Docker Documentation

Release 0

Team Docker

July 07, 2015

Contents

This documentation has the following resources:

CHAPTER 1

Concepts

Contents:

1.1 Introduction

1.1.1 Docker - The Linux container runtime

Docker complements LXC with a high-level API which operates at the process level. It runs unix processes with strong guarantees of isolation and repeatability across servers.

Docker is a great building block for automating distributed systems: large-scale web deployments, database clusters, continuous deployment systems, private PaaS, service-oriented architectures, etc.

- **Heterogeneous payloads** Any combination of binaries, libraries, configuration files, scripts, virtualenvs, jars, gems, tarballs, you name it. No more juggling between domain-specific tools. Docker can deploy and run them all.
- **Any server** Docker can run on any x64 machine with a modern linux kernel - whether it's a laptop, a bare metal server or a VM. This makes it perfect for multi-cloud deployments.
- **Isolation** docker isolates processes from each other and from the underlying host, using lightweight containers.
- **Repeatability** Because containers are isolated in their own filesystem, they behave the same regardless of where, when, and alongside what they run.

1.1.2 What is a Standard Container?

Docker defines a unit of software delivery called a Standard Container. The goal of a Standard Container is to encapsulate a software component and all its dependencies in a format that is self-describing and portable, so that any compliant runtime can run it without extra dependency, regardless of the underlying machine and the contents of the container.

The spec for Standard Containers is currently work in progress, but it is very straightforward. It mostly defines 1) an image format, 2) a set of standard operations, and 3) an execution environment.

A great analogy for this is the shipping container. Just like Standard Containers are a fundamental unit of software delivery, shipping containers (http://bricks.argz.com/ins/7823-1/12) are a fundamental unit of physical delivery.

Standard operations

Just like shipping containers, Standard Containers define a set of STANDARD OPERATIONS. Shipping containers can be lifted, stacked, locked, loaded, unloaded and labelled. Similarly, standard containers can be started, stopped, copied, snapshotted, downloaded, uploaded and tagged.

Content-agnostic

Just like shipping containers, Standard Containers are CONTENT-AGNOSTIC: all standard operations have the same effect regardless of the contents. A shipping container will be stacked in exactly the same way whether it contains Vietnamese powder coffee or spare Maserati parts. Similarly, Standard Containers are started or uploaded in the same way whether they contain a postgres database, a php application with its dependencies and application server, or Java build artifacts.

Infrastructure-agnostic

Both types of containers are INFRASTRUCTURE-AGNOSTIC: they can be transported to thousands of facilities around the world, and manipulated by a wide variety of equipment. A shipping container can be packed in a factory in Ukraine, transported by truck to the nearest routing center, stacked onto a train, loaded into a German boat by an Australian-built crane, stored in a warehouse at a US facility, etc. Similarly, a standard container can be bundled on my laptop, uploaded to S3, downloaded, run and snapshotted by a build server at Equinix in Virginia, uploaded to 10 staging servers in a home-made Openstack cluster, then sent to 30 production instances across 3 EC2 regions.

Designed for automation

Because they offer the same standard operations regardless of content and infrastructure, Standard Containers, just like their physical counterpart, are extremely well-suited for automation. In fact, you could say automation is their secret weapon.

Many things that once required time-consuming and error-prone human effort can now be programmed. Before shipping containers, a bag of powder coffee was hauled, dragged, dropped, rolled and stacked by 10 different people in 10 different locations by the time it reached its destination. 1 out of 50 disappeared. 1 out of 20 was damaged. The process was slow, inefficient and cost a fortune - and was entirely different depending on the facility and the type of goods.

Similarly, before Standard Containers, by the time a software component ran in production, it had been individually built, configured, bundled, documented, patched, vendored, templated, tweaked and instrumented by 10 different people on 10 different computers. Builds failed, libraries conflicted, mirrors crashed, post-it notes were lost, logs were misplaced, cluster updates were half-broken. The process was slow, inefficient and cost a fortune - and was entirely different depending on the language and infrastructure provider.

