DB_NAME=/violet_wolf/db
```

```
DB_PORT_6379_TCP_PORT=6379
DB_PORT=tcp://172.17.0.33:6379
DB_PORT_6379_TCP=tcp://172.17.0.33:6379
DB_PORT_6379_TCP_ADDR=172.17.0.33
DB_PORT_6379_TCP_PROTO=tcp
```

We can see that we've got a small list of environment variables prefixed with `DB`. The `DB` comes from the link alias specified when we launched the container. Let's use the `DB_PORT_6379_TCP_ADDR` variable to connect to our Redis container.

```
redis-cli -h $DB_PORT_6379_TCP_ADDR
redis 172.17.0.33:6379>
redis 172.17.0.33:6379> set docker awesome
OK
redis 172.17.0.33:6379> get docker
"awesome"
redis 172.17.0.33:6379> exit
```

We could easily use this or other environment variables in our web application to make a connection to our `redis` container.

4.6 SSH

:

- This example assumes you have Docker running in daemon mode. For more information please see *Docker*.
- **If you don't like sudo** then see *Giving non-root access*

Video:

I've created a little screencast to show how to create an SSHd service and connect to it. It is something like 11 minutes and not entirely smooth, but it gives you a good idea.

: This screencast was created before Docker version 0.5.2, so the daemon is unprotected and available via a TCP port. When you run through the same steps in a newer version of Docker, you will need to add `sudo` in front of each `docker` command in order to reach the daemon over its protected Unix socket.

You can also get this sshd container by using:

```
sudo docker pull dhrp/sshd
```

The password is `screencast`.

Video's Transcription:

```
# Hello! We are going to try and install openssh on a container and run it as a service
# let's pull ubuntu to get a base ubuntu image.
$ docker pull ubuntu
# I had it so it was quick
# now let's connect using -i for interactive and with -t for terminal
# we execute /bin/bash to get a prompt.
$ docker run -i -t ubuntu /bin/bash
# yes! we are in!
# now lets install openssh
$ apt-get update
$ apt-get install openssh-server
```

```
# ok. lets see if we can run it.
$ which sshd
# we need to create privilege separation directory
$ mkdir /var/run/sshd
$ /usr/sbin/sshd
$ exit
# now let's commit it
# which container was it?
$ docker ps -a |more
$ docker commit a30a3a2f2b130749995f5902f079dc6ad31ea0621fac595128ec59c6da07feea dhrp/sshd
# I gave the name dhrp/sshd for the container
# now we can run it again
$ docker run -d dhrp/sshd /usr/sbin/sshd -D # D for daemon mode
# is it running?
$ docker ps
# yes!
# let's stop it
$ docker stop 0ebf7cec294755399d063f4b1627980d4cbff7d999f0bc82b59c300f8536a562
$ docker ps
# and reconnect, but now open a port to it
$ docker run -d -p 22 dhrp/sshd /usr/sbin/sshd -D
$ docker port b2b407cf22cf8e7fa3736fa8852713571074536b1d31def3fdfcd9fa4fd8c8c5 22
# it has now given us a port to connect to
# we have to connect using a public ip of our host
$ hostname
# *ifconfig* is deprecated, better use *ip addr show* now
$ ifconfig
$ ssh root@192.168.33.10 -p 49153
# Ah! forgot to set root passwd
$ docker commit b2b407cf22cf8e7fa3736fa8852713571074536b1d31def3fdfcd9fa4fd8c8c5 dhrp/sshd
$ docker ps -a
$ docker run -i -t dhrp/sshd /bin/bash
$ passwd
$ exit
$ docker commit 9e863f0ca0af31c8b951048ba87641d67c382d08d655c2e4879c51410e0fedc1 dhrp/sshd
$ docker run -d -p 22 dhrp/sshd /usr/sbin/sshd -D
$ docker port a0aaa9558c90cf5c7782648df904a82365ebacce523e4acc085ac1213bfe2206 22
# *ifconfig* is deprecated, better use *ip addr show* now
$ ifconfig
$ ssh root@192.168.33.10 -p 49154
# Thanks for watching, Thatcher thatcher@dotcloud.com
```

4.6.1 :

For Ubuntu 13.10 using stackbrew/ubuntu, you may need do these additional steps:

1. change /etc/pam.d/sshd, pam_loginuid line 'required' to 'optional'
2. echo LANG="en_US.UTF-8" > /etc/default/locale

4.7 CouchDB

:

- This example assumes you have Docker running in daemon mode. For more information please see *Docker*.

- **If you don't like sudo** then see *Giving non-root access*

Here's an example of using data volumes to share the same data between two CouchDB containers. This could be used for hot upgrades, testing different versions of CouchDB on the same data, etc.

4.7.1

Note that we're marking `/var/lib/couchdb` as a data volume.

```
COUCH1=$(sudo docker run -d -p 5984 -v /var/lib/couchdb shykes/couchdb:2013-05-03)
```

4.7.2

We're assuming your Docker host is reachable at `localhost`. If not, replace `localhost` with the public IP of your Docker host.

```
HOST=localhost
URL="http://$HOST:$(sudo docker port $COUCH1 5984 | grep -Po '\d+$')/_utils/"
echo "Navigate to $URL in your browser, and use the couch interface to add data"
```

4.7.3

This time, we're requesting shared access to `$COUCH1`'s volumes.

```
COUCH2=$(sudo docker run -d -p 5984 --volumes-from $COUCH1 shykes/couchdb:2013-05-03)
```

4.7.4

```
HOST=localhost
URL="http://$HOST:$(sudo docker port $COUCH2 5984 | grep -Po '\d+$')/_utils/"
echo "Navigate to $URL in your browser. You should see the same data as in the first database"'!'
```

Congratulations, you are now running two Couchdb containers, completely isolated from each other *except* for their data.

4.8 PostgreSQL

:

- This example assumes you have Docker running in daemon mode. For more information please see *Docker*.
- **If you don't like sudo** then see *Giving non-root access*

: A shorter version of this blog post.

4.8.1 Installing PostgreSQL on Docker

Run an interactive shell in a Docker container.

```
sudo docker run -i -t ubuntu /bin/bash
```

Update its dependencies.

```
apt-get update
```

Install `python-software-properties`, `software-properties-common`, `wget` and `vim`.

```
apt-get -y install python-software-properties software-properties-common wget vim
```

Add PostgreSQL's repository. It contains the most recent stable release of PostgreSQL, `9.3`.

```
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | apt-key add -
echo "deb http://apt.postgresql.org/pub/repos/apt/ precise-pgdg main" > /etc/apt/sources.list.d/pgdg
apt-get update
```

Finally, install PostgreSQL 9.3

```
apt-get -y install postgresql-9.3 postgresql-client-9.3 postgresql-contrib-9.3
```

Now, create a PostgreSQL superuser role that can create databases and other roles. Following Vagrant's convention the role will be named `docker` with `docker` password assigned to it.

```
su postgres -c "createuser -P -d -r -s docker"
```

Create a test database also named `docker` owned by previously created `docker` role.

```
su postgres -c "createdb -O docker docker"
```

Adjust PostgreSQL configuration so that remote connections to the database are possible. Make sure that inside `/etc/postgresql/9.3/main/pg_hba.conf` you have following line:

```
host    all             all             0.0.0.0/0               md5
```

Additionaly, inside `/etc/postgresql/9.3/main/postgresql.conf` uncomment `listen_addresses` like so:

```
listen_addresses='*'
```

: This PostgreSQL setup is for development only purposes. Refer to PostgreSQL documentation how to fine-tune these settings so that it is secure enough.

Exit.

```
exit
```

Create an image from our container and assign it a name. The `<container_id>` is in the Bash prompt; you can also locate it using `docker ps -a`.

```
sudo docker commit <container_id> <your username>/postgresql
```

Finally, run the PostgreSQL server via `docker`.

```
CONTAINER=$(sudo docker run -d -p 5432 \
  -t <your username>/postgresql \
  /bin/su postgres -c '/usr/lib/postgresql/9.3/bin/postgres \
```

```
    -D /var/lib/postgresql/9.3/main \
    -c config_file=/etc/postgresql/9.3/main/postgresql.conf')
```

Connect the PostgreSQL server using `psql` (You will need the postgresql client installed on the machine. For ubuntu, use something like `sudo apt-get install postgresql-client`).

```
CONTAINER_IP=$(sudo docker inspect -format='{{.NetworkSettings.IPAddress}}' $CONTAINER)
psql -h $CONTAINER_IP -p 5432 -d docker -U docker -W
```

As before, create roles or databases if needed.

```
psql (9.3.1)
Type "help" for help.

docker=# CREATE DATABASE foo OWNER=docker;
CREATE DATABASE
```

Additionally, publish your newly created image on the Docker Index.

```
sudo docker login
Username: <your username>
[...]

sudo docker push <your username>/postgresql
```

4.8.2 PostgreSQL service auto-launch

Running our image seems complicated. We have to specify the whole command with `docker run`. Let's simplify it so the service starts automatically when the container starts.

```
sudo docker commit -run='{"Cmd": \
  ["/bin/su", "postgres", "-c", "/usr/lib/postgresql/9.3/bin/postgres -D \
  /var/lib/postgresql/9.3/main -c \
  config_file=/etc/postgresql/9.3/main/postgresql.conf"], "PortSpecs": ["5432"]}' \
  <container_id> <your username>/postgresql
```

From now on, just type `docker run <your username>/postgresql` and PostgreSQL should automatically start.

4.9 MongoDB

:

- This example assumes you have Docker running in daemon mode. For more information please see *Docker*.
- **If you don't like sudo** then see *Giving non-root access*

The goal of this example is to show how you can build your own Docker images with MongoDB pre-installed. We will do that by constructing a `Dockerfile` that downloads a base image, adds an apt source and installs the database software on Ubuntu.

4.9.1 `Dockerfile`

Create an empty file called `Dockerfile`:

```
touch Dockerfile
```

Next, define the parent image you want to use to build your own image on top of. Here, we'll use Ubuntu (tag: `latest`) available on the docker index:

```
FROM    ubuntu:latest
```

Since we want to be running the latest version of MongoDB we'll need to add the 10gen repo to our apt sources list.

```
# Add 10gen official apt source to the sources list
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | tee /etc/apt/sou
```

Then, we don't want Ubuntu to complain about init not being available so we'll divert `/sbin/initctl` to `/bin/true` so it thinks everything is working.

```
# Hack for initctl not being available in Ubuntu
RUN dpkg-divert --local --rename --add /sbin/initctl
RUN ln -s /bin/true /sbin/initctl
```

Afterwards we'll be able to update our apt repositories and install MongoDB

```
# Install MongoDB
RUN apt-get update
RUN apt-get install mongodb-10gen
```

To run MongoDB we'll have to create the default data directory (because we want it to run without needing to provide a special configuration file)

```
# Create the MongoDB data directory
RUN mkdir -p /data/db
```

Finally, we'll expose the standard port that MongoDB runs on, 27107, as well as define an `ENTRYPOINT` instruction for the container.

```
EXPOSE 27017
ENTRYPOINT ["usr/bin/mongod"]
```

Now, lets build the image which will go through the `Dockerfile` we made and run all of the commands.

```
sudo docker build -t <yourname>/mongodb .
```

Now you should be able to run `mongod` as a daemon and be able to connect on the local port!

```
# Regular style
MONGO_ID=$(sudo docker run -d <yourname>/mongodb)

# Lean and mean
MONGO_ID=$(sudo docker run -d <yourname>/mongodb --noprealloc --smallfiles)

# Check the logs out
sudo docker logs $MONGO_ID

# Connect and play around
mongo --port <port you get from `docker ps`>
```

Sweet!

4.10 Riak

:

- This example assumes you have Docker running in daemon mode. For more information please see *Docker*.
- **If you don't like sudo** then see *Giving non-root access*

The goal of this example is to show you how to build a Docker image with Riak pre-installed.

4.10.1 Dockerfile

Create an empty file called `Dockerfile`:

```
touch Dockerfile
```

Next, define the parent image you want to use to build your image on top of. We'll use Ubuntu (tag: `latest`), which is available on the docker index:

```
# Riak
#
# VERSION       0.1.0

# Use the Ubuntu base image provided by dotCloud
FROM ubuntu:latest
MAINTAINER Hector Castro hector@basho.com
```

Next, we update the APT cache and apply any updates:

```
# Update the APT cache
RUN sed -i.bak 's/main$/main universe/' /etc/apt/sources.list
RUN apt-get update
RUN apt-get upgrade -y
```

After that, we install and setup a few dependencies:

- `curl` is used to download Basho's APT repository key
- `lsb-release` helps us derive the Ubuntu release codename
- `openssh-server` allows us to login to containers remotely and join Riak nodes to form a cluster
- `supervisor` is used manage the OpenSSH and Riak processes

```
# Install and setup project dependencies
RUN apt-get install -y curl lsb-release supervisor openssh-server

RUN mkdir -p /var/run/sshd
RUN mkdir -p /var/log/supervisor

RUN locale-gen en_US en_US.UTF-8

ADD supervisord.conf /etc/supervisor/conf.d/supervisord.conf

RUN echo 'root:basho' | chpasswd
```

Next, we add Basho's APT repository:

```
RUN curl -s http://apt.basho.com/gpg/basho.apt.key | apt-key add --
RUN echo "deb http://apt.basho.com $(lsb_release -cs) main" > /etc/apt/sources.list.d/basho.list
RUN apt-get update
```

After that, we install Riak and alter a few defaults:

```
# Install Riak and prepare it to run
RUN apt-get install -y riak
RUN sed -i.bak 's/127.0.0.1/0.0.0.0/' /etc/riak/app.config
RUN echo "ulimit -n 4096" >> /etc/default/riak
```

Almost there. Next, we add a hack to get us by the lack of `initctl`:

```
# Hack for initctl
# See: https://github.com/dotcloud/docker/issues/1024
RUN dpkg-divert --local --rename --add /sbin/initctl
RUN ln -s /bin/true /sbin/initctl
```

Then, we expose the Riak Protocol Buffers and HTTP interfaces, along with SSH:

```
# Expose Riak Protocol Buffers and HTTP interfaces, along with SSH
EXPOSE 8087 8098 22
```

Finally, run `supervisord` so that Riak and OpenSSH are started:

```
CMD ["/usr/bin/supervisord"]
```

4.10.2 Create a `supervisord` configuration file

Create an empty file called `supervisord.conf`. Make sure it's at the same directory level as your `Dockerfile`:

```
touch supervisord.conf
```

Populate it with the following program definitions:

```
[supervisord]
nodaemon=true

[program:sshd]
command=/usr/sbin/sshd -D
stdout_logfile=/var/log/supervisor/%(program_name)s.log
stderr_logfile=/var/log/supervisor/%(program_name)s.log
autorestart=true

[program:riak]
command=bash -c ". /etc/default/riak && /usr/sbin/riak console"
pidfile=/var/log/riak/riak.pid
stdout_logfile=/var/log/supervisor/%(program_name)s.log
stderr_logfile=/var/log/supervisor/%(program_name)s.log
```

4.10.3 Build the Docker image for Riak

Now you should be able to build a Docker image for Riak:

```
docker build -t "<yourname>/riak" .
```

4.10.4 Next steps

Riak is a distributed database. Many production deployments consist of at least five nodes. See the docker-riak project details on how to deploy a Riak cluster using Docker and Pipework.

4.11 DockerSupervisor

:

- This example assumes you have Docker running in daemon mode. For more information please see *Docker*.
- **If you don't like sudo** then see *Giving non-root access*

Traditionally a Docker container runs a single process when it is launched, for example an Apache daemon or a SSH server daemon. Often though you want to run more than one process in a container. There are a number of ways you can achieve this ranging from using a simple Bash script as the value of your container's `CMD` instruction to installing a process management tool.

In this example we're going to make use of the process management tool, Supervisor, to manage multiple processes in our container. Using Supervisor allows us to better control, manage, and restart the processes we want to run. To demonstrate this we're going to install and manage both an SSH daemon and an Apache daemon.

4.11.1 Dockerfile

Let's start by creating a basic `Dockerfile` for our new image.

```
FROM ubuntu:latest
MAINTAINER examples@docker.io
RUN echo "deb http://archive.ubuntu.com/ubuntu precise main universe" > /etc/apt/sources.list
RUN apt-get update
RUN apt-get upgrade -y
```

4.11.2 Supervisor

We can now install our SSH and Apache daemons as well as Supervisor in our container.

```
RUN apt-get install -y openssh-server apache2 supervisor
RUN mkdir -p /var/run/sshd
RUN mkdir -p /var/log/supervisor
```

Here we're installing the `openssh-server`, `apache2` and `supervisor` (which provides the Supervisor daemon) packages. We're also creating two new directories that are needed to run our SSH daemon and Supervisor.

4.11.3 Supervisor

Now let's add a configuration file for Supervisor. The default file is called `supervisord.conf` and is located in `/etc/supervisor/conf.d/`.

```
ADD supervisord.conf /etc/supervisor/conf.d/supervisord.conf
```

Let's see what is inside our `supervisord.conf` file.

```
[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"
```

The `supervisord.conf` configuration file contains directives that configure Supervisor and the processes it manages. The first block `[supervisord]` provides configuration for Supervisor itself. We're using one directive, `nodaemon` which tells Supervisor to run interactively rather than daemonize.

The next two blocks manage the services we wish to control. Each block controls a separate process. The blocks contain a single directive, `command`, which specifies what command to run to start each process.

4.11.4 portsSupervisor

Now let's finish our `Dockerfile` by exposing some required ports and specifying the `CMD` instruction to start Supervisor when our container launches.

```
EXPOSE 22 80
CMD ["/usr/bin/supervisord"]
```

Here we've exposed ports 22 and 80 on the container and we're running the `/usr/bin/supervisord` binary when the container launches.

4.11.5

We can now build our new container.

```
sudo docker build -t <yourname>/supervisord .
```

4.11.6 Supervisor

Once we've got a built image we can launch a container from it.

```
sudo docker run -p 22 -p 80 -t -i <yourname>/supervisor
2013-11-25 18:53:22,312 CRIT Supervisor running as root (no user in config file)
2013-11-25 18:53:22,312 WARN Included extra file "/etc/supervisor/conf.d/supervisord.conf" during pa
2013-11-25 18:53:22,342 INFO supervisord started with pid 1
2013-11-25 18:53:23,346 INFO spawned: 'sshd' with pid 6
2013-11-25 18:53:23,349 INFO spawned: 'apache2' with pid 7
. . .
```

We've launched a new container interactively using the `docker run` command. That container has run Supervisor and launched the SSH and Apache daemons with it. We've specified the `-p` flag to expose ports 22 and 80. From here we can now identify the exposed ports and connect to one or both of the SSH and Apache daemons.

4.12 CFEngine

Create Docker containers with managed processes.

Docker monitors one process in each running container and the container lives or dies with that process. By introducing CFEngine inside Docker containers, we can alleviate a few of the issues that may arise:

- It is possible to easily start multiple processes within a container, all of which will be managed automatically, with the normal `docker run` command.
- If a managed process dies or crashes, CFEngine will start it again within 1 minute.
- The container itself will live as long as the CFEngine scheduling daemon (cf-execd) lives. With CFEngine, we are able to decouple the life of the container from the uptime of the service it provides.

4.12.1

CFEngine, together with the cfe-docker integration policies, are installed as part of the Dockerfile. This builds CFEngine into our Docker image.

The Dockerfile's `ENTRYPOINT` takes an arbitrary amount of commands (with any desired arguments) as parameters. When we run the Docker container these parameters get written to CFEngine policies and CFEngine takes over to ensure that the desired processes are running in the container.

CFEngine scans the process table for the `basename` of the commands given to the `ENTRYPOINT` and runs the command to start the process if the `basename` is not found. For example, if we start the container with `docker run "/path/to/my/application parameters"`, CFEngine will look for a process named `application` and run the command. If an entry for `application` is not found in the process table at any point in time, CFEngine will execute `/path/to/my/application parameters` to start the application once again. The check on the process table happens every minute.

Note that it is therefore important that the command to start your application leaves a process with the basename of the command. This can be made more flexible by making some minor adjustments to the CFEngine policies, if desired.

4.12.2

This example assumes you have Docker installed and working. We will install and manage `apache2` and `sshd` in a single container.

There are three steps:

1. Install CFEngine into the container.
2. Copy the CFEngine Docker process management policy into the containerized CFEngine installation.
3. Start your application processes as part of the `docker run` command.

The first two steps can be done as part of a Dockerfile, as follows.

```
FROM ubuntu
MAINTAINER Eystein Måløy Stenberg <eytein.stenberg@gmail.com>

RUN apt-get -y install wget lsb-release unzip

# install latest CFEngine
RUN wget -qO- http://cfengine.com/pub/gpg.key | apt-key add -
RUN echo "deb http://cfengine.com/pub/apt $(lsb_release -cs) main" > /etc/apt/sources.list.d/cfengine
RUN apt-get update
RUN apt-get install cfengine-community
```

```
# install cfe-docker process management policy
RUN wget --no-check-certificate https://github.com/estenberg/cfe-docker/archive/master.zip -P /tmp/ &
RUN cp /tmp/cfe-docker-master/cfengine/bin/* /var/cfengine/bin/
RUN cp /tmp/cfe-docker-master/cfengine/inputs/* /var/cfengine/inputs/
RUN rm -rf /tmp/cfe-docker-master /tmp/master.zip

# apache2 and openssh are just for testing purposes, install your own apps here
RUN apt-get -y install openssh-server apache2
RUN mkdir -p /var/run/sshd
RUN echo "root:password" | chpasswd  # need a password for ssh

ENTRYPOINT ["/var/cfengine/bin/docker_processes_run.sh"]
```

By saving this file as `Dockerfile` to a working directory, you can then build your container with the docker build command, e.g. `docker build -t managed_image`.

Start the container with `apache2` and `sshd` running and managed, forwarding a port to our SSH instance:

```
docker run -p 127.0.0.1:222:22 -d managed_image "/usr/sbin/sshd" "/etc/init.d/apache2 start"
```

We now clearly see one of the benefits of the cfe-docker integration: it allows to start several processes as part of a normal `docker run` command.

We can now log in to our new container and see that both `apache2` and `sshd` are running. We have set the root password to "password" in the Dockerfile above and can use that to log in with ssh:

```
ssh -p222 root@127.0.0.1

ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 07:48 ?        00:00:00 /bin/bash /var/cfengine/bin/docker_processes_run.sh &
root        18     1  0 07:48 ?        00:00:00 /var/cfengine/bin/cf-execd -F
root        20     1  0 07:48 ?        00:00:00 /usr/sbin/sshd
root        32     1  0 07:48 ?        00:00:00 /usr/sbin/apache2 -k start
www-data    34    32  0 07:48 ?        00:00:00 /usr/sbin/apache2 -k start
www-data    35    32  0 07:48 ?        00:00:00 /usr/sbin/apache2 -k start
www-data    36    32  0 07:48 ?        00:00:00 /usr/sbin/apache2 -k start
root        93    20  0 07:48 ?        00:00:00 sshd: root@pts/0
root       105    93  0 07:48 pts/0    00:00:00 -bash
root       112   105  0 07:49 pts/0    00:00:00 ps -ef
```

If we stop apache2, it will be started again within a minute by CFEngine.

```
service apache2 status
 Apache2 is running (pid 32).
service apache2 stop
         * Stopping web server apache2 ... waiting    [ OK ]
service apache2 status
 Apache2 is NOT running.
# ... wait up to 1 minute...
service apache2 status
 Apache2 is running (pid 173).
```

4.12.3

To make sure your applications get managed in the same manner, there are just two things you need to adjust from the above example:

- In the Dockerfile used above, install your applications instead of `apache2` and `sshd`.
- When you start the container with `docker run`, specify the command line arguments to your applications rather than `apache2` and `sshd`.

Contents:

5.1 Commands

Contents:

5.1.1 Command Line Help

To list available commands, either run `docker` with no parameters or execute `docker help`:

```
$ sudo docker
  Usage: docker [OPTIONS] COMMAND [arg...]
    -H=[unix:///var/run/docker.sock]: tcp://[host[:port]] to bind/connect to or unix://[/path/to/sock

  A self-sufficient runtime for linux containers.

