

Kapitel 4: Provisionierung



QA|WARE

Kurze Wiederholung: Virtualisierung

Virtualisierung

Virtualisierung: die Erzeugung von virtuellen Realitäten und deren Abbildung auf die physikalische Realität.

Zweck:

Multiplizität → Erzeugung mehrerer virtueller Realitäten innerhalb einer physikalischen Realität

Entkopplung → Bindung und Abhängigkeit zur Realität auflösen

Isolation → Physikalische Seiteneffekte zwischen den virtuellen Realitäten vermeiden



<http://www.techfak.uni-bielefeld.de>

Virtualisierungsarten



Virtualisierung ist stellvertretend für mehrere grundsätzlich verschiedene Konzepte und Technologien:

- Virtualisierung von Hardware-Infrastruktur
 - Emulation
 - Voll-Virtualisierung (Typ-2 Virtualisierung)
 - Para-Virtualisierung (Typ-1 Virtualisierung)
- Virtualisierung von Software-Infrastruktur
 - Betriebssystem-Virtualisierung (Containerization)
 - Anwendungs-Virtualisierung (Runtime)

Virtualisierungsarten



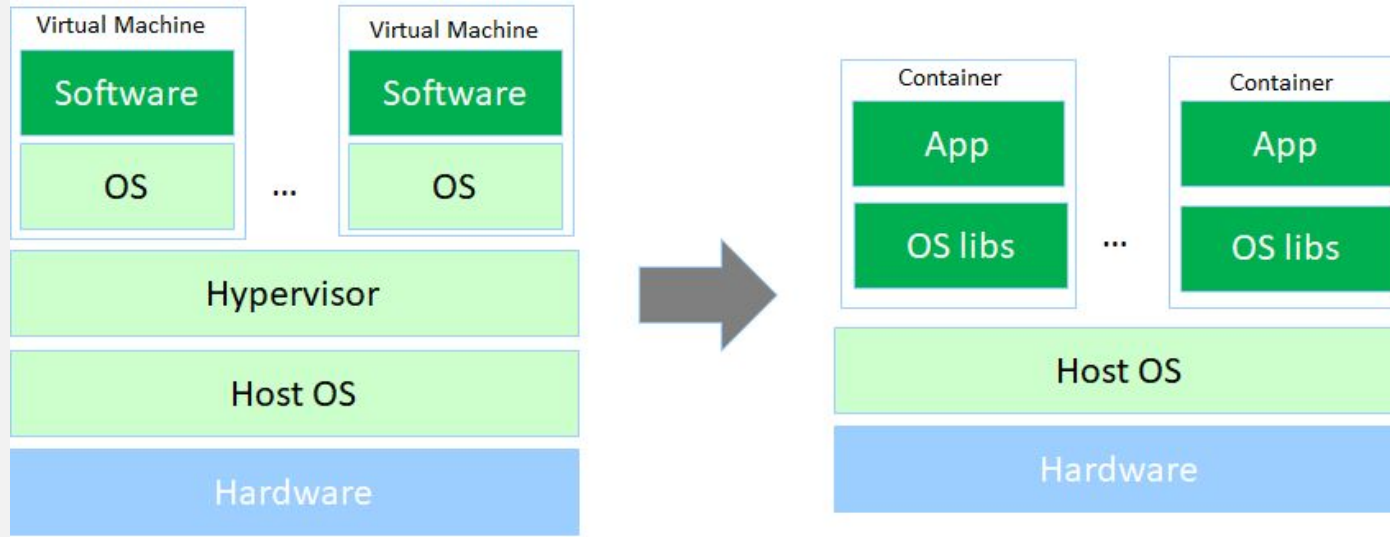
Das schafft Grundlagen für das Cloud Computing:

- Entkopplung von der Hardware für mehr Flexibilität im Betrieb und Robustheit bei Ausfällen
- Normierung von Ressourcen-Kapazitäten auf heterogener und wechselnder Hardware
- Zentrale Steuerung und Bereitstellung von Rechenressourcen über die mit Virtualisierung bereitgestellten Software-Defined-Resources

Betriebssystem-Virtualisierung / Containerisierung



QA|WARE



- 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

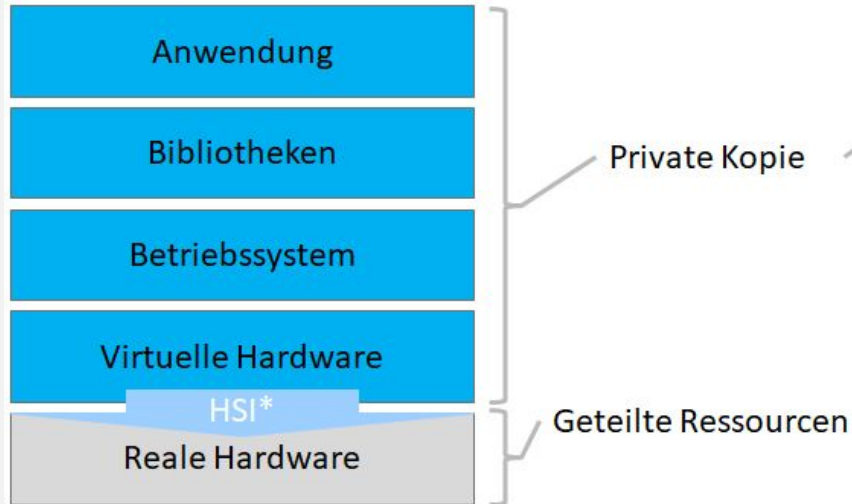
Hardware- vs. Betriebssystem-Virtualisierung



