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 author's 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: Deborah Baker

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

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 author's 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.
>
> 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 dmitri@aboutsqlserver.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.

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:

playbook would look something like this:

```
- name:
      install nginx
      package:
      name: nginx
```

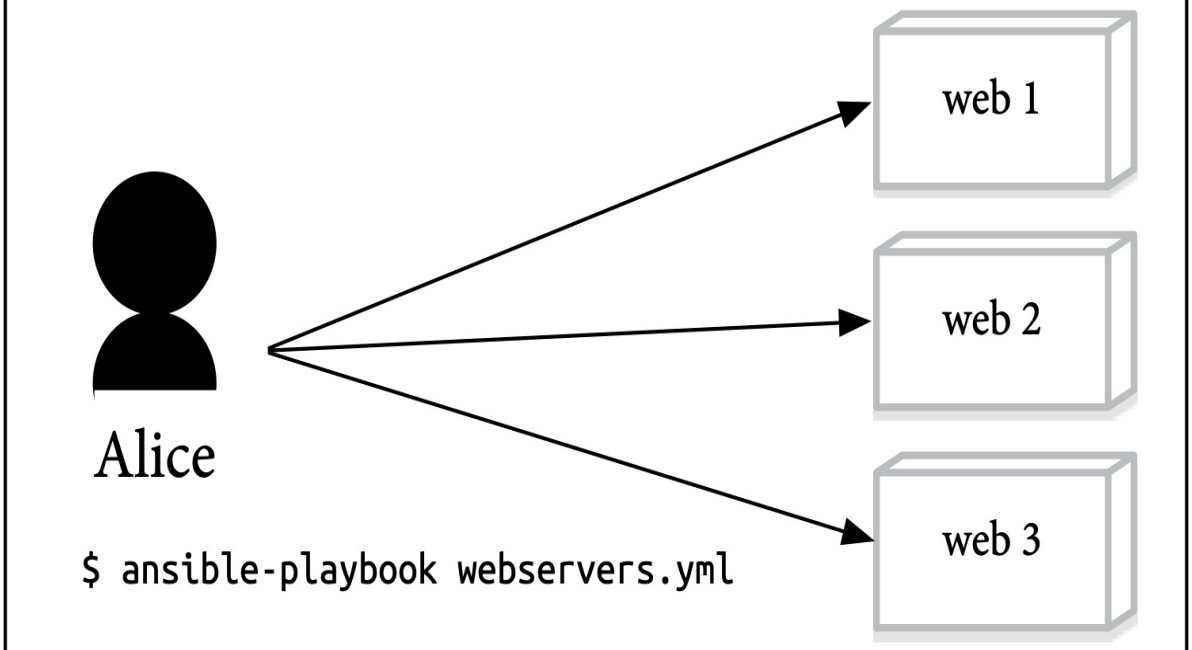
Ansible will do the following:

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:

```
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 webservers 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:")
          prinr(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

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 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 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
        $ python3 -m pip install --upgrade pip
        $ pip3 install wheel
        $ git clone https://github.com/ansible/ansible.git --recursive
        $ 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:

playbooks programs are:

