ANSIBLE

SUCCINCTLY

BY **ZORAN MAKSIMOVIC**

SUCCINCTLY EBOOK SERIES

Syncfusion®

Ansible Succinctly

By

Zoran Maksimovic

Foreword by Daniel Jebaraj

ISBN: 978-1-64200-216-4

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, CEO
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Zoran Maksimovic is a solution architect and software developer with more than 20 years of professional experience. He is passionate about programming and technology.

He specializes in Microsoft.NET, OOD, TDD, DDD (Domain Driven Development), CQRS/ES, Event streaming, and more recently, DevOps, Linux, and Python. He is also a big fan of Agile Methodology.

Zoran is also the author of *ServiceStack Succinctly*, *MongoDB 3 Succinctly*, and *Akka.NET Succinctly*.

Zoran enjoys guitar, baroque music, good food, and Italian wine. He is a proud father of three: Alexei, Xenia, and Sofia.

For more info, visit https://zoran.me.

Introduction

Ansible is an open-source automation engine used for provisioning, configuration management, application deployment, and orchestration. The core Ansible is written in Python and can be used on Unix-like machines or Microsoft Windows.

I wrote this book primarily to make you aware of infrastructure automation made in Ansible, and to help you start working with this technology in the fastest possible way. My hope is that after reading it, you will have enough knowledge to start coding and using Ansible effectively. While this book is not fully comprehensive of the technology, it should give a good grounding in performing the most useful operations.

Target audience

This book is intended for DevOps practitioners, or in general, software developers or readers with a notion of scripting or programming. It would be useful for any kind of application developer involved with the following areas:

- Infrastructure provisioning and automation.
- Application deployments.
- Configuration management.
- Interest in writing Infrastructure as Code (IaC).

You should already be familiar with Linux OS commands and Bash shell, and have some idea of any programming language, as those concepts are mostly left unexplained.

Git, Apache HTTP Server (httpd), Firewall, Linux in general, MongoDB, and others won't be covered in this book, so the expectation is that these topics are to be understood separately. Where appropriate, I've provided links (usually in the footer) for more information.

I've taken some shortcuts in this book—the intention is not to go too deep into the details, but rather to show the various concepts, options, and possibilities.

Additional resources

You can find a lot of additional information about Ansible directly on the Ansible website.

If you want to know more about some of the technologies mentioned in this book, take a look at the following resources:

- *Linux Succinctly*
- Bash (Unix shell)
- *MongoDB Succinctly*
- Firewalld

- Apache Server
- PHP

Ansible source code

Ansible is an open-source framework, and at the time of writing, it's hosted on GitHub. Ansible itself is written in Python.

Ansible useful links

There are several groups on the web that provide useful information and answer common questions. Here are a few links that you might find useful:

- Official Ansible website
- Ansible source code on GitHub
- Official Ansible Twitter account
- Ansible official documentation

Software requirements

To get the most out of this book and the included examples, you will need to have a version of the Microsoft Visual Studio Code IDE installed (or any other file editor you prefer).

All of the examples in this book have been written and tested on Microsoft Windows 10, Linux CentOS 8, and Microsoft Visual Studio Code.

Conventions used in the book

There are specific formats that you will see throughout this book to illustrate tips and tricks or other important concepts.

Note: This icon will identify things to note throughout the book.

Tip: This icon will identify tips and tricks throughout the book.

Code in this book

Source code is written in a consistent manner. Command prompt (terminal) code follows the following style.

Code Listing 1: Command prompt code style

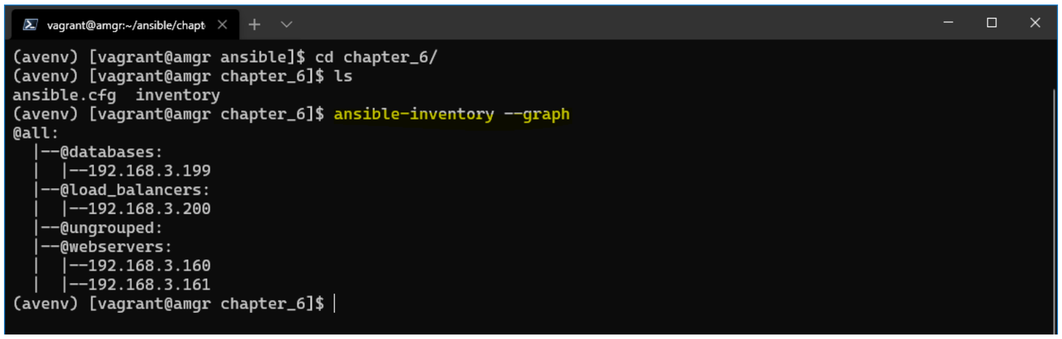
```
$ command
```

Most of the coding examples are written in `YAML`, and the following formatting style is used when working with it.

Code Listing 2: YAML code style

```yaml
---
- name: Web Server Playbook
  hosts: webservers
  become: yes

  tasks:
    - name: Pinging web server
      ansible.builtin.ping:
        data: pong
```

Most of the results of executing commands are shown as a windows terminal image.



```
(avenv) [vagrant@amgr ansible]$ cd chapter_6/
(avenv) [vagrant@amgr chapter_6]$ ls
ansible.cfg  inventory
(avenv) [vagrant@amgr chapter_6]$ ansible-inventory --graph
@all:
  |--@databases:
  |  |--192.168.3.199
  |--@load_balancers:
  |  |--192.168.3.200
  |--@ungrouped:
  |--@webservers:
  |  |--192.168.3.160
  |  |--192.168.3.161
(avenv) [vagrant@amgr chapter_6]$ 
```

Figure 1: Example of a result from the command line

Resources

You can check out the code mentioned in this book here.

Ansible version

All the examples and explanations apply to Ansible v2.10, which is the latest stable version at the time of writing.

Chapter 1 Introduction

Ansible is an open-source software, automation engine, and automation language mainly used in software configuration management, infrastructure provisioning, configuration management, application deployment, and orchestration. The Ansible automation engine executes Ansible playbooks.

The main qualities of Ansible are:

- **Simple**: Playbooks are readable and easy to understand. Playbooks will contain some tasks that will be executed in the order in which they are written. No special coding skills are required.
- **Powerful**: Ansible can manage infrastructure, networks, operating systems, and other resources, straight out of the box. It enables us to orchestrate the entire infrastructure and environment lifecycle (cloud and on premises).
- **Agentless**: Uses Open SSH and Windows Remote Management. No additional firewall ports need to be open.

Ansible, initially created in 2012, was acquired by Red Hat in 2015.

Why do we need Ansible?

As a software developer or system administrator, you are aware of how challenging it is to keep the application deployment and server's management efficient and reliable.

System administrators at one point in time managed servers by hand (and this is sometimes still the case). This obviously included installing the operating system and keeping it up to date, installing the software needed for the application to run, changing configuration for application deployment, and a myriad of other tasks.

Given the fact that we live in an information age, and that the usage of the typical applications, now internet-facing, has grown to an unthinkable size, manual system management simply doesn't work anymore.

Application development, now being very agile, has become quicker, as the time to market is one of the key factors (as it has always been!). Software releases have become more frequent, and scalability and elasticity of the applications are requiring an effort that can no longer be managed manually.

In simple terms: everything is more complex, bigger, and faster!

This is why configuration management tools such as Ansible, Puppet, Chef, and SaltStack came to thrive as solutions to the problems I just mentioned.

What can Ansible be used for?

Ansible has a wide range of usages, covered in the following sections.

Infrastructure provisioning

Infrastructure provisioning is the process of setting up the IT infrastructures, which refers to the components, hardware, and software needed to operate an application service or system. Bear in mind that *provisioning* is not the same as *configuration*, but both are steps in the deployment process.

Infrastructure as code (IaC) is a term that describes the ability to script and code the infrastructure, as we would do for other kinds of software. This obviously has the great benefit of being traceable, versioned, and rolled back if necessary.

Infrastructure has never stopped evolving, and Ansible offers great support for automation of the following aspects:

- **Virtualization**: How to provision infrastructure in minutes rather than days.
- **Containerization**: How to provision infrastructure in seconds instead of minutes.
- **Cloud-based resources**: How to provision resources you don't own.
- **Serverless**: How to provision infrastructure on demand.

The chance you are working in the cloud and using virtualization or container-based deployments is very high! Ansible, in that sense, has great support for all of the major cloud providers and supports industry-leading virtualization platforms such as VMware, Vagrant, and Red Hat Virtualization.

Support that Ansible gives is available for all of the major operating systems: Microsoft Windows, Linux (Ubuntu, CentOS, RHEL, Fedora, and others), Unix, and OS X.

An example of infrastructure provisioning might include all of the operations needed to create a new machine (server) and bring it to a working state, including defining the desired state of the system.

Configuration management

Configuration management is the process of maintaining infrastructure and software in a desired and consistent state. It's a way to make sure that a system performs as expected as changes are made over time.

This is particularly important when it comes to applying changes to the resources. The goal, especially with automation, is to keep the configuration changes transparent and documented. Without it, we couldn't simply know what changes have been applied to which resource, which again, makes it hard to maintain.

One example is that we would like the production system to have exactly the same settings as a development system (or vice-versa). In that way, we can ensure the consistency of the state of the environment.

With configuration management, you can accurately replicate an environment with the correct configurations and software, as it is documented!

The great benefits of automating and having the configuration management processes in place is that it makes the deployments faster, removes the possibility of the human errors, and manages the system in a predictable and stable state.

Application deployment

With Ansible, teams are able to manage the entire application lifecycle effectively from development to production. Ansible offers a simple way to deploy your multi-tier application in a reliable and consistent way.

Although Ansible does not directly perform source and version control, it has great support for application source control systems like Git and Subversion.

You can configure needed services as well as push application artifacts from one common place. Ansible doesn't require agents on remote systems, and it offers the possibility to execute a *playbook* that contains a list of tasks that will be executed in order. That order will always be consistent.

Orchestration

Ansible provides orchestration in the sense of aligning the business request with the applications, data, and infrastructure.

It obviously helps define the policies and service levels through *automated workflows*, *provisioning*, and *change management.* This creates an application-aligned infrastructure that can be scaled up or down based on the needs of each application. This is especially useful when working in an enterprise environment.

Chapter 2 High-Level View

When it comes to the architecture, Ansible is a straightforward automation engine. Its components and the relationships among them are shown in Figure 2.

Ansible works by connecting via **`ssh`** to the hosts (without the need for a special agent to be installed on the host itself), and by pushing *modules* to the hosts itself. The modules are then executed locally on the host, and the output is pushed back to the Ansible server.

Since it uses **`ssh`**, it can very easily connect to clients using **`SSH-Keys`** authentication, which simplifies the whole process.

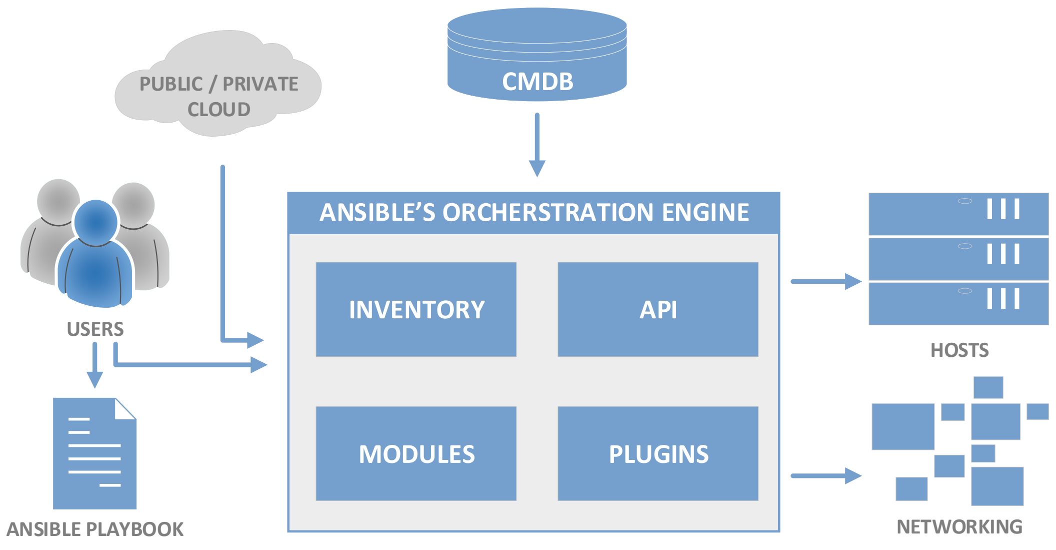


Figure 2: Ansible orchestration engine architecture

Users

Users are simply the developers, operators, and DevOps practitioners who are writing or executing the automation scripts. This automation and orchestration script is called the Ansible playbook. A user can execute those scripts by using the Ansible orchestration engine.

Modules

Modules are executed directly on remote hosts through playbooks or by running them individually from the command line. Modules, therefore, are the units of code Ansible executes.

Each module has a particular use, from administering users on a specific type of database, to managing VLAN interfaces on a specific type of network device.

Ansible allows users to write their own modules and provides out-of-the-box core (maintained by the Ansible team) or extras modules (maintained by community).

Some of the most commonly used modules are:

- File handling: file, stat, copy, template
- Remote execution: command, shell
- Service management: service
- Package management: apt, yum, bsd, ports
- Source control system: git, subversion

To get an idea of the scope of the available Ansible modules, take a look at the list of all modules.

Plugins

Plugins should not be confused with Ansible modules. While modules are executed on the managed hosts, plugins are extensions to the Ansible runtime. Operations such as data transformation, logging of the output, and inventory handling are plugins.

Plugins are often working in conjunction with modules.

Inventories

Ansible works against multiple managed nodes or hosts that are part of the infrastructure, and the list of those items is also known as the *inventory*.

Inventory is a file, defined in a YAML or INI format, that contains a list of hosts (nodes) along with their IP addresses, servers, and databases, which need to be managed. Ansible then takes action via a transport to connect to them: **`ssh`** for UNIX, Linux, or networking devices; and **`WinRM`** for Windows system.

Ansible playbooks

Playbooks are files (scripts) that combine configuration, deployment, and orchestration functions. Playbooks are executed to provide a way of automating the remote systems in a consistent and repeatable manner.

Playbooks will execute predefined tasks, such as installing a new package on a remote system, and tasks on their own will use modules to provide such a functionality. In that sense, playbooks can be seen as the ultimate place where all the automation code converges.

Playbooks are human-readable and use the **`YAML`** format, which is easy to write and understand.

Chapter 3 Environment Setup

Before starting to work with Ansible, we need to set up the environment against which we will be performing the examples and exercises.

For simplicity, this book assumes you are using Microsoft Windows 10 on a desktop computer. Therefore, we will be using Microsoft Windows 10 to perform the basic setup. Windows 10 will be purely used to host the VM. The rest of the examples will be run on **CentOS 8** Linux.

Figure 3 depicts the set of virtual machines we are going to create, which will be installed locally (on your desktop computer). The virtual machines are all based on CentOS 8 and will be running on VirtualBox. To have them up and running, you will need to have about 6GB of RAM available. However, it's not necessary that all the servers run at the same time, if that's the limitation you have in your environment.

The environment presented is a possible setup for a web application that is load balanced, that can be possibly scaled out (by attaching more web servers), and that has a backend system serving the data (database).

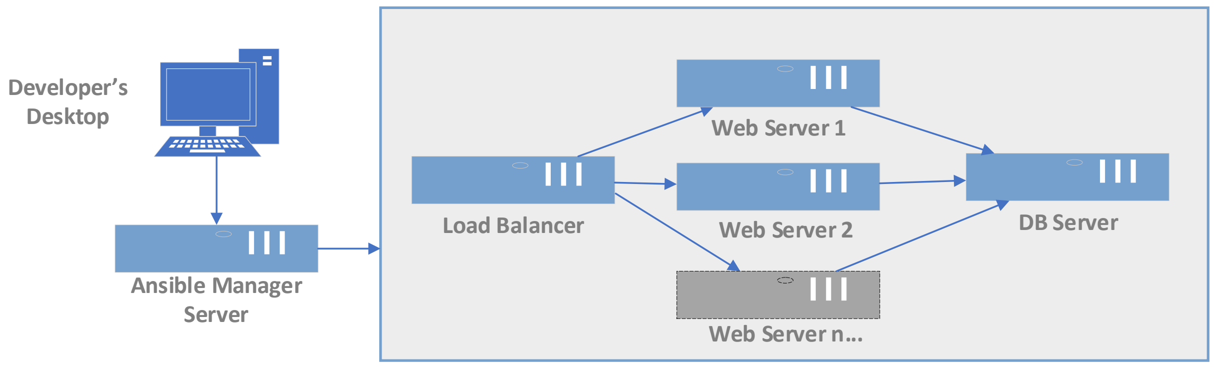


Figure 3: Development environment

Let's briefly explain each of the hosts mentioned in Figure 3.

Table 1: Server (Hosts) List

Server Purpose	Hostname	Description	(Static) IP Address
Ansible Manager Server	`amgr`	This is the virtual machine where Ansible will be installed. We will perform all of the Ansible operations directly from this server against other hosts.	`192.168.3.100`
Load Balancer	`lb`	Host responsible for redirecting `HTTP` requests to	`192.168.3.200`

Server Purpose	Hostname	Description	(Static) IP Address
		either web server 1 or web server 2.	

Web Server	`web160`
	`web161`
		Server hosting the web application.	`192.168.3.160`
			`192.168.3.161`

DB Server	`db`	Server hosting the database.	`192.168.3.199`

Note: It would be good to familiarize yourself with Table 1, since we will be using these hostnames and IP addresses throughout the examples in the book.

In order to have the recommended setup, you need to install the following tools on your desktop machine:

- Vagrant
- VirtualBox
- Git
- Visual Studio Code

Let's just briefly explain the tools, their purpose, and the installation procedure.

Vagrant

"Vagrant is a tool for building and managing virtual machine environments in a single workflow. With an easy-to-use workflow and focus on automation…" (Vagrant website).

We need Vagrant to create the Linux VMs, as stated previously.

You can install the Vagrant software by downloading it here. Just follow the installation wizard, which is quite straightforward. If there is already a version of Vagrant installed, it will be upgraded. You might need to restart your computer.

After the installation, please verify that the software has been installed correctly by opening the command prompt (PowerShell) and typing the following command.

Code Listing 3: Checking the Vagrant version

```
PS C:\>vagrant -v
```

You should receive no errors, and the result should be something like **`Vagrant 2.2.15`**, which is the version used at the time of writing this book.

We need also to install a Vagrant plugin called **`vagrant-hostmanager`**, which will help us in setting up the connectivity between hosts by manipulating the **`/etc/hosts`** file at the time of provisioning of the machines.

Please run the following command.

Code Listing 4: Installation of the vagrant-hostmanager plugin

```
PS C:\>vagrant plugin install vagrant-hostmanager
```

The version used at the time of writing is 1.8.9. To check that the plugin got installed (and the version), we can run the following command.

Code Listing 5: Check the installed Vagrant plugins

```
PS C:\>vagrant plugin list
```

VirtualBox

According to its website, VirtualBox is a powerful virtualization product for enterprise as well as home use. We will be using VirtualBox in order to run the aforementioned virtual machines.

On Windows, you can install VirtualBox by visiting this site. After downloading the application, install it by following the wizard. You can simply keep the default options.

Tip: Keep in mind that only one hypervisor can be active at the same time, so if your Windows 10 has the Hyper-V (or other) installed and active, you might need to deactivate it in order to run VirtualBox.

After installing VirtualBox, please also install the **VirtualBox Extension Pack**. The link to the extension pack is typically in the link provided.

At the time of writing, the version of the Virtual Box application is 6.1.18.

Visual Studio Code

Visual Studio Code is a free and open-source code-editing app we are going to use to work with Ansible scripts. Please install Visual Studio Code by following this link. The version used in this book is **1.55.2**.

After installing the tool, please install some plugins that will help you with the scripting, such as the ones listed in the following table.

Table 2: Visual Studio Code Plugins

Plugin Name	Plugin Identifier	Short Description
Vagrantfile Support	marcostazi.vs-code-vagrantfile	Provides syntax highlighting support for Vagrantfile, which is the Vagrant configuration file.
Jinja	wholroyd.jinja	Jinja template editing support.
Ansible	haaaad.ansible	Ansible language support.
Remote ssh connections support	ms-vscode-remote.remote-ssh	Microsoft plugin to connect to remote servers via ssh.
Remote ssh connections support	ms-vscode-remote.remote-ssh-edit	Microsoft plugin to connect to remote servers via ssh and edit files.
Remote ssh connections support	ms-vscode-remote.vscode-remote-extensionpack	Microsoft plugin.
Ansible Mod	sysninja.vscode-ansible-mod	Ansible editing helper functions.

By using the following commands and running them from the PowerShell command line, you can automate the installation of the plugins mentioned in the previous table.

Code Listing 6: Visual Studio Plugin Installation via command line

```
code --install-extension marcostazi.vs-code-vagrantfile
code --install-extension wholroyd.jinja
code --install-extension haaaad.ansible
code --install-extension ms-vscode-remote.remote-ssh
code --install-extension ms-vscode-remote.remote-ssh-edit
code --install-extension ms-vscode-remote.vscode-remote-extensionpack
code --install-extension sysninja.vscode-ansible-mod
```

Infrastructure installation procedure

Now that the tools are installed, please create a folder in which we will place the **`Vagrantfile`** configuration.

For example, I have created the following two folders:

- **C:\AnsibleSuccinctly** to hold all the Ansible scripts.
- **C:\AnsibleSuccinctly\Vagrant** (subfolder) to place the Vagrantfile to start the environment.

You can execute in the command line (PowerShell) the following script.

Code Listing 7: Folder creation

```
PS C:\>mkdir AnsibleSuccinctly\Vagrant
```

After creating the folders, we also need to create the Vagrantfile.

Code Listing 8: Create Vagrantfile via command line

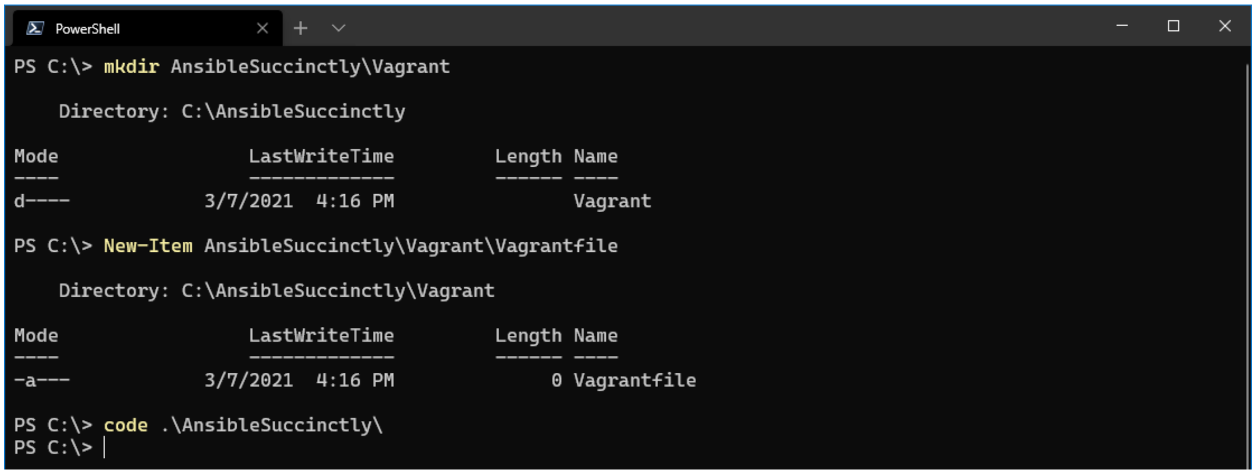
```
PS C:\>New-Item C:\AnsibleSuccinctly\Vagrant\Vagrantfile
```

Let's open the folder in Visual Studio Code.

Code Listing 9: Opening the folder with Visual Studio Code

```
PS C:\>code C:\AnsibleSuccinctly\
```

You should now have the basic folders and a file called Vagrantfile created and opened in Visual Studio Code.



```
PS C:\> mkdir AnsibleSuccinctly\Vagrant

    Directory: C:\AnsibleSuccinctly

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----           3/7/2021   4:16 PM                Vagrant

PS C:\> New-Item AnsibleSuccinctly\Vagrant\Vagrantfile

    Directory: C:\AnsibleSuccinctly\Vagrant

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---           3/7/2021   4:16 PM              0 Vagrantfile

PS C:\> code .\AnsibleSuccinctly\
PS C:\> |
```

Figure 4: Creation of the Vagrantfile

After running the commands, you should also see Visual Studio Code opening the folder **AnsibleSuccinctly**. You should get something similar to the following figure.

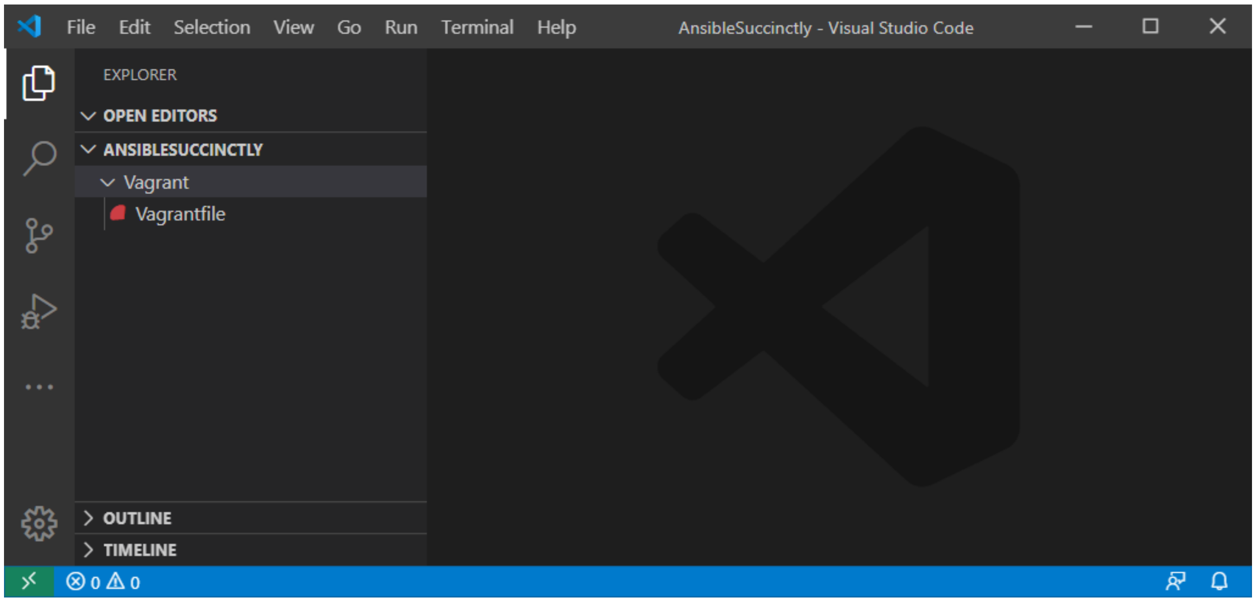

File Edit Selection View Go Run Terminal Help
AnsibleSuccinctly - Visual Studio Code
EXPLORER
OPEN EDITORS
ANSIBLESUCCINCTLY
Vagrant
Vagrantfile
OUTLINE
TIMELINE
0 0

Figure 5: Visual Studio Code view of the folder

Open the file called **Vagrantfile**, paste the following content, and save the file, which contains the script that will be executed to create the VMs.

Code Listing 10: Vagrantfile content

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

BOX_IMAGE = "Dougs71/CentOS-8.1.1911"
BOX_VERSION = "1.0.0"

Vagrant.configure("2") do |config|

  config.hostmanager.enabled = true
  config.hostmanager.manage_host = false

  #Ansible manager definition
  config.vm.define "amgr" do |amgr|
    amgr.vm.box = BOX_IMAGE
    amgr.vm.box_version = BOX_VERSION
    amgr.vm.hostname = 'amgr'
    amgr.vm.network :private_network, ip: "192.168.3.100"
    amgr.vm.provider :virtualbox do |v|
      v.memory = 2048
      v.cpus = 4
      v.name = "amgr"
    v.customize ["modifyvm", :id, "--natdnshostresolver1", "on"]
    end
```

```ruby
  end

  #Load balancer definition
  config.vm.define "lb" do |lb|
    lb.vm.box = BOX_IMAGE
    lb.vm.box_version = BOX_VERSION
    lb.vm.hostname = 'lb'
    lb.vm.network :private_network, ip: "192.168.3.200"
    lb.vm.provider :virtualbox do |v|
      v.memory = 1024
      v.cpus = 1
      v.name = "lb"
    end
  end

  #Database definition
  config.vm.define "db" do |db|
    db.vm.box = BOX_IMAGE
    db.vm.box_version = BOX_VERSION
    db.vm.hostname = 'db'
    db.vm.network :private_network, ip: "192.168.3.199"
    db.vm.provider :virtualbox do |v|
      v.memory = 1024
      v.name = "db"
    end
  end

  #Creation of the web application servers
  (160..161).each do |host|
    config.vm.define "web#{host}" do |web|
      web.vm.box = BOX_IMAGE
      web.vm.box_version = BOX_VERSION
      web.vm.hostname = 'web#{host}'
      web.vm.network :private_network, ip: "192.168.3.#{host}"
      web.vm.hostname = "web#{host}"
      web.vm.provider :virtualbox do |v|
        v.memory = 1024
        v.cpus = 1
        v.name = "web#{host}"
      end
    end
  end
end
```

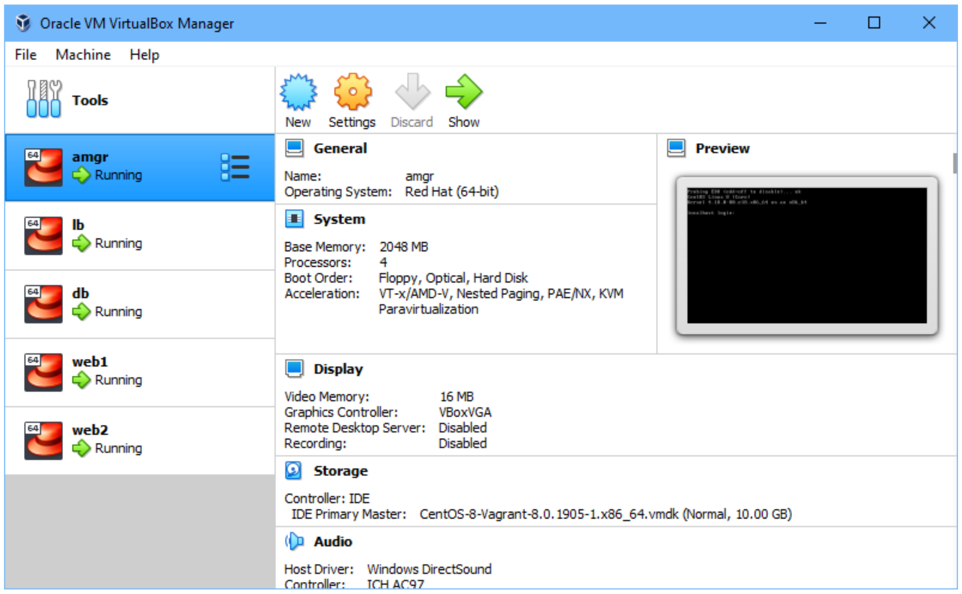
Finally, let’s create the VirtualBox images by running the following command in the Vagrant folder.

Note: Vagrant needs to download the CentOS 8 image from the Vagrant online repository. The speed will depend on speed of your internet connection. As an indication, you might experience 3–5 minutes of waiting time for the creation of all five machines.

Code Listing 11: Starting the VM creation

```
PS C:\>AnsibleSuccinctly\Vagrant>vagrant up
```

You might open VirtualBox at the same time and see in real time how the VMs are getting created. The result should look similar to what is shown in the following figure.



General
Name: amgr
Operating System: Red Hat (64-bit)

System
Base Memory: 2048 MB
Processors: 4
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX, KVM Paravirtualization

Display
Video Memory: 16 MB
Graphics Controller: VBoxVGA
Remote Desktop Server: Disabled
Recording: Disabled

Storage
Controller: IDE
IDE Primary Master: CentOS-8-Vagrant-8.0.1905-1.x86_64.vmdk (Normal, 10.00 GB)

Audio
Host Driver: Windows DirectSound
Controller: ICH AC97

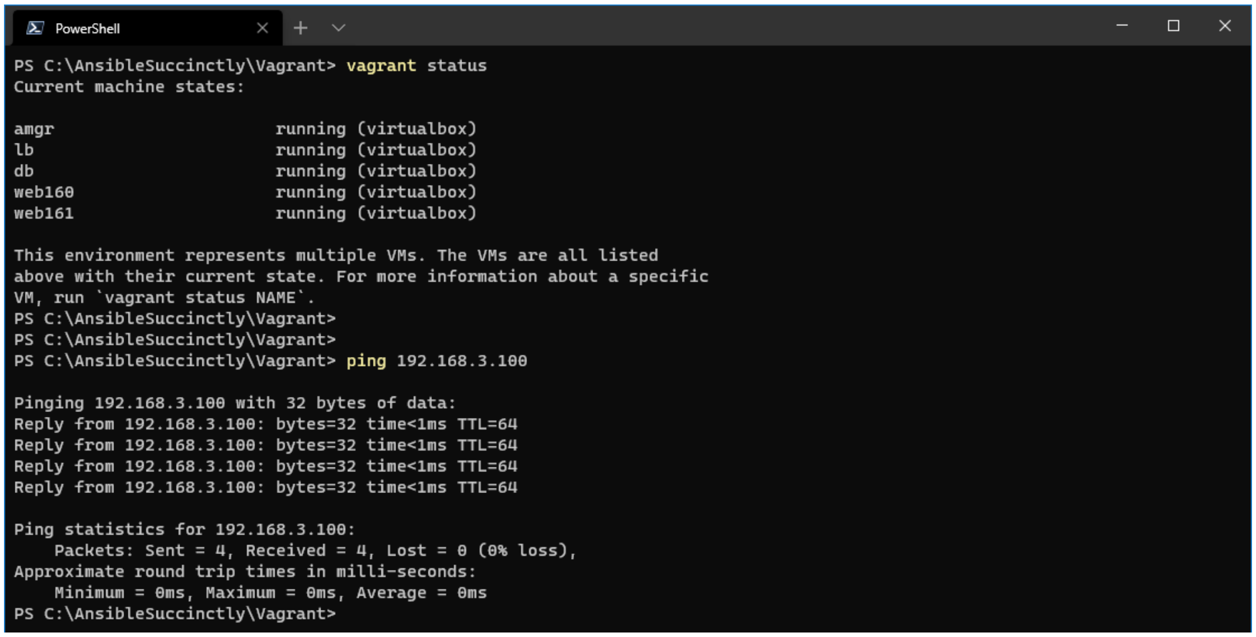
Figure 6: VirtualBox virtual machine status

When you run the status command, it should give you the view on the status of the VMs.

Code Listing 12: Vagrant status checking

```
PS C:\>AnsibleSuccinctly\Vagrant>vagrant status
```

Please note that, in the result, we can also see that we can ping the newly created server, as shown in the following figure.



