O'REILLY®

2nd Edition

Ansible Up & Running

AUTOMATING CONFIGURATION MANAGEMENT AND DEPLOYMENT THE EASY WAY

Lorin Hochstein & Rene Moser

Ansible: Up and Running

This Preview Edition of *Ansible: Up and Running*, 2nd Edition, Chapters 1–3, is a work in progress. The final book is currently scheduled for release in July 2017 and will be available at oreilly.com and other retailers once it is published.

Lorin Hochstein and Rene Moser

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Ansible: Up and Running

by Lorin Hochstein and Rene Moser

Printed in the United States of America.

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

Editor: Brian Anderson and Angela Rufino
Production Editor: Kristen Brown
Copyeditor:
Proofreader:

Indexer:
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Rebecca Demarest

May 2015:	First Edition
July 2017:	Second Edition

Revision History for the Second Edition

2017-05-24: First Release

978-1-491-97980-8

[LSI]

Table of Contents

1. Introduction. 1
- A Note About Versions 2
- Ansible: What Is It Good For? 2
- How Ansible Works 3
- What's So Great About Ansible? 5
  - Easy-to-Read Syntax 5
  - Nothing to Install on the Remote Hosts 5
  - Push-Based 5
  - Ansible Scales Down 6
  - Built-in Modules 6
  - Very Thin Layer of Abstraction 7
- Is Ansible Too Simple? 8
- What Do I Need to Know? 9
- What Isn't Covered 10
- Installing Ansible 10
- Setting Up a Server for Testing 11
  - Using Vagrant to Set Up a Test Server 11
  - Telling Ansible About Your Test Server 14
  - Simplifying with the ansible.cfg File 16
- Moving Forward 20

2. Playbooks: A Beginning. 21
- Some Preliminaries 21
- A Very Simple Playbook 22
  - Specifying an nginx Config File 24
  - Creating a Custom Homepage 25
  - Creating a Webservers Group 25
- Running the Playbook 26

Playbooks Are YAML 27
- Start of File 28
- Comments 28
- Strings 28
- Booleans 28
- Lists 29
- Dictionaries 29
- Line Folding 30

Anatomy of a Playbook 30
- Plays 32
- Tasks 33
- Modules 34
- Putting It All Together 35

Did Anything Change? Tracking Host State 35

Getting Fancier: TLS Support 36
- Generating TLS certificate 38
- Variables 38
- Generating the Nginx Configuration Template 40
- Handlers 41
- Running the Playbook 43

3. Inventory: Describing Your Servers. 45

The Inventory File 45

Preliminaries: Multiple Vagrant Machines 46

Behavioral Inventory Parameters 49
- ansible_connection 49
- ansible_shell_type 49
- ansible_python_interpreter 50
- ansible_*_interpreter 50
- Changing Behavioral Parameter Defaults 50

Groups and Groups and Groups 51
- Example: Deploying a Django App 52
- Aliases and Ports 54
- Groups of Groups 54
- Numbered Hosts (Pets versus Cattle) 55

Hosts and Group Variables: Inside the Inventory 56

Host and Group Variables: In Their Own Files 57

Dynamic Inventory 59
- The Interface for a Dynamic Inventory Script 60
- Writing a Dynamic Inventory Script 61
- Pre-Existing Inventory Scripts 65

Breaking Out the Inventory into Multiple Files 65

Adding Entries at Runtime with add_host and group_by 65
  add_host 65
  group_by 67

CHAPTER 1

Introduction

It's an interesting time to be working in the IT industry. We don't deliver software to our customers by installing a program on a single machine and calling it a day.[1] Instead, we are all slowly turning into system 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.

We also need to make sure we have the appropriate redundancies in place, so that when failures happen (and they will), our software systems will handle these failures 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 endpoints for managing virtual machine instances.[2]

You can wire up these services by hand: spinning up the servers you need, SSHing to 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 inside your application, doing it by hand is madness. There's a better way.

1 OK, nobody ever really delivered software like that.

2 Check out *The Practice of Cloud System Administration* and *Designing Data-Intensive Applications* for excellent books on building and maintaining these types of distributed systems.

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 version 2.3.0.0 of Ansible, which was the most recent release as of this writing. As backward compatibility is a major goal of the Ansible project, these examples should work unmodified in future versions of Ansible.

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. The author Ursula K. Le Guin invented the concept in her book *Rocannon's World*, and other sci-fi authors have since borrowed the idea from Le Guin.

More specifically, Michael DeHaan took the name Ansible from the book *Ender's Game* by Orson Scott Card. In that book, the ansible was used to control a large number of 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 *Chef*, *Puppet*, and *Salt*. When we talk about configuration management, we are typically talking about writing some kind of state description for our servers, and then using a tool to enforce that the servers are, indeed, in that state: the right packages are installed, configuration files contain 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.

These tools also can be used for doing *deployment* as well. When people talk about deployment, they are usually referring to the process of taking software that was written in-house, generating binaries or static assets (if necessary), copying the required files to the server(s), and then starting up the services. *Capistrano* and *Fabric* are two examples of open-source deployment tools. Ansible is a great tool for doing deployment as well as configuration management. Using a single tool for both configuration management and deployment makes life simpler for the folks responsible for operations.

Some people talk about the need for *orchestration* of deployment. This is where multiple remote servers are involved, and things have to happen in a specific order. For example, you need to bring up the database before bringing up the web servers, or you need to take web servers out of the load balancer one at a time in order to upgrade them without downtime. Ansible's good at this as well, and is designed from the ground up for performing actions on multiple servers. Ansible has a refreshingly simple model for controlling the order that actions happen in.

Finally, you'll hear people talk about *provisioning* new servers. In the context of public clouds such as Amazon EC2, this refers to spinning up a new virtual machine instance. Ansible's got you covered here, with a number of modules for talking to clouds, including EC2, Azure, Digital Ocean, Google Compute Engine, Linode, and Rackspace, as well as any clouds that support the OpenStack APIs.

Confusingly, the *Vagrant* tool, which we'll discuss later in this chapter, uses the term "provisioner" to refer to a tool that does the configuration management. So, Vagrant refers to Ansible as a kind of provisioner, where I think of Vagrant as a provisioner, since Vagrant is responsible for starting up virtual machines.

How Ansible Works

Figure 1-1 shows a sample use case of Ansible in action. A user we'll call Stacy 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:

- Install nginx
- Generate an nginx configuration file
- Copy over the security certificate
- Start the nginx service

In the next chapter, we'll discuss what's actually in this playbook. Stacy executes the playbook using the *ansible-playbook* command. In the example, the playbook is named *webservers.yml*, and is executed by typing:

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

Ansible will make SSH connections in parallel to web1, web2, and web3. It will execute the first task on the list on all three hosts simultaneously. In this example, the first task is installing the nginx apt package (since Ubuntu uses the apt package manager), so the task in the playbook would look something like this:

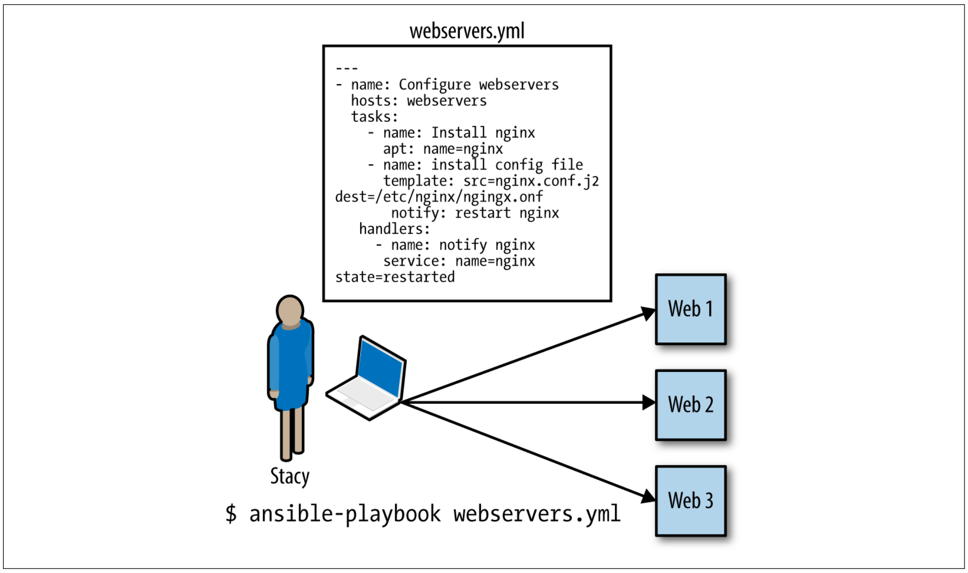
```
- name: Install nginx
  apt: name=nginx
```

Ansible will:

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 that:

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



```
---
- name: Configure webservers
  hosts: webservers
  tasks:
    - name: Install nginx
      apt: name=nginx
    - name: install config file
      template: src=nginx.conf.j2
dest=/etc/nginx/ngingx.onf
       notify: restart nginx
   handlers:
     - name: notify nginx
      service: name=nginx
state=restarted
```

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

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. Here are some of the things that drew me to Ansible in particular.

Easy-to-Read Syntax

Recall that Ansible configuration management scripts are called *playbooks*. Ansible's playbook syntax is built 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.

I like to think of Ansible playbooks as *executable documentation*. It's like the README file that describes the commands you had to type out to deploy your software, except that the instructions will never go out-of-date because they are also the code that gets executed directly.

Nothing to Install on the Remote Hosts

To manage a server with Ansible, the server needs to have SSH and Python 2.5 or later installed, or Python 2.4 with the Python *simplejson* library installed. There's no need to preinstall an agent or any other software on the host.

The control machine (the one that you use to control remote machines) needs to have Python 2.6 or later installed.

Some modules might require Python 2.5 or later, and some might have additional prerequisites. Check the documentation for each module to see whether it has specific requirements.

Push-Based

Some configuration management systems that use agents, such as Chef and Puppet, are "pull-based" by default. Agents installed on the servers periodically check in with a central service and pull down 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 which 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 server and does its thing.

The push-based approach has a significant advantage: you control when the changes happen to the servers. You don't need to wait around for a timer to expire. Advocates of the pull-based approach claim that pull is superior for scaling to large numbers of servers and for dealing with new servers that can come online anytime. However, as we'll discuss later in the book, Ansible has been used successfully in production with thousands of nodes, and has excellent support for environments where servers are dynamically added and removed.

