ADMIN:DOCS

Ansible Grundlagen: Automatisierung für Linux-Administratoren

Ansible Grundlagen: Agentless Automatisierung mit Playbooks, Inventories, Facts und Variablen für Linux-Admins. Praxisguide für skalierbare Infrastrukturen.

Kategorie: DevOps **Veröffentlicht:** 03.09.2025

In modernen IT-Infrastrukturen gehört die manuelle Konfiguration einzelner Server über interaktive SSH-Sitzungen der Vergangenheit an. Wer Dutzende oder Hunderte Linux-Maschinen verwalten muss, stößt mit klassischen Ad-hoc-Skripten schnell an Grenzen: Fehlende Idempotenz, inkonsistente Paketstände und unübersichtliche Konfigurationsabweichungen (Configuration Drift) führen zu instabilen Umgebungen. Genau an dieser Stelle etabliert sich Ansible als Industriestandard für Configuration Management, Bereitstellung und Orchestrierung.

Ansible verfolgt einen strikt deklarativen und agentenlosen Ansatz. Anstatt eigene Hintergrunddienste (Daemons) auf Zielsystemen vorauszusetzen, stützt sich das System auf etablierte Standardprotokolle: Eine zentrale Steuerinstanz (Control Node) verbindet sich via OpenSSH mit den Zielhosts (Managed Nodes) und führt dort kurzlebige Python-Modul-Payloads aus. Sämtliche Konfigurationszustände werden in lesbaren YAML-Dateien – den sogenannten Playbooks – abgebildet.

Dieser Leitfaden vermittelt die kerntechnischen Grundlagen von Ansible für Systemadministratoren und DevOps-Ingenieure. Von der Architektur und Installation über statische sowie dynamische Inventories, Ad-Hoc-Befehle und Playbooks bis hin zur präzisen Steuerung über Variablen, Facts und Jinja2-Templates werden alle Kernaspekte praxisnah und detailliert erläutert.

Zentrale Leitfragen dieses Artikels:

Quelle: https://admindocs.de/de/devops/ansible-grundlagen-automatisierung-fuer-linux-administratoren

- Architektur und Modell: Warum ermöglicht der agentenlose Ansatz einen deutlich schnelleren Einstieg als Puppet oder Chef?
- Inventarisierung: Wie lassen sich statische Serverlisten im INI- und YAML-Format aufbauen und dynamisch an Cloud-APIs (wie AWS EC2 oder Azure) anbinden?
- Ad-Hoc-Automatisierung: Wann sind einzeilige Befehle sinnvoll und wie steuern die integrierten Kernmodule privilegierte Operationen parallel?
- Playbooks und Idempotenz: Wie werden multi-stufige Software-Deployments mit Handlern, Schleifen und Fehlerbehandlungen reproduzierbar orchestriert?
- Dynamik und Templating: Wie erfassen Ansible Facts Systemzustände zur Laufzeit und wie rendern Jinja2-Vorlagen maßgeschneiderte Konfigurationsdateien?

> ⚠ **Voraussetzungen für den Praxisbetrieb:** Für das Nachvollziehen der praktischen Beispiele wird ein Linux-System (beispielsweise Ubuntu 22.04 LTS oder CentOS/RHEL) als Control Node benötigt. Auf den Zielsystemen (Managed Nodes) müssen ein OpenSSH-Server sowie eine funktionierende Python-3-Laufzeitumgebung installiert sein. Grundlegendes Verständnis der Linux-Kommandozeile, SSH-Schlüsselauthentifizierung sowie die Syntaxregeln von YAML werden vorausgesetzt.

Verwendete Markierungen:

□□ Praxistipps, Hintergrundinformationen und Empfehlungen für effiziente Workflows

⚠ Warnhinweise zu Sicherheitsrisiken, Berechtigungsfallen und Fehlkonfigurationen

□□ Praktische Implementierungsbeispiele mit Befehlen und Konfigurationsdateien

□ Typische Fehlerquellen, Ursachenanalysen und zielgerichtete Lösungsansätze

Ansible Basics: Einstieg in die Automatisierung

Die wiederholte manuelle Ausführung identischer Administrationsbefehle auf mehreren Servern kostet Zeit und birgt erhebliche Fehlerrisiken. Ansible transformiert diese Aufgaben in reproduzierbaren Infrastructure-as-Code (IaC).

Quelle: https://admindocs.de/de/devops/ansible-grundlagen-automatisierung-fuer-linux-administratoren

Architektur und kerntechnische Prinzipien

Ansible wurde ursprünglich von Michael DeHaan entwickelt und wird heute unter dem Dach von Red Hat als Open-Source-Projekt weitergeführt. Es unterscheidet sich in mehreren grundlegenden Aspekten von traditionellen Konfigurationswerkzeugen:

Agentless Architecture (Agentenlosigkeit)

Während Systeme wie Puppet, Chef oder SaltStack in der Regel proprietäre Agenten auf den Zielsystemen voraussetzen – inklusive regelmäßiger Updates, Zertifikatsverwaltung und offener Listener-Ports –, benötigt Ansible auf den Managed Nodes keine dedizierte Software. Ein normaler SSH-Zugang und ein installierter Python-Interpreter (mindestens Version 3.5) genügen. Der Control Node erzeugt zur Laufzeit temporäre Python-Skripte, überträgt diese per SFTP oder SCP auf das Zielsystem, führt sie dort isoliert aus und liest das Ergebnis als strukturiertes JSON-Objekt zurück. Nach Abschluss der Task wird das temporäre Skript auf dem Zielsystem restlos entfernt.

Deklaratives Paradigma und Idempotenz

In klassischen Shell-Skripten wird imperativ definiert, *wie* ein Schritt auszuführen ist (beispielsweise: `apt-get install nginx`). Führt man ein solches Skript mehrfach aus, kann es zu unbeabsichtigten Seiteneffekten oder Fehlern kommen, sofern keine expliziten Vorprüfungen programmiert wurden. Ansible arbeitet deklarativ: In Tasks wird der *gewünschte Zielzustand* beschrieben (beispielsweise: `state: present`). Die zugrunde liegenden Ansible-Module stellen sicher, dass Aktionen nur dann ausgeführt werden, wenn der Ist-Zustand vom Soll-Zustand abweicht. Dieser Grundsatz der Idempotenz garantiert, dass mehrfache Playbook-Durchläufe auf bereits korrekt konfigurierten Systemen keine Änderungen vornehmen (`changed: false`).

Push- statt Pull-Prinzip

Klassische Agenten-Systeme pollen regelmäßig einen zentralen Master-Server (Pull-Modell). Ansible initiiert Verbindungen dagegen aktiv vom Control Node aus (Push-Modell). Dies verleiht Administratoren die vollständige Kontrolle über den genauen Ausführungszeitpunkt von Wartungsfenstern, Patch-Zyklen und Deployments, ohne auf Polling-Intervalle warten zu müssen.

Einsatzbereiche und typische Anwendungsfälle

Quelle: https://admindocs.de/de/devops/ansible-grundlagen-automatisierung-fuer-linux-administratoren

- Konfigurationsmanagement: Vereinheitlichung von Systemdateien (`/etc/ssh/sshd_config`, `/etc/ntp.conf`, Benutzerkonten).
- Software-Deployment: Rollout von Web- und Datenbankservern inklusive reproduzierbarer Startkonfigurationen.
- Orchestrierung: Multi-Tier-Abläufe, bei denen Datenbanken vor Webservern migriert und Lastverteiler temporär aus dem Verbund genommen werden.
- Sicherheits- und Compliance-Audits: Regelmäßige Überprüfung sicherheitskritischer Kernel-Parameter, Berechtigungen und Firewall-Regeln.

Installationsmethoden und Versionsprüfung

Die Installation von Ansible erfolgt ausschließlich auf dem Control Node. Managed Nodes benötigen keine Ansible-Pakete. Je nach eingesetzter Linux-Distribution und operativen Anforderungen stehen verschiedene Installationspfade zur Verfügung.

□□ Praktisches Beispiel: Installation auf Debian und Ubuntu

Auf Systemen unter Debian GNU/Linux oder Ubuntu erfolgt die Installation über das offizielle Paket-Repository. Aktualisiere zunächst die Paketquellen und installiere das Basispaket:

```
sudo apt update && sudo apt upgrade -y
sudo apt install ansible -y
```

Für Umgebungen, die stets die aktuellste Upstream-Version oder gezielte Release-Stände erfordern, empfiehlt sich die Installation über den Python-Paketmanager `pip`:

```
sudo apt install python3-pip python3-venv -y
pip3 install ansible==2.14.0
```

Überprüfe die erfolgreiche Installation und die Umgebungskonfiguration mit dem Versionsbefehl:

```
ansible --version
```

Die Ausgabe liefert wesentliche Diagnoseinformationen über die Laufzeitumgebung:

```
ansible [core 2.14.0]
  config file = None
  configured module search path = ['/home/user/.ansible/plugins/modules',
'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3/dist-packages/ansible
  ansible collection location = ['/home/user/.ansible/collections',
'/usr/share/ansible/collections']
  executable location = /usr/bin/ansible
  python version = 3.10.6 (main, May 29 2023, 11:10:38) [GCC 11.3.0]
  jinja version = 3.0.3
  libyaml = True
```

Erklärung der Ausgabewerte:

- `config file`: Zeigt die aktuell aktive Konfigurationsdatei (`ansible.cfg`). `None` signalisiert, dass Standardwerte verwendet werden.
- `ansible python module location`: Pfad der zugrunde liegenden Python-Bibliotheken von Ansible.
- `python version`: Version des Interpreters auf dem Control Node (erforderlich: Python 3.8+ für moderne Ansible-Core-Releases).
- `jinja version`: Version der Templating-Engine Jinja2, die für Variablen- und Vorlagenersetzungen verwendet wird.

- `libyaml`: Zeigt an, ob der performante C-basierte YAML-Parser geladen werden konnte.