```
PS C:\AnsibleSuccinctly\Vagrant> vagrant status
Current machine states:

amgr                      running (virtualbox)
lb                        running (virtualbox)
db                        running (virtualbox)
web160                    running (virtualbox)
web161                    running (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
PS C:\AnsibleSuccinctly\Vagrant>
PS C:\AnsibleSuccinctly\Vagrant>
PS C:\AnsibleSuccinctly\Vagrant> ping 192.168.3.100

Pinging 192.168.3.100 with 32 bytes of data:
Reply from 192.168.3.100: bytes=32 time<1ms TTL=64
Reply from 192.168.3.100: bytes=32 time<1ms TTL=64
Reply from 192.168.3.100: bytes=32 time<1ms TTL=64
Reply from 192.168.3.100: bytes=32 time<1ms TTL=64

Ping statistics for 192.168.3.100:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
PS C:\AnsibleSuccinctly\Vagrant>
```

Figure 7: Status of VMs

Tip: At every restart of your PC/workstation, you might need to run the vagrant up command to start the VMs. Unless you destroy and recreate a VM, the state of the VM will remain intact.

Accessing the servers

There are two ways of accessing the servers that you should use.

Table 3: Accessing the VM via ssh

`vagrant ssh <machine-name>`	By using **`vagrant ssh`**, you may easily access any machine created by Vagrant. The following command: **`vagrant ssh amgr`** will automatically log you in to the **amgr** machine without asking for username and password. This command has to be run in the folder where the Vagrantfile is located, in our case C:\AnsibleSuccinctly\Vagrant. Examples:

	• `vagrant ssh web160` • `vagrant ssh db`

`ssh vagrant@<machine_ip>`	This is the standard `ssh` command. The vagrant machines have been set up so that they can be accessed from the Windows 10 client, but only by the IP address. But we have to supply the credentials:
	
	`username = vagrant`
	`password = vagrant`
	
	Examples:
	
	• `ssh vagrant@192.168.3.100`
	• `ssh vagrant@192.168.3.200`

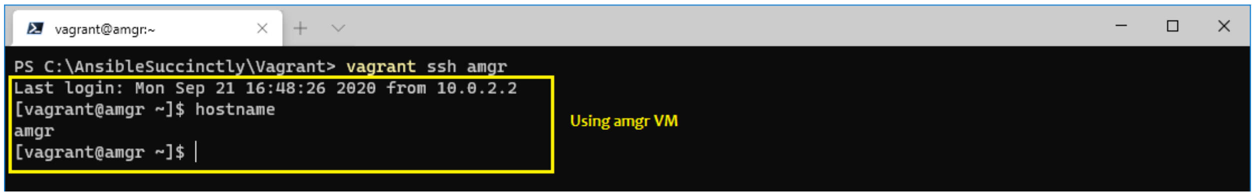
Other Vagrant commands

Vagrant has several commands you should be aware of.

Table 4: Vagrant commands

`vagrant up`	Given the presence of a Vagrantfile with the infrastructure definition, creates or starts the infrastructure. Use this command every time you want to start the VMs.
`vagrant destroy`	It can only be executed in the folder where the Vagrantfile is located. It destroys the VMs created, by literally deleting all of the files and configuration.
`vagrant reload`	Reloads the virtual machine by applying new settings from the Vagrantfile. Useful when you need to change network or synced folder settings.
`vagrant ssh`	It connects to a running Vagrant VM.
`vagrant halt`	Stops (pauses) the virtual machine.

Most of the time, we will be using **`vagrant ssh`**.



```
PS C:\AnsibleSuccinctly\Vagrant> vagrant ssh amgr
Last login: Mon Sep 21 16:48:26 2020 from 10.0.2.2
[vagrant@amgr ~]$ hostname
amgr
[vagrant@amgr ~]$ |
```

Figure 8: Using Vagrant ssh

Tip: Anytime you think you've done something wrong within the VM, you can always destroy it by running *`vagrant destroy <machine_name>`* ***and*** *`vagrant up <machine_name>`*. ***It's a very convenient way to experiment with machines and reset the environment at any time—and all of this within minutes.***

Visual Studio Code: connecting with the manager node

This step is optional, and while you might use **`vi`**, **`vim`**, **`nano`**, or your other favorite editor to edit the files directly on the Linux **`amgr`** node, I personally find it very useful to work from Visual Studio Code while being connected remotely to the **`amgr`** node.

VS Code gives us a very easy way of doing so. After starting up the VM with the **`vagrant up`** command, we can connect remotely to the **`amgr`** node by executing the following steps:

1. Open Visual Studio Code and click the Remote Explorer icon on the left-hand side of the editor.

2. Choose **SSH Targets** from the drop-down menu and click the **+** icon, as shown in Figure 9.

Follow the wizard by entering the following.

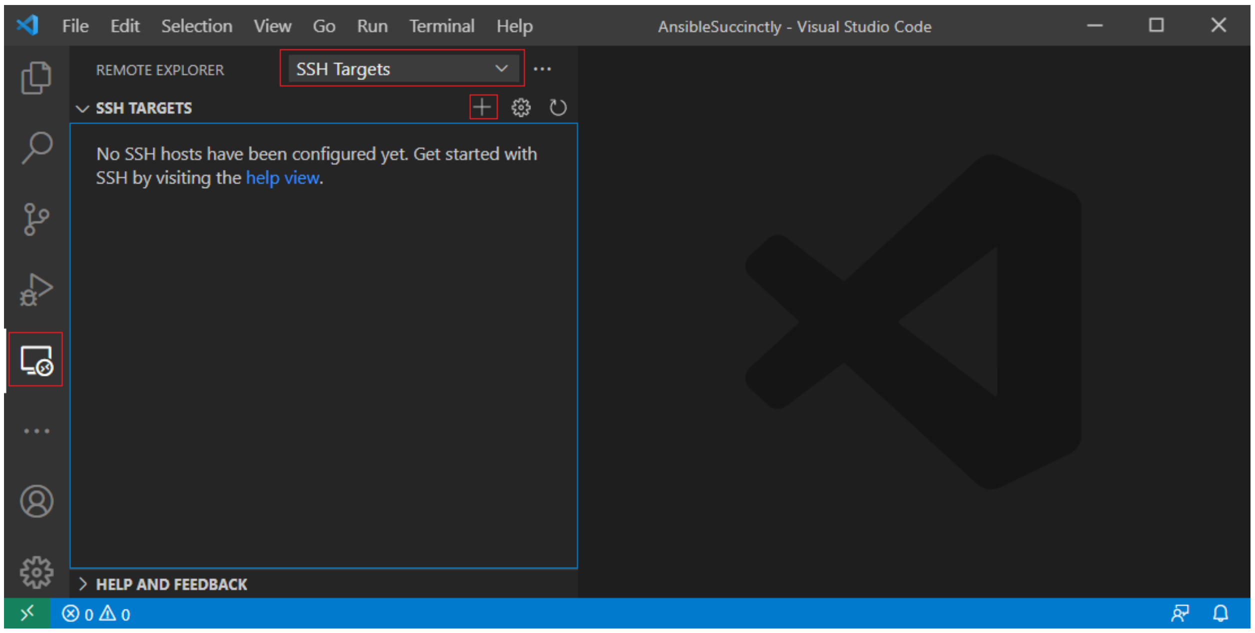
Code Listing 13: Connection settings VS Code

```
ssh vagrant@192.168.3.100
```

When you are prompted with **Select SSH configuration file to update**, choose the folder location as in the Code Listing.

Code Listing 14: Storing configuration settings

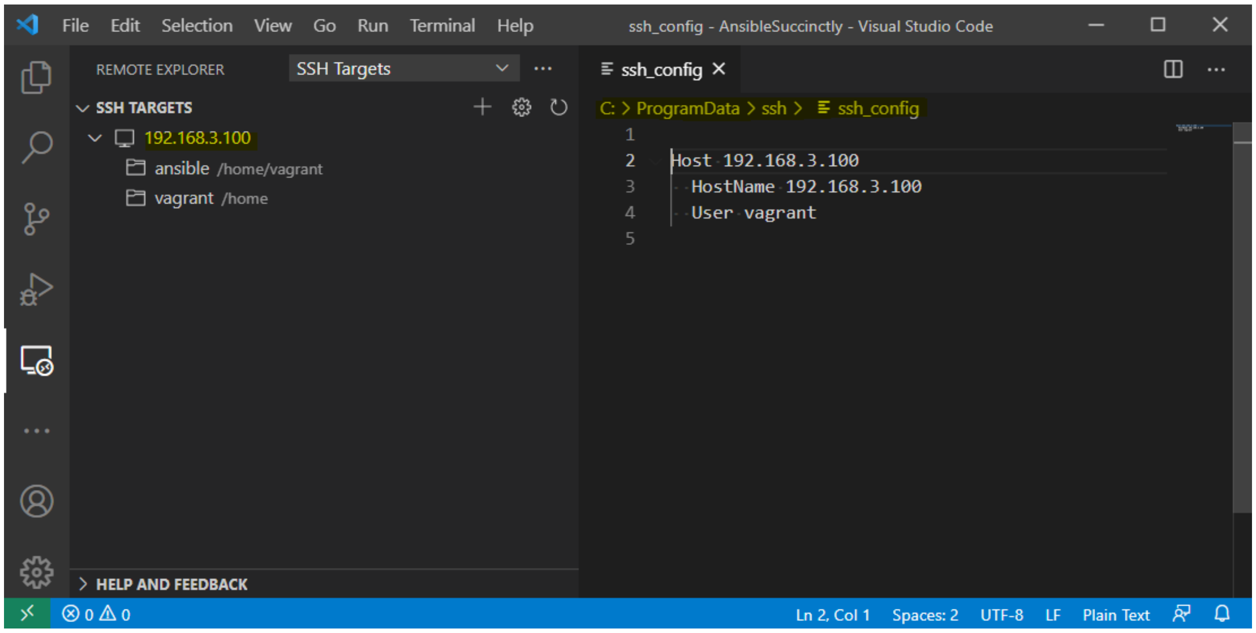
```
C:\ProgramData\ssh\ssh_config
```


File Edit Selection View Go Run Terminal Help
AnsibleSuccinctly - Visual Studio Code
REMOTE EXPLORER
SSH Targets
SSH TARGETS
HELP AND FEEDBACK

No SSH hosts have been configured yet. Get started with SSH by visiting the help view.

Figure 9: Add new remote connection

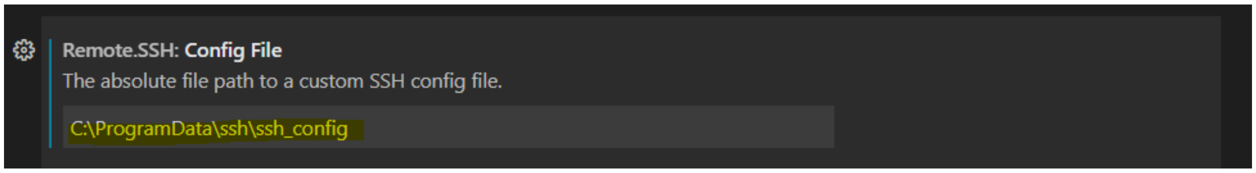
At the end of the process, you should see something like the following.


File Edit Selection View Go Run Terminal Help
ssh_config - AnsibleSuccinctly - Visual Studio Code
REMOTE EXPLORER
SSH Targets
SSH TARGETS
192.168.3.100
ansible /home/vagrant
vagrant /home
HELP AND FEEDBACK
ssh_config
C: > ProgramData > ssh > ssh_config
Ln 2, Col 1 Spaces: 2 UTF-8 LF Plain Text

```

Host 192.168.3.100
  HostName 192.168.3.100
  User vagrant

```

Figure 10: Setting up the remote connection

If not already present, make sure there is a path selected for the config file in the remote connection settings. To do so in VS Code, press **Ctrl+Shift+P** and type Remote-SSH: Settings. When the settings are open, specify the full path to the config file in the **RemoveSSH: Config File** entry, as shown in the following figure.



Remote.SSH: Config File

The absolute file path to a custom SSH config file.

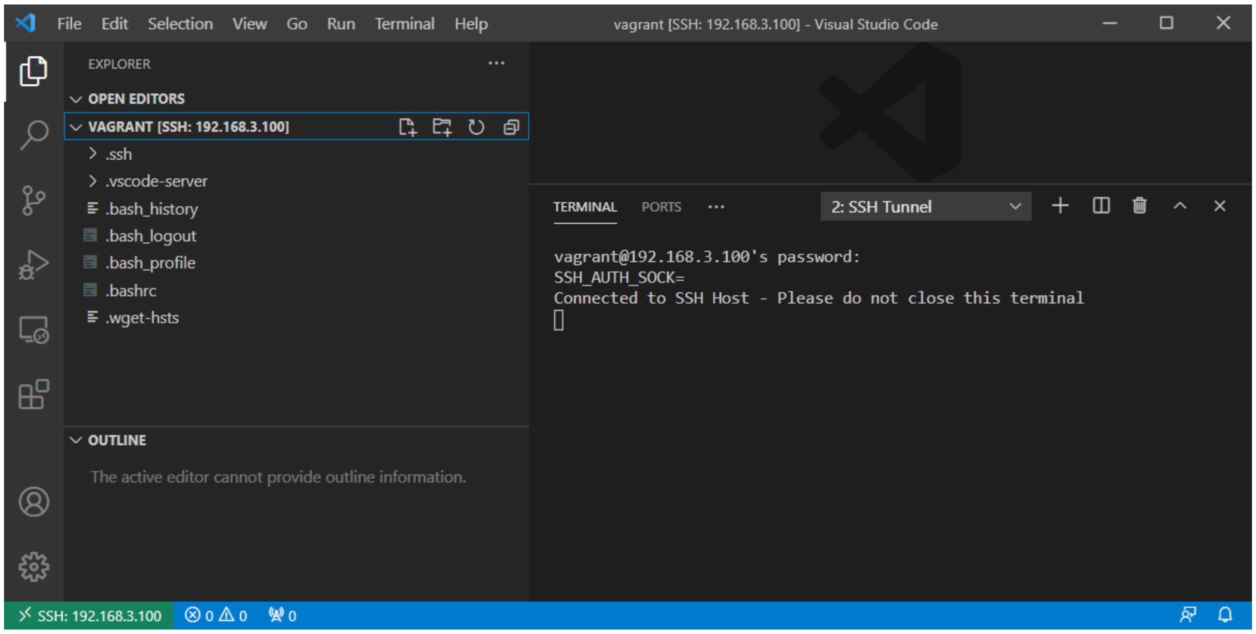
C:\ProgramData\ssh\ssh_config

Figure 11: Remote.SSH: config file setting

On the Remote Explorer, right-click the IP address and choose **Connect to Host in Current Window**. The process of connecting to the remote server starts.

You will be prompted for the password twice, so please enter **vagrant** as the password.

At the end of the process, you should have the result shown in Figure 12. You can open any folder on the server by choosing File >Open Folder.


File Edit Selection View Go Run Terminal Help
vagrant [SSH: 192.168.3.100] - Visual Studio Code
EXPLORER
OPEN EDITORS
VAGRANT [SSH: 192.168.3.100]
.ssh
.vscode-server
.bash_history
.bash_logout
.bash_profile
.bashrc
.wget-hsts
OUTLINE
The active editor cannot provide outline information.
TERMINAL
PORTS
2: SSH Tunnel
SSH: 192.168.3.100

```
vagrant@192.168.3.100's password:
SSH_AUTH_SOCK=
Connected to SSH Host - Please do not close this terminal
```

Figure 12: Opened folder on remote server

Now you can create your own folders, edit files, and execute commands directly from Visual Studio Code.

Note: All the figures in this book are showing examples using the command line. Use what you will find more convenient vagrant ssh, ssh vagrant@192.168.3.100, or VS Code remote connections.

Chapter 4 Installing Ansible

We will be installing the Ansible software on our Ansible Manager server **`amgr`**, which we started up previously. From there, we will be orchestrating the execution of the code. There are two main ways to install and use Ansible on the host:

- Using the operating system package manager (apt-get, yum), depending on the operating system in use.
- Install Ansible by using **`pip`**, which is the Python package manager.

Ansible creates new releases a couple of times per year. Due to this short release cycle, minor bugs will generally be fixed in the next release. Major bugs will still have maintenance releases when needed, though these are not so frequent.

Note: Although you can try both installation options, in this book we will be using the installation through pip.

Using the OS package manager

Installing Ansible by using the OS package manager obviously depends on the operating system (and distribution, in case of Linux). As mentioned previously, we are using CentOS.

Installing Ansible on CentOS

Throughout this book, we are using CentOS as our example Linux distribution. CentOS is a good choice, as it is secure, has a good package-management software, and is well documented. The chance of finding it in an enterprise environment is quite high.

Let's log into the **`amgr`** server by running the following command.

Code Listing 15: Log into the amgr host

```
PS C:\>AnsibleSuccinctly\Vagrant>vagrant ssh amgr

Or alternatively (when asked, provide vagrant as password)

PS C:\>ssh vagrant@192.168.3.100
```

Once logged into the **`amgr`** host, we'll use the Ansible installation command.

Code Listing 16: Installing Ansible command

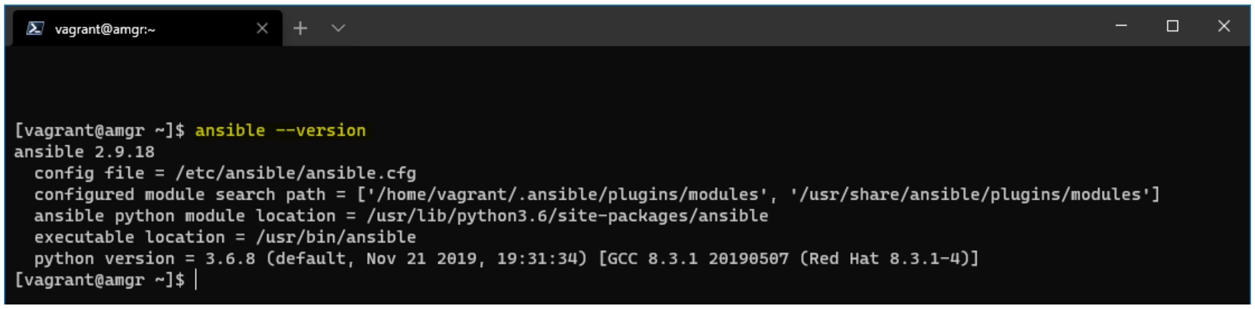
```
[vagrant@amgr ~]$ sudo yum install ansible
```

To check that Ansible has been properly installed, let's run the following command.

Code Listing 17: Checking the Ansible version

```
[vagrant@amgr ~]$ ansible --version
```

The result returned should be similar to what is shown in Figure 13. You can see that the result returned shows a lot of information about the installed software, such as the version (in our case Ansible 2.9.18) and the configuration settings, such as configuration file location and Ansible location.



```
[vagrant@amgr ~]$ ansible --version
ansible 2.9.18
  config file = /etc/ansible/ansible.cfg
  configured module search path = ['/home/vagrant/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3.6/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 3.6.8 (default, Nov 21 2019, 19:31:34) [GCC 8.3.1 20190507 (Red Hat 8.3.1-4)]
[vagrant@amgr ~]$
```

Figure 13: Ansible version information

Installing Ansible by using pip

The standard package manager for Python is **`pip`**. It allows you to install and manage additional packages that are not part of the Python standard library.

If **`pip`** is not already installed (as in our case currently), we should install it first.

Note: If you have followed the previous installation by using the package manager, you can simply destroy the amgr machine and create a new one to have a clean environment [`vagrant destroy amgr`, `vagrant up amgr`].

As we did previously, we should login to the **`amgr`** host first.

```
PS C:\>AnsibleSuccinctly\Vagrant> vagrant ssh amgr
```

Note: Throughout this book, we will be using Ansible installed by using pip and within a virtual environment.

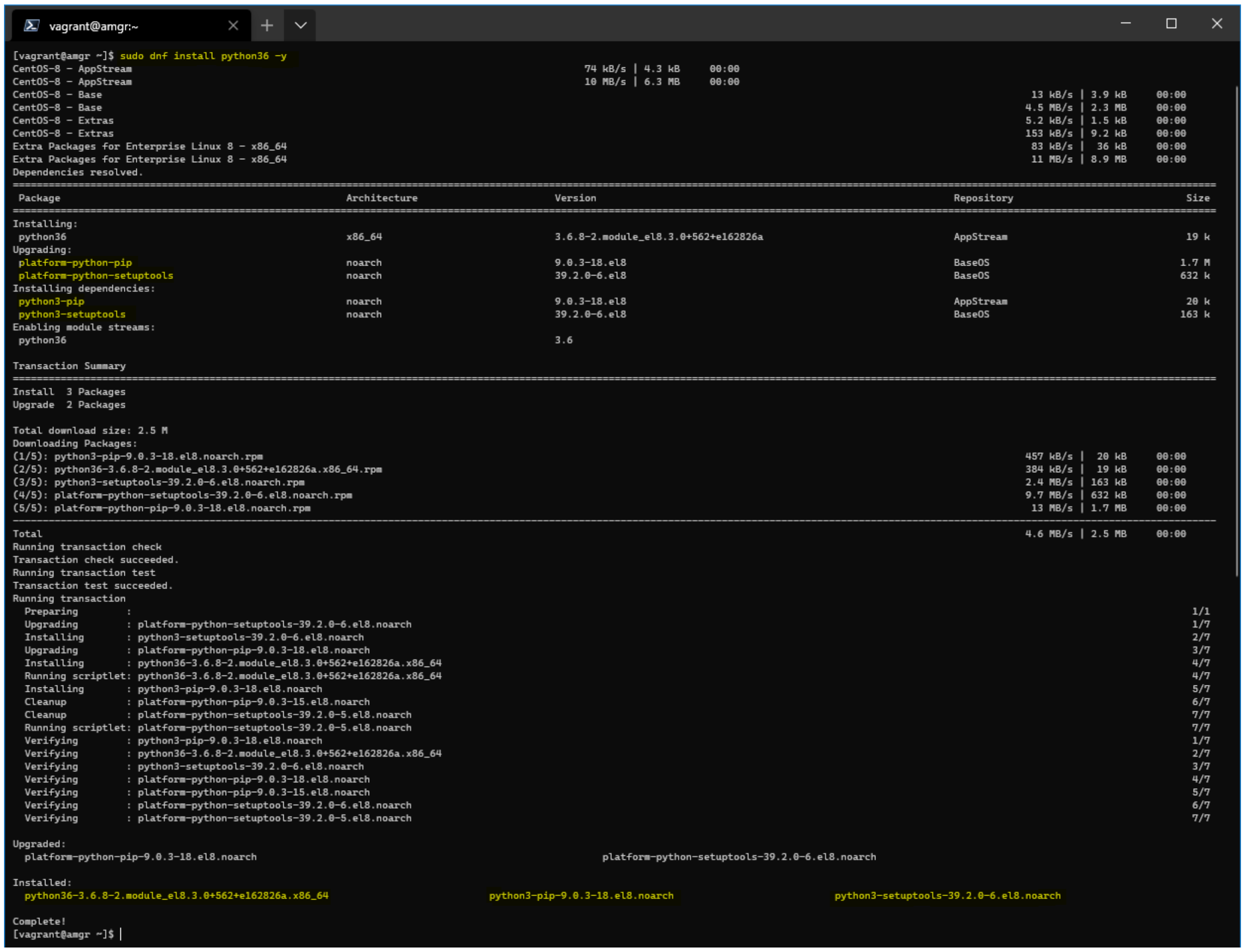
Installing Python and pip

To install **`pip`**, we need to install the Python framework, which already contains the **`pip`** tool as part of its package. Let's run the following command to install Python version 3.6.

Code Listing 18: Installation of Python 3.6

```
[vagrant@amgr ~]$ sudo dnf install python36 -y
```

The output of the command should look like the output shown in Figure 14. We can also see that together with Python 3.6, this command also installs the **pip** and **setuptools** packages.



```
[vagrant@amgr ~]$ sudo dnf install python36 -y
CentOS-8 - AppStream                                        74 kB/s | 4.3 kB     00:00
CentOS-8 - AppStream                                        10 MB/s | 6.3 MB     00:00
CentOS-8 - Base                                                                                              13 kB/s | 3.9 kB     00:00
CentOS-8 - Base                                                                                             4.5 MB/s | 2.3 MB     00:00
CentOS-8 - Extras                                                                                           5.2 kB/s | 1.5 kB     00:00
CentOS-8 - Extras                                                                                           153 kB/s | 9.2 kB     00:00
Extra Packages for Enterprise Linux 8 - x86_64                                                               83 kB/s |  36 kB     00:00
Extra Packages for Enterprise Linux 8 - x86_64                                                               11 MB/s | 8.9 MB     00:00
Dependencies resolved.
=================================================================================================================================
 Package                          Architecture         Version                                       Repository             Size
=================================================================================================================================
Installing:
 python36                         x86_64               3.6.8-2.module_el8.3.0+562+e162826a           AppStream              19 k
Upgrading:
 platform-python-pip              noarch               9.0.3-18.el8                                  BaseOS                1.7 M
 platform-python-setuptools       noarch               39.2.0-6.el8                                  BaseOS                632 k
Installing dependencies:
 python3-pip                      noarch               9.0.3-18.el8                                  AppStream              20 k
 python3-setuptools               noarch               39.2.0-6.el8                                  BaseOS                163 k
Enabling module streams:
 python36                                              3.6

Transaction Summary
=================================================================================================================================
Install  3 Packages
Upgrade  2 Packages

Total download size: 2.5 M
Downloading Packages:
(1/5): python3-pip-9.0.3-18.el8.noarch.rpm                                                                  457 kB/s |  20 kB     00:00
(2/5): python36-3.6.8-2.module_el8.3.0+562+e162826a.x86_64.rpm                                              384 kB/s |  19 kB     00:00
(3/5): python3-setuptools-39.2.0-6.el8.noarch.rpm                                                           2.4 MB/s | 163 kB     00:00
(4/5): platform-python-setuptools-39.2.0-6.el8.noarch.rpm                                                   9.7 MB/s | 632 kB     00:00
(5/5): platform-python-pip-9.0.3-18.el8.noarch.rpm                                                           13 MB/s | 1.7 MB     00:00
---------------------------------------------------------------------------------------------------------------------------------
Total                                                                                                       4.6 MB/s | 2.5 MB     00:00
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :                                                                                                         1/1
  Upgrading        : platform-python-setuptools-39.2.0-6.el8.noarch                                                          1/7
  Installing       : python3-setuptools-39.2.0-6.el8.noarch                                                                  2/7
  Upgrading        : platform-python-pip-9.0.3-18.el8.noarch                                                                 3/7
  Installing       : python36-3.6.8-2.module_el8.3.0+562+e162826a.x86_64                                                     4/7
  Running scriptlet: python36-3.6.8-2.module_el8.3.0+562+e162826a.x86_64                                                     4/7
  Installing       : python3-pip-9.0.3-18.el8.noarch                                                                         5/7
  Cleanup          : platform-python-pip-9.0.3-15.el8.noarch                                                                 6/7
  Cleanup          : platform-python-setuptools-39.2.0-5.el8.noarch                                                          7/7
  Running scriptlet: platform-python-setuptools-39.2.0-5.el8.noarch                                                          7/7
  Verifying        : python3-pip-9.0.3-18.el8.noarch                                                                         1/7
  Verifying        : python36-3.6.8-2.module_el8.3.0+562+e162826a.x86_64                                                     2/7
  Verifying        : python3-setuptools-39.2.0-6.el8.noarch                                                                  3/7
  Verifying        : platform-python-pip-9.0.3-18.el8.noarch                                                                 4/7
  Verifying        : platform-python-pip-9.0.3-15.el8.noarch                                                                 5/7
  Verifying        : platform-python-setuptools-39.2.0-6.el8.noarch                                                          6/7
  Verifying        : platform-python-setuptools-39.2.0-5.el8.noarch                                                          7/7

Upgraded:
  platform-python-pip-9.0.3-18.el8.noarch                          platform-python-setuptools-39.2.0-6.el8.noarch

Installed:
  python36-3.6.8-2.module_el8.3.0+562+e162826a.x86_64          python3-pip-9.0.3-18.el8.noarch          python3-setuptools-39.2.0-6.el8.noarch

Complete!
[vagrant@amgr ~]$ |
```

Figure 14: Installing Python 3.6

Once it’s installed, we will be working in a virtual environment, which we need to create first. This is very useful, as we can play around with packages installed by Python.

We need to create a folder called ansible, where we will keep the code, and afterward we will create and activate the virtual environment called **avenv** (which stands for ansible virtual environment, a random name chosen for convenience).

Code Listing 19: Creating the virtual environment

```
[vagrant@amgr ~]$ mkdir ansible
[vagrant@amgr ~]$ cd ansible
[vagrant@amgr ansible]$ python3 -m venv avenv
[vagrant@amgr ansible]$ source avenv/bin/activate
```

```
(avenv) [vagrant@amgr ansible]$
```

Activating the virtual environment will change your shell's prompt to show what virtual environment you're using, as you can see the **`(avenv)`** in front of the command prompt.

Note: Make sure to activate the environment every time before using Ansible by running *`source avenv/bin/activate`* ***in the ansible folder. To deactivate the virtual environment, just use the*** *`deactivate`* ***command.***

Before installing Ansible, it is good to take the latest version of **`pip`** and **`setuptools`** by running the following command. We will also install a package called **`wheel`**, which will help us install Ansible.

Code Listing 20: Upgrade pip, setuptools, and wheel command

```
(avenv) [vagrant@amgr ansible]$ pip3 install --upgrade pip setuptools wheel
```

Installing Ansible

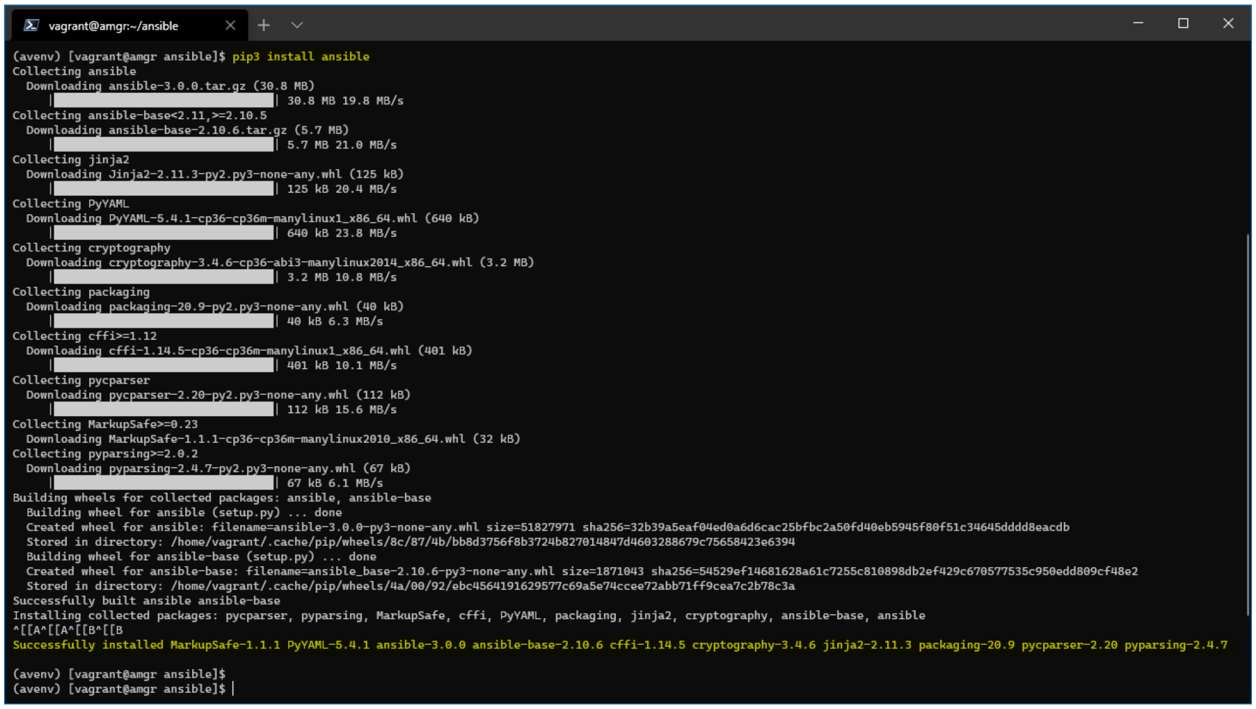
After **`pip`** and its dependencies have been properly installed and updated, we can install Ansible. Ansible is installed by the following command (inside the virtual environment).

Code Listing 21 : Installing Ansible with pip command

```
(avenv) [vagrant@amgr ansible]$ pip3 install ansible
```

Note: Installing Ansible also installs the dependencies such as jinja2, PyYAML, cryptography, and other packages.

The installation procedure is shown in the following figure.



