Ansible for DevOps

Server and configuration management for humans

Jeff Geerling

Ansible for DevOps

Server and configuration management for humans

Jeff Geerling

ISBN 978-0-9863934-0-2

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

This book is dedicated to my wife, Natalie, and my children.

Editing by Margie Newman and Katherine Geerling.

Contents

Foreword

Over the last few years, Ansible has rapidly become one of the most popular IT automation tools in the world. We've seen the open source community expand from the beginning of the project in early 2012 to over 1200 individual contributors today. Ansible's modular architecture and broad applicability to a variety of automation and orchestration problems created a perfect storm for hundreds of thousands of users worldwide.

Ansible is a general purpose IT automation platform, and it can be used for a variety of purposes. From configuration management: enforcing declared state across your infrastructure, to procedural application deployment, to broad multi-component and multi-system orchestration of complicated interconnected systems. It is agentless, so it can coexist with legacy tools, and it's easy to install, configure, and maintain.

Ansible had its beginnings in 2012, when Michael DeHaan, the project's founder, took inspiration from several tools he had written prior, along with some hands-on experience with the state of configuration management at the time, and launched the project in February of 2012. Some of Ansible's unique attributes like its module-based architecture and agentless approach quickly attracted attention in the open source world.

In 2013, Said Ziouani, Michael DeHaan, and I launched Ansible, Inc. We wanted to harness the growing adoption of Ansible in the open source world, and create products to fill the gaps in the IT automation space as we saw them. The existing tools were complicated, error-prone, and hard to learn. Ansible gave users across an IT organization a low barrier of entry into automation, and it could be deployed incrementally, solving as few or as many problems as the team needed without a big shift in methodology.

This book is about using Ansible in a DevOps environment. I'm not going to try to define what DevOps is or isn't, or who's doing it or not. My personal interpretation of the idea is that DevOps is meant to shorten the distance between the developers writing the code, and the operators running the application. Now, I don't believe

adding a new "DevOps" team in between existing development and operations teams achieves that objective! (Oops, now I'm trying for a definition, aren't I?)

Well, definitions aside, one of the first steps towards a DevOps environment is choosing tools that can be consumed by both developers and operations engineers. Ansible is one of those tools: you don't have to be a software developer to use it, and the playbooks that you write can easily be self-documenting. There have been a lot of attempts at "write once, run anywhere" models of application development and deployment, but I think Ansible comes the closest to providing a common language that's useful across teams and across clouds and different datacenters.

The author of this book, Jeff, has been a long-time supporter, contributor, and advocate of Ansible, and he's maintained a massive collection of impressive Ansible roles in Galaxy, the public role-sharing service maintained by Ansible, Inc. Jeff has used Ansible extensively in his professional career, and is eminently qualified to write to the end-to-end book on Ansible in a DevOps environment.

As you read this book, I hope you enjoy your journey into IT automation as much as we have. Be well, do good work, and automate everything.

Tim Gerla
Ansible, Inc. Co-Founder & CTO

Preface

Growing up, I had access to a world that not many kids ever get to enter. At the local radio stations where my dad was chief engineer, I was fortunate to get to see networks and IT infrastructure up close: Novell servers and old Mac and Windows workstations in the '90s; Microsoft and Linux-based servers; and everything in between. Best of all, he brought home decommissioned servers and copies of Linux burned to CD.

I began working with Linux and small-scale infrastructures before I started high school, and my passion for infrastructure grew as I built a Cat5 wired network and a small rack of networking equipment for a local grade school. 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 that 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. The time I invest in learning new infrastructure 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 these pages, I'll 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, 2015

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 1-100 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, this book should provide you with an intermediate- to expert-level understanding of Ansible and how you can use it to manage your infrastructure.

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:

```
1 ---
2 # This is the beginning of a YAML file.
```

