Ansible for DevOps

Server and configuration management for humans

Jeff Geerling

Ansible for DevOps

Server and configuration management for humans

Jeff Geerling

This book is for sale at http://leanpub.com/ansible-for-devops

This version was published on 2014-09-30

Tweet This Book!

Please help Jeff Geerling by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just purchased @Ansible4DevOps by @geerlingguy on @leanpub - https://leanpub.com/ansible-for-devops #ansible

The suggested hashtag for this book is #ansible.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#ansible

```
 __________________________
< To my wife and children. >
 --------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Contents

Preface

I am fortunate to have a radio engineer for a father. His radio stations' networks and IT infrastructure (everything from Novell servers and old Mac and Windows workstations in the 90s, to Microsoft and Linux-based servers and everything in-between) were maintained by the engineering staff, and my Dad showed me how it all worked. Even better, he brought home old decommissioned servers and copies of Linux he had burned to CD for me!

I was able to start working with Linux and small-scale infrastructures before I started high school (even building a Cat5 wired network and small rack of networking equipment for a local grade school!), and my passion for managing infrastructure grew. When I started developing full-time, what was once a hobby became a necessary part of my job, so I invested more time in managing infrastructure efficiently. Over the past ten years, I've gone from manually booting and configuring physical and virtual servers, to using relatively complex shell scripts to provision and configure servers, to using configuration management tools to manage many cloud-based servers.

When I began converting my infrastructure to code, some of the best tools for testing, provisioning, and managing my servers were still in their infancy, but they have since matured into fully-featured, robust tools I use every day. Vagrant is an excellent tool for managing local virtual machines to mimic real-world infrastructure locally (or in the cloud), and Ansible (the subject of this book) is an excellent tool for provisioning servers, managing their configuration, and deploying applications—even on my local workstation!

These tools are still improving rapidly, and I'm excited for what the future holds. New tools (like Docker) that are nearing production-ready status also excite me, and I know the time I invest in learning to use these tools well will be helpful for years to come (Ansible, Docker, and Vagrant seem a potent combination for both local and production infrastructure… but that's a little outside of *this* book's scope).

In the following pages, I will share with you all I've learned about Ansible—my favorite tool for server provisioning, configuration management, and application deployment. I hope you enjoy reading this book as much as I did writing it!

– Jeff Geerling, 2014

Who is this book for?

Many of the developers and sysadmins I work with are at least moderately comfortable administering a Linux server via SSH, and manage between one and one hundred servers.

Some of these people have a little experience with configuration management tools (usually with Puppet or Chef), and maybe a little experience with deployments and continuous integration using

tools like Jenkins, Capistrano, or Fabric. I am writing this book for these friends (who, I think, are representative of most people who have heard of and/or are beginning to use Ansible).

If you are interested in both development and operations, and have at least a passing familiarity with managing a server via the command line, you should end up with an intermediate to expert-level understanding of Ansible and how you can use it to manage your infrastructure after reading this book.

Typographic conventions

Ansible uses a simple syntax (YAML) and simple command-line tools (using common POSIX conventions) for all its powerful abilities. Code samples and commands will be highlighted throughout the book either inline (for example: `ansible [command]`), or in a code block (with or without line numbers) like:

```
---
# This is the beginning of a YAML file.
```

Some lines of YAML and other code examples require more than 80 characters per line, resulting in the code wrapping to a new line. Wrapping code is indicated by a colored `\` at the end of the line of code. For example:

```
# The line of code wraps due to the extremely long URL.
wget http://www.example.com/really/really/really/long/path/in/the/url/causes/the\
/line/to/wrap
```

When using the code, don't copy the `\` character, and make sure you don't use a newline between the first line with the trailing `\` and the next line.

Links to pertinent resources and websites are added inline, like the following link to Ansible[1], and can be viewed directly by clicking on them in eBook formats, or by following the URL in the footnotes.

Sometimes, asides are added to highlight further information about a specific topic:

Informational asides will provide extra information.

Warning asides will warn about common pitfalls and how they can be avoided.

[1] http://www.ansible.com/

Tip asides will give tips for deepening your understanding or optimizing your use of Ansible.

When displaying commands run in a terminal session, if the commands are run under your normal/non-root user account, the commands will be prefixed by the dollar sign ($). If the commands are run as the root user, they will be prefixed with the pound sign (#).

Please help improve this book!

This book is a work in progress, and is being expanded and updated on LeanPub at a rather rapid clip. If you think a particular section needs improvement, or find something missing, please contact me via Twitter (@geerlingguy[2]), Google+[3], a comment on this book's Feedback page on LeanPub[4], or whatever method is convenient for you.

Please note that, since the book is still being written and proofread, the book contains certain imperfections and oddities. I've tried to ensure every line of code, at least, works perfectly, but I may have an occasional typo—you've been warned!

About the Author

Jeff Geerling[5] is a developer who has worked in programming and devops for companies with anywhere between one to thousands of servers. He also manages many virtual servers for services offered by Midwestern Mac, LLC[6], and has been using Ansible to manage infrastructure since early 2013.

[2] https://twitter.com/geerlingguy
[3] https://plus.google.com/+JeffGeerling
[4] https://leanpub.com/ansible-for-devops/feedback
[5] http://jeffgeerling.com/
[6] http://www.midwesternmac.com/

Introduction

In the beginning, there were sysadmins

Deploying and managing servers reliably and efficiently has been a challenge since the beginning of networked computing. Historically, system administrators—walled off organizationally from the developers and users who interacted with the systems they administered—have managed servers by hand, installing software, changing configurations, and administering services on individual servers.

As data centers grew, and as the applications they hosted became more complex, administrators realized they couldn't scale their manual systems management as fast as the applications they were enabling. Thus server provisioning and configuration management tools began to flourish.

Server virtualization brought large-scale infrastructure management to the fore, and the number of servers managed by one admin, or a small team of admins, has grown by an order of magnitude. Instead of deploying, patching, and destroying every server by hand, admins are expected to be able to bring up new servers either automatically, or with minimal intervention. Many large-scale IT deployments involve hundreds or thousands of servers, and in many of the largest environments, server provisioning, configuration, and decommissioning are entirely automated.

Modern infrastructure management

As the systems that run applications become an ever more complex and integral part of the software they run, application developers themselves have begun to integrate their work more fully with operations personnel, and in many companies, development and operations work is almost fully integrated. Indeed, this can be a requirement for modern test-driven application design, as well as continuous integration.

As a software developer by trade, and a sysadmin by necessity, I have seen the power in uniting development and operations—more commonly referred to nowadays as DevOps. When developers begin to think of infrastructure as *part of their application*, stability and performance become normative. When sysadmins (most of whom have intermediate to advanced knowledge of the applications and languages being used on servers they manage) work tightly with developers, development velocity is improved, and more time can be spent doing 'fun' activities like performance tuning, experimentation, and getting things done (and less time is spent putting out fires).

DevOps is a loaded word; some people argue using the word to identify both the *movement* of development and operations working more closely to automate infrastructure-related processes, and the *personnel* who skew slightly more towards the system administration side of the equation, dilutes the word's meaning. I think the word has simply come to be a rallying cry for the employees who are dragging their startups, small businesses, and enterprises into a new era of infrastructure growth and stability. I'm not too concerned that the term has become more of a catch-all for modern infrastructure management. Spend less time arguing over the definition of the word, and more time making it mean something *to you*.

Ansible and Ansible, Inc.

Ansible was released in 2012 by Michael DeHaan (@laserllama[7] on Twitter), a developer who has been working with configuration management and infrastructure orchestration in one form or another for many years. Through his work with Puppet Labs and RedHat (where he worked on Cobbler[8], a configuration management tool and Func[9], a tool for communicating commands to remote servers), and some other projects[10], he experienced the trials and tribulations of many different organizations and individual sysadmins on their quest to simplify and automate their infrastructure management operations.

Additionally, Michael found many shops were using separate tools[11] for configuration management (Puppet, Chef, cfengine), server deployment (Capistrano, Fabric), and ad-hoc task execution (Func, plain SSH), and wanted to see if there was a better way. Ansible wraps up all three of these features into one tool, and does it in a way that's actually *simpler* and more consistent than any of the other task-specific tools!

Ansible aims to be:

1. **Clear** - Ansible uses a simple syntax (YAML) and is easy for anyone (developers, sysadmins, managers) to understand. APIs are simple and sensible.
2. **Fast** - Fast to learn, fast to set up—especially considering you don't need to install extra agents or daemons on all your servers!
3. **Complete** - Ansible does three things in one, and does them very well. Ansible's 'batteries included' approach means you have everything you need in one complete package.
4. **Efficient** - No extra software on your servers means more resources for your applications. Also, since Ansible modules work via JSON, you can easily extend Ansible with modules in a programming language you already know.

[7] https://twitter.com/laserllama
[8] http://www.cobblerd.org/
[9] https://fedorahosted.org/func/
[10] http://www.ansible.com/blog/2013/12/08/the-origins-of-ansible
[11] http://highscalability.com/blog/2012/4/18/ansible-a-simple-model-driven-configuration-management-and-c.html

5. **Secure** - Ansible uses SSH, and requires no extra open ports or potentially-vulnerable daemons on your servers.

Ansible also has a lighter side that gives the project a little personality. As an example, Ansible's major releases are named after Van Halen songs (e.g. 1.4 was named after 1980's "Could This Be Magic", and 1.5 after 1986's "Love Walks In"). Additionally, Ansible will use `cowsay`, if installed, to wrap output in an ASCII cow's speech bubble (this behavior can be disabled in Ansible's configuration).

Ansible, Inc.[12] was founded by Saïd Ziouani (@SaidZiouani[13] on Twitter) and Michael DeHaan, and oversees core Ansible development and provides support (such as Ansible Guru[14]) and extra tooling (such as Ansible Tower[15]) to organizations using Ansible. Hundreds of individual developers have contributed patches to Ansible, and Ansible is the most starred infrastructure management tool on GitHub (with over 4,000 stars as of this writing). Ansible, Inc. has proven itself to be a good steward and promoter of Ansible so far, and I see no indication of this changing in the future.

Ansible Examples

There are many Ansible examples (playbooks, roles, infrastructure, configuration, etc.) throughout this book. Most of the examples are in the Ansible for DevOps GitHub repository[16], so you can browse the code in its final state while you're reading the book. Some of the line numbering may not match the book *exactly* (especially if you're reading an older version of the book!), but I will try my best to keep everything synchronized over time.

Other resources

We'll explore all aspects of using Ansible to provision and manage your infrastructure in this book, but there's no substitute for the wealth of documentation and community interaction that make Ansible great. Check out the links below to find out more about Ansible and discover the community:

- Ansible Documentation[17] - Covers all Ansible options in depth. There are few open source projects with documentation as clear and thorough.
- Ansible Mailing List[18] - Discuss Ansible and submit questions with Ansible's community via this Google group.

[12] http://www.ansible.com/
[13] https://twitter.com/SaidZiouani
[14] http://www.ansible.com/guru
[15] http://www.ansible.com/tower
[16] https://github.com/geerlingguy/ansible-for-devops
[17] http://docs.ansible.com/
[18] https://groups.google.com/forum/#!forum/ansible-project

- Ansible on GitHub[19] - The official Ansible code repository, where the magic happens.
- Ansible Example Playbooks on GitHub[20] - Many examples for common server configurations.
- Getting Started with Ansible[21] - A simple guide to Ansible's community and resources.
- Ansible Blog[22]
- Ansible Weekly[23] - A newsletter about Ansible, including notable cowsay quotes!

I'd like to especially highlight Ansible's documentation (the first resource listed above); one of Ansible's greatest strengths is its well-written and extremely relevant documentation, containing a large number of relevant examples and continously-updated guides. Very few projects—open source or not—have documentation as thorough yet easy-to-read as Ansible's. This book is meant as a supplement to, not a replacement for, Ansible's documentation!

[19] https://github.com/ansible/ansible
[20] https://github.com/ansible/ansible-examples
[21] http://www.ansible.com/get-started
[22] http://www.ansible.com/blog
[23] http://devopsu.com/newsletters/ansible-weekly-newsletter.html

Chapter 1 - Getting Started with Ansible

Ansible and Infrastructure Management

On snowflakes and shell scripts

Many developers and system administrators manage servers by logging into them via SSH, making changes, and logging off. Some of these changes would be documented, some would not. If an admin needed to make the same change to many servers (for example, changing one value in a config file), the admin would manually log into *each* server and repeatedly make this change.

If there were only one or two changes in the course of the server's lifetime, and if the server were extremely simple (running only one process, with one configuration, and a very simple firewall), *and* if every change were thoroughly-documented, this process wouldn't be a problem.

But for almost every company in existence, servers are more complex—most run tens, sometimes hundreds of different applications. Most servers have complicated firewalls and dozens of tweaked configuration files. And even with change documentation, the manual process usually results in some servers or some steps being forgotten.

If the admins at these companies wanted to set up a new server *exactly* like one that is currently running, they would need to spend a lot of time going through all the installed packages, documenting configurations, versions, and settings, and would spend a lot of unnecessary time manually reinstalling, updating, and tweaking everything to get the server to run close to how the old server did.

Some admins may use shell scripts to try to reach some level of sanity, but I've yet to see a complex shell script that handles all edge cases correctly while synchronizing multiple servers' configuration and deploying new code.

Configuration management

Lucky for you, there are tools to help you avoid having these *snowflake servers*—servers that are uniquely-configured and impossible to recreate from scratch because they were hand-configured without documentation. Tools like CFEngine[24], Puppet[25] and Chef[26] became very popular in the mid and late 2000s.

[24] http://cfengine.com/

[25] http://puppetlabs.com/

[26] http://www.getchef.com/chef/

But there's a reason why many developers and sysadmins stick to shell scripting and command line configuration: it's simple, easy to use, and they've had years of experience using bash and command-line tools. Why throw all that out the window and learn a new configuration language and methodology?

Enter Ansible. Ansible was built (and continues to be improved) by developers and sysadmins who know the command line, and want to make a tool that helps them manage their servers exactly the same as they have in the past, but in a repeatable and centrally-managed way. Not only this, Ansible has many other tricks up its sleeve, making it a true swiss army knife for people involved in DevOps (not just the operations side).

One of Ansible's greatest strengths is its ability to run regular shell commands verbatim, so you can take existing scripts and commands, and work on converting them into idempotent playbooks as time allows. For someone (like me) who was comfortable with the command line, but never became proficient in more complicated tools like Puppet or Chef (which both required at least a *slight* understanding of Ruby and/or a custom language just to get started), Ansible was a breath of fresh air.

Ansible works by pushing changes out to all your servers (by default), and requires no extra software to be installed on your servers (thus no extra memory footprint, and no extra daemon to manage), unlike most other configuration management tools.

Idempotence is the ability to run an operation which produces the same result whether run once or multiple times (source[27]).

An important feature of a configuration management tool is its ability to ensure the same configuration is maintained whether you run it once or a thousand times. Many shell scripts have unintended consequences if run more than once, but Ansible can deploy the same configuration to a server over and over again without making any changes after the first deployment.

In fact, almost every aspect of Ansible modules and commands is idempotent, and for those that aren't, Ansible allows you to define when the given command should be run, and what constitutes a changed or failed command, so you can easily maintain an idempotent configuration on all your servers.

Installing Ansible

Ansible's only real dependency is Python. Once Python is installed, the simplest way to get Ansible running is to use `pip`, a simple package manager for Python.

If you're on a Mac, installing Ansible is a piece of cake:

[27] http://en.wikipedia.org/wiki/Idempotence#Computer_science_meaning

1. Install Homebrew[28] (get the installation command from the Homebrew website).
2. Install Python 2.7.x (`brew install python`).
3. Install Ansible (`sudo pip install ansible`).

You could also install Ansible via Homebrew with `brew install ansible`. Either way (`pip` or `brew`) is fine, but make sure you update Ansible using the same system with which it was installed!

If you're running Windows (i.e. you work for a large company that forces you to use Windows), it will take a little extra work to everything set up. There are two ways you can go about using Ansible if you use Windows:

1. The easiest solution would be to use a Linux virtual machine (with something like VirtualBox) to do your work. For detailed instructions, see Appendix A - Using Ansible on Windows workstations.
2. Ansible runs (somewhat) within an appropriately-configured Cygwin[29] environment. For setup instructions, please see my blog post Running Ansible within Windows[30]), and note that *running Ansible directly within Windows is unsupported and prone to breaking.*

If you're running Linux, chances are you already have Ansible's dependencies installed, but we'll cover the most common installation methods.

Primarily, if you have `python-pip` and `python-devel` (`python-dev` on Debian/Ubuntu) installed, you can just use `pip` to install ansible (this assumes you also have the 'Development Tools' package installed, so you have `gcc`, `make`, etc. available):

```
$ sudo pip install ansible
```

Using pip allows you to upgrade ansible with `pip install --upgrade ansible`.

Fedora/RHEL/CentOS:

The easiest way to install Ansible on a Fedora-like system is to use the official yum package, available via EPEL. Install EPEL's RPM if it's not already installed (see the info section below for instructions), then install Ansible:

```
$ yum -y install ansible
```

[28] http://brew.sh/
[29] http://cygwin.com/
[30] https://servercheck.in/blog/running-ansible-within-windows

On Fedora/RHEL/CentOS systems, `python-pip` and `ansible` are available via the EPEL repository[31]. If you run the command `yum repolist | grep epel` (to see if the EPEL repo is already available) and there are no results, you need to install it with the following command:

```
$ rpm -ivh http://dl.fedoraproject.org/pub/epel/6/x86_64/\
epel-release-6-8.noarch.rpm
```

Debian/Ubuntu:

The simplest way of installing Ansible is with `pip`, though there is a PPA available for Ubuntu to make installation even easier; see Rodney Quillo's Ansible PPA[32].

```
$ sudo apt-get update
$ sudo apt-get -y install python-pip python-dev
$ sudo pip install ansible
```

If you run a different flavor of Linux, the steps are similar—you need the Python development headers and pip, then you can use `pip install ansible`.

Once Ansible is installed, make sure it's working properly by entering `ansible --version` on the command line. You should see the currently-installed version:

```
$ ansible --version
ansible 1.8.0
```

Creating a basic inventory file

Ansible uses an inventory file (basically, a list of servers) to communicate with your servers. Like a hosts file (at `/etc/hosts`) that matches IP addresses to domain names, an Ansible inventory file matches servers (IP addresses or domain names) to groups. Inventory files can do a lot more, but for now, we'll just create a simple file with one server. Create a file at `/etc/ansible/hosts` (the default location for Ansible's inventory file), and add one server to it:

```
$ sudo mkdir /etc/ansible
$ sudo touch /etc/ansible/hosts
```

Edit this hosts file with nano, vim, or whatever editor you'd like, but note you'll need to edit it with sudo as root. Put the following into the file:

[31] https://fedoraproject.org/wiki/EPEL

[32] https://launchpad.net/~rquillo/+archive/ansible

```
[example]
www.example.com
```

...where `example` is the group of servers you're managing and `www.example.com` is the domain name (or IP address) of a server in that group. If you're not using port 22 for SSH on this server, you will need to add it to the address, like `www.example.com:2222`, since Ansible defaults to port 22 and won't get this value from your ssh config file.

Running your first Ad-Hoc Ansible command

Now that you've installed Ansible and created an inventory file, it's time to run a command to see if everything works! Enter the following in the terminal (we'll do something safe so it doesn't make any changes on the server):

```
$ ansible example -m ping -u [username]
```

...where `[username]` is the user you use to log into the server. If everything worked, you should see a message that shows `www.example.com | success >>`, then the result of your ping. If it didn't work, run the command again with `-vvvv` on the end to see verbose output. Chances are you don't have SSH keys configured properly—if you login with `ssh username@www.example.com` and that works, the above Ansible command should work, too.

Ansible assumes you're using passwordless (key-based) login for SSH (e.g. you login by entering `ssh username@example.com` and don't have to type a password). If you're still logging into your remote servers with a username and password, or if you need a primer on Linux remote authentication and security best practices, please read Chapter 10 - Server Security and Ansible. If you insist on using passwords, you can add the `--ask-pass` (`-k`) flag to ansible commands, but this entire book is written assuming passwordless authentication, so you'll need to keep this in mind every time you run a command or playbook.

Let's run a more useful command:

```
$ ansible example -a "free -m" -u [username]
```

In this example, we can quickly see memory usage (in a human readable format) on all the servers (for now, just one) in the `example` group. Commands like this can be helpful in quickly finding a server that has a value out of a normal range. I often use commands like `free -m` (to see memory statistics), `df -h` (to see disk usage statistics), and the like to make sure none of my servers is behaving erratically. While it's good to track these details in an external tool like Nagios[33], Munin[34], or Cacti[35], it's also nice to check these stats on all your servers with one simple command and one terminal window!

[33] http://www.nagios.org/
[34] http://munin-monitoring.org/
[35] http://www.cacti.net/

Summary

That's it! You've just learned about configuration management and Ansible, installed it, told it about your server, and ran a couple commands on that server through Ansible. If you're not impressed yet, that's okay—you've only seen the *tip* of the iceberg.

```
 ______________________________________
/ A doctor can bury his mistakes but an \
| architect can only advise his clients |
\ to plant vines. (Frank Lloyd Wright)  /
 --------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Chapter 2 - Local Infrastructure Development: Ansible and Vagrant

Prototyping and testing with local virtual machines

Ansible works great with any server to which you can connect—remote *or* local. For speedier testing and development of Ansible playbooks, and for testing in general, it's a very good idea to work locally. Local development and testing of infrastructure is both safer and faster than doing it on remote/live machines—especially in production environments!

In the past decade, test-driven development (TDD), in one form or another, has become the norm for much of the software industry. Infrastructure development hasn't been as organized until more recently, and best practices dictate that infrastructure (which is becoming more and more important to the software that runs on it) should be thoroughly tested as well.

Changes to software are tested either manually or in some automated fashion, and there are now systems that integrate with Ansible and other deployment and configuration management tools to allow some amount of infrastructure testing as well. Even if it's just testing a configuration change locally before applying it to production, that is a thousand times better than what, in the software development world, would be called 'cowboy coding'—working directly in a production environment, not documenting or encapsulating changes in code, and not having a way to roll back to a previous version.

The past decade has seen the growth of many virtualization tools that allow for flexible and very powerful infrastructure emulation, all from your local workstation! It's empowering to be able to play around with a config file, or tweak the order of a server update to perfection, over and over again, with no fear of breaking an important server. If you use a local virtual machine, there's no downtime for a server rebuild; just re-run the provisioning on a new VM, and you're back up and running in minutes, no one the wiser.

Vagrant[36], a server provisioning tool, and VirtualBox[37], a local virtualization environment, make a potent combination for testing infrastructure and individual server configurations locally. Both applications are free and open source, and work well on Mac, Linux, or Windows hosts.

We're going to set up Vagrant and VirtualBox so we can work on using Ansible to provision a new server.

[36] http://www.vagrantup.com/

[37] https://www.virtualbox.org/

Your first local server: Setting up Vagrant

To get started with your first local virtual server, you need to download and install Vagrant and VirtualBox, and set up a simple Vagrantfile, which will describe the virtual server.

1. Download and install Vagrant and VirtualBox (whatever version is appropriate for your OS): - Download Vagrant[38] - Download VirtualBox[39] (when installing, make sure the command line tools are installed so Vagrant can work with it)
2. Create a new folder somewhere on your hard drive where you will keep your Vagrantfile and provisioning instructions.
3. Open a Terminal or PowerShell window, and navigate to the folder you just created.
4. Add a CentOS (6.4) 64-bit 'box' using the `vagrant box add`[40] command: `vagrant box add centos64 http://puppet-vagrant-boxes.puppetlabs.com/centos-64-x64-vbox4210-nocm.box` (note: You can find a comprehensive list of different pre-made Linux boxes at Vagrantbox.es[41]), or check out the 'official' Vagrant Ubuntu boxes on the Vagrant wiki[42].
5. Create a default virtual server configuration using the box you just downloaded: `vagrant init centos64`
6. Boot your CentOS server: `vagrant up`

Vagrant has downloaded a pre-built 64-bit CentOS 6.4 virtual machine (you can build your own[43] virtual machine 'boxes', if you so desire), loaded it into VirtualBox with the configuration defined in the default Vagrantfile (which is now in the folder you created earlier), and booted the virtual machine.

Managing this virtual server is extremely easy: `vagrant halt` will shut down the VM, `vagrant up` will bring it back up, and `vagrant destroy` will completely delete the machine from VirtualBox. A simple `vagrant up` again will re-create it from the base box you originally downloaded.

Now that you have a running server, you can use it just like you would any other server, and you can connect via SSH. To connect, enter `vagrant ssh` from the folder where the Vagrantfile is located. If you want to connect manually, or connect from another application, enter `vagrant ssh-config` to get the required SSH details.

Using Ansible with Vagrant

Vagrant's ability to bring up preconfigured boxes is convenient on its own, but you could do similar things with the same efficiency using VirtualBox's (or VMWare's, or Parallels') GUI. Vagrant has some other tricks up its sleeve:

[38] http://www.vagrantup.com/downloads.html
[39] https://www.virtualbox.org/wiki/Downloads
[40] http://docs.vagrantup.com/v2/boxes.html
[41] http://www.vagrantbox.es/
[42] https://github.com/mitchellh/vagrant/wiki/Available-Vagrant-Boxes
[43] http://docs.vagrantup.com/v2/virtualbox/boxes.html

- **Network interface management**[44]: You can forward ports to a VM, share the public network connection, or use private networking for inter-VM and host-only communication.
- **Shared folder management**[45]: VirtualBox can set up shares between your host machine and VMs using NFS or (much slower) native folder sharing in VirtualBox.
- **Multi-machine management**[46]: Vagrant is able to configure and control multiple VMs within one Vagrantfile. This is important because, as is stated in the documentation, "Historically, running complex environments was done by flattening them onto a single machine. The problem with that is that it is an inaccurate model of the production setup, which can behave far differently."
- **Provisioning**[47]: When running `vagrant up` the first time, Vagrant automatically *provisions* the newly-minted VM using whatever provisioner you have configured in the Vagrantfile. You can also run `vagrant provision` after the VM has been created to explicitly run the provisioner again.

It's this last feature that is most important for us. Ansible is one of many provisioners integrated with Vagrant (others include basic shell scripts, Chef, Docker, Puppet, and Salt). When you call `vagrant provision` (or `vagrant up` the first time, Vagrant passes off the VM to Ansible, and tells Ansible to run a defined Ansible playbook. We'll get into the details of Ansible playbooks later, but for now, we're going to edit our Vagrantfile to use Ansible to provision our virtual machine.

Open the Vagrantfile that was created when we used the `vagrant init` command earlier. Add the following lines just before the final 'end' (Vagrantfiles use Ruby syntax, in case you're wondering):

```
1  # Provisioning configuration for Ansible.
2  config.vm.provision "ansible" do |ansible|
3    ansible.playbook = "playbook.yml"
4    # Run commands as root.
5    ansible.sudo = true
6  end
```

This is a very basic configuration to simply get you started using Ansible with Vagrant. There are many other Ansible options[48] you can use once we get deeper into using Ansible. For now, we just want to set up a very basic playbook—a simple file you create to tell Ansible how to configure your VM.

Your first Ansible playbook

Let's create the Ansible `playbook.yml` file now. Create an empty text file in the same folder as your Vagrantfile, and put in the following contents:

[44] http://docs.vagrantup.com/v2/networking/index.html
[45] http://docs.vagrantup.com/v2/synced-folders/index.html
[46] http://docs.vagrantup.com/v2/multi-machine/index.html
[47] http://docs.vagrantup.com/v2/provisioning/index.html
[48] http://docs.vagrantup.com/v2/provisioning/ansible.html

```
---
- hosts: all
  tasks:
  - name: Ensure NTP (for time synchronization) is installed.
    yum: pkg=ntp state=installed
  - name: Ensure NTP is running.
    service: name=ntpd state=started enabled=yes
```

I'll get into what this playbook is doing in a minute. For now, let's run the playbook on our VM. Make sure you're in the same directory as the Vagrantfile and new playbook.yml file, and enter `vagrant provision`. You should see status messages for each of the 'tasks' you defined, and then a recap showing what Ansible did on your VM—something like the following:

```
PLAY RECAP ********************************************************************
default                    : ok=3    changed=1    unreachable=0    failed=0
```

Ansible just took the simple playbook you defined, parsed the YAML syntax, and ran a bunch of commands via SSH to configure the server as you specified. Let's go through the playbook, step by step:

```
---
```

This first line is simply a marker showing that the rest of the document will be formatted in YAML (read a getting started guide for YAML[49]).

```
- hosts: all
```

This line tells Ansible to which hosts this playbook applies. `all` works here, since Vagrant is invisibly using its own Ansible inventory file (instead of the one we created earlier in `/etc/ansible/hosts`), which just defines the Vagrant VM.

```
  tasks:
```

All the tasks after this line will be run on all hosts (or, in our case, our one VM).

```
  - name: Ensure NTP daemon (for time synchronization) is installed.
    yum: pkg=ntp state=installed
```

This command is the equivalent of running `yum install ntp`, but is much more intelligent; it will check if ntp is installed, and, if not, install it. This is the equivalent of the following shell script:

[49] http://www.yaml.org/start.html

```
if ! rpm -qa | grep -qw ntp; then
    yum install ntp
fi
```

However, the above script is still not quite as robust as Ansible's `yum` command. What if ntpdate is installed, but not ntp? This script would require extra tweaking and complexity to match the simple Ansible yum command, especially after we explore the yum module more intimately (or the `apt` module, when using Ubuntu and Debian-flavored Linux).

```
6   - name: Ensure NTP is running.
7     service: name=ntpd state=started enabled=yes
```

This final task both checks and makes sure the ntpd service is started and running, and sets it to start at system boot. A shell script with the same effect would be:

```
# Start ntpd if it's not already running.
if ps aux | grep -v grep | grep "[n]tpd" > /dev/null
then
    echo "ntpd is running." > /dev/null
else
    /sbin/service ntpd restart > /dev/null
    echo "Started ntpd."
fi
# Make sure ntpd is enabled on system startup.
chkconfig ntpd on
```

You can see how things start getting complex in the land of shell scripts! And this shell script is still not as robust as what you get with Ansible. To maintain idempotency and handle error conditions, you'll have to do a lot of extra work with basic shell scripts than you do with Ansible.

