Ansible

Das Praxisbuch für Administratoren und DevOps-Teams

Kapitel 1
Einführung und Installation

In diesem Kapitel möchte ich Ihnen erklären, was Ansible ist, und auch, was es nicht ist. Danach zeige ich Ihnen eine Möglichkeit, mit der Sie auf direktem Wege zu einer Testumgebung gelangen, sodass wir schließlich auf die Ansible-Installation eingehen können.

1.1 Was ist Ansible?

Ansible ist ein Werkzeug zur Lösung von Problemen, die sich meist auf der »Ops«-Seite der DevOps-Welt stellen. Jeder, der schon einmal für eine kleine Anzahl von Servern verantwortlich war, wird bestätigen können, dass eine rein manuelle Administration bzw. Konfiguration zumindest sehr fehleranfällig, aufwendig und schlecht nachvollziehbar ist. Und jeder, der schon einmal für eine große Anzahl von Servern verantwortlich war, wird bestätigen, dass die rein manuelle Administration im Prinzip unmöglich ist.

Vielleicht haben Sie in einer solchen Situation schon mit Skripten gearbeitet (Shell, Python, Perl ...), oder Sie sind sogar bereits mit einem Alternativprodukt wie Puppet, Chef, SaltStack oder CFEngine vertraut. In jedem Fall freue ich mich, dass Sie sich nun mit Ansible beschäftigen möchten. Dieses Werkzeug bietet Ihnen u.a. folgende Möglichkeiten:

- Automatisierung der Provisionierung von Systemen
- Automatisierung des Software-Deployments
- Konfigurationsmanagement

Ansible ist in der Tat ein äußerst flexibles Werkzeug; ich erspare uns an dieser Stelle aber den viel zu oft strapazierten Vergleich mit Schweizer Schneidewerkzeugen. Sie können Ansible benutzen, um auf einer Maschine eine Software auszurollen, oder Sie können damit Konfigurationen auf Hunderten von Servern kontrollieren. Zudem ist es (relativ) leicht zu erlernen, und den meisten Anwendern macht es sogar Spaß!

Grundmerkmale

Ansible zeichnet sich durch folgende Eigenschaften aus:

- Ansible ist in Python programmiert. Ursprünglich wurde es in Python 2 entwickelt, ab Ansible Version 2.5 wurde aber auch Python 3 vollständig unterstützt.
- Ansible verwaltet seine Hosts in den allermeisten Fällen über SSH und benötigt auf diesen lediglich Python (zuzüglich einiger Python-Module in speziellen Fällen). Insbesondere ist Ansible *agentenlos*, da es keine speziellen Dienste auf den Zielsystemen benötigt.
- Ansible nutzt hauptsächlich YAML als Konfigurationssprache.
- Für Ansible stehen Tausende von Modulen für die verschiedensten Verwaltungsaufgaben zur Verfügung.
- Ansible bietet recht schnell Erfolgserlebnisse, skaliert aber auch gut in komplexen Szenarien.
- Ansible ist sehr ordentlich dokumentiert (siehe *https://docs.ansible.com*).

Kernkomponenten

In der Regel findet man in einem Ansible-Projekt folgende Komponenten:

1. Das *Inventory* ist ein Verzeichnis der Maschinen, die mit Ansible administriert werden sollen. Dieses muss in der Regel vom Anwender erstellt werden.
2. Die *Playbooks* sind vom Benutzer zu erstellende Prozeduren, die in einzelnen Schritten (*Tasks*) beschreiben, wie die jeweiligen Konfigurationsziele zu erreichen sind.
3. Die *Module* sind von Ansible zur Verfügung gestellte Funktionseinheiten, die die eigentliche Arbeit erledigen. Jeder Schritt in einem Playbook ist letztlich nichts anderes als der Aufruf eines Moduls.
4. *Rollen* dienen dazu, größere Projekte modular, wartbar und wiederverwendbar zu gestalten. Mindestens eines haben sie mit den Klassen aus der objektorientierten Programmierung gemeinsam: Ganz zu Anfang braucht man sie nicht, und später will man sie nicht mehr missen.

Was bedeutet »Ansible«?

Der Name »Ansible« kommt aus der Science-Fiction. Ein *Ansible* ist dort ein Gerät zur überlichtschnellen Kommunikation. Ursprünglich von Ursula K. Le Guin ersonnen, haben viele andere Autoren die Idee adaptiert. Ganz konkret hat der Ansible-Erfinder Michael DeHaan den Begriff aus dem Buch »Ender's Game« von Orson Scott Card entliehen. Dort wird das Ansible ebenfalls benutzt, um über sehr große Entfernungen mit Flotten oder Stützpunkten zu kommunizieren.

Deklarativ oder imperativ?

Konfigurationsmanagementwerkzeuge wie Ansible werden oft in die Kategorien *deklarativ* oder *imperativ* eingeteilt. Dies beschreibt dann den Charakter der im jeweiligen Produkt verwendeten DSL (Domain-Specific Language).

In deklarativen Sprachen beschreibt man den gewünschten Endzustand, wohingegen in imperativen Sprachen das *Wie*, also der Weg dorthin, beschrieben wird. Die Datenbanksprache SQL ist z. B. klar imperativ (`CREATE TABLE`, `INSERT INTO` …); ein schönes Beispiel für eine deklarative Sprache ist das XML-basierte Grafikformat SVG:

```
<svg xmlns="http://www.w3.org/2000/svg" height="480" width="680">
  <circle cx="340" cy="240" r="100" />
</svg>
```

Hier wird einfach ein Kreis »herbeigewünscht«, und irgendjemand muss sich darum kümmern, den Job zu erledigen.

In welche Kategorie fällt nun unser Ansible? Es ist nicht eindeutig zuzuordnen, wir haben es gewissermaßen mit einem Hybriden zu tun. Es finden sich sowohl eher deklarative Beispiele wie

```
- service:
    name: httpd
    state: started
```

als auch eher imperativ anmutende Aufrufe:

```
- copy:
    src: test.txt
    dest: /tmp
```

Letztlich ist es eine akademische Frage und oft auch ein wenig Auslegungssache. Das zweite Beispiel könnten Sie lesen als »Kopiere die Datei *test.txt* ins */tmp*-Verzeichnis« oder als »Ich wünsche mir eine Kopie der Datei *test.txt* im */tmp*-Verzeichnis«.

Falls Sie gelegentlich nichts Besseres zu tun haben, machen Sie einmal eine Google-Suche nach »Ansible deklarativ imperativ«. Da finden Sie ebenfalls Belege für alle möglichen Meinungen. Wir sollten die Fragestellung einfach abhaken und gegebenenfalls am DevOps-Stammtisch weiter darüber philosophieren.

1.2 Was ist Ansible nicht?

Nur sicherheitshalber, damit Sie nicht mit falschen Erwartungen an dieses Werkzeug herangehen, folgen nun auch einige Bemerkungen dazu, was Ansible nicht ist:

- Ansible ist kein Klickibunti-Werkzeug. Es besteht im Kern aus verschiedenen Kommandozeilentools nebst deren Input-Dateien, und das alles gilt es zu verstehen und zu beherrschen. Es gibt durchaus GUIs (grafische Anwendungen) für Ansible, aber ohne Verständnis für Ansibles interne Arbeitsweise können Sie mit denen relativ wenig anfangen.
- Ansible hat kein übermäßig hohes Abstraktionsniveau. Sie können Ansible nicht so etwas mitteilen wie: »Sorge für den Betrieb eines Webservers«. Vielmehr müssen Sie Schritt für Schritt erklären: »Installiere das Paket `httpd`, starte den Dienst `httpd` und integriere ihn in den Autostart, öffne via `firewalld` die Ports 80 und 443 etc.«

  Etwas salopp könnte man an dieser Stelle sagen: *Wenn Sie nicht wissen, wie es ohne Ansible geht, dann bringt Ihnen Ansible auch nichts.*
- Ansible ist kein reines Admin-Tool. Es ist ein Werkzeug aus der DevOps-Welt, und deswegen müssen Sie willens sein, sich auch mit Konzepten aus der Development-Welt auseinanderzusetzen – wie beispielsweise If-Abfragen, Schleifen, Fehlerbehandlung, Zugriff auf (komplexe) Datenstrukturen usw.
- Ungeachtet dessen ist Ansible aber *keine Programmiersprache* und soll auch per Designziel keine sein. (*Falls* Sie jedoch einige Programmierkenntnisse besitzen, wird es sicher kein Nachteil für Sie sein!)

1.3 Geschichte und Versionen

Im Jahr 2012 stellte Michael DeHaan die erste Version von Ansible vor. Das von ihm gegründete Unternehmen *Ansible Inc.* wurde Ende 2015 von Red Hat übernommen. Seitdem wird die Weiterentwicklung maßgeblich von Red Hat beeinflusst, was dem Produkt bislang sehr gut tut.

Versionen

Mit bzw. nach der Ansible-Version 2.10 wurden das Versionierungsschema und vor allem die interne Organisation der Weiterentwicklung stark verändert. Seitdem werden der Kern von Ansible und die Zusatzkomponenten wie Module und Plugins unabhängig voneinander entwickelt. Der Ansible-Kern wird mit dem alten Versionierungsschema weiterentwickelt und heißt nun *Ansible-Core* (Version ≥ 2.11; in Version 2.10 hieß er kurzzeitig *Ansible-Base*). Die in großer Zahl existierenden Zusatzkomponenten heißen jetzt *Collections* und folgen jeweils einem individuellen Versionierungsschema.

