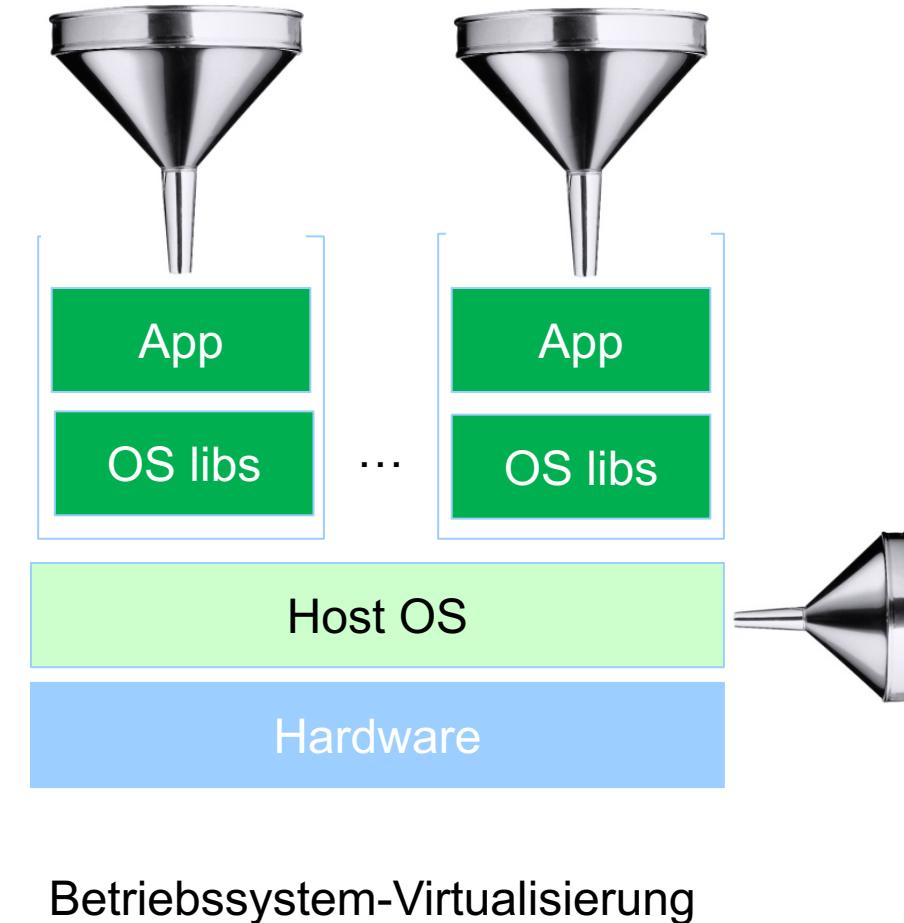
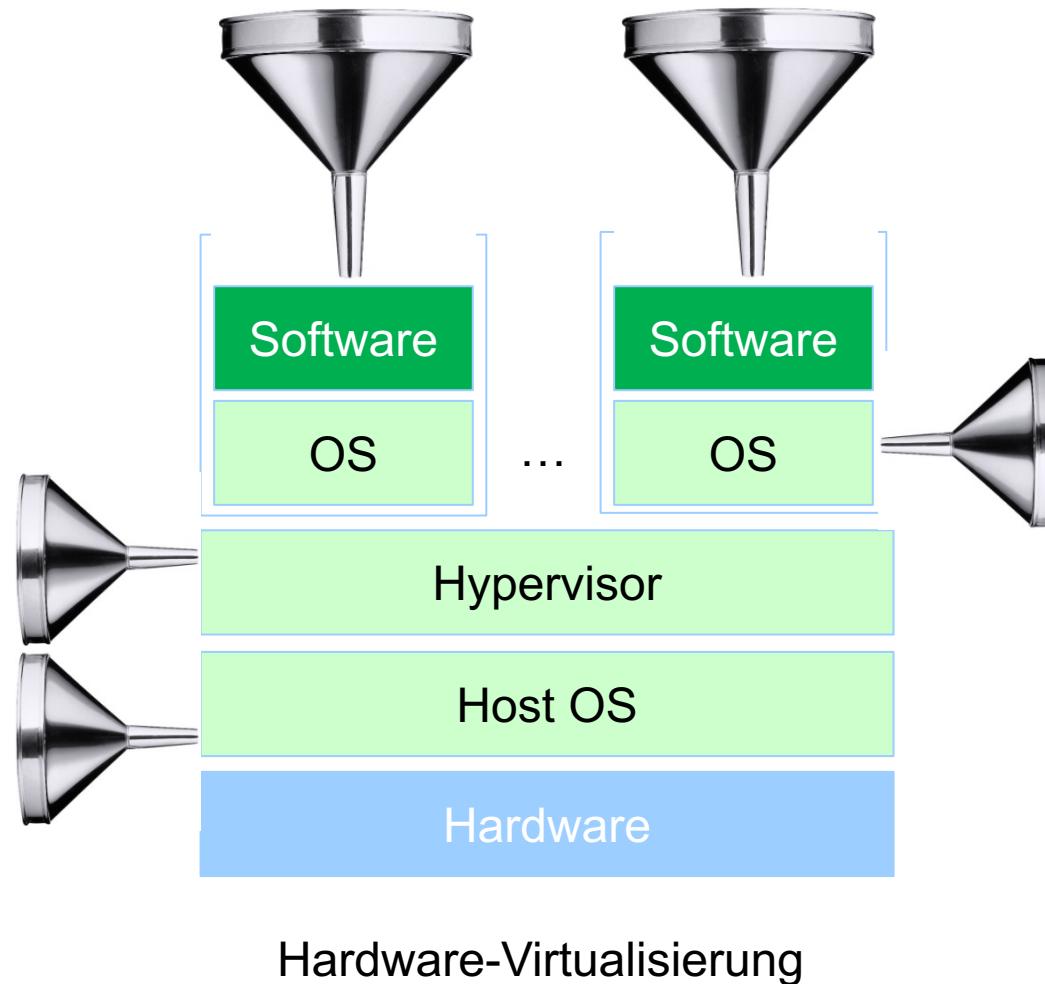