Industrial-grade delivery

There are 17 million shipping containers in existence, packed with every physical good imaginable. Every single one of them can be loaded on the same boats, by the same cranes, in the same facilities, and sent anywhere in the World with incredible efficiency. It is embarrassing to think that a 30 ton shipment of coffee can safely travel half-way across the World in *less time* than it takes a software team to deliver its code from one datacenter to another sitting 10 miles away.

With Standard Containers we can put an end to that embarrassment, by making INDUSTRIAL-GRADE DELIVERY of software a reality.

Standard Container Specification

(TODO)

Image format

Standard operations

- Copy
- Run
- Stop
- Wait
- Commit
- Attach standard streams
- List filesystem changes
- ...

Execution environment

Root filesystem

Environment variables

Process arguments

Networking

Process namespacing

Resource limits

Process monitoring

Logging

Signals

Pseudo-terminal allocation

Security

1.2 Building blocks

1.2.1 Images

An original container image. These are stored on disk and are comparable with what you normally expect from a stopped virtual machine image. Images are stored (and retrieved from) repository

Images are stored on your local file system under /var/lib/docker/images

1.2.2 Containers

A container is a local version of an image. It can be running or stopped, The equivalent would be a virtual machine instance.

Containers are stored on your local file system under /var/lib/docker/containers

CHAPTER 2

Installation

Contents:

2.1 Ubuntu Linux

Please note this project is currently under heavy development. It should not be used in production.

Right now, the officially supported distributions are:

- Ubuntu 12.04 (precise LTS) (64-bit)
- Ubuntu 12.10 (quantal) (64-bit)

2.1.1 Dependencies

The linux-image-extra package is only needed on standard Ubuntu EC2 AMIs in order to install the aufs kernel module.

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

2.1.2 Installation

Docker is available as a Ubuntu PPA (Personal Package Archive), hosted on launchpad which makes installing Docker on Ubuntu very easy.

Add the custom package sources to your apt sources list. Copy and paste the following lines at once.

```
sudo sh -c "echo 'deb http://ppa.launchpad.net/dotcloud/lxc-docker/ubuntu precise main' >> /etc/apt/s
```

Update your sources. You will see a warning that GPG signatures cannot be verified.

```
sudo apt-get update
```

Now install it, you will see another warning that the package cannot be authenticated. Confirm install.

```
sudo apt-get install lxc-docker
```

Verify it worked

```
docker
```

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

2.2 Binaries

Please note this project is currently under heavy development. It should not be used in production.

Right now, the officially supported distributions are:

- Ubuntu 12.04 (precise LTS) (64-bit)
- Ubuntu 12.10 (quantal) (64-bit)

2.2.1 Install dependencies:

```
sudo apt-get install lxc bsdtar
sudo apt-get install linux-image-extra-`uname -r`
```

The linux-image-extra package is needed on standard Ubuntu EC2 AMIs in order to install the aufs kernel module.

Install the docker binary:

```
wget http://get.docker.io/builds/Linux/x86_64/docker-master.tgz
tar -xf docker-master.tgz
sudo cp ./docker-master /usr/local/bin
```

Note: docker currently only supports 64-bit Linux hosts.

2.2.2 Run the docker daemon

```
sudo docker -d &
```

2.2.3 Run your first container!

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

Continue with the *Hello World* example.

2.3 Arch Linux

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

Installing on Arch Linux is not officially supported but can be handled via either of the following AUR packages:

- lxc-docker
- lxc-docker-git

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

2.3.1 Dependencies

Docker depends on several packages which are specified as dependencies in either AUR package.

- aufs3
- bridge-utils
- go
- iproute2
- linux-aufs_friendly
- lxc

2.3.2 Installation

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.

