



W rkshop

Barry Evans: using some slides from Stefan Koospal &

Mohsen Haghaieghshenasfard

Pictures: Stefan Koospal

:

https://creativecommons.org/licenses/by/3.0/de/legalcode



•

**Why ?**

•

Installing

Docker on

Linux

Running Your First

Image

•

The

Basic Commands

•

Create a Dockerized Cowsay

Application

•

Building Images from Dockerfile

•

Working

with Registries

•

Docker Fundamentals

•

Dockerfile Instructions

•

Connecting Containers to the World

•

Common Docker Commands

•

Managing

Containers

•

Practical Section

**Agenda**



**Why ?**

Resources

Updates

Fail Safety

Recovery

Roll Back

Security

Portability

Easy Application Delivery



**Virtualisation**

Hypervisor

(

different OS

)

–

VM

–

XEN

–

Virtual Box

OS

-

Container

(

Using

one

Kernel)

–

openvz

–

zone

–

jails

–

lxc



**Docker**

What

makes

the

different

to

lxc

,

openvz

,

jails

,

zones

–

Using

only

kernel

a

nd

network

–

No

special

kernel

–

Using

l

ayers

–

Offers

repositories

–

Offers

orchestration



**Docker (Wikipedia I)**

•

Container

•

Is

a

running

virtual

OS

executing

one

ore

more

applications

•

Image

•

Is

a portable

memmory

image

to

run

as

container

•

Dockerfile

•

Is

a

textfile

with

commands

to

create

an

image



**Docker (Wikipedia**

**II)**

•

Docker

Hub

•

A

registry

to

store

docker

images

•

libcontainer

•

An

interface

to

basic

functions

of

docker

•

Libswarm

(

Kubernetes

)

•

An

interface

for

orchestration

•

libchan

•

An

interface

to

the

docker

network



**Hypervisor**

**-**

**VMs Versus Docker Containers**

Hardware

Hardware

Hypervisor

Linux 64 Kernel ++

Docker

App1

App2

App3

Python

Java

Php

ssh

OS1

App4

App5

App3

Python

Java

Php

ssh

OS2

App1

App6

App2

Python

Java

Php

ssh

OS1

Debian

Suse

Layer Apache

Layer

nginx

java

php

java

python

python

A

6

A

4

A

2

A

3

A

5

A

Y

A

X

A

1

A

9

A

8

A

7

A

1

# Containers Versus VMs (cont.)

* Containers share resources with the host OS, which makes them an order of magnitude more efficient. Containers can be started and stopped in a fraction of a second.
* The portability of containers has the potential to eliminate a whole class of bugs caused by subtle changes in the running environment.
* The lightweight nature of containers means developers can run dozens of containers at the same time, making it possible to emulate a production-ready distributed system.
* Users can download and run complex applications without needing to spend hours on configuration and installation issues.

# The What and Why of Containers

* Containers are fundamentally changing the way we develop, distribute, and run software. Developers can build software locally, knowing that it will run identically regardless of host environment.
* Operations engineers can concentrate on networking, resources, and uptime and spend less time configuring environments. Containers are also an encapsulation of an application with its dependencies.
* Docker containers share the underlying resources of the Docker host. Containers are very small (some base OS images are less than 3MBs) start up very quickly (< 3/8s of a second) because you’re not booting a full operating system. You’re just starting a process.



**The What and**

**Why**

**Before**

**Docker**

•

Ship

packages: deb, rpm, gem, jar...

•

Dependency

hell.

•

"

Works on my machine."

•

Base

deployment often done from scratch (

debootstrap

...)

and unreliable.

**After**

**Docker**

•

Ship

container images with all their dependencies.

•

Break

image into layers.

•

Only

ship layers that have changed.

•

Save

disk, network, memory usage.



**Docker and Containers**

Containers are an old concept

. Some examples are:

•

UNIX

systems have had the chroot

command that

provides a simple

form of

filesystem

isolation.

•

FreeBSD has had the jail utility, which extended chroot sandboxing to

processes.

•

Solaris Zones offered a comparatively complete containerization technology

around

2001

but was limited to the Solaris OS

.

But:

**Docker**

took the existing Linux container technology and wrapped and extended

it

in various ways

—

primarily through portable images and a user

-

friendly

interface

—

to

create a complete solution for the creation and distribution of

containers.



**Docker Components**

The Docker platform has

two main components:

•

**Docker host**

which

provides a fast and convenient interface

for

creating

images and

running

containers

.

•

**Registry**

**Service (Docker Hub or Docker Trusted**

**Registry),**

Cloud

or server

based storage and distribution service for your

images. It provides

an

enormous number of public container images for download, allowing users to

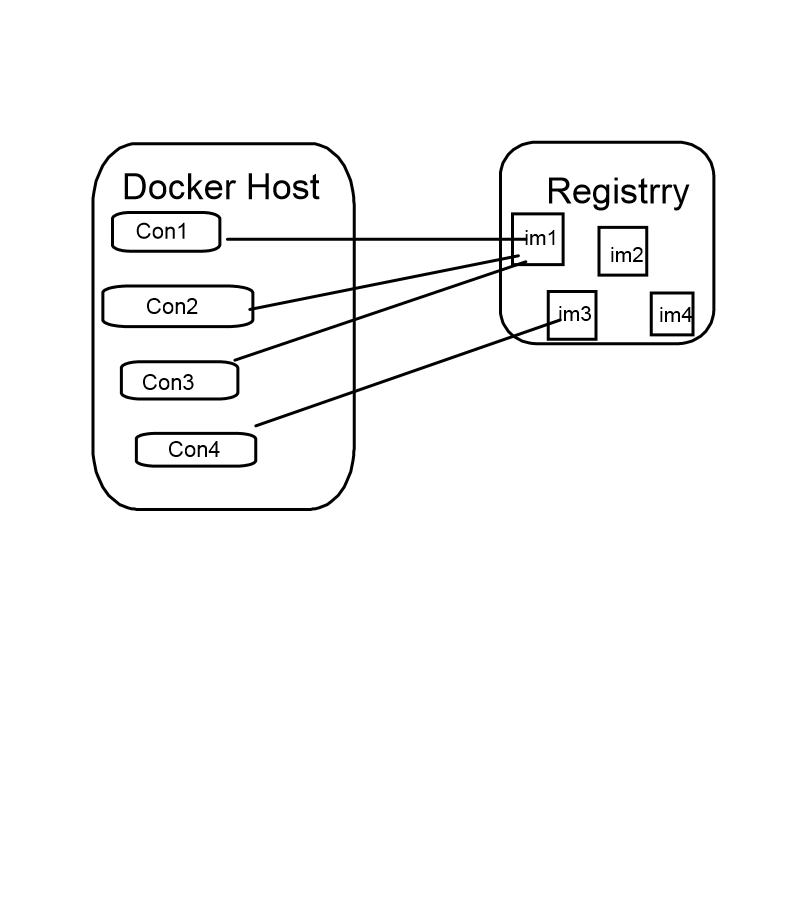
quickly get started and avoid duplicating work already done by others

.