QA|WARE

*) HSI = Hardware Software Interface
SCI = System Call Interface

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



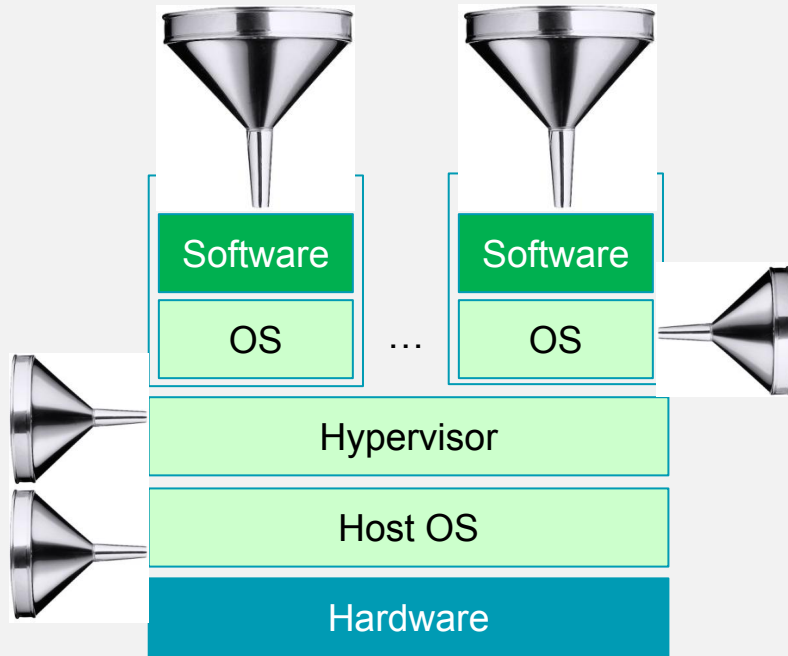
QA|WARE

Provisionierung

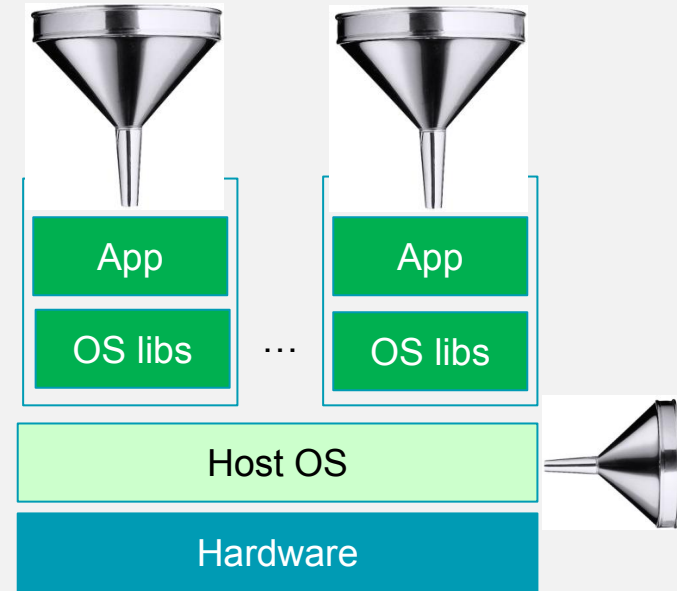
Provisionierung: Wie kommt Software in die Boxen?



QA|WARE



Hardware-Virtualisierung



Betriebssystem-Virtualisierung

Provisionierung ist die Bezeichnung für die automatisierte Bereitstellung von IT-Ressourcen.

<http://wirtschaftslexikon.gabler.de/Definition/provisionierung.html>

Eine kurze Geschichte der Systemadministration.



Ohne Virtualisierung (vor 2000)

- Manuelles Installieren von Betriebssystem auf dedizierter Hardware
- Manuelle Installation von Infrastruktur-Software
- Manuelle / Teilautomatisierte / Automatische Installation der Anwendungssoftware per Installer, Skript, proprietäre Lösungen

Virtualisierung einzelner Maschinen (2000 – heute)

- Manuelles Installieren von virtuellen Maschinen
- Manuelle Installation von Infrastruktur-Software
- Manuelle / Teilautomatisierte / Automatische Installation der Anwendungssoftware per Installer, Skript, proprietäre Lösungen

Eine kurze Geschichte der Systemadministration.



