qaware.de

Technische Hochschule Rosenheim

QA|WARE
SOFTWARE ENGINEERING

# Chapter 4: Provisioning

# Short recap: Virtualization

# Virtualization

**Virtualization**: the creation of virtual realities and their mapping onto physical reality.

<u>Purpose:</u>

- **Multiplicity:** Creation of multiple virtual realities within a single physical reality
- **Decoupling:** Dissolve the bond and dependency on reality
- **Isolation:** Avoiding physical side effects between virtual realities

# Virtualization types

**Virtualization** is representative of several fundamentally different concepts and technologies.

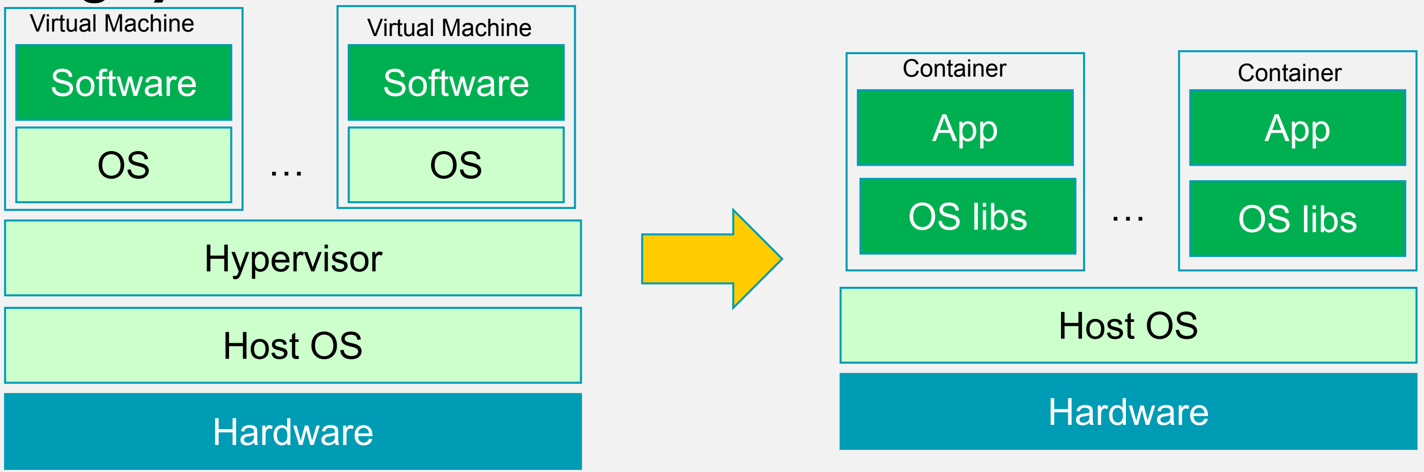
## Virtualization of hardware infrastructure

1. Emulation
2. Full virtualization (Type 2 virtualization)
3. Para virtualization (Type 1 virtualization)

## Virtualization of software infrastructure

4. Operating system virtualization (*Containerization*)
5. Application virtualization (*Runtime*)

# Operating system virtualization



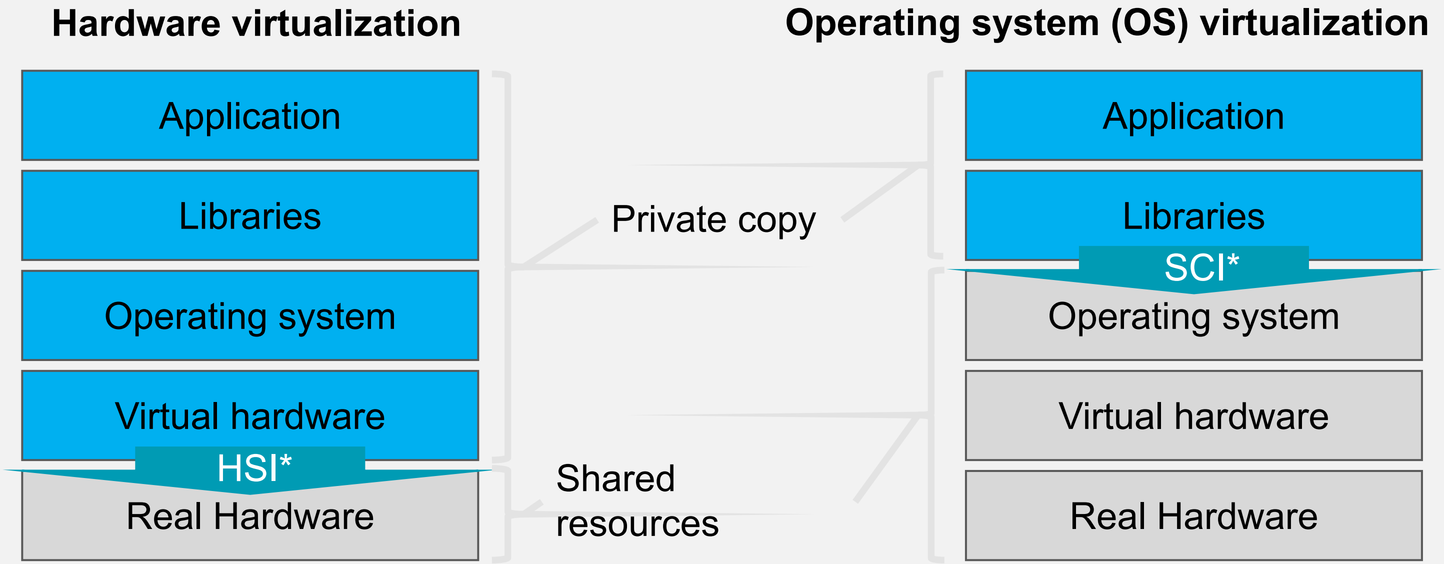
Lightweight virtualization approach: There is no hypervisor. Each app runs directly as a process in the host operating system. However, this is maximally isolated by corresponding OS mechanisms (e.g. Linux LXC).