```
(avenv) [vagrant@amgr ansible]$ pip3 install ansible
Collecting ansible
  Downloading ansible-3.0.0.tar.gz (30.8 MB)
     |████████████████████████████████| 30.8 MB 19.8 MB/s
Collecting ansible-base<2.11,>=2.10.5
  Downloading ansible-base-2.10.6.tar.gz (5.7 MB)
     |████████████████████████████████| 5.7 MB 21.0 MB/s
Collecting jinja2
  Downloading Jinja2-2.11.3-py2.py3-none-any.whl (125 kB)
     |████████████████████████████████| 125 kB 20.4 MB/s
Collecting PyYAML
  Downloading PyYAML-5.4.1-cp36-cp36m-manylinux1_x86_64.whl (640 kB)
     |████████████████████████████████| 640 kB 23.8 MB/s
Collecting cryptography
  Downloading cryptography-3.4.6-cp36-abi3-manylinux2014_x86_64.whl (3.2 MB)
     |████████████████████████████████| 3.2 MB 10.8 MB/s
Collecting packaging
  Downloading packaging-20.9-py2.py3-none-any.whl (40 kB)
     |████████████████████████████████| 40 kB 6.3 MB/s
Collecting cffi>=1.12
  Downloading cffi-1.14.5-cp36-cp36m-manylinux1_x86_64.whl (401 kB)
     |████████████████████████████████| 401 kB 10.1 MB/s
Collecting pycparser
  Downloading pycparser-2.20-py2.py3-none-any.whl (112 kB)
     |████████████████████████████████| 112 kB 15.6 MB/s
Collecting MarkupSafe>=0.23
  Downloading MarkupSafe-1.1.1-cp36-cp36m-manylinux2010_x86_64.whl (32 kB)
Collecting pyparsing>=2.0.2
  Downloading pyparsing-2.4.7-py2.py3-none-any.whl (67 kB)
     |████████████████████████████████| 67 kB 6.1 MB/s
Building wheels for collected packages: ansible, ansible-base
  Building wheel for ansible (setup.py) ... done
  Created wheel for ansible: filename=ansible-3.0.0-py3-none-any.whl size=51827971 sha256=32b39a5eaf04ed0a6d6cac25bfbc2a50fd40eb5945f80f51c34645dddd8eacdb
  Stored in directory: /home/vagrant/.cache/pip/wheels/8c/87/4b/bb8d3756f8b3724b827014847d4603288679c75658423e6394
  Building wheel for ansible-base (setup.py) ... done
  Created wheel for ansible-base: filename=ansible_base-2.10.6-py3-none-any.whl size=1871043 sha256=54529ef14681628a61c7255c810898db2ef429c670577535c950edd809cf48e2
  Stored in directory: /home/vagrant/.cache/pip/wheels/4a/00/92/ebc4564191629577c69a5e74ccee72abb71ff9cea7c2b78c3a
Successfully built ansible ansible-base
Installing collected packages: pycparser, pyparsing, MarkupSafe, cffi, PyYAML, packaging, jinja2, cryptography, ansible-base, ansible
^[[A^[[A^[[B^[[B
Successfully installed MarkupSafe-1.1.1 PyYAML-5.4.1 ansible-3.0.0 ansible-base-2.10.6 cffi-1.14.5 cryptography-3.4.6 jinja2-2.11.3 packaging-20.9 pycparser-2.20 pyparsing-2.4.7

(avenv) [vagrant@amgr ansible]$
(avenv) [vagrant@amgr ansible]$ |
```

Figure 15: Installation procedure of Ansible using pip

When you wish to update Ansible, we can first check whether there is a newer version by using the following command.

Code Listing 22: Check for a newer version

```
(avenv) [vagrant@amgr ~]$ pip3 list --outdated
```

We can update to a newer version, if it exists.

Code Listing 23: Update Ansible command

```
(avenv) [vagrant@amgr ~]$ pip3 install -U ansible
```

The version of Ansible used in the examples in this book is 2.10.8.

Chapter 5 Ansible Configuration

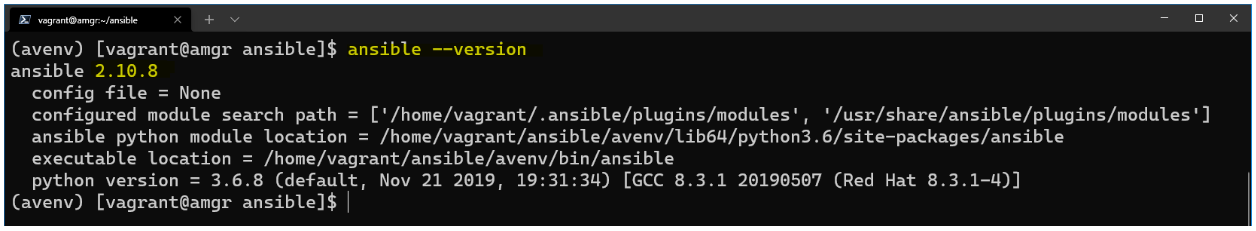
Certain settings in Ansible are adjustable via a configuration file (**ansible.cfg**). This file contains all of the Ansible configuration.

Installing Ansible via the **`pip`** tool doesn't create such a file. To verify this, we can run the **`version`** command as follows.

Code Listing 24: Ansible.cfg location

```
(avenv) [vagrant@amgr ~]$ ansible --version
```

You will get a similar output to the one shown in Figure 16, and you might notice that the config file setting is set to **None**. This means that when we install Ansible via the **`pip`** command, the ansible.cfg file is not set up by default. In the next chapter, I will provide more information about how to deal with the ansible.cfg file.



```
(avenv) [vagrant@amgr ansible]$ ansible --version
ansible 2.10.8
  config file = None
  configured module search path = ['/home/vagrant/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /home/vagrant/ansible/avenv/lib64/python3.6/site-packages/ansible
  executable location = /home/vagrant/ansible/avenv/bin/ansible
  python version = 3.6.8 (default, Nov 21 2019, 19:31:34) [GCC 8.3.1 20190507 (Red Hat 8.3.1-4)]
(avenv) [vagrant@amgr ansible]$
```

Figure 16: Ansible version information

Ansible will look at the configuration file in a variety of locations, with the following priority, and use the first file found (all others are ignored).

Table 5

`ANSIBLE_CONFIG`	If there is an environment variable **`ANSIBLE_CONFIG`** set up to point to any file on the file system. In the **`ANSIBLE_CONFIG`** we can specify the file location directly anywhere in the system before running Ansible, for example: **`export ANSIBLE_CONFIG=/path_to_file/ansible.cfg`**
`./ansible.cfg`	The . represents the current directory.
`~/.ansible.cfg`	~ is a shortcut for the user's home directory. In our case, this would be the vagrant user under its home directory.

	/home/vagrant.
`/etc/ansible/ansible.cfg`	File in a system location, globally defined.

Note: Throughout this book, the ansible.cfg file will be always placed in the "current folder" ./ansible.cfg. In this way, we can have different folders with different ansible.cfg files.

To find out all of the currently configured options, the **`ansible-config`** utility can help us.

Code Listing 25: Check current Ansible configuration

```
(avenv) [vagrant@amgr ~]$ ansible-config list
```

The output of this command is going to be a long list of configuration items with explanations.

In case the ansible.cfg file is present, we can see its content by running the following command.

Code Listing 26: Displaying the current config file

```
(avenv) [vagrant@amgr ~]$ ansible-config view
```

An error will be returned if the file is not present at any of the aforementioned locations.

An example of a complete ansible.cfg file can be found on the Ansible GitHub account.

This file can be copied in the current folder, and its default parameters could be changed. Alternatively, we can have an empty ansible.cfg file and simply set the values we want to override.

Another very useful command option is **`--only-changed`**, which will show only the values that we have potentially overwritten, and which are different from the default ones.

Code Listing 27: Displaying changes in ansible.cfg

```
(avenv) [vagrant@amgr ~]$ ansible-config dump --only-changed
```

There are quite few sections in the ansible.cfg file with subconfiguration keys.

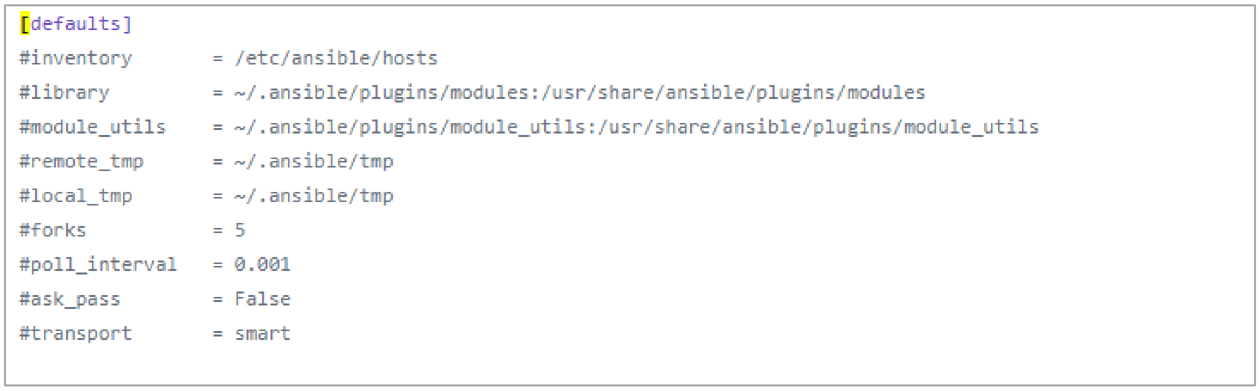
Table 6: Ansible.cfg configuration sections

`[defaults]`	Default generic Ansible settings.
`[inventory]`	Dynamic inventory plugin settings.
`[privilege_escalation]`	Ansible can use existing privilege escalation systems to allow a user to execute tasks as another user.

`[paramiko_connection]`	Paramiko is the default SSH connection implementation on Enterprise Linux 6 or earlier and is not used by default on other platforms.
`[ssh_connection]`	OpenSSH specific settings.
`[persistent_connection]`	When communicating with a remote device, you have control over how long Ansible maintains the connection to that device, as well as how long Ansible waits for a command to complete on that device.
`[sudo_become_plugin]`	User to be used as the sudo.
`[colors]`	Colors in the editor when running commands.
`[galaxy]`	Galaxy platform-specific settings.

The [defaults] section

Among other sections, the ansible.cfg file has a [defaults] section, which contains the basic configuration information, such as where to locate the inventory file, which user to use when connecting to remote hosts, and the remote host port.



```
[defaults]
#inventory        = /etc/ansible/hosts
#library          = ~/.ansible/plugins/modules:/usr/share/ansible/plugins/modules
#module_utils     = ~/.ansible/plugins/module_utils:/usr/share/ansible/plugins/module_utils
#remote_tmp       = ~/.ansible/tmp
#local_tmp        = ~/.ansible/tmp
#forks            = 5
#poll_interval    = 0.001
#ask_pass         = False
#transport        = smart
```

Figure 17: Snippet of a [defaults] section

Chapter 6 Ansible Inventory

The *inventory* or host configuration file is a file that defines the hosts or groups of hosts upon which commands, modules, and tasks in an Ansible Playbook will operate. In other words, it defines a list of systems that we wish to manage with Ansible.

Typically, this file is located in the /etc/ansible directory if Ansible is installed with the default Linux package manager. This file is not provided by the **`pip`** installation, so it has to be created manually.

Some important facts about the inventory file:

- The inventory file defines a collection of hosts that are target systems of the automation.
- The inventory file contains a list of hosts that can be grouped together into groups. One host can be part of multiple groups. For example, we could group hosts into webservers, databases, load balancers, and so on.
- Groups can be grouped together and managed collectively.
- The inventory file contains variables that could apply to either hosts or groups.
- The inventory file is a file that can be written in INI-style or YAML-style formats.
- It is possible to create an inventory file in a *dynamic* way, but this is outside of the scope of this book.

Inventory location

As mentioned previously, the file has a default location; however, we can create our own local version, just to be used by our project. Wherever this location is, it is controlled by the **ansible.cfg** file, which specifies the location of the inventory file that can be either local (relative), as in the following example, or absolute.

Code Listing 28: Inventory file location in ansible.cfg

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

Inventory file content

The inventory file format can be either INI- or YAML-based. It contains a list of hosts that can be specified as IP addresses, as qualified domain names, or both.

Code Listing 29: Inventory file in its simplest form

```
mail.example.com
192.168.3.100
web.mydomain.local
```

Host groups

We can organize the list of hosts in an intelligent way, so that whenever we want to apply some changes, those changes get applied in one go to several hosts belonging to the group.

The following example shows how we can define three arbitrary groups: **`webservers`**, **`databases`**, and **`production`**. Under each of them we can see that one server (**`web1.domain.com`**) can make part of two different groups.

Code Listing 30: Inventory file with groups defined

```
[webservers]
web1.domain.com
web2.domain.com
192.168.3.1

[databases]
db.domain.com

[production]
web1.domain.com
```

There are two groups that are defined by default in Ansible:

- **All**: Contains every host as defined in the inventory file.
- **Ungrouped**: Contains all hosts that don't have another group defined (aside from all).

This implies that every host will belong to at least one of the two groups.

Nested groups

It's also possible to "group the groups." This is achieved by appending **`:children`** to the group definition. We can define the list of hosts in Europe by putting together the list of Italian and Swiss hosts.

Code Listing 31: Inventory file with nested groups

```
[italy]
web1.domain.it
web2.domain.it
[switzerland]
web1.domain.ch
[europe:children]
italy
switzerland
```

Host ranges

It is also possible to define host ranges when defining hosts, in case we have a repetitive and large list of servers that otherwise would be cumbersome to handle manually.

Range is typically defined by **`[START:END]`**, and it can contain letters or numbers.

Code Listing 32: Range of host names

```
web[1:2].domain.com
[a:c].domain.com
192.168.3.[1:200]
192.168.[2:3].[1:200]
```

- The first case defines two web servers starting with **`web1.domain.com`** and **`web2.domain.com`**.
- The second case defines the list, such as: **`a.domain.com`**, **`b.domain.com`**, **`c.domain.com`**.
- The third case defines the servers in a range from **`192.168.3.1`**, **`192.168.3.2`** until **`192.168.3.200`**.
- The fourth example defines the range of:
    - **`192.168.2.1`**, **`192.168.2.2`**, … until **`192.168.2.200`**
    - **`192.168.3.1`**, **`192.168.3.2`**, … until **`192.168.3.200`**

There is also a possibility to create an *alias* (such as **`WS1`**), in case we have only the IP addresses. This is quite useful when displaying the information about the machine while executing commands, as the IP addresses might not give us enough information, especially if we have a lot of machines to manage.

Code Listing 33: Define an alias in the inventory file

```
some_server ansible_port=5555 ansible_host=192.0.2.5
```

Why it is it important to group the hosts together? This is mainly because it's more convenient to launch a command against a group (or **`all`**) of servers rather than doing it one by one, which would defeat the reason for having the inventory file altogether.

Host verification

Ansible offers the `ansible-inventory` command line tool, which is used to display or dump the configured inventory files as Ansible sees it. By default, it produces an output in JSON format, but it can also produce a YAML file, which is useful if we like to convert the format from INI-style to YAML.

Let's quickly check the command by creating a folder on the **`amgr`** node and naming it **chapter_6**. Let's also create two files: the inventory file with the content (Code Listing 34), and the ansible.cfg (Code Listing 35).

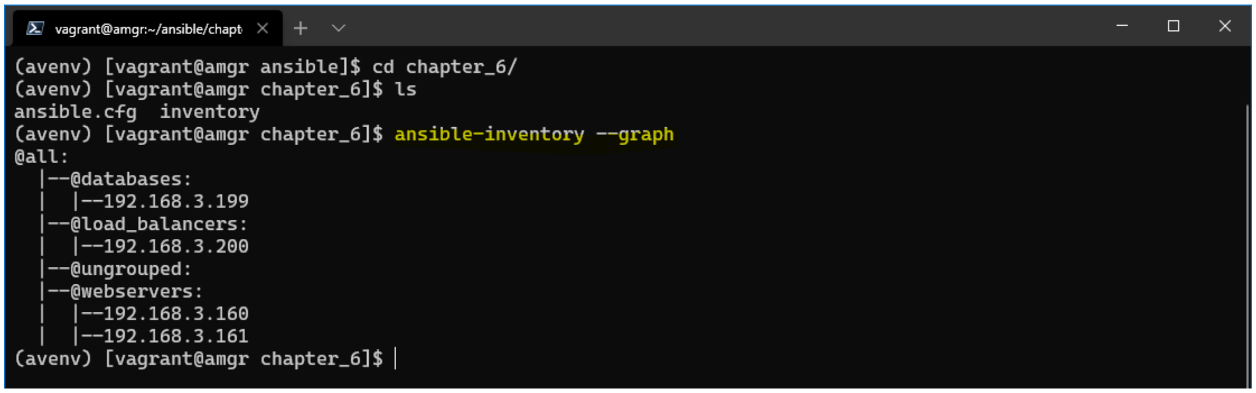
Code Listing 34: Inventory file

```
[webservers]
192.168.3.160
192.168.3.161

[load_balancers]
192.168.3.200

[databases]
192.168.3.199
```

Code Listing 35: ansible.cfg with inventory file specified

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

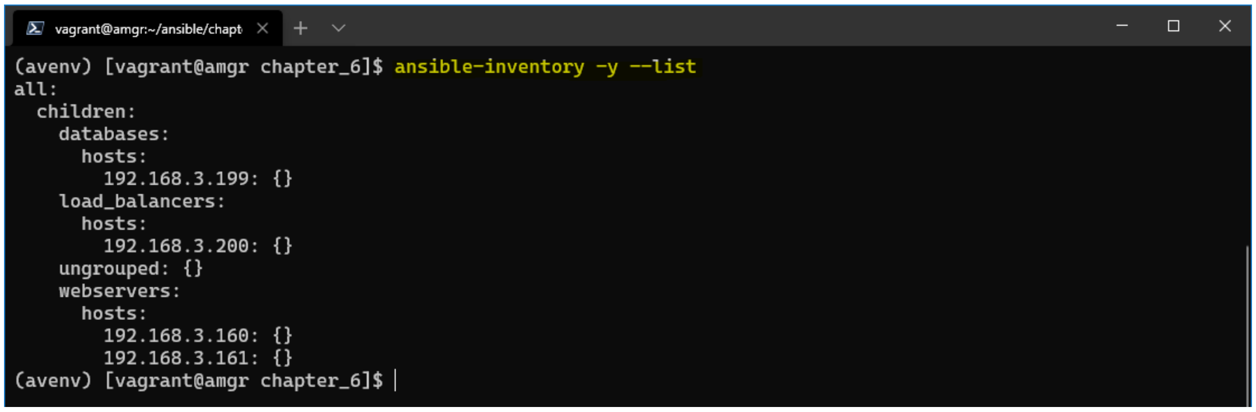
By running the **`ansible-inventory`** command, we can get a list of hosts as a graph.



```
(avenv) [vagrant@amgr ansible]$ cd chapter_6/
(avenv) [vagrant@amgr chapter_6]$ ls
ansible.cfg  inventory
(avenv) [vagrant@amgr chapter_6]$ ansible-inventory --graph
@all:
  |--@databases:
  |  |--192.168.3.199
  |--@load_balancers:
  |  |--192.168.3.200
  |--@ungrouped:
  |--@webservers:
  |  |--192.168.3.160
  |  |--192.168.3.161
(avenv) [vagrant@amgr chapter_6]$
```

Figure 18: Inventory content shown as graph

We can get a list of hosts converted into **`YAML`** format by supplying the **`-y`** argument and **`-list`**.



```
(avenv) [vagrant@amgr chapter_6]$ ansible-inventory -y --list
all:
  children:
    databases:
      hosts:
        192.168.3.199: {}
    load_balancers:
      hosts:
        192.168.3.200: {}
    ungrouped: {}
    webservers:
      hosts:
        192.168.3.160: {}
        192.168.3.161: {}
(avenv) [vagrant@amgr chapter_6]$
```

Figure 19: Inventory file shown as YAML

Dynamic inventories

It’s outside the scope of this book to discuss dynamic inventories, but it’s worth mentioning that there is such a possibility.

Dynamic inventories are particularly important in cases where the infrastructure is not predefined, or it might change overtime.

Ansible supports this scenario either through *inventory plugins*, which would then integrate with the data providers (cloud, LDAP), or by predefined, custom scripts that are custom built.

You can find more information by consulting the Ansible documentation.

Chapter 7 Connecting to Remote Environments

In order to execute any code, Ansible has to first connect to the host that is targeted for changes. That host could be of various types, such as Linux, Windows, or Kubernetes.

The beauty of Ansible is the pluggable architecture and its support for various environments. As shown in the following picture, there are various methods that Ansible supports given the targeted host type.

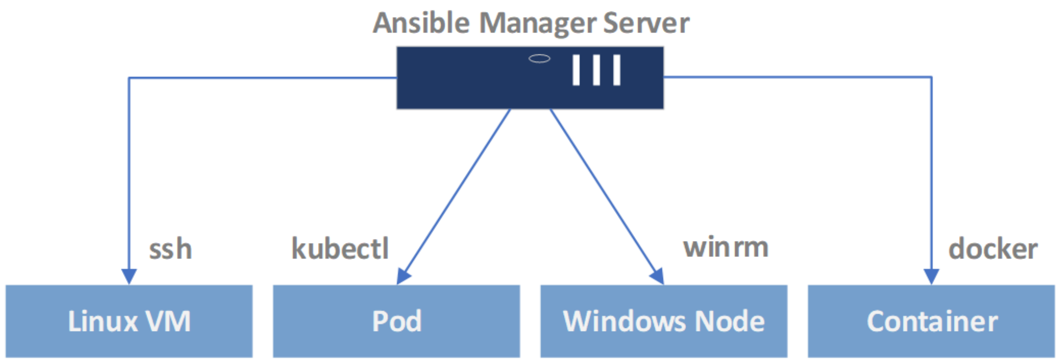


Figure 20: Ansible pluggable architecture

When it comes to connecting to the Linux hosts, the typical way is to use the OpenSSH (**`ssh`**). For connecting to Microsoft Windows hosts, the default is the **`winrm`** technology supported by Microsoft Windows, and so on.

Ansible supports other plugins, as well; you can find a full list here.

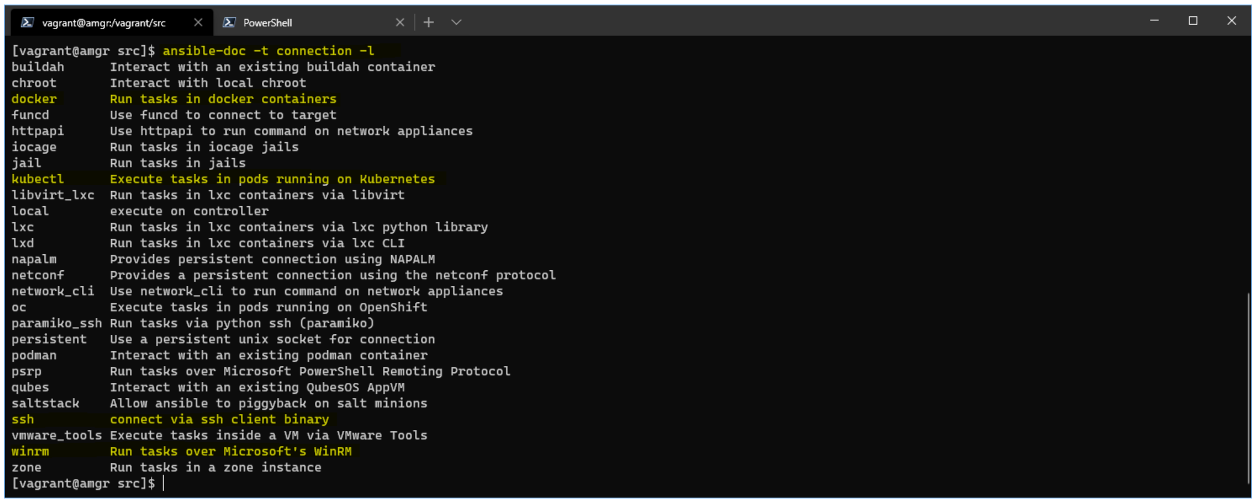
You can find the list of available and installed plugins by running the following command.

Code Listing 36: Find the list of connection plugins

```
(avenv) [vagrant@amgr ~]$ ansible-doc -t connection -l
```

The result shows the list of plugins currently available, as seen in Figure 21.

We won't go into the details of those connection types other than **`ssh`**, as this is the default mechanism used in Linux, and since all of our machines are Linux-based, this would be the context of this book.



```
[vagrant@amgr src]$ ansible-doc -t connection -l
buildah      Interact with an existing buildah container
chroot       Interact with local chroot
docker       Run tasks in docker containers
funcd        Use funcd to connect to target
httpapi      Use httpapi to run command on network appliances
iocage       Run tasks in iocage jails
jail         Run tasks in jails
kubectl      Execute tasks in pods running on Kubernetes
libvirt_lxc  Run tasks in lxc containers via libvirt
local        execute on controller
lxc          Run tasks in lxc containers via lxc python library
lxd          Run tasks in lxc containers via lxc CLI
napalm       Provides persistent connection using NAPALM
netconf      Provides a persistent connection using the netconf protocol
network_cli  Use network_cli to run command on network appliances
oc           Execute tasks in pods running on OpenShift
paramiko_ssh Run tasks via python ssh (paramiko)
persistent   Use a persistent unix socket for connection
podman       Interact with an existing podman container
psrp         Run tasks over Microsoft PowerShell Remoting Protocol
qubes        Interact with an existing QubesOS AppVM
saltstack    Allow ansible to piggyback on salt minions
ssh          connect via ssh client binary
vmware_tools Execute tasks inside a VM via VMware Tools
winrm        Run tasks over Microsoft's WinRM
zone         Run tasks in a zone instance
[vagrant@amgr src]$
```

Figure 21: Ansible connection plugins result

Ansible Manager Server configuration

One of the most common ways of connecting to remote hosts in Linux is to use **`ssh`**. When using Ansible, it's recommended to use the **`ssh`** key-based authentication.

To run the commands, Ansible should be using the unprivileged account that can use **`sudo`** to become **`root`** without supplying a password. Requiring a password during the command running can be cumbersome, as the operator needs manual interaction.

If we look at the ansible.cfg **`[privilege_escalation]`** section, we can see that Ansible by default is configured to support what I just described (the **`#`** in front of the key means that this setting is just commented out, making it de facto a default value).

Code Listing 37: Privilege escalation defaults

```
[privilege_escalation]
#become=True
#become_method=sudo
#become_user=root
#become_ask_pass=False
```

However, these settings can be also placed in the playbook to override the default settings, as we are going to see later.

Note: By default, Ansible will connect to the other host by using the user executing commands.

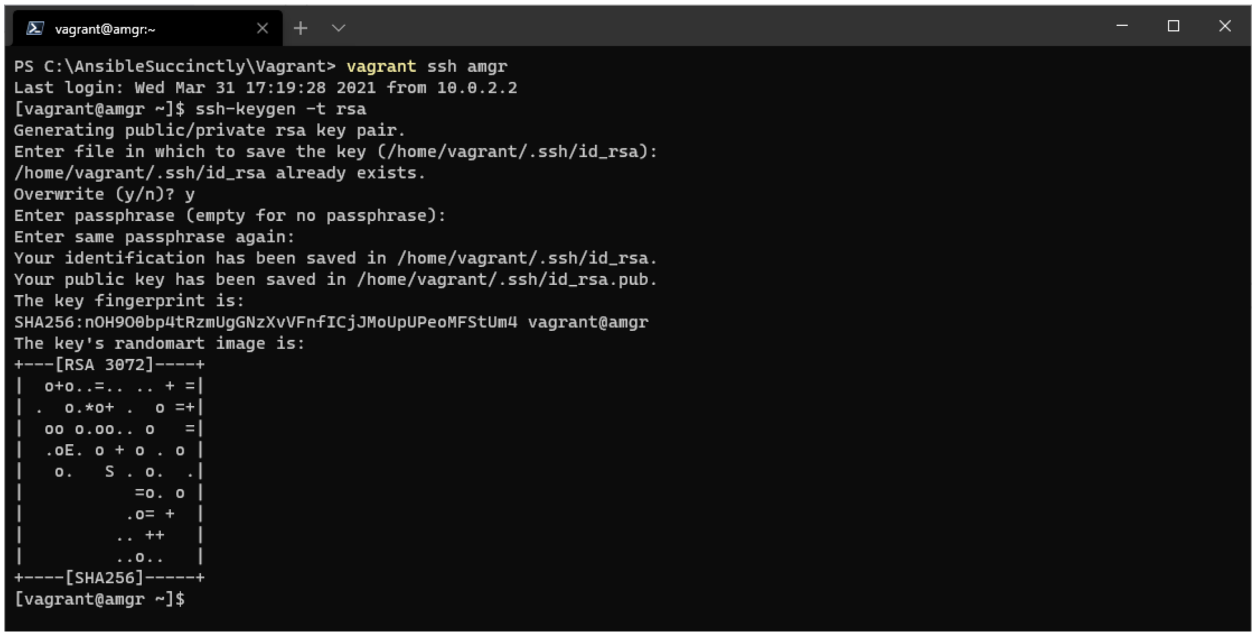
Generation of the ssh key

Before using Ansible without providing the username and password, we have to create the **`ssh`** key (as a file) and push this file to all the hosts we'd like to manage.

First, log in to the **`amgr`** server, and at the shell prompt, type the following command.

```
[vagrant@amgr ~]$ ssh-keygen -t rsa
```

You will be prompted for the location of the key file, and you can just keep the default values as supplied in the command prompt. Please do override the **`id_rsa`** file.



```
PS C:\AnsibleSuccinctly\Vagrant> vagrant ssh amgr
Last login: Wed Mar 31 17:19:28 2021 from 10.0.2.2
[vagrant@amgr ~]$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/vagrant/.ssh/id_rsa):
/home/vagrant/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/vagrant/.ssh/id_rsa.
Your public key has been saved in /home/vagrant/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:nOH9O0bp4tRzmUgGNzXvVFnfICjJMoUpUPeoMFStUm4 vagrant@amgr
The key's randomart image is:
+---[RSA 3072]----+
|  o+o..=.. .. + =|
| .  o.*o+ .  o =+|
|  oo o.oo.. o   =|
|  .oE. o + o . o |
|   o.   S . o.  .|
|           =o. o |
|          .o= +  |
|         .. ++   |
|         ..o..   |
+----[SHA256]-----+
[vagrant@amgr ~]$
```

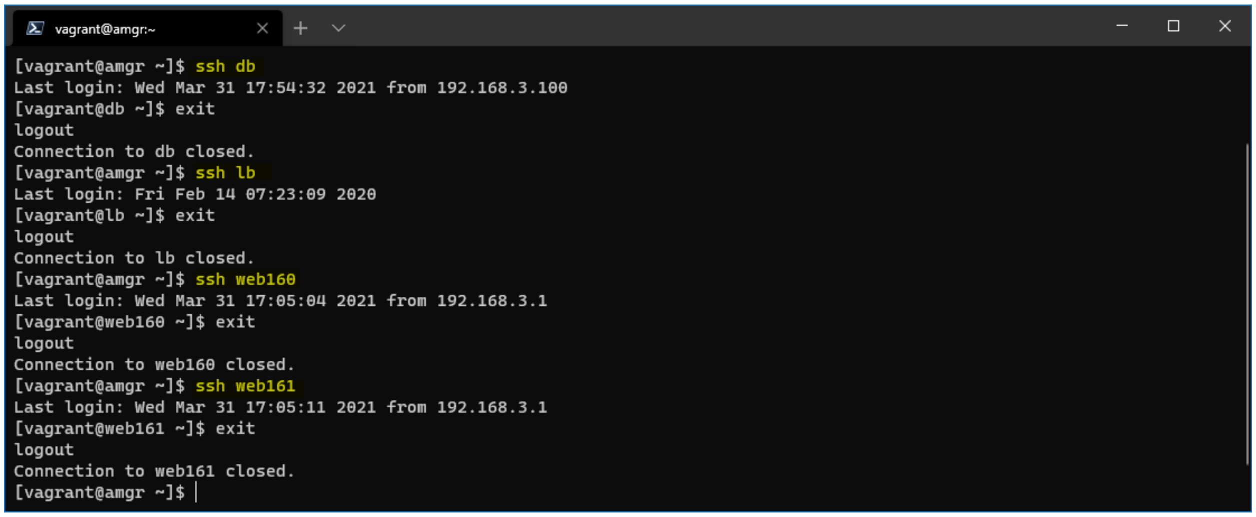
Figure 22: Overriding id_rsa

Once the **`ssh`** key is generated, we need to copy it to all the hosts. We can accomplish this by running the following command. You will be asked to provide a password for each server. Hopefully this is just a one-time operation.

Code Listing 38: Copy ssh key to remote hosts

```
[vagrant@amgr ~]$ hosts="web160 web161 db lb"
[vagrant@amgr ~]$ for host in $hosts; do
> ssh-copy-id -i ~/.ssh/id_rsa.pub vagrant@$host -o
StrictHostKeyChecking=no -f
> done
```

You can see that now we can log into the servers without supplying any username and password, as shown in the following figure.


vagrant@amgr:~

```
[vagrant@amgr ~]$ ssh db
Last login: Wed Mar 31 17:54:32 2021 from 192.168.3.100
[vagrant@db ~]$ exit
logout
Connection to db closed.
[vagrant@amgr ~]$ ssh lb
Last login: Fri Feb 14 07:23:09 2020
[vagrant@lb ~]$ exit
logout
Connection to lb closed.
[vagrant@amgr ~]$ ssh web160
Last login: Wed Mar 31 17:05:04 2021 from 192.168.3.1
[vagrant@web160 ~]$ exit
logout
Connection to web160 closed.
[vagrant@amgr ~]$ ssh web161
Last login: Wed Mar 31 17:05:11 2021 from 192.168.3.1
[vagrant@web161 ~]$ exit
logout
Connection to web161 closed.
[vagrant@amgr ~]$ 
```

Figure 23: No password required

Chapter 8 Running Ad-Hoc Commands

One of the easiest ways to interact with the hosts defined in the inventory is to run *ad-hoc commands*. An ad-hoc command is a one-time, quick, and easy operation—but in general, not reusable.

There are several tasks that are rarely repeated, so in that case, it's not that efficient to create playbooks. These can be whatever comes to mind, from rebooting servers or simply checking that the servers are up by using the **`ping`** command.

To run Ansible ad-hoc commands, we will directly use the **`ansible`** command line tool. The command looks like the following.

Code Listing 39: Ad-hoc command template

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

- **`Pattern`** defines the specific host or group of hosts as defined in the inventory against which we'd like to run the command.
- **`Module`** defines the command to be executed. This can be a simple **`ping`** command or something way more complex. Executing modules is *idempotent* (they only make changes if the change is needed). We are going to see a few examples of commands to understand this better.

You can retrieve the full list of available commands by running the following command.

Code Listing 40: Retrieving all modules

```
$ ansible-doc -l
```

There is a large amount of information retrieved; therefore, you can simply filter out the commands and retrieve what you need by using the **`grep`** command.

Code Listing 41: Retrieving all modules filtering for ping

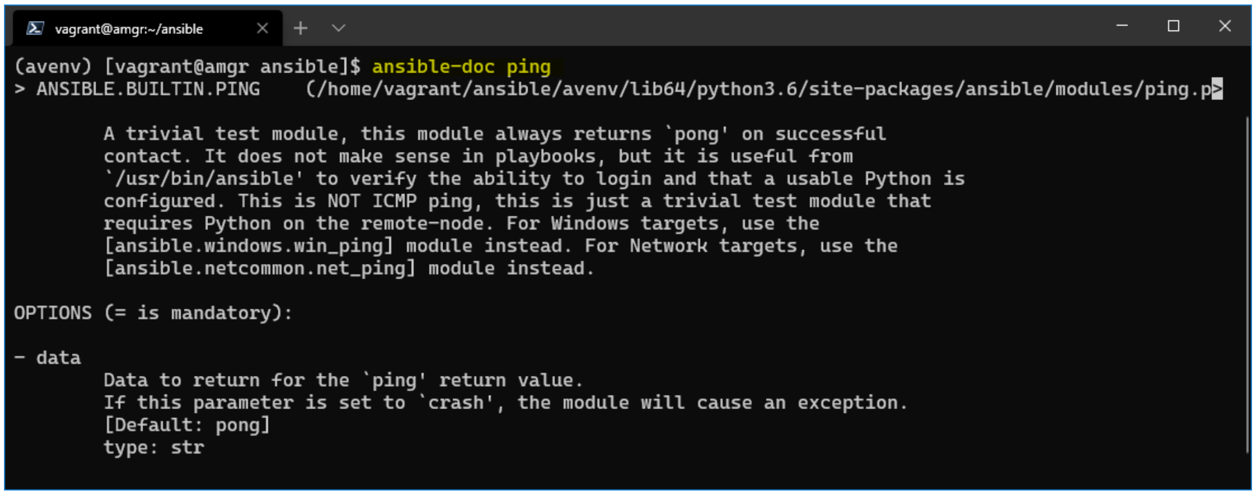
```
$ ansible-doc -l | grep ping
```

Documentation for a particular command is also available, and we can look for it by using the following command.

Code Listing 42: Retrieving ping documentation

```
$ ansible-doc ping
```

The result is shown in Figure 24.



