

Kapitel 3: Virtualisierung



Vorlesung
**CLOUD
COMPUTING**

Anforderungen an Cloud Computing

- Organisation der Ressourcen in einem Rechenzentrum
 - Effiziente Nutzung der gegebenen Ressourcen zur Minimierung der Kosten
 - Isolation der Ressourcen. Kunden sollen andere nicht sehen und auch nicht von ihnen beeinflusst werden. Seiteneffekte vermeiden. Security
 - Entkopplung von der Hardware für mehr Flexibilität im Betrieb und Robustheit bei Ausfällen
 - Ressourcen sollen flexibel vergeben werden. Steuerung mittels Software defined resources
- Lösung für diese Anforderungen ist Virtualisierung und macht Cloud Computing erst möglich.

Virtualisierungsarten

Virtualisierung ist stellvertretend für mehrere grundsätzlich verschiedene Konzepte und Technologien:

- Virtualisierung von Hardware-Infrastruktur
 - Emulation
 - Voll-Virtualisierung
 - Para-Virtualisierung
- Virtualisierung von Software-Infrastruktur
 - Betriebssystem-Virtualisierung (*Containerization*)
 - Anwendungs-Virtualisierung (*Runtime*)



Virtualisierungsarten: Hardwarevirtualisierung

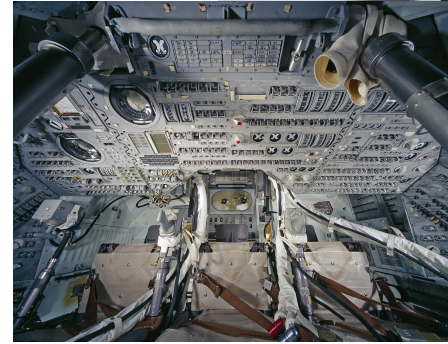
Was wird virtualisiert?

Hardwarevirtualisierung arbeitet auf Ebene der Rechnerarchitektur.

- **Prozessor**
 - Der State des Prozessors. Im wesentlichen Prozessorregister.
 - Maschinencode
 - Memory Management Unit
- **Hauptspeicher**
 - Linear adressierter physikalischer Speicher
- **Netzwerk**
 - z. B. Input-Output Stream von Ethernet Frames
- **Storage**
 - Blockspeicher (linear adressiert. Lesen und Speichern von Blöcken)
- **Grafikkarte**
 - z. B. Framebuffer (2D-Array mit Pixeldaten)
 - 3D Funktionalität (DirectX, OpenGL), siehe https://en.wikipedia.org/wiki/GPU_virtualization
 - Computing (KI, Simulationen) (Zunehmend wichtig für die Cloud)
- Evtl. Peripherie wie USB, Maus, Keyboard
- Timer, Interrupt Controller

Was ist Emulation?

- Emulation: Bildet die Hardware eines nicht vorhandenen oder nicht kompatiblen Rechnersystems oder Teile eines entsprechenden Rechnersystems nach
- Emulationen sind der Regel sehr langsam und nicht parallelisierbar
- Anwendungen
 - Alte Software konservieren. Zum Beispiel alte Spielkonsolen oder Apollo Guidance Computer: <https://svtsim.com/moonjs/agc.html>
 - Embedded Entwicklung ohne echte Hardware
 - Reine CPU Emulation
 - Rosetta von Apple. CPU Instruktionsübersetzung von Power-PC zu x86 mittels dynamic binary translation, also Just in Time Übersetzung. (2004). Gerade wird Rosetta 2 entwickelt für die Intel -> ARM Transition
 - <https://www.heise.de/news/Windows-on-ARM-Testphase-fuer-x64-Emulation-startet-im-November-2020-4918453.html>
 - QEMU User Mode Emulation
 - Voll-Virtualisierer (Hardware und spezielle Instruktionen der CPU)
 - QEMU und Bochs können Windows und Linux praktisch überall starten.

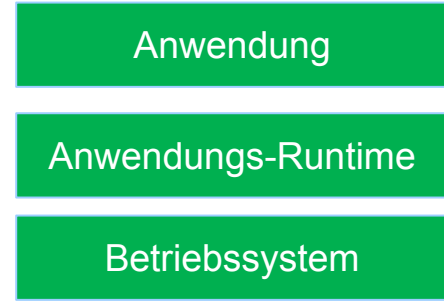


Windows 95 auf der Apple Watch

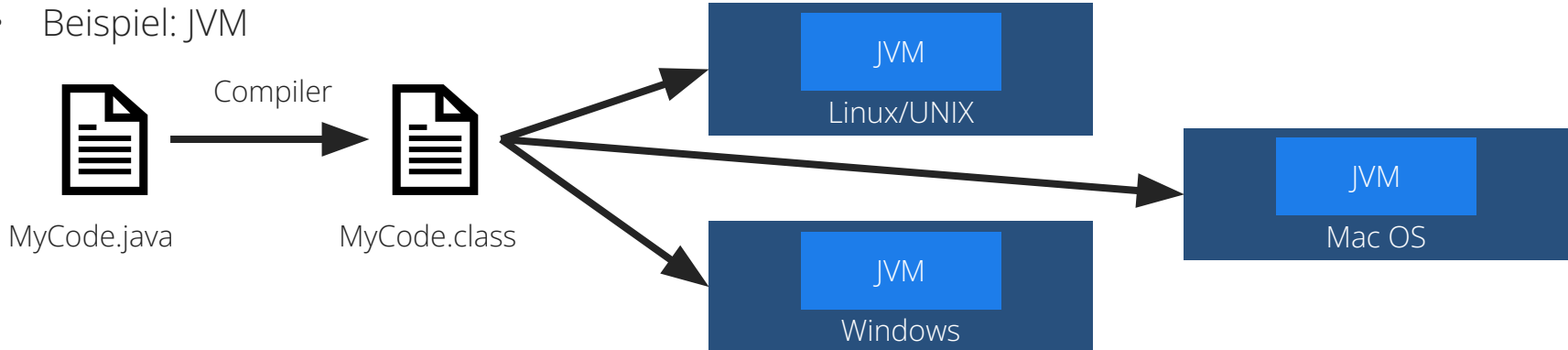


Zur Vollständigkeit: Was ist Anwendungs-Virtualisierung?

- Anwendungs-Virtualisierung: Stellt Anwendungen eine Programmierschnittstelle und eine Laufzeitumgebung (Runtime) zur Verfügung, die komplett vom darunter-liegenden Betriebssystem entkoppelt.
Zweck u.A.: Portable Anwendungen.