Keep in mind that if **linux-aufs_friendly** is not already installed that a new kernel will be compiled and this can take quite a while.

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

2.3.3 Starting Docker

Prior to starting docker modify your bootloader to use the **linux-aufs_friendly** kernel and reboot your system.

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
```

2.4 Using Vagrant

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

Requirements: This guide will setup a new virtual machine with docker installed on your computer. This works on most operating systems, including MacOX, Windows, Linux, FreeBSD and others. If you can install these and have at least 400Mb RAM to spare you should be good.

2.4.1 Install Vagrant and Virtualbox

1. Install virtualbox from https://www.virtualbox.org/ (or use your package manager)
2. Install vagrant from http://www.vagrantup.com/ (or use your package manager)
3. Install git if you had not installed it before, check if it is installed by running `git` in a terminal window

2.4.2 Spin it up

1. Fetch the docker sources (this includes the Vagrantfile for machine setup).

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

2. Run vagrant from the sources directory

```
vagrant up
```

Vagrant will:

- Download the 'official' Precise64 base ubuntu virtual machine image from vagrantup.com
- Boot this image in virtualbox
- Add the Docker PPA sources to /etc/apt/sources.lst
- Update your sources
- Install lxc-docker

You now have a Ubuntu Virtual Machine running with docker pre-installed.

2.4.3 Connect

To access the VM and use Docker, Run `vagrant ssh` from the same directory as where you ran `vagrant up`. Vagrant will connect you to the correct VM.

```
vagrant ssh
```

2.4.4 Run

Now you are in the VM, run docker

```
docker
```

Continue with the *Hello World* example.

2.5 Windows (with Vagrant)

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

2.5.1 Requirements

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).

2.5.2 Opening a command prompt

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.

2.5.3 Launch an Ubuntu virtual server

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.

2.5.4 Log onto your Ubuntu server

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
```

Using 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

Use 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 from a terminal

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 !

2.5.5 Running 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.

2.6 Amazon EC2

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

2.6.1 Installation

Docker can now be installed on Amazon EC2 with a single vagrant command. 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.

```
export AWS_ACCESS_KEY_ID=xxx
export AWS_SECRET_ACCESS_KEY=xxx
export AWS_KEYPAIR_NAME=xxx
export AWS_SSH_PRIVKEY=xxx
```

   The environment variables are:

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

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

```
echo $AWS_ACCESS_KEY_ID
```

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 the https://github.com/mitchellh/vagrant-aws

7. Connect to your machine

```
vagrant ssh
```

8. Your first command

   Now you are in the VM, run docker

```
    docker
```

Continue with the *Hello World* example.

2.7 Upgrading

These instructions are for upgrading your Docker binary for when you had a custom (non package manager) installation. If you istalled docker using apt-get, use that to upgrade.

Get the latest docker binary:

```
wget http://get.docker.io/builds/$(uname -s)/$(uname -m)/docker-master.tgz
```

Unpack it to your current dir

```
tar -xf docker-master.tgz
```

Stop your current daemon. How you stop your daemon depends on how you started it.

- If you started the daemon manually (`sudo docker -d`), you can just kill the process: `killall docker`
- If the process was started using upstart (the ubuntu startup daemon), you may need to use that to stop it

Start docker in daemon mode (-d) and disconnect (&) starting ./docker will start the version in your current dir rather than the one in your PATH.

Now start the daemon

```
sudo ./docker -d &
```

Alternatively you can replace the docker binary in `/usr/local/bin`

CHAPTER 3

Examples

Contents:

3.1 Running The Examples

All the examples assume your machine is running the docker daemon. To run the docker daemon in the background, simply type:

```
sudo docker -d &
```

Now you can run docker in client mode: all commands will be forwarded to the docker daemon, so the client can run from any account.

```
# now you can run docker commands from any account.
docker help
```

3.2 Hello World

Note: This example assumes you have Docker running in daemon mode. For more information please see *Running The Examples*

This is the most basic example available for using Docker.