Some lines of YAML and other code examples require more than 70 characters per line, resulting in the code wrapping to a new line. Wrapping code is indicated by a `\` at the end of the line of code. For example:

```
1 # The line of code wraps due to the extremely long URL.
2 wget http://www.example.com/really/really/really/long/path/in/the/ur\
3 l/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 to avoid them.

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!

New revisions of this book are published on a regular basis (you're reading version 1.13). If you think a particular section needs improvement or find something missing, please contact me via Twitter (@geerlingguy[2]), a comment on this book's Feedback page on LeanPub[3], or whatever method is convenient for you.

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

[2] https://twitter.com/geerlingguy

[3] https://leanpub.com/ansible-for-devops/feedback

All known issues with Ansible for DevOps will be aggregated on the book's online Errata[4] page.

About the Author

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

[4] https://www.ansiblefordevops.com/errata

Introduction

In the beginning, there were sysadmins

Since the beginning of networked computing, deploying and managing servers reliably and efficiently has been a challenge. Historically, system administrators were walled off from the developers and users who interact with the systems they administer, and they managed servers by hand, installing software, changing configurations, and administering services on individual servers.

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

Server virtualization brought large-scale infrastructure management to the fore, and the number of servers managed by one admin (or by a small team of admins), has grown by an order of magnitude. Instead of deploying, patching, and destroying every server by hand, admins now are expected to bring up new servers, either automatically or with minimal intervention. Large-scale IT deployments now may involve hundreds or thousands of servers; in many of the largest environments, server provisioning, configuration, and decommissioning are fully 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. In many companies, development and operations work is integrated. Indeed, this integration is a requirement for modern test-driven application design.

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 now as DevOps

or Reliability Engineering. 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 is spent doing 'fun' activities like performance tuning, experimentation, and getting things done, and less time 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 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. My advice: 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[5] 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 Red Hat (where he worked on Cobbler[6], a configuration management tool and Func[7], a tool for communicating commands to remote servers), and some other projects[8], he experienced the trials and tribulations of many different organizations and individual sysadmins on their quest to simplify and automate their infrastructure management operations.

[5] https://twitter.com/laserllama

[6] http://www.cobblerd.org/

[7] https://fedorahosted.org/func/

[8] https://www.ansible.com/blog/2013/12/08/the-origins-of-ansible

Additionally, Michael found many shops were using separate tools[9] 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, Ansible is extensible with modules written in a programming language you already know.
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 Led Zeppelin songs (e.g. 2.0 was named after 1973's "Over the Hills and Far Away", 1.x releases were named after Van Halen songs). 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.[10] was founded by Saïd Ziouani (@SaidZiouani[11] on Twitter) and Michael DeHaan, and oversees core Ansible development and provides services (such as Automation Jump Start[12]) and extra tooling (such as Ansible Tower[13]) to organizations using Ansible. Hundreds of individual developers have contributed

[9] http://highscalability.com/blog/2012/4/18/ansible-a-simple-model-driven-configuration-management-and-c.html
[10] http://www.ansible.com/
[11] https://twitter.com/SaidZiouani
[12] http://www.ansible.com/services
[13] https://www.ansible.com/tower

patches to Ansible, and Ansible is the most starred infrastructure management tool on GitHub (with over 10,000 stars as of this writing).

In October 2015, Red Hat acquired Ansible, Inc., and has proven itself to be a good steward and promoter of Ansible. 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[14], 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[15] - Covers all Ansible options in depth. There are few open source projects with documentation as clear and thorough.
- Ansible Glossary[16] - If there's ever a term in this book you don't seem to fully understand, check the glossary.
- Ansible Mailing List[17] - Discuss Ansible and submit questions with Ansible's community via this Google group.
- Ansible on GitHub[18] - The official Ansible code repository, where the magic happens.

[14] https://github.com/geerlingguy/ansible-for-devops
[15] https://docs.ansible.com/ansible/
[16] https://docs.ansible.com/ansible/glossary.html
[17] https://groups.google.com/forum/#!forum/ansible-project
[18] https://github.com/ansible/ansible

- Ansible Example Playbooks on GitHub[19] - Many examples for common server configurations.
- Getting Started with Ansible[20] - A simple guide to Ansible's community and resources.
- Ansible Blog[21]

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. This book is meant as a supplement to, not a replacement for, Ansible's documentation!

[19] https://github.com/ansible/ansible-examples
[20] https://www.ansible.com/get-started
[21] https://www.ansible.com/blog

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 a 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 or application containers. 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 good deal of time going through all of the installed packages, documenting configurations, versions, and settings; and they would spend a lot of unnecessary time manually reinstalling, updating, and tweaking everything to get the new 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[22], Puppet[23] and Chef[24] became very popular in the mid-to-late 2000s.

But there's a reason why many developers and sysadmins stick to shell scripting and command-line configuration: it's simple and 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. Ansible also has 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.

[22] http://cfengine.com/

[23] http://puppetlabs.com/

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

Idempotence is the ability to run an operation which produces the same result whether run once or multiple times (source[25]).

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 deploys 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:

1. Install Homebrew[26] (get the installation command from the Homebrew website).
2. Install Python 2.7.x (`brew install python`).
3. Install Ansible (`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:

[25] http://en.wikipedia.org/wiki/Idempotence#Computer_science_meaning
[26] http://brew.sh/

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[27] environment. For setup instructions, please see my blog post Running Ansible within Windows[28]), 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.

If you have `python-pip` and `python-devel` (`python-dev` on Debian/Ubuntu) installed, 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/Red Hat Enterprise Linux/CentOS:

The easiest way to install Ansible on a Fedora-like system is to use the official yum package. If you're running Red Hat Enterprise Linux (RHEL) or CentOS, you need to install EPEL's RPM before you install Ansible (see the info section below for instructions):

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

[27] http://cygwin.com/

[28] https://servercheck.in/blog/running-ansible-within-windows

On RHEL/CentOS systems, `python-pip` and `ansible` are available via the EPEL repository[29]. 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 commands:

```
# If you're on RHEL/CentOS 6:
$ rpm -ivh http://dl.fedoraproject.org/pub/epel/6/x86_64/\
epel-release-6-8.noarch.rpm
# If you're on RHEL/CentOS 7:
$ yum install epel-release
```

Debian/Ubuntu:

The easiest way to install Ansible on a Debian or Ubuntu system is to use the official apt package.

```
$ sudo apt-add-repository -y ppa:ansible/ansible
$ sudo apt-get update
$ sudo apt-get install -y ansible
```

If you get an error like "sudo: add-apt-repository: command not found", you're probably missing the `python-software-properties` package. Install it with the command:

```
$ sudo apt-get install python-software-properties
```

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:

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

```
$ ansible --version
ansible 2.1.0.0
```

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:

```
1 [example]
2 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.

This first example assumes you have a server set up that you can test with; if you don't already have a spare server somewhere that you can connect to, you might want to create a small VM using DigitalOcean, Amazon Web Services, Linode, or some other service that bills by the hour. That way you have a full server environment to work with when learning Ansible—and when you're finished testing, delete the server and you'll only be billed a few pennies!

Replace the `www.example.com` in the above example with the name or IP address of your server.

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, add the `--ask-pass` (`-k`) flag to Ansible commands (you may also need to install the `sshpass` package for this to work). 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.

Need a primer on SSH key-based authentication? Please read through Ubuntu's community documentation on SSH/OpenSSH/Keys[30].

Let's run a more useful command:

[30] https://help.ubuntu.com/community/SSH/OpenSSH/Keys

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

In this example, we quickly see memory usage (in a human readable format) on all the servers (for now, just one) in the `example` group. Commands like this are helpful for 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[31], Munin[32], or Cacti[33], it's also nice to check these stats on all your servers with one simple command and one terminal window!