- Beispiel: JVM



Virtualisierung, aber performant

- Emulationen erfüllen zwar viele Voraussetzungen für das Cloud Computing wie Isolation und Entkopplung, sind aber bei der Nachbildung einer ganzen Rechnerarchitektur sehr langsam und daher ungeeignet für den massenhaften produktiven Einsatz.
 - Hauptverantwortlicher dabei ist die CPU.
- Kann man die selben Ziele mit minimalem zusätzlichem Ressourcenaufwand erreichen?
 - Antwort Ja, aber nur wenn die Gast-Rechnerarchitektur des virtualisierten Systems die gleiche ist wie die Host-Rechnerarchitektur.
 - x86 Host → x86 Guest

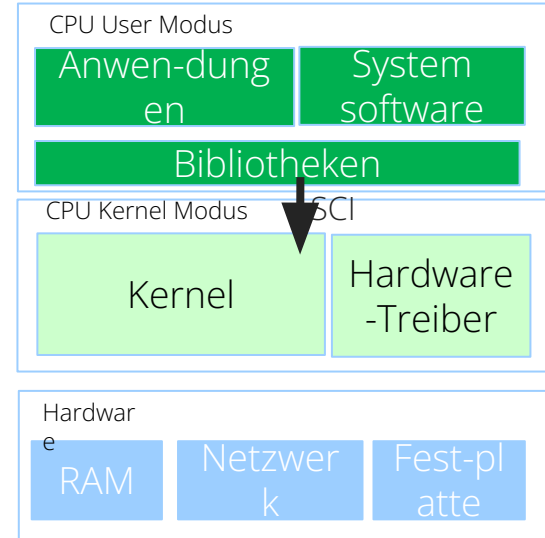
Klassischer Aufbau eines Betriebssystems mit Unterstützung der Rechnerarchitektur

CPU User Mode

- Niedrigste Berechtigungsstufe
- Keine direkten Hardwarezugriffe
- Speicherschutz über die Memory Management Unit

CPU Kernel Mode

- User Mode ruft Kernel über das System Call Interface (SCI) auf. Aktuell besteht das SCI bei Linux aus ca. 380 System Calls.
- Höchste Berechtigungsstufe
- Privileged CPU Instruktionen
- Zugriff auf Hardware mittels Treiber
- Übernimmt z. B. Dateisystemverwaltung und Scheduling der Anwendungen

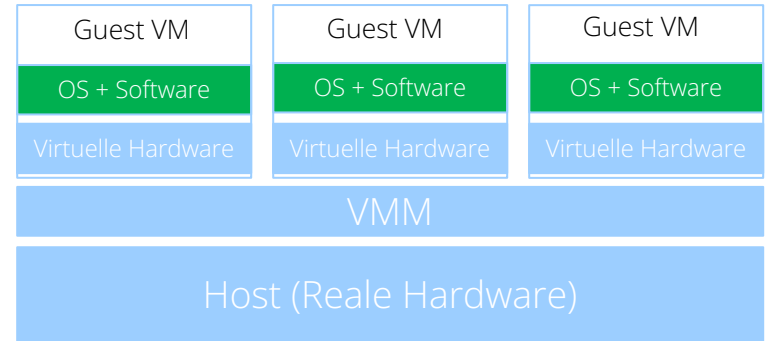


Unterstützung durch die Hardware

- Software based virtualization
 - In dem klassischen Betriebsmodell hat nur der Usermodus die notwendigen Isolationseigenschaften. Der Kernelmodus kann hier emuliert werden. z. B. mittels "trap-and-emulate". mit wenigstens 10% Performanceverlust.
- Hardware assisted virtualization
 - Oftmals wurden daher CPU Extensions entwickelt wie Intel-VT und AMD-V. Diese fügen einen neuen Prozessormodus (z. B. virtual execution mode) hinzu, bei dem sich das Gastbetriebssystem als mit vollen Privilegien arbeitend wahrnimmt, das Hostbetriebssystem jedoch geschützt bleibt
 - Virtuelle Hauptspeicher-Partition im echten physikalischen Speicher. (Die Null verschiebt sich). Management der realen Repräsentation mittels der Management Memory Unit (MMU).
 - Für die Durchreichung (Pass-Through) der Schnittstellen von echten Hardwaregeräten muss die Verschiebung der Null durch eine IOMMU (I/O Memory Management Unit) ausgeglichen werden.

Hardware-Virtualisierung: Begrifflichkeiten

- Durch Hardware-Virtualisierung werden die Ressourcen eines Rechnersystems aufgeteilt und von mehreren unabhängigen Betriebssystem-Instanzen genutzt.
- Anforderungen der Gastinstanzen werden von der Virtualisierungssoftware (Virtual Machine Monitor, VMM) abgefangen und auf die real vorhandene Hardware umgesetzt.
- Der VMM (oder Hypervisor) verteilt die Hardwareressourcen des Rechners an die VMs
- Es werden aber 2 Virtualisierungsmodi und 2 Arten von Hypervisor unterschieden



Host

- Der Rechner der eine oder mehrere virtuelle Maschinen ausführt und die dafür notwendigen Hardware-Ressourcen zur Verfügung stellt.

Guest

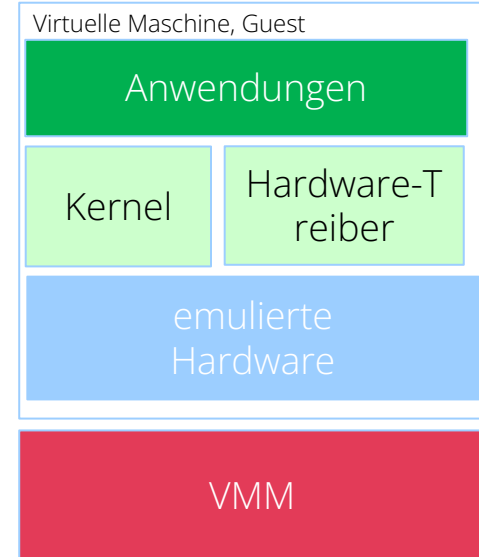
- Eine lauffähige / laufende virtuelle Maschine

VMM (Virtual Machine Monitor, auch Hypervisor genannt)

- Die Steuerungssoftware zur Verwaltung der Guests und der Host-Ressourcen

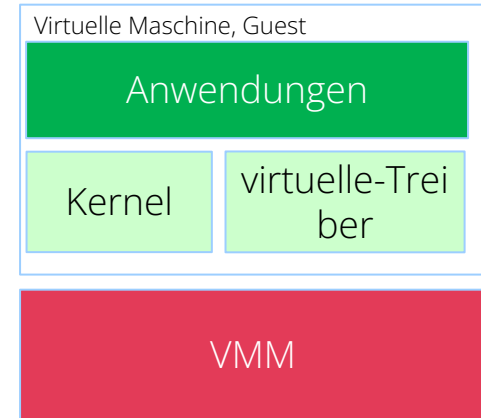
Voll-Virtualisierung

- Jedem Gastbetriebssystem steht ein eigener virtueller Rechner mit virtuellen Ressourcen wie CPU, Hauptspeicher, Laufwerken, Netzwerkkarten, usw. zur Verfügung.
- Das Gastbetriebssystem muss also nicht angepasst werden. Zum Zeitpunkt des Starts muss das Gastbetriebssystem nicht bekannt sein.
- Die VMM emuliert auch weiterhin echte Hardware wie Storage (SATA) und Netzwerk (Ethernet).
- Die VMM kann aber zur Beschleunigung oder zur besseren Nutzung (Grafik, Mouse) spezielle virtuelle Hardware zur Verfügung stellen.
 - z. B. Einfacher Pass-Through von USB
 - Fließender Übergang zur Paravirtualisierung
- Leistungsverlust: 1 - 5%



