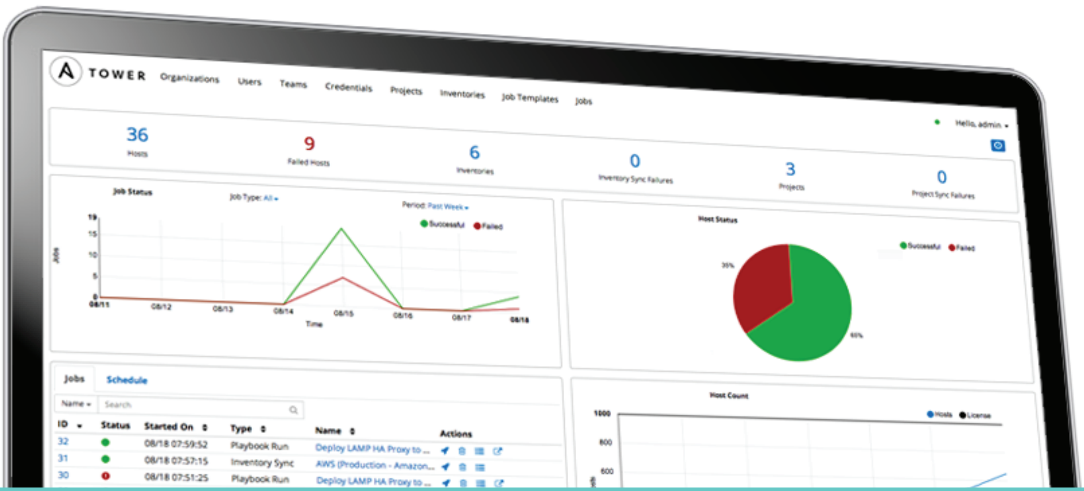
O'REILLY®

Ansible Up & Running

AUTOMATING CONFIGURATION MANAGEMENT AND DEPLOYMENT THE EASY WAY

Lorin Hochstein



ID	Status	Started On	Type	Name	Actions
32	🟢	08/18 07:59:52	Playbook Run	Deploy LAMP HA Proxy to ...	
31	🟢	08/18 07:57:15	Inventory Sync	AWS (Production - Amazon...	
30	🔴	08/18 07:51:25	Playbook Run	Deploy LAMP HA Proxy to ...	

"Ansible Tower has allowed us to provide better operations and security to our clients. It has also increased our efficiency as a team."

NASA uses Ansible Tower to centralize and control their Ansible automation initiative. With a real-time dashboard, role-based access control, credentials security, job scheduling, graphical inventory management and more, Ansible Tower is the best way to run Ansible in your organization, too.

Try Ansible Tower for free at ***ansible.com/tower***

ANSIBLE

Ansible: Up and Running

Lorin Hochstein

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Ansible: Up and Running

by Lorin Hochstein

Copyright © 2015 . All rights reserved.
Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Brian Anderson

Revision History for the :

See *http://oreilly.com/catalog/errata.csp?isbn=9781491915325* for release details.

This Preview Edition of Ansible: Up and Running, Chapters 1, 2, and 3, is a work in progress. The final book is currently scheduled for release in June 2015 and will be available at oreilly.com and other retailers once it is published.

ISBN: 978-1-491-91532-5

Table of Contents

Preface

Why I Wrote This Book

When I was writing my first web application, I remember the sense of accomplishment when the Django application I was writing was finally working on my desktop. I'd run **django manage.py runserver**, point my browser to *http://localhost:8000*, fill in data in the form boxes, and then everything happened as it should.

Then I discovered there were all of these… *things* I had to do, just to get the darned app to run on the Linux server. In addition to installing Django and my app onto the server, I had to install Apache and the mod_python module so that Apache could run Django apps. Then I had to figure out the right Apache configuration file incantation so that it would run my application and serve up the static assests properly.

None of it was *hard*, it was just a pain to get all of those details right. I didn't want to muck about with configuration files, I just wanted my app to run. Once I got it working, everything was fine, until eventually, several months later, I had to do it again, on a different server, and I had to start all over again.

Eventually, I discovered that this was Doing It Wrong. The Right Way to do this sort of thing has a name, and that name is *configuration management*. The great thing about using configuration management is that it's a mechanism of capturing knowledge that always stays up to date. No more hunting for the right doc page or searching through your old notes.

Recently, a colleague at work was interested in trying out Ansible for deploying a new project, and he asked me for a reference on how to apply the Ansible concepts in practice, beyond what was available in the official docs. I didn't know what else to recommend, so I decide to write something to fill the gap, and here it is. It comes too late for him, but hopefully you'll find it useful.

Who Should Read This Book

If you do systems administration or operations, this book is definitely for you. If you're a developer who is also responsible for doing deployments or configuration management, this book is for you, too.

Navigating This Book

I'm not a big fan of book outlines: Chapter 1 covers "so and so", Chapter 2 covers "such and such", that sort of thing. I strongly suspect that nobody ever reads them (I never do), and the table of contents is much easier to scan.

The book is written to be read start to finish, and later chapters build on the earlier ones. It's written largely tutorial style, so you should be able to follow along on your local machine. Most of the examples will be cloud and web focused.

