howest
hogeschool

Infrastructure Automation

GRADUAAT SYSTEEM- EN NETWERKBEHEER

Alex De Smet

INHOUD

1 INLEIDING

1.1 DEFINITIE

Infrastructure automation wordt gedefinieerd als volgt:

Gebruik maken van technologie om taken uit te voeren met een verminderde menselijke tussenkomst met als doel de software, netwerk, operating system (OS) en dataopslag componenten te beheren die gebruikt worden door IT services en oplossingen. [1]

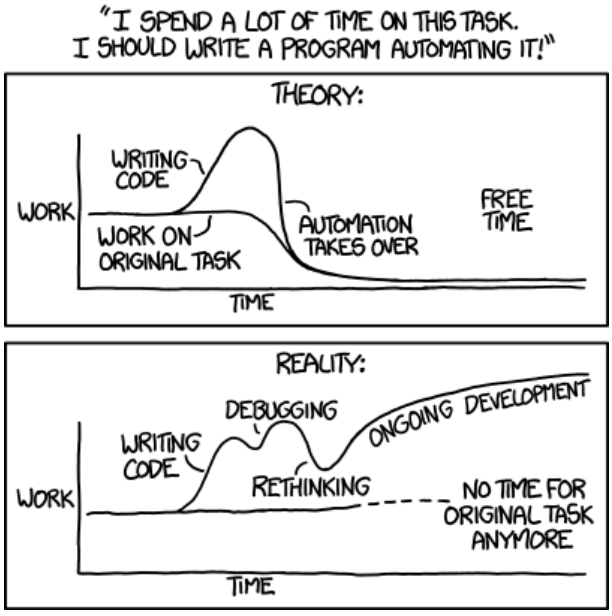
1.2 DE KOST

Als IT professional moeten we vaak repetitieve taken uitvoeren. Denk aan nieuwe user accounts aanmaken, bijkomende software installeren, backups nemen, rechten aanpassen, een nieuw toestel voor een collega installeren, … De tijd die we hieraan spenderen is een kost voor onze organisatie. De kost van repetitief, manueel werk zo laag mogelijk maken is het ware doel van automatisatie.

Een taak automatiseren kan veel tijd besparen in de toekomst. Echter zijn niet alle taken goede kandidaten om te automatiseren. Enkele voorwaarden zijn:

- De taak moet momenteel manueel en relatief frequent uitgevoerd worden
- De taak moet stabiel zijn. Als de vereisten te vaak wijzigen, spendeer je teveel tijd aan het aanpassen van je oplossing
- Je moet controle hebben over alle relevante componenten om de taak te kunnen automatiseren

Het al dan niet automatiseren van een taak kan een technische beslissing zijn om op een bepaalde gestroomlijnde manier te werken, of een business beslissing om tijd (=geld) te besparen. Hierbij is een goede inschatting van hoeveel tijd het kost om die taak te automatiseren van groot belang. Een verkeerde inschatting kan tot gevolg hebben dat je meer tijd spendeert aan het proberen automatiseren dan dat je zou spenderen moest je de taak gewoon manueel gedaan hebben.



Figuur 1: XKCD automation [2]

1.3 DE WAARDE [3]

1.3.1 CONSISTENTIE

Een mens die dezelfde taak een paar honderd keer uitvoert, zal die niet elke keer exact op dezelfde manier uitvoeren. Zelfs met de beste documentatie en wil van de wereld gaan slechts enkelen onder ons zo consistent zijn als een machine. Dit leidt onvermijdelijk tot fouten, stappen die over het hoofd gezien worden en uiteindelijk heeft dit impact op de betrouwbaarheid van de systemen die we beheren. In veel opzichten is het uitvoeren van goed afgebakende procedures één van de belangrijkste waarden van automatisatie

1.3.2 GECENTRALISEERD FOUTBEHEER

Als we onze systemen en services op een geautomatiseerde manier beheren, zijn we ook in staat om misconfiguraties of fouten in code sneller en definitief op te lossen. Moest je dit manueel doen kan je bepaalde onderdelen vergeten of bij het opzetten van een nieuw onderdeel in de toekomst deze stap overslaan. Als je de fix voor zo'n probleem toevoegt aan je automatisch proces dan ben je zeker dat dit in de toekomst meegenomen wordt.

1.3.3 SNELHEID

Nog een bijkomend voordeel van je systemen automatisch te beheren is dat je in het geval van een 'disaster' (datacenter down, server defect, brand, …) relatief snel terug operationeel kan zijn. Zo kan je op een automatische manier servers opnieuw laten aanmaken, alle nodige software installeren, specifieke settings aanpassen en je applicaties terug online brengen. Bedenkt even wat voor een werk dit zou zijn moest je dit allemaal manueel doen.

1.3.4 TIJDSBESPARING

De tijd die je bespaart door een taak niet langer manueel te moeten uitvoeren is één van de meest aangehaalde voordelen van automatisatie. Zoals eerder aangehaald is dit natuurlijk enkel het geval als je minder tijd spendeert aan het automatiseren van een taak dan dat het duurt om die taak manueel uit te voeren op lange termijn.

1.3.5 OVERDRAAGBAARHEID

Repetitieve taken door een automatisch proces laten uitvoeren heeft ook het voordeel dat iedereen die taak kan uitvoeren. Waarbij je anders specifieke kennis bij één of enkele personen hebt, wordt die nu gedeeld over het hele team. Exact op de manier zoals de originele auteur het bedoeld heeft. Dit levert niet alleen tijdsbesparing op voor de persoon die de taak uitvoert, maar voor iedereen die anders moet uitzoeken iets gebeurt.

Joseph Bironas, die een tijdlang Google's datacenters draaiende hield, beargumenteerde:

> *"If we are engineering processes and solutions that are not automatable, we continue having to staff humans to maintain the system. If we have to staff humans to do the work, we are feeding the machines with the blood, sweat, and tears of human beings. Think **The Matrix** with less special effects and more pissed off System Administrators."*

2 SOORTEN AUTOMATISATIE

In dit onderdeel bekijken we wat er precies allemaal geautomatiseerd kan worden en welke tools er bestaan. Dit is geen volledig overzicht van alle beschikbare tooling maar dient enkel om te illustreren wat er mogelijk is.

2.1 SCRIPTING

De meest fundamentele vorm van automatisatie is het bouwen van scripts om je werk gemakkelijker te maken. Scripts kunnen je helpen met automatiseren van taken op één systeem zoals bijvoorbeeld het nemen van backups, user account beheer, opschonen van data, …

Het mooie aan scripts is dat die slechts beperkt zijn tot je eigen fantasie. Met de juiste (combinatie van) tools heb je volledige vrijheid over wat je exact wilt gaan automatiseren.

Afhankelijk van het besturingssysteem heb je verschillende scriptingtalen waarvan je gebruik kan maken. Shell scripting in bash (linux), Powershell (Windows), Python, Go, … Vergelijk scripting in deze context met het schrijven van een programma met als doel een nieuw systeem te bouwen of een wijziging aan te brengen. Neem als voorbeeld onderstaand Powershell script om een nieuwe Hyper-V (virtualisatieoplossing van Microsoft) Virtual Machine aan te maken:

```
# Example generated by ChatGPT

# Hyper-V variables
$vmName = "NewVM"               # Name for the new VM
$vmMemoryStartup = 2GB          # Amount of memory for the VM (in bytes)
$vmHardDiskPath = "C:\VMs"      # Path to store the VM files
$vmHardDiskSize = 50GB          # Size of the VM's hard disk (in bytes)
$vmSwitch = "ExternalSwitch"    # Name of the Hyper-V virtual switch

# Create a new VM configuration
New-VM -Name $vmName -MemoryStartupBytes $vmMemoryStartup -Path
$vmHardDiskPath -NewVHDPath "$vmHardDiskPath\$vmName.vhdx" -NewVHDSizeBytes
$vmHardDiskSize

# Get the network adapter for the VM
$networkAdapter = Get-VMNetworkAdapter -VMName $vmName

# Connect the VM to a virtual switch
Connect-VMNetworkAdapter -VMName $vmName -SwitchName $vmSwitch

# Start the VM
Start-VM -Name $vmName

Write-Host "Virtual machine '$vmName' has been created and started."
```

Of onderstaand bash script dat een backup neemt van alle home-directories op een linux systeem en het resultaat upload naar een Amazon AWS S3 bucket.

```
#!/bin/bash
```

```
# Example generated by ChatGPT

# Variables
backupDir="/tmp/backups"
timestamp=$(date +%Y%m%d_%H%M%S)
backupFile="home_directories_$timestamp.tar.gz"
s3Bucket="your-s3-bucket-name"
awsProfile="your-aws-profile"  # Optional: If using profiles with AWS CLI

# Create a temporary directory for backups
mkdir -p "$backupDir"

# Backup home directories
tar -zcvf "$backupDir/$backupFile" /home/*

# Check if backup was successful
if [ $? -eq 0 ]; then
    echo "Backup completed successfully."

    # Upload backup file to S3
    aws s3 cp "$backupDir/$backupFile" "s3://$s3Bucket/$backupFile" --profile
"$awsProfile"

    # Check if upload was successful
    if [ $? -eq 0 ]; then
        echo "Backup file uploaded to S3 bucket: $s3Bucket"
    else
        echo "Failed to upload backup file to S3 bucket."
    fi

    # Remove temporary backup file
    rm "$backupDir/$backupFile"
else
    echo "Backup failed."
fi

# Remove temporary backup directory
rmdir "$backupDir"
```

2.2 INFRASTRUCTURE AS CODE (IAC)

2.2.1 WAT IS IAC? [4]

Infrastructure as Code is het beheer en provisionen (aanmaken) van systemen met behulp van code of configuratiebestanden. Met IaC zal je voor elk stuk van de infrastructuur objecten aanmaken in bestanden die je infrastructuur voorstellen. Deze bestanden bevatten dan alle instellingen voor je servers, databanken, load balancers, user accounts, permissies, …

Deze bestanden worden dan verwerkt door een IaC tool die je settings neemt en verwerkt tot effectieve infrastructuur. In combinatie met een versiebeheersysteem zoals Git kreeg je volledige observeerbaarheid (wat bestaat er allemaal?) en traceerbaarheid (wie heeft welke aanpassing gemaakt op welk moment) van je infrastructuur. In de meeste gevallen laat dit ook toe om terug te keren naar een vorige staat, een vorige 'versie' van je infrastructuur.

Er zijn verschillende IaC tools op de markt, hieronder vind je de meest gebruikte:

- Terraform
- AWS CloudFormation *
- Azure Resource Manager *
- Google Cloud Deployment Manager *
- Pulumi
- Ansible (zowel IaC als config management tool)
- Chef
- Puppet
- OpenTofu **

* Deze IaC tools zijn ingebed in hun eigen Cloud Platform en kunnen meestal enkel objecten beheren daarbinnen. Er zijn wat pogingen om multi-cloud setups ook te ondersteunen via deze tools maar voorlopig zijn die eerder beperkt.

** In 2023 besliste Hashicorp om Terraform onder een andere licentie uit te brengen. De open source community heeft als reactie hierop een fork gemaakt om de open source mentaliteit te kunnen blijven garanderen. Deze fork heet OpenTofu en is sinds januari 2024 GA (Generally Available). Op het moment van schrijven is de toekomst van deze fork nog onvoorspelbaar dus gaan we er momenteel niet verder op in.

IaC tools zijn ofwel declaratief ofwel imperatief. Declaratief betekent dat je beschrijft hoe je wilt dat de infrastructuur er uit ziet. Imperatief beschrijft een procedure hoe je een bepaalde taak bereikt, welke stappen er nodig zijn om tot het eindpunt te geraken. Je bepaalt als auteur zelf de volgorde in een imperatief proces, bij een declaratieve aanpak beslist de IaC tool hoe dat gebeurt.

2.2.2 TERRAFORM

In 2011 lanceerde AWS CloudFormation als IaC tool. Mitchel Hashimoto, de founder van Hashicorp, was zo onder de indruk van deze tool dat hij de dag nadien besloot om een open-source, cloud-agnostic (cloud-onafhankelijke) tool te maken die in staat was om dezelfde functionaliteit en workflows aan te bieden aan de bredere community. Het heeft nog drie jaar geduurd, tot 2014, voor Hashicorp Terraform 1.0 uitbracht. Die versie had enkel ondersteuning voor AWS en DigitalOcean. In de jaren die er op volgden kende Terraform geen groot succes. Downloads stagneerden en de adoptie door de community bleef uit. Tot 2016-2017 Terraform plots een vliegende doorstart nam met hun eerste partnership met Microsoft om hun Azure cloud platform volledig te ondersteunen. Momenteel is Terraform dé industriestandaard om Cloud en on-premise infrastructuur te beheren. [5]

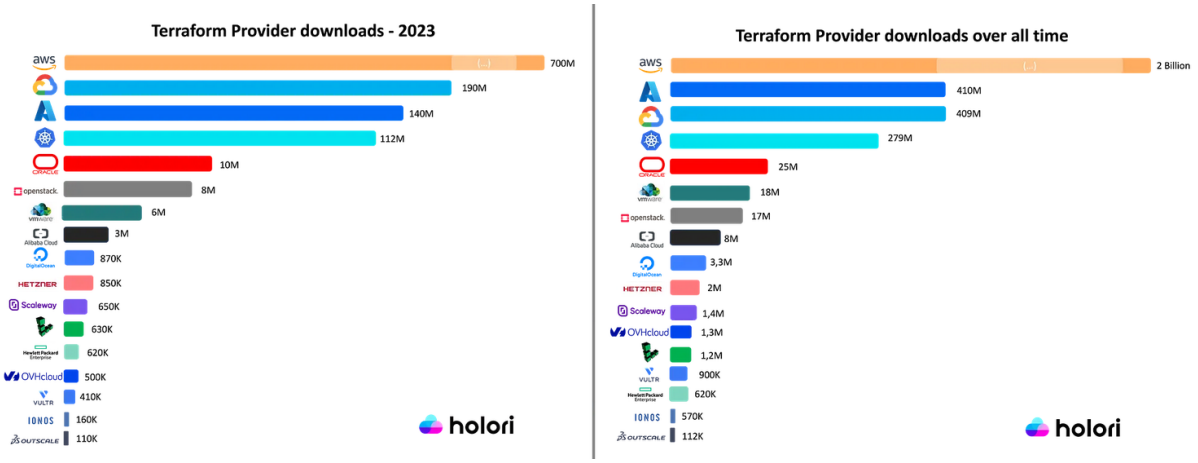
2.2.2.1 WERKING

Terraform maakt en beheert 'resources'. Dit kan van alles zijn: een volledige server (VM), een firewall regel, een IP-adres, … Deze resources worden aangemaakt via een Terraform provider.

Figuur 2: Basis werking Terraform [6]

Deze providers bevatten de definitie van hoe een resource eruit ziet én hoe die op het bijhorend (cloud) platform moet aangemaakt worden. Zo heb je providers voor elk cloud platform, maar ook voor andere producten zoals rediscloud, mongodb, kubernetes, … Een overzicht van alle providers vind je hier: https://registry.terraform.io/browse/providers

Het is mede door het grote aanbod van providers dat Terraform zo'n succes geworden is. Als je een resource aanmaakt via een provider dan zal die communiceren met de API van dat platform of product en zo de nieuwe objecten aanmaken. Hieronder zie een overzicht van het aantal keer dat een terraform provider gedownload is begin 2023 en all time. Waarbij het geen verassing mag zijn dat de grote cloud providers (AWS, GCP en Azure) dominant zijn.



Figuur 3: Terraform Provider downloads [7]

Terraform maakt gebruik van eigen ontwikkelde syntax, Hashicorp Configuration Language of HCL. Je zou dit kunnen vergelijken met JSON of YAML qua structuur maar die twee zijn bedoelt om data structuren te formatteren terwijl HCL bedoeld is als een configuratieformaat voor resources.