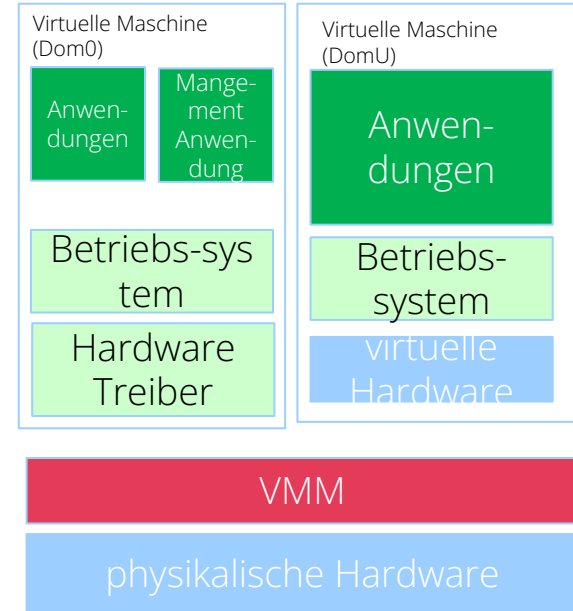
Para-Virtualisierung

- Dem Gast-Betriebssystem stehen keine direkt low-level virtualisierten Hardware-Ressourcen zur Verfügung sondern eine API.
 - Vereinfacht den Aufbau der VM
- Das Gast-Betriebssystem muss portiert werden.
 - Low Level Prozessorinstruktionen werden erst gar nicht ausgeführt oder durch API Aufrufe abgebildet
 - Virtuelle Treiber (z. B. virtio).
 - Vermeidung von Umformungen und Kopieraktionen durch Verwendung spezieller Treiber
 - Übertragung von IP Paketen und nicht von Ethernet Frames.
- Unterstützte Betriebssysteme und Hardware-Varianten aus Sicht des Gastes eingeschränkt pro Hypervisor-Implementierung.
- Leistungsverlust: 0 - 2%



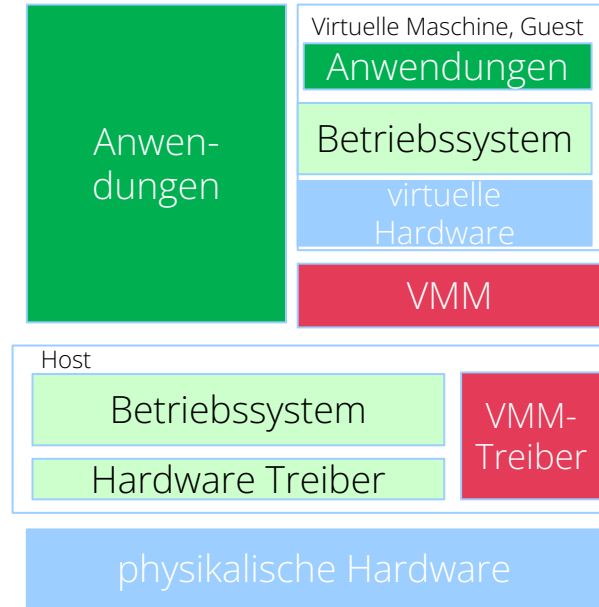
Typ 1: Bare-Metal Virtualisierung

- Der Hypervisor läuft direkt auf der verfügbaren Hardware. Er entspricht somit einem Betriebssystem, das ausschließlich auf Virtualisierung ausgerichtet ist.
- Der Hypervisor nutzt üblicherweise die Treiber eines Host-Betriebssystems, um auf die reale Hardware zuzugreifen. Damit brauchen im Hypervisor nicht aufwändig eigene Treiber implementiert werden.
- Vornehmlich mit Paravirtualisierung, da hier ebenfalls die Emulation der Hardware entfällt
- Ermöglicht einfacheren Pass-Through von echter Hardware. z. B. GPUs an einen Gast
- Beispiele: VMWare ESXi, Microsoft Hyper-V, XEN



Typ 2: Host Virtualisierung

- Der VMM läuft hosted als Anwendung unter dem Host-Betriebssystem
- Vornehmlich bei Voll-Virtualisierung verwendet
- Geringere Skalierbarkeit wegen Abhängigkeit zum Host System
- Mehr Overhead als Typ 1
- Beispiele:
 - Virtualbox
 - VMWare Workstation Player
 - Parallels
 - Achtung: Die Unterscheidung zwischen Typ 1 und Typ 2 ist in vielen Fällen nicht immer klar.



Virtualisierung im Enterprise Umfeld

Neben den bisher genannten Vorteilen bieten heutige VM Lösungen noch viele weitere Features

- Ressourcenverwaltung im laufenden Betrieb
 - Memory Ballooning – Hauptspeicher dynamisch vergrößern
 - Änderung der Anzahl an virtuellen Rechenkernen
 - Änderung der Festplattengröße mit virtuellen SANs (Storage Area Networks)
- Live Migration
 - Verschieben der laufenden physikalischen Maschine auf eine andere Hardware innerhalb von Millisekunden
 - CPU State
 - Speicher
 - Storage
 - Netzwerk
- (Echten) Zufall zu erzeugen ist in einer VM noch schwieriger als mit realer Hardware (z. B. mittels Maus und Tastatureingaben). Hier bieten die Hypervisors Schnittstellen an um zusätzlichen Zufall zu erhalten.