```
$ cd ./ansible
        $ 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
```

.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 vagrant 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 toother 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

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 author's 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.

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 dmitri@aboutsqlserver.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

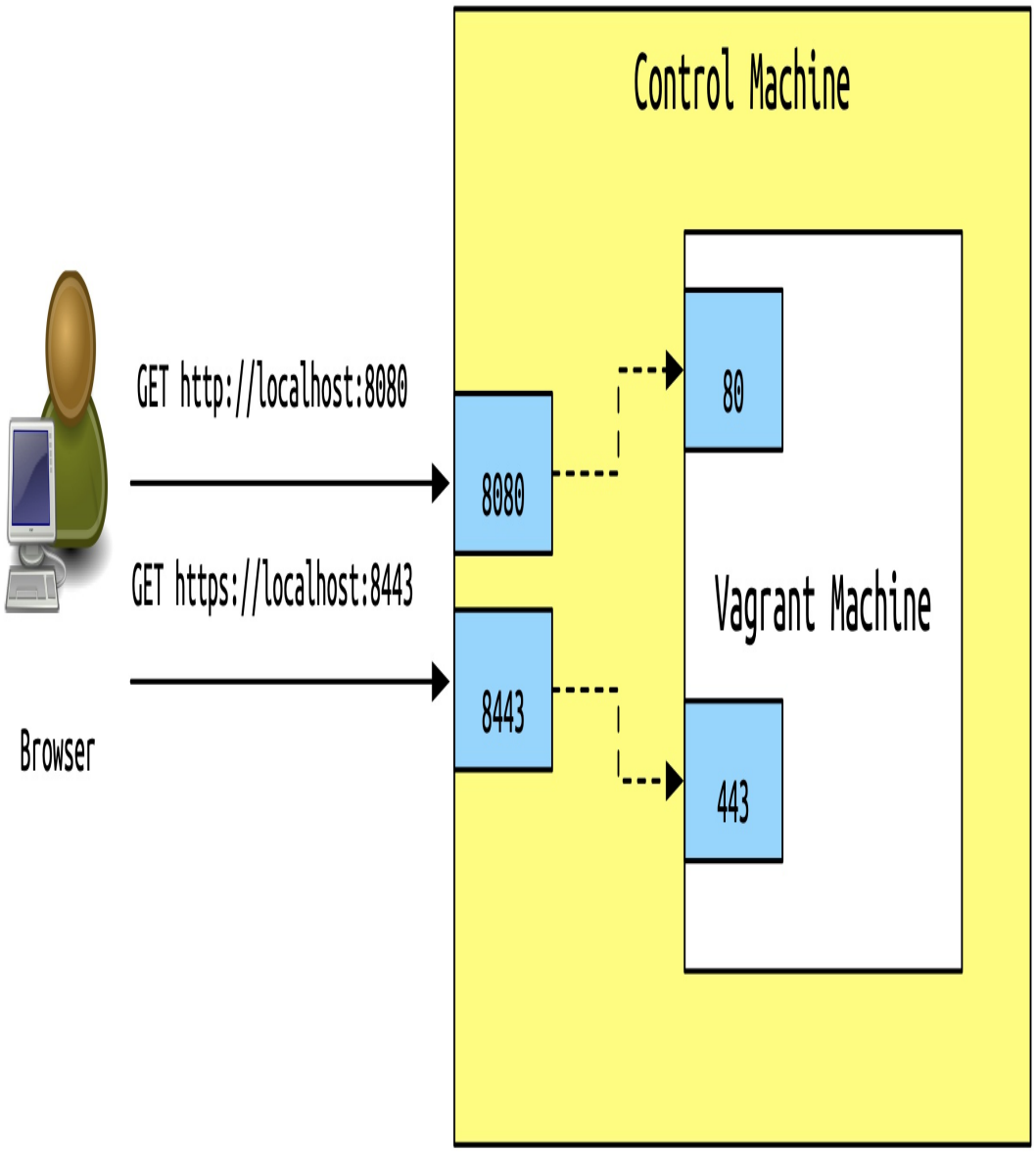


Figure 2-1. Exposing ports on a Vagrant machine

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.


Welcome to ansible
localhost:8080
Nginx, configured by Ansible
If you can see this, Ansible successfully installed nginx.
Running on testserver

Figure 2-2. Rendered HTML

Creating a Group

Let's create a webservers 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 [webservers] line above the testserver line, as shown in playbooks/inventory/vagrant.ini. This means that testserver is in the webservers group. The group can have variables defined (vars is s a shorthand for variables). Your file should look like example 2-4.

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

```
[webservers]
          testserver ansible_port=2202
          [webservers: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:
           |--@webservers:
           | |--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]

COWSAY

No O'Reilly book about Ansible would be complete without describing cowsay support.