Virtualisierung in der Cloud (seit 2010)

- Automatisches Bereitstellen von vorgefertigten virtuellen Maschinen und Containern
- Manuelle Installation der Infrastruktur-Software nur 1x im Clone-Master-Image
- Bereitstellen einer definierten Umgebung auf Knopfdruck

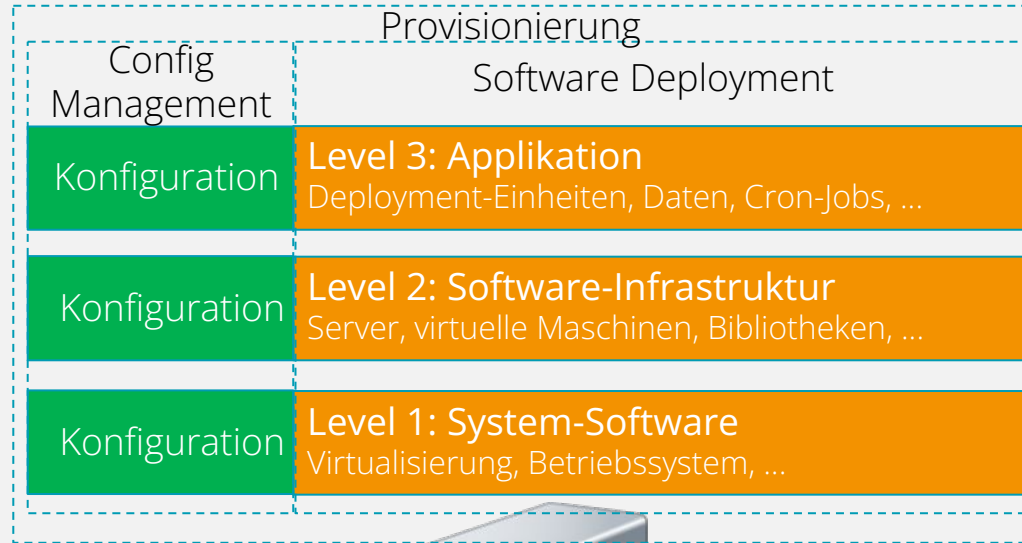
Infrastructure-as-Code (2010 – heute)

- Programmierung der Provisionierung und weiterer Betriebsprozeduren
- Code-basiert und unter Versionskontrolle

Provisionierung erfolgt auf drei verschiedenen Ebenen und in vier Stufen.



QA|WARE



Hardware

- Rechner
- Speicher
- Netzwerk-Equipment
- ...

Laufende Software!



Application Provisioning



Server Provisioning

Bereitstellung der notwendigen Software-Infrastruktur für die Applikation.



Bootstrapping

Bereitstellung der Betriebsumgebung für die Software-Infrastruktur.



Bare Metal Provisioning

Initialisierung einer physikalischen Hardware für den Betrieb.

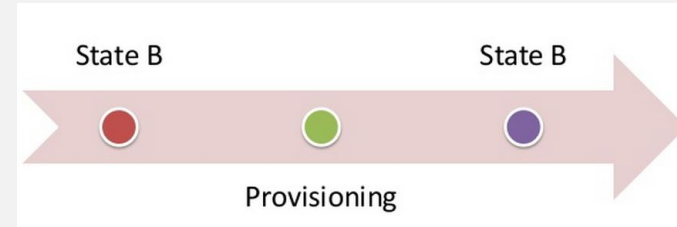
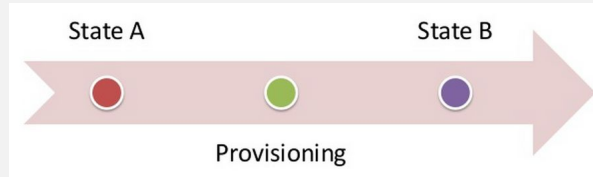
Konzeptionelle Überlegungen zur Provisionierung.



QA|WARE

Systemzustand := Gesamtheit der Software, Daten und Konfigurationen auf einem System.

Provisionierung := Überführung von einem System in seinem aktuellen Zustand auf einen Ziel-Zustand.



Was ein Provisionierungsmechanismus leisten muss:

1. Ausgangszustand feststellen
2. Vorbedingungen prüfen
3. Zustandsverändernde Aktionen ermitteln
4. Zustandsverändernde Aktionen durchführen
5. Nachbedingungen prüfen und ggf. Zustand zurücksetzen

Zusicherungen

Idempotenz: Die Fähigkeit eine Aktion durchzuführen und sie dasselbe Ergebnis erzeugt, egal ob sie einmal oder mehrfach ausgeführt wird.

Konsistenz: Nach Ausführung der Aktionen herrscht ein konsistenter Systemzustand. Egal ob einzelne, mehrere oder alle Aktionen gescheitert sind.

Die neue Leichtigkeit des Seins.