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 |
                ||     ||
```

[31] http://www.nagios.org/

[32] http://munin-monitoring.org/

[33] http://www.cacti.net/

Chapter 2 - Local Infrastructure Development: Ansible and Vagrant

Prototyping and testing with local virtual machines

Ansible works well 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 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; there are now systems that integrate both with Ansible and with 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 approach 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 to 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—with no one the wiser.

Vagrant[34], a server provisioning tool, and VirtualBox[35], 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 for easy testing with Ansible to provision a new server.

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 (whichever version is appropriate for your OS): - Download Vagrant[36] - Download VirtualBox[37] (when installing, make sure the command line tools are installed, so Vagrant 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, then navigate to the folder you just created.
4. Add a CentOS 7.x 64-bit 'box' using the `vagrant box add`[38] command: `vagrant box add geerlingguy/centos7` (note: HashiCorp's Atlas[39] has a comprehen-

[34] http://www.vagrantup.com/
[35] https://www.virtualbox.org/
[36] http://www.vagrantup.com/downloads.html
[37] https://www.virtualbox.org/wiki/Downloads
[38] http://docs.vagrantup.com/v2/boxes.html
[39] https://atlas.hashicorp.com/boxes/search

sive list of different pre-made Linux boxes. Also, check out the 'official' Vagrant Ubuntu boxes in Vagrant's Boxes documentation[40].

5. Create a default virtual server configuration using the box you just downloaded: `vagrant init geerlingguy/centos7`
6. Boot your CentOS server: `vagrant up`

Vagrant has downloaded a pre-built 64-bit CentOS 7 virtual machine (you can build your own[41] 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:

- **Network interface management**[42]: 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**[43]: Vagrant sets up shares between your host machine and VMs using NFS or (much slower) native folder sharing in VirtualBox.

[40] https://www.vagrantup.com/docs/boxes.html
[41] https://www.vagrantup.com/docs/virtualbox/boxes.html
[42] https://www.vagrantup.com/docs/networking/index.html
[43] https://www.vagrantup.com/docs/synced-folders/index.html

- **Multi-machine management**[44]: Vagrant is able to configure and control multiple VMs within one Vagrantfile. This is important because, as 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 behaves far differently."
- **Provisioning**[45]: 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  end
```

This is a very basic configuration to get you started using Ansible with Vagrant. There are many other Ansible options[46] 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.

[44] https://www.vagrantup.com/docs/multi-machine/index.html
[45] https://www.vagrantup.com/docs/provisioning/index.html
[46] https://www.vagrantup.com/docs/provisioning/ansible.html

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:

```
---
- hosts: all
  become: yes
  tasks:
  - name: Ensure NTP (for time synchronization) is installed.
    yum: name=ntp state=present
  - 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 a marker showing that the rest of the document will be formatted in YAML (read a getting started guide for YAML[47]).

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

```
2 - 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.

```
3   become: yes
```

Since we need privileged access to install NTP and modify system configuration, this line tells Ansible to use `sudo` for all the tasks in the playbook (you're telling Ansible to 'become' the root user with `sudo`, or an equivalent).

```
4   tasks:
```

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

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

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:

```
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 for Debian-flavored Linux, or `package` for OS-agnostic package installation).

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

This final task both checks and ensures that 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
    systemctl start ntpd.service > /dev/null
    echo "Started ntpd."
fi
# Make sure ntpd is enabled on system startup.
systemctl enable ntpd.service
```

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 more 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    become: yes
4    tasks:
5    - yum: name=ntp state=present
6    - 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 tasks) is not absolutely necessary. However, I'm a firm believer in thorough (but concise) documentation, so I almost always document what my tasks 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.

Cleaning Up

Once you're finished experimenting with the CentOS Vagrant VM, you can remove it from your system by running `vagrant destroy`. If you want to rebuild the VM again, run `vagrant up`. If you're like me, you'll soon be building and rebuilding hundreds of VMs and containers per week using Vagrant and Ansible!

Summary

Your workstation is on the path to 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. With one small example, you've got a glimpse at the simple-yet-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 used 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; for now, we'll explore how Ansible helps 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 servers in a streamlined fashion.

On any given day, a systems administrator has many tasks:

- 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 of commands (`ping` and `free -m`) on a server that 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.

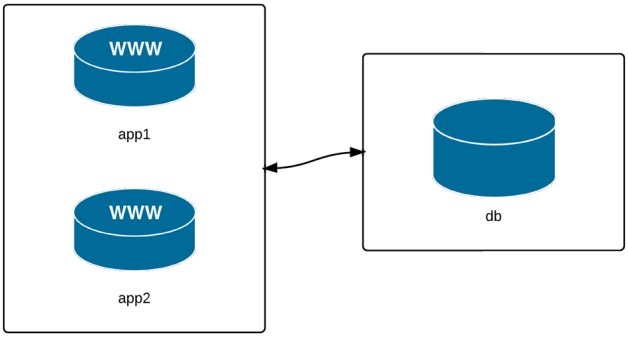
Some of the examples in this chapter will display how you can configure certain aspects of a server with ad-hoc commands. It is usually more appropriate to contain all configuration within playbooks and templates, so it's easier to provision your servers (running the playbook the first time) and then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

Earlier, we used Vagrant to boot up one virtual machine running CentOS 7. In that example, we used all of Vagrant's default configuration defined in the Vagrantfile. 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.

To begin, 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, add the following, and save the file:

```
1  # -*- mode: ruby -*-
2  # vi: set ft=ruby :
3
4  VAGRANTFILE_API_VERSION = "2"
5
6  Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
7    # General Vagrant VM configuration.
8    config.vm.box = "geerlingguy/centos7"
9    config.ssh.insert_key = false
10   config.vm.synced_folder ".", "/vagrant", disabled: true
11   config.vm.provider :virtualbox do |v|
12     v.memory = 256
13     v.linked_clone = true
14   end
15
```

```
16    # Application server 1.
17    config.vm.define "app1" do |app|
18      app.vm.hostname = "orc-app1.dev"
19      app.vm.network :private_network, ip: "192.168.60.4"
20    end
21
22    # Application server 2.
23    config.vm.define "app2" do |app|
24      app.vm.hostname = "orc-app2.dev"
25      app.vm.network :private_network, ip: "192.168.60.5"
26    end
27
28    # Database server.
29    config.vm.define "db" do |db|
30      db.vm.hostname = "orc-db.dev"
31      db.vm.network :private_network, ip: "192.168.60.6"
32    end
33  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 7.

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 box while building the example from Chapter 2, this process shouldn't take too long—maybe 3-5 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:

```
# Lines beginning with a # are comments, and are only included for
# illustration. These comments are overkill for most inventory files.

