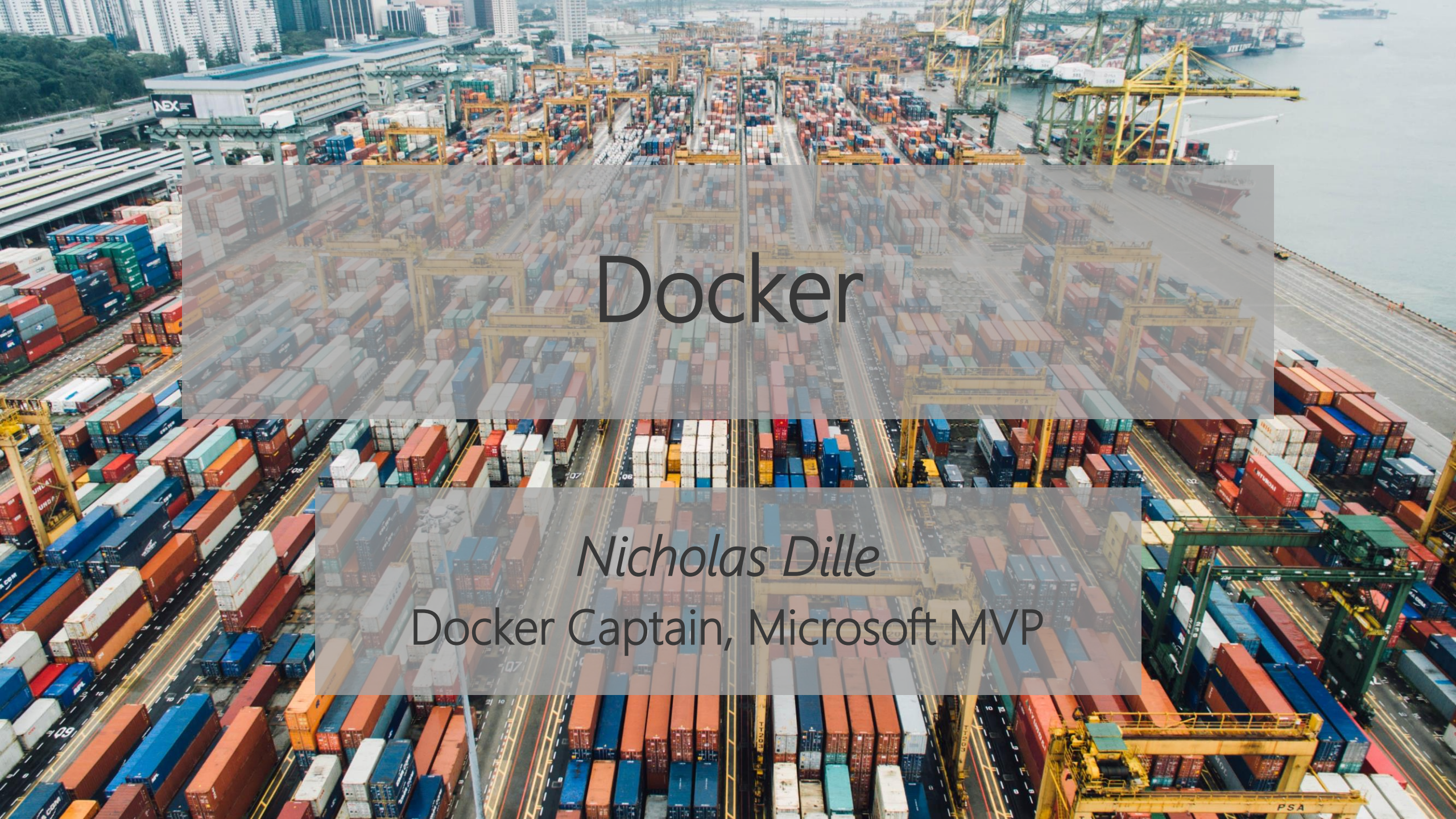


Docker

Nicholas Dille

Docker Captain, Microsoft MVP

Docker kann ich…
…und was nun?

Nicholas Dille

Docker Captain, Microsoft MVP

[Container Conf] 2018

Neues aus dem Docker-Universum

Nicholas Dille

Docker Captain, Microsoft MVP

[Container Conf] 2018

Qualitätssicherung von Container-Images

Nicholas Dille

Docker Captain, Microsoft MVP

[Container Conf] 2018

Is Docker making us more secure or less

Nicholas Dille

Docker Captain, Microsoft MVP

DEVSMEETUP

Nicholas Dille

Husband, Father, Geek, Speaker, Author

Proudly working at Haufe Group

Microsoft MVP since 2010

Docker Captain since 2017

dille.name

@NicholasDille

„Regeln“

Lieber Du als Sie

Keine Konsumveranstaltung

Stellt Fragen

Andere Meinungen / Erfahrungen sind wichtig

Seid neugierig und probiert aus

Erinnert mich an Pausen ;-)

Toiletten: links rum, Treppe runter

Agenda: Docker kann ich… und was nun?

Docker 101	Advanced Concepts	Remoting	Docker-in-Docker	Myths and (Anti)Patterns	Security	Windows Containers
Planning for Containers	Continuous Integration	Continuous Delivery	Testing	Monitoring	Running in Production	Ecosystem

Build your own Agenda

Docker 101	Advanced Concepts	Remoting	Docker-in-Docker	Myths and (Anti)Patterns	Security	Windows Containers
Planning for Containers	Continuous Integration	Continuous Delivery	Testing	Monitoring	Running in Production	Ecosystem

Build your own Agenda

	Duration	Introduction	Advanced	Automation	DIY
Docker 101	90m	X			
Advanced Concepts	60m	X	X		
Remoting	30m	X	X		
Docker-in-Docker	30m		X		
Myths and (Anti)Patterns	60m	X	X		
Security	60m		X		
Windows Containers	30m				
Planning for Containers	30m			X	
Continuous Integration	30m			X	
Continuous Delivery	60m			X	
Testing	30m			X	
Monitoring	45m				
Running in Production	60m			X	
Ecosystem	45m				

Play with Docker

Cloud-Service zum Erlernen von Docker

Kostenfreie Docker-Hosts für vier Stunden

Jeder Host nutzt Docker-in-Docker

https://labs.play-with-docker.com/

Tooling

Anmeldung per SSH (ssh -p 1022 10–0–1–3–48a594c4@host1.labs.play-with-docker.com)

Treiber für docker-machine (https://github.com/play-with-docker/docker-machine-driver-pwd)

Kudos an die Autoren

Docker Captains Marcos Nils und Jonathan Leibiusky

Prepare your environment

PWD

Go to https://play-with-docker.com

Create account and login

Create instance

Get example code

Clone GitHub repository

```
git clone https://github.com/nicholasdille/docker-ci-cd-examples.git
```

Prepare your environment

Create SSH key

Linux: `ssh-keygen -f ~/.ssh/id_rsa_lab`

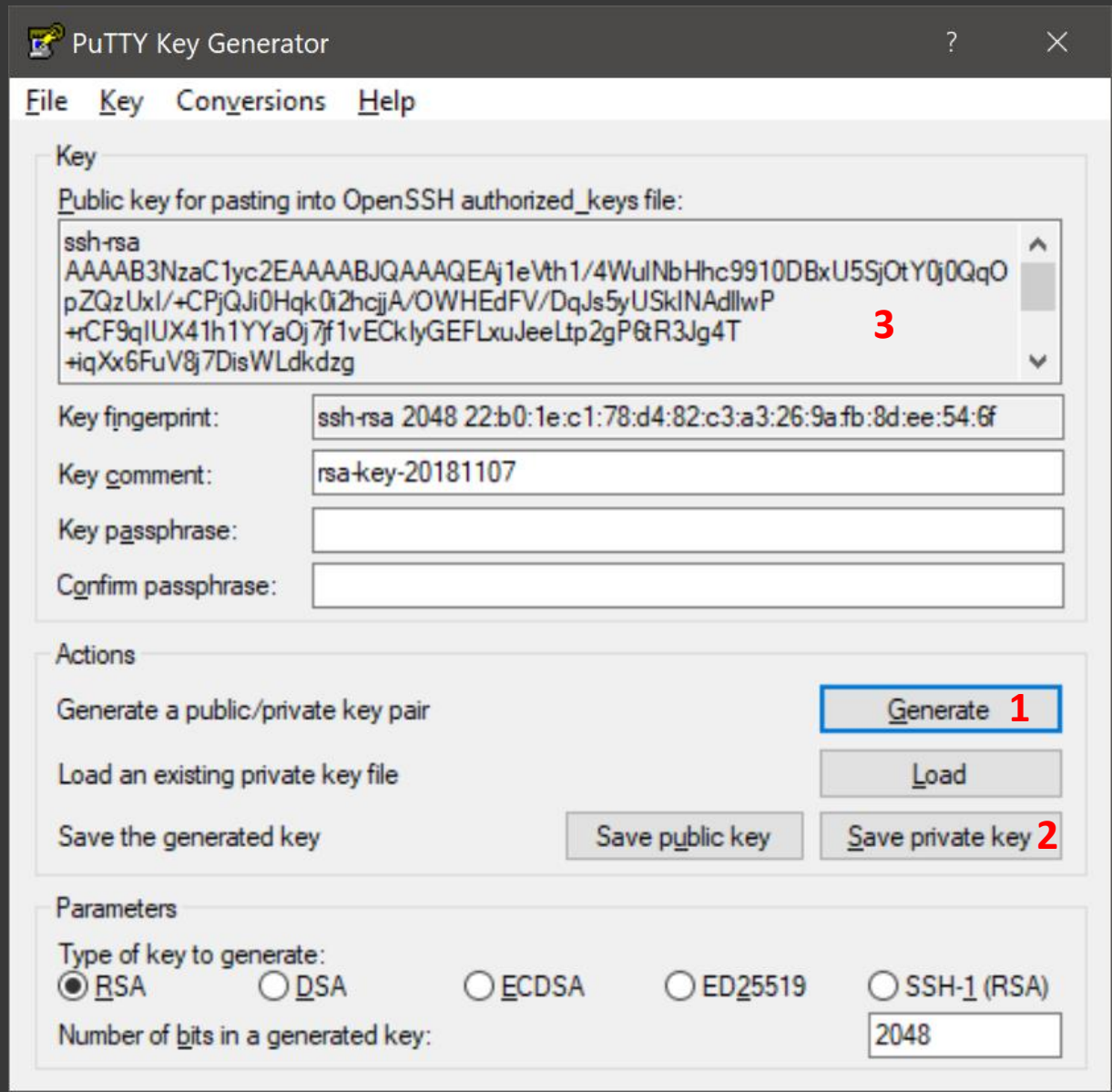
Windows: Use `puttygen`

- Click (1) and wiggle mouse
- Save private key (2) to file
- Send public key (3) to `containerconf2018@dille.name`