- Isolation of the process through kernel namespaces (regarding CPU, RAM and disk I/O) and containments
- Isolated file system
- Separate network interface

CPU/RAM overhead generally not measurable (~ 0%)

Startup time = start duration for the first process

# Hardware- vs. Operating system virtualization



- Better insulation
- Higher security

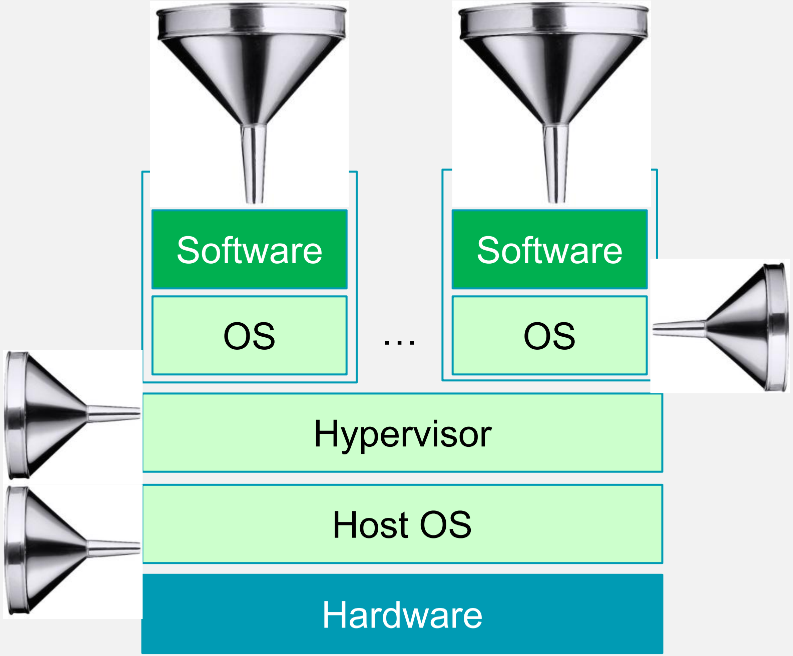
- Smaller private copy volume
- Lower overhead
- Faster startup time

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

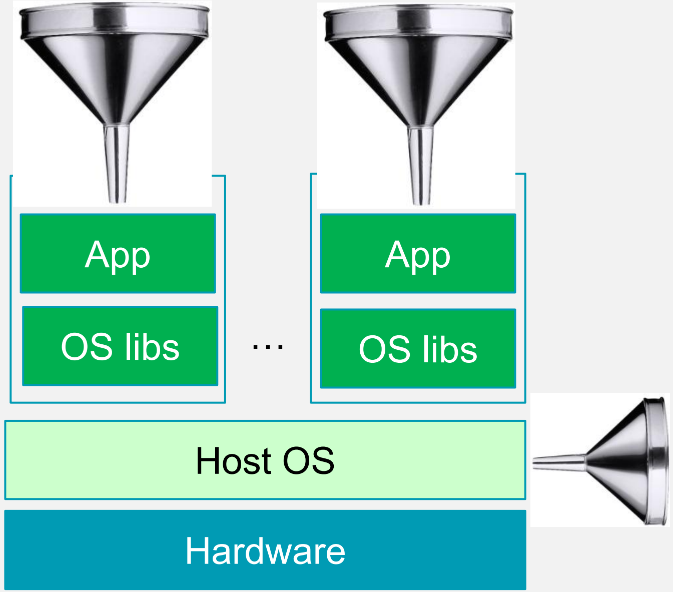
# Provisioning

QA|WARE

# Provisioning: How does software get into the boxes?



Hardware Virtualization



OS Virtualization

Provisioning is the term used to describe the automated provision of IT resources.

# A brief history of system administration.

## Without Virtualization (before 2000)

- Manual installation of operating system on dedicated hardware
- Manual installation of infrastructure software
- Manual / partially automated / automated installation of application software via installer, script, proprietary solutions

## Virtualization of individual machines (2000 – today)

- Manual installation of virtual machines
- Manual installation of infrastructure software
- Manual / partially automated / automated installation of application software via installers, scripts, proprietary solutions

# A brief history of system administration.

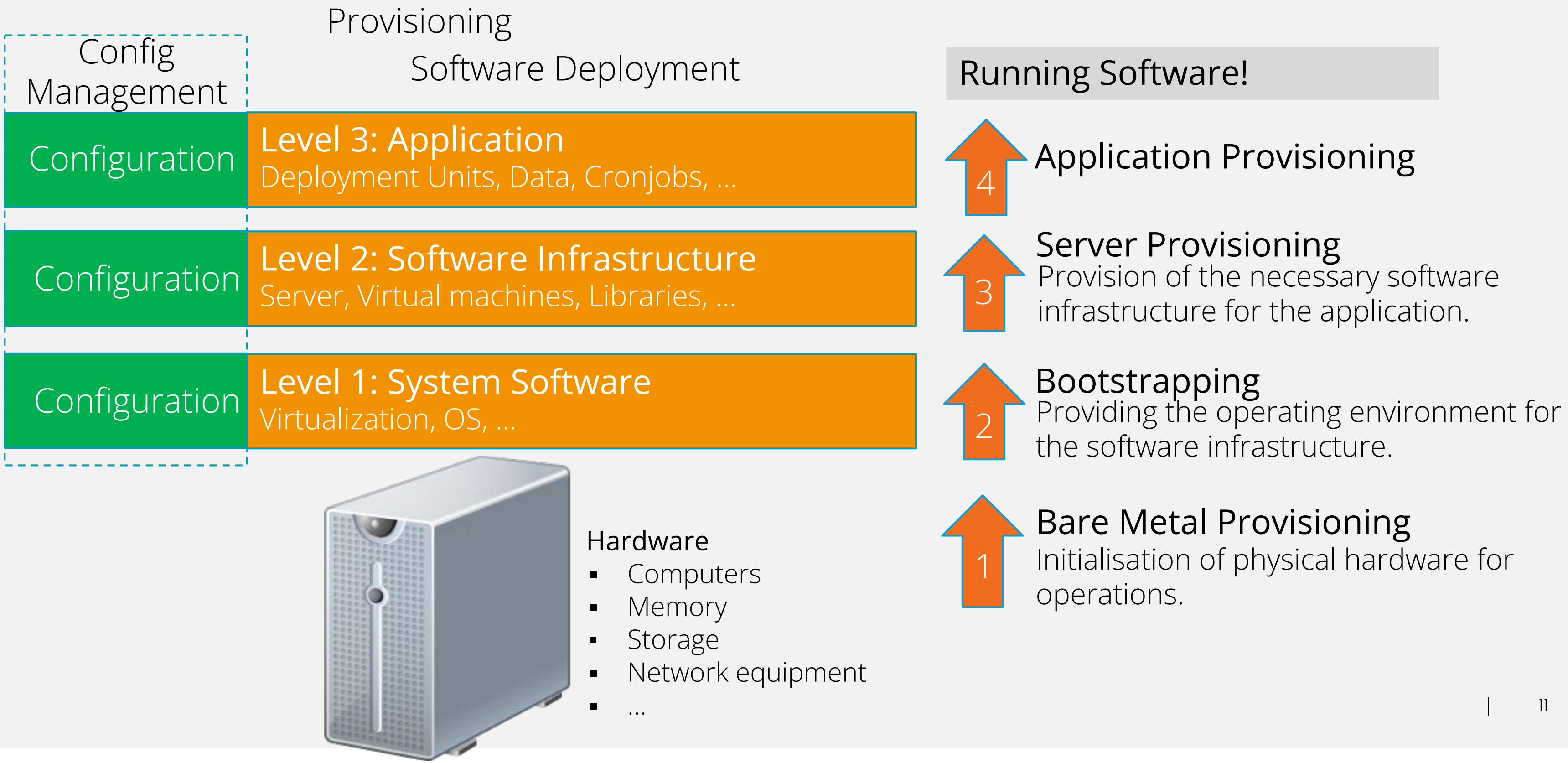
## Virtualization in the Cloud (since 2010)