QA|WARE

Old Style

Beliebiger
Zustand



1. Ausgangszustand feststellen
2. Vorbedingungen prüfen
3. Zustandsverändernde Aktionen ermitteln
4. Zustandsverändernde Aktionen durchführen
5. Nachbedingungen prüfen und ggF. Zustand zurücksetzen



Ziel-Zustand

New Style

„Immutable Infrastructure / Phoenix Systems“

Basis-Zustand



- ~~1. Ausgangszustand feststellen~~
- ~~2. Vorbedingungen prüfen~~
- ~~3. Zustandsverändernde Aktionen ermitteln~~
4. Zustandsverändernde Aktionen durchführen
5. Nachbedingungen prüfen und ggF. Zustand zurücksetzen



Ziel-Zustand

Immutable Infrastructure



An *immutable infrastructure* is another infrastructure paradigm in which servers are **never modified** after they're deployed. If something needs to be updated, fixed, or modified in any way, **new servers built from a common image with the appropriate changes** are provisioned to replace the old ones. After they're validated, they're put into use and **the old ones are decommissioned**.

The benefits of an immutable infrastructure include **more consistency and reliability** in your infrastructure and a **simpler, more predictable deployment process**. It mitigates or entirely **prevents** issues that are common in mutable infrastructures, like **configuration drift and snowflake servers**. However, using it efficiently often includes comprehensive deployment automation, fast server provisioning in a cloud computing environment, and solutions for handling stateful or ephemeral data like logs.

Quelle: <https://www.digitalocean.com/community/tutorials/what-is-immutable-infrastructure>



QA|WARE

Dockerfiles und Docker Compose

Provisionierung mit Dockerfile und Docker Compose



QAWARE

Deployment-Ebenen

Level 3: Applikation

Deployment-Einheiten, Daten, Cron-Jobs, ...

Level 2: Software-Infrastruktur

Server, virtuelle Maschinen, Bibliotheken, ...

Level 1: System-Software

Virtualisierung, Betriebssystem, ...

Docker-Image-Kette

Applikations-Image
(z.B. www.qaware.de)

Server Image
(z.B. NGINX)

Base Image
(z.B. Ubuntu)