Wait for response with…

…hostname

…username



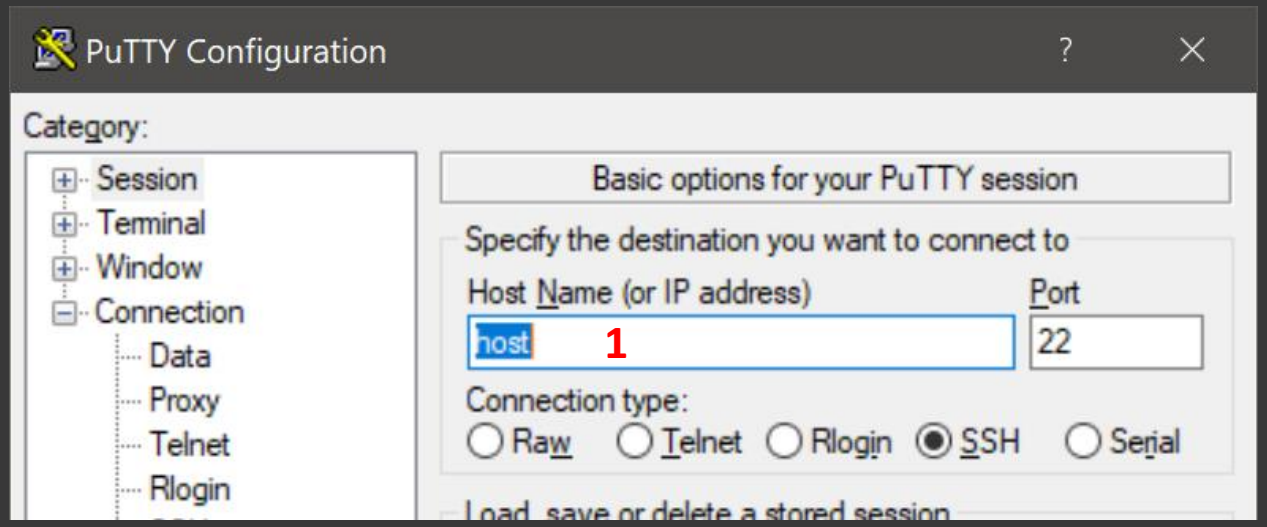
Prepare your environment

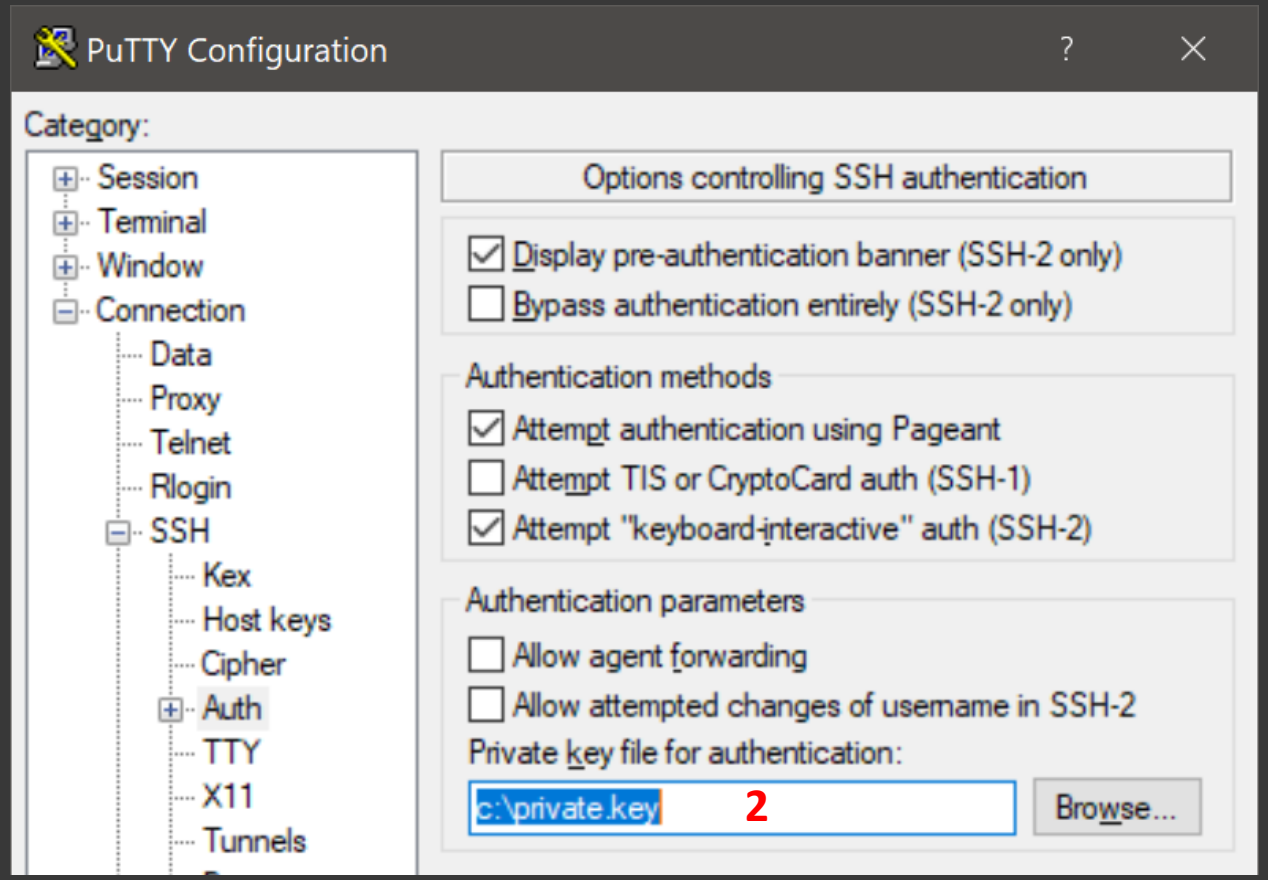
Test connection

Linux: `ssh -i ~/.ssh/id_rsa_lab user@host`

Windows: Using `putty`

- Enter hostname
- Select private key





Prepare your environment

Please check your desk

Piece of paper with…

Hostname

Username

Password

Please connect using SSH

Username: lab

Password: ‘Tismeilap

Demos

https://github.com/nicholasdille/docker-lab/tree/ContainerConf2018

Docker 101

From zero to ~hero

Names

The Good

Docker is a company

Docker is a container management tool

The Bad

To dockerize

The Ugly

I have an application in a docker

Was macht Docker so besonders?

Am Anfang waren logische Partitionen (LPARs)

Veröffentlicht 1972 von unseren Eltern

Dann kamen Linux Containers (LXC)

Veröffentlicht 2008 von unseren älteren Geschwistern

Schnittstelle zu Control Groups und Namespaces

Zuletzt erblickt Docker das Licht der Welt

Gegründet 2013 von Solomon Hykes

Revolution der Container-Verwaltung durch Automatisierung

Concepts

Process isolation

Containerized processes cannot see the host OS

Containerized processes cannot see into other containers

Kernel is responsible

Isolation from other containers and the host

Resource management

Containers are immutable

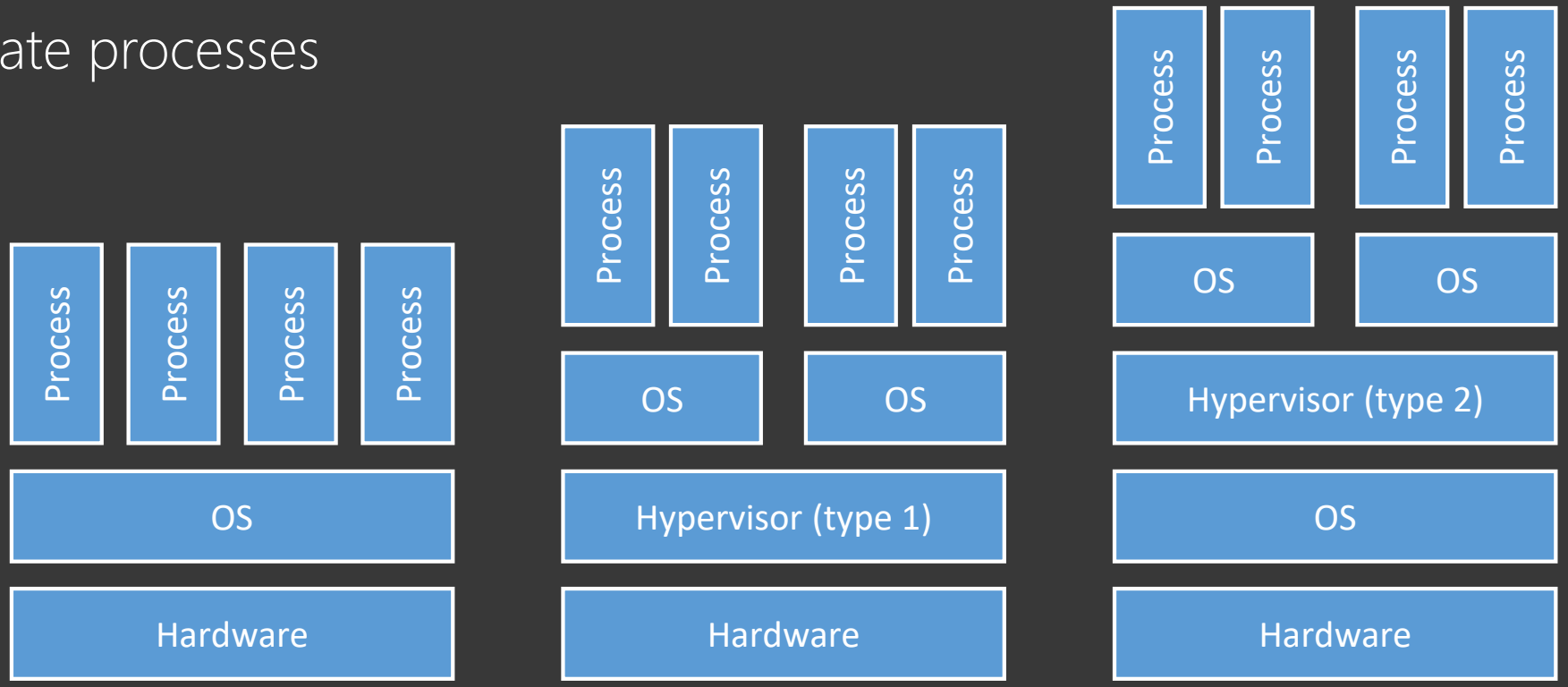
XXX

Container versus VM

Different levels of virtualizations

Hardware virtualization isolates operating systems

Containers isolate processes



Containers are just another option

What are containers?

#TBT: Container are process isolation

Implemented in the kernel

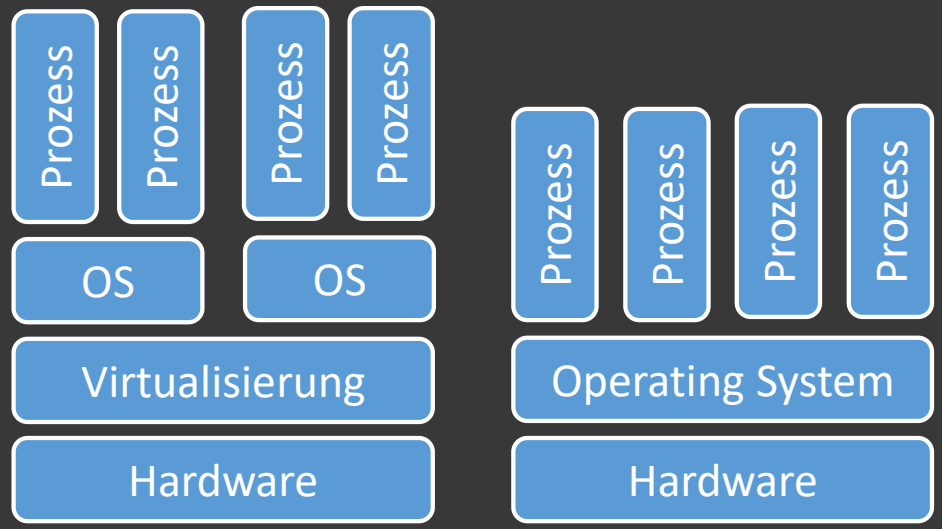
Resources are shared among all processes

Containers versus virtual machines

Other/higher layer of virtualization

Shared hardware and shared kernel

Containers are just another option



Advantages

Development

Reproducible environment

Packaged runtime environment

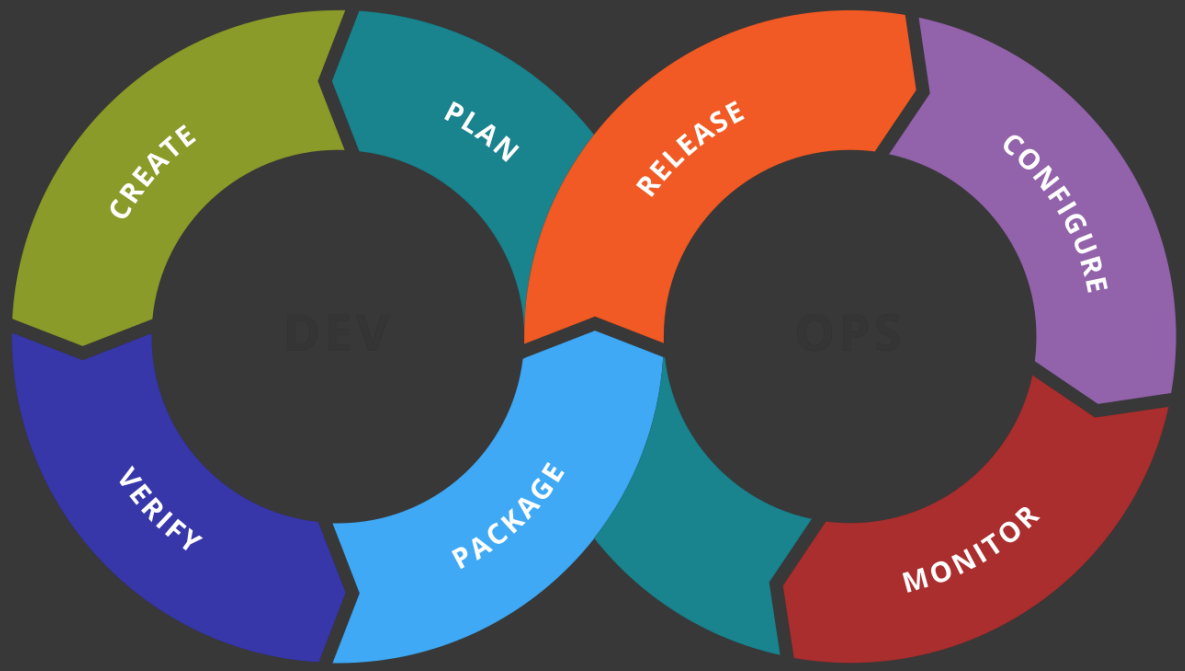
Deployable for testing

Operation

Lightweight virtualization

Density

Dependency management



Nomenclature

Container - running application
Dockerfile - deployment script
Image - packaged application
Registry - image store

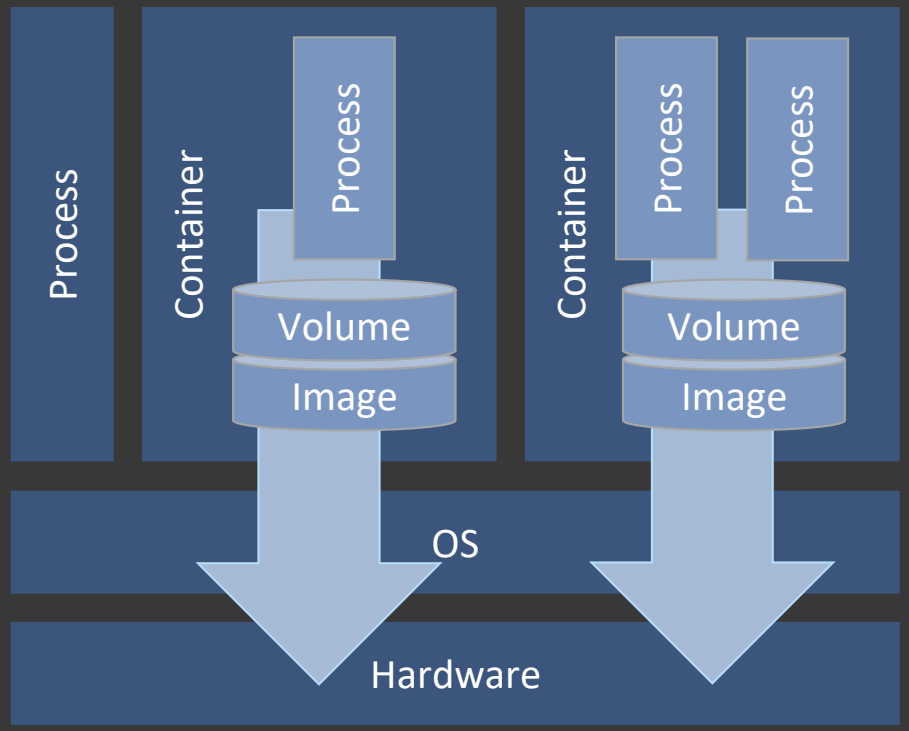
Container Management

Isolated process(es)

Shared, read-only image

Dedicated writable volume

Network configuration



My First Container

Hands-On

```
$ docker run -it ubuntu
root@12345678# hostname
root@12345678# whoami
root@12345678# ps faux
root@12345678# ls -l /
root@12345678# exit
$ docker run -it ubuntu ping localhost
```

Background containers

First process keeps container alive

Containers are not purged automatically

Hands-On

```
$ docker run -it ubuntu hostname
$ docker run -d ubuntu ping localhost
$ docker ps
$ docker stop <NAME>
$ docker rm <NAME>
```

Exploration

Name containers

```
$ docker run -d --name websrv nginx
$ docker ps
```

Learn about containers

```
$ docker logs websrv
$ docker inspect websrv
```

Execute commands inside containers

```
$ docker exec websrv ps faux
$ ps faux
```

Enter containers interactively

```
$ docker exec -it websrv bash
```

Networking

Internals

Daemon controls 172.16.0.0/12

Containers are assigned a local IP address

Outgoing traffic is translated (source NAT)