Kapitel 4: Provisionierung

vorlesung

**CLOUD
COMPUTING**

Provisionierung: Wie kommt Software in die Boxen?



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

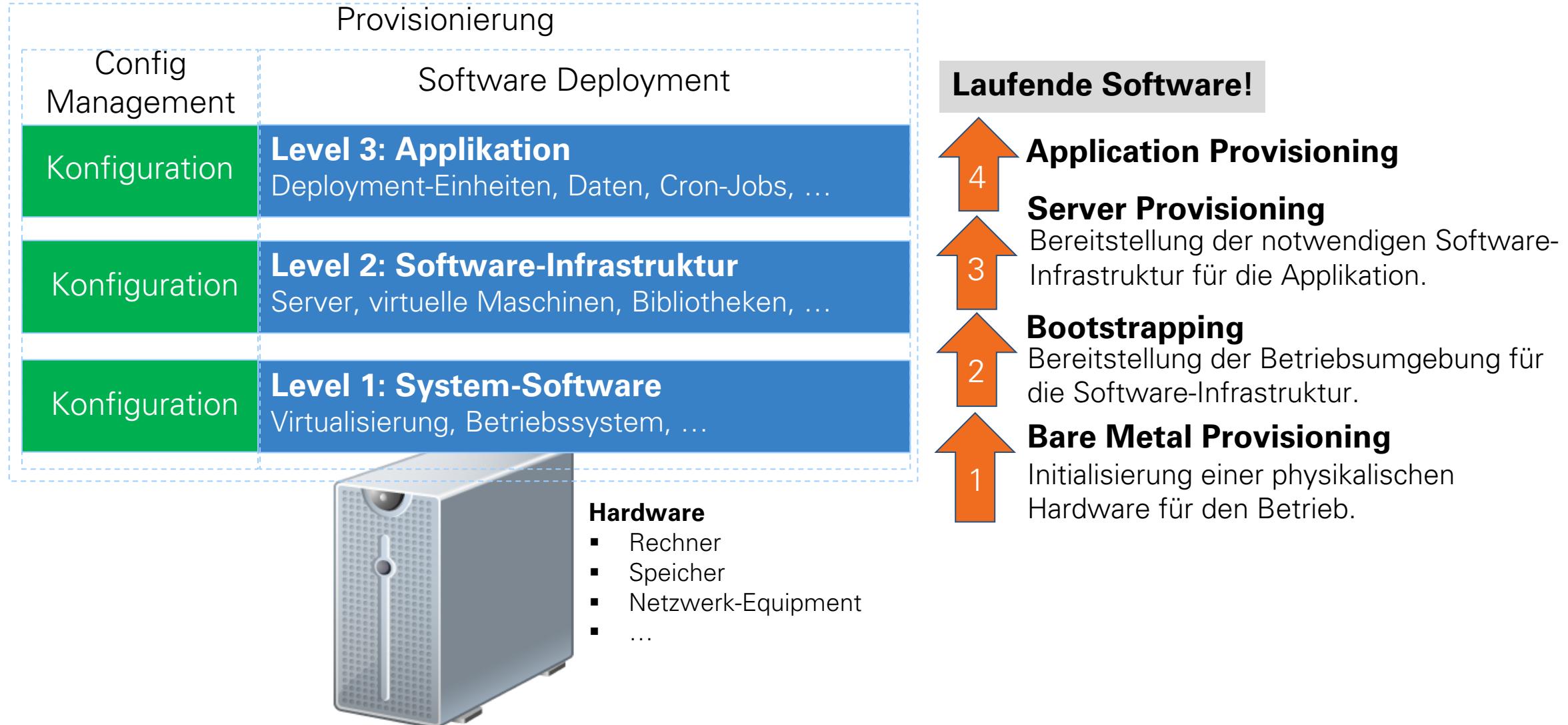
Virtualisierung in der Cloud (seit 2010)

