QAWARE SOFTWARE ENGINEERING

HM Hochschule München University of Applied Sciences

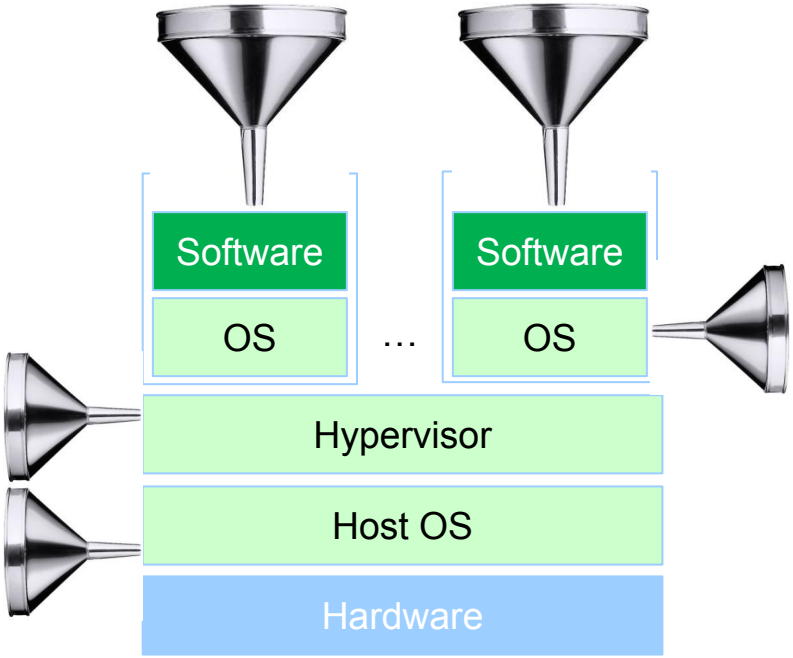
Cloud Computing

Provisionierung

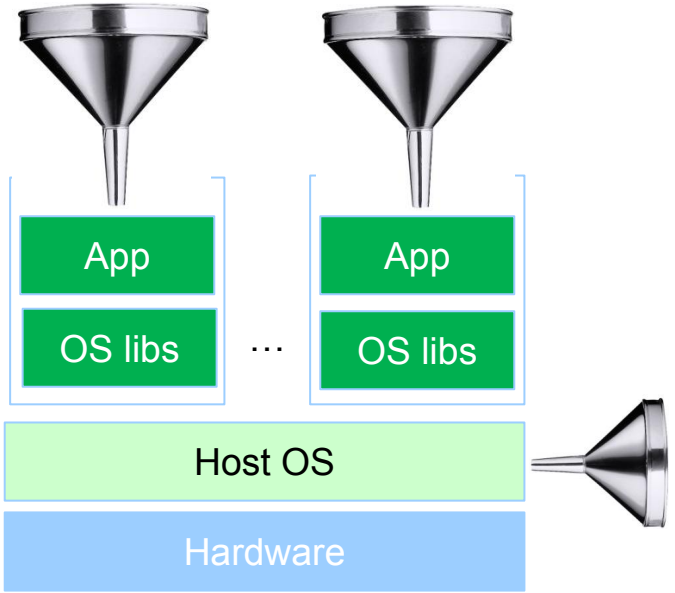
Bernhard Schaidhammer

2025-04-18

Provisionierung: Wie kommt Software in die Boxen?



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 / Configuration-as-Code (2010 – heute)

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

Provisierung erfolgt auf **3 Ebenen** und in **4 Stufen**

Provisionierung

Config Management	Software Deployment
Konfiguration	**Ebene 3: Applikation** Deployment-Einheiten, Daten, Cron-Jobs, ...
Konfiguration	**Ebene 2: Software-Infrastruktur** Server, virtuelle Maschinen, Bibliotheken, ...
Konfiguration	**Ebene 1: System-Software** Virtualisierung, Betriebssystem, ...

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

Laufende Software!