Online Resources

The official Ansible docs are a useful reference: *http://docs.ansible.com*

The Ansible code is on GitHub, split across three repositories:

- *https://github.com/ansible/ansible*
- *https://github.com/ansible/ansible-modules-core*
- *https://github.com/ansible/ansible-modules-extras*

Bookmark the Ansible module index: you'll be referring to it constantly as you use Ansible: *http://docs.ansible.com/modules_by_category.html*

Ansible Galaxy is a repository of Ansible roles contributed by the community: *https://galaxy.ansible.com/*

The Ansible Project Google Group is the place to go if you have any questions about Ansible: *https://groups.google.com/forum/#!forum/ansible-project*

If you're interested in contributing to Ansible development, check out the Ansible Development Google Group: *https://groups.google.com/forum/#!topic/ansible-devel/68x4MzXePC4*

There's an active *#ansible* IRC channel on *irc.freenode.net*, if you're looking for real-time help.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/<catalog page>*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Thanks to John Jarvis at edX for reviewing an early version of the book. Thanks to Isaac Saldana and Mike Rowan at SendGrid for being so supportive of this endeavor. Thanks to Michael DeHaan for creating Ansible and shepherding the community that sprung up around it, as well as for providing feedback on the book, including an explanation of why he chose to use the name *Ansible*. Thanks to my editor, Brian Anderson, for his endless patience in working with me.

Thanks to Mom and Dad for their unfailing support; my brother Eric, the actual writer in the family; my two sons, Benjamin and Julian. Finally, thanks to my wife, Stacy, for everything.

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 onto 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. Not to mention the secondary services that we also need to deploy and maintain, like logging, montioring 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.

You can wire up these services by hand: spinning up the servers you need, ssh'ing 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 by hand, 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.

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

1. OK, nobody ever really delivered software like that.

A note about versions

As of this writing, the most recent Ansible release is Ansible 1.7.2, with 1.8 under active development. I'll specify any features that are 1.8-specific, but otherwise I'll assume you're running Ansible 1.7 or greater.

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.

These tools can also 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 like 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 like EC2, Azure, Digital Ocean, Google Compute Engine, Linode, 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 an example use case of Ansible in action. Alice is using Ansible to configure three Ubuntu-based web servers to run nginx. She has written an Ansible script called `webservers.yml`. In Ansible-speak, a script is called a *playbook*. A playbook describes which *hosts* (what Ansible calls remote servers) to configure, and a ordered list of *tasks* to perform on those hosts. In this example, the hosts are web1, web2, and web3, and the tasks are things like:

- install nginx
- generate an nginx configuration file
- copy over the SSL certificate
- start the nginx service

In the next chapter, we'll discuss what's actually in that playbook. Alice executes the playbook using the `ansible-playbook` command. In the example, the playbook is named `webservers.yml`, and is executed by doing:

```
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
```

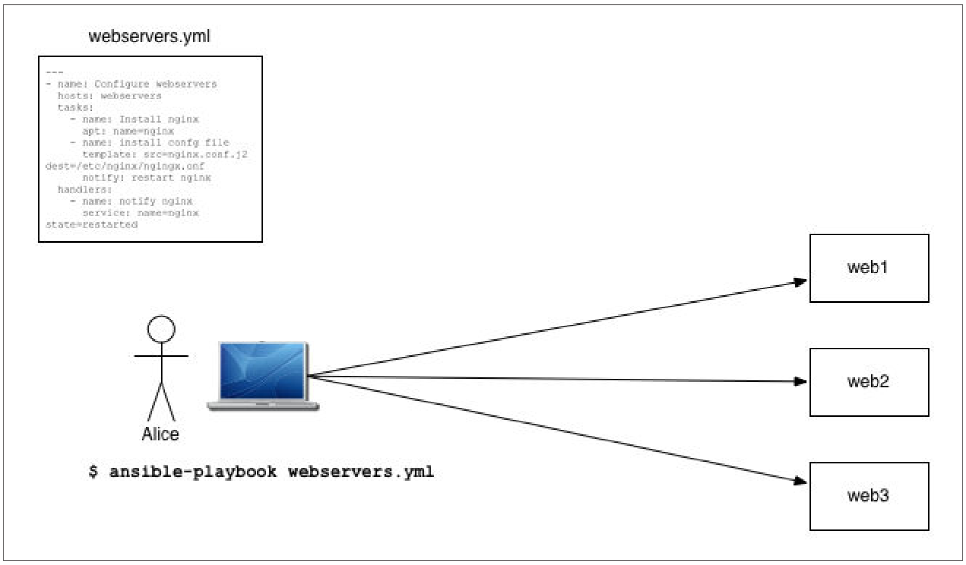
Ansible will:

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, web3
4. wait for the script to complete execution on all hosts

Ansible will then move to the next task in the list, and do the same thing. It's important to note that:

- Ansible runs in each task in parallel across all hosts.
- Ansible waits until all hosts have completed a task before moving to the next one.
- 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. Using Ansible to configuring 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.

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. Spoiler alert: Matt prefers Ansible.

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 for being 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 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 pre-install 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 greater installed.