Für die meisten Endanwender am interessantesten dürfte nun das *Ansible Community Package* sein, beginnend mit Version 3.0. Dies ist eigentlich nur eine Metadistribution, die die Version des enthaltenen Ansible-Kerns und die Auswahl und Versionen

der mitgelieferten Collections festlegt. Die Versionsnummern folgen nun außerdem dem Prinzip der semantischen Versionierung (`Major.Minor.Patch`). In der Übersicht stellt sich das wie folgt dar:

- Ansible 2.8 oder älter: klassisches Komplettpaket (für uns: »Steinzeit«)
- Ansible 2.9: klassisches Komplettpaket, Collections als Tech Preview
- Ansible 2.10: Ende der Vorbereitungen, Teilung in Ansible-Base und Collections
- Ansible 3.0: Community Package, bestehend aus Ansible-Base 2.10 + Collections
- Ansible 4.0: Community Package, bestehend aus Ansible-Core 2.11 + Collections
- Ansible 5.0: Community Package, bestehend aus Ansible-Core 2.12 + Collections
- ...
- Ansible 9.0: Community Package, bestehend aus Ansible-Core 2.16 + Collections
- Ansible 10.0: Community Package, bestehend aus Ansible-Core 2.17 + Collections

Den genauen Stand der Dinge können Sie bei Bedarf nachlesen unter *https://docs.ansible.com/ansible/latest/reference_appendices/release_and_maintenance.html*.

Unterstützte Python-Versionen

Es gibt bei Ansible durchaus Anforderungen bzgl. der eingesetzten Python-Version, und dabei muss man auch noch Control Host und Target Host unterscheiden. Auf der Webseite gibt es dazu die Ansible-Core-Supportmatrix, in der alle kompatiblen Versionen klar und übersichtlich dargestellt sind. Einige Schlussfolgerungen daraus sind:

- Auf dem Control Host endet der Python-2-Support mit Ansible-Core Version 2.11.
- Für Target Hosts endet der Python-2-Support mit Ansible-Core Version 2.16.
- Wenn Sie einen Control Host mit einem Standard-Rocky-9-Linux betreiben, werden Sie dort maximal ein Ansible-Core 2.15 installieren können, da die mitgelieferte Python-Version 3.9 in dieser Konstellation der limitierende Faktor ist.
- Wenn Sie Target Hosts mit einem Standard-openSUSE-Leap betreiben, darf ihr Control Host maximal mit Ansible-Core Version 2.16 laufen, denn openSUSE Leap kommt standardmäßig mit Python 3.6. (Es gibt aber auch ein Paket *python311*, das Sie auf den SUSE-Targets bei Bedarf zusätzlich installieren könnten.)

Anmerkung

Machen Sie sich bitte zu Anfang keine Sorgen um Versionen. In allen aktuellen Linux-Enterprise-Distributionen sind die Versionen der Ansible-Community- und Python-Packages so gewählt, dass Sie damit nur höchst selten irgendwelche Kompatibilitätsprobleme bekommen. Sie können Ihre Aufmerksamkeit also beruhigt auf interessantere Dinge richten!

1.4 Setup/Laborumgebung

Sie haben dieses Buch erworben, um sich mit Ansible vertraut zu machen. Der bewährte Weg dazu ist Learning by Doing, d. h., Sie sollten die für Sie relevanten Beispiele und Vorgehensweisen in der Praxis nachvollziehen. Idealerweise steht Ihnen dazu eine Test- bzw. Laborumgebung zur Verfügung, deren Wunschausstattung ich nun beschreiben möchte.

Zunächst einmal benötigen Sie irgendeinen UNIX-artigen Host, auf dem die Ansible-Software installiert wird (den sogenannten *Control Host* oder auch die *Control Machine*). Jede gängige halbwegs aktuelle Linux-Distribution ist dazu geeignet, aber es ginge auch mit macOS, Solaris etc. Eine native Windows-Version von Ansible gibt es nicht; über Umwege (z. B. Cygwin) kann man es aber durchaus auch auf Windows zum Laufen bringen. Bitte sehen Sie jedoch an dieser Stelle unbedingt davon ab, da zu viele Tricks und Kniffe nötig wären.

Außerdem benötigen Sie noch mindestens einen zusätzlichen Host, der via Ansible administriert werden soll (einen sogenannten *Target Host* oder *Managed Node*). Am Anfang werden wir hier Linux-Systeme voraussetzen; sehr viel später werden Sie sehen, dass man z. B. auch Windows-Systeme, Netzwerk-Devices oder Cloud-Umgebungen mit Ansible administrieren kann. Die wichtigste Voraussetzung an dieser Stelle ist, dass der Control Host seine Target Hosts (im Fall von Linux) über SSH erreichen kann.

Nun kann ich natürlich nicht wissen, wie Ihre Arbeits- bzw. Testumgebung aussieht. Vielleicht ist Ihr Arbeitsplatzrechner seit Langem ein Linux-System, und Sie haben bereits letzte Woche drei vServer in der Cloud angemietet – womit Sie eigentlich direkt loslegen könnten. Möglicherweise ist aber auch nichts von alledem der Fall.

Die für dieses Buch vorgeschlagene hauptsächliche Testumgebung sehen Sie in Abbildung 1.1. Ich zeige Ihnen auch sogleich, wie Sie zu einer solchen Umgebung gelangen können, sofern Sie nicht in Eigenregie eine aufbauen möchten.

Die Software *Vagrant* eignet sich hervorragend, um eine virtuelle Testumgebung bereitzustellen. Das Produkt setzt lediglich eine übliche Virtualisierungslösung voraus (z. B. VirtualBox, VMware, HyperV) und kann dann mittels einer Beschreibungsdatei (dem *Vagrantfile*) eine definierte Menge von virtuellen Maschinen provisionieren. Ich gehe an dieser Stelle einmal von VirtualBox als Virtualisierer aus. Die benötigte Software bekommen Sie hier:

- Vagrant (*https://www.vagrantup.com*)
- VirtualBox (*https://www.virtualbox.org*)

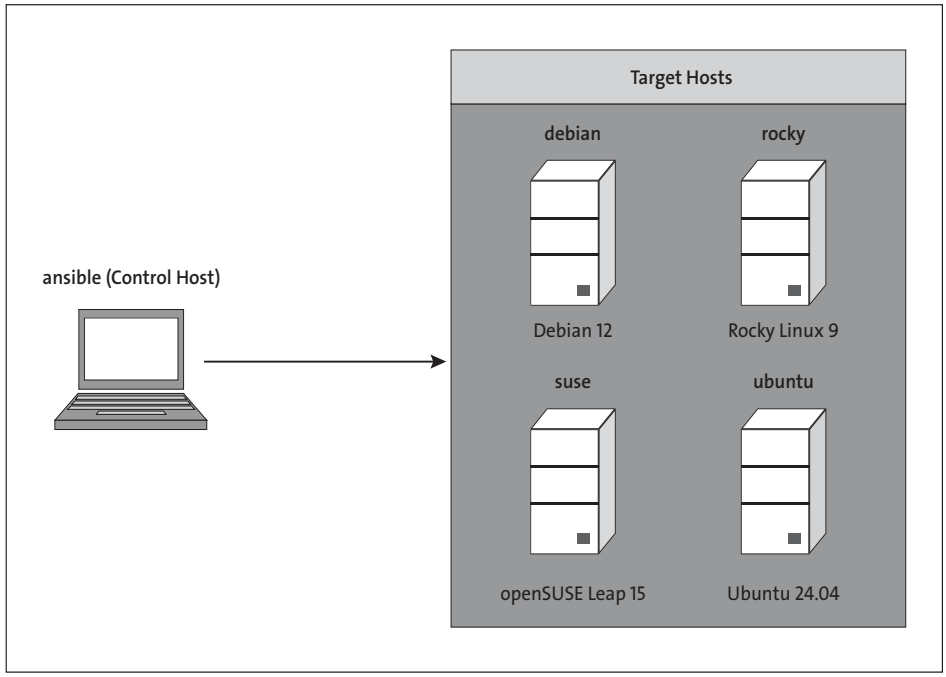

Abbildung 1.1 Unser Beispiel-Setup

Wichtig

Zum Zeitpunkt des Entstehens dieser Zeilen ist die VirtualBox-Version 7.1 verfügbar, die aber von der derzeit aktuellen Vagrant-Version noch nicht unterstützt wird. Wahrscheinlich sind diese Probleme bald behoben; aber wenn Sie hier nicht experimentierfreudig sind, rate ich einfach zur Verwendung der VirtualBox-Version 7.0.

Auf der VirtualBox-Download-Seite finden Sie rechts unten einen Link Previous Releases (*https://www.virtualbox.org/wiki/Download_Old_Builds*). Auf der darauffolgenden Seite wählen Sie dann einfach die 7.0er-Version und laden so das für Ihr System geeignete aktuellste 7.0er-Release herunter.

Wenn diese zwei Virtualisierungswerkzeuge nun auf Ihrem System zur Verfügung stehen, können wir uns der Erstellung der virtuellen Testsysteme zuwenden. Listing 1.1 zeigt ein *Vagrantfile*, das Sie dazu bitte einfach in irgendeinem neuen Verzeichnis ablegen:

```
#
# Please adjust if necessary:
#
```

```
NETWORK_PREFIX = "192.168.150"

Vagrant.configure("2") do |config|

  # Set default RAM size for all boxes:
  config.vm.provider "virtualbox" do |vb|
    vb.customize ["modifyvm", :id, "--memory", "1024"]
  end

  config.vm.define "ansible" do |ansible|
    ansible.vm.box = "bento/debian-12"
    ansible.vm.hostname = "ansible"
    ansible.vm.network :private_network, ip: "#{NETWORK_PREFIX}.100"
  end

  config.vm.define "debian" do |debian|
    debian.vm.box = "bento/debian-12"
    debian.vm.hostname = "debian"
    debian.vm.network :private_network, ip: "#{NETWORK_PREFIX}.10"
  end

  config.vm.define "rocky" do |rocky|
    rocky.vm.box = "bento/rockylinux-9"
    rocky.vm.hostname = "rocky"
    rocky.vm.network :private_network, ip: "#{NETWORK_PREFIX}.20"
  end

  config.vm.define "suse" do |suse|
    suse.vm.box = "bento/opensuse-leap-15"
    suse.vm.hostname = "suse"
    suse.vm.network :private_network, ip: "#{NETWORK_PREFIX}.30"
  end

  config.vm.define "ubuntu" do |ubuntu|
    ubuntu.vm.box = "bento/ubuntu-24.04"
    ubuntu.vm.hostname = "ubuntu"
    ubuntu.vm.network :private_network, ip: "#{NETWORK_PREFIX}.40"
  end

end
```

Listing 1.1 »Vagrantfile«: Provisionierung der Testumgebung

Diese Datei definiert die in Abbildung 1.1 gezeigten fünf Maschinen. Allerdings müssen Sie für Ihre Umgebung vorher noch etwas erledigen:

1. Legen Sie in der VirtualBox-Applikation ein neues *Host-only-Netzwerk* an.

   Das können Sie bequem mithilfe von Datei • Werkzeuge • Netzwerk-Manager erledigen. Dort klicken Sie auf Erzeugen, merken sich dann die IPv4-Adresse oder passen sie nach Ihren eigenen Vorstellungen an.