```
vagrant@amgr:~/ansible
(avenv) [vagrant@amgr ansible]$ ansible-doc ping
> ANSIBLE.BUILTIN.PING    (/home/vagrant/ansible/avenv/lib64/python3.6/site-packages/ansible/modules/ping.p

        A trivial test module, this module always returns `pong' on successful
        contact. It does not make sense in playbooks, but it is useful from
        `/usr/bin/ansible' to verify the ability to login and that a usable Python is
        configured. This is NOT ICMP ping, this is just a trivial test module that
        requires Python on the remote-node. For Windows targets, use the
        [ansible.windows.win_ping] module instead. For Network targets, use the
        [ansible.netcommon.net_ping] module instead.

OPTIONS (= is mandatory):

- data
        Data to return for the `ping' return value.
        If this parameter is set to `crash', the module will cause an exception.
        [Default: pong]
        type: str
```

Figure 24: Ping documentation

Some modules need some arguments to be passed to them, and for this we can supply them by specifying the **`-a`** option. For instance, Ansible has a module called **`command`**, which executes whatever command we want directly on the remote host.

Example command: ping

Let’s run an example. We can run the **`ping`** command—this time not as a module, but as an argument passed to the **`command`** module.

This will log in to the remote host, execute the **`ping www.microsoft.com -c 2`** command from the remote host, and return the result.

Code Listing 43: Inventory file to use in the example

```
[webservers]
192.168.3.160
192.168.3.161
```

And the ansible.cfg file as follows.

Code Listing 44: Ansible.cfg with web servers

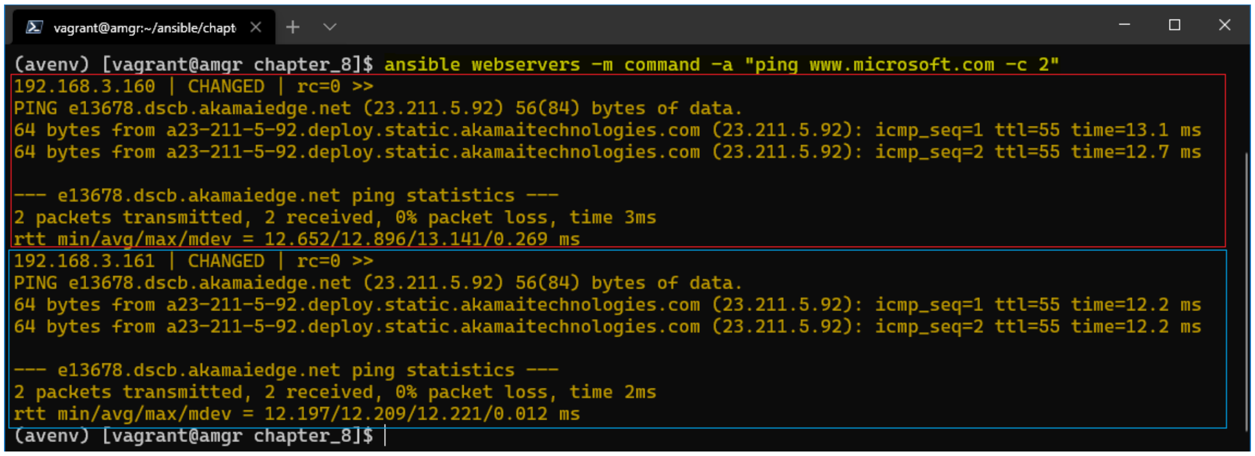
```
[defaults]
Inventory = ./inventory
```

Now, in the directory where we placed the two files, we can run the following command to ping the web servers.

Code Listing 45: Pinging www.microsoft.com from web servers

```
$ ansible webservers -m command -a  "ping www.microsoft.com -c 2"
```

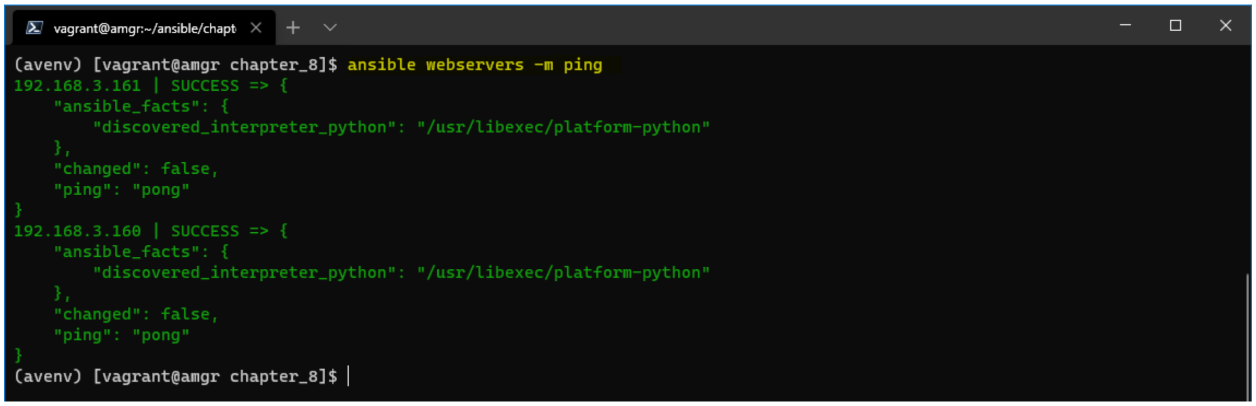
We can see that two results are returned, one per web server (as we have specified two web servers as the target of the module).



```
(avenv) [vagrant@amgr chapter_8]$ ansible webservers -m command -a "ping www.microsoft.com -c 2"
192.168.3.160 | CHANGED | rc=0 >>
PING e13678.dscb.akamaiedge.net (23.211.5.92) 56(84) bytes of data.
64 bytes from a23-211-5-92.deploy.static.akamaitechnologies.com (23.211.5.92): icmp_seq=1 ttl=55 time=13.1 ms
64 bytes from a23-211-5-92.deploy.static.akamaitechnologies.com (23.211.5.92): icmp_seq=2 ttl=55 time=12.7 ms

--- e13678.dscb.akamaiedge.net ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 3ms
rtt min/avg/max/mdev = 12.652/12.896/13.141/0.269 ms
192.168.3.161 | CHANGED | rc=0 >>
PING e13678.dscb.akamaiedge.net (23.211.5.92) 56(84) bytes of data.
64 bytes from a23-211-5-92.deploy.static.akamaitechnologies.com (23.211.5.92): icmp_seq=1 ttl=55 time=12.2 ms
64 bytes from a23-211-5-92.deploy.static.akamaitechnologies.com (23.211.5.92): icmp_seq=2 ttl=55 time=12.2 ms

--- e13678.dscb.akamaiedge.net ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 2ms
rtt min/avg/max/mdev = 12.197/12.209/12.221/0.012 ms
(avenv) [vagrant@amgr chapter_8]$ |
```

Figure 25: Pinging www.microsoft.com from web servers

Let’s see how this is different from running the **`ping`** module.



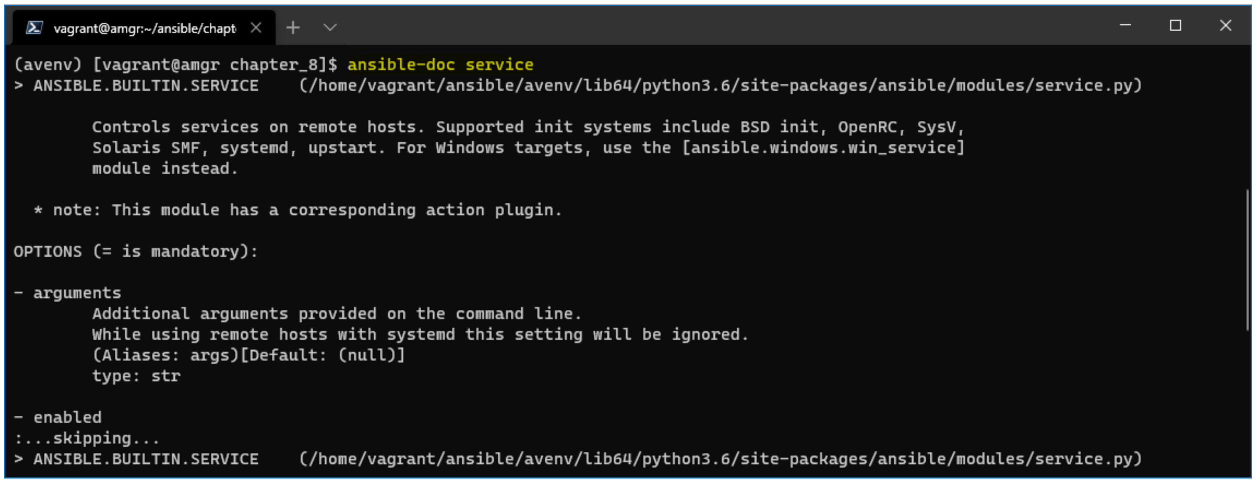
```
(avenv) [vagrant@amgr chapter_8]$ ansible webservers -m ping
192.168.3.161 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
192.168.3.160 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
(avenv) [vagrant@amgr chapter_8]$ |
```

Figure 26: Pinging web servers

Here we can also see two results, but this time the **`amgr`** is pinging the webservers (**`web160`**, **`web161`**) rather than the webservers themselves pinging www.microsoft.com.

Example command: service

Another very useful module is the **`service`** module. We can check the documentation to see what this module all about.



```
(avenv) [vagrant@amgr chapter_8]$ ansible-doc service
> ANSIBLE.BUILTIN.SERVICE    (/home/vagrant/ansible/avenv/lib64/python3.6/site-packages/ansible/modules/service.py)

        Controls services on remote hosts. Supported init systems include BSD init, OpenRC, SysV,
        Solaris SMF, systemd, upstart. For Windows targets, use the [ansible.windows.win_service]
        module instead.

  * note: This module has a corresponding action plugin.

OPTIONS (= is mandatory):

- arguments
        Additional arguments provided on the command line.
        While using remote hosts with systemd this setting will be ignored.
        (Aliases: args)[Default: (null)]
        type: str

- enabled
:...skipping...
> ANSIBLE.BUILTIN.SERVICE    (/home/vagrant/ansible/avenv/lib64/python3.6/site-packages/ansible/modules/service.py)
```

Figure 27: Service module documentation

The documentation says that it controls services on remote hosts. This means that we are able to control the status of services on a given host, such as by stopping or (re)starting the service.

In the next example, we are issuing a **`service`** command to restart **`sshd`** service running on the web servers.

If there is need for elevated rights, the **`-b`** option can be specified directly in the command line. This option is the same as when specifying **`become=True`** in the **ansible.cfg** file (which we have encountered previously).

Code Listing 46: Restarting sshd service on remote host

```
$ ansible webservers -m service -a "state=restarted name=sshd" -b
```

The following is the content of the **ansible.cfg** if we're not supplying the **`-b`** option.

Code Listing 47: Ansible.cfg with become option specified

```
[defaults]
Inventory = ./inventory
deprecation_warnings = False

[privilege_escalation]
become=True
```

Common modules

In the previous examples, we have seen how to use ad-hoc commands. Ansible offers many modules that could be used. The following sections cover some of the most common ones.

Ansible built-in modules

The following list is just an extract of the most-used modules. These modules are idempotent, which means that the operation, action, or request can be applied multiple times without changing the result (state of the system).

Table 7

`apt`	Manages apt-packages; useful for managing Linux distributions using **`apt`**.
`yum`	Manages yum packages; useful for managing Linux distributions using **`yum`**.
`dnf`	Manages dnf packages; useful for managing Linux distributions using **`dnf`**.
`copy`	Copies files to remote location.
`file`	Manages files and file properties.
`get_url`	Downloads files from http, https, or ftp to node.
`git`	Deploys software or files from git checkouts.
`reboot`	Reboots a machine.
`user`	Manages user accounts.

The full list of modules is available in the official Ansible documentation.

Command modules

In addition to using the built-in Ansible modules, you can run commands directly on the managed hosts. Those are very handy if there are no specific modules built for Ansible, but in general, the advice is to use the Ansible ones if possible.

These modules are not idempotent! In other words, issuing multiple identical commands may **not** have the same effect as issuing a command once.

Table 8

`command`	Runs a single command on the remote system.
`shell`	Runs a command on the managed host system's shell.
`raw`	Runs a command directly using the remote shell and bypasses the module subsystem, which is useful when the remote system cannot have Python installed.

Idempotent modules

We have already mentioned that Ansible runs the ad-hoc commands in an idempotent way. As is stated in the Ansible documentation:

"An operation is idempotent if the result of performing it once is exactly the same as the result of performing it repeatedly without any intervening actions."

Let's demonstrate this concept by using the **`group`** module, which creates a user group on the managed host, (in our case, web servers).

We have learned that we can investigate the documentation and check the options (arguments) the **`group`** command supports by running the **`ansible-doc group`** command-line command.

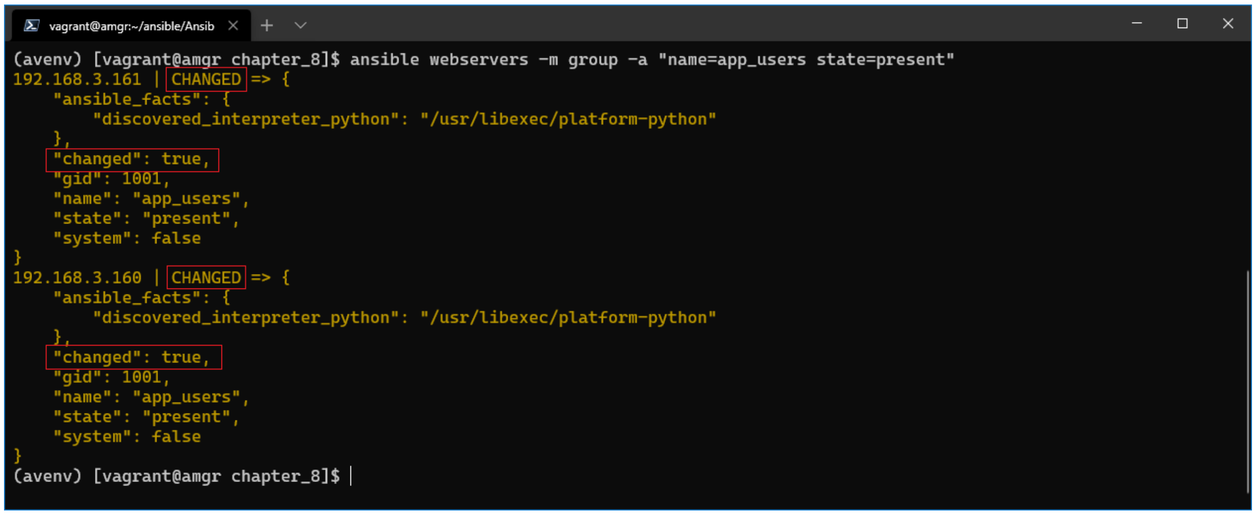
We can see that group supports several arguments:

- **`name`**: A mandatory argument that specifies the name of the group to be added or removed.
- **`state`**: An optional argument that can be either **`absent`** or **`present`**. If **`absent`** is specified, we will instruct Ansible that the final state of the group on the managed host should be, indeed, absent (removed). Otherwise, it will be **`present`**, which is the default value.

Code Listing 48: Create app_users group on webservers

```
$ ansible webservers -m group -a "name=app_users state=present" -b
```

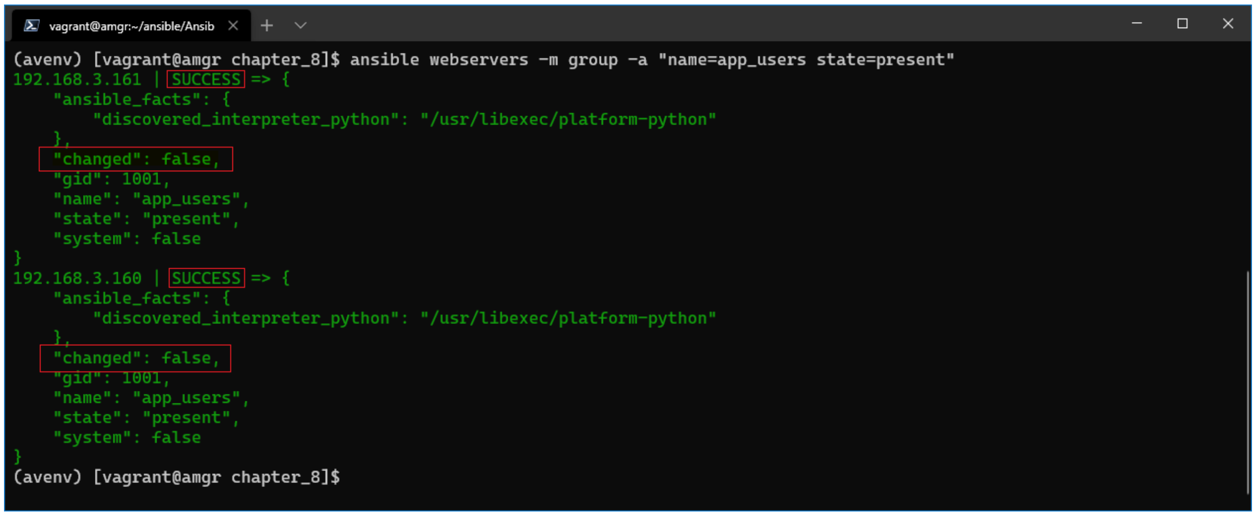
The result of running the command is shown in the following figure. We can see that the result of the command was **`CHANGED`**. This means that a change has been applied to the managed host. The **`changed`** property is also set to **`true`**, which tells us that the change has been applied, so the state of the system has changed.



```
(avenv) [vagrant@amgr chapter_8]$ ansible webservers -m group -a "name=app_users state=present"
192.168.3.161 | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": true,
    "gid": 1001,
    "name": "app_users",
    "state": "present",
    "system": false
}
192.168.3.160 | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": true,
    "gid": 1001,
    "name": "app_users",
    "state": "present",
    "system": false
}
(avenv) [vagrant@amgr chapter_8]$
```

Figure 28: Result of create app_users group on webservers

However, if we rerun the same command, the message we see is another one. You can see that the command was successful (**`SUCCESS`**), but the **`changed`** property is **`false`**. This means that the command executed didn't apply any change on the managed host.



```
(avenv) [vagrant@amgr chapter_8]$ ansible webservers -m group -a "name=app_users state=present"
192.168.3.161 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "gid": 1001,
    "name": "app_users",
    "state": "present",
    "system": false
}
192.168.3.160 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "gid": 1001,
    "name": "app_users",
    "state": "present",
    "system": false
}
(avenv) [vagrant@amgr chapter_8]$
```

Figure 29: (Re)running the command

Because this command is idempotent, it didn't change the state of the managed machine by running the same command twice.

Chapter 9 Ansible Playbook

An Ansible playbook is a file written in the Ansible automation language, and it's based on the YAML format. The playbook is Ansible's means to perform configuration, deployment, and orchestration.

As opposed to the ad-hoc commands we discussed previously, playbooks can declare configurations, but they can also orchestrate the steps to be executed. A playbook contains tasks that can be launched synchronously or asynchronously, depending on the use case.

One can think of a playbook as an entry point for all of the operations that we would like to execute in a given order against one or a set of managed hosts. In that sense, playbooks are meant to be kept in the source control (such as Git) and should be treated as any other application code.

Basic structure

The first thing to know about an Ansible playbook is that it's written in YAML. There are just a few rules to pay attention to when writing the code:

- The file typically should have the standard **`yml`** extension.
- The file must start with (three dashes) **`---`**.
- The file is indented by two spaces (not tabs), as emphasized by the orange highlights in the following code snippet.
- The items of the same level must be aligned.

Code Listing 49: Simple playbook example (webserver.yml)

```
---
- name: Web Server Playbook
  hosts: webservers
  become: yes

  tasks:
    - name: Pinging web server
      ansible.builtin.ping:
        data: pong
```

We have specified the **`name`** of the play, the **`hosts`** to which this code is going to be applied (group in the inventory file), and the **`become:yes`** option (to enable privilege escalation). These settings are global to the playbook.

We can see that there is a **`tasks`** section defined and aligned by two spaces. The **`tasks`** section is a list of individual tasks identified by the **`name`**, the **`module`**, and other possible arguments. It's very similar to what we have seen previously with the ad-hoc commands.

Although we see only one task in the previous example, we can specify more of them, and as we are going to see in the following chapters, combine them with variables, handlers, or roles to obtain a very powerful orchestration.

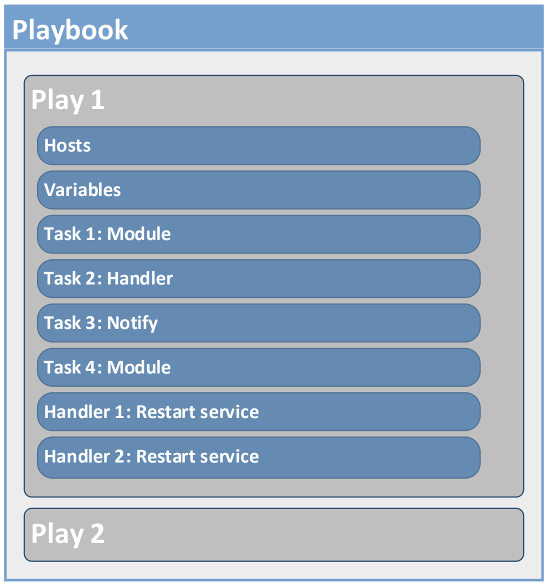


Figure 30: Typical playbook structure

Executing the playbook

To execute the playbook, we use the **`ansible-playbook`** command.

Code Listing 50: Execution of the webserver.yml playbook

```
$ ansible-playbook webserver.yml
```

The result is going to look similar to the following.

Code Listing 51: Playbook output

```
(avenv) [vagrant@amgr simple_playbook]$ ansible-playbook webserver.yml
PLAY [Web Server Playbook]
**************************************************************************
TASK [Gathering Facts]
******************************************************************************
ok: [193.168.3.161]
ok: [193.168.3.160]
TASK [Pinging web server]
***************************************************************************
ok: [193.168.3.161]
ok: [193.168.3.160]
```

```
PLAY RECAP
*******************************************************************************************
193.168.3.161 : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
193.168.3.160 : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
```

We can clearly see that the output contains the output grouped under the names of the sections as defined in the webserver.yml:

- **`Gathering Facts`**: This is the phase of retrieving information about all of the hosts listed in the inventory file, and that are in the context of the playbook. We can disable it by specifying the option **`gather_facts: false`** in the default section of the playbook.

- **`TASK [task_name]`**: For every task defined in the playbook and executed, there will be one section in the result. In our case, we have only one task. Pinging web server is executing the module **`ping`**, and it is displaying that the operation has been completed successfully, specifying the name of the host, as well.

- **`PLAY RECAP`**: After each run, there is a summary of the playbook execution stating if there were failures during the execution, and it constitutes a very nice report of the playbook execution itself. We can clearly see that there are a few pieces of information available:

    - **`ok=2`**: There are two operations completed successfully.
    - **`changed=0`**: Something has changed on that particular host. In our case, **`ping`** doesn't change anything, really.
    - **`unreachable=0`**: If playbooks are not able to reach some hosts, this will be shown here.
    - **`failed=0`**: If any operation fails, this will be shown in the recap, as in the task itself with more details.
    - **`skipped=0`**: Shows if there are tasks that are not being executed due to some conditions set.
    - **`rescued=0`**: Tasks that failed but recovered execution. There is a fallback solution in case a task fails to execute.
    - **`ignored=0`**: If there are some errors being ignored, the count will be shown here.

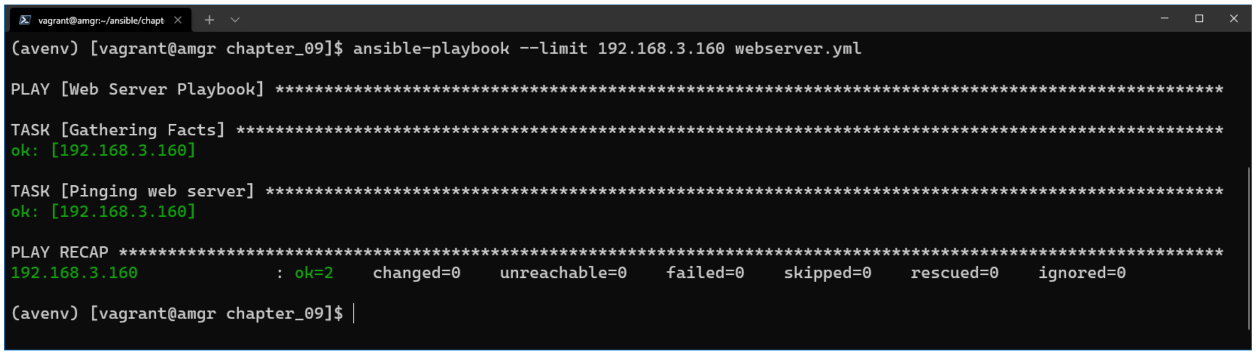
Limit option

There are a few options when using **`ansible-playbook`** commands, such as limiting the hosts against which we would like to run the command. This is done by using the **`--limit`** argument. This is useful if we would like to target just one server, for instance, without changing the playbook itself.

Code Listing 52: Targeting specific hosts

```
$ ansible-playbook --limit 192.168.3.160 webserver.yml
```

The result will show that the command was executed only on one server, as specified in the command.



```
(avenv) [vagrant@amgr chapter_09]$ ansible-playbook --limit 192.168.3.160 webserver.yml

PLAY [Web Server Playbook] *******************************************************************************************

TASK [Gathering Facts] ***********************************************************************************************
ok: [192.168.3.160]

TASK [Pinging web server] ********************************************************************************************
ok: [192.168.3.160]

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

(avenv) [vagrant@amgr chapter_09]$
```

Figure 31: Command executed only on one server

Checking the syntax

If we would like to check the syntax of the playbook without executing it, there is the **`--syntax-check`** option. This is very handy for figuring out if there are issues with the file.

Code Listing 53: playbook syntax checking

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

If an error is found, the line containing the issue will be displayed.

Dry run

To run the “dry run,” which is like a test mode, there is the **`-C`** option. Dry run mode will show the output of the specified change, but without changing the managed hosts. This is extremely useful when testing, as we can see which changes would occur if we execute this command.

Code Listing 54: Dry run option when executing playbooks

```
$ ansible-playbook -C webserver.yml
```

Variables

We have seen the most basic playbook content and how to execute it. Now we have to look into how to parametrize and make the playbooks more useful by specifying variables.

Variables provide a very convenient way to handle dynamic values. Variables could be about anything, such as a list of users, a list of software packages to install or uninstall, and services to start or stop.

It's obvious that having everything statically defined in the playbook would work, but it would also be a bit more cumbersome to handle, as this would typically result in a larger code base with some repetition, which would potentially increase the possibility of errors in code.

Naming convention

The variables have to start with a letter, and they can include underscores and numbers.

Table 9: Variables: naming convention

Valid variable name	Invalid
account_name	account name account-name account.name
account_nr_1	account-nr1 accountnr#1 account$1 1_account

Variables scope

The scope of the variable is the context within which it is defined. In other words, this defines in which parts of the program variables will be seen, applied, or used.

Ansible defines three scopes, summarized in the following table.

Table 10: Variables: scope

Scope	Description
Global	This is set by configuration, environment variables, and the command line. It is set to all hosts.
Host	Directly associated to a specific **`host`** or **`host groups`** (as defined in the inventory file). Those are variables defined in the inventory or in the **`host_vars`** directory.
Play	Scope applies to the play in which variables are declared. It applies to all hosts in the context of the current play. The **`vars`** directive in the playbook is where the variables are declared. Additionally, they can be defined by the **`include_vars`** task.

If a variable is defined in several scope levels, the value of the level that has the precedence would be taken as the variable value. The narrower in scope we go, the more precedence it has.

Play scope overrides the **host** variables, which override the **global** variables, which have more precedence over the variables defined in the inventory file. However, if the variable value is defined in the command line directly while executing the command, it has the highest precedence. By providing the **-e** option in the **ansible-playbook** command, we can override any value.

Declaring variables in the playbook

In the playbook, we can define the variables in two possible ways: either by declaring them explicitly using the **vars** directive, or by using the **vars_files** directive to include the file(s) where the variables are declared (in our case, in the vars/users.yml and vars/services.yml files).

Code Listing 55: Variables—declaring

```yaml
---
- name: Example with vars
  hosts: all
  vars:
    user_name: john
    user_description: "standard user"

- name: Example with vars_files
  hosts: all
  vars_files:
    - vars/users.yml
    - vars/services.yml
```

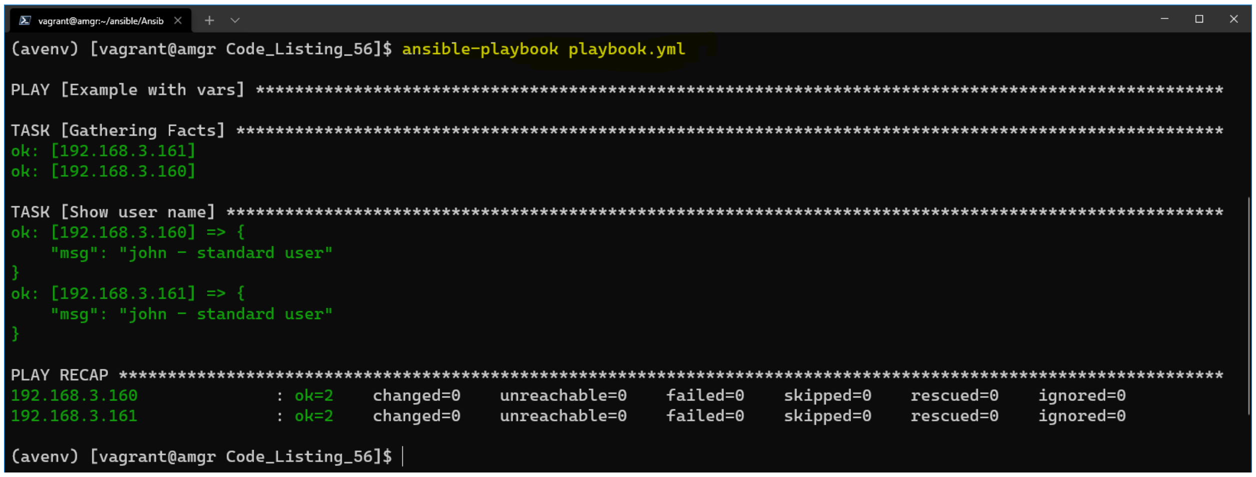
We can then reference those variables in the playbook by placing the variable name between double curly braces: **{{ name_of_variable }}**.

Code Listing 56: Variables—using

```yaml
---
- name: Example with vars
  hosts: all
  vars:
    user_name: john
    user_description: "standard user"

  tasks:
    - name: Show user name
      debug:
        msg: "{{ user_name }} - {{ user_description }}"
```

In this snippet, you can see that we are using the **`debug`** module. This module is useful when we want to display some information to the console, in this case the variables defined. The two variables previously defined are stored in the **`msg`** argument of the module. An important thing to notice is that we placed the variables between quotes.

Let's execute this code and see what the output will be, as shown in the following figure. We can clearly see the **`"msg": "john - standard user"`** is displayed in the output.