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

Push-based

Some configuration management systems that use agents, like 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 server. This means that making configuration management changes to servers looks like:

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: Wake up after periodic timer fires
4. Agent on server: Connect to configuration management central service
5. Agent on server: Download new configuration management scripts
6. Agent on server: Execute configuration management scripts locally

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: Connect to servers, execute modules.

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 the timer to to elapse on the servers.

Ansible scales down

Yes, Ansible can be used to manage hundreds or even thousands of nodes. But what got me hooked is that it scales down well: using Ansible to configure a single node is easy, you write a single Ansible script file. 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 the remote servers you're managing, 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 may have a different (and likely unintended) effect.

Very thin layer of abstraction

Some configuration management tools provide a layer of abstraction over the specifics of the different operating systems running on the remote servers, 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.

While this may sound like a disadvantage in theory, in practice I've found that it makes Ansible easier to work with. Ansible doesn't require that I learn a new set of Ansible abstractions that hide the differences between operating systems. It makes Ansible's surface area smaller: there's less you need to know before you can start doing things.

If you really want to, you can write your Ansible playbooks to take different actions depending on the operating system of the remote server, and I cover how to do that in this book. But I try to avoid that when I can.

The unit of reuse in the Ansible community is the module. Because of 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.

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

In practice, though, every organization sets up their servers a little bit differently, and you're best off writing playbooks for your organization rather than trying to reuse generic playbooks. The primary value at 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 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 around Ansible, as well as a proprietary web-based management tool called Ansible Tower.

Is Ansible *too* simple?

When I was working on this book, my editor mentioned to me that "some folks who use <*another 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 about Ansible meeting your needs.

But don't worry. As you'll soon learn, Ansible provides a lot more functionality than shell scripts. As we 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.

If you're worried about the scalability of SSH: as we'll discuss in later chapters, Ansible uses SSH multiplexing to optimize performance, and there are folks out there who are managing thousands of nodes with Ansible[2].

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 for you that you otherwise wouldn't know how to do.

For the purposes of this book, I assume you have familiarity with at least one Linux distribution (e.g., Ubuntu, RHEL, CentOS), and you know how to:

- connect to a remote machine using ssh
- interact with the bash command-line shell (pipes and redirection)
- install packages
- use the *sudo* command
- check and set file permissions
- start and stop services
- set environment variables
- write scripts (any language)

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

I won't assume you have any knowledge of any particular programming language. In particular, you don't need to know Python to use Ansible at all, even for doing more advanced things like writing your own module are creating a dynamic inventory script.

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.

2. For example, see "Using Ansible at Scale to Manage a Public Cloud" by Jesse Keating of Rackspace: *http://www.slideshare.net/JesseKeating/ansiblefest-rax*

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, as well describing how to perform certain tasks that aren’t obvious from checking the official documentation.

I don’t cover all of 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 that you check out the official Jinja2 documentation.

In version 1.7, Ansible added support for managing Windows servers. I don’t cover the Windows support in this book, because I don’t have experience managing Windows servers with Ansible, and because I think this is still a niche use case. A proper treatment of using Ansible with Windows hosts probably deserves its own book.

I don’t discuss Ansible Tower, which is a commercial web-based tool for managing Ansible, developed by Ansible, Inc. This book focuses on Ansible itself, which is fully open-source, including all of the modules.

Installing Ansible

If you’re running on a Linux machine, all of the major distributions package Ansible these days, so you should be able to install it using your native package manager.

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 doing:

```
sudo pip install ansible
```

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

```
curl https://raw.githubusercontent.com/mitsuhiko/pipsi/master/get-pipsi.py | \
    python
pipsi install ansible
```

If you go the *pipsi* route, you’ll need to update your PATH environment variable to include *\~/.local/bin*.

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

```
git clone https://github.com/ansible/ansible.git
```

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: *http://docs.ansible.com/intro_installation.html*
- pip: *http://pip.readthedocs.org/*
- pipsi: *https://github.com/mitsuhiko/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 (yes, they support Linux servers), Digital Ocean, Rackspace, SoftLayer, HP Public Cloud, Linode… you get the idea.

Using Vagrant to set up a test server

If you'd prefer not to spend the money running in the 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 a later chapter. For now, we'll just manage a Vagrant virtual machine as if it was a regular Linux server.

Vagrant needs the VirtualBox virtualizer to be installed on your machine. Download VirtualBox at *http://www.virtualbox.org* and then download Vagrant at *http://www.vagrantup.com*.

I recommend you create a directory for your Ansible playbooks and related files, I've called it `playbooks` below.

Run the following commands to create a Vagrant configuration file (Vagrantfile) for an Ubuntu 14.04 (Trusty Tahr) 64-bit virtual machine image, 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 may take a while depending on your internet connection.