If you really prefer using a pull-based model, Ansible has official support for pull mode, using a tool it ships with called *ansible-pull*. I don't cover pull mode in this book, but you can read more about it in the official documentation.

Ansible Scales Down

Yes, Ansible can be used to manage hundreds or even thousands of nodes. But what got me hooked is how it scales down. Using Ansible to configure a single node is easy; you simply write a single playbook. Ansible obeys Alan Kay's maxim: "Simple things should be simple, complex things should be possible."

Built-in Modules

You can use Ansible to execute arbitrary shell commands on your remote servers, but Ansible's real power comes from the collection of modules it ships with. You use modules to perform tasks such as installing a package, restarting a service, or copying a configuration file.

As we'll 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 was an account named "deploy" in the "web" group:

```
user: name=deploy group=web
```

Modules are also *idempotent*. If the "deploy" user doesn't exist, then Ansible will create it. If it does exist, then Ansible won't do anything. Idempotence is a nice property because it means that it's safe to run an Ansible playbook multiple times against a server. This is a big improvement over the homegrown shell script approach, where running the shell script a second time might have a different (and likely unintended) effect.

What About Convergence?

Books on configuration management often mention the concept of *convergence*. Convergence in configuration management is most closely associated with Mark Burgess and the *CFEngine* configuration management system he authored.

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

This idea of convergence doesn't really apply to Ansible, as Ansible doesn't have a notion of running multiple times to configure servers. Instead, Ansible modules are implemented in such a way that running an Ansible playbook a single time should put each server into the desired state.

If you're interested in what Ansible's author thinks of the idea of convergence, see Michael DeHaan's post in the Ansible Project newsgroup, entitled, "Idempotence, convergence, and other silly fancy words we use too often."

Very Thin Layer of Abstraction

Some configuration management tools provide a layer of abstraction so that you can use the same configuration management scripts to manage servers running different operating systems. For example, instead of having to deal with a specific package manager like yum or apt, the configuration management tool exposes a "package" abstraction that you use instead.

Ansible isn't like that. You have to use the apt module to install packages on apt-based systems and the yum module to install packages on yum-based systems.

Although this might sound like a disadvantage, in practice, I've found that it makes Ansible easier to work with. Ansible doesn't require that I learn a new set of abstractions that hide the differences between operating systems. This makes Ansible's surface area smaller; there's less you need to know before you can start writing playbooks.

If you really want to, you can write your Ansible playbooks to take different actions, depending on the operating system of the remote server. But I try to avoid that when I can, and instead I focus on writing playbooks that are designed to run on a specific operating system, such as Ubuntu.

The primary unit of reuse in the Ansible community is the module. Because the scope of a module is small and can be operating-system specific, it's straightforward to implement well-defined, shareable modules. The Ansible project is very open to accepting modules contributed by the community. I know because I've contributed a few.

Ansible playbooks aren't really intended to be reused across different contexts. In Chapter 8, we'll discuss *roles*, which is a way of collecting playbooks together so they are more reusable, as well as Ansible Galaxy, which is an online repository of these roles.

In practice, though, every organization sets up its servers a little bit differently, and you're best off writing playbooks for your organization rather than trying to reuse generic playbooks. I believe the primary value of looking at other people's playbooks is for examples to see how things are done.

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'll 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 proprietary web-based management tool called *Ansible Tower*, which is covered in ???. In October 2015, Red Hat acquired Ansible, Inc.

Is Ansible Too Simple?

When I was working on this book, my editor mentioned to me that "some folks who use the XYZ configuration management tool call Ansible a for-loop over SSH scripts." If you're considering switching over from another config management tool, you might be concerned at this point about whether Ansible is powerful enough to meet your needs.

As you'll soon learn, Ansible provides a lot more functionality than shell scripts. As I mentioned, Ansible's modules provide idempotence, and 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 maintain a non-trivial program written in shell. I'll always choose Ansible over shell scripts for config management tasks if given a choice.

And if you're worried about the scalability of SSH? As we'll discuss in Chapter 9, Ansible uses SSH multiplexing to optimize performance, and there are folks out there who are managing thousands of nodes with Ansible.[3]

I'm not familiar enough with the other tools to describe their differences in detail. If you're looking for a head-to-head comparison of config management tools, check out *Taste Test: Puppet, Chef, Salt, Ansible* by Matt Jaynes. As it happens, Matt prefers Ansible.

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's not the kind of tool that "automagically" does things that you otherwise wouldn't know how to do.

For this book, I assumed my readers would be familiar with at least one Linux distribution (e.g., Ubuntu, RHEL/CentOS, SUSE), and that they would 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, then you're good to go with Ansible.

I won't assume you have knowledge of any particular programming language. For instance, you don't need to know Python to use Ansible unless you want to write your own module.

Ansible uses the YAML file format and uses the Jinja2 templating languages, so you'll need to learn some YAML and Jinja2 to use Ansible, but both technologies are easy to pick up.

3 For example, see "Using Ansible at Scale to Manage a Public Cloud" by Jesse Keating, formerly of Rackspace.

What Isn't Covered

This book isn't an exhaustive treatment of Ansible. It's designed to get you productive in Ansible as quickly as possible and describe how to perform certain tasks that aren't obvious from glancing over the official documentation.

I don't cover the official Ansible modules in detail. There are over 200 of these, and the official Ansible reference documentation on the modules is quite good.

I only cover the basic features of the templating engine that Ansible uses, Jinja2, primarily because I find that I generally only need to use the basic features of Jinja2 when I use Ansible. If you need to use more advanced Jinja2 features in your templates, I recommend you check out the official Jinja2 documentation.

I don't go into detail about some features of Ansible that are mainly useful when you are running Ansible on an older version of Linux. This includes features such as the *paramiko* SSH client and *accelerated mode*.

Finally, there are several features of Ansible I don't cover simply to keep the book a manageable length. These features include pull mode, logging, connecting to hosts using protocols other than SSH, and prompting the user for passwords or input. I encourage you to check out the official docs to find out more about these features.

Installing Ansible

If you're running on a Linux machine, all of the major Linux distributions package Ansible these days, so you should be able to install it using your native package manager, although this might be an older version of Ansible. If you're running on Mac OS X, I recommend you use the excellent Homebrew package manager to install Ansible.

If all else fails, you can install it using *pip*, Python's package manager. You can install it as root by running:

```
$ sudo pip install ansible
```

If you don't want to install Ansible as root, you can safely install it into a Python *virtualenv*. If you're not familiar with virtualenvs, you can use a newer tool called *pipsi* that will automatically install Ansible into a virtualenv for you:

```
$ wget https://raw.githubusercontent.com/mitsuhiko/pipsi/master/get-pipsi.py
$ python get-pipsi.py
$ pipsi install ansible
```

If you go the pipsi route, you'll need to update your PATH environment variable to include *~/.local/bin*. Some Ansible plug-ins and modules might require additional Python libraries. If you've installed with pipsi, and you wanted to install *docker-py* (needed by the Ansible Docker modules) and *boto* (needed by the Ansible EC2 modules), you'd do it like this:

```
$ cd ~/.local/venvs/ansible
$ source bin/activate
$ pip install docker-py boto
```

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

```
$ git clone https://github.com/ansible/ansible.git --recursive
```

If you're running Ansible from the development branch, you'll 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.

```
$ cd ./ansible
$ source ./hacking/env-setup
```

For more details on installation see:

- Official Ansible install docs
- Pip
- Virtualenv
- Pipsi

Setting Up a Server for Testing

You'll 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 a public cloud service such as Amazon EC2, Google Compute Engine, Microsoft Azure,[4] Digital Ocean, Linode…you get the idea.

Using Vagrant to Set Up a Test Server

If you'd 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 Vagrant to boot a Linux virtual machine inside of your laptop, and we can use that as a test server.

Vagrant has built-in support for provisioning virtual machines with Ansible, but we'll talk about that in detail in Chapter 11. For now, we'll just manage a Vagrant virtual machine as if it were a regular Linux server.

4 Yes, Azure supports Linux servers.

Vagrant needs the VirtualBox virtualizer to be installed on your machine. Download VirtualBox and then download Vagrant.

I recommend you create a directory for your Ansible playbooks and related files. In the following example, I've named mine *playbooks*.

Run the following commands to create a Vagrant configuration file (Vagrantfile) for an Ubuntu 14.04 (Trusty Tahr) 64-bit virtual machine image,[5] and boot it.

```
$ mkdir playbooks
$ cd playbooks
$ vagrant init ubuntu/trusty64
$ vagrant up
```

The first time you do `vagrant up`, it will download the virtual machine image file, which might take a while depending on your Internet connection.

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

```
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/trusty64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/trusty64' is up to date...
==> default: Setting the name of the VM: playbooks_default_1474348723697_56934
==> default: Clearing any previously set forwarded ports...
==> 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: 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: Warning: Remote connection disconnect. Retrying...
    default: Warning: Remote connection disconnect. Retrying...
    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!
```

5 Vagrant uses the terms *machine* to refer to a virtual machine and *box* to refer to a virtual machine image.

```
==> default: Checking for guest additions in VM...
    default: The guest additions on this VM do not match the installed version of
    default: VirtualBox! In most cases this is fine, but in rare cases it can
    default: prevent things such as shared folders from working properly. If you see
    default: shared folder errors, please make sure the guest additions within the
    default: virtual machine match the version of VirtualBox you have installed on
    default: your host and reload your VM.
    default:
    default: Guest Additions Version: 4.3.36
    default: VirtualBox Version: 5.0
==> default: Mounting shared folders...
    default: /vagrant => /Users/lorin/dev/ansiblebook/ch01/playbooks
```

You should be able to SSH into your new Ubuntu 14.04 virtual machine by running:

```
$ vagrant ssh
```

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

```
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 3.13.0-96-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

  System information as of Fri Sep 23 05:13:05 UTC 2016

  System load:  0.76              Processes:           80
  Usage of /:   3.5% of 39.34GB   Users logged in:     0
  Memory usage: 25%               IP address for eth0: 10.0.2.15
  Swap usage:   0%

  Graph this data and manage this system at:
    https://landscape.canonical.com/

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.

New release '16.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.
```

Type **`exit`** to quit the SSH session.

This approach lets us interact with the shell, but Ansible needs to connect to the virtual machine using the regular SSH client, not the `vagrant ssh` command.