Installation auf RHEL, CentOS und Fedora:

Auf Enterprise-Linux-Derivaten wird zunächst das EPEL-Repository (Extra Packages for Enterprise Linux) aktiviert, bevor Ansible über den Paketmanager `dnf` installiert wird:

```
# Auf RHEL / CentOS Stream
sudo dnf install epel-release -y
sudo dnf install ansible -y

# Auf Fedora
sudo dnf install ansible -y
```

Installation auf macOS (als Entwickler-Control-Node):

```
brew install ansible
```

Isolierte Ausführung über Virtual Environments

Um Versionskonflikte zwischen globalen System-Python-Paketen und projektbezogenen Ansible-Modulen zu vermeiden, empfiehlt sich im professionellen Einsatz die Isolation über `venv`:

```
python3 -m venv ~/ansible-env
source ~/ansible-env/bin/activate
pip install --upgrade pip
```

```
pip install ansible
```

Fehlerbehebung bei der Ersteinrichtung

Falls der Aufruf von `ansible --version` oder grundlegende Kommandos fehlschlagen, sollten folgende Prüfpunkte abgearbeitet werden:

Python-Integrität prüfen:

Stelle sicher, dass `python3 --version` mindestens Python 3.8 ausgibt.

Paketkonflikte isolieren:

Bei paralleler Nutzung von Systempaketen und `pip` kann `pip3 show ansible` aufklären, welche Binärdatei vorrangig im `$PATH` liegt.

Fehlende Authentifizierungsbibliotheken:

Falls Zielsysteme interaktiv per Passwort statt SSH-Schlüssel verwaltet werden müssen, verlangt Ansible das Hilfsprogramm `sshpass`:

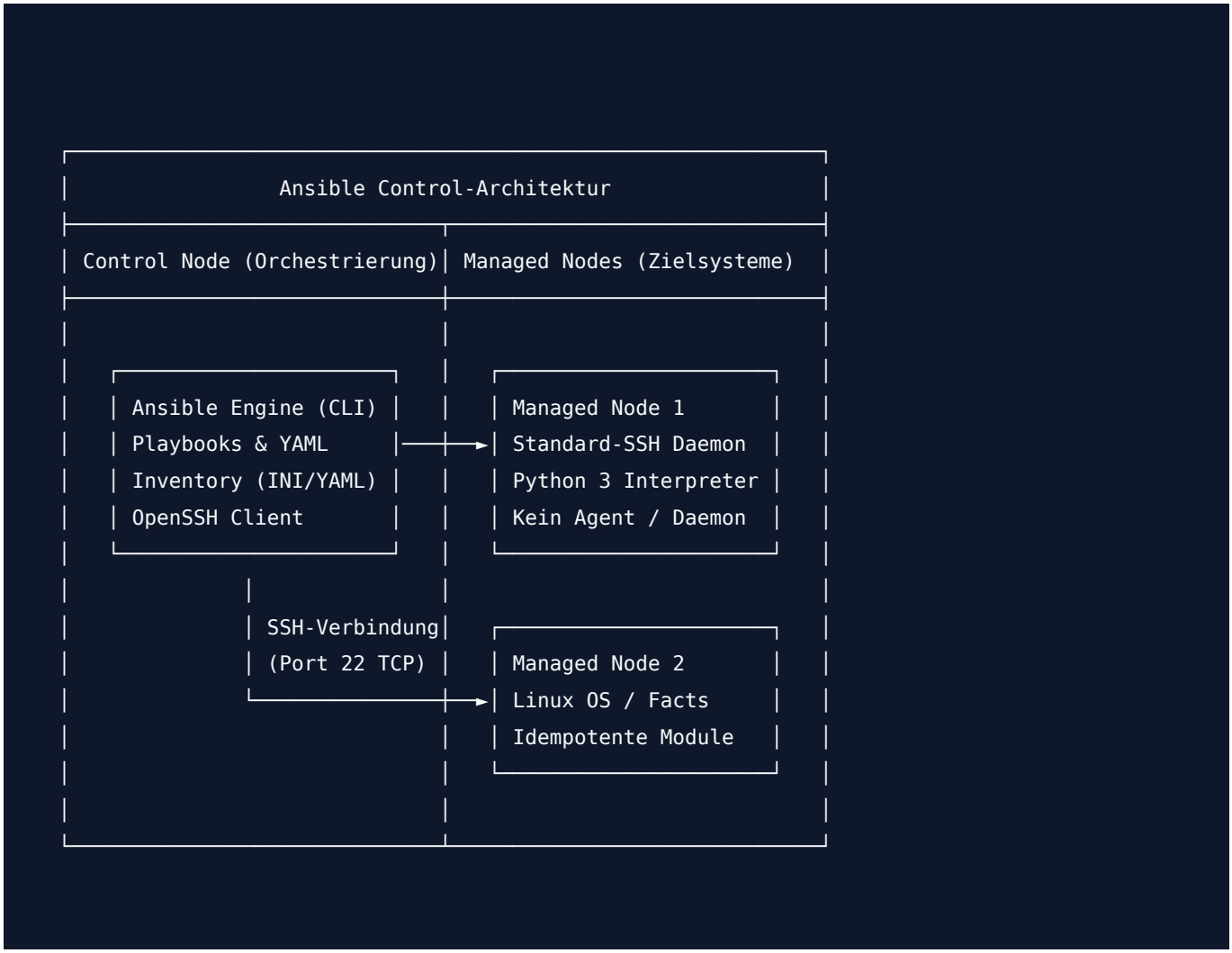
```
sudo apt install sshpass -y
```

Lokalen Loopback-Test ausführen:

```
ansible localhost -m ping
```

Grundlegende Architektur von Ansible

Das Zusammenspiel der Systemkomponenten folgt einer klar strukturierten Arbeitsteilung. Sämtliche Steuerungslogik, Inventare und Playbooks verbleiben zentral auf dem Control Node, während die Managed Nodes ausschließlich passive Ausführungsendpunkte darstellen.



Erste Schritte: Verbindung herstellen

Für die Kontaktaufnahme mit Zielsystemen benötigt Ansible eine Bestandsdatei (Inventory), in der Rechneradressen und Verbindungsparameter definiert sind.

Erstelle eine grundlegende Inventory-Datei namens `inventory.ini`:

```
[webservers]
web1 ansible_host=192.168.1.10 ansible_user=ubuntu
ansible_ssh_private_key_file=~/.ssh/id_rsa
web2 ansible_host=192.168.1.11 ansible_user=ubuntu
ansible_ssh_private_key_file=~/.ssh/id_rsa

[dbservers]
db1 ansible_host=192.168.1.20 ansible_user=ubuntu
ansible_ssh_private_key_file=~/.ssh/id_rsa

[all:vars]
ansible_python_interpreter=/usr/bin/python3
```

SSH-Schlüsselpaar generieren und verteilen:

Sofern noch kein dedizierter SSH-Schlüssel für die Automatisierung vorliegt, generiere ein modernes Ed25519- oder RSA-Schlüsselpaar auf dem Control Node und hinterlege den öffentlichen Schlüssel auf den Zielhosts:

```
ssh-keygen -t rsa -b 4096 -C "ansible-control@admindocs.local"
ssh-copy-id ubuntu@192.168.1.10
ssh-copy-id ubuntu@192.168.1.11
ssh-copy-id ubuntu@192.168.1.20
```

Verbindungstest via Ping-Modul:

Prüfe die Erreichbarkeit und die Funktionsfähigkeit des Python-Subsystems aller definierten Knoten mit dem Ad-hoc-Befehl `ping`:

```
ansible all -i inventory.ini -m ping -u ubuntu
```

Bei erfolgreicher Kommunikation antwortet jeder Knoten mit einem standardisierten JSON-Ergebnis:

```
web1 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
```

Typische Fehlerquellen bei der Ersteinrichtung:

- `UNREACHABLE`: Der Zielhost antwortet nicht auf Port 22 oder der angegebene SSH-Schlüssel wurde abgelehnt. Führe den Befehl mit dem Schalter `-vvv` aus, um den detaillierten SSH-Handshake einzusehen.
- `MODULE FAILURE: /usr/bin/python3: not found`: Auf dem Zielsystem ist kein Python installiert. Installiere Python minimal auf dem Zielsystem (`sudo apt install python3-minimal`).
- `Host key verification failed`: Der Fingerabdruck des Zielhosts ist in `~/.ssh/known_hosts` noch nicht registriert. Dies lässt sich für Testumgebungen über `host_key_checking = False` steuern.

Werkzeugvergleich im Infrastrukturumfeld

Die Wahl des Konfigurationswerkzeugs hängt maßgeblich von der Infrastrukturgröße, dem Team-Know-how und den Sicherheitsanforderungen ab:

Werkzeug	Architektur	Konfigurationssprache	Skalierungsmodell	Lernkurve	Primärer Einsatzzweck	B H
Ansible	Agentless (SSH/WinRM)	YAML / Jinja2	Push-basiert	Flach	Linux-Automatisierung, Multi-Cloud-Deployments	S L t
Puppet	Master / Agent	Puppet DSL / Ruby	Pull-basiert (30 min)	Moderat	Enterprise-Compliance, rigides Konfigurationsmanagement	M Z
Chef	Server / Agent	Ruby DSL	Pull-basiert	Steil	Komplexe Cloud-Anwendungs-Infrastrukturen	H P V
SaltStack	Master / Minion (ZeroMQ)	YAML / Python	Push & Event-Driven	Moderat	Sehr große Cluster mit Echtzeitanforderungen	E o
Shell-Skripte	Manuell / SSH-Wrapper	Bash / POSIX Sh	Linear / Imperativ	Niedrig	Schnelle lokale Ad-hoc-Aufgaben	K k F

> ⚠ **Sicherheitshinweis für den SSH-Zugang:** Konfiguriere auf Managed Nodes niemals den direkten administrativen Login via `root` über SSH. Verwende stets einen unprivilegierten Service-Account (beispielsweise `ansible` oder `devops`) mit sudo-Rechten und aktiviere die Berechtigungseskalation über `become: true` im Playbook.