  ...
```

5.1.2 Types of Options

Boolean

Boolean options look like `-d=false`. The value you see is the default value which gets set if you do **not** use the boolean flag. If you do call `run -d`, that sets the opposite boolean value, so in this case, `true`, and so `docker run -d` **will** run in "detached" mode, in the background. Other boolean options are similar – specifying them will set the value to the opposite of the default value.

Multi

Options like `-a=[]` indicate they can be specified multiple times:

```
docker run -a stdin -a stdout -a stderr -i -t ubuntu /bin/bash
```

Sometimes this can use a more complex value string, as for `-v`:

```
docker run -v /host:/container example/mysql
```

Strings and Integers

Options like `-name=""` expect a string, and they can only be specified once. Options like `-c=0` expect an integer, and they can only be specified once.

5.1.3 Commands

5.1.4 daemon

```
Usage of docker:
  -D, --debug=false: Enable debug mode
  -H, --host=[]: Multiple tcp://host:port or unix://path/to/socket to bind in daemon mode, single con
  --api-enable-cors=false: Enable CORS headers in the remote API
  -b, --bridge="": Attach containers to a pre-existing network bridge; use 'none' to disable containe
  --bip="": Use this CIDR notation address for the network bridge's IP, not compatible with -b
  -d, --daemon=false: Enable daemon mode
  --dns=[]: Force docker to use specific DNS servers
  -g, --graph="/var/lib/docker": Path to use as the root of the docker runtime
  --icc=true: Enable inter-container communication
  --ip="0.0.0.0": Default IP address to use when binding container ports
  --iptables=true: Disable docker's addition of iptables rules
  -p, --pidfile="/var/run/docker.pid": Path to use for daemon PID file
  -r, --restart=true: Restart previously running containers
  -s, --storage-driver="": Force the docker runtime to use a specific storage driver
  -v, --version=false: Print version information and quit
  -mtu, --mtu=0: Set the containers network MTU; if no value is provided: default to the default rout
```

The Docker daemon is the persistent process that manages containers. Docker uses the same binary for both the daemon and client. To run the daemon you provide the `-d` flag.

To force Docker to use devicemapper as the storage driver, use `docker -d -s devicemapper`.

To set the DNS server for all Docker containers, use `docker -d -dns 8.8.8.8`.

To run the daemon with debug output, use `docker -d -D`.

The docker client will also honor the `DOCKER_HOST` environment variable to set the `-H` flag for the client.

```
docker -H tcp://0.0.0.0:4243 ps
# or
export DOCKER_HOST="tcp://0.0.0.0:4243"
docker ps
# both are equal
```

To run the daemon with systemd socket activation, use `docker -d -H fd://`. Using `fd://` will work perfectly for most setups but you can also specify individual sockets too `docker -d -H fd://3`. If the specified socket activated files aren't found then docker will exit. You can find examples of using systemd socket activation with docker and systemd in the docker source tree.

5.1.5 attach

```
Usage: docker attach CONTAINER

Attach to a running container.
```

```
  --no-stdin=false: Do not attach stdin
  --sig-proxy=true: Proxify all received signal to the process (even in non-tty mode)
```

You can detach from the container again (and leave it running) with `CTRL-c` (for a quiet exit) or `CTRL-\` to get a stacktrace of the Docker client when it quits. When you detach from the container's process the exit code will be returned to the client.

To stop a container, use `docker stop`.

To kill the container, use `docker kill`.

Examples:

```
$ ID=$(sudo docker run -d ubuntu /usr/bin/top -b)
$ sudo docker attach $ID
top - 02:05:52 up  3:05,  0 users,  load average: 0.01, 0.02, 0.05
Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.1%us,  0.2%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:    373572k total,   355560k used,    18012k free,    27872k buffers
Swap:   786428k total,        0k used,   786428k free,   221740k cached

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1 root      20   0 17200 1116  912 R    0  0.3   0:00.03 top

 top - 02:05:55 up  3:05,  0 users,  load average: 0.01, 0.02, 0.05
 Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
 Cpu(s):  0.0%us,  0.2%sy,  0.0%ni, 99.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
 Mem:    373572k total,   355244k used,    18328k free,    27872k buffers
 Swap:   786428k total,        0k used,   786428k free,   221776k cached

   PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
       1 root      20   0 17208 1144  932 R    0  0.3   0:00.03 top


 top - 02:05:58 up  3:06,  0 users,  load average: 0.01, 0.02, 0.05
 Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
 Cpu(s):  0.2%us,  0.3%sy,  0.0%ni, 99.5%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
 Mem:    373572k total,   355780k used,    17792k free,    27880k buffers
 Swap:   786428k total,        0k used,   786428k free,   221776k cached

 PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
     1 root      20   0 17208 1144  932 R    0  0.3   0:00.03 top
^C$
$ sudo docker stop $ID
```

5.1.6 build

```
Usage: docker build [OPTIONS] PATH | URL | -
Build a new container image from the source code at PATH
  -t, --time="": Repository name (and optionally a tag) to be applied
         to the resulting image in case of success.
  -q, --quiet=false: Suppress verbose build output.
  --no-cache: Do not use the cache when building the image.
  --rm: Remove intermediate containers after a successful build
```

The files at PATH or URL are called the "context" of the build. The build process may refer to any of the files in the context, for example when using an *ADD* instruction. When a single Dockerfile is given as URL, then no context is set. When a Git repository is set as URL, then the repository is used as the context

:

Dockerfile Reference.

Examples:

```
$ sudo 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
Successfully built f52f38b7823e
```

This example specifies that the PATH is ., and so all the files in the local directory get tar'd and sent to the Docker daemon. The PATH specifies where to find the files for the "context" of the build on the Docker daemon. Remember that the daemon could be running on a remote machine and that no parsing of the Dockerfile happens at the client side (where you're running docker build). That means that *all* the files at PATH get sent, not just the ones listed to *ADD* in the Dockerfile.

The transfer of context from the local machine to the Docker daemon is what the docker client means when you see the "Uploading context" message.

```
$ sudo docker build -t vieux/apache:2.0 .
```

This will build like the previous example, but it will then tag the resulting image. The repository name will be vieux/apache and the tag will be 2.0

```
$ sudo docker build - < Dockerfile
```

This will read a Dockerfile from *stdin* without context. Due to the lack of a context, no contents of any local directory will be sent to the docker daemon. Since there is no context, a Dockerfile ADD only works if it refers to a remote URL.

```
$ sudo docker build github.com/creack/docker-firefox
```

This will clone the GitHub repository and use the cloned repository as context. The Dockerfile at the root of the repository is used as Dockerfile. Note that you can specify an arbitrary Git repository by using the git:// schema.

5.1.7 commit

```
Usage: docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

Create a new image from a container's changes

  -m, --message="": Commit message
  -a, --author="": Author (eg. "John Hannibal Smith <hannibal@a-team.com>"
  --run="": Configuration to be applied when the image is launched with `docker run`.
            (ex: -run='{"Cmd": ["cat", "/world"], "PortSpecs": ["22"]}')
```

Commit an existing container

```
$ sudo docker ps
ID                  IMAGE               COMMAND             CREATED             STATUS              P
c3f279d17e0a        ubuntu:12.04        /bin/bash           7 days ago          Up 25 hours
197387f1b436        ubuntu:12.04        /bin/bash           7 days ago          Up 25 hours
$ docker commit c3f279d17e0a  SvenDowideit/testimage:version3
f5283438590d
$ docker images | head
REPOSITORY                      TAG                 ID                  CREATED             VIRTUAL
SvenDowideit/testimage          version3            f5283438590d        16 seconds ago      335.7 M
```

Change the command that a container runs

Sometimes you have an application container running just a service and you need to make a quick change and then change it back.

In this example, we run a container with `ls` and then change the image to run `ls /etc`.

```
$ docker run -t -name test ubuntu ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  selinux  srv  sys  tmp
$ docker commit -run='{"Cmd": ["ls","/etc"]}' test test2
933d16de9e70005304c1717b5c6f2f39d6fd50752834c6f34a155c70790011eb
$ docker run -t test2
adduser.conf            gshadow          login.defs           rc0.d
alternatives            gshadow-         logrotate.d          rc1.d
apt                     host.conf        lsb-base             rc2.d
...
```

Full -run example

The `--run` JSON hash changes the `Config` section when running `docker inspect CONTAINERID` or `config` when running `docker inspect IMAGEID`.

(Multiline is okay within a single quote `'`)

```
$ sudo docker commit -run='
{
    "Entrypoint" : null,
    "Privileged" : false,
    "User" : "",
    "VolumesFrom" : "",
    "Cmd" : ["cat", "-e", "/etc/resolv.conf"],
    "Dns" : ["8.8.8.8", "8.8.4.4"],
```

```
    "MemorySwap" : 0,
    "AttachStdin" : false,
    "AttachStderr" : false,
    "CpuShares" : 0,
    "OpenStdin" : false,
    "Volumes" : null,
    "Hostname" : "122612f45831",
    "PortSpecs" : ["22", "80", "443"],
    "Image" : "b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc",
    "Tty" : false,
    "Env" : [
       "HOME=/",
       "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "StdinOnce" : false,
    "Domainname" : "",
    "WorkingDir" : "/",
    "NetworkDisabled" : false,
    "Memory" : 0,
    "AttachStdout" : false
}' $CONTAINER_ID
```

5.1.8 cp

```
Usage: docker cp CONTAINER:PATH HOSTPATH

Copy files/folders from the containers filesystem to the host
path.  Paths are relative to the root of the filesystem.
```

```
$ sudo docker cp 7bb0e258aefe:/etc/debian_version .
$ sudo docker cp blue_frog:/etc/hosts .
```

5.1.9 diff

```
Usage: docker diff CONTAINER

List the changed files and directories in a container's filesystem
```

There are 3 events that are listed in the 'diff':

1. `'A'` - Add
2. `'D'` - Delete
3. `'C'` - Change

For example:

```
$ sudo docker diff 7bb0e258aefe

C /dev
A /dev/kmsg
C /etc
A /etc/mtab
A /go
A /go/src
```

```
A /go/src/github.com
A /go/src/github.com/dotcloud
A /go/src/github.com/dotcloud/docker
A /go/src/github.com/dotcloud/docker/.git
....
```

5.1.10 events

```
Usage: docker events

Get real time events from the server

--since="": Show previously created events and then stream.
           (either seconds since epoch, or date string as below)
```

Examples

You'll need two shells for this example.

Shell 1: Listening for events

```
$ sudo docker events
```

Shell 2: Start and Stop a Container

```
$ sudo docker start 4386fb97867d
$ sudo docker stop 4386fb97867d
```

Shell 1: (Again .. now showing events)

```
[2013-09-03 15:49:26 +0200 CEST] 4386fb97867d: (from 12de384bfb10) start
[2013-09-03 15:49:29 +0200 CEST] 4386fb97867d: (from 12de384bfb10) die
[2013-09-03 15:49:29 +0200 CEST] 4386fb97867d: (from 12de384bfb10) stop
```

Show events in the past from a specified time

```
$ sudo docker events -since 1378216169
[2013-09-03 15:49:29 +0200 CEST] 4386fb97867d: (from 12de384bfb10) die
[2013-09-03 15:49:29 +0200 CEST] 4386fb97867d: (from 12de384bfb10) stop

$ sudo docker events -since '2013-09-03'
[2013-09-03 15:49:26 +0200 CEST] 4386fb97867d: (from 12de384bfb10) start
[2013-09-03 15:49:29 +0200 CEST] 4386fb97867d: (from 12de384bfb10) die
[2013-09-03 15:49:29 +0200 CEST] 4386fb97867d: (from 12de384bfb10) stop

$ sudo docker events -since '2013-09-03 15:49:29 +0200 CEST'
[2013-09-03 15:49:29 +0200 CEST] 4386fb97867d: (from 12de384bfb10) die
[2013-09-03 15:49:29 +0200 CEST] 4386fb97867d: (from 12de384bfb10) stop
```

5.1.11 export

```
Usage: docker export CONTAINER

Export the contents of a filesystem as a tar archive to STDOUT
```

For example:

```
$ sudo docker export red_panda > latest.tar
```

5.1.12 history

```
Usage: docker history [OPTIONS] IMAGE

Show the history of an image

  --no-trunc=false: Don't truncate output
  -q, --quiet=false: only show numeric IDs
```

To see how the `docker:latest` image was built:

```
$ docker history docker
ID                  CREATED             CREATED BY
docker:latest       19 hours ago        /bin/sh -c #(nop) ADD . in /go/src/github.com/dotcloud/docke
cf5f2467662d        2 weeks ago         /bin/sh -c #(nop) ENTRYPOINT ["hack/dind"]
3538fbe372bf        2 weeks ago         /bin/sh -c #(nop) WORKDIR /go/src/github.com/dotcloud/docker
7450f65072e5        2 weeks ago         /bin/sh -c #(nop) VOLUME /var/lib/docker
b79d62b97328        2 weeks ago         /bin/sh -c apt-get install -y -q lxc
36714852a550        2 weeks ago         /bin/sh -c apt-get install -y -q iptables
8c4c706df1d6        2 weeks ago         /bin/sh -c /bin/echo -e '[default]\naccess_key=$AWS_ACCESS_KE
b89989433c48        2 weeks ago         /bin/sh -c pip install python-magic
a23e640d85b5        2 weeks ago         /bin/sh -c pip install s3cmd
41f54fec7e79        2 weeks ago         /bin/sh -c apt-get install -y -q python-pip
d9bc04add907        2 weeks ago         /bin/sh -c apt-get install -y -q reprepro dpkg-sig
e74f4760fa70        2 weeks ago         /bin/sh -c gem install --no-rdoc --no-ri fpm
1e43224726eb        2 weeks ago         /bin/sh -c apt-get install -y -q ruby1.9.3 rubygems libffi-de
460953ae9d7f        2 weeks ago         /bin/sh -c #(nop) ENV GOPATH=/go:/go/src/github.com/dotcloud/
8b63eb1d666b        2 weeks ago         /bin/sh -c #(nop) ENV PATH=/usr/local/sbin:/usr/local/bin:/us
3087f3bcedf2        2 weeks ago         /bin/sh -c #(nop) ENV GOROOT=/goroot
635840d198e5        2 weeks ago         /bin/sh -c cd /goroot/src && ./make.bash
439f4a0592ba        2 weeks ago         /bin/sh -c curl -s https://go.googlecode.com/files/go1.1.2.sr
13967ed36e93        2 weeks ago         /bin/sh -c #(nop) ENV CGO_ENABLED=0
bf7424458437        2 weeks ago         /bin/sh -c apt-get install -y -q build-essential
a89ec997c3bf        2 weeks ago         /bin/sh -c apt-get install -y -q mercurial
b9f165c6e749        2 weeks ago         /bin/sh -c apt-get install -y -q git
17a64374afa7        2 weeks ago         /bin/sh -c apt-get install -y -q curl
d5e85dc5b1d8        2 weeks ago         /bin/sh -c apt-get update
13e642467c11        2 weeks ago         /bin/sh -c echo 'deb http://archive.ubuntu.com/ubuntu precise
ae6dde92a94e        2 weeks ago         /bin/sh -c #(nop) MAINTAINER Solomon Hykes <solomon@dotcloud
ubuntu:12.04        6 months ago
```

5.1.13 images

```
Usage: docker images [OPTIONS] [NAME]
```

```
List images

  -a, --all=false: show all images (by default filter out the intermediate images used to build)
  --no-trunc=false: Don't truncate output
  -q, --quiet=false: only show numeric IDs
  --tree=false: output graph in tree format
  --viz=false: output graph in graphviz format
```

Listing the most recently created images

```
$ sudo docker images | head
REPOSITORY                    TAG                 IMAGE ID            CREATED             VIRTUAL SI
<none>                        <none>              77af4d6b9913        19 hours ago        1.089 GB
committest                    latest              b6fa739cedf5        19 hours ago        1.089 GB
<none>                        <none>              78a85c484f71        19 hours ago        1.089 GB
docker                        latest              30557a29d5ab        20 hours ago        1.089 GB
<none>                        <none>              0124422dd9f9        20 hours ago        1.089 GB
<none>                        <none>              18ad6fad3402        22 hours ago        1.082 GB
<none>                        <none>              f9f1e26352f0        23 hours ago        1.089 GB
tryout                        latest              2629d1fa0b81        23 hours ago        131.5 MB
<none>                        <none>              5ed6274db6ce        24 hours ago        1.089 GB
```

Listing the full length image IDs

```
$ sudo docker images --no-trunc | head
REPOSITORY                    TAG                 IMAGE ID
<none>                        <none>              77af4d6b9913e693e8d0b4b294fa62ade6054e6b2f1ffb617ac
committest                    latest              b6fa739cedf5ea12a620a439402b6004d057da800f91c7524b5
<none>                        <none>              78a85c484f71509adeaace20e72e941f6bdd2b25b4c75da8693
docker                        latest              30557a29d5abc51e5f1d5b472e79b7e296f595abcf19fe6b919
<none>                        <none>              0124422dd9f9cf7ef15c0617cda3931ee68346455441d66ab8b
<none>                        <none>              18ad6fad340262ac2a636efd98a6d1f0ea775ae3d45240d3418
<none>                        <none>              f9f1e26352f0a3ba6a0ff68167559f64f3e21ff7ada60366e2c
tryout                        latest              2629d1fa0b81b222fca63371ca16cbf6a0772d07759ff80e8d1
<none>                        <none>              5ed6274db6ceb2397844896966ea239290555e74ef307030ebb
```

Displaying images visually

```
$ sudo docker images --viz | dot -Tpng -o docker.png
```

Displaying image hierarchy

```
$ sudo docker images --tree

-8dbd9e392a96 Size: 131.5 MB (virtual 131.5 MB) Tags: ubuntu:12.04,ubuntu:latest,ubuntu:precise
-27cf78414709 Size: 180.1 MB (virtual 180.1 MB)
  -b750fe79269d Size: 24.65 kB (virtual 180.1 MB) Tags: ubuntu:12.10,ubuntu:quantal
    -f98de3b610d5 Size: 12.29 kB (virtual 180.1 MB)
    | -7da80deb7dbf Size: 16.38 kB (virtual 180.1 MB)
    |   -65ed2fee0a34 Size: 20.66 kB (virtual 180.2 MB)
    |     -a2b9ea53dddc Size: 819.7 MB (virtual 999.8 MB)
    |       -a29b932eaba8 Size: 28.67 kB (virtual 999.9 MB)
```

```
|         -e270a44f124d Size: 12.29 kB (virtual 999.9 MB) Tags: progrium/buildstep:latest
-17e74ac162d8 Size: 53.93 kB (virtual 180.2 MB)
  -339a3f56b760 Size: 24.65 kB (virtual 180.2 MB)
    -904fcc40e34d Size: 96.7 MB (virtual 276.9 MB)
      -b1b0235328dd Size: 363.3 MB (virtual 640.2 MB)
        -7cb05d1acb3b Size: 20.48 kB (virtual 640.2 MB)
          -47bf6f34832d Size: 20.48 kB (virtual 640.2 MB)
            -f165104e82ed Size: 12.29 kB (virtual 640.2 MB)
              -d9cf85a47b7e Size: 1.911 MB (virtual 642.2 MB)
                -3ee562df86ca Size: 17.07 kB (virtual 642.2 MB)
                  -b05fc2d00e4a Size: 24.96 kB (virtual 642.2 MB)
                    -c96a99614930 Size: 12.29 kB (virtual 642.2 MB)
                      -a6a357a48c49 Size: 12.29 kB (virtual 642.2 MB) Tags: ndj/mongodb:latest
```

5.1.14 `import`

```
Usage: docker import URL|- [REPOSITORY[:TAG]]

Create an empty filesystem image and import the contents of the tarball
(.tar, .tar.gz, .tgz, .bzip, .tar.xz, .txz) into it, then optionally tag it.
```

At this time, the URL must start with `http` and point to a single file archive (.tar, .tar.gz, .tgz, .bzip, .tar.xz, or .txz) containing a root filesystem. If you would like to import from a local directory or archive, you can use the – parameter to take the data from *stdin*.

Examples

Import from a remote location

This will create a new untagged image.

```
$ sudo docker import http://example.com/exampleimage.tgz
```

Import from a local file

Import to docker via pipe and *stdin*.

```
$ cat exampleimage.tgz | sudo docker import - exampleimagelocal:new
```

Import from a local directory

```
$ sudo tar -c . | docker import - exampleimagedir
```

Note the `sudo` in this example – you must preserve the ownership of the files (especially root ownership) during the archiving with tar. If you are not root (or the sudo command) when you tar, then the ownerships might not get preserved.

5.1.15 info

```
Usage: docker info

Display system-wide information.

$ sudo docker info
Containers: 292
Images: 194
Debug mode (server): false
Debug mode (client): false
Fds: 22
Goroutines: 67
LXC Version: 0.9.0
EventsListeners: 115
Kernel Version: 3.8.0-33-generic
WARNING: No swap limit support
```

5.1.16 insert

```
Usage: docker insert IMAGE URL PATH

Insert a file from URL in the IMAGE at PATH
```

Use the specified `IMAGE` as the parent for a new image which adds a *layer* containing the new file. The `insert` command does not modify the original image, and the new image has the contents of the parent image, plus the new file.

Examples

Insert file from GitHub

```
$ sudo docker insert 8283e18b24bc https://raw.github.com/metalivedev/django/master/postinstall /tmp/p
06fd35556d7b
```

5.1.17 inspect

```
Usage: docker inspect CONTAINER|IMAGE [CONTAINER|IMAGE...]

Return low-level information on a container/image

  -f, --format="": Format the output using the given go template.
```

By default, this will render all results in a JSON array. If a format is specified, the given template will be executed for each result.

Go's text/template package describes all the details of the format.

Examples

Get an instance's IP Address

For the most part, you can pick out any field from the JSON in a fairly straightforward manner.

```
$ sudo docker inspect --format='{{.NetworkSettings.IPAddress}}' $INSTANCE_ID
```

List All Port Bindings

One can loop over arrays and maps in the results to produce simple text output:

```
$ sudo docker inspect -format='{{range $p, $conf := .NetworkSettings.Ports}} {{$p}} -> {{(index $con
```

Find a Specific Port Mapping

The `.Field` syntax doesn't work when the field name begins with a number, but the template language's `index` function does. The `.NetworkSettings.Ports` section contains a map of the internal port mappings to a list of external address/port objects, so to grab just the numeric public port, you use `index` to find the specific port map, and then `index` 0 contains first object inside of that. Then we ask for the `HostPort` field to get the public address.

```
$ sudo docker inspect -format='{{(index (index .NetworkSettings.Ports "8787/tcp") 0).HostPort}}' $INS
```

5.1.18 `kill`

```
Usage: docker kill [OPTIONS] CONTAINER [CONTAINER...]

Kill a running container (send SIGKILL, or specified signal)

  -s, --signal="KILL": Signal to send to the container
```

The main process inside the container will be sent SIGKILL, or any signal specified with option `--signal`.

Known Issues (kill)

- Issue 197 indicates that `docker kill` may leave directories behind and make it difficult to remove the container.
- Issue 3844 lxc 1.0.0 beta3 removed `lcx-kill` which is used by Docker versions before 0.8.0; see the issue for a workaround.

5.1.19 `load`

```
Usage: docker load < repository.tar

Loads a tarred repository from the standard input stream.
Restores both images and tags.
```

5.1.20 login

```
Usage: docker login [OPTIONS] [SERVER]

Register or Login to the docker registry server

-e, --email="": email
-p, --password="": password
-u, --username="": username

If you want to login to a private registry you can
specify this by adding the server name.

example:
docker login localhost:8080
```

5.1.21 logs

```
Usage: docker logs [OPTIONS] CONTAINER

Fetch the logs of a container

-f, --follow=false: Follow log output
```

The `docker logs` command is a convenience which batch-retrieves whatever logs are present at the time of execution. This does not guarantee execution order when combined with a `docker run` (i.e. your run may not have generated any logs at the time you execute `docker logs`).

The `docker logs --follow` command combines `docker logs` and `docker attach`: it will first return all logs from the beginning and then continue streaming new output from the container's stdout and stderr.

5.1.22 port

```
Usage: docker port [OPTIONS] CONTAINER PRIVATE_PORT

Lookup the public-facing port which is NAT-ed to PRIVATE_PORT
```

5.1.23 ps

```
Usage: docker ps [OPTIONS]

List containers

  -a, --all=false: Show all containers. Only running containers are shown by default.
  --no-trunc=false: Don't truncate output
  -q, --quiet=false: Only display numeric IDs
```

Running `docker ps` showing 2 linked containers.

```
$ docker ps
CONTAINER ID        IMAGE                          COMMAND                CREATED             STATUS
4c01db0b339c        ubuntu:12.04                   bash                   17 seconds ago      Up 16 se
d7886598dbe2        crosbymichael/redis:latest     /redis-server --dir    33 minutes ago      Up 33 mi
fd2645e2e2b5        busybox:latest                 top                    10 days ago         Ghost
```

The last container is marked as a `Ghost` container. It is a container that was running when the docker daemon was restarted (upgraded, or `-H` settings changed). The container is still running, but as this docker daemon process is not able to manage it, you can't attach to it. To bring them out of `Ghost` Status, you need to use `docker kill` or `docker restart`.

5.1.24 pull

```
Usage: docker pull NAME

Pull an image or a repository from the registry
```

5.1.25 push

```
Usage: docker push NAME

Push an image or a repository to the registry
```

5.1.26 restart

```
Usage: docker restart [OPTIONS] NAME

Restart a running container
```

5.1.27 rm

```
Usage: docker rm [OPTIONS] CONTAINER

Remove one or more containers
    --link="": Remove the link instead of the actual container
```

Known Issues (rm)

- Issue 197 indicates that `docker kill` may leave directories behind and make it difficult to remove the container.

Examples:

```
$ sudo docker rm /redis
/redis
```

This will remove the container referenced under the link `/redis`.