**Docker**

**Components (cont.)**



•

The What and Why of Containers

•

**Installing Docker on Linux Running Your First Image**

•

The

Basic Commands

•

Create a Dockerized Cowsay

Application

•

Building Images from Dockerfile

•

Working

with Registries

•

Docker Fundamentals

•

Dockerfile Instructions

•

Connecting Containers to the World

•

Common Docker Commands

•

Managing

Containers

•

Practical Section

**Agenda**



**Installing Docker on Linux**

By far the best way to install Docker on Linux is through the

**installation script**

provided

by Docker

.

You should be able to the use the script provided

at the following link

to

automatically

install Docker. The official instructions will tell you to simply

run:

**curl**

**-**

**sSL**

**https://get.docker.com/ |**

**sh**

**wget**

**-**

**qO**

**-**

**https://get.docker.com/**

**|**

**sh**

Note: installing

Docker

Requires 64 bit Linux and a least

**kernel 3.10**



**A**

**Q**

**uick Check**

Just

to make sure everything is installed correctly and working, try running

the

**docker**

**version**

command. The output is like this:

Client:

Version:

17.12.0

-

ce

API

version

:

1.35

Go

version

:

go1.9.2

Git

commit

:

c97c6d6

Built

:

Wed

Dec

27 20:11:19 2017

OS/

Arch

:

linux

/amd64

Server:

Engine:

Version:

17.12.0

-

ce

API

version

:

1.35 (

minimum

version

1.12)

Go

version

:

go1.9.2

Git

commit

:

c97c6d6

Built

:

Wed

Dec

27 20:09:54 2017

OS/

Arch

:

linux

/amd64

Experimental:

false

**Docker run ubuntu echo “Hello DubJUG”**



**Running Your First Image**

To

test Docker is installed correctly, try running:

#

Unable

to

find

image

'

ubuntu:latest

'

locally

latest

:

Pulling

from

library

/

ubuntu

1

be7f2b886e8: Pull

complete

6

fbc4a21b806: Pull

complete

c71a6f8e1378: Pull

complete

4

be3072e5a37: Pull

complete

06

c6d2f59700: Pull

complete

Digest: sha256:e27e9d7f7f28d67aa9e2d7540bdc2b33254b452ee8

e60f388875e5b7d9b2b696

Status:

Downloaded

newer

image

for

ubuntu:latest

Hello

DubJUG



**What**

**happens**

**now**

**?**

And

why

?

#

**Docker run ubuntu echo “Hello DubJUG”**



**Differences between containers and**

**images**

•

An image is a read

-

only filesystem.

•

A container is an encapsulated set of processes running in a read

-

write

copy of that filesystem.

•

To optimize container boot time, copy

-

on

-

write is used instead of regular

copy.

•

**docker run**

starts a container from a given image.



**Running Your First**

**Image (cont.)**

We can ask Docker to give us a shell inside a container with the following

command:

#

**docker**

**run**

**-**

**i**

**-**

**t**

**ubuntu**

**"/bin/**

**bash**

**"**

root@5aadb5ce8631:/# echo "

Hello

Containerworld

"

Hello

Containerworld

root@5aadb5ce8631:/#

exit

exit



•

The What and Why of Containers

•

Installing Docker on Linux Running Your First Image

•

**The Basic Commands**

•

Create a Dockerized Cowsay

Application

•

Building Images from Dockerfile

•

Working

with Registries

•

Docker Fundamentals

•

Dockerfile Instructions

•

Connecting Containers to the World

•

Common Docker Commands

•

Managing

Containers

•

Practical Section

**Agenda**



Let’s try to understand Docker a bit more by launching a container and seeing

what effect various commands and actions have. First, let’s launch a new

container;

but

this time, we’ll give it a new hostname with the

-

h flag

:

The name of

container

may be

**infallible\_bhaskara**

. Docker

-

generated

names

are a random adjective followed by the name of a famous scientist, engineer,

or

hacker

. You can instead set the name by using the

--

name

argument.

**The Basic Commands**

#

**docker**

**run**

**-**

**h**

**container**

**-**

**i**

**-**

**t**

**ubuntu**

**"/bin/**

**bash**

**"**

root@container

:

/#



Get

more information on a given container by

running

**docker**

**inspect**

with the

name or ID of the container

:

#

**docker**

**inspect**

**infallible\_bhaskara**

[

{

"Id":

"f5b0bd3817f632ad5e30efc13cd12fbe1e613a32990ab42f75fea332dc546cef",

"Created": "2018

-

02

-

06

T16:51:51.25395522Z",

"Path": "/bin/bash",

"

Args

": [],

"State": {

"Status": "running",

**The Basic**

**Commands (cont.)**



Use

**grep**

or the

**--**

**format argument**

to

filter for the

information we’re

interested

in

.

**The Basic**

**Commands (cont.)**

#

**docker inspect infallible bhaskara | grep IPAddress**

"

SecondaryIPAddresses

": null,

"

IPAddress

": "172.17.0.4

",

#

**docker**

**inspect**

**--**

**format {{.**

**NetworkSettings.IPAddress**

**}}**

**infallible\_bhaskara**

172.17.0.4



**docker**

**diff:**

root@container

:

/

tmp

# touch /

tmp

/xx

#

docker diff infallible\_bhaskara

C /tmp

A /

tmp/xx

Here is

the list of files that have changed in the running

container compared with

the original image.

**The Basic**

**Commands (cont.)**



**docker**

**logs:**

docker

logs

infallible\_bhaskara

root@container

:

/# cd

tmp

root@container

:

/

tmp

# touch /

tmp

/xx

If

you run this command with the name of your container, you will get a list of

everything that’s happened inside the container:

**The Basic**

**Commands (cont.)**



**docker**

**rm:**

To get rid of

the container

, use the docker rm command

**docker**

**rm**

**infallible\_bhaskara**

If you want to get rid of all your stopped containers, you can

use the following

command

which gets

the IDs

of all stopped containers. For example:

**The Basic**

**Commands (cont.)**

**docker**

**rm**

**-**

**v $(**

**docker**

**ps**

**-**

**aq**

**-**

**f**

**status**

**=**

**exited**

**)**



•

The What and Why of Containers

•

Installing Docker on Linux Running Your First Image

•

The Basic Commands

•

**Create a Dockerized Cowsay**

**Application**

•

Building Images from Dockerfile

•

Working

with Registries

•

Docker Fundamentals

•

Dockerfile Instructions

•

Connecting Containers to the World

•

Common Docker Commands

•

Managing

Containers

•

Practical Section

**Agenda**



**Create**

**a Dockerized**

**Cowsay Application**

#

**docker**

**run**

**--**

**name**

**cowsay**

**-**

**h**

**cowsay**

**-**

**i**

**-**

**t**

**ubuntu**

**"/bin/**

**bash**

**"**

root@cowsay

:

/#

apt

-

get

update

…

root@cowsay

:

/#

apt

-

get

install

-

y

fortune

-

mod

cowsay

...

root@cowsay

:

/#

/

usr

/games/fortune |/

usr

/games/

cowsay

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<

You will pass away very quickly.