We could be even more terse (and really demonstrate Ansible's powerful simplicity) and not use Ansible's `name` module to give human-readable names to each command, resulting in the following playbook:

```
1  ---
2  - hosts: all
3    tasks:
4    - yum: pkg=ntp state=installed
5    - service: name=ntpd state=started enabled=yes
```

Just as with code and configuration files, documentation in Ansible (e.g. using the `name` function and/or adding comments to the YAML for complicated plays) is not absolutely necessary. However, I'm a firm believer in thorough (but concise) documentation, so I almost always document what my plays will do by providing a `name` for each one. This also helps when you're running the playbooks, so you can see what's going on in a human-readable format.

Summary

Your workstation is on the path towards becoming an infrastructure-in-a-box, and you can now ensure your infrastructure is as well-tested as the code that runs on top if it. And with one small example, you've gotten a glimpse at the simple, but powerful, Ansible playbook. We'll dive deeper into Ansible playbooks later, and we'll also explore Vagrant a little more as we go.

```
 ______________________________________
/ I have not failed, I've just found   \
| 10,000 ways that won't work. (Thomas |
\ Edison)                              /
 --------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Chapter 3 - Ad-Hoc Commands

In the previous chapter, we ended our exploration of local infrastructure testing with Vagrant by creating a very simple Ansible playbook. Earlier still, we use a simple ansible ad-hoc command to run a one-off command on a remote server.

We'll dive deeper into playbooks in coming chapters, but for now, we'll explore how Ansible can help you quickly perform common tasks on and gather data from one or many servers with ad-hoc commands.

Conducting an orchestra

The number of servers managed by an individual administrator has risen dramatically in the past decade, especially as virtualization and growing cloud application usage has become standard fare. As a result, admins have had to find new ways of managing all these servers in a more streamlined fashion.

There are many tasks a systems administrator has to do on a day-to-day basis:

- Apply patches and updates via `yum`, `apt`, and other package managers.
- Check resource usage (disk space, memory, CPU, swap space, network).
- Check log files.
- Manage system users and groups.
- Manage DNS settings, hosts files, etc.
- Copy files to and from servers.
- Deploy applications or run application maintenance.
- Reboot servers.
- Manage cron jobs.

Nearly all of these tasks can be (and usually are) at least partially automated—but some often need a human touch, especially when it comes to diagnosing issues in real time. And in today's complex multi-server environments, logging into servers individually is not a workable solution.

Ansible allows admins to run ad-hoc commands on one or hundreds of machines at the same time, using the `ansible` command. In Chapter 1, we ran a couple commands (`date` and `free -m`) on a server we added to our Ansible inventory file. This chapter will explore ad-hoc commands and multi-server environments in much greater detail. Even if you decide to ignore the rest of Ansible's powerful features, you will be able to manage your servers much more efficiently after reading this chapter.

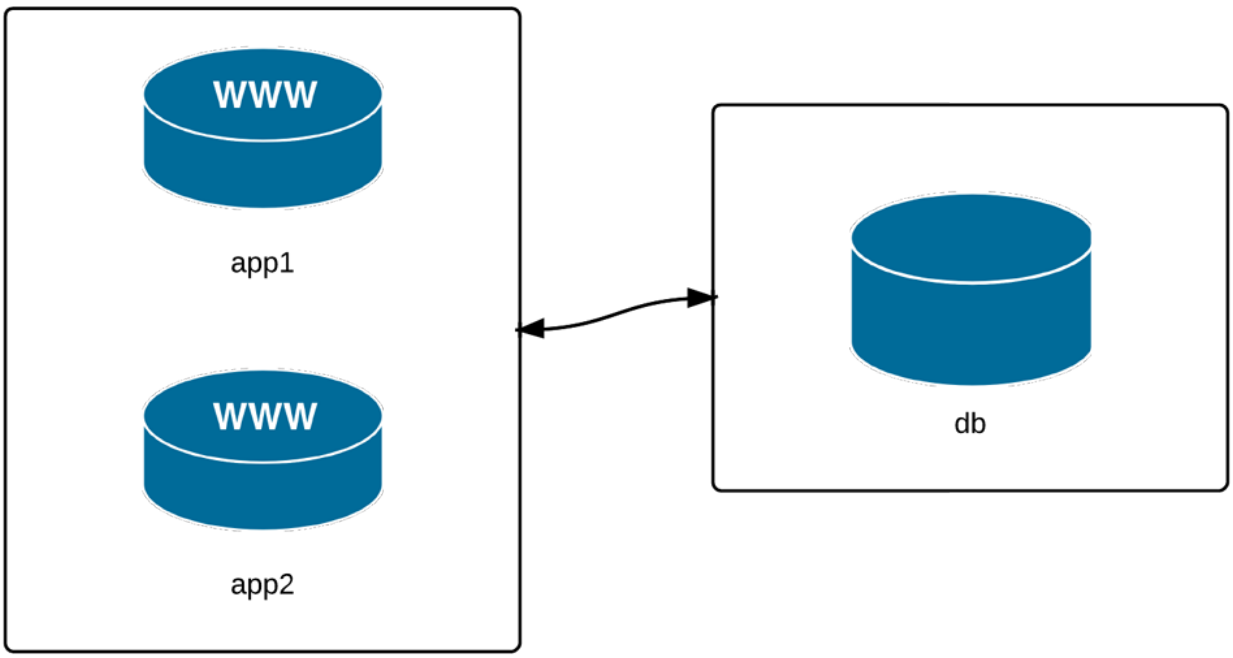
Some of the examples in this chapter will display how you can configure certain aspects of a server with ad-hoc commands. It is usually more appropriate to contain all configuration within playbooks and templates, so it's easier to provision your servers (running the playbook the first time) and then ensure their configuration is idempotent (you can run the playbooks over and over again and your servers will be in the correct state).

The examples in this chapter are for illustration purposes only, and all might not be applicable to your environment. But even if you *only* used Ansible for server management and running individual plays against groups of servers and didn't use Ansible's playbook functionality at all, you'd still have a great orchestration and deployment tool in Ansible!

Build infrastructure with Vagrant for testing

For the rest of this chapter, since we want to do a bunch of experimentation without damaging any production servers, we're going to use Vagrant's powerful multi-machine capabilities to configure a few servers which we will manage with Ansible.

Earlier, we used Vagrant to simply boot up one virtual machine running CentOS 6.4. In that example, we used all of Vagrant's default configuration defined in the Vagrantfile, but in this example, we'll use Vagrant's powerful multi-machine management features.



Three servers: two application, one database.

We're going to manage three VMs: two app servers and a database server. Many simple web applications and websites have a similar architecture, and even though this may not reflect the vast realm of infrastructure combinations that exist, it will be enough to highlight Ansible's server management abilities.

Create a new folder somewhere on your local drive (I like using `~/VMs/[dir]`), and create a new blank file named `Vagrantfile` (this is how we describe our virtual machines to Vagrant). Open the file in your favorite editor, and add the following, then save the file:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# Vagrantfile API/syntax version.
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

  # Configure VM settings for servers running in VirtualBox.
  config.vm.provider :virtualbox do |vb|
    vb.customize ["modifyvm", :id, "--memory", "256"]
  end

  # Application server 1.
  config.vm.define "app1" do |app|
    app.vm.hostname = "orc-app1.dev"

    app.vm.box = "centos64"
    app.vm.box_url = "http://puppet-vagrant-boxes.puppetlabs.com/centos\
    -64-x64-vbox4210-nocm.box"

    app.vm.network :private_network, ip: "192.168.60.4"
  end

  # Application server 2.
  config.vm.define "app2" do |app|
    app.vm.hostname = "orc-app2.dev"

    app.vm.box = "centos64"
    app.vm.box_url = "http://puppet-vagrant-boxes.puppetlabs.com/centos\
    -64-x64-vbox4210-nocm.box"

    app.vm.network :private_network, ip: "192.168.60.5"
  end
```

```
35
36    # Database server.
37    config.vm.define "db" do |db|
38      db.vm.hostname = "orc-db.dev"
39
40      db.vm.box = "centos64"
41      db.vm.box_url = "http://puppet-vagrant-boxes.puppetlabs.com/centos\
42      -64-x64-vbox4210-nocm.box"
43
44      db.vm.network :private_network, ip: "192.168.60.6"
45    end
46  end
```

This Vagrantfile defines the three servers we want to manage, and gives each one a unique hostname, machine name (for VirtualBox), and IP address. For simplicity's sake, all three servers will be running CentOS 6.4.

Open up a terminal window and change directory to the same folder where the Vagrantfile you just created exists. Enter `vagrant up` to let Vagrant begin building the three VMs. If you already downloaded the centos64 box while building the example from Chapter 2, this process shouldn't take too long—maybe 5-10 minutes.

While that's going on, we'll work on telling Ansible about the servers, so we can start managing them right away.

Inventory file for multiple servers

There are many ways you can tell Ansible about the servers you manage, but the most standard, and simplest, is to add them to your system's main Ansible inventory file, which is located at `/etc/ansible/hosts`. If you didn't create the file in the previous chapter, go ahead and create the file now; make sure your user account has read permissions for the file.

Add the following to the file:

```
1  # Lines beginning with a # are comments, and are only included for
2  # illustration. These comments are overkill for most inventory files.
3
4  # Application servers
5  [app]
6  192.168.60.4
7  192.168.60.5
8
9  # Database server
```

```
[db]
192.168.60.6

# Group 'multi' with all servers
[multi:children]
app
db

# Variables that will be applied to all servers
[multi:vars]
ansible_ssh_user=vagrant
ansible_ssh_private_key_file=~/.vagrant.d/insecure_private_key
```

Let's step through this example, group by group:

1. The first block puts both of our application servers into an 'app' group.
2. The second block puts the database server into a 'db' group.
3. The third block tells ansible to define a new group 'multi', with child groups, and we add in both the 'app' and 'db' groups.
4. The fourth block adds variables to the multi group that will be applied to *all* servers within multi and all its children.

We'll dive deeper into variables, group definitions, group hierarchy, and other Inventory file topics later. For now, we just want Ansible to know about our servers so we can start managing them quickly.

Save the updated inventory file, and then check to see if Vagrant has finished building the three VMs. Once Vagrant has finished, we can start managing the servers with Ansible.

Your first ad-hoc commands

One of the first things you need to do is check in on your servers. Let's make sure they're configured correctly, have the right time and date (we don't want any time synchronization-related errors in our application!), and have enough free resources to run an application.

Many of the things we're manually checking here should also be monitored by an automated system on production servers; the best way to prevent disaster is to know when it could be coming, and fix the problem *before* it happens. You should use tools like Munin, Nagios, Cacti, Hyperic, etc. to ensure you have a good idea of your servers' past and present resource usage! If you're running a website or web application available over the Internet, you should probably also use an external monitoring solution like Pingdom or Server Check.in.

Discover Ansible's parallel nature

First, I want to make sure Vagrant configured the VMs with the right hostnames. Use ansible with the `-a` argument 'hostname' to run hostname against all the servers:

```
$ ansible multi -a "hostname"
```

Ansible will run this command against all three of the servers, and return the results (if Ansible can't reach one a server, it will show an error for that server, but continue running the command on the others).

You may have noticed that the command was not run on each server in the order you'd expect. Go ahead and run the command a few more times, and see the order:

```
# First run results:                # Second run results:
192.168.60.5 | success | rc=0 >>   192.168.60.6 | success | rc=0 >>
orc-app2.dev                        orc-db.dev

192.168.60.6 | success | rc=0 >>   192.168.60.5 | success | rc=0 >>
orc-db.dev                          orc-app2.dev

192.168.60.4 | success | rc=0 >>   192.168.60.4 | success | rc=0 >>
orc-app1.dev                        orc-app1.dev
```

By default, Ansible will run your commands in parallel, using multiple process forks, so the command will complete more quickly. If you're managing a few servers, this may not be much quicker than running the command serially, on one server after the other, but even managing 5-10 servers, you'll notice a dramatic speedup if you use Ansible's parallelism (which is enabled by default).

Run the same command again, but this time, add the argument `-f 1` to tell Ansible to only use one fork (basically, perform the command on each server in sequence):

```
$ ansible multi -a "hostname" -f 1
192.168.60.4 | success | rc=0 >>
orc-app1.dev

192.168.60.5 | success | rc=0 >>
orc-app2.dev

192.168.60.6 | success | rc=0 >>
orc-db.dev
```

You can run the same command over and over again, and it will always return results in the same order. It's fairly rare that you will ever need to do this, but it's much more frequent that you'll want to *increase* the value (like `-f 10`, or `-f 25`... depending on how much your system and network connection can handle) to speed up the process of running commands on tens or hundreds of servers.

Most people place the target of the action (`multi`) before the command/action itself ("on X servers, run Y command"), but if your brain works in the reverse order ("run Y command on X servers"), you could put the target *after* the other arguments (`ansible -a "hostname" multi`)—the commands are equivalent.

Learning about your environment

Now that we know we can trust Vagrant's ability to set hostnames correctly, let's make sure everything else is in order.

First, let's make sure the servers have disk space available for our application:

```
$ ansible multi -a "df -h"
192.168.60.5 | success | rc=0 >>
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      8.4G  1.1G  7.0G  13% /
tmpfs                 120M     0  120M   0% /dev/shm
/dev/sda1             485M   33M  427M   8% /boot
/vagrant              233G  200G   34G  86% /vagrant

192.168.60.6 | success | rc=0 >>
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      8.4G  1.1G  7.0G  13% /
tmpfs                 120M     0  120M   0% /dev/shm
/dev/sda1             485M   33M  427M   8% /boot
/vagrant              233G  200G   34G  86% /vagrant

192.168.60.4 | success | rc=0 >>
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      8.4G  1.1G  7.0G  13% /
tmpfs                 120M     0  120M   0% /dev/shm
/dev/sda1             485M   33M  427M   8% /boot
/vagrant              233G  200G   34G  86% /vagrant
```

It looks like we have plenty of room for now; our application is pretty lightweight.

Second, let's also make sure there is enough memory on our servers:

```
$ ansible multi -a "free -m"
192.168.60.5 | success | rc=0 >>
             total       used       free     shared    buffers     cached
Mem:           238        172         66          0         11        103
-/+ buffers/cache:         57        181
Swap:          927          0        927

192.168.60.6 | success | rc=0 >>
             total       used       free     shared    buffers     cached
Mem:           238        172         66          0         11        103
-/+ buffers/cache:         56        181
Swap:          927          0        927

192.168.60.4 | success | rc=0 >>
             total       used       free     shared    buffers     cached
Mem:           238        172         66          0         11        103
-/+ buffers/cache:         56        181
Swap:          927          0        927
```

Memory is pretty tight, but since we're running three VMs on our localhost, we need to be a little conservative.

Third, let's make sure the date and time on each server is in sync:

```
$ ansible multi -a "date"
192.168.60.5 | success | rc=0 >>
Sat Feb  1 20:23:08 UTC 2014

192.168.60.4 | success | rc=0 >>
Sat Feb  1 20:23:09 UTC 2014

192.168.60.6 | success | rc=0 >>
Sat Feb  1 20:23:08 UTC 2014
```

Most applications are written with slight tolerances for per-server time jitter, but it's always a good idea to make sure the times on the different servers are as close as possible, and the simplest way to do that is to use the Network Time Protocol, which is easy enough to configure. We'll do that next, using Ansible's modules to make the process painless.

To get an exhaustive list of all the environment details ('facts', in Ansible's lingo) for a particular server (or a group of servers), use the command `ansible [host-or-group] -m setup`. This will give a list of every minute bit of detail about the server (including file systems, memory, OS, network interfaces… you name it, it's in the list).

Make changes using Ansible modules

We want to install the NTP daemon on the server so it can keep the time in sync. Instead of running the command `yum install -y ntp` on each of the servers, we'll use ansible's `yum` module to do the same (just like we did in the playbook example earlier, but this time using an ad-hoc command).

```
$ ansible multi -s -m yum -a "pkg=ntp state=installed"
```

Hopefully, you'll see three simple 'success' messages, reporting no change, since NTP was already installed on the three machines; but it's good to know everything is in working order.

The `-s` option (alias for `--sudo`) tells Ansible to run the command with sudo. This will work fine with our Vagrant VMs, but if you're running commands against a server where your user account requires a sudo password, you should also pass in `-k` (alias for `--ask-sudo-pass`), so you can enter your sudo password when Ansible needs it.

Now we'll make sure the NTP daemon is started and set to run on boot. We could use two separate commands, `service ntpd start` and `chkconfig ntpd on`, but we'll use Ansible's `service` module instead.

```
$ ansible multi -s -m service -a "name=ntpd state=started enabled=yes"
```

All three servers should show a success message like:

```
"changed": true,
"enabled": true,
"name": "ntpd",
"state": "started"
```

If you run the exact same command again, everything will be the same, but Ansible will report that nothing has changed, so the `"changed"` value becomes `false`.

When you use Ansible's modules instead of plain shell commands, you can use the powers of abstraction and idempotency offered by Ansible. Even if you're running shell commands, you could wrap them in Ansible's `shell` or `command` modules (like `ansible -m shell -a "date" multi`), but for these kind of commands, there's usually no need to use an Ansible module when running them ad-hoc.

The last thing we should do is check to make sure our servers are synced closely to the official time on the NTP server:

```
$ ansible multi -s -a "service ntpd stop"
$ ansible multi -s -a "ntpdate -q 0.rhel.pool.ntp.org"
$ ansible multi -s -a "service ntpd start"
```

For the `ntpdate` command to work, the `ntpd` service has to be stopped, so we stop the service, run the command to check our jitter, then start the service again.

In my test, I was within three one-hundredths of a second on all three servers—close enough for my purposes.

Configure groups of servers, or individual servers

Now that we've been able to get all our servers to a solid baseline (all of them have the correct time, at least), we need to set up the application servers, then the database server.

Since we set up two separate groups in our inventory file, `app` and `db`, we can target commands to just the servers in those groups.

Configure the Application servers

Our hypothetical web application uses Django, so we need to make sure Django and its dependencies are installed. Django is not in the official CentOS yum repository, but we can install it using Python's easy_install (which, conveniently, has an Ansible module).

```
$ ansible app -s -m yum -a "name=MySQL-python state=present"
$ ansible app -s -m yum -a "name=python-setuptools state=present"
$ ansible app -s -m easy_install -a "name=django"
```

You could also install django using `pip`, which can be installed via `easy_install` (since Ansible's `easy_install` module doesn't allow you to uninstall packages like `pip` can), but for simplicity's sake, we've installed it with `easy_install`.

Check to make sure Django is installed and working correctly.

```
$ ansible app -a "python -c 'import django; print django.get_version()'"
192.168.60.4 | success | rc=0 >>
1.6.1

192.168.60.5 | success | rc=0 >>
1.6.1
```

Things look like they're working correctly on our app servers. We can now move on to our database server.

Almost all of the configuration we've done in this chapter would be much better off in an Ansible playbook (which will be explored in greater depth throughout the rest of this book). This chapter is simply demonstrating how easy it is to manage multiple servers—for whatever purpose—using Ansible. Even if you set up and configure servers by hand using shell commands, using Ansible will save you a ton of time and help you do everything in the most secure and efficient manner possible.

Configure the Database servers

We configured the application servers using the `app` group defined in Ansible's main inventory, and we can configure the database server (currently the only server in the `db` group) using the similarly-defined `db` group.

Let's install MySQL, start it, and configure the server's firewall to allow access on MySQL's default port, 3306.

```
$ ansible db -s -m yum -a "name=mysql-server state=present"
$ ansible db -s -m service -a "name=mysqld state=started enabled=yes"
$ ansible db -s -a "iptables -F"
$ ansible db -s -a "iptables -A INPUT -s 192.168.60.0/24 -p tcp \
-m tcp --dport 3306 -j ACCEPT"
```

If you try connecting to the database from the app servers (or your host machine) at this point, you won't be able to connect, since MySQL still needs to be set up. Typically, you'd do this by logging into the server and running `mysql_secure_installation`. Luckily, though, Ansible can control a MySQL server with its assorted `mysql_*` modules. For now, we need to allow MySQL access for one user from our app servers. The MySQL module requires the `MySQL-python` module to be present on the managed server.

```
$ ansible db -s -m yum -a "name=MySQL-python state=present"
$ ansible db -s -m mysql_user -a "name=django host=% password=12345 \
priv=*.*:ALL state=present"
```

At this point, you should be able to create or deploy a Django application on the app servers, then point it at the database server with the username django and password 12345.

The MySQL configuration used here is for example/development purposes only! There are a few other things you should do to secure a production MySQL server, including removing the test database, adding a password for the root user account, restricting the IP addresses allowed to access port 3306 more closely, and some other minor cleanups. Some of these things will be covered later in this book, but, as always, you are responsible for securing your servers—make sure you're doing it correctly!

Make changes to just one server

Congratulations! You now have a small web application environment running Django and MySQL. It's not much, and there's not even a load balancer in front of the app servers to spread out the requests… but we've configured everything pretty quickly, and without ever having to log into a server. What's even more impressive is you could run any of the ansible commands again (besides a couple of the simple shell commands), and they wouldn't change anything—they would simply return `"changed": false`, giving you peace of mind that the original configuration is intact.

Now that your local infrastructure has been running a while, you notice (hypothetically, of course) that the logs indicate one of the two app servers' time has gotten way out of sync with the others, likely because the NTP daemon has crashed or somehow been stopped. Quickly, you enter the following command to check the status of `ntpd`:

```
$ ansible app -s -a "service ntpd status"
```

Then, you restart the service on the affected app server:

```
$ ansible app -s -a "service ntpd restart" --limit "192.168.60.4"
```

In this command, we used the `--limit` argument to limit the command to a specific host in the specified group. `--limit` will match either an exact string or a regular expression (prefixed with ~). The above command could be stated more simply if you want to apply the command to only the `.4` server (assuming you know there are no other servers with the an IP address ending in .4), the following would work exactly the same:

```
# Limit hosts with a simple pattern (asterisk is a wildcard).
$ ansible app -s -a "service ntpd restart" --limit "*.4"

# Limit hosts with a regular expression (prefix with a tilde).
$ ansible app -s -a "service ntpd restart" --limit ~".*\.4"
```

In these examples, we've been using IP addresses instead of hostnames, but in many real-world scenarios, you'll probably be using hostnames like nyc-dev-1.example.com, and being able to match on regular expressions can be helpful.

Try to reserve the `--limit` option for running commands on single servers. If you often find yourself running commands on the same set of servers using `--limit`, consider instead adding them to a group in your inventory file. That way you can just enter `ansible [my-new-group-name] [command]`, and save yourself a few keystrokes.

Manage users and groups

One of the most common uses for Ansible's ad-hoc commands in my day-to-day usage is user and group management. I don't know how many times I've had to re-read the man pages or do a Google search just to remember which arguments I need to create a user with or without a home folder, add the user to certain groups, etc.

Ansible's `user` and `group` modules make things pretty simple, and standard across any Linux flavor.

First, add an `admin` group on the app servers for the server administrators:

```
$ ansible app -s -m group -a "name=admin state=present"
```

The `group` module is pretty simple; you can remove a group by setting `state=absent`, set a group id with `gid=[gid]`, and indicate that the group is a system group with `system=yes`.

Now add the user `johndoe` to the app servers with the group I just created and give him a home folder in `/home/johndoe` (the default location for most Linux distributions). Simple:

```
$ ansible app -s -m user -a "name=johndoe group=admin createhome=yes"
```

If you want to automatically create an SSH key for the new user (if one doesn't already exist), you can run the same command with the additional parameter `generate_ssh_key=yes`. You can also set the UID of the user by passing in `uid=[uid]`, set the user's shell with `shell=[shell]`, and the password with `password=[encrypted-password]`.

What if you want to delete the account?

```
$ ansible app -s -m user -a "name=johndoe state=absent remove=yes"
```

You can do just about anything you could do with `useradd`, `userdel`, and `usermod` using Ansible's `user` module, except you can do it more easily. The official documentation of the User module[50] explains all the possibilities in great detail.

[50] http://docs.ansible.com/user_module.html

Manage files and directories

Another common use for ad-hoc commands is remote file management. Ansible makes it easy to copy files from your host to remote servers, create directories, manage file and directory permissions and ownership, and delete files or directories.

Get information about a file

If you need to simply check a file's permissions, MD5, or owner, use Ansible's `stat` module:

```
$ ansible multi -m stat -a "path=/etc/environment"
```

This gives the same information you'd get when running the `stat` command, but passes back information in JSON, which can be parsed a little more easily (or, later, used in playbooks to conditionally do or not do certain tasks).

Copy a file to the servers

You probably use `scp` and/or `rsync` to copy files and directories to remote servers, and while Ansible has recently gained an `rsync` module, most file copy operations can be completed with Ansible's `copy` module:

```
$ ansible multi -m copy -a "src=/etc/hosts dest=/tmp/hosts"
```

The `src` can be a file or a directory. If you include a trailing slash, only the contents of the directory will be copied into the `dest`. If you omit the trailing slash, the contents *and* the directory itself will be copied into the `dest`.

The `copy` module is perfect for single-file copies, and works very well with small directories. When you want to copy hundreds of files, especially in very deeply-nested directory structures, you should consider either copying then expanding an archive of the files with Ansible's `unarchive` module, or using Ansible's `synchronize` module.

Retrieve a file from the servers

The `fetch` module works almost exactly the same as the `copy` module, except in reverse. The major difference is that files will be copied down to the local `dest` in a directory structure that matches the host from which you copied them. For example, use the following command to grab the hosts file from the servers:

```
$ ansible multi -s -m fetch -a "src=/etc/hosts dest=/tmp"
```

Fetch will, by default, put the `/etc/hosts` file from each server into a folder in the destination with the name of the host (in our case, the three IP addresses), then in the location defined by `src`. So, the db server's hosts file will end up in `/tmp/192.168.60.6/etc/hosts`.

You can add the parameter `flat=yes`, and set the `dest` to `dest=/tmp/` (add a trailing slash), to make Ansible fetch the files directly into the `/tmp` directory. However, filenames must be unique for this to work, so it's not as useful when copying down files from multiple hosts. Only use `flat=yes` if you're copying files from a single host.

Create directories and files

You can use the `file` module to create files and directories (like `touch`), manage permissions and ownership on files and directories, modify SELinux properties, and create symlinks.

Here's how to create a directory:

```
$ ansible multi -m file -a "dest=/tmp/test mode=644 state=directory"
```

Here's how to create a symlink (set `state=link`):

```
$ ansible multi -m file -a "src=/src/symlink dest=/dest/symlink \
owner=root group=root state=link"
```

Delete directories and files

You can set the `state` to `absent` to simply delete a file or directory.

```
$ ansible multi -m file -a "dest=/tmp/test state=absent"
```

There are many simple ways to manage files remotely using Ansible. We've briefly covered the `copy` and `file` modules here, but be sure to read the documentation for the other file-management modules like `lineinfile`, `ini_file`, and `unarchive`. This book will cover these additional modules in depth in later chapters, when dealing with playbooks.

Run operations in the background

Some operations take quite a while (minutes or even hours). For example, when you run `yum update` or `apt-get update && apt-get dist-upgrade`, it could be a few minutes before all the packages on your servers are updated.

In these situations, you can tell Ansible to run the commands asynchronously, and poll the servers to see when the commands finish. When you're only managing one server, this is not really helpful, but if you have many servers, Ansible can *very* quickly (especially if you set a higher `--forks` value) start the command on all your servers, then sit and poll the servers for status until they're all up to date.

To run a command in the background, you set the following options:

- `-B <seconds>`: the maximum amount of time (in seconds) to let the job run.
- `-P <seconds>`: the amount of time (in seconds) to wait between polling the servers for an updated job status.

