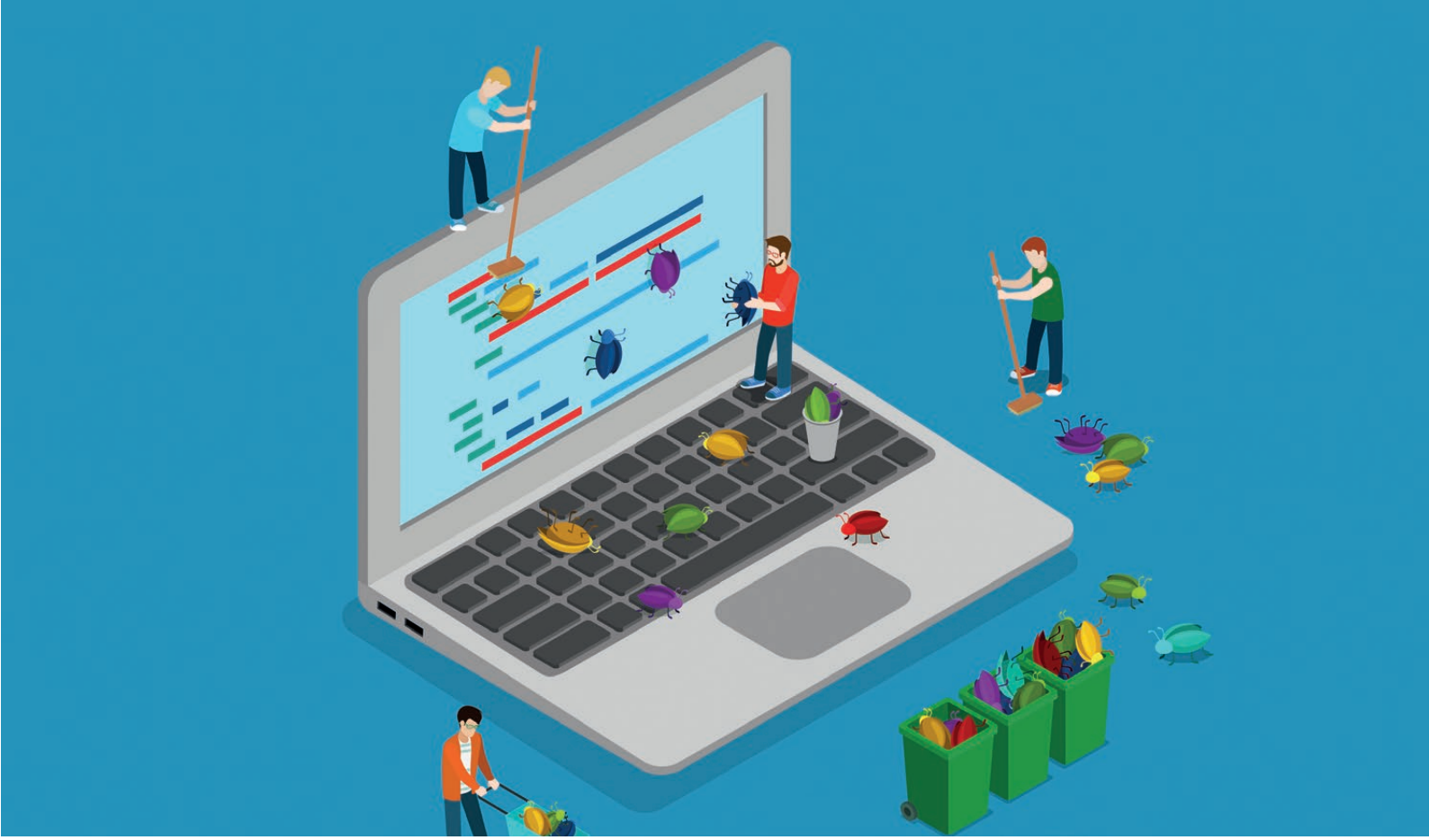

Automatisierte Qualitätssicherung von Ansible Playbooks

Sandra Parsick

Dieser Text wurde im Englischen im Cloud Report 01/2020 veröffentlicht

Wenn Serverinstanzen in der Cloud provisioniert werden, werden sie selten manuell aufgesetzt, sondern mithilfe von Provisionierungsskripten automatisiert. Diese beschreiben, wie der Server aufgesetzt werden soll. Dabei handelt es sich um normalen Code, so wie wir es von der normalen Softwareentwicklung kennen, nur dass dieser Code auf die Domäne Infrastruktur spezialisiert ist. In der Softwareentwicklung haben sich statische Code-Analyse („linten") und automatisierte Tests als Mittel für gute Wartbarkeit des Produktionscodes etabliert. Warum diese gute Praxis nicht auch für Infrastructure-as-Code (IaC) anwenden? Dieser Artikel beschreibt anhand des Provisionierungswerkzeugs Ansible, wie eine automatisierte Qualitätssicherung für Provisionierungsskripte aussehen kann.

Ansible beschreibt sich selbst als Werkzeug für das Konfigurationsmanagement, für die Verteilung von Software und für die Ausführung von Ad-hoc-Kommandos *[1]*. Ansible ist in Python geschrieben. Um es zu benutzen, muss man nicht zwangsläufig Python-Kenntnisse haben, da die Ansible-Skripte in YAML geschrieben sind. YAML ist eine Abstraktion von JSON mit dem Unterschied, dass sie besser lesbar ist *[2]*. Diese Ansible-Skripte werden „Playbooks" genannt.

Damit Ansible eingesetzt werden kann, muss auf den Zielmaschinen Python installiert und ein Zugriff auf diese Maschinen über SSH möglich sein. Ansible ist nur auf der Maschine zu installieren, von der die Playbooks ausgeführt werden („Host-Maschine"); sie muss ein Linux-System sein. Für Windows auf den Zielmaschinen bietet Ansible eine rudimentäre Unterstützung an. Wenn die in YAML geschriebenen Playbooks ausgeführt werden, übersetzt Ansible diese in Python-Skripte und führt sie auf den Zielmaschinen aus. Anschließend werden sie wieder von den Zielmaschinen entfernt. Ein typisches Ansible-Playbook sieht wie in *Listing 1* gezeigt aus. Für einen tieferen Einstieg in Ansible empfiehlt sich mein Artikel „Ansible für Entwickler" *[3]*.