Konfigurationsgrundlagen und Basiseinstellungen

Das Verhalten von Ansible lässt sich über die zentrale Datei `ansible.cfg` granular steuern. Ansible durchsucht dafür mehrere Verzeichnispfade in fester Reihenfolge:

1. Umgebungsvariable `$ANSIBLE_CONFIG`
2. `./ansible.cfg` (im aktuellen Arbeitsverzeichnis)
3. `~/.ansible.cfg` (im Benutzerverzeichnis)
4. `/etc/ansible/ansible.cfg` (systemweite Konfiguration)

Erstelle im Projektverzeichnis eine maßgeschneiderte `ansible.cfg`:

```
[defaults]
inventory = ./inventory.ini
remote_user = ubuntu
host_key_checking = False
forks = 10
timeout = 30
log_path = ./ansible.log

[privilege_escalation]
become = True
become_method = sudo
become_user = root
become_ask_pass = False
```

Erste administrative Paketinstallation:

Mit der gesetzten Konfiguration genügt ein kurzer Befehl, um Nginx auf der Gruppe `webservers` zu installieren:

```
ansible webservers -m apt -a "name=nginx state=present" --become
```

Inventories und Hosts

Die Inventory ist das Herzstück jeder Ansible-Umgebung. Sie definiert nicht nur Hostnamen und IP-Adressen, sondern bildet logische Gruppen, Umgebungsstufen und Variablenstrukturen ab.

Statische Inventories im INI-Format

Das INI-Format hat sich aufgrund seiner Einfachheit und Übersichtlichkeit für viele Linux-Administratoren als primärer Standard etabliert.

Hier ein umfassendes Beispiel für einen Web- und Datenbank-Stack mit Gruppenverschachtelung:

```
# Webserver-Gruppe mit individuellen Verbindungsparametern
[webservers]
web1 ansible_host=192.168.1.10 ansible_user=devops
ansible_ssh_private_key_file=~/.ssh/id_rsa ansible_become=true ansible_become_method=sudo
web2 ansible_host=192.168.1.11 ansible_user=devops
ansible_ssh_private_key_file=~/.ssh/id_rsa ansible_become=true ansible_become_method=sudo

# Datenbank-Gruppe mit systemspezifischen Variablen
[dbservers]
db1 ansible_host=192.168.1.20 ansible_user=devops
ansible_ssh_private_key_file=~/.ssh/id_rsa ansible_become=true
db2 ansible_host=192.168.1.21 ansible_user=devops
ansible_ssh_private_key_file=~/.ssh/id_rsa ansible_become=true

# Meta-Gruppe für die Produktionsumgebung
[production:children]
webservers
dbservers

# Globale Variablen für alle Hosts
[all:vars]
ansible_python_interpreter=/usr/bin/python3
ntp_server=ntp.example.com
log_level=info
```

```
# Gruppenspezifische Variablen für Webserver
[webservers:vars]
http_port=80
max_clients=200
web_server=nginx

# Gruppenspezifische Variablen für Datenbankserver
[dbservers:vars]
db_engine=mysql
db_port=3306
backup_frequency=daily
```

Hostprüfung und Variableninspektion:

```
# Erreichbarkeit der Webserver testen
ansible webservers -m ping

# Gesetzte Variablen einer Gruppe auslesen
ansible webservers -m debug -a "var=http_port"
```

Host-Variablen und IP-Bereiche

Einzelne Hosts können Werte aus Gruppenvariablen gezielt überschreiben:

```
[webservers]
web1 ansible_host=192.168.1.10 max_clients=300
web2 ansible_host=192.168.1.11
```

Für homogene Serverfarmen mit fortlaufender Nummerierung bietet das INI-Format praktische Range-Notationen:

```
[webservers]
web[01:10].example.com ansible_host=192.168.1.[10:19]
```

Hierarchische Strukturen im YAML-Format

Für hochgradig strukturierte oder verschachtelte Umgebungen bietet das YAML-Format den Vorteil, native Datentypen (Listen, Booleans, Dictionaries) exakt abzubilden.

Erstelle `inventory.yaml` mit sauberer 2-Space-Einrückung:

```
all:
  vars:
    ansible_python_interpreter: /usr/bin/python3
    ntp_server: ntp.example.com
    log_level: info
  children:
    webservers:
      hosts:
        web1:
          ansible_host: 192.168.1.10
          ansible_user: devops
          ansible_become: true
          ansible_become_method: sudo
          max_clients: 300
        web2:
          ansible_host: 192.168.1.11
          ansible_user: devops
          ansible_become: true
          ansible_become_method: sudo
      vars:
        http_port: 80
        web_server: nginx
    dbservers:
```

```
        hosts:
          db1:
            ansible_host: 192.168.1.20
            ansible_user: devops
            ansible_become: true
          db2:
            ansible_host: 192.168.1.21
            ansible_user: devops
            ansible_become: true
        vars:
          db_engine: mysql
          db_port: 3306
          backup_frequency: daily
      production:
        children:
          webservers:
          dbservers:
        vars:
          environment: prod
          monitoring_enabled: true
      staging:
        children:
          webservers:
        vars:
          environment: staging
          monitoring_enabled: false
```

Die Adressierung erfolgt analog zur INI-Datei:

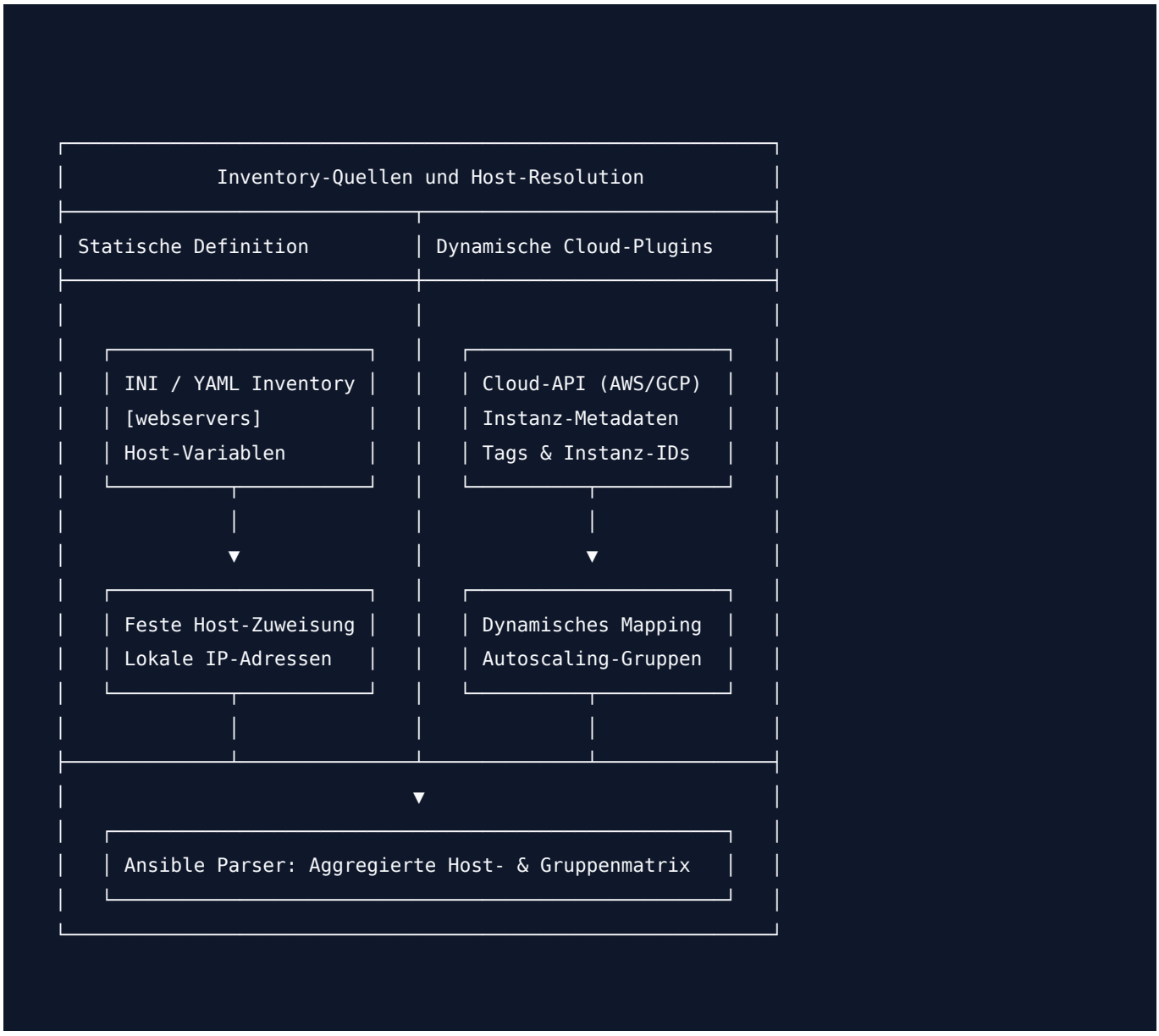
```
ansible -i inventory.yaml production -m ping
```

> **Praxistipp für Host-Organisation:** Verwende kurze, funktionale Host-Aliase (wie `web-prod-01`) anstelle kryptischer Cloud-Hostnamen (`ec2-198-51-100-24.compute-1.amazonaws.com`). Dies vereinfacht die Lesbarkeit von Logs, Fehlerausgaben und Playbook-

> Reports erheblich.

Dynamische Inventories für Cloud-Umgebungen

In dynamischen Cloud- und Containerumgebungen (wie AWS, Azure oder GCP) werden virtuelle Instanzen kontinuierlich neu erstellt oder terminiert. Statische Inventory-Dateien würden veralten. Hier kommen dynamische Inventory-Plugins zum Einsatz.