Om een goed idee te krijgen hoe Terraform werkt, kan je een kijkje nemen in Appendix A: Terraform AWS VM.

2.3 CONFIGURATION MANAGEMENT

Configuration Management (CM) is een onderdeel van de automatisatiewereld waarbij de focus niet ligt op het aanmaken van de infrastructuur zelf, maar de software en settings op die systemen die we nodig hebben om diensten aan te kunnen bieden.

Een goed voorbeeld om dit wat tastbaarder te maken is om te denken aan de server die we eerder aanmaakten via IaC. Stel dat van die VM een webserver willen maken die draait onder een aparte user

account met een firewall die onze machine beschermt dan is een IaC tool niet de best geschikte tool daarvoor. Hoewel de grens tussen IaC en CM bij sommige tools wat aan het vervagen is, kan je stellen dat een IaC de infrastructuur aanmaakt en een CM tool de software en settings beheert op die infrastructuur.

2.3.1 ANSIBLE

Ansbile werd gelanceerd in 2012 door Michael De Haan. In 2015 kocht RedHat deze tool en werd die omgedoopt tot het Red Hat Ansible Automation Platform. Ansible maakt gebruik van SSH en andere remote connectiemogelijkheden om taken (Tasks) op een geautomatiseerde manier uit te voeren. Om te starten met Ansible volstaat het om Python te installeren en SSH-toegang te hebben tot de systemen die je wilt beheren. Mede door de eenvoud van Ansible is deze tool zo populair geworden.

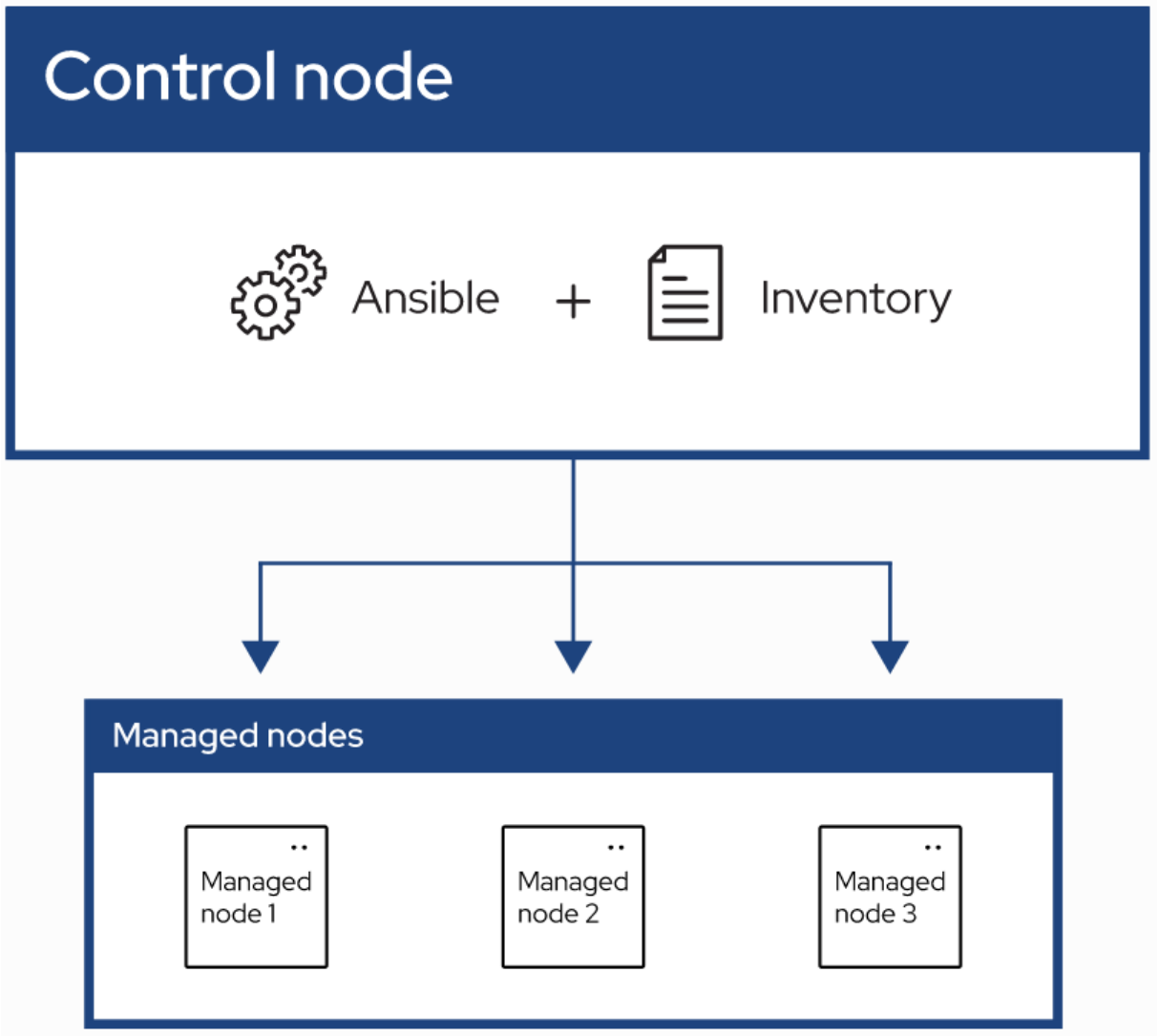
Met Ansible kan je zowel declaratief en imperatief werken. We gaan Ansible gebruiken in het verder verloop van deze cursus dus bespreken we de verschillende onderdelen in een apart hoofdstuk.

3 ANSIBLE

3.1 INLEIDING

Ansible is een configuration management tool dat gekend is voor zijn simpliciteit en uitbreidbaarheid. Je definieert de state en Ansible zorgt ervoor dat die state consistent gehouden wordt over alle beheerde toestellen heen.

De meeste Ansible omgevingen bestaan uit 3 componenten [8]:



1. Ansible Control Node: een systeem dat de Ansible software bevat.
2. Inventory: lijst van de nodes of hosts die je managed.
3. Managed nodes: systemen onder het beheer van Ansible.

3.1.1 ANSIBLE CONTROL NODE

Om de Ansible Control Node te maken hebben we een systeem nodig waarop we de nodige software installeren. Dit kan je eigen laptop zijn, een VM of container of zelfs deel uitmaken van een Continuous Deployment (CD) pipeline.

Omdat simpliciteit één van de kernwaarden van Ansible is, zijn het aantal vereisten om deze software werkende te krijgen vrij beperkt:

- Een recente, werkende Python installatie
- pipx (Python package installer) of pip
- Een UNIX systeem. Windows wordt niet ondersteund.
- (Optioneel) Een gebruiker met als naam “ansible”. Hoewel niet verplicht, zal dit de setup wel een wat makkelijker maken.

Eens aan deze voorwaarden voldaan is, kan je Ansible installeren onder de Ansible user account. Je hebt hierbij verschillende opties.

Optie 1: Package manager

Veel distributies bieden een package aan. Bekijk de info hier:

```
$ dnf info ansible

$ apt show ansible
```

Installeren doe je via:

```
$ sudo dnf install ansible

$ sudo apt install ansible
```

Optie 2: Python Installer for Python (pip/pipx)

Aangezien Ansible gebouwd werd in Python, kan je ook via pip of pipx installeren.

```
$ pipx install ansible-core

$ pipx install ansible

$ pipx ensure-path
```

Controle

Controleer of de installatie gelukt is via:

```
$ ansible --version (core)

$ ansible-community --version (full version)
```

Nu we de nodige tools hebben kunnen we Ansible al eens uitproberen op localhost, de Ansible Control Node zelf.

Met onderstaand commando controleer je of een Ansible host reageert:

```
$ ansible -m ping localhost
```

Output:

```
alex@debian:~$ ansible -m ping localhost
[WARNING]: No inventory was parsed, only implicit localhost is available
localhost | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Commando's laten uitvoeren kan je op volgende manier doen:

```
$ ansible localhost -a 'hostnamectl'
```

Output:

```
alex@debian:~$ ansible localhost -a 'hostnamectl'
[WARNING]: No inventory was parsed, only implicit localhost is available
localhost | CHANGED | rc=0 >>
 Static hostname: debian
       Icon name: computer-vm
         Chassis: vm
      Machine ID: e3c4fad885ca4b53ae241f10bb1ac85e
         Boot ID: d2b841613d78404983363ae36a099d55
  Virtualization: vmware
Operating System: Debian GNU/Linux 12 (bookworm)
          Kernel: Linux 6.1.0-16-amd64
    Architecture: x86-64
 Hardware Vendor: VMware, Inc.
  Hardware Model: VMware Virtual Platform
Firmware Version: 6.00
```

Vervang hostnamectl in bovenstaand commando door gelijk wat en Ansible zal het uitvoeren. Als het commando output teruggeeft, dan zal dat als response komen.

Let wel, op deze manier kan je geen complexe commando's gebruiken zoals onderstaand voorbeeld aantoont.

```
alex@debian:~$ ansible localhost -a 'ls -al'
[WARNING]: No inventory was parsed, only implicit localhost is available
localhost | CHANGED | rc=0 >>
total 32
drwx------ 3 alex alex 4096 Dec 18 14:15 .
drwxr-xr-x 3 root root 4096 Dec 18 11:31 ..
drwxr-xr-x 3 alex alex 4096 Dec 18 14:15 .ansible
-rw------- 1 alex alex  127 Dec 18 11:49 .bash_history
-rw-r--r-- 1 alex alex  220 Dec 18 11:31 .bash_logout
-rw-r--r-- 1 alex alex 3526 Dec 18 11:31 .bashrc
-rw-r--r-- 1 alex alex  807 Dec 18 11:31 .profile
-rw------- 1 alex alex   52 Dec 18 14:15 .Xauthority
alex@debian:~$ ansible localhost -a 'ls -al > output.txt'
[WARNING]: No inventory was parsed, only implicit localhost is available
localhost | FAILED | rc=2 >>
ls: cannot access '>': No such file or directory
ls: cannot access 'output.txt': No such file or directorynon-zero return code
alex@debian:~$ ansible localhost -a 'ls -al | grep bash'
[WARNING]: No inventory was parsed, only implicit localhost is available
localhost | FAILED | rc=2 >>
ls: cannot access '|': No such file or directory
ls: cannot access 'grep': No such file or directory
ls: cannot access 'bash': No such file or directorynon-zero return code
```

Natuurlijk is het de bedoeling dat we met onze Control Node andere hosts gaan beheren. Om dat te kunnen doen, moeten we ergens een lijst gaan bijhouden van welke hosts we beheren. Zo'n lijst heet een inventory.

3.1.2 INVENTORY

De Ansible inventory is een bestand dat alle Managed Nodes bevat. In zijn meest eenvoudige vorm is dit een lijst met IP-adressen of hostnames. Je kan hier echter meer structuur inbouwen om hosts te gaan groeperen. Hieronder enkele verschillende aanpakken.

Groeperen per functie:

```
[web]
web1.example.com
web2.example.com
[database]
db1.example.com
```

```
db2.example.com

[loadbalancer]

lb1.example.com
```

Groeperen per omgeving:

```
[production]

web1.example.com

db1.example.com

[staging]

web2.example.com

db2.example.com

[testing]

test1.example.com

test2.example.com
```

Geneste groepering:

```
[web]
web1.example.com
web2.example.com

[db]
db1.example.com
db2.example.com

[app:children]
web
db
```

In bovenstaand voorbeeld gebruiken we geneste groepering waarbij er één parent group is (app) die bestaat uit de web en db children groups.

Hoe je jouw inventory structureert, bepaal je zelf volgens wat logisch lijkt in jouw context. Denk hier goed over na omdat dit impact zal hebben op hoe je verder automatiseert.

Je kan ook gebruik maken van de twee default groups: *all* en *ungrouped*. De *all* group bevat alle hosts, de *ungrouped* zijn alle hosts die in geen enkele group zitten, behalve de *all* group.

3.2 PREREQUISITES

Ansible is een agentless configuration management tool. Dit betekent dat je geen bijkomende software op de nodes hoeft te installeren. Standaard wordt gebruik gemaakt van (Open)SSH verbindingen

waarover commando's gestuurd worden naar de Managed Nodes. Er zijn nog andere mogelijkheden om verbinding te maken. Die mogelijkheden worden aangeboden via Connection Plugins. Maar daarover later meer.

Om een SSH-verbinding op te zetten tussen de Control Node en de Managed Nodes heb je uiteraard een Linux gebruikersaccount nodig. Aangezien we administratieve taken zullen uitvoeren, zal dat account ook verhoogde rechten moeten hebben.

Het is een best practice om een aparte Ansible Linux gebruikersaccount aan te maken. Ansible verwacht dat we gaan werken met SSH-keys wat ook een stuk veiliger is dan password-based authentication.

Om te vermijden dat je dit werk elke keer manueel moet doen, zou je het aanmaken van het Ansible user account en het toevoegen van de public SSH-key kunnen integreren in je provisioning stap. Dit kan je bijvoorbeeld doen via een IaC tool naar keuze (bv. Terraform) of door templates van VM's te maken. Of we kunnen gebruik maken van onze versie Ansible installatie om de initiële instellingen te doen.

Op elke Managed Node moet je deze voorbereidende stappen doen:

Nieuwe gebruiker aanmaken (bv ansible):

```
$ sudo adduser ansible
```

Sudo privileges toekennen (toevoegen aan sudo groep):

```
$ sudo usermod -aG sudo ansible
```

Sudo kunnen uitvoeren zonder wachtwoord*:

```
$ sudo visudo
```

en dit toevoegen onderaan het bestand:

```
ansible ALL=(ALL) NOPASSWD: ALL
```

* sudo kunnen uitvoeren zonder wachtwoord is nodig omdat we bepaalde taken automatisch willen uitvoeren zonder input van een bijkomend commando.

Wat **sudo visudo** eigenlijk doet is het aanpassen van de /etc/sudoers file. Een alternatieve aanpak is het gebruiken van de /etc/sudoers.d/ directory. Op die manier je custom settings afsplitsen

Controle:

```
$ su ansible
ansible@HOW-CYWLJS3:~$ sudo cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
_apt:x:42:65534::/nonexistent:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:998:998:systemd Network Management:/:/usr/sbin/nologin
alex:x:1000:1000:,,,:/home/alex:/bin/bash
ansible:x:1001:1001:,,,:/home/ansible:/bin/bash
```

Als er geen wachtwoord gevraagd wordt bij het uitvoeren van een privileged commando is de test succesvol.

Het enige wat nu nog rest is om key-based authentication te configuren. Volg onderstaande stappen op de Control Node:

Genereer SSH keys:

SSH keys zijn maar zo sterk als het gebruikte algoritme. Het is altijd risicovol om bepaalde statements in een statisch document te verwerken, maar op het moment van schrijven ga je best voor ofwel een RSA key pair van ten minste 4096 bits groot of nog beter, een ED25519 key-pair. ED25519 heeft de volgende voordelen in vergelijking met RSA:

- Ander encryptiealgoritme (Elliptic Curve Cryptography of ECC) waardoor je minder bits nodig hebt om een even sterke beveiliging te verkrijgen als RSA.
- resulteert in kleinere keys waardoor je minder overhead en snellere verwerking krijgt
- snellere key generatie. Dit is vooral van belang in omgevingen waarbij je keys snel geroteerd (moeten) worden.

Een key-pair genereren doe je met het ssh-keygen commando:

```
# Elliptic Curve Cryptography key pair
$ ssh-keygen -t ed25519

OF

# RSA key pair (4096 bits)
$ ssh-keygen -t rsa -b 4096
```

Zorg ervoor dat je dit key pair enkel voor Ansible gebruikt zodat je dit snel kan roteren indien nodig. Bovenstaand commando geeft je twee bestanden (naamgeving is afhankelijk van het gekozen algoritme). Deze twee bestanden zijn je private key (zonder extensie) en je public key (.pub extensie)

ED25519: id_ed25519 & id_ed25519.pub

RSA: id_rsa & id_rsa.pub

Je private key is één van de belangrijkste secrets die je moet beveiligen. Hiermee gaan we namelijk alle Managed Nodes gaan beheren met een account die sudo rechten heeft. Het spreekt voor zich dat we deze key zo goed mogelijk moeten beschermen.

Als laatste stap moeten we de public key die we net aangemaakt hebben, koppelen aan de Ansible user op de Managed Node.

