Targeting Multiple Distributions

So far, throughout the previous chapters, we have been targeting a single operating system, Ubuntu, when running on our playbooks.

This chapter will examine how to work with multiple Linux distributions within the same roles and playbooks.

We will take the WordPress playbook and the roles we created in [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*, and do the following:

* Discover what the difference is between our two target operating systems
* Look at and implement our WordPress roles, making them work on both target operating systems
* Discuss and apply best practices for targeting multiple distributions

The chapter covers the following topics:

* Debian and Red Hat
* Multi-distribution considerations
* Adapting the roles
* Running the playbook

Technical requirements

Given that we will be launching two different operating systems, we will be changing the approach that we have taken in previous chapters and launching a pair of virtual machines in a cloud provider rather than two different virtual machines on our local machines.

The primary reason for this is that Multipass only really supports Ubuntu machines as it was created by Canonical, the creators and maintainers of Ubuntu, to give people a quick, easy, and consistent way to launch an Ubuntu virtual machine across multiple host platforms.

As we will be looking at automating cloud deployments in [*Chapter 9*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/9), *Moving to the Cloud*, we won’t use Ansible to deploy the cloud resources for this chapter.

For this chapter, I would recommend using a cloud provider such as **DigitalOcean** (https://www.digitalocean.com/) or **Linode** (http://www.linode.com/), both of whom support the operating systems we will be covering in this chapter and whose virtual machine costs start at around 5 USD per month.

Important note

This chapter will not cover how to launch virtual machines; if you are following along, please review your chosen cloud provider’s documentation. Additionally, for the full working code, please see the GitHub repository at https://github.com/PacktPublishing/Learn-Ansible-Second-Edition.

Debian and Red Hat

This is where the world of Linux operating systems can get slightly confusing. Although we launch Ubuntu 22.04 and Rocky Linux 9 virtual machines to run our playbooks against, we will reference Debian and Red Hat within the playbook code.

Why is that? The reason behind this lies in the lineage of Linux distributions. Ubuntu is a descendant of the Debian operating system, inheriting its package management system and many other features. Similarly, Rocky Linux is a descendant of Red Hat, designed to be a downstream, bug-for-bug compatible release with **Red Hat Enterprise Linux** (**RHEL**).

So, when we mention Debian and Red Hat in our playbooks, we’re referring to the fundamental bases from which our two operating systems, Ubuntu and Rocky Linux, have evolved.

In practical terms, the playbook code will often check the underlying distribution type to determine how to proceed with specific tasks. For example, the commands to install a software package on a Debian-based system such as Ubuntu might differ from those on a Red Hat-based system such as Rocky Linux.

Debian-based systems use the Debian package management system, with dpkg as the core utility, and often utilize either apt or apt-get, or in some cases all of them, for user-friendly interactions.

Red Hat-based systems employ the RPM package management system, using rpm as the core utility, often complemented by yum or its successor dnf for a more user-friendly interface for managing packages.

There are other differences, such as Debian and Red Hat-based systems that have different directory structures and configuration file locations, which can affect the system administration that we must consider in our playbook roles.

The biggest, at the time of writing, and most relevant difference between the two is licensing.

Debian is known for its strict adherence to free software principles. In contrast, Red Hat-based systems may incorporate more proprietary or closed-source software, especially in the case of Red Hat Enterprise Linux’s commercial enterprise distribution of Red Hat.

This came to a head in June 2023 when Red Hat altered its terms, ceasing the public availability of RHEL’s source code and restricting access solely to customers.

This move impacted downstream projects, relying on RHEL source code to create compatible distributions such as Rocky Linux. The change means that only customers bound by contracts preventing code sharing can access RHEL source code, aligning with the GPL license’s terms, which mandates source code availability only for binary users, who are, essentially, the paying customers in this scenario.

At the time of writing, the fallout from this change is still being felt, and the dust is still settling, although it does seem like distributions such as Rocky Linux have found ways of being compliant; see the *Further reading* section at the end of this chapter for more information.

So, back to our playbook, by referencing either (or both) Debian and Red Hat in the code, we create more adaptable roles that can handle different Linux distributions and their derivatives consistently.