>

----------------------------------

\

^\_\_^

\

(

oo

)

\

\_\_\_\_\_\_\_

(

\_\_

)

\

)

\

/

\

||

----

w |

|| ||



**docker**

**commit:**

To turn

the cowsay container

into an image

,

use

the docker

commit command. It

doesn’t matter if the container is running or stopped

.

root@cowsay

:

/ # exit

exit

#

**docker**

**commit**

**cowsay**

**test/**

**cowsay**

sha256:7a09e1aa2872ff37258e0557670bd8d9e166ddd9a5b400d510d5ac77c9b23ab2

The

returned value is the unique ID of our image.

**The Basic**

**Commands (cont.)**



Now we have an image with cowsay installed that we can run:

This

is great! However, there are a few problems. If we need to change something,

we have

to manually repeat our steps from that point.

**The Basic**

**Commands (cont.)**

~/

cowsay

#

**docker**

**run**

**test**

**/**

**cowsay**

**"/**

**usr**

**/**

**games**

**/**

**cowsay**

**" "Muh"**

\_\_\_\_\_

<

Muh

>

-----

\

^\_\_^

\

(

oo

)

\

\_\_\_\_\_\_\_

(

\_\_

)

\

)

\

/

\

||

----

w |

|| ||



For example, if we want to use a different base image;

•

we would have to start again from scratch.

•

More

importantly,

it isn’t

easily

repeatable; it’s

difficult and potentially error

-

prone to share or repeat

the set

of steps required to create the image.

The

solution to this is to use a

**Dockerfile**

to create

an automated build for the

image.

**The Basic**

**Commands (cont.)**



•

The What and Why of Containers

•

Installing Docker on Linux Running Your First Image

•

The Basic Commands

•

Create a Dockerized Cowsay Application

•

**Building Images from Dockerfile**

•

Working

with Registries

•

Docker Fundamentals

•

Dockerfile Instructions

•

Connecting Containers to the World

•

Common Docker Commands

•

Managing

Containers

•

Practical Section

**Agenda**



A Dockerfile is simply a text file that contains a set of steps that can be used to

create a

Docker image. Start by creating a new folder and file for this example:

#

**mkdir**

**cowsay**

#

**cd**

**cowsay**

~/

cowsay

#

**touch**

**Dockerfile**

And

insert the following contents into

Dockerfile

:

FROM

ubuntu

RUN

apt

-

get update && apt

-

get install

-

y fortune

-

mod

cowsay

**Building Images from Dockerfile**



We can now build the image by running the

**docker build**

command

inside the

same directory:

~/

cowsay

#

**ls**

**Dockerfile**

Dockerfile

~/

cowsay

#

**docker**

**build**

**-**

**t test/**

**cowsay**

**-**

**dockerfile**

**.**

Sending build context to Docker daemon 2.048kB

Step 1/2 : FROM

ubuntu

---

>

0458a4468cbc

Step 2/2 : RUN apt

-

get update && apt

-

get install

-

y fortune

-

mod

cowsay

---

>

Running in 7ddeeca5dca

9

….

removing

intermediate container 7ddeeca5dca9

---

>

72359aa0bff

8

Successfully built 72359aa0bff8

Successfully tagged test/

cowsay

-

dockerfile:latest

**Building Images from**

**Dockerfile (cont.)**



Then we can run the image in the same way as before

:

~/

cowsay

#

**docker**

**run test/**

**cowsay**

**-**

**dockerfile**

**/**

**usr**

**/games/**

**cowsay**

**"**

**Muh**

**"**

\_\_\_\_\_

<

Muh

>

-----

\

^\_\_^

\

(

oo

)

\

\_\_\_\_\_\_\_

(

\_\_

)

\

)

\

/

\

||

----

w |

|| ||

**Building Images from**

**Dockerfile (cont.)**



But

we can actually make things a little bit easier for the user by taking advantage

of the

ENTRYPOINT Dockerfile instruction. The ENTRYPOINT instruction lets us

specify

an executable

that is used to handle any arguments passed to docker run.

Add the following line to the bottom of the

Dockerfile

:

ENTRYPOINT "/

usr

/games/

cowsay

" "

Muh

“

~/

cowsay

#

**docker**

**build**

**-**

**t test/**

**cowsay**

**-**

**dockerfile**

**.**

~/

cowsay

#

**docker**

**run test/**

**cowsay**

**-**

**dockerfile**

\_\_\_\_\_

<

Muh

>

-----

\

^\_\_^

\

(

oo

)

\

\_\_\_\_\_\_\_

(

\_\_

)

\

)

\

/

\

||

----

w |

|| ||

**Building Images from**

**Dockerfile (cont.)**



Much easier! But now we’ve lost the ability to use the fortune command inside the

container as input to

cowsay

.

We

can fix this by providing our own

script. Create

a file

app.sh

with the following

contents and save it in the same directory as the

Dockerfile

.

**Building Images from**

**Dockerfile (cont.)**



#!/

bin/bash

if [ $#

-

eq

0

]

then

/

usr

/games/fortune|/

usr

/games/

cowsay

else

/

usr

/games/

cowsay

"$@"

fi

Set

the file to be executable

with:

~/

cowsay

#

**chmod**

**+**

**x**

**app.sh**

**Building Images from**

**Dockerfile (cont.)**



We next need to

modify the

Dockerfile to add the script into the image and call

it

as argument running the container.

Edit the Dockerfile so that it looks like

:

FROM

ubuntu

RUN

apt

-

get update && apt

-

get install

-

y fortune

-

mod

cowsay

COPY app.sh

/

The

COPY instruction simply copies a file from the host into the image’s filesystem,

the first argument being the file on the host and the second the destination

path, very similar to cp.

**Building Images from**

**Dockerfile (cont.)**



Try building a new image and running

the container starting app.sh without

arguments:

~/

cowsay

#

**docker**

**build**

**-**

**t**

**test/**

**cowsay**

**-**

**dockerfile**

**.**

~/

cowsay

#

**docker**

**run test/**

**cowsay**

**-**

**dockerfile**

**“./app.sh”**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<

Be different: conform. >

------------------------

\

^\_\_^

\

(

oo

)

\

\_\_\_\_\_\_\_

(

\_\_

)

\

)

\

/

\

||

----

w |

|| ||

**Building Images from**

**Dockerfile (cont.)**



And with arguments :

~/

cowsay

#

**docker**

**run test/**

**cowsay**

**-**

**dockerfile**

**"/app.sh"**

**“**

**muh**

**” “**

**muh**

**”**

\_\_\_\_\_\_\_\_\_

<

muh

muh

>

---------

\

^\_\_^

\

(

oo

)

\

\_\_\_\_\_\_\_

(

\_\_

)

\

)

\

/

\

||

----

w |

|| ||

**Building Images from**

**Dockerfile (cont.)**



how do we persist and

**back up**

our data?