```
$ ssh-copy-id -i <pad-naar-je-public-key> ansible@<host>
```

Test of je succesvol kan verbinden vanaf je Control Node naar de Managed Node:

```
$ ssh ansible@<host>
```

Als de test geslaagd is, wordt er geen wachtwoord gevraagd. Nu alles klaar staat, kunnen we starten met het verkennen van Ansible!

3.3 AD-HOC COMMANDO'S [9]

Ad-hoc commando's zijn manuele ingrepen op infrastructuur die je éénmalig uitvoert. Zo'n commando's zijn snel en gemakkelijk, maar niet herbruikbaar. We gebruiken deze commando's om een idee te krijgen hoe Ansible werkt.

De syntax voor een ad-hoc commando is altijd hetzelfde:

```
$ ansible [pattern] -m [module] -a "[module options]"
```

Dit commando zal een bepaalde module (-m) met bijhorende opties (-a) uitvoeren op het target (pattern). We bespreken modules in een verder stuk. Om het concept van ad hoc commando's toch nu al iets tastbaarder te maken, vind je hieronder een aantal voorbeelden.

Vb 1: Kopiëren van een bestand (/etc/hosts) naar alle hosts in de group "atlanta":

```
$ ansible atlanta -m ansible.builtin.copy -a "src=/etc/hosts dest=/tmp/hosts"
```

Vb 2: Tijdszone van de hosts op "Europe/Brussels" zetten in de group webservers:

```
$ ansible webservers -i inventory -a "timedatectl set-timezone Europe/Brussels" --become
```

Bij bovenstaand commando merk je misschien wat dingen op:

1. Er is geen module meegegeven? Dat komt omdat de default module voor Ansible de *ansible.builtin.command* module is. Daarmee kan je (beperkt) shell commando's uitvoeren.
2. -i inventory: hiermee geef je zelf het pad op naar je inventory bestand. Op deze manier kan je meerdere inventory files hebben op één control node.
3. --become: sommige administratieve taken kan je niet uitvoeren onder een standaard user account. Door de parameter *--become* activeer je priviledge escalation en zal het commando met verhoogde rechten uitgevoerd worden.

Vb 3 :‘Facts’ verzamelen:

```
$ ansible all -m ansible.builtin.setup
```

Dit commando zal informatie over de hosts zelf (metadata) opvragen en weergeven. Deze ‘facts’ kunnen we ook gebruiken om doelgericht bepaalde acties uit te voeren. Bijvoorbeeld enkel de hosts met meer dan 10 GB vrije disk ruimte een nieuw softwarepakket uitrollen. Of alle op RHEL gebaseerde hosts gaan targetten.

Om je een beter idee te geven hoe die facts er uit zien, vind je hieronder een voorbeeld output. Let op: hier werd behoorlijk wat output weggelaten om het leesbaar te houden. Voor een volledig overzicht van de facts voer je het commando best zelf eens uit.


```
192.168.203.136 | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "172.17.0.1",
            "192.168.49.1",
            "192.168.203.136"
        ],
        "ansible_architecture": "x86_64",
        "ansible_bios_date": "11/12/2020",
        "ansible_bios_vendor": "Phoenix Technologies LTD",
        "ansible_bios_version": "6.00",
        "ansible_board_asset_tag": "NA",
        "ansible_board_name": "440BX Desktop Reference Platform",
        "ansible_board_serial": "NA",
        "ansible_board_vendor": "Intel Corporation",
        "ansible_board_version": "None",
        "ansible_distribution": "Debian",
        "ansible_distribution_file_parsed": true,
        "ansible_distribution_file_path": "/etc/os-release",
        "ansible_distribution_file_variety": "Debian",
        "ansible_distribution_major_version": "12",
        "ansible_distribution_minor_version": "2",
        "ansible_distribution_release": "bookworm",
        "ansible_distribution_version": "12.2",
        "ansible_dns": {
            "domain": "localdomain",
            "nameservers": [
                "192.168.203.2"
            ],
            "search": [
                "localdomain"
            ]
        },
        "ansible_virtualization_tech_host": [],
        "ansible_virtualization_type": "VMware",
        "discovered_interpreter_python": "/usr/bin/python3",
```

```
        "gather_subset": [
            "all"
        ],
        "module_setup": true
    },
    "changed": false
}
```


3.4 MODULES

Modules zijn hoofdbouwstenen van Ansible. Dit zijn stukjes voorgebouwde code die je kan gebruiken om wijzigingen uit te voeren om de systemen die je beheert. Er zijn heel wat modules die gebouwd en onderhouden worden door het Ansible core team. Deze modules starten allemaal met de prefix 'ansible.builtin'. In het onderdeel ad-hoc commando's gebruikten we al een paar van deze modules:

- ansible.builtin.command: de default module waarmee je commando's kan uitvoeren
- ansible.builtin.setup: haalt de facts op van de hosts
- ansible.builtin.copy: kopieert bestanden naar de hosts

Je kan een overzicht van alle builtin modules terugvinden in de officiële documentatie:

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/index.html

De builtin modules staan garant voor kwaliteitsvolle én goed onderhouden modules. Er is echter ook een groot aantal modules dat door de community of externe vendoren werden gebouwd. Doordat Ansible open-source is, is de uitbreidbaarheid en flexibiliteit een grote troef.

Om een idee te krijgen hoeveel modules er bestaan, vind je hier alle modules in Ansible 2.9:

https://docs.ansible.com/ansible/2.9/modules/list_of_all_modules.html

Omdat Ansible veel vrijheid laat om een bepaalde taak tot een goed einde te brengen, heb je vaak verschillende mogelijkheden om een taak te verwezenlijken. Willen we bijvoorbeeld de hostname van een webserver aanpassen, kunnen we dat op deze manieren doen.

Via de hostname module:

```
$ ansible webserver1 -i inventory -m ansible.builtin.hostname -a "name=web-001" --become
```

Via een command (hostnamectl):

```
$ ansible webserver1 -i inventory -m ansible.builtin.command -a "hostnamectl set-hostname web-001" --become
```

Aangezien dit de default module is, kunnen we dit ook schrijven als:

```
$ ansible webserver1 -i inventory -a "hostnamectl set-hostname web-001" --become
```

Door het manipuleren van config files (2 stappen):

```
$ ansible webserver1 -i inventory -b -m lineinfile -a "path=/etc/hostname line='web-001' create=yes state=present"
```

```
$ ansible webservers -i inventory -b -m service -a "name=systemd-hostnamed state=restarted"
```

Hoewel de drie bovenstaande opties exact hetzelfde eindresultaat hebben, zit er een belangrijk verschil tussen de eerste en de laatste twee. Bij de eerste werk je declaratief en zal Ansible kijken of de state overeenkomt met wat je wilt bereiken. Enkel als er een actie nodig is, zal Ansible iets uitvoeren. Bij de laatste twee opties worden de commando's elke keer uitgevoerd, ongeacht of de hostname al goed stond of niet. Deze aanpak is dus imperatief.

Zoals je ziet is het relatief eenvoudig om via de CLI wijzigingen aan te brengen aan de infrastructuur die je beheert. Maar stel je eens volgend scenario voor.

Je hebt een host in je beheer waarbij je verschillende acties uitgevoerd hebt via de CLI:

- Hostname goed gezet
- Tijdszone aangepast
- Software geïnstalleerd via de package manager
- Files gekopieerd met allerlei settings
- Firewall regels aangepast
- …

Op een dag wil je om redundantie in te bouwen een tweede host aanmaken met exact dezelfde aanpassingen. In dit scenario zou je dan een extra regel toevoegen in je inventory bestand en dan elk CLI commando opnieuw uitvoeren totdat de state van de twee machines hetzelfde is. Niet bepaald praktisch.

Een beter alternatief om hetzelfde te bereiken is het gebruik van Ansible Playbooks.

3.5 SELECTEREN VAN GOEDE INFOBRONNEN

Ansible is een veel gebruikte tool binnen de configuration management wereld. Online vind je dan ook heel veel informatie en voorbeelden terug hoe je iets kan bereiken. Zoals alles op het internet varieert de kwaliteit van die bronnen dus is een kritisch oog aan te raden.

Als je iets wilt opzoeken, is de beste plaats om te starten de officiële Ansible documentatie op https://docs.ansible.com/ Deze bevat de meest recente info en volgt de best practices vooropgesteld door het core team. Deze documentatie is ook de hoofdbron geweest van deze cursus dus zal je misschien wat voorbeelden herkennen.

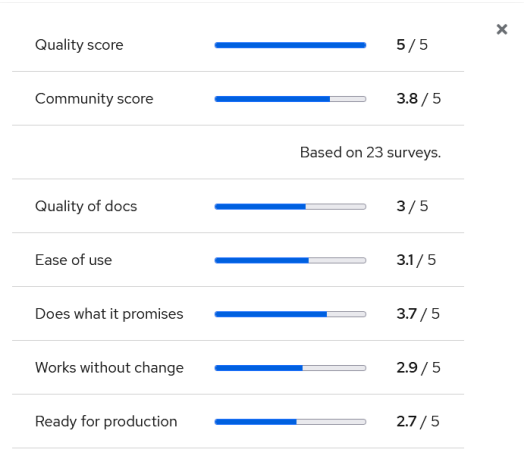
Als je geen toegang hebt tot de online resources kan je het **ansible-doc** commando gebruiken. Daarmee kan je van elke mdoule informatie opvragen via de CLI. Om bijvoorbeeld de werken van de ansible.builtin.copy module op te vragen volstaat het om **ansible-doc copy** uit te voeren:

```
alex@debian:~$ ansible-doc copy
> ANSIBLE.BUILTIN.COPY    (/usr/lib/python3/dist-packages/ansible/modules/copy.py)

        The `copy' module copies a file from the local or remote machine to a location on the remote machine. Use the
        [ansible.builtin.fetch] module to copy files from remote locations to the local box. If you need variable
        interpolation in copied files, use the [ansible.builtin.template] module. Using a variable in the `content' field will
        result in unpredictable output. For Windows targets, use the [ansible.windows.win_copy] module instead.

ADDED IN: historical

  * note: This module has a corresponding action plugin.

OPTIONS (= is mandatory):

- attributes
        The attributes the resulting filesystem object should have.
        To get supported flags look at the man page for `chattr' on the target system.
        This string should contain the attributes in the same order as the one displayed by `lsattr'.
        The `=' operator is assumed as default, otherwise `+' or `-' operators need to be included in the string.
        aliases: [attr]
        default: null
        type: str
        added in: version 2.3 of ansible-core


- backup
        Create a backup file including the timestamp information so you can get the original file back if you somehow
        clobbered it incorrectly.
        default: false
        type: bool
        added in: version 0.7 of ansible-core


- checksum
        SHA1 checksum of the file being transferred.
        Used to validate that the copy of the file was successful.
        If this is not provided, ansible will use the local calculated checksum of the src file.
        default: null
        type: str
        added in: version 2.5 of ansible-core
```

Om een overzicht te krijgen van wat je allemaal kan doen met deze tool, vraag je de help-pagina op:

```
$ ansible-doc -h
$ ansible-doc --help
```

Als je resources zoekt die door de community gebouwd werden, ga eerst eens kijken naar Ansible Galaxy (https://galaxy.ansible.com/ui/). Op die site vind je heel veel objecten die mogelijks doen wat je nodig hebt. Ook hier heb je veel opties die hetzelfde of gelijkaardige dingen doen. Een goede manier om vast te stellen of iets kwalitatief is, is het aantal downloads te bekijken. Een tweede check is de rating die andere community leden aan die resource gegeven hebben.



Figuur 4 Rating van een MySQL resource op Ansible Galaxy

Je kan ook terecht bij AI tools zoals ChatGPT om je te helpen bij het bouwen van je configuratie. Die informatie dubbelcheck je beter op fouten. Ook volgt de gegenereerde syntax niet altijd de best practices

vooropgesteld door de community en het core team. Zoals we later gaan zijn kan Ansible Lint hierbij helpen om je configuratie correct en goed opgebouwd te houden.

3.6 PLAYBOOKS [10]

3.6.1 INTRO

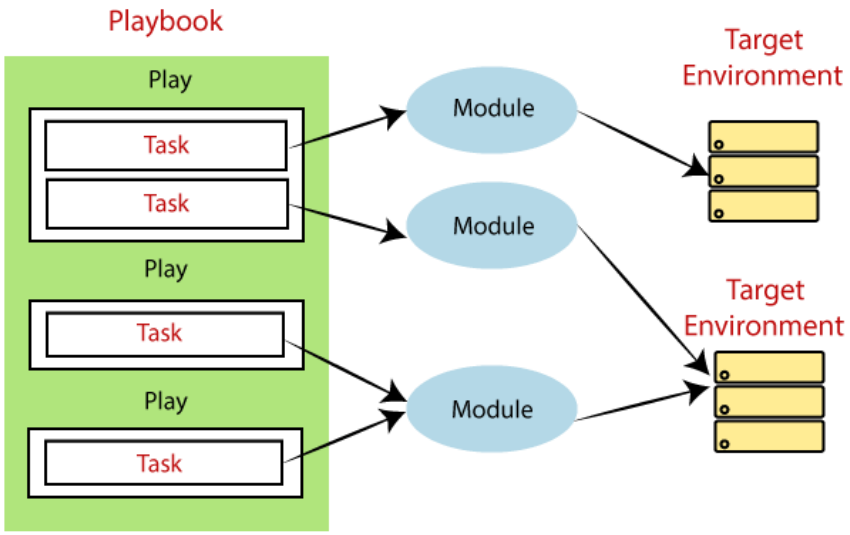
Ansible Playbooks zijn bestanden die verschillende taken logisch bundelen om een bepaalde state te bereiken. Deze taken worden in volgorde uitgevoerd en zorgen voor herhaalbare procedures. Een playbook tweemaal na elkaar laten uitvoeren mag geen probleem geven voor de uiteindelijke state van een host.

Playbooks worden gemaakt in YAML (Yet Another Markup Language). Elke YAML-file (ongeacht of je die gebruikt voor Ansible of niet) start met 3 liggende streepjes '---'. Hieronder vind je een voorbeeld van een playbook:

```
---
- name: Update web servers
  hosts: webservers
  remote_user: ansible

  tasks:
  - name: Ensure apache is at the latest version
    ansible.builtin.yum:
      name: httpd
      state: latest

  - name: Write the apache config file
    ansible.builtin.template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf

- name: Update db servers
  hosts: databases
  remote_user: ansible

  tasks:
  - name: Ensure postgresql is at the latest version
    ansible.builtin.yum:
      name: postgresql
      state: latest

  - name: Ensure that postgresql is started
    ansible.builtin.service:
      name: postgresql
      state: started
```

Bovenstaand playbook bevat 2 'plays'. Één om de group 'webservers' en één om group 'databases' te

updaten. Elke Play bestaat hier uit twee taken of tasks. Elke task is een module die aangesproken wordt. De samenhang van die objecten ziet er zo uit:



Figuur 5: Ansible Playbook architecture [11]

Een playbook runnen doe je met het **ansible-playbook** commando. Bijvoorbeeld:

```
$ ansible-playbook -i inventory my_playbook.yml
```

Dit zal de tasks gedefinieerd in *my_playbook.yml* uitvoeren op alle hosts in de inventory.

```
$ ansible-playbook -i inventory --limit webservers my_playbook.yml
```

Met een limit beperk je het uitvoeren van dat playbook tot enkel die group of host.