□□ Praktisches Beispiel: AWS EC2 Dynamic Inventory

Installiere zunächst die erforderlichen Python-SDK-Pakete auf dem Control Node:

```
pip install boto3 botocore
```

Erstelle die Plugin-Konfigurationsdatei `aws_ec2.yaml`:

```
plugin: amazon.aws.aws_ec2
regions:
  - eu-central-1
  - us-east-1
keyed_groups:
  - key: tags.Role
    prefix: role
  - key: tags.Environment
    prefix: env
hostnames:
  - private-ip-address
  - tag:Name
compose:
  ansible_host: private_ip_address
```

Stelle sicher, dass die AWS-Zugangsdaten über Umgebungsvariablen oder `~/.aws/credentials` verfügbar sind:

```
export AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
export AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
```

Teste die dynamische Abfrage mit dem Inventory-Inspektionsbefehl:

```
ansible-inventory -i aws_ec2.yaml --graph
```

Die Ausgabe zeigt die automatisch aus den AWS-Tags generierten Hostgruppen:

```
@all:
  |--@aws_ec2:
  |  |--10.0.1.45
  |  |--10.0.1.88
  |--@env_production:
  |  |--10.0.1.45
  |--@role_webservers:
  |  |--10.0.1.45
```

Nun können die dynamisch ermittelten Cloud-Instanzen direkt über ihre Tags angesprochen werden:

```
ansible -i aws_ec2.yaml role_webservers -m ping
```

Bewährte Vorgehensweisen für professionelle Inventare:

Kriterium	Empfohlene Praxis	Operativer Vorteil	Praxisbeispiel
Logische Gruppen	Trennung nach Funktion und Stage	Zielgenaue Playbook-Ausführung	`[webservers]`, `[dbservers]`, `[prod:children]`

Kriterium	Empfohlene Praxis	Operativer Vorteil	Praxisbeispiel
Gruppenvariablen	Gemeinsame Parameter zentral definieren	Reduziert Redundanzen und Tippfehler	`[webservers:vars]` `http_port=80`
Verzeichnis-Layout	Inventare in Unterdateien aufteilen	Klare Trennung zwischen Umgebungen	`inventories/production/`, `inventories/staging/`
Sicherheitsisolation	Passwörter und Tokens verschlüsseln	Schutz vertraulicher Credentials	`ansible-vault create group_vars/all/vault.yml`
Host-Validierung	Syntax vor dem Ausführen prüfen	Vermeidung von Runtime-Abbrüchen	`ansible-inventory -i inventory.ini --list`

Fehleranalyse bei dynamischen Inventories:

- `Missing dependency: boto3`: Das AWS-SDK fehlt im Python-Pfad von Ansible. Prüfe `pip list | grep boto3` innerhalb des aktiven Virtual Environments.
- `Access Denied / AuthFailure`: Der IAM-Benutzer besitzt unzureichende Leserechte für `ec2:DescribeInstances`. Hinterlege eine Richtlinie mit lesendem Zugriff.
- `Inventory parse error`: Falsche Dateiendung bei Plugins. Moderne Ansible-Cloud-Plugins verlangen strikt Endungen wie `.aws_ec2.yml` oder `.azure_rm.yml`.

⚠ **Sicherheitshinweis zu IP-Adressen und Cloud-Metadaten:** Verwende in Cloud-Umgebungen nach Möglichkeit interne private IP-Adressen und greife über ein VPN oder einen SSH-Bastion-Host (Jump Host) auf die Managed Nodes zu. Exponiere SSH-Ports (TCP 22) niemals ungeschützt in das öffentliche Internet.

Ad-Hoc-Befehle

Ad-Hoc-Befehle sind einzeilige Ausführungsbefehle, die Administratoren den sofortigen Zugriff auf die gesamte Flotte ermöglichen. Sie eignen sich hervorragend für schnelle Statusabfragen, Notfall-Patches oder punktuelle Wartungsarbeiten.

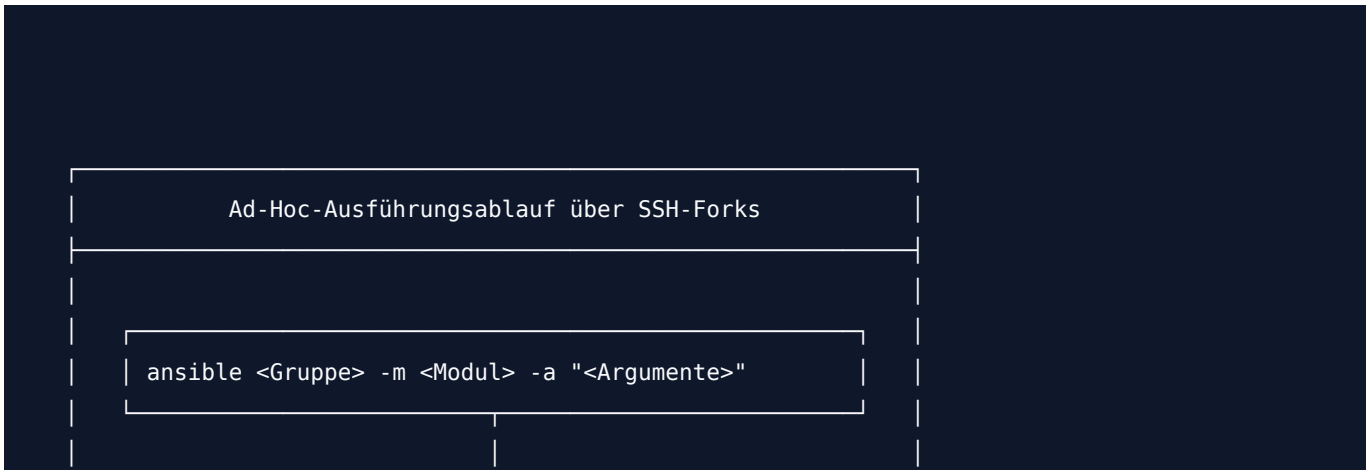
Syntax und parallele Befehlsausführung

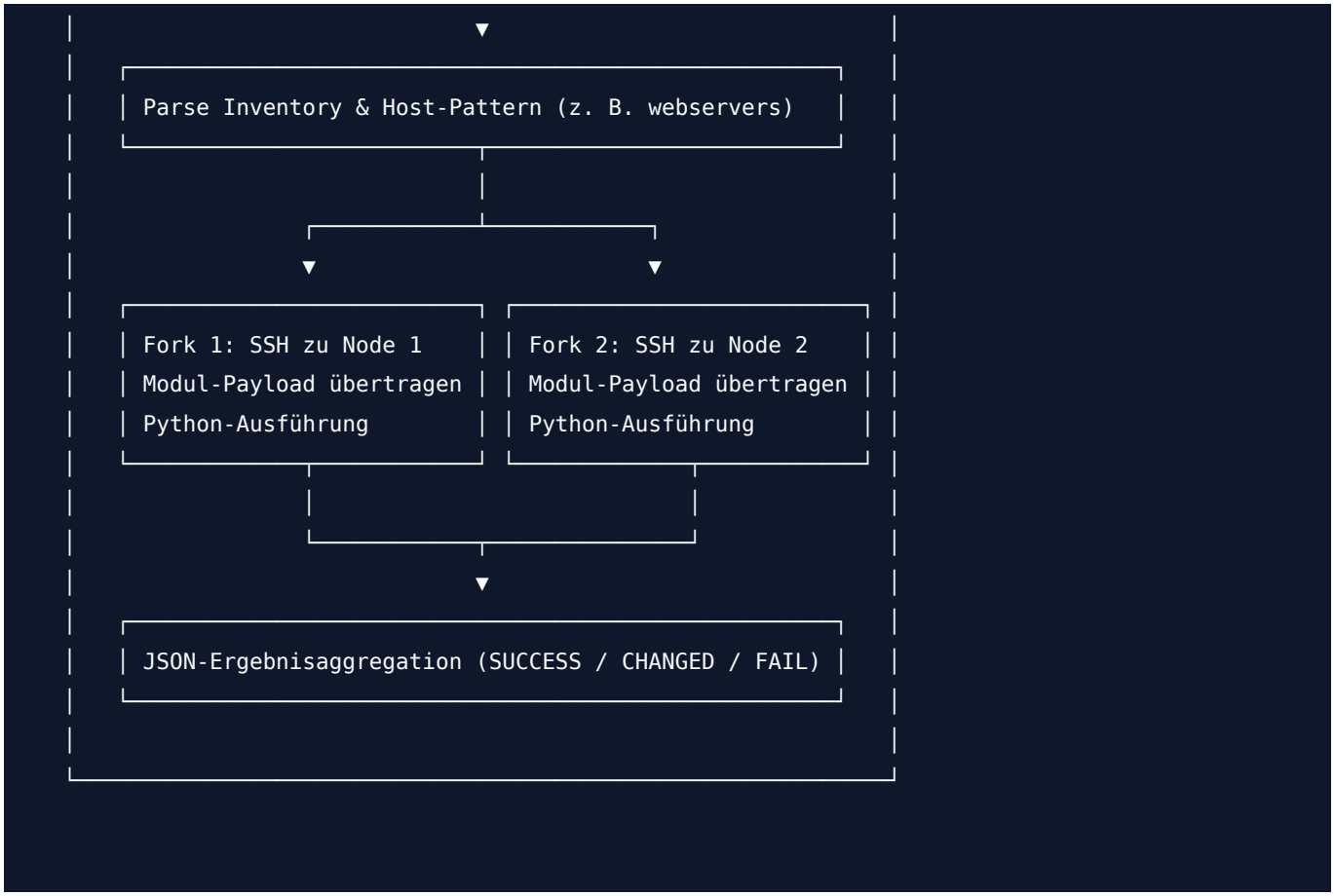
Die grundlegende Anatomie eines Ad-Hoc-Befehls lautet:

```
ansible <host-pattern> -m <modul_name> -a "<modul_argumente>" [optionen]
```

Ansible führt diese Kommandos hochgradig parallel aus. Über den Parameter `-f` (Forks) wird festgelegt, wie viele parallele SSH-Verbindungen gleichzeitig aufgebaut werden:

```
ansible webservers -m ping -f 10
```





Wichtige Kernmodule im operativen Einsatz

1. command vs. shell: Befehlsausführung verstehen

Das Modul `command` führt Befehle direkt auf dem Zielsystem aus. Es umgeht dabei eine Shell (`/bin/sh`), wodurch Shell-Sonderzeichen wie Pipes (`|`), Umleitungen (`>` , `<`) und Umgebungsvariablen nicht interpretiert werden. Dies macht das Modul besonders sicher gegen unbeabsichtigte Code-Injektionen:

```
ansible all -m command -a "/usr/bin/uptime"
```

Das Modul `shell` startet hingegen eine vollwertige Shell auf dem Zielsystem. Es erlaubt Pipes, Verkettungen und Wildcards, sollte jedoch mit Bedacht eingesetzt werden:

```
ansible webservers -m shell -a "uptime | awk '{print $3}' > /tmp/uptime.log" --become
```

