

Denis Racz

Einstieg in Ansible

Schritt für Schritt vom ersten Playbook
zur professionellen Automatisierung

Mit zahlreichen Praxisbeispielen und Best Practices



Inhaltsverzeichnis

Einleitung	13
1 Die Sprache der Infrastruktur: Willkommen bei Ansible	17
1.1 Warum man Infrastruktur heute nicht mehr »per Hand« verwaltet	17
1.1.1 Traditionelle IT vs. Infrastructure as Code	18
1.1.2 Deklarativ statt prozedural: Denken in Zielzuständen	18
1.1.3 Warum der deklarative Ansatz insbesondere Einsteigern hilft	20
1.2 Was ist Ansible?	21
1.2.1 Wie Ansible zu seinem Namen kam	21
1.2.2 Die Stärken von Ansible	21
1.2.3 Einsatzbereiche in der Praxis	23
1.2.4 Ist Ansible mächtig genug?	23
1.3 Einführung in Ansible	25
1.3.1 Die wichtigsten Komponenten	25
1.3.2 Das Push-Modell	27
1.3.3 Der Ablauf einer Playbook-Ausführung	28
1.3.4 Praxisbeispiel (konzeptionell): Einfache Webserver- Bereitstellung	29
1.4 Ein kurzer Blick auf den weiteren Buchverlauf	29
1.5 Zusammenfassung	29
1.6 Quiz: Einstieg in Ansible und Infrastructure as Code	30
2 Einstieg in die Praxis – Installation & erste Schritte mit Ansible	33
2.1 Systemvoraussetzungen	33
2.1.1 Technische Anforderungen für unsere Testumgebung	33
2.1.2 Softwarevoraussetzungen	34
2.1.3 Die einzige Voraussetzung auf Zielsystemen: Python 3	34
2.2 Installation von Ansible auf verschiedenen Plattformen	35
2.2.1 Ubuntu oder Debian	36
2.2.2 Rocky Linux oder RHEL	36
2.2.3 macOS (via Homebrew)	36
2.2.4 Windows (mit WSL2)	37

2.2.5	Ansible testen mit dem ping-Modul	39
2.3	Eine sichere Spielwiese: Testumgebung mit HashiCorp Vagrant	39
2.3.1	Was ist Vagrant?	39
2.3.2	Vagrant trifft VirtualBox.	40
2.3.3	Virtualisierung im BIOS/UEFI aktivieren	41
2.3.4	Installation von Vagrant und VirtualBox	43
2.3.5	Virtuelle Umgebung einrichten mit Vagrant	44
2.3.6	Häufige Stolpersteine beim ersten Start von Vagrant & VirtualBox	54
2.4	Eigene IP-Adressen für unsere VMs	55
2.4.1	Netzwerkconfiguration im Vagrantfile	56
2.4.2	Netzwerkconfiguration prüfen	57
2.4.3	SSH-Verbindung manuell testen	59
2.5	Das Inventory: So erfährt Ansible von Ihren Servern.	60
2.6	SSH-Zugriff komfortabel und ohne Rückfragen.	62
2.7	Testumgebung managen mit Vagrant	62
2.7.1	Bedenkenlos experimentieren: Snapshotting.	63
2.7.2	Vagrant-Box auf Werkzustand zurücksetzen mit destroy.	66
2.7.3	Snapshot oder destroy?	68
2.7.4	Vagrant-Grundbefehle (Cheat Sheet)	69
2.8	Visual Studio Code: Ein praktischer Editor	70
2.8.1	Die Vorteile von VS Code.	70
2.8.2	Installation	70
2.9	Zusammenfassung	71
2.10	Wie geht es weiter?	72
2.11	Quiz: Installation, erste Schritte und Testumgebungen mit Ansible	72
3	Ansible verstehen.	75
3.1	Die Grundstruktur: Inventories, Playbooks, Tasks, Module und mehr	75
3.1.1	Ordnerstruktur planen – Best Practices für Ansible-Projekte	75
3.1.2	Inventories – Ansible braucht ein Adressbuch	77
3.1.3	Playbooks, Roles und Tasks – die Bausteine der Automatisierung.	79
3.1.4	Module.	81
3.1.5	Fully Qualified Collection Names (FQCN) – »Langform« der Modulnamen.	84
3.1.6	Ansible-Konfigurationsdatei (ansible.cfg).	85
3.2	YAML-Grundlagen.	86
3.2.1	Was ist YAML?	87
3.2.2	Grundstruktur: Einrückung ist entscheidend	87

3.2.3	Die wichtigsten YAML-Elemente im Überblick	88
3.2.4	Kommentare	89
3.2.5	Multiline-Strings und spezielle Formate	89
3.2.6	YAML validieren und testen	89
3.3	Zusammenfassung	90
3.4	Wie geht es weiter?	90
3.5	Quiz: Ansible-Grundlagen	91
4	Aus Theorie wird Praxis: Playbooks schreiben und anwenden	93
4.1	Editor auf, Fokus an – Projekt starten	93
4.2	Das erste echte Playbook.	95
4.3	Playbook ausführen und Ansible live erleben	99
4.4	Idempotenz in Aktion	101
4.5	Playbook erweitern – die Webserver bekommen dynamischen Inhalt	103
4.5.1	Ein neuer Task für die Startseite.	104
4.5.2	Ein Template vorbereiten.	104
4.5.3	Handler einbinden.	105
4.5.4	Das Template füllen.	106
4.5.5	Dienst starten	107
4.5.6	Handler definieren.	108
4.5.7	Ausführung mit -D bzw. --diff.	110
4.5.8	Test der Webseiten.	110
4.6	Flexible Playbooks mit Variablen	113
4.6.1	So verwenden Sie Variablen in Tasks.	114
4.7	Das zweite Playbook: Die Basiskonfiguration	115
4.8	Fakten (Facts) sammeln mit dem setup-Modul	119
4.9	Mit Facts auf Systeminformationen zugreifen	121
4.10	Eigene Facts setzen – Systeme mit Zusatzinformationen anreichern	124
4.11	Handlers – reagieren, wenn sich etwas ändert	126
4.12	Ad-hoc-Module – schnell und gezielt.	128
4.12.1	Typische Use-Cases	128
4.12.2	Playbooks vs. Ad-hoc-Module	131
4.13	Debugging und Fehlersuche	132
4.13.1	Ansible mit Verbose-Flags ausführen.	133
4.13.2	Der Check-Modus hilft beim Verstehen.	133
4.13.3	Das debug-Modul: Was steckt in den Variablen?	135
4.13.4	Mit when gezielt testen und ausschließen	137
4.13.5	Playbook gegen einen einzelnen Host laufen lassen mit --limit	137
4.13.6	Default-Werte	137
4.13.7	Fazit.	138