Application Provisioning
DockerFile & Docker Compose



Server Provisioning
DockerFile



Bootstrapping
Docker Pull Base Image

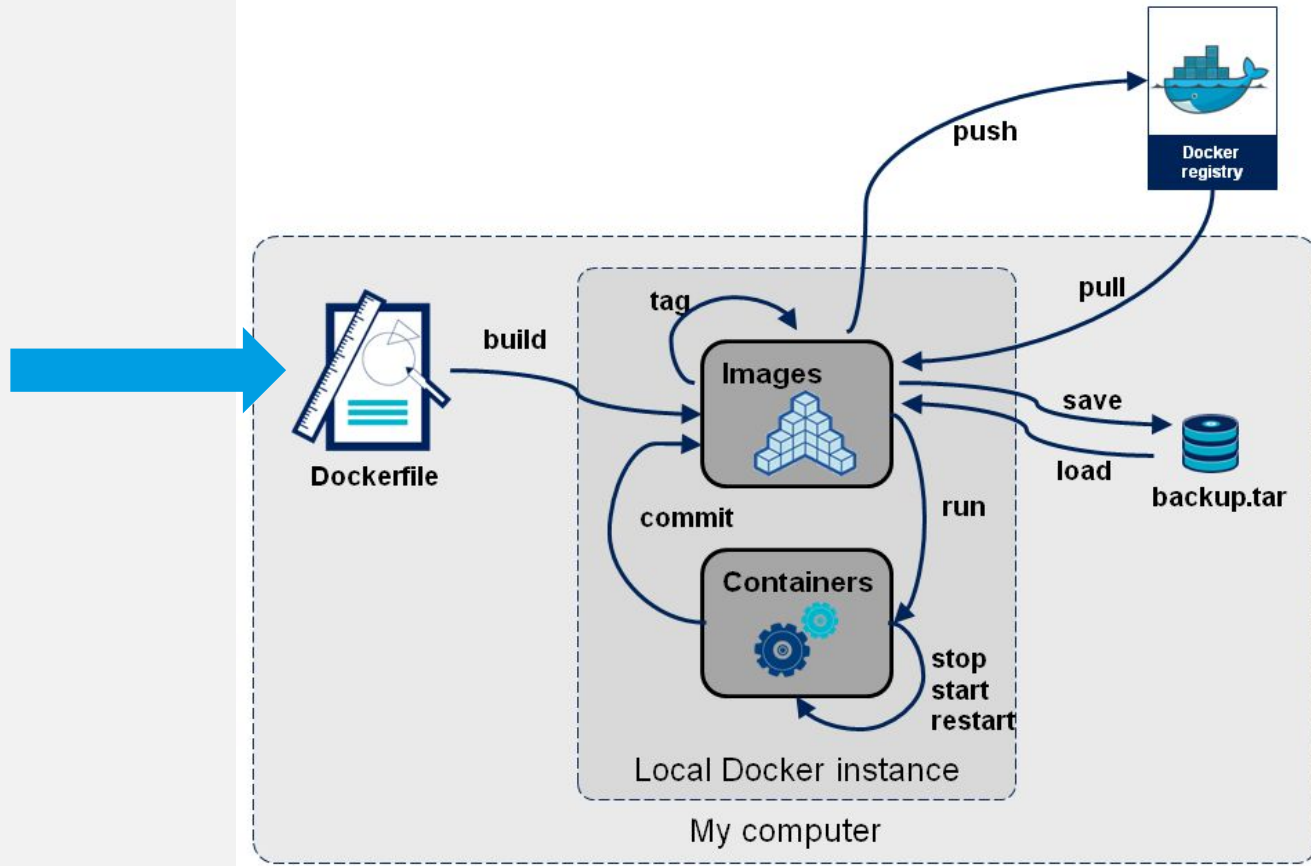


Bare Metal Provisioning
Docker Daemon installieren

Provisionierung von Images mit dem Dockerfile.



QA|WARE



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 wird zum Bau des Image verwendet



QA|WARE

```
FROM centos:centos8
```

```
RUN yum install -y epel-release && \  
    yum install -y && \  
    yum install -y php php-mysql php-fpm && \  
    sed -i -e "s/user = apache/user = nginx/g" /etc/php-fpm.d/www.conf && \  
    sed -i -e "s/group = apache/group = nginx/g" /etc/php-fpm.d/www.conf
```

```
EXPOSE 80
```

```
ENTRYPOINT php-fpm
```

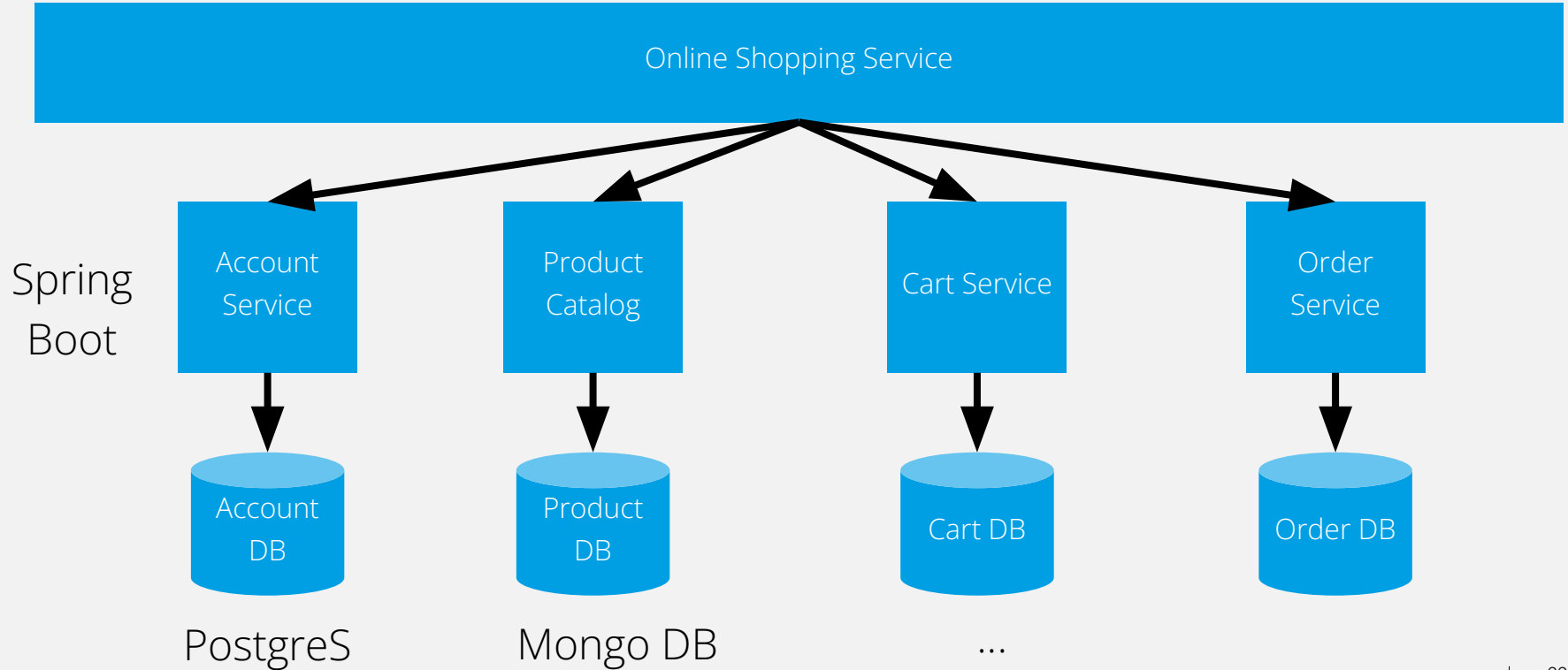

Wiederholung: Dockerfile-Kommandos

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 <code>docker run</code> .
ENTRYPOINT <command>	The process to be started at container startup
CMD <command>	Parameters to the entrypoint process if no parameters are passed with <code>docker run</code>
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

Was machen wir mit Multi-Container-Applikationen?