```
$ sudo docker rm --link /webapp/redis
/webapp/redis
```

This will remove the underlying link between `/webapp` and the `/redis` containers removing all network communication.

```
$ sudo docker rm `docker ps -a -q`
```

This command will delete all stopped containers. The command `docker ps -a -q` will return all existing container IDs and pass them to the `rm` command which will delete them. Any running containers will not be deleted.

5.1.28 rmi

```
Usage: docker rmi IMAGE [IMAGE...]

Remove one or more images
```

Removing tagged images

Images can be removed either by their short or long ID's, or their image names. If an image has more than one name, each of them needs to be removed before the image is removed.

```
$ sudo docker images
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
test1                     latest              fd484f19954f        23 seconds ago      7 B (virtual 4
test                      latest              fd484f19954f        23 seconds ago      7 B (virtual 4
test2                     latest              fd484f19954f        23 seconds ago      7 B (virtual 4

$ sudo docker rmi fd484f19954f
Error: Conflict, cannot delete image fd484f19954f because it is tagged in multiple repositories
2013/12/11 05:47:16 Error: failed to remove one or more images

$ sudo docker rmi test1
Untagged: fd484f19954f4920da7ff372b5067f5b7ddb2fd3830cecd17b96ea9e286ba5b8
$ sudo docker rmi test2
Untagged: fd484f19954f4920da7ff372b5067f5b7ddb2fd3830cecd17b96ea9e286ba5b8

$ sudo docker images
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
test1                     latest              fd484f19954f        23 seconds ago      7 B (virtual 4
$ sudo docker rmi test
Untagged: fd484f19954f4920da7ff372b5067f5b7ddb2fd3830cecd17b96ea9e286ba5b8
Deleted: fd484f19954f4920da7ff372b5067f5b7ddb2fd3830cecd17b96ea9e286ba5b8
```

5.1.29 run

```
Usage: docker run [OPTIONS] IMAGE[:TAG] [COMMAND] [ARG...]

Run a command in a new container

  -a, --attach=map[]: Attach to stdin, stdout or stderr
  -c, --cpu-shares=0: CPU shares (relative weight)
  --cidfile="": Write the container ID to the file
  -d, --detach=false: Detached mode: Run container in the background, print new container id
  -e, --env=[]: Set environment variables
  -h, --host="": Container host name
  -i, --interactive=false: Keep stdin open even if not attached
  --privileged=false: Give extended privileges to this container
  -m, --memory="": Memory limit (format: <number><optional unit>, where unit = b, k, m or g)
  -n, --networking=true: Enable networking for this container
```

```
-p, --publish=[]: Map a network port to the container
--rm=false: Automatically remove the container when it exits (incompatible with -d)
-t, --tty=false: Allocate a pseudo-tty
-u, --user="": Username or UID
--dns=[]: Set custom dns servers for the container
-v, --volume=[]: Create a bind mount to a directory or file with: [host-path]:[container-path]:[rw
--volumes-from="": Mount all volumes from the given container(s)
--entrypoint="": Overwrite the default entrypoint set by the image
-w, --workdir="": Working directory inside the container
--lxc-conf=[]: Add custom lxc options -lxc-conf="lxc.cgroup.cpuset.cpus = 0,1"
--sig-proxy=true: Proxify all received signal to the process (even in non-tty mode)
--expose=[]: Expose a port from the container without publishing it to your host
--link="": Add link to another container (name:alias)
--name="": Assign the specified name to the container. If no name is specific docker will generate
-P, --publish-all=false: Publish all exposed ports to the host interfaces
```

The `docker run` command first `creates` a writeable container layer over the specified image, and then `starts` it using the specified command. That is, `docker run` is equivalent to the API `/containers/create` then `/containers/(id)/start`.

The `docker run` command can be used in combination with `docker commit` to *change the command that a container runs*.

Known Issues (run -volumes-from)

- Issue 2702: "lxc-start: Permission denied - failed to mount" could indicate a permissions problem with AppArmor. Please see the issue for a workaround.

Examples:

```
$ sudo docker run --cidfile /tmp/docker_test.cid ubuntu echo "test"
```

This will create a container and print `test` to the console. The `cidfile` flag makes Docker attempt to create a new file and write the container ID to it. If the file exists already, Docker will return an error. Docker will close this file when `docker run` exits.

```
$ sudo docker run -t -i --rm ubuntu bash
root@bc338942ef20:/# mount -t tmpfs none /mnt
mount: permission denied
```

This will *not* work, because by default, most potentially dangerous kernel capabilities are dropped; including `cap_sys_admin` (which is required to mount filesystems). However, the `-privileged` flag will allow it to run:

```
$ sudo docker run --privileged ubuntu bash
root@50e3f57e16e6:/# mount -t tmpfs none /mnt
root@50e3f57e16e6:/# df -h
Filesystem      Size  Used Avail Use% Mounted on
none            1.9G     0  1.9G   0% /mnt
```

The `-privileged` flag gives *all* capabilities to the container, and it also lifts all the limitations enforced by the `device` cgroup controller. In other words, the container can then do almost everything that the host can do. This flag exists to allow special use-cases, like running Docker within Docker.

```
$ sudo docker  run -w /path/to/dir/ -i -t  ubuntu pwd
```

The -w lets the command being executed inside directory given, here /path/to/dir/. If the path does not exists it is created inside the container.

```
$ sudo docker  run  -v `pwd`:`pwd` -w `pwd` -i -t  ubuntu pwd
```

The -v flag mounts the current working directory into the container. The -w lets the command being executed inside the current working directory, by changing into the directory to the value returned by pwd. So this combination executes the command using the container, but inside the current working directory.

```
$ sudo docker run -v /doesnt/exist:/foo -w /foo -i -t ubuntu bash
```

When the host directory of a bind-mounted volume doesn't exist, Docker will automatically create this directory on the host for you. In the example above, Docker will create the /doesnt/exist folder before starting your container.

```
$ sudo docker run -t -i -v /var/run/docker.sock:/var/run/docker.sock -v ./static-docker:/usr/bin/doc
```

By bind-mounting the docker unix socket and statically linked docker binary (such as that provided by https://get.docker.io), you give the container the full access to create and manipulate the host's docker daemon.

```
$ sudo docker run -p 127.0.0.1:80:8080 ubuntu bash
```

This binds port 8080 of the container to port 80 on 127.0.0.1 of the host machine. explains in detail how to manipulate ports in Docker.

```
$ sudo docker run --expose 80 ubuntu bash
```

This exposes port 80 of the container for use within a link without publishing the port to the host system's interfaces. explains in detail how to manipulate ports in Docker.

```
$ sudo docker run --name console -t -i ubuntu bash
```

This will create and run a new container with the container name being console.

```
$ sudo docker run --link /redis:redis --name console ubuntu bash
```

The --link flag will link the container named /redis into the newly created container with the alias redis. The new container can access the network and environment of the redis container via environment variables. The --name flag will assign the name console to the newly created container.

```
$ sudo docker run --volumes-from 777f7dc92da7,ba8c0c54f0f2:ro -i -t ubuntu pwd
```

The --volumes-from flag mounts all the defined volumes from the referenced containers. Containers can be specified by a comma seperated list or by repetitions of the --volumes-from argument. The container ID may be optionally suffixed with :ro or :rw to mount the volumes in read-only or read-write mode, respectively. By default, the volumes are mounted in the same mode (read write or read only) as the reference container.

A complete example

```
$ sudo docker run -d --name static static-web-files sh
$ sudo docker run -d --expose=8098 --name riak riakserver
$ sudo docker run -d -m 100m -e DEVELOPMENT=1 -e BRANCH=example-code -v $(pwd):/app/bin:ro --name app
$ sudo docker run -d -p 1443:443 --dns=dns.dev.org -v /var/log/httpd --volumes-from static --link ria
$ sudo docker run -t -i --rm --volumes-from web -w /var/log/httpd busybox tail -f access.log
```

This example shows 5 containers that might be set up to test a web application change:

1. Start a pre-prepared volume image static-web-files (in the background) that has CSS, image and static HTML in it, (with a VOLUME instruction in the Dockerfile to allow the web server to use those files);

2. Start a pre-prepared `riakserver` image, give the container name `riak` and expose port `8098` to any containers that link to it;
3. Start the `appserver` image, restricting its memory usage to 100MB, setting two environment variables `DEVELOPMENT` and `BRANCH` and bind-mounting the current directory (`$(pwd)`) in the container in read-only mode as `/app/bin`;
4. Start the `webserver`, mapping port `443` in the container to port `1443` on the Docker server, setting the DNS server to `dns.dev.org`, creating a volume to put the log files into (so we can access it from another container), then importing the files from the volume exposed by the `static` container, and linking to all exposed ports from `riak` and `app`. Lastly, we set the hostname to `web.sven.dev.org` so its consistent with the pre-generated SSL certificate;
5. Finally, we create a container that runs `tail -f access.log` using the logs volume from the `web` container, setting the workdir to `/var/log/httpd`. The `-rm` option means that when the container exits, the container's layer is removed.

5.1.30 save

```
Usage: docker save image > repository.tar

Streams a tarred repository to the standard output stream.
Contains all parent layers, and all tags + versions.
```

5.1.31 search

```
Usage: docker search TERM

Search the docker index for images

 --no-trunc=false: Don't truncate output
 -s, --stars=0: Only displays with at least xxx stars
 -t, --trusted=false: Only show trusted builds
```

5.1.32 start

```
Usage: docker start [OPTIONS] CONTAINER

Start a stopped container

  -a, --attach=false: Attach container's stdout/stderr and forward all signals to the process
  -i, --interactive=false: Attach container's stdin
```

5.1.33 stop

```
Usage: docker stop [OPTIONS] CONTAINER [CONTAINER...]

Stop a running container (Send SIGTERM, and then SIGKILL after grace period)

  -t, --time=10: Number of seconds to wait for the container to stop before killing it.
```

The main process inside the container will receive SIGTERM, and after a grace period, SIGKILL

5.1.34 tag

```
Usage: docker tag [OPTIONS] IMAGE REPOSITORY[:TAG]

Tag an image into a repository

  -f, --force=false: Force
```

5.1.35 top

```
Usage: docker top CONTAINER [ps OPTIONS]

Lookup the running processes of a container
```

5.1.36 version

Show the version of the Docker client, daemon, and latest released version.

5.1.37 wait

```
Usage: docker wait [OPTIONS] NAME

Block until a container stops, then print its exit code.
```

5.2 Dockerfile Reference

Docker can act as a builder and read instructions from a text `Dockerfile` to automate the steps you would otherwise take manually to create an image. Executing `docker build` will run your steps and commit them along the way, giving you a final image.

Table of Contents

5.2.1 1. Usage

To *build* an image from a source repository, create a description file called `Dockerfile` at the root of your repository. This file will describe the steps to assemble the image.

Then call `docker build` with the path of your source repository as argument (for example, `.`):

```
sudo docker build .
```

The path to the source repository defines where to find the *context* of the build. The build is run by the Docker daemon, not by the CLI, so the whole context must be transferred to the daemon. The Docker CLI reports "Uploading context" when the context is sent to the daemon.

You can specify a repository and tag at which to save the new image if the build succeeds:

```
sudo docker build -t shykes/myapp .
```

The Docker daemon will run your steps one-by-one, committing the result to a new image if necessary, before finally outputting the ID of your new image. The Docker daemon will automatically clean up the context you sent.

Note that each instruction is run independently, and causes a new image to be created - so `RUN cd /tmp` will not have any effect on the next instructions.

Whenever possible, Docker will re-use the intermediate images, accelerating `docker build` significantly (indicated by `Using cache`:

```
$ docker build -t SvenDowideit/ambassador .
Uploading context 10.24 kB
Uploading context
Step 1 : FROM docker-ut
 ---> cbba202fe96b
Step 2 : MAINTAINER SvenDowideit@home.org.au
 ---> Using cache
 ---> 51182097be13
Step 3 : CMD env | grep _TCP= | sed 's/.*_PORT_\([0-9]*\)_TCP=tcp:\/\/\(.*\):\(.*\)/socat TCP4-LISTEN
 ---> Using cache
```

```
 ---> 1a5ffc17324d
Successfully built 1a5ffc17324d
```

When you're done with your build, you're ready to look into .

5.2.2 2. Format

The Dockerfile format is quite simple:

```
# Comment
INSTRUCTION arguments
```

The Instruction is not case-sensitive, however convention is for them to be UPPERCASE in order to distinguish them from arguments more easily.

Docker evaluates the instructions in a Dockerfile in order. **The first instruction must be 'FROM'** in order to specify the *Base Image* from which you are building.

Docker will treat lines that *begin* with `#` as a comment. A `#` marker anywhere else in the line will be treated as an argument. This allows statements like:

```
# Comment
RUN echo 'we are running some # of cool things'
```

5.2.3 3. Instructions

Here is the set of instructions you can use in a `Dockerfile` for building images.

3.1 FROM

```
FROM <image>
```

Or

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

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 .

`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.

If no `tag` is given to the `FROM` instruction, `latest` is assumed. If the used tag does not exist, an error will be returned.

3.2 MAINTAINER

```
MAINTAINER <name>
```

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

3.3 RUN

RUN has 2 forms:

- `RUN <command>` (the command is run in a shell - `/bin/sh -c`)
- `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 `/bin/sh`.

Known Issues (RUN)

- Issue 783 is about file permissions problems that can occur when using the AUFS file system. You might notice it during an attempt to `rm` a file, for example. The issue describes a workaround.
- Issue 2424 Locale will not be set automatically.

3.4 CMD

CMD has three forms:

- `CMD ["executable","param1","param2"]` (like an *exec*, preferred form)
- `CMD ["param1","param2"]` (as *default parameters to ENTRYPOINT*)
- `CMD command param1 param2` (as a *shell*)

There can only be one CMD 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 as well.

When used in the shell or exec formats, the `CMD` instruction sets the command to be executed when running the image. This is functionally equivalent to running `docker commit -run '{"Cmd": <command>}'` outside the builder.

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 *3.8 ENTRYPOINT*.

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

: Don't confuse `RUN` with `CMD`. `RUN` actually runs a command and commits the result; `CMD` does not execute anything

at build time, but specifies the intended command for the image.

3.5 EXPOSE

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

The EXPOSE instruction exposes ports for use within links. This is functionally equivalent to running docker commit -run '{"PortSpecs": ["<port>", "<port2>"]}' outside the builder. Refer to for detailed information.

3.6 ENV

```
ENV <key> <value>
```

The ENV instruction sets the environment variable <key> to the value <value>. This value will be passed to all future RUN instructions. This is functionally equivalent to prefixing the command with <key>=<value>

: The environment variables will persist when a container is run from the resulting image.

3.7 ADD

```
ADD <src> <dest>
```

The ADD instruction will copy new files from **<src>** and add them to the container's filesystem at path <dest>.

<src> must be the path to a file or directory relative to the source directory being built (also called the *context* of the build) or a remote file URL.

<dest> is the path at which the source will be copied in the destination container.

All new files and directories are created with mode 0755, uid and gid 0.

: if you build using STDIN (docker build - < somefile), there is no build context, so the Dockerfile can only contain an URL based ADD statement.

: if your URL files are protected using authentication, you will need to use an RUN wget , RUN curl or other tool from within the container as ADD does not support authentication.

The copy obeys the following rules:

- The <src> path must be inside the *context* of the build; you cannot ADD ../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 URL and <dest> does not end with a trailing slash, then a file is downloaded from the URL and copied to <dest>.
- If <src> is a URL and <dest> does end with a trailing slash, then the filename is inferred from the URL and the file is downloaded to <dest>/<filename>. For instance, ADD http://example.com/foobar / would create the file /foobar. The URL must have a nontrivial path so that an appropriate filename can be discovered in this case (http://example.com will not work).
- If <src> is a directory, the entire directory is copied, including filesystem metadata.

- If <src> is a *local* tar archive in a recognized compression format (identity, gzip, bzip2 or xz) then it is unpacked as a directory. Resources from *remote* URLs are **not** decompressed.

  When a directory is copied or unpacked, it has the same behavior as `tar -x`: the result is the union of

  1. whatever existed at the destination path and
  2. the contents of the source tree,

  with conflicts resolved in favor of "2." on a file-by-file basis.

- 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 <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.

3.8 ENTRYPOINT

ENTRYPOINT has two forms:

- `ENTRYPOINT ["executable", "param1", "param2"]` (like an *exec*, preferred form)
- `ENTRYPOINT command param1 param2` (as a *shell*)

There can only be one `ENTRYPOINT` in a Dockerfile. If you have more than one `ENTRYPOINT`, then only the last one in the Dockerfile will have an effect.

An `ENTRYPOINT` helps you to configure a container that you can run as an executable. That is, when you specify an `ENTRYPOINT`, then the whole container runs as if it was just that executable.

The `ENTRYPOINT` instruction adds an entry command that will **not** be overwritten when arguments are passed to `docker run`, unlike the behavior of `CMD`. This allows arguments to be passed to the entrypoint. i.e. `docker run <image> -d` will pass the "-d" argument to the ENTRYPOINT.

You can specify parameters either in the ENTRYPOINT JSON array (as in "like an exec" above), or by using a CMD statement. Parameters in the ENTRYPOINT will not be overridden by the `docker run` arguments, but parameters specified via CMD will be overridden by `docker run` arguments.

Like a `CMD`, you can specify a plain string for the ENTRYPOINT and it will execute in `/bin/sh -c`:

```
FROM ubuntu
ENTRYPOINT wc -l -
```

For example, that Dockerfile's image will *always* take stdin as input ("-") and print the number of lines ("-l"). If you wanted to make this optional but default, you could use a CMD:

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

3.9 VOLUME

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

The `VOLUME` instruction will create a mount point with the specified name and mark it as holding externally mounted volumes from native host or other containers. For more information/examples and mounting instructions via docker client, refer to documentation.

3.10 USER

```
USER daemon
```

The `USER` instruction sets the username or UID to use when running the image.

3.11 WORKDIR

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

The `WORKDIR` instruction sets the working directory in which the command given by `CMD` is executed.

3.11 ONBUILD

```
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 in advance instructions to run later, during the next build stage.

Here's how it works:

1. 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.
2. 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 *docker inspect*.
3. 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.
4. 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
[...]
```

5.2.4 4. Dockerfile Examples

```
# Nginx
#
# VERSION               0.0.1

FROM      ubuntu
MAINTAINER Guillaume J. Charmes <guillaume@dotcloud.com>

# make sure the package repository is up to date
RUN echo "deb http://archive.ubuntu.com/ubuntu precise main universe" > /etc/apt/sources.list
RUN apt-get update

RUN apt-get install -y inotify-tools nginx apache2 openssh-server

# Firefox over VNC
#
# VERSION               0.3

FROM ubuntu
# make sure the package repository is up to date
RUN echo "deb http://archive.ubuntu.com/ubuntu precise main universe" > /etc/apt/sources.list
RUN apt-get update

# Install vnc, xvfb in order to create a 'fake' display and firefox
RUN apt-get install -y x11vnc xvfb firefox
RUN mkdir /.vnc
# Setup a password
RUN x11vnc -storepasswd 1234 ~/.vnc/passwd
# Autostart firefox (might not be the best way, but it does the trick)
RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900
CMD    ["x11vnc", "-forever", "-usepw", "-create"]

# Multiple images example
#
# VERSION               0.1

FROM ubuntu
RUN echo foo > bar
# Will output something like ===> 907ad6c2736f

FROM ubuntu
RUN echo moo > oink
# Will output something like ===> 695d7793cbe4

# You'll now have two images, 907ad6c2736f with /bar, and 695d7793cbe4 with
# /oink.
```

5.3 Docker Run Reference

Docker runs processes in isolated containers. When an operator executes `docker run`, she starts a process with its own file system, its own networking, and its own isolated process tree. The *Image* which starts the process may define defaults related to the binary to run, the networking to expose, and more, but `docker run` gives final control

to the operator who starts the container from the image. That's the main reason *run* has more options than any other `docker` command.

Every one of the shows running containers, and so here we try to give more in-depth guidance.

Table of Contents

5.3.1 General Form

As you've seen in the , the basic *run* command takes this form:

```
docker run [OPTIONS] IMAGE[:TAG] [COMMAND] [ARG...]
```

To learn how to interpret the types of `[OPTIONS]`, see *Types of Options*.

The list of `[OPTIONS]` breaks down into two groups:

1. Settings exclusive to operators, including:
   - Detached or Foreground running,
   - Container Identification,
   - Network settings, and
   - Runtime Constraints on CPU and Memory
   - Privileges and LXC Configuration
2. Setting shared between operators and developers, where operators can override defaults developers set in images at build time.

Together, the `docker run [OPTIONS]` give complete control over runtime behavior to the operator, allowing them to override all defaults set by the developer during `docker build` and nearly all the defaults set by the Docker runtime itself.

5.3.2 Operator Exclusive Options

Only the operator (the person executing `docker run`) can set the following options.

Detached vs Foreground

When starting a Docker container, you must first decide if you want to run the container in the background in a "detached" mode or in the default foreground mode:

```
-d=false: Detached mode: Run container in the background, print new container id
```

Detached (-d)

In detached mode (`-d=true` or just `-d`), all I/O should be done through network connections or shared volumes because the container is no longer listening to the commandline where you executed `docker run`. You can reattach to a detached container with `docker` *attach*. If you choose to run a container in the detached mode, then you cannot use the `-rm` option.

Foreground

In foreground mode (the default when `-d` is not specified), `docker run` can start the process in the container and attach the console to the process's standard input, output, and standard error. It can even pretend to be a TTY (this is what most commandline executables expect) and pass along signals. All of that is configurable:

```
-a=[]           : Attach to ``stdin``, ``stdout`` and/or ``stderr``
-t=false        : Allocate a pseudo-tty
-sig-proxy=true: Proxify all received signal to the process (even in non-tty mode)
-i=false        : Keep STDIN open even if not attached
```

If you do not specify `-a` then Docker will attach everything (stdin,stdout,stderr). You can specify to which of the three standard streams (`stdin`, `stdout`, `stderr`) you'd like to connect instead, as in:

```
docker run -a stdin -a stdout -i -t ubuntu /bin/bash
```

For interactive processes (like a shell) you will typically want a tty as well as persistent standard input (`stdin`), so you'll use `-i -t` together in most interactive cases.

Container Identification

Name (-name)

The operator can identify a container in three ways:

- UUID long identifier ("f78375b1c487e03c9438c729345e54db9d20cfa2ac1fc3494b6eb60872e74778")
- UUID short identifier ("f78375b1c487")
- Name ("evil_ptolemy")

The UUID identifiers come from the Docker daemon, and if you do not assign a name to the container with `-name` then the daemon will also generate a random string name too. The name can become a handy way to add meaning to a container since you can use this name when defining *links* (or any other place you need to identify a container). This works for both background and foreground Docker containers.

PID Equivalent

And finally, to help with automation, you can have Docker write the container ID out to a file of your choosing. This is similar to how some programs might write out their process ID to a file (you've seen them as PID files):

```
-cidfile="": Write the container ID to the file
```

Network Settings

:: -n=true : Enable networking for this container -dns=[] : Set custom dns servers for the container

By default, all containers have networking enabled and they can make any outgoing connections. The operator can completely disable networking with `docker run -n` which disables all incoming and outgoing networking. In cases like this, you would perform I/O through files or STDIN/STDOUT only.

Your container will use the same DNS servers as the host by default, but you can override this with `-dns`.

Clean Up (-rm)

By default a container's file system persists even after the container exits. This makes debugging a lot easier (since you can inspect the final state) and you retain all your data by default. But if you are running short-term **foreground** processes, these container file systems can really pile up. If instead you'd like Docker to **automatically clean up the container and remove the file system when the container exits**, you can add the `-rm` flag:

```
-rm=false: Automatically remove the container when it exits (incompatible with -d)
```

Runtime Constraints on CPU and Memory

The operator can also adjust the performance parameters of the container:

```
-m="": Memory limit (format: <number><optional unit>, where unit = b, k, m or g)
-c=0 : CPU shares (relative weight)
```

The operator can constrain the memory available to a container easily with `docker run -m`. If the host supports swap memory, then the `-m` memory setting can be larger than physical RAM.

Similarly the operator can increase the priority of this container with the `-c` option. By default, all containers run at the same priority and get the same proportion of CPU cycles, but you can tell the kernel to give more shares of CPU time to one or more containers when you start them via Docker.

Runtime Privilege and LXC Configuration