# Application servers
[app]
192.168.60.4
192.168.60.5

# 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 to 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 to 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).

If Ansible reports `No hosts matched` or returns some other inventory-related error, try setting the `ANSIBLE_HOSTS` environment variable explicitly: `export ANSIBLE_HOSTS=/etc/ansible/hosts`. Generally Ansible will read the file in `/etc/ansible/hosts` automatically, but depending on how you installed Ansible, you may need to explicitly set `ANSIBLE_HOSTS` for the `ansible` command to work correctly.

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

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

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

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

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

192.168.60.4 | success | rc=0 >>
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 use only one fork (basically, to 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
```

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 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"
192.168.60.6 | success | rc=0 >>
Filesystem               Size  Used Avail Use% Mounted on
/dev/mapper/centos-root   19G 1014M   18G   6% /
devtmpfs                 111M     0  111M   0% /dev
tmpfs                    120M     0  120M   0% /dev/shm
tmpfs                    120M  4.3M  115M   4% /run
tmpfs                    120M     0  120M   0% /sys/fs/cgroup
/dev/sda1                497M  124M  374M  25% /boot
none                     233G  217G   17G  94% /vagrant

192.168.60.5 | success | rc=0 >>
Filesystem               Size  Used Avail Use% Mounted on
/dev/mapper/centos-root   19G 1014M   18G   6% /
devtmpfs                 111M     0  111M   0% /dev
tmpfs                    120M     0  120M   0% /dev/shm
tmpfs                    120M  4.3M  115M   4% /run
tmpfs                    120M     0  120M   0% /sys/fs/cgroup
/dev/sda1                497M  124M  374M  25% /boot
none                     233G  217G   17G  94% /vagrant

192.168.60.4 | success | rc=0 >>
Filesystem               Size  Used Avail Use% Mounted on
/dev/mapper/centos-root   19G 1014M   18G   6% /
devtmpfs                 111M     0  111M   0% /dev
tmpfs                    120M     0  120M   0% /dev/shm
tmpfs                    120M  4.3M  115M   4% /run
tmpfs                    120M     0  120M   0% /sys/fs/cgroup
/dev/sda1                497M  124M  374M  25% /boot
none                     233G  217G   17G  94% /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"
192.168.60.4 | success | rc=0 >>
             total       used       free     shared    buffers     cached
Mem:           238        187         50          4          1         69
-/+ buffers/cache:        116        121
Swap:         1055          0       1055

192.168.60.6 | success | rc=0 >>
             total       used       free     shared    buffers     cached
Mem:           238        190         47          4          1         72
-/+ buffers/cache:        116        121
Swap:         1055          0       1055

192.168.60.5 | success | rc=0 >>
             total       used       free     shared    buffers     cached
Mem:           238        186         52          4          1         67
-/+ buffers/cache:        116        121
Swap:         1055          0       1055
```

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 2021

192.168.60.4 | success | rc=0 >>
Sat Feb  1 20:23:08 UTC 2021

192.168.60.6 | success | rc=0 >>
Sat Feb  1 20:23:08 UTC 2021
```

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 for a group of servers), use the command `ansible [host-or-group] -m setup`. This will provide 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 to 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 "name=ntp state=present"
```

You should see three simple 'success' messages, reporting no change, since NTP was already installed on the three machines; this confirms 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 multi -m shell -a "date"`), 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 (e.g. all of them at least have the correct time), 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 state=present"
```

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` does), 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 >>
1.10b1

192.168.60.5 | success | rc=0 >>
1.10b1
```

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 demonstrates 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 MariaDB, start it, and configure the server's firewall to allow access on MariaDB's default port, 3306.

```
$ ansible db -s -m yum -a "name=mariadb-server state=present"
$ ansible db -s -m service -a "name=mariadb 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 MariaDB 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 MariaDB server with its assorted `mysql_*` modules. For now, we need to allow MySQL access for one user from our app servers. The MySQL modules require the `MySQL-python` module to be present on the managed server.

Why MariaDB and not MySQL? RHEL 7 and CentOS 7 have MariaDB as the default supported MySQL-compatible database server. Some of the tooling around MariaDB still uses the old 'MySQL*' naming syntax, but if you're used to MySQL, things work similarly with MariaDB.

```
$ 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 that 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 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; being able to match on regular expressions is often 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 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[48] explains all the possibilities in great detail.

Manage packages

We've already used the `yum` module on our example CentOS infrastructure to ensure certain packages are installed. Ansible has a variety of package management modules for any flavor of Linux, but there's also a generic `package` module that can be used for easier cross-platform Ansible usage.

If you want to install a generic package like `git` on any Debian, RHEL, Fedora, Ubuntu, CentOS, FreeBSD, etc. system, you can use the command:

```
$ ansible app -s -m package -a "name=git state=present"
```

`package` works much the same as `yum`, `apt`, and other package management modules. Later in the book we'll explore ways of dealing with multi-platform package management where package names differ between OSes.

[48] 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 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 more advanced file copy modules like `rsync`, 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` or `rsync` modules.

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 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 starts the command *very* quickly on all your servers (especially if you set a higher `--forks` value), then polls 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": "2021-02-09 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 -s -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] ...

