Kapitel 2

Basiseinrichtung und erstes Inventory-Management

In diesem Abschnitt finden Sie einen Vorschlag, 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, 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 *hallo* oder *ansible-test* würde es genauso tun. Mit der

vorgeschlagenen Struktur sind Sie aber schon ein wenig auf die Zukunft vorbereitet: Irgendwann werden Sie mehrere Projekte parallel managen wollen, und Sie werden nicht nur an Projekten arbeiten, sondern auch Rollen, Module, Plugins oder Collections entwickeln.

2.2 Grundkonfiguration (»ansible.cfg«)

Da wir Ansible im Folgenden mit einigen vom Default abweichenden Einstellungen betreiben wollen, benötigen wir eine Konfigurationsdatei. Diese Datei wird an folgenden Stellen gesucht (der Reihe nach, die erste gewinnt):

1. Inhalt der Umgebungsvariablen `ANSIBLE_CONFIG`
2. *ansible.cfg* im aktuellen Verzeichnis
3. *~/.ansible.cfg*
4. */etc/ansible/ansible.cfg*

Gemäß unseren Vorgaben bietet sich Möglichkeit Nr. 2 an. Legen Sie diese erste Version in Ihr Projektverzeichnis *~/ansible/projects/start/*:

```
[defaults]
inventory = ./inventory
```

Listing 2.1 »ansible.cfg«: erste Version einer Konfigurationsdatei

Ansible wird an dieser Stelle mit einem einfachen INI-Format konfiguriert. Die Erklärung der bislang einzigen Einstellung ist einfach; Sie setzen damit den Pfad zu einer (noch zu erstellenden) Inventory-Datei. Der Default wäre hier */etc/ansible/hosts* gewesen, was für uns völlig ungeeignet ist. Solange Sie sich nun im Basisordner des Projekts befinden, sollte Ansible die neue Konfigurationsdatei finden und nutzen:

```
$ ansible --version
ansible 2.10.8
  config file = /home/vagrant/ansible/projects/start/ansible.cfg
  […]
```

Sobald Sie aber Ihr aktuelles Arbeitsverzeichnis ändern, würde Ansible sofort auf eine andere (oder gar keine) Konfiguration zurückgreifen, was nicht in unserem Sinne ist. Da es aber auch nicht immer bequem ist, alle Aufrufe im Basisordner zu machen, empfiehlt es sich zusätzlich, mit der Umgebungsvariablen `ANSIBLE_CONFIG` stets auf unsere Konfigurationsdatei zu verweisen (Möglichkeit 1 aus obiger Liste).

Ich möchte Sie hier unbedingt ermutigen, dies mit der Software *direnv* zu lösen. Denn natürlich könnten Sie diese Umgebungsvariable beispielsweise in Ihrer *.bashrc* setzen, aber über kurz oder lang werden Sie mehrere Ansible-Projekte mit jeweils eigener

Konfiguration verwalten, und dann wird es wieder schwierig. Machen Sie also bitte gegebenenfalls einen kurzen Ausflug zu Anhang A, wo Sie erfahren, wie Sie *direnv* installieren und initialisieren. Danach platzieren Sie folgende Datei *.envrc* in Ihrem Projektordner:

```
export ANSIBLE_CONFIG=$(expand_path ansible.cfg)
```

Listing 2.2 ».envrc«: »direnv«-Konfiguration für Ansible

Damit wird Ansible nun stets auf unsere soeben erstellte Konfigurationsdatei zugreifen, sobald Sie sich im Projektordner oder in dessen Unterverzeichnissen aufhalten.

Weitere, bereits an dieser Stelle interessante Konfigurationsparameter

Die beiden folgenden Konfigurationsparameter sind bereits an dieser Stelle von Interesse:

- **`log_path`**
  Falls gewünscht, können Sie Ansible mit der Direktive `log_path` dazu veranlassen, alle Aktionen mitzuprotokollieren. Die Logdatei ist frei wählbar, nur das Verzeichnis, das die Logdatei enthalten soll, muss existieren. Wenn also z. B. das Verzeichnis *~/logs/* existiert, könnten Sie folgende Einstellung vornehmen:

  ```
  # [defaults]
  log_path = ~/logs/ansible.log
  ```

  Listing 2.3 »ansible.cfg«: Ausschnitt

- **`private_key_file`**
  Wenn der Dateiname Ihres privaten SSH-Schlüssels vom Default abweicht, können Sie diesen über die Konfiguration bekannt machen, um sich die ständige Angabe mittels `--private-key` auf der Kommandozeile zu ersparen:

  ```
  # [defaults]
  private_key_file = ~/.ssh/ansible_key
  ```

  Listing 2.4 »ansible.cfg«: Ausschnitt

2.3 Erstellen und Verwalten eines statischen Inventorys

Um Ansible mitzuteilen, um welche Hosts es sich in Zukunft kümmern soll, können Sie im einfachsten Fall mit einer *Inventory-Datei* arbeiten (auch als *statisches Inventory* bezeichnet). Unser erstes Ziel soll sein, mittels einer solchen Datei den beliebten und wichtigen Ping-Test zu ermöglichen:

```
$ ansible all -m ping
```

Probieren Sie es gern jetzt schon aus; es kann natürlich nicht funktionieren, da Ansible keine Idee hat, welche Hosts mit `all` gemeint sind.

In unserer *ansible.cfg* haben wir ja schon den Pfad zur Inventory-Datei festgelegt (*./inventory*). Andere Dateinamen, die man oft an dieser Stelle sieht, sind *hosts* oder *hosts.ini*. Ich persönlich mag die Endung `.ini` an dieser Stelle nicht, weil der Dateiinhalt schlicht nicht dem INI-Format entspricht. Und weil Sie mittelfristig einen Ordner namens *host_vars* haben werden, nervt es beim ⇆-Drücken, wenn der Inventory-Dateiname mit »ho« beginnt. (Sie sehen schon: Wenn die Software keine strengen Vorgaben macht, entscheidet der Geschmack, und über den lässt sich trefflich streiten.)

Es folgt nun ein exemplarischer Inhalt für die Inventory-Datei, abgestimmt auf unsere Testumgebung:

```
[test_hosts]
debian
rocky
suse
ubuntu

[test_hosts:vars]
ansible_python_interpreter=/usr/bin/python3
ansible_ssh_common_args='-o StrictHostKeyChecking=no -o UserKnownHostsFile=↲
/dev/null'
ansible_user=vagrant
```

Listing 2.5 »inventory«

Hinweis

In diesem Dateiformat gibt es leider keine korrekte Möglichkeit, überlange Zeilen umzubrechen. Das Symbol ↲ soll deswegen anzeigen, dass die folgende Zeile in Wirklichkeit eine Fortsetzung der aktuellen Zeile ist.

Die verwendete Syntax ist nicht ganz klar zu benennen; sagen wir einfach einmal, dass es sich um ein INI-ähnliches Format handelt. Aber davon abgesehen, sollte der Inhalt eigentlich recht gut verständlich sein:

- Mit Namen in eckigen Klammern können Sie Gruppen definieren (in diesem Beispiel: `test_hosts`). Es besteht keine Pflicht, Hosts in Gruppen zu organisieren, aber es ist sehr üblich und nützlich. Eine spezielle Gruppe namens `all`, die alle Hosts enthält, ist übrigens stets automatisch vorhanden und muss nicht eigens von Ihnen definiert werden.

- Einzelhosts können mit zusätzlichen »Schlüssel=Wert«-Zuweisungen parametrisiert werden. Wir nutzen das in diesem Beispiel nicht.
- Hostgruppen können mit dem Zusatz `:vars` ebenfalls parametrisiert werden (hier: `test_hosts:vars`). Als exemplarische Anwendung haben wir Ansible durchgängig die Verwendung von Python 3 vorgeschrieben und für alle Hosts unserer Testgruppe das Prüfen und Ablegen von SSH-Host-Keys deaktiviert, da Labor- bzw. Test-Hosts dazu neigen, öfter mal ihre Host-Keys zu wechseln (z. B. durch erneutes Provisionieren/Installieren). In unserer Testumgebung können wir auf Ereignisse wie

  ```
  The authenticity of host '[…]' can't be established.
  ECDSA key fingerprint is 8a:a3:a5:43:c6:e4:6c:11:3a:c9:2b:97:94:23:40:c9.
  Are you sure you want to continue connecting (yes/no)?
  ```

  oder

  ```
  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  @    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
  ```

  getrost verzichten; im Produktivbetrieb rate ich natürlich strengstens von dieser Einstellung ab (bzw. sollte sie dort nur verwendet werden, wenn Sie sich über die Implikationen völlig im Klaren sind)!

  Außerdem haben wir sicherheitshalber mit `ansible_user` den Account definiert, mit dem sich Ansible anmelden soll. (Defaultmäßig wird – wie auch bei SSH üblich – der lokale Aufrufaccount angenommen.)
- Sollten Sie übrigens in Ihrer Umgebung mit der untypischen UNIX-Passwort-Authentifizierung arbeiten, müssten Sie die entsprechenden Inventory-Hosts noch mit dem zusätzlichen Parameter `ansible_password=<LOGIN_PASSWORT>` versehen und zusätzlich das Paket *sshpass* installieren.