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

Infrastructure-as-Code (2010 – heute)

- Programmierung der Provisionierung und weiterer Betriebsprozeduren

Provisierung erfolgt auf drei verschiedenen Ebenen und in vier Stufen.



Konzeptionelle Überlegungen zur Provisionierung.

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 das selbe 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.

Old Style



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

New Style

„Immutable Infrastructure / Phoenix Systems“



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

Eine Übersicht gängiger Provisionierungswerkzeuge.

Imperativ

Shell Scripting



PowerShell



Shell Abstraktion



ANSIBLE

Deskriktiv

Zustandsautomaten



Dockerfiles und Docker Compose

Provisionierung mit DockerFile und Docker Compose

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

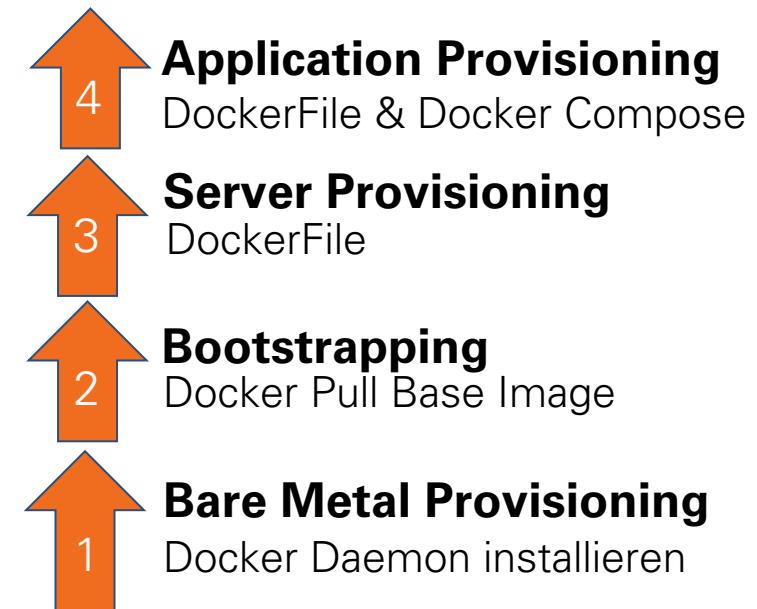
(z.B. www.qaware.de)

