Automated Quality Assurance for Ansible Playbooks

When server instances are provisioned in the cloud, they are rarely set up manually, but automated using provisioning scripts. The provisioning scripts describe how the server should be set up. This is normal code, as we know it from normal software development, except that this code is specialized on the domain infrastructure. In software development, static code analysis ("linting") and automated tests have established themselves as means for good maintainability of the production code. Why not apply this good practice to "Infrastructure as Code" as well? This article uses the provisioning tool Ansible to describe how automated quality assurance for provisioning scripts can look like.

Brief Introduction to Ansible

Ansible describes itself as a tool for configuration management, software distribution, and for executing ad hoc commands[1]. Ansible is written in Python. To use it, the developer does not necessarily need to have Python knowledge, since the Ansible scripts are written in YAML. YAML is an abstraction of JSON with the difference that it is more readable[2]. The ansible scripts are called "Playbooks".

In order to use Ansible, Python must be installed on the target machines and access to these machines must be possible via SSH. Ansible must only be installed on the machine running the Playbooks ("host machine"); it must be a Linux system. For target machines running Windows, Ansible provides rudimentary support. When the Playbooks written in YAML are executed, Ansible translates them into Python scripts and executes them on the target machines. They are then removed from the target machines. A typical Ansible Playbook looks like this (listing 1):

How a QA Pipeline Works for Ansible Playbooks

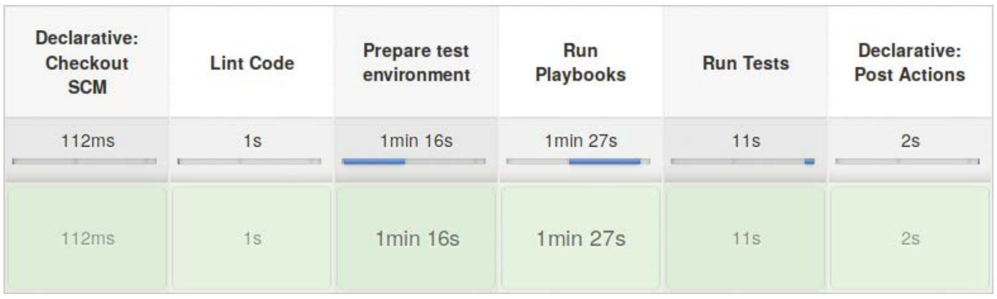
In software development, static code analysis and automated tests are performed continuously in a pipeline on a CI server. This concept is to be reused for Ansible Playbooks.

First, the developer stores the provisioning scripts in a version control system. Changes to the scripts that are checked in to the version control system trigger a pipeline. A pipeline for provisioning scripts runs through several steps (see also figure 1): First, a lint checks the code to see whether it is syntactically correct and whether it follows best practice. If the lint has nothing to complain about, a test environment is prepared. The provisioning scripts run against these test environments. If they have run without errors, tests are run to verify that everything has been configured as expected. At the end, the test environment is destroyed again.

```
- 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: Ansible Playbook: For a deeper introduction to Ansible, we recommend the author's article "Ansible für Entwickler"[3].



Declarative: Checkout SCM	Lint Code	Prepare test environment	Run Playbooks	Run Tests	Declarative: Post Actions
112ms	1s	1min 16s	1min 27s	11s	2s
112ms	1s	1min 16s	1min 27s	11s	2s

Figure 1. All steps in a pipeline

Lint are tools that perform static code analysis.

In the following, the individual steps are presented in more detail and how they are then integrated into a pipeline. This is demonstrated using a git repository that includes an Ansible playbook, scripts to build the test environment, the tests, and a description of the pipeline. The Ansible playbook will install an OpenJDK 8 and a Tomcat instance. The complete project can be found on Github[4].

The starting point is the Git repository, which initially only receives the Ansible Playbook.

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

Static code analysis with ansible-lint

First, a lint checks whether the Ansible Playbooks are syntactically correct and whether they are written according to best practice rules. For Ansible Playbooks there is the lint `ansible-lint`[5]. It is executed on the CLI (listing 2):

In this example, `ansible-lint` finds several rule violations. If single rules should not be checked, there are two ways to exclude them. The rule can be switched off either globally for the whole project, or for individual cases (so-called false-positive cases). For the global setting a `.ansible-lint` file is placed in the root directory of the project.

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

```
$ 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 2

In this configuration file you maintain an `exclude` list:

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

In this configuration file further behavior can be configured, e.g. in which path they are stored. More information can be found on the project page[4].

If the developer wants to exclude false-positive cases from the check, she leaves a comment in the Ansible Playbook.

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

If the developer lacks further rules, then she can define further rules herself with the help of Python scripts (see also documentation[5]). Listing 3.

Setting up and Destroying the Test Environment with Terraform

After the lint has successfully passed, the test environment for the Ansible Playbooks should be built. This is done with the help of Terraform[6] and the Hetzner Cloud[7]. Terraform helps developers to provision cloud infrastructure with code.

Before the developer can get started with the Terraform script, she must store a public SSH key in the Hetzner Cloud account and generate an API token. The Public SSH Key is later stored in the server instance to be created, so that Ansible can connect to this instance.

With the help of Terraform, the developer describes which server type she wants, which operating system, in which location the server instance is hosted and what is to be provisioned as the basic setup. For this purpose, the developer creates a `testinfrastrucuture.tf` file in the root directory of the project (listing 4).

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

As a basic setup, the developer specifies which public SSH key should be stored on the server (`ssh_keys`) and that Python should be installed (`provisioner remote-exec`). The public SSH key and Python are needed so that later Ansible can execute its scripts on this server.