```
(avenv) [vagrant@amgr Code_Listing_56]$ ansible-playbook playbook.yml

PLAY [Example with vars] ****************************************************************************************************

TASK [Gathering Facts] ******************************************************************************************************
ok: [192.168.3.161]
ok: [192.168.3.160]

TASK [Show user name] *******************************************************************************************************
ok: [192.168.3.160] => {
    "msg": "john - standard user"
}
ok: [192.168.3.161] => {
    "msg": "john - standard user"
}

PLAY RECAP ******************************************************************************************************************
192.168.3.160              : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
192.168.3.161              : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

(avenv) [vagrant@amgr Code_Listing_56]$
```

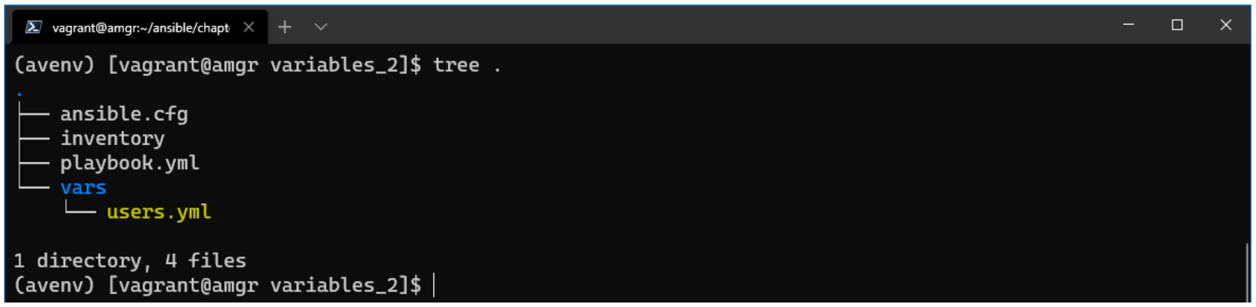
Figure 32: Retrieving variable values

This is the same output we would have when using the the **`var_files`** directive. In the users.yml file, we would place the same content as in the **`vars`** section.

Code Listing 57: Content of the users.yml file

```yaml
user_name: john
user_description: "standard user"
```

By looking at the folder structure, we can see there is a **`users.yml`** file in the vars folder.



```
(avenv) [vagrant@amgr variables_2]$ tree .
.
├── ansible.cfg
├── inventory
├── playbook.yml
└── vars
    └── users.yml

1 directory, 4 files
(avenv) [vagrant@amgr variables_2]$
```

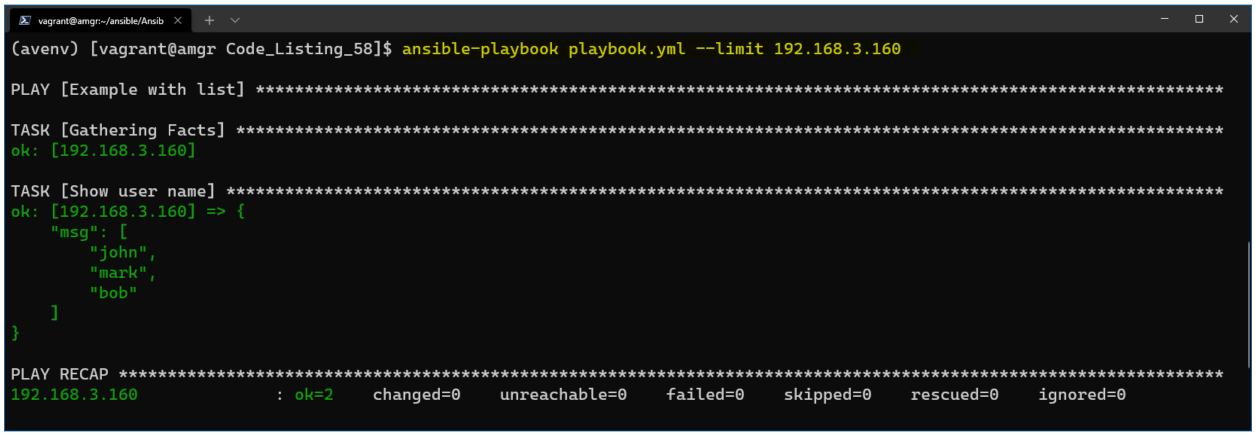
Figure 33: Folder structure with vars directory

It's worth noting that we can also define a list or dictionary of parameters.

Code Listing 58: Playbook with a list

```
---
- name: Example with list
  hosts: all
  vars:
    users:
      - john
      - mark
      - bob

  tasks:
    - name: Show user name
      debug:
        msg: "{{ users }}"
```

This code returns the following result.



```
(avenv) [vagrant@amgr Code_Listing_58]$ ansible-playbook playbook.yml --limit 192.168.3.160

PLAY [Example with list] ****************************************************************************************

TASK [Gathering Facts] ******************************************************************************************
ok: [192.168.3.160]

TASK [Show user name] *******************************************************************************************
ok: [192.168.3.160] => {
    "msg": [
        "john",
        "mark",
        "bob"
    ]
}

PLAY RECAP ******************************************************************************************************
192.168.3.160              : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

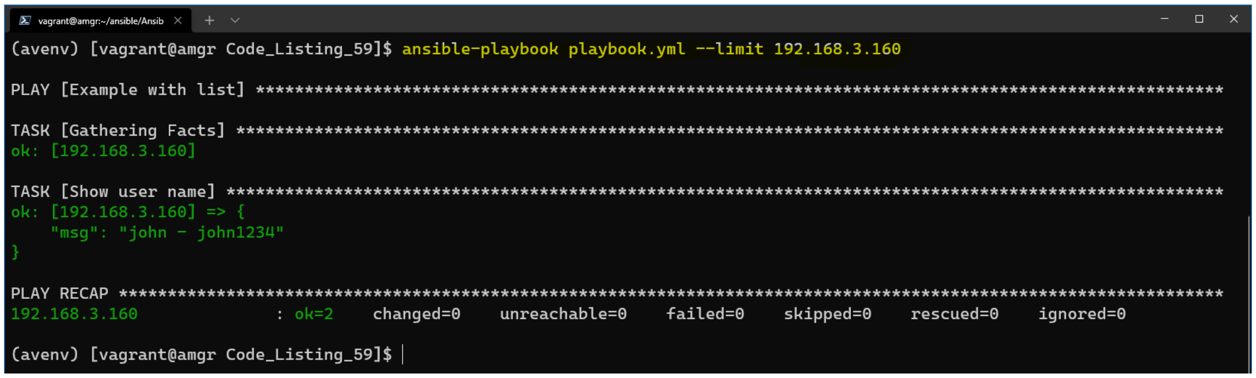
Figure 34: Result by using a list

Or we can define a dictionary of values, like in the following.

Code Listing 59: Playbook with dictionary

```
---
- name: Example with list
  hosts: all
  vars:
    users:
      john:
        name: john
        default_password: john1234
      mark:
        name: mark
        default_password: mark1234
      bob:
        name: bob
        default_password: bob1234
  tasks:
    - name: Show user name
      debug:
        msg: "{{ users['john']['name'] }} -
        {{ users['john']['default_password']}}"
```

This code returns the following result.



```
(avenv) [vagrant@amgr Code_Listing_59]$ ansible-playbook playbook.yml --limit 192.168.3.160

PLAY [Example with list] **************************************************************************************************

TASK [Gathering Facts] ****************************************************************************************************
ok: [192.168.3.160]

TASK [Show user name] *****************************************************************************************************
ok: [192.168.3.160] => {
    "msg": "john - john1234"
}

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

(avenv) [vagrant@amgr Code_Listing_59]$
```

Figure 35: Running playbook by using dictionary variable

Declaring group or host variables

Group or host variables can be declared either in the inventory file or as specific files in the **`group_vars`** or **`host_vars`** directories, in the same location as the inventory file.

The naming convention for files is driven by using the same host names or group names as defined in the inventory file.

Say we had an inventory file as follows.

Code Listing 60: Inventory file

```
[webservers]
web160
web161
[load_balancers]
lb

[databases]
db
```

The following is the folder structure with files containing group or host files with the same names as defined in the inventory file.

Code Listing 61: Directory structure of group and host vars

```
\
|-- group_vars
|    |-- all
|    |-- webservers
|    |-- load_balancers
|    |-- databases
|-- host_vars
|    |-- web160
|    |-- web180
|-- playbook.yml
|-- ansible.cfg
|-- inventory
```

These variables will be loaded by default, depending on what is declared in the playbook **`hosts`** section.

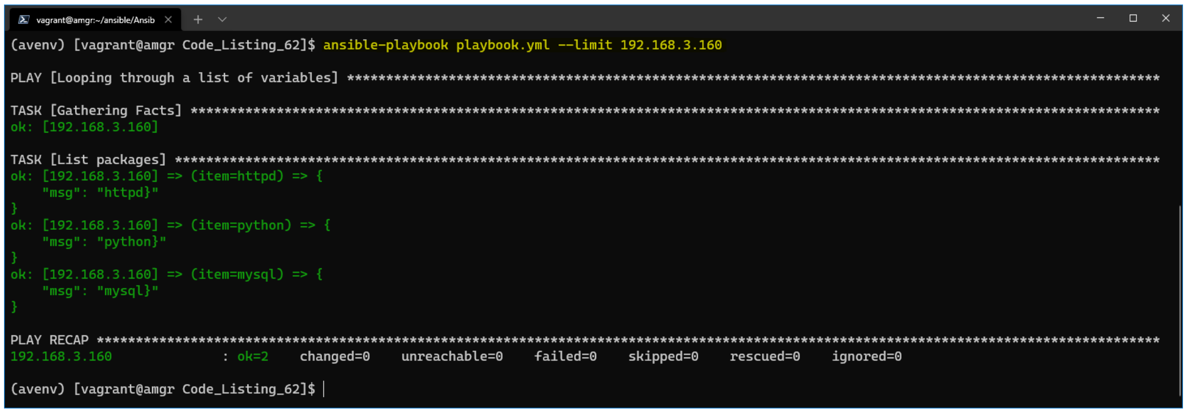
Looping through variables

Ansible defines the **`loop`** keyword that enables looping through variables within a given task. A special variable called **`item`** holds the current item value during the iteration through values.

Code Listing 62: Looping

```
---
- name: Looping through a list of variables
  hosts: all
  vars:
    packages:
      - httpd
      - python
      - mysql
  tasks:
    - name: List packages
      debug:
        msg: "{{ item }}}"
      loop: "{{ packages }}"
```

When we execute the playbook, in the output we can see three distinct **`msgs`** being returned to us. In fact, this is executing the task as many times as there are items in the list.



```
(avenv) [vagrant@amgr Code_Listing_62]$ ansible-playbook playbook.yml --limit 192.168.3.160

PLAY [Looping through a list of variables] ***************************************************************************************

TASK [Gathering Facts] ***********************************************************************************************************
ok: [192.168.3.160]

TASK [List packages] *************************************************************************************************************
ok: [192.168.3.160] => (item=httpd) => {
    "msg": "httpd}"
}
ok: [192.168.3.160] => (item=python) => {
    "msg": "python}"
}
ok: [192.168.3.160] => (item=mysql) => {
    "msg": "mysql}"
}

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

(avenv) [vagrant@amgr Code_Listing_62]$
```

Figure 36: Result of looping through variables

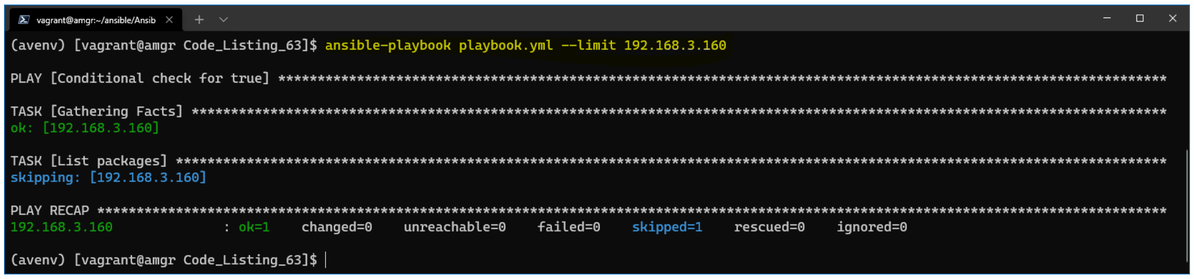
Conditional statements

Ansible supports conditional statements. Similar to an **`if`** statement in programming languages such as Python or C#, Ansible uses the keyword **`when`** to check whether a condition is being satisfied or not.

Code Listing 63: Conditional statement example

```
---
- name: Conditional check for true

  hosts: all
  vars:
    preinstall_package: false
  tasks:
    - name: List packages
      debug:
        msg: "executed"
      when: preinstall_package
```

In this case, the debug task will not be executed, as the **`preinstall_package`** is set to **`false`**. We can clearly see in the output that the task is skipped.



```
(avenv) [vagrant@amgr Code_Listing_63]$ ansible-playbook playbook.yml --limit 192.168.3.160

PLAY [Conditional check for true] ************************************************************************************************

TASK [Gathering Facts] ***********************************************************************************************************
ok: [192.168.3.160]

TASK [List packages] *************************************************************************************************************
skipping: [192.168.3.160]

PLAY RECAP ***********************************************************************************************************************
192.168.3.160              : ok=1    changed=0    unreachable=0    failed=0    skipped=1    rescued=0    ignored=0

(avenv) [vagrant@amgr Code_Listing_63]$
```

Figure 37: Conditional statement result

Checking for true or false values is just one of the possibilities. There are other predefined keywords, such as **`is defined`**, where the variable is checked for its existence. The following table defines other possibilities.

Table 11: Conditional statements

Operation	Example

Equal "some string"
Equal some number
	`package_name == "httpd"`
	`port_number == 80`

Less than
Less than or equal to
Greater than
Greater than or equal to
	`port_number < 80`
	`port_number <= 80`
	`port_number > 80`
	`port_number >= 80`

Not equal to	`package_name != "httpd"`
	`port_number != 80`

Variable exists
Variable doesn't exist
	`port_number is defined`
	`port_number is not defined`

Boolean check for true
Boolean check for false
1, True, yes: evaluate to true
0, False, no: evaluate to false
	`user_exists`
	`not user_exists`

Value present in a list of values	`username in user_list`

Conditions can be multiple, and we can separate them by using the **`or`** and **`and`** keywords.

Code Listing 64: Using and in condition

```
when: username == “john” and groupname == “admin”
```

The equivalent to the **`and`** statement can also be written as a list.

Code Listing 65: Alternative syntax for conditional and

```
when:
  - username = "john"
  - groupname = "admin"
```

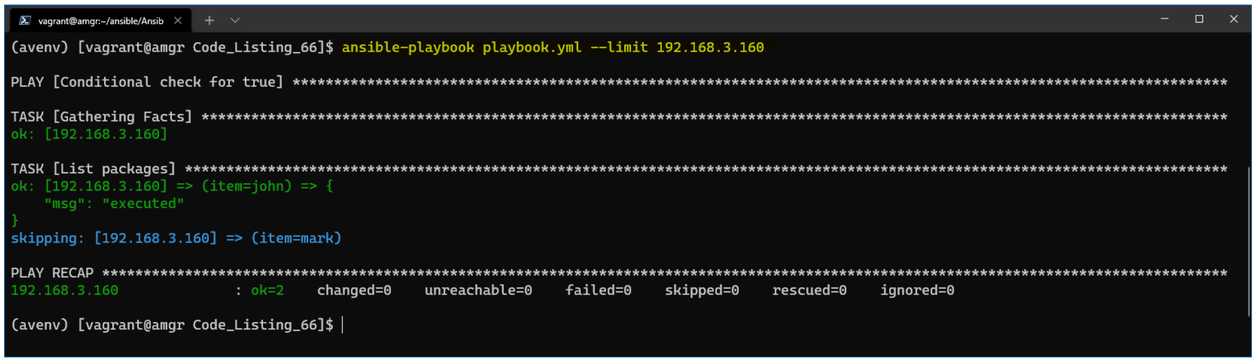
Combining loops and conditional statements

With the knowledge of how to write loops and conditional statements, we can certainly combine the two, making the playbook execution even more powerful.

Code Listing 66: Combining loop and when statement

```yaml
---
- name: Conditional check for true
  hosts: all
  vars:
    users:
      - john
      - mark
  tasks:
    - name: List packages
      debug:
        msg: "executed"
      loop: "{{ users }}"
      when: item == "john"
```

As you may already expect, this task will be executed only if the name of the user is **`john`**.



```
(avenv) [vagrant@amgr Code_Listing_66]$ ansible-playbook playbook.yml --limit 192.168.3.160

PLAY [Conditional check for true] ***************************************************************************************************************

TASK [Gathering Facts] ***************************************************************************************************************************
ok: [192.168.3.160]

TASK [List packages] *****************************************************************************************************************************
ok: [192.168.3.160] => (item=john) => {
    "msg": "executed"
}
skipping: [192.168.3.160] => (item=mark)

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

(avenv) [vagrant@amgr Code_Listing_66]$
```

Figure 38: Result—loop and when combined

Chapter 10 Ansible Playbook Handlers

There are situations where we want a task to run only when a change is made on a managed host. For example, we may want to restart a particular service if a task updates the configuration of a service, or we might reboot the machine after some package installation.

Ansible uses handlers to address this use case. We may ask, why not simply create a task at the end of the playbook that would reboot the server, or something similar? This task would need to depend on the execution of other tasks and to check the status of each of them to decide if something has to happen or not.

Ansible has solved this issue elegantly by introducing handlers. Handlers are tasks that only run when notified by other tasks, and only when the change happens on the managed host. If a task doesn't notify the handler, it won't run.

A nice thing about handlers is that they run only once, even if notified by multiple tasks. This fits perfectly, for instance, with the reboot use-case where it doesn't make sense to reboot the server after each task requiring it, but only once when all the tasks have been executed.

Handlers have a unique name globally and typically get placed at the end of the playbook. All of the modules used by tasks can also be used in the handler, so technically everything we can do in a task, we can do also in a handler itself.

Let's see how to declare and link a handler to a task.

Code Listing 67: Handler definition

```
---
- name: Handler restarting httpd service after installation
  hosts: webservers
  become: yes
  gather_facts: false

  tasks:
    - name: Install apache package
      yum:
        name: httpd
        state: present
      notify: Restart apache

  handlers:
    - name: Restart apache
      service:
        name: httpd
        state: restarted
```

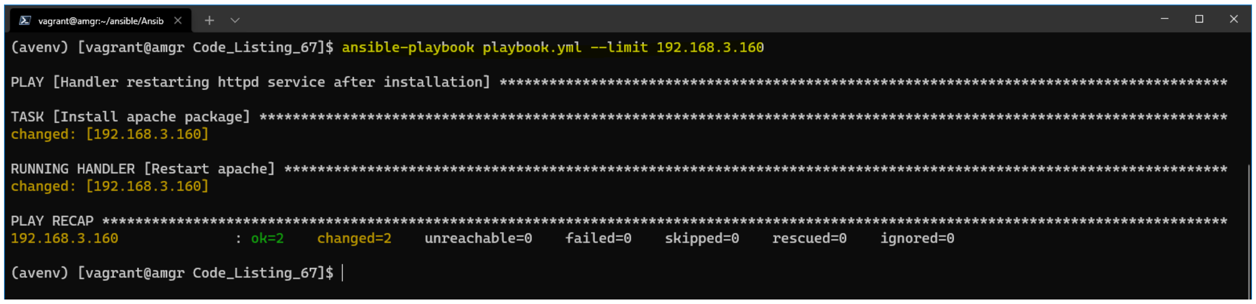
In the previous code snippet, we can see a playbook containing one task and one handler.

The task's responsibility is to install the **`httpd`** (Apache HTTP Server) by using the **`yum`** package manager. The **`state`** present in the task means that we want the **`httpd`** service to be present on the system as a result of running the task.

We can also see that there is a **`notify`** keyword after the task (aligned to the name and module), and the handler's name to which the link is made (**`notify: Restart apache`**).

On the other hand, at the bottom of the playbook, we define a handler in the same way we would define a task: by specifying a **`name`**, **`module`**, and eventual **`arguments`**. The state **`restarted`** simply means that we want the service to be restarted.

By running the playbook, we will get the following result.



```
(avenv) [vagrant@amgr Code_Listing_67]$ ansible-playbook playbook.yml --limit 192.168.3.160

PLAY [Handler restarting httpd service after installation] ****************************************************************************************

TASK [Install apache package] ********************************************************************************************************************
changed: [192.168.3.160]

RUNNING HANDLER [Restart apache] *****************************************************************************************************************
changed: [192.168.3.160]

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

(avenv) [vagrant@amgr Code_Listing_67]$
```

Figure 39: Result—handler restarting Apache package

We can see that the service got installed on the managed host **`192.168.3.160 (web160)`** and that the **`httpd`** service got restarted.

Here are some more facts about handlers to keep in mind:

- We can declare more than one handler (in the example, we only had one).
- The order in which the handlers are executed depends on the order in which they are called (not declared) by the task.
- The handler will be executed only after all tasks are executed!
- If the task doesn't change the system, the handler won't run.
- In handlers, we can use variables in the same way we use them normally in tasks.

Chapter 11 Templating

In this chapter, we are going to discuss the Jinja2 templates. Jinja2 is a templating language for Python. Ansible uses Jinja2 templating to enable the dynamic creation of the content. The dynamic content is going to be driven by the variables used within a playbook. This is useful when we need to apply changes to the static content and adapt it to the managed hosts.

For instance, a load balancer configuration file might need to be updated with the list of available web servers. Rather than hard-coding this information in the load balancer's config file, and in fact duplicating this information that we need later on to maintain, we can simply utilize the list of hosts we already have predefined in our inventory file, change the content of the config file, and ship it to the load balancer.

When playbooks are executed, these variables get replaced by actual values defined in the playbooks. This way, templating offers an efficient and flexible solution to create or alter content with ease.

Jinja2 basic syntax

A Jinja2 template file is a text file that contains variables that get evaluated and replaced by actual values upon runtime or code execution. In a Jinja2 template file, you will find the following syntax:

- **`{{ }}`**: Used for embedding variables and using their value during code execution. For example, a simple syntax using the double curly braces is as shown: The **`{{ webserver }}`** is running on **`{{ nginx-version }}`**.
- **`{%  %}`**: Used for control statements such as **`loops`** and **`if-else`** statements.
- **`{#  #}`**: These denote comments that describe a task.

We can perform conditional statements such as loops and **`if-else`** statements, and transform the data using filters and more.

Code Listing 68: Example of a Jinja2 loop

```
{% for host in groups['webservers'] %}
    {{ host }}
{% endfor %}
```

Jinja2 module

To invoke the transformation of the template, we have to integrate it within a playbook. Ansible offers a module called `template`.

Code Listing 69: Basic template module with arguments

```
tasks:
  - name: Jinja2 template
    template:
      src: haproxy.cfg
      dest: /etc/haproxy/haproxy.cfg
      owner: root
      group: root
      mode: 0644
```

`Template` has a few arguments we could supply:

- **`src`**: The source file we would like to transform; in this case, haproxy.cfg.
- **`dest`**: Where we want to copy the content after the transformation.
- **`owner`**: Name of the user who should own the file/directory, as would be fed to *chown*.
- **`group`**: Name of the user who should own the file/directory, as would be fed to *chown*.
- **`mode`**: The permissions the resulting file or directory should have.

To debug the template, we could utilize the debug module with a special lookup function. This is particularly useful as the content won't be deployed to any host, but we could see the result of the transformation.

The lookup plugin is an Ansible extension to the Jinja2 templating language. For more information, you can run **`ansible-doc -t lookup -l`** to list all available lookup plugins. For more information about the lookup template plugin, you can always consult the documentation by running the **`ansible-doc -t lookup template`** command.

Code Listing 70: Task used for displaying transformations locally

```
tasks:
  - name: Show the template content
    debug:
      msg: "{{ lookup('template', './haproxy.cfg') }}"
```

Let's run this code in debug mode.

The inventory file we are going to use is as follows, and we can see that we are defining a **`webservers`** group with two servers.

Code Listing 71: Inventory file

```
[webservers]
web160
web161

[load_balancers]
lb
```

Let's create a file named **`webservers.j2`** and put it in the local folder, where we run the **`playbook.yml`**. The content of the **`webservers.j2`** is shown in Code Listing 72. This is actually our template file. Typically the file can have the **`.j2`** extension, but this is purely optional, as it might be useful when using text editors with the Jinja2 syntax option.

`group['webservers']` is something we haven't yet seen. It's a built-in collection that would return the content of the hosts as defined in the inventory file.

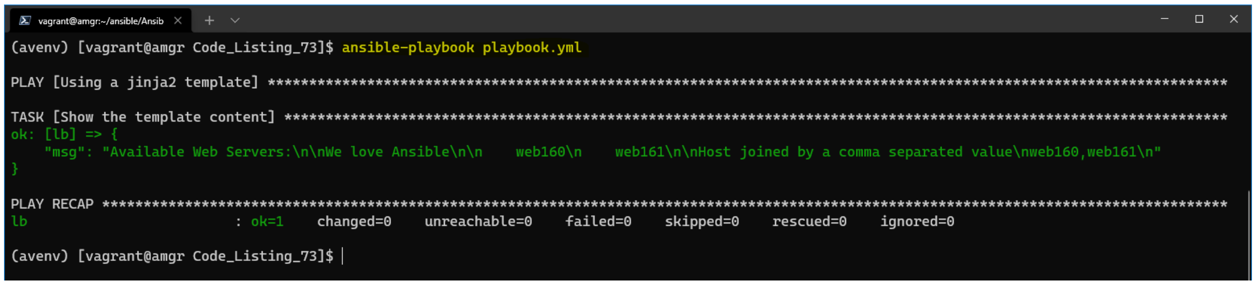
Code Listing 72: Webservers.j2 template file

```
Available Web Servers:

{# message variable defined in the template #}
{{ message }}

{# displaying list of hosts as defined in the inventory#}
{% for host in groups['webservers'] %}
    {{ host }}
{% endfor %}

{# message variable defined in the template #}
Host joined by a comma separated value
{{ groups['webservers'] | join(",") }}
```

Code Listing 73: Playbook running the template

```
---
- name: Using a jinja2 template
  hosts: load_balancers
  gather_facts: false
  vars:
    - message: "We love Ansible"

  tasks:
    - name: Show the template content
      debug:
        msg: "{{ lookup('template', './webservers.j2') }}"
```

The result of executing the playbook follows.



```
(avenv) [vagrant@amgr Code_Listing_73]$ ansible-playbook playbook.yml

PLAY [Using a jinja2 template] ************************************************************************************************************

TASK [Show the template content] **********************************************************************************************************
ok: [lb] => {
    "msg": "Available Web Servers:\n\nWe love Ansible\n\n    web160\n    web161\n\nHost joined by a comma separated value\nweb160,web161\n"
}

PLAY RECAP ********************************************************************************************************************************
lb                         : ok=1    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

(avenv) [vagrant@amgr Code_Listing_73]$
```

Figure 40: Debug output of the template

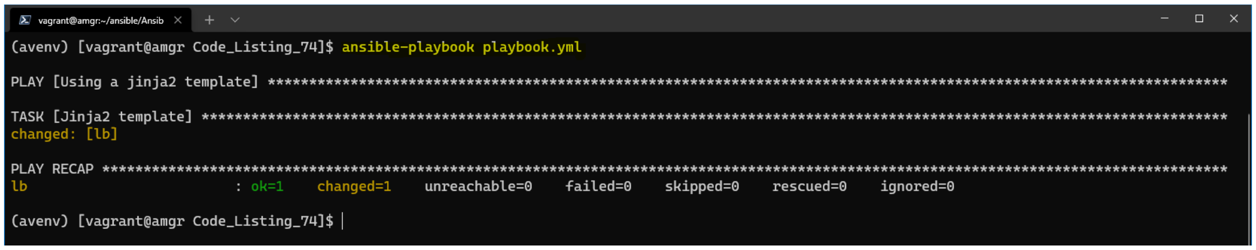
To run the code against the load balancer, let's change the playbook to look as follows.

Code Listing 74: Template module used in a task

```
---
- name: Using a jinja2 template
  hosts: load_balancers
  gather_facts: false
  vars:
    - message: "We love Ansible"

  tasks:
    - name: Jinja2 template
      template:
        src: webservers.j2
        dest: /home/vagrant
```

We are using the template module rather than the one used for the debugging purpose.

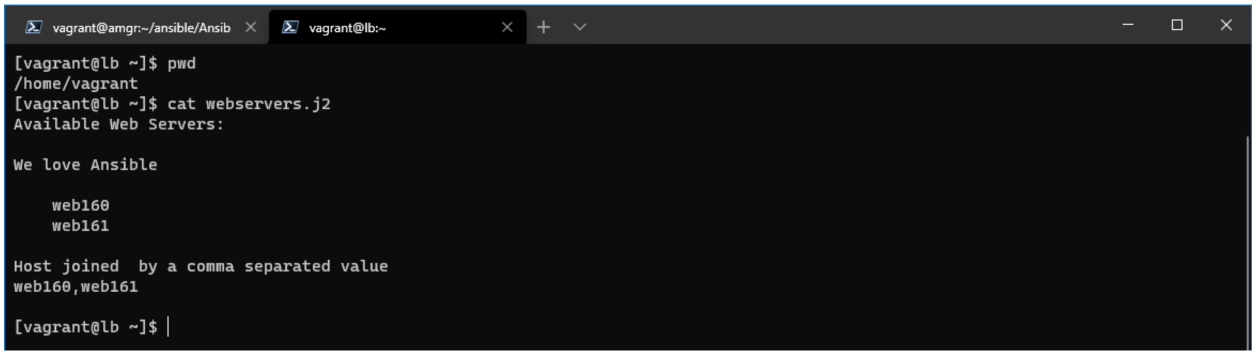
With the newly defined task, we are going to transform the file locally, and then the file will be copied to the load balancer at the destination folder **/home/vagrant**, as specified in the **`dest`** argument.



```
(avenv) [vagrant@amgr Code_Listing_74]$ ansible-playbook playbook.yml

PLAY [Using a jinja2 template] ************************************************************************************************************

TASK [Jinja2 template] ********************************************************************************************************************
changed: [lb]

PLAY RECAP ********************************************************************************************************************************
lb                         : ok=1    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

(avenv) [vagrant@amgr Code_Listing_74]$
```

Figure 41: Task with deployment on the load balancer

The task went fine, but let's check on the load balancer server and see the content of the file we have transformed and copied across. By using the **`cat`** command, we can see the content of the file.



```
[vagrant@lb ~]$ pwd
/home/vagrant
[vagrant@lb ~]$ cat webservers.j2
Available Web Servers:

We love Ansible

    web160
    web161

Host joined  by a comma separated value
web160,web161

[vagrant@lb ~]$
```

Figure 42: Load balancer home directory

The content of the file is as shown in the following code listing.

Code Listing 75: Result of the command execution

```
We love Ansible

    web160
    web161

Host joined by a comma separated value
web160,web161
```

Chapter 12 Ansible Vault: Data Encryption

In this chapter, we will describe how Ansible can protect sensitive data.

Whenever we want to automate something, sooner or later there is a moment where we encounter a password, API keys, certificates, or some other sensitive content.

As with any other code, we would use a versioning system such as Git or Subversion to keep our code versioned. This automatically means that we would expose sensitive information to people who are not intended to see it.

Rather than leaving this content (playbooks, variable files, etc.) visible in plaintext, Ansible offers a tool called **`Ansible Vault`** that allows data encryption and decryption. Only after encrypting the sensitive content should we feel safe about putting it into a source control.

To use Ansible Vault, we need to encrypt the data by providing a password, and the same for decrypting the content. This password of passwords obviously should be kept in some (other) safe place.

Ansible-vault command line tool

Ansible offers a command line tool called **`ansible-vault`**, which will enable us to encrypt the content.

Encrypt

The **`encrypt`** command encrypts the existing file. We can also pass the **`--output`** argument to specify the name of the newly encrypted file. If the **`--output`** is not provided, the file will simply be overwritten by the encrypted content.

Code Listing 76: Encrypt a file

```
$ ansible-vault encrypt <filename> --output=<new_filename>
```

Ansible offers the ability to encrypt multiple files with different passwords; each file would have its own. This can be done by supplying the **`--vault-id`** parameter to the **`encrypt`** command. In our case, we have given an identifier called **`secret@prompt`**. Prompt, in this case, means that the password will be supplied in the command line.

Code Listing 77: Specify value-id

```
$ ansible-vault encrypt --value-id secret@prompt <file>
```

Decrypt

The **decrypt** command decrypts the existing file.