4.14	Zusammenfassung	138
4.15	Wie geht es weiter?	139
4.16	Quiz: Inventories, Playbook-Architektur und YAML-Grundlagen.	139
5	Vault und Roles: Sichere Variablen und strukturierte Automatisierung	141
5.1	Sicher mit Secrets – Einführung in Ansible Vault	142
5.1.1	Warum überhaupt Secrets verwalten?	142
5.1.2	Was ist Ansible Vault?	142
5.1.3	Verschlüsselte Variablen (Vault-Variablen) anlegen	143
5.1.4	Ein Playbook mit Vault-Variablen verwenden	145
5.1.5	Secrets aktualisieren: Vault-Werte austauschen.	146
5.1.6	Secrets in Dateien verwalten	148
5.2	Einführung in Roles – einmal schreiben, überall nutzen.	149
5.2.1	Roles erstellen und einsetzen	150
5.2.2	Ansible Galaxy nutzen: Fertige Roles aus der Community	152
5.2.3	Praxisbeispiel: WordPress mit Apache und MariaDB	153
5.3	Ansible Galaxy: Vorgefertigte Roles nutzen.	154
5.3.1	Use-Case: MariaDB-Role aus Galaxy verwenden	155
5.3.2	Struktur und Dokumentation von Galaxy-Roles	156
5.3.3	Vorteile und Grenzen.	156
5.4	Eigene Ansible-Roles versionieren, dokumentieren und mit anderen teilen.	157
5.4.1	Use-Case: WordPress-Roles im Team teilen	157
5.4.2	Veröffentlichung auf Ansible Galaxy	159
5.5	Zusammenfassung	160
5.6	Wie geht es weiter?	160
5.7	Quiz: Secrets & Roles	161
6	Hands-on: WordPress und MariaDB provisionieren und konfigurieren	163
6.1	Zielsetzung.	163
6.1.1	Das Zielbild im Überblick	163
6.2	Vorbereitung: Verzeichnis für die Roles erstellen.	164
6.3	Das Playbook vorbereiten: Roles orchestrieren	165
6.4	Apache-Role	167
6.4.1	SELinux-Policy verwalten	171
6.4.2	Die fertige Apache-Role	173
6.5	WordPress vorbereiten	176
6.5.1	WordPress auf beiden Webservern installieren	176

6.5.2	Konfigurationsdatei erstellen	179
6.5.3	Die fertige WordPress-Role	181
6.6	Variablen definieren und schützen	183
6.7	Datenbank bereitstellen – die Rolle mariadb	185
6.7.1	Datenbank anlegen	189
6.7.2	Datenbankbenutzer anlegen	191
6.7.3	Das gesamte Playbook	192
6.8	Optional: Playbook testen	194
6.8.1	Manuelle Verifikation auf db1	194
6.8.2	Test im Browser: Ist WordPress erreichbar?	195
6.9	Wiederverwendbarkeit in der Praxis	198
6.9.1	Use-Case 1: Gleiches Setup – neue Umgebung (Prod statt Lab)	198
6.9.2	Use-Case 2: Neue Anwendung – alte Roles wiederverwenden	204
6.9.3	Mehrere Datenbanken mit derselben Role anlegen: Ein erster Blick auf Schleifen in Ansible	210
6.10	Zusammenfassung	214
6.11	Wie geht es weiter?	215
7	Flexibilität und Stabilität: Erweiterte Steuerungsmechanismen in Ansible	217
7.1	Bedingte Anweisungen mit when – gezielt Aufgaben steuern	217
7.1.1	Use-Case: Nur unter Ubuntu ausführen	217
7.1.2	Use-Case: Datenbank-Task nur ausführen, wenn Variable gesetzt ist	218
7.2	Bedingungen im Detail: Vergleichsoperatoren und Prüfmechanismen	218
7.2.1	Prüfen, ob eine Variable existiert	219
7.2.2	Auf Nicht-Vorhandensein prüfen	219
7.2.3	Werte vergleichen	220
7.2.4	Prüfen, ob ein Wert in einer Liste enthalten ist	220
7.2.5	Kombinieren von Bedingungen	221
7.2.6	Optionale Konfigurationsdatei bereitstellen	221
7.2.7	Use-Case: Optionale Apache-Konfiguration pro Host	221
7.2.8	Operatoren auf einen Blick	224
7.3	Schleifen und Iterationen	224
7.3.1	Use-Case: Zwei Datenbanken anlegen	225
7.3.2	Use-Case: Mehrere MariaDB-Benutzer anlegen	225
7.3.3	Moderne vs. klassische Schleifen in Ansible: loop und with_<lookup>	231
7.4	Wiederholungen mit until – Aufgaben mit Retry-Logik	236

7.5	Standardwerte für optionale Variablen setzen	237
7.5.1	Use-Case: Benutzer mit optionalem Passwort	238
7.5.2	Weitere Einsatzmöglichkeiten	239
7.6	Fehlerbehandlung	239
7.6.1	Fehlerbehandlung mit block, rescue, always	239
7.6.2	Fehler gezielt auswerten mit failed_when und ignore_errors	241
7.7	Noch mehr Flexibilität mit Tags – gezielte Ausführung von Aufgaben	244
7.7.1	Was sind Tags in Ansible?	244
7.7.2	Use-Case: Roles im WordPress-Playbook gezielt anstoßen	245
7.7.3	Tags gezielt ausschließen	250
7.7.4	Spezialfall: always und never	251
7.8	Zusammenfassung	252
7.9	Wie geht es weiter?	252
7.10	Quiz: Playbooks intelligent steuern: Bedingungen, Schleifen und Fehlerbehandlung	253
8	Windows-Umgebungen verwalten	255
8.1	Kommunikation mit Windows – WinRM aktivieren und konfigurieren	255
8.2	Ein Windows-Testsystem mit Vagrant einrichten	258
8.3	Windows-Tasks mit Ansible automatisieren	261
8.3.1	Einen Ordner erstellen	262
8.3.2	Gruppe anlegen	263
8.3.3	Lokalen Benutzer anlegen	263
8.3.4	Windows Features aktivieren	264
8.3.5	Chocolatey: Paketverwaltung für Windows	264
8.3.6	Windows-Updates durchführen	265
8.3.7	Das Playbook ausführen und verifizieren	265
8.4	Zusammenfassung	269
8.5	Wie geht es weiter?	269
8.6	Quiz: Windows mit Ansible automatisieren	270
9	Container mit Ansible verwalten – Docker und Podman im Fokus	271
9.1	Grundlagen: Docker und Podman mit Ansible	272
9.1.1	Container-Images verwalten	272
9.1.2	Container entfernen, aktualisieren, verwalten	274
9.1.3	Volumes und Persistenz	276
9.2	Container-Images mit Ansible bauen	276
9.2.1	Use-Case: Eigene statische Website als Container	276

9.3	Container-Images in eine Registry pushen	279
9.4	Use-Case: Einen Nextcloud-Container aus einer Registry bereitstellen	281
9.5	Container reboot-safe machen: Systemd-Integration für Podman	284
9.6	Zusammenfassung	288
9.7	Wie geht es weiter?	289
9.8	Quiz: Container mit Ansible automatisieren	289
10	Qualitätssicherung für Ansible: Molecule & Linting	291
10.1	Grenzen klassischer Tests und wie Molecule sie überwindet	292
10.1.1	Der Dry-Run-Modus (--check)	292
10.1.2	Vagrant als Testumgebung – gut für den Einstieg, aber nicht skalierbar.	292
10.1.3	Unterschied zu QA-Tests mit Molecule	293
10.2	Vorbereitungen: Molecule installieren.	293
10.2.1	Use-Case: MariaDB-Roles testen mit Molecule	295
10.2.2	Molecule initialisieren	295
10.2.3	Testumgebung definieren: molecule.yml.	297
10.2.4	Converge-Playbook: converge.yml.	298
10.2.5	Testumgebung erstellen: create.yml.	299
10.2.6	Testumgebung sauber entfernen: destroy.yml.	302
10.2.7	Funktioniert die MariaDB-Datenbank wirklich? verify.yml.	303
10.2.8	Die wichtigsten Molecule-Befehle im Überblick	306
10.2.9	Molecule-Test in Aktion.	306
10.3	Mehrere Plattformen gleichzeitig testen	313
10.4	Playbook-Syntax prüfen: Linter im Einsatz	314
10.4.1	Linting Tools installieren.	314
10.4.2	Linting-Tests mit ansible-lint.	314
10.4.3	YAML-Syntax prüfen mit yamllint	320
10.4.4	Eigene Ansible-Lint-Regeln	323
10.5	Linting im Molecule-Kontext	324
10.6	Zusammenfassung	324
10.7	Wie geht es weiter?	325
10.8	Quiz: Qualitätssicherung für Ansible	325
11	CI/CD mit GitHub Actions	327
11.1	Use-Case: Automatisiertes Testen von Roles beim Push ins Repository.	327
11.1.1	Ihr erster Workflow	328
11.1.2	Von Fehlern lernen – aus Theorie wird Praxis	330
11.2	Tieferer Blick in einen Job	332