Update servers asynchronously, monitoring progress

Let's run `yum -y update` on all our servers to get them up to date. If we leave out `-P`, Ansible defaults to polling every 10 seconds:

```
$ ansible multi -s -B 3600 -a "yum -y update"
background launch...


192.168.60.6 | success >> {
    "ansible_job_id": "763350539037",
    "results_file": "/root/.ansible_async/763350539037",
    "started": 1
}

... [other hosts] ...
```

Wait a little while (or a *long* while, depending on how old the system image is we used to build our example VMs!), and eventually, you should see something like:

```
<job 763350539037> finished on 192.168.60.6 => {
    "ansible_job_id": "763350539037",
    "changed": true,
    "cmd": [
        "yum",
        "-y",
        "update"
    ],
    "delta": "0:13:13.973892",
    "end": "2014-02-09 04:47:58.259723",
    "finished": 1,

... [more info and stdout from job] ...
```

While a background task is running, you can also check on the status elsewhere using Ansible's `async_status` module, as long as you have the `ansible_job_id` value to pass in as `jid`:

```
$ ansible multi -m async_status -a "jid=763350539037"
```

Fire-and-forget tasks

You may also need to run occasional long-running maintenance scripts, or other tasks that take many minutes or hours to complete, and you'd rather not babysit the task. In these cases, you can set the `-B` value as high as you want (be generous, so your task will complete before Ansible kills it!), and set `-P` to '0', so Ansible fires off the command then forgets about it:

```
$ ansible multi -B 3600 -P 0 -a "/path/to/fire-and-forget-script.sh"
background launch...


192.168.60.5 | success >> {
    "ansible_job_id": "204960925196",
    "results_file": "/root/.ansible_async/204960925196",
    "started": 1
}

... [other hosts] ...

$
```

You won't be able to track progress using the `jid` anymore, but it's helpful for 'fire-and-forget' tasks.

For tasks you don't track remotely, it's usually a good idea to log the progress of the task *somewhere*, and also send some sort of alert on failure—especially, for example, when running backgrounded tasks that perform backup operations, or when running business-critical database maintenance tasks.

You can also run plays in Ansible playbooks in the background, asynchronously, by defining an `async` and `poll` parameter on the play. We'll discuss playbook task backgrounding more in later chapters.

Check log files

Sometimes, when debugging application errors, or diagnosing outages or other problems, you need to check server log files. Any common log file operation (like using `tail`, `cat`, `grep`, etc.) works through the `ansible` command, with a few caveats:

1. Operations that continuously monitor a file, like `tail -f`, won't work via Ansible, because Ansible only displays output after the operation is complete, and you won't be able to send the Control-C command to stop following the file. Someday, the async module might have this feature, but for now, it's not possible.
2. It's not a good idea to run a command that returns a huge amount of data via stdout via Ansible. If you're going to cat a file larger than a few KB, you should probably log into the server(s) individually.
3. If you redirect and filter output from a command run via Ansible, you need to use the `shell` module instead of Ansible's default `command` module (add `-m shell` to your commands).

As a simple example, let's view the last few lines of the messages log file on each of our servers:

```
$ ansible multi -s -a "tail /var/log/messages"
```

As stated in the caveats, if you want to filter the messages log with something like `grep`, you can't use Ansible's default `command` module, but instead, `shell`:

```
$ ansible multi -s -m shell -a "tail /var/log/messages | \
grep ansible-command | wc -l"

192.168.60.5 | success | rc=0 >>
12

192.168.60.4 | success | rc=0 >>
12

192.168.60.6 | success | rc=0 >>
14
```

This command shows how many ansible commands have been run on each server (the numbers you get may be different).

Manage cron jobs

Periodic tasks run via cron are managed by a system's crontab. Normally, to change cron job settings on a server, you would log into the server, use `crontab -e` under the account where the cron jobs reside, and type in an entry with the interval and job.

Ansible makes managing cron jobs easy with its `cron` module. If you want to run a shell script on all the servers every day at `4` a.m., you can add the cron job with:

```
$ ansible multi -s -m cron -a "name='daily-cron-all-servers' \
hour=4 job='/path/to/daily-script.sh'"
```

Ansible will assume `*` for all values you don't specify (valid values are `day`, `hour`, `minute`, `month`, and `weekday`). You could also specify special time values like `reboot`, `yearly`, or `monthly` using `special_time=[value]`. You can also set the user the job will run under via `user=[user]`, and you can create a backup of the current crontab by passing `backup=yes`.

What if we want to remove the cron job? Simple enough, use the same `cron` command, and pass the name of the cron job you want to delete, and `state=absent`:

```
$ ansible multi -s -m cron -a "name='daily-cron-all-servers' state=absent'"
```

You can also use Ansible to manage custom crontab files; use the same syntax as you used earlier, but specify the location to the cron file with: `cron_file=cron_file_name` (where `cron_file_name` is a cron file located in `/etc/cron.d`).

Ansible denotes Ansible-managed crontab entries by adding a comment on the line above the entry like `#Ansible: daily-cron-all-servers`. It's best to leave things be in the crontab itself, and always manage entries via ad-hoc commands or playbooks using Ansible's `cron` module.

Deploy a version-controlled application

For simple application deployments, where you may simply need to update a git checkout, or copy a new bit of code to a group of servers, then run a command to finish the deployment, Ansible's ad-hoc mode can help. For more complicated deployments, you can use Ansible playbooks and rolling update features (which will be discussed in later chapters) to ensure successful deployments with zero downtime.

In the example below, I'll assume we're running a simple application on one or two servers, in the directory `/opt/myapp`. This directory is a git repository cloned from a central server or a service like GitHub, and application deployments and updates are done by simply updating the clone, then running a shell script at `/opt/myapp/scripts/update.sh`.

First, update the git checkout to the application's new version branch, 1.2.4, on all the app servers:

```
$ ansible app -s -m git -a "repo=git://example.com/path/to/repo.git \
dest=/opt/myapp update=yes version=1.2.4"
```

Ansible's git module lets you specify a branch, tag, or even a specific commit with the `version` parameter (in this case, we chose to checkout tag `1.2.4`, but if you run the command again with a branch name, like `prod`, Ansible will happily do that instead). To force Ansible to update the checked-out copy, we passed in `update=yes`. The `repo` and `dest` options should be self-explanatory.

Then, run the application's `update.sh` shell script:

```
$ ansible app -s -a "/opt/myapp/update.sh"
```

Ad-hoc commands are fine for the simple deployments (like our example above), but you should use Ansible's more powerful and flexible application deployment features described later in this book if you have complex application or infrastructure needs. See especially the 'Rolling Updates' section later in this book.

Ansible's SSH connection history

One of Ansible's greatest features is its ability to function without running any extra applications or daemons on the servers it manages. Instead of using a proprietary protocol to communicate with

the servers, Ansible uses the standard and secure SSH connection that is commonly used for basic administration on almost every Linux server running today.

Since a stable, fast, and secure SSH connection is the heart of Ansible's communication abilities, Ansible's implementation of SSH has continually improved throughout the past few years—and is still improving today.

One thing that is universal to all of Ansible's SSH connection methods is that Ansible uses the connection to transfer one or a few files defining a play or command to the remote server, then runs the play/command, then deletes the transferred file(s), and reports back the results. This sequence of events may change and become more simple/direct with later versions of Ansible (see the notes on Ansible 1.5 below), but a fast, stable, and secure SSH connection is of paramount importance to Ansible.

Paramiko

In the beginning, Ansible used paramiko—an open source SSH2 implementation for Python—exclusively. However, as a single library for a single language (Python), development of paramiko doesn't keep pace with development of OpenSSH (the standard implementation of SSH used almost everywhere), and its performance and security is slightly worse than OpenSSH—at least to this writer's eyes.

Ansible continues to support the use of paramiko, and even chooses it as the default for systems (like RHEL 5/6) which don't support ControlPersist—an option present only in OpenSSH 5.6 or newer. (ControlPersist allows SSH connections to persist so frequent commands run over SSH don't have to go through the initial handshake over and over again until the `ControlPersist` timeout set in the server's SSH config is reached.)

OpenSSH (default)

Beginning in Ansible 1.3, Ansible defaulted to using native OpenSSH connections to connect to servers supporting ControlPersist. Ansible had this ability since version 0.5, but didn't default to it until 1.3.

Most local SSH configuration parameters (like hosts, key files, etc.) are respected, but if you need to connect via a port other than port 22 (the default SSH port), you need to specify the port in an inventory file (`ansible_ssh_port` option) or when running `ansible` commands.

OpenSSH is faster, and a little more reliable, than paramiko, but there are ways to make Ansible faster still.

Accelerated Mode

While not too helpful for ad-hoc commands, Ansible's Accelerated mode can achieve greater performance for playbooks. Instead of connecting repeatedly via SSH, Ansible connects via SSH

initially, then uses the AES key used in the initial connection to communicate further commands and transfers via a separate port (5099 by default, but this is configurable).

The only extra package required to use accelerated mode is `python-keyczar`, and almost everything you can do in normal OpenSSH/Paramiko mode works in Accelerated mode, with two exceptions when using `sudo`:

- Your sudoers file needs to have `requiretty` disabled (comment out the line with it, or set it per user by changing the line to `Defaults:username !requiretty`).
- You must disable sudo passwords by setting `NOPASSWD` in the sudoers file.

Accelerated mode can offer 2-4 times faster performance (especially for things like file transfers) compared to OpenSSH, and you can enable it for a playbook by adding the option `accelerate: true` to your playbook, like so:

```
---
- hosts: all
  accelerate: true
  ...
```

It goes without saying, if you use accelerated mode, you need to have the port through which it communicates open in your firewall (port 5099 by default, or whatever port you set with the `accelerate_port` option after `accelerate`).

Accelerate mode is a spiritual descendant of the now-deprecated 'Fireball' mode, which used a similar method for accelerating Ansible communications, but required ZeroMQ to be installed on the controlled server (which is at odds with Ansible's simple no-dependency, no-daemon philosophy), and didn't work with sudo commands at all.

Faster OpenSSH in Ansible 1.5+

Ansible 1.5 and later contains a very nice improvement to Ansible's default OpenSSH implementation.

Instead of copying files, running them on the remote server, then removing them, the new method of OpenSSH transfer will simply send and execute commands for most Ansible modules directly over the SSH connection.

This method of connection is only available in Ansible 1.5+, and it can be enabled by adding `pipelining=True` under the `[ssh_connection]` section of the Ansible configuration file (`ansible.cfg`, which will be covered in more detail later).

The `pipelining=True` configuration option won't help much unless you have removed or commented the `Defaults requiretty` option in `/etc/sudoers`. This is commented out in the default configuration for most OSes, but you might want to double-check this setting to make sure you're getting the fastest connection possible!

If you're running a recent version of Mac OS X, Ubuntu, Windows with Cygwin, or most other OS for the host from which you run `ansible` and `ansible-playbook`, you should be running OpenSSH version 5.6 or later, which works perfectly with the `ControlPersist` setting used with all of Ansible's SSH connections settings.

If the host on which Ansible runs has RHEL or CentOS, however, you might need to update your version of OpenSSH so it supports the faster/persistent connection method. Any OpenSSH version 5.6 or greater should work. To install a later version, you can either compile from source, or use a different repository (like CentALT[51] and `yum update openssh`.

Summary

In this chapter, you learned how to build a multi-server infrastructure for testing on your local workstation using Vagrant, and you configured, monitored, and managed the infrastructure without ever logging in to an individual server. You also learned how Ansible connects to remote servers, and how we can use the `ansible` command to perform tasks on many servers quickly in parallel, or one by one.

By now, you should be getting familiar with the basics of Ansible, and you should be able to start managing your own infrastructure more efficiently.

```
 ______________________________________
/ It's easier to seek forgiveness than \
\ ask for permission. (Proverb)        /
 --------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

[51] http://mirror.neu.edu.cn/CentALT/readme.txt

Chapter 4 - Ansible Playbooks

Power plays

Like many other configuration management solutions, Ansible uses a metaphor to describe its configuration files. They are called 'playbooks', and they list a set of tasks ('plays' in Ansible parlance) that will be run against a particular server or set of servers.

Playbooks are written in YAML[52], a human-readable data format that is very popular for defining configuration in a simple text format. They can be included within other playbooks, and certain metadata and options can cause different plays or playbooks to be run in different scenarios on different servers. Think of a football (American, not the game played with your feet) playbook; there are many different plays that can be run, and the coaches will pick the plays appropriate for a given situation. You will write many different plays, and you can use some on all servers, some on a few types of servers, and some on just one or two servers—as the current situation requires.

Ad-hoc commands alone can make Ansible a powerful tool; playbooks turn Ansible into a top-notch server provisioning and configuration management tool.

What attracts most DevOps personnel to Ansible is the fact that it is easy to convert shell scripts (or one-off shell commands) directly into Ansible plays. Consider the following script, which simply installs Apache on a RHEL/CentOS server:

Shell Script

```
# Install Apache.
yum install --quiet -y httpd httpd-devel
# Copy configuration files.
cp /path/to/config/httpd.conf /etc/httpd/conf/httpd.conf
cp /path/to/config/httpd-vhosts.conf /etc/httpd/conf/httpd-vhosts.conf
# Start Apache and configure it to run at boot.
service httpd start
chkconfig httpd on
```

To run the shell script (in this case, a file named `shell-script.sh` with the contents as above), you would call it directly from the command line:

[52] http://docs.ansible.com/YAMLSyntax.html

```
# (From the same directory in which the shell script resides).
$ ./shell-script.sh
```

Ansible Playbook

```
---
- hosts: all
  tasks:
  - name: Install Apache.
    command: yum install --quiet -y httpd httpd-devel
  - name: Copy configuration files.
    command: >
      cp /path/to/config/httpd.conf /etc/httpd/conf/httpd.conf
  - command: >
      cp /path/to/config/httpd-vhosts.conf /etc/httpd/conf/httpd-vhosts.conf
  - name: Start Apache and configure it to run at boot.
    command: service httpd start
  - command: chkconfig httpd on
```

To run the Ansible Playbook (in this case, a file named `playbook.yml` with the contents as above), you would call it using the `ansible-playbook` command:

```
# (From the same directory in which the playbook resides).
$ ansible-playbook playbook.yml
```

Ansible is powerful in that you can quickly transition to using playbooks if you know how to write standard shell commands—the same commands you've been using for years—and then rebuild your configuration to be more robust and take advantage of Ansible's features as you get time.

In the above playbook, we use Ansible's `command` module to run standard shell commands. We're also giving each play a 'name', so when we run the playbook, the play has human-readable output on the screen or in the logs. The command module has some other tricks up its sleeve (which we'll see later), but for now, you can be assured that shell scripts can be translated directly into Ansible playbooks without much hassle.

The greater-than sign (`>`) immediately following the `command:` module directive tells YAML "automatically quote the next set of indented lines as one long string, with each line separated by a space". It helps improve task readability (especially with modules which use many parameters like `copy`, `template`, or `lineinfile`, so you can place each parameter on its own line. It's easier to see what each option is doing, and your version control software is better able to identify changes on a line-by-line basis.

The above playbook will perform *exactly* like the shell script, but we can improve things greatly by using some of Ansible's built-in modules to handle the heavy lifting:

Revised Ansible Playbook - Now with idempotence!

```
---
- hosts: all
  tasks:
  - name: Install Apache.
    yum: pkg={{ item }} state=present
    with_items:
    - httpd
    - httpd-devel
  - name: Copy configuration files.
    copy: >
      src={{ item.src }}
      dest={{ item.dest }}
      owner=root group=root mode=644
    with_items:
    - {
      src: "/path/to/config/httpd.conf",
      dest: "/etc/httpd/conf/httpd.conf"
    }
    - {
      src: "/path/to/config/httpd-vhosts.conf",
      dest: "/etc/httpd/conf/httpd-vhosts.conf"
    }
  - name: Make sure Apache is started and configure it to run at boot.
    service: name=httpd state=started enabled=yes
```

Now we're getting somewhere. Let me walk you through this simple playbook:

1. The first line, `---`, is how we mark this document as using YAML syntax (like using `<html>` at the top of an HTML document, or `<?php` at the top of a block of PHP code).
2. The second line, `tasks:`, tells Ansible that what follows is a list of tasks to run as part of this playbook.
3. The first task begins with `name: Install Apache.`. `name` is not a module that does something to your server; rather, it's a way of giving a human-readable description to the play that follows. Seeing "Install Apache" is more relevant than seeing "yum pkg=httpd state=installed"... but if you drop the name line completely, that won't cause any problem.
    - We use the `yum` module to install Apache. Instead of the command `yum -y install httpd httpd-devel`, we can describe to Ansible exactly what we want. Ansible will take the

`items` array we pass in (`{{ variable }}` references a variable in Ansible's playbooks). We tell yum to make sure the packages we define are installed with `state=present`, but we could also use `state=latest` to ensure the latest version is installed, or `state=absent` if we want to make sure the package is *not* installed.
    - Ansible allows simple lists to be passed into plays using `with_items`: Simply define a list of items below, and each line will be passed into the play, one by one. In this case, each of the items will be substituted for the `{{ item }}` variable.
4. The second task again starts with a human-readable name (which could be left out if you'd like).
    - We use the `copy` module to copy files from a source (on our local workstation) to a destination (the server being managed). We can also pass in more variables, like file metadata including ownership and permissions (`owner`, `group`, and `mode`).
    - In this case, we are using an array with multiple elements for variable substitution; you use the syntax `{var1: value, var2: value}` to define each element (it can have as many variables as you want within, or even nested levels of variables!). When you reference the variables in the play, you simply use a dot to access the variable within the item, so `{{ item.var1 }}` would access the first variable. In our example, `item.src` access the `src` in each item.
5. The third task also uses a name to describe it in a human-readable format.
    - We use the `service` module to describe the desired state of a particular service, in this case `httpd`, Apache's http daemon. We want it to be running, so we set `state=started`, and we want it to run at system startup, so we say `enabled=yes` (the equivalent of running `chkconfig httpd on`).

The great thing about the way we've reformatted this list of commands Ansible can keep track of the state of everything on all our servers. If you run the playbook the first time, it will provision the server by ensuring Apache is installed and running, and your custom configuration is in place.

Even better, the *second* time you run it, it won't actually do anything—besides telling you nothing has changed—as long as the server is in the correct state. So, with this one short playbook, we're able to provision and ensure the proper configuration for an Apache web server. Additionally, you will could run the playbook with the `--check` option (see the next section below) to ensure that the configuration matches what's defined in the playbook, without actually running the tasks on the server.

If you ever want to update your configuration, or install another httpd package, you can simply update the file locally, or add the package to the `with_items` list, and run the playbook again. Whether you have one or a thousand servers, all of their configurations will be updated to match your playbook—and Ansible will tell you if anything ever changes (you're not making ad-hoc changes on individual production servers, *are you*?).

Running Playbooks with `ansible-playbook`

If we run the playbooks in the examples above (which are set to run on `all` hosts), then the playbook would be run against every host defined in your Ansible inventory file (see Chapter 1's basic inventory file example).

Limiting playbooks to particular hosts and groups

You can limit a playbook to specific groups or individual hosts by changing the `hosts:` definition. It can be set to a `all` hosts, a `group` of hosts defined in your inventory, multiple groups of hosts (e.g. `webservers,dbservers`), individual hosts (e.g. `atl.example.com`), or a mixture of hosts. You can even do wildcard matches, like `*.example.com`, to match all subdomains of a top-level domain.

You can also limit the hosts on which the playbook is run via the `ansible-playbook` command:

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

In this case (assuming your inventory file contains a `webservers` group), even if the playbook is set to `hosts: all`, or includes hosts in addition to what's defined in the `webservers` group, it will only be run on the hosts defined in `webservers`.

You could also limit the playbook to one particular host:

```
$ ansible-playbook playbook.yml --limit xyz.example.com
```

If you want to simply see a list of hosts that would be affected by your playbook before you actually run it, use `--list-hosts`:

```
$ ansible-playbook playbook.yml --list-hosts
```

Running this should give output like:

```
playbook: playbook.yml

  play #1 (all): host count=4
    127.0.0.1
    192.168.24.2
    foo.example.com
    bar.example.com
```

(Where `count` is the count of servers defined in your inventory, and following is a list of all the hosts defined in your inventory).

Setting user and sudo options with `ansible-playbook`

If no `user` is defined alongside the `hosts` in a playbook, Ansible assumes you'll connect as the user defined in your inventory file for a particular host, and then will fall back to your local user account name. You can explicitly define a remote user to use for remote plays using the `--remote-user` (`-u`) option:

```
$ ansible-playbook playbook.yml --remote-user=johndoe
```

In some situations, you will need to pass along your sudo password to the remote server to perform commands via `sudo`. In these situations, you'll need use the `--ask-sudo-pass` (`-K`) option. You can also explicitly force all tasks in a playbook to use sudo with `--sudo`. Finally, you can define the sudo user for tasks run via `sudo` (the default is root) with the `--sudo-user` (`-U`) option.

For example, the following command will run our example playbook with sudo, performing the tasks as the sudo user `janedoe`, and Ansible will prompt you for the sudo password:

```
$ ansible-playbook playbook.yml --sudo --sudo-user=janedoe --ask-sudo-pass
```

If you're not using key-based authentication to connect to your servers (read my warning about the security implications of doing so in Chapter 1), you can use `--ask-pass`.

Other options for `ansible-playbook`

The `ansible-playbook` command also allows for some other common options:

- `--inventory=PATH` (`-i PATH`): Define a custom inventory file (default is the default Ansible inventory file, usually located at `/etc/ansible/hosts`).
- `--verbose` (`-v`): Verbose mode (show all output, including output from successful options). You can pass in `-vvvv` to give every minute detail.
- `--extra-vars=VARS` (`-e VARS`): Define variables to be used in the playbook, in `"key=value,key=value"` format.
- `--forks=NUM` (`-f NUM`): Number for forks (integer). Set this to a number higher than `5` to increase the number of servers on which Ansible will run tasks concurrently.
- `--connection=TYPE` (`-c TYPE`): The type of connection which will be used (this defaults to `ssh`; you might sometimes want to use `local` to run a playbook on your local machine, or on a remote server via cron).
- `--check`: Run the playbook in Check Mode ('Dry Run'); all tasks defined in the playbook will be checked against all hosts, but none will actually be run.

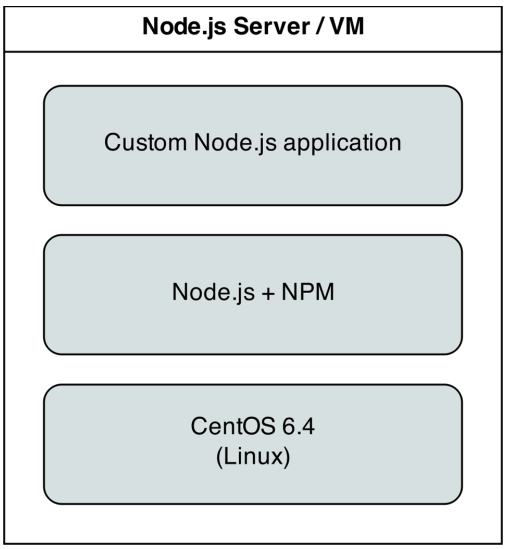
There are some other options and configuration variables that are important to get the most out of `ansible-playbook`, but this should be enough to get you started running the playbooks in this chapter on your own servers or virtual machines.

The rest of this chapter uses more realistic Ansible playbooks. All the examples in this chapter can be found in Jeff Geerling's Ansible for DevOps GitHub repository[53], and you can clone that repository to your computer (or browse the code online) to follow along more easily.

Real-world playbook: CentOS Node.js app server

The first example, while being helpful for someone who might want to post a simple static web page to a clunky old Apache server, is not a good representation of a real-world scenario. I'm going to run through some more complex playbooks that do many different things, most of which are actually being used to manage production infrastructure today.

The first playbook will configure a CentOS server with Node.js, and install and start a simple Node.js application. The server will have a very simple architecture:



Node.js app on CentOS.

To start things off, we need to create a YAML file (`playbook.yml` in this example) to contain our playbook. Let's keep things simple:

[53] https://github.com/geerlingguy/ansible-for-devops

```
- hosts: all

  tasks:
```

First, define a set of hosts (`all`) on which this playbook will be run (see the section above about limiting the playbook to particular groups and hosts), then tell ansible that what follows will be a list of tasks to run on the hosts.

Add extra repositories

Adding extra package repositories (yum or apt) is one thing many admins will do before any other work on a server to ensure that certain packages are available, or are at a later version than the ones in the base installation.

In the shell script below, we want to add both the EPEL and Remi repositories, so we can get some packages like Node.js or later versions of other necessary software (these examples presume you're running RHEL/CentOS 6.x):

```
# Import EPEL GPG Key - see: https://fedoraproject.org/keys
wget https://fedoraproject.org/static/0608B895.txt \
  -O /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6

# Import Remi GPG key - see: http://rpms.famillecollet.com/RPM-GPG-KEY-remi
wget http://rpms.famillecollet.com/RPM-GPG-KEY-remi \
  -O /etc/pki/rpm-gpg/RPM-GPG-KEY-remi
rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-remi

# Install EPEL and Remi repos.
rpm -Uvh --quiet \
  http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
rpm -Uvh --quiet \
  http://rpms.famillecollet.com/enterprise/remi-release-6.rpm

# Install Node.js (npm plus all its dependencies).
yum --enablerepo=epel install node
```

This shell script uses the rpm command to import the EPEL and Remi repository GPG keys, then adds the repositories, and finally installs Node.js. It works okay for a simple deployment (or by hand), but it's silly to run all these commands (some of which could take time or stop your script entirely if your connection is flaky or bad) if the result has already been achieved (namely, two repositories and their GPG keys have been added).

If you wanted to skip a couple steps, you could skip adding the GPG keys, and just run your commands with `--nogpgcheck` (or, in Ansible, set the `disable_gpg_check` parameter of the yum module to `yes`), but it's a good idea to leave this enabled. GPG stands for *GNU Privacy Guard*, and it's a way that developers and package distributors can sign their packages (so you know it's from the original author, and hasn't been modified or corrupted). Unless you *really* know what you're doing, don't disable security settings like GPG key checks.

Ansible can make things a little more robust. Even though the following is slightly more verbose, it performs the same actions in a more structured way, which is simpler to understand, and can work with variables other nifty Ansible features we'll discuss later:

```
4  - name: Import EPEL and Remi GPG keys.
5    rpm_key: key={{ item }} state=present
6    with_items:
7    - "https://fedoraproject.org/static/0608B895.txt"
8    - "http://rpms.famillecollet.com/RPM-GPG-KEY-remi"
9
10 - name: Install EPEL and Remi repos.
11   command: rpm -Uvh --force {{ item.href }} creates={{ item.creates }}
12   with_items:
13   - {
14     href: "http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.no\
15 arch.rpm",
16     creates: "/etc/yum.repos.d/epel.repo"
17   }
18   - {
19     href: "http://rpms.famillecollet.com/enterprise/remi-release-6.rpm",
20     creates: "/etc/yum.repos.d/remi.repo"
21   }
22
23 - name: "Common: Disable firewall (since this is a dev environment)."
24   service: name=iptables state=stopped enabled=no
25
26 - name: Install Node.js (npm plus all it's dependencies).
27   yum: pkg=npm state=present enablerepo=epel
28
29 - name: "Node: Install forever module (to run our Node.js app)."
30   npm: name=forever global=yes state=latest
```

Let's walk through this playbook step-by-step:

1. `rpm_key` is a very simple Ansible module that takes and imports an RPM key from a URL or file, or the keyid of an key that is already present, and can ensure the key is either present or

absent (the `state` parameter). We want to import two keys—one for EPEL from the Fedora project, and one for the Remi Repository.
2. Since Ansible doesn't have a built-in rpm module, we simply use the rpm command, but we use Ansible's `command` module, which allows us to do two things:
    1. Use the `creates` parameter to tell Ansible when to *not* run the command (in this case, we tell Ansible what file is present after the `rpm` command successfully completes).
    2. Use an multidimensional array of items (`with_items`) so we can define URLs and the resulting file that can be checked with `creates`.
3. `yum` installs Node.js (along with all the required packages for `npm`, Node's package manager) if it's not present, and allows the EPEL repo to be searched via the `enablerepo` parameter (you could also explicitly *disable* a repository using `disablerepo`).
4. Since NPM is now installed, we can use Ansible's `npm` module to install a Node.js utility, `forever`, so we can easily launch our app and keep it running. Setting `global` to `yes` tells NPM to install the `forever` node module in `/usr/lib/node_modules/` so it will be available to all users and Node.js apps on the system.

We're beginning to have a nice little Node.js app server set up. Let's set up a little Node.js app that responds to HTTP requests on port 80.

Deploy a Node.js app

The next step is to install a simple Node.js app on our server. First, we'll create a really simple Node.js app by creating a new folder, `app`, in the same folder as your playbook.yml. Create a new file, `app.js`, in this folder, with the following contents:

```
1  // Load the express module.
2  var express = require('express'),
3  app = express.createServer();
4
5  // Respond to requests for / with 'Hello World'.
6  app.get('/', function(req, res){
7      res.send('Hello World!');
8  });
9
10 // Listen on port 80 (like a true web server).
11 app.listen(80);
12 console.log('Express server started successfully.');
```

Don't worry about the syntax or the fact that this is Node.js. We just need a quick example to deploy. This example could've been written in Python, Perl, Java, PHP, or another language, but since Node is a very simple language (JavaScript) that runs in a very simple and lightweight environment, it's a nice (and easy) language to use when testing things or prodding your server.

Since this little app is dependent on Express (a simple http framework for Node), we also need to tell NPM about this dependency via a `package.json` file in the same folder as `app.js`:

```
{
  "name": "examplenodeapp",
  "description": "Example Express Node.js app.",
  "author": "Jeff Geerling <geerlingguy@mac.com>",
  "dependencies": {
    "express": "3.x.x"
  },
  "engine": "node >= 0.10.6"
}
```

Now, add the following to your playbook, to copy the entire app to the server, and then have NPM download the required dependencies (in this case, `express`):

```
- name: "Node: Ensure Node.js app folder exists."
  file: path={{ node_apps_location }} state=directory

- name: "Node: Copy example Node.js app to server."
  copy: src=app dest={{ node_apps_location }}

- name: "Node: Install app dependencies defined in package.json via npm."
  npm: path={{ node_apps_location }}/app
```

First, we ensure the directory where our app will be installed exists, using the `file` module. The `{{ node_apps_location }}` variable used in each command can be defined under a `vars` section at the top of our playbook, in your inventory, or on the command line when calling `ansible-playbook`.

Second, we copy the entire app folder up to the server, using Ansible's `copy` command, which intelligently distinguishes between a single file or a directory of files, and recurses through the directory, similar to recursive scp or rsync.

Ansible's `copy` module works very well for single or small groups of files, and recurses through directories automatically. If you are copying hundreds of files, or deeply-nested directory structures, `copy` will get bogged down. In these situations, consider using the `synchronize` module if you need to copy a full directory, or `unarchive` if you want to copy up an archive and have it expanded in place on the server.

Third, we use `npm` again, this time, with no extra arguments besides the path to the app. This tells NPM to parse the package.json file and ensure all the dependencies are present.

We're *almost* finished! The last step is to start the app.

Launch a Node.js app

We'll now use `forever` (which we installed earlier) to start the app.