If you have the cowsay program installed on your local machine, Ansible output will include a cow in ascii-art like this:

```
 < PLAY [Configure webserver with nginx] >
 ----------------------------------------
  \ ^__^
  \ (oo)\_______
    (__)\ )\/\
     ||----w |
     || ||
```

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,tux,
```

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: ---

H[illegible]

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.

Now, you can put Boolean values in quotation marks! (This is called "quoting".

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
           appt: '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",
           "appt": 742,
           "city": "Springfield",
           "state": "North Takoma"
           }
```

```
 }
}
```

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

```
address: { street: Evergreen Terrace, appt: '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:
```

```
        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

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 this out when the play starts to run.

`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 if needed.

`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 webservers -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 webservers.

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

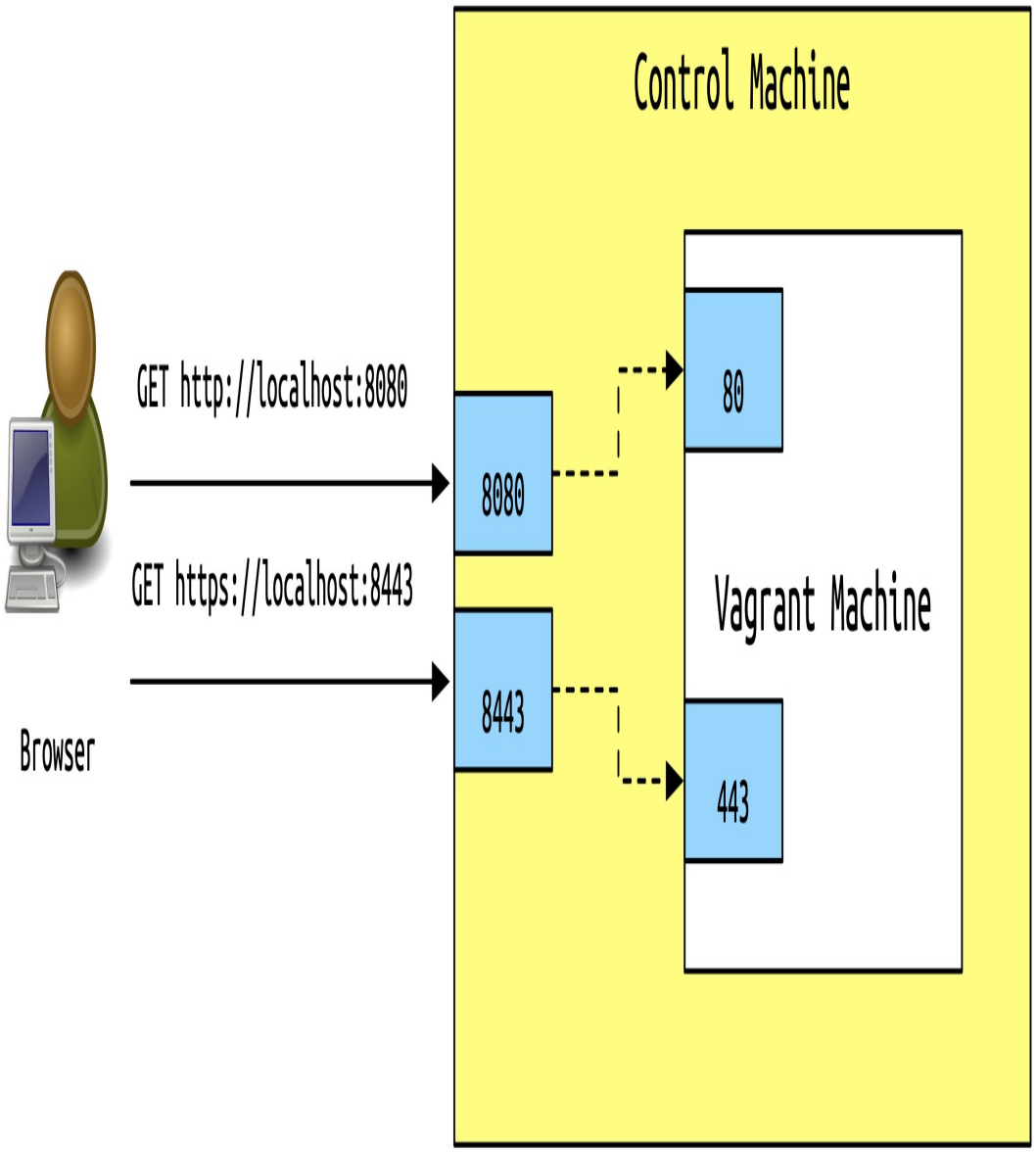


Figure 2-3. Entity-relationship diagram of a playbook

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
```

```
- 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;
```

```
    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/webservers-tls.yml