Containers are not reachable directly

Incoming traffic requires published port

Published ports are mapped from the host to the container

Only one container can use a published port

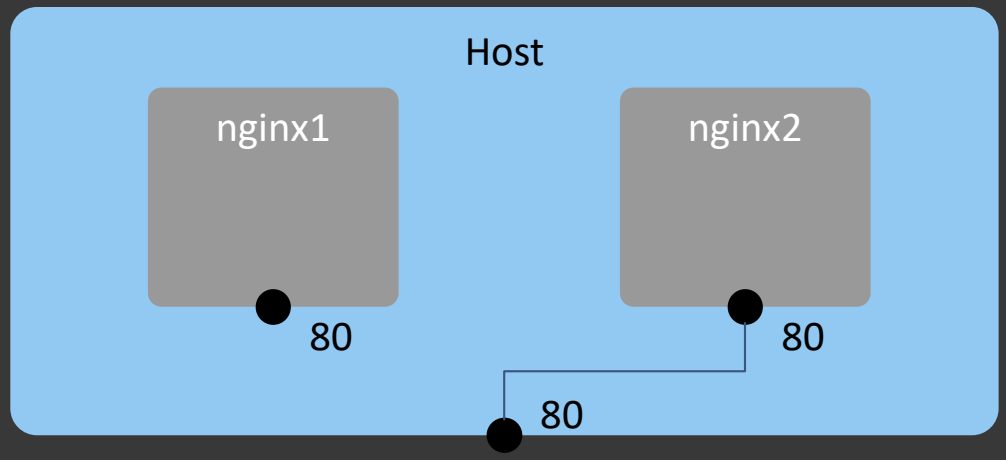
Hands-On

```
$ ifconfig docker0

$ docker run -d --name nginx1 nginx

$ docker run -d --name nginx2
-p 80:80 nginx

$ docker ps
```



Volumes

Non-persistent storage

```
$ docker run -it ubuntu
root@12345678# touch /file.txt
root@12345678# ls -l /
root@12345678# exit
$ docker run -it ubuntu
```

Locally persistent storage

```
$ docker run -it -v /source:/source
ubuntu
root@12345678# touch /file
root@12345678# ls -l /
root@12345678# exit
$ docker run -it -v /source:/source
ubuntu
root@12345678# ls -l /
```

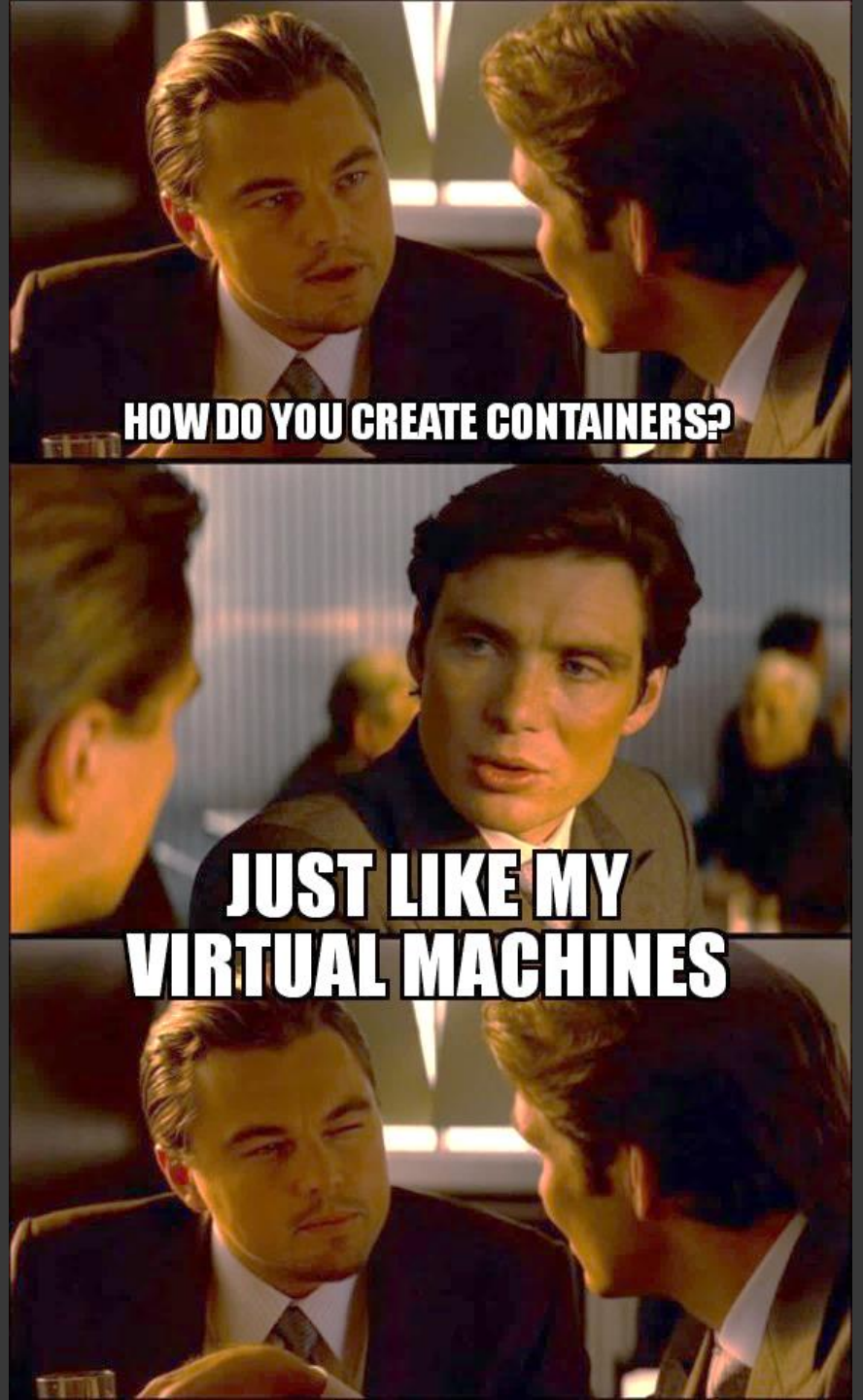
Image Management

Images are served from Docker Hub

Images are named as user/name:tag

Hands-On

```
$ docker pull centos
$ docker rmi centos
```


HOW DO YOU CREATE CONTAINERS?
JUST LIKE MY
VIRTUAL MACHINES

Custom Images

Custom behaviour

Based on existing image

Adds tools and functionality

Simple but sufficient scripting language

Hands-On

```
$ cat Dockerfile
FROM ubuntu:xenial
RUN apt update && apt -y install nginx

$ docker build --tag myimage .
```

Image Registries

Docker Hub is not the only source for images

Private registries based on Docker Distribution

Hands-On (???)

```
$ docker tag myimage nicholasdille/coolnginx
$ docker push nicholasdille/coolnginx
```

Private Registry

Security

`localhost:5000` is preconfigured as insecure registry

Other registries must be secure (HTTPS)

Hands-On

```
$ docker run -d --name registry -p 5000:5000 registry
$ docker tag ubuntu localhost:5000/groot/ubuntu
$ docker push localhost:5000/groot/ubuntu
```

Advanced Concepts

From ~hero to hero

Image Management

Downside of docker CLI

Unable to search for image tags

Unable to remove images/tags from registry

Tools filling the gap

Search for image tags: docker-ls, docker-browse

Remove images/tags: docker-rm (part of docker-ls)

Multi-Stage Builds

Neue Syntax für Dockerfile

Bauen in mehreren Schritten

Mehrere FROM-Abschnitte

Vorteile

Trennen von Build Env und Runtime

Kleinere Images

Weniger Pakete installiert

Pipelines statt Multi-Stage Builds

Kernel internal

Control groups (cgroups)

Resource management for CPU, memory, disk, network

Limitations and priorization

Accounting

Namespaces

Isolation of resources used in cgroups

What images are made of

Images have a naming scheme

`registry/name:tag`

On Docker Hub

- Official images: **`docker.io/_/name:tag`**
- Community images: **`docker.io/user/name:tag`**

Images are described by manifests

Images consist of layers

Instructions in Dockerfile create layers

What images are made of

Images

Name describes purpose

Tag describes version or variant

Images consist of layers

Layers…

…enable reuse

…increase download speed

```
$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
9fb6c798fa41: Pull complete
3b61febd4aef: Pull complete
9d99b9777eb0: Pull complete
d010c8cf75d7: Pull complete
7fac07fb303e: Pull complete
Digest: sha256:31371c117d65387be2640b8254464102c36
Status: Downloaded newer image for ubuntu:latest
```

Dockerfile and Layers

```
$ cat Dockerfile
```

```
FROM ubuntu:xenial
LABEL maintainer=nicholas.dille@haufe-lexware.com
ENV JAVA_VERSION=8u181
RUN apt-get update \
 && apt-get -y install openjdk-8-jdk-
headless=${JAVA_VERSION}*
ADD HelloWorld.java /tmp
RUN javac /tmp/HelloWorld.java
CMD [ "java", "-classpath", "/tmp", "HelloWorld" ]
```

```
$ docker history hello-world
```

LABEL maintainer=nicholas.dille@haufe-lexware.com	2b99001c3d7f	0B
ENV JAVA_VERSION=8u181	72e9a6db461b	0B
RUN apt-get update \ && apt-get -y install openjdk-8-jdk-headless=${JAVA_VERSION}*	ad01f9c12cb6	225MB
ADD HelloWorld.java /tmp	542fdbd5f9b7	112B
RUN javac /tmp/HelloWorld.java	4121bfb4af35	426B
CMD ["java", "-classpath", "/tmp", "HelloWorld"]	c732f0a4d1e7	0B

Image Manifests

Images are described by manifests

Manifests list layers belonging to an image

Layers are stored as blobs

Manifests reference image configuration

Information is referenced by SHA256 hashes

Image Configuration

Lists image layers with commands

Stored as blob

Multi-Arch Images

Images only work on a single platform

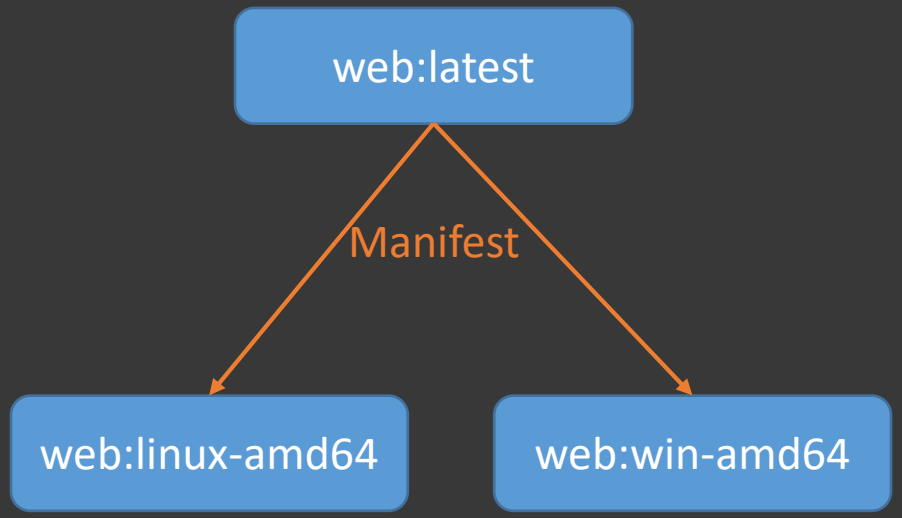
But containers are supported on multiple architectures and operating systems

Virtual images to the rescue

Manifest links to multiple images for supported platforms

Now integrated in Docker CLI (`docker manifest`)

Based on manifest-tool (by Docker Captain Phil Estes)



Official images are already multi-arch

Multi-Arch Images: openjdk

```
$ docker run mplatform/mquery openjdk:8-jdk
Image: openjdk:8-jdk
 * Manifest List: Yes
 * Supported platforms:
   - linux/amd64
   - linux/arm/v5
   - linux/arm/v7
   - linux/arm64/v8
   - linux/386
   - linux/ppc64le
   - linux/s390x

$ docker run mplatform/mquery openjdk:8-jdk-nanoserver
Image: openjdk:8-jdk-nanoserver
 * Manifest List: Yes
 * Supported platforms:
   - windows/amd64:10.0.14393.1770
```

Multi-Arch Images: hello-world

```
$ docker run mplatform/mquery hello-world
Image: hello-world
 * Manifest List: Yes
 * Supported platforms:
   - linux/amd64
   - linux/arm/v5
   - linux/arm/v7
   - linux/arm64/v8
   - linux/386
   - linux/ppc64le
   - linux/s390x
   - windows/amd64:10.0.14393.1770
   - windows/amd64:10.0.16299.19
```