Server Image

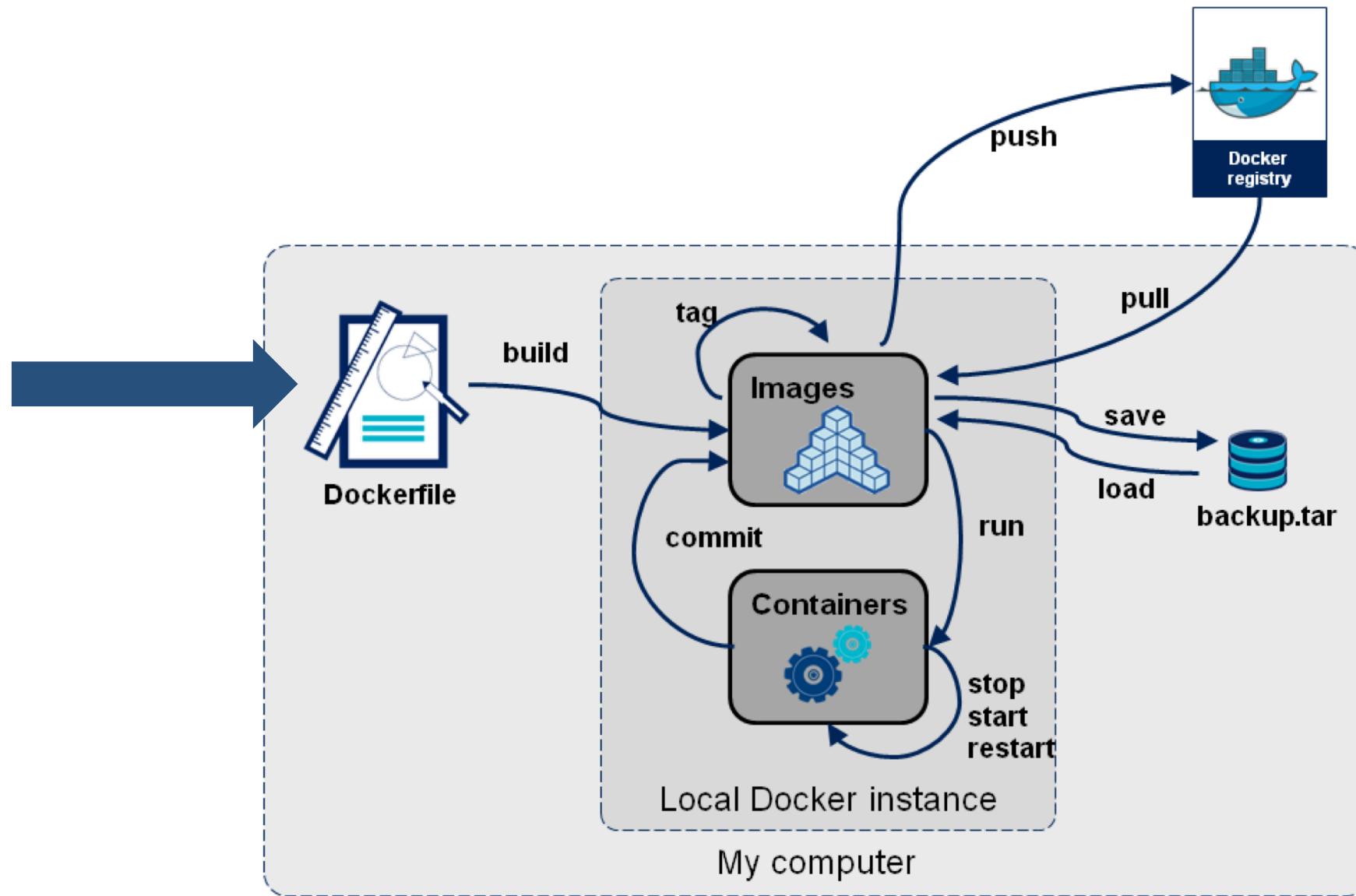
(z.B. NGINX)

Base Image

(z.B. Ubuntu)



Provisionierung von Images mit dem Dockerfile.



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

```
FROM centos:7.4.1708

RUN yum install -y epel-release && \
    yum install -y wget nginx && \
    yum install -y php php-mysql php-fpm && \
    sed -i -e "s/;\?cgi.fix_pathinfo\s*=\s*\!1/cgi.fix_pathinfo = 0/g" /etc/php.ini && \
    sed -i -e "s/daemonize = no/daemonize = yes/g" /etc/php-fpm.conf && \
    sed -i -e "s/;\?listen.owner\s*=\s*nobody/listen.owner = nobody/g" /etc/php-fpm.d/www.conf && \
    sed -i -e "s/;\?listen.group\s*=\s*nobody/listen.group = nobody/g" /etc/php-fpm.d/www.conf && \
    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

COPY docker/php.conf /etc/nginx/default.d/
EXPOSE 80
ENTRYPOINT php-fpm && nginx -g 'daemon off;'
```

Dockerfile Commands

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 –e 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