Hardware-Virtualisierung mit Vagrant und VirtualBox

Hardware-Virtualisierung: Vagrant und VirtualBox



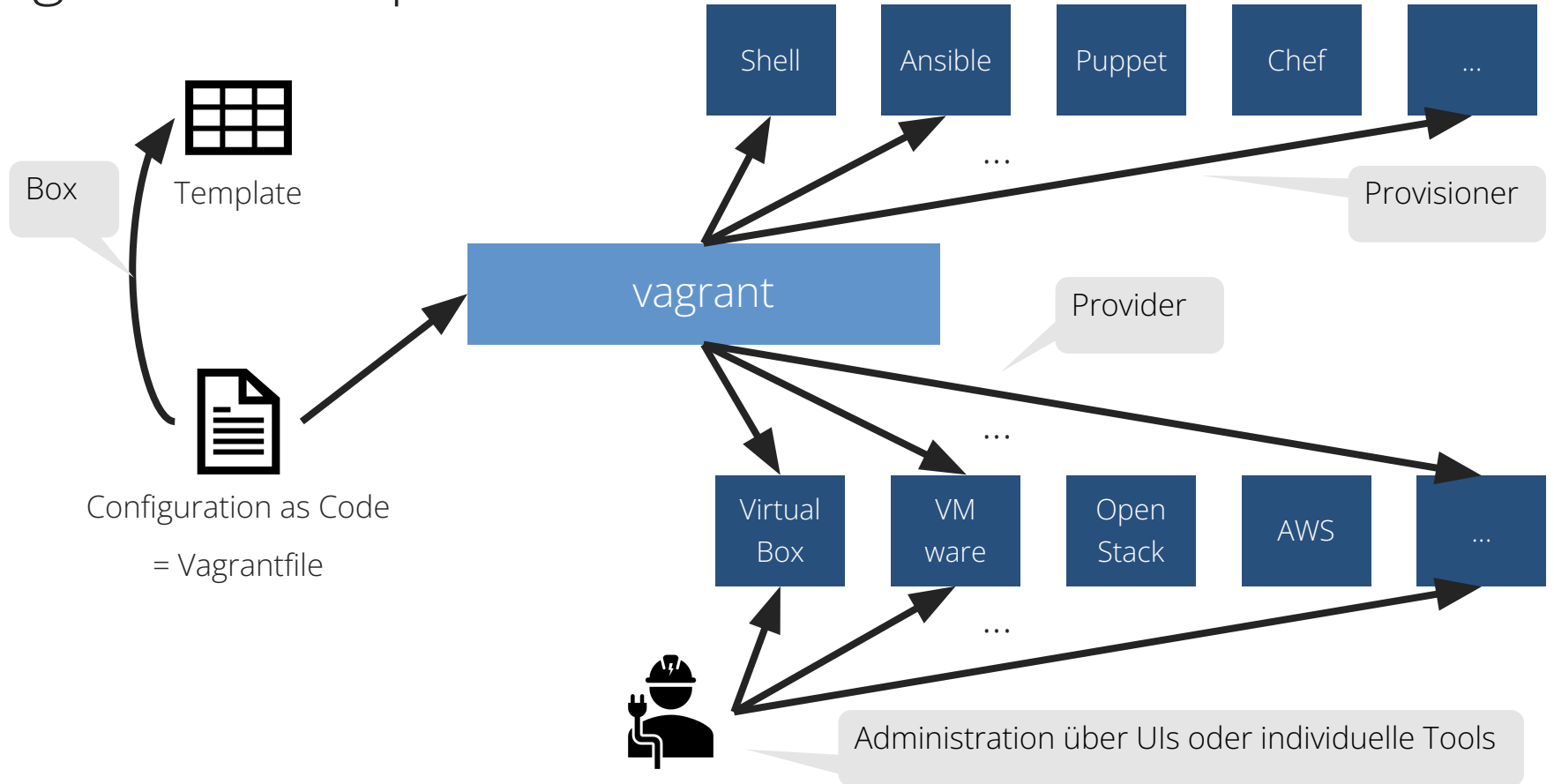
+



Open Source Typ 2 Virtualisierungs-Software (Voll-Virtualisierung) für Windows, Linux, MacOS und Solaris.

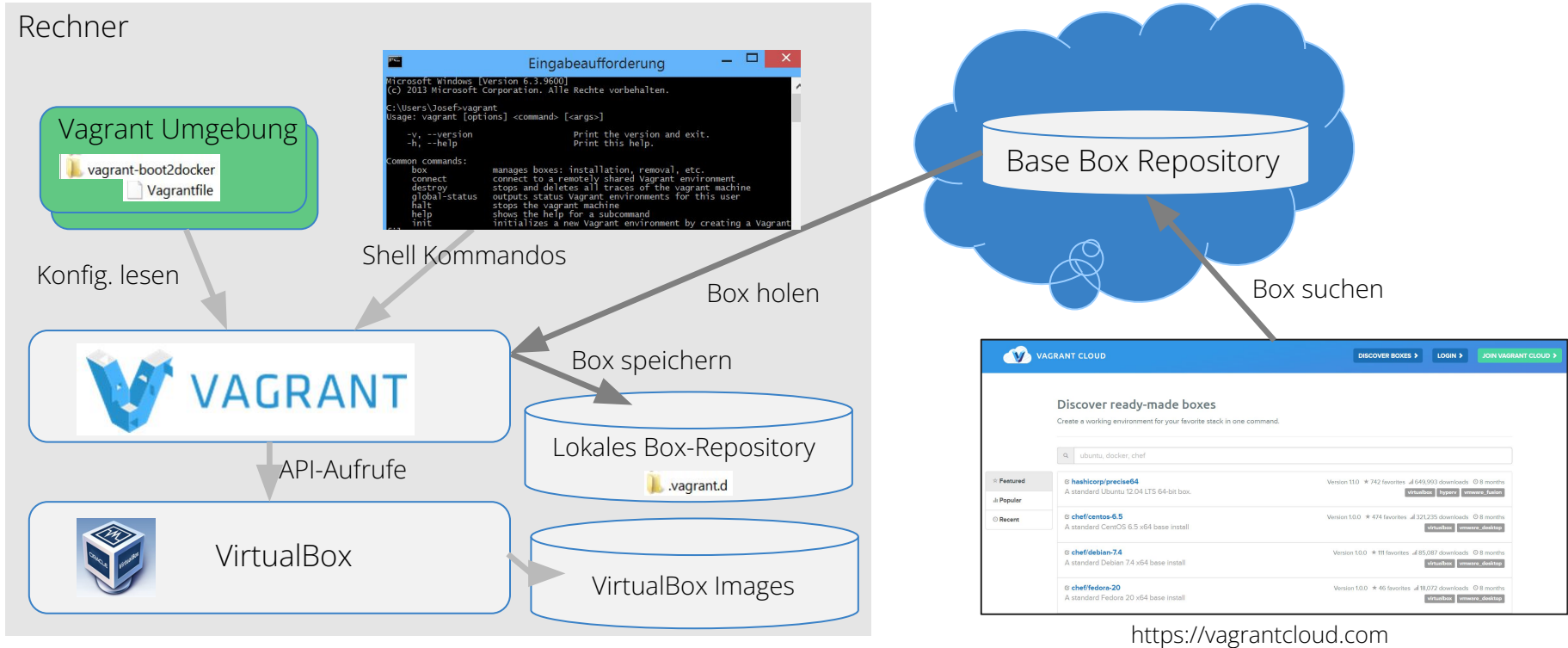
Automationssoftware für virtuelle Umgebungen auf einem Rechner. Virtuelle Maschinen per Kommandozeile erstellen und steuern.

Vagrant: Konzepte



Demo

Vagrant: Eine schematische Übersicht.



Das Vagrantfile beschreibt die zu erstellende virtuelle Maschine.

```
# -*- mode: ruby -*-  
# vi: set ft=ruby :
```

Vagrantfiles werden in Ruby geschrieben

```
# Vagrantfile API/syntax version. Don't touch unless you know what you're doing!  
VAGRANTFILE_API_VERSION = "2"
```

```
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
```

```
  # My base box  
  config.vm.box = "chef/ubuntu-14.04"
```

Definition der Basis-Box

```
  # Define shell provisioning  
  config.vm.provision :shell, path: "bootstrap.sh"
```

Konfiguration der Provisionierung

```
  # Define docker provisioning  
  config.vm.provision "docker" do |d|  
    d.run "nginx1", image: "dockerfile/nginx", args: "-p 8080:80", daemonize: true  
    d.run "nginx2", image: "dockerfile/nginx", args: "-p 9080:80", daemonize: true  
    d.run "haproxy", image: "dockerfile/haproxy", args: "-p 80:80 --link nginx1:nginx1 --link nginx2:nginx2 -v /vagrant:/haproxy-override"  
  end
```

```
  # Configure VirtualBox  
  config.vm.provider "virtualbox" do |v|  
    v.memory = 1024  
    v.cpus = 4  
  end
```

Konfiguration des Virtualisierungs-Providers

```
  # Forward ports  
  config.vm.network :forwarded_port, host: 80, guest: 80  
  config.vm.network :forwarded_port, host: 8080, guest: 8080  
  config.vm.network :forwarded_port, host: 9080, guest: 9080
```

Konfiguration des Netzwerks

```
end
```

Ein typischer Arbeitsablauf mit Vagrant.