11.3	Tests auf mehreren Distributionen	332
11.4	Zusammenfassung	334
11.5	Wie geht es weiter?	334
11.6	Quiz: Ansible-Pipelines – Fehler finden, fixen, deployen.	334
12	Ansible in der Cloud – dynamische Inventories & AWS-Integration	337
12.1	Vorbereitung	337
12.1.1	AWS-Zugangsdaten	337
12.1.2	Abhängigkeiten installieren	340
12.1.3	Zugangsdaten sicher speichern	342
12.1.4	Zugangsdaten testen	343
12.2	Use-Case: MariaDB in AWS	344
12.2.1	Playbook anpassen	344
12.2.2	Dynamisches Inventory	352
12.2.3	Role anwenden	354
12.3	Best Practices für Ansible mit AWS	356
12.3.1	Sicherheit geht vor	356
12.3.2	Inventories & Struktur	357
12.3.3	Automation & Ausführung	357
12.3.4	Organisatorische Best Practices	357
12.4	Zusammenfassung	358
12.5	Wie geht es weiter?	358
12.6	Quiz: Ansible trifft AWS & dynamische Inventories	358
13	Packer – Golden Images für AWS mit Ansible bauen	361
13.1	Use-Case: MariaDB für ein Entwicklerteam	361
13.2	Packer	361
13.3	Projektstruktur	363
13.4	Packer-Template mit HCL erstellen	363
13.5	Ansible-Playbook für Provisionierung	366
13.6	AMI bauen mit Packer und Ansible	367
13.7	EC2-Instanzen mit dem eigenen AMI starten	369
13.8	Zusammenfassung	370
13.9	Wie geht es weiter?	371
13.10	Quiz: Packer – Golden Images	371
14	Ansible mit grafischen Oberflächen (GUIs)	373
14.1	GUIs für Ansible	374
14.1.1	Semaphore UI	374
14.1.2	Weitere GUIs	376
14.2	Installation von Semaphore UI	377

14.3	Optional: Installation mit dem Container-Konfigurator	378
14.4	Erstes Projekt in Semaphore UI erstellen	380
14.5	Strategische Automatisierung: Der Weg zu GitOps	386
14.6	Zusammenfassung	386
14.7	Wie geht es weiter?	387
14.8	Quiz: Arbeiten mit Semaphore	387
15	Ansible mit künstlicher Intelligenz (KI)	389
15.1	Konkurrenz oder Verstärkung?	389
15.1.1	Warum Infrastructure as Code weiterhin unverzichtbar ist	389
15.1.2	KI als Werkzeug für Ansible – nicht als Ersatz	390
15.2	Use-Case: Webserver-Deployment mit KI	391
15.3	Best Practices mit KI	394
15.3.1	Klein anfangen – schrittweise Vertrauen aufbauen	394
15.3.2	Iterativ verbessern – KI als Startpunkt nutzen	394
15.4	Weitere Einsatzmöglichkeiten von KI	395
15.4.1	Optimierungsvorschläge für Playbooks	395
15.4.2	Dokumentation erstellen	396
15.4.3	Sicherheits- und Best-Practice-Checks	396
15.4.4	Fehlersuche und Debugging	396
15.5	GitHub Copilot – KI-gestützte Ansible-Entwicklung	397
15.5.1	Was ist GitHub Copilot?	397
15.5.2	Einrichtung in VS Code	397
15.6	Zusammenfassung	399
15.7	Quiz: Künstliche Intelligenz und Ansible	400
15.8	Wie geht es weiter?	401
16	Best Practices, Strategien und Ausblick	403
16.1	Best Practices	403
16.1.1	Keep it simple – Einfachheit vor Komplexität	403
16.1.2	Modulare Struktur mit Roles und Collections	403
16.1.3	Inventories sauber trennen	404
16.1.4	Playbooks lesbar und wartbar halten	404
16.1.5	Code-Style und Qualitätssicherung	404
16.1.6	Idempotenz und Desired State	404
16.1.7	Sicherheit geht vor	405
16.1.8	Deployment und Lifecycle	405
16.1.9	Performance und Skalierung	405
16.1.10	Kultur und Zusammenarbeit	405
16.2	Die Zukunft von Ansible und Infrastructure as Code	406
16.2.1	Terraform trifft Ansible	406
16.2.2	Marke bleibt – Ökosystem wächst	407

16.2.3	Open-Source-Debatte bleibt spannend	407
16.2.4	Ausblick: Was bedeutet das für Sie als Ansible-Nutzer?	407
16.3	Abschließende Worte	408
16.3.1	Ansible lebt von der Community.	408
16.3.2	Ihr Weg mit Ansible ist noch nicht zu Ende	409
16.3.3	Zum Abschluss ein Dank.	409
	Stichwortverzeichnis	411

Die Sprache der Infrastruktur: Willkommen bei Ansible

1.1 Warum man Infrastruktur heute nicht mehr »per Hand« verwaltet

Stellen Sie sich vor, Sie müssten in einem Unternehmen mit zehn, zwanzig oder gar hundert Servern jeden einzelnen von Hand einrichten: Benutzerkonten anlegen, Software installieren, Konfigurationsdateien anpassen, Systemupdates einspielen. Jede noch so kleine Änderung müssten Sie manuell auf jedem System wiederholen. Fehler wären vorprogrammiert – sei es durch Flüchtigkeit, durch missverständliche Anleitungen oder durch minimale Unterschiede zwischen den Systemen.

Früher war das tatsächlich oft Realität. Traditionell wurden IT-Systeme manuell konfiguriert, doch in einer Zeit, in der IT-Systeme immer komplexer werden, Cloud-Plattformen wie AWS, Azure oder Google Cloud Platform dominieren und Unternehmen auf schnelle Reaktionsfähigkeit angewiesen sind, ist dieses Vorgehen schlicht nicht mehr zeitgemäß.

Genau hier setzt Infrastructure as Code (IaC) an: Anstatt manuell Server und Systeme zu konfigurieren, wird Infrastruktur wie ein Softwareprojekt behandelt – in Code gegossen, versioniert, überprüft und automatisiert ausgeführt.