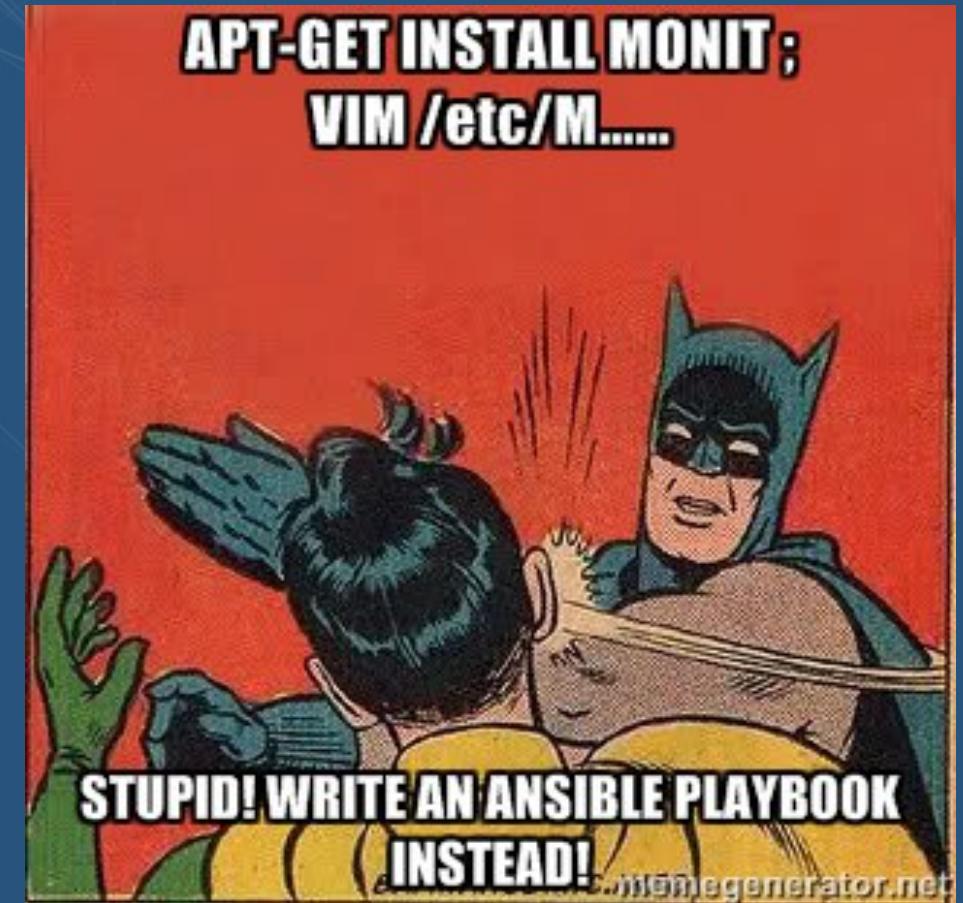
Nutzung von Docker Compose für Multi-Container Apps.

docker-compose.yml

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

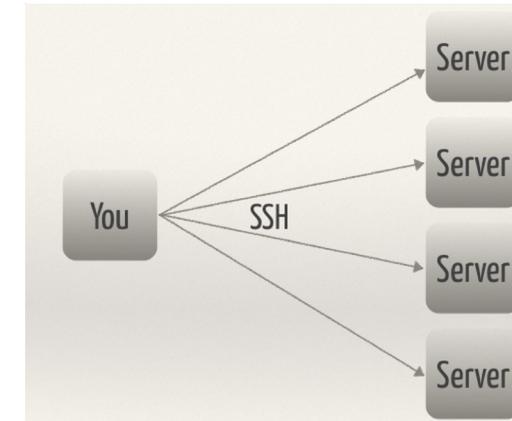
```
$ docker-compose up -d --build
$ docker-compose stop
$ docker-compose rm -s -f
```

Ansible



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.
Deklarativer 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

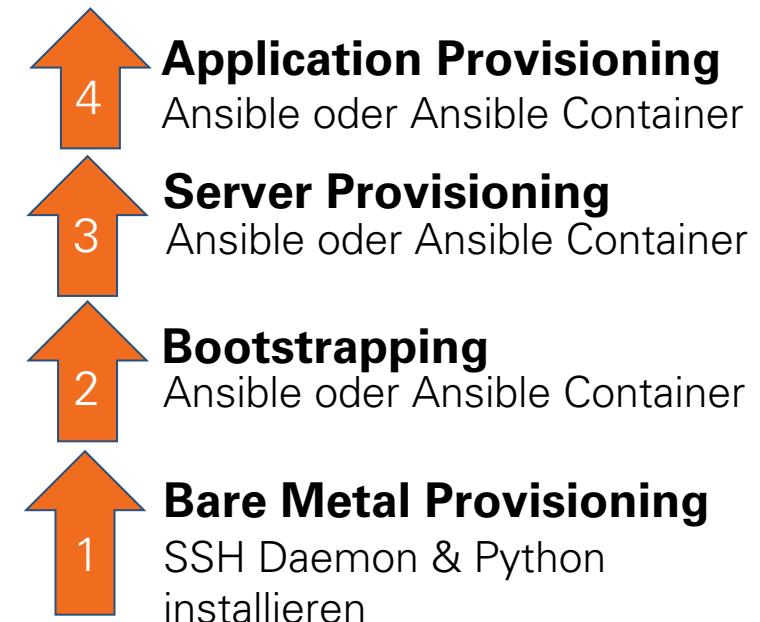
(z.B. www.qaware.de)

Server Image

(z.B. NGINX)

Base Image

(z.B. Ubuntu)



Die wichtigsten zu erstellenden Dateien bei einer Provisionierung mit Ansible.