2. Paketmanagement: apt, yum und das universelle package-Modul

Ansible bietet distributionsspezifische Module sowie das abstrakte Modul `package`, das den nativen Paketmanager des Zielsystems (APT, DNF, Pacman) automatisch erkennt:

```
# Debian/Ubuntu spezifisch mit Cache-Aktualisierung
ansible webservers -m apt -a "name=nginx state=present update_cache=yes
cache_valid_time=3600" --become

# RHEL/CentOS spezifisch
ansible dbservers -m yum -a "name=mariadb-server state=present" --become

# Distributionsunabhängig (universell)
ansible all -m package -a "name=htop state=present" --become
```

3. Benutzer- und Schlüsselverwaltung

Benutzerkonten lassen sich inklusive Gruppenzugehörigkeiten, Shell und Home-Verzeichnis idempotent einrichten:

```
ansible dbservers -m user -a "name=appuser uid=1050 state=present shell=/bin/bash
groups=wheel append=yes" --become
```

Hinterlege den öffentlichen SSH-Schlüssel für den neu erstellten Benutzer:

```
ansible dbservers -m authorized_key -a "user=appuser state=present key='{{ lookup('file',
'~/.ssh/id_rsa.pub') }}'" --become
```

4. Dateiübertragung und Statusprüfung

Das Modul `copy` überträgt Dateien vom Control Node auf die Zielsysteme. Durch `backup=yes` legt Ansible vor dem Überschreiben automatisch ein Sicherheitsbackup der Zieldatei an:

```
ansible webservers -m copy -a "src=/local/configs/nginx.conf dest=/etc/nginx/nginx.conf
owner=root group=root mode=0644 backup=yes" --become
```

Überprüfe Existenz, Berechtigungen und Checksummen vorhandener Dateien mit `stat`:

```
ansible webservers -m stat -a "path=/etc/nginx/nginx.conf get_checksum=yes"
```

5. Dienstverwaltung und Systemneustart

Dienste (systemd) werden über das Modul `service` gesteuert:

```
# Dienst neu starten
ansible webservers -m service -a "name=nginx state=restarted" --become
```

```
# Dienst aktivieren (Boot-Autostart) und starten
ansible dbservers -m service -a "name=mariadb state=started enabled=yes" --become
```

Ein kontrollierter Neustart ganzer Servergruppen lässt sich über das spezialisierte `reboot`-Modul abwickeln, das automatisch wartet, bis die Maschine wieder auf Port 22 antwortet:

```
ansible webservers -m reboot -a "msg='Geplante Wartung durch AdminDocs'
reboot_timeout=600" --become
```

Modul-Übersichtstabelle

Die folgende Übersicht fasst die wichtigsten Ad-Hoc-Module zusammen:

Modul	Hauptfunktion	Idempotent	Typische Schlüsselargumente	Operatives Einsatzszenario
ping	Konnektivitäts- und Python-Check	Ja	Keine	Verbindungsdiagnose
command	Direkte Befehlsausführung	Nein	`cmd`, `chdir`, `creates`, `removes`	Standardbefehle ohne Shell-Features
shell	Ausführung mit Shell-Features	Nein	`cmd`, `executable`, `creates`	Pipelines, Redirects, Umgebungsvariablen
package	Universelles Paketmanagement	Ja	`name`, `state=present/absent`	Gemischte Umgebungen (Debian & RHEL)
apt / yum	Spezifisches Paketmanagement	Ja	`name`, `state`, `update_cache`	Exakte Paketpflege mit Cache-Parametern
copy	Dateiübertragung (Lokal zu Remote)	Ja	`src`, `dest`, `owner`, `mode`, `backup`	Bereitstellung von Konfigurationen

Modul	Hauptfunktion	Idempotent	Typische Schlüsselargumente	Operatives Einsatzszenario
stat	Metadaten und Hashprüfung	Ja	`path`, `get_checksum`	Vorprüfungen in Skripten
service	systemd-Dienststeuerung	Ja	`name`, `state=started/stopped`, `enabled`	Daemon-Lebenszyklus und Autostart
user	Benutzer- und Gruppenverwaltung	Ja	`name`, `uid`, `groups`, `state`, `shell`	Standardisierung von Service-Accounts
setup	System-Facts einsammeln	Ja	`filter`, `gather_subset`	Hardware- und OS-Inventarisierung

Grenzen von Ad-Hoc-Befehlen

Ad-Hoc-Befehle stoppen dort, wo komplexe Arbeitsabläufe beginnen. Sie bieten:

- Keine Ereignissteuerung (Handler für Dienste nach Dateiänderungen)
- Keine bedingte Ausführungslogik (`when`-Klauseln über mehrere Schritte)
- Keine integrierten Rollback- und Rettungsmechanismen (`block` / `rescue`)
- Keine strukturierte Versionierbarkeit in Git-Repositories

Sobald mehrere Aufgaben in definierter Reihenfolge ausgeführt und dokumentiert werden müssen, ist der Wechsel zu strukturierten Playbooks erforderlich.

> ⚠ **Warnung zum Einsatz des Shell-Moduls:** Das Modul `shell` meldet standardmäßig immer `changed: true`, da Ansible nicht wissen kann, welche Operationen innerhalb des Shell-Befehls stattfanden. Um Idempotenz zu wahren, sollten Argumente wie `creates=/pfad/zur/datei` gesetzt werden, damit der Befehl übersprungen wird, wenn die Zieldatei bereits existiert.

> **Typische Fehlerquellen bei Ad-Hoc-Ausführungen:**
>
> - `Permission denied`: Die auszuführende Operation erfordert Root-Privilegien. Ergänze das Kommando um `--become`.
> - `Missing arguments`: Modul-Argumente müssen als ein einziger String an `-a` übergeben werden (z. B. `-a "name=nginx state=present"`).
> - `No hosts matched`: Das Host-Pattern entspricht keinem Rechner in der Inventory. Prüfe mit `ansible <pattern> --list-hosts`.

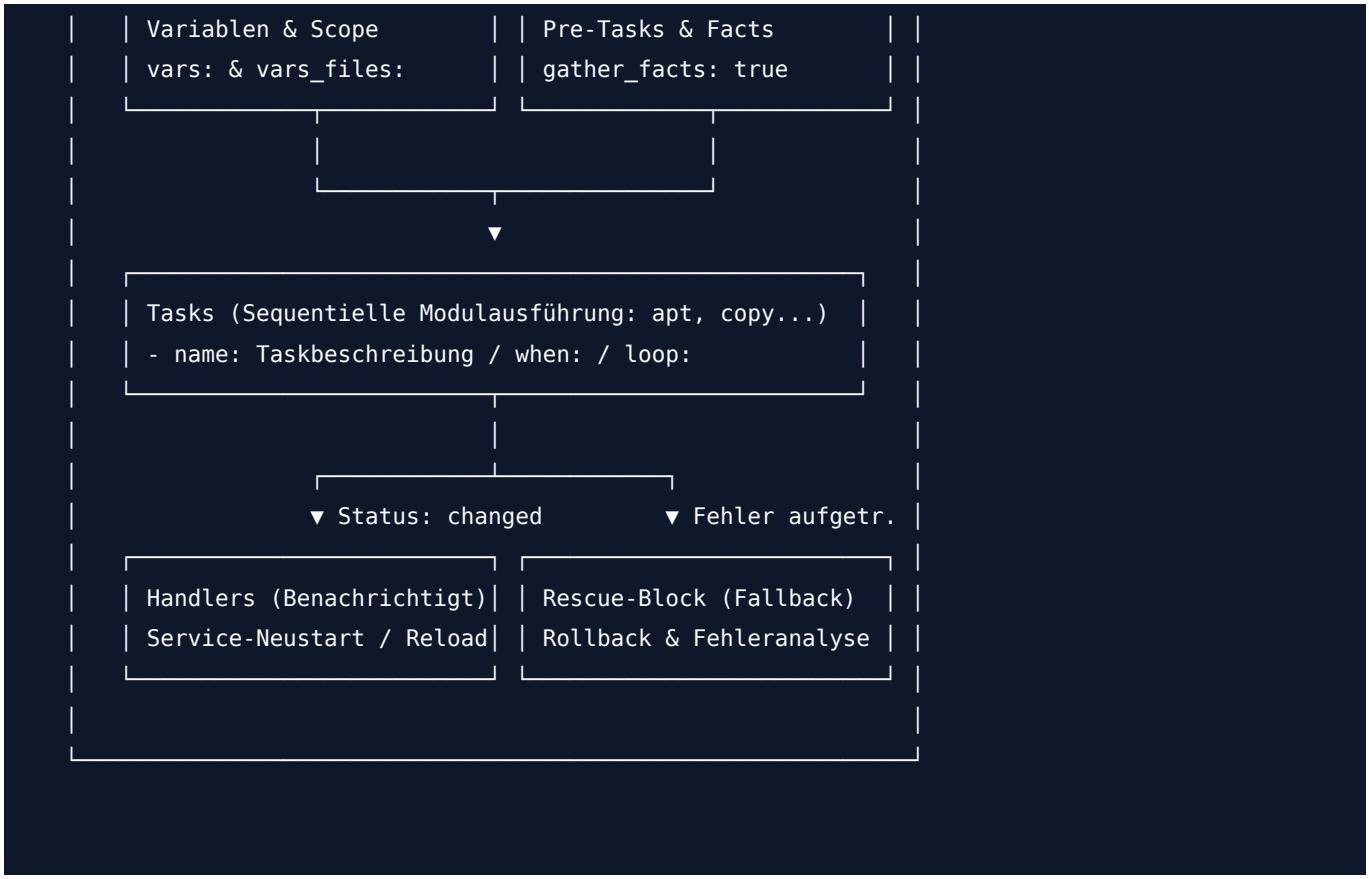
Playbooks: Strukturierte Automatisierung

Playbooks sind das Herzstück reproduzierbarer Automatisierung. In strukturierten YAML-Dokumenten wird der Soll-Zustand ganzer Infrastrukturen beschrieben, versioniert und schrittweise umgesetzt.

Aufbau und Basiselemente eines Playbooks

Ein Playbook besteht aus einem oder mehreren „Plays“. Jedes Play ordnet einer Zielgruppe von Hosts eine geordnete Liste von Aufgaben (Tasks) zu.