#	Befehle auf Kommandozeile	Bedeutung
1	<code>mkdir <box-dir></code> <code>cd <box-dir></code>	Verzeichnis für Vagrant Umgebung erstellen und dorthin wechseln
2	<code>vagrant init [<box-name>] [<box-url>]</code>	Eine Vagrant Umgebung initialisieren. Dabei wird zunächst nur eine Datei <i>Vagrantfile</i> erstellt und initial mit dem Namen und der URL der Box (falls angegeben) initialisiert.
3		Vagrantfile anpassen nach Bedarf (z.B. IP vergeben, Port-Mapping zwischen Host und Guest, Verzeichnis-Share zwischen Host und Guest, ...)
4	<code>vagrant up</code>	Startet die virtuelle Maschine (Box → virtuelle Maschine) und konfiguriert sie entsprechend dem Vagrantfile
5	<code>vagrant ssh</code>	Per SSH auf die virtuelle Maschine verbinden
6	<code>exit</code>	Die SSH Kommandozeile in der virtuellen Maschine verlassen
7	<code>vagrant halt</code>	Die virtuelle Maschine stoppen

Weitere nützliche Kommandos:

- `reload`: Startet eine VM neu und aktualisiert die Konfiguration entsprechend dem Vagrantfile
- `package`: Erstellt aus einer virtuellen Maschine wieder eine Box

Weitere Kommandos: <http://docs.vagrantup.com/v2/cli/index.html>

Vagrant Befehle auf Kommandozeile

- vagrant box add – allows you to install a box (or VM) to the local machine
- vagrant box remove – removes a box from the local machine
- vagrant box list – lists the locally installed Vagrant boxes
- vagrant init – initializes a project to use Vagrant
- vagrant up – starts up the vagrant VM
- vagrant suspend – saves the state of the current VM.
- vagrant resume – will load up the suspended VM.
- vagrant halt – will shut down the VM, saving configuration. (restart with 'up' command)
- vagrant destroy – will destroy the VM with all config changes.
- vagrant reload – apply Vagrant configuration changes (like port forwarding) without rebuilding the VM.
- vagrant status – tells you the current state of the Vagrant project's VM
- vagrant gem – install Vagrant plugins via RubyGems
- vagrant ssh – short cut to SSH into the running VM
- vagrant package – create a distribution of the VM you have running.
- vagrant <command> -help - Command that will provide man pages for a vagrant command.



Virtualisierungsarten: Betriebssystemvirtualisierung

Hardwarevirtualisierer sind Schwergewichte

- Jede VM inkludiert eine virtuelle Kopie eines kompletten Betriebssystems und benötigt signifikante RAM und CPU Ressourcen, die nur schwer dynamisch geändert werden können
- Softwareentwicklung mit VMs ist träger und komplexer
- Aufgrund der Größe der Images ist die Portabilität ein Problem.
- Kompatibilität mit anderen VM Lösungen nicht vorhanden. Wechsel zwischen Rechenzentren nicht einfach möglich.

Demo

Der Urgroßvater: chroot (Jahr 1982)

- chroot gilt als Urgroßvater der Betriebssystemvirtualisierung (Jahr 1982)
- chroot setzt aus Sicht der laufenden Applikation das root Filesystem neu
 - Ermöglicht die Isolation des Filesystems
- Kein Overhead. Implementiert in 2 dutzend Zeilen C-Code im Kernel
- Benötigt root-Rechte
- Keine Netzwerk-Isolation, keine Prozess-Isolation, keine Disk Quotas, keine CPU Quotas, keine I/O Limitierung
- chroot Prozess sieht weiterhin fast alles vom System

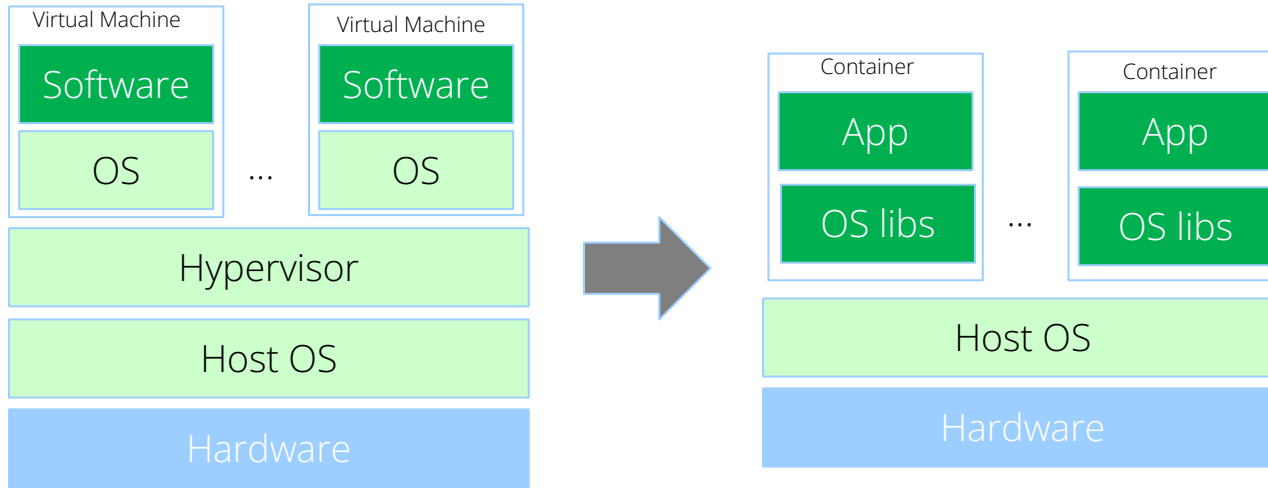
Linux Kernel Namespaces (Jahr 2002)

- Isolation durch Sichtbarkeit
- Ein Feature des Linux-Kernels, das die Sicht und den Zugriff auf das System einschränkt:
 - Prozessraum / Prozess-Ids
 - Netzwerk-Schnittstellen
 - Host-Name
 - Dateisystem-Mounts
 - IPC (Inter-Prozess-Kommunikation)
 - Benutzerkonten
 - Zeit
- Die Einschränkungen sind dabei für den isolierten Prozess transparent.
- Namespaces können geschachtelt sein.
- siehe [https://success.docker.com/KBase/Introduction to User Namespaces in Docker Engine](https://success.docker.com/KBase/Introduction%20to%20User%20Namespaces%20in%20Docker%20Engine)

Linux cgroups (Jahr 2007)

- Isolation durch Grenzen
- Ein Feature des Linux-Kernels, das maßgeblich durch Google entwickelt wurde
- Gruppiert Prozesse zu Gemeinschaften mit definiertem und beschränktem Ressourcen-Zugriff auf:
 - Prozessor
 - Hauptspeicher
 - I/O (insb. Netzwerk)
 - Disk
- Die Prozess-Gruppen können geschachtelt sein.
- cgroups stellen dabei für die Prozessgruppen sicher, dass
 - die Ressourcen limitiert sind und die definierten Grenzen nicht überschritten werden
 - die aktuell verbrauchten Ressourcen kontinuierlich gemessen und protokolliert werden
 - dass bei Überschreitung der definierten Grenzen die Prozess-Gruppen eingefroren und neu gestartet werden
- Siehe <https://docs.docker.com/engine/docker-overview/>