For this, we don’t want to use the standard container filesystem

—

instead we need

something that can be easily shared between the container and the host or other

containers. Docker provides this through the concept of volumes.

**Volumes**

are files

or directories

that are directly mounted on the host and not

part of the normal

union file

system. This means they can be shared with other

containers and all changes

will be

made directly to the host filesystem.

**Building Images from Dockerfile (cont.)**



**Volumes:**

There are two ways of declaring a directory as a

volume;

•

using

the VOLUME instruction inside a

Dockerfile

Volume /data

•

specifying the

-

v

flag to docker run.

#

**docker**

**run**

**--**

**name**

**dhost**

**-**

**h**

**dhost**

**-**

**v /data**

**-**

**i**

**-**

**t**

**ubuntu**

**"/bin/bash**

**"**

Both

the following Dockerfile instruction and docker run command have the effect

of creating a volume as /data inside a

container.

**Building Images from Dockerfile (cont.)**

**Building Images from Dockerfile (cont.)**

## Volumes (cont.)

* By default, the directory or file will be mounted on the host inside your Docker installation directory (normally /var/lib/docker/ non persistent).
* It is possible to specify the host directory to use as the mount via the docker run command (this directory is persistent)

## #mkdir -p /vol/dhost

#**docker run --name dhost -h dhost -v /vol/dhost:/data -i -t ubuntu "/bin/bash"** • It isn’t possible to specify a host directory inside a Dockerfile for reasons of portability and security (the file or directory may not exist in other systems, and containers shouldn’t be able to mount sensitive files like etc/passwd without explicit permission).



•

The What and Why of Containers

•

Installing Docker on Linux Running Your First Image

•

The Basic Commands

•

Create a Dockerized Cowsay Application

•

Building Images from Dockerfile

•

**Working with Registries**

•

Docker Fundamentals

•

Dockerfile Instructions

•

Connecting Containers to the World

•

Common Docker Commands

•

Managing

Containers

•

Practical Section

**Agenda**



Now that we’ve created something amazing, how can we

**share**

it with others?

•

When we

first ran the Debian image at the start of the

workshop,

it was

downloaded from

the

**official**

**Docker registry**

—

the

**Docker Hub**

.

•

Similarly

, we can upload our own

images to

the Docker Hub for

**others**

to

download and use.

**Working with Registries**



In order to upload our cowsay image, you will

need:

•

to

**sign up**

for an account with the Docker

Hub;

•

Then,

**tag**

**the image**

into an appropriately named repository and use the

**docker push**

command to upload it to the Docker Hub.

**Working with Registries (cont.)**



Before that, add

a

**MAINTAINER**

instruction to the Dockerfile, which simply sets

the author contact information for the image

:

FROM

ubuntu

MAINTAINER Stefan Koospal <stefan@koospal.de>

**Working with Registries (cont.)**



Now

rebuild

the image and upload it to the Docker Hub. This time, you will need to

use a

**repository name**

that starts with your username on the Docker Hub (in

this

case,

koospal

)

,

followed by / and whatever name you want to give the image.

For example

:

~/

cowsay

#

**docker**

**build**

**-**

**t**

**koospal**

**/**

**cowsay**

**-**

**dockerfile**

**.**

~/

cowsay

#

**docker**

**login**

~/

cowsay

#

**docker**

**push**

**koospal**

**/**

**cowsay**

**-**

**dockerfile**

**Working with Registries (cont.)**



As I didn’t specify a tag after the repository name, it was automatically assigned

the latest tag. To specify a tag, just add it after the repository name with a

colon.

#

**docker**

**build**

**–**

**t**

**koospal**

**/**

**cowsay**

**-**

**dockerfile:stable**

Once

the upload has completed, the world can download your image via the

docker pull

command:

#

**docker**

**pull**

**koospal**

**/**

**cowsay**

**-**

**dockerfile**

**Working with Registries (cont.)**



#

**docker**

**pull**

**openjdk**

#

**mkdir**

**openjdk**

**-**

**jshell**

**; cd**

**openjdk**

**-**

**jshell**

Create

Dockerfile

for

jshell

https://github.com/docker

-

library/openjdk/blob/a893fe3cd82757e7bccc0948c88bfee09bd916c3/9

-

jdk/Dockerfile

#

**docker**

**build**

**-**

**t test/**

**openjdk**

**-**

**jshell**

**.**

#

**docker**

**run**

**-**

**i**

**-**

**t test/**

**openjdk**

**-**

**jshell**

Feb 08, 2018 1:34:22 PM java.util.prefs.FileSystemPreferences$1 run

INFO: Created user preferences directory.

| Welcome to

JShell

--

Version 9.0.1

| For an introduction type: /help

intro

jshell

>

**Pull**

**openjdk**

**from Docker**

**-**

**Registry**



•

The What and Why of Containers

•

Installing Docker on Linux Running Your First Image

•

The Basic Commands

•

Create a Dockerized Cowsay Application

•

Building Images from Dockerfile

•

Working with Registries

•

**Docker Fundamentals**

•

Dockerfile Instructions

•

Connecting Containers to the World

•

Common Docker Commands

•

Managing

Containers

•

Practical Section

**Agenda**



The

major components of a Docker

installation:

•

**Docker daemon**

, which is responsible for creating, running, and monitoring

containers, as well as building and storing images,

and launched

by running

docker

daemon

, which is normally taken care of by the host OS

.

•

**Docker**

**client**

is

used to talk to the Docker daemon via HTTP. By default, this

happens over a Unix domain socket, but it can also use a TCP socket to enable

remote clients or a file descriptor for

system

-

managed

sockets.

•

**Docker registries**

store and distribute images.

**The Docker Architecture**



**The Docker Architecture (cont.)**

Docker Host

Registry

Docker

Daemon

C

o

n

C

o

n

Docker

Client



•

Each

instruction in

a Dockerfile results in a new

**image layer**

, which can also

be used to start a

container. The

new layer is created by starting a container

using the image of the

previous layer

, executing the Dockerfile instruction and

saving a new image

.

•

When a Dockerfile instruction successfully completes, the intermediate

container will be deleted. Since each instruction results in an static image

—

essentially just a filesystem and some metadata

—

all running processes in the

instruction will be stopped

.

•

If you want a service or process to start with the container, it must be launched

from an

**ENTRYPOINT**

or

**CMD**

instruction.

**Image Layer**



You can see the full set of layers that make up an image by running the docker

history

command.

One example is:

~/

cowsay

#

**docker**

**history**

**koospal**

**/**

**cowsay**

**-**

**dockerfile**

IMAGE CREATED

CREATED

BY SIZE COMMENT

49

e038393108 25 minutes ago /bin/

sh

-

c #(

nop

COPY file:c22006eaeae75fd8… 103B

)

5

ed11d75f720 25 minutes ago /bin/

sh

-

c apt

-

get update && apt

-

get install… 85.4MB

534160f2

aa5d 25 minutes ago /bin/

sh

-

c #(

nop

)

MAINTAINER Stefan Koospal… 0B

0458a4468

cbc 2 weeks ago /bin/

sh

-

c #(

nop

)

CMD ["/bin/bash"] 0B

**Image Layer (cont.)**



When creating your own images, you will need to decide which base image to start

from

:

•

The best

-

case scenario is

just to use

an

**existing**

**image**

and mount your

configuration files and/or data into it. This is

to

be the case for common

application software, such as databases and web servers, where there are

official images available

.