4 **Application Provisioning**

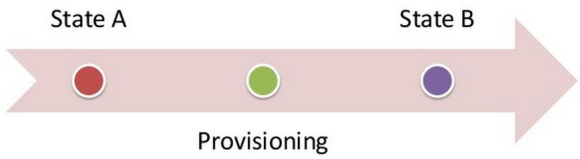
3 **Server Provisioning**
Bereitstellung der notwendigen Software-Infrastruktur für die Applikation.

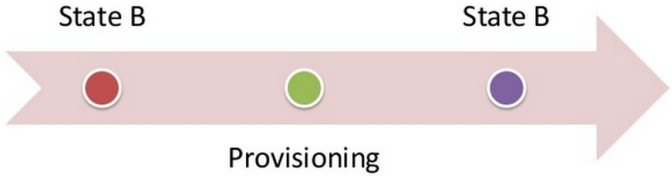
2 **Bootstrapping**
Bereitstellung der Betriebsumgebung für die Software-Infrastruktur.

1 **Bare Metal Provisioning**
Initialisierung einer physikalischen Hardware für den Betrieb.

Konzeptionelle Überlegungen zur Provisionierung

Systemzustand := Gesamtheit der Software, Daten und Konfigurationen auf einem System.
Provisionierung := Überführung eines Systems von seinem aktuellen Zustand auf den 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

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

Dockerfile und Docker Compose

Provisionierung mit Dockerfile und Docker Compose

Deployment-Ebenen

Ebene 3: Applikation
Deployment-Einheiten, Daten, Cron-Jobs, ...

Ebene 2: Software-Infrastruktur
Server, virtuelle Maschinen, Bibliotheken, ...

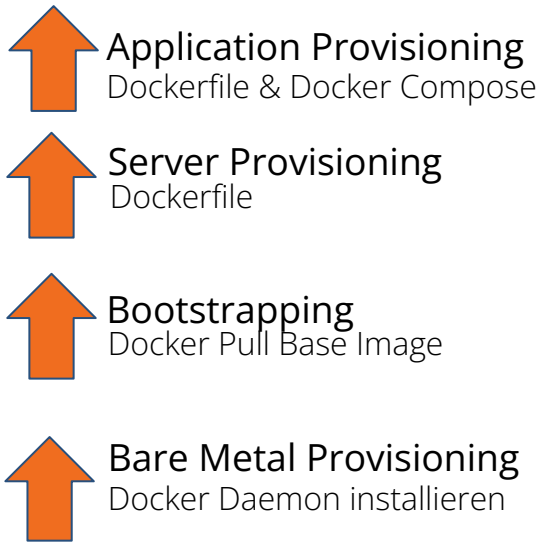
Ebene 1: System-Software
Virtualisierung, Betriebssystem, ...