Tell Vagrant to output the SSH connection details by typing:

```
$ 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:

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

Vagrant 1.7 changed how it handled private SSH keys. Starting with 1.7, Vagrant 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 1.7.

In your case, every field should likely be the same except for the path of the Identity-File.

Confirm that you can start an SSH session from the command line using this information. In my case, the SSH command is:

```
$ ssh vagrant@127.0.0.1 -p 2222 -i /Users/lorin/dev/ansiblebook/ch01/
playbooks/.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 file.

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 some additional arguments to tell Ansible how to connect to it. We'll give our Vagrant server the alias of `testserver`.

Create a file called *hosts* in the *playbooks* directory. This file will serve as the inventory file. If you're using a Vagrant machine as your test server, your *hosts* file should look like Example 1-1. I've broken the file contents up across multiple lines so that it fits on the page, but it should be all on one line in your file, without any backslashes.

Example 1-1. playbooks/hosts

```
testserver ansible_host=127.0.0.1 ansible_port=2222 \
 ansible_user=vagrant \
 ansible_private_key_file=.vagrant/machines/default/virtualbox/private_key
```

Here we see one of the drawbacks of using Vagrant. We have to explicitly pass in extra arguments to tell Ansible how to connect. In most cases, we won't need this extra data.

Later on in this chapter, we'll see how we can use the *ansible.cfg* file to avoid having to be so verbose in the inventory file. In later chapters, we'll see how to use Ansible variables to similar effect.

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 (all on one line):

```
testserver ansible_host=ec2-203-0-113-120.compute-1.amazonaws.com \
 ansible_user=ubuntu ansible_private_key_file=/path/to/keyfile.pem
```

Ansible supports the ssh-agent program, so you don't need to explicitly specify SSH key files in your inventory files. See "SSH Agent" 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 very 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 *hosts* and invoke the `ping` module:

```
$ ansible testserver -i hosts -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 succeeded, output will look like this:

```
testserver | success >> {
    "changed": false,
    "ping": "pong"
}
```

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

```
$ ansible testserver -i hosts -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"` text is output that is specific to the *ping* module.

The ping module doesn't actually do anything other than check that Ansible can start an SSH session with the servers. It's a useful tool for testing that Ansible can connect to the server.

Simplifying with the ansible.cfg File

We had to type a lot of text in the inventory file to tell Ansible about our test server. Fortunately, Ansible has a number of ways you can specify these sorts of variables so we don't have to put them all in one place.

Right now, we'll use one such mechanism, the *ansible.cfg* file, to set some defaults so we don't need to type as much.

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*

I typically put an *ansible.cfg* in the current directory, alongside my playbooks. That way, I can check it into the same version control repository my playbooks are in.

Example 1-2 shows an *ansible.cfg* file that specifies the location of the inventory file (*inventory*), the user to SSH (*remote_user*), and the SSH private key (*private_key_file*). This assumes you're using Vagrant. If you're using your own server, you'll need to set the *remote_user* and *private_key_file* values accordingly.

Our example 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 recreate a Vagrant machine. However, disabling host key checking can be a security risk when connecting to other servers over the network. If you're not familiar with host keys, they are covered in detail in ???.

Example 1-2. ansible.cfg

```
[defaults]
inventory = hosts
remote_user = vagrant
private_key_file = .vagrant/machines/default/virtualbox/private_key
host_key_checking = False
```

Ansible and Version Control

Ansible uses */etc/ansible/hosts* as the default location for the inventory file. However, I never use this because I like to keep my inventory files version controlled alongside my playbooks.

Although we don't cover the topic of version control in this book, I strongly recommend you use a version control system like Git for maintaining all of 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 to get started.

With our default values set, we no longer need to specify the SSH user or key file in our *hosts* file. Instead, it simplifies to:

```
testserver ansible_host=127.0.0.1 ansible_port=2222
```

We can also invoke ansible without passing the `-i hostname` arguments, like so:

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

I 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 our server, we can use:

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

Output should look like this:

```
testserver | success | rc=0 >>
 17:14:07 up  1:16,  1 user,  load average: 0.16, 0.05, 0.04
```

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

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

If our command contains spaces, we need to quote it so that the shell passes the entire string as a single argument to Ansible. For example, to view the last several lines of the */var/log/dmesg* logfile:

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

The output from my Vagrant machine looks like this:

```
testserver | success | rc=0 >>
[    5.170544] type=1400 audit(1409500641.335:9): apparmor="STATUS" operation=
"profile_replace" profile="unconfined" name="/usr/lib/NetworkManager/nm-dhcp-c
lient.act on" pid=888 comm="apparmor_parser"
[    5.170547] type=1400 audit(1409500641.335:10): apparmor="STATUS" operation=
"profile_replace" profile="unconfined" name="/usr/lib/connman/scripts/dhclient-
script" pid=888 comm="apparmor_parser"
[    5.222366] vboxvideo: Unknown symbol drm_open (err 0)
[    5.222370] vboxvideo: Unknown symbol drm_poll (err 0)
[    5.222372] vboxvideo: Unknown symbol drm_pci_init (err 0)
[    5.222375] vboxvideo: Unknown symbol drm_ioctl (err 0)
[    5.222376] vboxvideo: Unknown symbol drm_vblank_init (err 0)
[    5.222378] vboxvideo: Unknown symbol drm_mmap (err 0)
[    5.222380] vboxvideo: Unknown symbol drm_pci_exit (err 0)
[    5.222381] vboxvideo: Unknown symbol drm_release (err 0)
```

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

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

The output looks something like this:

```
testserver | success | rc=0 >>
Aug 31 15:57:49 vagrant-ubuntu-trusty-64 ntpdate[1465]: /
adjust time server 91.189
94.4 offset -0.003191 sec
Aug 31 16:17:01 vagrant-ubuntu-trusty-64 CRON[1480]: (root) CMD (   cd /
&& run-p
rts --report /etc/cron.hourly)
Aug 31 17:04:18 vagrant-ubuntu-trusty-64 ansible-ping: Invoked with data=None
Aug 31 17:12:33 vagrant-ubuntu-trusty-64 ansible-ping: Invoked with data=None
Aug 31 17:14:07 vagrant-ubuntu-trusty-64 ansible-command: Invoked with executable
None shell=False args=uptime removes=None creates=None chdir=None
Aug 31 17:16:01 vagrant-ubuntu-trusty-64 ansible-command: Invoked with executable
None shell=False args=tail /var/log/messages removes=None creates=None chdir=None
Aug 31 17:17:01 vagrant-ubuntu-trusty-64 CRON[2091]: (root) CMD (   cd /
&& run-pa
rts --report /etc/cron.hourly)
Aug 31 17:17:09 vagrant-ubuntu-trusty-64 ansible-command: Invoked with /
executable=
N one shell=False args=tail /var/log/dmesg removes=None creates=None chdir=None
Aug 31 17:19:01 vagrant-ubuntu-trusty-64 ansible-command: Invoked with /
executable=
None shell=False args=tail /var/log/messages removes=None creates=None chdir=None
Aug 31 17:22:32 vagrant-ubuntu-trusty-64 ansible-command: Invoked with /
executable=
one shell=False args=tail /var/log/syslog removes=None creates=None chdir=None
```

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

You aren't just 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 using the follow command:

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

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 by doing:

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

We need the `-b` argument to become the root user because only root can install the nginx package and restart services.

Moving Forward

To recap, in this introductory chapter, we've covered the basic concepts of Ansible at a high level, including how it communicates with remote servers and how it differs from other configuration management tools. We'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. In the next chapter, we'll cover playbooks, where the real action is.

CHAPTER 2

Playbooks: A Beginning

Most of your time in Ansible will be spent writing *playbooks*. A playbook is the term that Ansible uses for a configuration management script. Let's look at an example: installing the nginx web server and configuring it for secure communication.

If you're following along in this chapter, you should end up with the files listed here:

- *playbooks/ansible.cfg*
- *playbooks/hosts*
- *playbooks/Vagrantfile*
- *playbooks/web-notls.yml*
- *playbooks/web-tls.yml*
- *playbooks/files/nginx.key*
- *playbooks/files/nginx.crt*
- *playbooks/files/nginx.conf*
- *playbooks/templates/index.html.j2*
- *playbooks/templates/nginx.conf.j2*

Some Preliminaries

Before we can run this playbook against our Vagrant machine, we need to expose ports 80 and 443, so we can access them. As shown in Figure 2-1, we are going to configure Vagrant so that requests to ports 8080 and 8443 on our local machine are forwarded 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*.

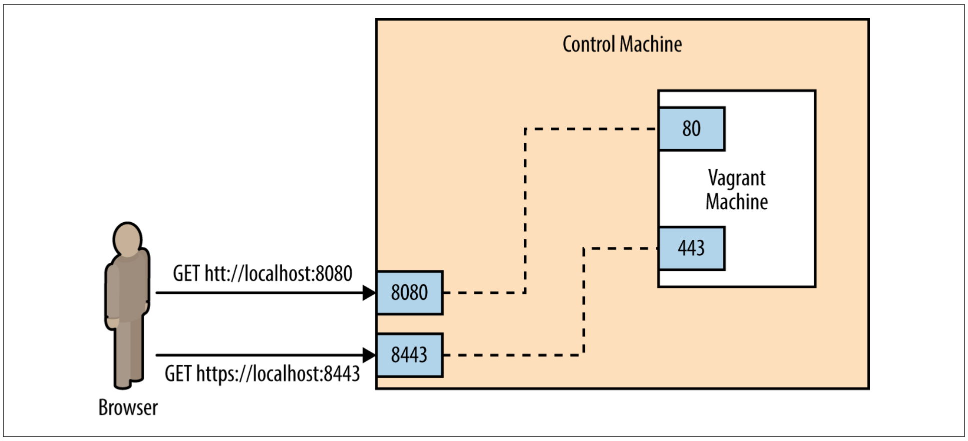


Figure 2-1. Exposing ports on Vagrant machine

Modify your *Vagrantfile* so it looks like this:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.network "forwarded_port", guest: 80, host: 8080
  config.vm.network "forwarded_port", guest: 443, host: 8443
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. Once you make the changes, tell Vagrant to have them go into effect by running:

```
$ vagrant reload
```

You should see output that includes:

```
==> default: Forwarding ports...
    default: 80 => 8080 (adapter 1)
    default: 443 => 8443 (adapter 1)
    default: 22 => 2222 (adapter 1)
```

A Very Simple Playbook

For our first example playbook, we'll configure a host to run an nginx web server. For this example, we won't configure the web server to support TLS encryption. This will make setting up the web server simpler, but a proper website should have TLS encryption enabled, and we'll cover how to do that later on in this chapter.