•

Sometimes you really just need a small but

**complete Linux distro**

. T

he

alpine image, which is only just over 5 MB in size but still has an extensive

packager manager for easily installing applications and tools.

The Debian

images are second option.

**Base Images**



•

The What and Why of Containers

•

Installing Docker on Linux Running Your First Image

•

The Basic Commands

•

Create a Dockerized Cowsay Application

•

Building Images from Dockerfile

•

Working with Registries

•

Docker Fundamentals

•

**Dockerfile Instructions**

•

Connecting Containers to the World

•

Common Docker Commands

•

Managing

Containers

•

Practical Section

**Agenda**

# Dockerfile Instructions

## ADD

Copies files from the build context or remote URLs into the image. If an archive file is added from a local path, it will automatically be unpacked. As the range of functionality covered by ADD is quite large, it’s generally best to prefer the simpler COPY command for copying files and directories in the build context and RUN instructions with curl or wget to download remote resources.

## COPY

Used to copy files from the build context into the image. It has two forms, COPY src dest\_ and COPY ["src", "dest"], both of which copy the file or directory at src in the build context to dest inside the container. The JSON array format is required if the paths have spaces in them. Wildcards can be used to specify multiple files or directories. Note that you cannot specify src paths outside the build context (e.g., ../another\_dir/myfile will not work).

# Dockerfile Instructions (cont.)

## CMD

Runs the given instruction when the container is started. If an ENTRYPOINT has been defined, the instruction will be interpreted as an argument to the ENTRY POINT (in this case, make sure you use the exec format). The CMD instruction is overridden by any arguments to docker run after the image name. Only the last CMD instruction will have an effect, and any previous CMD instructions will be overridden (including those in base images).

## ENTRYPOINT

Sets an executable (and default arguments) to be run when the container starts. Any CMD instructions or arguments to docker run after the image name will be passed as parameters to the executable. ENTRYPOINT instructions are often used to provide “starter” scripts that initialize variables and services before interpreting any given arguments.



CMD

and

ENTRYPOINT commands

allow us to

**set the default command**

to run

in a container

.

**Defining a default**

**command**

When people run our container, we want to greet them with a nice hello message,

and using

a custom

font. For

that, we will execute

:

#

**figlet**

**-**

**f script**

**hello**

•

-

f script tells

f

iglet

to use a fancy font.

•

hello

is the message that we want it to display.

**Dockerfile**

**Instructions (cont.)**



**Adding CMD to our Dockerfile**

•

CMD defines a default command to run when none is

given.

•

It

can appear at any point in the

file.

•

Each

CMD will replace and override the previous

one.

•

As

a result, while you can have multiple CMD lines, it is useless.

**Dockerfile**

**Instructions (cont.)**

FROM

ubuntu

RUN

apt

-

get

update &&

apt

-

get

install

-

y

figlet

CMD

figlet

-

f

script

hello



**Build and test our**

**image**

Let's build it

:

~/

figlet

#

**docker**

**build**

**-**

**t**

**figlet**

**.**

And

run it:

~/

figlet

#

**docker**

**run**

**-**

**t**

**figlet**

\_ \_ \_

| | | | | |

| | \_ | | | | \_\_

|/

\

|/ |/ |/ /

\

\_

| |\_/|\_\_/|\_\_/|\_\_/

\

\_\_/

**Dockerfile**

**Instructions (cont.)**



**Overriding**

**CMD**

If

we want to get a shell into our container (instead of running

figlet

)

, we just have

to specify

a different program to run

:

~/

figlet

#

**docker**

**run**

**-**

**h**

**figlet**

**-**

**it**

**figlet**

**"/bin/bash"**

root@figlet

:

/#

•

We

specified

bash.

•

It

replaced the value of CMD.

**Dockerfile**

**Instructions (cont.)**



**Using**

**ENTRYPOINT**

We want to be able to specify a different message on the command line, while

retaining

figlet

and some default parameters.

In other words, we would like to be able to do this:

#

**docker**

**run**

**figlet**

**salut**

\_

| |

, \_\_, | | \_|\_

/

\

\_/ | |/ | | |

\

/

\

\_/|\_/|\_\_/

\

\_/|\_/|\_/

**Dockerfile**

**Instructions (cont.)**



**Using**

**CMD and ENTRYPOINT together**

What if we want to define a default URL for our container?

Then

we will use ENTRYPOINT and CMD together.

•

ENTRYPOINT

will define the base command for our

container.

•

CMD

will define the default parameter(s) for this command.

**Dockerfile**

**Instructions (cont.)**



Using the exec format (

Json

)

FROM

ubuntu

RUN apt

-

get update && apt

-

get install

-

y

figlet

ENTRYPOINT ["

figlet

","

-

f","script

"]

CMD

hello

•

ENTRYPOINT defines a base command (and its parameters) for the

container.

•

If we don't specify extra command

-

line arguments when starting the

container, the value of CMD is appended.

•

Otherwise, our extra command

-

line arguments are used instead of

CMD.

**Dockerfile**

**Instructions (cont.)**



**Build**

**and test our image**

Let's build it

:

~/

figlet

#

**docker**

**build**

**-**

**t**

**figlet**

**.**

And

run it

:

#

**docker**

**run**

**figlet**

**salut**

\_

| |

, \_\_, | | \_|\_

/

\

\_/ | |/ | | |

\

/

\

\_/|\_/|\_\_/

\

\_/|\_/|\_/

**Dockerfile**

**Instructions (cont.)**



**Overriding ENTRYPOINT**

What if we want to run a shell in our container?

We cannot just do docker run

figlet

bash because that would just tell

figlet

to

display the word "bash."

We use the

--

entrypoint

parameter:

**Dockerfile**

**Instructions (cont.)**

#

**docker**

**run**

**-**

**it**

**-**

**h**

**figlet**

**--**

**entrypoint**

**"/bin/bash"**

**figlet**



• CMD defines a default command to run when none is given.

• It can appear at any point in the file.

• Each CMD will replace and override the previous one.

• As a result, while you can have multiple CMD lines, it is useless.

**Dockerfile**

**Instructions (cont.)**



**ENV**

•

Sets environment variables inside the image. These can be referred to in

subsequent instructions. For example

:

FROM

ubuntu

ENV MYVERSION 2.7

RUN apt

-

get update && apt

-

get install

-

y

figlet

RUN apt

-

get install

-

y python${MYVERSION}

-

minimal

•

The variables will also be available inside the image

.