Volume Management

Bind mount

Map a local directory into the container

Easy to exchange data with non-containerized processes

Volume

Managed by Docker daemon

Works well for processing data from containers only

tmpfs

Real temporary data

Removed with containers

Volume Management

Persistent data

Necessity for all production deployments

Backends like databases require persistent data

NFS from central storage

Usually available on-opremises and in clouds

Volume plugins

Delegation of storage handling

Example: sshfs

docker-compose

Infrastructure-as-Code

Deployment of multiple containers

`docker-compose.yml` defines networks, volumes and services

Declarative syntax

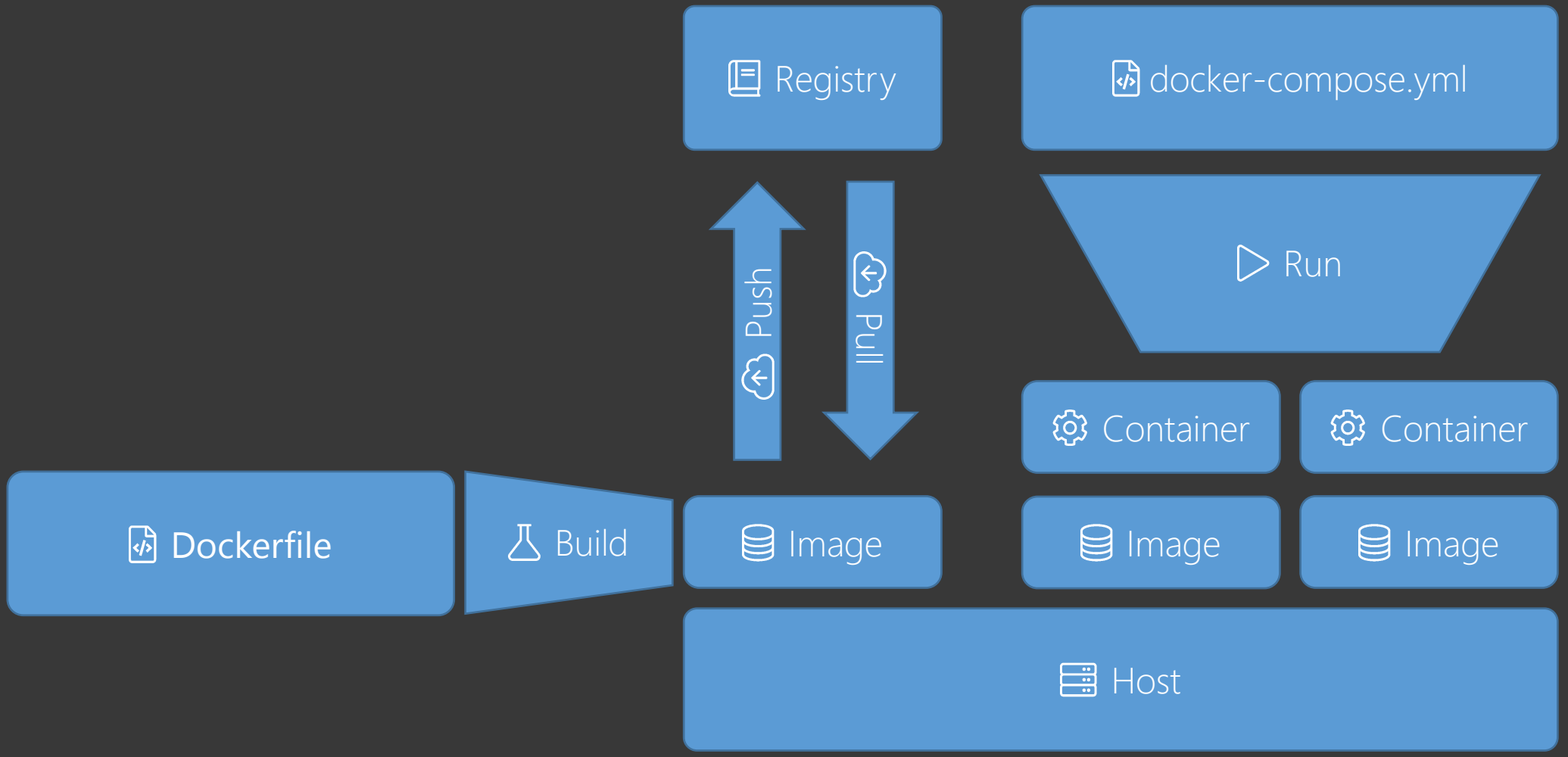
Easy to use

Service Discovery

Dependencies

Scalability

Tools Overview



Network Management

Flavours

bridge

- Default network
- Containers receive private IP address
- SNAT for egress
- Port publishing for ingress

host

- No network isolation
- Direct access to host networks

none

- No network

overlay

- Build networks across multiple hosts

Network Context

Default network context

Used by docker run

Containers are “on their own”

User-defined networks

Automatic DNS resolution of service names to containers

Used by docker-compose

Build from git

Build context

Docker build expects `Dockerfile` in specified directory:

```
docker build <directory>
```

Directory content is packaged and sent to daemon

Excludes defined by `.dockerignore`

Build from remote location

Format: `<url>#<branch>:<directory>`

Example:

```
docker build \
    github.com/nicholasdille/docker-lab#master:advanced/multi_stage
```

docker-app

Reuse application stacks

Distribute compose files using any Docker registry

Can include accompanying files

Requires docker-compose version 3.2

Internals

Stored like an image

Has an manifest with config and one layer

Layer containers deployment information

Reverse Proxy

Multiple service on a host

Port conflicts

Reverse proxy

Routing of requests to correct backend containers

Well-known: nginx, haproxy, traefik

Features

Manual or automatic wiring

ACME (Let's Encrypt)

Container Native Builds

Commands run containerized

Availability of tools

Choice of version

Isolation from host / infrastructure

Tools

Drone CI

Concourse CI

(and more!)

Uses volumes

Clone into volume

Mount volume to every build step

Windows Containers

Other sections

Advantages of containers

Accelerate deployment

Isolation

Standardization through automation

Packaged runtime environment

Notes

Flavours (Windows / Hyper-V containers)

Base images (nanoserver, windowsservercore, windows)

LTSC vs. SAC

Mapping named pipe (Windows Server 1709)

LCOW (Windows Server 1803)

Support (Windows Server 2016 and later)

Licensing (Windows Server Standard vs. Datacenter)

[Notes]

Introduced in Windows 10 and Windows Server 2016

Two flavours: Windows containers, Hyper-V containers

Fast development in SAC releases

Base images: nanoserver, windowsservercore, windows

LCOW

Support and licensing

Comparison with Linux containers

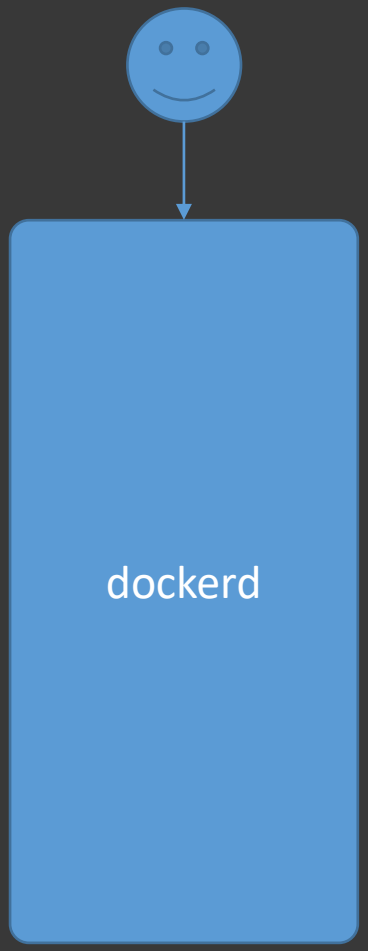
Alternate shell (PowerShell instead of cmd.exe)

Alternate escape character (backtick instead of backslash)

Remoting

Docker Under the Hood

What it looks like

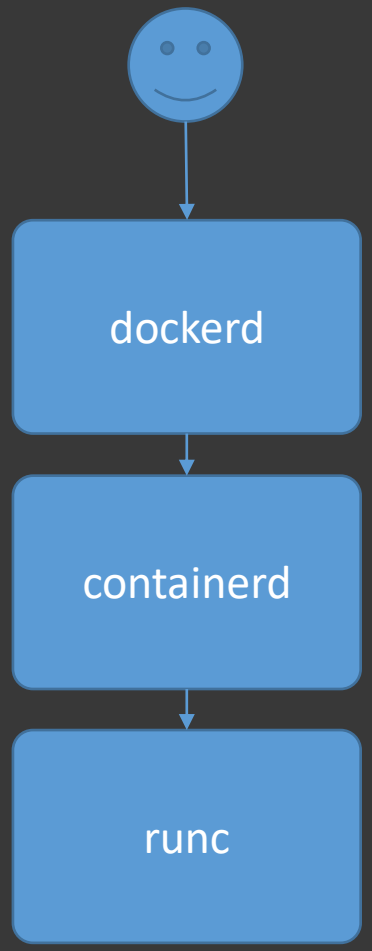


API endpoint
Integration with ecosystem
Automation

Distribution (pull/push)
Container storage
Metrics

Isolates processes
Interacts with kernel
Lightweight

How it works



Available on socket (and TCP)
Offers certificate authentication
Talks to containerd using gRPC

Belongs to CNCF
Implements OCI image spec
Donated by Docker

Belongs to OCI
Implements OCI runtime spec
Donated by Docker in 2015

Docker Engine API

Docker CLI is an API client

All commands are translated into API calls

e.g. docker version → /version

Backwards compatible

New features only available in new CLI

Docker can be controlled using curl

Not very admin friendly ;-)

docker-machine

Management of multiple Docker hosts

Docker remoting using TCP socket

Installs certificates to secure communication

Configuration is stored in `~/.docker/machine`

Support for cloud providers

Drivers extend the functionality

Automatic creation of VMs

Examples

```
$ docker-machine create \
  --driver generic \
  --generic-ip-address=10.0.0.100 \
  --generic-ssh-key ~/.ssh/id_rsa \
  vmname
```

Remoting using SSH

Revolutionary because certificate authentication is hard

Docker 18.09

Installed on client and server

Client requires SSH access to server

Launches remote `docker system dial-stdio`
to proxy API calls to `/var/run/docker.sock`

Usage

```
docker -H ssh://<user>@<host> version
```

Docker Inception

Docker-out-of-Docker (DooD)

Containerized Docker CLI

Allows using other version

Works by Mapping the socket or named pipe

Creates conflicts with other containers

Interferes with the host

Example

```
docker run -it --rm \
    --volume /var/run/docker.sock:/var/run/docker.sock \
    docker:18.06
    docker version
```

Docker-in-Docker (DinD)

Inception

Running a containerized Docker daemon

DinD requires a privileged container

Privileged containers enable host breakouts

Hands-On

```
$ docker run -d --rm --privileged --name dind docker:stable-dind

$ export HOST_IP=$(docker inspect -f '{{range
.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' dind)

$ export DOCKER_HOST="tcp://${HOST_IP}:2375"

$ docker version
```

Patterns and Anti-Patterns

Notes from the Field

Patterns and Anti-Patterns

Designing Images

Tagging Strategies

Latest has no universal meaning

Individual version / build number

Meaningful named tags

`stable` for tested images

`dev` or **`vnext`** for untested images

More tags = choice

`v1.1.0` should also be tagged as **`v1.1`**, **`v1`** and **`stable`**

Latest **`v1.1.0-alpine`** should also be tagged as **`stable-alpine`**

One Process per Container

The optimist

Separating functionality

Enabling scalability

The realist

Multiple processes in a container may make sense

Depends on server

Build versus Runtime

Build parameters

Versions of tools to be installed

Features to enable

Use build arguments

- Define with `ARG` statement in `Dockerfile`
- Supply on build: `docker build --build-arg <name>=<value>`

Runtime parameters

Configure behaviour

Use environment variables (`ENV` statement in `Dockerfile`)

See Tweaking runtime behaviour

Using Proxy Servers

Do not hardcore in Dockerfile

During build

```
docker run --build-arg http_proxy --build-arg https_proxy .
```

During runtime

```
docker run --env http_proxy --env https_proxy ubuntu
```

Docker daemon

Honours environment variables `http_proxy`, `https_proxy` and `no_proxy`

Tweaking Runtime Behaviour

ENV

Do not hardcode values into commands

Use environment variables

Set reasonable defaults

CMD and ENTRYPOINT

Changes behaviour on start

Shell and exec notation

Determines whether a command is wrapped by a new shell

Version Pinning versus Using latest

Downsides of using latest

Breaks reproducibility

Causes conflict with two services based on the same image

Version pinning in Dockerfile

Hard/impossible to determine running image version (see microlabeling)

Upsides of using latest

No need for version pinning

Always receive updates

Strong downs but weak ups

Derive from code

Using community images is like buying a pig in a poke

Community images may not receive updates

Community images may follow undesirable paths

Community images may introduce security issues

Community images may not be updated at all

Solution

Fork code repository and build yourself

Patterns and Anti-Patterns

Building Images

Signal Processing (PID 1)

Even containerized services want to exit gracefully

Only containerized PID 1 receives signals

Use **`exec`** when starting from scripts

Multiple processes require an init process

Choices include **`supervisor`**, **`dumb-init`**, **`tini`**

How to prevent init processes

Isolate in sidekicks

Readability beats size

Myth: More layers reduce access time

My own tests prove otherwise

Layers improve performance on pull (parallel downloads)

Recommendation: One layer per installed tool

Base and Derived Images

Separate functionality into chains of images

dind → dind-gocd-agent

→ linux-agent → linux-agent-gocd

→ linux-agent-jenkins

→ linux-agent-gitlab

Enables code reuse

Isolates changes

Order of Statements

Build arguments

Used for controlling version pinning

Environment variables

Used for tweaking runtime behaviour

Tools and dependencies

E.g. Install distribution packages

New functionality

Packages and scripts required for the purpose of the image

The build cache helps you build faster!

Validate Downloads

Distribution packages are validated

Most package manager include validation

Downloads from the web

Obtain file hash from the web

Create file hash after manual download

Check file hash during image build

```
$ echo “${HASH} *${FILENAME}“ | sha256sum
```

Run as USER

By default everything as root

Bad idea™

```
FROM ubuntu
# install
USER go
```

Force user context

Add USER statement after setting up image

Some services handle this for you (nginx)

Downstream images

Change to root

Install more tools

Change back to user

```
FROM derived
USER root
# install
USER go
```

Use microlabeling

Mark images with information about origin

Easily find corresponding code

Use image annotations by the OCI

Deprecated: https://label-schema.org

Also breaks the build cache :-(

Example…

Use microlabeling

Dockerfile

```
FROM ubuntu:xenial-20180123

LABEL \
org.opencontainers.image.created="2018-01-31T20:00:00Z+01:00" \
org.opencontainers.image.authors="nicholas@dille.name" \
org.opencontainers.image.source="https://github.com/nicholasdille/docker" \
org.opencontainers.image.revision="566a5e0" \
org.opencontainers.image.vendor="Nicholas Dille"

#...
```

Tipps and Tricks

Pull during build

Prevent usage of outdated images

```
docker build --pull …
```

Timezones

Synchronize time

```
docker run -v /etc/localtime:/etc/localtime …
```

Derive dynamically

Build argument defines default base image

```
ARG VERSION=xenial-20180123
FROM ubuntu:${VERSION}
```

Patterns and Anti-Patterns

Running Containers

Pitfall of using latest

YNKWYGG

You Never Know What You're Gonna Get

Outdated image

New containers are started based on existing images → Pull policy

Multiple services using different latest

Same image but rolled out at different times

Reschedule will break at least one of them

Cleaning up automatically

Handling containers required testing

Run containers to test something

Run tools distributed in containers

Many exited containers remain behind

Temporary containers can be removed automatically

docker run --rm ...

Housekeeping

Cleanup before build

Create sane environment to work with

Cleanup after build

Save space

Commands

```
docker ps -aq | xargs -r docker rm -f
docker images -q | xargs -r docker rmi -f
```

Custom Formats

Default output is very wide

Output of most Docker commands creates line breaks

Define condensed output

Most Docker commands allow custom formats using `--format`

File Permissions on Volumes

Problem statement

Use containerized tool with bind mount (mapped local directory)

Creating files on volumes get owner from container

Often creates root-owned files and directories

Those cannot be removed by user

Solution

Launch container with different user

May break container!