   Im Buch nutze ich exemplarisch:

   ```
   IPv4-Adresse:    192.168.150.1/24
   ```

   Wenn nichts dagegenspricht, wäre es durchaus sinnvoll, wenn Sie dieser Vorgabe folgen. Ansonsten müssen Sie später ggf. einige wenige Beispieldateien anpassen, was aber sicher auch kein Problem darstellt.

   Den DHCP-Server dieses neuen Netzwerks setzen Sie bitte auf **deaktiviert**. Wir brauchen ihn nicht und wollen auch nicht riskieren, dass er uns irgendwie »dazwischenfunkt«:

   ```
   DHCP-Server:    deaktiviert
   ```

   Falls Sie VirtualBox unter Linux betreiben: Per Default ist *keine* beliebige Wahl von Host-only-Netzwerkadressen möglich. Sie müssen das als Root zunächst erlauben, indem Sie eine Datei */etc/vbox/networks.conf* mit folgendem Inhalt erzeugen:

   ```
   * 0.0.0.0/0 ::/0
   ```

   Listing 1.2 »/etc/vbox/networks.conf«: beliebige Netzwerke unter Linux erlauben

   Gegebenenfalls müssen Sie danach die VirtualBox-App einmal schließen und wieder öffnen.

2. Gleichen Sie nun das `NETWORK_PREFIX` im *Vagrantfile* (falls nötig) mit Ihrer VirtualBox-Konfiguration ab.

Die virtuellen Maschinen werden mit 1 GByte RAM pro Stück provisioniert. In der Summe benötigt unser Setup also zunächst maximal 5 GByte RAM, real gemessen sicher aber etwas weniger. Wenn Ihr Hostsystem diese Größenordnung an Hauptspeicher nicht hergibt, müssten Sie den einen oder anderen Target Host einfach weglassen (was jedoch schade wäre, da spätere Beispiele mit all diesen Hosts arbeiten).

Nach dieser Vorbereitung können Sie das ganze Szenario nun online bringen. Wechseln Sie dazu in der Kommandozeilenumgebung in das Verzeichnis, in dem Sie das *Vagrantfile* abgelegt haben, und geben Sie dort Folgendes ein:

```
$ vagrant up
```

Nach einiger Wartezeit sollten alle Maschinen laufen und entsprechend ihren konfigurierten IP-Adressen an-**ping**-bar sein:

```
$ ping 192.168.150.100
$ ping 192.168.150.10
$ ping 192.168.150.20
[…]
```

Sie können sich nun beispielsweise am Control Host anmelden; falls Sie unter Windows arbeiten, können Sie gern eine gewohnte Software wie *PuTTY* dazu verwenden. Username und Passwort sind jeweils `vagrant`. Die *Vagrant*-Software bietet selbst auch einen bequemen Weg zur Anmeldung:

```
$ vagrant ssh ansible
vagrant@ansible:~$ exit
```

Anmerkung

Der `vagrant`-Useraccount hat per Default uneingeschränkte **sudo**-Rechte. Wenn Sie also Root-Rechte benötigen, setzen Sie einfach **sudo** vor Ihr Kommando, oder Sie nehmen sich bei Bedarf eine dauerhafte Root-Shell mittels **sudo -i**.

Und bevor wir das Thema Vagrant wieder verlassen – Tabelle 1.1 zeigt noch eine Übersicht der allerwichtigsten *Vagrant*-Kommandos:

Befehl	Wirkung
vagrant global-status	Übersicht: Was läuft und was läuft nicht?
vagrant box list	bereits heruntergeladene Images auflisten
vagrant box update	heruntergeladene Images aktualisieren
vagrant up [**<NAME>…**]	Maschine(n) hochfahren
vagrant halt [**<NAME>…**]	Maschine(n) anhalten
vagrant reload [**<NAME>…**]	Maschine(n) neu starten
vagrant destroy [**<NAME>…**]	Maschine(n) wegwerfen
vagrant snapshot push [**<NAME>…**]	Maschine(n) snapshotten
vagrant snapshot pop [**<NAME>…**]	zu diesem Snapshot zurückkehren
vagrant snapshot list	alle Snapshots auflisten

Tabelle 1.1 Die wichtigsten Vagrant-Kommandos

Maschinenbezogene Kommandos wie `up`, `halt` etc. müssen dabei stets im Verzeichnis oder unterhalb desjenigen Verzeichnisses aufgerufen werden, das das zugehörige Vagrantfile enthält!

1.5 Ansible-Installation auf dem Control Host

Ein Ansible-Community-Paket ist in den Hauptpaketquellen aller wichtigen Distributionen vorhanden, nur unter Rocky/RHEL ist die Situation etwas anders. Dort bedient man sich dann in der Regel im EPEL (siehe unten). Tabelle 1.2 zeigt den Stand der Dinge im Herbst 2024.

Distribution	paketierte Version
Debian 12	7.7.0
Debian 11	2.10.8
Rocky Linux 9	*ansible-core* 2.14.14
Rocky Linux 8	–
openSUSE Leap 15.6	9.4.0
openSUSE Leap 15.5	2.9.27
Ubuntu 24.04 LTS	9.2.0
Ubuntu 22.04 LTS	2.10.8

Tabelle 1.2 Ansible-Versionen in verschiedenen Distributionen

Anmerkung

Der Deutlichkeit halber möchte ich hier noch mal klarstellen: Da Ansible *agentenlos* ist, ist die Installation der Ansible-Software *nur* auf dem Control Host nötig – im Fall unserer Laborumgebung ist das ein Debian-12-System.

Auf den Zielsystemen benötigen Sie momentan nichts (außer SSH und Python).

Die Installation aus Distributionspaketen ist im Allgemeinen recht schnell erledigt:

- **Debian 12** (in der Laborumgebung unser Control Host), **Debian 11**

```
# apt update
# apt install ansible
```

- **Rocky Linux 9**

  Wie in Tabelle 1.2 ersichtlich, ist im Standardlieferumfang von Rocky Linux 9 nur das *ansible-core*-Paket vorhanden. Sobald Sie Module bzw. Collections benötigen, die nicht in diesem Basispaket enthalten sind, müssten Sie diese direkt aus dem Internet nachinstallieren (siehe Abschnitt 7.1). In vielen Umgebungen wird die Verwendung des *ansible*-Packages aus dem EPEL-Zusatzrepository wohl einfacher sein, weswegen ich hier erst einmal diesen Weg empfehle:

  ```
  Zunächst aktivieren Sie das EPEL-Zusatzrepository:
  # dnf install epel-release
  # dnf install ansible
  ```

  Unter Rocky Linux 8 verfahren Sie der Einfachheit halber ganz genauso.

- **openSUSE Leap 15.6**

  ```
  # zypper install ansible
  ```

  Wenn Sie noch eine ältere openSUSE-Version betreiben, ist die paketierte Ansible-Version möglicherweise zu alt. Sie müssten im Bedarfsfall also auf eine Installation mit PIP (siehe unten) ausweichen.

- **Ubuntu 24.04 LTS**, **Ubuntu 22.04 LTS**

  ```
  $ sudo apt update
  $ sudo apt install ansible
  ```

Wenn die Installation erfolgreich beendet wurde, machen Sie doch gleich einen Testaufruf:

```
$ ansible --version
ansible [core 2.14.16]
  config file = None
  […]
```

Lassen Sie uns auf dem Control Host auch noch */etc/hosts*-Einträge für alle beteiligten Hosts vornehmen, denn auch im Testlabor wollen wir das Arbeiten mit IP-Adressen möglichst vermeiden. Wenn Ihnen in der Realität eine DNS-basierte Namensauflösung zur Verfügung steht, können Sie sich diesen Schritt natürlich sparen:

```
#
# Please adjust according to your environment:
#
192.168.150.10  debian.example.org  debian
192.168.150.20  rocky.example.org   rocky
192.168.150.30  suse.example.org    suse
192.168.150.40  ubuntu.example.org  ubuntu
```

Listing 1.3 »/etc/hosts«: exemplarischer Ausschnitt

1.6 Installation via PIP (+ Virtualenv)

Anmerkung

Wenn das Ansible-Paket Ihrer Distribution bzw. Ihres Zusatzrepositorys hinreichend aktuell ist und Sie sich vorerst auch nicht für das Virtualenv-Feature von Python interessieren, können Sie diesen Abschnitt gern überspringen und gleich in Abschnitt 1.7 weiterlesen.

Als zweite Installationsvariante gibt es den Weg über das Python-Paketmanagement. Dies würden Sie z. B. in den folgenden Fällen in Betracht ziehen:

- Ihre Distribution stellt nur veraltete bzw. gar keine geeigneten Pakete zur Verfügung.
- Eventuelle Zusatzrepositorys sind für Sie keine Option.
- Sie möchten gerne verschiedene Ansible-Versionen parallel betreiben.

Zunächst müssen Sie als Root für die nötigen Voraussetzungen sorgen. Sie benötigen zwei Distributionspakete; diese ziehen dann automatisch alle weiteren Abhängigkeiten nach:

```
$ sudo apt install python3-pip python3-venv
```

Unter SUSE und Rocky bräuchten Sie nur *python3-pip*, denn *venv* ist in Python 3 ein Standardmodul. Eine normale Debian-Python-Installation kommt jedoch etwas schlanker daher; das Paket *python3-full* wäre hier die Rundum-sorglos-Lösung.