Dies bringt zahlreiche Vorteile mit sich:

- **Konsistenz:** Jede Umgebung (z.B. Staging-, Test- und Produktionsumgebungen) kann identisch eingerichtet werden.
- **Wiederholbarkeit:** Änderungen und Deployments können automatisiert und reproduzierbar durchgeführt werden.
- **Effizienz:** Automatisierung spart Zeit und reduziert menschliche Fehler.
- **Skalierbarkeit:** Infrastruktur kann schnell und flexibel an neue Anforderungen angepasst werden.

Infrastructure as Code bedeutet, dass Sie Ihre IT-Infrastruktur – also z.B. Server, Netzwerke, Datenbanken oder Firewalls – mithilfe von Textdateien beschreiben, die ein Programm später automatisch ausführt. Diese Beschreibungen werden oft in einer menschenlesbaren Sprache verfasst, beispielsweise YAML (Yet Another Markup Language), wie es Ansible nutzt.

Analogie

So wie ein Architekt einen Bauplan erstellt, der festlegt, wie ein Haus gebaut werden soll, schreiben Sie als IT-Administrator einen »Bauplan« für Ihre Infrastruktur. Und Ansible ist dann die Baufirma, die diesen Plan automatisch umsetzt – immer gleich, immer zuverlässig.

1.1.1 Traditionelle IT vs. Infrastructure as Code

Merkmal	Traditionelle IT-Verwaltung	Infrastructure as Code
Provisionierung	Manuelle Einrichtung	Automatisiert per Code
Konfiguration	Manuelle Änderung	Deklarative Playbooks
Reproduzierbarkeit	Schwer nachvollziehbar	Versionierte Konfiguration
Fehlersuche	Zeitaufwendig, inkonsistent	Strukturierte, dokumentierte Prozesse

Tabelle 1.1: Vergleich von traditioneller IT und Infrastructure as Code

Die Umstellung auf Infrastructure as Code bringt zahlreiche Vorteile mit sich:

- **Automatisierung:** Aufgaben, die früher manuell erledigt wurden, laufen jetzt automatisiert ab.
- **Konsistenz:** Alle Systeme werden identisch konfiguriert – Konfigurationsdrift wird vermieden.
- **Versionierbarkeit:** Jede Änderung am Infrastruktur-Code kann nachvollzogen, dokumentiert und bei Bedarf rückgängig gemacht werden.
- **Reproduzierbarkeit:** Ein neuer Server kann auf Knopfdruck genauso eingerichtet werden wie ein bestehender.
- **Teamarbeit:** Der Code kann gemeinsam entwickelt, geprüft und getestet werden – ganz wie bei der Softwareentwicklung.

Diese Eigenschaften machen IaC nicht nur in großen IT-Abteilungen interessant, sondern auch für kleine Teams, Start-ups oder Einzelpersonen, die ihre Arbeitsweise professionalisieren wollen.

1.1.2 Deklarativ statt prozedural: Denken in Zielzuständen

Ein zentrales Konzept, das Ansible von vielen traditionellen Ansätzen zur Systemverwaltung unterscheidet, ist die sog. *deklarative Arbeitsweise*. Statt in Anweisungen (»Tu dies, dann das«) denken Sie in Zielzuständen: Sie beschreiben, *wie* ein System am Ende aussehen soll, und Ansible kümmert sich darum, die nötigen Schritte dorthin zu finden und umzusetzen.

Traditionelle (imperative) Verwaltung

In klassischen Skripten – etwa mit Bash, PowerShell oder Python – schreiben Sie Schritt für Schritt vor, *wie* ein gewünschtes Ergebnis zu erreichen ist:

```
#!/bin/bash

# Benutzer anlegen
useradd erika

# Home-Verzeichnis anlegen
mkdir -p /home/erika/.ssh
chown -R erika:erika /home/erika
chmod 700 /home/erika/.ssh

# SSH-Key setzen
echo "ssh-rsa AAAAB3..." > /home/erika/.ssh/authorized_keys
chown erika:erika /home/erika/.ssh/authorized_keys
chmod 600 /home/erika/.ssh/authorized_keys
```

Der Administrator legt dabei exakt fest, welche Schritte in welcher Reihenfolge ausgeführt werden. Das zentrale Problem solcher Skripte ist jedoch, dass sie nicht zustandsbasiert arbeiten. Sie prüfen in der Regel nicht, ob ein gewünschter Zustand bereits erreicht ist, und lassen sich daher nur eingeschränkt wiederverwenden oder gefahrlos mehrfach ausführen. Existiert ein Benutzer bereits oder ist ein SSH-Schlüssel schon hinterlegt, schlägt das Skript häufig fehl – oder liefert in leicht veränderten Umgebungen plötzlich andere Ergebnisse.

Der deklarative Ansatz von Ansible

Mit Ansible definieren Sie, **was** gelten soll, nicht, **wie** man dorthin kommt. Zum Beispiel:

```
- name: Benutzer anlegen
  ansible.builtin.user:
    name: erika
    state: present

- name: SSH-Key setzen
  ansible.posix.authorized_key:
    user: erika
    key: "ssh-rsa AAAAB3..."
```

Sie müssen dabei nicht mehr jeden einzelnen Shell-Befehl kennen oder darauf achten, ob ein Benutzer bereits existiert, ob das SSH-Verzeichnis korrekt angelegt ist oder ob die Dateiberechtigungen stimmen. Das jeweilige Ansible-Modul kapselt diese Details und stellt sicher, dass der gewünschte Zustand zuverlässig, wiederholbar und nachvollziehbar erreicht wird.

Nur wenn der gewünschte Zustand noch nicht erreicht ist, greift Ansible ein. Dieses Prinzip nennt man Idempotenz: Die gleiche Automatisierung kann mehrfach ausgeführt werden, ohne Systeme jedes Mal erneut zu verändern oder »kaputt zu konfigurieren«.

Die deklarative Arbeitsweise mag anfangs etwas ungewohnt wirken, doch sie bildet die Grundlage für stabile, überprüfbare und automatisierte IT-Infrastruktur. Sie schreiben nicht mehr Code, um Aufgaben zu erledigen, sondern beschreiben den gewünschten Zustand, und Ansible sorgt für die Umsetzung.

1.1.3 Warum der deklarative Ansatz insbesondere Einsteigern hilft

Der deklarative Ansatz verändert die Lernkurve für Einsteiger grundlegend. Statt komplexe Befehlsfolgen und deren Seiteneffekte im Detail verstehen zu müssen, beschreiben Sie mit Ansible den gewünschten Zielzustand eines Systems. Wie dieser Zustand erreicht wird, übernimmt Ansible.

Früher war es notwendig, tiefgreifende technische Details zu kennen, um etwa ein RAID korrekt einzurichten oder Benutzer konsistent anzulegen. Mit Ansible lassen sich dieselben Ergebnisse heute mit wenigen, gut lesbaren YAML-Zeilen beschreiben. Das senkt die Einstiegshürde deutlich:

- **Komplexe Aufgaben werden verständlich** – auch ohne jahrelange Erfahrung.
- **Anleitungen zur Automatisierung sind gut lesbar** – man erkennt sofort, welches Ziel damit erreicht werden soll.
- **Fehleranfällige manuelle Schritte entfallen** – Ansible sorgt für Wiederholbarkeit und Korrektheit.
- **Einsteiger werden schneller produktiv** – viele Aufgaben lassen sich schon nach wenigen Wochen sicher automatisieren.