First, we'll see what happens when we run the playbook in Example 2-1, and then we'll go over the contents of the playbook in detail.

Example 2-1. web-notls.yml

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

    - name: copy nginx config file
      copy: src=files/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=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
        mode=0644

    - name: restart nginx
      service: name=nginx state=restarted
```

Why Do You Use "True" in One Place and "Yes" in Another?

Sharp-eyed readers might have noticed that Example 2-1 uses `True` in one spot in the playbook (to enable sudo) and `yes` in another spot in the playbook (to update the apt cache).

Ansible is pretty flexible on how you represent truthy and falsey values in playbooks. Strictly speaking, module arguments (like `update_cache=yes`) are treated differently from values elsewhere in playbooks (like `sudo: True`). Values elsewhere are handled by the YAML parser and so use the YAML conventions of truthiness, which are:

YAML truthy
: `true`, `True`, `TRUE`, `yes`, `Yes`, `YES`, `on`, `On`, `ON`, `y`, `Y`

YAML falsey
: `false`, `False`, `FALSE`, `no`, `No`, `NO`, `off`, `Off`, `OFF`, `n`, `N`

Module arguments are passed as strings and use Ansible's internal conventions, which are:

module arg truthy
: `yes`, `on`, `1`, `true`

module arg falsey
: `no`, `off`, `0`, `false`

I tend to follow the examples in the official Ansible documentation. These typically use `yes` and `no` when passing arguments to modules (since that's consistent with the module documentation), and `True` and `False` elsewhere in playbooks.

Specifying an nginx Config File

This playbook requires two additional files before we can run it. First, we need to define 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 almost always need to customize this, so we'll overwrite the default configuration file with our own as part of this playbook. As we'll see later, we'll need to modify this configuration file to support TLS. Example 2-2 shows a basic nginx config file. Put it in *playbooks/files/nginx.conf*.[1]

Example 2-2. files/nginx.conf

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

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

        server_name localhost;

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

An Ansible convention is to keep files in a subdirectory named *files* and Jinja2 templates in a subdirectory named *templates*. I'll follow this convention throughout the book.

1 Note that while 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.

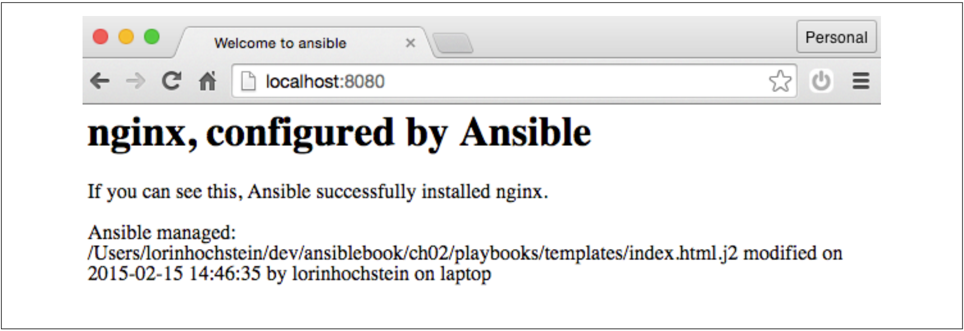
Creating a Custom Homepage

Let's add a custom homepage. We're going to use Ansible's template functionality so that Ansible will generate the file from a template. Put the file 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>{{ ansible_managed }}</p>
  </body>
</html>
```

This template references a special Ansible variable named *ansible_managed*. When Ansible renders this template, it will replace this variable with information about when the template file was generated. Figure 2-2 shows a screenshot of a web browser viewing the generated HTML.



nginx, configured by Ansible

If you can see this, Ansible successfully installed nginx.

Ansible managed:
/Users/lorinhochstein/dev/ansiblebook/ch02/playbooks/templates/index.html.j2 modified on
2015-02-15 14:46:35 by lorinhochstein on laptop

Figure 2-2. Rendered HTML

Creating a Webservers 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 contain our testserver.

Inventory files are in the *.ini* file format. We'll go into this format in detail later in the book. Edit your *playbooks/hosts* file to put a `[webservers]` line above the `testserver` line, as shown in Example 2-4. This indicates that testserver is in the webservers group.

Example 2-4. playbooks/hosts

```
[webservers]
testserver ansible_host=127.0.0.1 ansible_port=2222
```

You should now be able to ping the webservers group using the `ansible` command-line tool:

```
$ ansible webservers -m ping
```

The output should look like this:

```
testserver | success >> {
    "changed": false,
    "ping": "pong"
}
```

Running the Playbook

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

```
$ ansible-playbook web-notls.yml
```

Example 2-5 shows what the output should look.

Example 2-5. Output of ansible-playbook

```
PLAY [Configure webserver with nginx] *********************************

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
```

Cowsay

If you have the *cowsay* program installed on your local machine, then Ansible output will look like this instead:

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

If you don't want to see the cows, you can disable cowsay by setting the `ANSIBLE_NOCOWS` environment variable like this:

```
$ export ANSIBLE_NOCOWS=1
```

You can also disable cowsay by adding the following to your *ansible.cfg* file.

```
[defaults]
nocows = 1
```

If you didn't get any errors,[2] you should be able to point your browser to *http://localhost:8080* and see the custom HTML page, as shown in Figure 2-2.

If your playbook file is marked as executable and starts with a line that looks like this:[3]

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

then you can execute it by invoking it directly, like this:

```
$ ./web-notls.yml
```

Playbooks Are YAML

Ansible playbooks are written in YAML syntax. YAML is a file format similar in intent to JSON, but generally easier for humans to read and write. Before we go over the playbook, let's cover the concepts of YAML that are most important for writing playbooks.

2 If you encountered an error, you might want to skip to ??? for assistance on debugging.

3 Colloquially referred to as a "shebang."

Start of File

YAML files are supposed to start with three dashes to indicate the beginning of the document:

```
---
```

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

Comments

Comments start with a number sign and apply to the end of the line, the same as in shell scripts, Python, and Ruby:

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

Strings

In general, YAML strings don't have to be quoted, although you can 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:

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

There are some scenarios in Ansible where you will need to quote strings. These typically involve the use of `{{ braces }}` for variable substitution. We'll get to those later.

Booleans

YAML has a native Boolean type, and provides you with a wide variety of strings that can be interpreted as true or false, which we covered in "Why Do You Use "True" in One Place and "Yes" in Another?" on page 23.

Personally, I always use `True` and `False` in my Ansible playbooks.

For example, this is a Boolean in YAML:

```
True
```

The JSON equivalent is:

```
true
```

Lists

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

They are delimited with hyphens, like this:

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

The JSON equivalent is:

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

(Note again how we didn't have to quote the strings in YAML, even though they have spaces in them.)

YAML also supports an inline format for lists, which looks like this:

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

Dictionaries

YAML *dictionaries* are like objects in JSON, dictionaries in Python, or hashes in Ruby. Technically, these are called *mappings* in YAML, but I call them *dictionaries* here to be consistent with the official Ansible documentation.

They look like this:

```
address: 742 Evergreen Terrace
city: Springfield
state: North Takoma
```

The JSON equivalent is:

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

YAML also supports an inline format for dictionaries, which looks like this:

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

Line Folding

When writing playbooks, you'll often encounter 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, but you want Ansible to treat the string as if it were a single line.

You can do this with YAML using line folding with the greater than (>) character. The YAML parser will replace line breaks with spaces. For example:

```
address: >
    Department of Computer Science,
    A.V. Williams Building,
    University of Maryland
city: College Park
state: Maryland
```

The JSON equivalent is:

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

Anatomy of a Playbook

Let's take a look at our playbook from the perspective of a YAML file. Here it is again, in Example 2-6.

Example 2-6. web-notls.yml

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

    - name: copy nginx config file
      copy: src=files/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=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
```

```
        mode=0644

    - name: restart nginx
      service: name=nginx state=restarted
```

In Example 2-7, we see the JSON equivalent of this file.

Example 2-7. JSON equivalent of web-notls.yml

```
[
  {
    "name": "Configure webserver with nginx",
    "hosts": "webservers",
    "become": true,
    "tasks": [
      {
        "name": "Install nginx",
        "apt": "name=nginx update_cache=yes"
      },
      {
        "name": "copy nginx config file",
        "template": "src=files/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=templates/index.html.j2 dest=/usr/share/nginx/html/
        index.html mode=0644"
      },
      {
        "name": "restart nginx",
        "service": "name=nginx state=restarted"
      }
    ]
  }
]
```

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 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.

Plays

Looking at either the YAML or JSON representation, it should be clear that a playbook is a list of dictionaries. Specifically, a playbook is a list of *plays*.

Here's the play[4] from our example:

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

    - name: copy nginx config file
      copy: src=files/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=templates/index.html.j2
                dest=/usr/share/nginx/html/index.html mode=0644


    - name: restart nginx
      service: name=nginx state=restarted
```

Every play must contain:

- A set of *hosts* to configure
- A list of *tasks* to be executed on those hosts

Think of a play as the thing that connects hosts to tasks.

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

`name`
: A comment that describes what the play is about. Ansible will print this out when the play starts to run.

4 Actually, it's a list that contains a single play.

`become`
: If true, Ansible will run every task by becoming (by default) the root user. This is useful when managing Ubuntu servers, since by default you cannot SSH as the root user.