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

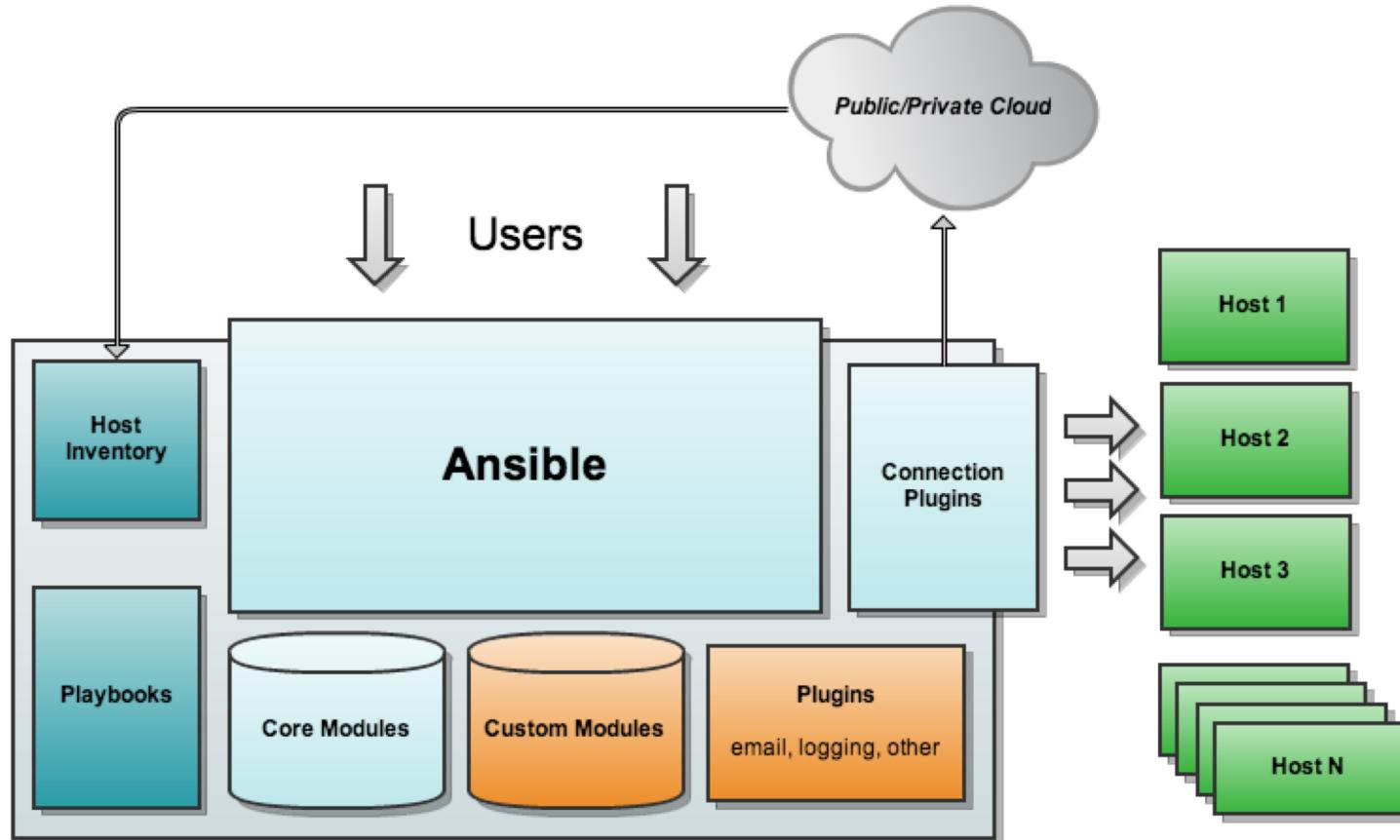
Hosts

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



Es stehen in Ansible viele vorgefertigte Module zur Verfügung.

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](#)

http://docs.ansible.com/modules_by_category.html
http://docs.ansible.com/list_of_all_modules.html

Die Provisionierung wird über die Kommandozeile gesteuert.

■ Ad-hoc Kommandos

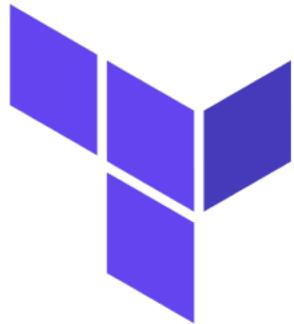
- **ansible <host gruppe> -i <inventory-file>**
 -m <modul> -a „<argumente>“ -f <parallelism>