Praktisches Beispiel: Vollständiges Webserver-Playbook

Erstelle die Datei `setup_webserver.yaml`:

```
---
- name: Setup und Absicherung des Nginx-Webservers
  hosts: webservers
  become: true
  vars:
    http_port: 80
    web_package: nginx
    server_admin: admin@admindocs.local

  tasks:
    - name: Paket-Cache auf Debian-Systemen aktualisieren
      apt:
        update_cache: yes
```

```
      cache_valid_time: 3600
    when: ansible_os_family == 'Debian'

  - name: Nginx-Paket installieren
    package:
      name: "{{ web_package }}"
      state: present

  - name: Maßgeschneiderte Konfigurationsdatei bereitstellen
    copy:
      src: ./files/nginx.conf
      dest: /etc/nginx/nginx.conf
      owner: root
      group: root
      mode: '0644'
      backup: yes
    notify: Nginx neu laden

  - name: Sicherstellen, dass Nginx gestartet und im Autostart registriert ist
    service:
      name: "{{ web_package }}"
      state: started
      enabled: true

handlers:
  - name: Nginx neu laden
    service:
      name: "{{ web_package }}"
      state: reloaded
```

Ausführung und Syntaxprüfung:

Vor der tatsächlichen Ausführung im Produktivnetz sollte jedes Playbook validiert werden:

```
# 1. Syntaxprüfung durchführen
ansible-playbook setup_webserver.yaml --syntax-check
```

```
# 2. Trockenlauf mit Diff-Anzeige
ansible-playbook setup_webserver.yaml --check --diff

# 3. Tatsächliche Ausführung
ansible-playbook setup_webserver.yaml
```

Kernkomponenten des Playbooks im Überblick:

- `hosts`: Bestimmt die Zielgruppe aus dem Inventory (z. B. `webservers` oder `all:!db1` für Ausschlüsse).
- `become`: Aktiviert die Privilegien-Eskalation via sudo für das gesamte Play.
- `vars`: Lokale Variablen, die innerhalb der Tasks referenziert werden.
- `tasks`: Die sequentielle Liste der auszuführenden Aktionen.
- `handlers`: Reaktive Tasks, die nur dann am Ende des Plays ausgeführt werden, wenn eine überwachende Task mit `notify` eine Zustandsänderung (`changed: true`) gemeldet hat.

Kontrollstrukturen: Loops, Conditionals und Handlers

1. Iterationen mit Schleifen (Loops)

Anstatt dieselbe Task mehrfach für verschiedene Pakete oder Benutzer zu kopieren, bündelt `loop` die Ausführung:

```
- name: Erforderliche Systempakete installieren
  package:
    name: "{{ item }}"
    state: present
  loop:
    - curl
    - htop
    - rsync
    - ufw
```

2. Bedingungen mit when-Direktiven

Tasks lassen sich gezielt an Betriebssysteme, Kernel-Stände oder benutzerdefinierte Flags binden:

```
- name: Firewall-Port auf RedHat-Systemen öffnen
  firewalld:
    port: "{{ http_port }}/tcp"
    permanent: true
    state: enabled
  when: ansible_os_family == 'RedHat'

- name: Firewall-Port auf Debian/Ubuntu via UFW öffnen
  ufw:
    rule: allow
    port: "{{ http_port }}"
    proto: tcp
  when: ansible_os_family == 'Debian'
```

□□ Praktisches Beispiel: Webserver-Deployment mit Firewall und Validierung

Dieses fortgeschrittene Playbook demonstriert das Zusammenspiel aus Installation, Firewall-Absicherung, dynamischer Index-Erstellung und anschließender Funktionsprüfung über das `uri`-Modul:

```
---
- name: Enterprise Web-Stack Rollout mit Verifikation
  hosts: webservers
  become: true
  vars:
    http_port: 80
    app_title: "AdminDocs Production Server"
```

```
tasks:
  - name: Nginx und UFW installieren
    package:
      name:
        - nginx
        - ufw
      state: present
    register: pkg_result

  - name: HTTP-Port in Firewall freischalten
    ufw:
      rule: allow
      port: "{{ http_port }}"
      proto: tcp
    when: pkg_result.changed

  - name: UFW-Dienst aktivieren
    ufw:
      state: enabled

  - name: Statische Landingpage ausliefern
    copy:
      content: "<h1>{{ app_title }}</h1><p>Verwaltet durch Ansible.</p>"
      dest: /var/www/html/index.html
      owner: www-data
      group: www-data
      mode: '0644'

  - name: Funktionsprüfung per HTTP-Request durchführen
    uri:
      url: "http://localhost:{{ http_port }}"
      return_content: yes
      status_code: 200
    register: web_response
    failed_when: "'AdminDocs Production' not in web_response.content"

  - name: Erfolgsmeldung im Log ausgeben
    debug:
      msg: "Webserver erfolgreich verifiziert. HTTP-Status: {{ web_response.status }}"
```

Fehlerbehandlung, Blocks und Debugging

Komplexe Deployments benötigen Ausfallsicherheit. Mit der Konstruktion `block`, `rescue` und `always` bietet Ansible strukturierte Fehlerbehandlung analog zu Try-Catch-Blöcken:

```
---
- name: Robuste Benutzerkonfiguration mit Recovery
  hosts: all
  become: true
  vars:
    required_users:
      - name: devuser
        group: developers
      - name: audituser
        group: auditor

  tasks:
    - name: Primärer Ausführungsblock
      block:
        - name: Entwickler-Gruppe anlegen
          group:
            name: developers
            state: present

        - name: Benutzerkonten initialisieren
          user:
            name: "{{ item.name }}"
            groups: "{{ item.group }}"
            state: present
          loop: "{{ required_users }}"

      rescue:
        - name: Fehlerdiagnose erfassen
          debug:
            msg: "Fehler beim Anlegen der Benutzerkonten auf {{ inventory_hostname }}.
Starte Rollback..."

        - name: Sicherheitslogging ausführen
          shell: "logger -t ansible 'User provisioning failed on host'"
```

```
        always:
          - name: Abschluss-Check ausführen
            debug:
              msg: "Benutzer-Provisionierungslauf beendet."
```

Diagnose und Fehlersuche im Playbook-Betrieb:

Syntaxprüfung:

```
ansible-playbook site.yaml --syntax-check
```

Ausführung bei einer bestimmten Task starten:

```
ansible-playbook site.yaml --start-at-task="HTTP-Port in Firewall freischalten"
```

Interaktiver Einzelschritt-Modus:

```
ansible-playbook site.yaml --step
```

Host-Einschränkung zur Laufzeit:

```
ansible-playbook site.yaml --limit web1
```

Aufgabenliste anzeigen ohne Ausführung:

```
ansible-playbook site.yaml --list-tasks
```

Best Practice	Operative Umsetzung	Nutzen
Deskriptive Task-Namen	`name: Nginx-Dienst im systemd aktivieren`	Aussagekräftige Logs und CLI-Ausgaben
Handler-Nutzung	`notify: Dienst neu laden`	Verhindert unnötige Neustarts bei unveränderten Dateien
Trockenlauf vor Rollout	`--check --diff`	Visualisiert Konfigurationsabweichungen vor der Änderung
Versionskontrolle	Playbooks strikt in Git pflegen	Nachvollziehbarkeit und Team-Kollaboration
Modulare Struktur	Aufgaben in `tasks/main.yml` auslagern	Hohe Wiederverwendbarkeit in Projekten

> ⚠ **Warnung vor Einrückungsfehlern in YAML:** In YAML sind Tabulatoren (`\t`) als Einrückungszeichen verboten. Verwende ausschließlich Leerzeichen (empfohlen: 2 Leerzeichen pro Ebene). Ein einziger Tabulator führt zum sofortigen Parse-Abbruch (`YAML syntax error`).

> **Typische Fehlerquellen bei Playbook-Läufen:**
>
> - `fatal: [web1]: FAILED! => {"changed": false, "msg": "No package matching 'nginx' found"}`: Der lokale Paket-Cache ist veraltet. Ergänze die Task um `update_cache: yes`.
> - `Handler wurde nicht ausgeführt`: Ein Handler läuft nur dann, wenn die auslösende Task den Status `changed: true` meldet. Wurde die Datei nicht modifiziert, bleibt der Handler inaktiv.
> - `Variable is undefined`: Eine im Playbook referenzierte Variable existiert weder im Play, noch in den Inventory- oder Host-Dateien. Setze sichere Fallbacks mit `{{ var | default('wert') }}`.

Variablen und Facts

Variablen und Facts verleihen Playbooks die nötige Flexibilität. Anstatt Konfigurationswerte statisch festzuschreiben, passen sich dynamische Playbooks an Umgebungsstufen, Betriebssystemfamilien und Hardwareressourcen an.

Definition und Gültigkeitsbereiche von Variablen

In Ansible können Variablen auf verschiedenen Ebenen definiert werden:

1. Inline im Playbook

```
---
- name: Demonstration von Inline-Variablen
  hosts: all
  vars:
    app_name: core-service
    service_port: 8080
```

```
      allowed_ips:
        - 10.0.0.1
        - 10.0.0.2
      database:
        name: production_db
        pool_size: 25
    tasks:
      - name: Konfigurationswerte ausgeben
        debug:
          msg: "Dienst {{ app_name }} lauscht auf Port {{ service_port }} mit DB {{
database.name }}"
```

2. Externe Variablendateien (vars_files)

Für größere Setups werden Variablen in dedizierten Dateien unter `vars/` organisiert:

```
# vars/app_settings.yaml
app_name: core-service
http_port: 80
environment: production
database_driver: mysql
max_connections: 500
```

Einbindung im Playbook:

```
- name: Playbook mit externen Variablen
  hosts: webservers
  vars_files:
    - vars/app_settings.yaml
  tasks:
    - name: Web-Port anwenden
```

```
        debug:
          msg: "Aktiver Port: {{ http_port }}"
```

3. Verzeichnisbasierte Variablen: group_vars und host_vars

Ansible lädt automatisch Variablendateien aus den Verzeichnissen `group_vars/` und `host_vars/`, wenn diese relativ zur Inventory-Datei oder zum Playbook liegen:

```
inventories/production/
├── hosts.ini
├── group_vars/
│   ├── all.yaml          # Gilt für alle Knoten
│   ├── webservers.yaml   # Gilt für Gruppe webservers
│   └── dbservers.yaml    # Gilt für Gruppe dbservers
└── host_vars/
    └── web1.yaml         # Überschreibt Werte exklusiv für Host web1
```

Variablen-Precedence und Vorrangregeln

Da Variablen an vielen Orten definiert werden können, besitzt Ansible eine strikte Rangordnung (Precedence). Ein Wert auf höherer Stufe überschreibt identische Variablennamen auf niedrigeren Stufen.