Zusätzlich unterstützt Ansible diesen Ansatz durch bewusst einfache Rahmenbedingungen. Ansible arbeitet agentlos (engl. *agentless*): Auf den Zielsystemen muss keine zusätzliche Software installiert werden, abgesehen von einer vorhandenen Python-Laufzeitumgebung, die in den meisten Linux-Distributionen bereits mitgeliefert wird. Es läuft auf gängigen Plattformen wie Linux, macOS oder Windows (via WSL) und kommt mit einer überschaubaren Installation aus. Die verwendete YAML-Syntax ist bewusst einfach gehalten und erleichtert das Lesen und Verstehen auch ohne tiefgehende Programmierkenntnisse.

Mit Ansible denken Sie daher nicht mehr in einzelnen Befehlen, sondern in gewünschten Zuständen. Das vereinfacht nicht nur den Alltag erfahrener Admi-

nistratoren, sondern ermöglicht auch Einsteigern, früh produktive und nachvollziehbare Automatisierungen umzusetzen.

1.2 Was ist Ansible?

Ansible ist eines der führenden Werkzeuge für Infrastructure as Code und zeichnet sich durch seine einfache Handhabung und hohe Flexibilität aus. Es wurde 2012 von Michael DeHaan entwickelt und hat sich seitdem zu einem Standard für IT-Automatisierung entwickelt.

1.2.1 Wie Ansible zu seinem Namen kam

Der Name Ansible stammt – wie so viele Begriffe in der Tech-Welt – aus der Science-Fiction. Genauer ausgedrückt beschreibt ein Ansible ein fiktives Gerät, das in der Lage ist, Informationen augenblicklich über beliebige Entfernungen hinweg zu übertragen – also schneller als Lichtgeschwindigkeit. Die Idee dieses überlichtschnellen Kommunikationsmittels wurde erstmals von der amerikanischen Autorin Ursula K. LeGuin in ihrem Roman »Raconnas Welt« (1966) eingeführt. In ihrem Werk diente das Ansible als Verbindung zwischen weit entfernten Welten.

Später wurde der Begriff von anderen Science-Fiction-Autoren weiterverwendet, darunter auch Orson Scott Card in »Das große Spiel« (1977). In diesem Roman spielt das Ansible eine zentrale Rolle in der militärischen Fernsteuerung ganzer Raumschiff-Flotten – ohne Zeitverzögerung, zuverlässig und synchron.

Diese Metapher griff Michael DeHaan, einer der ursprünglichen Entwickler von Ansible, auf. Für ihn war der Name eine perfekte Beschreibung dessen, was seine Software leisten sollte: eine zentrale Instanz, die viele entfernte Systeme steuern kann – sofort, koordiniert und ohne unnötigen Ballast. Statt Befehle manuell auf jedem Server auszuführen, übernimmt Ansible diese Aufgabe automatisch, als stünde es »in direkter Verbindung« mit allen Maschinen – ganz wie das fiktive Original. So wurde aus einer literarischen Vision eine moderne Automatisierungslösung – und aus dem Begriff *Ansible* ein Synonym für effiziente IT-Steuerung über jede Distanz hinweg.

1.2.2 Die Stärken von Ansible

Ansible hat sich in den letzten Jahren als eines der am häufigsten eingesetzten Werkzeuge für IT-Automatisierung etabliert. Der Grund dafür liegt weniger in einzelnen Spezialfunktionen als in einem insgesamt sehr pragmatischen Ansatz: Ansible ist leicht zu erlernen, arbeitet agentenlos und beschreibt Infrastrukturzustände klar und deklarativ.

Statt komplexer Programmlogik formulieren Sie in übersichtlichen YAML-Dateien, wie ein System aussehen soll. Ansible übernimmt anschließend die Umsetzung dieses Zielzustands. Dadurch lassen sich wiederkehrende Aufgaben automatisieren, menschliche Fehler reduzieren und konsistente, reproduzierbare Umgebungen schaffen – vom einzelnen Server bis hin zu größeren, mehrstufigen Architekturen.

Gegenüber anderen Automatisierungstools bietet Ansible mehrere Vorteile :

- **Agentenloses Arbeiten:** Auf den Zielsystemen ist keine dauerhaft laufende Software erforderlich. Ansible nutzt bestehende Mechanismen wie SSH, WinRM oder APIs, was Betrieb und Absicherung vereinfacht.
- **Einfache und gut lesbare Syntax:** Automatisierungen werden in YAML beschrieben und sind auch ohne Programmierhintergrund gut nachvollziehbar.
- **Deklarativer Ansatz und Idempotenz:** Aufgaben werden nur dann ausgeführt, wenn sie tatsächlich notwendig sind. Wiederholte Ausführungen führen nicht zu unerwünschten Nebeneffekten.
- **Modularität und Wiederverwendbarkeit:** Durch Roles, Templates und Variablen lassen sich Automatisierungen sauber strukturieren und projektübergreifend einsetzen.
- **Breite Unterstützung:** Ansible kann Linux-, macOS- und Windows-Systeme ebenso verwalten wie Netzwerkgeräte, Virtualisierungsplattformen, Container-Umgebungen und Cloud-Dienste.
- **Kostenlos:** Es ist frei verfügbar und wird aktiv weiterentwickelt.



Abb. 1.1: Einige Systeme und Plattformen, mit denen Ansible interagieren kann

Anders als bei komplexeren Tools wie Terraform oder Puppet können Sie mit Ansible oft schon nach wenigen Stunden erste produktive Ergebnisse erzielen. Deshalb eignet es sich hervorragend für den Einstieg in die Welt der IT-Automatisierung.

Praxistipp

Ansible eignet sich besonders gut für Aufgaben wie das Installieren von Softwarepaketen, das Anpassen von Konfigurationsdateien oder das Starten von Diensten – also für das, was Systemadministratoren tagtäglich tun.

1.2.3 Einsatzbereiche in der Praxis

Ansible ist ein vielseitiges Werkzeug zur Automatisierung von IT-Infrastruktur. Es eignet sich besonders für Aufgaben, die wiederholt, zuverlässig und auf mehreren Systemen gleichzeitig ausgeführt werden müssen.

Typische Einsatzbereiche sind:

- Konfigurationsmanagement: Systeme in einen definierten Zustand versetzen, z.B. Pakete installieren oder Dienste aktivieren
- Provisionierung: Neue Server oder Cloud-Ressourcen automatisiert bereitstellen
- Deployment: Anwendungen inklusive Abhängigkeiten auf mehreren Hosts ausrollen
- Orchestrierung: Abhängige Systeme in der richtigen Reihenfolge einrichten oder steuern (z.B. erst Datenbank, dann Webserver)
- Patch-Management: Sicherheitsupdates zeitgleich auf vielen Systemen verteilen

Kurz: Ansible ist dann besonders hilfreich, wenn viele Systeme effizient, wiederholbar und nachvollziehbar verwaltet werden sollen, egal ob im Rechenzentrum, in der Cloud oder im gemischten Betrieb.

1.2.4 Ist Ansible mächtig genug?

Viele IT-Administratoren, die seit Jahren Shell- oder Bash-Skripte zur Systemautomatisierung einsetzen, stehen Ansible zunächst mit einer gewissen Skepsis gegenüber. Sie fragen sich, ob ein deklaratives Tool wie Ansible tatsächlich alle komplexen Anforderungen abdecken kann oder ob es nicht doch sinnvoller ist, bei ihren bewährten, individuell geschriebenen Skripten zu bleiben.