`vars`
: A list of variables and values. We'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
  apt: name=nginx update_cache=yes
```

The `name` is optional, so it's perfectly valid to write a task like this:

```
- apt: name=nginx update_cache=yes
```

Even though names are optional, I recommend you use them because they serve as good reminders for the intent of the task. (Names will be very useful when somebody else is trying to understand your playbook, including yourself in six months.) As we've seen, Ansible will print out the name of a task when it runs. Finally, as we'll see in ???, 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.

Every task must contain a key with the name of a module and a value with the arguments to that module. In the preceding example, the module name is *apt* and the arguments are `name=nginx update_cache=yes`.

These arguments tell the apt module to install the package named *nginx* and to update the package cache (the equivalent of doing an `apt-get update`) before installing the package.

It's important to understand that, from the point of the view of the YAML parser used by the Ansible frontend, the arguments are treated as a string, not as a dictionary. This means that if you want to break up arguments into multiple lines, you need to use the YAML folding syntax, like this:

```
- name: install nginx
  apt: >
      name=nginx
      update_cache=yes
```

Ansible also supports a task syntax that will let you specify module arguments as a YAML dictionary, which is helpful when using modules that support complex arguments. We'll cover that in ???.

Ansible also supports an older syntax that uses `action` as the key and puts the name of the module in the value. The preceding example also can be written as:

```
- name: install nginx
  action: apt name=nginx update_cache=yes
```

Modules

Modules are scripts[5] that come packaged with Ansible and perform some kind of action on a host. Admittedly, that's a pretty generic description, but there's enormous variety across Ansible modules. The modules we use in this chapter are:

apt
: Installs or removes packages using the apt package manager.

copy
: Copies a file from local machine to the hosts.

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 modules. Think of it as man pages for Ansible modules. For example, to show the documentation for the *service* module, run:

```
$ ansible-doc service
```

If you use Mac OS X, there's a wonderful documentation viewer called Dash that has support for Ansible. Dash indexes all of the Ansible module documentation. It's a commercial tool ($19.99 as of this writing), but I find it invaluable.

Recall from the first chapter 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.

5 The modules that ship with Ansible all are written in Python, but modules can be written in any language.

There are over 200 modules that ship with Ansible, and this number grows with every release. You can also find third-party Ansible modules out there, or write your own.

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 is an entity-relationship diagram that depicts this relationship between playbooks, plays, hosts, tasks, and modules.

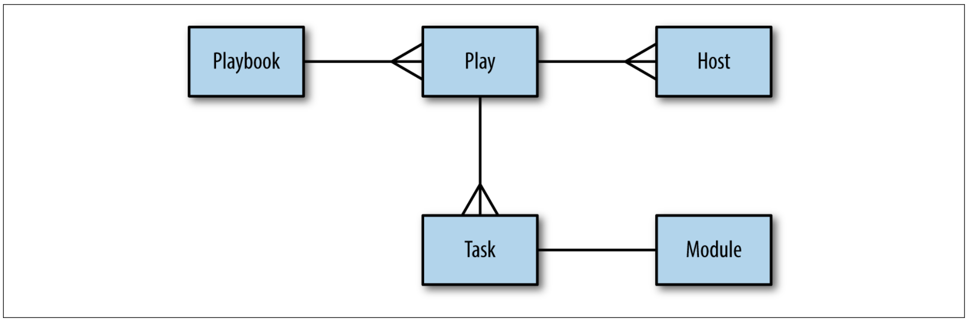


Figure 2-3. Entity-relationship diagram

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 Example 2-5, notice that the status for some of the tasks is *changed*, and the status for some others is *ok*. For example, the `install nginx` task has status *changed*, which appears as yellow on my terminal.

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

The `enable configuration`, on the other hand, has 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 if the state of the host needs to be changed before taking any action. If the state of the host matches the arguments of the module, then 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 state of the host and the arguments to the module, then Ansible will change the state of the host and return *changed*.

In the example output just shown, the *install nginx* task was changed, which meant 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 configuration file on the server that was identical to the file I was copying over. The reason for this is that the *nginx.conf* file I used in my playbook is the same as the *nginx.conf* file that gets installed by the *nginx* package on Ubuntu.

As we'll see later in this chapter, Ansible's detection of state change can be used to trigger additional actions through the use of *handlers*. But, even without using handlers, it is still a useful form of feedback to see whether your hosts are changing state as the playbook runs.

Getting Fancier: TLS Support

Let's move on to a more complex example: We're going to modify the previous playbook so that our webservers support TLS. The new features here are:

- Variables
- Handlers

TLS versus SSL

You might be familiar with the term *SSL* rather than *TLS* in the context of secure web servers. SSL is an older protocol that was used to secure communications between browsers and web servers, and it has been superseded by a newer protocol named TLS.

Although many continue to use the term *SSL* to refer to the current secure protocol, in this book, I use the more accurate *TLS*.

Example 2-8 shows what our playbook looks like with TLS support.

Example 2-8. web-tls.yml

```
- name: Configure webserver with nginx and tls
  hosts: webservers
  become: True
  vars:
    key_file: /etc/nginx/ssl/nginx.key
    cert_file: /etc/nginx/ssl/nginx.crt
    conf_file: /etc/nginx/sites-available/default
    server_name: localhost
  tasks:
    - name: Install nginx
      apt: name=nginx update_cache=yes cache_valid_time=3600

    - name: create directories for ssl certificates
      file: path=/etc/nginx/ssl state=directory

    - name: copy TLS key
      copy: src=files/nginx.key dest={{ key_file }} owner=root mode=0600
      notify: restart nginx

    - name: copy TLS certificate
      copy: src=files/nginx.crt dest={{ cert_file }}
      notify: restart nginx

    - name: copy nginx config file
      template: src=templates/nginx.conf.j2 dest={{ conf_file }}
      notify: restart nginx

    - name: enable configuration
      file: dest=/etc/nginx/sites-enabled/default src={{ conf_file }} state=link
      notify: restart nginx

    - name: copy index.html
      template: src=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
             mode=0644
  handlers:
    - name: restart nginx
      service: name=nginx state=restarted
```

Generating TLS certificate

We need to manually generate a TLS certificate. In a production environment, you'd purchase your TLS certificate from a certificate authority, or use a free service like Let's Encrypt which Ansible supports via the `letsencrypt` module. We'll use a self-signed certificate, since we can generate those for free.

Create a *files* subdirectory of your *playbooks* directory, and then generate the TLS certificate and key:

```
$ mkdir files
$ openssl req -x509 -nodes -days 3650 -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* directory. The certificate has an expiration date of 10 years (3,650 days) from the day you created it.

Variables

The play in our playbook now has a section called *vars*:

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

This section defines four variables and assigns a value to each variable.

In our example, each value is a string (e.g., `/etc/nginx/ssl/nginx.key`), 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 using the `{{ braces }}` notation. Ansible will replace these braces with the value of the variable.

Consider this task in the playbook:

```
- name: copy TLS key
  copy: src=files/nginx.key dest={{ key_file }} owner=root mode=0600
```

Ansible will substitute `{{ key_file }}` with `/etc/nginx/ssl/nginx.key` when it executes this task.

When Quoting Is Necessary

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

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

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

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

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

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

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

Unfortunately, just quoting the argument string won't resolve the problem, either.

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

This will make the YAML parser happy, but the output isn't what you expect:

```
TASK: [show a debug message] ************************************************
ok: [localhost] => {
    "msg": "The"
}
```

The `debug` module's *msg* argument requires a quoted string to capture the spaces. In this particular case, we need to quote both the whole argument string and the *msg* argument. Ansible supports alternating single and double quotes, so you can do this:

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

This yields the expected output:

```
TASK: [show a debug message] ************************************************
ok: [localhost] => {
    "msg": "The debug module will print a message: neat, eh?"
}
```

Ansible is pretty good at generating meaningful error messages if you forget to put quotes in the right places and end up with invalid YAML.

Generating the Nginx Configuration Template

If you've done web programming, you've likely used a template system to generate HTML. In case you haven't, a template is just a text file that has some special syntax for specifying variables that should be replaced by values. If you've ever received an automated email from a company, they're probably using an email template as shown in Example 2-9.

Example 2-9. An email template

```
Dear {{ name }},

You have {{ num_comments }} new comments on your blog: {{ blog_name }}.
```

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. If you've ever used a templating library such as Mustache, ERB, or the Django template system, 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 hard-coding 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 2-10. templates/nginx.conf.j2

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

        listen 443 ssl;

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

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

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

We 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 three variables:

`server_name`
: The hostname of the web server (e.g., `www.example.com`)

`cert_file`
: The path to the TLS certificate

`key_file`
: The path to the TLS private key

We define these variables in the playbook.

Ansible also uses the Jinja2 template engine to evaluate variables in playbooks. Recall that we saw the `{{ conf_file }}` syntax in the playbook itself.

> Early versions of Ansible used a dollar sign (`$`) to do variable interpolation in playbooks instead of the braces. You used to dereference variable *foo* by writing `$foo`, where now you write `{{ foo }}`. The dollar sign syntax has been deprecated; if you encounter it in an example playbook you find on the Internet, then you're looking at older Ansible code.

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.

Handlers

Looking back at our *web-tls.yml* playbook, note that there are two new playbook elements we haven't discussed yet. 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` key. For example:

```
    - name: copy TLS key
      copy: src=files/nginx.key dest={{ key_file }} owner=root mode=0600
      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[6] 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 of the tasks 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 after all of the tasks are run at the end of the play. They only run once, even if they are notified multiple times. 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.

The official Ansible docs mention that the only common uses for handlers are for restarting services and for reboots. Personally, I've only ever used them for restarting services. Even then, it's a pretty small optimization, since we can always just unconditionally restart the service at the end of the playbook instead of notifying it on change, and restarting a service doesn't usually take very long.

Another pitfall with handlers that I've encountered is that they can be troublesome when debugging a playbook. It goes something like this:

1. I run a playbook.
2. One of my tasks with a *notify* on it changes state.
3. An error occurs on a subsequent task, stopping Ansible.
4. I fix the error in my playbook.
5. I run Ansible again.

6 Alternatively, we could reload the configuration file using `state=reloaded` instead of restarting the service.

6. None of the tasks report a state change the second time around, so Ansible doesn't run the handler.

Read more about advanced handler usages and applications in ???.

Running the Playbook

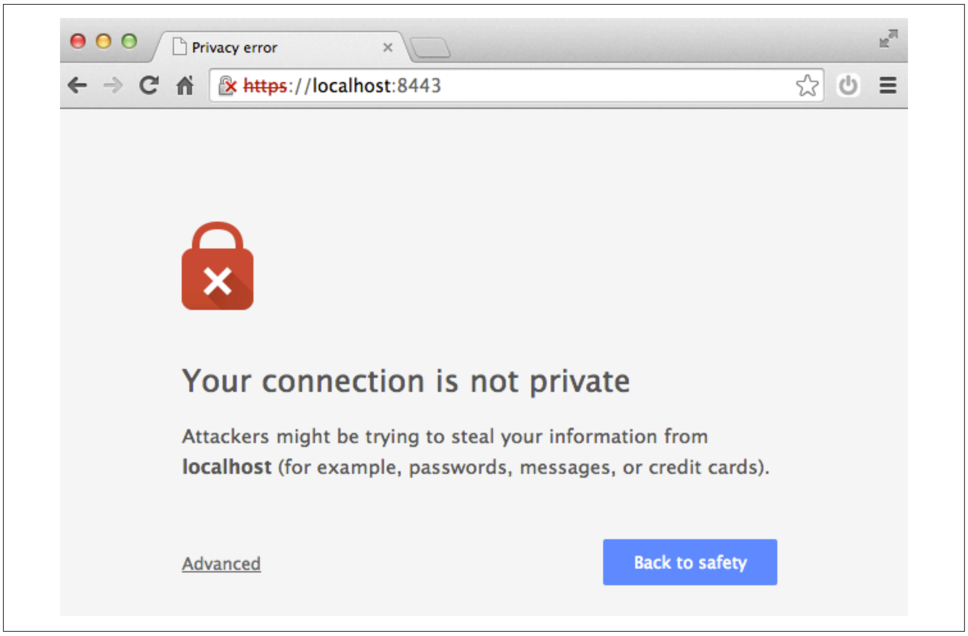
As before, we use the `ansible-playbook` command to run the playbook.

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

The output should look something like this:

```
PLAY [Configure webserver with nginx and tls] *********************************

GATHERING FACTS ***************************************************************
ok: [testserver]

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

TASK: [create directories for tls certificates] *******************************
changed: [testserver]

TASK: [copy TLS key] **********************************************************
changed: [testserver]

TASK: [copy TLS certificate] **************************************************
changed: [testserver]

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

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

NOTIFIED: [restart nginx] *****************************************************
changed: [testserver]

PLAY RECAP ********************************************************************
testserver                 : ok=8    changed=6    unreachable=0    failed=0
```

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


Privacy error

https://localhost:8443

Your connection is not private

Attackers might be trying to steal your information from **localhost** (for example, passwords, messages, or credit cards).

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

Don't worry, though; that error is expected, as we generated a self-signed TLS certificate, and web browsers like Chrome only trust certificates that have been issued from a proper authority.

We covered a lot of the "what" of Ansible in this chapter, describing what Ansible will do to your hosts. The handlers we discussed here are just one form of control flow that Ansible supports. In a later chapter, we'll see iteration and conditionally running tasks based on the values of variables.

In the next chapter, we'll talk about the "who"; in other words, how to describe the hosts that your playbooks will run against.

CHAPTER 3
Inventory: Describing Your Servers

So far, we've been working with only one server (or *host*, as Ansible calls it). In reality, you're going to be managing multiple hosts. The collection of hosts that Ansible knows about is called the *inventory*.

The Inventory File

The default way to describe your hosts in Ansible is to list them in text files, called *inventory files*. A very simple inventory file might just contain a list of hostnames, as shown in Example 3-1.

Example 3-1. A very simple inventory file

```
ontario.example.com
newhampshire.example.com
maryland.example.com
virginia.example.com
newyork.example.com
quebec.example.com
rhodeisland.example.com
```

Ansible uses your local SSH client by default, which means that it will understand any aliases that you set up in your SSH config file. This does not hold true if you configure Ansible to use the Paramiko connection plug-in instead of the default SSH plug-in.

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

Although Ansible adds the localhost to your inventory automatically, you have to have at least one other host in your inventory file; otherwise, ansible-playbook will terminate with the error:

```
ERROR: provided hosts list is empty
```

In the case where you have no other hosts in your inventory file, you can explicitly add an entry for localhost like this:

```
localhost ansible_connection=local
```

Preliminaries: Multiple Vagrant Machines

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

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

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

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

Next, edit your Vagrantfile so it looks like Example 3-2.

Example 3-2. Vagrantfile with three servers

```
VAGRANTFILE_API_VERSION = "2"

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

  config.vm.define "vagrant1" do |vagrant1|
    vagrant1.vm.box = "ubuntu/trusty64"
    vagrant1.vm.network "forwarded_port", guest: 80, host: 8080
    vagrant1.vm.network "forwarded_port", guest: 443, host: 8443
  end
  config.vm.define "vagrant2" do |vagrant2|
    vagrant2.vm.box = "ubuntu/trusty64"
    vagrant2.vm.network "forwarded_port", guest: 80, host: 8081
    vagrant2.vm.network "forwarded_port", guest: 443, host: 8444
  end
  config.vm.define "vagrant3" do |vagrant3|
    vagrant3.vm.box = "ubuntu/trusty64"
    vagrant3.vm.network "forwarded_port", guest: 80, host: 8082
    vagrant3.vm.network "forwarded_port", guest: 443, host: 8445
  end
end
```

Vagrant 1.7+ defaults to using a different SSH key for each host. Example 3-2 contains the line to revert to the earlier behavior of using the same SSH key for each host:

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

Using the same key on each host simplifies our Ansible setup because we can specify a single SSH key in the *ansible.cfg* file. You'll need to edit the *host_key_checking* value in your *ansible.cfg*. Your file should look like Example 3-3.

Example 3-3. ansible.cfg

```
[defaults]
inventory = inventory
remote_user = vagrant
private_key_file = ~/.vagrant.d/insecure_private_key
host_key_checking = False
```

For now, we'll assume each of these servers can potentially be a web server, so Example 3-2 maps ports 80 and 443 inside each Vagrant machine to a port on the local machine.

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

```
$ vagrant up
```

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

```
Bringing machine 'vagrant1' up with 'virtualbox' provider...
Bringing machine 'vagrant2' up with 'virtualbox' provider...
Bringing machine 'vagrant3' up with 'virtualbox' provider...
...
    vagrant3: 80 => 8082 (adapter 1)
    vagrant3: 443 => 8445 (adapter 1)
    vagrant3: 22 => 2201 (adapter 1)
==> vagrant3: Booting VM...
==> vagrant3: Waiting for machine to boot. This may take a few minutes...
    vagrant3: SSH address: 127.0.0.1:2201
    vagrant3: SSH username: vagrant
    vagrant3: SSH auth method: private key
    vagrant3: Warning: Connection timeout. Retrying...
==> vagrant3: Machine booted and ready!
==> vagrant3: Checking for guest additions in VM...
==> vagrant3: Mounting shared folders...
    vagrant3: /vagrant => /Users/lorin/dev/oreilly-ansible/playbooks
```

Let's create an inventory file that contains these three machines.

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

```
$ vagrant ssh-config
```

The output should look something like this:

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

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

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

We can see that `vagrant1` uses port 2222, `vagrant2` uses port 2200, and `vagrant3` uses port 2201.

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

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

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

```
$ ansible vagrant2 -a "ip addr show dev eth0"
```

On my machine, the output looks like this:

```
vagrant2 | success | rc=0 >>
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
group default qlen 1000
    link/ether 08:00:27:fe:1e:4d brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global eth0
```

```
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fefe:1e4d/64 scope link
       valid_lft forever preferred_lft forever
```

Behavioral Inventory Parameters

To describe our Vagrant machines in the Ansible inventory file, we had to explicitly specify the hostname (127.0.0.1) and port (2222, 2200, or 2201) that Ansible's SSH client should connect to.

Ansible calls these variables *behavioral inventory parameters*, and there are several of them you can use when you need to override the Ansible defaults for a host (see Table 3-1).

Table 3-1. Behavioral inventory parameters

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

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

ansible_connection

Ansible supports multiple *transports*, which are mechanisms that Ansible uses to connect to the host. The default transport, *smart*, will check to see if the locally installed SSH client supports a feature called *ControlPersist*. If the SSH client supports ControlPersist, Ansible will use the local SSH client. If the SSH client doesn't support ControlPersist, then the smart transport will fall back to using a Python-based SSH client library called *paramiko*.

ansible_shell_type

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

Ansible also accepts `csh`, `fish`, and (on Windows) `powershell` as valid values for this parameter. I've never encountered a need for changing the shell type.

ansible_python_interpreter

Because the modules that ship with Ansible are implemented in Python 2, Ansible needs to know the location of the Python interpreter on the remote machine. You might need to change this if your remote host does not have a Python 2 interpreter at */usr/bin/python*. For example, if you are managing hosts that run Arch Linux, you will need to change this to */usr/bin/python2*, because Arch Linux installs Python 3 at */usr/bin/python*, and Ansible modules are not (yet) compatible with Python 3.

ansible_*_interpreter

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

Changing Behavioral Parameter Defaults

You can override some of the behavioral parameter default values in the `[defaults]` section of the *ansible.cfg* file (Table 3-2). Recall that we used this previously to change the default SSH user.

Table 3-2. Defaults that can be overridden in ansible.cfg

Behavioral inventory parameter	ansible.cfg option
ansible_port	remote_port
ansible_user	remote_user
ansible_private_key_file	private_key_file
ansible_shell_type	executable (see the following paragraph)

The ansible.cfg `executable` config option is not exactly the same as the `ansible_shell_type` behavioral inventory parameter. Instead, the executable specifies the full path of the shell to use on the remote machine (e.g., */usr/local/bin/fish*). Ansible will look at the name of the base name of this path (in the case of */usr/local/bin/fish*, the basename is *fish*) and use that as the default value for `ansible_shell_type`.

Groups and Groups and Groups

When performing configuration tasks, we typically want to perform actions on groups of hosts, rather than on an individual host.

Ansible automatically defines a group called *all* (or *), which includes all of the hosts in the inventory. For example, we can check if the clocks on the machines are roughly synchronized by running:

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

or

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

The output on my system looks like this:

```
vagrant3 | success | rc=0 >>
Sun Sep  7 02:56:46 UTC 2014

vagrant2 | success | rc=0 >>
Sun Sep  7 03:03:46 UTC 2014

vagrant1 | success | rc=0 >>
Sun Sep  7 02:56:47 UTC 2014
```

We can define our own groups in the inventory file. Ansible uses the *.ini* file format for inventory files. In the *.ini* format, configuration values are grouped together into sections.

Here's how we would specify that our vagrant hosts are in a group called *vagrant*, along with the other example hosts we mentioned at the beginning of the chapter:

```
ontario.example.com
newhampshire.example.com
maryland.example.com
virginia.example.com
newyork.example.com
quebec.example.com
rhodeisland.example.com

[vagrant]
vagrant1 ansible_host=127.0.0.1 ansible_port=2222
vagrant2 ansible_host=127.0.0.1 ansible_port=2200
vagrant3 ansible_host=127.0.0.1 ansible_port=2201
```

We could have also listed the vagrant hosts at the top, and then also in a group, like this:

```
maryland.example.com
newhampshire.example.com
newyork.example.com
ontario.example.com
```

