Hes·so // VALAIS WALLIS
Haute Ecole de Gestion
Hochschule für Wirtschaft

Filière Informatique de gestion
Studiengang Wirtschaftsinformatik

University of Applied Sciences and Arts Western Switzerland
Business Information Technology

IT Infrastructure Automation

August 2021

Written by : Bruchez Jonas
Professor in charge : Barmaz Xavier
Submitted August 3, 2021

Abstract

The purpose of this document is to define the state of the art regarding information technology infrastructure automation. Nowadays, companies need a way to automate their infrastructure to reduce their expenses and avoid wasting time. System administrators could then work with more efficiency and focus on solving challenges in the company that actually require human intervention. An overview and analysis of the main existing open-source automation software available on the market has been performed with the objective of defining what kind of software a company should choose depending on their needs. The analysis showed that Ansible and Puppet were the most ideal software according to the criteria established. An in-depth lab has been established using the selected software. The installation guide explains the different steps required to set up the environment and the configuration guide offers a detailed example configuration using each software. The lab revealed a large difference between the use of Ansible and Puppet. The quality and amount of information found about Ansible online was greatly superior and the ease of use was even more obvious than expected.

Keywords : Automation, Infrastructure as Code, Configuration management, Server/Agents

Foreword

This thesis was written as the final work of my Bachelor studies in Business Information Technology. The information technology infrastructure may be of significant size in a company and operators need tools to help them save time on repetitive tasks that do not necessarily require insight from them. The subject of this report is the analysis and testing of different software providing these tools and determine the best solution regarding infrastructure automation currently.

The difficulty encountered was to select the right information among contradictory comparisons among the references used to write this report. It is also important to note that software and hardware both evolve very quickly, therefore we cannot ensure that the conclusion and results reached will still apply in few years.

This report has been written with the assumption that the reader has a basic understanding of information technology infrastructure and Linux architecture.

The research, analysis and testing took place from May 2021 to August 2021.

USB Flash Drive Structure

You will find, attached to this report, a USB flash drive containing the following files :

- 01_Report
- 02_Installation_Guide
- 03_Configuration_Guide
- 04_Poster

Special Thanks

I would like to thank Mr. Xavier Barmaz, professor at the University of Applied Sciences of Western Switzerland, for following my work and giving me great advice.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
DevOps	Development / Operations
DSL	Domain Specific Language
DNS	Domain Name Service
FQDN	Fully Qualified Domain Name
GUID	Globally Unique Identifier
HCL	HashiCorp Configuration Language
HTML	HyperText Markup Language
IaC	Infrastructure as Code
IT	Information Technology
JSON	JavaScript Object Notation
OOS	Open-Source Software
OS	Operating System
PC	Personal Computer
PEM	Privacy Enhanced Mail
RHEL	Red Hat Enterprise Linux
RSA	Rivest-Shamir-Adleman (protocol)
SQL	Structured Query Language
SSH	Secure Shell Protocol
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
VCS	Version Control System
VM	Virtual Machine
WinRM	Windows Remote Management
WSL	Windows Subsystem for Linux
YAML	YAML Ain’t Markup Language (previously Yet Another Markup Language)

Introduction

Nowadays, companies are starting to work with an increasing number of software, infrastructures and devices, allowing always more possibilities to organize their Information Technology (IT) environment. However, the more complex and numerous the possibilities, the harder it is to set it all up or to simply keep track of all processes running through a company. Automation grants considerably more control over the whole IT infrastructure and system administrators gain a lot of time using the right tools to execute tasks that would otherwise be long and repetitive.

Therefore, plenty of automation software were created to help with this issue. They allow system administrators to create scripts executing repeatable instructions and processes to drastically reduce the human interaction with IT systems. For instance, a server with this kind of software installed would be able to manage other servers on the same network by installing, updating or deploying applications, modules, etc.

The goal of this Bachelor's thesis is to identify software that can be used for automation, analyse and test their features and capabilities and, in the end, pick the most interesting ones to compare them in a more complex testing scenario. The final objective being the discovery of the "best" open-source software available to this day according to a certain set of criteria and conditions.

To achieve this goal, we use the abundant documentation found on the Internet as well as the books available on the intranet of the University of Applied Sciences. All sources are available in the references at the end of this report. Furthermore, in order to test all software, we use Oracle VM VirtualBox, an open-source tool allowing users to create and manage virtual machines. Manipulating this tool proves to be very convenient to quickly set up a network environment where tests can be easily performed.

1. State of the Art

1.1. IT Infrastructure

The IT infrastructure can quickly become too complex to manage easily. Let us first define what we designate to be part of the IT infrastructure in a company :

- **Hardware**

Hardware includes servers, personal computers (PCs) as well as networking equipment such as routers, switches, etc.

- **Software**

Software refers to applications and operating systems (OS) running on any piece of hardware. We also include virtual machines (VMs) as they emulate operating systems using hardware resources.

- **Networking**

Networking makes hardware and software communicate inside and outside of the company. A company network requires configuration (using routers and switches as mentioned above), internet connectivity, firewalls and other security measures. In summary, a network links the whole infrastructure together and helps it communicate with the outside world.

1.2. Infrastructure Automation

In the past, operators had to manage their IT infrastructure manually, which was a difficult process as they needed to configure all servers within a company themselves. This fact alone tells us that managing such an infrastructure was very expensive, slow and even inconsistent as no one is safe from making a mistake nearly impossible to spot easily in such a long and repetitive process. Manual maintenance of these same servers was also further increasing the total costs.

Automation is the act of orchestrating processes and writing configuration scripts that will operate and act on managed nodes without human intervention. A node is defined as a machine whose configuration is automated by a server hosting a configuration management software.

Automation is not about replacing human operators, but instead to free them from long and repetitive tasks so they can focus on more complex issues of their everyday work that require actual insight from them, rather than logical rules that can be scripted.

As for every repetitive task, automation also reduces the number of potential human errors and provides more consistent reliability on a company IT infrastructure.

1.2.1. DevOps

DevOps (Development & Operations) is a set of practices combining Development and IT operations within a company.

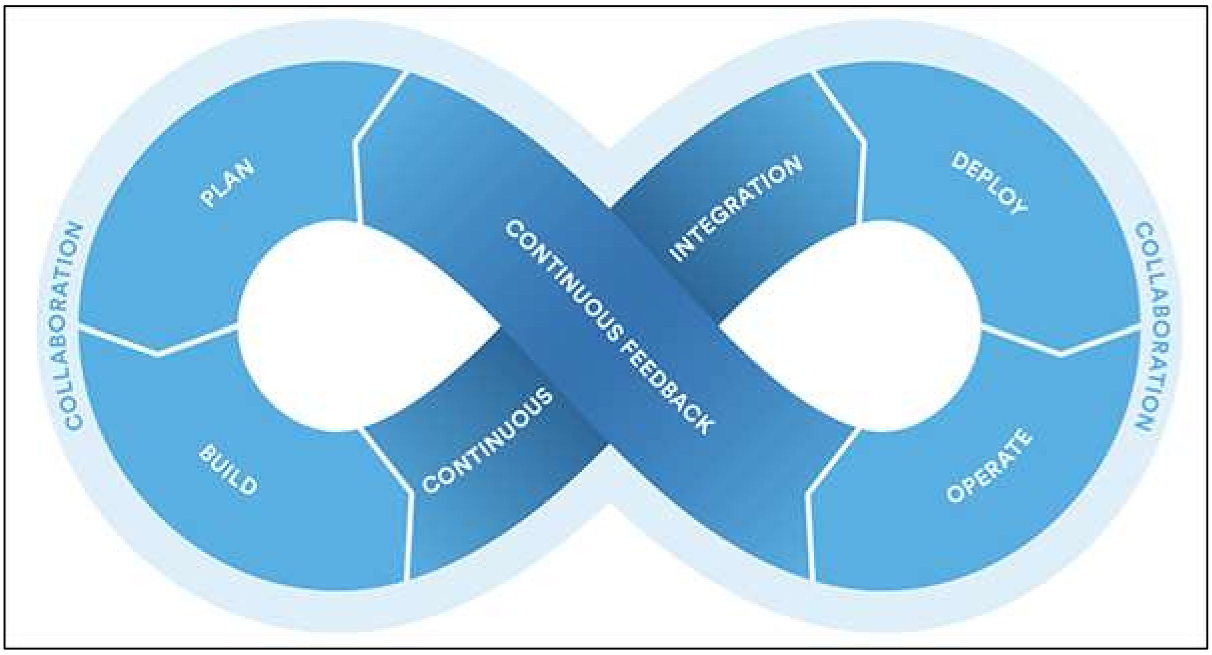


Figure 1 - DevOps architecture (Krebsbach, 2016)

DevOps works as follows : On the left side of the diagram above is the development part (Plan & Build), and on the right side the IT operations management part (Deploy & Operate). These two sectors work together in a DevOps scenario by communicating constantly with each other.

The dev team provides continuous integration, which is the practice of automating the integration of changes made to the code from multiple developers into a single software project. On the other hand, the IT operators give them continuous feedback from customers on the software deployed.

Using DevOps is a good practice and has proven very efficient for IT companies, although its success depends on the team's ability to adapt and respect the procedures.

Automation software are usually built to work well with DevOps.

1.2.2. Infrastructure as Code

Infrastructure as Code (IaC) is the practice of codifying and managing the IT infrastructure with software, rather than through physical hardware components. It simply means to automate all tasks related to the IT infrastructure using tools and programs.

Most automation software provide IaC, but not necessarily on the same level. We can differentiate three main categories of tasks that can be automated when a new application must be deployed in a company :

1. **Infrastructure provisioning** : Setup and maintain new servers, apply network configuration.

2. **Configuration management** : Install and maintain software and packages required to deploy the new application.

3. **Application deployment** : Deploy and maintain the new application on servers to start the DevOps workflow.

Each category can then be separated in two phases, the **initial setup phase**, which simply is the initial configuration of each category mentioned above, and the **maintaining phase**, which as its name implies requires maintenance of the infrastructure and installed software to manage changes and updates. To each software can be attributed a main category, although it is common for an application to offer features coming from more than one category.

1.2.3. Idempotence

IaC is an important practice for the DevOps process, as it provides idempotence. From the official Microsoft documentation website, “idempotence is the property that a deployment command always sets the target environment into the same configuration, regardless of the environment's starting state. Idempotency is achieved by either automatically configuring an existing target or by discarding the existing target and recreating a fresh environment.” (Jacobs & Kaim, 2021). That definition implies that each same execution in an environment provides the same result. For instance, if we attempt to install an application that does not exist in an environment but exists in another instance of that environment, the end result will be the same : The application is installed. The software installation will not be executed twice. This helps companies tracking states and bugs resulting therefrom more easily.

1.2.4. Procedural vs Declarative

In IaC, there are two possibilities for the configuration in all three categories : **procedural language** (also called **imperative language**) and **declarative language**. Procedural language refers to classic logical code, specifying each instruction a program must go through to reach its goal. On the other hand, declarative language is the fact of telling the program what needs to be done by specifying the final state only and let it find a way to generate the necessary steps to finish the configuration according to specific rules and restrictions (the outcome is specified while the process is not).

The software we talk about are either procedural or declarative. Declarative language is often preferred as it gives abstraction. Furthermore, if a step using procedural language fails, we would have to handle the whole configuration and consider what did succeed, which can quickly become complicated. A declarative language would not have this problem ; no matter how many times a declarative script is run, the environment will stay the same, as only the outcome is specified.

1.2.5. Mutable vs Immutable

There are **mutable** and **immutable** IaC tools. Mutable refers to an infrastructure that needs to be updated when needed, while immutable refers to the need of replacing the old IaC at each new deployment to guarantee an exact set of specifications that can be expected. The latter is better for testing and usually the preferred method with DevOps.

Mutable IaC tools introduce more risk, as updates may not fully succeed. It also adds more complexity because of that potential risk because an important update requires to understand the state of a machine before and after the change, and any partial update failure would create a new unknown state to manage.

Immutable IaC tools almost never update the infrastructure, but instead create a whole new environment and apply the desired configuration. The traffic can then be redirected to the new instance and, when no issues remain, the old instance can be disposed of. However, as the old machine is destroyed, the data on it should be externalized (in a database for instance) to avoid any losses.

1.2.6. Agent vs Agentless

Finally, we also talk about **agent** and **agentless** tools. An agentless tool does not require any agents to be installed on the nodes it manages. This is practical if nodes do not support the installation of software on them, such as older networking equipment. However, an agentless system has the disadvantage of having no way for the managed nodes to report back after the execution of a configuration on them.

Agentless tools are push-based, which means that the server pushes the configuration onto the nodes. On the other hand, agent tools are usually pull-based because the agents installed on the nodes automatically pull configuration updates regularly.

1.2.7. Cloud Automation

Cloud automation refers to the help of technology to reduce human assistance for processes present in the Cloud. We can define the Cloud as the Internet and all the data, software and services that can be accessed remotely via servers.

When IT teams have to manage both on-site and cloud-based environments, it quickly becomes complicated and expensive. The main difference between standard IT infrastructure automation and Cloud automation is that the latter is more specific to the automation of online components, such as websites, web services and databases running in the Cloud.

1.2.8. Network Automation

Network automation refers to the help of technology to reduce human assistance to setup and manage the networking infrastructure of a company.

Automating a network infrastructure can be very convenient to avoid complicated steps that require operators to travel around company buildings to manually configure routers, switches and other networking equipment. Manual configuration often leads to errors and inconsistencies, and due to the distant placement and organization of networking components, it is difficult to manage and time-consuming to troubleshoot.

1.2.9. Best Practices for IT Automation

Nowadays, companies that wish to use an automation software must consider the different notions we have described to determine what architecture suits them best depending on their current resources and needs.

Furthermore, when managing and maintaining an infrastructure in a company, standard practices are always generally the same. It applies to automation as well :

- Document all processes and comment configuration files.
- Keep a clear and understandable directory structure and naming conventions.
- Setup a Version Control System (VCS) to avoid data loss.
- Use available tools and prevention to hide sensitive data and improve security.

2. Software Analysis

The main automation software used nowadays have been analysed and tested to discover the most useful in a given situation.

Note that not all functionalities are presented, it is only meant to be an overview of each software architecture and features with pros, cons and analysis in regard to the notions we have developed in the last section.

A few generic definitions have been written for your convenience, as they are important terms and concepts that are mentioned quite a few times in this report.

2.1. Concepts

2.1.1. Open-source software

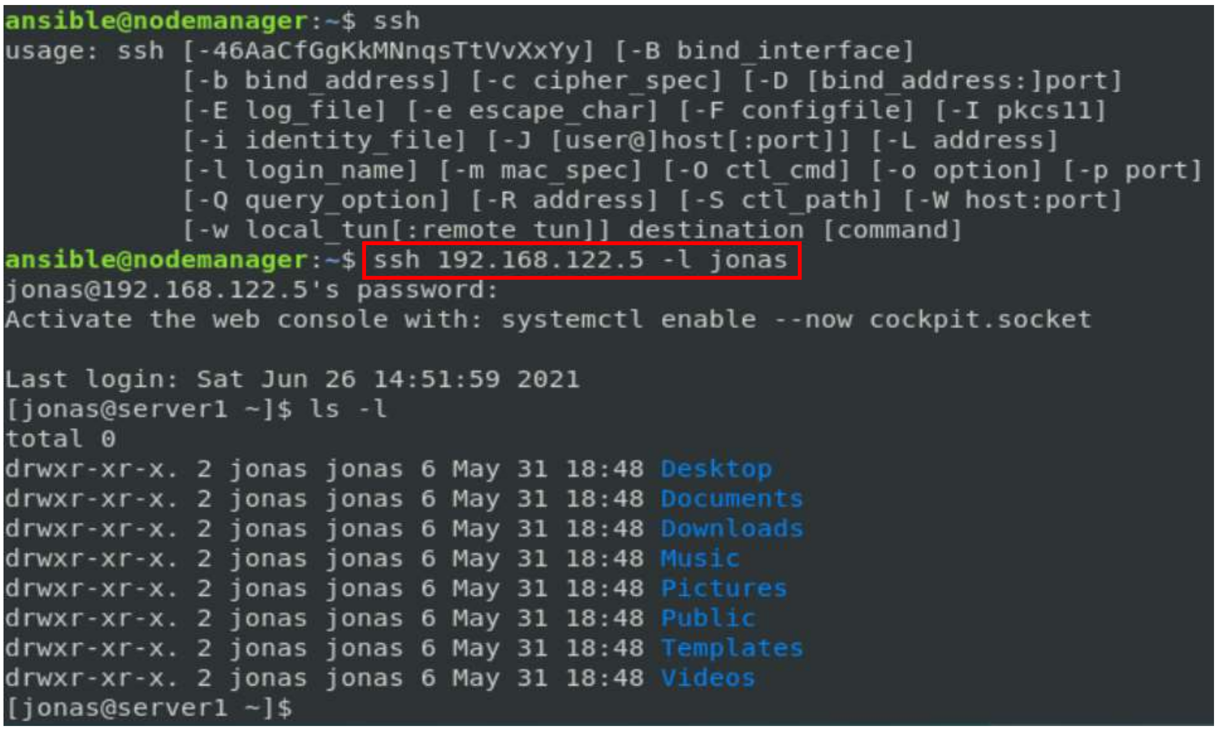
Open-source software (OOS) defines software whose copyright holders (developer company) grants users the rights to use and modify the software as well as its source code without limitations.

For instance, the open-source operating system Linux was first released in 1991, although many developers decided to create their own version of the OS as the source-code was available to anyone. Many Linux distributions exist today (Ubuntu, Debian, openSUSE, Kali, etc.).

Open-source is usually free, although it is possible that the company behind it asks for payment in exchange for technical support.

2.1.2. SSH

Secure Shell Protocol (SSH) is a security network protocol working over the Transmission Control Protocol (TCP) allowing secure communication over any unsecure network. Once the client connects to a server using SSH, that server can be controlled like a local computer.



```
ansible@nodemanager:~$ ssh
usage: ssh [-46AaCfGgKkMNnqsTtVvXxYy] [-B bind_interface]
           [-b bind_address] [-c cipher_spec] [-D [bind_address:]port]
           [-E log_file] [-e escape_char] [-F configfile] [-I pkcs11]
           [-i identity_file] [-J [user@]host[:port]] [-L address]
           [-l login_name] [-m mac_spec] [-O ctl_cmd] [-o option] [-p port]
           [-Q query_option] [-R address] [-S ctl_path] [-W host:port]
           [-w local_tun[:remote_tun]] destination [command]
ansible@nodemanager:~$ ssh 192.168.122.5 -l jonas
jonas@192.168.122.5's password:
Activate the web console with: systemctl enable --now cockpit.socket

Last login: Sat Jun 26 14:51:59 2021
[jonas@server1 ~]$ ls -l
total 0
drwxr-xr-x. 2 jonas jonas 6 May 31 18:48 Desktop
drwxr-xr-x. 2 jonas jonas 6 May 31 18:48 Documents
drwxr-xr-x. 2 jonas jonas 6 May 31 18:48 Downloads
drwxr-xr-x. 2 jonas jonas 6 May 31 18:48 Music
drwxr-xr-x. 2 jonas jonas 6 May 31 18:48 Pictures
drwxr-xr-x. 2 jonas jonas 6 May 31 18:48 Public
drwxr-xr-x. 2 jonas jonas 6 May 31 18:48 Templates
drwxr-xr-x. 2 jonas jonas 6 May 31 18:48 Videos
[jonas@server1 ~]$
```

Figure 2 - SSH connexion

In the example above, a connexion is performed on a server in the same network with the IP address "192.168.122.5". The "-l" argument allows us to choose which user to use on the node we connect to.

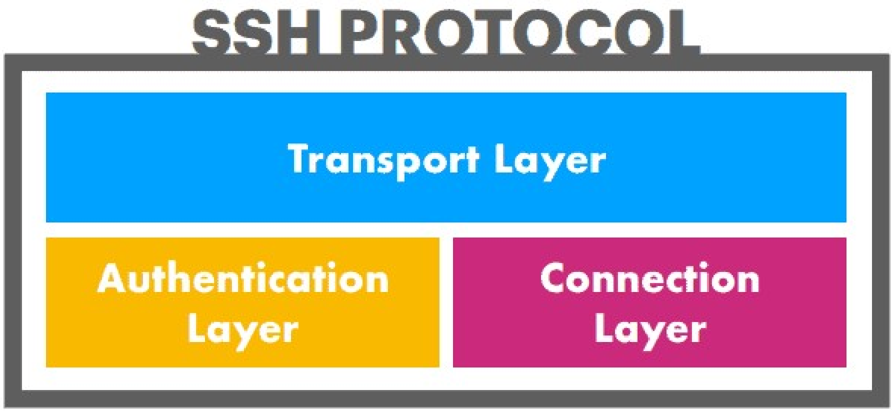


Figure 3 - SSH layers (Aleksic, 2020)

SSH is essentially composed of three layers. The **transport layer** ensures a secure communication between the client and the host for the whole duration of the communication using encryption and decryption. The **authentication layer** authenticates the client to the server, either using a password or a set of SSH keys. The **connection layer** manages the communication between the machines after the authentication and closes the connexion at the end of an SSH session.

On Linux distributions, when a connexion is made for the first time on a new host, the latter is saved in a hosts file (Location : ~/.ssh/known_hosts).

2.1.3. RSA

The Rivest-Shamir-Adleman (RSA) protocol is an asymmetric algorithm used in IT security. RSA encrypts messages with a public key. Messages may only be decrypted with a private key, which is kept secret.

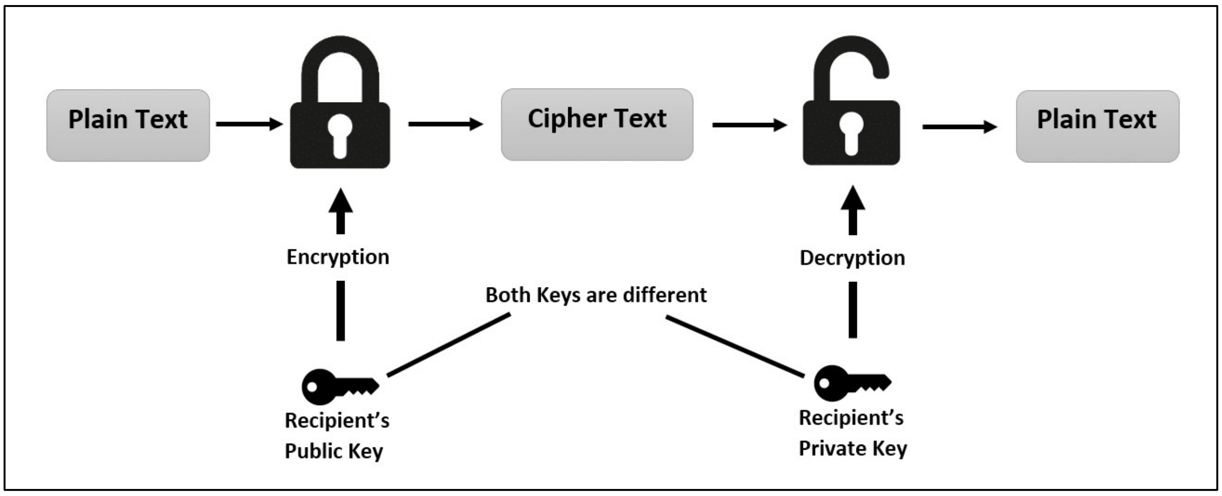


Figure 4 - Simplified RSA diagram (javainterviewpoint, 2019)

The public key is established with a mathematical function using the product of two very large prime numbers. On the other hand, the private key is built with the actual prime numbers. It is nearly impossible to find the private key with the public key unless spending a near infinite amount of time or waiting for quantum computers capable of breaking RSA.

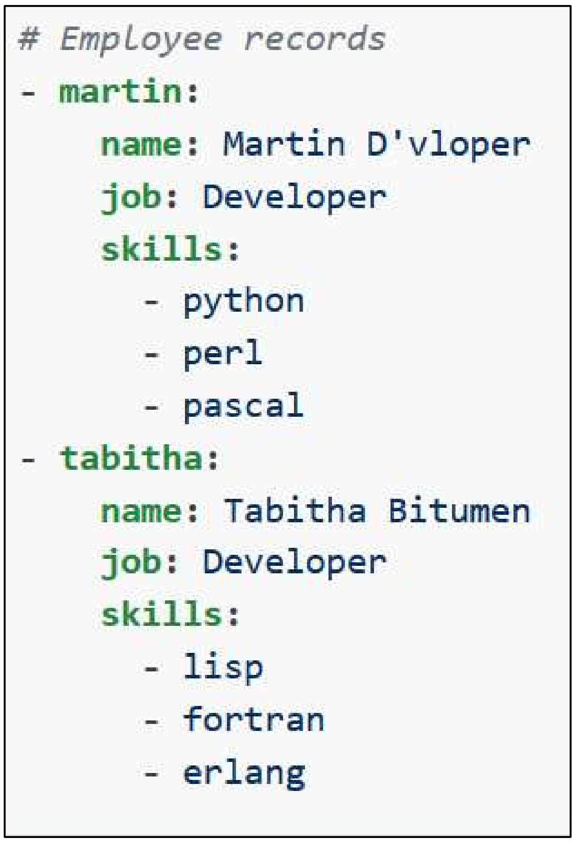
2.1.4. WinRM

Windows Remote Management (WinRM) is an implementation of the WS-Management Protocol, developed by Microsoft and used to exchange information between Windows servers or to control a Windows server from a Linux client.

2.1.5. YAML

“YAML Ain’t Markup Language” (YAML) is a data-serialization language whose precise syntax makes it easily readable by both humans and computers. Such a file uses indentation with white spaces to indicate if elements are nested or not. Indentation, colons and dashes are the three main syntax components of YAML files.

The YAML language is mostly used to write configuration files.



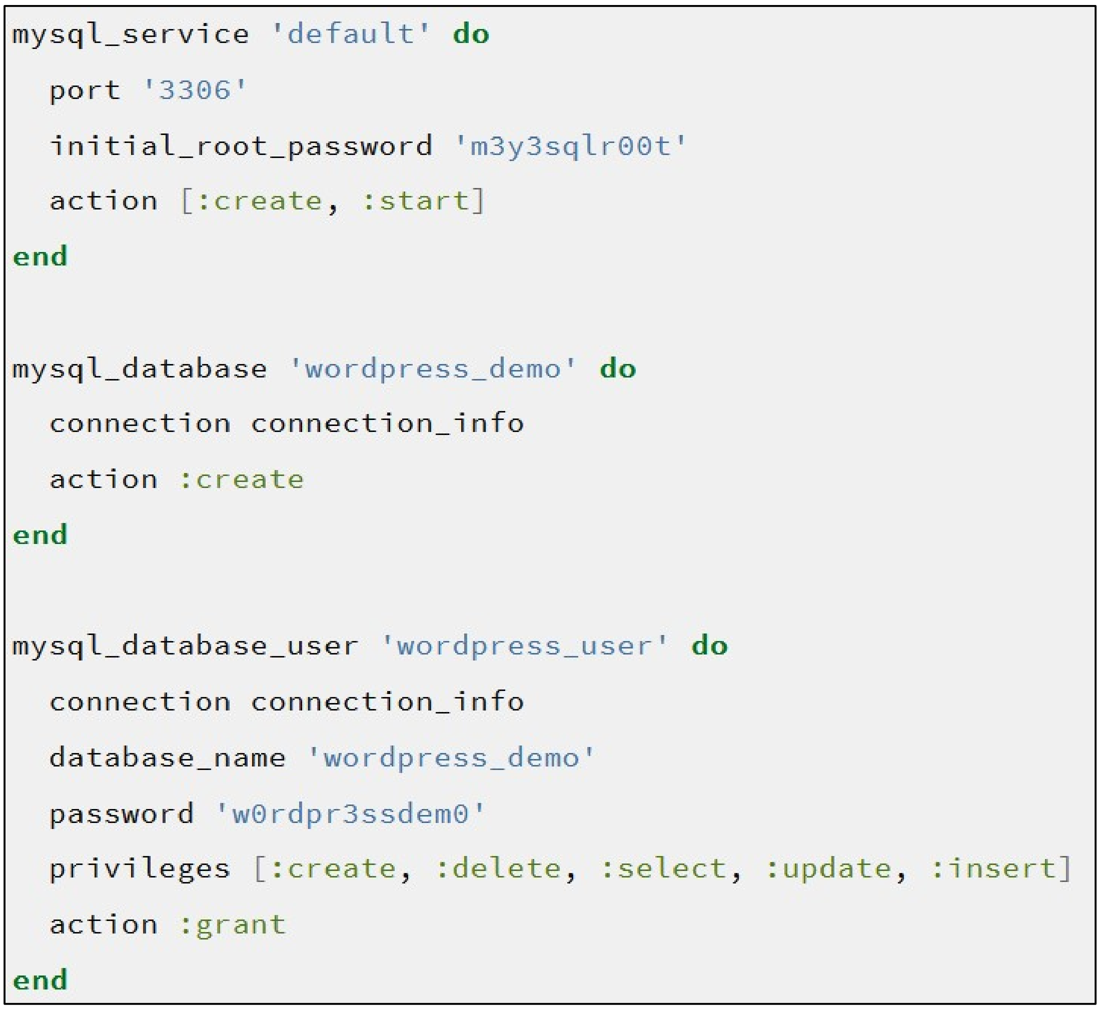
```
# Employee records
- martin:
    name: Martin D'vloper
    job: Developer
    skills:
      - python
      - perl
      - pascal
- tabitha:
    name: Tabitha Bitumen
    job: Developer
    skills:
      - lisp
      - fortran
      - erlang
```

Figure 5 - YAML structure (Ansible Documentation, n.d.)

Above is an example of a simple YAML file taken from the official Ansible documentation. We clearly see a resemblance with the JavaScript Object Notation (JSON) language.

2.1.6. Ruby

Ruby is an open-source, high-level and object-oriented programming language. It is more complex and allows more detailed operations than YAML as the latter does not compare to an actual programming language.



```ruby
mysql_service 'default' do
  port '3306'
  initial_root_password 'm3y3sqlr00t'
  action [:create, :start]
end

mysql_database 'wordpress_demo' do
  connection connection_info
  action :create
end

mysql_database_user 'wordpress_user' do
  connection connection_info
  database_name 'wordpress_demo'
  password 'w0rdpr3ssdem0'
  privileges [:create, :delete, :select, :update, :insert]
  action :grant
end
```

Figure 6 - Ruby structure (Chef Documentation, n.d.)

Above is a sample of code from the official Chef documentation depicting actions performed on a Structured Query Language (SQL) service.

2.2. Ansible

2.2.1. Architecture

Ansible is an open-source configuration management software. It was created in 2012 and is owned by the software company Red Hat since 2015. The software is written in the Python programming language. Furthermore, Ansible is agentless, which means no agents need to be installed on the physical or virtual devices managed. Instead, a new connexion will be opened using SSH from a server using Ansible to any of the linked devices.

Ansible's architecture works as follows : A main server with Ansible installed on it is able to communicate with other Linux machines using **SSH** and with Windows machines using **WinRM**. Note that since Ansible 2.8, an experimental feature allows the use of SSH to manage Windows servers as well, although that feature is experimental and may not work properly.

The main server ("control node" or "node manager") may then execute commands or scripts ("playbooks") on all the devices ("nodes") specified in a list of hosts ("inventory").