Diese Sorge ist verständlich, aber unbegründet. Tatsächlich bietet Ansible nicht nur eine alternative Methode zur Automatisierung, sondern eröffnet Möglichkei-

ten, die weit über klassische Shell-Skripte hinausgehen – insbesondere was Wartbarkeit, Wiederverwendbarkeit, Lesbarkeit und Flexibilität betrifft.

Skripte ersetzen – ohne Einbußen bei der Kontrolle

Ein zentrales Missverständnis ist die Annahme, Ansible sei »zu einfach« oder »zu starr«, um komplexe Szenarien zu bewältigen. Dabei kann Ansible:

- komplexe Abläufe dynamisch steuern,
- Daten aus vorherigen Tasks wiederverwenden,
- abhängig vom Systemzustand verschiedene Aktionen ausführen,
- Schleifen, Bedingungen und Variablenkontexte nutzen
- und sogar direkt Shell-Befehle einbetten, wenn nötig.

All das in einer klar strukturierten, modularen und wiederverwendbaren Form.

Beispiel: Passwortwechsel während eines Datenbank-Setups

Ein häufiger Zweifel bei deklarativen Werkzeugen ist die Frage, ob sie auch mit Zustandswechseln innerhalb eines Ablaufs umgehen können. Genau das ist in der Praxis jedoch oft notwendig, etwa bei der initialen Einrichtung einer Datenbank.

Ein typisches Beispiel: Sie richten eine neue Datenbankinstanz ein, die zunächst mit einem bekannten Default-Passwort wie »changeme« erreichbar ist. Ziel der Automatisierung ist es

1. sich mit dem Default-Passwort zu verbinden,
2. ein neues Passwort zu setzen,
3. und anschließend direkt mit den neuen Zugangsdaten weiterzuarbeiten, ohne den Ablauf zu unterbrechen oder ein zweites Mal starten zu müssen.

Mit Bash-, PowerShell- oder Python-Skripten ist das häufig mit temporären Hilfsdateien, Zwischenspeicherung oder komplexen Kontrollstrukturen verbunden. In Ansible hingegen lässt sich dieses Szenario elegant in einem einzigen Playbook abbilden.

Ansible hingegen arbeitet kontextbasiert weiter, ohne dass ein zweites Skript gestartet werden oder manuell eingegriffen werden muss. Es nutzt zunächst das alte Passwort, wechselt dann intern zu den neuen Zugangsdaten und setzt den Ablauf nahtlos fort.

Warum das mehr ist als nur ein »Ersatz« für Bash-Skripte

Das vorherige Beispiel zeigt bewusst mehr als nur eine technische Spielerei. Es verdeutlicht, dass Ansible nicht lediglich einzelne Befehle automatisiert, sondern komplexe Abläufe strukturiert abbilden kann – inklusive Zustandswechseln,

Abhängigkeiten und Wiederverwendbarkeit. Genau an dieser Stelle unterscheidet sich Ansible grundlegend von klassischen Skriptlösungen.

Zusätzlich zu solchen dynamischen Kontrollflüssen bietet Ansible:

- **Templating mit Jinja2:** Konfigurationsdateien lassen sich mit Platzhaltern und einfacher Logik generieren.
- **Roles und Wiederverwendung:** Aufgaben können modular aufgebaut, standardisiert und projektübergreifend verwendet werden.
- **Variablen in mehreren Ebenen:** Werte lassen sich Host-, Gruppen- oder kontextabhängig sowie zur Laufzeit setzen.
- **Handler und Idempotenz:** Änderungen werden nur dann durchgeführt, wenn sie tatsächlich notwendig sind – ideal für wiederholte Ausführungen.
- **Vault für sichere Passwörter:** Zugangsdaten und sensible Daten können verschlüsselt verwaltet werden.

Ansible ist damit kein reiner Ersatz für Bash-Skripte, sondern ein vollständiges Automatisierungs-Framework für produktive Umgebungen. Es nimmt Administratoren keine Kontrolle ab, sondern ersetzt implizite Logik durch nachvollziehbare Struktur, Wiederverwendbarkeit und Sicherheit.

Sie müssen sich daher nicht zwischen Skripten und Ansible entscheiden. Vielmehr erlaubt Ansible, bestehende Automatisierungen schrittweise zu verbessern und gleichzeitig verständlicher und wartbarer zu gestalten – für Sie selbst ebenso wie für Ihre Kolleginnen und Kollegen.

1.3 Einführung in Ansible

1.3.1 Die wichtigsten Komponenten

Bevor Sie mit Ansible produktiv arbeiten können, ist es wichtig, die grundlegenden Bausteine zu verstehen, aus denen sich Ihre Automatisierung zusammensetzt. Ansible verfolgt dabei einen modularen und klar strukturierten Ansatz. Vier zentrale Konzepte bilden das Fundament: Playbooks, Module, Inventories und Roles. Sie greifen wie Zahnräder ineinander und ermöglichen es, selbst komplexe Aufgaben systematisch und wiederverwendbar zu automatisieren.

Im Mittelpunkt stehen die Playbooks. Dabei handelt es sich um einfache Textdateien im YAML-Format, in denen beschrieben wird, welche Aufgaben auf welchen Systemen ausgeführt werden sollen. Ein Playbook ist gewissermaßen das »Drehbuch« Ihrer Automatisierung – es legt fest, welche Schritte wann und in welcher Reihenfolge erfolgen.

Damit diese Aufgaben nicht jedes Mal im Detail programmiert werden müssen, greift Ansible auf eine große Sammlung von Modulen zurück. Diese Module sind

kleine, spezialisierte Programme, die bestimmte Aufgaben übernehmen – zum Beispiel das Anlegen von Benutzerkonten, das Installieren von Software oder das Starten eines Dienstes. Sie sind der eigentliche »Motor« unter der Haube und ersparen Ihnen viel Aufwand.

Ein weiterer zentraler Bestandteil ist das sogenannte Inventory. Hierbei handelt es sich um eine Datei oder Datenquelle, in der festgelegt ist, auf welchen Servern Ansible die beschriebenen Aufgaben ausführen soll. Diese Server können lokal, in einer Cloud oder in einer hybriden Umgebung liegen. Durch die klare Trennung zwischen *Was* (Playbook) und *Wo* (Inventory) bleibt Ihre Automatisierung übersichtlich und flexibel.

Für größere oder wiederkehrende Aufgaben empfiehlt es sich, mit Roles zu arbeiten. Roles bündeln alle relevanten Bestandteile – also Aufgaben, Variablen, Vorlagen und Dateien – in einer einheitlichen Struktur. Dadurch lassen sich einzelne Automatisierungskomponenten leicht wiederverwenden, teilen oder modular kombinieren. Wenn Sie zum Beispiel eine Role für das Einrichten eines Webserver erstellt haben, können Sie diese in verschiedenen Projekten oder Umgebungen immer wieder einsetzen, ohne von vorn beginnen zu müssen.