```
$ ansible-playbook --syntax-check my_playbook.yml
```

Om te controleren of je YAML correct opgebouwd is, gebruik je de **--syntax-check** optie. Alle opties voor dit commando bekijk je met:

```
$ ansible-playbook --help
```

Probeer je playbooks te organiseren volgens het functie van de host die je beheert. Dit kan overeenkomen met de groups die je aanmaakt in je inventory maar dat hoeft zeker niet zo. Bekijk telkens de specifieke eisen van je project en maak een structuur dat logisch lijkt voor jou.

3.6.2 PLAYBOOK VOORBEELD

Ansible is gebouwd om modulair en onderhoudbaar te zijn. Om die reden zijn er verschillende mogelijkheden om je configuratie op te bouwen. Hieronder volgt een voorbeeld om die flexibiliteit aan te tonen.

Vb: Applicatie die uit meerdere, duidelijk afgescheiden onderdelen bestaat. Die applicatie bestaat uit een Python applicatie als API en een webserver als frontend in javascript. Het playbook zou er als volgt kunnen uitzienn (niet erg als je nog niet alles snapt momenteel):

```yaml
- hosts: app_servers
  become: true

  tasks:
    - name: Install OS packages required for the application
      package:
        name: "{{ item }}"
        state: present
      loop:
        - python3
        - firewalld
        - nginx
        # Add other required packages here

    - name: Create 'app' user with specific credentials
      user:
        name: app
        shell: /bin/bash
        # Add other user attributes like uid, password, etc. if needed
        createhome: yes

    - name: Copy configuration files for the application
      copy:
        src: /path/to/local/config/file
        dest: /destination/path/on/remote/server
        # Add other configuration files and destinations

    # FIREWALL CONFIGURATION
    - name: Allow HTTP traffic (port 80) through firewalld
      firewalld:
        service: http
        permanent: yes
        state: enabled

    - name: Allow HTTPS traffic (port 443) through firewalld
      firewalld:
        service: https
        permanent: yes
        state: enabled

    - name: Reload firewalld to apply changes
      service:
        name: firewalld
        state: reloaded
    # Reload firewalld to apply new rules

    - name: Install Python and required packages for API
      apt:
```

```
      name: "{{ item }}"
      state: present
    loop:
      - python3
      - python3-pip
      - # And other necessary packages for your Python app

  - name: Copy API systemd service file
    copy:
      src: /path/to/local/api.service
      dest: /etc/systemd/system/api.service
    notify:
      - Reload systemd

  - name: Start API service
    systemd:
      name: api
      state: started
      enabled: yes

  - name: Copy frontend tar file to the server
    copy:
      src: /path/to/local/frontend.tar.gz
      dest: /path/on/remote/server/frontend.tar.gz

  - name: Unarchive the website
    unarchive:
      src: /path/on/remote/server/frontend.tar.gz
      dest: /path/on/remote/server/extracted_frontend
      remote_src: yes
```

Merk op dat deze manier van werken functioneel werkt, maar vrij snel onoverzichtelijk kan worden. In een later stadium gaan we zien hoe we onze playbooks beter kunnen structureren met roles.

3.6.3 VARIABELEN [12]

Variabelen worden binnen Ansible gebruikt om de verschillen tussen systemen op te vangen. Op die manier kan je stukken configuratie hergebruiken en hoef je minder onderdelen te herhalen. Je kan variabelen gebruiken in playbooks, in je inventory of in herbruikbare bestanden (zie verder). Je kan zelfs (stukken van) de output van één task gebruiken als input variabele voor een volgende task.

3.6.3.1 EENVOUDIGE VARIABELEN

Je hebt binnen Ansible een heleboel ingebouwde variabelen die je kan gebruiken, maar je kan natuurlijk ook zelf variabelen gaan instellen. Variabelen zijn key-value paren:

```
os: debian
version: 12.4
script_dir: /opt/scripts
```

Variabele namen zijn strings (tekstuele waardes) die onderhevig zijn een aantal regels om geldige variabele namen te zijn:

- Geldige tekens zijn letters, cijfers en underscores
- Kunnen beginnen met een underscore. In de meeste programmeertalen heeft dit een speciale betekenis, bij Ansible is dit niet het geval.
- Python keywords zoals *def*, *async*, *break*, … mogen niet. Voor een volledig overzicht, zie: https://docs.python.org/3/reference/lexical_analysis.html#keywords
- Playbook keywords mag je ook niet gebruiken. Bijvoorbeeld *hosts*, *environment*, *tasks*, … Zie https://docs.ansible.com/ansible/latest/reference_appendices/playbooks_keywords.html#playbook-keywords

Eens je een variabele gedefinieerd hebt, kan je die gebruiken door de variabele naam tussen twee paar accolades te zetten. Bv:

```
{{ script_dir }}
```

Een voorbeeldplaybook maakt veel duidelijk:

```
- hosts: web_servers
  vars:
    app_version: "v1.2.3"
    path_to_website: "/var/www/my_website"

  tasks:
    - name: Display defined variables
      debug:
        msg: "App Version: {{ app_version }}, Website Path: {{ path_to_website
}}"

    - name: Deploy website to defined path
      copy:
        src: /local/path/to/website
        dest: "{{ path_to_website }}"
```

Je ziet dat de variabele stukken tussen quotes staan. Dit heeft te maken met de manier waarop YAML werkt. Een dubbele punt die gevolgd wordt door een accolade wordt geïnterpreteerd als de start van een YAML dictionary.

3.6.3.2 YAML GOTCHA'S [13]

We staan even stil bij een paar mogelijke valkuilen bij het definiëren van YAML syntax. Zoals we hierboven al aangeven zal een : gevolgd door { de start aangeven van een YAML dictionary. Op dezelfde manier zal een : gevolgd door een spatie ": " de start van een mapping betekenen en een spatie gevolgd door een hekje " #" is de start van een commentaar block.

Deze regels gaan dus allemaal een foutmelding geven:

```
version: v1.12.2: Bookworm
win_dir: c:
```

Maar dit werkt wel nog want er staat geen spatie na de dubbele punt:

```
win_dir: c:\windows
```

Om de twee bovenstaande voorbeelden wel te doen werken, kan je quotes toevoegen:

```
version: 'v1.12.2: Bookworm'
win_dir: 'c:'
```

Je kan de single quotes ook vervangen door double quotes:

```
version: "v1.12.2: Bookworm"
win_dir: "c:"
```

Het verschil tussen het gebruik van single en double quotes is dat je *escape karakters* kan gebruiken bij double quotes. Bijvoorbeeld:

```
foo: "a \t TAB and a \n NEWLINE"
```

De \t en \n gaan effectief vervangen worden door een tab en een nieuwe regel in de uiteindelijke string. Voor een volledige lijst van geldige escape karakters, zie de YAML specificatie: https://yaml.org/spec/1.2.2/#57-escaped-characters

3.6.3.3 COMPLEXE VARIABELEN

Booleans

Ansible aanvaardt heel wat mogelijkheden om een boolean waarde te definiëren. In de officiële documentatie wordt vooral *true/false* gebruikt, maar weet dat de volgende waarden ook geldig zijn:

Geldige waarde	Beschrijving
True , 'true' , 't' , 'yes' , 'y' , 'on' , '1' , 1 , 1.0	Waar
False , 'false' , 'f' , 'no' , 'n' , 'off' , '0' , 0 , 0.0	Onwaar

Lijsten

Een lijst is een variabele dat meerdere waardes bevat. Je kan die als een lijst met items definiëren of tussen vierkante haakjes [ ]. Deze twee vormen gaan exact dezelfde lijst vormen:

```yaml
apps:
  - api
  - auth
  - backend
  - frontend

apps: [api, auth, backend, frontend]
```

Dezelfde YAML regels dienen hier gerespecteerd te worden. Bijvoorbeeld in dit voorbeeld zal je een foutmelding krijgen op regel 2:

```yaml
apps:
  - api-{{ api_version }} # Geldig
  - {{ auth_version }}-auth # Foutief gevormd
```

```
  - backend
  - frontend
```

Zoals we ondertussen weten, kunnen we dit als volgt aanpassen:

```
apps:
  - api-{{ api_version }} # Geldig
  - "{{ auth_version }}-auth" # Geldig
  - backend
  - frontend
```

Om je YAML overzichtelijk te houden, kan je ervoor kiezen om in zo'n geval al je waarden tussen quotes te zetten.

Om de waarden uit een lijst te gebruiken in je Playbooks, kan je de index gebruiken:

```
apps:
  - api # index 0
  - auth # index 1
  - backend # index 2
  - frontend # index 3

backend-app: apps[2]
```

Dictionaries

Een dictionary bevat waarden die ook uit key-value paren bestaan. Dit in tegenstelling tot een lijst die eenvoudige waarden (tekst, getallen, booleans, …) bevat.

```
foo:
  field1: one
  field2: two
```

De waarde uit die dictionary halen kan je via de 'bracket [ ]' of 'dot .' notatie doen:

```
foo.field1 # dot notatie
foo['field1'] # bracket notatie
```

We gaan verder de bracket notatie gebruiken omdat die in alle gevallen zal werken.

3.6.3.4 VARIABELEN REGISTREREN

In sommige gevallen wil je waarden uit een task ophalen en gebruiken in een andere task. Dit kan je doen de output van een task op te slaan in een variabele met het *register* argument. Als je bijvoorbeeld wilt gaan kijken hoe lang je servers al actief zijn en de output weergeven kan dat als volgt:

```
- hosts: your_servers
```

```
  become: true

  tasks:
    - name: Get system uptime
      ansible.builtin.command: uptime
      register: uptime_output

    - name: Display system uptime
      ansible.builtin.debug:
        msg: "System uptime is {{ uptime_output['stdout'] }}"
```

De echte kracht in het registreren van variabelen op basis van output is dat je in playbooks bepaalde tasks afhankelijk kan maken. Onderstaand playbook zal enkel de task om updates te installeren uitvoeren als er effectief updates beschikbaar zijn:

```
- hosts: your_servers
  become: true

  tasks:
    - name: Check for pending updates
      ansible.builtin.command: apt list --upgradable
      register: update_output
      changed_when: update_output['stdout'] != ""

    - name: Display pending updates
      ansible.builtin.debug:
        msg: "Pending updates: {{ update_output['stdout'] }}"
      when: update_output['stdout'] != ""

    - name: Perform updates if pending
      ansible.builtin.apt:
        upgrade: yes
      when: update_output['stdout'] != ""
```

Merk op dat we het sleutelwoord *when* gebruiken om Ansible te laten beslissen of een bepaalde task al dan niet uitgevoerd moet worden. Deze *when* statements kan je behoorlijk uitgebreid en complex maken om exact te matchen met de situatie die je wilt.

Bovenstaande when statements zijn een voorbeeld van conditionals. Met een conditional kan bepaalde tasks laten uitvoeren zolang de voorwaarde voldaan is. Deze voorwaarden worden opgebouwd in een templating taal Jinja2. We gebruiken Jinja2 als templating engine in het onderdeel Roles, maar we bespreken het nu al eens om conditionals te kunnen opbouwen.

3.6.4 JINJA2

Jinja2 is een templating taal die gebruikt wordt in Ansible om voorwaarden op te stellen. Net zoals elke taal zijn er bepaalde regels die je moet volgen om geldige syntax te bouwen. In dit stuk overlopen we de belangrijkste mogelijkheden met een sterke focus op expressies.

3.6.4.1 OPERATOREN

Operatoren zijn speciale tekens die een bepaalde actie triggeren. Je hebt rekenkundige, logische, vergelijkings- en andere operatoren.

Rekenkundige operatoren

Operator	Doel
+	Telt twee waarden op. Kan ook gebruikt worden om twee strings aan elkaar te hangen (concateneren), maar dit is niet de aanbevolen manier. Zie de ~ operator iets verderop
-	Trekt de tweede waarde af van de eerste
*	Vermenigvuldigt twee waarden. Kan ook gebruikt worden om een stuk te herhalen: {{ '=' * 80 }} print 80 keer een = teken
/	Deelt twee getallen.
//	Deelt twee getallen en geeft enkel het resultaat voor de komma terug. {{ 20 // 7 }} = 2
%	Geeft de restwaarde van een deling. {{ 11 % 7 }} = 4
**	Gebruikt om machten te berekenen. {{ 2**3 }} = 8

Voorbeelden:

```
# Optellen
result: "{{ 5 + 3 }}" # 8

# Aftrekken
result: "{{ 10 - 4 }}" # 6

# Vermenigvuldigen
result: "{{ 6 * 7 }}" # 42

# Delen
result: "{{ 20 / 5 }}" # 4
```

Vergelijkingsoperatoren

Operator	Doel
==	Controleert of twee objecten gelijk zijn
!=	Controleert of twee object niet gelijk zijn
>	Waar als linkerkant groter is dan de rechterkant
>=	Waar als linkerkant groter of gelijk aan is dan de rechterkant
<	Waar als linkerkant kleiner is dan de rechterkant
<=	Waar als linkerkant kleiner of gelijk aan is dan de rechterkant

Voorbeelden

```
# Gelijk aan
result: "{{ 5 == 5 }}" # True

# Niet gelijk aan
result: "{{ 10 != 7 }}" # True

# Groter dan
result: "{{ 15 > 10 }}" # True

# Kleiner dan
result: "{{ 25 < 25 }}" # False

# Groter dan of gelijk aan
result: "{{ 30 >= 30 }}" # True

# Kleiner dan of gelijk aan
result: "{{ 60 <= 50 }}" # False
```

Logische operatoren

Om meerdere expressies aan elkaar te koppelen kan je gebruik maken van logische operatoren.

Operator	Doel
and	Waar als links én rechts ook waar zijn
or	Waar als links of rechts waar zijn
not	Draait de logica om
(expr)	Groepeert een expressie. Handig in combinatie met not bv.

Voorbeelden

```
# AND operator
result: "{{ True and False }}" # False

# OR operator
result: "{{ True or False }}" # True

# NOT operator
result: "{{ not True }}" # False
```

Andere operatoren

Operator	Doel
in	Controleer of waarde in een lijst of dictionary zit
is	Voert een test uit (https://jinja.palletsprojects.com/en/3.1.x/templates/#filters)
|	Past een filter toe (https://jinja.palletsprojects.com/en/3.1.x/templates/#filters)
~	Gebruikt om strings samen te voegen
()	Gebruikt om een functie op te roepen
. / []	Dot of bracket notatie om een waarde uit een complex object te halen

Voorbeelden

```
# Concatenation
result: "{{ 'Hello ' ~ 'World' }}" # Hello World

# Contains
result: "{{ 'ansible' in 'I love ansible' }}" # true


# Gelijkheid van complexe objecten
vars:
  var1: [1, 2, 3]
  var2: [1, 2, 3]

tasks:
  - name: Controleer als var1 gelijk is aan var2
    debug:
      msg: "{{ var1 is var2 }}" # True
```

3.6.5 CONDITIONALS [14]

Nu we beter begrijpen hoe we expressies kunnen opbouwen, kunnen we begrijpen hoe in Ansible en Jinja2 expressies samenkomen om conditionals te maken.

In een vorig voorbeeld gebruikten we al een conditional om te kijken of een variabele niet leeg was:

```
    - name: Perform updates if pending
      ansible.builtin.apt:
        upgrade: yes
      when: update_output['stdout'] != ""
```

Met wat we nu weten van Jinja2 kunnen we veel complexere logica bouwen. Één van de meest voorkomende situaties is dat je bepaalde taken enkel wilt uitvoeren op een bepaald OS.

Je zou zelf kunnen proberen achterhalen welk OS er draait, dat registreren in een variabele en dan die variabele gebruiken (zie eerder voorbeeld van de software updates). Maar dat is behoorlijk wat werk dat eigenlijk niet nodig is. We kunnen de Ansible facts hiervoor gebruiken.

Herinner je dat Ansible facts een uitgebreid object is met allemaal metadata van de hosts waarmee Ansible communiceert. Dit object volledig bespreken zou ons veel te ver leiden, daarom bespreken we hieronder een aantal entries die vaak gebruikt worden in tasks.

Gestripte versie van ansible_facts:

```
"ansible_facts": {
    # Info over het OS zelf
    "ansible_architecture": "x86_64",
    "ansible_distribution": "Debian",
    "ansible_distribution_major_version": "12",
    "ansible_distribution_minor_version": "2",
    "ansible_distribution_release": "bookworm",
    "ansible_distribution_version": "12.2",
    "ansible_os_family": "Debian",

    # Welke shell wordt er gebruikt?
    "ansible_env": {
        "HOME": "/home/alex",
        "HOSTTYPE": "x86_64",
        "SHELL": "/bin/bash",
    },

    # Info over het netwerk
    "ansible_eth0": {
        "active": true,
        "device": "eth0",
        "ipv4": {
            "address": "172.18.35.169",
            "broadcast": "172.18.47.255",
            "netmask": "255.255.240.0",
            "network": "172.18.32.0",
            "prefix": "20"
        },
        "ipv6": [
            {
                "address": "fe80::215:5dff:fe3d:86b7",
                "prefix": "64",
            }
        ],
    },

    # Memory en CPU informatie
    "ansible_memtotal_mb": 15828,
    "ansible_processor_cores": 6,
```

```
    "ansible_processor_count": 1,
    "ansible_processor_nproc": 12,
    "ansible_processor_threads_per_core": 2,
    "ansible_processor_vcpus": 12,

    # Is selinux enabled?
    "ansible_selinux": {
        "status": "disabled"
    }
}
```

Hieronder volgen een paar voorbeelden hoe je deze facts kunt gebruiken in je tasks, maar laat dit je zeker niet beperken in je creativiteit. Er zijn nog veel uitgebreidere en complexere voorbeelden mogelijk. Merk wel op dat je de Ansible prefix mag weglaten in zo'n geval (bv. ansible_processor_count => ansible_facts['processor_count']).

Alle Debian hosts afsluiten:

```
tasks:
  - name: Shut down Debian flavored systems
    ansible.builtin.command: /sbin/shutdown -t now
    when: ansible_facts['os_family'] == "Debian"
```

Yum update uitvoeren, maar enkel op RHEL 9 hosts:

```
- name: Run yum update
  ansible.builtin.yum:
    name: "*"
    state: latest
  when: ansible_distribution == 'RedHat' and
ansible_distribution_major_version == '9'
```

Als je de 'and' operator gebruikt, dan kan je het when statement ook als lijst schrijven:

```
- name: Run yum update
  ansible.builtin.yum:
    name: "*"
    state: latest
  when:
    - ansible_facts['distribution'] == 'RedHat'
    - ansible_facts['distribution_major_version'] == '9'
```

Dat maakt langere conditionals een stuk leesbaarder.

Je kan gerust ook meermaals hetzelfde fact hergebruiken om een andere combinatie te verkrijgen:

```
- name: Shut down CentOS 6 and Debian 7 systems
  ansible.builtin.command: /sbin/shutdown -t now
  when: (ansible_facts['distribution'] == "CentOS" and
ansible_facts['distribution_major_version'] == "6") or
       (ansible_facts['distribution'] == "Debian" and
ansible_facts['distribution_major_version'] == "7")
```

In bovenstaand voorbeeld worden alle expressies binnen de ronde haakjes samengenomen. Je zou die lange expressie dus als deze pseudo-code kunnen schrijven: OS==Centos 6 or OS==RHEL 7

3.6.6 LOOPS [15]

Soms wil je bepaalde taken meerdere keren laten uitvoeren. Ansible voorziet hiervoor een lusmechanisme dat gebruik maakt van de keywords *loop*, *with_<lookup>* en *until*. Loop werd geïntroduceerd in Ansible versie 2.5 en zal voor de meeste use cases geschikt zijn. Voor de volledigheid overlopen we ook nog even de andere twee.

Met loop en with_<lookup> herhaal je een bepaalde task voor elk element dat je meegeeft. De belangrijkste verschillen tussen beide zijn:

- with_<lookup> maakt gebruik van plugins. Zo heb je with_items, with_lists, with_indexed_items, … Elke plugin verandert iets aan de manier waarop de lus zal werken.
- Het loop keyword zonder extra's komt overeen met with_items en wordt gebruikt voor eenvoudige loops.
- Meestal kan je with_<lookup> vervangen door loop [16]
- Het loop keyword aanvaardt lijsten, maar default geen strings als input. Je kan hier rond via de Jinja2 functie *query* [17]

Met een *until* loop kunnen we een task blijven uitvoeren totdat een bepaalde voorwaarde voldaan is. Dit kan handig zijn als je moet wachten op een externe component waar je geen controle over hebt. Een voorbeeld om het duidelijk te maken:

```
- name: Retry a task until a certain condition is met
  ansible.builtin.shell: /usr/bin/foo
  register: result
  until: result.stdout.find("all systems go") != -1
  retries: 5
  delay: 10
```

Deze lus zal het commando '/usr/bin/foo' tot 5 keer blijven herhalen totdat de output de tekst "all systems go" bevat. Tussen elke poging wacht Ansible 10 seconden.

Als je start met loops kan het soms verwarrend zijn wat je nu exact moet gebruiken. In deze cursus gaan we ons focussen op het gebruik van loop.

3.6.6.1 STANDAARD LOOPS

Een standaard loop zal itereren over een lijst van waarden. De task zal éénmaal uitgevoerd worden per entry in de lijst. Op die manier kan je bv. meerdere software packages in één task bundelen:

```yaml
- name: Install software packages
  hosts: your_hosts
  become: true  # If sudo privileges are required

  tasks:
    - name: Install packages
      ansible.builtin.package:
        name: "{{ item }}"
        state: present
      loop:
        - python3
        - firewalld
        - htop
        - vim
```

Je kan ook die lijst ergens anders definiëren als een var block en dan als variabele gebruiken binnen de task zelf:

```yaml
- name: Install software packages using a loop
  hosts: your_hosts
  become: true  # If sudo privileges are required

  vars:
    packages_to_install:
      - python3
      - firewalld
      - htop
      - vim

  tasks:
    - name: Install packages
      ansible.builtin.package:
        name: "{{ item }}"
        state: present
      loop: "{{ packages_to_install }}"
```

Deze laatste manier is natuurlijk beter als je die lijst meerdere keren wilt gebruiken.

3.6.6.2 LOOPS OVER COMPLEXE OBJECTEN

Je kan ook loopen over objecten met een complexe structuur zoals lijsten of dictionaries.

Loopen over een lijst:

```yaml
- name: Add several users
  ansible.builtin.user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
```

```
  loop:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

Loopen over een dictionary:

Hiervoor moeten we die dictionary wel nog omzetten naar een itemized list met de filter dict2items [18] zodat loop zijn werk kan doen.

```
- name: Looping over a dictionary using dict2items

  vars:
    tag_data:
      Environment: dev
      Application: payment

  ansible.builtin.debug:
    msg: "{{ item.key }} - {{ item.value }}"
  loop: "{{ tag_data | dict2items }}"


# Output
Environment - dev
Application - payment
```

Je kan zelfs loopen over geneste lijsten zolang die syntactisch kloppen met wat de module verwacht:

```
- name: Create users with different attributes using nested lists
  hosts: your_hosts
  become: true  # If sudo privileges are required

  vars:
    user_info:
      - name: user1
        groups:
          - group1
          - group2
        shell: /bin/bash
        home: /home/user1
      - name: user2
        groups:
          - group2
          - group3
        shell: /bin/zsh
        home: /home/user2

  tasks:
    - name: Create users with different attributes
```

```
    ansible.builtin.user:
      name: "{{ item.name }}"
      groups: "{{ item.groups }}"
      shell: "{{ item.shell }}"
      home: "{{ item.home }}"
      state: present
    loop: "{{ user_info }}"
```

3.6.7 ROLES [19]

Met playbooks kunnen we al behoorlijk wat structuur verkrijgen in onze Ansible configuratie. We ontbreken enkel nog een stuk modulariteit als we bijvoorbeeld verschillende playbooks hebben:

- Webservers.yml
- Databases.yml
- Fileservers.yml

Stel dat je elke playbook start met dezelfde acties:

- Algemene OS-settings zoals timezone, hostname
- Configureren van IP-adressen
- Instellen van remote access
- User accounts aanmaken voor remote management
- Updaten van de package manager
- Installeren van algemene systeemsoftware: logging agent, firewall, …
- …

Dan zou je deze acties in alle drie de playbooks telkens moeten herhalen. Copy pasten is niet veel werk, maar het onderhouden van dezelfde taken kan dat wel snel worden. Om het DRY-principe (Don't Repeat Yourself) ook hier te gebruiken, splitsen we deze taken beter af in een herbruikbare component. Roles zijn de Ansible oplossing om zo'n componenten te bouwen.

3.6.7.1 OPBOUW VAN EEN ROLE

Een role bestaat uit een vooraf bepaalde structuur die toelaat om automatisch bestanden, tasks, handlers en andere Ansible objecten in te laden. Dit gebeurt automatisch zolang je de opgelegde mappenstructuur respecteert. Die structuur overlopen we hieronder, maar ziet er alvast zo uit:

```
roles/
    common/               # Alles hieronder stelt een role voor
        tasks/            #
            main.yml      # Bestand met alle taken
        handlers/         #
            main.yml      # Bestand met "handlers"
        templates/        # Bestanden om te gebruiken met de template
            ntp.conf.j2   # Template file in Jinja2
        files/            # Bestanden om te gebruiken in je tasks
            bar.txt       # Tekstfile om bv. te kopiëren
            foo.sh        # Scriptfile om te laten uitvoeren op de host
```

```
        vars/               #
            main.yml        # Variabelen die gekoppeld zijn aan deze role
        defaults/           #
            main.yml        # Lagere prioriteitsvariabelen voor deze role
        meta/               #
            main.yml        # Role dependencies
        library/            # Custom modules
        module_utils/       # Custom module_utils
        lookup_plugins/     # Roles kunnen ook andere plugins bevatten, zoals
lookup in dit geval

    webtier/                # Tweede rol, zal zelfde structuur volgen als
“common” hierboven
    monitoring/             # ""
    fooapp/                 # ""
```

Op het top level start je met de “Roles” map. Hierin maak je een map per role. De submappen van de role zijn als volgt:

- tasks: deze bevat één of meerder yaml-bestanden die de stappen bevatten die verwacht van deze role.
- handlers: Dit zijn taken die je kan definiëren die enkel uitgevoerd worden wanneer die opgeroepen worden vanuit andere tasks. Een voorbeeld om het duidelijk te maken:
  In de handlers map maak je een main.yaml bestand aan met deze inhoud:

```yaml
# handlers/main.yml
- name: Restart SSH service
  ansible.builtin.service:
    name: sshd
    state: restarted
```

In de tasks map heb je een main.yaml met deze inhoud:

```yaml
# tasks/main.yml
- name: Copy SSH configuration file
  ansible.builtin.copy:
    src: files/sshd_config
    dest: /etc/ssh/sshd_config
  notify: Restart SSH service # match met naam van de handler
```

Met deze configuratie zal de handler “Restart SSH sevice” opgeroepen worden na het kopiëren van het “sshd_config” bestand uit de “files” map in deze role naar de destination map op de host.

Merk op dat je op deze manier veel voorkomende stukken logica kan hergebruiken in plaats van telkens te herhalen.

- templates: Deze map bevat bestanden in de Jinja2 template taal. Met deze bestanden kan je dus templates maken van bestanden die je aanvult met Ansible variabelen voordat die doorgestuurd worden naar de hosts. Je maakt als het ware een blueprint van een bestand dat je uniek kan maken door de variabelen die je gebruikt. Een voorbeeld van een template bestand ziet er als volgt uit:

```
# templates/app_config.conf.j2

# Application Configuration File

[App]
app_name = {{ app_name }}
app_version = {{ app_version }}
debug_mode = {{ debug_mode }}
```

De variabelen, te herkennen aan de dubbelle accolades, “app_name”, “app_version” en “debug_mode” kunnen dan vervangen worden via waardes in vars/main.yml (zie verder) of doorgegeven worden via het playbook. Je gebruikt bovenstaande template in een task.

```
# tasks/main.yml

- name: Generate application configuration file
  ansible.builtin.template:
    src: templates/app_config.conf.j2
    dest: /path/to/application/app_config.conf
```

Deze voorbeelden zijn eenvoudig gehouden om het principe van template bestanden aan te tonen, maar weet dat je Jinja2 expressies kan toevoegen om je templating tot een hoger niveau te tillen zoals we eerder zagen.

- files: In deze map plaats je bestanden die gebruikt worden door je Ansbile role. Dit kan van alles zijn: een configuratiebestand, een SSH-public key, een tekstbestand, een script, …

- vars: In de vars map kan je, zoals de naam al doet vermoeden, variabelen definiëren die je verder gebruikt in je tasks. De bedoeling hiervan is om je role config zo generiek mogelijk te houden. Hiermee bedoelen we dat je geen host-specifieke waarden in je tasks, templates of handlers zet. Als je dit goed doet, maak je een volledig herbruikbare role. Een voorbeeld om het tastbaar te maken:

  Stel dat je een webserver wilt configureren/beheren via Ansible. Je maakt een role aan om de tasks te bundelen

```
# tasks/main.yml
```

```
- name: Install Nginx
  ansible.builtin.apt:
    name: nginx

- name: Ensure web server is running
  ansible.builtin.service:
    name: nginx
    state: started

- name: Configure web server
  ansible.builtin.template:
    src: templates/nginx.conf.j2
    dest: "{{ web_server_root }}/nginx.conf"

- name: Ensure configuration file permissions are set
  ansible.builtin.file:
    path: "{{ web_server_root }}/nginx.conf"
    owner: www-data
    group: www-data
    mode: '0644'
```

In dit bestand gebruiken we de variabele “web_server_root” twee keer. Er wordt ook gebruik gemaakt van een template voor de nginx web server configuratie.

template/nginx.conf.j2 bevat deze inhoud:

```
# templates/nginx.conf.j2

# Nginx Configuration File

server {
    listen {{ web_server_port }};
    server_name {{ web_server_domain }};

    root {{ web_server_root }};
    index index.html index.htm;

    # Other configurations...
}
```

Zoals je ziet staan er zowel in de template file als in tasks\main.yml variabelen die invloed hebben op de werking van het playbook. In plaats van de echte waarde op meerdere plaatsen bij te houden, bundelen we die in een vars\main.yml bestand:

```
# vars/main.yml
```

```yaml
# Web Server Settings
web_server_port: 80
web_server_domain: "example.com"
web_server_root: "/var/www/html"
```

Op die manier kan je heel eenvoudig bv de poort waarop je webserver luistert aanpassen zonder dat je door al je Ansible bestanden moet gaan en overal de waarde aanpast. Het gebruik van variabelen vraagt initieel wat meer werk om alles af te splitsen, maar leidt gegarandeerd tot snelheidswinst achteraf en duidelijkheid in je playbooks.

- defaults: In deze map kan je ook een bestand met variabelen zetten, maar die hebben een lagere prioriteit. Dit wil zeggen dat vanaf één van de variabelen in het defaults\main.yml bestand overschreven wordt op een andere plaats, die laatste waarde gebruikt wordt. Deze map dient eigenlijk om een soort fallback mechanisme in te bouwen. Als je vergeet één van de variabelen in te stellen, blijft het playbook wel nog gewoon werken.

  Een andere gebruiksmogelijkheid is als je heel veel inzet op Ansible binnen je organizatie, dan kan je roles ergens centraal gaan bundelen. Dan is het defaults\main.yml bestand handig om te weten welke variabelen verwacht worden. Anders zou je door alle bestanden in de role moeten gaan om diezelfde informatie te verkrijgen.