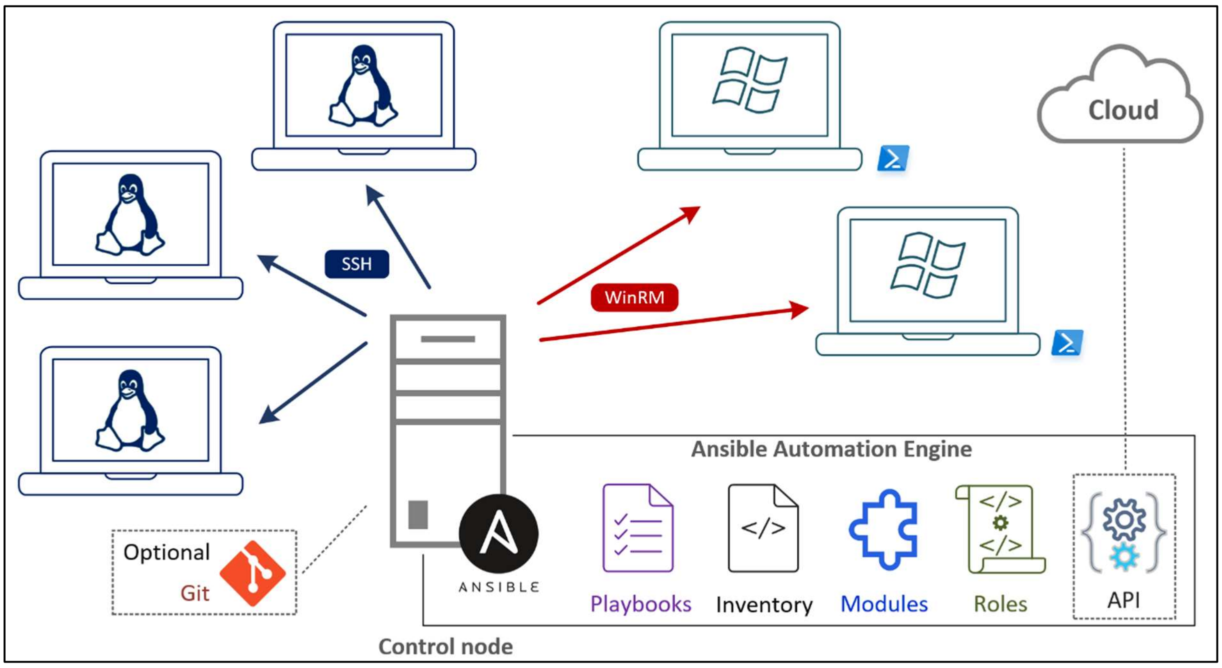


Figure 7 - Ansible architecture (Ansible Verwendungsszenarien, n.d.)

The automation is performed using scripts and tools such as playbooks, inventory files, modules and roles.

Ansible is Python-based and uses Jinja2, a templating language necessary to write more complex scripts.

The professional Ansible version, named "Ansible Tower" is free to use to manage up to 10 nodes. This version provides support from Ansible as well as a graphical user interface.

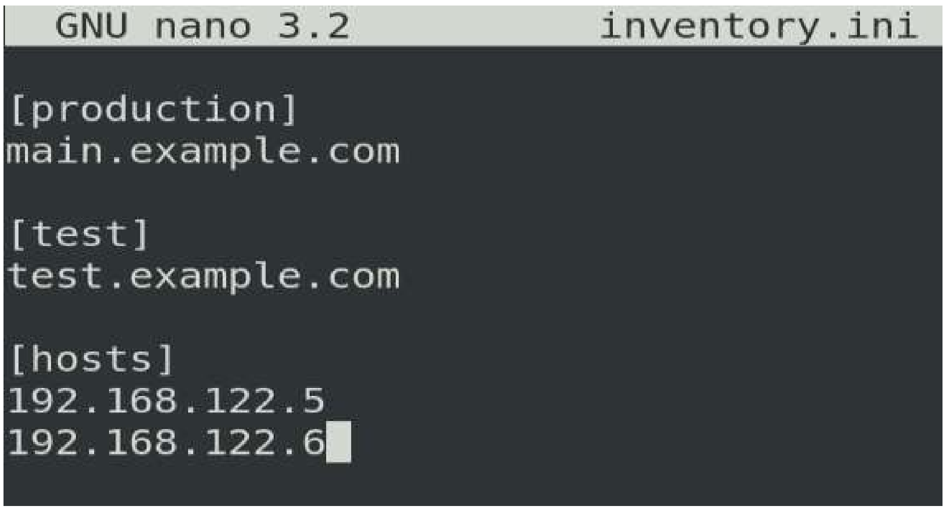
2.2.2. Installation

Installing Ansible is not too complicated, although it may change slightly depending on the OS version of the node manager. The installation can be made with a few commands ensuring the presence of a few required packages as well.

Ansible is agentless as mentioned before, so there are no agents to install on the nodes. However, the fact that WinRM is necessary to communicate with Windows machines makes the configuration more complicated than with Linux nodes.

2.2.3. Inventory

The inventory file defines the hosts affected by the execution of Ansible playbooks. Hosts may be put together into groups using square brackets (“[]”).



```
  GNU nano 3.2              inventory.ini

[production]
main.example.com

[test]
test.example.com

[hosts]
192.168.122.5
192.168.122.6
```

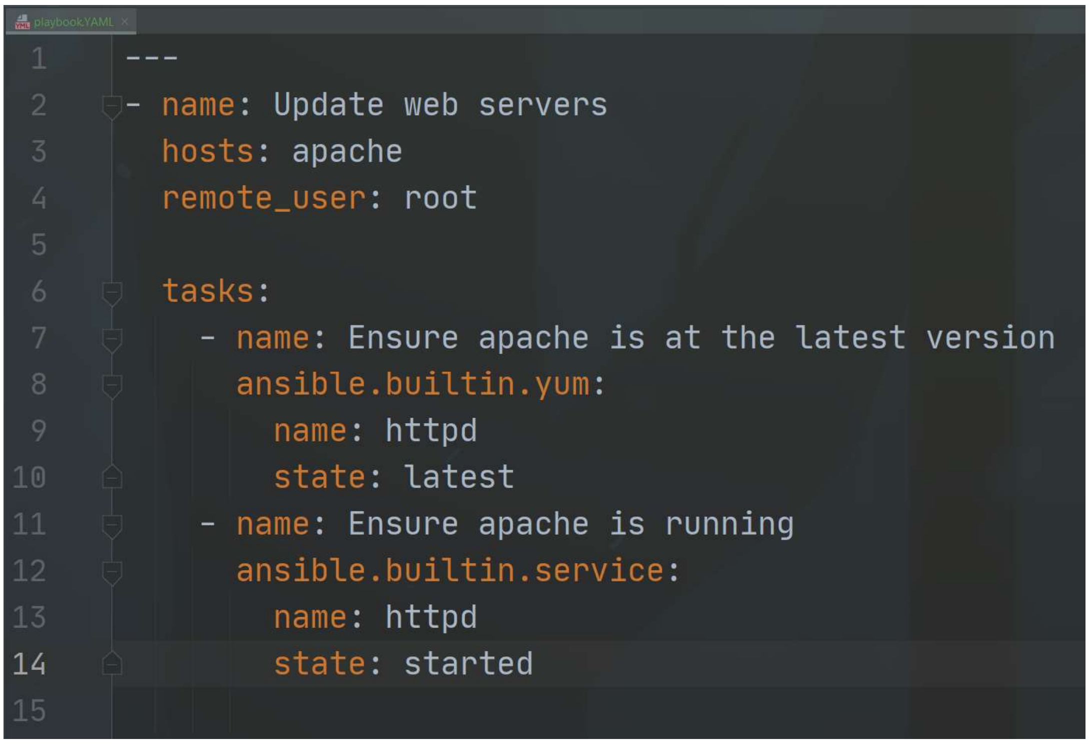
Figure 8 - Ansible inventory file example

For instance, in the inventory file above, three groups have been set up : “production”, “test” and “hosts”. It would now be possible to execute commands on the server “main.example.com” by specifying that host in a playbook. It would also be possible to install a program on all machines in the “hosts” group.

2.2.4. Playbooks

A playbook is a YAML script used to organize tasks that need to be run on any number of managed servers. It orchestrates how, at what time and which modules should be executed.

Ansible reads the YAML syntax to understand what it needs to execute and on which servers.



```
1   ---
2   - name: Update web servers
3     hosts: apache
4     remote_user: root
5
6     tasks:
7       - name: Ensure apache is at the latest version
8         ansible.builtin.yum:
9           name: httpd
10          state: latest
11      - name: Ensure apache is running
12        ansible.builtin.service:
13          name: httpd
14          state: started
15
```

Figure 9 - Playbook example

Above is an example of a simple playbook file. On row 2, a **name** is given to a “play”.

On row 3, we choose which **hosts** will be affected by the current “play”. We can indicate a group of hosts, a hostname or an IP address directly.

On row 4, the keyword “remote_user” defines which user the server we are connecting to will be using to execute the different tasks.

After specifying the identity and connexion details, we can now look at the actual **tasks** to be performed in this case, starting on row 6.

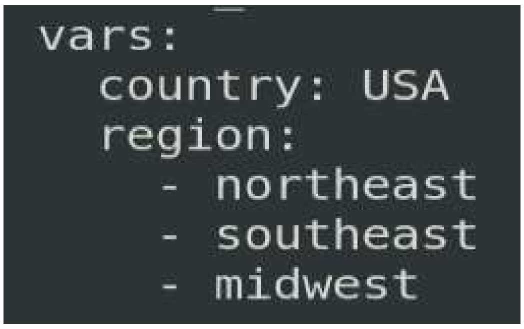
On row 7, the keyword “name” is present again, although this time it defines a task that we want to execute on the node. In this case, we want to ensure that the web server Apache is at its latest version on the server.

On row 8, we specify which module the server should use as we need a certain tool to execute that specific task. In this case, the package manager “yum” is used. It is one of the tools to get, install and delete packages and is mainly used on operating systems managed by the Red Hat company.

On row 9 and 10, we state the name of the service we want to install “HTTPd” which stands for “Hypertext Transfer Protocol daemon”. That program is also known as “Apache” or “Apache Web Server”. Finally, we indicate the Apache version we want to install (if not present) or update to (if already present) ; “latest” means the latest stable version of the software.

On rows 11 to 14, another task is present to ensure that Apache is in the state “running” using the Linux command “service”.

Note that it is possible to declare variables at the top of a playbook :



```
vars:
   country: USA
   region:
      - northeast
      - southeast
      - midwest
```

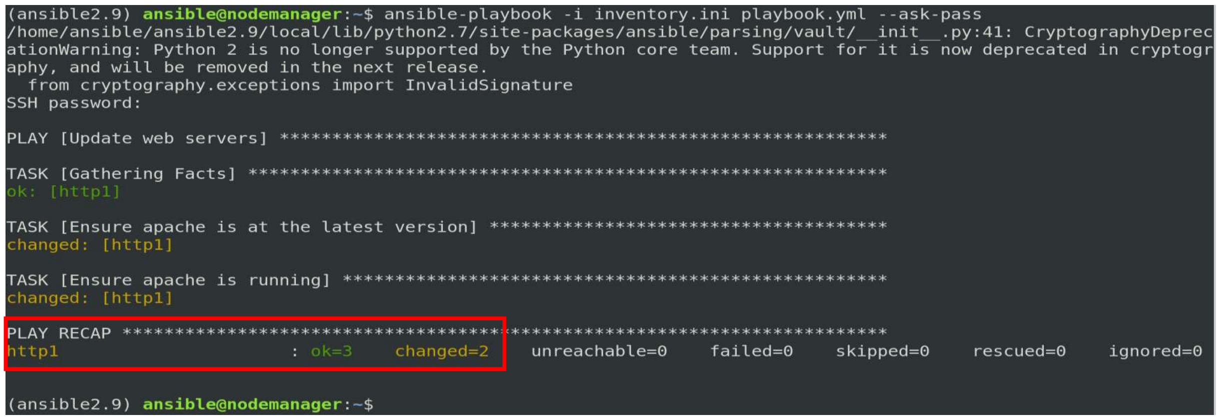
Figure 10 - Ansible variables example

The variables can then be used. For instance, using the example above, “country” would return “USA” and “region[2]” would return “midwest”.

To run a playbook, we use the command “ansible-playbook” :

ansible-playbook -i inventory.ini playbook.yml --ask-pass

Where “-i” specifies the inventory file and “--ask-pass” corresponds to the password of the user specified in the playbook (**root** in our case).



```
(ansible2.9) ansible@nodemanager:~$ ansible-playbook -i inventory.ini playbook.yml --ask-pass
/home/ansible/ansible2.9/local/lib/python2.7/site-packages/ansible/parsing/vault/__init__.py:41: CryptographyDeprec
ationWarning: Python 2 is no longer supported by the Python core team. Support for it is now deprecated in cryptogr
aphy, and will be removed in the next release.
  from cryptography.exceptions import InvalidSignature
SSH password:

PLAY [Update web servers] **********************************************************

TASK [Gathering Facts] *************************************************************
ok: [http1]

TASK [Ensure apache is at the latest version] **************************************
changed: [http1]

TASK [Ensure apache is running] ****************************************************
changed: [http1]

PLAY RECAP *************************************************************************
http1                      : ok=3    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0


(ansible2.9) ansible@nodemanager:~$
```

Figure 11 - Playbook successfully executed 1

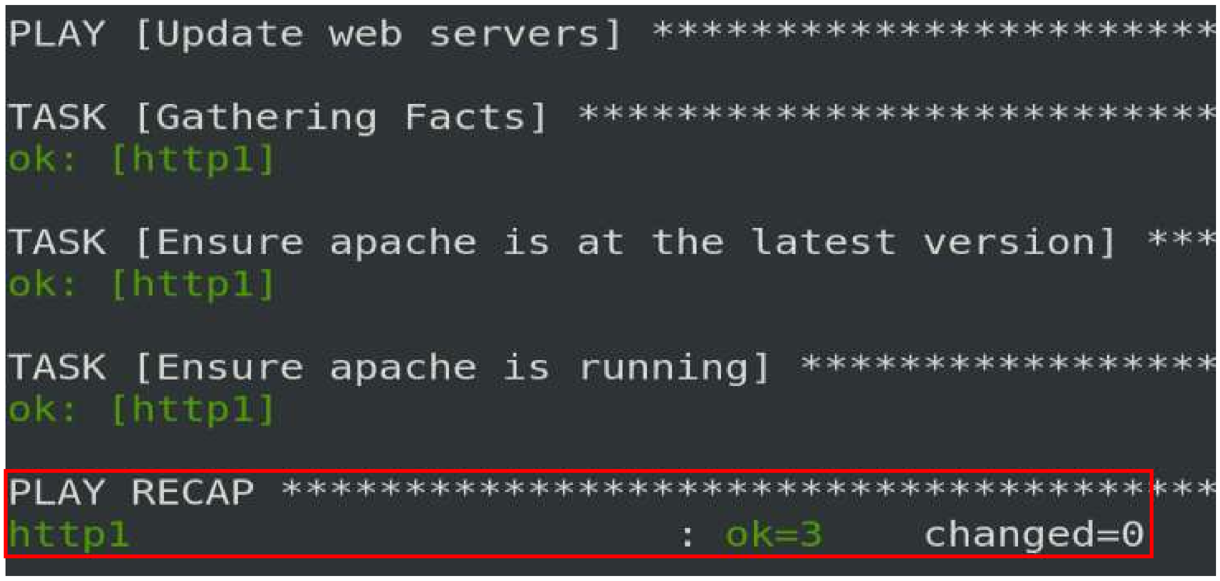
As we can see in the “play recap”, the playbook has been executed successfully on all hosts in the “apache” group (only one in this example, a web server whose hostname is “http1”).

We can see the name of each task that has been performed with the status related to it :

- **ok** : If the execution of a task was successful.
- **changed** : If the state of the target server was modified by a task (installation, update, etc.).
- **unreachable** : If the target server was unreachable (most likely a login or authentication issue).

- **failed** : If a task was considered to be a failure according to conditions in the playbook.
- **skipped** : If a task was skipped according to conditions in the playbook.
- **rescued** : If an error occurred in a block of code but a “rescue” task solved the problem (considered successful).
- **ignored** : If an unsuccessful task was considered to be ignored in the playbook.

If the same playbook is run again, we would get the following output :



```
PLAY [Update web servers] ***********************

TASK [Gathering Facts] **************************
ok: [http1]

TASK [Ensure apache is at the latest version] ***
ok: [http1]

TASK [Ensure apache is running] *****************
ok: [http1]

PLAY RECAP **************************************
http1                      : ok=3    changed=0
```

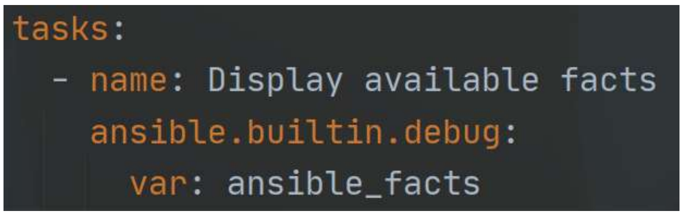
Figure 12 - Playbook successfully executed 2

We can see that the playbook was executed correctly, and nothing changed on the server as the modifications have already been taken into account with the previous command.

This was a simple example, but there are plenty of other possibilities with Ansible to get the most optimized automation depending on our needs.

2.2.5. Conditionals

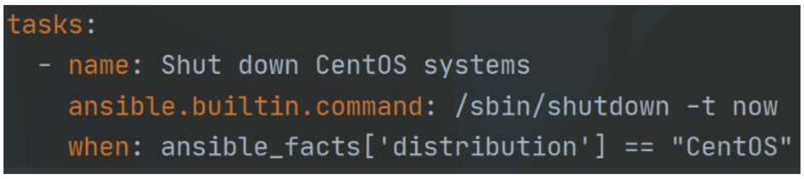
Using the “when” keyword, a condition can be applied to a task. For example, a task can be executed only when a certain service is enabled. It is also possible to use the “ansible_facts” variable, which helps to decide whether to execute a task or not based on a fact. A fact corresponds to an attribute of the node.



```yaml
tasks:
  - name: Display available facts
    ansible.builtin.debug:
      var: ansible_facts
```

Figure 13 - Display Ansible facts

We can display all the available facts of a machine with the “ansible_facts” variable like in the picture above. Ansible then shows all facts directly in the console.

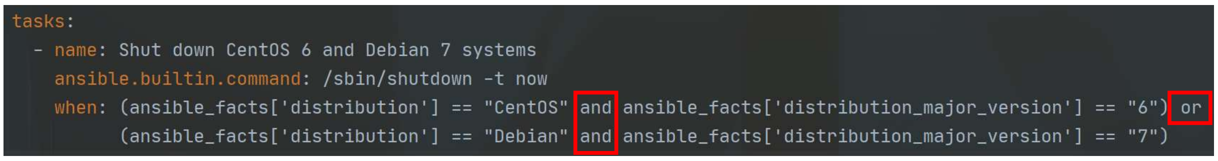


```yaml
tasks:
  - name: Shut down CentOS systems
    ansible.builtin.command: /sbin/shutdown -t now
    when: ansible_facts['distribution'] == "CentOS"
```

Figure 14 - Ansible distribution fact

In the picture above for instance, we use the “distribution” fact to shut down all machines selected if their OS distribution is “CentOS”.

Playbooks can be enhanced with “and”, ”or” and “when” to add more than one condition to a single task like in the example below.

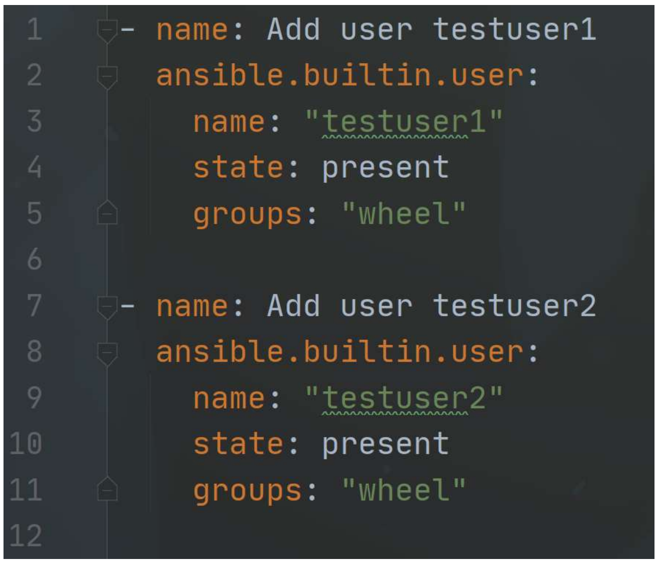


```yaml
tasks:
  - name: Shut down CentOS 6 and Debian 7 systems
    ansible.builtin.command: /sbin/shutdown -t now
    when: (ansible_facts['distribution'] == "CentOS" and ansible_facts['distribution_major_version'] == "6") or
          (ansible_facts['distribution'] == "Debian" and ansible_facts['distribution_major_version']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

Figure 15 - Conditions and/or

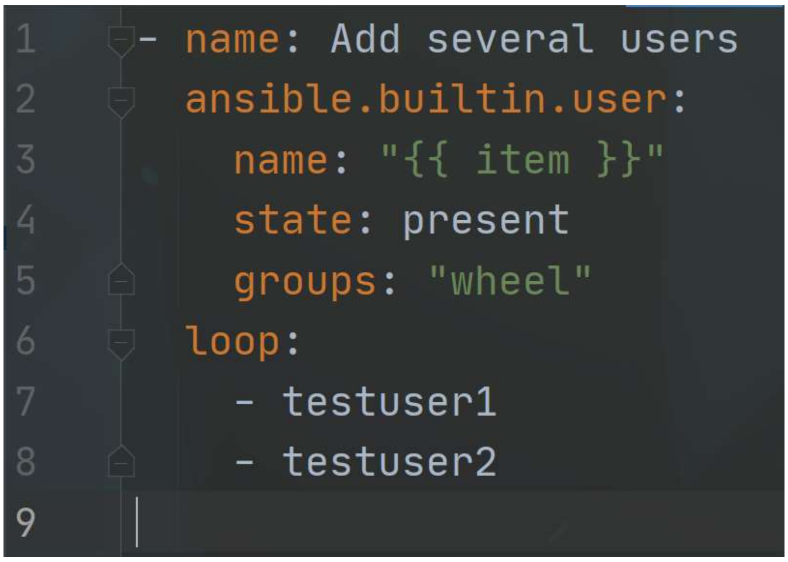
2.2.6. Loops

Ansible provides the “loop” keyword to execute a task multiple times.



```yaml
- name: Add user testuser1
  ansible.builtin.user:
    name: "testuser1"
    state: present
    groups: "wheel"

- name: Add user testuser2
  ansible.builtin.user:
    name: "testuser2"
    state: present
    groups: "wheel"

```

Figure 16 - Ansible user creation (classic)



```
1 - name: Add several users
2   ansible.builtin.user:
3     name: "{{ item }}"
4     state: present
5     groups: "wheel"
6   loop:
7     - testuser1
8     - testuser2
9
```

Figure 17 - Ansible user creation (loop)

Above are two samples of code doing the exact same thing : Creating two new users. On the second picture however, we use a loop on row 6 that iterates through the elements of the list below it. The code from row 2 to 5 then repeats itself, replacing the “{{ item }}” variable by the value of each element of the list. A list can also be provided in a separate variables file.

More complex loops are also available, such as the possibility to repeat a task a certain number of times until a condition is met or iterating over a nested list. More information can be found in the official Ansible documentation.

2.2.7. Roles

A role is a tree structure composed of directories and YAML configuration files. Roles are used to group multiple tasks together into one container (role) to optimize the automation with an organized directory structure allowing dependencies. If a role structure is generic enough, it can be shared with other users so they can use the same role in their own Ansible architecture. In the same way, an external role can be imported if it fits our needs and does not need too many changes to be used efficiently in our automation.

A role can be created manually by simply using the Linux command “mkdir” (“make directory”) to set up a directory named “roles” containing each of the different roles we want to use. Below is an example of a role that would be contained in the “roles” folder.

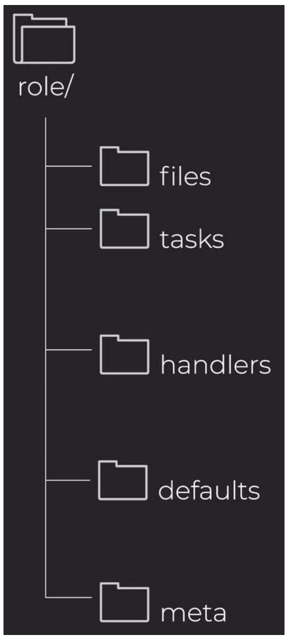


Figure 18 - Ansible role structure (Domont, 2021)

Ansible also provides the possibility to use the command “ansible-galaxy” to create roles. Ansible Galaxy is a free online repository for finding and downloading roles and collections that can be used in an automation.

2.2.8. Modules

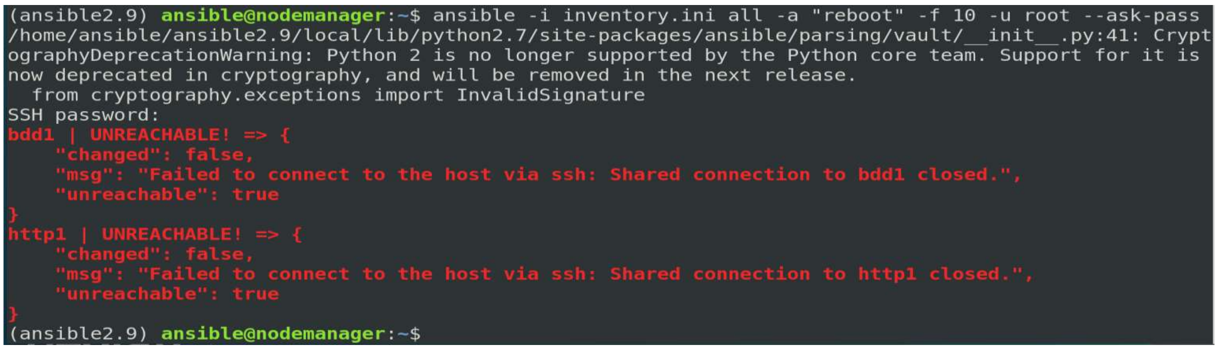
Ansible includes a great number of modules (module library) that can be executed through playbooks or directly on the nodes. Users can also write their own modules.

A module is a small piece of code that can be run from a command line or in a playbook task. For instance, commands such as “yum” or “reboot” are modules that are managed by Ansible if present.

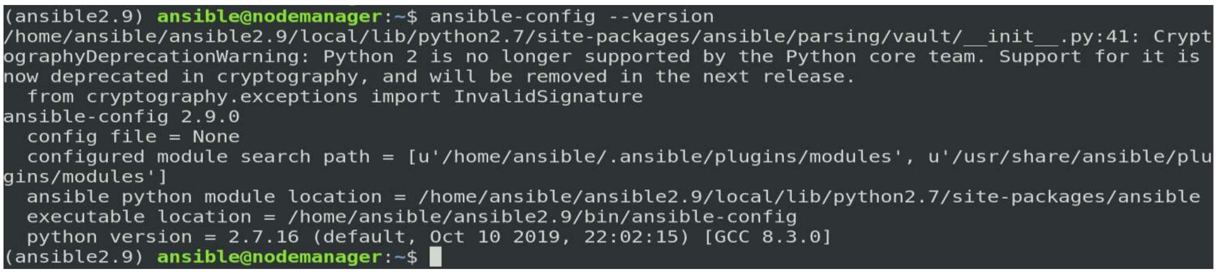
2.2.9. Commands

Below are the main commands provided by the Ansible software to manage remote nodes and use other tools.

- **ansible** : Run a single task on a set of nodes (depending on inventory file selected).


```
(ansible2.9) ansible@nodemanager:~$ ansible -i inventory.ini all -a "reboot" -f 10 -u root --ask-pass
/home/ansible/ansible2.9/local/lib/python2.7/site-packages/ansible/parsing/vault/__init__.py:41: Crypt
ographyDeprecationWarning: Python 2 is no longer supported by the Python core team. Support for it is
now deprecated in cryptography, and will be removed in the next release.
  from cryptography.exceptions import InvalidSignature
SSH password:
bdd1 | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh: Shared connection to bdd1 closed.",
    "unreachable": true
}
http1 | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh: Shared connection to http1 closed.",
    "unreachable": true
}
(ansible2.9) ansible@nodemanager:~$
```
Figure 19 - ansible ad hoc command example

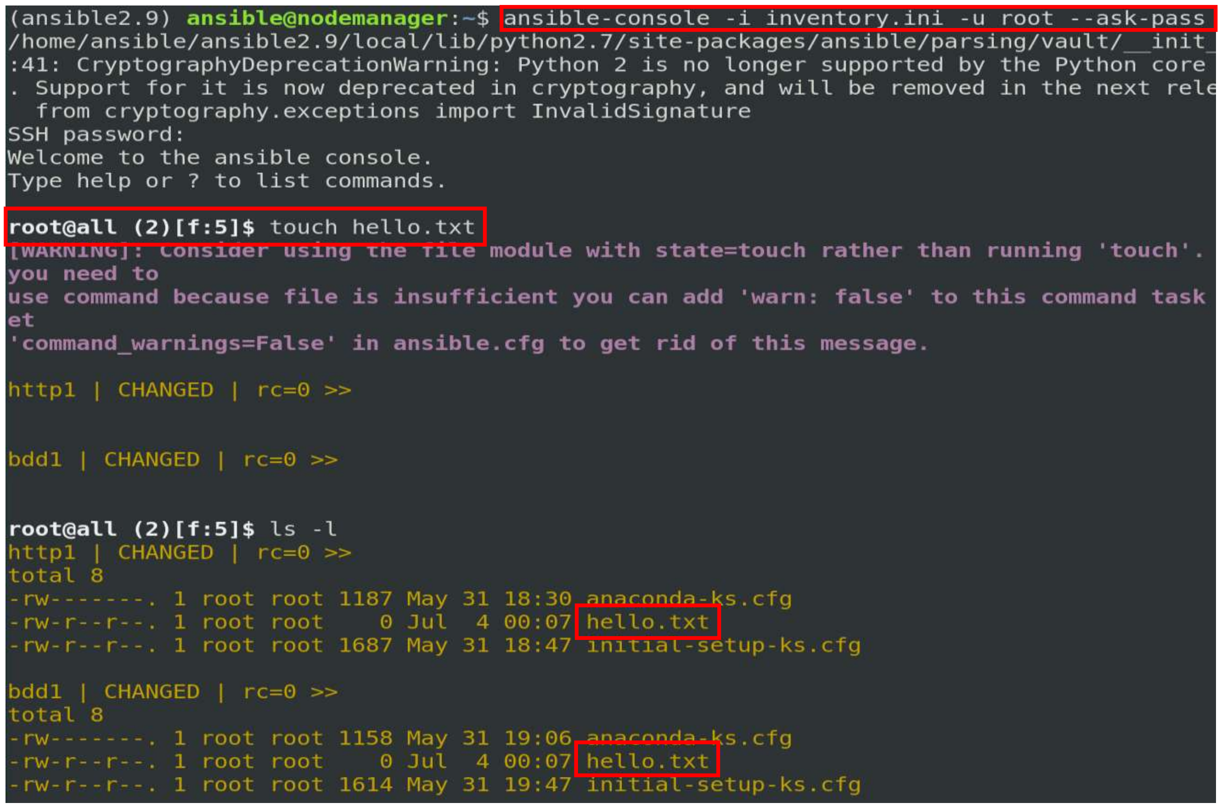
- **ansible-config** : Display configuration settings.



```
(ansible2.9) ansible@nodemanager:~$ ansible-config --version
/home/ansible/ansible2.9/local/lib/python2.7/site-packages/ansible/parsing/vault/__init__.py:41: Crypt
ographyDeprecationWarning: Python 2 is no longer supported by the Python core team. Support for it is
now deprecated in cryptography, and will be removed in the next release.
  from cryptography.exceptions import InvalidSignature
ansible-config 2.9.0
  config file = None
  configured module search path = [u'/home/ansible/.ansible/plugins/modules', u'/usr/share/ansible/plu
gins/modules']
  ansible python module location = /home/ansible/ansible2.9/local/lib/python2.7/site-packages/ansible
  executable location = /home/ansible/ansible2.9/bin/ansible-config
  python version = 2.7.16 (default, Oct 10 2019, 22:02:15) [GCC 8.3.0]
(ansible2.9) ansible@nodemanager:~$
```

Figure 20 - ansible-config example

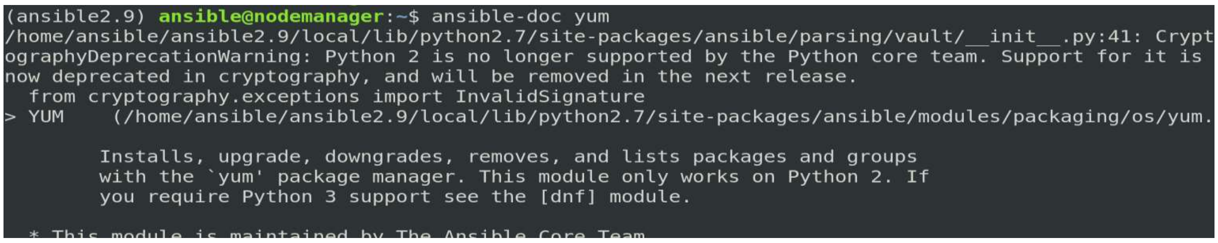
- **ansible-console** : Execute commands directly on a set of nodes (depending on inventory file selected).