$
```

Running the command this way doesn't allow status updates via `async_status` and a jid, but you can still inspect the file `~/.ansible_async/<jid>` on the remote server. This option is usually more 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 tasks in Ansible playbooks in the background, asynchronously, by defining an `async` and `poll` parameter on the play. We'll discuss playbook task backgrounding 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., 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 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 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, 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 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.

If you get a message saying "Failed to find required executable git", you will need to install Git on the server. To do so, run the ad-hoc command `ansible package -s -m yum -a "name=git state=present"`.

If you get a message saying the Git server has an "unknown hostkey", add the option `accept_hostkey=yes` to the command, or add the hostkey to your server's `known_hosts` file before running this command.

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 achieves greater performance for playbooks. Instead of connecting repeatedly via SSH, Ansi-

ble 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 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 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, either compile from source, or use a different repository (like CentALT[49] 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 to use the `ansible` command to perform tasks on many servers quickly in parallel, or one by one.

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

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 |
                ||     ||
```

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. In American football, a team follows a set of pre-written playbooks as the basis for a bunch of plays they execute to try to win a game. In Ansible, you write playbooks (a list of instructions describing the steps to bring your server to a certain configuration state) that are then *play*ed on your servers.

Playbooks are written in YAML[50], a simple human-readable syntax popular for defining configuration. Playbooks may be included within other playbooks, and certain metadata and options cause different plays or playbooks to be run in different scenarios on different servers.

Ad-hoc commands alone 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 installs Apache on a RHEL/CentOS server:

Shell Script

[50] https://docs.ansible.com/ansible/YAMLSyntax.html

```
# Install Apache.
yum install --quiet -y httpd httpd-devel
# Copy configuration files.
cp httpd.conf /etc/httpd/conf/httpd.conf
cp 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:

```
# (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 httpd.conf /etc/httpd/conf/httpd.conf
    - command: >
        cp 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 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 as you get time, rebuild your configuration to take advantage of Ansible's helpful features.

In the above playbook, we use Ansible's `command` module to run standard shell commands. We're also giving each task a 'name', so when we run the playbook, the task 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, be assured shell scripts are 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 in some cases. There are different ways of describing configuration using valid YAML syntax, and these methods are discussed in-depth in the YAML Conventions and Best Practices section in Appendix B.

This book uses three different task-formatting techniques: For tasks which require one or two simple parameters, Ansible's shorthand syntax (e.g. `yum: name=apache2 state=present`) is used. For most uses of `command` or `shell`, where longer commands are entered, the `>` technique mentioned above is used. For tasks which require many parameters, YAML object notation is used—placing each key and variable on its own line. This assists with readability and allows for version control systems to easily distinguish changes line-by-line.

The above playbook will perform *exactly* like the shell script, but you 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
  become: yes

  tasks:
    - name: Install Apache.
      yum: name={{ item }} state=present
      with_items:
        - httpd
        - httpd-devel
    - name: Copy configuration files.
      copy:
        src: "{{ item.src }}"
        dest: "{{ item.dest }}"
        owner: root
        group: root
        mode: 0644
      with_items:
        - src: "httpd.conf"
          dest: "/etc/httpd/conf/httpd.conf"
        - src: "httpd-vhosts.conf"
          dest: "/etc/httpd/conf/httpd-vhosts.conf"
    - name: Make sure Apache is started now and 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, `- hosts: all` defines the first (and in this case, only) *play*, and tells Ansible to run the play on `all` hosts that it knows about.
3. The third line, `become: yes` tells Ansible to run all the commands through `sudo`, so the commands will be run as the root user.

4. The fifth line, `tasks:`, tells Ansible that what follows is a list of tasks to run as part of this playbook.
5. 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 name=httpd state=present"... 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` to make sure the package is *not* installed.
    - Ansible allows simple lists to be passed into tasks using `with_items`: 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.
6. 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 could 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 use a dot to access the variable within the item, so `{{ item.var1 }}` would access the first variable. In our example, `item.src` accesses the `src` in each item.
7. 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`).

With this playbook format, 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 (if the server is in the correct state), it won't actually do anything besides tell you nothing has changed. So, with this one short playbook, you're able to provision and ensure the proper configuration for an Apache web server. Additionally, running the playbook with the `--check` option (see the next section below) verifies 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, either 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. The value can be set to `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 wild card 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 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 `remote_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-become-pass` (`-K`) option. You can also explicitly force all tasks in a playbook to use sudo with `--become` (`-b`. Finally, you can define the sudo user for tasks run via `sudo` (the default is root) with the `--become-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 --become --become-user=janedoe \
--ask-become-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.

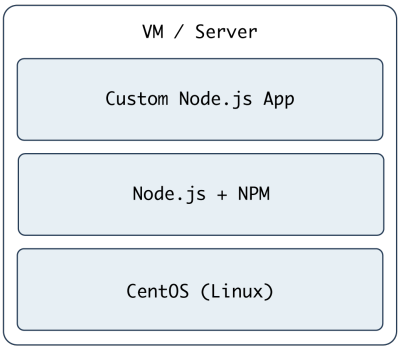
The rest of this chapter uses more realistic Ansible playbooks. All the examples in this chapter are in the Ansible for DevOps GitHub repository[51], and you can clone that repository to your computer (or browse the code online) to follow along more easily. The GitHub repository includes Vagrantfiles with each example, so you can build the servers on your local host using Vagrant.