#

**docker**

**run**

**-**

**it**

**-**

**h**

**figlet**

**--**

**entrypoint**

**"/bin/bash"**

**figlet**

root@figlet

:

/# echo $MYVERSION

1.1

**Dockerfile**

**Instructions (cont.)**

# Dockerfile Instructions (cont.)

## EXPOSE

Indicates to Docker that the container will have a process listening on the given port or ports. This information is used by Docker when linking containers (see “Linking Containers”) or publishing ports by supplying the -P argument to docker run; by itself the EXPOSE instruction will not affect networking.

## FROM

Sets the base image for the Dockerfile; subsequent instructions build on top of this image. The base image is specified as IMAGE:TAG (e.g., debian:wheezy). If the tag is omitted, it is assumed to be latest, but I strongly recommend you always set the tag to a specific version to avoid surprises. Must be the first instruction in a Dockerfile.



**MAINTAINER**

Sets the “Author” metadata on the image to the given string. You can retrieve this

with docker inspect

-

f {{.Author}} IMAGE. Normally used to set the name and

contact details of the maintainer of the image

.

**ONBUILD**

Specifies an instruction to be executed later, when the image is used as the base

layer to another image. This can be useful for processing data that will be added in

a child image (e.g., the instruction may copy in code from a chosen directory and

run a build script on the data).

**RUN**

Runs the given instruction inside the container and commits the result.

**Dockerfile**

**Instructions (cont.)**

# Dockerfile Instructions (cont.)

## USER

Sets the user (by name or UID) to use in any subsequent RUN, CMD, or ENTRYPOINT instructions. Note that UIDs are the same between the host and container, but usernames may be assigned to different UIDs, which can make things tricky when setting permissions.

## VOLUME

Declares the specified file or directory to be a volume. If the file or directory already exists in the image, it will copied into the volume when the container is started. If multiple arguments are given, they are interpreted as multiple volume statements. You cannot specify the host directory for a volume inside a Dockerfile for portability and security reasons. For more information, see “Managing Data with Volumes and Data Containers”.



**WORKDIR**

Sets the working directory for any subsequent RUN, CMD, ENTRYPOINT, ADD, or

COPY instructions. Can be used multiple times. Relative paths may be used and

are resolved relative to the previous WORKDIR

.

**Dockerfile**

**Instructions (cont.)**



•

The What and Why of Containers

•

Installing Docker on Linux Running Your First Image

•

The Basic Commands

•

Create a Dockerized Cowsay Application

•

Building Images from Dockerfile

•

Working with Registries

•

Docker Fundamentals

•

Dockerfile Instructions

•

**Connecting Containers to the World**

•

Common Docker Commands

•

Managing

Containers

•

Practical Section

**Agenda**



Say you’re running a web server inside a container. How do you provide the

outside world with access? The answer is to “publish” ports with the

-

p or

-

P

commands. This command forwards ports on the host to the container. For

example

:

#

**docker**

**pull**

**nginx**

…

#

**docker**

**run**

**-**

**h www**

**--**

**name www**

**-**

**d**

**-**

**p 8000:80**

**nginx**

…

#

**docker**

**ps**

**CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS**

**NAMES**

**4**

**fd7edf272bb**

**nginx**

**"**

**nginx**

**-**

**g 'daemon of…" About a minute ago Up About a minute 0.0.0.0:8000**

**-**

**>80**

**/**

**tcp**

**www**

#

**curl**

**localhost:8000**

<!

DOCTYPE html

>

<

h1>Welcome to

nginx

!<

/h

1>

**Connecting Containers to the World**

# Connecting Containers to the World (cont.)

The -p 8000:80 argument has told Docker to forward port 8000 on the host to port 80 in the container. Alternatively, the -P argument can be used to tell Docker to automatically select a free port to forward to on the host. For example:

# **ID=$(docker run -h www --name www -d -P nginx)** # **docker port $ID 80**

0.0.0.0:32768

## # curl localhost:32768

<!DOCTYPE html>

…

<h1>Welcome to nginx!</h1>

The primary advantage of the -P command is that you are no longer responsible for keeping track of allocated ports, which becomes important if you have several containers publishing ports. In these cases you can use the docker port command to discover the port allocated by Docker.



•

The What and Why of Containers

•

Installing Docker on Linux Running Your First Image

•

The Basic Commands

•

Create a Dockerized Cowsay Application

•

Building Images from Dockerfile

•

Working with Registries

•

Docker Fundamentals

•

Dockerfile Instructions

•

Connecting Containers to the World

•

**Common Docker Commands**

•

Managing

Containers

•

Practical Section

**Agenda**



It is

the

most complex command and supports a large list of potential arguments.

The arguments allow users to configure how the image is run, override Dockerfile

settings,

configure

networking, and set privileges and resources for the container

.

**-**

**a,**

**--**

**attach**

Attaches the given stream (STDOUT, etc.) to the terminal. If unspecified, both

STDOUT

and

STDERR

are attached. If unspecified and the container is started in

interactive mode (

-

i

)

,

STDIN

is also attached.

Incompatible with

-

d

**The run Command**



**-**

**d,**

**--**

**detach**

Runs the container in “detached” mode. The command will run the container in

the background and return the container ID

.

**-**

**i**

**,**

**--**

**interactive**

Keeps

STDIN

open (even when it’s not attached). Generally used with

-

t to start

an interactive container session. For example

:

#

**docker**

**run**

**-**

**h**

**ubuntu**

**-**

**i**

**-**

**t**

**ubuntu**

**"/bin/bash"**

root@ubuntu

:

/# echo "hello world"

hello world

**The run**

**Command (cont.)**



**--**

**restart**

Configures when Docker will attempt to restart an exited container. The argument

no will never attempt to restart a container, and always will always try to

restart, regardless of exit status. The on

-

failure argument will attempt to restart

containers that exit with a nonzero status and can take an optional argument

specifying the number of times to attempt to restart before giving up (if not

specified, it will retry forever). For example, docker run

--

restart

onfailure

:

10

postgres

will launch the

postgres

container and attempt to restart it

10

times if it exits with a nonzero code.

**The run**

**Command (cont.)**



**--**

**rm**

Automatically removes the container when it exits. Cannot be used with

-

d.

**-**

**t,**

**--**

**tty**

Allocates a pseudo

-

TTY. Normally used with

-

i

to start an interactive container.

The following options allow setting of container names and variables:

**The run**

**Command (cont.)**



**-**

**e,**

**--**

**env**

Sets environment variables inside the container. For example

:

#

**docker**

**run**

**-**

**h**

**ubuntu**

**-**

**e var1="hello"**

**-**

**t**

**ubuntu**

**env**

PATH=/

usr

/local/

sbin

:

/

usr

/local/bin:/

usr

/

sbin

:

/

usr

/bin:/

sbin

:

/bin

HOSTNAME=

ubuntu

TERM=

xterm

var1=hello

HOME=/

root

Also note the

--

env

-

file option for passing variables in via a file.