```
-privileged=false: Give extended privileges to this container
-lxc-conf=[]: Add custom lxc options -lxc-conf="lxc.cgroup.cpuset.cpus = 0,1"
```

By default, Docker containers are "unprivileged" and cannot, for example, run a Docker daemon inside a Docker container. This is because by default a container is not allowed to access any devices, but a "privileged" container is given access to all devices (see lxc-template.go and documentation on cgroups devices).

When the operator executes `docker run -privileged`, Docker will enable to access to all devices on the host as well as set some configuration in AppArmor to allow the container nearly all the same access to the host as processes running outside containers on the host. Additional information about running with `-privileged` is available on the Docker Blog.

An operator can also specify LXC options using one or more `-lxc-conf` parameters. These can be new parameters or override existing parameters from the lxc-template.go. Note that in the future, a given host's Docker daemon may not use LXC, so this is an implementation-specific configuration meant for operators already familiar with using LXC directly.

5.3.3 Overriding `Dockerfile` Image Defaults

When a developer builds an image from a *Dockerfile* or when she commits it, the developer can set a number of default parameters that take effect when the image starts up as a container.

Four of the `Dockerfile` commands cannot be overridden at runtime: `FROM, MAINTAINER, RUN`, and `ADD`. Everything else has a corresponding override in `docker run`. We'll go through what the developer might have set in each `Dockerfile` instruction and how the operator can override that setting.

CMD (Default Command or Options)

Recall the optional `COMMAND` in the Docker commandline:

```
docker run [OPTIONS] IMAGE[:TAG] [COMMAND] [ARG...]
```

This command is optional because the person who created the `IMAGE` may have already provided a default `COMMAND` using the `Dockerfile` `CMD`. As the operator (the person running a container from the image), you can override that `CMD` just by specifying a new `COMMAND`.

If the image also specifies an `ENTRYPOINT` then the `CMD` or `COMMAND` get appended as arguments to the `ENTRYPOINT`.

ENTRYPOINT (Default Command to Execute at Runtime

```
-entrypoint="": Overwrite the default entrypoint set by the image
```

The ENTRYPOINT of an image is similar to a `COMMAND` because it specifies what executable to run when the container starts, but it is (purposely) more difficult to override. The `ENTRYPOINT` gives a container its default nature or behavior, so that when you set an `ENTRYPOINT` you can run the container *as if it were that binary*, complete with default options, and you can pass in more options via the `COMMAND`. But, sometimes an operator may want to run something else inside the container, so you can override the default `ENTRYPOINT` at runtime by using a string to specify the new `ENTRYPOINT`. Here is an example of how to run a shell in a container that has been set up to automatically run something else (like `/usr/bin/redis-server`):

```
docker run -i -t -entrypoint /bin/bash example/redis
```

or two examples of how to pass more parameters to that ENTRYPOINT:

```
docker run -i -t -entrypoint /bin/bash example/redis -c ls -l
docker run -i -t -entrypoint /usr/bin/redis-cli example/redis --help
```

EXPOSE (Incoming Ports)

The `Dockerfile` doesn't give much control over networking, only providing the `EXPOSE` instruction to give a hint to the operator about what incoming ports might provide services. The following options work with or override the

Dockerfile's exposed defaults:

```
-expose=[]: Expose a port from the container
            without publishing it to your host
-P=false  : Publish all exposed ports to the host interfaces
-p=[]     : Publish a container's port to the host (format:
            ip:hostPort:containerPort | ip::containerPort |
            hostPort:containerPort)
            (use 'docker port' to see the actual mapping)
-link=""  : Add link to another container (name:alias)
```

As mentioned previously, EXPOSE (and -expose) make a port available **in** a container for incoming connections. The port number on the inside of the container (where the service listens) does not need to be the same number as the port exposed on the outside of the container (where clients connect), so inside the container you might have an HTTP service listening on port 80 (and so you EXPOSE 80 in the Dockerfile), but outside the container the port might be 42800.

To help a new client container reach the server container's internal port operator -expose'd by the operator or EXPOSE'd by the developer, the operator has three choices: start the server container with -P or -p, or start the client container with -link.

If the operator uses -P or -p then Docker will make the exposed port accessible on the host and the ports will be available to any client that can reach the host. To find the map between the host ports and the exposed ports, use docker port)

If the operator uses -link when starting the new client container, then the client container can access the exposed port via a private networking interface. Docker will set some environment variables in the client container to help indicate which interface and port to use.

ENV (Environment Variables)

The operator can **set any environment variable** in the container by using one or more -e flags, even overriding those already defined by the developer with a Dockefile ENV:

```
$ docker run -e "deep=purple" -rm ubuntu /bin/bash -c export
declare -x HOME="/"
declare -x HOSTNAME="85bc26a0e200"
declare -x OLDPWD
declare -x PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
declare -x PWD="/"
declare -x SHLVL="1"
declare -x container="lxc"
declare -x deep="purple"
```

Similarly the operator can set the **hostname** with -h.

-link name:alias also sets environment variables, using the *alias* string to define environment variables within the container that give the IP and PORT information for connecting to the service container. Let's imagine we have a container running Redis:

```
# Start the service container, named redis-name
$ docker run -d -name redis-name dockerfiles/redis
4241164edf6f5aca5b0e9e4c9eccd899b0b8080c64c0cd26efe02166c73208f3

# The redis-name container exposed port 6379
$ docker ps
CONTAINER ID        IMAGE                       COMMAND                CREATED             STATUS
4241164edf6f        dockerfiles/redis:latest    /redis-stable/src/re   5 seconds ago       Up 4 second
```

```
# Note that there are no public ports exposed since we didn't use -p or -P
$ docker port 4241164edf6f 6379
2014/01/25 00:55:38 Error: No public port '6379' published for 4241164edf6f
```

Yet we can get information about the Redis container's exposed ports with `-link`. Choose an alias that will form a valid environment variable!

```
$ docker run -rm -link redis-name:redis_alias -entrypoint /bin/bash dockerfiles/redis -c export
declare -x HOME="/"
declare -x HOSTNAME="acda7f7b1cdc"
declare -x OLDPWD
declare -x PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
declare -x PWD="/"
declare -x REDIS_ALIAS_NAME="/distracted_wright/redis"
declare -x REDIS_ALIAS_PORT="tcp://172.17.0.32:6379"
declare -x REDIS_ALIAS_PORT_6379_TCP="tcp://172.17.0.32:6379"
declare -x REDIS_ALIAS_PORT_6379_TCP_ADDR="172.17.0.32"
declare -x REDIS_ALIAS_PORT_6379_TCP_PORT="6379"
declare -x REDIS_ALIAS_PORT_6379_TCP_PROTO="tcp"
declare -x SHLVL="1"
declare -x container="lxc"
```

And we can use that information to connect from another container as a client:

```
$ docker run -i -t -rm -link redis-name:redis_alias -entrypoint /bin/bash dockerfiles/redis -c '/redi
172.17.0.32:6379>
```

VOLUME (Shared Filesystems)

```
-v=[]: Create a bind mount with: [host-dir]:[container-dir]:[rw|ro].
       If "container-dir" is missing, then docker creates a new volume.
-volumes-from="": Mount all volumes from the given container(s)
```

The volumes commands are complex enough to have their own documentation in section . A developer can define one or more `VOLUME`s associated with an image, but only the operator can give access from one container to another (or from a container to a volume mounted on the host).

USER

The default user within a container is `root` (id = 0), but if the developer created additional users, those are accessible too. The developer can set a default user to run the first process with the `Dockerfile USER` command, but the operator can override it

```
-u="": Username or UID
```

WORKDIR

The default working directory for running binaries within a container is the root directory (`/`), but the developer can set a different default with the `Dockerfile WORKDIR` command. The operator can override this with:

```
-w="": Working directory inside the container
```

5.4 APIs

Your programs and scripts can access Docker's functionality via these interfaces:

5.4.1 Registry & Index Spec

1. The 3 roles

1.1 Index

The Index is responsible for centralizing information about:

- User accounts
- Checksums of the images
- Public namespaces

The Index has different components:

- Web UI
- Meta-data store (comments, stars, list public repositories)
- Authentication service
- Tokenization

The index is authoritative for those information.

We expect that there will be only one instance of the index, run and managed by Docker Inc.

1.2 Registry

- It stores the images and the graph for a set of repositories
- It does not have user accounts data
- It has no notion of user accounts or authorization
- It delegates authentication and authorization to the Index Auth service using tokens
- It supports different storage backends (S3, cloud files, local FS)
- It doesn't have a local database
- Source Code

We expect that there will be multiple registries out there. To help to grasp the context, here are some examples of registries:

- **sponsor registry**: such a registry is provided by a third-party hosting infrastructure as a convenience for their customers and the docker community as a whole. Its costs are supported by the third party, but the management and operation of the registry are supported by dotCloud. It features read/write access, and delegates authentication and authorization to the Index.
- **mirror registry**: such a registry is provided by a third-party hosting infrastructure but is targeted at their customers only. Some mechanism (unspecified to date) ensures that public images are pulled from a sponsor registry to the mirror registry, to make sure that the customers of the third-party provider can "docker pull" those images locally.

- **vendor registry**: such a registry is provided by a software vendor, who wants to distribute docker images. It would be operated and managed by the vendor. Only users authorized by the vendor would be able to get write access. Some images would be public (accessible for anyone), others private (accessible only for authorized users). Authentication and authorization would be delegated to the Index. The goal of vendor registries is to let someone do "docker pull basho/riak1.3" and automatically push from the vendor registry (instead of a sponsor registry); i.e. get all the convenience of a sponsor registry, while retaining control on the asset distribution.
- **private registry**: such a registry is located behind a firewall, or protected by an additional security layer (HTTP authorization, SSL client-side certificates, IP address authorization...). The registry is operated by a private entity, outside of dotCloud's control. It can optionally delegate additional authorization to the Index, but it is not mandatory.

:

The latter implies that while HTTP is the protocol of choice for a registry, multiple schemes are possible (and in some cases, triv

- HTTP with GET (and PUT for read-write registries);
- local mount point;
- remote docker addressed through SSH.

The latter would only require two new commands in docker, e.g. `registryget` and `registryput`, wrapping access to the local filesystem (and optionally doing consistency checks). Authentication and authorization are then delegated to SSH (e.g. with public keys).

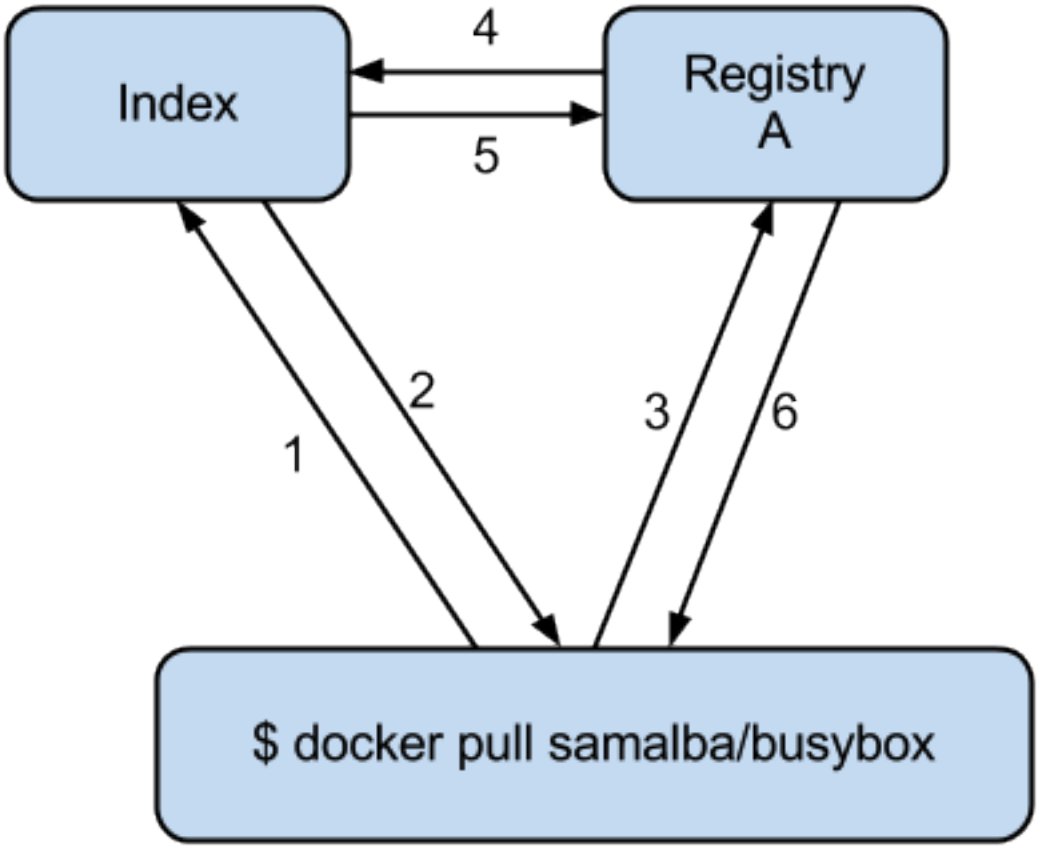
1.3 Docker

On top of being a runtime for LXC, Docker is the Registry client. It supports:

- Push / Pull on the registry
- Client authentication on the Index

2. Workflow

2.1 Pull



1. Contact the Index to know where I should download "samalba/busybox"
2. Index replies: a. `samalba/busybox` is on Registry A b. here are the checksums for `samalba/busybox` (for all layers) c. token
3. Contact Registry A to receive the layers for `samalba/busybox` (all of them to the base image). Registry A is authoritative for "samalba/busybox" but keeps a copy of all inherited layers and serve them all from the same location.
4. registry contacts index to verify if token/user is allowed to download images
5. Index returns true/false lettings registry know if it should proceed or error out
6. Get the payload for all layers

It's possible to run:

```
docker pull https://<registry>/repositories/samalba/busybox
```

In this case, Docker bypasses the Index. However the security is not guaranteed (in case Registry A is corrupted) because there won't be any checksum checks.

Currently registry redirects to s3 urls for downloads, going forward all downloads need to be streamed through the registry. The Registry will then abstract the calls to S3 by a top-level class which implements sub-classes for S3 and

local storage.

Token is only returned when the `X-Docker-Token` header is sent with request.

Basic Auth is required to pull private repos. Basic auth isn't required for pulling public repos, but if one is provided, it needs to be valid and for an active account.

API (pulling repository foo/bar):

1. **(Docker -> Index) GET /v1/repositories/foo/bar/images**

   Headers: Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ== X-Docker-Token: true

   Action: (looking up the foo/bar in db and gets images and checksums for that repo (all if no tag is specified, if tag, only checksums for those tags) see part 4.4.1)

2. (Index -> Docker) HTTP 200 OK

   Headers:

   - Authorization: Token signature=123abc,repository="foo/bar",access=write
   - X-Docker-Endpoints: registry.docker.io [, registry2.docker.io]

   Body: Jsonified checksums (see part 4.4.1)

3. **(Docker -> Registry) GET /v1/repositories/foo/bar/tags/latest**

   Headers: Authorization: Token signature=123abc,repository="foo/bar",access=write

4. (Registry -> Index) GET /v1/repositories/foo/bar/images

   Headers: Authorization: Token signature=123abc,repository="foo/bar",access=read

   Body: <ids and checksums in payload>

   Action: (Lookup token see if they have access to pull.)

   If good: HTTP 200 OK Index will invalidate the token

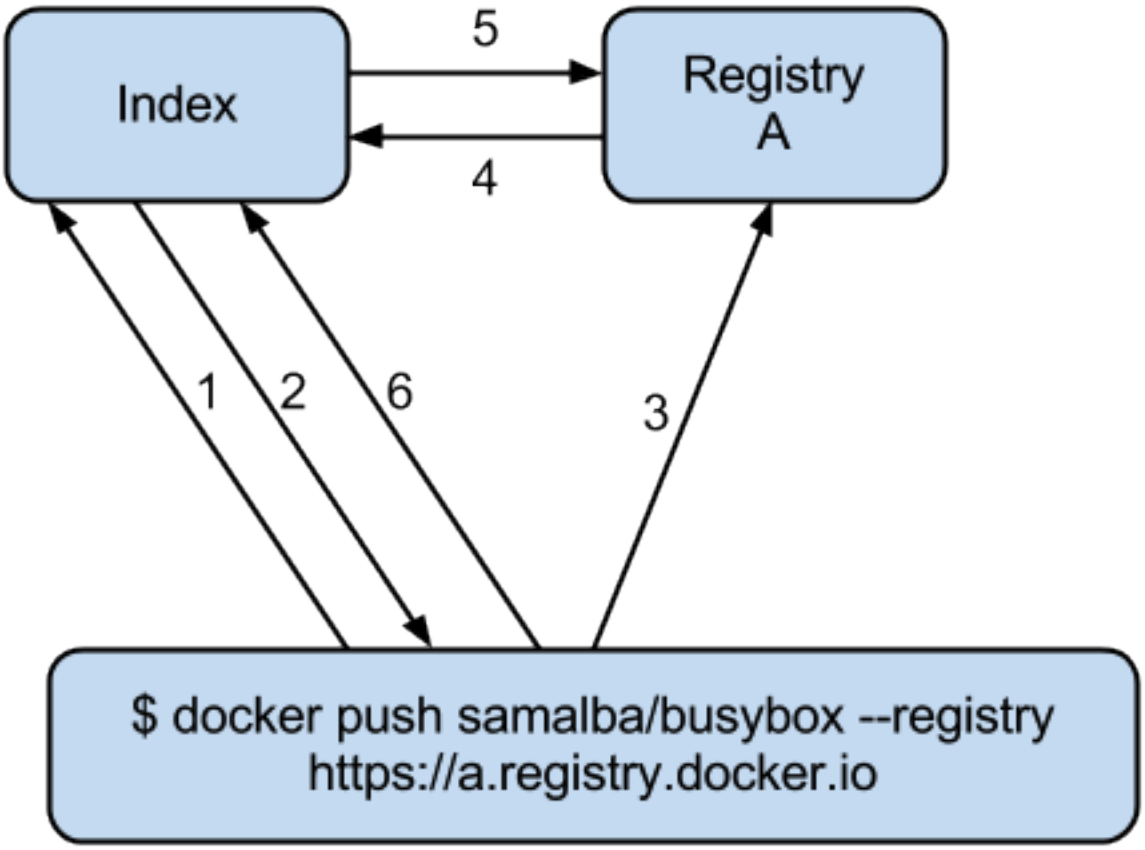
   If bad: HTTP 401 Unauthorized

5. **(Docker -> Registry) GET /v1/images/928374982374/ancestry**

   Action: (for each image id returned in the registry, fetch /json + /layer)

: If someone makes a second request, then we will always give a new token, never reuse tokens.

2.2 Push



1. Contact the index to allocate the repository name "samalba/busybox" (authentication required with user credentials)
2. If authentication works and namespace available, "samalba/busybox" is allocated and a temporary token is returned (namespace is marked as initialized in index)
3. Push the image on the registry (along with the token)
4. Registry A contacts the Index to verify the token (token must corresponds to the repository name)
5. Index validates the token. Registry A starts reading the stream pushed by docker and store the repository (with its images)
6. docker contacts the index to give checksums for upload images

---

: It's possible not to use the Index at all! In this case, a deployed version of the Registry is deployed to store and serve images. Those images are not authenticated and the security is not guaranteed.

: Index can be replaced! For a private Registry deployed, a custom Index can be used to serve and validate token according to different policies.

Docker computes the checksums and submit them to the Index at the end of the push. When a repository name does not have checksums on the Index, it means that the push is in progress (since checksums are submitted at the end).

API (pushing repos foo/bar):

1. **(Docker -> Index) PUT /v1/repositories/foo/bar/**

   Headers: Authorization: Basic sdkjfskdjfhsdkjfh== X-Docker-Token: true

   Action::

   - in index, we allocated a new repository, and set to initialized

   Body:: (The body contains the list of images that are going to be pushed, with empty checksums. The checksums will be set at the end of the push):

   ```
   [{"id": "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f"}]
   ```

2. **(Index -> Docker) 200 Created**

   Headers:

   - WWW-Authenticate: Token signature=123abc,repository="foo/bar",access=write
   - X-Docker-Endpoints: registry.docker.io [, registry2.docker.io]

3. **(Docker -> Registry) PUT /v1/images/98765432_parent/json**

   Headers: Authorization: Token signature=123abc,repository="foo/bar",access=write

4. **(Registry->Index) GET /v1/repositories/foo/bar/images**

   Headers: Authorization: Token signature=123abc,repository="foo/bar",access=write

   Action::

   - **Index:** will invalidate the token.
   - **Registry:** grants a session (if token is approved) and fetches the images id

5. **(Docker -> Registry) PUT /v1/images/98765432_parent/json**

   Headers::

   - Authorization: Token signature=123abc,repository="foo/bar",access=write
   - Cookie: (Cookie provided by the Registry)

6. **(Docker -> Registry) PUT /v1/images/98765432/json**

   Headers: Cookie: (Cookie provided by the Registry)

7. **(Docker -> Registry) PUT /v1/images/98765432_parent/layer**

   Headers: Cookie: (Cookie provided by the Registry)

8. **(Docker -> Registry) PUT /v1/images/98765432/layer**

   Headers: X-Docker-Checksum: sha256:436745873465fdjkhdfjkgh

9. **(Docker -> Registry) PUT /v1/repositories/foo/bar/tags/latest**

   Headers: Cookie: (Cookie provided by the Registry)

   Body: "98765432"

10. (Docker -> Index) PUT /v1/repositories/foo/bar/images

    Headers: Authorization: Basic 123oislifjsldfj== X-Docker-Endpoints: registry1.docker.io (no validation on this right now)

Body: (The image, id's, tags and checksums)

[{"id": "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f", "checksum": "b486531f9a779a0c17e3ed29dae8f12c4f9e89cc6f0bc3c38722009fe6857087"}]

Return HTTP 204

: If push fails and they need to start again, what happens in the index, there will already be a record for the namespace/name, but it will be initialized. Should we allow it, or mark as name already used? One edge case could be if someone pushes the same thing at the same time with two different shells.

If it's a retry on the Registry, Docker has a cookie (provided by the registry after token validation). So the Index won't have to provide a new token.

2.3 Delete

If you need to delete something from the index or registry, we need a nice clean way to do that. Here is the workflow.

1. Docker contacts the index to request a delete of a repository `samalba/busybox` (authentication required with user credentials)
2. If authentication works and repository is valid, `samalba/busybox` is marked as deleted and a temporary token is returned
3. Send a delete request to the registry for the repository (along with the token)
4. Registry A contacts the Index to verify the token (token must corresponds to the repository name)
5. Index validates the token. Registry A deletes the repository and everything associated to it.
6. docker contacts the index to let it know it was removed from the registry, the index removes all records from the database.

: The Docker client should present an "Are you sure?" prompt to confirm the deletion before starting the process. Once it starts it can't be undone.

API (deleting repository foo/bar):

1. **(Docker -> Index) DELETE /v1/repositories/foo/bar/**

   Headers: Authorization: Basic sdkjfskdjfhsdkjfh== X-Docker-Token: true

   Action::

   - in index, we make sure it is a valid repository, and set to deleted (logically)

   Body:: Empty

2. **(Index -> Docker) 202 Accepted**

   Headers:

   - WWW-Authenticate: Token signature=123abc,repository="foo/bar",access=delete
   - X-Docker-Endpoints: registry.docker.io [, registry2.docker.io] # list of endpoints where this repo lives.

3. **(Docker -> Registry) DELETE /v1/repositories/foo/bar/**

   Headers: Authorization: Token signature=123abc,repository="foo/bar",access=delete

4. **(Registry->Index) PUT /v1/repositories/foo/bar/auth**

Headers: Authorization: Token signature=123abc,repository="foo/bar",access=delete

Action::

- **Index:** will invalidate the token.
- **Registry:** deletes the repository (if token is approved)

5. **(Registry -> Docker) 200 OK** 200 If success 403 if forbidden 400 if bad request 404 if repository isn't found
6. (Docker -> Index) DELETE /v1/repositories/foo/bar/

   Headers: Authorization: Basic 123oislifjsldfj== X-Docker-Endpoints: registry-1.docker.io (no validation on this right now)

   Body: Empty

   Return HTTP 200

3. How to use the Registry in standalone mode

The Index has two main purposes (along with its fancy social features):

- **Resolve short names (to avoid passing absolute URLs all the time)**
  - username/projectname -> https://registry.docker.io/users/<username>/repositories/<projectname>/
  - team/projectname -> https://registry.docker.io/team/<team>/repositories/<projectname>/
- Authenticate a user as a repos owner (for a central referenced repository)

3.1 Without an Index

Using the Registry without the Index can be useful to store the images on a private network without having to rely on an external entity controlled by Docker Inc.

In this case, the registry will be launched in a special mode (–standalone? –no-index?). In this mode, the only thing which changes is that Registry will never contact the Index to verify a token. It will be the Registry owner responsibility to authenticate the user who pushes (or even pulls) an image using any mechanism (HTTP auth, IP based, etc...).

In this scenario, the Registry is responsible for the security in case of data corruption since the checksums are not delivered by a trusted entity.

As hinted previously, a standalone registry can also be implemented by any HTTP server handling GET/PUT requests (or even only GET requests if no write access is necessary).

3.2 With an Index

The Index data needed by the Registry are simple:

- Serve the checksums
- Provide and authorize a Token

In the scenario of a Registry running on a private network with the need of centralizing and authorizing, it's easy to use a custom Index.

The only challenge will be to tell Docker to contact (and trust) this custom Index. Docker will be configurable at some point to use a specific Index, it'll be the private entity responsibility (basically the organization who uses Docker in a private environment) to maintain the Index and the Docker's configuration among its consumers.

4. The API

The first version of the api is available here: https://github.com/jpetazzo/docker/blob/acd51ecea8f5d3c02b00a08176171c59442df8b3/doc
repositories-push-pull.md

4.1 Images

The format returned in the images is not defined here (for layer and JSON), basically because Registry stores exactly the same kind of information as Docker uses to manage them.

The format of ancestry is a line-separated list of image ids, in age order, i.e. the image's parent is on the last line, the parent of the parent on the next-to-last line, etc.; if the image has no parent, the file is empty.