Die wichtigsten Ebenen im Überblick (von niedrig nach hoch sortiert):

Rang	Definitionsebene	Typisches Einsatzgebiet	Überschreibbar durch
1 (Niedrig)	`role defaults` (`defaults/main.yml`)	Standardwerte in wiederverwendbaren Rollen	Praktisch jede andere Definition
2	`inventory group_vars/*`	Basiswerte ganzer Servergruppen	Host-Variablen, Playbook-Vars

Rang	Definitionsebene	Typisches Einsatzgebiet	Überschreibbar durch
3	`inventory host_vars/*`	Hostspezifische Abweichungen (IPs, Disks)	Playbook-Vars
4	`playbook vars` (im Play-Header)	Globale Einstellungen für das Play	Task-Vars, Extra-Vars
5	`playbook vars_files`	Externe Konfigurationsdateien	Task-Vars, Extra-Vars
6	`host facts` (automatisch gesammelt)	Hardware-, Netzwerk- und OS-Daten	Task-Vars, Extra-Vars
7	`task vars` (in der Task definiert)	Gültig nur für die einzelne Task	Extra-Vars
8 (Höchste)	`extra vars` (`-e "key=val"`)	Manuelle Overrides an der Kommandozeile	Durch nichts überschreibbar

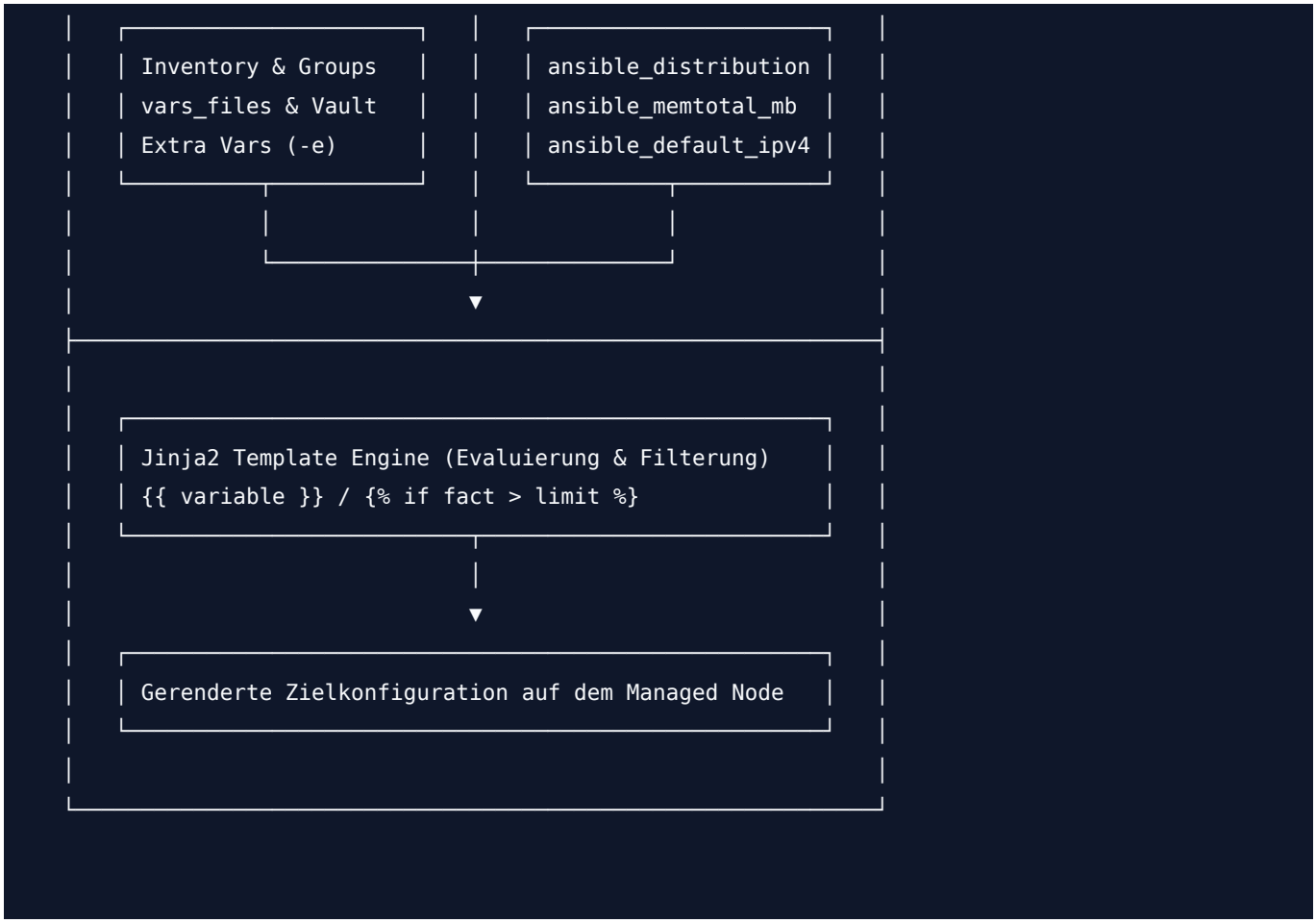
□□ Praktisches Beispiel: Overrides an der Kommandozeile

```
ansible-playbook deploy.yaml -e "environment=staging http_port=8080"
```

Systemanalyse mit Ansible Facts

Facts sind Systeminformationen, die Ansible zu Beginn jedes Plays automatisch von den Managed Nodes über das `setup`-Modul ermittelt.

```
┌──────────────────────────────────────────────────────────┐
│           Variablenauflösung und Fact-Aggregation        │
├──────────────────────────────┬───────────────────────────┤
│ Variablenquellen             │ Zielsystem-Metadaten (Facts) │
├──────────────────────────────┼───────────────────────────┤
│                              │                           │
```



Häufig genutzte Standard-Facts:

- `ansible_os_family`: Betriebssystemfamilie (`Debian`, `RedHat`, `Archlinux`)
- `ansible_distribution`: Exakte Distribution (`Ubuntu`, `Debian`, `CentOS`, `Fedora`)
- `ansible_distribution_version`: Distributionsstand (z. B. `22.04`)
- `ansible_memtotal_mb`: Gesamter physikalischer Arbeitsspeicher in Megabyte
- `ansible_processor_vcpus`: Anzahl verfügbarer virtueller CPU-Kerne
- `ansible_default_ipv4.address`: Primäre IPv4-Adresse des Zielhosts

□□ Praktisches Beispiel: Fact-gesteuerte Installation

Das folgende Playbook passt Paketnamen und Speichereinstellungen dynamisch an das Zielsystem an:

```
---
- name: Intelligente Konfiguration basierend auf System-Facts
  hosts: all
  become: true
  gather_facts: true

  tasks:
    - name: Webserver-Paket abhängig von der OS-Familie wählen
      package:
        name: "{{ 'apache2' if ansible_os_family == 'Debian' else 'httpd' }}"
        state: present

    - name: Systemdaten im Log ausgeben
      debug:
        msg: >
          Host: {{ ansible_hostname }} |
          OS: {{ ansible_distribution }} {{ ansible_distribution_version }} |
          RAM: {{ ansible_memtotal_mb }} MB |
          CPUs: {{ ansible_processor_vcpus }}

    - name: PHP-Memory-Limit an verfügbaren RAM anpassen
      lineinfile:
        path: /etc/app.conf
        line: "php_memory_limit = {{ (ansible_memtotal_mb * 0.25) | int }}M"
        create: yes
      when: ansible_memtotal_mb > 2048
```

Eigene Facts (Custom Facts) bereitstellen

Zusätzlich zu den Standard-Facts können Administratoren auf Zielsystemen statische oder dynamische Custom Facts hinterlegen. Diese müssen im Verzeichnis `/etc/ansible/facts.d/` mit der Dateiendung `.fact` abgelegt werden:

```
sudo mkdir -p /etc/ansible/facts.d
sudo tee /etc/ansible/facts.d/datacenter.fact << 'EOF'
[location]
rack = R42
datacenter = FRA1
environment = production
EOF
```

Ansible liest diese Werte automatisch in die Struktur `ansible_local` ein:

```
- name: Custom Fact abfragen
  debug:
    msg: "Dieser Server steht im Rechenzentrum {{
ansible_local.datacenter.location.datacenter }}"
```

Fact Caching für performante Großumgebungen

In Umgebungen mit Hunderten Servern kann das Abrufen der Facts zu Beginn jedes Playbooks spürbare Zeit in Anspruch nehmen. Durch Fact Caching in `ansible.cfg` werden die Metadaten lokal zwischengespeichert:

```
[defaults]
gathering = smart
fact_caching = jsonfile
fact_caching_connection = /tmp/ansible_fact_cache
fact_caching_timeout = 86400
```

Konfigurations-Templating mit Jinja2

Während einfache Dateien mit `copy` übertragen werden, ermöglicht das Modul `template` die dynamische Erzeugung von Konfigurationsdateien über die Templating-Engine Jinja2.

Erstelle die Vorlagendatei `templates/nginx_vhost.conf.j2`:

```
# Automatisch generiert durch Ansible - Manuelle Änderungen werden überschrieben
server {
    listen {{ http_port }};
    server_name {{ ansible_fqdn }};

    root /var/www/html;
    index index.html;

    # Dynamische Thread-Berechnung basierend auf vCPUs
    worker_processes {{ ansible_processor_vcpus }};

{% if ansible_memtotal_mb > 4096 %}
    # High-Performance Memory Cache
    fastcgi_cache_path /var/cache/nginx levels=1:2 keys_zone=APP:100m inactive=60m;
{% else %}
    # Standard Memory Settings
    fastcgi_cache_path /var/cache/nginx levels=1:2 keys_zone=APP:20m inactive=30m;
{% endif %}

    location / {
        try_files $uri $uri/ =404;
    }
}
```

Bereitstellung im Playbook:

```
- name: Nginx-VirtHost aus Jinja2-Template erzeugen
```

```
    template:
      src: templates/nginx_vhost.conf.j2
      dest: /etc/nginx/sites-available/default
      owner: root
      group: root
      mode: '0644'
    notify: Nginx neu laden
```

Nützliche Jinja2-Filter und Magic Variables

- `default`: Setzt einen Fallback-Wert, falls die Variable ungesetzt ist:

  ```
  {{ app_port | default(8080) }}
  ```

- `join`: Verbindet Listenelemente zu einem String:

  ```
  {{ allowed_hosts | join(', ') }}
  ```

- `upper` / `lower`: Wandelt Strings in Groß- bzw. Kleinbuchstaben um.
- `inventory_hostname`: Enthält den Namen des aktuellen Zielhosts, wie er in der Inventory definiert ist.
- `groups['webservers']`: Liefert eine Liste aller Hostnamen aus der Gruppe `webservers`.