Security

Containers FTW

Isolation

Process thinks it owns the OS

Resource management

Runtime package

Everything the containers needs (chroot)

Distributable package

Automation

Builds are reproducible

Deployments are faster

Additional Line of Defense

Hard to analyze surroundings

Unable to see other processes (cgroups)

Isolated resources (namespaces)

Kernel manages isolation

XXX

Automation leads to Transparency

Reproducibility / Reliability

xxx

Infrastructure-as-Code

xxx

Audit

xxx

Client / Server Architecture

Only root can start containers

Docker CLI talks to Docker Engine API

User access through group membership:

```
# ls -l /var/run/docker.sock
srw-rw---- 1 root docker 0 Jan 28 16:35 /var/run/docker.sock
```

User access after login to server!

User / Group ID

Privilege Escalation

User in container is the same as on host

```
docker run -it ubuntu
```

Statement `USER` in `Dockerfile` can be overridden:

```
docker run -it --user 0:0 ubuntu
```

Containers are just an isolated process!

Volumes

Privilege Escalation

Mount arbitrary host directories

Access host files with user/group ID from container:

```
docker run -it --volume /:/host ubuntu
```

Can also be combied with `--user` to make this work for any image

Privileged Containers

Host breakout

If able to start containers, just leave the isolation:

```
docker run --privileged --pid=host -it alpine:3.8 \
    nsenter -t 1 -m -u -n -i sh
```

Work with namespaces (`nsenter`)

Uses process tree of host (`--pid=host`)

Get namespace from PID 1 (`-t 1`)

Enter namespaces required for shell (`-m -u -n -i`)

Remediation: Using Capabilities

Avoid privileged containers

Often only some capabilities are required

Explicitly add or remove capabilities

```
docker run -d --cap-add SYS_ADMIN ubuntu
```

Careful

Adding all capabilities is equivalent to privileged execution

Image Builds

Credential disclosure

Files removed in higher layer are „invisible“

But they are still present

Example Dockerfile

```
FROM ubuntu
ADD id_rsa /root/.ssh/
RUN scp user@somewhere:/tmp/data .
RUN rm /root/.ssh/id_rsa
```

Network access

Access to Docker Engine API

TCP socket (usually 2375 for HTTP and 2376 for HTTPS)

Server authentication

Create certificate authority

Create server certificate

Client can check authenticity based on CA

Client authentication

Create client certificate based on same CA

Server can check authenticity based on CA

Unsecured TCP Access

Host break in

Anyone can start containers:

```
docker --host 1.2.3.4:2375 run -it --privileged ubuntu
```

Privileged containers enable host breakouts

Workarounds

Always run with certificate authentication!

Or better: Don't expose the Docker daemon on TCP

Or even better: Use remoting over SSH in Docker 18.09+ (more info here)

Remediation: User Namespace Remapping

How it works

User IDs in containers are mapped to dedicated range on the host

Example: Container UID 0 is mapped to host ID 12345

Advantages

Containers cannot modify files as root

```
docker run -it --rm --volume /:/host ubuntu
```

Disadvantages

Files written to local volumes are owned by „strange“ IDs

Public Registry

Poisoned images

Who do you trust?

Community images?

- Regular security updates? Unknown!
- Version updates? Unknown!

Official images?

- Regular security updates? Probably
- Version updates? Sure!

If suspicious, derive from code!

Remediation: Image Signing

Notary

Protects manifest from changes

Digital signature is added to manifest before upload to registry

Complex to set up

Docker Content Trust

Part of Docker Enterprise (paid)

Based on Notary

Remediation: Docker Bench Security

Check host for security issues

Conveniently packaged:

```
# docker run -it --net host --pid host --userns host --cap-add audit_control \
>     -v /var/lib:/var/lib -v /var/run/docker.sock:/var/run/docker.sock -v /usr/lib/systemd:/usr/lib/systemd \
>     -v /etc:/etc --label docker_bench_security \
>     docker/docker-bench-security

# ------------------------------------------------------------------------------
# Docker Bench for Security v1.3.4
#
# Docker, Inc. (c) 2015-
#
# Checks for dozens of common best-practices around deploying Docker containers in production.
# Inspired by the CIS Docker Community Edition Benchmark v1.1.0.
# ------------------------------------------------------------------------------
```