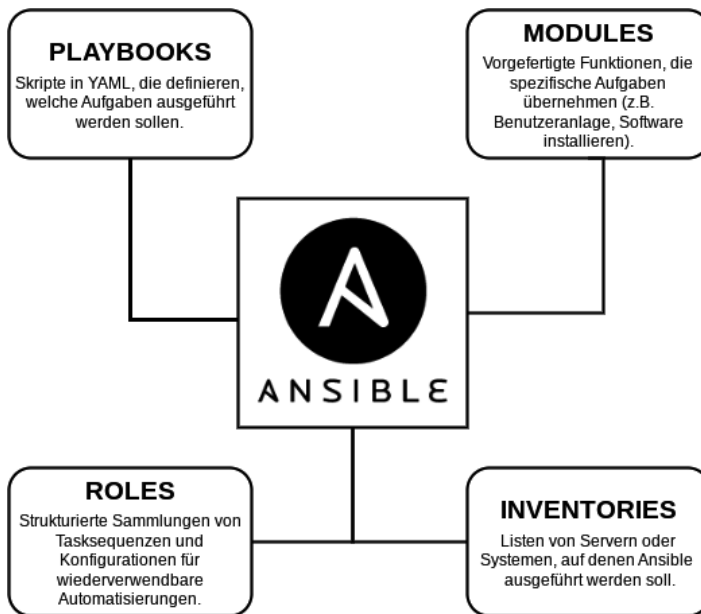


Abb. 1.2: Die Komponenten von Ansible

Zusammen ermöglichen Ihnen diese vier Komponenten ein sauberes, nachvollziehbares und skalierbares Arbeiten mit Ansible – ganz gleich, ob Sie eine Handvoll Testsysteme oder eine komplexe Serverlandschaft verwalten möchten.

Stichwortverzeichnis

- 404 242
- skip-tags 250
- != 220
- == 220
- A**
- Abhängigkeit 171, 176
- Ad-hoc-Modul 128
 - Nachteile 131
 - Vorteile 131
- Adressbuch 60, 77
- Agentless 20
- Agentlos 20
- always 239, 251
- Amazon Machine Image (AMI) 361
- AMI
 - Build 367
 - ID 369
- and 221
- Ansible 21
 - Installation 35
- ansible-lint 314
- ansible.builtin.debug 135
- ansible.cfg 76, 85, 86
- Ansible Automation Platform (AAP) 376
- Ansible Galaxy 82, 152, 154, 157
 - Readme 156
 - veröffentlichen 159
 - Versionskonflikte 156
- Anthropic 389
- Anweisung
 - bedingte 217
- Apache 78
 - 403-Fehler 172
 - Role 167
- APIs 22
- Arbeitsweise
 - deklarative 18
- Automatisierung 21, 403
- AWS 17, 78, 337, 344
 - Abhängigkeiten installieren 340
 - Automation 357
 - Best Practices 356
 - botocore 341
 - Fehler 344
 - Python 342
 - Root-Benutzer 339
 - Sicherheit 356
 - Struktur 357
 - Zugangsdaten 337
 - Zugangsdaten speichern 342
 - Zugangsdaten testen 343
- AWX 376
- Azure 17, 78, 337
- B**
- Basiskonfiguration 115
- Bedingte Anweisung 217
- Bedingungen 218
- Benutzer
 - mehrere anlegen 225
- Bind-Adresse 186
- block 239
- Boolean 172
- C**
- ChatGPT 389
- Check-Modus 133
- Chocolatey 264
- CI/CD 41
- CI/CD-Pipeline 327, 386
 - Variable 114
- Claude Code 389
- Cleanup 239
- Cloud
 - Testen 293
- Cloud-Integration 337
- Collection 82, 84
- command 103
- community.general 82
- Container 271
 - automatisch starten 284
 - bauen 276
 - löschen 275
 - Persistenz 276
 - verwalten 274
- Container-Engines 271
- Container-Konfigurator 378
- Containerfile 276
- Control Node 27, 35, 261
- curl 110
- D**
- Daten
 - sensible 141
- Datenbank
 - Backup einspielen 245
 - bereitstellen 185
 - Verifikation 194
- Datenbankserver 52
- Datenbankverbindung
 - testen 237
- Debian 36
- debug-Direktive 135
- Debugging 132, 251
 - when 137
- default 238
- Default-Werte 137

- Deployment 23
- DevOps 103
- Dictionary 88
- Dienst
 - abfragen 129
 - starten 107
 - Status 130
- dnf 97
- DNS
 - Auflösung prüfen 237
- Docker 271
- Dockerfile 276
- Docker Hub 279
- Dokumentation 158, 171
- Dry-Run-Modus 292
- E**
- EC2
 - eigenes AMI 369
- EC2-Instanz 344, 351
- EC2-Keypair 345
- Exit-Code 243
- Exit-Code 1 242
- F**
- Facts 119, 121, 218
 - eigene setzen 124
- failed_when 242, 243
 - block-Kontext 243
 - Listen 243
- Fakten siehe Facts 119
- Fehlerbehandlung 239
- Fehlerstatus 242
- Fehlersuche 132
- file context mapping definitions 171
- Firewall 111
 - Ports öffnen 169
- foo 219
- FQCN 84
- G**
- GCP 337
- Git 157, 159
 - neues Repository 382
- GitHub 381
- Personal Access
 - Token 381
- GitHub Actions 327
 - Job 332
 - mehrere Distributionen 332
 - Workflow 328
- GitHub Copilot 389, 397
 - in VS Code 397
- GitOps 386
- Golden Image 361, 366, 370
 - Sicherheits 370
- Google Cloud Platform 17
 - group_vars 76
- Gruppe
 - Variable 113
 - verschachtelte 61
- GUI 373
- H**
- Handler 25, 105, 126
 - Anwendungsfälle 127
 - definieren 108
 - Fehlerbehandlung 241
 - Hierarchie 128
 - Syntax 126
- HashiCorp 39
- HashiCorp Cloud Platform 45
- HashiCorp Packer 361
- HCL 363
- Homebrew 36
- Host 221
 - definieren 96
 - Variable 114
- host_vars 76
- Hostname 121
- HTTPS 257, 258
- Hyper-V 38
- I**
- IaC 17
- IAM 367
- Idempotenz 20, 100, 101
- ignore_errors 242
- Image 44
 - Variablen 280
 - veröffentlichen 280
 - verwalten 272
- in 220
- Infrastructure as Code 17
- Installation 35
- inventories 76
- Inventory 26, 27, 60, 77, 383
 - Beispiel 77
 - dynamisches 78, 337, 352
 - statisches 78
- IP 56, 199
- IP-Adresse
 - private 56
- is defined 218
- is not defined 219
- J**
- Jinja2 104
 - Dateiendung 105
 - Variable 115
- JSON 82
- K**
- Key-Value-Paar 88
- Kommentar 89
- Komponenten 25
 - wichtigste 25
- Konfiguration
 - Host-spezifische 221
 - systemübergreifend 115
- Konfigurationsdatei 85
 - Priorität 85
- Konfigurationsmanagement 23
- Kubernetes 271
- Künstliche Intelligenz 389
 - als Assistent 391
 - Best Practices 394
 - Halluzinationen 390

L

Lab 81
 LAMP-Setup 163
 LeGuin, Ursula K. 21
 Linter 314
 Linux
 Codename 121
 Versionsnummer 121
 Linux-Distribution 121
 Liste 88, 211
 Log 374
 Logfile 239
 loop 169, 210, 224

M

macOS 36
 Python 37
 Managed Node 27
 MariaDB 153, 163, 225
 Testumgebung 361
 Maschine
 virtuelle 98
 Matrix-Strategie 332
 Modul 25, 27, 80, 81
 Ad-hoc 128
 Collections 82
 Dokumentation 109
 finden 82
 shell 130
 Übersicht 97
 Molecule 291
 Befehle 306
 Container-Image 298
 Installation 293
 mehrere Plattformen 302
 mehrere Plattformen gleichzeitig testen 313
 Playbooks 297
 Role 298
 Role testen 295
 Szenarien erstellen 296
 Testumgebung erstellen 299
 molecule.yml 297