Funktionsweise einer QA-Pipeline für Ansible-Playbooks

In der Softwareentwicklung werden statische Code-Analyse und automatisierte Tests kontinuierlich in einer Pipeline auf einem CI-Server durchgeführt. Dieses Konzept soll für Ansible-Playbooks wiederverwendet werden.

Als Erstes legen wir die Provisionierungsskripte in ein Versionskontrollsystem ab. Änderungen an den Skripten, die im Versionskontrollsystem eingecheckt werden, triggern eine Pipeline an. Eine Pipeline für Provisionierungsskripte durchläuft mehrere Schritte *(siehe Abbildung 1)*: Zuerst überprüft ein Lint den Code daraufhin, ob dieser syntaktisch richtig ist und Best Practices folgt. Wenn der Lint nichts zu beanstanden hat, wird eine Testumgebung vorbereitet. Gegen diese Testumgebungen laufen die Provisionierungsskripte. Wenn sie ohne Fehler durchgelaufen sind, werden Tests ausgeführt, um zu überprüfen, ob alles so konfiguriert wurde wie erwartet. Am Ende wird die Testumgebung wieder abgebaut.

Im Folgenden werden die einzelnen Schritte genauer vorgestellt und gezeigt, wie sie in einer Pipeline eingebunden werden. Das wird mithilfe eines Git-Repository demonstriert, das ein Ansible-Playbook enthält, Skripte, um die Testumgebung aufzubauen, die Tests und eine Beschreibung der Pipeline. Das Ansible-Playbook wird ein OpenJDK 8 und eine Tomcat-Instanz installieren *(siehe Listing 2)*. Das komplette Projekt ist auf GitHub zu finden *[4]*. Als Startpunkt dient das Git-Repository, das erstmal nur das Ansible-Playbook erhält.

```
- hosts: ansible-test-instance
  vars:
  tomcat_version: 9.0.27
  tomcat_base_name: apache-tomcat-{{ tomcat_version }}
  #catalina_opts: "-Dkey=value"
  tasks:
  - name: install java
    apt:
  name: openjdk-8-jdk
  state: present
    become: yes
    become_method: sudo
```

Listing 1

```
.
├── LICENSE
├── README.md
└── setup-tomcat.yml
```

Listing 2

Statische Code-Analyse mit ansible-lint

Zuerst überprüft ein Lint, ob die Ansible-Playbooks syntaktisch korrekt sind und ob sie nach Best-Practices-Regeln geschrieben sind. Für Ansible-Playbook gibt es den Lint „*ansible-lint*" *[5]*. Es wird auf der CLI ausgeführt *(siehe Listing 3)*.