QA|WARE



Docker Compose /1



QA|WARE

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

(<https://docs.docker.com/compose/>)

Docker Compose /2



QA|WARE

Bei der Nutzung von Docker Compose führt man im Wesentlichen die folgenden 3 Schritte aus:

1. Für alle eigenen Anteile an der Anwendung schreibt man ein `Dockerfile`. Für alle fremden Anteile sucht man das passende Image.
2. Alle Services/Bestandteile, aus denen die Anwendung besteht, definiert man in der `docker-compose.yml`. Dadurch werden diese in derselben isolierten Umgebung ausgeführt.
3. Über `docker compose up` startet man dann alle Bestandteile auf einmal.

Zusätzlicher Komfort im Vergleich zu Docker:

- Auf dem gleichen Host kann mehrfach die gleiche isolierte Umgebung gestartet werden (z.B. interessant für Buildserver)
- Daten in gemounteten Volumes bleiben auch beim Neustart erhalten
- Nur tatsächlich geänderte Images werden bei einem Neustart neu gebaut
- Konfiguration über Variablen möglich

Anwendungsfälle sind in der Praxis vor allem:

- Lokale Entwicklung
- Automatisierte Tests

Nutzung von Docker Compose für Multi-Container Apps.



docker-compose.yml

\$ docker compose build

\$ docker compose up -d

\$ docker compose stop

\$ docker compose rm -s -f

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ./code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```



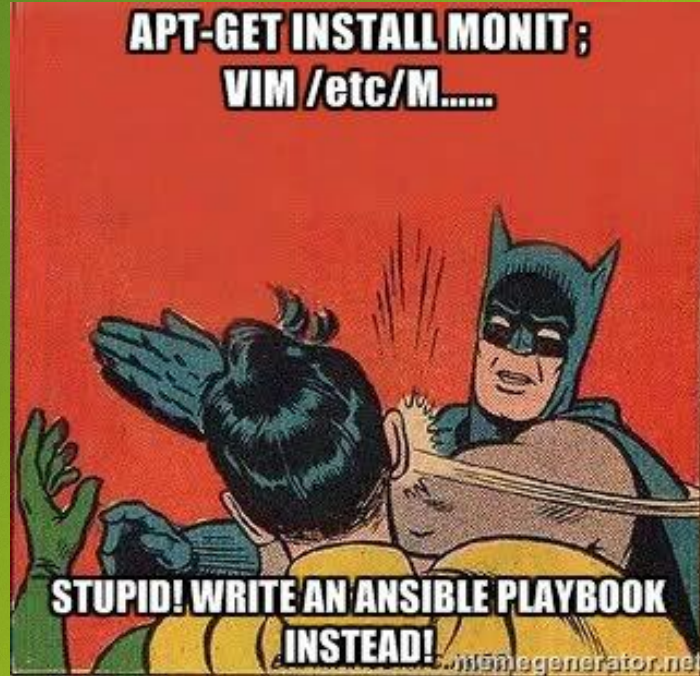
QA|WARE

Übung 1: Docker und Docker Compose

Ansible

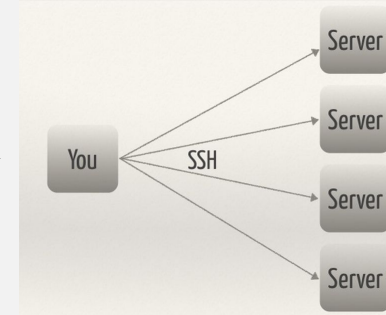


QA|WARE



Ansible

- Open-Source-Provisionierungswerkzeug von Red Hat
- Ausgelegt auf die Provisionierung großer heterogener IT-Landschaften
- Entwickelt in der Sprache Python
- Push-Prinzip: Benötigt im Vergleich zu anderen Lösung weder einen Agenten auf den Ziel-Rechnern (SSH & Python reicht) noch einen zentralen Provisionierungs-Server
- Variante ansible-container zur Provisionierung von Containern
- Ist einfach zu erlernen im Vergleich zu anderen Lösungen. Deklativer Stil.
- Umfangreiche Bibliothek vorgefertigter Provisionierungs-Aktionen inkl. Community-Funktion (<https://galaxy.ansible.com>) und Beispielen (<https://github.com/ansible/ansible-examples>)



Provisionierung mit Ansible



Deployment-Ebenen

Level 3: Applikation

Deployment-Einheiten, Daten, Cron-Jobs, ...

Level 2: Software-Infrastruktur

Server, virtuelle Maschinen, Bibliotheken, ...

Level 1: System-Software

Virtualisierung, Betriebssystem, ...