Multi-distribution considerations

Looking at each of the Ansible built-in modules used in the three roles, stack\_install, stack\_config, and wordpress, we are using a few that will not work on our newly introduced Rocky Linux box.

Let’s quickly work through each module and consider what we need to change or take into account when targeting two different distributions.

The Stack Install role

This role uses the following built-in modules:

* ansible.builtin.apt
* ansible.builtin.apt\_key
* ansible.builtin.apt\_repository

We use these modules to update our operating system, add the NGINX mainline repository, and install all the packages we require for our WordPress installation.

As these modules all deal with package management, we won’t be able to reuse any of these tasks, meaning that we will need to split the role into two parts: one that deals with Debian-based systems and the other for Red Hat systems.

Additionally, we won’t be able to reuse the variables, as there are subtle differences in the package names between the two distributions.

This means that our best approach to this role is to use two different sets of tasks depending on the distribution Ansible is targeting. Luckily, there are built-in Ansible modules that make this approach simple. After reviewing the modules in the two remaining roles, we will cover these in the next section.

The Stack Config role

This role is slightly different from the previous one in that we don’t need to split the tasks into two here; most of the tasks will work across both our Linux distributions.

This means that the tasks which make use of the following modules won’t need any changes:

* ansible.builtin.group: Creating a group is the same for both distributions
* ansible.builtin.user: Creating a user is the same for both distributions
* ansible.builtin.template: This only renders and copies files to the target hosts
* ansible.builtin.file: This only copies files to the target hosts
* ansible.builtin.copy: This only copies files on the target hosts
* ansible.builtin.lineinfile: This only searches for text and, if required, updates it within the files on the target hosts
* ansible.builtin.service: This is supported on both distributions
* ansible.builtin.stat: Only checks for the presence of a file on the host’s file system
* ansible.builtin.mysql\_user: As this interacts with the database service, it is distribution agnostic
* ansible.builtin.mysql\_db: As with the previous task, it interacts with the database service

This list is mostly true; however, the file paths will change between the two distributions.

Still, as we already mentioned in [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*, when we looked at the variables for the Stack Config role, we are referencing files that contain the variables we want to load into the playbook run, so we will need to load in an additional set of variables for the distribution as well as the standard ones.

We will need to execute some additional tasks as part of adding the second distribution. Some Red Hat distributions come with a firewall enabled out of the box and SELinux enabled, so we will need to perform some Red Hat-only tasks at the end.

**SELinux**, or to give it its full name, **Security-Enhanced Linux**, is a security module of the Linux kernel that provides a mechanism for supporting access control security policies.

However, we can keep these tasks within the main.yml file rather than loading a different set of tasks by getting creative with the conditions when calling the tasks.

The WordPress role

As the previous two roles have already installed and configured everything that we need to run our WordPress installation, this role is entirely distribution agnostic, and we don’t need to make any changes to the tasks within the role. If you remember, in [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*, when we ran the command to configure WordPress, we set the following fact:

- name: "Set a fact for the wordpress domain"

  ansible.builtin.set\_fact:

    wordpress\_domain: "{{ ansible\_ssh\_host }}"

    os\_family: "{{ ansible\_distribution }} {{ ansible\_distribution\_version }}"

CopyExplain

This used the facts gathered by Ansible when first connecting to the host to figure out which distribution and version we were connecting to; we will expand on this logic as we dive deeper into the changes outlined in this section for the Stack Install and Config roles.

Adapting the roles

So, how do we build the logic into our roles to execute only certain parts of them on different operating systems? As we know, the package names will be different. How do we define different sets of variables per operating system?

Operating system family

We have looked at the ansible.builtin.setup module in [*Chapter 1*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/1), *Installing and Running Ansible*; this module gathers facts about our target hosts.

One of these facts is ansible\_os\_family; this tells us the type of operating system we are running.