```
39 - name: "Node: Check list of Node.js apps running."
40   command: forever list
41   register: forever_list
42   changed_when: false
43
44 - name: "Node: Start example Node.js app."
45   command: forever start {{ node_apps_location }}/app/app.js
46   when: "forever_list.stdout.find('{{ node_apps_location}}/app/app.js') == -1"
```

In the first play, we're doing two new things:

1. `register` creates a new variable, `forever_list`, to be used in the next play to determine when to run the play. `register` stashes the output (stdout, stderr) of the defined command in the variable name passed to it.
2. `changed_when` tells Ansible explicitly when this play results in a change to the server. In this case, we know the `forever list` command will never change the server, so we just say `false`—the server will never be changed when the command is run.

The second play actually starts the app, using forever. We could also start the app by calling `node {{ node_apps_location }}/app/app.js`, but we would not be able to control the process easily, and we would also need to use `nohup` and `&` to avoid Ansible hanging on this play.

Forever tracks the Node apps it manages, and we use Forever's `list` option to print a list of running apps. The first time we run this playbook, the list will obviously be empty—but on future runs, if the app is running, we don't want to start another instance of it. To avoid that situation, we tell ansible when we want to start the app with `when`. Specifically, we tell Ansible to start the app only when the app's path in *not* in the `forever list` output.

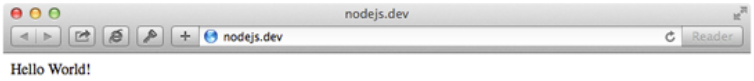
Node.js app server summary

At this point, you have a complete playbook that will install a simple Node.js app which responds to HTTP requests on port 80 with "Hello World!".

To run the playbook on a server (in our case, we could just set up a new VirtualBox VM for testing, either via Vagrant or manually), use the following command (pass in the `node_apps_location` variable via the command):

```
$ ansible-playbook playbook.yml \
--extra-vars="node_apps_location=/usr/local/opt/node"
```

Once the playbook has finished configuring the server and deploying your app, visit `http://hostname/` in a browser (or use `curl` or `wget` to request the site), and you should see the following:



Node.js Application home page.

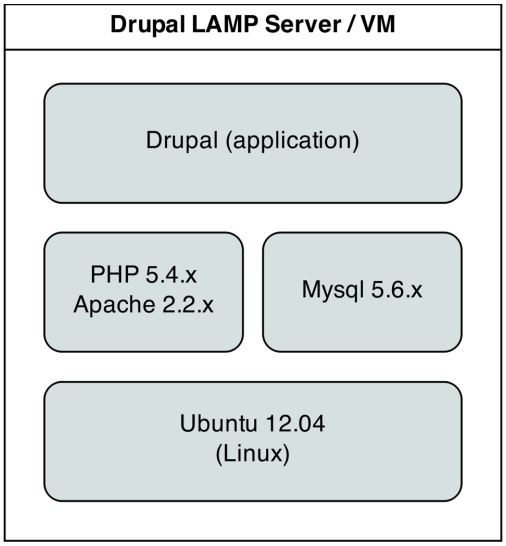
Simple, but very powerful. We've configured an entire Node.js application server In fewer than fifty lines of YAML!

You can find the entire example Node.js app server playbook in this book's code repository at https://github.com/geerlingguy/ansible-for-devops[54], in the `nodejs` directory.

Real-world playbook: Ubuntu LAMP server with Drupal

At this point, you should be getting comfortable with Ansible playbooks and the YAML syntax used to define them. Up to this point, most examples have assumed you're working with a CentOS, RHEL, or Fedora server. Ansible plays nicely with other flavors of Linux and BSD-like systems as well. In the following example, we're going to set up a traditional LAMP (Linux, Apache, MySQL, and PHP) server using Ubuntu 12.04 to run a Drupal website.

[54] https://github.com/geerlingguy/ansible-for-devops



Drupal LAMP server.

Include a variables file, and discover `pre_tasks` and `handlers`

For this playbook, we're going to start organizing our playbook a little more efficiently. Instead of defining any requiring variables be passed in via the command line, let's begin the playbook by telling Ansible our variables are stored in a separate `vars.yml` file:

```
1  - hosts: all
2
3    vars_files:
4    - vars.yml
```

Using one or more variable files, rather than defining everything inline, cleans up your main playbook file, and lets you organize all your configurable variables in one place. For now, we don't have any variables to add; we'll define the contents of `vars.yml` later. For now, create the empty file, and continue on to the next section of the playbook, `pre_tasks`:

```
5    pre_tasks:
6    - name: Update apt cache if needed.
7      apt: update_cache=yes cache_valid_time=3600
```

Ansible lets you run plays before or after the main set of plays using `pre_tasks` and `post_tasks`. In this case, we need to ensure that our apt cache is updated before we run the rest of the playbook, so we have the latest package versions on our server. We use Ansible's `apt` module and simply tell it to update the cache if it's been more than 3600 seconds (1 hour) since the last update.

With that out of the way, we'll add another new section to our playbook, `handlers`:

```
8   handlers:
9   - name: restart apache
10    service: name=apache2 state=restarted
```

`handlers` are special kinds of plays that you can run at the end of a group of plays by adding the `notify` option to any of the plays in that group. The handler will only be called if one of the plays notifying the handler makes a change to the server (and doesn't fail), and it will only be notified at the *end* of the group of plays.

To call this handler, you simply add the option `notify: restart apache` after defining the rest of a play. We've defined this handler so we can restart the `apache2` service after a configuration change, which will be explained below.

Just like variables, handlers and plays may be placed in separate files and included in your playbook to keep things tidy (we'll discuss this in chapter 6). For simplicity's sake, though, the examples in this chapter are shown as in a single playbook file. We'll discuss different playbook organization methods later.

By default, Ansible will stop all playbook execution when a task fails, and won't even notify any handlers that may need to be triggered. In some cases, this can lead to unintended side effects. If you want to make sure handlers always run after a task uses `notify` to call the handler, even in case of playbook failure, add `--force-handlers` to your `ansible-playbook` command.

Basic LAMP server setup

The first step towards building an application server that depends on the LAMP stack is to build the actual LAMP part of it. This is the simplest process, but still requires a little extra work for our particular server. We want to install Apache, MySQL and PHP, but we'll also need a couple other dependencies, and we want a particular version of PHP (5.5), which is only available in an extra apt repository.

```
  tasks:
  - name: "Common: Get software for apt repository management."
    apt: pkg={{ item }} state=installed
    with_items:
    - python-apt
    - python-pycurl

  - name: "Common: Add ondrej repository for later versions of PHP."
    # Note: You can remove '-oldstable' for PHP 5.5.x.
    apt_repository: repo='ppa:ondrej/php5-oldstable'

  - name: "Common: Install Apache, MySQL, PHP, and other dependencies."
    apt: pkg={{ item }} state=installed
    with_items:
    - git
    - curl
    - sendmail
    - apache2
    - php5
    - php5-common
    - php5-mysql
    - php5-cli
    - php5-curl
    - php5-gd
    - php5-dev
    - php5-mcrypt
    - php-apc
    - php-pear
    - python-mysqldb
    - mysql-server

  - name: "Common: Disable the firewall (since this is for local dev only)."
    service: name=ufw state=stopped

  - name: "Common: Start Apache, MySQL, and PHP."
    service: name={{ item }} state=started enabled=yes
    with_items:
    - apache2
    - mysql
```

In this playbook, I've decided to add a simple prefix to each named play, so I can more easily follow the playbook's progress when it's running. I've begun with the common LAMP setup:

1. Install a couple helper libraries which allow Python to manage apt more precisely (`python-apt` and `python-pycurl` are required for the `apt_repository` module to do its work).
2. Since the default apt repositories for Ubuntu 12.04 don't include PHP 5.4.x (or any later versions), install ondrej's `PHP5-oldstable` repository, containing PHP 5.4.25 (at the time of this writing) and other associated PHP packages.
3. Install all the required packages for our LAMP server (including all the php5 extensions we need to run Drupal).
4. Disable the firewall entirely, for testing purposes. If on a production server or any server exposed to the Internet, you should instead have a restrictive firewall only allowing access on ports 22, 80, 443, and other necessary ports.
5. Start up all the required services, and make sure they're enabled to start on system boot.

Configure Apache

The next step is configuring Apache so it will work correctly with Drupal. Out of the box, Apache doesn't have mod_rewrite enabled on Ubuntu 12.04. To remedy that situation, you can use the command `sudo a2enmod rewrite`, but Ansible has a handy `apache2_module` module that will simplify the task.

Additionally, we need to add a VirtualHost entry to tell Apache where the site's document root is, and any other options for the site.

```
50    - name: "Apache: Enable Apache rewrite module (required for Drupal)."
51      apache2_module: name=rewrite state=present
52      notify: restart apache
53
54    - name: "Apache: Add Apache virtualhost for Drupal 8 development."
55      template: >
56        src=templates/drupal.dev.conf.j2
57        dest=/etc/apache2/sites-available/{{ domain }}.dev.conf
58        owner=root group=root mode=644
59      notify: restart apache
60
61    - name: "Apache: Symlink Drupal virtualhost to sites-enabled."
62      file: >
63        src=/etc/apache2/sites-available/{{ domain }}.dev.conf
64        dest=/etc/apache2/sites-enabled/{{ domain }}.dev.conf
65        state=link
66      notify: restart apache
67
68    - name: "Apache: Remove default virtualhost file."
69      file: >
```

```
70        path=/etc/apache2/sites-enabled/000-default
71        state=absent
72      notify: restart apache
```

The first command enables all the required Apache modules by symlinking them from `/etc/apache2/mods-availab` to `/etc/apache2/mods-enabled`.

The second command copies a Jinja2 template we define inside the templates folder to Apache's `sites-available` folder, with the correct owner and permissions. Additionally, we `notify` the `restart apache` handler, because copying in a new VirtualHost means Apache needs to be restarted to pick up the change.

Let's look at our Jinja2 template (denoted by the extra `.j2` on the end of the filename), `drupal.dev.conf.j2`:

```
1  <VirtualHost *:80>
2      ServerAdmin webmaster@localhost
3      ServerName {{ domain }}.dev
4      ServerAlias www.{{ domain }}.dev
5      DocumentRoot {{ drupal_core_path }}
6      <Directory "{{ drupal_core_path }}">
7          Options FollowSymLinks Indexes
8          AllowOverride All
9      </Directory>
10 </VirtualHost>
```

This is a fairly standard Apache VirtualHost definition, but we have a few Jinja2 template variables mixed in. The syntax for printing a variable in a Jinja2 template is the same syntax we use in our Ansible playbooks—two brackets around the variable's name (like so: `{{ variable }}`).

There are three variables we will need (`drupal_core_version`, `drupal_core_path`, and `domain`), so we can add them to the empty `vars.yml` file we created earlier:

```
1  ---
2  # The core version you want to use (e.g. 6.x, 7.x, 8.0.x).
3  drupal_core_version: "8.0.x"
4
5  # The path where Drupal will be downloaded and installed.
6  drupal_core_path: "/var/www/drupal-{{ drupal_core_version }}-dev"
7
8  # The resulting domain will be [domain].dev (with .dev appended).
9  domain: "drupaltest"
```

Now, when Ansible reaches the play that copies this template into place, the Jinja2 template will have the variable names replaced with the values `8.0.x` and `drupaltest` (or whatever values you'd like!).

The last two plays (lines 12-19) enable the VirtualHost we just added, and remove the default VirtualHost definition, which we no longer need.

At this point, you could start the server, but Apache will likely throw an error since the VirtualHost you've defined doesn't yet exist (there's no directory at `{{ drupal_core_path }}` yet!). This is why using `notify` is important—instead of adding a play after these three steps to restart Apache, which will fail the first time you run the playbook, notify will wait until after we've finished all the other steps in our main group of plays (giving us time to finish setting up the server), *then* restart Apache.

Configure PHP with `lineinfile`

We briefly mentioned `lineinfile` earlier in the book, when discussing file management and ad-hoc task execution. Modifying PHP's configuration is a perfect way to demonstrate `lineinfile`'s simplicity and usefulness:

```
74    - name: "PHP: Enable upload progress via APC."
75      lineinfile: >
76        dest=/etc/php5/conf.d/20-apc.ini
77        regexp="^apc\.rfc1867"
78        line="apc.rfc1867 = 1"
79        state=present
80      notify: restart apache
```

Ansible's `lineinfile` module does a simple task: ensures that a particular line of text exists (or doesn't exist) in a file.

In this example, we need to enable APC's `rfc1867` option so Drupal can use APC's file upload progress tracking (there are better ways of doing this, but for our simple server, this will suffice).

First, we tell `lineinfile` the location of the file, in the `dest` parameter. Then, we give a regular expression (Python-style) to define what the line looks like (in this case, the line starts with the exact phrase "apc.rfc1867"—we had to escape the period since it is a special character in regular expressions). Next, we tell `lineinfile` exactly how the resulting line should look. Finally, we explicitly state that we want this line to be present (with the `state` parameter).

Ansible will take the regular expression, and see if there's a matching line. If there is, Ansible will make sure the line matches the `line` parameter. If not, Ansible will add the line as defined in the `line` parameter. Ansible will only report a change if it had to add or change the line to match `line`.

Configure MySQL

The next step is to remove MySQL's default test database, and create a database (named for the domain we specified earlier) for our Drupal installation to use.

```
80    - name: "MySQL: Remove the MySQL test database."
81      mysql_db: db=test state=absent
82
83    - name: "MySQL: Create a database for Drupal."
84      mysql_db: db={{ domain }} state=present
```

MySQL installs a database named `test` by default, and it is recommended that you remove the database as part of MySQL's included `mysql_secure_installation` tool. The first step in configuring MySQL is removing this database. Next, we create a database named `{{ domain }}`—the database is named the same as the domain we're using for the Drupal site.

Ansible works with many databases out of the box (MongoDB, MySQL, PostgreSQL, Redis and Riak as of this writing). In MySQL's case, Ansible uses the MySQLdb Python package (`python-mysqldb`) to manage a connection to the database server, and assumes the default root account credentials ('root' as the username with no password). Obviously, leaving this default would be a bad idea! On a production server, one of the first steps should be to change the root account password, limit the root account to localhost, and delete any nonessential database users.

If you use different credentials, you can add a .my.cnf file to your remote user's home directory containing the database credentials so Ansible can connect to the MySQL database without leaving passwords in your Ansible playbooks or variable files. Otherwise, you can prompt the user running the Ansible playbook for a MySQL username and password. This option, using prompts, will be discussed later in the book.

Install Composer and Drush

Drupal has a command-line companion in the form of Drush. Drush is developed independently of Drupal, and provides a full suite of CLI commands to manage Drupal. Drush, like most modern PHP tools, integrates with external dependencies defined in a `composer.json` file which describes the dependencies to Composer.

We could just download Drupal and perform some setup in the browser by hand at this point, but the goal of this playbook is to have a fully-automated and idempotent Drupal installation. So, we need to install Composer, then Drush:

```
85    - name: "Composer: Install Composer into the current directory."
86      shell: >
87        curl -sS https://getcomposer.org/installer | php
88        creates=/usr/local/bin/composer
89
90    - name: "Composer: Move Composer into globally-accessible location."
91      shell: >
92        mv composer.phar /usr/local/bin/composer
93        creates=/usr/local/bin/composer
```

The first command runs Composer's php-based installer, which generates a 'composer.phar' PHP application archive. This archive is then copied (using the `mv` shell command) to the location `/usr/local/bin/composer` so we can use the simple `composer` command to install all of Drush's dependencies. Both commands are set to only run if the `/usr/local/bin/composer` file doesn't already exist (using the `creates` parameter).

Why use `shell` instead of `command`? Ansible's `command` module is the preferred option for running commands on a host (when an Ansible module won't suffice), and it works in most scenarios. However, `command` doesn't run the command via the remote shell `/bin/sh`, so options like `<`, `>`, `|`, and `&`, and local environment variables like `$HOME` won't work. `shell` allows you to pipe command output to other commands, access the local environment, etc.

There are two other modules which assist in executing shell commands remotely: `script` executes shell scripts (though it's almost always a better idea to convert shell scripts into idempotent Ansible playbooks!), and `raw` executes raw commands via SSH (it should only be used in circumstances where you can't use one of the other options).

It's best to use an Ansible module for every task. If you have to resort to a regular command-line command, try the the `command` module first. If you require the options mentioned above, use `shell`. Use of `script` or `raw` should be exceedingly rare, and won't be covered in this book.

Now, we'll install Drush using the latest version from GitHub:

```
94    - name: "Drush: Check out drush master branch."
95      git: repo=https://github.com/drush-ops/drush.git dest=/opt/drush
96
97    - name: "Install Drush dependencies with Composer."
98      shell: >
99        /usr/local/bin/composer install
100       chdir=/opt/drush
101       creates=/opt/drush/vendor/autoload.php
102
103   - name: "Drush: Create drush bin symlink."
```

```
104     file: >
105       src=/opt/drush/drush
106       dest=/usr/local/bin/drush
107       state=link
```

Earlier in the book, we cloned a git repository using an ad-hoc command. In this case, we're defining a play that uses the `git` module to clone Drush from its repository URL on GitHub. Since we want the master branch, we simply pass in the `repo` (repository URL) and `dest` (destination path) parameters.

After drush is downloaded to `/opt/drush`, we use Composer to install all the required dependencies. In this case, we want Ansible to run `composer install` in the directory `/opt/drush` (this is so Composer can find drush's `composer.json` file automatically), so we pass along the parameter `chdir=/opt/drush`. Once Composer is finished, the file `/opt/drush/vendor/autoload.php` will be created, so we use the `creates` parameter to tell Ansible to skip this step if the file already exists (for idempotency).

Finally, we create a symlink from `/usr/local/bin/drush` to the executable at `/opt/drush/drush`, so we can call the `drush` command anywhere on the system.

Install Drupal with Git and Drush

We'll use `git` again to clone Drupal to the apache document root we defined earlier in our virtual host configuration, then we'll run Drupal's installation via drush, and fix a couple other file permissions issues so Drupal loads correctly within our VM.

```
108   - name: "Drupal: Check out Drupal Core to the Apache docroot."
109     git: >
110       repo=http://git.drupal.org/project/drupal.git
111       version={{ drupal_core_version }}
112       dest={{ drupal_core_path }}
113
114   - name: "Drupal: Install Drupal."
115     command: >
116       drush si -y --site-name="{{ drupal_site_name }}" --account-name=admin
117       --account-pass=admin --db-url=mysql://root@localhost/{{ domain }}
118       chdir={{ drupal_core_path }}
119       creates={{ drupal_core_path }}/sites/default/settings.php
120     notify: restart apache
121
122   # SEE: https://drupal.org/node/2121849#comment-8413637
123   - name: "Drupal: Set permissions properly on settings.php."
124     file: >
125       path={{ drupal_core_path }}/sites/default/settings.php
```

```
126       mode=744
127
128   - name: "Drupal: Set permissions properly on files directory."
129     file: >
130       path={{ drupal_core_path }}/sites/default/files
131       mode=777
132       state=directory
133       recurse=yes
```

First, we cloned Drupal's git repository, using the `version` defined in our `vars.yml` file as `drupal_core_version`. The `git` module's `version` parameter defines the branch (`master`, `8.0.x`, etc.), tag (`1.0.1`, `7.24`, etc.), or individual commit hash (`50a1877`, etc.) to clone.

Next, we used Drush's `si` command (short for `site-install`) to run Drupal's installation (which configures the database, runs some maintenance, and sets some default configuration settings for the site). We passed in a few variables, like the `drupal_core_version` and `domain`; we also added a `drupal_site_name`, so add that variable to your `vars.yml` file:

```
10  # Your Drupal site name.
11  drupal_site_name: "D8 Test"
```

Also, Drupal's installation process results in the creation of a 'settings.php' file, so we use the location of that file with the `creates` parameter to let Ansible know if the site's already installed (so we don't accidentally try installing it again!). Once the site is installed, we also restart Apache for good measure (using `notify` again, like we did when updating Apache's configuration).

The final two tasks set permissions on Drupal's settings.php and files folder to `744` and `777`, respectively.

Drupal LAMP server summary

At this point, if you access the server at http://drupaltest.dev/ (assuming you've pointed drupaltest.dev to your server or VM's IP address), you'll see Drupal's default home page, and you could login with 'admin'/'admin'. (Obviously, you'd set a secure password on a production server!).

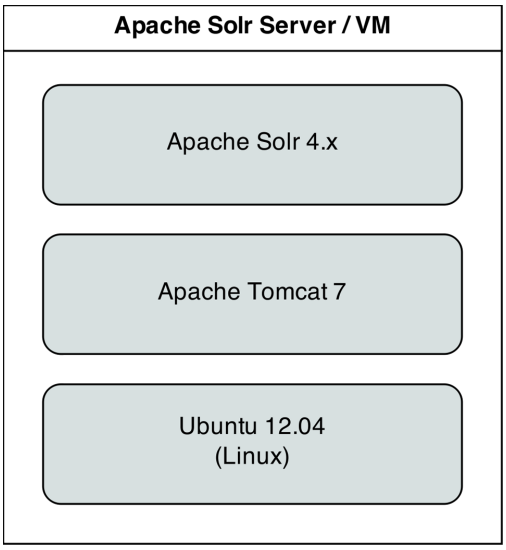
A similar server configuration, running Apache, MySQL, and PHP, can be used to run many popular web frameworks and CMSes besides Drupal, including Symfony, Wordpress, Joomla, Laravel, etc.

You can find the entire example Drupal LAMP server playbook in this book's code repository at https://github.com/geerlingguy/ansible-for-devops[55], in the `drupal` directory.

[55] https://github.com/geerlingguy/ansible-for-devops

Real-world playbook: Ubuntu Apache Tomcat server with Solr

Apache Solr is a fast and scalable search server optimized for full-text search, word highlighting, faceted search, fast indexing, and more. It's a very popular search server, and it's pretty easy to install and configure using Ansible. In the following example, we're going to set up Apache Solr using Ubuntu 12.04 and Apache Tomcat.



Apache Solr Server.

Include a variables file, and discover `pre_tasks` and `handlers`

Just like the previous LAMP server example, we'll begin this playbook by telling Ansible our variables will be in a separate `vars.yml` file:

```
1  - hosts: all
2
3    vars_files:
4    - vars.yml
```

Let's quickly create the `vars.yml` file, while we're thinking about it. Create the file in the same folder as your Solr playbook, and add the following contents:

```
1  download_dir: /tmp
2  solr_dir: /opt/solr
```

These two variables define two paths we'll use while downloading and installing Apache Solr.

Back in our playbook, after the `vars_files`, we also need to make sure the apt cache is up to date, using `pre_tasks` like the previous example:

```
5   pre_tasks:
6   - name: Update apt cache if needed.
7     apt: update_cache=yes cache_valid_time=3600
```

Like the Drupal playbook, we again use `handlers` to define certain tasks that are notified by tasks in the `tasks` section. This time, we just need a handler to restart `tomcat7`, the Java servlet container that powers Apache Solr:

```
8   handlers:
9   - name: restart tomcat
10    service: name=tomcat7 state=restarted
```

We can call this handler with the option `notify: restart tomcat` in any play in our playbook.

Install Apache Tomcat 7

It's easy enough to install Tomcat 7 on an Ubuntu Precise server; there are packages in the default apt repositories, so we just need to make sure they're installed, and that the `tomcat7` service is enabled and started:

```
11  tasks:
12  - name: Install Tomcat 7.
13    apt: "pkg={{ item }} state=installed"
14    with_items:
15      - tomcat7
16      - tomcat7-admin
17
18  - name: Ensure Tomcat 7 is started and enabled on boot.
19    service: name=tomcat7 state=started enabled=yes
```

That was easy enough! We used the `apt` module to install two packages, `tomcat7` and `tomcat7-admin` (so we can log into Tomcat's administrative backend), then started `tomcat7` and set it to start when the system boots.

Install Apache Solr

Ubuntu 12.04 includes a package for Apache Solr, but it installs a very old version, so we'll install the latest version of Solr from source. The first step is downloading the source:

```
20 - name: Download Solr.
21   get_url: >
22     url=http://apache.osuosl.org/lucene/solr/4.7.1/solr-4.7.1.tgz
23     dest={{ download_dir }}/solr-4.7.1.tgz
24     sha256sum=4a546369a31d34b15bc4b99188984716bf4c0c158c0e337f3c1f98088aec70ee
```

We're installing Apache Solr 4.7.1, the latest version at the time of this book's writing. When downloading files from remote servers, the `get_url` module provides more flexibility and convenience than raw `wget` or `curl` commands.

You have to pass `get_url` a `url` (the source of the file to be downloaded), and a `dest` (the location where the file will be downloaded). If you pass a directory to the `dest` parameter, Ansible will place the file inside, but will always re-download the file on subsequent runs of the playbook (and overwrite the existing download if it has changed). To avoid this extra overhead, we give the full path to the downloaded file.

We also use `sha256sum`, an optional parameter, for peace of mind; if you are downloading a file or archive that's critical to the functionality and security of your application, it's a good idea to check the file to make sure it is exactly what you're expecting. `sha256sum` compares a hash of the data in the downloaded file to a 256-bit hash that you specify (use `shasum -a 256 /path/to/file` to get the `sha256sum` of a file). If the checksum doesn't match the supplied hash, Ansible will fail and discard the freshly-downloaded (and invalid) file.

```
25 - name: Expand Solr.
26   command: >
27     tar -C /tmp -xvzf {{ download_dir }}/solr-4.7.1.tgz
28     creates={{ download_dir }}/solr-4.7.1/dist/solr-4.7.1.war
29
30 - name: Copy Solr into place.
31   command: >
32     cp -r {{ download_dir }}/solr-4.7.1 {{ solr_dir }}
33     creates={{ solr_dir }}/dist/solr-4.7.1.war
```

We need to expand the Apache Solr archive, then copy it into place. For both of these steps, use the built-in `tar` and `cp` utilities (with the appropriate options) to do the work. Setting `creates` tells Ansible to skip these steps in subsequent runs, since the Solr war file will already be in place.

```
34 # Use shell so commands are passed in correctly.
35 - name: Copy Solr components into place.
36   shell: >
37     cp -r {{ item.src }} {{ item.dest }}
38     creates={{ item.creates }}
39   with_items:
40     # Solr example configuration and war file.
41     - {
42       src: "{{ solr_dir }}/example/webapps/solr.war",
43       dest: "{{ solr_dir }}/solr.war",
44       creates: "{{ solr_dir }}/solr.war"
45     }
46     - {
47       src: "{{ solr_dir }}/example/solr/*",
48       dest: "{{ solr_dir }}/",
49       creates: "{{ solr_dir }}/solr.xml"
50     }
51     # Solr log4j logging configuration.
52     - {
53       src: "{{ solr_dir }}/example/lib/ext/*",
54       dest: "/var/lib/tomcat7/shared/",
55       creates: "/var/lib/tomcat7/shared/log4j-1.2.16.jar"
56     }
57     - {
58       src: "{{ solr_dir }}/example/resources/log4j.properties",
59       dest: "/var/lib/tomcat7/shared/classes",
60       creates: "/var/lib/tomcat7/shared/classes/log4j.properties"
61     }
62   notify: restart tomcat
```

The next task copies into place certain directories and files required to run Apache Solr.

Nothing too special here, but this example illustrates how you can use comments within `with_items` lists to help clarify the items in the list. We could've added each command as its own task, but doing it this way reduces the total number of Ansible tasks and allows us to move the `with_items` list to an external variable if desired.