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 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:

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



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:

```
---
- 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 7.x):

```
# Import Remi GPG key.
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 Remi repo.
rpm -Uvh --quiet \
  http://rpms.remirepo.net/enterprise/remi-release-7.rpm

# Install EPEL repo.
yum install epel-release

# 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 makes 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 works with variables other nifty Ansible features we'll discuss later:

```
9       - name: Install Remi repo.
10        yum:
11          name: "http://rpms.remirepo.net/enterprise/remi-release-7.rp\
12  m"
13          state: present
14
15      - name: Import Remi GPG key.
16        rpm_key:
17          key: "http://rpms.remirepo.net/RPM-GPG-KEY-remi"
18          state: present
19
20      - name: Install EPEL repo.
21        yum: name=epel-release state=present
22
23      - name: Ensure firewalld is stopped (since this is for testing).
24        service: name=firewalld state=stopped
25
26      - name: Install Node.js and npm.
27        yum: name=npm state=present enablerepo=epel
28
29      - name: Install Forever (to run our Node.js app).
30        npm: name=forever global=yes state=present
```

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

1. We can install extra yum repositories using the `yum` module. Just pass in the URL to the repo `.rpm` file, and Ansible will take care of the rest.
2. `rpm_key` is a very simple Ansible module that takes and imports an RPM key from a URL or file, or the key id of a key that is already present, and ensures the key is either present or absent (the `state` parameter). We're importing one key, for Remi's repository.
3. `yum` installs the EPEL repository (much simpler than the two-step process we had to follow to get Remi's repository installed!).
4. Since this server is being used only for test purposes, we disable the system firewall so it won't interfere with testing (using the `service` module).

5. `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`).
6. Since NPM is now installed, we use Ansible's `npm` module to install a Node.js utility, `forever`, to 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 simple language (JavaScript) that runs

in a lightweight environment, it's an easy language to use when testing things or prodding your server.

Since this little app is dependent on Express (an 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"
}
```

We need to copy the entire app to the server, and then have NPM download the required dependencies (in this case, `express`), so add these tasks to your playbook:

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

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

    - name: Install app dependencies defined in package.json.
      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 either using the `synchronize` or `rsync` module to copy a full directory, or `unarchive` to copy 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.

```
40      - name: Check list of running Node.js apps.
41        command: forever list
42        register: forever_list
43        changed_when: false
44
45      - name: Start example Node.js app.
46        command: "forever start {{ node_apps_location }}/app/app.js"
47        when: "forever_list.stdout.find('{{ node_apps_location }}/\
48    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.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 is *not* in the `forever list` output.

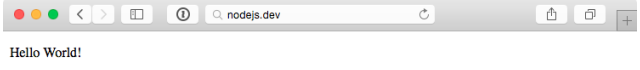
Node.js app server summary

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:



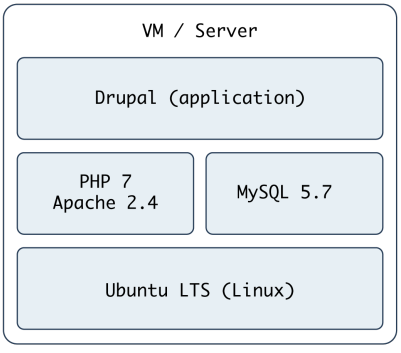
Node.js Application home page.

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

The entire example Node.js app server playbook is in this book's code repository at https://github.com/geerlingguy/ansible-for-devops[52], 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 14.04 to run a Drupal website.



Drupal LAMP server.

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

To make our playbook more efficient and readable, let's begin the playbook (named `playbook.yml`) by instructing Ansible to load in variables from a separate `vars.yml` file:

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

```
---
- hosts: all

  vars_files:
    - vars.yml
```

Using one or more included variable files cleans up your main playbook file, and lets you organize all your configurable variables in one place. At the moment, 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`:

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

Ansible lets you run tasks before or after the main set of tasks 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 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`:

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

`handlers` are special kinds of tasks you run at the end of a group of tasks by adding the `notify` option to any of the tasks in that group. The handler will only be called if one of the tasks 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 tasks.

To call this handler, 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 tasks 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 leads 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.

```
15  tasks:
16    - name: Get software for apt repository management.
17      apt: name={{ item }} state=present
18      with_items:
19        - python-apt
20        - python-pycurl
21
22    - name: "Install Apache, MySQL, PHP, and other dependencies."
23      apt: name={{ item }} state=present
24      with_items:
25        - git
26        - curl
27        - sendmail
```

```
28          - apache2
29          - php7.0-common
30          - php7.0-cli
31          - php7.0-dev
32          - php7.0-gd
33          - php7.0-curl
34          - php7.0-json
35          - php7.0-opcache
36          - php7.0-xml
37          - php7.0-mbstring
38          - php7.0-pdo
39          - php7.0-mysql
40          - php-apcu
41          - libpcre3-dev
42          - libapache2-mod-php7.0
43          - python-mysqldb
44          - mysql-server
45
46      - name: Disable the firewall (since this is for local dev only).
47        service: name=ufw state=stopped
48
49      - name: "Start Apache, MySQL, and PHP."
50        service: "name={{ item }} state=started enabled=yes"
51        with_items:
52          - apache2
53          - mysql
```

In this playbook, we begin with a 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. Install all the required packages for our LAMP server (including all the PHP extensions Drupal requires).

3. 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.
4. 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's current release. To remedy that situation, you could use the command `sudo a2enmod rewrite`, but Ansible has a handy `apache2_module` module that will do the same thing with idempotence.

We also need to add a VirtualHost entry to give Apache the site's document root and provide other options for the site.

```
58      - name: Enable Apache rewrite module (required for Drupal).
59        apache2_module: name=rewrite state=present
60        notify: restart apache
61
62      - name: Add Apache virtualhost for Drupal 8 development.
63        template:
64          src: "templates/drupal.dev.conf.j2"
65          dest: "/etc/apache2/sites-available/{{ domain }}.dev.conf"
66          owner: root
67          group: root
68          mode: 0644
69        notify: restart apache
70
71      - name: Symlink Drupal virtualhost to sites-enabled.
72        file:
73          src: "/etc/apache2/sites-available/{{ domain }}.dev.conf"
74          dest: "/etc/apache2/sites-enabled/{{ domain }}.dev.conf"
75          state: link
76        notify: restart apache
```

```
77
78      - name: Remove default virtualhost file.
79        file:
80          path: "/etc/apache2/sites-enabled/000-default.conf"
81          state: absent
82        notify: restart apache
```