- meta: In deze map staat informatie over de role zelf. Dit kan gaan over dependencies (andere roles) of op welke platformen dit role zal werken. Een voorbeeld:

```yaml
# meta/main.yml

dependencies:
  - role: common_tasks
  - role: web_server
    vars:
      web_server_port: 8080
  # Other role dependencies...
```

Deze role heeft twee andere roles als dependencies. Door andere roles als dependencies op te lijsten, zorg je ervoor dat Ansible altijd de volgorde van uitvoering respecteert. Zo zal in bovenstaand voorbeeld deze rol pas uitgevoerd worden als de role “common_tasks” én de role “web_server” uitgevoerd werden. Merk ook op dat je via dit bestand “dependency variabelen” kan meegeven. In dit geval de web_server_port aan de role web_server.

- Library, module_utils en lookup_plugin: deze mappen dienen om je role verder uit te breiden met custom modules (library), python scripts gebruikt door die modules (module_utils) of door extra data op te halen (lookup_plugins).

Als je bepaalde mappen niet nodig hebt, mag je die weglaten in jouw configuratie. Elke setup is uniek dus pas aan wat goed voelt in jouw situatie.

Als je deze folderstructuur automatisch wil laten generen, kan dat via het **ansible-galaxy** commando:

```
$ ansible-galaxy init my_role
```

3.6.7.2 USE CASE

Nu we weten hoe een role opgebouwd wordt, kunnen we aantonen waarom die zinvol zijn. Laten we vertrekken vanaf dit playbook:

```yaml
---
- name: Complete Server Setup
  hosts: all
  become: true
  gather_facts: true

  tasks:
    - name: Update apt cache and upgrade packages
      ansible.builtin.apt:
        update_cache: yes
        upgrade: yes

    - name: Install required packages
      ansible.builtin.apt:
        name: "{{ item }}"
        state: present
      loop:
        - nginx
        - python3
        - python3-pip
        # Other packages...

    - name: Configure nginx
      ansible.builtin.template:
        src: templates/nginx.conf.j2
        dest: /etc/nginx/nginx.conf

    - name: Clone application repository
      ansible.builtin.git:
        repo: https://github.com/example/myapp.git
        dest: /opt/myapp
        version: master

    - name: Install application dependencies
      ansible.builtin.pip:
        requirements: /opt/myapp/requirements.txt

    - name: Configure application settings
      ansible.builtin.template:
        src: templates/app_config.ini.j2
        dest: /opt/myapp/config.ini
```

```
    - name: Ensure application service is running
      ansible.builtin.systemd:
        name: myapp
        state: started
        enabled: yes
```

Hoewel dit functioneel perfect werkt, hebben we alle tasks in één groot bestand gezet. Iets wat snel onoverzichtelijk kan worden. Als we met roles werken ziet onze playbook er zo uit:

```
---
- name: Complete Server Setup using Roles
  hosts: all
  become: true
  gather_facts: true

  roles:
    - common
    - app_deployment
    # Other roles...
```

Geen tasks rechtstreeks in het playbook, maar enkel referenties naar de roles. De role common bevat dan deze logica:

```
# roles/common/tasks/main.yml

- name: Update apt cache and upgrade packages
  ansible.builtin.apt:
    update_cache: yes
    upgrade: yes

- name: Install required packages
  ansible.builtin.apt:
    name: "{{ item }}"
    state: present
  loop:
    - nginx
    - python3
    - python3-pip
    # Other packages...

- name: Configure nginx
  ansible.builtin.template:
    src: templates/nginx.conf.j2
    dest: /etc/nginx/nginx.conf
```

en de app_deployment role deze :

```
# roles/app_deployment/tasks/main.yml

- name: Clone application repository
  ansible.builtin.git:
    repo: https://github.com/example/myapp.git
    dest: /opt/myapp
    version: master

- name: Install application dependencies
  ansible.builtin.pip:
    requirements: /opt/myapp/requirements.txt

- name: Configure application settings
  ansible.builtin.template:
    src: templates/app_config.ini.j2
    dest: /opt/myapp/config.ini

- name: Ensure application service is running
  ansible.builtin.systemd:
    name: myapp
    state: started
    enabled: yes
```

Zoals je ziet verleggen we het inhoudelijke werk van het playbook naar de roles (in de map tasks). Op die manier kan elke rol op zichzelf bestaan en bevat die enkel de logica om één doel te bereiken. Dit maakt je configuratie veel overzichtelijker en maakt elke rol op zich eenvoudiger.

Het opdelen in rollen kan in het begin wat moeilijk zijn. Waar leg je de grens tussen de verschillende functionele onderdelen? Behoort een bepaalde taak tot role X of Y? Er zijn hiervoor geen gouden regels die je in elke situatie kan toepassen, maar hou onderstaande tips in het achterhoofd en je komt al heel ver.

3.6.7.3 OPDELING

Modulair en herbruikbaar

Een rol moet functionaliteit bevatten dat door meerdere playbooks kan gebruikt worden. Door een role generiek te maken, kan je dezelfde configuratiebestanden meerdere keren hergebruiken. Via variabelen pas je het resultaat aan zoals gewenst.

Duidelijk en afgelijnd

Een rol moet één duidelijk doel hebben. Een rol web_server beperkt zich dan ook tot tasks die dienen om de web server werkend te krijgen. Vermijd te grote, moeilijke roles die meerdere blokken proberen te combineren. Het is beter om veel kleine roles te hebben dan een paar grotere. Via de naamgeving kan je het doel ook afbakenen voor jezelf zodat het duidelijk is wat in deze role past en wat niet.

- roles\web_server
- roles\database_server
- roles\app_deployment
- roles\app_monitoring_tools
- …

Vermijd dubbel werk

Als je een bepaalde tasks meerdere keren uitvoert in verschillende playbooks of roles, dan is de kans groot dat je dat stukje logica nog eens kan afsplitsen.

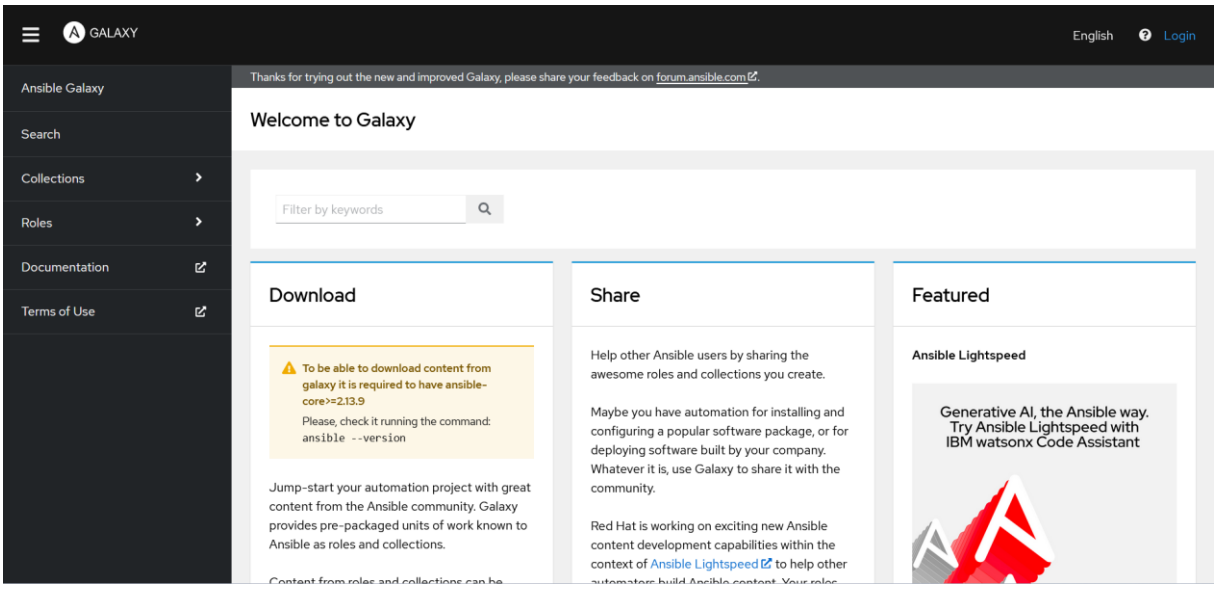
3.6.7.4 ANSIBLE GALAXY

Ansible Galaxy is een gratis website waarop je onder andere roles kan vinden gebouwd door de Ansible community. Deze roles worden ook via een rating systeem beoordeeld zodat je een beter zicht krijgt op de kwaliteit.

Zoals bij alles open source, is het aan jou om te beoordelen of je een bepaalde component wil inbouwen in je workflow of niet.

Een Galaxy role installeren doe je via:

```
$ ansible-galaxy collection install
```



Figuur 6 Ansible Galaxy website

3.7 MASTERING ANSIBLE

3.7.1 ANSIBLE LINT

Ansible Lint is een command-line tool die playbooks, roles en andere Ansible resources gaat linten (opmaken en fouten detecteren) volgens heel wat vooringestelde regels. Het doel is om een gestandaardiseerde opbouw van Ansible config te krijgen om fouten en bugs te vermijden. Deze regels werden opgebouwd als een community project en vallen nu onder Ansible Galaxy.

Je kan Ansible Lint installeren via je package manager of via pip3.

```
$ dnf install ansible-lint
$ apt install ansible-lint
$ pip3 install ansible-lint
```

Controleer je installatie via:

```
$ ansible-lint –version
```

Ansible Lint maakt gebruik van profielen om je configuratie te controleren of je de vooropgestelde best practices wel gebruikt. Een eenvoudige test van bv. een playbook is:

```
$ ansible-lint playbook.yml
```

Om te zien welke profielen en regels er gebruikt worden kan je deze opties gebruiken:

```
# Toon alle profielen
$ ansible-lint -P

# Toon alle regels
$ ansible-lint -L
```

Je kan Ansible Lint gebruiken lokaal op je Control Node of toevoegen aan deploy flow zoals in een CI/CD pipeline.

3.7.2 BECOME [20]

Doorheen dit document hebben we hier en daar gebruik gemaakt van het *become* keyword of plaatsten we *become=true* in onze playbooks. In dit onderdeel bekijken we hoe dit keyword effectief werkt.

Via het become keyword kan je aan privilege escalation doen. Dit betekent dat je meer rechten krijgt dan wat je normaal hebt onder de huidige gebruiker. In de linux wereld vertaalt dit gedrag zich in meerdere oplossingen, afhankelijk van de distro waarop je werkt. Sommige van onderstaande tools zouden toch bekend in de oren moeten klinken:

- sudo
- su
- pfexec
- doas
- pbrun
- dzdo
- ksu
- …

Je kan het become-gedrag echter zelf gaan aanpassen naar je noden. Dit kan je zelfs doen per play of task. Standaard zal het toevoegen van become je root-rechten geven. Via *become_user* kan je zelf een gebruiker opgeven. Op die manier kan je een bepaalde task onder een specifieke gebruiker laten draaien.

Hieronder vind je een overzicht van de parameter die het become-gedrag beïnvloeden.

- **become**: op true zetten schakelt dit mechanisme in
- **become_user**: bepaalt welke user je gebruikt. Opgelet: dit schakelt het become-gedrag niet automatisch in dus zal je ergens anders become:true moeten zetten. Standaarduser is root

- **become_method**: hiermee kan je bepalen welke tool je wilt gebruiken om privilege escalation te doen.
- **become_flags**: Voegt extra parameters toe aan je become_method

Volgend voorbeeld runt een commando als de user *nobody* met een custom shell:

```
- name: Run a command as nobody
  command: somecommand
  become: true
  become_method: su
  become_user: nobody
  become_flags: '-s /bin/sh'
```

Om aan privilege escalation te kunnen doen, moet de user waaronder Ansible verbinding maakt uiteraard de nodige rechten daarvoor hebben. Dat is de reden waarom we initieel de Ansible user als sudo user moesten configureren zonder wachtwoord. Als je dit gedrag toch wilt gaan beveiligen met een wachtwoord, kan dat natuurlijk ook. Het enige wat je nodig hebt is een user met privilege escalation rechten en de parameter **--ask-become-pass** toevoegen aan je command-line.

3.7.3 INFRASTRUCTUUR ONDER VERSIEBEHEER

Naarmate je Ansible configuratie groeit en je misschien met meerdere mensen samen wilt gaan werken aan playbooks groeit de noodzaak om je config files onder versiebeheer te plaatsen.

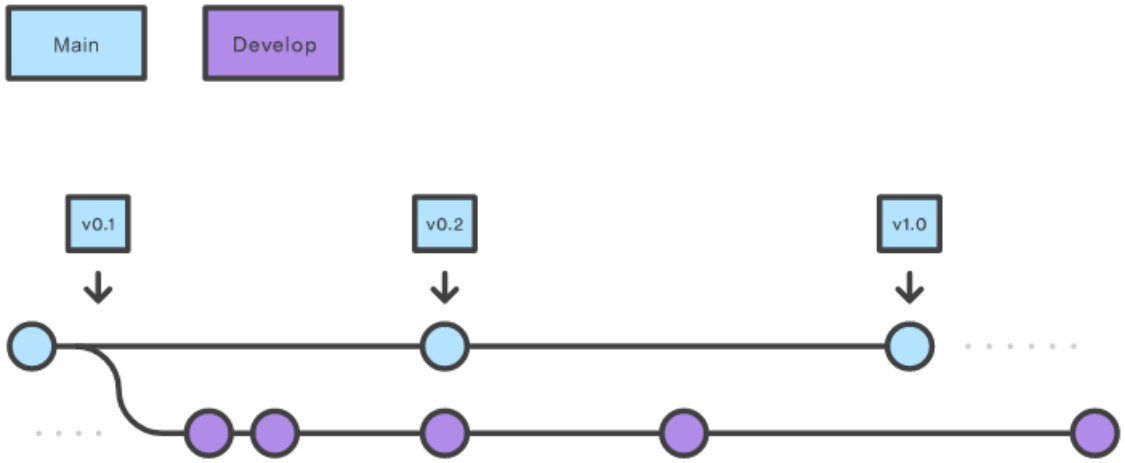
De industriestandaard voor versiebeheer is Git. Hoewel dit geen cursus 'hoe gebruik ik git', moet je toch een paar basisconcepten begrijpen zodat je deze tool kan inzetten om je bestanden in te bewaren.

De hoofddoelstellingen van het gebruik van versiebeheer voor infrastructuur configuratie is om:

- Traceerbaarheid: wie paste wat aan, wanneer?
- Rollback: als er iets misgaat kan je in de meeste gevallen eenvoudig terugkeren naar een vorige staat
- Single source of truth: enkel wat onder versiebeheer zit, telt. Manuele ingrepen kunnen én zullen worden overschreven.