Der schnellste Weg zur Ansible-Installation wäre nun die direkte Installation über das Python-Paketmanagement, wofür Sie auch nur das *python3-pip*-Paket gebraucht hätten:

```
Als Root, wenn Sie das wirklich wollen:
sudo pip3 install ansible
```

Empfohlen wird das aber nicht mehr, und ein aktuelles System der Debian-Familie müsste sogar mit der Option `--break-system-packages` überredet werden.

Eine sehr gute Alternative ist die Installation in einer Virtualenv-Umgebung. Diese Methode hat den Vorteil, dass Sie problemlos als unprivilegierter Benutzer arbeiten können und sogar beliebig viele Versionen parallel betreiben könnten. Und Sie vermeiden damit das potenzielle Problem, dass PIP irgendwelche Änderungen am System durchführt, die nur schwer wieder rückgängig zu machen sind.

Geben Sie die folgenden Befehle also gerne als unprivilegierter User ein:

```
Eine virtuelle Umgebung einrichten (das Verzeichnis ist frei wählbar):
$ python3 -m venv ~/venv/ansible

Betreten der Umgebung:
$ source ~/venv/ansible/bin/activate

Bei der *Erstnutzung* der Umgebung ggf. PIP aktualisieren.
Zumindest, falls ansonsten in der Folge ernste Probleme auftreten:
$ pip3 install --upgrade pip


Ansible-Installation (ohne "==VERSION" erhalten Sie die aktuellste mit Ihrem
Python mögliche Version):
$ pip3 install ansible==9.2.0
[…]

$ ansible --version
ansible [core 2.16.8]
[…]

Verlassen der Umgebung:
$ deactivate
```

Wenn Sie möchten, könnten Sie nun beliebig viele solcher Virtualenv-Umgebungen mit verschiedenen Ansible-Versionen oder anderer beliebiger Python-Software verwalten.

1.7 Authentifizierung und Autorisierung auf den Target Hosts

Kommen wir nun zu den Target Hosts. Wir wollen auf ihnen mit Ansible administrative Tätigkeiten verrichten, also müssen pro Zielhost einige Fragen geklärt werden (wobei wir hier bis auf Weiteres von Linux-Hosts sprechen):

1. Welcher Benutzeraccount kann von außen via SSH erreicht werden, und welche Authentifizierungsmethode soll dabei verwendet werden?

   Am gängigsten ist hier die Public-Key-Authentifizierung, aber mitunter stehen auch nur klassische Passwörter zur Verfügung.

   Seltener anzutreffen, aber nicht unüblich ist noch die SSH-Kerberos-Authentifizierung. Diese einzurichten, ist ein völlig eigenständiges und hochkomplexes Thema, das wir hier nicht behandeln können. Wenn Sie aber eine fertig eingerichtete

Umgebung vor sich haben, ist aus Sicht der Ansible-Kommandozeile keine weitere Konfiguration erforderlich – ein gültiges Kerberos-Ticket kann von Ansible genauso verwendet werden wie von einem menschlichen User.

2. Wenn Ansible sich erfolgreich authentifiziert hat: Stehen dann bereits Root-Rechte zur Verfügung, oder muss noch eine Identitätsänderung bzw. Rechteerhöhung durchgeführt werden?

   Wenn ja, mit welcher Methode (`su`/`sudo`)?

Der aktuelle Ist-Zustand unserer Laborumgebung ist hier sehr einheitlich: Wir erreichen alle Systeme über den Account `vagrant` mit dem Passwort `vagrant`. Die Rechteerhöhung wird mit `sudo` durchgeführt.

Ein erster Test mit UNIX-Passwort-Authentifizierung

> **Anmerkung**
>
> Wir werden im Labor durchgängig mit SSH-Schlüsseln arbeiten. Wenn das auch Ihre bevorzugte Authentifizierungsmethode ist bzw. Passwörter in Ihrem Umfeld keine Rolle spielen, können Sie gerne sofort im nächsten Abschnitt weiterlesen.

Sie wissen wahrscheinlich, dass eine Automatisierung über SSH mit Passwörtern eher schwierig ist. Nehmen wir für den Moment einfach mal an, dass wir nichts an der Authentifizierungsmethode ändern wollen. Dann müssen Sie auf dem Control Host das Paket *sshpass* installieren:

```
Nur dann nötig, wenn sich Ansible mit Passwörtern authentifizieren soll:
$ sudo apt install sshpass
```

Wenn Sie das Paket installiert haben, können Sie Ihre Zielhosts im Hinblick auf Ansible bereits testen. Voraussetzung dafür ist allerdings, dass unser Control Host die Hostschlüssel aller Target Hosts kennt. Das geht im Labor sehr einfach mit dem **`ssh-keyscan`**-Kommando:

```
$ mkdir -p ~/.ssh && chmod 700 ~/.ssh
$ ssh-keyscan -t rsa debian rocky suse ubuntu > ~/.ssh/known_hosts
```

Im Folgenden sehen Sie zwei Testaufrufe, bei denen Sie jeweils aufgefordert werden, das Login-Passwort einzugeben. Unser Ziel ist es, jeweils als Antwort auf das Ansible-`ping` ein grünes `pong` zu sehen. Dadurch ist dann u.a. sichergestellt, dass auf der Gegenseite ein Python-Interpreter vorhanden ist. Alle Warnmeldungen können Sie vorerst ignorieren:

```
$ ansible all -i debian,rocky,suse,ubuntu -u vagrant -m ping -k
[…]
```

```
Ein einzelnes Zielsystem: Beachten Sie das Komma!
$ ansible all -i debian, -u vagrant -m ping -k
[…]
```

Die Semantik der `-i`-Option werden wir später noch klären (beachten Sie jedoch das Komma hinter einem einzelnen Hostnamen). Die Option `-u` bestimmt den Login-User, `-k` fragt nach dem Login-Passwort. Mit der zusätzlichen Option `-b` können Sie nach dem Login eine Identitätsänderung zum Root-User veranlassen. Die Default-Methode dafür ist `sudo`, es sei denn, Sie wählen mit `--become-method` eine andere. Wenn dazu eine weitere Passworteingabe erforderlich ist, müssen Sie noch die zusätzliche Option `-K` angeben.

Sehen Sie hier einige Aufrufe, die für *andere* Umgebungen geeignet sein könnten:

```
$ ansible all -i somecentos, -m ping -u ansible -k -b --become-method su -K

$ ansible all -i someubuntu, -m ping -u ansible -k -b -K
```

1.8 Einrichten der SSH-Public-Key-Authentifizierung

Für den Betrieb von Ansible richten Sie typischerweise (und bitte nun auch in unserem Labor) eine SSH-Public-Key-Authentifizierung vom Control Host zu den Target Hosts ein. Wir nutzen als Ausgangsaccount auf unserem Control Host den eingerichteten User `vagrant`. (Dauerhaftes Arbeiten als Root ist auch hier weder nötig noch sinnvoll.)

Ich gehe davon aus, dass Sie wissen, was zur Einrichtung der Public-Key-Authentifizierung zu tun ist. Falls nicht, folgt hier eine Quick-and-dirty(!)-Vorgehensweise für die Testumgebung:

```
vagrant@ansible:~$ ssh-keygen
(… alle Defaults akzeptieren …)

Verteilen des Schlüssels:
vagrant@ansible:~$ ssh-copy-id vagrant@debian
vagrant@ansible:~$ ssh-copy-id vagrant@rocky
vagrant@ansible:~$ ssh-copy-id vagrant@suse
vagrant@ansible:~$ ssh-copy-id vagrant@ubuntu
```

Falls Sie für Ihre Umgebung etwas mehr Hintergrundwissen zum Thema SSH benötigen, könnten Sie z. B. in Anhang C hineinlesen. Dort finden Sie weitergehende Informationen zu sicheren privaten Schlüsseln, SSH-Agenten und vieles anderes mehr.

Anmerkung

Sie können natürlich auch gern einen Schlüssel mit einem vom Default abweichenden Dateinamen generieren bzw. verwenden (z. B. *~/.ssh/ansible_key*). Bei **ssh-copy-id** müssen Sie diesen dann mittels `-i` angeben.

1.9 Ein Ad-hoc-Test ohne jegliche Konfiguration

Mit einem Ansible-`ping`-Aufruf können (und sollten!) Sie nun testen, ob die Ansible-Arbeitsumgebung so weit korrekt eingerichtet ist. Bevor es losgeht, möchte ich noch ein paar Anmerkungen loswerden:

- Wenn der Dateiname Ihres privaten SSH-Schlüssels vom Default abweicht, ergänzen Sie bitte jeweils `--private-key <PFAD>`.
- Die Angabe von `-u <USERNAME>` ist eigentlich überflüssig, da der Zielaccount genau wie der Ausgangsaccount lautet.

```
$ ansible all -i debian,rocky,suse,ubuntu -u vagrant -m ping
debian | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
rocky | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
[WARNING]: Platform linux on host suse is using the discovered
Python interpreter at /usr/bin/python3.6, but future installation
of another Python interpreter could change the meaning of that path.
See https://docs.ansible.com/[…] for more information.
suse | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3.6"
    },
    "changed": false,
    "ping": "pong"
```

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

Trotz eventueller Warn- bzw. Hinweismeldungen sollten Sie hoffentlich feststellen können, dass alle Zielsysteme mit einem freundlichen `pong` antworten. Ab Version 2.8 ist Ansible wesentlich smarter geworden, was die Erkennung des Python-Interpreters auf der Gegenseite betrifft. Es lässt nun aber auch alle Welt mit ausgiebigen Warn- und Hinweismeldungen an seiner Klugheit teilhaben; wir werden das in Kürze per Konfiguration etwas ruhiger gestalten.

Sollten Sie an dieser Stelle ernsthafte Fehlermeldungen sehen, so haben diese in aller Regel mit einem tatsächlich nicht vorhandenen Python-Interpreter zu tun. Oder Ihr Ansible ist zu alt und erkennt die gegebenenfalls vorhandene Python-Version nicht richtig. Statten Sie im ersten Fall Ihre Zielsysteme mit einer Python-Installation aus, und im zweiten Fall rate ich Ihnen dazu, ein Ansible in einer hinreichend aktuellen Version zu betreiben.