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

The second command copies a Jinja2 template we define inside a `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 add them to the empty `vars.yml` file we created earlier:

```
1  ---
2  # The core version you want to use (e.g. 8.1.x, 8.2.x).
3  drupal_core_version: "8.1.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 `/var/www/drupal-8.1.x-dev` and `drupaltest` (or whatever values you'd like!).

The last two tasks (lines 71-82) 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 exist (there's no directory at `{{ drupal_core_path }}` yet!). This is why using `notify` is important—instead of adding a task 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 tasks (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:

```
84      - name: Adjust OpCache memory setting.
85        lineinfile:
86          dest: "/etc/php/7.0/apache2/conf.d/10-opcache.ini"
87          regexp: "^opcache.memory_consumption"
88          line: "opcache.memory_consumption = 96"
89          state: present
90        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 adjust PHP's default `opcache.memory_consumption` option so the Drupal codebase can be compiled into PHP's system memory for much faster page load times.

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 "opcache.memory_consumption"). 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 and user (named for the domain we specified earlier) for our Drupal installation to use.

```
92      - name: Remove the MySQL test database.
93        mysql_db: db=test state=absent
94
95      - name: Create a database for Drupal.
96        mysql_db: "db={{ domain }} state=present"
97
98      - name: Create a MySQL user for Drupal.
99        mysql_user:
100         name: "{{ domain }}"
101         password: "1234"
102         priv: "{{ domain }}.*:ALL"
103         host: localhost
104         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, and a MySQL user for Drupal.

Ansible works with many databases out of the box (MongoDB, MySQL/-MariaDB, 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 to allow Ansible to 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, and use both to install Drupal:

```
106     - name: Download Composer installer.
107       get_url:
108         url: https://getcomposer.org/installer
109         dest: /tmp/composer-installer.php
110         mode: 0755
111
112     - name: Run Composer installer.
113       command: >
114         php composer-installer.php
115         chdir=/tmp
116         creates=/usr/local/bin/composer
117
118     - name: Move Composer into globally-accessible location.
119       shell: >
120         mv /tmp/composer.phar /usr/local/bin/composer
121         creates=/usr/local/bin/composer
```

The first two commands download and run Composer's php-based installer, which generates a 'composer.phar' PHP application archive in `/tmp`. This archive is then copied (using the `mv` shell command) to the location `/usr/local/bin/composer` so we can use the `composer` command to install all of Drush's dependencies. The latter two commands are set to run only 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:

```
123     - name: Check out drush master branch.
124       git:
125         repo: https://github.com/drush-ops/drush.git
126         dest: /opt/drush
127
128     - name: Install Drush dependencies with Composer.
129       shell: >
130         /usr/local/bin/composer install
131         chdir=/opt/drush
132         creates=/opt/drush/vendor/autoload.php
133
134     - name: Create drush bin symlink.
135       file:
136         src: /opt/drush/drush
137         dest: /usr/local/bin/drush
138         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, pass in the `repo` (repository URL) and `dest` (destination path) parameters, and the `git` module will check out `master` by default.

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 finds 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, install Drupal's dependencies with Composer, run Drupal's installation via Drush, and fix a couple other file permissions issues so Drupal loads correctly within our VM.

```
140     - name: Check out Drupal Core to the Apache docroot.
141       git:
142         repo: http://git.drupal.org/project/drupal.git
143         version: "{{ drupal_core_version }}"
144         dest: "{{ drupal_core_path }}"
145
146     - name: Install Drupal dependencies with Composer.
147       shell: >
148         /usr/local/bin/composer install
149         chdir={{ drupal_core_path }}
150         creates={{ drupal_core_path }}/sites/default/settings.php
151
```

```
152     - name: Install Drupal.
153       command: >
154         drush si -y --site-name="{{ drupal_site_name }}"
155         --account-name=admin
156         --account-pass=admin
157         --db-url=mysql://{{ domain }}:1234@localhost/{{ domain }}
158         chdir={{ drupal_core_path }}
159         creates={{ drupal_core_path }}/sites/default/settings.php
160       notify: restart apache
161
162     # SEE: https://drupal.org/node/2121849#comment-8413637
163     - name: Set permissions properly on settings.php.
164       file:
165         path: "{{ drupal_core_path }}/sites/default/settings.php"
166         mode: 0744
167
168     - name: Set permissions on files directory.
169       file:
170         path: "{{ drupal_core_path }}/sites/default/files"
171         mode: 0777
172         state: directory
173         recurse: yes
```

First, we clone 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`, `8.1.x`, etc.), tag (`1.0.1`, `8.1.5`, etc.), or individual commit hash (`50a1877`, etc.) to clone.

Before installing Drupal, we must install Drupal's dependencies using Composer (just like with Drush). For both this and the next task, we only want to run them the first time we install Drupal, so we can use the creation of a `settings.php` file along with the `creates` parameter to let Ansible know if the dependencies and database have already been installed.