```
#!/usr/bin/env ansible-playbook
          ---
          - name: Configure webserver with Nginx and TLS
           hosts: webservers
           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

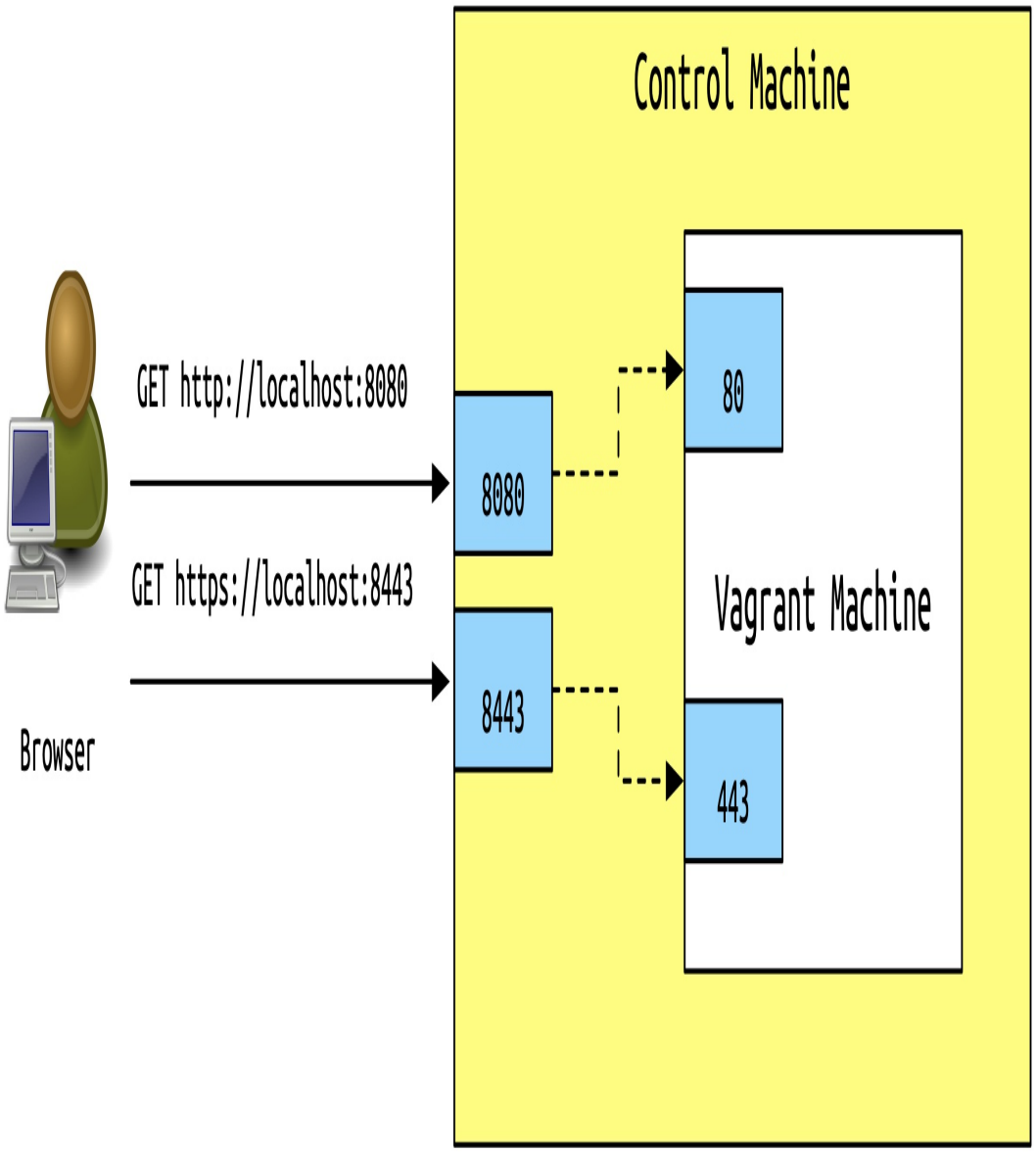


Figure 2-4. Browsers such as Chrome don't trust self-signed TLS certificates.

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.

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.