- Automatic provision of pre-built virtual machines and containers
- Manual installation of infrastructure software only once in the clone master image
- Provision of a defined environment at the push of a button

## Infrastructure-as-Code (2010 – today)

- Programming of provisioning and other operational procedures
- Code-based and under version control

# Provisioning takes place on three different levels and in four stages.



Config Management

Provisioning

Software Deployment

Configuration | **Level 3: Application** Deployment Units, Data, Cronjobs, …

Configuration | **Level 2: Software Infrastructure** Server, Virtual machines, Libraries, …

Configuration | **Level 1: System Software** Virtualization, OS, …

Hardware

- Computers
- Memory
- Storage
- Network equipment
- …

Running Software!

4 Application Provisioning

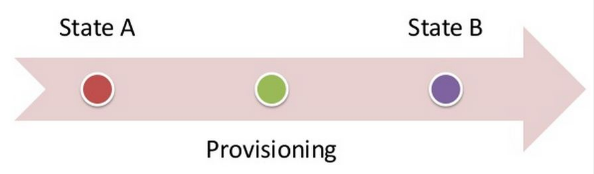
3 Server Provisioning
Provision of the necessary software infrastructure for the application.

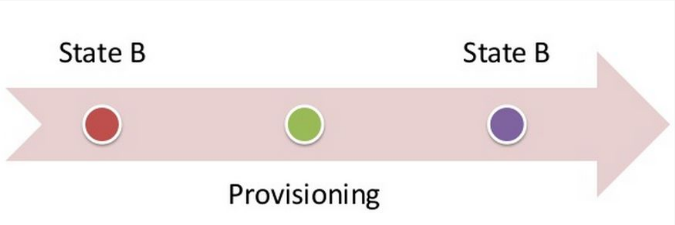
2 Bootstrapping
Providing the operating environment for the software infrastructure.

1 Bare Metal Provisioning
Initialisation of physical hardware for operations.

# Conceptual considerations for provisioning.

**System status** := The totality of software, data and configurations on a system.
**Provisioning** := Transfer from a system's current state to a target state.





What a provisioning mechanism has to do:

1. Determine the initial state
2. Check preconditions
3. Determine state-changing actions
4. Execute state-changing actions
5. Check postconditions and reset state if necessary

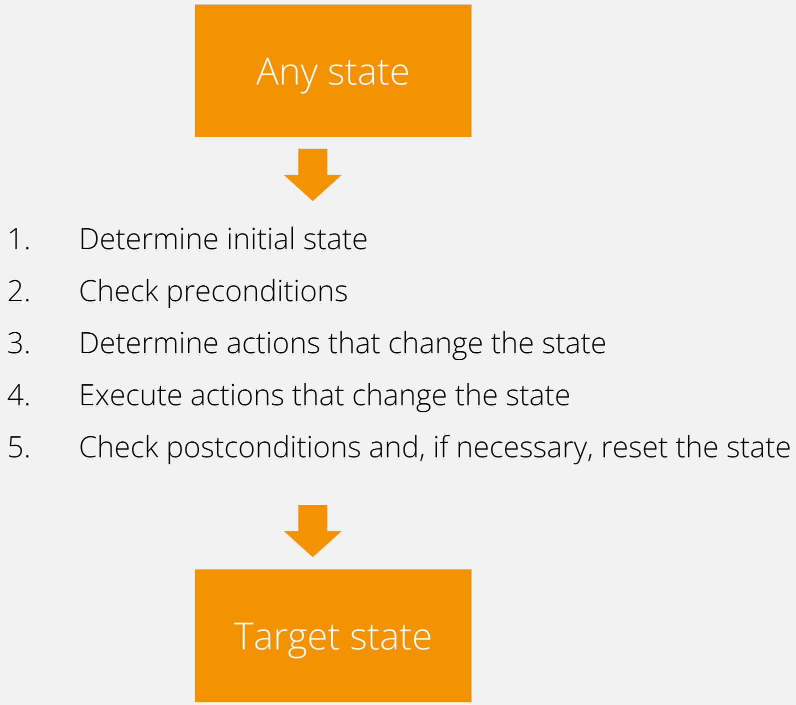
Guarantees

**Idempotency**: The ability of an action to produce the same result whether it is performed once or multiple times.

**Consistency**: After the actions have been carried out, the system state is consistent, regardless of whether individual, several or all actions have failed.

# The new lightness of being.

## Old Style

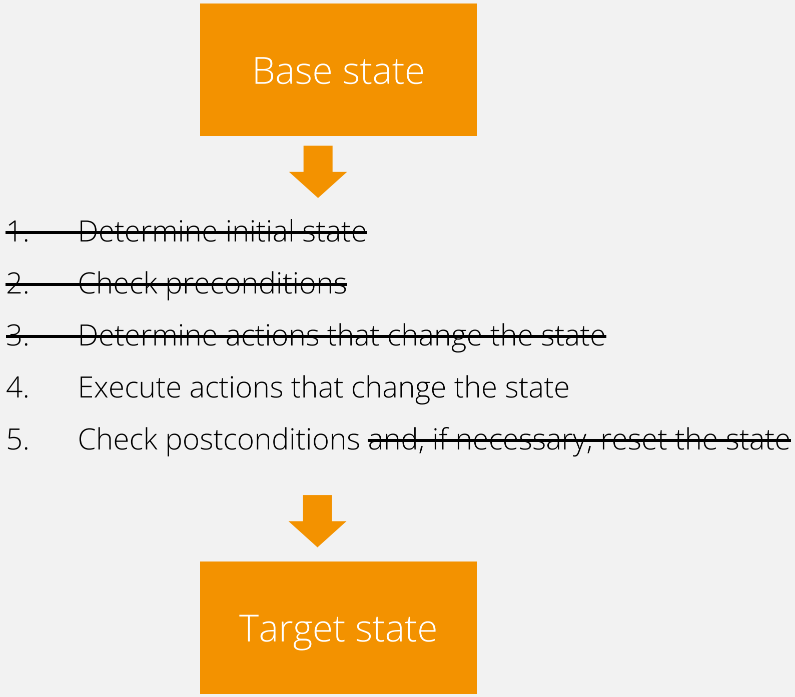


Any state

1. Determine initial state
2. Check preconditions
3. Determine actions that change the state
4. Execute actions that change the state
5. Check postconditions and, if necessary, reset the state