To demonstrate this, I have launched two hosts, one running Ubuntu 22.04, and the second running Rocky Linux 9 as its operating system. I have created an inventory file which looks like the following:

RedHat ansible\_host=178.79.178.78.nip.io

Debian ansible\_host=176.58.114.60.nip.io

[ansible\_hosts]

RedHat

Debian

[ansible\_hosts:vars]

ansible\_connection=ssh

ansible\_user=root

ansible\_private\_key\_file=~/.ssh/id\_rsa

host\_key\_checking=False

CopyExplain

Important note

The preceding inventory file is only for illustrative purposes; if you are following along, you will need to update it to consider your host IP addresses, user names, and private key file locations.

With the hosts up and running, we can target each one individually using the following commands:

$ ansible -i hosts RedHat -m ansible.builtin.setup | grep ansible\_os\_family

$ ansible -i hosts Debian -m ansible.builtin.setup | grep ansible\_os\_family

CopyExplain

Running these two commands should show you something like the following terminal output:

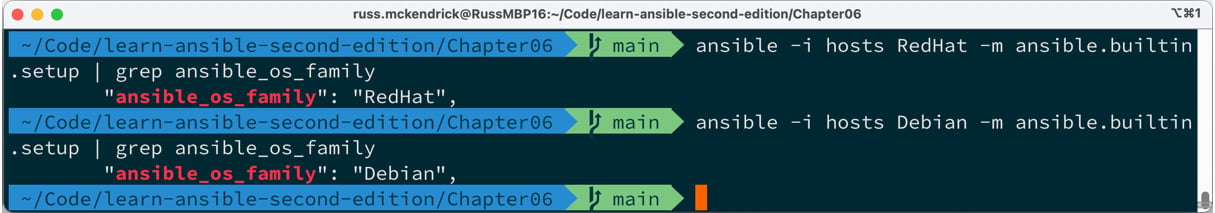


Figure 6.1 – Checking the values of ansible\_os\_family

As you can see, each of the two hosts correctly returns the operating system family.

We can take this one step further and update our commands to the following:

$ ansible -i hosts RedHat -m ansible.builtin.setup | grep ansible\_distribution

$ ansible -i hosts Debian -m ansible.builtin.setup | grep ansible\_distribution

CopyExplain

This gives the following output:

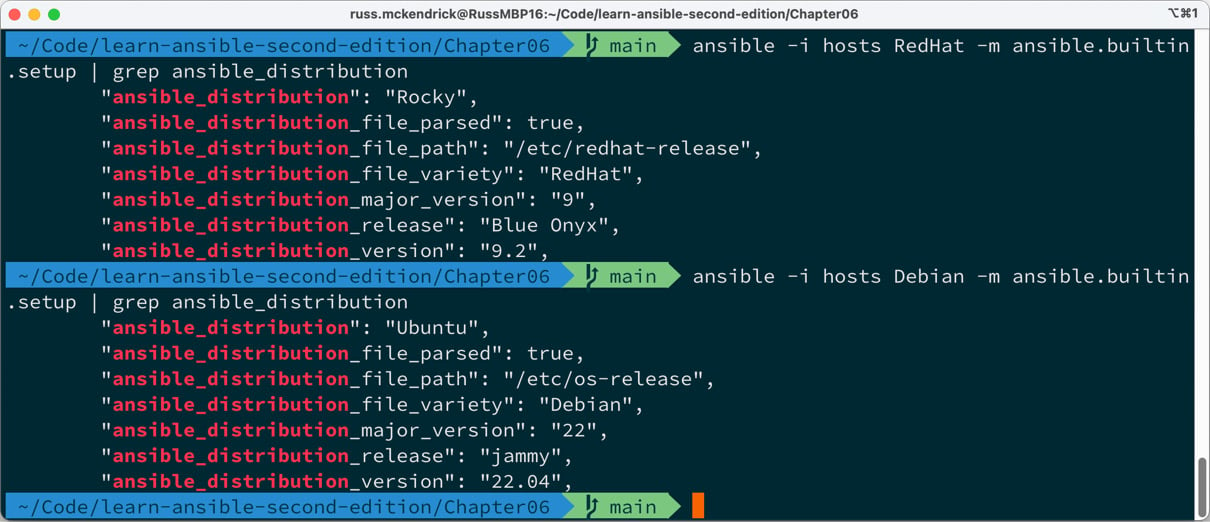


Figure 6.2 – Checking the values of ansible\_distribution

As you can see, this gives much more detail on the operating system itself and not just the flavor of Linux; it is based on RedHat or Debian.