https://github.com/docker/docker-bench-security

Remediation: Kubernetes Pod Security Policy

Prevent common security issues

Describe what to allow and prohibit

Example:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
spec:
  privileged: false
  requiredDropCapabilities:
    - ALL
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    rule: 'MustRunAsNonRoot'
```

https://kubernetes.io/docs/concepts/policy/pod-security-policy/#example-policies

Summary

Containers…

…are fun

…help us be faster

…help us be more reliable

Security…

…is often ignored

…is worse

…remediations make containers less useful

Container Concepts

Process isolation

Container = Process(es)

Kernel features

Namespaces for isolation

Control groups for resource management

Containers are Microservices

Separate functionality

Do not mix

Improves scalability

Minimize functionality

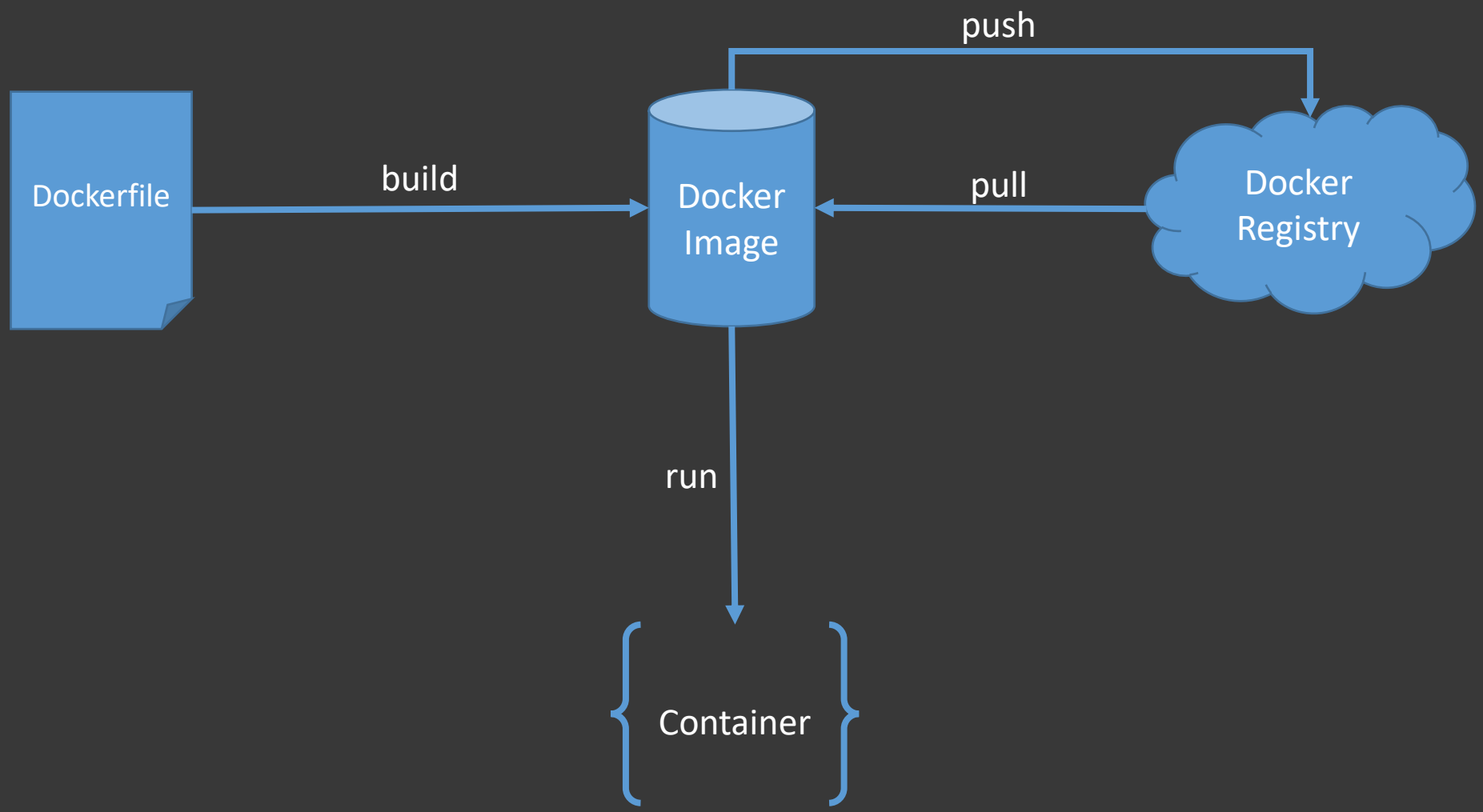
Single purpose

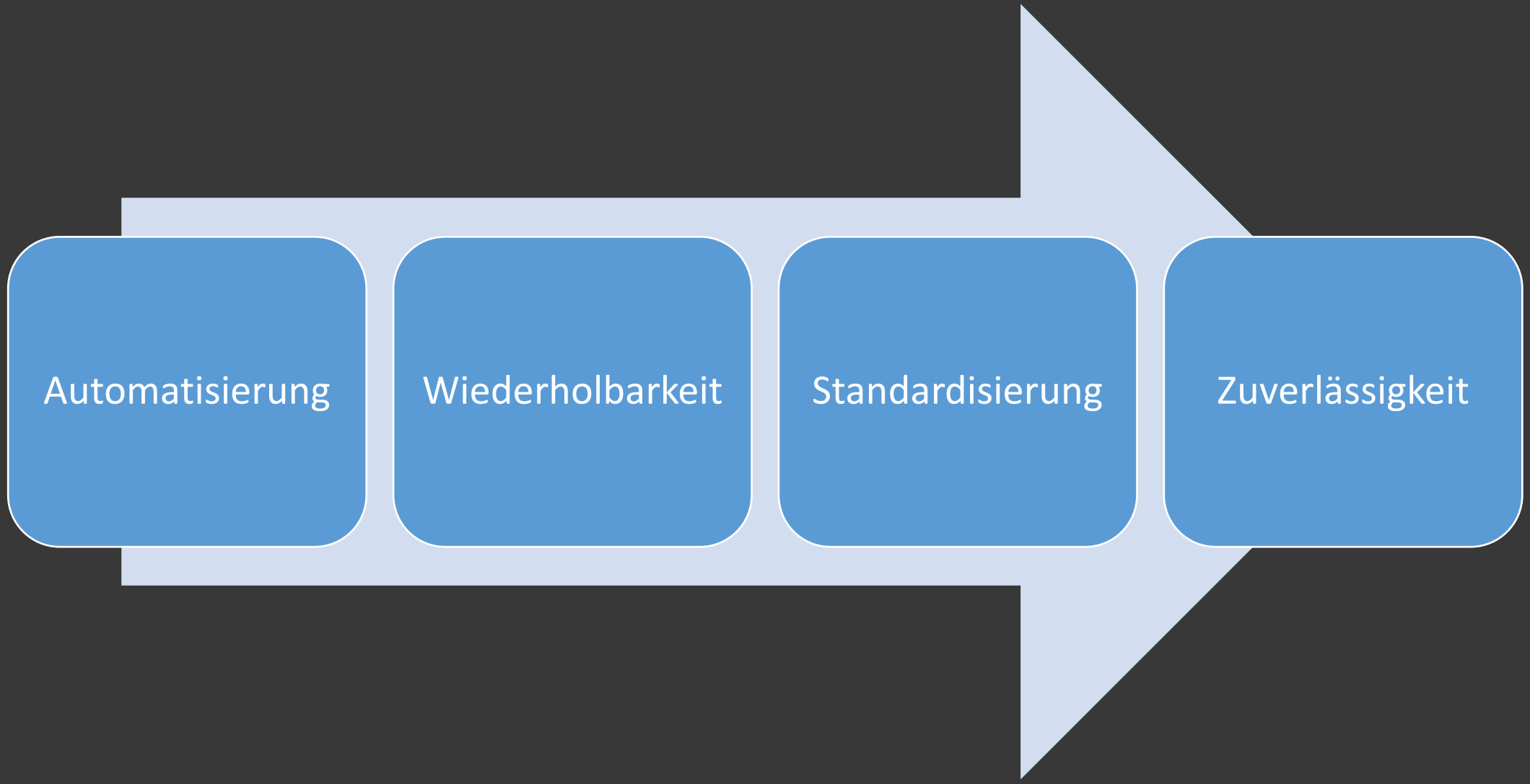
Preferably only one process

Simplicitly

Improves updating

Docker Workflow




Automatisierung
Wiederholbarkeit
Standardisierung
Zuverlässigkeit

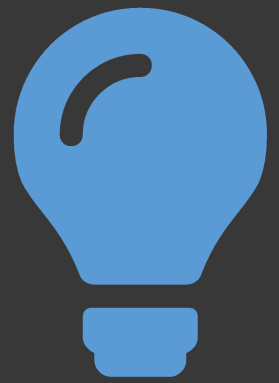
Learn from software development

Short feedback loops

Agile development

Fail early

Automated testing



Learn from Software Development

Short feedback loops

Run builds on every commit

Build must include tests

Build artifacts should be packaged/deployed

Fail early

Do not ignore failed tests

Assume responsibility

Task is done when build and tests are successful

Continuous…

…Integration

Automated build

Automated tests on commit

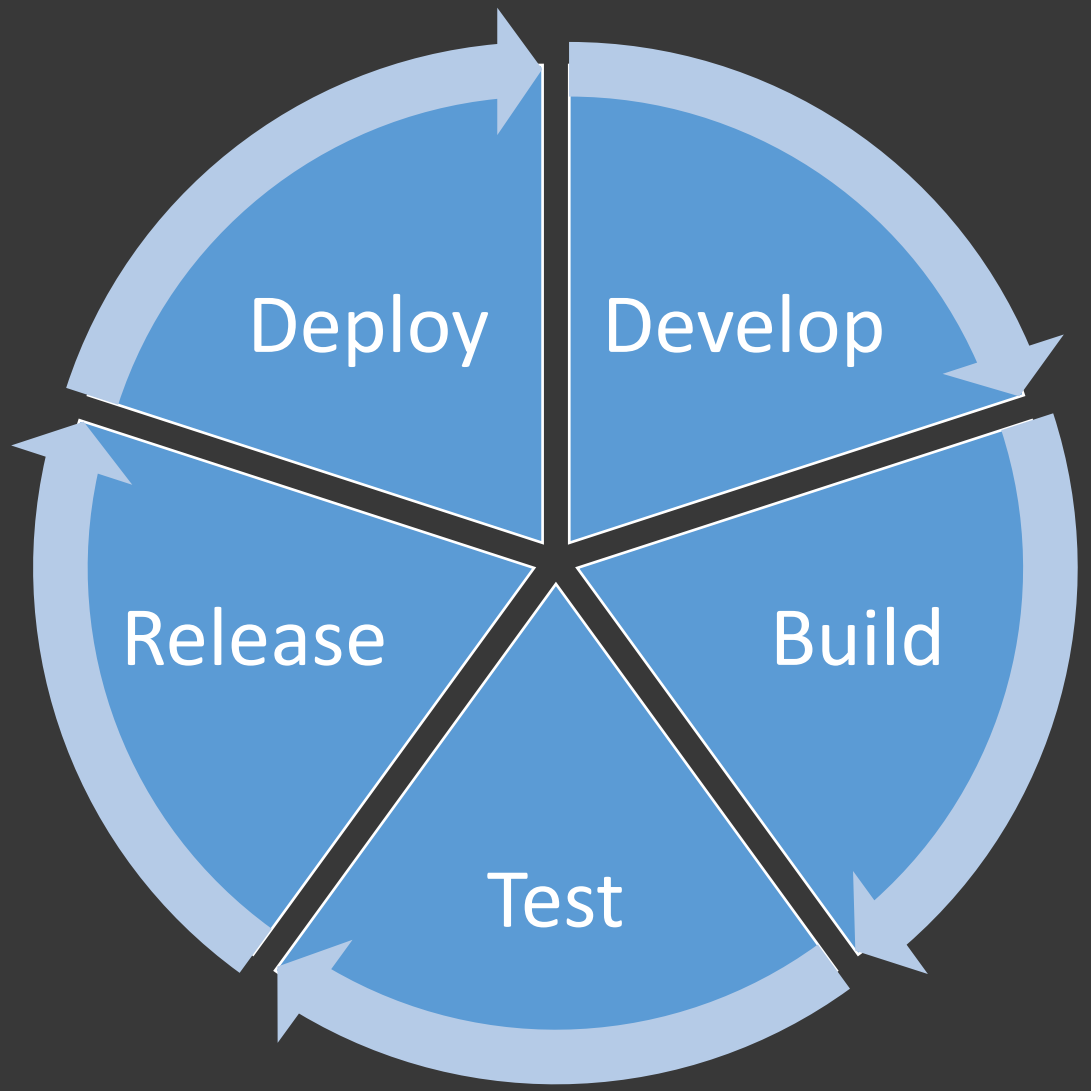
…Deployment

Automated installation but…

…manual approval for production

…Delivery

Automated publishing to production



Pipeline Concepts

Basic features

Scheduling on changes

Exit on error

Serialized execution

Stages / steps

Parallel execution

Jobs / groups

No tool discussion

This is about concepts

Tools are interchangable

Stack

Code repositories

Gitea

Container registry

Docker distribution (registry)

CI/CD

Drone CI

Artifact store

WebDAV

Repository Structure

Pipeline-as-Code

Instructions stored with code

One repository per component

Separate loosely coupled components

Single repository for dependent components, e.g. CLI and backend

Plan for reuse

Base images are like libraries

Separate build and deployment

Multiple builds, one deployment

Builds triggered on code commits

Deployments triggered on…

- …completed builds
- …changed deployment instructions

Challenge

Deploy matching components

Git does not know how many commits have been pushed

Pipeline does not know which commit was successfully build

Separate build and package

One build, many packages

Builds triggered on code commits

Packaging triggered on completed builds

Packages are deployed

Archives

Package managers (Maven, npm, nuget etc.)

Docker images

Pipeline Steps

Pipeline Dependencies

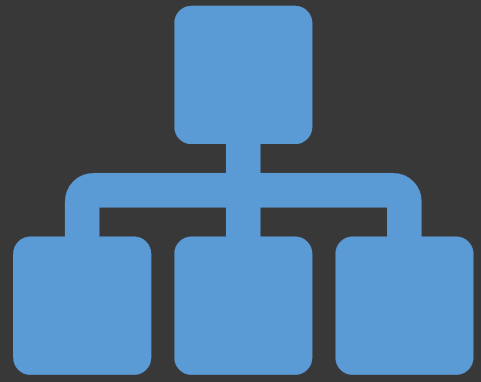
Multiple pipelines?

Instructions tied to code?

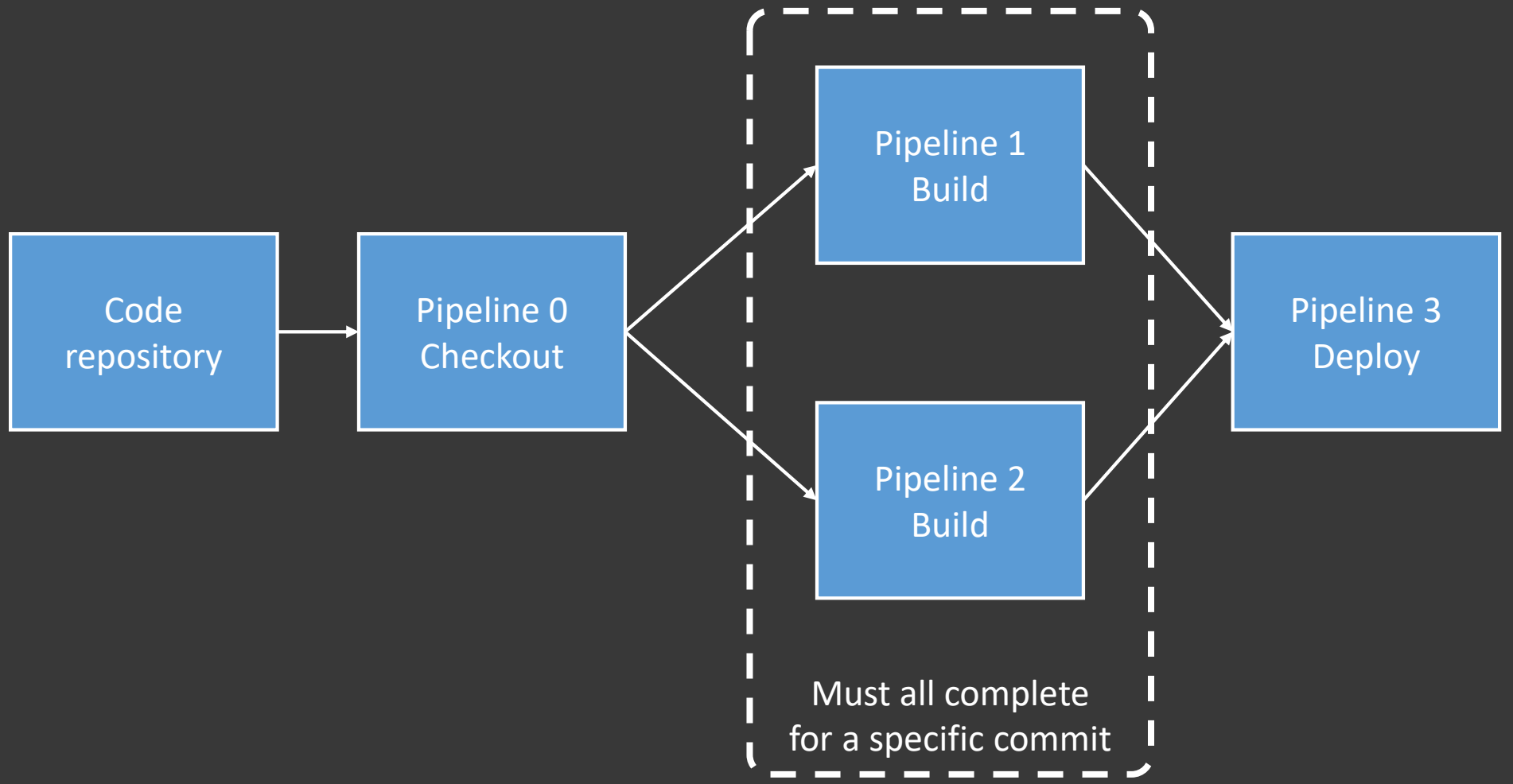
Every dependency will trigger pipeline

Possibly many rebuilds

Synchronization difficult



Fan In


Code repository
Pipeline 0 Checkout
Pipeline 1 Build
Pipeline 2 Build
Pipeline 3 Deploy
Must all complete for a specific commit

Fan-In

Synchronize builds

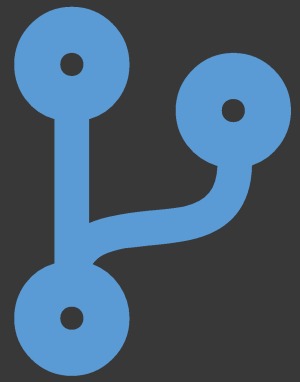
Multiple components based on same dependency

Enforce identical dependencies

Otherwise: Unmatched libraries

Different behavior

Bugs

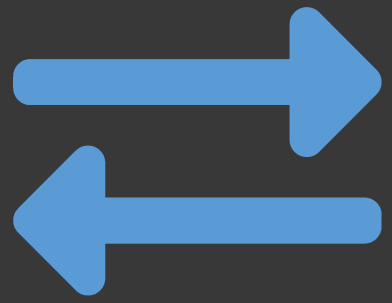


Dependencies versus Triggers

Dependency

Pipeline is triggered automatically by upstream

Controller needs to know about all pipelines



Trigger

Downstream is triggered explicitly

Dependency > Trigger

Pipeline with inputs and outputs

Instead of order of execution

Artifacts

Pass results between pipelines

Prevent rebuilds

Immutable build results

Pull only once

Everything is based on the same commit

Pass as artifact

Packaging

One build, many packages

Single compilation

Multiple distribution packages (Maven, npm, nuget etc.)

Separate pipelines or pipeline steps

Parallel creation of packages

Testing

Containers

Existence of files

Running processes

Open ports

Specialized tools

serverspec

Pipeline Steps

Tagging

Unique build number

Use consecutive numbering

Do not use git commit hash

Tag untested builds

Assign `:dev` tag

Tag tested builds

Assign `:stable` tag

Deployment

Environments

Dev, QA, Live

Myth: Zero downtime

Databases usually break this

Blue/green deployment

No incremental changes

Deploy from scratch

Rolling upgrade

Applicable for certain workloads

Parameterize

Reuse build instructions

Environments

Dev, QA, Live

Deployment scenarios

On-Premises

Hoster

Cloud provider

Tooling

Build script

Independent of build system

Full instructions stored in repository

Integration

Better usuability

Prepared build steps

Hidden complexity

Secrets

Sensitive data

Passwords

Key pairs

Out-of-band management

Do not create/populate from CI/CD

Secrets live longer than pipelines

Clean

...in pipeline

House keeping after pipelines end

Remove containers

Optional: Remove images

...in registry

House keeping in regular intervals

Remove old builds (what is old?)

Remove unused development build

Build Cache

Accelerated container builds

Incremental changes running on the same Docker daemon

CI/CD breaks the cache

Consecutive builds run on different agents

Cache warming

Pull latest artifact

Build based on artifact

Multi-Stage Builds

Separate build and runtime environment

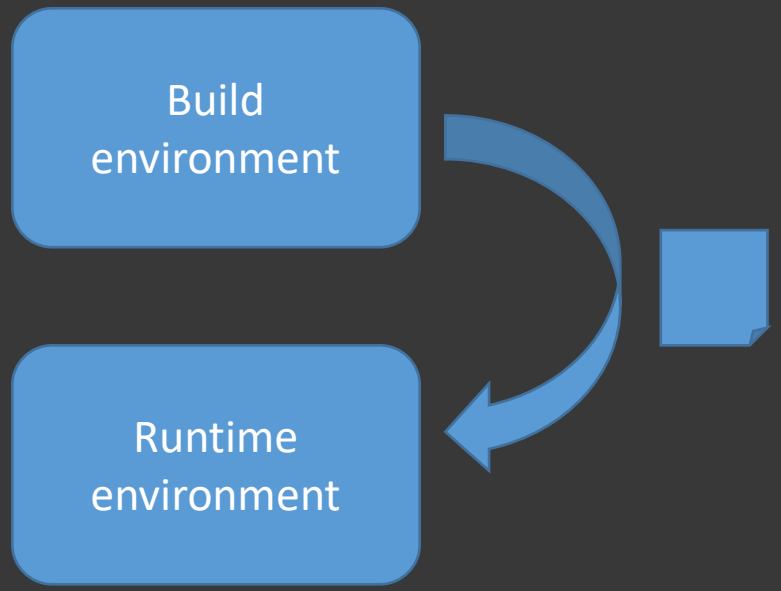
Poor man's pipeline

Two or more steps

Usually for build and runtime environment

Prerequisites

Docker CE 17.05



Kubernetes

Concepts

More explicit than docker-compose

Pods are the smallest unit

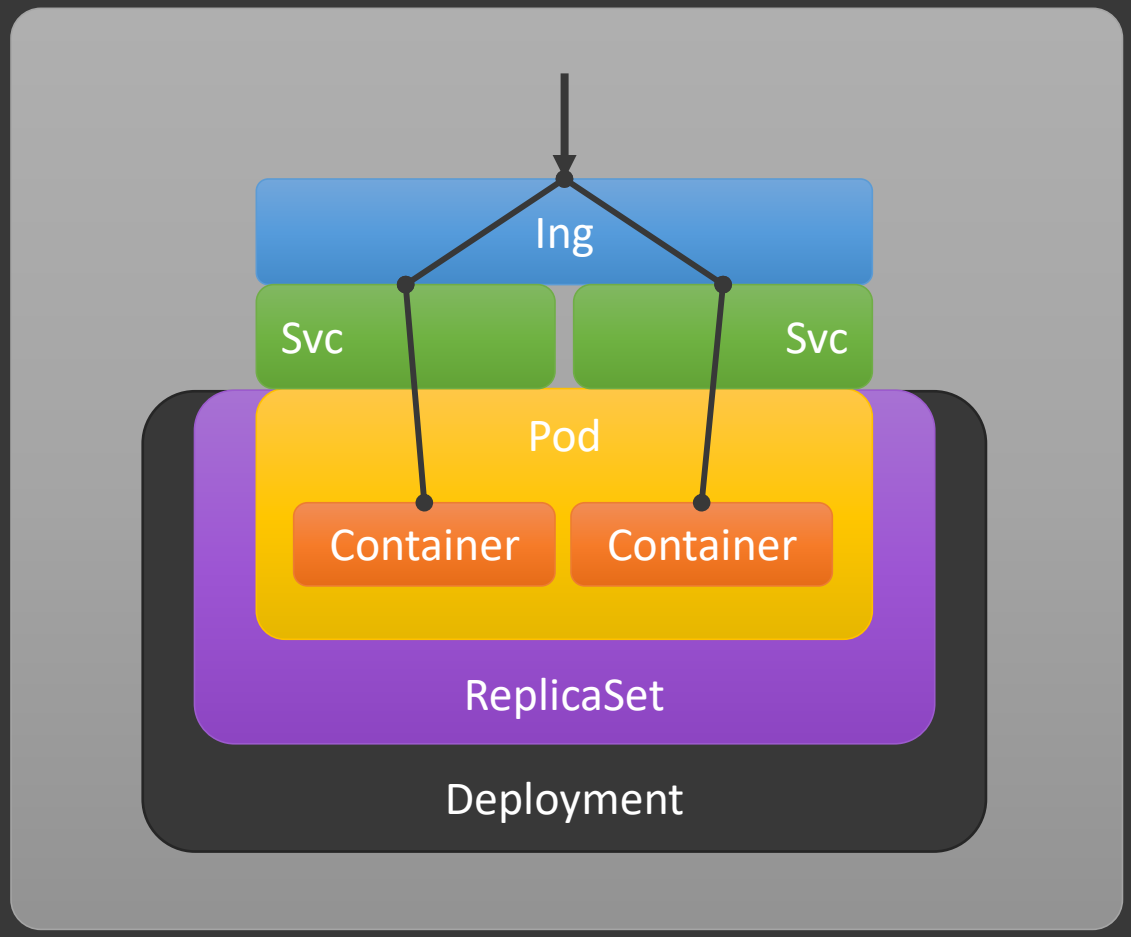
Pods contain containers

ReplicaSet manage the number of replicas

Deployments add deployment strategies

Services (Svc) define private network endpoints

Ingresses (Ing) provide external access



Lessons Learned

Pipeline-as-Code

- … because build instructions belong with the code

Pipeline steps

- Build, test, package, test, deploy

Parameters

- Reusable build instructions

Tagging

- Create traceable images

Testing

Quality Assurance - Theory

Software development

ISO 9126

Functionality

Are the required functions available?

Reliability

Does it repeatedly work correctly?

Usability

Is it easy to use?

Efficiency

How efficient is it?

Maintainability

Is it easy to modify including 3rd parties?

Portability

Is it usable in another environment?

Quality Assurance - Measures

Measures to take

Functionality Testing Security	**Reliability** Testing	**Usability** Documentation
Efficiency Minimalistic	**Maintainability** Documentation	**Portability** Minimalistic

Minimalistic Images

Minimize functionality

Image with single purpose

Improves scalability

Isolate functionality in base and derived images

Improves maintainability

Enables code reuse

Plan for one or very few processes

One process may not always work

Plan for signal handling

Multiple processes require init process

One Process per Container

See link

Signal Processing

See link, link, link

Documentation of Images

Where to document

Add README.md to repository

Fill description on Docker Hub

What to document

Purpose

Tagging strategy

Dependencies

Configuration

Examples

Comments

Tagging Strategy

See link

Pipeline Steps

See link

Testing

Availability of files

Only test critical files like `/entrypoint.sh`

Versions of tools

Test versions to prevent breaking changes

Test to document image contents

Running processes

Make sure background processes are started

Functionality

Test processes produce correct results, e.g. REST API

Add quality gate to automation

Automated Testing

RSpec

BDD framework for Ruby

serverspec

Based on RSpec

DSL feels more like writing ruby

Testinfra

Based on python

DSL also more like writing python

serverspec: Example