```
63 - name: Ensure solr example directory is absent.
64   file: >
65     path={{ solr_dir }}/example
66     state=absent
67
68 - name: Set up solr data directory.
69   file: >
70     path={{ solr_dir }}/data
71     state=directory
72     owner=tomcat7 group=tomcat7
```

The latest version of Apache Solr searches through all the directories inside `{{ solr_dir }}` recursively, loading any potential search configuration it finds. Since we copied over one of the examples to use as the server's default search core, Solr would see it as a duplicate of one of the examples and crash. So, we can use the `file` module with a `path` to the example directory to make sure the directory is gone (`state=absent`).

After removing the example directory (and in future runs, ensuring it's still gone), we set up the data directory where Solr will store index data, ensuring it exists as a directory, and is owned by the `tomcat7` user and group.

```
73 - name: Configure solrconfig.xml for new data directory.
74   lineinfile: >
75     dest={{ solr_dir }}/collection1/conf/solrconfig.xml
76     regexp="^.*<dataDir.+$"
77     line="<dataDir>${solr.data.dir:{{ solr_dir }}/data}</dataDir>"
78     state=present
```

As we found earlier, `lineinfile` is a helpful module for ensuring consistent configuration file settings with idempotence. In this case, we need to make sure the `<dataDir>` line in our default search core's configuration file is set to a specific value.

```
79 - name: Set permissions for solr home.
80   file: >
81     path={{ solr_dir }}
82     recurse=yes
83     owner=tomcat7 group=tomcat7
```

To set ownership options on the entire contents of the `{{ solr_dir }}` correctly, we use the `file` module with the `recurse` parameter set to `yes`. This is equivalent to the shell command `chown -R tomcat7:tomcat7 {{ solr_dir }}`.

```
- name: Add Catalina configuration for solr.
  template: >
    src=templates/solr.xml.j2
    dest=/etc/tomcat7/Catalina/localhost/solr.xml
    owner=root group=tomcat7 mode=644
  notify: restart tomcat
```

The final task copies a template file (`solr.xml.j2`) to the remote host, substituting variables via Jinja2 syntax, and sets the file's ownership and permissions as needed for Tomcat.

Before the task can run, the local template file will need to be created. Create a 'templates' folder in the same directory as your Apache Solr playbook, and create a new file named `solr.xml.j2` inside, with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<Context docBase="{{ solr_dir }}/solr.war" debug="0" crossContext="true">
  <Environment name="solr/home" type="java.lang.String" \
  value="{{ solr_dir }}" override="true"/>
</Context>
```

You can run the playbook with `$ ansible-playbook [playbook-name.yml]`, and after a few minutes (depending on your server's Internet connection speed), you should be able to access the Solr admin interface at http://example.com:8080/solr (where 'example.com' is your server's hostname or IP address).

Apache Solr server summary

The configuration we used when deploying Apache Solr allows for a multicore setup, so you could add more 'search cores' via the admin interface (as long as the directories and core schema configuration is in place in the filesystem), and have multiple indexes for multiple websites and applications.

A playbook similar to the one above is used as part of the infrastructure for Hosted Apache Solr[56], a service run by the author which runs a hosted version of Apache Solr particularly for Drupal search indexes.

You can find the entire example Apache Solr server playbook in this book's code repository at https://github.com/geerlingguy/ansible-for-devops[57], in the `solr` directory.

[56] http://hostedapachesolr.com/

[57] https://github.com/geerlingguy/ansible-for-devops

Summary

At this point, you should be getting comfortable with Ansible's *modus operandi*. Playbooks are the heart of Ansible's configuration management and provisioning functionality, and the same modules and similar syntax can be used with ad-hoc commands for deployments and general server management.

Now that you're familiar with playbooks, we'll explore more advanced concepts in building playbooks, like organization of plays, conditionals, variables, and more. Later, we'll explore how we can use playbooks within roles to make them infinitely more flexible, and save time setting up and configuring your infrastructure.

```
 ________________________________________
/ If everything is under control, you are \
\ going too slow. (Mario Andretti)        /
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Chapter 5 - Ansible Playbooks - Beyond the Basics

The playbooks and simple playbook organization we used in the previous chapter cover many common use cases, but when discussing the breadth of system administration needs, there are thousands more little features of Ansible that you need to know.

We'll cover things like how to run plays with more granularity, how to organize your plays and playbooks for simplicity and usability, and other advanced playbook topics that will help you manage your infrastructure with even more confidence.

Handlers

TODO:

- Emphasize more advanced scenarios (handlers calling other handlers, notify multiple handlers per task, etc.).

Environment variables

Ansible allows you to work with environment variables in a variety of ways. First of all, if you simply need to set some environment variables for your remote user account, you can do that by adding lines to the remote user's `.bash_profile`, like so:

```
- name: Add an environment variable to the remote user's shell.
  lineinfile: dest=~/.bash_profile regexp=^ENV_VAR= line=ENV_VAR=value
```

All subsequent plays would then have access to this environment variable (remember, of course, only the `shell` module will understand shell commands that use environment variables!). To use an environment variable in further plays, it's recommended you use a play's `register` option to store the environment variable in a variable Ansible can use later, for example:

```
1 - name: Add an environment variable to the remote user's shell.
2   lineinfile: dest=~/.bash_profile regexp=^ENV_VAR= line=ENV_VAR=value
3
4 - name: Get the value of the environment variable we just added.
5   shell: 'source ~/.bash_profile && echo $ENV_VAR'
6   register: foo
7
8 - name: Print the value of the environment variable.
9   debug: msg="The variable is {{ foo.stdout }}"
```

We use `source ~/.bash_profile` in line 4 because Ansible needs to make sure it's using the latest environment configuration for the remote user. In some situations, the plays all run over a persistent or quasi-cached SSH session, over which `$ENV_VAR` wouldn't yet be defined.

(This is also the first time the `debug` module has made an appearance. It will be explored more in-depth along with other debugging techniques later.).

Why `~/.bash_profile`? There are many different places you can store environment variables, including `.bashrc`, `.profile`, and `.bash_profile` in a user's home folder. In our case, since we want the environment variable to be available to Ansible, which runs a pseudo-TTY shell session, in which case `.bash_profile` is used to configure the environment. You can read more about shell session configuration and these dotfiles here: Configuring your login sessions with dotfiles[58].

Linux will also read global environment variables added to `/etc/environment`, so you can add your variable there:

```
- name: Add a global environment variable.
  lineinfile: dest=/etc/environment regexp=^ENV_VAR= line=ENV_VAR=value
  sudo: yes
```

In any case, it's pretty simple to manage environment variables on the server with `lineinfile`. If your application requires many environment variables (as is the case in many Java applications), you might consider using `copy` or `template` with a local file instead of using `lineinfile` with a large list of items.

Per-play environment variables

You can also set the environment for just one play, using the `environment` option for that play. As an example, let's say you need to set an http proxy for a certain file download. This can be done with:

[58] http://mywiki.wooledge.org/DotFiles

```
- name: Download a file, using example-proxy as a proxy.
  get_url: url=http://www.example.com/file.tar.gz dest=~/Downloads/
  environment:
    http_proxy: http://example-proxy:80/
```

That could be rather cumbersome, though, especially if you have many plays that require a proxy or some other environment variable. In this case, you can pass an environment in via a variable in your playbook's `vars` section (or via an included variables file), like so:

```
vars:
  var_proxy:
    http_proxy: http://example-proxy:80/
    https_proxy: https://example-proxy:443/
    [etc...]

tasks:
- name: Download a file, using example-proxy as a proxy.
  get_url: url=http://www.example.com/file.tar.gz dest=~/Downloads/
  environment: var_proxy
```

If a proxy needs to be set system-wide (as is the case behind many corporate firewalls), I like to do so using the global `/etc/environment` file:

```
# In the 'vars' section of the playbook (set to 'absent' to disable proxy):
proxy_state: present

# In the 'tasks' section of the playbook:
- name: Configure the proxy.
  lineinfile: >
    dest=/etc/environment
    regexp={{ item.regexp }}
    line={{ item.line }}
    state={{ proxy_state }}
  with_items:
  - { regexp: "^http_proxy=", line: "http_proxy=http://example-proxy:80/" }
  - { regexp: "^https_proxy=", line: "https_proxy=https://example-proxy:443/" }
  - { regexp: "^ftp_proxy=", line: "ftp_proxy=http://example-proxy:80/" }
```

Doing it this way allows me to configure whether the proxy is enabled per-server (using the `proxy_state` variable), and with one play, set the http, https, and ftp proxies. You can use a similar kind of play for any other types of environment variables you need to set system-wide.

You can test remote environment variables using the `ansible` command: `ansible test -m shell -a 'echo $TEST'`. When doing so, be careful with your use of quotes and escaping—you might end up using double quotes where you meant to use single quotes, or vice-versa, and end up printing a local environment variable instead of one from the remote server!

Variables

Variables in Ansible work just like variables in most other systems. Variables always begin with a letter (`[A-Za-z]`), and can include any number of underscores (_) or numbers (`[0-9]`).

Valid variable names include `foo`, `foo_bar`, `foo_bar_5`, and `fooBar`, though the standard is to use all lowercase letters, and typically avoid numbers in variable names (no `camelCase` or `UpperCamelCase`).

Invalid variable names include `_foo`, `foo-bar`, `5_foo_bar`, `foo.bar` and `foo bar`.

In an inventory file, a variable's value is assigned using an equals sign, like so:

```
foo=bar
```

In a playbook or variables include file, a variable's value is assigned using a colon, like so:

```
foo: bar
```

Playbook Variables

There are many different ways you can define variables to use in plays.

Variables can be passed in via the command line, when calling `ansible-playbook`, with the `--extra-vars` option:

```
ansible-playbook example.yml --extra-vars "foo=bar"
```

You can also pass in extra variables using quoted JSON, YAML, or even by passing a JSON or YAML file directly, like `--extra-vars "@even_more_vars.json"` or `--extra-vars "@even_more_vars.yml`, but at this point, you might be better off using one of the other methods below.

Variables may be included inline with the rest of a playbook, in a `vars` section:

```
---
- hosts: example
  vars:
    foo: bar
  tasks:
    # Prints "Variable 'foo' is set to bar".
    - debug: msg="Variable 'foo' is set to {{ foo }}"
```

Variables may also be included in a separate file, using the `vars_files` section:

```
---
# Main playbook file.
- hosts: example
  vars_files:
    - vars.yml
  tasks:
    - debug: msg="Variable 'foo' is set to {{ foo }}"
```

```
---
# Variables file 'vars.yml' in the same folder as the playbook.
foo: bar
```

Notice how the variables are all at the root level of the YAML file. They don't need to be under any kind of `vars` heading when they are included as a standalone file.

Variable files can also be imported conditionally. Say, for instance, you have one set of variables for your CentOS servers (where the Apache service is named `httpd`), and another for your Debian servers (where the Apache service is named `apache2`). In this case, you could use a conditional `vars_files` include:

```
---
- hosts: example
  vars_files:
    - [ "apache_{{ ansible_os_family }}.yml", "apache_default.yml" ]
  tasks:
    - service: name={{ apache }} state=running
```

Then, add two files in the same folder as your example playbook, `apache_CentOS.yml`, and `apache_default.yml`. Define the variable `apache: httpd` in the CentOS file, and `apache: apache2` in the default file.

As long as your remote server has `facter` or `ohai` installed, Ansible will be able to read the OS of the server, translate that to a variable (`ansible_os_family`), and include the vars file with the resulting name. If ansible can't find a file with that name, it will use the second option (`apache_default.yml`). So, on a Debian or Ubuntu server, Ansible would correctly use `apache2` as the service name, even though there is no `apache_Debian.yml` or `apache_Ubuntu.yml` file available.

Inventory variables

Variables may also be added via Ansible inventory files, either inline with a host definition, or after a group:

```
# Host-specific variables (defined inline).
[washington]
app1.example.com proxy_state=present
app2.example.com proxy_state=absent

# Variables defined for the entire group.
[washington:vars]
cdn_host=washington.static.example.com
api_version=3.0.1
```

If you need to define more than a few variables, especially variables that apply to more than one or two hosts, inventory files can be cumbersome. In fact, Ansible's documentation recommends *not* storing variables within the inventory. Instead, you can use `group_vars` and `host_vars` YAML variable files within a specific path, and Ansible will assign them to individual hosts and groups defined in your inventory.

For example, to apply a set of variables to the host `app1.example.com`, create a blank file named `app1.example.com` at the location `/etc/ansible/host_vars/app1.example.com`, and add variables as you would in an included `vars_files` YAML file:

```
---
foo: bar
baz: qux
```

To apply a set of variables to the entire `washington` group, create a similar file in the location `/etc/ansible/group_vars/washington` (substitute `washington` for whatever group name's variables you're defining).

You can also put these files (named the same way) in `host_vars` or `group_vars` directories in your playbook's directory. Ansible will use the variables defined in the inventory `/etc/ansible/[host|group]_vars` directory first (if the appropriate files exist), then it will use variables defined in the playbook directories.

Another alternative to using `host_vars` and `group_vars` is to use conditional variable file imports, as was mentioned above.

Registered Variables

There are many times that you will want to run a command, then use its return code, stderr, or stdout to determine whether to run a later task. For these situations, Ansible allows you to use `register` to store the output of a particular command in a variable at runtime.

In the previous chapter, we used `register` to get the output of the `forever list` command, then used the output to determine whether we needed to start our Node.js app:

```
39 - name: "Node: Check list of Node.js apps running."
40   command: forever list
41   register: forever_list
42   changed_when: false
43
44 - name: "Node: Start example Node.js app."
45   command: forever start {{ node_apps_location }}/app/app.js
46   when: "forever_list.stdout.find('{{ node_apps_location}}/app/app.js') == -1"
```

In that example, we used a string function built into Python (`find`) to search for the path to our app, and if it was not present, the Node.js app was started.

We will explore the use of `register` further later in this chapter.

Accessing Variables

Simple variables (gathered by Ansible, defined in inventory files, or defined in playbook or variable files) can be used as part of a task using syntax like `{{ variable }}`. For example:

```
- command: /opt/my-app/rebuild {{ my_environment }}
```

When the command is run, Ansible will substitute the contents of `my_environment` for `{{ my_environment }}`. So the resulting command would be something like `/opt/my-app/rebuild dev`.

Many variables you will use are structured as arrays (or 'lists'), and simply accessing the array `foo` would not give you enough information to be useful (except when passing in the array in a context where Ansible will use the entire array, like when using `with_items`).

If you define a list variable like so:

```
foo_list:
  - one
  - two
  - three
```

You could access the first item in that array with either of the following syntax:

```
foo[0]
foo|first
```

Note that the first line uses standard Python array access syntax ('retrieve the first (0-indexed) element of the array'), whereas the second line uses a convenient *filter* provided by Jinja2. Either way is equally valid and useful, and it's really up to you whether you like the first or second technique.

For larger and more structured arrays (for example, when retrieving the IP address of the server using the facts Ansible gathers from your server), you can access any part of the array by drilling through the array keys, either using bracket (`[]`) or dot (`.`) syntax. For example, if you would like to retrieve the information about the `eth0` network interface, you could first take a look at the entire array using `debug` in your playbook:

```
# In your playbook.
tasks:
  - debug: var=ansible_eth0
```

```
TASK: [debug var=ansible_eth0] ************************************
ok: [webserver] => {
    "ansible_eth0": {
        "active": true,
        "device": "eth0",
        "ipv4": {
            "address": "10.0.2.15",
            "netmask": "255.255.255.0",
            "network": "10.0.2.0"
        },
        "ipv6": [
            {
                "address": "fe80::a00:27ff:feb1:589a",
                "prefix": "64",
                "scope": "link"
            }
        ],
```

```
        "macaddress": "08:00:27:b1:58:9a",
        "module": "e1000",
        "mtu": 1500,
        "promisc": false,
        "type": "ether"
    }
}
```

Now that you know the overall structure of the variable, you can use either of the following techniques to retrieve only the IPv4 address of the server:

```
{{ ansible_eth0.ipv4.address }}
{{ ansible_eth0['ipv4']['address'] }}
```

Host and Group variables

Ansible conveniently lets you define or override variables on a per-host or per-group basis. As we learned earlier, your inventory file can define groups and hosts like so:

```
[group]
host1
host2
```

The simplest way to define variables on a per-host or per-group basis is to do so directly within the inventory file:

```
[group]
host1 admin_user=jane
host2 admin_user=jack
host3

[group:vars]
admin_user=john
```

In this case, Ansible will use the group default variable 'john' for `{{ admin_user }}`, but for `host1` and `host2`, the admin users defined alongside the hostname will be used.

This is convenient and works well when you need to define a variable or two per-host or per-group, but once you start getting into more involved playbooks, you might need to add a few (3+) host-specific variables. In these situations, you can define the variables in a different place to make maintenance and readability much easier.

`group_vars` and `host_vars`

Ansible will search within the same directory as your inventory file (or inside `/etc/ansible` if you're using the default inventory file at `/etc/ansible/hosts`) for two specific directories: `group_vars` and `host_vars`.

You can place YAML files inside these directories named after the group name or hostname defined in your inventory file. Continuing our example above, let's move the specific variables into place:

```
---
# File: /etc/ansible/group_vars/group
admin_user: john
```

```
---
# File: /etc/ansible/host_vars/host1
admin_user: jane
```

Even if you're using the default inventory file (or an inventory file outside of your playbook's root directory), Ansible will also use host and group variables files located within your playbook's own `group_vars` and `host_vars` directories. This is convenient when you want to package together your entire playbook and infrastructure configuration (including all host/group-specific configuration) into a source-control repository.

You can also define a `group_vars/all` file that would apply to *all* groups, as well as a `host_vars/all` file that would apply to *all* hosts. Usually, though, it's a better idea to define sane defaults in your playbooks and roles (which will be discussed later).

Magic variables with host and group variables and information

If you ever need to retrieve a specific host's variables from another host, Ansible provides a magic `hostvars` variable containing all the defined host variables (from inventory files and any discovered YAML files inside `host_vars` directories).

```
# From any host, returns "jane".
{{ hostvars['host1']['admin_user'] }}
```

There are a variety of other variables Ansible provides that you may need to use from time to time:

- `groups`: A list of all group names in the inventory.
- `group_names`: A list of all the groups of which the *current* host is a part.
- `inventory_hostname`: The hostname of the current host, according to the *inventory* (this can differ from `ansible_hostname`, which is the hostname reported by the system).

- `inventory_hostname_short`: The first part of `inventory_hostname`, up to the first period.
- `play_hosts`: All hosts on which the current play will be run.

Please see Magic Variables, and How To Access Information About Other Hosts[59] in Ansible's official documentation for the latest information and further usage examples.

Facts (Variables derived from system information)

By default, whenever you run an Ansible playbook, Ansible first gathers information ("facts") about each host in the play. You may have noticed this whenever we ran playbooks in earlier chapters:

```
$ ansible-playbook playbook.yml

PLAY [group] ***************************************************************

GATHERING FACTS ************************************************************
ok: [host1]
ok: [host2]
ok: [host3]
```

Facts can be extremely helpful when you're running playbooks; you can use gathered information like host IP addresses, CPU type, disk space, operating system information, and network interface information to change when certain tasks are run, or change certain information used in configuration files.

To get a list of every gathered fact available, you can use the `ansible` command with the `setup` module:

```
$ ansible munin -m setup
munin.midwesternmac.com | success >> {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "167.88.120.81"
        ],
        "ansible_all_ipv6_addresses": [
            "2604:180::a302:9076",
[...]
```

If you don't need to use facts, and would like to save a few seconds per-host when running playbooks (this can be especially helpful when running an Ansible playbook dozens or hundreds of servers), you can set `gather_facts: no` in your playbook:

[59] http://docs.ansible.com/playbooks_variables.html#magic-variables-and-how-to-access-information-about-other-hosts

```
- hosts: db
  gather_facts: no
```

Many of my own playbooks and roles use facts like `ansible_os_family`, `ansible_hostname`, and `ansible_memtotal_mb` to register new variables or in tandem with `when`, to determine whether to run certain tasks.

If you have Facter[60] or Ohai[61] installed on a remote host, Ansible will also include their gathered facts as well, prefixed by `facter_` and `ohai_`, respectively. If you're using Ansible in tandem with Puppet or Chef, and are already familiar with those system-information-gathering tools, you can conveniently use them within Ansible as well. If not, Ansible's Facts are usually sufficient for whatever you need to do, and can be made even more flexible through the use of Local Facts.

If you run a playbook against similar servers or virtual machines (e.g. all your servers are running the same OS, same hosting provider, etc.), facts are almost always consistent in their behavior. When running playbooks against a diverse set of hosts (for example, hosts with different OSes, virtualization stacks, or hosting providers), know that some facts may contain different information than you were expecting. For Server Check.in[62], I have servers from no less than five different hosting providers, running on vastly different hardware, so I am sure to monitor the output of my `ansible-playbook` runs for abnormalities, especially when adding new servers to the mix.

Local Facts (Facts.d)

Another way of defining host-specific facts is to place `.fact` file in a special directory on remote hosts, `/etc/ansible/facts.d/`. These files can be either JSON or INI files, or you could use executables that return JSON. As an example, create the file `/etc/ansible/facts.d/settings.fact` on a remote host, with the following contents:

```
1 [users]
2 admin=jane,john
3 normal=jim
```

Next, use Ansible's `setup` module to display the new facts on the remote host:

[60] https://tickets.puppetlabs.com/browse/FACT
[61] http://docs.getchef.com/ohai.html
[62] https://servercheck.in/

```
$ ansible hostname -m setup -a "filter=ansible_local"
munin.midwesternmac.com | success >> {
    "ansible_facts": {
        "ansible_local": {
            "settings": {
                "users": {
                    "admin": "jane,john",
                    "normal": "jim"
                }
            }
        }
    },
    "changed": false
}
```

If you are using a playbook to provision a new server, and part of that playbook adds a local `.fact` file which generates local facts that are used later, you can explicitly tell Ansible to reload the local facts using a task like the following:

```
1 - name: Reload local facts.
2   setup: filter=ansible_local
```

While it may be tempting to use local facts rather than host_vars or other variable definition methods, remember that it's often better to build your playbooks in a way that doesn't rely (or care about) specific details of individual hosts. Sometimes it is necessary to use local facts (especially if you are using executables in facts.d to define the facts based on changing local environments), but it's almost always better to keep configuration in a central repository, and move away from host-specific facts.

Note that `setup` module options (like `filter`) won't work on remote Windows hosts, as of this writing.

Variable Precedence

TODO:

- Variable Precedence[63]
- Variable Precedence Graph[64] by drybjed[65], with caveat by SVG[66]

[63] http://docs.ansible.com/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable

[64] http://i.imgur.com/7WOcL1L.png

[65] https://twitter.com/drybjed/status/463770587924426752

[66] https://twitter.com/svg/status/463801958692712449

If/then/when - Conditionals

Many plays need only be run in certain circumstances. Some plays use modules with built-in idempotence (as is the case when ensuring a yum or apt package is installed), and you usually don't need to define further conditional behaviors for these plays.

However, there are many plays—especially those using Ansible's `command` or `shell` modules—which require further input as to when they're supposed to run, whether they've changed anything after they've been run, or when they've failed to run.

We'll cover all the main conditionals behaviors you can apply to Ansible plays, as well as how you can tell Ansible when a play has done something to a server or failed.

Jinja2 Expressions, Python built-ins, and Logic

Before discussing all the different uses of conditionals in Ansible, it's worthwhile to at least quickly cover a small part of Jinja2 (the syntax Ansible uses both for templates and for conditionals), and available Python functions (often referred to as 'built-ins'). Ansible uses expressions and built-ins with `when`, `changed_when`, and `failed_when` so you can describe these things to Ansible with as much precision as possible.

Jinja2 allows the definition of literals like strings (`"string"`), integers (`42`), floats (`42.33`), lists (`[1, 2, 3]`), tuples (like lists, but can't be modified) dictionaries (`{key: value, key2: value2}`), and booleans (`true` or `false`).

Jinja2 also allows basic math operations, like addition, subtraction, multiplication and division, and comparisons (== for equality, `!=` for inequality, `>=` for greater than or equal to, etc.). Logical operators are `and`, `or`, and `not`, and you can group expressions by placing them within parenthesis.

If you're familiar with almost any programming language, you will probably pick up basic usage of Jinja2 expressions in Ansible very quickly.

For example:

```
# The following expressions evaluate to 'true':
1 in [1, 2, 3]
'see' in 'Can you see me?'
foo != bar
(1 < 2) and ('a' not in 'best')

# The following expressions evaluate to 'false':
4 in [1, 2, 3]
foo == bar
(foo != foo) or (a in [1, 2, 3])
```

Jinja2 also offers a helpful set of 'tests' you can use to test a given object. For example, if you define the variable `foo` for only a certain group of servers, but not others, you can use the expression `foo is defined` with a conditional to evaluate to 'true' if the variable is defined, or false if not.

There are many other checks you can perform as well, like `undefined` (the opposite of `defined`), `equalto` (works like ==), `even` (returns true if the variable is an even number), `iterable` (if you can iterate over the object). We'll cover the full gamut later in the book, but for now, know that you can use Ansible conditionals with Jinja2 expressions to do some powerful things!

For the few cases where Jinja2 doesn't provide enough power and flexibility, you can invoke Python's built-in library functions (like `string.split`, `[number].is_signed()`) to manipulate variables and determine whether a given task should be run, resulted in a change, failed, etc.

As an example, I need to parse version strings from time to time, to find the major version of a particular project. Assuming the variable `software_version` is set to `4.6.1`, I can get the major version by splitting the string on the `.` character, then using the first element of the array. I can check if the major version is `4` using `when`, and choose to run (or not run) a certain task:

```
1 - name: Do something only for version 4 of the software.
2   [task here]
3   when: software_version.split('.')[0] == '4'
```

It's generally best to stick with simpler Jinja2 filters and variables, but it's nice to be able to use Python when you're doing more advanced variable manipulation.

register

In Ansible, any play can 'register' a variable, and once registered, that variable will be available to all subsequent plays. Registered variables work just like normal variables or host facts.

Many times, you may need the output (stdout or stderr) of a shell command, and you can get that in a variable using the following syntax:

```
- shell: my_command_here
  register: my_command_result
```

Later, you can access stdout (as a string) with `my_command_result.stdout`, and stderr with `my_command_result.stderr`.

Registered facts are very helpful for many types of plays, and can be used both with conditionals (defining when and how a play runs), and in any part of the play. As an example, if you have a command that outputs a version number string like "10.0.4", and you register the output as `version`, you can use the string later when doing a code checkout by printing the variable `{{ version.stdout }}`.

If you want to see the different properties of a particular registered variable, you can run a playbook with `-v` to inspect play output. Usually, you'll get access to values like `changed` (whether the play resulted in a change), `delta` (the time it took to run the play), `stderr` and `stdout`, etc. Some Ansible modules (like `stat`) add much more data to the registered variable, so always inspect the output with `-v` if you need to see what's inside.

when

One of the most helpful extra keys you can add to a play is a `when` statement. Let's take a look at a simple use of `when`:

```
- yum: pkg=mysql-server state=present
  when: is_db_server
```

The above statement assumes you've defined the `is_db_server` variable as a boolean (`true` or `false`) earlier, and will run the play if the value is true, or skip the play when the value is false.

If you only define the `is_db_server` variable on database servers (meaning there are times when the variable may not be defined at all), you could run plays conditionally like so:

```
- yum: pkg=mysql-server state=present
  when: (is_db_server is defined) and is_db_server
```

`when` is even more powerful if used in conjunction with variables registered by previous plays. For example, we want to check the status of a running application, and run a play only when that application reports it is 'ready' in its output:

```
- command: my-app --status
  register: myapp_result

- command: do-something-to-my-app
  when: "'ready' in myapp_result.stdout"
```

These examples are a little contrived, but they illustrate basic uses of `when` in your plays. Here are some examples of uses of `when` in real-world playbooks:

```
# From our Node.js playbook - register a command's output, then see
# if the path to our app is in the output. Start the app if it's
# not present.
- command: forever list
  register: forever_list
- command: forever start /path/to/app/app.js
  when: "forever_list.stdout.find('/path/to/app/app.js') == -1"

# Run 'ping-hosts.sh' script if 'ping_hosts' variable is true.
- command: /usr/local/bin/ping-hosts.sh
  when: ping_hosts

# Run 'git-cleanup.sh' script if a branch we're interested in is
# missing from git's list of branches in our project.
- command: chdir=/path/to/project git branch
  register: git_branches
- command: /path/to/project/scripts/git-cleanup.sh
  when: "(is_app_server == true) and ('interesting-branch' not in \
  git_branches.stdout)"

# Downgrade PHP version if the current version contains '5.4'.
- shell: php --version
  register: php_version
- shell: yum -y downgrade php*
  when: "'5.4' in php_version.stdout"

# Copy a file to the remote server if the hosts file doesn't exist.
- stat: path=/etc/hosts
  register: hosts_file
- copy: src=path/to/local/file dest=/path/to/remote/file
  when: hosts_file.stat.exists == false
```

`changed_when` and `failed_when`

Just like `when`, you can use `changed_when` and `failed_when` to influence Ansible's reporting of when a certain task results in changes or failures.

It is difficult for Ansible to determine if a given command results in changes, so if you use the `command` or `shell` module without also using `changed_when`, Ansible will always report a change. Most Ansible modules report whether they resulted in changes correctly, but you can also override this behavior by invoking `changed_when` yourself.

When using PHP Composer as a command to install project dependencies, it's useful to know when Composer installed something, or when nothing changed. Here's an example:

```
- name: Install dependencies via Composer.
  command: "/usr/local/bin/composer global require phpunit/phpunit --prefer-dist"
  register: composer
  changed_when: "'Nothing to install or update' not in composer.stdout"
```

You can see we used `register` to store the results of the command, then we checked whether a certain string was in the registered variable's stdout. Only when Composer doesn't do anything will it print "Nothing to install or update", so we use that string to tell Ansible if the task resulted in a change.

Many command-line utilities print results to stderr instead of stdout, so `failed_when` can be used to tell Ansible when a task has *actually* failed and is not just reporting its results in the wrong way. Here's an example where we need to parse the stderr of a Jenkins CLI command to see if Jenkins did, in fact, fail to perform the command we requested:

```
- name: Import a Jenkins job via CLI.
  shell: >
    java -jar /opt/jenkins-cli.jar -s http://localhost:8080/
    create-job "My Job" < /usr/local/my-job.xml
  register: import
  failed_when: "import.stderr and 'already exists' not in import.stderr"
```

In this case, we only want Ansible to report a failure when the command returns an error, **and** that error doesn't contain 'already exists'. It's debatable whether the command should report a job already exists via stderr, or just print the result to stdout... but it's easy to account for whatever the command does with Ansible!

`ignore_errors`

Sometimes there are commands that should be run always, and they often report errors. Or there are scripts you might run that output errors left and right, and the errors don't actually indicate a problem, but they're just annoying (and they cause your playbooks to stop executing).

For these situations, you can simply add `ignore_errors` to the task, and Ansible will remain blissfully unaware of any problems running a particular task. Be careful using this, though; it's usually best if you can find a way to work with and around the errors generated by tasks so playbooks *do* fail if there are actual problems.

Local Actions and Delegation

TODO:

- Delegation, Rolling Updates, and Local Actions[67]
- Waiting for reboots (`wait_for`).
- `delegate_to` for particular tasks.
- `local_action` shorthand syntax (equivalent to `delegate_to: 127.0.0.1`).
  - Test: Does `local_action` use SSH connection? Even for `command`/`shell`?
  - Test and explain difference between `--connection=local` and `local_action`.

Prompts

TODO.

Tags

TODO.

Summary

TODO.