3.7.3.1 GITFLOW

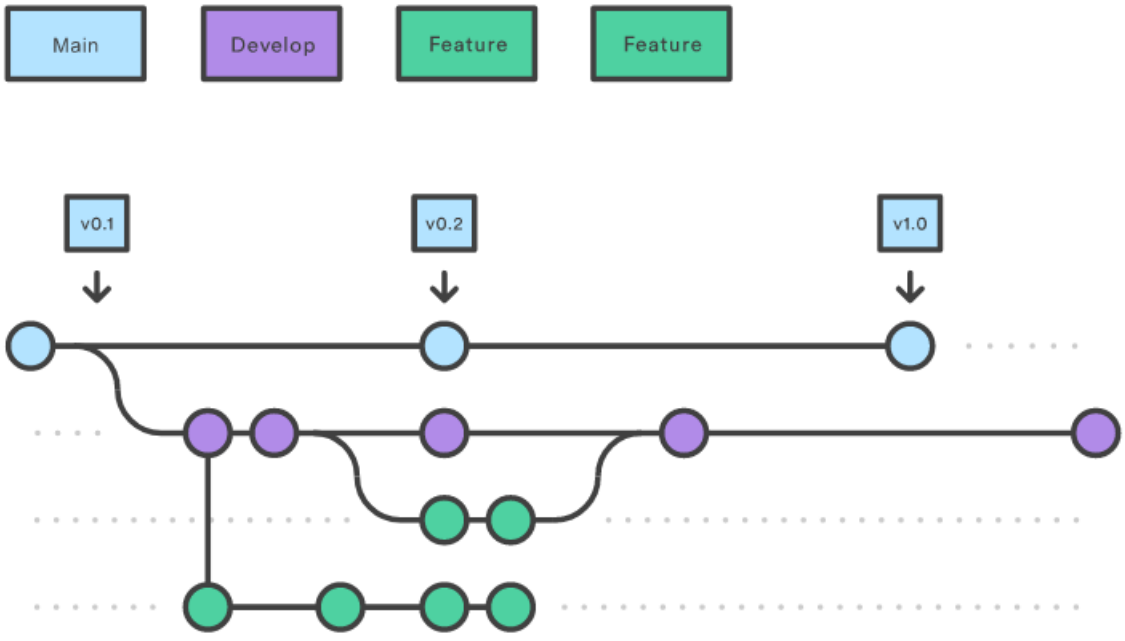
Er zijn veel verschillende strategiën die je kan volgen om git in te zetten in combinatie met Ansible. Je kan werken met GitFlow dat meerdere branches gebruikt om 'releases' te maken. Een release kan je in deze context beschouwen als een stabiele vorm van je configuratie. Praktisch ziet dat er zo uit:



Figuur 7 Gitflow met main en develop branch [21]

Je hebt een main-branch waarin de stabiele versie bijgehouden worden en eventueel van een versienummer voorzien wordt. In een aparte branch, hier develop genaamd, maak je wijzigingen en test je nieuwe roles of playbooks uit. Eens die stabiel en getest zijn, kan je die mergen naar de main branch.

Als je terechtkomt in een groot team waar verschillende mensen in dezelfde repository werken, is het vaak niet genoeg om samen in een develop-branch te werken omdat je dan tegelijkertijd wijzigingen aan dezelfde bestanden moet maken. In zo'n geval is het beter om te werken met feature branches waarin je kleine toevoegingen ontwikkelt en die heel frequent merget in de develop-branch eens die stabiel zijn. In schema-vorm ziet dat er als volgt uit:



Figuur 8 Gitflow met Feature Branches [21]

Mergen van verschillende branches kan je eenvoudig doen met het **git merge** commando. Stel dat de wijzigingen uit de develop-branch willen samenvoegen (mergen) met de main-branch dan ziet dat commando er zo uit:

```
# Dubbelchecken of we op de main-branch zitten:
$ git branch -v

# Indien niet op main, switchen naar die branch:
$ git checkout main

# develop mergen in main
$ git merge develop
```

3.7.3.2 PULL REQUESTS

Pull requests zijn een manier om gecontroleerd te gaan mergen. Door te werken via pull requests kan je extra controles toevoegen zoals:

- Een collega/manager die de wijzigingen moet goedkeuren voor de merge
- Testen van je configuratie op fouten of opmaakinconsistenties (denk aan Ansible-Lint)
- Integratie met bots die allerlei acties uitvoeren (meldingen versturen, tickets bewerken, issues afsluiten, …)

Pull requests zijn beschikbaar op platformen zoals GitHub, GitLab, BitBucket, … en voegen functionaliteit toe aan de basistoolset van git.

3.7.4 ANSIBLE CONFIGURATION SETTINGS [22]

Tot nu toe hebben we Ansible out-of-the-box gebruikt zonder iets te wijzigen aan de standaardinstallatie van onze Control Node. We hebben verschillende manieren om onze installatie te tweaken:

- Het ansible.cfg configuratiebestand
- Environment variabelen
- Opties via de CLI doorgegeven
- Playbook variabelen en keywords

Om te zien welke configuratieopties momenteel gebruik worden, voer dit commando uit:

```
$ ansible-config list
```

Je zal zien dat de lijst met gebruikte configuratieopties behoorlijk lang is. Het is zeker niet de bedoeling dat je deze opties allemaal beheerst, maar wel dat je weet waar je dingen kan aanpassen.

3.7.4.1 ANSIBLE.CFG

Ansible gaat op verschillende manieren proberen naar dit bestand te zoeken om bijkomende settings in te laden. Die manieren zijn:

- De environment variabele ANSIBLE_CONFIG bevat de locatie naar dit bestand
- ansible.cfg (in huidige directory)
- ~/.ansible.cfg (in de home directory van deze user)
- /etc/ansible/ansible.cfg

Ansible zal bovenstaande lijst van boven naar onder verwerken en de settings samenvoegen. Let dus op moest je op meerdere locaties een ansible.cfg bestand hebben. Op die manier kan je settings definiëren system-wide, op user niveau en per directory. Dit kan handig zijn als je bepaalde settings wilt gaan overschrijven en niet system-wide wilt definiëren. Als je een configuratiebestand wilt genereren met alle regels in commentaar kan dat via dit commando:

```
$ ansible-config init --disabled > ansible.cfg
```

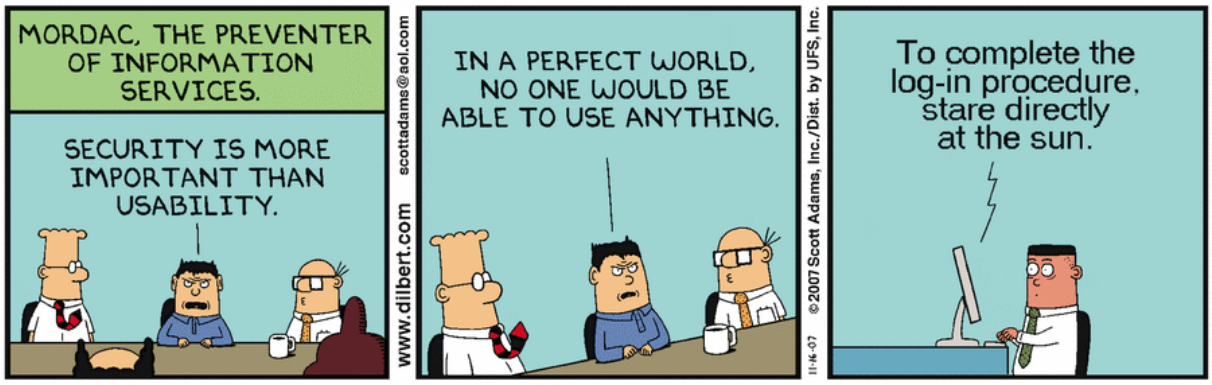
Deze uitgebreide lijst van opties kan dienen als basis voor je custom configuratie. We overlopen enkele opties om het nut van deze file aan te tonen hieronder.

Parameter	Doel
DEFAULT_BECOME_METHOD	Privilege escalation methode. Default waarde is sudo
DEFAULT_BECOME_USER	Default user is root
DEFAULT_FORKS	Aantal parallelle handlers (forks) dat Ansible gebruikt
DEFAULT_PRIVATE_KEY_FILE	Locatie van de key file dat gebruikt wordt voor de (Open)SSH-verbinding

Een lijst van de meest gebruikte opties vind je hier:
https://docs.ansible.com/ansible/latest/reference_appendices/config.html#common-options

3.7.5 ANSIBLE VAULT [20]

Als je infrastructuur beheert met een IaC of Config Management tool als Ansible, kom je onvermijdelijk terecht in situaties waarbij je gevoelige informatie nodig hebt in je automatisatieproces. Denk aan wachtwoorden van gebruikers, SSH-keys die geïmporteerd moeten worden, API-keys of andere gevoelige informatie. Omgaan met gevoelige data in een automatisch systeem is altijd een compromis zoeken tussen veiligheid en gebruiksgemak.



Ansible Vault geeft ons de mogelijkheid om gevoelige informatie te beschermen door middel van encryptie. Hiervoor wordt een het krachtige AES-256 encryptieprotocol gebruikt. Zoals de naam het zelf aangeeft, is dit een beveiligde kluis waarin je informatie kan opslaan. Deze kluis wordt beveiligd met één of meerdere wachtwoorden. Om dit principe niet te complex te maken gaan we ervan uitgaan dat we één wachtwoord gebruiken om alle data te beveiligen.

3.7.5.1 ENCRYPTEREN EN DECRYPTEREN

Bij het versleutelen van een bestand met Ansible Vault zal gevraagd worden om een wachtwoord in te stellen. Zorg dat je dit wachtwoord niet vergeet want zonder dit wachtwoord zal het bestand niet meer leesbaar zijn.

Om bijvoorbeeld een bestaand bestand geheim.yaml te encrypteren, gebruik je volgend commando:

```
$ ansible-vault encrypt geheim.yaml
```

Het is ook mogelijk om via Ansible Vault een nieuw geëncrypteerd bestand aan te maken:

```
$ ansible-vault create geheim.yaml
```

Als dit bestand toch niet geëncrypteerd moest zijn, dan kan je dit ongedaan maken met:

```
$ ansible-vault decrypt geheim.yaml
```

3.7.5.2 GEËNCRYPTEERDE BESTANDEN BEKIJKEN/WIJZINGEN

Omdat het nogal omslachtig zou zijn om bij elke wijziging de bestanden te moeten decrypteren en opnieuw te encrypteren met een paswoord, kunnen we de bestanden via Ansible Vault rechtstreeks bekijken en aanpassen.

Een geëncrypteerd bestand lezen:

```
$ Ansible-vault view geheim.yaml
```

Een geëncrypteerd bestand bewerken:

```
$ ansible-vault edit geheim.yaml
```

Het wachtwoord aanpassen:

```
$ ansible-vault rekey geheim.yaml
```

3.7.5.3 VARIABELEN ENCRYPTEREN

Stel dat we in onze inventory hosts.ini het root paswoord willen instellen voor alle Ubuntu apparaten. Zonder Ansible Vault zou je dit in je inventory toevoegen:

```
[ubuntu:vars]
ansible_become=yes
ansible_become_password='hunter2'
```

In principe zou je dan het volledige inventory bestand met de Vault kunnen encrypteren, maar we kunnen ook enkel het paswoord encrypteren. Hiervoor zullen we een apart bestand maken waarin we het wachtwoord opslaan onder een variabele waarnaar we kunnen verwijzen.

We doen dit in 3 stappen:

1. Het paswoordbestand geheim.yaml aanmaken.

Hierin maken we een variabele aan voor het wachtwoord. De naam van de variabele mag je zelf kiezen, in dit voorbeeld zullen we de variabele ubuntu_become_pass gebruiken.

```
$ ansible-vault create geheim.yaml
```

Inhoud geheim.yaml:

```
ubuntu_become_pass: "hunter2"
```

2. De variabele toevoegen aan de inventory hosts.ini.

```
[ubuntu:vars]
ansible_become=yes
ansible_become_password='{{ ubuntu_become_pass }}'
```

3. Het paswoordbestand importeren in je playbook main.yaml. Dit doen we met behulp van vars_files.

```
- name: Ansible test playbook
  hosts: ubuntu

  vars_files:
    - geheim.yaml

  tasks:
    - name: ...
```

Om het playbook te kunnen uitvoeren, zullen we nu de parameter --ask-vault-pass moeten meegeven aan het ansible-playbook commando. Dit zorgt ervoor dat Ansible het Vault wachtwoord vraagt voordat het playbook uitgevoerd wordt. Als je dit niet meegeeft, dan zal Ansible een foutmelding genereren omdat het bestand geheim.yaml niet gedecrypteerd kan worden.

```
$ ansible-playbook main.yaml --ask-vault-pass
```

Playbook uitvoeren zonder --ask-vault-pass

Als we de --ask-vault-pass vlag niet telkens willen meegeven bij het uitvoeren van paybooks, dan kunnen we de volgende lijn toevoegen aan ansible.cfg:

ask_vault_pass = true

Dit zal ervoor zorgen dat er bij uitvoering van elk playbook automatisch gevraagd wordt naar het vault wachtwoord.

Om te voorkomen dat we telkens het wachtwoord moeten ingeven, kunnen we ook werken met een vault password file. Hiervoor maken we een apart bestand aan dat enkel het vault wachtwoord bevat. Stel de permissies voor dit bestand zo in dat enkel jij dit kan uitlezen.

```
$ vim ~/.vault_pass

$ chmod 600 ~/.vault_pass
```

Nu moeten we ansible.cfg nog aanpassen zodat er verwezen wordt naar dit bestand. Verwijder de lijn met ask_vault_pass = true en voeg bij defaults de volgende lijn toe:

vault_password_file = ~/.vault_pass

Hierna kunnen we het playbook gewoon uitvoeren zonder extra vlaggen:

```
$ ansible-playbook main.yaml
```

3.7.6 CONNECTION PLUGINS

Tot nu toe gebruikten we enkel (Open)SSH als connectiemogelijkheid (connection plugin) om verbinding te maken naar de hosts. Er bestaan echter nog verschillende mogelijkheden. Om te zien welke plugins er ingeschakeld zijn in jouw installatie, voer dit commando uit:

```
$ ansible-doc -t connection -l
```

Voorbeeldoutput :

```
ansible.builtin.local           execute on controller
ansible.builtin.paramiko_ssh    Run tasks via python ssh (paramiko)
ansible.builtin.psrp            Run tasks over Microsoft PowerShell Remoting Protocol
ansible.builtin.ssh             connect via SSH client binary
ansible.builtin.winrm           Run tasks over Microsoft's WinRM
ansible.netcommon.grpc          Provides a persistent connection using the gRPC protocol
ansible.netcommon.httpapi       Use httpapi to run command on network appliances
ansible.netcommon.libssh        Run tasks using libssh for ssh connection
ansible.netcommon.netconf       Provides a persistent connection using the netconf protocol
ansible.netcommon.network_cli   Use network_cli to run command on network appliances
ansible.netcommon.persistent    Use a persistent unix socket for connection
community.aws.aws_ssm           execute via AWS Systems Manager
community.docker.docker         Run tasks in docker containers
community.docker.docker_api     Run tasks in docker containers
community.docker.nsenter        execute on host running controller container
community.general.chroot        Interact with local chroot
community.general.funcd         Use funcd to connect to target
community.general.iocage        Run tasks in iocage jails
community.general.jail          Run tasks in jails
community.general.lxc           Run tasks in lxc containers via lxc python library
community.general.lxd           Run tasks in lxc containers via lxc CLI
community.general.qubes         Interact with an existing QubesOS AppVM
community.general.saltstack     Allow ansible to piggyback on salt minions
community.general.zone          Run tasks in a zone instance
community.libvirt.libvirt_lxc   Run tasks in lxc containers via libvirt
community.libvirt.libvirt_qemu  Run tasks on libvirt/qemu virtual machines
community.okd.oc                Execute tasks in pods running on OpenShift
community.vmware.vmware_tools   Execute tasks inside a VM via VMware Tools
containers.podman.buildah       Interact with an existing buildah container
containers.podman.podman        Interact with an existing podman container
kubernetes.core.kubectl         Execute tasks in pods running on Kubernetes
```

Om specifieke documentatie over een plugin op te vragen, geef je dit mee als parameter:

```
$ ansible-doc -t connection <plugin name>
```

3.7.6.1 MICROSOFT POWERSHELL REMOTING PROTOCOL

Als voorbeeld nemen we de ansible.builtin.psrp connection plugin die remote Powershell gebruikt om de configuratie te wijzigen.