```
(ansible2.9) ansible@nodemanager:~$ ansible-console -i inventory.ini -u root --ask-pass
/home/ansible/ansible2.9/local/lib/python2.7/site-packages/ansible/parsing/vault/__init_
:41: CryptographyDeprecationWarning: Python 2 is no longer supported by the Python core
. Support for it is now deprecated in cryptography, and will be removed in the next rele
  from cryptography.exceptions import InvalidSignature
SSH password:
Welcome to the ansible console.
Type help or ? to list commands.

root@all (2)[f:5]$ touch hello.txt
[WARNING]: Consider using the file module with state=touch rather than running 'touch'.
you need to
use command because file is insufficient you can add 'warn: false' to this command task
et
'command_warnings=False' in ansible.cfg to get rid of this message.

http1 | CHANGED | rc=0 >>


bdd1 | CHANGED | rc=0 >>


root@all (2)[f:5]$ ls -l
http1 | CHANGED | rc=0 >>
total 8
-rw-------. 1 root root 1187 May 31 18:30 anaconda-ks.cfg
-rw-r--r--. 1 root root    0 Jul  4 00:07 hello.txt
-rw-r--r--. 1 root root 1687 May 31 18:47 initial-setup-ks.cfg

bdd1 | CHANGED | rc=0 >>
total 8
-rw-------. 1 root root 1158 May 31 19:06 anaconda-ks.cfg
-rw-r--r--. 1 root root    0 Jul  4 00:07 hello.txt
-rw-r--r--. 1 root root 1614 May 31 19:47 initial-setup-ks.cfg
```

Figure 21 - ansible-console example

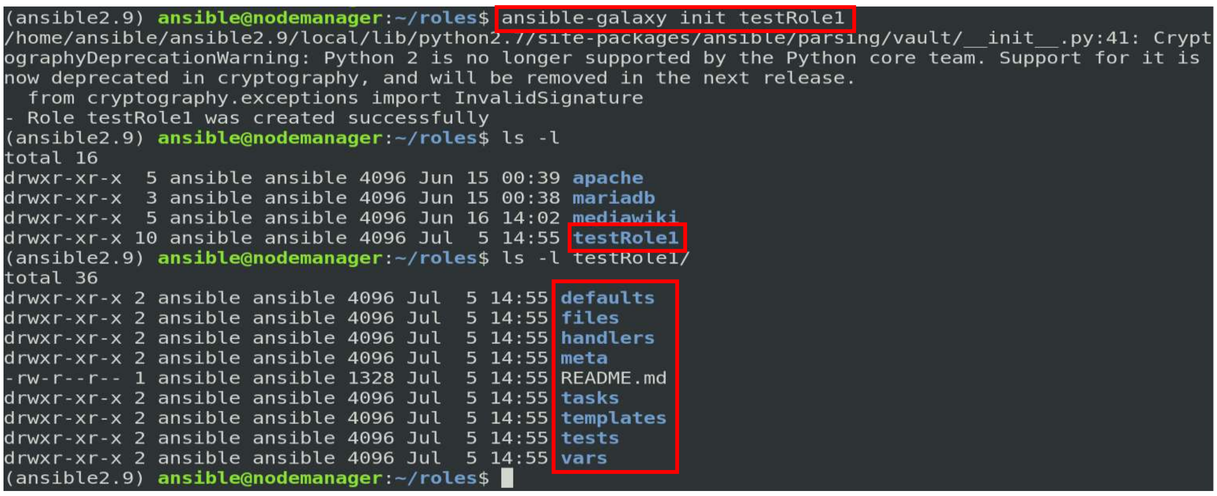
- **ansible-doc** : Display information on any module.



```
(ansible2.9) ansible@nodemanager:~$ ansible-doc yum
/home/ansible/ansible2.9/local/lib/python2.7/site-packages/ansible/parsing/vault/__init__.py:41: Crypt
ographyDeprecationWarning: Python 2 is no longer supported by the Python core team. Support for it is
now deprecated in cryptography, and will be removed in the next release.
  from cryptography.exceptions import InvalidSignature
> YUM    (/home/ansible/ansible2.9/local/lib/python2.7/site-packages/ansible/modules/packaging/os/yum.

        Installs, upgrade, downgrades, removes, and lists packages and groups
        with the `yum' package manager. This module only works on Python 2. If
        you require Python 3 support see the [dnf] module.

  * This module is maintained by The Ansible Core Team
```

Figure 22 - ansible-doc example

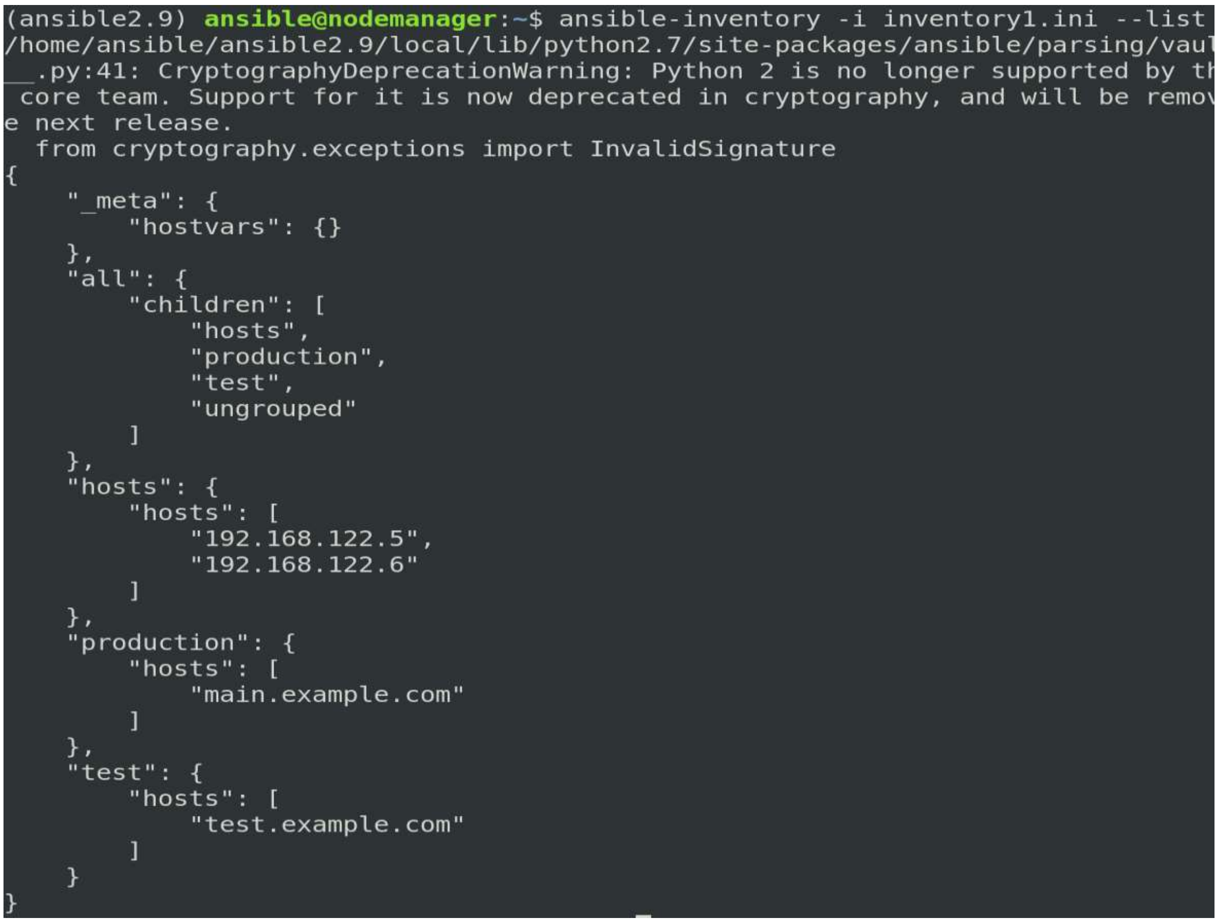
- **ansible-galaxy** : Create and manage roles using the repository for Ansible roles and collections (Ansible Galaxy).



```
(ansible2.9) ansible@nodemanager:~/roles$ ansible-galaxy init testRole1
/home/ansible/ansible2.9/local/lib/python2.7/site-packages/ansible/parsing/vault/__init__.py:41: Crypt
ographyDeprecationWarning: Python 2 is no longer supported by the Python core team. Support for it is
now deprecated in cryptography, and will be removed in the next release.
  from cryptography.exceptions import InvalidSignature
- Role testRole1 was created successfully
(ansible2.9) ansible@nodemanager:~/roles$ ls -l
total 16
drwxr-xr-x  5 ansible ansible 4096 Jun 15 00:39 apache
drwxr-xr-x  3 ansible ansible 4096 Jun 15 00:38 mariadb
drwxr-xr-x  5 ansible ansible 4096 Jun 16 14:02 mediawiki
drwxr-xr-x 10 ansible ansible 4096 Jul  5 14:55 testRole1
(ansible2.9) ansible@nodemanager:~/roles$ ls -l testRole1/
total 36
drwxr-xr-x 2 ansible ansible 4096 Jul  5 14:55 defaults
drwxr-xr-x 2 ansible ansible 4096 Jul  5 14:55 files
drwxr-xr-x 2 ansible ansible 4096 Jul  5 14:55 handlers
drwxr-xr-x 2 ansible ansible 4096 Jul  5 14:55 meta
-rw-r--r-- 1 ansible ansible 1328 Jul  5 14:55 README.md
drwxr-xr-x 2 ansible ansible 4096 Jul  5 14:55 tasks
drwxr-xr-x 2 ansible ansible 4096 Jul  5 14:55 templates
drwxr-xr-x 2 ansible ansible 4096 Jul  5 14:55 tests
drwxr-xr-x 2 ansible ansible 4096 Jul  5 14:55 vars
(ansible2.9) ansible@nodemanager:~/roles$
```

Figure 23 - ansible-galaxy example

- **ansible-inventory** : Display inventory information in a JSON format.



```
(ansible2.9) ansible@nodemanager:~$ ansible-inventory -i inventory1.ini --list
/home/ansible/ansible2.9/local/lib/python2.7/site-packages/ansible/parsing/vaul
__.py:41: CryptographyDeprecationWarning: Python 2 is no longer supported by th
 core team. Support for it is now deprecated in cryptography, and will be remov
e next release.
  from cryptography.exceptions import InvalidSignature
{
    "_meta": {
        "hostvars": {}
    },
    "all": {
        "children": [
            "hosts",
            "production",
            "test",
            "ungrouped"
        ]
    },
    "hosts": {
        "hosts": [
            "192.168.122.5",
            "192.168.122.6"
        ]
    },
    "production": {
        "hosts": [
            "main.example.com"
        ]
    },
    "test": {
        "hosts": [
            "test.example.com"
        ]
    }
}
```

Figure 24 - ansible-inventory example

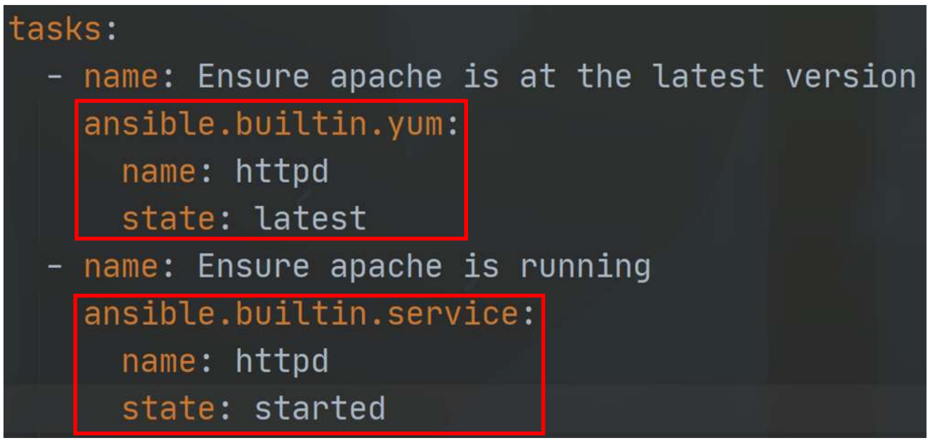
- **ansible-playbook** : Execute a playbook.

- **ansible-pull** : Pull a remote copy of ansible on each managed server if a VCS has been set up. This can be used to change the default push architecture to a pull architecture.

- **ansible-vault** : Encrypt or decrypt Ansible structured data files.

2.2.10. Language

Ansible uses a procedural language, which means that each step to achieve a goal must be specified.



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

Figure 25 - Ansible procedural language example

In the sample of code above, we can clearly see the specific instructions used in playbooks. Each step to achieve a goal is written in detail. The command to use ("yum", "service") is specified with the package name and state as well.

Ansible scripts run using parallel execution on the agents by default.

2.2.11. Ansible pros

- **Easy to learn and configure**

Ansible's architecture is easy to understand and learn. The YAML configuration files are simple to read and allow operators to quickly start writing code. Ansible is also easy and fast to install, configure and manage.

- **Agentless**

Ansible does not rely on any "agents", which means nothing needs to be installed on the nodes. It works as a "push configuration", the server pushes commands on the nodes directly (different from "pull configuration"). The lack of any agents also allows easy management for older networking devices for which it may be difficult or impossible to install specific software on.

- **Ansible Galaxy service**

As mentioned previously, Ansible Galaxy is a repository for predefined and custom roles and collections that a user can download into his Ansible configuration directly. The command "ansible-galaxy" also offers an easy way to create new roles.

- **Good documentation**

The official Ansible website offers a good documentation on the software with good explanations and examples. It is however not as complete as other software created beforehand.

2.2.12. Ansible cons

- **Limited support for Windows**

Ansible cannot run on a Windows machine. Ansible may only manage Windows machines from a Linux host, although it is less convenient that managing Linux machines. However, it is possible to manage nodes with Ansible on a Windows OS using the Windows Subsystem for Linux (WSL).

- **YAML configuration files**

While YAML is easy to learn, superior to JSON for configuration services and even supports comments directly in the files, it still is a difficult language to debug, and you have with Ansible no way to know if a specific task will fail until it is executed. Actual programming languages such as Ruby are more powerful in that regard and can handle more complex tasks.

2.2.13. When to use Ansible

Ansible is a fair choice for a company that wants a software that is easy to install, configure and manage. The complexity is however limited due to the YAML language being a data serialization language rather than a programming language.

Furthermore, if operators do not know or do not necessarily have the time to learn a new programming language such as Ruby, Ansible offers a system using YAML configuration files, which are very simple to handle even for a non-programmer.

The fact that Ansible is agentless permits a one-way communication with the agents, which requires less resources. It is also preferable if the company possesses older networking equipment that does not support the installation of any agents on them (routers and switches for instance). Managed nodes have however no way to report back to the server in case of a failure on them that Ansible did not detect directly after the playbook execution.

Ansible's infrastructure as code is procedural, which may require more complex management in terms of scaling, but the simplicity of YAML compensates for that.

Ansible is owned by Red Hat, which can make it a better choice for a company if Red Hat products are already being used in their infrastructure, such as Red Hat Enterprise Linux (RHEL) and CentOS operating systems.

2.3. Chef

2.3.1. Architecture

Chef (previously known as Opscode) is an open-source configuration management software developed by the company of the same name and written in the Ruby programming language. It was initially released in 2009 and uses a Ruby-based domain-specific language (DSL). A DSL, as its name implies, is specialized to a particular use.

Users can either use Chef Infra, which is the main automation platform whose architecture is described below, or chef-solo, a command line tool that does not require the configuration of a Chef Infra server. However, because of that fact, chef-solo does not provide centralized distribution of "cookbooks" (collection of configuration files) to the nodes (No need for a main server, only a workstation).

Using Chef Infra, a system administrator would typically work from their workstation and issue commands from there and upload configuration files ("cookbooks") to a server. The latter would then load these "cookbooks" to the correct nodes that request a configuration update.

Chef is not agentless like Ansible, thus a Chef client must be installed on all managed nodes.

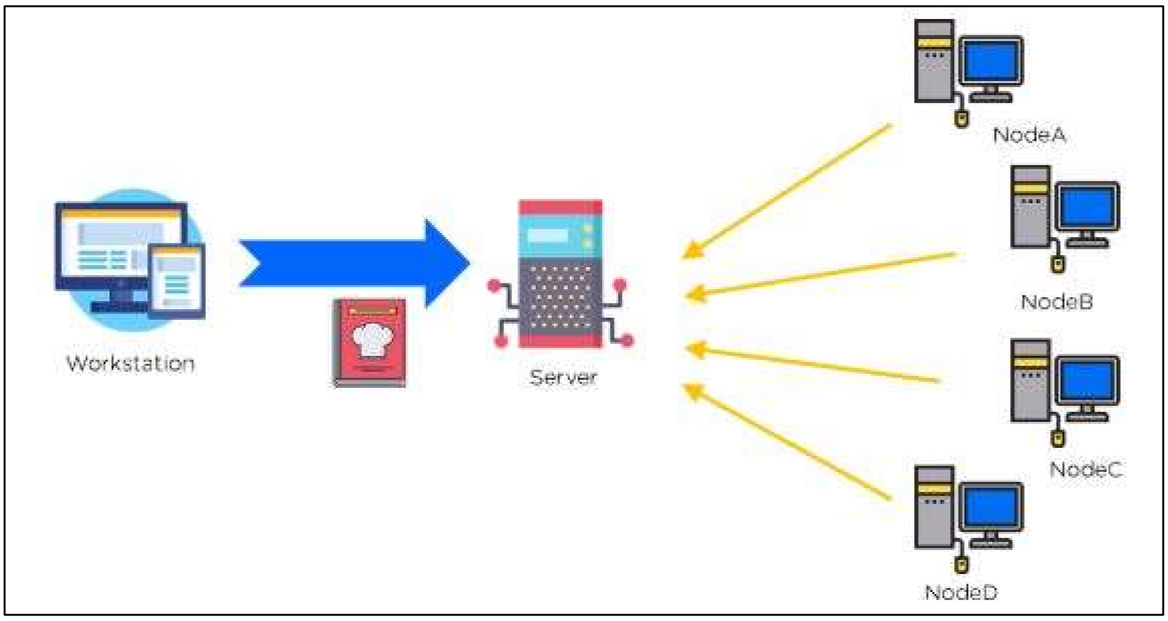


Figure 26 - Chef architecture (Gaba, 2021)

To show the different features of this software, we will use chef Infra in the different examples.

The professional Chef version, named "Chef Enterprise" is free to use to manage up to 5 nodes, 2 users, and with no support included.

2.3.2. Chef Workstation

A Chef workstation is where a user may create recipes, attributes, templates and other components to produce cookbooks. The Chef workstation may be installed on a Linux, macOS or Windows machine.

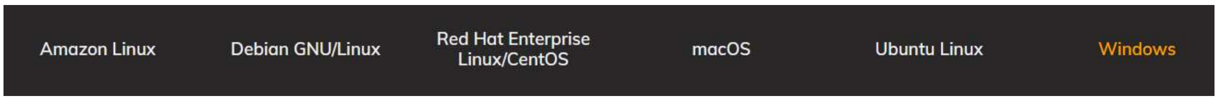
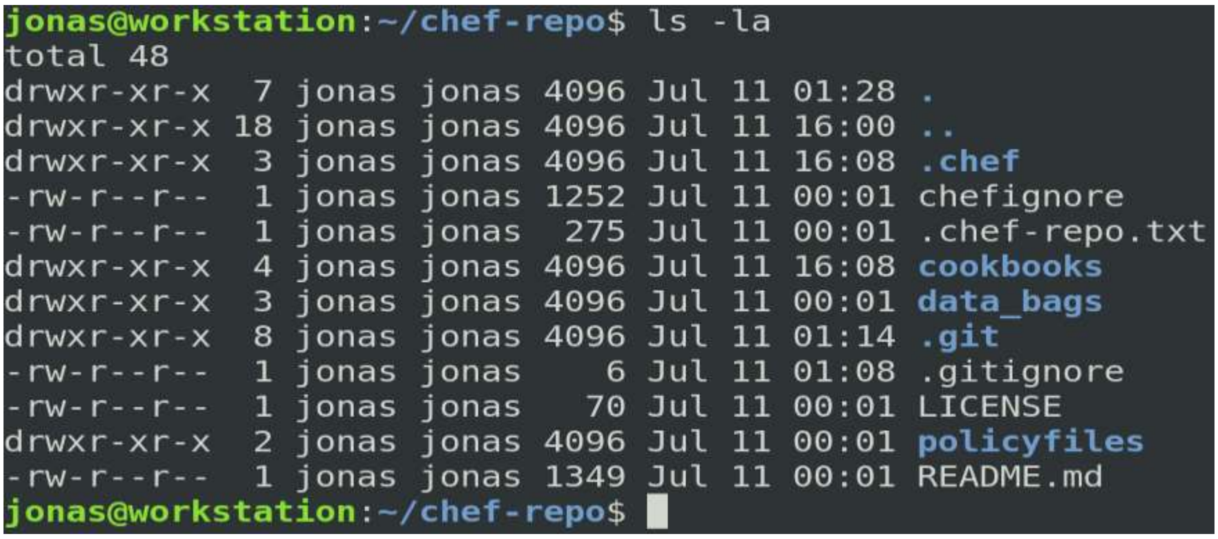


Figure 27 - Supported OS for a Chef workstation (Chef Downloads, n.d.)

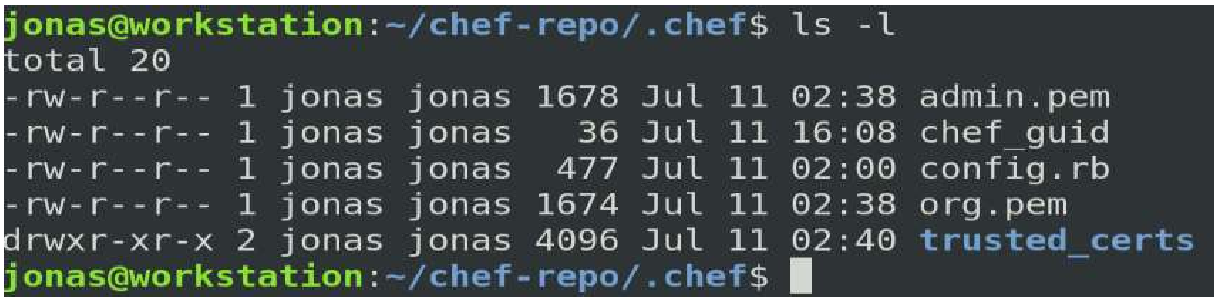
The workstation contains a directory named “chef-repo”, which is where cookbooks can be created and managed. Any components contained in a cookbook are also stored there. This directory is very important and as such a VCS should be configured to avoid any data loss and gain version control.



```
jonas@workstation:~/chef-repo$ ls -la
total 48
drwxr-xr-x  7 jonas jonas 4096 Jul 11 01:28 .
drwxr-xr-x 18 jonas jonas 4096 Jul 11 16:00 ..
drwxr-xr-x  3 jonas jonas 4096 Jul 11 16:08 .chef
-rw-r--r--  1 jonas jonas 1252 Jul 11 00:01 chefignore
-rw-r--r--  1 jonas jonas  275 Jul 11 00:01 .chef-repo.txt
drwxr-xr-x  4 jonas jonas 4096 Jul 11 16:08 cookbooks
drwxr-xr-x  3 jonas jonas 4096 Jul 11 00:01 data_bags
drwxr-xr-x  8 jonas jonas 4096 Jul 11 01:14 .git
-rw-r--r--  1 jonas jonas    6 Jul 11 01:08 .gitignore
-rw-r--r--  1 jonas jonas   70 Jul 11 00:01 LICENSE
drwxr-xr-x  2 jonas jonas 4096 Jul 11 00:01 policyfiles
-rw-r--r--  1 jonas jonas 1349 Jul 11 00:01 README.md
jonas@workstation:~/chef-repo$
```

Figure 28 - chef-repo structure

“.chef” contains various Privacy Enhanced Mail (PEM) files, which consist of private keys imported from the Chef server. They were created along with the user and the organization directly on the Chef server. This hidden directory (starting with “.”) also contains another folder named “trusted_certs” with certificate (CRT) files that the workstation trusts. We will talk more about these certificates in the “Installation” section below. Finally, “.chef” contains its unique globally unique identifier (GUID) and a “config.rb” file written in Ruby containing information about the Chef server.



```
jonas@workstation:~/chef-repo/.chef$ ls -l
total 20
-rw-r--r-- 1 jonas jonas 1678 Jul 11 02:38 admin.pem
-rw-r--r-- 1 jonas jonas   36 Jul 11 16:08 chef_guid
-rw-r--r-- 1 jonas jonas  477 Jul 11 02:00 config.rb
-rw-r--r-- 1 jonas jonas 1674 Jul 11 02:38 org.pem
drwxr-xr-x 2 jonas jonas 4096 Jul 11 02:40 trusted_certs
jonas@workstation:~/chef-repo/.chef$
```

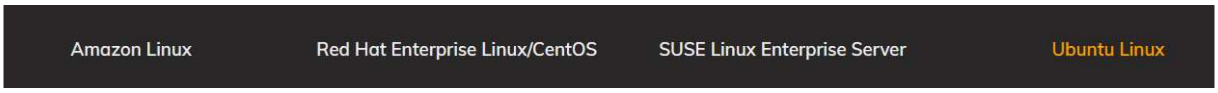
Figure 29 - .chef repo structure

Going back to the parent folder “chef-repo”, we can now take a look at the “cookbooks” directory. It contains all components needed in the making of a cookbook. We will talk more about these in the “Cookbooks” section below.

The main command used on a Chef workstation is “knife”. It is used as an interface between the workstation and the Chef server and helps managing nodes, cookbooks, recipes, roles, and other resources.

2.3.3. Chef Server

The Chef server is the link between the workstations and the nodes. Communication is ensured using public key encryption to guarantee that only trusted machines can communicate with the Chef server. The Chef server may only be installed on a Linux machine.



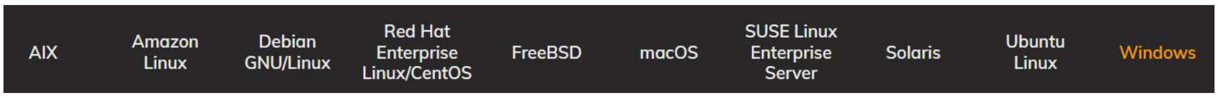
Amazon Linux	Red Hat Enterprise Linux/CentOS	SUSE Linux Enterprise Server	Ubuntu Linux

Figure 30 - Supported OS for a Chef server (Chef Downloads, n.d.)

We can use the command “chef-server-ctl” with any existing subcommand to execute configuration and maintenance tasks on the Chef server.

2.3.4. Chef Client

A Chef client is what needs to be installed on all nodes that must be managed by Chef. The Chef client may be installed on a Linux, macOS or Windows machine.



AIX	Amazon Linux	Debian GNU/Linux	Red Hat Enterprise Linux/CentOS	FreeBSD	macOS	SUSE Linux Enterprise Server	Solaris	Ubuntu Linux	Windows

Figure 31 - Supported OS for a Chef client (Chef Downloads, n.d.)

2.3.5. Installation

Installing Chef requires more time than with Ansible, as the architecture is more complex.

First, the Chef workstation, server and clients must be installed on all machines that are part of our Chef architecture. Note that it is possible to have more than one workstation.

- **Chef Server**

The Chef server needs to be linked to a workstation and nodes it manages. To achieve this goal, we use different users. The only existing user after the Chef server installation is the default superuser named “pivotal”, although we can create our own using the following command :

sudo* chef-*server-ctl user-create [username] [firstname] [lastname] [email] ‘[password]’ --filename ~/.chef/[username].pem

We also need an organization which to add our new user. An organization is the top-level entity for role-based access in the Chef server. It can be done using the following command :

sudo* chef-*server-ctl org-create [org name] "[org full name]" --association_user [username] --filename ~/.chef/[org name].pem

The .pem files created will be passed to the workstation later.

- **Chef Workstation**

On the Chef workstation, the first step is to create a chef directory using the following command :

chef generate* repo *chef-repo

The DNS (or the "/etc/hosts" file as a temporary measure) needs to be configured so that the workstation can communicate with the server and nodes.

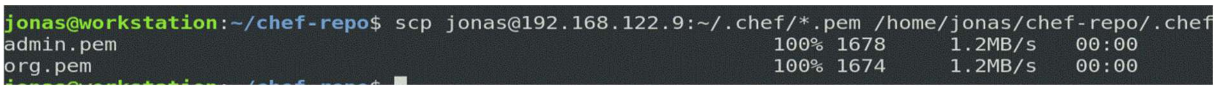
An RSA key-pair must be created to gain access to the Chef server :

ssh-keygen -b 4096

The public key is then sent to the server :

ssh-copy*-id *[Chef username]@[server ip address]

Inside the recently created "chef-repo", the hidden subdirectory ".chef" needs to be added to store the RSA private keys that we can fetch from the Chef server (user and organization keys).

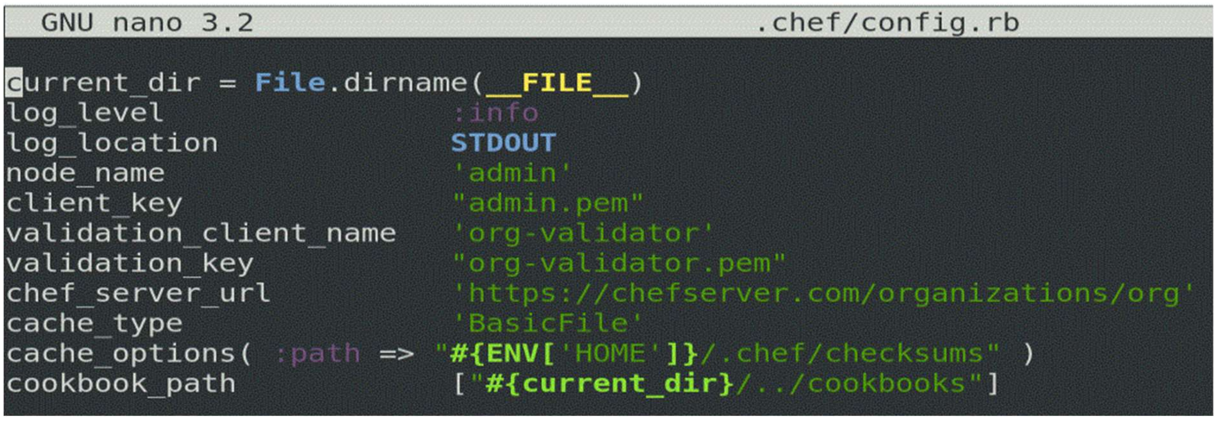


```
jonas@workstation:~/chef-repo$ scp jonas@192.168.122.9:~/.chef/*.pem /home/jonas/chef-repo/.chef
admin.pem                                                   100% 1678     1.2MB/s   00:00
org.pem                                                     100% 1674     1.2MB/s   00:00
```

Figure 32 - scp command to fetch .pem files

A VCS, such as Git, should be added too to manage the chef-repo.

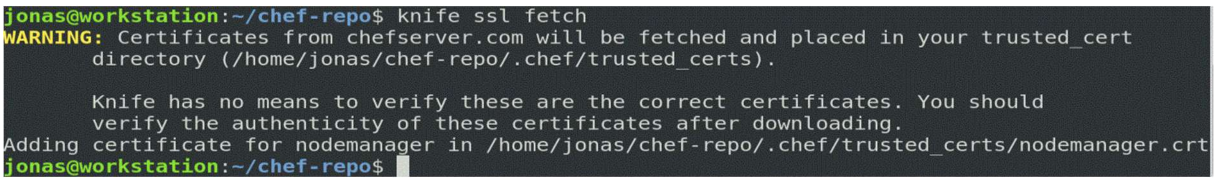
Then, a ".chef/config.rb" file must be created. It will contain information allowing the communication with the Chef server using the keys we transferred earlier.