Code Listing 78: Decrypt a file

```
$ ansible-vault decrypt <filename>
```

If the file has been given a **–vault-id** during the encryption, as we have seen in the previous example, we could provide the **--vault-id** in the command line with the same name as we provided during the encryption.

Code Listing 79: Decrypt a file supplying the vault-id

```
$ ansible-vault decrypt --vault-id secret@prompt <filename>
```

View

The **view** command enables us to see the content.

Code Listing 80: View encrypted file

```
$ ansible-vault view <filename>
```

Edit

The **edit** command allows us to edit the file. When we use the **edit** command, an editor will open the file ready to be edited in the command line.

Code Listing 81: Edit encrypted file

```
$ ansible-vault edit <filename>
```

Rekey

Changing a password for an already encrypted file is rather simple using the **rekey** command. This command is useful, as otherwise we would need to do two operations—decrypt and encrypt—to achieve the same thing. We can provide multiple files to the command.

Code Listing 82: Changing the encryption password

```
$ ansible-vault rekey <filename> <filename2>
```

Using secrets within the playbook

When referencing an encrypted file within an Ansible playbook, we need to provide a password for the content to be decrypted during the execution. The **`ansible-playbook`** command offers an option to supply the password through the **`--vault-id`** option.

The **`@prompt`** option will prompt the user to provide the password in the command line.

Code Listing 83: Execute playbook by providing the password

```
$ ansible-playbook --vault-id @prompt playbook.yml
```

Suppressing the output

There are situations where the secret might be displayed during the execution of the playbook. We can suppress the output by using the **`no_log: true`** directive within the task.

Code Listing 84: Suppressing the output

```
…
  tasks:
    - name: print variable
      debug:
        msg: {{ secret_variable }}
      no_log: true
```

Example code

Let's create a file called **`secret_file.yml`** with the following content.

Code Listing 85: content of secret_file.yml

```
password: some_very_secret_password
```

And let's encrypt this file with the **`encrypt`** command.

Code Listing 86: Encrypting the file

```
(avenv) [vagrant@amgr vault]$ ansible-vault encrypt secret_file.yml
```

You will be asked for a password, which will be used to encrypt the file. After supplying the password (in my case, the password is **1234**) and reopening the file, we will see that the file has been encrypted, and that it is unreadable by a human.

Code Listing 87: Secret_file.yml after encryption

```
$ANSIBLE_VAULT;1.1;AES256
66626463316264313732366638323532653130363232383131643434643235393763343039383362
3363356465383565613938623666306161663463396361370a366362623965663437353934636165
32373964313433343130343634663061393166656333373466663631383631643063623336313830
3234386136663232650a343862336163616264666565326630353934343635643663376536373464
35343664626635626263383764663139653165363835323332336632316462656262663839633833
3762343138656233333531326139336633393431323438663233
```

Let’s now create a playbook where we will use this secret information.

Code Listing 88: Playbook that references encrypted file

```
---
- name: Handling secret information
  hosts: lb
  become: yes
  gather_facts: false

  tasks:
    - name: Load encrypted variable
      include_vars:
        file: secret_file.yml

    - name: Retrieve information from the secret_file
      debug:
        msg: "{{ password }}"

    - name: Retrieve information from the secret_file but not show output
      debug:
        msg: "{{ password }}"
      no_log: true
```

We are already familiar with the **`tasks`** section. We can see a new module called **`include_vars`**, which is responsible for loading files with variables so that those are available to other tasks. This is very handy, as we can reference our encrypted file (**`secret_file.yml`**).

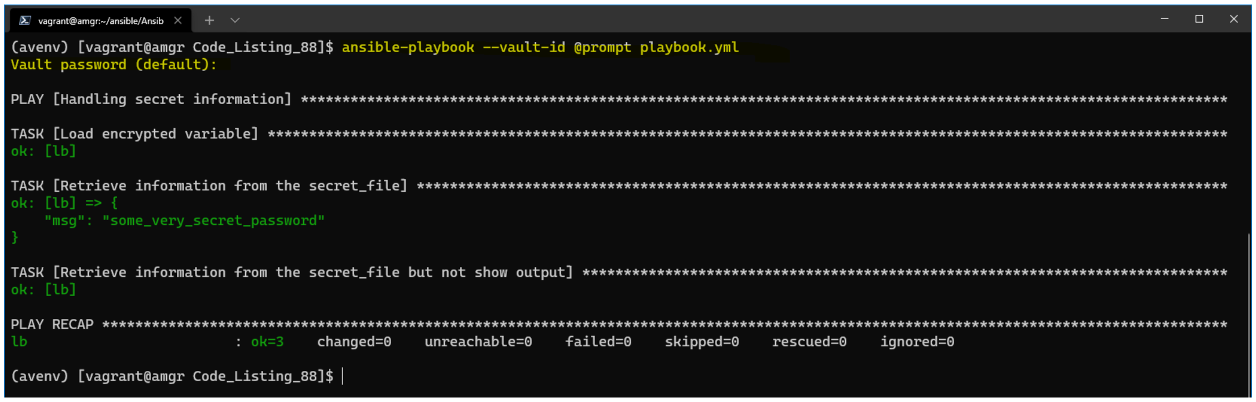
Additionally, we would like to display the password value in the subsequent two tasks; one, however, should suppress the output as we have previously seen, by using the **`no_log: true`** argument.

We can now execute the playbook and see that providing the **`--vault-id`** parameter with **`@prompt`** will actually prompt for a password. I will supply the password used for encrypting the file.

Code Listing 89: Run playbook with secret data

```
(avenv) [vagrant@amgr Code_Listing_88]$ ansible-playbook --vault-id @prompt
playbook.yml
```

The result looks like the following figure.



```
(avenv) [vagrant@amgr Code_Listing_88]$ ansible-playbook --vault-id @prompt playbook.yml
Vault password (default):

PLAY [Handling secret information] ****************************************************************************************************************

TASK [Load encrypted variable] ********************************************************************************************************************
ok: [lb]

TASK [Retrieve information from the secret_file] **************************************************************************************************
ok: [lb] => {
    "msg": "some_very_secret_password"
}

TASK [Retrieve information from the secret_file but not show output] ******************************************************************************
ok: [lb]

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

(avenv) [vagrant@amgr Code_Listing_88]$
```

Figure 43: Result when executing the playbook

We can clearly see that in the first task, the output gets properly shown, while in the second, that’s not the case.

Chapter 13 Ansible Runtime Facts

When executing playbooks, Ansible retrieves certain information and stores it for possible reuse. Information returned in Ansible terms are called *facts*.

We can utilize and reuse this information to decide on taking certain actions or simply use this information in the configuration when deploying some artifacts to other systems. A typical example is the IP address. We can retrieve one IP address from one system and reuse this information when configuring another system.

Code Listing 90: Retrieve and display facts

```
---
- name: Retrieving and displaying facts
  hosts: webservers
  become: yes
  gather_facts: true

  tasks:
    - name: Retrieve server information
      debug:
        var: ansible_facts
```

In the example shown in Code Listing 90, we can retrieve all the information about the webservers in scope. Pay attention to the **`gather_facts: true`**, as this has to be enabled in order to retrieve the variables.

Before starting to execute the tasks, Ansible will have its own internal task that will indeed gather the information about the machines that are in scope for the given run.

Code Listing 91: Running playbook to retrieve facts

```
(avenv) [vagrant@amgr Code_Listing_90]$ ansible-playbook --limit web160
playbook.yml
```

The output of this command is a long list of information provided.

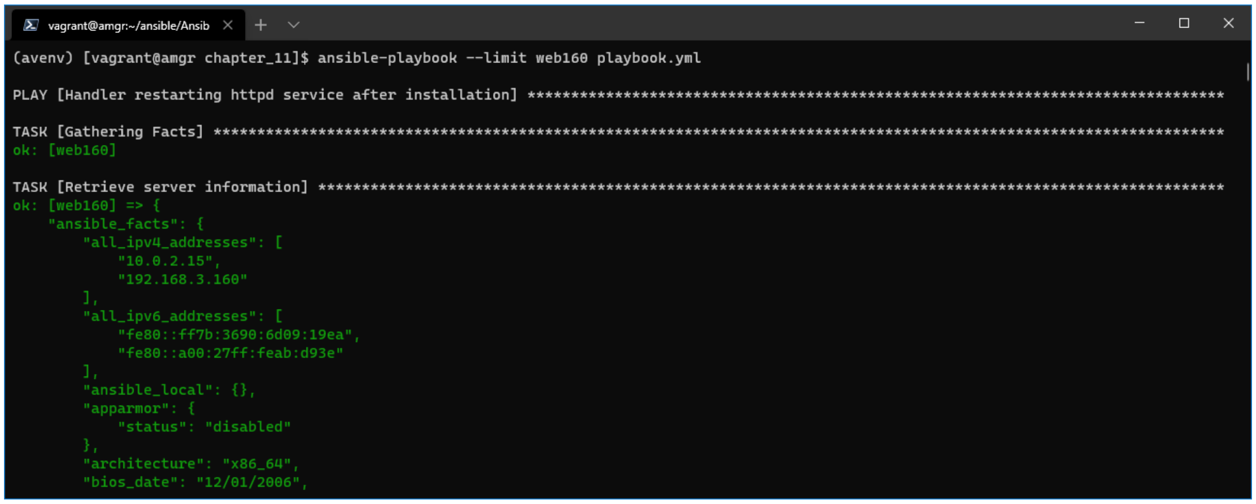

```
(avenv) [vagrant@amgr chapter_11]$ ansible-playbook --limit web160 playbook.yml

PLAY [Handler restarting httpd service after installation] **************************************************************************

TASK [Gathering Facts] **************************************************************************************************************
ok: [web160]

TASK [Retrieve server information] **************************************************************************************************
ok: [web160] => {
    "ansible_facts": {
        "all_ipv4_addresses": [
            "10.0.2.15",
            "192.168.3.160"
        ],
        "all_ipv6_addresses": [
            "fe80::ff7b:3690:6d09:19ea",
            "fe80::a00:27ff:feab:d93e"
        ],
        "ansible_local": {},
        "apparmor": {
            "status": "disabled"
        },
        "architecture": "x86_64",
        "bios_date": "12/01/2006",
```

Figure 44: Ansible facts output

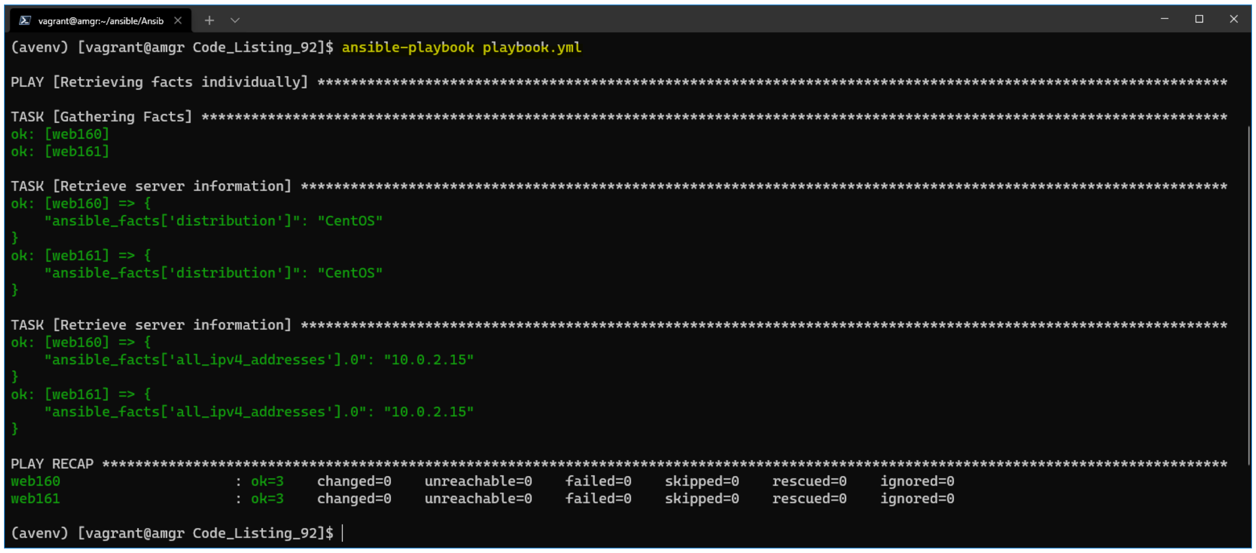
We can certainly obtain the individual values from the returned long list of values.

Let’s say that we would like to retrieve the Linux distribution and the first IP address from the server in question.

Code Listing 92: Retrieving individual values

```
---
- name: Retrieving facts individually
  hosts: webservers
  become: yes
  gather_facts: true

  tasks:
    - name: Retrieve server information
      debug:
        var: ansible_facts['distribution']

    - name: Retrieve server information
      debug:
        var: ansible_facts['all_ipv4_addresses'].0
```

In the output, we can see that the distribution is **`CentOS`** and the private IP address (in this case) is **`10.0.2.15`**. To obtain the first value from an array of values, we used the **`.0`** notation.



```
(avenv) [vagrant@amgr Code_Listing_92]$ ansible-playbook playbook.yml

PLAY [Retrieving facts individually] *************************************************************************************************************

TASK [Gathering Facts] ***************************************************************************************************************************
ok: [web160]
ok: [web161]

TASK [Retrieve server information] ***************************************************************************************************************
ok: [web160] => {
    "ansible_facts['distribution']": "CentOS"
}
ok: [web161] => {
    "ansible_facts['distribution']": "CentOS"
}

TASK [Retrieve server information] ***************************************************************************************************************
ok: [web160] => {
    "ansible_facts['all_ipv4_addresses'].0": "10.0.2.15"
}
ok: [web161] => {
    "ansible_facts['all_ipv4_addresses'].0": "10.0.2.15"
}

PLAY RECAP ***************************************************************************************************************************************
web160                     : ok=3    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
web161                     : ok=3    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

(avenv) [vagrant@amgr Code_Listing_92]$
```

Figure 45: Result of retrieving individual facts

It’s now even more evident how we can use this information in the task, where we can decide whether to install a package given the distribution name of the host, or some other variables that might be useful to base the decision on.

Chapter 14 Ansible Tags

Sometimes we have the requirement of only running one specific task within a plethora of tasks configured in a playbook. In other words, instead of executing the playbook itself, we might choose to run only a part. Ansible enables this scenario with the **`tags`** attribute. Tags are annotations to the task that identify or group them with other tasks.

Let's quickly see an example of how to configure a tag.

Code Listing 93: Playbook with tags

```
---
- name: Tags Playbook
  hosts: localhost
  gather_facts: true
  connection: local
  tasks:
    - name: Display information
      debug:
        msg:
          - "Distro of {{ ansible_facts['hostname'] }}:
{{ ansible_facts['distribution'] }}"
          - "IP of {{ ansible_facts['hostname'] }}:
{{ ansible_default_ipv4.address }}"
      tags: info

    - name: Apply changes
      debug:
        msg: "Some changes executed"
      tags: execute

    - name: Post execution
      debug:
        msg: "Command executed successfully"
      tags: [info, execute]

    - name: Never
      debug:
        msg: "This command has to be explicitly called"
      tags: [never, debug]
```

Through the **`tags`** keyword, we have marked all of the tasks, de facto attaching a label to them. We can see that more than one tag can be assigned at the same time. As mentioned previously, this is very useful if we want to group certain tasks together, and sometimes one given task may belong to more than one group.

There are two special tags defined by Ansible: **`never`** and **`always`**. The **`never`** tag, if specified, will prevent the execution of the tasks, unless this is not explicitly specified to run. On the other hand, **`always`** is the default value of any tag.

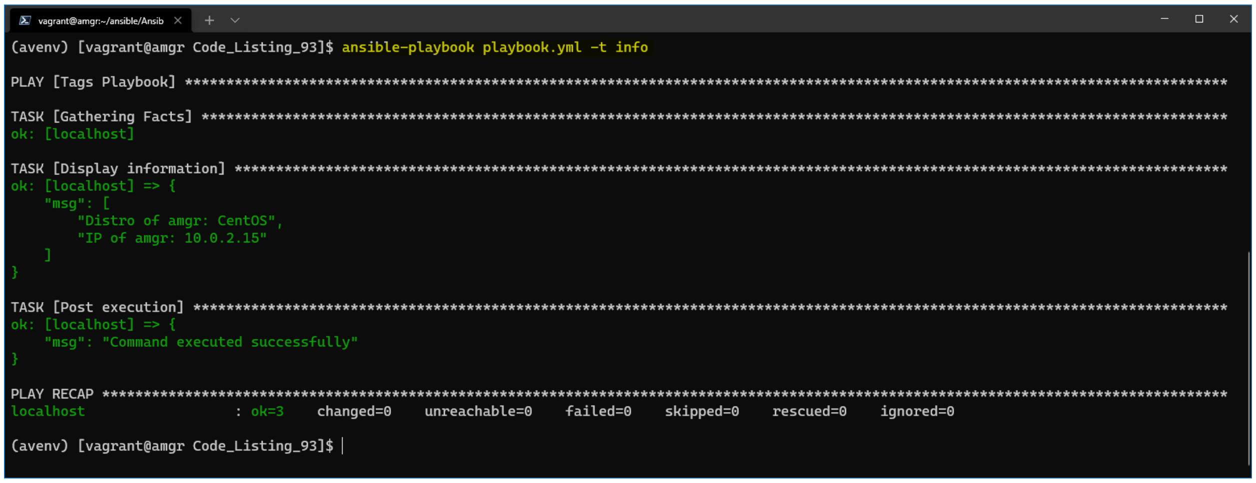
When executing the playbook, we have a few possibilities on how to include or exclude certain tags from being executed

Table 12: Tag command line options

Example	Description
`ansible-playbook p.yml` **`-t all`**	**`all`** is a special keyword that will run all the tasks (except tasks marked as **`never`**).
`ansible-playbook p.yml` **`-t tagged`**	**`tagged`** is a special keyword that will run all of the tasks that have been explicitly tagged (at least one tag).
`ansible-playbook p.yml` **`-t untagged`**	**`untagged`** is a special keyword that will run all of the tasks that have *not* been explicitly tagged (at least one tag).
`ansible-playbook p.yml` **`-t "info, debug"`**	Executes tasks with multiple tags.
`ansible-playbook p.yml` **`--skip-tags info`**	Runs all the tags, but not the one(s) specified.
`ansible-playbook p.yml` **`--list-tags`**	Lists all of the tags defined in the current playbook.
`ansible-playbook p.yml` **`-t info --list-tasks`**	Lists all the tasks that are tagged with the label **`info`**.

Let’s see a few examples.

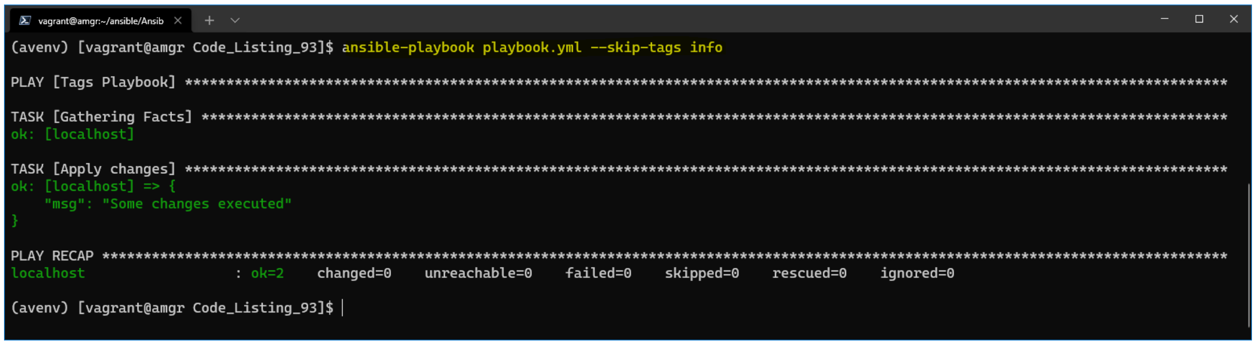
If we set the **playbook.yml** file to run only the tasks labeled **`info`**, only the tasks named **`Display information`** and **`Post execution`** will run, as shown in the following figure.



```
(avenv) [vagrant@amgr Code_Listing_93]$ ansible-playbook playbook.yml -t info

PLAY [Tags Playbook] *************************************************************************************************************

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

TASK [Display information] *******************************************************************************************************
ok: [localhost] => {
    "msg": [
        "Distro of amgr: CentOS",
        "IP of amgr: 10.0.2.15"
    ]
}

TASK [Post execution] ************************************************************************************************************
ok: [localhost] => {
    "msg": "Command executed successfully"
}

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

(avenv) [vagrant@amgr Code_Listing_93]$ |
```

Figure 46: Running tasks tagged "info"

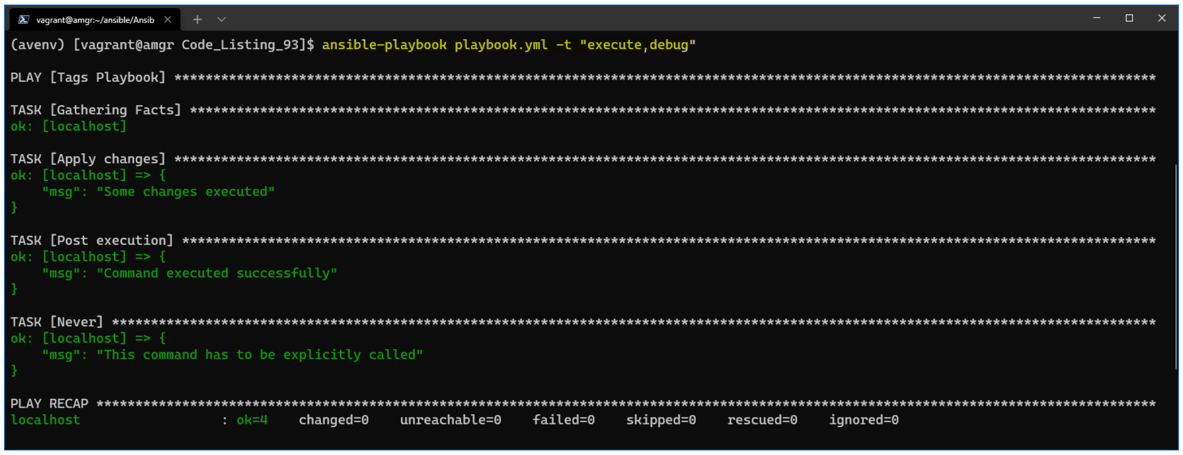
Instead, if we were to run everything but not tags with the info label, then only the **`Apply changes`** task would be executed.



```
(avenv) [vagrant@amgr Code_Listing_93]$ ansible-playbook playbook.yml --skip-tags info

PLAY [Tags Playbook] *************************************************************************************************************

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

TASK [Apply changes] *************************************************************************************************************
ok: [localhost] => {
    "msg": "Some changes executed"
}

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

(avenv) [vagrant@amgr Code_Listing_93]$ |
```

Figure 47: Skipping all tasks tagged "info"

In both cases, we can see that the task named **`never`** was never executed. If we want to execute this task, too, we need to explicitly specify it when running the playbook.



```
(avenv) [vagrant@amgr Code_Listing_93]$ ansible-playbook playbook.yml -t "execute,debug"

PLAY [Tags Playbook] *************************************************************************************************************

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

TASK [Apply changes] *************************************************************************************************************
ok: [localhost] => {
    "msg": "Some changes executed"
}

TASK [Post execution] ************************************************************************************************************
ok: [localhost] => {
    "msg": "Command executed successfully"
}

TASK [Never] *********************************************************************************************************************
ok: [localhost] => {
    "msg": "This command has to be explicitly called"
}

PLAY RECAP ***********************************************************************************************************************
localhost                  : ok=4    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 48: Explicitly executing the “debug” and “execute” tasks

Chapter 15 Ansible Roles

In Chapter 9, we saw how to work with the Ansible Playbook and how to utilize tasks. We are able to automate a large number of processes in this way.

In a more complex environment, we are automating and managing multiple environments or different products, which will undoubtedly increase code redundancy and complexity. At that point, it becomes quite difficult to manage everything in one Ansible playbook file.

Another cool feature that exists in other programming and scripting languages is code reuse, and Ansible in that sense is not any different. When writing code to automate webservers or databases, or managing another kind of host, we would also like to be able to *share* this work with others.

Ansible has created the concept of a *role* to help solve these issues. Each role is basically limited to a particular functionality or desired output, with all the necessary steps to provide that result. You might think of it as a library or module in other programming languages.

The Ansible role:

- Allows code reuse and makes the Ansible projects more manageable.
- Allows the creation of generic code that can be shared between teams or projects.
- Contains a set of (pre)packaged tasks.
- Has to be used within playbook.
- Has a predefined directory structure.
- Is written in YAML, as is the case for playbook or tasks.

Note: The concept of the Ansible role is simple: it is a group of variables, tasks, files, and handlers that are stored in a standardized file structure.

Role’s directory structure

The Ansible role has a predefined and standardized directory structure where files are organized into subdirectories for placing items such as variables, tasks, and handlers.

There are two ways of creating such a structure: manually, or by using the **`ansible-galaxy`** command. As we like automation, we will utilize the command to generate a skeleton folder structure for a role called **`webservice`**.

Code Listing 94: Initializing a role skeleton structure

```
$ ansible-galaxy init webserver
```

When we run the command, a predefined directory structure will be created for us. Figure 49 shows the directory hierarchy.

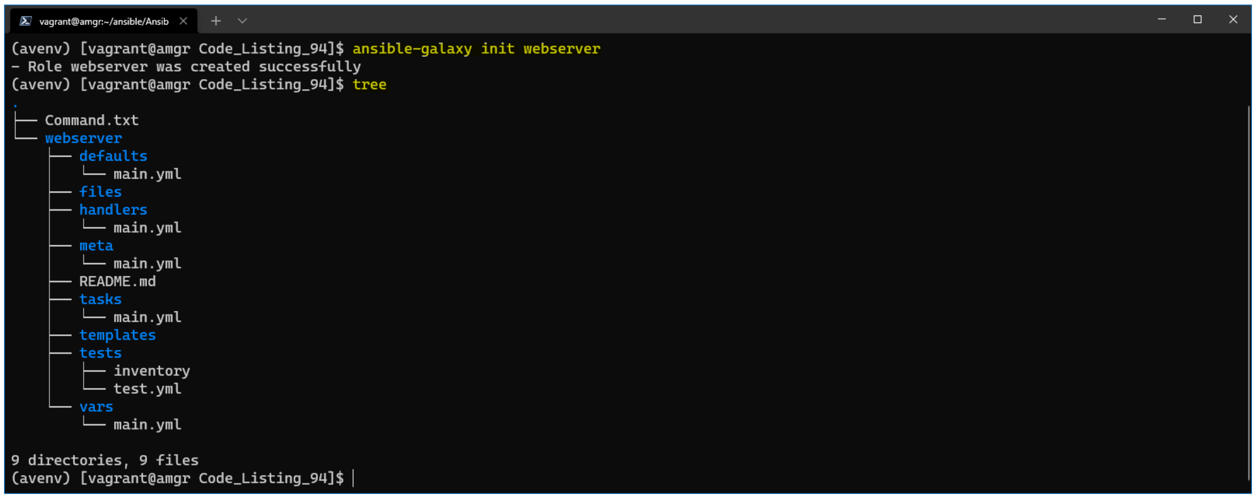

```
(avenv) [vagrant@amgr Code_Listing_94]$ ansible-galaxy init webserver
- Role webserver was created successfully
(avenv) [vagrant@amgr Code_Listing_94]$ tree
.
├── Command.txt
└── webserver
    ├── defaults
    │   └── main.yml
    ├── files
    ├── handlers
    │   └── main.yml
    ├── meta
    │   └── main.yml
    ├── README.md
    ├── tasks
    │   └── main.yml
    ├── templates
    ├── tests
    │   ├── inventory
    │   └── test.yml
    └── vars
        └── main.yml

9 directories, 9 files
(avenv) [vagrant@amgr Code_Listing_94]$ |
```

Figure 49: Role Skeleton creation

We can see that the command has created the skeleton with a few subfolders and files.

Table 13: Role directory structure explained

Directory	Description
`\`	The root directory is named after the role name.
`defaults`	Contains default variables for the role. Variables in this directory have the lowest priority, so they are easy to override.
`files`	Contains (static) files that are to be copied to the remote host.
`handlers`	Contains handler definitions to be used by the role.
`meta`	Contains the general information about the role itself, such as author, description, and license, as well the dependencies to other roles.
`README.md`	Can contain information/documentation about the role.
`tasks`	Contains the main list of steps (tasks) to be executed by the role. Similar to what we define in a playbook.
`templates`	Jinja2 templates referenced by the role tasks.
`Tests` - `inventory` - `test.yml`	The **`inventory`** file and **`test.yml`** playbook that can be used for testing.
`vars`	Variables (with high precedence) used internally by the role.

All of the subdirectories contain a **main.yml** file, which is the default file to be included in the execution pipeline.

There are three ways to start working with roles:

- Create a playbook first, and when it's getting too complex, start translating and porting this code to a role.
- Start working on and creating the role from the beginning. This comes with the experience and the actual need of the application.
- Reuse an already available role, something we will explore in Chapter 16.

MongoDB custom role

In this section, we will automate the MongoDB installation with some prerequisites for a newly created role. The goal of this exercise is to install the MongoDB on the **db** server and install the MongoDB client on the two available webservers (web161, web162).

MongoDB is a NoSQL database, and it's often used as the backend of web applications. The procedure to follow to install the MongoDB on the CentOS server is described on the MongoDB website.

What is important to understand in general is that Ansible just gives a means of automation, but not the actual recipe of how exactly each application works. So, looking into the official documentation is crucial to understanding what to automate.

By reading the official MongoDB documentation, we will see that we need to perform the following operations:

1. Add the **yum** repository, as CentOS doesn't have MongoDB available by default.
2. Install MongoDB.
3. Open Firewall ports to be able to access it from other servers (web server).
4. Start the service.

While there are many other operations that could be added, such as configuring the **ulimit** and other settings for better performance, we will omit those for the sake of brevity.

Let's start by creating the inventory file. In our case, the hostname is called **db** (**192.168.3.199**).

Code Listing 95: Inventory file

```
[database]
db ansible_host=192.168.3.199

[webservers]
web160 ansible_host=192.168.3.160
web161 ansible_host=192.168.3.161
```

In the following **ansible.cfg** code, we can see that there is a **roles_path** property being set to the local folder called **roles**. The **role_path** is defining where Ansible is going to look for the roles by default.

Code Listing 96: ansible.cfg with roles defined

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

With this information, we are now ready to create our role by using the command line tool **`ansible-galaxy`**.

Role creation

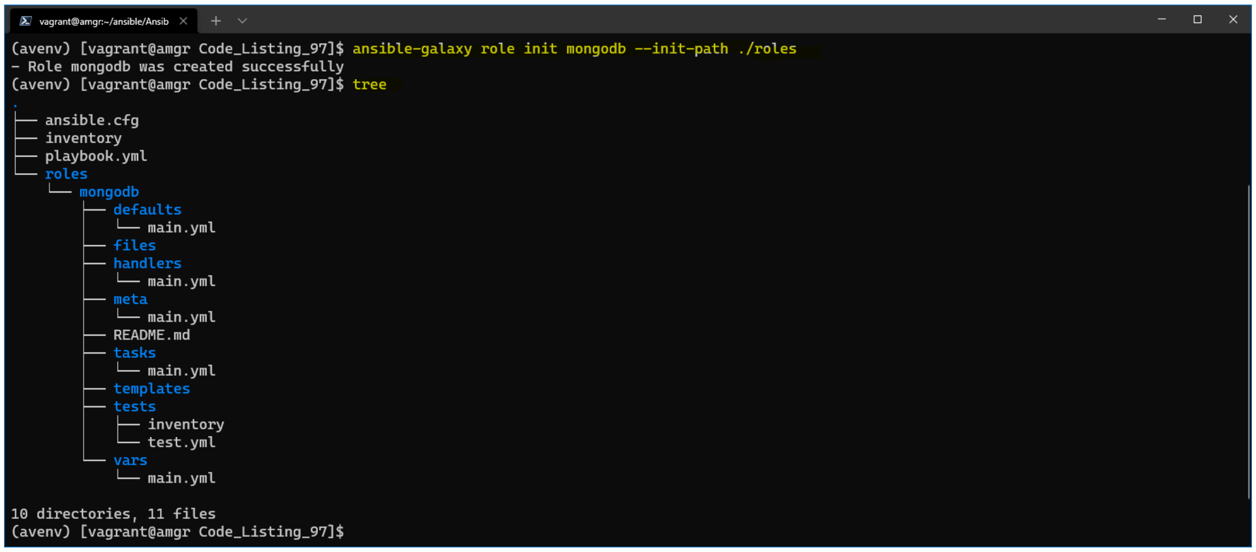
The **`mongodb`** role is going to contain two types of automation: one for installing the **`mongodb server`**, and another one to install the **`mongodb client`**.

Let's create the role called **`mongodb`** and place it under the ./roles folder as defined in the argument **`--init-path`**. This code should be executed in the root folder, where the **ansible.cfg** or **playbook.yml** files are placed; otherwise, please do specify the full path to the roles directory.

Code Listing 97: Code to initiate a skeleton of a role called mongodb

```
$ ansible-galaxy role init mongodb --init-path ./roles
```

As shown in the following figure, we can see that the skeleton of the role has been successfully created under the roles folder.