Machen Sie nun den Ping-Test – er sollte komplett grün sein:

```
$ ansible all -m ping
debian | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
ubuntu | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
suse | SUCCESS => {
```

```
    "changed": false,
    "ping": "pong"
}
rocky | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Die Reihenfolge der Server-Ausgaben ist hier nicht beeinflussbar und wegen paralleler Abarbeitung gewissermaßen zufällig. Dazu aber später mehr.

Python-Probleme?

Falls Sie wider Erwarten noch Probleme mit der zu verwendenden Python-Version haben, dann schauen Sie bitte mal in die offizielle Dokumentation unter *http://docs.ansible.com/ansible/latest/reference_appendices/interpreter_discovery.html*. Ab Ansible 2.8 gibt es hier recht differenzierte Einstellungsmöglichkeiten.

2.4 Inventory-Aliasse und Namensbereiche

Bislang sind wir davon ausgegangen, dass die Hosts im Inventory über ihre dort verzeichneten Namen erreichbar sind; d.h., ein `server57.example.org` im Inventory wäre auch für **`ssh`**, **`ping`** und viele andere Tools unter diesem Namen erreichbar.

Das muss aber nicht der Fall sein. Vielleicht haben Ihre Target Hosts keine zugeordneten Namen oder es gibt zwar welche, aber sie gefallen Ihnen einfach nicht.

Das können Sie alles mit *Aliassen* (also alternativen Namen) im Inventory ausgleichen; beispielsweise so:

```
s1  ansible_host=192.168.150.10
s2  ansible_host=192.168.150.20
s3  ansible_host=192.168.150.30
s4  ansible_host=192.168.150.40
```

Listing 2.6 »inventory«: Aliasse für IP-Adressen

oder so:

```
s1  ansible_host=debian.example.org
s2  ansible_host=rocky.example.org
s3  ansible_host=suse.example.org
s4  ansible_host=ubuntu.example.org
```

Listing 2.7 »inventory«: Aliasse für Namen

Dabei müssen die `ansible_host`-Namen natürlich wieder auflösbar sein. Für den Fall, dass Ihre Hosts nicht auf dem Standard-Port 22 erreichbar sind, steht Ihnen natürlich auch der Parameter `ansible_port` zur Verfügung:

```
s1  ansible_host=192.168.150.10  ansible_port=2201
s2  ansible_host=192.168.150.20  ansible_port=2202
s3  ansible_host=192.168.150.30  ansible_port=2203
s4  ansible_host=192.168.150.40  ansible_port=2204
```

Listing 2.8 »inventory«: Aliasse mit zusätzlicher Port-Angabe

Es gibt durchaus noch mehr solcher `ansible_*`-Parameter, aber für den Moment sollten Sie gut versorgt sein. Wer Interesse hat, kann gern einen Blick auf die Webseite *http://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html*, Abschnitt »Connecting to hosts: behavioral inventory parameters«, werfen.

Namensbereiche

Wenn die Namen Ihrer Zielsysteme nach einem abzählbaren Schema vergeben sind, müssen Sie nicht jedes System einzeln aufführen. Sie können viel einfacher mit einer Range-Angabe arbeiten:

```
[webservers]
www[01:50].example.org
```

Listing 2.9 »inventory«: Bereiche von Hosts

Im Bereich der einstelligen Zahlen bekommen Sie dadurch Namen mit führenden Nullen, wie z. B. `www07.example.org`. Lassen Sie hingegen die führende Null in der Range-Angabe weg, bekommen Sie auch keine führenden Nullen im Ergebnis. Ranges von Buchstaben (z. B. `[a:h]`) funktionieren übrigens auch.

2.5 Jenseits von Ping

Ein funktionierender Ping-Test ist ein wichtiger Schritt auf dem Weg zur Basiseinrichtung, aber da er auf den Zielsystemen keinerlei Admin-Rechte erfordert, kann er leider *nicht* zeigen, ob auch die für alle ernsthafteren Aufgaben erforderliche Rechteerhöhung funktioniert. Und natürlich tut sie das momentan nicht. Lassen Sie sich als Beweis einmal die erste Zeile der Datei */etc/shadow* auf allen Zielsystemen ausgeben; eine Anforderung, für die Sie zweifellos Root-Rechte benötigen:

```
$ ansible all -a "head -1 /etc/shadow"
debian | FAILED | rc=1 >>
head: cannot open '/etc/shadow' for reading: Permission denied
```

```
ubuntu | FAILED | rc=1 >>
head: cannot open '/etc/shadow' for reading: Permission denied

rocky | FAILED | rc=1 >>
head: cannot open '/etc/shadow' for reading: Permission denied

suse | CHANGED | rc=0 >>
head: cannot open '/etc/shadow' for reading: Permission denied
```

Damit dies richtig funktioniert, muss Ansible nach der Authentifizierung noch eine Rechteerhöhung durchführen. In unserer Laborumgebung hat der User `vagrant` uneingeschränkte, passwortfreie `sudo`-Rechte; wir kommen deswegen mit einem einzigen zusätzlichen Parameter (`ansible_become=yes`) zum Ziel:

```
[test_hosts]
debian
rocky
suse
ubuntu

[test_hosts:vars]
ansible_python_interpreter=/usr/bin/python3
ansible_ssh_common_args='-o StrictHostKeyChecking=no -o UserKnownHostsFile=↲
/dev/null'
ansible_user=vagrant
ansible_become=yes
```

Listing 2.10 »inventory«

Noch mal der Deutlichkeit halber: Das ist in unserer Laborumgebung nur deswegen bereits ausreichend, weil der User `vagrant` mit passwortfreien `sudo`-Rechten ausgestattet ist. In der Folge erfahren Sie, welche Möglichkeiten Ihnen zur Verfügung stehen, wenn die Rechteerhöhung in Ihrer Umgebung anders organisiert ist.

Parameter zur Rechteerhöhung: Übersicht

Ganz allgemein stehen Ihnen zur Konfiguration der Rechteerhöhung folgende Parameter zur Verfügung:

- **`ansible_become`**
  Der Hinweis an Ansible, dass auf dem Zielhost eine Rechteerhöhung stattfinden muss. Der Default ist `no`.
- **`ansible_become_method`**
  Die dazu zu verwendende Methode. Der Default ist `sudo`.

- **`ansible_become_user`**
  Der Zielaccount des Identitätswechsels. Der Default ist `root`, was in einem solchen Kontext allermeistens passt. Deswegen taucht dieser Parameter in obigem Beispiel auch überhaupt nicht auf.
- **`ansible_become_pass`**
  Das zur Rechteerhöhung gegebenenfalls benötigte Passwort.

Zum Passwort gibt es noch etwas zu bemerken: Im Testbetrieb geht es ja möglicherweise in Ordnung, Klartextpasswörter in einer Konfigurationsdatei vorzuhalten; im Produktivbetrieb wahrscheinlich eher nicht. Welche Alternativen hätten Sie hier?

- Sie könnten sich stattdessen mit der Option `--ask-become-pass / -K` bei jedem Aufruf eines Ansible-Kommandos prompten lassen.
- Sie könnten zumindest im Falle von `sudo` per Konfiguration auf den Target Hosts auf die Passworteingabe verzichten; z. B. für alle Mitglieder der Gruppe `sudo`:

```
# Allow members of group sudo to execute any command
%sudo   ALL=(ALL:ALL) NOPASSWD: ALL
```

Listing 2.11 »/etc/sudoers«: Vollständiger Verzicht auf Passwort-Authentifizierung

Oder nur für einen speziellen Account, so wie es auf den Vagrant-Maschinen vorkonfiguriert ist:

```
vagrant ALL=(ALL) NOPASSWD: ALL
```

Listing 2.12 »/etc/sudoers«: Verzicht auf Passwort-Authentifizierung für vagrant

Beide Optionen wollen vielleicht nicht so recht überzeugen, und für `su` hätten Sie immer noch gar keine automatische Lösung. Ich kann Sie letztlich an dieser Stelle nur auf später vertrösten, denn wenn Sie bald mehr über Ansible wissen, können Sie Hosts oder Gruppen durchaus auch an anderen Stellen parametrisieren und diese Stellen bei Bedarf auch verschlüsseln. Informationen dazu finden Sie in Abschnitt 10.1. In unserem Testlabor sind wir mit dem aktuellen Inventory jedenfalls erst einmal für alle weiteren (administrativen) Aktionen gut aufgestellt:

```
$ ansible all -a "head -1 /etc/shadow"
debian | CHANGED | rc=0 >>
root:!:18333:0:99999:7:::

suse | CHANGED | rc=0 >>
bin:!:18332::::::

ubuntu | CHANGED | rc=0 >>
root:!:18352:0:99999:7:::
```

```
rocky | CHANGED | rc=0 >>
root:$6$wOrl./pGMSOmXY$qD14wtpRzgtOdXI2PJ[…]
```

2.6 Ein etwas komplexeres Beispiel

Unsere Laborumgebung ist zugegebenermaßen ziemlich glatt geschliffen: Trotz vier verschiedener Betriebssysteme haben wir eine völlig einheitliche Authentifizierung und Rechteerhöhung. Deswegen möchte ich Ihnen auch mal ein Inventory für die folgende hypothetische, etwas »rauere« Umgebung zeigen:

- ein Debian-10-System, das direkten Root-Zugang mit SSH-Schlüssel erlaubt
- ein CentOS-7-System, das den Zugang über den Account `ansible` mit dem Passwort `Supergeheim22!` gestattet. Die Rechteerhöhung geschieht mittels `su`, und wir haben nur Python 2 zur Verfügung.
- ein Ubuntu-18.04-System, das den Zugang für den User `user1` mit dem SSH-Schlüssel aus der Datei *keys/ubuntu* erlaubt. Der User hat uneingeschränkte `sudo`-Rechte, und sein Passwort lautet `vmware`.

Mit folgender Inventory-Datei könnten Sie diese Anforderungen abdecken:

```
[debian_hosts]
debian10    ansible_host=192.168.150.10   ansible_user=root

[centos_hosts]
centos7     ansible_host=192.168.150.20

[ubuntu_hosts]
ubuntu1804 ansible_host=192.168.150.40

[centos_hosts:vars]
ansible_user=ansible
ansible_password=Supergeheim22!
ansible_become=yes
ansible_become_method=su
ansible_become_pass=RootPasswOrd
ansible_python_interpreter=/usr/bin/python2

[ubuntu_hosts:vars]
ansible_user=user1
ansible_ssh_private_key_file=keys/ubuntu
ansible_become=yes
ansible_become_pass=vmware
```



```
[all:vars]
ansible_python_interpreter=/usr/bin/python3
ansible_ssh_common_args='-o StrictHostKeyChecking=no -o UserKnownHostsFile=↲
/dev/null'
```

Listing 2.13 »inventory«

Hauptsächlich sind hier folgende Erkenntnisse zu gewinnen:

- Der Einsatz von Gruppen zur Parametrisierung ist meist vorteilhaft.
- Die Parametrisierung von Einzelhosts sollte man sich hingegen weitestgehend ersparen. IP-Adressen sind natürlich eine individuelle Angelegenheit, und für die eine `ansible_user`-Einstellung beim Debian-System hätte sich momentan keine Gruppe gelohnt. Bei zwei oder mehreren solcher Systeme hätte das schon wieder anders ausgesehen.
- Speziellere Einstellungen übertrumpfen allgemeinere Einstellungen (hier zu sehen bei `ansible_python_interpreter`). Host-spezifische Parameter wirken natürlich am stärksten.

2.7 Alternative bzw. mehrere Inventorys

Wenn die Menge der mit Ansible verwalteten Systeme anwächst, werden Sie feststellen, dass es oft nicht mehr ganz passend ist, alle Systeme in einer einzigen Inventory-Datei zu verwalten. Ein typisches Beispiel sind verschiedene Bereitstellungsumgebungen wie Development, Testing oder Production, wobei Sie aber z. B. stets alle Maschinen einer Umgebung mit dem Ansible-Bezeichner »`all`« ansprechen möchten. Später werden Sie auch noch *dynamische Inventorys* kennenlernen, die Sie ganz zwangsläufig getrennt von den statischen Inventorys behandeln müssen.

In erster Ausbaustufe könnten Sie einfach verschiedene Inventory-Dateien bereitstellen. Natürlich kann es nur ein Default-Inventory geben (das Sie in der *ansible.cfg* festgelegt haben), aber bei jedem geeigneten Ansible-Kommando können Sie mit dem Schalter `-i` ein alternatives Inventory angeben, z. B.:

```
$ ansible all -i production-inv -m ping
```

Anmerkung

Erinnern Sie sich noch an die allerersten Testaufrufe, als Sie noch keinerlei Konfiguration hatten? Etwa:

```
$ ansible all -i debian, -m ping
```

Der Schalter `-i` erlaubt es auch, die Liste der Zielhosts kommasepariert direkt auf der Kommandozeile anzugeben. Wenn man aber nur einen einzigen Host angeben möchte, würde Ansible ohne das Komma am Ende den Hostnamen für einen Dateinamen halten und der Aufruf würde natürlich fehlschlagen.

Eine weitere Möglichkeit zur Auswahl eines Inventorys besteht im Setzen der Umgebungsvariablen `ANSIBLE_INVENTORY`:

```
$ export ANSIBLE_INVENTORY=production-inv
$ ansible […]
```

In Kapitel 6 werden Sie unter anderem die *host_vars*- bzw. *group_vars*-Parametrisierung kennenlernen; dabei handelt es sich um eine Möglichkeit zur Parametrisierung von Hosts, die über gewisse Verzeichnisse parallel zum jeweiligen Inventory realisiert wird. Spätestens in dem Fall wäre es sinnvoll, verschiedene Inventorys in einer Verzeichnishierarchie zu organisieren – etwa so:

```
ansible/
|-- inventories/
|   |-- devel/
|   |   |-- group_vars/
|   |   |-- host_vars/
|   |   `-- inventory
|   |-- testing/
|   |   |-- group_vars/
|   |   |-- host_vars/
|   |   `-- inventory
|   `-- production/
|       |-- group_vars/
|       |-- host_vars/
|       `-- inventory
|-- […]
```

Mehrere Inventorys simultan verwenden

Es ist auch möglich, mehrere Inventorys simultan zu verwenden. Dazu wenden Sie die `-i`-Option einfach wiederholt an:

```
$ ansible -i <inventory_1> -i <inventory_2> […]
```

Wenn Sie mit der Umgebungsvariablen arbeiten, sind die Inventorys kommasepariert anzugeben:

```
$ export ANSIBLE_INVENTORY=<inventory_1>,<inventory_2>[,…]
```


Leerzeichen müssen hier natürlich vermieden werden (oder die rechte Seite muss entsprechend quotiert werden).

Es ist sogar möglich, als Inventory ein übergeordnetes Verzeichnis anzugeben, das beliebige Inventorys in beliebiger Hierarchietiefe enthält. Ansible bildet dann automatisch die »Summe« über alle enthaltenen Inventorys.

Achtung

Achten Sie daher unbedingt darauf, in einem solchen Gemisch noch den Überblick zu behalten!

Ich möchte das Thema »Inventory« an dieser Stelle nun erst einmal beiseitelegen, weil wir noch andere Dinge vor uns haben und uns hier nicht zu sehr in Details verlieren wollen.

Ich werde aber in Kapitel 12 noch einmal darauf zurückkommen, weil es durchaus noch einiges dazu zu sagen gibt. Beispielsweise werden wir uns dort die bereits erwähnten dynamischen Inventorys anschauen.

Kapitel 8

Modularisierung mit Rollen und Includes

Würden Sie mit den bisherigen Mitteln reale Server konfigurieren, so hätten Sie schnell Playbooks mit mehreren Hundert enthaltenen Tasks. Das wäre weder in puncto Übersicht noch in puncto Wartbarkeit und Wiederverwendbarkeit wünschenswert.

8.1 Erstellung und Verwendung von Rollen

Um diesem Problem zu begegnen, möchte ich Ihnen nun *Rollen* vorstellen. Diese sind in Ansible der primäre Mechanismus, um die Tasks eines Playbooks sinnvoll und wiederverwendbar zu organisieren.

8.1.1 Das Rollenkonzept in Ansible

In Ansible hat jede Rolle einen Namen, der eins zu eins auf ein Verzeichnis umgesetzt wird. Sinnvolle Namen wären also z. B.:

- »*apache*«
- »*debian/buster/openldap*«

Wie tief Sie Ihre Rollenhierarchien organisieren, ist persönliche Geschmackssache bzw. von der konkreten Problemstellung abhängig. Im Allgemeinen bevorzugt man eher flache oder gar keine Hierarchien.

Anmerkung

Bei der Wahl des Rollennamens sollten Sie sich möglichst auf ASCII-Buchstaben, Zahlen, Unterstriche und Verzeichnistrenner beschränken, also auf Zeichen aus der Menge `[a-zA-Z0-9_/]`.

Sie können problemlos auch andere Zeichen wie Punkt ».« oder Minus »-« verwenden, aber wenn Sie Ihre Rolle eines Tages in eine Collection migrieren möchten, wären diese nicht mehr erlaubt.

Suchpfade

Rollen werden unterhalb des Ordners *roles/* gesucht, und zwar *relativ zum jeweiligen Playbook*. (Und noch an ein paar anderen Stellen, die für unser Projekt nicht relevant sind.)

Um das Ganze übersichtlicher zu halten, empfehle ich, eher einen Ordner *roles/* parallel zum Ordner *playbooks/* verwenden:

```
$ cd ~/ansible/projects/start; mkdir roles
```

Damit Ansible auch dort nach Rollen sucht, müssten Sie noch den Rollensuchpfad erweitern. Dazu steht Ihnen die Konfigurationsdirektive `roles_path` zur Verfügung. Auch die Angabe mehrerer Verzeichnisse ist möglich; trennen Sie diese einfach mit einem Doppelpunkt. Folgende Konfiguration ist also erforderlich:

```
# [defaults]
roles_path = ./roles
```

Listing 8.1 »ansible.cfg«: Rollensuchpfad erweitern

Die Default-Suchpfade bleiben dabei erhalten. Wenn eine Rolle nicht gefunden wird, gibt Ansible in der Fehlermeldung aber auch alle Suchpfade aus; damit sollten Sie eventuelle Probleme schnell beheben können.

Struktur von Rollen

Alle Dateien, die zu einer Rolle gehören, werden durch gewisse Unterverzeichnisse strukturiert. Hier eine Übersicht:

- **<ROLLE>/tasks/main.yml**
  Die Tasks der Rolle
- **<ROLLE>/files/***
  Dateien zum Hochladen
- **<ROLLE>/templates/***
  Jinja-Templates zum Hochladen
- **<ROLLE>/handlers/main.yml**
  Handler, die zur Rolle gehören
- **<ROLLE>/vars/main.yml**
  Variablen, die sehr starke Priorität haben
- **<ROLLE>/defaults/main.yml**
  Variablen mit sehr schwacher Priorität
- **<ROLLE>/meta/main.yml**
  Metainformationen; z. B. Abhängigkeiten von anderen Rollen

Anmerkung

Verzeichnisse, die Sie nicht benötigen, können Sie gern einfach komplett weglassen. Wenn Ihre Rolle beispielsweise keine Templates benutzt, brauchen Sie auch den Unterordner *templates/* nicht.

8.1.2 Ein einfaches Beispiel für eine Rolle

Um das Ganze etwas zu konkretisieren, wollen wir nun eine erste, sehr simple Rolle erstellen. Nennen wir sie der Einfachheit halber `hallo`. Ausgehend vom Wurzelordner unseres Projekts, benötigen Sie also folgende Struktur:

```
roles/
`-- hallo/
    `-- tasks/
        `-- main.yml
```

Legen Sie die Verzeichnishierarchie unterhalb von *roles/* bitte entsprechend an. Den Inhalt der Datei *main.yml* halten wir auch sehr einfach:

```
---
- name: Hallo sagen
  debug: msg="Hallo aus einer Rolle!"
```

Listing 8.2 »hallo/tasks/main.yml«

Um eine Rolle zu laden und zu starten, würden Sie normalerweise ein Playbook anlegen – im nächsten Abschnitt komme ich sogleich darauf zu sprechen. Selbstverständlich können Sie eine Rolle aber auch mit einem `include_role`-Aufruf ad hoc starten:

```
$ ansible localhost -m include_role -a name=hallo
localhost | SUCCESS => {
    "changed": false,
    "include_args": {
        "name": "hallo"
    }
}
localhost | SUCCESS => {
    "msg": "Hallo aus einer Rolle!"
}
```

8.1.3 Rollen in einem Playbook verwenden

Wenn Sie eine Rolle zur Verfügung haben und nutzen möchten, müssen Sie diese nur in ein Playbook einbinden.

Es stehen Ihnen hier zwei grundsätzliche syntaktische Varianten zur Verfügung. Die eine haben Sie bereits im Ad-hoc-Kommando kennengelernt; sie funktioniert natürlich auch im Playbook:

```
---
- hosts: all

  tasks:
    - include_role:
        name: hallo
```

Listing 8.3 »hallo-rolle_v1.yml«: Einbinden einer Rolle in ein Playbook

Die zweite Variante wendet man eher an, wenn durchgängig nur noch mit Rollen gearbeitet wird. Sie ersetzen hier `tasks` durch `roles`:

```
---
- hosts: all

  roles:
    - name: hallo
```

Listing 8.4 »hallo-rolle_v2.yml«: Einbinden einer Rolle in ein Playbook

Anstelle des Attributs `name` können Sie auch `role` verwenden:

```
---
- hosts: all

  roles:
    - role: hallo
```

Listing 8.5 »hallo-rolle_v3.yml«: Einbinden einer Rolle in ein Playbook

Oder Sie können das Attribut gleich ganz weglassen:

```
---
- hosts: all

  roles:
    - hallo
```

Listing 8.6 »hallo-rolle_v4.yml«: Einbinden einer Rolle in ein Playbook

Rollen parametrisieren

Sehr nützlich ist noch die Möglichkeit, eine Rolle bei ihrem Aufruf mit Parametern (Variablen) zu versorgen, die dann auch nur innerhalb der Rolle sichtbar sind:


```
roles:
  - name: hallo
    farbe: blau
    zahl: 57
```

Wenn Sie die Variante des Rollenaufrufs mittels `include_role` bevorzugen, sähe die Parametrisierung wie folgt aus:

```
tasks:
  - include_role:
      name: hallo
    vars:
      farbe: blau
      zahl: 57
```

Erinnern Sie sich an die Präzedenzregeln aus Abschnitt 6.1.3? Mit diesen Methoden der Parametrisierung lägen Sie an Position 20, was kaum zu übertrumpfen ist.

Plays mit Rollen und Tasks: »pre_tasks« und »post_tasks«

In einem Play(book) können Sie durchaus Rollen *und* auch einfache Tasks verwenden:

```
roles:
  [...]

tasks:
  [...]
```

Denken Sie aber daran, dass Sie hier aus technischer Sicht eine Map verwenden, und diese hat keine Reihenfolge! Ob Sie also zuerst Ihre `roles` und dann Ihre `tasks` im Playbook aufführen oder umgekehrt, spielt *keine* Rolle! Wichtig ist allein, wie Ansible intern vorgeht:

- Erst werden alle Rollen unter `roles` ausgeführt.
- Danach werden alle Tasks unter `tasks` ausgeführt.

Wenn Sie unbedingt mischen wollen/müssen, dann sollte zumindest für alle Leserinnen und Leser des Playbooks die Reihenfolge sofort klar sein.

Es empfiehlt sich also eher die Variante mit `include_role` (dann hätten Sie nur Tasks), oder Sie spezifizieren Ihre Tasks unter einer `pre_tasks`- bzw. `post_tasks`-Sektion im Play. Damit erreichen Sie, was der Name schon vermuten lässt: eine möglichst frühe bzw. späte Ausführung dieser Tasks. Sind in einem Play Debian-Systeme beteiligt, sieht man z. B. oft die Aktualisierung des Software-Verzeichnisses als Pre-Task:

```
pre_tasks:
  - name: Update apt cache if needed
```

```
    apt: update_cache=yes cache_valid_time=3600
    tags: always
```

Listing 8.7 Beispiel für die Anwendung von »pre_tasks«

8.1.4 Abhängigkeiten zwischen Rollen

Eine Rolle (sagen wir: »X«) kann Abhängigkeiten von anderen Rollen definieren (z. B. von »Y« und von »Z«). Wenn Sie nun X anwenden, würden automatisch *zunächst* Y und Z ausgeführt.

Technisch gelingt dies über eine Datei *main.yml* im Rollenverzeichnis *meta/*:

```
---
dependencies:
  - Y
  - Z
```

Listing 8.8 »X/meta/main.yml«: Metainformationen zu einer Rolle

Sie könnten die Basisrollen dabei auch parametrisieren:

```
---
dependencies:
  - role: Y
    var1: value1
    var2: value2

  - role: Z
    var3: value3
```

Listing 8.9 »X/meta/main.yml«

Das Ganze ist auch ziemlich intelligent implementiert: Wenn zwei verschiedene Rollen dieselbe Basisrolle verwenden, so würde diese auch nur einmal aufgerufen (*wenn* die Parametrisierung exakt gleich ist!).

Bewertung

Aus technischer Sicht sind Role Dependencys eine Spitzenidee. Rolle A ruft automatisch B und C auf, dadurch ruft B automatisch erst einmal D und E auf ... In der Praxis werden Sie aber ziemlich oft mit offenem Mund davorsitzen und sich fragen: »Warum hat er das jetzt gemacht?« Und zwar selbst dann, wenn Sie das vor drei Tagen erst alles ganz allein eingerichtet haben.

Aber probieren Sie es gern aus, und strafen Sie mich Lügen. Ich möchte jedenfalls an dieser Stelle keine Empfehlung für oder gegen dieses Feature aussprechen.

8.1.5 Erstellen neuer Rollen mit »ansible-galaxy«

Das Erstellen einer Verzeichnisstruktur für eine neue Rolle ist relativ mühsam und auch etwas fehlerträchtig. Ein Tippfehler beim Anlegen eines der Verzeichnisse, z. B. *handler* anstatt *handlers*, genügt, damit es nicht funktioniert.

Lassen Sie sich lieber vom Kommando `ansible-galaxy` unterstützen. Wechseln Sie in das Verzeichnis, in dem Ihre neue Rolle entstehen soll (in unserem Fall *roles/*), und machen Sie dann folgenden Aufruf:

```
$ ansible-galaxy role init <ROLLENNAME>

Oder kürzer ("role" ist das Default-Subkommando):
$ ansible-galaxy init <ROLLENNAME>
```

Probieren Sie es aus:

```
$ ansible-galaxy init testrolle
- Role testrolle was created successfully

$ tree testrolle
testrolle
|-- defaults
|   `-- main.yml
|-- files
|-- handlers
|   `-- main.yml
|-- meta
|   `-- main.yml
|-- README.md
|-- tasks
|   `-- main.yml
|-- templates
|-- tests
|   |-- inventory
|   `-- test.yml
`-- vars
    `-- main.yml
```

Das ist mehr als komplett. Ordner, die Sie nicht benötigen, können Sie gerne löschen. Der *meta/*-Ordner wird hauptsächlich dann benötigt, wenn Sie mit Role Dependencys arbeiten oder Ihre Rolle in der Ansible Galaxy veröffentlichen wollen, und den *tests/*-Ordner brauchen Sie hauptsächlich dann, wenn CI-Systeme wie Jenkins oder Travis CI ins Spiel kommen.

Bug in Debian 11-Version?

Merkwürdigerweise fehlen bei der in Debian 11 paketierten Ansible-Version nach einem `ansible-galaxy init` die Verzeichnisse *files/* und *templates/*. Sollte Ihre Version davon auch betroffen sein: Da müssen Sie im Bedarfsfall doch noch mit `mkdir` ran!

Das Kommando `ansible-galaxy role init` hat auch noch einige Optionen. Für uns an dieser Stelle interessant sind `--force` (eine bereits bestehende Rolle überschreiben), `--init-path <PFAD>` (einen anderen Zielordner als das aktuelle Verzeichnis wählen) und `--role-skeleton <PFAD>` (eine andere Rolle als Vorlage nutzen).

8.2 Das Online-Repository Ansible Galaxy

Die *Ansible Galaxy* ist *das* Online-Repository für Rollen und Collections. Sie finden die Projekt-Homepage unter *https://galaxy.ansible.com/*. Wenn Sie dort nach Rollen stöbern möchten, betreten Sie zunächst den Bereich SEARCH. Neben dem Textfeld zur Suche blenden Sie dann die FILTERS ein, und bei TYPE wählen Sie `Role`. So beinhalten Ihre Suchergebnisse nur Rollen. Auch Collections können Rollen enthalten, aber sie werden primär als Distributionsformat für Module oder Plugins verwendet.

Wenn Sie eine interessante Rolle gefunden haben, können Sie diese direkt herunterladen, und zwar mit:

```
$ ansible-galaxy install <author.rolename>
```

(Immer vorausgesetzt, ein direkter Internetzugriff ist möglich.) Wenn Sie nur mal schauen möchten, reicht vielleicht auch der Blick ins Repository; Sie finden rechts oben meist eine Verknüpfung zum GITHUB REPO.

Achtung

Wie Sie sich sicher denken können, ist es im Allgemeinen nicht empfehlenswert, »irgendwelche« Galaxy-Rollen ohne vorherige Prüfung einfach so zu übernehmen und anzuwenden. Bei Autoren, denen Sie vertrauen, oder »prominenten« Community-Collections können Sie das natürlich gegebenenfalls etwas entspannter sehen.

8.3 Verwendung von Imports/Includes

Neben den Rollen steht in Ansible noch ein einfacheres Konzept zur Modularisierung zur Verfügung: Mit einem Import oder Include können Sie Tasks oder Variablen aus

externen Dateien einbinden. Auch das wird gerne genutzt, um Übersicht und Wiederverwendbarkeit zu erhöhen.

8.3.1 »import_tasks« und »include_tasks«

In umfangreicheren Rollen ist z. B. die *tasks/main.yml* oft in weitere Dateien aufgeteilt:

```
---
- import_tasks: mach_dies.yml
- import_tasks: mach_jenes.yml
```

Listing 8.10 »tasks/main.yml«: exemplarische Anwendung von Imports

Vor Ansible Version 2.4 gab es eine Direktive `include`, die aber seit 2.4 als deprecated (veraltet) gilt. Das aktuelle Ansible unterscheidet zwischen `import_tasks` und `include_tasks`; obiges Beispiel hätte aber auch mit `include_tasks` funktioniert. Was also ist der Unterschied? Vereinfacht gesagt, erfolgt ein Import während der Parsezeit und ein Include während der Laufzeit des Playbooks. Sie verwenden also:

- **Imports für statische Angelegenheiten**, die von Anfang an feststehen
- **Includes für dynamische Angelegenheiten**, die sich erst während der Laufzeit ergeben

Beispielsweise ist ein wiederholtes Einbinden in einer Schleife oder das Einbinden einer Datei mit dynamischem Dateinamen überhaupt nur mit `include_tasks` möglich:

```
- include_tasks: part.yml
  with_items:
    - foo
    - bar

- include_tasks: "{{ ansible_os_family }}.yml"
```

Listing 8.11 Zwei Beispiele, die nur mit »include_tasks« realisierbar sind

Die Details können Sie bei Bedarf unter *http://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse.html* nachlesen.

8.3.2 »include_tasks« und Tags

Die Kombination von `include_tasks` und Tags birgt einige Tücken. Betrachten wir das folgende Beispiel:

```
---
- hosts: localhost
  gather_facts: false
```

```
  tasks:
    - include_tasks: some_tasks.yml
      tags: tasks
```

Listing 8.12 »main-playbook.yml«

Und die zu inkludierende Datei:

```
---
- debug:
    msg: Ein erster Task
  tags: task1

- debug:
    msg: Ein zweiter Task
  tags: task2
```

Listing 8.13 »some_tasks.yml«

Zur Diskussion steht jetzt der folgende Aufruf:

```
$ ansible-playbook main-playbook.yml -t tasks
```

Das Include als solches wird ausgeführt, nicht aber die inkludierten Tasks! (Mit `import_tasks` würde das übrigens funktionieren; allerdings hat man nicht immer die Wahl.)

Vielleicht finden Sie das aber völlig logisch und stellen sogleich auch fest, dass der folgende Aufruf wie erwartet funktioniert:

```
$ ansible-playbook main-playbook.yml -t tasks,task1
```

Es bleibt aber die Frage, was zu tun wäre, wenn die Tasks in der inkludierten Datei überhaupt keine Tags hätten. Wie schafft man es dann, sie im Ganzen zur Ausführung zu bringen? Hier gibt es zwei Möglichkeiten.

Die erste ist, einen Block ins Spiel zu bringen; das sieht aus wie fauler Zauber, aber wird durchaus auch von offizieller Stelle so vorgeschlagen:

```
---
- hosts: localhost
  gather_facts: false

  tasks:
    - block:
        - include_tasks: some_tasks.yml
      tags: tasks
```

Listing 8.14 »main-playbook2.yml«

Die zweite Möglichkeit beruht auf dem `include_tasks`-Parameter `apply`, mit dem Sie verschiedene Keywords in die Tasks der inkludierten Datei »hineinvererben« können. Das Ganze wirkt deutlich komplizierter, und bringt zumindest für den Use Case der Tags auch sonst keine Vorteile:

```
---
- hosts: localhost
  gather_facts: false

  tasks:
    - include_tasks:
        file: some_tasks.yml
        apply:
          tags: tasks
      tags: tasks
```

Listing 8.15 »main-playbook3.yml«

8.3.3 Dynamisches Laden von Variablen mit »include_vars«

Das `include_vars`-Modul ist zum dynamischen Nachladen von Variablen gedacht. Es wird gern auf die folgende Weise zum Ausgleichen von Distributionsunterschieden genutzt:

```
---
- name: Include OS-specific settings
  include_vars: "{{ ansible_os_family }}.yml"
```

Listing 8.16 »tasks/main.yml«: dynamisches Laden von Variablen-Settings

Das haben wir ja auch schon in Abschnitt 6.5.4 gesehen. In einer Rolle angewandt, ist das Suchverzeichnis bei relativen Pfadangaben der *vars/*-Ordner.

Wenn Sie nur wenige Systemfamilien haben, die aus der Reihe tanzen, ist auch ein Ansatz wie dieser sehr schlau:

```
---
- name: Include OS-specific settings
  include_vars: "{{ item }}"
  with_first_found:
    - "{{ ansible_os_family }}.yml"
    - defaults.yml
```

Listing 8.17 »tasks/main.yml«: Beispiel für den Einsatz von »with_first_found« in einer Rolle

Wie der Name bereits vermuten lässt, versucht `with_first_found` der Reihe nach die angegebenen Dateien zu finden und bereitzustellen. Der erste Treffer gewinnt, danach wird die Suche beendet.

8.4 Noch mal Apache

Jetzt, da alle Techniken in der Theorie besprochen sind, möchte ich noch einmal unser Apache-Thema aufgreifen und das Playbook aus Abschnitt 6.5.5 in eine Rolle verwandeln. Nennen wir sie der Einfachheit halber »apache«.

Erzeugen Sie als Startpunkt im *roles/*-Ordner eine leere Hülle:

```
$ cd roles
$ ansible-galaxy init apache
- Role apache was created successfully

Für diese Rolle überflüssige Verzeichnisse entfernen:
$ rm -rf apache/{files,meta,tests}

Ggf. templates/ anlegen:
$ mkdir -p apache/templates
```

Zunächst zum Wichtigsten: den Tasks. Im Prinzip übernehmen Sie einfach die Taskliste (sie war im Playbook in Ordnung und ist es natürlich immer noch). In Listing 8.18 sehen Sie aber die spannendere Version mit der externen Template-Datei. Auf das Präfix `apache_` in den Namen der zu inkludierenden Variablendateien können wir nun verzichten, da im Rollenkontext keine Kollisionsgefahr mehr besteht:

```
---
- name: Systemspezifische Parameter laden
  include_vars: >
    {{ ansible_distribution | lower | replace(" ", "-") }}.yml

- name: Paketlisten auf Debian-Systemen aktualisieren
  apt:
    update_cache: yes
    cache_valid_time: 3600
  when: ansible_os_family == "Debian"

- name: Apache-Paket installieren
  package:
    name: "{{apache_package_name}}"

- name: Dienst starten und in Bootprozess integrieren
  service:
    name: "{{apache_service_name}}"
    state: started
    enabled: yes

- name: Minimale Startseite einrichten
  template:
    src: index.html.j2
    dest: "{{apache_document_root}}/index.html"
    mode: 0644

- name: Plugin-Config hochladen
  copy:
    dest: "{{apache_config_directory}}/redirect.conf"
    content: |
      Redirect /go http://www.google.de
  notify: reload apache

- name: Config aktivieren (nur auf Debian-System nötig)
  command:
    cmd: a2enconf redirect
    creates: /etc/apache2/conf-enabled/redirect.conf
  when: ansible_os_family == "Debian"
```

Listing 8.18 »apache/tasks/main.yml«: Rollen-Tasks mit externer Template-Datei

Die OS-spezifischen Variablen-Files übernehmen Sie auch eins zu eins in den *vars/*-Ordner der Rolle; nur verzichten Sie wie gesagt auf das nun unnötige Präfix `apache_` in den Dateinamen:

```
---
apache_package_name: apache2
apache_service_name: apache2
apache_document_root: /var/www/html
apache_config_directory: /etc/apache2/conf-available
```

Listing 8.19 »apache/vars/debian.yml«

```
---
apache_package_name: httpd
apache_service_name: httpd
apache_document_root: /var/www/html
apache_config_directory: /etc/httpd/conf.d
```

Listing 8.20 »apache/vars/rocky.yml«

```
---
apache_package_name: apache2
apache_service_name: apache2
apache_document_root: /srv/www/htdocs
apache_config_directory: /etc/apache2/conf.d
```

Listing 8.21 »apache/vars/opensuse-leap.yml«

```
---
apache_package_name: apache2
apache_service_name: apache2
apache_document_root: /var/www/html
apache_config_directory: /etc/apache2/conf-available
```

Listing 8.22 »apache/vars/ubuntu.yml«

Die externe Template-Datei *index.html.j2* packen Sie in den *templates/*-Ordner der Rolle. Ein kleiner Unterschied zur alten Version ist, dass Sie für die Variable `apache_farbe` keinen Default mehr im Template bereitstellen, sondern im *defaults/*-Ordner der Rolle, worauf wir gleich zu sprechen kommen:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>{{inventory_hostname}}</title>
  </head>
  <body bgcolor="{{apache_farbe}}">
    <h1>Willkommen auf {{inventory_hostname}}!</h1>
    <h2>Lernen Sie unser Team kennen:</h2>
    <ul>
    {% for host in groups['all'] %}
      <li>
      {% if host != inventory_hostname %}
      <a href="http://{{ hostvars[host]['ansible_all_ipv4_addresses'] |
                         map('regex_search','^192\..*')              |
                         select('string')                           |
                         first }}">
        {{host}}
      </a>
      {% else %}
        <span style="background-color: LightCyan">{{host}}</span>
      {% endif %}
      </li>
    {% endfor %}
```



```
    </ul>
  </body>
</html>
```

Listing 8.23 »apache/templates/index.html.j2«

Auch den einzigen Handler dürfen wir nicht vergessen; er gehört (wie Sie sich sicher bereits denken können) in die Datei *handlers/main.yml*:

```
---
- name: reload apache
  service:
    name: "{{apache_service_name}}"
    state: reloaded
```

Listing 8.24 »apache/handlers/main.yml«

Sorgen wir schließlich noch für eine Default-Parametrisierung der Rolle. Eine der Best Practices beim Erstellen von Rollen lautet: Nach Möglichkeit sollten stets sinnvolle Defaultwerte für alle konfigurierbaren Parameter mitgeliefert werden. Dafür ist die Rollendatei *defaults/main.yml* vorgesehen:

```
---
apache_farbe: LightBlue
```

Listing 8.25 »apache/defaults/main.yml«

Tipp

Versuchen Sie möglichst, alle Ihre Rollenparameter mit einem gemeinsamen Präfix zu versehen (hier: »`apache_`«). Das erhöht die Übersicht und beugt möglichen Namenskollisionen vor.

Rollen-Defaults rangieren übrigens auf Platz 2 der Präzedenzliste und können daher im Prinzip durch jeden anderen Mechanismus zur Variablendeklaration übertrumpft werden. Bevor Sie die neue Rolle nun ausprobieren: Entfernen Sie gegebenenfalls bitte alle Altlasten an folgenden Orten, die mit dem Apache-Thema zu tun haben:

- *playbooks/vars*
- *playbooks/templates*
- *group_vars/all.yml*

Ansonsten könnte es sein, dass beim Testen etwas nur zufällig funktioniert, weil sich ein Play an diesen Orten bedient, anstatt – wie es sein soll – in den Rollenunterordnern!

Testen können Sie nun ad hoc oder mit einem einfachen Playbook wie diesem, das nichts anderes mehr tut, als unsere neue Rolle aufzurufen:

```
---
- hosts: all

  roles:
    - name: apache
```

Listing 8.26 »apache4.yml«

Wenn alles so läuft wie gedacht, sollten unsere Apache-Server uns nun mit einer hellblauen Startseite begrüßen!

8.5 Dokumentation

Wir müssen noch über das lästige Thema »Dokumentation« sprechen. Denn leider werden sogar Sie selbst nach spätestens zwei Wochen nicht mehr wissen, was Ihre Rollen tun – geschweige denn, welche Parameter möglich bzw. erforderlich sind. Wenn Sie sich jetzt noch Kolleginnen oder Kollegen vorstellen, die Ihre Rolle zum ersten Mal verwenden möchten, ohne den Quellcode von A bis Z studieren zu müssen ...

Um das Problem in den Griff zu bekommen, wird in der Praxis meist ein Gemisch aus zwei Ansätzen verfolgt:

1. Guter Code dokumentiert sich selbst.
2. Eine beigefügte README-Datei sorgt für den Rest.

Die nächsten Abschnitte zeigen dazu einige Best Practices.