Target state

## New Style
„Immutable Infrastructure / Phoenix Systems“



Base state

~~1. Determine initial state~~

~~2. Check preconditions~~

~~3. Determine actions that change the state~~

4. Execute actions that change the state
5. Check postconditions ~~and, if necessary, reset the state~~

Target state

# 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**.

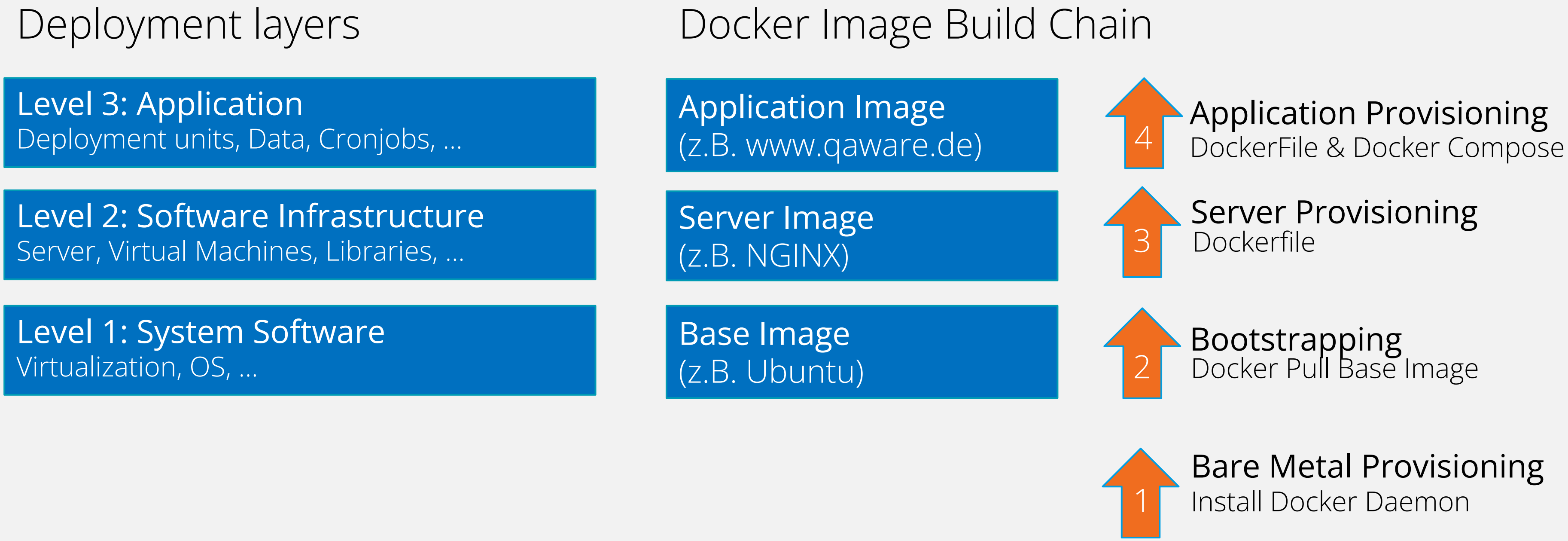
The benefits of an immutable infrastructure include **more consistency and reliability** in your infrastructure and a **simpler, more predictable deployment process**. It mitigates or entirely **prevents** issues that are common in mutable infrastructures, like **configuration drift and snowflake servers**. 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

# Dockerfiles and Docker Compose

QA|WARE

# Provisioning with Dockerfile and Docker Compose



## Deployment layers

**Level 3: Application**
Deployment units, Data, Cronjobs, …

**Level 2: Software Infrastructure**
Server, Virtual Machines, Libraries, …

**Level 1: System Software**
Virtualization, OS, …

## Docker Image Build Chain

**Application Image**
(z.B. www.qaware.de)

**Server Image**
(z.B. NGINX)

**Base Image**
(z.B. Ubuntu)

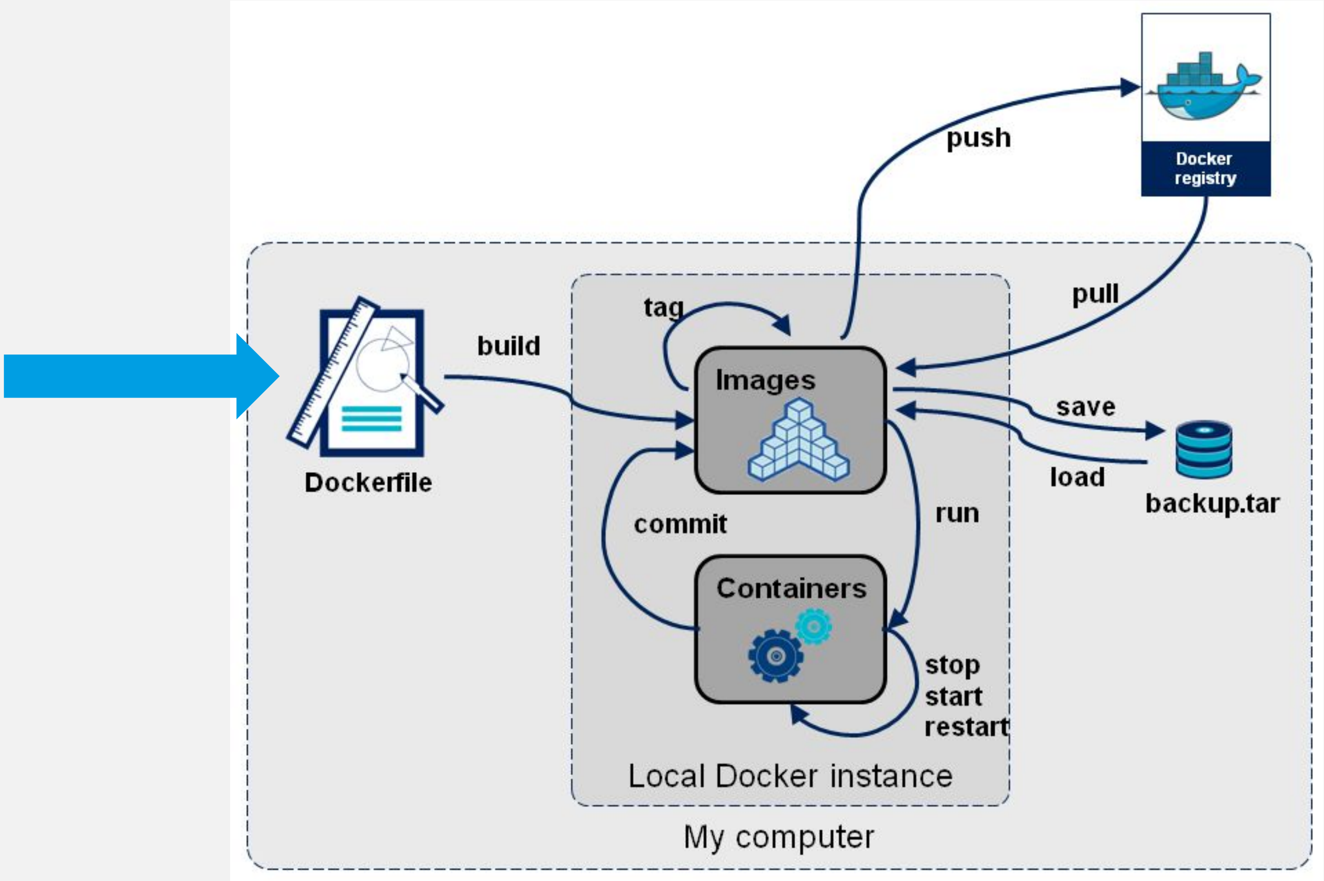
4 **Application Provisioning**
DockerFile & Docker Compose