1.10 Noch ein Hinweis zur Migration von älteren Versionen

Ansible hat durchaus schon eine bewegte Entwicklungsgeschichte hinter sich. Besonders (aber nicht nur) beim Sprung auf Version 2.0 gab es viele inkompatible Änderungen. Ab Version 2.4 wurde es dann deutlich ruhiger.

Wenn Sie ältere Ansible-Projekte auf eine neuere Version migrieren wollen und dabei auf Schwierigkeiten stoßen, könnten Sie gegebenenfalls einen Blick in die Porting Guides werfen: *https://docs.ansible.com/ansible/devel/porting_guides/porting_guides.html*.

Da dort aber immer nur die Änderungen zwischen aufeinanderfolgenden Versionen beschrieben werden, wird Ihnen das im Zweifelsfall nicht viel helfen, wenn Ihr Playbook aus der 2.4er-Zeit nicht mehr unter Ansible 2.17 läuft. Die häufigste Ursache solcher Probleme sind aber in aller Regel Imkompatibilitäten zwischen älteren und neueren Versionen einzelner Module – genaues Begutachten der Warn- oder Fehlermeldungen + Suchmaschine + Konsultation der aktuellen Modul-Hilfsseite führt daher sicherlich (etwas Erfahrung vorausgesetzt) schneller zum Ziel.

Kapitel 2
Basiseinrichtung und erstes Inventory-Management

In diesem Abschnitt finden Sie einen Vorschlag dazu, wie Sie Ihre Arbeit mit Ansible auf Verzeichnisebene strukturieren können. Außerdem beschäftigen wir uns mit dem *Inventory* (sozusagen dem Verzeichnis der Target Hosts), das Sie für nahezu jedes Betriebsszenario von Ansible benötigen.

2.1 Verzeichnisstruktur einrichten

Ansible macht uns im Prinzip kaum feste Vorgaben, wenn es darum geht, welche Strukturen wir auf Verzeichnis- und Dateiebene anzulegen haben. Ich erachte in aller Regel die folgenden zwei Vorgaben als sinnvoll:

- Ein Ansible-Projekt sollte sich innerhalb bzw. unterhalb eines einzigen Ordners befinden. Vorteil: Diesen Ordner können Sie jederzeit einfach sichern oder in ein Versionskontrollsystem einchecken, ohne dass Sie befürchten müssen, etwas Wichtiges vergessen zu haben.
- Wir wollen auf dem Control Host als unprivilegierter User arbeiten (wie schon beim Einrichten der SSH-Public-Key-Authentifizierung praktiziert).

Wegen des zweiten Punktes scheidet der Ordner */etc/ansible*, der zumindest früher bei der Installation aus Distributionspaketen standardmäßig angelegt wurde, schon einmal aus. Ignorieren Sie ihn einfach, falls er auf Ihrem System noch vorhanden sein sollte.

Jedes typische Ansible-Projekt braucht eine Konfiguration, ein Inventory und ein oder mehrere Playbooks. Starten wir mal ganz neutral mit einem Projektverzeichnis namens *~/ansible/projects/start*, das dann in Kürze all diese Dinge enthalten wird:

```
$ mkdir -p ~/ansible/projects/start && cd $_
```

Die Bash-Spezialvariable `$_` nutze ich gerne und oft: Sie enthält das letzte Argument des letzten Kommandos.

Braucht man aber gleich schon eine solch tiefe Verzeichnishierarchie? Am Anfang im Prinzip nicht, ein Verzeichnis *hello* oder *ansible-test* würde es genauso tun. Mit der

Kapitel 5
Playbooks und Tasks: die Grundlagen

Playbooks sind der Einstieg in jedes Ansible-Projekt. Im einfachsten Fall enthalten sie eine geordnete Menge von einzelnen Schritten (*Tasks*), die beschreiben, wie die gewünschten Konfigurationsziele für die Target Hosts zu erreichen sind.

5.1 Hallo Ansible – das allererste Playbook

Obwohl Ansible keine Programmiersprache im engeren Sinne ist, wollen wir doch der guten alten Tradition folgen und als erste Amtshandlung einmal ein »Hallo Welt«-Playbook erstellen. Playbooks werden in YAML verfasst und tragen per Konvention die Dateiendung `.yml` oder `.yaml`. Das Playbook aus Listing 5.1 ist so ziemlich das einfachste, das überhaupt möglich ist:

```
---
- hosts: localhost

  tasks:
    - debug: msg="Hello Ansible!"
```

Listing 5.1 »hello-ansible.yml«: ein allererstes Playbook

Achtung

Wenn Sie das YAML-Format nicht hinreichend gut kennen, sollten Sie *wirklich* erst Kapitel 4 lesen.

Das Wesentliche bei YAML ist die korrekte Einrückung der Textbausteine – diese darf auf keinen Fall einfach irgendwie geschehen!

Damit unser Projektordner über kurz oder lang nicht zu unübersichtlich wird, schlage ich vor, für Playbooks einen Ordner *playbooks/* anzulegen:

```
$ cd ~/ansible/projects/start
$ mkdir playbooks
```

Speichern Sie die Playbook-Datei in diesen neuen Ordner.

Bevor wir das Ganze starten, wollen wir zunächst den Inhalt klären:

- Die drei Minuszeichen sind bekanntlich eine YAML-Markierung, die den Beginn des Dokuments kennzeichnen. Viele Ansible-Anwender lassen sie weg, was kein Problem ist, da mehrere YAML-Dokumente in einer Datei unter Ansible sowieso nicht möglich sind.

  Nur der Vollständigkeit halber sei erwähnt: Ansible nutzt als Parser das Python-PyYAML-Modul, das zurzeit die YAML-Version 1.1 implementiert.
- Das Schlüsselwort `hosts` gibt an, welche Target Hosts Sie ansprechen wollen. Anstelle von `localhost` wäre auch ein beliebiges Pattern (vergleiche Abschnitt 3.6) möglich. Sehr oft sieht man hier den Gruppennamen `all`, um alle Inventory-Hosts anzusprechen.

  Auch eine Liste von Patterns wäre möglich, z. B.:

  ```
  - hosts:
      - debian
      - rocky
  ```

  Oder in der kompakten YAML-Listenform:

  ```
  - hosts: [debian, rocky]
  ```

  Oder eben sogar als Ansible-Pattern:

  ```
  - hosts: debian, rocky
  ```
- Das Schlüsselwort `tasks` zeigt an, dass nun die Arbeitsschritte folgen, die die Target Hosts in den gewünschten Zielzustand überführen sollen.
- Der Aufruf des `debug`-Moduls ist momentan unser einziger Task; mit dem Parameter `msg` kann man die gewünschte Ausgabe bestimmen.

Auf Anhieb ist wahrscheinlich nicht völlig klar, warum das Playbook mit einer Liste beginnt. (YAML-technisch gesehen, ist der Inhalt unseres Playbooks eine Liste mit einem einzigen Element!) Die Erklärung ist folgende: Ein Playbook kann sich aus mehreren sogenannten *Plays* zusammensetzen – diese bestehen jeweils aus einer Menge von Hosts, denen gewisse Tasks zugeordnet sind. Playbooks mit mehr als einem Play werden Sie aber erst sehr viel später benötigen.

Um das Ganze (endlich) zu starten, verwenden Sie das Kommando **`ansible-playbook`** und übergeben den Pfad zum Playbook als Parameter. Wenn Ihr aktuelles Arbeitsverzeichnis also der Wurzelordner unseres Projekts ist, dann sieht das so aus:

```
$ ansible-playbook playbooks/hello-ansible.yml

PLAY [localhost] ***********************************************************
```

```
TASK [Gathering Facts] *********************************************************
ok: [localhost]

TASK [debug] *******************************************************************
ok: [localhost] => {
    "msg": "Hello Ansible!"
}

PLAY RECAP *********************************************************************
localhost                  : ok=2    changed=0    unreachable=0    failed=0
                             skipped=0    rescued=0    ignored=0
```

Auch die Ausgabe hat etwas Klärungsbedarf:

- Die Ausgabe wird zunächst mit `PLAY` und der Auflistung der entsprechenden Zielsysteme (in unserem Beispiel lediglich `localhost`) eingeleitet. Sie können ein Play auch mit einem `name:`-Attribut versehen, dann würde an dieser Stelle der Wert dieses Attributs ausgegeben. Wenn das Playbook nur ein einziges Play enthält, macht man das in der Regel aber nicht.
- Es folgen nun alle Tasks mit einer kurzen Beschreibung und einem entsprechenden Ergebnis.
- Den Task `Gathering Facts` haben wir gar nicht bestellt, aber es ist das Default-Verhalten von Ansible, zu Beginn eines Plays erst einmal Informationen über die beteiligten Zielsysteme einzusammeln. Ich komme darauf in Abschnitt 6.3 noch ausführlich zu sprechen. In den meisten realen Playbooks braucht man diese Fakten auch; in unserem einfachen Beispiel könnte man durch die Angabe von `gather_facts: no` oder `gather_facts: false` auch auf das Einsammeln verzichten und so etwas Zeit sparen:

  ```
  - hosts: localhost
    gather_facts: no
  ```

- Die letzte Zeile liefert schließlich einen Abschlussbericht: Auf dem Zielsystem `localhost` wurden zwei Tasks mit dem gewünschten (erfolgreichen) Ergebnis beendet (`ok=2`), es gab keine Änderungen (`changed=0`), das Zielsystem war erreichbar (`unreachable=0`), und es schlugen keine Tasks fehl (`failed=0`). Die weiteren Felder `skipped`, `rescued` und `ignored` können Sie zunächst getrost ignorieren.