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

```
A `Vagrantfile` has been placed in this directory. You are now
ready to `vagrant up` your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
`vagrantup.com` for more information on using Vagrant.

Bringing machine 'default' up with 'virtualbox' provider...
==> default: Box 'ubuntu/trusty64' could not be found. Attempting to find and install...
    default: Box Provider: virtualbox
    default: Box Version: >= 0
==> default: Loading metadata for box 'ubuntu/trusty64'
    default: URL: https://vagrantcloud.com/ubuntu/trusty64
==> default: Adding box 'ubuntu/trusty64' (v14.04) for provider: virtualbox
    default: Downloading: https://vagrantcloud.com/ubuntu/trusty64/version/1/provider/
             virtualbox.box
==> default: Successfully added box 'ubuntu/trusty64' (v14.04) for 'virtualbox'!
==> 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: vm_default_1409457679518_47647
==> 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 => 2222 (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: Connection timeout. Retrying...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Mounting shared folders...
    default: /vagrant => /Users/lorinhochstein/playbooks
```

You should be able to ssh into your new Ubuntu 14.04 virtual machine by doing:

```
vagrant ssh
```

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

```
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-35-generic x86_64)

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

  System information as of Sun Aug 31 04:07:21 UTC 2014

  System load:  0.0               Processes:           73
  Usage of /:   2.7% 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.


Last login: Sun Aug 31 04:07:21 2014 from 10.0.2.2
```

Type `exit` to quit the ssh session.

This approach works for us interacting 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 doing:

```
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/lorinhochstein/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL
```

The important lines are:

```
  HostName 127.0.0.1
  User vagrant
  Port 2222
  IdentityFile /Users/lorinhochstein/.vagrant.d/insecure_private_key
```

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

Confirm 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/lorinhochstein/.vagrant.d/insecure_private_key
```

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

Telling Ansible about your test server

Ansible can only manage servers that 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 an alias: `testserver`.

Create a file called *inventory* in the *playbooks* directory, this will serve as the inventory file.

If you're using a Vagrant virtual machine as your test server, your inventory file should look like Example 1-1, except that you should replace */Users/lorinhochstein* with your own home directory. I've broken the file contents up across multiple lines so it fits on the page, but it should be all on one line in your file.

Example 1-1. playbooks/inventory

```
testserver ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
ansible_ssh_user=vagrant
ansible_ssh_private_key_file=/Users/lorinhochstein/.vagrant.d/insecure_private_key
```

Here we see one of the drawbacks of using Vagrant. We have to explicitly pass in a bunch of 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_ssh_host=ec2-203-0-113-120.compute-1.amazonaws.com
ansible_ssh_user=ubuntu ansible_ssh_private_key_file=/path/to/keyfile.pem
```

We'll use the `ansible` command-line tool to verify 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 *inventory* and invoke the `ping` module:

```
ansible testserver -i inventory -m ping
```

If it succeeded, output will look like this:

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

We can see that the module succeeded. The `"changed": false` part of the outputtells us that executing the module did not change the state of the server. The `"ping": "pong"` output is just module-specific output.

The ping module doesn't actually do anything other than check that Ansible can start an ssh session with the servers. It's mostly useful for testing that you 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.

Listing Example 1-2 shows an *ansible.cfg* file that specifies the location of the inventory file (hostfile), the user to ssh as (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.

Example 1-2. ansible.cfg

```
[defaults]
hostfile = inventory
remote_user = vagrant
private_key_file =/Users/lorinhochstein/.vagrant.d/insecure_private_key
```

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.

While 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 *inventory* file. Instead, it simplifies to:

```
testserver ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
```

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

```
ansible testserver -m ping
```

I like to use `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 do:

```
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 just do this instead:

```
ansible testserver -a uptime
```

If our command contains spaces, we need to quote it so 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:

```
testserver | success | rc=0 >>
[    5.170544] type=1400 audit(1409500641.335:9): apparmor="STATUS" operation="pro
ile_replace" profile="unconfined" name="/usr/lib/NetworkManager/nm-dhcp-client.act
on" pid=888 comm="apparmor_parser"
[    5.170547] type=1400 audit(1409500641.335:10): apparmor="STATUS" operation="pr
file_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 `-s` flag to tell Ansible to `sudo` as root. For example, to access */var/log/syslog* requires root access:

```
ansible testserver -s -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 by doing:

```
ansible testserver -s -m apt -a name=nginx
```

If installing nginx fails for you, you may 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 -s -m service -a "name=nginx state=restarted"
```

We need the `-s` argument to use sudo since only root can install the nginx package.

Moving forward

To recap, in this introductory chapter we've covered the basic concepts of Ansible at 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, which is 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 SSL support.

If you're following along in this chapter, you should end up with the files listed in Example 2-1

Example 2-1. Files

```
playbooks/Vagrantfile
playbooks/web-nossl.yml
playbooks/files/index.html
playbooks/files/nginx.key
playbooks/files/nginx.crt
playbooks/files/nginx.conf
playbooks/templates/nginx.conf.j2
```

Some preliminaries

Before we can actually run this playbook against our Vagrant instance, we need to expose ports 80 and 443 so we can access them. 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. This will allow you to access the web server running inside Vagrant at *http://localhost:8080* and *https://localhost:8443*

Once you make the changes, tell Vagrant to have them go into effect by doing:

```
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)
```

We also need to manually generate an SSL certificate. In a production environment, you'd purchase your SSL certificate from a certificate authority. We're just going to generate a self-signed certificate.