```
(avenv) [vagrant@amgr Code_Listing_97]$ ansible-galaxy role init mongodb --init-path ./roles
- Role mongodb was created successfully
(avenv) [vagrant@amgr Code_Listing_97]$ tree
.
├── ansible.cfg
├── inventory
├── playbook.yml
└── roles
    └── mongodb
        ├── defaults
        │   └── main.yml
        ├── files
        ├── handlers
        │   └── main.yml
        ├── meta
        │   └── main.yml
        ├── README.md
        ├── tasks
        │   └── main.yml
        ├── templates
        ├── tests
        │   ├── inventory
        │   └── test.yml
        └── vars
            └── main.yml

10 directories, 11 files
(avenv) [vagrant@amgr Code_Listing_97]$
```

Figure 50: Creation of the mongodb role

Variables

Under the **vars** folder, let's open the **main.yml** file and set the following variables.

Code Listing 98: Content of vars/main.yml

```
---
# vars file for mongodb
mongo_db_version: "4.4"
mongo_db_journal_enabled: "false"
mongo_db_server_port: 27017
```

Here we are defining the version of the MongoDB we would like to install, and two more configuration options to be set after the MongoDB installation.

Tasks

Now we can start filling out the tasks by opening the main.yml file under the **tasks** folder. This task is just a bit longer, but we will go through each task and explain it.

Code Listing 99: Main.yml under tasks folder

```
---
- name: Add yum MongoDB repository
  ansible.builtin.template:
    src: mongodb-org.repo.j2
    dest: /etc/yum.repos.d/mongodb-org-{{ mongo_db_version }}.repo
    mode: 0644
  tags: [never, mongodbclient, mongodbserver]

- name: Install MongoDB server
  ansible.builtin.yum:
    name: mongodb-org
    update_cache: yes
    state: present
  tags: [never, mongodbserver]

- name: Install MongoDB client
  ansible.builtin.yum:
    name: mongodb-org-shell
    update_cache: yes
    state: present
  tags: [never, mongodbclient]

- name: Change the MongoDB configuration file
  ansible.builtin.template:
    src: mongod.conf.j2
    dest: /etc/mongod.conf
    mode: 0644
  notify: restart_mongo_db
  tags: [never, mongodbserver]

- name: Start MongoDB service
  ansible.builtin.systemd:
    name: mongod
    state: started
  tags: [never, mongodbserver]

- name: Open Firewall Port 27017
  ansible.posix.firewalld:
    zone: public
    rich_rule: >
      rule family="ipv4" source address="{{ hostvars[item].ansible_host }}"
      port protocol="tcp" port="{{ mongo_db_server_port }}" accept
    permanent: yes
    immediate: yes
    state: enabled
  with_items: "{{ groups['webservers'] }}"
  tags: [never, mongodbserver]
```

The first thing to notice is the file indentation. There is no need to specify the **`tasks`** keyword like we were doing in the playbook, as Ansible will automatically assume that the file under the **tasks\main.yml** file contains tasks.

We can see six tasks defined. Let's explain what they do in detail.

Add yum MongoDB repository

As the CentOS **`yum`** package manager doesn't have the MongoDB repository predefined, we have to add it to the list of available repositories. To achieve this, we are using a template file located in the **templates** folder called mongodb-org.repo.j2. The transformed file is then going to be sent to the default **`yum`** configuration location, which is /etc/yum.repos.d.

Code Listing 100: Content of the file templates/mongodb-org.repo.j2

```
[mongodb-org-{{ mongo_db_version }}]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/redhat/$releasever/mongodb-
org/{{ mongo_db_version }}/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-{{ mongo_db_version }}.asc
```

This file internally uses the variable called **mongo_db_version**. This variable is defined in the vars/main.yml file. This file contains the variables defined for the given role.

After executing the playbook, we should see a transformed file to be delivered into the /etc/yum.repos.d folder on the database server.

Install MongoDB server

This task is responsible for installing the MongoDB (**`mongodb-org`**) package. We do this only after the repository location has been added to **`yum`**. With the state **`present`**, we are telling Ansible to install the package.

This code corresponds as if we were executing the following command on the managed server:

```
$ sudo yum install -y mongodb-org
```

This command will install all of the necessary components of MongoDB.

Install MongoDB client

This task is responsible for installing the MongoDB client package. We do this only after the repository location has been added to **`yum`**. With state **`present`**, we are telling Ansible to actually install the package.

This code corresponds as if we were executing the following command on the managed server:

```
$ sudo yum install -y mongodb-org-shell
```

This command will install all of the necessary components of MongoDB shell.

Change the MongoDB configuration file

After installing MongoDB, and before starting it to run as a service, we are going to perform just a few customizations in the MongoDB configuration file.

Again, we are using a template located under **templates/mongod.conf.j2** that, once transformed, will be sent to the database server at the location **/etc/mongod.conf**.

One thing to notice here is that the change in the configuration file triggers a handler called **`restart_mongo_db`**, which is defined in the **handlers/main.yml** file as follows.

Code Listing 101: Content of the file handlers/main.yml

```
---
# handlers file for mongodb
- name: restart_mongo_db
  ansible.builtin.systemd:
    name: mongod
    state: restarted
  tags: [never, mongodbserver]
```

Open Firewall Port 27017

This task is supposed to run after MongoDB has been installed. The task is responsible for opening the port 27017 on the database server to allow only connections from the web server(s), hence the use of **`groups['webservers']`**. This is important, as in general it's a good practice to secure the system to only those hosts that need to communicate to the database.

The variable **`mongo_db_server_port`** is defined in the **vars/main.yml** file.

Code Listing 102: Content of mongod.conf.j2

```
# mongod.conf

# for documentation of all options, see:
#   http://docs.mongodb.org/manual/reference/configuration-options/

# where to write logging data.
systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb/mongod.log

# Where and how to store data.
storage:
  dbPath: /var/lib/mongo
  journal:
    enabled: {{ mongo_db_journal_enabled }}
#  engine:
#  wiredTiger:

# how the process runs
processManagement:
  fork: true  # fork and run in background
  pidFilePath: /var/run/mongodb/mongod.pid  # location of pidfile
  timeZoneInfo: /usr/share/zoneinfo

# network interfaces
net:
  port: 27017
  bindIp: 127.0.0.1, {{ ansible_all_ipv4_addresses.0 }} # Enter 0.0.0.0,::
to bind to all IPv4 and IPv6 addresses or, alternatively, use the
net.bindIpAll setting.

#security:

#operationProfiling:

#replication:

#sharding:

## Enterprise-Only Options
```

```
#auditLog:

#snmp:
```

Some attention has to be paid to the **`ansible_all_ipv4_addresses.0`** variable. If there are multiple network adapters, this might not work. Make sure to readapt the value to **`ansible_all_ipv4_addresses.0 or .1,`** depending on where the public IP address is kept.

Playbooks

We are going to have two playbooks to be placed in the root directory of our project: **database.yml** and **webservers.yml**. The first will be responsible for the installation and setup of the **`mongodb`** on the **`db`** server, while the other will be responsible for installing the MongoDB client application on the web server, so that we query the database and test what have we done so far.

Code Listing 103: Content of the database.yml playbook

```
---
- name: Installation of the MongoDB database
  hosts: database
  become: yes
  gather_facts: yes

  roles:
    - mongodb
```

We can see that the playbook now is quite simple to read, as we have eliminated the tasks and handlers.

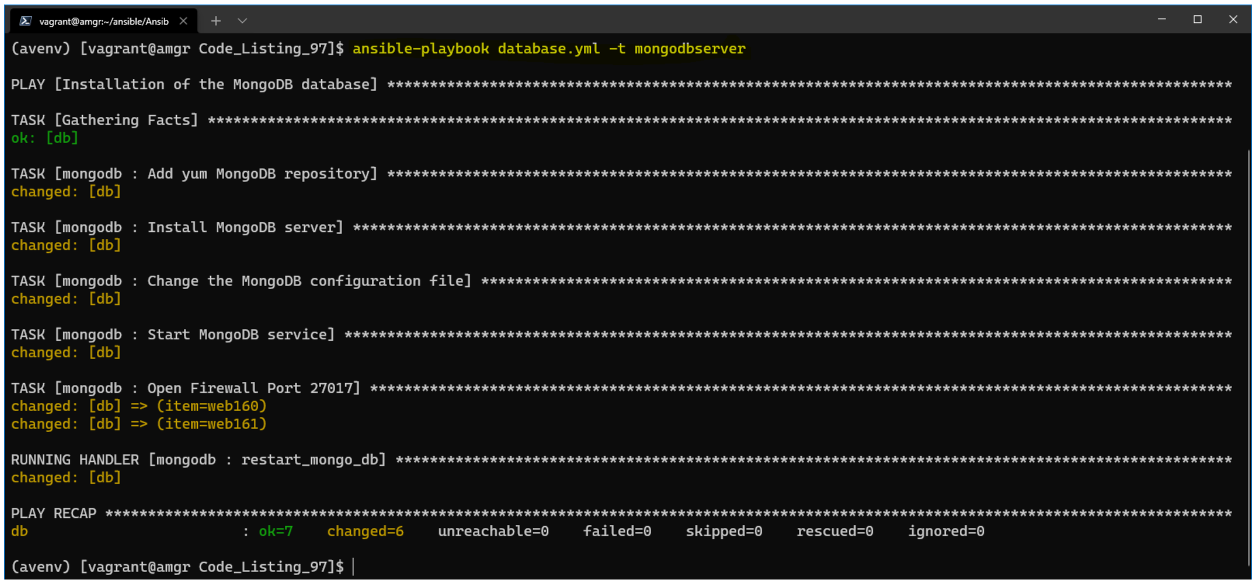
The new keyword we use is **`roles`**. Under the **`roles`**, in general, we can specify more than one role. Roles would be executed in exactly the same order we place them in a list. In our case, we only have one, so we are specifying the **`mongodb`** role.

To run the code against the **`db`** server, however, we will be using the **`tag`** to specify that we only want to install the server (without the client).

Code Listing 104: Execution of the database.yml playbook

```
$ ansible-playbook database.yml -t mongodbserver
```

We can see that by running this code, we only install the **`mongodb`** server.



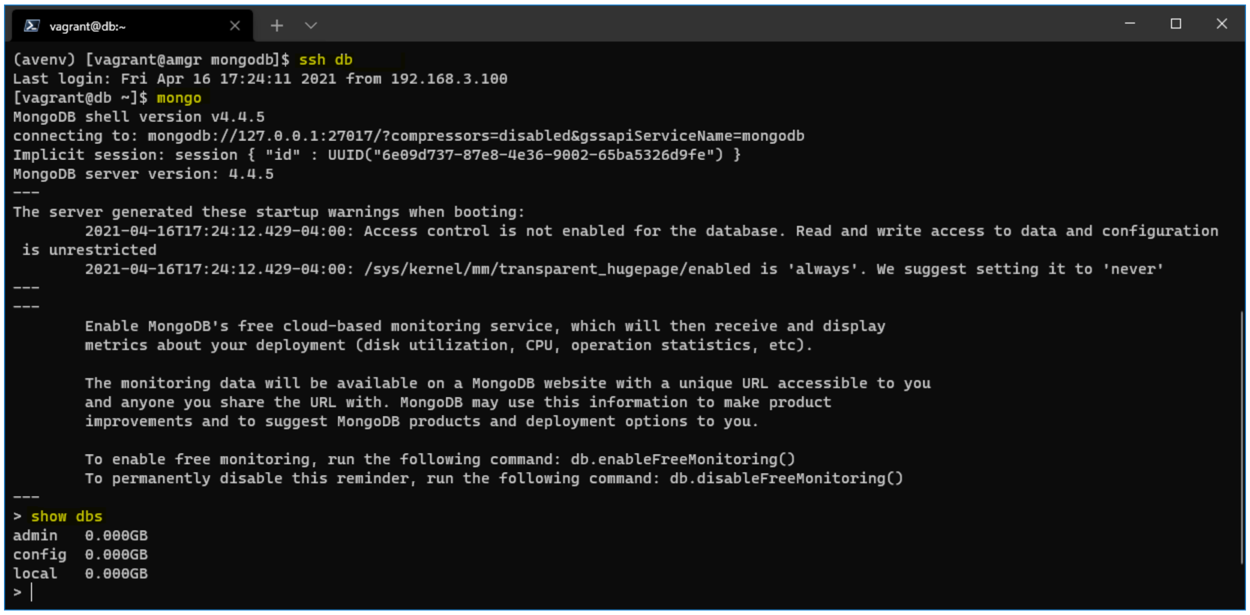
```
(avenv) [vagrant@amgr Code_Listing_97]$ ansible-playbook database.yml -t mongodbserver

PLAY [Installation of the MongoDB database] ************************************************************************************************

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

TASK [mongodb : Add yum MongoDB repository] ************************************************************************************************
changed: [db]

TASK [mongodb : Install MongoDB server] ****************************************************************************************************
changed: [db]

TASK [mongodb : Change the MongoDB configuration file] *************************************************************************************
changed: [db]

TASK [mongodb : Start MongoDB service] *****************************************************************************************************
changed: [db]

TASK [mongodb : Open Firewall Port 27017] **************************************************************************************************
changed: [db] => (item=web160)
changed: [db] => (item=web161)

RUNNING HANDLER [mongodb : restart_mongo_db] ***********************************************************************************************
changed: [db]

PLAY RECAP *********************************************************************************************************************************
db                         : ok=7    changed=6    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

(avenv) [vagrant@amgr Code_Listing_97]$ |
```

Figure 51: Result of the execution of the database.yml playbook

To test that the database has been properly installed, we can directly log in on the **db** server and run the **mongo** command. The **mongo** command is the MongoDB client tool that gets installed with the server.

After launching the **mongo** command without any parameter, we will automatically log into the localhost MongoDB instance. If this is successful, this would mean that MongoDB is up and running.

We are showing an additional command, **show dbs**, which will list all of the currently available databases.



```
(avenv) [vagrant@amgr mongodb]$ ssh db
Last login: Fri Apr 16 17:24:11 2021 from 192.168.3.100
[vagrant@db ~]$ mongo
MongoDB shell version v4.4.5
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("6e09d737-87e8-4e36-9002-65ba5326d9fe") }
MongoDB server version: 4.4.5
---
The server generated these startup warnings when booting:
        2021-04-16T17:24:12.429-04:00: Access control is not enabled for the database. Read and write access to data and configuration
 is unrestricted
        2021-04-16T17:24:12.429-04:00: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest setting it to 'never'
---
---
        Enable MongoDB's free cloud-based monitoring service, which will then receive and display
        metrics about your deployment (disk utilization, CPU, operation statistics, etc).

        The monitoring data will be available on a MongoDB website with a unique URL accessible to you
        and anyone you share the URL with. MongoDB may use this information to make product
        improvements and to suggest MongoDB products and deployment options to you.

        To enable free monitoring, run the following command: db.enableFreeMonitoring()
        To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
> |
```

Figure 52: Checking on the server if MongoDB runs

The next playbook is about installing the webservers.

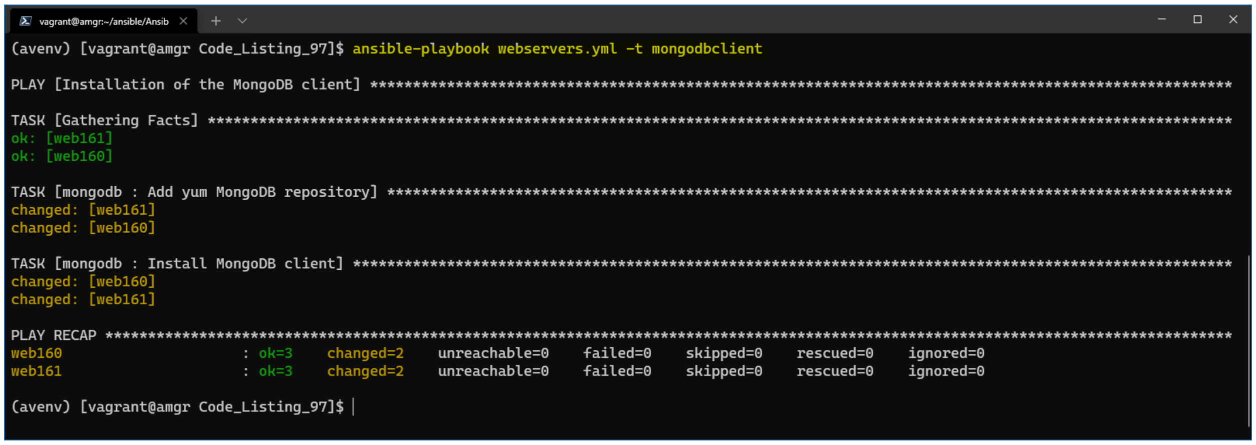
Code Listing 105: Content of webservers.yml playbook

```
---
- name: Installation of the MongoDB client
  hosts: webservers
  become: yes
  gather_facts: yes

  roles:
    - mongodb
```

We can run the playbook:

```
$ ansible-playbook webservers.yml -t mongodbclient
```

And see that both configured web servers were updated, as shown in the following figure.



```
(avenv) [vagrant@amgr Code_Listing_97]$ ansible-playbook webservers.yml -t mongodbclient

PLAY [Installation of the MongoDB client] **************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************************************
ok: [web161]
ok: [web160]

TASK [mongodb : Add yum MongoDB repository] ************************************************************************************************
changed: [web161]
changed: [web160]

TASK [mongodb : Install MongoDB client] ****************************************************************************************************
changed: [web160]
changed: [web161]

PLAY RECAP *********************************************************************************************************************************
web160                     : ok=3    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
web161                     : ok=3    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

(avenv) [vagrant@amgr Code_Listing_97]$
```

Figure 53: MongoDB client installed on webservers

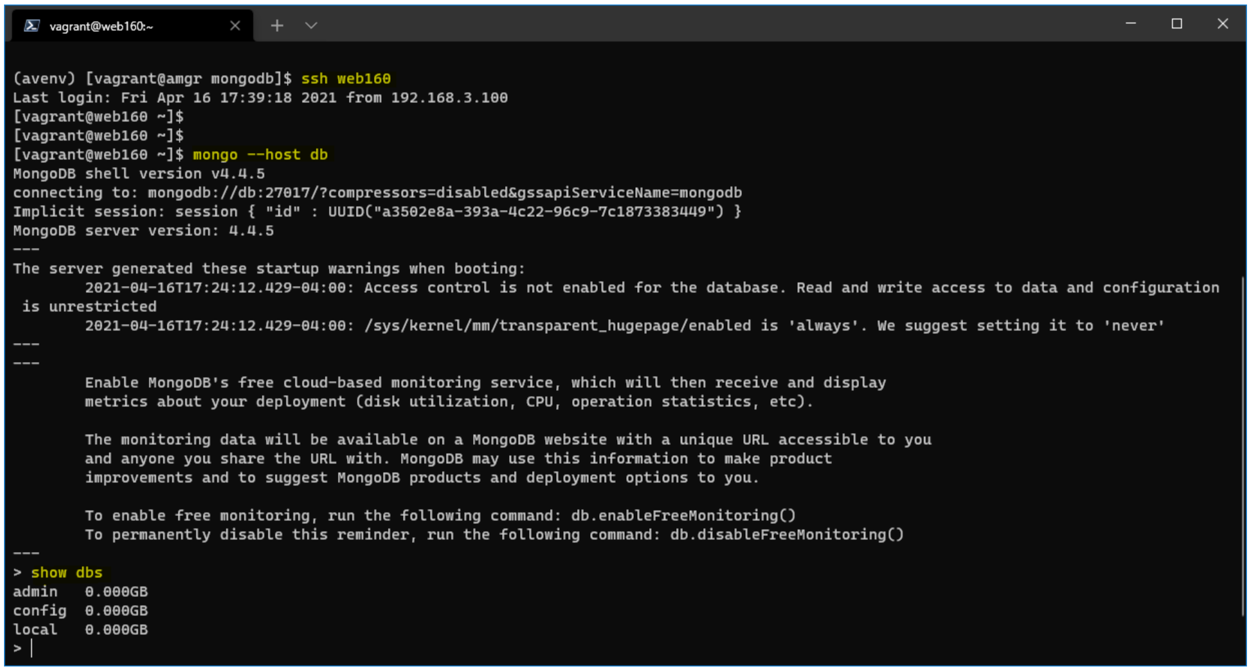
We can now finally test that we can run queries from the web server against the **db**-installed MongoDB.

We need to log in on one of the webservers, let's say **web160**, and run the following command.

Code Listing 106: Mongo client connection

```
$ mongo --host db
```

We can see that we are successfully logged in to the MongoDB server and getting the result by executing a query.



```
(avenv) [vagrant@amgr mongodb]$ ssh web160
Last login: Fri Apr 16 17:39:18 2021 from 192.168.3.100
[vagrant@web160 ~]$
[vagrant@web160 ~]$
[vagrant@web160 ~]$ mongo --host db
MongoDB shell version v4.4.5
connecting to: mongodb://db:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("a3502e8a-393a-4c22-96c9-7c1873383449") }
MongoDB server version: 4.4.5
---
The server generated these startup warnings when booting:
        2021-04-16T17:24:12.429-04:00: Access control is not enabled for the database. Read and write access to data and configuration
 is unrestricted
        2021-04-16T17:24:12.429-04:00: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest setting it to 'never'
---
---
        Enable MongoDB's free cloud-based monitoring service, which will then receive and display
        metrics about your deployment (disk utilization, CPU, operation statistics, etc).

        The monitoring data will be available on a MongoDB website with a unique URL accessible to you
        and anyone you share the URL with. MongoDB may use this information to make product
        improvements and to suggest MongoDB products and deployment options to you.

        To enable free monitoring, run the following command: db.enableFreeMonitoring()
        To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
>
```

Figure 54: MongoDB client connecting to db

Chapter 16 Ansible Galaxy

In the previous chapter, we saw that we can create roles from scratch. We introduced the concept of code sharing and code reuse. We also used the **`ansible-galaxy`** command line to initiate the role skeleton.

Ansible has taken this concept further and created Ansible Galaxy, Ansible's official hub for sharing Ansible content. You can visit the web application here.

Ansible Galaxy is essentially a large public repository of Ansible roles. We can programmatically interact with the repository by using the already mentioned command line tool **`ansible-galaxy`**.

We can use the **`ansible-galaxy`** tool to list, install, or remove existing roles prepackaged and maintained by someone else.

By running the following command, we can see all the operations supported by the tool.

Code Listing 107: Ansible-Galaxy role help command

```
$ ansible-galaxy role --help
```

Table 14 describes some useful commands.

Table 14: Ansible Galaxy useful commands

`ansible-galaxy search <role-name>`	Searches for a role on the Ansible Galaxy platform with a given role name.

`ansible-galaxy install <role-name>`
 	
`example:`
 	
`ansible-galaxy install geerlingguy.apache`
 	
`ansible-galaxy install -r requirements.yml`
	Installs the package from the repository.
	
	By specifying the **`roles_path`** in the ansible.cfg file or using the **`--roles-path`** attribute on the command directly, we can tightly control where the role gets installed.
	
	Multiple roles can be installed at once by specifying the list in a file (requirements.yml) and using the **`-r`** attribute in the command line to specify the file name.

`ansible-galaxy remove <role-name>`	Removes (deletes) the role from the local folder.
`ansible-galaxy info <role-name>`	Retrieves more information about the package itself.

Using roles

In this chapter, we are going to see how we can use the prebuilt roles and integrate them in our solution, where we will:

- Install and configure the httpd server on our two web servers.
- Deploy a simple webpage to the webserver(s).
- Configure the load balancer to route the calls to the web server.

This is a typical setup for a web infrastructure where we expect a higher load and ability to scale out by adding additional web servers over time. This is why we placed a load balancer that can efficiently distribute incoming network traffic across a group of backend servers (in our case, web servers).

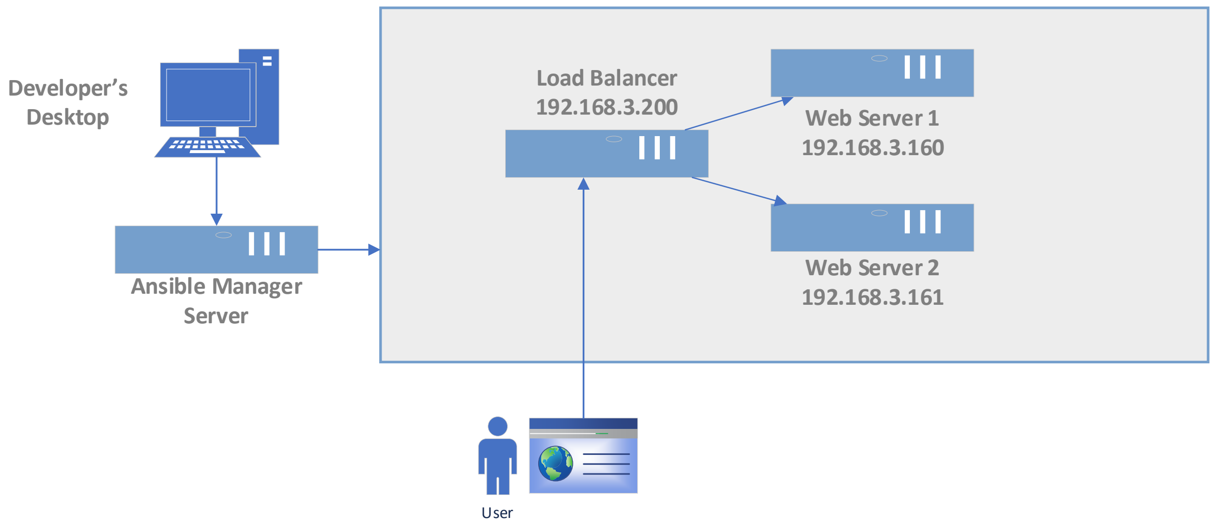

Developer's Desktop
Ansible Manager Server
Load Balancer 192.168.3.200
Web Server 1 192.168.3.160
Web Server 2 192.168.3.161
User

Figure 55: Load-balanced websites

Web server setup

Let's start by creating the inventory file where we will specify the web servers and the load balancer server.

Code Listing 108: inventory file

```
[loadbalancer]
lb ansible_host=192.168.3.200

[webservers]
web160 ansible_host=192.168.3.160
web161 ansible_host=192.168.3.161
```

The ansible.cfg is very basic, and it contains only a reference to the inventory file, something we have seen previously in **roles_path**, which is the directory from where the roles will be installed or loaded.

Code Listing 109: Ansible.cfg file

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

So far, we have enough information to start working on the playbook for the web server. We can now try to work with the role downloaded; in our case, this will be the package called **geerlingguy.apache**. It's a very good practice to navigate to the role project website where we can see some examples of usage and customization. Links to the pages are typically shown in the Ansible Galaxy website, if needed. The number of downloads and the score are a sign of the quality of the package itself.

Alternatively, we could also consult the README.md file bundled with the installation, which contains similar information.

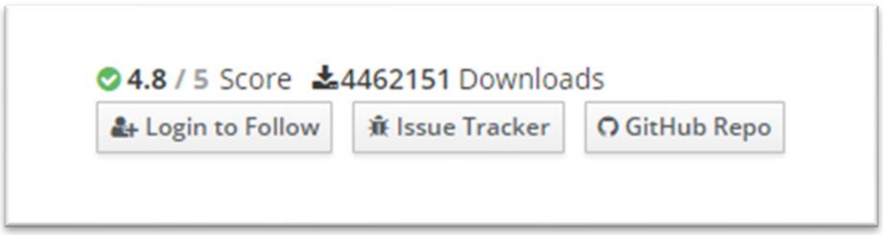


Figure 56: Extract from the galaxy.ansible.com on the role

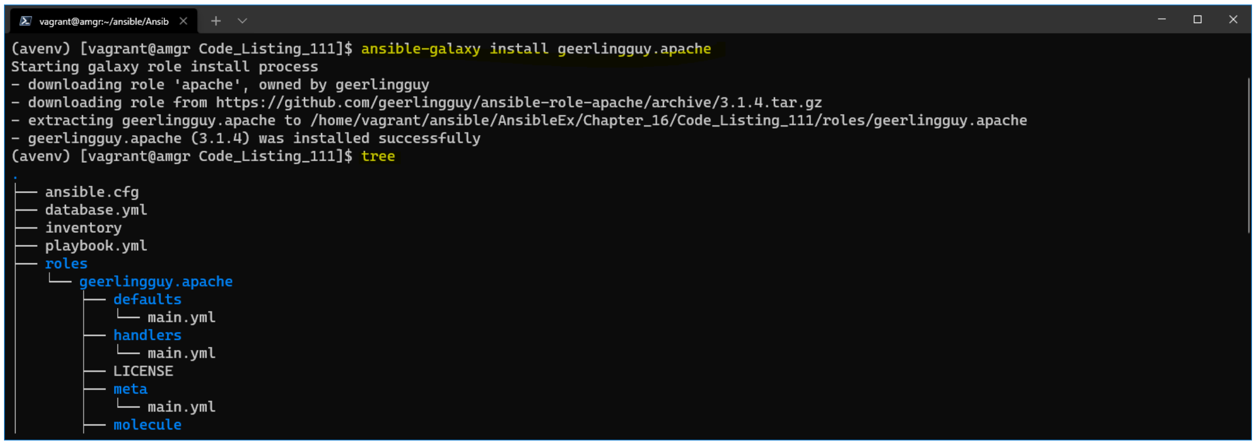
This package is responsible for installing the Apache Server (httpd) on various Linux distributions (RedHat, Debian, etc.), and it's quite generic in what it can do.

Let's start exploring it by installing the role with the following command.

Code Listing 110: Installing apache role

```
$ ansible-galaxy install geerlingguy.apache
```

After running the command, we can see that in the **./roles** folder, we have the fully downloaded role.



```
(avenv) [vagrant@amgr Code_Listing_111]$ ansible-galaxy install geerlingguy.apache
Starting galaxy role install process
- downloading role 'apache', owned by geerlingguy
- downloading role from https://github.com/geerlingguy/ansible-role-apache/archive/3.1.4.tar.gz
- extracting geerlingguy.apache to /home/vagrant/ansible/AnsibleEx/Chapter_16/Code_Listing_111/roles/geerlingguy.apache
- geerlingguy.apache (3.1.4) was installed successfully
(avenv) [vagrant@amgr Code_Listing_111]$ tree
.
├── ansible.cfg
├── database.yml
├── inventory
├── playbook.yml
├── roles
│   └── geerlingguy.apache
│       ├── defaults
│       │   └── main.yml
│       ├── handlers
│       │   └── main.yml
│       ├── LICENSE
│       ├── meta
│       │   └── main.yml
│       ├── molecule
```

Figure 57: Role downloaded and installed in the ./roles directory

In our case, we will simply use the default options without any customizations. Now we can create a **webserver.yml** playbook. As we can see, it has very basic information that we have already seen previously, including hosts against which to act, elevation by using the **become** keyword, and the **role** section itself.

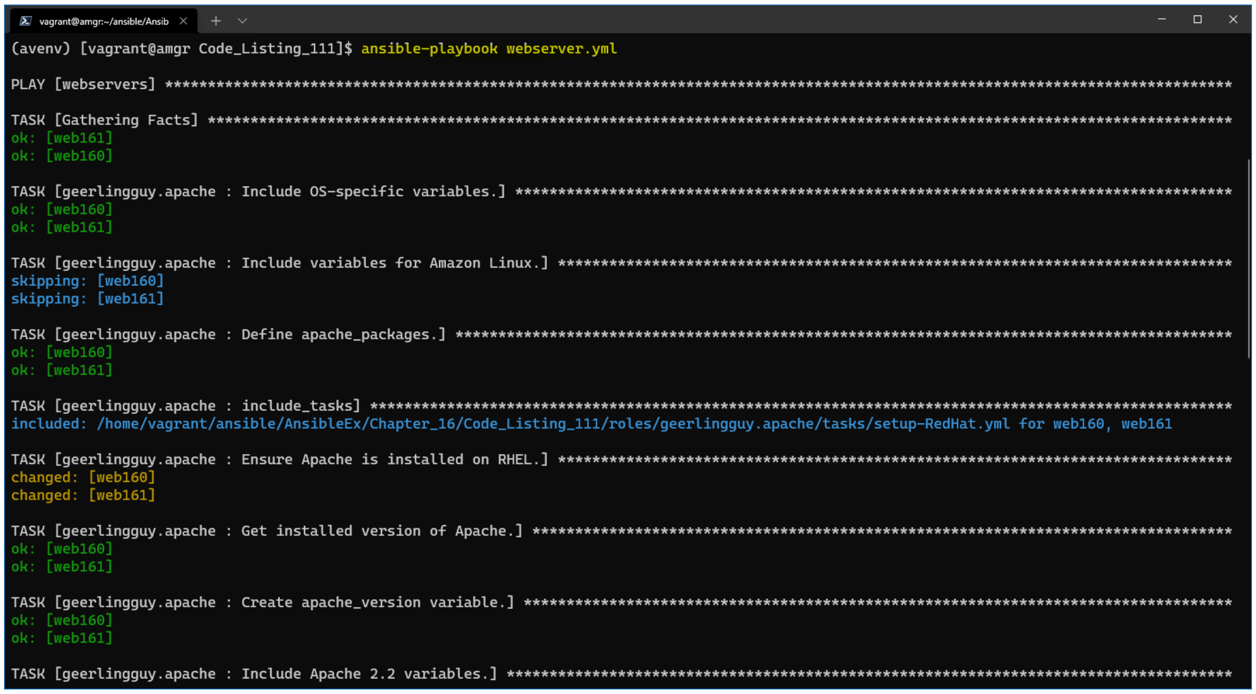
An additional point to mention is that we are also installing the PHP package as a dependency needed to run the website.

Code Listing 111: Webserver.yml playbook

```yaml
---

- hosts: webservers
  become: true

  tasks:
    - name: Enable running PHP code on Apache
      yum:
        name: "{{ item }}"
        update_cache: yes
        state: latest
      loop:
        - php

      notify: restart apache

  roles:
  - role: geerlingguy.apache
```

After running the playbook, we can check if the outcome of the installation is correct.

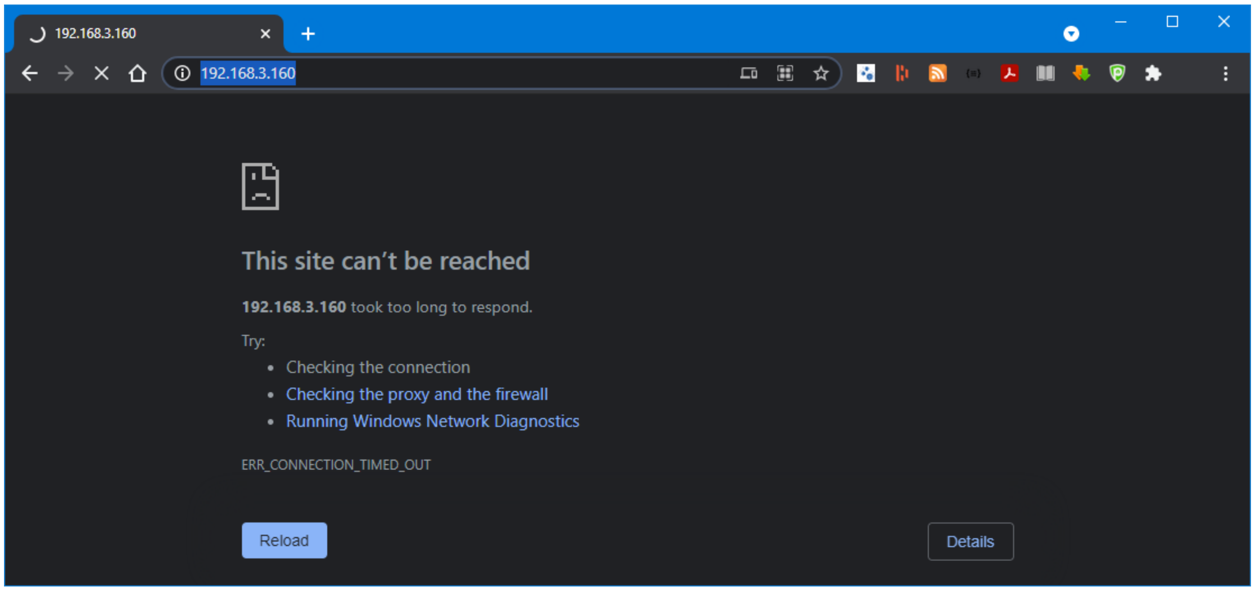
The full output is quite large, but we can see in Figure 58 that the task has executed correctly, and that the script includes the task called **`setup-RedHat.yml`**. We mentioned previously that this role supports various Linux distributions.