Docker-Image-Kette

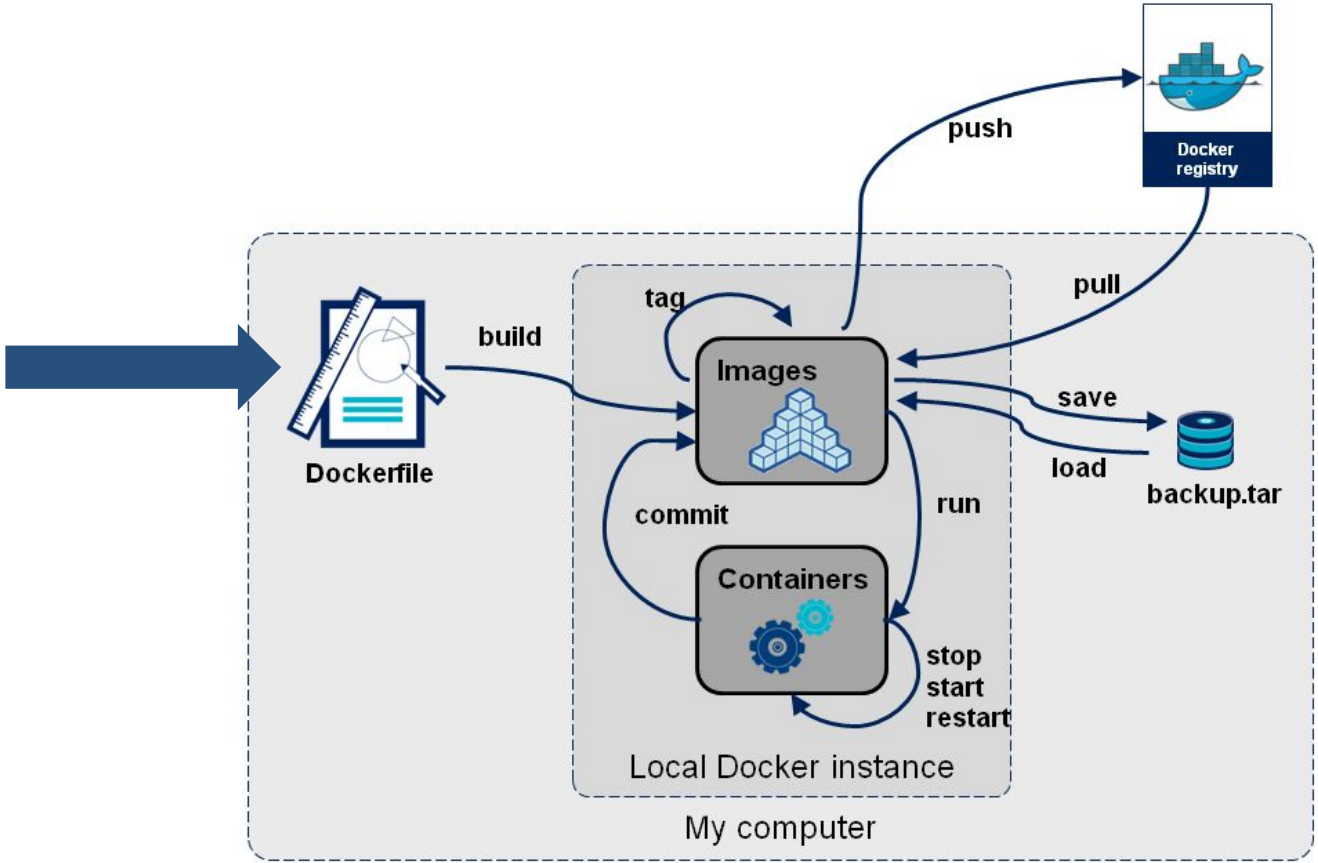
Applikations-Image
(z.B. www.qaware.de)

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> . [-f <Dockerfile>]`

Das Dockerfile wird zum Bau des Image verwendet

```
FROM centos:7.9.2009@sha256:be65f488b7764ad3638f236b7b515b3678369a5124c47b8d32916d6487418ea4

RUN yum install -y epel-release-7-11 \
 && yum install -y nginx-1.12.2-3.el7 php-5.4.16-48.el7 php-fpm-5.4.16-48.el7 \
 && yum clean -y all \
 && rm -rf /var/cache \

 && 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;'
```

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 –e command line parameter of `docker run`.
ENTRYPOINT <command>	The process to be started at container startup
CMD <command>	Default set of arguments passed to the process from ENTRYPOINT (Can be overwritten)
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