■ 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>**

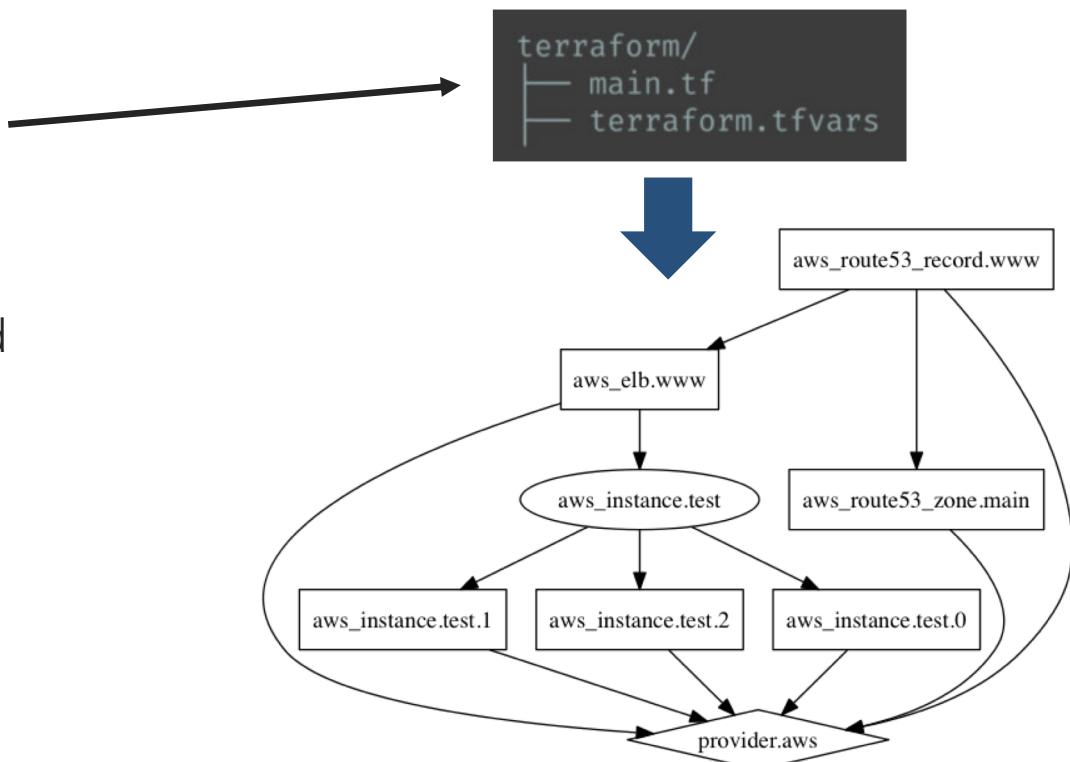


HashiCorp
Terraform

Write, Plan, and Create Infrastructure as Code

Terraform

- Entwickelt von HashiCorp
- Open Source, in Go geschrieben
- Kommandozeilenwerkzeug
- Direkte Anbindung vieler Cloud Provider (AWS, Azure, OTC, ...)
- Deklarative Programmierung
 - **Write**: Beschreibung Zielzustand über eine domänenspezifische Sprache HCL (HashiCorp Configuration Language)
 - **Plan** (`terraform plan`): Ist-Zustand ermitteln. Notwendige Änderungen planen (entsprechend Abhängigkeiten geordnet und parallelisiert, Unterbrechungen möglichst minimal)
 - **Apply** (`terraform apply`): Idempotente Herstellung des Zielzustands. Der Zustand (`.tfstate` Datei) wird dabei lokal oder in einem Remote Store (S3, HTTP, ...) gespeichert

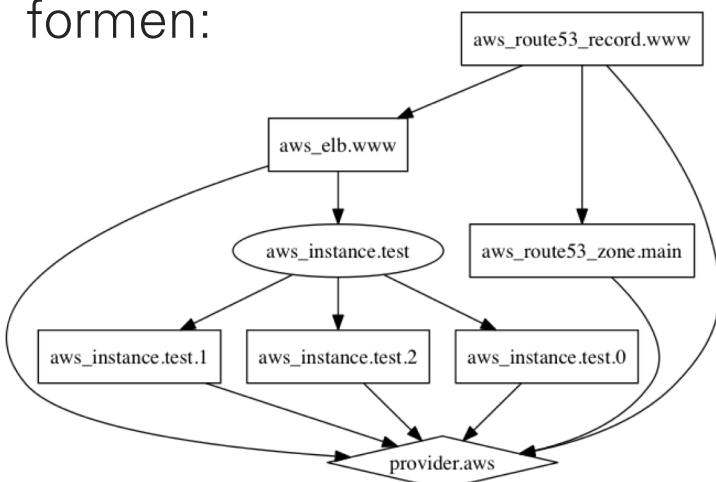


Die Kern-Entitäten eines Terraform-Skripts

Ressource: Provisionierte Komponente der Infrastruktur

```
resource "aws_instance" "web" {  
    ami          = "ami-408c7f28"  
    instance_type = "t1.micro"  
}
```