Finally, we run the following command:

$ ansible -i hosts ansible\_hosts -m ansible.builtin.setup | grep ansible\_os\_family

CopyExplain

This will target both hosts within the same Ansible run and return a terminal output that should look like the following:

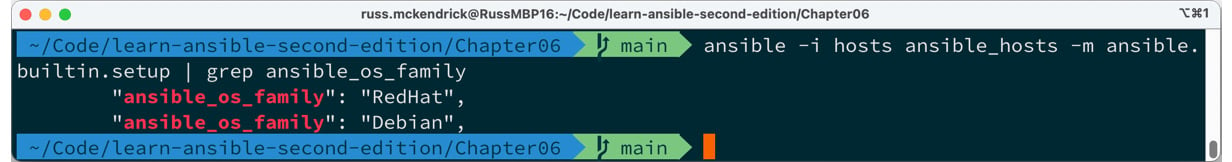


Figure 6.3 – Checking the values of ansible\_distribution in a single run

Now that we can identify which operating system is in use on each host, we can start adapting the roles to consider the changes we discussed in the previous section of this chapter.

The Stack Install role

The first part of the role we will look at is the content of roles/stack\_install/tasks/main.yml. The previous version of the role contained all of the tasks to install the repos and packages for our Ubuntu server; all of those tasks should be moved to a file called roles/stack\_install/tasks/Debian.yml, and a new file called roles/stack\_install/tasks/RedHat.yml should have been created; finally, we should update roles/stack\_install/tasks/main.yml so that it has the following contents.

Here are the three task loads in the variables file for the operating system we are targeting:

- name: "Include the operating system specific variables"

  ansible.builtin.include\_vars: "{{ ansible\_os\_family }}.yml"

CopyExplain

As you can see, this uses the ansible.builtin.include\_vars module to load variables from the variables path within the roles folder, which would be roles/stack\_install/vars/.

Then, it loads a file called RedHat.yml or Debian.yml; these two file names are populated using the {{ ansible\_os\_family }} variable in the task, meaning that the variables relevant to the operating system being targeted are loaded.

If you look in the repository on GitHub, you will notice that, although being subtle, there are differences in the packages listed in the system\_packages, extra\_packages, and stack\_packages package lists.

The next task uses the when condition when calling the ansible.builtin.import\_tasks module, first of all for the Debian-based system:

- name: "Install the stack on Debian based systems"

  ansible.builtin.import\_tasks: "Debian.yml"

  when: ansible\_os\_family == 'Debian'

CopyExplain

In our case, this means that when the Ansible playbook is targeting a Debian-based host, it will load the tasks from roles/stack\_install/tasks/Debian.yml, which are essentially the same as those we discussed at length in [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*, and execute them against the host.

The next task does the same function, but this time for Red Hat-based hosts, using the tasks listed in the roles/stack\_install/tasks/RedHat.yml file:

- name: "Install the stack on RedHat based systems"

  ansible.builtin.import\_tasks: "RedHat.yml"

  when: ansible\_os\_family == 'RedHat'

CopyExplain

The roles/stack\_install/tasks/RedHat.yml file contains three tasks, which are pretty much the same as the Debian.yml tasks.

We start the role by running an update of all the installed packages:

- name: "Update all of the installed packages"

  ansible.builtin.dnf:

    name: "\*"

    state: "latest"

    update\_cache: true

CopyExplain

As you can see, this uses the ansible.builtin.dnf modules rather than the ansible.builtin.apt one.