```
  GNU nano 3.2                              .chef/config.rb

current_dir = File.dirname(__FILE__)
log_level                :info
log_location             STDOUT
node_name                'admin'
client_key               "admin.pem"
validation_client_name   'org-validator'
validation_key           "org-validator.pem"
chef_server_url          'https://chefserver.com/organizations/org'
cache_type               'BasicFile'
cache_options( :path => "#{ENV['HOME']}/.chef/checksums" )
cookbook_path            ["#{current_dir}/../cookbooks"]
```

Figure 33 - config.rb file

Finally, we can copy the self-signed Secure Sockets Layer (SSL) certificates from the server generated at the installation of the Chef server. This will add that certificate to the trusted authorities of the workstation.



```
jonas@workstation:~/chef-repo$ knife ssl fetch
WARNING: Certificates from chefserver.com will be fetched and placed in your trusted_cert
       directory (/home/jonas/chef-repo/.chef/trusted_certs).

       Knife has no means to verify these are the correct certificates. You should
       verify the authenticity of these certificates after downloading.
Adding certificate for nodemanager in /home/jonas/chef-repo/.chef/trusted_certs/nodemanager.crt
jonas@workstation:~/chef-repo$
```

Figure 34 - knife ssl fetch command

- **Chef Nodes**

The Chef client should ideally be installed from the Chef workstation using the "knife bootstrap" command :

knife bootstrap [node IP address] -x root -P [password] --node-name [node name]

SSH must be installed and enabled on each node.

2.3.6. Recipes

A recipe is a set of instructions written in Ruby for one specific task. For instance, a recipe can be configured to deploy custom software to any managed nodes.

2.3.7. Cookbooks

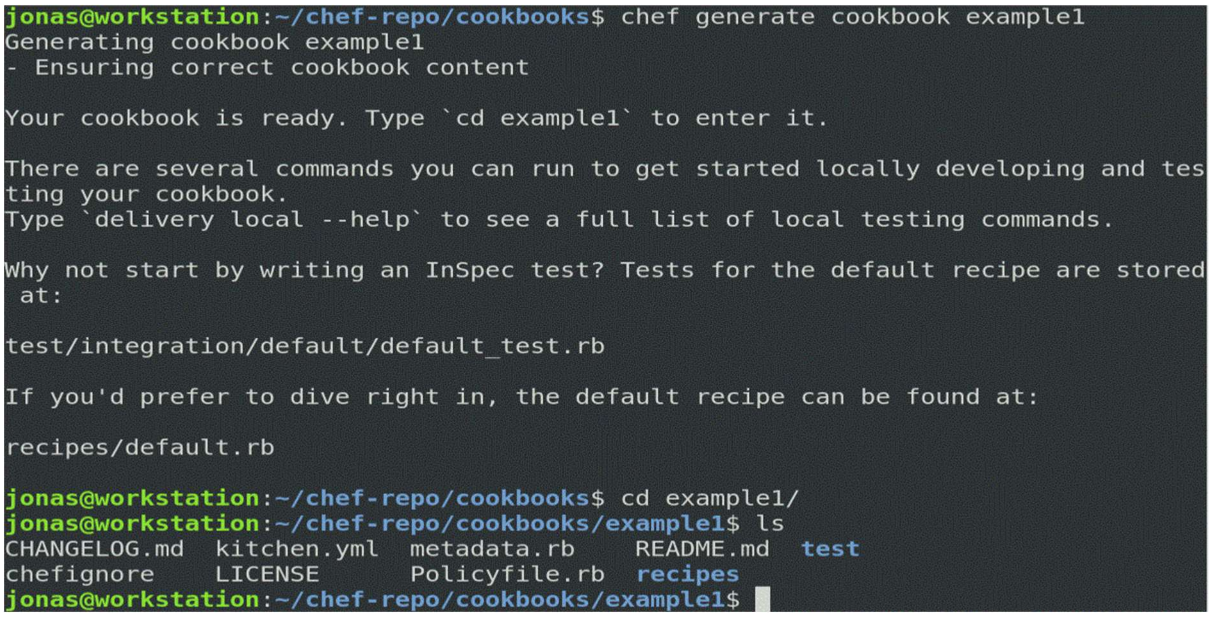
A cookbook is a collection of recipes, attributes, resources, templates, libraries and any other components allowing the creation of a functioning system. It can be compared to an Ansible role.

Users do not necessarily have to write all the code themselves ; they can also download existing cookbooks and reuse them. For instance, a Cisco cookbook exists with plenty of recipes allowing the management of Cisco devices such as routers or switches.

Cookbooks are usually created on a workstation or downloaded from the Chef Supermarket (repository of cookbooks). These configurations are pushed to the Chef server, and then pulled by the different nodes using the Chef client installed on them.

If we now take an example, let us imagine we have a Chef workstation (Debian), a Chef server (Debian) and a Chef client (CentOS), all present in the same network.

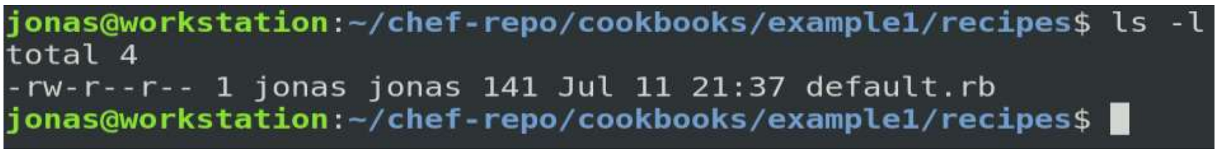
We can create a new cookbook from the workstation using the "chef generate cookbook" command.



```
jonas@workstation:~/chef-repo/cookbooks$ chef generate cookbook example1
Generating cookbook example1
- Ensuring correct cookbook content

Your cookbook is ready. Type `cd example1` to enter it.

There are several commands you can run to get started locally developing and tes
ting your cookbook.
Type `delivery local --help` to see a full list of local testing commands.

Why not start by writing an InSpec test? Tests for the default recipe are stored
 at:

test/integration/default/default_test.rb

If you'd prefer to dive right in, the default recipe can be found at:

recipes/default.rb

jonas@workstation:~/chef-repo/cookbooks$ cd example1/
jonas@workstation:~/chef-repo/cookbooks/example1$ ls
CHANGELOG.md  kitchen.yml  metadata.rb    README.md  test
chefignore    LICENSE      Policyfile.rb  recipes
jonas@workstation:~/chef-repo/cookbooks/example1$
```

Figure 35 - Cookbook creation

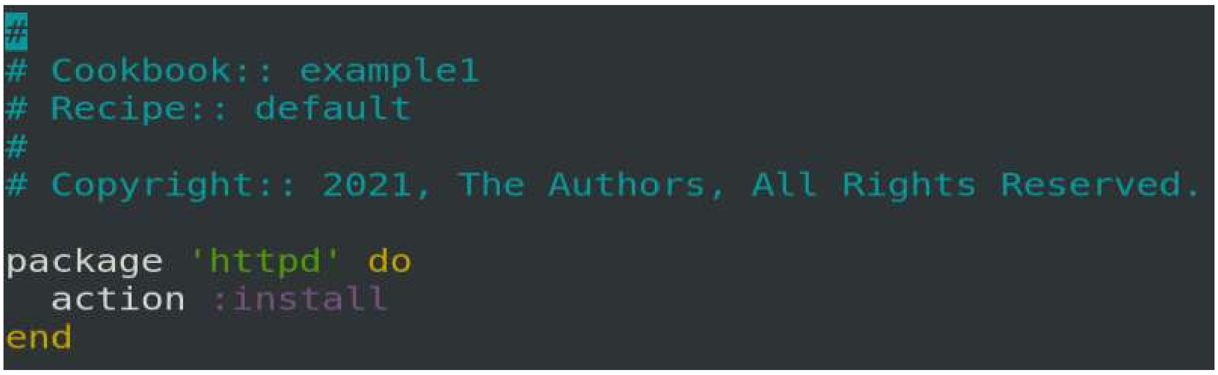
The "recipes" folder is already created with a ruby file named "default.rb" inside.



```
jonas@workstation:~/chef-repo/cookbooks/example1/recipes$ ls -l
total 4
-rw-r--r-- 1 jonas jonas 141 Jul 11 21:37 default.rb
jonas@workstation:~/chef-repo/cookbooks/example1/recipes$
```

Figure 36 - Default recipe

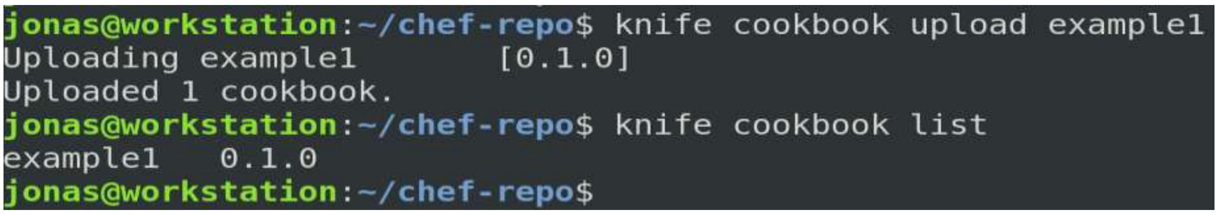
We can edit this file to install the Apache server.



```
#
# Cookbook:: example1
# Recipe:: default
#
# Copyright:: 2021, The Authors, All Rights Reserved.

package 'httpd' do
  action :install
end
```

Figure 37 - Apache installation recipe

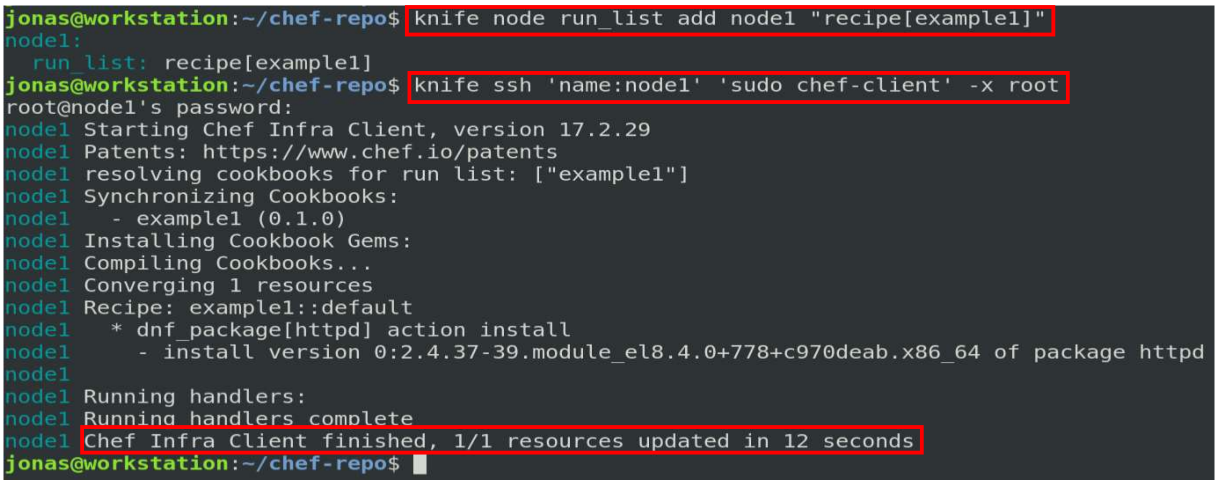
Once the recipe is ready, we can upload the whole cookbook to the VCS we set up.



```
jonas@workstation:~/chef-repo$ knife cookbook upload example1
Uploading example1       [0.1.0]
Uploaded 1 cookbook.
jonas@workstation:~/chef-repo$ knife cookbook list
example1   0.1.0
jonas@workstation:~/chef-repo$
```

Figure 38 - Cookbook uploaded to VCS

Then, we add the recipe (from our cookbook "example1") to the run-list. A run-list contains all necessary recipes to configure a node into a desired state.



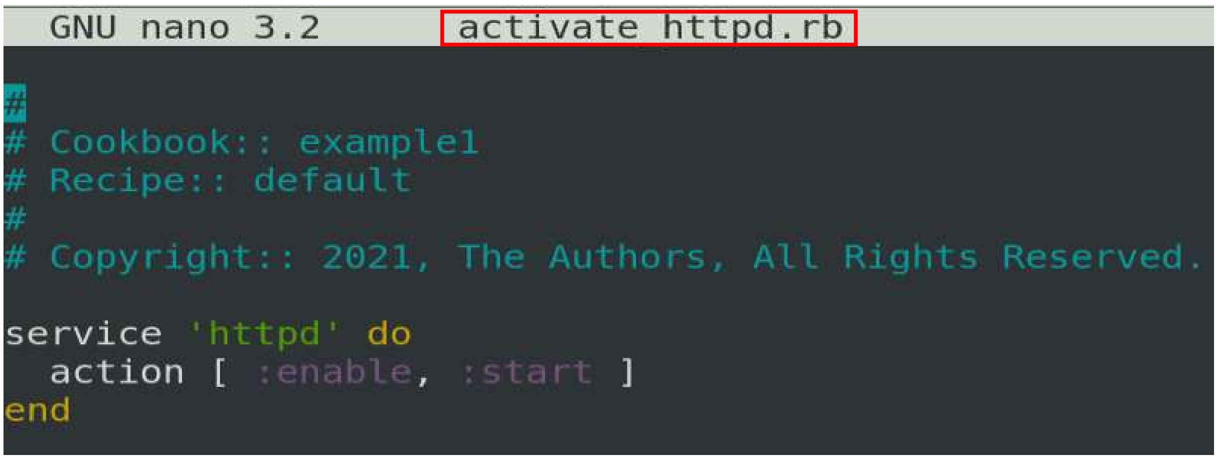
```
jonas@workstation:~/chef-repo$ knife node run_list add node1 "recipe[example1]"
node1:
  run_list: recipe[example1]
jonas@workstation:~/chef-repo$ knife ssh 'name:node1' 'sudo chef-client' -x root
root@node1's password:
node1 Starting Chef Infra Client, version 17.2.29
node1 Patents: https://www.chef.io/patents
node1 resolving cookbooks for run list: ["example1"]
node1 Synchronizing Cookbooks:
node1   - example1 (0.1.0)
node1 Installing Cookbook Gems:
node1 Compiling Cookbooks...
node1 Converging 1 resources
node1 Recipe: example1::default
node1   * dnf_package[httpd] action install
node1     - install version 0:2.4.37-39.module_el8.4.0+778+c970deab.x86_64 of package httpd
node1
node1 Running handlers:
node1 Running handlers complete
node1 Chef Infra Client finished, 1/1 resources updated in 12 seconds
jonas@workstation:~/chef-repo$
```

Figure 39 - Apache installation on the node

Finally, we can run the command "sudo chef-client" on the node so that the latter will synchronize its status with the Chef server, executing any recipes on its run-list.

Every time a change is made to any component, the cookbook must be uploaded again to the Chef server so that nodes can pull changes with the "chef-client" command.

For instance, if we decide to add the recipe below :



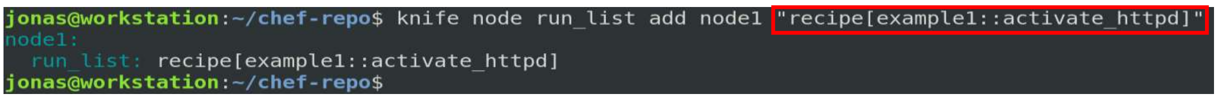
```
  GNU nano 3.2        activate_httpd.rb

#
# Cookbook:: example1
# Recipe:: default
#
# Copyright:: 2021, The Authors, All Rights Reserved.

service 'httpd' do
  action [ :enable, :start ]
end
```

Figure 40 - activate_httpd.rb recipe

We would need to run the following command again :

knife cookbook upload example1

And then, we need to specify which file to add to our run-list as it is not the default recipe :



```
jonas@workstation:~/chef-repo$ knife node run_list add node1 "recipe[example1::activate_httpd]"
node1:
  run_list: recipe[example1::activate_httpd]
jonas@workstation:~/chef-repo$
```

Figure 41 - knife node run_list add command

Obviously, many recipes can be added to the run-list at the same time. Recipes will then be executed in a certain order starting with the first one added to that run-list.

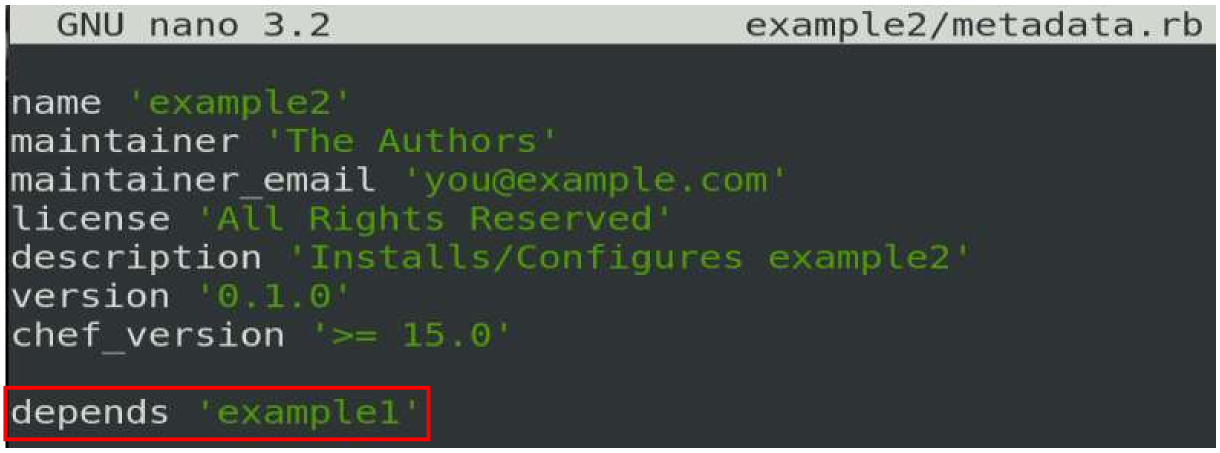
2.3.8. Dependencies

A cookbook can have a dependency on a recipe that is located in another cookbook ; in that case, the dependency must be declared in the "metadata.rb" file located in that cookbook folder.

If we imagine the following folder structure on our Chef workstation :

- **cookbooks**
  - example1
    - recipes
      - *install_httpd.rb*
  - example2
    - recipes
      - *activate_httpd.rb*
    - *metadata.rb*

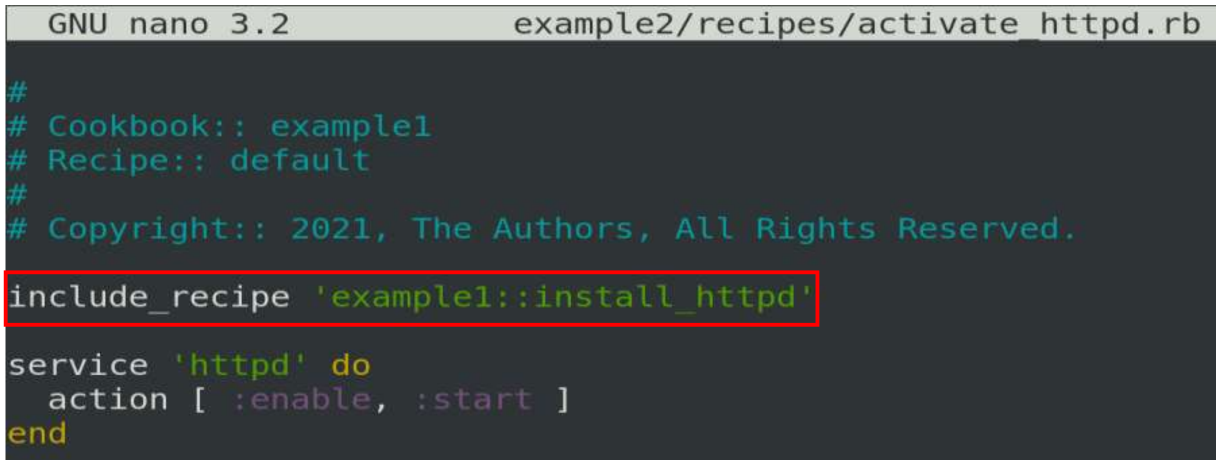
We could, using dependencies, add only the “activate_httpd.rb” recipe to the run-list even if Apache has not been installed on the node yet by using the keyword “depends” in the “metadata.rb” file of the “example2” cookbook :



```
  GNU nano 3.2                    example2/metadata.rb

name 'example2'
maintainer 'The Authors'
maintainer_email 'you@example.com'
license 'All Rights Reserved'
description 'Installs/Configures example2'
version '0.1.0'
chef_version '>= 15.0'

depends 'example1'
```

Figure 42 - depends keyword

This way, it is then possible to use the “include_recipe” keyword in a recipe file to run a script from another cookbook :



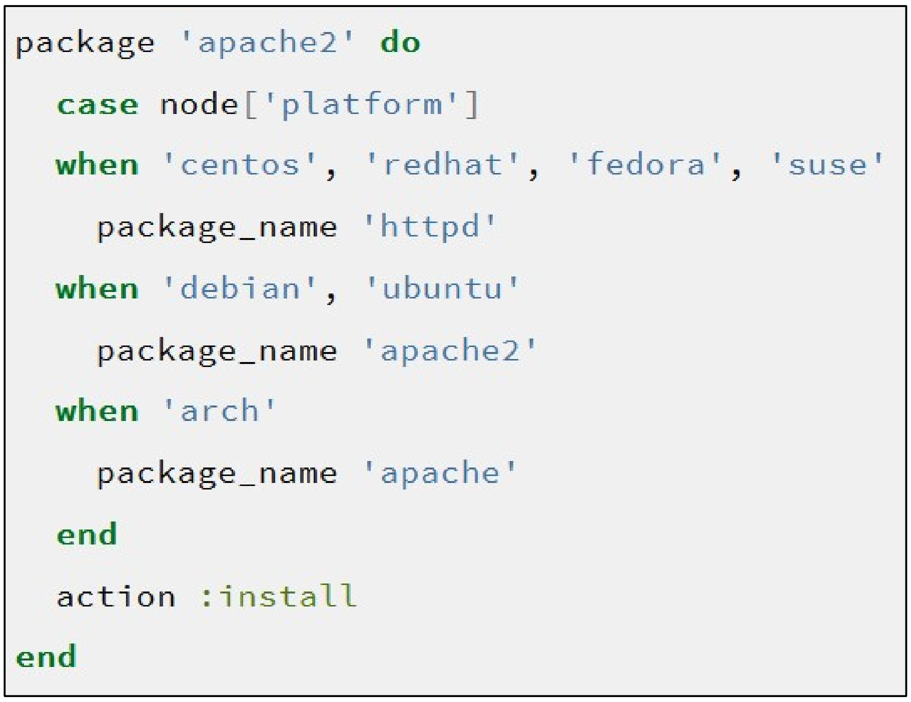
```
  GNU nano 3.2            example2/recipes/activate_httpd.rb

#
# Cookbook:: example1
# Recipe:: default
#
# Copyright:: 2021, The Authors, All Rights Reserved.

include_recipe 'example1::install_httpd'

service 'httpd' do
  action [ :enable, :start ]
end
```

Figure 43 - include_recipe keyword

If the recipe above is added to the run-list alone, Chef will first execute the “install_httpd” recipe from the other cookbook.

2.3.9. Conditionals

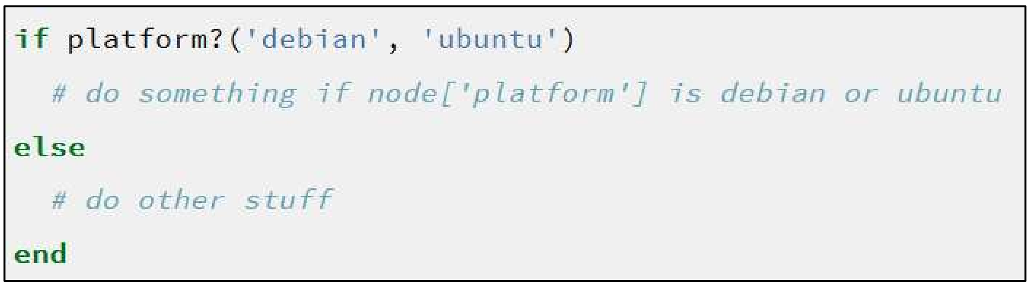
Conditionals are made possible using keywords such as “case”, “if”, “else” and “when”.



```
package 'apache2' do
  case node['platform']
  when 'centos', 'redhat', 'fedora', 'suse'
    package_name 'httpd'
  when 'debian', 'ubuntu'
    package_name 'apache2'
  when 'arch'
    package_name 'apache'
  end
  action :install
end
```

Figure 44 - Apache installation depending on platform (Chef Documentation, n.d.)

In the picture above taken from the official Chef documentation, we can see a recipe for the installation of Apache. We check the OS platform of each node and install Apache using the right package name depending on that platform name.



```
if platform?('debian', 'ubuntu')
  # do something if node['platform'] is debian or ubuntu
else
  # do other stuff
end
```

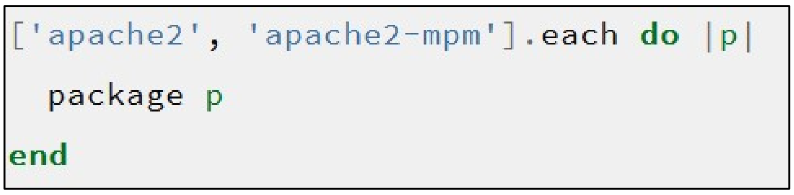
Figure 45 - Check platform condition (Chef Documentation, n.d.)

Above is a more standard condition, checking the node’s OS platform again.

“elsif” and “unless” keywords are also available for convenience to simplify the code when needed.

2.3.10. Loops

Loops are made possible using the keyword ".each".



```
['apache2', 'apache2-mpm'].each do |p|
  package p
end
```

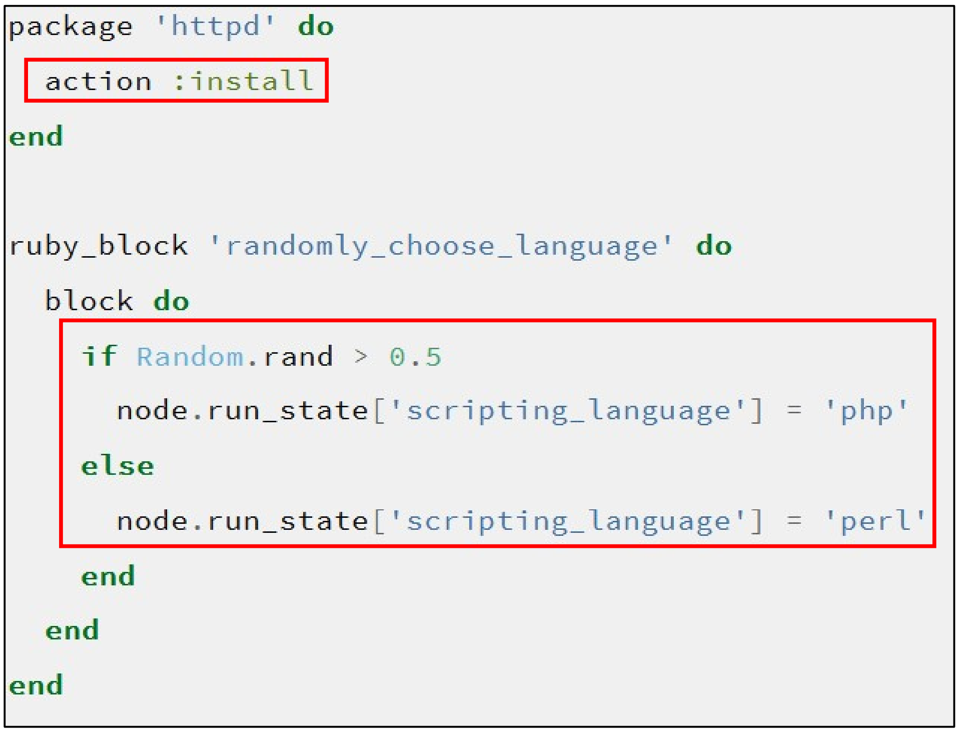
Figure 46 - Loop over array of package names (Chef Documentation, n.d.)

In the picture above, we are looping through a two-elements array, and we can execute code for each of those elements, named "p" for "package" in this example.

The keyword "for" does not exist in the native version of Chef.

2.3.11. Language

Chef uses a procedural language, which means that each step to achieve a goal must be specified.