Download the base container

```
# Download a base image
docker pull base
```

The *base* image is a minimal *ubuntu* based container, alternatively you can select *busybox*, a bare minimal linux system. The images are retrieved from the docker repository.

```
#run a simple echo command, that will echo hello world back to the console over standard out.
docker run base /bin/echo hello world
```

Explanation:

- **"docker run"** run a command in a new container
- **"base"** 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

Continue to the *Hello World Daemon* example.

3.3 Hello World Daemon

Note: This example assumes you have Docker running in daemon mode. For more information please see *Running The Examples*

The most boring daemon ever written.

This example assumes you have Docker installed and with the base image already imported `docker pull base`. We will use the base 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=$(docker run -d base /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 base image.

- **"docker run -d "** run a command in a new container. We pass "-d" so it runs as a daemon.
- **"base"** 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.

```
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.

```
docker attach $CONTAINER_ID
```

Attach to the container to see the results in realtime.

- **"docker attach**" This will allow us to attach to a background process to see what is going on.
- **$CONTAINER_ID** The Id of the container we want to attach too.

```
docker ps
```

Check the process list to make sure it is running.

- **"docker ps"** this shows all running process managed by docker

```
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.

```
docker ps
```

Make sure it is really stopped.

Video:

See the example in action

Continue to the *Building a python web app* example.

3.4 Building a python web app

Note: This example assumes you have Docker running in daemon mode. For more information please see *Running The Examples*

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:

```
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=$(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. BUILD_JOB will be set with the new container_id.

```
docker attach $BUILD_JOB
[...]
```

We attach to the new container to see what is going on. Ctrl-C to disconnect

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

Save the changed 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=$(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.

```
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=$(docker port $WEB_WORKER 5000)
```

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

```
curl http://`hostname`:$WEB_PORT
  Hello world!
```

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

Video:

See the example in action

Continue to *Create an ssh daemon service*.

3.5 Create a redis service

Note: This example assumes you have Docker running in daemon mode. For more information please see *Running The Examples*

Very simple, no frills, redis service.

3.5.1 Open a docker container

```
docker run -i -t base /bin/bash
```

3.5.2 Building your image

Update your docker container, install the redis server. Once installed, exit out of docker.

```
apt-get update
apt-get install redis-server
exit
```

3.5.3 Snapshot the installation

```
docker ps -a  # grab the container id (this will be the last one in the list)
docker commit <container_id> <your username>/redis
```

3.5.4 Run the service

Running the service with *-d* runs the container in detached mode, leaving the container running in the background. Use your snapshot.

```
docker run -d -p 6379 <your username>/redis /usr/bin/redis-server
```

Test 1

Connect to the container with the redis-cli.

```
docker ps  # grab the new container id
docker inspect <container_id>    # grab the ipaddress of the container
redis-cli -h <ipaddress> -p 6379
redis 10.0.3.32:6379> set docker awesome
OK
redis 10.0.3.32:6379> get docker
"awesome"
redis 10.0.3.32:6379> exit
```

Test 2

Connect to the host os with the redis-cli.

```
docker ps  # grab the new container id
docker port <container_id> 6379  # grab the external port
ifconfig   # grab the host ip address
redis-cli -h <host ipaddress> -p <external port>
redis 192.168.0.1:49153> set docker awesome
OK
redis 192.168.0.1:49153> get docker
"awesome"
redis 192.168.0.1:49153> exit
```

3.6 Create an ssh daemon service

Note: This example assumes you have Docker running in daemon mode. For more information please see *Running The Examples*

Video:

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

You can also get this sshd container by using