8.5.1 Code-Konventionen

Nehmen wir einmal großzügig an, alle Beteiligten haben eine Idee, worum es sich bei einer Rolle namens »`apache`« ungefähr handeln könnte. Dann bleiben als wichtigste Fragen:

- Mit welchen Parametern kann ich das Verhalten der Rolle beeinflussen?
- Gibt es sogar Pflichtparameter, ohne die die Rolle überhaupt nicht arbeiten kann?

Als Coding-Konvention könnte man nun vereinbaren, dass *alle* Parameter stets in der Datei *defaults/main.yml* zusammen mit einem kleinen Kommentar aufgeführt sind. Hier sehen Sie ein reines Beispiel, das *nicht* die Funktionalität unserer momentanen Apache-Rolle widerspiegelt:


```
---
# Default-Hintergrundfarbe für die Startseite:
apache_farbe: LightBlue

# Redirect für die Google-Weiterleitung:
apache_google_redirect: /go

# User/Passwort für den Zugriff auf geschützte Bereiche:
apache_admin_user: chef
apache_admin_pass: # no default
```

Listing 8.27 »defaults/main.yml«

Die letzte Zeile ist bemerkenswert: Sie möchten zwar den Parameter spezifizieren, aber absichtlich *keinen* Wert, denn Default-Passwörter waren ja noch nie eine gute Idee. Rein YAML-technisch hätten Sie auch jede andere der in Abschnitt 4.6 beschriebenen Methoden nutzen können; versuchen Sie, sich im Team auf eine zu einigen.

Konsequenterweise muss die Rolle jetzt aber auch möglichst schnell aussteigen, wenn der Parameter vom Anwender nicht gesetzt wird. Erinnern Sie sich an das `assert`-Modul? Setzen Sie es in einem sehr frühen Task ein:

```
---
- assert:
    that:
      - apache_admin_pass != None
    fail_msg: Setzen Sie bitte ein Passwort mit apache_admin_pass
```

Listing 8.28 »tasks/main.yml«

8.5.2 »README.md«

Der zweite Ansatz, der gewisse Coding-Konventionen hervorragend ergänzt, ist das Mitliefern einer README-Datei. Ein simples Textformat wäre hier schon ausreichend, aber in den letzten Jahren hat sich das *Markdown*-Format zum De-facto-Standard für solche Zwecke etabliert. Markdown ist eine simple Auszeichnungssprache und ziemlich leicht zu lernen (siehe z. B. *http://markdown.de*). Außerdem hat es den Vorteil, dass es auch mit einem simplen Textbetrachter gut lesbar ist (im Unterschied zu Formaten wie HTML oder LaTeX).

Bei Bedarf können Sie Markdown leicht in »schönere« Formate konvertieren; Git-Server mit grafischer Oberfläche wie Gitea oder GitLab tun das schon vollautomatisch. Auch die Ansible Galaxy tut dies, und die vom Kommando `ansible-galaxy init` produzierte *README.md* entspricht genau den dortigen Vorgaben.

Wenn Sie nicht planen, Ihre Rollen in der Galaxy zu veröffentlichen, sollten Sie eine eigene Schablone entwickeln, die für Ihr Unternehmen gut passt und von allen Beteiligten gemocht wird. Beispielsweise so etwas wie in Listing 8.29:

```
# Ansible-Rolle: apache

Eine einfache Apache-Installation auf verschiedenen Linux-Systemen

## Voraussetzungen

Die Firewall muss deaktiviert sein, oder Port 80 muss anderweitig
geöffnet werden.

## Geeignet für

Debian 11, Rocky 8, openSUSE 15, Ubuntu 20.04

## Parameter mit Default-Werten

    apache_farbe: LightBlue
    apache_google_redirect: /go
    apache_admin_user: chef
    apache_admin_pass: # no default

## Beschreibung

Diese Rolle wurde rein zu Demonstrationszwecken erstellt. Sie installiert
auf verschiedenen Linux-Systemen einen Apache-Webserver, startet ihn
und richtet eine einfache Startseite ein. Eine Weiterleitungsregel
nach Google wird ebenfalls konfiguriert.

Mit dem Parameter `apache_farbe` kann die Hintergrundfarbe der Startseite
bestimmt werden. Alle anderen Parameter sind nur zur internen Verwendung.

## Beispiel-Playbook

```
- hosts: all

  roles:
    - name: apache
```
```



```
## Autor

Willi Winzig <winzig@example.org>
```

Listing 8.29 »apache/README.md«: exemplarische Rollendokumentation

So etwas können Sie natürlich mit jedem Texteditor bearbeiten. Wenn Sie aber schon während des Schreibens sofort überprüfen möchten, ob alles gut aussieht, empfiehlt sich die Verwendung eines Editors wie *ReText*, der einen Vorschau- bzw. WYSIWYG-Modus anbietet.

Manpages

Natürlich können Sie aus dem Markdown-Format auch eine klassische Manpage erzeugen. Die Software *Pandoc* leistet hier gute Dienste. In Listing 8.30 sehen Sie einen Vorschlag für ein Shellskript `andoc`, das einen `pandoc`-Aufruf kapselt:

```
#!/bin/bash
ROLE=$1;
ROLES_PATHS=(~/ansible/roles ~/adm/ansible/roles)

if [ -z "$ROLE" ]; then
    echo "What role documentation do you want?"
    exit 1
fi

for p in "${ROLES_PATHS[@]}"; do
    README=$p/$ROLE/README.md
    test -e $README && break
done

if [ ! -e $README ]; then
    echo "$ROLE: No such role"
    exit 1
fi

pandoc -s $README -t man -V title="Ansible Role" -V section=7 | man -l -
```

Listing 8.30 »andoc«: ein Markdown-zu-Manpage-Konverter

`ROLES_PATHS` muss gegebenenfalls noch an Ihre Rollensuchpfade angepasst werden. Eine Manpage-artige Hilfsseite bekommen Sie nun mit dem Aufruf:

```
$ andoc <ROLLE>
```

8.6 Wiederverwendung von Rollen

Nach allem, was Sie bislang in diesem Kapitel erfahren haben, bleibt womöglich immer noch die Frage, wie Rollen in der Realität zur Wiederverwendung kommen sollen. Wir haben Rollen im *roles/*-Ordner unseres Ansible-Projekts angesiedelt, womit sie für alle Playbooks innerhalb dieses Projekts verfügbar sind. Aber was ist mit dem nächsten Projekt, das diese Rollen ebenfalls verwenden möchte?

Die logische und gute Idee wäre, die Rollen losgelöst in ein eigenes Projekt zu verschieben. Und dabei sollten Sie sie auch gleich in eine Collection-Struktur einbetten, wozu aber unbedingt ein Ansible ≥ 2.10 benötigt wird. (Sie könnten ein ziemlich vergleichbares Setup auch mit älteren Ansible-Versionen und ohne Collections hinbekommen, aber Collections sind in Ansible sowohl der Stand der Dinge als auch die Zukunft. Vergessen Sie also gegebenenfalls Ihre Steinzeit-Version von Ansible, und machen Sie ein Upgrade!)

Erweiterung der Verzeichnisstruktur

Um zu demonstrieren, wie das konkret aussehen könnte, sollten wir mit einer Kopie des bisherigen Projekts weitermachen. Dort hat ja alles gut funktioniert, und diesen Stand möchten wir auf keinen Fall verlieren:

```
$ cd ~/ansible/projects
$ cp -a start demo1
```

Für die neu ins Spiel kommenden Collections legen wir eine eigene Verzeichnisstruktur an und wechseln auch gleich hinein:

```
$ mkdir -p ~/ansible/collections/ansible_collections && cd $_
```

[!]

Wichtig

Das eigentlich redundante Zwischenverzeichnis *ansible_collections* ist technisch leider notwendig!

Nun erzeugen wir das Basislayout für eine neue Collection, gleich noch zuzüglich eines Ordners für Rollen:

```
$ ansible-galaxy collection init local.www
- Collection local.www was created successfully
$ mkdir local/www/roles
```

Nun kann die Apache-Rolle aus ihrer bisherigen Stelle in die Collection »umziehen«:

```
$ mv ../../projects/demo1/roles/apache local/www/roles
```


In einem Projekt-Playbook, das diese Rolle nutzen möchte, würde dann der FQCN (Fully Qualified Collection Name) zur Verwendung kommen – z. B. so:

```
---
- hosts: all

  roles:
    - name: local.www.apache
```

Listing 8.31 »apache5.yml«

Aber noch wird die Rolle nicht gefunden, da wir ihre übergeordnete Collection an einer Nicht-Standard-Stelle »installiert« haben.

Stellen wir uns für den Moment einmal vor, dass sowohl unser Projekt als auch die Collection als Git-Repository vorliegen, wie es Tabelle 8.1 beschreibt:

Verzeichnis	Git-Repo
projects/demo1	`ansible/project-demo1`
collections/ansible_collections/local/www	`ansible/collection-local-www`

Tabelle 8.1 Hypothetische Git-Repositorys

Wenn Sie einen Git-Server vergleichbar zu dem aus Kapitel 9 zur Verfügung haben, können Sie es gern auch testweise realisieren. Ansonsten lesen Sie erst einmal weiter, glauben es einstweilen und probieren es gegebenenfalls später aus.

Unser Ziel muss nun sein, dass ein hypothetischer Anwender lediglich das Projekt `project-demo1` auschecken muss und danach in Schritt 2 über einen einfachen Automatismus alle benötigten Collections nachinstallieren kann. Schließlich sollte ein reiner Anwender über diese internen Abhängigkeiten möglichst nichts wissen müssen. Sie sollten übrigens nicht nur die menschlichen Anwender im Blick haben, sondern auch Webanwendungen wie in Kapitel 9, mit denen Sie vielleicht irgendwann Ihre Playbooks starten möchten.

In einer solchen Situation wird das Problem noch viel offensichtlicher, denn eine Anwendung kann keine *README.txt*-Datei lesen, in der »Du musst vorher diese und jene Collections installieren!« steht. Die Lösung für Mensch und Maschine ist eine Projektdatei *collections/requirements.yml*, in der alle Abhängigkeiten in einem standardisierten Format spezifiziert sind:

```
---
collections:
```

```
- name: http://git.example.org/ansible/collection-local-www.git
  type: git
```

Listing 8.32 »collections/requirements.yml«

Mit dem folgenden Aufruf lassen sich nun alle aufgeführten Abhängigkeiten nachinstallieren, die dann per Default im Collection-Suchpfad *~/.ansible/collections/* landen:

```
$ ansible-galaxy install -r collections/requirements.yml -f
```

Die Option `-r` erwartet als Parameter die Requirements-Datei, und `-f` erzwingt das Überschreiben bereits installierter Collections, um auf jeden Fall den letzten Stand aus dem Versionskontrollsystem zu bekommen.