```
package 'httpd' do
  action :install
end

ruby_block 'randomly_choose_language' do
  block do
    if Random.rand > 0.5
      node.run_state['scripting_language'] = 'php'
    else
      node.run_state['scripting_language'] = 'perl'
    end
  end
end
```

Figure 47 - Chef procedural language example (Chef Documentation, n.d.)

As we can see in the picture above, we notice the very structured syntax and pattern indicating a procedural language. Indeed, the instructions are very precise and complete to achieve a particular goal. For instance : We specify the package "httpd", then we have the action of installing that package. We also see if/else blocks containing detailed steps and instructions.

Chef scripts run using sequential execution on the agents by default.

2.3.12. Chef pros

- **Ruby DSL**

As Ruby is an actual programming language, more complex automation scenarios can be imagined and implemented more easily, which makes it superior to languages such as YAML and JSON.

- **Complete documentation**

The official Chef website offers a complete documentation on the software with good explanations and examples.

- **Chef Supermarket**

The Chef repository provides a great range of cookbooks that can be easily imported in a configuration.

2.3.13. Chef cons

- **Lengthy initial configuration**

The initial configuration of Chef takes time as we need to setup a main Chef server, workstations, and install clients on all managed nodes. The whole architecture must then be able to communicate, a goal achieved only after setting up certificates and trusted authorities. So, the configuration is not as easy as other agentless software may be. A VCS is also mandatory by default to upload files from a workstation to the main server.

- **Not very intuitive**

Chef is a software that requires time to learn, especially if users are not familiar with Ruby, which itself must be thoroughly learnt to be used with efficiency.

2.3.14. When to use Chef

Companies that already have operators knowing the Ruby language could use Chef without too many issues.

The procedural language makes it easier to use for programmers, but system administrators can use it as well if they do not mind the learning curve of Ruby.

The server/agents architecture allows a two-way communication, which means that agents will constantly check for configuration updates, the disadvantage being the reduction of flexibility if simple commands need to be executed on a small number of machines.

2.4. Puppet

2.4.1. Architecture

Puppet is an open-source configuration management software developed by the company of the same name and written in Ruby. It was initially released in 2005 and uses its own unique DSL, using the ".pp" extension for its configuration files. Puppet also provides a resource application programming interface (API) to write Ruby functions.

Puppet shares similarities with Chef. It also works with a server/agents architecture and is not agentless as well. The server is called the "master" or "Puppet master", an operator can use it to issue commands. Puppet does not require a workstation like Chef does. Many master servers may be configured in case the main master server goes down.

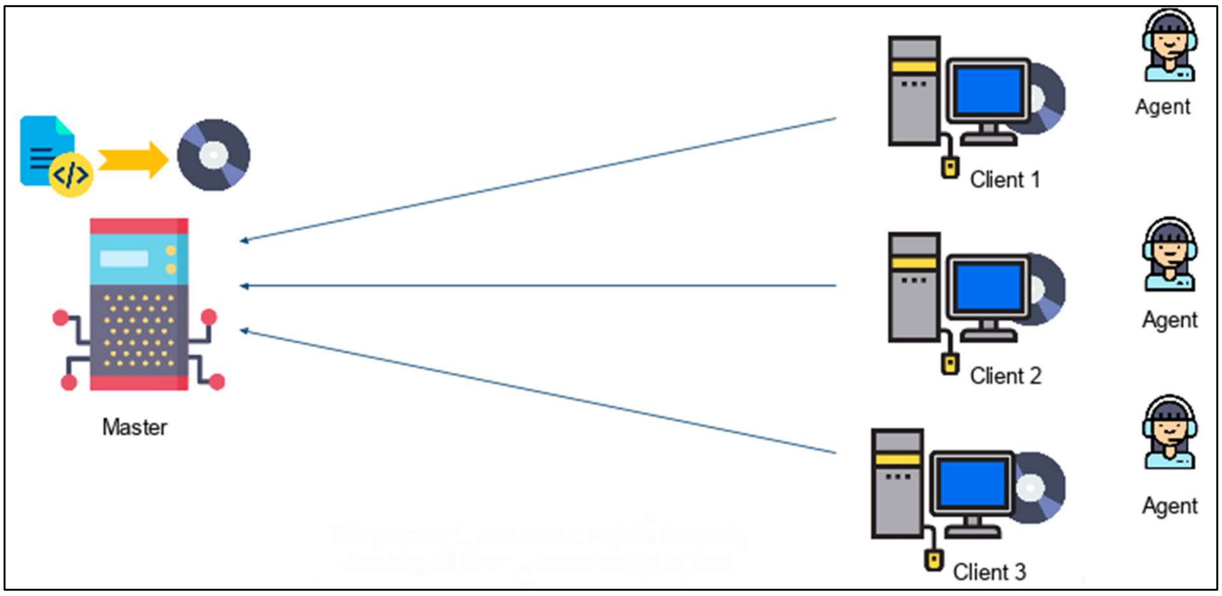


Figure 48 - Puppet architecture (Simplilearn, 2021)

The architecture works as follows : A client node (slave) sends "facts" (properties) about itself to the primary server (master), requesting a "catalog". The server then compiles and sends the node's catalog to the client. A catalog can be defined as the desired state for a node. Once the agent receives it, it compares the catalog to the node's state by checking each resource described and make necessary changes. Finally, the agent sends a report back to the server that can be read by an operator.

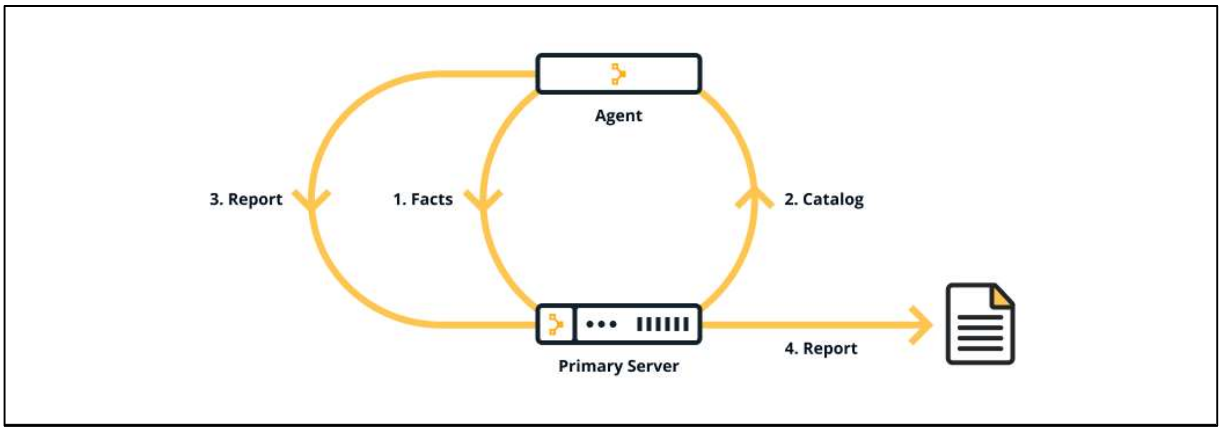
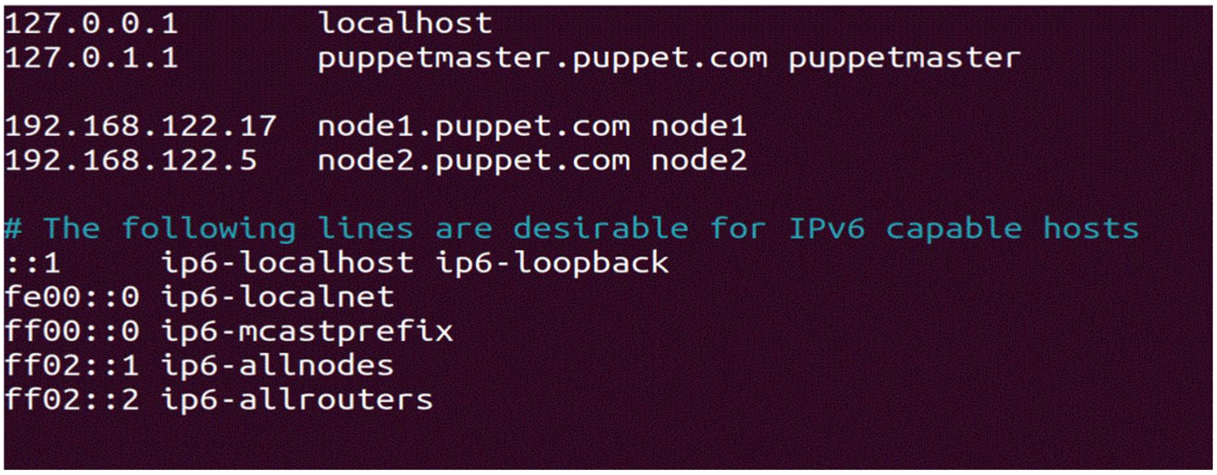


Figure 49 - Puppet server-agent process (Puppet Documentation, n.d.)

The professional Puppet version, named "Puppet Enterprise" is free to use to manage up to 10 nodes.

2.4.2. Installation

After a simple installation of the Puppet server on the master and Puppet clients on the nodes, all nodes must be authenticated by the master before exchanging configuration files. This can be done by first setting up the "/etc/hosts" file of all machines.

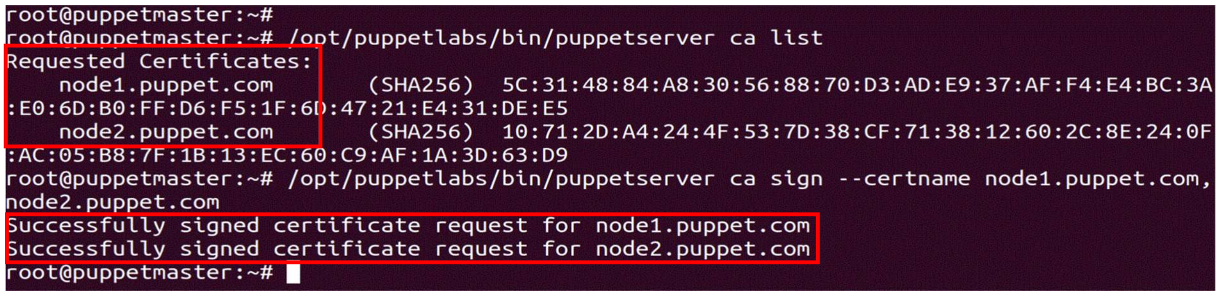


```
127.0.0.1       localhost
127.0.1.1       puppetmaster.puppet.com puppetmaster

192.168.122.17  node1.puppet.com node1
192.168.122.5   node2.puppet.com node2

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Figure 50 - /etc/hosts file example on a Puppet master

All nodes must also have an entry for the Puppet master in their own hosts file.

The nodes may then request a connexion to the Puppet master. They can automatically generate a new certificate that the master will have to sign using its fully qualified domain name (FQDN).



```
root@puppetmaster:~#
root@puppetmaster:~# /opt/puppetlabs/bin/puppetserver ca list
Requested Certificates:
    node1.puppet.com       (SHA256)  5C:31:48:84:A8:30:56:88:70:D3:AD:E9:37:AF:F4:E4:BC:3A
:E0:6D:B0:FF:D6:F5:1F:6D:47:21:E4:31:DE:E5
    node2.puppet.com       (SHA256)  10:71:2D:A4:24:4F:53:7D:38:CF:71:38:12:60:2C:8E:24:0F
:AC:05:B8:7F:1B:13:EC:60:C9:AF:1A:3D:63:D9
root@puppetmaster:~# /opt/puppetlabs/bin/puppetserver ca sign --certname node1.puppet.com,
node2.puppet.com
Successfully signed certificate request for node1.puppet.com
Successfully signed certificate request for node2.puppet.com
root@puppetmaster:~#
```

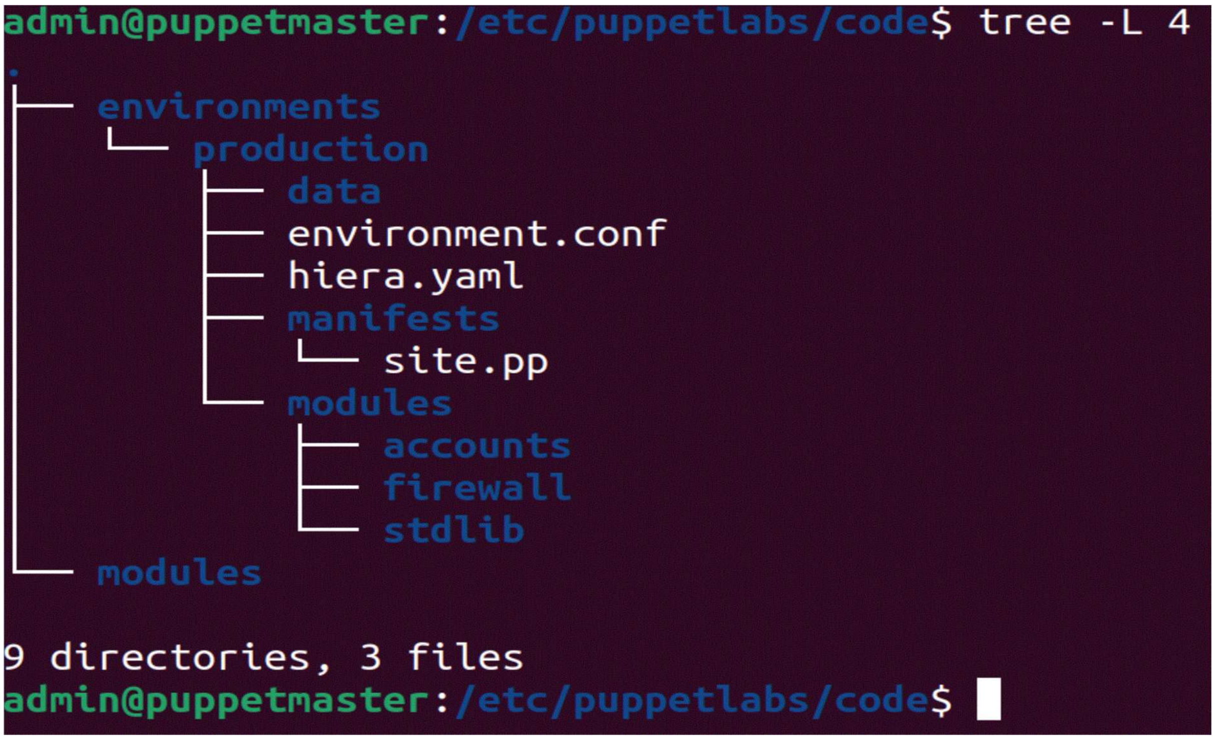
Figure 51 - Puppet master approving nodes certificates

Once these steps have been performed, configuration files can be written on the Puppet master and pulled by the nodes.

2.4.3. Modules

A module represents a specific series of tasks in a Puppet infrastructure. It can be compared to a role (Ansible) or cookbook (Chef). In the Puppet file structure, a module is a directory containing files, libraries, functions, tasks, etc.

Each file that contains code is called a “manifest” and ends with the “.pp” extension.



```
admin@puppetmaster:/etc/puppetlabs/code$ tree -L 4
.
├── environments
│   └── production
│       ├── data
│       ├── environment.conf
│       ├── hiera.yaml
│       ├── manifests
│       │   └── site.pp
│       └── modules
│           ├── accounts
│           ├── firewall
│           └── stdlib
└── modules

9 directories, 3 files
admin@puppetmaster:/etc/puppetlabs/code$
```

Figure 52 - Puppet code structure

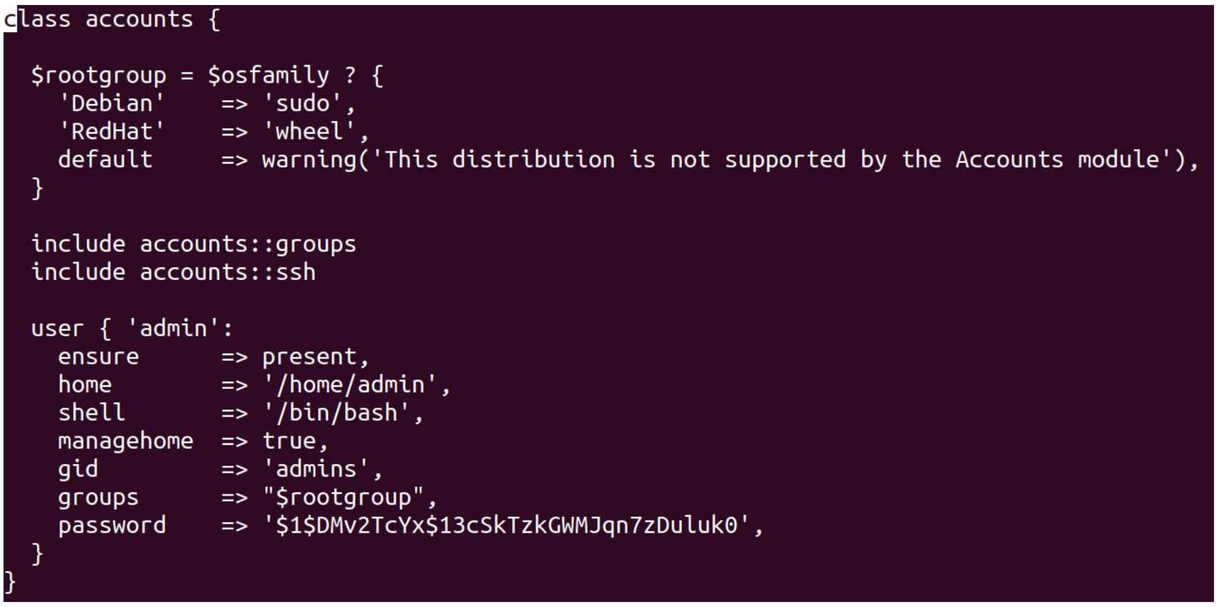
Above is the default code architecture. The main environment, “production”, contains three folders : “data”, “manifests” and “modules”. The first one contains various data files. The second directory usually contains the main manifest, also called the site manifest, which is the starting point in the making of the catalog. The third directory contains all modules, including manifest files for specific configuration tasks.

The “modules” directory outside of the “environments” folder was used in previous Puppet versions, there are no real reason to use it.

In the example above, three modules are present. The “stdlib” module is a default library provided by Puppet containing resources that can be automatically loaded by other modules. The other two modules are “accounts” and “firewall”. The first one has been created manually to set up a limited user and group on the Puppet master as well as on the nodes. The root user has also been disabled for security purposes. The second module was downloaded from the Puppet Forge, a repository of modules that can be compared to Ansible Galaxy or the Chef Supermarket. It is possible to download a module from the Puppet Forge using the following command :

puppet module install [module name]

As stated earlier, manifests are written in a specific DSL unique to Puppet.



```
class accounts {

  $rootgroup = $osfamily ? {
    'Debian'    => 'sudo',
    'RedHat'    => 'wheel',
    default     => warning('This distribution is not supported by the Accounts module'),
  }

  include accounts::groups
  include accounts::ssh

  user { 'admin':
    ensure      => present,
    home        => '/home/admin',
    shell       => '/bin/bash',
    managehome  => true,
    gid         => 'admins',
    groups      => "$rootgroup",
    password    => '$1$DMv2TcYx$13cSkTzkGWMJqn7zDuluk0',
  }
}
```

Figure 53 - init.pp file

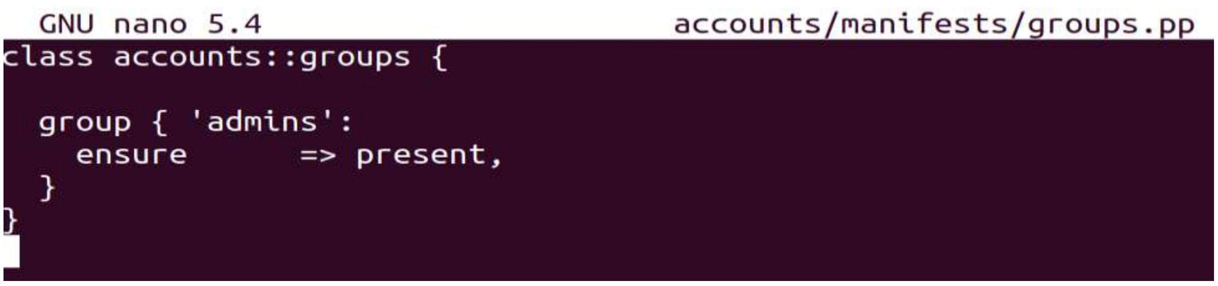
Above is an example of a file named “init.pp”, a manifest from the “accounts” module.

We define a class “accounts” containing user and group information.

We then declare a “$rootgroup” variable (starting with “$”). We want to create a new limited user with administrator privileges, but the group name might be different depending on the OS of a node, which is why we create a variable to check the OS family “fact” of each node. A condition is used to set either “sudo” or “wheel” as the superuser group.

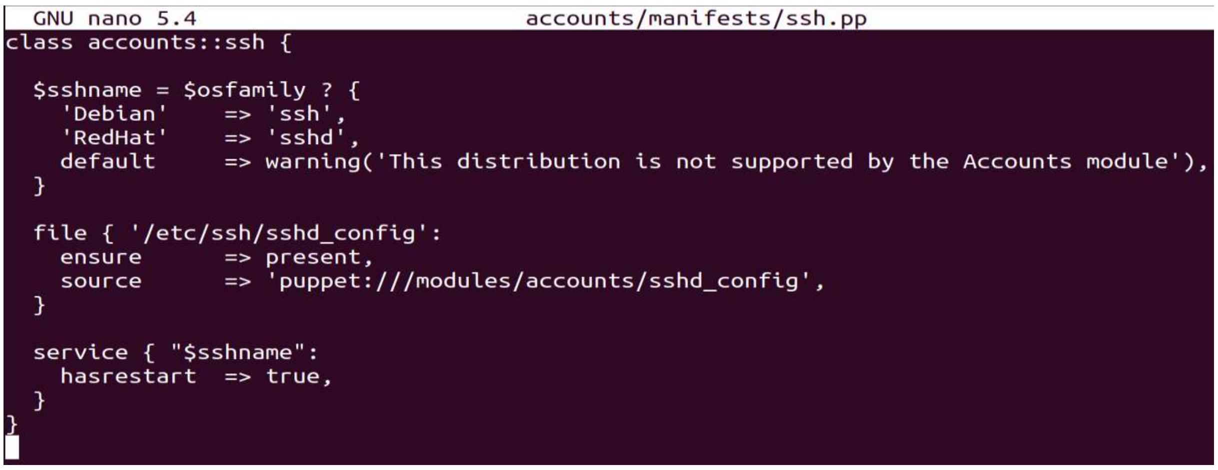
Note that the keywords “if” and “else”, “unless” and “case” can also be used to manifest conditions.

Next, two “include” keywords call other manifests ensuring the creation of the “admins” group as well as the installation of the OpenSSH service. The code from the two manifests below can then be executed before the rest of the code from the “accounts” class.



```
  GNU nano 5.4                    accounts/manifests/groups.pp
class accounts::groups {

  group { 'admins':
    ensure       => present,
  }
}
```

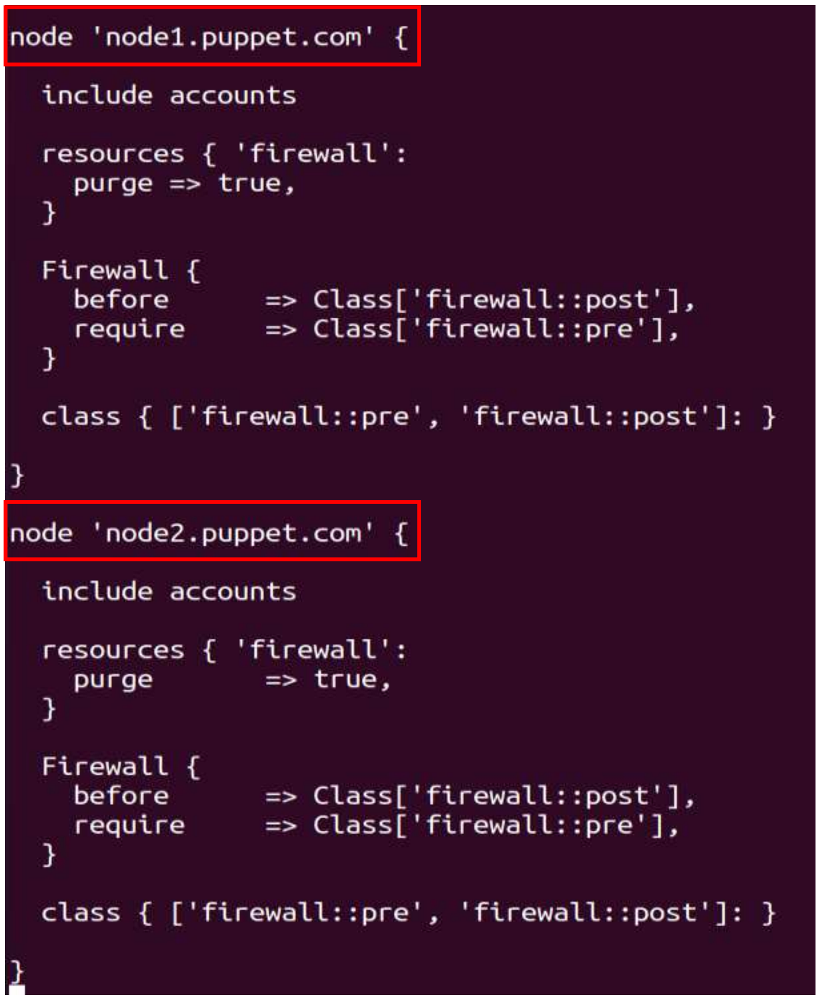
Figure 54 - groups.pp file



```
  GNU nano 5.4                              accounts/manifests/ssh.pp
class accounts::ssh {

  $sshname = $osfamily ? {
    'Debian'     => 'ssh',
    'RedHat'     => 'sshd',
    default      => warning('This distribution is not supported by the Accounts module'),
  }

  file { '/etc/ssh/sshd_config':
    ensure       => present,
    source       => 'puppet:///modules/accounts/sshd_config',
  }

  service { "$sshname":
    hasrestart   => true,
  }
}
```

Figure 55 - ssh.pp file

Finally, in the example of the “init.pp” file, the last part creates a user resource that will be created on the nodes, named “admin”.

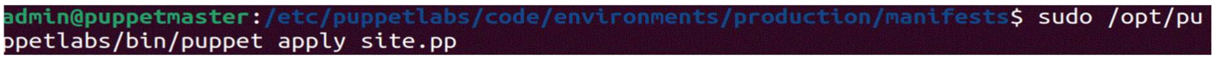


```
node 'node1.puppet.com' {

  include accounts

  resources { 'firewall':
    purge => true,
  }

  Firewall {
    before      => Class['firewall::post'],
    require     => Class['firewall::pre'],
  }

  class { ['firewall::pre', 'firewall::post']: }

}
node 'node2.puppet.com' {

  include accounts

  resources { 'firewall':
    purge       => true,
  }

  Firewall {
    before      => Class['firewall::post'],
    require     => Class['firewall::pre'],
  }

  class { ['firewall::pre', 'firewall::post']: }

}
```

Figure 56 - site.pp file sample

Once the different modules have been configured, the main or site manifest can be established. We can see both nodes specified above (“node1” and “node2”) as well as what modules need to be applied to them (“accounts” and “firewall”).

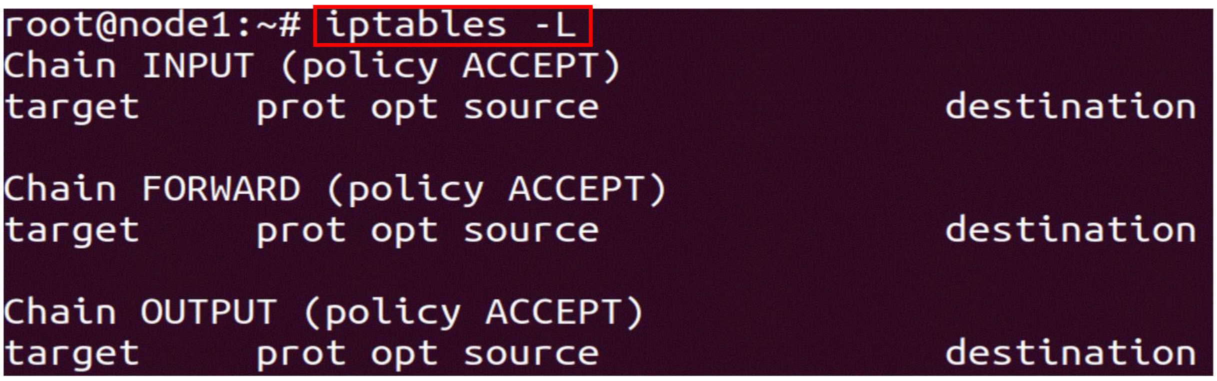
Using the “puppet apply” command, it is possible to apply a module on the local machine.



```
admin@puppetmaster:/etc/puppetlabs/code/environments/production/manifests$ sudo /opt/pu
ppetlabs/bin/puppet apply site.pp
```

Figure 57 - puppet apply command

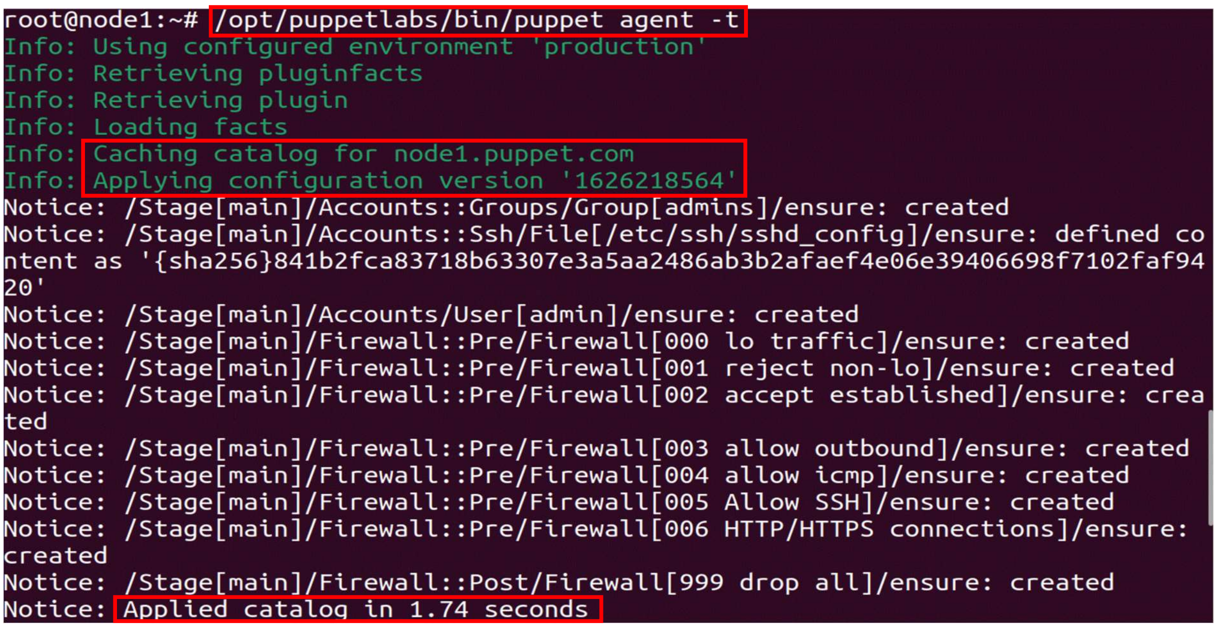
Agents can pull changes using the “puppet agent -t” command. Agents automatically pull updates every 30 minutes by default.



```
root@node1:~# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Figure 58 - iptables command before catalog update

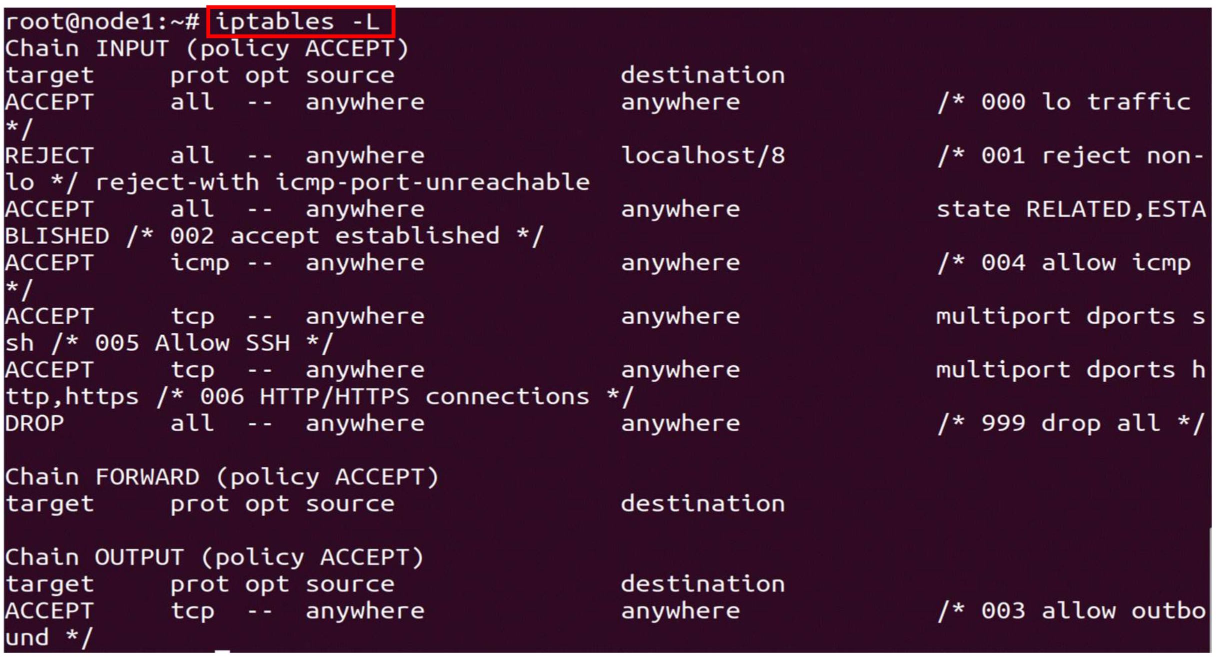
In the picture above, the “iptables -L” command is executed. It displays the firewall configuration on the node.



```
root@node1:~# /opt/puppetlabs/bin/puppet agent -t
Info: Using configured environment 'production'
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Loading facts
Info: Caching catalog for node1.puppet.com
Info: Applying configuration version '1626218564'
Notice: /Stage[main]/Accounts::Groups/Group[admins]/ensure: created
Notice: /Stage[main]/Accounts::Ssh/File[/etc/ssh/sshd_config]/ensure: defined co
ntent as '{sha256}841b2fca83718b63307e3a5aa2486ab3b2afaef4e06e39406698f7102faf94
20'
Notice: /Stage[main]/Accounts/User[admin]/ensure: created
Notice: /Stage[main]/Firewall::Pre/Firewall[000 lo traffic]/ensure: created
Notice: /Stage[main]/Firewall::Pre/Firewall[001 reject non-lo]/ensure: created
Notice: /Stage[main]/Firewall::Pre/Firewall[002 accept established]/ensure: crea
ted
Notice: /Stage[main]/Firewall::Pre/Firewall[003 allow outbound]/ensure: created
Notice: /Stage[main]/Firewall::Pre/Firewall[004 allow icmp]/ensure: created
Notice: /Stage[main]/Firewall::Pre/Firewall[005 Allow SSH]/ensure: created
Notice: /Stage[main]/Firewall::Pre/Firewall[006 HTTP/HTTPS connections]/ensure:
created
Notice: /Stage[main]/Firewall::Post/Firewall[999 drop all]/ensure: created
Notice: Applied catalog in 1.74 seconds
```

Figure 59 - Node update using Puppet

In the picture above, the catalog for the current node is being fetched from the Puppet master. The Puppet client then updates the node to the desired configuration from the “firewall” module we used.