```
GET /v1/images/<image_id>/layer
PUT /v1/images/<image_id>/layer
GET /v1/images/<image_id>/json
PUT /v1/images/<image_id>/json
GET /v1/images/<image_id>/ancestry
PUT /v1/images/<image_id>/ancestry
```

4.2 Users

4.2.1 Create a user (Index) POST /v1/users

Body: {"email": "sam@dotcloud.com", "password": "toto42", "username": "foobar"'}

Validation:

- **username**: min 4 character, max 30 characters, must match the regular expression [a-z0-9_].
- **password**: min 5 characters

Valid: return HTTP 200

Errors: HTTP 400 (we should create error codes for possible errors) - invalid json - missing field - wrong format (username, password, email, etc) - forbidden name - name already exists

: A user account will be valid only if the email has been validated (a validation link is sent to the email address).

4.2.2 Update a user (Index) PUT /v1/users/<username>

Body: {"password": "toto"}

: We can also update email address, if they do, they will need to reverify their new email address.

4.2.3 Login (Index) Does nothing else but asking for a user authentication. Can be used to validate credentials. HTTP Basic Auth for now, maybe change in future.

GET /v1/users

Return:

- Valid: HTTP 200
- Invalid login: HTTP 401
- Account inactive: HTTP 403 Account is not Active

4.3 Tags (Registry)

The Registry does not know anything about users. Even though repositories are under usernames, it's just a namespace for the registry. Allowing us to implement organizations or different namespaces per user later, without modifying the Registry's API.

The following naming restrictions apply:

- Namespaces must match the same regular expression as usernames (See 4.2.1.)
- Repository names must match the regular expression [a-zA-Z0-9-_.]

4.3.1 Get all tags

GET /v1/repositories/<namespace>/<repository_name>/tags

Return: HTTP 200 { "latest": "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f", "0.1.1": "b486531f9a779a0c17e3ed29dae8f12c4f9e89cc6f0bc3c38722009fe6857087" }

4.3.2 Read the content of a tag (resolve the image id)

GET /v1/repositories/<namespace>/<repo_name>/tags/<tag>

Return: "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f"

4.3.3 Delete a tag (registry)

DELETE /v1/repositories/<namespace>/<repo_name>/tags/<tag>

4.4 Images (Index)

For the Index to "resolve" the repository name to a Registry location, it uses the X-Docker-Endpoints header. In other terms, this requests always add a `X-Docker-Endpoints` to indicate the location of the registry which hosts this repository.

4.4.1 Get the images

GET /v1/repositories/<namespace>/<repo_name>/images

Return: HTTP 200 [{"id": "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f", "checksum": "md5:b486531f9a779a0c17e3ed29dae8f12c4f9e89cc6f0bc3c38722009fe6857087"}]

4.4.2 Add/update the images

You always add images, you never remove them.

PUT /v1/repositories/<namespace>/<repo_name>/images

Body: [{"id": "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f", "checksum": "sha256:b486531f9a779a0c17e3ed29dae8f12c4f9e89cc6f0bc3c38722009fe6857087"}]

Return 204

4.5 Repositories

4.5.1 Remove a Repository (Registry)

DELETE /v1/repositories/<namespace>/<repo_name>

Return 200 OK

4.5.2 Remove a Repository (Index)

This starts the delete process. see 2.3 for more details.

DELETE /v1/repositories/<namespace>/<repo_name>

Return 202 OK

5. Chaining Registries

It's possible to chain Registries server for several reasons:

- Load balancing
- Delegate the next request to another server

When a Registry is a reference for a repository, it should host the entire images chain in order to avoid breaking the chain during the download.

The Index and Registry use this mechanism to redirect on one or the other.

Example with an image download:

On every request, a special header can be returned:

```
X-Docker-Endpoints: server1,server2
```

On the next request, the client will always pick a server from this list.

6. Authentication & Authorization

6.1 On the Index

The Index supports both "Basic" and "Token" challenges. Usually when there is a `401 Unauthorized`, the Index replies this:

```
401 Unauthorized
WWW-Authenticate: Basic realm="auth required",Token
```

You have 3 options:

1. Provide user credentials and ask for a token

   Header:

   - Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
   - X-Docker-Token: true

   In this case, along with the 200 response, you'll get a new token (if user auth is ok): If authorization isn't correct you get a 401 response. If account isn't active you will get a 403 response.

   Response:

   - 200 OK
   - X-Docker-Token: Token signature=123abc,repository="foo/bar",access=read

2. Provide user credentials only

   Header: Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==

3. Provide Token

   Header: Authorization: Token signature=123abc,repository="foo/bar",access=read

6.2 On the Registry

The Registry only supports the Token challenge:

```
401 Unauthorized
WWW-Authenticate: Token
```

The only way is to provide a token on `401 Unauthorized` responses:

```
Authorization: Token signature=123abc,repository="foo/bar",access=read
```

Usually, the Registry provides a Cookie when a Token verification succeeded. Every time the Registry passes a Cookie, you have to pass it back the same cookie.:

```
200 OK
Set-Cookie: session="wD/J7LqL5ctqw8haL10vgfhrb2Q=?foo=UydiYXInCnAxCi4=&timestamp=RjEzNjYzMTQ5NDcuNDc(
```

Next request:

```
GET /(...)
Cookie: session="wD/J7LqL5ctqw8haL10vgfhrb2Q=?foo=UydiYXInCnAxCi4=&timestamp=RjEzNjYzMTQ5NDcuNDc0NjQz
```

7 Document Version

- 1.0 : May 6th 2013 : initial release
- 1.1 : June 1st 2013 : Added Delete Repository and way to handle new source namespace.

5.4.2 Docker Registry API

1. Brief introduction

- This is the REST API for the Docker Registry
- It stores the images and the graph for a set of repositories
- It does not have user accounts data
- It has no notion of user accounts or authorization
- It delegates authentication and authorization to the Index Auth service using tokens
- It supports different storage backends (S3, cloud files, local FS)
- It doesn't have a local database
- It will be open-sourced at some point

We expect that there will be multiple registries out there. To help to grasp the context, here are some examples of registries:

- **sponsor registry**: such a registry is provided by a third-party hosting infrastructure as a convenience for their customers and the docker community as a whole. Its costs are supported by the third party, but the management and operation of the registry are supported by dotCloud. It features read/write access, and delegates authentication and authorization to the Index.
- **mirror registry**: such a registry is provided by a third-party hosting infrastructure but is targeted at their customers only. Some mechanism (unspecified to date) ensures that public images are pulled from a sponsor registry to the mirror registry, to make sure that the customers of the third-party provider can "docker pull" those images locally.
- **vendor registry**: such a registry is provided by a software vendor, who wants to distribute docker images. It would be operated and managed by the vendor. Only users authorized by the vendor would be able to get write access. Some images would be public (accessible for anyone), others private (accessible only for authorized

users). Authentication and authorization would be delegated to the Index. The goal of vendor registries is to let someone do "docker pull basho/riak1.3" and automatically push from the vendor registry (instead of a sponsor registry); i.e. get all the convenience of a sponsor registry, while retaining control on the asset distribution.

- **private registry**: such a registry is located behind a firewall, or protected by an additional security layer (HTTP authorization, SSL client-side certificates, IP address authorization...). The registry is operated by a private entity, outside of dotCloud's control. It can optionally delegate additional authorization to the Index, but it is not mandatory.

: Mirror registries and private registries which do not use the Index don't even need to run the registry code. They can be implemented by any kind of transport implementing HTTP GET and PUT. Read-only registries can be powered by a simple static HTTP server.

:

The latter implies that while HTTP is the protocol of choice for a registry, multiple schemes are possible (and in some cases, triv

- HTTP with GET (and PUT for read-write registries);
- local mount point;
- remote docker addressed through SSH.

The latter would only require two new commands in docker, e.g. `registryget` and `registryput`, wrapping access to the local filesystem (and optionally doing consistency checks). Authentication and authorization are then delegated to SSH (e.g. with public keys).

2. Endpoints

2.1 Images

Layer

GET /v1/images/(*image_id*)**/layer**

get image layer for a given `image_id`

Example Request:

```
GET /v1/images/088b4505aa3adc3d35e79c031fa126b403200f02f51920fbd9b7c503e87c7a2c/layer HTTP/1.1
Host: registry-1.docker.io
Accept: application/json
Content-Type: application/json
Authorization: Token signature=123abc,repository="foo/bar",access=read
```

Parameters

- **image_id** – the id for the layer you want to get

Example Response:

```
HTTP/1.1 200
Vary: Accept
X-Docker-Registry-Version: 0.6.0
Cookie: (Cookie provided by the Registry)

{layer binary data stream}
```

Status Codes

- **200** – OK
- **401** – Requires authorization
- **404** – Image not found

PUT /v1/images/(*image_id*)**/layer**

put image layer for a given `image_id`

Example Request:

```
PUT /v1/images/088b4505aa3adc3d35e79c031fa126b403200f02f51920fbd9b7c503e87c7a2c/layer HTTP/1.1
Host: registry-1.docker.io
Transfer-Encoding: chunked
Authorization: Token signature=123abc,repository="foo/bar",access=write

{layer binary data stream}
```

Parameters

- **image_id** – the id for the layer you want to get

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json
X-Docker-Registry-Version: 0.6.0

""
```

Status Codes

- **200** – OK
- **401** – Requires authorization
- **404** – Image not found

Image

PUT /v1/images/(*image_id*)**/json**

put image for a given `image_id`

Example Request:

```
PUT /v1/images/088b4505aa3adc3d35e79c031fa126b403200f02f51920fbd9b7c503e87c7a2c/json HTTP/1.1
Host: registry-1.docker.io
Accept: application/json
Content-Type: application/json
Cookie: (Cookie provided by the Registry)

{
    id: "088b4505aa3adc3d35e79c031fa126b403200f02f51920fbd9b7c503e87c7a2c",
    parent: "aeee6396d62273d180a49c96c62e45438d87c7da4a5cf5d2be6bee4e21bc226f",
    created: "2013-04-30T17:46:10.843673+03:00",
    container: "8305672a76cc5e3d168f97221106ced35a76ec7ddbb03209b0f0d96bf74f6ef7",
    container_config: {
        Hostname: "host-test",
        User: "",
```

```
            Memory: 0,
            MemorySwap: 0,
            AttachStdin: false,
            AttachStdout: false,
            AttachStderr: false,
            PortSpecs: null,
            Tty: false,
            OpenStdin: false,
            StdinOnce: false,
            Env: null,
            Cmd: [
            "/bin/bash",
            "-c",
            "apt-get -q -yy -f install libevent-dev"
            ],
            Dns: null,
            Image: "imagename/blah",
            Volumes: { },
            VolumesFrom: ""
        },
        docker_version: "0.1.7"
    }
```

Parameters

- **image_id** – the id for the layer you want to get

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json
X-Docker-Registry-Version: 0.6.0

""
```

Status Codes

- **200** – OK
- **401** – Requires authorization

GET /v1/images/(*image_id*)**/json**

get image for a given `image_id`

Example Request:

```
GET /v1/images/088b4505aa3adc3d35e79c031fa126b403200f02f51920fbd9b7c503e87c7a2c/json HTTP/1.1
Host: registry-1.docker.io
Accept: application/json
Content-Type: application/json
Cookie: (Cookie provided by the Registry)
```

Parameters

- **image_id** – the id for the layer you want to get

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json
X-Docker-Registry-Version: 0.6.0
X-Docker-Size: 456789
X-Docker-Checksum: b486531f9a779a0c17e3ed29dae8f12c4f9e89cc6f0bc3c38722009fe6857087

{
    id: "088b4505aa3adc3d35e79c031fa126b403200f02f51920fbd9b7c503e87c7a2c",
    parent: "aeee6396d62273d180a49c96c62e45438d87c7da4a5cf5d2be6bee4e21bc226f",
    created: "2013-04-30T17:46:10.843673+03:00",
    container: "8305672a76cc5e3d168f97221106ced35a76ec7ddbb03209b0f0d96bf74f6ef7",
    container_config: {
        Hostname: "host-test",
        User: "",
        Memory: 0,
        MemorySwap: 0,
        AttachStdin: false,
        AttachStdout: false,
        AttachStderr: false,
        PortSpecs: null,
        Tty: false,
        OpenStdin: false,
        StdinOnce: false,
        Env: null,
        Cmd: [
        "/bin/bash",
        "-c",
        "apt-get -q -yy -f install libevent-dev"
        ],
        Dns: null,
        Image: "imagename/blah",
        Volumes: { },
        VolumesFrom: ""
    },
    docker_version: "0.1.7"
}
```

Status Codes

- **200** – OK
- **401** – Requires authorization
- **404** – Image not found

Ancestry

GET /v1/images/(*image_id*)**/ancestry**

get ancestry for an image given an `image_id`

Example Request:

```
GET /v1/images/088b4505aa3adc3d35e79c031fa126b403200f02f51920fbd9b7c503e87c7a2c/ancestry HTTP/1.
Host: registry-1.docker.io
Accept: application/json
Content-Type: application/json
Cookie: (Cookie provided by the Registry)
```

Parameters

- **image_id** – the id for the layer you want to get

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json
X-Docker-Registry-Version: 0.6.0

["088b4502f51920fbd9b7c503e87c7a2c05aa3adc3d35e79c031fa126b403200f",
 "aeee63968d87c7da4a5cf5d2be6bee4e21bc226fd62273d180a49c96c62e4543",
 "bfa4c5326bc764280b0863b46a4b20d940bc1897ef9c1dfec060604bdc383280",
 "6ab5893c6927c15a15665191f2c6cf751f5056d8b95ceee32e43c5e8a3648544"]
```

Status Codes

- **200** – OK
- **401** – Requires authorization
- **404** – Image not found

2.2 Tags

GET /v1/repositories/(*namespace*)**/**
*repository***/tags** get all of the tags for the given repo.

Example Request:

```
GET /v1/repositories/foo/bar/tags HTTP/1.1
Host: registry-1.docker.io
Accept: application/json
Content-Type: application/json
X-Docker-Registry-Version: 0.6.0
Cookie: (Cookie provided by the Registry)
```

Parameters

- **namespace** – namespace for the repo
- **repository** – name for the repo

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json
X-Docker-Registry-Version: 0.6.0

{
    "latest": "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f",
    "0.1.1":  "b486531f9a779a0c17e3ed29dae8f12c4f9e89cc6f0bc3c38722009fe6857087"
}
```

Status Codes

- **200** – OK

- **401** – Requires authorization
- **404** – Repository not found

GET /v1/repositories/(*namespace*)**/***repository*/**tags/***tag* get a tag for the given repo.

Example Request:

```
GET /v1/repositories/foo/bar/tags/latest HTTP/1.1
Host: registry-1.docker.io
Accept: application/json
Content-Type: application/json
X-Docker-Registry-Version: 0.6.0
Cookie: (Cookie provided by the Registry)
```

Parameters

- **namespace** – namespace for the repo
- **repository** – name for the repo
- **tag** – name of tag you want to get

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json
X-Docker-Registry-Version: 0.6.0

"9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f"
```

Status Codes

- **200** – OK
- **401** – Requires authorization
- **404** – Tag not found

DELETE /v1/repositories/(*namespace*)**/***repository*/**tags/***tag* delete the tag for the repo

Example Request:

```
DELETE /v1/repositories/foo/bar/tags/latest HTTP/1.1
Host: registry-1.docker.io
Accept: application/json
Content-Type: application/json
Cookie: (Cookie provided by the Registry)
```

Parameters

- **namespace** – namespace for the repo
- **repository** – name for the repo
- **tag** – name of tag you want to delete

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json
X-Docker-Registry-Version: 0.6.0

""
```

Status Codes

- **200** – OK
- **401** – Requires authorization
- **404** – Tag not found

PUT /v1/repositories/(*namespace*)**/**
*repository***/tags/***tag* put a tag for the given repo.

Example Request:

```
PUT /v1/repositories/foo/bar/tags/latest HTTP/1.1
Host: registry-1.docker.io
Accept: application/json
Content-Type: application/json
Cookie: (Cookie provided by the Registry)

"9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f"
```

Parameters

- **namespace** – namespace for the repo
- **repository** – name for the repo
- **tag** – name of tag you want to add

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json
X-Docker-Registry-Version: 0.6.0

""
```

Status Codes

- **200** – OK
- **400** – Invalid data
- **401** – Requires authorization
- **404** – Image not found

2.3 Repositories

DELETE /v1/repositories/(*namespace*)**/**
*repository***/** delete a repository

Example Request:

```
DELETE /v1/repositories/foo/bar/ HTTP/1.1
Host: registry-1.docker.io
Accept: application/json
Content-Type: application/json
Cookie: (Cookie provided by the Registry)

""
```

Parameters

- **namespace** – namespace for the repo
- **repository** – name for the repo

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json
X-Docker-Registry-Version: 0.6.0

""
```

Status Codes

- **200** – OK
- **401** – Requires authorization
- **404** – Repository not found

2.4 Status

GET /v1/_ping

Check status of the registry. This endpoint is also used to determine if the registry supports SSL.

Example Request:

```
GET /v1/_ping HTTP/1.1
Host: registry-1.docker.io
Accept: application/json
Content-Type: application/json

""
```

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json
X-Docker-Registry-Version: 0.6.0

""
```

Status Codes

- **200** – OK

3 Authorization

This is where we describe the authorization process, including the tokens and cookies.

TODO: add more info.

5.4.3 Docker Index API

1. Brief introduction

- This is the REST API for the Docker index
- Authorization is done with basic auth over SSL
- Not all commands require authentication, only those noted as such.

2. Endpoints

2.1 Repository

Repositories

User Repo

PUT /v1/repositories/(*namespace*)**/**
*repo_name***/** Create a user repository with the given `namespace` and `repo_name`.

Example Request:

```
PUT /v1/repositories/foo/bar/ HTTP/1.1
Host: index.docker.io
Accept: application/json
Content-Type: application/json
Authorization: Basic akmklmasadalkm==
X-Docker-Token: true

[{"id": "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f"}]
```

Parameters

- **namespace** – the namespace for the repo
- **repo_name** – the name for the repo

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json
WWW-Authenticate: Token signature=123abc,repository="foo/bar",access=write
X-Docker-Token: signature=123abc,repository="foo/bar",access=write
X-Docker-Endpoints: registry-1.docker.io [, registry-2.docker.io]

""
```

Status Codes

- **200** – Created
- **400** – Errors (invalid json, missing or invalid fields, etc)
- **401** – Unauthorized
- **403** – Account is not Active

DELETE /v1/repositories/(*namespace*)**/**
*repo_name***/** Delete a user repository with the given `namespace` and `repo_name`.

Example Request:

```
DELETE /v1/repositories/foo/bar/ HTTP/1.1
Host: index.docker.io
Accept: application/json
Content-Type: application/json
Authorization: Basic akmklmasadalkm==
X-Docker-Token: true

""
```

Parameters

- **namespace** – the namespace for the repo
- **repo_name** – the name for the repo

Example Response:

```
HTTP/1.1 202
Vary: Accept
Content-Type: application/json
WWW-Authenticate: Token signature=123abc,repository="foo/bar",access=delete
X-Docker-Token: signature=123abc,repository="foo/bar",access=delete
X-Docker-Endpoints: registry-1.docker.io [, registry-2.docker.io]

""
```

Status Codes

- **200** – Deleted
- **202** – Accepted
- **400** – Errors (invalid json, missing or invalid fields, etc)
- **401** – Unauthorized
- **403** – Account is not Active

Library Repo

PUT /v1/repositories/(*repo_name*)**/**
Create a library repository with the given `repo_name`. This is a restricted feature only available to docker admins.

When namespace is missing, it is assumed to be `library`

Example Request:

```
PUT /v1/repositories/foobar/ HTTP/1.1
Host: index.docker.io
Accept: application/json
Content-Type: application/json
Authorization: Basic akmklmasadalkm==
X-Docker-Token: true

[{"id": "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f"}]
```

Parameters

- **repo_name** – the library name for the repo

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json
WWW-Authenticate: Token signature=123abc,repository="library/foobar",access=write
X-Docker-Token: signature=123abc,repository="foo/bar",access=write
X-Docker-Endpoints: registry-1.docker.io [, registry-2.docker.io]

""
```

Status Codes

- **200** – Created
- **400** – Errors (invalid json, missing or invalid fields, etc)
- **401** – Unauthorized
- **403** – Account is not Active

DELETE /v1/repositories/(*repo_name*)**/**

Delete a library repository with the given `repo_name`. This is a restricted feature only available to docker admins.

When namespace is missing, it is assumed to be `library`

Example Request:

```
DELETE /v1/repositories/foobar/ HTTP/1.1
Host: index.docker.io
Accept: application/json
Content-Type: application/json
Authorization: Basic akmklmasadalkm==
X-Docker-Token: true

""
```

Parameters

- **repo_name** – the library name for the repo

Example Response:

```
HTTP/1.1 202
Vary: Accept
Content-Type: application/json
```

```
WWW-Authenticate: Token signature=123abc,repository="library/foobar",access=delete
X-Docker-Token: signature=123abc,repository="foo/bar",access=delete
X-Docker-Endpoints: registry-1.docker.io [, registry-2.docker.io]

""
```

Status Codes

- **200** – Deleted
- **202** – Accepted
- **400** – Errors (invalid json, missing or invalid fields, etc)
- **401** – Unauthorized
- **403** – Account is not Active

Repository Images

User Repo Images

PUT /v1/repositories/(*namespace*)**/***repo_name*`/images` Update the images for a user repo.

Example Request:

```
PUT /v1/repositories/foo/bar/images HTTP/1.1
Host: index.docker.io
Accept: application/json
Content-Type: application/json
Authorization: Basic akmklmasadalkm==

[{"id": "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f",
"checksum": "b486531f9a779a0c17e3ed29dae8f12c4f9e89cc6f0bc3c38722009fe6857087"}]
```

Parameters

- **namespace** – the namespace for the repo
- **repo_name** – the name for the repo

Example Response:

```
HTTP/1.1 204
Vary: Accept
Content-Type: application/json

""
```

Status Codes

- **204** – Created
- **400** – Errors (invalid json, missing or invalid fields, etc)
- **401** – Unauthorized
- **403** – Account is not Active or permission denied

GET /v1/repositories/(*namespace*)**/**
*repo_name***/images** get the images for a user repo.

Example Request:

```
GET /v1/repositories/foo/bar/images HTTP/1.1
Host: index.docker.io
Accept: application/json
```

Parameters

- **namespace** – the namespace for the repo
- **repo_name** – the name for the repo

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json

[{"id": "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f",
"checksum": "b486531f9a779a0c17e3ed29dae8f12c4f9e89cc6f0bc3c38722009fe6857087"},
{"id": "ertwetewtwe38722009fe6857087b486531f9a779a0c1dfddgfgsdgdsgds",
"checksum": "34t23f23fc17e3ed29dae8f12c4f9e89cc6f0bsdfgfsdgdsgdsgerwgew"}]
```

Status Codes

- **200** – OK
- **404** – Not found

Library Repo Images

PUT /v1/repositories/(*repo_name*)**/images**
Update the images for a library repo.

Example Request:

```
PUT /v1/repositories/foobar/images HTTP/1.1
Host: index.docker.io
Accept: application/json
Content-Type: application/json
Authorization: Basic akmklmasadalkm==

[{"id": "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f",
"checksum": "b486531f9a779a0c17e3ed29dae8f12c4f9e89cc6f0bc3c38722009fe6857087"}]
```

Parameters

- **repo_name** – the library name for the repo

Example Response:

```
HTTP/1.1 204
Vary: Accept
Content-Type: application/json

""
```

Status Codes

- **204** – Created
- **400** – Errors (invalid json, missing or invalid fields, etc)
- **401** – Unauthorized
- **403** – Account is not Active or permission denied

GET /v1/repositories/(*repo_name*)**/images**
get the images for a library repo.