Create a `files` subdirectory of your `playbook` directory, then generate the SSL certificate and key:

```
mkdir files
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
    -keyout files/nginx.key -out files/nginx.crt
```

It will ask you some questions. The only one that really matters is:

```
Common Name (e.g. server FQDN or YOUR name) []
```

Answer `localhost`.

It should generate the files *nginx.key* and *nginx.crt* in the *files* directory.

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 SSL[1] encryption. This will make setting up the web server simpler, but any real website really should have SSL encryption enabled, and we'll cover how to do that later on in this chapter.

Example 2-2. web-nossl.yml

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

1. Technically the protocol is called TLS, but for historical purposes many people use the term SSL, which refers to an older protocol

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

First, we'll see what happens when we run this playbook, then we'll go over the contents of the playbook in detail.

Why do you use "True" in one place and "yes" in another?

Ansible is pretty flexible on how you represent truthy and falsy values in playbooks. Strictly speaking, module arguments (like `name=yes`) are treated differently from values elsewhere in playbooks. That's because values elsewhere are handled by the YAML parser and so use the YAML convention 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 offiical 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 that with our own configuration file as part of this playbook. As we'll see later, we'll need to modify this configuration file to support SSL. Here is a basic Nginx config file. Put it in *playbooks/files/nginx.conf*.

Example 2-3. 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 called `templates`. I'll follow this convention throughout the book.

Creating a custom home page

Let's add a custom homepage. Put it in `playbooks/files/index.html`. Here's an example:

```
<html>
  <head>
    <title>Welcome to Nginx, via Ansible!</title>
  </head>
  <body>
  <h1>Nginx, configured by Ansible</h1>
  <p>If you can see this, Ansible successfully installed Nginx.</p>
  </body>
</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 INI file format. We'll go into this format in detail later in the book. Edit your `playbooks/inventory` file to put a `[webservers]` line above the `testserver` line. This indicates that `testserver` is in the `webservers` group.

Example 2-4. playbooks/inventory

```
[webservers]
testserver ansible_ssh_host=127.0.0.1 ansible_ssh_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-nossl.yml
```

The output should look like this:

Example 2-5. Output of ansible-playbook

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

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
```

If you didn't get any errors, you should be able to point your browser to *http://localhost:8080* and see the custom HTML page.

What's this "gathering facts" business?

You may have noticed the following lines of output when Ansible first starts to run:

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

When Ansible starts executing a play, the first thing it does is collect information about the server it is connecting to, including which operating system is running, hostname, IP and MAC addresses of all interfaces, and so on.

You can then use this information later on in the playbook. For example, you may need the IP address of the machine for populating a configuration file.

You can also turn off fact gathering if you don't need it, in order to save some time. We'll cover the use of facts and how to disable them in a later chapter.

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, it's instructive to understand the concepts of YAML that are most important for writing playbooks.

Start of file

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

```
---
```

Comments

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

- 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`

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

For example, this 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, it 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, it looks like this:

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

Line folding

When writing playbooks, you'll often encounter situations where passing a lot of arguments to a module, and you want to break this up across multiple lines in your file, but you want Ansible to treat the string as if it was 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 this playbook from the perspective of a YAML file. Here it is again:

Example 2-6. web-nossl.yml

```
---
- name: Configure webserver with nginx
  hosts: webservers
  sudo: 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
      copy: src=files/index.html dest=/usr/share/nginx/html/index.html mode=0644
    - name: restart nginx
      service: name=nginx state=restarted
```

The JSON equivalent of this file is:

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

```
[
  {
    "name": "Configure webserver with nginx and ssl",
    "hosts": "webservers",
    "sudo": 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\n"
      },
      {
        "name": "copy index.html",
        "copy": "src=files/index.html 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 from our example:

```
name: Configure webserver with nginx and ssl
hosts: webservers
sudo: 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
    copy: src=files/index.html 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.

`sudo`
: If true, Ansible will run every task by sudo'ing as (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 with 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. It's useful as a comment for the intent of the task (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 a later chapter, you can use the **--start-at-task <task name>** flag to tell **ansible-playbook** to start a playbook in the middle of a task, 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 example above, 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 front-end, 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, like the `ec2` module. We'll cover that in a later chapter.

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

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

Modules

Modules are scripts[2] that come packaged with Ansible that 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 restart a service.
- *template* generates a file from a template and copies it to the hosts.

We didn't use the *template* module in this playbook, but we'll be using it in the next one.

Recall from the first chapter that Ansible executes a host on a task by generating a custom script based on the module name and arguments, then copies this script to the host and runs it.

As of this writing, there are over two hundred modules that ship with Ansible, and this number grows with every point 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 2-1 is an entity-relationship diagram that depicts this relationship between *play-books*, *plays*, *hosts*, *tasks*, and *modules*.