3 **Server Provisioning**
Dockerfile

2 **Bootstrapping**
Docker Pull Base Image

1 **Bare Metal Provisioning**
Install Docker Daemon

# Provisioning of Images with a Dockerfile.



# Provisioning of Images with a Dockerfile.

A Dockerfile generates a new image based on another image. This automates the following actions:

- Configuration of the image and the resulting containers
- Execution of provisioning actions

A Dockerfile is thus an image representation as an alternative to a physical image (a building share vs. a building component).

- Repeatability in the construction of containers
- Automated creation of images without having to distribute them
- Flexibility in the configuration and in the software versions used
- Simple syntax and therefore easy to use

Command: `docker build -t <target_image_name> <Dockerfile>`

# The Dockerfile is used to build the image.

```
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
```

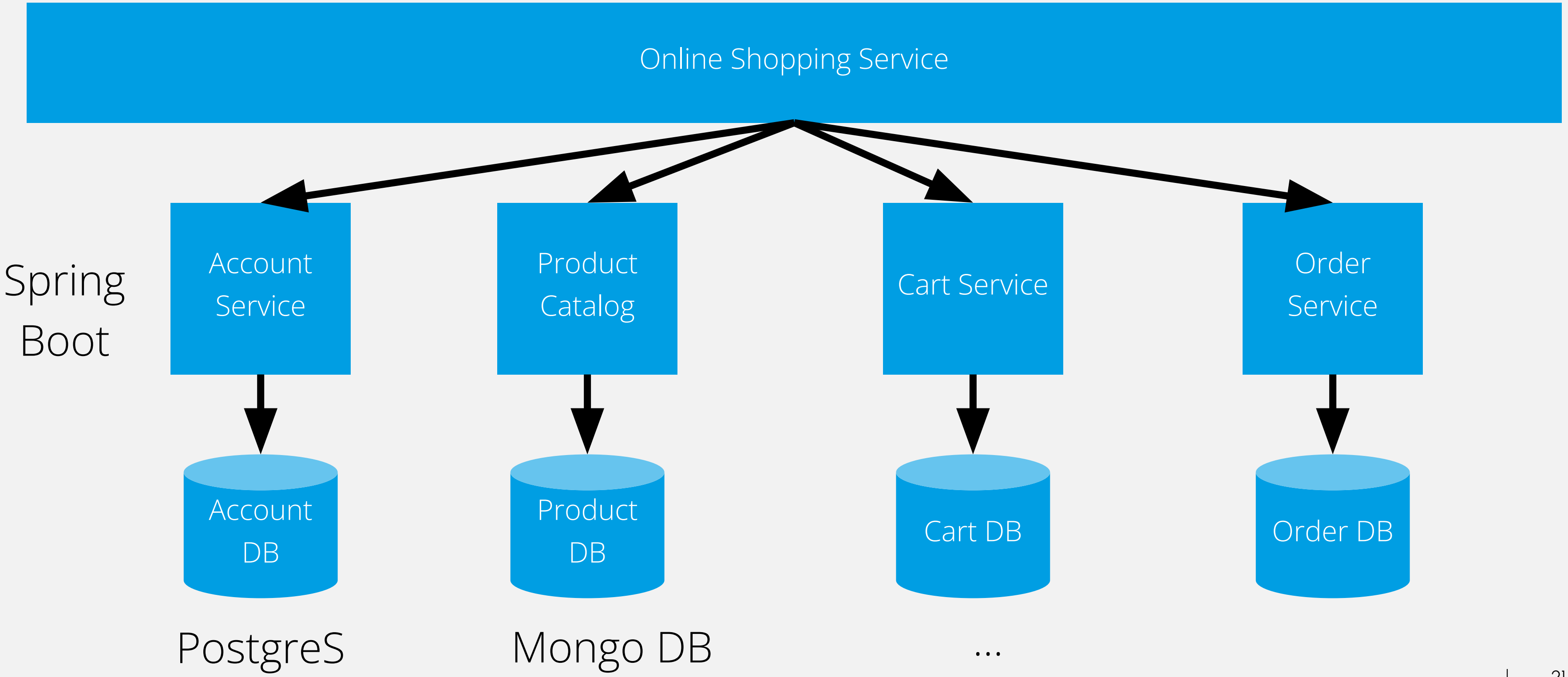
# Recap: Dockerfile commands

| 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 `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. Mostly for documentation purposes |
| USER <name> | Sets the user for all container commands |

http://docs.docker.com/engine/reference/builder

# What do we do with multi-container applications?

QA|WARE



# Docker Compose /1

*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

When using Docker Compose, you essentially follow these three steps:

1. For all your own application components, you write a Dockerfile. For all third-party components, you look for the appropriate image.
2. All services/components that make up the application are defined in the docker-compose.yml. This ensures that they are executed in the same isolated environment.
3. You can then use `docker compose up` to start all components at once.

Additional comfort compared to Docker:

- Multiple instances of the same isolated environment can be started on the same host (e.g. interesting for build servers)
- Data in mounted volumes is retained even after a restart
- Only images that have actually changed are rebuilt when a restart occurs
- Configuration via variables possible

In practice, the main areas of application are:

- Local development
- Automated testing

# Using Docker Compose for multi-container apps.

docker-compose.yml

$ docker compose build

$ docker compose up -d

$ docker compose stop

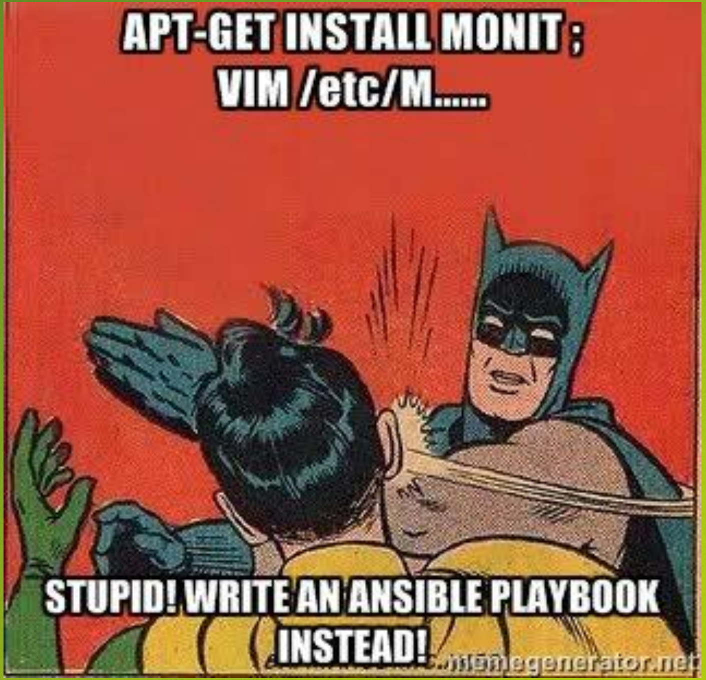
$ docker compose rm –s -f

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

# Exercise 1: Docker and Docker Compose

QA|WARE

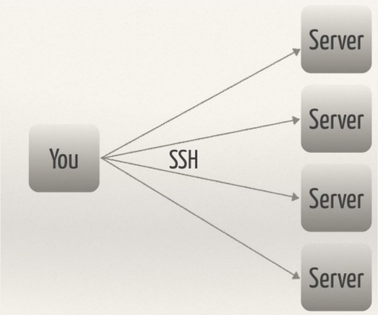
# Ansible


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

# Ansible

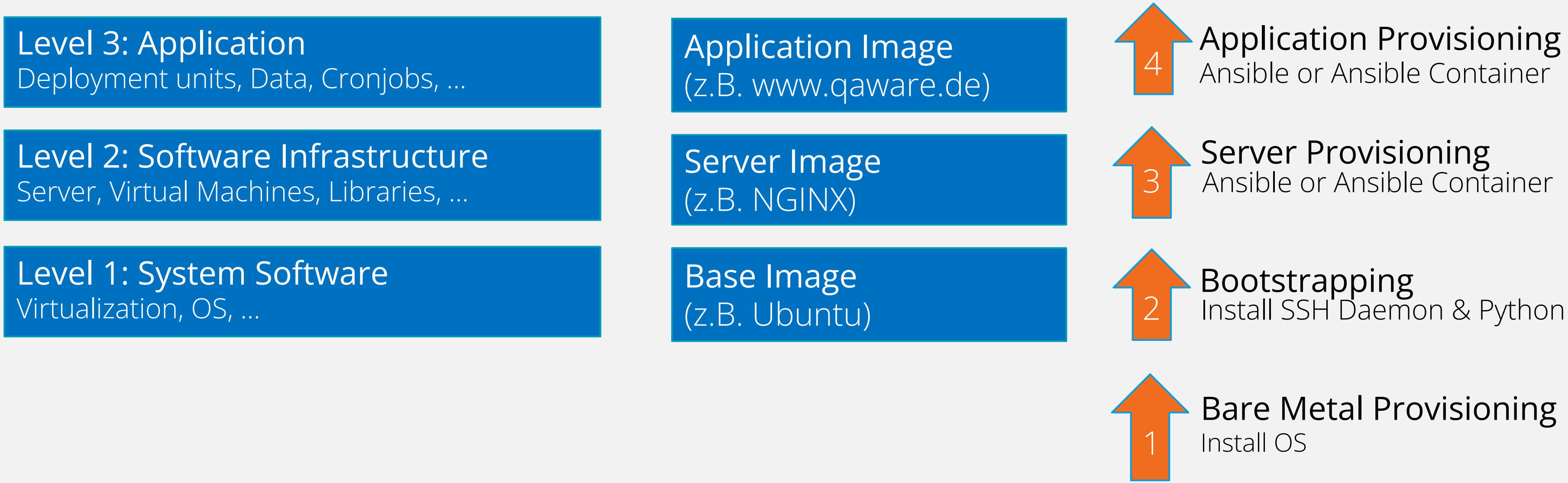
- Red Hat's open-source provisioning tool
- Designed for provisioning large heterogeneous IT landscapes
- Developed in the Python language
- Push principle: Unlike other solutions, it requires neither an agent on the target computers (SSH & Python is sufficient) nor a central provisioning server
- ansible-container variant for provisioning containers
- Is easy to learn compared to other solutions. Declarative style.
- Extensive library of ready-made provisioning actions, including a community function.

(https://galaxy.ansible.com) und Beispielen (https://github.com/ansible/ansible-examples)



# Provisioning with Ansible

QA|WARE



## Deployment layers

**Level 3: Application**
Deployment units, Data, Cronjobs, …

**Level 2: Software Infrastructure**
Server, Virtual Machines, Libraries, …

**Level 1: System Software**
Virtualization, OS, …

## Docker image or VM chain

**Application Image**
(z.B. www.qaware.de)

**Server Image**
(z.B. NGINX)

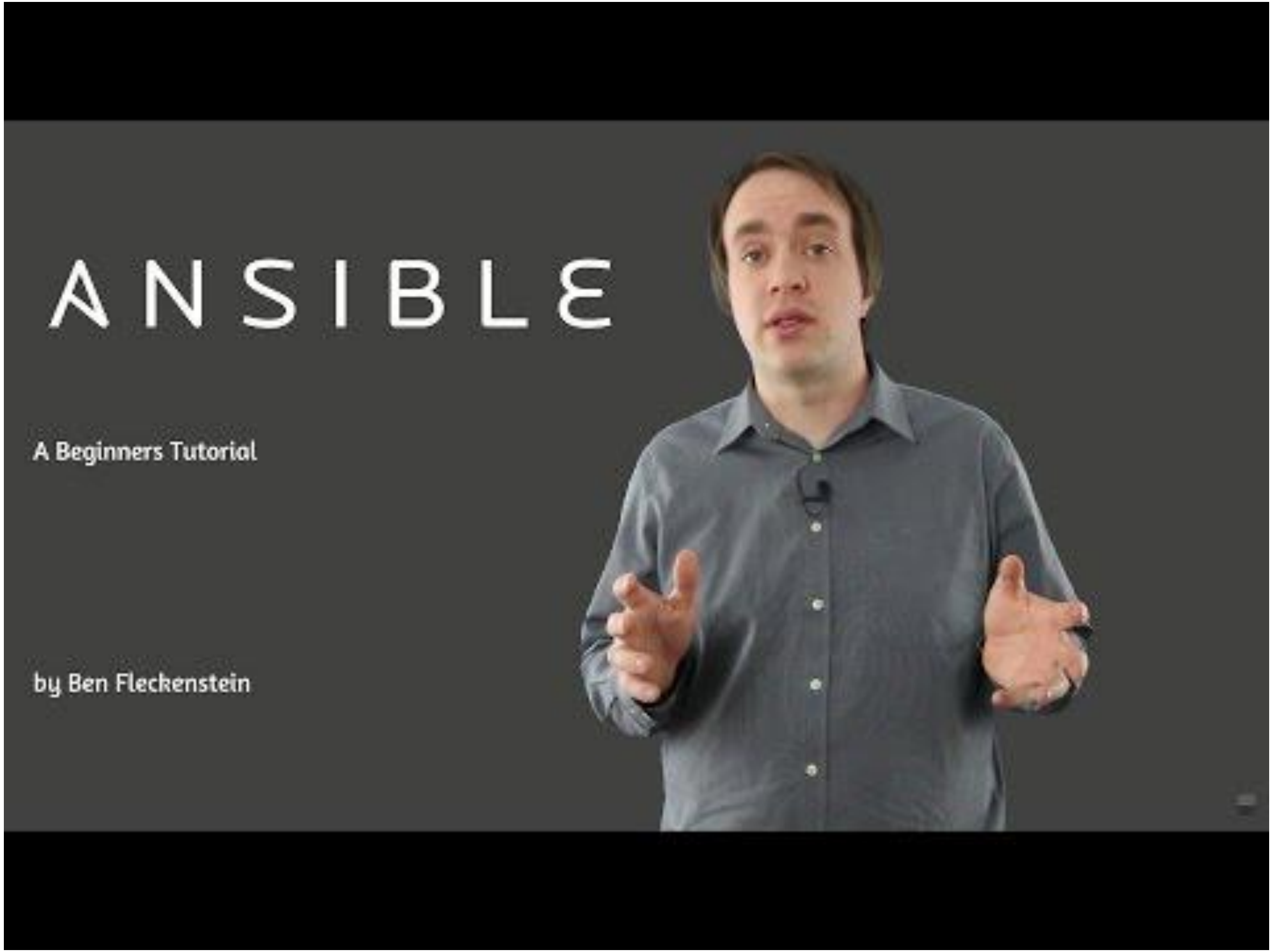
**Base Image**
(z.B. Ubuntu)