Example Request:

```
GET /v1/repositories/foobar/images HTTP/1.1
Host: index.docker.io
Accept: application/json
```

Parameters

- **repo_name** – the library name for the repo

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json

[{"id": "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f",
"checksum": "b486531f9a779a0c17e3ed29dae8f12c4f9e89cc6f0bc3c38722009fe6857087"},
{"id": "ertwetewtwe38722009fe6857087b486531f9a779a0c1dfddgfgsdgdsgds",
"checksum": "34t23f23fc17e3ed29dae8f12c4f9e89cc6f0bsdfgfsdgdsgdsgerwgew"}]
```

Status Codes

- **200** – OK
- **404** – Not found

Repository Authorization

Library Repo
PUT /v1/repositories/(*repo_name*)**/auth**
authorize a token for a library repo

Example Request:

```
PUT /v1/repositories/foobar/auth HTTP/1.1
Host: index.docker.io
Accept: application/json
Authorization: Token signature=123abc,repository="library/foobar",access=write
```

Parameters

- **repo_name** – the library name for the repo

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json

"OK"
```

Status Codes

- **200** – OK
- **403** – Permission denied
- **404** – Not found

User Repo
PUT /v1/repositories/(*namespace*)**/***repo_name***/auth** authorize a token for a user repo

Example Request:

```
PUT /v1/repositories/foo/bar/auth HTTP/1.1
Host: index.docker.io
Accept: application/json
Authorization: Token signature=123abc,repository="foo/bar",access=write
```

Parameters

- **namespace** – the namespace for the repo
- **repo_name** – the name for the repo

Example Response:

```
HTTP/1.1 200
Vary: Accept
Content-Type: application/json

"OK"
```

Status Codes

- **200** – OK
- **403** – Permission denied
- **404** – Not found

2.2 Users

User Login
GET /v1/users
If you want to check your login, you can try this endpoint

Example Request:

```
GET /v1/users HTTP/1.1
Host: index.docker.io
Accept: application/json
Authorization: Basic akmklmasadalkm==
```

Example Response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

OK
```

Status Codes

- **200** – no error
- **401** – Unauthorized
- **403** – Account is not Active

User Register
`POST /v1/users`

Registering a new account.

Example request:

```
POST /v1/users HTTP/1.1
Host: index.docker.io
Accept: application/json
Content-Type: application/json

{"email": "sam@dotcloud.com",
 "password": "toto42",
 "username": "foobar"'}
```

Json Parameters

- **email** – valid email address, that needs to be confirmed
- **username** – min 4 character, max 30 characters, must match the regular expression [a-z0-9_].
- **password** – min 5 characters

Example Response:

```
HTTP/1.1 201 OK
Vary: Accept
Content-Type: application/json

"User Created"
```

Status Codes

- **201** – User Created
- **400** – Errors (invalid json, missing or invalid fields, etc)

Update User
`PUT /v1/users/`(*username*)**`/`**

Change a password or email address for given user. If you pass in an email, it will add it to your account, it will not remove the old one. Passwords will be updated.

It is up to the client to verify that that password that is sent is the one that they want. Common approach is to have them type it twice.

Example Request:

```
PUT /v1/users/fakeuser/ HTTP/1.1
Host: index.docker.io
Accept: application/json
Content-Type: application/json
Authorization: Basic akmklmasadalkm==

{"email": "sam@dotcloud.com",
 "password": "toto42"}
```

Parameters

- **username** – username for the person you want to update

Example Response:

```
HTTP/1.1 204
Vary: Accept
Content-Type: application/json

""
```

Status Codes

- **204** – User Updated
- **400** – Errors (invalid json, missing or invalid fields, etc)
- **401** – Unauthorized
- **403** – Account is not Active
- **404** – User not found

2.3 Search

If you need to search the index, this is the endpoint you would use.

Search

`GET /v1/search`

Search the Index given a search term. It accepts GET only.

Example request:

```
GET /v1/search?q=search_term HTTP/1.1
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{"query":"search_term",
```

```
 "num_results": 3,
 "results" : [
    {"name": "ubuntu", "description": "An ubuntu image..."},
    {"name": "centos", "description": "A centos image..."},
    {"name": "fedora", "description": "A fedora image..."}
  ]
}
```

Query Parameters

- **q** – what you want to search for

Status Codes

- **200** – no error
- **500** – server error

5.4.4 Docker Remote API

1. Brief introduction

- The Remote API is replacing rcli
- By default the Docker daemon listens on unix:///var/run/docker.sock and the client must have root access to interact with the daemon
- If a group named *docker* exists on your system, docker will apply ownership of the socket to the group
- The API tends to be REST, but for some complex commands, like attach or pull, the HTTP connection is hijacked to transport stdout stdin and stderr
- Since API version 1.2, the auth configuration is now handled client side, so the client has to send the authConfig as POST in /images/(name)/push

2. Versions

The current version of the API is 1.9

Calling /images/<name>/insert is the same as calling /v1.9/images/<name>/insert

You can still call an old version of the api using /v1.0/images/<name>/insert

v1.9

Full Documentation `docker_remote_api_v1.9`

What's new

`POST /build`

New! This endpoint now takes a serialized ConfigFile which it uses to resolve the proper registry auth credentials for pulling the base image. Clients which previously implemented the version accepting an AuthConfig object must be updated.

v1.8

Full Documentation `docker_remote_api_v1.8`

What's new

`POST /build`

New! This endpoint now returns build status as json stream. In case of a build error, it returns the exit status of the failed command.

`GET /containers/`(*id*)**`/json`**

New! This endpoint now returns the host config for the container.

`POST /images/create`

`POST /images/`(*name*)**`/insert`**

`POST /images/`(*name*)**`/push`**

New! progressDetail object was added in the JSON. It's now possible to get the current value and the total of the progress without having to parse the string.

v1.7

Full Documentation `docker_remote_api_v1.7`

What's new

`GET /images/json`

The format of the json returned from this uri changed. Instead of an entry for each repo/tag on an image, each image is only represented once, with a nested attribute indicating the repo/tags that apply to that image.

Instead of:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "VirtualSize": 131506275,
    "Size": 131506275,
    "Created": 1365714795,
    "Id": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
    "Tag": "12.04",
    "Repository": "ubuntu"
  },
  {
    "VirtualSize": 131506275,
    "Size": 131506275,
    "Created": 1365714795,
    "Id": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
    "Tag": "latest",
    "Repository": "ubuntu"
  },
  {
    "VirtualSize": 131506275,
    "Size": 131506275,
    "Created": 1365714795,
    "Id": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
    "Tag": "precise",
```

```
    "Repository": "ubuntu"
  },
  {
    "VirtualSize": 180116135,
    "Size": 24653,
    "Created": 1364102658,
    "Id": "b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc",
    "Tag": "12.10",
    "Repository": "ubuntu"
  },
  {
    "VirtualSize": 180116135,
    "Size": 24653,
    "Created": 1364102658,
    "Id": "b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc",
    "Tag": "quantal",
    "Repository": "ubuntu"
  }
]
```

The returned json looks like this:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "RepoTags": [
      "ubuntu:12.04",
      "ubuntu:precise",
      "ubuntu:latest"
    ],
    "Id": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
    "Created": 1365714795,
    "Size": 131506275,
    "VirtualSize": 131506275
  },
  {
    "RepoTags": [
      "ubuntu:12.10",
      "ubuntu:quantal"
    ],
    "ParentId": "27cf784147099545",
    "Id": "b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc",
    "Created": 1364102658,
    "Size": 24653,
    "VirtualSize": 180116135
  }
]
```

GET /images/viz

This URI no longer exists. The `images -viz` output is now generated in the client, using the `/images/json` data.

v1.6

Full Documentation `docker_remote_api_v1.6`

What's new

POST /containers/(*id*)**/attach**

New! You can now split stderr from stdout. This is done by prefixing a header to each transmition. See `POST /containers/(id)/attach`. The WebSocket attach is unchanged. Note that attach calls on the previous API version didn't change. Stdout and stderr are merged.

v1.5

Full Documentation `docker_remote_api_v1.5`

What's new

POST /images/create

New! You can now pass registry credentials (via an AuthConfig object) through the *X-Registry-Auth* header

POST /images/(*name*)**/push**

New! The AuthConfig object now needs to be passed through the *X-Registry-Auth* header

GET /containers/json

New! The format of the *Ports* entry has been changed to a list of dicts each containing *PublicPort*, *PrivatePort* and *Type* describing a port mapping.

v1.4

Full Documentation `docker_remote_api_v1.4`

What's new

POST /images/create

New! When pulling a repo, all images are now downloaded in parallel.

GET /containers/(*id*)**/top**

New! You can now use ps args with docker top, like *docker top <container_id> aux*

GET /events:

New! Image's name added in the events

v1.3

docker v0.5.0 51f6c4a

Full Documentation `docker_remote_api_v1.3`

What's new

GET /containers/(*id*)**/top**

List the processes running inside a container.

GET /events:

New! Monitor docker's events via streaming or via polling

Builder (/build):

- Simplify the upload of the build context
- Simply stream a tarball instead of multipart upload with 4 intermediary buffers
- Simpler, less memory usage, less disk usage and faster

: The /build improvements are not reverse-compatible. Pre 1.3 clients will break on /build.

List containers (/containers/json):

- You can use size=1 to get the size of the containers

Start containers (/containers/<id>/start):

- You can now pass host-specific configuration (e.g. bind mounts) in the POST body for start calls

v1.2

docker v0.4.2 2e7649b

Full Documentation `docker_remote_api_v1.2`

What's new The auth configuration is now handled by the client.

The client should send it's authConfig as POST on each call of /images/(name)/push

`GET /auth`
Deprecated.

`POST /auth`
Only checks the configuration but doesn't store it on the server

Deleting an image is now improved, will only untag the image if it has children and remove all the untagged parents if has any.

`POST /images/<name>/delete`
Now returns a JSON structure with the list of images deleted/untagged.

v1.1

docker v0.4.0 a8ae398

Full Documentation `docker_remote_api_v1.1`

What's new
`POST /images/create`
`POST /images/`(*name*)**`/insert`**

`POST /images/`(*name*)**`/push`**
Uses json stream instead of HTML hijack, it looks like this:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"status":"Pushing..."}
{"status":"Pushing", "progress":"1/? (n/a)"}
{"error":"Invalid..."}
...
```

v1.0

docker v0.3.4 8d73740

Full Documentation `docker_remote_api_v1.0`

What's new Initial version

5.4.5 Docker Remote API Client Libraries

These libraries have not been tested by the Docker Maintainers for compatibility. Please file issues with the library owners. If you find more library implementations, please list them in Docker doc bugs and we will add the libraries here.

Language/Framework	Name	Repository	Status
Python	docker-py	https://github.com/dotcloud/docker-py	Active
Ruby	docker-client	https://github.com/geku/docker-client	Outdated
Ruby	docker-api	https://github.com/swipely/docker-api	Active
Javascript (NodeJS)	docker.io	https://github.com/appersonlabs/docker.io Install via NPM: *npm install docker.io*	Active
Javascript	docker-js	https://github.com/dgoujard/docker-js	Active
Javascript (Angular) **WebUI**	dockerui	https://github.com/crosbymichael/dockerui	Active
Java	docker-java	https://github.com/kpelykh/docker-java	Active
Erlang	erldocker	https://github.com/proger/erldocker	Active
Go	go-dockerclient	https://github.com/fsouza/go-dockerclient	Active
PHP	Alvine	http://pear.alvine.io/ (alpha)	Active

6.1 Contributing to Docker

Want to hack on Docker? Awesome!

The repository includes all the instructions you need to get started.

The developer environment Dockerfile specifies the tools and versions used to test and build Docker.

If you're making changes to the documentation, see the README.md.

The documentation environment Dockerfile specifies the tools and versions used to build the Documentation.

Further interesting details can be found in the Packaging hints.

6.2 Setting Up a Dev Environment

To make it easier to contribute to Docker, we provide a standard development environment. It is important that the same environment be used for all tests, builds and releases. The standard development environment defines all build dependencies: system libraries and binaries, go environment, go dependencies, etc.

6.2.1 Step 1: Install Docker

Docker's build environment itself is a Docker container, so the first step is to install Docker on your system.

You can follow the install instructions most relevant to your system. Make sure you have a working, up-to-date docker installation, then continue to the next step.

6.2.2 Step 2: Check out the Source

```
git clone http://git@github.com/dotcloud/docker
cd docker
```

To checkout a different revision just use `git checkout` with the name of branch or revision number.

6.2.3 Step 3: Build the Environment

This following command will build a development environment using the Dockerfile in the current directory. Essentially, it will install all the build and runtime dependencies necessary to build and test Docker. This command will take some time to complete when you first execute it.

```
sudo make build
```

If the build is successful, congratulations! You have produced a clean build of docker, neatly encapsulated in a standard build environment.

6.2.4 Step 4: Build the Docker Binary

To create the Docker binary, run this command:

```
sudo make binary
```

This will create the Docker binary in `./bundles/<version>-dev/binary/`

Using your built Docker binary

The binary is available outside the container in the directory `./bundles/<version>-dev/binary/`. You can swap your host docker executable with this binary for live testing - for example, on ubuntu:

```
sudo service docker stop ; sudo cp $(which docker) $(which docker)_ ; sudo cp ./bundles/<version>-dev
```

: Its safer to run the tests below before swapping your hosts docker binary.

6.2.5 Step 5: Run the Tests

To execute the test cases, run this command:

```
sudo make test
```

Note: if you're running the tests in vagrant, you need to specify a dns entry in the command (either edit the Makefile, or run the step manually):

```
sudo docker run -dns 8.8.8.8 -privileged -v `pwd`:/go/src/github.com/dotcloud/docker docker hack/make
```

If the test are successful then the tail of the output should look something like this

```
--- PASS: TestWriteBroadcaster (0.00 seconds)
=== RUN TestRaceWriteBroadcaster
--- PASS: TestRaceWriteBroadcaster (0.00 seconds)
=== RUN TestTruncIndex
--- PASS: TestTruncIndex (0.00 seconds)
=== RUN TestCompareKernelVersion
--- PASS: TestCompareKernelVersion (0.00 seconds)
=== RUN TestHumanSize
--- PASS: TestHumanSize (0.00 seconds)
=== RUN TestParseHost
--- PASS: TestParseHost (0.00 seconds)
=== RUN TestParseRepositoryTag
--- PASS: TestParseRepositoryTag (0.00 seconds)
=== RUN TestGetResolvConf
```

```
--- PASS: TestGetResolvConf (0.00 seconds)
=== RUN TestCheckLocalDns
--- PASS: TestCheckLocalDns (0.00 seconds)
=== RUN TestParseRelease
--- PASS: TestParseRelease (0.00 seconds)
=== RUN TestDependencyGraphCircular
--- PASS: TestDependencyGraphCircular (0.00 seconds)
=== RUN TestDependencyGraph
--- PASS: TestDependencyGraph (0.00 seconds)
PASS
ok      github.com/dotcloud/docker/utils        0.017s
```

6.2.6 Step 6: Use Docker

You can run an interactive session in the newly built container:

```
sudo make shell

# type 'exit' or Ctrl-D to exit
```

6.2.7 Extra Step: Build and view the Documentation

If you want to read the documentation from a local website, or are making changes to it, you can build the documentation and then serve it by:

```
    sudo make docs
# when its done, you can point your browser to http://yourdockerhost:8000
    # type Ctrl-C to exit
```

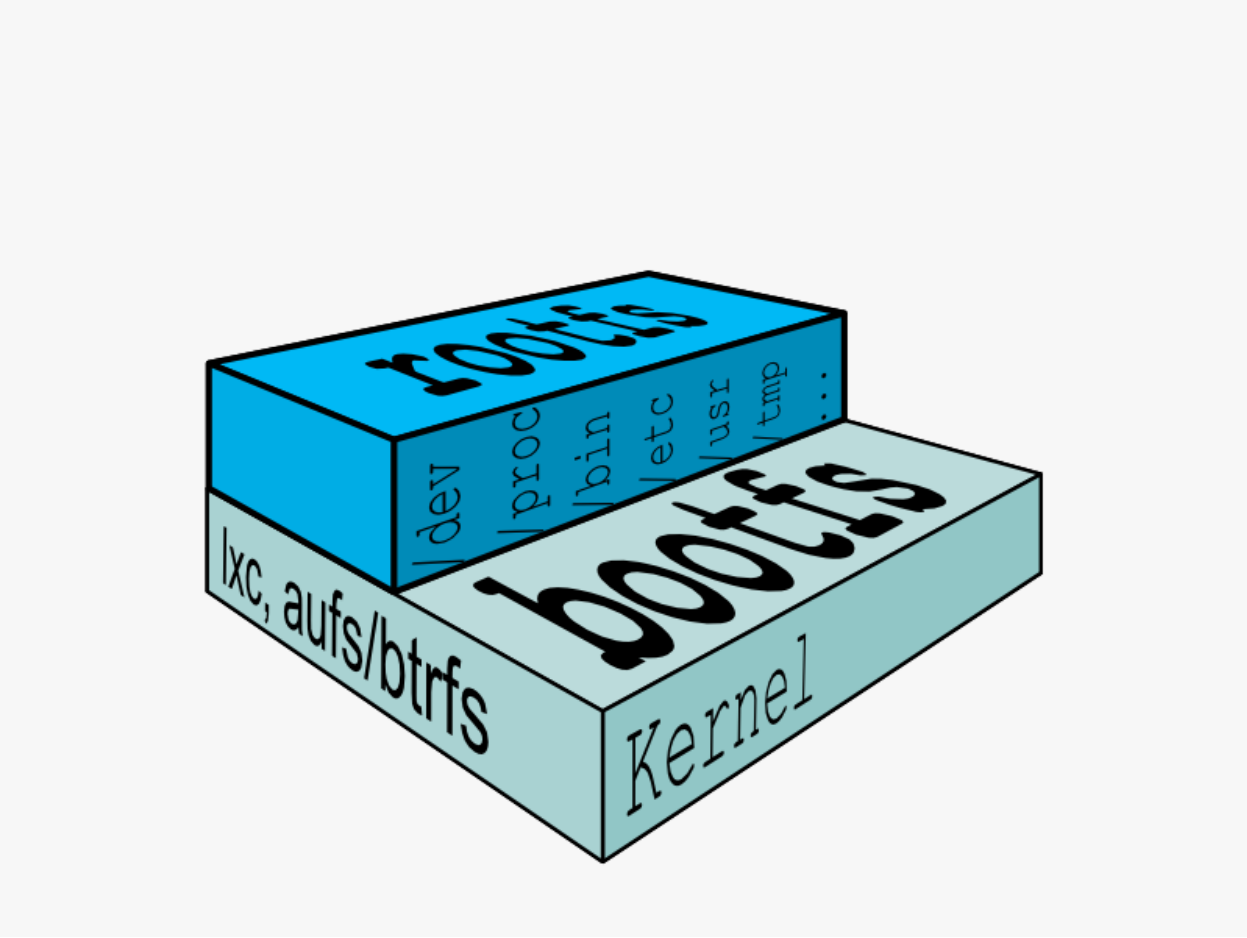
Need More Help?

If you need more help then hop on to the #docker-dev IRC channel or post a message on the Docker developer mailing list.

Definitions of terms used in Docker documentation.

Contents:

7.1 File System



In order for a Linux system to run, it typically needs two file systems:

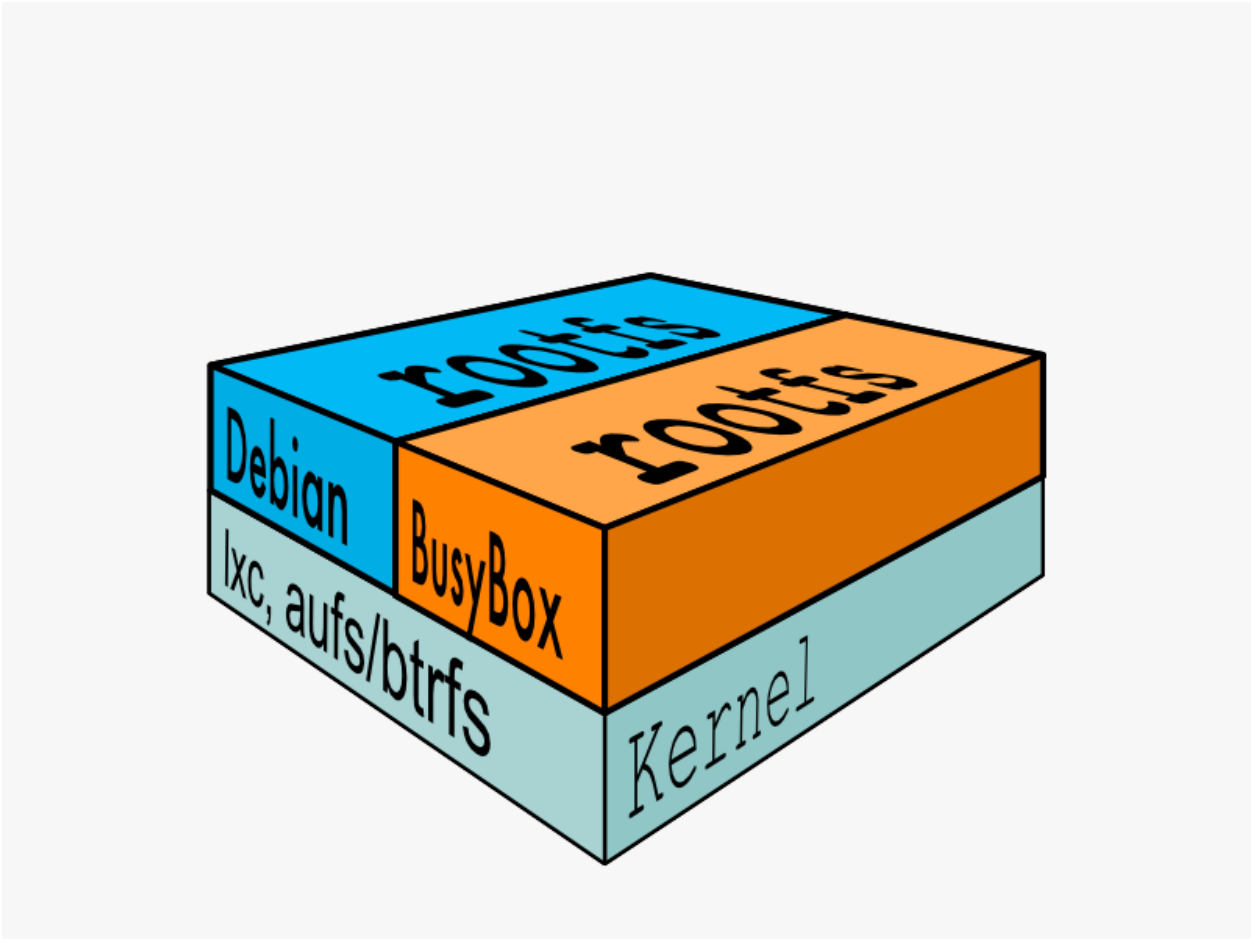
1. boot file system (bootfs)

2. root file system (rootfs)

The **boot file system** contains the bootloader and the kernel. The user never makes any changes to the boot file system. In fact, soon after the boot process is complete, the entire kernel is in memory, and the boot file system is unmounted to free up the RAM associated with the initrd disk image.

The **root file system** includes the typical directory structure we associate with Unix-like operating systems: `/dev`, `/proc`, `/bin`, `/etc`, `/lib`, `/usr`, and `/tmp` plus all the configuration files, binaries and libraries required to run user applications (like bash, ls, and so forth).

While there can be important kernel differences between different Linux distributions, the contents and organization of the root file system are usually what make your software packages dependent on one distribution versus another. Docker can help solve this problem by running multiple distributions at the same time.