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

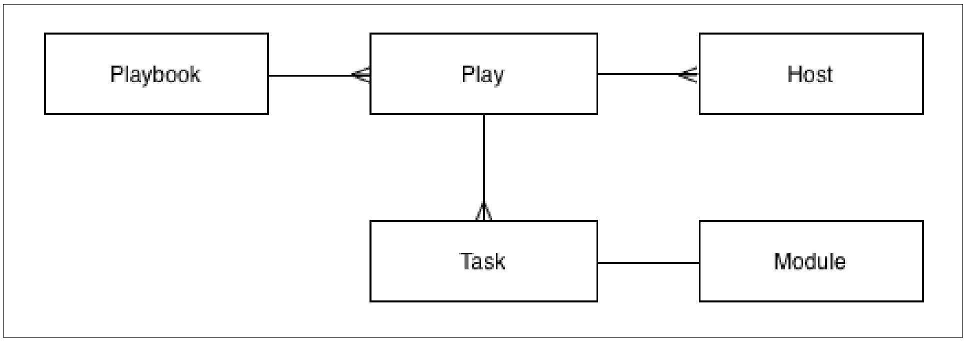


Figure 2-1. Entity-relationship diagram

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 in my terminal.

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

The *enable configuration*, on the other hand, has status *ok*, which appears as green in 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*.

Looking back at the two examples above, 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 the 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 on 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: SSL support

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

- variables
- templates
- handlers

Here's what our playbook looks like with SSL support.

Example 2-8. web-ssl.yml

```
---
- name: Configure webserver with nginx and ssl
  hosts: webservers
  sudo: 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
    - name: create directories for ssl certificates
      file: path=/etc/nginx/ssl state=directory
    - name: copy SSL key
      copy: src=files/nginx.key dest={{ key_file }} owner=root mode=0600
      notify: restart nginx
    - name: copy SSL 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
  handlers:
    - name: restart nginx
      service: name=nginx state=restarted
```

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 here, 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 SSL 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.

The YAML parser will misinterpret a variable reference as the beginning of an in-line dictionary, if it's used immediately after the module is specified. For 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, we 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 we 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 then `msg` argument. Ansible supports alternating single and double quotes, so this works:

```
- 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:

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, especially if, say, four different program files need the same bit of configuration data (say, the IP address of your server, or the database credentials), and this information changes and you have to go and edit those files. It's much better to take the info that's specific to your deployment, write it down in exactly 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 like Mustache or the Django template system, Jinja2 will feel very familiar.

Nginx's configuration file needs information about where to find the SSL 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*:

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 SSL certificate

`key_file`
: the path to the SSL 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 version 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.
>
> This is one of the only changes that have occurred to Ansible syntax since its release. Generally speaking, the language has been remarkably stable because backwards compatibility is one of the main goals of the Ansible project.

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 at *http://jinja.pocoo.org/docs/dev/templates/* 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-ssl.yml*, 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 SSL 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 only runs 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 example above, the handler's name is `restart nginx`. For an nginx server, we'd need to restart it if any of the following happens:

- the SSL key changes
- the SSL certificate changes
- the configuration file changes
- the configuration's enabled status changes

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 only run after all of the tasks are run, and they only run once, even if they are notified multiple times. They always run in the order that they appear in the play, not the notification order.

The official Ansible docs mention that the only common use cases 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:

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

A workaround for this scenario is to pass the `--force-handlers` flag to `ansible-playbook` the second time you invoke it, so that Ansible will run the handlers unconditionally.

Running the playbook

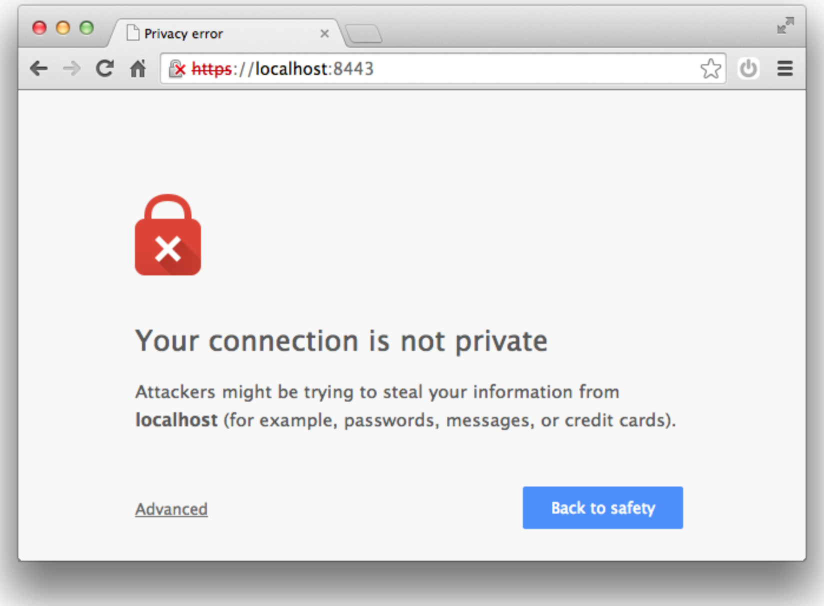
As before, we use the `ansible-playbook` command to run the playbook.