[!]

Wichtig

Da man den Namen der Requirements-Datei sowieso angeben muss, hätte sie auch irgendwie heißen und irgendwo liegen können. Aber die Datei *collections/requirements.yml* wird von Webanwendungen wie AWX standardmäßig gesucht und gegebenenfalls verarbeitet.

Sie sollten sich also daran halten – außer Sie schließen die zukünftige Verwendung solcher Werkzeuge kategorisch aus.

Die Situation des Entwicklers

Sollten Sie in der Situation sein, dass Sie sowohl Collections als auch zugehörige Projekte entwickeln, dann liegen ja beide Verzeichnisse schon in der aktuellsten Version auf Ihrer Festplatte, und Sie werden wahrscheinlich keine Lust haben, nach jeder kleinen Änderung in der Collection das obige **`ansible-galaxy install`**-Kommando aufzurufen. (Ganz zu schweigen vom jeweils vorweg erforderlichen **`git commit/push`**.)

Stattdessen könnten Sie den Collection-Suchpfad dauerhaft erweitern, entweder über die *ansible.cfg*:

```
# [default]
collections_paths = ~/ansible/collections
```

Listing 8.33 »ansible.cfg«

Oder über die Umgebungsvariable `ANSIBLE_COLLECTIONS_PATHS`, die Sie in der *.envrc* des Projekts unterbringen können (Sie arbeiten doch mit *direnv*, oder?):

```
export ANSIBLE_CONFIG=$(expand_path ansible.cfg)
export ANSIBLE_COLLECTIONS_PATHS=~/ansible/collections
```

Listing 8.34 ».envrc«: »direnv«-Konfiguration für Ansible

Beides hätte aber einen *gravierenden* Nachteil: Diese Einträge bestimmen leider auch, wohin neue Collections per Default installiert werden. Wenn sie dauerhaft im Projekt (also im Git-Repo) festgeschrieben werden, kann das also unliebsame Konsequenzen haben, sowohl für die Anwenderschaft, als auch für Sie beim entwickeln (z. B. wenn Sie doch mal das **`ansible-galaxy install`**-Kommando testen).

[+]

Tipp

Mein Vorschlag für Collection- bzw. Rollenentwickler ist an dieser Stelle, die Umgebungsvariable `ANSIBLE_COLLECTIONS_PATHS` wirklich nur interaktiv bei Bedarf zu setzen. Das können Sie z. B. mit einem Alias relativ unaufwendig erledigen.

Kapitel 12

Inventory-Management: fortgeschrittene Methoden

In Kapitel 2 haben wir einfache statische Inventorys betrachtet und mit Absicht noch einige Punkte zum Thema Inventory-Management offen gelassen. Nun ist ein guter Zeitpunkt, um noch einmal auf Inventorys zurückzukommen.

12.1 Das Kommando »ansible-inventory«

Lassen Sie uns mit einem weiteren nützlichen Tool aus der Ansible-Suite beginnen: `ansible-inventory`. Mit diesem Kommando können Sie Informationen über Ihr Inventory gewinnen und die Parametrisierung von Hosts darstellen. Versuchen Sie einmal folgende Aufrufe:

```
Übersicht (ASCII-Grafik); ohne und mit Gruppenparametrisierung:
ansible-inventory --graph
ansible-inventory --graph --vars

Das Gleiche, eingeschränkt auf eine Gruppe:
ansible-inventory --graph <GROUPNAME>
ansible-inventory --graph --vars <GROUPNAME>

Übersicht (JSON bzw. YAML) mit Parametrisierung:
ansible-inventory --list
ansible-inventory --list --yaml

Übersicht über die Parametrisierung (JSON bzw. YAML) von einzelnen Hosts:
ansible-inventory --host <HOSTNAME>
ansible-inventory --host <HOSTNAME> --yaml
```

Hier sehen Sie eine exemplarische Ausgabe:

```
$ ansible-inventory --graph
@all:
  |--@test_hosts:
  |  |--debian
  |  |--rocky
```

```
  |  |--suse
  |  |--ubuntu
  |--@ungrouped:
```

12.2 Verschachtelte Gruppen

Anders als bei Benutzergruppen in Linux können Gruppen in einem Ansible-Inventory durchaus verschachtelt sein, d. h., Sie können Gruppen von Gruppen definieren. Eine solche »Supergruppe« definieren Sie im statischen Inventory, indem Sie an den Gruppennamen das Suffix `:children` anhängen:

```
[debian_hosts]
debian

[rocky_hosts]
rocky

[suse_hosts]
suse

[ubuntu_hosts]
ubuntu

[apt_hosts:children]
debian_hosts
ubuntu_hosts

[rpm_hosts:children]
rocky_hosts
suse_hosts
```

Listing 12.1 »inventory«: Beispiel für verschachtelte Gruppen

Sie sehen in diesem Beispiel die zwei Gruppen `apt_hosts` und `rpm_hosts`, die dann jeweils die Mitglieder der aufgeführten einfachen Gruppen enthalten. Supergruppen können natürlich mit `:vars` genauso parametrisiert werden, wie Sie das von den normalen Gruppen bereits kennen. Testen wir, ob es auch funktioniert:

```
$ ansible-inventory --graph apt_hosts
@apt_hosts:
  |--@debian_hosts:
  |  |--debian
  |--@ubuntu_hosts:
  |  |--ubuntu
```


Die Verschachtelungstiefe ist übrigens nicht begrenzt; wenn es für Sie zweckdienlich ist, können Sie Gruppen von Gruppen von Gruppen von ... definieren.

12.3 »On the fly«-Inventorys erstellen mit »add_host«

Mitunter möchten Sie mit Ansible Target-Hosts managen, die nicht in einem Inventory verzeichnet sind – beispielsweise, weil die Systeme eben erst provisioniert wurden und Sie kein adäquates dynamisches Inventory-Handling zur Verfügung haben.

In solchen Situationen kann manchmal das `add_host`-Modul hilfreich sein, mit dem Sie Hosts und Gruppen zu einem temporären Inventory hinzufügen können, um in einem darauffolgenden Play mit diesen zu arbeiten.

Dieses Inventory befindet sich lediglich im Hauptspeicher und ist nach dem Playbook-Lauf wieder vergessen.

Im folgenden Beispiel steckt der Task im ersten Play zwei unserer schon vorhandenen Hosts unter neuem Namen in eine Gruppe namens `temp_group`. Das zweite Play arbeitet dann mit diesen Hosts:

```
---
- hosts: localhost
  tasks:
    - name: Hosts bekannt machen
      add_host:
        hostname: "{{item.ip}}"
        groups: temp_group
        ansible_become: yes
        ansible_ssh_common_args: >
          -o StrictHostKeyChecking=no
          -o UserKnownHostsFile=/dev/null
      with_items:
        - ip: 192.168.150.10
        - ip: 192.168.150.20

- hosts: temp_group
  tasks:
    - debug: msg="Hallo {{inventory_hostname}}"
    - command: head -1 /etc/shadow
```

Listing 12.2 »add_host.yml«

Um ganz sicher zu beweisen, dass unser vorhandenes statisches Inventory das Verhalten nicht beeinflusst, sorgen wir beim Aufruf des Playbooks für ein leeres Inventory:

```
$ ansible-playbook -i, add_host.yml
[WARNING]: provided hosts list is empty, only localhost is available.
Note that the implicit localhost does not match 'all'

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

TASK [Hosts bekanntmachen] *****************************************************
ok: [localhost] => (item={'ip': '192.168.150.10'})
ok: [localhost] => (item={'ip': '192.168.150.20'})

PLAY [temp_group] **************************************************************

TASK [debug] *******************************************************************
ok: [192.168.150.10] => {
    "msg": "Hallo 192.168.150.10"
}
ok: [192.168.150.20] => {
    "msg": "Hallo 192.168.150.20"
}

TASK [command] *****************************************************************
changed: [192.168.150.20]
changed: [192.168.150.10]

PLAY RECAP *********************************************************************
192.168.150.10             : ok=2    changed=1    unreachable=0    failed=0
192.168.150.20             : ok=2    changed=1    unreachable=0    failed=0
localhost                  : ok=1    changed=0    unreachable=0    failed=0
```

12.4 Dynamische Gruppen mit »group_by«

Ein weiteres, sehr mächtiges Feature von Ansible ist das `group_by`-Modul, mit dem Sie während eines Playbook-Laufs dynamische Gruppen bilden können. Wozu dient das?

Stellen Sie sich vor, Sie haben in Ihrem Inventory eine Gruppierung nach Standorten oder Themengebieten (Mailserver, Webserver etc.) vorgenommen, brauchen nun aber eine Gruppierung nach Betriebssystemen (Debian, SUSE ...).

Sofern es Ihnen genügt, dass diese neue Gruppierung nicht dauerhaft im Inventory verzeichnet ist, sondern nur dynamisch bei Bedarf zur Verfügung steht, kommen Sie mit `group_by` schnell zum Ziel. Im Prinzip können Sie alle bekannten Facts benutzen, um eine Gruppeneinteilung vorzunehmen.

Sehen Sie im Playbook aus Listing 12.3, wie im ersten Play eine Gruppierung nach Betriebssystemfamilien gebildet wird und in weiteren Plays die Hosts gruppenweise abgearbeitet werden:

```
---
- name: Gruppieren nach Systemfamilien
  hosts: all

  tasks:
    - group_by: key=os_{{ ansible_os_family }}


- name: Alle aus der Debian-Familie abarbeiten
  hosts: os_Debian

  tasks:
    - debug: msg="Hallo {{ inventory_hostname }}"


- name: Und nun der Rest
  hosts: all:!os_Debian

  tasks:
    - debug: msg="Hallo {{ inventory_hostname }}"
```

Listing 12.3 »group_by.yml«: dynamische Gruppierung nach Betriebssystemfamilien

Und so sieht der Ablauf aus:

```
$ ansible-playbook group_by.yml
[…]
PLAY [Alle aus der Debian-Familie abarbeiten] **********************************

TASK [debug] *******************************************************************
ok: [debian] => {
    "msg": "Hallo debian"
}
ok: [ubuntu] => {
    "msg": "Hallo ubuntu"
}

PLAY [Und nun der Rest] ********************************************************

TASK [debug] *******************************************************************
ok: [rocky] => {
```

```
    "msg": "Hallo rocky"
}
ok: [suse] => {
    "msg": "Hallo suse"
}
[…]
```

Natürlich müssen Sie die ganzen entstandenen Gruppen von Hosts nicht alle verwenden (es hätte ja auch noch `os_Suse` und `os_RedHat` gegeben). Ein `group_by` stellt Gruppen nur zur Verfügung – was Sie damit anfangen, ist allein Ihnen überlassen.