In diesem Beispiel findet ansible-lint mehrere Regelverstöße. Sollen einzelne Regeln nicht gecheckt werden, gibt es zwei Möglichkeiten, diese zu exkludieren. Einmal können die Regeln global für das ganze Projekt ausgeschaltet werden oder für einzelne Fälle (sogenannte „false-positive" Fälle). Für die globale Einstellung legen wir eine .ansible-lint-Datei im Root-Verzeichnis des Projektes ab *(siehe Listing 4)*.

Abbildung 1: Alle Schritte einer Pipeline (Quelle: Sandra Parsick)

```
$ ansible-lint setup-tomcat.yml
[502] All tasks should be named
setup-tomcat.yml:30
Task/Handler: file name=/opt __file__=setup-tomcat.yml __line__=31 mode=511
owner=tomcat group=tomcat

[502] All tasks should be named
setup-tomcat.yml:57
Task/Handler: find patterns=*.sh paths=/opt/{{ tomcat_base_name }}/bin
```

Listing 3

```
.
├── .ansible-lint
├── LICENSE
├── README.md
└── setup-tomcat.yml
```

Listing 4

```
skip_list:
- skip_this_tag
- and_this_one_too
- skip_this_id
- '401
```

Listing 5

In dieser Konfigurationsdatei pflegen wir eine *exclude*-Liste *(siehe Listing 5)*. Darin können wir noch weitere Verhaltensweisen konfigurieren, wie zum Beispiel, in welchem Pfad sie abgelegt sind. Mehr Informationen dazu auf der Projektseite *[4]*.

Möchten wir false-positive Fälle vom Check herausnehmen, dann hinterlassen wir im Ansible-Playbook einen entsprechenden Kommentar dazu *(siehe Listing 6)*.

Fehlen uns noch weitere Regeln, dann können wir diese mithilfe von Python-Skripten selbst definieren *(siehe auch die Dokumentation [5])*.

Testumgebung aufsetzen und abbauen mit Terraform

Nachdem der Lint erfolgreich durchgelaufen ist, soll die Testumgebung für die Ansible-Playbooks aufgebaut werden. Das wird mithilfe von Terraform *[6]* und der Hetzner Cloud *[7]* gemacht. Terraform hilft uns, Cloud-Infrastruktur mit Code zu provisionieren.

Bevor wir mit dem Terraform-Skript loslegen können, müssen wir im Hetzner-Cloud-Konto einen Public-SSH-Key ablegen und ein API-Token generieren. Der Public-SSH-Key wird später in die anzulegende Server-Instanz abgelegt, damit Ansible sich mit dieser Instanz verbinden kann.

```
- file: # noqa 502
    name: /opt
    mode: 0777
    owner: tomcat
    group: tomcat
  become: yes
  become_method: sudo
```

Listing 6

Mithilfe von Terraform beschreiben wir, welchen Server-Typ wir haben wollen, welches Betriebssystem, in welchem Standort die Server-Instanz gehostet und was als Grund-Setup provisioniert werden soll. Dazu legen wir im Root-Verzeichnis des Projekts eine *testinfrastrucuture.tf*-Datei an *(siehe Listing 8)*.

Als Grund-Setup geben wir an, welcher Public-SSH-Key auf dem Server abgelegt werden soll (*ssh_keys*) und dass Python installiert wird (*provisioner remote-exec*) *(siehe Listing 9)*. Der Public-SSH-Key und Python werden benötigt, damit später Ansible seine Skripte auf diesem Server ausführen kann.

Da die Test-Server-Instanz in der Hetzner-Cloud betrieben werden soll, muss Terraform das benötigte Provider-Plug-in nachinstallieren. Dazu rufen wir „*terraform init*" im Ordner auf, in dem *testinfrastructure.tf* liegt. Dann ist alles vorbereitet, um die Server-Instanz zu provisionieren.