4 **Application Provisioning**
Ansible or Ansible Container

3 **Server Provisioning**
Ansible or Ansible Container

2 **Bootstrapping**
Install SSH Daemon & Python

1 **Bare Metal Provisioning**
Install OS



ANSIBLE

A Beginners Tutorial

by Ben Fleckenstein

# Ansible – Concepts & terms

- Inventory
- Modules
- Tasks
- Roles
- Playbook

Description of the machines via IP, short names or URLs

Definition of variables for individual hosts or groups

Groups combine several machines

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

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

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

# Ansible – Concepts & terms

Inventory

Modules

Tasks

Roles

Playbook

- Modules allow interaction via Ansible:
  - Write own modules
  - Use official Ansible Modules (Core), they are part of Ansible
  - Use community Modules (Extras)
- Examples:
  - **File handling**: file, copy, template
  - **Remote execution**: command, shell
  - **Package management**: apt, yum

# Ansible – Concepts & terms

- Inventory
- Modules
- Tasks
- Roles
- Playbook

- Each task describes a provisioning action
- Example: Installing packages via `apt`
- In doing so, the task calls a module that implements the current task.

- Execution via ad hoc commands:

```
ansible -m <module> -a <arguments> <server>
```

# Ansible – Concepts & terms

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 – Concepts & terms

Inventory

Modules

Tasks

Roles

Playbook

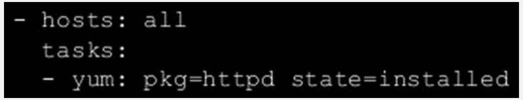
- Playbooks as a base for config management & orchestration

```
- 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
[...]
```

# The most important files to create when provisioning with Ansible.

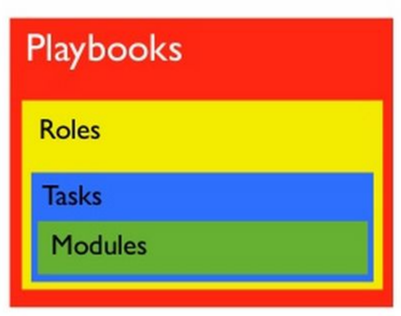
## Playbook (YAML syntax)

Provisioning script.



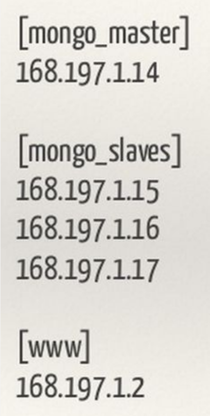
```
- hosts: all
  tasks:
  - yum: pkg=httpd state=installed
```

- *Module* = Implementation of a provisioning action
- *Task* = Description of a provisioning action
- *Role* = Execution of tasks on hosts or host groups

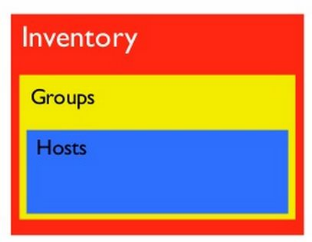


## Inventory

Hosts

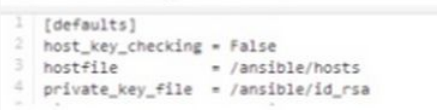


```
[mongo_master]
168.197.1.14

[mongo_slaves]
168.197.1.15
168.197.1.16
168.197.1.17

[www]
168.197.1.2
```