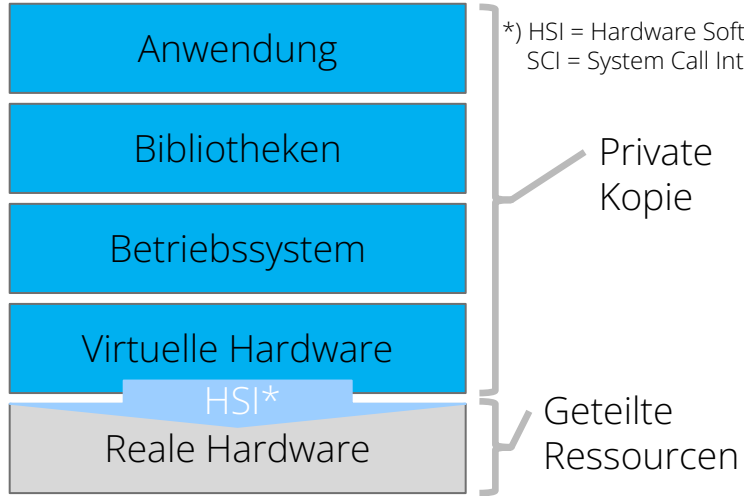
Betriebssystem-Virtualisierung



- Leichtgewichtiger Virtualisierungsansatz: Es gibt keinen Hypervisor. Jede App läuft direkt als Prozess im Host-Betriebssystem. Dieser ist jedoch maximal durch entsprechende OS-Mechanismen isoliert (z.B. Linux LXC).
- Isolation des Prozesses durch Kernel Namespaces (bzgl. CPU, RAM und Disk I/O) und Containments
- Isoliertes Dateisystem
- Eigene Netzwerk-Schnittstelle
- CPU- / RAM-Overhead in der Regel nicht messbar (~ 0%)
- Startup-Zeit = Startdauer für den ersten Prozess

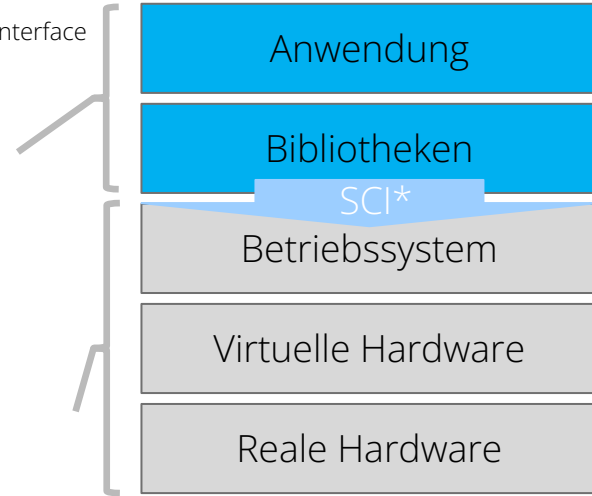
Hardware- vs. Betriebssystem-Virtualisierung

Hardware-Virtualisierung



- Benötigt Hardwareunterstützung
- Höhere Sicherheit. Die HSIs sind einfach.
- Stärkere Isolation.
- Hohes Volumen, Hohe Startzeit
- Unterschiedliche Betriebssysteme

Betriebssystem-(OS-)Virtualisierung



- Ist eine reine Softwarelösung
- Geringere Sicherheit: System Call Interface ist sehr mächtig und komplex
- Geringeres Volumen, Geringerer Overhead, Kürzere Startup-Zeit
- Betriebssystem fest

Kontainerisierung ist angekommen!

Google Runs All Software In Containers

May 28, 2014 by Timothy Prickett Morgan



The overhead of full-on server virtualization is too much for a lot of hyperscale datacenter operators as well as their peers (some might say rivals) in the supercomputing arena. But the ease of management and resource allocation control that comes from virtualization are hard to resist and this has fomented a third option between bare metal and server virtualization. It is called containerization and Google recently gave a glimpse into how it is using containers at scale on its internal infrastructure as well as on its public cloud.

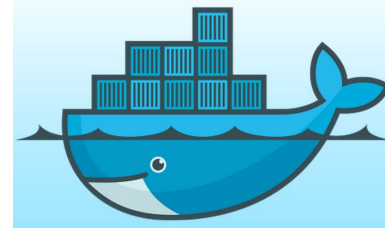
We are talking about billions of containers being fired up a week here, just so you get a sense of the scale.

Beispielhafte Technologien

Hardware-Virtualisierung:
Vagrant und VirtualBox



Betriebssystem-Virtualisierung:
Docker



Betriebssystem-Virtualisierung mit Docker (2013)

Containerization mit Docker: Standardisierung



<http://www.srf.ch/kultur/im-fokus/brasilien/favelas-im-wandel-die-armen-muessen-weichen>



Standard format for operations: start, stop, configure, wire, debug + software logistics.

Docker

- Docker ist eine Automationsumgebung für Betriebssystem-Virtualisierung.
- Aktuell unterstützt Docker Linux als Host-Betriebssystem.
 - Seit 2016 steht eine Windows-Variante zur Verfügung, die mit Hyper-V (Typ 1) virtualisierung läuft.
 - Seit 2020 steht mit WSL2 (Windows Subsystem for Linux) auch eine parallel laufende Linux-Kernel zur Verfügung auf dem Docker ausgeführt werden kann.
- Docker ist als Werkzeug eines Cloud-Anbieters entstanden und ist mittlerweile eines der sichtbarsten und aktivsten Open-Source-Ökosysteme.

In a Nutshell, docker...

... has had 228,212 commits made by 5,912 contributors
representing 10,917,944 lines of code

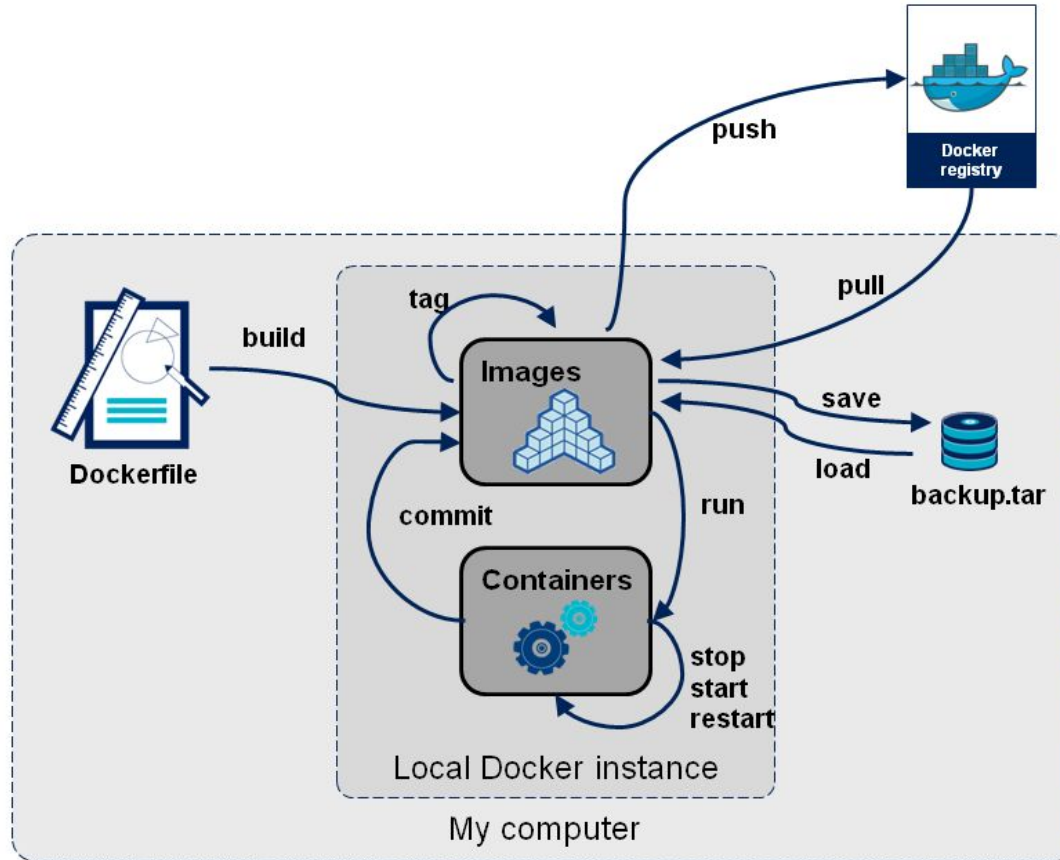
... is mostly written in Go
with an average number of source code comments