```
 ______________________________________
/ Men have become the tools of their \
\ tools. (Henry David Thoreau)       /
 --------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

[67] http://docs.ansible.com/playbooks_delegation.html

Chapter 6 - Playbook Organization - Roles and Includes

So far, we've used fairly straightforward examples in this book. Most examples are ad-hoc for a particular server, and listing all tasks in one long listing makes for a fairly long playbook.

Ansible is very flexible when it comes to organizing your plays in more efficient ways so you can make your playbooks more maintainable, reusable, and powerful. We'll look at two ways to split up plays more efficiently: using includes and roles. Finally, we'll explore Ansible Galaxy, a repository of some community-maintained roles that help configure common packages and applications.

Includes

We've already seen one of the most basic ways of including other files in Chatper 4, when `vars_files` was used to place variables into a separate `vars.yml` file instead of inline with the playbook:

```
- hosts: all

  vars_files:
  - vars.yml
```

Tasks can easily be included in a similar way. In the `tasks:` section of your playbook, you can add `include` directives like so:

```
tasks:
  - include: included-playbook.yml
```

Just like with variable include files, tasks are formatted in a flat list in the included file. As an example, the `included-playbook.yml` could look like:

```
---
- name: Add profile info for user.
  copy: >
    src=example_profile
    dest=/home/{{ username }}/.profile
    owner={{ username }} group={{ username }} mode=744

- name: Add private keys for user.
  copy: >
    src={{ item.src }}
    dest=/home/.ssh/{{ item.dest }}
    owner={{ username }} group={{ username }} mode=600
  with_items: ssh_private_keys

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

In this case, you'd probably want to name the file `user-config.yml`, since it's used to configure a user account and restart some service. Now, in this and any other playbook that provisions or configures a server, if you want to configure a particular user's account, add the following in your playbook's `tasks` section:

```
- include: example-app-config.yml
```

We used `{{ username }}` and `{{ ssh_private_keys }}` variables in this include file instead of hard-coded values so we could make this include file reusable. You could define the variables in your playbook's inline variables or an included variables file, but Ansible also lets you pass variables directly into includes using normal YAML syntax. For example:

```
- { include: user-config.yml, username: johndoe, ssh_private_keys: [] }
- { include: user-config.yml, username: janedoe, ssh_private_keys: [] }
```

To make the syntax more readable, you can use structured variables, like so:

```
- include: user-config.yml
  vars:
    username: johndoe
    ssh_private_keys:
      - { src: /path/to/johndoe/key1, dest: id_rsa }
      - { src: /path/to/johndoe/key2, dest: id_rsa_2 }
- include: user-config.yml
  vars:
    username: janedoe
    ssh_private_keys:
      - { src: /path/to/janedoe/key1, dest: id_rsa }
      - { src: /path/to/janedoe/key2, dest: id_rsa_2 }
```

Include files can even include other files, so you could have something like the following:

```
tasks:
  - include: user-config.yml
```

inside `user-config.yml`

```
- include: ssh-setup.yml
```

Handler includes

Handlers can be included just like tasks, within a playbook's `handlers` section. For example:

```
handlers:
  - include: included-handlers.yml
```

This can be helpful in limiting the noise in your main playbook, since handlers are usually used for things like restarting services or loading a configuration, and can distract from the playbook's primary purpose.

Playbook includes

Playbooks can even be included in other playbooks, by simply using the same `include` syntax in the top level of your playbook. For example, if you have two playbooks—one to set up your webservers (`web.yml`), and one to set up your database servers (`db.yml`), you could use the following playbook to run both at the same time:

```
- hosts: all
  remote_user: root

  tasks:
    ...

- include: web.yml
- include: db.yml
```

This way, you can create playbooks to configure all the servers in your entire infrastructure, then create a master playbook that includes each of the individual playbooks. When you want to initialize your entire infrastructure, make changes across your entire fleet of servers, or just check to make sure their configuration matches your playbook definitions, you can run one `ansible-playbook` command!

Complete includes example

What if I told you we could remake the 137-line Drupal LAMP server playbook from Chapter 4 in just 21 lines? With includes, it's easy; just break out each of the sets of tasks into their own include files, and you'll end up with a main playbook like this:

```
1  ---
2  - hosts: all
3
4    vars_files:
5      - vars.yml
6
7    pre_tasks:
8      - name: Update apt cache if needed.
9        apt: update_cache=yes cache_valid_time=3600
10
11   handlers:
12     - include: handlers/handlers.yml
13
14   tasks:
15     - include: tasks/common.yml
16     - include: tasks/apache.yml
17     - include: tasks/php.yml
18     - include: tasks/mysql.yml
19     - include: tasks/composer.yml
20     - include: tasks/drush.yml
21     - include: tasks/drupal.yml
```

All you need to do is create two new folders in the same folder where you saved the Drupal `playbook.yml` file, `handlers` and `tasks`, then create files inside for each section of the playbook.

For example, inside `handlers/handlers.yml`, you'd have, simply:

```
---
- name: restart apache
  service: name=apache2 state=restarted
```

And inside `tasks/drush.yml`:

```
---
- name: "Check out drush master branch."
  git: repo=https://github.com/drush-ops/drush.git dest=/opt/drush

- name: "Install Drush dependencies with Composer."
  shell: >
    /usr/local/bin/composer install
    chdir=/opt/drush
    creates=/opt/drush/vendor/autoload.php

- name: "Create drush bin symlink."
  file: >
    src=/opt/drush/drush
    dest=/usr/local/bin/drush
    state=link
```

Separating all the tasks into separate includes files means you'll have more files to manage for your playbook, but it helps keep the main playbook more compact (meaning it's easier to see all the installation and configuration steps the playbook contains), and also separates tasks into individual, easily-maintainable groupings. Instead of having to browse one playbook with twenty-three separate tasks, you now maintain eight included files with two to five tasks, each.

It's much easier to maintain a more granular set of plays than one very long playbook. However, there's no reason to try to *start* writing a playbook with lots of individual includes. Most of the time, it's best to start with a monolithic playbook while you're working on the setup and configuration details, then move sets of tasks out to included files after you start seeing logical groupings.

You can also use tags (demonstrated in the previous chapter) to limit the playbook run to a certain include file. Using the above example, if you wanted to add a 'drush' tag to the included drush file (so you could run `ansible-playbook playbook.yml --tags=drush` and only run the drush tasks), you can change line 20 to the following:

```
20  - include: tasks/drush.yml tags=drush
```

You can find the entire example Drupal LAMP server playbook using include files in this book's code repository at https://github.com/geerlingguy/ansible-for-devops[68], in the `includes` directory.

You can't use variables for task include file names (like you could with `include_vars` directives, e.g. `include_vars: "{{ ansible_os_family }}.yml"` as a task, or with `vars_files`). There's usually a better way than conditional task includes to accomplish conditional task inclusion using a different playbook structure, or roles, which we will discuss next.

Roles

Including playbooks inside other playbooks makes your playbook organization a little more sane, but once you start wrapping up your entire infrastructure's configuration in playbooks, you might end up with something resembling Russian nesting dolls.

Wouldn't it be nice if there were a way to take bits of related configuration, and package them together nicely? Additionally, what if we could take these packages (often configuring the same thing on many different servers) and make them flexible so we can use the same package throughout our infrastructure, with slightly different settings on individual servers or groups of servers?

Ansible Roles can do all that, and more!

Let's dive into what makes an Ansible role by taking one of the playbook examples from Chapter 4 and splitting it into a more flexible structure using roles.

Role scaffolding

Instead of requiring you to explicitly include certain files and playbooks in a role, Ansible automatically includes any `main.yml` files inside specific directories that make up the role.

There are only two directories required to make a working Ansible role:

```
role_name
  --> meta
  --> tasks
```

If you create a directory structure like the one shown above, with a `main.yml` file in each directory, Ansible will run all the tasks defined in `tasks/main.yml` if you call the role from your playbook using the following syntax:

[68] https://github.com/geerlingguy/ansible-for-devops

```
---
- hosts: all
  roles:
    - role_name
```

Your roles (each one as its own directory, where the directory name is used by Ansible as the name of the role) can live in a couple different places—in the default global Ansible role path (configurable in `/etc/ansible/ansible.cfg`), or in a 'roles' folder directly within the same directory as your main playbook file.

Another simple way to build the scaffolding for a role (complete with all the available options/directories, a README file, and a structure suitable for contributing the role to Ansible Galaxy (we'll get to Galaxy in a little bit!) so it can easily be shared), is to use the `ansible-galaxy init` command. Running the command will create an example role in the current working directory, which you can then modify to suit your needs.

Building your first role

Let's jump right into cleaning up our Node.js server example from Chapter four, and break out one of the main parts of the configuration—installing Node.js and any npm modules we need for our server.

Create a `roles` folder in the same directory as your main `playbook.yml` file like we created in Chapter 4's first example, and inside that folder, create a new folder `nodejs` (which will be our role's name). Create two folders inside the `nodejs` role directory, `meta` and `tasks`.

Inside the `meta` folder, add a simple `main.yml` file with the following contents:

```
---
dependencies: []
```

The meta information for your role is defined in this file. In basic examples and simple roles, you just need to list any role dependencies (other roles that are required to be run before the current role can do its work), but you can add much more to this file to describe your role to Ansible and to Ansible Galaxy. We'll dive deeper into the meta information later. For now, save the file and head over to the `tasks` foler.

Create a `main.yml` file in this folder, and add the following contents (basically copying and pasting the configuration from the Chapter 4 example):

```
---
- name: Install Node.js (npm plus all its dependencies).
  yum: pkg=npm state=present enablerepo=epel

- name: Install forever module (to run our Node.js app).
  npm: name=forever global=yes state=latest
```

The Node.js directory structure should now look like the following:

```
nodejs-app/
  app/
    app.js
    package.json
  playbook.yml
  roles/
    nodejs/
      meta/
        main.yml
      tasks/
        main.yml
```

You now have a complete Ansible role that you can use in your node.js server configuration playbook. Delete the Node.js app installation lines from the other project, and reformat the playbook so the other tasks run first (in a `pre_tasks:` section instead of `tasks:`), then the role is included, then the rest of the tasks (in the main `tasks:` section). Something like:

```
pre_tasks:
  # EPEL/GPG setup, firewall configuration...

roles:
  - nodejs

tasks:
  # Node.js app deployment tasks...
```

You can view the full example of this playbook in the ansible-for-devops code repository[69].

Once you finish reformatting the main playbook, everything would run exactly the same during an `ansible-playbook`, with the exception of the tasks inside the `nodejs` role being prefixed with `nodejs | [Task name here]`.

[69] https://github.com/geerlingguy/ansible-for-devops/blob/master/nodejs-role/playbook.yml

This little bit of extra data shown during playbook runs is useful because it shows you from exactly what role tasks are being run, without you having to add in descriptions as part of the `name` values of the tasks.

Our role isn't all that helpful at this point, though, because it still does only one thing, and it's not really flexible enough to be used on other servers that might need different Node.js modules to be installed.

More flexibility with role vars and defaults

To make our role more flexible, we can make it use a list of npm modules instead of a hardcoded value, then allow playbooks using the role to define what modules they want to use.

When running a role's tasks, Ansible picks up variables defined in a role's `vars/main.yml` file and `defaults/main.yml` (I'll get to the differences between the two later), but will allow your playbooks to override the defaults or other role-provided variables if you want.

Modify the `tasks/main.yml` file to use a list variable and iterate through the list to install as many packages as your playbook wants:

```
1  ---
2  - name: Install Node.js (npm plus all its dependencies).
3    yum: pkg=npm state=present enablerepo=epel
4
5  - name: Install npm modules required by our app.
6    npm: name={{ item }} global=yes state=latest
7    with_items: node_npm_modules
```

Let's provide a sane default for the new `node_npm_modules` variable in `defaults/main.yml`:

```
1  ---
2  node_npm_modules:
3    - forever
```

Now, if you run the playbook as-is, it will still do the exact same thing—install the `forever` module. But since the role is more flexible, we could create a new playbook like our first, but add a variable (either in a `vars` section or in an included file via `vars_files`) to override the default, like so:

```
node_npm_modules:
  - forever
  - async
  - request
```

When you run the playbook with this custom variable (we didn't change *anything* with our `nodejs` role), all three of the above npm modules will be installed.

Hopefully you're beginning to see how this can be powerful!

Imagine if you had a playbook structure like:

```
---
- hosts: appservers
  roles:
    - yum-repo-setup
    - firewall
    - nodejs
    - app-deploy
```

Each one of the roles would live in its own isolated world, and could be shared with other servers and groups of servers in your infrastructure.

- A `yum-repo-setup` role could enable certain repositories and import their GPG keys.
- A `firewall` role could have per-server or per-inventory-group options for ports and services to allow or deny.
- An `app-deploy` role could deploy your app to a directory (configurable per-server) and set certain app options per-server or per-group.

All these things become very easy to manage when you have small bits of functionality separated into different roles. Instead of managing 100+ lines of playbook tasks, and manually prefixing every `name:` with something like "Common |" or "App Deploy |", you now manage a few roles with 10-20 lines of YAML each.

On top of that, when you're building your main playbooks, they can be extremely simple (like the above example), enabling you to see *everything* being configured and deployed on a particular server without scrolling through dozens of included playbook files and hundreds of tasks.

Variable precedence: Note that Ansible handles variables placed in included files in `defaults` with less precedence than those placed in `vars`. If you have certain variables you need to allow hosts/playbooks to easily override, you should probably put them into `defaults`. If they are common variables that should almost always be the values defined in your role, put them into `vars`. For more on variable precedence, see the aptly-named "Variable Precedence" section in the previous chapter.

Other role parts: handlers, files, and templates

Handlers

In one of the prior examples, we introduced handlers—tasks that could be called via the `notify` option after any playbook task resulted in a change—and an example handler for restarting Apache was given:

```
handlers:
  - name: restart apache
    service: name=apache2 state=restarted
```

In Ansible roles, handlers are first-class citizens, alongside tasks, variables, and other configuration. You can store handlers directly inside a `main.yml` file inside a role's `handlers` directory. So if we had a role for Apache configuration, our `handlers/main.yml` file could look like the following:

```
---
- name: restart apache
  command: service apache2 restart
```

You can call handlers defined in a role's handlers folder just like you would handlers included directly in your playbooks (e.g. `notify: restart apache`).

Files and Templates

For the following examples, let's assume our role is structured with files and templates inside `files` and `templates` directories, respectively:

```
roles/
  example/
    files/
      example.conf
    meta/
      main.yml
    templates/
      example.xml.j2
    tasks/
      main.yml
```

when copying a file directly to the server, add the filename or the full path from within a role's `files` directory, like so:

```
- name: Copy configuration file to server directly.
  copy: >
    src=example.conf
    dest=/etc/myapp/example.conf
    mode=644
```

Similarly, when specifying a template, add the filename or the full path from within a role's `templates` directory, like so:

```
- name: Copy configuration file to server using a template.
  template: >
    src=example.xml.j2
    dest=/etc/myapp/example.xml
    mode=644
```

The `copy` module simply copies files from within the module's `files` folder, and the `template` module runs given template files through the Jinja2 templating engine, merging in any variables available during your playbook run before copying the file to the server.

Organizing more complex and cross-platform roles

For simple package installation and configuration roles, you can get by with placing all tasks, variables, and handlers directly in the respective `main.yml` file Ansible automatically loads. But you can also *include* other files from within a role's `main.yml` files if needed.

As a rule of thumb, I like to keep my playbook and role task files under 100 lines of YAML if at all possible (that way it's easier for me to keep the entire set of tasks in my head while working through any issues). If I start nearing that limit, I usually split the tasks into logical groupings, and include files from the `main.yml` file.

Let's take a look at the way my `geerlingguy.apache` role is set up (it's available on Ansible Galaxy[70] and can be downloaded to your roles directory with the command `ansible-galaxy install geerlingguy.apache`; we'll discuss Ansible Galaxy itself later).

Initially, the role's main `tasks/main.yml` file looked something like the following (generally speaking):

[70] https://galaxy.ansible.com/list#/roles/428

```
- name: Ensure Apache is installed (via apt).

- name: Configure Apache with lineinfile.

- name: Enable Apache modules.
```

Soon after creating the role, though, I wanted to make the role work with both Debian and RedHat hosts. I could've simply added two sets of tasks in the `main.yml` file, resulting in twice the number of plays and a bunch of extra `when` statements:

```
- name: Ensure Apache is installed (via apt).
  when: ansible_os_family == 'Debian'

- name: Ensure Apache is installed (via yum).
  when: ansible_os_family == 'RedHat'

- name: Configure Apache with lineinfile (Debian).
  when: ansible_os_family == 'Debian'

- name: Configure Apache with lineinfile (Redhat).
  when: ansible_os_family == 'RedHat'

- name: Enable Apache modules (Debian).
  when: ansible_os_family == 'Debian'

- name: Other OS-agnostic tasks...
```

If I had gone this route, and continued with the rest of the playbook tasks in one file, I would've quickly surpassed my informal 100-line limit. So I chose to use includes in my main tasks file:

```
- name: Include OS-specific variables.
  include_vars: "{{ ansible_os_family }}.yml"

- include: setup-RedHat.yml
  when: ansible_os_family == 'RedHat'

- include: setup-Debian.yml
  when: ansible_os_family == 'Debian'

- name: Other OS-agnostic tasks...
```

Two important things to notice about this style of distribution-specific inclusion:

1. When including vars files (with `include_vars`), you can actually *use variables in the name of the file*. This is very handy for a variety of use cases, and here we're simply including a vars file in the format `distribution_name.yml`. For our purposes, since the role will be used on Debian and RedHat-based hosts, we can create `Debian.yml` and `RedHat.yml` files in our role's `defaults` and `vars` folders, and put distribution-specific variables there.
2. When including playbook files (with `include`), you can't use variables in the name of the file, but you can do the next best thing: include the files by name explicitly, and use a condition to tell Ansible whether to run the tasks inside (the `when` condition will be applied to every task inside the included playbook).

After setting things up this way, I put RedHat and CentOS-specific tasks (like `yum` tasks) into `tasks/setup-RedHat.yml`, and Debian and Ubuntu-specific tasks (like `apt` tasks) into `tasks/setup-Debian.yml`. There are other ways of making roles work cross-platform, but using distribution-specific variables files and included playbooks is one of the simplest.

Now this Apache role can be used across different distributions, and with clever usage of variables in tasks and in configuration templates, it can be used in a very wide variety of infrastructure that needs Apache installed.

Ansible Galaxy

Ansible roles are powerful and flexible; they allow you to encapsulate sets of configuration and deployable units of playbooks, variables, templates, and other files, so you can easily reuse them across different servers.

It's annoying to have to start from scratch every time, though; wouldn't it be better if people could share roles for commonly-installed applications and services? Enter Ansible Galaxy[71].

Ansible Galaxy, or just 'Galaxy', is a repository of community-contributed roles for common Ansible content. There are already hundreds of roles available which can configure and deploy common applications, and they're all available through the `ansible-galaxy` command, introduced in Ansible 1.4.2.

Galaxy offers the ability to add, download, and rate roles, and you can register either using a social account or a normal account on the site (though you don't need an account to install and use roles from Galaxy).

Getting roles from Galaxy

One of the primary functions of the `ansible-galaxy` command is retrieving roles from Galaxy. Roles must be downloaded before they can be used in playbooks.

Remember the basic LAMP (Linux, Apache, MySQL and PHP) server we installed earlier in the book? Let's create it again, but this time, using a few roles from Galaxy:

[71] https://galaxy.ansible.com/

```
$ ansible-galaxy install geerlingguy.apache geerlingguy.mysql geerlingguy.php
```

The latest version or a role will be downloaded if no version is specified. To specify a version, add the version after the role name, for example: `$ ansible-galaxy install geerlingguy.apache,1.0.0`.

Ansible Galaxy is still evolving rapidly, and has already seen many small improvements. There are a few areas where Galaxy could use some improvement (like browsing for roles by Operating System in the online interface, or automatically downloading roles that are included in playbooks), but most of these little bugs or rough areas will be fixed in time. Please check Ansible Galaxy's About[72] page and stay tuned to Ansible's blog for the latest updates.

A LAMP server in six lines of YAML

Now that we have these roles installed (Apache, MySQL, and PHP), we can quickly create a LAMP server. This example assumes you already have a CentOS-based linux VM or server booted and can connect to it or run Ansible as a provisioner via Vagrant on it, and that you've run the `ansible-galaxy install` command above to download the required roles.

First, create an Ansible playbook named `lamp.yml` with the following contents:

```
1  ---
2  - hosts: all
3    roles:
4      - geerlingguy.mysql
5      - geerlingguy.apache
6      - geerlingguy.php
```

Now, run the playbook against a host:

```
$ ansible-playbook -i path/to/custom-inventory lamp.yml
```

After a few minutes, an entire LAMP server should be set up and running. If you add in a few variables, you can configure virtualhosts, PHP configuration options, MySQL server settings, etc.

We've effectively reduced about thirty lines of YAML (from previous examples dealing with LAMP or LAMP-like servers) down to three. Obviously, the roles have extra code in them, but the power

[72] https://galaxy.ansible.com/intro

here is in abstraction. Since most companies have many servers using similar software, but with slightly different configurations, having centralized, flexible roles saves a lot of repetition.

You could think of Galaxy roles (which typically install and configure common software like Apache or MySQL, or set up security rules or frameworks) as glorified packages; they not only install software, but they configure it *exactly* how you want it, every time, with minimal manual labor. Additionally, many of these roles work across different flavors of Linux and UNIX, so you have better configuration portability!

A Solr server in six lines of YAML

Let's grab a few more roles and build an Apache Solr search server, which requires Java and Apache Tomcat to be installed and configured.

```
$ ansible-galaxy install geerlingguy.java geerlingguy.tomcat6 geerlingguy.solr
```

Then create a playbook named `solr.yml` with the following contents:

```
---
- hosts: all
  roles:
    - geerlingguy.java
    - geerlingguy.tomcat6
    - geerlingguy.solr
```

Now we have a fully-functional Solr server, and we could add some variables to configure it exactly how we want, by using a non-default port, or changing the memory allocation for Tomcat6.

I think you might get the point. Now, I could've also left out the `java` and `tomcat6` roles, since they'll be automatically picked up during installation of the `geerlingguy.solr` role (they're listed in the `solr` role's dependencies).

A role's page on the Ansible Galaxy website highlights available variables for setting things like what version of Solr to install, where to install it, etc. (as an example, view the geerlingguy.solr Galaxy page[73]).

Using community-maintained roles, you can build a wide variety of servers with minimal effort. Instead of having to maintain lengthy playbooks and roles unique to each server, Galaxy lets you build a list of the required roles, and a few variables that set up the servers with the proper versions and paths. Configuration management with Ansible Galaxy becomes *true* configuration management—you get to spend more time managing your server's configuration, and less time on packaging and building individual services!

[73] https://galaxy.ansible.com/list#/roles/445

Helpful Galaxy commands

Some other helpful `ansible-galaxy` commands you might use from time to time:

- `ansible-galaxy list` displays a list of installed roles, with version numbers
- `ansible-galaxy remove [role]` removes an installed role
- `ansible-galaxy init` can be used to create a role template suitable for submission to Ansible Galaxy

You can configure the default path where Ansible roles will be downloaded by editing your `ansible.cfg` configuration file (normally located in `/etc/ansible/ansible.cfg`), and setting a `roles_path` in the `[defaults]` section.

Contributing to Ansible Galaxy

If you've been working on some useful Ansible roles, and you'd like to share them with others, all you need to do is make sure they follow Ansible Galaxy's basic template (especially within the `meta/main.yml` and `README.md` files). To get started, use `ansible-galaxy init` to generate a basic Galaxy template, and make your own role match the Galaxy template's structure.

Then push your role up to a new project on GitHub (I usually name my Galaxy roles like `ansible-role-[rolename]`, so I can easily see them when browsing my repos on GitHub), and add a new role while logged into galaxy.ansible.com.

Summary

Using includes and Ansible roles organizes Playbooks and makes them maintainable. This chapter introduced different ways of using `include`, the power and flexible structure of roles, and how you can utilize Ansible Galaxy, the community repository of configurable Ansible roles that do just about anything.

```
 ________________________________________
/ When the only tool you own is a hammer, \
| every problem begins to resemble a      |
\ nail. (Abraham Maslow)                  /
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Chapter 7 - Inventories

TODO.

Chapter 8 - Ansible Modules

TODO.

Chapter 9 - Deployments with Ansible

TODO:

- Server Check.in[74] infrastructure walkthrough
- Rolling Updates
- `serial` batch size (instead of running each play in parallel on all servers) - can be defined per-play or per-playbook.
- Manage load balancers (BigIP, ELB, netscaler, etc.) in [pre|post]_task.
- `max_fail_percentage`
- `delegate_to` / `local_action`
- Notifications with Ansible[75]

[74] https://servercheck.in/

[75] http://www.ansible.com/blog/listen-to-your-servers-talk

Chapter 10 - Server Security and Ansible

One of the first configuration steps that should be performed on any new server—especially any server with any exposure (direct or indirect) to the public Internet)—is security configuration.

I debated adding this chapter earlier in the book, as the importance of a secure configuration (especially when automating server configuration, application deployments, etc.) cannot be understated. But as it is, I decided to focus on Ansible's core functionality before giving a general overview of security, especially pertaining to Linux servers.

There are nine basic measures that must be taken to make sure that servers are secure from unauthorized access or intercepted communications:

1. Use secure and encrypted communication.
2. Disable root login and use `sudo`.
3. Remove unused software, open only required ports.
4. Use the principle of least privilege.
5. Update the OS and installed software.
6. Use a properly-configured firewall.
7. Make sure log files are populated and rotated.
8. Monitor logins and block suspect IP addresses.
9. Use SELinux (Security-Enhanced Linux).

Your infrastructure is as weak as the weakest server; in many high-profile security breaches, one poorly-secured server acts as a gateway into the rest of the network. Don't let your servers be *those* servers! Good security also helps you achieve the holy grail of system administration—100% uptime.

In this chapter, you'll learn about Linux security and how Ansible can help secure your servers, following the basic topics above.

A brief history of SSH and remote access

In the beginning, computers were the size of large conference rooms. A punch card reader would merrily accept pieces of paper that instructed the computer to do something, and then a printer would etch the results into another piece of paper. Thousands of mechanical parts worked harmoniously (when they *did* work) to compute relatively simple commands.