Listing 12.4 zeigt noch eine andere Idee – eine Gruppierung nach der Major-Kernelversion:

```
---
- name: Gruppieren nach Major-Kernelversion
  hosts: all
  tasks:
    - group_by: key=kernel_{{ ansible_kernel | regex_search('^\d+') }}


- name: Alle Hosts mit Kernel 4.x
  hosts: kernel_4
  tasks:
    - debug: msg="Hier läuft noch ein 4er-Kernel ({{ansible_kernel}})"

- name: Alle Hosts mit Kernel 5.x
  hosts: kernel_5
  tasks:
    - debug: msg="Hier läuft ein 5er-Kernel ({{ansible_kernel}})"
```

Listing 12.4 »group_by_kernelversion.yml«: Gruppierung nach der Kernelversion

Der Jinja-Filter `regex_search` ist vergleichbar mit einem `grep -o` auf der Kommandozeile: Nur der Teil des Inputs, der auf den regulären Ausdruck passt, wird zurückgegeben. Und da der Ausdruck `'^\d+'` einfach bedeutet »Finde eine oder mehr Ziffern am Anfang des Textes«, bekommen wir damit die Major-Version des Kernels geliefert und können sie zur Gruppeneinteilung verwenden:

```
$ ansible-playbook group_by_kernelversion.yml
[…]
PLAY [Alle Hosts mit Kernel 4.x] ***************************************

TASK [debug] ***********************************************************
ok: [rocky] => {
```



```
    "msg": "Hier läuft noch ein 4er-Kernel (4.18.0-348.2.1.el8_5.x86_64)"
}

PLAY [Alle Hosts mit Kernel 5.x] ***************************************

TASK [debug] ***********************************************************
ok: [debian] => {
    "msg": "Hier läuft ein 5er-Kernel (5.10.0-10-amd64)"
}
ok: [suse] => {
    "msg": "Hier läuft ein 5er-Kernel (5.3.18-150300.59.49-default)"
}
ok: [ubuntu] => {
    "msg": "Hier läuft ein 5er-Kernel (5.4.0-91-generic)"
}
[…]
```

12.5 Dynamische bzw. externe Inventorys

Unser wohlbekanntes statisches Inventory kommt in der Praxis mitunter an seine Grenzen. Wenn beispielsweise die Menge der Target Hosts stark fluktuiert und/oder diese sowieso schon an anderer Stelle inventarisiert sind, dann kann die (zusätzliche) Pflege eines statischen Ansible-Inventorys sehr aufwendig und fehleranfällig werden. Abhilfe schafft hier Ansibles Fähigkeit, *dynamische* bzw. *externe Inventorys* einzubinden. Dazu gibt es grundsätzlich zwei Methoden:

1. Inventory-Skripte (die klassische Methode)
2. Inventory-Plugins (die neuere Methode)

Bei beiden Methoden können Sie entweder auf bereits existierende Lösungen zurückgreifen oder selbst neue Lösungen entwickeln (sprich: programmieren). Der vielleicht entscheidende Unterschied ist, dass Sie ein Inventory-Skript in jeder beliebigen Programmiersprache verfassen können, ein Inventory-Plugin jedoch nur in Python.

Da ich persönlich seit über 20 Jahren alle Programmieraufgaben mit Perl erledige, zu Python jedoch nie einen rechten Zugang gefunden habe (es mag mich einfach nicht), werde ich für Methode 1 nur mit einem Beispiel in Perl dienen können. Methode 2 werden wir in der Praxis in Kapitel 13 betrachten; dort geht um Cloud-Umgebungen, in denen dynamische Inventorys naturgemäß relativ schnell zum Thema werden.

Der Einstiegspunkt der offiziellen Dokumentation in diese Thematik ist übrigens *http://docs.ansible.com/ansible/latest/user_guide/intro_dynamic_inventory.html*.

12.5.1 Beispiel: ein Inventory-Skript in Perl

Angenommen, Sie verwalten Ihre Hosts ohnehin schon in einer Datenbank. Ich habe hier einmal zwei völlig simple »Tabellen« in Form von CSV-Dateien erstellt (die erste Zeile definiert die Spaltennamen und damit für uns auch die Bedeutung). Die Tabellen sind nicht in Normalform, es sind keine verschachtelten Gruppen möglich, und das Modell hat bestimmt noch viele andere Unzulänglichkeiten, aber es verdeutlicht hoffentlich die Strategie:

```
host:user:become:method:password
debian:vagrant:yes:sudo:
rocky:vagrant:yes:su:vagrant
suse:vagrant:yes:sudo:
ubuntu:vagrant:yes:sudo:
```

Listing 12.5 »hosts.csv«

```
group:members
test_hosts:debian,rocky,suse,ubuntu
apt_hosts:debian,ubuntu
rpm_hosts:rocky,suse
```

Listing 12.6 »groups.csv«

Daraus müssen wir jetzt einen JSON-Output erzeugen, der im Wesentlichen der Ausgabe von `ansible-inventory --list` entspricht. Genauer ist das beschrieben unter: *http://docs.ansible.com/ansible/latest/dev_guide/developing_inventory.html#developing-inventory-scripts*.

Das folgende Perl-Skript aus Listing 12.7 ist sogar etwas aufwendiger als eigentlich nötig, da die CSV-Datei mit einem »Datenbanktreiber« verarbeitet wird. Dadurch würde sich am Skript aber auch nicht viel ändern, wenn tatsächlich eine echte SQL-Datenbank eingesetzt würde.

```
#!/usr/bin/perl
use strict;
use warnings;
use DBI;
use JSON;
use FindBin;
use Getopt::Long;

my %opts;

# Die zwei möglichen Optionen:
GetOptions(\%opts, "list", "host=s");
```



```
# Verbindung zur Datenquelle ( = CSV-Datei ) herstellen
my $dbh = DBI->connect ("dbi:CSV:f_dir=${FindBin::Bin}", undef, undef, {
    f_encoding       => "utf8",
    csv_eol          => "\n",
    csv_sep_char     => ":",
});
$dbh->{csv_tables}{hosts}   = { file => "hosts.csv"  };
$dbh->{csv_tables}{groups}  = { file => "groups.csv"  };

# Grundeinstellungen:
my %host_defaults = (
    ansible_ssh_common_args =>
    "-o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null",
    ansible_python_interpreter => "/usr/bin/python3"
);
my $hosts = {};

# Abfrage aller Hosts:
my $sth = $dbh->prepare(
    "SELECT host, user, become, method, password FROM hosts"
);
$sth->execute;

while (my ($host, $user, $become, $method, $password)
       = $sth->fetchrow_array) {
    $hosts->{$host} = {
        %host_defaults,
        ansible_user => $user,
    };
    if ($become eq 'yes') {
        $hosts->{$host}{ansible_become} = 'yes';
        $hosts->{$host}{ansible_become_method} = $method;
    };
    if ($password) {
        $hosts->{$host}{ansible_become_pass} = $password;
    };
}

if (defined $opts{host}) { # Aufruf: --host <HOSTNAME>
    my $json = JSON->new;
    print $json->pretty->encode( $hosts->{ $opts{host} } );
    exit;
}
```

```
# Ansonsten muss es wohl der Aufruf --list sein:
exit unless defined $opts{list};

my $info   = {};
my $groups = {};

# Abfrage aller Gruppen:
$sth = $dbh->prepare("SELECT group, members FROM groups");
$sth->execute;

while (my ($group, $members) = $sth->fetchrow_array) {
    my @members = split(/,/, $members);
    $groups->{$group} = [@members];
}

$info = $groups;
$info->{_meta}{hostvars} = $hosts;

my $json = JSON->new;
print $json->pretty->encode( $info  );
```

Listing 12.7 »dyninv.pl«: ein examplarisches Inventory-Skript in Perl

Legen Sie alle Dateien nebeneinander im Projektordner ab. Bevor Sie das Programm wie beabsichtigt nutzen können, müssen Sie aber noch zwei Dinge erledigen:

1. Die benötigten Perl-Module installieren. Auf unserem Debian-Control-Host-System reicht dazu ein:

   # **sudo apt install libdbd-csv-perl libjson-perl**

2. Das Programm ausführbar machen:

   $ **chmod +x dyninv.pl**

Nun sollten Sie es starten können:

```
$ ./dyninv.pl --list
[…]

$ ./dyninv.pl --host debian
[…]
```

Entscheidend ist aber, dass es auch im Ansible-Kontext wie gewünscht funktioniert. Sie verwenden ein ausführbares(!) Inventory-Skript, indem Sie es einfach als Inventory-Quelle angeben; z. B. mit dem Schalter `-i`. Versuchen Sie es:


```
$ ansible-inventory -i ./dyninv.pl --list
[…]

$ ansible all -i ./dyninv.pl -m ping
[…]

$ ansible all -i ./dyninv.pl -a "head -1 /etc/shadow"
[…]
```

12.5.2 Verwenden von Inventory-Plugins

Wenn Sie ein Inventory spezifizieren (z. B. mit dem `-i`-Schalter), so gibt es noch die weitere Möglichkeit, die Konfigurationsdatei eines *Inventory-Plugins* anzugeben. Die eben beschriebene Inventory-Skript-Variante ist intern letztlich auch »nur« durch ein Plugin realisiert.

Die Konfigurationsdatei muss Ansible-typisch im YAML-Format vorliegen, die konkrete Syntax ist jeweils in der Dokumentation des jeweiligen Plugins beschrieben. Die allgemeine, offizielle Dokumentation zum Thema ist *http://docs.ansible.com/ansible/latest/plugins/inventory.html*, und einen Überblick über alle auf Ihrem System verfügbaren Inventory-Plugins gibt Ihnen der folgende Befehl:

```
$ ansible-doc -t inventory -l
```

Wie bereits erwähnt, möchte ich an dieser Stelle aber auf Kapitel 13 verweisen, in dem wir uns einige Cloud-Umgebungen mit ihren zugehörigen Inventory-Plugins genauer anschauen werden.