Next up, we have the task that installs the NGINX mainline repo:

- name: "Add the NGINX mainline repo"

  ansible.builtin.yum\_repository:

    name: "{{ nginx\_repo.name }}"

    description: "{{ nginx\_repo.description }}"

    baseurl: "{{ nginx\_repo.baseurl }}"

    gpgcheck: "{{ nginx\_repo.gpgcheck }}"

    enabled: "{{ nginx\_repo.enabled }}"

CopyExplain

Although this uses the ansible.builtin.yum\_repository module, DNF will pick up the new repo once it is added. This is also the only task we need to run to add the repo, and adding a Yum repository is very different from adding a repository on a Debian-based system.

The final task for Red Hat-based systems is to install all the packages, including the NGINX one from the mainline repository we just enabled by, again, calling the ansible.builtin.dnf module:

- name: "Update cache and install the stack packages"

  ansible.builtin.dnf:

    state: "present"

    update\_cache: true

    pkg: "{{ system\_packages + extra\_packages + stack\_packages }}"

CopyExplain

As you can see, with a little change to the logic in which the tasks are being called, it was relativity painless to update the role to target Debian and Red Hat distributions.

For the next role we need to change, the Stack Config role, we will take a slightly different approach to considering the different operating system distributions.

The Stack Config role

Apart from a single task at the start and half a dozen at the end, the bulk of this role remains as-is.

There are some changes to the default variables file in the roles/stack\_config/default/main.yml file; first off, the following variables are added:

selinux:

  http\_permissive: true

firewall\_comands:

  - "firewall-cmd --zone=public --add-port=80/tcp --permanent"

  - "firewall-cmd --zone=public --add-port=80/tcp"

CopyExplain

As I am sure you can guess from their names, these deal with SELinux and the Firewall.

The next change is to move the mysql\_socket\_path, php\_fpm\_path, php\_ini\_path, and php\_service\_name variables to distribution-specific files at roles/stack\_config/vars/Debian.yml and roles/stack\_config/vars/RedHat.yml.

As we have already discussed, one of the key differences between the two distributions is the paths to both the core files and the configuration files for the services we installed during the Stack Install role.

In the roles/stack\_config/vars/Debian.yml file, we have the following:

mysql\_socket\_path: "/var/run/mysqld/mysqld.sock"

php\_fpm\_path: "/etc/php/8.1/fpm/pool.d/www.conf"

php\_ini\_path: "/etc/php/8.1/fpm/php.ini"

php\_service\_name: "php8.1-fpm"

CopyExplain

However, for the roles/stack\_config/vars/RedHat.yml file, we need to define the following:

mysql\_socket\_path: "/var/lib/mysql/mysql.sock"

php\_fpm\_path: "/etc/php-fpm.d/www.conf"

php\_ini\_path: /etc/php.ini

php\_service\_name: "php-fpm"

CopyExplain

As you can see, at first glance, they look a little similar, but the paths and file names are different.

These files are called by a task, which is the same as we used at the start of the Stack Install role:

- name: Include the operating system specific variables

  ansible.builtin.include\_vars: "{{ ansible\_os\_family }}.yml"

CopyExplain

From here, all the original tasks we covered in [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*, are called and executed, ending with the task that removes the test MySQL database.

From here, in the role, we have the tasks that consider the additional steps needed to configure our Red Hat-based host, starting with configuring SELinux; for our role, we need to enable the policy that allows web servers to run, for which, on a lot of Red Hat distributions, is blocked by default.

The task to do this looks like the following:

- name: "Set the selinux allowing httpd\_t to be permissive is required"

  community.general.selinux\_permissive:

    name: httpd\_t

    permissive: true

  when: selinux.http\_permissive and ansible\_os\_family == 'RedHat'

CopyExplain

As you can see, the when condition here ensures that the task is only executed when the selinux.http\_permissive variable is set to true, and the ansible\_os\_family is equal to RedHat.

While our Debian-based system will meet the selinux.http\_permissive condition, the task will be skipped on those hosts, as it doesn’t meet the second condition.

Finally, we have the tasks for configuring the firewalld service, which is the default firewall on most modern Red Hat-based distributions.

Important note

Although we are using firewall-cmd in this section, there is an Ansible module that supports the firewalld service called ansible.posix.firewalld. As this is the only instance in the title that we will be targeting a Red Hat-based operating system with, we have, instead, used ansible.builtin.command to show how we can meet more complex conditions based on the output commands.