... has a well established, mature codebase
maintained by a very large development team
with decreasing Y-O-Y commits

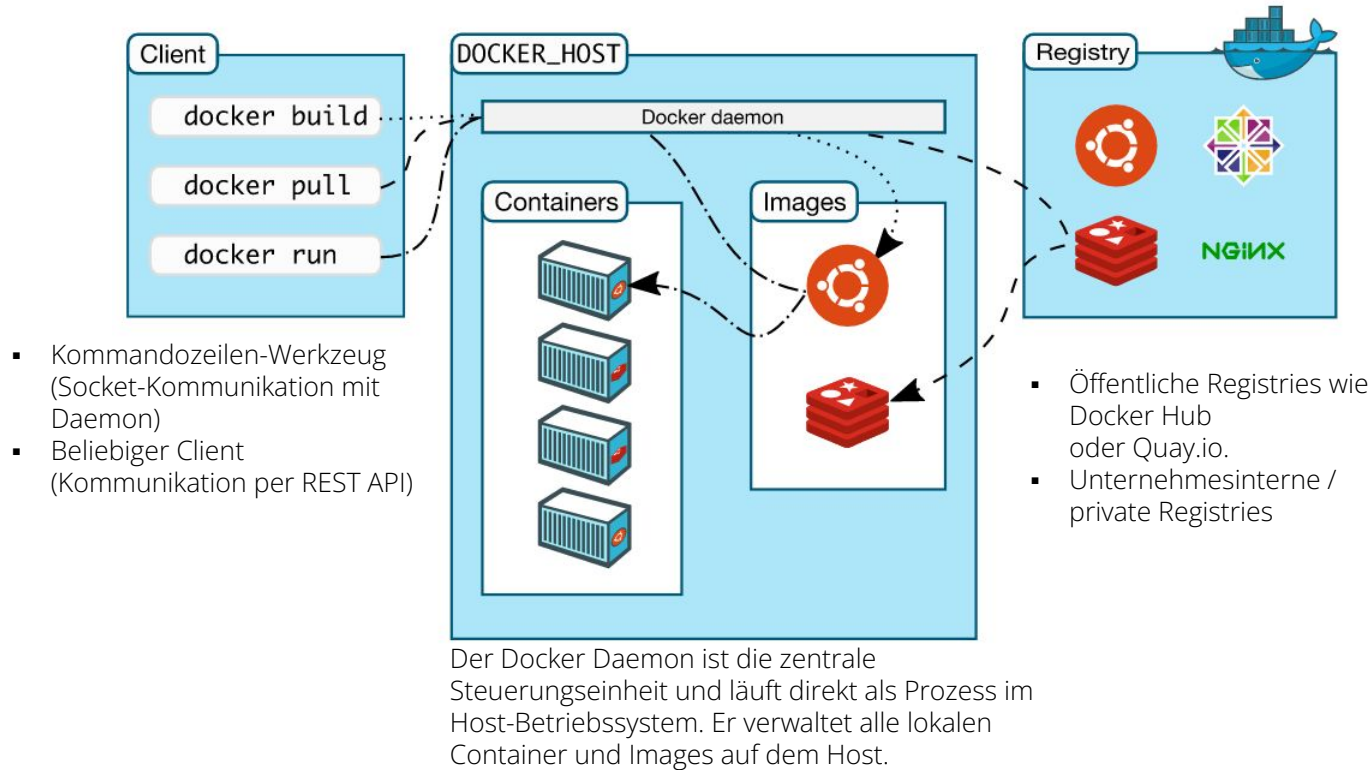
... took an estimated 3,459 years of effort (COCOMO
model)
starting with its first commit in January, 2012
ending with its most recent commit 4 months ago

<https://www.openhub.net/p/docker>

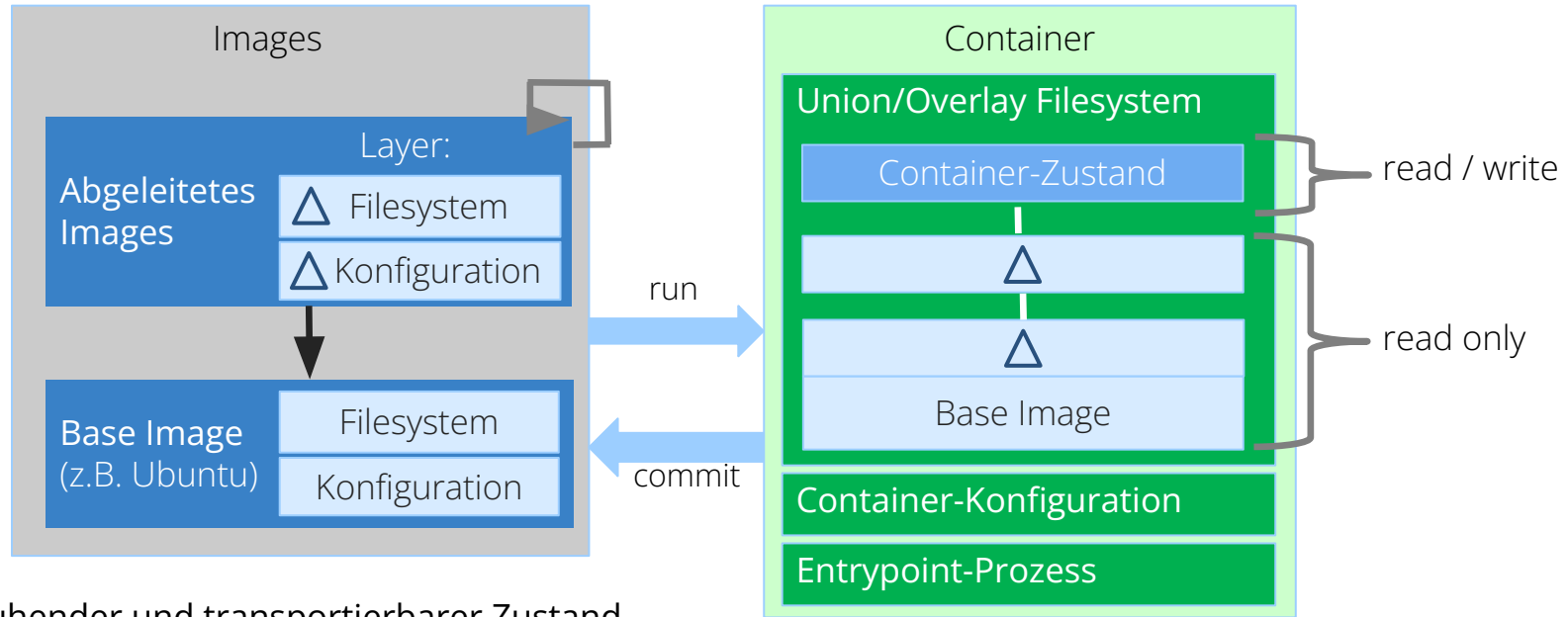
Der Docker Workflow.