https://docs.docker.com/reference/dockerfile/

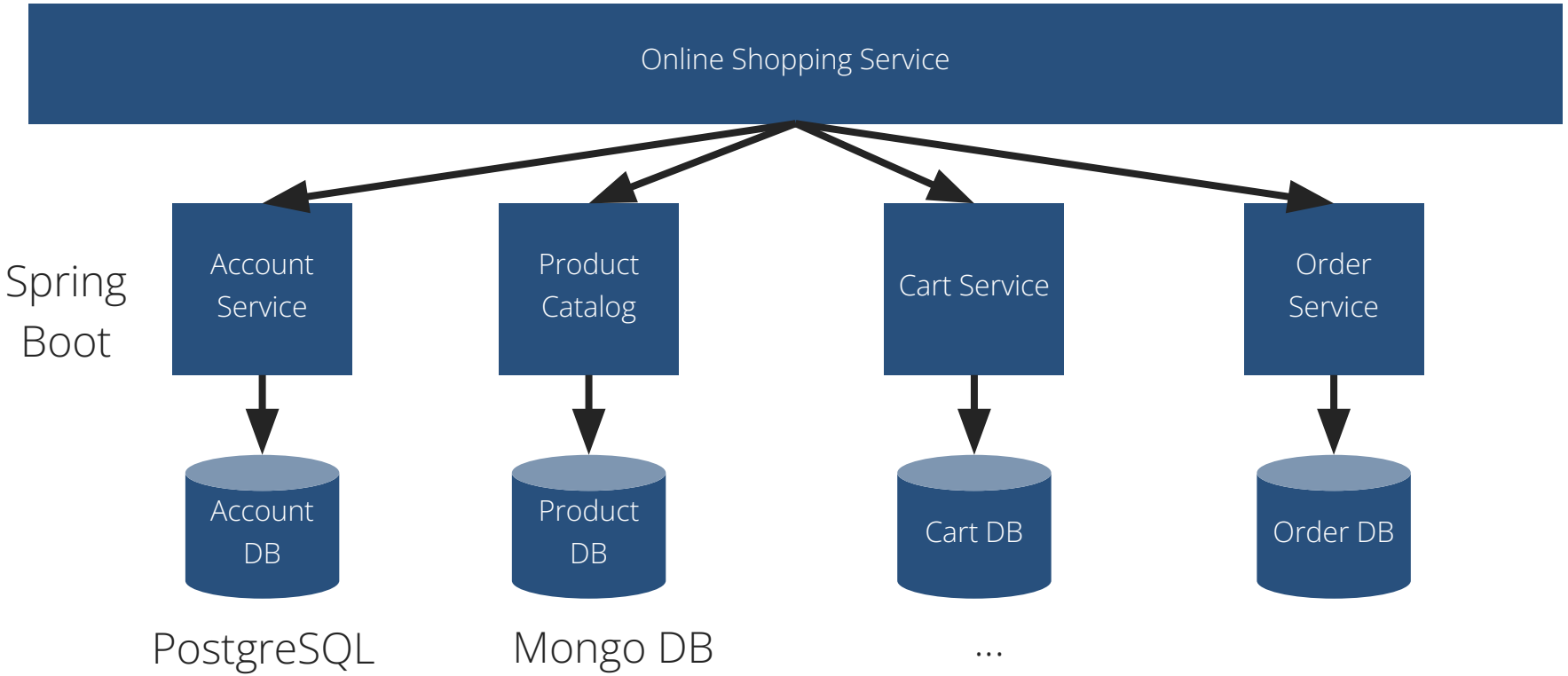
Dockerfile ENTRYPOINT vs CMD

```
FROM alpine
ENTRYPOINT ["echo"]
CMD ["Hello, World!"]
```

```
docker run my-image
# Ausgabe: Hello, World!
```

```
docker run my-image "Hi there!"
# Ausgabe: Hi there!
```

Was machen wir mit Multi-Container Applikationen?



Docker Compose

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

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

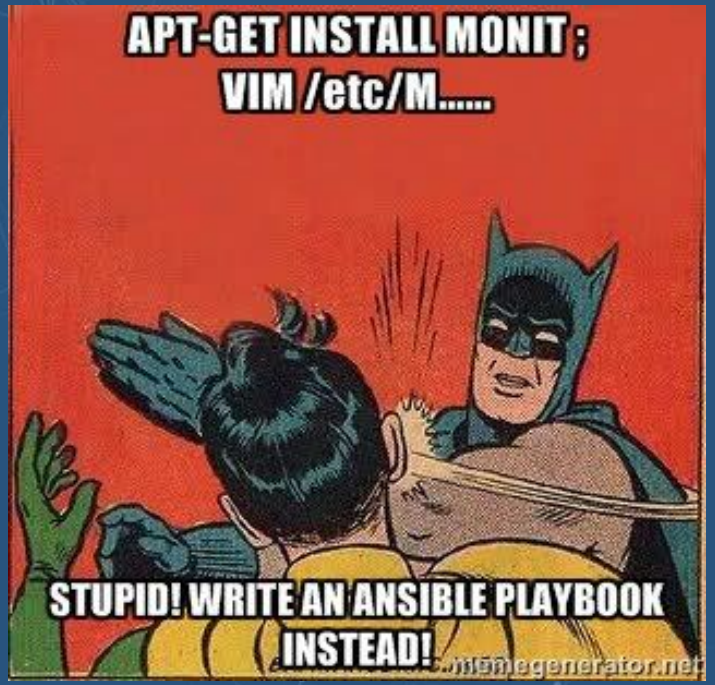
Nutzung von Docker Compose für Multi-Container Apps

```
$ docker compose build

$ docker-compose up -d

$ docker-compose stop

$ docker-compose rm -s -f
```