MySQL 226

N

Namenskonflikte 84
 Netzwerk
 internes 56
 Netzwerkkonfiguration 196
 Netzwerkschnittstelle 58
 Nextcloud 272, 281
 nginx 276

O

Oberfläche
 grafische 373
 Operations 374
 Operatoren 224
 or 221
 Oracle 40
 Orchestrierung 23
 Ordnerstruktur 75

P

Packer 361
 Build des Templates 365
 Projektstruktur 363
 Template 363
 Paket
 installieren 98
 Zustand 98
 Paketquelle
 aktualisieren 118
 Passwort
 aus Datei lesen 148
 Passwortwechsel 24
 Patch-Management 23
 phpBB 204
 ping 39, 261
 Pipelines 41
 Playbook 25, 27, 28, 79, 93, 113
 -i 78
 auf Host beschränken 137
 ausführen 99
 erneut ausführen 100

erstellen 95
 generieren 391
 testen 303

playbooks 76
 Podman 271, 272
 Port 81, 196
 offener 237
 WinRM 260
 PowerShell 256, 258, 374
 Privilege Escalation 96
 Produktivumgebung 81
 Projekt
 Grundstruktur 75
 laden 93
 Struktur 75
 Provisionierung 23, 40, 205
 Prüfmechanismen 218
 Pull-Modell 28
 Puppet 23
 Push-Modell 27
 vs. Pull-Modell 28
 Python 34, 255, 261
 Python 3 34
 pywinrm 261

Q

Qualitätskontrolle 327
 Qualitätssicherung 291
 Query 232

R

RBAC 373
 README.md 76, 158
 reboot-safe 284
 Repository 327
 rescue 239
 RHEL 36
 Rocky 36, 44
 Role 26, 27, 80, 141, 149
 Ansible Galaxy 152
 Apache 167
 AWS 152
 Bedingungen 154
 Cloud 354
 dokumentieren 157, 158

- gezielt ausführen 245
 - Git 159
 - in einem Playbook
 - verwenden 151
 - orchestrieren 165
 - praktische Umsetzung 163
 - Praxisbeispiel 153
 - Reihenfolge der Ausführung 167
 - Struktur automatisch erstellen 204
 - Test 327
 - testen 295
 - Variable 114, 152
 - Veröffentlichung 159
 - versionieren 157
 - Verzeichnisstruktur 150
 - vorgefertigte 154
 - wechseln 206
 - Role-Based Access Control 373
 - roles 76
 - Rolle siehe Role 26
 - Rundeck 376
- S**
- Schleife 210, 224, 231
 - klassische 231
 - moderne 231
 - Schlüsselspeicher 380
 - Secret 141, 142
 - aktualisieren 146
 - SELinux 171, 172
 - Semaphore UI 374
 - aufrufen 378
 - Container 377, 378
 - Git 380
 - Installation 377
 - Inventory erstellen 383
 - Log 384
 - Playbook ausführen 384
 - Port 378
 - Projekt anlegen 380
 - Task-Vorlage 384
 - Server
 - einrichten 45
 - service 129
 - setup (Modul) 120
 - shell 103
 - Snapshot 63, 68
 - Socket
 - öffnen 237
 - Software 34
 - SSH 49, 200
 - privater Schlüssel 59
 - Warnungen vermeiden 62
 - SSH-Key 61
 - SSH-Verbindung
 - testen 59, 351
 - String 89
 - Vergleich 220
 - Subscription-Management 36
 - systemd 284
 - Systemd-Integration 284
 - systemd-Unit 285
 - Systemvoraussetzungen 33

T

 - Tag 244
 - ausschließen 250
 - innerhalb einer Role 247
 - mehrere kombinieren 247
 - Syntax 244
 - Use-Case 245
 - Task 80, 141
 - definieren 97
 - einzelnen ausführen 247
 - selektiv ausführen 244
 - Template 103, 141, 180
 - Terraform 23, 374
 - Test 292
 - Daten 305
 - Datenbank 305
 - inhaltlicher 303
 - Port-Test 305
 - Testen
 - automatisierte Role-Tests 327
 - mehrere Testfälle 296
 - Testumgebung 39, 45, 63, 98, 165, 292
 - Systemvoraussetzungen 33
 - Texteditor 70
 - Trainingsszenario 53

U

 - Ubuntu 36
 - Umgebung
 - wechseln 198
 - until 236
 - Use-Case 157, 198, 204, 221

V

 - Vagrant 33, 39, 292
 - auf Werkszustand zurücksetzen 66
 - Box 44
 - Box-Version 47, 52
 - destroy 66
 - Installation 43
 - Netzwerkkonfiguration 56
 - Produktivumgebung simulieren 198
 - Snapshot 63
 - Status prüfen 99
 - Windows 256
 - Windows-Testsystem 258
 - Vagrant Cloud 44
 - Vagrantfile 46, 56
 - Variable
 - auslesen 219
 - Best Practices 114
 - default 238
 - defaults/main.yml 114
 - definieren 183, 200
 - Facts 119

- fehlende 237
- group_vars/ 113
- host_vars/ 114
- im Playbook 113
- Inhalt abfragen 135
- in Tasks 114
- is defined 219
- Kommandozeile 114
- optionale 221
- Präfix 123
- Speicherort 184
- Standardwert 237
- Übersicht aller speziellen 107
- vars/main.yml 114
- Vault 142, 163
 - Block 144
 - Passwort 184
- Vault-Variable 143
 - aktualisieren 146
 - verwenden 145
- Verbose 133
- Verbose-Flag 133
- Vergleichsoperator 218
- verschlüsseln
 - Variable 143
- Verschlüsselung 143, 149
- VirtualBox 34
 - Installation 43
- Virtualisierung 33
 - im BIOS/UEFI aktivieren 41
 - Laptop 42
 - Vagrant 34, 39
 - VirtualBox 34, 40
- Virtuelle Maschine
 - verbinden mit 51
- Virtuelle Umgebung
 - erstellen 44
- Visual Studio Code 34, 70
- VM 45
 - Windows 259
- Volume 276
- VS Code 70, 89
 - Installation 70
- W**
- Webserver 78
 - aktualisieren 250
- Werkzustand 134
- when 118, 217
- Wiederholung 236
- Windows 37, 255
 - Benutzer anlegen 263
 - Benutzerkonten 257
 - Dienste aktivieren 264
 - Gruppen anlegen 263
 - Module 269
 - Ordner erstellen 262
 - Pakete 265
 - Paketverwaltung 264
 - Playbook ausführen 265
 - Tasks automatisieren 261
 - Updates 265
- Windows-Host
 - Verbindung testen 261
- Windows Server 259
- Windows Subsystem for Linux 37
- WinRM 22, 255
 - aktivieren 255
 - Firewall 256
 - Port 260
- with_items 232
- WordPress 153, 163
 - Fehler 198
 - herunterladen 177
 - Installation 163
 - Konfigurationsdatei 179
 - PHP-Komponenten 176
 - vorbereiten 176
- WSL 37
- X**
- Xcode 37
- Y**
- YAML 17, 86
 - Einrückungen 87
 - Syntax prüfen 314
 - validieren 89
- yamllint 320
- Z**
- Zustand 87