Die Docker Architektur.



Im Zentrum von Docker stehen Images und Container.




Ruhender und transportierbarer Zustand

- Ein Image basiert i.d.R. auf einem anderen Image und speichert nur das Delta Δ zu diesem Image.
- Ausnahme: Das Base-Image

Laufender Zustand

- Ein Container läuft so lange wie sein Entrypoint-Prozess im Vordergrund läuft. Docker merkt sich den Container-Zustand.







hub.docker.com ist die öffentliche Standard-Registry für Docker Images.

 Explore Help

Q solr Sign up Log In

Repositories (555)

All

 solr official	280 STARS	100K+ PULLS	> DETAILS
 anapsix/solr public automated build	1 STARS	8.0K PULLS	> DETAILS
 apopelo/solr public automated build	1 STARS	1.6K PULLS	> DETAILS
 ckan/solr public automated build	1 STARS	896 PULLS	> DETAILS
 harisekhon/solr public automated build	0 STARS	6.6K PULLS	> DETAILS
 makuk66/docker-solr public automated build	81 STARS	50K+ PULLS	> DETAILS

Provisionierung von Images mit dem Dockerfile

Ein Dockerfile erzeugt auf Basis eines anderen Images ein neues Images. Dabei werden die folgenden Aktionen automatisiert:

- Konfiguration des Images und der daraus resultierenden Container
- Ausführung von Provisionierungs-Aktionen

Ein Dockerfile ist somit eine Image-Repräsentation alternativ zu einem physischen Image (Bauanteilung vs. Bauteil).

- Wiederholbarkeit beim Bau von Containern
- Automatisierte Erzeugung von Images ohne diese verteilen zu müssen
- Flexibilität bei der Konfiguration und bei den benutzten Software-Versionen
- Einfache Syntax und damit einfach einsetzbar

Befehl: `docker build -t <ziel_image_name> <Dockerfile>`

Das Dockerfile definiert Aufbau und Inhalt des Image.

My Image

Layer 8

Layer 7

Layer 6

Layer 5

Layer 4

Layer 3

Layer 2

Layer 1

Niemals „latest“ verwenden. Antipattern

FROM qaware/alpine-k8s-ibmjava8:8.0-3.10

LABEL maintainer="QAware GmbH <qaware-oss@qaware.de>"

RUN mkdir -p /app

COPY build/libs/zwitscher-service-1.0.1.jar /app/zwitscher-service.jar

COPY src/main/docker/zwitscher-service.conf /app/

ENV JAVA_OPTS -Xmx256m

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "/app/zwitscher-service.jar"]

Dockerfile Kommandos

Element	Meaning
FROM <image-name>	Sets to base image (where the new image is derived from)
MAINTAINER <author>	Document author
RUN <command>	Execute a shell command and commit the result as a new image layer (!)
ADD <src> <dest>	Copy a file into the containers. <src> can also be an URL. If <src> refers to a TAR-file, then this file automatically gets un-tared.
VOLUME <container-dir> <host-dir>	Mounts a host directory into the container.
ENV <key> <value>	Sets an environment variable. This environment variable can be overwritten at container start with the <code>-e</code> command line parameter of docker run .
ENTRYPOINT <command>	The process to be started at container startup
CMD <command>	Parameters to the entrypoint process if no parameters are passed with docker run
WORKDIR <dir>	Sets the working dir for all following commands
EXPOSE <port>	Informs Docker that a container listens on a specific port and this port should be exposed to other containers
USER <name>	Sets the user for all container commands

Typische Kommandos eines Docker Workflows

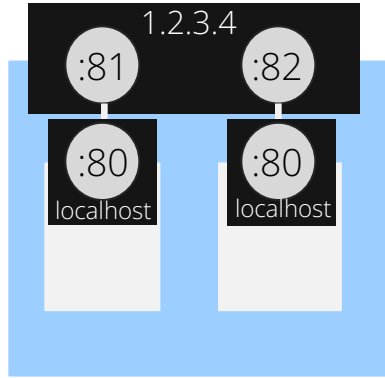
Command	Action
<code>docker build -t <image> .</code>	Build Docker image from „Dockerfile“ with given tag in current directory
<code>docker images</code>	Prints all local images
<code>docker run</code> <code> -d</code> <code> -v <volume mounts></code> <code> -p <host-port>:<container-port></code> <code> <image> <entrypoint process></code>	Run a Docker image: Creates and runs a container. <ul style="list-style-type: none">▪ in background▪ with host directory mounted into the container▪ with port forwarding from host to container▪ image name (and optional entrypoint process)
<code>docker run</code> <code> -ti</code> <code> <image> /bin/sh</code>	Run a Docker image and open a shell within the container <ul style="list-style-type: none">▪ ... with forwarding of local terminal▪ Image name and shell (or „/bin/bash“)
<code>docker ps -a</code>	Prints all containers (without -a = only running containers)
<code>docker commit <container> qaware/foo</code>	Store container as local image
<code>docker kill <container></code> <code>docker rm <container></code>	Terminate container (send SIGKILL to entrypoint process) Remove container
<code>docker rmi -f <image></code>	Remove local image

Hilfreiche Kommandos für Container Troubleshooting

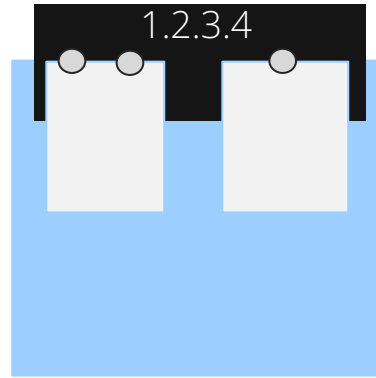
Command	Action
<code>docker inspect <container></code>	Shows container metadata (e.g. IP)
<code>docker logs <container></code>	Prints container syslog
<code>docker top <container></code>	Prints all running processes within a container (like <code>ps -a</code> within the container)
<code>docker exec -ti <container> /bin/sh</code>	Connect terminal to running container
<code>docker stats <container></code>	Shows container runtime statistics (e.g. CPU usage, IO intensity, ...)
<code>docker system prune</code>	Removes all stopped containers, all unused images and all unused volumes
<code>docker history <image></code>	Show the Dockerfile commands for each image layer

Die Docker Networking Modes.

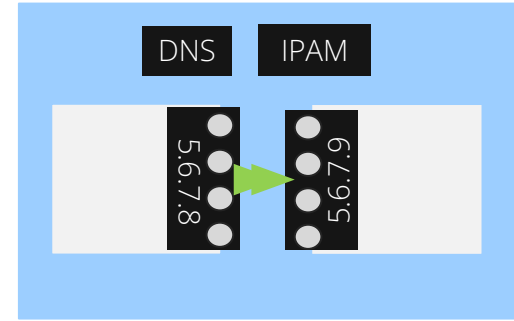
- Docker erlaubt ein getrenntes Netzwerk zwischen docker containern aufzubauen.



Bridge



Host



Overlay Network

`docker network ls`

`docker network inspect bridge`

`docker network create --driver overlay multi-host-network`

`docker network connect multi-host-network container1`

○ Bound port

■ Network interface

□ Guest

■ Host

Anhang