```
quebec.example.com
rhodeisland.example.com
vagrant1 ansible_host=127.0.0.1 ansible_port=2222
vagrant2 ansible_host=127.0.0.1 ansible_port=2200
vagrant3 ansible_host=127.0.0.1 ansible_port=2201
virginia.example.com

[vagrant]
vagrant1
vagrant2
vagrant3
```

Example: Deploying a Django App

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

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

In later chapters, we will work through a detailed example of deploying this kind of Django-based application, although our example won't use Celery or RabbitMQ.

We need to deploy this application into different types of environments: production (the real thing), staging (for testing on hosts that our team has shared access to), and vagrant (for local testing).

When we deploy to production, we want the entire system to respond quickly and be reliable, so we:

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

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

For our staging environment, imagine that we want to use fewer hosts than we do in production in order to save costs, especially since the staging environment is going to see a lot less activity than production. Let's say we decide to use only two hosts for staging; we'll put the web server and task queue on one staging host, and RabbitMQ and Postgres on the other.

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

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

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

```
[production]
delaware.example.com
georgia.example.com
maryland.example.com
newhampshire.example.com
newjersey.example.com
newyork.example.com
northcarolina.example.com
pennsylvania.example.com
rhodeisland.example.com
virginia.example.com

[staging]
ontario.example.com
quebec.example.com

[vagrant]
vagrant1 ansible_host=127.0.0.1 ansible_port=2222
vagrant2 ansible_host=127.0.0.1 ansible_port=2200
vagrant3 ansible_host=127.0.0.1 ansible_port=2201

[lb]
delaware.example.com

[web]
georgia.example.com
newhampshire.example.com
newjersey.example.com
ontario.example.com
vagrant1

[task]
newyork.example.com
northcarolina.example.com
maryland.example.com
ontario.example.com
vagrant2
```

```
[rabbitmq]
pennsylvania.example.com
quebec.example.com
vagrant3

[db]
rhodeisland.example.com
virginia.example.com
quebec.example.com
vagrant3
```

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

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

Aliases and Ports

We described our Vagrant hosts like this:

```
[vagrant]
vagrant1 ansible_host=127.0.0.1 ansible_port=2222
vagrant2 ansible_host=127.0.0.1 ansible_port=2200
vagrant3 ansible_host=127.0.0.1 ansible_port=2201
```

The names `vagrant1`, `vagrant2`, and `vagrant3` here are *aliases*. They are not the real hostnames, but instead are useful names for referring to these hosts.

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

However, we can't actually run what you see in Example 3-5.

Example 3-5. This doesn't work

```
[vagrant]
127.0.0.1:2222
127.0.0.1:2200
127.0.0.1:2201
```

The reason is that Ansible's inventory can associate only a single host with *127.0.0.1*, so the vagrant group would contain only one host instead of three.

Groups of Groups

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

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

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

Numbered Hosts (Pets versus Cattle)

The inventory file shown in Example 3-4 looks complex. In reality, it describes only 15 different hosts, which doesn’t sound like a large number in this cloudy scale-out world. However, even dealing with 15 hosts in the inventory file can be cumbersome because each host has a completely different hostname.

Bill Baker of Microsoft came up with the distinction between treating servers as *pets* versus treating them like *cattle*.[1] We give pets distinctive names, and we treat and care for them as individuals. On the other hand, when we discuss cattle, we refer to them by identification number.

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

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

If you prefer to have a leading zero (e.g., *web01.example.com*), then specify a leading zero in the range, like this:

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

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

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

1 This term has been popularized by Randy Bias of Cloudscaling.

Hosts and Group Variables: Inside the Inventory

Recall how we specified behavioral inventory parameters for Vagrant hosts:

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

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

```
newhampshire.example.com color=red
maryland.example.com color=green
ontario.example.com color=blue
quebec.example.com color=purple
```

This variable can then be used in a playbook, just like any other variable.

Personally, I don't often attach variables to specific hosts. On the other hand, I often associate variables with groups.

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

To set everything up, we need to do the following:

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

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

Example 3-6 shows one way we can specify this information as group variables in the inventory file.

Example 3-6. Specifying group variables in inventory

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

[production:vars]
db_primary_host=rhodeisland.example.com
db_primary_port=5432
db_replica_host=virginia.example.com
db_name=widget_production
db_user=widgetuser
db_password=pFmMxcyD;Fc6)6
rabbitmq_host=pennsylvania.example.com
rabbitmq_port=5672


[staging:vars]
db_primary_host=quebec.example.com
db_primary_port=5432
db_name=widget_staging
db_user=widgetuser
db_password=L@4Ryz8cRUXedj
rabbitmq_host=quebec.example.com
rabbitmq_port=5672

[vagrant:vars]
db_primary_host=vagrant3
db_primary_port=5432
db_name=widget_vagrant
db_user=widgetuser
db_password=password
rabbitmq_host=vagrant3
rabbitmq_port=5672
```

Note how group variables are organized into sections named `[<group name>:vars]`.

Also note how we took advantage of the `all` group that Ansible creates automatically to specify variables that don't change across hosts.

Host and Group Variables: In Their Own Files

The inventory file is a reasonable place to put host and group variables if you don't have too many hosts. But as your inventory gets larger, it gets more difficult to manage variables this way.

Additionally, though Ansible variables can hold Booleans, strings, lists, and dictionaries, in an inventory file, you can specify only Booleans and strings.

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

Ansible looks for host variable files in a directory called *host_vars* and group variable files in a directory called *group_vars*. Ansible expects these directories to be either in the directory that contains your playbooks or in the directory adjacent to your inventory file. In our case, those two directories are the same.

For example, if I had a directory containing my playbooks at */home/lorin/playbooks/* with an inventory file at */home/lorin/playbooks/hosts*, then I would put variables for the *quebec.example.com* host in the file */home/lorin/playbooks/host_vars/quebec.example.com*, and I would put variables for the production group in the file */home/lorin/playbooks/group_vars/production*.

Example 3-7 shows what the */home/lorin/playbooks/group_vars/production* file would look like.

Example 3-7. group_vars/production

```
db_primary_host: rhodeisland.example.com
db_primary_port=5432
db_replica_host: virginia.example.com
db_name: widget_production
db_user: widgetuser
db_password: pFmMxcyD;Fc6)6
rabbitmq_host:pennsylvania.example.com
rabbitmq_port=5672
```

Note that we could also use YAML dictionaries to represent these values, as shown in Example 3-8.

Example 3-8. group_vars/production, with dictionaries

```
db:
    user: widgetuser
    password: pFmMxcyD;Fc6)6
    name: widget_production
    primary:
        host: rhodeisland.example.com
        port: 5432
    replica:
        host: virginia.example.com
        port: 5432

rabbitmq:
    host: pennsylvania.example.com
    port: 5672
```

If we choose YAML dictionaries, that changes the way we access the variables:

```
{{ db_primary_host }}
```

versus:

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

If you want to break things out even further, Ansible will allow you to define *group_vars/production* as a directory instead of a file, and let you place multiple YAML files that contain variable definitions.

For example, we could put the database-related variables in one file and the RabbitMQ-related variables in another file, as shown in Examples 3-9 and 3-10.

Example 3-9. group_vars/production/db

```
db:
    user: widgetuser
    password: pFmMxcyD;Fc6)6
    name: widget_production
    primary:
        host: rhodeisland.example.com
        port: 5432
    replica:
        host: virginia.example.com
        port: 5432
```

Example 3-10. group_vars/production/rabbitmq

```
rabbitmq:
    host: pennsylvania.example.com
    port: 6379
```

In general, I find it's better to keep things simple rather than split variables out across too many files.

Dynamic Inventory

Up until this point, we've been explicitly specifying all of our hosts in our hosts inventory file. However, you might have a system external to Ansible that keeps track of your hosts. For example, if your hosts run on Amazon EC2, then EC2 tracks information about your hosts for you, and you can retrieve this information through EC2's web interface, its Query API, or through command-line tools such as *awscli*. Other cloud providers have similar interfaces. Or, if you're managing your own servers and are using an automated provisioning system such as Cobbler or Ubuntu MAAS, then your provisioning system is already keeping track of your servers. Or, maybe you have one of those fancy configuration management databases (CMDBs) where all of this information lives.

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

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

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

```
$ chmod +x dynamic.py
```

The Interface for a Dynamic Inventory Script

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

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

Showing host details

To get the details of the individual host, Ansible will call the inventory script like this:

```
$ ./dynamic.py --host=vagrant2
```

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

```
{ "ansible_host": "127.0.0.1", "ansible_port": 2200,
  "ansible_user": "vagrant"}
```

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

Listing groups

Dynamic inventory scripts need to be able to list all of the groups, and details about the individual hosts. For example, if our script is called *dynamic.py*, Ansible will call it like this to get a list of all of the groups:

```
$ ./dynamic.py --list
```

The output should look something like this:

```
{"production": ["delaware.example.com", "georgia.example.com",
                "maryland.example.com", "newhampshire.example.com",
                "newjersey.example.com", "newyork.example.com",
                "northcarolina.example.com", "pennsylvania.example.com",
                "rhodeisland.example.com", "virginia.example.com"],
 "staging": ["ontario.example.com", "quebec.example.com"],
 "vagrant": ["vagrant1", "vagrant2", "vagrant3"],
 "lb": ["delaware.example.com"],
 "web": ["georgia.example.com", "newhampshire.example.com",
         "newjersey.example.com", "ontario.example.com", "vagrant1"]
 "task": ["newyork.example.com", "northcarolina.example.com",
          "ontario.example.com", "vagrant2"],
 "rabbitmq": ["pennsylvania.example.com", "quebec.example.com", "vagrant3"],
 "db": ["rhodeisland.example.com", "virginia.example.com", "vagrant3"]
}
```

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

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

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

```
"_meta" :
  { "hostvars" :
    "vagrant1" : { "ansible_host": "127.0.0.1", "ansible_port": 2222,
                   "ansible_user": "vagrant"},
    "vagrant2": { "ansible_host": "127.0.0.1", "ansible_port": 2200,
                   "ansible_user": "vagrant"},
    ...
}
```

Writing a Dynamic Inventory Script

One of the handy features of Vagrant is that you can see which machines are currently running using the `vagrant status` command. Assuming we had a Vagrant file that looked like Example 3-2, if we ran `vagrant status`, the output would look like Example 3-11.