```
ansible-playbook web-ssl.yml
```

The output should look something like this:

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

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

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

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

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

TASK: [copy SSL 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 horrible error that says something like "Your connection is not private".


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 2-2. Chrome does not want you to visit this page

That error is expected. We generated a self-signed SSL certificate, and web browsers like Chrome only trust certificates that have been issued from a proper authority.

Enabling cowsay

No text about Ansible would be complete without describing cowsay support.

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

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

You can specify an alternate cow by setting the ANSIBLE_COW_SELECTION environment variable like this:

```
export ANSIBLE_COW_SELECTION=tux
```

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

```
cowsay -l
```

If, for some reason, 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 (which talk more about this file in later chapters):

```
[defaults]

nocows = 1
```

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", 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 them). 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, for example:

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 plugin instead of the default ssh plugin.

There is one host that Ansible automatically adds to the inventory by default, *localhost*. Even if your inventory file is empty, you can still connect to localhost. Ansible understands that *localhost* refers to your local machine, and so it will interact with it directly rather than ssh. You can even pass Ansible an empty file as an inventory file, and it will still be able to interact with localhost.

Example 3-2. Ansible knows localhost, even with an empty inventory file

```
ansible -i /dev/null localhost -m ping
```

Preliminaries: multiple Vagrant VMs

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 doing:

```
vagrant destroy --force
```

If you don't do 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 this:

Example 3-3. Vagrantfile with three servers

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  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
```

For now, we'll assume each of these servers can potentially be a web server, so we'll map ports 80 and 443 inside each of them to a port on the local machine.

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

```
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/lorinhochstein/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 doing:

```
vagrant ssh-config
```

The output should look something like:

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

We can see the *vagrant1* uses port 2222, *vagrant2* uses port 2200, and *vagrant3* uses port 2201.

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

```
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201
```

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

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

On my machine, the output looks like:

```
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.

Table 3-1. Behavioral parameters

Name	Default	Description
ansible_ssh_host	name of host	hostname to ssh to
ansible_ssh_port	22	port to ssh to
ansible_ssh_user	root	user to ssh as
ansible_ssh_pass	*none*	password to use for ssh authentication
ansible_connection	smart	how Ansible will connect to host (see below)
ansible_ssh_private_key_file	*none*	ssh private key to use for ssh authentication

Name	Default	Description
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 use the OpenSSH's *ssh* program as the SSH client if it is installed on the local machine that Ansible is running on if the SSH client supports a feature called *ControlPersist*. 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*.

Connections are an advanced topic which we'll discuss in a later chapter. It's unlikely that you'll need to change this for most use cases.

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 paramters 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

All of 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 may 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*, since Arch Linux installs Python 3 at */usr/bin/python*, and Ansible modules are not 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 a later chapter on custom modules.

Changing behavioral parameter defaults

You can override some of the behavioral parameter default values in the `[defaults]` section of the *ansible.cfg* file. 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_ssh_port	remote_port
ansible_ssh_user	remote_user
ansible_ssh_private_key_file	private_key_file
ansible_shell_type	executable (see below)

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

We generally want to perform actions on groups of hosts, rather than on indvidual 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 doing:

```
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 .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_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_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_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201
virginia.example.com

[vagrant]
vagrant1
vagrant2
vagrant3
```

Example: deploying a Rails app

Imagine you're responsible for deploying a Rails-based web application. The app needs to support the following services:

- The actual Rails web app itself, run by a *Unicorn* HTTP server.
- An Nginx web server, which will sit in front of Unicorn and serve static assets
- A Sidekiq task queue which will execute long-running jobs on behalf of the web app
- A Redis data store which serves as the back-end for Sidekiq

- A Postgres database that serves as the persistent store

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 be perform quickly and be reliable, so we do things like:

- 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 Unicorn, Sidekiq, Redis, 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 Redis server and two database servers, that's ten 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 only use two hosts for staging: we'll put the web server and task queue on one staging host, and Redis 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 Redis and Postgres.

Here's 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 Rails 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_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_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

[redis]
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 parameters for the vagrant instances once.

Aliases and ports

We described our vagrant hosts like this:

```
[vagrant]
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_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 do this:

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 only associate a single host with *127.0.0.1*, so the *vagrant* group would only contain 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 Rails and its dependencies. We might fined it useful to define a "rails" group that contains both of these two groups.You would add this to the inventory file:

```
[rails: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 *tasks* as groups and not as hosts.

Numbered hosts (pets vs. cattle)

The inventory file shown in Example 3-4 looks complex. It describes fifteen different hosts. That doesn't sound like a large number in this cloudy scale-out world, but even dealing with fifteen hosts in the inventory file can cumbersone because each host has a completely different hostname.

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

The cattle approach is much more scalable, and Ansible supports it well by allowing you supporting numeric patterns.

1. This term has been popularized by Randy Bias of Cloudscaling, *http://www.slideshare.net/randybias/pets-vs-cattle-the-elastic-cloud-story*.