Relative Pfade bei Playbook-Aufrufen

Mit unserer gewählten Verzeichnisstruktur müssen Sie im Allgemeinen mit der kleinen Unbequemlichkeit leben, alle Aufrufe stets aus dem Wurzelordner des Projekts tätigen zu müssen:

```
$ ansible-playbook playbooks/hello-ansible.yml
```

Es sei denn, Sie verwenden (wie in Abschnitt 2.2 empfohlen) die Software *direnv* oder eine vergleichbare schlaue Methode, die Ihnen mehr Bewegungsspielraum gibt. Ich werde jedenfalls ab jetzt in aller Regel in den Aufrufbeispielen auf die Angabe des Pfadbestandteils *playbooks/* verzichten, um uns diese Redundanz zu ersparen:

```
$ ansible-playbook hello-ansible.yml
```

Ansprechen der eigentlichen Target Hosts

Sicher ist `localhost` ein nicht so häufiges Ziel für Verwaltungsaufgaben. Lassen Sie uns stattdessen einmal alle wirklichen Ziele im Testlabor ansprechen:

```
---
- hosts: all

  tasks:
    - debug: msg="Hello Ansible!"
```

Listing 5.2 »hello-all.yml«: Hallo an alle

Und um das Ganze etwas spannender zu machen, fahren Sie doch eine Maschine (z. B. `suse`) vorher mal herunter (**`vagrant halt suse`** im *vagrant/*-Ordner). Das Ergebnis sieht dann in etwa so aus:

```
$ ansible-playbook hello-all.yml

PLAY [all] *********************************************************************

TASK [Gathering Facts] *********************************************************
ok: [rocky]
ok: [debian]
ok: [ubuntu]
fatal: [suse]: UNREACHABLE! => {"changed": false, "msg":
"Failed to connect to the host via ssh: ssh: connect to host suse port 22:
Connection timed out", "unreachable": true}

TASK [debug] *******************************************************************
ok: [debian] => {
    "msg": "Hello Ansible!"
}
ok: [rocky] => {
    "msg": "Hello Ansible!"
}
```

```
ok: [ubuntu] => {
    "msg": "Hello Ansible!"
}

PLAY RECAP *********************************************************************
debian                     : ok=2    changed=0    unreachable=0    failed=0
rocky                      : ok=2    changed=0    unreachable=0    failed=0
suse                       : ok=0    changed=0    unreachable=1    failed=0
ubuntu                     : ok=2    changed=0    unreachable=0    failed=0
```

Hier fällt zunächst auf, dass es pro Task keine fixe Reihenfolge der Zielsysteme zu geben scheint. Durch die parallele Abarbeitung ist das auch tatsächlich ziemlich zufällig; wir hatten dies in Abschnitt 3.4.1 bereits besprochen.

Außerdem ist zu bemerken, dass `suse` komplett aus dem Spiel ausgeschieden ist, nachdem seine Unerreichbarkeit festgestellt wurde, denn der zweite Task hat erst gar nicht mehr versucht, diesen Host anzusprechen.

Schließlich sollte die abschließende Statistik auch hier keine Rätsel aufgeben. (Ich habe obige Ausgabe aus Platzgründen etwas gekürzt.) Wenn Sie das Beispiel im Labor so nachvollzogen haben, denken Sie daran, die ausgeschaltete Maschine wieder mitspielen zu lassen (**`vagrant up suse`**).

5.2 Formulierung von Tasks

Die allermeisten Tasks bestehen aus dem Aufruf eines Ansible-Moduls, gegebenenfalls mit geeigneter Parametrisierung.

Wichtig

Der Deutlichkeit halber möchte ich dies noch einmal klar festhalten: Sie können pro Task nur *ein* Modul aufrufen! Wenn Ihre Aufgabe mit zwei (oder mehr) Modulaufrufen erledigt werden muss, benötigen Sie zwei (oder mehr) Tasks.

Davon abgesehen, gestattet Ansible bei der Formulierung von Tasks aber durchaus gewisse Freiheiten – umso mehr sollten Sie sich im Team möglichst früh auf einen Style Guide bzw. einige Best Practices einigen.

Das Wichtigste ist, dass jeder Task ein `name`-Attribut bekommt. Bei einem `debug`-Task muss man das sicher nicht ganz so eng sehen, aber für die meisten anderen Arten von Tasks sollten Sie es sich zur Regel machen. Eine Leerzeile zwischen den einzelnen Tasks schadet sicher auch nicht. Sehen Sie hier ein Beispiel, wie man es *besser nicht* macht:

```
tasks:
  - debug: msg="Hello"
  - command: df -h /
  - service: name=sshd state=started
```

Listing 5.3 Drei Tasks in unschönem Stil

Viel besser wäre es so: Die Task-Namen erscheinen beim Playbook-Lauf auch in der Ausgabe, was Übersicht und Orientierung immens erleichtert.

```
tasks:
  - name: Some test output
    debug: msg="Hallo"

  - name: Determine root file system space usage
    command: df -h /

  - name: Start SSH service
    service: name=sshd state=started
```

Listing 5.4 Drei Tasks; bereits viel lesbarer

Anmerkung

Diese Tasks sind als reine Beispiele für guten und weniger guten Stil gedacht. Was `command`- oder `service`-Aufrufe wirklich tun, werden wir in Kürze besprechen.

Der letzte der drei Tasks ist ein typisches Beispiel für einen parametrisierten Modulaufruf, bei dem die Parameter in der Form **`<name1=value1> <name2=value2> ...`** spezifiziert werden können.

Um lange (oder gar überlange) Zeilen zu vermeiden, kommt hier manchmal ein YAML-Block-Ausdruck zum Einsatz:

```
  - name: Start SSH service
    service: >
      name=sshd
      state=started
```

Alternativ können und sollten Sie die Parameter eher als YAML-Map spezifizieren. Achten Sie dabei penibel auf die Einrückebenen:

```
  - name: Start SSH service
    service:
      name: sshd
      state: started
```

Tipp

Die Formulierung von Modulparametern als YAML-Map sollte die Methode der Wahl sein – zumindest sobald Sie zwei oder mehr Parameter übergeben möchten!

Maps haben keine Reihenfolge

Auch darauf sei an dieser Stelle noch einmal hingewiesen: Eine Map als Datenstruktur ist eine *ungeordnete* Menge von Schlüssel-Wert-Paaren. Deswegen ist es aus technischer Sicht auch völlig egal, in welcher Reihenfolge Sie diese hinschreiben. Hier sehen Sie drei Beispiele, die intern völlig identisch sind:

```
- name: Start SSH service
  service:
    name: sshd
    state: started

- name: Start SSH service
  service:
    state: started
    name: sshd

- service:
    state: started
    name: sshd
  name: Start SSH service
```

Sehr wohl ist das Ganze aber wieder eine Frage des Geschmacks – und im Team sehr schnell auch von Style Guides. Ein `name`-Attribut nach unten zu setzen, ist sicher ziemlich ungewöhnlich, und Sie würden sich damit recht schnell unbeliebt machen.

5.3 Beenden von Plays

Wollen Sie (meist zu Testzwecken) ein Play vorzeitig beenden, so steht Ihnen die Anweisung

```
- meta: end_play
```

zur Verfügung, die Sie auf derselben Einrückebene wie einen normalen Task platzieren. Das Play wird dann an dieser Stelle beendet.

5.4 Der problematische Doppelpunkt

Früher oder später möchten Sie irgendwo einen Doppelpunkt verwenden, z. B. in einer Debug-Message oder wie im Beispiel aus Listing 5.5 in einem Task-Namen:

```
---
- hosts: localhost

  tasks:
    - name: Tell me: What's the problem here?
      debug: msg="This won't work"
```

Listing 5.5 »error-colon.yml«: ein fehlerhaftes Playbook

Da ein Doppelpunkt, auf den ein Leerzeichen folgt, in YAML eine besondere Bedeutung hat, werden Sie sich hier einen Syntax-Error einhandeln:

```
ERROR! Syntax Error while loading YAML.
[...]
The offending line appears to be:

  tasks:
    - name: Tell me: What's the problem here?
                   ^ here
```

Wenn Sie nicht auf Doppelpunkte verzichten können oder wollen, schafft oft eine einfache Quotierung Abhilfe (`"..."` oder `'...'`). Sollte jedoch auch der `msg`-Parameter des `debug`-Moduls einen Doppelpunkt enthalten, müssen Sie aus internen Gründen sogar doppelt quotieren:

```
---
- hosts: localhost

  tasks:
    - name: "Tell me: What's the problem here?"
      debug: 'msg="Now it works: Yay!"'
```

Listing 5.6 »colon.yml«: alle Doppelpunkte korrekt eingesetzt

Natürlich könnten die Rollen der doppelten und einfachen Hochkommas dabei auch vertauscht werden. Einfacher wäre es in diesem Fall jedoch gewesen, die YAML-Map-Parametrisierung zu wählen, denn dann würde eine simple Quotierung ausreichen:

```
---
- hosts: localhost

  tasks:
    - name: "Tell me: What's the problem here?"
      debug:
        msg: "Now it works: Yay!"
```

Listing 5.7 »colon2.yml«: ebenfalls alles korrekt

5.5 Fehlerbehandlung, Retry-Files

Fehler passieren – leider sogar mitunter dann, wenn man selbst alles richtig gemacht hat. In diesem Abschnitt wollen wir klären, wie Ansible mit Fehlern umgeht.

Man unterscheidet zwei Arten von Fehlern:

1. Parsezeit-Fehler
2. Laufzeit-Fehler

Parsezeit-Fehler sind Fehler, die Ansible entdecken kann, bevor das Playbook überhaupt richtig »losläuft«: Vertipper bei Modulnamen oder anderen Schlüsselwörtern, falsche YAML-Einrückung (sehr beliebt) oder Ähnliches. Dieser Fehlertyp ist eigentlich recht langweilig: Ansible gibt mehr oder weniger hilfreiche Erklärungen aus, Sie beheben den Fehler und versuchen es noch einmal.

Die eigentlich relevanten Fehler sind hingegen die *Laufzeit-Fehler*: Auf irgendeinem Host kann ein Task nicht erfolgreich durchgeführt werden – sozusagen ein schwerer Ausnahmefehler. Standardmäßig verhält sich Ansible bei einem Laufzeit-Fehler wie folgt:

Verhalten bei Laufzeit-Fehlern

Wenn ein Task auf einem Host in einen Fehler läuft, wird der entsprechende Host von allen weiteren Aktionen ausgeschlossen. Für die übrigen bis hierhin fehlerfreien Hosts geht das Play weiter.