```
root@node1:~# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere             /* 000 lo traffic
*/
REJECT     all  --  anywhere             localhost/8          /* 001 reject non-
lo */ reject-with icmp-port-unreachable
ACCEPT     all  --  anywhere             anywhere             state RELATED,ESTA
BLISHED /* 002 accept established */
ACCEPT     icmp --  anywhere             anywhere             /* 004 allow icmp
*/
ACCEPT     tcp  --  anywhere             anywhere             multiport dports s
sh /* 005 Allow SSH */
ACCEPT     tcp  --  anywhere             anywhere             multiport dports h
ttp,https /* 006 HTTP/HTTPS connections */
DROP       all  --  anywhere             anywhere             /* 999 drop all */

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             anywhere             /* 003 allow outbo
und */
```

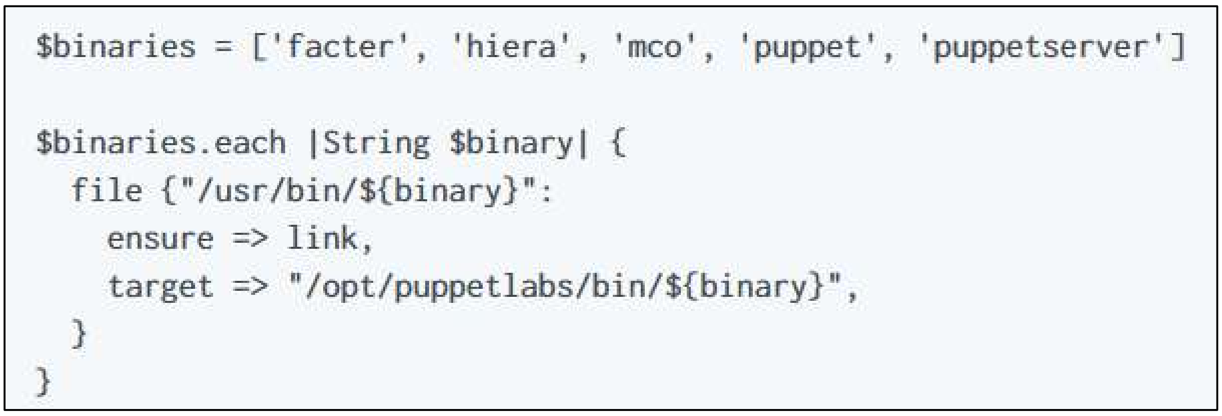
Figure 60 - iptables command after catalog update

If the “iptables -L” command is executed again after the change, we can see that the firewall configuration has been correctly updated. The new user “admin” has also been created and the root user has been disabled when connecting remotely to the nodes.

We will not go into more details here as the Puppet documentation is clear enough and this report is not meant to be an in-depth tutorial. This example simply shows the specific DSL used by Puppet with variables, conditions, and dependencies.

2.4.4. Loops

Puppet does not support native “for” loops. However, the “.each” keyword is available.



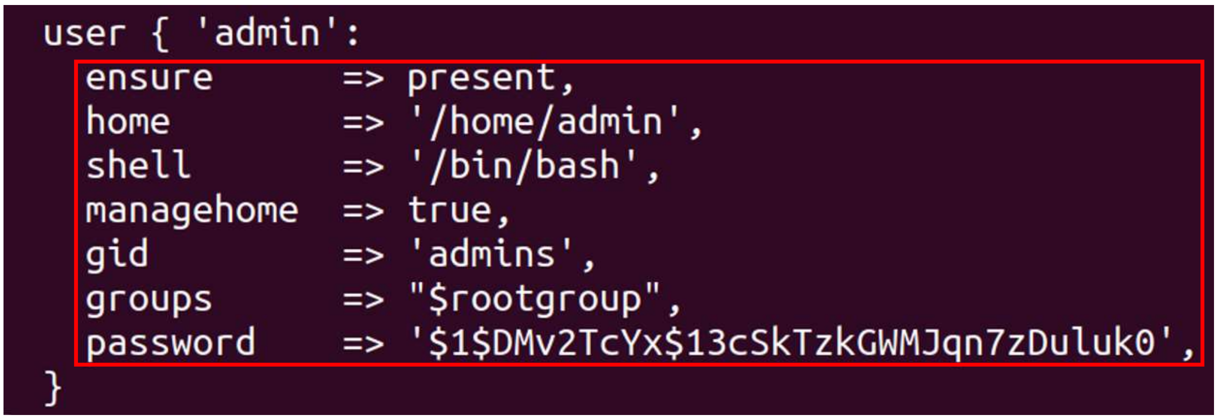
```
$binaries = ['facter', 'hiera', 'mco', 'puppet', 'puppetserver']

$binaries.each |String $binary| {
  file {"/usr/bin/${binary}":
    ensure => link,
    target => "/opt/puppetlabs/bin/${binary}",
  }
}
```

Figure 61 - Puppet loop example (Puppet Documentation, n.d.)

In the example above, symbolic links are created to reference different files or folders. A loop is set up around the “$binaries” array with the “.each” keyword. The “$binary” variable created can be used in the block to ensure the creation of the link for each string in the array.

2.4.5. Language

Puppet uses a declarative language, which means that only the desired state is specified, with no detailed steps to reach that goal.



```
user { 'admin':
  ensure      => present,
  home        => '/home/admin',
  shell       => '/bin/bash',
  managehome  => true,
  gid         => 'admins',
  groups      => "$rootgroup",
  password    => '$1$DMv2TcYx$13cSkTzkGWMJqn7zDuluk0',
}
```

Figure 62 - Puppet declarative language example

In the example code above, we can see a desired state, which is the creation of an "admin" user, and the corresponding restrictions.

Puppet scripts run using sequential execution on the agents by default.

2.4.6. Puppet pros

- **Puppet DSL**

The Puppet DSL combined with Ruby functions allows more complexity than languages such as YAML or JSON.

- **Complete documentation**

The official Puppet website offers a complete documentation on the software with good explanations and examples.

- **Puppet Forge**

The Puppet repository provides a great number of modules to download and import directly in a configuration.

2.4.7. Puppet cons

- **Lengthy initial configuration**

The initial configuration of Puppet takes time as we need to setup the Puppet master and install agents on the nodes. The Puppet master must then sign the nodes certificates to enable communication. The initial configuration is different from Chef, but like the latter, Puppet is not easy to set up.

- **Not very intuitive**

Puppet's architecture is pretty hard to assimilate. The configuration language is complex to learn, although Ruby is only mandatory to write more advanced functions.

2.4.8. When to use Puppet

Puppet uses a declarative configuration language, which improves scalability opportunities. Declarative language is generally used more by system administrators and less by programmers, the latter usually work with procedural language.

Puppet is a fair choice if operators are comfortable with declarative language and do not want to spend time learning a programming language. Ruby fundamentals are still necessary to use the software to its full potential as its DSL is less powerful without the Ruby functions API.

The initial configuration takes some time, but the server/agents architecture provides a convenient way to update managed nodes as they automatically pull updates from the catalog on start-up and every 30 minutes by default.

2.5. Salt

2.5.1. Architecture

SaltStack (or Salt) is an open-source configuration management software developed by the company of the same name in 2011. The virtualization technology company VMware bought Salt in October 2020 and provides full support for it since March 2021. The software is written in Python and uses the YAML language for its configuration files.

Salt works with a server/agent architecture. A main server, called "master", communicates with the managed nodes, named "minions".

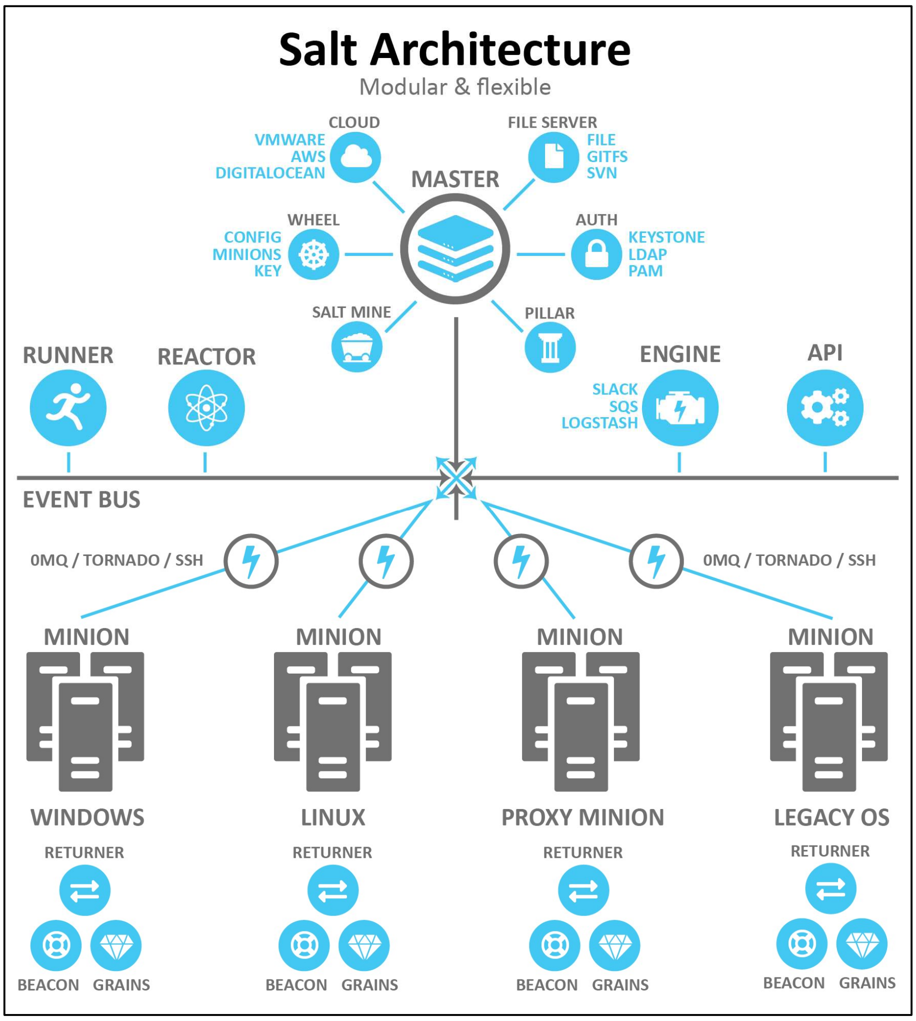


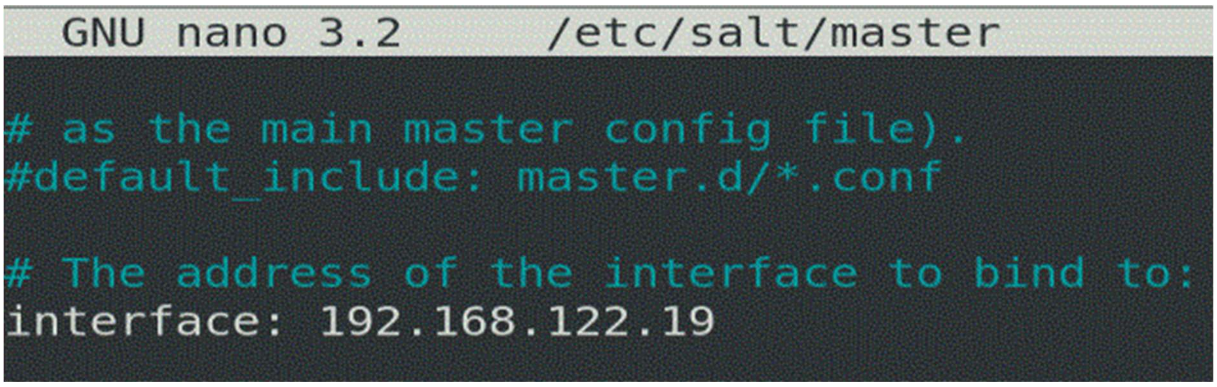
Figure 63 - Salt architecture (Salt system architecture, 2021)

Above is the official Salt architecture diagram taken from the VMware website. The Salt master can be installed on any Linux system, and the Salt minions may be either Windows, Linux or MacOS machines.

A node's property, such as the current operating system for instance, is called "grain". Grains can be used to define groups of Salt minions that need configuring. Minions can be configured with commands or states.

2.5.2. Installation

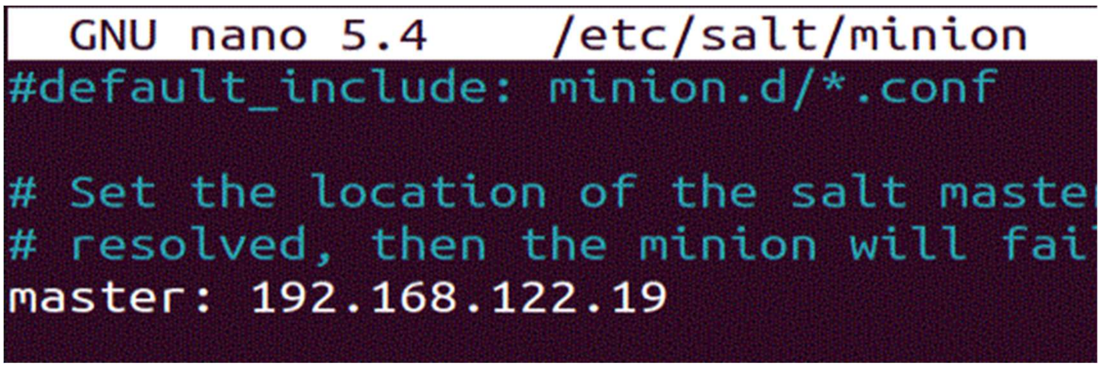
The installation of Salt is not as easy as with agentless software such as Ansible ; it shares more similarities with Chef and Puppet in that regard. The first step is to install the Salt-master and the Salt-minion packages on the machines and edit their hosts files to allow them to communicate using hostnames or FQDNs.



```
  GNU nano 3.2        /etc/salt/master

# as the main master config file).
#default_include: master.d/*.conf

# The address of the interface to bind to:
interface: 192.168.122.19
```

Figure 64 - Salt master file

Once the installation of the packages is complete, the master configuration file, located in "/etc/salt", needs to be edited to specify the Salt master's IP address near the top of the file. The salt-master service must be restarted.

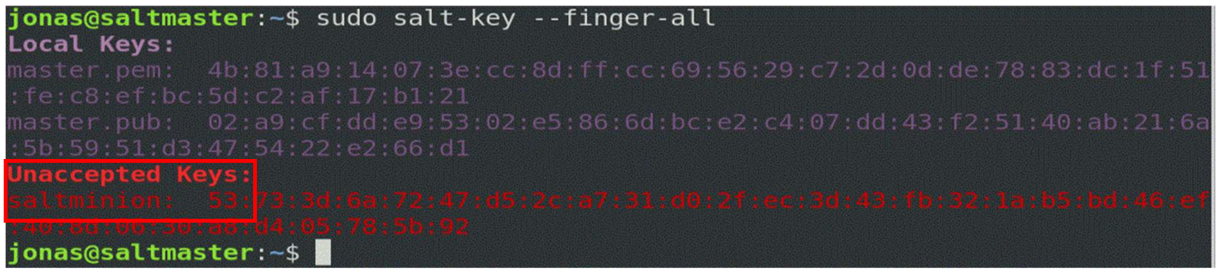


```
  GNU nano 5.4     /etc/salt/minion
#default_include: minion.d/*.conf

# Set the location of the salt maste
# resolved, then the minion will fai
master: 192.168.122.19
```

Figure 65 - Salt minion file (master property)

Respectively, the "/etc/salt/minion" configuration file must be edited on all minions to specify the Salt master's IP address. The salt-minion service must be restarted.

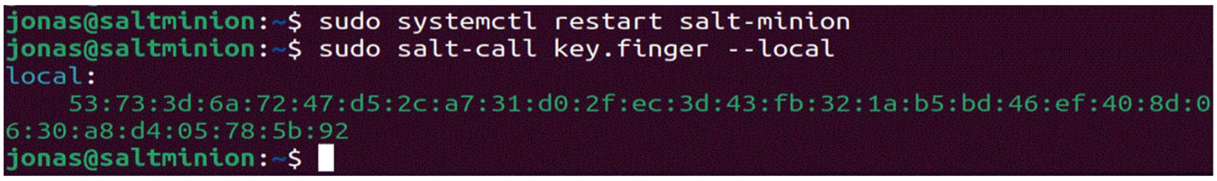
Once both services have been enabled with the correct configuration, it is possible to list the minions' key fingerprints from the Salt master.



```
jonas@saltmaster:~$ sudo salt-key --finger-all
Local Keys:
master.pem:  4b:81:a9:14:07:3e:cc:8d:ff:cc:69:56:29:c7:2d:0d:de:78:83:dc:1f:51
:fe:c8:ef:bc:5d:c2:af:17:b1:21
master.pub:  02:a9:cf:dd:e9:53:02:e5:86:6d:bc:e2:c4:07:dd:43:f2:51:40:ab:21:6a
:5b:59:51:d3:47:54:22:e2:66:d1
Unaccepted Keys:
saltminion:  53:73:3d:6a:72:47:d5:2c:a7:31:d0:2f:ec:3d:43:fb:32:1a:b5:bd:46:ef
:40:8d:00:30:a8:d4:05:78:5b:92
jonas@saltmaster:~$ 
```

Figure 66 - salt-key command with unaccepted key

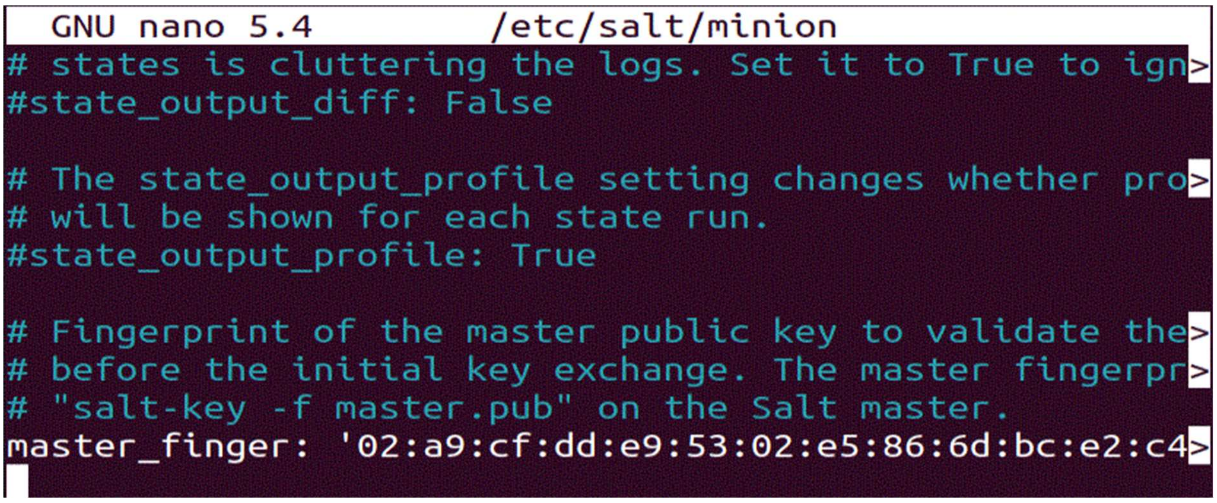
We can see in the picture above the local keys from the Salt master as well as the unaccepted key from the minion requesting to be linked.



```
jonas@saltminion:~$ sudo systemctl restart salt-minion
jonas@saltminion:~$ sudo salt-call key.finger --local
local:
    53:73:3d:6a:72:47:d5:2c:a7:31:d0:2f:ec:3d:43:fb:32:1a:b5:bd:46:ef:40:8d:0
6:30:a8:d4:05:78:5b:92
jonas@saltminion:~$ 
```

Figure 67 - salt-call command with minion key

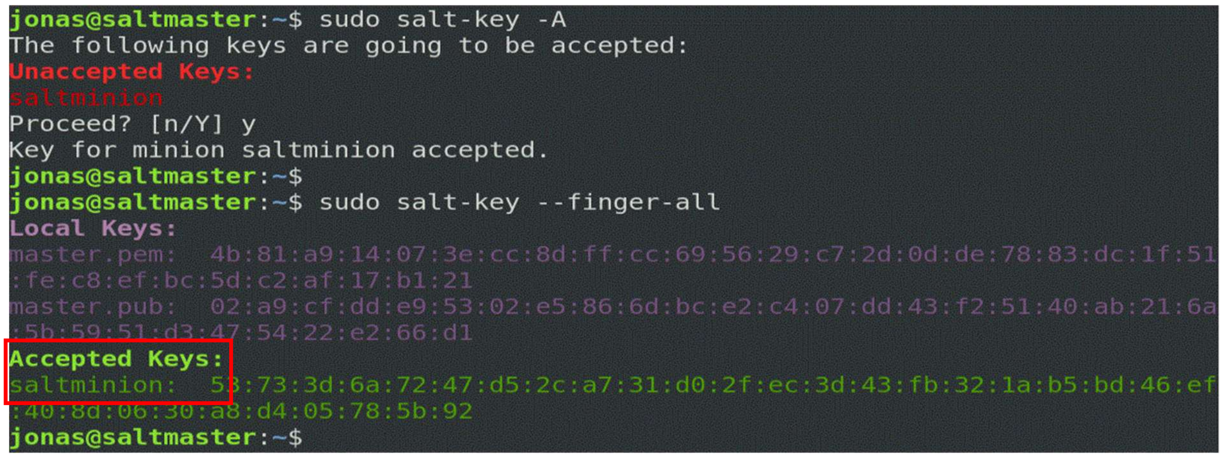
We can see that the key fingerprint of the minion matches with the one listed by the Salt master.



```
  GNU nano 5.4          /etc/salt/minion
# states is cluttering the logs. Set it to True to ign>
#state_output_diff: False

# The state_output_profile setting changes whether pro>
# will be shown for each state run.
#state_output_profile: True

# Fingerprint of the master public key to validate the>
# before the initial key exchange. The master fingerpr>
# "salt-key -f master.pub" on the Salt master.
master_finger: '02:a9:cf:dd:e9:53:02:e5:86:6d:bc:e2:c4>
```

Figure 68 - Salt minion file (master_finger property)

The “master.pub” key must then be reported in the minion file to be authenticated by the server.



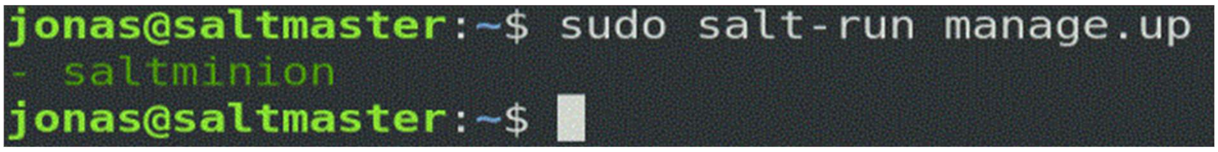
```
jonas@saltmaster:~$ sudo salt-key -A
The following keys are going to be accepted:
Unaccepted Keys:
saltminion
Proceed? [n/Y] y
Key for minion saltminion accepted.
jonas@saltmaster:~$
jonas@saltmaster:~$ sudo salt-key --finger-all
Local Keys:
master.pem:  4b:81:a9:14:07:3e:cc:8d:ff:cc:69:56:29:c7:2d:0d:de:78:83:dc:1f:51
:fe:c8:ef:bc:5d:c2:af:17:b1:21
master.pub:  02:a9:cf:dd:e9:53:02:e5:86:6d:bc:e2:c4:07:dd:43:f2:51:40:ab:21:6a
:5b:59:51:d3:47:54:22:e2:66:d1
Accepted Keys:
saltminion:  53:73:3d:6a:72:47:d5:2c:a7:31:d0:2f:ec:3d:43:fb:32:1a:b5:bd:46:ef
:40:8d:06:30:a8:d4:05:78:5b:92
jonas@saltmaster:~$
```

Figure 69 - Salt master validating the minion's key fingerprint

Finally, the key fingerprint can be accepted by the Salt master and the automated configuration of the minions is now possible.

2.5.3. Commands

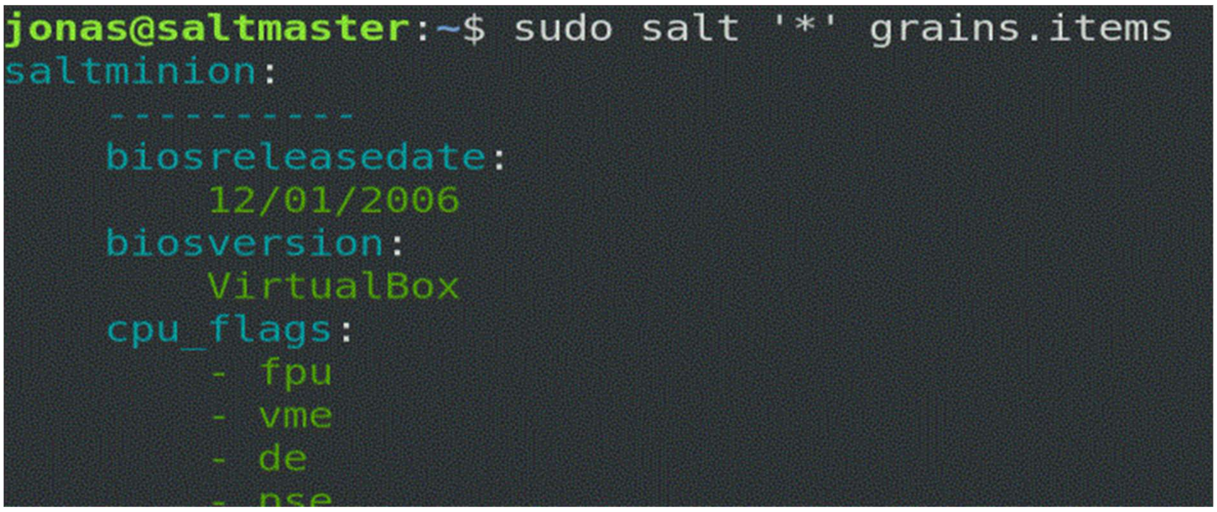
The Salt master may list all minions with the “salt-run manage.up” command :



```
jonas@saltmaster:~$ sudo salt-run manage.up
- saltminion
jonas@saltmaster:~$ 
```

Figure 70 - salt-run manage.up command

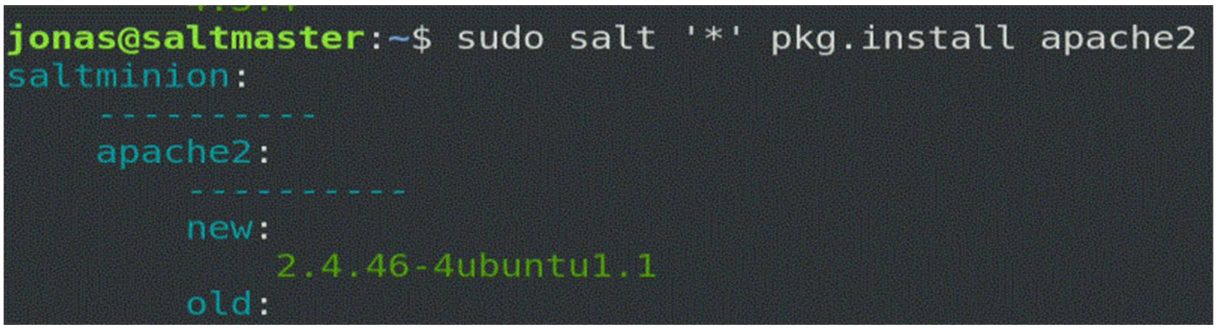
It is also possible to list a minion’s properties, called “grains” as stated before. Note that to run a command on all minions, an asterisk can be used between single quotes (‘*’). To run a command on a specific minion, the hostname must be specified between single quotes.



```
jonas@saltmaster:~$ sudo salt '*' grains.items
saltminion:
    ----------
    biosreleasedate:
        12/01/2006
    biosversion:
        VirtualBox
    cpu_flags:
        - fpu
        - vme
        - de
        - pse
```

Figure 71 - Minion's grains list

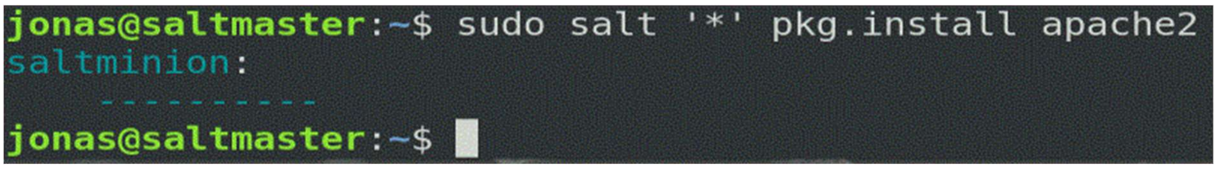
Using the “pkg.install” module, the Salt master automatically recognizes the operating system of its nodes and controls the distribution’s package manager. This means that the correct command is automatically used to install new packages on the managed nodes (apt, yum, etc.).



```
jonas@saltmaster:~$ sudo salt '*' pkg.install apache2
saltminion:
    ----------
    apache2:
        ----------
        new:
            2.4.46-4ubuntu1.1
        old:
```

Figure 72 - pkg.install command 1

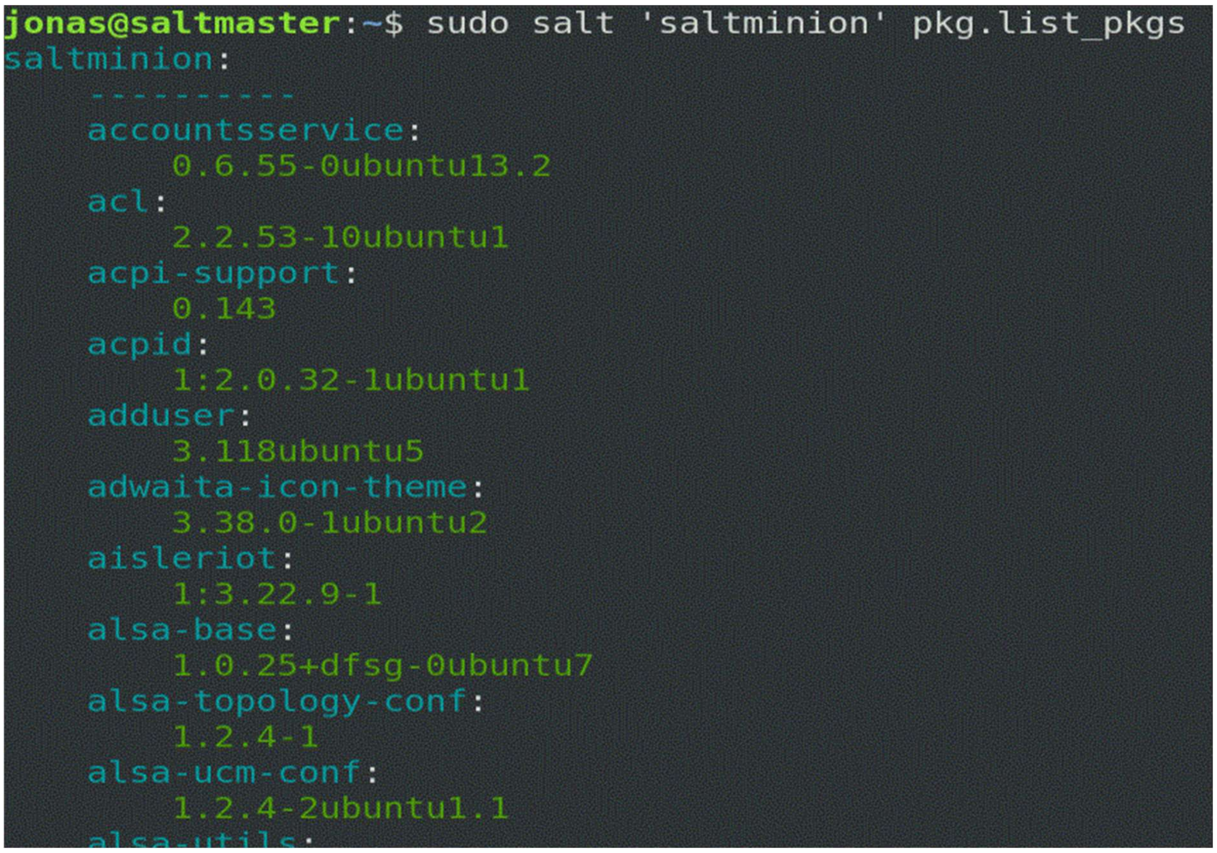
If the same command is run again, no changes are applied.