```
docker pull dhrp/sshd
```

The password is 'screencast'

CHAPTER 4

Contributing

4.1 Contributing to Docker

Want to hack on Docker? Awesome! There are instructions to get you started on the website: http://docker.io/gettingstarted.html

They are probably not perfect, please let us know if anything feels wrong or incomplete.

4.1.1 Contribution guidelines

Pull requests are always welcome

We are always thrilled to receive pull requests, and do our best to process them as fast as possible. Not sure if that typo is worth a pull request? Do it! We will appreciate it.

If your pull request is not accepted on the first try, don't be discouraged! If there's a problem with the implementation, hopefully you received feedback on what to improve.

We're trying very hard to keep Docker lean and focused. We don't want it to do everything for everybody. This means that we might decide against incorporating a new feature. However, there might be a way to implement that feature *on top of* docker.

Discuss your design on the mailing list

We recommend discussing your plans on the mailing list before starting to code - especially for more ambitious contributions. This gives other contributors a chance to point you in the right direction, give feedback on your design, and maybe point out if someone else is working on the same thing.

Create issues...

Any significant improvement should be documented as a github issue before anybody starts working on it.

...but check for existing issues first!

Please take a moment to check that an issue doesn't already exist documenting your bug report or improvement proposal. If it does, it never hurts to add a quick "+1" or "I have this problem too". This will help prioritize the most common problems and requests.

Conventions

Fork the repo and make changes on your fork in a feature branch:

- If it's a bugfix branch, name it XXX-something where XXX is the number of the issue
- If it's a feature branch, create an enhancement issue to announce your intentions, and name it XXX-something where XXX is the number of the issue.

Submit unit tests for your changes. Go has a great test framework built in; use it! Take a look at existing tests for inspiration. Run the full test suite on your branch before submitting a pull request.

Make sure you include relevant updates or additions to documentation when creating or modifying features.

Write clean code. Universally formatted code promotes ease of writing, reading, and maintenance. Always run `go fmt` before committing your changes. Most editors have plugins that do this automatically, and there's also a git pre-commit hook:

```
curl -o .git/hooks/pre-commit https://raw.github.com/edsrzf/gofmt-git-hook/master/fmt-check && chmod
```

Pull requests descriptions should be as clear as possible and include a reference to all the issues that they address.

Code review comments may be added to your pull request. Discuss, then make the suggested modifications and push additional commits to your feature branch. Be sure to post a comment after pushing. The new commits will show up in the pull request automatically, but the reviewers will not be notified unless you comment.

Before the pull request is merged, make sure that you squash your commits into logical units of work using `git rebase -i` and `git push -f`. After every commit the test suite should be passing. Include documentation changes in the same commit so that a revert would remove all traces of the feature or fix.

Commits that fix or close an issue should include a reference like `Closes #XXX` or `Fixes #XXX`, which will automatically close the issue when merged.

Add your name to the AUTHORS file, but make sure the list is sorted and your name and email address match your git configuration. The AUTHORS file is regenerated occasionally from the git commit history, so a mismatch may result in your changes being overwritten.

4.2 Setting up a dev environment

Instructions that have been verified to work on Ubuntu 12.10,

```
sudo apt-get -y install lxc wget bsdtar curl golang git

export GOPATH=~/go/
export PATH=$GOPATH/bin:$PATH

mkdir -p $GOPATH/src/github.com/dotcloud
cd $GOPATH/src/github.com/dotcloud
git clone git@github.com:dotcloud/docker.git
cd docker

go get -v github.com/dotcloud/docker/...
go install -v github.com/dotcloud/docker/...
```

Then run the docker daemon,

```
sudo $GOPATH/bin/docker -d
```

Run the `go install` command (above) to recompile docker.

CHAPTER 5

Commands

Contents:

5.1 The basics

5.1.1 Starting Docker

If you have used one of the quick install paths', Docker may have been installed with upstart, Ubuntu's system for starting processes at boot time. You should be able to run `docker help` and get output.

If you get `docker: command not found` or something like `/var/lib/docker/repositories: permission denied` you will need to specify the path to it and manually start it.