Wir müssen dem apply-Befehl die Variable hcloud_token mit dem API-Token, den wir vorher in der Hetzner-Cloud-Konsole generiert haben, mitgeben *(siehe Listing 10)*. Sobald der Server zur Verfügung steht, können wir die Ansible Playbooks und die Tests gegen diesen Server ausführen. Unabhängig vom Erfolg der Playbooks oder der Tests wird die Serverinstanz mithilfe von Terraform wieder abgebaut. Auch beim destroy-Befehl muss das API-Token mitgegeben werden *(siehe Listing 11)*.

Ansible-Playbooks ausführen

Nachdem wir die Server-Instanz mit Terraform erstellt haben, führen wir Ansible-Playbooks gegen diese Server-Instanz aus. In einer klassischen Infrastruktur hätte die Server-Instanz eine feste IP-Adresse und wir hätten diese IP-Adresse in Ansibles sogenanntes „statisches Inventory" eingetragen *(siehe Listing 12)*, damit Ansible weiß, mit welchem Server es sich verbinden soll.

Da in der Cloud die Server bei jedem Bereitstellen eine neue IP-Adresse zugeteilt bekommen, können wir in diesem Fall nicht das statische Inventory benutzen. Da wir gegen die Hetzner-Cloud gehen, verwenden wir das Ansible-Inventory-Plug-in „*hcloud*". Dazu muss ein *inventory*-Ordner mit der Datei *test.hcloud.yml* erstellt werden *(siehe Listing 13)*. Beim Dateinamen ist das Suffix „*hcloud.yml*" wichtig *(siehe Listing 14)*.

Im Anschluss müssen wir den richtigen Servernamen, den wir vorher im Terraform-Skript definiert haben, im Playbook unter „*hosts*" eintragen *(siehe Listing 15)*.

Beim Ausführen der Playbooks müssen wir noch darauf achten, dass wir den richtigen Private-SSH-Key mitgeben und dass das API-

```
from ansiblelint import AnsibleLintRule
class DeprecatedVariableRule(AnsibleLintRule):
  id = 'ANSIBLE0001'
  shortdesc = 'Deprecated variable declarations'
  description = 'Check for lines that have old style ${var}'
  + 'declarations'
  tags = { 'deprecated' }
  def match(self, file, line):
     return '${' in line
```

Listing 7

Token in der System-Environment-Variablen *HCLOUD_TOKEN* definiert ist *(siehe Listing 16)*.

Das API-Token können wir auch in der Inventory-Datei *test.hcloud.yml* definieren. Da diese Datei aber in einem Version-Control-System abgelegt wird, ist davon abzuraten, da keine Credentials in einem VCS gespeichert werden sollten.

```
.
└──
├──.ansible-lint
├──LICENSE
├──README.md
├──setup-tomcat.yml
├──terraform.tfvars
└──testinfrastructure.tf
```

Listing 8

```
# testinfrastructure.tf
variable "hcloud_token" {}
variable "ssh_key" {}
# Configure the Hetzner Cloud Provider
provider "hcloud" {
  token = var.hcloud_token
}
# Create a server
resource "hcloud_server" "ansible-tests" {
  name = "ansible-tests"
  image = "ubuntu-18.04"
  server_type = "cx11"
  location = "nbg1"
  ssh_keys = [
    "ansible-test-infrastructure"
  ]
  provisioner "remote-exec" {
    inline = [
    "while fuser /var/lib/apt/lists/lock >/dev/null 2>&1; do sleep 1; done",
    "apt-get -qq update -y",
    "apt-get -qq install python -y",
  ]
  connection {
    type = "ssh"
    user = "root"
    private_key = file(var.ssh_key)
    host = hcloud_server.ansible-tests.ipv4_address
  }
 }
}
```

Listing 9

```
terraform apply -var="hcloud_token=…"
```

Listing 10

```
terraform destroy -var="hcloud_token=…"
```

Listing 11

Funktionale Tests mit Testinfra

Nachdem die Testumgebung aufgesetzt wurde und die Playbooks erfolgreich durchliefen, sollen noch Tests ausgeführt werden, die überprüfen, ob das OpenJDK-Package installiert wurde und ob die Datei */opt/tomcat/bin/catalina.sh* auf dem Server existiert.