Docker-Image- oder VM-Kette

Applikations-Image
(z.B. www.qaware.de)

Server Image
(z.B. NGINX)

Base Image
(z.B. Ubuntu)



Application Provisioning
Ansible oder Ansible Container



Server Provisioning
Ansible oder Ansible Container



Bootstrapping
SSH Daemon & Python
installieren



Bare Metal Provisioning
Betriebssystem installieren

ANSIBLE

A Beginners Tutorial

by Ben Fleckenstein



Ansible – Konzepte & Begriffe

Beschreibung der
Maschinen über IP,
Shortnames oder URLs



QA|WARE

Inventory

Modules

Tasks

Roles

Playbook

Gruppen fassen mehrere
Maschinen zusammen

```
[webserver]
my-web-server.example.com
my-other-web-server.example.com
```

```
[appserver-master]
app1-master ansible_ssh_host=myapp.example.net httpsports=9090
app2-master ansible_ssh_host=myapp2.example.net
httpsports=9091
```

```
[appserver-slaves]
app1-slave ansible_ssh_host=myapp3.example.net httpsports=9090
app2-slave ansible_ssh_host=myapp4.example.net httpsports=9091
```

hosts

Definition von Variablen
für einzelne Hosts oder
Gruppen

Ansible – Konzepte & Begriffe



QAWARE

Inventory

Modules

Tasks

Roles

Playbook

- Modules erlauben die Interaktion über Ansible
- Man kann
 - selbst Modules schreiben
 - offizielle Ansible Modules nutzen (Core), diese sind schon Teil von Ansible
 - Modules aus der Community nutzen (Extras)
- Beispiele:
 - **File handling:** file, copy, template
 - **Remote execution:** command, shell
 - **Package management:** apt, yum

Ansible – Konzepte & Begriffe



Inventory

Modules

Tasks

Roles

Playbook

- Jeder Task beschreibt eine Provisionierungs-Aktion
- Beispiel: Installieren von Paketen über apt
- Dabei ruft der Task ein Modul auf, das die aktuelle Aufgabe umsetzt.

- Ausführung über Ad Hoc Commands

`ansible -m <module>`

`-a <arguments> <server>`

Ansible – Konzepte & Begriffe



QA|WARE

Inventory

Modules

Tasks

Roles

Playbook

```
# roles/example/tasks/main.yml
- name:
  import_tasks: redhat.yml
  when: ansible_facts['os_family']|lower == 'redhat'
- import_tasks: debian.yml
  when: ansible_facts['os_family']|lower == 'debian'

# roles/example/tasks/redhat.yml
- ansible.builtin.yum:
  name: "httpd"
  state: present

# roles/example/tasks/debian.yml
- ansible.builtin.apt:
  name: "apache2"
  state: present
```

Ansible – Konzepte & Begriffe



QAWARE

Inventory

Modules

Tasks

Roles

Playbook

- Playbooks als Basis für Config Management & Orchestrierung

```
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
    - name: ensure apache is at the latest version
      ansible.builtin.yum:
        name: httpd
        state: latest
  [...]
```

Die wichtigsten zu erstellenden Dateien bei einer Provisionierung mit Ansible.



QA|WARE

Playbook (YAML-Syntax) Provisionierungs-Skript.

```
- hosts: all
  tasks:
  - yum: pkg=httpd state=installed
```

- *Modul* = Implementierung einer Provisionierungs-Aktion
- *Task* = Beschreibung einer Provisionierungs-Aktion
- *Role* = Ausführung von Tasks auf Hosts oder Host-Gruppen

Playbooks

Roles

Tasks

Modules

Inventory Hosts

```
[mongo_master]
168.197.1.14
```

```
[mongo_slaves]
168.197.1.15
168.197.1.16
168.197.1.17
```

```
[www]
168.197.1.2
```

Inventory

Groups

Hosts

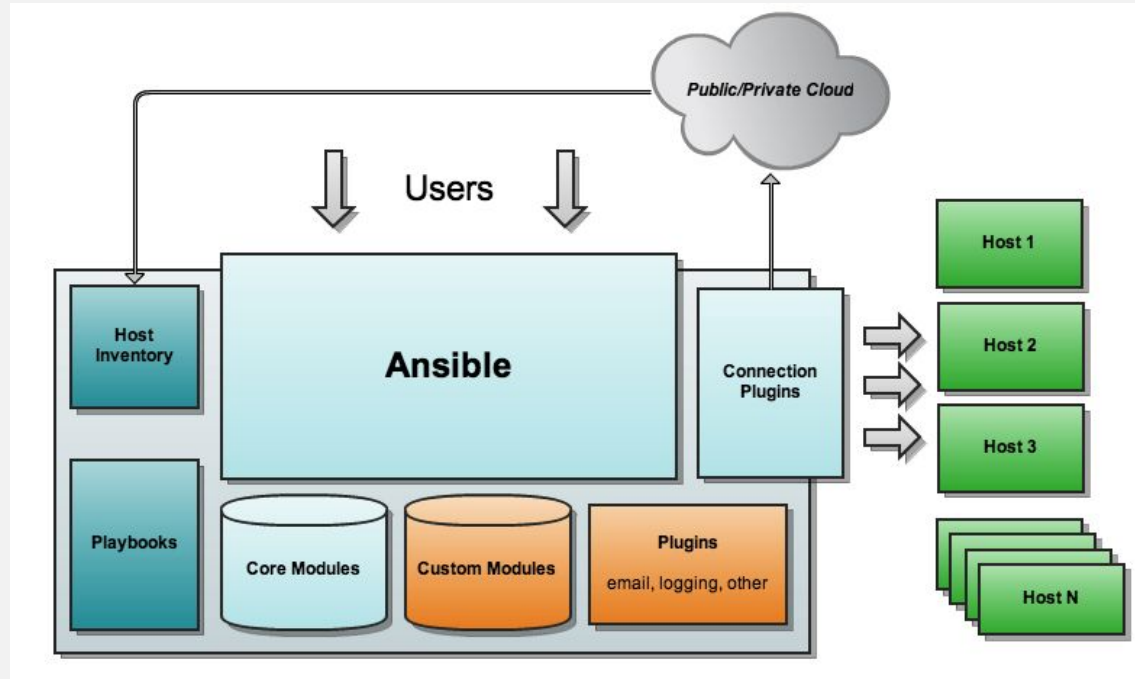
Ansible Konfiguration ansible.cfg

```
1 [defaults]
2 host_key_checking = False
3 hostfile           = /ansible/hosts
4 private_key_file    = /ansible/id_rsa
```