**The run**

**Command (cont.)**



**-**

**h,**

**--**

**hostname**

Sets the container’s

Unix

host name to NAME. For example:

$

**docker run**

**-**

**h "**

**myhost**

**“**

**ubuntu**

**hostname**

myhost

**--**

**name**

**NAME**

Assigns the name

NAME

to the container. The name can then be used to address

the container in other Docker commands.

**The run**

**Command (cont.)**



**-**

**v,**

**--**

**volume**

There are two forms of the argument to set up a volume (a file or directory

within a container that is part of the native host filesystem, not the container’s

union file system). The first form only specifies the directory within the container

and will bind to a host directory of Docker’s choosing. The second form

specifies the host directory to bind to

.

**--**

**volumes**

**-**

**from**

Mounts volumes from the specified container. Often used in association with

data containers

**The run**

**Command (cont.)**



**--**

**expose**

Equivalent of Dockerfile EXPOSE instruction. Identifies the port or port range as

being used in the container but does not open the port. Only really makes sense

in association with

-

P and when linking containers.

**--**

**link**

Sets up a private network interface to the specified container.

**-**

**p,**

**--**

**publish**

“Publishes” a port on the container, making it accessible from the host. If the host

port is not defined, a random high

-

numbered port will chosen, which can be

discovered by using the

**docker port**

command. The host interface on which to

expose the port may also be specified.

**The run**

**Command (cont.)**



**--**

**expose**

Equivalent of Dockerfile EXPOSE instruction. Identifies the port or port range as

being used in the container but does not open the port. Only really makes sense

in association with

-

P and when linking containers.

**--**

**link**

Sets up a private network interface to the specified container.

**-**

**p,**

**--**

**publish**

“Publishes” a port on the container, making it accessible from the host. If the host

port is not defined, a random high

-

numbered port will chosen, which can be

discovered by using the

**docker port**

command. The host interface on which to

expose the port may also be specified.

**The run**

**Command (cont.)**



**-**

**P,**

**--**

**publish**

**-**

**all**

Publish all exposed ports on the container to the host. A random high

-

numbered

port will be chosen for each exposed port. The docker port command can be

used to see the mapping

.

The following options directly override Dockerfile settings:

**--**

**entrypoint**

Sets the

entrypoint

for the container to the given argument, overriding any ENTRY

POINT instruction in the Dockerfile

.

**The run**

**Command (cont.)**



**-**

**u,**

**--**

**user**

Sets the user that commands are run under. May be specified as a username or

UID. Overrides USER instruction in Dockerfile.

**-**

**w,**

**--**

**workdir**

Sets the working directory in the container to the provided path. Overrides any

value in the Dockerfile.

**The run**

**Command (cont.)**



•

The What and Why of Containers

•

Installing Docker on Linux Running Your First Image

•

The Basic Commands

•

Create a Dockerized Cowsay Application

•

Building Images from Dockerfile

•

Working with Registries

•

Docker Fundamentals

•

Dockerfile Instructions

•

Connecting Containers to the World

•

Common Docker Commands

•

**Managing**

**Containers**

•

Practical Section

**Agenda**



**docker attach [OPTIONS] CONTAINER**

The attach command allows the user to view or interact with the main process

inside the container. For example

:

#

**ID**

**=**

**$**

**(**

**docker**

**run**

**-**

**d**

**ubuntu**

**sh**

**-**

**c "while true; do echo**

**Tick;sleep**

**1**

**;done**

**")**

#

**docker attach $ID**

tick

t

ick

...

Note

that using CTRL

-

C to quit will end the process and cause the container to

exit

.

**Managing Containers (cont.)**



**docker create**

Creates a container from an image but does not start it. Takes most of the same

arguments as docker run. To start the container, use docker start.

**docker**

**cp**

Copies files and directories between a container and the host.

**docker**

**exec**

Runs a command inside a container. Can be used to perform maintenance tasks

or as a replacement for

ssh

to log in to a container.

**Managing Containers (cont.)**



**docker**

**exec**

Runs a command inside a container. Can be used to perform maintenance tasks

or as a replacement for

ssh

to log in to a container

. For example:

#

**ID**

**=**

**$**

**(**

**docker**

**run**

**-**

**d**

**ubuntu**

**sh**

**-**

**c "while true; do sleep 1;done**

**")**

#

**docker**

**exec $ID echo "**

**Hello“**

Hello

#

**docker**

**exec $ID /bin/bash**

root@e299debda797

:

/# ls

bin dev home lib64

mnt

proc run

srv

tmp

var

boot

etc

lib media opt root

sbin

sys

usr

root@e299debda797:/# exit

**Managing Containers (cont.)**



**docker**

**kill**

Sends a signal to the main process (PID 1) in a container. By default, sends a

SIGKILL, which will cause the container to exit immediately. Alternatively, the

signal can be specified with the

-

s argument. The container ID is returned.

For example

:

#

**ID**

**=**

**$**

**(**

**docker**

**run**

**-**

**d**

**ubuntu**

**bash**

**-**

**c "trap 'echo got**

**-**

**sig' 2;while**

**true;do**

**sleep 1;done")**

#

**docker**

**kill**

**-**

**s 2 $ID**

ca8bf50b4b303fa72c9494503e832a977c6a7121ba08cb2ffe2f144fcaed4ba4

#

**docker**

**logs $ID**

got

-

sig

#

**docker**

**kill**

**$ID**

**Managing Containers (cont.)**

# Managing Containers (cont.)

## docker pause

Suspends all processes inside the given container. The processes do not receive any signal that they are being suspended and consequently cannot shut down or clean up. The processes can be restarted with docker unpause. docker pause uses the Linux cgroups freezer functionality internally. This command contrasts with docker stop, which stops the processes and sends signals observable by the processes.

## docker restart

Restarts one or more containers. Roughly equivalent to calling docker stop followed by docker start on the containers. Takes an optional argument -t that specifies the amount of time to wait for the container to shut down before it is killed with a SIGTERM.



**docker**

**rm**

Removes one or more containers. Returns the names or IDs of

successfully

deleted containers. By default, docker rm will not remove any volumes. The

-

f

argument can be used to remove running containers, and the

-

v argument will

remove volumes created by the container (as long as they aren’t bind mounted or

in use by another container).

For example, to delete all stopped containers

:

#

**docker**

**rm**

**-**

**v $(**

**docker**

**ps**

**-**

**aq**

**)**

121

bedfbc

193

fd6c883a46e1

f1e4edf9055d

**Managing Containers (cont.)**



**docker**

**start**

Starts a stopped container (or containers). Can be used to restart a container that

has exited or to start a container that has been created with docker create but

never launched

.

**docker stop**

Stops (but does not remove) one or more containers. After calling docker stop

on a container, it will transition to the “exited” state. Takes an optional argument

-

t which specifies the amount of time to wait for the container to shutdown

before it is killed with a SIGTERM

.

**docker**

**unpause**

Restarts a container previously paused with docker pause.

**Managing Containers (cont.)**



**docker info**

Prints various information on the Docker system and host.