Es existieren einige Testframeworks für Provisionierungswerkzeuge, zum Beispiel ServerSpec *[8]*, Goss *[9]* und Testinfra *[10]*. Der Hauptunterschied zwischen den Testframeworks ist, in welcher Syntax die Tests geschrieben werden. Bei ServerSpec werden die Tests in Ruby-Syntax geschrieben, bei Goss in YAML-Syntax und bei Testinfra in Python-Syntax. Die Arbeitsweise der Testframeworks ist gleich. Sie verbinden sich auf einen Server, der vorher mit Provisionierungswerkzeugen provisioniert wurde, und prüfen, ob die Serverprovisionierung den Testbeschreibungen entspricht.

Hier werden die Tests mit Testinfra geschrieben. Dazu legen wir im Root-Verzeichnis des Projekts den Ordner „*tests*" an. Dort werden

```
#statische Inventory
[ansible-tests]
78.47.150.245
```

Listing 12

```
# aktuelle Projektstruktur
.
├── .ansible-lint
├── ansible.cfg
├── inventory
│   └── test.hcloud.yml
├── LICENSE
├── README.md
├── setup-tomcat.yml
├── terraform.tfvars
└── testinfrastructure.tf
```

Listing 13

```
# test.hcloud.yml
plugin: hcloud
```

Listing 14

```
Ansible-Playbook Snippet
- hosts: ansible-test-instance
```

Listing 15

```
$ export HCLOUD_TOKEN=...
$ ansible-playbook --private-key=/home/sparsick/.ssh/id_hetzner_ansible_test -i
inventory/test.hcloud.yml setup-tomcat.yml
```

Listing 16

```
.
├── .ansible-lint
├── ansible.cfg
├── inventory
│   └── test.hcloud.yml
├── LICENSE
├── README.md
├── setup-tomcat.yml
├── terraform.tfvars
├── testinfrastructure.tf
└── tests
    └── test_tomcat.py
```

Listing 17

```
# test_tomcat.py
def test_openjdk_is_installed(host):
  openjdk = host.package("openjdk-8-jdk")
  assert openjdk.is_installed

def test_tomcat_catalina_script_exist(host):
  assert host.file("/opt/tomcat/bin/catalina.sh").exists
```

Listing 18

die Tests abgelegt *(siehe Listing 17)*. In *test_tomcat.py* schreiben wir die Tests *(siehe Listing 18)*.

Testinfra bringt fertige Module mit, die die Testbeschreibung vereinfachen. In diesen Tests benutzen wir zum Beispiel *host.package*, um den Package-Manager abfragen zu können, oder *host.file*, um die Existenz einer bestimmten Datei zu testen.

Testinfra unterstützt verschiedene Möglichkeiten, um sich mit dem Server zu verbinden. Eine Möglichkeit ist, die Verbindungskonfiguration von Ansible wiederzuverwenden. Da Ansible hier ein dynamisches Inventory benutzt und Testinfra nicht alle Informationen aus diesem dynamischen Inventory lesen kann, müssen wir einige Konfigurationen explizit in der Ansible-Konfigurationsdatei *ansible.cfg* eintragen *(siehe Listing 19)*. Diese Datei wird im Root-Verzeichnis des Projektes abgelegt. Dann können wir die Tests ausführen *(siehe Listing 20)*.

Zusammenführung in einer Pipeline

Nachdem jeder Schritt in der Pipeline einzeln betrachtet wurde, sollen die Schritte in eine Pipeline im CI-Server Jenkins zusammen-

```
# ansible.cfg
[defaults]
remote_user=root
private_key_file = /home/sparsick/.ssh/id_hetzner_ansible_test
```

Listing 19

```
$ py.test --connection=ansible --ansible-inventory=inventory/test.hcloud.yml --force
-ansible -v tests/*.py
============================================================
===================== test session starts ==================
============================================================
platform linux2 -- Python 2.7.15+, pytest-3.6.3, py-1.5.4, pluggy-0.13.0 --
/usr/bin/python
cachedir: .pytest_cache
rootdir: /home/sparsick/dev/workspace/ansible-testing-article, inifile:
plugins: testinfra-3.2.0
collected 2 items
tests/test_tomcat.py::test_openjdk_is_installed[ansible://ansible-test-instance]
PASSED
[ 50%]
tests/test_tomcat.py::test_tomcat_catalina_script_exist[ansible://ansible-test-
instance] PASSED
[100%]
============================================================
================== 2 passed in 11.52 seconds ===============
============================================================
```