Architektur von Ansible



QA|WARE



Es stehen in Ansible viele vorgefertigte Module zur Verfügung.



QA|WARE

Module Index

- [All Modules](#)
- [Cloud Modules](#)
- [Commands Modules](#)
- [Database Modules](#)
- [Files Modules](#)
- [Inventory Modules](#)
- [Messaging Modules](#)
- [Monitoring Modules](#)
- [Network Modules](#)
- [Notification Modules](#)
- [Packaging Modules](#)
- [Source Control Modules](#)
- [System Modules](#)
- [Utilities Modules](#)
- [Web Infrastructure Modules](#)
- [Windows Modules](#)

https://docs.ansible.com/ansible/2.9/modules/modules_by_category.html

Die Provisionierung wird über die Kommandozeile gesteuert.



QA|WARE

- Ad-hoc-Kommandos

- `ansible <host gruppe> -i <inventory-file> -m <modul> -a „<argumente>“ -f <parallelism>`

- Beispiele:

- › `ansible all -m ping`

- › `ansible all -a „/bin/echo hello“`

- › `ansible web -m apt -a „name=nginx state=installed“`

- › `ansible web -m service -a „name=nginx state=started“`

- › `ansible all -a "/sbin/reboot" -f 10`

- Playbooks ausführen

- `ansible-playbook <playbook.yaml>`



QA|WARE

Übung 2: Ansible



QA|WARE

Packer

Packer



Packer is an open source tool for creating identical machine images for multiple platforms from a single source configuration. Packer is lightweight, runs on every major operating system, and is highly performant, creating machine images for multiple platforms in parallel. Packer does not replace configuration management like Chef or Puppet. In fact, when building images, Packer is able to use tools like Chef or Puppet to install software onto the image.

A machine image is a single static unit that contains a pre-configured operating system and installed software which is used to quickly create new running machines. Machine image formats change for each platform. Some examples include AMIs for EC2, VMDK/VMX files for VMware, OVF exports for VirtualBox, etc.

<https://www.packer.io/intro>

- Geschrieben in Go
- Templatisiert den Bau von Images
- Bestehende Provisionierungsskripte (z.B. Ansible) können wiederverwendet werden
- Ermöglicht den Bau von Images für mehrere Plattformen mit einer gemeinsamen Konfiguration

Packer Terminologie (<https://www.packer.io/docs/terminology>)

Artifacts

- Ergebnis eines Packer Builds, z.B. ein Dateiordner oder ein Set von AMI IDs

Builds

- Tasks, die ein Image für eine bestimmte Plattform erzeugen

Builders

- erzeugen einen bestimmten Image Typ
- z.B. VirtualBox, Amazon EC2, Docker

Commands

- Unter-Commands, die man mit Packer ausführen kann, z.B. packer build

Post-Processors

- erzeugen aus Artifacts neue Artifacts (z.B. Komprimierung, Tagging, Publishing)

Provisioners

- installieren und konfigurieren Software in einer laufenden Instanz, bevor daraus ein statisches Artifact erzeugt wird

Templates

- JSON Files, die den Packer Build konfigurieren

Beispiel



QA|WARE

```
packer {
  required_plugins {
    docker = {
      version = ">= 0.0.7"
      source = "github.com/hashicorp/docker"
    }
  }
}

source "docker" "ubuntu" {
  image = "ubuntu:xenial"
  commit = true
}

...
```

```
...

build {
  name = "learn-packer"
  sources = [
    "source.docker.ubuntu"
  ]
  provisioner "shell" {
    environment_vars = [
      "FOO=hello world",
    ]
    inline = [
      "echo Adding file to Docker
Container",
      "echo \"FOO is $FOO\" > example.txt",
    ]
  }
}
```

Packer



QA|WARE

<https://www.youtube.com/watch?v=Mj-yqt7ejnM>



QA|WARE

Übung 3: Packer (optional)