Om deze plugin te kunnen gebruiken moet je Powershell Remoting activeren op de Windows host:
Enable-PSRemoting -Force

Op de Ansible Control Node heb je de winrm package nodig. Op debian is dat:

```
$ apt install python3-winrm
```

In ansible.cfg enable je psrp als transport:

```
[defaults]
transport = psrp
```

Na het instellen van authenticatie (https://docs.ansible.com/ansible/latest/os_guide/windows_winrm.html#winrm-authentication-options) kan je playbooks maken die gebruik maken van deze connection plugin. Bijvoorbeeld:

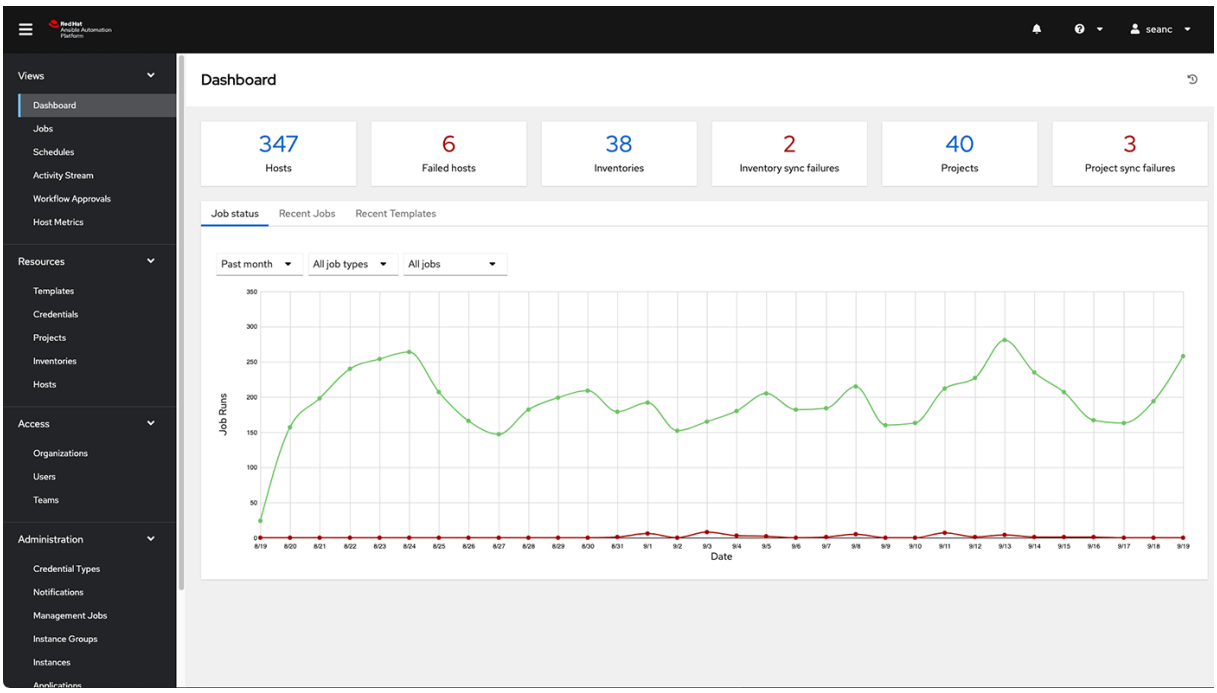
```
- hosts: windows_hosts
  gather_facts: no
  vars:
    ansible_connection: psrp
  tasks:
    - name: Execute a PowerShell command
      ansible.windows.win_shell: Get-Process
```

3.7.7 GRAFISCHE OMGEVINGEN

Om Ansible nog gebruiksvriendelijker te maken, zijn er verschillende opties om een grafische interface te gebruiken. We maken hierin het onderscheid tussen wat RedHat voorziet en een paar open source alternatieven.

3.7.7.1 REDHAT ANSIBLE GUI

De eerste optie is de Ansible Automation Controller.



Figuur 9 Ansible Automation Controller

Deze tool is een vervanger voor Ansible Tower, de vroegere web interface van Ansible. Beide tools werden gebouwd en onderhouden door RedHat zelf maar zijn dus niet gratis te gebruiken. Met Automation Controller kan je:

- Workflows bouwen om complexe processen te automatiseren
- Visibility krijgen over hoe je playbooks lopen en de status van de Managed Nodes of hosts

- Je krijgt een single source of truth omdat alle logdata samengevoegd wordt en in dashboards gegoten wordt.
- De Ansible Control Node opschalen als de belasting stijgt.

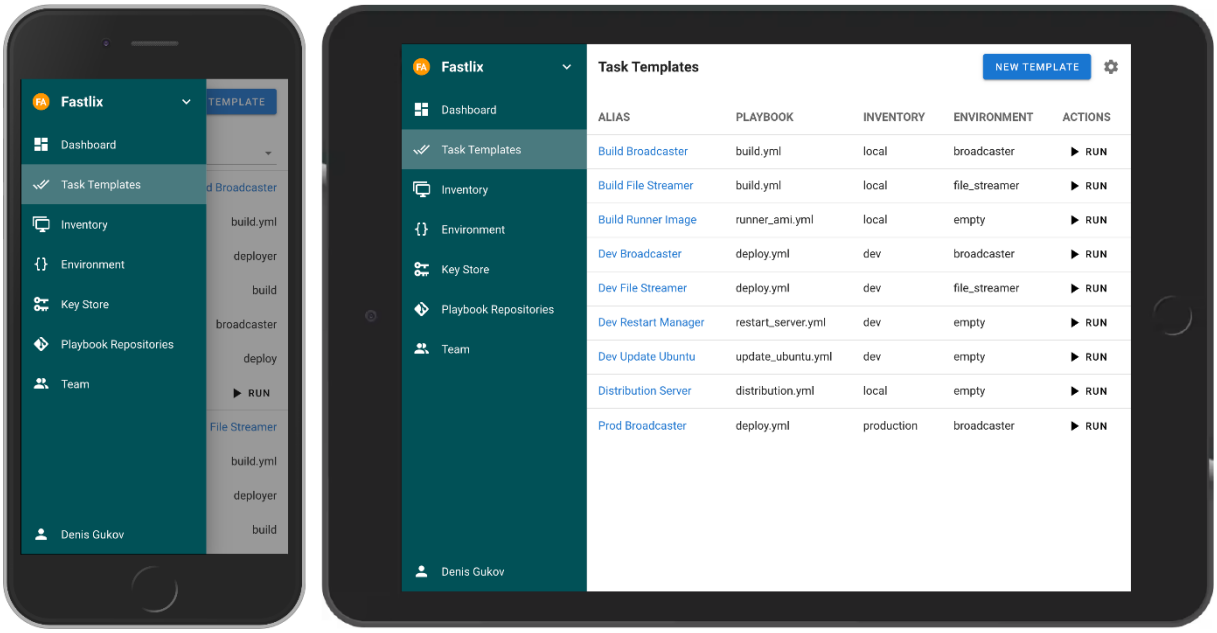
3.7.7.2 OPEN SOURCE GUI

Ansible AWX

AWX is de open source versie van Ansible Tower. Waar de twee vroeger naast elkaar stonden, besliste RedHat om te stoppen met Tower en de open source variant AWX mee te ondersteunen. Het idee hierachter is dat ze mee helpen met de ontwikkeling van het product en de community ondersteunen om zo tot stabiele versies te komen die ze dan zelf kunnen gebruiken als basis voor hun Automation Controller. Je mag dus een heel gelijkaardige featureset verwachten tussen AWX en Automation Controller/Tower.

Semaphore

Deze tool werd vanaf de grond opgebouwd in Go en mikt vooral op een moderne user interface die gebruiksgemak voorop stelt.



Figuur 10 Semaphore interface

Er bestaan twee versies van dit product: de open source variant die je zelf kan opzetten of de cloud based oplossing die werkt met een subscription model.

Het is niet het doel van deze cursus om een vergelijking te maken tussen de verschillende grafische omgevingen of een oordeel te vellen wat er beter is. Veruit de beste manier om te beslissen is om deze tools zelf eens uit te proberen en te zien wat werkt in jouw context.

4 APPENDIX A: TERRAFOM AWS VM

Om de terminologie van Terraform in prakijk te brengen overlopen we hieronder de stappen die nodig zijn om via Terraform een Elastic Compute instance (EC2), een VM op AWS, aan te maken. Dit onderdeel bevat dummy syntax om het principe duidelijk te maken.

Voordat je kan starten met schrijven van HCL, moet je eerst de CLI installeren waarmee je Terraform provider zal communiceren met gekozen platform. Voor de drie grootste cloud providers zijn dat deze:

- AWS CLI: https://aws.amazon.com/cli/
- Azure CLI: https://learn.microsoft.com/en-us/cli/azure/install-azure-cli
- GCP gcloud CLI: https://cloud.google.com/sdk/docs/install

Deze CLI's maken verbinding naar het cloud platform en voorzien de Terraform Provider van de nodige code om de API's aan te spreken.

De volgende vereiste voor we kunnen starten is het installeren van Terraform zelf. Die vind je hier: https://developer.hashicorp.com/terraform/install

Nu alles klaar staat kunnen we starten. Je structureert Terraform files in .tf files.

De eerste stap is het aanmaken van de provider:

```
provider "aws" {
  region = "eu-west-3"  # Replace with your desired region
  # Optional: You can specify a profile name if using AWS CLI profiles
  # profile = "your_aws_cli_profile_name"
}
```

Deze zal gebruik maken van de credentials op je CLI, maar dat kan overschreven worden in de provider config indien nodig.

Als we een nieuwe provider toevoegen, moeten we die initialiseren via het **terraform init** commando:

```
> Terraform init
Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions matching ">= 3.0, < 4.0"...
- Installing hashicorp/aws v3.62.0...
- Installed hashicorp/aws v3.62.0 (signed by HashiCorp)

Terraform has been successfully initialized!
```

Nadien kunnen we onze eerste resource toevoegen, een aws_instance met bijhorend IP én een security group (soort firewall) dat SSH-toegang toelaat:

```
# Create an Elastic IP
resource "aws_eip" "example" {
```

```
  instance = null  # This will be set dynamically when the EC2 instance is
created
}

# Create a security group allowing SSH access
resource "aws_security_group" "ssh_access" {
  name        = "ssh-access"
  description = "Allow inbound SSH traffic"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]  # Allow SSH access from anywhere (for
demonstration purposes; restrict as needed)
  }
}

# Launch an EC2 instance
resource "aws_instance" "example" {
  ami           = "ami-xxxxxxxxxxxxx"  # Replace with your desired AMI
  instance_type = "t2.micro"           # Replace with your desired instance
type
  key_name      = "your-keypair"       # Replace with your SSH key pair

  # Associate the EC2 instance with the created security group
  vpc_security_group_ids = [aws_security_group.ssh_access.id]

  # Associate the Elastic IP with the EC2 instance
  provisioner "local-exec" {
    command = "aws ec2 associate-address --instance-id ${self.id} --
allocation-id ${aws_eip.example.id} --region ${var.aws_region}"
  }

  # Tag the instance for better organization
  tags = {
    Name = "ExampleInstance"
  }
}
```

Om deze objecten aan te maken, voeren we het **terraform apply** commando uit. Terraform zal weergeven welke resources aangemaakt of vernietigd gaan worden en je de vooruitgang van die acties tonen.

```
> terraform apply


Plan: X to add, Y to change, Z to destroy.

Changes to Outputs:
```

```
  + aws_eip_example = (known after apply)

  # Other changes...

Do you want to perform these actions?

  Terraform will perform the actions described above.

  Only 'yes' will be accepted to approve.

Enter a value: yes

aws_security_group.ssh_access: Creating...

aws_security_group.ssh_access:    Creation    complete    after    Xs    [id=sg-
xxxxxxxxxxxxxxxx]

aws_instance.example: Creating...

aws_instance.example: Still creating... [10s elapsed]

aws_instance.example: Still creating... [20s elapsed]

aws_instance.example: Creation complete after 30s [id=i-xxxxxxxxxxxxxxxx]

aws_eip.example: Creating...

aws_eip.example: Creation complete after Ys [id=eipalloc-xxxxxxxxxxxxxxxx]

Apply complete! Resources: X added, Y changed, Z destroyed.
```

5 BRONNEN

[1] RedHat, „What is infrastructure automation?,” 22 03 2023. [Online]. Available: https://www.redhat.com/en/topics/automation/what-is-infrastructure-automation. [Geopend 15 11 2023].

[2] XKCD, „Automation,” [Online]. Available: https://xkcd.com/1319/. [Geopend 15 11 2023].

[3] J. P. N. R. M. C. J. Betsy Beyer, „Chapter 7 - The Evolution of Automation at Google,” in *Site Reliability Engineering - how Google runs production systems*, O'Reilly, 2016, p. 524.

[4] RedHat, „What is Infrastructure as Code (IaC)?,” RedHat, 11 05 2022. [Online]. Available: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac. [Geopend 15 11 2023].

[5] HashiCorp, „The Story of HashiCorp Terraform with Mitchell Hashimoto,” HashiCorp, 13 07 2021. [Online]. Available: https://www.hashicorp.com/resources/the-story-of-hashicorp-terraform-with-mitchell-hashimoto. [Geopend 15 11 2023].

[6] Terraform, „What is Terraform (intro),” [Online]. Available: https://developer.hashicorp.com/terraform/intro. [Geopend 15 11 2023].

[7] A. Guérin, „What the Terraform Provider registry says on the cloud market share,” 03 11 2023. [Online]. Available: https://medium.com/@alexandre_43174/what-the-terraform-provider-registry-says-on-the-cloud-market-share-7b3f9c689da6. [Geopend 15 11 2023].

[8] RedHat, „Getting started with Ansible,” RedHat, 22 11 2023. [Online]. Available: https://docs.ansible.com/ansible/latest/getting_started/index.html. [Geopend 04 12 203].

[9] RedHat, „Introduction to ad hoc commands,” RedHat, 22 11 2023. [Online]. Available: https://docs.ansible.com/ansible/latest/command_guide/intro_adhoc.html. [Geopend 5 12 2023].

[10] RedHat, „Ansible Playbooks,” RedHat, 07 12 2023. [Online]. Available: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html. [Geopend 11 12 2023].

[11] javaTpoint, „Ansible Playbooks,” [Online]. Available: https://www.javatpoint.com/ansible-playbooks. [Geopend 11 12 2023].

[12] RedHat, „Using Variables,” RedHat, 12 12 2023. [Online]. Available: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_variables.html. [Geopend 18 12 2023].

[13] RedHat, „Gotchas,” RedHat, 12 12 2023. [Online]. Available: https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html#yaml-syntax. [Geopend 19 12 2023].

[14] RedHat, „Conditionals," RedHat, 12 12 2023. [Online]. Available: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_conditionals.html. [Geopend 20 12 2023].

[15] RedHat, „Loops," RedHat, 12 12 2023. [Online]. Available: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_loops.html. [Geopend 20 12 2023].

[16] RedHat, „Migrating to loop," RedHat, 12 12 2023. [Online]. Available: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_loops.html#migrating-to-loop. [Geopend 20 12 2023].

[17] RedHat, „Forcing lookups to return lists: query and wantlist=True," RedHat, 12 12 2023. [Online]. Available: https://docs.ansible.com/ansible/latest/plugins/lookup.html#query. [Geopend 20 12 2023].

[18] RedHat, „ansible.builtin.dict2items filter – Convert a dictionary into an itemized list of dictionaries," RedHat, 12 12 2023. [Online]. Available: https://docs.ansible.com/ansible/latest/collections/ansible/builtin/dict2items_filter.html. [Geopend 20 12 2023].

[19] RedHat, „Roles," RedHat, 12 12 2023. [Online]. Available: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_reuse_roles.html. [Geopend 15 12 2023].

[20] RedHat, „Understanding privilege escalation: become," RedHat, 12 12 2023. [Online]. Available: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_privilege_escalation.html. [Geopend 12 20 2023].

[21] Atlassian, „GitFlow Workflow," Atlassian, [Online]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow. [Geopend 21 12 2023].

[22] RedHat, „Ansible Configuration Settings," RedHat, 12 12 2023. [Online]. Available: https://docs.ansible.com/ansible/latest/reference_appendices/config.html. [Geopend 20 12 2023].

howest.be