**docker**

**help**

Prints usage and help information for the given subcommand. Identical to running

a command with the

--

help flag.

**docker**

**version**

Prints Docker version information for client and server as well as the version of

Go used in compilation.

**Docker Info**



**docker info**

Prints various information on the Docker system and host.

**docker**

**help**

Prints usage and help information for the given subcommand. Identical to running

a command with the

--

help flag.

**docker**

**version**

Prints Docker version information for client and server as well as the version of

Go used in compilation.

**Docker Info (cont.)**



**docker**

**diff**

Shows changes made to the containers filesystem compared to the image it was

launched from. For example

:

#

**ID=$(**

**docker**

**run**

**-**

**d**

**ubuntu**

**touch /NEWFILE)**

#

**docker**

**diff $ID**

A /

NEWFILE

**docker events**

Prints real

-

time events from the daemon. Use CTRL

-

C to quit.

**Container Info**



**docker**

**inspect**

Provides detailed information on given containers or images. The information

includes most configuration information and covers network settings and volume

mappings. The command can take one argument,

-

f, which is used to supply

a Go template that can be used to format

and

filter the output

.

**docker logs**

Outputs the “logs” for a container. This is simply everything that has been written

to STDERR or STDOUT inside the container.

**Container Info (cont.)**



**docker**

**port**

Lists the exposed port mappings for the given container. Can optionally be given

the internal container port and protocol to look up. Often used after docker run

-

P <image> to discover the assigned ports.

For example

:

#

**ID=$(**

**docker**

**run**

**-**

**h www**

**--**

**name www**

**-**

**d**

**-**

**P**

**nginx**

**)**

#

**docker**

**port $ID**

80

/

tcp

-

> 0.0.0.0:32769

#

**docker**

**port $ID 80**

0.0.0.0:32769

#

**docker**

**port $ID 80/**

**tcp**

0.0.0.0:32769

**Container Info (cont.)**



**docker**

**ps**

Provides high

-

level information on current containers, such as the name, ID, and

status. Takes a lot of different arguments, notably

-

a for getting all containers,

not just running ones. Also note the

-

q argument, which only returns the container

IDs and is very useful as input to other commands such as docker rm.

**Container Info (cont.)**



**docker**

**top**

Provides information on the running processes inside a given container. In effect,

this command runs the UNIX ps utility on the host and filters for processes in

the given container.

For

example

:

#

**docker**

**top $ID**

UID PID PPID C STIME TTY TIME

CMD

root 5091 5077 0 14:31 ? 00:00

:00

nginx

master process

:

nginx

-

g daemon off;

systemd

5118 5091 0 14:31 ? 00:00:00

+

nginx

:

worker process

#

**ps**

**-**

**aux |grep**

**5091**

root

5091

0.0 0.0 32552 5172 ?

Ss

14:31

0:00

nginx

:

master process

nginx

-

g daemon off

;

**Container Info (cont.)**



**docker**

**build**

Builds an image from a Dockerfile.

**docker**

**commit**

Creates an image from the specified container.

By

default, containers are paused

prior to commit, but this can

be turned

off with the

--

pause=false argument. Takes

-

a and

-

m arguments for

setting metadata. For

example

:

#

**ID**

**(**

**=**

**$**

**docker**

**run**

**-**

**d**

**ubuntu**

**touch /NEWFILE)**

#

**docker**

**commit**

**-**

**a "Stefan Koospal"**

**-**

**m "Comment" $ID**

**newfile:test**

sha256:c9c7833762d4bb8fbaa29c8c82c60230029311c4784b814597c9eb0b822eeb1a

#

**docker**

**images**

REPOSITORY TAG IMAGE ID CREATED SIZE

newfile

test c9c7833762d4 About a minute ago 112MB

**Docker Images**



**docker**

**history**

Outputs information on each of the layers in an image.

**docker**

**images**

Provides a list of local images, including information such as repository name,

tag name, and size.

Takes

several

arguments; in

particular, note

-

q, which only

returns

the image IDs and is useful

as input

to other commands such as docker

rmi

.

#

**docker**

**images**

test/

figlet

latest 571b6e1056e2 2 hours ago 153MB

test/

openjdk

-

jshell

latest

fbe3a756f5c4 6 days ago 910MB

test/

openjdk

latest

fbe3a756f5c4 6 days ago 910MB

test/java

latest fbe3a756f5c4 6 days ago 910MB

test/

cowsay

-

dockerfile

latest

ac747299997b 6 days ago 197MB

**Docker Images (cont.)**



**docker**

**import**

Creates an image from an archive file containing a filesystem, such as that

created by

docker export. The archive may be identified by a file path or URL or

streamed through STDIN (by using the

-

flag).

**docker load**

Loads a repository from a tar archive passed via STDIN. The repository may

contain several

images and tags.

**Docker Images (cont.)**

# Docker Images (cont.)

## docker rmi

Deletes the given image or images. Images are specified by ID or repository and tag name. If a repository name is supplied but no tag name, the tag is assumed to be latest. To delete images that exist in multiple repositories, specify that image by ID and use the -f argument. You will need to run this once per repository.

## docker save

Saves the named images or repositories to a tar archive, which is streamed to STDOUT (use -o to write to a file). Images can be specified by ID or as repository:tag. If only a repository name is given, all images in that repository will be saved to the archive, not just the latest tag.

# Docker Images (cont.)

## docker rmi

Deletes the given image or images. Images are specified by ID or repository and tag name. If a repository name is supplied but no tag name, the tag is assumed to be latest. To delete images that exist in multiple repositories, specify that image by ID and use the -f argument. You will need to run this once per repository.

## docker save

Saves the named images or repositories to a tar archive, which is streamed to STDOUT (use -o to write to a file). Images can be specified by ID or as repository:tag. If only a repository name is given, all images in that repository will be saved to the archive, not just the latest tag.



**docker**

**login**

Register with, or log in to, the given registry server. If no server is specified, it is

assumed to be the Docker Hub. The process will interactively ask for details if

required, or they can be supplied as arguments.

**docker logout**

Logs out from a Docker registry. If no server is specified, it is assumed to be the

Docker Hub.

**Using the Registry**



**docker**

**pull**

Downloads the given image from a registry.

Use

the

-

a argument to download

all images from a repository.

**docker**

**push**

Pushes an image or repository to the registry. If no tag is given, this will push all

images in the repository to the registry, not just the one marked latest.

**docker**

**search**

Prints a list of public repositories on the Docker Hub matching the search term.

Limits results to 25 repositories.

**Using the**

**Registry (cont.)**



•

The What and Why of Containers

•

Installing Docker on Linux Running Your First Image

•

The Basic Commands

•

Create a Dockerized Cowsay Application

•

Building Images from Dockerfile

•

Working with Registries

•

Docker Fundamentals

•

Dockerfile Instructions

•

Connecting Containers to the World

•

Common Docker Commands

•

Managing Containers

•

**Practical Section**

**Agenda**



**Set**

**up**

**a virtual machine with**

**docker**

**to host a web page**