For example, if your servers were named *web1.example.com*, *web2.example.com*, …, *web20.example.com* 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. So, if you wanted to do use the convention *web-a.example.com*, *web-b.example.com*, …, *web-t.example.com*, then you can do this:

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

Hosts and group variables, inside the inventory

Recall how we specified behavioral inventory parameters for Vagrant hosts:

```
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_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 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 playbook.

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

Circling back to our Rails example: the web application and task queue service need to communicate with Redis 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 actually reach the database) as well as by username and password, where Redis is secured only by the network layer.

To set everything up, we need to:

- Configure the web servers with the hostname, port, username, password of the primary postgres server, as well as the name of the database

- Configure the task queues with the hostname, port, username, password of the primary postgres server, as well as the name of the database
- Configure the web servers with the hostname and port of the Redis server
- Configure the task queueservers with the hostname and port of the Redis server
- Configure the primary postgres server with the hostname, port, 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.

Here's 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
redis_host=pennsylvania.example.com
redis_port=6379


[staging:vars]
db_primary_host=quebec.example.com
db_name=widget_staging
db_user=widgetuser
db_password=L@4Ryz8cRUXedj
redis_host=quebec.example.com
redis_port=6379

[vagrant:vars]
db_primary_host=vagrant3
db_primary_port=5432
db_primary_port=5432
db_name=widget_vagrant
db_user=widgetuser
db_password=password
redis_host=vagrant3
redis_port=6379
```

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 them this way.

Additionally, while Ansible variables can hold booleans, strings, lists, and dictionaries, you can only specify booleans and strings.

Ansible offers a more scalable approach to keeping track of host and group variables. You can create a separate variable file for each host and each group. These files are 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/inventory*, 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*.

Here's an example 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_replica_host: virginia.example.com
db_name: widget_production
db_user: widgetuser
db_password: pFmMxcyD;Fc6)6
redis_host:pennsylvania.example.com
```

Note that we could also use YAML dictionaries to represent these values.

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
```

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

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

```
    {{ db_primary_host }}
```

vs.

```
    {{ 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 allow you to place multiple YAML files that contain variable definitions.

For example, we could put the database-related variables in one file, and the redis-related variables in another fiel, like this:

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/redis

```
---
redis:
    host: pennsylvania.example.com
    port: 6379
```

Adding entries at runtime with add_host and group_by

Ansible will let you add hosts and groups to the inventory during the execution of a playbook. This is useful if you are using Ansible to, say, spin up a new virtual machine instance inside an infrastructure-as-a-service cloud.

add_host

The *add_host* module adds a host to the inventory. For example, if our Vagrantfile only describes one machine, we could bring it up and add it to the inventory.

Invoking it looks like this:

```
add_host: name=hostname groups=foo,bar,baz myvar=myval
```

Specifying the list of groups and additional variables is optional.

Here's the *add_host* command in action:

```
---
- 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 server
      command: vagrant up
    - name: add the Vagrant hosts to the inventory
      add_host: >
            name=vagrant
            ansible_ssh_host=127.0.0.1
            ansible_ssh_port=2222
            ansible_ssh_user=vagrant
            ansible_ssh_private_key_file=/Users/lorinhochstein/.vagrant.d/insecure_private_key

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

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 already 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 with the command module.

group_by

Example 3-11. Creating ad-hoc groups based on Linux distribution

```
---
- name: group hosts by distribution
  hosts: myhosts
  gather_facts: True
  tasks:
    - name: create groups based on distro
      group_by: key={{ ansible_distribution }}

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

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

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

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 or 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 the example apt the top of this section, 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.

While using *group_by* is one way to achieve conditional behavior in Ansible, I've never found much use for it. In the next chapter, we'll see how to use the *when* task parameter to do things like take different actions based on the distribution. We'll also discuss facts in more detail then.

Dynamic inventory

In many cases, you are likely to 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 their web interface, their Query API, or through command-line tools like *awscli*. Other cloud providers have similar interfaces. Or, if you're managing your own servers and are using an automated provisioning system like 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.

Ansible supports a feature called *dynamic inventory*. 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

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"],
 "redis": ["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.

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_ssh_host": "127.0.0.1", "ansible_ssh_port": 2200,
  "ansible_ssh_user": "vagrant"}
```

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

Writing a dynamic inventory script

As an example, let's write a dynamic inventory script for Vagrant. Yes, there's one included with Ansible already, but it's helpful to go through the exercise.

First, we need to figure out how to get the information from Vagrant about which machines are running and how to connect to them. We can get a list of running hosts in a format that is easy to parse by doing:

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

The output looks like:

```
1410577818,vagrant1,provider-name,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,provider-name,virtualbox
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,provider-name,virtualbox
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) 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 do:

```
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/lorinhochstein/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL
```

Our dynamic inventory script will need to call these command, 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/lorinhochstein/.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 doing:

```
sudo pip install paramiko
```

Here's 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/devel/plugins/inventory/vagrant.py
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(',')
        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_ssh_host': c['hostname'],
            'ansible_ssh_port': c['port'],
            'ansible_ssh_user': c['user'],
            'ansible_ssh_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 of GitHub. You can grab these by going to the Ansible GitHub repo at *https://github.com/ansible/ansible* and browsing to the *plugins/inventory* directory.

Many of these inventory scripts have an accompanying configuration file. In a later chapter, 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 (either via the *hostfile* parameter in *ansible.cfg* or using the `-i` flag on the command-line). Ansible will process all of the files and will merge the results into a single inventory.

For example, our directory structure could look like this:

```
inventory/hosts
inventory/vagrant.py
```

and our *ansible.cfg* file would contain the lines:

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

This covers everything about Ansible's inventory. In the next chapter, we'll revisit playbooks and discuss how to implement more complex behaviors.