As time progressed, computers became somewhat smaller, and interactive terminals became more user-friendly, but they were still wired directly into the computer being used. Mainframes came to the fore in the 1960s, originally used via typewriter and teletype interfaces, then via keyboards and small text displays. As networked computing became more mainstream in the 1970s and 1980s, remote terminal access was used to interact with the large central computers.

The first remote terminal interfaces assumed a high level of trust between the central computer and all those on the network, because the small, centralized networks used were physically isolated from one another.

Telnet

In the late 1960s, the Telnet protocol was defined and started being used over TCP networks (normally on port 23) for remote control over larger private networks, and eventually the public Internet.

Telnet's underlying technology (a text-based protocol to transfer data between different systems) was the basis for many foundational communications protocols in use today, including HTTP, FTP, and POP3. However, plain text streams are not secure, and even with the addition of TLS and SASL, Telnet was never very secure by default. With the advent of SSH (which we'll get to in a bit), the protocol has declined in popularity for most remote administration purposes.

Telnet still has uses like configuring devices over local serial connections, or checking if a particular service is operating correctly on a remote server (like an HTTP server on port 80, mysql on port 3306, or munin on port 4949), but it is not installed by default on modern Linux distributions.

Plain text communications over a network are only as secure as the network's weakest link. In the early days of computer networking, networks were usually isolated to a specific company or educational institution, so transmitting things like passwords or secrets in plain text using the TCP protocol wasn't such a bad idea. Every part of the network (cabling, switches, and routers) was contained inside a secured physical perimeter. When connections started moving to the public Internet, this changed.

TCP packets can be intercepted over the Internet, at any point between the client and server, and these packets can easily be read if not encrypted. Therefore, plain text protocols are highly insecure, and should never be used to transmit sensitive information or system control data. Even on highly secure networks with properly-configured firewalls, it's a bad idea to use insecure communication methods like plain text rlogin and telnet connections for authentication and remote control.

Try running `traceroute google.com` in your terminal. Look at each of the hops between you and Google's CDN. Do you know who controls each of the devices between your computer and Google? Do you trust these operators with all of your personal or corporate secrets? Probably not. Each of these connection points—and each network device and cable connecting them—is a weak point exposing you to a man-in-the-middle attack. Strong encryption is needed between your computer and the destination if you want to ensure data security.

rlogin, rsh and rcp

`rlogin` was introduced in BSD 4.2 in 1983, and has been distributed with many UNIX-like systems alongside Telnet until recently. rlogin was used widely during the 80s and much of the 90s.

Just like Telnet, a user could log into the remote system with a password, but rlogin additionally allowed automatic (passwordless) logins for users on trusted remote computers. rlogin also worked better than telnet for remote administration, as it worked correctly with certain characters and commands where telnet required extra translation.

However, like Telnet, rlogin still used plain text communications over TCP port 513 by default. On top of that, rlogin also didn't have many safeguards against clients spoofing their true identities. Some of rlogin's intrinsic flaws were highlighted in a 1998 report by Carnegie Mellon, rlogin: The Untold Story[76].

`rsh` ("remote shell") is a command line program used alongside rlogin to execute individual shell commands remotely, and `rcp` ("remote copy") is used for remote file copies. `rsh` and `rcp` inherited the same security problems as rlogin, since they use the same connection method (over different ports).

SSH

Secure Shell was created in 1995 by Finland native Tatu Ylönen, in response to a password-sniffing attack[77] at his university. Seeing the flaws in plain text communication for secure information, Tatu created Secure Shell/SSH with a strong emphasis on encryption and security.

His version of SSH was developed for a few years as freeware with liberal licensing, but as his SSH Communications Security Corporation[78] began limiting the license and commercializing SSH, alternative forks began to gain in popularity. The most popular fork, OSSH, by Swedish programmer Bjoern Groenvall, was chosen as a starting point by some developers from the OpenBSD project.

OpenBSD was (and still is!) a highly secure, free version of BSD UNIX, and the project's developers needed a secure remote communication protocol, so a few project members worked to clean up and improve OSSH[79] so it could be included in OpenBSD's 2.6 release in December 1999. From there, it was quickly ported and adopted for all major versions of Linux, and is now ubiquitous in the world of POSIX-compliant operating systems.

How does SSH work, and what makes it better than telnet or rlogin? It starts with the basic connection. SSH connection encryption works similarly to SSL for secure HTTP connections, but its authentication layer adds more security:

1. When you enter `ssh user@example.host` to connect to the `example.host` server as `user`, your client and the host exchange keys.

[76] http://resources.sei.cmu.edu/asset_files/TechnicalReport/1998_005_001_16670.pdf
[77] http://en.wikipedia.org/wiki/Secure_Shell#Version_1.x
[78] http://www.ssh.com/
[79] http://www.openbsd.org/openssh/history.html

2. If you're connecting to a host the first time, or if the host's key has changed since last time you connected (this happens often when connecting via DNS rather than directly by IP), SSH will prompt you for your approval of the host key.
3. If you have a private key in your `~/.ssh` folder that matches one of the keys in `~/.ssh/authorized_keys` on the remote system, the connection will continue to step 4. Otherwise, if password authentication is allowed, SSH will prompt you for your password. There are other authentication methods as well, such as Kerberos, but they are less common and not covered in this book.
4. The transferred key is used to create a session key that's used for the remainder of the connection, encrypting all communication with a cipher such as AES, 3DES, Blowfish or RC4 ('arcfour').
5. The connection remains encrypted and persists until you exit out of the remote connection (in the case of an interactive session), or until the operation being performed (an `scp` or `sftp` file transfer, for example) is complete.

SSH uses encrypted keys to identify the client and host (which adds a layer of security over `telnet` and `rlogin`'s defaults), and then sets up a per-session encrypted channel for further communication. This same connection method is used for interactive `ssh` sessions, as well as for services like:

- `scp` (secure copy), SSH's counterpart to rlogin's `rcp`.
- `sftp` (secure FTP), SSH's client/server file transfer protocol.
- SSH port forwarding (so you can run services securely over remote servers).
- SSH X11 forwarding (so you can use X windows securely).

(A full list of features is available on OpenBSD's site: OpenSSH Features[80]).

The full suite of SSH packages also includes helpful utilities like `ssh-keygen`, which generates public/private key pairs suitable for use when connecting via SSH. You can also install the utility `ssh-copy-id`, which speeds up the process of manually adding your identity file to a remote server.

SSH is fairly secure by default—certainly more so than telnet or rlogin's default configuration—but for even greater security, there are a few extra settings you should use (all of these settings are configured in `/etc/ssh/sshd_config`, and require a restart of the `sshd` service to take effect):

1. **Disable password-based SSH authentication**. Even though passwords are sent in the clear, disabling password-based authentication makes it impossible for brute-force password attacks to even be *attempted*, even if you have the additional (and recommended) layer of something like Fail2Ban running. Set `PasswordAuthentication no` in the configuration.

[80] http://www.openbsd.org/openssh/features.html

2. **Disable root account remote login**. You shouldn't log in as the root user regardless (use `sudo` instead), but to reinforce this good habit, disable remote root user account login by setting `PermitRootLogin no` in the configuration. If you need to perform actions as root, either use `sudo` (preferred), or if it's absolutely necessary to work interactively as root, login with a normal account, then `su` to the root account.
3. **Explicitly allow/deny SSH for users**. You can enable or disable SSH access for particular users on your system with `AllowUsers` and `DenyUsers`. To allow only 'John' to log in, the rule would be `AllowUsers John`. To allow any user *except* John to log in, the rule would be `DenyUsers John`.
4. **Use a non-standard port**. You can change the default SSH port from 22 to something more obscure, like 2849, and prevent thousands of 'script kiddie' attacks that simply look for servers responding on port 22. While security through obscurity is no substitute for actually securing SSH overall, it can provide a slight extra layer of protection. To change the port, set `Port [new-port-number]` in the configuration.

We'll cover how Ansible can help configure some of these particular options in SSH in the next section.

The evolution of SSH and the future of remote access

It has been over a decade since OpenSSH became the *de facto* standard of remote access protocols, and in that time, Internet connectivity has changed dramatically. For reliable, low-latency LAN and Internet connections, SSH is still the king due to its simplicity, speed, and security. But in high-latency environments (think 3G or 4G mobile network connections, or satellite uplinks), using SSH can be a slow and painful experience.

In some circumstances, just *establishing a connection* can take some time. Additionally, once connected, the delay inherent in SSH's TCP interface (where every packet must reach its destination and be acknowledged before further input will be accepted) means entering commands or viewing progress over a high-latency connection is an exercise in frustration.

Mosh[81], "the mobile shell", a new alternative to SSH, uses SSH to establish an initial connection, then synchronizes the following local session with a remote session on the server via UDP.

Using UDP instead of TCP requires Mosh to do a little extra behind-the-scenes work to synchronize the local and remote sessions (instead of simply sending all local keystrokes over the wire serially via TCP, then waiting for stdout and stderr to be returned, like SSH).

Mosh also promises better UTF-8 support than SSH, and is well supported by all the major POSIX-like operating systems (and can even run inside Google Chrome!).

It will be interesting to see where the future leads with regard to remote terminal access, but one thing is for sure: Ansible will continue to support the most secure, fast, and reliable connection methods to help you build and manage your infrastructure!

[81] https://www.usenix.org/system/files/conference/atc12/atc12-final32.pdf

Use secure and encrypted communication

We spent a lot of time discussing SSH's heritage and the way it works because it is, in many ways, the foundation of a secure infrastructure—in almost every circumstance, you will allow SSH remote access for your servers, so it's important you know how it works, and how to configure it to ensure you always administer the server securely, over an encrypted connection.

Let's look at the security settings configured in `/etc/ssh/sshd_config` (mentioned earlier), and how we can control them with Ansible.

For our secure server, we want to disable password-based SSH authentication (make sure you can already log in via your SSH key before you do this!), disable remote root login, and change the port over which SSH operates. Let's do it!

```
- hosts: example
  tasks:
    - name: Update SSH configuration to be more secure.
      lineinfile: >
        dest=/etc/ssh/sshd_config
        regexp="{{ item.regexp }}"
        line="{{ item.line }}"
        state=present
      with_items:
        - {
          regexp: "^PasswordAuthentication",
          line: "PasswordAuthentication no"
        }
        - {
          regexp: "^PermitRootLogin",
          line: "PermitRootLogin no"
        }
        - {
          regexp: "^Port",
          line: "Port 2849"
        }
      notify: restart ssh

  handlers:
    - name: restart ssh
      service: name=ssh state=restarted
```

In this extremely simple playbook, we set three options in SSH configuration (`PasswordAuthentication no`, `PermitRootLogin no`, and `Port 2849`) using Ansible's `lineinfile` module, then use a handler

we define in the `handlers` section to restart the ssh service. (Note that the task and handler defined here would probably be in separate files in a real-world playbook).

Note that if you change certain SSH settings, like the port for SSH, you will need to make sure Ansible's inventory is updated. You can explicitly define the SSH port for a host with the option `ansible_ssh_port`, and the local path to a private key file (identity file) with `ansible_ssh_private_key_file`, though Ansible uses keys defined by your `ssh-agent` setup, so typically a manual definition of the key file is not required.

Disable root login and use `sudo`

We've already disabled root login with Ansible's `lineinfile` module in the previous section, but we'll cover a general Linux best practice here: don't use the root account if you don't absolutely need to use it.

Linux's `sudo` allows you (or other users) to run certain commands with root privileges (by default—you can also run commands as another user), ensuring you can do things that need elevated privileges without requiring you to be logged in as root (or another user).

Using sudo also forces you to be more explicit when performing certain actions with security implications, which is always a good thing. You don't want to accidentally delete a necessary file, or turn off a required service, which is easy to do if you're root.

In Ansible, it's preferred you log into the remote server with a normal or admin-level system account, and use the `sudo` parameter with a value of `yes` with any play or playbook include that needs elevated privileges. For example, if restarting Apache requires elevated privileges, you would write the play like so:

```
- name: Restart Apache.
  service: name=httpd state=restarted
  sudo: yes
```

You can also add `sudo_user: [username]` to a task to specify a specific user account to use with `sudo` (this will only apply if `sudo` is already set on the task or in the playbook).

You can also use Ansible to control sudo's configuration, defining who should have access to what commands and whether the user should be required to enter a password, among other things.

As an example, we can set up the user `johndoe` with permission to use any command as root via `sudo` by adding a line in the `/etc/sudoers` file with Ansible's `lineinfile` module:

```
- name: Add sudo group rights for deployment user.
  lineinfile: >
    dest=/etc/sudoers
    regexp='^%johndoe'
    line='johndoe ALL=(ALL) NOPASSWD: ALL'
    state=present
```

If you're ever editing the sudoers file by hand, you should use `visudo`, which validates your changes and makes sure you don't break sudo when you save the changes. When using Ansible with `lineinfile`, you have to use caution when making changes, and make sure your syntax is correct.

Another way of changing the sudoers file, and ensuring the integrity of the file, is to create a sudoers file locally, and copy it using Ansible's `copy` module, with a validation command, like so:

```
- name: Copy validated sudoers file into place.
  copy: >
    src=sudoers
    dest=/etc/sudoers
    validate='visudo -cf %s'
```

The `%s` is simply a placeholder for the file's path, and will be filled in by Ansible before the sudoers file is copied into its final destination. The same parameter can be passed into Ansible's `template` module, if you need to copy a filled-in template to the server instead of a static file.

The sudoers file syntax is very powerful and flexible, but also a bit obtuse. Read the entire Sudoers Manual[82] for all the details, or check out the sample sudoers file[83] for some practical examples.

Remove unused software, open only required ports

Before the widespread use of configuration management tools for servers, when snowflake servers were the norm, many servers would become bloated with extra software no longer in active use, open ports for old services that are no longer needed, and old configuration settings serving no purpose other than to act as a potential attack vector.

If you're not actively using a piece of software, or there's a cron task running that isn't required, get rid of it. If you're using Ansible for your entire infrastructure, this shouldn't be an issue, since you could just bring up new servers to replace old ones when you have major configuration and/or package changes. But if not, consider adding in a 'cleanup' role or at least a task to remove packages that shouldn't be installed, like:

[82] http://www.sudo.ws/sudoers.man.html
[83] http://www.sudo.ws/sudo/sample.sudoers

```
- name: Remove unused packages.
  apt: pkg={{ item }} state=absent purge=yes
  with_items:
    - apache2
    - nano
    - mailutils
```

With modules like `yum`, `apt`, `file`, and `mysql_db`, a `state=absent` parameter means Ansible will remove whatever packages, files or databases you want, and will check to make sure this is still the case during future runs of your playbook.

Opening only required ports (and, as a secondary) is something to be done by a simple set of firewall rules. This will be covered fully in the "Use a properly-configured firewall" section, but as an example, don't leave port 25 open on your server unless your server will be used as an SMTP relay server. Further, make sure the services you have listening on your open ports are configured to only allow access from trusted clients.

Use the principle of least privilege

Users, applications, and processes should only be able to access information (files) and resources (memory, network ports, etc) that are necessary for their operation.

Many of the other basic security measures in this chapter are tangentially related to the principle of least privilege, but user account configuration and file permissions are two main areas that are directly related to the principle.

User account configuration

New user accounts, by default, have fairly limited permissions on a Linux server. They usually have a home folder, over which they have complete control, but any other folder or file on the system is only available for reading, writing, or execution if the folder has group permissions set.

Usually, users can gain access to other files and services through two methods:

1. Adding the user to another group with wider access privileges.
2. Allowing the user to use the `sudo` command to execute commands and access files as `root` or another user.

For the former method, please read the next section on file permissions to learn how to limit access. For the latter, please make sure you understand the use of sudoers as explained earlier in this chapter.

File permissions

Every Ansible module that deals with files has file ownership and permission parameters available, including `owner`, `group`, and `mode`. Almost every time you handle files (using `copy`, `template`, `file`, etc.), you should explicitly define the correct permissions and ownership. For example, for a configuration file (in our example, the GitLab configuration file) that should *only* be readable or writeable by the root user, set the following:

```
1  - name: Configure the GitLab global configuration file.
2    file: >
3      path=/etc/gitlab/gitlab.rb
4      owner=root group=root mode=600
```

File permissions may seem a bit obtuse, and sometimes, they may cause headaches. But in reality, using octal numbers to represent file permissions is a helpful way to encapsulate a lot of configuration in three simple numbers. The main thing to remember is the following: for each of the file's *user*, *group*, and for *everyone* (each of the three digits), use the following digits to represent permission levels:

```
7: rwx (read/write/execute)
6: rw- (read/write)
5: r-x (read/execute)
4: r-- (read)
3: -wx (write/execute)
2: -w- (write)
1: --x (execute)
0: --- (no permissions)
```

In simpler terms, 4 = read, 2 = write and 1 = execute. Therefore read (4) and write (2) is 6 in the octal representation, and read (4) and execute (1) is 5.

Less experienced admins are overly permissive, setting files and directories to `777` to fix issues they have with their applications. To allow one user (for example, your webserver user, `httpd` or `nginx`) access to a directory or some files, you should consider setting the directory's or files' *group* to the user's group instead of giving permissions to *every user on the system*!

For example, if you have a directory of web application files, the *user* (or in Ansible's terminology, "owner") might be your personal user account, or a deployment or service account on the server. Set the *group* for the files to a group the webserver user is in, and the webserver should now be able to access the files (assuming you have the same permissions set for the user and group, like `664`).

Update the OS and installed software

Every year, hundreds of security updates are released for the packages running on your servers, some of them fixing critical bugs. If you don't keep your server software up to date, you will be extremely vulnerable, especially when large exposures like Heartbleed[84] are uncovered.

At a minimum, you should schedule regular patch maintenance and package upgrade windows, and make sure you test the upgrades and patches on non-critical servers to make sure your applications work *before* applying the same on your production infrastructure.

With Ansible, since you already have your entire infrastructure described via Ansible inventories, you should be able to use a command like the following to upgrade all installed packages on a RedHat-based system:

```
$ ansible webservers -m yum -a "name=* state=latest"
```

On a Debian-based system, the syntax is similar:

```
$ ansible webservers -m apt -a "upgrade=dist update_cache=yes"
```

The above commands will upgrade *everything* installed on your server. Sometimes, you only want to install security-related updates, or exclude certain packages. In those cases, you need to configure `yum` or `apt` to tell them what to do (edit `/etc/yum.conf` for `yum` on RedHat-based systems, or use `apt-mark hold [package-name]` to keep a certain package at its current version on Debian-based systems).

Automating updates

Fully automated daily or weekly package and system upgrades provide even greater security. Not every environment or corporation can accommodate frequent automated upgrades (especially if your application has been known to break due to past package updates, or relies on custom builds or specific package versions), but if you can do it for your servers, it will increase the depth of your infrastructure's security.

As mentioned in an earlier sidebar, GPG package signature checking is enabled by default for all package-related functionality. It's best to leave GPG checks in place, and import keys from trusted sources when necessary, *especially* when using automatic updates, if you want to prevent potentially insecure packages from being installed on your servers!

[84] http://heartbleed.com/

Automating updates for RedHat-based systems

RedHat 6 and later (and modern versions of Fedora, and RedHat derivatives like CentOS) uses a simple cron-based package, `yum-cron`, for automatic updates. For basic, set-and-forget usage, install `yum-cron` and make sure it's started and set to run on system boot:

```
- name: Install yum-cron.
  yum: pkg=yum-cron state=installed

- name: Ensure yum-cron is running and enabled on boot.
  service: name=yum-cron state=started enabled=yes
```

Further configuration (such as packages to exclude from automatic updates) can be done in the `yum.conf` file, at `/etc/yum.conf`.

Automating updates for Debian-based systems

Debian and its derivatives typically use the `unattended-upgrades` package to configure automatic updates. Like `yum-cron`, it is easy to install, and its configuration is placed in a variety of files within `/etc/apt/apt.conf.d/`:

```
- name: Install unattended upgrades package.
  apt: pkg=unattended-upgrades state=installed

- name: Copy unattended-upgrades configuration files in place.
  template: >
    src=../templates/{{ item }}.j2
    dest=/etc/apt/apt.conf.d/{{ item }}
    owner=root group=root mode=0644
  with_items:
    - 10periodic
    - 50unattended-upgrades
```

The template files copied in the second task should look something like the following:

```
# File: /etc/apt/apt.conf.d/10periodic
APT::Periodic::Update-Package-Lists "1";
APT::Periodic::Download-Upgradeable-Packages "1";
APT::Periodic::AutocleanInterval "7";
APT::Periodic::Unattended-Upgrade "1";
```

This file provides configuration for the `apt` script that runs as part of the unattended upgrades package, and tells apt whether to enable unattended upgrades.

```
# File: /etc/apt/apt.conf.d/50unattended-upgrades
Unattended-Upgrade::Automatic-Reboot "false";

Unattended-Upgrade::Allowed-Origins {
        "Ubuntu lucid-security";
//      "Ubuntu lucid-updates";
};
```

This file provides further configuration for unattended upgrades, like whether to automatically restart the server for package and kernel upgrades that require a reboot (make sure, if you have this set to `false`, that you get notifications or check in on your servers so you know when they'll need a manual reboot!), or what `apt` sources should be checked for updated packages.

Use a properly-configured firewall

TODO.

Make sure log files are populated and rotated

TODO.

Monitor logins and block suspect IP addresses

TODO.

Use SELinux (Security-Enhanced Linux)

TODO:

- Manage selinux with configuration management[85]

Summary and further reading

This chapter contains a broad overview of some Linux security best practices, and how Ansible can help you conform to them. There is a wealth of good information on the Internet to help you secure your servers, including articles and publications like the following:

[85] https://speakerdeck.com/arrfab/manage-selinux-with-your-cfgmtmt-solution

- Linode Library: Linux Security Basics[86]
- My First Five Minutes on a Server[87]
- 20 Linux Server Hardening Security Tips[88]
- Unix and Linux System Administration Handbook[89]

Also, some of the security configuration in this chapter is encapsulated in a simple Ansible role on Ansible Galaxy, which you can use for your own servers: security role by geerlingguy[90].

```
 ______________________________________
/ Bad planning on your part does not  \
| constitute an emergency on my part. |
\ (Proverb)                           /
 --------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

[86] https://library.linode.com/security/basics
[87] http://plusbryan.com/my-first-5-minutes-on-a-server-or-essential-security-for-linux-servers
[88] http://www.cyberciti.biz/tips/linux-security.html
[89] http://www.admin.com/
[90] https://galaxy.ansible.com/list#/roles/1030

Chapter 11 - Automating Your Automation with Ansible Tower

Throughout this book, all the examples use Ansible's CLI to run playbooks and report back the results. For smaller teams, especially when everyone on the team is well-versed in how to use Ansible, YAML syntax, and follows security best practices with playbooks and variables files, using the CLI can be a sustainable approach.

But for many organizations, there are needs that stretch basic CLI use too far:

- The business needs detailed reporting of infrastructure deployments and failures, especially for audit purposes.
- Team-based infrastructure management requires varying levels of involvement in playbook management, inventory management, and key and password access.
- A thorough visual overview of the current and historical playbook runs and server health helps identify potential issues before they affect the bottom line.
- Playbook scheduling can help ensure infrastructure remains in a known state.

Ansible Tower checks off these items—and many more—and provides a great mechanism for team-based Ansible usage. The product is currently free for teams managing ten or fewer servers (it's basically an 'unlimited trial' mode), and has flexible pricing for teams managing dozens to thousands of servers.

While this book includes a brief overview of Tower, and how you can get started with Tower, it is highly recommended that you read through Ansible, Inc's extensive Tower User Guide[91], which includes details this book won't be covering such as LDAP integration and multiple-team playbook management workflows.

Getting and Installing Ansible Tower

TODO.

Using Ansible Tower

TODO.

[91] http://releases.ansible.com/ansible-tower/docs/tower_user_guide-latest.pdf

Tower Alternatives

TODO:

- Jenkins[92]
- Rundeck[93]
- Go CI[94]

Summary

TODO.

```
 ________________________________________
/ The first rule of any technology used  \
| in a business is that automation       |
| applied to an efficient operation will |
| magnify the efficiency. The second is  |
| that automation applied to an          |
| inefficient operation will magnify the |
\ inefficiency. (Bill Gates)             /
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

[92] http://jenkins-ci.org/
[93] http://rundeck.org/
[94] http://www.go.cd/

Chapter 12 - Etc...

This chapter is a placeholder for further topics, including those in the list below. Some of these topics may be broken out into individual chapters, or integrated into other prior chapters.

TODO:

- Windows Support[95]
- Ansible configuration (`/etc/ansible/ansible.cfg`)
- Best Practices (and 'real world' scenarios)
- Looping (if not covered in-depth earlier)
- Ansible Vault
- Asynchronous actions, Polling, distributed Ansible
- Error handling
- Server security with Ansible (especially SSH agent/key auth configuration)

Testing Ansible Playbooks

At this point, you should be able to convert almost any bit of your infrastructure's configuration into Ansible playbooks, roles, and inventories. And before deploying any changes, you should be testing the changes in a non-production environment (just like you would with application releases). Manually running a playbook that configures your entire infrastructure, then making sure it does what you expect, is a good start towards order and stability.

Since everything's in code, and since all you're doing is clicking "Go" and checking the result, why not automate this process?

Just like application code, you should test your infrastructure code. And lucky for you, there are already many ways to do this! This section will cover different levels of infrastructure testing, and highlight tools and techniques you can use to make sure you have thoroughly-tested everything before it goes to production.

Unit, Integration and Functional Testing

When determining how you should test your infrastructure, you need to understand the different kinds of testing, and then determine the kinds of testing on which you should focus more effort.

[95] http://docs.ansible.com/intro_windows.html

Unit testing, when applied to applications, is testing that applies to the smallest units of code (usually functions or class methods). In Ansible, unit testing would typically apply to individual playbooks. You could run individual playbooks in an isolated environment, but that's often not worth the effort. What *is* worth your effort is at least checking the playbook syntax, to make sure you didn't just commit a YAML file that will break an entire deployment because of a missing quotation mark, or a whitespace issue!

Integration testing, which is definitely more valuable when it comes to Ansible, is the testing of small groupings of individual units of code, to make sure they work correctly together. Breaking your infrastructure definition into many task-specific roles and playbooks allows you to do this; if you've structured your playbooks so they have no or limited dependencies, you could test each role individually in a fresh virtual machine, before you use the role as part of a full infrastructure deployment.

Functional testing involves the whole shebang. Basically, you set up a complete infrastructure environment, and then run tests against it to make sure *everything* was successfully installed, deployed, and configured. Ansible's own reporting is helpful in this kind of testing (and often, all that is necessary), and there are also external tools that can be used to test infrastructure even more deeply.

It is often possible to perform all the testing you need on your own local workstation, using Virtual Machines (as demonstrated in earlier chapters), using tools like VirtualBox or VMWare Workstation or Fusion. And with most cloud services providing robust control APIs and hourly billing, it's often simple, inexpensive, and just as fast to test directly on cloud instances that mirror your production infrastructure!

We'll begin with some of the simplest tests you can run against Ansible configuration, along with some common debugging techniques, then progress on to some more advanced, full-fledged functional testing methods using tools to fully automate the process.

Debugging and Asserting

TODO:

- `debug`
- `assert`

Checking syntax and performing dry runs

TODO:

- `ansible-playbook --syntax-check`
- `ansible-playbook --check`
- Ansible lint[96]

[96] https://github.com/willthames/ansible-lint

Automated testing on GitHub using Travis CI

Automated testing using a continuous integration tool like Travis CI (which is free for public projects and integrated very well with GitHub) allows you to run tests against Ansible playbooks or roles you have hosted on GitHub with every commit.

There are four main things that should be tested when building and maintaining Ansible playbooks or roles:

1. The playbook or role's syntax (are all the .yml files formatted correctly?).
2. Whether the playbook or role will run through all the included tasks without failing.
3. The playbook or role's idempotence (if run again, it should not make any changes!).
4. The playbook or role's success (does the role do what it should be doing?).

Ultimately, the most important aspect is #4, because what's the point of a playbook or role if it doesn't do what you want it to do (e.g. start a web server, configure a database, deploy an app, etc.)?

We're going to assume, for the rest of this example, that you're testing a role you have on GitHub, though the example can be applied just as easily for standalone Ansible playbooks.

Setting up a role for testing

Since you're going to need a simple Ansible playbook and inventory file to test your role, you can create both inside a new 'tests' directory in your Ansible role:

```
# Directory structure:
my_role/
  tests/
    test.yml <-- your test playbook
    inventory <-- an inventory file to use with the playbook
```

Inside the inventory file, add:

```
localhost
```

We just want to tell Ansible to run commands on the local machine (we'll use the –connection=local option when running the test playbook).

Inside `test.yml`, add:

```
---
- hosts: localhost
  remote_user: root
  roles:
    - github-role-project-name
```

Substitude your own role name for `github-role-project-name` (e.g. `ansible-role-django`). This is a typical Ansible playbook, and we tell Ansible to run the tasks on localhost, with the `root` user (otherwise, you could run tasks with `travis` if you want, and use `sudo` on certain tasks). You can add `vars`, `vars_files`, etc. if you want, but we'll keep things simple, because for many smaller roles, the role is pre-packaged with sane defaults and all the other info it needs to run.