```
vagrant@amgr:~/ansible/Ansib
(avenv) [vagrant@amgr Code_Listing_111]$ ansible-playbook webserver.yml

PLAY [webservers] ***************************************************************************************************************

TASK [Gathering Facts] **********************************************************************************************************
ok: [web161]
ok: [web160]

TASK [geerlingguy.apache : Include OS-specific variables.] **********************************************************************
ok: [web160]
ok: [web161]

TASK [geerlingguy.apache : Include variables for Amazon Linux.] *****************************************************************
skipping: [web160]
skipping: [web161]

TASK [geerlingguy.apache : Define apache_packages.] *****************************************************************************
ok: [web160]
ok: [web161]

TASK [geerlingguy.apache : include_tasks] ***************************************************************************************
included: /home/vagrant/ansible/AnsibleEx/Chapter_16/Code_Listing_111/roles/geerlingguy.apache/tasks/setup-RedHat.yml for web160, web161

TASK [geerlingguy.apache : Ensure Apache is installed on RHEL.] *****************************************************************
changed: [web160]
changed: [web161]

TASK [geerlingguy.apache : Get installed version of Apache.] ********************************************************************
ok: [web160]
ok: [web161]

TASK [geerlingguy.apache : Create apache_version variable.] *********************************************************************
ok: [web160]
ok: [web161]

TASK [geerlingguy.apache : Include Apache 2.2 variables.] ***********************************************************************
```

Figure 58: Result of running the webserver.yml playbook

Now we can check if the web server is running properly on both the web servers by calling the http://192.168.3.160 on the desktop machine, and we will notice that result is not being returned.



```
192.168.3.160
192.168.3.160

This site can't be reached

192.168.3.160 took too long to respond.

Try:
  • Checking the connection
  • Checking the proxy and the firewall
  • Running Windows Network Diagnostics

ERR_CONNECTION_TIMED_OUT

Reload                                   Details
```

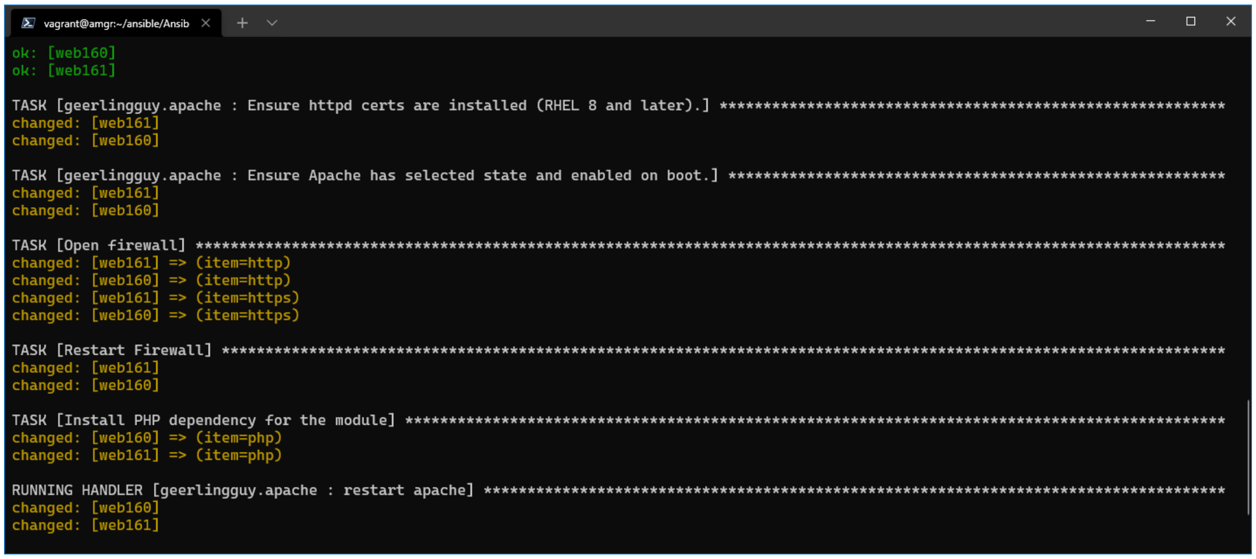
Figure 59: Website not displaying

This is mainly because on the web server, the default firewall ports are not open. So, let's change our playbook by adding two more tasks. Firewall has to be open for both **http** and **https** protocols on the two web servers. The two additional tasks to be added to playbook are as follows.

Code Listing 112: Open Firewall to ports 80/443

```yaml
    - name: Open firewall
      firewalld:
        service: "{{ item }}"
        state: enabled
        immediate: yes
        permanent: yes
      loop:
        - http
        - https

    - name: Restart Firewall
      systemd:
        name: firewalld
        state: restarted
```

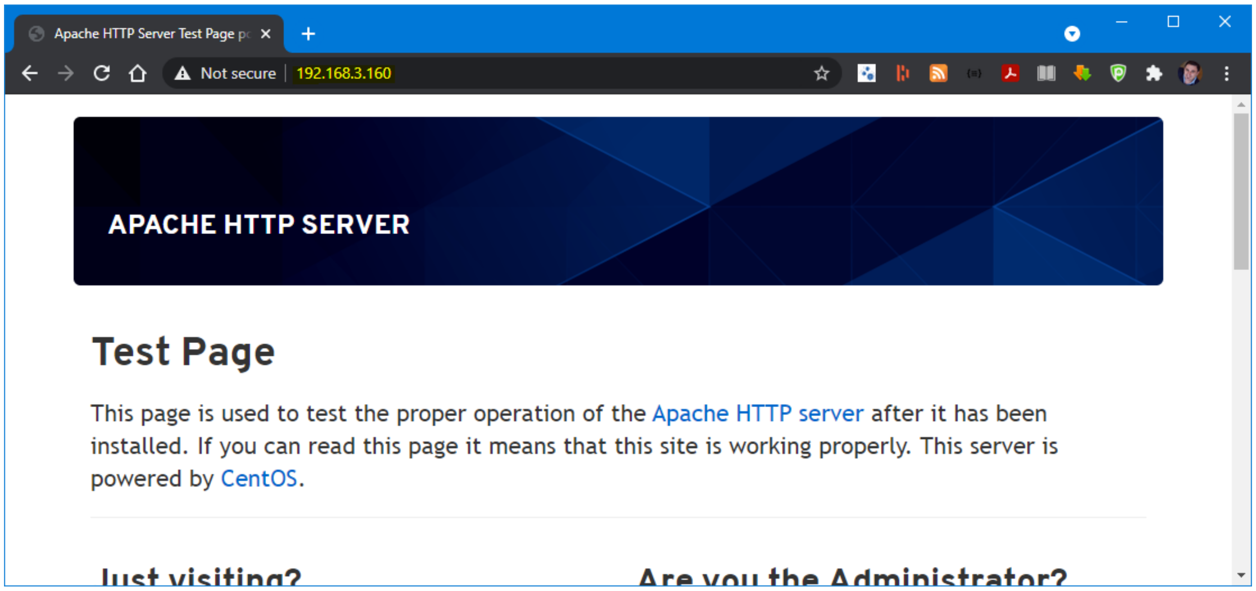
After rerunning the playbook, we can see the two tasks being executed and completing.



```
ok: [web160]
ok: [web161]

TASK [geerlingguy.apache : Ensure httpd certs are installed (RHEL 8 and later).] ***********************************************************
changed: [web161]
changed: [web160]

TASK [geerlingguy.apache : Ensure Apache has selected state and enabled on boot.] **********************************************************
changed: [web161]
changed: [web160]

TASK [Open firewall] ***********************************************************************************************************************
changed: [web161] => (item=http)
changed: [web160] => (item=http)
changed: [web161] => (item=https)
changed: [web160] => (item=https)

TASK [Restart Firewall] ********************************************************************************************************************
changed: [web161]
changed: [web160]

TASK [Install PHP dependency for the module] ***********************************************************************************************
changed: [web160] => (item=php)
changed: [web161] => (item=php)

RUNNING HANDLER [geerlingguy.apache : restart apache] **************************************************************************************
changed: [web160]
changed: [web161]
```

Figure 60: Firewall tasks executed successfully

Now we can also retry checking the browser to see if the page will be displayed properly. Refreshing the page now, we get the result: the default page that Apache Server displays upon installation. We will get the same result for both web servers.


Apache HTTP Server Test Page pc

Not secure | 192.168.3.160

APACHE HTTP SERVER

Test Page

This page is used to test the proper operation of the Apache HTTP server after it has been installed. If you can read this page it means that this site is working properly. This server is powered by CentOS.

Just visiting?   Are you the Administrator?

Figure 61: Browsing the webpage on the web servers

Deploying the website

The next step is to deploy our own website. We are going to do this by first downloading the files from GitHub locally to the Ansible Manager Server, and then uploading them to the web servers. This is one of the ways to perform the deployment. The alternative would be to execute the **`git clone`** directly on the webservers. The way to go would depend on the limitations applied to the network, for instance, as the servers will not always have internet connection.

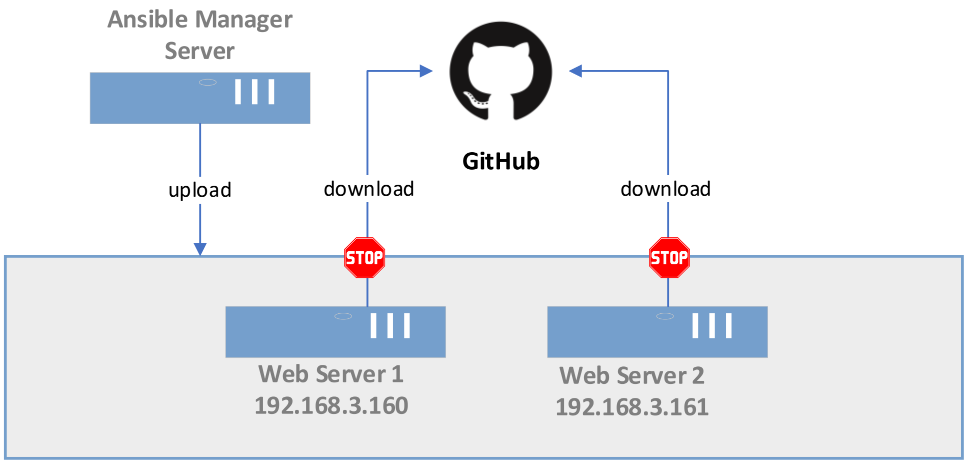


Figure 62: Site deployment

Now we will add two new tasks to the **`webservers.yml`** file to be able to perform the checkout locally, and after the checkout, to deploy the files to the remote server.

Code Listing 113: Code needed to deploy the application (code previously shown is omitted)

```
  …
  vars:
    local_git_directory: site
…
    - name: Check out a git repository on the control node
      local_action:
        module: ansible.builtin.git
        repo: https://github.com/zoranmax/ansible-succinctly-book.git
        dest: "{{ local_git_directory }}"
        force: yes
      run_once: true
      tags: [never, deploy]

    - name: Copy web site
      copy:
        src: "{{ local_git_directory }}/website/"
        dest: /var/www/html
      tags: [never, deploy]

…
```

Checkout

To check out the repository from GitHub, we will use the **`local_action`** module in combination with **`ansible.builtin.git`**, which we haven't seen before.

The **`local_action`** will execute the code on the control node (**`amgr`**) rather than on the managed machine (in our case, web servers). We also had to specify that this task should be run only once: **`run_once: true`**.

The Git module used supports several attributes; here we are only using what is really strictly necessary:

- **`repo`**: Specifies the URL to the Git repository.
- **`dest`**: Defines the destination directory where the code will be checked out.
- **`force`**: If set to **`true`**, will override the folder with new data.

In this example, the repository is a real repository hosted on GitHub, and it only contains two files: **`index.php`** and the **`syncfusion logo`**. The idea is simply to show the concept rather than deploying large websites.

The task will check out the files in the folder defined in the **`local_git_directory`** variable, defined at the top of the playbook, which in our case is a local folder **./site**.

Copy the website

Copying files involves the module called **`copy`**. This module can copy single files or directories:

- **`src`**: File or folder to be copied over.
- **`dest`**: Destination on the managed server.

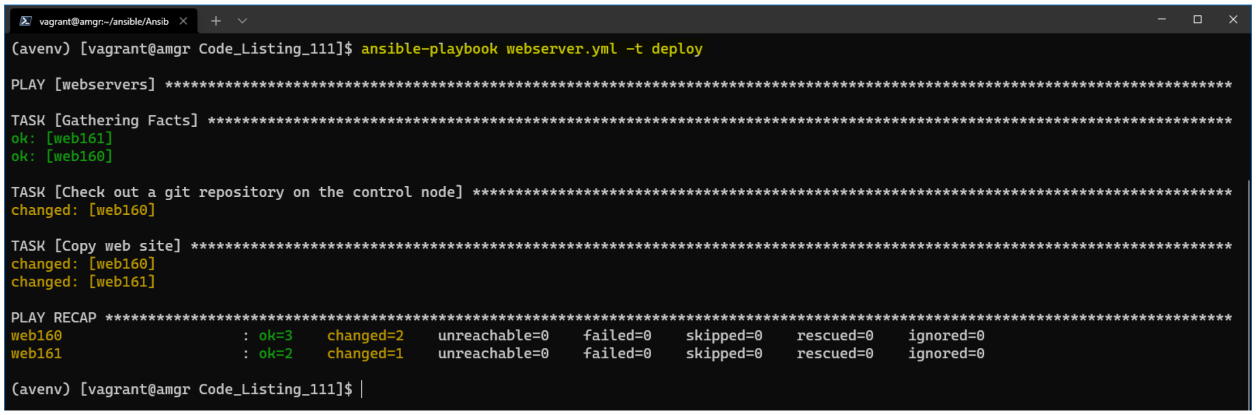
In our case, we will copy the content of the checkout GitHub code we previously placed in the **./site** folder to the default Apache Server folder where the website is hosted: **/var/www/html**.

Running the code

Both deployment tasks have been marked with the tags [**`never`**, **`deploy`**], so they would never be invoked unless we explicitly specify the tag to run, which we will be doing this time.

```
$ansible-playbook webserver.yml -t deploy
```

Once executed, the result will look like the following.



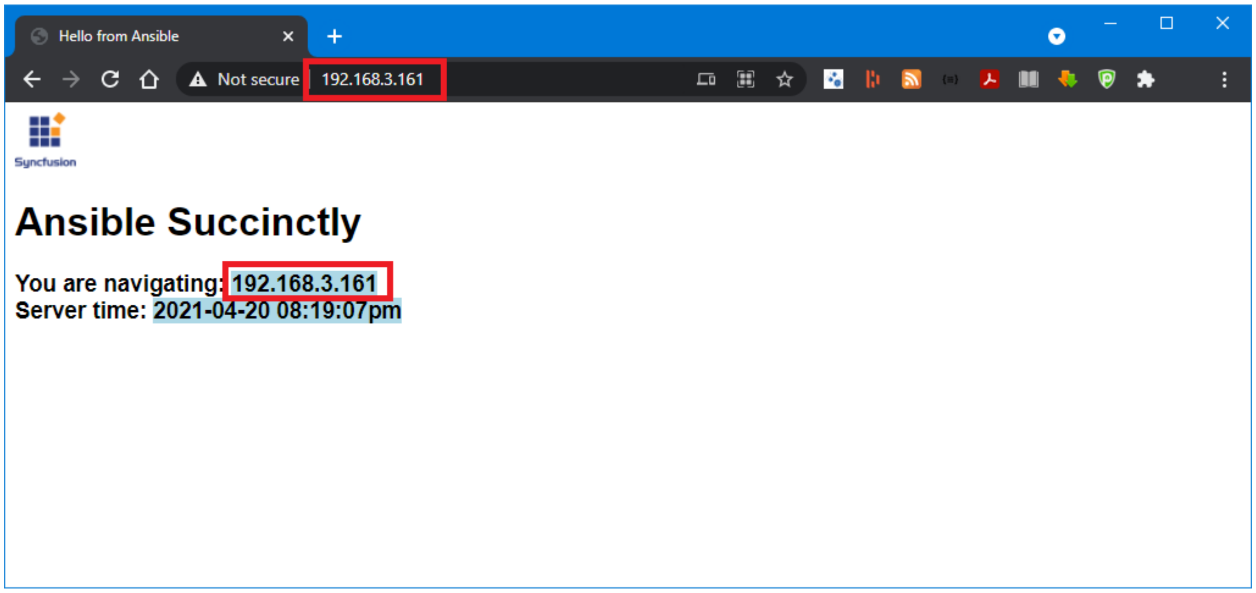
```
(avenv) [vagrant@amgr Code_Listing_111]$ ansible-playbook webserver.yml -t deploy

PLAY [webservers] ************************************************************************************************************************

TASK [Gathering Facts] *******************************************************************************************************************
ok: [web161]
ok: [web160]

TASK [Check out a git repository on the control node] ************************************************************************************
changed: [web160]

TASK [Copy web site] *********************************************************************************************************************
changed: [web160]
changed: [web161]

PLAY RECAP *******************************************************************************************************************************
web160                     : ok=3    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
web161                     : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

(avenv) [vagrant@amgr Code_Listing_111]$
```

Figure 63: Result—deploying website

Now that we have deployed the simple website to both servers, we can navigate through the browser and check if the deployment was successful.

We can now open the URL directly from the Windows desktop and navigate to http://192.168.3.160 or http://192.168.3.161, which are the two IP addresses of the web servers we originally set up.

The page should be displayed as shown in the following image.



Ansible Succinctly

You are navigating: 192.168.3.161
Server time: 2021-04-20 08:19:07pm

Figure 64: Browser showing the deployed page

Load balancer example

So far, we have seen how to configure a simple website by using Ansible and Apache Server, and how to deploy the application.

The next step is to configure the load balancer (or reverse-proxy).

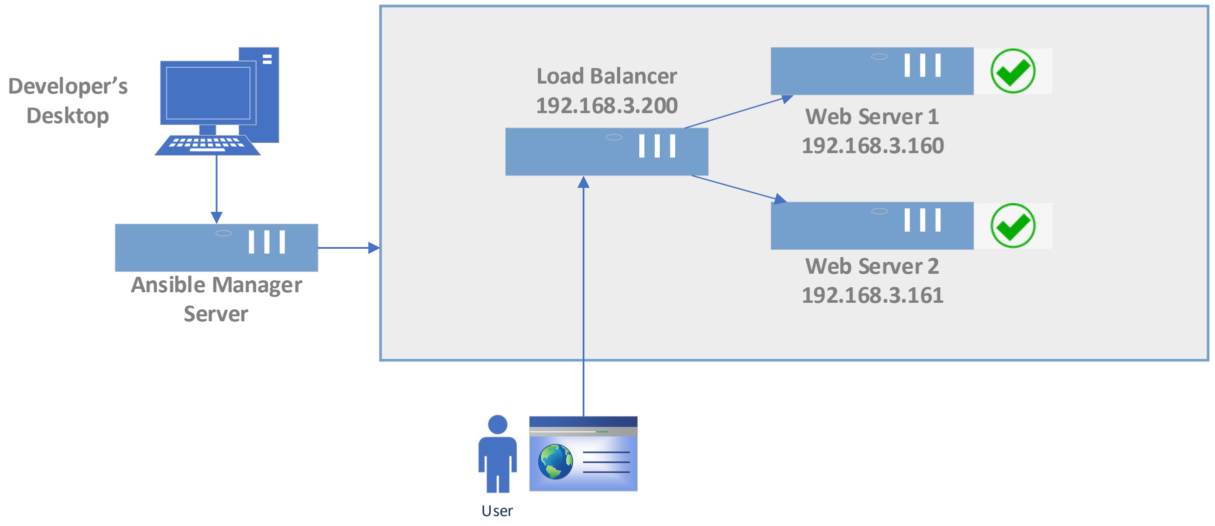

Developer's Desktop
Ansible Manager Server
Load Balancer 192.168.3.200
Web Server 1 192.168.3.160
Web Server 2 192.168.3.161
User

Figure 65: Load balancer (reverse proxy) setup

There are many great software load balancers that can be used for this purpose, such as Nginx, HAProxy, and Citrix ADC.

In our example, we are going to use HAProxy, which is an open-source product, so we can also benefit from the fact that it is free.

We can reuse the same directory where we defined the playbook for the web server, and download the role **`geerlingguy.haproxy`**.

We are already familiar with the command, so we can simply run it.

Code Listing 114: Installing geerlingguy.haproxy

```
$ansible-galaxy install geerlingguy.haproxy
```

The next step is to create a new playbook called **load_balancer.yml** with the following content.

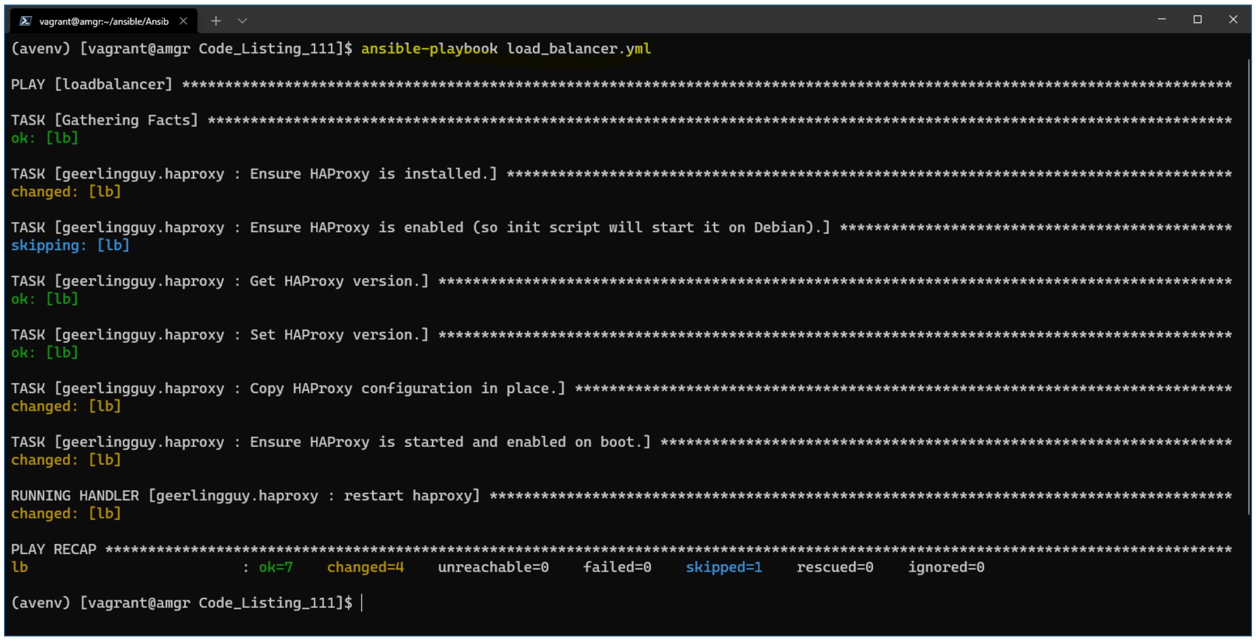
Code Listing 115: Load_balancer.yml playbook

```
---
- hosts: loadbalancer
  become: true
  vars:
    haproxy_backend_balance_method: 'roundrobin'
    haproxy_backend_servers:
      - name: webserver1
        address: 192.168.3.160:80
      - name: webserver2
        address: 192.168.3.161:80

  tasks:
    - name: Open firewall
      firewalld:
        service: "{{ item }}"
        state: enabled
        immediate: yes
        permanent: yes
      loop:
        - http
        - https

    - name: Restart Firewall
      systemd:
        name: firewalld
        state: restarted
  roles:
  - role: geerlingguy.haproxy
```

By looking at the documentation of the role on GitHub, we can see that it has many configuration options. By checking the **`defaults/main.yml`**, we can see all available options. We are going to configure the basic ones to let us perform the work:

- **`haproxy_backend_balance_method`**: Defines the algorithm the HAProxy software is going to use, such as roundrobin, leastconn, or source. We will use roundrobin in this example. roundrobin will just pick the next server and start over at the top of the list.
- **`haproxy_backend_servers`**: Sets the list of servers that would be included in the load-balanced list. This means the servers in this list will be handled by the load balancer.

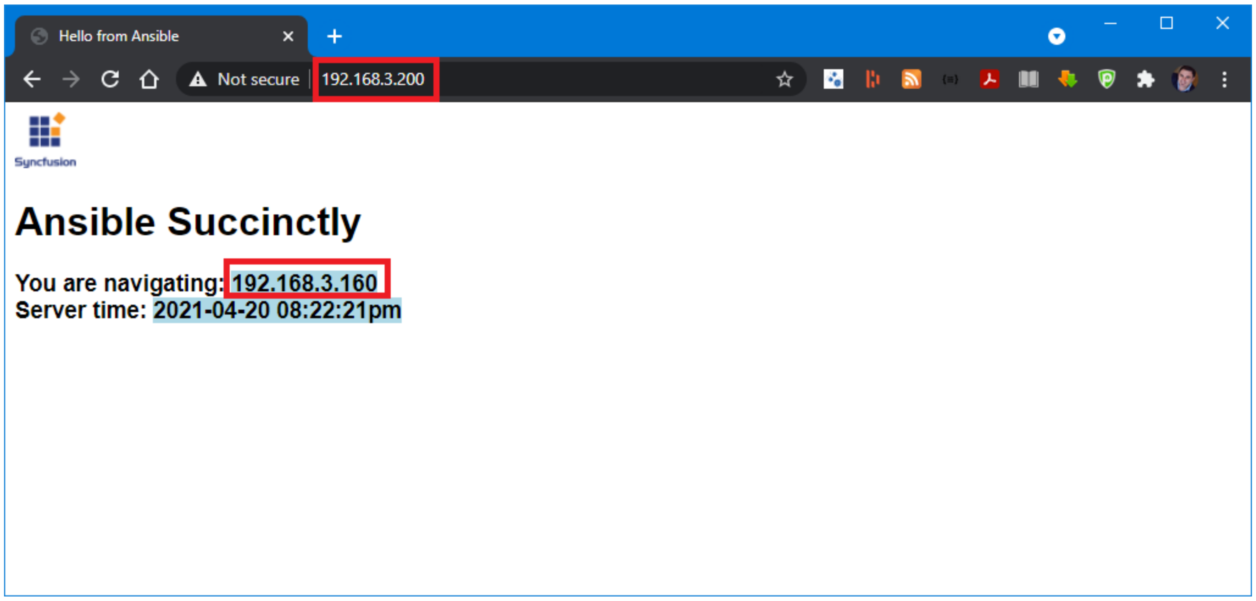
By running the playbook **`load_balancer.yml`**, we get the following result.



```
(avenv) [vagrant@amgr Code_Listing_111]$ ansible-playbook load_balancer.yml

PLAY [loadbalancer] ****************************************************************************************************************

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

TASK [geerlingguy.haproxy : Ensure HAProxy is installed.] **************************************************************************
changed: [lb]

TASK [geerlingguy.haproxy : Ensure HAProxy is enabled (so init script will start it on Debian).] ***********************************
skipping: [lb]

TASK [geerlingguy.haproxy : Get HAProxy version.] **********************************************************************************
ok: [lb]

TASK [geerlingguy.haproxy : Set HAProxy version.] **********************************************************************************
ok: [lb]

TASK [geerlingguy.haproxy : Copy HAProxy configuration in place.] ******************************************************************
changed: [lb]

TASK [geerlingguy.haproxy : Ensure HAProxy is started and enabled on boot.] ********************************************************
changed: [lb]

RUNNING HANDLER [geerlingguy.haproxy : restart haproxy] ****************************************************************************
changed: [lb]

PLAY RECAP *************************************************************************************************************************
lb                         : ok=7    changed=4    unreachable=0    failed=0    skipped=1    rescued=0    ignored=0

(avenv) [vagrant@amgr Code_Listing_111]$
```

Figure 66: Installation of the HAProxy on the load balancer server

Everything seems to be working okay. To test that the load balancer (**`lb 192.168.3.200`**) will return any data, we can again browse the content by using the browser by navigating to http://192.168.3.200. Now we can clearly see that the HAProxy is redirecting the calls to one of the servers we previously configured.



Hello from Ansible

Not secure 192.168.3.200

Ansible Succinctly

You are navigating: 192.168.3.160
Server time: 2021-04-20 08:22:21pm

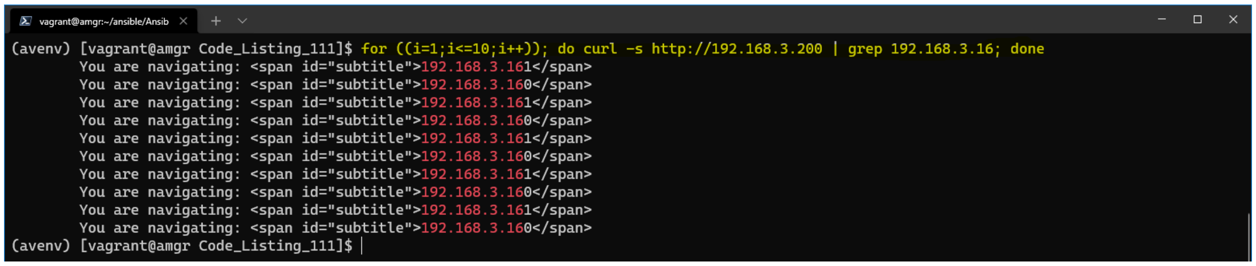
Figure 67: HAProxy using the content from web160 (192.168.3.160)

To demonstrate that the reverse-proxy will redirect the URL in a round-robin way, we can also use the **`curl`** command, and issue 10 calls in a sequence by running the following bash command on the **`amgr`** server.

Code Listing 116: Call the reverse-proxy URL 10 times in a sequence

```
$for ((i=1;i<=10;i++)); do curl -s http://192.168.3.200 | grep 192.168.3;
done
```

In the result, we will clearly see that the servers are going to be selected alternatively.



```
(avenv) [vagrant@amgr Code_Listing_111]$ for ((i=1;i<=10;i++)); do curl -s http://192.168.3.200 | grep 192.168.3.16; done
        You are navigating: <span id="subtitle">192.168.3.161</span>
        You are navigating: <span id="subtitle">192.168.3.160</span>
        You are navigating: <span id="subtitle">192.168.3.161</span>
        You are navigating: <span id="subtitle">192.168.3.160</span>
        You are navigating: <span id="subtitle">192.168.3.161</span>
        You are navigating: <span id="subtitle">192.168.3.160</span>
        You are navigating: <span id="subtitle">192.168.3.161</span>
        You are navigating: <span id="subtitle">192.168.3.160</span>
        You are navigating: <span id="subtitle">192.168.3.161</span>
        You are navigating: <span id="subtitle">192.168.3.160</span>
(avenv) [vagrant@amgr Code_Listing_111]$
```

Figure 68: Servers returning the result

Final Words

I would like to thank you for reading this book—I really hope you enjoyed the content and the examples. Ansible has been growing over the last few years into a very mature product, and it is production-ready.

I tried to present to you the basic (but still very useful) use cases that you will encounter in your professional life while remaining true to the *Succinctly* name.

This book should give you a good foundation to get started with Ansible, and the rest of the discovering should be an evolutive process on your side. The official Ansible documentation is another helpful resource with many examples.

One important thing to get from this book is that first you need to understand *what* you are trying to automate, understand the technology (web server, operating system, database, infrastructure, etc.), and then *how* to automate it by using Ansible.

Typically, you will spend more time investigating the configuration of a particular software component rather than how to automate it with Ansible.

I encourage you to use Ansible in your next automation project.