Listing 20

```
# Jenkinsfile
pipeline {
  agent any
  environment {
    HCLOUD_TOKEN = credentials('hcloud-token')
  }
  stages {
    stage('Lint Code') {
      steps {
        sh 'ansible-lint setup-tomcat.yml'
      }
    }
    stage('Prepare test environment') {
      steps {
        sh 'terraform init'
        sh 'terraform apply -auto-approve -var="hcloud_token=$   {HCLOUD_TOKEN}"'
      }
    }
    stage('Run Playbooks') {
      steps {
        sh 'ansible-playbook -i inventory/test.hcloud.yml setup-tomcat.yml'
      }
    }
    stage('Run Tests') {
      steps {
        sh 'py.test --connection=ansible -ansible -inventory=inventory/test.hcloud.yml --force-ansible -v tests/*.py'
      }
    }
}
post {
  always {
    sh 'terraform destroy -auto-approve -var="hcloud_token=${HCLOUD_TOKEN}"'
  }
 }
}
```

Listing 21

geführt werden. Die Pipeline beschreiben wir in einem Jenkins-File. Diese Jenkins-File beschreibt vier Stages und eine Post-Action. Je eine Stage für den Code-Check, das Aufbauen der Testumgebung und das Ausführen der Playbooks und die letzte Stage für die Ausführung. In der Post-Action wird die Testumgebung abgebaut, egal, ob in den Stages Fehler auftraten *(siehe Listing 21)*. Damit diese Pipeline auch funktioniert, müssen wir in Jenkins das API-Token der Hetzner-Cloud hinterlegen. Damit das Token nicht im Klartext gespeichert wird, legen wir es im Bereich „*Credentials*" als *Secret Text* ab und vergeben eine ID, die dann per *credential*-Methode im Jenkins-File wieder abgerufen werden kann (hier: *hcloud-token*).

Vereinfachung für Ansible Role

Für Ansible Role, einer Strukturierungsmöglichkeit in Ansible, um Playbooks über mehrere Projekte wiederzuverwenden, gibt es eine Vereinfachung für diese vorgestellte Pipeline. Bei Ansible Role können wir mithilfe von Molecule *[11]* die komplette Pipeline und die vorgestellten Werkzeuge in einem Rutsch konfigurieren. Wir benötigen dann auch nur einen Befehl (*molecule test*), um die komplette Pipeline auszuführen. Eine sehr gute Einführung in Molecule gibt der Blog Post „Test-driven infrastructure development with Ansible & Molecule" *[12]* von Jonas Hecht.

Weiterführende Informationen

[1] *https://docs.ansible.com/*
[2] *https://en.wikipedia.org/wiki/YAML*
[3] *https://www.sandra-parsick.de/publication/ansible-fuer-dev/*
[4] *https://github.com/sparsick/ansible-testing-article/tree/cloudreport19*
[5] *https://github.com/ansible/ansible-lint*
[6] *https://www.terraform.io/*
[7] *https://www.hetzner.com/cloud*
[8] *https://serverspec.org/*
[9] *https://github.com/aelsabbahy/goss*
[10] *https://github.com/philpep/testinfra*
[11] *https://github.com/ansible/molecule*
[12] *https://blog.codecentric.de/en/2018/12/test-driven-infrastructure-ansible-molecule/*

Sandra Parsick
mail@sandra-parsick.de

Sandra Parsick ist als freiberufliche Softwareentwicklerin und Consultant im Java-Umfeld tätig. Seit 2008 beschäftigt sie sich mit agiler Softwareentwicklung in verschiedenen Rollen. Ihre Schwerpunkte liegen im Bereich der Java-Enterprise-Anwendungen, agilen Methoden, Software Craftsmanship und in der Automatisierung von Softwareentwicklungsprozessen. Darüber schreibt sie gerne Artikel und spricht gerne auf Konferenzen. In ihrer Freizeit engagiert sich Sandra Parsick in der Softwerkskammer Ruhrgebiet, einer Regionalgruppe der Software Craftmanship Community im deutschsprachigen Raum. Seit 2019 ist sie Mitglied im Oracle Groundbreaker Ambassador Programm.