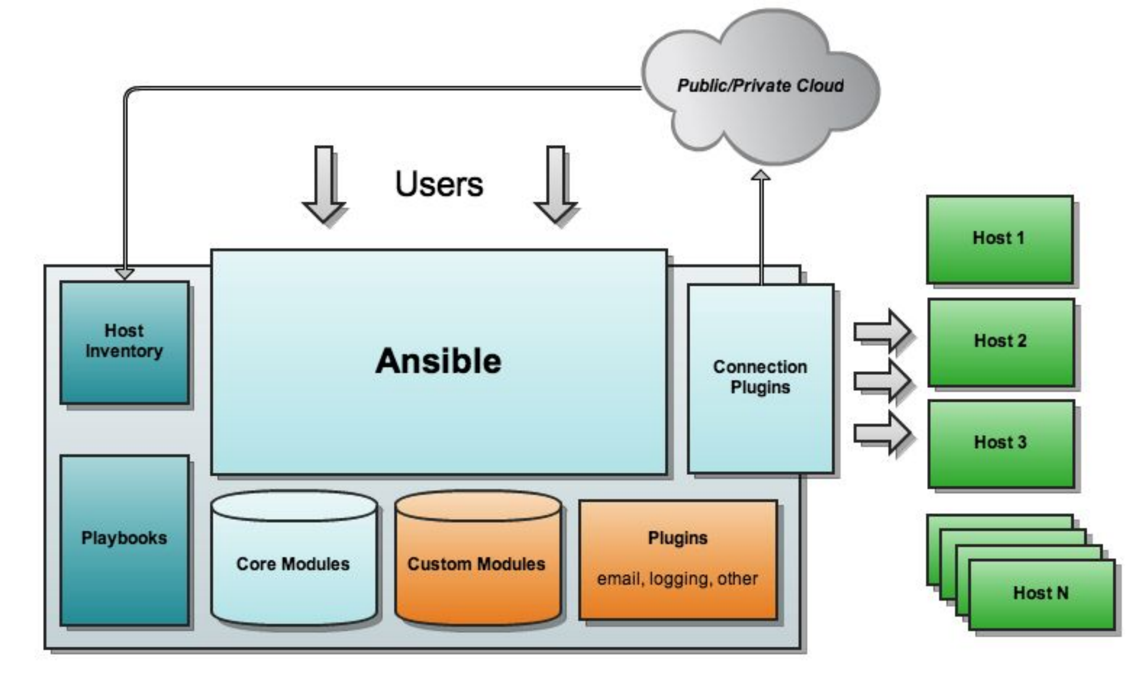
## Ansible configuration

ansible.cfg

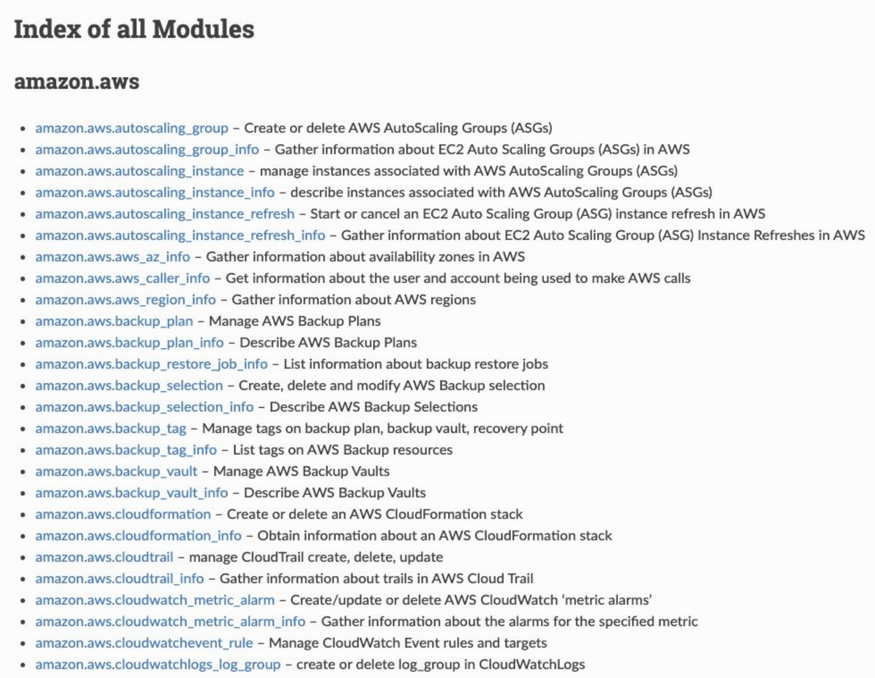


```
[defaults]
host_key_checking = False
hostfile          = /ansible/hosts
private_key_file  = /ansible/id_rsa
```

# Architecture of Ansible



# There are many pre-built modules available in Ansible.



## Index of all Modules

### amazon.aws

- amazon.aws.autoscaling_group – Create or delete AWS AutoScaling Groups (ASGs)
- amazon.aws.autoscaling_group_info – Gather information about EC2 Auto Scaling Groups (ASGs) in AWS
- amazon.aws.autoscaling_instance – manage instances associated with AWS AutoScaling Groups (ASGs)
- amazon.aws.autoscaling_instance_info – describe instances associated with AWS AutoScaling Groups (ASGs)
- amazon.aws.autoscaling_instance_refresh – Start or cancel an EC2 Auto Scaling Group (ASG) instance refresh in AWS
- amazon.aws.autoscaling_instance_refresh_info – Gather information about EC2 Auto Scaling Group (ASG) Instance Refreshes in AWS
- amazon.aws.aws_az_info – Gather information about availability zones in AWS
- amazon.aws.aws_caller_info – Get information about the user and account being used to make AWS calls
- amazon.aws.aws_region_info – Gather information about AWS regions
- amazon.aws.backup_plan – Manage AWS Backup Plans
- amazon.aws.backup_plan_info – Describe AWS Backup Plans
- amazon.aws.backup_restore_job_info – List information about backup restore jobs
- amazon.aws.backup_selection – Create, delete and modify AWS Backup selection
- amazon.aws.backup_selection_info – Describe AWS Backup Selections
- amazon.aws.backup_tag – Manage tags on backup plan, backup vault, recovery point
- amazon.aws.backup_tag_info – List tags on AWS Backup resources
- amazon.aws.backup_vault – Manage AWS Backup Vaults
- amazon.aws.backup_vault_info – Describe AWS Backup Vaults
- amazon.aws.cloudformation – Create or delete an AWS CloudFormation stack
- amazon.aws.cloudformation_info – Obtain information about an AWS CloudFormation stack
- amazon.aws.cloudtrail – manage CloudTrail create, delete, update
- amazon.aws.cloudtrail_info – Gather information about trails in AWS Cloud Trail
- amazon.aws.cloudwatch_metric_alarm – Create/update or delete AWS CloudWatch 'metric alarms'
- amazon.aws.cloudwatch_metric_alarm_info – Gather information about the alarms for the specified metric
- amazon.aws.cloudwatchevent_rule – Manage CloudWatch Event rules and targets
- amazon.aws.cloudwatchlogs_log_group – create or delete log_group in CloudWatchLogs

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

# The provisioning is controlled via the command line.

- Ad-hoc commands:
  - `ansible <host group> -i <inventory-file> -m <module> -a „<arguments>“ -f <parallelism>`
  - Examples:
    - `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`
- Execute playbooks:
  - `ansible-playbook <playbook.yaml>`

# Exercise 2: Ansible

# Packer

# Packer

QA|WARE

*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*

- Written in Go
- Templatizes the building of images
- Existing provisioning scripts (e.g. Ansible) can be reused
- Enables the building of images for multiple platforms with a common configuration

# Packer terminology (https://www.packer.io/docs/terminology)

Artifacts

- The result of a packer build, e.g. a folder of files or a set of AMI IDs

Builds

- Tasks that create an image for a specific platform

Builders

- create a specific image type
- e.g. VirtualBox, Amazon EC2, Docker

Commands

- Subcommands that can be executed with packer, e.g. packer build

Post processors

- Create new artifacts from existing artifacts (e.g. compression, tagging, publishing)

Provisioners

- Install and configure software in a running instance before creating a static artifact from it

Templates

- JSON Files, that configure the Packer Build

# Example

```
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",
    ]
  }
}
```

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

# Packer

https://www.youtube.com/watch?v=r0I4TTO957w

# Exercise 3: Packer (optional)

QA|WARE