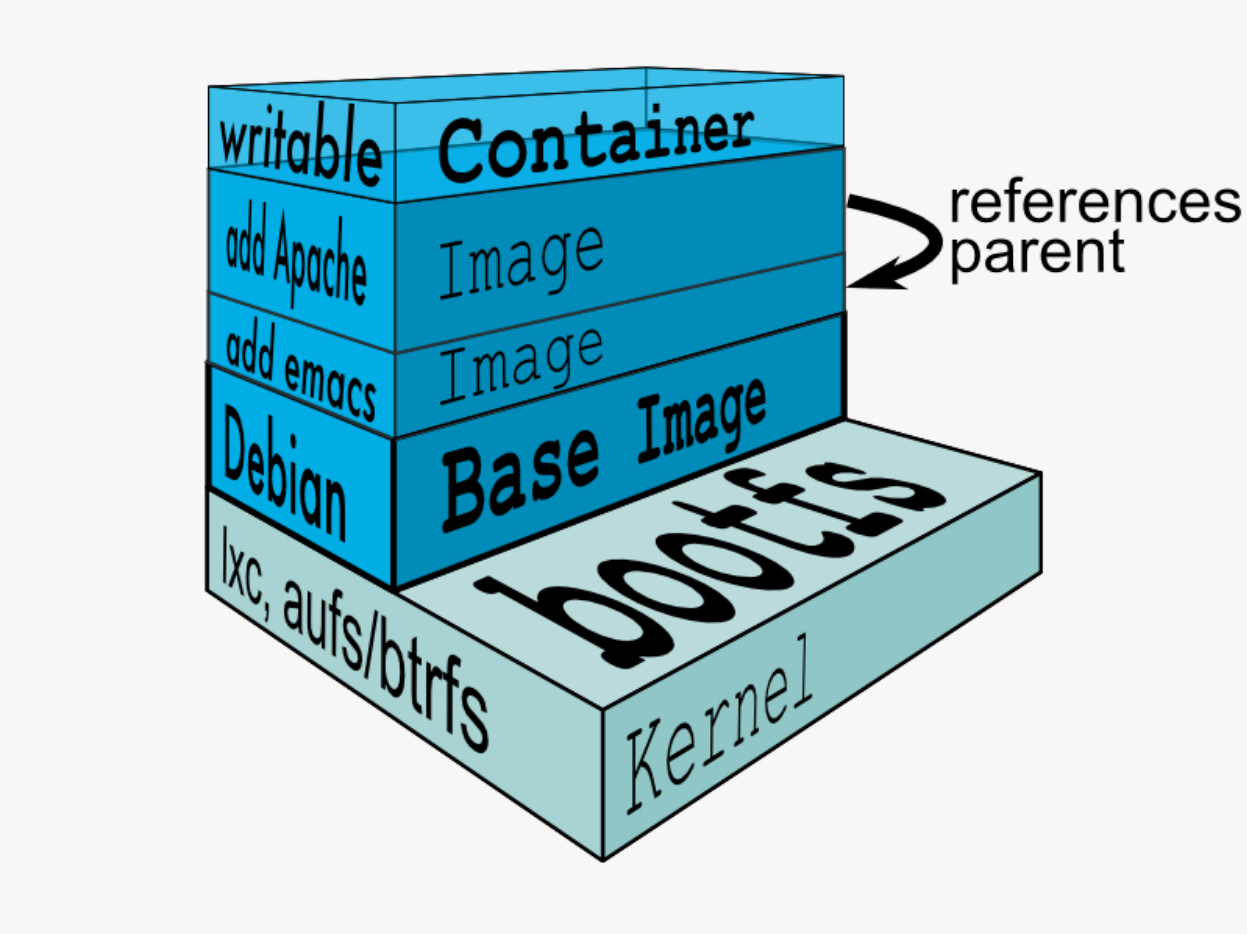
7.2 Layers

In a traditional Linux boot, the kernel first mounts the root *File System* as read-only, checks its integrity, and then switches the whole rootfs volume to read-write mode.

7.2.1 Layer

When Docker mounts the rootfs, it starts read-only, as in a traditional Linux boot, but then, instead of changing the file system to read-write mode, it takes advantage of a union mount to add a read-write file system *over* the read-only

file system. In fact there may be multiple read-only file systems stacked on top of each other. We think of each one of these file systems as a **layer**.

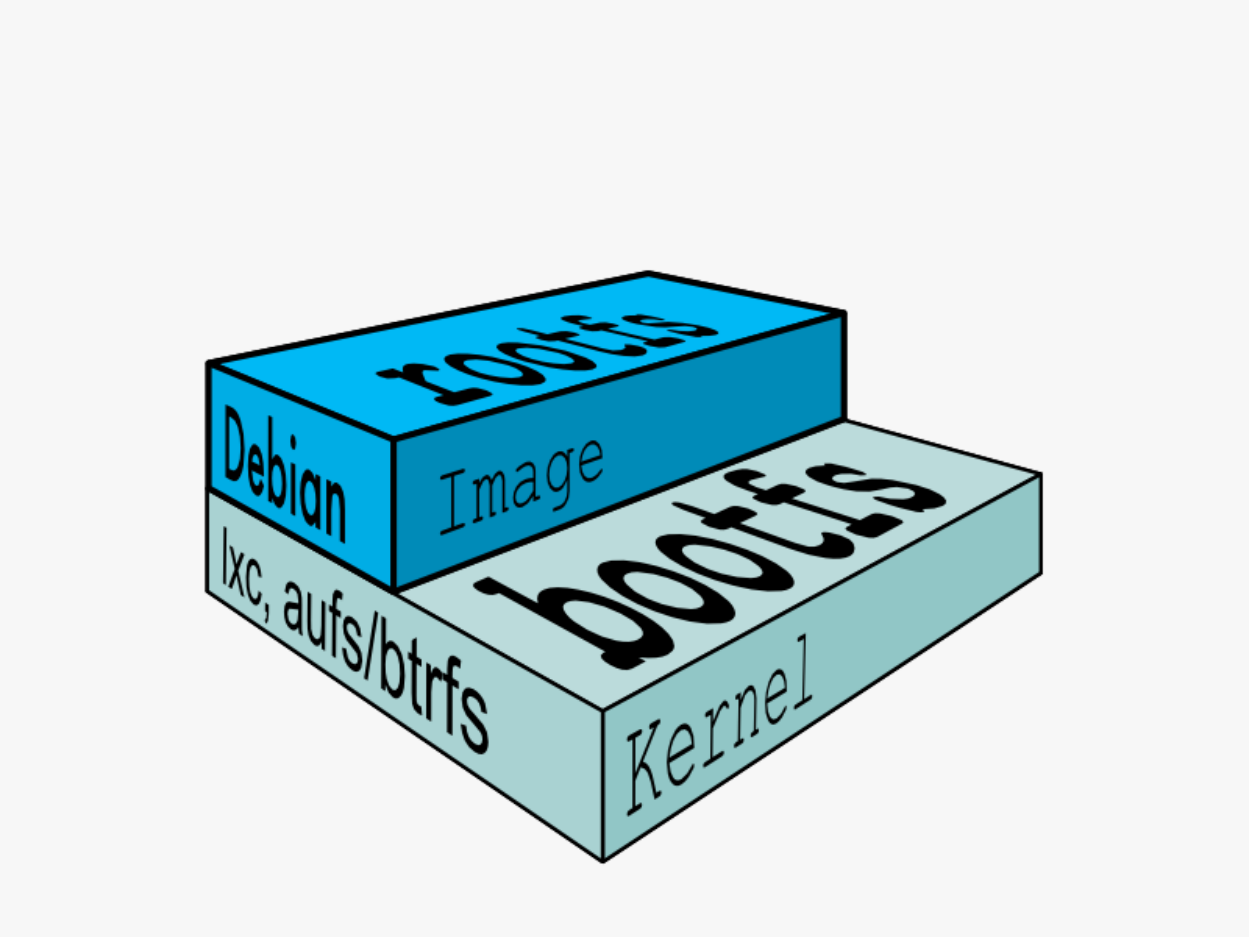


At first, the top read-write layer has nothing in it, but any time a process creates a file, this happens in the top layer. And if something needs to update an existing file in a lower layer, then the file gets copied to the upper layer and changes go into the copy. The version of the file on the lower layer cannot be seen by the applications anymore, but it is there, unchanged.

7.2.2 Union File System

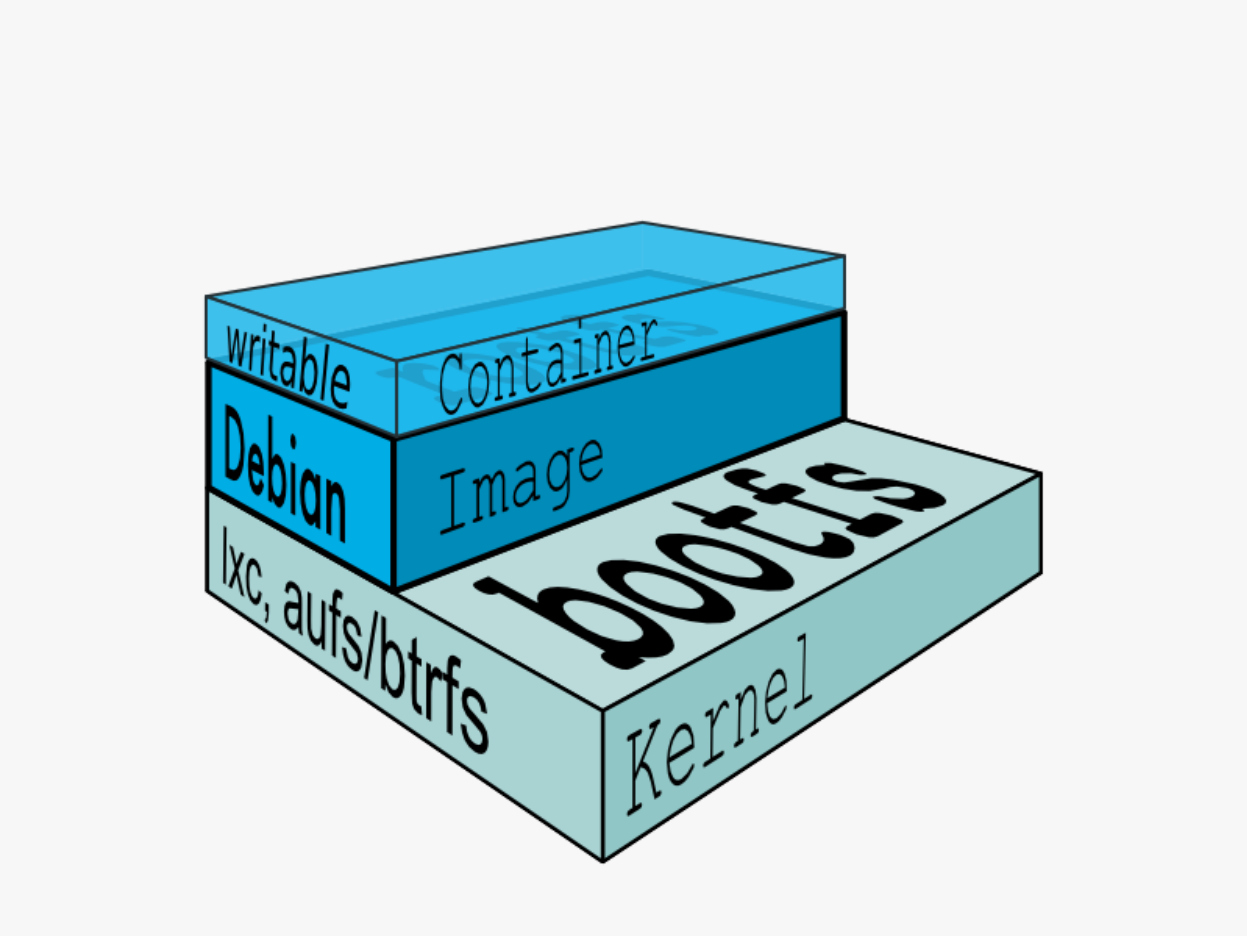
We call the union of the read-write layer and all the read-only layers a **union file system**.

7.3 Image

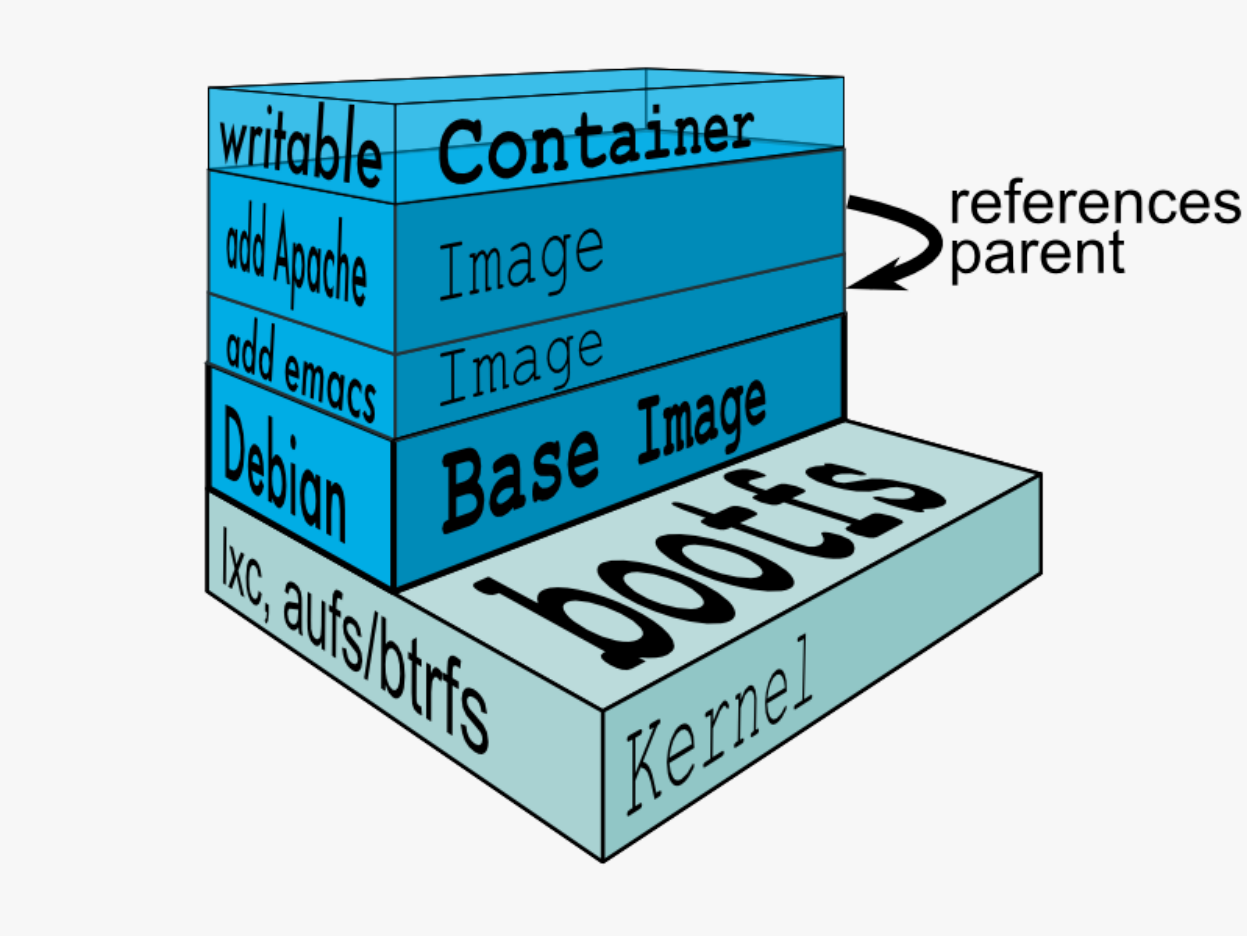


In Docker terminology, a read-only *Layer* is called an **image**. An image never changes.

Since Docker uses a *Union File System*, the processes think the whole file system is mounted read-write. But all the changes go to the top-most writeable layer, and underneath, the original file in the read-only image is unchanged. Since images don't change, images do not have state.


writable
Container
Debian
Image
bootfs
lxc, aufs/btrfs
Kernel

7.3.1 Parent Image



Each image may depend on one more image which forms the layer beneath it. We sometimes say that the lower image is the **parent** of the upper image.

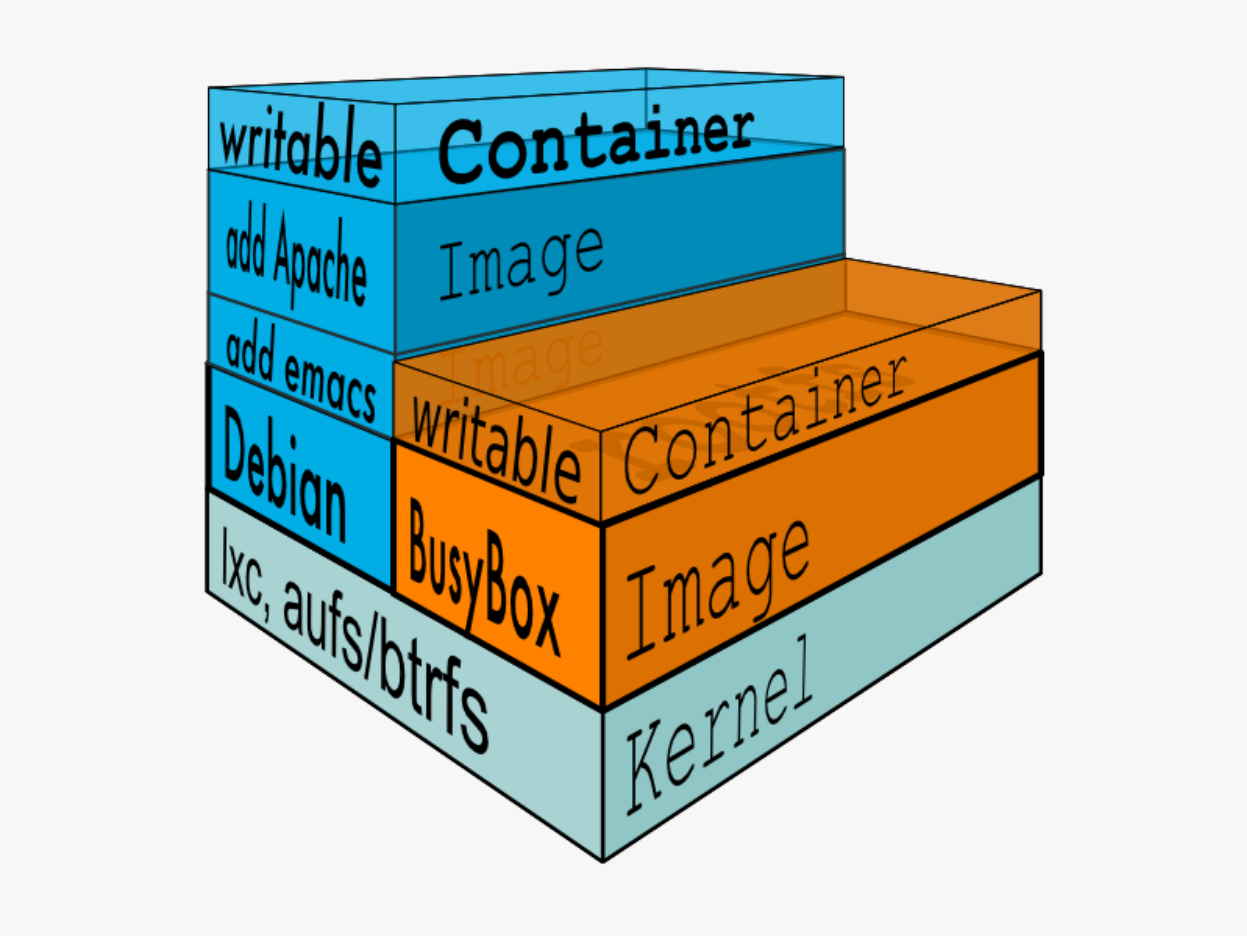
7.3.2 Base Image

An image that has no parent is a **base image**.

7.3.3 Image IDs

All images are identified by a 64 hexadecimal digit string (internally a 256bit value). To simplify their use, a short ID of the first 12 characters can be used on the command line. There is a small possibility of short id collisions, so the docker server will always return the long ID.

7.4 Container



Once you start a process in Docker from an *Image*, Docker fetches the image and its *Parent Image*, and repeats the process until it reaches the *Base Image*. Then the *Union File System* adds a read-write layer on top. That read-write layer, plus the information about its *Parent Image* and some additional information like its unique id, networking configuration, and resource limits is called a **container**.

7.4.1 Container State

Containers can change, and so they have state. A container may be **running** or **exited**.

When a container is running, the idea of a "container" also includes a tree of processes running on the CPU, isolated from the other processes running on the host.

When the container is exited, the state of the file system and its exit value is preserved. You can start, stop, and restart a container. The processes restart from scratch (their memory state is **not** preserved in a container), but the file system is just as it was when the container was stopped.

You can promote a container to an *Image* with `docker commit`. Once a container is an image, you can use it as a parent for new containers.

7.4.2 Container IDs

All containers are identified by a 64 hexadecimal digit string (internally a 256bit value). To simplify their use, a short ID of the first 12 characters can be used on the commandline. There is a small possibility of short id collisions, so the docker server will always return the long ID.

8.1 Docker

Adapted from Containers & Docker: How Secure are They?

There are three major areas to consider when reviewing Docker security:

- the intrinsic security of containers, as implemented by kernel namespaces and cgroups;
- the attack surface of the Docker daemon itself;
- the "hardening" security features of the kernel and how they interact with containers.

8.1.1 Kernel Namespaces

Docker containers are essentially LXC containers, and they come with the same security features. When you start a container with `docker run`, behind the scenes Docker uses `lxc-start` to execute the Docker container. This creates a set of namespaces and control groups for the container. Those namespaces and control groups are not created by Docker itself, but by `lxc-start`. This means that as the LXC userland tools evolve (and provide additional namespaces and isolation features), Docker will automatically make use of them.

Namespaces provide the first and most straightforward form of isolation: processes running within a container cannot see, and even less affect, processes running in another container, or in the host system.

Each container also gets its own network stack, meaning that a container doesn't get a privileged access to the sockets or interfaces of another container. Of course, if the host system is setup accordingly, containers can interact with each other through their respective network interfaces — just like they can interact with external hosts. When you specify public ports for your containers or use *links* then IP traffic is allowed between containers. They can ping each other, send/receive UDP packets, and establish TCP connections, but that can be restricted if necessary. From a network architecture point of view, all containers on a given Docker host are sitting on bridge interfaces. This means that they are just like physical machines connected through a common Ethernet switch; no more, no less.

How mature is the code providing kernel namespaces and private networking? Kernel namespaces were introduced between kernel version 2.6.15 and 2.6.26. This means that since July 2008 (date of the 2.6.26 release, now 5 years ago), namespace code has been exercised and scrutinized on a large number of production systems. And there is more: the design and inspiration for the namespaces code are even older. Namespaces are actually an effort to reimplement the features of OpenVZ in such a way that they could be merged within the mainstream kernel. And OpenVZ was initially released in 2005, so both the design and the implementation are pretty mature.

8.1.2 Control Groups

Control Groups are the other key component of Linux Containers. They implement resource accounting and limiting. They provide a lot of very useful metrics, but they also help to ensure that each container gets its fair share of memory, CPU, disk I/O; and, more importantly, that a single container cannot bring the system down by exhausting one of those resources.

So while they do not play a role in preventing one container from accessing or affecting the data and processes of another container, they are essential to fend off some denial-of-service attacks. They are particularly important on multi-tenant platforms, like public and private PaaS, to guarantee a consistent uptime (and performance) even when some applications start to misbehave.

Control Groups have been around for a while as well: the code was started in 2006, and initially merged in kernel 2.6.24.

8.1.3 Docker Daemon Attack Surface

Running containers (and applications) with Docker implies running the Docker daemon. This daemon currently requires root privileges, and you should therefore be aware of some important details.

First of all, **only trusted users should be allowed to control your Docker daemon**. This is a direct consequence of some powerful Docker features. Specifically, Docker allows you to share a directory between the Docker host and a guest container; and it allows you to do so without limiting the access rights of the container. This means that you can start a container where the `/host` directory will be the `/` directory on your host; and the container will be able to alter your host filesystem without any restriction. This sounds crazy? Well, you have to know that **all virtualization systems allowing filesystem resource sharing behave the same way**. Nothing prevents you from sharing your root filesystem (or even your root block device) with a virtual machine.

This has a strong security implication: if you instrument Docker from e.g. a web server to provision containers through an API, you should be even more careful than usual with parameter checking, to make sure that a malicious user cannot pass crafted parameters causing Docker to create arbitrary containers.

For this reason, the REST API endpoint (used by the Docker CLI to communicate with the Docker daemon) changed in Docker 0.5.2, and now uses a UNIX socket instead of a TCP socket bound on 127.0.0.1 (the latter being prone to cross-site-scripting attacks if you happen to run Docker directly on your local machine, outside of a VM). You can then use traditional UNIX permission checks to limit access to the control socket.

You can also expose the REST API over HTTP if you explicitly decide so. However, if you do that, being aware of the abovementioned security implication, you should ensure that it will be reachable only from a trusted network or VPN; or protected with e.g. `stunnel` and client SSL certificates.

Recent improvements in Linux namespaces will soon allow to run full-featured containers without root privileges, thanks to the new user namespace. This is covered in detail here. Moreover, this will solve the problem caused by sharing filesystems between host and guest, since the user namespace allows users within containers (including the root user) to be mapped to other users in the host system.

The end goal for Docker is therefore to implement two additional security improvements:

- map the root user of a container to a non-root user of the Docker host, to mitigate the effects of a container-to-host privilege escalation;
- allow the Docker daemon to run without root privileges, and delegate operations requiring those privileges to well-audited sub-processes, each with its own (very limited) scope: virtual network setup, filesystem management, etc.

Finally, if you run Docker on a server, it is recommended to run exclusively Docker in the server, and move all other services within containers controlled by Docker. Of course, it is fine to keep your favorite admin tools (probably at least an SSH server), as well as existing monitoring/supervision processes (e.g. NRPE, collectd, etc).