```
require "serverspec"
require "docker"

describe "Dockerfile" do
  before(:all) do
    image = Docker::Image.build_from_dir('.')

    set :os, family: :debian
    set :backend, :docker
    set :docker_image, image.id
  end

  it "installs the correct Ubuntu" do
    expect(os_ver).to include("Ubuntu 14")
  end

  it "installs required packages" do
    expect(package("nodejs")).to be_installed
  end

  def os_ver
    command("lsb_release -a").stdout
  end
end
```

Demo: Testing using goss

Automated server validation

Unit tests for servers

Tests are expressed in YAML

Demo

$ dgoss edit nginx

goss autoadd nginx

$ dgoss run nginx

Security

Secure images

Regular builds
Security scans
Credential disclosure
Image signing

Secure runtime

Separate registry for production
Using latest
Privileged containers
Namespace remapping

Security through Automation

Automated builds

Build on change

Regular build triggered by timer

Automated scans

Scan for vulnerable dependencies

Add quality gate to automation

Products/services: CoreOS Clair, JFrog Xray, Docker Hub (and many more)

Prevent Credential Disclosure

See link

Image Signing

Registry uses SHA256 digests

Image manifest references image configuration

Image manifest references blobs

Digests alone do not ensure integrity

Image signing prevents modifications

Cryptographic signatures on image manifests

Docker Trusted Registry in Docker Enterprise Edition

Based on Notary

Isolated Registry for Production

Problem statement

Single registry cannot prevent tampering

Permissions per image not available or feasible

Integrity through isolation

Prevent tampering with production grade images

Permissions through separation

Only very few accounts (preferably automation accounts) are able to access registry

Using latest

See link

Privilege Escalation using Volumes

Accessing root-owned files

Host directories are accessed with used ID from container

Example: `docker run -it --volume /:/host ubuntu`

Generating root-owned files by accident

Using containerized tools

```
docker run -v $(pwd):/src --workdir /src nodejs npm install
```

Solution: `docker run -v $(pwd):/src -u $(id -u) …`

Privileged Containers

See link, link

Namespace Remapping

See link

Lessons Learned

QA also applies to images

Automation is key

Repetition creates confidence

Security is also important for containerization

Knowledge transfer helps

Running in Production

Orchestrators

Purpose

Manage multiple container hosts

Schedule services across all hosts

Restarted stopped / killed / lost containers

Well-known orchestrators

Kubernetes

Docker Swarm

Rancher Cattle (1.6)

Load Balancing

Do not publish ports

Only one container can use one port

Hosts will quickly run out of ports

Services are expected to run on well-known ports (443 for HTTPS)

Deploy containers to handle LB

Forwards traffic to target containers

Use Server Name Indication (SNI) for multiple HTTPS services on the same port (443)

Persistent Storage

Docker volumes are locally persistent

Data only lived on a specific host

Volume drivers integrate with storage system

Volumes are mounted from storage system

Volumes are available on all hosts

Resource Management

Reservations

Containers can have assigned minimum resources

```
$ docker run --memory-reservation 1024m …
```

Limitations

Upper limits apply only if resources are available