Since the test server instance is to be operated in the Hetzner Cloud, Terraform must install the required provider plug-in. The developer calls `terraform init` in the folder containing `testinfrastructure.tf`.

Then everything is ready to provision the server instance.

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

The developer must give the `apply` command the variable `hcloud_token` with the API token, which the developer generated before in the Hetzner Cloud console.

As soon as the server is available, the developer can execute the Ansible Playbooks and the tests against this server. Regardless of the success of the Playbooks or the tests, the server instance is destroyed with the help of Terraform. The `destroy` command also requires the API token.

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

```python
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 3

```
# 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 4: testinfrastructure.tf

Running Ansible Playbooks

After the developer has created the server instance with Terraform, she executes the Ansible Playbook against this server instance. In a classical infrastructure, the server instance would have a fixed IP address and the developer would have entered this IP address in the Ansibles static inventory so that Ansible knows which server to connect to *static Inventory*.

```
[ansible-tests]
78.47.150.245
```

Since the servers in the cloud are assigned a new IP address each time they are deployed, the developer cannot use the static inventory in this case. Since the developer goes against the Hetzner Cloud, she uses the Ansible Inventory Plugin `hcloud`. Therefore an `inventory` folder with the file `test.hcloud.yml` has to be created.

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

The suffix `hcloud.yml` is important for the file name. *test.hcloud.yml*

```
plugin: hcloud
```

The developer then has to enter the correct server name in the Playbook under `hosts`, which she previously defined in the terraform script. *Ansible-Playbook Snippet*

```
- hosts: ansible-test-instance
```

When running the Playbooks the developer has to make sure that she gives the correct Private SSH key and that the API token is defined in the system environment variable `HCLOUD_TOKEN`.

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

The API token can also be defined by the developer in the inventory file `test.hcloud.yml`. However, since this file is stored in a version control system, it is not advisable to do this, since no credential should be stored in a VCS.

Functional Tests with Testinfra

After the test environment has been set up and the Playbooks have passed successfully, tests should be run to check if the openjdk package has been installed and if the file `/opt/tomcat/bin/catalina.sh` exists on the server.

There are some test frameworks for provisioning tools such as ServerSpec[8], Goss[9] and Testinfra[10]. The main difference between the test frameworks is the syntax in which the tests are written. For ServerSpec the tests are described in Ruby syntax, for Goss in YAML syntax and for testinfra in Python syntax.

The way the test frameworks work is the same. They connect to a server that was previously provisioned with provisioning tools and check whether the server provisioning corresponds to the test descriptions.

Here, the tests are written with Testinfra. To do this, the developer creates the `tests` folder in the root directory of the project. This is where the tests are stored.

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

The developer writes the tests in `test_tomcat.py`. (listing 5)

Testinfra comes with ready-made modules that simplify the test description. In these tests the developer uses e.g. `host.package` to query the Package Manager or `host.file` to test the existence of a certain file.

Testinfra supports several ways to connect to the server. One way is to reuse Ansible's connection configuration. Since Ansible uses a dynamic inventory here, and Testinfra cannot read all the information from that dynamic inventory, the developer must explicitly enter some configurations in the Ansible configuration file `ansible.cfg`. Listing 6.

This file is stored in the root directory of the project. The developer can then run the tests. (listing 7)

```
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 5: test_tomcat.py

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

Listing 6: ansible.cfg

```
$ 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- in-
stance] PASSED
[100%]

========================================================================================
=
=================== 2 passed in 11.52 seconds
========================================================================================
=
===================
```

Listing 7

```
#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 8: Jenkinsfile

Pipeline Integration

After each step in the pipeline has been considered individually, the steps are to be merged into a pipeline in CI-Server Jenkins.

The pipeline is described by the developer in a Jenkins file. This Jenkins file describes four stages and one post-action. One stage each for the code check, build test environment and execute playbooks and the last stage for the execution. In the post-action, the test environment is dismantled, regardless of whether errors occurred in the stages (listing 8).

In order for this pipeline to work, the developer must deposit the API token of the Hetzner Cloud in the Jenkins. So that the token is not stored in plain text, the developer stores it in the `Credential` area as `Secret Text` and assigns an ID, which he can then retrieve using the `credential` method in the Jenkins file (here: `hcloud-token`).

Simplification for Ansible Role

For Ansible Role, a structuring option in Ansible to reuse Playbooks across multiple projects, there is a simplification for this presented pipeline. With Ansible Role, the developer can use Molecule to configure the complete pipeline and the presented tools in one go. She then only needs one command (`molecule test`) to execute the complete pipeline. A very good introduction to Molecule is given in the blog post "Test-driven infrastructure development with Ansible & Molecule"[11,12] by Jonas Hecht.

Conclusion

This article provides an overview of how to build a quality assurance pipeline for Infrastructure as Code.

Sources:
- 1. https://docs.ansible.com/
- 2. https://en.wikipedia.org/wiki/YAML
- 3. https://www.sandra-parsick.de/publication/ansible-fuer-dev/ (German)
- 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

Sandra Parsick works as a freelance software developer and consultant in the Java environment.
Since 2008 she is working with agile software development in different roles. Her focus is on Java enterprise applications, agile methods, software craftsmanship and the automation of software development processes.
She enjoys writing articles and speaking at conferences.
In her spare time Sandra Parsick is involved in the Softwerkskammer Ruhrgebiet, a regional group of the Software Craftmanship Community in the German speaking area. Since 2019 she is a member of the Oracle Groundbreaker Ambassador Program.

E-Mail: mail@sandra-parsick.de
Twitter: @SandraParsick
Homepage: https://www.sandra-parsick.de