The next step is to add a `.travis.yml` file to your role so Travis CI will pick it up and use it for testing. Add that file to the root level of your role, and add the following to kick things off:

```
---
language: python
python: "2.7"

before_install:
  # Make sure everything's up to date.
  - sudo apt-get update -qq

install:
  # Install Ansible.
  - pip install ansible

  # Add ansible.cfg to pick up roles path.
  - "printf '[defaults]\nroles_path = ../' > ansible.cfg"

script:
  # We'll add some commands to test the role here.
```

The only surprising part here is the `printf` line in the `install` section; I've added that line to create a quick and dirty `ansible.cfg` configuration file Ansible will use to set the `roles_path` one directory up from the current working directory. That way, we can include roles like `github-role-project-name`, or if we use `ansible-galaxy` to download dependencies (as another command in the install section), we can just use `- galaxy-role-name-here` to include that role in our `test.yml` playbook.

Now that we have the basic structure, it's time to start adding the commands to test our role.

Testing the role's syntax

This is the easiest test; `ansible-playbook` has a built in command that will check a playbook's syntax (including all the included files and roles), and return `0` if there are no problems, or an error code and some output if there were any syntax issues.

```
ansible-playbook -i tests/inventory tests/test.yml --syntax-check
```

Add this as a command in the `script` section of `.travis.yml`:

```
script:
  # Check the role/playbook's syntax.
  - ansible-playbook -i tests/inventory tests/test.yml --syntax-check
```

If there are any syntax errors, Travis will fail the build and output the errors in the log.

Role success - first run

The next aspect to check is whether the role runs correctly or fails on its first run.

```
# Run the role/playbook with ansible-playbook.
- "ansible-playbook -i tests/inventory tests/test.yml --connection=local --sudo"
```

This is a basic ansible-playbook command, which runs the playbook `test.yml` against the local host, using `--sudo`, and with the inventory file we added to the role's `tests` directory.

Ansible returns a non-zero exit code if the playbook run fails, so Travis will know whether the command succeeded or failed.

Role idempotence

Another important test is the idempotence test—does the role change anything if it runs a second time? It should not, since all tasks you perform via Ansible should be idempotent (ensuring a static/unchanging configuration on subsequent runs with the same settings).

```
# Run the role/playbook again, checking to make sure it's idempotent.
- >
  ansible-playbook -i tests/inventory tests/test.yml --connection=local --sudo
  | grep -q 'changed=0.*failed=0'
  && (echo 'Idempotence test: pass' && exit 0)
  || (echo 'Idempotence test: fail' && exit 1)
```

This command runs the exact same command as before, but pipes the results through grep, which checks to make sure 'changed' and 'failed' both report `0`. If there were no changes or failures, the idempotence test passes (and Travis sees the `0` exit and is happy), but if there were any changes or failures, the test fails (and Travis sees the `1` exit and reports a build failure).

Role success - final result

The last thing I check is whether the role actually did what it was supposed to do. If it configured a web server, is the server responding on port 80 or 443 without any errors? If it configured a command line application, does that command line application work when invoked, and do the things it's supposed to do?

```
# Request a page via the web server, to make sure it's running and responds.
- "curl http://localhost/"
```

In this example, I'm testing a web server by loading 'localhost'; curl will exit with a 0 status (and dump the output of the web server's response) if the server responds with a 200 OK status, or will exit with a non-zero status if the server responds with an error status (like 500) or is unavailable.

Taking this a step further, you could even run a deployed application or service's own automated tests after ansible is finished with the deployment, thus testing your infrastructure and application in one go—but we're getting ahead of ourselves here... that's a topic for later!

Some notes about Travis CI

There are a few things you need to know about Travis CI, especially if you're testing Ansible, which will rely heavily on the VM environment inside which it is running:

- **Ubuntu 12.04**: As of this writing, the only OS available via Travis CI is Ubuntu 12.04. Most of my roles work with Ubuntu/Debian/RedHat/CentOS, so it's not an issue for me... but if your roles strictly target a non-Debian-flavored distro, you probably won't get much mileage out of Travis.
- **Preinstalled packages**: Travis CI comes with a bunch of services installed out of the box, like MySQL, Elasticsearch, Ruby, etc. In the `.travis.yml` `before_install` section, you may need to do some `apt-get remove --purge [package]` commands and/or other cleanup commands to make sure the VM is fresh for your Ansible role's run.

- **Networking/Disk/Memory**: Travis CI continously shifts the VM specs you're using, so don't assume you'll have X amount of RAM, disk space, or network capacity. You can add commands like `cat /proc/cpuinfo`, `cat /proc/meminfo`, `free -m`, etc. in the `.travis.yml` `before_install` section if you need to figure out the resources available in your VM.

See much more information about the VM environment on the Travis CI Build Environment page[97].

Real-world examples

I have integrated this style of testing into many of the roles I've submitted to Ansible Galaxy; here are a few example roles that use Travis CI integration in the way I've outlined in this blog post:

- https://github.com/geerlingguy/ansible-role-apache
- https://github.com/geerlingguy/ansible-role-gitlab
- https://github.com/geerlingguy/ansible-role-mysql

Automated testing with `test-runner`

TODO:

- debops/test-runner[98]

Automated testing with Jenkins CI

TODO.

Functional testing using serverspec

TODO:

- server-spec[99]
- Caveat: Ansible already testing things as it goes. Do you really need another layer of testing?

Further notes on testing and Ansible

TODO:

- Integrating Testing With Ansible Playbooks[100]

[97] http://docs.travis-ci.com/user/ci-environment/
[98] https://github.com/debops/test-runner
[99] http://serverspec.org/
[100] http://docs.ansible.com/test_strategies.html

Appendix A - Using Ansible on Windows workstations

Ansible works primarily over the SSH protocol, which is supported natively by most every server, workstation, and operating system on the planet, with one exception—Microsoft's venerable Windows OS.

To use SSH on Windows, you need additional software. But Ansible also requires other utilities and subsystems only present on Linux or other UNIX-like operating systems. This poses a problem for many system administrators who are either forced to use or have chosen to use Windows as their primary OS.

This appendix will guide Windows users through the author's preferred method of using Ansible on a Windows workstation.

Ansible 1.7 and later can be used to manage Windows hosts (see Ansible's Windows Support[101] documentation) (but it can't be run from within Windows natively). You will still need to follow the instructions here to run the Ansible client on a Windows host, if you are stuck on Windows and want to use Ansible to manage other (Windows, Linux, Mac, etc.) hosts.

Prerequisites

Our goal is to have a virtual machine running Linux running on your computer. The easiest way to do this is to download and install Vagrant and VirtualBox (both 100% free!), and then use Vagrant to install Linux, and PuTTY to connect and use Ansible. Here are the links to download these applications:

1. Vagrant[102]
2. VirtualBox[103]
3. PuTTY[104]

Once you've installed all three applications, you can use either the command prompt (`cmd`), Windows PowerShell, or a Linux terminal emulator like Cygwin to boot up a basic Linux VM with Vagrant (if you use Cygwin, which is not covered here, you could install its SSH component and use it for SSH, and avoid using PuTTY).

[101] http://docs.ansible.com/intro_windows.html
[102] http://www.vagrantup.com/downloads.html
[103] https://www.virtualbox.org/
[104] http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Set up an Ubuntu Linux Virtual Machine

Open PowerShell (open the Start Menu or go to the Windows home and type in 'PowerShell'), and change directory to a place where you will store some metadata about the virtual machine you're about to boot. I like having a 'VMs' folder in my home directory to contain all my virtual machines:

```
# Change directory to your user directory.
PS > cd C:/Users/[username]
# Make a 'VMs' directory and cd to it.
PS > md -Name VMs
PS > cd VMs
# Make a 'Ubuntu64' directory and cd to it.
PS > md -Name ubuntu-precise-64
PS > cd ubuntu-precise-64
```

Now, use `vagrant` to create the scaffolding for our new virtual machine:

```
PS > vagrant init precise64 http://files.vagrantup.com/precise64.box
```

Vagrant creates a 'Vagrantfile' describing a basic Ubuntu Precise (12.04) 64-bit virtual machine in the current directory, and is now ready for you to run `vagrant up` to download and build the machine. Run `vagrant up`, and wait for the box to be downloaded and installed:

```
PS > vagrant up
```

After a few minutes, the box will be downloaded and a new virtual machine set up inside VirtualBox. Vagrant will boot and configure the machine according to the defaults defined in the Vagrantfile. Once the VM is booted, and you're back at the command prompt, it's time to log into the VM.

Log into the Virtual Machine

Use `vagrant ssh-config` to grab the SSH connection details, which you will then enter into PuTTY to connect to the VM.

```
PS > vagrant ssh-config
```

It should show something like:

```
Host default
  Hostname 127.0.0.1
  User vagrant
  Port 2222
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile C:/Users/[username]/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL
```

The lines we're interested in are the Hostname, User, Port, and IdentityFile.

Launch PuTTY, and enter the connection details:

- **Host Name (or IP address)**: 127.0.0.1
- **Port**: 2222

Click Open to connect (you can save the connection details by entering a name in the 'Saved Sessions' field and clicking 'Save' to save the details), and if you receive a Security Alert concerning the server's host key, click 'Yes' to tell PuTTY to trust the host.

PuTTY will ask for login credentials; we'll use the default login for a Vagrant box (`vagrant` for both the username and password):

```
login as: vagrant
vagrant@127.0.0.1's password: vagrant
```

You should now be connected to the virtual machine, and see the message of the day:

```
Welcome to Ubuntu 12.04 LTS (GN/Linux 3.2.0-23-generic x86_64)

  * Documentation: https://help.ubuntu.com/
Welcome to your Vagrant-built virtual machine.
Last login: <date> from <IP address>
vagrant@precise64:~$
```

If you see this prompt, you're logged in, and you can start administering the VM. The next (and final) step is to install Ansible.

This example uses PuTTY to log into the VM, but other applications like Cygwin[105] or Git for Windows[106] work just as well, and may be easier to use. Since these alternatives have built-in SSH support, you don't need to do any extra connection configuration, or even launch the apps manually; just `cd` to the same location as the Vagrantfile, and enter `vagrant ssh`!

Install Ansible

Before installing Ansible, make sure your package list is up to date by updating apt-get:

```
$ sudo apt-get update
```

Ansible can be installed in a variety of ways, but the easiest is to use `pip`, a simple Python package manager. Python should already be installed on the system, but `pip` may not be, so let's install it, along with Python's development header files (which are in the `python-dev` package).

```
$ sudo apt-get install -y python-pip python-dev
```

After the installation is complete, installing Ansible is simple:

```
$ sudo pip install ansible
```

After Ansible and all its dependencies are downloaded and installed, make sure Ansible is running and working:

```
$ ansible --version
ansible 1.8.0
```

Upgrading Ansible is also easy with pip: Run `sudo pip install --upgrade ansible` to get the latest version.

[105] http://cygwin.com/install.html
[106] http://git-scm.com/download/win

Summary

You should now have Ansible installed within a virtual machine running on your Windows workstation. You can control the virtual machine with Vagrant (`cd` to the location of the Vagrantfile), using `up` to boot or wake the VM, `halt` to shut down the VM, or `suspend` to sleep the VM. You can log into the VM using PuTTY and manually entering a username and password, or using Cygwin or Git's Windows shell and the `vagrant ssh` command.

Use Ansible from within the virtual machine just as you would on a Linux or Mac workstation directly. If you need to share files between your Windows environment and the VM, Vagrant conveniently maps `/vagrant` on the VM to the same folder where your Vagrantfile is located. You can also connect between the two via other methods (SSH, SMB, SFTP etc.) if you so desire.

Appendix B - Ansible Best Practices and Conventions

Ansible is a simple, flexible tool, and allows for a variety of organization methods and configuration syntaxes. You might like to have many tasks in one main file, or few tasks in many files. You might prefer defining variables in group variable files, host variable files, inventories, or elsewhere, or you might try to find ways of avoiding variables in inventories altogether.

There are few *universal* best practices in Ansible, but this appendix contains many helpful suggestions for organizing playbooks, writing tasks, using roles, and otherwise build infrastructure with Ansible.

TODO:

- Incorporate other commonly-asked-about best practices (ongoing).
- Best Practices[107]
- Ansible (Real Life) Good Practices[108]

Playbook Organization

As playbooks are Ansible's bread and butter, it's important to organize them in a logical manner, so you can easily write, debug, and maintain them.

Write comments and use `name` liberally

Many tasks you write will be fairly obvious when you write them, but less so six months later when you are making changes. Just like application code, Ansible playbooks should be documented, at least minimally, so you can spend less time familiarizing yourself with what a particular task is supposed to do, and more time fixing problems or extending your playbooks.

In YAML, you can write a comment by simply starting a line with a hash (`#`). If your comment spans multiple lines, start each line with `#`.

It's also a good idea to use a `name` for every task you write, besides the most trivial. If you're using the `git` module to check out a specific tag, use a `name` to indicate what repository you're using, why a tag instead of a commit hash, etc. This way, whenever your playbook is run, you'll see the comment you wrote and be assured what's going on.

[107] http://docs.ansible.com/playbooks_best_practices.html

[108] http://www.reinteractive.net/posts/167-ansible-real-life-good-practices

```
- hosts: all

  tasks:

    # This task takes up to five minutes and is required so we will have
    # access to the images used in our application.
    - name: Copy the entire file repository to the application.
      copy:
        src: ...
```

This advice assumes, of course, that your comments actually indicate what's happening in your playbooks! Generally, I use full sentences with a period for all comments and `name`s, but if you'd like to use a slightly different style, that's not an issue. Just try to be consistent, and remember that *bad comments are worse than no comments at all.*

Include related variables and tasks

If you find yourself writing a playbook that's over 50-100 lines and configures three or four different applications or services, it may help to separate each group of tasks into a separate file, and use `include` to place them in a playbook.

Additionally, variables are usually better left in their own file and included using `vars_files` rather than defined inline with a playbook.

```
- hosts: all

  vars_files:
    - vars/main.yml

  handlers:
    - include: handlers/handlers.yml

  tasks:
    - include: tasks/init.yml
    - include: tasks/database.yml
    - include: tasks/app.yml
```

Using a more hierarchical model like this allows you to see what your playbook is doing at a higher level, and also lets you manage each portion of a configuration or deployment separately. I generally split tasks into separate files once I reach 15-20 tasks in a given file.

Use Roles to bundle logical groupings of configuration

Along the same lines as using included files to better organize your playbooks and separate bits of configuration logically, Ansible roles can supercharge your ability to manage infrastructure well.

Using loosely-coupled roles to configure individual components of your servers (like databases, application deployments, the networking stack, monitoring packages, etc.) allows you to write configuration once, and use it on all your servers, regardless of their role.

Consider that you will probably configure something like NTP (Network Time Protocol) on every single server you manage, or at a minimum, set a timezone for the server. Instead of adding two or three tasks to every playbook you manage, set up a role (maybe call it `time` or `ntp`) that does this configuration, and use a few variables to allow different groups of servers to have customized settings.

Additionally, if you learn to build roles in a generic fashion, and for multiple platforms, you could even share it on Ansible Galaxy so others can use the role and help you make it more robust and efficient!

YAML Conventions and Best Practices

YAML is a human-readable, machine-parseable syntax that allows for almost any list, map, or array structure to be described using a few basic conventions, so it is a great fit for a configuration management tool. Consider the following method of defining a list (or 'collection') of widgets:

```
widget:
  - foo
  - bar
  - fizz
```

This would translate into Python (using the `PyYAML` library employed by Ansible) as the following:

```
translated_yaml = {'widget': ['foo', 'bar', 'fizz']}
```

And what about a structured list/map in YAML?

```
widget:
  foo: 12
  bar: 13
```

The Python that would result:

```
translated_yaml = {'widget': {'foo': 12, 'bar': 13}}
```

A few things to note with both of the above examples:

- YAML will try to determine the type of an item automatically. So `foo` in the first example would be translated as a string, `true` or `false` would be a boolean, and `123` would be an integer. You can read the official documentation for further insight, but for our purposes, realize you can minimize surprises by declaring strings with quotes (`''` or `""`).
- Whitespace matters! YAML uses spaces (literal space characters—*not* tabs) to define structure (mappings, array lists, etc.), so set your editor to use spaces for tabs. You can technically use either a tab or a space to delimit parameters (like `apt: name=foo state=installed`—you can use either a tab or a space between parameters), but it's generally preferred to use spaces everywhere, to minimize errors and display irregularities across editors and platforms.
- YAML syntax is robust and well-documented. Read through the official YAML Specification[109] and/or the PyYAMLDocumentation[110] to dig deeper.

YAML for Ansible tasks

Consider the following task:

```
- name: Install foo.
  apt: pkg=foo state=installed
```

All well and good, right? Well, as you get deeper into Ansible and start defining more complex configuration, you might start seeing tasks like the following:

```
- name: Copy Phergie shell script into place.
  template: src=templates/phergie.sh.j2 dest=/opt/phergie.sh owner={{ phergie_us\
er }} group={{ phergie_user }} mode=755
```

The one-line syntax (which uses Ansible-specific `key=value` shorthand for defining parameters) has some positive attributes:

- Simpler tasks (like installations and copies) are compact and readable (`apt: pkg=apache2 state=installed` is just about as simple as `apt-get install -y apache2`; in this way, an Ansible playbook feels very much like a shell script.
- Playbooks are more compact, and more configuration can be displayed on one screen.
- Ansible's official documentation follows this format, as do many existing roles and playbooks.

[109] http://www.yaml.org/spec/1.2/spec.html
[110] http://pyyaml.org/wiki/PyYAMLDocumentation

However, as highlighted in the above example, there are a few issues with this `key=value` syntax:

- Smaller monitors, terminal windows, and source control applications will either wrap or hide part of the task line.
- Diff viewers and source control systems generally don't highlight intra-line differences as well as full line changes.
- Variables and parameters are converted to strings, which may or may not be desired.

Ansible's shorthand syntax can be troublesome for complicated playbooks and roles, but luckily there are other ways you can write tasks which are better for narrower displays, version control software and diffing.

Three ways to format Ansible tasks

The following methods are most often used to define Ansible tasks in playbooks:

Shorthand/one-line (`key=value`)

Ansible's shorthand syntax uses `key=value` parameters after the name of a module as a key:

```
- name: Install Nginx.
  yum: pkg=nginx state=installed
```

For any situation where an equivalent shell command would roughly match what I'm writing in the YAML, I prefer this method, since it's immediately obvious what's happening, and it's highly unlikely any of the parameters (like `state=installed`) will change frequently during development.

Ansible's official documentation generally uses this syntax, so it maps nicely to examples you'll find from Ansible, Inc. and many other sources.

Structured map/multi-line (`key: value`)

You can define a structured map of parameters (using `key: value`, with each parameter on its own line) for a task:

```
- name: Copy Phergie shell script into place.
  template:
    src: "templates/phergie.sh.j2"
    dest: "/home/{{ phergie_user }}/phergie.sh"
    owner: "{{ phergie_user }}"
    group: "{{ phergie_user }}"
    mode: 0755
```

A few notes on this syntax:

- The structure is all valid YAML, and functions similarly to Ansible's shorthand syntax.
- Strings, booleans, integers, octals, etc. are all preserved (instead of being converted to strings).
- Each parameter *must* be on its own line, so you can't chain together `mode: 0755, owner: root, user: root` to save space.
- YAML syntax highlighting (if you have an editor that supports it) works slightly better for this format than `key=value`, since each key will be highlighted, and values will be displayed as constants, strings, etc.

Folded scalars/multi-line (›)

You can also use the › character to break up Ansible's shorthand `key=value` syntax over multiple lines.

```
- name: Copy Phergie shell script into place.
  template: >
    src=templates/phergie.sh.j2
    dest=/home/{{ phergie_user }}/phergie.sh
    owner={{ phergie_user }} group={{ phergie_user }} mode=755
```

In YAML, the › character denotes a *folded scalar*, where every line that follows (as long as it's indented further than the line with the ›) will be joined with the line above by a space. So the above YAML and the earlier `template` example will function exactly the same.

This syntax allows arbitrary splitting of lines on parameters, but it does not preserve value types (`0775` would be converted to a string, for example).

While this syntax is often seen in the wild, I don't recommend it except for certain situations, like tasks using the `command` and `shell` modules with extra options:

```
- name: Install Drupal.
  command: >
    drush si -y
    --site-name="{{ drupal_site_name }}"
    --account-name=admin
    --account-pass={{ drupal_admin_pass }}
    --db-url=mysql://root@localhost/{{ domain }}
    chdir={{ drupal_core_path }}
    creates={{ drupal_core_path }}/sites/default/settings.php
```

If you can find a way to run a command without having to use `creates` and `chdir`, or very long commands (which are arguably unreadable either in single *or* multiline format!), it's better to do that instead of this monstrosity.

Sometimes, though, the above is as good as you can do to keep unwieldy tasks sane.

Using `ansible-playbook`

Generally, running playbooks from your own computer or a central playbook runner is preferable to running Ansible playbooks locally (using `--connection=local`), since you can avoid installing Ansible and all its dependencies on the system you're provisioning. Because of Ansible's optimized use of SSH for connecting to remote machines, there is usually minimal difference in performance running Ansible locally or from a remote workstation (barring network flakiness or a high-latency connection).

Use Ansible Tower

If you are able to use Ansible Tower to run your playbooks, this is even better, as you'll have a central server running Ansible playbooks, logging output, compiling statistics, and even allowing a team to work together to build servers and deploy applications in one place.

Specify `--forks` for playbooks running on > 5 servers

If you are running a playbook on a large number of servers, consider increasing the number of `forks` Ansible uses to run tasks simultaneously. The default, `5`, means Ansible will only run a given task on 5 servers at a time. Consider increasing this to 10, 15, or however many connections your local workstation and ISP can handle—this can dramatically reduce the amount of time it takes a playbook to run.

Use Ansible's Configuration file

Ansible's main configuration file, in `/etc/ansible/ansible.cnf`, can contain a wealth of optimizations and customizations that help you run playbooks and ad-hoc tasks more easily, faster, or with better output than stock Ansible provides.

Read through the official documentation's Ansible Configuration File[111] page for details on options you can customize in `ansible.cnf`.

Summary

One of Ansible's strengths is its flexibility; there are often multiple 'right' ways of accomplishing your goals. I have chosen to use the methods I outlined above as they have proven to help me write and maintain a variety of playbooks and roles with minimal headaches.

It's perfectly acceptable to try a different approach; as with most programming and technical things, being *consistent* is more important than following a particular set of rules, especially if that set of rules isn't universally agreed upon. Consistency is especially important when you're not working solo—if every team member used Ansible in a different way, it would become difficult to share work very quickly!

[111] http://docs.ansible.com/intro_configuration.html

Appendix C - Jinja2 and Ansible

TODO:

- Jinja2 Docs[112]
- Using Variables: About Jinja2[113]

Summary

TODO.

[112] http://jinja.pocoo.org/docs/

[113] http://docs.ansible.com/playbooks_variables.html#using-variables-about-jinja2

Glossary

This glossary is by no means comprehensive, but it contains a short summary of most of the terms you'll encounter when using or discussing Ansible. Ansible's official documentation site has an excellent Glossary[114] as well, and should always be consulted for the latest and most canonical definitions.

Accelerated Mode

TODO.

Action

TODO.

Ad-Hoc Task

TODO.

Check Mode

TODO.

Conditional

TODO.

Diff Mode

TODO.

[114] http://docs.ansible.com/glossary.html

DSL

TODO.

Facts (see also: Variables)

TODO.

Forks

TODO.

Group

TODO.

Handler

TODO.

Host (see also: Node)

TODO.

Idempotency

TODO.

Include

TODO.

Inventory

TODO.

Jinja2

TODO.

JSON

TODO.

Limit

TODO.

Local Action

TODO.

Module

TODO.

Node

TODO.

Notify

TODO.

Paramiko

TODO.

Play

TODO.

Playbook

TODO.

Pull Mode

TODO.

Push Mode

TODO.

Role

TODO.

Rolling Update

TODO.

Serial (see also: Rolling Update)

TODO.

Sudo

TODO.

SSH

TODO.

Tag

TODO.

Task

TODO.

Template

TODO.

When

TODO.

Variables (see also: Facts)

TODO.

YAML

TODO.

Changelog

This log will track changes between releases of the book. Until version 1.0, each release number will correlate to the amount complete, so 'Version 0.33' equals 33% complete. After the final, complete book, major version numbers will track editions.

Hopefully these notes will help readers figure out what's changed since the last time they've downloaded the book.

Version 0.60 (2014-09-30)

- Wrote most of appendix b (Best Practices).
- Updated definition of idempotence in chapter 1.
- Fixed a few LeanPub-related code formatting issues.
- Many grammar fixes throughout the book (thanks to Jon Forrest!).
- Some spelling and ad-hoc command fixes (thanks to Hugo Posca!).
- Had a baby (thus the dearth of updates from 8/1-10/1 :-).
- Wrote introduction and basic structure of chapter 11 (Ansible Tower).

Version 0.58 (2014-08-01)

- Even more sections on variables in chapter 5 (almost finished!).
- Fixed a few old Ansible and Drupal version references.
- Added a playbook include tag example in chapter 6.
- Completed first draft of chapter 6.
- Fixed broken handler in chapter 4's Tomcat handler (thanks to Joel Shprentz!).
- Fixed a missing closing quotation in chapter 3 example (thanks to Jonathan Nakatsui!).

Version 0.56 (2014-07-20)

- Filled in many more sections on variables in chapter 5.
- Some editing in chapter 6.
- Side work on some supplemental material for a potential chapter on Docker.

Version 0.54 (2014-07-02)

- Finished roles section in chapter 6.
- Fixed a few code examples for better style in chapter 4.
- Fixed references to official code repository in chapters 4 and 6.

Version 0.53 (2014-06-28)

- Added note about Windows Support[115] in appendix a.
- Wrote large portion of roles section in chapter 6.

Version 0.52 (2014-06-14)

- Adjusted some code listings to make more readable line breaks.
- Added section on Ansible testing with Travis CI in chapter 12.
- Expanded mention of Ansible's excellent documentation in introduction.
- Greatly expanded security coverage in chapter 10.
- Added link to security role on Ansible Galaxy in chapter 10.

Version 0.50 (2014-05-05)

- Wrote includes section in chapter 6.
- Added links to code repository examples in chapters 4 and 6.
- Fixed broken internal links.
- Fixed typos in chapter 10.
- Added note about `--force-handlers` (new in Ansible 1.6) in chapter 4.
- Use Ansible's `apache2_module` module for LAMP example in chapter 4.
- Moved Jinja2 chapter to appendix c.
- Removed 'Variables' chapter (variables will be covered in-depth elsewhere).
- Added Appendix B - Ansible Best Practices and Conventions.
- Started tagging code in Ansible for DevOps GitHub repository[116] to match manuscript version (starting with this version, 0.50).
- Fixed various layout issues.

[115] http://docs.ansible.com/intro_windows.html
[116] https://github.com/geerlingguy/ansible-for-devops

Version 0.49 (2014-04-24)

- Completed history of SSH in chapter 10.
- Clarified definition of the word 'DevOps' in chapter 1.
- Added section "Testing Ansible Playbooks" in chapter 14.
- Added links to Ansible for DevOps GitHub repository[117] in the introduction and chapter 4.

Version 0.47 (2014-04-13)

- Added Apache Solr example in chapter 4.
- Updated VM diagrams in chapter 4.
- Added information about `ansible-playbook` command in chapter 4 (thanks to a reader's suggestion!).
- Clarified code example in preface.

Version 0.44 (2014-04-04)

- Expanded chapter 10 (security).
- Fixed formatting issues in Warning/Info/Tip asides.
- Fixed formatting of some code examples to prevent line wrapping.
- Added section on Ansible Galaxy in chapter 6.
- Updated installation section in chapter 1 with simplified install processes.
- Added warnings concerning faster SSH in Ansible 1.5+ (thanks to @LeeVanSteerthem[118]).

Version 0.42 (2014-03-25)

- Added history of SSH section.
- Expanded chapter 10 (security).
- Many small spelling and grammar mistakes corrected.
- Fixed formatting of info/warning/tip asides.

[117] https://github.com/geerlingguy/ansible-for-devops
[118] https://twitter.com/LeeVanSteerthem

Version 0.38 (2014-03-11)

- Added Appendix A - Using Ansible on Windows workstations (thanks to a reader's suggestion!).
- Updated chapter 1 to include a reference to appendix a.
- Clarified and expanded installation instructions for Mac and Linux in chapter 1.
- Added chapter 10 - Server Security and Ansible
- Updated chapter 1 to include a reference to chapter 10.
- Added notes and TODOs to a few more areas of the book (random).

Version 0.35 (2014-02-25)

- Added this changelog.
- Split out roles and playbook organization into its own chapter.
- Expanded 'Environment Variables' section in chapter 5.
- Expanded 'Variables' section in chapter 5.
- MORE COWBELL! (Cowsay motivational quotes at the end of every completed chapter).
- Fixed NTP installation examples in chapter 2 (thanks to a reader's suggestion!).

Version 0.33 (2014-02-20)

- Initial published release, up to chapter 4, part of chapter 5.