Example 3-11. Output of vagrant status

```
$ vagrant status
Current machine states:

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

```
This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
```

Because Vagrant already keeps track of machines for us, there's no need for us to write a list of the Vagrant machines in an Ansible inventory file. Instead, we can write a dynamic inventory script that queries Vagrant about which machines are currently running.

Once we've set up a dynamic inventory script for Vagrant, even if we alter our Vagrantfile to run different numbers of Vagrant machines, we won't need to edit an Ansible inventory file.

Let's work through an example of creating a dynamic inventory script that retrieves the details about hosts from Vagrant.[2]

Our dynamic inventory script is going to need to invoke the `vagrant status` command. The output shown in Example 3-11 is designed for humans to read, rather than for machines to parse. We can get a list of running hosts in a format that is easier to parse with the `--machine-readable` flag, like so:

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

The output looks like this:

```
1474694768,vagrant1,metadata,provider,virtualbox
1474694768,vagrant2,metadata,provider,virtualbox
1474694768,vagrant3,metadata,provider,virtualbox
1410577818,vagrant1,state,running
1410577818,vagrant1,state-human-short,running
1410577818,vagrant1,state-human-long,The VM is running. To stop this VM%!(VAGRANT
_COMMA) you can run `vagrant halt` to\nshut it down forcefully%!(VAGRANT_COMMA)
or you can run `vagrant suspend` to simply\nsuspend the virtual machine. In
either case%!(VAGRANT_COMMA to restart it again%!(VAGRANT_COMMA)\nsimply run
`vagrant up`.
1410577818,vagrant2,state,running
1410577818,vagrant2,state-human-short,running
1410577818,vagrant2,state-human-long,The VM is running. To stop this VM%!(VAGRANT
_COMMA) you can run `vagrant halt` to\nshut it down forcefully%!(VAGRANT_COMMA)
or you can run `vagrant suspend` to simply\nsuspend the virtual machine. In
either case%!(VAGRANT_COMMA) to restart it again%!(VAGRANT_COMMA)\nsimply run
`vagrant up`.
1410577818,vagrant3,state,running
1410577818,vagrant3,state-human-short,running
1410577818,vagrant3,state-human-long,The VM is running. To stop this VM%!(VAGRANT
_COMMA) you can run `vagrant halt` to\nshut it down forcefully%!(VAGRANT_COMMA)
```

2 Yes, there's a Vagrant dynamic inventory script included with Ansible already, but it's helpful to go through the exercise.

```
or you can run `vagrant suspend` to simply\nsuspend the virtual machine. In
either case%!(VAGRANT_COMMA) to restart it again%!(VAGRANT_COMMA)\nsimply
run `vagrant up`.
```

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

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

The output looks like:

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

Our dynamic inventory script will need to call these commands, parse the outputs, and output the appropriate JSON. We can use the Paramiko library to parse the output of `vagrant ssh-config`. Here's an interactive Python session that shows how to use the Paramiko library to do this:

```
>>> import subprocess
>>> import paramiko
>>> cmd = "vagrant ssh-config vagrant2"
>>> p = subprocess.Popen(cmd.split(), stdout=subprocess.PIPE)
>>> config = paramiko.SSHConfig()
>>> config.parse(p.stdout)
>>> config.lookup("vagrant2")
{'identityfile': ['/Users/lorin/.vagrant.d/insecure_private_key'],
 'loglevel': 'FATAL', 'hostname': '127.0.0.1', 'passwordauthentication': 'no',
 'identitiesonly': 'yes', 'userknownhostsfile': '/dev/null', 'user': 'vagrant',
 'stricthostkeychecking': 'no', 'port': '2200'}
```

You will need to install the Python Paramiko library in order to use this script. You can do this with pip by running:

```
$ sudo pip install paramiko
```

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

Example 3-12. vagrant.py

```
#!/usr/bin/env python
# Adapted from Mark Mandel's implementation
# https://github.com/ansible/ansible/blob/stable-2.1/contrib/inventory/vagrant.py
# License: GNU General Public License, Version 3 <http://www.gnu.org/licenses/>
```

```
import argparse
import json
import paramiko
import subprocess
import sys


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


def list_running_hosts():
    cmd = "vagrant status --machine-readable"
    status = subprocess.check_output(cmd.split()).rstrip()
    hosts = []
    for line in status.split('\n'):
        (_, host, key, value) = line.split(',')[:4]
        if key == 'state' and value == 'running':
            hosts.append(host)
    return hosts


def get_host_details(host):
    cmd = "vagrant ssh-config {}".format(host)
    p = subprocess.Popen(cmd.split(), stdout=subprocess.PIPE)
    config = paramiko.SSHConfig()
    config.parse(p.stdout)
    c = config.lookup(host)
    return {'ansible_host': c['hostname'],
            'ansible_port': c['port'],
            'ansible_user': c['user'],
            'ansible_private_key_file': c['identityfile'][0]}


def main():
    args = parse_args()
    if args.list:
        hosts = list_running_hosts()
        json.dump({'vagrant': hosts}, sys.stdout)
    else:
        details = get_host_details(args.host)
        json.dump(details, sys.stdout)

if __name__ == '__main__':
    main()
```

Pre-Existing Inventory Scripts

Ansible ships with several dynamic inventory scripts that you can use. I can never figure out where my package manager installs these files, so I just grab the ones I need directly off GitHub. You can grab these by going to the Ansible GitHub repo and browsing to the *contrib/inventory* directory.

Many of these inventory scripts have an accompanying configuration file. In ???, we'll discuss the Amazon EC2 inventory script in more detail.

Breaking Out the Inventory into Multiple Files

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

For example, our directory structure could look like this: *inventory/hosts* and *inventory/vagrant.py*.

Our *ansible.cfg* file would contain these lines:

```
[defaults]
inventory = inventory
```

Adding Entries at Runtime with add_host and group_by

Ansible will let you add hosts and groups to the inventory during the execution of a playbook.

add_host

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

Why Do I Need add_host if I'm Using Dynamic Inventory?

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

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

We'll cover a cloud computing example that uses the *add_host* module in ???.

Invoking the module looks like this:

```
add_host name=hostname groups=web,staging myvar=myval
```

Specifying the list of groups and additional variables is optional.

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

```
- name: Provision a vagrant machine
  hosts: localhost
  vars:
    box: trusty64
  tasks:
    - name: create a Vagrantfile
      command: vagrant init {{ box }} creates=Vagrantfile

    - name: Bring up a vagrant machine
      command: vagrant up

    - name: add the vagrant machine to the inventory
      add_host: >
            name=vagrant
            ansible_host=127.0.0.1
            ansible_port=2222
            ansible_user=vagrant
            ansible_private_key_file=/Users/lorin/.vagrant.d/
            insecure_private_key

- name: Do something to the vagrant machine
  hosts: vagrant
  become: yes
  tasks:
    # The list of tasks would go here
    - ...
```

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

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

Note that we made use of the `creates=Vagrantfile` parameter in this task:

```
- name: create a Vagrantfile
  command: vagrant init {{ box }} creates=Vagrantfile
```

This tells Ansible that if the *Vagrantfile* file is present, the host is already in the correct state, and there is no need to run the command again. It's a way of achieving idempotence in a playbook that invokes the command module, by ensuring that the (potentially non-idempotent) command is run only once.

group_by

Ansible also allows you to create new groups during execution of a playbook, using the *group_by* module. This lets you create a group based on the value of a variable that has been set on each host, which Ansible refers to as a *fact*.[3]

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

Or, if we want to group our hosts by Linux distribution (e.g., Ubuntu, CentOS), we can use the *ansible_distribution* fact.

```
- name: create groups based on Linux distribution
  group_by: key={{ ansible_distribution }}
```

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

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

```
- name: group hosts by distribution
  hosts: myhosts
  gather_facts: True
```

3 We cover facts in more detail in ???.

```
  tasks:
    - name: create groups based on distro
      group_by: key={{ ansible_distribution }}

- name: do something to Ubuntu hosts
  hosts: Ubuntu
  tasks:
    - name: install htop
      apt: name=htop
    # ...

- name: do something else to CentOS hosts
  hosts: CentOS
  tasks:
    - name: install htop
      yum: name=htop
    # ...
```

Although using `group_by` is one way to achieve conditional behavior in Ansible, I've never found much use for it. In ???, we'll see an example of how to use the `when` task parameter to take different actions based on variables.

That about does it for Ansible's inventory. In the next chapter, we'll cover how to use variables. See ??? for more details about *ControlPersist*, also known as SSH multiplexing.

About the Authors

Lorin Hochstein was born and raised in Montreal, Quebec, though you'd never guess he was a Canadian by his accent, other than his occasional tendency to say "close the light." He is a recovering academic, having spent two years on the tenure track as an assistant professor of computer science and engineering at the University of Nebraska-Lincoln, and another four years as a computer scientist at the University of Southern California's Information Sciences Institute. He earned his BEng. in Computer Engineering at McGill University, his MS in Electrical Engineering at Boston University, and his PhD in Computer Science at the University of Maryland, College Park. He is currently a Senior Software Engineer at Netflix, where he works on the Chaos Engineering team.

René Moser is living in Switzerland with his wife and 3 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 since 15 years, recently 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.

Colophon

The animal on the cover of *Ansible: Up and Running* is a Holstein Friesian *(Bos primigenius)*, often shortened to Holstein in North America and Friesian in Europe. This breed of cattle originated in Europe in what is now the Netherlands, bred with the goal of obtaining animals that could exclusively eat grass—the area's most abundant resource—resulting in a high-producing, black-and-white dairy cow. Holstein-Friesians were introduced to the US from 1621 to 1664, but American breeders didn't become interested in the breed until the 1830s.

Holsteins are known for their large size, distinct black-and-white markings, and their high production of milk. The black and white coloring is a result of artificial selection by the breeders. Healthy calves weigh 90–100 pounds at birth; mature Holsteins can weigh up to 1280 pounds and stand at 58 inches tall. Heifers of this breed are typically bred by 13 to 15 months; their gestation period is nine and a half months.

This breed of cattle averages about 2022 gallons of milk per year; pedigree animals average 2146 gallons per year, and can produce up to 6898 gallons in a lifetime.

In September 2000, the Holstein became the center of controversy when one of its own, Hanoverhill Starbuck, was cloned from frozen fibroblast cells recovered one month before his death, birthing Starbuck II. The cloned calf was born 21 years and 5 months after the original Starbuck.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from Lydekker's *Royal Natural History, Vol. 2*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.