Sensible Daten mit Ansible Vault schützen

Passwörter, API-Tokens, SSH-Private-Keys und Zertifikate dürfen niemals im Klartext in Repositories abgelegt werden. Mit Ansible Vault werden Dateien oder einzelne Variablenwerte

symmetrisch (AES-256) verschlüsselt.

□□ Praktisches Beispiel: Verschlüsselte Variablendatei verwalten

```
# Neue verschlüsselte Datei erstellen (fordert Passwort an)
ansible-vault create vars/vault.yaml

# Bestehende Klartextdatei nachträglich verschlüsseln
ansible-vault encrypt vars/secrets.yaml

# Verschlüsselte Datei im Editor bearbeiten
ansible-vault edit vars/vault.yaml

# Verschlüsselte Datei zur Ansicht entschlüsseln
ansible-vault view vars/vault.yaml
```

Inhalt der Datei `vars/vault.yaml`:

```
vault_db_password: "SuperSecretProductionDatabasePassword42!"
vault_api_key: "k8s-secret-token-xyz-987"
```

Einbindung im Playbook:

```
---
- name: Playbook mit geschützten Zugangsdaten
  hosts: dbservers
  become: true
  vars_files:
```

```
    - vars/vault.yaml
  tasks:
    - name: Datenbank-Benutzer mit Vault-Passwort konfigurieren
      mysql_user:
        name: dbadmin
        password: "{{ vault_db_password }}"
        priv: "*.*:ALL"
        state: present
```

Ausführung eines Playbooks mit Vault-Schutz:

```
# Passwort interaktiv abfragen
ansible-playbook deploy.yaml --ask-vault-pass

# Passwort aus einer geschützten Datei lesen
ansible-playbook deploy.yaml --vault-password-file ~/.vault_pass
```

> **Typische Fehlerquellen bei Variablen und Facts:**
>
> - `The field 'vars' has an invalid value`: Variablen-Namen dürfen keine Bindestriche (`-`) enthalten, sondern müssen mit Unterstrichen (`_`) formatiert werden (z. B. `http_port` statt `http-port`).
> - `AnsibleUndefinedVariable`: Jinja2-Parser-Fehler wegen nicht existierender Variablen. Fange optionale Variablen stets mit `| default()` ab.
> - `Vault password incorrect`: Beim Ausführen wurde das falsche Vault-Passwort eingegeben oder die Datei wurde mit einem anderen Vault-ID-Label verschlüsselt.

> ⚠ **Sicherheitshinweis zu unverschlüsselten Zugangsdaten:** Hinterlege niemals Passwörter oder private Schlüsseldateien im Klartext in Versionskontrollsystemen (Git). Richte Pre-Commit-Hooks ein, um versehentliches Einchecken unverschlüsselter Vault-Dateien zuverlässig zu verhindern.

Befehlsreferenz (Cheatsheet)

Die folgende Referenztabelle fasst die essenziellen Befehle und Optionen für den täglichen administrativen Einsatz von Ansible zusammen:

Kategorie	Befehl / Syntax	Wichtige Optionen	Zweck / Beschreibung
Konnektivität	`ansible <pattern> -m ping`	`-i <inv>`, `-u <user>`	Verbindung und Python-Subsystem auf Zielsystemen prüfen
Facts & Hardware	`ansible <pattern> -m setup`	`-a "filter=ansible_*"`	Sämtliche Systemmetadaten (OS, IP, CPU, RAM) einsammeln
Befehlsausführung	`ansible <pattern> -m command -a "<cmd>"`	`-f <forks>`, `--become`	Sichere Befehlsausführung ohne Shell-Interpolation
Shell & Pipelines	`ansible <pattern> -m shell -a "<cmd>"`	`creates=/path`	Befehlsausführung mit Pipes (`|`), Redirects und Wildcards

Kategorie	Befehl / Syntax	Wichtige Optionen	Zweck / Beschreibung
Paketverwaltung	`ansible <pattern> -m package -a "name=<pkg> state=present"`	`--become`, `--check`	Idempotente Paketinstallation über den nativen Paketmanager
Dienststeuerung	`ansible <pattern> -m service -a "name=<svc> state=started"`	`enabled=yes`, `--become`	Daemons starten, stoppen, neu laden und für Autostart aktivieren
Dateiübertragung	`ansible <pattern> -m copy -a "src=<src> dest=<dst>"`	`mode=0644`, `backup=yes`	Dateien bereitstellen mit automatischer Sicherungskopie
Benutzerverwaltung	`ansible <pattern> -m user -a "name=<usr> state=present"`	`groups=wheel`, `append=yes`	Lokale Benutzerkonten und Gruppenzuweisungen verwalten
Playbook-Syntax	`ansible-playbook <playbook.yaml> --syntax-check`	Keine	YAML-Struktur und Task-Syntax vor der Ausführung validieren
Dry-Run (Trockenlauf)	`ansible-playbook <playbook.yaml> --check --diff`	`-vvv`	Geplante Änderungen simulieren und Diffs anzeigen
Ziel-Limitierung	`ansible-playbook <playbook.yaml> --limit <host>`	`--start-at-task="<name>"`	Ausführung auf bestimmte Hosts oder Tasks einschränken
Extra Variables	`ansible-playbook <playbook.yaml> -e "<k>=<v>"`	`-e "@vars.json"`	Variablen zur Laufzeit mit höchster Priorität übersteuern

Kategorie	Befehl / Syntax	Wichtige Optionen	Zweck / Beschreibung
Inventory-Graph	`ansible-inventory -i <inv> --graph`	`--vars`	Hierarchische Baumansicht aller Gruppen und Hosts ausgeben
Vault-Erstellung	`ansible-vault create <secret.yaml>`	Keine	Neue AES-256-verschlüsselte Variablendatei anlegen
Vault-Verschlüsselung	`ansible-vault encrypt <file.yaml>`	`ansible-vault decrypt`	Bestehende Dateien nachträglich ver- oder entschlüsseln
Vault-Ausführung	`ansible-playbook site.yaml --ask-vault-pass`	`--vault-password-file`	Playbook mit passwortgeschützten Vault-Dateien ausführen

Weiterführende Ressourcen

Vertiefende Informationen, Spezifikationen und Community-Module finden sich in den folgenden offiziellen Quellen:

Ressource	Beschreibung	Link
Offizielle Ansible-Dokumentation	Vollständiges Referenzhandbuch, Installationsanleitungen und Release-Notes	docs.ansible.com
Ansible Getting Started Guide	Schritt-für-Schritt-Anleitung für den Einstieg in die grundlegenden Konzepte	Ansible Getting Started
Inventory Management Handbuch	Detaillierte Dokumentation zu statischen INI/YAML-Inventories und Variablenmustern	Ansible Inventory Guide

Ressource	Beschreibung	Link
Ansible Module & Collections Index	Vollständiges Verzeichnis aller built-in Module mit Argumenten und Beispielen	Ansible Module Index
Playbook Architecture Guide	Umfassende Anleitung zu Tasks, Handlern, Loops, Blocks und Fehlerbehandlung	Ansible Playbook Intro
Ansible Vault Handbuch	Best Practices zur sicheren Verschlüsselung vertraulicher Credentials	Ansible Vault Guide
Ansible Galaxy	Offizieller Community-Hub für wiederverwendbare Rollen und Collections	galaxy.ansible.com
Ansible GitHub Repository	Quellcode, Issue-Tracker und technische Diskussionen des Open-Source-Projekts	github.com/ansible/ansible

Fazit

Ansible schließt die Lücke zwischen manueller Systemadministration und hochkomplexen Enterprise-Orchestrierungswerkzeugen. Durch den konsequenten Verzicht auf Zielsystem-Agenten und die Nutzung bestehender Sicherheitsstandards wie OpenSSH bietet es einen unübertroffen schnellen Einstieg in die Welt von Infrastructure-as-Code.

Die Stärke von Ansible liegt in seiner Berechenbarkeit: Deklarative Playbooks und das strikte Prinzip der Idempotenz sorgen dafür, dass Serverkonfigurationen nachvollziehbar, versionierbar und frei von schleichenden Abweichungen bleiben. Mit der Beherrschung von Inventories, Ad-Hoc-Befehlen, robusten Playbook-Strukturen sowie der präzisen Variablensteuerung über Facts und Jinja2-Templates steht Linux-Administratoren ein solides Fundament für den produktiven Betrieb zur Verfügung.

> □□ **Praxistipp für den Produktivbetrieb:** Starte nicht mit gewaltigen, monolithischen Playbooks. Teile wiederkehrende Administrationsaufgaben von Beginn an in modulare Einheiten auf. Nutze `group_vars` und `host_vars` für umgebungsspezifische Parameter und sichere sensible Zugangsdaten ausnahmslos über Ansible Vault ab. Wer diesen strukturierten Ansatz verinnerlicht, kann seine Infrastruktur mühelos von einer Handvoll

> Testservern auf Hunderte Produktivinstanzen skalieren.

Im nächsten Schritt empfiehlt sich die Vertiefung in **Ansible Roles** und **Collections**, um Playbooks in standardisierte Verzeichnisbäume mit vordefinierten Standardwerten, Templates und Tasks zu strukturieren. In Kombination mit CI/CD-Pipelines (wie GitLab CI oder GitHub Actions) wird Ansible damit zum zentralen Motor einer vollautomatisierten, modernen Bereitstellungskette.

Quelle: https://admindocs.de/de/devops/ansible-grundlagen-automatisierung-fuer-linux-administratoren