```
# Run docker in daemon mode
sudo <path to>/docker -d &
```

5.1.2 Running an interactive shell

```
# Download a base image
docker pull base

# Run an interactive shell in the base image,
# allocate a tty, attach stdin and stdout
docker run -i -t base /bin/bash
```

5.1.3 Starting a long-running worker process

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

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

# Kill the job
docker kill $JOB
```

5.1.4 Listing all running containers

```
docker ps
```

5.1.5 Expose a service on a TCP port

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

# Which public port is NATed to my container?
PORT=$(docker port $JOB 4444)

# Connect to the public port via the host's public address
# Please note that because of how routing works connecting to localhost or 127.0.0.1 $PORT will not w
IP=$(ifconfig eth0 | perl -n -e 'if (m/inet addr:([\d\.]+)/g) { print $1 }')
echo hello world | nc $IP $PORT

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

5.1.6 Committing (saving) an image

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 `docker images`

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

# List your containers
docker images
```

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

Read more about *Working with the repository* or continue to the complete *Command Line Interface*

5.2 Working with the repository

5.2.1 Connecting to the repository

You create a user on the central docker repository by running

```
docker login
```

If your username does not exist it will prompt you to also enter a password and your e-mail address. It will then automatically log you in.

5.2.2 Committing a container to a named image

In order to commit to the repository it is required to have committed your container to an image with your namespace.

```
# for example docker commit $CONTAINER_ID dhrp/kickassapp
docker commit <container_id> <your username>/<some_name>
```

5.2.3 Pushing a container to the repository

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

Now you can commit this image to the repository

```
# for example docker push dhrp/kickassapp
docker push <image-name>
```

5.3 Command Line Interface

5.3.1 Docker Usage

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

```
$ docker
  Usage: docker COMMAND [arg...]

  A self-sufficient runtime for linux containers.

  ...
```

5.3.2 Available Commands

`attach` – Attach to a running container

```
Usage: docker attach CONTAINER

Attach to a running container
```

`build` – Build a container from Dockerfile via stdin

```
Usage: docker build -
Example: cat Dockerfile | docker build -
Build a new image from the Dockerfile passed via stdin
```

`commit` – Create a new image from a container's changes

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

Create a new image from a container's changes

  -m="": Commit message
```

`diff` – Inspect changes on a container's filesystem

```
Usage: docker diff CONTAINER [OPTIONS]

Inspect changes on a container's filesystem
```

`export` – Stream the contents of a container as a tar archive

```
Usage: docker export CONTAINER

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

`history` – Show the history of an image

```
Usage: docker history [OPTIONS] IMAGE

Show the history of an image
```

`images` – List images

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

List images

  -a=false: show all images
  -q=false: only show numeric IDs
```

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

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

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

`info` – Display system-wide information

```
Usage: docker info

Display system-wide information.
```

`inspect` – Return low-level information on a container

```
Usage: docker inspect [OPTIONS] CONTAINER

Return low-level information on a container
```

`kill` – Kill a running container

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

Kill a running container
```

`login` – Register or Login to the docker registry server

```
Usage: docker login

Register or Login to the docker registry server
```

`logs` – Fetch the logs of a container

```
Usage: docker logs [OPTIONS] CONTAINER

Fetch the logs of a container
```

`port` – Lookup the public-facing port which is NAT-ed to PRIVATE_PORT

```
Usage: docker port [OPTIONS] CONTAINER PRIVATE_PORT

Lookup the public-facing port which is NAT-ed to PRIVATE_PORT
```

`ps` – List containers

```
Usage: docker ps [OPTIONS]

List containers

  -a=false: Show all containers. Only running containers are shown by default.
  -notrunc=false: Don't truncate output
  -q=false: Only display numeric IDs
```

`pull` – Pull an image or a repository from the docker registry server

```
Usage: docker pull NAME

Pull an image or a repository from the registry
```

`push` – Push an image or a repository to the docker registry server

```
Usage: docker push NAME

Push an image or a repository to the registry
```

`restart` – Restart a running container

```
Usage: docker restart [OPTIONS] NAME

Restart a running container
```

`rm` – Remove a container

```
Usage: docker rm [OPTIONS] CONTAINER

Remove a container
```

`rmi` – Remove an image

```
Usage: docker rmimage [OPTIONS] IMAGE

Remove an image
```

`run` – Run a command in a new container

```
Usage: docker run [OPTIONS] IMAGE COMMAND [ARG...]