docker-compose.yml

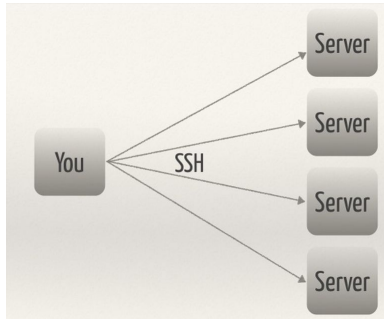
```
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - "redis:cache"
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Ansible


APT-GET INSTALL MONIT ;
VIM /etc/M.....
STUPID! WRITE AN ANSIBLE PLAYBOOK
INSTEAD!
memegenerator.net

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
- Ist einfach zu erlernen im Vergleich zu anderen Lösungen
- Deklarativer Stil
- Umfangreiche Bibliothek vorgefertigter Provisionierungs-Aktionen inkl. Community-Funktion (https://galaxy.ansible.com)



Provisionierung mit Ansible

Deployment-Ebenen

Ebene 3: Applikation
Deployment-Einheiten, Daten, Cron-Jobs, ...

Ebene 2: Software-Infrastruktur
Server, virtuelle Maschinen, Bibliotheken, ...

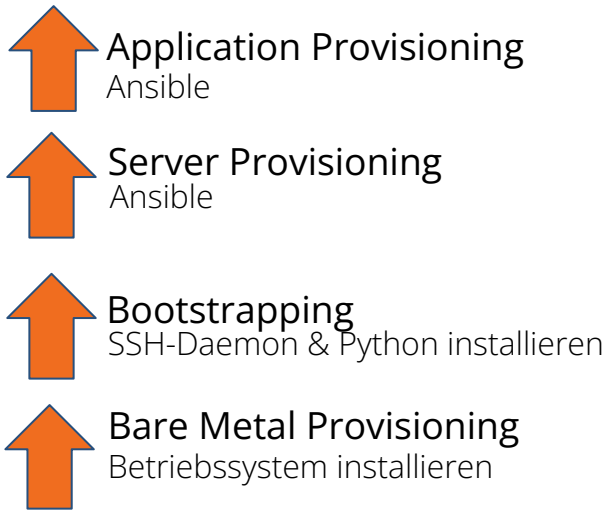
Ebene 1: System-Software
Virtualisierung, Betriebssystem, ...

VM-Kette

Applikation
(z.B. www.qaware.de)

Server
(z.B. NGINX)

Betriebssystem
(z.B. Ubuntu)



Ansible – Konzepte & Begriffe

Inventory

Modules

Tasks

Roles

Playbook

Beschreibung der Maschinen über IP, Shortnames oder URLs

hosts

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

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

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

Gruppen fassen mehrere Maschinen zusammen

Definition von Variablen für einzelne Hosts oder Gruppen

Ansible – Konzepte & Begriffe

Inventory

Modules

Tasks

Roles

Playbook

- Module erlauben die Interaktion über Ansible
- Man kann
  - selbst Module schreiben
  - offizielle Ansible Module nutzen (Core), diese sind schon Teil von Ansible
  - Module 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

- Inventory
- Modules
- Tasks
- **Roles**
- Playbook

roles/webserver/tasks/main.yml