Like some of the roles we have included in other chapters, configuring the firewall is a task we only have to do once. The first thing we will do is check for the presence of a file at ~/firewall-configured and register the results:

- name: "Check to see if the ~/firewall-configured file exists"

  ansible.builtin.stat:

    path: "~/firewall-configured"

  register: firewall\_configured

CopyExplain

Next, we need to check whether firewalld is running, but only if it’s a RedHat distribution. To do this, we need to run the firewall-cmd --state shell command and the output result is registered in the fireweall\_status variable:

- name: "Check if firewalld is running"

  ansible.builtin.command: firewall-cmd --state

  register: fireweall\_status

  when: ansible\_os\_family == 'RedHat'

CopyExplain

Now, as the remaining tasks could also be executed on a Debian-based host, we need to take that into account as we now have a variable containing the stdout of the command we ran called fireweall\_status, which won’t be present, resulting in an error that would stop playbook execution on a Debian-based host:

- name: "Set a fact so the playbook can continue if running on a Debian based system"

  ansible.builtin.set\_fact:

    fireweall\_status:

      stdout: notrunning

  when: ansible\_os\_family == 'Debian'

CopyExplain

As you can see from the preceding task, if ansible\_os\_family is Debian, we are setting the fireweall\_status.stdout variable to notrunning.

Now we have all of the information we need to make a decision on whether we should run the commands to configure the firewall, the following conditions need to be met:

* The firewall-cmd --state command returns running
* The operating system is RedHat
* The ~/firewall-configured file does not exist

If all three of these conditions are met, which are defined in the following task, then the commands to configure the firewall to open and allow traffic on port 80 are executed:

- name: "Run the firewall-cmd commands if the firewall-cmd --state command returns running"

  ansible.builtin.command: "{{ item }}"

  with\_items: "{{ firewall\_comands }}"

  when: fireweall\_status.stdout == "running" and ansible\_os\_family == 'RedHat' and not firewall\_configured.stat.exists

CopyExplain

The final task then creates the ~/firewall-configured file so that the commands are not executed again:

- name: "Create the ~/firewall-configured file"

  ansible.builtin.file:

    path: ~/firewall-configured

    state: touch

    mode: "0644"

  when: not firewall\_configured.stat.exists

CopyExplain

It does this on both distributions, as it doesn’t matter if it is set on Debian-based systems, and we don’t want to run the commands regardless; on Red Hat systems, it will mean that any subsequent executions of the playbook will not be able to meet the three conditions where the commands are executed to configure the firewall service.

The WordPress role

As already mentioned, we do not have to make any changes to this role.

Running the playbook

There are no changes to our site.yml file, meaning that we need to run the following command to run the playbook:

$ ansible-playbook -i hosts site.yml

CopyExplain

There is way too much output to cover here, but I will include some of the highlights from the playbook execution, starting with the gathering of the facts:

TASK [Gathering Facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [Debian]

ok: [RedHat]

CopyExplain

Now that Ansible knows about our two hosts, it makes a start on running the tasks; here are the updated ones from the Stack Install role:

TASK [roles/stack\_install : update apt-cache and upgrade packages] \*\*\*\*\*\*\*\*\*\*\*\*\*

skipping: [RedHat]

changed: [Debian]

CopyExplain

As you can see, this was the apt one, and the dnf one looks like this:

TASK [roles/stack\_install : update all of the installed packages] \*\*\*\*\*\*\*\*\*\*\*\*\*\*

skipping: [Debian]

changed: [RedHat]

CopyExplain

Now, moving onto the Stack Config role, this is where tasks are being run on both distributions:

TASK [roles/stack\_config : add the wordpress group] \*\*\*\*\*\*\*

changed: [RedHat]

changed: [Debian]

CopyExplain

To update the firewall on just the Red Hat-based distribution, we do the following:

TASK [roles/stack\_config : run the firewall-cmd commands if the firewall-cmd --state command returns running] \*\*\*

skipping: [Debian] => (item=firewall-cmd --zone=public --add-port=80/tcp --permanent)

skipping: [Debian] => (item=firewall-cmd --zone=public --add-port=80/tcp)

skipping: [Debian]

changed: [RedHat] => (item=firewall-cmd --zone=public --add-port=80/tcp --permanent)

changed: [RedHat] => (item=firewall-cmd --zone=public –

-add-port=80/tcp)

CopyExplain

Finally, we complete the playbook run:

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Debian                     : ok=44   changed=29   unreachable=0    failed=0    skipped=7    rescued=0    ignored=2

RedHat                     : ok=45   changed=34   unreachable=0    failed=0    skipped=6    rescued=0    ignored=2

CopyExplain

All of which means that I should now have two WordPress installations:

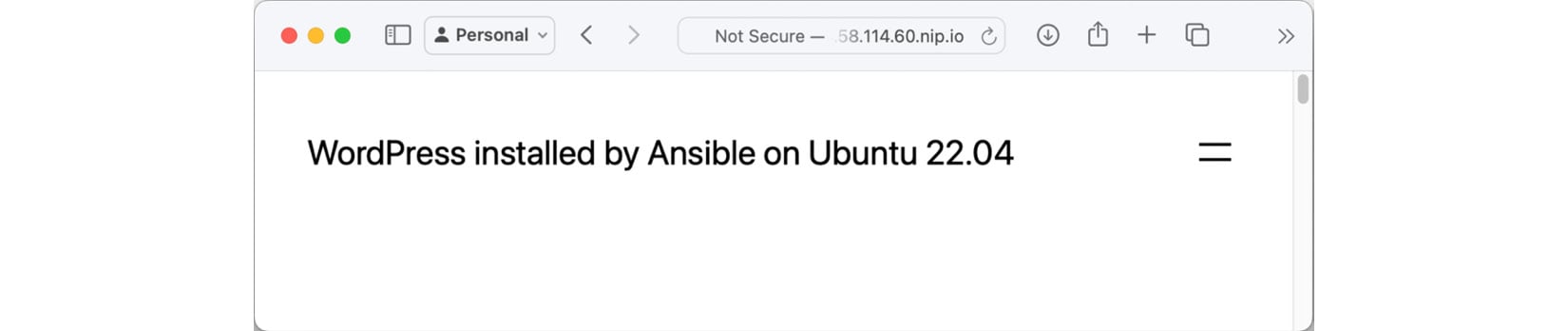


Figure 6.4 – WordPress running on Ubuntu 22.04

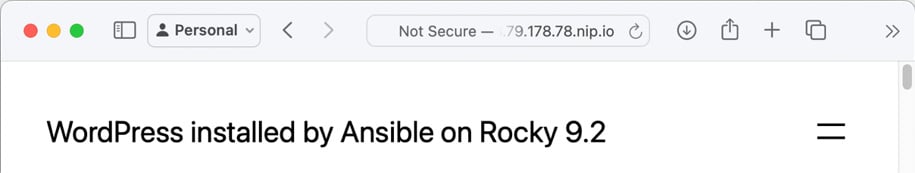


Figure 6.5 – WordPress running on Rocky Linux 9.2

While the preceding screens aren’t the most exciting of websites, as you can see, we have WordPress up and running on two different operating systems.

At this point, if you have been following along, don’t forget to delete any resources you have deployed to run your playbooks against.

Summary

In this chapter, we have adapted the WordPress installation playbook we wrote in [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*, to target multiple operating systems. We did this by using Ansible’s built-in auditing module to determine which operating system the playbook is running against and running only the tasks that will work on the two target distributions.

While targeting multiple Linux distributions is one use for the approach we have taken with the conditions we have been using, I am sure that you will already have some ideas on how you could use some of the logic we used in your projects, such as bootstrapping different software based on the role on a virtual machine host, etc.

Additionally, this approach is beneficial when publishing your roles to Ansible Galaxy, as discussed in [*Chapter 2*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/2), *Exploring Ansible Galaxy*, by making the operating system agnostic.

You may have noticed so far that we have been targeting Linux virtual machines; in the next chapter, we will look at Ansible support for Windows-based operating systems.