Sie können dies mit folgendem kleinen Playbook aus Listing 5.8 testen, das nur auf Debian-artigen Systemen erfolgreich ist. (Wenn Sie in Ihrem Inventory keine Unterscheidung zwischen Debian- und Nicht-Debian-Systemen treffen können, nehmen Sie bitte irgendein anderes Kommando, das nur auf einem Teil Ihrer Target Hosts erfolgreich sein kann.)

```
---
- hosts: all
  gather_facts: no

  tasks:
    - name: Search for "debian" in /etc/os-release
      command: grep -i debian /etc/os-release

    - debug: msg="Let's continue..."
```

Listing 5.8 »fail-on-non-debian.yml«: Schlägt auf Nicht-Debian-Systemen fehl

Hier der Effekt (mit etwas gekürzter Ausgabe):

```
$ ansible-playbook fail-on-non-debian.yml

PLAY [all] *********************************************************************

TASK [Search for "debian" in /etc/os-release] **********************************
changed: [ubuntu]
changed: [debian]
fatal: [rocky]: FAILED! => {"changed": true, "cmd": ["grep", "-i", "debian",
fatal: [suse]: FAILED! => {"changed": true, "cmd": ["grep", "-i", "debian",

TASK [debug] *******************************************************************
ok: [debian] => {
    "msg": "Let's continue..."
}
ok: [ubuntu] => {
    "msg": "Let's continue..."
}

PLAY RECAP *********************************************************************
debian                     : ok=2    changed=1    unreachable=0    failed=0
rocky                      : ok=0    changed=0    unreachable=0    failed=1
suse                       : ok=0    changed=0    unreachable=0    failed=1
ubuntu                     : ok=2    changed=1    unreachable=0    failed=0
```

Retry-Files

Im Fehlerfall kann Ansible eine Datei namens *<PLAYBOOKNAME>.retry* anlegen, die die Namen aller Failed Hosts enthält. Diese können Sie bei Bedarf bei einem erneuten Playbook-Lauf zum Limitieren verwenden (Option `--limit @<RETRYFILE>`), womit die

Menge der Target Hosts gegebenenfalls sehr verkleinert wird. Die Hosts, auf denen bereits alles erfolgreich gelaufen ist, muss man ja schließlich nicht noch einmal behelligen – auch das Feststellen der Idempotenz belastet die Systeme ein wenig.

Es gibt zwei Konfigurationsdirektiven, die dieses Verhalten steuern:

- `retry_files_enabled` legt fest, ob überhaupt Retry-Files erzeugt werden (Default ab Ansible 2.8: `false`, vorher `true`).
- `retry_files_save_path` legt ein Verzeichnis fest, in das die Retry-Files geschrieben werden (per Default neben dem Playbook).

Meine Empfehlung: Retry-Files sind grundsätzlich nicht schlecht, nur der Default-Ablageort stört manchmal. Wenn Sie das Ganze aktivieren möchten, könnten Sie Ansible wie folgt konfigurieren:

```
# [defaults]
retry_files_enabled = yes
retry_files_save_path = ~/.ansible/retry-files
```

Listing 5.9 »ansible.cfg«: Ausschnitt zum Aktivieren von Retry-Files

5.6 Tags

Mitunter ist es wünschenswert, anstelle eines kompletten Playbooks nur Teile davon laufen zu lassen. Wenn Sie beispielsweise gerade den 37. Task eines Playbooks entwickeln, kann es recht lästig sein, bei einem Testlauf immer die ersten 36 Tasks abwarten zu müssen.

Eine Möglichkeit in Ansible ist, Tasks optional mit *Tags* zu versehen, wie es das folgende Beispiel aus Listing 5.10 zeigt:

```
---
- hosts: localhost
  gather_facts: no

  tasks:
    - name: Say hello
      debug: msg="Hello Ansible!"
      tags: hello

    - name: Another message
      debug: msg="Everything is fine so far"
      tags: [test, message]
```

```
  - name: That's it
    debug: msg="Finished"
    tags: always
```

Listing 5.10 »tags.yml«: ein Beispiel für den Einsatz von Tags

Beim zweiten Task sehen Sie, dass auch Listen von Tags möglich sind. Das spezielle Tag `always` bewirkt, dass der entsprechende Task immer ausgeführt wird (außer er wird explizit geskippt). Denkbare Aufrufe wären jetzt z. B.:

```
Alle mit "hello" markierten Tasks:
$ ansible-playbook tags.yml -t hello

Alle mit "hello" oder "test" markierten Tasks:
$ ansible-playbook tags.yml -t hello,test

Alle Tasks außer die mit "message" markierten:
$ ansible-playbook tags.yml --skip-tags message
```

Später kann das alles etwas kniffliger werden, sobald Sie Tasks entwickeln, die aufeinander aufbauen. Wenn z. B. ein früherer Task eine Variable erzeugt, die von einem späteren Task verwendet wird, können Sie logischerweise den späteren Task nicht losgelöst alleine verwenden.

Wichtig

Wenn Sie mit Tags arbeiten, achten Sie stets darauf, dass auch *alle* Tasks, die Sie benötigen, entsprechend getaggt sind!

Sie haben bereits das spezielle Tag `always` gesehen; Tabelle 5.1 zeigt alle zur Verfügung stehenden »Spezialitäten«:

Tag bzw. Aufruf	Bedeutung
Tag: `always`	Der Task wird immer ausgeführt (es sei denn, er wurde explizit geskippt).
Tag: `never`	Der Task wird nie ausgeführt (es sei denn, er wurde explizit angefordert).
`ansible-playbook […] -t tagged`	Nur Tasks ausführen, die wenigstens ein Tag haben.
`ansible-playbook […] -t untagged`	Nur Tasks ausführen, die kein Tag haben.

Tabelle 5.1 Spezielle Tags und Aufrufe

5.7 Das Kommando »ansible-playbook«

Das Kommando **ansible-playbook** ist bei der Arbeit mit Ansible sicher das zentrale Werkzeug (jedenfalls solange Sie die Kommandozeile nutzen). Wie jedes Linux-Kommando bietet es eine Vielfalt von Optionen; ich gebe nun zur Referenz einmal eine Übersicht der wichtigsten Möglichkeiten.

ansible-playbook wird grundsätzlich wie folgt aufgerufen:

```
ansible-playbook [OPTIONS] playbook.yml [playbook2.yml ...]
```

Ja, Sie können **ansible-playbook** auch mit mehreren Playbooks aufrufen, was aber in der Praxis eher selten vorkommt. Dabei werden alle enthaltenen Plays in der entsprechenden Reihenfolge ausgeführt, und spätere Plays können auch auf Ergebnisse früherer Plays zugreifen.

Tabelle 5.2 gibt eine Übersicht der nützlichsten Optionen.

Option	Bedeutung
`-l` <HOST_OR_GROUP>	Aktionen auf gewisse Hosts beschränken
`--list-hosts`	Ausgabe der Hosts, die vom Aufruf betroffen sind
`--list-tasks`	Übersicht über die Tasks eines Playbooks
`--start-at-task` <NAME>	Starte mit Task <NAME>; Joker wie »*« sind möglich
`-i` <INVENTORY_FILE>	Alternatives Inventory benutzen; siehe Abschnitt 2.8
`-v`	Verbose-Mode (bis zu vier Wiederholungen möglich)
`-f` <NUM>	Anzahl der Forks; siehe Abschnitt 3.4.1
`--syntax-check`	syntaktische Überprüfung des Playbooks
`--check`	simulierter Lauf, »Dry Run«; siehe Abschnitt 10.2.6
`--diff`	Dateiänderungen anzeigen, oft mit `--check`; siehe Abschnitt 10.2.6
`--step`	Schritt-für-Schritt-Abarbeitung

Tabelle 5.2 »ansible-playbook«: die wichtigsten Optionen

Option	Bedeutung
`-t <TAGS>`	nur markierte Tasks ausführen; siehe Abschnitt 5.6
`-e <VAR=VALUE>` / `-e @<file.yml>`	Parametrisierung; siehe Abschnitt 6.1.2

Tabelle 5.2 »ansible-playbook«: die wichtigsten Optionen (Forts.)

Das sollte für die meisten Anwendungen ausreichen. Rufen Sie bitte, um die komplette Übersicht zu erhalten, den Befehl **`ansible-playbook -h`** auf.

5.8 Eine exemplarische Apache-Installation

Wenden wir uns nun erstmalig einer anspruchsvolleren, aber dennoch überschaubaren Aufgabe zu: einer Apache-Installation mit kleinen Anpassungen, nämlich der Startseite und der Konfiguration. Das Ziel soll hier sein, gewisse typische Vorgehensweisen zu demonstrieren, und nicht etwa, ein komplettes Beispiel für eine Real-Life-Webserverinstallation zu liefern.

Außerdem wollen wir uns zunächst einmal auf Target Hosts aus der Debian-Familie beschränken, da der Apache-Server in unseren Testdistributionen doch sehr unterschiedlich gehandhabt wird. Diese Unterschiede (elegant) in den Griff zu bekommen, wird eine Aufgabe für später sein.

Nennen Sie Ihr Playbook z. B. *apache.yml*. Wir werden uns Task für Task voranarbeiten; am Ende finden Sie das komplette Playbook am Stück.

5.8.1 Schritt für Schritt

Zu Beginn klären wir die beteiligten Hosts und leiten die Tasks-Sektion ein:

```
---
- hosts: [debian, ubuntu]

  tasks:
```

Sie sehen hier die zwei Zielhosts, dargestellt mit der Inline-Notation einer YAML-Liste, die ein wenig kompakter daherkommt.

Beginnen wir mit dem ersten Task. Auf Debian-artigen Systemen sollte man stets mit aktuellen Paketlisten arbeiten, wenn man Installationen oder Upgrades durchführt. Debian-Anwender kennen dafür die Kommandos **`apt update`** oder **`apt-get update`**; im

Kapitel 7
Module und Collections verwenden

Die Module sind gewissermaßen die Arbeitstiere von Ansible. Mit ihnen lassen sich klar umrissene Aufgaben (*Tasks*) zuverlässig erledigen. Das ist ja bekanntlich auch Bestandteil der klassischen UNIX-Philosophie: Werkzeuge, mit denen man jeweils eine Sache ordentlich und richtig beherrschen kann, sind das A und O. Natürlich gibt es mitunter auch Module, die »zu viel« können, aber Ausnahmen bestätigen die Regel.