```
- import_tasks: redhat.yml
  when: ansible_facts['os_family']|lower == 'redhat'
- import_tasks: debian.yml
  when: ansible_facts['os_family']|lower == 'debian'
```

roles/webserver/tasks/redhat.yml

```
- ansible.builtin.yum:
    name: "httpd"
    state: present
```

roles/webserver/tasks/debian.yml

```
- ansible.builtin.apt:
    name: "apache2"
    state: present
```

Ansible – Konzepte & Begriffe

Inventory

Modules

Tasks

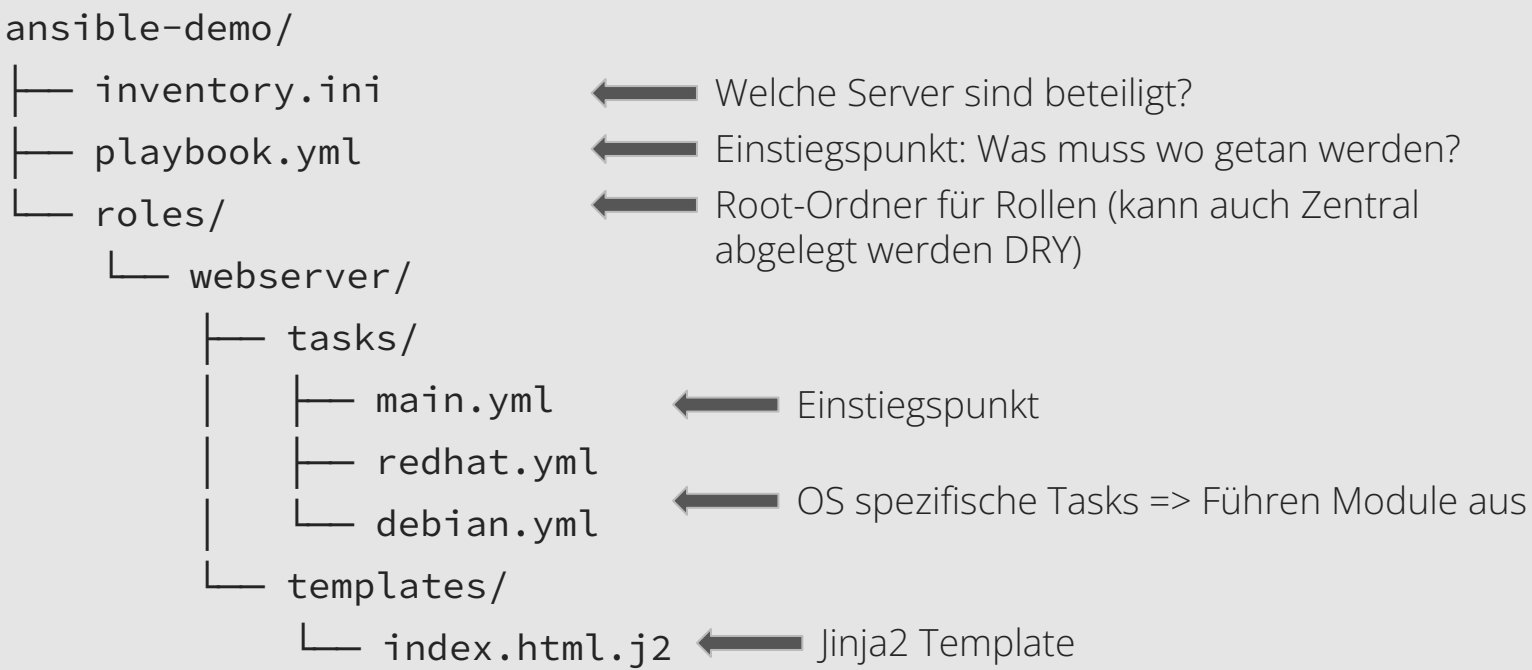
Roles

Playbook

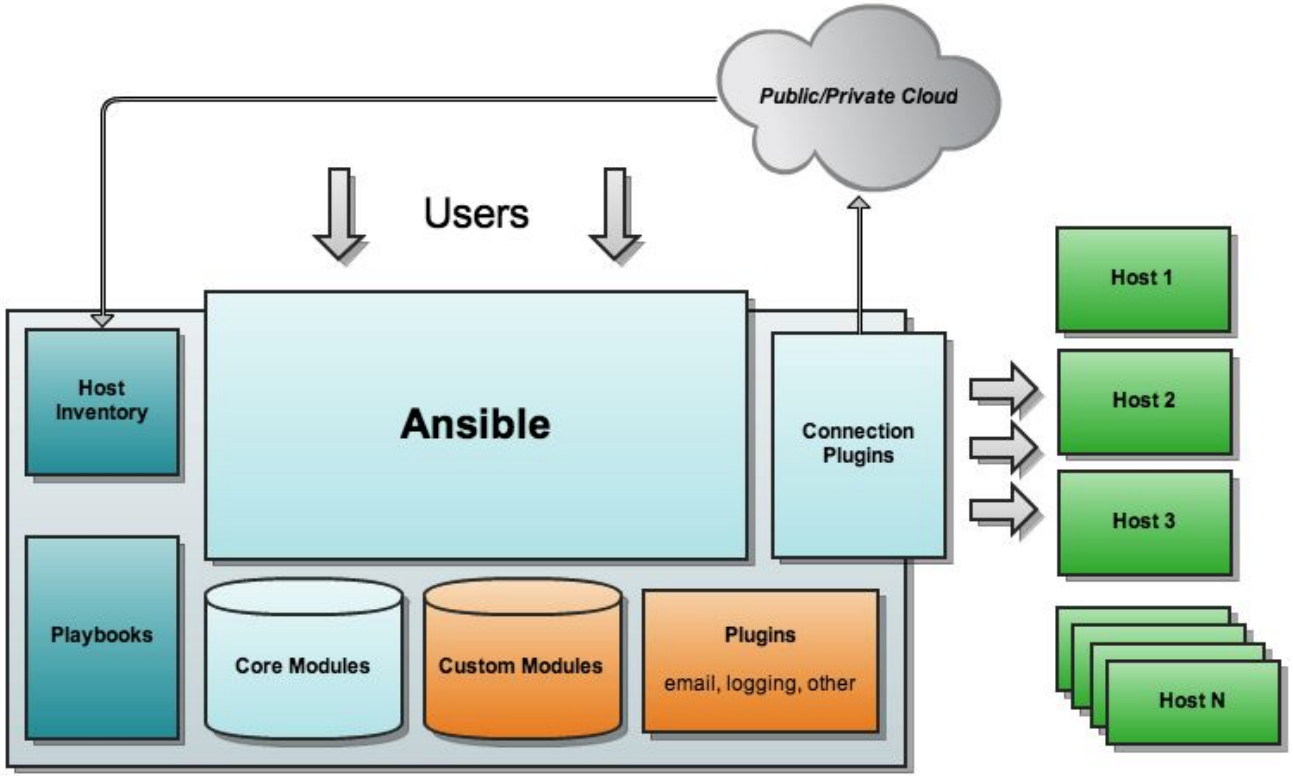
- Playbooks als Basis für Config Management & Orchestrierung

```
- 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
  roles:
      - webserver
```

Ansible Überblick



Architektur von Ansible



Vorgefertigte Module und Plugins

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/latest/collections/index_module.html
https://docs.ansible.com/ansible/latest/collections/all_plugins.html

Die Provisionierung wird über die Kommandozeile gesteuert

Ad-hoc Kommandos:

- `ansible <host-group> -i <inventory-file> -m <module> -a "<arguments>" -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>`

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://developer.hashicorp.com/packer/docs/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

```
packer {
  required_plugins {
    docker = {
      source = "github.com/hashicorp/docker"
      version = ">= 1.0.9"
    }
  }
}

source "docker" "debian" {
  image = "debian:bookworm"
  commit = true
}

…
```

```
…

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

Quelle: https://developer.hashicorp.com/packer/tutorials/docker-get-started/docker-get-started-provision