```
jonas@saltmaster:~$ sudo salt '*' pkg.install apache2
saltminion:
    ----------
jonas@saltmaster:~$ 
```

Figure 73 - pkg.install command 2

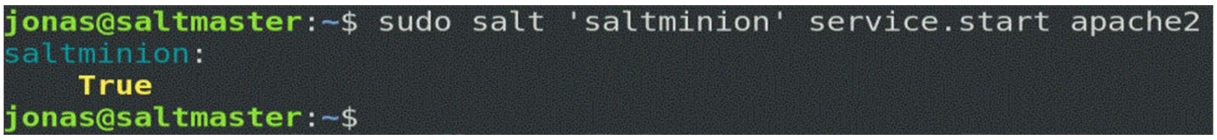
Just like the grains, we can also display the packages installed on the minions.



```
jonas@saltmaster:~$ sudo salt 'saltminion' pkg.list_pkgs
saltminion:
    ----------
    accountsservice:
        0.6.55-0ubuntu13.2
    acl:
        2.2.53-10ubuntu1
    acpi-support:
        0.143
    acpid:
        1:2.0.32-1ubuntu1
    adduser:
        3.118ubuntu5
    adwaita-icon-theme:
        3.38.0-1ubuntu2
    aisleriot:
        1:3.22.9-1
    alsa-base:
        1.0.25+dfsg-0ubuntu7
    alsa-topology-conf:
        1.2.4-1
    alsa-ucm-conf:
        1.2.4-2ubuntu1.1
    alsa-utils:
```

Figure 74 - Minion's packages list

The “service” command allows to manage a specific service.

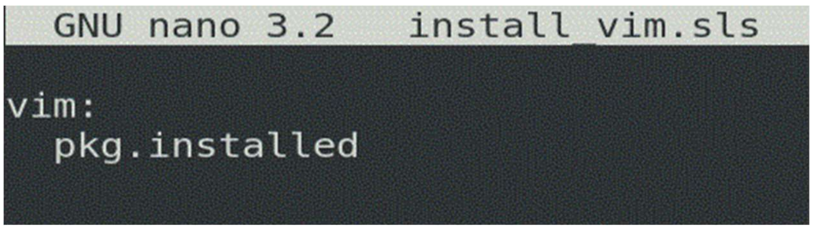


```
jonas@saltmaster:~$ sudo salt 'saltminion' service.start apache2
saltminion:
    True
jonas@saltmaster:~$
```

Figure 75 - service.start command

2.5.4. States

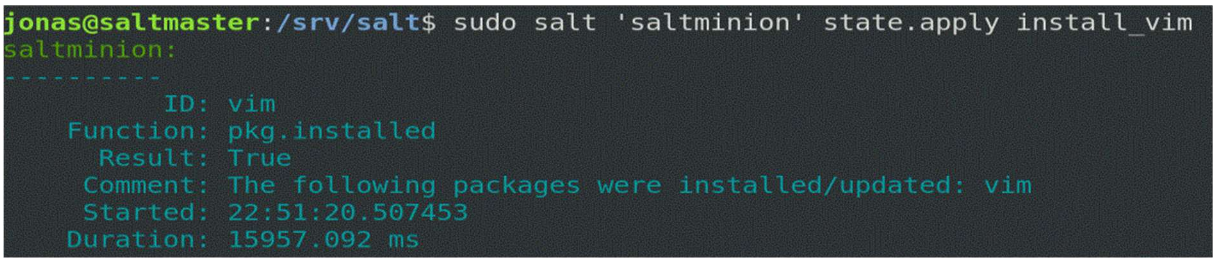
A Salt State corresponds to a file written in YAML describing a state we want a group of minions to be in. It can be compared to a playbook (Ansible), recipe (Chef) or manifest (Puppet). These files are usually placed in the “/srv/salt” directory.



```
  GNU nano 3.2    install_vim.sls

vim:
  pkg.installed
```

Figure 76 - install_vim.sls 1

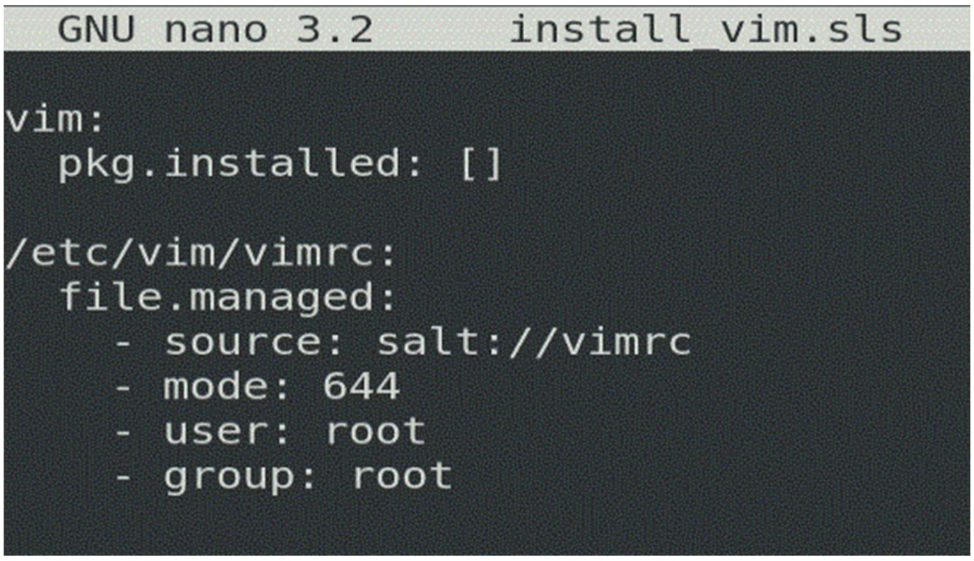
In the picture above, you can see a simple example of a State file, using the ".sls" extension (for "Salt State file") to be recognized by Salt. We indicate that we want the "vim" package to be installed on the minions affected by this script.



```
jonas@saltmaster:/srv/salt$ sudo salt 'saltminion' state.apply install_vim
saltminion:
----------
          ID: vim
    Function: pkg.installed
      Result: True
     Comment: The following packages were installed/updated: vim
     Started: 22:51:20.507453
    Duration: 15957.092 ms
```

Figure 77 - state.apply command

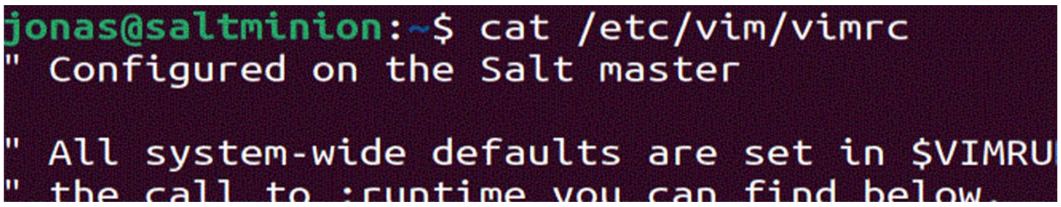
Any Salt State can then be applied to a group of minions. Note that the ".sls" extension must not be specified in the command ("install_vim" and not "install_vim.sls").



```
  GNU nano 3.2      install_vim.sls

vim:
  pkg.installed: []

/etc/vim/vimrc:
  file.managed:
    - source: salt://vimrc
    - mode: 644
    - user: root
    - group: root
```

Figure 78 - install_vim.sls 2

Configuration files may also be copied to the minions. In the picture above, we specify that we want a file named "vimrc" (located in "/etc/vim") created from the existing file on the server ("/srv/salt" on the Salt master) and copied to the minion to use the same configuration.



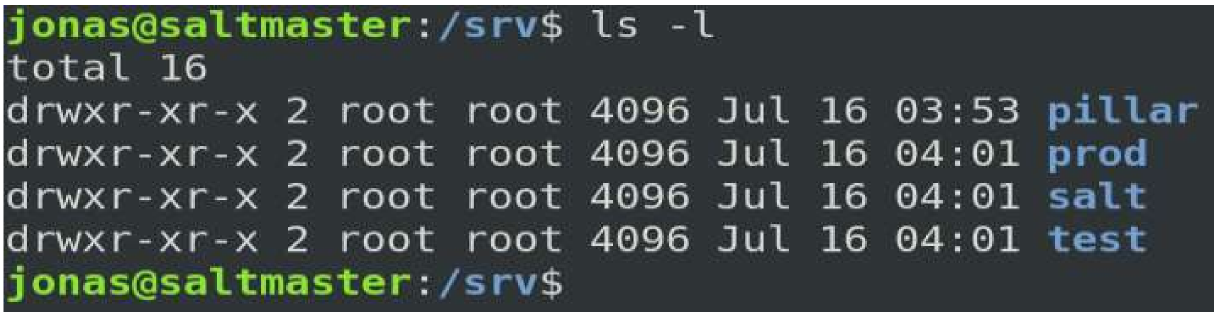
```
jonas@saltminion:~$ cat /etc/vim/vimrc
" Configured on the Salt master

" All system-wide defaults are set in $VIMRU
" the call to :runtime you can find below.
```

Figure 79 - vimrc configuration file copied to the minion

If we run the script again, the file is created on the minion and replaces any file with the same name if it exists.

2.5.5. Environments

An environment refers to a top-level Salt directory. However, there can be more than one environment.



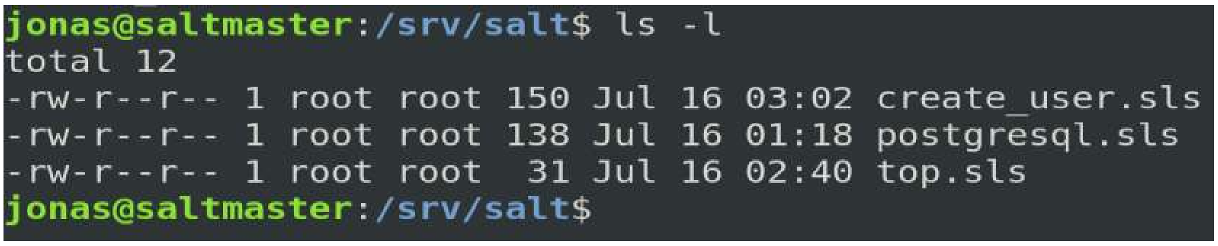
```
jonas@saltmaster:/srv$ ls -l
total 16
drwxr-xr-x 2 root root 4096 Jul 16 03:53 pillar
drwxr-xr-x 2 root root 4096 Jul 16 04:01 prod
drwxr-xr-x 2 root root 4096 Jul 16 04:01 salt
drwxr-xr-x 2 root root 4096 Jul 16 04:01 test
jonas@saltmaster:/srv$
```

Figure 80 - /srv structure example 1

For instance, we can create an environment for production and another one for testing purposes. Each environment is isolated unless explicit references are used.

2.5.6. Top files

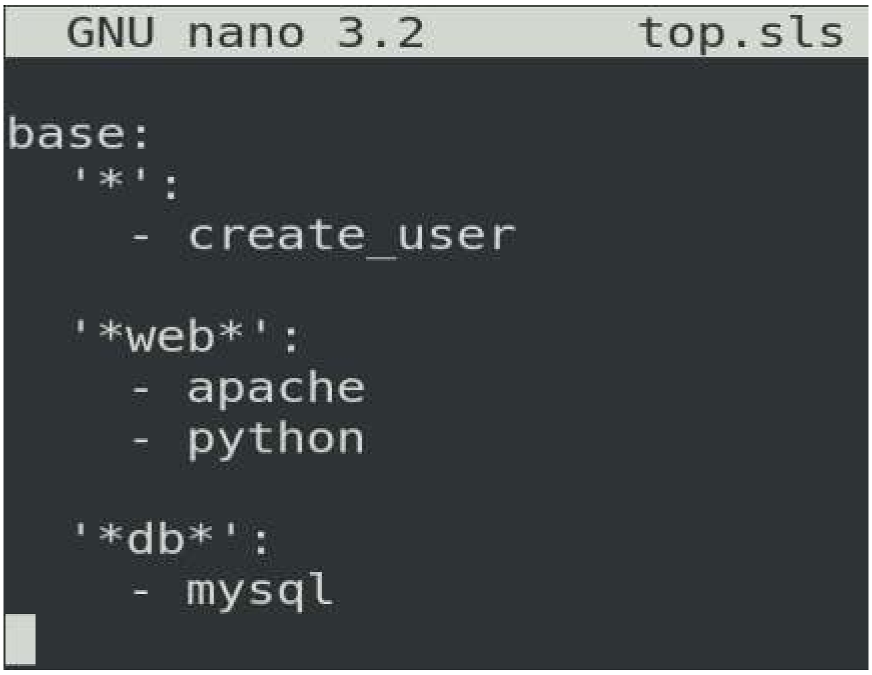
A top file is a ".sls" file that contains mapping information between groups of minions and which tasks should be run on them.



```
jonas@saltmaster:/srv/salt$ ls -l
total 12
-rw-r--r-- 1 root root 150 Jul 16 03:02 create_user.sls
-rw-r--r-- 1 root root 138 Jul 16 01:18 postgresql.sls
-rw-r--r-- 1 root root  31 Jul 16 02:40 top.sls
jonas@saltmaster:/srv/salt$
```

Figure 81 - /srv/salt structure example

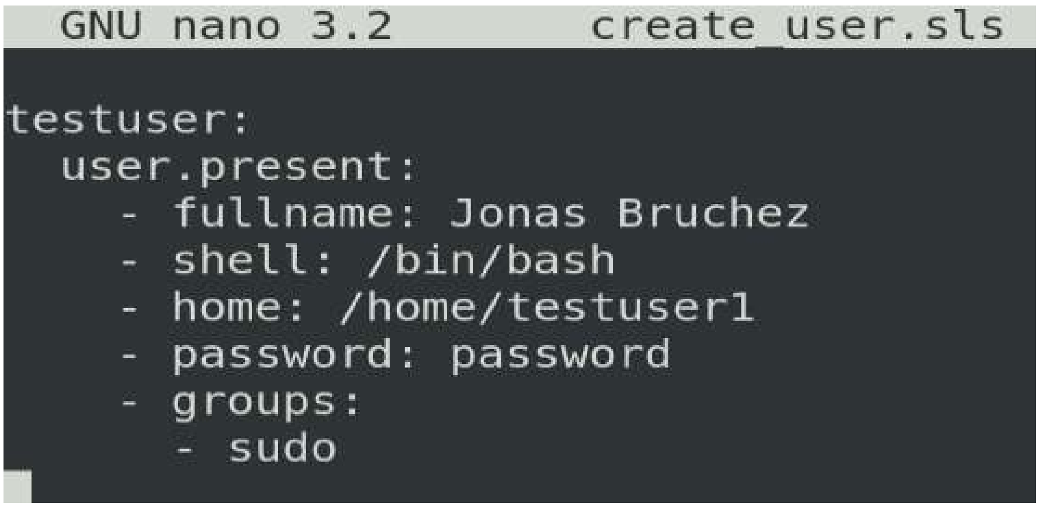
There can only be one top file per directory. We have one in the example above in the default "salt" environment. A top file always starts with "base:" and is present in the top directory of an environment.



```
  GNU nano 3.2          top.sls

base:
  '*':
    - create_user

  '*web*':
    - apache
    - python

  '*db*':
    - mysql
```

Figure 82 - top.sls file example

In this example, we created a simple top file with three groups : The asterisk symbol means that all minions will see the "create_user" State run on them, "*web*" means that all minions containing "web" in their FQDN will be impacted, and the same principle applies to the "*db*" group.



```
  GNU nano 3.2          create_user.sls

testuser:
  user.present:
    - fullname: Jonas Bruchez
    - shell: /bin/bash
    - home: /home/testuser1
    - password: password
    - groups:
      - sudo
```

Figure 83 - create_user.sls file

Above is the "create_user.sls" file, which creates a simple user named "testuser".

Next, we can simply type the following command :

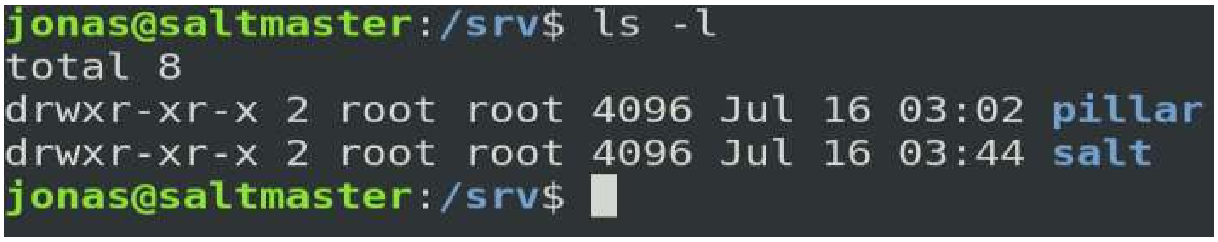
salt '*' state.apply

There is no need to specify which file to run, as Salt knows to look by default into the top file of the environment. Our user is then created on all nodes.

2.5.7. Pillars

A pillar is a file made to contain sensitive information that would not be secure in a standard State file. Pillar information is transmitted only to minions that need it, hiding it from other minions even if they are affected by the same configuration State, as only the reference to a pillar would be visible in it.

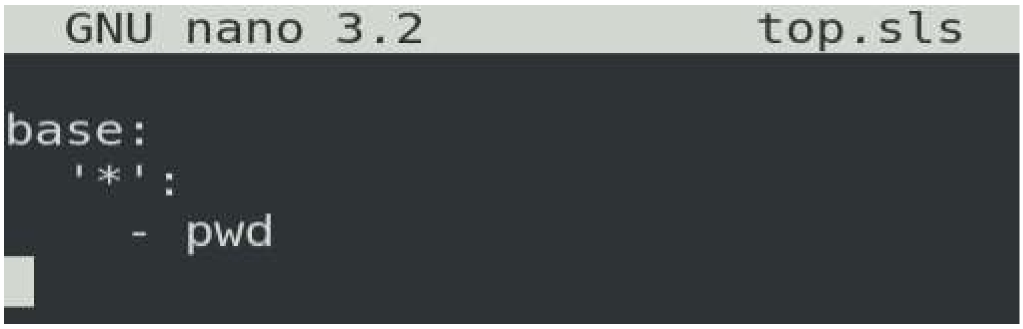
As you may have noticed in the file "create_user.sls", we wrote the password in plain text. We can use pillars to fix that.



```
jonas@saltmaster:/srv$ ls -l
total 8
drwxr-xr-x 2 root root 4096 Jul 16 03:02 pillar
drwxr-xr-x 2 root root 4096 Jul 16 03:44 salt
jonas@saltmaster:/srv$
```

Figure 84 - /srv structure example 2

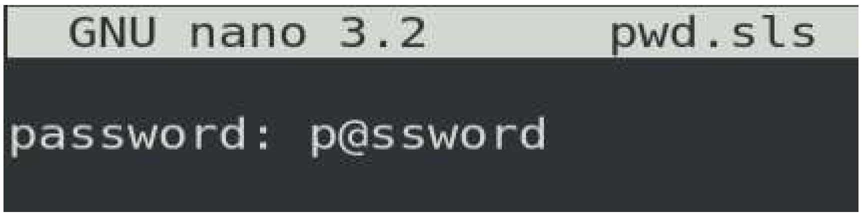
We need to create a new folder that needs to be named "pillar" by default. Inside, we create two files, named "top.sls" and "pwd.sls".



```
  GNU nano 3.2                 top.sls

base:
  '*':
    - pwd
```

Figure 85 - top.sls file example in /srv/pillar

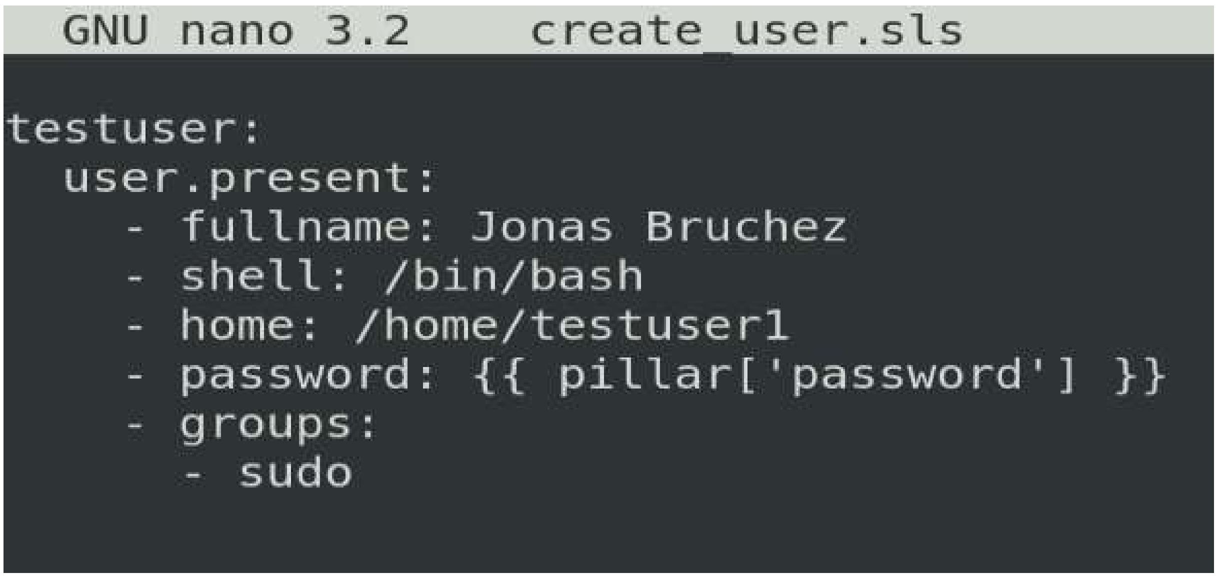
The top file contains the "base" keyword, an asterisk indicating that all minions will save this pillar information in this example, and the script containing the information we want.



```
  GNU nano 3.2           pwd.sls

password: p@ssword
```

Figure 86 - pwd.sls file

In "pwd.sls", we have our variable named "password".



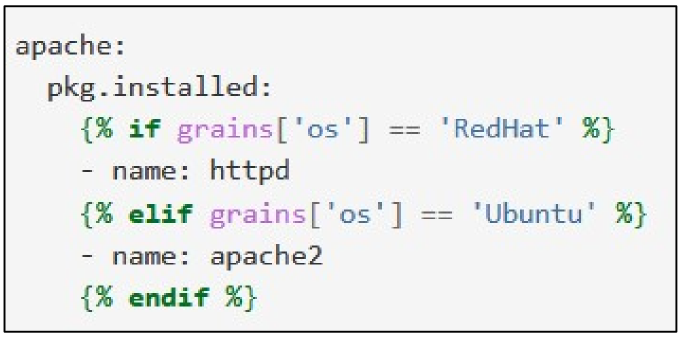
```
  GNU nano 3.2      create_user.sls

testuser:
  user.present:
    - fullname: Jonas Bruchez
    - shell: /bin/bash
    - home: /home/testuser1
    - password: {{ pillar['password'] }}
    - groups:
      - sudo
```

Figure 87 - create_user.sls file with pillar

We can edit our “create_user.sls” file to hide the password and use the variable we set in the pillar. The variable will be sent only to minions that need it.

2.5.8. Conditionals

Conditions can be used in Salt States using the “if”, “else”, “elif” and “endif” statements.



```
apache:
  pkg.installed:
    {% if grains['os'] == 'RedHat' %}
    - name: httpd
    {% elif grains['os'] == 'Ubuntu' %}
    - name: apache2
    {% endif %}
```

Figure 88 - Salt condition example (SaltStack, n.d.)

In this example from the official Salt documentation, a minion’s OS is being checked to ensure that the correct package name is used for the installation.

2.5.9. Loops

Salt State files support loops, using the “for”, “in” and “endfor” statements.

```
{% for usr in ['moe','larry','curly'] %}
{{ usr }}:
  user.present
{% endfor %}
```

Figure 89 - Salt loop example (SaltStack, n.d.)

In the example above, the code runs through an array of three strings and stores them temporally in the "usr" variable to check if that string is an existing username.

2.5.10. Formulas

Formulas are pre-written Salt States. They can be used for many different tasks and any of them can be downloaded from the official SaltStack GitHub account. It is also possible to create custom formulas using the provided template.

2.5.11. Language

Salt uses a declarative language, which means that only the desired state is specified, with no detailed steps to reach that goal.

```
apache:
  pkg.installed: []
  service.running:
    - watch:
      - pkg: apache
      - file: /etc/httpd/conf/httpd.conf
      - user: apache
  user.present:
    - uid: 87
    - gid: 87
    - home: /var/www/html
    - shell: /bin/nologin
    - require:
      - group: apache
  group.present:
    - gid: 87
    - require:
      - pkg: apache
```

Figure 90 - Salt declarative language example (SaltStack, n.d.)

In the example above, we clearly see the same declarative pattern used with Puppet. The only difference being the fact that Salt uses the YAML declarative language like Ansible.

Salt scripts run using parallel execution on the agents by default.

2.5.12. Salt pros

- **Simple management**

The initial configuration is not too complicated, and the infrastructure is clear.

- **Configuration language**

YAML is a language easy to understand and learn, even for non-programmers.

2.5.13. Salt cons

- **YAML configuration files**

Like Ansible, the complexity of YAML is limited compared to programming languages such as Ruby.

- **Lacking documentation**

The Salt documentation is sadly less organized and sometimes lacking compared to other documentations. This might be because of the recent takeover of Salt by VMware ; we find documentation about the software on their website as well as on the Salt Project website which can make it confusing.

- **Less popular than other software**

A problem with Salt is that there is less information available on the Internet as the community is significantly smaller.

2.5.14. When to use Salt

Salt was released fairly recently compared to other automation software and chose a more flexible and simple approach that is easy to understand.

Salt uses YAML, which is a language that system administrators tend to use more rather than standard programming languages such as Ruby.

2.6. Terraform

2.6.1. Architecture

Terraform is an open-source infrastructure provisioning software developed by the company HashiCorp in 2014. The software is written in the Go programming language. It uses the HashiCorp Configuration Language (HCL) with the ".tf" extension for its configuration files. That language is inspired by Nginx (web server) configuration files.

Terraform is different from the four previous software described. It is mainly an infrastructure provisioning software, rather than a configuration management software. This implies that the use of Terraform is focused on the automation of the provisioning of the infrastructure, rather than on its configuration. For instance, it can be convenient for users that are looking to prepare their infrastructure so they can then deploy their application on it. Terraform is considered to be an IaC tool for Cloud environments.

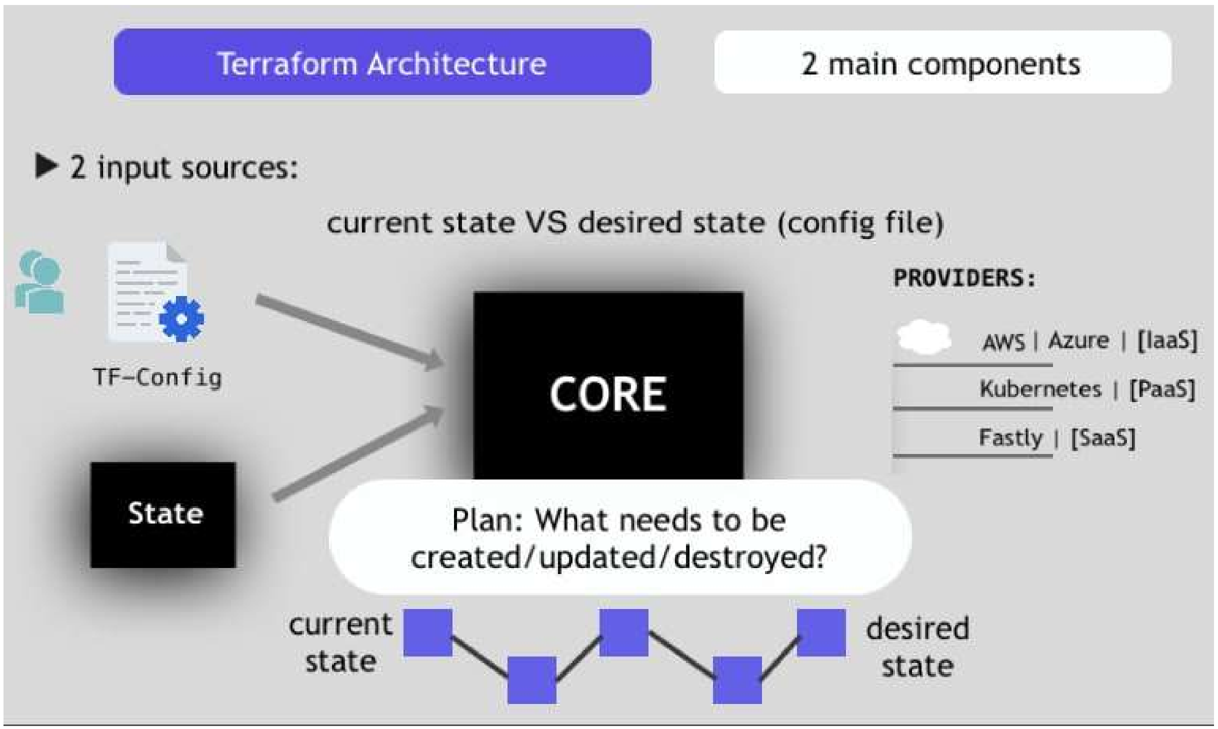


Figure 91 - Terraform architecture (Janashia, 2020)

The architecture of Terraform works as follows : Its core is a statically-compiled binary written in Go, this means that the build has a predictable behaviour as all bindings are performed at compile time (and not at running time). It is completely agentless.

The core needs two components as input to create a provisioning plan : the **Terraform configuration** and the **state**. The Terraform configuration refers to the HCL files written by a user specifying what needs to be created and provisioned, and what the final state should be. The state refers to the current state of the infrastructure, if any ; this is needed for Terraform to update an instance of the infrastructure when needed, however the instance is usually recreated as the software chooses an immutable approach. The state file is also useful to store bindings and dependencies between remote objects and the infrastructure using them.

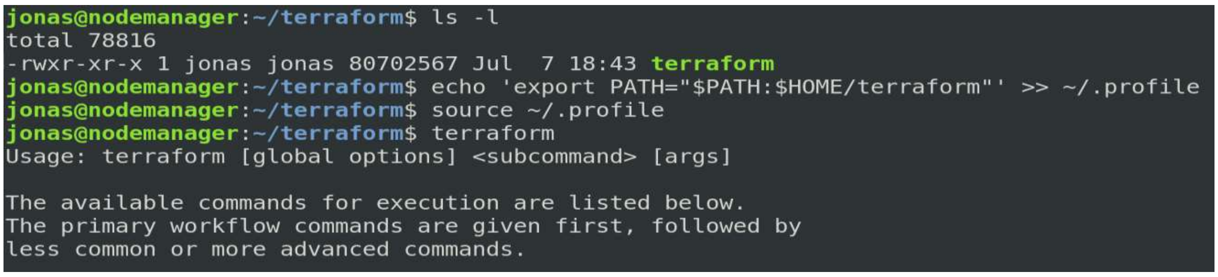
With the configuration and state files, Terraform computes the most optimal way to obtain the desired environment state described. To reach that goal, **providers** are used to implement resources that Terraform can manage. Providers are distributed separately and are independent from Terraform.

Providers and pre-written reusable configuration (modules) can be chosen from the Terraform online repository, called “Terraform Registry”.

2.6.2. Installation

As stated above, Terraform is agentless, which makes its installation fast and easy. The software can be downloaded from the official Terraform website. It may be installed on a Windows, macOS or Linux machine.

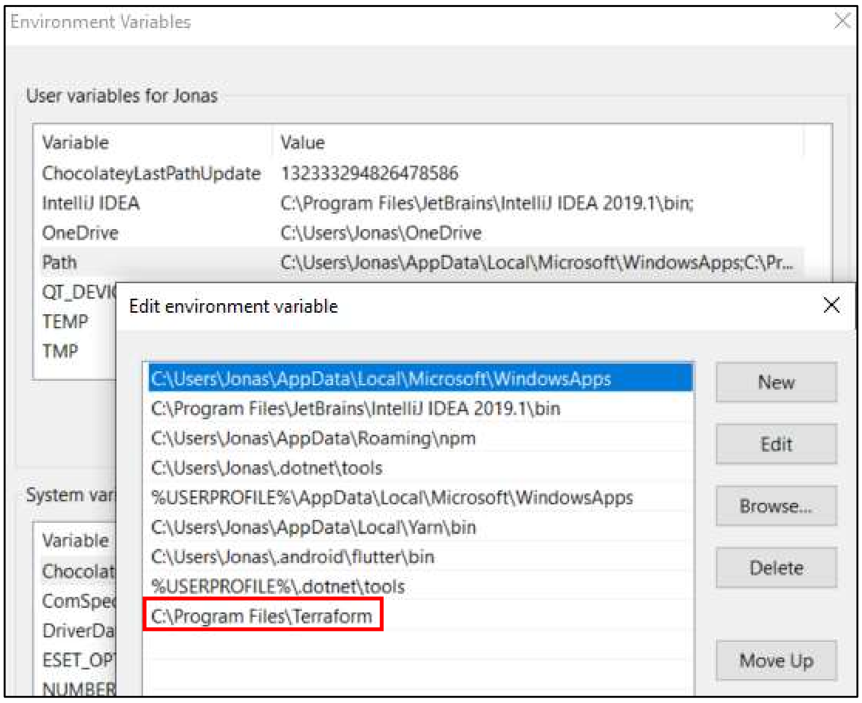
Once downloaded, the file must be unzipped. It contains an executable that can be run as a command on Linux systems.