Ich möchte Ihnen in diesem Kapitel einen ersten Überblick darüber geben, welche Module Sie *typischerweise* relativ oft brauchen werden. Viele davon haben Sie in den vorherigen Kapiteln auch schon gesehen.

Zu Beginn müssen wir aber – um eine Verständnisgrundlage zu schaffen – einen Blick auf *Collections* werfen, die u.a. zur Verteilung von Modulen genutzt werden.

7.1 Collections

Collections in Ansible sind ein Distributionsformat, mit dem Zusatzkomponenten wie Playbooks, Module, Rollen und Plugins auf standardisierte Art und Weise verpackt und angeboten werden können. Sie sind ein vergleichsweise neuer Bestandteil von Ansible, der mit Version 2.10 vollständig umgesetzt wurde.

Mit ein Grund für die Einführung von Collections war, dass das alte Entwicklungskonzept von Ansible (»Alles was man braucht, ist im Lieferumfang enthalten«, auch: »Batteries Included«) irgendwann nicht mehr tragfähig war. Es wurde immer schwieriger, mit den zu dieser Zeit weit über 3000 Modulen, die zum großen Teil von der Community gepflegt werden, ein vernünftiges Release hinzubekommen.

Also trennte man den Kern von Ansible (*ansible-core*) und die von der Community beigesteuerten Module und Plugins in unterschiedliche Entwicklungspfade auf. (In Abschnitt 1.3 hatte ich das bereits etwas genauer aufgeschlüsselt.)

7.1.1 Eine Minimalumgebung mit »ansible-core«

Wenn Sie Interesse haben, sehen wir uns das einmal in der Praxis an. In einer Virtualenv-Umgebung können Sie das risikofrei durchführen, ohne einer produktiven Installation in die Quere zu kommen.

Richten wir zunächst eine Virtualenv-Umgebung ein (siehe dazu auch Abschnitt 1.6). Auch an dieser Stelle sei noch einmal betont, dass Sie dafür keine Root-Rechte benötigen:

```
$ python3 -m venv ~/venv/ansible-core
$ source ~/venv/ansible-core/bin/activate
```

Nun können Sie das aktuellste stabile *ansible-core*-Paket installieren, das mit Ihrer Python-Version kompatibel ist:

```
$ pip3 install ansible-core
```

Das geht relativ schnell. Ein Grund ist sicher, dass wir nun statt mehreren Tausend Modulen nur noch ein paar Dutzend wichtige Basismodule mit an Bord haben:

```
$ ansible-doc -l | wc -l
70
```

Wenn Sie `| wc -l` weglassen, können Sie sich die Liste auch gern anschauen. Es sind viele alte Bekannte dabei. Möglicherweise sehen Sie aber auch auf den zweiten Blick, dass viele Ihrer Lieblingsmodule fehlen.

Ein beliebiges Beispiel für ein Modul, das in einer »normalen« Ansible-Installation enthalten ist, jedoch hier nicht, ist `ini_file`:

```
$ ansible-doc ini_file
[WARNING]: module ini_file not found in:
[...]
```

Dieses Modul findet sich mittlerweile in der Collection `community.general`, was Sie der Dokumentation *https://docs.ansible.com/ansible/latest/collections/community/general/ini_file_module.html* entnehmen können, die Sie bei einer Recherche sofort finden.

7.1.2 Collections managen

Collections installieren

Mit dem Kommando **`ansible-galaxy`** können Sie u.a. Collections direkt aus dem Internet installieren. Der einfachste Aufruf dafür wäre:

```
$ ansible-galaxy collection install <COLLECTION_NAME>
```

Wir gehen immer noch davon aus, dass wir uns in der eben aktivierten Virtualenv-Umgebung befinden. Dennoch würde eine normale Collection-Installation nach *~/.ansible/collections* schreiben, was später auch unser produktives Ansible beeinflussen könnte. Um dies zu vermeiden, legen wir ein neues Verzeichnis an, z. B.:

```
$ mkdir -p ~/coredemo/collections
```

Dieses Verzeichnis kann nun als Installationsziel gesetzt werden:

```
$ ansible-galaxy collection install \
    -p ~/coredemo/collections community.general
```

Damit Ansible diese Stelle bei der Suche nach Collections auch berücksichtigt, können wir (in unserem Fall temporär) mit der Umgebungsvariablen `ANSIBLE_COLLECTIONS_PATH` arbeiten:

```
$ export ANSIBLE_COLLECTIONS_PATH=~/coredemo/collections
```

Das Modul `ini_file` (und sehr viele andere) stehen nun wieder wie gewohnt zur Verfügung:

```
$ ansible-doc ini_file
```

```
$ ansible-doc -l | wc -l
659
```

Installierte Collections auflisten

Einen Überblick über alle installierten Collections und deren jeweilige Version erhalten Sie mit:

```
$ ansible-galaxy collection list
```

Den Collection-Suchpfad per Konfiguration erweitern

Als Alternative zur Umgebungsvariablen `ANSIBLE_COLLECTIONS_PATH` gibt es natürlich auch einen entsprechenden Konfigurationsparameter; er heißt erwartungsgemäß `collections_path`. (Die Umgebungsvariable ist aber wie üblich »stärker« – also Vorsicht, wenn Sie beides verwenden!)

Leider gibt es keine einfache Methode, den Suchpfad nur zu *erweitern*. Sie müssen dazu den aktuellen Stand herausfinden und Ihren Pfad bzw. Ihre Pfade ergänzen. Der Ansible-Default ist typischerweise `~/.ansible/collections:/usr/share/ansible/collections`; verifizieren Sie das aber noch einmal, beispielsweise mit:

```
$ ansible-config dump | grep COLLECTIONS_PATH
```

Möchten Sie nun in Ihrem Projekt beispielsweise den Suchpfad um den Ordner *~/coredemo/collections* erweitern, dann könnte die Konfiguration so aussehen:

```
# [defaults]
collections_path = [... bisherige Suchpfade ...]:~/coredemo/collections
```

Listing 7.1 »ansible.cfg«: Collection-Suchpfad erweitern

Anmerkung

Machen Sie das im Labor jetzt bitte nicht; stattdessen können Sie nun die Virtualenv-Umgebung mit **deactivate** wieder verlassen (falls Sie diesen kleinen Ausflug überhaupt mitgemacht haben).

7.1.3 Der FQCN (Fully Qualified Collection Name)

Vor allem, wenn Sie schon etwas länger mit Ansible arbeiten, ist Ihnen vielleicht aufgefallen, dass in der Dokumentation mittlerweile stets mit vollqualifizierten Modulnamen gearbeitet wird, also beispielsweise mit

```
- community.general.ini_file:
    path: /tmp/test.ini
    […]
```

anstatt (wie früher) mit:

```
- ini_file:
    path: /tmp/test.ini
    […]
```

Ansible setzt damit ein typisches Namespace-Konzept um, das in erster Linie Namenskollisionen vermeidet. Es könnte ja theoretisch noch eine andere Collection geben, die ebenfalls ein Modul namens `ini_file` enthält.

Aktuell verwenden viele Ansible-Anwender (ich eingeschlossen) bei »Standardmodulen« aber nach wie vor die kurze Form, obwohl die offizielle Dokumentation nahelegt, dies nicht mehr zu tun. Der einfache Grund dafür ist, dass die Kurzform immer noch funktioniert und die lange Form (außer mehr Tipparbeit) keinen wirklichen Mehrwert bringt.

Dafür mussten die Ansible-Entwickler natürlich sorgen, ansonsten hätten sehr viele Menschen Tausende von Tasks in Hunderten von Playbooks und Rollen ändern müssen, was sehr viele sehr ärgerlich gemacht hätte. Deswegen wurde entschieden, dass

alle Module, Plugins und sonstige Objekte, die bis zum Stichtag der Umstellung im alten Ansible enthalten waren, auch in Zukunft mit der kurzen Form ansprechbar bleiben.

Falls Sie sehen möchten, wie das gemacht wird, und Sie die Virtualenv-Umgebung aus dem vorherigen Abschnitt aufgesetzt haben, probieren Sie einmal folgenden Befehl aus:

```
$ grep -r ini_file ~/venv/ansible-core
```

Sie finden dadurch eine Datei *ansible_builtin_runtime.yml*, die Unmengen von Redirects definiert (kurzer Name → langer Name).

Aber auch Anhänger der Kurzform werden hin und wieder vollqualifizierte Namen verwenden: entweder wenn Module verwendet werden, die es in den alten Tagen noch nicht gab, oder wenn diese seit damals ihren Namen geändert haben. Ein x-beliebiges Beispiel für den zweiten Fall ist das Modul `community.crypto.x509_certificate` zur Zertifikatsverwaltung. Das hieß in der guten alten Zeit `openssl_certificate` und könnte in der Tat nach wie vor mit diesem Namen angesprochen werden. Den neuen Namen jedoch gibt es nur im Langformat, und um Verwirrung zu vermeiden, sollte man Ansible-Objekte durchaus stets mit aktuellem Namen ansprechen.

7.1.4 Zwischenfazit

Solange Sie ein Community Package von Ansible verwenden und mit dem Lieferumfang zufrieden sind, brauchen Sie sich keine allzu großen Gedanken um Collections zu machen. Aber auch das Nachinstallieren von Collections wäre ja kein Problem, wie wir gesehen haben.

Sollten Sie aber irgendwann eigene Ansible-Komponenten wie Module oder Plugins entwickeln *und* diese in einem standardisierten Format verteilen wollen, wird das Thema durchaus um einiges interessanter. Wir werden zu gegebener Zeit noch einmal darauf zurückkommen.

Somit kommen wir als Nächstes also endlich zum Hauptanliegen dieses Kapitels: den Modulen.

7.2 Module

Ein Ansible Community Package enthält (je nach Version) ca. 6000 bis 8000 Module. Eine Übersicht über alle lokal verfügbaren Module erhalten Sie mit:

```
$ ansible-doc -l
```