Run a command in a new container

  -a=map[]: Attach to stdin, stdout or stderr.
  -d=false: Detached mode: leave the container running in the background
  -e=[]: Set environment variables
  -h="": Container host name
  -i=false: Keep stdin open even if not attached
  -m=0: Memory limit (in bytes)
  -p=[]: Map a network port to the container
  -t=false: Allocate a pseudo-tty
  -u="": Username or UID
```

`start` – Start a stopped container

```
Usage: docker start [OPTIONS] NAME

Start a stopped container
```

`stop` – Stop a running container

```
Usage: docker stop [OPTIONS] NAME

Stop a running container
```

`tag` – Tag an image into a repository

```
Usage: docker tag [OPTIONS] IMAGE REPOSITORY [TAG]

Tag an image into a repository

  -f=false: Force
```

`version` – Show the docker version information

`wait` – Block until a container stops, then print its exit code

```
Usage: docker wait [OPTIONS] NAME

Block until a container stops, then print its exit code.
```

CHAPTER 6

Builder

Contents:

6.1 Docker Builder

Table of Contents

6.1.1 1. Format

The Docker builder format is quite simple:

```
instruction arguments
```

The first instruction must be *FROM*

All instruction are to be placed in a file named *Dockerfile*

In order to place comments within a Dockerfile, simply prefix the line with "#"

6.1.2 2. Instructions

Docker builder comes with a set of instructions:

1. FROM: Set from what image to build
2. RUN: Execute a command
3. INSERT: Insert a remote file (http) into the image

2.1 FROM

```
FROM <image>
```

The *FROM* instruction must be the first one in order for Builder to know from where to run commands.

FROM can also be used in order to build multiple images within a single Dockerfile

2.2 RUN

```
RUN <command>
```

The *RUN* instruction is the main one, it allows you to execute any commands on the *FROM* image and to save the results. You can use as many *RUN* as you want within a Dockerfile, the commands will be executed on the result of the previous command.

2.3 INSERT

```
INSERT <file url> <path>
```

The *INSERT* instruction will download the file at the given url and place it within the image at the given path.

Note: The path must include the file name.

6.1.3 3. Dockerfile Examples

```
# Nginx
#
# VERSION               0.0.1
# DOCKER-VERSION        0.2

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

run apt-get install -y inotify-tools nginx apache openssh-server
insert https://raw.github.com/creack/docker-vps/master/nginx-wrapper.sh /usr/sbin/nginx-wrapper
```

```
# Firefox over VNC
#
# VERSION               0.3
# DOCKER-VERSION        0.2

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 to do it, but it does the trick)
run bash -c 'echo "firefox" >> /.bashrc'
```

CHAPTER 7

FAQ

7.1 Most frequently asked questions.

1. **How much does Docker cost?**

   Docker is 100% free, it is open source, so you can use it without paying.

2. **What open source license are you using?**

   We are using the Apache License Version 2.0, see it here: https://github.com/dotcloud/docker/blob/master/LICENSE

3. **Does Docker run on Mac OS X or 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 MacOSX and Windows intallation guides.

4. **How do containers compare to virtual machines?**

   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.

5. **Can I help by adding some questions and answers?**

   Definitely! You can fork the repo and edit the documentation sources.

42. **Where can I find more answers?**

    You can find more answers on:

    - 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.