... haben Abhängigkeiten zueinander, die einen DAG formen:



Provider: Integration der zu provisionierenden Infrastruktur

Alicloud	Archive	AWS
Azure	Bitbucket	CenturyLinkCloud
Chef	Circonus	Cloudflare
CloudScale.ch	CloudStack	Cobbler
Consul	Datadog	DigitalOcean
DNS	DNSMadeEasy	DNSimple
Docker	Dyn	External
Fastly	GitHub	Gitlab

```
provider "aws" {  
    access_key = "${var.aws_access_key}"  
    secret_key = "${var.aws_secret_key}"  
    region     = "us-east-1"  
}
```

OpsGenie	Oracle Public Cloud	Oracle Cloud Platform
OVH	Packet	PagerDuty
Palo Alto Networks	PostgreSQL	PowerDNS
ProfitBricks	RabbitMQ	Rancher
Random	Rundeck	Scaleway
SoftLayer	StatusCake	Spotinst
Template	Terraform	Terraform Enterprise
TLS	Triton	UltraDNS
Vault	VMware vCloud Director	VMware NSX-T

Provisioner: Ausführung von Änderungen auf Ressourcen.

```
resource "aws_instance" "web" {  
    # ...  
  
    provisioner "local-exec" {  
        command = "echo ${self.private_ip} > file.txt"  
    }  
}
```

Provisionierung mit Terraform

Deployment-Ebenen

Level 3: Applikation

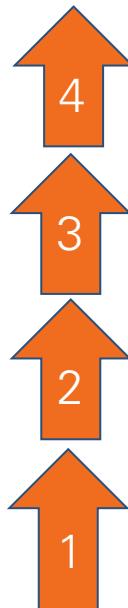
Deployment-Einheiten, Daten, Cron-Jobs, ...

Level 2: Software-Infrastruktur

Server, virtuelle Maschinen, Bibliotheken, ...

Level 1: System-Software

Virtualisierung, Betriebssystem, ...



Application Provisioning

Terraform steuert an (Provisioner oder Provider)

Server Provisioning

Terraform steuert an (Provisioner oder Provider)

Bootstrapping

Terraform steuert an (Provider)

Bare Metal Provisioning

Terraform steuert an (Matchbox)

Beispiel

```
# New resource for the S3 bucket our application will use.
resource "aws_s3_bucket" "example" {
    # NOTE: S3 bucket names must be unique across _all_ AWS accounts, so
    # this name must be changed before applying this example to avoid naming
    # conflicts.
    bucket = "terraform-getting-started-guide"
    acl     = "private"
}

# Change the aws_instance we declared earlier to now include "depends_on"
resource "aws_instance" "example" {
    ami          = "ami-2757f631"
    instance_type = "t2.micro"

    # Tells Terraform that this EC2 instance must be created only after the
    # S3 bucket has been created.
    depends_on = ["aws_s3_bucket.example"]
}
```

Anhang

Dockerfile Best Practices

A Docker build must be repeatable.

- A build at a later time must produce an identical image.
- Keep care with versions
 - All files for the image are stored in the repository of the Dockerfile
 - No LATEST tag, use explicit versions instead
 - Always define a version when installing software

```
RUN apt-get update && apt-get install -y ruby1.9.1
```

Concatenate associated commands in the `RUN` command

- Every RUN command produces a Layer
- Less Layers are better for building and contributing images
- Concatenate commands with \

Installation of several software packages

```
RUN apt-get update && apt-get install -y wget \
    git-core=1:1.9.1-1 \
    subversion=1.8.8-1ubuntu3.2 \
    ruby=1:1.9.3.4 && \
    apt-get clean
```

Remove temporary files

- Remove all temporary files of the build process to produce small Docker Images
- Use the clean command
- Don't use the clean command in a separate RUN command (it is not possible to clean a different Layer)

Installation of a Linux Package with YUM

```
RUN yum -y install mypackage1 && \
    yum -y install mypackage2 && \
    yum clean all -y
```

Publish important ports with **EXPOSE**

- EXPOSE makes a port accessible for the host system or other containers
- Exposed Ports
 - are shown by the docker ps command
 - are listed in the image meta data by the docker inspect command
 - will be connected automatically by linked containers



EXPOSE 12340

Define Environment Variables

- Visible in Dockerfile
- Can be used during Build and Execution
- Can be overwritten at the start of a container

```
ENV JAVA_HOME /opt/java-oracle/jdk1.8.0_92
```

```
ENV MAVEN_HOME /usr/share/maven
```