To install Drupal, we use 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: "Drupal Test"
```

Once the site is installed, we also restart Apache for good measure (using `notify` again, like we did when updating Apache's configuration).

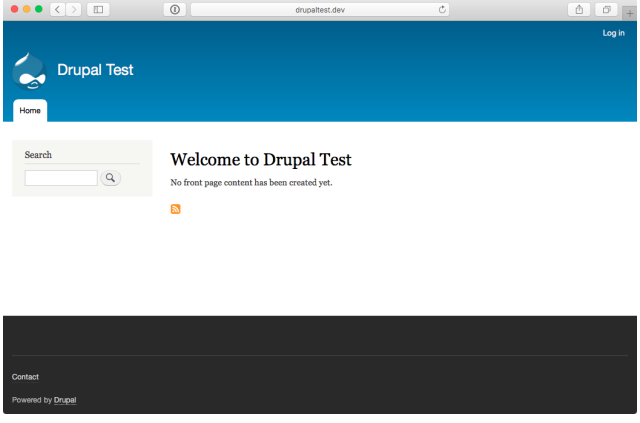
The final two tasks set permissions on Drupal's settings.php and files folder to 744 and 777, respectively.

Drupal LAMP server summary

To run the playbook on a server (either via a local VM for testing or on another server), use the following command:

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

After the playbook completes, 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!).



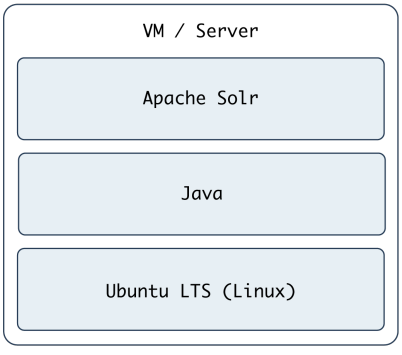
Drupal 8 default home page.

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.

The entire example Drupal LAMP server playbook is in this book's code repository at https://github.com/geerlingguy/ansible-for-devops[53], in the `drupal` directory.

Real-world playbook: Ubuntu 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 16.04 and Java.



Apache Solr Server.

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

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

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

```
---
- hosts: all

  vars_files:
  - 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:

```
download_dir: /tmp
solr_dir: /opt/solr
solr_version: 6.1.0
solr_checksum: sha1:41045799ed9b5f826b0dcab4b28b3b1986afa523
```

These variables define two paths we'll use while downloading and installing Apache Solr, and the version and file download checksum for downloading Apache Solr's source.

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:

```
  pre_tasks:
    - name: Update apt cache if needed.
      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 `solr`, a service that will be configured by the Apache Solr installation:

```
  handlers:
    - name: restart solr
      service: name=solr state=restarted
```

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

Install Java 8

It's easy enough to install Java 8 on Ubuntu, as it's in the default apt repositories. We just need to make sure the right package is installed:

```
15   tasks:
16     - name: Install Java.
17       apt: name=openjdk-8-jdk state=present
```

That was easy enough! We used the `apt` module to install `openjdk-8-jdk`.

Install Apache Solr

Ubuntu's LTS release includes a package for Apache Solr, but it installs an older version, so we'll install the latest version of Solr from source. The first step is downloading the source:

```
19     - name: Download Solr.
20       get_url:
21         url: "https://archive.apache.org/dist/lucene/solr/\
22 {{ solr_version }}/solr-{{ solr_version }}.tgz"
23         dest: "{{ download_dir }}/solr-{{ solr_version }}.tgz"
24         checksum: "{{ solr_checksum }}"
```

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 `checksum`, 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.

`checksum` compares a hash of the data in the downloaded file to a hash that you specify (and which is provided alongside the downloads on the Apache Solr website). If the checksum doesn't match the supplied hash, Ansible will fail and discard the freshly-downloaded (and invalid) file.

We need to expand the Solr archive so we can run the installer inside, and we can use the `creates` option to make this operation idempotent:

```
25      - name: Expand Solr.
26        unarchive:
27          src: "{{ download_dir }}/solr-{{ solr_version }}.tgz"
28          dest: "{{ download_dir }}"
29          copy: no
30          creates: "{{ download_dir }}/solr-{{ solr_version }}/\
31  README.txt"
```

If you read the `unarchive` module's documentation, you might notice you could consolidate both the `get_url` and `unarchive` tasks into one task by setting `src` to the file URL. Doing this saves a step in the playbook and is generally preferred, but in Apache Solr's case, the original .tgz archive must be present to complete installation, so we still need both tasks.

Now that the source is present, run the Apache Solr installation script (provided inside the Solr archive's `bin` directory) to complete Solr installation:

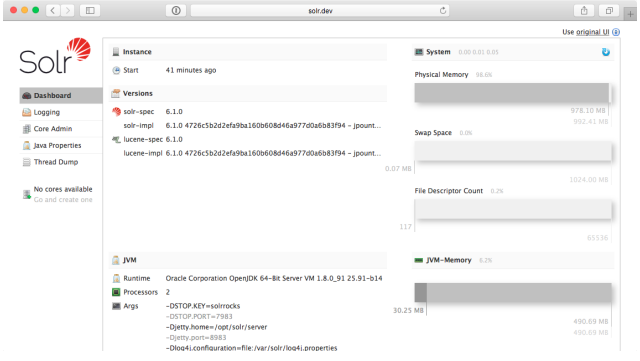
```
32      - name: Run Solr installation script.
33        shell: >
34          {{ download_dir }}/solr-{{ solr_version }}/bin/install_solr_\
35  service.sh
36          {{ download_dir }}/solr-{{ solr_version }}.tgz
37          -i /opt
38          -d /var/solr
39          -u solr
40          -s solr
41          -p 8983
42          creates={{ solr_dir }}/bin/solr
```

In this example, the options passed to the installer are hardcoded (e.g. the `-p 8983` tells Apache Solr to run on port `8983`), and this works fine, but if you're going to reuse this playbook for many different types of Solr servers, you should probably configure many of these options with variables defined in `vars.yml`. This exercise is left to the reader.

Finally, we need a task that runs at the end of the playbook to make sure Apache Solr is started, and will start at system boot:

```
43      - name: Ensure solr is started and enabled on boot.
44        service: name=solr state=started enabled=yes
```

Run the playbook with `$ ansible-playbook playbook.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://solr.dev:8983/solr (where 'solr.dev' is your server's hostname or IP address):



Solr Admin page.

Apache Solr server summary

The configuration we used when deploying Apache Solr allows for a multi core 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[54], a service I run which hosts Apache Solr search cores for Drupal websites.

The entire example Apache Solr server playbook is in this book's code repository at https://github.com/geerlingguy/ansible-for-devops[55], in the `solr` directory.

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 tasks, conditionals, variables, and more. Later, we'll explore the use of playbooks with roles to make them infinitely more flexible and to save time setting up and configuring your infrastructure.

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

[54] https://hostedapachesolr.com/

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