```
$ docker run --memory 4096m …
```

Alpine versus Distribution Image

Minimal image size

Ubuntu: 150MB

Alpine: 5MB

Advantages

Faster pull, fast deployments

Disadvantages

Alpine links against libmusl

Well-known tools have less/different parameters

Containerizing System Services

Requirements

Few to no prerequisites for Docker hosts

Automatically created using orchestrator or `docker-machine`

Containerized services

Can be deployed from orchestrator

Health is monitored by orchestrator

Example: ntpd

Third Party Dependencies

Dependency hell

Just like software

Developers rely on existing libraries

Those libraries can and will have security issues

Security scans

Scan for known issues in packaged dependencies

Sometimes hard to remediate

Change Management

ITIL

Company often work by ITIL

Change management is used to document changes to live sytems

Changes usually need to be schedules in advance

Automated deployments

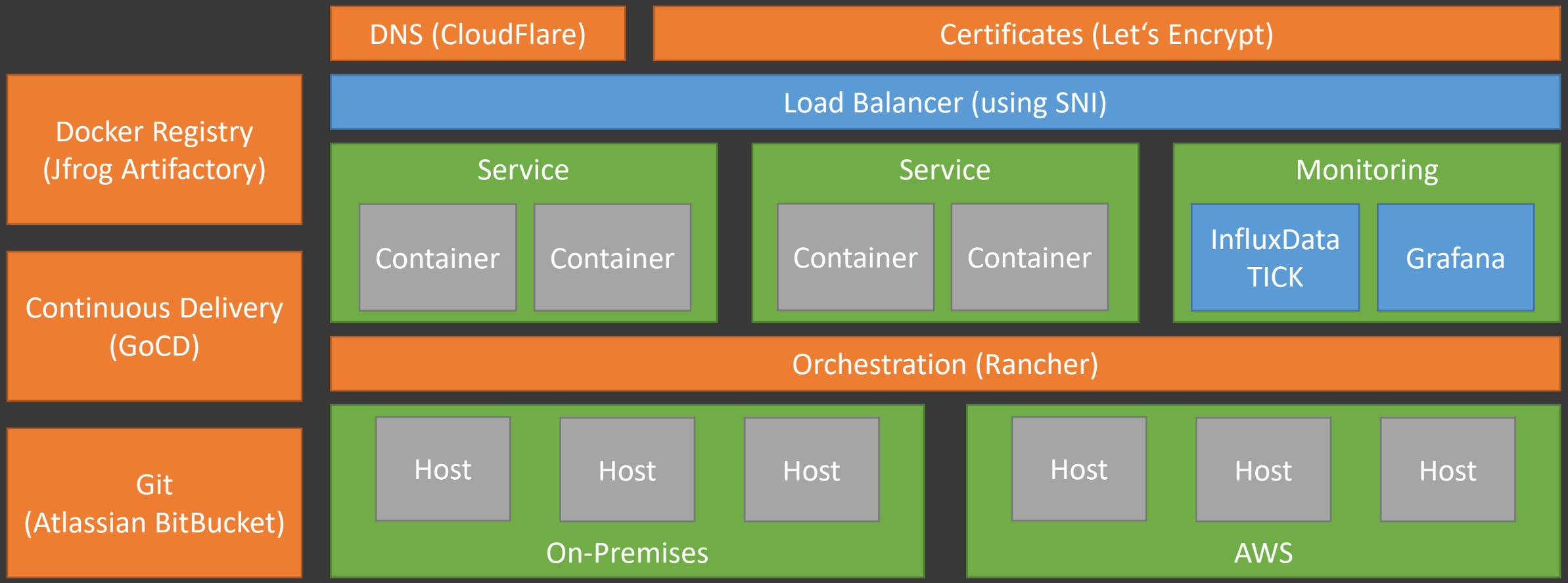
…can cause rollouts without planning a change

…require trust between change manager and DevOps teams

…must be automatically appoved

…must limit change to agreed upon maintenance windows

Example Stack



Monitoring

Monitoring

Basic monitoring out of the box

Containers can be monitored by a single agent on the host

Hands-On

Look at grafana

Show usage of mapped named pipe

Ecosystem 2017

Offerings by Docker

Enterprise Edition

Universal Control Plane | Trusted Registry | Docker Certified

Community Edition

Docker CLI | dockerd | Docker Hub | Machine | 

The Moby Projekt

Announced at DockerCon (May 2017)

Released components of Docker CE

Projects

runC / containerd / LinuxKit / SwarmKit

Notary / Registry / Compose

libnetwork / DataKit / BuildKit / VPNKit / InfraKit / HyperKit

Downstream projects

Docker CE / EE

Balena (Container Engine für IoT)

Governed by Technical Steering Committee

Offerings by Docker

Enterprise Edition

- Universal Control Plane
- Trusted Registry
- Docker Certified

Community Edition

- Docker CLI
- dockerd
- Docker Hub
- Machine
- 

Moby

- SwarmKit
- containerd
- runC
- LinuxKit
- Compose
- Registry
- Notary
- libnetwork
- DataKit
- BuildKit
- VPNKit
- InfraKit
- HyperKit

LinuxKit

Minimal, immutable Linux

Definition in YAML

Build with Moby

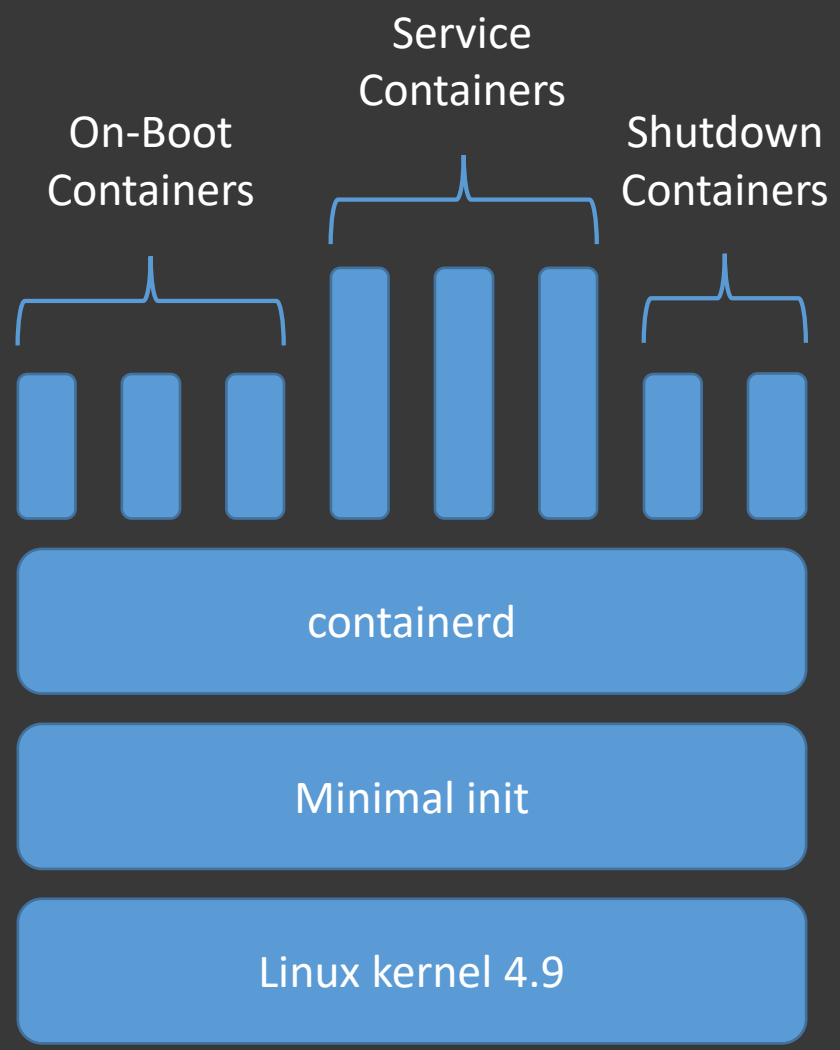
Based on containerd

Everything is a container (including system daemons)

Images are based on Alpine Linux

Images are signed using Notary

Supports multiple platforms



containerd

Manages containers

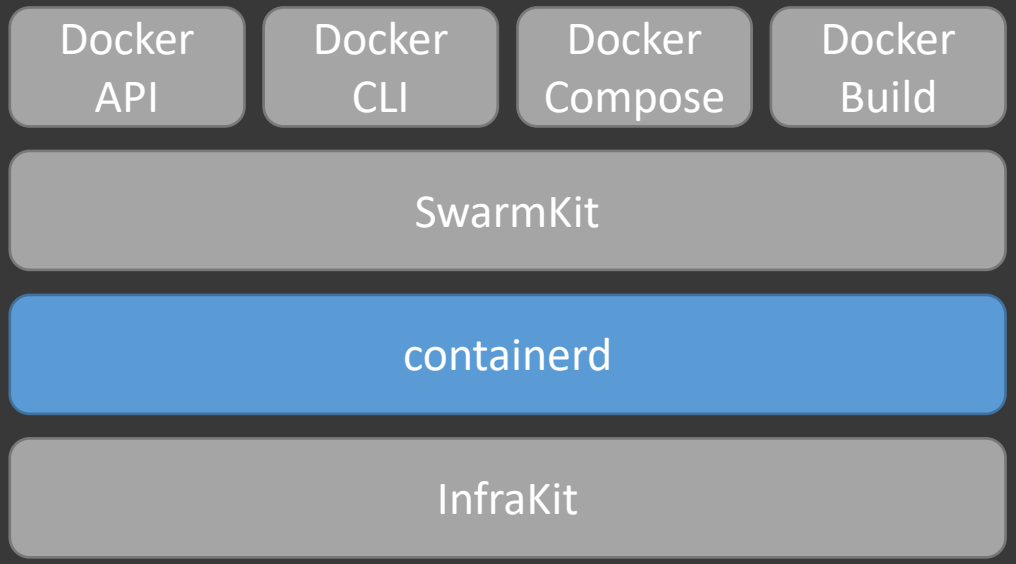
Based on runC (container runtime)

Adds pull and push of images

gRPC API using UNIX socket

CLI `ctr`

Owned by CNCF



Notary

Ensures data integrity

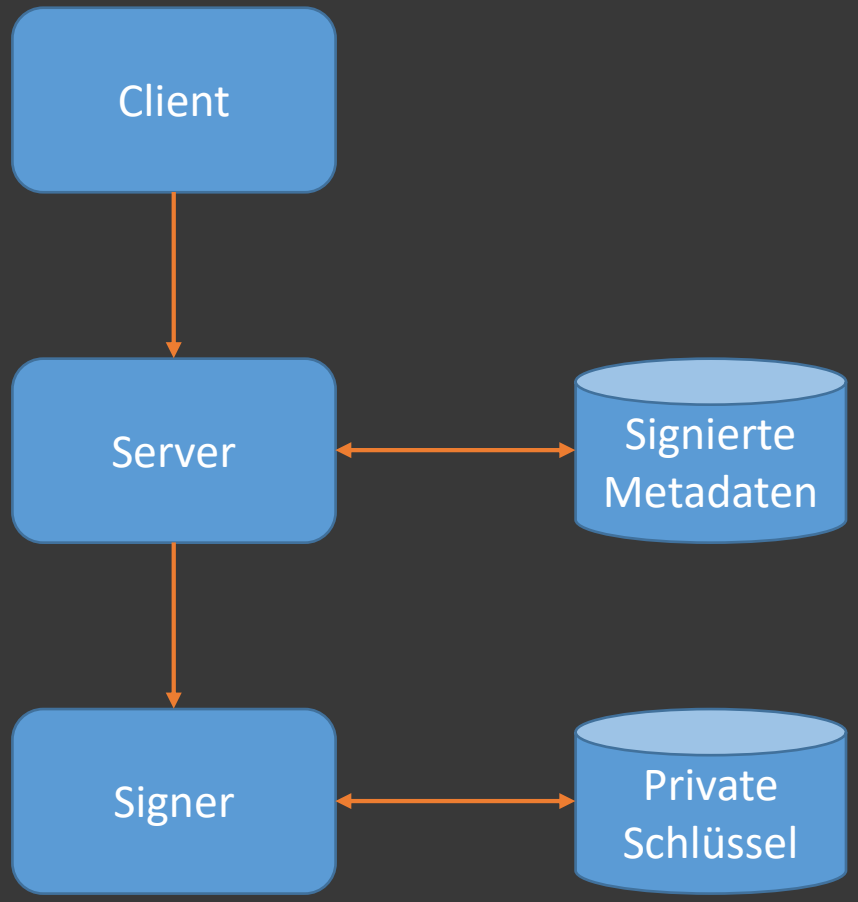
Can sign arbitrary data

Verification of signatures

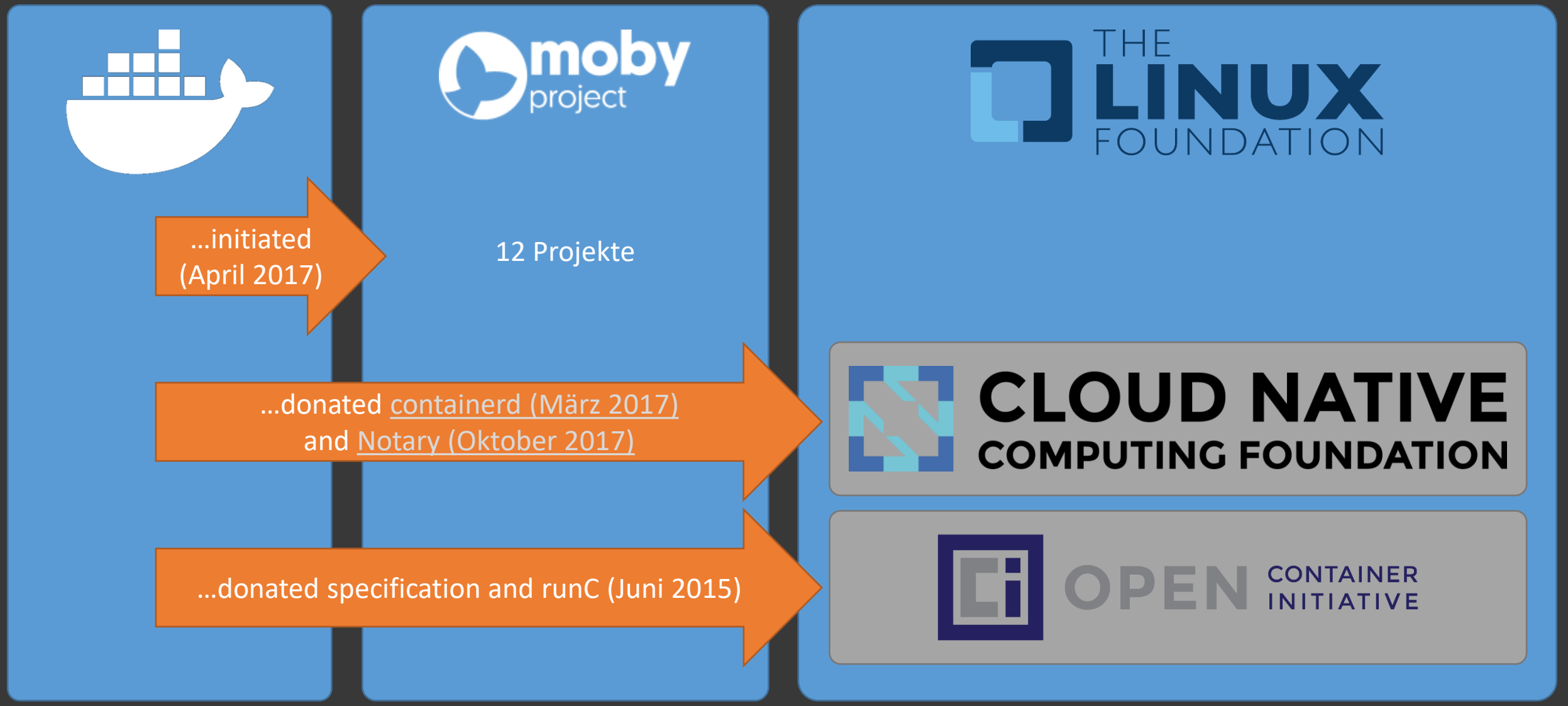
Client/server architecture

Upstream project for Docker Content Trust

Owned by CNCF



Docker and the ecosystem



Docker ❤ Kubernetes

Announced at DockerCon EU (October 2017)

Integrated in Docker Universal Control Plane (UCP)

Manages stacks based on SwarmKit and Kubernetes

Translates docker-compose.yml

Bundled Docker CLI and Kubernetes CLI

Installed on new nodes via Docker UCP

Included in Docker für Mac

Included in Docker für Windows

Docker for Mac/Windows installs Kubernetes (single node)

Offerings by Docker

Enterprise Edition

Universal Control Plane

Trusted Registry

Docker Certified

certified kubernetes 1.8

Community Edition

Docker CLI

dockerd

Docker Hub

Machine

Moby

SwarmKit

containerd

runC

LinuxKit

Compose

Registry

Notary

libnetwork

DataKit

BuildKit

VPNKit

InfraKit

HyperKit

Kubernetes

Opinionated View on the Docker Ecosystem 2018

Containers are commodity

Knowledge is spreading

Increasing need for advanced topics

Deployment options

Containerization is accepted as additional option

Evolution instead of revolution

Pace is slowing

Features are nice-to-have

DockerCon

Docker Enterprise Edition

Federated application management across clouds

Kubernetes on Windows Server

Deployment of cross-platform applications

Deployment on Docker Swarm as well as Kubernetes

Deployment of OSS serverless frameworks

Docker Desktop

Template based image creation

docker-app

XXX see link

New lifecycle for Docker CE

Stable channel

6-month release cycle

Edge channel

Deprecated in favour of nightly build

Nightly build

Access new features

The State of Docker

Transformation

Technology-driven company for over 10 years

Change to focus on enterprise customers

Kubernetes

Embraced in Docker Enterprise Edition

Side-by-side with Docker Swarm

Community

Standardization in Moby project and OCI

Docker 18.09

Remoting using SSH

Third option in addition to...

... Socket

... TCP

BuildKit

Integration of more Moby projects

Updated containerd

Separation is becoming more obvious

Docker Under the Hood

See link

Docker 18.09: Remoting using SSH

So far so good

Remoting using `tcp://<host>:<port>` (use certificates!)

Tunneling the socket using SSH (`ssh -L /tmp/my-docker.sock:/var/run/docker.sock`)

Brave new world

Remoting using `ssh://<user>@<host>`

Docker 18.09 is required on client and server

Under the hood, `ssh` is used to start `docker system dial-stdio`

Docker 18.09: BuildKit

BuildKit is GA

Generic library

Frontend for Dockerfile

Integrated as optional feature

Enabled with (`export DOCKER_BUILDKIT=1`)

Improved build performance

2x from empty state

7x from cache

2.5x with updated source code

9x when using `--cache-from`

Docker 18.09: BuildKit

Demo

$ DOCKER_BUILDKIT=1 docker build .

```
/tmp/test # docker build .
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM alpine
latest: Pulling from library/alpine
4fe2ade4980c: Already exists
Digest: sha256:621c2f39f8133acb8e64023a94dbdf0d5ca81896102b9e57c0dc184cadaf5528
Status: Downloaded newer image for alpine:latest
 ---> 196d12cf6ab1
Step 2/2 : RUN touch /tmp/test
 ---> Running in d9d8725b8959
Removing intermediate container d9d8725b8959
 ---> 61257dbb5c13
Successfully built 61257dbb5c13
/tmp/test # DOCKER_BUILDKIT=1 docker build .
[+] Building 0.0s (6/6) FINISHED
 => [internal] load build definition from Dockerfile                                0.0s
 => => transferring dockerfile: 31B                                                 0.0s
 => [internal] load .dockerignore                                                   0.0s
 => => transferring context: 2B                                                     0.0s
 => [internal] load metadata for docker.io/library/alpine:latest                    0.0s
 => [1/2] FROM docker.io/library/alpine                                             0.0s
 => CACHED [2/2] RUN touch /tmp/test                                                0.0s
 => exporting to image                                                              0.0s
 => => exporting layers                                                             0.0s
 => => writing image sha256:4cb2738ef8af06b34753a2c1758a963ed449880715f30cb0cfcc3b0845e6f079  0.0s
/tmp/test #
```

Docker 18.09: Build Secrets

Part of BuildKit

BuildKit must be enabled

Files can be mounted into `/run/secrets/` during `RUN`

Prevents copying secrets into image layers

Extended syntasx for RUN statement

Enabled by comment in Dockerfile

New mount parameter (`RUN --mount=type=secret,id=mysite.key ...`)

Build requires additional parameter
(`docker build --secret id=mysite.key,src=./mysite.key ...`)

Docker 18.09: SSH Forwarding

Part of BuildKit

BuildKit must be enabled

SSH agent socket can be mounted during `RUN`

Prevents copying secrets into image layers

Extended syntasx for RUN statement

Enabled by comment in Dockerfile

New mount parameter (`RUN --mount=type=ssh ...`)

Build requires additional parameter
(`docker build --ssh default ...`)

Custom SSH socket or private keys are also supported

docker-compose 1.23

New default naming scheme

<project>_<service>_<index>_<slug>

Parallelization

`docker-compose build` will now build up to 5 images in parallel

Remember

Variable overrides in file `.env`

Kubernetes Everywhere

Only supported orchestrator

Products focus on Kubernetes

Example: Rancher 2.x

Integrations

Product feature

Value Add

Automation

Example: GitLab, GoCD and many more

Kubernetes Everywhere

69 Certified Service Providers

47 Distributions

6 local machine solutions

15 hosted solutions

20 turn key solutions

12 on-premises solutions

19 custom solutions

Prominent solutions

Azure Kubernetes Service (AKS)

AWS Elastic Container Service for Kubernetes (EKS)

Google Kubernetes Engine (GKE)

AWS Fargate

Hyper.sh

https://landscape.cncf.io/format=landscape

The State of Docker Swarm

Major efforts

Zero downtime rolling upgrades

Swarm networking for Windows Server 1709/1803

Improved scalability for VIP LB

Resilience on high latency networks

`docker stack` works against k8s as well

Always use Swarm even on single node

Secrets

Healthchecks

Rolling upgrades

Ready for scale-out

Windows Containers

Mapping named pipe

Added in Windows Server 1709

Allows for Docker-out-of-Docker

Linux Container on Windows (LCOW)

Added in Windows Server 1803

Allows for Linux and Windows containers side-by-side

New base image called `windows`

Added in Windows Server 2019

Contains most OS components

(Re-)released on November 13th

Windows Containers

Prediction: Native Linux containers on Windows

Windows Subsystem for Linux (WSL) already isolated Linux

Windows kernel knows how to isolate Windows processes

Lessons Learned

Containers are commodity

Kubernetes is the most popular orchestrator

Docker Swarm is still in active development

Windows is on everyone's roadmap

Windows will run Linux containers natively

Summary

aaa