```
jonas@nodemanager:~/terraform$ ls -l
total 78816
-rwxr-xr-x 1 jonas jonas 80702567 Jul  7 18:43 terraform
jonas@nodemanager:~/terraform$ echo 'export PATH="$PATH:$HOME/terraform"' >> ~/.profile
jonas@nodemanager:~/terraform$ source ~/.profile
jonas@nodemanager:~/terraform$ terraform
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.
```

Figure 92 - terraform command on Linux

On Windows, once added to the path, Terraform commands are recognized and can be input in the command line or PowerShell.


Environment Variables

User variables for Jonas

Variable	Value
ChocolateyLastPathUpdate	132333294826478586
IntelliJ IDEA	C:\Program Files\JetBrains\IntelliJ IDEA 2019.1\bin;
OneDrive	C:\Users\Jonas\OneDrive
Path	C:\Users\Jonas\AppData\Local\Microsoft\WindowsApps;C:\Pr...
QT_DEVI	
TEMP	
TMP	

System var

Variable
Chocolat
ComSpec
DriverDa
ESET_OP
NUMBER

Edit environment variable

```
C:\Users\Jonas\AppData\Local\Microsoft\WindowsApps
C:\Program Files\JetBrains\IntelliJ IDEA 2019.1\bin
C:\Users\Jonas\AppData\Roaming\npm
C:\Users\Jonas\.dotnet\tools
%USERPROFILE%\AppData\Local\Microsoft\WindowsApps
C:\Users\Jonas\AppData\Local\Yarn\bin
C:\Users\Jonas\.android\flutter\bin
%USERPROFILE%\.dotnet\tools
C:\Program Files\Terraform
```

Figure 93 - Terraform executable file location added to the path

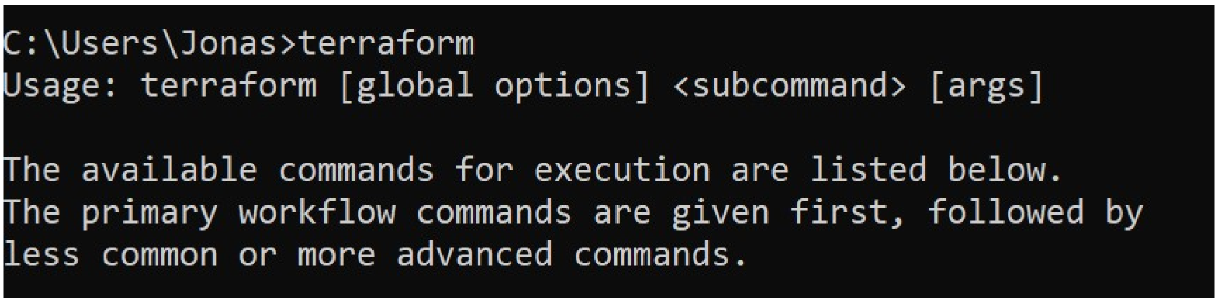
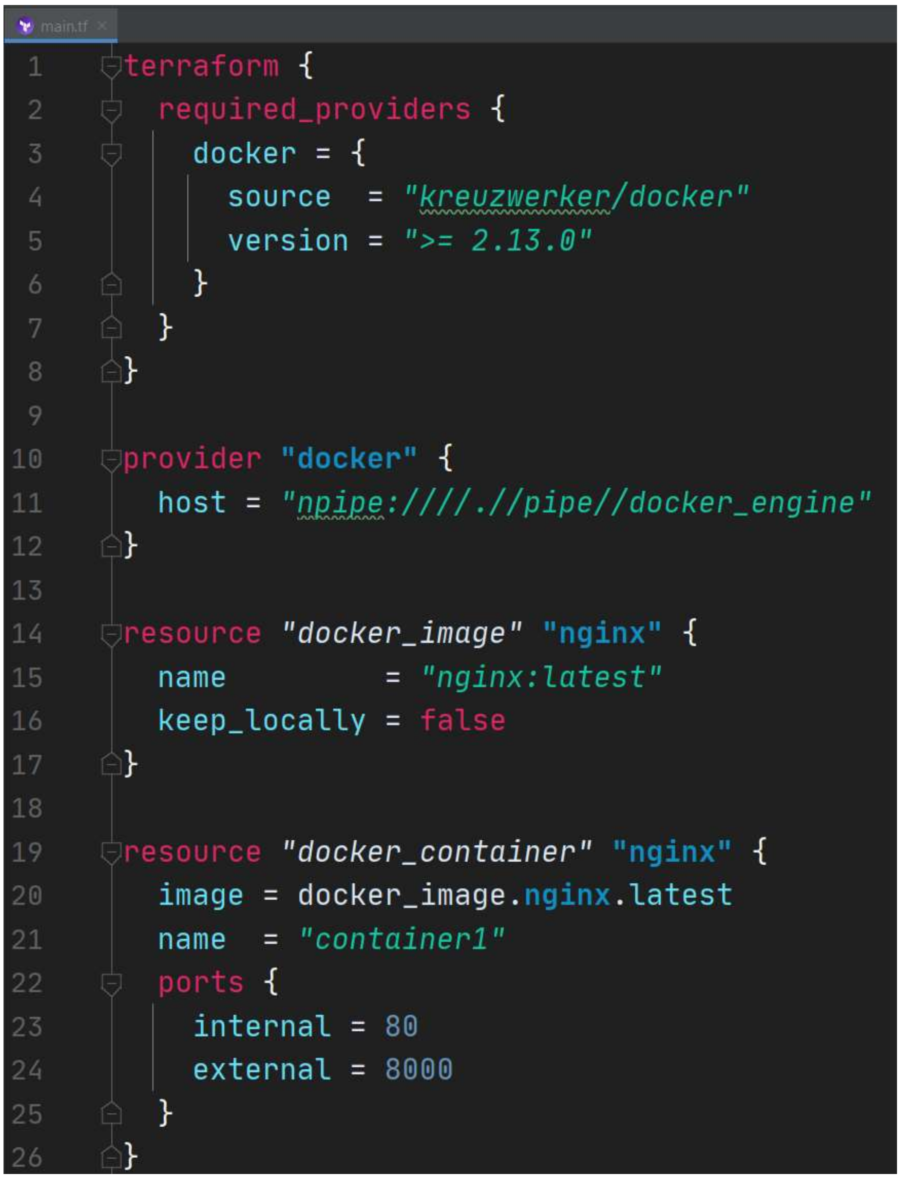

```
C:\Users\Jonas>terraform
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.
```

Figure 94 - terraform command on Windows

2.6.3. Configuration file

The main Terraform configuration file is divided in several distinct parts, describing which providers to use, which resources to create and how. We are describing how we want our infrastructure to look like (declarative language).



```
main.tf
terraform {
  required_providers {
    docker = {
      source  = "kreuzwerker/docker"
      version = ">= 2.13.0"
    }
  }
}

provider "docker" {
  host = "npipe:////.//pipe//docker_engine"
}

resource "docker_image" "nginx" {
  name         = "nginx:latest"
  keep_locally = false
}

resource "docker_container" "nginx" {
  image = docker_image.nginx.latest
  name  = "container1"
  ports {
    internal = 80
    external = 8000
  }
}
```

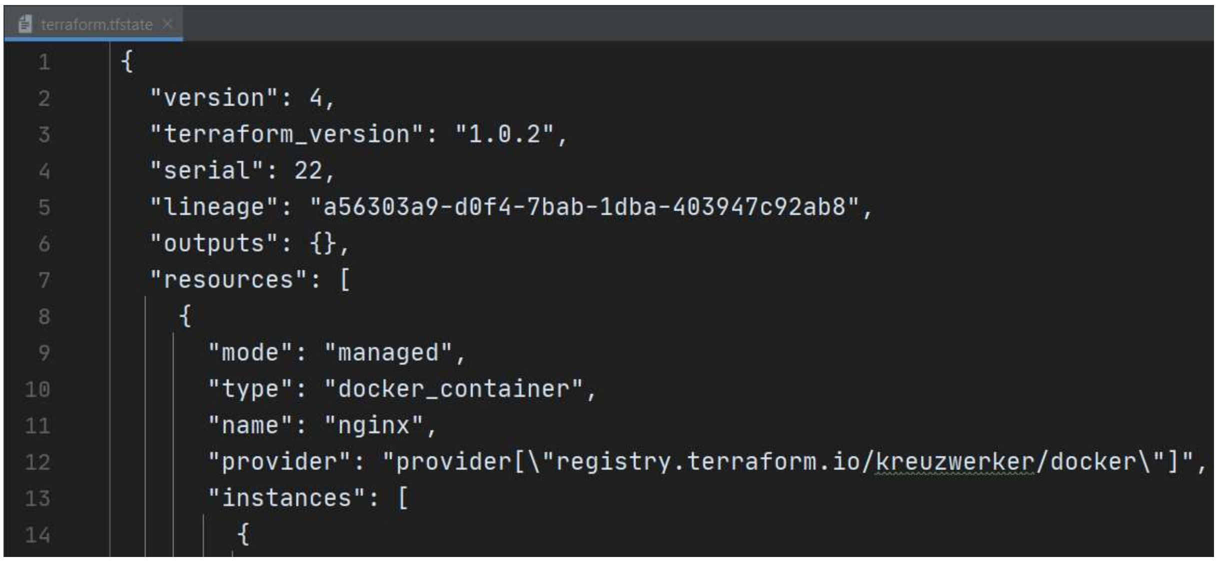
Figure 95 - Terraform configuration file example

Above is a simple example of a Terraform configuration. We are using Docker as a provider to create a Nginx web server. We distinct the “terraform” block that contains general settings including the providers that Terraform will use, the “provider” block that configures a specified provider, and the “resource” block used to describe a component of the infrastructure. There can be several providers and resources.

If a configuration file is edited, the infrastructure can be updated or recreated.

2.6.4. State file

Once the user is satisfied with the configuration, the latter can be applied and executed. Terraform indicates precisely what will be created and, once it is done, will update the state contained in the ".tfstate" file.



```
{
  "version": 4,
  "terraform_version": "1.0.2",
  "serial": 22,
  "lineage": "a56303a9-d0f4-7bab-1dba-403947c92ab8",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "docker_container",
      "name": "nginx",
      "provider": "provider[\"registry.terraform.io/kreuzwerker/docker\"]",
      "instances": [
        {
```

Figure 96 - Terraform state file sample

2.6.5. Terraform pros

- **HCL configuration files**

The HCL declarative language offers a clear and human-friendly way of provisioning an infrastructure.

- **Immutable architecture**

Terraform's immutable architecture allows applications to be deployed on a precise and stable environment.

- **Providers**

The plugins available from a very large number of providers offers a lot of choices and possibilities for an infrastructure.

- **Agentless**

Terraform is only needed as a command line tool on one machine, no other instances of the software need to be installed anywhere else.

2.6.6. Terraform cons

- **Software limitations**

While Terraform does its job well in collaboration with plenty of providers, its capabilities are still considerably limited to infrastructure provisioning and does not provide as many features as a configuration management software would. This is why Terraform is often used along with a configuration management tool.

2.6.7. When to use Terraform

Terraform should be used by companies wishing to provision their cloud-based infrastructure and manage it using the software's immutable approach. Because of that fact, the company should have its data stored remotely.

The software is particularly useful when a lot of tests are being performed on a deployed application and the environment needs to be the same at every iteration to provide the best comparison results.

3. Software Comparison

3.1. Summary of Software Characteristics

All software analysed are similar in some ways and different in others. They serve the same purpose of automation, but their functionalities and complexity vary.

Characteristic	Definition
Open-source	Defines if the software is open-source or not.
Tool type	The type of service provided by the software.
Programming language	The language the software is written in.
Configuration language	The language used to create and manage configuration files.
Language type	Defines if the software uses a procedural or declarative language.
Architecture	Defines if the software uses agents on managed nodes or not.
Server Compatibility	Operating systems the server may be installed on.
Agent Compatibility	Operating systems the agents may be installed on (server/agents). Operating systems supported as nodes (agentless).
Infrastructure	Defines if the software works with a mutable or immutable infrastructure pattern.
Scalability	Defines if the software supports scalability.
Module repository	Module repository provided by the developers, if any.

Table 1 - Software characteristics definition

The following table has been created as a **summary of facts** about all presented software. All software are open-source and have a free version available, although user interfaces and support are provided with a paid subscription.

3.1.1. Software characteristics table

	Ansible	Chef	Puppet	Salt	Terraform
Open-source	Yes	Yes	Yes	Yes	Yes
Tool type	Configuration management	Configuration management	Configuration management	Configuration management	Infrastructure provisioning
Programming language	Python	Ruby	Ruby	Python	Go
Configuration language	YAML	Ruby-based DSL	Puppet DSL	YAML	HCL
Language type	Procedural	Procedural	Declarative	Declarative	Declarative
Architecture	Agentless	Server/Agent	Server/Agent	Server/Agent	Agentless
Server Compatibility	Linux	Linux	Linux	Linux	Linux / Mac / Windows
Nodes Compatibility	Linux / Mac / Windows	Linux / Mac / Windows	Linux / Mac / Windows	Linux / Mac / Windows	Linux / Mac / Windows
Infrastructure	Mutable	Mutable	Mutable	Mutable	Immutable
Scalability	Yes	Yes	Yes	Yes	Yes
Module repository	Ansible Galaxy	Chef Supermarket	Puppet Forge	SaltStack Formulas (GitHub)	Terraform Registry

Table 2 - Software characteristics

3.2. Criteria Definition

Only Ansible, Chef, Puppet and Salt are compared. Terraform serves a similar automation purpose but is not comparable with the others as it is the only software focusing on infrastructure provisioning.

To compare all software with the most neutral point of view, a **decision matrix** has been created with the results explained below in the synthesis. To each criterion is attributed a **value** from 0 to 2.

- 0 if the software does not meet expectations.
- 1 if the software partially meets expectations.
- 2 if the software fully meets expectations.

The two software with the most points at the end are used for further testing. The guides on how to reproduce it (installation and configuration) can be found attached to this report.

The **weight** of each criterion defines its importance, as they are not all crucial. A high weight represents a higher importance, going from 1 to 3. The weight has been decided depending on what we determined are the most important aspects companies are looking for when getting a new software.

Criterion	Definition	Weight
Implementation	The initial installation and configuration are easily performed.	2
Accessibility	The architecture and configuration language are accessible to new users.	3
Complexity	The architecture and configuration language allows powerful operations and provides enough complexity.	3
Compatibility	The software supports a large variety of operating systems.	2
Scalability	The software architecture offers good scalability opportunities.	2
Performance	The software is performant, and its operations are done quickly.	2
Module repository	The software provides a well-organized remote module repository.	1
Documentation	The documentation provided is clear and complete.	1
Support	The developer and community are largely present online and provide considerable support.	1

Table 3 - Criteria definition

3.2.1. Decision matrix

	Implementation		Accessibility		Complexity		Compatibility		Scalability		Performance		Module repository		Documentation		Support		Total
Weight	2		3		3		2		2		2		1		1		1		
Ansible	2	**4**	2	**6**	1	**3**	2	**4**	2	**4**	2	**4**	2	**2**	1	**1**	2	**2**	**30**
Chef	1	**2**	0	**0**	2	**6**	2	**4**	2	**4**	1	**2**	2	**2**	2	**2**	2	**2**	**24**
Puppet	1	**2**	1	**3**	2	**6**	2	**4**	2	**4**	1	**2**	2	**2**	2	**2**	2	**2**	**27**
Salt	1	**2**	2	**6**	1	**3**	2	**4**	2	**4**	1	**2**	1	**1**	1	**1**	1	**1**	**24**

Table 4 - Decision matrix

3.3. Synthesis of the results

Looking at each criterion from left to right, we first have the implementation representing how easy and fast it is to perform the initial installation and configuration of the software. Ansible is the winner in that category, as its agentless architecture makes its installation really quick without the need to care about any agents. Other software in that category are more similar and require longer management before the communication can be established between the server and agents.

Regarding the accessibility, Ansible and Salt definitely are the easiest tools to manage due to their configuration language being YAML. Puppet provides its own language specific for its configuration, which makes the troubleshooting easier and does not require the user to learn Ruby. On the other hand, Chef uses a Ruby-based DSL and a procedural language which makes it the harder to learn, especially for non-programmers.

The complexity that the configuration language allows is also important ; Chef and Puppet both possess a powerful language inspired by an actual programming language, Ruby, which makes that complexity possible more easily and optimally than with YAML.

All software have great compatibility besides the limitations making the server side only available on Linux machines.

Likewise, their scalability opportunities are great and makes the expansion from a few servers to a hundred easy due to the nature of their architecture.

Next, the performance is slightly better with Ansible, which does not need to run any agents on the nodes. The other three are pretty similar with negligible differences in that regard.

Each software possesses a remote module repository where it is possible to download reusable modules to include in a particular project. Salt simply lacks the functionality to search efficiently because it has a GitHub repository rather than a proprietary repository.

Chef and Puppet both are more mature than Ansible and Salt, therefore their documentations are more complete than the other two. As a reminder, Chef and Puppet were initially released respectively in 2009 and 2005, while Ansible and Salt were released later respectively in 2011 and 2012.

Finally, each software offers a good support if companies pay for it. If we ignore that fact to strictly focus on a purely free usage of the tools, we notice that the online presence of Salt is smaller compared to the others three. Chef and Puppet both have a solid userbase because of their age, and Ansible too due to its increasing success the past few years.

As a conclusion to this analysis, it is important to note that in the end, there are no better or worse software because each of them targets a different userbase. We did however establish a ranking based on the facts we explained in this report for each software. According to our analysis, Ansible comes first with 30 points, Puppet second with 27 points, and last come Chef and Salt with 24 points.

4. Software Testing

4.1. Selected Software

Following the analysis, the software selected for the establishment of virtual labs are Ansible and Puppet.

4.2. Testing Scenario

The objective is to create a reproducible scenario with Ansible and Puppet defined by specific use cases. The guides can be found attached to this report, they are named :

- **02_Installation_Guide**
- **03_Configuration_Guide**

For each software, the steps for the installation and use cases for the configuration are the following :

1. **Installation**
    a. Installation of VirtualBox
    b. Creation of several virtual machines
    c. Configuration of subnetworks for each environment
    d. Installation of the operating systems on the virtual machines
    e. Installation of the configuration management software

2. **Configuration**
    a. Creation of a simple user
    b. Configuration of a Nginx web server
    c. Configuration of a MySQL database
    d. Deployment of updates and upgrades

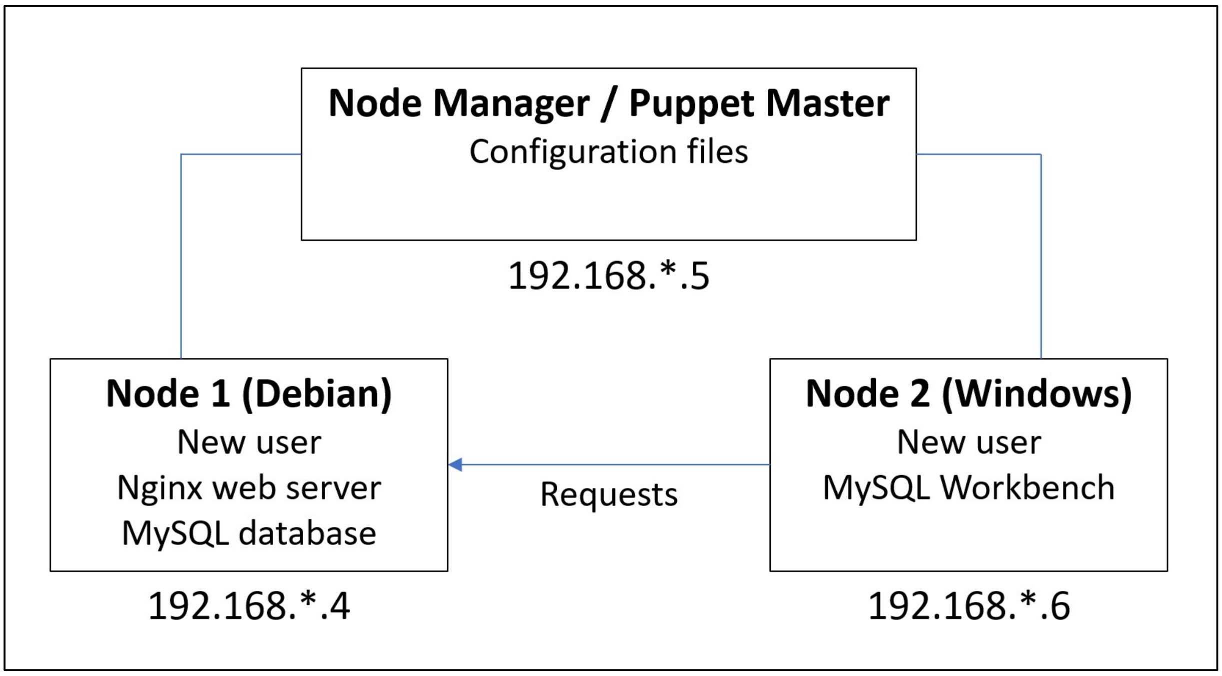


Figure 97 - Testing scenario schema

In the picture above, we observe the final structure of each lab. The node manager (Ansible) and Puppet master configure both nodes remotely. The verification then takes place with requests from the Windows node to the Debian node to make sure that everything works as intended.

4.3. Conclusion

After more in-depth testing, it has been established that Ansible was superior in several ways. The information found about the software online was of better quality and more abundant than Puppet's. The ease of use of the configuration language was even more obvious than expected ; there was very few research needed to execute the use cases in the Ansible environment, while the Puppet environment requested a lot more work. This can be explained by the difficulty to learn the language as well as the fact that it provides a lot of functionalities through modules from the Puppet Forge ; Ansible has more native functionalities, at least in the scope of the testing that took place.

For all the functionalities Ansible has demonstrated, it would currently be our favourite choice for an implementation of a configuration management software in a company.

5. References

Aleksic, M. (2020, December 17). *How Does SSH Work*. Retrieved from PhoenixNAP: https://phoenixnap.com/kb/how-does-ssh-work

Alfke, M., Franceschi, A., Frank, F., Pastor, J. S., & Uphillis, T. (2017). *Puppet: Mastering Infrastructure Automation.* Packt Publishing.

Ansible. (n.d.). Retrieved from Ansible Website: https://www.ansible.com/

Ansible Documentation. (n.d.). Retrieved from Ansible Documentation: https://docs.ansible.com/

Ansible Verwendungsszenarien. (n.d.). Retrieved from Kreyman: https://www.kreyman.de/index.php/cloud-microsoft-azure-aws-amazon-web-services/213-ansible-verwendungsszenarien

Bigelow, S. J. (2020, November). *Infrastructure as code*. Retrieved from SearchITOperations: https://searchitoperations.techtarget.com/definition/Infrastructure-as-Code-IAC

Bruce, S. (2020, December 18). *Ansible vs Chef: Which Configuration Management Tool Is Best?* Retrieved from Career Karma: https://careerkarma.com/blog/ansible-vs-chef/

Chef Documentation. (n.d.). Retrieved from Chef Documentation: https://docs.chef.io/

Chef Downloads. (n.d.). Retrieved from Chef: https://downloads.chef.io/

Dadgar, A. (2018, November 15). *What is Mutable vs. Immutable Infrastructure?* Retrieved from HashiCorp: https://www.hashicorp.com/resources/what-is-mutable-vs-immutable-infrastructure

DevOps principles. (n.d.). Retrieved from Atlassian: https://www.atlassian.com/devops/what-is-devops

Dharmalingam, N. (2019, November 5). *Advantages and Disadvantages of Ansible*. Retrieved from Whizlabs: https://www.whizlabs.com/blog/ansible-advantages-and-disadvantages/

Dharmalingam, N. (2020, February 11). *Chef vs Puppet vs Ansible*. Retrieved from Whizlabs: https://www.whizlabs.com/blog/chef-vs-puppet-vs-ansible/

Dharmalingam, N. (2020, February 19). *Introduction to Chef*. Retrieved from Whizlabs: https://www.whizlabs.com/blog/chef-introduction/

Domain-specific language. (2021, May 28). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Domain-specific_language

Domont, A. (2021, May 12). *Utilisez Ansible pour automatiser vos tâches de configuration*. Retrieved from OpenClassrooms: https://openclassrooms.com/en/courses/2035796-utilisez-ansible-pour-automatiser-vos-taches-de-configuration

Ewart, J., Marschall, M., & Waud, E. (2017). *Chef: Powerful Infrastructure Automation.* Packt Publishing.

Gaba, I. (2021, April 1). *What Is Chef: Here's What You Need to Know*. Retrieved from Simplilearn: https://www.simplilearn.com/tutorials/chef-tutorial/what-is-chef

HashiCorp. (n.d.). *Explore our tutorials to automate your workflows*. Retrieved from HashiCorp Learn: https://learn.hashicorp.com/

HashiCorp. (n.d.). *Packer Documentation*. Retrieved from HashiCorp: https://www.packer.io/docs

HashiCorp. (n.d.). *Vagrant Documentation*. Retrieved from HashiCorp: https://www.vagrantup.com/docs

Henderson, B. (2018, April 24). *Connecting to a Windows Host*. Retrieved from Ansible: https://www.ansible.com/blog/connecting-to-a-windows-host

Infrastructure as code. (2021, June 15). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Infrastructure_as_code

Jacobs, M., & Kaim, E. (2021, May 12). *What is Infrastructure as Code?* Retrieved from Microsoft Documentation: https://docs.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code

Janashia, N. (2020, July 4). *Terraform simply explained*. Retrieved from DEV Community: https://dev.to/techworld_with_nana/terraform-simply-explained-m

javainterviewpoint. (2019, March 11). *Java RSA Encryption and Decryption Example | ECB Mode + 4096 Bits + OAEPWITHSHA-512ANDMGF1PADDING*. Retrieved from Java Interview Point: https://www.javainterviewpoint.com/rsa-encryption-and-decryption/

Johari, A. (2019, November 26). *Chef vs Puppet vs Ansible vs Saltstack: Which Works Best For You?* Retrieved from Edureka: https://www.edureka.co/blog/chef-vs-puppet-vs-ansible-vs-saltstack/

Krebsbach, H. (2016, November 9). *Picking the right tools for DevOps communication*. Retrieved from Atlassian: https://www.atlassian.com/blog/devops/picking-right-tools-devops-communication

Krout, E. (2021, May 21). *A Beginner's Guide to Chef*. Retrieved from Linode: https://www.linode.com/docs/guides/beginners-guide-chef/

Linode. (2020, October 7). *Getting Started with Salt - Basic Installation and Setup*. Retrieved from Linode: https://www.linode.com/docs/guides/getting-started-with-salt-basic-installation-and-setup/

Linode. (2020, October 7). *How To Install a Chef Server Workstation on Ubuntu 18.04*. Retrieved from Linode: https://www.linode.com/docs/guides/install-a-chef-server-workstation-on-ubuntu-18-04/

Linode. (2021, June 11). *Getting Started with Puppet - Basic Installation and Setup*. Retrieved from Linode: https://www.linode.com/docs/guides/getting-started-with-puppet-6-1-basic-installation-and-setup/

Lyman, J. (2020, October 7). *Getting Started With Ansible - Basic Installation and Setup*. Retrieved from Linode: https://www.linode.com/docs/guides/getting-started-with-ansible/

Myers, C. (2016). *Learning SaltStack : build, manage, and secure your infrastructure by utilizing the power of SaltStack.* Packt Publishing.

OpenSSH/SSH Protocols. (2020, November 9). Retrieved from Wikibooks: https://en.wikibooks.org/wiki/OpenSSH/SSH_Protocols

Puppet. (n.d.). Retrieved from Puppet Website: https://puppet.com/

Puppet Documentation. (n.d.). Retrieved from Puppet Documentation: https://puppet.com/docs/

RedHat. (n.d.). *What is cloud automation?* Retrieved from RedHat: https://www.redhat.com/en/topics/automation/what-is-cloud-automation

RedHat. (n.d.). *What is provisioning?* Retrieved from RedHat: https://www.redhat.com/en/topics/automation/what-is-provisioning

RSA (cryptosystem). (2021, June 8). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/RSA_(cryptosystem)

Ruby (programming language). (2021, July 5). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Ruby_(programming_language)

Salt Project. (n.d.). Retrieved from Salt Project Website: https://saltproject.io/

Salt system architecture. (2021, April 13). Retrieved from VMware Documentation: https://docs.vmware.com/en/vRealize-Automation/8.4/install-configure-saltstack-config/GUID-8FC70D95-3317-4324-A5BD-D213CE9B029E.html

SaltStack. (n.d.). *Salt Project Documentation*. Retrieved from Salt Project Documentation: https://docs.saltproject.io/en/latest/

Shirinkin, K. (2017). *Getting started with Terraform : manage production infrastructure as a code.* Packt Publishing.

Simplilearn. (2021, February 9). *What is Puppet: Components and Writing Manifests Explained*. Retrieved from Simplilearn: https://www.simplilearn.com/what-is-puppet-article

Srivastava, D. (2021, February 12). *Chef vs. Puppet vs. Ansible vs. Saltstack: A Complete Comparison*. Retrieved from Medium: https://medium.com/successivetech/chef-vs-puppet-vs-ansible-vs-saltstack-a-complete-comparison-9af8f1790c0d

Strutt, S. (2018, November 13). *Choosing an Infrastructure as Code tool.* Retrieved from IBM: https://www.ibm.com/cloud/blog/chef-ansible-puppet-terraform

Terra, J. (2021, April 12). *Ansible vs Chef: What's the Difference?* Retrieved from Simplilearn: https://www.simplilearn.com/ansible-vs-chef-differences-article

Terraform. (n.d.). Retrieved from Terraform Website: https://www.terraform.io/

Terraform. (n.d.). *Terraform Documentation*. Retrieved from Terraform Documentation: https://www.terraform.io/docs/index.html

UpGuard. (2020, October 19). *Ansible vs Chef Updated for 2020 [Infographic]*. Retrieved from UpGuard: https://www.upguard.com/blog/ansible-vs-chef

UpGuard. (2020, August 2020). *Chef vs Salt: Which One to Choose?* Retrieved from UpGuard: https://www.upguard.com/blog/salt-vs-chef

Veritis. (n.d.). *Chef Vs Puppet Vs Ansible – Comparison of DevOps Configuration Management Tools*. Retrieved from Veritis: https://www.veritis.com/blog/chef-vs-puppet-vs-ansible-comparison-of-devops-management-tools/

VMware Documentation. (n.d.). Retrieved from VMware Documentation: https://docs.vmware.com/

Wikipedia. (2021, June 21). *Secure Shell Protocol*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Secure_Shell_Protocol

Windows Remote Management. (2018, May 31). Retrieved from Microsoft: https://docs.microsoft.com/en-us/windows/win32/winrm/portal

YAML. (2021, June 23). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/YAML

Author's Declaration

I hereby certify that I have written the present Bachelor's thesis on my own, without any help other than listed in the reference section, and that I have not used any sources other than the ones specifically mentioned. I will not give any copies of this report to anyone without the authorisation of both the RF and the supervisor of the Bachelor's thesis. This does not include the persons who have provided me with the key information required for writing this thesis and which are listed hereafter :

Xavier Barmaz

August 3, 2021
Jonas Bruchez