8.1.4 Linux Kernel Capabilities

By default, Docker starts containers with a very restricted set of capabilities. What does that mean?

Capabilities turn the binary "root/non-root" dichotomy into a fine-grained access control system. Processes (like web servers) that just need to bind on a port below 1024 do not have to run as root: they can just be granted the `net_bind_service` capability instead. And there are many other capabilities, for almost all the specific areas where root privileges are usually needed.

This means a lot for container security; let's see why!

Your average server (bare metal or virtual machine) needs to run a bunch of processes as root. Those typically include SSH, cron, syslogd; hardware management tools (to e.g. load modules), network configuration tools (to handle e.g. DHCP, WPA, or VPNs), and much more. A container is very different, because almost all of those tasks are handled by the infrastructure around the container:

- SSH access will typically be managed by a single server running in the Docker host;
- `cron`, when necessary, should run as a user process, dedicated and tailored for the app that needs its scheduling service, rather than as a platform-wide facility;
- log management will also typically be handed to Docker, or by third-party services like Loggly or Splunk;
- hardware management is irrelevant, meaning that you never need to run `udevd` or equivalent daemons within containers;
- network management happens outside of the containers, enforcing separation of concerns as much as possible, meaning that a container should never need to perform `ifconfig`, `route`, or ip commands (except when a container is specifically engineered to behave like a router or firewall, of course).

This means that in most cases, containers will not need "real" root privileges *at all*. And therefore, containers can run with a reduced capability set; meaning that "root" within a container has much less privileges than the real "root". For instance, it is possible to:

- deny all "mount" operations;
- deny access to raw sockets (to prevent packet spoofing);
- deny access to some filesystem operations, like creating new device nodes, changing the owner of files, or altering attributes (including the immutable flag);
- deny module loading;
- and many others.

This means that even if an intruder manages to escalate to root within a container, it will be much harder to do serious damage, or to escalate to the host.

This won't affect regular web apps; but malicious users will find that the arsenal at their disposal has shrunk considerably! You can see the list of dropped capabilities in the Docker code, and a full list of available capabilities in Linux manpages.

Of course, you can always enable extra capabilities if you really need them (for instance, if you want to use a FUSE-based filesystem), but by default, Docker containers will be locked down to ensure maximum safety.

8.1.5 Other Kernel Security Features

Capabilities are just one of the many security features provided by modern Linux kernels. It is also possible to leverage existing, well-known systems like TOMOYO, AppArmor, SELinux, GRSEC, etc. with Docker.

While Docker currently only enables capabilities, it doesn't interfere with the other systems. This means that there are many different ways to harden a Docker host. Here are a few examples.

- You can run a kernel with GRSEC and PAX. This will add many safety checks, both at compile-time and run-time; it will also defeat many exploits, thanks to techniques like address randomization. It doesn't require Docker-specific configuration, since those security features apply system-wide, independently of containers.
- If your distribution comes with security model templates for LXC containers, you can use them out of the box. For instance, Ubuntu comes with AppArmor templates for LXC, and those templates provide an extra safety net (even though it overlaps greatly with capabilities).
- You can define your own policies using your favorite access control mechanism. Since Docker containers are standard LXC containers, there is nothing "magic" or specific to Docker.

Just like there are many third-party tools to augment Docker containers with e.g. special network topologies or shared filesystems, you can expect to see tools to harden existing Docker containers without affecting Docker's core.

8.1.6 Conclusions

Docker containers are, by default, quite secure; especially if you take care of running your processes inside the containers as non-privileged users (i.e. non root).

You can add an extra layer of safety by enabling Apparmor, SELinux, GRSEC, or your favorite hardening solution.

Last but not least, if you see interesting security features in other containerization systems, you will be able to implement them as well with Docker, since everything is provided by the kernel anyway.

For more context and especially for comparisons with VMs and other container systems, please also see the original blog post.

8.2

So you want to create your own *Base Image*? Great!

The specific process will depend heavily on the Linux distribution you want to package. We have some examples below, and you are encouraged to submit pull requests to contribute new ones.

8.2.1 Getting Started

In general, you'll want to start with a working machine that is running the distribution you'd like to package as a base image, though that is not required for some tools like Debian's Debootstrap, which you can also use to build Ubuntu images.

It can be as simple as this to create an Ubuntu base image:

```
$ sudo debootstrap raring raring > /dev/null
$ sudo tar -C raring -c . | sudo docker import - raring
a29c15f1bf7a
$ sudo docker run raring cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=13.04
DISTRIB_CODENAME=raring
DISTRIB_DESCRIPTION="Ubuntu 13.04"
```

There are more example scripts for creating base images in the Docker GitHub Repo:

- BusyBox
- CentOS / Scientific Linux CERN (SLC) on Debian/Ubuntu or on CentOS/RHEL/SLC/etc.

- Debian / Ubuntu

8.3

Linux Containers rely on control groups which not only track groups of processes, but also expose metrics about CPU, memory, and block I/O usage. You can access those metrics and obtain network usage metrics as well. This is relevant for "pure" LXC containers, as well as for Docker containers.

8.3.1 Control Groups

Control groups are exposed through a pseudo-filesystem. In recent distros, you should find this filesystem under `/sys/fs/cgroup`. Under that directory, you will see multiple sub-directories, called devices, freezer, blkio, etc.; each sub-directory actually corresponds to a different cgroup hierarchy.

On older systems, the control groups might be mounted on `/cgroup`, without distinct hierarchies. In that case, instead of seeing the sub-directories, you will see a bunch of files in that directory, and possibly some directories corresponding to existing containers.

To figure out where your control groups are mounted, you can run:

```
grep cgroup /proc/mounts
```

8.3.2 Enumerating Cgroups

You can look into `/proc/cgroups` to see the different control group subsystems known to the system, the hierarchy they belong to, and how many groups they contain.

You can also look at `/proc/<pid>/cgroup` to see which control groups a process belongs to. The control group will be shown as a path relative to the root of the hierarchy mountpoint; e.g. `/` means "this process has not been assigned into a particular group", while `/lxc/pumpkin` means that the process is likely to be a member of a container named `pumpkin`.

8.3.3 Finding the Cgroup for a Given Container

For each container, one cgroup will be created in each hierarchy. On older systems with older versions of the LXC userland tools, the name of the cgroup will be the name of the container. With more recent versions of the LXC tools, the cgroup will be `lxc/<container_name>.`

For Docker containers using cgroups, the container name will be the full ID or long ID of the container. If a container shows up as ae836c95b4c3 in `docker ps`, its long ID might be something like `ae836c95b4c3c9e9179e0e91015512da89fdec91612f63cebae57df9a5444c79`. You can look it up with `docker inspect` or `docker ps -notrunc`.

Putting everything together to look at the memory metrics for a Docker container, take a look at `/sys/fs/cgroup/memory/lxc/<longid>/`.

8.3.4 Metrics from Cgroups: Memory, CPU, Block IO

For each subsystem (memory, CPU, and block I/O), you will find one or more pseudo-files containing statistics.

Memory Metrics: `memory.stat`

Memory metrics are found in the "memory" cgroup. Note that the memory control group adds a little overhead, because it does very fine-grained accounting of the memory usage on your host. Therefore, many distros chose to not enable it by default. Generally, to enable it, all you have to do is to add some kernel command-line parameters: `cgroup_enable=memory swapaccount=1`.

The metrics are in the pseudo-file `memory.stat`. Here is what it will look like:

```
cache 11492564992
rss 1930993664
mapped_file 306728960
pgpgin 406632648
pgpgout 403355412
swap 0
pgfault 728281223
pgmajfault 1724
inactive_anon 46608384
active_anon 1884520448
inactive_file 7003344896
active_file 4489052160
unevictable 32768
hierarchical_memory_limit 9223372036854775807
hierarchical_memsw_limit 9223372036854775807
total_cache 11492564992
total_rss 1930993664
total_mapped_file 306728960
total_pgpgin 406632648
total_pgpgout 403355412
total_swap 0
total_pgfault 728281223
total_pgmajfault 1724
total_inactive_anon 46608384
total_active_anon 1884520448
total_inactive_file 7003344896
total_active_file 4489052160
total_unevictable 32768
```

The first half (without the `total_` prefix) contains statistics relevant to the processes within the cgroup, excluding sub-cgroups. The second half (with the `total_` prefix) includes sub-cgroups as well.

Some metrics are "gauges", i.e. values that can increase or decrease (e.g. swap, the amount of swap space used by the members of the cgroup). Some others are "counters", i.e. values that can only go up, because they represent occurrences of a specific event (e.g. pgfault, which indicates the number of page faults which happened since the creation of the cgroup; this number can never decrease).

cache the amount of memory used by the processes of this control group that can be associated precisely with a block on a block device. When you read from and write to files on disk, this amount will increase. This will be the case if you use "conventional" I/O (`open`, `read`, `write` syscalls) as well as mapped files (with `mmap`). It also accounts for the memory used by `tmpfs` mounts, though the reasons are unclear.

rss the amount of memory that *doesn't* correspond to anything on disk: stacks, heaps, and anonymous memory maps.

mapped_file indicates the amount of memory mapped by the processes in the control group. It doesn't give you information about *how much* memory is used; it rather tells you *how* it is used.

pgfault and pgmajfault indicate the number of times that a process of the cgroup triggered a "page fault" and a "major fault", respectively. A page fault happens when a process accesses a part of its virtual memory space which is nonexistent or protected. The former can happen if the process is buggy and tries to access an invalid address (it will then be sent a `SIGSEGV` signal, typically killing it with the famous `Segmentation fault`

message). The latter can happen when the process reads from a memory zone which has been swapped out, or which corresponds to a mapped file: in that case, the kernel will load the page from disk, and let the CPU complete the memory access. It can also happen when the process writes to a copy-on-write memory zone: likewise, the kernel will preempt the process, duplicate the memory page, and resume the write operation on the process' own copy of the page. "Major" faults happen when the kernel actually has to read the data from disk. When it just has to duplicate an existing page, or allocate an empty page, it's a regular (or "minor") fault.

swap the amount of swap currently used by the processes in this cgroup.

active_anon and inactive_anon the amount of *anonymous* memory that has been identified has respectively *active* and *inactive* by the kernel. "Anonymous" memory is the memory that is *not* linked to disk pages. In other words, that's the equivalent of the rss counter described above. In fact, the very definition of the rss counter is **active_anon + inactive_anon - tmpfs** (where tmpfs is the amount of memory used up by `tmpfs` filesystems mounted by this control group). Now, what's the difference between "active" and "inactive"? Pages are initially "active"; and at regular intervals, the kernel sweeps over the memory, and tags some pages as "inactive". Whenever they are accessed again, they are immediately retagged "active". When the kernel is almost out of memory, and time comes to swap out to disk, the kernel will swap "inactive" pages.

active_file and inactive_file cache memory, with *active* and *inactive* similar to the *anon* memory above. The exact formula is cache = **active_file + inactive_file + tmpfs**. The exact rules used by the kernel to move memory pages between active and inactive sets are different from the ones used for anonymous memory, but the general principle is the same. Note that when the kernel needs to reclaim memory, it is cheaper to reclaim a clean (=non modified) page from this pool, since it can be reclaimed immediately (while anonymous pages and dirty/modified pages have to be written to disk first).

unevictable the amount of memory that cannot be reclaimed; generally, it will account for memory that has been "locked" with `mlock`. It is often used by crypto frameworks to make sure that secret keys and other sensitive material never gets swapped out to disk.

memory and memsw limits These are not really metrics, but a reminder of the limits applied to this cgroup. The first one indicates the maximum amount of physical memory that can be used by the processes of this control group; the second one indicates the maximum amount of RAM+swap.

Accounting for memory in the page cache is very complex. If two processes in different control groups both read the same file (ultimately relying on the same blocks on disk), the corresponding memory charge will be split between the control groups. It's nice, but it also means that when a cgroup is terminated, it could increase the memory usage of another cgroup, because they are not splitting the cost anymore for those memory pages.

CPU metrics: `cpuacct.stat`

Now that we've covered memory metrics, everything else will look very simple in comparison. CPU metrics will be found in the `cpuacct` controller.

For each container, you will find a pseudo-file `cpuacct.stat`, containing the CPU usage accumulated by the processes of the container, broken down between `user` and `system` time. If you're not familiar with the distinction, `user` is the time during which the processes were in direct control of the CPU (i.e. executing process code), and `system` is the time during which the CPU was executing system calls on behalf of those processes.

Those times are expressed in ticks of 1/100th of a second. Actually, they are expressed in "user jiffies". There are `USER_HZ` *"jiffies"* per second, and on x86 systems, `USER_HZ` is 100. This used to map exactly to the number of scheduler "ticks" per second; but with the advent of higher frequency scheduling, as well as tickless kernels, the number of kernel ticks wasn't relevant anymore. It stuck around anyway, mainly for legacy and compatibility reasons.

Block I/O metrics

Block I/O is accounted in the `blkio` controller. Different metrics are scattered across different files. While you can find in-depth details in the blkio-controller file in the kernel documentation, here is a short list of the most relevant

ones:

blkio.sectors contain the number of 512-bytes sectors read and written by the processes member of the cgroup, device by device. Reads and writes are merged in a single counter.

blkio.io_service_bytes indicates the number of bytes read and written by the cgroup. It has 4 counters per device, because for each device, it differentiates between synchronous vs. asynchronous I/O, and reads vs. writes.

blkio.io_serviced the number of I/O operations performed, regardless of their size. It also has 4 counters per device.

blkio.io_queued indicates the number of I/O operations currently queued for this cgroup. In other words, if the cgroup isn't doing any I/O, this will be zero. Note that the opposite is not true. In other words, if there is no I/O queued, it does not mean that the cgroup is idle (I/O-wise). It could be doing purely synchronous reads on an otherwise quiescent device, which is therefore able to handle them immediately, without queuing. Also, while it is helpful to figure out which cgroup is putting stress on the I/O subsystem, keep in mind that is is a relative quantity. Even if a process group does not perform more I/O, its queue size can increase just because the device load increases because of other devices.

8.3.5 Network Metrics

Network metrics are not exposed directly by control groups. There is a good explanation for that: network interfaces exist within the context of *network namespaces*. The kernel could probably accumulate metrics about packets and bytes sent and received by a group of processes, but those metrics wouldn't be very useful. You want per-interface metrics (because traffic happening on the local `lo` interface doesn't really count). But since processes in a single cgroup can belong to multiple network namespaces, those metrics would be harder to interpret: multiple network namespaces means multiple `lo` interfaces, potentially multiple `eth0` interfaces, etc.; so this is why there is no easy way to gather network metrics with control groups.

Instead we can gather network metrics from other sources:

IPtables

IPtables (or rather, the netfilter framework for which iptables is just an interface) can do some serious accounting.

For instance, you can setup a rule to account for the outbound HTTP traffic on a web server:

```
iptables -I OUTPUT -p tcp --sport 80
```

There is no `-j` or `-g` flag, so the rule will just count matched packets and go to the following rule.

Later, you can check the values of the counters, with:

```
iptables -nxvL OUTPUT
```

Technically, `-n` is not required, but it will prevent iptables from doing DNS reverse lookups, which are probably useless in this scenario.

Counters include packets and bytes. If you want to setup metrics for container traffic like this, you could execute a `for` loop to add two `iptables` rules per container IP address (one in each direction), in the `FORWARD` chain. This will only meter traffic going through the NAT layer; you will also have to add traffic going through the userland proxy.

Then, you will need to check those counters on a regular basis. If you happen to use `collectd`, there is a nice plugin to automate iptables counters collection.

Interface-level counters

Since each container has a virtual Ethernet interface, you might want to check directly the TX and RX counters of this interface. You will notice that each container is associated to a virtual Ethernet interface in your host, with a name like `vethKk8Zqi`. Figuring out which interface corresponds to which container is, unfortunately, difficult.

But for now, the best way is to check the metrics *from within the containers*. To accomplish this, you can run an executable from the host environment within the network namespace of a container using **ip-netns magic**.

The `ip-netns exec` command will let you execute any program (present in the host system) within any network namespace visible to the current process. This means that your host will be able to enter the network namespace of your containers, but your containers won't be able to access the host, nor their sibling containers. Containers will be able to "see" and affect their sub-containers, though.

The exact format of the command is:

```
ip netns exec <nsname> <command...>
```

For example:

```
ip netns exec mycontainer netstat -i
```

`ip netns` finds the "mycontainer" container by using namespaces pseudo-files. Each process belongs to one network namespace, one PID namespace, one `mnt` namespace, etc., and those namespaces are materialized under `/proc/<pid>/ns/`. For example, the network namespace of PID 42 is materialized by the pseudo-file `/proc/42/ns/net`.

When you run `ip netns exec mycontainer ...`, it expects `/var/run/netns/mycontainer` to be one of those pseudo-files. (Symlinks are accepted.)

In other words, to execute a command within the network namespace of a container, we need to:

- Find out the PID of any process within the container that we want to investigate;
- Create a symlink from `/var/run/netns/<somename>` to `/proc/<thepid>/ns/net`
- Execute `ip netns exec <somename> ...`.

Please review *Enumerating Cgroups* to learn how to find the cgroup of a pprocess running in the container of which you want to measure network usage. From there, you can examine the pseudo-file named `tasks`, which containes the PIDs that are in the control group (i.e. in the container). Pick any one of them.

Putting everything together, if the "short ID" of a container is held in the environment variable `$CID`, then you can do this:

```
TASKS=/sys/fs/cgroup/devices/$CID*/tasks
PID=$(head -n 1 $TASKS)
mkdir -p /var/run/netns
ln -sf /proc/$PID/ns/net /var/run/netns/$CID
ip netns exec $CID netstat -i
```

8.3.6 Tips for high-performance metric collection

Note that running a new process each time you want to update metrics is (relatively) expensive. If you want to collect metrics at high resolutions, and/or over a large number of containers (think 1000 containers on a single host), you do not want to fork a new process each time.

Here is how to collect metrics from a single process. You will have to write your metric collector in C (or any language that lets you do low-level system calls). You need to use a special system call, `setns()`, which lets the current process

enter any arbitrary namespace. It requires, however, an open file descriptor to the namespace pseudo-file (remember: that's the pseudo-file in `/proc/<pid>/ns/net`).

However, there is a catch: you must not keep this file descriptor open. If you do, when the last process of the control group exits, the namespace will not be destroyed, and its network resources (like the virtual interface of the container) will stay around for ever (or until you close that file descriptor).

The right approach would be to keep track of the first PID of each container, and re-open the namespace pseudo-file each time.

8.3.7 Collecting metrics when a container exits

Sometimes, you do not care about real time metric collection, but when a container exits, you want to know how much CPU, memory, etc. it has used.

Docker makes this difficult because it relies on `lxc-start`, which carefully cleans up after itself, but it is still possible. It is usually easier to collect metrics at regular intervals (e.g. every minute, with the collectd LXC plugin) and rely on that instead.

But, if you'd still like to gather the stats when a container stops, here is how:

For each container, start a collection process, and move it to the control groups that you want to monitor by writing its PID to the tasks file of the cgroup. The collection process should periodically re-read the tasks file to check if it's the last process of the control group. (If you also want to collect network statistics as explained in the previous section, you should also move the process to the appropriate network namespace.)

When the container exits, `lxc-start` will try to delete the control groups. It will fail, since the control group is still in use; but that's fine. You process should now detect that it is the only one remaining in the group. Now is the right time to collect all the metrics you need!

Finally, your process should move itself back to the root control group, and remove the container control group. To remove a control group, just `rmdir` its directory. It's counter-intuitive to `rmdir` a directory as it still contains files; but remember that this is a pseudo-filesystem, so usual rules don't apply. After the cleanup is done, the collection process can exit safely.

9.1 Most frequently asked questions.

9.1.1 ?

Docker100%, it is open source, so you can use it without paying.

9.1.2 ?

We are using the Apache License Version 2.0, see it here: https://github.com/dotcloud/docker/blob/master/LICENSE

9.1.3 Mac OS X Windows?

Not at this time, Docker currently only runs on Linux, but you can use VirtualBox to run Docker in a virtual machine on your box, and get the best of both worlds. Check out the *Mac OS X* and *WindowsDocker* installation guides.

9.1.4 ?

They are complementary. VMs are best used to allocate chunks of hardware resources. Containers operate at the process level, which makes them very lightweight and perfect as a unit of software delivery.

9.1.5 What does Docker add to just plain LXC?

Docker is not a replacement for LXC. "LXC" refers to capabilities of the Linux kernel (specifically namespaces and control groups) which allow sandboxing processes from one another, and controlling their resource allocations. On top of this low-level foundation of kernel features, Docker offers a high-level tool with several powerful functionalities:

- . Docker defines a format for bundling an application and all its dependencies into a single object which can be transferred to any Docker-enabled machine, and executed there with the guarantee that the execution environment exposed to the application will be the same. LXC implements process sandboxing, which is an important pre-requisite for portable deployment, but that alone is not enough for portable deployment. If you sent me a copy of your application installed in a custom LXC configuration, it would almost certainly not run on my machine the way it does on

yours, because it is tied to your machine's specific configuration: networking, storage, logging, distro, etc. Docker defines an abstraction for these machine-specific settings, so that the exact same Docker container can run - unchanged - on many different machines, with many different configurations.

- . Docker is optimized for the deployment of applications, as opposed to machines. This is reflected in its API, user interface, design philosophy and documentation. By contrast, the `lxc` helper scripts focus on containers as lightweight machines - basically servers that boot faster and need less RAM. We think there's more to containers than just that.
- . Docker includes *a tool for developers to automatically assemble a container from their source code*, with full control over application dependencies, build tools, packaging etc. They are free to use `make, maven, chef, puppet, salt,` Debian packages, RPMs, source tarballs, or any combination of the above, regardless of the configuration of the machines.
- . Docker includes git-like capabilities for tracking successive versions of a container, inspecting the diff between versions, committing new versions, rolling back etc. The history also includes how a container was assembled and by whom, so you get full traceability from the production server all the way back to the upstream developer. Docker also implements incremental uploads and downloads, similar to `git pull`, so new versions of a container can be transferred by only sending diffs.
- . Any container can be used as a *"base image"* to create more specialized components. This can be done manually or as part of an automated build. For example you can prepare the ideal Python environment, and use it as a base for 10 different applications. Your ideal Postgresql setup can be re-used for all your future projects. And so on.
- . Docker has access to a public registry where thousands of people have uploaded useful containers: anything from Redis, CouchDB, Postgres to IRC bouncers to Rails app servers to Hadoop to base images for various Linux distros. The *registry* also includes an official "standard library" of useful containers maintained by the Docker team. The registry itself is open-source, so anyone can deploy their own registry to store and transfer private containers, for internal server deployments for example.
- . Docker defines an API for automating and customizing the creation and deployment of containers. There are a huge number of tools integrating with Docker to extend its capabilities. PaaS-like deployment (Dokku, Deis, Flynn), multi-node orchestration (Maestro, Salt, Mesos, Openstack Nova), management dashboards (docker-ui, Openstack Horizon, Shipyard), configuration management (Chef, Puppet), continuous integration (Jenkins, Strider, Travis), etc. Docker is rapidly establishing itself as the standard for container-based tooling.

9.1.6 Docker?

There's a great StackOverflow answer showing the differences.

9.1.7

Not at all! Any data that your application writes to disk gets preserved in its container until you explicitly delete the container. The file system for the container persists even after the container halts.

9.1.8 How far do Docker containers scale?

Some of the largest server farms in the world today are based on containers. Large web deployments like Google and Twitter, and platform providers such as Heroku and dotCloud all run on container technology, at a scale of hundreds

of thousands or even millions of containers running in parallel.

9.1.9 Docker?

Currently the recommended way to link containers is via the *link* primitive. You can see details of how to work with links here.

Also of useful when enabling more flexible service portability is the Ambassador linking pattern.

9.1.10 Docker?

Any capable process supervisor such as http://supervisord.org/, runit, s6, or daemontools can do the trick. Docker will start up the process management daemon which will then fork to run additional processes. As long as the processor manager daemon continues to run, the container will continue to as well. You can see a more substantial example that uses supervisord here.

9.1.11 Docker?

Linux:

- Ubuntu 12.04, 13.04 et al
- Fedora 19/20+
- RHEL 6.5+
- Centos 6+
- Gentoo
- ArchLinux
- openSUSE 12.3+

Cloud:

- Amazon EC2
- Google Compute Engine
- Rackspace

9.1.12 ?

Definitely! You can fork the repo and edit the documentation sources.

9.1.13 ?

You can find more answers on:

- Docker user mailinglist
- Docker developer mailinglist
- IRC, docker on freenode
- GitHub

- Ask questions on Stackoverflow
- Join the conversation on Twitter

Looking for something else to read? Checkout the *Hello World* example.

/auth

/build

/commit

/containers

/events

/events:

/images

/info

/v1

/version